La cámara

A. Gavilanes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática
Universidad Complutense de Madrid

La cámara

- La cámara se define con el comando:

  `glm::lookAt(eye, look, up);`

  donde **`dvec3 eye`** es el *punto* donde se encuentra (el ojo de) la cámara, **`dvec3 look`** es el *punto* al que mira la cámara, y **`dvec3 up`** es el *vector* que indica cómo está orientada la cámara. En función de estos tres elementos se define el sistema de coordenadas de la cámara **`{u, v, n, eye}`**, como veremos después

- Las coordenadas de **`eye`**, **`look`** y **`up`** se expresan usando el sistema de referencia global.

- El comando **`lookAt(eye, lookX, up)`** define la llamada matriz de vista

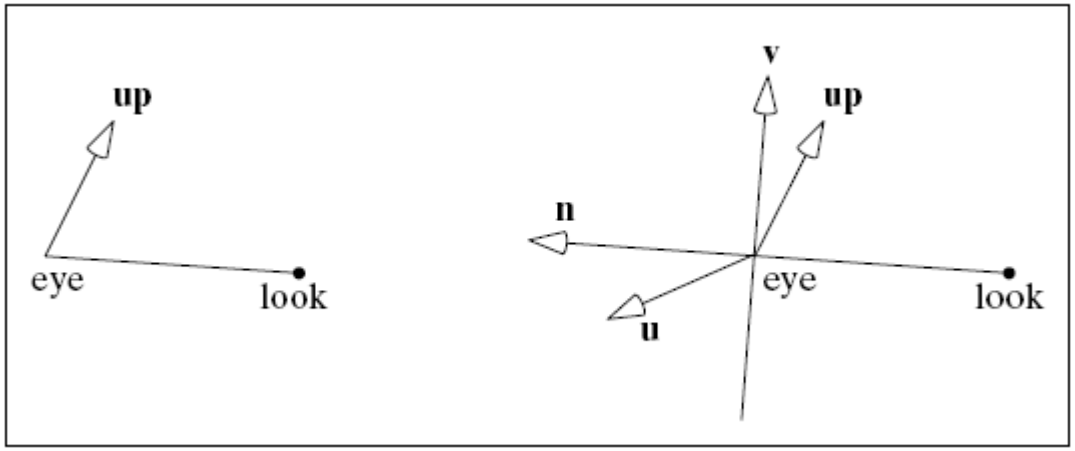
  `glm::dmat4 viewMat = glm::lookAt(eyeX, look, up);`

  que es la inversa de la matriz del sistema de referencia de la cámara. Por tanto, la matriz de vista pasa de coordenadas globales a coordenadas de cámara.

Sistema de coordenadas de la cámara

- A partir de **eye, look** y **up** se determina el sistema de coordenadas de la cámara:

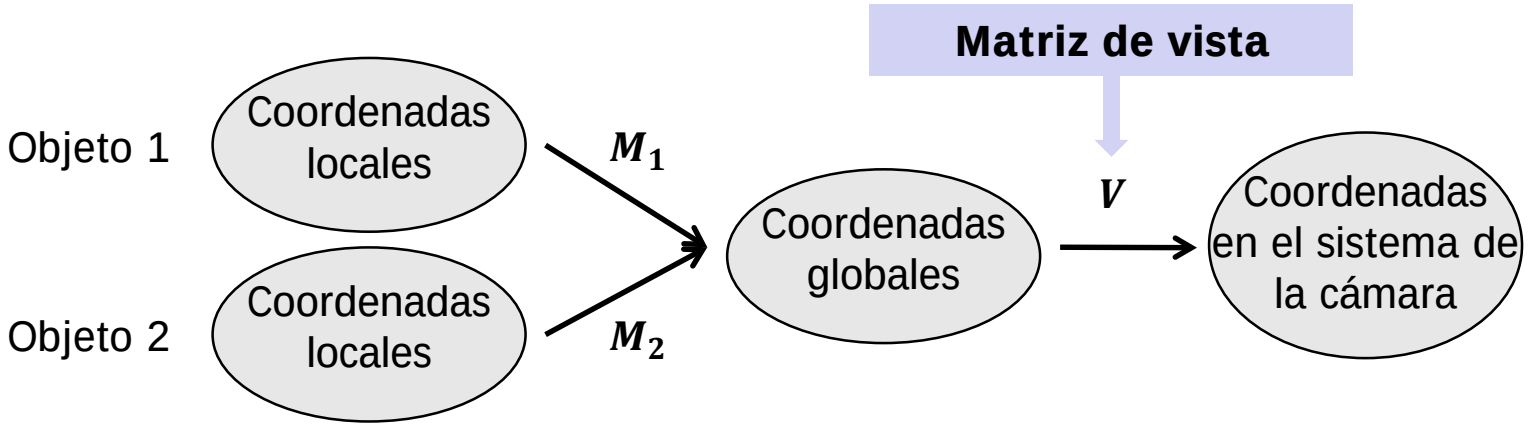
```
n = (eye-look).normalizar()
u = (up×n).normalizar()
v = n×u
Origen = eye
```



- Recuerda que la matriz que pasa del sistema de coordenadas de la cámara al sistema de coordenadas globales es entonces:

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

- Pero a la hora de modelar interesa pasar de coordenadas globales a coordenadas de la cámara:



- La matriz que pasa del sistema global al sistema de la cámara es entonces:

$$V = \begin{pmatrix} u_x & v_x & n_x & e_x \\ u_y & v_y & n_y & e_y \\ u_z & v_z & n_z & e_z \\ 0 & 0 & 0 & 1 \end{pmatrix}^{-1} = \begin{pmatrix} u_x & u_y & u_z & -\overrightarrow{Oe} \cdot u \\ v_x & v_y & v_z & -\overrightarrow{Oe} \cdot v \\ n_x & n_y & n_z & -\overrightarrow{Oe} \cdot n \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Compruébese

Observaciones

- Lo habitual, cuando se invoca **`lookAt(…)`** con datos concretos, es tomar **`up=dvec3(0, 1, 0)`** para indicar que la cámara está derecha.
  - ¿Vista inversa de la escena? ¿Vista cenital?

