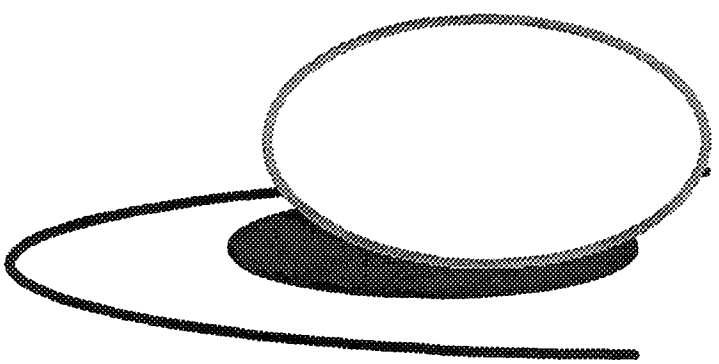
UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

ESCUELA UNIVERSITARIA DE INFORMÁTICA

GRÁFICOS 3D



Agustín Trujillo Pino

GRÁFICOS 3D.

PRIMERA EDICIÓN.

Agustín Trujillo Pino.

ISBN : 84-8499-958-0

Fotocopias y encuadernación realizadas en el Departamento de Informática y Sistemas de la Universidad de las Palmas de Gran Canaria el 10 de Enero de 1997.

ESCUELA UNIVERSITARIA DE INFORMÁTICA.

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

PRÓLOGO

En sus comienzos, el uso de los gráficos generados por ordenador tuvo como principal motivo el poder representar, de una forma más comprensible, los datos númericos. Hoy en día, y debido fundamentalmente al gran avance de la tecnología, el uso de los ordenadores se ha extendido a todas las ramas de la producción, la ciencia, el diseño, el arte y otras actividades humanas. El desarrollo que se ha producido en esta rama de la informática, alcanzado en los últimos treinta años, ha sido muy grande.

Para intentar seguir al ritmo de esta evolución, se consideró que era necesario actualizar el temario de la asignatura de Informática Gráfica, ya que cuando ésta fue concebida allá por los años 80, se centraba fundamentalmente en gráficos bidimensionales. Así, se estudiaban algoritmos para dibujar primitivas 2D, transformaciones geométricas, algoritmos de recorte 2D, relleno de áreas, etc. Al final del curso se daba una introducción a las 3D, pero sin entrar en mucho detalle.

En la actualidad sucede todo lo contrario. La mayoría de las aplicaciones existentes usan las tres dimensiones: ingeniería asistida por ordenador, simulación y animación, visualización científica, video-juegos, educación, interfaces de usuario, arte, sistemas de información geográfica, sistemas médicos, y un largo etcétera. Por lo tanto se ha decidido alterar el temario de manera que incluya todos estos conceptos y técnicas recientes, los cuales darán una visión más profunda y amplia de la Informática Gráfica a los universitarios que cursen la asignatura. Lo anterior no supone que se haya eliminado la parte 2D. Sigue siendo necesario el estudiar los fundamentos básicos bidimensionales antes de poder extenderse a las 3D teniendo una buena base.

Con este texto de apuntes se ha intentado rellenar el vacío que existía en este tema, sobre todo de cara a los estudiantes, ya que la gran mayoría de los libros de Informática Gráfica existentes están escritos en inglés, y además son un poco caros y difíciles de conseguir. De esta manera pueden disponer de una **guía** de la asignatura (de la parte de 3D), en donde se hacen referencia a todos los contenidos que se imparten pero sin profundizar demasiado en ellos. Y ése es su objetivo. Lógicamente, dicha guía deberá ser completada individualmente por cada estudiante mediante notas de clase, tutorías y consultas bibliográficas de los libros de texto.

El texto se ha dividido en seis capítulos. En el primero de ellos se estudian las **transformaciones geométricas** 3D. Como comentábamos antes, el espacio bidimensional pronto se queda corto, y se acaba por generalizar las transformaciones 3D mediante una extensión de las matrices y coordenadas homogéneas vistas en el plano. En el segundo capítulo se aborda la generación de una imagen bidimensional a partir de una **vista** en el espacio 3D. En una situación general, un objeto o conjunto de ellos formando una escena puede ser visto desde cualquier posición espacial en donde se encuentre el observador, y además éste puede estar mirando en cualquier dirección arbitraria del espacio.

También se analizan los algoritmos de recorte, para poder calcular qué partes de la escena se encuentran fuera de la vista del observador, para eliminarlas de la imagen.

Los siguientes capítulos están dedicados al **modelado geométrico**. El capítulo tres trata sobre la representación de curvas y superficies, de una forma diferente y más exacta que la simple malla tradicional de segmentos o polígonos conectados. La aproximación general consiste en usar funciones de mayor grado que las funciones lineales lo cual permite una mejor aproximación y con menor información a almacenar, lo que supone un gran ahorro del volumen de espacio de almacenamiento, y una manipulación interactiva mucho más sencilla que con las funciones lineales. En el capítulo cuarto se estudia el concepto de sólido y su importancia así como las técnicas más importantes que permiten representarlos: la representación por fronteras y la representación volumétrica.

Los últimos dos capítulos tratan el tema de la **visualización**, es decir, el proceso mediante el cual un modelo matemático en la memoria del ordenador llega a aparentar ser un objeto real en la pantalla. Este proceso se realiza en dos fases. La primera de ellas se ve en el capítulo quinto, donde se analizan las técnicas de visibilidad, que identifican aquellas partes de la escena que son visibles desde un punto de vista específico. La segunda fase consiste en establecer los criterios que permitena calcular el color de cada pixel de la imagen final, y es estudiada en el capítulo sexto.

Para la construcción de estos apuntes se han utilizado los principales libros básicos sobre la materia, y casi todos ellos se encuentran disponibles en la biblioteca del departamento (aunque no en un gran número). Todos ellos están citados al final del texto. También es necesario mencionar que la mayor parte de las ilustraciones y algoritmos pertenecen a esos mismos libros, debido sobre todo al esfuerzo que requeriría por nuestra parte crear otras similares.

Por otro lado me gustaría agradecer a las personas que me han ayudado de alguna manera a editar estos apuntes, como Domingo Martín de la Universidad de Granada, Francisco de Asís Conde de la Universidad de Jaen, y José Cortés de la Universidad de Sevilla, y a todos los que no nombraré aún debiendo hacerlo.

Agustín Trujillo Pino, 9 de Enero de 1997

Contents

1 TRANSFORMACIONES GEOMÉTRICAS EN 3-D **5**

1.1 SISTEMAS DE COORDENADAS. 5

1.2 TRANSFORMACIONES 6

Tapering 14

Twisting 15

Bending 16

1.3 COMPOSICIÓN DE TRANSFORMACIONES 17

Ejemplo de composición 19

1.4 ROTACIÓN GENERAL 23

1.5 TRANSFORMACIÓN DE SISTEMAS DE COORDENADAS 27

Ejemplo 2D 28

Caso general 2D 30

Caso General 3D 31

2 VISTAS EN 3D **33**

2.1 PROYECCIONES 33

2.2 TRANSFORMACIÓN DE VISTA 38

2.3 RECORTE 3D 47

Recorte por un volumen rectangular 52

Recorte por un volumen en perspectiva 54

3 REPRESENTACIÓN DE CURVAS Y SUPERFICIES 57

3.1 MALLAS POLIGONALES (MESHES) 57

3.2 CURVAS CÚBICAS PARAMÉTRICAS 61

3.3 SUPERFICIES BICÚBICAS PARAMÉTRICAS 78

3.4 SUPERFICIES CUÁDRICAS 88

4 REPRESENTACIÓN DE SÓLIDOS 91

4.1 MODELOS DE ALAMBRE 92

4.2 REPRESENTACIÓN POLIGONAL 93

Modelado de objetos poligonales 94

4.3 PATCHES BICÚBICOS 99

Modelado de objetos mediante patches bicúbicos 101

Superficies Ruled (Ruled Surfaces) 103

4.4 GEOMETRÍA SÓLIDA CONSTRUCTIVA (CSG) 104

4.5 TÉCNICAS DE SUBDIVISIÓN ESPACIAL 107

Subdivisión adaptativa 113

5 VISIBILIDAD 115

5.1 CONCEPTOS Y TÉCNICAS GENERALES 116

5.2 ELIMINACIÓN DE LÍNEAS OCULTAS 131

5.3 ELIMINACIÓN DE SUPERFICIES OCULTAS 139

Barrido con z-buffer 142

Procedimiento mejorado 145

6 ILUMINACIÓN 155

6.1 MODELOS DE ILUMINACIÓN 155

Reflexión lambertiana 157

Atenuación de la fuente de luz 158

Superficies y luces con color 159

El modelo de iluminación de Phong 160

Cálculo del vector de reflexión 161

Múltiples fuentes de luz 162

6.2 MODELOS DE SOMBREADO PARA POLÍGONOS 162

6.3 DETALLES DE LA SUPERFICIE 170

6.4 SOMBRAS 171

6.5 TRANSPARENCIAS 176

6.6 ALGORITMOS DE ILUMINACIÓN GLOBAL 177

6.7 RAY-TRACING 179

Chapter 1 TRANSFORMACIONES GEOMÉTRICAS EN 3-D

Es muy fácil crear objetos tridimensionales en el ordenador, simplemente dando las coordenadas (x, y, z) de sus vértices y aristas. Pero en la mayoría de los casos se hace necesario el poder alterar dichos objetos. Las aplicaciones de diseño permiten al usuario manipular y modificar las partes que componen una pieza; las animaciones por ordenador se producen moviendo la cámara o los objetos dentro de la escena. Todos estos cambios en cuanto a la orientación, forma y tamaño de los objetos es llevado a cabo por las **transformaciones geométricas**, alterando las coordenadas de los objetos. Veamos cómo podemos implementarlas.

1.1 SISTEMAS DE COORDENADAS.

Una escena tridimensional viene definida normalmente en términos de puntos, líneas y planos. Para poder especificar estas primitivas es preciso definir previamente un sistema de coordenadas sobre el cual referenciarse. La definición de un sistema de coordenadas tridimensional puede extenderse fácilmente a partir de un sistema bidimensional, (x, y), al cual se le añade una tercera dimensión, z, en una dirección perpendicular a los dos ejes anteriores.

Cualquier punto en este nuevo sistema vendrá representado lógicamente por una terna de valores, (x, y, z), donde cada coordenada representa la distancia, positiva o negativa, en la dirección del eje correspondiente, entre el punto y el origen del sistema, representado éste por el punto $(0, 0, 0)$. Así, por ejemplo, un cubo vendría representado tal y como se muestra en la figura 1.

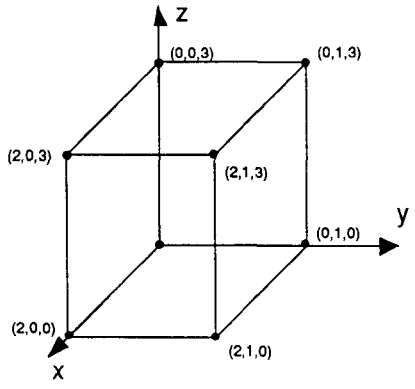


Figure 1

Al sistema de coordenadas que aparece en la figura se le conoce como sistema de mano derecha, no porque lo haya pintado un diestro, sino porque la dirección positiva del eje z viene dada por el producto vectorial $x \times y$. Intentaré explicar el por qué de esta definición con palabras, con lo fácil que es explicarlo con gestos. Si colocamos la mano derecha en forma de puño en el origen de coordenadas, apoyada sobre el plano xy, pero de tal manera que nuestro brazo apunte en la dirección positiva de x, y el eje y salga perpendicularmente hacia nuestra izquierda, al levantar el pulgar, éste indicará la dirección positiva del eje z. En un sistema de mano izquierda, la dirección de z será la contraria. Dicho de otra manera, la dirección positiva del eje z es la contraria a la del producto vectorial $x \times y$.

El hecho de permitir los dos diferentes sistemas de coordenadas es porque aún no se ha llegado a un acuerdo por parte de los autores. Unos usan el de mano derecha y otros el de mano izquierda. Esto a veces complica al estudiante cuando recopila información de distintos libros, ya que en unos la formulación matemática viene expresada al revés que en los otros. En este libro voy a decantarme por el sistema de mano derecha, en primer lugar porque creo que es el más usado, y en segundo lugar, porque me gusta más. De todas maneras, lo verdaderamente importante aquí no es cuál elegir, sino mantener el que se haya elegido durante todo el texto.

1.2 TRANSFORMACIONES

Las transformaciones en 3D son también una simple extensión de las transformaciones bidimensionales. De hecho, todas las transformaciones bidimensionales pueden describirse mediante transformaciones 3D. Lo único que hay que tener en cuenta es que vamos a trabajar con una coordenada más.

Recordemos que las transformaciones geométricas en 2D venían representadas por matrices 3x3. ¿ A qué era debido esto, si en realidad los puntos sólo tienen dos coordenadas? Las respuesta era porque íbamos a trabajar con coordenadas homogéneas. Esto es, un punto 2D, de coordenadas (x, y), en coordenadas homogéneas vendrá representado por una terna, (x, y, w), en donde w valdrá normalmente 1. En un caso general, las verdaderas coordenadas del punto serán $(x/w, y/w)$. El paso a coordenadas homogéneas era necesario porque existían transformaciones, como por ejemplo la traslación, que no podían representarse en forma matricial 2x2, pero sí es posible con matrices 3x3. De hecho, utilizando matrices 2x2 podemos usar todas las combinaciones lineales posibles del tipo $ax+by$, mientras que con matrices 3x3, el tipo de combinación es $ax + by + c$, como lo es la traslación.

Por lo tanto, en nuestro sistema 3D, los puntos vendrán representados por cuatro valores, (x, y, z, w), donde w normalmente valdrá 1, y las transformaciones 3D vendrán representadas por matrices 4x4. De este modo, trasformar un punto $P = (x, y, z, 1)$ mediante una transformación representada por la matriz M, nos dará unas nuevas coordenadas para el punto:

$$P' = P \times M$$

Una propiedad muy importante que van a cumplir estas transformaciones es que cuando creemos una trasformación compleja por medio de transformaciones simples concatenadas, ésta vendrá representada simplemente por el producto de las matrices que representan a cada una de las transformaciones, respetando siempre el orden en el que se realizaron.

Esta propiedad es explotada a fondo por las tarjetas gráficas existentes en el mercado, sobre todo para las tarjetas de estaciones gráficas, como Silicon Graphics. Imaginemos que estamos trabajando con una aplicación de modelado gráfico, en donde tenemos un objeto en pantalla perfectamente definido y ubicado en nuestro sistema 3D. El programa normalmente nos va a permitir irle aplicando diversas transformaciones al objeto: moverlo, rotarlo en torno a un eje, escalarlo, deformarlo, etc. El resultado final de todas estas transformaciones puede representarse por una única matriz, la cual es el resultado de haber multiplicado las matrices de todas las transformaciones aplicadas.

Lo que hace la máquina es tener almacenada siempre en memoria de vídeo la matriz de la transformación aplicada. Por cada transformación que queramos añadir, la matriz en memoria se multiplica por la nueva y el resultado se vuelve a almacenar en memoria. Así, representar el objeto en pantalla siempre se hace multiplicando la matriz actual por las coordenadas del objeto. Por otra parte, la multiplicación de matrices 4x4 viene ya implementada en el hardware de la tarjeta, por lo que la velocidad de proceso es altísima. Debido a esto es por lo que haremos constantemente hincapié a lo largo de este texto en la necesidad de expresar todas las transformaciones en matrices 4x4. De esta manera, nuestra aplicación optimizará al máximo los recursos de la máquina, y por otro lado, nuestros programas quedarán mucho mas concisos y generales, ya que el procedimiento de aplicar una trasnformación siempre se reduce a una multiplicación, independientemente de los valores que tenga la matriz, es decir, independientemente de la transformación concreta.

1.2.1 Traslación

La transformación de traslación desplaza las coordenadas del punto t_x unidades en la dirección x, t_y unidades en la dirección y, y t_z unidades en la dirección z. La matriz de traslación será por lo tanto:

$$T(t_x, t_y, t_z) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{pmatrix}$$

y las coordenadas homogéneas del punto transformado:

$$(x', y', z', 1) = (x, y, z, 1) \times T(t_x, t_y, t_z)$$

donde:

$$\left.\begin{array}{l} x' = x + t_x \\ y' = y + t_y \\ z' = z + t_z \end{array}\right\}$$

La transformación inversa se obtiene negando los valores de la traslación, es decir, viene dada por $T(-t_x, -t_y, -t_z)$.

1.2.2 Escalado

El escalado es la trasformación que expande o contrae el valor de una o varias de las coordenadas. Al igual que en el caso bidimensional, el escalado siempre deja un punto fijo, en este caso, el $(0, 0, 0)$. Esto significa que si queremos agrandar o empequeñecer los vértices de un objeto, este deberá estar centrado en el origen. De lo contrario, el objeto, además de cambiar de tamaño, cambiará también de posición. Por lo tanto, cuando lo queramos realizar es un cambio de tamaño de los vértices de un objeto pero sin alterar su posición, habrá que realizar primero una traslación para llevar el vértice al origen de coordenadas, luego aplicar el correspondiente escalado, y en último lugar aplicar la traslación inversa para llevarlo nuevamente a su sitio. Esta forma de trabajar (colocar el objeto en la forma idónea para aplicar la trasformación, y luego deshacer la colocación) será muy común en las demás transformaciones geométricas que vayamos viendo.

La transformación del escalado va a multiplicar la coordenada x por el factor e_x, la coordenada y por el factor e_y, y la coordenada z por el factor e_z . Lógicamente, si $e_i > 1$ la coordenada crecerá, si $e_i < 1$ la coordenada decrecerá, y si $e_i = 1$ la coordenada no variará. La matriz será por lo tanto:

$$E(e_x, e_y, e_z) = \begin{pmatrix} e_x & 0 & 0 & 0 \\ 0 & e_y & 0 & 0 \\ 0 & 0 & e_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

y las coordenadas homogéneas del punto transformado:

$$(x', y', z', 1) = (x, y, z, 1) \times E(e_x, e_y, e_z)$$

donde:

$$\left.\begin{array}{l} x = xe_x \\ y = ye_y \\ z = ze_z \end{array}\right\}$$

La transformación inversa se obtiene invirtiendo los factores de escala, es decir, viene dada por $E(1/e_x, 1/e_y, 1/e_z)$.

1.2.3 Rotación Plana

Por rotación plana vamos a entender cuando el eje de rotación es uno de los tres ejes de coordenadas. El ángulo de rotación será positivo cuando al colocarnos sobre la dirección positiva del eje mirando hacia el origen de coordenadas (véase figura 2), la rotación sea en la dirección contraria a las agujas del reloj. Así, esta rotación cumple la propiedad de que los puntos permanecen en el mismo plano perpendicular al eje de rotación en el que se encontraban.

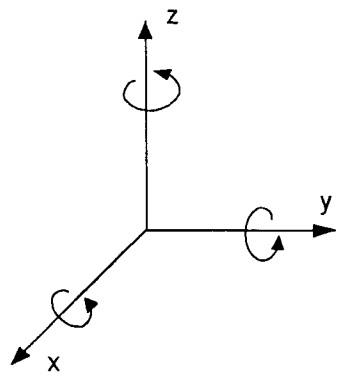


Figure 2

Una rotación bidimensional sobre el origen puede verse como una rotación del plano xy alrededor del eje z. Por lo tanto, su matriz de transformación se define a partir de la matriz 2D:

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

y las coordenadas homogéneas del punto transformado:

$$(x', y', z', 1) = (x, y, z, 1) \times R_z(\theta)$$

donde:

$$\left.\begin{array}{l} x' = x\cos\theta - y\sin\theta \\ y' = x\sin\theta + y\cos\theta \\ z' = z \end{array}\right\}$$

Puede observarse como la coordenada z de los puntos permanece invariante. La expresión para las otras dos coordenadas puede verse en el ejemplo de la figura 3.

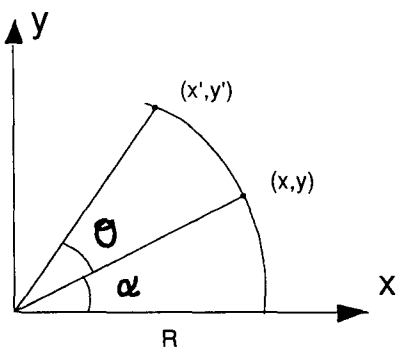


Figure 3

Supongamos que tenemos el punto P de coordenadas (x, y) y queremos rotarlo en el eje z un ángulo de θ radianes. En realidad, las coordenadas de P pueden verse como $(R\cos\alpha, R\sin\alpha)$, siendo R la distancia de P al origen, y α el argumento de P. Entonces, las coordenas del punto P' una vez transformado serán $(x', y') = (R\cos(\alpha + \theta), R\sin(\alpha + \theta))$. Desarrollando un poco esta expresión:

$$\begin{aligned} x &= R\cos\theta\cos\alpha - R\sin\theta\sin\alpha = x\cos\theta - y\sin\theta \\ y &= R\sin\theta\cos\alpha + R\cos\theta\sin\alpha = x\sin\theta + y\cos\theta \end{aligned}$$

Un detalle importante a tener en cuenta es que, como hemos dicho antes, la rotación de los puntos es siempre alrededor del origen de coordenadas. Cuando querramos rotar un punto alrededor de otro, será preciso trasladar ambos para que este último caiga en el origen, proceder con la rotación, y luego deshacer la traslación realizada. Este procedimiento de mover, transformar y deshacer el movimiento inicial va a ser muy común a lo largo del texto. Volveremos a él más adelante.

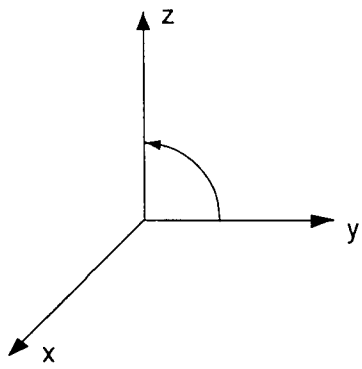


Figure 4

Para obtener las matrices de rotación en los ejes x e y se procede de igual manera. Para el caso del eje x, partiremos también de la rotación 2D, solo que esta vez consideraremos que el plano donde se rota es el yz (véase figura 4).

Así, la matriz de rotación nos queda:

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

y las coordenadas homogéneas del punto transformado:

$$(x', y', z', 1) = (x, y, z, 1) \times R_x(\theta)$$

donde:

$$\left.\begin{array}{l} x' = x \\ y' = y\cos\theta - z\sin\theta \\ z' = y\sin\theta + z\cos\theta \end{array}\right\}$$

Para el eje y hacemos lo mismo, pero esta vez, como se aprecia en la figura 5, la rotación es en el plano zx, y la dirección positiva será como siempre la contraria al reloj, es decir, de z a x.

De esta forma obtenemos la matriz de rotación:

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

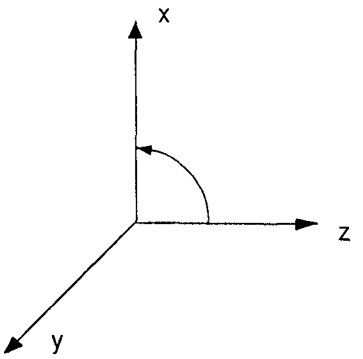


Figure 5

y las coordenadas homogéneas del punto transformado:

$$(x', y', z', 1) = (x, y, z, 1) \times R_y(\theta)$$

donde:

$$\left.\begin{array}{l} x' = x\cos\theta + z\sin\theta \\ y' = y \\ z' = -x\sin\theta + z\cos\theta \end{array}\right\}$$

Para todas ellas, la transformación inversa consiste en aplicar la misma rotación pero con un ángulo de $-\theta$ radianes. Siempre que hablemos de transformación inversa o matriz inversa, nos referiremos tanto a que la transformación es la contraria a la original, como a que matemáticamente la matriz es la inversa de la matriz que representa la transformación original.

1.2.4 Afilamiento (Shear)

El afilamiento es la transformación que arrastra todos los puntos de una recta fijada que pasa por el origen de coordenadas sobre uno de los ejes principales. Veamos primero el caso 2D que es más sencillo de visualizar.

En la figura 6 puede verse como todos los puntos sobre la recta $y = -bz$ son arrastrados hasta el eje z, haciendo su componente $y' = 0$, y dejando invariante la componente z ($z' = 0$). Los puntos con z positiva que estén a la derecha de la recta pasarán a tener $y' > 0$, y los que estén a la izquierda, $y' < 0$. En realidad el afilamiento puede interpretarse como una traslación en la que la cantidad a trasladar depende del valor de una de las coordenadas, en este caso de la z. No debe confundirse con una rotación, puesto que la componente z no varía. La expresión para el punto transformado es:

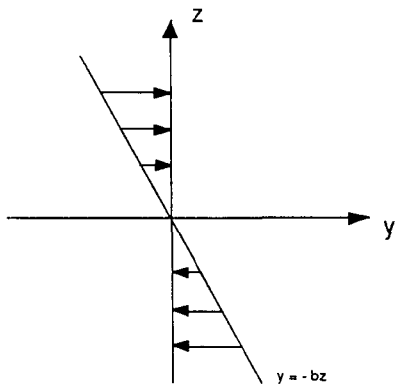


Figure 6

$$\left.\begin{array}{l} y' = y - (-bz) = y + bz \\ z' = z \end{array}\right\}$$

Del mismo modo arrastramos los puntos de la línea $z = -az$ hasta el eje z:

$$\left.\begin{array}{l} x' = x - (-az) = x + az \\ z' = z \end{array}\right\}$$

Mezclando estas dos transformaciones obtenemos el afilamiento 3D, que toma una línea arbitraria 3D que pasa por el origen de coordenadas y lo mueve al eje z, dejando fijos los valores de z. La matriz de transformación resultante nos queda:

$$A_z(a,b) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ a & b & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Esta transformación nos será de mucha utilidad en el siguiente tema, aunque ahora no lo parezca.

1.2.5 Deformaciones

Las anteriores tranformaciones que hemos visto se aplican siempre a puntos. Ahora bien, podemos considerar a un objeto como un conjunto de puntos conectados entre sí, por lo que si transformamos de igual manera cada uno de sus puntos, diremos que hemos transformado el objeto. Estas transformaciones son lineales, y lo que hacían era mover un objeto (rotación y traslación) o escalarlo, manteniendo siempre intacta la forma del objeto (excepto cuando el escalado no era uniforme en todas las componentes, lo cual producía un estrechamiento o alargamiento).

En esta sección vamos a ver otro tipo de transformaciones llamadas deformaciones, que ya no son lineales. En general, vamos a considerar las transformaciones del tipo

$$\left.\begin{aligned} x' &= F_x(x,y,z) \\ y' &= F_y(x,y,z) \\ z' &= F_z(x,y,z) \end{aligned}\right\}$$

donde ahora las funciones F pueden no ser lineales. Existen multitud de deformaciones posibles. En realidad, cualquier función que imaginemos puede representar una deformación, pero a saber cómo nos quedará el objeto! Veamos tres tipos.

Tapering

El tapering es una extensión del escalado. Se elige un eje de tapering que va a permanecer fijo, y se escalan las otras dos coordenadas dependiendo del valor de la primera. Es como si hiciéramos un escalado diferencial o progresivo. Así por ejemplo, el tapering sobre el eje z será:

$$\left.\begin{aligned} x' &= rx \\ y' &= ry \\ z' &= z \end{aligned}\right\} \text{ siendo } r = f(z) \text{ una función lineal o no dependiente del valor de } z.$$

Puede verse el resultado de aplicar esta transformación a un objeto cualquiera[1] en la figura 7.

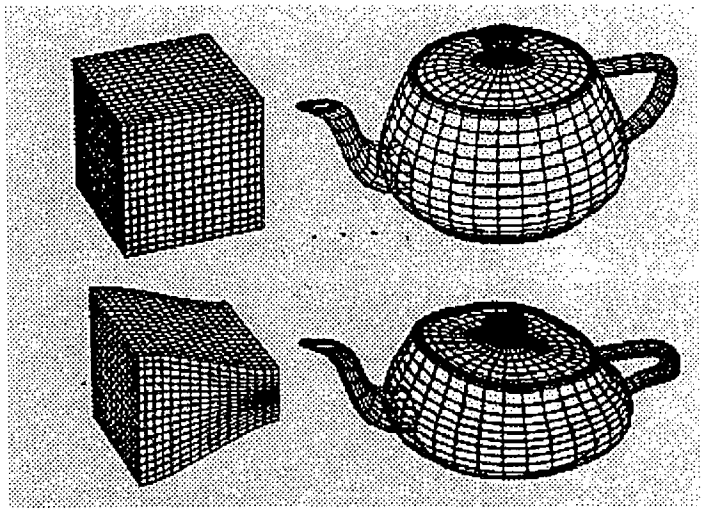


Figure 7

[1]El objeto tetera del ejemplo no es en realidad un objeto "cualquiera". Fue diseñado en la Universidad de Utah hace bastantes años, y se ha convertido prácticamente en un estandar sobre el que mostrar los efectos de todo tipo de algoritmos. No hay libro de gráficos en el que no aparezca.

No debe confundirse este tipo de deformación con otro método diferente que existe en muchas aplicaciones software de modelado gráfico, como es la *deformación de caja*. Este tipo de deformación consiste en colocar el objeto a transformar en en interior de un cubo o caja, y posteriormente ir moviendo y modificando los puntos sobre la caja. Al final, todo lo que hayamos hecho sobre la caja se verá reflejado en el objeto.

Un detalle importante en este punto es recordar la necesidad de mallar la superficie del objeto que queramos deformar. Por ejemplo, imaginemos que en el ejemplo de la figura anterior, la función $f(z)$ que controlaba el tapering no fuese lineal sino que fuese por ejemplo senoidal. Lo que obtendríamos sería una caja con un perfil senoidal. Pero si la caja no estuviese mallada, es decir, si sólo tuviésemos 8 puntos pertenecientes a la caja, correspondientes a sus 8 vértices, el perfil seguiría siendo recto, pues transformaríamos cada uno de los vértices y luego los conectaríamos con segmentos rectos, perdiendo así todo el detalle interior, que era precisamente lo que queríamos obtener. De aquí surge la necesidad de mallar.

Además, el detalle o finura de la malla va a depender también de qué grado de deformación queramos. Lógicamente, si la transformación es bastante lineal no es preciso mallar en muchos puntos, pero si la función tiene mucha variación, cuantos más puntos nuevos obtengamos mucho mejor será el resultado.

Este tipo de cuestiones nos los vamos a encontrar a lo largo de todo este texto. Recordemos que estamos siempre utilizando un ordenador y una pantalla para representar escenas y objetos del mundo real, y mientras que estos son sólidos y continuos, la pantalla está discretizada en pixels, y nuestros programas van a tratar con puntos individuales. Por lo tanto, no hay que ser muy listo para pensar que cuantos más puntos tengamos sobre el objeto, más realista será su representación en pantalla, pero más costo computacional será necesario por parte de los algoritmos. Este compromiso va a estar siempre presente en los gráficos por ordenador.

Twisting

Al igual que el tapering podía verse como un escalado diferencial, el twisting, además de ser un baile, puede verse como una rotación diferencial. Para "twistear"[2] un objeto sobre el eje z aplicamos la siguiente tranformación:

$$\left.\begin{array}{l} x' = x\cos\theta - y\sin\theta \\ y' = x\sin\theta + y\cos\theta \\ z' = z \end{array}\right\}$$ siendo $\theta = f(z)$ una función lineal o no dependiente del valor de z.

El resultado puede verse en la figura 8.

Nótese en este ejemplo que la necesidad de mallar el objeto previamente a la de-

[2]Puede ser una mala manía el conjugar verbos ingleses como si fueran españoles, pero suena igual de mal intentar encontrar un vocablo español de significado similar y luego conjugarlo.

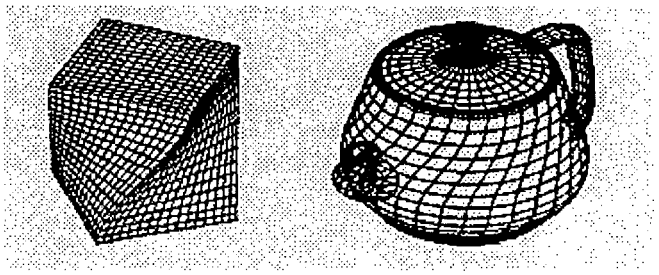


Figure 8

formación es aún más evidente en este caso que en el anterior. Aplicarle un twist a un cubo de 8 vértices y 6 caras puede resultarnos en 6 nuevas caras que no son verdaderos polígonos, es decir, sus 4 vértices no son coplanarios, lo cual nos llevará a errores de representación. Así mismo, puede observarse como la resolución del mallado debe ser proporcional a la magnitud de la deformación, como indicábamos antes.

Otro detalle que hay que mencionar es cuando queramos aplicar el twist sobre un eje arbitrario que no coincide con ninguno de los ejes de coordenadas. Los pasos a seguir serán: trasladar o rotar el objeto y el eje hasta hacer que éste coincida con un eje principal, aplicar el twist, y deshacer el movimiento inicial.

Bending

Esta última deformación que vamos a ver es un poco más complicada que las dos anteriores, por lo que no expondré sus expresiones analíticas. No quiero que se asusten ya desde el primer tema. Simplemente diré que la deformación consiste en indicar una región dentro de la cual va a aplicarse la deformación, y dentro de ella los puntos son rotados y trasladados de acuerdo a dichas expresiones. El resultado puede verse en la figura 9.

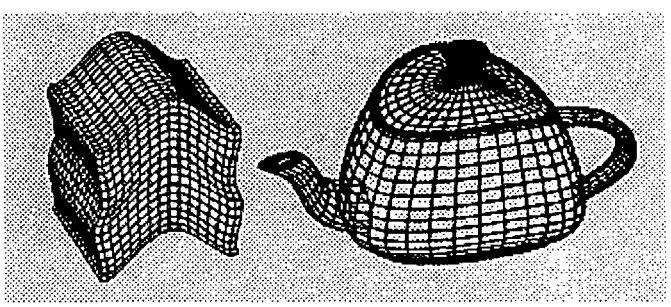


Figure 9

Combinando diferentes deformaciones se facilita mucho la labor de modelado. Por ejemplo, si quisieramos modelar una caracola como la de la figura 10 en base a ir

definiendo puntos y caras sería una tarea muy complicada. Sin embargo, para obtener la figura se ha partido de un cilindro, al que se le ha aplicado un twisting, y posteriormente un tapering. El resultado es inmediato.

Figure 10

1.3 COMPOSICIÓN DE TRANSFORMACIONES

Las rotaciones, escalados y afilamientos son transformaciones lineales, ya que los nuevos puntos se calculan a partir de combinaciones lineales de las componentes anteriores

$$P' = ax + by + cz$$

Las deformaciones por tanto no lo son. Sin embargo la traslación sí podía expresarse de esta manera. El problema era que incluía un término independiente en la expresión. Por ello pasábamos a coordenadas homogéneas, ya que de esta forma la traslación también se convertía en una combinación lineal

$$P' = ax + by + cz + d$$

Nótese que las deformaciones no son transformaciones lineales, aunque las hayamos representado en forma matricial.

Vamos a definir **transformación afín** a una combinación de transformaciones lineales aplicadas a uno o varios puntos (a un objeto). Cada transformación vendrá representada por una única matriz, la cual se obtiene multiplicando las matrices de cada una de las transformaciones, y en el mismo orden en el que queremos que se apliquen. Es decir, si un punto P es trasladado 2 unidades en x, y posteriormente escalado al doble su componente y, obtenemos:

$$P' = P \times T(2,0,0); \quad P'' = P' \times E(1,2,1);$$

donde P' es el punto trasladado solamente, y P'' el punto final. Puede observarse como este punto final puede obtenerse directamente a partir del punto inicial de la siguiente manera:

$$P'' = P' \times E(2,1,1) = P \times T(2,0,0) \times E(1,2,1) = P \times M$$

donde M es el producto de ambas transformaciones. De hecho ya hemos comentado que esto es lo que hacen la mayoría de las tarjetas gráficas: guardar siempre la matriz última M que representa la combinación total de transformaciones que se han aplicado sobre un objeto, y visualizar dicho objeto en pantalla simplemente multiplicando las coordenadas de sus puntos por la matriz.

Las transformaciones afines cumplen la propiedad de preservar el paralelismo de las líneas, pero no así sus ángulos y longitudes, como puede verse en el ejemplo de la figura 11.

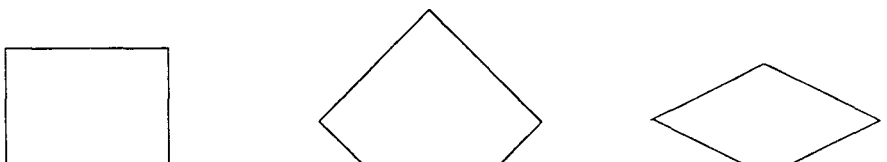


Figure 11

En este punto vuelvo a reiterar que todas las tranformaciones son siempre aplicadas a las coordenadas de los puntos individualmente. Para transformar segmentos de líneas, simplemente transformaremos sus dos extremos, y el resultado lo volveremos a unir mediante un nuevo segmento recto. Para transformar por lo tanto un objeto compuesto de vértices y aristas, transformaremos como hemos dicho cada una de sus aristas para obtener el objeto transformado.

Cuando sea necesario transformar un plano, del que sólo conocemos su ecuación, y no sus aristas como en el caso anterior, lo ideal sería obtener la ecuación del plano transformado, ya que conociendo su ecuación, dicho plano está perfectamente definido para todos sus puntos. Veamos el procedimiento.

Sea $ax+by+cz+d = 0$ la ecuación de dicho plano. Esta ecuación puede expresarse en forma matricial, definiendo un vector $N = (a,b,c,d)$, con lo cual la ecuación queda $N \times P^T = 0$, siendo $P = (x,y,z,1)$, el cual puede representar las coordenadas homogéneas de cualquier punto perteneciente al plano. Además, por definición, el vector normal al plano es $n = (a,b,c)$.

Sea M la transformación afín que se le quiere aplicar al plano. Por lo tanto, cualquier

punto P perteneciente al plano se verá transformado en un punto $P' = P \times M$. Evidentemente, lo que no se puede hacer, y que es lo que hacen todos los estudiantes en el examen, es multiplicar directamente los coeficientes del plano por la matriz M, lo cual en todo caso nos daría los nuevos valores transformados de un supuesto punto de coordenadas (a, b, c) que ni siquiera pertenece al plano.

En realidad, sabiendo que los tres primeros coeficientes del plano representan las componentes de su vector normal, lo que habría que hacer es encontrar el nuevo vector normal del plano transformado. Es decir, encontrar la matriz Q tal que $N' = N \times Q$. Nótese como M no es necesariamente igual a Q, como puede verse en ejemplo de la figura 12, donde tenemos un plano P cuya normal viene dada por el vector N, y al que queremos aplicarle un escalado. El plano resultante es P'. Sin embargo, si le hubiésemos aplicado la matriz de la transformación correspondiente al vector N nos hubiera salido el vector N' que, como vemos, no se corresponde con la verdadera normal de P'.

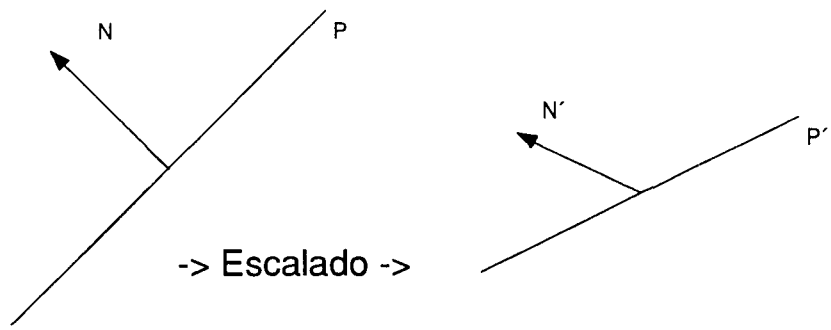


Figure 12

El objetivo se convierte por tanto en encontrar la matriz Q a partir de M. La ecuación del nuevo plano transformado será $N' \times (P')^T = 0$. Por lo tanto, desarrollando un poco:

$$N' \times (P')^T = 0 \Rightarrow N \times Q \times (P \times M)^T = N \times Q \times M^T \times P^T = 0$$

Esto nos lleva a que $Q \times M^T$ debe ser multiplo de la matriz identidad, ya que $N \times P^T = 0$. Por lo tanto, obtenemos finamente que

$$\boxed{Q = (M^T)^{-1}}$$

Ejemplo de composición

Veamos a continuación un ejemplo en el que se muestra como transformar un objeto para llevarlo a donde queramos, en base a irle aplicando sucesivas transformaciones. Sea el objeto de la figura 13 formado por los segmentos P_1P_2 y P_1P_3, el cual queremos

llevarlo desde su posición original A hasta la posición final indicada B. Lo que nos piden es la matriz de la trasformación afín que me lleva de una posición a la otra. El proceso normal para atacar este tipo de problemas en dividirlo en subproblemas más simples.

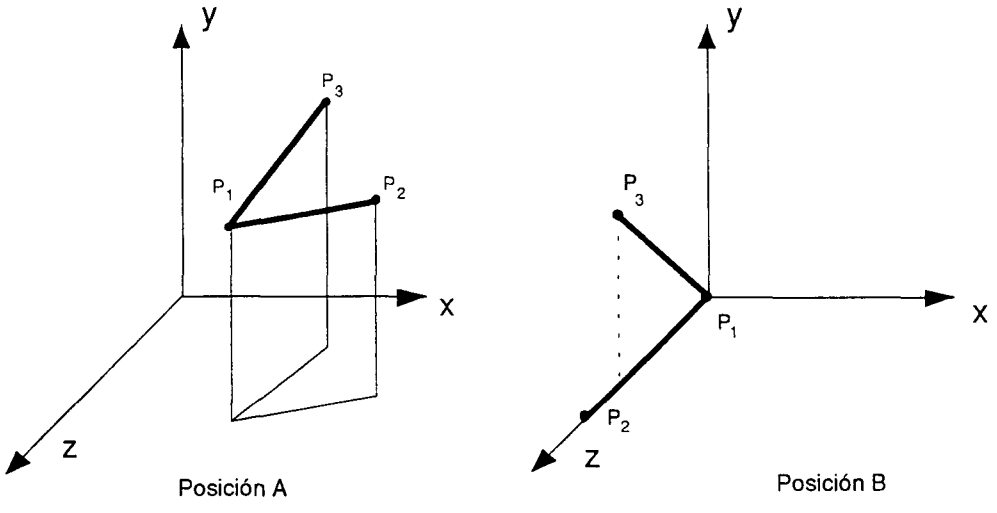


Figure 13

a) En primer lugar, vamos a trasladar $P_1 = (x_1, y_1, z_1)$ al origen de coordenadas. De esta forma, podemos aplicarle rotaciones al objeto y sabremos que P_1 permanecerá fijo. La matriz necesaria va a ser:

$$T(-x_1, -y_1, -z_1) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -x_1 & -y_1 & -z_1 & 1 \end{pmatrix}$$

Aplicando esta matriz a los tres puntos obtendremos:

$$P_1' = P_1 \times T(-x_1, -y_1, -z_1) = (0, 0, 0, 1)$$
$$P_2' = P_2 \times T(-x_1, -y_1, -z_1) = (x_2 - x_1, y_2 - y_1, z_2 - z_1, 1)$$
$$P_3' = P_3 \times T(-x_1, -y_1, -z_1) = (x_3 - x_1, y_3 - y_1, z_3 - z_1, 1)$$

b) A continuación vamos a rotar sobre el eje y hasta llevar el segmento $P_1'P_2'$ sobre el plano yz (figura 14).

Como la rotación debe ser negativa, ya que viajamos hacia la derecha, vamos a considerar que el ángulo total es $\theta - \pi/2$, y así θ será positivo. Es solamente una cuestión de sintaxis. Sin embargo, para rellenar los valores de la matriz de rotación no

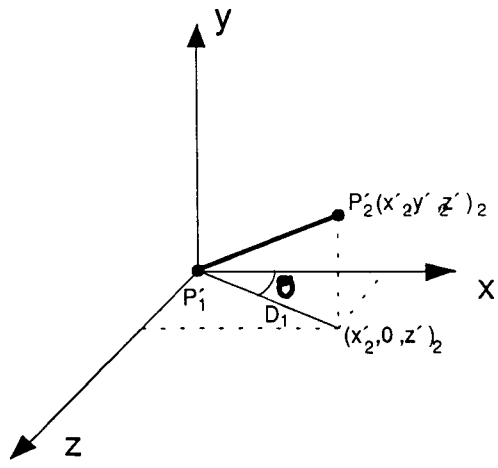


Figure 14

necesitamos tanto el valor del ángulo θ en concreto, sino más bien el de su seno y su coseno. Estos valores pueden calcularse a partir de los datos del apartado anterior:

$$\cos(\theta - \pi/2) = \sin\theta = \frac{z_2'}{D_1}$$
$$\sin(\theta - \pi/2) = -\cos\theta = -\frac{x_2'}{D_1}$$

siendo $D_1 = \sqrt{(z_2')^2 + (x_2')^2}$. De esta forma, la matriz de rotación puede ponerse como

$$R_y(\theta - \pi/2) = \begin{pmatrix} \frac{z_2'}{D_1} & 0 & \frac{x_2'}{D_1} & 0 \\ 0 & 1 & 0 & 0 \\ -\frac{x_2'}{D_1} & 0 & \frac{z_2'}{D_1} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Tras esta transformación, los puntos quedan como sigue:

$$P_1'' = P_1' \times R_y(\theta - \pi/2) = (0,0,0,1)$$
$$P_2'' = P_2' \times R_y(\theta - \pi/2) = (0, y_2', D_1, 1)$$
$$P_3'' = P_3' \times R_y(\theta - \pi/2) = ... = (x_3'', y_3'', z_3'', 1)$$

Fíjense como ya el punto P_2'' tiene su componente x igual a 0, señal de que está sobre el plano yz. El punto P_1'' permanece en el origen.

c) Lo siguiente que vamos a hacer será rotar sobre el eje x para así hacer coincidir el segmento $P_1''P_2''$ con el eje z (figura 15).

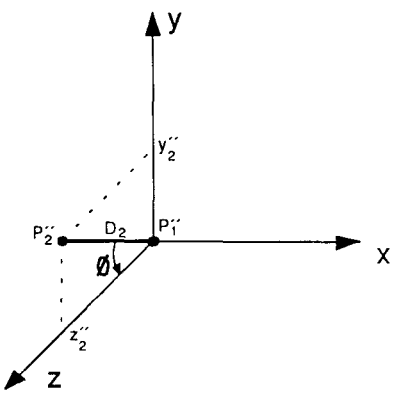


Figure 15

Al igual que antes, no estamos interesados en el valor en sí del ángulo ϕ sino de su seno y coseno:

$$\cos\phi = \frac{z_2''}{D_2}; \quad \sin\phi = \frac{y_2''}{D_2}$$

siendo $D_2 = |P_1P_2| = \sqrt{(x_2')^2 + (y_2')^2 + (z_2')^2}$. La matriz de rotación queda como sigue:

$$R_x(\phi) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{z_2''}{D_2} & \frac{y_2''}{D_2} & 0 \\ 0 & -\frac{y_2''}{D_2} & \frac{z_2''}{D_2} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

y los puntos al final de este paso son:

$$\begin{aligned} P_1''' &= P_1'' \times R_x(\phi) = (0,0,0,1) \\ P_2''' &= P_2'' \times R_x(\phi) = (0,0,D_2,1) \\ P_3''' &= P_3'' \times R_x(\phi) = \ldots = (x_3''', y_3''', z_3''', 1) \end{aligned}$$

Se ve como P_1 continúa inalterable, y P_2 ya está sobre la recta z.

d) El último paso será, lógicamente, rotar sobre z hasta llevar P_3''' sobre el plano yz. Dense cuenta que, tras el paso anterior, el segmento $P_1'''P_3'''$ puede estar en cualquier orientación en el espacio (figura 16).

El coseno y seno del ángulo que necesitamos van a ser:

$$\cos\alpha = \frac{y_3'''}{D_3}; \quad \sin\alpha = \frac{x_3'''}{D_3}$$

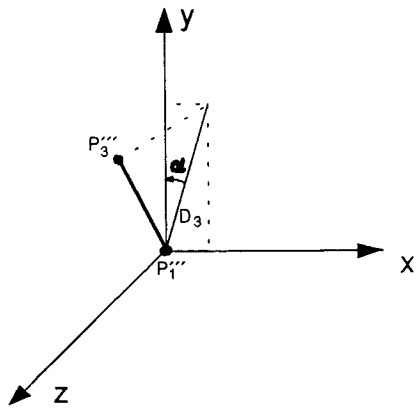


Figure 16

siendo $D_3 = \sqrt{(x_3''')^2 + (y_3''')^2}$. La matriz de rotación queda como sigue:

$$R_z(\alpha) = \begin{pmatrix} \frac{y_3'''}{D_3} & \frac{x_3'''}{D_3} & 0 & 0 \\ -\frac{x_3'''}{D_3} & \frac{y_3'''}{D_3} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

y los puntos al final de este paso son:

$$\begin{aligned} P_1'''' &= P_1''' \times R_z(\alpha) = (0,0,0,1) \\ P_2'''' &= P_2''' \times R_z(\alpha) = (0,0,D_2,1) \\ P_3'''' &= P_3''' \times R_z(\phi) = ... = (x_3'''', y_3'''', z_3'''', 1) \end{aligned}$$

Por lo tanto, la transformación afín resultante que habrá que aplicar será la combinación de estas cuatro transformaciones que hemos visto. Así, el resultado final es:

$$\boxed{M = T(-x_1, -y_1, -z_1) \times R_y(\theta - \pi/2) \times R_x(\phi) \times R_z(\alpha)}$$

1.4 ROTACIÓN GENERAL

Las rotaciones vistas anteriormente son conocidas como rotaciones planas, debido a que la rotación siempre se produce en uno de los planos principales. Es decir, el eje de rotación coincide con uno de los ejes de coordenadas. Sin embargo, en un caso general esto no siempre va a ser así. Imaginemos que tenemos modelado en el ordenador un muñeco, y queremos que éste gire su muñeca derecha. Obviamente, su antebrazo puede estar orientado en cualquier posición del espacio 3D, y el eje de rotación, que va a coincidir con su antebrazo, vendrá dado por una recta totalmente arbitraria, que puede que ni siquiera pase por el origen de coordenadas.

Por lo tanto, el problema de esta sección va a consistir en encontrar la matriz de la transformación que rota un cuerpo θ radianes sobre un eje arbitrario. Como dato inicial debemos conocer la expresión para dicha recta. Supongamos que ésta nos viene dada en base a dos puntos, P_1 y P_2. Podemos formar la ecuación de la recta en paramétricas fácilmente de la siguiente manera:

$$\begin{cases} x = (x_2 - x_1)t + x_1 = at + x_1 \\ y = (y_2 - y_1)t + y_1 = bt + y_1 \\ z = (z_2 - z_1)t + z_1 = ct + z_1 \end{cases}$$

donde el vector $v = (a, b, c)$ indica la dirección de la recta, como se aprecia en la figura 17.

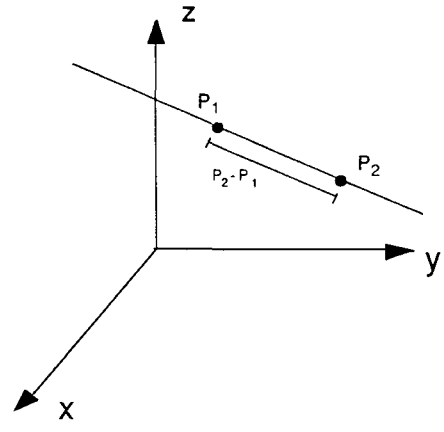


Figure 17

Procederemos igual que en el ejemplo de la sección anterior, dividiendo el problema en subproblemas. El objetivo será modificar el objeto y el eje hasta que coincidir que éste último coincida con uno de los ejes principales, aplicar la rotación plana, y deshacer posteriormente todo las transformacione previas para devolver ambos a su ubicación original.

a) Trasladar el punto P_1 al origen de coordenadas, para lo cual usaremos la matriz:

$$T(-x_1, -y_1, -z_1) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -x_1 & -y_1 & -z_1 & 1 \end{pmatrix}$$

Una vez hecha la transformación, los nuevos puntos son:

$$P_1' = P_1 \times T(-x_1, -y_1, -z_1) = (0, 0, 0, 1)$$
$$P_2' = P_2 \times T(-x_1, -y_1, -z_1) = (a, b, c, 1)$$

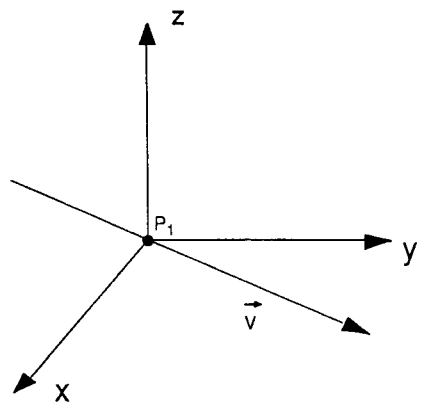


Figure 18

y el resultado es el mostrado en la figura 18.

b) Rotar en el eje x hasta que la recta se coloque sobre el plano zx. Para calcular el ángulo ϕ necesario consideraremos la proyección sobre el plano yz (figura 19).

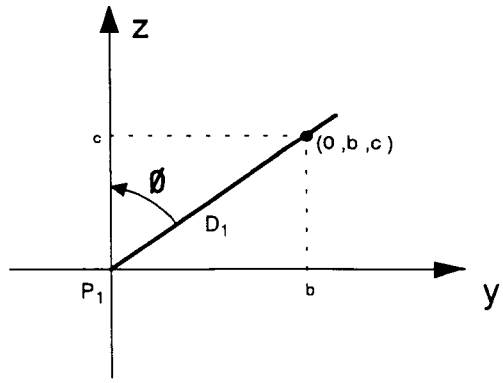


Figure 19

De aquí podemos obtener los valores para el seno y coseno, y así calcular la matriz de rotación:

$$\cos\phi = \frac{c}{D_1}; \quad \sin\phi = \frac{b}{D_1}$$

siendo $D_1 = \sqrt{b^2 + c^2}$. La matriz resultante nos queda:

$$R_x(\phi) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{c}{D_1} & \frac{b}{D_1} & 0 \\ 0 & -\frac{b}{D_1} & \frac{c}{D_1} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Los puntos quedan así:

$$P_1'' = P_1' \times R_x(\phi) = (0, 0, 0, 1)$$
$$P_2'' = P_2' \times R_x(\phi) = (a, 0, D_1, 1)$$

c) Rotar en el eje y hasta que la recta coincida con el eje z. Al igual que antes, consideremos su proyección sobre el plano zx para calcular el ángulo α necesario (figura 20).

