

Introducción a la cámara

Informática Gráfica I

Material de: **Ana Gil Luezas**
Adaptado por: **Elena Gómez y Rubén Rubio**
`{mariaelena.gomez,rubenrub}@ucm.es`



Contenido

1 Matrices

- Posición y orientación
- Marco de la cámara
- Matriz de vista
- Clase Cámara

2 Transformaciones

- Transformaciones

3 Desplazamientos de la cámara

- Rotaciones
- Transformaciones

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

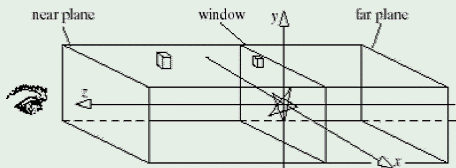
```
glm::dmat4 mViewMat = glm::lookAt(mEye, mLook, mUp);  
// Atributo de la clase Camera
```

Posición y orientación de la cámara

Ejemplos

Los argumentos **eye**, **look** y **up** se usan en **coordenadas globales**.

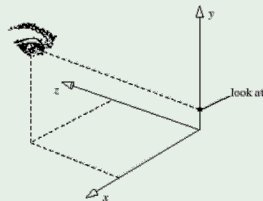
```
mViewMat = lookAt(mEye, mLook, mUp);
```



```
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);
```



Marco de la cámara

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

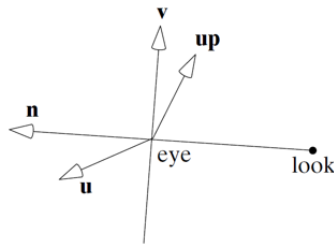
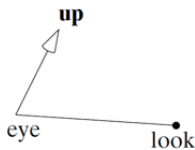
```

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 (MC) =

$$\begin{pmatrix} u_x & v_x & n_x & e_x \\ u_y & v_y & n_y & e_y \\ u_z & v_z & n_z & e_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



Producto vectorial

- $\text{cross}(a, b) = -\text{cross}(b, a) = \begin{pmatrix} i & j & k \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{pmatrix} = \begin{pmatrix} a_y b_z - a_z b_y \\ a_z b_x - a_x b_z \\ a_x b_y - a_y b_x \end{pmatrix}$

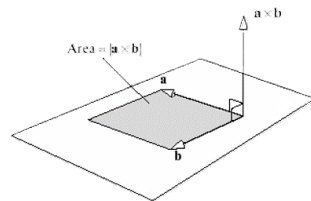
- $|\text{cross}(a, b)| = |a||b|\sin(\theta)$

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

- Producto escalar:

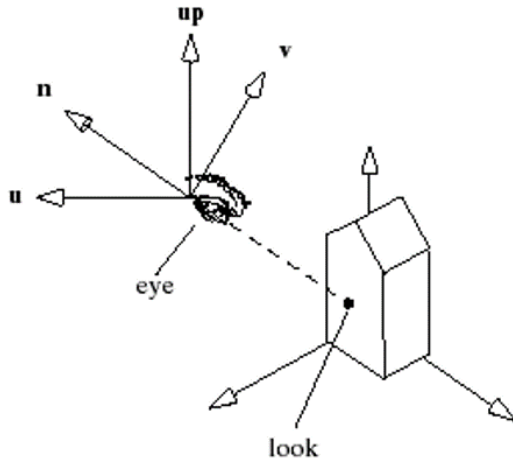
$$a \cdot b = |a||b|\cos(\theta)$$

Si $\theta = 90 \rightarrow \cos(\theta) = 0 \rightarrow a \perp b$



Regla de la mano derecha
(o del sacacorchos)

Marco de la cámara



- La cámara mira hacia $-n$.
- El sistema (u, v, n) es **ortonormal**: vectores ortogonales de magnitud uno \rightarrow la inversa de la matriz 3×3 (u, v, n) es la traspuesta.

$$M_c = \begin{pmatrix} u_x & v_x & n_x & e_x \\ u_y & v_y & n_y & e_y \\ u_z & v_z & n_z & e_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Matriz de vista

- 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).

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

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

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

```
// Atributo de la clase Camera
// Ejes de la cámara
mRight = row(mViewMat, 0);
mUpward = row(mViewMat, 1);
mFront = - row(mViewMat, 2);
```


La clase Cámara

- En la clase `Camera` añadimos:

- los atributos:

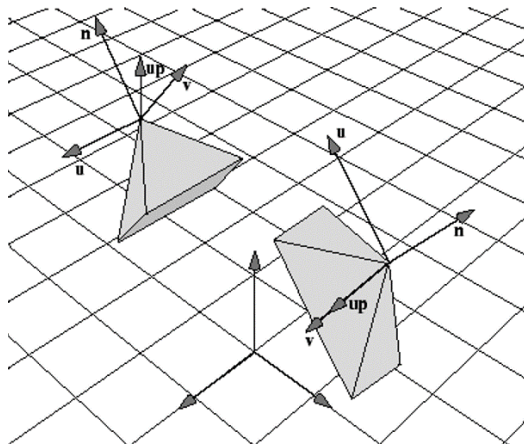
```
dvec3 mRight, mUpward, mFront;  
// para los ejes right=u, upward=v, front=-n
```

- 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();  
}
```

Transformaciones de la Cámara

¿Relativas al sistema global o a la propia cámara?