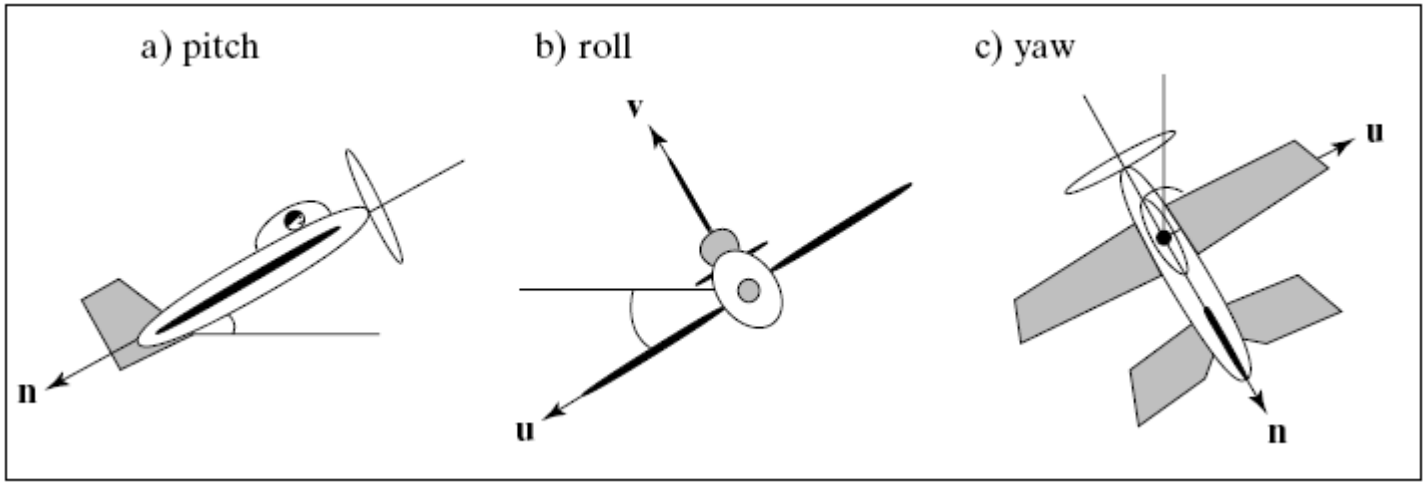
  También es habitual manejar una visión frontal de la escena haciendo **`eye=dvec3(100, 100, 100)`** y **`look=dvec3(0, 0, 0)`**.
- Conocida la matriz de vista **`V`** mediante el comando **`lookAt(…)`**, por ejemplo, los vectores que definen el sistema de coordenadas de la cámara se pueden obtener a partir de ella así:
  - **`u = row(V, 0), v = row(V, 1), n=row(V, 2)`**

  donde **`glm::row(M, k)`** es una función de la librería **`gtc`** que devuelve las tres primeras componentes de la fila **`k`** de la matriz **`M`**. Para usar este comando es necesario hacer la inclusión:

```
#include <gtc/matrix_access.hpp>
```

Rotaciones clásicas de la cámara

- Rotaciones con respecto a los ejes del sistema de coordenadas de la cámara.



- El movimiento **`pitch`** es una rotación con respecto al eje **`u`**
- El movimiento **`yaw`** es una rotación con respecto al eje **`v`**
- El movimiento **`roll`** es una rotación con respecto al eje **`n`**

Movimientos de la cámara

- Llevaremos a cabo los movimientos de la cámara moviendo la cámara, no moviendo la escena.
- Los movimientos se pueden clasificar en tres grandes grupos:
  I. Los realizados mediante el ratón, que veremos más adelante
  II. Aquellos en que se calculan nuevos valores de **`eye`**, **`look`** y **`up`** y luego se ejecuta **`lookAt(…)`**. Ejemplos:
    - Rotar la cámara alrededor del eje **`X`**: **`look`** y **`up`** no cambian, **`eye`** recorre los puntos de una circunferencia alrededor del eje **`X`**.
    - Mostrar la vista cenital de la escena: **`look`** no cambia, **`eye`** se sitúa sobre la escena y **`up`** se define para que la cámara pueda mostrar la escena desde arriba.

III. Aquellos en que se transforma el sistema de coordenadas de la cámara (mediante una matriz M). El proceso que se sigue es el siguiente:

1. Calcular las coordenadas del nuevo sistema de coordenadas de la cámara en función del sistema de cámara original:

$$\boldsymbol{u}' = M\boldsymbol{u} \quad \boldsymbol{v}' = M\boldsymbol{v} \quad \boldsymbol{n}' = M\boldsymbol{n} \quad \boldsymbol{e}' = M\boldsymbol{e}$$

2. Expresar el nuevo sistema de cámara en coordenadas globales:

$$\boldsymbol{u}' = V^{-1}M\boldsymbol{u} \quad \boldsymbol{v}' = V^{-1}M\boldsymbol{v}$$
$$\boldsymbol{n}' = V^{-1}M\boldsymbol{n} \quad \boldsymbol{e}' = V^{-1}M\boldsymbol{e}$$

