Univ. Michoacana de San Nicolas de Hgo.
Facultad de Ingeniería Eléctrica
Notas de Graficación

José Antonio Camarena Ibarrola

Marzo de 2010

Índice general

Capítulo 1

Introducción

Estas notas fueron desarrolladas como apoyo a los estudiantes que cursan la materia de Graficación en la Carrera de Ingeniería en Computación. El programa de esta materia incluye temas de diversos libros los cuales son difíciles de conseguir y de algunos artículos científicos, las presentes notas son un compendio que se apega a los temas que indica el programa. Los ejemplos que se incluyen han sido desarrollados a mucho mayor detalle que lo que usualmente aparece en la bibliografía. Finalmente sucede sobre todo en los estudiantes de Licenciatura que se les dificulta la comprensión de material escrito en Inglés, estas notas son un apoyo también para aquellos alumnos con dificultades de lectura del inglés técnico del área de Graficación. Estas notas se encuentran a disposición de los estudiantes o de cualquier interesado en los temas de este curso en http://lc.fie.umich.mx/∼camarena/NotasGraficacion.pdf. Se agradecen las observaciones y comentarios, favor de dirigirlos a camarena@umich.mx

Atentamente: Dr. José Antonio Camarena Ibarrola. Autor

Los algoritmos de graficación resuelven problemas relacionados con las aplicaciones gráficas en donde una computadora digital es de gran ayuda. Ejemplos de tales aplicaciones son:

- Despliegue de funciones matemáticas de dos variables que difícilmente se pueden dibujar a mano.
- Realización de prototipos. Antes de construir un dispositivo, equipo o incluso automóviles y aviones es importante que sus diseños puedan apreciarse y corregirse de manera eficaz y económica.
- Simulación por computadora. Para saber si una solución a un problema de ingeniería realmente funciona resulta muy práctico observar la solución en acción, un ejemplo de esto son los simuladores de tráfico donde al construir digamos un distribuidor vial podemos validar su correcto diseño o jugar con el para hacerle mejoras
- Video-juegos. Muchos programas de simulación terminan luego convirtiéndose en video-juegos. Sin embargo, los video-juegos son una industria por si misma que ha evolucionado a grandes pasos.
- Animación por computadora. En realidad los video-juegos son una rama de la animación por computadora donde el usuario interactúa, sin embargo, la animación por computadora ha permitido la producción de películas en menor tiempo, a menor costo y con mejores efectos especiales.
- Recorridos virtuales. La arquitectura se ha beneficiado de esta rama, en lugar de hacer una maqueta, un arquitecto puede diseñar un inmueble y presentarlo a un cliente dándole un paseo virtual donde es posible realizar modificaciones a su gusto.

Capítulo 2

Primitivas de Graficación

2.1. Algoritmos de trazado de líneas

La línea recta tiene la forma:

$$y = mx + b \tag{2.1}$$

donde m es la pendiente de la recta definida como:

$$m = \frac{\Delta y}{\Delta x} \tag{2.2}$$

b es el punto donde la recta intersecta al eje y (la recta $x = 0$) y la pendiente m es grande si a un Δx pequeño le corresponde un Δy grande como se muestra en la Figura 2.1.

Si deseamos trazar una recta de (x_{ini}, y_{ini}) a (x_{fin}, y_{fin}), entonces:

$$m = \frac{y_{fin} - y_{ini}}{x_{fin} - x_{ini}} \tag{2.3}$$

En base a esto, podemos pensar en un algoritmo simple para trazar una recta, el cual consiste en ir dando a x todos los valores enteros consecutivos desde x_{ini} hasta x_{fin} y para cada valor de x determinar el valor de y correspondiente. El k-ésimo valor de y (y_k) en términos de x_k (el k-ésimo valor de x) está dado por:

$$y_k = mx_k + b \tag{2.4}$$

Para el siguiente cálculo tendríamos:

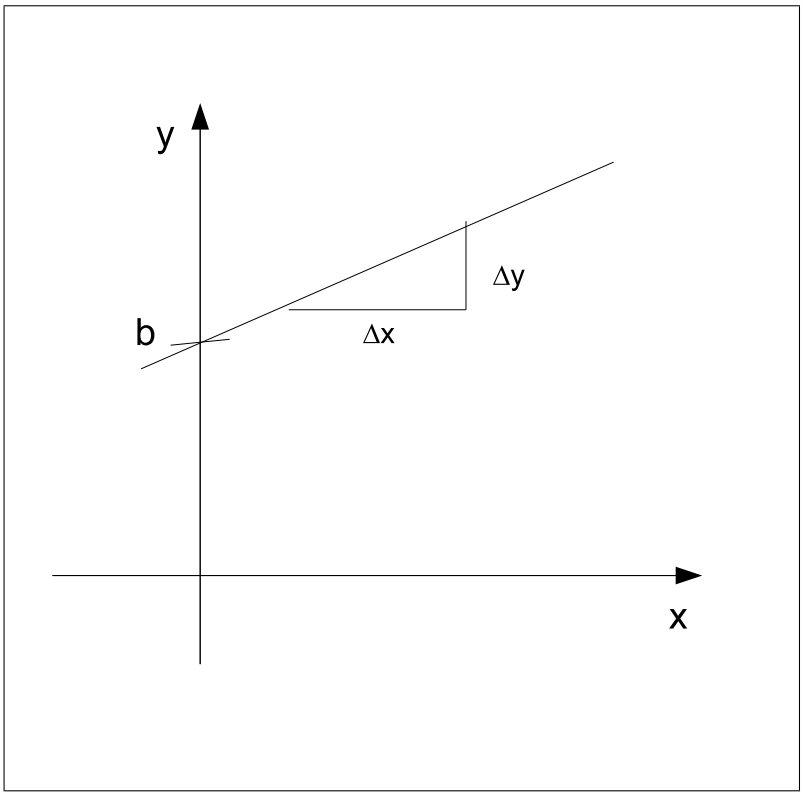

Figura 2.1: La recta

$$y_{k+1} = mx_{k+1} + b \tag{2.5}$$

Como dijimos x se incrementa en uno cada vez por lo tanto:

$$x_{k+1} = x_k + 1 \tag{2.6}$$

Podemos entonces ahorrar cálculos al expresar y_{k+1} en términos de y_k como a continuación:

$$y_{k+1} = m(x_k + 1) + b = (mx_k + b) + m \tag{2.7}$$

De donde:

$$y_{k+1} = y_k + m \tag{2.8}$$

El algoritmo entonces consiste en hacer $y_1 = y_{ini}$ y luego hacer usar sucesivamente (2.8) y (2.6) hasta que x_k sea igual a x_{fin}

Input: x_{ini},y_{ini},x_{fin},y_{fin}
Output: Traza Línea recta
if $x_{fin} > x_{ini}$ **then**
  swap(x_{fin},x_{ini});
  swap(y_{fin},y_{ini});
end
$k = 1$;
$x_k = x_{ini}$;
$y_k = y_{ini}$;
plot(floor(x_k),floor(y_k));
while $x_k \leq x_{fin}$ **do**
  $k = k + 1$;
  $y_k = y_{k-1} + m$;
  $x_k = x_{k-1} + 1$;
  plot(floor(x_k),floor(y_k));
end

Algorithm 1: Algoritmo simple para trazar una recta

2.1.1. Algoritmo DDA (Digital Diferential Analizer)

El problema con el Algoritmo 1 simple de trazado de rectas es que a medida que cambia la pendiente de una recta, el número de pixels cambia a

pesar de tratarse de rectas de la misma longitud, esto implica que la brillantez de la recta depende de su pendiente, para remediar esto, se debe elegir como variable de rastreo a la variable que vaya a tener mayor variación, es decir, si $\Delta x > \Delta y$ se debe elegir a x como variable de rastreo y de lo contrario a y, nos referimos como variable de rastreo a la que se incrementa en uno en cada iteración, el Algoritmo DDA hace esto y hace uso de la función signo para que el mismo código se pueda utilizar independientemente de la pendiente de la recta o del cuadrante en el que se quiere trazar la recta.

Input: x_{ini},y_{ini},x_{fin},y_{fin}
Output: Traza Línea recta
if $|x_{fin} - x_{ini}| \geq |y_{fin} - y_{ini}|$ **then**
  longitud=$|x_{fin} - x_{ini}|$;
else
  longitud=$|y_{fin} - y_{ini}|$;
end
$\Delta x = (x_{fin} - x_{ini})/longitud$;
$\Delta y = (y_{fin} - y_{ini})/longitud$;
$x = x_{ini} + 0{,}5 * Signo(\Delta x)$;
$y = y_{ini} + 0{,}5 * Signo(\Delta y)$;
$k = 1$;
while $k \leq longitud$ **do**
  plot(floor(x),floor(y));
  $x = x + \Delta x$;
  $y = y + \Delta y$;
  $k = k + 1$;
end

Algorithm 2: Algoritmo DDA

2.1.2. Algoritmo de Bresenham

El problema con el Algoritmo DDA es que hace uso de aritmética de punto flotante y eso lo hace lento y difícil de implementar en hardware. El algoritmo de Bresenham [1] hace uso de aritmética entera, para entenderlo veamos primero una versión preliminar que todavía utiliza flotantes. Si se quiere trazar una recta en el primer octante, entonces la pendiente es un valor entre 0 y 1, en tal caso, realmente en cada paso solo x se incrementa y respecto a y solo tenemos que decidir si y se incrementa o se queda igual.

Llamamos error e a la distancia entre la línea ideal que se desea trazar

y el pixel que queda mas cerca de dicha línea, si el pixel mas cercano queda por arriba de la línea el error será positivo y si el pixel mas cercano queda por debajo de la línea el error será negativo. El algoritmo de Bresenham hace uso solo de dicho signo para decidir cual pixel debe elegirse, si se elige no incrementar y, el error se acumula y si se incrementa y se le resta 1 al error, a continuación se muestra el algoritmo de Bresenham en su versión simple que usa flotantes

Input: x_{ini},y_{ini},x_{fin},y_{fin}
Output: Traza Línea recta
$x = x_{ini}$;
$y = y_{ini}$;
$\Delta x = x_{fin} - x_{ini}$;
$\Delta y = y_{fin} - y_{ini}$;
$e = \Delta y/\Delta x - 1/2$;
for $i = 1$ **to** Δx **do**
  Plot(floor(x),floor(y));
  while $e \geq 0$ **do**
    $y = y + 1$;
    $e = e - 1$;
  end
  $x = x + 1$;
  $e = e + \Delta y/\Delta x$;
end

Algorithm 3: Algoritmo Bresenham que usa flotantes

Para convertir este algoritmo en uno que solo use flotantes usamos $\hat{e} = 2e\Delta x$ de manera que en el Algoritmo 3, la línea:
$e = \Delta x/\Delta y - 1/2$
se convierte en:

$2e\Delta x = 2\Delta y - \Delta x$

es decir en la línea:

$\hat{e} = 2\Delta y - \Delta x$

Mientras que la línea:

$e = e - 1$

se convierte en:

$$2e\Delta x = 2e\Delta x - 2\Delta x$$

es decir:

$$\hat{e} = \hat{e} - 2\Delta x$$

Finalmente, la línea:

$$e = e + \Delta y/\Delta x$$

se convierte en:

$$2e\Delta x = 2e\Delta x + 2\Delta y$$

es decir:

$$\hat{e} = \hat{e} + 2\Delta y$$

El algoritmo de trazado de líneas rectas para el primer octante que solo utiliza aritmética entera queda:

Input: x_{ini},y_{ini},x_{fin},y_{fin}
Output: Traza Línea recta
$x = x_{ini}$;
$y = y_{ini}$;
$\Delta x = x_{fin} - x_{ini}$;
$\Delta y = y_{fin} - y_{ini}$;
$\hat{e} = 2\Delta y - \Delta x$;
for $i = 1$ **to** Δx **do**
  Plot(x,y);
  while $\hat{e} \geq 0$ **do**
    $y = y + 1$;
    $\hat{e} = \hat{e} - 2\Delta x$;
  end
  $x = x + 1$;
  $\hat{e} = \hat{e} + 2\Delta y$;
end

Algorithm 4: Algoritmo Bresenham para el primer octante, solo usa aritmética entera

El algoritmo 4 solo funciona en el primer octante.

2.2. Algoritmos de trazado de circunferencias

La ecuación de una circunferencia con radio R y centrada en (x_c, y_y) es:

$$(x - x_c)^2 + (y - y_c)^2 = R^2 \tag{2.9}$$

Si despejamos y de la ecuación anterior tendremos:

$$y = \sqrt{R^2 - (x - x_c)^2} + y_c \tag{2.10}$$

El método mas simple para trazar una circunferencia de radio R y centro en (x_c, y_c) consiste en asignar a x todos los valores enteros consecutivos desde $x_c - R$ hasta $x_c + R$ y por cada valor de x determinar el correspondiente valor de y mediante la ecuación (2.10), la cual por cierto regresa en realidad dos valores por lo que cada valor de x determina dos puntos de la circunferencia. El algoritmo sería el siguiente:

Input: x_c,y_c,R
Output: Traza Circunferencia de radio R y centro en (x_c, y_c)
$x = x_c - R$;
while $x \leq x_c + R$ **do**
  $y_1 = \sqrt{R^2 - (x - x_c)^2} + y_c$;
  $y_2 = -\sqrt{R^2 - (x - x_c)^2} + y_c$;
  Plot(floor(x),floor(y_1));
  Plot(floor(x),floor(y_2));
  $x = x + 1$;
end

Algorithm 5: Algoritmo simple para trazar un círculo de radio R centrado en (x_c, y_c)

Al utilizar el algoritmo 5, la circunferencia luce bien en la parte superior y en la inferior pero en la parte izquierda y derecha no se queda muy bien definida pues en esta parte se encienden menos pixels. Para remediar esto, se puede cambiar el rol de x y y en los octantes 1,4,5 y 8. Aún así podríamos observar que la brillantez de la circunferencia no es homogénea, para remediar esto se puede utilizar las ecuaciones paramétricas que definen la circunferencia:

$$\begin{aligned} x &= Rcos(\theta) + x_c \\ y &= Rsen(\theta) + y_c \end{aligned} \tag{2.11}$$

El procedimiento consiste entonces en darle un valor inicial a θ de cero e incrementarlo gradualmente hasta 2π, de esta manera generamos un circulo que luce homogéneo.

2.2.1. Algoritmo de Bresenham para trazado de circunferencias

Los dos procedimientos planteados en esta sección utilizan demasiadas operaciones, sobre todo de aritmética de punto flotante, cuando deseamos trazar cientos de círculos, comenzamos a pensar en que debemos encontrar una forma mas eficiente de hacerlo.

Estamos de acuerdo en que si somos capaces de trazar el círculo en el origen, moverlo de manera en que su centro quede ubicado en (x_c, y_c) implica simplemente sumar x_c a todas las coordenadas x y sumar y_c a todas

las coordenadas y, por lo tanto, podemos despreocuparnos por ahora de la posición de la circunferencia e implementar un algoritmo que solo requiera el parámetro R.

Si aprovechamos la simetría de un círculo ubicado en el origen, podemos simplemente determinar cuales pixels se deberían encender en un octante y en el resto de los octantes encender los que ocupan las posiciones simétricas correspondientes, así después de encender el pixel (x,y) procedemos a encender los pixels (x,-y),(-x,y),(-x,-y), (y,x),(y,-x),(-y,x) y (-y,-x).

El algoritmo de Bresenham se encarga de determinar los pixels que habría que encender en el segundo octante utilizando para ello solo aritmética entera, x comienza entonces con el valor de cero y y con el valor de R, se procede entonces incrementando x de uno en uno, en ese octante, y tiene solo dos posibilidades, se decrementa o mantiene su valor, al igual que el algoritmo de Bresenham para trazar líneas, utiliza solo el signo del error para decidir. El algoritmo del punto medio es una simplificación del algoritmo de Bresenham y lo describiremos a detalle a continuación

2.2.2. Algoritmo del punto medio para trazado de circunferencias

La función $f_{circ}(x, y)$ define si el punto de coordenadas (x,y) se encuentra dentro o fuera del círculo de radio R centrado en el origen, la definición formal sería:

$$f_{circ}(x,y) = x^2 + y^2 - R^2 = \begin{cases} < 0 & \text{(x,y) está dentro del círculo} \\ 0 & \text{(x,y) está justo en el círculo} \\ > 0 & \text{(x,y) está fuera del círculo} \end{cases} \tag{2.12}$$

Al trazar el segmento de círculo correspondiente al segundo octante en el sentido de las manecillas del reloj, x comienza con el valor cero y se incrementa en uno en cada paso de manera que:

$$x_{k+1} = x_k + 1 \tag{2.13}$$

La y por otro lado, comienza con el valor R y en cada paso se debe decidir si mantiene su valor, es decir si $y_{k+1} = y_k$ o si se debe decrementar su valor en uno, es decir si $y_{k+1} = y_k - 1$. Para decidirlo, evaluamos la función f_{circ}

en el punto medio, es decir, determinamos $p_k = f_{circ}(x_k + 1, y_k - 1/2)$, esto lo hacemos en cada paso. El signo de p_k nos indica si el punto medio entre $(x_k + 1, y_k)$ y $(x_k + 1, y_k - 1)$ queda dentro del círculo ($p_k < 0$) en cuyo caso debemos hacer $y_{k+1} = y_k$ ya que el pixel $(x_k + 1, y_k)$ queda mas cerca del circulo ideal que el pixel $(x_k + 1, y_k - 1)$. Si por el contrario, $p_k > 0$, entonces el punto medio queda fuera del circulo ideal y debemos elegir $(x_k + 1, y_k - 1)$. Si $p_k = 0$, entonces en realidad no importa cual de los dos pixels se escoja.

Para economizar cálculos podemos expresar p_{k+1} recursívamente, es decir, en términos de p_k:

$$p_{k+1} = (x_{k+1} + 1)^2 + (y_{k+1} - 1/2)^2 - R^2 \tag{2.14}$$

Como se explicó, si $p_k \leq 0$, entonces $y_{k+1} = y_k$, entonces:

$$\begin{aligned} p_{k+1} &= ((x_k + 1) + 1)^2 + (y_k - 1/2)^2 - R^2 \\ &= (x_k + 1)^2 + 2(x_k + 1) + 1 + (y_k - 1/2)^2 - R^2 \\ &= 2p_k + 2x_k + 3 = 2p_k + 2x_{k+1} + 1 \end{aligned} \tag{2.15}$$

si en cambio, $p_k > 0$, entonces $y_{k+1} = y_k - 1$, entonces:

$$\begin{aligned} p_{k+1} &= ((x_k + 1) + 1)^2 + ((y_k - 1) - 1/2)^2 - R^2 \\ &= (x_k + 1)^2 + 2(x_k + 1) + 1 + ((y_k - 1/2) - 1)^2 - R^2 \\ &= (x_k + 1)^2 + 2(x_k + 1) + 1 + (y_k - 1/2)^2 - 2(y_k - 1/2) + 1 - R^2 \\ &= [(x_k + 1)^2 + (y_k - 1/2)^2 - R^2] + 2x_{k+1} + 1 - 2y_k + 2 \\ &= p_k + 2x_{k+1} - 2y_{k+1} + 1 \end{aligned} \tag{2.16}$$

Al inicio $x_0 = 0$ y $y_0 = R$, entonces:

$$\begin{aligned} p_0 &= 1 + (R - 1/2)R^2 - R^2 \\ &= 1 + R^2 - R + 1/4 - R^2 \\ &= 5/4 - R \end{aligned} \tag{2.17}$$

Como queremos utilizar solo aritmética entera aproximamos $p_0 \approx 1 - R$

El algoritmo del punto medio para trazar un círculo de radio R en el origen es:

Input: x_c,y_c,R
Output: Traza Circunferencia de radio R centrada en (x_c, y_c
$x = 0$;
$y = R$;
p=1-R;
while $x \leq y$ **do**
  Plot($x_c + x, y_c + y$);
  Plot($x_c + x, y_c - y$);
  Plot($x_c - x, y_c + y$);
  Plot($x_c - x, y_c - y$);
  Plot($x_c + y, y_c + x$);
  Plot($x_c + y, y_c - x$);
  Plot($x_c - y, y_c + x$);
  Plot($x_c - y, y_c - x$);
  $x = x + 1$;
  if $p \leq 0$ **then**
    p=p+2x+1;
  else
    y=y-1;
    p=p+2x-2y+1;
  end
end

Algorithm 6: Algoritmo del punto medio para trazar un círculo de radio R centrado en (x_c, y_c)

2.3. Algoritmos de generación de elipses

La ecuación de una elipse con radios R_x y R_y, centrada en (x_c, y_y) es:

$$\frac{(x - x_c)^2}{R_x^2} + \frac{(y - y_c)^2}{R_y^2} = 1 \tag{2.18}$$

Esta ecuación se reduce a la de una circunferencia cuando $R_x = R_y$.
Si despejamos y de la ecuación anterior tendremos:

$$y = \sqrt{R_y^2 - \frac{R_y^2}{R_x^2}(x - x_c)^2} + y_c \tag{2.19}$$

El método mas simple para trazar una elipse de radios R_x y R_y centrada en (x_c, y_c) consiste en asignar a x todos los valores enteros consecutivos desde $x_c - R_x$ hasta $x_c + R_x$ y por cada valor de x determinar el correspondiente valor de y mediante la ecuación (2.19), la cual por cierto regresa en realidad dos valores por lo que cada valor de x determina dos puntos de la elipse. El algoritmo sería el siguiente:

Input: x_c,y_c,R_x,R_y
Output: Traza elipse de radios R_x y R_y con centro en (x_c, y_c)
$x = x_c - R_x$;
while $x \leq x_c + R_x$ **do**
  $y_1 = \sqrt{R_y^2 - \frac{R_y^2}{R_x^2}(x - x_c)^2} + y_c$;
  $y_2 = -\sqrt{R_y^2 - \frac{R_y^2}{R_x^2}(x - x_c)^2} + y_c$;
  Plot(floor(x),floor(y_1));
  Plot(floor(x),floor(y_2));
  $x = x + 1$;
end

Algorithm 7: Algoritmo simple para trazar una elipse de radios R_x y R_y centrada en (x_c, y_c)

Al utilizar el algoritmo 7, la elipse luce bien en la parte superior y en la inferior pero en la parte izquierda y derecha no se queda muy bien definida pues en esta parte se encienden menos pixels, para remediar esto se puede utilizar las ecuaciones paramétricas que definen la elipse:

$$\begin{aligned} x &= R_x cos(\theta) + x_c \\ y &= R_y sen(\theta) + y_c \end{aligned} \tag{2.20}$$

El procedimiento consiste entonces en darle un valor inicial a θ de cero e incrementarlo gradualmente hasta 2π, de esta manera generamos una elipse que luce homogénea.

2.3.1. Algoritmo de Bresenham para generación de elipses

Los dos procedimientos planteados en esta sección utilizan demasiadas operaciones, sobre todo de aritmética de punto flotante, cuando deseamos

trazar cientos de elipses, comenzamos a pensar en que debemos encontrar una forma mas eficiente de hacerlo.

Estamos de acuerdo en que si somos capaces de trazar la elipse en el origen, moverla de manera en que su centro quede ubicado en (x_c, y_c) implica simplemente sumar x_c a todas las coordenadas x y sumar y_c a todas las coordenadas y, por lo tanto, podemos despreocuparnos por ahora de la posición de la elipse e implementar un algoritmo que solo requiera los parámetro R_x y R_y.

Si aprovechamos la simetría de una elipse ubicada en el origen, podemos simplemente determinar cuales pixels se deberían encender en un cuadrante y en el resto de los cuadrantes encender los que ocupan las posiciones simétricas correspondientes, así después de encender el pixel (x,y) procedemos a encender los pixels (x,-y),(-x,y),(-x,-y).

El algoritmo de Bresenham se encarga de determinar los pixels que habría que encender en el primer cuadrante utilizando para ello solo aritmética entera, x comienza entonces con el valor de cero y y con el valor de R_y, se procede entonces incrementando x de uno en uno, desde que la pendiente de la elipse es de cero hasta que la pendiente es de -1, entonces se cambian los roles de x y y, al igual que el algoritmo de Bresenham para trazar circunferencias, utiliza solo el signo del error para decidir. El algoritmo del punto medio es una simplificación del algoritmo de Bresenham y lo describiremos a detalle a continuación

2.3.2. Algoritmo del punto medio para generación de elipses

La función $f_{elip}(x, y)$ define si el punto de coordenadas (x,y) se encuentra dentro o fuera de la ellipse de radios R_x y R_y centrado en el origen, la definición formal sería:

$$f_{elip}(x,y) = R_y^2 x^2 + R_x^2 y^2 - R_x^2 R_y^2 = \begin{cases} < 0 & \text{(x,y) está dentro de la elipse} \\ 0 & \text{(x,y) está justo en la elipse} \\ > 0 & \text{(x,y) está fuera de la elipse} \end{cases} \tag{2.21}$$

La derivada de la elipse en el punto x, y se puede obtener mediante el método de derivada total de una función implícita, es decir, para una función $f(x, y) = 0$ la derivada total es:

$$\frac{\partial f(x,y)}{\partial x}dx + \frac{\partial f(x,y)}{\partial y}dy = 0 \tag{2.22}$$

Aplicado a la ecuación de la elipse, obtenemos:

$$\frac{2x}{R_x^2}dx + \frac{2y}{R_y^2}dy = 0 \tag{2.23}$$

Despejando dy/dx obtenemos:

$$\frac{dy}{dx} = -\frac{R_y^2 x}{R_x^2 y} \tag{2.24}$$

Se comienza trazando el segmento de elipse que comienza en $x = 0$ y $y = R_y$ donde $dy/dx = 0$ y se incrementa x de uno en uno en cada paso de manera que:

$$x_{k+1} = x_k + 1 \tag{2.25}$$

En cada paso se debe decidir si y mantiene su valor, es decir si $y_{k+1} = y_k$ o si se debe decrementar su valor en uno, es decir si $y_{k+1} = y_k - 1$. Para decidirlo, evaluamos la función f_{elip} en el punto medio, es decir, determinamos $p1_k = f_{elip}(x_k + 1, y_k - 1/2)$, esto lo hacemos en cada paso. El signo de $p1_k$ nos indica si el punto medio entre $(x_k + 1, y_k)$ y $(x_k + 1, y_k - 1)$ queda dentro de la elipse ($p1_k < 0$) en cuyo caso debemos hacer $y_{k+1} = y_k$ ya que el pixel $(x_k + 1, y_k)$ queda mas cerca de la elipse ideal que el pixel $(x_k + 1, y_k - 1)$. Si por el contrario, $p1_k > 0$, entonces el punto medio queda fuera de la elipse ideal y debemos elegir $(x_k + 1, y_k - 1)$. Si $p1_k = 0$, entonces en realidad no importa cual de los dos pixels se escoja. En cada paso de este procedimiento se debe evaluar la derivada mediante la ecuación (2.24), cuando la derivada tenga un valor inferior a -1 debemos cambiar los roles que juegan x y y.