Transformaciones de la Cámara

¿Relativas al sistema global o a la propia cámara?

- **Ejemplo:** trayectoria circular de la cámara 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 de la cámara en el **eje X** global

```
eye.x += incX; // -> Cambia la dirección de vista (-n = look - eye)  
viewMat = lookAt(eye, look, up);
```

- **Ejemplo:** movimiento horizontal en el **eje u** de la cámara

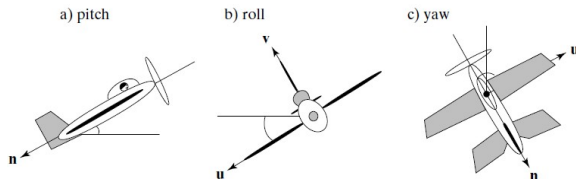
```
eye += u * cs; // -> Cambia la dirección de vista (-n = look - eye)  
viewMat = lookAt(eye, look, up);
```

Desplazamientos de la Cámara

- 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 ...
}
```

Rotaciones de la Cámara



- **Pitch (cabeceo):** mirar hacia arriba y abajo.
Rotación en el eje u (eje X de la cámara)
- **Yaw (guiñada):** mirar a izquierda y derecha.
Rotación en el eje v (eje Y de la cámara)
- **Roll (alabeo):**
Rotación en el eje n (eje Z de la cámara)

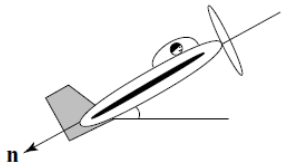
Algunas de estas rotaciones producen cambios en la dirección de vista (**front**)

Rotaciones de la Cámara

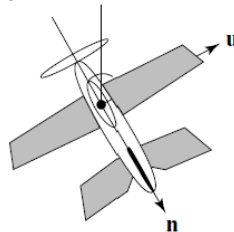
- Si se cambia la dirección de vista (**front**) desplazando **look** en los ejes de la cámara **u** y **v**, se pueden aproximar las rotaciones yaw y pitch:

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

a) pitch



c) yaw



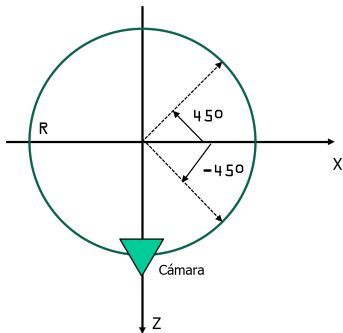
Transformaciones sobre el sistema global

- 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 mRadio, mAng;`
 - Y definimos un método para desplazar **eye** por 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();  
}
```

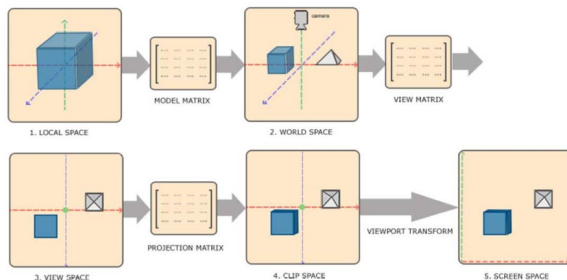
Ejercicio 22

- Si la cámara está en el eje X positivo, el ángulo es 0.
- Si la cámara está en el eje X negativo, el ángulo es 180.
- Si la cámara está en el eje Z positivo, ...
- La cámara 2D está en el eje ... La cámara 3D está en ...



