MODELADO Y VISUALIZACIÓN GRÁFICA

PRÁCTICA 4

TRANSFORMACIÓN VISTA

Organizamos la práctica según el siguiente índice:

0. Funciones necesarias.

1. Transformación cámara, M_{cam}

2. Transformación de proyección, $M_{proy} = M_{orth}M_{persp}$

3. Transformación de encuadre o viewport, M_{vp}

4. Ejercicio completo.

0. Funciones necesarias.

A continuación, con vistas a facilitar y agilizar su uso, se incluye una celda ejecutable con un compendio de las rutinas definidas en las sesiones previas.

In [1]:
```
def vector_puntos(P,Q):

    # Input: puntos P y Q
    # Output: vector PQ

    return (P.augment(Q))*matrix([[-1],[1]])

def producto_escalar(U,V):

    # Input: vectores U y V
    # Output: escalar U*V

    return (transpose(U)*V)[0,0]

def modulo(U):

    # Input: vector U
    # Output: escalar ||U||

    return sqrt(producto_escalar(U,U))
```

```
def vector_unitario(U):

    # Input: vector U
    # Output: vector unitario en la misma dirección y sentido que U

    return U/modulo(U)

def coseno_del_angulo_que_forman(U,V):

    # Input: vectores U y V
    # Output: escalar que representa el coseno del ángulo que forman U y V

    return producto_escalar(U,V)/(modulo(U)*modulo(V))

def producto_vectorial(U,V):

    # Input: vectores U y V
    # Output: vector producto vectorial UxV

    W0 = U[1,0]*V[2,0] - U[2,0]*V[1,0]
    W1 = -U[0,0]*V[2,0] + U[2,0]*V[0,0]
    W2 = U[0,0]*V[1,0] - U[1,0]*V[0,0]
    return matrix([[W0],[W1],[W2],[0]])

def baricentro(vertices):

    # Input: matriz de vértices por columnas con coordenadas homogéneas de una cara (q
    # Output: baricentro de la cara.


    V=vertices
    B = matrix([[sum(V.row(0))],[sum(V.row(1))],[sum(V.row(2))],[sum(V.row(3))]])/sum(
    return B

def dibujar_punto(P,**kwds):

    #Input: punto P
    #Output: gráfica del punto

    return points(P[0:3].column(0),**kwds)
```

Volver al índice

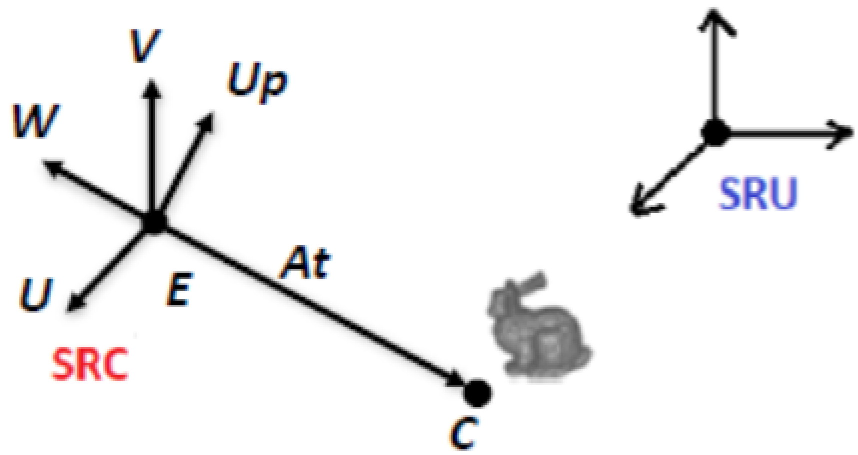
1. Transformación cámara, M_{cam}.

La **transformación cámara**, cuya matriz denotamos por M_{cam}, lleva los puntos del $SRU = \{O, X, Y, Z\}$ al sistema de referencia de la cámara $SRC = \{E, U, V, W\}$. Los parámetros de la cámara deben ser:

- $E =$ posición de la cámara.
- $C =$ punto de mita ($At = EC$)
- Vector orientación Up.

