Transformación de vista

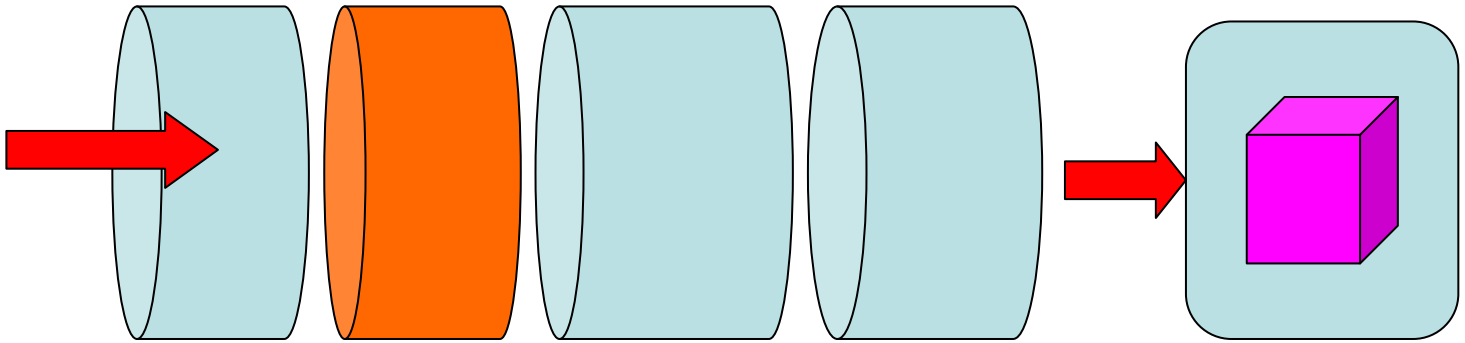
(X1,Y1, Z1)

(X2, Y2, Z2)

(X3,Y3, Z3)…

geometría

Coordenadas del objeto (en punto flotante)



T. Afines **T. Vista** **Recorte y T. Proyección** **Dibujado de primitivas (Rasterización)**

Coordenadas de Vista

(en punto flotante)

Transformación de Vista en 2D

- Se encuentra situada en la parte del procesamiento geométrico de los datos.

- Constituye la interfaz entre el tratamiento de datos en coma flotante (geometría) y el tratamiento en enteros (operaciones con pixeles).

Justificación de la Transformación

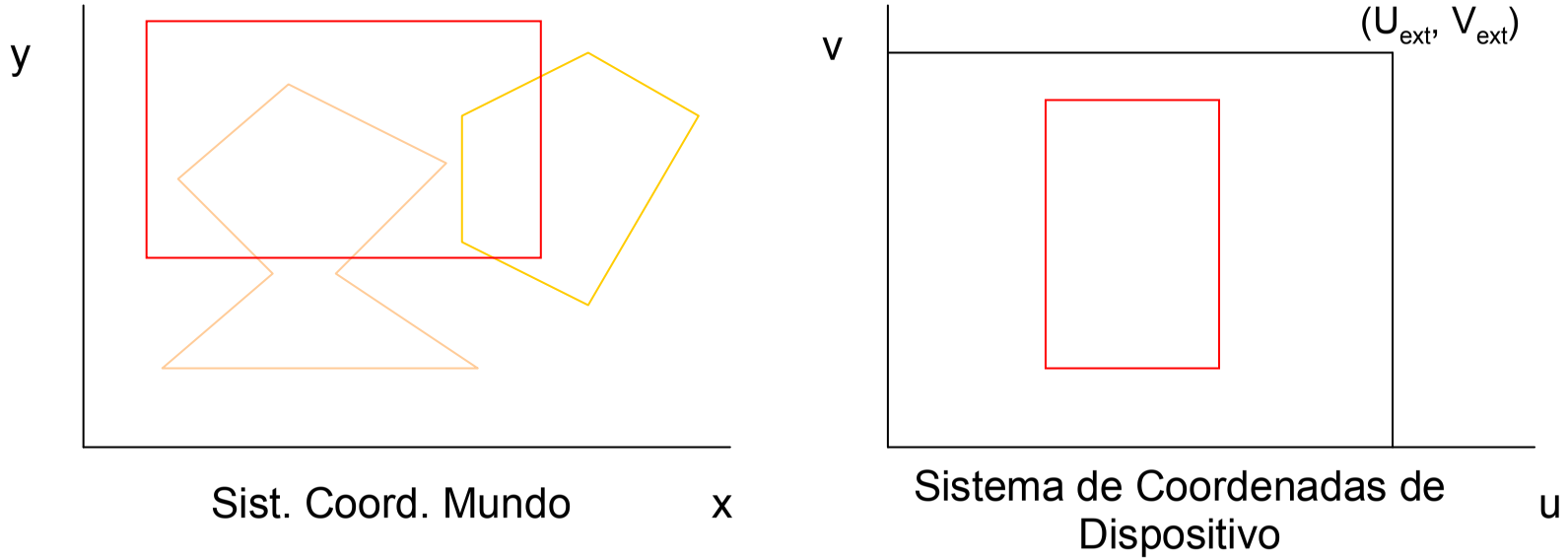
- Los datos geométricos están referidos al "Sistema de Coordenadas del Mundo" que tiene unas escalas adecuadas a la escena que se quiere representar (Kilómetros, Años-Luz, milímetros, micras...).

- La escena se representa en un dispositivo de visualización (monitor, impresora,...) 2D en "Coordenadas de Dispositivo" de tipo entero.

- La idea de la transformación Ventana-puerto de Vista consiste en realizar una correspondencia entre los puntos geométricos que deben ser dibujados y los píxeles del dispositivo de salida.

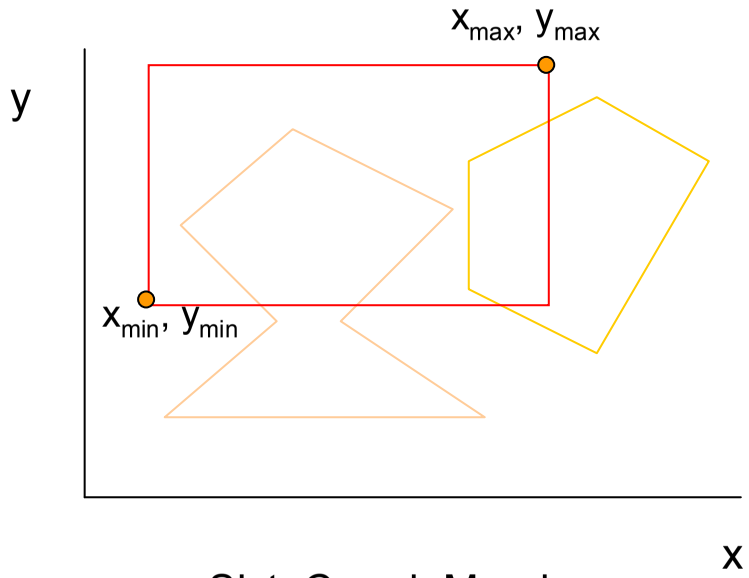
Descripción de la Transformación

- Definamos en el sistema de coordenadas del mundo una ventana llamada **“Ventana del Mundo**”. Igualmente definamos otra ventana sobre un sistema de coordenadas de dispositivo que llamaremos **“Puerto de Vista”.** Este último sistema es una abstracción de la superficie del dispositivo de visualización, y asumimos que sus valores tienen un rango.



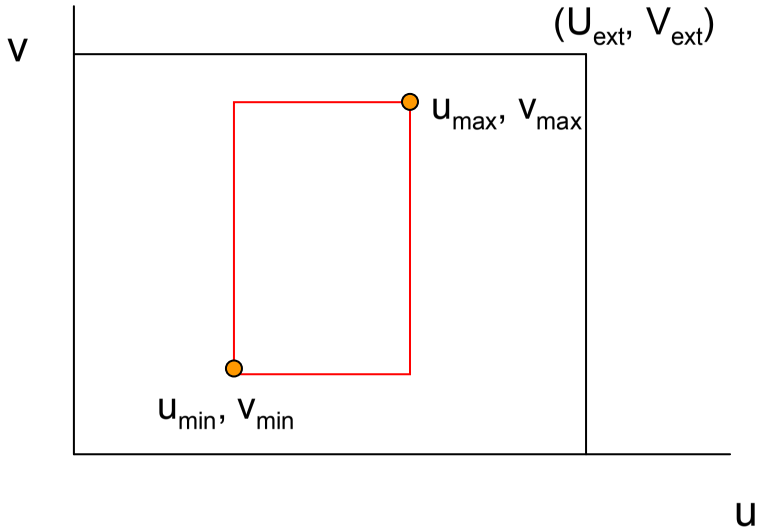
Descripción de la Transformación

- Sean (x_{min}, y_{min}) y (x_{max}, y_{max}) los extremos de la ventana del mundo.
- Sean (u_{min}, v_{min}) y (u_{max}, v_{max}) los extremos del Puerto de Vista.



Sist. Coord. Mundo

(coord. Punto flotante)

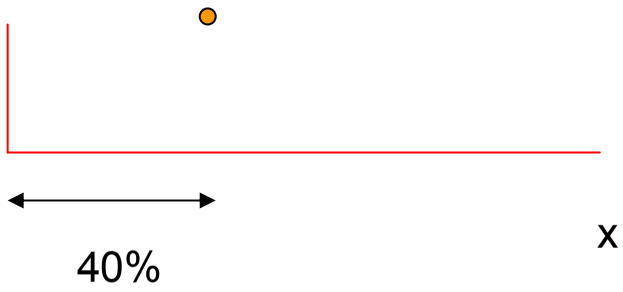


Sistema de Coordenadas de Dispositivo

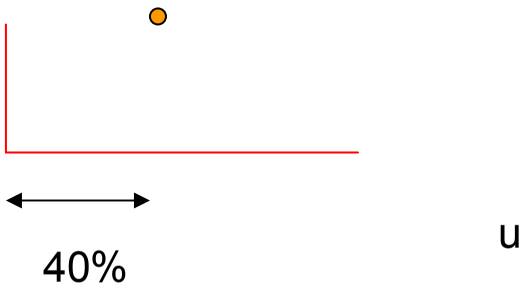
Descripción de la Transformación

- Sin perder generalidad **suponemos que las ventanas están alineadas con los sistemas de referencia.**

- La transformación debe mantener la proporcionalidad de los puntos geométricos:



Ventana del Mundo



Puerto de Vista

Descripción de la Transformación

- Por lo tanto la transformación debe ser lineal

$$u = A\,x + C$$

$$v = B\,y + D$$

- Donde A y B escalan los puntos. C y D los trasladan.