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

3. Calcular la matriz de vista correspondiente:

$$\boldsymbol{V}' = \begin{pmatrix} \boldsymbol{u}'_x & \boldsymbol{v}'_x & \boldsymbol{n}'_x & \boldsymbol{e}'_x \\ \boldsymbol{u}'_y & \boldsymbol{v}'_y & \boldsymbol{n}'_y & \boldsymbol{e}'_y \\ \boldsymbol{u}'_z & \boldsymbol{v}'_z & \boldsymbol{n}'_z & \boldsymbol{e}'_z \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{1} \end{pmatrix}^{-1} = \begin{pmatrix} \boldsymbol{u}'_x & \boldsymbol{u}'_y & \boldsymbol{u}'_z & -\overrightarrow{\boldsymbol{Oe}'} \cdot \boldsymbol{u}' \\ \boldsymbol{v}'_x & \boldsymbol{v}'_y & \boldsymbol{v}'_z & -\overrightarrow{\boldsymbol{Oe}'} \cdot \boldsymbol{v}' \\ \boldsymbol{n}'_x & \boldsymbol{n}'_y & \boldsymbol{n}'_z & -\overrightarrow{\boldsymbol{Oe}'} \cdot \boldsymbol{n}' \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{1} \end{pmatrix}$$

4. Cargar la matriz resultante como la nueva matriz de vista:

Ejemplo. Rotación de la cámara sobre el eje N (roll)

1. El nuevo eje $\boldsymbol{u}'$ expresado en el sistema de la cámara es:

$$\boldsymbol{u'}_{\boldsymbol{camera}} = \begin{pmatrix} cos\,\theta & -sin\,\theta & 0 & 0 \\ sin\,\theta & cos\,\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} cos\,\theta \\ sin\,\theta \\ 0 \\ 0 \end{pmatrix}$$

2. El nuevo eje $\boldsymbol{u}'$ expresado en coordenadas globales es entonces:

$$\boldsymbol{u'}_{\boldsymbol{global}} = \begin{pmatrix} u_x & v_x & n_x & e_x \\ u_y & v_y & n_y & e_y \\ u_z & v_z & n_z & e_z \\ 0 & 0 & 0 & 1 \end{pmatrix} u'_{camera} = \begin{pmatrix} u_x cos\,\theta + v_x\, sin\,\theta \\ u_y cos\,\theta + v_y\, sin\,\theta \\ u_z cos\,\theta + v_z\, sin\,\theta \\ 0 \end{pmatrix}$$

Del mismo modo se calculan el resto de elementos del nuevo sistema:

$$\boldsymbol{v'}_{\boldsymbol{global}} = \begin{pmatrix} -u_x \sin\theta + v_x \cos\theta \\ -u_x \sin\theta + v_x \cos\theta \\ -u_x \sin\theta + v_x \cos\theta \\ 0 \end{pmatrix} \quad \boldsymbol{n'}_{\boldsymbol{global}} = \begin{pmatrix} \boldsymbol{n_x} \\ \boldsymbol{n_y} \\ \boldsymbol{n_z} \\ \boldsymbol{0} \end{pmatrix} \quad \boldsymbol{e'}_{\boldsymbol{global}} = \begin{pmatrix} \boldsymbol{e_x} \\ \boldsymbol{e_y} \\ \boldsymbol{e_z} \\ \boldsymbol{0} \end{pmatrix}$$

3. A partir del sistema de la cámara $\{\boldsymbol{u'}_{\boldsymbol{global}}, \boldsymbol{v'}_{\boldsymbol{global}}, \boldsymbol{n'}_{\boldsymbol{global}}, \boldsymbol{e'}_{\boldsymbol{global}}\}$, se calcula la matriz de vista correspondiente.

Navegación de la cámara

- La cámara se transforma usando su propio sistema de referencia (=> movimiento de la cámara en primera persona).
- Transformación del sistema de referencia típicas:
  - ✓ Rotación sobre algunos de sus ejes: pitch (U), yaw (V) y roll (N).
  - ✓ Traslación a lo largo del eje N ($t = (0,0,k,0)$ en coordenadas de cámara).
- A partir del sistema de cámara actual $\{u, v, n, e\}$, se calcula el nuevo sistema de cámara $\{u', v', n', e'\}$. La siguiente tabla sintetiza el resultado de las cuentas explicadas anteriormente. Siempre se usan coordenadas globales.

Transformación del sistema de la cámara	u'	v'	n'	e'
Rotación sobre U	u	$\cos\theta\, v + \sin\theta\, n$	$-\sin\theta\, v + \cos\theta\, n$	e
Rotación sobre V	$\cos\theta\, u - \sin\theta\, n$	v	$\sin\theta\, u + \cos\theta\, n$	e
Rotación sobre N	$\cos\theta\, u + \sin\theta\, v$	$-\sin\theta\, u + \cos\theta\, v$	n	e
Traslación de vector $(0,0,k,0)$	u	v	n	$e + kn$

La clase Camera. Atributos

- Debes modificar tu clase **`Camera`** tal como se diga en las transparencias que vienen a continuación
- Añadir o actualizar los siguientes atributos protegidos:
  - Los que permiten definir la orientación y el sistema de coordenadas de la cámara, inicializados tal como se muestra a continuación

```
glm::dvec3 eye = { 1000.0, 1000.0, 1000.0 };
glm::dvec3 look = { 0.0, 0.0, 0.0 };
glm::dvec3 up = { 0.0, 1.0, 0.0 };
glm::dvec3 front = { 0.0, 0.0, -1.0 }; // Observa que es -n
glm::dvec3 u = { 1.0, 0.0, 0.0 };
glm::dvec3 v = { 0.0, 1.0, 0.0 };
```

  - Los que permiten definir las dimensiones del volumen de vista, algunos de ellos inicializados tal como se muestra. Los otros se inicializan en la constructora

```
GLdouble xRight, xLeft, yTop, yBot;
GLdouble nearVal = 500;
GLdouble farVal = 10000;
```

La clase Camera. Atributos

- Para programar ciertos movimientos de la cámara se usan dos atributos nuevos:
  - **`radio`**: es el radio de una esfera imaginaria sobre cuya superficie se mueve la cámara
  - **`ang`**: es la longitud a la que se encuentra la cámara en esa esfera imaginaria
- Como se hace con los atributos **`u`**, **`v`**, **`n`**, según veremos después, para procurar que el enlace de unos movimientos de la cámara con otros sea coherente (no se produzcan saltos) es preciso mantener actualizados los atributos **`radio`** y **`ang`** después de realizar ciertos movimientos de la cámara.