A partir de dichos parámetros se determinan los vectores de la base del SRC:

- $W = \dfrac{-At}{||At||}$
- $U = \dfrac{Up \times W}{||Up \times W||}$
- $V = W \times U$



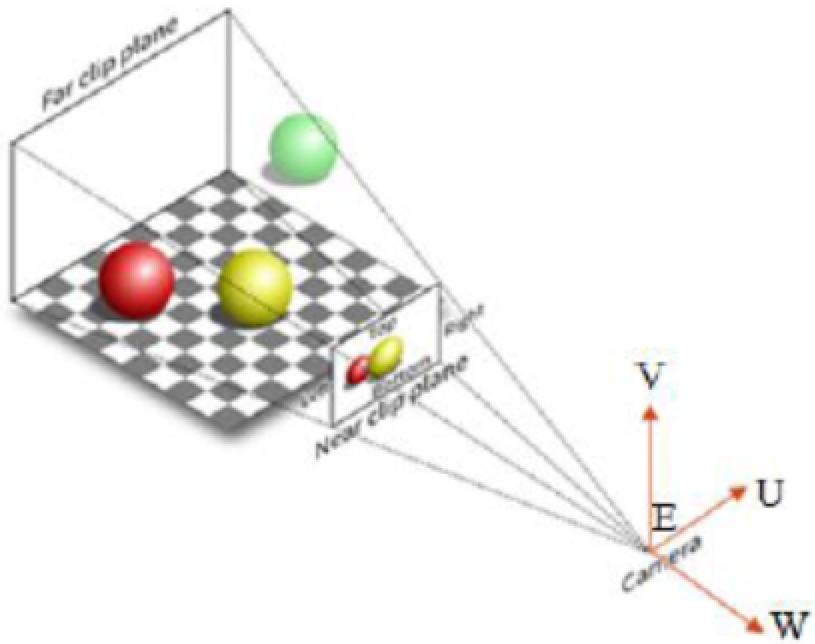
Las siguientes funciones **transf_camara(E,At,Up)** y **transf_camara_inv(E,At,Up)**, devuelven la matrices de las transformaciones directa e inversa, respectivamente:

In [2]:
```
def transf_camara(E,At,Up):

    # SRU -> SRC

    W = -At/modulo(At)
    U = producto_vectorial(Up,W)
    U = U/modulo(U)
    V = producto_vectorial(W,U)

    a = producto_escalar(U,E)
    b = producto_escalar(V,E)
    c = producto_escalar(W,E)

    M_cam = matrix([[U[0,0],U[1,0],U[2,0],-a],[V[0,0],V[1,0],V[2,0],-b],[W[0,0],W[1,0

    return M_cam

#####################################################################################

def transf_camara_inv(E,At,Up):

    # SRC -> SRU

    W = -At/modulo(At)
    U = producto_vectorial(Up,W)
    U = U/modulo(U)
    V = producto_vectorial(W,U)

    M_cam_inv = matrix([[U[0,0],V[0,0],W[0,0],E[0,0]],[U[1,0],V[1,0],W[1,0],E[1,0]],[U
```

```
    return M_cam_inv
```

Si respecto al SRC se fijan los planos *near* ($W = n < 0$) y *far* ($W = f < n$) y los puntos de amplitud horizontal (*left*, l; *right*, r; *bottom*, b; y *top*, t), queda unívocamente determinado el espacio que ve la cámara, el **view frustum o pirámide truncada**.

La función **view_frustum(l,r,b,t,n,f)** permite visualizar este espacio (los planos que lo delimitan se dibujan con textura transparente, para permitir ver qué queda en su interior; todos ellos con color amarillo, salvo los planos *bottom*, en *blue*; y *right*, en *red*).