Vista cenital de la órbita de la cámara alrededor del eje Y y radio R

Transformaciones

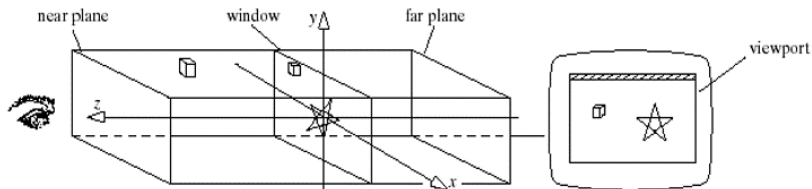


```
void Entity::
render(dmat4 const& modelViewMat){
    dmat4 aMat = modelViewMat
                * mModelMat;
    upload(aMat);
    mesh -> render();
}
```

```
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** (*viewport*) 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);
```

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

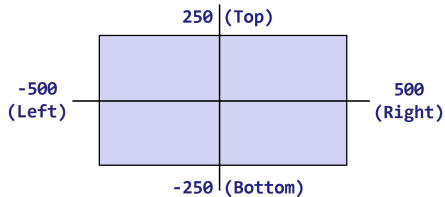
```
mProjMat = frustum(xLeft, xRight, yBottom, yTop, mNear, mFar);
```

`xLeft`, `xRight`, `yBottom` → Ventana de vista

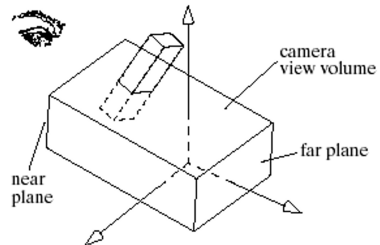
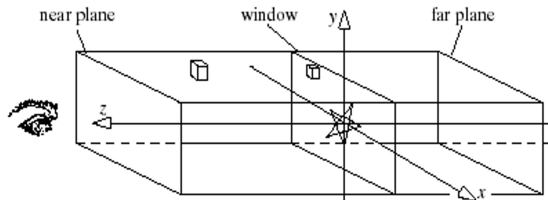
`mNear`, `mFar` → Distancias al ojo

Proyección ortogonal

```
glm::ortho(Left, Right, Bottom, Top, Near, Far);  
glm::ortho(-500, 500, -250, 250, 500, 10000);  
// En coordenadas de la cámara
```

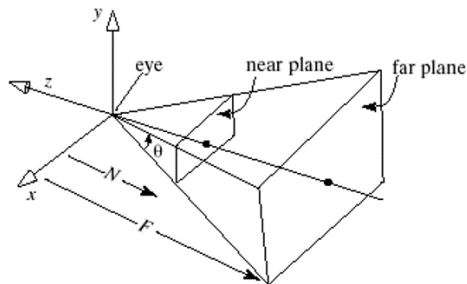


Los rectángulos cercano y lejano son iguales y perpendiculares al eje n



Proyección perspectiva

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



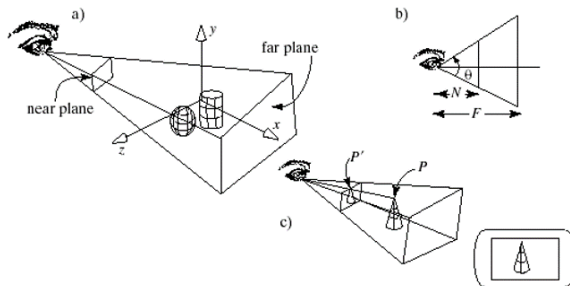
El Rectángulo cercano en coordenadas de la cámara

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.

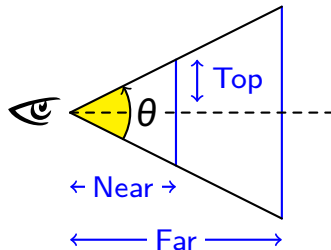


Nota

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

Proyección perspectiva

- La posición de la cámara (**eye**), **Near** y **Top** establecen el ángulo del campo de visión en el eje Y (**fovy**).
 - $\tan(\text{fovy}/2) = \text{Top} / \text{Near}$
 - Para **fovy** = 60: $\tan(30) = 0,5773 \rightarrow \text{Near} = 2 * \text{Top}$
 - Para **fovy** = 90: $\tan(45) = 1 \rightarrow \text{Near} = \text{Top}$



Proyección perspectiva

- También podemos definir volúmenes en la proyección perspectiva con la función:

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

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

- Equivale a

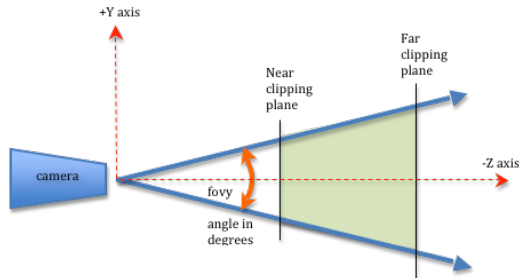
```
glm::frustum(Left, Right, Bottom, Top, Near, Far)
```

```
Top = Near.tan(Fovy / 2.0)
```

```
Bot = -Top
```

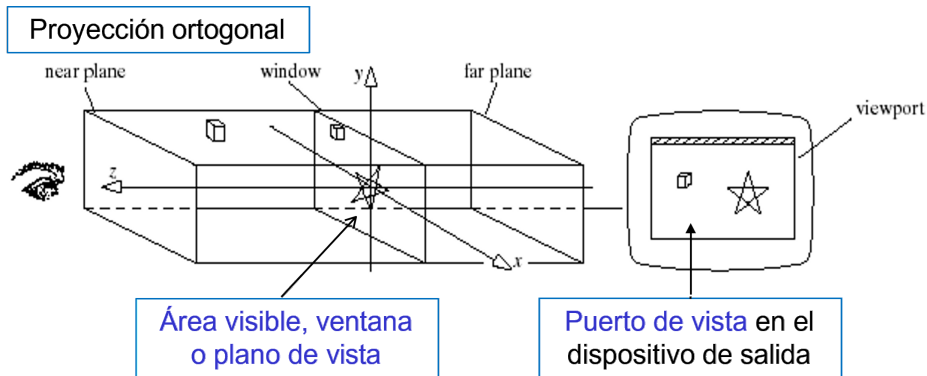
```
Right = Top.AspectRatio
```

```
Left = -Right
```



Proyección y puerto de vista

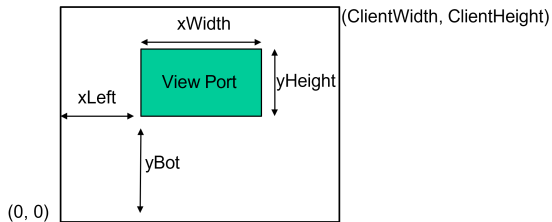
- La proyección obtenida en la ventana de vista se transfiere al puerto de vista establecido en la ventana de visualización.



Puerto de vista

- 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() {  
    glViewport(xLeft, yBot, xWidth, yHeight);  
    // Los parámetros son de tipo entero (píxeles)  
}
```

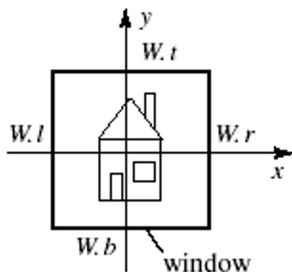


- Puerto de vista ocupando toda el área cliente de la ventana:

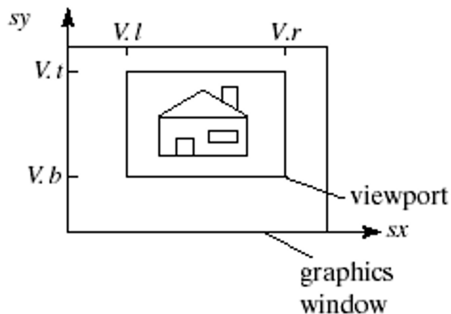
```
glViewport(0, 0, ClientWidth, ClientHeight);
```

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



Eventos de ventana y zoom

```
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);
        break;
```

```
    case '-':
        mCamera->setScale(-0.01);
        break;...
```

```
    glutPostRedisplay(); }
```

Varios puertos de vista

- 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
}
```

Varios puertos de vista

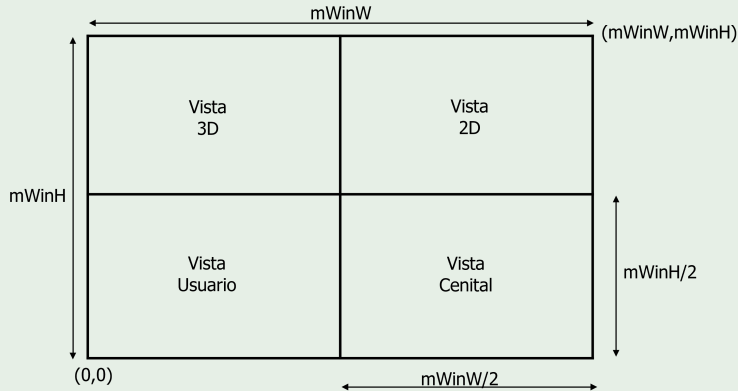
Ejemplo: visualizar 4 vistas de la misma escena

- 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: visualizar 4 vistas de la misma escena

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



Varios puertos de vista

Ejemplo: visualizar 4 vistas de la misma escena

```
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 ventana 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: visualizar 4 vistas de la misma escena

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

Varios puertos de vista

Ejemplo: visualizar 4 vistas de la misma escena - 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);
```

Varios puertos de vista

Ejemplo: visualizar 4 vistas de la misma escena - 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);
```

Ejemplo: visualizar 4 vistas de la misma escena - vista 3D

```
// ...
```

Varios puertos de vista

Ejemplo: visualizar 4 vistas de la misma escena - 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);
```