

# Cámara

- ☐ Matriz de modelado y matriz de vista
- ☐ Transformaciones: relativas al sistema global o a la propia cámara
- ☐ Desplazamiento en cada uno de sus ejes
- ☐ Cambios en la dirección de vista
- ☐ Proyecciones: ortogonal y perspectiva
- ☐ Varios puertos de vista

Ana Gil Luezas  
Departamento de Sistemas Informáticos y Computación  
Facultad de Informática  
Universidad Complutense de Madrid

## Posición y orientación de la cámara

- ❑ Para colocar la cámara podemos establecer, en coordenadas cartesianas (globales), un punto para su posición (**eye**), el punto al que mira (**look**) y la inclinación (**up**):

```
glm::dvec3 mEye, mLook, mUp; // Atributos de la clase Camera
```

**eye**, **look** y **up** definen un marco de coordenadas: el marco de la cámara, o la matriz de modelado de la cámara.

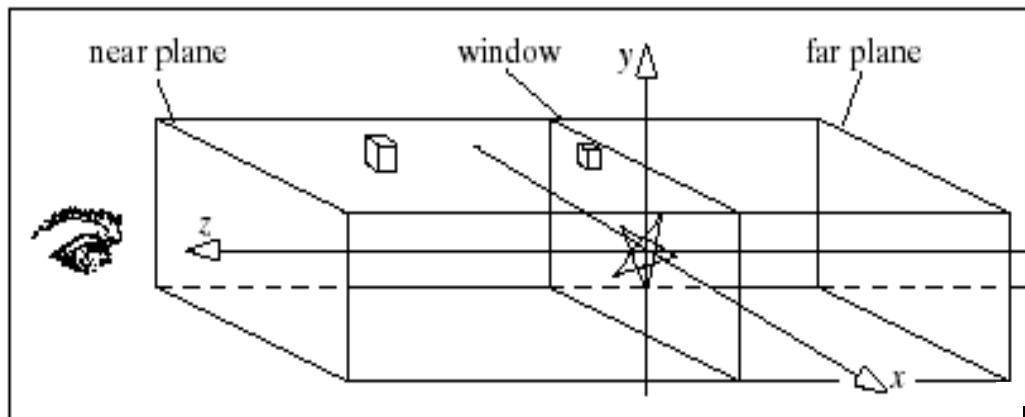
- ❑ **lookAt(eye,look,up)** genera la matriz de vista: la inversa de la matriz de modelado de la cámara

```
glm::dmat4 mViewMat = glm::lookAt(mEye, mLook, mUp);  
// Atributo de la clase Camera
```

# Posición y orientación de la cámara

Ejemplos: Los argumentos *eye*, *look* y *up* se dan en coordenadas globales.

```
mViewMat = lookAt( mEye, mLook, mUp);
```



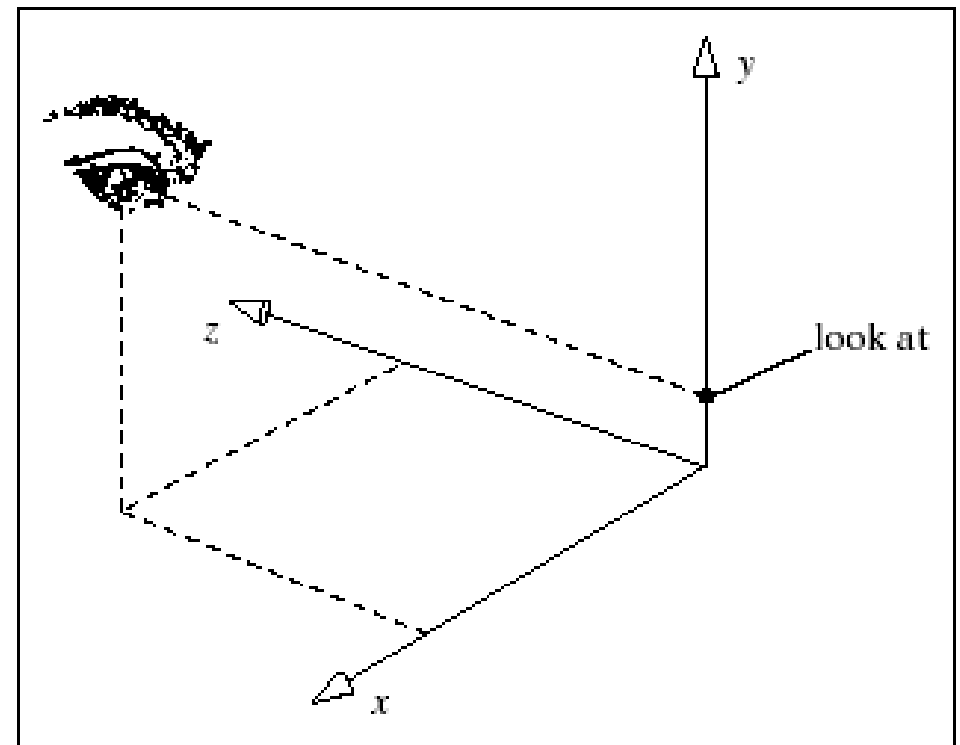
**set2D() (vista Frontal):**

```
mEye = dvec3(0, 0, 500);  
mLook = dvec3(0, 0, 0);  
mUp = dvec3(0, 1, 0);
```

**set3D():**

```
mEye = dvec3(100, 100, 100);  
mLook = dvec3(0, 10, 0);  
mUp = dvec3(0, 1, 0);
```

¿Vista cenital?



## Marco de la cámara

□ **eye**, **look** y **up** definen el marco de coordenadas  $M_c = (u, v, n, e)$  dado por:

**n (z)** = normalize(**eye** – **look**)      // **-n** es la dirección de vista (front)

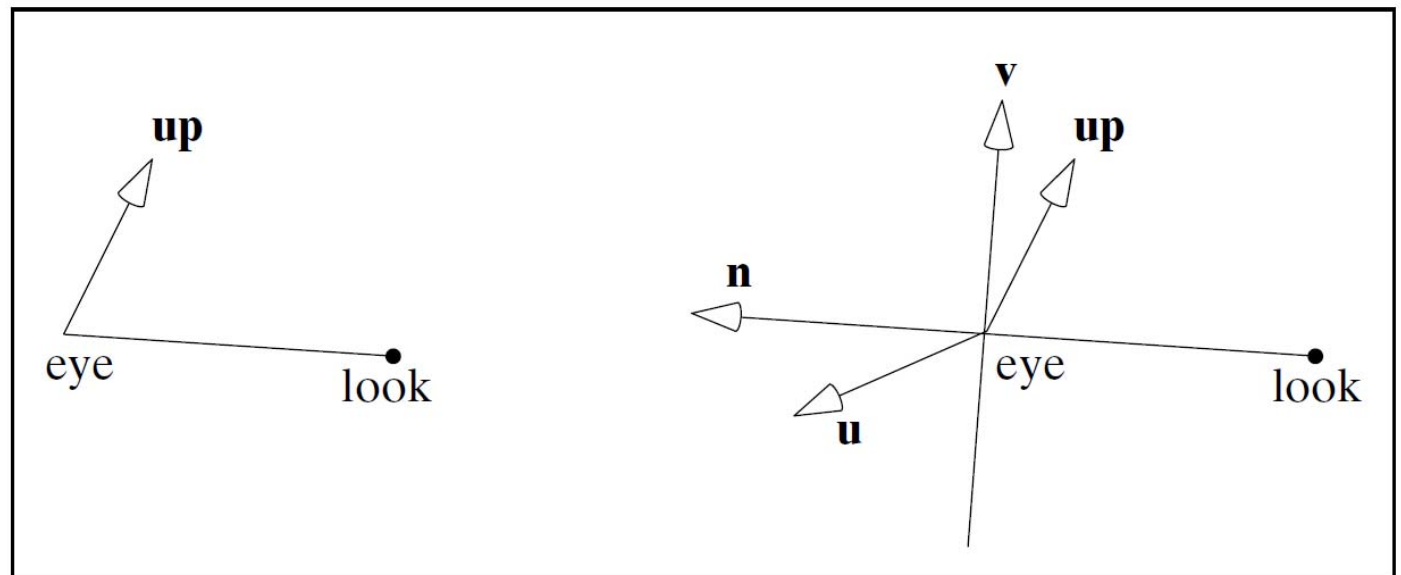
**u (x)** = normalize(cross(**up**, **n**))    // ortogonal a **up** y **n** (right)

**v (y)** = normalize(cross(**n**, **u**))    // ortogonal a **n** y **u** (upward)

**e** = **eye**

Matriz del marco ( $M_c$ )

$$\begin{pmatrix} ux & vx & nx & ex \\ uy & vy & ny & ey \\ uz & vz & nz & ez \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



$$\text{cross}(\mathbf{a}, \mathbf{b}) = -\text{cross}(\mathbf{b}, \mathbf{a}) =$$

$$\begin{vmatrix} i & j & k \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix} = \begin{pmatrix} a_y b_z - a_z b_y \\ a_z b_x - a_x b_z \\ a_x b_y - a_y b_x \end{pmatrix}$$

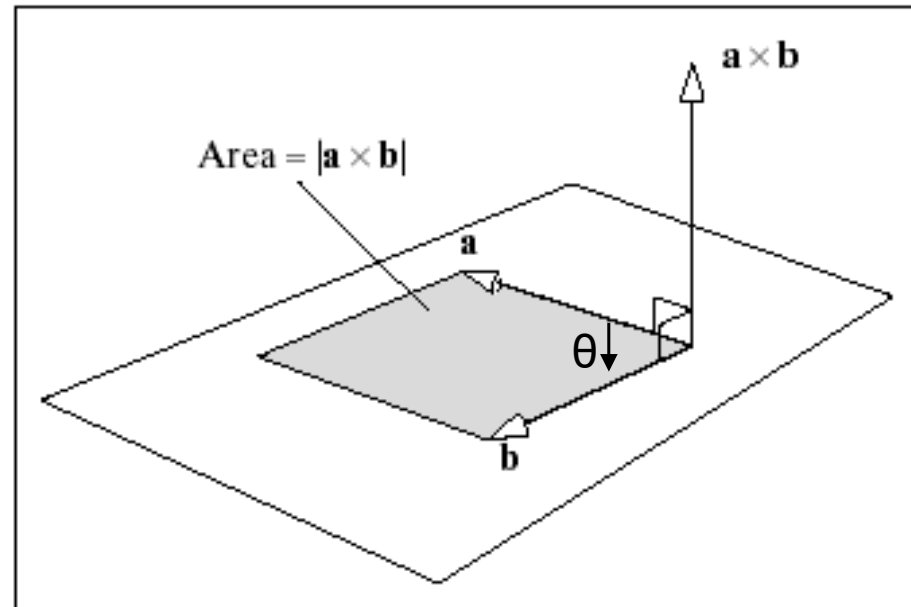
$$|\text{cross}(\mathbf{a}, \mathbf{b})| = |\mathbf{a}| |\mathbf{b}| \sin \theta$$

$$\theta = 0 \rightarrow \sin(0) = 0 \rightarrow \mathbf{a} \parallel \mathbf{b} \rightarrow \text{Error}$$

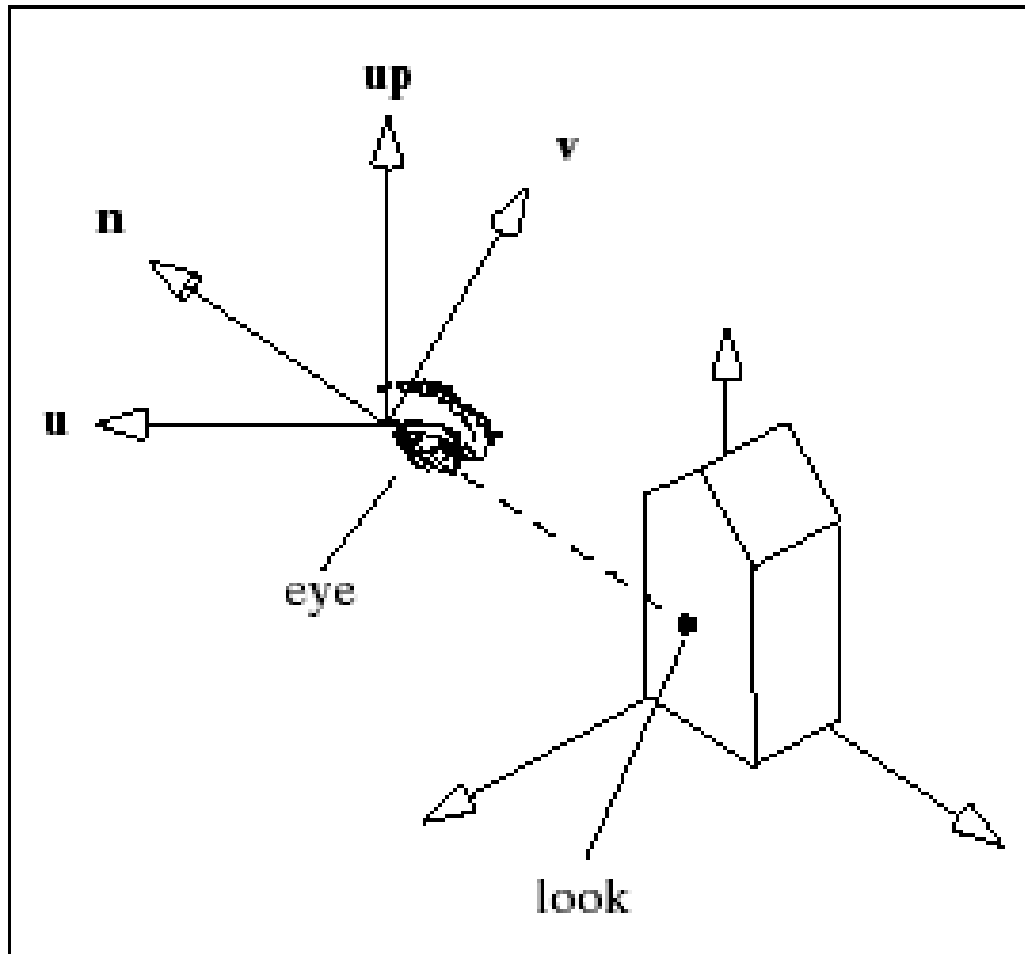
Producto escalar:

$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}| |\mathbf{b}| \cos \theta$$

$$\theta = 90 \rightarrow \cos(90) = 0 \rightarrow \mathbf{a} \perp \mathbf{b}$$



Regla de la mano derecha  
o del sacacorchos



- ❑ La cámara mira hacia  **$-\mathbf{n}$** .
- ❑ El sistema  **$(\mathbf{u}, \mathbf{v}, \mathbf{n})$**  es **ortonormal** : vectores ortogonales de magnitud uno  
->

la inversa de la matriz 3x3  
 **$(\mathbf{u}, \mathbf{v}, \mathbf{n})$**  es la traspuesta.

$$M_c = \begin{pmatrix} \begin{matrix} ux & vx & nx \\ uy & vy & ny \\ uz & vz & nz \end{matrix} & \begin{matrix} ex \\ ey \\ ez \end{matrix} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- La matriz de vista ( $V = \text{viewMat} = \text{lookAt}(\text{eye}, \text{look}, \text{up});$ ) es la inversa del marco de la cámara ( $M_c$ )

$$V = \begin{pmatrix} ux & uy & uz & dx \\ vx & vy & vz & dy \\ nx & ny & nz & dz \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

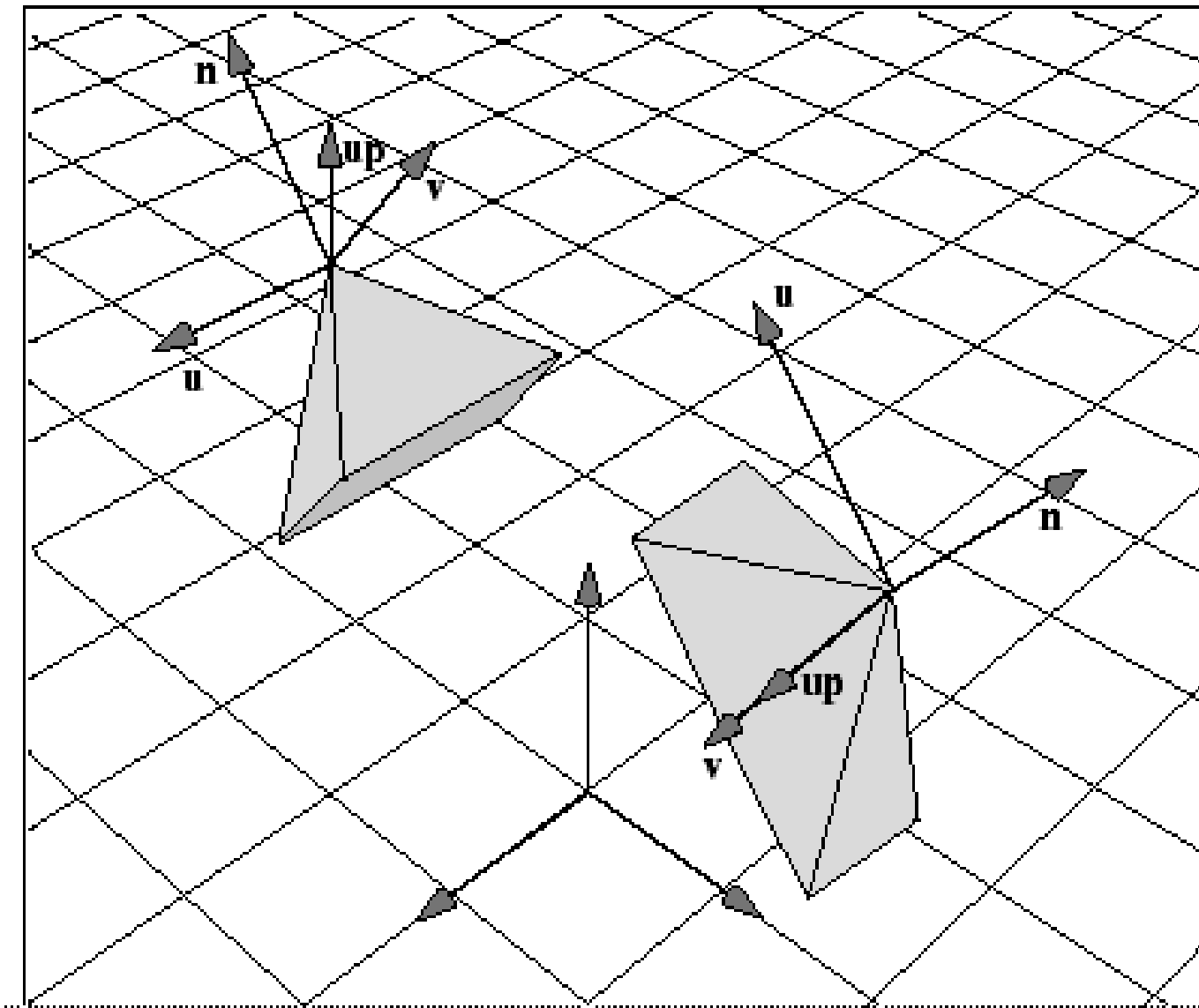
donde  $d = (-e \cdot u, -e \cdot v, -e \cdot n)$

$$M_c = \begin{pmatrix} ux & vx & nx & ex \\ uy & vy & ny & ey \\ uz & vz & nz & ez \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

// Atributo de la clase Camera  
// Ejes de la cámara

$\text{mRight} = \text{row}(\text{mViewMat}, 0);$   
 $\text{mUpward} = \text{row}(\text{mViewMat}, 1);$   
 $\text{mFront} = -\text{row}(\text{mViewMat}, 2);$

¿Relativas al sistema global o a la propia cámara?





¿Relativas al sistema global o a la propia cámara?

- ❑ **Ejemplo:** realizar una trayectoria circular alrededor de la escena, mirando al centro del círculo.

```
eye.x = center.x + cos(radians(ang)) * radius;
```

```
eye.z = center.z + -sin(radians(ang)) * radius;
```

```
viewMat = lookAt(eye, center, dvec3(0, 1, 0));
```

- ❑ **Ejemplo:** Movimiento horizontal en el eje X del sistema global

```
eye.x += incX;    -> Cambia la dirección de vista (-n = look - eye)
```

```
viewMat = lookAt(eye, look, up);
```

- ❑ **Ejemplo:** Movimiento horizontal en el eje **u** del sistema de la cámara

```
eye += u * cs;    -> Cambia la dirección de vista (-n = look - eye)
```

```
viewMat = lookAt(eye, look, up);
```

### Relativos a la propia cámara

□ Mover la cámara **cs** en uno de sus ejes: mover el ojo (**eye**)

- En el eje u:  $\text{eye} += u * \text{cs}$
- En el eje v:  $\text{eye} += v * \text{cs}$
- En el eje n:  $\text{eye} += n * \text{cs}$

También modifica la dirección de vista ( $-\mathbf{n} = \text{look} - \text{eye}$ )

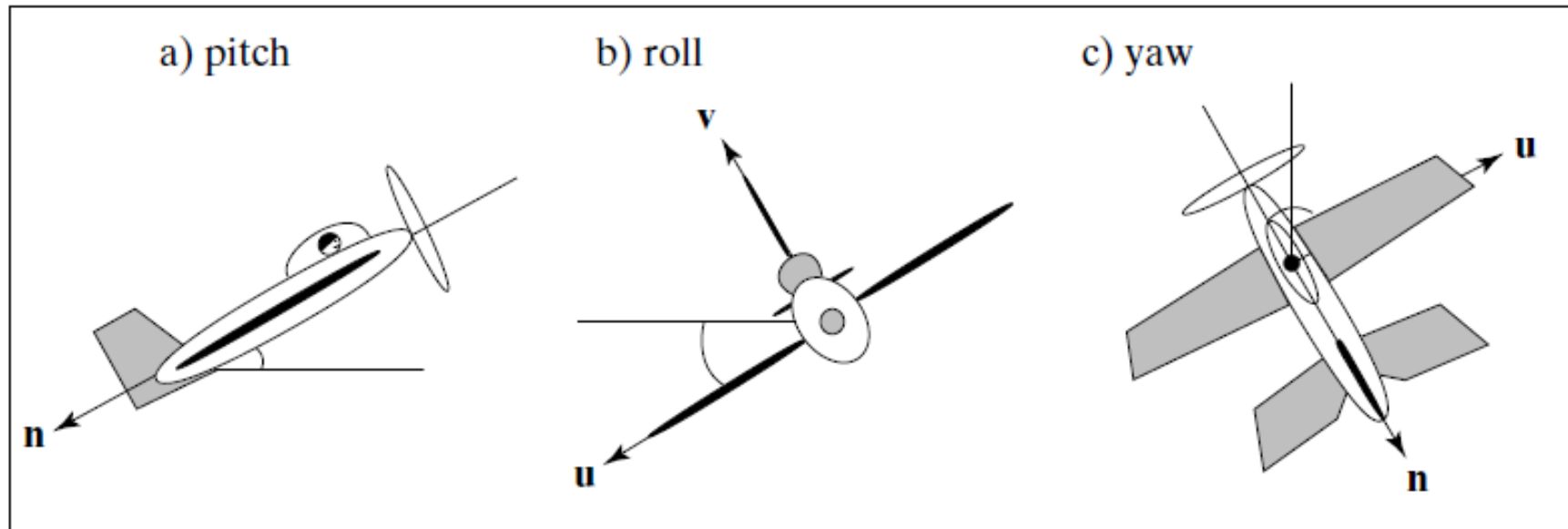
Si no queremos cambiar la dirección de vista ->

mover look de la misma forma.

□ Mover **look cs** en los ejes de la cámara (modifica la dirección de vista)

- En el eje u:  $\text{look} += u * \text{cs}$
- En el eje v:  $\text{look} += v * \text{cs}$
- En el eje n:  $\text{look} += n * \text{cs}$

# Rotaciones de la cámara



Cambios en la dirección de vista (**front**):

**Pitch:** mirar hacia arriba y abajo.

Rotación en el eje  $u$

**Yaw:** mirar a izquierda y derecha.

Rotación en el eje  $v$

**Roll:** Rotación en el eje  $n$

- ❑ En la clase `Camera` añadimos los atributos:

```
dvec3 mRight, mUpward, mFront;
```

```
// para los ejes right=u, upward=v, front=-n
```

Y los métodos :

```
void Camera::setAxes() {  
    mRight = row(mViewMat, 0);  
    mUpward = row(mViewMat, 1);  
    mFront = - row(mViewMat, 2);  
}
```

```
void Camera::setVM() {  
    mViewMat = lookAt(mEye, mLook, mUp);  
    setAxes();  
}
```

### ❑ Desplazamientos en los ejes de la propia cámara

Para desplazar *eye* en los ejes de la cámara, sin cambiar la dirección de vista:

```
void Camera::moveUD(GLdouble cs) { // Up / Down
    mEye += mUpward * cs;
    mLook += mUpward * cs;
    setVM();
}

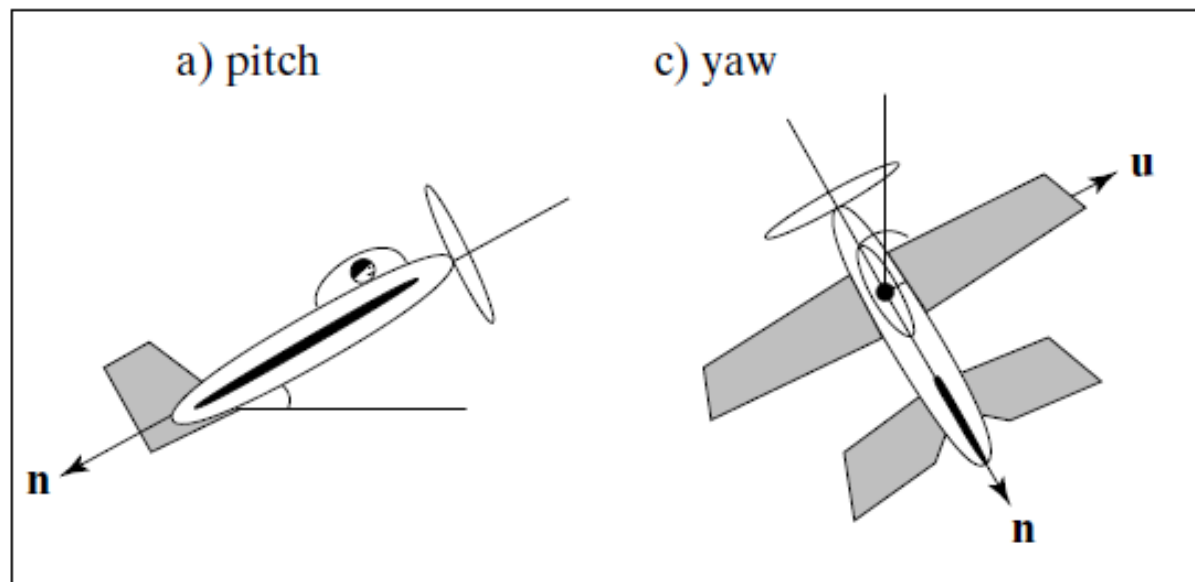
void Camera::moveLR(GLdouble cs) { // Left / Right
    ... }

void Camera::moveFB(GLdouble cs) { // Forward / Backward
    ... }
```

- ❑ Para modificar la dirección de vista (**front**) podemos desplazar **look** en los ejes de la cámara u y v:

```
void Camera::lookUD(GLdouble cs) { // Up / Down  
    mLook += mUpward * cs;  
    setVM();  
}
```

```
void Camera::lookLR(GLdouble cs) { // Left / Right  
    ... }
```



## Transformaciones sobre el sistema global

### ❑ Transformaciones relativas al sistema global

Queremos realizar con la cámara una **trayectoria circular alrededor de look**

Añadimos a la **clase Camera** atributos para gestionar el **radio** y el ángulo de la circunferencia: **GLdouble mAng;**

Y definimos un método para desplazar **eye** en la circunferencia, a la vez que se permite subir y bajar la cámara

```
void orbit (GLdouble incAng, GLdouble incY) {  
    mAng += incAng;  
    mEye.x = mLook.x + cos(radians(mAng)) * mRadio;  
    mEye.z = mLook.z - sin(radians(mAng)) * mRadio;  
    mEye.y += incY;  
    setVM();  
}
```

- ❑ Podemos utilizar el **ratón** para las transformaciones de la **cámara**
  - Define las **funciones static** para los callback de los **eventos del ratón**  
`glutMouseFunc(s_mouse); // cuando se presiona o suelta un botón`  
`glutMotionFunc(s_motion); // cuando se mueve con un botón presionado`  
`glutMouseWheelFunc(s_mouseWheel); // cuando se gira una rueda`
  - Y los correspondientes **métodos**  
`void mouse(int button, int state, int x, int y);`  
`void motion(int x, int y);`  
`void mouseWheel(int wheelNumber, int direction, int x, int y);`



```
void mouse(int button, int state, int x, int y):  
    // state indica si el botón se ha presionado o soltado: GLUT_UP / _DOWN  
    // button es el botón que se ha presionado o soltado:  
        GLUT_LEFT / _RIGHT_BUTTON  
    // Guardamos el botón del mouse en un atributo de la aplicación  
    mMouseButton = button;  
    // (x, y) es la posición del ratón en la ventana,  
    // siendo (0,0) la esquina (left, top)  
    // Guardamos las coordenadas del mouse en un atributo de la aplicación  
    mMouseCoord = dvec2(x, mWinHeight - y);
```

Nota: Para adecuar las coordenadas del ratón al origen del puerto de vista  
(0, 0) = (left, bottom) hay que invertir la coordenada y:

$$y = mWinHeight - y;$$

- ❑ Orbital la cámara con el botón izquierdo del ratón

```
void motion(int x, int y):
```

```
if (mMouseButt == GLUT_LEFT_BUTTON) {
```

```
    // guardamos la anterior posición en var. temp.
```

```
    dvec2 mp = mMouseCoord ;
```

```
    // Guardamos la posición actual
```

```
    mMouseCoord = dvec2(x, mWinHeight - y);
```

```
    mp = mMouseCoord - mp; // calculamos el desplazamiento realizado
```

```
    camera.orbit(mp.x * 0.05, mp.y); // sensitivity = 0.05
```

```
    glutPostRedisplay();
```

```
}
```

```
else if (mMouseButt == GLUT_RIGHT_BUTTON) {...}
```

- ❑ Desplazar la cámara en la dirección de vista

```
void mouseWheel(int wheelNumber, int direction, int x, int y):
```

```
    // direction es la dirección de la rueda (+1 / -1)
```

```
    if (direction == 1) mCamera.moveFB(5);
```

```
    else mCamera.moveFB(-5);
```

```
    glutPostRedisplay();
```

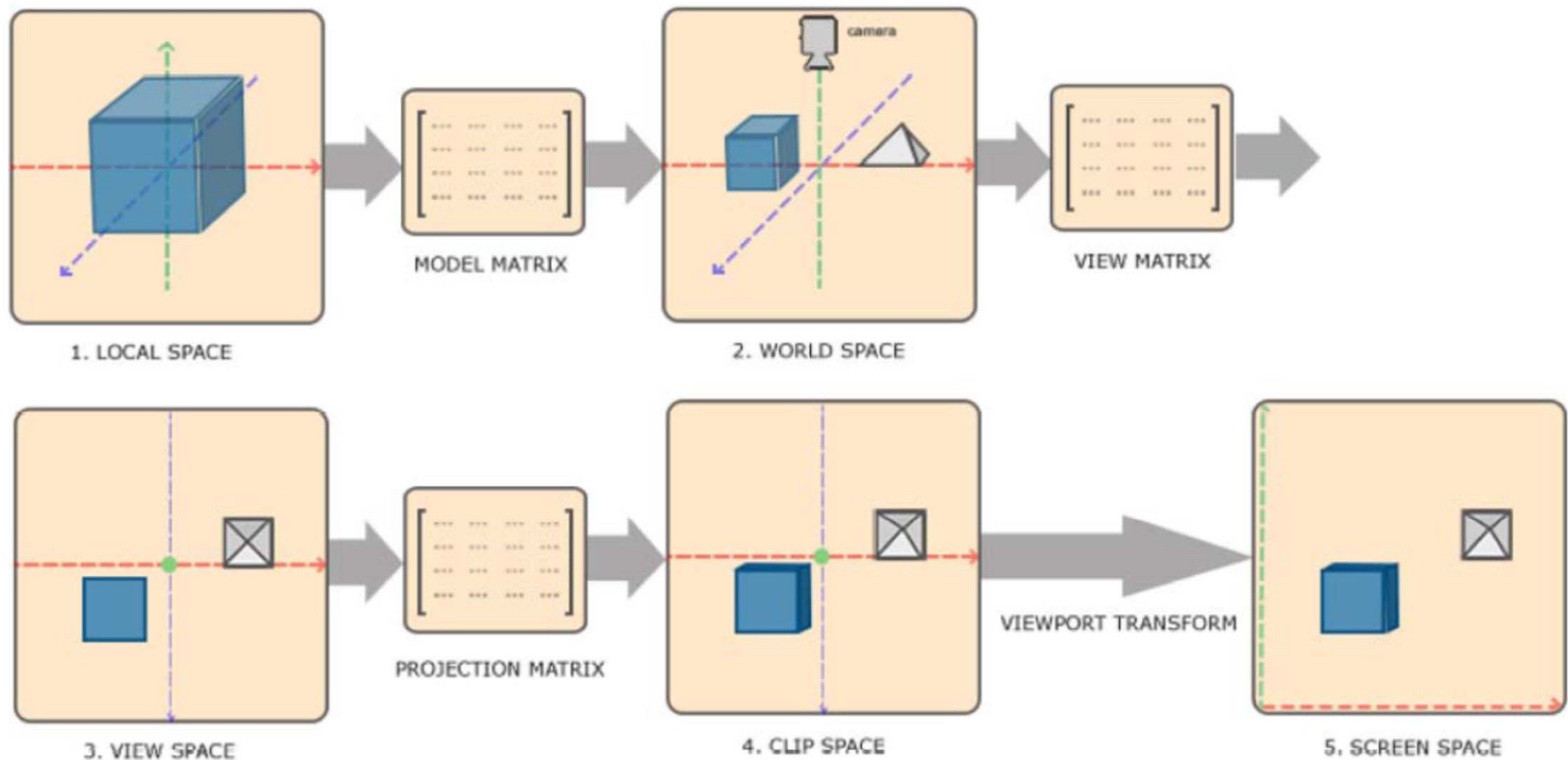
Podemos preguntar si está pulsada alguna de las teclas:

GLUT\_ACTIVE\_CTRL / \_ALT / \_SHIFT

```
int m = glutGetModifiers();
```

```
if (m == 0) // ninguna está presionada
```

# Transformaciones

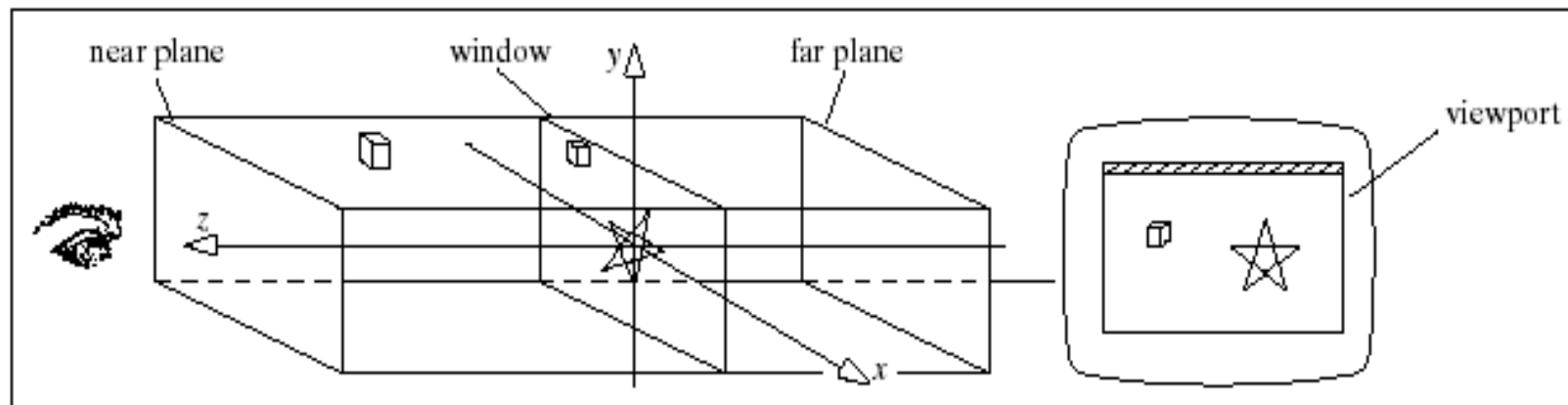


```
void Entity::render(dmat4 const& modelViewMat){  
    dmat4 aMat = modelViewMat * mModelMat;  
    upload(aMat);  
    mesh -> render();  
}
```

```
void Camera::uploadPM() {  
    glMatrixMode(GL_PROJECTION);  
    glLoadMatrixd(value_ptr(mProjMat));  
    glMatrixMode(GL_MODELVIEW);  
}
```

## Volumen y plano de vista

- ❑ El **volumen de vista (VV)** se establece con respecto a la cámara.  
El volumen de vista define la **matriz de proyección**.
- ❑ El volumen de vista se delimita por **dos rectángulos (cercano y lejano)** perpendiculares al eje **n**. El plano cercano se asocia con el plano de proyección o **plano (ventana) de vista**.
- ❑ En el **puerto de vista** se mostrarán los objetos que quedan dentro del volumen de vista una vez proyectados sobre el plano de vista.



## Proyección ortogonal y perspectiva

- ❑ Para establecer la **matriz de proyección** (mProjMat):

```
void Camera::uploadPM() {  
    glMatrixMode(GL_PROJECTION);  
    glLoadMatrix(value_ptr(mProjMat));  
    glMatrixMode(GL_MODELVIEW);  
}
```

- ❑ **Ortogonal**: paralelepípedo en coordenadas de la cámara

`mProjMat = ortho(xLeft, xRight, yBottom, yTop, mNear, mFar);`

Ventana de vista

Distancias al ojo

- ❑ **Perspectiva**: pirámide truncada en coordenadas de la cámara

`mProjMat = frustum(xLeft, xRight, yBottom, yTop, mNear, mFar);`

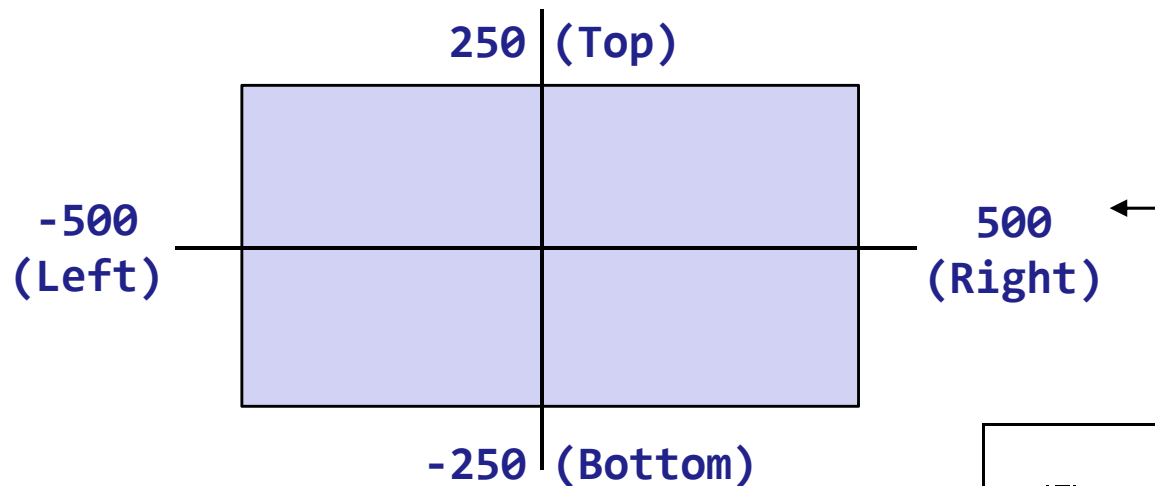
Ventana de vista

Distancias al ojo

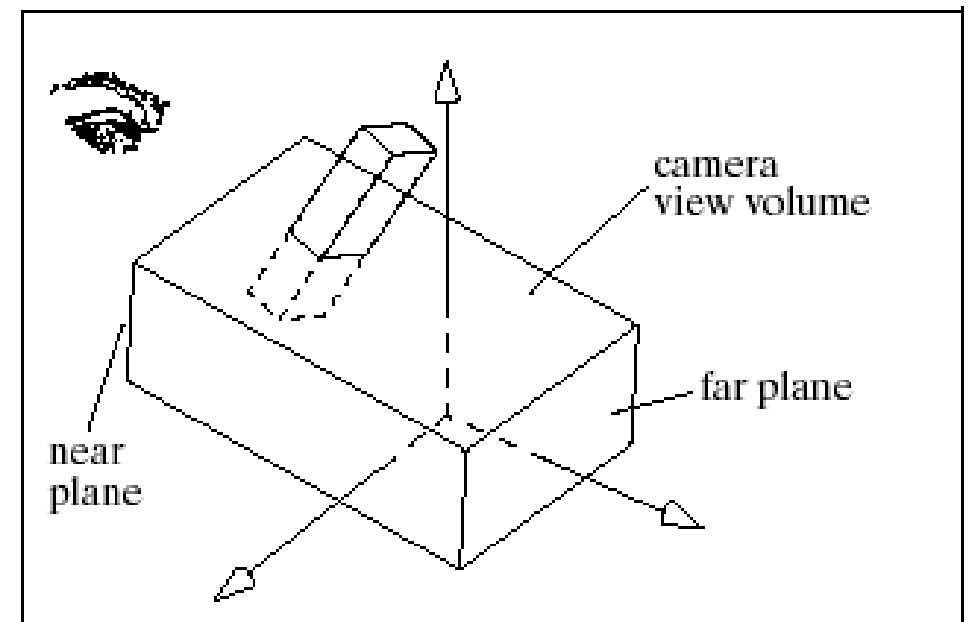
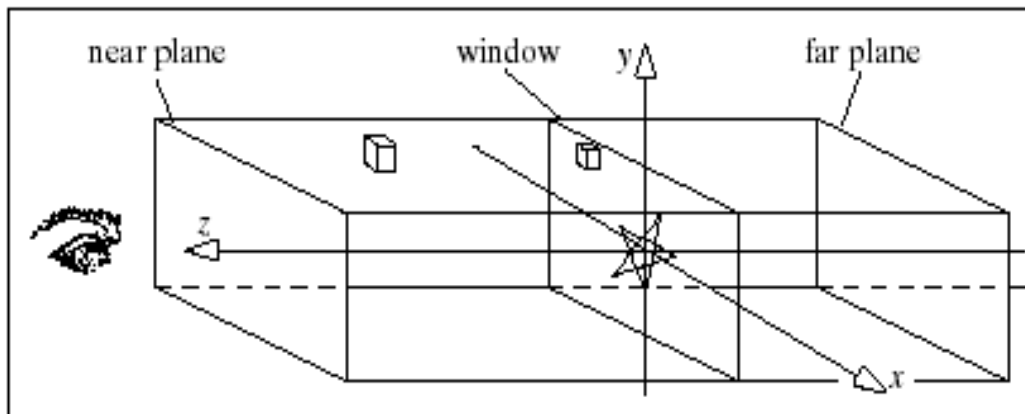
# Proyección ortogonal

```
glm::ortho(Left, Right, Bottom, Top, Near, Far);  
glm::ortho(-500, 500, -250, 250, 500, 10000);
```

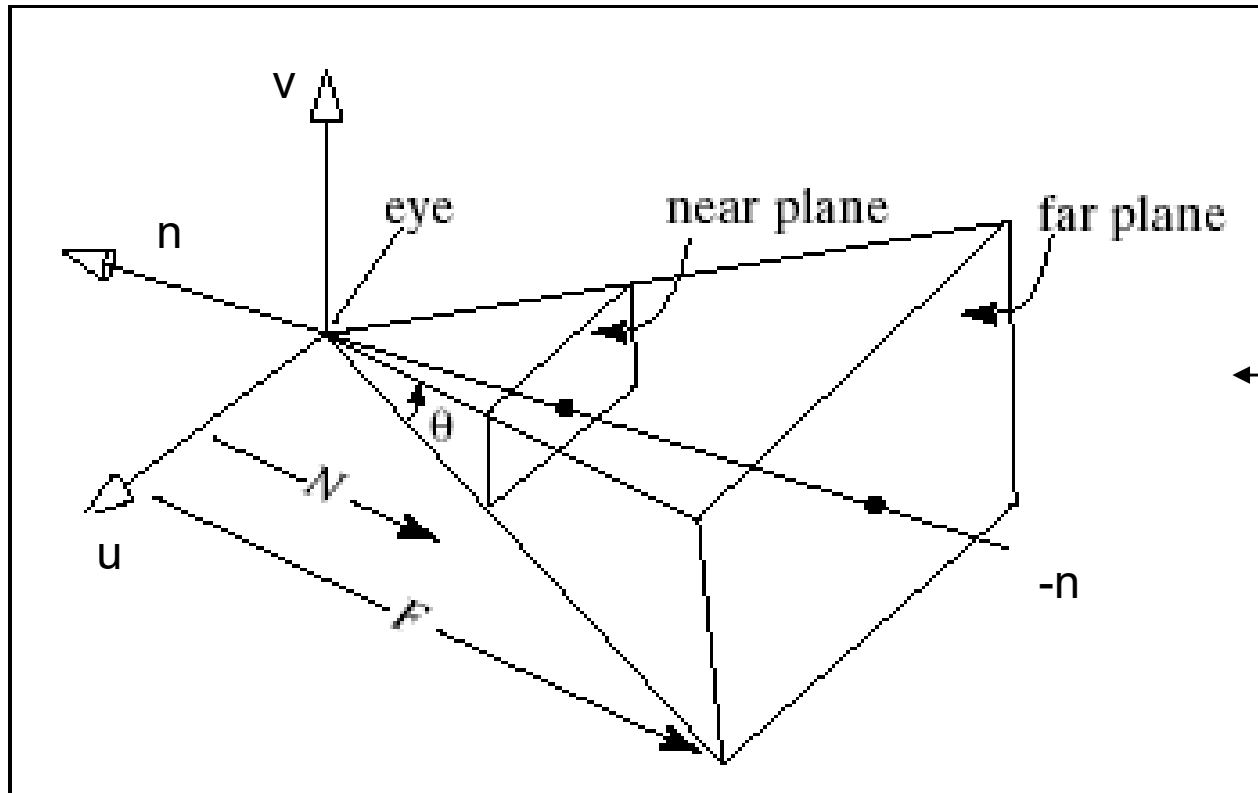
En coordenadas  
de la cámara



Los rectángulos cercano y lejano son iguales y perpendiculares al eje n



```
glm::frustum (-500, 500, -250, 250, 500, 10000);
```



El Rectángulo cercano en coordenadas de la cámara

Rectángulos perpendiculares al eje n

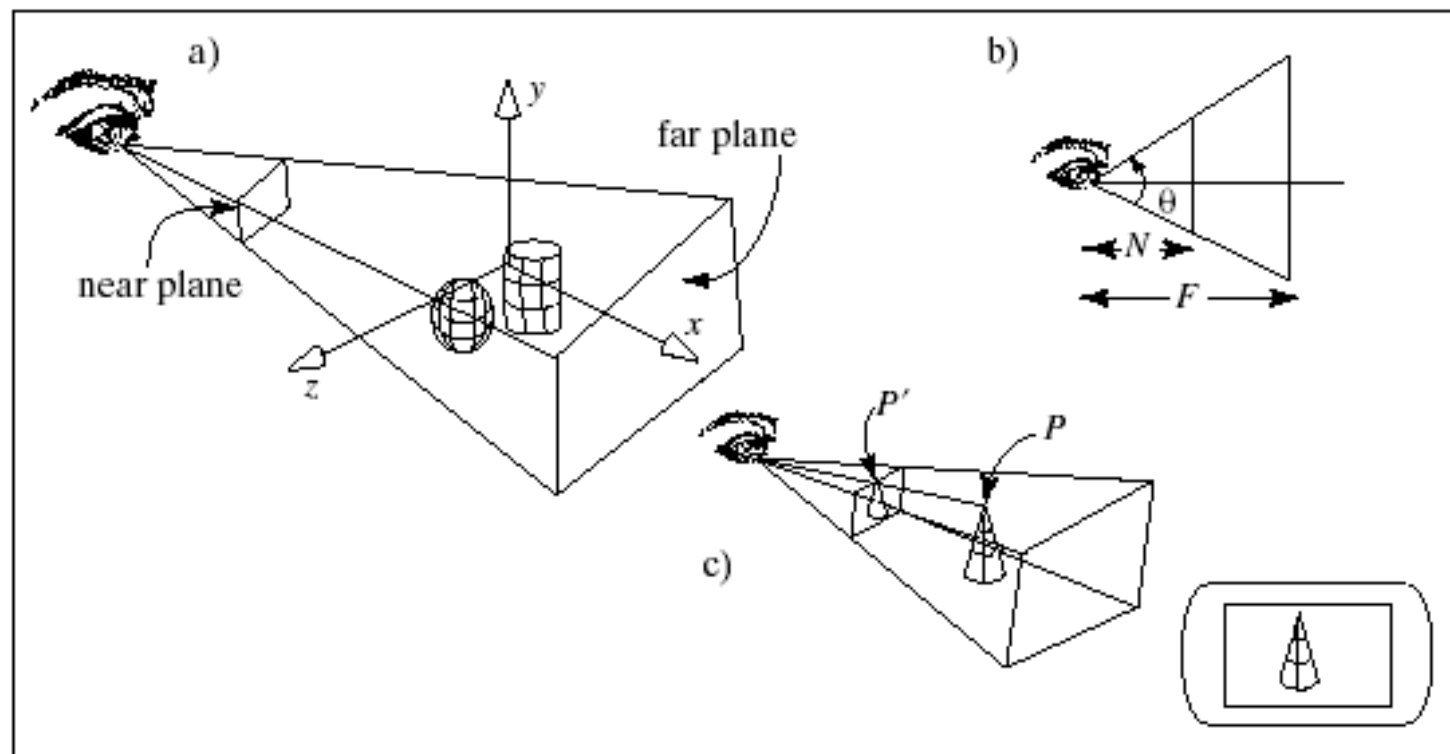
El rectángulo lejano queda definido trazando las líneas de proyección que van desde el ojo, pasando por las cuatro esquinas del rectángulo cercano.

Es necesario que las distancias Far y Near cumplan:  $Far > Near > 0$ .



# Proyección perspectiva

La proyección de un vértice es la intersección con el plano cercano de la línea que va desde el vértice al ojo. Todos los puntos de una línea de proyección proyectan en el mismo punto.



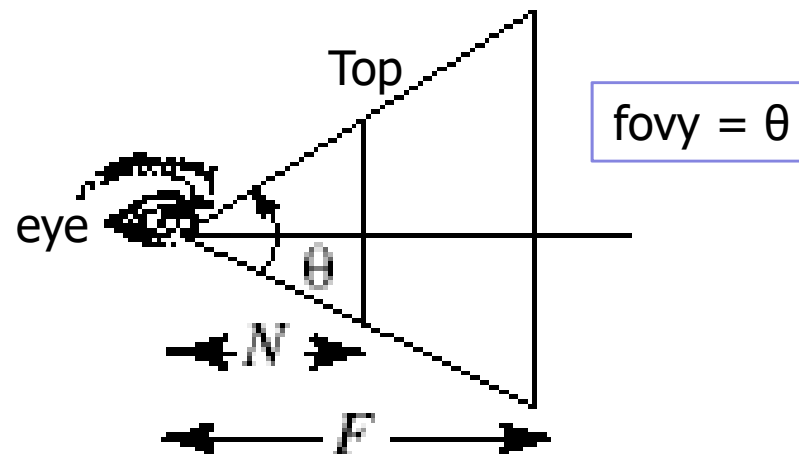
Nota: La matriz de proyección no realiza la proyección completa, deja pendiente la división perspectiva (en la 4ª coordenada  $w$ )

La posición de la cámara (eye), Near y Top establecen el ángulo del campo de visión (**fovy**).

$$\tan(\text{fovy}/2) = \text{Top} / \text{Near}$$

Para  $\text{fovy} = 60$ :  $\tan(30) = 0,5773 \rightarrow \text{Near} = 2 * \text{Top}$

Para  $\text{fovy} = 90$ :  $\tan(45) = 1 \rightarrow \text{Near} = \text{Top}$



También podemos definir volúmenes con perspectiva con la función

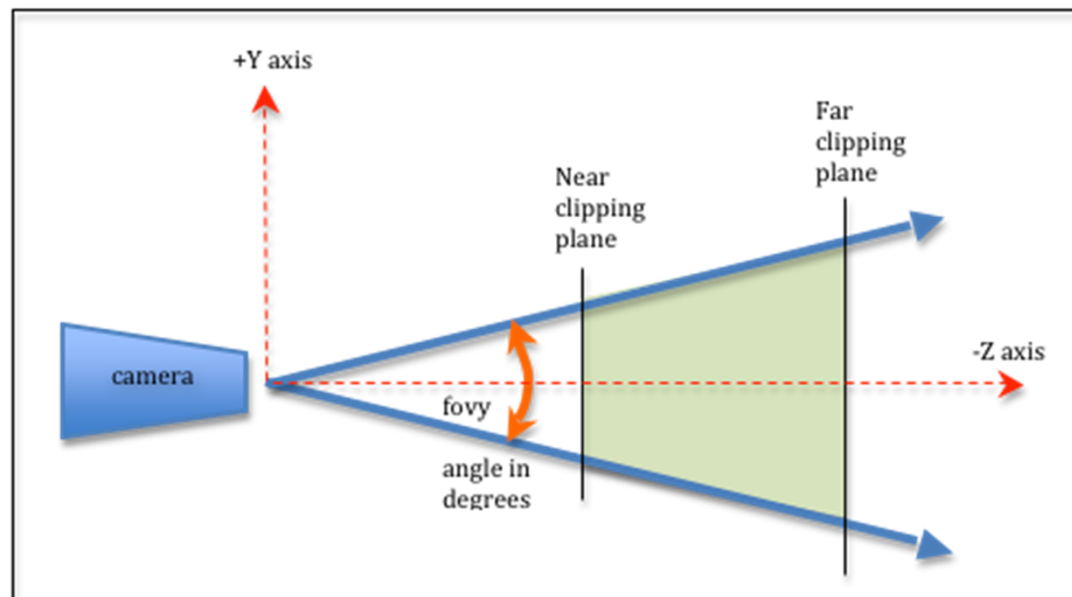
```
glm::perspective(Fovy, AspectRatio, Near, Far);
```

donde  $\text{AspectRatio} = \text{Ancho/Alto}$ , por ejemplo 4/3, 16/9

Equivale a

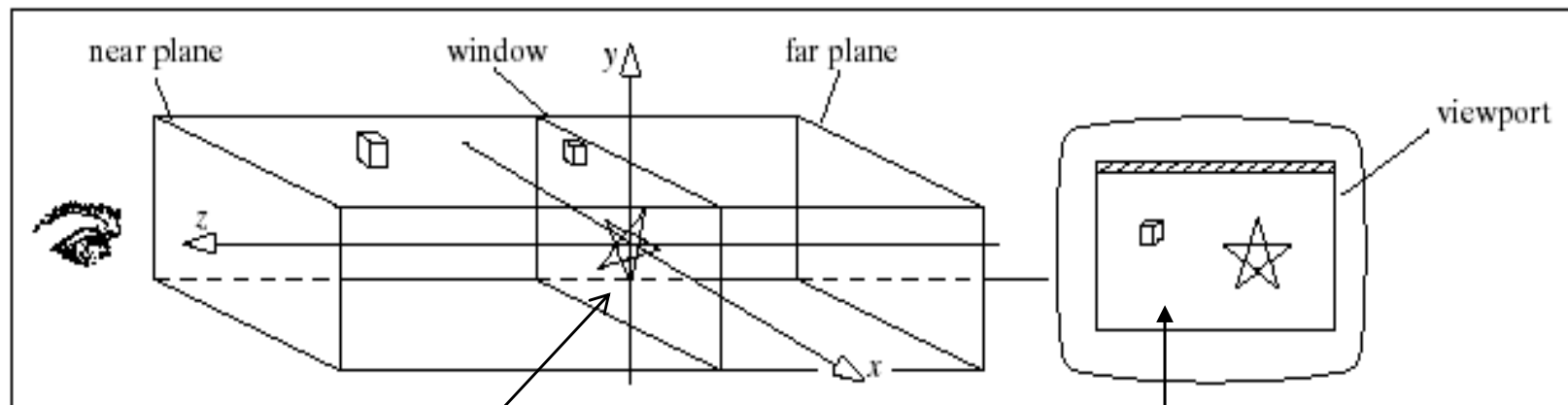
```
glm::frustum(Left, Right, Bottom, Top, Near, Far)
```

con argumentos:

$$\text{Top} = \text{Near} \cdot \tan(\text{Fovy} / 2.0)$$
$$\text{Bot} = -\text{Top}$$
$$\text{Right} = \text{Top} \cdot \text{AspectRatio}$$
$$\text{Left} = -\text{Right}$$


- ❑ La proyección obtenida en la **ventana de vista** se transfiere al **puerto de vista** establecido en la ventana de visualización de la aplicación.

## Proyección ortogonal



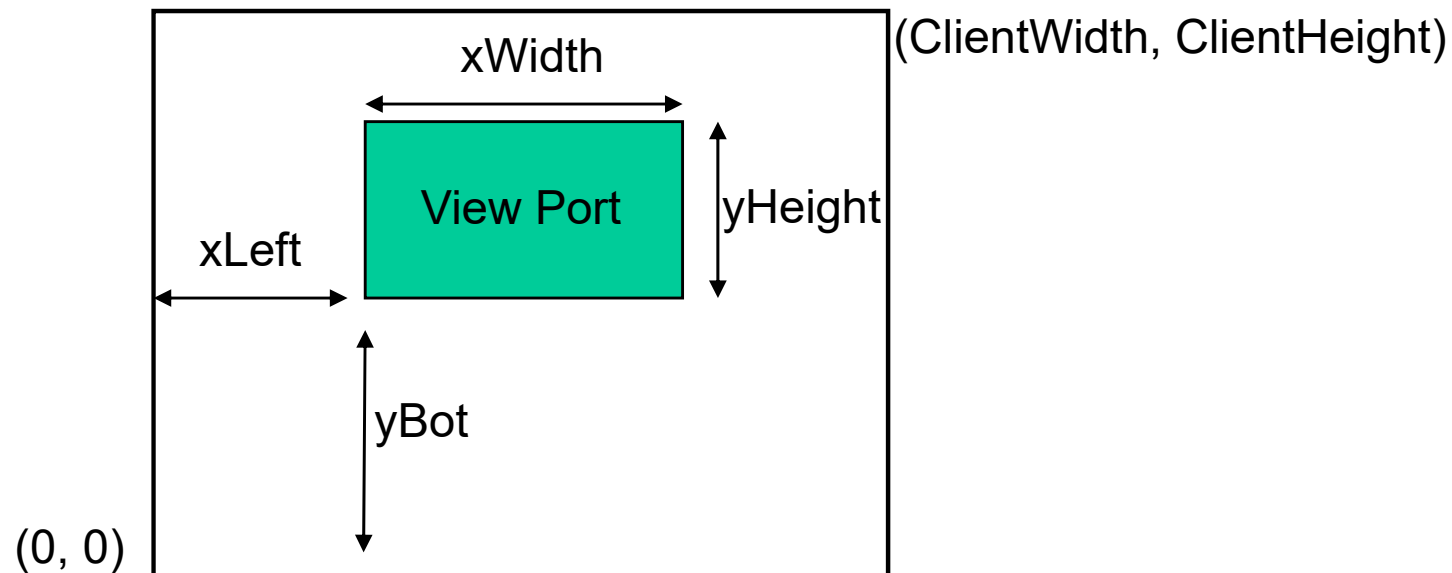
Área visible, ventana  
o plano de vista

Puerto de vista en el  
dispositivo de salida

- El **puerto de vista** es un rectángulo, del área cliente de la ventana, alineado con los ejes. Para fijar el puerto de vista:

```
void Viewport::upload() {  
    glVertexport(xLeft, yBot, xWidth, yHeight);  
}
```

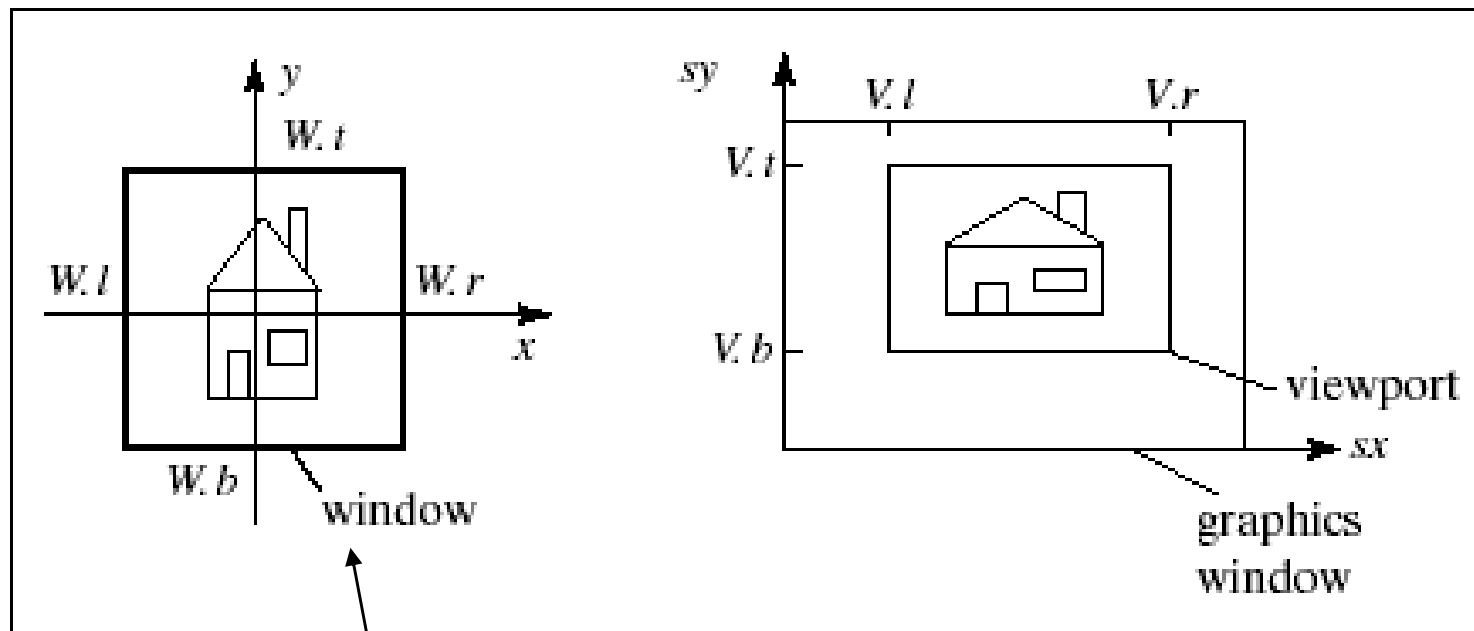
Los parámetros son de tipo entero (píxeles)



- Puerto de vista ocupando toda el área cliente de la ventana:  
`glViewport(0, 0, ClientWidth, ClientHeight);`

## Relación entre el plano de vista y el puerto de vista

- ❑ La relación entre el **puerto de vista** y el **plano de vista** establece una **escala** y una **traslación**. La escala puede deformar la imagen obtenida.
- ❑ Para una escala 1:1 ambos rectángulos deben ser del mismo tamaño.



Área visible, ventana  
o plano de vista

- ```
void resize(int newWidth, int newHeight) { // IG1App  
    mViewPort -> setSize(newWidth, newHeight); // Resize Viewport  
    // Resize Scene Visible Area -> para que no cambie la escala  
    mCamera -> setSize(mViewPort->width(), mViewPort->height());  
}
```
- ```
void key(unsigned char key, int x, int y) { // IG1App  
    ...  
    case '+':  
        mCamera->setScale(+0.01); // zoom in (increases the scale)  
        break;  
    case '-':  
        mCamera->setScale(-0.01); // zoom out (decreases the scale)  
        break;  
    ...  
    glutPostRedisplay(); }
```

❑ Podemos renderizar en varios puertos de vista para mostrar en la misma ventana:

- Diferentes vistas de la misma escena
- Distintas escenas

```
void display() // IG1App
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); //-> una vez

    mScene->render(...); //-> varias veces: cambiando el puerto de vista
                        // y la cámara o la escena (-> ej. display4V)

    glutSwapBuffers(); //-> una vez
}
```



- **Ejemplo:** visualizar 4 vistas de la misma escena. Las vistas son: 2D, 3D, Cenital y la de la cámara que maneja el usuario con los eventos del ratón (el atributo **mCamera** de **IG1App**).

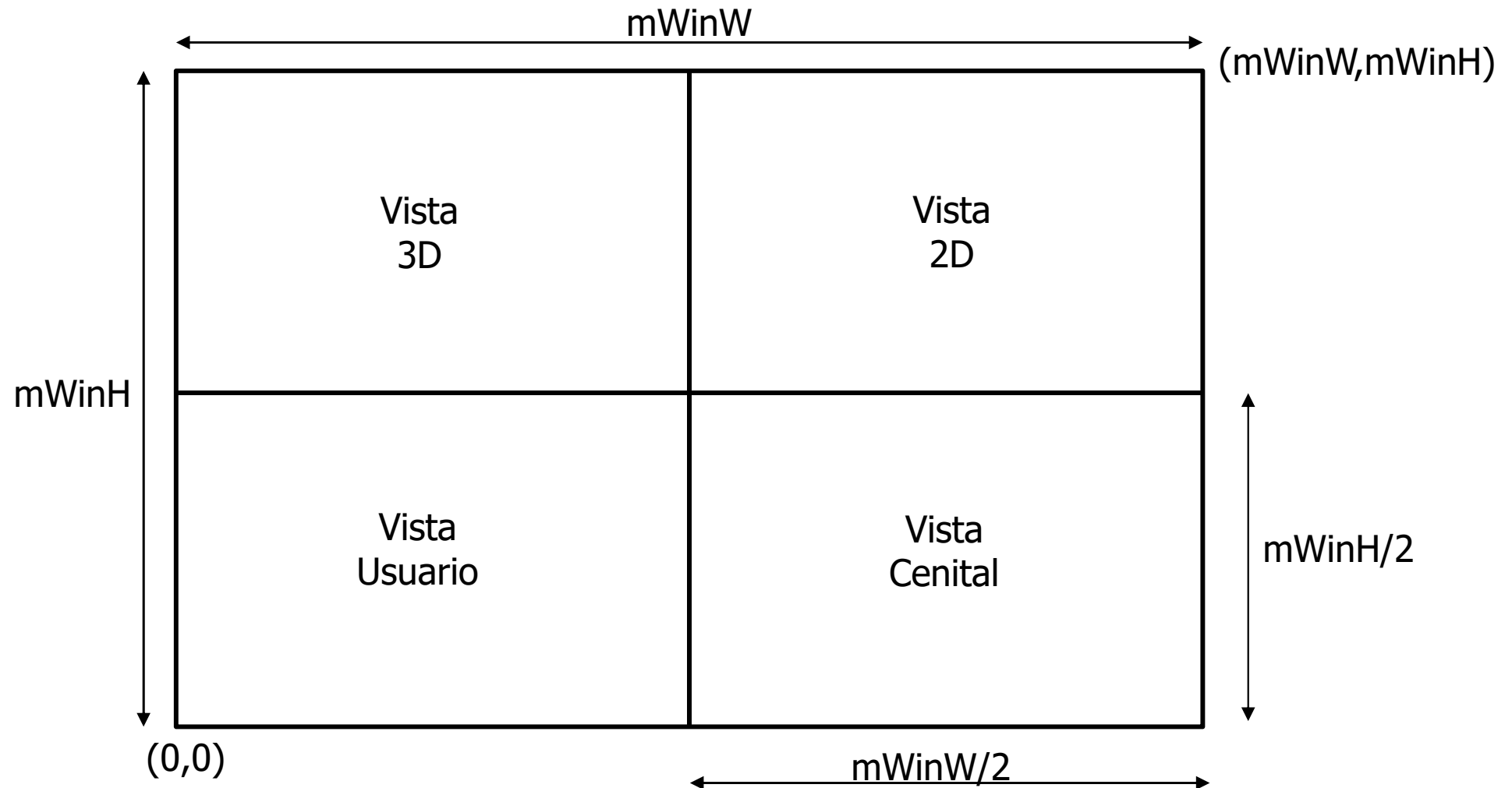
Las 4 vistas comparten la proyección controlada por el usuario con los eventos de la aplicación, es decir:

- Si el usuario establece una proyección perspectiva, las cuatro vistas serán con perspectiva
- Si el usuario cambia la escala, las cuatro vistas tendrán la misma escala.

Como todos los cambios que realiza el usuario se realizan sobre el atributo **mCamera** de la aplicación, usaremos la proyección de este atributo para las demás vistas, pero tenemos que colocar la cámara en distintas posiciones para las distintas vistas.

## Varios puertos de vista: Ejemplo

Ventana de la aplicación ( $mWinW \times mWinH$ )  
con 4 puertos de vista ( $mWinW/2 \times mWinH/2$ )



## Varios puertos de vista: Ejemplo

```
void IG1App::display4V() { // se llama en display()
    // para renderizar las vistas utilizamos una cámara auxiliar:
    Camera auxCam = * mCamera; // copiando mCamera
    // el puerto de vista queda compartido (se copia el puntero)
    Viewport auxVP = * mViewPort; // lo copiamos en una var. aux. para
    // el tamaño de los 4 puertos de vista es el mismo, lo configuramos
    mViewPort->setSize(mWinW / 2, mWinH / 2 );
    // igual que en resize, para que no cambie la escala,
    // tenemos que cambiar el tamaño de la venta de vista de la cámara
    auxCam.setSize(mViewPort->width(), mViewPort->height());
    // vista Usuario ->
    // vista 2D ->
    // vista 3D ->
    // vista Cenital ->
    *mViewPort = auxVP; // restaurar el puerto de vista
```

→ NOTA

## Varios puertos de vista: Ejemplo

### vista Usuario

```
// el tamaño de los 4 puertos de vista es el mismo (ya configurado),  
// pero tenemos que configurar la posición  
mViewport->setPos(0, 0);  
// el tamaño de la ventana de vista es el mismo para las 4 vistas (ya  
    configurado)  
// y la posición y orientación de la cámara es la del usuario (ya  
    configurado -> copiado de mCamera)  
// renderizamos con la cámara y el puerto de vista configurados  
mScene->render(auxCam);
```

→ **NOTA:** al usar una cámara auxiliar auxCam que comparte el puerto de vista con la cámara principal mCamera, el puerto de vista de la cámara principal quedará modificado. Por eso debemos restaurar el puerto de vista mViewport, con los valores guardados en auxVP.  
Otra opción sería añadir a la clase Camera métodos para copiar su configuración y no compartir el puerto de vista.

### vista 2D

```
// el tamaño de los 4 puertos de vista es el mismo (ya configurado),  
// pero tenemos que configurar la posición  
mViewPort->setPos(mWinW / 2, mWinH /2 );  
// el tamaño de la ventana de vista es el mismo para las 4 vistas (ya  
// configurado)  
// pero tenemos que cambiar la posición y orientación de la cámara  
auxCam.set2D();  
// renderizamos con la cámara y el puerto de vista configurados  
mScene->render(auxCam);
```

### vista 3D

```
// ...
```

### vista Cenital

```
// el tamaño de los 4 puertos de vista es el mismo (ya configurado),  
// pero tenemos que configurar la posición  
mViewport->setPos(mWinW / 2, 0);  
// el tamaño de la ventana de vista es el mismo para las 4 vistas (ya  
// configurado)  
// pero tenemos que cambiar la posición y orientación de la cámara  
auxCam.setCenital();  
// renderizamos con la cámara y el puerto de vista configurados  
mScene->render(auxCam);
```