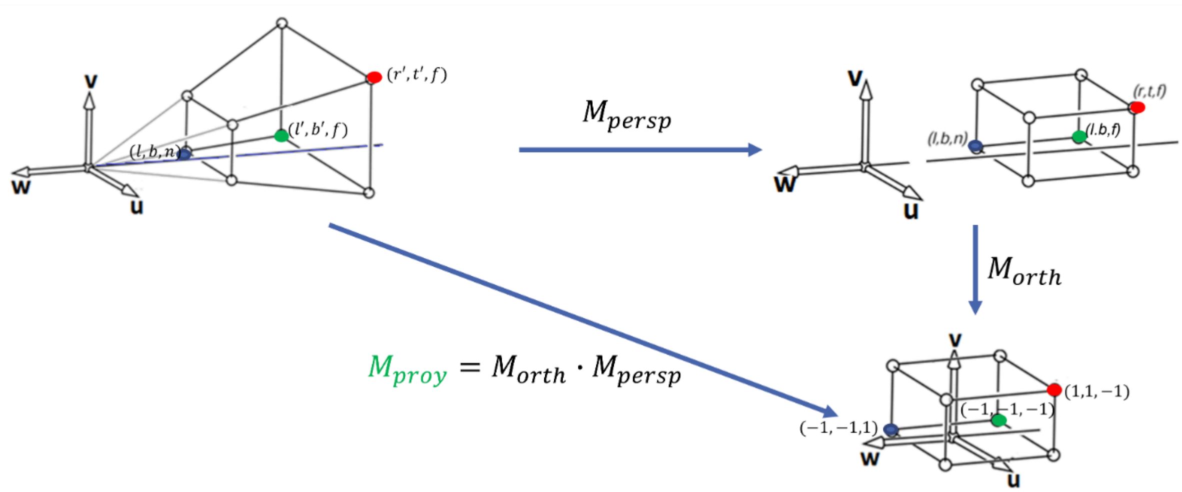


In [3]:
```
def view_frustum(l,r,b,t,n,f):

    # Input: dimensiones del view frustum
    # Output: view furstum.

    r2,l2,t2,b2 = var('r2,l2,t2,b2')
    r2 = r*f/n
    l2 = l*f/n
    t2 = t*f/n
    b2 = b*f/n
    suelo = polygon([[r,b,n],[r2,b2,f],[l2,b2,f],[l,b,n]], color = 'blue', opacity=.25
    techo = polygon([[r,t,n],[r2,t2,f],[l2,t2,f],[l,t,n]], color = 'yellow', opacity=.
    derecha = polygon([[r,b,n],[r2,b2,f],[r2,t2,f],[r,t,n]], color = 'red', opacity=.2
    izquierda = polygon([[l,b,n],[l2,b2,f],[l2,t2,f],[l,t,n]], color = 'yellow', opaci
    frente = polygon([[l,b,n],[r,b,n],[r,t,n],[l,t,n]], color = 'yellow', opacity=.25)
    fondo = polygon([[l2,b2,f],[r2,b2,f],[r2,t2,f],[l2,t2,f]], color = 'yellow', opaci

    return suelo + techo + derecha + izquierda + fondo + frente
```

Volver al índice

2. Transformación de proyección, $M_{proy} = M_{orth} \cdot M_{persp}$.

La **transformación de proyección** lleva los puntos del interior del view frustum (en coordenaadas del SRC) al volumen de vista canónico $[-1,1] \times [-1,1] \times [-1,1]$. Se hace en dos pasos: primero, **la transformación de perspectiva** (que **es proyectiva**, no afín, por lo que **se debe normalizar la salida para la cuarta coordenada sea 1**), lleva el view frustum a un cubo determinado por los vértices (l,b,f) y (r,t,n); después, la transformación ortográfica lleva este cubo al volumen de vista canónico.