- Imponiendo las proporcionalidades en ambos ejes:

$$\frac{u - u_{min}}{u_{max} - u_{min}} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

$$\frac{v - v_{min}}{v_{max} - v_{min}} = \frac{y - y_{min}}{y_{max} - y_{min}}$$

Descripción de la Transformación

- Despejando:

$$u = \frac{(u_{max} - u_{min})\,(x - x_{min})}{x_{max} - x_{min}} + u_{min} = \frac{(u_{max} - u_{min})}{x_{max} - x_{min}}\,x + u_{min} - \frac{(u_{max} - u_{min})}{x_{max} - x_{min}}\,x_{min}$$

- Con lo que:

$$A = \frac{(u_{max} - u_{min})}{x_{max} - x_{min}} \qquad C = u_{min} - A\,x_{min}$$

Descripción de la Transformación

- Análogamente se hace con la componente v del puerto de vista:

$$v = \frac{(v_{max} - v_{min})\,(y - y_{min})}{y_{max} - y_{min}} + v_{min} = \frac{(v_{max} - v_{min})}{y_{max} - y_{min}}\,y + v_{min} - \frac{(v_{max} - v_{min})}{y_{max} - y_{min}}\,y_{min}$$

- Con lo que:

$$B = \frac{(v_{max} - v_{min})}{y_{max} - y_{min}} \qquad D = v_{min} - B\,y_{min}$$

Propiedades de la transformación

- Los puntos interiores de la ventana del mundo se corresponden con puntos interiores del puerto de vista. Análogamente ocurre con los exteriores.

- $P = (x,y) = (x_{min} - k, y_{max} + k) / k > 0 \Rightarrow (u,v) = (u_{min} - k \frac{u_{max} - u_{ma}}{x_{max} - x_{ma}}, v_{max} + k \frac{v_{max} - v_{ma}}{y_{max} - y_{ma}})$

- Igualmente ocurre con los puntos representativos:

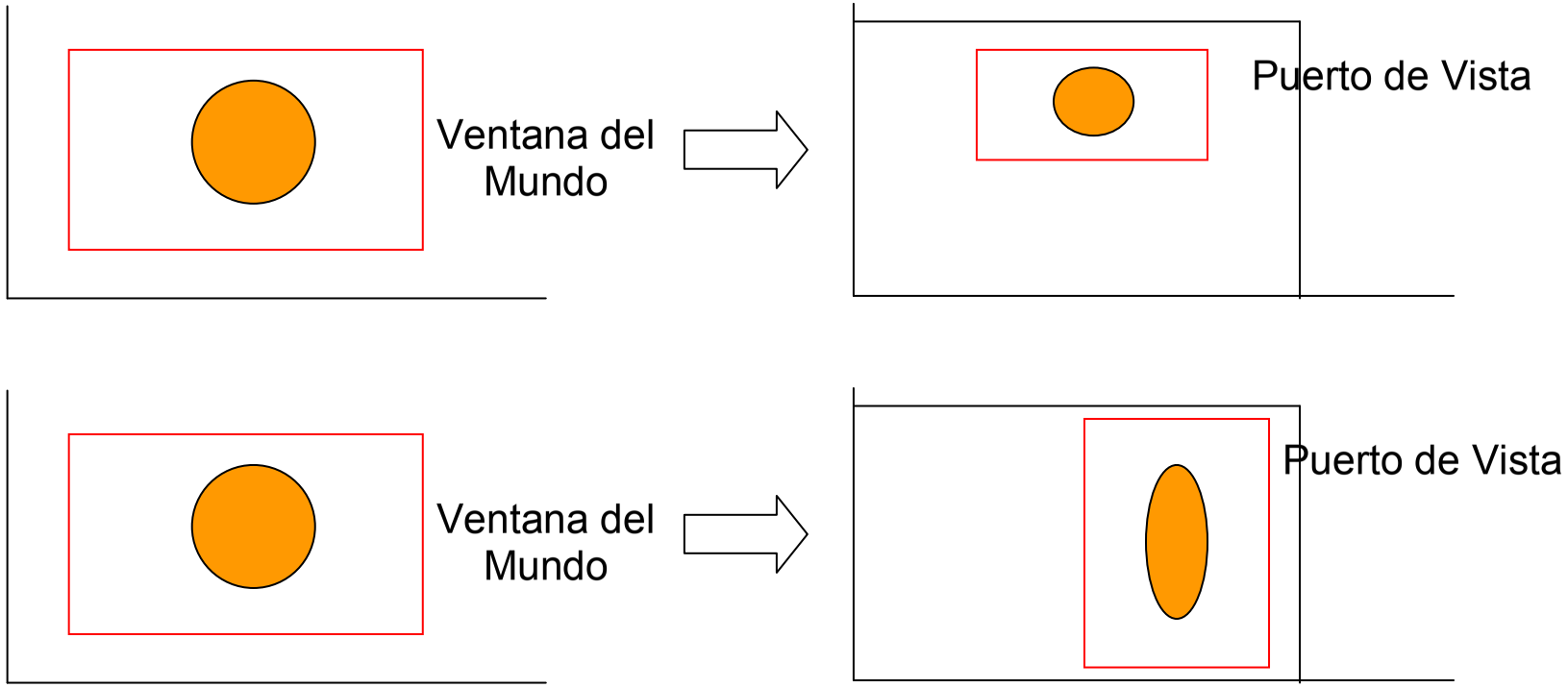
- $p = (x,y) = (x_{min}, y_{min}) \Rightarrow (u,v) = (u_{min}, v_{min})$

- $p = (x,y) = (x_{max}, y_{max}) \Rightarrow (u,v) = (u_{max}, v_{max})$

- $p = (x,y) = ((x_{min} + x_{max}) / 2, (y_{min} + y_{max}) / 2) \Rightarrow (u,v) = ((u_{max} + u_{min}) / 2, (v_{max} + v_{min}) / 2)$

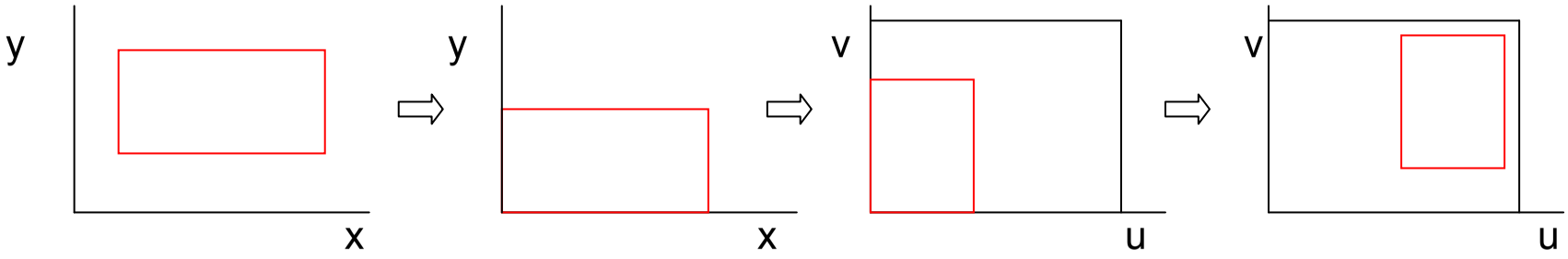
Propiedades de la transformación

- Si la razón entre la anchura y la altura de la ventana del mundo y el puerto de vista es la misma, no habrá distorsión en la imagen, en otro caso sí la habrá.



Expresión matricial

- Podemos entender esta transformación como una composición de transformaciones de escalado y traslación para convertir una ventana en la otra.



Ventana del mundo

Traslación al origen

Escalado a la s dimensiones del Puerto de Vista

Traslación al destino en el Sist. C. dispositivo

$$M_{w->v} = T(u_{min}, v_{min}) \cdot S(\frac{u_{max} - u_{min}}{x_{max} - x_{min}}, \frac{v_{max} - v_{min}}{y_{max} - y_{min}}) \cdot T(- x_{min}, -y_{min})$$

Expresión matricial

$$M_{w->v} = \begin{bmatrix} 1 & 0 & u_{min} \\ 0 & 1 & v_{min} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \frac{u_{max} - u_{min}}{x_{max} - x_{min}} & 0 & 0 \\ 0 & \frac{v_{max} - v_{min}}{y_{max} - y_{min}} & 0 \\ 0 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & - x_{min} \\ 0 & 1 & -y_{min} \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \frac{u_{max} - u_{min}}{x_{max} - x_{min}} & 0 & \frac{u_{max} - u_{min}}{x_{max} - x_{min}} & -x_{min} + u_{min} \\ 0 & \frac{v_{max} - v_{min}}{y_{max} - y_{min}} & \frac{v_{max} - v_{min}}{y_{max} - y_{min}} & -y_{min} + v_{min} \\ 0 & 0 & 1 \end{bmatrix}$$

Decisiones de Implementación: OpenGl

- OpenGl orienta su salida hacia una pantalla de monitor. El puerto de vista se especifica en pixeles de pantalla.

  glViewport(int izda, int abajo, int inc_u, int inc_v)

- Internamente sí hay un cambio de la ventana del mundo a un sistema de coordenadas canónico

  (x_{norm} , y_{norm}) $\quad -1 \leq x_{norm} , y_{norm} \leq 1$

  aunque es “transparente” al programador. Con los datos de glViewport, se define un origen en el centro del puerto de vista (Or_u, Or_v)

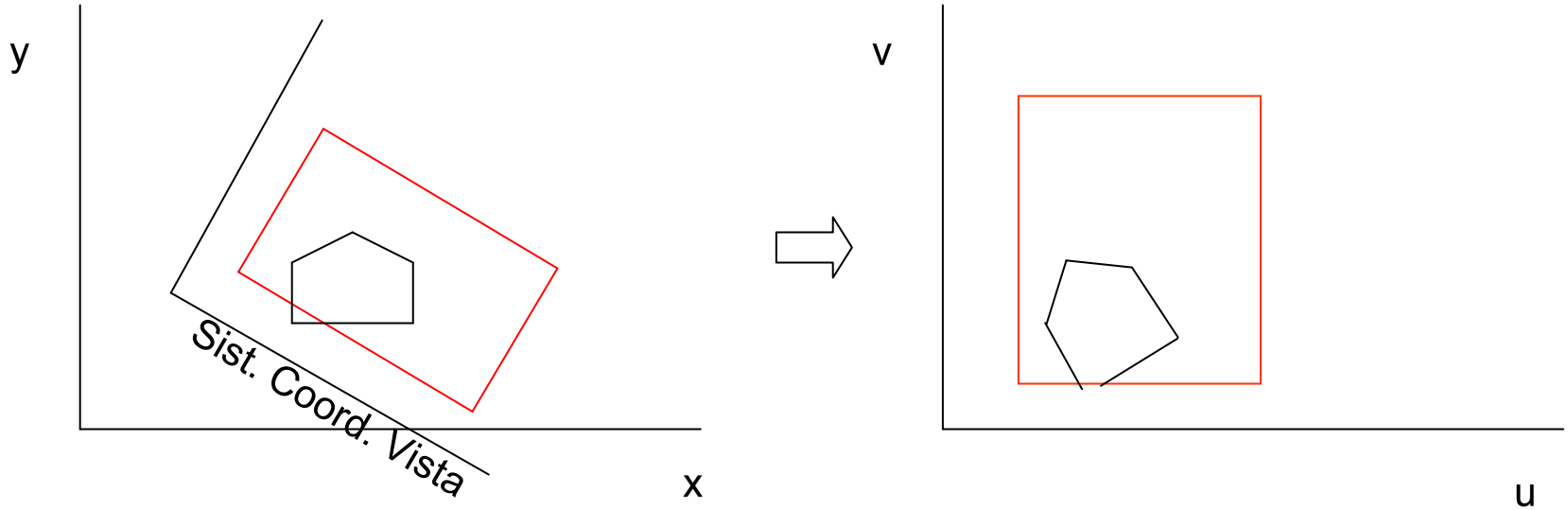
- Se establece la correspondencia :

  $x_{disp} = (inc\_u / 2) x_{norm} + Or\_u$

  $y_{disp} = (inc\_v / 2) y_{norm} + Or\_v$

Cuestiones asociadas

- La transformación de vista permite tratar a la Ventana del Mundo como una cámara. Se puede observar la escena con una orientación arbitraria en el plano modificando la orientación de la Ventana del Mundo con esta transformación.
- En este caso la Ventana del Mundo se alinea con el Sistema de Coordenadas de vista.



Cuestiones asociadas (2)

El sistema de coordenadas de vista no tiene más trascendencia en 2D pero es ineludible en el proceso de visualización en 3D. Ya que:

+ En él se sitúa el punto de vista de la escena
+ En él se aplica la proyección geométrica

Manejo de las ventanas de la transformación de Vista 2D

- Efecto de Zoom.

Podemos conseguir este efecto variando uniformemente las dimensiones de la Ventana del Mundo manteniendo igual las dimensiones del puerto de vista.

- Efecto de Mosaico.

Podemos hacer un mosaico en nuestra superficie de visualización usando como tesela el Puerto de Vista, determinando un tamaño adecuado de éste y trasladándolo por la superficie de visualización sin refrescar ésta.

Estos Usos con OpenGl

- En la biblioteca OpenGl existen dos instrucciones para configurar las ventanas de la transformación:

void gluOrtho2D(Gldouble left, Gldouble right, Gldouble bottom, Gldouble top);

Configura la Ventana del Mundo

void glViewport(Glint x, Glint y, Glint width, Glint height);

Configura el Puerto de Vista

- Se ha preparado una clase *dibujadino* que dibuja un dinosaurio a partir de los datos geométricos en un fichero. Estos datos son puntos que definen líneas poligonales en el rango de $0 \leq x \leq 640$, $0 \leq y \leq 480$
- Esta clase deriva de otra llamada lienzo que configura la Ventana del Mundo y el puerto de Vista

Ejemplo

```
class lienzo{
protected:

    //dimensiones del área de dibujo
    int altura;
    int anchura;
    //origen de la ventana respecto de la pantalla
    int origenx;
    int origeny;

public:

     //constructor que inicializa parámetros de configuración
     lienzo(int alt, int anch, int orx, int ory);

     //configuración de la ventan del mundo
     void confVentanaMundo(float izda, float dcha, float abajo, float
                           arriba);

     //configuración del puerto de vista
     void confPuertoVista(int izda, int dcha, int abajo, int arriba);

};
```

Ejemplo

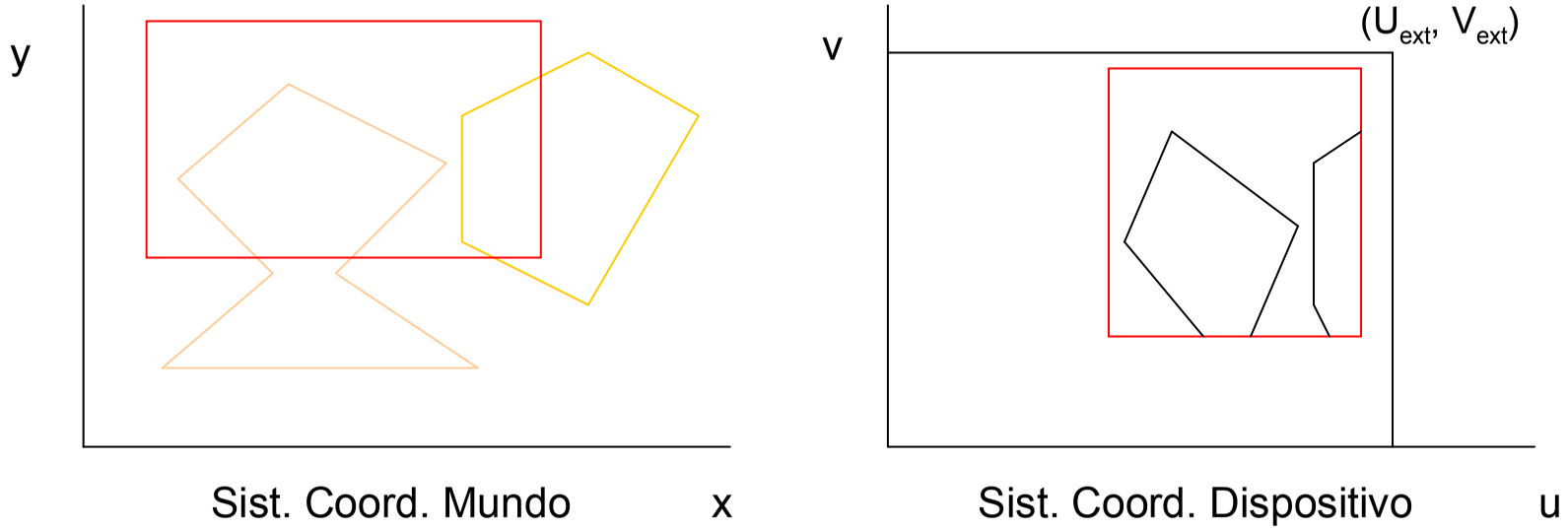
```
class dibujadino : lienzo {
   char nomfich[20] ;
   int dibujar_dino(void);

public:
   dibujadino(int alt, int anch, int orx, int ory);
   void dibujo_normal(void);
   void dibujo_con_zoom(void);
   void dibujo_mosaico(void);
};
   void dibujadino::dibujo_normal(void){

   confVentanaMundo(0.0,(float)anchura, 0.0, (float)altura);
   confPuertoVista(0,anchura, 0, altura);
   dibujar_dino();
   }
```

Ejemplo

```
void dibujadino::dibujo_con_zoom(void){

    confVentanaMundo(0.0,100.0, 350.0, 480.0 ); //Original: 640,480
    confPuertoVista(0,anchura, 0, altura);
                                                    //aparece distorsión
    dibujar_dino();
}


void dibujadino::dibujo_mosaico(void){
   int i,j;

   confVentanaMundo(0.0,(float) anchura, 0.0, (float) altura);

   for(i=0;i<10;i++)
      for(j=0;j<10;j++){
         confPuertoVista(i*64, (i+1)*64   , j*48, (j+1)*48);
         dibujar_dino();
      }
}
```

Cuestiones asociadas: Recorte

- El recorte es un proceso geométrico que consiste en dos pasos:
  + Averiguar si un punto geométrico cae dentro o fuera de la ventana de recorte.
  + En caso de que caiga fuera, encontrar la proyección de ese punto sobre la arista de la ventana.



Cuestiones asociadas: Recorte

- Formalmente, la única restricción en nuestro modelo de visualización para el recorte es que éste debe realizarse en coord. del mundo o en coord. Normalizadas de dispositivo, antes de la fase de tratamiento de píxeles.

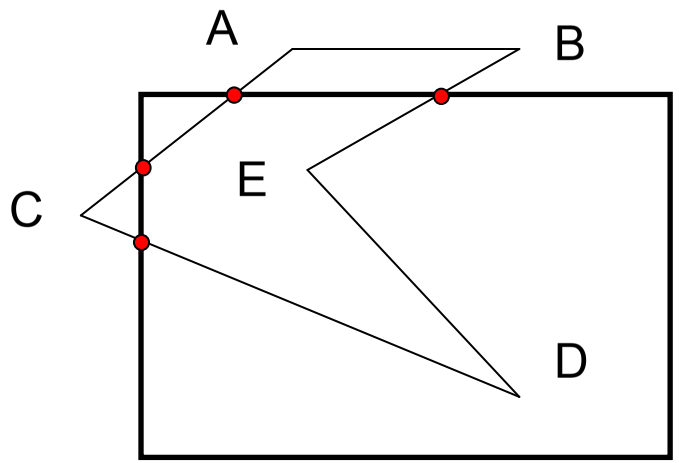
- La opción más conveniente es encadenar la matriz de transformación Ventana - Puerto de vista con las matrices de transformación de la geometría y posteriormente, realizar el recorte. Esta composición de matrices, genera una nueva matriz denominada matriz de **MODELVIEW**

+ OpenGl realiza el recorte automáticamente sobre la nueva ventana en que se ha transformado la ventana del mundo en el sistema de coordenadas normalizadas.