Para economizar cálculos podemos expresar $p1_{k+1}$ recursívamente, es decir, en términos de $p1_k$:

$$p1_{k+1} = (x_{k+1} + 1)^2 R_y^2 + (y_{k+1} - 1/2)^2 R_x^2 - R_x^2 R_y^2 \tag{2.26}$$

Como se explicó, si $p1_k \leq 0$, entonces $y_{k+1} = y_k$, entonces:

$$\begin{aligned} p1_{k+1} &= ((x_k+1)+1)^2R_y^2 + (y_k-1/2)^2R_x^2 - R_x^2R_y^2 \\ &= (x_k+1)^2R_y^2 + 2(x_k+1)R_y^2 + R_y^2 + (y_k-1/2)^2R_x^2 + R_x^2R_y^2 \\ &= p1_k + 2(x_k+1)R_y^2 + R_y^2 \\ &= p1_k + 2x_{k+1}R_y^2 + R_y^2 \end{aligned} \tag{2.27}$$

si en cambio, $p1_k > 0$, entonces $y_{k+1} = y_k - 1$, entonces:

$$\begin{aligned} p1_{k+1} &= ((x_k+1)+1)^2R_y^2 + ((y_k-1)-1/2)^2R_x^2 - R_x^2R_y^2 \\ &= (x_k+1)^2R_y^2 + 2(x_k+1)R_y^2 + R_y^2 + (y_k-1/2)^2R_x^2 - 2(y_k-1/2)R_x^2 + R_x^2 + R_x^2R_y^2 \\ &= p1_k + 2(x_k+1)R_y^2 + R_y^2 - 2(y_k-1/2)R_x^2 + R_x^2 \\ &= p1_k + 2x_{k+1}R_y^2 + R_y^2 - 2y_kR_x^2 + R_x^2 + R_x^2 \\ &= p1_k + 2x_{k+1}R_y^2 + R_y^2 - 2y_kR_x^2 + 2R_x^2 \\ &= p1_k + 2x_{k+1}R_y^2 + R_y^2 - 2R_x^2(y_k-1) \\ &= p1_k + 2x_{k+1}R_y^2 + R_y^2 - 2R_x^2y_{k+1} \end{aligned} \tag{2.28}$$

Al inicio $x_0 = 0$ y $y_0 = R_y$, entonces:

$$\begin{aligned} p1_0 &= R_y^2 + (R_y - 1/2)^2R_x^2 - R_x^2R_y^2 \\ &= R_y^2 + R_y^2R_x^2 - R_yR_x^2 + \frac{1}{4}R_x^2 - R_x^2R_y^2 \\ &= R_y^2 - R_yR_x^2 + \frac{1}{4}R_x^2 \end{aligned} \tag{2.29}$$

Cuando la pendiente de la elipse es inferior a -1, comenzamos a trazar la segunda parte de la elipse del primer cuadrante, en esta región,

y se decrementa de uno en uno en cada paso de manera que:

$$y_{k+1} = y_k - 1 \tag{2.30}$$

En cada paso se debe decidir si x mantiene su valor, es decir si $x_{k+1} = x_k$ o si se debe incrementar su valor en uno, es decir si $x_{k+1} = x_k + 1$. Para decidirlo, evaluamos la función f_{elip} en el punto medio, es decir, determinamos

$p2_k = f_{elip}(x_k + 1/2, y_k - 1)$, esto lo hacemos en cada paso. El signo de $p2_k$ nos indica si el punto medio entre (x_k, y_k) y $(x_k + 1, y_k - 1)$ queda dentro de la elipse ($p2_k < 0$) en cuyo caso debemos hacer $x_{k+1} = x_k + 1$ ya que el pixel (x_k+1, y_k+1) queda mas cerca de la elipse ideal que el pixel (x_k, y_k-1). Si por el contrario, $p2_k > 0$, entonces el punto medio queda fuera de la elipse ideal y debemos elegir $(x_k, y_k - 1)$. Si $p2_k = 0$, entonces en realidad no importa cual de los dos pixels se escoja.

Para economizar cálculos podemos expresar $p2_{k+1}$ recursívamente, es decir, en términos de $p2_k$:

$$p2_{k+1} = (x_{k+1} + 1/2)^2 R_y^2 + (y_{k+1} - 1)^2 R_x^2 - R_x^2 R_y^2 \tag{2.31}$$

Como se explicó, si $p2_k \leq 0$, entonces $x_{k+1} = x_k + 1$, entonces:

$$\begin{aligned} p2_{k+1} &= ((x_k + 1) + 1/2)^2 R_y^2 + (y_k - 1 - 1)^2 R_x^2 - R_x^2 R_y^2 \\ &= (x_k + 1/2)^2 R_y^2 + 2(x_k + 1/2) R_y^2 + R_y^2 + (y_k - 1)^2 R_x^2 - 2(y_k - 1) R_x^2 + R_x^2 - R_x^2 R_y^2 \\ &= p2_k + 2x_k R_y^2 + R_y^2 + R_y^2 - 2(y_k - 1) R_x^2 + R_x^2 \\ &= p2_k + 2x_k R_y^2 + 2R_y^2 - 2y_k R_x^2 + 2R_x^2 + R_x^2 \\ &= p2_k + 2x_{k+1} R_y^2 - 2y_{k+1} R_x^2 + R_x^2 \end{aligned} \tag{2.32}$$

si en cambio, $p2_k > 0$, entonces $x_{k+1} = x_k$, entonces:

$$\begin{aligned} p2_{k+1} &= (x_k + 1/2)^2 R_y^2 + (y_k - 1 - 1)^2 R_x^2 - R_x^2 R_y^2 \\ &= (x_k + 1/2)^2 R_y^2 + (y_k - 1)^2 R_x^2 - 2(y_k - 1) R_x^2 + R_x^2 - R_x^2 R_y^2 \\ &= p2_k - 2(y_k - 1) R_x^2 + R_x^2 \\ &= p2_k - 2y_k R_x^2 + 2R_x^2 + R_x^2 \\ &= p2_k - 2y_{k+1} R_x^2 + R_x^2 \end{aligned} \tag{2.33}$$

El algoritmo del punto medio para trazar una elipse de radios R_x y R_y en el origen es:

Input: x_c,y_c,R_x,R_y
Output: Traza elipse de radios R_x, R_y centrada en $(x_c, y_c$
$x = 0$;
$y = R_y$;
$p1 = R_y^2 - R_y R_x^2 + \frac{1}{4}R_x^2$;
while $2R_y^2 x \leq 2R_x^2 y$ **do**
  Plot$(x_c + x, y_c + y)$;
  Plot$(x_c + x, y_c - y)$;
  Plot$(x_c - x, y_c + y)$;
  Plot$(x_c - x, y_c - y)$;
  $x = x + 1$;
  if $p1 \leq 0$ **then**
    $p1 = p1 + 2xR_y^2 + R_y^2$;
  else
    y=y-1;
    $p1 = p1 + 2xR_y^2 + R_y^2 - 2R_x^2 y$;
  end
end
$p2 = R_y^2(x + 1/2)^2 + R_x^2(y - 1)^2 - R_x^2 R_y^2$;
while $y > 0$ **do**
  $y = y - 1$;
  if $p2 \leq 0$ **then**
    $x = x + 1$;
    $p2 = p2 + 2xR_y^2 - 2R_x^2 y + R_x^2$;
  else
    $p2 = p2 - 2yR_x^2 + R_x^2$;
  end
  Plot$(x_c + x, y_c + y)$;
  Plot$(x_c + x, y_c - y)$;
  Plot$(x_c - x, y_c + y)$;
  Plot$(x_c - x, y_c - y)$;
end

Algorithm 8: Algoritmo del punto medio para trazar una elipse de radios R_x,R_y centrada en (x_c, y_c)

2.4. Polilíneas

Una polilínea es una figura que como su nombre lo indica consta de varias líneas rectas, estas líneas están conectadas y ordenadas de manera que el final coincide con el inicio de la segunda, el final de la segunda coincide con el inicio de la tercera y así sucesivamente. Existen dos tipos de polilíneas, en las cerradas el final de la última línea coincide con el inicio de la primera formando así una figura cerrada que puede por ejemplo rellenarse. Las polilíneas abiertas no forman polígonos sino que forman simplemente líneas quebradas que pueden usarse para definición de extremos u otras aplicaciones.

2.5. Curvas Splines cúbicas naturales

Un spline solía ser un instrumento utilizado por dibujantes para hacer curvas suaves, consistía en una banda flexible que se hacía pasar por puntos específicos, actualmente una spline es una curva formada por segmentos polinomiales que satisfacen condiciones de continuidad entre ellos. Podemos clasificar a las splines como:

1. Splines de Interpolación. Aquellas que pasan justo por los puntos de control

2. Splines de Aproximación. Aquellas que pasan cerca de los puntos de control

En cualquier caso, los puntos de control definen un polígono donde el spline queda confinado.

Las condiciones de continuidad pueden ser:

1. Continuidad paramétrica. Si las derivadas paramétricas al final de cada segmento coincide exactamente con las derivadas paramétricas al inicio del siguiente segmento. Dependiendo de la mayor derivada en la que se exige esto, decimos que es continuidad C^0, C^1 o C^2, si se exige coincidencia con la derivada cero (Continuidad de orden cero), la primer derivada (Continuidad de primer orden) o la segunda derivada (Continuidad de segundo orden)

2. Continuidad geométrica. Si las derivadas de cada para de segmentos son proporcionales (no necesariamente iguales) en su frontera común. Les

denominamos G^0, G^1 o G^2 para referirnos a la continuidad geométrica de orden cero, primero y segundo respectivamente.

Los splines cúbicos naturales tienen continuidad C^2. Si tenemos n puntos de control, entonces tenemos n segmentos cúbicos y por tanto 4n coeficientes que determinar. En cada uno de los n-1 puntos de control interiores tenemos 4 condiciones de continuidad (una para hacer coincidir el final de un segmento con el punto de control, otra para que el inicio del siguiente segmento también pase por el punto de control, una mas para hacer coincidir la primera derivada y otra más para la segunda derivada). De ahí que obtengamos 4n-4 ecuaciones, obtenemos una ecuación más para que el primer segmento inicie en el primer punto de control y otra ecuación para que el último segmento termine en el último punto de control. Por lo tanto necesitamos dos ecuaciones más, una alternativa de solución consiste en especificar las derivadas de primer orden del primero y del último punto de control. Otra alternativa es agregar dos puntos de control ficticios adicionales (uno antes del primero y otro después del último)

El problema con las splines cúbicas naturales es que si cualquiera de los puntos de control es modificado, la curva completa (todos los segmentos) deben ser re-calculados, es decir, los splines naturales no permiten control local.

2.5.1. Splines de Hermite

Las Splines de Hermite permiten control local porque cada segmento de la curva solo depende de las restricciones de sus puntos extremos.

Si P(u) representa la curva paramétrica que inicia en el punto de control p_k y termina en p_{k+1}, entonces, las restricciones que definen la curva en ese segmento son:

$$\begin{aligned} P(0) &= p_k \\ P(1) &= p_{k+1} \\ P'(0) &= DP_k \\ P'(1) &= DP_{k+1} \end{aligned} \tag{2.34}$$

El polinomio paramétrico $P(u)$ es:

$$P(u) = au^3 + bu^2 + cu + d \tag{2.35}$$

O bien:

$$P(u) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ u \\ d \end{bmatrix} \tag{2.36}$$

$P'(u)$ es:

$$P(u) = 3au^2 + 2bu + c \tag{2.37}$$

O bien:

$$P(u) = \begin{bmatrix} 3u^2 & 2u & 1 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \\ u \\ d \end{bmatrix} \tag{2.38}$$

Es claro que $P(0) = d$ y que $P(1) = a + b + c + d$, además $P'(0) = c$ y $P'(1) = 3a+2b+c$, por lo tanto las restricciones para la curva en el segmento que va de p_k a p_{k+1} se pueden expresar como:

$$\begin{bmatrix} p_k \\ p_{k+1} \\ DP_k \\ DP_{k+1} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} \tag{2.39}$$

Resolviendo este sistema de ecuaciones obtenemos:

$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} p_k \\ p_{k+1} \\ DP_k \\ DP_{k+1} \end{bmatrix} \tag{2.40}$$

$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = M_H \begin{bmatrix} p_k \\ p_{k+1} \\ DP_k \\ DP_{k+1} \end{bmatrix} \tag{2.41}$$

M_H es la matriz de Hermite

$$P(u) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} p_k \\ p_{k+1} \\ DP_k \\ DP_{k+1} \end{bmatrix} \tag{2.42}$$

$$P(u) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \begin{bmatrix} 2p_k - 2p_{k+1} + DP_k + DP_{k+1} \\ 3p_k - 3p_{k+1} - 2DP_k - DP_{k+1} \\ DP_k \\ p_k \end{bmatrix} \tag{2.43}$$

$$P(u) = u^3(2p_k - 2p_{k+1} + DP_k + DP_{k+1}) + u^2(3p_k - 3p_{k+1} - 2DP_k - DP_{k+1}) + uDP_k + p_k \tag{2.44}$$

Reagrupando:

$$P(u) = p_k(2u^3 - 3u^2 + 1) + p_{k+1}(-2u^3 + 3u^2) + DP_k(u^3 - 2u^2 + u) + DP_{k+1}(u^3 - u^2) \tag{2.45}$$

$$P(u) = p_k H_0(u) + p_{k+1} H_1(u) + DP_k H_2(u) + DP_{k+1} H_3(u) \tag{2.46}$$

En la Figura 2.2 se presentan las funciones $H_0(u)$, $H_1(u)$, $H_2(u)$ y $H_3(u)$:

El problema con la utilización de las splines de Hermite es que estas requieren de la especificación de las derivadas en cada punto de control, para remediar este problema, las splines cardinales realizan una estimación de dichas derivadas mediante:

$$P'(0) = \frac{1}{2}(1 - t)(p_{k+1} - p_{k-1}) \tag{2.47}$$

$$P'(1) = \frac{1}{2}(1 - t)(p_{k+2} - p_k) \tag{2.48}$$

donde t es un parámetro que define la tensión de los segmentos de cada spline cardinal

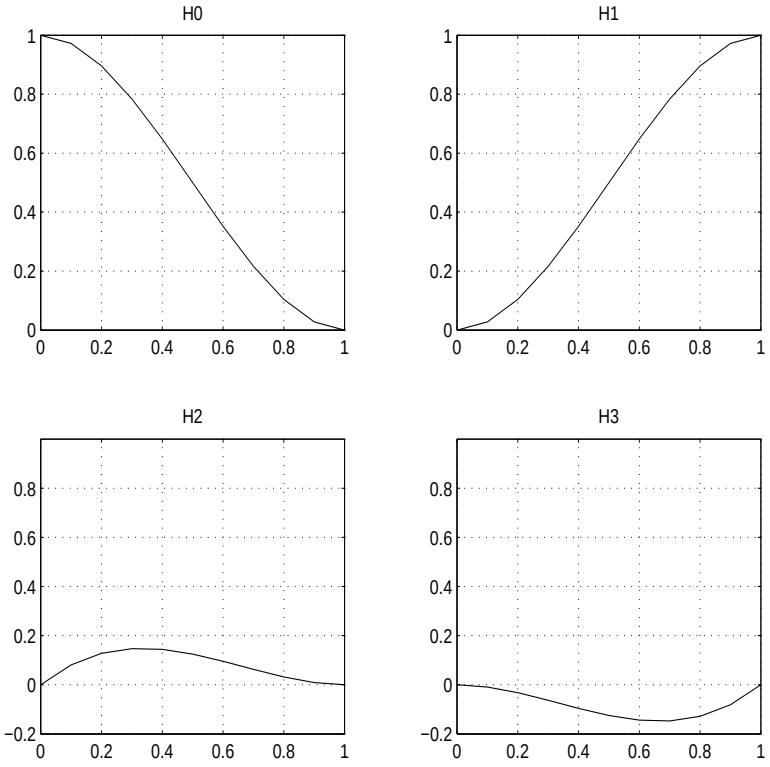

Figura 2.2: Funciones de Hermite

2.5.2. Curvas de Bezier

Estas splines de aproximación inventadas por el Ingeniero Pierre Bezier para el diseño de carrocerías Renault no requieren de la especificación de pendientes en los puntos de control ni de agregar puntos adicionales ni de ningún parámetro de tensión. Por su simplicidad son muy populares y aparecen en muchos paquetes de dibujo y de pintura.

El polinomio que describe la posición (en dos o en tres dimensiones) de cada punto que conforma una curva de Bezier para los n puntos de control que la definen es:

$$P(u) = \sum_{k=0}^{n} p_k B_{k,n}(u) \tag{2.49}$$

Donde $B_{k,n}(u)$ es la k-ésima función de combinación para $n+1$ puntos de control $(p_0, p_1, ..., p_n)$ y están definidas como:

$$B_{k,n}(u) = \binom{n}{k} u^k (1-u)^{n-k} \tag{2.50}$$

Por ejemplo, para 4 puntos de control, tendríamos las cuatro funciones de combinación siguientes:

$$\begin{aligned} B_{0,3}(u) &= (1-u)^3 \\ B_{1,3}(u) &= 3u(1-u)^2 \\ B_{2,3}(u) &= 3u^2(1-u) \\ B_{3,3}(u) &= u^3 \end{aligned} \tag{2.51}$$

Estas funciones se muestran en la Figura 2.3, en la esquina superior izquierda se muestra $B_{0,3}$, en la esquina superior derecha se muestra $B_{1,3}$, en la esquina inferior izquierda se muestra $B_{2,3}$ y en la esquina inferior derecha se muestra $B_{3,3}$.

2.5.3. Splines B

Las Splines B son posiblemente las splines mas utilizadas [2], al igual que las curvas de Bezier son splines de aproximación, presentan 2 ventajas respecto a las splines de Bezier:

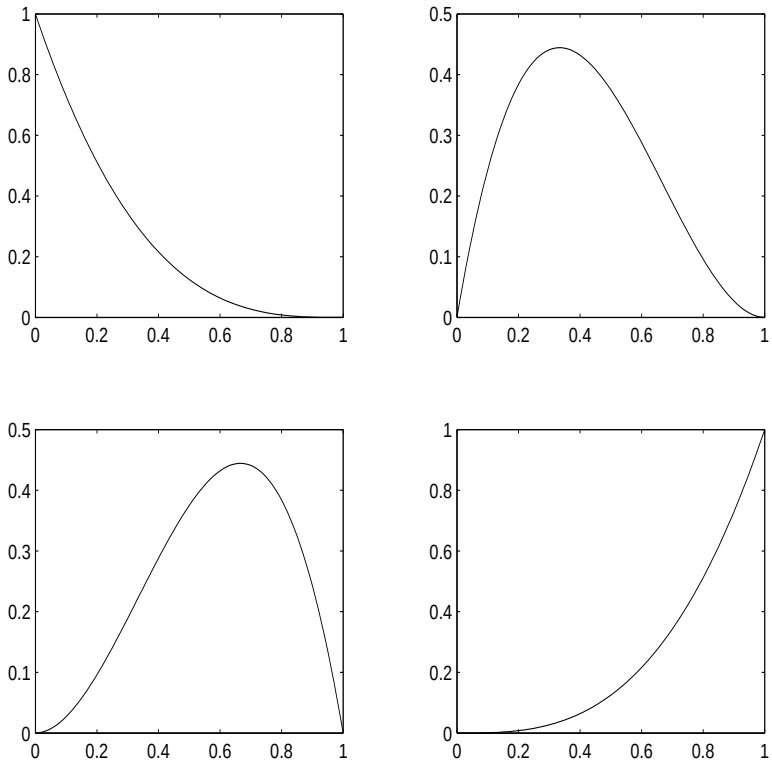

Figura 2.3: Funciones de Combinación de Bezier para 4 puntos de control

1. El grado del polinomio no tiene que aumentar al aumentar el número de puntos de control

2. Cada punto de control tiene un efecto mas local y por ende se tiene un mejor control acerca de la forma de la curva

A cambio, las splines B son mas complicadas que las splines de Bezier. Una spline B se define mediante la función:

$$P(u) = \sum_{k=0}^{n} p_k B_{k,d}(u) \qquad u_{min} \leq u \leq u_{max} \tag{2.52}$$

donde p_k es el k-ésimo punto de control del total de $n+1$ puntos de control. Las funciones de combinación $B_{k,d}$ son polinomios de grado $d-1$, el parámetro d puede tomar cualquier valor entre 2 y $n-1$, para $d=2$ la "curva" es en realidad una polilínea. A diferencia de las Splines de Bezier, el parámetro de control u no necesariamente varia de 0 a 1 y el control local se logra definiendo funciones de combinación sobre subíntervalos del intervalo total de variación de u. Las funciones de combinación se definen mediante las fórmulas recursivas de Cox-deBoor:

$$\begin{aligned} B_{k,1} &= \begin{cases} 1 & u_k \leq u \leq u_{k+1} \\ 0 & \text{de lo contrario} \end{cases} \\ B_{k,d} &= \frac{u - u_k}{u_{k+d-1} - u_k} B_{k,d-1} + \frac{u_{k+d} - u}{u_{k+d} - u_{k+1}} B_{k+1,d-1} \end{aligned} \tag{2.53}$$

Cada función de combinación está definida sobre d subintervalos y a cada valor extremo del intervalo se le denomina nudo, el conjunto de nudos se denomina vector de nudos. Se puede elegir cualquier conjunto de valores como nudos mientras estos estén en el intervalo de u_{min} a u_{max}, esta formulación requiere que en las fórmulas recursivas de Cox-deBoor cualquier evaluación de $0/0$ se le asigne el valor de 0.

Algunas propiedades de las Splines B son:

- para $n+1$ puntos de control tenemos $n+1$ funciones funciones de combinación

- Cada función de combinación de define sobre d subintervalos comenzando por el nodo de valor u_k

- el vector de nodos tiene $n+d+1$ elementos definiendo $n+d$ subintervalos
- Cada sección de la curva se ve influenciada por d puntos de control
- cada punto de control afecta a lo mas a d secciones de la curva

2.5.3.1. Splines B uniformes

Cuando el espaciado entre valores de nodos es uniforme (Ej {-1.5, -1.0, -0.5, 0.0, 0.5, 1.0, 1.5}), la curva resultante es una Spline B uniforme. El vector de nodos se puede normalizar para tener valores entre 0 y 1 (Ej. {0,0, 0,2, 0,4, 0,6, 0,8, 1,0}), sin embargo, por facilidad es común usar valor enteros consecutivos partiendo de 0 (Ej. $\{0, 1, 2, 3, 4, 5, 6, 7\}$).