Las siguientes funciones **transf_persp(n,f)**, **transf_morth(l,r,b,t,n,f)** y **transf_persp_inv(n,f)**, **transf_morth_inv(l,r,b,t,n,f)**, devuelven la matrices de las transformaciones directas e inversas, respectivamente. Nótese que a lo largo del proceso, se debe usar el método **normalizar(matrix)** para normalizar las coordenadas homogéneas de los puntos de que se trate.

In [4]:
```
def transf_persp(n,f):

    # Input: coordenadas planos near y far.
    # Output: matriz de la transformación proyectiva de perspectiva (luego hay que nor

    M_persp = matrix([[n, 0, 0, 0],[0, n, 0, 0],[0, 0, n+f, -f*n],[0, 0, 1, 0]])

    return M_persp

#####################################################################################

def transf_persp_inv(n,f):

    # Input: coordenadas planos near y far.
    # Output: matriz de la transformación inversa de perspectiva, trabajando en proyec

    M_persp_inv = matrix([[f, 0, 0, 0],[0, f, 0, 0],[0, 0, 0, f*n],[0, 0, -1, n+f]])

    return M_persp_inv

#####################################################################################
```

```
def normalizar(matrix):

    # Input: matriz de puntos en coordenadas homogéneas eventualmente no normalizadas
    # Output: matriz de mismos puntos en coordenadas homogéneas normalizadas

    Smatrix = matrix
    ncols = matrix.ncols()
    for j in range(ncols):
            Smatrix[:,j] = matrix[:,j]/matrix[3,j]

    return Smatrix.n()

#####################################################################################

def transf_morth(l,r,b,t,n,f):

    # Input: dimensiones del view frustum
    # Output: matriz de la transformación ortogonal, que lo lleva al volumen de vista

    M_morth = matrix([[2/(r-l), 0, 0, -(r+l)/(r-l)],[0, 2/(t-b), 0, -(t+b)/(t-b)],[0,

    return M_morth

#####################################################################################

def transf_morth_inv(l,r,b,t,n,f):

    # Input: dimensiones del view frustum
    # Output: matriz de la transformación ortogonal inversa.

    M_morth_inv = matrix([[(r-l)/2, 0, 0, (r+l)/2],[0, (t-b)/2, 0, (t+b)/2],[0, 0, (n-

    return M_morth_inv
```

Volver al índice

Campo de visión de la cámara:

A veces, en lugar de conocer directamente las dimensiones del view frustum $\{n, f, r, l, t, b\}$, conocemos los **ángulos de apertura horizontal y vertical** de la cámara, fov_x y fov_y, respectivamente. A partir de dichos ángulos, es posible determinar las dimensiones del view frustum:

- $r = |n| tg\left(\frac{fov_x}{2}\right)$, $l = -r$
- $t = |n| tg\left(\frac{fov_y}{2}\right)$, $b = -t$

La funciones **transf_morth_fov** y **transf_morth_fov_inv** implementadas a continuación devuelven la matrices de la transformación ortográfica directa e inversa, conocidos los ángulos de visión de la cámara:

In [5]:
```
def transf_morth_fov(fovx,fovy,n,f):

    # Input: ángulos de visión fovx y fovy y planos n y f.
```

```
    # Output: matriz de la transformación ortogonal, que lo lleva al volumen de vista

    r = abs(n*tan(fovx/2))
    l = -r
    t = abs(n*tan(fovy/2))
    b = -t

    M_morth = matrix([[2/(r-l), 0, 0, -(r+l)/(r-l)],[0, 2/(t-b), 0, -(t+b)/(t-b)],[0,

    return M_morth

#########################################################################################

def transf_morth_fov_inv(fovx,fovy,n,f):

    # Input: ángulos de visión fovx y fovy y planos n y f.
    # Output: matriz de la transformación ortogonal inversa.

    r = abs(n*tan(fovx/2))
    l = -r
    t = abs(n*tan(fovy/2))
    b = -t

    M_morth = matrix([[(r-l)/2, 0, 0, (r+l)/2],[0, (t-b)/2, 0, (t+b)/2],[0, 0, (n-f)/2

    return M_morth
```