Recorte

- Vamos a tratar el recorte de líneas sobre una ventana genérica que denominaremos “Ventana de recorte”. Esta puede ser la ventana del mundo o bien la ventana normalizada del Viewport.

- En estas condiciones el recorte puede tener cuatro casos diferentes:



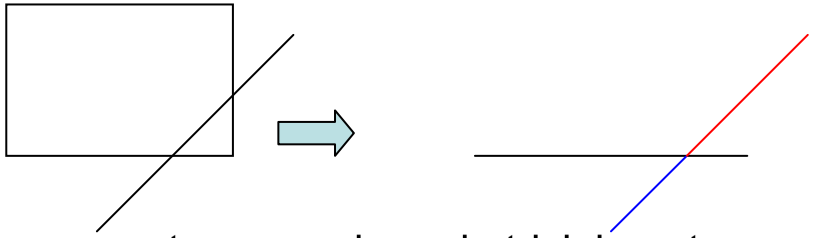
AB es completamente exterior a la ventana

AC, BE tienen una parte al interior

ED es completamente interior

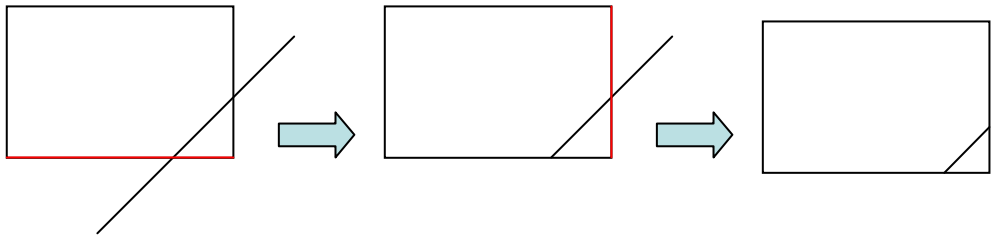
Recorte: Cohen - Sutherland

- El algoritmo realiza un test inicial sobre la línea para determinar si el cálculo de las intersecciones se puede evitar. Esto es posible si la línea está totalmente fuera de la ventana o bien está completamente interior (AB o ED)
- Si no es así, la línea se divide en dos segmentos respecto de un lado de la ventana



- De tal manera que uno de los segmentos es rechazado trivialmente

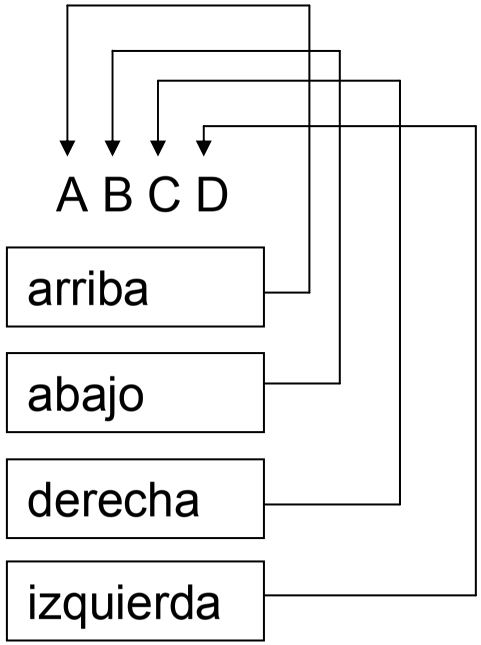
- Se repita esta operación iterativamente sobre los cuatro lados de la ventana de recorte.

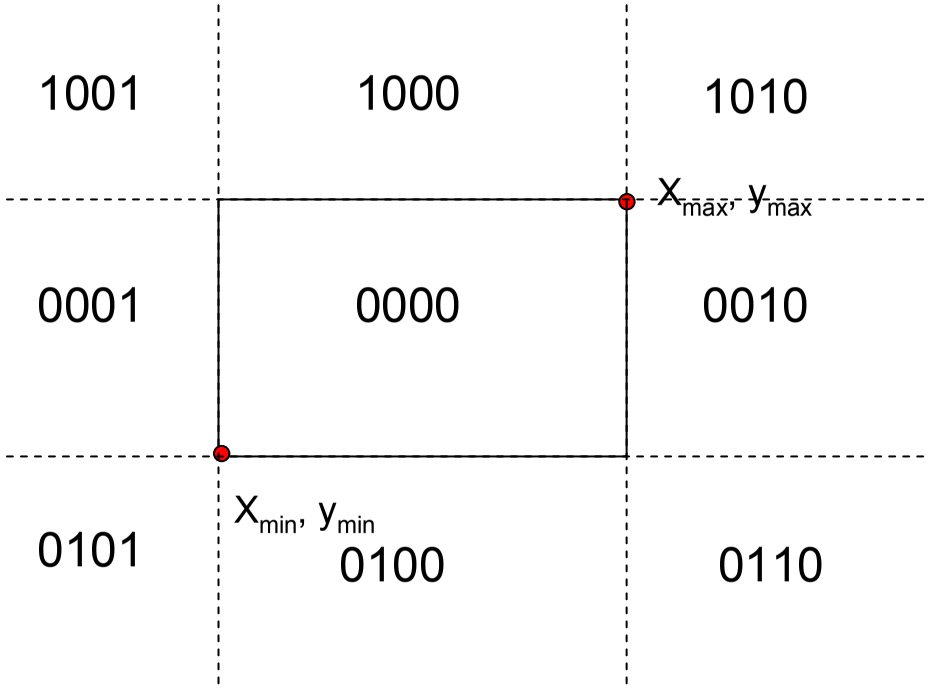


Recorte: Cohen – Sutherland (2)

- El algoritmo es especialmente eficiente en los casos en que los rechazos o las aceptaciones triviales abundan. Estos casos se suelen dar cuando:
  - La ventana del mundo es grande y engloba a la totalidad de la escena
  - La ventana es muy pequeña respecto a la escena y hay muchas líneas que pueden ser rechazadas.

Es necesario definir un método rápido para la identificación de esos casos



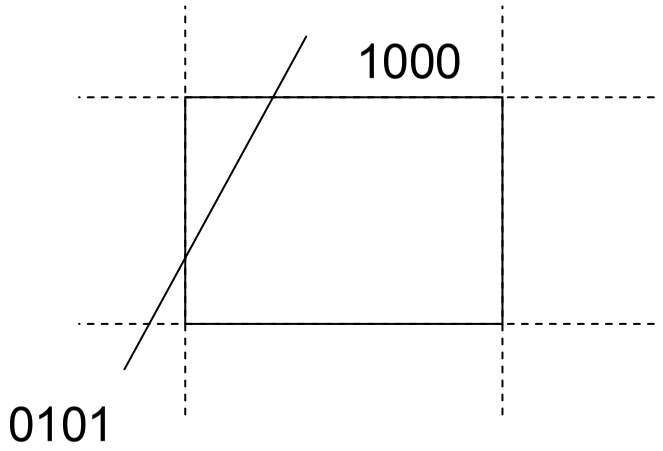


Recorte: Cohen – Sutherland (3)

- Dado un punto P(x,y) ¿Cómo conocer el código que le asocia a una región?

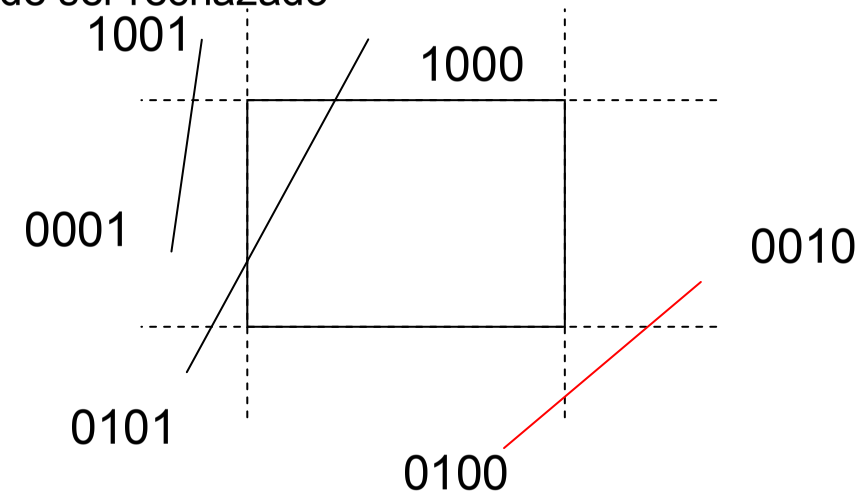
- El código lo almacenamos en **4bits**
- *1er bit:* bit de signo de $y_{max} - y$
- *2do. bit:* bit de signo de $y - y_{min}$
- *3er bit*: bit de signo de $x_{max} - x$
- *4° bit :* bit de signo de $x - x_{min}$

- A cada extremo de la recta a recortar se le asigna su correspondiente código.



Recorte: Cohen – Sutherland (3)

- Determinar si una recta es aceptada o rechazada en su totalidad es muy sencillo
- Si los dos extremos tienen código cero, el segmento es trivialmente aceptado
- Es decir **!(codigo1 OR codigo2)**
- Si la operación ENTRE BITS **AND** entre los dos vértices **NO es CERO**, el segmento puede ser rechazado



1000 AND 0101 = 0

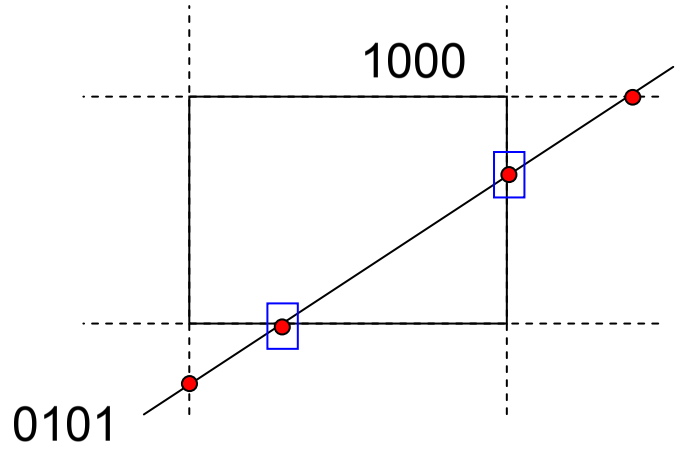
0001 AND 1001 = 1

0100 AND 0010 = 0

Como se ve, este criterio no rechaza alguna configuración de los segmentos que debería ser trivialmente rechazada. Solo rechaza las rectas que caen en uno de los 4 semiplanos en que se divide el espacio

Recorte: Cohen – Sutherland (4)

- Si la línea no puede ser aceptada o rechazada directamente, se pone en marcha un proceso de DIVIDE Y VENCERÄS donde el segmento es subdividido en dos partes por arista de la ventana con la cual intersecta, En cada división, el segmento pierde una parte y se recalcula la codificación del nuevo vértice.
- El orden de recorrido de las aristas es arbitrario y siempre el mismo



Orden pésimo para esta recta: (•)

Izda, abajo, arriba, dcha

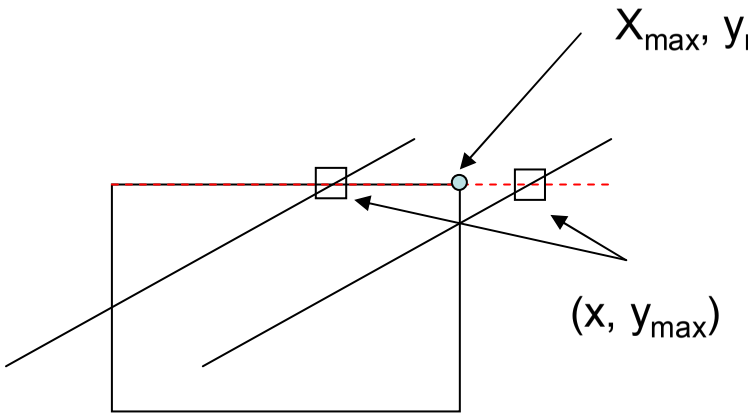
Orden óptimo para esta recta: (□)

Abajo, dcha, izda, arriba

Como estadísticamente no hay direcciones privilegiadas, no hay un orden de recorrido privilegiado.

Recorte: Cohen – Sutherland (5)

- Cálculo de las intersecciones con los ejes.
- $x = x_0 + 1/m\ (y-y_0) = x_0 + (x_1 - x_0) / (y_1-y_0)\ (y-y_0)$
- $y = y_0 - m\ (x - x_0) = y_0 + (y_1 - y_0) / (x_1-x_0)\ (x-x_0)$
- Para la intersección con el lado superior

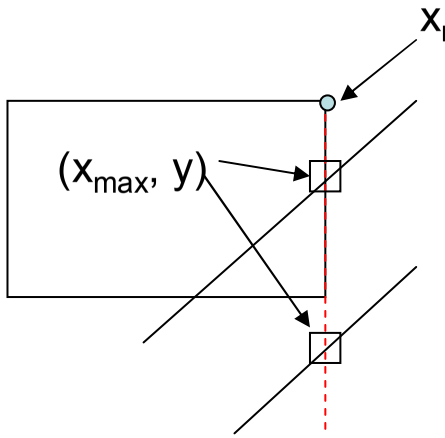


$$\mathbf{x} = x_0 + ((x_1-x_0)/ (y_1-y_0)\ )\ (y_{max}-y_0)$$

$$y = y_{max}$$

Recorte: Cohen – Sutherland (6)

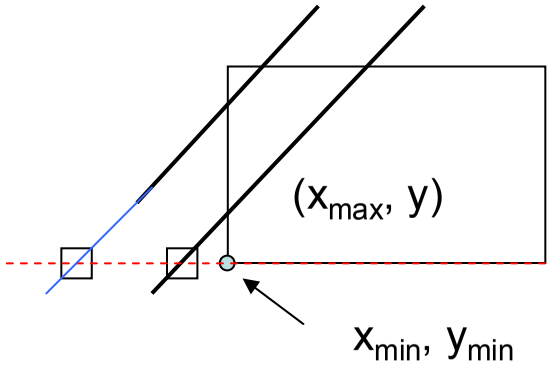
- Para el lado derecho



$$\mathbf{y} = y_0 + ((y_1-y_0)/ (x_1-x_0)\ )\ (x_{max}-x_0)$$

$$x = x_{max}$$

Para el lado inferior

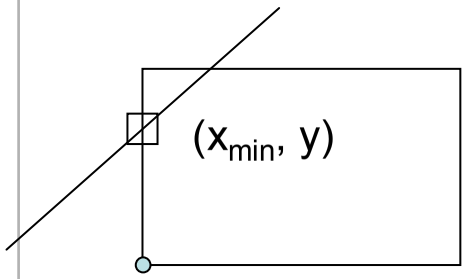


$$\mathbf{x} = x0 + ((x1-x0)/ (y1-y0)\ )\ (y_{min}-y0)$$

$$y = y_{min}$$

Recorte: Cohen – Sutherland (7)

- Para el lado derecho



$$\mathbf{y} = y_0 + ((y_1-y_0)/ (x_1-x_0)\ )\ (x_{min}-x_0)$$

$$x = x_{min}$$

- Como vemos hay un gasto inútil de cálculo de intersecciones con las prolongaciones de los lados de la ventana.
- El algoritmo de Nicholl-Lee es una mejora de este algoritmo que evita estos cálculos (ver Foley-Van Dam)

- Ventajas: El algoritmo de Cohen Sutherland es extrapolable directamente al recorte en 3D.

```
typedef unsigned int outcode;
enum {TOP = 0x1, BOTTOM = 0x2, RIGHT = 0x4, LEFT = 0x8};

void CohenSutherlandLineClipAndDraw (
    double x0, double y0, double x1, double y1, double xmin, double xmax,
    double ymin, double ymax, int value)
/* Cohen-Sutherland clipping algorithm for line P0 = (x0, y0) to P1 = (x1, y1) and */
/* clip rectangle with diagonal from (xmin, ymin) to (xmax, ymax) */
{
    /* Outcodes for P0, P1, and whatever point lies outside the clip rectangle */
    outcode outcode0, outcode1, outcodeOut;
    boolean accept = FALSE, done = FALSE;
    outcode0 = CompOutCode (x0, y0, xmin, xmax, ymin, ymax);
    outcode1 = CompOutCode (x1, y1, xmin, xmax, ymin, ymax);
    do {
        if (!(outcode0 | outcode1)) {              /* Trivial accept and exit */
            accept = TRUE; done = TRUE;
        } else if (outcode0 & outcode1)            /* Logical and is true, so trivial reject and exit */
            done = TRUE;
        else {
            /* Failed both tests, so calculate the line segment to clip: */
            /* from an outside point to an intersection with clip edge. */
            double x, y;
            /* At least one endpoint is outside the clip rectangle; pick it. */
            outcodeOut = outcode0 ? outcode0 : outcode1;
            /* Now find intersection point; */
            /* use formulas y = y0 + slope * (x - x0), x = x0 + (1/slope) * (y - y0). */
            if (outcodeOut & TOP) {                /* Divide line at top of clip rect */
                x = x0 + (x1 - x0) * (ymax - y0) / (y1 - y0);
                y = ymax;
            } else if (outcodeOut & BOTTOM) {      /* Divide line at bottom edge of clip rect */
                x = x0 + (x1 - x0) * (ymin - y0) / (y1 - y0);
                y = ymin;
            } else if (outcodeOut & RIGHT) {       /* Divide line at right edge of clip rect */
                y = y0 + (y1 - y0) * (xmax - x0) / (x1 - x0);
                x = xmax;
            } else {                               /* Divide line at left edge of clip rect */
                y = y0 + (y1 - y0) * (xmin - x0) / (x1 - x0);
                x = xmin;
            }
            /* Now we move outside point to intersection point to clip, */
            /* and get ready for next pass. */
            if (outcodeOut == outcode0) {
                x0 = x; y0 = y; outcode0 = CompOutCode (x0, y0, xmin, xmax, ymin, ymax);
            } else {
                x1 = x; y1 = y; outcode1 = CompOutCode (x1, y1, xmin, xmax, ymin, ymax);
            }
        }  /* Subdivide */
    } while (done == FALSE);
    if (accept)
        MidpointLineReal (x0, y0, x1, y1, value);    /* Version for double coordinates */
}  /* CohenSutherlandLineClipAndDraw */
```

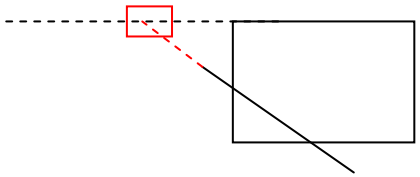
Fig. 3.41 *(Cont.)*

```
outcode CompOutCode (
    double x, double y, double xmin, double xmax, double ymin, double ymax);
{
    outcode code = 0;
    if (y > ymax)
        code |= TOP;
    else if (y < ymin)
        code |= BOTTOM;
    if (x > xmax)
        code |= RIGHT;
    else if (x < xmin)
        code |= LEFT;
    return code;
}  /* CompOutCode */
```

Fig. 3.41 Cohen–Sutherland line-clipping algorithm.

Recorte: Cyrus-Beck

- En el algoritmo de Cohen-Sutherland ocurría que el recorte de las líneas se producía independientemente de que calcularámos un corte real con la ventana de recorte o un corte ficticio con la prolongación de la ventana o del segmento



- El algoritmo de Cyrus-Beck utiliza una representación paramétrica de las rectas que evita el cálculo del punto de recorte hasta que sabemos positivamente cuáles son los lados sobre los que hay que recortar. Se trata de determinar paramétricamente los puntos de corte del segmento (o su prolongación) con los lados de la ventana (o su prolongación) y después calcular los puntos que realmente intersecten con la ventana.

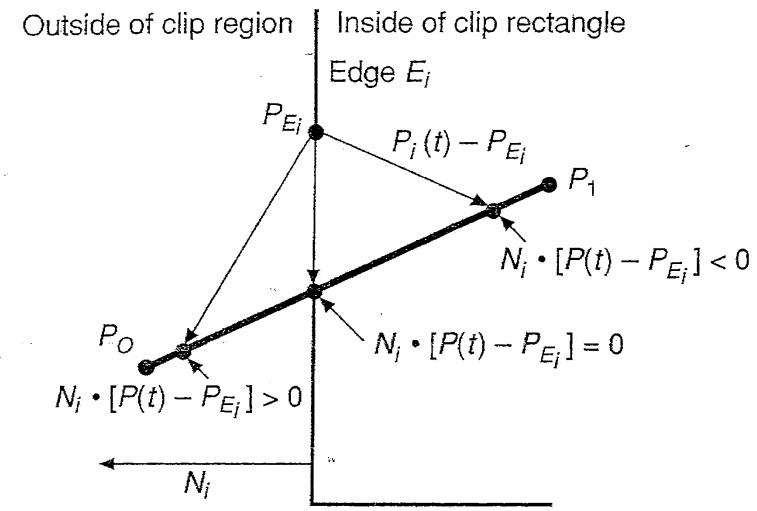
- Este algoritmo puede usarse con muy poca modificación sobre el espacio 3D.

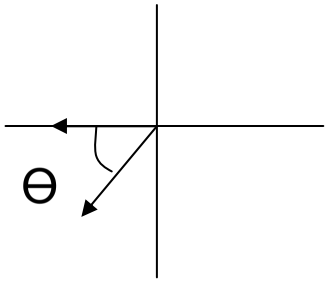
- Consideremos la ecuación paramétrica de la recta

$$\mathbf{P(t) = P_0 + (P_1 - P_0)\, t} \qquad 0 \leq t \leq 1$$

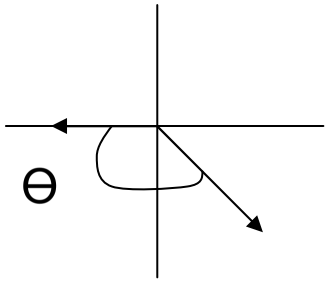
Recorte: Cyrus-Beck (2)

- Vamos a averiguar el valor de **t** para el punto de corte de la recta con una arista de la ventana del mundo
- Sea $\mathbf{P_{Ei}}$ un punto cualquiera de la arista E_i (tomaremos como punto más sencillo un extremo de dicha arista)
- Sea $\mathbf{N_i}$ la normal a dicha arista





Θ < 90



Θ > 90

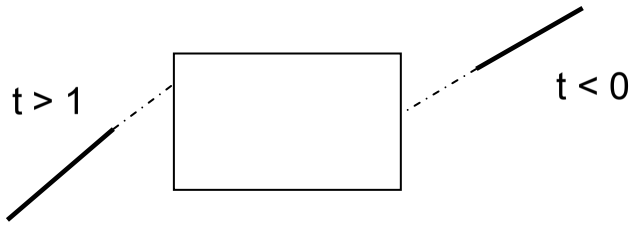
N P = |**N**| |**P**| cos Θ

Recorte: Cyrus-Beck (3)

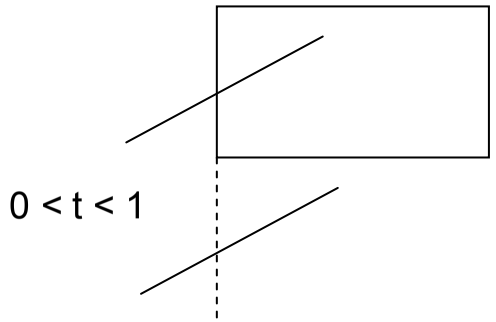
- En la intersección se cumple: $N_i (P(t) – P_{Ei}) = 0$
- $N_i (P_0 + (P_1 - P_0)\, t – P_{Ei}) = 0$
- Como la suma de vectores es conmutativa, podemos agrupar de la siguiente forma:
- $N_i (P_0 – P_{Ei}) + N_i ((P_1 - P_0)\, t\ ) = 0$
- Sea $D = P_1 - P_0$
- Despejando el parámetro t
- $$\mathbf{t = \frac{N_i (P_0 – P_{Ei})}{- N_i D}}$$ Condición de intersección del segmento con la arista
- Esta ecuación tiene un punto singular cuando $\mathbf{N_i D = 0}$ es decir cuando:
  - **D = 0** (es decir que no hay segmento a recortar sino un punto
  - $\mathbf{N_i \perp D}$ (son vectores perpendiculares)

Recorte: Cyrus-Beck (4)

- Se calculan **los cuatro valores de t** (uno para cada arista)
- Si $t \notin [0,1]$ esto significa que la intersección se produce fuera del segmento

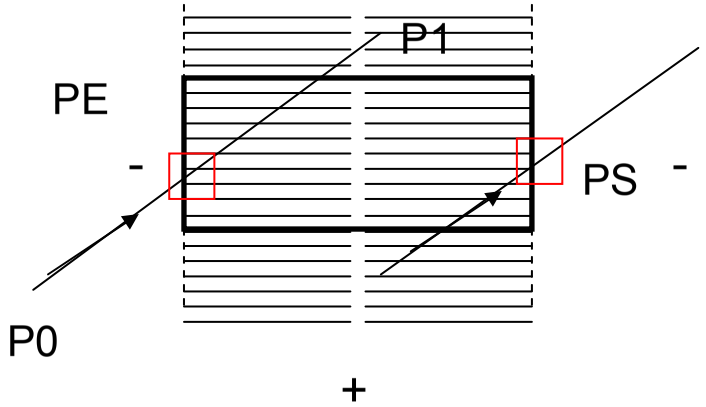


- Sino, necesitamos saber si la intersección cae en la arista de la ventana o en una prolongación suya.



Recorte: Cyrus-Beck (5)

- Clasificamos cada una de las cuatro intersecciones como
- PE (punto de entrada) si dirigiéndonos desde P0 a P1 cruzamos la arista de la ventana introduciéndonos en la parte del semiplano donde se encuentra el área de la ventana



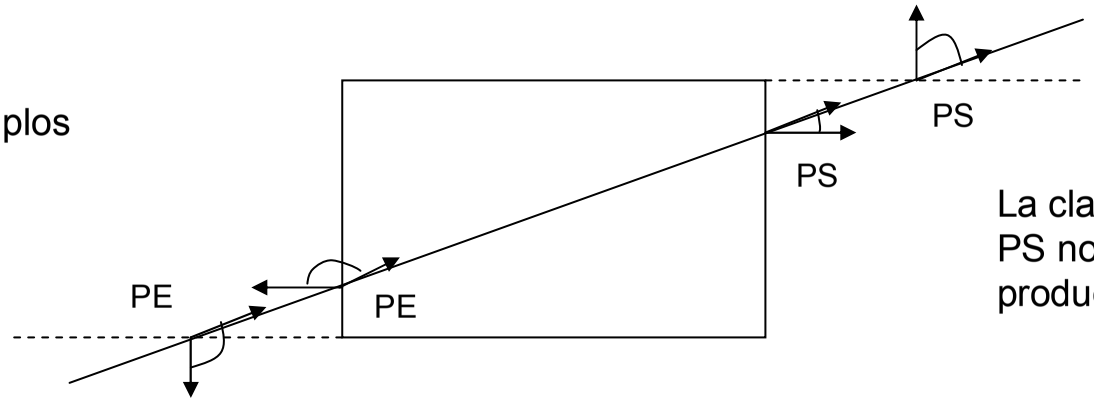
- Igualmente, en caso contrario lo definiremos como PS.

- Formalmente **PE** ocurre cuando $\mathbf{N_i\ D < 0}$ **es decir** $\mathbf{\Theta > 90}$
- **PS** ocurre cuando $\mathbf{N_i\ D > 0}$ **es decir** $\mathbf{\Theta < 90}$

- Fijémonos que esta cantidad la hemos calculado para obtener el valor del parámetro **t**, con lo que clasificar a t como PS o PE no cuesta cálculo adicional.

Recorte: Cyrus-Beck (6)

- Ejemplos



La clasificación de PE o PS nos la da el signo del producto escalar de $\mathbf{N}_i$ **D**

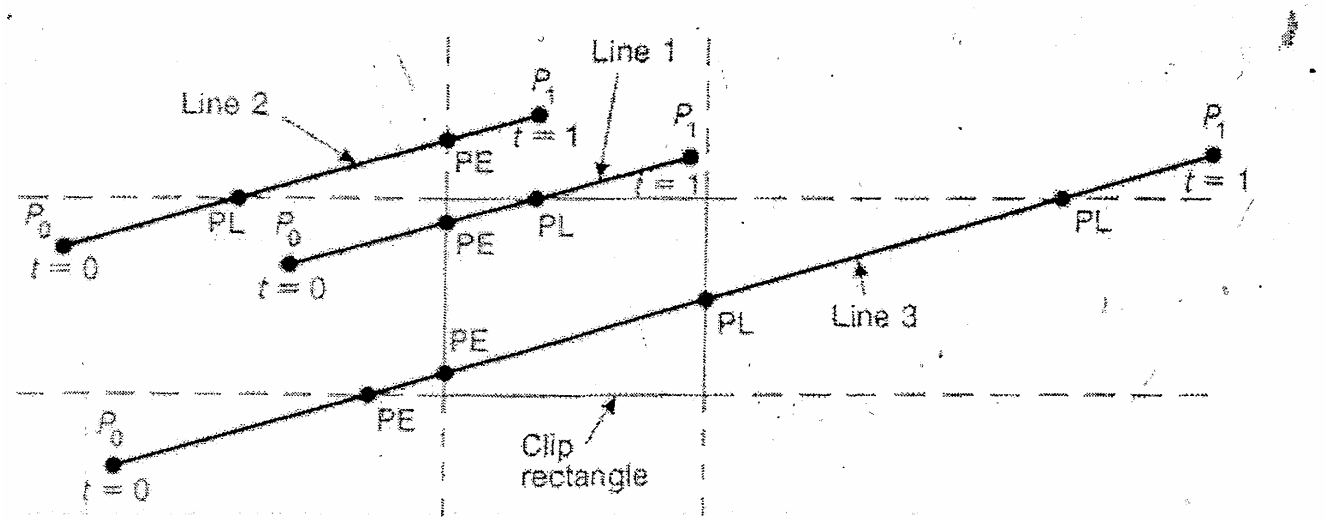
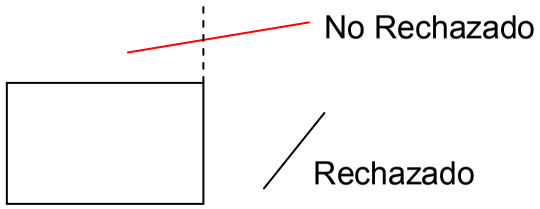


Fig. 3.43 Lines lying diagonal to the clip rectangle.

Recorte: Cyrus-Beck (7)

- Esquema del **algoritmo** de Cyrus Beck
- 1. Calculo las N_i (una por cada arista de la ventana). Elijo por cada arista un extremo E_i
- 2. Dado un segmento a recortar, calculo los cuatro valores de t usando la expresión que hemos obtenido y los clasifico como PE o PS

- 3. Si todos los valores de t son mayor que 1 o menor que 0, podemos rechazar el segmento



¡Cuidado! No todos los segmentos que están fuera son rechazados por este criterio

- 4. Sino, elijo el **MAYOR PE (que llamo t_E)** y el **MENOR PS (que llamo t_S)**

- 5. **Si $t_E > t_S$, el segmento es RECHAZADO**
- **Sino, se calculan los puntos $P(t_E)$ y $P(t_s)$ que son los puntos de recorte buscados.**

Recorte: Cyrus-Beck (8)

- **Observaciones:**
- El cálculo de t_i i=1,2,3,4 es más rápido que el cálculo de las cuatro intersecciones, incluso si el número medio de intersecciones a calcular por Cohen-Sutherland es mayor que 2 y menos de 3 ya sale rentable

- $t_{zda} = \dfrac{-(x_0 - x_{min})}{(x_1 - x_0)}$ frente a $y_{izda} = \dfrac{x_{min} - x}{x' - x}(y' - y_{min}) + y_{min}$

- La expresión concreta de t depende de la forma de la ventana de recorte. Recordemos que el algoritmo de Cyrus-Beck es válido para cualquier polígono de recorte cóncavo (o cualquier poliedro en 3D).

- Cuando la ventana de recorte es rectangular, la expresiones son muy simples

TABLE 3.1 CALCULATIONS FOR PARAMETRIC LINE CLIPPING ALGORITHM*

Clip edge$_i$	Normal N_i	P_{E_i}	$P_0 - P_{E_i}$	$t = \dfrac{N_i \cdot (P_0 - P_{E_i})}{-N_i \cdot D}$
left: $x = x_{min}$	$(-1, 0)$	(x_{min}, y)	$(x_0 - x_{min}, y_0 - y)$	$\dfrac{-(x_0 - x_{min})}{(x_1 - x_0)}$
right: $x = x_{max}$	$(1, 0)$	(x_{max}, y)	$(x_0 - x_{max}, y_0 - y)$	$\dfrac{(x_0 - x_{max})}{-(x_1 - x_0)}$
bottom: $y = y_{min}$	$(0, -1)$	(x, y_{min})	$(x_0 - x, y_0 - y_{min})$	$\dfrac{-(y_0 - y_{min})}{(y_1 - y_0)}$
top: $y = y_{max}$	$(0, 1)$	(x, y_{max})	$(x_0 - x, y_0 - y_{max})$	$\dfrac{(y_0 - y_{max})}{-(y_1 - y_0)}$

Recorte: Cyrus-Beck (9)

```
begin
    precalculate N_i and select a P_Ei for each edge;

    for each line segment to be clipped
      if P_1 = P_0 then
        line is degenerate so clip as a point;
      else
        begin
          t_E = 0; t_L = 1;
          for each candidate intersection with a clip edge
            if N_i • D <> 0 then  {Ignore edges parallel to line for now}
              begin
                calculate t;
                use sign of N_i • D to categorize as PE or PL;
                if PE then t_E = max(t_E, t );
                if PL then t_L = min(t_L, t )
              end;
          if t_E <> t_L then
            return nil
          else
            return P(t_E) and P(t_L) as true clip intersections
        end
end
```

¡ojo este pseudocódigo no considera todavía las líneas paralelas a la ventana

Fig. 3.44 Pseudocode for Cyrus–Beck parametric line clipping algorithm.

Recorte: Cyrus-Beck (10)

```
void Clip2D (double *x0, double *y0, double *x1, double *y1, boolean *visible)
/* Clip 2D line segment with endpoints (x0, y0) and (x1, y1), against upright */
/* clip rectangle with corners at (xmin, ymin) and (xmax, ymax); these are */
/* globals or could be passed as parameters also. The flag visible is set TRUE. */
/* if a clipped segment is returned in endpoint parameters. If the line */
/* is rejected, the endpoints are not changed and visible is set to FALSE. */
{
    double dx = *x1 - *x0;
    double dy = *y1 - *y0;
    /* Output is generated only if line is inside all four edges. */
    *visible = FALSE;
    /* First test for degenerate line and clip the point; ClipPoint returns */
    /* TRUE if the point lies inside the clip rectangle. */
    if (dx == 0 && dy == 0 && ClipPoint (*x0, *y0))
        *visible = TRUE;
    else {
        double tE = 0.0;
        double tL = 1.0;
        if (CLIPt (dx, xmin - *x0, &tE, &tL))                    /* Inside wrt left edge */
            if (CLIPt (-dx, *x0 - xmax, &tE, &tL))               /* Inside wrt right edge */
                if (CLIPt (dy, ymin - *y0, &tE, &tL))            /* Inside wrt bottom edge */
                    if (CLIPt (-dy, *y0 - ymax, &tE, &tL)) {     /* Inside wrt top edge */
                        *visible = TRUE;
                        /* Compute PL intersection, if tL has moved */
                        if (tL < 1) {
                            *x1 = *x0 + tL * dx;
                            *y1 = *y0 + tL * dy;
                        }
                        /* Compute PE intersection, if tE has moved */
                        if (tE > 0) {
                            *x0 += tE * dx;
                            *y0 += tE * dy;
                        }
                    }
    }
} /* Clip2D */
```

Puede darse el caso de no tener que acabar de calcular todos los valores de t

```
boolean CLIPt (double denom, double num, double *tE, double *tL)
{
    double t;

    if (denom > 0) {            /* PE intersection */
        t = num / denom;        /* Value of t at the intersection */
        if (t > tL)             /* tE and tL crossover */
            return FALSE;       /* so prepare to reject line */
        else if (t > tE)        /* A new tE has been found */
            tE = t;
    } else if (denom < 0) {     /* PL intersection */
        t = num / denom;        /* Value of t at the intersection */
        if (t < tE)             /* tE and tL crossover */
            return FALSE;       /* so prepare to reject line */
        else if (t < tL)        /* A new tL has been found */
            tL = t;
    } else if (num > 0)         /* Line on outside of edge */
        return FALSE;
    return TRUE;
} /* CLIPt */
```

Caso de línea paralela a alguna arista

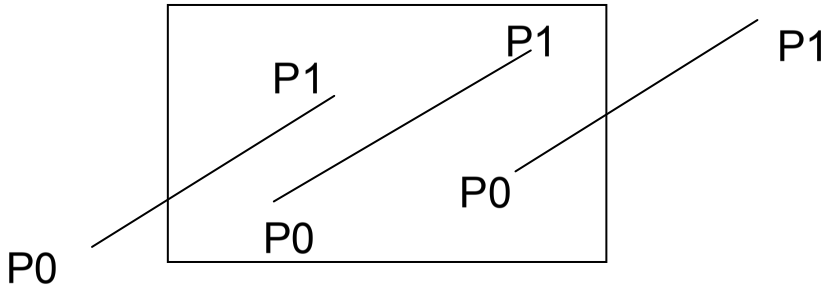
TABLE 3.1 CALCULATIONS FOR PARAMETRIC LINE CLIPPING ALGORITHM*

Clip edge $_i$	Normal N_i	P_{E_i}	$P_0 - P_{E_i}$	$t = \frac{N_i \cdot (P_0 - P_{E_i})}{-N_i \cdot D}$
left: $x = x_{min}$	$(-1, 0)$	(x_{min}, y)	$(x_0 - x_{min}, y_0 - y)$	$\frac{-(x_0 - x_{min})}{(x_1 - x_0)}$
right: $x = x_{max}$	$(1, 0)$	(x_{max}, y)	$(x_0 - x_{max}, y_0 - y)$	$\frac{(x_0 - x_{max})}{-(x_1 - x_0)}$
bottom: $y = y_{min}$	$(0, -1)$	(x, y_{min})	$(x_0 - x, y_0 - y_{min})$	$\frac{-(y_0 - y_{min})}{(y_1 - y_0)}$
top: $y = y_{max}$	$(0, 1)$	(x, y_{max})	$(x_0 - x, y_0 - y_{max})$	$\frac{(y_0 - y_{max})}{-(y_1 - y_0)}$

Foley, Van Dam

Recorte: Cyrus-Beck (11)

- **Detalles:**
- ¿Qué ocurre con el algoritmo para líneas que tienen un extremo en el interior de la ventana de recorte? ¿Y para líneas totalmente interiores a la ventana?



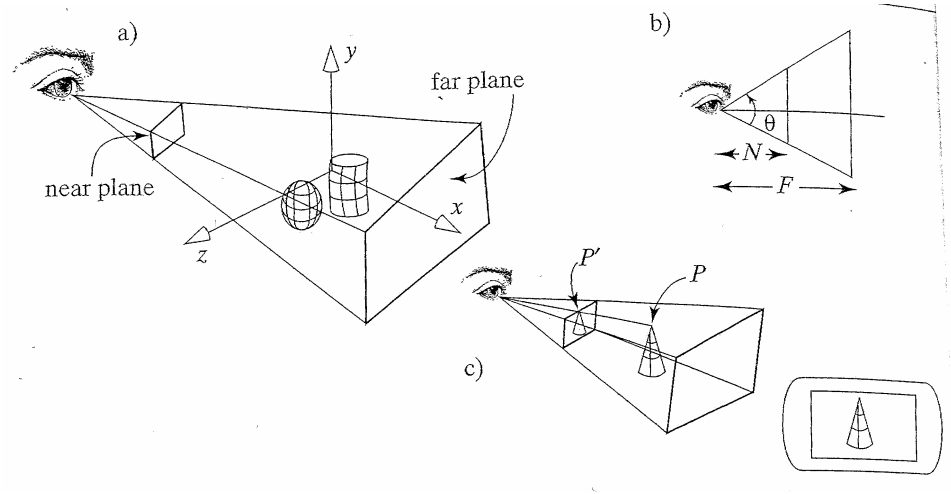
- ¿Podemos elegir como punto inicial P0 cualquiera de los dos vértices?