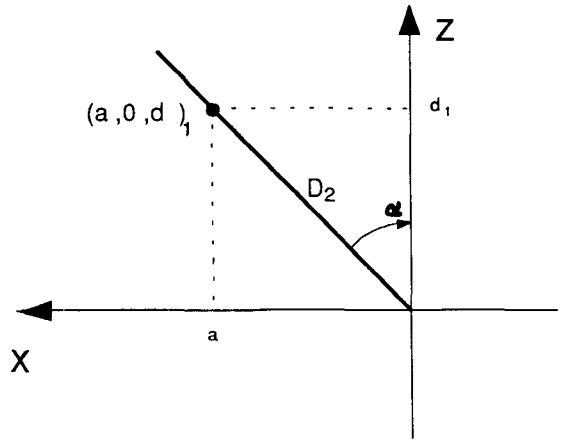


Figure 20

Deducimos a continuación los valores trigonométricos necesarios:

$$\cos(-\alpha) = \cos\alpha = \frac{D_1}{D_2}; \quad \sin(-\alpha) = -\sin\alpha = -\frac{a}{D_2}$$

siendo $D_2 = \sqrt{a^2 + D_1^2} = \sqrt{a^2 + b^2 + c^2}$. Fíjense que la rotación es negativa pues he considerado que la recta estaba situada a la izquierda del eje z. La matriz resultante es:

$$R_y(-\alpha) = \begin{pmatrix} \frac{D_1}{D_2} & 0 & \frac{a}{D_2} & 0 \\ 0 & 1 & 0 & 0 \\ -\frac{a}{D_2} & 0 & \frac{D_1}{D_2} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Los puntos quedan:

$$P_1''' = P_1'' \times R_y(-\alpha) = (0, 0, 0, 1)$$
$$P_2''' = P_2'' \times R_y(-\alpha) = (0, 0, D_2, 1)$$

d) Llegados a este punto, la recta de rotación coincide con el eje z. Además, hay que recordar que estas transformaciones se han ido aplicando no sólo a la recta sino

también al objeto que queremos rotar. De esta manera, la posición relativa del objeto con la recta permanece invariante, y es precisamente este hecho lo que provoca que cuando ahora rotemos sobre el eje z, y luego deshagamos los cambios, el resultado será como si realmente hubiéramos girado el objeto en torno a la recta (figura 21).

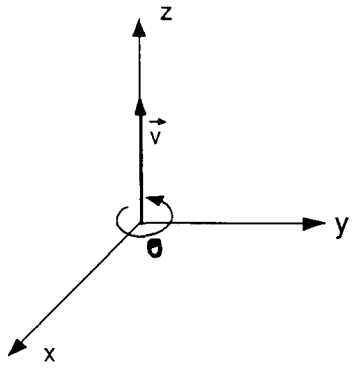


Figure 21

Para rotar en el eje z, simplemente hay que multiplicar por $R_z(\theta)$.

e) Hacer la rotación inversa del apartado c), multiplicando por $R_y(\alpha)$.

f) Hacer la rotación inversa del apartado b), multiplicando por $R_x(-\phi)$.

g) Hacer la traslación inversa del apartado a), multiplicando por $T(x_1, y_1, z_1)$.

Finalmente, la matriz de transformación resultante para la rotación general será el resultado de multiplicar las siete matrices anteriores y en el mismo orden, es decir:

$$M = T(-x_1, -y_1, -z_1) \times R_x(\phi) \times R_y(-\alpha) \times R_z(\theta) \times R_y(\alpha) \times R_x(-\phi) \times T(x_1, y_1, z_1)$$

1.5 TRANSFORMACIÓN DE SISTEMAS DE COORDENADAS

Normalmente, los objetos que se han definido en un sistema de coordenadas local, pueden luego tener que expresarse en términos de otro sistema de coordenadas diferente. Por ejemplo, si estamos diseñando una casa, puede que ya tengamos el mobiliario diseñado de antemano (sillas, mesas, etc.). Lo normal es que cada uno de esos objetos haya sido definido en función de un sistema de coordenadas diferente a cada uno, situado en en centro de cada objeto, o en una esquina, etc. Cuando queramos visualizar la casa entera con su mobiliario colocado, los valores de los puntos de cada objeto ya no serán los mismos, pues ahora estarán todos referidos a un único sistema de coordenadas global. Es decir, en una escena, todos los valores de los puntos deben estar en términos de un único sistema de coordenadas, y precisamente dichos valores son los que indican

a cuántas unidades de distancia en la dirección correspondiente se encuentra el punto alejado del origen de coordenadas.

Las tranformaciones vistas hasta ahora se aplican a los puntos, variando los valores de sus coordenadas, mientras que el sistema de coordenadas permanecía fijo. A veces ocurrirá el caso contrario. Por ejemplo, veremos en el siguiente tema que para mostrar una escena en el ordenador necesitamos definir siempre una cámara (virtual, naturalmente), la cual indica donde está situado el punto de vista. Imaginemos que el sistema de coordenadas global de la escena está ubicado en el centro de la cámara. Esto implica que cuando ésta se mueva, aunque el objeto permanezca fijo, el movimiento de la cámara provoca que los valores de los puntos del objeto varíen.

Estos dos casos anteriores demuestran la necesidad de encontrar un procedimiento para dado un nuevo sistema de coordenadas, encontrar los nuevos valores de las componentes de un punto para que pase a estar referenciado al nuevo sistema. Además, como siempre, lo ideal será que dicho procedimiento pueda ponerse en forma matricial, para que así dicha matriz pueda interpretarse como una transformación más. Ya indicamos anteriormente la utilidad de colocarlo todo en forma matricial (que no es siempre para fastidiar, como piensan algunos).

Ejemplo 2D

Veámoslo primero en el caso bidimensional, donde es más fácil deducirlo, y luego lo extenderemos al caso 3D. En la figura 22, tenemos el punto P de coordendas $(8,4)$ con respecto al sistema de referencia xy, y queremos saber qué coordenadas tendrá con respecto al nuevo sistema $x'y'$, cuya diferencia con el sistema original es que está rotado un ángulo de 45 grados.

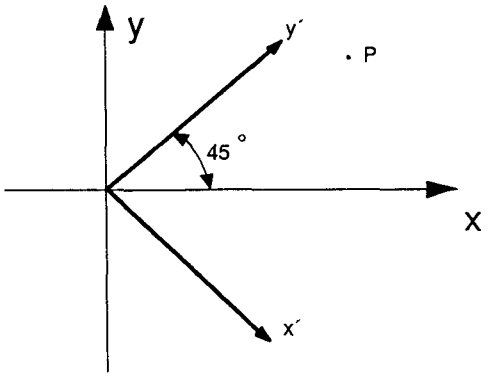


Figure 22

Como hemos dicho, la transformación que deberíamos aplicar al sistema $x'y'$ para llevarlo al xy es una rotación de $+45$ grados. ¿Qué ocurriría si aplicamos esta misma

transformación al punto P? Si lo hacemos, visualmente lo que va a ocurrir es que el punto $P' = P \times R(45)$ se va a colocar en una posición relativa al sistema xy que va ser totalmente idéntica a la que tenía el punto P original con respecto al sistema $x'y'$. Por lo tanto, las coordenadas de P', que van a estar referenciadas al sistema xy, van a tener idénticos valores que las componentes de P pero referenciadas al sistema $x'y'$.

Por lo tanto, la matriz que me cambia del sistema de referencia xy al nuevo $x'y'$ es la misma que necesitaríamos para llevar el $x'y'$ al xy. En este caso será:

$$R(\pi/4) = \begin{pmatrix} \cos(\pi/4) & \sin(\pi/4) \\ -\sin(\pi/4) & \cos(\pi/4) \end{pmatrix} = \begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ -1/\sqrt{2} & 1/\sqrt{2} \end{pmatrix}$$

y el punto P' valdrá:

$$P' = P \times R(\pi/4) = (2\sqrt{2}, 6\sqrt{2})$$

Modifiquemos un poco el ejemplo. Ahora el sistema $x'y'$, aparte de estar girado con respecto al xy, está también trasladado a unidades en la dirección x, y b unidades en la dirección y (figura 23). Es decir, el origen de coordenadas del sistema $x'y'$ está en la posición (a, b) con respecto al sistema xy. ¿Cuál será la nueva matriz de cambio de sistema de referencia?

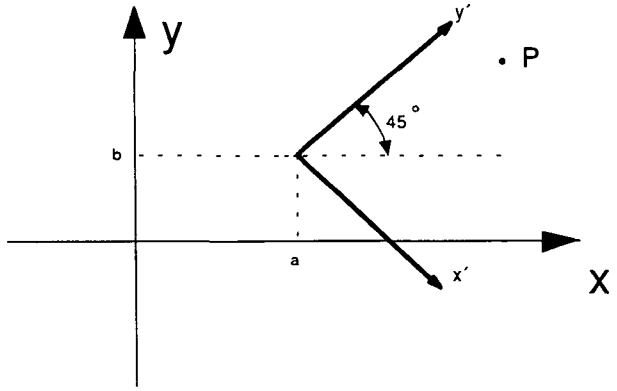


Figure 23

Como hemos deducido antes, la matriz será la misma que me lleva del sistema $x'y'$ al xy. Necesitaré una traslación y una rotación para conseguirlo. Además, un detalle importante: **trasladar antes de rotar.** ¿Por qué? Es bien fácil. Si rotas primero, el sistema $x'y'$ efectivamente se coloca en la misma orientación que el sistema xy, pero el problema es que el origen del $x'y'$ ya no va a estar en la posición (a, b), sino que también se habrá movido hacia la izquierda. Si ahora haces la traslación, los orígenes de ambos sistemas no van a coincidir. En adelante recuerda esta propiedad.

Por lo tanto, la transformación a aplicar al punto P será:

$$P' = P \times T(-a, -b) \times R(\pi/4) = P \times M$$

siendo M la matriz resultante para el cambio de sistema de referencia.

Càso general 2D

A la hora de enfrentarnos ante un caso general de transformación de sistemas de referencia, la situación normal con que nos vamos a encontrar es la de no poder calcular con tanta facilidad como en el ejemplo anterior cuál es el ángulo de rotación entre ambos sistemas, y por lo tanto, la matriz de rotación necesaria habrá que intentar buscarla por otras vías.

Supongamos el caso más general, en donde el nuevo sistema de coordenadas $x'y'$ viene especificado por la posición de su origen O' y por las direcciones de sus dos ejes (u, v), con respecto al sistema xy por supuesto. Supongamos también que los vectores u y v son unitarios, esto es, de longitud 1. En caso negativo los normalizaremos[3].

En teoría, para construir la matriz de rotación, los valores que nos interesan son no tanto el ángulo en sí a rotar, sino más bien el valor de su seno y coseno. Si recordamos la expresión de la matriz de rotación, ésta es:

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

Veamos como podemos rellenar dichos valores. Imaginemos el ejemplo de la figura 24, en donde ambos sistemas coinciden en sus orígenes pero no en las direcciones de los ejes.

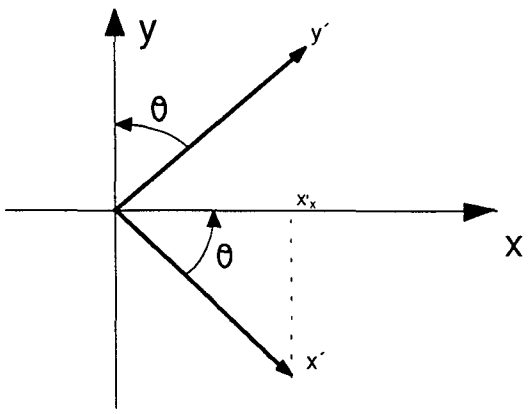


Figure 24

[3]Sea el vector $v = (a, b)$ cuya longitud viene dada por $l_v = \sqrt{a^2 + b^2}$. Si $l_v \neq 1$, el vector no es unitario. Para normalizarlo, se divide cada componente por su longitud: $u = (\frac{a}{l_v}, \frac{b}{l_v})$. Ahora, u representa la misma dirección que v, y su longitud vale $l_u = \sqrt{\frac{a^2}{l_v^2} + \frac{b^2}{l_v^2}} = \frac{\sqrt{a^2+b^2}}{l_v} = 1$

El vector x' o u tendrá por componentes $(l_u \cos\theta, -l_u \sin\theta)$, y el vector y' o v será $(l_v \sin\theta, l_v \cos\theta)$, siendo θ el ángulo existente entre x' y x, que es el mismo que existe entre y' e y. Pero lo importante aquí no es el valor del ángulo θ, que en realidad no lo conocemos. Lo importante es que, sabiendo que los vectores del sistema $x'y'$ están normalizados, esto es, $l_u = l_v = 1$, el vector x' va a coincidir con los dos primeros elementos de la primera columna de la matriz de rotación, y que el vector y' va a coincidir con los dos primeros elementos de la segunda columna de la matriz de rotación. Por lo tanto, construir la matriz de rotación necesaria para la transformación es crear la matriz:

$$R = \begin{pmatrix} u_x & v_x & 0 \\ u_y & v_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Así, en el ejemplo anterior, los ejes de coordenadas del sistema $x'y'$ venían dados por los vectores $u = (1/\sqrt{2}, -1/\sqrt{2})$, y $v = (1/\sqrt{2}, 1/\sqrt{2})$, con lo cual se forma directamente la matriz de rotación necesaria.

Caso General 3D

En el caso general tridimensional, vamos a tener el sistema de coordenadas $\{x', y', z'\}$ definido en términos de las coordenadas $\{x, y, z\}$ por medio de los vectores unitarios perpendiculares siguientes:

$$\left.\begin{array}{l} u = (u_x, u_y, u_z) \\ v = (v_x, v_y, v_z) \\ w = (w_x, w_y, w_z) \end{array}\right\}$$

Para crear la matriz de rotación es tan fácil como construir la siguiente matriz:

$$R = \begin{pmatrix} u_x & v_x & w_x & 0 \\ u_y & v_y & w_y & 0 \\ u_z & v_z & w_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Supongamos ahora que los orígenes de ambos sistemas no coinciden, por lo que necesitamos además una matriz de traslación previa. Esta matriz también se calcula de forma directa, pues los valores a trasladar son simplementes los valores negados de las componentes del origen del sistema $\{x'y'z'\}$, o sea el punto $O' = (x_0, y_0, z_0)$. Así, la matriz de cambio de sistema nos queda:

$$M = T(-x_0, -y_0, -z_0) \times R$$

Pero vayamos a una situación aún más general. Supongamos que los ejes (u, v, w) del nuevo sistema de referencia no son unitarios, y que precisamente esto no sea un hecho intrascendente, sino que sirva para indicar que el nuevo sistema está a una

escala diferente que el original. Es decir, puede que las unidades del sistema original representen metros, y la del nuevo centímetros. En ese caso, siguiendo la filosofía de que para formar la matriz de cambio de sistema se procede igual que si construyéramos la matriz que me lleva del sistema viejo al nuevo, habrá que construir una matriz de escalado e incluirla en la construcción de M.

Supongamos la situación en la que ambos sistemas coinciden en sus orígenes y en la dirección de sus ejes, pero no así en la longitud de ellos. Sea l_u la longitud del vector u medida por supuesto en unidades del sistema $\{x, y, z\}$. Por lo tanto, para pasar del sistema nuevo al viejo será preciso dividir la componente u_x del eje (que será la única distinta de cero) por la longitud del vector u. Se procede igual con los otros dos ejes. Esto es lo mismo que decir que debe aplicarse una matriz de escalado como la siguiente:

$$E\left(\frac{1}{l_u}, \frac{1}{l_v}, \frac{1}{l_w}\right) = \begin{pmatrix} 1/l_u & 0 & 0 & 0 \\ 0 & 1/l_v & 0 & 0 \\ 0 & 0 & 1/l_w & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

De esta manera, en la situación general, el orden para construir la matriz de cambio de sistema será:

a) realizar una traslación para hacer coincidir ambos orígenes,

b) realizar una rotación para hacer coincider los ejes de coordenadas, construyendo la matriz a partir de los valores de las componentes de los vectores (u, v, w), y

c) realizar un escalado para hacer coincidir las longitudes de los ejes.

La matriz resultante final de la transformación queda:

$$\boxed{M = T(-x_0, -y_0, -z_0) \times R \times E(1/l_u, 1/l_v, 1/l_w)}$$

Obsérvese que la matriz de escalado siempre debe ir al final, ya que de lo contrario, si escalamos al principio, aparte de obtener las longitudes correctas, el origen del sistema nuevo se ve desplazado, con lo cual ya luego no van a poder coincidir ambos orígenes.

Chapter 2 VISTAS EN 3D

La pantalla del ordenador va a ser siempre nuestro dispositivo de salida final. Cualquier objeto o escena tridimensional que tengamos modelado, por muy complejo que sea, siempre quedará reducido a una simple imagen que aparacerá en la pantalla, la cual dependerá de dónde se encuentra situado el supuesto observador de dicha escena, y en qué dirección la está mirando. El cálculo de dicha imagen va a ser el objeto de estudio de este capítulo.

2.1 PROYECCIONES

Una pantalla puede ser considerada como un plano bidimensional donde se va a plasmar la imagen de la escena, al igual que sobre el negativo de una cámara fotográfica queda impresa la foto de la escena que se está viendo. Los puntos de dicha imagen tendrán por lo tanto 2 coordenadas, frente a las tres componentes que tiene cada punto de nuestra escena 3D. Vamos a llamar **proyección** a la transformación que realiza la conversión de una representación 3D a una en 2D. Este proceso no es exclusivo de la informática gráfica. Los artistas usan una proyección perspectiva para crear cuadros realistas; los arquitectos usan varias proyecciones paralelas para representar diferentes vistas de un edificio; los cartógrafos usan proyecciones cónicas para dibujar mapas del mundo. Éstos son ejemplos de los diferentes tipos de proyecciones y sus usos.

La proyección de un objeto 3D viene definida por rayos protectados que emanan desde un centro de proyección, pasan a través de cada punto del objeto, e intersectan un plano de proyección donde se forma la imagen. La figura 1 muestra dos proyecciones diferentes de la misma línea. Afortunadamente, la proyección de una línea sigue siendo una línea, por lo que sólo es necesario proyectar sus dos extremos para obtener la línea proyectada.

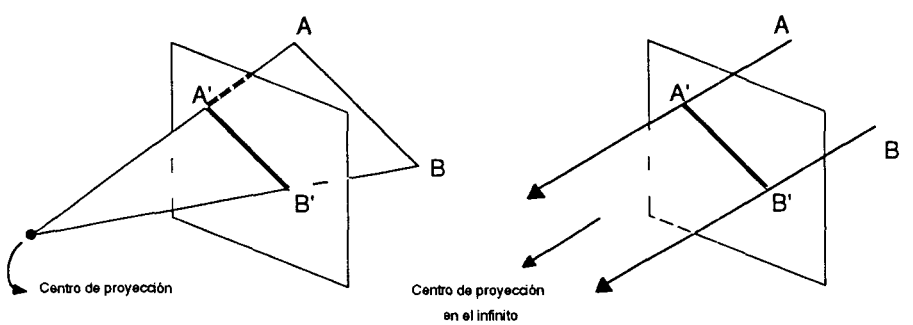


Figure 1

La clase de proyecciones más usada, y que usaremos en este texto, es la proyección geométrica plana. El término "plano" es debido a que la proyección es sobre un plano en vez de sobre una superficie curva, y a que los rayos usados son rectos. Existen otro tipo de proyecciones diferentes, como lo es por ejemplo la proyección empleada por el sistema *OmniMax,* usado en las salas donde la pantalla es una cúpula, en donde la superficie de proyección es el interior de una semiesfera. O como la proyección usada para realizar los mapamundi en los atlas que comentábamos antes, en donde se utilizan rayos de proyección curvos.

Las proyecciones geométricas planas pueden dividirse en dos clases básicas: **perspectiva** y **paralela**. La diferencia estriba en la relación del centro de proyección con el plano de proyección. Si la distancia desde uno al otro es finita, entonces se trata de una proyección perspectiva. Si por el contrario la distancia es infinita, se tratará entonces de una proyección paralela. La figura anterior muestras estos dos casos. La proyección paralela recibe su nombre debido al hecho de que al estar el centro de proyección en el infinito, todos los rayos de proyección son paralelos. Cuando se define una proyección perspectiva, hay que especificar su centro de proyección; para una proyección paralela, hay que dar una dirección de proyección.

2.1.1 Proyección Paralela

El método más simple de proyectar una escena tridimensional en una imagen bidimensional es descartar una de las coordenadas. Es decir, por ejemplo, considerar como plano de proyección el plano xy y como dirección de proyección la del eje z. Este tipo de proyección recibe el nombre de proyección paralela **ortográfica**, y el proceso de cálculo de la imagen es tan sencillo como eliminar la componente z de todos los puntos de la escena y quedarnos solamente con los valores (x, y). En la figura 2 puede verse un ejemplo de la proyección de un cubo.

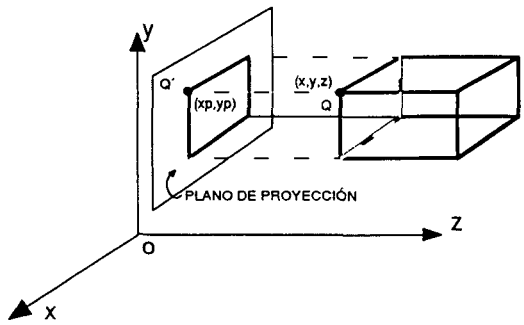


Figure 2

La mayor desventaja de la proyección paralela es la pérdida de la información sobre

la profundidad de los objetos. Un observador que viese la proyección de un cubo tal y como se aprecia en la figura anterior vería un simple cuadrado y no tendría idea del objeto 3D original. Esto quita realismo a la imagen obtenida. Sin embargo tiene una serie de ventajas, como es el hecho de saber que las líneas paralelas en la escena permanecen paralelas en la imagen, y que los ángulos se mantienen en aquellas superficies paralelas al plano de proyección.

2.1.2 Proyección Perspectiva

El efecto visual de una proyección perspectiva es similar al que se produce en los sistemas fotográficos y en el sistema visual humano: el tamaño de la proyección de un objeto varía inversamente con la distancia del objeto al centro de proyección. Es decir, los objetos cercanos al observador lucen más grandes que aquellos que estén más al fondo. Así, aunque la proyección perspectiva de un objeto tiende a parecer más realista, ésta no es muy útil a la hora de adquirir la forma exacta y las medidas de las longitudes del objeto. Las distancias ya no son las mismas que en 3D, los ángulos son falsos, e incluso las líneas paralelas de la escena dejan de serlo en la imagen.

En esta proyección, los rayos conectan el ojo (centro de proyección) con cada punto del objeto 3D. El punto de intersección del rayo con el plano de proyección nos da el punto proyectado, tal y como se muestra en la figura. Al contrario de lo que ocurre en la proyección paralela, los rayos no son paralelos sino que convergen todos en el ojo.

Veamos a continuación cómo podemos encontrar la expresión analítica para la proyección perspectiva, la cual puede interpretarse como una transformación más como las que vimos en el capítulo anterior, ya que transforma las componentes de los puntos en otros nuevos valores. La ventaja de esta interpretación es que si se expresa en forma matricial conseguimos acelerar todo el proceso de cálculo, tal y como vimos con el resto de transformaciones.

Antes de comenzar, hay que recordar que el sistema de referencia que vamos a usar para el desarrollo va a ser de mano izquierda. La razón es porque la mayor parte de los trabajos hechos sobre proyecciones asumen que la componente z mide la profundidad de los puntos, es decir, los valores de la componente z crecen a medida que nos alejamos del ojo (visualmente es como si eje z saliera de nuestro ojo y atravesase la pantalla). Además, la situación normal en un plano, como lo es la pantalla, es la de que el eje x representa la horizontal y crezca de izquierda a derecha, y que el eje y mida la vertical o altura, y crezca de abajo arriba. Por lo tanto, combinando estos tres ejes, obtenemos un sistema de mano izquierda.

Supongamos que el plano de proyección es paralelo al plano xy, colocado a una distancia D del ojo, el cual está colocado a su vez en el origen de coordenadas, y queremos calcular la proyección del cubo que aparece en la figura 3.

Sea $Q = (x_1, y_1, z_1)$ uno de los vértices del cubo, el cual se proyecta sobre el punto

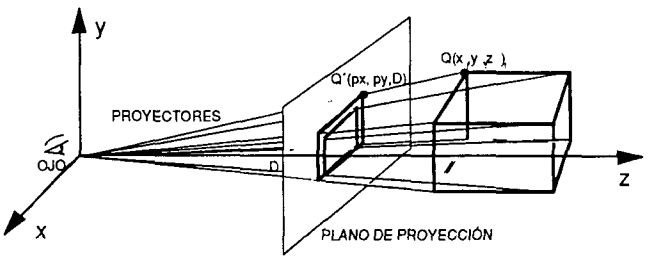


Figure 3

$Q' = (p_x, p_y, D)$, cuyas componentes deseamos conocer. Puede verse como dicho punto es la intersección del rayo de proyección que une el ojo con el punto Q, con el plano de proyección. Nótese que la tercera coordenada ya la conocemos, pues coincidirá con la distancia del plano de proyección al origen, ya que todos los puntos proyectados van a encontrarse en dicho plano. Una forma sencilla para calcular las otras dos componentes es usando las leyes de los triángulos semejantes. En la figura 4 aparece una visión desde arriba (desde el eje y), desde donde observamos el plano zx, y otra vista desde un lateral (desde el eje x), donde se ve el plano yz.

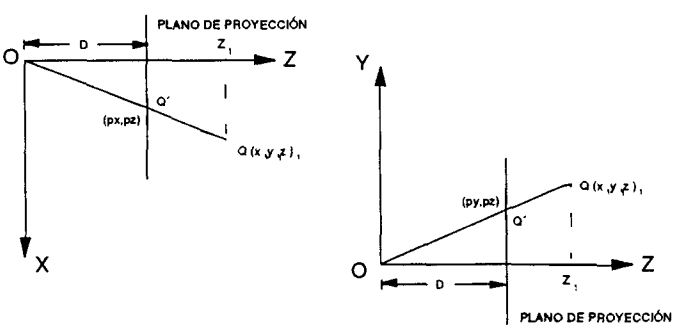


Figure 4

De aquí obtenemos:

$$\frac{p_x}{D} = \frac{x_1}{z_1} \Rightarrow p_x = D\frac{x_1}{z_1}$$

$$\frac{p_y}{D} = \frac{y_1}{z_1} \Rightarrow p_y = D\frac{y_1}{z_1}$$

Ya que el valor de z representa la distancia de un punto al ojo, la profundidad del punto está incluida en estas expresiones. Así, los objetos que estén bastante alejados del ojo tendrán componentes pequeñas debido al valor grande de la z del denominador, y aparecerán pequeñitos en la imagen.

Un objeto puede tener cualquier valor de la componente z. Esto incluye casos donde el punto está por detrás del plano de proyección o incluso detrás del ojo. Matemáticamente no hay ningún problema, pues la expresión nos da los resultados correctos, salvo que z fuese cero, es decir, que el punto estuviese en el mismo ojo. Ahí sí tendríamos problemas (conjuntivitis, ...). Sin embargo, la interpretación va a depender totalmente de nuestra aplicación. A simple vista parece que lo más logico será mostrar exclusivamente los puntos que quedan por delante del plano de proyección y considerar invisibles los que esten por detrás, aunque existen aplicaciones que a lo mejor si desean mostrar también estos puntos.

¿Qué ocurre si movemos el plano de proyección, es decir, si variamos D? Si el plano se aleja del ojo, los objetos aparecerán mayores en la imagen, pero si consideramos que dicho plano es finito (que será lo normal), la totalidad de la escena que ve el observador se ve reducida. Si por el contrario acercamos el plano al ojo, ahora los objetos serán más pequeños, pero se incluirá una vista más amplia. Es el proceso idéntico que cuando jugamos con el zoom de una cámara.

Ahora que hemos obtenido las expresiones para los puntos proyectados, lo siguiente será expresarlas en forma matricial, como si de una transformación más se tratara. La matriz que realiza la proyección perspectiva P será:

$$P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1/D \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

De esta manera, para obtener las coordenadas del punto Q' sólo hay que hacer:

$$Q' = Q \times P = \left(x_1, y_1, z_1, \frac{z_1}{D}\right) = \left(D\frac{x_1}{z_1}, D\frac{y_1}{z_1}, D, 1\right)$$

Como dicho punto tiene la componente homogénea diferente a 1, las coordenadas reales del punto Q' vendrán dadas por el resultado de dividir las otras tres componentes por la homogénea. Podría parecer una bobería el seguir usando cuatro componentes para el punto Q', ya que con mantener los dos primeros valores ya tendríamos localizado al punto. Sin embargo, la ventaja de seguir trabando con puntos de 4 componentes, aparte de dar mayor uniformidad a todo el sistema, es que no perdemos la información del punto original Q, lo cual sería imposible si sólo usásemos dos o tres componentes. Es decir, en cualquier momento podemos deshacer el cambio, ya que sabemos que las tres primeras componentes indican la posición de Q.

Ya hemos indicado que las líneas rectas al proyectarse siguen siendo rectas en la imagen. Eso nos daba la ventaja de que para proyectar una escena compuesta por objetos definidos por aristas, sólo necesitábamos proyectar los extremos de dichas líneas, y posteriormente conectar dichos puntos proyectados con nuevas líneas rectas. En la figura 5 puede verse el efecto de la proyección perspectiva de un cuadrado. Obsérvese también como a pesar de que el objeto original era un cuadrado, la perspectiva deforma los ángulos y lo que se ve en la imagen es un trapecio. Sin embargo, es precisamente este efecto el que provoca que el sistema visual humano interprete la profundidad.

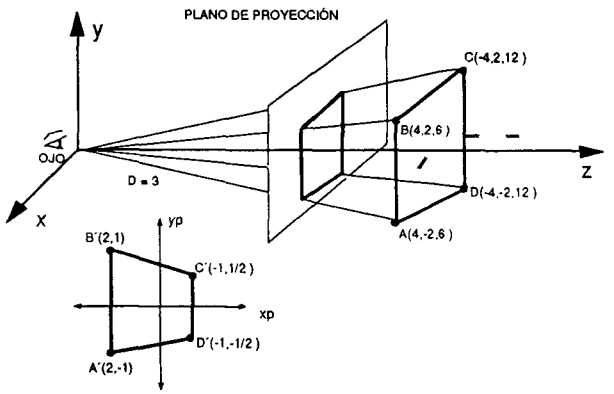


Figure 5

2.2 TRANSFORMACIÓN DE VISTA

En la sección previa veíamos cómo proyectar una escena 3D sobre un plano. Para hacerlo, asumíamos que el ojo estaba en el origen de coordenadas de un sistema de mano izquierda, y que el plano de proyección era paralelo al plano xy. Desafortunadamente, esta situación no es la más general en el mundo real. En un caso general, el plano de proyección puede estar en cualquier situación del espacio 3D, y el ojo no tiene por qué estar fijo en el origen de coordenadas.

Por ejemplo, imaginemos que estamos desarrollando un simulador de vuelo o un video-juego en donde tenemos perfectamente modelado una escena completa que abarca una zona donde tenemos montañás, árboles, casas, coches, hormigas gigantes, etc. Todo este sistema estará en función de un único sistema de referencia 3D, al que llamaremos sistema global, el cual será de mano derecha. El observador de nuestra aplicación estará lógicamente sentado a los mandos del avión, el cual sobrevuela la escena. El cristal de la cabina será nuestro plano de proyección, el cual puede estar en cualquier orientación (sobre todo cuando ya sepamos hacer loopings), los ojos del piloto pueden estar en cualquier punto del interior de la cabina (suponiendo que no vale sacar la cabeza fuera), y además éste puede dirigir la mirada en cualquier dirección. Esto sí que

es una situación general, y como vemos, es mucho más compleja que la que veíamos en la sección anterior. Esta situación puede verse en la figura 6.

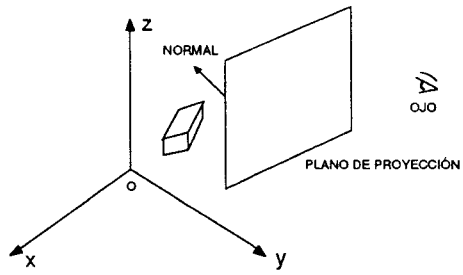


Figure 6

Cada uno de los objetos de la escena vendrán definidos en función de sus propios sistemas de referencia locales. Es la situación normal. Por ejemplo, para crear un pequeño bosque de árboles, haré uso de un objeto árbol individual con sus coordenadas locales, y lo colocaré en su posición correcta en función del sistema 3D global de la escena. Así habrá que proceder con todos los objetos de la escena, colocándolos en el sistema de referencia global mediante las apropiadas transformaciones geométricas. Una vez hecho esto, vamos a llamar **transformación de vista** al proceso de transformar las coordenadas de los puntos de la escena, los cuales se hallaban referenciados al sistema de mano derecha global de la escena, a un nuevo sistema de coordenadas de mano izquierda cuyo origen está en el ojo, y cuyo eje z indica la dirección de vista. En realidad, no es más que un cambio de sistema de referencia como los que veíamos en el capítulo anterior. El resultado será que ahora la componente z indica la profundidad del objeto en relación con el observador, lo cual coincide con la situación de partida de la sección previa.

Finalmente, y como espero que ya se imaginarán, la última etapa de la transformación consistirá en realizar la proyección (perspectiva normalmente) tal y como ya se ha visto, para convertir los puntos desde el sistema de mano izquierda del ojo hasta un sistema de coordenadas bidimensionales situado en el plano de proyección. Todo este proceso es análogo a producir una vista o foto de la escena tal y como sería vista por el supuesto obsevador a traves de una ventana rectangular situada en el plano de proyección.

2.2.1 Parámetros de vista

Veamos a continuación todo los datos previos que necesitamos conocer para poder crear la foto de la escena. En primer lugar se encuentra el **plano de proyección**, el cual vendrá definido por un punto sobre el plano y una normal (vector perpendicular) al plano. Este punto, al que llamaremos **punto de referencia de vista**, VRP (view

reference point) va a jugar un papel fundamental en la transformación de vista. Los otros parámetros para la vista vendrán definidos con coordenadas relativas a este punto. La **normal al plano**, VPN (view plane normal) indica la orientación del plano de proyección. Junto con el punto de referencia el plano de proyección está perfectamente determinado, como puede verse en la figura 7.

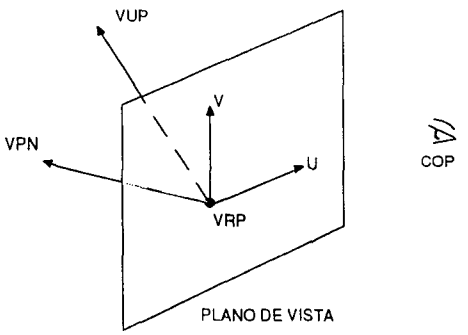


Figure 7

La utilidad de la normal al plano es la de definir un nuevo sistema de coordenadas de mano izquierda. Para hacer esto, necesitamos otro vector perpendicular al VPN y que descanse sobre el plano. El **vector de verticalidad** VUP (view up) va a definir la dirección de lo que el observador va a considerar como su vertical, y no podrá ser paralelo a la normal del plano. Este vector es muy importante, por lo que hay que entederlo muy bien. Recordemos que el objetivo de esta transformación era el transformar las coordenadas de los puntos de la escena a un sistema de coordenadas cuyo origen estaba en el ojo, y cuyo eje z indicaba la dirección de vista. Lógicamente, el vector normal VPN hará de eje z, pero necesitamos saber cuál va a ser la nueva dirección y. En el ejemplo del piloto, esta cuestión es similar a decir: conocemos cuál es la dirección de vista, pues viene dada por la normal al plano (el cristal de la cabina), pero ¿cuál es el vector que mide la vertical? Fíjense que el avión puede estar en cualquier orientación, ¡hasta incluso boca abajo! En ese caso, el eje y del nuevo sistema coincidirá con el eje $-y$ del sistema global de la escena.

Por lo tanto, el vector VUP simplemente va a indicar la verticalidad del observador. Además, puede que de hecho el vector VUP no descanse sobre el plano, es decir, no sea inicialmente perpendicular a la normal VPN, aunque eso no es problema, ya que con lo que vamos a trabajar va a ser con su proyección sobre el plano. Este vector proyectado es el que va a definir el eje y del sistema centrado en el ojo, y a la vez, la dirección v del plano de proyección, como se ve en la figura anterior. El tercer eje, el eje u del sistema bidimensional y x del sistema 3D del ojo, vendrá dado simplemente por el producto vectorial de VPN por V. Este eje también descansará sobre el plano, y será perpendicular a VPN y a V.

El punto de referencia de vista, VRP, va a ser el origen del sistema bidimensional

$\{u, v\}$, pero no va a ser el origen del sistema 3D que vamos a definir, el cual va a estar centrado en el ojo o **centro de proyección**, COP (center of projection). Las coordenadas del ojo vendrán definidas relativas al VRP, en unidades del sistema global. En la figura anterior se aprecian estos nuevos parámetros. También puede verse como efectivamente el vector VUP puede interpretarse como la dirección vertical de la cabeza del observador.

Llegados a este punto ya disponemos de todos los parámetros necesarios para construir la transformación de perspectiva proyectando la escena sobre el plano de proyección. Sin embargo, en la mayoría de los casos, el observador no va a poder visualizar la escena completa. Una ventana rectangular ubicada en el plano recortará parte de la imagen. Es lógico, ya que el campo de visión humano no es infinito, y la pantalla donde vamos a mostrar la foto también es finita. Esta ventana bidimensional tendrá sus bordes paralelos a los ejes u y v, y su tamaño y su posición centro del plano vendrá definida en términos de distancia al VRP (ver figura 8). Esta ventana junto con el ojo, COP, van a definir un **volumen de vista** con forma de pirámide rectangular, el cual va a incluir la parte de la escena 3D que el observado puede ver. Es decir, todos aquellos objetos que se encuentre fuera de este volumen pueden ser eliminados de antemano, ya que con total seguridad no van a aparecer en la imagen. La línea que une el COP con el centro de la ventana especifica la **dirección de vista**.

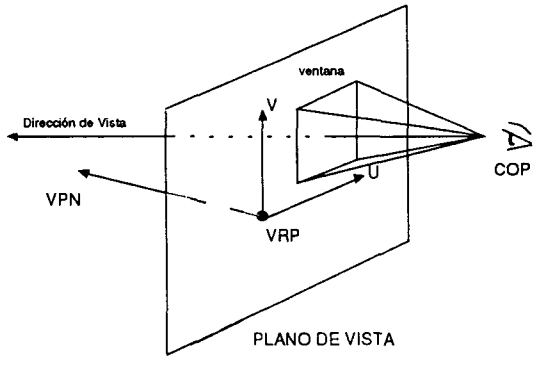


Figure 8

Si existe algún objeto en el interior del volumen de vista que esté muy cercano al ojo, éste puede bloquear todo el resto de la escena, lo que quizás no nos interese. Similarmente, un objeto que esté muy lejano para el cual sea necesario un gran coste computacional para poder calcular su proyección, puede que en la imagen final sólo ocupe unos pocos pixels y no merezca la pena. Por todo ello, a veces es muy útil eliminar los objetos muy cercanos o muy lejanos al COP. Esto se realiza especificando dos planos de recorte adicionales, uno frontal y otro trasero, ambos paralelos al plano de proyección, como se aprecia en la figura 9. El volumen de vista es ahora finito, y tiene forma de pirámide rectangular truncada.

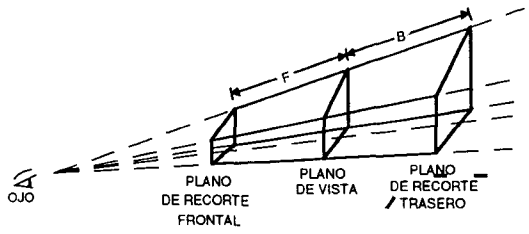


Figure 9

2.2.2 Construcción de la matriz de vista

Suponiendo que todo lo anterior esté perfectamente claro (y si no volver a la sección 2 y leerlo de nuevo), ya estamos preparados para construir la transformación de vista, la cual nos va a transformar la escena descrita en el sistema de coordenadas global de la escena a un nuevo sistema de mano izquierda centrado en el ojo, cuyo eje y indica la vertical del observador, y cuyo eje z indica la dirección de vista. En realidad esta transformación no es más que una de las etapas necesarias para poder mostrar en pantalla la imagen de una escena sintética. Todo este proceso de creación de la foto no es más que un procedimiento secuencial compuesto por 5 etapas:

Etapa 1. Modelado. Se posicionan todos los objetos para crear la escena referenciada a un único sistema de coordenadas 3D al que llamamos sistema global.

Etapa 2. Transformación de Vista. Se transforma la escena a un nuevo sistema de mano izquierda centrado en el ojo, cuyo eje z indica la dirección de vista, y cuyos ejes x e y indican la horizontal y vertical respectivamente.

Etapa 3. Recorte. A partir del volumen de vista se eliminan todos los objetos que caigan fuera.

Etapa 4. Proyección. Se realiza la proyección perspectiva para obtener una imagen realista de la escena.

Etapa 5. Display. Se transforma la imagen obtenida en la ventana del plano de proyección a un área determinada de la pantalla del ordenador.

Pero sigamos con la transformación de vista. El primer paso será obtener los cuatro parámetros de vista necesarios: el punto de referencia de vista VRP, la normal al plano VPN, el vector de verticalidad VUP, y el centro de proyección COP. Además habrá que conocer la ventana sobre el plano donde se va a proyectar la imagen. Supongamos que ésta nos viene dada mediante las coordenadas de sus esquinas inferior izquierda y superior derecha, esto es (u_{min}, v_{min}) y (u_{max}, v_{max}), tal y como se ve en la figura 10. Recordemos que estas coordenadas vienen definidas relativas al VRP. Otro dato que necesitamos saber también, caso de que se especifique, es la situación de los planos

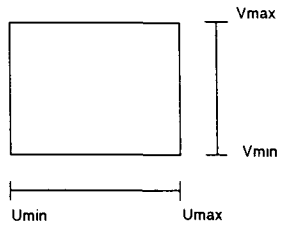


Figure 10

de recorte frontal y trasero. El plano trasero vendrá definido por un valor B, el cual indicará una distancia positiva desde el plano de proyección. El plano frontal vendrá definido por un valor F, el cual podrá ser positivo o negativo, según esté situado por delante o por detrás del plano de proyección (véase la figura 11)

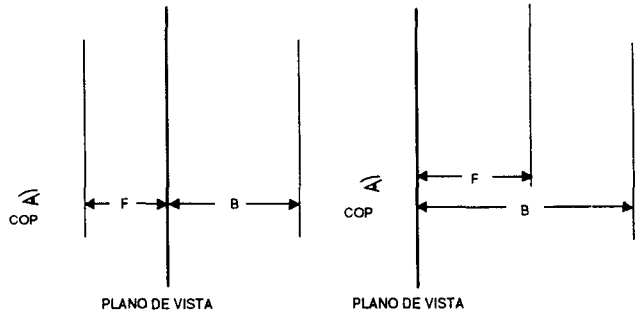


Figure 11

El siguiente paso es construir el sistema bidimensional $\{u, v\}$. El vector v no es más que la proyección del vector VUP sobre el plano, el cual puede calcularse como

$$\overrightarrow{v} = \overrightarrow{VUP} - (VPN \cdot VUP)\overrightarrow{VPN}$$

suponiendo que el vector VPN sea unitario, tal y como se ve en la figura 12. El vector u es simplemente aquel que permite que $\{u, v, VPN\}$ sea un sistema de coordenadas de mano izquierda. La solución es:

$$u = VPN \times v$$

El siguiente paso es un sencillo cambio de sistema de coordenadas, tal y como veíamos en el capítulo anterior, para pasar del sistema global de la escena al nuevo sistema centrado en el ojo. Recordemos que la matriz que me producía dicho cambio era

$$M = T \times R \times E$$

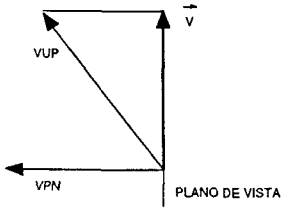


Figure 12

donde T era la traslación que me llevaba el origen del sistema nuevo al viejo, R la rotación que hacía coincidir ambos sistemas, y E la que unificaba las escalas. En nuestro caso, el escalado no es necesario. Veamos como construir las dos primeras matrices.

Ya que el centro de proyección COP viene definido relativo al VRP, la matriz de traslación será:

$$T = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -(VRP_x + COP_x) & -(VRP_y + COP_y) & -(VRP_z + COP_z) & 1 \end{pmatrix}$$

Para la matriz de rotación recordemos que simplemente había que colocar los ejes del nuevo sistema en las tres primeras columnas de la matriz, lo que nos queda:

$$R = \begin{pmatrix} u_x & v_x & -VPN_x & 0 \\ u_y & v_y & -VPN_y & 0 \\ u_z & v_z & -VPN_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Obsérvese el detalle de que el vector VPN va con signo negativo debido a que la construcción de la matriz está pensada para dos sistemas de mano derecha, por lo que el eje z debería tener el sentido contrario. Finalmente, hay que volver a restituir el sentido de la z, es decir, volver a invertir su signo. Esto se logra con una matriz de inversión de la z:

$$Ch = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

En las figuras 13 y 14 se muestra los estados inicial y final respectivamente de la transformación.

Obsérvese cómo el sistema de coordenadas global se ha transformado en el sistema de mano izquierda centrada en el ojo. El punto de referencia de vista VRP se ha

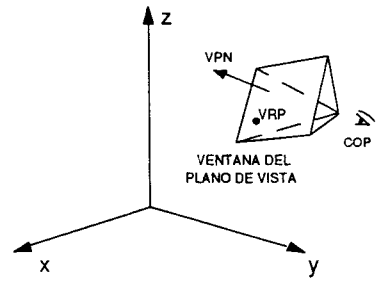


Figure 13

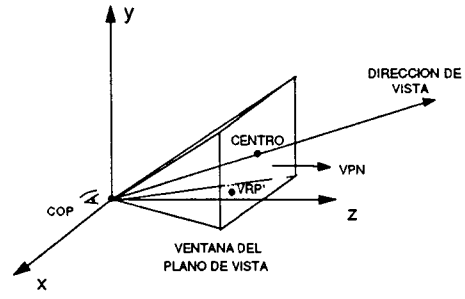


Figure 14

transformado en el punto

$$VRP' = VRP \times T \times R \times Ch$$

Aún nos queda una transformación más. Si nos fijamos en la figura anterior, la dirección de vista, esto es, la línea que une el ojo y el centro de la ventana, no coincide con el eje z. Sin embargo, ambas deberían coincidir para poder aplicar la proyección perspectiva tal y como la habíamos visto anteriormente. Lo que procede por tanto es realizar un afilamiento en la dirección z, de tal manera que todo lo que el ojo ve justo de frente pase a estar situado sobre el eje z. Además, de esta manera, no sólo conseguiremos centrar la dirección de vista sino también el volumen de vista (con lo que el proceso de recorte posterior será más sencillo).

Para calcular el afilamiento primero calcularemos las coordenadas del centro de la ventana. Ya que el COP está ahora en el origen de coordenadas, y el eje z es perpendicular al plano de vista, la componente z de cualquier punto sobre dicho plano será VRP'_z. Así, el centro de la ventana tendrá como componente z :

$$c_z = VRP'_z$$

Para evaluar las componentes x e y, examinemos la figura 15. Obsérvese como

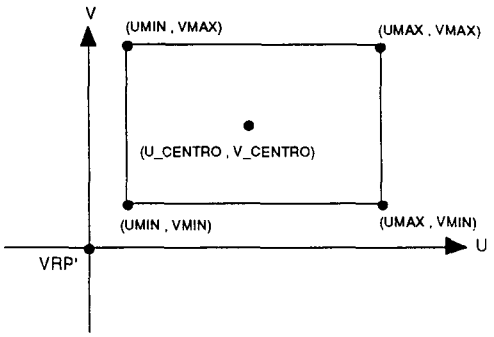


Figure 15

los ejes u y v son ahora paralelos a los ejes x e y. La componente v del centro es la mitad entre sus valores máximo y mínimo, y lo mismo para la componente u. Ya que estos valores vienen referenciados relativos a VRP'_z, las componentes para el centro nos quedan:

$$\begin{aligned} c_x &= VRP'_x + \frac{u_{max} + u_{min}}{2} \\ c_y &= VRP'_y + \frac{v_{max} + v_{min}}{2} \end{aligned}$$

Habiendo calculado las coordenadas de un punto sobre la línea que queremos "afilar", los valores de la tercera fila de la matriz de afilamiento corresponden a los coeficientes negativos de las componentes x e y, divididos por la componente z[1] :

$$A_z = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -\frac{c_x}{c_z} & -\frac{c_y}{c_z} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Esta tranformación enviará el centro de la ventana sobre el eje z, dejando la esquina superior derecha de la ventana en el punto $\{(u_{max} - u_{min})/2, (v_{max} - v_{min})/2, VRP'_z)\}$. En la figura 16 puede verse el resultado final. Toda la escena ha sido reorientada en función de la dirección de vista, de tal manera que la imagen que obtendremos será exactamente la que el supuesto observador vería sin ningún tipo de distorsión. Así hemos concluido esta sección, donde se ha construido la matriz de transformación de vista, la cual se obtenía como el producto de:

$$\boxed{M = T \times R \times Ch \times A_z}$$

[1]La recta que queremos afilar viene dada por la expresión $x = \frac{c_x}{c_z} z; y = \frac{c_y}{c_z} z$. Para formar la matriz de afilamiento hay que negar estos coeficientes.

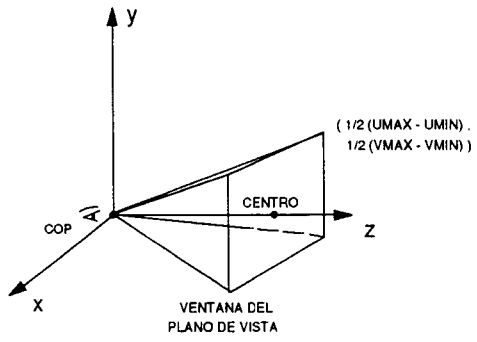


Figure 16

2.3 RECORTE 3D

Antes de proyectar la escena para obtener la imagen sobre el plano de vista, es necesario recortar o eliminar todos los objetos que caigan en el exterior del volumen de vista (truncado o no), ya que éstos no van a aparecer en nuestra imagen. Para hacer esto, debemos primero calcular las ecuaciones de los seis planos que definen dicho volumen y entonces, para todas las líneas que definen la imagen, calcular su intersección con cada uno de estos planos. Sólamente aquellos segmentos que se encuentren en el interior del volumen serán proyectados sobre el plano de vista.

Dependiendo de qué tipo de proyección vayamos a realizar, el volumen de vista será diferente. Es decir, para una proyección perspectiva, el volumen de vista será lógicamente una pirámide truncada. Por el contrario, si la proyección es paralela, el volumen será un prisma rectangular. Estos volúmenes pueden verse en la figura 17.

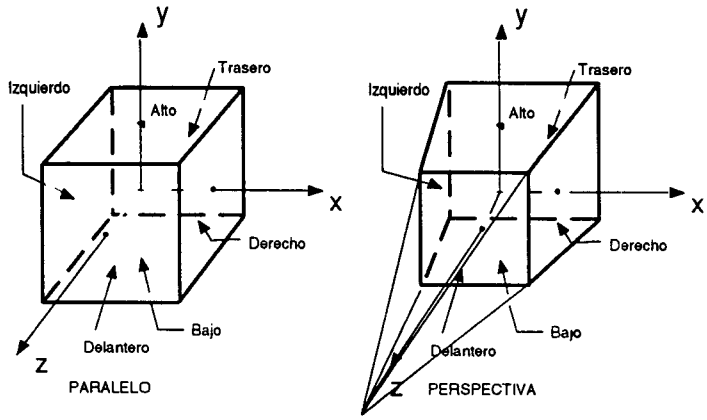


Figure 17

Existen distintos algoritmos de recorte que pueden usarse. Veamos unos cuantos.

2.3.1 Algoritmo de Cohen-Sutherland 3D

Al igual que en el caso 2D, este algoritmo clasifica las líneas en invisibles o visibles usando un código binario para cada uno de sus dos extremos. La diferencia con el caso bidimensional es que ahora necesitaremos seis bits por punto, ya que ahora hay seis planos. Cada bit indicará si el punto está a un lado o a otro de cada uno de los seis planos que definen el volumen, siendo 0 si el punto se encuentra en el interior del volumen, y 1 si está fuera. Así por ejemplo el primer bit puede representar la situación del punto respecto a la cara superior del volumen, el segundo bit con respecto a la cara derecha, etc. De esta forma, un punto que se encontrase por encima del volumen tendría por código 100000. Todos los puntos del interior del volumen van a tener por código 000000.

Para el caso de la proyección paralela, donde el volumen es un prisma, el cálculo de los código para cada punto es muy sencillo, ya que las caras del prisma son paralelas a una de las dirección del sistema de coordenada, y por tanto sólo hay que comparar el valor de esa componente del punto con el valor para la cara. Por ejemplo, si la cara de arriba tiene una altura $y = 1$, todos los puntos cuya componente y sea mayor que 1 estarán fuera del volumen.

Para el caso de la proyección perspectiva, el cálculo es ligeramente (sólo ligeramente) más complicado. Consideremos una vista aérea del volumen de recorte (figura 18). En

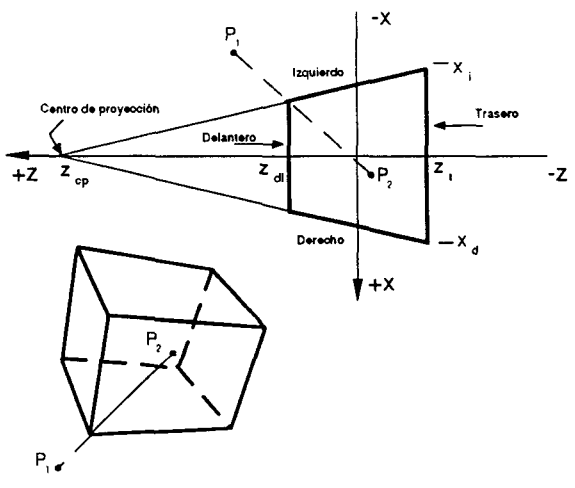


Figure 18

esta figura, la línea de abajo representa la cara derecha del volumen. Su ecuación será:

$$x = \frac{z - z_{COP}}{z_Y - z_{COP}} x_R = a_1 z + a_0$$

donde $a_1 = x_R/(z_Y - z_{COP})$ y $a_0 = -a_1 z_{COP}$. Esta ecuación puede ser usada para determinar si un punto está a un lado o a otro del plano. Es decir, para cada punto que queramos analizar, procederemos a evaluar la función

$$f_R(x, z) = x - a_1 z - a_0$$

donde x y z son las componentes de dicho punto. Es fácil observar como cuando esta función valga cero indicará que dicho punto está sobre la cara derecha, ya que verifica la ecuación de dicho plano. Cuando $f_R > 0$ significará que el valor de su componente x es mayor que $a_1 z + a_0$, lo cual quiere decir que el punto estará a la derecha del plano, y por lo tanto, fuera del volumen, con lo cual pondremos su bit correspondiente a 0. En el caso en que $f_R < 0$ y por un razonamiento similar, el punto estará a la izquierda de la cara y por tanto dentro del volumen, por lo que pondremos su bit a 1.

Para los restantes cinco planos crearemos funciones similares, de tal manera que cada una de ellas nos irá indicando el valor de cada bit correspondiente al código de cada punto. Además, los coeficientes de dichas funciones (a_1 y a_0 para el caso de f_R) son conocidos a priori, por lo que pueden calcularse de antemano. De esta forma, para evaluar cada bit sólo necesitaremos un producto, una suma y una comparación.