Una ventaja de los Splines B uniformes es que las funciones de combinación son periódicas, esto significa que todas las funciones de combinación tienen la misma forma, cada una es simplemente una versión desplazada de la anterior:

$$B_{k,d}(u) = B_{k+1,d}(u + \Delta u) = B_{k+2,d}(u + 2\Delta u) \tag{2.54}$$

donde Δu es el intervalo entre nodos adyacentes

Para ejemplificar considere el caso en que $n = d = 3$, entonces el vector de nodos que define $n + d = 6$ subíntervalos debe tener $n + d + 1 = 7$ valores ($\{0, 1, 2, 3, 4, 5, 6\}$). Cada función de combinación abarca $d = 3$ subintervalos, utilizando la recurrencia de Cox-deBoor obtenemos:

$$\begin{aligned} B_{0,3}(u) &= \frac{u-0}{2-0}B_{0,2}(u) + \frac{3-u}{3-1}B_{1,2}(u) \\ &= \frac{u}{2}\left[\frac{u-0}{1-0}B_{0,1}(u) + \frac{2-u}{2-1}B_{1,1}(u)\right] + \frac{3-u}{2}\left[\frac{u-1}{2-1}B_{1,1}(u) + \frac{3-u}{3-2}B_{2,1}(u)\right] \\ &= \begin{cases} \frac{1}{2}u^2 & 0 \le u < 1 \\ \frac{1}{2}u(2-u) + \frac{1}{2}(3-u)(u-1) & 1 \le u < 2 \\ \frac{1}{2}(3-u)^2 & 2 \le u < 3 \end{cases} \end{aligned} \tag{2.55}$$

El resto de las funciones de combinación se obtienen aprovechando la periodicidad, ed decir:

$$B_{1,3}(u) = B_{0,3}(u-1) \tag{2.56}$$

$$= \begin{cases} \frac{1}{2}(u-1)^2 & 1 \leq u < 2 \\ \frac{1}{2}(u-1)(3-u) + \frac{1}{2}(4-u)(u-2) & 2 \leq u < 3 \\ \frac{1}{2}(4-u)^2 & 3 \leq u < 4 \end{cases} \tag{2.57}$$

$$B_{2,3}(u) = B_{1,3}(u-1) \tag{2.58}$$

$$= \begin{cases} \frac{1}{2}(u-2)^2 & 2 \leq u < 3 \\ \frac{1}{2}(u-2)(4-u) + \frac{1}{2}(5-u)(u-3) & 3 \leq u < 4 \\ \frac{1}{2}(5-u)^2 & 4 \leq u < 5 \end{cases} \tag{2.59}$$

$$B_{3,3}(u) = B_{2,3}(u-1) \tag{2.60}$$

$$= \begin{cases} \frac{1}{2}(u-3)^2 & 3 \leq u < 4 \\ \frac{1}{2}(u-3)(5-u) + \frac{1}{2}(6-u)(u-4) & 4 \leq u < 5 \\ \frac{1}{2}(6-u)^2 & 5 \leq u < 6 \end{cases} \tag{2.61}$$

Todas las funciones de combinación existen en el intervalo que está definido desde $u_{d-1} = 2$ hasta $u_{n+1} = 4$, solo en ese intervalo se cumple el requerimiento de que:

$$\sum_{k=0}^{n} B_{k,d}(u) = 1 \tag{2.62}$$

La curva inicia cuando $u = 2$ y termina cuando $u = 4$, en el primer segmento:

$$p_0 \left[\frac{1}{2}(3-u)^2\right] + p_1 \left[\frac{1}{2}(u-1)(3-u) + \frac{1}{2}(u-2)(4-u)\right] + p_2 \left[\frac{1}{2}(u-2)^2\right] \tag{2.63}$$

en el extremo inicial de la cual $u = 2$

$$P_{ini} = p_0 \left[\frac{1}{2}\right] + p_1 \left[\frac{1}{2}\right] = \frac{p_0 + p_1}{2} \tag{2.64}$$

mediante un análisis similar concluimos que

$$P_{fin} = \frac{p_2 + p_3}{2} \tag{2.65}$$

Con esto concluimos que el punto inicial de la Spline B uniforme esta justo enmedio de los puntos de control p_0 y p_1, mientras que el final de la spline esta a la mitad del penúltimo y el último punto de control (p_2 y p_3)

2.6. Estructura de un Programa OpenGL

Java OpenGL (JOGL) es una biblioteca que permite acceder a OpenGL mediante programación en Java. Actualmente está siendo desarrollado por el Game Technology Group de Sun Microsystems, y es la implementación de referencia para JSR-231 (Java Bindigs for OpenGL).

JOGL permite acceder a la mayoría de características disponibles para los programadores de C, con la excepción de las llamadas a ventanas realizadas en GLUT (ya que Java contiene sus propios sistemas de ventanas, AWT y Swing), y algunas extensiones de OpenGL.

Para trabajar con JoGL descargar

https://jogl.dev.java.net/files/documents/27/947/jogl-linux.tar.gz

Descomprimir y agregar jogl.jar al directorio lib/ext/ del SDK o JRE instalado, alternativamente, agregar jogl.jar a la variable CLASSPATH. Agregar el archivo jogl.dll al directorio bin/ del SDK o JRE instalado, alternativamente, copiarlo al directorio de trabajo.

El programa Java debe importar las clases del paquete JoGL incluidas en el archivo archivo jogl.jar, para ello use la directiva:

```
import net.java.games.jogl.*
```

Para hacer un programa en Java con salida gráfica es usual utilizar la clase java.awt.Frame.

```
public static void main(String[] args) {
    Frame frame = new Frame("Hello World");
```

Una vez creada la ventana/frame debemos agregarle un GLCanvas o un GLJPanel al frame para poder utilizar los servicios de OpenGL. Podemos utilizar el método factory para crear el GLCanvaso el GLJPanel, luego lo

agregamos al frame de esta forma podemos combinar las capacidades de OpenGL con las de la GUI de Java.

El método factory requiere que se le especifiquen las "capabilities" del canvas/panel, para nuestro propósito especificaremos las capacidad por defecto de GLCapability mediante la creación de un objeto nuevo de la clase GLCapability.

NOTA: GLCanvas desciende de la clase java.awt.Canvas y GLJPanel desciende de la clase javax.swing.JPanel y por lo tanto heredan sus métodos. Al presente solo la clase GLCanvas se beneficia de tarjetas de video con aceleradores gráficos, de manera que aunque la librería SWING sea mas popular, el utilizarla podría resultar en un desempeño reducido.

```
GLCanvas canvas = GLDrawableFactory.getFactory().createGLCanvas(new
GLCapabilities());

frame.add(canvas);
```

Ahora que el Frame tiene un GLCanvas, establecemos algunos parámetros adicionales para el frame incluyendo el tamaño de la ventana, el color del fondo y que hacer cuando el usuario decide cerrar la ventana.

```
frame.setSize(300, 300);
frame.setBackground(Color.WHITE);

frame.addWindowListener(new WindowAdapter()
{
    public void windowClosing(WindowEvent e)
    {
        System.exit(0);
    }
});
```

Finalmente, hay que indicarle al frame que se muestre y comience a aceptar las entradas del usuario:

```
frame.show();
```

El método reshape de la interfaz GLEventListener se llama cuando la ventana cambia de tamaño y se considera el lugar apropiado para establecer

el puerto de visión y la perspectiva, en el ejemplo se establece una perspectiva con un ángulo de visión de 45 grados, y un valor de cercanía y lejanía de 1 y 20 respectivamente

```
public void reshape(GLDrawable gLDrawable, int x, int y,
   int width, int height) {
   final GL gl = gLDrawable.getGL();
   final GLU glu = gLDrawable.getGLU();
   if (height <= 0) // avoid a divide by zero error!
     height = 1;
   final float h = (float)width / (float)height;
   gl.glViewport(0, 0, width, height);

   //Selecciona la Matriz de Proyeccion
   gl.glMatrixMode(GL.GL_PROJECTION);
   // Y borrala (Hazla la matriz identidad)
   gl.glLoadIdentity();

   glu.gluPerspective(45.0f, h, 1.0, 20.0);

   //Selecciona la Matriz de Modelado
   gl.glMatrixMode(GL.GL_MODELVIEW);
   // Y borrala (Hazla la matriz identidad)
   gl.glLoadIdentity();
}
```

El método init de GLEventListener es llamado inmediatamente después de que se crea el contexto de OpenGL, usualmente, aquí se ejecutan acciones que requieren hacerse una sola vez al inicio como por ejemplo ubicación de luces, en nuestro caso solo habilitamos el sombreado suave (GL_SMOOTH), establecemos el color negro como fondo y agregamos el KeyListener para que detecte cuando se pulsan teclas

```
public void init(GLDrawable gLDrawable) {
  final GL gl = gLDrawable.getGL();
  gl.glShadeModel(GL.GL_SMOOTH);
  // El cuarto parámetro (alfa) es la transparencia y se usa en iluminació
  gl.glClearColor(1.0f, 1.0f, 1.0f, 0.5f);
```

```
    gLDrawable.addKeyListener(this);
  }
```

El método display de GLEVentListener es el lugar indicado para poner el código relativo a los objetos que queremos mostrar en la ventana, en este caso solo limpiamos la pantalla

```
public void display(GLDrawable gLDrawable)  {
  final GL gl = gLDrawable.getGL();
  // Limpia la pantalla y el buffer de profundidad
  gl.glClear(GL.GL_COLOR_BUFFER_BIT | GL.GL_DEPTH_BUFFER_BIT);
  // Borra la Matriz de Modelado (ModelView)
  gl.glLoadIdentity();
}
```

2.7. Despliege de lineas, triángulos, cuadrados, circunferencias, etc mediante OpenGL

El siguiente código despliega 3 puntos en 2D

```
gl.glBegin (GL.GL_POINTS);
gl.glVertex2i (100,50);
gl.glVertex2i (100,130);
gl.glVertex2i (150,130);
gl.glEnd ();
```

El siguiente código despliega dos líneas paralelas en 2D

```
gl.glBegin (GL.GL_LINES);
gl.glVertex2i (50, 200);
gl.glVertex2i (75, 250);
gl.glVertex2i (60, 200);
gl.glVertex2i (85, 250);
gl.glEnd();
```

El siguiente código despliega una polilínea abierta

```
gl.glBegin (GL.GL_LINE_STRIP);
gl.glVertex2i (100, 200);
gl.glVertex2i (150, 250);
gl.glVertex2i (100, 250);
gl.glVertex2i (150, 200);
gl.glEnd();
```

El siguiente código despliega una polilínea cerrada

```
gl.glBegin (GL.GL_LINE_LOOP);
gl.glVertex2i (200, 200);
gl.glVertex2i (250, 250);
gl.glVertex2i (200, 250);
gl.glVertex2i (250, 200);
gl.glEnd();
```

El siguiente código despliega dos triánngulos

```
gl.glBegin (GL.GL_TRIANGLES);
gl.glVertex2i (400, 50);
gl.glVertex2i (400, 100);
gl.glVertex2i (420, 75);
gl.glVertex2i (425, 50);
gl.glVertex2i (425, 100);
gl.glVertex2i (445, 75);
gl.glEnd();

gl.glRecti (200, 50, 250, 150);
```

El siguiente código despliega una polígono, a diferencia de una polilínea cerrada, este tiene un color de relleno (por defecto)

```
gl.glBegin (GL.GL_POLYGON);
gl.glVertex2i (300, 50);
gl.glVertex2i (350, 60);
gl.glVertex2i (375, 100);
gl.glVertex2i (325, 115);
gl.glVertex2i (300, 75);
gl.glEnd();
```

Capítulo 3

Algoritmos de Relleno de áreas

Para rellenar un área, por ejemplo de un polígono, podríamos simplemente verificar para cada uno de los pixels si está dentro o fuera del área a rellenar, esta estrategia es demasiado costosa, podríamos sin embargo ahorrar muchos cálculos si restringimos los pixels de manera que se revisen solo aquellos pixels que estén dentro del rectángulo de menor tamaño que confine al polígono en cuestión, esta estrategia funciona mejor con unos polígonos que con otros, es decir, el ahorro depende de la forma del polígono.

3.1. Relleno mediante ordenamiento de aristas

Se puede aprovechar la propiedad conocida como "coherencia espacial" que nos dice que los pixels que están juntos muy probablemente tendrán las mismas características (ej. color), en particular, los pixels que pertenecen a la misma linea de rastreo tendrán "coherencia de linea de rastreo". Considere por ejemplo el polígono de la Figura 3.1, la línea de rastreo B es dividida en regiones por las intersecciones con las líneas que conforman el polígono, el primer segmento de la línea de rastreo queda fuera del polígono, mientras que el segmento que va de 4 a 5 está formado por pixels que deben ponerse en el color de relleno ya que quedan dentro del polígono, el segmento de 5 a 6 debe quedar en el color del fondo puesto que son pixels que quedan fuera del polígono, el segmento de 6 a 7 queda dentro y el resto del segmento (de 7 en adelante) queda fuera. Hay que tener en cuenta que los vértices son puntos que al intersectar a la correspondiente línea de rastreo pueden o no marcar

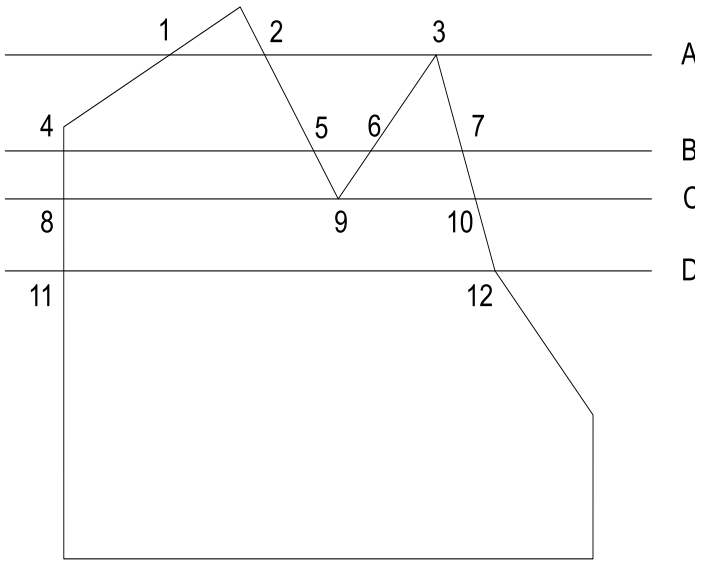

Figura 3.1: Polígono de ejemplo para rellenar por linea de rastreo

el inicio o fin de un segmento de pixels que pertenecen al área de relleno, por ejemplo, en la línea de rastreo A, el punto 3 no marca el inicio de un segmento de relleno; en la línea de rastreo C, el punto 9 tampoco marca el final del segmento de la línea que está dentro del área de relleno, en cambio, el la línea de rastreo D, el punto 12 (que también es un vértice) sí marca al final de un segmento de la línea que queda dentro del área de relleno. En este proceso, las líneas horizontales deben ser ignoradas. Tomando en cuenta estas observaciones, el algoritmo de relleno por ordenamiento de aristas q recibe la lista de vertices P_1, P_2,...,P_n que definen el polígono a rellenar y define cuales pixels deben ponerse al color del relleno:

1. $i = 1$
2. Mientras $i < n$

   Si la linea que va del vértice P_i al vértice P_{i+1} no es horizontal,

entonces utilizando el algoritmo de Bresenham determina los pixels (x, y) que conforman la línea que va del vértice P_i al vértice P_{i+1}

3. Si la línea que va del vértice P_n al vértice P_1 no es horizontal entonces utilizando el algoritmo de Bresenham determina los pixels (x, y) que conforman la línea que va del vértice P_n al vértice P_1

4. Ordenar la colección completa de pixels primero por y (línea de rastreo) y luego por x

5. Como resultado del ordenamiento anterior se detectan duplicados, estos corresponden con vértices. Conservar estos duplicados si el vértice corresponde con un máximo o con un mínimo, eliminar uno de los dos duplicados en caso contrario.

6. Por cada línea de rastreo de la lista ordenada

   $j = 1$

   Mientras hay mas elementos en la lista ordenada de la actual línea de rastreo, tomar dos elementos de la lista x_j y x_{j+1} y poner los pixels en la actual línea de rastreo que van de x_j a x_{j+1} en el color de relleno

   $j = j + 2$

El paso 4 del algoritmo de relleno por ordenamiento de aristas es el mas costoso, es precisamente en donde se ordenan todos los pixels que forman el contorno del polígono. Podemos ahorrarnos el paso 4 si al ir generando los pixels por donde pasan las líneas que van de cada vértice al siguiente se insertan en una lista ordenada en lugar de posponer el ordenamiento para cuando esta lista esté completa.

3.2. Relleno mediante complementación

Este algoritmo hace uso del hecho de que al realizar la operación XOR entre un valor y el valor actual de un pixel, se obtienen los efectos descritos en la Tabla 3.1, observe en esta tabla que encender un pixel que ya está encendido es equivalente a apagarlo. Este hecho es claro cuando trabajamos con dibujos en blanco y negro, es decir, sabemos que dibujar una linea encima de otra idéntica a la que se esta trazando utilizando la operación XOR es equivalente a borrarla. Un comportamiento similar ocurre cuando se trabaja con

líneas de cualquier color y el color del fondo, es decir, si se dibuja una línea azul sobre un fondo rojo utilizando la operación XOR la línea que se obtiene es en realidad verde, pero al volver a dibujar la línea azul sobre dicha línea verde recién trazada (de nuevo con la operación xor) se obtiene una línea roja igual que el fondo, por lo tanto la línea ya no se ve, ¡se ha borrado!. Por lo tanto el efecto de que dibujar dos veces una misma línea con la operación xor es equivalente a no dibujar nada es válido para dibujos a color y no solo en blanco y negro.

Tabla 3.1: Operación XOR

valor	pixel	resultado
0	0	0
0	1	1
1	0	1
1	1	0

A diferencia del algoritmo de relleno por ordenamiento de aristas, el algoritmo de relleno mediante complementación no necesita realizar ningún ordenamiento ni mantener ninguna lista ordenada, el algoritmo consiste en:

1. Por cada arista no horizontal del polígono

   descubrir las coordenadas (x, y) por las que pasa la arista mediante el algoritmo de Bresenham

   Por cada x

   Realizar la operación XOR entre el color de relleno y el color existente en el dibujo desde (x, y) hacia la derecha hasta (x, y_{max}) donde y_{max} es el número de columnas (pixels horizontales) de la ventana de dibujo

En las figuras 3.2 y 3.3 se muestra un ejemplo donde se rellenó un polígono tomando las aristas en el orden d,b,e,c,a. Las aristas horizontales no participan en el proceso (arista f en el ejemplo). El orden en que se tomen las aristas es irrelevante, el algoritmo de relleno de polígonos mediante complementación funciona, se deja al lector como ejercicio probar otro orden de aristas

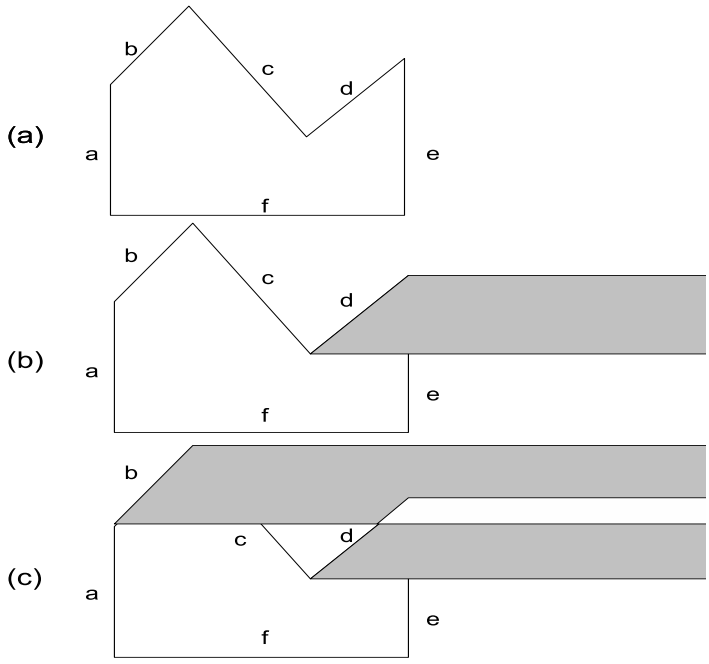

Figura 3.2: Relleno mediante complementación

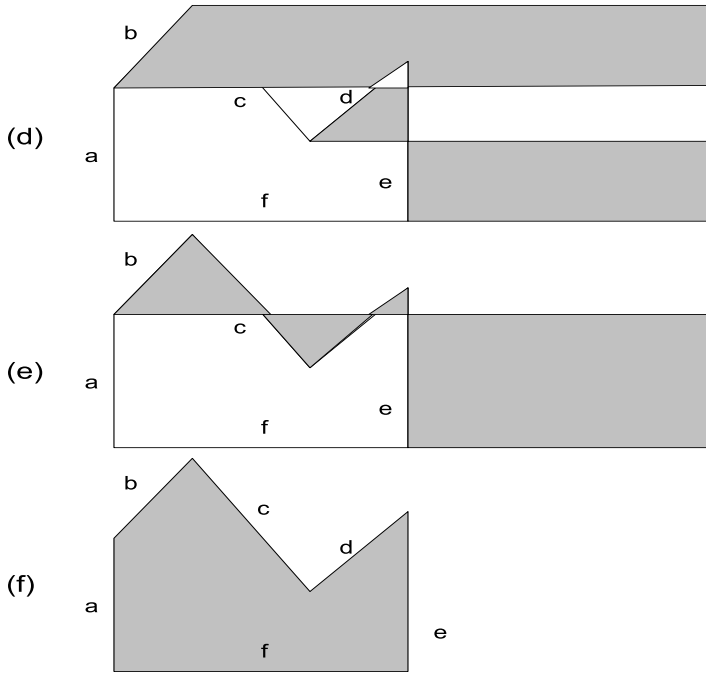

Figura 3.3: Relleno mediante complementación (continuación)

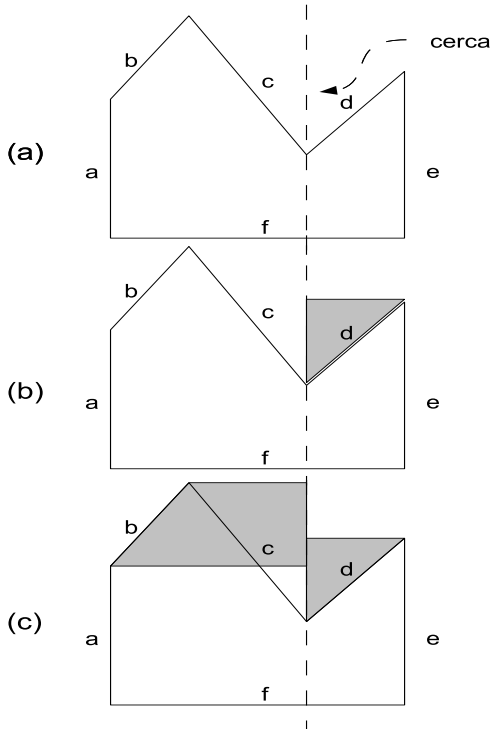

Figura 3.4: Relleno mediante complementación usando una cerca

3.2.1. Modificación mediante el uso de una cerca

El algoritmo de relleno mediante complementación no requiere dedicar tiempo para ordenar o mantener ordenada ninguna lista de pixels que conforman aristas, pero en cambio dedica demasiado tiempo a cambiar el color de demasiados pixels, algunos de ellos varias veces, este tiempo se puede reducir drásticamente si se emplea una cerca, la cerca es una línea vertical infinita imaginaria que pasa por cualquier vértice del polígono, entonces, en lugar de modificar los pixels a la derecha cada arista se modifican los pixels que están entre la arista y la cerca solamente, en la Figura 3.4 y 3.5 se muestra un ejemplo de este procedimiento utilizando la misma secuencia de aristas (d,b,c,a).