- El funcionamiento real de Clipt() en el algoritmo propuesto (Foley) es ligeramente distinto de la teoría.

- ¿Qué ocurre para segmentos que son paralelos a alguna arista de la ventana?

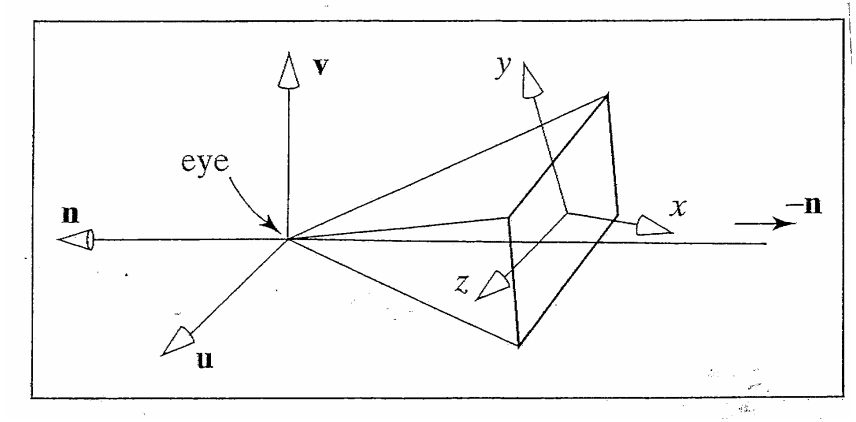
Transformación de vista en 3D

- Vamos a definir un nuevo sistema de coordenadas que es el **S.C. de Vista**. Es también un sistema en notación de punto flotante.
- La **transformación de vista** tiene como objetivo situar el punto de vista de la escena.
- Además es en el sistema de coordenadas de vista donde se define el volumen de visión y la proyección sobre un plano que posteriormente será renderizada.



- Vamos a definir un sistema de coordenadas de vista, cuyo origen estará centrado en la posición de la cámara y tendrá tres direcciones ortogonales definidas por **n**, **u**, **v**

Transformación de vista en 3D (2)



En OpenGl **n** es la dirección opuesta a la dirección de mirada de la cámara. **u** tiene la dirección ortogonal a la que establece el "arriba" de la cámara (que no el eje y de la cámara) y **u** es un vector ortonormal a ambos

Esta configuración puede variar para otras librerías gráficas

Dados el punto donde se sitúa la cámara y un punto a donde se dirige la mirada, (generalmente el origen de coordenadas), obtenemos el vector **n**

N = (**posición de la cámara** – **punto donde mira**)

n = **N** / |**N**| (vector normalizado)

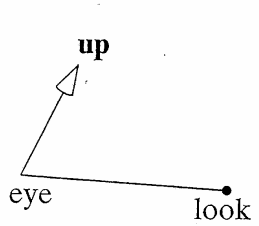
U = **arriba** x **n**

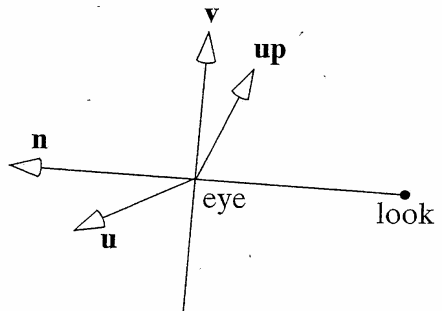
u = **U** / |**U**|

V es un vector perpendicular a ambos

V = **n** x **u**

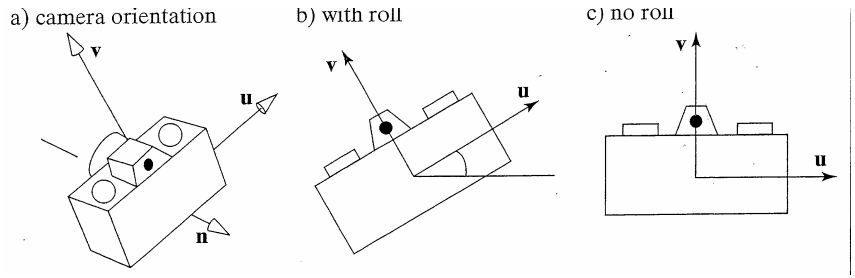
v = **V** / |**V**|

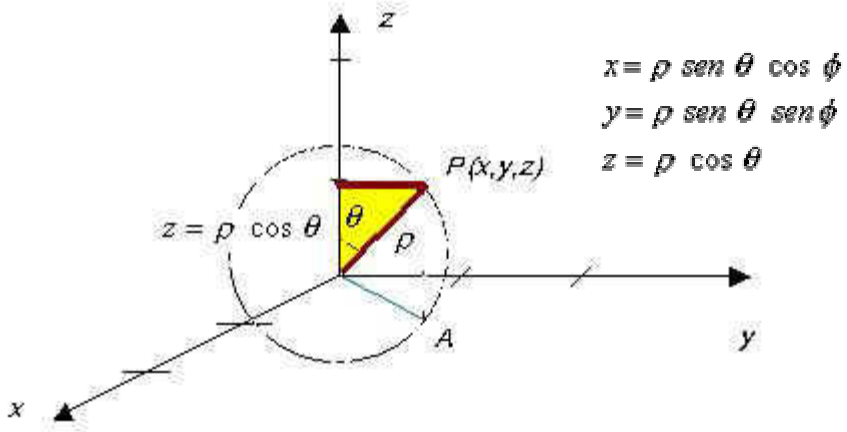




La dirección del vector **arriba** establece la dirección positiva del vector **v**

Transformación de vista en 3D (3)





Una manera de hacer cambiar de orientación a la cámara es modificar los componentes de estos vectores respecto del sistema de coordenadas del mundo, usando coord. Esféricas

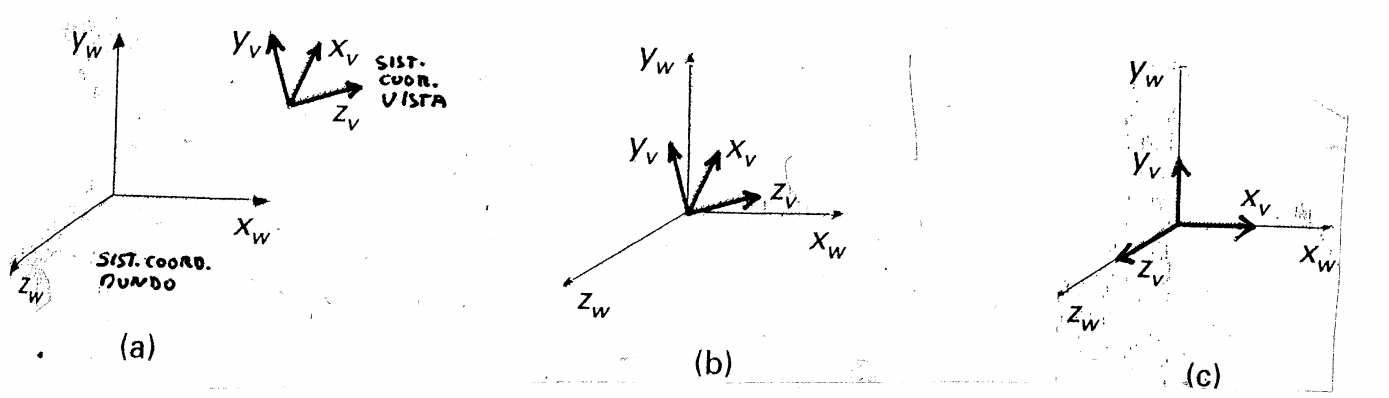
Modificando Θ y ρ, obtenemos nuevas orientaciones de, por ejemplo **u**, y esto genera un cambio del punto de vista

Otra manera es modificando los dos vectores que configuran el sistema de vista. Para el ejemplo del “roll” , modificaríamos el vector “arriba”. Aunque esto es menos intuitivo que el otro método

Transformación de vista en 3D (4)

- Una vez tenemos definido el sistema de coordenadas de vista ¿cómo expresar la transformación que nos lleva desde el sistema de coordenadas del mundo al de vista?

- Método 1:



Traslación al origen de coordenadas del mundo (SCM)

Rotación alrededor del eje X del SCM para trasladar **n** al plano XZ (R_x)

Rotación alrededor de Y del SCM para alinear los ejes de **z** y **v** (R_y)

Rotación alrededor de Z del SCM para alinear los ejes **y** y **v** (R_z)

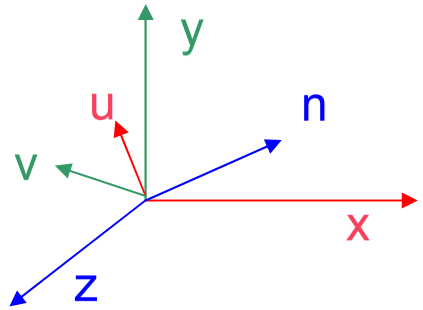
Traslación al punto de situación de la cámara (x_c, y_c, z_c) (T)

La matriz total será: $R_z\ R_y\ R_x\ T$

Transformación de vista en 3D (5)

- Los vectores **n**, **u**, **v** formar una matriz de transformación de sólido rígido que orienta adecuadamente el sistema de coordenadas de vista respecto del SCM

- **R** = $\begin{pmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$



- La transformación de vista total será

  V = T R

  a las coordenadas del SC de Vista se les llama también coordenadas del ojo (eye coordinates)

- **Esta matriz se pre-multiplica por la matriz de transformación afín que sitúa los a un objeto en el mundo configurando la matriz de MODELVIEW que se aplica directamente a la geometría de dicho objeto**

- **MODELVIEW = V M**

Transformación de vista en la Opengl

- Existe la instrucción:

```
glLookAt(ojo.x,ojo.y,ojo.z,look.x,look.y,look.x,arriba.x,arriba.y,arriba.z)
```

Donde `ojo` es la posición de la cámara en el mundo, `look` es el punto hacia donde se mira y `arriba` es el vector que apunta hacia el zenit de la cámara.

Como sabemos es la información necesaria para calcular los vectores **n**,**u**,**v** y construir la matriz de vista

Esta instrucción debe ir siempre "blindada" por la activación de la pila de transformaciones de modelview

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glLookAt(….);
```

Veamos un ejemplo proveniente de la página web de Nate Robbins (enlace en la página web de la asignatura)