Volver al índice

3. Transformación de encuadre o viewport, M_{vp}.

La transformación de encuadre lleva el cubo canónico a la pantalla de dimensiones $n_x \times n_y$ pixeles, de manera continua, de forma que cada píxel (de coordenadas enteras) es el centro de un cuadrado de lado 1; por lo que la pantalla abarca un rectángulo continuo de dimensiones $[-0.5, -0,5] \times [n_x - 0.5, n_y - 0,5]$. Está implementado mediante las funciones **transf_vp(nx,ny)** y **transf_vp_inv(nx,ny)**, que devuelven la matrices de las transformaciones directa e inversa, respectivamente.

In [6]:
```
def transf_vp(nx,ny):

    # Input: dimensiones de la pantalla en píxeles.
    # Output: matriz de la transformación de encuadre (viewport).

    M_vp = matrix([[nx/2, 0, 0, (nx-1)/2],[0, ny/2, 0, (ny-1)/2],[0, 0, 1, 0],[0, 0, 0

    return M_vp

#########################################################################################

def transf_vp_inv(nx,ny):

    # Input: dimensiones de la pantalla en píxeles.
    # Output: matriz de la transformación inversa de encuadre (viewport).
```

```
    M_vp_inv = matrix([[2/nx, 0, 0, (1-nx)/nx],[0, 2/ny, 0, (1-ny)/ny],[0, 0, 1, 0],[0
    
    return M_vp_inv
```

El último paso, es redondear el punto al píxel que corresponda, situado en el centro del cuadrado de lado 1 y coordenadas enteras de que se trate. La funcion **dar_pixel(P)** devuele los píxeles que corresponden en pantalla a una matriz de puntos P tras la transformación de encuadre.

In [7]:
```
def dar_pixel(P):
    
    # Input: matriz de puntos en pantalla, tras transformación de encuadre
    # Output: píxeles de los puntos P en pantalla
    
    Q = copy(P)
    for i in range(P.ncols()):
        Q[0,i] = round(P[0,i])
        Q[1,i] = round(P[1,i])
    
    return [int(Q[0,0]), int(Q[1,0])]
```

Relación de aspecto:

Para que no se produzca una distorsión visual, debe verificarse una *relación de aspecto* entre las proporciones de anchura y altura del view frustrum tras la transformación de perspectiva, y las proporciones de anchura y altura de la pantalla:

$$\frac{r-l}{t-b} = \frac{n_x}{n_y}.$$

Esta proporción suele representarse a partir de su fracción irreducible $\frac{a}{b}$ en la forma $a : b$. Dependiento del soporte, hay diferentes estándares de proporción; consúltese, por ejemplo, Wikipedia para más información.

Volver al índice

4. Ejercicio completo:

Se considera una cara poligonal cuya matriz de vértices $P = (P_0 P_1 P_2 P_3 P_4 P_5)$ en el sistema de referencia universal viene dada por:

$$P = \begin{pmatrix} 1 & -1 & -2 & -1 & 1 & 2 \\ 1 & 1 & 0 & -1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}_{SRU}$$

Supongamos una cámara situada en el punto $E = \begin{pmatrix} 0 \\ 0 \\ 4 \\ 1 \end{pmatrix}_{SRU}$, mirando hacia el origen de coordenadas y orientación dada por el vector $Up = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}_{SRU}$.

a) Averiguar cuánto debe ser, como mínimo, el campo de visión de la cámara para que el view frustum contenga a dicha cara.

