Gráficos por Computador. Prácticas de Laboratorio

Práctica 3. Modelado Visual

El objetivo de la práctica es el dominio de los conceptos de transformación de un sistema al sistema de la vista definido por una cámara sintética. Para ello se propone la construcción de clases que permitan definir y manipular sistemas visuales basados en el modelo de cámara sintética como el que utilizan las librerías gráficas habituales como OpenGL. La práctica capacita para el desarrollo de transformaciones entre diferentes sistemas de referencia.

Práctica 3. Sistema interactivo multivista (1.50 puntos)

Objetivo:
Desarrollar y validar las clases *Camara*, *CamaraOrtografica* y *CamaraPerspectiva* según el modelo de cámara sintética estudiado.

Descripción:
Se va a implementar las clases que figuran en el fichero de definición *Camara.h* que se encuentra disponible en el repositorio de código de la asignatura. Se construirá un programa para demostrar la utilización de varias cámaras simultáneamente para visualizar un objeto como suele suceder en aplicaciones de CAD.

Proceso:

1. Estudiar el tema de Modelado Visual de teoría
2. Analizar el código suministrado y la documentación de la práctica
3. Desarrollar la parte común de la transformación de la vista en Camara::setView()
4. Completar Camara::shot() aplicando la transformación de la vista al punto y devolviendo su homogéneo
5. Desarrollar en CamaraOrtográfica::setView() la parte de la transformación de la vista particular a este tipo de cámaras
6. Comprobar el funcionamiento de CuboPar
7. Hacer lo mismo para la cámara perspectiva (dos pasos anteriores)
8. Desarrollar el sistema multivista para el Cubo y los ejes
9. Desarrollar las ampliaciones propuestas

Sesiones: 3

Apoyo:

- *Camara.h*: Fichero de definición de las clases *Camara*, *CamaraOrtografica* y *CamaraPerspectiva.*
- *Camara.cpp*: Fichero de implementación de la clases anteriores. Se suministra incompleto para que el alumno desarrolle aquellas partes que faltan.
- *CuboPar.cpp*: Fichero de validación de la clase *CamaraOrtografica*. La figura 1 muestra la salida gráfica del ejecutable.
- *CuboPer.cpp*: Fichero de validación de la clase *CamaraPerspectiva*. La figura 2 muestra la salida gráfica del ejecutable.
- *MultiVista.cpp*: Fichero de código esqueleto de una aplicación multivista.

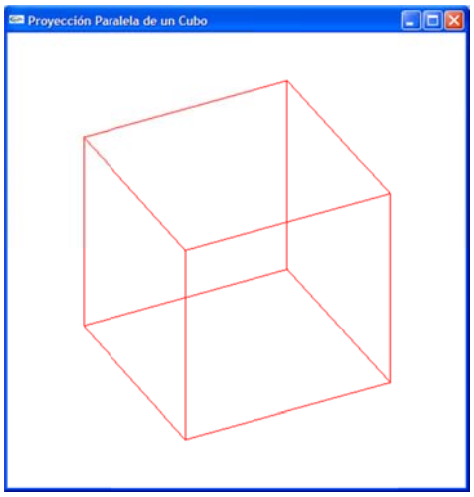


Fig. 1: Salida de CuboPar

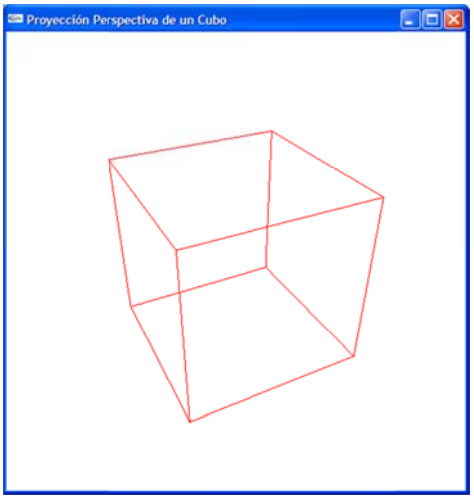


Fig. 2: Salida de CuboPer

Comentarios al código:

En el código se destaca en color rojo los métodos a implementar.

Clase *Camara* y derivadas. Se ha construido una clase genérica con los atributos y métodos comunes a las clases derivadas *CamaraOrtografica* y *CamaraPerspectiva*. El método virtual protegido *setView( )* implementará las operaciones comunes en vista ortográfica y perspectiva de traslación de *pov* (punto origen de la vista) al origen, cálculo de los ejes *u*, *v* y *w* y la rotación del sistema de la cámara para hacer coincidir ambos sistemas (vista y referencia). En definitiva, construye la matriz $R_N * T_N$. Este método debe desarrollarlo el alumno. El método *shot( )* transforma un punto al sistema de referencia de la vista, aplicando la matriz de la vista al punto y devolviendo el punto homogéneo (división por cuarta coordenada).

```
class Camara
{
protected:
        Punto pov;                                  // Posicion de la camara (def: 0,0,0)
        Vector look;                                // Orientacion de la camara (def: 0,0,-1)
        Vector up;                                  // Arriba en la camara (def: 0,1,0)
        float aspectRatio;                          // Razon de aspecto (def: 4/3)
        float near;                                 // Distancia al plano frontal (def: 1)
        float far;                                  // Distancia al plano trasero (def: 10)
        Transformacion view;                        // Matriz de transformación al sistema de la vista
        int ready;                                  // Indica si la camara esta lista para disparar
        virtual void setView();                     // Actualiza la vista
public:
        Camara();                                   // Constructor por defecto
        void at(Punto pos);                         // Posiciona la camara
        void lookAt(Punto poi);                     // Orienta la camara mirando hacia el punto de interés poi
        void lookTo(Vector to);                     // Orienta la camara mirando en esa direccion
        void setVertical(Vector v);                 // Indica el vector up
        void setAspectRatio(float ratio);           // Cambia las proporciones de la foto
        void setFOV(float neardistance, float fardistance);  // Indica los limites del campo visual
        Punto shot(Punto p);                        // Transforma un punto al sistema de referencia propio
        Transformacion getview();                   // Devuelve la matriz de la vista
};
```

Cada clase derivada, *CamaraOrtografica* y *CamaraPerspectiva*, implementará la parte diferente $S_{NO} * T_{CERCA}$ o $M_{PP} * S_{NP}$ respectivamente. Se recomienda no efectuar la proyección ortográfica M_{ORTO} para conservar la *z* de los puntos para posibles futuros usos (recortado, visibilidad, etc). Así, el resultado de *calcularVista( )* en vista perspectiva debe ser una transformación composición de las matrices:

$$M_{PO} * S_{NP} * R_N * T_N$$

y el de la vista ortográfica:

$$S_{NO} * T_{CERCA} * R_N * T_N$$

quedando la matriz de la vista lista para aplicar a los puntos en *shot( )*.

```
class CamaraOrtografica: public Camara
{
protected:
        float height;                       // Altura de la foto (def: 2)
        void setView();                     // Calculo de la transformacion de la vista
public:
        CamaraOrtografica();                // Constructor por defecto
        void setHeight(float h);            // Cambia la altura del cuadro que saldra en la foto (zoom)
        void getParam(Punto &posicion, Vector &hacia, Vector &vertical, float &alt, float &anch, float &cca, float &ljs)const;
};
class CamaraPerspectiva: public Camara
{
protected:
        float verticalAngle;                // Apertura vertical del objetivo
        void setView();                     // Calcula de la transformacion de la vista  (incluida transformacion perspectiva)
public:
        CamaraPerspectiva();                // Constructor por defecto
        void setVerticalAperture(float av); // Cambia la apertura vertical del objetivo
        void getParam(Punto &posicion, Vector &hacia, Vector &vertical,   float &angV, float &angH, float &cerca, float &lejos)const;
};
```

Se recuerda que las funciones matemáticas trigonométricas aceptan como argumento radianes mientras que con los parámetros de la vista se trabaja en grados.

Se recuerda que el parámetro *aspectRatio* en la vista perspectiva relaciona las tangentes del semiángulo horizontal y del semiángulo vertical así: *tan(verticalAngle/2)*aspectRatio = tan(horizontalAngle/2).*

MultiVista. El código suministrado incluye las funciones básicas de GLUT para la activación de la ventana de dibujo y las funciones típicas de *reshape()* y *display()*. En la práctica concurren tres rectángulos, la ventana del mundo real de 8 unidades de alto y con la misma razón de aspecto que el marco, la ventana canónica de la vista de 2x2 y el marco de dibujo de 500x500 inicialmente y que el usuario puede redimensionar arrastrando con el cursor.

La razón de aspecto se fija en *reshape()* así:

```
camara.setAspectRatio(w/(float)h);
```

donde *w* y *h* son las dimensiones del marco de dibujo.

Cada cámara toma una fotografía del objeto diferente con una ventana del mundo real de 8aspectRatio x 8. Esta foto se enmarca en una ventana canónica de 2x2. Para hacerla coincidir con un cuadrante de 1x1 de esta misma ventana hay escalar la foto ½ en cada dirección y trasladarla ±½ en cada dirección según el cuadrante de destino. Obsérvese que la razón de aspecto se conserva al ser los escalados iguales en *x* e *y*. Por último, las cuatro fotos se encajan en el marco de dibujo gracias a las funciones:

```
glViewport(w,h);
glOrtho2D(-1,1,-1,1,-10000,10000);
```

que hacen coincidir la ventana de la vista de 2x2 con un marco de *w*x*h*.

Se definirán tres cámaras ortográficas y una perspectiva. Cada una tomará una foto que se mostrará en cuatro marcos (noroeste, noreste, sudoeste y sudeste) distribuidos en la ventana.

```
#define ALTO  8.0      //Medidas de la ventana del mundo real

//Cámaras
CamaraPerspectiva canon;
CamaraOrtografica cenital,perfil,frontal;

//Transformaciones que se aplican a las fotos para situarlas en la ventana
Transformacion NO,NE,SO,SE;
```

Las cámaras se sitúan en posición y se orientan en *myinit( )*.

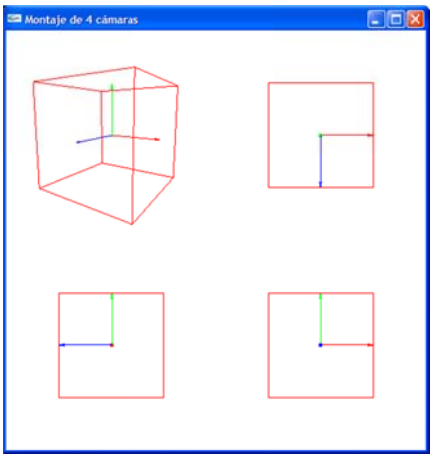
Las transformaciones necesarias para situar la foto en su cuadrante se definen en *myinit( )*. Por ejemplo,

```
NO.translation(Vector(-1/2,1/2,0.0));
NO.scale(1/2,1/2);
```

En la función *display( )* de atención al evento de redibujo, se debe hacer las cuatro fotos de los objetos a representar.

```
void display(void)
{
 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
 //Para cada cámara y transformación de cuadrante
 displayAxis(perfil,SE);
 ...
 displayCube(perfil,SE);
 ...
 glFlush();
}
```

La figura 3 muestra una posible salida de la aplicación a construir. Obviamente es posible variar el tamaño de la ventana del mundo real, incluir transformaciones del modelo previas a la de la vista, etc.



tivista

Evaluación de la práctica

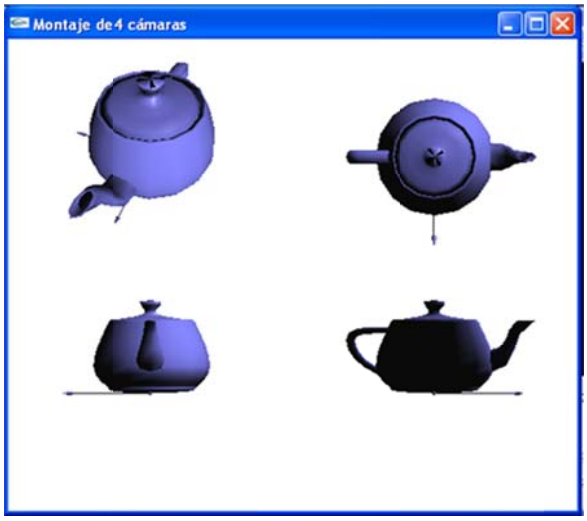
La práctica puntúa 1,5 puntos.

Parte mínima. Se obtienen 0,75 puntos si:

- Se construye correctamente la clase Camara y sus derivadas según los requisitos
- Dibujo correcto de CuboPar y CuboPer
- Se construye un programa que dibuje un Cubo y unos ejes, usando cuatro cámaras (planta, alzado, perfil y perspectiva)

Parte adicional. Se valorará, para los 0,75 puntos restantes, lo siguiente:

- La interacción mediante ratón en la vista perspectiva para variar el punto de vista (posición y zoom) manteniendo el punto de interés (origen)
- La animación de la figura en la vista perspectiva
- El dibujo de una tetera utilizando Teapot.h y SuperficieBezier
- La edición interactiva de la figura geométrica por arrastre de puntos en las vistas de planta, alzado y perfil actualizándose el resto de vistas convenientemente
- Cualquier otra mejora a la aplicación



La figura 4 ofrece un ejemplo de las ampliaciones de la práctica.