Una vez aclarado cómo podemos calcular los códigos para cada punto, el algoritmo queda como sigue: para cada línea de la escena se calculan los códigos de sus dos extremos, y se realiza una operación AND entre ellos. Dependiendo del resultado:

a) distinto de cero: la línea es totalmente invisible[2].

b) igual a cero sin ser ambos códigos cero: la línea es parcialmente visible.

c) ambos códigos iguales a cero: la línea es totalmente visible.

Una vez hemos procesado todas las líneas, aquéllas que sean invisibles quedan descartadas, aquéllas que sean totalmente visibles son las que pasarán a la fase de proyección, y aquéllas que sean parcialmente visibles habrá que calcular cuál es el punto de intersección con el volumen de vista, y excluir el segmento que quede por fuera.

Para calcular la intersección de la línea con la cara puede hacerse de la siguiente manera: supongamos que la ecuación del plano que representa a cada cara la tenemos almacenada en la forma $ax + by + cz + d = 0$. Para cada línea que procesemos, cuyos

[2]La explicación de por qué esto es así es idéntica a la que se da en el caso 2D, por lo que la omitimos aquí.

extremos sean P_1 y P_2, podemos expresar su ecuación en parámetricas como

$$\left.\begin{array}{l} x = P_{1x} + (P_{2x} - P_{1x})t = x_0 + x_1 t \\ y = P_{1y} + (P_{2y} - P_{1y})t = y_0 + y_1 t \\ z = P_{1z} + (P_{2z} - P_{1z})t = z_0 + z_1 t \end{array}\right\} \forall t \in [0,1]$$

Así, calcular la intersección entre la línea y el plano consiste simplemente en encontrar el punto que verifica ambas ecuaciones:

$$\begin{aligned} 0 &= a(x_0 + x_1 t) + b(y_0 + y_1 t) + c(z_0 + z_1 t) + d \\ t &= -\frac{ax_0 + by_0 + cz_0 + d}{ax_1 + by_1 + cz_1} \end{aligned}$$

Una vez obtenido el valor de t podemos obtener las componentes (x, y, z) del punto intersección. Podría pensarse que quizás sería más sencillo olvidarnos de la fase de calcular los códigos de los puntos para ver si tienen intersección, y en lugar de eso calcular directamente este valor t para todos los segmentos: si t se ecuentra en el intervalo $(0, 1)$ entonces existen intersección del segmento con la cara, y si no no. El problema es que evaluar este cálculo requiere muchas más operaciones aritméticas por cada punto que en el caso anterior, en donde a_1 y a_0 eran previamente conocidos.

2.3.2 Algoritmo de subdivisión del punto medio

Es una extensión del algoritmo anterior. Lo que hace es evitar el tener que calcular las intersecciones para los segmentos. Para ello, cuando se encuentra un segmento parcialmente visible, divide el segmento en dos y recursivamente trata por separado cada uno de ellos. Al final se quedará con el o los segmentos internos que deben dibujarse.

Por ejemplo, consideremos la situación de la anterior figura 18. Los extremos del segmento son $P_1 = (-600, -600, 600)$ y $P_2 = (100, 100, -100)$. El volumen de recorte viene dado por las coordenadas $x_R = y_T = 500$, $x_L = y_B = -500$, $z_H = 357.14$ y $z_Y = -500$. El centro de proyección es $Z_{COP} = 2500$. Las funciones para el test de los puntos nos salen:

$$\begin{aligned} \text{Derecha} &: f_R = 6x + z - 2500 \\ \text{Izquierda} &: f_L = 6x - z + 2500 \\ \text{Arriba} &: f_T = 6y + z - 2500 \\ \text{Abajo} &: f_B = 6y - z + 2500 \\ \text{Frente} &: f_H = z - 357.14 \\ \text{Atrás} &: f_Y = z + 2500 \end{aligned}$$

Con estas funciones los códigos resultantes salen 010101 para P_1 y 000000 para P_2. Puesto que ambos puntos no son cero, la línea no es totalmente visible. La intersección

lógica de ambos tampoco es cero, por lo que no es totalmente invisible. Esto implica que el segmento es parcialmente visible, con lo cual se procede a su subdivisión iterativa:

$(-600,-600,600)$	$(100,100,-100)$	$(-250,-250,250)$	se dibuja P_mP_2
$(-600,-600,600)$	$(-250,-250,250)$	$(-425,-425,425)$	se elimina P_1P_m
$(-425,-425,425)$	$(-250,-250,250)$	...	...

Se continúa de esta manera hasta que el segmento resultante sea lo suficientemente pequeño. Aunque a simple vista pueda parecer más costoso de evaluar que el algoritmo anterior, la diferencia radica en que usamos aritmética entera en lugar de real.

2.3.3 Algoritmo de Cyrus-Beck 3D

Este algoritmo es también una generalización de su versión bidimensional, sustituyendo los bordes de la región de recorte 2D por planos, y considerando las tres componentes de cada vector. Observemos la situación general de la figura 19, en donde tenemos un segmento arbitrario $\overline{P_1P_2}$ y un volumen de recorte.

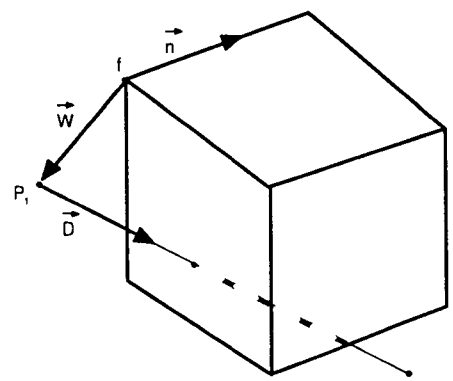


Figure 19

Los puntos del segmento P_1P_2 vienen dados por la expresión:

$$P(t) = P_1 + (P_2 - P_1)t, \forall t \in [0,1]$$

La intersección de esta línea con el plano que representa a una de las caras vendrá data por el t que cumpla

$$\overrightarrow{n} \cdot \overrightarrow{[P(t) - f]} = 0$$

donde $\overrightarrow{n}$ representa el vector normal a dicho plano, y f es un punto perteneciente a la cara. ¿Cuál es el significado de esta fórmula? En realidad no es más que un producto escalar de dos vectores, el cual como todos sabemos (o deberíamos saber) depende del

coseno del ángulo que forman ambos (el vector normal y el vector formado entre el punto f y un punto del segmento). Por lo tanto, es fácil ver como para $t = 0 \Rightarrow P(t) = P_1$, y el vector con origen en f forma un ángulo obtuso con el vector normal, por lo que el producto escalar es negativo. Para $t = 1 \Rightarrow P(t) = P_2$, y el ángulo entre ambos vectores es menor que 90°, siendo el producto escalar positivo. Finalmente, sólamente para el punto que esté sobre la cara existirá un ángulo de 90°, anulando así el producto escalar.

Esta expresión también puede representarse como

$$\begin{aligned} 0 &= \vec{n} \cdot \overrightarrow{[P(t) - f]} = \vec{n} \cdot \overrightarrow{[P_1 + (P_2 - P_1)t - f]} = \\ &= \vec{n} \cdot \overrightarrow{(P_1 - f)} + \vec{n} \cdot \overrightarrow{(P_2 - P_1)}t = \vec{n} \cdot \vec{w} + t(\vec{n} \cdot \vec{D}) \end{aligned}$$

siendo $\vec{D} = \overrightarrow{(P_2 - P_1)}$ y $\vec{w} = \overrightarrow{(P_1 - f)}$

La intersección por tanto viene dada por el t siguiente:

$$t = -\frac{\vec{n} \cdot \vec{w}}{\vec{n} \cdot \vec{D}}$$

Además, si consideramos que el sentido del vector normal es hacia el interior del volumen de recorte, obtendremos los siguientes resultados en función del signo del numerador:

$$\begin{aligned} &\text{si } \vec{n} \cdot \vec{w} < 0 \Rightarrow P_1 \text{ está fuera del volumen} \\ &\text{si } \vec{n} \cdot \vec{w} = 0 \Rightarrow P_1 \text{ está sobre el plano} \\ &\text{si } \vec{n} \cdot \vec{w} > 0 \Rightarrow P_1 \text{ está dentro del volumen}^3 \end{aligned}$$

Otra conclusión que obtenemos es que dependiendo del signo del denominador podemos saber si el t calculado indica el comienzo o el final de la parte del segmento que se encuentra dentro del volumen. Es decir, si $D \cdot n$ es positivo, y llamamos t_i al punto intersección del segmento con la cara, sabemos que dicho t_i es el t más pequeño que se encuentra en el interior del volumen. Es decir, la parte de la línea que se encuentra dentro del volumen comienza en $t = t_i$ y continúa para todo $t > t_i$. Si $D \cdot n < 0$ estaríamos en el caso contrario. Esta información será usada posteriormente por el algoritmo.

Veamos un ejemplo de recorte para comentar más detenidamente el algoritmo.

Recorte por un volumen rectangular

Sea la situación de la figura 20 donde $P_1 = (-2, -1, -1/2)$ y $P_2 = (3/2, 3/2, -1/2)$.

Las normales a cada una de las seis caras del volumen son:

$$\begin{aligned} &n_T = (0, -1, 0); \quad n_R = (-1, 0, 0); \quad n_H = (0, 0, -1); \\ &n_B = (0, 1, 0); \quad n_L = (1, 0, 0); \quad n_Y = (0, 0, 1); \end{aligned}$$

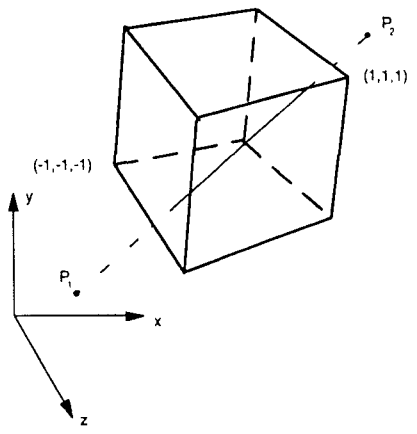


Figure 20

Los puntos en cada plano se seleccionan por simple inspección. Lo más sencillo es coger dos esquinas opuestas del volumen, es decir:

$$f_T = f_R = f_H = (1,1,1); \quad f_B = f_L = f_Y = (-1,-1,-1)$$

La dirección para la línea P_1P_2 es $D = (P_2 - P_1) = (7/2, 5/2, -1)$.

El algoritmo procede como sigue: para cada línea de la escena creamos una tabla en donde calculamos su intersección con cada una de las seis caras del volumen, así como otros resultados que nos serán útiles. En este ejemplo concreto la tabla resultante nos quedará:

PLANO	$\overrightarrow{n}$	f	$\overrightarrow{w}$	$\overrightarrow{w} \cdot \overrightarrow{n}$	$\overrightarrow{D} \cdot \overrightarrow{n}$	t_{min}	t_{max}
Derecha	$(-1,0,0)$	$(1,1,1)$	$(-3,-2,-1/2)$	3	$-7/2$		6/7
Izquierda	$(1,0,0)$	$(-1,-1,-1)$	$(-1,0,3/2)$	-1	$7/2$	2/7	
Arriba	$(0,-1,0)$	$(1,1,1)$	$(-3,-2,-1/2)$	2	$-5/2$		4/5
Abajo	$(0,1,0)$	$(-1,-1,-1)$	$(-1,0,3/2)$	0	$5/2$	0	
Frente	$(0,0,-1)$	$(1,1,1)$	$(-3,-2,-1/2)$	$1/2$	1	$-1/2$	
Atrás	$(0,0,1)$	$(-1,-1,-1)$	$(-1,0,3/2)$	$3/2$	-1		$3/2$

Por ejemplo, para la cara derecha, P_1 se encuentra por el lado interior de la cara $(w \cdot n > 0)$, y el valor del t de la intersección será el mayor valor posible de t que se halle dentro del volumen $(D \cdot n < 0)$. Es por esto por lo que hay dos columnas donde colocar los valores de t : t_{min} si el punto representa el valor mínimo de t $(D \cdot n > 0)$ o t_{max} si representa el valor máximo $(D \cdot n < 0)$.

Tras haber calculado la tabla, se elige el máximo t_1 de la columna de mínimos (t_{min}) y el mínimo t_2 de la columna de máximos (t_{max}), y estos dos puntos son los que

representarán la porción de segmento que queda en el interior del volumen. En este ejemplo, $t_1 = 2/7$ y $t_2 = 4/5$. La línea resultante será el segmento $P'_1P'_2$, siendo

$$\begin{aligned} P'_1 &= P(t_1) = P_1 + \overrightarrow{D}t_1 = (-1, -2/7, 3/14) \\ P'_2 &= P(t_2) = P_1 + \overrightarrow{D}t_2 = (4/5, 1, -3/10) \end{aligned}$$

Un detalle a tener en cuenta es que siempre t_1 debe ser menor que t_2, y que ambos deben encontrarse en el intervalo $[0, 1]$.

Recorte por un volumen en perspectiva

Se resuelve de manera idéntica al caso anterior. Sea el ejemplo de la figura 21.

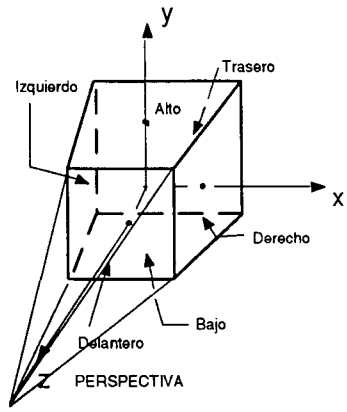


Figure 21

donde el centro de proyección (COP) se encuentra en $(0, 0, -5)$, y las esquinas de la ventana de recorte son $(1, 1)$ y $(-1, -1)$.

Los vectores que unen el COP con las esquinas de la ventana son

$$v_1 = (1, 1, -5); \quad v_2 = (-1, 1, -5); \quad v_3 = (-1, -1, -5); \quad v_4 = (1, -1, -5)$$

Multiplicando vectorialmente estos cuatro vectores podemos calcular cuatro de las normales a las caras:

$$\begin{aligned} n_L &= v_3 \times v_2 = (10, 0, -2); & n_R &= v_1 \times v_4 = (-10, 0, -2); \\ n_T &= v_2 \times v_1 = (0, -10, -2); & n_B &= v_4 \times v_3 = (0, 10, -2); \end{aligned}$$

Las otras dos son simplemente vectores paralelos al eje z:

$$n_H = (0, 0, -1); \quad n_Y = (0, 0, 1);$$

Los puntos pertenecientes al plano serán:

$$f_T = f_L = f_B = f_R = (0,0,5); \quad f_H = (0,0,1); \quad f_Y = (0,0,-1)$$

Ahora ya podemos calcular la tabla correspondiente a la arista que figura en el dibujo anterior:

PLANO	$\overrightarrow{n}$	f	$\overrightarrow{w}$	$\overrightarrow{w} \cdot \overrightarrow{n}$	$\overrightarrow{D} \cdot \overrightarrow{n}$	t_{min}	t_{max}
Derecha	$(-10,0,-2)$	$(0,0,5)$	$(-2,-1,-9/2)$	29	-33		0.879
Izquierda	$(10,0,-2)$	$(0,0,5)$	$(-2,-1,-9/2)$	-11	37	0.297	
Arriba	$(0,-10,-2)$	$(0,0,5)$	$(-2,-1,-9/2)$	19	-23		0.826
Abajo	$(0,10,-2)$	$(0,0,5)$	$(-2,-1,-9/2)$	-1	27	0.037	
Frente	$(0,0,-1)$	$(0,0,1)$	$(-2,-1,-1/2)$	$1/2$	1	-0.5	
Atrás	$(0,0,1)$	$(0,0,-1)$	$(-2,-1,3/2)$	$3/2$	-1		1.5

Por último se escoge el máximo para t_{min} ($t_1 = 0.297$) y el mínimo para t_{max} ($t_2 = 0.826$), ajustándolos siempre al intervalo $[0,1]$ si fuera el caso, y verificando que $t_{min} < t_{max}$. Esto se cumple y por lo tanto el segmento resultante del interior del volumen de recorte es el segmento $P_1'P_2'$, siendo

$$\begin{aligned} P_1' &= P(t_1) = P_1 + \overrightarrow{D}t_1 = (-0.961, -0.258, 0.203) \\ P_2' &= P(t_2) = P_1 + \overrightarrow{D}t_2 = (0.891, 1.065, -0.323) \end{aligned}$$

Chapter 3 REPRESENTACIÓN DE CURVAS Y SUPERFICIES

La mayor parte de los objetos existentes en el mundo real presentan formas continuas y suaves, y para nada pueden asemejarse a formas poligonales. La mayoría de las aplicaciones gráficas necesitan modelar objetos presentes en el mundo real, y por lo tanto se hace necesario poder generar curvas y superficies de una forma más exacta que una simple sucesión de segmentos rectos cortos. El diseño asistido por ordenador (CAD), los tipos de letra de alta calidad, los trazos de un artista, el path que debe seguir una cámara a lo largo de una secuencia; todos ellos contienen superficies y curvas suaves.

La necesidad de poder representar curvas y superficies viene por dos sentidos. En primer lugar para modelar objetos existentes (coches. caras, montañas, etc.), en donde una descripción matemática del objeto puede no estar disponible. Por supuesto, podemos usar para modelar las coordenadas de los infinitos puntos del objeto, lo cual no es nada viable. También podemos representarlo aproximadamente usando planos, esferas u otras primitivas sencillas, fáciles de describir matemáticamente, pero seguiríamos sin lograr una representación adecuada.

En segundo lugar para modelar algo que aún no exista físicamente, por ejemplo cuando se está diseñando un nuevo prototipo de coche o de avión. Para crearlo, el usuario puede ir esculpiéndolo interactivamente, o bien especificar su descripción matemática, o dar una descripción aproximada para luego ir retocándola.

En este capítulo vamos a ver el modelado de curvas y superficies paramétricas, con lo cual obtendremos una visión general del área de modelado geométrico, el cual es un amplísimo campo dentro de la informática gráfica.

3.1 MALLAS POLIGONALES (MESHES)

Las mallas poligonales o meshes fueron el primer sistema de representación de objetos, ya que era el más evidente y práctico para las máquinas existentes cuando comenzaron las primeras aplicaciones gráficas. Todavía hoy en día se siguen usando, por supuesto, salvo en aquellas aplicaciones que precisen mayor exactitud tanto en el modelado como en los cálculos posteriores (áreas, volúmenes, etc.). Vamos a definir por malla poligonal a un conjunto de aristas, vértices y polígonos conectados donde cada arista es compartida al menos por dos polígonos. Una arista conecta dos vértices, y un polígono es una secuencia cerrada de aristas. Una arista puede ser compartida por dos polígonos, y un vértices es compartido al menos por dos aristas.

Un mesh puede representarse de varias maneras, cada una con sus ventajas y desventajas. Es tarea del programador de la aplicación el elegir la representación más apropi-

ada, o incluso es posible elegir varias: una para almacenamiento, otra para uso interno de la aplicación, otra para que el usuario interactúe con ella, etc. Las operaciones típicas que pueden hacerse sobre un mesh son encontrar todas las aristas incidentes a un vértice, encontrar los polígonos que comparten una arista o un vértice, encontrar los vértices conectados por una arista, encontrar las aristas de un polígono, visualizar el mesh, e identificar errores en la representación (como una arista o un vértice perdido). En general, mientras más explícita sea la representación, más rápidas serán las operaciones, pero más espacio de almacenamiento van a requerir.

Tipos de Representación

a) Representación explícita. Cada polígono viene representado por una lista de coordenadas de vértices:

$$P_i = ((x_1, y_1, z_1), (x_2, y_2, z_2), ..., (x_n, y_n, z_n))$$

Los vértices se almacenan en el mismo orden que si recorriéramos el polígono a traves de su contorno. Existe una arista entre cada dos vértices consecutivos del vector, y una arista más que conecta el último vértice con el primero.

Para un simple polígono, esta representación es bastante eficiente, pero para un mesh se pierde mucho espacio, ya que las coordenadas de los vértices compartidos se hallan duplicadas. Además, otra desventaja es que no hay representación explícita de las aristas y los vértices compartidos. Por ejemplo, para arrastar un vértice y todas sus aristas incidentes de forma interactiva, debemos primero encontrar todos los polígonos que comparten dicho vértice.

Para visualizar el mesh necesitaremos transformar cada vértice y recortar cada arista de cada polígono. Si dibujamos las aristas, aquellas que estén compartidas se dibujarán dos veces, lo cual puede causar problemas, aparte de tardar el doble de tiempo.

b) Punteros a una lista de vértices. Cada vértice del mesh se almacena una sola vez, en una lista de vértices:

$$V = ((x_1, y_1, z_1), (x_2, y_2, z_2), ..., (x_n, y_n, z_n))$$

Cada polígono se define como una lista de índices o punteros a la lista de vectores. Así, un polígono formado por los vértices 3,5,6 y 10 vendría representado por el vector

$$P_i = (3, 5, 7, 10)$$

Esta representación, que puede verse en la figura 1, tiene varias ventajas sobre el tipo anterior. Ya que cada vértice se almacena sólo una vez se ahorra mucho espacio. Por otro lado, la modificación de las coordenadas de los vértices se hace de forma directa, sin repercutir en la información de los polígonos. Sin embargo, aún sigue

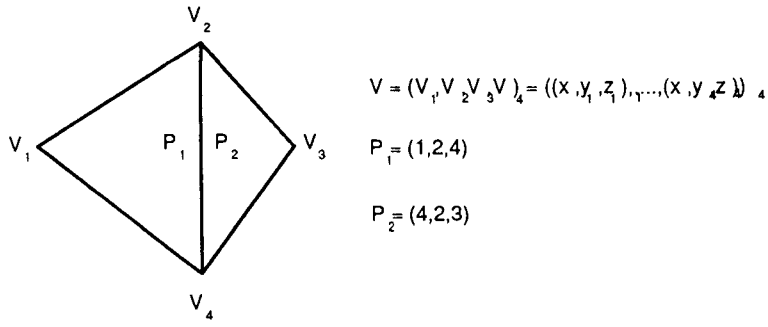


Figure 1

siendo complicado el encontrar los polígonos que comparten una arista, y las aristas compartidas continúan dibujándose dos veces.

c) Punteros a una lista de aristas. Seguimos teniendo una lista V de vértices, y además vamos a representar cada arista por separado como un nuevo vector, en donde se indica cuáles son sus dos vértices extremos y a qué dos polígonos pertenece:

$$P_i = (A_1, A_2, ..., A_n); \quad A_j = (V_1, V_2, P_1, P_2)$$

Cuando una arista pertenece a un solo polígono (en los contornos del mesh) P_2 se pone a nulo. Un ejemplo puede verse en la figura 2.

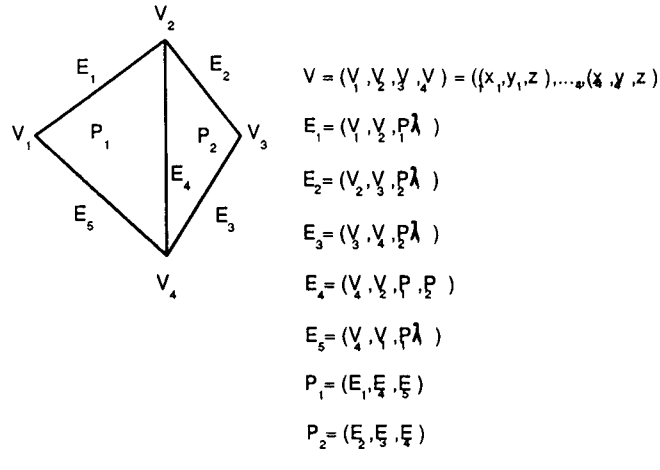


Figure 2

Para visualizar el mesh simplemente hay que dibujar todas las aristas, en lugar de dibujar los polígono, evitando así las transformaciones redundantes.

En ninguna de estas tres representaciones es fácil determinar qué aristas son incidentes a un vértice, pues habría que inspeccionar todas las aristas. Por supuesto que podemos añadir información que explícitamente indique tales relaciones, pero a cambio de pagar más coste de almacenamiento.

Consistencia de las representaciones

Los meshes suelen generarse interactivamente, a menudo de forma automática como resultado de digitalizaciones, por lo cual es inevitable encontrar errores en la representación. Por lo tanto se hace necesario asegurarse que los polígonos se encuentran todos cerrados, que todas las aristas son usadas al menos una vez, y que cada vértice es referenciado por al menos dos aristas. En algunas aplicaciones incluso se exige que el mesh esté completamente conectado (cualquier vértice puede alcanzarse desde cualquier otro viajando por las aristas), que sea topológicamente plano (las relaciones binarias en los vértices definidas por las aristas puede representarse por un grafo planar) y que no contenga agujeros. De las tres representaciones anteriores, el esquema de punteros a aristas es el más sencillo de chequear, ya que contiene la máxima información.

La ecuación del plano

Cuando trabajamos con cada polígono a veces se hace necesario disponer de la ecuación del plano sobre el que se encuentra. En algunos casos dicha ecuación es conocida implícitamente debido al método de construcción interactivo que se ha usado para definir el polígono. Pero si no se conoce, siempre podemos usar las coordenadas de los tres vértices para encontrar el plano. La ecuación de un plano se define como:

$$ax + by + cz + d = 0$$

Los coeficientes a, b, c definen la normal al plano, $N = (a, b, c)$. Esta normal es fácil de calcular si disponemos de tres puntos, P_1, P_2, P_3, pertenecientes al plano. Sólo hay que evaluar el producto vectorial

$$N = \overrightarrow{P_1P_2} \times \overrightarrow{P_1P_3}$$

Si el producto es cero significará que los tres puntos son colineales y que los tres puntos no definen un plano, por lo que necesitaremos un vértice más. Una vez calculados los coeficientes a, b, c sólo queda calcular d, para lo cual evaluaremos la ecuación del plano en cualquiera de los tres puntos dados y la despejaremos.

Si hay más de tres vértices en el polígono, éstos pueden no ser coplanares, lo cual puede acarrear numerosos problemas. En estos casos es mejor usar otra técnica diferente para encontrar los coeficientes que mejor aproximan el plano a todos los vértices. Una vez calculados podemos determinar entonces una medida de lo "no-plano" que es nuestro polígono, en base a calcular las distancias perpendiculares al plano de cada

vértice. La distancia D de un punto a un plano viene dada por la expresión

$$D = \frac{ax + by + cz + d}{\sqrt{a^2 + b^2 + c^2}}$$

Esta distancia puede ser positiva o negativa dependiendo de a qué lado del plano se encuentra el punto. Si el vértice se encuentra sobre el plano, $D = 0$.

3.2 CURVAS CÚBICAS PARAMÉTRICAS

Cualquier línea curva puede representarse mediante un conjunto de segmentos rectos. Cuanto mayor sea el número de ellos mejor será la aproximación. De igual manera, cualquier tipo de superficie puede aproximarse por un conjunto de polígonos conectados (mesh), acercándonos más a la superficie original cuanto más polígonos usemos. El principal problema de este tipo de aproximaciones es la enorme cantidad de coordenadas de cada punto (extremos de cada segmento recto en las curvas, o vértices de cada polígono en las superficies) que debemos almacenar para lograr una aproximación razonable. Además, otro problema adicional es que la manipulación interactiva de los datos para aproximar una forma dada es tediosa, debido a los numerosos puntos que hay que posicionar correctamente.

Para solucionar este hándicap, existe una forma de representación mas manipulable y compacta para representar curvas y superficies. La aproximación general consiste en usar funciones de mayor grado que las funciones lineales (que es lo que son los segmentos rectos y los polígonos). Por ejemplo, si tenemos tres puntos no podemos obtener una recta que pase por los tres, si éstos no son colineales. Esto es debido a que la expresión para una recta es $y = ax + b$, por lo que sólo disponemos de dos coeficientes (a y b) para intentar cumplir tres condiciones (una para obligar que pase por cada punto); necesitaríamos guardar dos rectas. Sin embargo, con un polinomio de grado dos tendríamos tres coeficientes, con lo que sí conseguiríamos obtener una única función que pasara por los tres puntos. De ahí que el usar funciones de mayor grado nos permiten aproximar mejor y con menos información un trozo de curva que mediante segmentos rectos.

Estas funciones seguirán siendo todavía aproximaciones a la forma deseada, pero requerirán mucho menos espacio de almacenamiento y ofrecerán una manipulación interactiva mucho más sencilla que con las funciones lineales. Para este tipo de aproximaciones de mayor grado existen tres métodos posibles:

a) Podemos expresar las componentes y y z como **funciones explícitas** de x

$$\left.\begin{array}{l} y = f(x) \\ z = g(x) \end{array}\right\}$$

Un primer problema de esta representación es que es imposible obtener múltiples valores de y para un valor de x, por lo que las curvas tales como circunferencias y elipses deben

representarse por varios segmentos curvos. Un segundo problema es que esta definición no es invariante frente a rotaciones; describir una versión rotada de la curva requeriría un gran coste e incluso se necesitaría dividir la curva en varias partes. Un problema más es que describir curvas que posean tangente vertical es muy difícil, ya que es complicado representar una pendiente infinita.

b) Podemos elegir como modelo de la curva la solución de una **ecuación implícita** de la forma

$$f(x, y, z) = 0$$

Esta solución también supone nuevos problemas. En primer lugar, la ecuación dada puede tener más soluciones de las que queremos. Por ejemplo, para modelar una circunferencia podemos usar la ecuación $x^2 + y^2 = 1$, pero ¿cómo podríamos modelar una semicircunferencia? Deberíamos añadir requisitos tales como $x \geq 0$ que no se hayan incluidos dentro de la propia ecuación. En segundo lugar, si deseamos unir dos segmentos curvos definidos en forma implícita, puede ser muy difícil determinar si coinciden sus tangentes en el punto de unión. La continuidad de las tangentes es un detalle crítico en la mayoría de las aplicaciones.

c) La **representación paramétrica** para las curvas,

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

soluciona todos los problemas de los dos métodos anteriores, y ofrece además una serie de ventajas adicionales que la hacen ideal para lo que queríamos en un principio. Las curvas paramétricas sustituyen el uso de las pendientes geométricas (las cuales pueden ser infinitas) por vectores tangentes paramétricos (los cuales nunca serán infinitos). De esta manera, una curva va a aproximarse por segmentos curvos polinómicos en lugar de segmentos rectos como antes. Cada segmento curvo Q de la curva completa vendrá representado por tres funciones, x, y, y z, las cuales serán polinomios cúbicos en t.

Se suelen usar los polinomios cúbicos porque los de grado dos ofrecen poca flexibilidad a la hora de controlar la forma de la curva, y los de grado mayor que tres pueden introducir rizos innecesarios y además requieren más computación. Ninguna representación de grado inferior a tres permite que un segmento curvo pase a través de dos puntos dados con dos derivadas específicas en cada uno. Dado un polinomio cúbico con cuatro coeficientes, los usaremos como incógnitas para poder cumplir el sistema de ecuaciones formado por las cuatro condiciones que hemos dicho. Esto es similar a lo que hacemos con una recta: dos coeficientes y dos condiciones por cumplir (los dos extremos). Las derivadas de cada extremos vienen ya definidas por la propia recta, y no pueden ser controladas independientemente. Con polinomios cuadráticos (de grado dos) tenemos tres coeficientes, lo cual nos sirve para los dos extremos y alguna condición más.

Otro detalle más es que las curvas polinómicas cúbicas son las curvas de menor grado que pueden ser no planares en 3D. Es decir, un polinomio de grado dos con

tres coeficientes viene dado por tres puntos, los cuales siempre forman un plano en el espacio. La curva resultante por lo tanto permanecerá sobre ese plano.

Los polinomios cúbicos que definen un segmento curvo $Q(t) = [x(t), y(t), z(t)]$ son de la forma

$$\left.\begin{array}{l} x(t) = a_x t^3 + b_x t^2 + c_x t + d_x \\ y(t) = a_y t^3 + b_y t^2 + c_y t + d_y \\ z(t) = a_z t^3 + b_z t^2 + c_z t + d_z \end{array}\right\} \forall t \in [0,1]$$

Para tratar con segmentos finitos de la curva, y sin pérdida de generalidad, hemos restringido el parámetro t al intervalo $[0,1]$. Definiendo el vector $T = [t^3, t^2, t, 1]$ y definiendo también la matriz C de coeficientes de los tres polinomios como

$$C = \begin{pmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \\ d_x & d_y & d_z \end{pmatrix}$$

podemos reescribir la ecuación para la curva como

$$Q(t) = \begin{pmatrix} x(t) & y(t) & z(t) \end{pmatrix} = T \times C$$

En la figura 3 pueden verse dos segmentos curvos cúbicos paramétricos unidos y sus respectivos polinomios. Además, puede verse la habilidad de la representación paramétrica para representar de forma sencilla múltiples valores de y para un solo valor de x mediante polinomios explícitos en t.

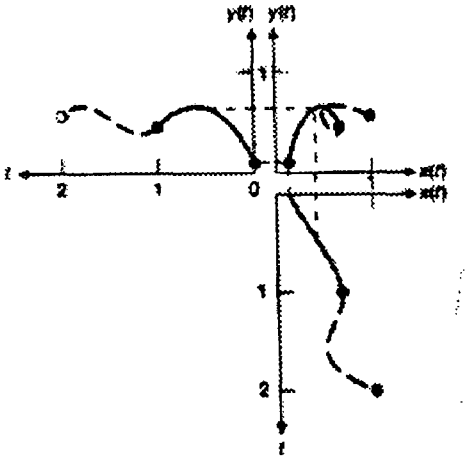


Figure 3

La derivada del vector $Q(t)$ es el vector tangente paramétrico de la curva. Aplicando esta definición obtenemos la siguiente ecuación:

$$\frac{d}{dt}Q(t) \;=\; Q'(t) = \begin{pmatrix} x'(t) & y'(t) & z'(t) \end{pmatrix} = \frac{d}{dt}T \times C =$$

$$= \begin{pmatrix} 3t^2 & 2t & 1 & 0 \end{pmatrix} \times C =$$
$$= \begin{pmatrix} 3a_x t^2 + 2b_x t + c_x & 3a_y t^2 + 2b_y t + c_y & 3a_z t^2 + 2b_z t + c_z \end{pmatrix}$$

Si dos segmentos curvos se unen en un extremo, diremos que la curva posee **continuidad geométrica** G^0. Si las direcciones (pero no necesariamente las magnitudes) de los vectores tangentes de ambos vectores son iguales en el punto común, diremos que la curva tiene continuidad geométrica G^1. En los objetos hechos en CAD (Diseño Asistido por Ordenador) frecuentemente se requiere la continuidad G^1 entre segmentos consecutivos. Esta continuidad significa que las pendientes de los segmentos son iguales en el punto de unión. Para que dos vectores tangentes V_1 y V_2 tengan la misma dirección debe cumplirse que $V_1 = k \cdot V_2$, con $k > 0$.

Si los vectores tangentes de dos segmentos curvos son iguales (tanto en dirección como en magnitud) en su punto de unión, diremos que la curva tiene continuidad paramétrica C^1. En general, si la dirección y magnitud del vector $d^n Q(t)/dt^n$ es igual para ambos segmentos, diremos que la curva tiene continuidad C^n. En la figura 4 pueden verse varias curvas con grados diferentes de continuidad. No se olviden que cualquier segmento curvo es siempre continuo en su interior; aquí nos estamos refiriendo a la continuidad de los extremos.

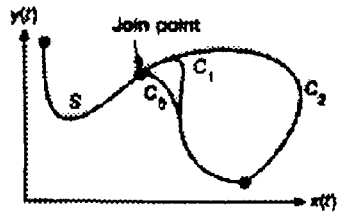


Figure 4

El vector tangente $Q'(t)$ mide la velocidad de un punto de la curva con respecto al parámetro t. Similarmente, la segunda derivada de $Q(t)$ representa la aceleración. Si una cámara se mueve a lo largo de una curva paramétrica a incrementos constantes de tiempo y va grabando un fotograma en cada paso, el vector tangente mide la velocidad de la cámara a lo largo de la curva. Tanto la velocidad como la aceleración de la cámara en los puntos extremos entre segmentos debe ser continua, para evitar movimientos bruscos en la secuencia resultante. Es precisamente esta aceleración en el punto de unión de la figura previa la que provoca que el segmento C^2 se alargue más a la derecha que el segmento C^1, antes de doblarse hacia el extremo final.

En general, la continuidad C^1 implica G^1, pero no al revés, ya que la continuidad G^1 es menos restrictiva que C^1. Sin embargo, visualmente no se aprecia diferente, como puede verse en la figura 5.

El dibujo de una curva paramétrica es completamente diferente al de una función

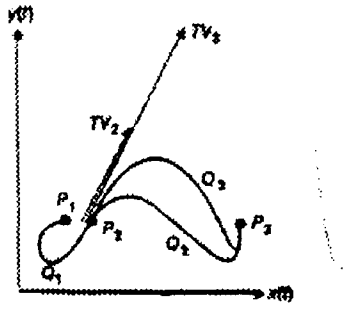


Figure 5

ordinaria, en la cual la variable independiente se va trazando sobre el eje x y la dependiente sobre el eje y. En un plot paramétrico, la variable independiente t nunca se dibuja. Esto significa que no puede determinarse a simple vista el vector tangente a la curva, ya que aunque la dirección es posible de determinar, no ocurre lo mismo con su magnitud. Esto puede verse en el siguiente ejemplo: sea el segmento $\gamma(t)$, $0 \leq t \leq 1$, cuyo vector tangente en $t = 0$ será $\gamma'(0)$. Si creamos un nuevo segmento, $\eta(t) = \gamma(2t)$, $0 \leq t \leq 1/2$, entonces los dibujos de ambas curvas son idénticos, y sin embargo, $\eta'(0) = 2\gamma'(0)$[1]. Esto demuestra que dos curvas con idéntico dibujo tienen diferentes vectores tangente. Este es el motivo para la definición de la continuidad geométrica: para que dos segmentos se unan suavemente sólo es preciso que las direcciones de las tangentes coincidan, independientemente de sus magnitudes.

Un segmento de curva $Q(t)$ viene definido, como ya hemos dicho, por dos puntos extremos, dos vectores tangentes y una continuidad entre segmentos consecutivos. Cada polinomio cúbico tiene cuatro coeficientes, por lo que se necesitan cuatro condiciones que nos permitan formular cuatro ecuaciones cuyas incógnitas a calcular sean precisamente estos cuatro coeficientes. Existen diferentes tipos de curvas cúbicas, que veremos posteriormente en este capítulo. Para ver como dichos coeficientes dependen de las cuatro condiciones, consideremos que nuestro segmento de curva viene definido como

$$Q(t) = T \cdot C$$

siendo $C = M \cdot G$, donde a su vez M es una **matriz base** de 4x4, y G un vector columna de cuatro elementos de requisitos geométricos, llamado **vector de geometría**. Los requisitos geométricos van a ser precisamente las condiciones establecidas, tales como los puntos extremos o los vectores tangentes, que definen la curva en concreto. Definimos G_x para referirnos al vector columna formado exclusivamente por las componentes x del vector de geometría. G_y y G_z se definen de forma similar. Tanto M como G diferirán para cada tipo de curva cúbica.

Los elementos de M y G son constantes, por lo que el producto $T \cdot M \cdot G$ nos da

[1] $\eta(t) = \gamma(2t) = \gamma(u)$, siendo $u = 2t$

$$\eta'(t) = \frac{d\eta(t)}{dt} = \frac{d\gamma(u)}{dt} = \frac{d\gamma(u)}{du}\frac{du}{dt} = 2\gamma'(u)$$

tres polinomios cúbicos en t. Expandiendo el producto $Q(t) = T \cdot M \cdot G$ obtenemos[2]

$$Q(t) = \begin{pmatrix} x(t) & y(t) & z(t) \end{pmatrix} = \begin{pmatrix} t^3 & t^2 & t & 1 \end{pmatrix} \begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{pmatrix} \begin{pmatrix} G_1 \\ G_2 \\ G_3 \\ G_4 \end{pmatrix}$$

El resultado para el polinomio $x(t)$ queda

$$\begin{aligned} x(t) \;=\; & (t^3 m_{11} + t^2 m_{21} + t m_{31} + m_{41}) g_{1x} + (t^3 m_{12} + t^2 m_{22} + t m_{32} + m_{42}) g_{2x} \\ & + (t^3 m_{13} + t^2 m_{23} + t m_{33} + m_{43}) g_{3x} + (t^3 m_{14} + t^2 m_{24} + t m_{34} + m_{44}) g_{4x} \end{aligned}$$

Los polinomios $y(t)$ y $z(t)$ quedan de forma similar.

Si observamos atentamente este polinomio, puede verse como la curva es una suma ponderada de los elementos de la matriz de geometría. Es decir, para cada valor de t

$$x(t) = b_1(t) g_{1x} + b_2(t) g_{2x} + b_3(t) g_{3x} + b_4(t) g_{4x}$$

donde los pesos que multiplican a cada elemento de G son los polinomios cúbicos en t. A estas funciones se les denonima **funciones blending** (to blend=doblar) ya que dependiendo de la forma de estas funciones se van interpolando los valores geométricos de la curva para cada valor de t. Estas funciones se obtienen haciendo $B = T \cdot M$.

Esta forma de representación puede parecer compleja, pero en realidad no es más que una generalización de la aproximación lineal usando segmentos rectos (polinomios de grado 1), para la cual sólo se necesitan dos condiciones geométricas, que son los dos puntos extremos. Así, cada segmento viene definido por la recta que pasa entre los puntos G_1 y G_2 :

$$\begin{cases} x(t) = g_{1x}(1-t) + g_{2x}(t) \\ y(t) = g_{1y}(1-t) + g_{2y}(t) \\ z(t) = g_{1z}(1-t) + g_{2z}(t) \end{cases}$$

¿Cómo se vería esta aproximación lineal usando la misma sintaxis que empleamos para la curva cúbica? Sea el segmento recto (en un caso bidimensional se verá más fácil) de la figura 6.

En coordenadas paramétricas parece claro que la ecuación será de la forma:

$$\left.\begin{aligned} x(t) &= x_1 + t(x_2 - x_1) \\ y(t) &= y_1 + t(y_2 - y_1) \end{aligned}\right\} \text{ siendo } 0 \le t \le 1$$

Esto puede ponerse también como

$$\begin{aligned} x(t) &= (1-t)x_1 + t x_2 \\ y(t) &= (1-t)y_1 + t y_2 \end{aligned}$$

[2]Nótese que cada G_i es en realidad una terna de valores xyz. Es decir, aunque lo llamemos vector de geometría, a efectos de números reales, G sería una matriz 4x3, donde G_x sería la primera columna, G_y la segunda y G_z la tercera.

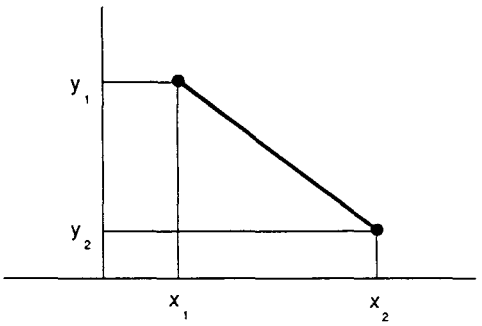


Figure 6

Por lo tanto el segmento quedará como sigue:

$$Q(t) = \begin{pmatrix} x(t) & y(t) \end{pmatrix} = B \cdot G = B \cdot \begin{pmatrix} x_1 & y_1 \\ x_2 & y_2 \end{pmatrix}$$

De aquí deducimos que el tamaño de la matriz B ha de ser 1x2. El elemento b_{11} será igual al polinomio que multiplica tanto a x_1 como a y_1, es decir, $b_{11}(t) = (1-t)$. De igua manera deducimos $b_{12}(t) = t$. Finalmente, ya podemos calcular la matriz base:

$$\begin{aligned} B &= T \cdot M = \begin{pmatrix} t & 1 \end{pmatrix} \begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix} \\ M &= \begin{pmatrix} -1 & 1 \\ 1 & 0 \end{pmatrix} \end{aligned}$$

En resumen, un segmento recto vendrá dado por la ecuación

$$Q(t) = T \cdot M \cdot G = \begin{pmatrix} t & 1 \end{pmatrix} \begin{pmatrix} -1 & 1 \\ 1 & 0 \end{pmatrix} G$$

siendo G el vector de geometría formado por los dos puntos extremos.

Dejando a un lado este ejemplo 2D, pasemos a continuación a ver las matrices base para dos tipos distintos de curvas cúbicas: las curvas de Hermite, y las curvas de Bezier. Existen más tipos de curvas, pero aquí sólo veremos estos dos.

3.2.1 Curvas de Hermite

Un segmento curvo de Hermite viene determinado por sus dos puntos extremos, P_1 y P_4, y dos vectores tangentes en dichos puntos, R_1 y R_4. Así, el vector de geometría de

Hermite viene definido por

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

y las expresiones para el polinomio cúbico de cada componente (x por ejemplo) será de la forma:

$$x(t) = \begin{pmatrix} t^3 & t^2 & t & 1 \end{pmatrix} \cdot M_H \cdot G_{Hx}$$

Las condiciones para que la curva comienze y finalize en cada extremo son:

$$\begin{aligned} x(0) &= P_{1x} = \begin{pmatrix} 0 & 0 & 0 & 1 \end{pmatrix} \cdot M_H \cdot G_{Hx} \\ x(1) &= P_{4x} = \begin{pmatrix} 1 & 1 & 1 & 1 \end{pmatrix} \cdot M_H \cdot G_{Hx} \end{aligned}$$

Para obtener las otras dos ecuaciones referentes a las tangentes habra que encontrar la derivada de x respecto de t :

$$\begin{aligned} x'(t) &= \begin{pmatrix} 3t^2 & 2t & 1 & 0 \end{pmatrix} \cdot M_H \cdot G_{Hx} \\ x'(0) &= R_{1x} = \begin{pmatrix} 0 & 0 & 1 & 0 \end{pmatrix} \cdot M_H \cdot G_{Hx} \\ x'(1) &= R_{4x} = \begin{pmatrix} 3 & 2 & 1 & 0 \end{pmatrix} \cdot M_H \cdot G_{Hx} \end{aligned}$$

Coloquemos ahora las cuatro ecuaciones en forma matricial:

$$\begin{pmatrix} P_1 \\ P_4 \\ R_1 \\ R_4 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{pmatrix} \cdot M_H \cdot G_{Hx}$$

Ahora fijémonos en una cosa curiosa. En realidad, el vector columna que aparece a la izquierda de la igualdad coincide con G_{Hx}! Por lo tanto, el producto de la matriz 4x4 por M_H debe ser igual a la matriz identidad para que la igualdad se cumpla. De aquí deducimos M_H

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

M_H es por supuesto única, y puede usarse en la expresión $x(t) = T \cdot M \cdot G_{Hx}$ para encontrar el polinomio $x(t)$ basado en el vector de geometría G_{Hx}. De igual forma,

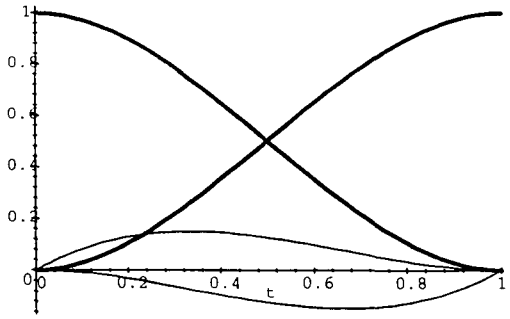
$y(t) = T \cdot M \cdot G_{Hy}$ y $z(t) = T \cdot M \cdot G_{Hz}$. Así, la expresión para el segmento completo puede escribirse como

$$\boxed{Q(t) = \begin{pmatrix} x(t) & y(t) & z(t) \end{pmatrix} = T \cdot M_H \cdot G_H}$$

Expandiendo el producto $T \cdot M_H$ obtenemos las **funciones blending de Hermite** B_H como los polinomios que pesan a cada elemento del vector de geometría:

$$\begin{aligned} Q(t) &= T \cdot M_H \cdot G_H = B_H \cdot G_H = \\ &= (2t^3 - 3t^2 + 1)P_1 + (-2t^3 + 3t^2)P_4 + (t^3 - 2t^2 + t)R_1 + (t^3 - t^2)R_4 \end{aligned}$$

En la figura siguiente pueden verse las cuatro funciones blending



Las líneas gruesas representan a los polinomios que multiplican a P_1 y a P_4 (el de P_1 empieza en 1) y las finas a R_1 y a R_4

Nótese que en $t = 0$, solamente la función que multiplica a P_1 es distinta de cero, por lo que solamente P_1 afecta a la curva en $t = 0$. A medida que t crece, los otros tres polinomios comienzan a tener influencia.

En la figura 7 se ven las cuatro funciones blending ya pesadas por las componentes y de un vector de geometría, su suma $y(t)$, y la curva paramétrica 2D final $Q(t)$. Nótese que el eje de representación de ésta última figura es xy y no ty.

En la figura 8 se ilustra la influencia que sobre la curva ejerce la variación de la magnitud de la tangente R_1. Como comentábamos antes, la tangente R_1 viene a medir la derivada en t del inicio de la curva, es decir, como si fuese su velocidad inicial. A mayor magnitud, mayor velocidad tendrá la curva. Ya que el segmento se mantiene constante en el intervalo $(0, 1)$, es decir, t varía entre 0 y 1 (el tiempo es el mismo), al incrementar la velocidad se incrementa la longitud de la curva. El efecto visual es que la curva se estira en el sentido de la tangente.

En la figura 9 se aprecia el efecto de variar la dirección de la tangente. Lógicamente, la curva partirá en la misma dirección que tenga R_1, e irá variando suavemente hasta llegar a la posición final llegando siempre con una dirección R_4.

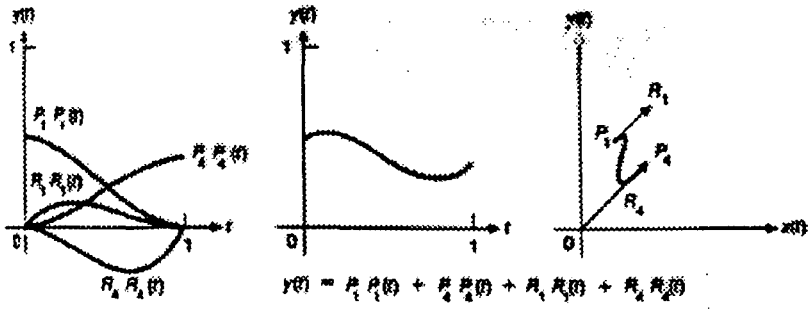


Figure 7

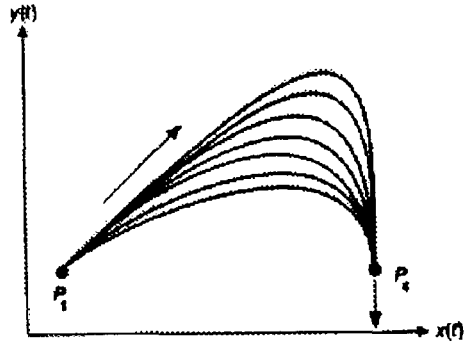


Figure 8

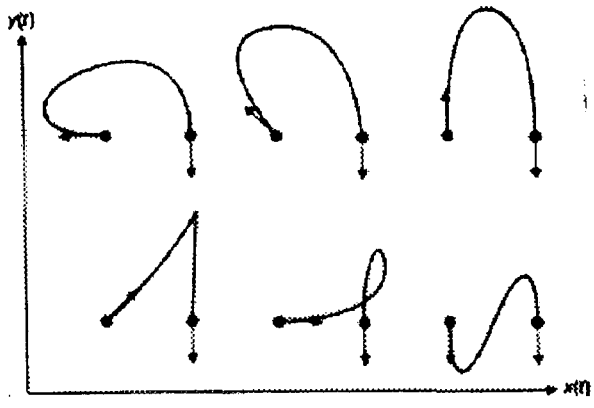


Figure 9

En los sistemas gráficos actuales, el software que permite al diseñador modelar las curvas es capaz de mostrarnos las tangentes de cada segmento para que el usuario interactivamente vaya variando dichos vectores, y en tiempo real ir viendo como va quedando la curva completa. Esta situación se muestra en la figura 10, donde puede verse una curva de Hermite compuesta por dos segmentos. Los puntos extremos pueden reposicionarse arrastrándolos con el ratón, y los vectores tangentes son alargados o encogidos también con el ratón arrastrando las puntas de los vectores.

Figure 10

También puede apreciarse como, para que la curva aparezca suave y sin esquinas, debe mantenerse la continuidad G_1 entre cada dos segmentos consecutivos. De esta manera, si forzamos al software para que nos mantenga siempre esta restricción, las tangentes final e inicial de dos segmentos conectados siempre serán colineales, como puede verse. Matemáticamente, esta restricción implica que los dos vectores de geometría deben ser de la forma

$$\begin{pmatrix} P_1 \\ P_4 \\ R_1 \\ R_4 \end{pmatrix} \text{ y } \begin{pmatrix} P_4 \\ P_7 \\ kR_4 \\ R_7 \end{pmatrix}, \text{ siendo } k > 0$$

Es decir, debe haber un punto común entre ambos segmentos, y los vectores tangentes deben tener igual dirección. La condición más restrictiva de C_1 se consigue haciendo $k = 1$, coincidiendo entonces dirección y magnitud. En la figura 11 pueden verse dos segmentos con continuidad G_1.

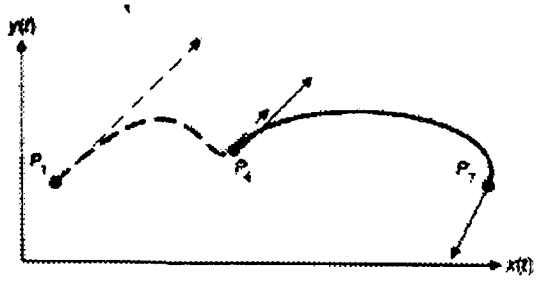


Figure 11

La forma de dibujar una curva de Hermite es relativamente sencillo. Sólo hay que ir evaluando $x(t)$,$y(t)$y $z(t)$para $n$sucesivos valores de $t$separados por un incremento constante δ.Es decir

$$\left.\begin{array}{l} x(t) = a_x t^3 + b_x t^2 + c_x t + d_x \\ y(t) = a_y t^3 + b_y t^2 + c_y t + d_y \\ z(t) = a_z t^3 + b_z t^2 + c_z t + d_z \end{array}\right\} \text{ donde } t = 0, \delta, 2\delta, ..., 1$$

Otra forma más eficiente es usar la regla de Horne para polinomios

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

lo cual reduce el cálculo a 9 multiplicaciones y 10 sumas para cada punto 3D. Una vez calculados todos los puntos, éstos pueden unirse mediante segmentos rectos en la pantalla.

Otro detalle muy importante a tener en cuenta sobre las curvas cúbicas es que, al ser combinaciones lineales de los cuatro elementos del vector de geometría, para transformar una curva simplemente transformaremos su vector de geometría, y usaremos este nuevo vector obtenido para mostrar la curva transformada. Esto significa que **las curvas son invariantes frente a rotaciones, traslaciones y escalados**. Esta estrategia es por tanto muchísimo más eficiente que generar la curva como una serie de segmentos rectos cortos y transformar cada segmento individual.