Recuerda: Para que el view frustum contenga a un vértice cuyas coordenadas en el SRC son $\begin{pmatrix} x_{cam} \\ y_{cam} \\ z_{cam} \\ 1 \end{pmatrix}_{SRC}$, debe verificarse:

- $n \geq z_{cam} \geq f$
- $\frac{fov_x}{2} \geq arctg\left(\frac{|x_{cam}|}{|z_{cam}|}\right)$
- $\frac{fov_y}{2} \geq arctg\left(\frac{|y_{cam}|}{|z_{cam}|}\right)$

Por tanto, que contenga a TODOS los vértices, debe verificarse:

- $n \geq z_{cam} \geq f$, para todos los vértices.
- $\frac{fov_x}{2} \geq max\left\{arctg\left(\frac{|x_{cam}|}{|z_{cam}|}\right), \text{ para todos los vértices}\right\} \rightarrow fov_x \geq 2max\left\{arctg\left(\frac{|x_{cam}|}{|z_{cam}|}\right),\right.$
- $\frac{fov_y}{2} \geq max\left\{arctg\left(\frac{|y_{cam}|}{|z_{cam}|}\right), \text{ para todos los vértices}\right\} \rightarrow fov_y \geq 2max\left\{arctg\left(\frac{|y_{cam}|}{|z_{cam}|}\right),\right.$

Por tanto, en primer lugar tenemos que calcular las coordenadas de los vértices de P en el SRC:

In [8]:
```
E = matrix([[0],[0],[4],[1]])
C = matrix([[0],[0],[0],[1]])
At = vector_puntos(E,C)
Up = matrix([[0],[1],[0],[0]])
Mcam = transf_camara(E,At,Up)
P = matrix([[1,-1,-2,-1,1,2],[1,1,0,-1,-1,0],[0,0,0,0,0,0],[1,1,1,1,1,1]])
Pcam = Mcam*P
show('Vértices P en el SRC =', Pcam)
```

$$\text{Vértices P en el SRC} = \begin{pmatrix} 1 & -1 & -2 & -1 & 1 & 2 \\ 1 & 1 & 0 & -1 & -1 & 0 \\ -4 & -4 & -4 & -4 & -4 & -4 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

En segundo lugar, calculamos:

- $M_x = 2max\left\{arctg\left(\frac{|x_{cam}|}{|z_{cam}|}\right), \text{ para todos los vértices}\right\}$
- $M_y = 2max\left\{arctg\left(\frac{|y_{cam}|}{|z_{cam}|}\right), \text{ para todos los vértices}\right\}$

In [9]:
```
Mx = 2*max(arctan(abs(Pcam[0,j]/Pcam[2,j])) for j in range(Pcam.ncols()))
My = 2*max(arctan(abs(Pcam[1,j]/Pcam[2,j])) for j in range(Pcam.ncols()))

#Pasamos de radianes a grados
Mx = (Mx*180/pi).n()
My = (My*180/pi).n()

show('fovx >= ', Mx, ' grados')
show('fovy >= ', My, ' grados')
```

$$\texttt{fovx >=}53.1301023541560 \texttt{ grados}$$

$$\texttt{fovy >=}28.0724869358530 \texttt{ grados}$$

Por lo tanto, el mínimo ángulo de visión debe ser el mayor de los anteriores:

In [10]:
```
show('fov >=', max(Mx, My), ' grados')
```

$$\texttt{fov >=}53.1301023541560 \texttt{ grados}$$

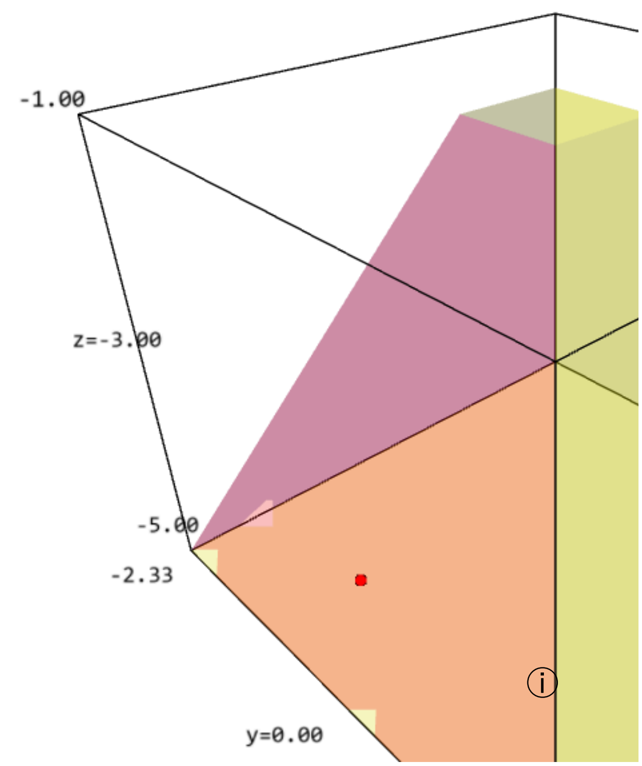
b) Suponiendo un campo de visión $fov = 50^o$ y planos near y far dados por $n = -1$ y $f = -5$, comprobar que los vértices P_2 y P_5 se encuentran fuera del view frustum.

Vamos a hacerlo de 3 formas:

- **Forma 1:** Comprobación visual.
- **Forma 2:** Comprobando que para los vértices P_2 y P_5 no se verifica alguna de la condiciones siguientes:
  - $-1 \geq z_{cam} \geq -5$
  - $\frac{fov}{2} = 25 \geq arctg\left(\frac{|x_{cam}|}{|z_{cam}|}\right)$
  - $\frac{fov}{2} = 25 \geq arctg\left(\frac{|y_{cam}|}{|z_{cam}|}\right)$

- **Forma 3:** Obteniendo las coordenadas de P_2 y P_5 en el volumen canónico y, comprobando que alguna de ellas no está entre -1 y 1.

In [11]:
```
#########################################################
# Forma 1:
#########################################################
n = -1
f = -5
fov = (50*pi/180).n() #pasamos los grados a radianes
r = abs(n)*tan(fov/2)
l = -r
t = r
b = l

vf = view_frustum(l,r,b,t,n,f)
P2 = dibujar_punto(Pcam[:,2],size = 20,color = 'red')
P5 = dibujar_punto(Pcam[:,5],size = 20,color = 'red')
show(vf + P2 + P5)
```



In [12]:
```
#########################################################
# Forma 2:
#########################################################

print('¿Está la tercera coordenada entre n y f?')
show(n >= Pcam[2,2] >= f)
show(n >= Pcam[2,5] >= f)

print('¿Es fov/2 mayor o igual que el ángulo necesario para que P2 esté contenido?')
show(fov/2 >= arctan(abs(Pcam[0,2])/abs(Pcam[2,2])).n())
show(fov/2 >= arctan(abs(Pcam[1,2])/abs(Pcam[2,2])).n())

print('¿Es fov/2 mayor o igual que el ángulo necesario para que P5 esté contenido?')
```

```
show(fov/2 >= arctan(abs(Pcam[0,5])/abs(Pcam[2,5])).n())
show(fov/2 >= arctan(abs(Pcam[1,5])/abs(Pcam[2,5])).n())
```

`¿Está la tercera coordenada entre n y f?`

$$\mathrm{True}$$

$$\mathrm{True}$$

`¿Es fov/2 mayor o igual que el ángulo necesario para que P2 esté contenido?`

$$\mathrm{False}$$

$$\mathrm{True}$$

`¿Es fov/2 mayor o igual que el ángulo necesario para que P5 esté contenido?`

$$\mathrm{False}$$

$$\mathrm{True}$$

In [13]:
```
####################################################
# Forma 3:
####################################################

M = transf_morth_fov(fov,fov,n,f)*transf_persp(n,f)
P2caal = M*Pcam[:,2]
P5caal = M*Pcam[:,5]
show('Coordenadas de P2 en el volumen canónico =', normalizar(P2caal))
show('Coordenadas de P5 en el volumen canónico =', normalizar(P5caal))
```

$$\texttt{Coordenadas de P2 en el volumen canónico =} \left(\begin{array}{r} -1.07225346025478 \\ -0.000000000000000 \\ -0.875000000000000 \\ 1.00000000000000 \end{array}\right)$$

$$\texttt{Coordenadas de P5 en el volumen canónico =} \left(\begin{array}{r} 1.07225346025478 \\ -0.000000000000000 \\ -0.875000000000000 \\ 1.00000000000000 \end{array}\right)$$