La clase Camera. Atributos

- Estos y otros atributos se inicializan tal como se muestra a continuación:

```
GLdouble ang = 0.0;
GLdouble radio = 1000.0;
GLdouble factScale = 1;
bool orto = true;
```

- Los que ya existían, relacionados con la visibilidad de la escena

```
glm::dmat4 viewMat; // view matrix (= inverse of camera matrix)
glm::dmat4 projMat; // projection matrix
Viewport* vp;       // view port
```

La clase Camera. Funcionalidad protegida

- Añadir a/o redefinir en la clase **`Camera`** los siguientes métodos:
  - **`void setAxes()`**: método protegido que da valor a **`u`**, **`v`**, **`front`** a partir de las filas de la matriz de vista. Recuerda que **`front`** es **`-n.`** Se usa **`row()`** para definirlo.
  - **`void setVM()`**: método protegido que invoca **`lookAt()`** con los valores de **`eye`**, **`look`**, **`up`** para dar valor a la matriz de vista, y actualiza después los ejes con el método anterior.
  - **`void uploadPM():`** método protegido que fija la matriz de proyección (ortogonal o perspectiva) según sea el valor del booleano **`orto`**. Se explica más adelante cómo definirlo.

La clase Camera. Funcionalidad pública

- Añadir a/o redefinir en la clase **`Camera`** los siguientes métodos:
  - **`void set3D()`**: método público que sitúa el ojo en el punto **`(frente, frente, frente)`**, mira al origen y pone la cámara derecha, donde **`frente=radio*cos(ang)`** y **`ang=-45º`**
  - **`void setCenital()`**: método público que muestra una vista cenital de la escena, situando el ojo a una altura igual a **`radio`**, mirando al origen y orientando la cámara en la dirección **`(Z, -Z)`**. El método actualiza **`ang`** a **`-90º`**, por ejemplo, para que, al recuperarse de este movimiento, se vuelva a ver la imagen derecha

La clase Camera. Movimientos

- Añadir a/o redefinir en, la clase **`Camera`** los siguientes métodos públicos:
  - **`void moveLR(GLdouble cs)`**: movimiento de la cámara a izquierda o derecha, sobre el eje **`U`**, una distancia **`cs`**. Cambia pues **`eye`** y, de acuerdo con ello, **`look`**:

```
eye += u * cs;

look += u * cs;

setVM();
```

  - **`void moveFB(GLdouble cs)`**, **`void moveUD(GLdouble cs)`**: movimientos análogos sobre los ejes **`N`** y **`V`**, respectivamente
  - **`void lookLR(GLdouble cs)`**, **`void lookUD(GLdouble cs)`**: movimientos análogos sobre los ejes **`U`** y **`V`**, respectivamente, pero sin cambiar **`eye`**, solo **`look`**

- Añadir a/o redefinir en, la clase **Camera** los siguientes métodos públicos:
  - **void orbit(GLdouble ax)**: orbita la cámara alrededor de **look**, haciendo que **eye** describa el paralelo que se encuentra a altura **eye.y**, en sentido antihorario:

```
ang += ax;
eye.x = look.x + cos(radians(ang)) * radio;
eye.z = look.z - sin(radians(ang)) * radio;
setVM();
```

  - **void orbit(GLdouble ax, Gldouble ay)**: método similar al anterior, pero se deja que la cámara cambie su altura una cantidad **ay**. Obsérvese que **ax** es un double que se refiere a radianes, mientras que **ay** es una longitud

```
ang += ax;
eye.x = look.x + cos(radians(ang)) * radio;
eye.z = look.z - sin(radians(ang)) * radio;
eye.y +=ay;
setVM();
```

Eventos de ratón

- Para programar los eventos de ratón sigue estos pasos:
  - Añadir los siguientes callbacks a **`main`**.
    - **`void mouse(int button, int state, int x, int y);`**
      - Se genera cuando se presiona o se suelta un botón del ratón (**`button`**), y recoge en coordenadas de la ventana **`(x, y)`** el momento en que el estado (**`state`**) del botón cambió y pasó a estar pulsado o a estar soltado
    - **`void motion(int x, int y);`**
      - Se genera cuando un botón del ratón se encuentra pulsado y recoge, en coordenadas de la ventana **`(x, y)`**, el lugar donde se soltó

Eventos de ratón

- Para programar los eventos de ratón sigue estos pasos:
  - Registrar los respectivos callbacks en **`main`**.

```
glutMouseFunc(mouse);

glutMotionFunc(motion);
```

  - Declarar dos variables en **`main`** para guardar las coordenadas del ratón y el botón pulsado.

```
glm::dvec2 mCoord;

int mBot=0;
```

Programación de los eventos de ratón

- Programar **mouse()** de forma que se registren los valores lanzados en las variables globales declaradas al efecto en **main**:

```
void mouse(int button, int state, int x, int y) {
        mBot = button;
        mCoord = glm::dvec2(x, y);
}
```

  - Recuerda que la variable **y** se refiere a coordenadas de ventana y esta tiene su origen en la esquina superior izquierda, mientras que en el puerto de vista el origen está en la esquina inferior izquierda. El paso de una a otra es:

```
y(viewport) = glutGet(GLUT_WINDOW_HEIGHT)-y;
```

Programación de los eventos de ratón

- Para programar **`motion()`** debes tener en cuenta que :
  - Las constantes para referirse a cada botón del ratón son:

    `GLUT_LEFT_BUTTON/GLUT_RIGHT_BUTTON`

  - Por cierto, las constantes para referirse al estado del botón, si soltado o pulsado, son, respectivamente:

    `GLUT_UP/GLUT_DOWN`
- Programar **`motion()`** de forma que la cámara orbite con el botón izquierdo y se desplace a izquierda o derecha, con el botón derecho

Programación de los eventos de ratón

```
void motion(int x, int y) {
        // Guardar los valores de mCoord cuando se pulsó el botón
        glm::dvec2 mp = mCoord;
        mCoord = glm::dvec2(x, y);
        // Calcular el desplazamiento habido
        mp = (mCoord - mp);
        if (mBot == GLUT_LEFT_BUTTON)
          // Recuerda que mp.x son radianes. Redúcelos a tu gusto
          camera.orbit(mp.x*0.05, mp.y);
        else if (mBot == GLUT_RIGHT_BUTTON) {
                camera.moveUD(mp.y);
                camera.moveLR(-mp.x);
        }
        glutPostRedisplay();
}
```

Volumen de vista

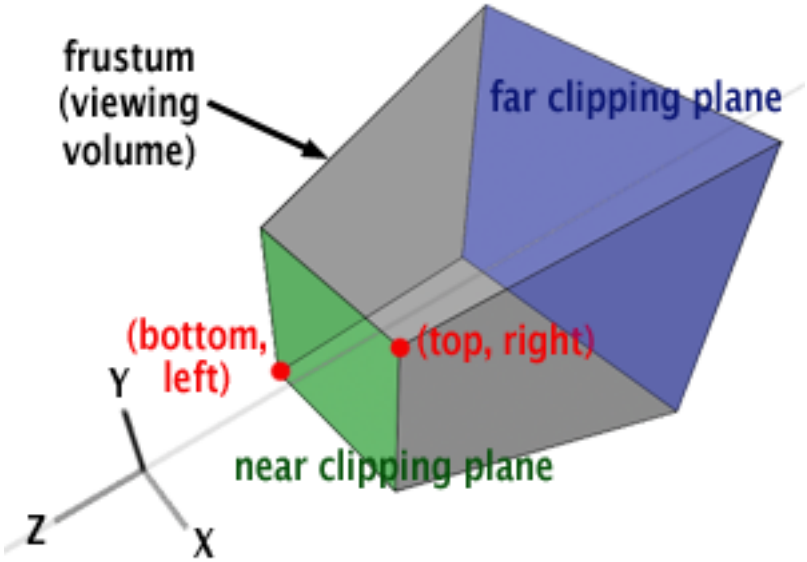
- Es la parte de la escena que es visible por la cámara.
- El contenido del volumen de vista es proyectado sobre un plano de proyección o plano de vista (mediante la matriz de proyección) y la proyección obtenida es mostrada en el puerto de vista (mediante la matriz del puerto de vista).
- Para definir los límites del volumen de vista se añaden, a la clase **`Camera`**, los atributos:

```
GLdouble xRight, xLeft, yTop, yBot;
GLdouble nearVal = 1, farVal = 10000;
```

junto con el atributo (y su accesora):

```
glm::dmat4 projMat;
```

para definir la matriz de proyección, que pasa de coordenadas de cámara a coordenadas del volumen de vista.



La matriz de proyección

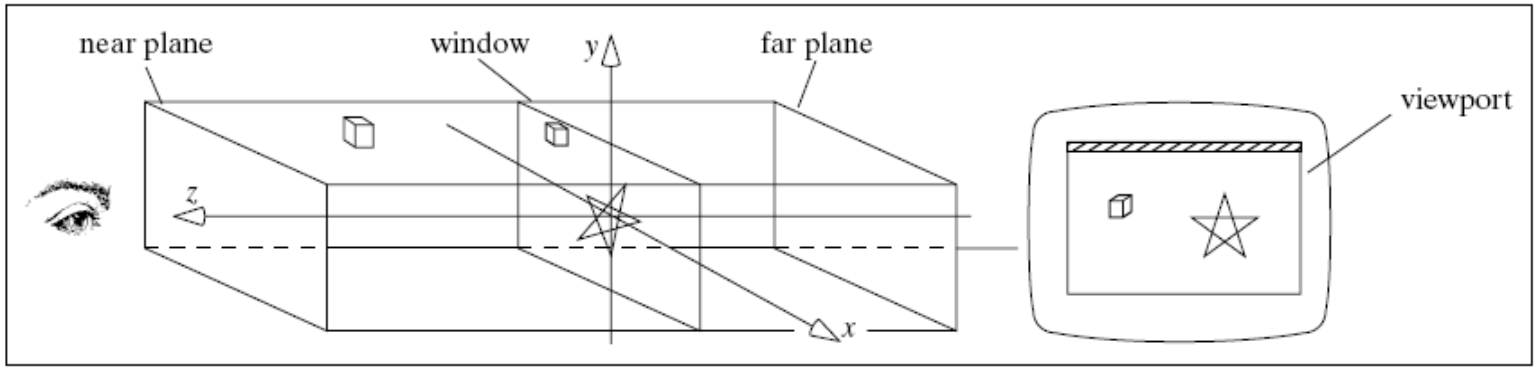
- La matriz de proyección es almacenada en OpenGL con el método:

```
void Camera::uploadPM() {

    glMatrixMode(GL_PROJECTION);
    glLoadMatrixd(value_ptr(projMat));
    glMatrixMode(GL_MODELVIEW);

}
```

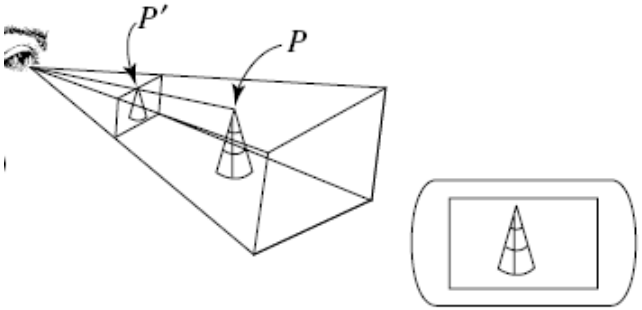
- Observa cómo, después de cargar la matriz en el modo **`GL_PROJECTION`**, se recupera el modo **`GL_MODELVIEW`**, que es el que se suele manejar.
- Ojo, después se explica cómo modificar este método para tener en cuenta el tipo de proyección con la variable booleana **`orto`**.

Volumen de vista

- En OpenGL, hay dos formas predefinidas de proyectar el volumen de vista: la proyección ortogonal



y la proyección perspectiva.

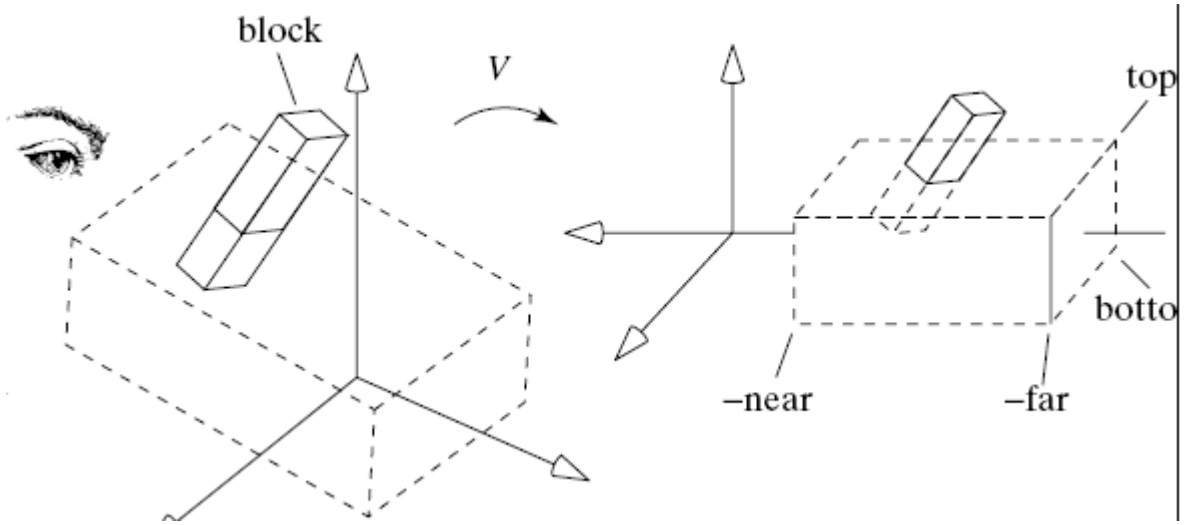


- La proyección ortogonal se obtiene y se guarda con el comando:

```
projMat=ortho(xLeft, xRight, yBot, yTop, nearVal, farVal);
```

donde **xLeft, xRight, yBot, yTop** son coordenadas en los ejes **U** y **V** de la cámara, respectivamente, y **nearVal, farVal** son las distancias de la cámara al plano cercano y lejano, respectivamente, medidas en la parte negativa del eje **N**.

- Los límites del volumen de vista se expresan en el sistema de la cámara.
- En la proyección ortogonal, el volumen de vista es un paralelepípedo.



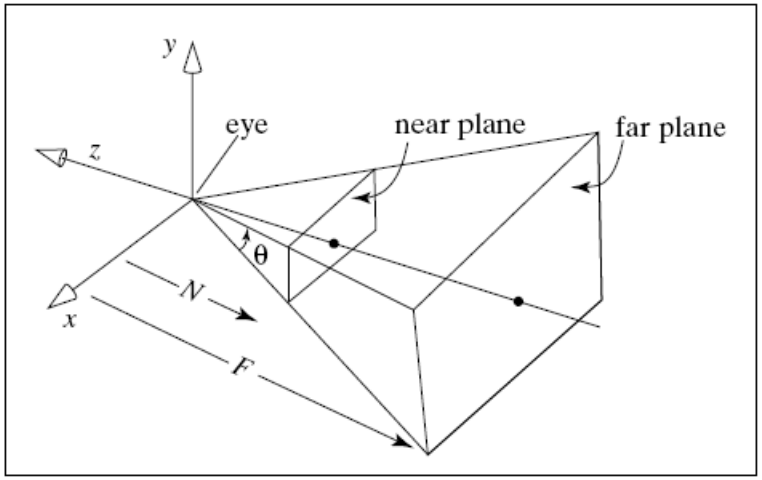
Proyección perspectiva

- La proyección perspectiva se obtiene y se guarda con el comando:

```
projMat = frustum(xLeft, xRight, yBot, yTop, nearVal, farVal);
```

  donde **`xLeft, xRight, yBot, yTop, nearVal, farVal`** se definen como en la proyección ortogonal. Se debe cumplir que **`0<nearVal<farVal`**.

- En la proyección perspectiva, el volumen de vista es una pirámide truncada (*frustum*, en inglés) en cuyo ápice se encuentra el ojo de la cámara.



Proyección perspectiva

- La proyección perspectiva también se puede definir y guardar con el comando:

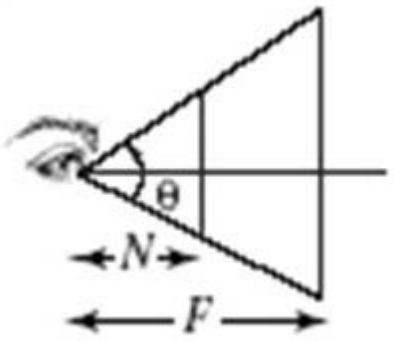
```
projMat = perspective(fovy, aspect, nearVal, farVal);
```

  donde **fovy** (acrónimo del inglés *field of view Y*) es la apertura en el eje **V**, y **aspect** es la proporción de las dimensiones del plano cercano (que es igual a la proporción del plano lejano), esto es **ancho/alto** (del plano de proyección).

- A diferencia del comando anterior, con este comando la pirámide truncada es necesariamente simétrica con respecto a los ejes **U** y **V** de la cámara.

- Se puede pasar de este comando al otro teniendo en cuenta que:

```
yTop = nearVal*tan(fovy/2.0)

yBottom = -yTop

xRight = yTop*aspect

xLeft = -xRight
```



Observaciones

- Cuando se proyectan sobre el plano de vista, los puntos de los objetos que se encuentran en la misma línea de proyección se proyectan en el mismo punto, ganando el más cercano.
- En la proyección ortogonal, las líneas de proyección son paralelas entre sí con lo que los objetos no disminuyen de tamaño cuando se proyectan.
- En la proyección perspectiva, las líneas de proyección convergen en el ápice del *frustum* con lo que los objetos lejanos se proyectan en objetos más pequeños que los objetos cercanos.
- Como se ha dicho, en la proyección perspectiva:

  `tan(fovy/2.0) = yTop/nearVar`

  con lo que si, por ejemplo, **`fovy=60º`**, entonces **`tan(30)≈0.5773`** y se suele tomar **`nearVal=2*yTop`**. Mientras que si **`fovy=90º`**, entonces **`tan(45)=1`** y se suele tomar **`nearVal=yTop`**.

Efectos derivados de la proyección

- Zoom por variación de los parámetros **`yTop`** o **`yBot`** en la proyección ortogonal.
- Zoom por variación del parámetro **`fovy`** en la proyección perspectiva.
- Acercamiento de la cámara sobre el eje **`N`** de la cámara, en la proyección ortogonal.
- Acercamiento de la cámara sobre el eje **`N`** de la cámara, en la proyección perspectiva.

- **`void uploadPM()`:** método público que fija y guarda la matriz de proyección, pero ahora teniendo en cuenta el factor de escala y haciendo que la proyección sea ortogonal o perspectiva, según el valor del booleano **`orto`**.

```
if (orto) {
    nearVal = 1;
    projMat = ortho(xLeft*factScale, xRight*factScale, …, nearVal,
      farVal);
}
else {
    nearVal = 2 * yTop;
    projMat = frustum(xLeft*factScale, …, nearVal, farVal);
}
glMatrixMode(GL_PROJECTION);
glLoadMatrixd(value_ptr(projMat));
glMatrixMode(GL_MODELVIEW);
```

- **`void uploadScale(GLdouble s):`** método público que cambia el factor de escala y fija el volumen de vista en consonancia con ello.

```
factScale -= s;
if (factScale < 0) factScale = 0.01;
uploadPM();
```

Evento de la rueda del ratón

- Para programar este evento sigue estos pasos:
  - Registrar el callback en **`main`**.

```
glutMouseWheelFunc(mouseWheel);
```

  - Añadir el siguiente callback a **`main`** y definirlo tal como se muestra en la siguiente transparencia:

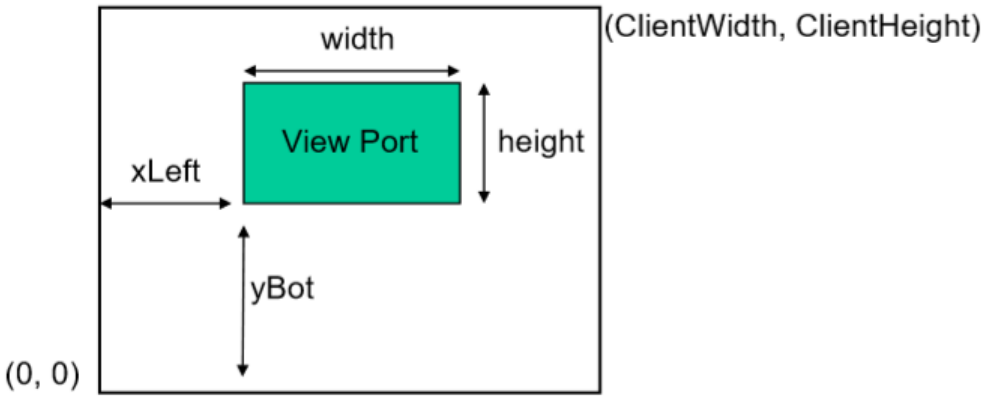
```
void mouseWheel(int n, int d, int x, int y)
```

Evento de la rueda del ratón

```
void mouseWheel(int n, int d, int x, int y) {
        // Se identifica cuántas teclas de las siguientes están pulsadas
        // GLUT_ACTIVE_CTRL/_ALT/_SHIFT
        int m = glutGetModifiers();
        if (m == 0) {  // Es decir, si ninguna tecla está pulsada,
                // se desplaza la cámara en la dirección de vista
                // d=+1/-1, rueda hacia delante/hacia atrás
                if (d == 1) camera.moveFB(5);
                else camera.moveFB(-5);
        }
        else if (m == GLUT_ACTIVE_CTRL) {
                if (d == 1) camera.uploadScale(0.1);
                else camera.uploadScale(-0.1);
        }
        glutPostRedisplay();
}
```

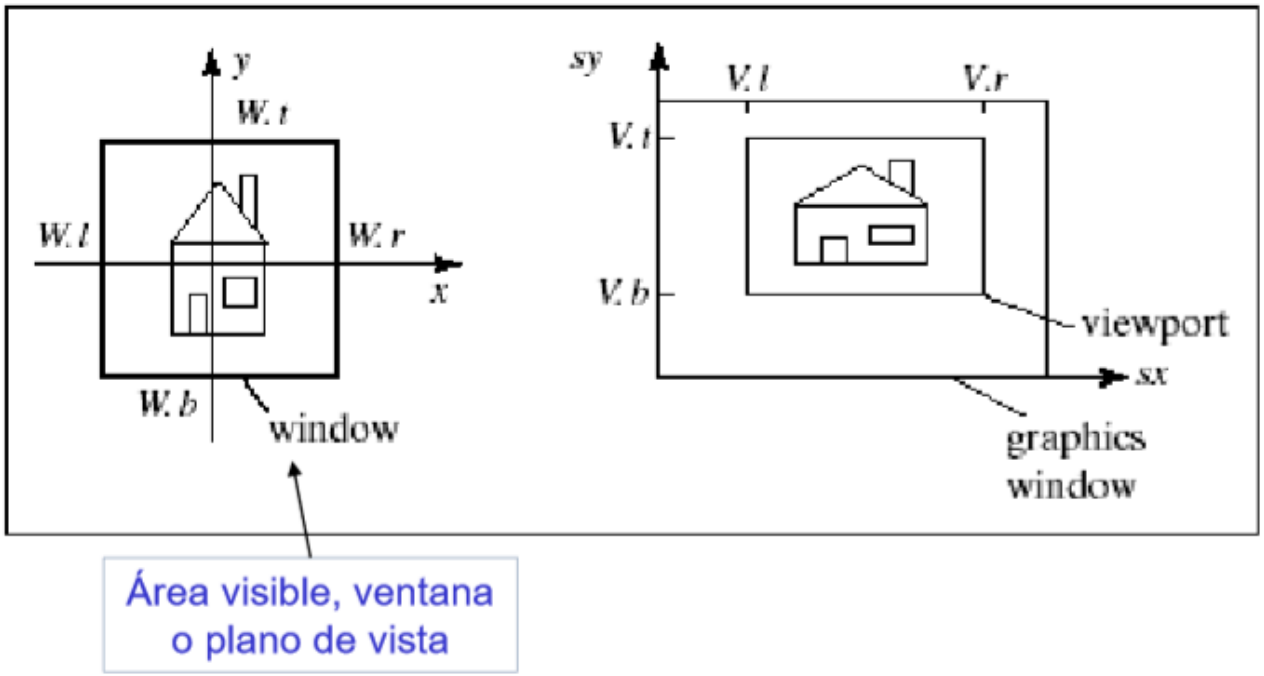
Puerto de vista

- El puerto de vista es un rectángulo alineado con los ejes que se encuentra sobre el área cliente de la ventana en la que OpenGL dibuja.
- Para delimitar el puerto de vista usaremos 4 variables (cuyo valor se mide en píxeles): **`xLeft`**, **`yBot`**, **`width`**, y **`height`**. Las dos primeras sirven para determinar las coordenadas de la esquina inferior izquierda del rectángulo y las dos últimas, para determinar su ancho y alto, respectivamente.
- En el puerto de vista el origen de coordenadas se encuentra en la esquina inferior izquierda de manera que las **`x`** crecen hacia la derecha, y las **`y`**, hacia arriba.
- Generalmente haremos que el puerto de vista ocupe todo el área cliente de la ventana, de manera que **`xLeft=yBot=0`**, y **`width`** y **`height`** corresponderán al tamaño del área cliente.



Puerto de vista

- El paso del plano de vista (con la escena proyectada) al puerto de vista supone una traslación y una escalación. Esta última puede deformar la imagen original.
- Para evitar deformaciones la proporción de ambos rectángulos debe ser la misma.



La clase Viewport

- Tiene los atributos:

```
GLint xLeft=0, yBot=0; // Para la esquina inferior izquierda
GLsizei width, height; // Para el ancho y alto del rectángulo
```

- La funcionalidad principal de esta clase es:
  - `void uploadPos(GLint al, GLint ab):` fija (xLeft, yBot)
  - `void uploadSize(GLsizei aw, GLsizei ah):` fija (width, height)
  -
```
void Viewport::upload() {
  glViewport(xLeft, yBot, width, height);
}
```

- Para fijar el puerto de vista ocupando todo el área cliente de la ventana:

```
glViewport(0, 0, CLIENT_WIDTH, CLIENT_HEIGHT);
```

- La clase **`Viewport`** aparece en:
  - La variable global **`Viewport viewPort(800, 600)`** que fija el puerto de vista sobre el que se construye inicialmente la cámara **`Camera camera(&viewPort);`**
  - El *callback* **`resize()`**

```
void resize(int newWidth, int newHeight)
{
  // Redimensiona el puerto de vista
  viewPort.uploadSize(newWidth, newHeight);
  // Redimensiona el área visible de la escena
  camera.uploadSize(viewPort.getW(), viewPort.getH());
}
```

Embaldosado (Tiling)

- El embaldosado se ocupa de renderizar en varios puertos de vista y mostrar, en cada uno de ellos, diferentes (o la misma, repetida) vistas de una escena, diferentes escenas, etc.
- El ejemplo siguiente muestra la misma escena en varios puertos de vista, todos con las mismas dimensiones, y todos de la misma proporción que el área cliente de la ventana **`CLIENT_WIDTHxCLIENT_HEIGHT`**. Para ello:
  - Se divide el ancho del área cliente en **`nCol`** columnas, y este será el ancho de los puertos de vista
  - Se calcula el alto de los puertos de vista de manera que la proporción con el área cliente sea la misma
  - Se cambia **`display()`** de **`main`** por:

```
if (baldosas)
   // Se muestran, por ejemplo, 4 puertos de vista
  embaldosar(4);
  else scene.render(camera.getViewMat());
   // Ojo, esto no recupera la vista previa al embaldosado
```

Embaldosado (Tiling)

```
void embaldosar(int nCol) {
    GLdouble SVAratio = (camera.xRight-camera.xLeft) / (camera.yTop-camera.yBot);
    GLdouble w = (GLdouble)CLIENT_WIDTH / (GLdouble)nCol;
    GLdouble h = w / SVAratio;
    for (GLint c = 0; c < nCol; c++) {
        GLdouble currentH = 0;
        while ((currentH + h) <= CLIENT_HEIGHT) {
                Viewport* vp = new Viewport((GLint)w, (GLint)h);
                vp->uploadPos((GLint)(c*w), (GLint)currentH);
                vp->upload();
                camera.setVP(vp);
                scene.render(camera.getViewMat());
                currentH += h;
        }
    }
}
```