3.2.2 Curvas de Bezier

Un segmento curvo de Bezier viene determinado por sus dos puntos extremos, P_1y P_4,y por otros dos puntos, P_2y P_3,que indirectamente especifican los vectores tangentes en los extremos. Estos dos puntos no pertenecen a la curva, como puede verse en la figura 12. Los vectores tangentes vienen determinados por los vectores P_1P_2y P_3P_4,y

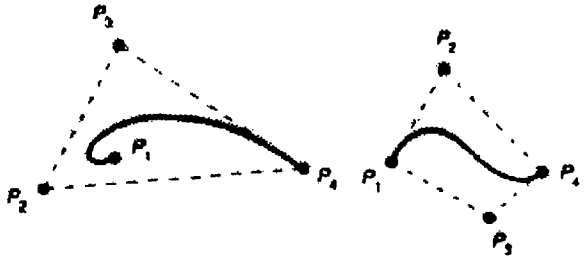


Figure 12

su relación con los vectores tangente R_1y R_4de Hermite es

$$\begin{aligned} R_1 &= Q'(0) = 3(P_2 - P_1) \\ R_4 &= Q'(1) = 3(P_4 - P_3) \end{aligned}$$

Por lo tanto, vemos que en realidad una curva Bezier es como una curva de Hermite, salvo que el vector de geometría es diferente, ya que en vez de consistir en dos puntos y dos vectores, consiste en cuatro puntos, esto es

$$G_B = \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{pmatrix}$$

Para encontrar la relación existente entre el vector de geometría de Hermite, G_H,y el de Bezier, G_B,vamos a calcular la matriz M_{HB}que cumple $G_H = M_{HB} \cdot G_B$:

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

Para encontrar la matriz base de Bezier usaremos la expresión de la curva de Hermite:

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

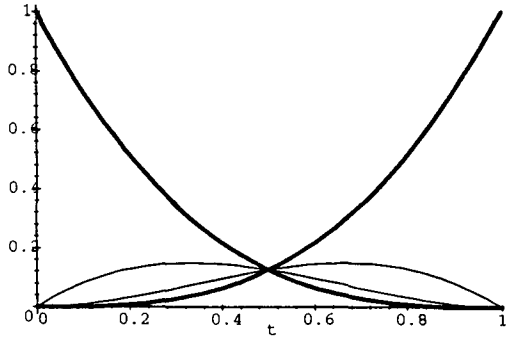
De aquí obtenemos la matriz de Bezier:

$$M_B = M_H \cdot M_{HB} = \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

La expresión para la curva es entonces

$$Q(t) = (1-t)^3 P_1 + 3t(1-t)^2 P_2 + 3t^2(1-t)P_3 + t^3 P_4$$

Los cuatro polinomios $B_B = T \cdot M_B$,que aparecen multiplicando a los elementos del vector de geometría en la ecuación anterior, son conocidos como **polinomios de Bernstein**, y aparecen en la figura siguiente:



Las líneas gruesas representan a los polinomios que multiplican a P_1 y a P_4 y las finas a P_2 y a P_3

Puede verse como en $t = 0$ solamente el polinomio que multiplica a P_1 es distinto de cero, por lo que la curva es igual a P_1. Similarmente, en $t = 1$ sólo el polinomio de P_4 es diferente de cero, estando la curva en P_4.

Si examinamos los cuatro polinomios de Bernstein nos damos cuenta de que su suma vale siempre 1 para todo t entre 0 y 1, y además, ninguno de los cuatro polinomios se hace nunca inferior a cero. De esta manera $Q(t)$ es en realidad una media ponderada de los cuatro puntos de control. Esto significa que cada segmento curvo está completamente contenido en el **convex hull**[3] formado por los cuatro puntos de control.

Esta propiedad garantiza que cualquier punto del segmento curvo estará en el interior de dicho volumen, como resultado de la media ponderada. Esto puede verse claramente en un caso de dos puntos (un segmento recto) o tres puntos (un triángulo). Cualquier suma ponderada de los puntos (siendo uno la suma de los pesos y todos ellos positivos) cae siempre en el interior de la recta o del triángulo respectivamente (véase la figura anterior).

Esta propiedad es muy útil en la fase de recorte: en lugar de pasarle al algoritmo de recorte todos los segmentos de la curva para determinar su visibilidad, es mucho más eficiente aplicar primero el algoritmo a los convex hull de cada uno. Si el convex hull está completamente dentro del volumen de recorte, el segmento de la curva puede dibujarse enteramente. Si el convex hull está completamente fuera del volumen, el segmento será totalmente invisible. Sólo en aquellos casos en los que el convex hull intersecte al volumen de recorte será cuando pasemos la expresión del segmento al algoritmo del recorte. Obviamente es mucho más rápido calcular el recorte de un polígono o poliedro que el de una función cúbica paramétrica.

Al igual que ocurría con las curvas de Hermite, entre dos segmentos consecutivos debe garantizarse al menos la continuidad G_1. Para ello debe cumplirse que

$$P_3 - P_4 = k(P_4 - P_5), \text{ siendo } k > 0$$

Es decir, los tres puntos P_3, P_4 y P_5 deben ser distintos y colineales. Para cumplir C_1 habrá que hacer $k = 1$. En la figura 13 pueden verse dos segmentos de Bezier consecutivos con continuidad G_1. Obsérvese como cada segmento queda siempre en el interior de su convex hull.

3.3.3 Subdivisión de curvas

Supongamos que tenemos creada una serie conectada de segmentos curvos para intentar aproximar una forma geométrica que estamos diseñando. Una vez hechos podemos

[3]**Me niego a traducir esta expresión. Si la curva es en 2D significa "la envolvente convexa del polígono formado por los cuatro puntos de control". ¿Queda claro?. Si la curva es en 3D, entonces sustituir "polígono" por "poliedro".**

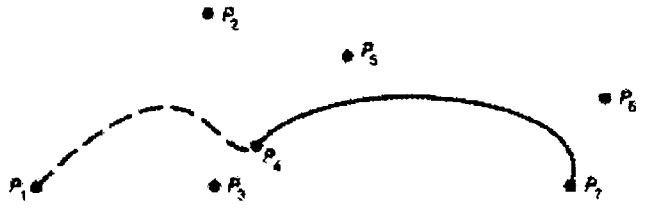


Figure 13

manipular los puntos de control para variar la forma de la curva, pero probablemente no podamos obtener toda la exactitud que quisiéramos. Probablemente, esto será debido a que no hemos creado los suficientes puntos de control para lograr el efecto deseado. Para solucionar este problema existen dos soluciones.

La primera se conoce como **elevación del grado**, es decir, incrementamos el grado del polinomio de 3 a 4 o más. Esta técnicas se hace necesaria a veces, sobre todo si necesitamos garantizar continuidades de grado alto (C^2 ó C^3), pero no es muy conveniente, debido a que se necesitan más condiciones para el segmento (al tener más coeficientes) y al coste computacional adicional para evaluar la curva.

La segunda solución es mucho más práctica, y consiste en subdividir uno o más segmentos de la curva en dos nuevos segmentos. Por ejemplo, un segmento Bezier compuesto por cuatro puntos de control puede subdividirse en dos nuevos segmentos con un total de siete puntos de control (los nuevos segmentos tienen un punto en común). Estos dos nuevos segmentos coinciden perfectamente con el segmento original, hasta que movamos alguno de los puntos de control.

Veamos cómo sería el algoritmo para un segmento Bezier. Sea $Q(t)$ el segmento definido por los puntos P_1, P_2, P_3 y P_4, y queremos encontrar una curva definida por los puntos L_1, L_2, L_3 y L_4 que coincida con la mitad izquierda de la curva original (left) y otra curva definida por los puntos R_1, R_2, R_3 y R_4 que coincida con la mitad derecha (right). Es decir, la curva izquierda ha de coincidir con $Q(t)$ en el intervalo $0 \leq t < 1/2$, y la curva derecha ha de coincidir en el intervalo $1/2 \leq t < 1$.

Para realizar la subdivisión vamos a usar una técnica de construcción geométrica desarrollada por **Casteljau** para evaluar una curva de Bezier para cualquier valor de t. En nuestro caso nos interesa usar $t = 1/2$, aunque más adelante generalizaremos. El algoritmo es el siguiente. Los segmentos rectos P_1P_2, P_2P_3 y P_3P_4 los dividimos por la mitad y nos quedamos con el punto medio de cada uno, los cuales llamaremos L_2, H y R_3 respectivamente. A su vez, volvemos a tomar los nuevos segmentos L_2H y HR_3 y de nuevo nos quedamos con el punto medio para obtener dos nuevos puntos: L_3 y R_2. Por último, este nuevo segmento L_3R_2 lo volvemos a dividir y nos quedamos con su punto medio L_4 (= R_1). El resultado puede verse en la figura 14.

Haciendo $L_1 = P_1$, el segmento Bezier formado por los puntos de control L_1, L_2, L_3

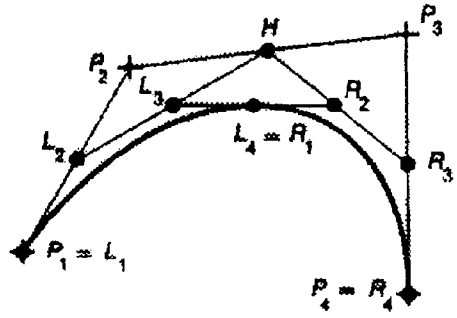


Figure 14

y L_4 coincide perfectamente con la mitad izquierda de $Q(t)$. De forma similar, haciendo $R_4 = P_4$, el otro segmento formado por los puntos R_1, R_2, R_3 y R_4 coincide con la mitad derecha. De esta forma, y retornando al ejemplo inicial, si estamos modelando una forma y nos interesa tener más precisión en un segmento determinado, subdividiríamos la curva con esta técnica y obtendríamos el mismo dibujo inicialmente pero con dos segmentos en lugar de uno, es decir, con más puntos de control. Además esta técnica se presta a la recursividad. Si aún con la subdivisión no logramos la precisión requerida, los segmentos obtenidos pueden ser subdivididos nuevamente aplicando de nuevo la técnica.

Como siempre, lo interesante es reorganizar los cálculos en forma matricial. Nos interesaría calcular por un lado la matriz D_B^L que al mutiplicarla por el vector de geometría del segmento original nos diera el vector de geometría del nuevo segmento izquierdo, y otra matriz D_B^R que nos diera el vector de geometría del derecho. Revisemos la expresiones para los puntos:

$$
\begin{aligned}
&L_1 = P_1; \quad R_4 = P_4 \\
&L_2 = \frac{P_1 + P_2}{2}; \quad H = \frac{P_2 + P_3}{2}; \quad R_3 = \frac{P_3 + P_4}{2} \\
&L_3 = \frac{L_2 + H}{2} = \frac{P_1 + 2P_2 + P_3}{4}; \quad R_2 = \frac{H + R_3}{2} = \frac{P_2 + 2P_3 + P_4}{4} \\
&L_4 = R_1 = \frac{L_3 + R_2}{2} = \frac{P_1 + 3P_2 + 3P_3 + P_4}{8}
\end{aligned}
$$

con lo que ya podemos obtener la expresión matricial:

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

Fíjense que cada uno de los nuevos puntos de control L_i y R_i son una suma ponderada de los puntos P_i, con los pesos todos positivos y cuya suma es siempre 1. Por lo tanto, todos estos nuevos puntos se encuentran en el interior del convex hull. Esto significa que dichos puntos se hallan más cerca de la curva $Q(t)$ que los puntos originales. Esta propiedad nos será muy útil en la siguiente sección.

Por otra parte, el hecho de dividir la curva en $t = 1/2$ suele ser la opción más conveniente en la mayoría de los casos, pero lo ideal sería que el propio usuario de forma interactiva indicara en qué punto de la curva hacer la subdivisión, aprovechando la técnica de Casteljau. Esta técnica ya habíamos comentado que servía para evaluar la curva en un punto t cualquiera, no necesariamente $t = 1/2$. Si quisiéramos calcular cuál es el punto de la curva para un cierto $0 < t_1 < 1$, la técnica trabajaría de forma similar, salvo que en lugar de dividir siempre por la mitad, dividiríamos usando el ratio $t_1 : (1 - t_1)$, es decir, si por ejemplo $t_1 = 1/3$ dividiríamos cada segmento dejando 1/3 a la izquierda y 2/3 a la derecha[4]. El punto final obtenido, L_4, es el punto perteneciente a la curva, que es igual a $Q(t_1)$, y además, se sigue cumpliendo que el segmento formado por los puntos L_i coincide con el primer tercio de $Q(t)$, y el formado por los puntos R_i coincide con los dos últimos tercios de $Q(t)$.

3.3.4 Dibujo de curvas

Hay dos formas básicas para dibujar una curva paramétrica. La primera es por **evaluación iterativa** de $x(t)$, $y(t)$ y $z(t)$ para pequeños incrementos de la variable t, y dibujando líneas rectas entre cada par de valores. Este método ya lo contamos anteriormente, y vimos que el coste computacional necesario era de 9 multiplicaciones y 10 sumas por cada punto 3D que queramos calcular. Una vez calculados, estos puntos se unen mediante segmentos rectos.

Una observación importante es la siguiente: si el número de puntos evaluados, n, es relativamente pequeño, la curva puede verse en pantalla como una sucesión de segmentos rectos cortos, es decir, de forma similar a cuando usamos listas de aristas. Sin embargo, hay una diferencia muy grande con respecto a este otro formato, y es que seguimos manteniendo la información de la curva completa, y en cualquier momento, variando n, podemos tener la curva de forma más exacta, cosa que se hace imposible si lo que tenemos almacenado es un conjunto finito de aristas y puntos.

La segunda manera se conoce como **subdivisión recursiva**, ya que usa la técnica de Casteljau para ir subdividiendo cada segmento hasta que los nuevos segmentos sean lo suficientemente planos que puedan ser aproximados por una recta. El algoritmo general sería el siguiente:

Procedure DibujaCurvaRecursivo (curva, ε)

[4]Atención: éste es un error típico. Para encontrar el punto C que divide un tramo AB con el ratio $1/3 : 2/3$ se calcula haciendo $C = A + \frac{B-A}{3}$, pero nunca de la forma $C = \frac{A+B}{3}$. ¿Está claro?

if TestPlanaridad (curva, ε) then
    DibujaRecta (curva)
else
    SubdivideCurva (curva, Lcurva, Rcurva)
    DibujaCurvaRecursivo (Lcurva, ε)
    DibujaCurvaRecursivo (Rcurva, ε)
end

La representación Bezier es muy apropiada para ser dibujada con este método. La subdivisión es rápida, pues ya vimos en la sección anterior que se necesitaban 6 sumas y 6 divisiones. El test de planaridad para un segmento Bezier es bastante sencillo. Una posible medida de cuan plano es el segmento podría ser considerar la distancia de P_2 y P_3 al segmento P_1P_4, como se ve en la figura 15. Si esta distancia es inferior al umbral ε podemos considerar que el segmento curvo es lo suficientemente plano. Si no es así, seguiríamos subdividiendo, y como comentábamos antes, con cada nueva subdivisión, los nuevos puntos de control obtenidos se encontrarán más cerca de la curva, por lo que los nuevos convex hulls formados van siendo cada vez más pequeños y estrechos.

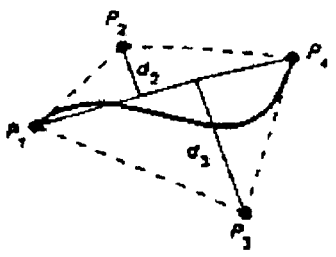


Figure 15

La gran ventaja de la subdivisión recursiva es que evita los cálculos innecesarios en aquellas zonas donde la curva sea más o menos plana. En estos lugares, apenas habrá que subdividir, mientras que en las zonas donde la curvatura sea más elevada se calcularán más puntos para poder visualizarla. En el metodo iterativo, al tener que elegir un incremento de t constante, deberíamos coger un incremento pequeño para poder mostrar las zonas de alta curvatura, pero este incremento es usado también en las zonas planas perdiendo mucho tiempo de cálculo innecesario. Por otro lado, la desventaja es que el cálculo para el test de planaridad es cómputo extra que no teníamos en el primer método.

3.3 SUPERFICIES BICÚBICAS PARAMÉTRICAS

Las superficies bicúbicas paramétricas son una generalización de las curvas cúbicas paramétricas. Recordemos que la expresión general de una curva era $Q(t) = T \cdot M \cdot G$. donde G era el vector de geometría. Este vector representaba los valores geométricos

de la curva (por ejemplo, las coordenadas de los puntos extremos y de los vectores tangente) y por lo tanto era un vector constante independiente del parámetro t.

Para realizar el paso a superficies renombremos la variable t por s, es decir, $Q(s) = S \cdot M \cdot G$. Una vez hecho esto, el truco consiste en considerar que ahora el vector G ya no es constante, sino que varía en 3D a lo largo de una cierta trayectoria parametrizada por t. Es decir, si antes tenía un punto P_1 que indicaba el comienzo de la curva, ahora tendré $P_1(t)$, que me va a indicar el comienzo de la superficie. La expresión resultante es:

$$Q(s,t) = S \cdot M \cdot G(t) = S \cdot M \cdot \begin{pmatrix} G_1(t) \\ G_2(t) \\ G_3(t) \\ G_4(t) \end{pmatrix}$$

Ahora, para un cierto t_1 fijo, $Q(s,t_1)$ es una curva, ya que $G(t_1)$ es constante. Haciendo a continuación $t = t_2$, siendo $t_2 - t_1$ un valor muy pequeño, $Q(s,t_2)$ será una nueva curva ligeramente distinta de la anterior. Repitiendo este proceso para muchos valores distintos de t entre 0 y 1 obtendremos una familia completa de curvas muy pegadas entre sí. El conjunto de todas esas curvas definirá una superficie. Si $G(t)$ son curvas cúbicas, diremos que la superficie es bicúbica (dos variables, tercer grado) paramétrica.

La forma general de la curva vendrá separada en sus tres componentes:

$$\begin{cases} x(s,t) = S \cdot M \cdot G_x(t) \\ y(s,t) = S \cdot M \cdot G_y(t) \\ z(s,t) = S \cdot M \cdot G_z(t) \end{cases}$$

Al igual que ocurría con las curvas, en donde existían varios tipos dependiendo de la matriz base, aquí también existirán otros tantos tipos de superficies distintas. Veamos las de Hermite y las de Bezier.

3.3.1 Superficies de Hermite

Para derivar la expresión para las superficies de Hermite partiremos de la expresión de su curva. Voy a desarrollarlo sólo para la componente x. Las otras dos componentes se realizarían de la misma manera. Como hacíamos antes, renombremos la variable t por s y hagamos que el vector de geometría de Hermite G_H deje de ser constante y pase a depender del parámetro t :

$$x(s,t) = S \cdot M_H \cdot G_{Hx}(t) = S \cdot M_H \cdot \begin{pmatrix} P_{1x}(t) \\ P_{4x}(t) \\ R_{1x}(t) \\ R_{4x}(t) \end{pmatrix}$$

Las funciones $P_{1x}(t)$ y $P_{4x}(t)$ definen las componentes x de los puntos extremos de la curva de parámetro s. Similarmente, $R_{1x}(t)$ y $R_{4x}(t)$ son los vectores tangentes a esos puntos. Para cualquier valor de t, existen dos puntos extremos fijos y dos vectores tangentes concretos. En la figura 16 se muestran las funciones $P_1(t)$ y $P_4(t)$, y las curvas cúbicas en s definidas para los valores de $t = 0.0, 0.2, 0.4, 0.6, 0.8$ y 1.0.

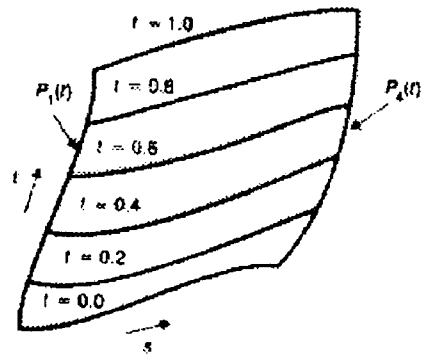


Figure 16

Desarrollemos ahora las expresiones para cada uno de los elementos de $G_{Hx}(t)$:

$$P_{1x}(t) = T \cdot M_H \cdot \begin{pmatrix} g_{11x} \\ g_{12x} \\ g_{13x} \\ g_{14x} \end{pmatrix}; \quad P_{4x}(t) = T \cdot M_H \cdot \begin{pmatrix} g_{21x} \\ g_{22x} \\ g_{23x} \\ g_{24x} \end{pmatrix}$$

$$R_{1x}(t) = T \cdot M_H \cdot \begin{pmatrix} g_{31x} \\ g_{32x} \\ g_{33x} \\ g_{34x} \end{pmatrix}; \quad R_{4x}(t) = T \cdot M_H \cdot \begin{pmatrix} g_{41x} \\ g_{42x} \\ g_{43x} \\ g_{44x} \end{pmatrix}$$

donde g_{1ix} y g_{2ix} son las componentes x de los elementos i-ésimo del vector de geometría para la curvas $P_1(t)$ y $P_4(t)$ respectivamente, y g_{3ix} y g_{4ix} son las componentes x de los elementos i-ésimo del vector de geometría para la curvas $R_1(t)$ y $R_4(t)$. Es decir, g_{23} por ejemplo será la componente x del vector tangente inicial para la curva $P_4(t)$.

Fíjense bien además en un detalle muy importante: para un cierto t_1 fijo, $P_i(t_1)$ representan puntos concretos, mientras que $R_i(t_1)$ representan vectores. Por lo tanto, evaluando para muchos valores de t, $P_i(t)$ puede verse como una sucesión de puntos pegados entre sí formando una curva, mientras que $R_i(t)$ sería como una familia de vectores que van variando suavemente partiendo de $R_i(0)$ hasta llegar a $R_i(1)$.

Reescribiendo las cuatro expresiones anteriores en una única ecuación obtenemos:

$$\begin{pmatrix} P_{1x}(t) & P_{4x}(t) & R_{1x}(t) & R_{4x}(t) \end{pmatrix} = T \cdot M_H \cdot \mathbf{G}_{Hx}^T$$

siendo

$$\mathbf{G}_{Hx} = \begin{pmatrix} g_{11x} & g_{12x} & g_{13x} & g_{14x} \\ g_{21x} & g_{22x} & g_{23x} & g_{24x} \\ g_{31x} & g_{32x} & g_{33x} & g_{34x} \\ g_{41x} & g_{42x} & g_{43x} & g_{44x} \end{pmatrix}$$

Ahora, realizamos la traspuesta en ambos lados y nos queda:

$$\begin{pmatrix} P_{1x}(t) \\ P_{4x}(t) \\ R_{1x}(t) \\ R_{4x}(t) \end{pmatrix} = \begin{pmatrix} g_{11x} & g_{12x} & g_{13x} & g_{14x} \\ g_{21x} & g_{22x} & g_{23x} & g_{24x} \\ g_{31x} & g_{32x} & g_{33x} & g_{34x} \\ g_{41x} & g_{42x} & g_{43x} & g_{44x} \end{pmatrix} \cdot M_H^T \cdot T^T = \mathbf{G}_{Hx} \cdot M_H^T \cdot T^T$$

Por último sustituimos esta expresión en la que teníamos al comienzo para $x(s,t)$ ($y(s,t)$ y $z(s,t)$ se obtienen de forma similar):

$$\boxed{\begin{cases} x(s,t) = S \cdot M_H \cdot \mathbf{G}_{Hx} \cdot M_H^T \cdot T^T \\ y(s,t) = S \cdot M_H \cdot \mathbf{G}_{Hy} \cdot M_H^T \cdot T^T \\ z(s,t) = S \cdot M_H \cdot \mathbf{G}_{Hz} \cdot M_H^T \cdot T^T \end{cases}}$$

Las tres matrices 4x4 $\mathbf{G}_{Hx}$, $\mathbf{G}_{Hy}$ y $\mathbf{G}_{Hz}$ juegan el mismo papel en las superficies de Hermite que la matriz G_H para las curvas[5]. El significado de los 16 elementos de $\mathbf{G}_H$ puede entenderse en función del desarrollo que hemos hecho. El elemento g_{11x} es en realidad $x(0,0)$, ya que es el punto inicial para $P_{1x}(t)$, que a su vez es el punto inicial para $x(s,0)$. Similarmente, g_{12x} es $x(0,1)$ ya que es el punto final de $P_{1x}(t)$, que a su vez es el punto inicial para $x(s,1)$. Por otro lado, g_{13x} es $\partial x/\partial t(0,0)$ ya que representa el vector inicial para $P_{1x}(t)$, y g_{33x} es $\partial^2 x/\partial s\partial t(0,0)$ ya que es el vector inicial de $R_{1x}(t)$, que a su vez representa la pendiente inicial de $x(s,0)$.

Usando estas interpretaciones, podemos reescribir la matriz $\mathbf{G}_{Hx}$ de esta manera:

$$\mathbf{G}_{Hx} = \begin{pmatrix} x(0,0) & x(0,1) & \frac{\partial}{\partial t}x(0,0) & \frac{\partial}{\partial t}x(0,1) \\ x(1,0) & x(1,1) & \frac{\partial}{\partial t}x(1,0) & \frac{\partial}{\partial t}x(1,1) \\ \frac{\partial}{\partial s}x(0,0) & \frac{\partial}{\partial s}x(0,1) & \frac{\partial^2}{\partial s\partial t}x(0,0) & \frac{\partial^2}{\partial s\partial t}x(0,1) \\ \frac{\partial}{\partial s}x(1,0) & \frac{\partial}{\partial s}x(1,1) & \frac{\partial^2}{\partial s\partial t}x(1,0) & \frac{\partial^2}{\partial s\partial t}x(1,1) \end{pmatrix}$$

Dividamos la matriz en cuatro submatrices de 2x2. La submatriz superior izquierda de $\mathbf{G}_{Hx}$ contiene las coordenadas x de las cuatro esquinas de la superficie. Las submatrices superior derecha e inferior izquierda indican las componentes x de los vectores

[5]Cuando digo matrices de 4x4 me refiero a que cada elemento de la matriz es un punto 3D. En términos de números reales, la resolución de las matrices sería de 4x4x3 (48 números).

tangentes a lo largo de cada dirección paramétrica de la superficie. La submatriz inferior derecha muestra las derivadas parciales cruzadas en las cuatro esquinas. Estas cuatro derivadas son llamadas twists, ya que su valor mide, por así decirlo, el grado de ondulación de la esquina. En la figura 17 se indican los 16 elementos de la matriz en relación con la superficie que definen.

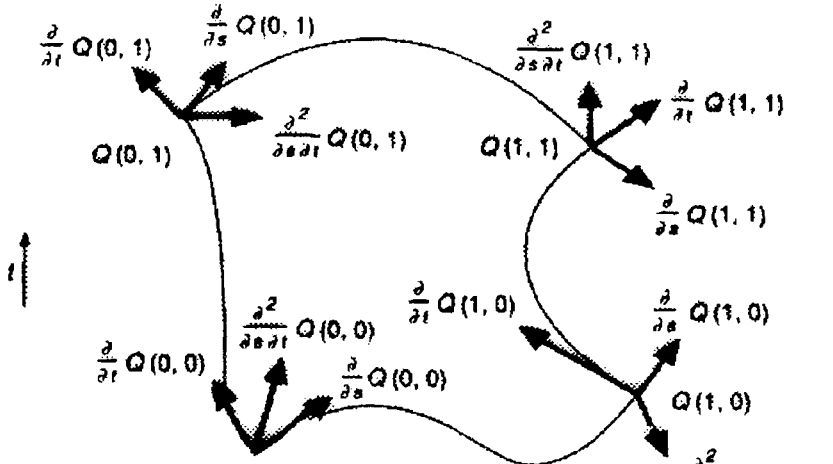


Figure 17

Dejemos a un lado la matriz $\mathbf{G}_H$ y recordemos de nuevo la expresión de un segmento curvo de Hermite. Una curva de Hermite era en realidad una sucesión de segmentos conectados entre sí para formar la curva completa. La expresión $Q(t) = T \cdot M \cdot G$ para todo $t \in [0,1]$ era en realidad la expresión de cada segmento, y para cada uno existía un vector G específico. Ahora retornemos a la superficie. A la expresión que acabamos de obtener $Q(s,t) = S \cdot M \cdot \mathbf{G} \cdot M^T \cdot T^T$ para todo $s,t \in [0,1]$ la llamaremos **patch**, y llamaremos superficie de Hermite a una colección de patches conectados entre sí. Cada uno de estos patches tendrá su matriz $\mathbf{G}$ específica.

Además, al igual que en las curvas había que garantizar la continuidad G^1 o C^1 entre segmentos consecutivos, de la misma forma hay que garantizarla entre los patches conectados. Aunque hay algunas diferencias. Antes, cada segmento mantenía la continuidad en los dos puntos extremos que compartía con sus dos segmentos vecinos. Ahora, cada patch debe mantener la continuidad en sus cuatro bordes que a su vez son compartidos por sus cuatro patches vecinos.

Así, mantener la continuidad C^0 entre dos patches significa que ambos están pegados perfectamente por uno de sus bordes, es decir, el borde final de un patch y el borde inicial del vecino son idénticos. Esto además implica que los puntos de control para los dos patches que identifican el borde deben coincidir. Para mantener la continuidad C^1

es preciso que los puntos de control tanto de la curva que identifica el borde en ambos patches, como los que identifican los vectores tangentes perpendiculares a lo largo de ese borde sean idénticos.

Por ejemplo, supongamos que el patch A está conectado en $s = 1$ con el patch B en $s = 0$, como muestra la figura 18. Para garantizar la continuidad G^1 entre ambos

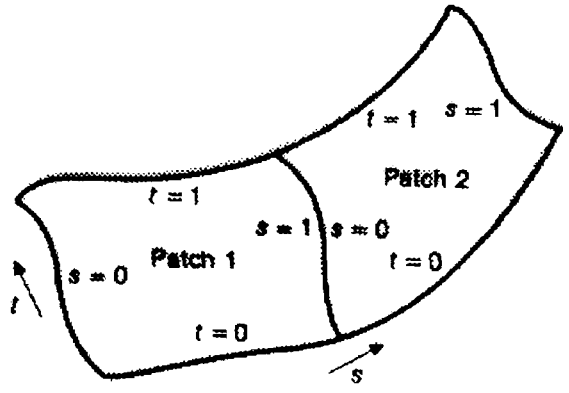


Figure 18

sus matrices deben ser de esta forma:

$$A: \begin{pmatrix} - & - & - & - \\ g_{21} & g_{22} & g_{23} & g_{24} \\ - & - & - & - \\ g_{41} & g_{42} & g_{43} & g_{44} \end{pmatrix}; \quad B: \begin{pmatrix} g_{21} & g_{22} & g_{23} & g_{24} \\ - & - & - & - \\ kg_{41} & kg_{42} & kg_{43} & kg_{44} \\ - & - & - & - \end{pmatrix}$$

siendo $k > 0$. Por supuesto, con $k = 1$ garantizaríamos C^1. En una superficie completa, cada patch deberá cumplir esta restricción con cada uno de sus cuatro vecinos.

3.3.2 Superficies de Bezier

La expresión para las superficies de Bezier se derivan exactamente de la misma manera que hicimos para las de Hermite. El resultado es:

$$\begin{cases} x(s,t) = S \cdot M_B \cdot \mathbf{G}_{Bx} \cdot M_B^T \cdot T^T \\ y(s,t) = S \cdot M_B \cdot \mathbf{G}_{By} \cdot M_B^T \cdot T^T \\ z(s,t) = S \cdot M_B \cdot \mathbf{G}_{Bz} \cdot M_B^T \cdot T^T \end{cases}$$

La matriz de geometría de Bezier, $\mathbf{G}_B$, a diferencia de la de Hermite, consiste de 16 puntos de control, como puede verse en la figura 19.

Las superficies de Bezier son muy usadas para el diseño interactivo por las mismas que eran utilizadas las curvas: tenemos un control preciso del patch moviendo los

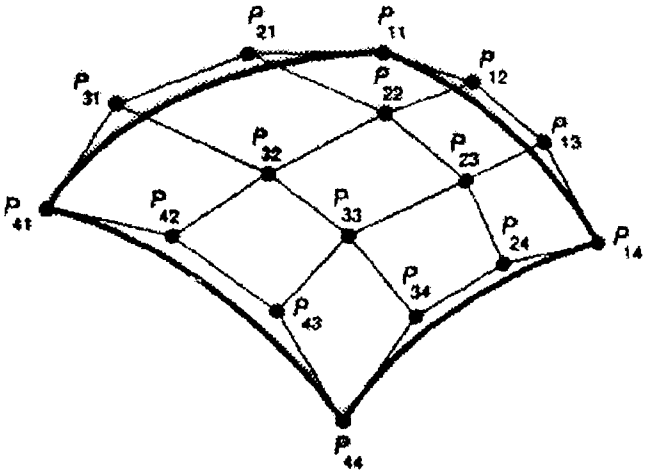


Figure 19

puntos de control, se sigue cumpliendo la propiedad de estar incluido dentro de su convex-hull, y la subdivisión en patches más pequeños se hace de forma sencilla. En la figura 20 puede apreciarse el efecto que obtenemos en el patch cuando estiramos uno de sus puntos de control.

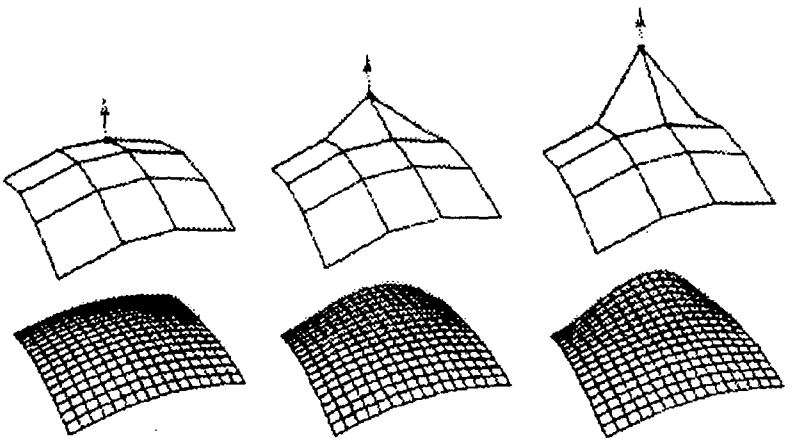


Figure 20

La continuidad C^0 entre dos patches vecinos se consigue haciendo coincidir los cuatro puntos de control que tienen en común. La continuidad G^1 ocurre cuando los dos conjuntos de puntos de control a cada lado del borde común entre ambos son colineales con los puntos del borde. En el ejemplo mostrado en la figura 21 se ven dos patches con continuidad G^1, y se aprecia cómo los segmentos $P_{13}P_{14}P_{15}$, $P_{23}P_{24}P_{25}$, $P_{33}P_{34}P_{35}$ y $P_{43}P_{44}P_{45}$ son todos rectos, y además tienen el mismo ratio k entre sus dos

partes, es decir, la distancia entre P_{43} y P_{44} es k veces la que existe entre P_{44} y P_{45}. Obviamente, la continuidad C^1 ocurrirá para $k = 1$.

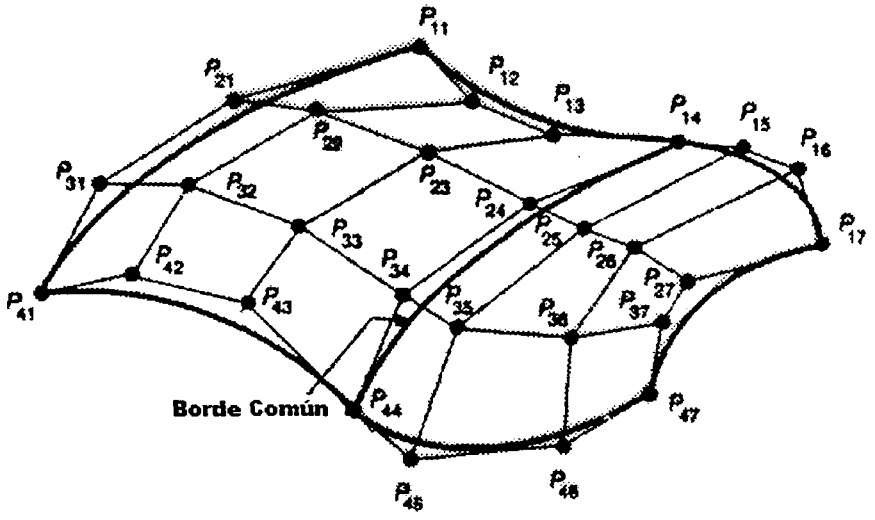


Figure 21

3.3.3 Cálculo de normales a una superficie

El cálculo de normales a una superficie, sea ésta del tipo que sea, es siempre necesario para calcular la iluminación del objeto, para encontrar los objetos más cercanos, para calcular reflexiones, etc. En el caso de las superficies bicúbicas paramétricas es bien sencillo calcularlas.

Comenzemos calculando el vector tangente en la dirección s de la superficie $Q(s,t)$:

$$\begin{aligned}\frac{\partial}{\partial s}Q(s,t) &= \frac{\partial}{\partial s}(S\cdot M\cdot \mathbf{G}\cdot M^T\cdot T^T) = \frac{\partial}{\partial s}(S)\cdot M\cdot \mathbf{G}\cdot M^T\cdot T^T = \\ &= \begin{pmatrix} 3s^2 & 2s & 1 & 0\end{pmatrix}\cdot M\cdot \mathbf{G}\cdot M^T\cdot T^T\end{aligned}$$

De igula manera calculamos el vector tangente en la dirección t :

$$\begin{aligned}\frac{\partial}{\partial t}Q(s,t) &= \frac{\partial}{\partial t}(S\cdot M\cdot \mathbf{G}\cdot M^T\cdot T^T) = S\cdot M\cdot \mathbf{G}\cdot M^T\cdot \frac{\partial}{\partial t}T^T = \\ &= S\cdot M\cdot \mathbf{G}\cdot M^T\cdot \begin{pmatrix} 3t^2 & 2t & 1 & 0\end{pmatrix}^T\end{aligned}$$

Recordemos que $Q(s,t)$ representa al punto (s,t) de la superficie Q, y que es por lo tanto una terna de valores reales (sus coordenadas x, y, z). Por lo tanto, estos vectores también nos salen como ternas de valores (las componentes x, y, z del vector).

Llamemos entonces x_s, y_s, z_s a las tres componentes del vector tangente en s, y x_t, y_t, z_t a las del vector tangente en t. Ambos vectores tangente son paralelos a la superficie en el punto (s,t), y por lo tanto, su producto vectorial es perpendicular a la superficie. La expresión para la normal en dicho punto es por tanto (figura 22):

$$\boxed{N(s,t) = \frac{\partial}{\partial s}Q(s,t) \times \frac{\partial}{\partial t}Q(s,t)} = \begin{pmatrix} y_s z_t - y_t z_s & z_s x_t - z_t x_s & x_s y_t - x_t y_s \end{pmatrix}$$

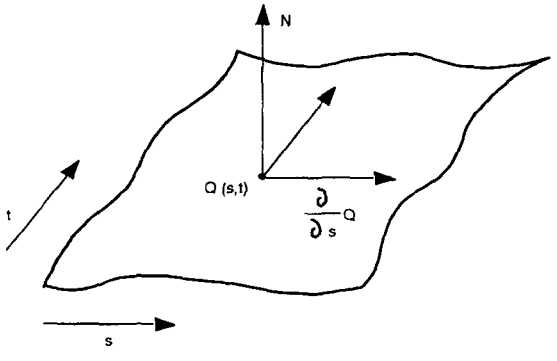


Figure 22

3.3.4 Dibujo de superficies

Al igual que ocurría para dibujar curvas, las superficies pueden dibujarse por evaluación iterativa de los polinomios bicúbicos o por subdivisión, la cual es esencialmente una evaluación adaptativa (dependiendo de la planaridad del patch) de los polinomios.

La evaluación iterativa consiste en dibujar varias curvas a lo largo de t para valores de s concretos, y luego dibujar varias curvas a lo largo s para valores de t concretos. Cada una de estas curvas es una cúbica por lo que aprovecha el algoritmo de la sección anterior. El resultado puede verse en la figura 23.

El algoritmo sería el siguiente:

Procedure DibujaPatch ()
δ=1/n "incremento para dibujar cada curva"
δ_s=1/(n_s-1) "incremento en s entre cada curva en t"
δ_t=1/(n_t-1) "incremento en t entre cada curva en s"
For s=0 to 1 by δ_s do "dibujamos una curva para cada s"
    For t=0 to n-δ by δ do "pintamos la curva 0<t<1"
        DibujaRecta (t,t+δ)

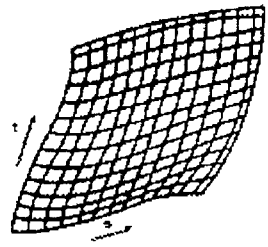


Figure 23

For t=0 to 1 by δ_t do "dibujamos una curva para cada t"
For s=0 to n-δ by δ do "pintamos la curva 0<s<1"
DibujaRecta (s,s+δ)

Este algoritmo es aún más costoso que para las curvas, ya que la ecuación de la superficie debe ser evaluada $2n^2$ veces.

El segundo método es una extensión del procedimiento para las curvas que vimos en la sección anterior. Los patches son subdivididos en nuevos patches hastas que éstos sean similares a cuadriláteros cuasi planos, los cuales se dibujan como polígonos planos. Para evaluar el test de planaridad, una posible solución es hallar el plano que pasa por tres de las cuatro esquinas del patch y encontrar la distancia de cada uno de los 13 puntos de control restantes a dicho plano. Si la mayor de esas distancias está por debajo de un cierto umbral ε entonces el patch es lo suficientemente plano. El algoritmo se muestra a continuación:

Procedure DibujaPatchRecursivo (patch, ε)
If TestPlanaridad (patch, ε) then
DibujaCuadrilátero (patch)
else
SubdividePatch (patch, patch11, patch12, patch21, patch22)
DibujaPatchRecursivo (patch11, ε)
DibujaPatchRecursivo (patch12, ε)
DibujaPatchRecursivo (patch21, ε)
DibujaPatchRecursivo (patch22, ε)
end

La subdivisión del patch se realiza partiendo por la mitad a lo largo de un parámetro, por ejemplo s, y entonces subdividir cada uno de los dos patches resultantes por la mitad a lo largo del otro parámetro t. Para ello partimos de los 16 puntos de control iniciales del patch, y aplicamos el método de subdivisión de curvas para cada conjunto de cuatro puntos de control en la dirección del parámetro s, con lo que obtenemos un conjunto de 7x4 puntos (ya que el método de subdivisión de curvas me pasaba de un segmento de 4 puntos de control a 2 nuevos segmentos de 4 puntos con uno en común = 7 puntos). A continuación paso a subdividir los conjuntos de cuatro puntos de control en la dirección

t, obteniendo finalmente un conjunto de 7x7 puntos. Estos 49 puntos, cogidos en cuatro grupos de 16 (los puntos del medio son compartidos) forman los cuatro nuevos patches. Este proceso puede verse en la figura 24.

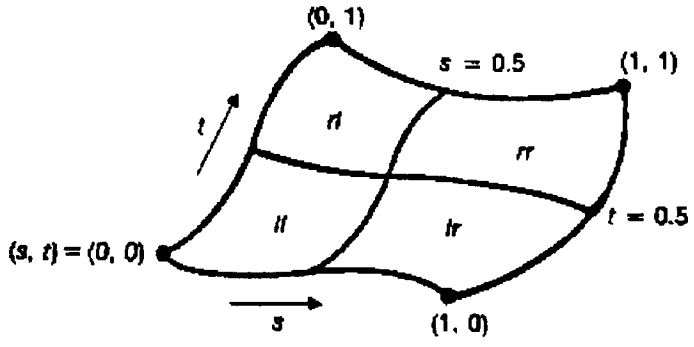


Figure 24

El procedimiento para dibujar el cuadrilátero debe pintarlo como un polígono sombreado. El problema es si el valor ε no es lo suficientemente pequeño, porque en ese caso los cuatro esquinas del patch no serían coplanares. La mejor forma de hacerlo para evitar problemas es calcular el punto promedio de las esquinas y usarlo para pintar cuatro triángulos, tal y como se muestra en la figura 25. La ventaja de los triángulos es que éstos sí que son siempre planos.

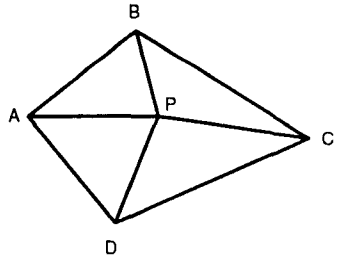


Figure 25

3.4 SUPERFICIES CUÁDRICAS

Las superficies cuádricas naturales (esferas, conos, cilindros, paraboloides, etc.) juegan un importante papel en la fabricación de partes mecánicas, así como en la descripción de superficies manufacturadas. Por ejemplo, los balones de fútbol y baloncesto son esféricos, los embudos son cónicos, las latas de cerveza son cilíndricas, y las antenas

parabólicas no lo son. Las cuádricas son también importantes en la descripción de superficies más complejas. En general, siempre que dispongamos de la expresión analítica de la superficie, lo mejor es usar una superficie cuádrica.

La familia de superficies cuádricas viene definida por las funciones de la forma

$$f(x,y,z) = ax^2 + by^2 + cz^2 + 2dxy + 2eyz + 2fxz + 2gx + 2hy + 2jz + k = 0$$

Por ejemplo, si $a = b = c = -k = 1$ y los restantes coeficientes son cero, tendríamos la esfera unidad definida en el origen; si $a = ... = f = 0$, la ecuación representa un plano; y así muchos ejemplos más. Todos ellos diremos que pertenecen a la familia de las superficies cuádricas.

Aparte de ser usadas en la mayoría de los procesos de fabricación de piezas se usan también en aplicaciones especializadas como el modelado molecular. Entre sus ventajas cabe destacar lo fácil que es cálcular la normal, testear de si un punto pertenece o no a la superficie, calcular la componente z para una x e y fija (muy importante para el cálculo de las superficies ocultas) y calcular intersecciones entre varias superficies.

Como siempre, vamos a intentar expresar la ecuación anterior en forma matricial:

$$P^T \cdot Q \cdot P = 0$$

$$\text{siendo } Q = \begin{pmatrix} a & d & f & g \\ d & b & e & h \\ f & e & c & j \\ g & h & j & k \end{pmatrix} \text{ y } P = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

La ventaja de tenerla así expresada es que podemos aplicarle a Q cualquier transformación afín y obtener la expresión de la cuádrica transformada. Sea M la matriz de dicha transformación, la nueva cuádrica transformada nos quedaría:

$$\boxed{Q' = (M^{-1})^T \cdot Q \cdot M^{-1}}$$

La expresión para el vector normal a la superficie cuádrica $f(x,y,z) = 0$ en cualquier punto será

$$\boxed{N(x,y,z) = \begin{pmatrix} \frac{df}{dx} & \frac{df}{dy} & \frac{df}{dz} \end{pmatrix}}$$

Esto es mucho más fácil que en el caso de las superficies bicúbicas, ya que aquí podemos calcular la expresión general para la normal, cosa que no podíamos hacer con éstas, sino sólamente calcular para puntos fijos.

Chapter 4 REPRESENTACIÓN DE SÓLIDOS

Al igual que un conjunto de líneas y curvas 2D no tiene necesariamente que describir un área cerrada, un conjunto de planos y superficies 3D no tiene por qué representar un volumen cerrado. Sin embargo en muchas aplicaciones es importante considerar a un objeto como un volumen perfectamente cerrado, y poder distinguir entre interior, exterior y superficie del objeto 3D. Así por ejemplo, en las aplicaciones de CAD/CAM en donde disponemos de un objeto modelado según su descripción geométrica, podemos calcular infinidad de operaciones sobre él antes de haberlo construido físicamente: puede determinarse si un objeto se solapa o intersecta con otro (un brazo robot puede esquivar los objetos del entorno), calcular volúmenes y centros de masa (en el diseño de un coche, éste debe ser lo más aerodinámico posible), calcular la resistencia a la presión o a la temperatura (mediante el método de elementos finitos), etc. Todas estas aplicaciones son ejemplos de **modelado de sólidos**.

En el mundo real todo está construido por átomos, los cuales se juntan formando moléculas, y así sucesivamente hasta formar objetos tales como manzanas, coches o este libro. Lo ideal sería que en el ordenador pudiera disponer igualmente de un elemento mínimo similar al átomo con el que construir todos los objetos posibles. Desafortunadamente esto no es así. La memoria del ordenador sólamente puede almacenar elementos binarios que representan números o símbolos, los cuales pueden emplearse para representar coordenadas, ecuaciones, tangentes, propiedades físicas, etc. Por lo tanto puede afirmase que no existe un esquema de modelado universal que permita construir todos los objetos presentes en la realidad (ni tampoco los abstractos).

Por ejemplo, un simple cubo se compone de 8 vértices, 12 aristas y 6 caras. Quizás la mejor forma de representación sería almacenar las coordenadas cartesianas de los 8 vértices, con los cuales podríamos reconstruir las aristas y caras. Pero ¿podríamos usar la misma técnica para representar una esfera? En realidad sí, pero deberíamos usar un enorme conjunto de vértices para poder representarla lo más aproximada posible. Una mejor técnica de representación sería usar su ecuación $x^2 + y^2 + z^2 < r^2$, donde incluso podríamos saber qué puntos pertenecen al interior, al exterior o a la superficie de la esfera. Por otro lado, ¿qué hay de objetos más complejos como el fuego, una planta, o una nube? La geometría convencional no puede con ellos, y se necesitan técnicas más complejas como sistemas de partículas o fractales.

Finalmente, consideremos el modelo de la famosa tetera de Utah, el cual tiene un cuerpo, una tapa, un asa y un caño por donde saldría el té. Pero si examinamos el interior de su geometría descubrimos que ¡no tiene agujero! Sin embargo, la pregunta es ¿realmente importa? Si estamos trabajando con una herramienta CAD y el objetivo final es construir la tetera, por supuesto que sí. Seguramente deberíamos poder calcular su peso, su centro de gravedad, el área de su superficie y su momento de inercia, por

lo que es esencial disponer de una descripción lo más exacta posible. Pero si por el contrario estamos desarrollando una película de dibujos y la tetera es un elemento más del escenario, entonces no nos interesaría tanta exactitud. Resumientdo, esquemas de representación hay muchos, y cuál elijamos dependerá de la aplicación con la que trabajemos y del tipo de objeto.

4.1 MODELOS DE ALAMBRE

El método más simple de construir un objeto 3D es hacerlo a partir de una colección de líneas que representen los bordes rectos que identifiquen la geometría del objeto en cuestión. Tal método se conoce como modelo de alambre (**wire-frame**) ya que al dibujarlo en pantalla parece que está construido por trozos rectos de alambre. Por ejemplo, la representación de una mesa sería algo parecido a la figura 1:

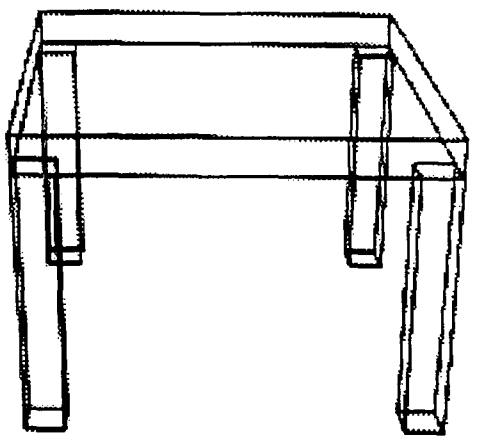


Figure 1

Debido a que la única información de que disponemos del objeto es el conjunto de aristas, es imposible para los programas eliminar las líneas que quedan ocultas por alguna superficie, cosa que ocurre en la realidad. Esto es porque no disponemos de datos referentes a las caras del objeto, por lo que la imagen resultante puede considerarse como una vista transparente del objeto. Esta propiedad puede ser útil en algunas ocasiones. Por ejemplo, en las aplicaciones de diseño arquitectónico, como AutoCad, cuando se tienen muchos objetos en pantalla simultáneamente se hace muy lento trabajar debido a la complejidad de cada objeto, por lo que en la fase de interacción con el usuario, la vista en la pantalla muestra a todos los objetos en wire-frame, lo cual es mucho más rápido, y sólo en la imagen final es cuando mostramos los objetos completamente.

Si intentáramos mejorar la estructura de datos añadiendo información de cómo las aristas están conectadas para formar superficies, estaríamos hablando ya de otros modelos más completos: los modelos poligonales.

4.2 REPRESENTACIÓN POLIGONAL

Es la representación clásica para los gráficos 3D. Un objeto viene representado por un mesh de polígonos, como los que se veían en el capítulo anterior. En un caso general, aquellas caras del objeto que no sean planas sino superficies curvas serán aproximadas también como nuevos meshes, como se ve en la figura 2.

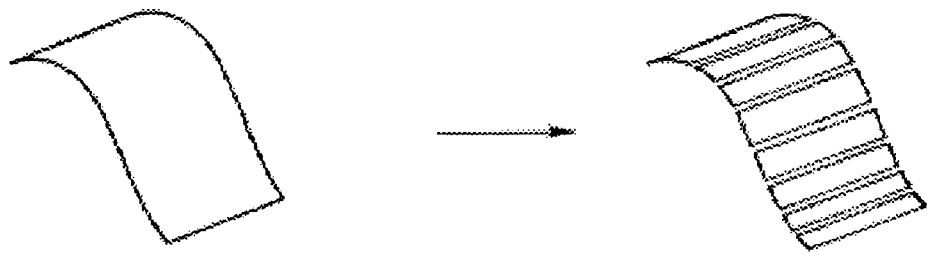


Figure 2

Este tipo de representación también recibe el nombre de **representación de fronteras** (boundary representation o **B-rep**) ya que es en realidad una descripción topológica y geométrica de la frontera o superficie del objeto. Es un esquema muy utilizado en Informática Gráfica, entre otras razones porque el modelado de objetos poligonales es simple de realizar. Sin embargo existen ciertas dificultades prácticas. La exactitud del modelo, es decir, la diferencia entre la representación facetada y la superficie real curva del objeto suele ser arbitraria. Para lograr una buena calidad en la imagen, el tamaño de los polígonos individuales debe depender de la curvatura espacial local. Donde la curvatura cambie rápidamente se necesitarán más polígonos por unidad de área de la superficie. Este factor debe ser tenido en cuenta a la hora de construir los polígonos.

La representación poligonal no sólamente se usa como estructura de datos al modelar, sno también como forma intermedia de otros muchas estructuras más complejas. Esto es así debido a que esta representación ha sido la técnica tradicional durante muchos años, y la mayoría de los algoritmos desarrollados en el campo de los gráficos trabajan con esta representación. Por ejemplo, los algoritmos de iluminación de polígonos han sido tan optimizados que algunos tarjetas gráficas de ordenador ya los llevan implementados en hardware. Por lo tanto, si disponemos de un objeto definido mediante patches bicúbicos como los que veíamos en el capítulo anterior, una buena estrategia de visualización sería convertirlos a mallas poligonales y mandarlos directamente al hardware.

En el caso más sencillo, un mesh poligonal es una estructura de datos que consiste en una lista de polígonos representados por las coordenadas (x, y, z) de sus vértices. Ya vimos en el capítulo anterior los distintos subtipos de representación posibles. Además de esta información podemos almacenar como parte de la representación del objeto otra

información geométrica que pueda ser usada en procesos de cálculo posteriores. Por ejemplo, ya hemos dicho que para muchos procesos es necesario conocer las normales a un punto de la superficie. Si las calculásemos todas inicialmente y las almacenáramos en la propia estructura de datos, los cálculos posteriores serían más rápido. Por supuesto, la contrapartida es que nuestro modelo requeriría más espacio de almacenamiento.