Observamos que fallan las coordenadas x de los transformados de P_2 y de P_5, menor que -1 y mayor que 1, respectivamente.

c) Suponiendo una pantalla de 600×400 píxeles, calcular la coordenadas en píxeles del punto Q resultante de proyectar el baricentro del triángulo $P_0P_3P_4$ en la pantalla.

In [23]:
```
nx = 600
ny = 400
```

```
P0P3P4 = P.matrix_from_columns([0,3,4])
B_SRU = baricentro(P0P3P4)
show('Baricentro en el SRU: ',B_SRU)
Bscreen = normalizar(transf_vp(nx,ny)*transf_morth(l,r,b,t,n,f)*transf_persp(n,f)*Mcam
show('Coordenadas en pantalla del transformado del Baricentro: ', Bscreen.n())
show('El baricentro se proyecta en el pixel ', dar_pixel(Bscreen.n()))
```

$$\texttt{Baricentro en el SRU:}\begin{pmatrix} \frac{1}{3} \\ -\frac{1}{3} \\ 0 \\ 1 \end{pmatrix}$$

$$\texttt{Coordenadas en pantalla del transformado del Baricentro:}\begin{pmatrix} 353.11267301273 \\ 163.75821799150 \\ -0.875000000000000 \\ 1.00000000000000 \end{pmatrix}$$

$$\texttt{El baricentro se proyecta en el pixel}\,[353, 164]$$

d) Comprueba que efectivamente, los vértices P_2 y P_5 se salen de la pantalla al proyectar.

In [24]:
```
P2screen = normalizar(transf_vp(nx,ny)*transf_morth(l,r,b,t,n,f)*transf_persp(n,f)*Pca
P5screen = normalizar(transf_vp(nx,ny)*transf_morth(l,r,b,t,n,f)*transf_persp(n,f)*Pca
```

In [25]:
```
show('P2 se proyecta en el pixel ', dar_pixel(P2screen.n()))
show('P5 se proyecta en el pixel ', dar_pixel(P5screen.n()))
```

$$\texttt{P2 se proyecta en el pixel}\,[-22, 200]$$

$$\texttt{P5 se proyecta en el pixel}\,[621, 200]$$

Se puede observar que, en ambos casos, se sale de la pantalla; P_2 por la izquierda y P_5 por la derecha.

e) Calcula el punto de la cara poligonal P que se proyectaría en el pixel $[460, 307]$ de la pantalla y tiene componente de profundidad $z = -0.875$.

Calculamos la matriz de la transformación inversa y definimos el punto $Q = \begin{pmatrix} 460 \\ 307 \\ -0.875 \\ 1 \end{pmatrix}$.

In [26]:
```
M_inv = transf_camara_inv(E,At,Up)*transf_persp_inv(n,f)*transf_morth_inv(l,r,b,t,n,f)
Q = matrix([[460,307,-0.875,1]]).transpose()
```

In [27]:
```
show('Q_SRU = ', normalizar(M_inv*Q))
```

$$\texttt{Q\_SRU =} \left(\begin{array}{r} 0.997898388451698 \\ 1.00256146503325 \\ -0.000000000000000 \\ 1.00000000000000 \end{array}\right)$$

Por tanto, el pixel $[460, 307]$ se corresponde con el vértice $P_0 = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \end{pmatrix}_{SRU}$ de la cara poligonal (observar que variación se debe a los errores de redondeo).

Volver al índice

In [ ]: