

3

Computación Gráfica en Dos Dimensiones



3.1 Estructuras de Datos y Primitivas

En lo que sigue de este libro suponemos que existe un procedimiento `putpixel(x,y,c)` o equivalente, y que además es posible determinar el rango de las coordenadas de pantalla `maxx`, `maxy` y el rango de definiciones de colores `numcol`. De esa manera podemos definir estructuras de datos para manejar los elementos de pantalla:

```
type xres = 0..maxx;
      yres = 0..maxy;
      col = 0..numcol;

type pixel = record x:xres;
                   y:yres;
                   c:col
               end;
```

Los algoritmos de discretización de primitivas nos permiten representar los distintos elementos gráficos con cierta independencia de dispositivo. Si elaboramos un sistema en el cual todas las diversas entidades gráficas están compuestas solamente por dichas primitivas, entonces el mismo puede transportarse de un dispositivo de raster a otro de mayor o menor resolución utilizando la discretización de primitivas ya mostrada, o también puede portarse a un dispositivo de vectores utilizando directamente las primitivas que éste provee.

Por lo tanto es necesario definir un conjunto de estructuras de datos para poder representar las primitivas en el espacio de la escena, y estructurar todas las entidades gráficas en términos de dicha representación. Por ejemplo podemos definir tipos para las siguientes entidades:

```
type punto = record x,y:real;
                  c:col
                end;

type linea = record p0,p1:punto;
                 c:col
               end;

type circulo = record centro:punto;
                 radio:real;
                 c:col
               end;

type cuad = record p0,p1,p2,p3:punto
                c:col
              end;
```

Los procedimientos para graficar las primitivas gráficas pueden implementarse de muchas maneras. Una opción es la siguiente:

```
procedure graficar_punto(p:punto);
var pl:pixel;
begin
  pl.x:=round(p.x);
  pl.y:=round(p.y);
  putpixel(pl.x,pl.y,p.c);
end;

procedure graficar_linea(l:linea);
var p0,p1:pixel;
begin
  p0.x:=round(l.p0.x);
  p0.y:=round(l.p0.y);
  p1.x:=round(l.p1.x);
  p1.y:=round(l.p1.y);
  linea(p0.x,p1.x,p0.y,p1.y,l.c); %por ejemplo, Bresenham
end;
```

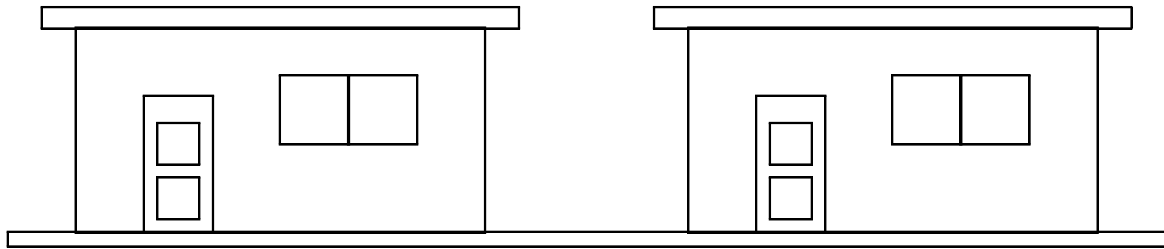


Figura 3.1 Una escena compuesta exclusivamente de instancias de cuadrados.

Una ventaja de manejar las entidades de esta manera es que la estructura de tipos nos protege de cometer errores en la llamada de los procedimientos. Por ejemplo, podemos olvidar el significado de los parámetros de llamada del procedimiento `linea` (¿Era (x_0, x_1, y_0, y_1) o (x_0, y_0, x_1, y_1) ?). Pero el procedimiento `graficar_linea` recibe un segmento de línea especificado por dos puntos, y entonces no hay error posible. Por otra parte, si el punto o la línea se van fuera del área graficable, la estructura de tipos nos avisa en tiempo de ejecución sin que se genere un error por invadir áreas de memoria no asignadas. La forma de manejar estos problemas sin producir un error de tipo será expuesta en la sección 3.4.

3.2 Transformaciones y Coordenadas Homogeneas

Una de las técnicas más poderosas de modelado de entidades gráficas consiste en descomponer ingeniosamente las mismas en términos de primitivas. Por ejemplo, la escena mostrada en la Figura 3.1 está compuesta exclusivamente por cuadrados, cada uno ubicado en un lugar específico y con una escala (tamaño) adecuada.

Sin embargo, describir la escena como *un conjunto* de instancias de cuadrados es quedarse en la superficie del fenómeno. Lo verdaderamente importante es que hay una descomposición jerárquica que es más rica y versátil. Podemos pensar que la escena está compuesta por un suelo y dos instancias de casa. Cada casa está compuesta por un techo, un frente, una puerta y una ventana, etc. De esa manera, las entidades gráficas son *jerarquías* de estructuras, que se definen como la composición de instancias de estructuras más simples, hasta terminar en el cuadrado. Es necesario, entonces, definir una única vez cada estructura.

3.2.1 Transformaciones afines

Para utilizar una entidad varias veces, es decir, para que ocurran varias instancias de una entidad, es necesario hacerla “aparecer” varias veces con distintas transformaciones. Por ejemplo, en la Figura 3.1 aparecen dos instancias de casa, cada una con una traslación distinta. Cada elemento de la casa está definido también con transformaciones de otros elementos, las cuales deben *concatenarse* con la transformación correspondiente a cada casa para obtener el resultado final. Por lo tanto, las transformaciones desempeñan un papel decisivo en los modelos de la Computación Gráfica, en particular las transformaciones rígidas o lineales.

Como es sabido, la traslación, rotación (alrededor del origen) y escalamiento, conforman una *base funcional* para las transformaciones rígidas en un espacio lineal. Es decir, toda transformación afín o lineal puede ponerse en términos de una aplicación de estas tres operaciones.

Utilizaremos por un momento la notación $p = \begin{bmatrix} x \\ y \end{bmatrix}$ para referirnos a un punto en un espacio de dos dimensiones (ver Apéndices A y B). El resultado de aplicar a p una transformación de escalamiento, por ejemplo, es un punto $p' = \begin{bmatrix} x' \\ y' \end{bmatrix}$ tal que

$$x' = e_x x, \quad y' = e_y y.$$

Es más conveniente una notación matricial:

$$p' = E \cdot p, \text{ es decir, } \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} e_x & 0 \\ 0 & e_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}.$$

Del mismo modo puede representarse para rotar un ángulo θ alrededor del origen:

$$p' = R \cdot p, \text{ es decir, } \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}.$$

Es un inconveniente que no pueda representarse la traslación como preproducto por una matriz:

$$x' = t_x + x, \quad y' = t_y + y.$$

De esa manera se pierden dos propiedades convenientes:

1. Toda transformación de un punto es su preproducto por una matriz cuadrada.
2. La *concatenación* de transformaciones es el preproducto de las matrices respectivas.

3.2.2 Coordenadas homogéneas

Es necesario tener en cuenta que los espacios lineales (Euclídeos) son conjuntos de valores vectoriales cerrados bajo suma y multiplicación escalar (ver Apéndice B). Por lo tanto, los puntos en estos espacios no tienen una representación que los distinga. Una de las confusiones más nocivas en la Computación Gráfica se debe a que la representación de un *punto* y un *vector* en un espacio Euclídeo de n dimensiones puede hacerse con una n -upla de reales, en particular, representando a un punto p con un vector que va del origen a p . Uno de los problemas de esta confusión es que, si bien la suma de vectores es independiente del sistema de coordenadas (al estar los vectores “libres” en el espacio), la “suma” de puntos pasa a depender de la elección particular del origen (ver Figura 3.2).

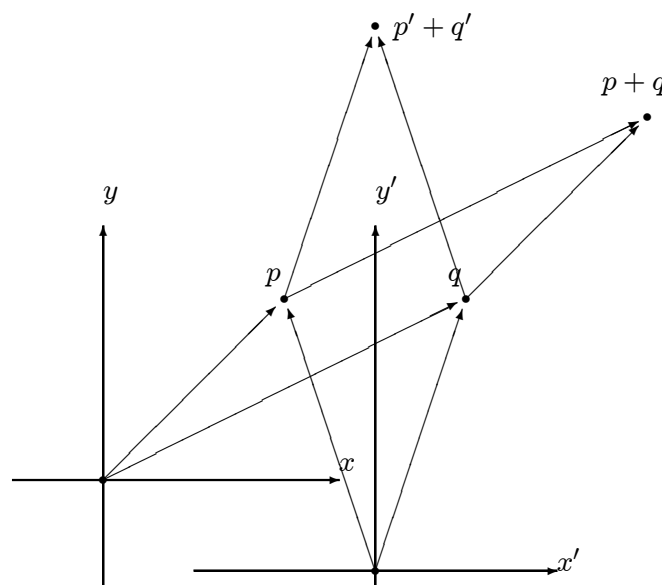


Figura 3.2 La “suma” de puntos está mal definida en un espacio vectorial dado que depende de la elección particular de un sistema de coordenadas.

Sin embargo, en un espacio vectorial de n dimensiones no afín, es posible encontrar subespacios afines de $n - 1$ dimensiones, es decir, subespacios en los cuales la combinación afín de elementos está bien definida. Por ejemplo, en un plano, dados dos puntos p y q , la recta que pasa por dichos puntos es un espacio afín unidimensional que contiene todas las combinaciones afines de p y q . La proyección (intersección) de la suma vectorial de p y q con dicha recta es un punto que está a mitad de camino entre dichos

puntos, independientemente del sistema de coordenadas elegido (ver Figura 3.3). Si realizamos la suma vectorial $p + 9q$ y proyectamos, se obtiene un punto que esta a un 10% de distancia de q y a 90% de p sobre la recta.

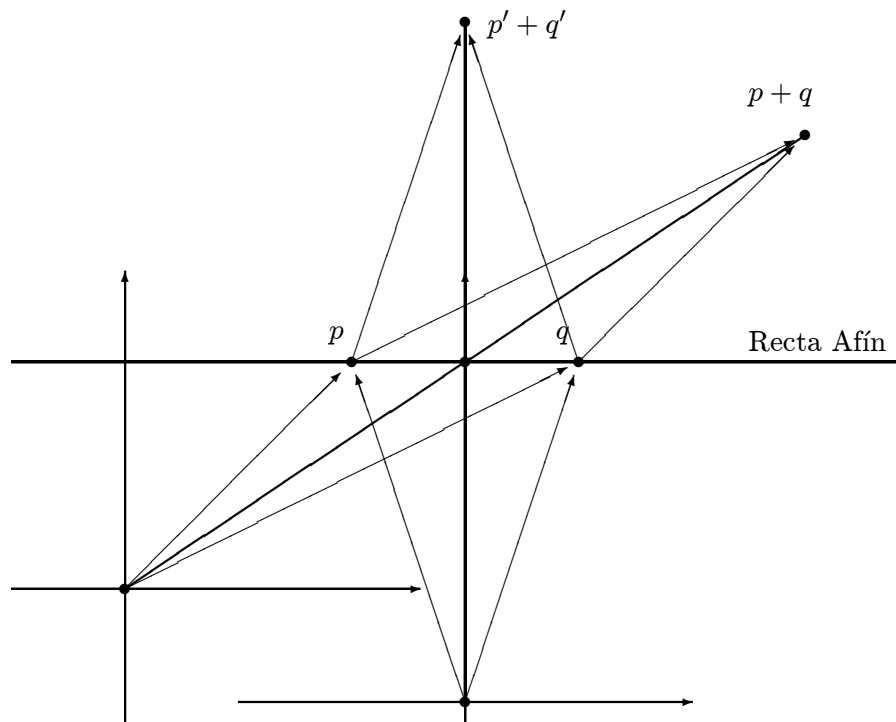


Figura 3.3 La combinación afín de puntos está bien definida en un subespacio afín del espacio vectorial.

Todos los puntos de la recta, como dijimos, pueden representarse como combinación afín de p y q , es decir:

$$r = sp + tq,$$

donde $s + t = 1$. Si además tanto t como s son no negativos, entonces la combinación afín se denomina *convexa* porque el punto resultante pertenece al segmento de recta que va de p a q . Una inspección más detallada nos permite expresar el resultado de una combinación afín del siguiente modo:

$$r = sp + tq = (1 - t)p + tq = p + t(q - p),$$

la cual es una expresión conocida para representar un segmento en forma paramétrica.

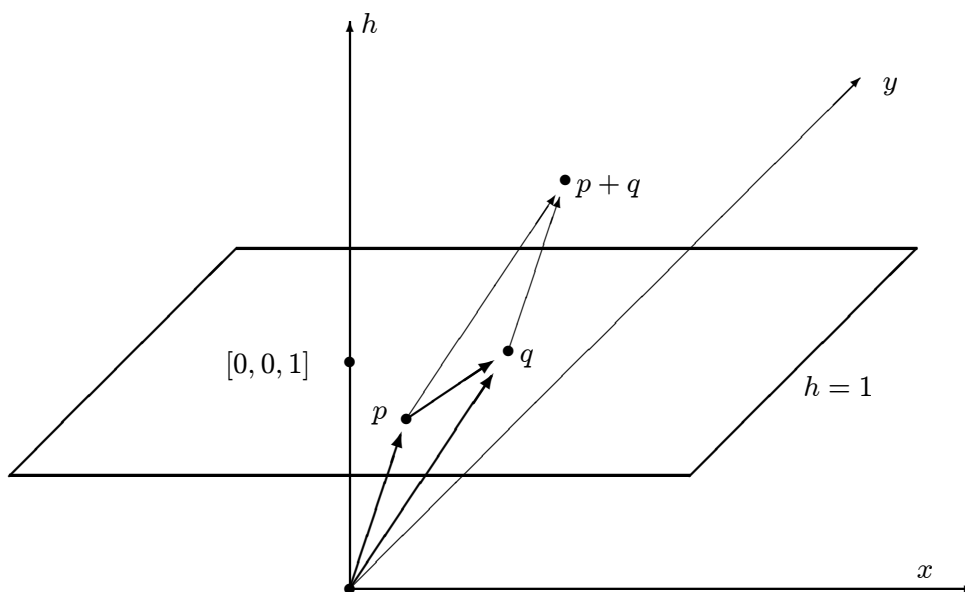


Figura 3.4 Homogenización del espacio R^2 .

Recíprocamente, todo espacio E de n dimensiones puede hacerse afín dentro de un espacio vectorial V de $n + 1$ dimensiones por medio de un procedimiento denominado *homogenización*. Para ello se considera que E está “inmerso” dentro de V como un hiperplano para un valor no trivial (distinto de cero) de una nueva variable h llamada variable de homogenización. En la Figura 3.4 se muestra la homogenización del espacio R^2 como (hiper)plano $h = 1$ de un espacio vectorial R^3 . Todo elemento del espacio homogéneo debe considerarse proyectado sobre el plano $h = 1$. De esa manera, la suma de puntos se transforma en una combinación afín.

Es importante observar que en un espacio afín de estas características la representación de un punto pasa a ser

$$p = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix},$$

y la representación de un vector (como resta de puntos) es

$$v = \begin{bmatrix} a \\ b \\ 0 \end{bmatrix}.$$

Por lo tanto, se supera la confusión entre puntos y vectores. También es importante destacar la equivalencia entre puntos para cualquier valor no negativo de h , dado que

todos los puntos que se proyectan al mismo lugar en el plano $h = 1$ son equivalentes:

$$\begin{bmatrix} xh_1 \\ yh_1 \\ h_1 \end{bmatrix} \sim \begin{bmatrix} xh_2 \\ yh_2 \\ h_2 \end{bmatrix} \sim \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Para pasar del espacio homogéneo $n + 1$ -dimensional al lineal n -dimensional es necesario dividir las primeras n componentes por la última:

$$\begin{bmatrix} x \\ y \\ h \end{bmatrix} \rightsquigarrow \begin{bmatrix} \frac{x}{h} \\ \frac{y}{h} \\ 1 \end{bmatrix}.$$

Este costo adicional de dos cocientes, sin embargo, permite el beneficio de una representación uniforme de las transformaciones. Además, en 3D es necesaria la división para realizar la transformación perspectiva, por lo que el costo en realidad no existe (ver Sección 6.3).

3.2.3 Transformaciones revisitadas

Podemos solucionar el inconveniente mencionado en la subsección 3.2.1 por medio del uso de coordenadas homogéneas. De esa manera, una matriz de escalamiento es:

$$p' = E \cdot p, \text{ es decir, } \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} e_x & 0 & 0 \\ 0 & e_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Del mismo modo puede representarse la rotación

$$p' = R \cdot p, \text{ es decir, } \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix},$$

y también la traslación $x' = t_x + x, y' = t_y + y$:

$$p' = T \cdot p, \text{ es decir, } \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}.$$

Observar que la concatenación no conmuta en general, es decir, es importante el orden en el que se aplican (premultiplican) cada una de las transformaciones.

3.3 Representación Estructurada

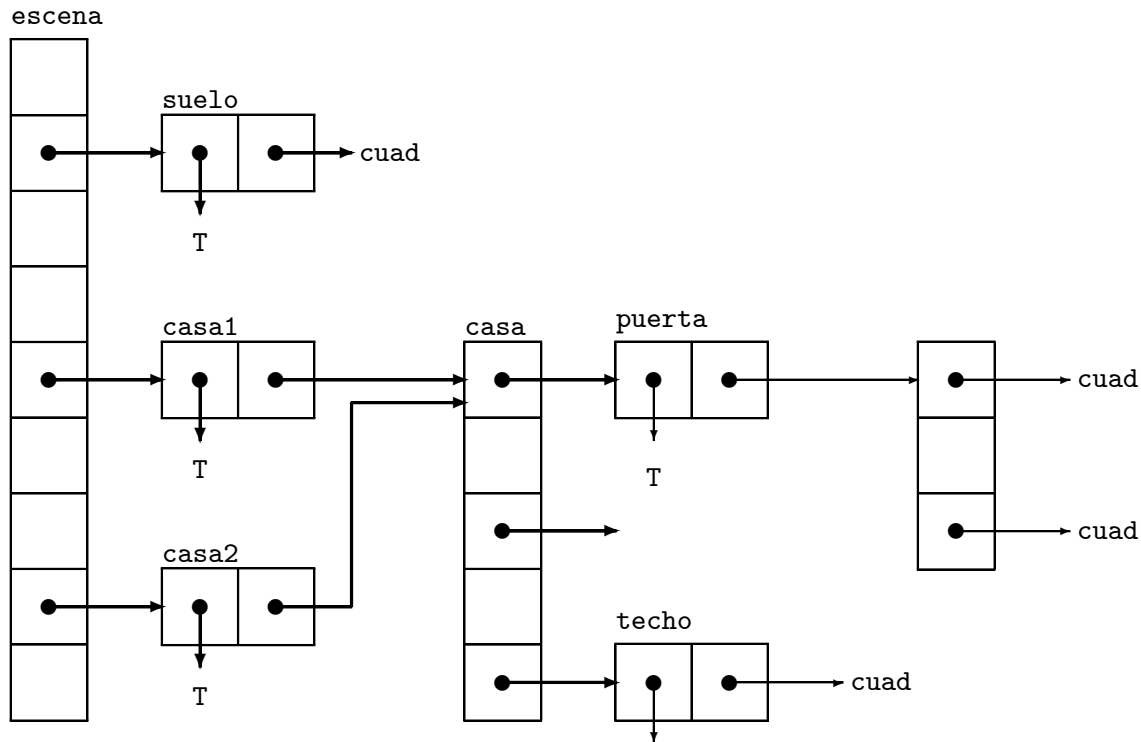


Figura 3.5 Estructuras de datos que representan la escena de la Figura 3.1.

Podemos ahora retornar a nuestro ejemplo de escena mostrado en la Figura 3.1. La representación de objetos se realiza por medio de una estructuración jerárquica en forma de grafo, de modo tal que cada objeto se compone de una definición y una transformación “de instancia” (es decir, dónde debe aparecer el objeto definido), y cada definición es una colección de objetos. En la Figura 3.5 es posible ver parte de la estructura necesaria para definir la escena de la Figura 3.1.

Es importante tener en cuenta que si bien las estructuras son recursivas (un objeto es un par definición-transformación, y una definición es una colección de objetos), las mismas terminan indefectiblemente en objetos primitivos, en este caso, cuadrados. Para graficar la escena, entonces, es necesario recorrer la estructura **escena**. Para cada objeto apuntado por la misma, se guarda la transformación en una pila y se recorre recursivamente. La recursión termina cuando la definición no es una estructura de objetos sino un **cuad**, en cuyo caso se transforman los cuatro vértices del mismo por el

preproducto de todas las matrices apiladas y se grafican los segmentos de recta que los unen.

En la Figura 3.6 se observan las declaraciones de tipos y los procedimientos que implementan estos algoritmos. Es importante notar que la definición de un objeto se realiza por medio de registros variantes, de manera de poder guiar la recursión.

Una forma de hacer este cómputo más eficiente es utilizar una matriz de “transformación corriente”, a la cual se multiplica cada una de las matrices que se van apilando. Cuando se agota una rama de la recursión es necesario desapilar la última transformación y recalcular la transformación corriente. El trozo de código de la Figura 3.6 ilustra una implementación posible de estas ideas, mientras que en la Figura 3.7 se muestra parte de la inicialización de la estructura de información para representar una escena como la de la Figura 3.8.

Es muy importante destacar la flexibilidad que tiene este esquema para el manejo de las escenas. Modificar la transformación de una de las instancias de *casa* modifica *todo* lo que se grafica para dicha casa, y modificar la transformación que define uno de los elementos de la casa modifica cómo se grafica el mismo en *todas* las casas (ver Figura 3.8).

Es necesario tener en cuenta que las matrices homogéneas de transformación normalmente son inversibles en forma muy sencilla (la inversa de la escala es escalar por la inversa, la inversa de la rotación es rotar por el ángulo inverso, y la inversa de la traslación es trasladar por la distancia invertida). Esto hace que la actualización de la transformación corriente pueda hacerse posmultiplicando por la inversa de la matriz que se saca de la pila.

3.4 Windowing y Clipping

La representación estructurada de entidades gráficas permite una cierta abstracción del dispositivo gráfico de salida, independizándonos del tipo de primitivas que soporta y de la tecnología del mismo. Sin embargo, aún queda por resolver el problema del rango de valores de los pixels representables. Puede suceder que los gráficos que se acomodaban bien a la salida en una situación particular, al cambiar de resolución se vean de manera inadecuada. Esto puede empeorar aún más si nuestra aplicación produce una salida gráfica sin utilizar la pantalla completa, por ejemplo en un sistema cuya interfase trabaja por medio de ventanas.

```

type entidades = (cuads,defis);
    cuad = array [0..3] of punto;
    transf = array [0..2] of array [0..2] of real;
    defi = array[0..9] of ^objeto;
    objeto = record    t:^transf;
                      case tipo:entidades of
                          defis:(d:^defi);
                          cuads:(c:^cuad);
                      end;

procedure graf_cuad(o:objeto; at:transf);
var t,t1:transf;
    cu:cuad;
begin
    t:=o.t^;          cu:=o.c^;
    prod_matriz(t,at,t1);
    transformar(cu,t1);
    linea(cu[0],cu[1]);    linea(cu[1],cu[2]);
    linea(cu[2],cu[3]);    linea(cu[3],cu[0]);
end;

procedure graf_obj(o:objeto;at:transf);
var t,t1:transf;
    d:defi;
    i:integer;
begin
    case o.tipo of
        cuads : graf_cuad(o,at);
        defis : begin
            t:=o.t^;      d:=o.d^;
            prod_matriz(t,at,t1);
            i:=0;
            while d[i]<>nil do begin
                graf_obj(d[i]^,t1);
                i:=i+1;
            end;
        end;
    end;
end;

```

Figura 3.6 Estructuras de datos y procedimientos para recorrer una representación estructurada de escenas.

```

procedure graficar;
var   cu:cuad;
      t1, ... :transf;
      o1, ... :objeto;
      casa,sitio,puerta,escena:defi;
begin
  {inicializacion objeto 1}
  o1.t:=@t1;      o1.c:=@cu;
  t1[0][0]:=2.2;  t1[0][1]:=0;    t1[0][2]:=0;
  ...
  {inicializacion objeto 2}
  o2.t:=@t2;      o2.c:=@cu;
  t2[0][0]:=2.2;  t2[0][1]:=0;    t2[0][2]:=0;
  ...
  {definicion de sitio como conteniendo objetos 1 y 2}
  sitio[0]:=@o1;  sitio[1]:=@o2;  sitio[2]:=nil;

  {inicializacion objeto 3: sitio con su transformacion}
  o3.t:=@t3;      o3.d:=@sitio;
  t3[0][0]:=1;    t3[0][1]:=0;    t3[0][2]:=0;
  ...
  {inicializacion objeto 4}
  o4.t:=@t4;      o4.c:=@cu;
  t4[0][0]:=1;    t4[0][1]:=0;    t4[0][2]:=0;
  ...
  {idem objetos 5 y 6}

  {definicion de puerta como conteniendo objetos 4, 5 y 6}
  puerta[0]:=@o4;  puerta[1]:=@o5;  puerta[2]:=@o6;  puerta[3]:=nil;

  {inicializacion objeto 7: puerta con su transformacion}
  o7.t:=@t7;      o7.d:=@puerta;
  t7[0][0]:=1;    t7[0][1]:=0;    t7[0][2]:=0;
  {idem para definicion de ventana}

  {definicion de casa como conteniendo objetos 3, 7 y 9}
  casa[0]:=@o3;   casa[1]:=@o7;    casa[2]:=@o9;  casa[3]:=nil;

  {inicializacion objeto 10: casa con su transformacion}
  o10.t:=@t10;    o10.d:=@casa;
  t10[0][0]:=100; t10[0][1]:=0;    t10[0][2]:=300;
  ...
  {inicializacion objeto 11: casa con otra transformacion}
  o11.t:=@t11;    o10.d:=@casa;
  ...
  {definicion de escena como conteniendo objetos 10 y 11}
  escena[0]:=@o10;  escena[1]:=@o11;  escena[2]:=nil;

  {inicializacion objeto 12: escena con su transformacion}
  o12.t:=@t12;    o12.d:=@escena;
  ...
  graf_obj(o11,identidad);
end;

```

Computación Gráfica 61

Figura 3.7 Parte de la inicialización de las estructuras necesarias para representar la escena de la Figura 3.8.

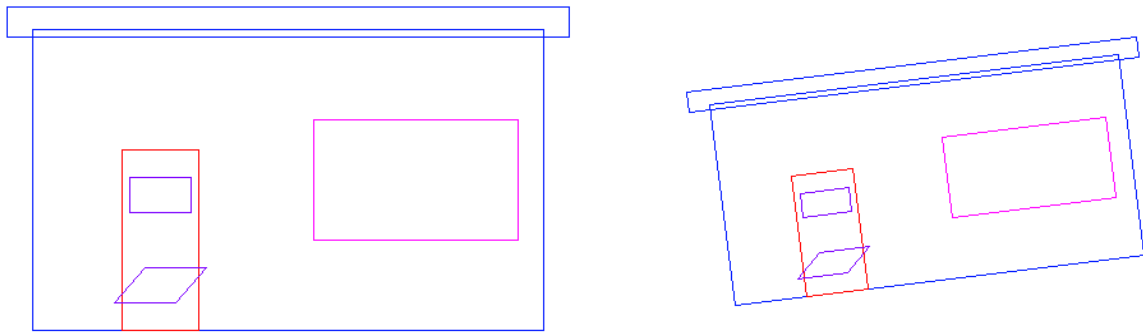


Figura 3.8 Efecto de modificar una instancia de casa y parte de la definición de la ventana.

Para solucionar este problema se define un nuevo sistema de coordenadas, denominado sistema de coordenadas del mundo, del cual se representará gráficamente un segmento denominado ventana o *window*. Dicho segmento es puesto en correspondencia con un sector de la pantalla (subconjunto del sistema de coordenadas físico) denominado *viewport*. La operación de transformar las entidades gráficas del window al viewport se denomina *windowing*, y la de eliminar la graficación de entidades que caen fuera del viewport se denomina *clipping*.

3.4.1 Windowing

Las aplicaciones en Computación Gráfica trabajan en sistemas de coordenadas adecuados para cada caso particular. Por ejemplo, un histograma meteorológico puede estar en un sistema de coordenadas en milímetros de lluvia vs. tiempo. El dispositivo de salida, por su parte, trabaja en su propio sistema de coordenadas físico. Para lograr una independencia de dispositivo, es necesario trabajar en un espacio que tenga un sistema de coordenadas normalizado. Por dicha razón se define el sistema de coordenadas del mundo, dentro del cual la porción visible de la escena es la contenida en la ventana (en algunos casos se sugiere utilizar una ventana normalizada 0.0—1.0 en x e y , denominada NDC, Normalised Device Coordinate, garantizando un estándar gráfico).

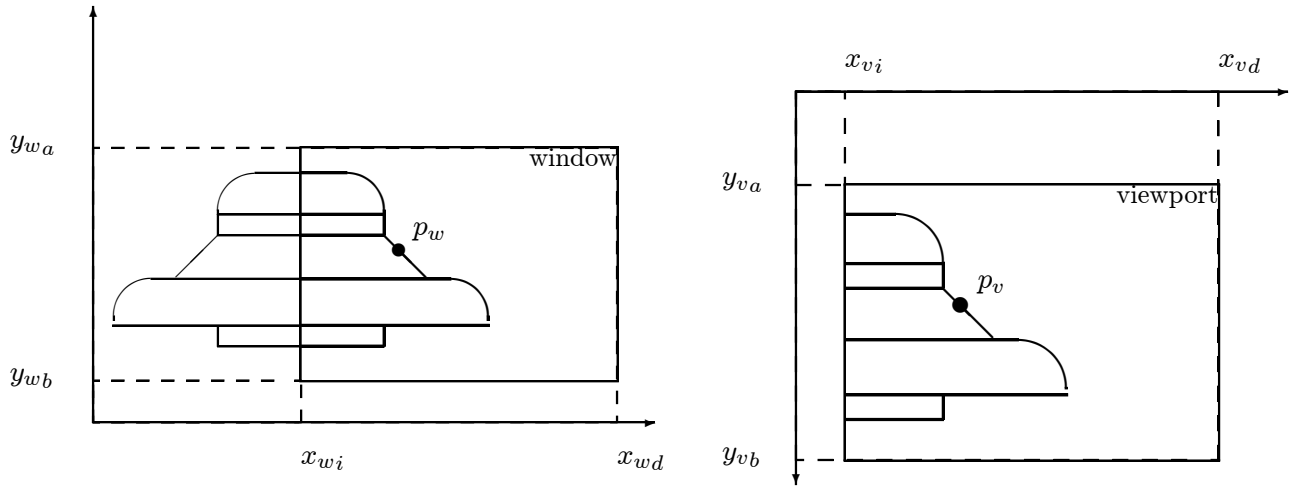


Figura 3.9 Elementos que definen la transformación de windowing.

Sean x_{wi} y x_{wd} los límites en x (izquierdo y derecho respectivamente), y y_{wa} y y_{wb} los límites en y (superior e inferior respectivamente) del window. Dicha ventana debe ser mapeada al subconjunto de la imagen denominado *viewport* definido en el sistema de coordenadas de la pantalla (ver Figura 3.9), cuyos límites son x_{vi} y x_{vd} para x (izquierdo y derecho respectivamente), y y_{va} y y_{vb} para y (superior e inferior respectivamente). Por lo tanto, es necesario encontrar los parámetros de una transformación de windowing que lleva un punto p_w del window a un punto p_v del viewport.

Podemos encontrar dichos parámetros planteando semejanzas entre segmentos. Por ejemplo, para la coordenada x de un punto p_w en el window es posible encontrar

$$\frac{x_w - x_{wi}}{x_{wd} - x_{wi}} = \frac{x_v - x_{vi}}{x_{vd} - x_{vi}},$$

de donde se sigue

$$x_v = \frac{(x_w - x_{wi})}{(x_{wd} - x_{wi})}(x_{vd} - x_{vi}) + x_{vi}.$$

Llamando a al factor constante $\frac{(x_{vd} - x_{vi})}{(x_{wd} - x_{wi})}$ obtenemos

$$x_v = (x_w - x_{wi})a + x_{vi} = ax_w + b,$$

donde $b = x_{vi} - ax_{wi}$.

De haber utilizado el sistema NDC, la expresión se simplifica a

$$x_v = x_w(x_{vd} - x_{vi}) + x_{vi}.$$

Por último, es necesario discretizar el punto obtenido para transformarlo en pixel (por medio del `round` de x_v).

Es necesario enfatizar un punto de gran importancia en el análisis de la eficiencia de los algoritmos utilizados en Computación Gráfica. Todas las primitivas que hemos utilizado son invariantes frente a las transformaciones que se utilizan. Esto garantiza que es posible aplicar todas las transformaciones (`windowing` incluida) a los puntos que definen a una entidad gráfica, y luego aplicar la discretización sobre los puntos transformados. Es decir, conmutan las operaciones de transformar y de discretizar. De no haber sucedido así, no hubiésemos tenido otro remedio que discretizar en el espacio de la escena, y luego transformar cada punto de la discretización, lo cual es evidentemente de un costo totalmente inaceptable.

3.4.2 Clipping

Como ya mencionáramos, al aplicar `windowing` es necesario “recortar” la graficación de aquellas partes de la escena que son transformadas a lugares que quedan fuera del viewport. La manera más directa de realizar esta tarea es modificar los algoritmos de discretización de primitivas para que utilicen las coordenadas del viewport (por ejemplo, definidas con variables globales) y grafiquen un pixel sólo si está dentro del mismo.

```
procedure linea_clip(p0,p1:punto;c:col);
...
if (xvi<=x) and (x<=xvd) and { xvi, xvd, yva, yvb }
    (yva<=y) and (y<=yvb) { son variables globales }
    then putpixel(x,y,col);
...
end;
```

Es muy probable que un mecanismo de este estilo esté implementado dentro de la tarjeta gráfica para proteger el acceso a lugares de memoria inadecuados cuando se reclama un `putpixel` fuera del área de pantalla. De todas maneras, es fácil ver que si gran parte de nuestra escena está fuera del window, un algoritmo por el estilo perderá cantidades de tiempo enormes chequeando pixel por pixel de entidades gráficas que caen fuera del área graficable.

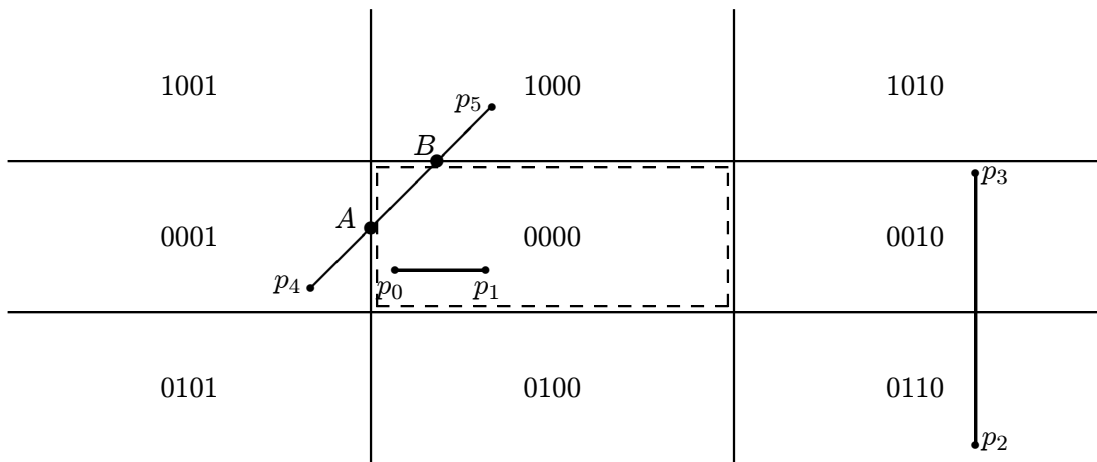


Figura 3.10 *Cómo se etiquetan los extremos de los segmentos en el algoritmo Cohen-Sutherland para clipping de segmentos.*

Una técnica mucho más práctica fue propuesta por Cohen y Sutherland para el clipping de segmentos de rectas. La filosofía subyacente es clasificar los segmentos como totalmente visibles o totalmente invisibles (en ambos casos no hay que aplicar clipping!). Aquellos que escapan a la clasificación son subdivididos, y luego se clasifican los subsegmentos resultantes.

La clave del método está en etiquetar los píxeles extremos de los segmentos con un cuádruplo de booleanos. Cada uno de ellos es el resultado de preguntar si dicho extremo está fuera del viewport por exceder alguno de sus cuatro límites, en un orden arbitrario. En la Figura 3.10, por ejemplo, se observa cómo quedan etiquetados los píxeles extremos en función del área de pantalla en que caen, al ser etiquetados en el orden originalmente propuesto por los autores (yva, yvb, xvd, xvi). El viewport está delimitado con una caja punteada.

Podemos ver que el segmento de recta que va de p_0 a p_1 cae totalmente dentro del viewport, y por lo tanto cada píxel de su discretización puede graficarse en forma incondicional. En cambio, el segmento que va de p_2 a p_3 cae totalmente fuera del área graficable, y por lo tanto ninguno de sus píxel debe graficarse. Una situación más compleja ocurre con el segmento que va de p_4 a p_5 , dado que tiene partes graficables y partes no graficables.

Las etiquetas de los pixels extremos del segmento permiten clasificar estos tres casos de manera sencilla:

- Si el *or* de ambos extremos es cero, el segmento es totalmente graficable.
- Si el *and* de ambos extremos no es cero, el segmento no es graficable.
- En todo otro caso el segmento puede tener partes graficables y partes no graficables.

En el segmento de recta que va de p_0 a p_1 , ambos extremos están etiquetados con 0000 y por lo tanto estamos en el primer caso. En cambio, en el segmento que va de p_2 a p_3 tiene etiquetas 0010 y 0110 y por lo tanto estamos en el segundo caso. Con el segmento que va de p_4 a p_5 tenemos etiquetas 0001 y 1000, cuyo *or* no es cero y cuyo *and* es cero, por lo que estamos en el tercer caso. Aquí es necesario subdividir el segmento y aplicar recursivamente a cada subsegmento.

Para subdividir un segmento es posible encontrar la intersección del mismo con alguno de los cuatro “ejes” del viewport, dado que todo segmento parcialmente visible intersecta por lo menos a uno de dichos ejes. Por ejemplo, se subdivide el segmento en el punto A , obteniéndose un segmento de p_4 a A (totalmente fuera del viewport) y otro segmento de A a p_5 parcialmente graficable, que se puede subdividir en B en dos segmentos, uno totalmente fuera del viewport y otro totalmente dentro. Otra estrategia, que algunos proclaman más rápida, consiste en subdividir los segmentos por su punto medio, el cual es sencillo de encontrar como promedio de sus extremos. Esta forma de subdividir es análoga a una búsqueda binaria, y por lo tanto la cantidad de subdivisiones es del orden del logaritmo de la longitud del segmento de recta.

3.4.3 La “tubería” de procesos gráficos

Corresponde realizar en este punto una pequeña revisión del capítulo para tener una visión integrada de los distintos procesos involucrados. El punto de vista más importante para tener en cuenta es que se definen las entidades gráficas como compuestas por primitivas discretizables, cuyos parámetros geométricos son puntos. Por ejemplo, describimos una escena como un conjunto de cuadrados transformados, y cada cuadrado es un conjunto de segmentos de recta uniendo los vértices del mismo. Por lo tanto, la tarea esencial del sistema gráfico es indicarle al mecanismo de discretización la ubicación de dichos puntos.

De esa manera, podemos pensar que los procesos del sistema gráfico se ubican en una “tubería”, que va tomando puntos de la definición de las entidades gráficas, los va transformando por las diversas transformaciones que ocurren, y los deposita en los algoritmos de discretización de primitivas. A dichos procesos se agrega lo visto en esta sección, es decir la transformación de windowing que relaciona el sistema de coordenadas del mundo con el sistema de coordenadas de pantalla, y el clipping de las partes no graficables de la escena.

También es necesario agregar una transformación más, denominada transformación del mundo, que lleva a los objetos del sistema de coordenadas de la escena al sistema de coordenadas del mundo en el cual está definido el window (probablemente en NDC). Esta última transformación es también importante porque permite alterar la ubicación y tamaño de *toda* la escena, para realizar efectos de zooming, desplazamiento, rotación, etc. En la Figura 3.11 podemos ver cómo se estructuran todos estos procesos.

3.5 Implementación de Bibliotecas Gráficas

El propósito de esta sección es brindar algunos conceptos introductorios relacionados con la implementación de bibliotecas (o “librerías”) gráficas. En la actualidad, la biblioteca gráfica casi excluyente es OpenGL, de la cual nos ocuparemos desde el punto de vista del usuario en el Capítulo 6.

Una biblioteca gráfica es una pieza de software destinada a facilitar el desarrollo de aplicaciones sobre un conjunto de dispositivos gráficos determinados. Un ejemplo sería desarrollar sobre una PC los procesos gráficos descritos hasta ahora, sin tener software gráfico específico. Para ello tenemos que solucionar dos problemas. Primero es necesaria la interacción con el hardware, por ejemplo por medio del uso del sistema de interrupciones del sistema operativo. Segundo, es necesario definir un conjunto de operaciones que permitan al usuario manejar el paquete.

Podemos especificar un conjunto de requisitos para un sistema de estas características.

Simplicidad: La interfase con el usuario debe ser amigable, fácil de aprender, y que utilice un lenguaje sencillo. Una de las alternativas más obvias es utilizar una interfase de ventanas con menús descolgables.

Consistencia: Se espera que un sistema se comporte de una manera consistente y predecible. Una forma de lograr dicho comportamiento es a través del uso de

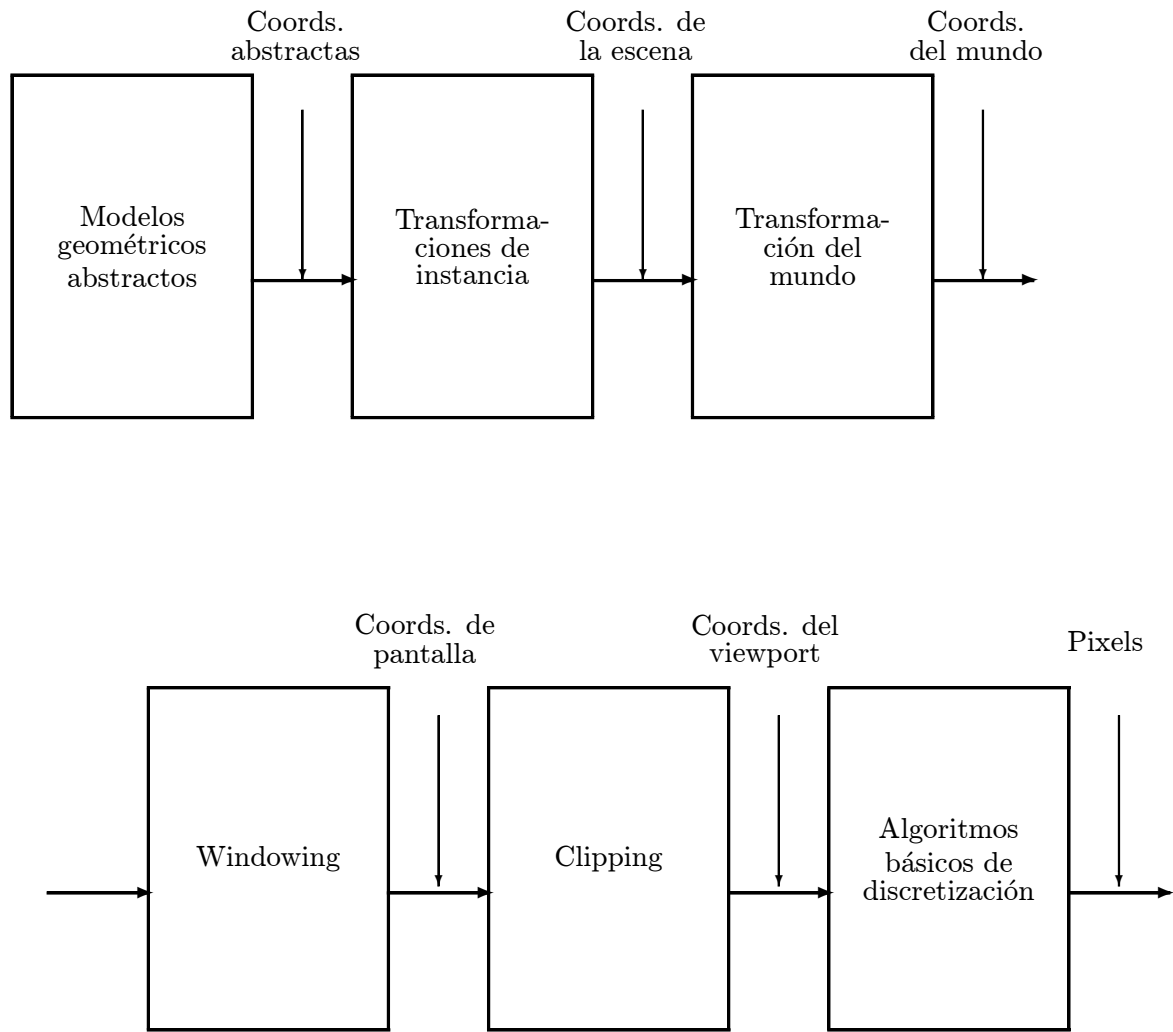


Figura 3.11 Estructura de la "tubería" de procesos gráficos.

modelos conceptuales. También se espera que no ocurran excepciones, es decir, funciones que actúan incorrectamente en algunos casos particulares.

Complejidad: Es necesario encontrar un conjunto razonablemente pequeño de funciones que manejen convenientemente un rango amplio de aplicaciones. También es necesario que no ocurran omisiones irritantes que deban ser suplidas desde el nivel de la aplicación.

Integridad: Los programadores de aplicaciones y los usuarios normalmente tienen una extraordinaria capacidad para vulnerar la integridad de los sistemas. Por lo tanto debe existir un buen manejo de errores operativos y funcionales, normalmente impidiendo el acceso o la manipulación indebida del sistema. Los mensajes de error que se generan en tiempo de ejecución deben ser útiles para los usuarios. Debe existir una buena documentación que sea comprensible para un usuario promedio.

Economía: En tiempo de ejecución, tamaño y recursos utilizados.

Podemos también describir brevemente las capacidades de un paquete gráfico a partir de una clasificación del tipo de funciones y procedimientos que implementa:

Procedimientos de seteo: Se utilizan para inicializar o cambiar el estado de las variables globales del sistema.

- **initgraph:** Se utiliza para inicializar el paquete a un determinado modo gráfico.
- **cls:** Para limpiar el viewport.
- **color:** Para setear el color corriente con el que se graficará en adelante.
- **move:** Para desplazar el cursor gráfico.

Funciones primitivas gráficas: Son las entidades gráficas más sencillas, en función de las cuales se estructuran todas las demás. Por ejemplo las funciones

- **line:** Se utiliza para graficar un segmento de recta con el color corriente, desde la posición del cursor hasta un lugar determinado del viewport.
- **point:** Para prender un pixel con el color corriente y en el lugar determinado por el cursor.
- **circle:** Para trazar una circunsferencia del color corriente, centrada en el cursor y de un radio determinado.
- **poly:** Para trazar una poligonal.

Funciones de transformación: Usualmente para poder modificar la transformación del mundo. Es más intuitivo referirse a dicha matriz a través de funciones `rotate`, `translate`, `scale`.

Procedimientos de graficación: Muchas aplicaciones gráficas se utilizan para la generación de diagramas, histogramas y otro tipo de representación de datos. Para ello es muy útil proveer procedimientos como `graphpaper`, `graphdata`, `bardata`, que grafican en el viewport el contenido de determinadas estructuras de datos utilizando escalas y métodos establecidos.

Funciones de windowing: Es importante también que se pueda programar desde el nivel de aplicación el uso de ventanas con clipping. Para ello es necesaria la implementación de funciones `setwindow`, `setviewport` que permitan asignar desde la aplicación los parámetros correspondientes al sistema gráfico.

Modelos y estructuras: Es importante que desde la aplicación se puedan definir objetos complejos como estructuras de otros objetos. Para ello es necesario tener procedimientos como `define .. as record ... end` que permiten describir una entidad gráfica como la unión de entidades más sencillas.

3.6 Ejercicios

1. Modificar los algoritmos del capítulo anterior para utilizar los tipos de datos definidos en este capítulo.
2. Modificar los mismos algoritmos para utilizar las primitivas provistas por el compilador. Comparar los resultados.
3. Implementar la representación de una escena con entidades gráficas estructuradas. Graficar la misma modificando las transformaciones de instancia y la transformación del mundo.
4. Implementar las rutinas de windowing y clipping. Graficar las escenas del ejercicio anterior en una ventana que muestre parte de la escena. Graficar las mismas escenas con toda la pantalla como viewport, sin y con clipping. Evaluar los tiempos.

3.7 Bibliografía recomendada

El manejo de las transformaciones en 2D y las coordenadas homogéneas puede consultarse en el Capítulo 4 y el apéndice II del libro de Newman y Sproull [66], aunque utiliza los puntos como vector fila, y las transformaciones como posmultiplicación. Es también recomendable el Capítulo 5 del libro de Foley et. al. [33].

La descripción de los modelos estructurados de entidades gráficas puede consultarse en el Capítulo 9 del Newman-Sproull, y algunos aspectos avanzados en el Capítulo 2 del libro de Giloi [40]. Una descripción del mismo tema, pero en 3D, figura en el Capítulo 7 del Foley.

El tema de windowing y clipping de segmentos de recta puede leerse en el Capítulo 5 del Newman-Sproull y en el Capítulo 3 del Foley. Blinn discute en [13] la descripción y ventajas del windowing sobre viewports no cuadrados (los cuales son los más usuales, sin duda). El clipping de entidades más complejas, por ejemplo polígonos, caracteres o círculos, puede ser bastante difícil. Recomendamos la lectura de [79] para el clipping de polígonos no convexos, y de [11] para el clipping de entidades en general.

La implementación de bibliotecas gráficas (*graphic packages*) está adecuadamente tratado en el Capítulo 6 del Newman-Sproull. Otros aspectos más detallados pueden consultarse en el Capítulo 8 del libro de Giloi.