Otra idea conveniente puede ser ordenar los poligonos en una estructura jerárquica, como en la figura 3, en donde los polígonos se agrupan en superficies, y éstas en objetos. De esta manera, un cilindro posee tres superficies: dos caras planas arriba y abajo y una superficie curva. La razón para esta agrupación es que así podemos distinguir entre aristas que son parte de la aproximación (las aristas entre rectángulos adyacentes sobre la superficie curva del cilindro) y aristas que existen en la realidad. Esto puede ser muy útil por ejemplo para los algoritmos de iluminación a la hora de sombrear sin cambios abruptos la pared del cilindro.

Figure 3

La característica importante de esta representación es que los polígonos son entidades independientes entre sí y por lo tanto pueden ser tratados individualmente. Es decir, calcular el área de la superficie de un objeto sería tan fácil como calcular el área individual de cada polígono y sumarlas todas.

Modelado de objetos poligonales

Aunque el mesh poligonal es la forma de representación más común en informática gráfica, el modelado aunque es simple es también tedioso. La popularidad de esta representación ha sido debida a la facilidad para modelar, la aparición de técnicas de iluminación para objetos poligonales, y el hecho importante de que no existe restricción alguna de la forma o complejidad del objeto que se está modelando.

Las estrategias de modelado son principalmente de fuerza bruta. Una vez tenemos una aproximación inicial del objeto a modelar, interactivamente vamos moviendo los vértices y estirando y encogiendo los polígonos hasta lograr el objeto final, bien de forma ordenada mediante código o macros, o bien de forma arbitraria. Para poder conseguir la aproximación inicial hay varias estrategias dependiendo del objeto y del

material de que dispongamos:

a) **Modelado manual.** La forma más fácil de modelar un objeto real es manualmente mediante un digitalizador 3D. El operador dispone de una especie de lápiz con el que va tocando la superficie del objeto. El ordenador detecta la coordenada (x, y, z) de la punta del lápiz en el espacio y almacena ese punto. El usuario usa su experiencia y su juicio para colocar los puntos sobre el objeto donde quiere que vayan los vértices de los polígonos, aumentando la densidad en las zonas de alta curvatura. Una vez en el ordenador todos esos puntos se genera una red que forma la superficie del objeto. Donde las líneas curvas de la red intersecten se define la posición de los vértices de los polígonos.

b) **Generación automática.** Un dispositivo capaz de crear una malla poligonal de alta resolución de un objeto real es un scanner laser 3D. Se coloca el objeto sobre una tabla rotatoria delante del rayo. La tabla además se mueve verticalmente. Cuando empieza el proceso, el rayo va detectando un conjunto de contornos (la intersección del objeto con un conjunto de planos paralelos muy cercanos entre sí) a diferentes alturas, midiendo la distancia a la superficie del objeto. Un algoritmo de "skinning" (skin = piel) va procesando cada par de contornos convirtiéndo los datos de la superficie en un alto número de polígonos. En la figura 4 se aprecia la evolución de este algoritmo, y también un caso concreto: una cabeza de estatua poligonizada con esta estrategia usando 400.000 poligonos.

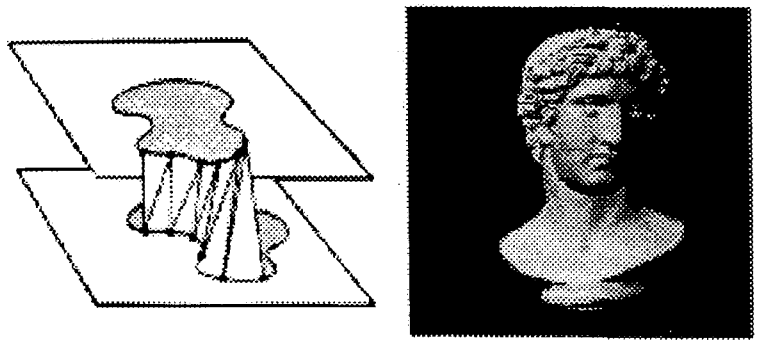


Figure 4

c) **Modelado matemático.** Está claro que cuando disponemos de la descripción matemática del objeto a modelar, lo más fácil es usar un algoritmo que vaya evaluando la ecuación en distintos puntos y considerando cada resultado como el vértice de un futuro polígono. La resolución de dichos polígonos se controla también directamente por el algoritmo de generación, por lo que habrá que tener en cuenta en qué zonas deben evaluarse mayor cantidad de puntos.

d) **Extruding.** Ésta es una de las técnicas más rápidas para construir objetos con forma de paralelepípedos. Partiendo de un polígono, el cual consideraremos que

representa un corte transversal de nuestro objeto, lo desplazamos a lo largo de un eje de coordenadas y ya tenemos el objeto, como el que aparece en la figura 5.

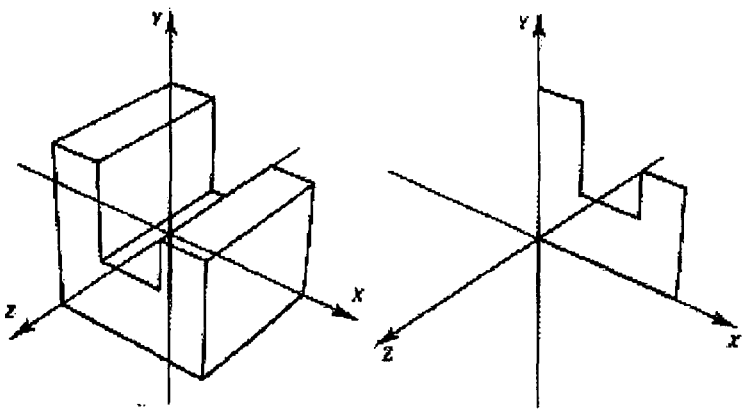


Figure 5

La tarea de construir la superficie corre a cargo del programa correspondiente que la realiza de forma automática. Todo lo que se requiere conocer es simplemente las coordenadas de los vértices del polígono que representa la sección transversal, y la longitud de la extrusión.

Un desarrollo de esta técnica consiste en construir no sólamente un polígono al principio y otro al final, sino una familia entera de polígonos a diferentes distancias discretas, a la vez que el polígono va siendo rotado alrededor del eje. Esto produce objetos "twisted" (¿retorcidos?) como el que se ve en la figura 6.

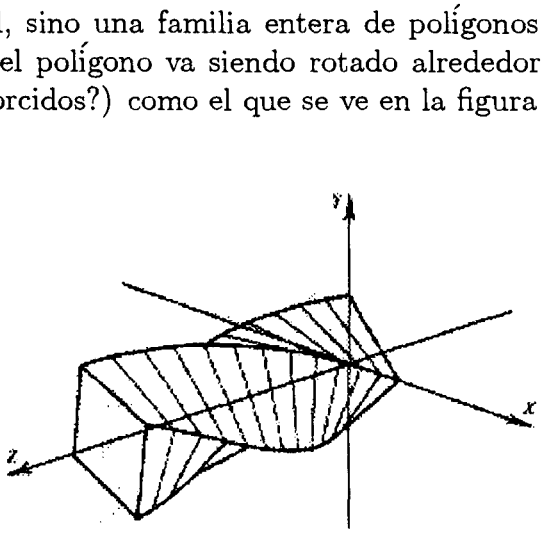


Figure 6

Y todavía podemos seguir complicándolo, haciendo que incluso el propio polígono vaya cambiando de forma a medida que nos vamos moviendo. Idealmente, un buen programa debería permitir al usuario especificar las secciones inicial y final (pudiendo ser distintas entre sí) y un ángulo de rotación para aplicar a la sección en cada etapa.

e) **Sweeping.** Es una extensión del extruding. Si en esta técnica íbamos moviendo el polígono por uno de los ejes, ahora vamos a moverlo a lo largo de una curva arbitraria en el espacio (por ejemplo cualquier curva cúbica paramétrica). De esta manera podemos construir objetos aún más variados, como el que muestra la figura 7.

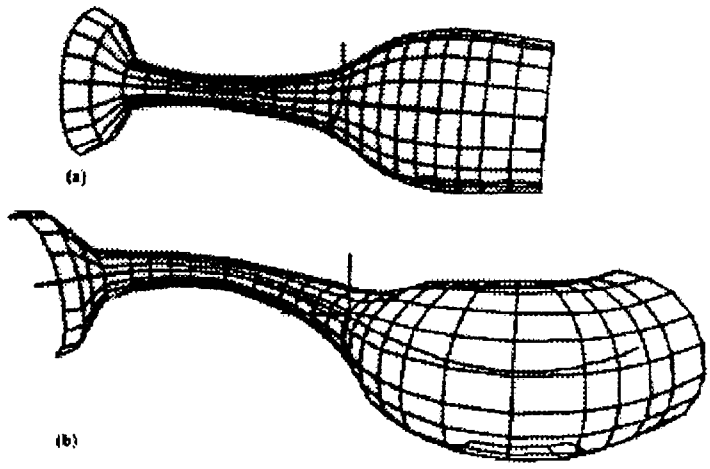


Figure 7

Sin embargo existen ligeros problemas que no aparecían en el extruding. Consideremos el simple caso de mover una sección transversal constante a lo largo de la curva $Q(t)$. Para ello habrá que definir intervalos en la curva en los cuales ir colocando el polígono, y luego ir uniendo vértice a vértice con el polígono anterior mediante líneas rectas. Las cuestiones que se plantean son: ¿qué intervalos debo escoger?, ¿cuál es la orientación del polígono a medida que se mueve por la curva?

En cuanto a la primera pregunta, dividir el parámetro t en intervalos iguales no tiene por qué dar los mejores resultados. Además, intervalos iguales en t no representan necesariamente intervalos iguales en la curva[1]. Lo ideal es que los intervalos dependan de la curvatura de cada zona de la curva. Si ésta es alta debe existir un número alto de polígonos para representar con detalle dicha zona. La forma más directa de hacer esto es usar el algoritmo de subdivisión de la curva hasta que consigamos tramos planos, y colocar un polígono entre cada dos tramos.

En cuanto a la segunda pregunta, necesitamos definir un sistema de referencia que viaje por la curva, y que mantenga el polígono siempre perpendicular a ésta. Para lograr esto necesitamos definir tres vectores ortogonales entre sí que formen los ejes de coordenadas. Uno de ellos será la tangente en t a la curva, y otro indicará en todo momento la vertical con respecto a la curva. El tercer vector es simplemente el producto vectorial de los dos anteriores, tal y como se muestra en la figura 8.

[1]Recordemos que t podía considerarse como la velocidad a lo largo de la curva. Valores de t igualmente distanciados no implican puntos de la curva igualmente distanciados.

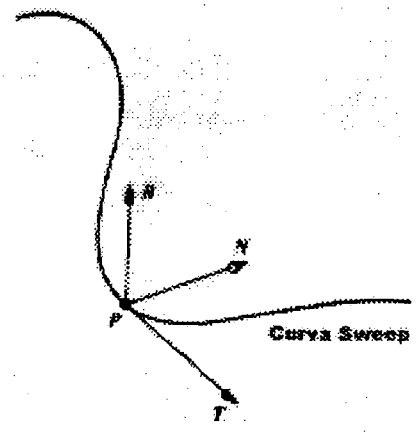


Figure 8

f) **Superficies de Revolución.** Esta técnica es la idónea para construir objetos tales como vasos, esferas, botellas, y en general aquellos que tengan simetría con respecto a un eje central. Para generar dicho objeto sólo es necesario un polígono que representará el contorno 2D de un lado de la figura. Dicho polígono es rotado 360° alrededor de un eje hasta lograr una superficie cerrada que represente al objeto. En la figura 9 puede verse un ejemplo de construcción de una copa de vino: Fíjense que la

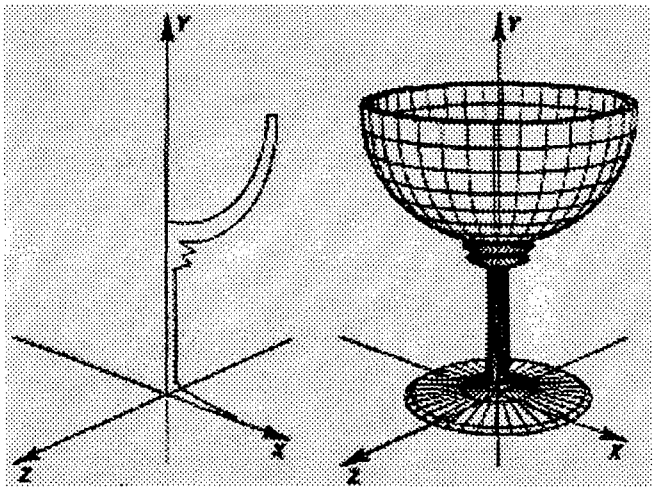


Figure 9

superficie final se ha construido en base a polígonos, y el número de ellos depende del número de vértices existentes en la curva original del contorno, y del incremento del ángulo en cada etapa de la construcción.

Al igual que ocurría con las dos técnicas anteriores, las superficies de revolución puede construirse automáticamente. El usuario solo debe especificar la curva que representa el contorno, el número de incrementos durante la rotación, y el eje sobre el cual

se va a hacer la construcción. Tal programa sólo fabricaría objetos simétricos, aunque podrían crearse mejoras, como por ejemplo que en la etapa de construcción, la curva que identifica el contorno fuera variando durante la rotación, o incluso que el propio eje de rotación fuera moviéndose a lo largo de una curva arbitraria. El límite está en la imaginación.

4.3 PATCHES BICÚBICOS

Si consideramos la representación anterior mediante una malla de polígonos, y sustituimos cada uno de ellos por un patch bicúbico de los que veíamos en el tema anterior, estaremos hablando entonces de una red de patches (**patch mesh**). Cada patch es una superficie curva $Q(s,t)$ donde $0 \leq s,t \leq 1$, los cuales mantienen algún tipo de continuidad entre sus patches vecinos.

Para la mayoría de las aplicaciones, el modelar o construir una estructura de datos que represente un objeto tridimensional es más difícil si se usan patches bicúbicos que usando representación poligonal, ya que usando un digitilizador 3D o un scanner con el software apropiado ya hemos visto que podemos generar rápidamente gran cantidad de polígonos que representen objetos muy complejos. Cuando trabajamos con patches, suponiendo que éstos sean de Bezier, necesitaremos 16 puntos de control para cada patch, y además debemos mantener la integridad de la representación, en cuanto a mantener los requisitos de continuidad entre patches. Por todo ello, la descripción de una malla de patches suele generarse de forma semi-automática, como veremos en la siguiente sección.

Pero a pesar de esta desventaja, la diferencia en calidad es la que garantiza su uso extendido en las aplicaciones de CAD. La representación es sencilla, y usando el software apropiado para poder modificar la posición de los puntos de control podemos ajustar la forma del objeto que estemos modelando. Esto puede verse en la figura 10.

Otra gran ventaja es que al tener una descripción analítica exacta de la superficie, las propiedades físicas tales como masa, volumen, área y momentos de inercia pueden extraerse de dicha descripción. Esta propiedad es totalmente aprovechada por las aplicaciones CAD. Por último, hay que mencionar que la representación de objetos de alta resolución por medio de polígonos necesita un alto coste de almacenamiento y de tiempo de cómputo. De hecho, en escenas con muchos objetos complejos casi debe ser tratado como una base de datos. Por el contrario, usando patches bicúbicos obtenemos una representación mucho más exacta con mucho menos coste de memoria.

Veamos un ejemplo donde se comparan ambas estructuras de datos, y por supuesto tomaremos como objeto modelo la famosa tetera de Utah. En la figura 11a) se han dibujado varias líneas con s y t constante para cada patch. El objeto está compuesto de 32 patches de Bezier, y se puede ver sombreado y con línea gruesa un patch concreto. En la figura 11b) se ven los puntos de control en wire-frame (la zona sombreada se corresponde con los puntos de control del patch sombreado). En la figura 11c) se

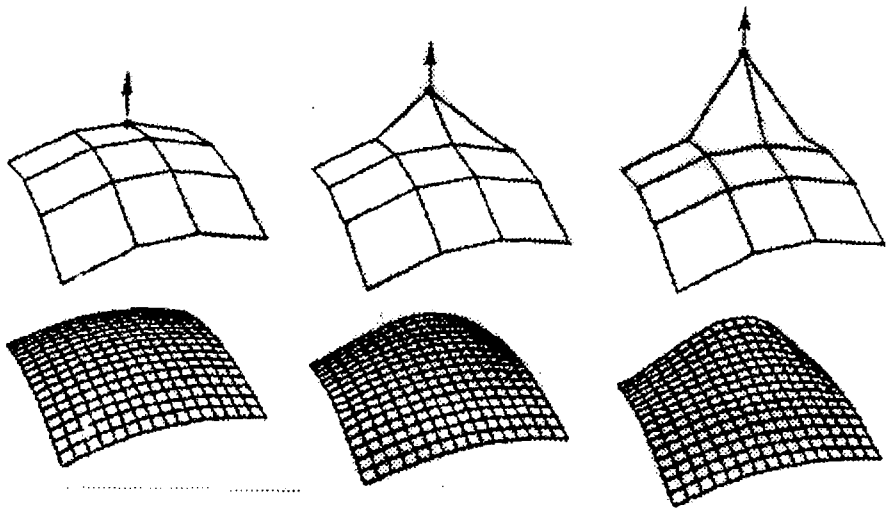


Figure 10

muestra un wire-frame de los bordes de cada patch.

Como para cada patch son necesarios 16 puntos de control, para almacenar los 32 patches necesitaremos $32 \times 16 = 512$ puntos de control, aunque en realidad son menos, ya que la mayoría de los patches comparten 12 puntos de control con sus patches vecinos. En este caso han sido necesarios 306 puntos de control. Cada punto requiere 3 coordenadas, donde cada una es un número real (4 bytes). Esto nos lleva a que necesitamos $306 \times 3 = 918$ números reales (3'6 Kb).

Ahora veamos lo que supondría mantener el mismo objeto mediante mallas poligonales. En primer lugar, una resolución equiparable visualmente a la del dibujo sería sustituir cada patch por 64 polígonos, y digo visualmente porque recordemos que usando patches bicúbicos podemos aumentar la resolución cuando queramos (los polígonos una vez elegidos son fijos) y además realizar cálculos exactos sobre la geometría del objeto. Pero bueno, pues aún así, requeriríamos $64 \times 32 = 2048$ polígonos, que a cuatro vértices por polígono nos salen $2048 \times 4 = 8192$ puntos, de 3 coordenadas cada uno: $8192 \times 3 = 24576$ números reales (96 Kb). Es decir, una representación "inexacta" de la misma tetera requeriría $96/3.6 = 26.6$ veces más de memoria.

Por si fuera poco aún existe otra ventaja adicional. Supongamos que queremos obtener una imagen de una cierta escena en la que hay numerosos objetos, entre ellos la tetera, y ésta queda bastante lejos del observador, por lo que sólo se proyectará sobre una pequeña área sobre la pantalla. Usando un modelo poligonal realizaremos un cierto coste computacional para dibujar el objeto en pantalla, y dicho coste será independiente del tamaño que éste ocupe en la imagen final. Pero si estamos usando una red de patches, y además usamos un esquema de subdivisión que vaya convirtiendo los patches a polígonos hasta que éstos sean planos o hasta que su proyección se halle contenida en un único pixel de la pantalla, el coste en ese caso será mucho menor. Sólo

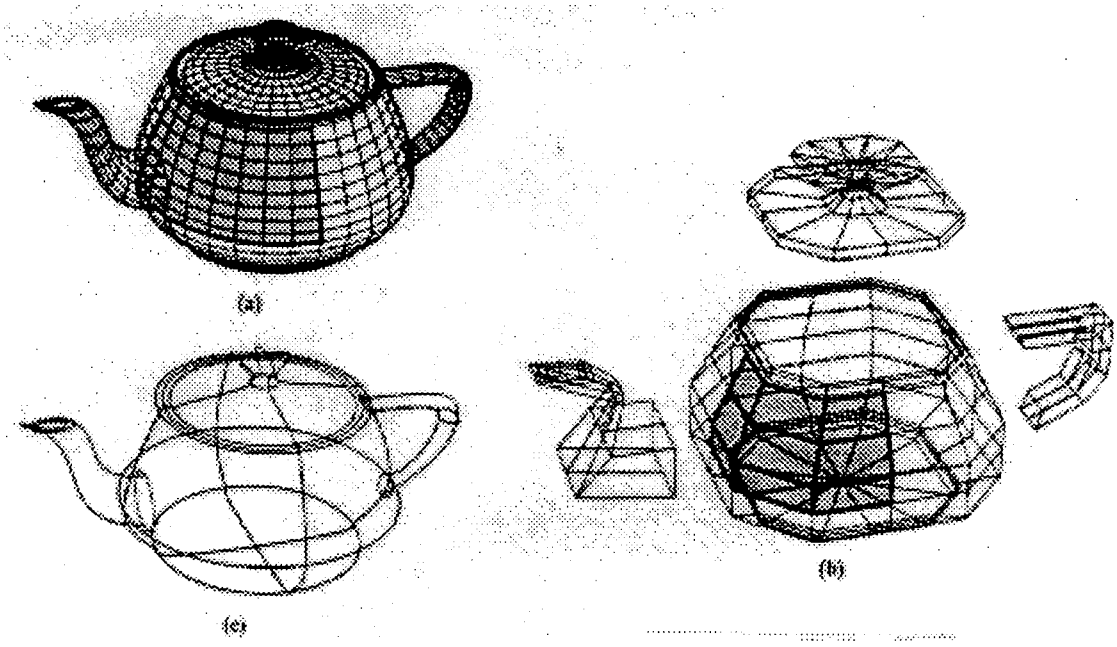


Figure 11

hay que ver que el esquema poligonal estaría sobreescribiendo 2048 polígonos diferentes sobre los mismos pixels.

Modelado de objetos mediante patches bicúbicos

Existen dos estrategias prácticas de modelado que pueden usarse para obtener una aproximación inicial de patches, a la que luego habra que ajustarle sus puntos de control.

a) **Surface Fitting.** (¿ajustado de superficie?) Es básicamente un método particular de realizar interpolación de superficie. Se dispone de un conjunto de puntos que descansan sobre la superficie del objeto que queremos representar, y el objetivo es producir una descripción de los patches a partir de dichos puntos. De forma similar a cómo ajustamos una línea a través de un conjunto de puntos del plano, ajustaremos una superficie a través de un conjunto de puntos del espacio 3D. Así esta técnica puede usarse para modelar objetos de los que conocemos ya algunos puntos, bien sea porque el objeto es abstracto, o porque los hemos recogido previamente con un digitalizador.

Por supuesto, la diferencia entre esta técnica y obtener una representación poligonal con dichos puntos es que obtenemos una superficie continua a partir de los puntos. Sin embargo, hay que tener en cuenta el detalle que supone que en el caso de un objeto real, la superficie definida por los patches no coincida exactamente con la superficie de la cual se extrajeron los puntos digitalizados. La exactitud de la representación va a depender del número de puntos iniciales, así como de la extensión espacial de los

patches en la red.

El principio del proceso es el siguiente: partimos de un conjunto de puntos 3D. El siguiente paso es ajustar una curva a través de los puntos en dos direcciones paramétricas perpendiculares, s y t. Esta red de curvas se divide en un conjunto de cuadriláteros curvilíneos, los cuales forman los bordes de los patches individuales. En cada cuadrilátero se crean los puntos de control necesarios para cada patch, y finalmente se construye la red de patches. En la figura 12 se ilustra el proceso seguido, así como un ejemplo en el que a partir de los puntos digitalizados de una cara se ha construido primero la red de patches, y luego se ha sombreado.

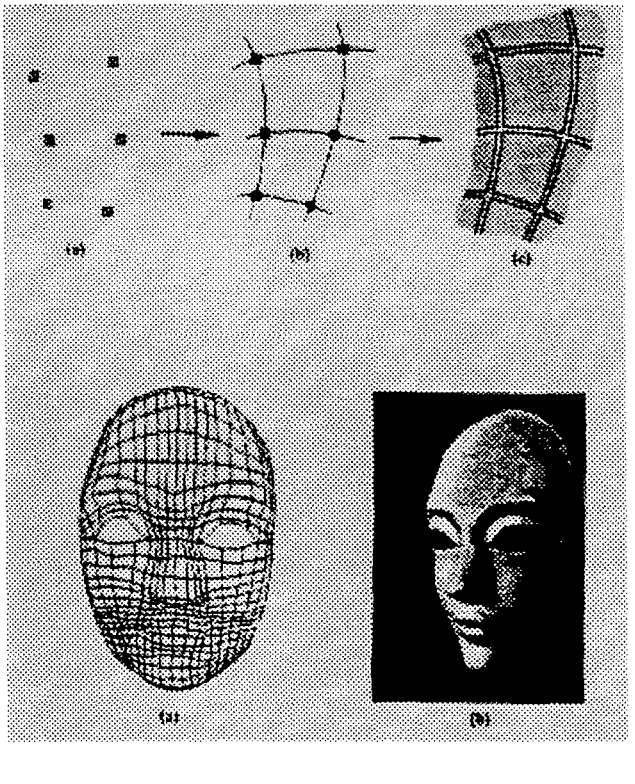


Figure 12

b) **Cross-sectional sweeping.** (¿?) De la misma forma que realizábamos un sweeping de un polígono alrededor de una curva arbitraria para generar un objeto poligonal, en este caso el polígono se convierte en una nueva curva cúbica. Esta curva, que representaría la sección transversal de la superficie a modelar, se va colocando a intervalos apropiados a lo largo de la curva que indica la trayectoria (curva de sweep), usando las mismas técnicas que comentábamos con anterioridad. En la figura 13 se ilustra este proceso, donde pueden verse dos curvas seccionales colocadas en dos posiciones consecutivas a lo largo de la curva de sweep. A partir de los dos extremos de estos dos segmentos generamos otras dos curvas. Estas cuatro curvas forman los bordes de un único patch, y de ellas obtenemos la descripción necesaria para dicho patch. Luego se siguen colocando más curvas seccionales.

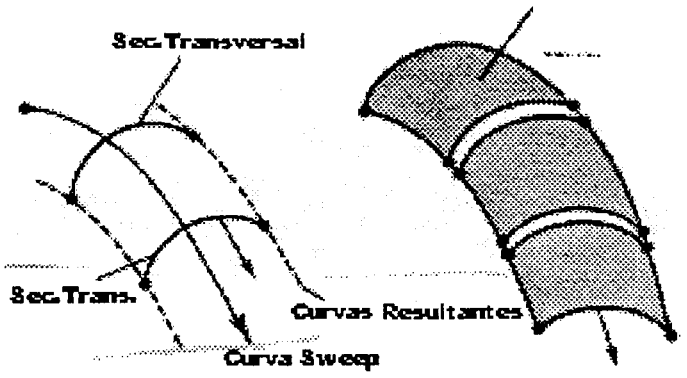


Figure 13

Superficies Ruled (Ruled Surfaces)

Este tipo de superficies está a mitad de camino entre los patches cúbicos y los polígonos. En una de las direcciones parámetricas seguimos teniendo segmentos curvos, pero en la otra dirección sólo tenemos rectas, como se ve en la figura 14. Un ejemplo pueden ser las paredes de los cilindros y los conos. Un método conveniente para crear tales superficies es usar una aproximación paramétrica para las dos curvas, e ir interpolando linealmente el espacio entre ambas. Es decir, si $Q_1(t)$ y $Q_2(t)$ representan las dos curvas, siendo $0 \leq t \leq 1$, la superficie vendría dada por

$$Q(s,t) = (1-s)Q_1(t) + sQ_2(t), \text{ siendo } 0 \leq t, s \leq 1$$

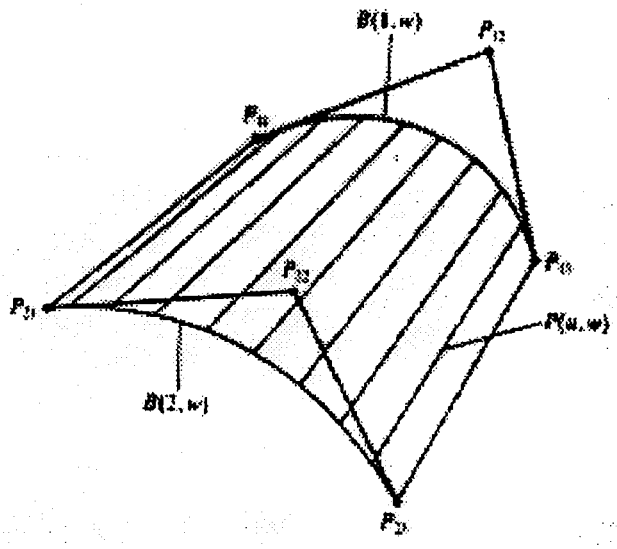


Figure 14

En realidad, las curvas de la dirección t no tienen por qué ser curvas cúbicas. Alguna de ellas podría ser una recta, o una semicircunferencia, y seguirían siendo superficies de este tipo. Incluso ambas podrían ser rectas, en cuyo caso recibirían el nombre de **superficies bilineales**.

4.4 GEOMETRÍA SÓLIDA CONSTRUCTIVA (CSG)

Los dos modelos anteriores, mallas de polígonos y de patches, pertenecen a la categoría de los esquemas de representación de fronteras. En estos esquemas la construcción de los objetos 3D se hace a partir de una descripción de su superficie. Pero existen otra categoría de esquemas, conocidas como esquemas de **representación volumétrica**, que como su propio nombre indica describen el volumen interno al objeto, pudiendo por tanto almacenar propiedades del interior, tales como peso, presión interna, o incluso composición química.

La geometría sólida constructiva (CSG en inglés) es un tipo de representación volumétrica que facilita en gran medida la interactividad en el modelado de sólidos. La idea es llegar a construir los objetos mediante una combinación adecuada de volúmenes básicos elementales, conocidos como **primitivas geométricas**, las cuales pueden ser esferas, conos, cilindros, cubos, etc. Estas primitivas se combinan usando un conjunto de operadores booleanos y una serie de transformaciones lineales.

Cada objeto se almacena como un árbol. Los nodos hoja contienen a las primitivas, y los nodos no terminales almacenan los operadores booleanos o las transformaciones lineales que deben aplicarse. Debido a que la representación no sólamente define la forma del objeto sino además todas las etapas necesarias para su construcción, el realizar modificaciones a los objetos es muy sencillo. Por ejemplo, incrementar el diámetro de un agujero que atraviese un sólido rectangular implica una sencilla modificación (aumentar el radio de la primitiva cilindro). Esto constrasta con la representación por fronteras, donde efectuar la misma alteración no es nada trivial.

El conjunto de operadores booleanos que se usa para la representación también es usado como técnica del interface de usuario. Éste puede especificar primitivas y combinarlas usando estos operadores. La representación del objeto es una "grabación" de las operaciones que interactivamente ha ido ejecutando el usuario. Es por esto por lo que se dice que el modelado y la representación no están separados: el modelado se convierte en la representación. Veamos un ejemplo que demuestre tal afirmación, con la figura 15.

Existen distintos tipos de operaciones booleanas. En la figura 15 pueden verse las tres operaciones más frecuentes entre sólidos: la **unión**, que incluye todos los puntos que pertenezcan al interior de cualquiera de los objetos, la **substracción**, que elimina los puntos del primer objeto que pertenecen también al segundo, y la **intersección**, que sólamente mantiene los puntos pertenecientes a ambos objetos. Así, con estos tres pasos hemos creado el objeto de la figura de una forma super sencilla. Imaginen crear

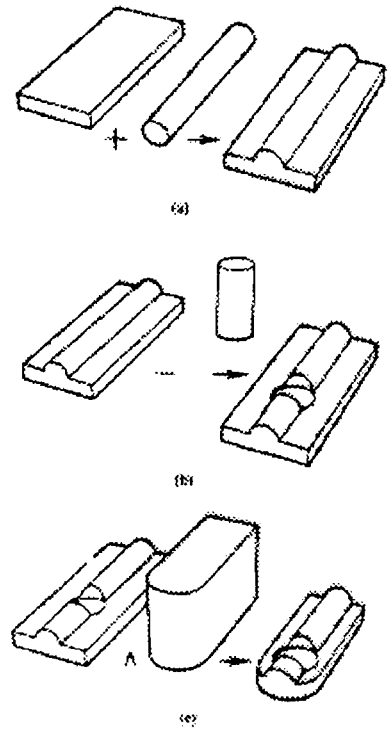


Figure 15

el mismo objeto usando una representación poligonal (o incluso alámbrica). Claro está que el tipo de objetos ideal para esta técnica son aquellos que poseen una geometría inherente clara, como por ejemplo todo tipo de piezas mecánicas. Las aplicaciones CAD/CAM son sus clientes más allegados.

En la figura 16 puede verse una representación que refleja la construcción de un objeto simple. Aparecen tres primitivas en las hojas del árbol: dos cajas y un cilindro. Las cajas se combinan mediante una unión y luego se genera un agujero en medio definiendo un cilindro y luego restando. Obsérvese como las dos cajas de los nodos hojas son diferentes entre sí. ¿Significa esto que son dos primitivas diferentes? No, existe una única primitiva para las cajas, que puede ser un cubo unidad inicialmente. Cualquier otra caja, sea ésta más ancha o menos alta, se generará a partir de dicha primitiva mediante la transformación lineal correspondiente.

La figura 17 muestra dos ejemplos en los que puede apreciarse la verdadera potencia de esta técnica. En el primero de ellos, dos piezas desarrolladas por separado se combinan para formar la configuración deseada mediante un operador unión seguido por un operador de substracción. El segundo ejemplo muestra un objeto complejo construido a partir de la unión de varios cilindros, los cuales han sido restados a una esfera. Este segundo objeto es casi imposible de describir de forma tan exacta con ningún otro método.

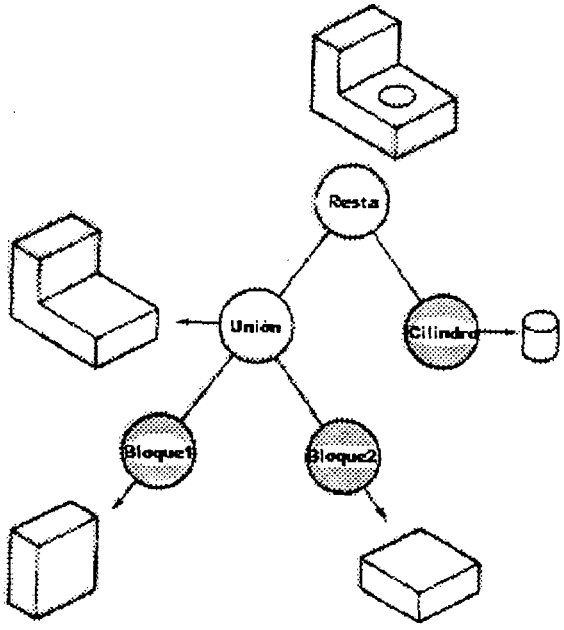


Figure 16

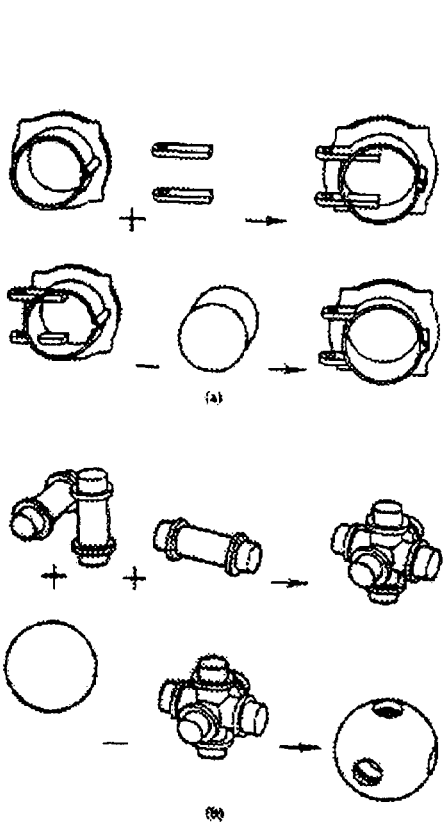


Figure 17

Pero no todo son ventajas. Un problema práctico es el tiempo de cálculo requerido para producir una imagen del modelo. Otro problema es la limitación de las operaciones disponibles para crear y modificar el sólido. Las operaciones booleanas son globales, afectan al sólido completo. Una operación local, como una modificación detallada en una cara de un objeto complejo no es fácilmente implementable usando el conjunto de operadores booleanos. Este hecho ha provocado que muchas aplicaciones usen B-rep internamente para la representación de los objetos. Es decir, aunque en la fase de construcción vayamos creando el árbol de operaciones, finalmente lo tranformamos en una malla poligonal para crear la imagen o almacenar la estructura de datos.

Una ventaja más: el esquema CSG permite de forma fácil examinar el interior de un objeto modelado con esta técnica. Esto se logra realizando la intersección del objeto con un plano de corte que lo divida en dos, con lo que podemos ver con detalle cualquier corte transversal. Esto se usa mucho en publicidad o en vídeos explicativos donde se quiera mostrar con detalle el interior de algún objeto.

Pero para poder lograr toda esta potencia, se necesita implementar algún mecanismo en el ordenador (lógicamente) que soporte la matemática y la geometría que lleva inherente esta técnica. Básicamente, se suele almacenar un sistema de inecuaciones para cada primitiva que indica para un punto de coordenadas (x, y, z) si se encuentra dentro o fuera del objeto. Por ejemplo, para una esfera con centro en el punto (x_c, y_c, z_c) y radio r es tán fácil como

$$(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 \leq r^2$$

Para una caja serían necesarias una inecuación para cada una de las 6 caras, que indicara si un punto está por el lado de fuera o por el de dentro del plano que contiene a la cara. Un punto estaría dentro de la caja si verificase las seis inecuaciones. Con esta información, la aplicación de los operadores booleanos van combinando los sistemas de inecuaciones hasta llegar al objeto final.

4.5 TÉCNICAS DE SUBDIVISIÓN ESPACIAL

Este tipo de técnicas son métodos que consideran el espacio completo del entorno del objeto. Dicho espacio es subdividido en zonas o celdas adjuntas y no intersectantes, y cada uno de esas zonas o celdas es etiquetada con el objeto al que pertenece (si es que pertenece a alguno). Suele usarse como estructura de datos secundaria o auxiliar. Por ejemplo, generar la imagen de un objeto representado por un árbol CSG directamente no es tarea fácil. Una alternativa válida puede ser convertir el árbol a un esquema de subdivisión espacial y generar la imagen a partir de éste último.

Dentro de este tipo de técnicas existen varios modelos.

4.5.1 Descomposición en celdas

Es una de las formas más generales de subdivisión espacial. Se define un conjunto de celdas primitivas básicas, con las cuales se forma el objeto (es como jugar al Lego). Estas celdas no pueden solaparse, y el único operador booleano permitido es la unión. La estructura de datos que representa al objeto es simplemente un array de celdas (cada celda con su posición y tamaño). Un ejemplo puede verse en la figura 18.

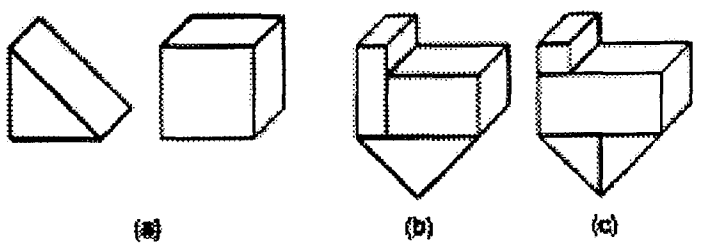


Figure 18

4.5.2 Enumeración de la ocupación espacial

Es un caso especial de descomposición en celdas, en la cual el objeto se descompone en celdas idénticas alineadas según una malla regular. Estas celdas reciben el nombre de **voxels**, por su similitud con los pixels. El proceso es muy parecido. Representar mediante pixels un círculo en la pantalla es tan fácil como discretizar la pantalla (dividir toda la pantalla en una malla de pequeños cuadraditos) e indicar para cada pixel si pertenece o no al interior del círculo (por ejemplo los que pertenezcan se pintarán de un color y los que no de otro). De forma similar se realiza el proceso 3D: el espacio alrededor del objeto es discretizado en una malla 3D de pequeños cubitos, y cada cubito es etiquetado indicando si pertenece o no al interior del objeto. La figura 19 muestra un donut representado con esta técnica.

El voxel típico suele ser un cubo, pero no es el único tipo de primitiva que podemos usar. A diferencia de la técnica anterior de descomposición en celdas, sólo existe un tipo de primitiva, y tanto la posición como el tamaño de cada una de ellas es único e invariante, y vendrá definido por la resolución de la malla. Para cada voxel solamente podemos indicar ausencia o existencia de algún objeto de forma booleana. Es decir, o pertenece en su totalidad o no pertenece. No vale decir que la mitad derecha del voxel pertenece al objeto y la otra mitad no. Es por eso que en la figura anterior se nota el efecto escalera o aliasing en la frontera del objeto. La única manera de evitarlo es aumentando la resolución de la malla de voxels, haciendo éstos más pequeños. Para representar un objeto sólo hay que decidir qué voxels pertenecen al objeto y cuáles no. De esta manera el objeto vendrá especificado por un array de voxels ocupados.

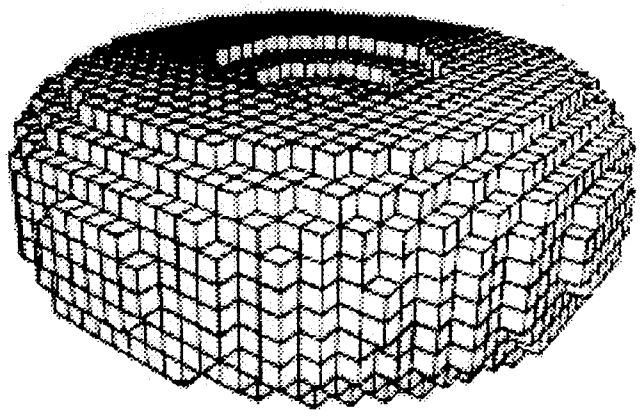


Figure 19

Usando esta técnica es fácil saber si un punto del espacio pertenece o no a un objeto, o determinar cuándo dos objetos son adyacentes. Esta propiedad es explotada en la mayoría de las aplicaciones biomédicas para representar datos volumétricos obtenidos desde tomografías (TACs). La gran ventaja es que, al contrario que ocurría con los esquemas B-rep, tengo información sobre el interior de objeto. Por ejemplo, en la figura anterior del donut, los voxels que caen en el interior del donut no se ven, pero sí están almacenados en la estructura de datos. Por ello, en una aplicación médica puede tenerse por ejemplo un brazo modelado con esta técnica, y almacenar en cada voxel del brazo no sólo su pertenencia a éste, sino además a qué tipo de tejido pertenece (hueso, nervio, vena, etc.), con lo que el médico usuario del sistema podría visualizar en pantalla sólamente el tipo de tejido que deseara.

Entre sus desventajas está la ya comentada de la prohibición de una ocupación "parcial" dentro de un mismo voxel. Esto provoca que los únicos sólidos que pueden ser representados con total exactitud son aquellos cuyas caras sean paralelas a las caras de los voxels, y cuyos vértices coincidan exactamente con la malla. Al igual que ocurre con los pixels en un bitmap, las celdas pueden ser tan pequeñas como se quiera para aumentar la exactitud de la representación, a costa de un aumento importante de memoria, ya que para representar un objeto con n voxels de resolución en cada dimensión necesitaremos almacenar n^3 voxels.

4.5.3 Octrees

Los octrees son una variación jerárquica del esquema anterior, que intenta evitar el excesivo coste de almacenamiento. La idea es describir de forma jerárquica mediante un arbol octal la distribución de los voxels, en lugar de hacerlo mediante una lista exhaustiva de ellos. Está basado en la misma estructura que los quadtrees, empleados en el tratamiento de imágenes. Veamos primero en qué consisten los quadtrees y luego veamos su extensión a 3D. Supongamos que tenemos la imagen de la figura 20, y

queremos obtener una representación de ella por medio de un quadtree. Se supone que la imagen representa una escena bidimensional a nivel de pixels.

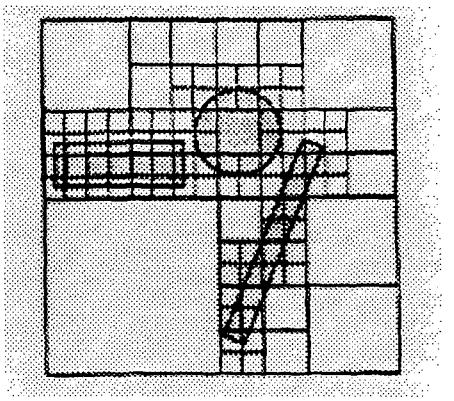


Figure 20

Comenzamos a crear el árbol considerando una región cuadrada que represente el espacio total de la escena (en 3D sería un cubo). Esta región vendrá representada por el nodo raíz del árbol. Como la región se encuentra ocupada por objetos en su interior, la subdividimos en cuatro subregiones, representadas por los cuatro nodos hijos en el árbol (en 3D serían 8 hijos). A continuación cualquiera de las subregiones que se encuentre ocupada por objetos se va subdividiendo recursivamente, hasta que el tamaño de la subregión se corresponda con la resolución máxima permitida (pixels en 2D y voxels en 3D). De esta manera, al final del proceso tendremos un árbol que me representa la totalidad de la escena, como se ve en la figura 21. El orden en el que se almacenan los hijos debe ser establecido previamente.

Existen dos tipos de nodos hoja en el árbol. Unos corresponderán a subregiones que no contienen a ningún objeto, mientras que otras corresponderán a voxels de mínimo tamaño que se encuentran ocupados por parte de algún objeto. Obsérvese que en el caso 2D, los objetos se representan por su frontera, y el espacio de su interior se cuenta como espacio no ocupado. En el caso 3D, los objetos se representan por su superficie, y sólo podremos subdividir regiones que contengan parte de la superficie.

Una vez visto el quadtree, la definición para el octree es automática. El nodo raíz representa el volumen total de la escena. Cada nodo se divide en ocho subregiones (octantes), y recursivamente se van subdividiendo como en el caso 2D hasta que el volumen al que representan esté vacío o se llegue al tamaño mínimo para un voxel. Existen dos formas diferentes para representar una escena mediante un octree. La primera es tal y como acabamos de ver, obteniendo una representación completa de los objetos en la escena. El conjunto de voxels ocupados por el objeto constituye la representación del objeto. Esta forma viene muy bien para escenas simples con pocos objetos.

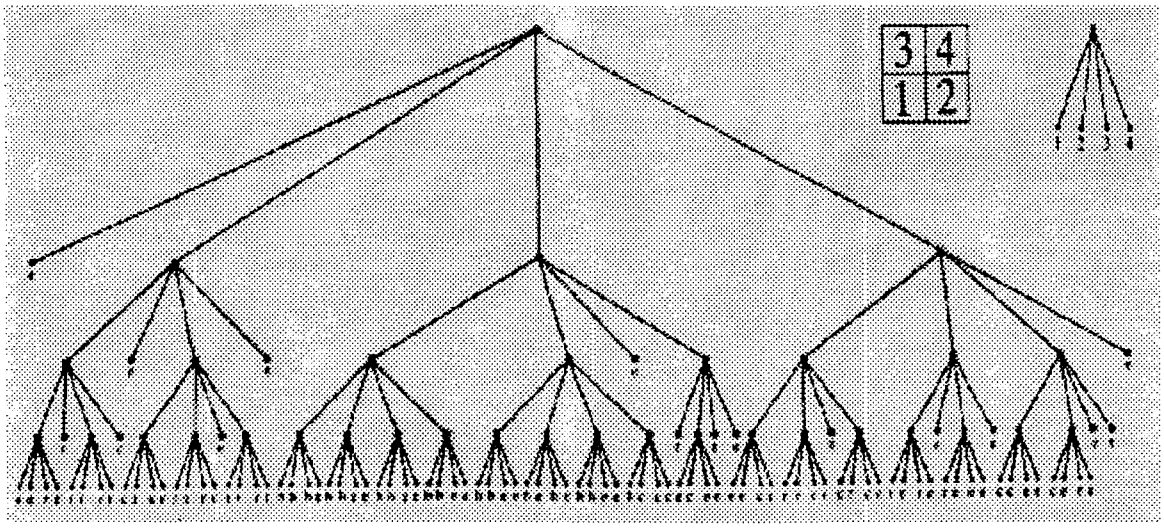


Figure 21

Sin embargo, para una escena compleja, se requiere un alto coste computacional para descomponer el espacio ocupado en un largo número de voxels, así como un enorme volumen de memoria. Una alternativa común es usar una estructura de datos estándar para representar a los objetos, y usar el octree exclusivamente para representar la distribución de los objetos dentro de la escena. En este caso, un nodo hoja que representase una región ocupada indicaría un puntero a la estructura de datos de cualquier objeto que intersectase la región. Esta alternativa se muestra en la figura 22, aunque usando quadtrees para verlo más fácil.

En este caso, el proceso de subdivisión finaliza cuando la región ocupe sólamente un objeto. Una región representada por un nodo hoja no tiene por qué estar completamente ocupada por el objeto asociado a ella. La forma del objeto en el interior de la región vendrá descrita por su representación en la estructura de datos correspondiente, la cual vendrá compuesta por polígonos o patches, en el caso de estar trabajando con modelos de superficie para los objetos. En general, una región ocupada representada por un nodo hoja podrá intersectar con varios polígonos, en cuyo caso vendrá representada por una lista de punteros a las estructuras de datos de los objetos.

4.5.4 Árboles BSP

Este tipo de árboles siguen siendo en esencia octrees, pero la estructura de links en el árbol es ligeramente diferente. Las siglas BSP vienen del inglés (como siempre) y significan partición espacial binaria. Es decir, en cada subdivisión dividimos la región en dos mitades, por lo que el número de hijos de cada nodo siempre es dos (los que tengan hijos). En la figura 23 se muestra un ejemplo, aunque de nuevo en 2D donde es más fácil de ver.

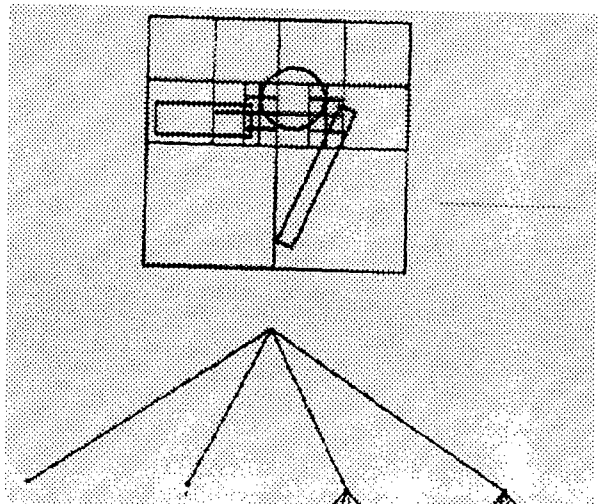


Figure 22

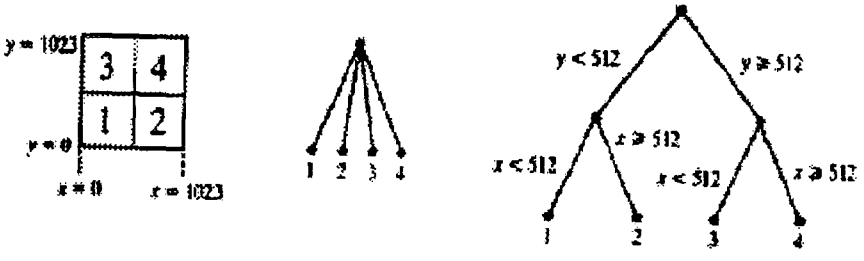


Figure 23

En dicha figura puede verse la representación para un solo nivel de subdivisión usando un quadtree, y la misma representación, que necesita dos niveles, usando un árbol BSP. En el primer nivel se ha dividido a lo largo del eje x, obteniendo la mitad superior y la inferior de la región inicial. En el segundo final, cada subregión se vuelve a subdividir por la mitad, pero esta vez a lo largo del eje y. La extensión a 3D es directa: haría falta un nivel más para subdividir a lo largo del eje z. Cada nodo no terminal del árbol representa un plano de corte que divide el espacio ocupado en dos mitades. Un nodo hoja representa una región que no ha sido subdividida y que contiene punteros a las estructuras de datos que representan a los objetos que intersectan la región.

Subdivisión adaptativa

Cuando usamos un árbol BSP para representar una subdivisión del espacio en celdas cúbicas, en realidad no hay prácticamente diferencia con el hecho de usar un octree. Sin embargo, la verdadera utilidad de los árboles BSP es cuando no exigimos que las subdivisiones sean siempre en celdas cúbicas, es decir, que cada región se subdividad siempre en dos mitades iguales. De hecho, la idea original de este tipo de árboles consideraba el hecho de poder dividir el espacio usando planos de corte con cualquier orientación. Por ejemplo, existen aplicaciones que eligen los planos de corte para que en cada subdivisión separen dos objetos entre sí, dejando uno a un lado y otro al otro, con lo que el número de subdivisiones, y por tanto de niveles del árbol, es mucho más pequeño.

Por otro lado, normalmente los objetos de la escena no se encuentran uniformemente distribuidos dentro del espacio. Más aún cuando dichos objetos son los patches o polígonos que aproximan las superficies de los objetos reales. Un objeto real vendrá representado por un largo conjunto de patches conectados en el espacio, y existirán regiones extensas de espacio vacío entre los objetos. Cuando elijamos la posición para los planos en función de la distribución de los objetos en la escena, diremos que hemos aplicado subdivisión adaptativa para simplificar el árbol BSP.

En la figura 24 podemos ver un ejemplo de esta subdivisión frente a la subdivisión normal. Supongamos la región que se muestra conteniendo 16 objetos, etiquetados desde a hasta p. Estos objetos se encuentran distribuidos de forma no uniforme en el interior de la región. En la figura a) vemos el BSP tradicional, de forma similar a como lo haría un quadtree. La máxima profundidad de este árbol es 8, que además es la longitud máxima de búsqueda requerida para identificar a qué región pertenece un punto determinado. En la figura b) se ha usado subdivisión adaptativa. En cada paso, se ha elegido como plano de corte aquél que divide la región de tal forma que a cada lado del plano existan el mismo número de objetos. Esto produce un árbol BSP más balanceado en el que la lóngitud máxima de búsqueda se ha reducido a 4.

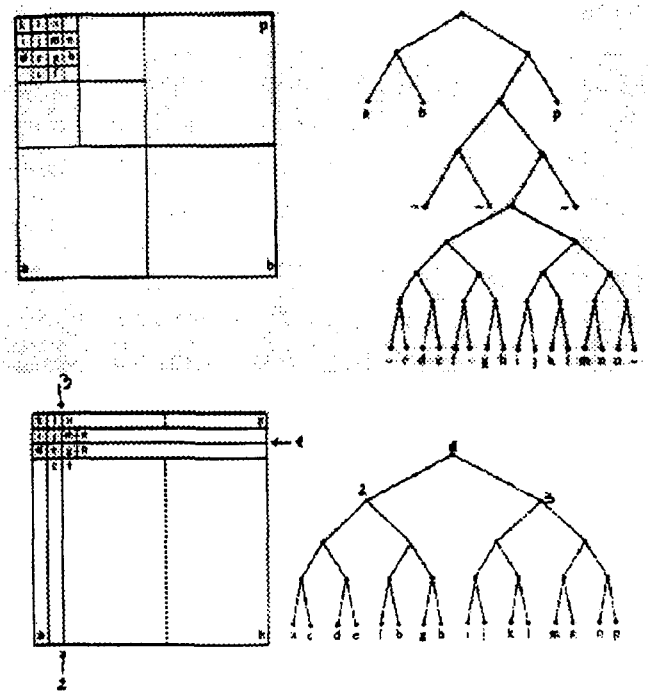


Figure 24

Chapter 5 VISIBILIDAD

Una vez modelada una escena tridimensional, compuesta quizás por miles de polígonos o superficies paramétricas, falta aún el problema de visualizarla. Como ya hemos comentado anteriormente, el primer paso consiste en transformar, mediante las ecuaciones de perspectiva, todo el modelo, habiendo aplicado previamente las transformaciones de vista pertinentes. Un problema que se plantea en esta etapa es decidir cuáles de los infinitos puntos que poseen los objetos deberían ser puestos en perspectiva, en orden a sintetizar visualmente su forma. Para superficies poliédricas, la solución suele estar en transformar sólo los vértices de los polígonos constituyentes, restituyendo los lados y caras a posteriori. Para una superficie regular, no existen tales vértices y es mucho más delicada la selección, optándose en la mayoría de los casos por convertir la superficie en una malla de polígonos planos y procesarla como si de un poliedro se tratara.

En cualquier caso, la mera puesta en perspectiva de una escena no basta para comunicar a un observador real las propiedades matemático-físicas de un modelo, el cual sólo existe en forma de largas cadenas de números o de ecuaciones. El proceso por el cual un modelo matemático llega a aparentar ser un objeto real, con diferentes niveles de abstracción por parte del observador, se conoce por **rendering**. Este proceso consta de dos fases claramente diferenciadas.

La primera fase consiste en la identificación de aquellas partes de la escena que son visibles desde un punto de vista específico. En el mundo real, los objetos más cercanos al observador ocultan a los más distantes, debido a la interacción de la luz entre los distintos objetos opacos y transparentes. Por lo tanto es necesario determinar qué partes de los objetos no son visibles al observador, bien porque son superficies que se encuentran por la parte de detrás del objeto con respecto a él, o bien porque existen otros objetos por delante que impiden su visión. En un sistema informático, hemos visto cómo los objetos se hallan descritos según una variedad de tipos de modelos, y básicamente cada uno de ellos puede reducirse a una enorme colección de números. Las relaciones espaciales entre los objetos también se almacenan de forma numérica mediante coordenadas, y son estos datos toda la información que se dispone para poder detectar qué objetos o partes de ellos son visibles al observador y cuáles están ocultas. A lo largo de este capítulo veremos los diferentes procedimientos que se han desarrollado para resolver este problema.

La segunda fase, una vez determinadas todas las zonas visibles de los objetos que van a aparecer en la imagen final, consiste en establecer los cirterios que permitan dibujarlas o colorearlas con el máximo realismo visual. Es decir, hay que calcular de la forma más exacta posible cuál es el comportamiento de la luz dentro de la escena, y cómo ésta interactúa con las superficies de los objetos, considerando todos los efectos que pueden producirse: sombras, transparencias, reflexiones, refracciones, etc. El objetivo final es obtener cuál es el color final para cada pixel de la imagen. Esta fase la abordaremos en el capítulo siguiente.

5.1 CONCEPTOS Y TÉCNICAS GENERALES

Veamos a continuación una visión general del problema de la visibilidad dentro del proceso de creación de una imagen realista. Para ello comenzaremos dando una definición, y posteriormente veremos una serie de técnicas generales que intentan simplificar el problema.

5.1.1 Definición

Podemos definir el concepto de visibilidad de un modo matemáticamente muy simple, pero que encierra una enorme coplejidad algorítmica. Consideremos de forma abstracta un conjunto de puntos S (que va a representar la superficie de un objeto concreto) contenido en $\Re^3$ y un observador P que no pertenece a S. Diremos que Q (punto de S) es visible desde P, si el segmento abierto PQ no intersecta a S. Es decir:

$$\begin{aligned} \overline{PQ} \quad &: \quad R(t) = tQ + (1-t)P, \text{ siendo } 0 < t < 1 \\ Q \text{ visible} \quad &\Leftrightarrow \quad R(t) \notin S, \forall t \in (0,1) \end{aligned}$$

En caso de no verificarse esta condición, el punto Q se denomina invisible respecto a P y ello quiere decir que existe algún punto $R(t_0)$, para algún valor intermedio t_0 perteneciente a $(0,1)$, que está en la misma "visual" que Q, pero más cerca del observador, como el ejemplo de la figura 1.

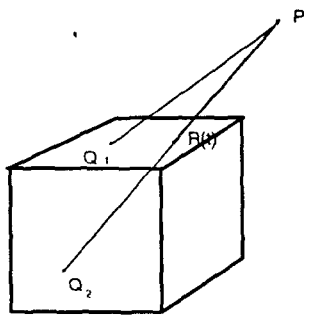


Figure 1

Esta sencilla definición no muestra la dificultad real del proceso. En primer lugar, debe estar claro que se trata de un problema de **cálculo de intersecciones**. Por otra parte, se comprende que no podemos determinar la condición de visibilidad para cada punto de una escena, pues, aparte de que contiene infinitos, incluso realizando una selcción de unos pocos miles de ellos, la comprobación de la condición anterior sería inaceptablemente lenta. Observen que averiguar si un segmento intersecta al objeto S, el cual puede estar compuesto por miles de polígonos o de superficies paramétricas es,

de por sí, una tarea que conlleva un cálculo desmesurado, que no podemos permitirnos sólo para averiguar la visibilidad de un punto.

Esto quiere decir que los algoritmos encaminados a resolver el problema de visibilidad no van a ser en absoluto simples y que necesitarán sacar el máximo partido a propiedades específicas de cada escena en concreto. Como veremos, una primera característica es que existen métodos exclusivos para superficies poliédricas, mientras que otros estarán diseñados específicamente para cuádricas o paramétricas.

5.1.2 Contorno aparente

Existe una propiedad que cumplen la mayoría de los objetos poliédricos (sobre todo los convexos) que puede sernos muy útil. Sea N el vector normal hacia fuera de uno de los polígonos o caras del objeto, y sea Q un punto del interior de dicha cara. Se cumple siempre que para que la cara sea visible debe cumplir

$$(P - Q) \cdot N > 0$$

Esto puede verse en la figura 2.

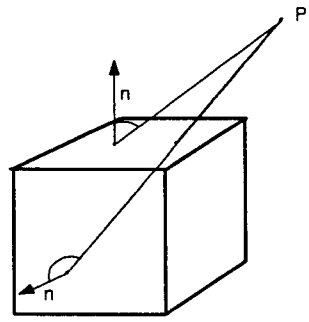


Figure 2

Observen que esta condición es totalmente lógica. Por un lado, el vector $P - Q$ representa la dirección de vista en la que está mirando el observador. Por otro lado, el vector N es la normal al polígono, e indica la dirección a la que la cara se esta enfrentando. Resulta evidente ver que si el observador está viendo dicho polígono en la realidad (es decir, la cara de delante del polígono, pues la cara de detrás no cuenta porque apunta hacia el interior del objeto) es porque la normal apunta hacia el plano de visión del observador y no hacia atras, por lo que el ángulo entre la normal y la dirección de vista debe ser agudo, y por lo tanto, el producto escalar de ambos vectores debe ser positivo[1].

[1] El producto escalar de dos vectores, A y B, se define como

$$A \cdot B = |A| \cdot |B| \cdot \cos \alpha$$

Para ser más exactos, diremos que la condición sólo se cumple en un sentido. Es decir, todas las caras que son visibles cumplen esta condición, pero no todas las caras que cumplen la condición son visibles, ya que puede haber otros polígonos entre la cara y el observador. Por ello, esta condición es muy útil para objetos convexos, porque en esos casos si se cumple la condición en ambos sentidos.

Esta propiedad se cumple también en muchas superficies curvas. En la figura 3 se ve el ejemplo de una esfera. Esta condición juega un papel importante en la teoría de la visibilidad. Fijémonos en la figura el ángulo que forma algunos puntos de la supefircie de la esfera entre el vector normal y el de la dirección de vista. Para Q_1 el ángulo es menor que 90, y por lo tanto es visible. Para Q_2 el ángulo es superior, y no es visible. Los puntos denotados por Q_c son aquéllos para los cuales el ángulo es exactamente 90^o y por tanto el producto $(P-Q)N = 0$. Estos puntos definen una línea de puntos visibles que delimitan la frontera entre el conjunto de puntos visibles e invisibles. Para el caso de la esfera, esta línea es una circunferencia, que recibe el nombre de **contorno aparente**.

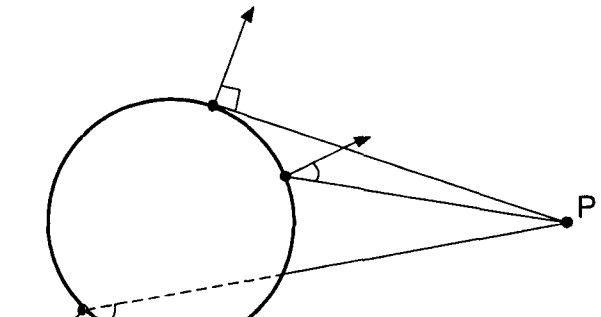


Figure 3

Para un objeto genérico S de $\Re^3$ en donde todos los puntos de su superficie posean un vector normal, entonces el contorno aparente está formado por los puntos Q_c que, siendo visibles, verifican

$$N \cdot (P - Q_c) = 0$$

Es decir, todos aquellos puntos en los que la recta PQ_c está contenida en el plano tangente a la superficie en Q_c. En la figura 4 puede verse el contorno aparente para el ejemplo de la esfera.

Generalmente, también se consideran pertenecientes al contorno aparente los puntos frontera que son visibles. En la figura 5 se ha representado una figura regular vista por un observador situado frente al texto y su contorno aparente, incluyendo los puntos frontera (en este caso son todos visibles).

---

donde $|A|$ y $|B|$ son los módulos de ambos vectores, y α es el ángulo que forman entre ellos.

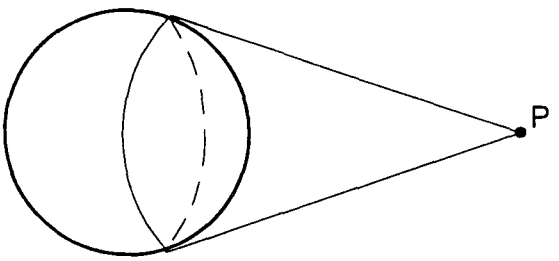


Figure 4

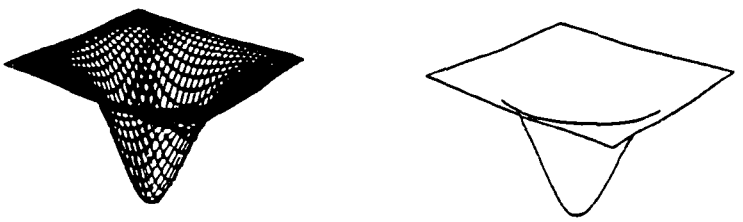


Figure 5

Para una superficie poliédrica el contorno aparente no puede definirse en términos del vector normal, pues éste sólo existe en el interior de las caras y en un punto interior a una cara no puede pasarse de una zona visible a otra invisible (a menos que esté parcialmente oculta por otro polígono, en cuyo caso sería este último el que definiría el contorno, o parte de él). Por tanto, los puntos del contorno aparente sólo se producen sobre las aristas y no merece la pena entrar en detalles, pues va a quedar automáticamente representado una vez resuelta la visibilidad para la superficie poliédrica, como se aprecia en la figura 6.

Figure 6

En realidad, el problema ocurre con las superficies regulares, en las que, a menos que se calcule explícitamente, no quedará correctamente dibujado. Un ejemplo podemos verlo en la figura 7, que muestra el problema que aparece cuando la superficie se trata de representar mediante conjuntos de secciones: el conjunto aparente no es una de tales secciones y no queda, por tanto, dibujado. Para el caso de la esfera, calcular la expresión

para el contorno aparente no es muy complicado, pero en general y para superficies más complejas, la determinación analítica del contorno aparente es virtualmente imposible.

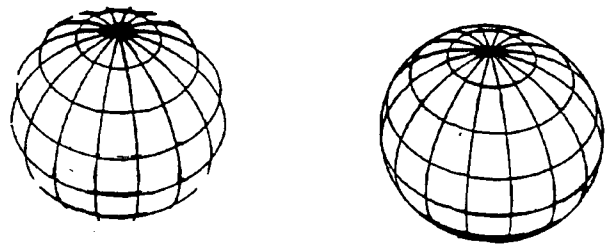


Figure 7

5.1.3 Parte anterior y posterior

El concepto de parte anterior y posterior de una superficie es de gran importancia en visibilidad y está íntimamente relacionado con el de contorno aparente. Es igualmente, muy difícil de definir rigurosamente, pero aquí nos bastará con una aproximación. Curiosamente, es más cómodo (y más útil en la práctica) caracterizar la parte posterior de un objeto (precisamente la que no ve el observador).

Consideremos un punto Q invisible. Por definición, esto requiere decir que el segmento abierto PQ intersecta a la propia superficie. Supongamos que lo hace en un número finito de puntos. Si este número es impar, entonces Q pertenece a la parte posterior de la superficie (como se ve en la figura 8 para el punto Q_1). En caso contrario, siendo Q visible o invisible indistintamente, so el número de intersecciones es par, entonces Q pertenece a la parte anterior (como Q_2).

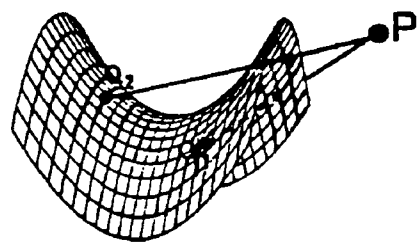


Figure 8

Conviene resaltar, aunque sea obvio a partir de la definición, que los puntos de la parte posterior son siempre invisibles, mientras que los de la anterior pueden ser tanto visibles como no serlo. Es decir, la parte anterior es la que "encara" al observador.

Sin embargo, no todos sus puntos son necesariamente visibles, pues pueden quedar ocultos por zonas de la propia superficie que se interpongan entre ellos y el observador. En cambio, la parte posterior es la que "encara" en dirección contraria y sus puntos no pueden verse, pues existen zonas de la superficie interpuestas (que son de la parte anterior).

El concepto de la parte posterior es muy útil siempre y cuando no empleemos directamente su definición. De hecho, no se trata de elegir un punto invisible sobre una superficie y determinar si pertenece o no a la parte posterior, sino al contrario: obtener la parte posterior de la superficie mediante métodos indirectos, para deducir que toda ella es invisible. Si la superficie posee vector normal en todos sus puntos, entonces es muy sencillo decidir si cualquiera de ellos es invisible por pertenecer a la cara posterior. La condición para un punto Q es

$$N \cdot (P - Q) < 0$$

La ecuación del plano tangente a la superficie en Q es

$$n_x(x - x_0) + n_y(y - y_0) + n_z(z - z_0) = 0$$

siendo $N = (n_x, n_y, n_z)$ y $Q = (x_0, y_0, z_0)$. Si sustituimos en la ecuación las coordenadas del observador, $P = (a, b, c)$:

$$n_x(a - x_0) + n_y(b - y_0) + n_z(c - z_0) = N \cdot (P - Q)$$

Por tanto, podemos concluir que un punto Q es de la parte posterior (y en consecuencia es invisible), si al sustituir las coordenadas del observador en la ecuación del plano tangente, se obtiene un valor negativo.

Para una superficie poliédrica, la definición de parte anterior y posterior es válida, pero aquí se pueden presentar problemas más a menudo si un polígono lo ve el observador de perfil. Esto ocurre precisamente cuando $N(P - Q) = 0$, (en cuyo caso sucede para todos los puntos de este polígono) y es sólo una cuestión de conveniencia establecer si la cara se considera anterior o posterior. Generalmente, los algoritmos de visibilidad simplemente descartan el polígono, como si no existiese. Exceptuando estos casos degenerados, todo polígono verifica que el signo de $N(P - Q)$ es constante cualquiera que seal el punto Q de su plano. Por lo tanto se sigue manteniendo la condición de que al sustituir las coordenadas del observador en la expresión del plano tangente, si ésta da negativo, el polígono completo se etiqueta como perteneciente a la parte posterior del objeto, ya que todos sus puntos serán invisibles.

Para poder evaluar esta condición necesitaremos que las ecuaciones de los planos que contienen a cada polígono estén disponibles a partir del modelador. Así, el único coste de la comprobación consiste en calcular el signo de la expresión

$$Aa + Bb + Cd + D$$

siendo $Ax+By+Cz+D = 0$ la ecuación de un plano, lo cual requiere sólo 3 productos y 3 sumas, más una comparación. La eliminación de polígonos de una superficie poliédrica mediante detección de que pertenezcan a la parte posterior se considera como un paso previo a la aplicación de cualquier algoritmo de visibilidad. Típicamente divide por dos el número de polígonos totales, lo que generalmente se traduce en dividir por cuatro el coste total requerido para resolver el problema de visibilidad.

Existe una propiedad muy interesante de coherencia que poseen los objetos convexos y es que todos los puntos de la superficie anterior son visibles. Quiere esto decir que el test de comprobación del signo de la expresión anterior aplicado a una cara decide unívocamente si ésta es completamente visible o completamente invisible. Esto constituye un procedimiento extraordinariamente rápido para resolver la visibilidad de uno de estos objetos. En la práctica, sin embargo, raramente se necesita representar un sólido convexo aislado, pero es conveniente tenerlo en cuenta para los que sí sean. Po otro lado, cualquier sólido no convexo puede subdivirse en varios que sí lo sean, y aplicar la técnica a cada uno de los nuevos sólidos generados.

Curiosamente, existen muchos objetos no convexos que presentan esta propiedad de coherencia, dependiendo de la posición del observador. Por ejemplo, en la figura 9 se muestra un mismo sólido visto desde dos puntos diferentes. En a) todas las caras anteriores son visibles y por tanto puede representarse simplemente eliminando las caras posteriores y dibujando las anteriores. Sin embargo, en b) existen caras anteriores (cara A), que no son totalmente visibles y no pueden, en consecuencia, ser dibujadas directamente.

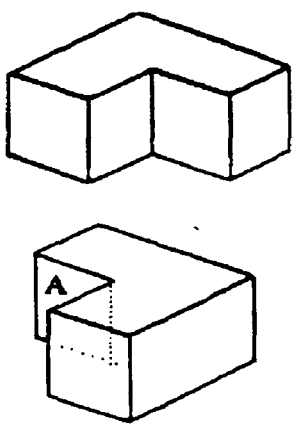


Figure 9

En la figura 10 se observa otro objeto con esta misma propiedad dependiendo del punto de vista. Se han representado las caras anteriores en su totalidad, lo que ha resuelto el problema de visibilidad.

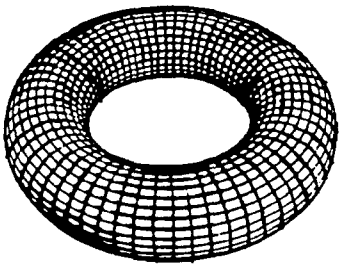


Figure 10

5.1.4 Prioridad. Técnica de parcelación

El concepto de prioridad es uno de los más importantes en visibilidad. Ahonda en las propiedades topológicas de una escena en general y de sus objetos constituyentes. Por una parte, permite tratar escenas compuestas por objetos heterogéneos, cada uno procesado con un algoritmo de visibilidad adaptado a sus características, y por otra, en un mismo algoritmo, acelera considerablemente los cálculos, al reducir de forma drástica las comparaciones y la determinación de intersecciones. La idea subyacente en prioridad queda recogida en la figura 11.

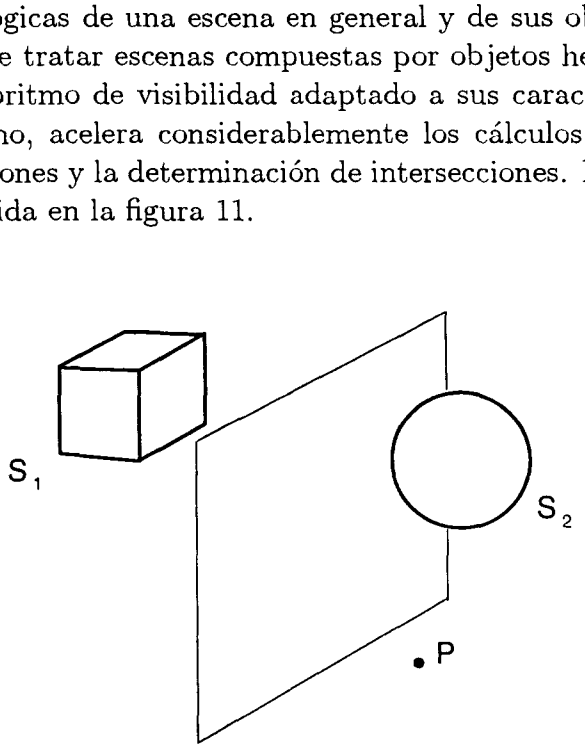


Figure 11

En ella se muestran dos objetos, S_1 y S_2, y un plano de separación entre ellos; esto es, un plano que divide el espacio en dos partes denominadas parcelas o **clusters**, cada una de las cuales contiene totalmente a uno de los objetos. Fijado entonces un observador P, el semiespacio en el que está situado define la prioridad de S_1 y S_2 respecto a él, estableciendo que S_i tiene mayor prioridad que S_j si P pertenece a la misma parcela que S_i, es decir, si ambos, el observador y el objeto S_i, están en el mismo semiespacio. Por ejemplo, en la figura anterior, S_2 tendrá mayor prioridad que S_1. La

utilidad del concepto de prioridad quedará ahora aclarada: ningún punto de S_2 puede ser ocultado por algún punto de S_1. Por tanto, la visibilidad del objeto S_2 es local y no depende en absoluto de S_1. Quiere esto decir que para determinar si un punto Q de S_2 es visible, el segmento abierto QP sólo debe ser verificado contra puntos de la propia superficie, eliminando con ello una gran cantidad de cálculo.

Naturalmente, la mayoría de los algoritmos no se basan en la idea primitiva de comparar el segmento que une un punto con el observador con los restantes puntos; pero el argumento anterior pone de manifiesto cómo un uso inteligente de este concepto permitiría, no sólo acelerar la resolución del proceso de visibilidad reduciendo dramáticamente el cálculo de intersecciones, sino incluso tratar los objetos S_1 y S_2 con algoritmos distintos. Esta técnica puede ser utilizada, en general, con cualquier algoritmo de visibilidad, principalmente en combinación con el método del pintor que describiremos más adelante.

La situación de la figura anterior puede generalizarse a un conjunto cualquiera de n objetos: si existen planos de separación entre ellos, la escena se dice parcelada o dividida en clusters. Una vez fijada la posición del observador y determinada su situación en relación con los planos, puede asignarse una prioridad (un número) a cada objeto o conjunto de ellos, de forma que aquéllos con mayor prioridad nunca podrán ser ocultados, ni en parte ni en todo, por otros con menor prioridad. En un caso extremo, cada polígono de una escena puede llegar a definir un plano distinto de separación; quedando ésta subdividida en cientos o quizás miles de clusters, y teniendo cada uno de los polígonos una prioridad distinta.

Hay que hacer notar que la técnica de parcelación puede considerarse un proceso a priori. En efecto, aparte de la dificultad de obtener los planos de parcelación, si existen, éstos son calculados en la fase de modelado; es decir, son absolutamente independientes del observador.

El principal problema que puede presentarse es la no existencia de planos de separación, sobre todo cuando un algoritmo de visibilidad concreto intenta llegar al límite de separar la escena en clusters divididos por los propios planos que contienen a cada uno de los polígonos. En la figura 12 se muestran dos situaciones típicas que imposibilitan esta subdivisión y cada una es un ejemplo de una propiedad, que al no cumplirse, impide que la clasificación por prioridades sea una relación de orden.

En a), el polígono A oculta en parte a B, y éste a su vez oculta a C. Si la relación de prioridad fuera transitiva, debería implicarse que el polígono A oculta a C, cuando en realidad ocurre lo contrario. En b) se refleja la no antisimetría: los objetos A y B mutuamente se ocultan en parte. La resolución de estos problemas pasa frecuentemente, bien por renunciar a separar físicamente estos objetos, o bien por seccionar algunos polígonos en dos partes por el plano que contiene a uno de ellos.

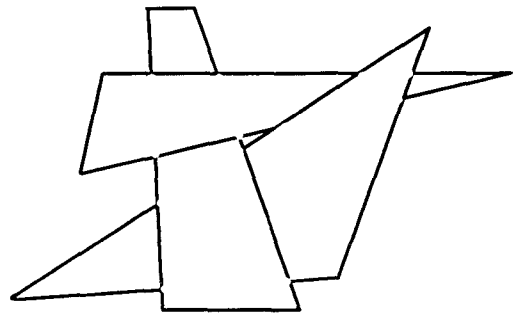


Figure 12

5.1.5 Método del pintor

Es un procedimiento de resolución automática del problema de visibilidad cuando se conocen las prioridades de un conjunto de polígonos; es decir, su ordenación relativa respecto al observador. Se denomina así porque sigue la misma técnica que un pintor para representar los objetos que solapan sobre el lienzo; pero cuyas profundidades o distancias al observador (en este caso el propio pintor) son aproximadamente conocidas. No importa en este caso saber con exactitud cuál es la distancia precisa de un árbol, una casa o una montaña. Lo esencial aquí es que la montaña, por ejemplo, está máslejos que la casa y ésta a su vez está más lejos que el árbol. Es decir, lo importante es conocer las prioridades de estos tres objetos. En este caso, la montaña sería completamente dibujada en primer lugar (figura 13.a), a continuación, con los colores apropiados, se pintaría encima la casa, ocultando automáticamente los colores de la montaña en aquellas zonas en las que ambos objetos solapen (figura 13.b). Finalmente, el árbol se dibujaría como una tercera capa de pintura en los lugares previamente ocupados por la casa (figura 13.c). En determinados puntos del lienzo existirían simultáneamente hasta tres colores; pero sólo el pintado en último lugar sería visible en la imagen definitiva.

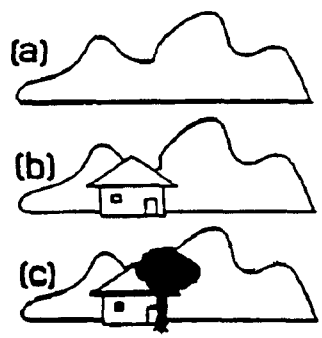


Figure 13

La aplicación del método del pintor al problema de visibilidad es entonces obvia si los objetos están clasificados por prioridades: en primer lugar se dibuja el polígono con menor prioridad conectando los pixels apropiados con su color; a continuación el siguiente polígono en orden de prioridad. Si en algún pixel solaparan, en él quedaría el color del segundo polígono. Así se continúa hasta el último polígono, el que está más cercano al observador y tiene, por tanto, mayor prioridad.

Existe un problema con esta técnica, que es traducción directa de otro que aparece en el caso del pintor real: en general, cualquier punto sobre el lienzo ha sido pintado varias veces, con colores distintos, permaneciendo sólo el dibujado en último lugar. Esto significa que se ha utilizado (gastado) mucha más pintura de la necesaria a cambio de no tener que trazar la silueta de unos objetos respecto a otros. Por otra parte, si una determinada zona ha recibido, por ejemplo, cinco capas de color, en realidad se ha empleado cinco veces más tiempo en dibujarla que si se hubiera conocido de antemano su color final.

En nuestro caso, la analogía es sencilla: a cambio de no tener que calcular las múltiples intersecciones entre diferentes polígonos de una escena (para determinar cuál oculta a cuál, o lo que es lo mismo, cuál es la silueta de unos respecto a otros) será necesario dibujarlos todos por completo, aunque muchos pixels cambiarán sucesivamente su color hasta alcanzar el definitivo. Por ejemplo, considerando una pantalla de 1024×768 pixels, un algoritmo de visibilidad ordinario los conectaría una sóla vez, haciendo por tanto aproximadadmente 800000 llamadas a la función $pixel(x, y)$. Un algoritmo basado en el método del pintor que, en promedio, cambiara el color de cada pixel unas diez veces, necesitaría invocar la función ocho millones de veces. El problema más grave es, sin embargo que el cálculo del color de un polígono en un pixel puede llegar a ser muy complejo, sobre todo si se está utilizando un modelo de iluminación. Por tanto, debe contrapesarse si interesa multiplicar este cálculo, digamos por 10 (este dato es evidentemente desconocido a prior) a cambio de evitar el cálculo de intersecciónes.

En cualquier caso, el método del pintor es una excelente técnica en combinación con aquéllas basadas en prioridad. Como una última aplicación, citemos que si una escena está compuesta por objetos de distinto tipo, que necesitarían algoritmos diferentes para resolver su visibilidad y no podrían tratarse en consecuencia como un todo, si se conoce la prioridad relativa entre ellos, pueden dibujarse en sucesión, cada uno con su propio algoritmo, siguiendo el método del pintor.

5.1.6 Técnicas de comparación

Todos los algoritmos de visibilidad requieren la comparación entre diferentes objetos en una escena. Por ejemplo, en una superficie poliédrica, en general hay que descender hasta el extremo de tener que decidir si un punto es visible comparándolo con el resto de los objetos. Se trata entonces de procurar que el número de puntos que necesitan complejos cálculos de intersecciones y de toma de mútiples decisiones, se reduzca a un

mínimo. Esto se consigue comparando entre sí entidades mayores que un punto: lados, polígonos o incluso sólidos enteros. La forma más sencilla de implementar esta idea consiste en asociar a cada una de estas entidades un **volumen de inclusión**; es decir, una región -plana o tridimensional- que la acote por completo. Para comparar dos polígonos, por ejemplo, en realidad se comparan sus volúmenes de inclusión, operación que se supone es mucho más sencilla.

En la figura 14 los dos polígonos no pueden intersectar, dado que no lo hacen las correspondientes circunferencias (en este caso hemos elegido usar circunferencias como volúmenes de inclusión. Podría usarse cualquier otra figura, o incluso usar figuras distintas para cada objeto). Observemos que decidir si dos circunferencias intersectan se reduce a verificar si la distancia entre sus centros es menor o mayor que la suma de los respectivos radios.

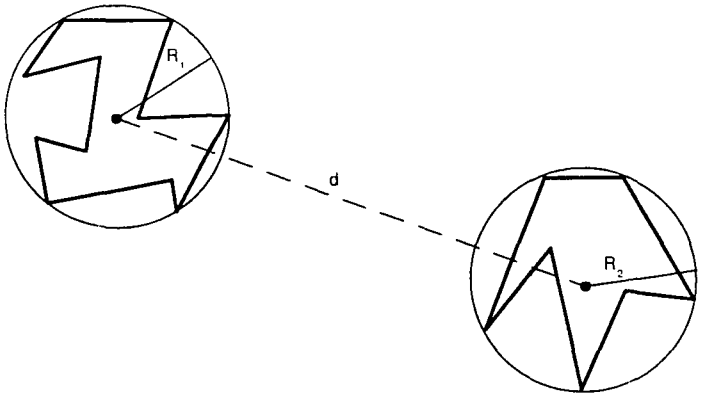


Figure 14

En el caso de que dos volúmenes de inclusión intersecten, nada puede decidirse y los polígonos podrían aún no intersectar, o sí hacerlo. Un test más complejo, a nivel de lados, debería ser aplicado para determinar cuál de las dos situaciones se produce. En la figura 15 se ve un ejemplo de cada caso.

La fuerza de la técnica está en que rápidamente se detectan objetos completos que no pueden intersectarse (y por tanto, ninguno oculta a otro, si la comparación se ha efectuado sobre el plano transformado por las ecuaciones de perspectiva). Para un objeto dado, por ejemplo un polígono, sólo aquéllos otros objetos (incluso sólidos) que estén suficientemente cerca como para que intersecten sus volúmenes de inclusión, deben ser considerados y pasados a rutinas de cálculo de intersecciones entre entidades más pequeñas (lados).

La determinación de volúmenes de inclusión es, de nuevo, una técnica de preprocesado que siempre tiene lugar en la fase de modelado (como una última etapa). Para superficies poliédricas, es común almacenar en la base de datos, junto con cada sólido

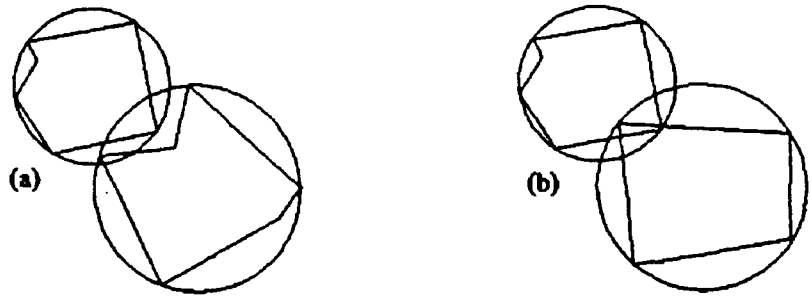


Figure 15

individual, sus coordenadas máxima y mínima, tanto en x como en y y en z. Estos valores determinan automáticamente un prisma de inclusión para el sólido que podría ser utilizado en comparaciones posteriores. A menos que existan problemas de almacenamiento, la anterior operación se realiza también para cada uno de los polígonos constituyentes de cada uno de los sólidos. Finalmente, si es posible, se almacena esta información incluso para cada lado. Evidentemente, cuanto mayor sea la jerarquización de los volúmenes de inclusión, mayor será la base de datos correspondiente a una escena; pero esto se traducirá indefectiblemente en una reducción considerable en los tiempos empleados para resolver el problema de visibilidad.

Pensemos que, en general, el número de operaciones requerido por un algoritmo de visibilidad es proporcional al cuadrado del número de objetos a comparar a nivel de cálculo de intersecciones. Si una escena contien 10.000 lados, este número será del orden de 100 millones de operaciones, si se compara directamente lado con lado. En cambio, si la escena está distribuida entre 10 sólidos, cada uno con aproximadamente 1000 lados, un primer test entre los sólidos requeriría sólo 100 operaciones. Si de él se dedujera que un lado determinado sólo puede ser intersectado por, digamos dos sólidos, éste sólo tendría que ser comparado con unos 2000 lados. Como regla general, es conveniente darse cuenta de que cada vez que, mediante cualquier técnica, se reduce a la mitad el número de objetos a comparar, el número total de operaciones desciende hasta la cuarta parte.

5.1.7 Clasificación de los algoritmos de visibilidad

Como ya hemos visto, salvo en casos excepcionales, no es posible determinar la visibilidad de puntos sobre una superficie utilizando directamente la definición. Debemos, por tanto, idear métodos que profundicen en la estructura de los objetos y descubran relaciones geométricas e incluso topológicas que permitan, al menos en ciertos casos, resolver el problema.

En primer lugar, tengamos en cuenta que la definición de visibilidad, precisamente por ser rigurosa y aplicable a conjuntos arbitrarios de puntos, no toma en consideración

el hecho de que las superficies en las que estamos interesados no son conjuntos cualesquiera de $\Re^3$, sino que poseen propiedades tales como continuidad, convexidad, etc. Esto quiere decir que si, por ejemplo, averiguamos que un punto P es visible, entonces los puntos de su entorno también lo serán (a menos que P pertenezca al contorno aparente). Más aún, podemos asegurar que cualquier entorno de P, de la forma que sea, que no corte al contorno aparente, estará compuesto por puntos visibles.

Si bien es cierto que en general es imposible obtener el contorno aparente, e incluso saber si un punto pertenece o no a él, la observación anterior pone de manifiesto que a veces puede ser posible determinar simultáneamente la visibilidad o invisibilidad no ya de puntos aislados, sino de trozos enteros de superficie. Un caso extremo de explotación de esta propiedad se produce para los objetos convexos donde, como hemos visto, podemos dividir la superficie exactamente en dos partes y decidir, con un mínimo de cálculo, que todos los puntos de una de las dos partes son visibles y todos los de la otra, invisibles.

Este tipo de propiedades de una superficie se denomina **coherencia** y cada algoritmo está pensado para sacar el máximo partido a una o varias formas de coherencia, dependiendo de aquéllas que presente una superficie determinada. Por ejemplo, en un objeto poliédrico, todos los puntos de una cara son coherentes en el sentido de que pertenecen a un mismo plano. La manera más sencilla de aprovechar esta forma de coherencia es observar que un punto de un plano nunca puede ocultar a otro punto del mismo plano (a menos que éste se vea de perfil). Debe estar claro, por tanto, que no existe un algoritmo universal para el problema de visibilidad (realmente sí existe pero es impracticable) y que cada clase de superficie posee sus propios algoritmos, que obtienen máximo rendimiento del aprovechamiento de propiedades de coherencia específicas.

Un problema previo al de visibilidad, pero muy relacionado con él, es decidir qué puntos de una superficie deben ser representados para que un observador real pueda restituir su forma y tenga la sensación de que efectivamente está viendo el objeto que sólo existe como modelo matemático. Los primeros algoritmos de dibujo asistido por ordenador representaban conjuntos de secciones, de forma análoga a como habitualmente se hace a mano para dibujar una superficie regular. De igual modo, los objetos poliédricos se representaban mediante sus aristas. Dos ejemplos de estos casos pueden verse en la figura 16.

Éste es un método muy interesante que aprovecha la capacidad de la mente para integrar y reconocer formas sólo a partir de sus contornos y ha sido ampliamente utilizado en dibujo técnico para la definición sobre un plano de objetos tridimensionales. Para representar en ordenador una superficie mediante secciones o aristas, un algoritmo de visibilidad debe determinar qué parte de cada línea en cuestión es visible o invisible. Este tipo de algoritmos se denomina, por tanto, de **eliminación de líneas ocultas** y, en el caso de superficies regulares, no resuleven totalmente el problema, principalmente en lo que al contorno aparente se refiere, pero pueden conseguir una muy buena aproximación. Las figuras anteriores han sido realizadas mediante dos de estos algoritmos.

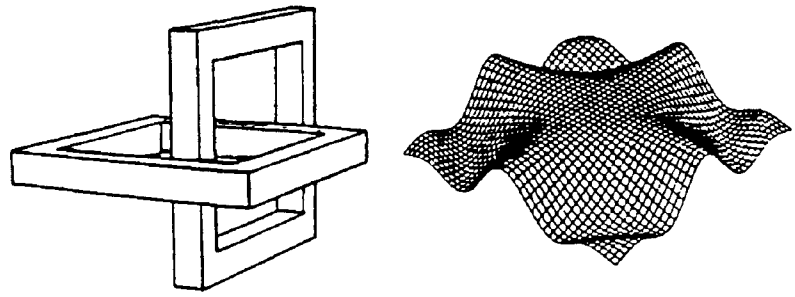


Figure 16

La aparición y posterior difusión de monitores, al principio en blanco y negro, y después en color, pero en cualquier caso con la disponibilidad de diferentes niveles de intensidad en cada pixel, abrió la posibilidad de obtener imágenens con mayor grado de realismo. Para ello se desarrollaron rápidamente procedimientos para simular cómo una persona pinta, más bien que dibuja, una escena utilizando diferentes colores y tonalidades. Ahora lo que se pretende es representar la forma en sí, mediante manchas de color. En la figura 17 puede verse la diferencia entre dibujar las aristas sin considerar las caras, o por el contrario representar cada una con un color o intensidad de gris distinta, en función de la luz que reciba. De este modo, un observador comprende perfectamente su forma e incluso adivina las aristas (aunque no estén dibujadas físicamente) debido a la variación brusca de intensidad (discontinuidad) que se produce en ellas.

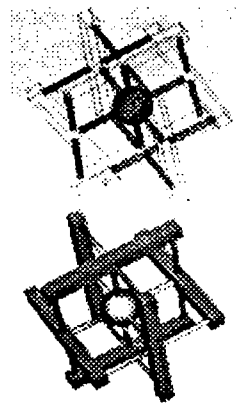


Figure 17

Los algoritmos que trabajan de esta manera, se denominan de **eliminación de superficies ocultas**, indicando con ello que operan globalmente con trozos de supeficie (quizás tan pequeños como un pixel) para determinar su visibilidad y, en su caso, eliminarlos. Estos dos tipos conforman una categoría en la cual clasificar los distintos algoritmos de visibilidad.

Por otra parte, debido a las propias características del proceso de visualización, un algoritmo de visibilidad puede operar directamente con el modelo matemático 3D, utilizando por tanto, coordenadas del mundo real, o sobre el modelo transformado por las ecuaciones de la perspectiva y encuadre, trabajando entonces en 2D y en un sistema de referencia con coordenadas enteras. Los del primer grupo se denominan **algoritmos espaciales** o de **precisión a nivel de objeto**. Realizan los cálculos en aritmética real y, al determinar la visibilidad sobre el modelo 3D, sólo se necesita transformar (perspectiva, transformación de vista) la parte visible.

Por su parte, los algoritmos del segundo grupo requieren transformar toda la escena (con el conste computacional que esto pueda traer consigo), pero tienen la indudable ventaja de reducir los cálculos de visibilidad a dos dimensiones e incluso trabajar en aritmética entera. Éstos se denominan **algoritmos de imagen** o de **precisión a nivel de imagen**. En líneas generales, puede decirse que el tiempo de ejecución de un algoritmo espacial depende de la complejidad de la escena tridimensional, mientras que el de uno de tipo imagen depende principalmente de la complejidad de la parte visible.

5.2 ELIMINACIÓN DE LÍNEAS OCULTAS

El problema de la visibilidad es complejo y costoso, y obtener una escena dibujada con líneas ocultas eliminadas puede necesitar entre diez y cien veces más tiempo que la misma escena representada con todas sus líneas sin eliminar. También debe tenerse en cuenta, principalmente a nivel de implementación, que en un modelo relativamente complicado, se presentarán con seguridad todos los casos degenerados que teóricamente puedan suceder. Por ejemplo, para escenas poliédricas compuestas por miles de polígonos, es casi seguro que alguno de ellos aparecerá de perfil para el observador. Igualmente, puede afirmarse que habrá vértices situados en una misma visual, de modo que en el plano de proyección se representarán sobre el mismo punto. Un algoritmo robusto debe tratar correctamente con estas situaciones, en orden a que no se produzcan defectos inaceptables en la imagen final.

5.2.1 Algoritmo del horizonte flotante.

Este algoritmo trabaja sobre superficies que puedan expresarse mediante una ecuación explícita:

$$z = f(x, y)$$

donde $a \leq x \leq b$ y $c \leq y \leq d$. En la figura 18 se ve un ejemplo, donde la función se encuentra definida para los puntos del interior del rectángulo de vértices (a, c) y (b, d).

Aunque existen muchas variantes, vamos a considerar sólamente la que permite representar secciónes de x constante y de y constante, es decir, secciones paralelas a los planos yz y xz. El efecto visual producido es el de una malla deformada por el

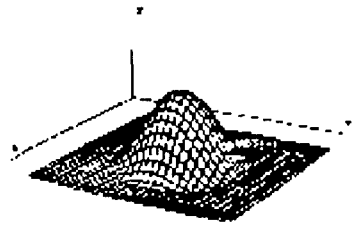


Figure 18

volumen de la superficie, consiguiéndose así una percepción aceptable de su forma y de sus características.

El algoritmo es de tipo imagen: opera sobre las proyecciones de todos los puntos situados sobre las secciones a representar. Dado que el cálculo de la función $f(x, y)$ puede no ser sencillo, se produce un gasto notable de tiempo en el procesado de puntos que después resultarán ser invisibles. El principio fundamental del algoritmo reside en la ordenación automática por prioridades que presenta cada conjunto de secciones en relación al observador. En la figura 19 se plantea la situación, habiéndose dibujado algunas secciones de x constante (que aparecen como líneas rectas, pues están vistas desde arriba) y el observador P.

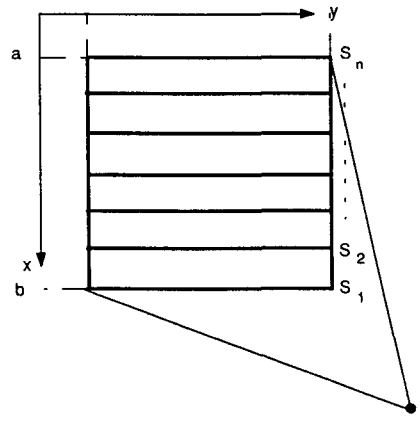


Figure 19

Asímismo, se pone de manifiesto la prioridad de los planos verticales que contienen a las secciones, estando numerada como S_1 la más cercana al observador P (la de máxima prioridad), S_2 la siguiente y así sucesivamente. Por tanto, el algoritmo utiliza implícitamente la técnica de parcelación. No es conveniente, sin embargo, emplear el método del pintor, pues necesitaríamos para ello información de color en cada curva, que se transmitiría automáticamente al espacio entre dos de ellas. El algoritmo obtiene

partido inmediato de la clasificación por prioridades: la sección S_1 es siempre visible y puede ser dibujada por completo. La sección S_2 sólo puede ser ocultada, si lo es, por la anterior S_1. Por tanto, basta compararla con ella. La tercera sección, S_3, sólo puede ser ocultada por S_1 y S_2. Así sucesivamente, como se ve en la figura 19.

Pero veamos el problema con más detalle. Si consideramos el polígono cuyo lado superior es la propia curva y sus restantes tres lados son como en la figura 20, para que un punto sobre la curva S_2 quede oculto, la visual que lo une con el observador debe intersectar el plano que contiene a S_1. Una vez transformados todos los puntos por las ecuaciones de perspectiva y transformación de vista, este hecho se pondría de manifiesto porque el punto en cuestión sobre S_2 tendría menor ordenada (en el sistema de referencia de la pantalla) que el punto S_1 con la misma abscisa.

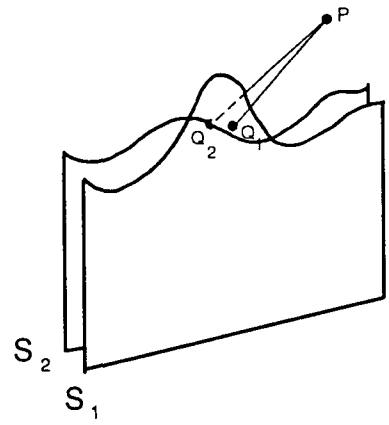


Figure 20

Por la misma razón, todos los puntos de S_2 que, una vez transformados, tengan mayor ordenada que el correspondiente sobre S_1 de igual abcisa, deben ser visibles. Esto proporciona una método teóricamente muy sencillo de determinar la visibilidad de la segunda sección.

Con la tercera sección las cosas no son tan fáciles; pero aún podremos arreglarlo: un punto sobre S_3 será invisible si en el plano de la pantalla tiene ordenada menor que el punto con igual abcisa en cualquiera de las dos curvas previamente dibujadas; es decir, si su ordenada es menor que la máxima ordenada dibujada hasta ahora con esa misma abcisa. Este mismo razonamiento es válido para las siguientes secciones: un punto sobre la sección S_i es invisible si, en el plano de la pantalla, su ordenada es menor que la máxima ordenada dibujada correspondiente a su misma abcisa. De esta forma, el algoritmo progresa de delante hacia atrás manteniendo en todo momento dibujada una curva, que no corresponde realmente a ninguna sección, compuesta por las máximas ordenadas calculadas en cada abcisa. Esta curva es la silueta o contorno aparente de la parte de superficie procesada y es común denominarla **horizonte**, de ahí el nombre

del algoritmo, pues en cada sección queda calculado un nuevo horizonte, que fluctúa en las distintas etapas.

En la práctica es imposible determinar con precisión absoluta si un punto es visible o no. De hecho, no podemos comprobar si la ordenada del punto en cuestión es mayor o menor que la máxima ordenada dibujada en la misma abcisa, sencillamente porque calcular esto requeriría disponer de la ecuación de la curva del horizonte, y esto evidentemente no es factible. Por tanto, necesitamos hacer una aproximación que se traduzca en una tabla de valores para dicha curva, que debe ser inicializada con los valores correspondientes al horizonte original (sección S_1) y se actualice en cada etapa. Por ejemplo, podemos definir un vector con tantas componentes como la resolución horizontal del monitor

$$H[0], H[1], ..., H[n-1]$$

Este vector se inicializa a ceros, de forma que al procesar la primera sección, todos sus puntos resulten visibles. La ordenada de cualquiera de ellos se almacena en la componente cuyo subíndice sea más próximo a la correspondiente abcisa. En cualquier etapa se procede comparando la ordenada y de un punto con el valor almacenado en $H[(int)x]$, que contendrá la máxima ordenada calculada en la abcisa entera más próxima. Si es menor, el punto se declara invisible y se continúa con el siguiente sobre la misma sección. En cambio, si es mayor, no sólo se dibuja el punto (por ser visible), sino que su ordenada pasa a ser considerada la máxima correspondiente a la abcisa entera asociada y se almacena en el vector H. De esta forma, al finalizar el proceso de una sección, el vector H contiene una aproximación discreta a la curva del horizonte, como se ve en la figura 21.

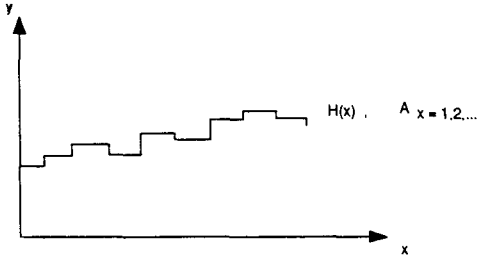


Figure 21

La implementación del algoritmo requiere prevenir algunos problemas que se presentan en la práctica. El primero y más evidente es que, cuando se calculan puntos sucesivamente sobre una sección, una vez proyectados, sus abcisas no sean consecutivas (por ejemplo, dos puntos visibles consecutivos pueden tener por coordenadas (x_1, y_1) y $(x_1 + 10, y_2)$. Esto implica que no sólo deben actualizarse los valores del vector H en las componentes x_1 y $x_1 + 10$, sino en todas las intermedias. Pero estos valores son a

su vez desconocidos. La forma más fácil de solventar este problema consiste en realizar una nueva aproximación: interpolar linealmente entre las ordenadas y_1 e y_2.

En la figura 22.a) se muestra el resultado obtenido de la aplicación del algoritmo tal y como ha sido descrito. Es relativamente pobre, debido a que con el dibujo sólo de las secciones de x constante es muy difícil comprender la forma de la superficie. Esto tiene fácil arreglo: volver a aplicar el algoritmo pero para las zonas de y constante. Superponiendo ambas imágenes obtendremos un mejor resultado (figura 22.b).

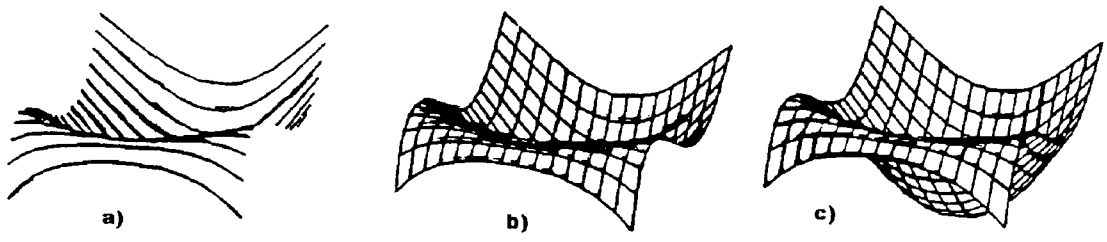


Figure 22

Un último problema queda aún por resolver, pues la imagen aún está inacabada: faltan zonas de la superficie que deberían ser visibles y que han sido omitidas al quedar por debajo del horizonte en cada etapa. La razón de esto es la siguiente: la clase de superficie que estamos representando poseen dos caras: la superior, que puede caracterizarse porque el vector normal tiene z positiva, y la inferior en la que ocurre lo contrario. Podemos imaginar esto pensando que la superficie tiene grosor infinitesimal y cada una de sus dos caras está pintada de un color. El algoritmo tal y como se ha descrito sólo dibujaría la cara superior.

Afortunadamente es muy fácil corregir esta situación. Revisando la idea en la que se basaba y fijándonos ahora en la cara inferior, podemos establecer la siguiente condición: un punto sobre una sección es visible si su ordenada es menor que la mínima ordenada previamente dibujada. por tanto en orden a implementar el algoritmo para que también contemple la cara inferior, necesitaremos otro vector G que, en esta ocasión, almacene los valores mínimos calculados hasta el momento y que sería inicializado a la máxima ordenada posible. El test completo de visibilidad quedaría

$$(x, y) \text{ es visible si } y > H[x] \text{ ó } y < G[x]$$

El resultado final se muestra en la figura 22.c). A continuación en la figura 23 se muestran varias superficies dibujadas con este algoritmo.

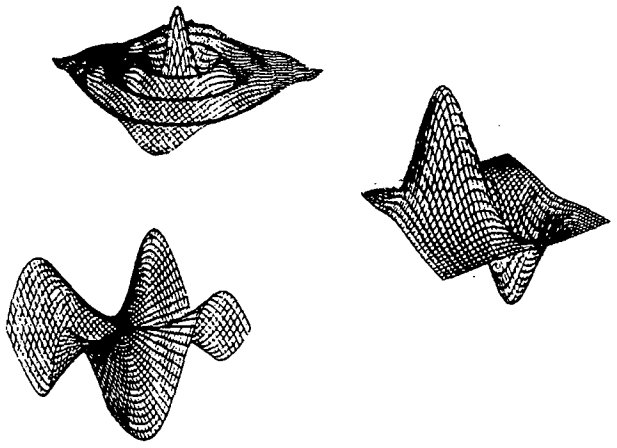


Figure 23

5.2.2 Eliminación para superficies poliédricas

Dado que mediante superficies poliédricas puede modelarse prácticamente cualquier tipo de objeto (incluso superficies paramétricas si son aproximadas con suficiente precisión), es lógico pensar que un considerable esfuerzo fuera dedicado, tras la aparición de los primeros dispositivos de dibujo, al problema de visibilidad para las aristas de cualquier polígono. El primero surgió en 1963, y sólo funcionaba para sólidos convexos. Desde entonces han surgido diferentes versiones para tratar escenas más generales, pero todas comparten en el fondo la técnica del algoritmo inicial, que es la que vamos a estudiar. El algoritmo genérico consta de 4 pasos secuenciales, los cuales iremos viendo con el ejemplo de la figura 24.

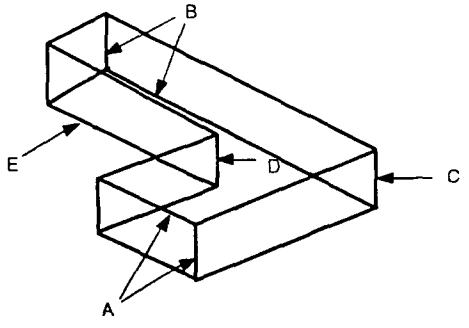


Figure 24

1. En primer lugar, se determinan las caras anteriores y posteriores. Recordemos que para ello bastaba con sustituir las coordenadas del observador en la ecuación de cada plano, la cual debe estar disponible desde la fase de modelado.

2. Para cada arista de la escena, se compara el carácter de los dos polígonos a los que pertenece:

2a) Si los dos son anteriores, la arista se declara **potencialmente visible** (aristas A en la figura) y necesitará ser estudiada en detalle más adelante.

2b) Si los dos son posteriores, la arista es **totalmente invisible** y se elimina de la estructura (aristas B en la figura).

2c) Si un polígono es anterior y el otro posterior, antes de decidir, debemos comprobar si se trata de una arista convexa o cóncava. En el primer caso, la arista se clasifica como potencialmente visible (C en la figura), mientras que en el segundo, se descarta al ser totalmente invisible (D en la figura).

En el resto del algoritmo sólo se toman en consideración las aristas declaradas como potencialmente visibles. Notemos que algunas, como A, resultarán ser totalmente visibles, pero otras, como E, constarán de tramos visibles y de otros invisibles.

3. A continuación se proyectan todos los vértices de la escena (incluso aquellos que son extremos de aristas declaradas invisibles). Evidentemente las ecuaciones empleadas deben mantener coordenadas (x, y, z), aunque en el momento de dibujar sólo se utilicen x e y. En lo que sigue llamaremos **lado** a lo que hasta ahora era una arista, entendiendo que un lado es un segmento bidimensional (la proyección de la arista).

4. Estudiar en sucesión las aristas potencialmente visibles (lados). Para cada lado:

4a) Comprobar si intersecta con alguno de los restantes lados. Esta etapa requiere una rutina de comparación de segmentos en 2D que decida rápidamente si no intersectan. En caso de existencia de intersección, los lados quedarán divididos en varios tramos, tantos como intersecciones se hayan encontrado (en la figura 25, el lado E se divide en los tramos E_1 y E_2). Lo importante ahora es que cada uno de estos tramos es totalmente visible o totalmente invisible. Basta por tanto con averiguar la visibilidad de uno de sus puntos.

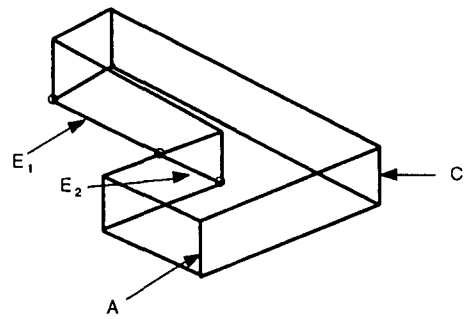


Figure 25

4b) Para cada uno de los tramos en que ha quedado subdividido el lado se calcula su punto medio. Este punto será invisible (y por lo tanto todo el tramo lo es) si y sólo si se cumplen las dos siguientes condiciones: el punto es interior a algún polígono (en el plano xy) y el plano que contiene en $\Re^3$ a este polígono separa en semiespacios distintos al punto y al observador. En caso contrario, el punto es visible y todo el tramo con él. Una vez procesado el último lado la imagen mostrará sólo aquella parte de cada arista que es realmente visible.

El corazón del algoritmo es por supuesto la fase 4b), en donde habrá que realizar un test de inclusión del punto para cada polígono de la escena. Supongamos una escena compuesta por 400 caras y 1000 aristas, y que por término medio cada una quede subdividida (en el plano imagen) en cinco tramos. En ese caso habría que comprobar la visibilidad de 5000 puntos. Uno cualquiera de ellos debe ser comparado con cada uno de los 400 polígonos, lo que produce aproximadamente 2 millones de tests de inclusión. Evidentemente, esta fase del algoritmo debe ser acelerada al máximo mediante las técnicas descritas anteriormente, las cuales pueden reducir el tiempo total de ejecución notablemente.

La segunda parte de la fase 4b) es mucho más sencilla: basta con sustituir las coordenadas (x, y, z) del punto en cuestión en la ecuación del plano del polígono y comprobar su signo. El punto estará en un semiespacio distinto que el observador si este signo es negativo.

La versión básica del algoritmo no permite algunas posibilidades interesantes de la escena. Por ejemplo, si se desa aproximar una superficie paramétrica por una poligonal para poder aplicarle esta técnica, algunos lados como los del borde de la superficie sólo pertenecerían a un polígono, circunstancia no contemplada. Otra posibilidad no prevista es la de dos sólidos que puedan interferir entre ellos, como los de la figura 26, que pueden aparecer cuando trabajemos con operadores booleanos como en CSG. Una solución puede ser crear el objeto formado por la unión de ambos antes de visualizar la imagen.

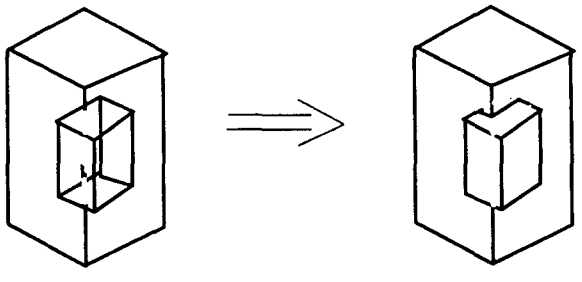


Figure 26

5.3 ELIMINACIÓN DE SUPERFICIES OCULTAS

Cuando un modelo debe ser representado con mayor grado de realismo que un algoritmo de líneas, necesitamos operar con "manchas" de color, del mismo modo que lo hace un pintor, aunque no necesariamente siguiendo su misma técnica.Exceptuando los métodos directamente basados en priordad, que en general trabajan con polígonos completos, y por tanto con zonas coloreadas relativamente grandes, los restantes algoritmos utilizan técnicas de tipo "puntillista"; esto es, se basan en la idea de que, en último extremo, la región del "lienzo" más pequeña que puede ser coloreada individualmente es un pixel. En consecuencia, se concentran en determinar cuál debería ser el color correcto en cada pixel, en orden a mostrar una imagen realista del modelo matemático.

Todos los algoritmos de eliminación de superficies ocultas constan de dos etapas. En primer lugar, se trata de detectar zonas de la superficie (quizás del tamaño de un pixel, o incluso menor) que son visibles. A continuación, y este problema no aparecía en los algoritmos de línea, debe decidirse de qué forma colorear estas zonas visibles. Esto, a primera vista, puede parecer una cuestión irrelevante. Pero imaginemos que deseamos obtener una imagen de un cubo. A menos que queramos pintar cada cara de un color distinto, el resultado será algo parecido al de la figura 27.

Figure 27

Por tanto, no podemos establecer simplemente que el color de un objeto es, digamos azul; pues sólo conseguiríamos una silueta rellena de este color. Una solución parcial a este problema sería asignar una tonalidad de azul distinta a cada cara; pero esto sería un procedimiento diseñado sobre la marcha para esta escena en concreto y tan artificial como numerar las caras del cubo y decidir a dedo que sólo deben ser visibles las marcadas con el 1, 2 y 5. Naturalmente, lo que se pretende es que el propio ordenador averigüe qué caras son visibles en función de la posición del observador,k y de la misma forma, decida qué tonalidad de azul le corresponde a cada una, de acuerdo con algún criterio que simule el proceso por el cual en un objeto de un único color, cada cara presenta en realidad una tonalidad distinta de ese color.

Este último enfoque, que es el más correcto, lo analizaremos con más detalle en el próximo capítulo, donde se estudiará la forma en la que la luz interacciona con los objetos produciendo la sensación al obsrevador de que están coloreados en mayor o menor medida. Por el momento, y una vez aclarada la cuestión, prescindiremos de este problema y supondremos que, una vez se ha determinado una zona visible en una

superficie, sabemos calcular su color (por el procedimiento que sea).

5.3.1 Algoritmo del Z-buffer

Fue desarrollado por E. Catmull en 1974, en una época en la que el coste de las memorias de ordenador era aún considerable, por lo que no estuvo excesivamente extendido debido a sus grandes necesidades de almacenamiento. Hoy en día y gracias a la disponibilidad a bajo precio de memorias con capacidad suficiente, incluso en equipos pequeños, se ha erigido en uno de los más populares, principalmente por su extrema simplicidad, que ho hace muy adecuado para ser implementado en hardware. De hecho, se ha convertido en un estándar que es incorporado de fábrica en la mayoría de estaciones gráficas con prestaciones medias.

Básicamente, consiste en disponer de dos grandes matrices o buffers con las mismas dimensiones que la resolución del monitor. El primero de ellos, denominado **frame buffer**, suele estar incorporado en la memoria de vídeo y sencillamente almacena en cada posición (x, y) un número correspondiente al color que se mostrará en el pixel de coordenadas (x, y). Este buffer está siempre presente y es independiente del algoritmo empleado para dibujar una imagen. Realmente, cuando dibujamos un pixel mediante una llamada a una función del tipo $pixel(x, y, color)$, lo que el procesador hace es almacenar el número indicado por *color* en la posición (x, y). Posteriormente, será el controlador de vídeo el que, de acuerdo con los ciclos de refresco del monitor, lea el valor almacenado en cada una de las posiciones del frame buffer y represente el punto correspondiente de la pantalla con el color apropiado.

La clave del algoritmo está en la segunda matriz, denominada buffer de profundidad o **z-buffer**. En él se almacenará la componente z (de ahí el nombre) del punto de la escena más cercano al observador que se proyecta sobre cada pixel. Naturalmente, se supone que todos los objetos han sido previamente transformados por las ecuaciones de la perspectiva y de transformación de vista, de modo que la coordenada z de un punto representa efectivamente su "profundidad" (ver figura 28).

El desarrollo del algoritmo puede esquematizarse en los siguientes pasos:

1. Inicializar a 1 la matriz $Z[x, y]$ que constituye el z-buffer. Inicializar una matriz $F[x, y]$, que representa al frame buffer, con el valor del color de fondo (esta inicialización, en la práctica, consiste sencillamente en borrar la pantalla).

2. Eliminar las caras posteriores y transformar por perspectiva y encuadre los vértices de las caras anteriores, escalando la componente z entre 0 y 1 (dividiendo todas por la mayor). De esta forma, la matriz $Z[x, y]$, que se había inicializado a 1, indica que en cada pixel l más cercano al observador es el fondo de la escena, dado que aún no ha sido procesado ningún polígono.

3. Para cada polígono transformado:

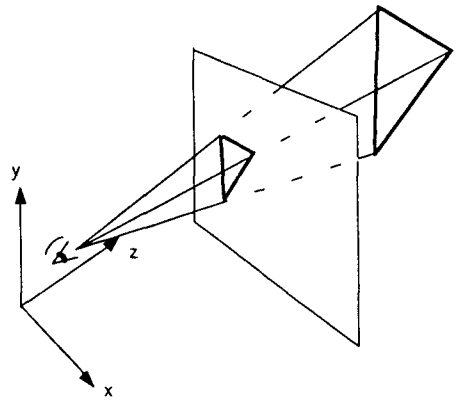


Figure 28

3a. Calcular todos los pixels que contiene. Esto es equivalente a rellenar el polígono mediante un algoritmo de relleno, sólo que ahora no estamos interesados en rellenar efectivamente el polígono, sino sólo en averiguar las coordenadas de estos pixels.

3b. Para cada uno de estos pixels, calcular la profundidad z del polígono en dicho pixel. Puesto que se supone conocida la ecuación del plano del polígono, de aquí podremos obtenerla:

$$Ax + By + Cz + D = 0 \Rightarrow z = -\frac{Ax + By + D}{C}$$

donde (x, y) son las coordenadas del pixel. En este paso hay que tener en cuenta un detalle muy importante. Si la proyección fuera paralela, efectivamente obtendríamos la coordenada de z correcta. Pero si la proyección es en perspectiva, el hecho de evaluar la ecuación el punto (x, y) que representa a un pixel no da necesariamente el valor de z del punto en cuestión. Por ejemplo, puede que para el pixel $(20, 20)$ la coordenada z que nos salga sea 100, ya que el punto $(20, 20, 100)$ es el punto del plano con esos valores de x e y. Sin embargo, el punto que va a representarse sobre ese pixel puede ser otro punto distinto, debido al efecto de la perspectiva. En este caso, debemos trabajar con la ecuación del plano transformado, es decir, el plano sobre el que se encuentran los puntos (x, y, z), siendo x e y las coordenadas de los pixels, y z la verdadera profundidad del punto representado por ese pixel.

A continuación comparamos esa profundidad con la que está almacenada en el z-buffer en la posición (x, y), es decir, con el valor de $Z[x, y]$. Si el $z < Z[x, y]$, el polígono se encuentra en el pixel (x, y) más cercano al observador que cualquier otro previamente procesado. Por tanto, debemos actualizar las dos matrices con la nueva información:

$$\begin{aligned} Z[x, y] &= z \\ F[x, y] &= color(x, y, z) \end{aligned}$$

donde $color(x, y, z)$ indica el color que debe tener el polígono en curso en el punto

(x, y, z). En general, ese color no va a ser uniforme en todo el polígono, sino que va a depender de las coordenadas de cada punto en particular.

Si $z > Z[x, y]$, el polígono se encuentra en el pixel (x, y) más distante del observador que algún otro previamente procesado, y cuya profundidad es precisamente la almacenada en $Z[x, y]$). Se conservan entonces los valores, tanto de $Z[x, y]$ como de $F[x, y]$, pues mantienen los valores correctos, hasta ahora, de la profundidad y el color del polígono más cercano en el punto (x, y).

Una vez finalizado el test de profundidad se continúa en el paso 3b con el siguiente pixel. Concluido el proceso para todo el polígono en curso, se continúa con el siguiente polígono en el paso 3. Cuando todos los polígonos, y por tanto cada uno de sus pixels constituyentes han sido procesados, la matriz $F[x, y]$ contendrá el color correcto que debe ser mostrado para cada pixel, con el problema de visibilidad consiguientemente resuelto.

Son evidentes, tanto la extrema simplicidad del algoritmo, que se basa en procedimientos ya conocidos para rellenar polígonos, como la enorme cantidad de cálculo que requiere: cada pixel es evaluado tantas veces como polígonos a los que pertenece. En efecto, notemos que el z-buffer no es sino una versión discreta, a nivel de pixel, del método del pintor, con la diferencia de que no siempre se comienza con el color del punto más lejano.

Una característica muy importante del algoritmo es que su tiempo de ejecución prácticamente no depende de la complejidad de la escena, sino sólo de su parte visible. Quiere esto decir que dos escenas distintas con 1000 y 10000 plígonos respectivamente serían resueltas en el mismo tiempo aproximadamente, siempre y cuando las imágenes finales fueran del mismo orden de complejidad.

El algoritmo del z-buffer es, comparativamente, mucho más costoso que la mayoría. Sin embargo, su constancia en tiempo de ejecución frente al de los restantes, que aumentan de forma cuadrática con el número de polígonos, lo convierten en una elección acertada para escenas de gran complejidad, las cuales son, por otra parte, cada vez más comunes en aplicaciones gráficas.

Por último, mencionar que este algoritmo no requiere que los objetos sean siempre poligonales. De hecho, uno de sus grandes atractivos es que cualquier tipo de objeto puede ser utilizado, siempre y cuando sea posible calcular el color y la profundidad de cada uno de los pixels de su proyección. Es más, no necesitaremos implementar algoritmos de intersección explícitos.

Barrido con z-buffer

Una forma de evitar la gran cantidad de memoria que requiere la matriz del algoritmo (para una resolución de $1024x768$ y 32 bits por pixel son necesarios 3Mb) consiste en resolver el problema de visibilidad línea a línea sobre el monitor comenzando, por

ejemplo, por la superior y descendiendo hata llegar a la de más abajo. De este modo, podemos disponer de un z-buffer lineal; es decir, un vector $Z[x]$ con tantos elementos como la resolución horizontal de la pantalla, y suponer que asímismo el frame buffer está constituido por otro vector $F[x]$ con la misma dimensión. Por lo demás, el algoritmo es idéntico al descrito anteriormente, con la única salvedad de que antes de comenzar cada línea hay que reinicializar los dos vectores.

El problema es que cada polígono debe considerarse muchas veces (una por cada linea de barrido que cruce al polígono). Para resolver el problema de evitar la repetición innecesaria de un mismo cálculo sobre líneas distintas pueden introducirse varias mejoras, como veremos con el siguiente algoritmo.

5.3.2 Algoritmo de línea de rastreo (algoritmo de Watkins)

Se trata de un algoritmo de barrido que resuelve el problema de visibilidad línea a línea, pero con un enfoque distinto al anterior. Estudiemos cuál es su idea subyacente: si se intersecta la escena, ya transformada, por un plano horizontal $y = y_i$ correspondiente a la ordenada de una línea cualquiera de barrido, esto producirá una serie de "rodajas" bidimensionales que pueden compararse entre sí, en 2D, para determinar qué parte de ellas es la más cercana al observador y es, por tanto, visible (ver figura 29)

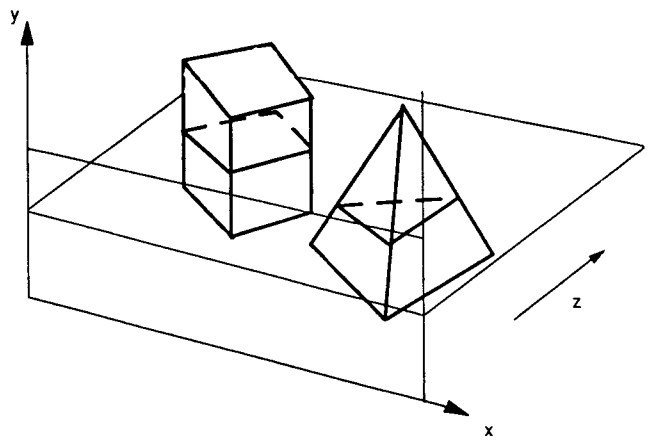


Figure 29

Estas secciones planas de la escena son en realidad polígonos en el plano $y = y_i$. En la figura 30 se han representado los correspondientes a la figura 29, vistos desde arriba, de modo que el eje x corre hacia la derecha y el eje z hacia el fondo de la escena (que ahora aparece vertical).

Proyectando todos los vértices de estos polígonos sobre el eje de abscisas obtenemos

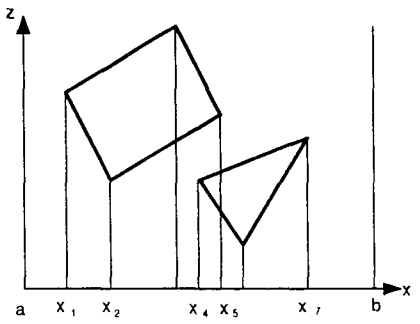


Figure 30

los puntos x_1, x_2, ..., x_n donde también hay que incluir los extremos de la pantalla, $a = 0$ y $b = x_{max}$. El intervalo $[a, b]$ queda entonces dividido en subintervalos (a, x_1), (x_1, x_2), ..., (x_{n-1}, x_n) y (x_n, b) que suelen denominarse **spans** o tramos. El algoritmo clasifica estos tramos en tres categorías diferentes:

1. No contienen ningún segmento (por ejemplo, (a, x_1) y (x_7, b) en la figura). En ellos se muestra el color de fondo.

2. Contienen un único segmento (no hay ninguno en la figura, pues todos los polígonos son cerrados, por lo que como mínimo siempre hay dos segmentos al menos). En estos spans, si existiesen, se mostraría el color del único polígono de la escena 3D que ha producido el segmento.

3. Contienen más de un segmento. Este caso es el más conflictivo pues requiere averiguar cuál de los segmentos es el más cercano al observador. En la figura 31 se muestra el span (x_4, x_5) con cuatro segmentos: dos de ellos pertenecientes al triángulo de la figura 30 y otros dos al cuadrado.

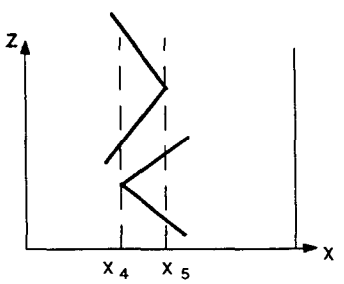


Figure 31

Observemos que en el interior del intervalo, nunca se altera la ordenación relativa de los segmentos, de forma que aquél que tiene menor componente z en un punto

cualquiera, también la tiene menor en todos los demás puntos. Podemos elegir el punto medio del intervalo:

$$x = \frac{x_4 + x_5}{2}$$

y realizar una comprobación de profundidad en dicho punto. De nuevo y de una manera análoga al algoritmo del z-buffer, dado que la coordenada y es conocida (y constante para todos los segmentos) y disponemos de las ecuaciones de los planos, la obtención de la profundidad z en el punto medio es trivial. Una vez dterminado el segmento visible (menor z), el span se muestra con el color del polígono 3D que produjo el segmento en cuestión. De esta forma, cuando se han clasificado todos los spans y dibujado cada uno con el color apropiado, la línea i-ésima de la pantalla está resuelta en cuanto a visibilidad se refiere y puede pasar a procesarse la siguiente.

Procedimiento mejorado

El algoritmo tal y como ha sido descrito es rápido y potente, pero peca de la ineficiencia de necesitar calcular en cada línea de barrido la intersección del correspondiente plano con toda la escena. Veamos cómo podemos mejorar esto. En una etapa de preproceso de la escena (dependiente ya del observador), se construye un array de punteros con tantas entradas como líneas de barrido haya (desde la entrada para la línea $y = 0$ hasta la línea $y = y_{max}$). Cada uno de estos punteros va a apuntar a una lista encadenada de aristas. La estructura de datos para cada arista se compone de los siguientes campos:

1. Coordenada x del extremo de la arista con menor y.

2. Coordenada y del otro extremo (y máxima de la arista).

3. Incremento de x para pasar de una línea de barrido a la siguiente. En realidad, este valor viene indicado por la inversa de la pendiente de la recta que contiene a la arista

$$\begin{aligned} y_i &= mx_i + n \\ y_{i+1} &= m(x_i + \Delta x) + n = (mx_i + n) + m\Delta x = y_i + m\Delta x \Rightarrow \\ m\Delta x &= y_{i+1} - y_i = 1 \Rightarrow \Delta x = \frac{1}{m} \end{aligned}$$

4. Un identificador del polígono al cual pertenece la arista.

Cada arista irá incluida en la lista encadenada correspondiente a la mínima y de sus dos extremos. Por otro lado, cada polígono vendrá representado por otra estructura de datos, compuesta por estos campos:

1. La ecuación del plano.

2. Información sobre el color del polígono.

3. Un flag indicando si el polígono se encuentra activo o no para una línea de barrido determinada.

Aunque todo este preproceso puede parecer costoso, téngase en cuenta que realmente no es necesaria ninguna comparación entre lados o polígonos. Cada arista, así como cada polígono, se procesa una única vez, de modo que en la práctica, esta fase previa es muy rápida. El algoritmo comienza por la línea de barrido $y = 0$ y continúa sucesivamente para cada línea $y = y_i$. En todo momento tendremos una **lista de aristas activas**, que nos va a indicar qué aristas son las que intervienen en cada línea de barrido que procesemos (que en realidad son las que indican los spans). Los pasos para cada línea son:

1. Examinar el contenido del array de punteros para la posición correspondiente a $y = y_i$ para detectar si en esta línea comienza alguna nueva arista, en cuyo caso se añade a la lista de aristas activas.

2. Procesar la lista de aristas activas, según el test de profundidad que vimos antes. Para arista activa, su estructura de datos va a indicar su abscisa x correspondiente a la ordenada y_i en curso (que se actualizará más adelante, al cambiar de línea). Estas abscisas son los puntos $x_1, x_2, ..., x_n$ que dividen el rango horizontal de la pantalla en spans. Este conjunto se divide de menor a mayor, dejando los más cercanos primero, y se procede a completar la línea calculando el color correcto para cada span como veíamos antes.

3. Actualizar la lista de aristas activas. Si álguna ya ha llegado a su y máxima, se elimina de la lista de aristas activas. Si no, se calcular la nueva abscisa x para utilizarla en la siguiente línea

$$x_i = x_i + \Delta x$$

Veamos un ejemplo para aclarar un poco el procedimiento, porque a lo mejor no se ha entendido del todo (o no se ha entendido nada). Tomemos la siguiente escena de la figura 32, formada por dos triángulos.

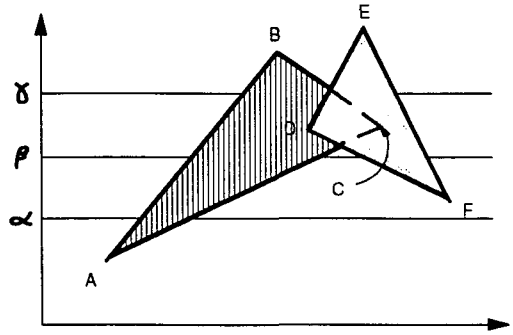


Figure 32

Cuando procesemos la línea $y = \alpha$, habrán sólamente dos aristas activas: AB y AC, en ese orden, lo cual nos da 3 spans: $(0, AB)$, (AB, AC) y (AC, x_{max}). Para el primer span no hay ningún polígono activo, con lo cual pintamos con el color de fondo. Al entrar en el segundo span tocamos la arista AB, por lo que negamos el flag del polígono al cual pertenece, el cual pasa a ser verdadero, indicando que sólo un polígono está activo en ese momento. Por lo tanto lo pintamos con el polor de ese polígono. Al entrar en el último span tocamos la arista AC, e invertimos el flag del polígono, volviendolo a poner a 0, con lo cual no hay ningún polígono activo y se pinta de nuevo con el color de fondo.

Para la línea $y = \beta$ vamos a tener cuatro aristas activas: AB, AC, FD y FE, lo que nos da 5 spans. El primero es del color de fondo. Luego tocamos la arista AB perteneciente al triángulo izquierdo, con lo cual pintamos con ese color. Entonces tocamos la arista AC con lo cual se desactiva dicho triángulo y volvemos al color de fondo. A continuación viene la arista FD que activa el triángulo de la derecha, pintando con ese color hasta llegar a FE que vueve a desactivarse.

Para la línea $y = \gamma$ tenemos también cuatro aristas: AB, DE, CB y FE. Cuando estemos entre AB y DE sólo estará activado el triángulo izquierdo, con lo cual se pinta con su color. A continuación aparece la arista DE que activa el triángulo derecho, con lo cual aparecen dos polígonos activos. En ese caso debemos decidir cuál es el más cercano al observador para elegir el color correcto (en este caso el derecho). Luego tocamos la arista CB con lo que se desactiva el triángulo izquierdo y volvemos a tener un único polígono activo, el cual usamos para pintar el span.

La fuerza del algoritmo está en utilizar la lista de aristas activas, lo que hace que el número de comparaciones en la etapa 2 se reduzca al mínimo: sólo entre aristas que realmente interceptan la línea de barrido. Por otra parte, el uso inteligente de la coherencia entre líneas sucesivas, evita la necesidad de calcular las intersecciones del plano horizontal que contiene a la línea en curso con la escena: las abscisas de estas intersecciones se actualizan, de una línea a la siguiente, mediante una simple adición.

5.3.3 Métodos basados en prioridad

El concepto de prioridad, relacionado con la técnica de parcelación, es utilizado por una gran variedad de algoritmos. El núcleo fundamental de éstos consisten en conseguir establecer la relación de prioridad entre los polígonos, para luego ordenarlos e irlos dibujando según la técnica del pintor.

Supongamos que toda la escena ya ha sido transformada al sistema de referencia del observador, y para ordenar los polígonos utilizaremos la máxima profundidad de cada uno, es decir, el mayor valor de la componente z de cada vértice en un polígono, como se ve en la figura 33.

Así, en esta clasificación inicial de la figura, el polígono 1 sería considerado anterior

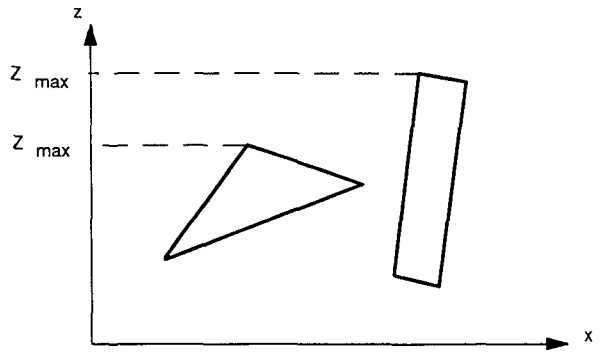


Figure 33

al polígono 2, pues $z_{max1} < z_{max2}$. Pero observemos que esta ordenación no presupone que el polígono 1 tenga mayor prioridad que el 2, pues realmente en la figura ocurre lo contrario. Sin embargo, el algortimo utiliza una propiedad de coherencia espacial por la cual, la situación planteada en la figura ocurre muy pocas veces. Por tanto, una vez realizada la clasificación previa, el algoritmo se centra en detectar estos casos especiales y resolverlos.

La lista de polígonos se recorre entonces en sucesión, comenzando por el teóricamente más lejano al observador, aquél cuya z_{max} es mayor, y se procede hasta llegar al más cercano. En una etapa cualquiera, cuando se está procesando el polígono P_i, se supone que todos los anteriores P_1, P_2, ..., P_{i-1} ya están correctamente ordenados por prioridad, de modo que el problema es determinar si P_i está en el lugar que le corresponde o debe ser movido en la lista hacia más adelante (nunca hacia atrás), lo cual ocurriría si alguno de los que le preceden P_{i+1}, P_{i+2}, ..., P_n realmente tuviera menor prioridad. Para ello, se examina la lista de estos polígonos que, hasta el momento, son considerados más cercanos al observador al tener z_{max} menor que la de P_i.

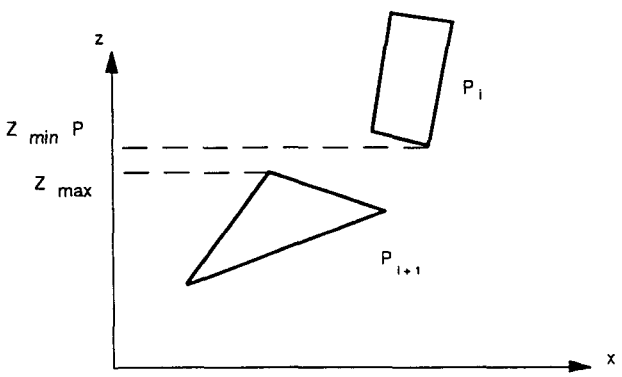


Figure 34

Llamemos por simplicidad P al polígono P_i y Q al siguiente polígono P_{i+1}. Un primer test comprueba si la mínima componente z de P es mayor que la máxima de Q. En caso afirmativo, como se ve en la figura 34 debido a la preordenación de los polígonos, esto mismo sucede para todos los restantes polígonos y por tanto, P no puede ocultar a ninguno de ellos: P está correctamente colocado en la lista y se pasa a procesar el siguiente. En caso de que el test falle, como ocurriría en la primera figura de este método, entonces se dice que los polígonos **solapan en profundidad,** y es necesario un análisis mucho más cuidadoso de la situación. Este se hace mediante una serie de tests en cadena, cada uno más costoso de realizar que el anterior, pero cuyo orden está diseñado para obtener máxima ventaja de las propiedades de coherencia, de forma que las comprobaciones más complejas sólo son realizadas cuando se dan circunstancias verdaderamente excepcionales. Estos tests son los siguientes:

1. ¿Son disjuntos P y Q en x?

2. ¿Son disjuntos P y Q en y?

3. ¿Está P completamente contenido en el semiespacio definido por el plano de Q más lejano al observador?

4. ¿Está Q completamente contenido en el semiespacio definido por el plano de P más cercano al observador?

5. ¿Son disjuntas las proyecciones de P y Q sobre el plano xy?

En el momento en el que una de estas cuestiones tenga respuesta afirmativa, el polígono P se declara correctamente colocado en la lista y se procede con el siguiente polígono P_{i+1}. En cambio, si el polígono P pasa negativamente todos los tests, es más que probable que no esté correctamente situado, pues posiblemente oculte a Q, teniendo mayor prioridad que él. En este caso, se intercambian los órdenes de P y Q y este último pasa a ser el polígono en curso. Todo el proceso debe repetirse para comprobar si ahora es Q el que está correctamente colocado.

Si en la comparación siguiente de Q con P todos los tests dieran de nuevo resultado negativo, nos encontraremos ante un **solapamiento cíclico** como el de la figura 35, en la que P oculta a Q y recíprocamente, Q oculta a P. En este caso P y Q no pueden ser ordenados por prioridad entre sí, y la única solución es dividir P a lo largo del plano que contiene a Q, como se ve en la figura. Los dos subpolígonos en que queda seccionado P se incluyen individualmente en la lista de polígonos aún sin procesar y se continúa según el esquema anteriormente descrito.

Al final del proceso, la lista de polígonos posiblemente contendrá un número mayor de éllos que los originalmente presentes, debido a las subdivisiones realizadas, pero ahora estará correctamente ordenada por prioridades y puede verterse a la pantalla comenzando con el de menor prioridad y continuando hasta llegar al de máxima prioridad, siguiendo el esquema del método del pintor.

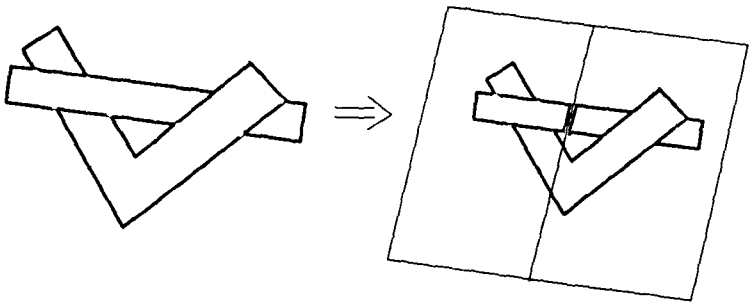


Figure 35

Como puede verse, la fase de clasificación puede llegar a ser muy costosa computacionalmente, y por ello sólo se utiliza en escenas donde a priori se sabe de la inexistencia de solapamientos cíclicos, y más aún, donde un gran número de polígonos pueden ordenarse por prioridad incluso antes de fijar la posición del observador. Por esto es muy utilizado en simuladores de vuelo y conducción: en la escena que simula el terreno sobrevolado, todos los polígonos tienen siempre menor prioridad que los objetos del primer plano, como pueden ser árboles, casas u otros aviones. En estos casos, el modelo se parcela muy eficientemente, parte en fase de modelado y parte en tiempo real. Dentro de cada parcela se establece la relación de prioridad muchas veces independientemente de la posición del observador.

5.3.4 Algoritmos para superficies curvas

Todos los algoritmos que hemos presentado, a excepción del z-buffer, han sido descritos para objetos formados por superficies poligonales. Aquellos objetos que estén formados por superficies curvas deberán ser aproximados por pequeños polígonos si queremos aplicarle una de estas técnicas. Pero existen otros algoritmos que pueden dibujarnos este tipo de objetos sin necesidad de realizar la aproximación poligonal previa. Uno de ellos consiste simplemente en subdividir recursivamente cada patch en cuatro nuevos patches, hasta que su proyección sobre la pantalla ocupe como mucho un pixel. Finalmente se aplica la técnica del z-buffer a esos patches para ver si son visibles en dicho pixel. El código para este algoritmo sería el siguiente:

```
Procedure VisualizaPatch ()
Calcula la proyección del patch
if patch cubre <=1 pixel then
        if z del patch < z_min then DibujaPixel
else
        Subdivide patch en cuatro
        Visualiza cada patch
end
```

Uno de los pasos más costosos es el de calcular el tamaño de la proyección del patch. Pero podemos acelerarlo si en lugar de usar el patch usamos un cuadrilátero formado por las cuatro esquinas del patch. Obtendremos aproximadamente el mismo resultado de forma más rápida.

Otra técnica alternativa se basa en la subdivisión adaptativa de cada patch hasta alcanzar un cierto umbral que catalogue al patch como suficientemente plano. Este umbral puede depender de la resolución de la pantalla y de la orientación del área que estamos subdividiendo con respecto al plano de proyección. De esta forma evitamos cálculo innecesario.

5.3.5 Trazado de rayos (Ray Tracing)

Es un procedimiento exhaustivo que aplica la definición de visibilidad a puntos seleccionados de la escena. Al contrario que los algoritmos anteriores no utiliza propiedades de coherencia en los objetos y trata cada punto individualmente sin tomar en consideración la información obtenida en el procesado de puntos previos. Sin embargo, incorpora dos elementos que hacen de esta técnica una de las más potentes herramientas de rendering fotorrealista: la posibilidad de que la escena esté compuesta por objetos completamente heterogéneos, y la facilidad con la que se simulan multitud de fenómenos ópticos, como son la reflexión y refracción de la luz al interaccionar con una superficie, la generación de sombras arrojadas por unos sólidos sobre otros o la incorporación de texturas, que permiten pintar sobre una superficie el detalle deseado, desde una rugosidad hasta un rótulo. Todas estas cualidades las veremos en el capítulo siguiente concerniente a la iluminación; por ahora sólo nos quedaremos con el núcleo básico que constituye la primera etapa del proceso total de generación de la imagen: la determinación de la parte visible de la escena.

Consideremos una situación general, donde tenemos un objeto S de $\Re^3$ y un observador P exterior a él, como se ve en la figura 36. Supongamos definido un plano de proyección mediante un vector D y su distancia al observador d. En estas condiciones, el proceso usual consiste en aplicar al objeto S, sucesivamente, un recorte tridimensional, realizar la transformación de vista, y finalmente la proyección perspectiva sobre la pantalla. En una o varias de estas etapas se intercala un algoritmo de visibilidad, de modo que realmente sólo quede representada la parte visible del objeto S.

La técnica del ray-tracing opera de una forma muy distinta, que se muestra esquematizada en la figura 37. Sobre el plano de proyección, se selecciona una ventana rectangular con las mismas proporciones que la pantalla. Esta ventana se subdivide en pixels, cada una de las cuales se corresponde con un verdadero pixel sobre el monitor[2].

[2]Reconozco que a veces uso indistintamente las palabras "pantalla", "monitor" o "imagen" para referirme a lo mismo. Lo importante aquí es entender que el objetivo final es generar una imagen bidimensional de la escena, la cual no es más que un array 2D de pixels, sin importar realmente si la voy a mostrar en pantalla o la voy a grabar en el disco.

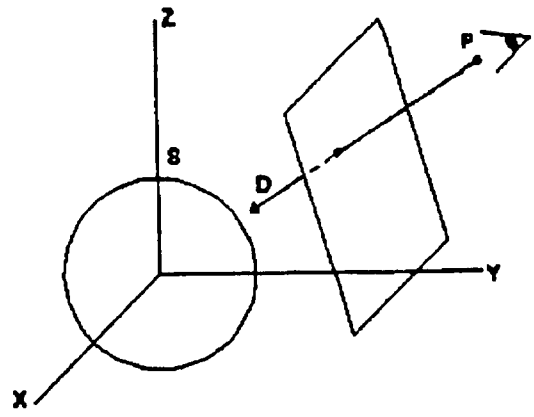


Figure 36

El procedimiento consiste entonces en lanzar rayos desde la posición del observador a través del centro de estos pixels, propagándolos por la escena hasta que intersecten un objeto. Este primer punto de intersección Q es el que determina el color del correspondiente pixel de acuerdo con algún modelo de iluminación, como veremos en el próximo capítulo.

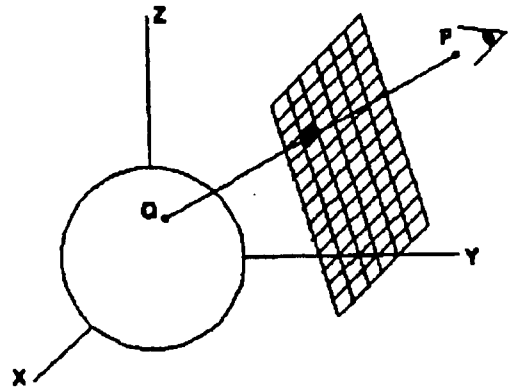


Figure 37

Podemos ver una diferencia importante entre este método de muestreo de una escena y los conceptos de visibilidad y perspectiva. Al definir la visibilidad, considerábamos un punto Q sobre un objeto y se trataba de decidir si era o no visible, uniéndolo mediante un segmento con el observador. En caso afirmativo, la intersección de este segmento con el plano de proyección producía la imagen perspectiva del punto Q. Por tanto, se suponía conocido que el punto Q efectivamente pertenecía al objeto y el problema era, de una parte, establecer su visibilidad, y de otra, calcular su proyección. La técnica del ray-tracing plantea la cuestión exactamente al revés: se supone conocida cuál va a ser la proyección o perspectiva y el problema es entonces averiguar qué puntos de la

escena se proyectarán sobre él, tomando el más cercano como visible.

Para fijar ideas, consideremos que la escena está compuesta por una superficie regular que puede expresarse por medio de una ecuación implícita $F(x, y, z) = 0$. Definimos un rayo como una semirrecta con origen en el observador $P = (a, b, c)$ y dirección y sentido, los dados por un vector $R = (r_x, r_y, r_z)$, que supondremos normalizado

$$\begin{aligned} 1 &= r_x^2 + r_y^2 + r_z^2 \\ (x, y, z) &= (a, b, c) + t(r_x, r_y, r_z) \end{aligned}$$

Para cada pixel de la imagen, definiremos el rayo que partiendo del ojo atraviese dicho pixel. Se trata entonces de obtener la intersección entre este rayo y la superficie, y para ello, sustituimos las coordenadas (x, y, z) anteriores en la ecuación de la superficie

$$F(a + tr_x, b + tr_y, c + tr_z) = 0$$

Obtenemos una ecuación en la variable t, cuyas soluciones reales, si es que existen, debemos calcular. Fíjense que sólo estamos interesados en aquéllas soluciones que verifiquen $t > d$, es decir, sólo buscamos intersecciones que se produzcan más allá del plano de proyección, con lo cual se realiza automáticamente un recorte de la escena contra el plano de proyección. Por tanto, la complejidad de la técnica de ray-tracing consiste precisamente en la resolución de la ecuación anterior, que en general puede ser no lineal y requerir métodos numéricos de cálculo de raíces.

Para hacernos una idea de la magnitud de la complejidad del algoritmo, pensemos que para generar una imagen de 1024×768 pixels necesitaremos calcular aproximadamente unos 800.000 rayos, y consecuentemente, el número de operaciones requeridas en cada pixel para calcular la intersección ha de multiplicarse por esta cifra. El valor resultante es elevadísimo, y eso que hasta ahora hemos considerado un único objeto presente en la escena. Si hubiesen n objetos, habría que volver a multiplicar por n en cada pixel. Por otra parte, no hemos considerado aún el cálculo del color en los puntos visibles, el cual necesita al menos 20 operaciones por pixel con un modelo simple de iluminación.

Debe por tanto quedar claro que la técnica de ray-tracing es super costosa y que sólo se justifica su empleo por el hecho de poder visualizar realísticamente superficies regulares, cuyo tratamiento no es posible con los algoritmos descritos en el presente capítulo y, sobre todo, por la posibilidad de incorporar modelos de iluminación para simular un número de fenómenos ópticos interesantes.

En una escena más general, podemos suponer que cada sólido individual posee una ecuación implícita, aunque posiblemente el rango de sus variables esté limitado. Por ejemplo, un polígono puede describirse por la ecuación implícita de su plano

$$Ax + By + Cz + D = 0$$

sujeta a la restricción de que los puntos (x, y, z) permanezcan en el interior del recinto delimitado por el polígono. Por tanto, si al sustituir la ecuación de un rayo en la del

plano, obtenemos el valor t_1 (único, pues la ecuación es lineal) que, a su vez, determina la intersección

$$(x_1, y_1, z_1) = (a, b, c) + t_1(r_x, r_y, r_z)$$

aún debe decidirse si el punto (x_1, y_1, z_1) está efectivamente incluido dentro de las fronteras del polígono, como se ve en la figura 38.

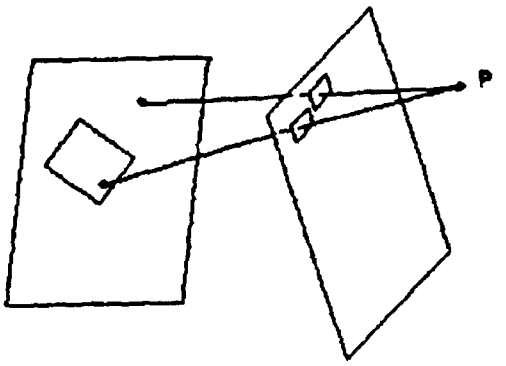


Figure 38

De esta forma, y esto es un detalle importantísimo, cada objeto presente en la escena puede tratarse con una rutina de intersección rayo-sólido independientemente del resto de los objetos, por lo que en la misma escena pueden aparecer tipos de objetos muy diversos. En una etapa final, estas intersecciones serían cotejadas para obtener la más cercana al observador (aquella cuyo t_1 sea menor).

Chapter 6 ILUMINACIÓN

La etapa final en la generación de imágenes realistas, una vez generadas las proyecciones en perspectiva de los objetos, consiste en aplicar los efectos de la luz sobre las superficies visibles en la escena. Para ello se hará uso de un modelo de iluminación con el que se calculará la intensidad de la luz que debería verse en un punto dado sobre la superficie del objeto. Posteriormente, un algoritmo de iluminación o **rendering** usará las intensidades calculadas por el modelo de iluminación para determinar la intensidad de luz para todos y cada uno de los pixels proyectados desde las diferentes superficies de la escena. El rendering de la superficie puede realizarse aplicando el modelo de iluminación a todos los puntos de la supeficie visible, lo cual requeriría un coste excesivo, o bien interpolando las intensidades en el interior de las superficies a partir de un pequeño conjunto de puntos ya calculados. Algunos algoritmos trabajan en el espacio de la imagen (2D) y usan estos esquemas de interpolación, mientras que otros como el ray-tracing invocan al modelo de iluminación para cada pixel.

El poder modelar los colores y demás efectos de la luz que se producen en la realidad es un proceso complejo. Fundamentalmente, estos efectos se describen mediante modelos que consideren la interacción de la energía electromagnética con las superficies de los objetos. Una vez conocidos los parámetros sobre las propiedades ópticas de las superficies (opacas, transparentes, rugosas, etc.), sus posiciones relativas dentro de la escena, el color y la posición de las fuentes de luz, y la posición y orientación del observador, el modelo de iluminación calcula la intensidad proyectada desde un punto particular de una superficie a lo largo de una dirección de vista especificada.

6.1 MODELOS DE ILUMINACIÓN

6.1.1 Luz ambiente

Es el modelo más simple de iluminación que podamos imaginar: cada objeto se muestra usando una intensidad intrínseca a él. Pensemos por ejemplo que se trata de una escena sin fuentes de luz adicionales, donde los objetos son auto-luminosos, cada uno con su propia intensidad. Cada objeto aparecería como una silueta monocromática, a menos que partes de él, como algunos polígonos del objeto, tuvieran una intensidad asociada diferente al resto del objeto. La escena sería algo así como la figura 1.

Podemos expresar un modelo de iluminación por medio de una ecuación de iluminación en función de variables asociadas al punto del objeto que estamos calculando. La ecuación de iluminación para este modelo sería

$$I = k_i$$

donde I es la intensidad resultante, y el coeficiente k_i es la intensidad intrínseca del objeto i. Ya que esta ecuación de iluminación no contiene términos que dependan de la

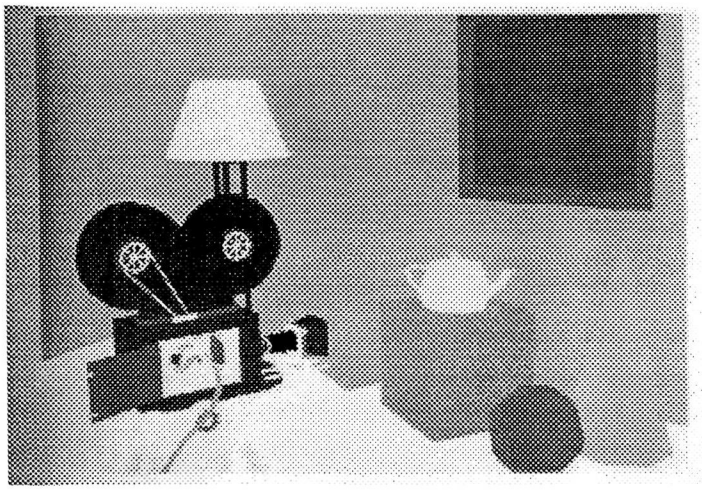


Figure 1

posición del punto que estamos calculando, podemos evaluarla una sola vez para cada objeto.

Imaginemos ahora que en lugar de que cada objeto posea una luminosidad propia, existe una fuente de luz difusa no direccional, resultado de las múltiples reflexiones de la luz por las numerosas superficies existentes en la escena. A esta luz se le conoce como **luz ambiente**. Si asumimos que la luz ambiente impregna por igual sobre todas las superficies en todas las direcciones, entonces nuestra ecuación se convierte en:

$$I = I_a k_a$$

donde I_a es la intensidad de la luz ambiente, la cual es constante para toda la escena. La cantidad de luz ambiente reflejada por la superficie de un objeto vendrá determinada por el **coeficiente de reflexión ambiental**, k_a, el cual está entre 0 y 1. Este coeficiente es una propiedad del material con el que se supone que esta construido el objeto. Es decir, podemos pensar en él como si fuese la característica principal de cada tipo de material concreto (madera, metal, plástico, etc.). Al igual que otras propiedades, este coeficiente es en realidad una conveniencia empírica que no se corresponde directamente con ninguna propiedad física de los materiales existentes en la realidad.

6.1.2 Reflexión difusa

Aunque los objetos que se encuentran iluminados por la luz ambiente son más o menos brillantes en una cantidad proporcional al intensidad ambiental, todavía siguen estando uniformemente iluminados a lo largo de sus superficies. Consideremos ahora que iluminamos un objeto con una **fuente de luz puntual**, cuyos rayos emana uniformemente en todas las direcciones desde un punto. El brillo del objeto variará de una parte a otra, dependiendo de la dirección y distancia a la fuente de luz.

Reflexión lambertiana

Las superficies que aparecen con igual luminosidad desde todos los ángulos, como las paredes de una habitación, se dice que poseen **reflexión difusa** o **lambertiana**, debido a que reflejan la luz con igual intensidad en todas direcciones. Para una superficie dada, la luminosidad depende sólamente del ángulo θ entre la dirección $\overline{L}$ de la fuente de luz y la normal a la superficie, $\overline{N}$, como vemos en la figura 2.

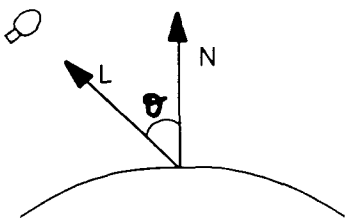


Figure 2

Examinemos por qué ocurre este fenómeno. Hay dos factores que influyen. En primer lugar, como vemos en la figura 3, un rayo de luz que intersecte una superficie cubre un área cuyo tamaño es inversamente proporcional al coseno del ángulo θ que forma con la normal $\overline{N}$. Si el rayo tiene un area diferencial infinitamente pequeña, dA, entonces el rayo intersecta un área $dA/\cos\theta$ sobre la superficie. Así, para un rayo de luz incidente, la cantidad de energía lumínica que cae sobre dA es proporcional al $\cos\theta$, independientemente del material de la superficie.

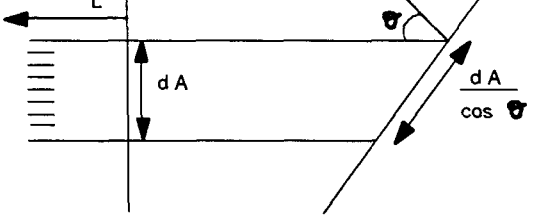


Figure 3

En segundo lugar, debemos considerar la cantidad de luz que ve el observador. Las superficies lambertianas tienen la propiedad, a menudo conocida como ley de Lambert, que la cantidad de luz reflejada desde un área diferencial unitaria dA hacia el observador es directamente proporcional al coseno del ángulo entre la dirección del observador y la normal $\overline{N}$. Ya que la cantidad de área de superficie vista es inversamente proporcional al coseno de este ángulo, estos dos factores se anula mutuamente. Por ejemplo, a medida

que se incremente el ángulo de visión, el observador verá mayor área de la superficie, pero la cantidad de luz reflejada con ese ángulo por unidad de área es proporcionalmente menor. De esta manera, para las superficies lambertianas, la cantidad de luz vista por el observador es independiente de la dirección de vista y es proporcional sólamente al coseno de θ, que es el ángulo con el que incide la luz.

La ecuación de iluminación difusa es

$$I = I_p k_d \cos\theta$$

donde I_p es la intensidad de la fuente de luz puntual; k_d es el **coeficiente de reflexión difusa** del material, y es una constante entre 0 y 1 que varía para cada material. El ángulo θ debe estar entre 0^o y 90^o si es que la fuente de luz tiene algún efecto directo sobre el punto que estamos calculando. En caso contrario, la luz no le incidiría. Suponiendo que los vectores $\overline{N}$ y $\overline{L}$ han sido normalizados, podemos reescribir la ecuación anterior por medio del producto escalar de ambos

$$I = I_p k_d \left(\overline{N} \cdot \overline{L}\right)$$

Si la fuente de luz puntual está lo suficientemente lejos del objeto al que está iluminando, prácticamente obtendremos el mismo ángulo en todas las superficies que compartan la misma normal. En este caso, diremos que se trata de una **fuente de luz direccional**, y $\overline{L}$ será un valor constante para la fuente de luz. En la figura 4 pueden verse varias esferas iluminadas con una fuente de luz puntual, utilizando el modelo anterior para calcular la intensidad de cada pixel en donde la esfera es visible.

Figure 4

Los objetos calculados con este modelo lucen como iluminados por un flash en una habitación oscura. Por esto se suele añadir un término ambiental para parecer más realista:

$$I = I_a k_a + I_p k_d \left(\overline{N} \cdot \overline{L}\right)$$

Con esta ecuación el resultado aparece en la figura 5.

Atenuación de la fuente de luz

Si la proyección de dos superficies paralelas del mismo material se superponen en una image, con el modelo anterior no distinguiremos donde acaba una y empieza la otra,

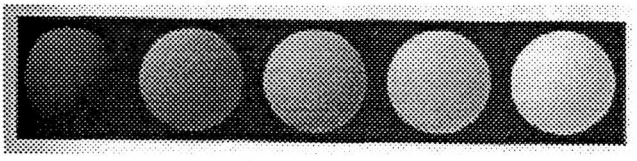


Figure 5

ya que el valor calculado es independiente de la distancia de cada objeto a la fuente. Para solucionar este problema, introducimos un **factor de atenuación,** f_{at}, quedando la ecuación

$$I = I_a k_a + f_{at} I_p k_d \left(\overline{N} \cdot \overline{L}\right)$$

Un factor de atenuación típico, que tenga en cuenta el hecho de que la energía que parte de una fuente de luz y llega a la superficie de un objeto decrece de forma proporcional con el cuadrado de la distancia entre ambas, d_L. Es decir

$$f_{at} = \frac{1}{d_L^2}$$

Superficies y luces con color

Hasta ahora hemos descrito luces y superficies monocromáticas. Si queremos tener en cuenta el color de cada una, hay que escribir ecuaciones separadas para cada componente del modelo de color que estemos usando. Vamos a representar el **color difuso** de un objeto por un valor de O_d para cada componente de color. Por ejemplo, la terna (O_{dR}, O_{dG}, O_{dB}) define las componentes difusas roja, verde y azul de un objeto en el sistema de color RGB[1]. En este caso, las tres componentes primarias de la fuente de luz, I_{pR}, I_{pG}, I_{pB}, serán reflejadas en proporción a $k_d O_{dR}$, $k_d O_{dG}$, y $k_d O_{dB}$ respectivamente. Así, para la componente λ (siendo λ cualquiera de las 3, R, G o B) tendremos:

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + f_{at} I_{p\lambda} k_d O_{d\lambda} \left(\overline{N} \cdot \overline{L}\right)$$

El uso de un simple coeficiente para escalar la expresión en cada una de las ecuaciones permite al usuario controlar la cantidad de reflexión ambiental y difusa, sin alterar las proporciones de sus componentes.

6.1.3 Reflexión especular

La reflexión especular es la que se produce sobre las superficies brillantes, como el metal. Iluminar por ejemplo una manzana (recién lavada) con una fuente de luz blanca

[1]El sistema de color RGB es el más extendido en las aplicaciones gráficas. Representa un color mediante tres valores: uno que indica la componente roja, otro la azul y otro la verde. Jugando con las combinaciones de las tres componentes podemos formar cualquier color. Cuando los tres valen cero tendremos el color negro, y cuando valgan 1 tendremos el blanco.

produce un zona circular con mucha intensidad, producto de la reflexión especular, mientras que el resto de la manzana queda iluminada por la reflexión difusa. Además, dicha zona circular la veremos blanca, que es el color de la luz, y no roja. Si ahora movemos la cabeza veremos cómo esa zona circular se mueve también. Esto es debido a que las superficies especulares no reflejan la luz en todas las direcciones por igual. En un superfices totalmente especular (un espejo perfecto) la luz sólo se refleja en la dirección de reflexión $\overline{R}$, que viene dada por la dirección simétrica del vector $\overline{L}$ con respecto a $\overline{N}$. De esta manera, un observador podría ver la luz reflejada a traves del espejo sólo cuando el ángulo α de la figura 6 sea cero. α representa el ángulo entre $\overline{R}$ y la dirección del observador $\overline{V}$.

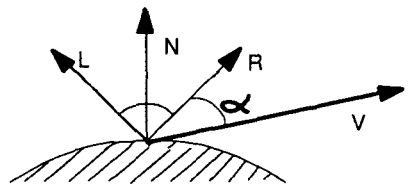


Figure 6

El modelo de iluminación de Phong

Phong desarrolló en 1975 un modelo de iluminación para superficies especulares no perfectas, tales como la manzana. Asumía que la reflectancia especular máxima se producía cuando α era cero, e iba decreciendo a medida que α aumentaba. Esta caída era proporcional al $\cos^n \alpha$, donde n era el **exponente de reflexión especular** del material. Los valores típicos de n varían desde 1 a varios cientos, dependiendo del tipo de material que estemos simulando. Un valor de 1 produce una zona brillante muy ancha, mientras que valores mayores simulan una zona brillante casi puntual (ver figura 7). Para un espejo perfecto, n sería infinito.

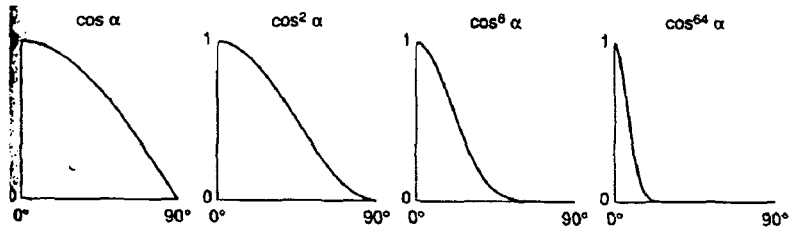


Figure 7

La cantidad de luz incidente reflejada especularmente depende del ángulo de incidencia θ. Si llamamos $W(\theta)$ a la fracción de luz reflejada especularmente, el modelo de

Phong sería el siguiente:

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + f_{at} I_{p\lambda} \left[k_d O_{d\lambda} \cos\theta + W(\theta) \cos^n \alpha \right]$$

Si la dirección de reflexión $\overline{R}$, y la dirección del observador $\overline{V}$ están normalizadas, entonces

$$\cos\alpha = \overline{R} \cdot \overline{V}$$

Además, $W(\theta)$ suele ser una constante k_s, el **coeficiente de reflexión especular** del material, el cual toma valores entre 0 y 1. La ecuación resultante puede entonces reescribirse de la siguiente manera:

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + f_{at} I_{p\lambda} \left[k_d O_{d\lambda} (\overline{N} \cdot \overline{L}) + k_s (\overline{R} \cdot \overline{V})^n \right]$$

Fíjense ahora en otro detalle. El color de la componente especular en el modelo de iluminación de Phong no depende de ninguna propiedad del material. Sin embargo, a veces la reflexión especular se ve afectada por las propiedades de la superficie, y en general, puede tener un color diferente que el de la reflexión difusa, sobre todo cuando el objeto está compuesto por varios tipos de materiales diferentes. Para acomodar este efecto podemos reescribir la ecuación como

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + f_{at} I_{p\lambda} \left[k_d O_{d\lambda} (\overline{N} \cdot \overline{L}) + k_s O_{s\lambda} (\overline{R} \cdot \overline{V})^n \right]$$

donde $O_{s\lambda}$ es el **color especular** del objeto. En la figura 8 podemos ver una esfera iluminada con este modelo para diferentes valores de k_s y n.

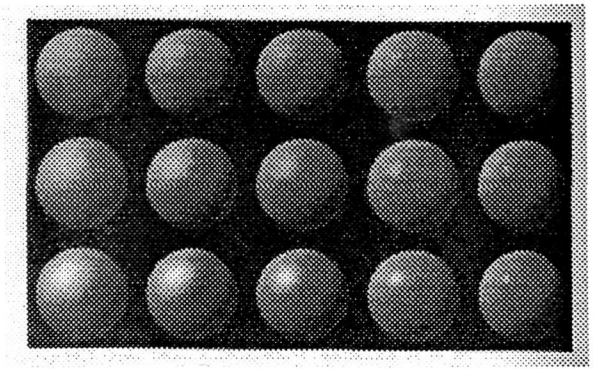


Figure 8

Cálculo del vector de reflexión

Tenemos que $\overline{R}$ es el vector simétrico de $\overline{L}$ con respecto a $\overline{N}$. Como vemos en la figura 9, podemos calcular la expresión para este vector mediante simple geometría.

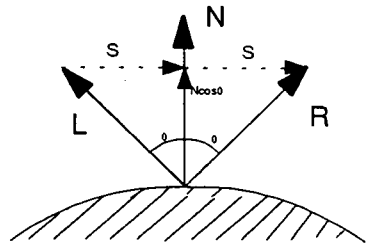


Figure 9

Si suponemos $\overline{N}$ y $\overline{L}$ normalizados, la proyección de $\overline{L}$ sobre $\overline{N}$ es $\overline{N}\cos\theta$. Tenemos entonces que

$$\overline{R} = \overline{N}\cos\theta + \overline{S}$$

donde $\left|\overline{S}\right| = \sin\theta$. Pero este vector $\overline{S}$ podemos ponerlo como

$$\overline{S} = \overline{N}\cos\theta - \overline{L}$$

Por lo tanto

$$\overline{R} = 2\overline{N}\cos\theta - \overline{L} = 2\overline{N}\left(\overline{N}\cdot\overline{L}\right) - \overline{L}$$

Si la fuente de luz se encuentra en el infinito, $\overline{N}\cdot\overline{L}$ es constante para un polígono dado, mientras que $\overline{R}\cdot\overline{V}$ varía a lo largo del polígono.

Múltiples fuentes de luz

Si existen en la escena m fuentes de luz, entonces se deben sumar los términos correspondientes a cada fuente

$$I_\lambda = I_{a\lambda}k_aO_{d\lambda} + \sum_{1\le i\le m} f_{ati}I_{p\lambda i}\left[k_dO_{d\lambda}(\overline{N}\cdot\overline{L_i}) + k_sO_{s\lambda}(\overline{R_i}\cdot\overline{V})^n\right]$$

Esta sumatoria da la posibilidad de un error, debido al hecho que I_λ puede exceder el máximo de intensidad posible para un pixel, por lo que deberá ser tenido en cuenta. La solución más simple es considerar todos los valores de I_λ y dividir por el máximo en caso de que éste supere la máxima de un pixel.

6.2 MODELOS DE SOMBREADO PARA POLÍGONOS

Hasta ahora debe haber quedado claro que para sombrear o iluminar una superficie sólo hace falta calcular la normal a la superficie en cada punto visible y aplicar el modelo de

iluminación deseado a dicho punto. Desafortunadamente, este modelo de sombreado de fuerza bruta es demasiado costoso. Vamos a describir a continuación modelos más eficientes.

6.2.1 Sombreado constante

También conocido como **flat shading**, es el más simple que existe. Consiste en aplicar el modelo de iluminación una sóla vez sobre un punto del polígono para determinar un único valor de intensidad, y usar dicho valor para colorear todo el polígono. Esta aproximación es válida siempre y cuando se den estas tres condiciones:

1) La fuente de luz está en el infinito, por lo que $\overline{N} \cdot \overline{L}$ es constante sobre la cara entera del polígono.

2) El observador también se encuentra en el infinito, por lo que $\overline{N} \cdot \overline{V}$ es constantes sobre el polígono.

3) El polígono representa la superficie real del objeto, y no es una aproximación de una superficie curva.

Si cualquiera de las dos primeras condiciones no se cumplen, entonces, considerar este método requerirá determinar un único valor para el producto de $\overline{L}$ y de $\overline{V}$. Por ejemplo, podemos elegir el centro del polígono, o uno de sus vértices, para calcular el producto en ese punto y suponer que se mantiene fijo para el resto del polígono. Por lo tanto, con este método no obtendremos las variaciones de luz que se produzcan en el interior del polígono y que deberían ocurrir realmente.

Si la tercera condición es la que no se cumple, e iluminamos cada polígono de forma individual y constante, el resultado sería que no veríamos la superficie total del mesh como una superficie suave, sino como una malla de polígonos, como se aprecia en la figura 10. Esto es debido a que dos polígonos adyacente dentro de la malla tendrán diferente orientación, y por lo tanto tendrán diferentes intensidades a lo largo de sus bordes. La solución más simple sería usar una malla de más resolución, pero se seguirían viendo los saltos de intensidad en la imagen.

De lo anterior podemo concluir entonces que este modelo determina el color de cada polígono de forma individual. Pero existen dos modelos de sombreado básicos para meshes poligonales que toman ventaja de la información proveniente por los polígonos adyacentes para simular una superficie suave. Estos modelos son el sombreado de Gouraud y el sombreado de Phong (no confundir con el modelo de iluminación de Phong). Las estaciones gráficas actuales suelen traer implementado en hardware uno de estos modelos.

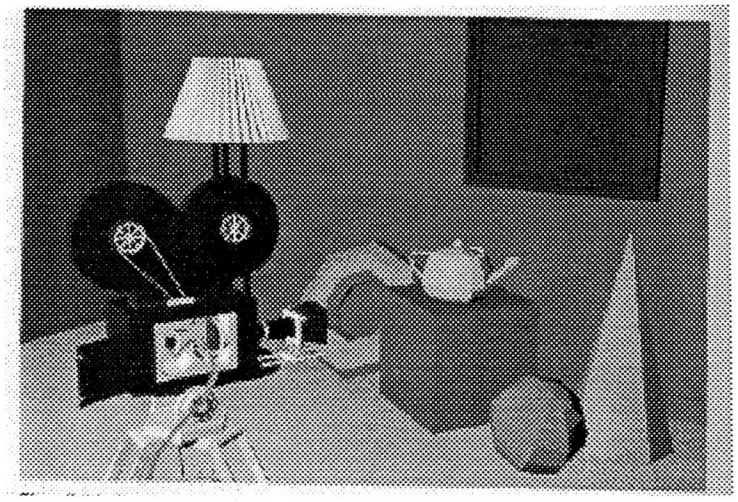


Figure 10

6.2.2 Sombreado de Gouraud

Este modelo de sombreado elimina las discontinuidades de la intensidad en los bordes de los polígonos adyacentes. Se basa en calcular primeramente la iluminación de cada vértice de la malla, y posteriormente interpolar para los puntos interiores a cada polígono dichos valores, de tal manera que no hayan saltos bruscos de intensidad. Este proceso requiere por tanto conocer la normal para cada vértice del mesh. Estas normales pueden calcularse directamente a partir de una descripción analítica de la superficie. Alternativamente, si las normales de los vértices no están almacenadas en la estructura de datos del mesh y tampoco podemos determinarla a partir de la descripción de la superficie, otra aproximación consiste en promediar las normales de todos los polígonos que comparten cada vértice, como se ve en la figura 11.

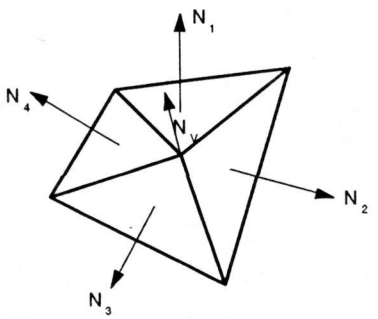


Figure 11

Una vez conocemos las normales a cada vértice, el siguiente paso es aplicar el modelo

de iluminación a cada punto para obteners sus intensidades. Finalmente, sombrearemos cada polígono mediante una interpolación lineal de dichas intensidades, primero a lo largo de cada borde, y posteriormente a lo largo de cada línea de barrido de la pantalla, como se ve en la figura 12.

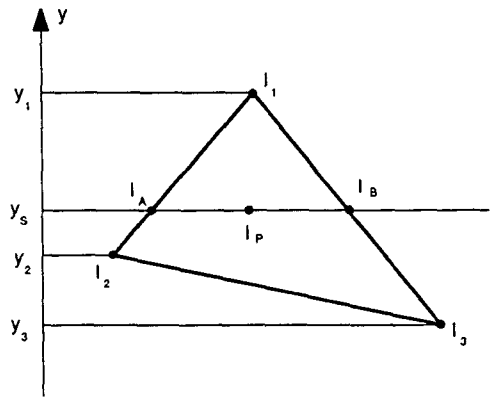


Figure 12

Para calcular el valor de I_p primero calculamos I_a e I_b, y luego interpolamos entre ambas, usando estas expresiones

$$
\begin{aligned}
I_a &= I_1 - (I_1 - I_2)\frac{y_1 - y_s}{y_1 - y_2} \\
I_b &= I_1 - (I_1 - I_3)\frac{y_1 - y_s}{y_1 - y_3} \\
I_p &= I_b - (I_b - I_a)\frac{x_b - x_p}{x_b - x_a}
\end{aligned}
$$

El resultado puede verse en las figuras 13 y 14 (la primera con reflexión difusa y la segunda con reflexión especular).

Esta interpolación puede integrarse fácilmente dentro del algoritmo de scan-line o línea de rastreo que veíamos en el tema anterior. Si almacenamos para cada borde la intensidad del vértice inicial y el cambio de intensidad por cada cambio unitario en y, podemos ir calculando la intensidad de cada línea de barrido mediante interpolación entre los valores de cada borde.

6.2.3 Sombreado de Phong

Este modelo interpola los vectores normales a la superficie en lugar de interpolar las intensidades. El método de interpolación es el mismo que en el de Gouraud, es decir, primero a lo largo de un borde, y luego a lo largo de una línea de barrido. Así, para

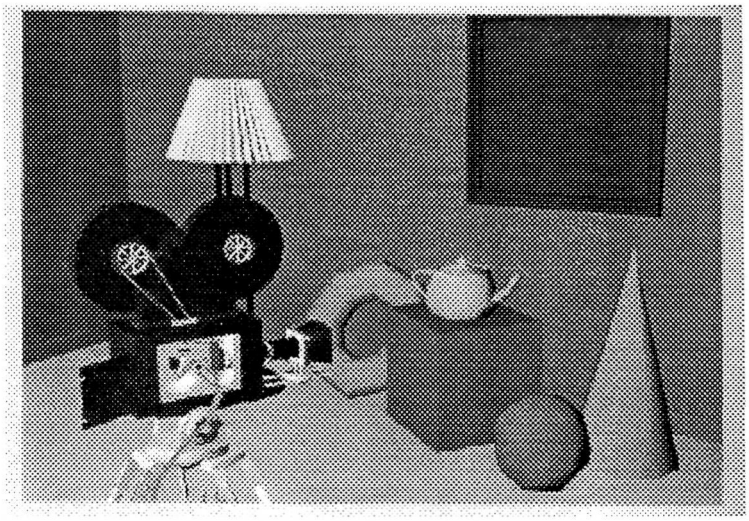


Figure 13

cada pixel podemos obtener su normal, y una vez normalizada, podremos usarla para evaluar el modelo de iluminación para dicho pixel y calcular su intensidad. En la figura 15 podemos ver las normales de dos vértices contiguos y las normales interpoladas a lo largo del borde, antes y después de la normalización.

En la figura 16 se ve un ejemplo de este modelo de sombreado, usando una ecuación de iluminación con un término de reflexión especular.

Si la comparamos con el ejemplo logrado con el modelo de Gouraud podemos observar varias diferencias. Por ejemplo, las zonas de mayor brillo aparecen de forma más realista. Un ejemplo de esto lo tenemos en la figura siguiente. Supongamos que en el término de iluminación de Phong $\cos^n \alpha$, el valor de n es muy grande, y por otra parte, en un vértice tenemos un valor pequeño de α y en los vértices adyacente tenemos valores grandes. La intensidad asociada al vértice de α pequeña estará muy bien iluminada, mientras que los otros vértices se verán apagados (figura 17.b). Si hubiésemos usado el modelo de Gouraud, entonces la intensidad sobre el polígono quedaría interpolada, dando lugar a una zona muy ancha de luz (figura 17.a). Por otro lado, si un pico de luz cae en el interior del polígono y no en los vértices, usando el modelo de Gouraud perderemos totalmente esa iluminación en el polígono (figura 17.c), ya que ningún punto interior puede ser más brillante que el más brillante de los vértices que estamos interpolando, cosa que no ocurriría con Phong (figura 17.d).

Pero no todo son ventajas. La principal desventaja del modelo de Phong frente al de Gouraud es que es más costoso. Mientras que con Gouraud sólo evaluamos la ecuación de iluminación una vez, y luego procedemos a la interpolación, con Phong hay que interpolar primero las normales (el coste de interpolación es aproximadamente el mismo) y luego evaluar la ecuación para cada pixel.

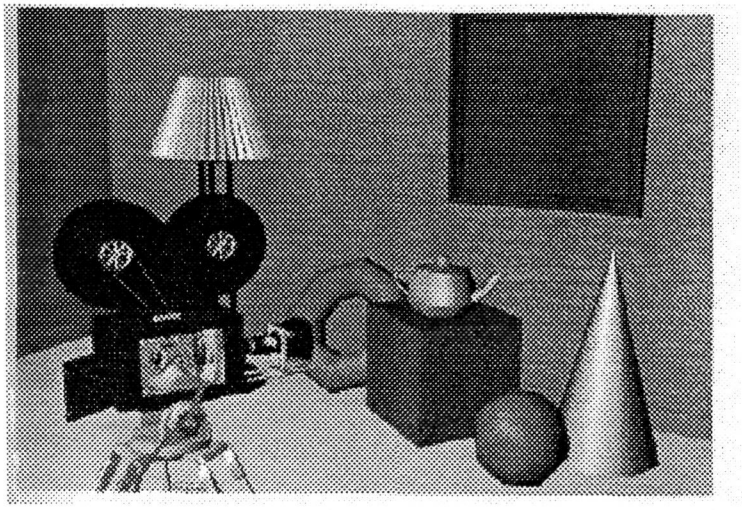


Figure 14

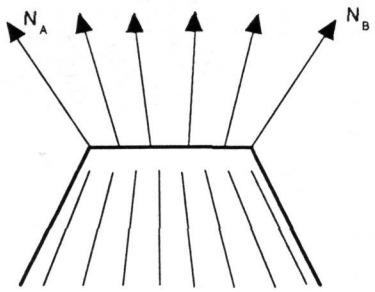


Figure 15

6.2.4 Problemas del sombreado interpolado

Los problemas comunes que nos encontramos en estos dos modelos son:

a) **Silueta poligonal.** No importa lo buena que sea la aproximación de un modelo interpolado para sombrear una superficie curva, que siempre se apreciará la silueta poligonal del mesh claramente (véase figura 16). Podemos solucionarlo descomponiendo en polígonos más pequeños, pero aumentando el coste computacional.

b) **Distorsión de la perspectiva.** Ya que la interpolación se ejecuta tras haber realizado la perspectiva del objeto, aparecen fallos. Por ejemplo, en la figura 12, la interpolación lineal produce que la intensidad se vaya incrementando por una cantidad constante a lo largo de cada borde. Consideremos que el vértice 1 está más lejos (en

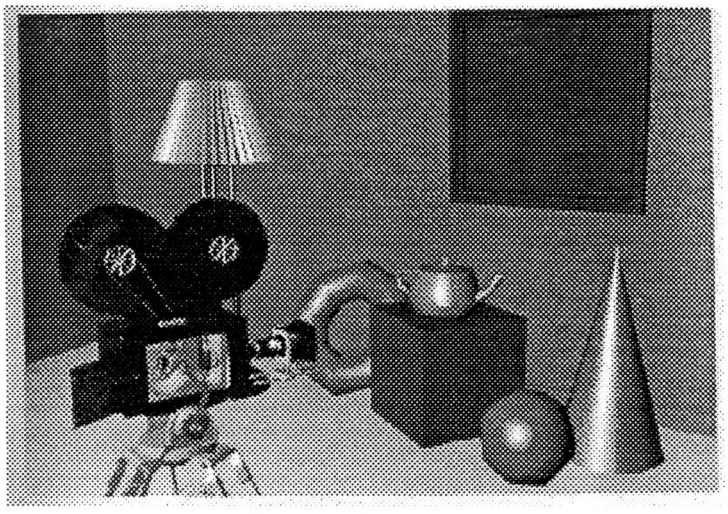


Figure 16

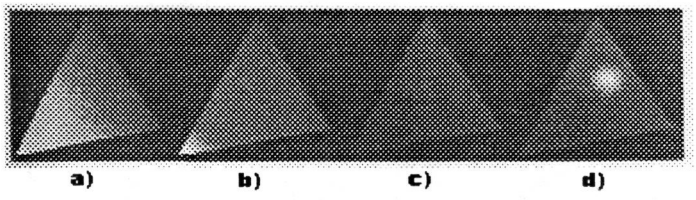


Figure 17

z) que el vértice 2. Entonces,

$$y_s = \frac{y_1 + y_2}{2} \Rightarrow I_s = \frac{I_1 + I_2}{2}, \text{ pero } z_s \neq \frac{z_1 + z_2}{2}$$

c) **Dependencia de la orientación**. Los resultados de la interpolación dependen de la orientación del polígono proyectado. Ya que los valores se interpolan entre vértices y a través de líneas horizontales, los resultados pueden variar si rotamos el polígono (véase figura 18). Este efecto se nota sobre todo cuando la orientación cambia suavemente entre los frames sucesivos de una animación. Esto no ocurre si todos los polígonos fuesen triángulos.

d) **Problemas en los vértices compartidos**. Pueden aparecer discontinuidades cuando dos polígonos adyacentes compartan un vértice que descanse sobre un borde común. En la figura 19, el vértices C está siendo compartido por los dos polígonos de la derecha pero no por el de la izquierda. La información calculada para C directamente no coincidirá seguramente con la calculada por interpolación entre los vértices A y B, lo que provocará una discontinuidad. Para solucionar este error habrá que insertar un vértice extra.

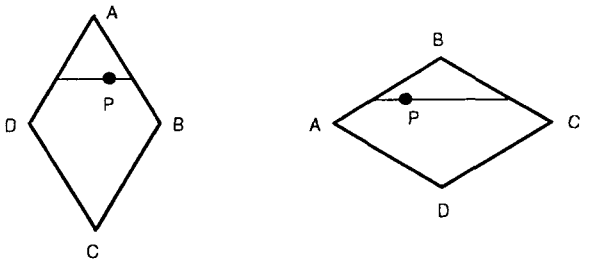


Figure 18

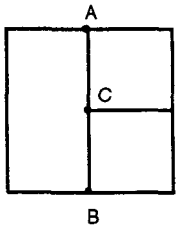


Figure 19

e) **Errores en las normales a los vértices.** Las normales calculadas pueden no ser las correctas para representar la geometría de la superficie. Por ejemplo, si calculamos las normales de los vértices de la figura 20 promediando las normales de las caras adyacentes, todas las normales nos saldrían paralelas, resultando que no habría variación de intensidad si la fuente de luz es distante. Para solucionarlo habrá que dividir en más polígonos.

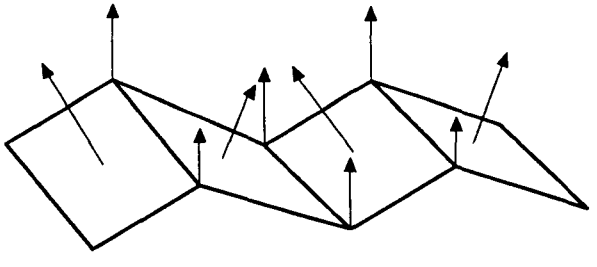


Figure 20

Aún a pesar de todos estos inconvenientes que producen que los algoritmos de rendering tengan que hacer mucho más gasto computacional para resolverlos, los algoritmos orientados a polígonos siguen siendo más fáciles y rápidos de procesar que los

algoritmos que trabajan con las superficies curvas directamente, a pesar de no tener todos estos problemas.

6.3 DETALLES DE LA SUPERFICIE

Si aplicamos cualquiera de los modelos de sombreado anteriores a superficies planas o bicúbicas, obtendremos superficies suaves y uniformes, en contraste con la mayoría de superficies existentes en la realidad, en donde las superficies representan muchos detalles concretos. Veamos cómo podemos simular algunos de estos detalles.

6.3.1 Polígonos de detalle

Es la aproximación más simple para mostrar ciertas características (como puertas, ventanas y letreros) sobre un polígono base (la pared de un edificio). Cada polígono de detalle es coplanar con su polígono base, y no necesita compararse con los demás cuando ejecutemos los algoritmos de detección de la superficie visible. Cuando el polígono vaya a colorearse, sus polígonos de detalle y sus propiedades de material tendrán preferencia para aquellas partes del polígono base que vayan a ser cubiertas por éstos.

6.3.2 Mapeo de texturas

Cuando el nivel de detalle de un objeto es muy fino, el modelado con polígonos se hace muy costoso. Lo mejor en esos casos es mapear una imagen, sea digitalizada o sintética, sobre una superficie. A la imagen la llamaremos **mapa de texturas**, y a sus elementos individuales o pixels los llamaremos **texels**. El mapa de texturas rectangular viene definido según su sistema de referencia (u, v). También puede definirse la textura de forma procedural. En las figuras 21 y 22 puede verse un ejemplo.

Para cada pixel de la imagen que estamos creando, vemos a qué texels corresponde, y usamos estos valores para sustituir o escalar una o varias propiades del material, como por ejemplo sus componentes de color difusas. Como vemos en la figura 23, el mapeo de texturas se realiza en tres pasos: en primer lugar, mapeamos las cuatro esquinas del pixel sobre la superficie. Para un patch bicúbico, este mapeo definirá un conjunto de puntos en el espacio de coordenadas (s, t) de la superficie. En segundo lugar, esos puntos se mapean en el espacio de coordenadas (u, v) de la textura. Los cuatro puntos (u, v) del mapa de texturas definen un cuadrilátero. Finalmente, todos los texels de dentro del cuadrilátero se suman, pesando cada uno por la fracción de texel que aparece dentro del cuadrilátero.

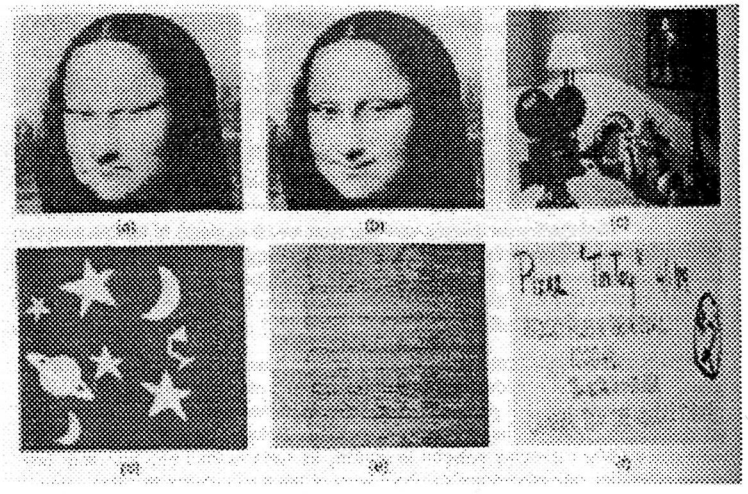


Figure 21

6.3.3 Mapeo de rugosidades (bump maping)

El mapeo de texturas afecta sólamente al sombreado de la superficie, pero ésta sigue siendo geométricamente suave. Si el mapa de textura es una fotografía de una superficie rugosa, la superficie sintética no dará el pego, ya que la dirección de la luz que se usó para crear el mapa de textura seguramente será diferente de la dirección de la luz que está iluminando la superficie. Pero existe un método que nos va a permitir simular la aparencia de una geometría modificada que evite el modelado geométrico explícito. Esta técnica consiste en perturbar la normal de la superficie antes de usarla en el modelo de iluminación, de la misma forma que una rugosidad real sobre una superficie altera el valor de la normal. Este método se llama bump maping.

Llamaremos bump map a un array bidimensional de desplazamientos, cada uno de los cuales puede usarse para simular el desplazamiento de un punto sobre una superficie, ligeramente hacia un lado o hacia otro de su posición actual. El procedimiento consiste en ir variando todas las normales según el bump map, y usando las nuevas normales en la ecuación de iluminación. Los resultados son muy convincentes, como puede verse en la figura 24. Un inconveniente es que la textura del objeto no afecta a su silueta.

6.4 SOMBRAS

Los algoritmos de superficie visible determinan qué superficies pueden ser vistas desde la posición del observador. Pues bien, un algoritmo de sombras es casi lo mismo: determina qué superficies son vistas desde la fuente de luz. Las que no sean vistas significará que se encuentran en sombras. Cuando en la escena tenemos varias fuentes de luz, habrá que clasificar las superficies relativas a cada una de ellas.

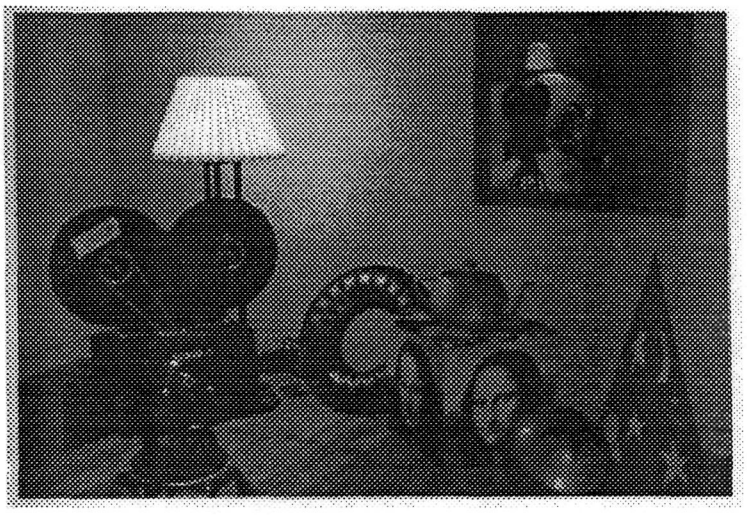


Figure 22

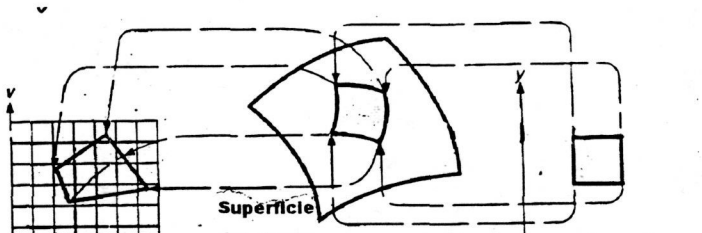


Figure 23

Considerando sólamente fuentes de luz puntuales[2], la visibilidad de un punto de la escena es una variable booleana: o le da la luz o no le da. Cuando un punto no es visto por una fuente de luz, entonces el cálculo de la iluminación debe tenerlo en cuenta. Si queremos añadir sombras en la ecuación de iluminación, ésta quedaría:

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + \sum_{1 \le i \le m} S_i f_{ati} I_{p\lambda i} \left[k_d O_{d\lambda} (\overline{N} \cdot \overline{L_i}) + k_s O_{s\lambda} (\overline{R_i} \cdot \overline{V})^n \right]$$

donde S_i vale 0 si el punto no ve a la luz, y 1 en caso contrario. Fíjense que las áreas donde no dé directamente ninguna fuente de luz sigue estando iluminada por la luz ambiente.

[2]Si la fuente de luz tuviera un área, la tarea sería más complicada, pues habrían zonas de sombra y zonas de penumbra.

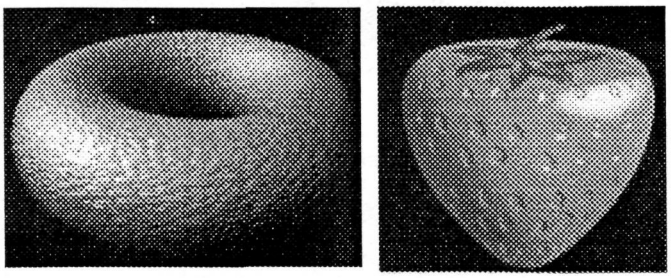


Figure 24

6.4.1 Algoritmo de sombras de doble pasada

Determina las sombras antes de determinar las superficies visibles. Para ello se procesa la descripción de la escena usando el mismo algoritmo dos veces, primero desde la fuente de luz, y luego desde el observador. Como el cálculo de sombras no depende del observador, puede ejecutarse sólo una vez cuando haya que calcular varias imágenes de la misma escena desde varios puntos de vista.

El resultado de la primera fase es una lista de polígonos (de trozos de los polígonos originales). Estos nuevos polígonos son transformados de nuevo al sistema de coordenadas global, y se mezclan con la base de datos original de la escena como si fueran polígonos de detalle, crendo así una base de datos independiente del punto de vista (figura 25).

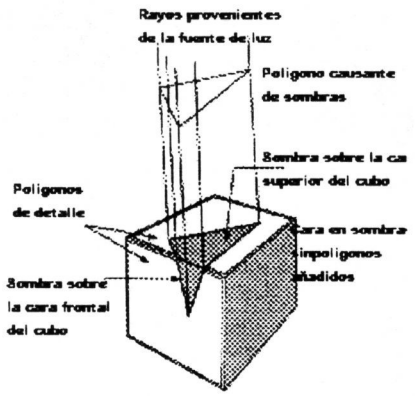


Figure 25

Finalmente, aplicamos un algoritmo de superficies ocultas y un algoritmo de iluminación. Las superficies visibles que estén cubiertas por polígonos de detalle serán tratadas como que les da la luz directamente, mientras que las zonas sin polígono de detalle serán consideradas como sombras. Cuando tengamos múltiples fuentes de luz,

simplemente hay que hacer una pasada más de la base de datos para cada fuente.

6.4.2 Volúmenes de sombras

Esta técnica describe la generación de sombrar a partir de la creación de un **volumen de sombra** para cada objeto. Este volumen consiste en todo el espacio de la escena que se encuentra sin luz debido a que el objeto la está tapando, y viene definido por la fuente de luz y por el objeto en cuestión, aparte de por un conjunto de **polígonos de sombra** invisibles, como se ve en la figura 26.

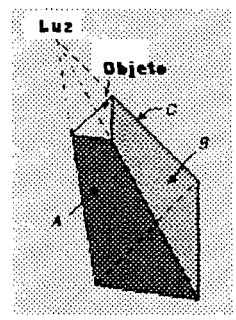


Figure 26

Como se ve, existe un polígono de sombra cuadrilátero para cada borde de la silueta del objeto relativo a la fuente de luz. Uno de los lados es el propio borde del objeto, y dos lados más están formados por dos líneas que salen de la fuente de luz y pasan por cada extremo del borde. Los volúmenes de sombra sólo se calculan para los polígonos orientados hacia la luz. La parte de arriba del volumen viene dada por el polígono objeto original, y la otra parte viene dada por una copia escalada del polígono original. Esta copia se desplaza una distancia desde la luz tal que consideremos que a esa distancia la luz no afecta a la escena.

Los polígonos de sombra no van a aparecer en la imagen final de la escena, pero nos van a indicar qué objetos se encuentran en sombra y cuáles no. Por ejemplo, en la figura 26, un polígono de sombra enfrentado hacia el observador (cara anterior), como el A o el B provoca que los objetos de detrás de él estén en sombra. Un polígono de sombra contrario al observador (cara posterior), como el polígono C, cancela este efecto. Consideremos un vector que vaya desde el punto de vista del observador V hasta un punto concreto de la escena. Este punto estará en sombra si el vector intersecta más polígonos de sombra anteriores que posteriores. De esta manera, en la figura 27.a), los puntos A y C están en sombra, y el punto B está iluminado.

Por otro lado, si V está en sombra, hay un caso adicional donde el punto puede estar en sombra: cuando todos los polígonos de sombra posteriores de los que sombrean

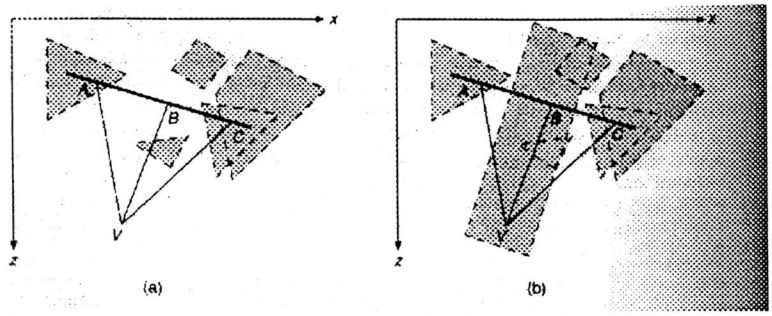


Figure 27

a V no han sido todavía encontrados. Así, en la figura 27.b), los tres puntos A, B y C están en sombra.

Aunque es posible calcular un volumen de sombra para cada polígono de la escena, es mucho más rentable calcular un simple volumen para cada poliedro conectado. De esta manera, el volumen de sombra se generaría a partir de aquellas aristas que fuesen bordes de la silueta relativa a la fuente de luz, esto es, el contorno del objeto.

6.4.3 Algoritmos de sombras de z-buffer

Esta técnica consiste en ejecutar dos veces el algoritmo de z-buffer: primero desde la fuente de luz y luego desde el observador. En la figura 28.a) tenemos una escena iluminada desde el punto L. En la figura 28.d) tenemos la imagen vista desde V pero sin sombras.

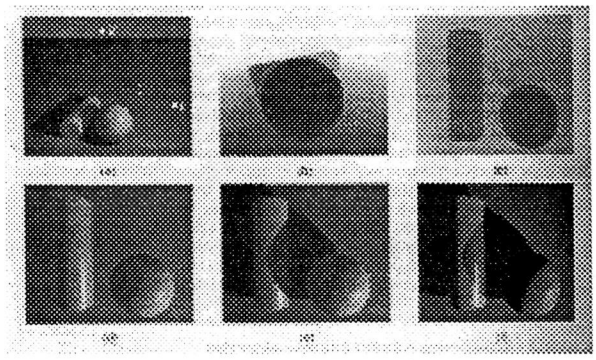


Figure 28

El algoritmo comienza calculando y almacenando el z-buffer de la imagen desde el punto de vista de la fuente de luz (figura 28.b) (cuanto más claro más lejos). Luego, se calcula el z-buffer (figura 28.c) y la imagen (figura 28.e) desde el punto de vista del

observador usando un algoritmo z-buffer con la siguiente modificación: para cada pixel visible, sus coordenadas (x_0, y_0, z_0) con referencia al observador son transformadas a las coordenadas (x'_0, y'_0, z'_0) con referencia a la luz. Estas coordenadas x'_0 e y'_0 se usan para seleccionar el valor z_L en el z-buffer rerente a la luz para compararlo con el valor z'_0. Si z_L está más cerca que z'_0, entonces debe haber algún objeto que bloquea la luz desde el punto, y por tanto el pixel se considera en sombra. De lo contrario, el punto está iluminado por la fuente.

Haciendo una analogía con el mapeo de texturas, podemos pensar en el z-buffer de la luz como un **mapa de sombras**. En la figura 29 se ve el mapa de sombras usado para generar nuestra escena ejemplo.

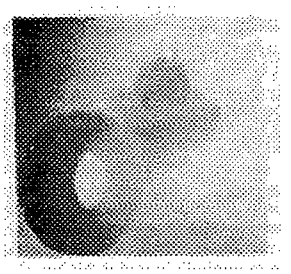


Figure 29

6.5 TRANSPARENCIAS

Al igual que muchas superficies pueden tener reflexión especular y reflexión difusa, aquéllas que dejan pasar la luz pueden ser transparentes o translúcidos. Los objetos **transparentes**, como el cristal, permiten una visión clara de lo que hay que detrás, aunque en general existe un efecto de refracción. Los objetos **translúcidos**, como el cristal escarchado, poseen una transmisión difusa. Los rayos se entremezclan al pasar por este tipo de superficies, y los objetos de detras aparecen como desenfocados.

6.5.1 Transparencia no refractiva

La aproximación más simple para simular la transparencia ignora la refracción, por lo que los rayos de luz no ven alterada su dirección al atravesar la superficie. Así, todo aquello que sea visible en la línea de vista a través de la superficie transparente está además geométricamente situado en esa misma línea. Consideremos la figura 30, donde el polígono transparente 1 está por delante del polígono 2.

Un método típico para determinar la intensidad del pixel en la intersección de la proyección de ambos polígonos consiste en aproximar mediante interpolación lineal sus

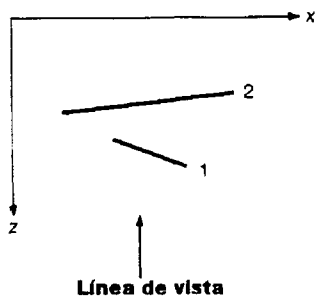


Figure 30

intensidades individuales calculadas para ambos:

$$I_\lambda = (1 - k_{t1})I_{\lambda 1} + k_{t1}I_{\lambda 2}$$

El **coeficiente de transmisión** k_{t1} mide la transparencia del polígono 1, y su valor está entre 0 (opaco) y 1 (totalmente transparente).

Muchos de los algoritmos de visibilidad que vimos en el capítulo anterior pueden adaptarse para incorporar el cálculo de transparencias. Por ejemplo, en el z-buffer, podemos dejar para el final a todos los polígonos transparentes, combinando sus colores con aquellos que ya están en el buffer, pero sin modificar el z-buffer.

6.5.2 Transparencia refractiva

Es más difícil de modelar que la anterior, ya que las líneas geométricas y ópticas de visión no coinciden. Si consideramos refracción en la figura 31, veríamos el objeto A a través de la línea de visión indicada. Si por el contrario no se considera refracción, el objeto que veríamos sería el B.

La relación entre el ángulo de incidencia θ_i y el ángulo de refracción θ_t viene dado por la ley de Snell

$$\frac{\sin\theta_i}{\sin\theta_t} = \eta_{r\lambda}$$

donde $\eta_{r\lambda}$ es el **índice de refracción** del material a través del cual pasa la luz. Cuando este coeficiente vale 1 no existe refracción. En cado de que exista, este coeficiente tomará valores mayores que 1.

6.6 ALGORITMOS DE ILUMINACIÓN GLOBAL

Un modelo de iluminación calcula el color en un punto en función de la luz directamente emitida sobre él por las fuentes de luz, y también en función de la luz que llega al punto

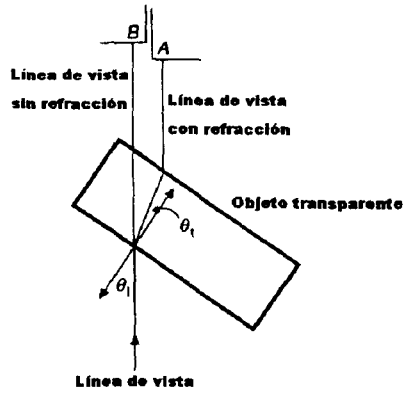


Figure 31

después de diversas reflexiones y transparencias debido a otras superfices en la escena. Esta luz indirectamente reflejada y transmitida es conocida como **iluminación global**. Por el contrario, la iluminación local es la luz que provienen directamente desde las fuentes de luz en dirección a dicho punto. Una vez dicho esto, podemos ver que hemos intentado simular la iluminación global mediante un término de iluminación ambiental que era constante para todos los puntos de todos los objetos. No dependía de las posiciones de dichos objetos ni de la del observador. Tampoco dependía de la presencia o ausencia de objetos cercanos que pudieran bloquear la luz ambiente.

La mayor parte de la luz que inunda una escena real no proviene directamente de las fuentes de luz, sino que procede de múltiples reflexiones y refracciones entre todos los objetos existentes en la escena. Llamaremos **algoritmos de iluminación global** a aquéllos que consideren la totalidad de la escena para calcular las intensidades de cada punto visible, teniendo en cuenta todos los efectos ópticos que puedan darse entre los objetos. Dos son los métodos que hasta la fecha han aparecido para tratar de simular estos efectos: ray-tracing y radiosidad.

El método de ray-tracing es un método dependiente de la posición del observador, que discretiza el plano de vista en pixels. Para cada pixel irá lanzando un rayo que partiendo del ojo cruce a través del pixel e intersecte con un objeto para calcular su color, considerando sombras, reflexiones y refracciones. Este método va muy bien para manejar fenómenos especulares que sean altamente dependientes de la posición del observador, pero necesitan mucho cálculo extra para simular fenómenos difusos donde haya un pequeño cambio de luz a lo largo de áreas grandes de la imagen.

La radiosidad por el contrario es un método independiente del observador. En lugar de discretizar el plano de vista, se discretizan todas las superficies de la escena en pequeños patches, y procede a evaluar la ecuación de iluminación para todos los patches de la escena, considerando el intercambio de energía lumínica entre él y todo el resto. Una vez calculada la intensidad de cada patch, y determinada la posición

del observador, puede calcularse la imagen final por medio de un algoritmo de visibilidad. Simula de forma muy eficiente los fenómenos difusos, pero requiere una capacida inmensa de almacenamiento para conseguir simular los fenómenos especulares.

Veamos a continuación con más profunidad el método de ray-tracing.

6.7 RAY-TRACING

Es actualmente la simulación más completa de un modelo de iluminación-reflexión. Los modelos de reflexión local que hemos visto anteriormente pueden interpretarse como simplificaciones del ray-tracing. En ambos modelos, la luz se simula mediante rayos infinitamente delgados que viajan en línea recta a través de un medio homogéneo. La diferencia entre un simple modelo de reflexión y un modelo de ray-tracing es la "profundidad" de la interacción entre los rayos de luz y los objetos de la escena.

La filosofía básica es que un observador ve un punto sobre una superficie como resultado de la interacción de la superficie en ese punto con rayos que emanan desde cualquier lugar de la escena. En los modelos de reflexión simples, sólamente se considera la interacción de los puntos de la superficie con la iluminación directa proveniente de las fuentes de luz. En general, un rayo de luz puede alcanzar la superficie indirectamente vía reflexión en otras superficies, vía transmisión a través de objetos parcialmente transparentes, o vía una combinación de ambas. Esto es lo que definíamos como iluminación global.

El método posee un número de desventajas. La más importante es su extremado coste computacional. Una imagen puede tardar muchos minutos, horas o incluso días para ser calculada. Su principal ventaja es que es una solución parcial al problema de la iluminación global, combinando elegantemente en un único modelo la eliminación de superficies ocultas, cálculo de la iluminación directa, cálculo de la iluminación global, y cálculo de sombras.

6.7.1 Algoritmo básico

El algoritmo trabaja por completo en el espacio de la escena. Para cada punto del plano de la imagen, las superficies visibles, y por tanto, su color e intensidad, se obtienen lanzando un rayo desde el ojo que atraviese el punto de la imagen y penetre en la escena. Si este rayo intersecta un objeto, entonces se evalúa la iluminación local para obtener el color resultado de la iluminación directa en ese punto, es decir, debido a las fuentes de luz que dan sobre la superficie. Si el objeto es parcialmente reflectivo, parcialmente transparente o ambos, el color del punto en la imagen incluirá entonces alguna proporción de estos dos nuevos rayos, reflejado y transmitido. Estos dos rayos deben trazarse entonces a continuación para poder calcular su valor.

Determinar un color para cada uno de estos rayos puede requerir que trazemos aún

más rayos en las siguientes intersecciones que nos encontremos. Para determinar el color del punto original sobre la imagen tendremos que tener calculados todos estos rayos anteriormente. El trazado de un rayo particular termina cuando no intersecta con ningún objeto de la escena, o cuando ya han ocurrido tantas intersecciones con otras superficies que su contribución al valor final del pixel va a ser despreciable.

El algoritmo consiste entonces en un bucle que barre todos los pixels de la imagen, donde para cada pixel se comienza trazando un rayo entre el ojo y el centro del pixel. Por lo tanto, el cálculo para cada rayo es independiente del resto de rayos, por lo que la implementanción en paralelo de este algoritmo es trivial.

La técnica de ray-tracing, como veíamos en el capítulo anterior, consistía inicialmente en un algoritmo de visibilidad. En este caso, el algoritmo paraba en la primera intersección del rayo con la superficie (véase figura 32). Tal intersección determinaba los puntos visibles en la imagen. Una simple implementación consistía en buscar la intersección entre el rayo y todas las superficies de la escena, y quedarnos con la más cercana al observador.

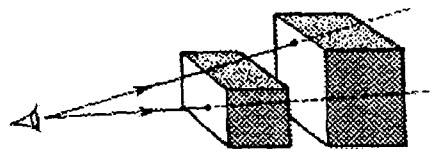


Figure 32

Si aplicásemos este algortimo a una escena que contenga objetos reflectivos, el resultado no sería satisfactorio. Aquellas superficies de un objeto que quedasen ocultas al observador si dicho objeto estuviera aislado, podría reflejarse en la superficie frontal de un objeto cercano (véase figura 33).

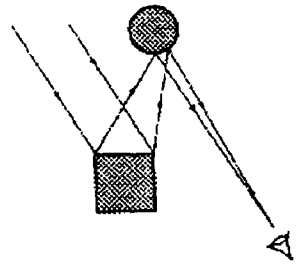


Figure 33

6.7.2 Implementación recursiva

Fijémonos en la figura 34. El algoritmo comienza lanzado un rayo (rayo 1) desde el ojo a través de uno de los pixels de la imagen hacia el interior de la escena, la cual está compuesta por tres esferas semi-transparentes (que reflejan y refractan). El color final del pixel será el resultado de un análisis de sus intersecciones con la superficie del primer objeto encontrado a lo largo del rayo.

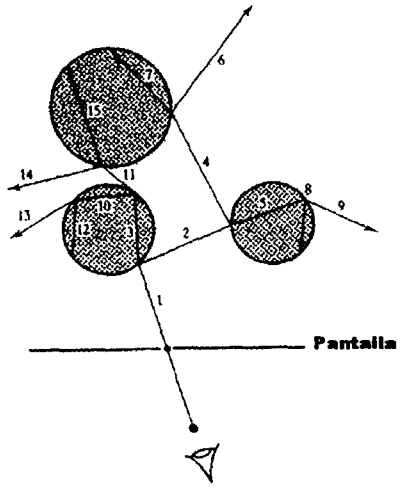


Figure 34

Este color proveniente de la superficie que haya intersectado en primer lugar, va a estar formado por tres contribuciones:

a) el color local debido a la iluminación directa de las fuentes de luz,

b) el color debido a la reflexión de un rayo (rayo 2) proveniente de la dirección de reflexión, y

c) el color debido a la transmisión de un rayo (rayo 3) proveniente de la dirección de refracción.

Para calcular el color debido al rayo 2 habrá que trazar un nuevo rayo en la dirección indicada y ver qué ocurre en la primera intersección que se encuentre. Para calcular el color en este nuevo punto de intersección, habrá que volver a tener en cuenta las tres contribuciones: local, reflejada y transmitida. Las dos últimas se calculan lanzando dos nuevos rayos (4 y 5), y así sucesivamente. De la misma manera se procede con el rayo 3 y con los posteriores. En la figura 35 puede verse el árbol completo de análisis de intersección necesarios para obtener el color percibido por el rayo 1, donde hemos asumido una profundidad máxima de 4. Es decir, después de cuatro intersecciones, cualquier contribución al valor final del color del pixel será despreciable.

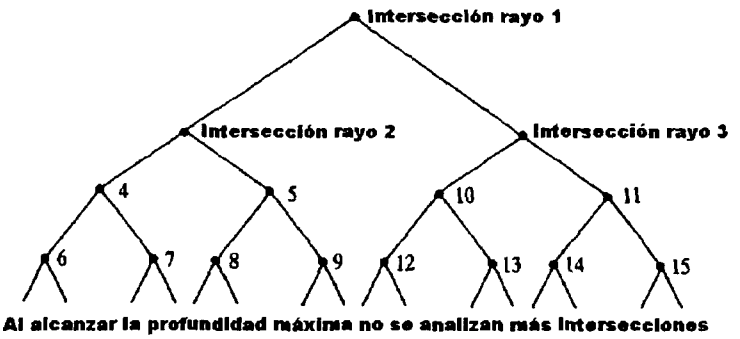


Figure 35

De cara a la implementación recursiva propiamente dicha, para descubrir el color del rayo 1 podríamos llamar a un procedimiento recursivo con los parámetros que representan el punto inicial y la dirección del rayo, junto con un valor de profundidad que indique por cuantas intersecciones anteriores hemos pasado. El algoritmo podría ser el siguiente:

Procedure TrazaRayo (comienzo, dirección, profundidad, color)
if profundidad > max.profundidad then color=negro;
else
color.local = contribución del modelo de color local;
TrazaRayo (p. de intersección, d. de reflexión, prof.+1, color.reflejado);
TrazaRayo (p. de intersección, d. de refracción, prof.+1, color.refractado);
Combinar (color, color.local, color.reflejado, color.refractado);
end

En la figura 36 podemos ver una imagen generada por este algoritmo a partir de una escena con tres esferas. En la primera imagen se ha usado una profundidad máxima de 6. Las figuras siguientes se han obtenido haciendo que la profundidad máxima tome los valores 1, 2, 3 y 4.

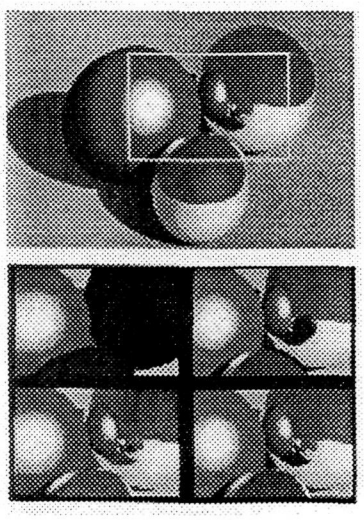


Figure 36

BIBLIOGRAFÍA BÁSICA

"3D Computer Graphics". Alan Watt. Addison-Wesley. 1993
"Computer Graphics". Foley, vanDam, Feiner, Hughes. Addison-Wesley. 1990
"Computer Graphics". Hearn, Baker. Prentice Hall. 1994

BIBLIOGRAFÍA COMPLEMENTARIA

"3D Computer Animation". John Vince. Addison-Wesley. 1992
"Advanced Animation and Rendering Techniques". Watt, Watt. Addison-Wesley. 1992
"Computer Graphics". F.S.Hill. Maxwelll MacMillan. 1990
"Mathematical Elements for Computer Graphics". Rogers, Adams. McGraw Hill. 1989
"Three-Dimensional Computer Animation". Michael O'Rourke. W.W.Norton. 1995
"Procedural Elements for Computer Graphics". David Rogers. McGraw Hill. 1985