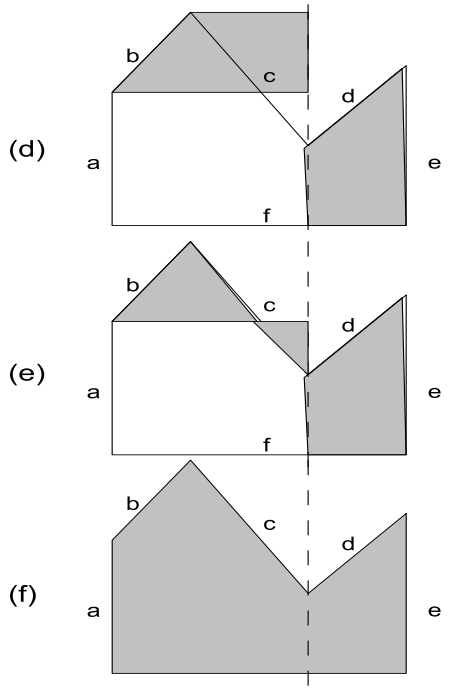

Figura 3.5: Relleno mediante complementación usando una cerca(continuación)

3.3. Algoritmo simple de siembra de semilla

Este algoritmo recibe un pixel inicial interior al área que se quiere rellenar, llamamos a este pixel semilla, el algoritmo pone este pixel en el color del relleno y enseguida verifica los pixels que están alrededor de este, para cada pixel circundante aquellos que no estén en el color del relleno o en el color del borde los convierte en semillas nuevas. Para cada semilla nueva se repite el proceso. Este algoritmo se puede implementar en forma iterativa haciendo uso de una pila o bien en forma recursiva. El Algoritmo 9 muestra la versión iterativa, la versión recursiva se muestra a continuación:

```
rellena(int x,int y) {
    setPixel(x,y,colorRelleno);
    color=getPixel(x+1,y);
    if ((color!=colorRelleno)&&(color!=colorBorde)) rellena(x+1,y);
    color=getPixel(x-1,y);
    if ((color!=colorRelleno)&&(color!=colorBorde)) rellena(x-1,y);
    color=getPixel(x,y+1);
    if ((color!=colorRelleno)&&(color!=colorBorde)) rellena(x,y+1);
    color=getPixel(x,y-1);
    if ((color!=colorRelleno)&&(color!=colorBorde)) rellena(x,y-1);
}
```

Input: coordenadas de la semilla x,y, colorRelleno, colorBorde
Output: Figura rellena
push(x,y)
while *pop(x,y)* **do**
  setPixel(x,y,colorRelleno)
  if $color(x-1,y) \notin \{colorRelleno, colorBorde\}$ **then**
    push(x-1,y)
  end
  if $color(x+1,y) \notin \{colorRelleno, colorBorde\}$ **then**
    push(x+1,y)
  end
  if $color(x,y-1) \notin \{colorRelleno, colorBorde\}$ **then**
    push(x,y-1)
  end
  if $color(x,y+1) \notin \{colorRelleno, colorBorde\}$ **then**
    push(x,y+1)
  end
end

Algorithm 9: Algoritmo de semilla para rellenar áreas

Tanto el algoritmo 9 como su versión recursiva mostrada antes utilizan conectividad 4, es decir, cada pixel se considera conectado únicamente a los 4 pixels que están al Norte, Sur, Este y Oeste del mismo, es fácil modificarlos para que tengan conectividad 8, es decir que se considere también a los pixels que están al Sureste, Noreste, Noroeste y Suroeste, esta decisión debe ser congruente con la forma de dibujar las figuras.

```
Input: coordenadas de la semilla x,y, colorRelleno, colorBorde
Output: Figura rellena
push(x,y)
while pop(x,y) do
    colorearArriba=false; colorearAbajo=false; xSalva=x;
    while color(x, y) ≠ colorBorde do
        setPixel(x,y,colorRelleno)
        if not colorearArriba then
            if color(x, y − 1) ∉ {colorRelleno, colorBorde} then
                push(x,y-1); colorearArriba=true;
            end
        end
        if not colorearAbajo then
            if color(x, y + 1) ∉ {colorRelleno, colorBorde} then
                push(x,y+1); colorearAbajo=true;
            end
        end
        x=x+1;
    end
    x=xSalva-1;
    while color(x, y) ≠ colorBorde do
        setPixel(x,y,colorRelleno)
        if not colorearArriba then
            if color(x, y − 1) ∉ {colorRelleno, colorBorde} then
                push(x,y-1); colorearArriba=true;
            end
        end
        if not colorearAbajo then
            if color(x, y + 1) ∉ {colorRelleno, colorBorde} then
                push(x,y+1); colorearAbajo=true;
            end
        end
        x=x-1;
    end
end
```

Algorithm 10: Algoritmo de semilla por línea de rastreo

3.4. Siembra de semilla por línea de rastreo

Al aplicar el algoritmo de semilla tal y como se explicó en la sección anterior para rellenar un área grande, muy probablemente se tendrán problemas de memoria, esto es debido a que al sacar un elemento de la pila, se pueden introducir cuatro (u ocho si se optó por esa conectividad). El algoritmo de siembra de semilla por línea de rastreo soluciona este problema, al insertar un máximo de 2 elementos a la pila por cada línea de pixels que pone en el color de relleno, el algoritmo saca la semilla de la pila y rellena la línea de rastreo a la que corresponde la semilla hacia la derecha y hacia la izquierda y al hacerlo checa los pixels que están en la línea de rastreo inmediata superior e inmediata inferior, si encuentra al menos un pixel de la línea inmediata superior que no esté en el color del relleno mete ese pixel a la pila pero solo uno de toda la línea, hace lo mismo con la línea inmediata inferior. El método se muestra con detalle en el Algoritmo 10

3.5. Funciones de OpenGL para manejo del color de las figuras y del fondo

Al ejecutar la función:

void glColor3f(GLfloat red, GLfloat green, GLfloat blue)

especificamos el color que OpenGL utilizará al trazar cualquier cosa en ahí en adelante. Los tres parámetros que recibe deberán tener un valor entre 0.0 y 1.0 y define el porcentaje de rojo, de verde y de azul respectivamente. Si además deseamos especificar el parámetro alfa, utilizamos la función:

void glColor4f(GLfloat red, GLfloat green, GLfloat blue, GLfloat alpha)

El valor alfa se utiliza para mezclar colores usando su valor como porcentaje de dicho color. Utilizando mezclado de colores se pueden lograr ciertos efectos como dibujar figuras semitransparentes.

La función:

void glClearColor3f(GLfloat red, GLfloat green, GLfloat blue)

Nos permite especificar el color del fondo

Capítulo 4

Algoritmos de Recorte

En muchas aplicaciones se debe seleccionar una área de recorte que define lo que será visible por ejemplo para hacer un acercamiento a una sección de interés

4.1. Códigos de región para determinar la visibilidad de líneas

Una manera rápida para decidir cuales líneas son totalmente visibles, parcialmente visibles o trivialmente invisibles es utilizar los códigos de Cohen, este método funciona para areas de recorte rectangulares. La Figura 4.1 define los códigos de Cohen, el primer bit (el menos significativo) indica que el vértice está a la izquierda del área de recorte, el segundo bit indica que el punto este se encuentra a la derecha, el tercero y cuarto bit indican que el punto está por arriba y por debajo del área de recorte respectivamente.

Una línea es totalmente visible si los códigos de Cohen de ambos extremos son 0000. Si al hacer la operación AND entre los códigos de Cohen de los extremos de una línea se obtiene un resultado distinto de 0000, entonces la línea es trivialmente invisible puesto que ambos extremos se encuentran a la izquierda o ambos están arriba, etc. Ahora bien, si la operación AND entre los dos extremos de una línea resulta en 0000, esto no significa que la línea sea totalmente visible sino que puede ser parcialmente visible o incluso totalmente invisible. Por supuesto, si uno de los extremos tiene un Código de Cohen de 0000 y el otro extremo tiene un código diferente, entonces la línea es parcialmente visible. En la Figura 4.2 se muestra un conjunto de

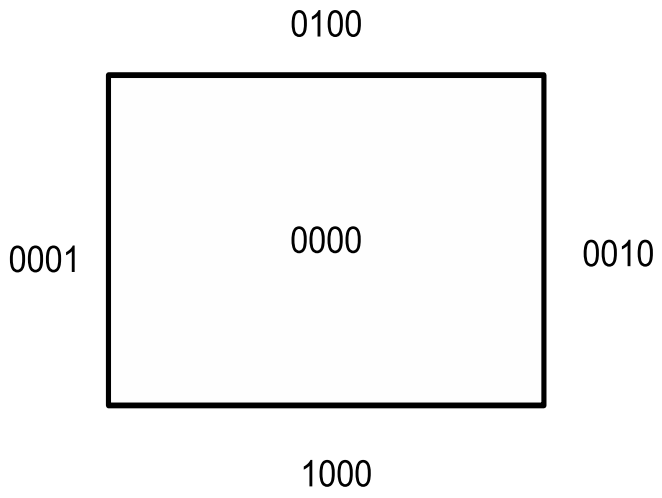

Figura 4.1: Códigos de Cohen para determinar visibilidad

líneas que en base al área de recorte. La Tabla 4.1 muestra los códigos de Cohen correspondientes a las líneas de la Figura 4.2. Observe que el método no puede resolver el caso de las líneas ab ni il.

Tabla 4.1: Códigos de Cohen de las líneas de la Figura 4.2

línea	Código 1	Código 2	AND	Observación
ab	0001	1000	0000	Parcialmente visible
cd	0000	0000	0000	Totalmente visible
ef	0100	0000	0000	Parcialmente visible
gh	0010	0010	0010	Trivialmente invisible
il	1000	0010	0000	Totalmente invisible

4.2. Algoritmo de recorte explícito en 2D

Para líneas que no son "trivialmente invisibles" o totalmente visibles, es necesario encontrar su intersección con las líneas que conforman el área rect-

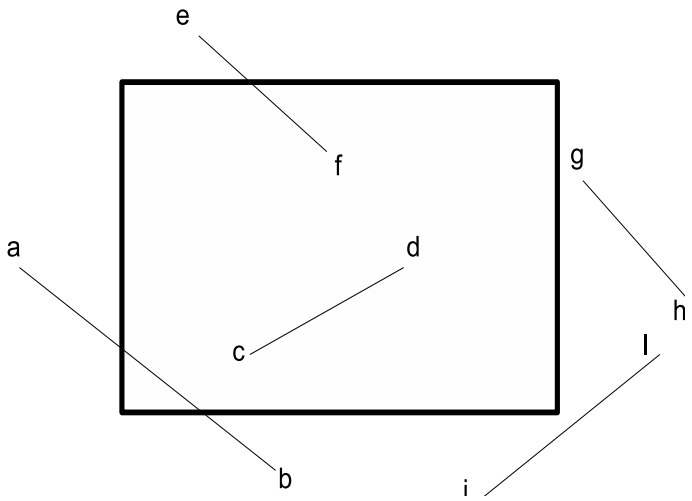

Figura 4.2: Líneas a recortar y área de recorte

angular de recorte, las cuales son:

$y = y_{sup}$ extremo superior
$y = y_{inf}$ extremo inferior
$x = x_{izq}$ extremo izquierda
$x = x_{der}$ extremo derecho

La ecuación de una línea recta que pasa por los puntos (x_1, y_1) y (x_2, y_2) es:

$$y = mx + b \tag{4.1}$$

donde $m = \frac{y_2 - y_1}{x_2 - x_1}$
Como la línea pasa por (x_1, y_1), entonces:

$$y_1 = mx_1 + b \tag{4.2}$$

Por lo que:

$$b = y_1 - mx_1 \tag{4.3}$$

Por tanto:

$$y = mx + y_1 - mx_1 \tag{4.4}$$

Agrupando:

$$y = m(x - x_1) + y_1 \tag{4.5}$$

De manera similar, despejando x de (4.1)

$$x = \frac{y}{m} - \frac{b}{m} \tag{4.6}$$

Sustituyendo (4.3) en la ecuación anterior obtenemos:

$$x = \frac{y}{m} - \frac{y_1 - mx_1}{m} \tag{4.7}$$

De donde:

$$x = \frac{y - y_1}{m} + x_1 \tag{4.8}$$

Por lo tanto, Utilizando (4.5) obtenemos la intersección de la recta con con el extremo derecho ocurre en:

$$(x_{der}, \frac{y_2 - y_1}{x_2 - x_1}(x_{der} - x_1) + y_1) \tag{4.9}$$

La intersección con el extremo izquierdo ocurre en:

$$(x_{izq}, \frac{y_2 - y_1}{x_2 - x_1}(x_{izq} - x_1) + y_1) \tag{4.10}$$

Utilizando (4.8) obtenemos la intersección de la recta con con el extremo superior ocurre en:

$$((y_{sup} - y_1)\frac{x_2 - x_1}{y_2 - y_1} + x_1, y_{sup}) \tag{4.11}$$

La intersección con el extremo inferior ocurre en:

$$((y_{inf} - y_1)\frac{x_2 - x_1}{y_2 - y_1} + x_1, y_{inf}) \tag{4.12}$$

Por supuesto, hay que descartar intersecciones que no ocurran en la recta sino en alguna prolongación de la recta

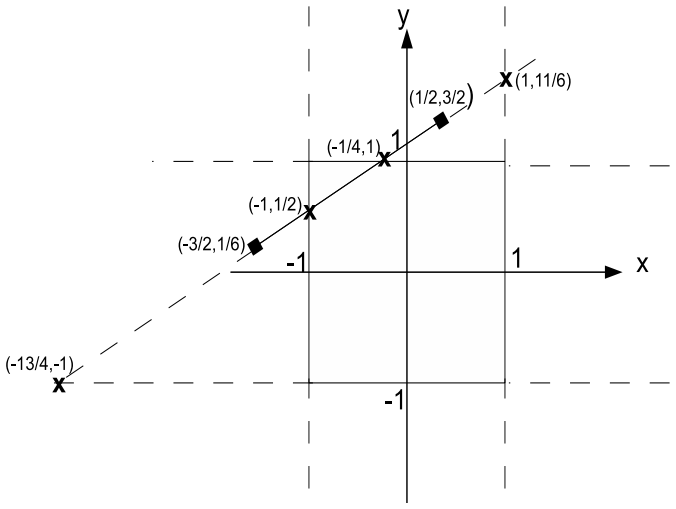

Figura 4.3: Recorte Explícito

4.2.1. Ejemplo

Recorte la línea que va de (-3/2,1/6) a (1/2,3/2) de acuerdo al área de recorte de la Figura 4.3

La intersección con el extremo derecho ocurre en:

$$\Big(1, \frac{3/2 - 1/6}{1/2 - (-3/2)}(1 - (-3/2)) + 1/6\Big) = \Big(1, (\frac{2}{3})(\frac{5}{2}) + \frac{1}{6}\Big) = \Big(1, \frac{11}{6}\Big)$$

la cual se descarta porque ocurre mas a la derecha del extremo derecho de la línea ($11/6 > 1$). La intersección con el extremo izquierdo ocurre en:

$$\Big(-1, \frac{3/2 - 1/6}{1/2 - (-3/2)}(-1 - (-3/2)) + 1/6\Big) = \Big(-1, (\frac{2}{3})(\frac{1}{2}) + \frac{1}{6}\Big) = \Big(-1, \frac{1}{2}\Big)$$

La intersección con el extremo superior ocurre en:

$$\Big((1 - 1/6)\frac{1/2 - (-3/2)}{3/2 - 1/6} + (-3/2), 1\Big) = \Big((\frac{5}{6})(\frac{3}{2}) - \frac{3}{2}, 1\Big) = \Big(-\frac{1}{4}, 1\Big)$$

Finalmente, la intersección con el extremo inferior ocurre en:

$$\Big((-1-1/6)\frac{1/2-(-3/2)}{3/2-1/6}+(-3/2),-1\Big)=\Big((-\frac{7}{6})(\frac{3}{2})-\frac{3}{2},-1\Big)=\Big(-\frac{13}{4},-1\Big)$$

El cual también se descarta porque ocurre mas a la izquierda del principio de la línea ($-13/4 < -3/2$). Los puntos de intersección no descartados nos indican que la línea recortada va de (-1,1/2) a (-1/4,1).

4.3. Algoritmo de Sutherland-Cohen

- Por cada extremo de la ventana rectangular de recorte efectuar los pasos (a), (b) y (c)

  - (a) Para la línea P_1P_2, determine si la línea es totalmente visible o si puede ser descartada como trivialmente invisible

  - (b) Si P_1 está fuera de la ventana de recorte continúa, de lo contrario intercambia P_1 y P_2

  - (c) Reemplaza P_1 por la intersección de P_1P_2 con el extremo en turno

4.4. Algoritmo de la subdivisión del punto medio

Encontrar la intersección de una línea con un área de recorte rectangular puede hacerse por bisecciones, es decir, dividiendo la línea a la mitad y analizando ambas partes, una de ellas se descarta ya sea por ser trivialmente invisible o por ser completamente visible. La búsqueda de la intersección continua en la otra mitad de la línea. Este método tiene la enorme ventaja de poderse implementar muy fácilmente en hardware e incluso en caso de implementarse en software tiene la ventaja de usar solo aritmética entera dado que la operación de dividir entre dos es en realidad un corrimiento hacia la derecha. El Algoritmo 11 es una versión recursiva de este método

Recorta(P_1,P_2)
if P_1P_2 *es una línea trivialmente invisible* **then**
  Descarta el segmento de línea (P_1,P_2);
  Termina;
end
if P_1P_2 *es una línea totalmente visible* **then**
  Incluye como visible el segmento de línea (P_1,P_2);
  Termina;
end
$P_m = (P_1 + P_2)/2$
Recorta(P_1P_m);
Recorta(P_mP_2);

Algorithm 11: Algoritmo de subdivisión del punto medio

4.5. Algoritmo de Cyrus-Beck para recorte de regiones convexas

El Algoritmo de Cyrus-Beck hace uso del hecho de que un punto a está dentro de un área de recorte convexa respecto a cierta línea que define un borde si se cumple la desigualdad:

$$n \cdot (b - a) > 0 \tag{4.13}$$

donde n es un vector normal a la línea que define el borde y b es cualquier punto en ese borde

Por supuesto, a y b son a la vez puntos y vectores que van del origen a la ubicación de los mismos. En realidad la Ecuación (4.13) nos dice que si a está dentro del área de recorte, entonces el vector que va de b hacia a (es decir, el vector $b - a$) tiene un ángulo interno menor de 90 grados respecto al vector normal n. Por otro lado, el punto a se encuentra fuera del área de recorte si se cumple:

$$n \cdot (b - a) < 0 \tag{4.14}$$

Finalmente, el punto a se encuentra justo en el borde si se cumple:

$$n \cdot (b - a) = 0 \tag{4.15}$$

El Algoritmo de Cyrus-Beck hace uso de la ecuación paramétrica de una línea recta que va de P_1 a P_2 que es:

$$P(t) = P_1 + (P_2 - P_1)t \tag{4.16}$$

donde t es el parámetro que vale 0 al principio de la línea (En P_1) y 1 al final de la misma (En P_2)

Aplicando (4.15) podemos obtener el valor de t para el cual la línea coincide con la frontera del área de recorte despejándolo de:

$$n \cdot (P(t) - f) = 0 \tag{4.17}$$

donde f es cualquier punto de la frontera en cuestión

Lo que la ecuación (4.17) nos dice es que un vector que corre a lo largo de la frontera $(P(t) - f)$ tiene exactamente 90 grados respecto a la normal a la línea que define dicha frontera (n). Despejando t de (4.17) y luego sustituyéndola en $P(t)$ obtenemos las coordenadas donde la línea es cortada por la frontera en cuestión, el proceso se debe repetir para cada una de las fronteras (líneas borde) que definen el área de recorte convexa.

Sustituyendo (4.16 en (4.17) obtenemos:

$$n \cdot (P_1 + (P_2 - P_1)t - f) = 0 \tag{4.18}$$

Rearreglando;

$$n \cdot (P_1 - f) + n \cdot (P_2 - P_1)t = 0 \tag{4.19}$$

O bien:

$$n \cdot w + (n \cdot D)t = 0 \tag{4.20}$$

donde $D = P_2 - P_1$ es la directriz ya que define la dirección de la línea $P(t)$ y $w = P_1 - f$

Despejando t de (4.20):

$$t = -\frac{n \cdot w}{n \cdot D} \tag{4.21}$$

Si $n \cdot D > 0$, entonces el valor de t corresponde a un lugar cercano al inicio de la línea, si por el contrario $n \cdot D < 0$, el valor de t corresponde a un lugar cercano al final de la línea.

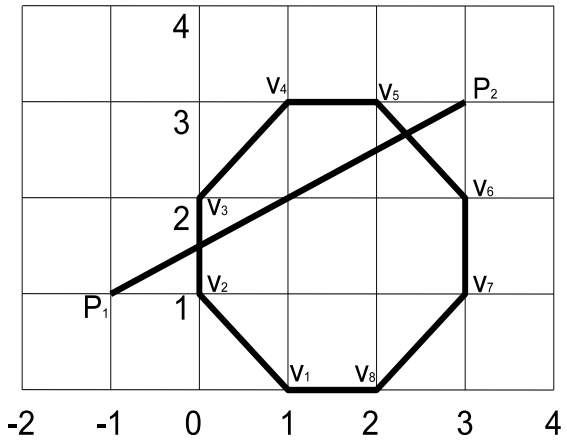

Figura 4.4: Recorte en un área convexa

Cuando una línea se recorta en un área definida por un polígono convexo, las intersecciones de la línea en cuestión con las diferentes aristas del polígono se pueden agrupar en dos conjuntos, a saber, las intersecciones que están cerca del inicio de la línea y las que están cerca del final de la línea, de cada grupo se elige solo una, necesitamos a la intersección mas alejada del principio de la línea de entre aquellas clasificadas como cerca del inicio. Del otro extremo, debemos elegir a la intersección mas alejada del final de la línea del grupo de intersecciones consideradas como cercanas al final de esta.

4.5.1. Ejemplo

Recortar la línea que va del punto (-1,1) al punto (3,3) de acuerdo al área de recorte mostrada en la Figura 4.4.

Solución:

La directriz D es:

$$D = P_2 - P_1 = \begin{bmatrix} 3 \\ 3 \end{bmatrix} - \begin{bmatrix} -1 \\ 1 \end{bmatrix} = \begin{bmatrix} 4 \\ 2 \end{bmatrix} \tag{4.22}$$

Para la arista V_5V_6, la normal que apunta hacia adentro del área de recorte es:

$$n = \begin{bmatrix} -1 \\ -1 \end{bmatrix} \tag{4.23}$$

Entonces:

$$n \cdot D = \begin{bmatrix} -1 \\ -1 \end{bmatrix} \cdot \begin{bmatrix} 4 \\ 2 \end{bmatrix} = -6 \leq 0 \tag{4.24}$$

Lo cual significa que la intersección de la línea con la arista V_5V_6 está cerca del final de la línea.

tomando a $f = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$ como un punto que forma parte de la arista V_5V_6 determinamos w:

$$w = P_1 - f = \begin{bmatrix} -1 \\ 1 \end{bmatrix} - \begin{bmatrix} 2 \\ 3 \end{bmatrix} = \begin{bmatrix} -3 \\ -2 \end{bmatrix} \tag{4.25}$$

De ahí que:

$$n \cdot w = \begin{bmatrix} -1 \\ -1 \end{bmatrix} \cdot \begin{bmatrix} -3 \\ -2 \end{bmatrix} = 5 \tag{4.26}$$

Finalmente:

$$t = -\frac{n \cdot w}{n \cdot D} = -\frac{5}{-6} = \frac{5}{6} \tag{4.27}$$

Lo cual significa que esta arista intersecta a la línea en:

$$P(\frac{5}{6}) = P_1 + (P_2 - P_1)\frac{5}{6} = \begin{bmatrix} -1 \\ 1 \end{bmatrix} + \left(\begin{bmatrix} 3 \\ 3 \end{bmatrix} - \begin{bmatrix} -1 \\ 1 \end{bmatrix}\right)\frac{5}{6} = \begin{bmatrix} 7/3 \\ 8/3 \end{bmatrix} \tag{4.28}$$

El proceso se repite para cada una de las aristas, el resultado se resume en la Tabla 4.2. En la columnas etiquetadas t_i y t_f se encuentran los valores de t correspondientes a las intersecciones de la línea P_1P_2 con cada arista, cuando el valor $n \cdot D$ es negativo el valor t se ubica en la columna t_f para recordar que es una intersección cercana al final de la línea, si en cambio $n \cdot D$ es una cantidad positiva entonces el valor t se ubica en la columna t_i dado que se trata de una intersección cercana al inicio de la línea. Por supuesto,

los valores de t inferiores a cero o superiores a uno no corresponden a lugares entre el principio y el final de línea sino a prolongaciones de esta y pueden ser descartados, de cualquier manera solo se elige realmente a un solo valor de la columna t_i, aquel que es el mayor de todos, en nuestro ejemplo $t = 1/4$. De la misma manera, solo elegimos a un solo valor de la columna t_f, es decir al menor de todos, en nuestro ejemplo $t = 5/6$.

Tabla 4.2: Recorte de la línea P_1P_2 mostrada en la Figura 4.4 en el área convexa detallada en la misma figura

Arista	n	f	w	$n \cdot w$	$n \cdot D$	t_i	t_f
V_1V_2	$\begin{bmatrix} 1 \\ 1 \end{bmatrix}$	$\begin{bmatrix} 1 \\ 0 \end{bmatrix}$	$\begin{bmatrix} -2 \\ 1 \end{bmatrix}$	-1	6	$\frac{1}{6}$	
V_2V_3	$\begin{bmatrix} 1 \\ 0 \end{bmatrix}$	$\begin{bmatrix} 0 \\ 2 \end{bmatrix}$	$\begin{bmatrix} -1 \\ -1 \end{bmatrix}$	-1	4	$\frac{1}{4}$	
V_3V_4	$\begin{bmatrix} 1 \\ -1 \end{bmatrix}$	$\begin{bmatrix} 0 \\ 2 \end{bmatrix}$	$\begin{bmatrix} -1 \\ -1 \end{bmatrix}$	0	2	0	
V_4V_5	$\begin{bmatrix} 0 \\ -1 \end{bmatrix}$	$\begin{bmatrix} 2 \\ 3 \end{bmatrix}$	$\begin{bmatrix} -3 \\ -2 \end{bmatrix}$	2	-2		1
V_5V_6	$\begin{bmatrix} -1 \\ -1 \end{bmatrix}$	$\begin{bmatrix} 2 \\ 3 \end{bmatrix}$	$\begin{bmatrix} -3 \\ -2 \end{bmatrix}$	5	-6		$\frac{5}{6}$
V_6V_7	$\begin{bmatrix} -1 \\ 0 \end{bmatrix}$	$\begin{bmatrix} 3 \\ 1 \end{bmatrix}$	$\begin{bmatrix} -4 \\ 0 \end{bmatrix}$	4	-4		1
V_7V_8	$\begin{bmatrix} -1 \\ 1 \end{bmatrix}$	$\begin{bmatrix} 3 \\ 1 \end{bmatrix}$	$\begin{bmatrix} -4 \\ 0 \end{bmatrix}$	4	-2		2
V_8V_1	$\begin{bmatrix} 0 \\ 1 \end{bmatrix}$	$\begin{bmatrix} 1 \\ 0 \end{bmatrix}$	$\begin{bmatrix} -2 \\ 1 \end{bmatrix}$	1	2	$-\frac{1}{2}$	

Para terminar el ejemplo solo resta decir que la línea recortada comienza en $t = 1/4$ y termina en $t = 5/6$, lo cual corresponde a una nueva línea que va de (0,3/2) a (7/3,8/3).

Capítulo 5

Pipeline de visualización bidimensional

Un paquete gráfico debe permitir a un usuario especificar cual parte de una escena se quiere desplegar y en que lugar en el dispositivo de salida, esto implica aplicar transformaciones de coordenadas mundiales a coordenadas de dispositivo que pueden incluir rotaciones, escalamientos y recortes de las partes de la escena que hayan quedado fuera del área seleccionada para ser desplegada.

5.1. Coordenadas locales, coordenadas mundiales, puerto de visión

Las figuras aisladas están normalmente definidas en coordenadas de modelado (MC) las cuales son relativas a alguna parte de la misma figura, por ejemplo el centro de la misma, así los vértices de un cuadrado pudieran ser: {(1,1),(1,-1),(-1,1),(-1,-1)}. Al establecer una escena, las diferentes figuras que la componen son convertidas a coordenadas mundiales (WC), entonces los vértices o puntos de control tiene valores relativos a un lugar específico de la escena misma.

Un área de la escena seleccionada para ser desplegada se denomina "ventana". Un área del dispositivo de salida donde la ventana será mapeada se denomina "puerto de visión". La ventana especifica lo que será desplegado mientras que el puerto de visión especifica donde será desplegado. El mapeo de una parte de la escena en coordenadas mundiales a coordenadas de dis-

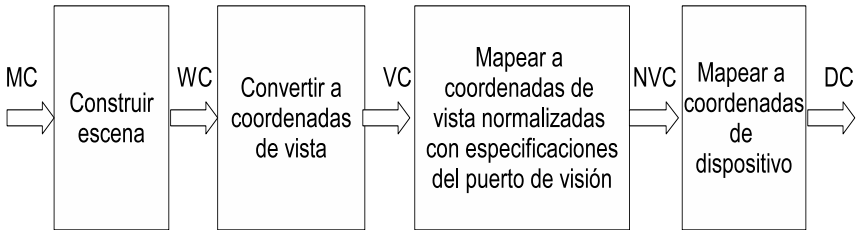

Figura 5.1: Pipeline de visualización bidimensional

positivo implica normalmente mas de una transformación. Opcionalmente podemos convertir a coordenadas de vista (VC) como un método para establecer orientaciones arbitrarias de la ventana, enseguida se pueden definir coordenadas normalizadas del puerto de visión (en el rango de cero a uno) y mapear las coordenadas de vista a coordenadas de vista normalizadas (NVC). En el paso final, todas las coordenadas contenidas en el puerto de visión son traducidas a coordenadas de dispositivo (DC). La Figura 5.1 resume este proceso.

Cambiando la posición del puerto de visión podemos ver objetos en diferentes lugares del dispositivo de salida, al cambiar el tamaño del puerto de visión podemos lograr efectos de magnificación como viendo algo a traves de una lupa. En cambio, si hacemos la ventana mas pequeña tendremos efectos de acercamiento (zoom in) a alguna parte específica de la escena, por supuesto, al hacer la ventana mas grande haríamos un alejamiento (zoom out). Se pueden lograr efectos de "paneo" moviendo la ventana por diferentes lugares de la escena y manteniendo su tamaño fijo.

Manejar coordenadas de vista normalizadas de cero a uno es de utilidad para separar las transformaciones de vista y otras de las que son específicas del dispositivo,de esta manera el paquete de graficación es independiente del dispositivo de salida, el resultado queda en un cuadrado unitario que será fácilmente mapeado a las coordenadas de cualquier dispositivo de salida.

Normalmente todas las transformaciones son preparadas para aplicarse via una sola matriz de transformación para ahorrar cálculos, el recorte de las líneas que quedan fuera del puerto de visión se hace entonces como último

paso, esta es una de las más importantes aplicaciones de los algoritmos de recorte.

5.2. Funciones de OpenGL para visualización bidimensional

El método reshape de la interfaz GLEventListener se activa ante cualquier cambio del tamaño de la ventana de despliegue (no confundir con la ventana que toma una parte de la escena), tambien se activa la primera vez que se lanza la aplicación. El método reshape recibe el tamaño de la ventana de despliegue en los parámetros width y height, estos parámetros son justamente los que necesitamos para establecer el tamaño del puerto de visión que es exactamente lo que hace la línea:

```
gl.glViewport( 0, 0, width, height );
```

Cuando se pretende graficar solo en dos dimensiones se puede hacer uso de una función de GLU (Graphics Library Utilities) denominada gluOrtho2D donde se define el rango de valores de coordenadas horizontales y verticales que se pretende manejar, así por ejemplo para indicar que en el eje horizontal los valores posibles para dibujar irán desde cero hasta 450 mientras que en el eje vertical estos podrán variar desde cero hasta 375 usaríamos la línea:

```
glu.gluOrtho2D( 0.0, 450.0, 0.0, 375.0);
```

Como en este caso no deseamos hacer ninguna operación de proyección (proyectar figuras 3D en algún plano) se incluyen las líneas:

```
gl.glMatrixMode( GL.GL_PROJECTION );
gl.glLoadIdentity();
```

La función completa reshape de este ejemplo queda:

```
public void reshape (GLDrawable drawable,int x,int y,int width,int height) {
    GL gl = drawable.getGL();
    GLU glu = drawable.getGLU();
    gl.glViewport( 0, 0, width, height );
```

```
        gl.glMatrixMode( GL.GL_PROJECTION );
        gl.glLoadIdentity();
        glu.gluOrtho2D( 0.0, 450.0, 0.0, 375.0);
    }
```

Capítulo 6

Transformaciones geométricas

Las transformaciones geométricas sirven para modificar la forma y ubicación de los objetos, en la Figura 6.1 un punto es modificado en su posición.

Sin embargo, una transformación geométrica puede verse también como aquella que nos lleva de un sistema de coordenadas a otro, en la Figura 6.2 la misma transformación de la Figura 6.1 se interpreta ahora como la que cambia un punto de un sistema de coordenadas $x - y$ a un sistema de coordenadas $u - v$ y vicebersa.

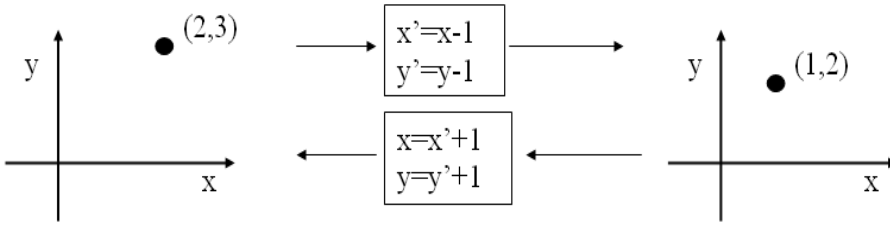

Figura 6.1: Primera interpretación de una transformación

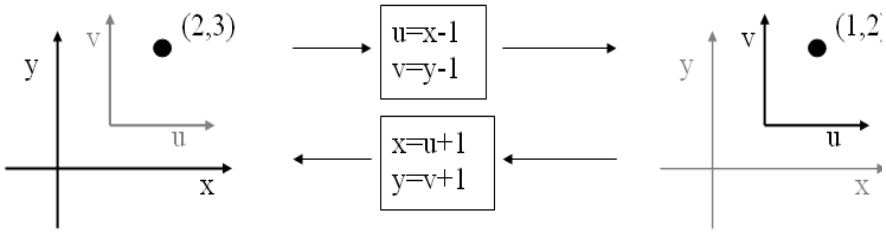

Figura 6.2: Segunda interpretación de una transformación

6.1. Transformaciones afines

Una transformación afín es aquella que se puede escribir de la forma:

$$\begin{aligned} x' &= a_{xx}x + a_{xy}y + b_x \\ y' &= a_{yx}x + a_{yy}y + b_y \end{aligned} \tag{6.1}$$

donde a_{xx}, a_{xy}, b_x, a_{yx}, a_{yy} y b_y son constantes

Una transformación afín es una operación lineal, esto implica que si se aplica la transformación a todos los puntos que forman una línea se obtiene otra línea que también se puede obtener aplicando la transformación solo a los extremos de la línea y se generan los puntos intermedios mediante algún algoritmo de trazado de líneas como el de Bresenham, como corolario podríamos decir que si se transforma un punto que está a la mitad de la línea, este punto de ubicaría justamente a la mitad de la línea transformada.

Ejemplos de transformaciones afines son la traslación, la rotación, el escalamiento, la reflexión y la inclinación, cualquier composición de estas operaciones también sería una transformación afín.

Si la transformación afín solo hace uso de rotaciones, traslaciones y reflexiones entonces las líneas paralelas al transformarlas continúan siendo paralelas, se conservan los ángulos entre cualquier pareja de líneas.

6.2. Transformaciones geométricas bidimensionales básicas

6.2.1. Traslación

Trasladamos un punto en la posición (x,y) a la posición (x',y') mediante la operación:

$$\begin{aligned} x' &= x + t_x \\ y' &= y + t_y \end{aligned} \tag{6.2}$$

En forma vectorial

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix} \tag{6.3}$$

que en forma compacta puede representarse como:

$$p' = p + t \tag{6.4}$$

donde t es un vector de traslación

6.2.2. Escalamiento

Escalamos mediante la operación:

$$\begin{aligned} x' &= S_x x \\ y' &= S_y y \end{aligned} \tag{6.5}$$

En forma matricial

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} S_x & 0 \\ 0 & S_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{6.6}$$

que en forma compacta puede representarse como:

$$p' = Sp \tag{6.7}$$

donde S es la matriz de escalamiento.

6.2.3. Rotación

Para deducir la fórmula para rotar un punto alrededor del origen observemos la Figura 6.3, es claro que:

$$\begin{aligned} x' &= r\ cos(\theta + \phi) \\ y' &= r\ sen(\theta + \phi) \end{aligned} \tag{6.8}$$

recordando las identidades trigonométricas:

$$\begin{aligned} cos(\alpha + \beta) &= cos(\alpha)cos(\beta) - sen(\alpha)sen(\beta) \\ sen(\alpha + \beta) &= cos(\alpha)sen(\beta) + sen(\alpha)cos(\beta) \end{aligned} \tag{6.9}$$

podemos convertir (6.9) en:

$$x' = r\ cos(\theta)cos(\phi) - r\ sen(\theta)sen(\phi) \tag{6.10}$$
$$y' = r\ cos(\theta)sen(\phi) + r\ sen(\theta)cos(\phi)$$

De la Figura 6.3 podemos observar también que:

$$x = r\ cos(\phi) \tag{6.11}$$
$$y = r\ sen(\phi)$$

Sustituyendo (6.12) en (6.11) obtenemos:

$$x' = x\ cos(\theta) - y\ sen(\theta) \tag{6.12}$$
$$y' = y\ cos(\theta) + x\ sen(\theta)$$

que en forma matricial es:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} cos(\theta) & -sen(\theta) \\ sen(\theta) & cos(\theta) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{6.13}$$

y en forma compacta:

$$p' = Rp \tag{6.14}$$

donde R es la matriz de rotación, la cual por cierto es ortogonal, lo que significa que su inversa es igual a su transpuesta, por lo tanto una rotación en sentido inverso se puede obtener multiplicando por R^t o bien cambiando θ por $-\theta$. Es fácil comprobar que R es ortogonal verificando que $R^tR = I$, también porque el producto interno de las dos columnas o de los dos renglones es igual a cero. R no solo es ortogonal sino que también es ortonormal ya que la norma de cualquier vector columna o renglón es igual a uno ya que $cos^2\theta + sen^2\theta = 1$.

6.3. Coordenadas Homogéneas

Hemos establecido que una traslación es una suma vectorial ($p' = p + t$), mientras que un escalamiento y una rotación es un producto de una matriz

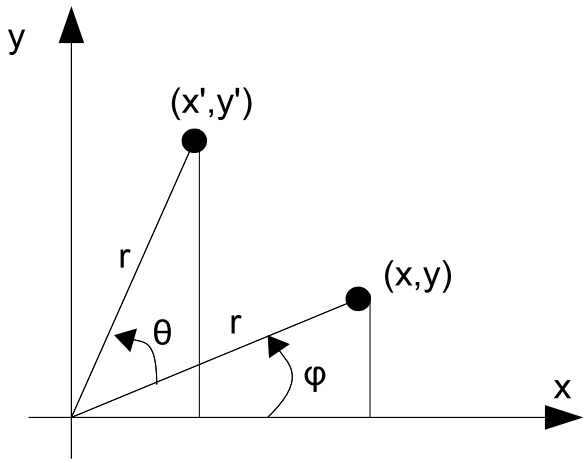

Figura 6.3: Rotación de un punto alrededor del origen

por un vector ($p' = Sp$ y $p' = Rp$ respectivamente) podríamos generalizar las transformaciones de manera que siempre consistieran en un producto matricial y una suma vectorial (lo cual no sería eficiente), es decir, la traslación sería $p' = Ip + t$, la rotación sería $p' = Rp + 0$ y el escalamiento sería $p' = Sp + 0$, aún así no habríamos resuelto el problema de que las transformaciones no son fáciles de componer. Las transformaciones compuestas son las que consisten de varias trasformaciones sucesivas, por ejemplo, una rotación seguida de un escalamiento y luego una rotación. Las coordenadas homogéneas solucionan el problema de las transformaciones sucesivas de una manera eficiente al definir todas las transformaciones como un producto matricial. Las coordenadas (x,y) se representan mediante una tercia ordenada (x_h, y_h, h) donde:

$$\begin{aligned} x &= \frac{x_h}{h} \\ y &= \frac{y_h}{h} \end{aligned} \tag{6.15}$$

Una coordenada específica (x,y) tiene un número infinito de representaciones en coordenadas homogéneas (hx, hy, h), la alternativa mas simple

($h = 1$) es también la mas usada, así, excepto en algunas aplicaciones como en la transformación tridimensional a coordenadas de vista, las coordenadas (x,y) las representamos simplemente como (x,y,1) en coordenadas homogéneas. Ahora podemos expresar la operación de traslación como en (6.16), la operación de escalamiento como en (6.17) y la de rotación como en (6.18), es decir, todas de la misma manera, una multiplicación matricial por una matriz de 3 por 3 genérica donde los elementos de la diagonal están relacionados con el escalamiento, los elementos de la tercer columna están relacionados con la traslación y la rotación afecta a la submatriz de 2 por 2 superior izquierda.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{6.16}$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{6.17}$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} cos(\theta) & -sen(\theta) & 0 \\ sen(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{6.18}$$

6.4. Transformaciones compuestas

Ya hemos visto que las coordenadas homogéneas eficientizan la aplicación de transformaciones sucesivas al permitir que todas las transformaciones consistan de multiplicaciones de matrices cuadradas, veamos ahora en acción esta idea con dos ejemplos, el escalamiento respecto a un punto fijo y la rotación respecto a un punto arbitrario

6.5. Escalamiento respecto a un punto fijo

El escalamiento simple lleva implícito una traslación posiblemente involuntaria como se puede observar en la Figura 6.4 donde a cada vértice del triángulo de líneas continuas se le ha aplicado un factor de escala $S_x = S_y = 2$ para obtener el correspondiente vértice del triángulo escalado que se muestra en líneas punteadas.

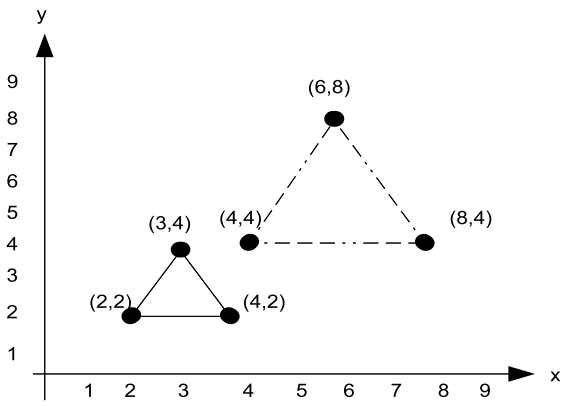

Figura 6.4: Escalamiento Simple

Es posible realizar el escalamiento respecto a un punto fijo (x_f, y_f), para ello necesitamos realizar tres transformaciones, primero debemos realizar una traslación de manera tal que el punto designado como fijo coincida con el origen, enseguida se hace el escalamiento simple. El punto que esta en el origen naturalmente no resulta modificado en este paso. Finalmente se realiza una traslación de forma que el punto fijo regrese a su lugar. El escalamiento respecto a un punto fijo se logra entonces mediante (6.19):

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_f \\ 0 & 1 & y_f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_f \\ 0 & 1 & -y_f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{6.19}$$

La ventaja de trabajar en coordenadas homogéneas se puede apreciar ahora, podemos realizar el producto matricial para obtener (6.20) y luego 6.21:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_f \\ 0 & 1 & y_f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} S_x & 0 & -S_x x_f \\ 0 & S_y & -S_y y_f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{6.20}$$

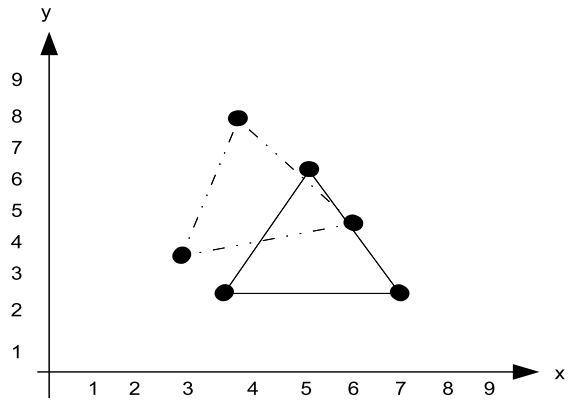

Figura 6.5: Rotación respecto al origen

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} S_x & 0 & x_f(1-S_x) \\ 0 & S_y & y_f(1-S_y) \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{6.21}$$

Por lo tanto la matriz de transformación para realizar el escalamiento respecto a un punto fijo es:

$$\begin{bmatrix} S_x & 0 & x_f(1-S_x) \\ 0 & S_y & y_f(1-S_y) \\ 0 & 0 & 1 \end{bmatrix} \tag{6.22}$$

6.5.1. Rotación respecto a un punto arbitrario

Cuando se realiza una rotación respecto al origen existe también una traslación implícita como se aprecia en la Figura 6.5

Es posible realizar una rotación respecto a un punto (x_r, y_r) seleccionado como eje de rotación como en la Figura 6.6, para ello, debemos componer tres transformaciones, la primera es una traslación tal que el punto elegido como eje de rotación coincida con el origen como en la Figura 6.7, la segunda es la rotación respecto al origen como en la Figura 6.8, obviamente el punto que

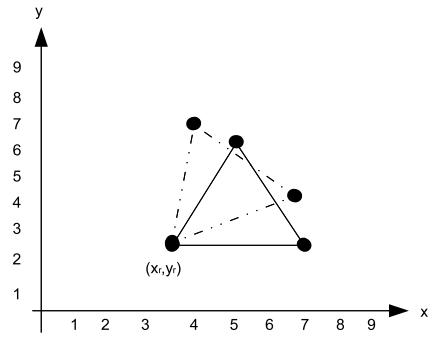

Figura 6.6: Rotación respecto a un eje de rotación (x_r, y_r)

está en el origen no será afectado en esta etapa. La última transformación es una traslación que devuelva al punto (x_r, y_r) al lugar donde se encontraba originalmente como en la Figura 6.9.

La rotación respecto a un eje de rotación se logra entonces mediante (6.23)

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_r \\ 0 & 1 & y_r \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} cos\theta & -sen\theta & 0 \\ sen\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_r \\ 0 & 1 & -y_r \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (6.23)$$

Realizando el producto matricial obtenemos (6.24) y luego 6.25:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_r \\ 0 & 1 & y_r \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} cos\theta & -sen\theta & y_r sen\theta - x_r cos\theta \\ sen\theta & cos\theta & -x_r sen\theta - y_r cos\theta \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (6.24)$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} cos\theta & -sen\theta & x_r(1-cos\theta) + y_r sen\theta \\ sen\theta & cos\theta & y_r(1-cos\theta) - x_r sen\theta \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (6.25)$$

Por lo tanto la matriz de transformación para realizar la rotación respecto a un eje arbitrario es:

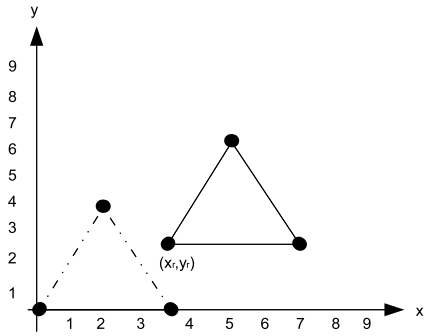

Figura 6.7: Traslación de forma que el eje de rotación coincida con el origen

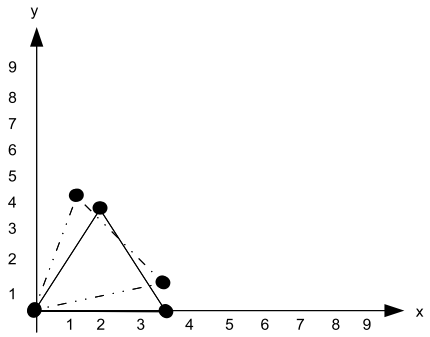

Figura 6.8: Rotación respecto al origen

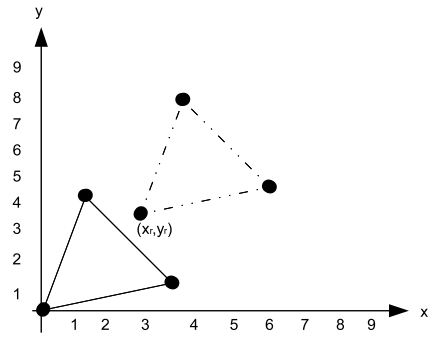

Figura 6.9: Traslación de manera que el eje de rotación regrese a su ubicación original

$$\begin{bmatrix} cos\theta & -sen\theta & x_r(1-cos\theta)+y_r sen\theta \\ sen\theta & cos\theta & y_r(1-cos\theta)-x_r sen\theta \\ 0 & 0 & 1 \end{bmatrix} \tag{6.26}$$

6.6. Reflexiones

Para reflejar una figura verticalmente utilizamos la transformación (6.27), en la Figura 6.10 (a) se muestra un ejemplo, la reflexión horizontal se logra mediante (6.28), en la Figura 6.10 (b) se muestra un ejemplo.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{6.27}$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{6.28}$$

Para reflejar una figura respecto al origen utilizamos la transformación (6.29), en la Figura 6.11 se muestra un ejemplo. Para reflejar una figura

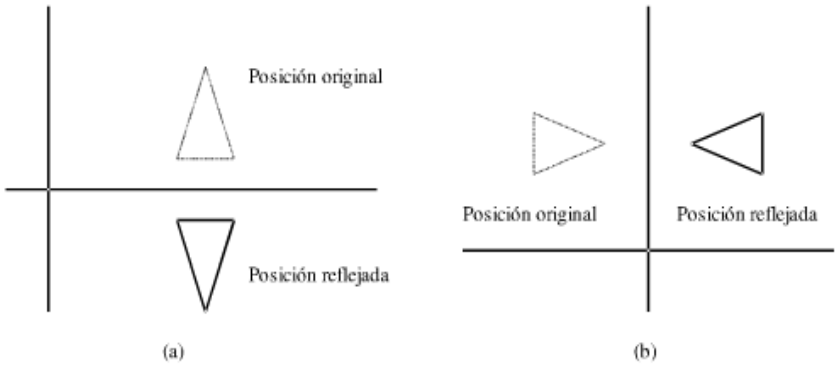

Figura 6.10: (a) Reflexión vertical, (b) Reflexión horizontal

respecto a la recta $y = x$ utilizamos la transformación (6.30), en la Figura 6.12 se muestra un ejemplo.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{6.29}$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{6.30}$$

6.7. Transformaciones Geométricas en 3D Simples

Fácilmente podemos generalizar las transformaciones geométricas para que poderlas aplicar a figuras tridimensionales. En tres dimensiones las coordenadas homogéneas son representadas por vectores de 4 componentes y las matrices de transformación son de 4 por 4, a continuación describiremos como realizar traslaciones, escalamientos y rotaciones tridimensionales:

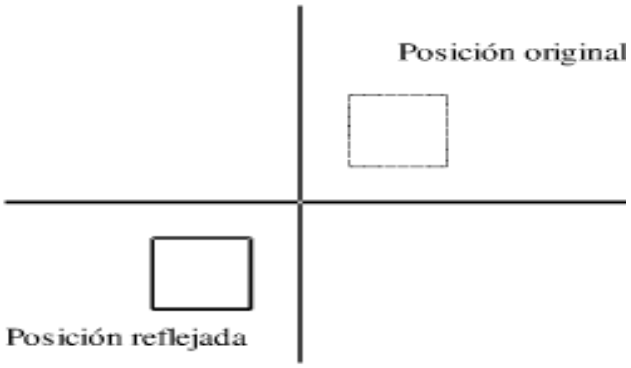

Figura 6.11: Reflexión respecto al origen

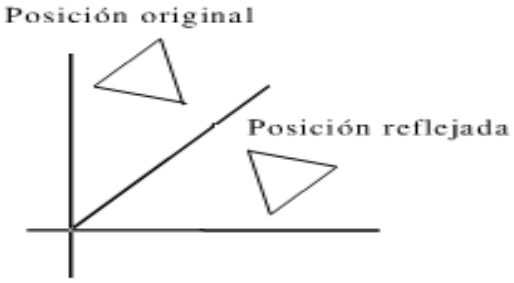

Figura 6.12: Reflexión respecto a la recta y=x

6.7.1. Escalamiento

Para trasladar un punto p utilizamos la transformación $p' = Sp$, en forma expandida es:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{6.31}$$

6.7.2. Traslación

Para trasladar un punto p utilizamos la transformación $p' = Tp$, en forma expandida es:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{6.32}$$

6.7.3. Rotación respecto al eje X

Para trasladar un punto p utilizamos la transformación $p' = R_x p$, en forma expandida es:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sen\theta & 0 \\ 0 & sen\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{6.33}$$

6.7.4. Rotación respecto al eje Y

Para trasladar un punto p utilizamos la transformación $p' = R_y p$, en forma expandida es:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} cos\theta & 0 & -sen\theta & 0 \\ 0 & 1 & 0 & 0 \\ sen\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{6.34}$$

6.7.5. Rotación respecto al eje Z

Para trasladar un punto p utilizamos la transformación $p' = R_z p$, en forma expandida es:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} cos\theta & -sen\theta & 0 & 0 \\ sen\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{6.35}$$

6.8. Rotación respecto a un eje arbitrario

No siempre queremos rotar alrededor de uno de los tres ejes, necesitamos una matriz de transformación que nos permita realizar rotar un cierto ángulo θ alrededor de un eje arbitrario, para determinar dicha matriz existen tres métodos populares, a continuación describiremos cada uno de ellos

6.8.1. Determinación de la Matriz de transformación por Composición de matrices

Para rotar un punto alrededor de un eje arbitrario un cierto ángulo θ debemos realizar las siguientes transformaciones:

1. Trasladar el punto de manera que el eje de rotación pase por el origen
2. Rotar el punto alrededor del eje x un ángulo α tal que el eje de rotación resida en el plano xz
3. Rotar el punto alrededor del eje y un ángulo β tal que el eje de rotación coincida con el eje z
4. Rotar el punto alrededor del eje z un ángulo θ
5. Rotar el punto alrededor del eje y un ángulo $-\beta$
6. Rotar el punto alrededor del eje x un ángulo $-\alpha$
7. Trasladar el punto de manera que el eje de rotación regrese a su lugar original

En forma compacta, podemos expresar la rotación de un punto un ángulo θ alrededor de un eje arbitrario de la siguiente manera:

$$p' = T^{-1}R_x^{-1}(\alpha)R_y^{-1}(\beta)R_z(\theta)R_y(\beta)R_x(\alpha)Tp \tag{6.36}$$

denotamos como T^{-1} a la transformación inversa a la transformación T, es decir, si la matriz de transformación T traslada un punto a una nueva ubicación, entonces la transformación inversa T^{-1} regresa a dicho punto a su ubicación original.

Si el eje de rotación está definido como una línea que pasa por los puntos $P_1 = (x_1, y_1, x_1)$ y $P_2 = (x_2, y_2, z_2)$, entonces la matriz de traslación que haría que dicho eje pase por el origen puede ser:

$$T = \begin{bmatrix} 1 & 0 & 0 & -x_1 \\ 0 & 1 & 0 & -y_1 \\ 0 & 0 & 1 & -z_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.37}$$

El eje de rotación se puede definir como un vector unitario $u = (a, b, c)$. Como dijimos, se requiere rotar u alrededor del eje x un ángulo α de manera que una vez rotado este coincida con el plano xz. Para determinar α proyectamos el vector u sobre el plano yz como en la Figura 6.13 obteniendo así el vector $u' = (0, b, c)$. Si a u' lo rotamos un ángulo α alrededor del eje x obtendríamos el vector unitario $u_z = (0, 0, 1)$. El producto escalar entre u' y u_z nos permite obtener el valor de $cos(\alpha)$

$$u' \cdot u_z = |u'||u_z|cos(\alpha) = (0, b, c) \cdot (0, 0, 1) = c \tag{6.38}$$

donde $|u'| = d = \sqrt{b^2 + c^2}$ y $|u_z| = 1$
entonces

$$cos(\alpha) = \frac{c}{d} \tag{6.39}$$

El producto vectorial entre u' y u_z nos permite obtener el valor de $sen(\alpha)$

$$|u' \times u_z| = |u'||u_z|sen(\alpha) = |(0, b, c) \times (0, 0, 1)| = |(b, 0, 0)| = b \tag{6.40}$$

donde $|u'| = d = \sqrt{b^2 + c^2}$ y $|u_z| = 1$
entonces

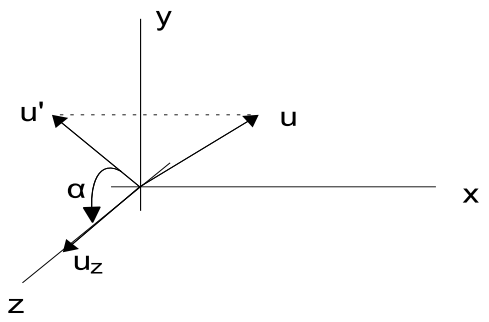

Figura 6.13: proyección de u en el plano yz para determinar el ángulo α

$$sen(\alpha) = \frac{b}{d} \tag{6.41}$$

Finalmente:

$$R_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\alpha) & -sen(\alpha) & 0 \\ 0 & sen(\alpha) & cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c/d & -b/d & 0 \\ 0 & b/d & c/d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.42}$$

En la Figura 6.14 se muestra el vector u'' que es el vector u una vez que se ha rotado alrededor del eje x el ángulo α requerido, por lo cual $u'' = (a, 0, d)$, esto es, la componente x no ha cambiado puesto que justamente la rotación fué respecto al eje x, la componente en y es cero puesto que u'' reside en el plano xz (es decir, el plano y=0) y la componente en z coincide con la magnitud de la proyección de u sobre el plano yz (el plano x=0). El producto escalar entre u'' y u_z nos permite obtener el valor de $cos(\beta)$

$$u'' \cdot u_z = |u'||u_z|cos(\beta) = (a, 0, d) \cdot (0, 0, 1) = d \tag{6.43}$$

donde $|u''| = \sqrt{a^2 + d^2} = \sqrt{a^2 + b^2 + c^2} = 1$
entonces

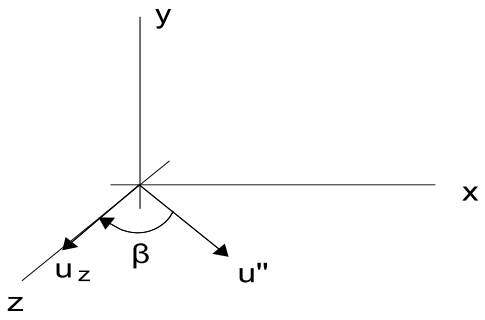

Figura 6.14: El vector u" gira un ángulo β
para alinearse con el eje z

$$cos(\beta) = d \tag{6.44}$$

El producto vectorial entre u'' y u_z nos permite obtener el valor de $sen(\beta)$

$$|u'' \times u_z| = |u''||u_z|sen(\beta) = |(a, 0, d) \times (0, 0, 1)| = |(0, 0, -a)| = a \tag{6.45}$$

entonces

$$sen(\beta) = a \tag{6.46}$$

Finalmente:

$$R_y(\beta) = \begin{bmatrix} cos(\beta) & 0 & -sen(\beta) & 0 \\ 0 & 1 & 0 & 0 \\ sen(\beta) & 0 & cos(\beta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} d & 0 & -a & 0 \\ 0 & 1 & 0 & 0 \\ a & 0 & d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.47}$$

La matriz que realiza la rotación alrededor del eje z el ángulo θ (especificado por el usuario es por supuesto:

$$R_z(\theta) = \begin{bmatrix} cos(\theta) & -sen(\theta) & 0 & 0 \\ sen(\theta) & cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.48}$$

Ahora se tienen todas la matrices especificadas en (6.36), solo habría que realizar dicho producto matricial

6.8.2. Determinación de la Matriz de transformación por conjunto de vectores ortogonales

Cualquier transformación geométrica es equivalente a un cambio de coordenadas de referencia y la rotación tridimensional respecto a un eje arbitrario no es la excepción. Si cambiamos el sistema de coordenadas referencial de manera que el eje z del nuevo sistema de coordenadas coincida con el eje de rotación especificado por el usuario, entonces ya solo habría que hacer una rotación alrededor del nuevo eje z. Para cambiar el sistema de coordenadas al nuevo sistema de coordenadas de referencia formado por los vectores unitarios u'_x, u'_y y u'_z se debe utilizar una matriz de transformación tal que la submatriz superior izquierda de 3x3 constituya una matriz ortogonal, esto implica que los vectores u'_x, u'_y y u'_z son todos ortogonales entre sí. La matriz de transformación sería entonces:

$$\begin{bmatrix} u'_{x,1} & u'_{x,2} & u'_{x,3} & 0 \\ u'_{y,1} & u'_{y,2} & u'_{y,3} & 0 \\ u'_{z,1} & u'_{z,2} & u'_{z,3} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.49}$$

donde $u'_x = (u'_{x,1}, u'_{x,2}, u'_{x,3})$, $u'_y = (u'_{y,1}, u'_{y,2}, u'_{y,3})$ y $u'_z = (u'_{z,1}, u'_{z,2}, u'_{z,3})$

Ahora bien, de acuerdo al plan, hacemos $u'_z = u$, es decir, el eje de rotación indicado por el usuario, para u'_y una opción es realizar el producto cruz de u'_z con u_x (el vector unitario que apunta al actual eje x) y luego normalizarlo para su magnitud sea unitaria, así:

$$u'_y = \frac{u'_z \times u_x}{|u'_z \times u_x|} \tag{6.50}$$

Finalmente, el tercer vector (u'_x)es simplemente un vector ortogonal al plano formado por los dos vectores que ya tenemos, entonces:

$$u'_x = u'_z \times u'_y \tag{6.51}$$

6.8.3. Determinación de la Matriz de transformación mediante Cuaterniones

Los cuaterniones son de utilidad en la geometría fractal y en la animación, veremos ahora que también son útiles para realizar rotaciones alrededor de un eje arbitrario. Un cuaternión se puede ver de dos maneras, la primera de ellas como un número complejo que tiene una parte real y tres partes imaginarias, $(q = s + ia + jb + kc)$cada una de ellas se reconoce por el operador imaginario i,j o k, bajo esta definición tenemos que:

$$i^2 = j^2 = k^2 = -1 \tag{6.52}$$

La segunda manera en que podemos visualizar un cuaternión es como un par formado por un escalar s y un vector v, de esta manera:

$$\begin{aligned} ij &= \ \ k \\ ik &= -j \\ ji &= -k \\ jk &= \ \ i \\ ki &= \ \ j \\ kj &= -i \end{aligned}$$

Sean los cuaterniones $q_1 = (s_1, v_1)$ y $q_{2=(s_2,v_2)}$, la suma de estos está dada por $q_1 + q_2 = (s_1 + s_2, v_1 + v_2)$, la diferencia está dada por $q_1 - q_2 = (s_1 - s_2, v_1 - v_2)$ y el producto está dado por: $q_1 q_2 = (s_1 s_2 - v_1 \cdot v_2, s_1 v_2 + x_2 v_1 + v_1 \times v_2)$.

Un punto P en 3D podemos tratarlo como un cuaternión si establecemos que su parte escalar es simplemente cero. Para rotar dicho punto alrededor de un eje arbitrario definido por el vector u un cierto ángulo θ realizamos el siguiente producto de cuaterniones:

$$p' = qpq^{-1} \tag{6.53}$$

donde $p' = (0, P')$ y P' es el punto rotado, $p = (0, P)$ y P es el punto antes de ser rotado y $q = (cos(\theta/2), usen(\theta/2)$. Como q es un cuaternión unitario se cumple que su inverso multiplicativo $q^{-1} = (cos(\theta/2), -usen(\theta/2)$, es decir simplemente la parte vectorial cambia de signo. θ es obviamente el ángulo que se rota y u el vector alrededor del cual se rotará al punto P.

La operación de rotación utilizando cuaterniones en lugar de matrices es muy eficiente puesto que se requieren menos operaciones aritméticas para llevarla a cabo, también requiere menos almacenamiento pues un cuaternión consta de 4 números en oposición a los 9 que requiere una matriz de rotación 3D (y 16 cuando se usan coordenadas homogéneas), sin embargo para aprovechar la composición de transformaciones es fácil obtener la matriz de transformación equivalente simplemente agrupando los términos de manera adecuada.

6.9. Transformaciones geométricas con OpenGL

Opengl cuenta con tres funciones para llevar a cabo transformaciones geométricas tridimensionales, para la traslación usamos

```
glTranslatef(tx,ty,tz)
```

donde tx, ty y tz son los parámetros de traslación en x, y y z respectivamente. Para el escalamiento usamos:

```
glScalef(Sx,Sy,Sz)
```

donde Sx, Sy y Sz son los parámetros de escalamiento en x, y y z respectivamente. Para la rotación usamos:

```
glRotatef(teta,x,y,z)
```

donde teta es el ángulo en grados que deseamos rotar alrededor de un vector especificado por x,y y z, por ejemplo para rotar 30 grados alrededor del eje y usaríamos:

```
glRotatef(30,0,1,0)
```

6.10. Manejo de pilas de matrices con OpenGL

Considere la función display siguiente:

```
void display(void) {
 glTranslatef(0.0, 0.0, .5);
 glBegin(GL.GL_TRIANGLES);
 glVertex3f( 0.0f, 1.0f, 0.0f);
 glVertex3f(-1.0f,-1.0f, 0.0f);
 glVertex3f( 1.0f,-1.0f, 0.0f);
 glEnd();
 }
```

Si obligamos a que se llame varias veces a la función display(), vemos que el triángulo se va moviendo progresivamente a lo largo de eje Z. La razón de este movimiento es que en la función display está incluida una llamada a glTranslatef(). Como se ha explicado anteriormente, las funciones de traslación multiplican la matriz actual por una matriz de traslación creada con los argumentos que se le pasan, por tanto, sucesivas llamadas a la función display() provocan sucesivas multiplicaciones de la matriz actual con el efecto que se observa de incrementar la traslación. Para solucionar este problema OpenGL dispone de unos stacks o pilas de matrices, que permiten almacenar y recuperar una matriz anterior. Aunque OpenGL dispone de pilas para las matrices GL-MODELVIEW y GL-PROJECTION, sólo se suele utilizar la pila de GL-MODELVIEW. Una pila es un almacén con funcionamiento LIFO, el último en entrar es el primero en salir, por lo que suele comparar a una pila de platos en la que sólo se puede dejar uno encima de la pila o coger el superior que es el último depositado. La pila de matrices tiene el mismo funcionamiento sustituyendo los platos por matrices. La matriz superior de la pila es sobre la que se aplican las distintas transformaciones, multiplicándola por la matriz que generan las distintas funciones.

Para poder guardar una determinada matriz y posteriormente recuperarla OpenGL dispone de las dos funciones comentadas: glPushMatrix() y glPopMatrix(). La función glPushMatrix() realiza una copia de la matriz superior y la pone encima de la pila, de tal forma que las dos matrices superiores son iguales. En la Figura 6.15 se observa la pila en la situación inicial con una sola matriz, al llamar a la función glPushMatrix() se duplica la matriz superior. Las siguientes transformaciones que se realizan se aplican sólo a la

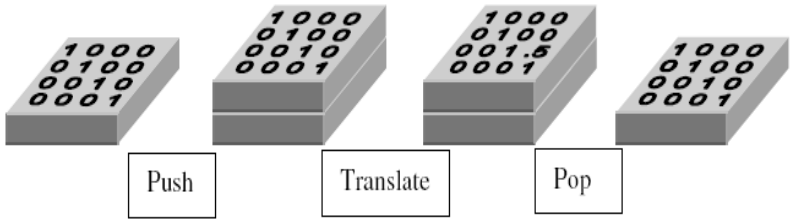

Figura 6.15: La pila de OpenGL

matriz superior de la pila, quedando la anterior con los valores que tenía en el momento de llamar a la función glPushMatrix(). La función glPopMatrix() elimina la matriz superior, quedando en la parte superior de la pila la matriz que estaba en el momento de llamar a la función glPushMatrix().

Modifiquemos la función display() de manera que al llamar a la función glPushMatrix() se realize una copia de la matriz actual. La traslación en el eje Z se realiza en la matriz superior de la pila, es decir, en la copia de la matriz, de tal forma que al llamar a la función glPopMatrix(), como se muestra en la Figura 6.15, se elimina la matriz superior, que es la que tenía el efecto de esta transformación, quedando la matriz que estaba en el momento de llamar a glPushMatrix().

```
void display(void) {
glPushMatrix();
 glTranslatef(0.0, 0.0, .5);
 glBegin(GL.GL_TRIANGLES);
 glVertex3f( 0.0f, 1.0f, 0.0f);
 glVertex3f(-1.0f,-1.0f, 0.0f);
 glVertex3f( 1.0f,-1.0f, 0.0f);
 glEnd();
 glPopMatrix();
 }
```

Capítulo 7

Visualización 3D

Ya hemos hablado de transformaciones tridimensionales de objetos en 3D pero la pantalla de la computadora (o el papel de la impresora) son superficies, es decir son de solo dos dimensiones, por lo tanto necesitamos de una forma de proyectar nuestros objetos tridimensionales en la pantalla

7.1. Proyección en paralelo

La proyección mas simple posible consiste en simplemente descartar la coordenada z, este es un caso especial del métodos denominado "proyección paralela". Una proyección paralela se forma extendiendo líneas paralelas desde cada vértice del objeto hasta que intersecten el plano de visión, los puntos de intersección con el plano son los vértices proyectados y se conectan mediante líneas de la misma manera que los vértices originales están conectados en el objeto tridimensional, ver Figura 7.1 [3].

El caso particular donde el plano de visión es el plano xy y la dirección de proyección es paralela al eje z corresponde con el simple descarte de la coordenada z. En una proyección paralela general se puede elegir cualquier dirección de proyección, digamos que la dirección de proyección está definida por el vector (x_p, y_p, z_p) y que la imagen se proyecta sobre el plano xy. Si tenemos un punto en el objeto cuyas coordenadas son (x_1, y_1, z_1) y queremos determinar el punto proyectado (x_2, y_2) entonces podemos escribir las ecuaciones paramétricas de la línea en la dirección de proyección:

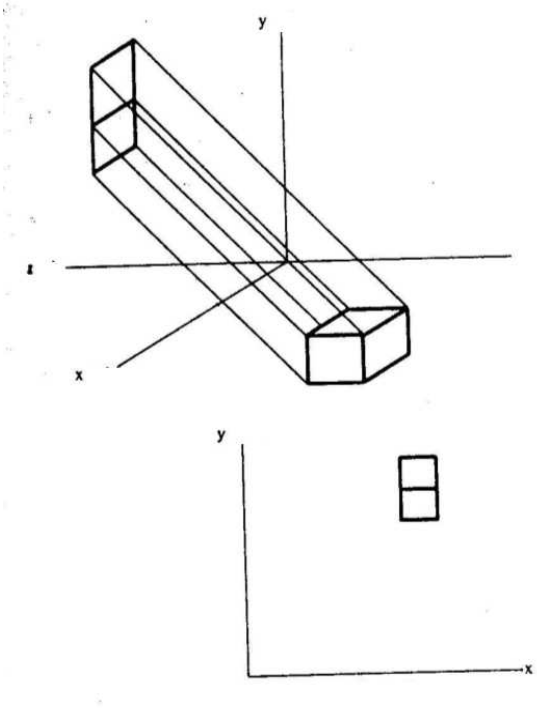

Figura 7.1: Proyección Paralela [3]

$$\begin{aligned} x &= x_1 + x_p u \\ y &= y_1 + y_p u \\ z &= z_1 + z_p u \end{aligned} \tag{7.1}$$

Ahora, dónde intersecta esta línea con el plano xy?, eso es lo mismo que preguntarse cuanto vale x y y cuando z=0, entonces:

$$u = -\frac{z_1}{z_p} \tag{7.2}$$

sustituyendo este valor en las otras dos ecuaciones tenemos:

$$\begin{aligned} x_2 &= x_1 - z_1 x_p / z_p \\ y_2 &= y_1 - z_1 y_p / z_p \end{aligned} \tag{7.3}$$

Lo cual en forma matricial en coordenadas homogéneas se expresa:

$$\begin{bmatrix} x_2 \\ y_2 \\ z_2 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & -x_p/z_p & 0 \\ 0 & 1 & -y_p/z_p & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix} \tag{7.4}$$

7.2. Proyección en perspectiva

En una proyección en perspectiva, mientras más lejos se encuentra un objeto del observador, mas pequeño parece ser, esto proveé al observador con una indicación de profundidad, es decir, que partes de un objeto se encuentran mas cercanas o mas lejanas. En la proyección en perspectiva las líneas de proyección no son paralelas sino que convergen en un punto llamado centro de proyección, estas serían las trayectorias que seguirían los rayos de luz proviniendo del objeto hacia el ojo del observador, es la intersección de estas líneas con el plano de visión que determinan la imagen proyectada, observe la Figura 7.2

La línea que va del centro de proyección (x_c, y_c, z_c) hacia el punto a proyectar (x_1, y_1, z_1) tiene las siguientes ecuaciones paramétricas:

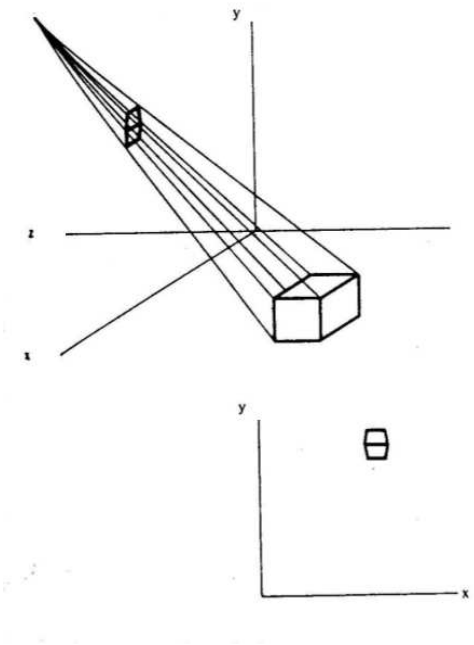

Figura 7.2: Proyección en perspectiva [3]

$$\begin{aligned} x &= x_c + (x_1 - x_c)u \\ y &= y_c + (y_1 - y_c)u \\ z &= z_c + (z_1 - z_c)u \end{aligned} \tag{7.5}$$

El punto proyectado (x_2, y_2) es aquel donde esta línea intersecta el plano de visión, si por ejemplo el plano de visión es el plano xy (el plano z=0) entonces:

$$u = -\frac{z_c}{z_1 - z_c} \tag{7.6}$$

sustituyendo este valor en las otras dos ecuaciones tenemos:

$$\begin{aligned} x_2 &= x_c - z_c \frac{x_1 - x_c}{z_1 - z_c} \\ y_2 &= y_c - z_c \frac{y_1 - y_c}{z_1 - z_c} \end{aligned} \tag{7.7}$$

Lo cual en forma matricial en coordenadas homogéneas se expresa:

$$\begin{bmatrix} x_2 w_2 \\ y_2 w_2 \\ z_2 w_2 \\ w_2 \end{bmatrix} = \begin{bmatrix} -z_c & 0 & x_c & 0 \\ 0 & -z_c & y_c & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -z_c \end{bmatrix} \begin{bmatrix} x_1 w_1 \\ y_1 w_1 \\ z_1 w_1 \\ w_1 \end{bmatrix} \tag{7.8}$$

7.3. Pipeline de visualizacion tridimensional

Un "pipeline de visualización 3D" es una especie de línea de producción que toma la descripción de una escena tridimensional y la muestra en un dispositivo de despliegue 2D

1. Sistema de coordenadas locales. Para facilitar el modelado, los vértices de un objeto están referidas normalmente respecto a un punto situado dentro del mismo objeto, por ejemplo el centro del mismo. Definir un objeto en coordenadas locales facilita su instanciación.

2. Sistema de coordenadas mundiales. Todos los objetos de una escena son transformados a un sistema de coordenadas común, en este sistema se definen las fuentes de luz, la posición y orientación de la cámara así como los atributos de las superficies como colores y texturas.

3. Sistema de coordenadas de visión (espacio de la cámara o de la vista). En este punto se establecen parámetros de visión, así, se establece un "volumen de visión" y se recorta todo lo que queda fuera de ese volumen de visión, en este espacio se lleva a cabo la eliminación de caras ocultas mediante el algoritmo de Robert (proceso conocido como "Culling")

4. Sistema de coordenadas de pantalla 3D. En el espacio de vista, el volumen de visión tiene la forma de una pirámide trunca, este espacio es transformado a otro con forma de caja con z como la profundidad de la caja, en este espacio se lleva a cabo la eliminación de superficies que no están al frente (Se conoce a este proceso como HSR por Hidden Surface Removal). También se realiza en este espacio la iluminación (decidir de que color queda cada cosa de acuerdo a la interacción luminosa entre los objetos de la escena)

5. Sistema de coordenadas de despliegue. Convertir la pantalla virtual a coordenadas reales de la pantalla

7.4. Volumen de visión

Para definir el volumen de visión debemos definir un plano de visión con un cierto ancho y una altura, además hay que establecer un plano de recorte cercano y uno lejano. El plano de recorte cercano oculta las cosas que se supone están detrás de la cámara. El plano de recorte lejano oculta las cosas que quedan demasiado lejos para poderlas ver. En la Figura 7.3 se muestra el volumen de visión que es la parte del "Frustum" que contiene la escena visible.

7.5. Funciones de visualización tridimensional de OpenGL

La función glFrustum [4] sirve para especificar el volumen de visión:

```
glFrustum(left, right, bottom, top, near, far)
```

Por ejemplo:

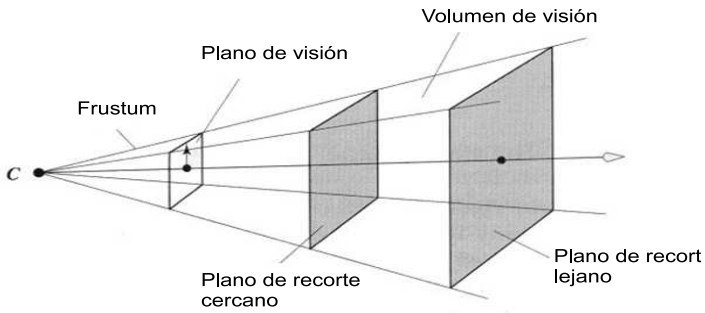

Figura 7.3: Volumen de visión

```
JGl.glMatrixMode(JGl.GL_PROJECTION);
JGl.glLoadIdentity();
JGl.glFrustum(-1, 1, -1, 1, 1.0, 20.0);
```

Alternativamente, podemos usar la función de GLU, gluPErspective para crear un volumen de visión simétrico

```
gluPerspective(fovy, aspect, near, far)
```

donde fovy es el ángulo del campo de visión en el plano yz el cual debe estar en el rango de 0 a 180 grados, aspect es la relación ancho/alto del frustum, se recomienda hacer que coincida con la relación ancho/alto de la ventana de despliegue, near y far deben ser valores positivos. Ejemplo:

```
JGlu.gluPerspective(60.0, 1.0, 1.0, 20.0);
```

Capítulo 8

Supresión de Líneas y Superficies ocultas

Un problema importante en el proceso de desplegar imágenes que luzcan reales es el de diferenciar las partes visibles de aquellas que no se podrían ver desde la posición de algún observador. Existen varios métodos para lograr esto, expondremos aquí algunos de los mas importantes.

8.1. Supresión de segmentos de líneas ocultas

No solamente para mejorar el aspecto de una gráfica o de un objeto sino para eliminar la ambigüedad de una figura y por ende poderla entender, es necesario eliminar las líneas o los segmentos de línea que no deberían verse. Para ilustrar esto, veamos como al eliminar ciertas líneas de un ambiguo cubo de alambre mostrado en la Figura 8.1(a), obtenemos el cubo no ambiguo que se muestra en la Figura 8.1(b) y al eliminar otras líneas obtenemos el cubo no ambiguo mostrado en la Figura 8.1(c)

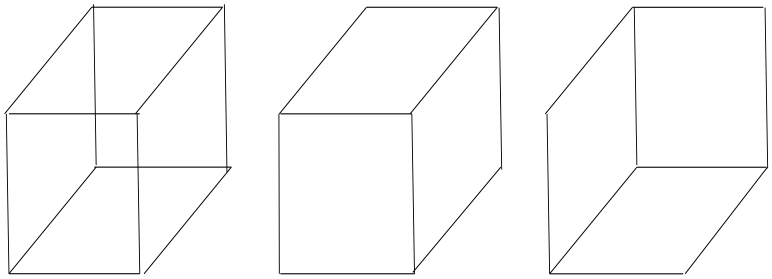

Figura 8.1: Al ocultar líneas de un cubo de alambre se elimina la ambigüedad implicita

8.1.1. Algoritmo del horizonte flotante

Este algoritmo se utiliza para desplegar funciones de dos variables $y = f(x, z)$. La función se discretiza, para cada valor discreto de z se traza la gráfica $y = f(x, z = cte)$, cada valor de z corresponde con un plano, comenzamos por valores de z que están mas cercanos al observador y con valores de z que se van alejando de este. Observe en la Figura 8.2 que un segmento de la gráfica $y = f(x, z_4)$ debe ocultarse puesto que la gráfica anterior $y = f(x, z_3)$ la cubre, esta es la base del algoritmo del horizonte flotante que se puede enunciar de la siguiente manera:

Si para cada valor dado de x, el valor y de la curva en el plano actual es mayor que el valor y para todas las curvas anteriores, entonces la curva es visible para ese valor específico de x, en caso contrario será invisible.

Para implementar este algoritmo simplemente hay que mantener un arreglo del mismo tamaño que la resolución de la imagen en el eje x. Los valores de este arreglo representan el horizonte y este horizonte "flota" a medida que cada curva es dibujada.

Este algoritmo funciona bien a menos que alguna curva adopte valores por debajo de los que se han presentado en planos anteriores, por ejemplo, en la Figura 8.3 la curva $z = z_5$ cae por debajo de todas las anteriores y debe ser visible en el segmento en el que eso sucede, algo similar se puede decir

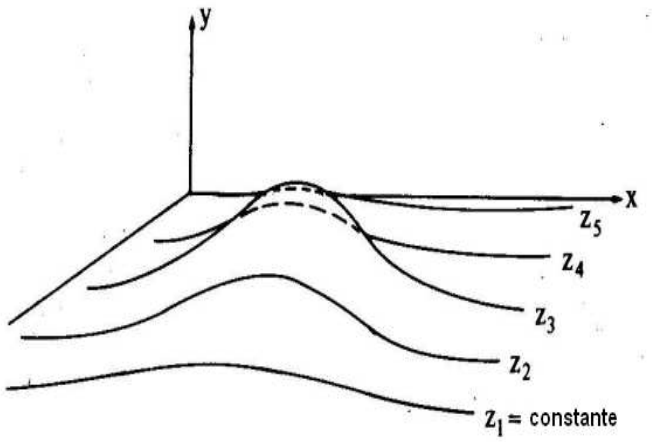

Figura 8.2: Graficación de una función $y = f(x, z)$ mediante el algoritmo del horizonte flotante

para $z = z_6$. El algoritmo del horizonte flotante se debe adaptar para tomar en cuenta esa posibilidad, el algoritmo sería entonces:

Si para algún valor dado de x, el valor y correspondiente de la curva en el plano actual es superior al máximo valor o inferior al mínimo valor entre todas las curvas anteriores, entonces la curva actual en dicho valor x es visible, si no se cumple ninguna de las dos cosas, entonces es invisible.

Para implementar el algoritmo del horizonte flotante se requieren entonces dos arreglos uno para almacenar los valores máximos y el otro para almacenar los mínimos, a estos arreglos se les denominan horizontes flotantes superior e inferior respectivamente.

8.2. Determinación de la ecuación de un plano

Podemos determinar la ecuación del plano en el que reside cualquier cara de un cuerpo rígido modelado por un número finito de ellas. Cualquier punto (x,y,z) que pertenezca al plano cumple con su ecuación que es:

$$ax + by + cz + d = 0 \tag{8.1}$$

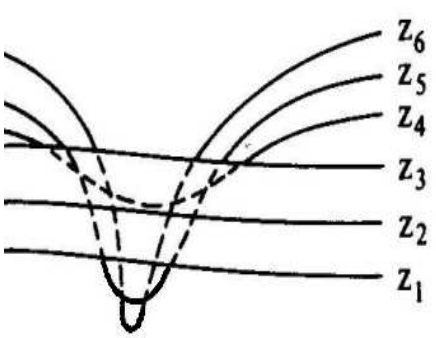

Figura 8.3: Necesidad de un horizonte inferior

o su forma normalizada (para la cual d=1) que es:

$$ax + by + cz = -1 \tag{8.2}$$

Como tenemos tres incógnitas (a, b, c), entonces necesitamos tres ecuaciones, es decir tres puntos que pertenezcan al plano (siempre y cuando no sean co-lineales), utilizamos por facilidad vértices que definen la cara cuyo plano queremos determinar, entonces si tenemos 3 vértices (x_1, y_1, z_1), (x_2, y_2, z_2) y (x_3, y_3, z_3) entonces tenemos que resolver el sistema de ecuaciones:

$$ax_1 + by_1 + cz_1 = -1 \tag{8.3}$$
$$ax_2 + by_2 + cz_2 = -1 \tag{8.4}$$
$$ax_3 + by_3 + cz_3 = -1 \tag{8.5}$$

en forma matricial tenemos:

$$\begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix} \tag{8.6}$$

8.3. Determinación del vector Normal a un Plano

Una vez que tenemos la ecuación de un plano $ax+by+cz+d=0$ entonces el vector que es normal a ese plano es:

$$N = (a, b, c) \tag{8.7}$$

también podemos determinar el vector normal al plano haciendo el producto cruz o producto vectorial de dos vectores que residan en ese plano, así es que si tenemos los vértices (x_1, y_1, z_1), (x_2, y_2, z_2) y (x_3, y_3, z_3), entonces podemos hacer el producto cruz del vector $(x_1 - x_2, y_1 - y_2, z_1 - z_2)$ y $(x_1 - x_3, y_1 - y_3, z_1 - z_3)$. El vector que resulte será un vector normal al plano al que pertenecen esos tres puntos no colineales.

8.4. Detección de caras posteriores

La detección de caras posteriores es un proceso conocido como "Culling" consiste en eliminar aquellas caras de los objetos de una escena que no pueden ser vistas desde la perspectiva de cierto observador, este proceso no resuelve el problema mas general en el que unos objetos tapan a otros o ciertas partes de estos, simplemente se encarga de averiguar cuales caras están ocultas por el mismo objeto al que pertenecen. El algoritmo de Robert resuelve este problema de forma muy simple, solo funciona para objetos convexos pero si el objeto es cóncavo siempre se puede dividir en varios objetos convexos.

8.4.1. Algoritmo de Roberts

Este método simplemente dice que una cara es posterior y por ende invisible a un observador si se cumple que

$$N_p \cdot N < 0 \tag{8.8}$$

donde N_p es el vector normal al plano al que pertenece la cara en cuestión y N es el vector que describe la dirección de la vista del observador, es decir, si el ángulo entre N_p y N es un ángulo obtuso entonces la cara es posterior, si el ángulo entre esos dos vectores es agudo, entonces es una cara visible,

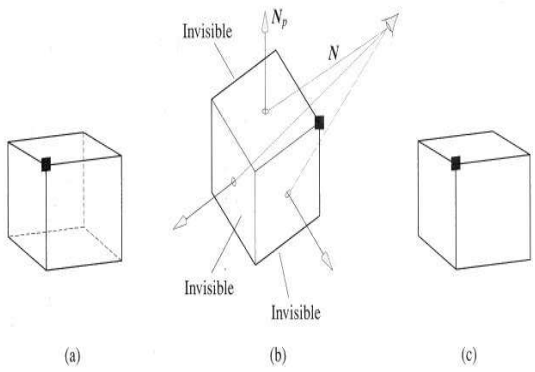

Figura 8.4: Algoritmo de Robert para detectar caras posteriores

observe la Figura 8.4. Finalmente, las líneas compartidas entre dos caras posteriores son líneas invisibles.

8.5. Algoritmo del Buffer Z o Buffer de profundidad

Las partes visibles de una escena se pueden descubrir haciendo algo similar al algoritmo del horizonte flotante, es decir, podemos mantener un arreglo con las distancias desde el observador hasta la superficie mas cercana en cada dirección que quede dentro del volumen de visión, a diferencia del algoritmo del horizonte flotante, el arreglo que necesitamos aquí es un arreglo bidimensional del tamaño de la resolución de la imagen deseada. La distancia del observador a la superficie mas cercana es la profundidad (en lugar del horizonte) y esta se mide normalmente por la coordenada z, por esta razón al buffer de profundidad también le llamamos buffer Z. El algoritmo del buffer Z o buffer de profundidad para efectos de detección de superficies visibles o lo que es lo mismo la eliminación de superficies ocultas (HSR por Hidden Surface Removal) consiste en lo siguiente:

Supongamos que todos los objetos que forman la escena tienen caras, no solo se debe pensar en objetos regulares como los cubos que tienen 6 caras o pirámides (con 4 caras) sino cualquier tipo de objeto, por ejemplo, una tetera

se puede modelar con alrededor de 500 caras triangulares.

Por cada objeto que forma parte de una escena y por cada cara que forma parte del objeto, determinar los puntos que la forman, por cada punto (x,y,z) de estos, comparar su profundidad z con la profundidad almacenada en el buffer Z para el correspondiente (x,y), si la profundidad del punto es menor, reemplaza a la del buffer Z. Los valores de profundidad de una cara se pueden obtener a partir de la ecuación del plano correspondiente a esa cara mediante:

$$z = \frac{-ax - by - d}{c} \tag{8.9}$$

Una vez que se hayan procesado todos los puntos de todas las caras de todos los objetos de la escena obtendremos en el buffer Z las profundidades mas cercanas y podremos saber de que color debe ser cada pixel (x,y) si además de mantener el buffer Z mantenemos un buffer paralelo denominado "buffer del color". Cada vez que un valor de profundidad sustituye un valor en el buffer Z se sabe a que cara pertenece el nuevo valor de profundidad y el color de dicha cara se puede almacenar en el buffer del color. En resumen:

```
IF zs < zbuffer[x,y] THEN
    write intensity to colorbuffer[x,y];
    write zs to zbuffer[x,y];
END;
```

La principal ventaja de este método es su simplicidad, las desventajas son:

- Demasiados cálculos innecesarios puesto que en demasiadas ocasiones los pixels sobreescriben pixels anteriores
- Posibles errores de cuantización
- No soporta objetos transparentes

8.6. Algoritmo del Buffer A para superficies transparentes

Como se comentó en la sección anterior, el algoritmo del buffer Z no puede manejar objetos transparentes o con cierto grado de transparencia

(translucidez). El algoritmo del buffer A es en realidad una modificación del algoritmo del buffer Z para soportar objetos translúcidos, la única razón por la que recibe ese nombre es por que la A está en el extremo opuesto a la Z en el alfabeto.

En el buffer A, en lugar de almacenar simplemente un valor de profundidad se almacena un apuntador a una lista ligada. Si el primer elemento de la lista es un objeto opaco, entonces la lista termina ahí, si en cambio se trata de un objeto translúcido, entonces la lista continúa con el siguiente objeto mas cercano en la misma dirección, el cual también pudiera ser otro objeto translúcido, la lista continúa de manera que el último objeto es opaco. Para saber el color final de un pixel se debe recorrer la lista asociada a ese pixel, el color de los objetos translúcidos y el objeto opaco de la lista correspondiente al pixel se combina para obtener el color final.

8.7. Algoritmo del Buffer Z por línea de rastreo

Para acelerar el método del buffer Z, se puede proceder por línea de rastreo, por cada línea de rastreo, las posiciones horizontales adyacentes a lo largo de la línea difieren an solo una unidad, si se ha determinado que la profundidad en la posición (x,y) es z, entonces la profundidad de la siguiente posición (x+1,y) será

$$z' = \frac{-a(x+1) - by - d}{c} = z - \frac{a}{c} \tag{8.10}$$

El valor a/c permanece constante mientras no nos cambiemos de superficie de manera que al proceder por línea de rastreo, las profundidades se van determinando mediante una simple suma.

8.8. Método de proyección de rayos

En lugar de procesar cada punto de cada polígono de cada objeto de una escena para determinar cual es la profundidad de la superficie mas cercana al observador en cada pixel del plano de visión podemos de otra manera. El método de proyección de rayos consiste en trazar una línea perpendicular al plano de visión, determinar a cuales polígonos intersecta esta línea y elegir

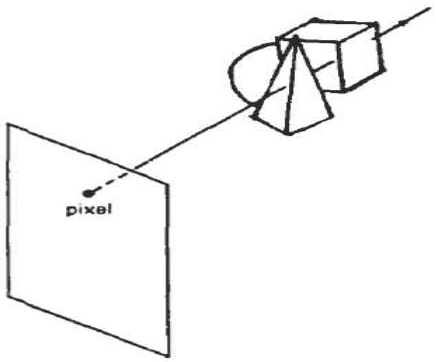

Figura 8.5: Detección de superficies visibles mediante el método de trazado de rayos

la intersección mas cercana al plano de visión, esto se hace por cada pixel del plano de visión. En la Figura 8.5 se ha trazado un rayo para cierto pixel y este intersecta 4 polígonos (2 caras de la pirámide y dos caras del cubo), pero la intersección mas cercana al plano de visión es la cara que resultaría visible al menos en lo que concierne al pixel de donde surgió el rayo. En el método del buffer de profundidad se procesan cada uno de los polígonos de la escena en orden para determinar el mas cercano para cada pixel mientras que en este método se toman los pixels en orden y se sigue un rayo por cada pixel, este método ahorra cálculos.

8.9. Método del árbol BSP

El algoritmo del pintor es probablemente la mas simple de las técnicas para resolver el problema de HSR, se trata de ordenar los polígonos en orden ascendente de profundidad (z) y simplemente desplegarlos completos comenzando por los de mayor profundidad, los de menor profundidad cubrirán a los de mayor profundidad de la misma manera que en un lienzo al pintar un árbol, este cubre la parte del paisaje que queda detrás de el. El algoritmo del pintor falla cuando los polígonos se traslapan como en la Figura 8.6.

Para solucionar el problema de los polígonos traslapados, se puede dividir

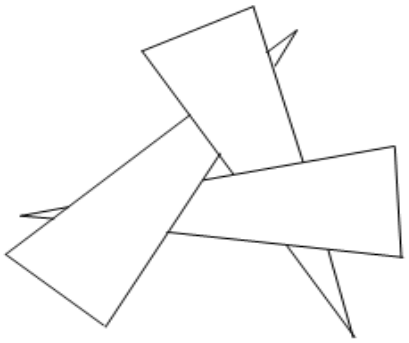

Figura 8.6: El algoritmo del pintor no funciona con polígonos traslapados

el espacio sucesivamente hasta que las diferentes regiones en las que el espacio se haya dividido sean lo suficientemente pequeñas para que en su interior ningún polígono se traslape en profundidad con ningún otro polígono. Un árbol BSP (por Binary Space Partitioning) sirve para representar la manera en que el espacio ha sido dividido. El espacio es particionado estableciendo planos que dividen a los polígonos en aquellos que quedan de un lado y aquellos que quedan del otro lado. Un árbol BSP tiene tantos niveles como planos divisorios, cada plano divisorio divide el espacio en dos (de ahí lo de binario). En la Figura 8.7, el primer nivel del árbol corresponde con el plano y=512 y el segundo nivel corresponde con el plano x=512, dividiendo de esta manera el espacio en 4 regiones.

8.10. Funciones de OpenGL para suprimir superficies ocultas

OpenGL implementa tanto la eliminación de caras posteriores (Algoritmo de Robert) como la eliminación de superficies ocultas mediante el método del buffer Z de profundidad. La eliminación de caras posteriores se lleva a efecto mediante las funciones:

```
glEnable(GL_CULL_FACE);
```

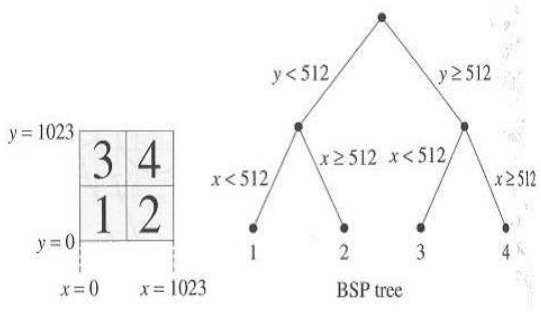

Figura 8.7: El árbol BSP

```
glCullFace(modo);
```

donde modo puede tomar el valor de GL-FRONT, GL-BACK o GL-FRONT-AND-BACK. Utilizamos GL-BACK (El modo predeterminado) para eliminar caras posteriores, si usamos GL-FRONT se eliminan las caras frontales, esto tiene aplicación por ejemplo para ver el interior de edificios de departamentos. Si utilizamos GL-FRONT-AND-BACK se eliminan tanto las caras frontales como las posteriores, quedando solamente los puntos y líneas que no forman polígonos. La eliminación de caras se desactiva mediante la función:

```
glDisable(GL_CULL_FACE);
```

Para usar las rutinas de detección de visibilidad mediante buffer de profundidad debemos solicitar que configure un buffer de profundidad y un buffer de refresco (el buffer del color), esto lo hacemos de la siguiente manera:

```
glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB|GLUT_DEPTH);
```

Para inicializar el buffer de profundidad y el buffer del color hacemos:

```
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
```

Como los valores de profundidad en el sistema de coordenadas de pantalla 3D están normalizados y quedan en el rango de cero a uno, esta instrucción asigna puros valores de 1.0 al buffer de profundidad. Finalmente, la detección de la visibilidad se activa mediante:

```
glEnable(GL_DEPTH_TEST);
```

y se desactiva mediante:

```
glDisable(GL_DEPTH_TEST);
```

Capítulo 9

Iluminación

Los conceptos de la óptica son útiles para decidir como deben iluminarse (o sombrearse) las diferentes partes de una escena. Podemos utilizar los conocidos aspectos acerca de la interacción de la luz con una superficie en términos de las propiedades de la superficie misma y de la naturaleza de la luz incidente para determinar el tono final que se debe dar a cada pixel de la escena. Sabemos que hay dos tipos fundamentales de reflexión de la luz, a saber, reflexión difusa y reflexión especular.

9.1. Reflexión Difusa

Dependiendo de las propiedades de una superficie no transparente, cuando un rayo de luz incide sobre ella, pueden ocurrir tres cosas, el rayo puede reflejarse en una dirección única como por ejemplo cuando el rayo incide en un espejo, denominamos a este tipo de reflexión como "especular perfecta", ver Figura 9.1(a). Si el rayo es reflejado en varias direcciones pero principalmente en una dirección predominante, decimos que la reflexión es "especular imperfecta", ver la Figura9.1(b), este tipo de reflexión la encontramos en objetos brillantes, por ejemplo en un florero de plata. Finalmente, si después de incidir un rayo de luz en una superficie el rayo es reflejado de manera uniforme en todas direcciones, decimos que la reflexión es "difusa" como en la Figura 9.1(c), este tipo de reflexión la encontramos en objetos opacos.

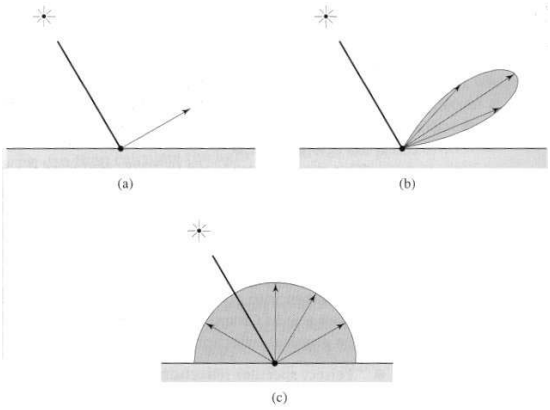

Figura 9.1: Tipos de reflexión luminosa

9.2. Ecuación de Lambert

En la reflexión difusa la intensidad de luz parece ser la misma cuando la superficie se observa desde cualquier dirección. La intensidad no depende pues de la ubicación del observador pero si depende de la orientación que la superficie tiene respecto a la fuente de luz. La ecuación de Lambert nos dice que la intensidad de luz en una superficie debida a la reflexión difusa se calcula como:

$$I_d = I_i cos\theta = I_i(L \cdot N) \quad 0 \leq \theta \leq 2\pi \tag{9.1}$$

donde I_i es la intensidad de la luz incidente, θ es el ángulo entre el vector L que apunta hacia la fuente de luz desde la superficie en cuestión y el vector N que es normal a la superficie, ver la Figuras 9.2 y 9.3

Para múltiples fuentes de luz, la ecuación de Lambert se convierte en

$$I_d = \sum_n I_{i,n}(L_n \cdot N) \tag{9.2}$$

donde $I_{i,n}$ es la intensidad de luz que incide desde la n-ésima fuente de luz, L_n es el vector que apunta hacia la n-ésima fuente de luz y N es de nuevo el vector normal a la superficie en el punto en donde se está calculando la iluminación (N no depende de n)

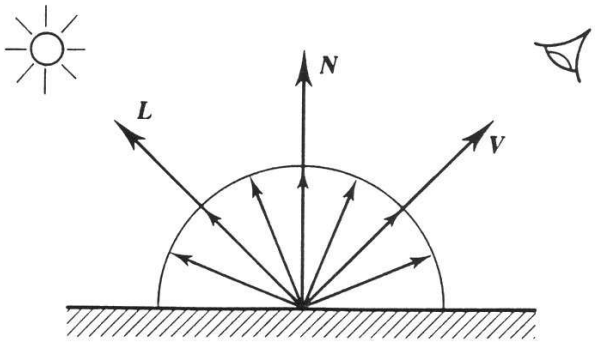

Figura 9.2: La reflexión difusa y definición de los vectores L y N involucrados en la ecuación de Lambert

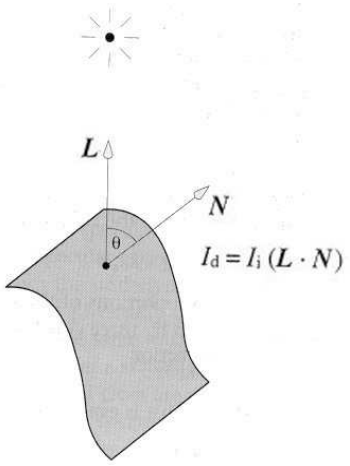

Figura 9.3: La ecuación de Lambert

9.3. Reflexión Especular

La cantidad de luz que se ve reflejada en un punto específico de una superficie donde la luz se refleja de forma especular depende de la posición del observador y no solo de la posición de la fuente de luz, suponga por ejemplo que un florero de plata es iluminado por una lámpara incandescente y hay varios observadores ubicados en posiciones distintas, entonces ellos verán que el florero refleja una luz brillante pero cada uno de ellos verá ese brillo reflejado en un lugar distinto en el florero. Por la razón aquí expuesta es que la fórmula para calcular la intensidad de luz que un punto de una superficie refleja especularmente toma en cuenta no solo la dirección hacia la fuente de luz y la dirección normal a la superficie sino la dirección hacia el observador, la fórmula es:

$$I_s = I_i cos^n \Omega = I_i (R \cdot V)^n \tag{9.3}$$

donde I_i es la intensidad de la luz incidente, V es el vector que tiene la dirección hacia el observador Ω es el ángulo entre la dirección hacia el observador y la dirección R que es el vector L "reflejado" a través del vector N (Normal al plano), así el ángulo θ entre L y N es el mismo ángulo que hay entre N y R, vea la Figura 9.4. Finalmente, n es un indicador del grado de perfección de la reflexión especular, a medida que n aumenta la reflexión especular se parece mas a la reflexión especular perfecta, vea la Figura 9.5.

9.4. Modelado de la Iluminación

Las reflexión no necesariamente es solamente difusa o solamente especular, se puede dar una combinación de reflexiones, es decir, la luz reflejada puede tener un componente difuso y un componente especular e incluso un tercer componente que es la luz reflejada que no proviene de una fuente de luz puntual sino de la contribución de luz reflejada de todas direcciones (la luz que rebota de todos los objetos y que va a dar al punto donde estamos calculando el brillo), llamamos a este componente la luz ambiental. La siguiente ecuación toma en cuanta estos tres factores y se conoce como el "Modelo de iluminación de Phong"

$$I = k_a I_a + I_i (k_d (L \cdot N) + k_s (R \cdot V)^n) \tag{9.4}$$

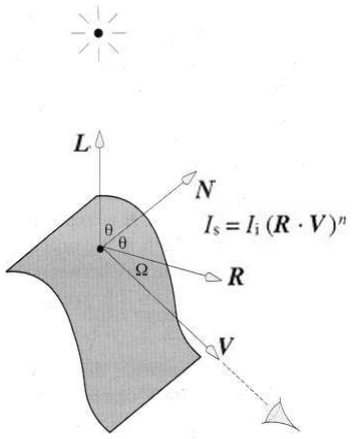

Figura 9.4: Determinación de la reflexión especular

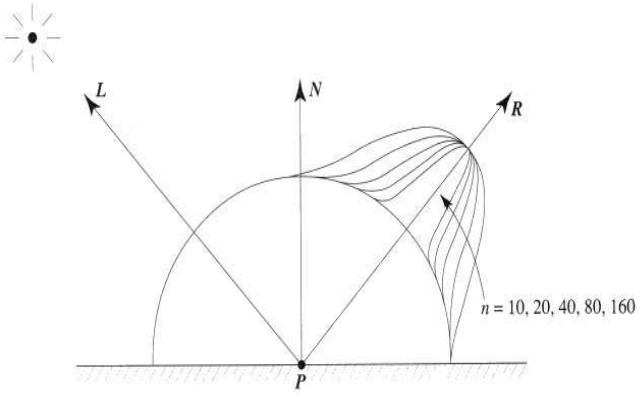

Figura 9.5: Impacto del exponente n (grado de perfección de la reflexión especular)

donde k_a es el factor de reflexión de la superficie a la luz ambiental, k_d y k_s son el factor de reflexión difusa y de la reflexión especular de la superficie respectivamente.

En algunos paquetes gráficos el modelo de iluminación de Phong se modifica para incluir un factor de atenuación, después de todo, las lámparas tienen normalmente un cierto alcance y la luz se va atenuando a medida que nos alejamos de estas, en la ecuación:

$$I = k_a I_a + f_{att} I_i (k_d (L \cdot N) + k_s (R \cdot V)^n) \tag{9.5}$$

donde $f_{att} = 1/d^2$ y d es la distancia a la fuente de luz.

El modelo de iluminación de Phong y cualquiera de sus variantes es considerado un modelo de iluminación local debido a que solo toma en cuanta aspectos locales (excepto por el término de luz ambiental) pero existen modelos de iluminación mucho mas elaborados que toman en cuenta no solo la luz que llega directamente de la fuente de luz sino indirectamente, es decir, reflejada de otros objetos cercanos al punto donde queremos determinar el nivel de iluminación, este tipo de modelos se conocen como "globales" y los mas conocidos son el método de trazado de rayos y el de radiosidad que se discuten mas adelante.

9.5. Determinación del vector Normal a un vértice

Hemos visto como determinar el vector normal a un plano, sin embargo, las interpolaciones necesarias para renderizar un objeto de manera que este se vea mas real como el sombreado de Gourad o el de Phong que se verán en las siguientes dos secciones utilizan el concepto de vector normal a un vértice, el cual es simplemente el promedio de los vectores normales a los polígonos que comparten al vértice como se aprecia en la Figura 9.6, entonces, el vector Normal a un vértice se determina mediante la siguiente fórmula:

$$N_v = \frac{\sum_{k=i}^{n} N_k}{|\sum_{k=i}^{n} N_k|} \tag{9.6}$$

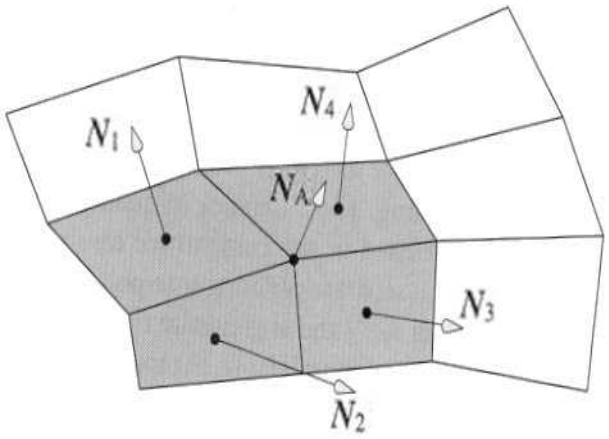

Figura 9.6: Las normales que participan en la determinación de la normal al vértice A

9.6. Sombreado de Gourad

Para lograr que un objeto construido a base de cientos de polígonos luzca como si estuviera hecho por suaves superficies curvas se puede utilizar la interpolación de Gourad, también llamada sombreado de Gourad. Hay que entender que los valores de intensidad de luz obtenidos con el modelo de Phong o alguna variante se calculan solo en los vértices, si mantenemos esa intensidad constante hasta llegar al siguiente vértice, entonces los polígonos que conforman el objeto van a ser muy evidentes. Si en cambio, los valores de intensidad se van cambiando gradualmente entre un vértice y otro (por interpolación entre las intensidades de ambos vértices), entonces obtendremos un objeto que luce bastante real. El proceso de sombreado de Gourad de cada polígono de que está hecho el objeto consiste en:

- Determinar el vector normal en cada uno de los vértices del polígono
- Aplicar el modelo de iluminación de Phong o alguna variante para determinar la intensidad en cada vértice
- Interpolar linealmente las intensidades de los vértices para determinar

la intensidad en cada pixel que forma parte del polígono

Se van determinando los valores de intensidad a lo largo de cada línea de rastreo donde se ubica el polígono. Asi, considere que se van a determinar los valores a los largo de la línea de rastreo que se muestra en la Figura 9.7, para ello debemos determinar primero las intensidades en los extremos de la línea de rastreo en la frontera del polígono, es decir los valores I_a e I_b de la siguiente manera:

$$I_a = \frac{I_2(y_s - y_0) + I_0(y_2 - y_s)}{y_2 - y_0} \tag{9.7}$$

$$I_b = \frac{I_1(y_s - y_0) + I_0(y_1 - y_s)}{y_1 - y_0} \tag{9.8}$$

Una vez que se han calculado las intensidades en los extremos de la línea de rastreo la intensidad en cada pixel que forma parte de la línea se determina interpolando I_a e I_b de la siguiente manera:

$$I_s = \frac{I_a(x_b - x_s) + I_b(x_s - x_a)}{x_b - x_a} \tag{9.9}$$

Para acelerar el sombreado de Gourad se pueden realizar cálculos incrementales a lo largo de la línea de rastreo, es decir, modificar el valor de la intensidad del pixel anterior simplemente sumándole una constante ΔI_s (diferente para cada línea de rastreo) para determinar la intensidad del pixel actual de la siguiente manera:

$$I_{s,n} = I_{s,n-1} + \Delta I_s \tag{9.10}$$

donde $\Delta I_s = (I_b - I_a)/(x_b - x_a)$

9.7. Sombreado de Phong

El sombreado de Gourad no es adecuado cuando para la reflexión especular, la solución consiste en que en lugar de interpolar intensidades luminosas interpolemos las normales de los vértices de manera que obtengamos un vector normal en cada pixel del polígono. Por ejemplo, en la Figura 9.8 diferentes vectores normales son obtenidos interpolando los vectores N_A y

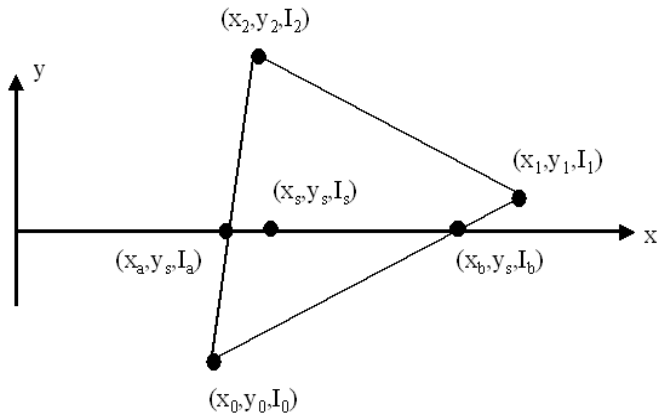

Figura 9.7: Interpolación de Gourad a lo largo de una línea de rastreo

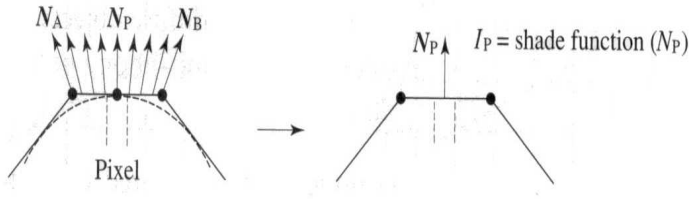

Figura 9.8: Interpolación de Phong

N_B, mas parecidos a N_A para pixeles cercanos al vértice izquierdo y cada vez mas parecidos a N_B a medida que nos acercamos al vértice derecho.

Una vez que se han obtenidos los vectores normales para cada pixel del polígono se calcula la intensidad luminosa de acuerdo al modelo de Phong o alguna de sus variantes. Este método es obviamente mas costoso que el del sombreado de Gourad pero se obtiene mejores resultados en lo concerniente a la reflexión especular, por supuesto si no hay reflexión especular no tiene caso utilizar la interpolación de Phong.

9.8. Funciones de OpenGL para manejo de Iluminación

Para definir una fuente de luz, debemos establecer sus parámetros. La siguientes dos líneas establecen una fuente de luz en la posición (-2,2,2), el cuarto elemento del arreglo light-pos indica si los tres elementos anteriores del arreglo se deben interpretar como una posición (uno) o como una dirección (cero) lo cual implica que la luz se considera local o muy lejana respectivamente [5].

```
GLFloat light_pos[]=(-2.0,2.0,2.0,1.0);
glLightfv(GL_LIGHT0,GL_POSITION,light_pos);
```

Las siguientes dos líneas sirven para especificar la luz ambiental como blanca (primeros tres valores del arreglo light-Ka)

```
GLFloat light_Ka[]=(1.0,1.0,1.0,1.0);
glLightfv(GL_LIGHT0,GL_AMBIENT,light_Ka);
```

Las siguientes dos líneas sirven para especificar la luz difusa como blanca (primeros tres valores del arreglo light-Kd)

```
GLFloat light_Kd[]=(1.0,1.0,1.0,1.0);
glLightfv(GL_LIGHT0,GL_DIFFUSE,light_Kd);
```

Las siguientes dos líneas sirven para especificar la luz especular como blanca (primeros tres valores del arreglo light-Ks)

```
GLFloat light_Ks[]=(1.0,1.0,1.0,1.0);
glLightfv(GL_LIGHT0,GL_SPECULAR,light_Ks);
```

Los coeficientes de reflexión dependen del material del que están hechos los objetos, las siguientes dos líneas sirven para especificar una superficie que al incidir luz ambiental blanca la reflejará roja

```
GLFloat material_Ka[]=(1.0,0.0,0.0,1.0);
glMaterialfv(GL_FRONT,GL_AMBIENT,material_Ka);
```

Las siguientes dos líneas sirven para especificar los coeficientes de reflexión difusa para cada uno de los tres colores básicos.

```
GLFloat material_Ka[]=(1.0,1.0,1.0,1.0);
glMaterialfv(GL_FRONT,GL_DIFFUSE,material_Kd);
```

Las siguientes dos líneas sirven para especificar los coeficientes de reflexión especular para cada uno de los tres colores básicos.

```
GLFloat material_Ka[]=(1.0,1.0,1.0,1.0);
glMaterialfv(GL_FRONT,GL_SPECULAR,material_Kd);
```

Las siguientes dos líneas sirven para especificar que un objeto tiene luz propia y el color de esta.

```
GLFloat material_Ke[]=(1.0,1.0,1.0,1.0);
glMaterialfv(GL_FRONT,GL_EMISSION,material_Ke);
```

La siguiente línea sirve para especificar que tan perfecta es la reflexión especular, es decir, el exponente n del modelo de iluminación de Phong, en este ejemplo se le asigna el valor de 10.

```
glMaterialfv(GL_FRONT,GL_SHININESS,10);
```

9.9. Iluminación por proyección de Rayos

El modelo de reflexión de Phong es un modelo local puesto que solo considera la reflexión de la luz que proviene directamente de la(s) fuente(s) de luz. Un modelo de reflexión global toma en cuenta también la luz que llega de manera indirecta, es decir, la luz reflejada de un tercer objeto [6], [7].

El modelo de iluminación por proyección de rayos toma en cuenta la interacción especular global y es dependiente del observador. Existen dos

formas de realizar la proyección de los rayos, de la fuente de luz hacia los polígonos o de los polígonos hacia la fuente de luz, siendo esta última la mas fácil de implementar.

La proyección recursiva de rayos es un método que resuelve simultáneamente los siguientes problemas:

- Detección de superficies visibles
- Sombreado por iluminación directa
- Agregar efectos de reflexión especular global
- Determinación de la geometría de las sombras

El modelo considera que la intensidad de luz en cada punto tienen tres componentes, la contribución de luz reflejada de otros objetos, la contribución de luz transmitida (en caso de ser un objeto translúcido) que proviene de otro objeto y la contribución local. La intensidad de luz $I(P)$ en cierto punto P se determina sumando tres términos, el primero es la contribución local ($I_{local}(P)$) y se determina con el modelo de Phong, el segundo es la cantidad de luz reflejada dependiendo del coeficiente de reflexión K_{rg} y de la cantidad de luz $I(P_r)$ reflejada de por otro punto (P_r) y que a su vez se determina de la misma manera (es un método recursivo). El tercer término es la cantidad de luz transmitida (refractada) que depende del coeficiente de refracción K_{tg}) y de la cantidad de luz refractada $I(P_t)$ que proviene de algún punto P_t y que a su vez se calcula de la misma manera. En resumen usamos la siguiente fórmula:

$$I(P) = I_{local}(P) + K_{rg}I(P_r) + K_{tg}I(P_t) \tag{9.11}$$

La determinación de $I(P)$ en la fórmula anterior es un proceso recursivo ya que para determinar $I(P_r)$ e $I(P_t)$ se hace uso de la misma fórmula.

Cabe aclarar que en este método la reflexión difusa se calcula solo de manera local, es decir, sin tomar en cuenta la luz que llega de manera indirecta, solo toma en cuenta la luz que llega directamente de la fuente de luz, en cambio, se modela la reflexión especular de manera global.

En la Figura 9.9 se muestra el método de proyección de rayos, en su versión mas simple se proyecta un solo rayo por cada pixel del plano de visión, dicho rayo sigue una dirección ortogonal al plano de visión. Se encuentra la primera intersección del rayo inicial con un punto, en el ejemplo este punto ha

sido etiquetado cono el número 1 y pertenece a una esfera semi-transparente. Para determinar la intensidad luminosa en el punto 1, se suman las contribuciones de las luces que le llegan:

1. Directamente de la fuente de luz. Con la línea punteada se indica la dirección hacia la fuente de luz, también se indica la dirección de la normal a la superficie donde está el punto, la dirección del observador sería la del rayo inicial. Estas direcciones se deben conocer para poder calcular la contribución local usando el modelo de Phong o alguna de sus variantes.

2. La luz que llega al punto 1 reflejada de el objeto mas cercano que se encuentre en la dirección de reflexión R, recordemos que esta dirección está a un ángulo θ respecto a la normal, mismo ángulo que tiene el rayo incidente respecto a la normal. En el ejemplo, el rayo proyectado en la dirección de reflexión intersecta con el punto 4 que pertenece a un objeto opaco, la cantidad de luz en el punto 4 a su vez se calcula siguiendo el mismo método, es decir calculando la contribución directa de la fuente de luz y la que le llega reflejada de algún otro objeto en la dirección de reflexión (dado que es un objeto opaco no se sigue la dirección de refracción).

3. La luz que llega al punto uno refractada desde el objeto mas cercano en la dirección de transmisión que en este ejemplo es donde se ubica el punto 2 que en este caso pertenece al mismo objeto semi-transparente y cuya intensidad se calcula de la misma manera.

Tal como se ilustra en la Figura 9.9 para implementar este método conviene construir un árbol donde se van guardando las diferentes contribuciones de luz, las hojas de este árbol son las primeras en tener completa su información y la raíz es la última en completarla, entonces se puede decir cual es la intensidad de luz que corresponde a un pixel. La altura del árbol es un parámetro que se interpreta como el número de intersecciones que se desea encontrar cuando se le da seguimiento al rayo, un árbol de mayor altura corresponde con una mejor renderización pero evidentemente mas costosa.

Para encontrar la dirección de refracción T usamos la ley de Snell

$$T = -\frac{\eta_1}{\eta_2}L - \left(cos\theta_2 - \frac{\eta_1}{\eta_2}cos\theta_1\right)N \tag{9.12}$$

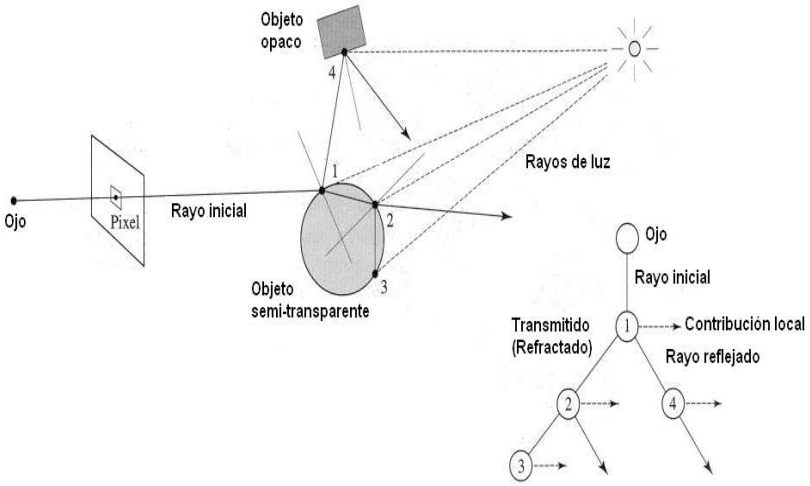

Figura 9.9: Iluminación mediante proyección de rayos

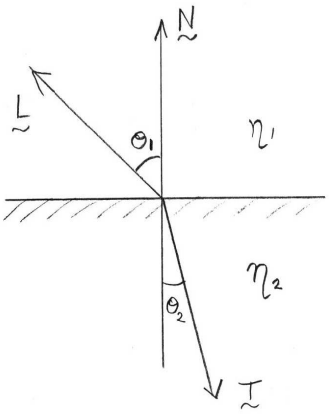

Figura 9.10: Determinación de la Dirección de Refracción

donde η_1 es el índice de refracción del material incidente y η_2 es el índice de refracción del material refractante. En la tabla 9.1 se muestran los índices de refracción de algunos materiales [8].

9.10. Radiosidad

Este método toma en cuenta la interacción global de la reflexión difusa y es independiente de la posición del observador, a diferencia del método de proyección de rayos el método de radiosidad no resuelve el problema de la detección de superficies visibles. El método de radiosidad divide a todos los objetos de una escena en parches y cada parche recibe luz de todos los demás parches de la escena y no solo de una dirección como en el método de proyección de rayos. Se asume que todas las superficies son difusores prefectos, es decir, que la luz incidente es reflejada de manera uniforme en todas direcciones. Cada parche recibe luz de cada uno de los demás parches y la interacción entre ellos depende de la relación geométrica que hay entre los mismos y solo de eso por lo cual la solución es independiente de la ubicación del observador. Como la complejidad de este método es $O(n^2)$, entonces se procura hacer parches grandes (compuestos de muchos polígonos).

Un "factor de forma" nos indica en que proporción la luz reflejada por un parche incide en otro parche. El factor de forma entre dos parches de

Tabla 9.1: Indices de refracción de algunos materiales

Azúcar	1.56
Diamante	2.417
Mica	1.56-1.6
Benceno	1.504
Glicerina	1.47
Agua	1.333
Alcohol etílico	1.362
Aceite de oliva	1.46

área A_i y A_j se calcula integrando el área de ambos parches para todos los diferenciales de área de ambos parches como se indica en la siguiente ecuación. En la Figura 9.11 se definen gráficamente las variables involucradas

$$F_{A_iA_j} = F_{ij} = \frac{1}{A_i}\int_{A_i}\int_{A_j}\frac{cos\phi_i cos\phi_j}{\pi r^2}dA_j dA_i \tag{9.13}$$

Si r es razonablemente grande:

$$F_{ij} \approx \int_{A_j}\frac{cos\phi_i cos\phi_j}{\pi r^2}dA_j \tag{9.14}$$

La radiosidad B se define como la energía luminosa por unidad de área que sale de una superficie en cada instante, la cual es la suma de la energía luminosa reflejada y la luz propia que algunas superficies tienen capacidad de producir. La radiosidad multiplicada por el área es igual a la energía propia emitida mas la energía reflejada. Para un parche A_i:

$$B_i dA_i = E_i dA_i + R_i \int_j B_j F_{ij} dA_j \tag{9.15}$$

donde R_i es la fracción de luz reflejada (el coeficiente de reflexión)
La relación de reciprocidad nos dice que:

$$F_{ij}A_i = F_{ji}A_j \tag{9.16}$$

dividiendo la ecuación (9.15) por dA_i obtenemos:

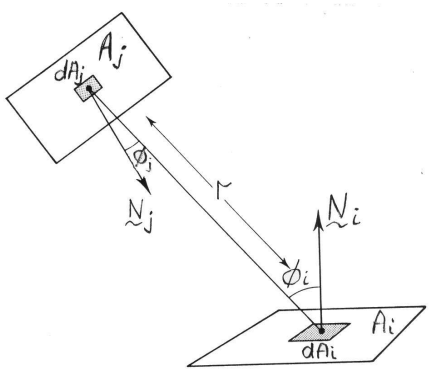

Figura 9.11: Determinación de los factores de forma

$$B_i = E_i + R_i \int_j B_j F_{ij} \tag{9.17}$$

En un ambiente discreto, la integral es reemplazada por una sumatoria:

$$B_i = E_i + R_i \sum_{j=1}^{n} B_j F_{ij} \tag{9.18}$$

Tal ecuación existe para cada parche, rearreglando:

$$B_i - R_i \sum_{j=1}^{n} B_j F_{ij} = E_i \tag{9.19}$$

Involucrando a todos los parches, (eq:radio4) se convierte en un sistema de n ecuaciones simultáneas con n incógnitas (los valores B_i). El método de radiosidad consiste justamente en resolver este sistema de ecuaciones:

$$\begin{bmatrix} 1 - R_1F_{11} & -R_1F_{12} & ... & -R_1F_{1n} \\ -R_2F_{21} & 1 - R_2F_{22} & ... & -R_2F_{2n} \\ : & : & : & : \\ -R_nF_{n1} & 1 - R_nF_{n2} & ... & -R_nF_{nn} \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ : \\ B_n \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ : \\ E_n \end{bmatrix} \tag{9.20}$$

E_i son las fuentes de luz de la escena, la mayoría de los elementos de ese vector valen cero. Para resolver este sistema se suele utilizar un algoritmo iterativo como el Gauss-Seidel para poder obtener soluciones parciales (después de un cierto número de iteraciones) debido a que la solución exacta es demasiado costosa

9.11. Funciones de OpenGL para iluminación

OpenGl no soporta las técnicas de radiosidad ni de trazado de rayos [7], [5], [4].

Bibliografía

[1] D. F. Rogers, *Procedural Elements for Computer Graphics.* MacGraw Hill, 1985.

[2] D. Hearn and P. M. Baker, *Gráficos por computadora con OpenGL. 3a Edición*, 3rd ed. Prentice Hall, 2006.

[3] S. Harrington, *Computer Graphics. A programming approach.* MacGraw Hill, 1987.

[4] D. Shreiner, M. Woo, J.Ñeider, and T. Davis, *OpenGL Programming Guide. (The Red book) Fifth edition*, 5th ed.

[5] S. R. Buss, *3-D Computer Graphics. A Mathematical Introduction with OpenGL.* Cambridge University Press, 2003.

[6] A. Watt, *3D Computer Graphics. Tercera Edition*, 3rd ed. Addisson Wesley, 2000.

[7] L. Ammeraal and K. Zhang, *Computer Graphics for Java Programmers. Second Edition*, 2nd ed. John Wiley and Sons, 2007.

[8] N. I. Koshkin and M. G. Shirkévich, *Manual de Física elemental.* MIR, 1975.