Informática Gráfica I

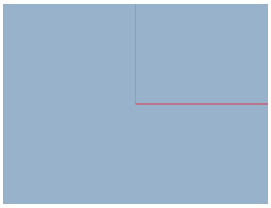
Curso 2023/2024

Facultad de Informática
Universidad Complutense de Madrid

Entrega I Apartados del 1 al 17
Fecha de entrega: 22 de febrero de 2024

Apartado 1

Localiza el comando que fija el color de fondo y cambia el color a $(0.6, 0.7, 0.8)$.



Apartado 2

En la clase `Mesh`, define el método:

```
static Mesh* generateRegularPolygon(GLuint num, GLdouble r)
```

que genere los `num` vértices que forman el polígono regular inscrito en la circunferencia de radio r, sobre el plano $Z = 0$, centrada en el origen. Utiliza la primitiva `GL_LINE_LOOP`. Recuerda que las ecuaciones de una circunferencia de centro $C = (C_x, C_y)$ y radio R sobre el plano $Z = 0$ son:

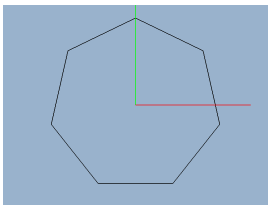
$$x = C_x + R \cdot \cos(\alpha)$$

$$y = C_y + R \cdot \sin(\alpha)$$

Genera los vértices empezando por el que se encuentra en el eje Y (α=90°) y, para los siguientes, aumenta el ángulo en `360°/num` (ojo con la división). Usa las funciones trigonométricas `cos(alpha)` y `sin(alpha)` de **glm**, que requieren que el ángulo `alpha` esté en radianes, para lo que puedes usar el conversor de **glm** para `radians(alpha)`, que pasa alfa grados a radianes.

Apartado 3

Define la clase `RegularPolygon` que hereda de `Abs_Entity` y cuya malla se construye usando el método del apartado anterior. Incorpora un objeto de esta nueva clase a la escena. En la captura adjunta se muestra un heptágono regular.



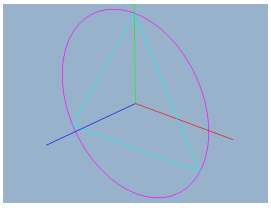
Apartado 4

Añade a la clase `Abs_Entity` un atributo **glm**`::dvec4 mColor`, para dotar de color a una entidad, sin tener que dar color a los vértices de su malla. Inicializa este atributo a 1 en la constructora (`mColor(1)`), y define sus métodos `get` y `set`. Modifica el método `render()` de la clase `Polygon` para que tenga en cuenta el color. Para establecer el color utiliza el comando `glColor4d(mColor.r, mColor.g, mColor.b, mColor.a)`.

No olvides restablecer el color por defecto antes de terminar la renderización.

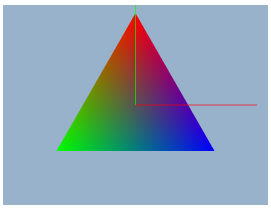
Apartado 5

Añade a la escena un triángulo cian y una circunferencia magenta como objetos de la clase `RegularPolygon`, tal como se muestra en la figura.



Apartado 6

Define la clase `RGBTriangle` que hereda de `Abs_Entity` y cuyos objetos se renderizan como el de la captura de la imagen. Observa que solo tienes que añadir colores apropiados a los vértices de una malla triangular de la clase `RegularPolygon`. Añade uno de estos triángulos a la escena.



Apartado 7

Redefine el método `render()` para establecer que el triángulo se rellene por la cara **FRONT** mientras que por la cara **BACK** se dibuja con líneas. Haz lo mismo, pero que las caras traseras se dibujen con puntos.

Apartado 8

Define el método:

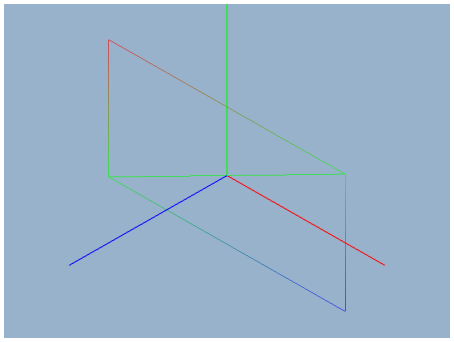
```
static Mesh* generateRectangle(GLdouble w, GLdouble h)
```

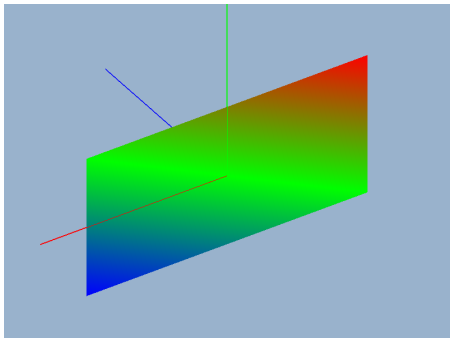
que genera los cuatro vértices del rectángulo centrado en el origen, sobre el plano $Z = 0$, de ancho w y alto h. Utiliza la primitiva `GL_TRIANGLE_STRIP`.

Define el método:

```
static Mesh* generateRGBRectangle(GLdouble w, GLdouble h)
```

que añade un color primario a cada vértice (un color se repite), como se muestra en las capturas. Define la clase `RGBRectangle` que hereda de `Abs_Entity`, y añade una entidad de esta clase a la escena. Redefine su método `render()` para establecer que los triángulos se rellenen por la cara **BACK** y se muestren con líneas, por la cara **FRONT**.



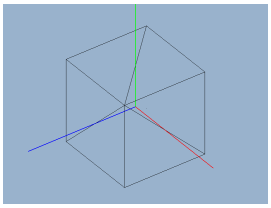


Apartado 9

Define el método:

```
static Mesh* generateCube(GLdouble length)
```

que construye la malla de un cubo (hexaedro) con arista de tamaño `length`, centrado en el origen. Define la clase `Cube` que hereda de `Abs_Entity`, y añade una entidad de esta clase a la escena. Renderízalo con las caras frontales en modo línea (con color negro) y las traseras, en modo punto, como en la captura adjunta.

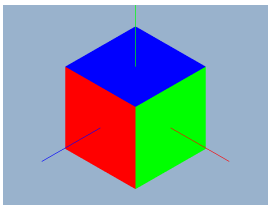


Apartado 10

Extiende la malla anterior con color en los vértices definiendo el método estático:

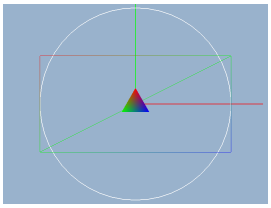
```
static Mesh* generateRGBCubeTriangles(GLdouble length)
```

El color es el que se muestra en la captura. Define la clase `RGBCube` que hereda de `Abs_Entity`, y añade una entidad de esta clase a la escena.



Apartado 11

Construye sendas escenas, una bidimensional con un rectángulo como el del apartado 8 que contiene en su interior un pequeño triángulo RGB, como el del 6, al que rodea una circunferencia como la del apartado 5, y otra escena tridimensional con un cubo como el del apartado anterior.



Apartado 12

Define el método setter `setScene(id)` del atributo `mId` de la clase `Scene`. Implementa los eventos de teclado **'0'** y **'1'** para permitir cambiar entre la escena **0**, que será la bidimensional, y la **1**, que será la tridimensional.

Apartado 13

Añade a la clase `Abs_Entity` un método `virtual void update() {}` que se usa para modificar la `mModelMat` de aquellas entidades que la cambien, por ejemplo, en animaciones. Añade a la clase `Scene` un método `void update()` que haga que las entidades de `gObjects` se actualicen mediante su método `update()`. Define en `IG1App` el evento de la tecla **'u'** para hacer que la escena se actualice con una llamada a su método `update()`.

Apartado 14

Coloca el triángulo RGB de la escena **0** en el punto **(R, 0)**, siendo R el radio de la circunferencia de esa escena.

Apartado 15

Define el método `update()` en la clase `RGBTriangle` de forma que el triángulo de esta clase de la escena **0**, rote en horario sobre sí mismo a la par que lo hace en anti horario sobre la circunferencia.

Apartado 16

Programa el método `update()` (sin argumentos) en la clase `IG1App`, que es usado por el callback de `glutIdleFunc`. Haz que este método se ejecute en el evento de teclado **'U'**.

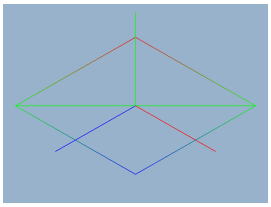
Apartado 17

(Opcional) Programa el método `update()` de la clase `RGBCube` tal como se muestra en la grabación "*demo de la escena 1*".

Entrega II Apartados del 18 al 38
Fecha de entrega: 8 de marzo de 2024

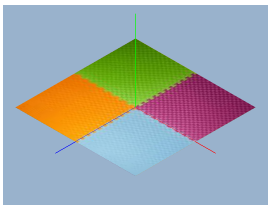
Apartado 18

Define la entidad `Ground` cuyos objetos se renderizan como rectángulos centrados en el origen que descansan sobre el plano $Y = 0$. En la constructora, utiliza la malla `generateRectangle()` definida más arriba y establece la matriz de modelado para que el suelo se muestre siempre en posición horizontal.



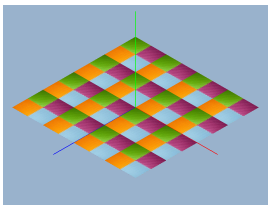
Apartado 19

Define en la clase `Mesh` el método `static Mesh* generateRectangleTexCor(GLdouble w, GLdouble h)` que añade coordenadas de textura para cubrir el rectángulo con una textura. Modifica la constructora de `Ground` y su renderización para que se muestre el suelo con la textura de la captura adjunta. No olvides modularla con el color blanco.



Apartado 20

Modifica el método del apartado anterior y define el método `static Mesh* generaRectangleTexCor(GLdouble w, GLdouble h, GLuint rw, GLuint rh)` que genera coordenadas de textura para que la imagen de la textura embaldose el suelo, repitiéndola rw veces a lo ancho y rh veces a lo alto. En la captura, $rw = rh = 4$.

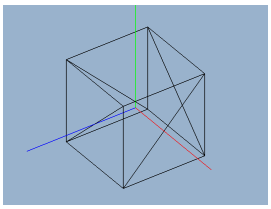


Apartado 21

Define el método `static Mesh* generateBoxOutline(GLdouble length)` que genera los vértices del contorno de un cubo, sin tapas, centrado en los tres ejes, con lado de tamaño longitud. Utiliza la primitiva `GL_TRIANGLE_STRIP`. Recuerda que el número de vértices, con esta primitiva, es 10: los 8 del cubo más 2 repetidos para cerrar el contorno.

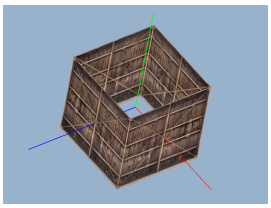
Apartado 22

Define la clase `BoxOutline` que hereda de `Abs_Entity` y cuyos objetos se renderizan como cubos sin tapas, usando la malla del apartado anterior. En la captura se muestra el contorno de una caja, con las caras frontales y traseras en modo línea.



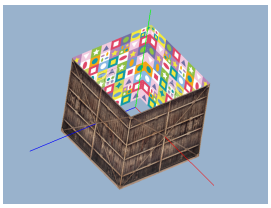
Apartado 23

Añade coordenadas de textura a la malla del contorno de una caja, definiendo el método `static Mesh* generateBoxOutlineTexCor(GLdouble length)`. Haz que la escena contenga el contorno de una caja con la textura que se muestra en la captura adjunta, repetida por las caras del contorno.



Apartado 24

Modifica el método `render()` de la clase `BoxOutline` de forma que se pueda renderizar la caja con dos texturas, una para el exterior y otra para el interior. Añade para ello otro atributo de tipo `Texture*` y utiliza apropiadamente el *culling*.

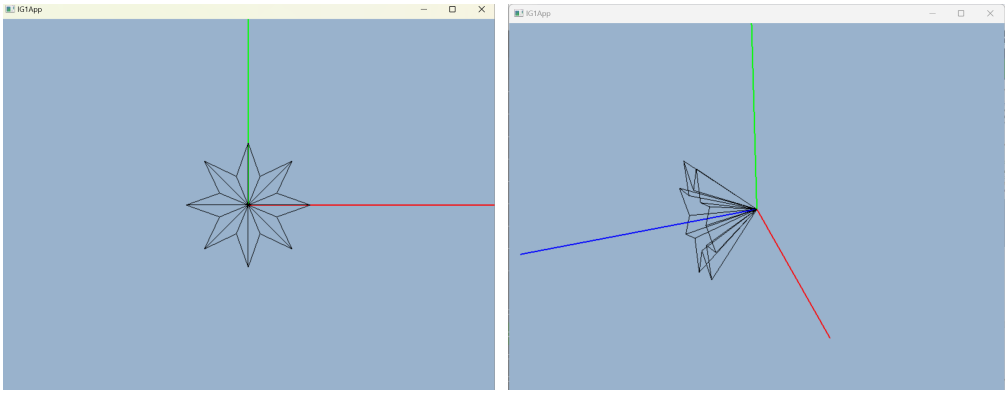


Apartado 25

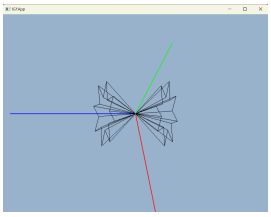
Define el método:

```
static Mesh* generateStar3D(GLdouble re, GLuint np, GLdouble h)
```

que genera los vértices de una estrella de `np` puntas situadas en los puntos de una circunferencia de radio exterior `re` centrada en el plano $Z = h$, como la que se muestra en las capturas. Utiliza la primitiva `GL_TRIANGLE_FAN` tomando como primer vértice el origen $(0, 0, 0)$. Los puntos internos se encuentran en una circunferencia de radio $ri = re/2$.



Apartado 26



Define la clase `Star3D` que hereda de `Abs_Entity` y cuyos objetos se renderizan en estrellas como las mostradas. Modifica el método `render()` de esta clase de forma que se muestren no una sino dos estrellas unidas por el origen, tal como se muestra en la captura.

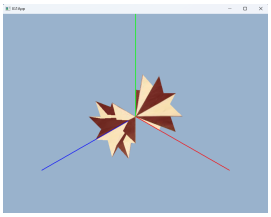
Apartado 27

Define el método `update()` de la clase `Star3D` de forma que las dos estrellas enfrentadas roten coordinadamente sobre su eje Z a la vez que giran sobre su eje Y.

Apartado 28

Define el método:

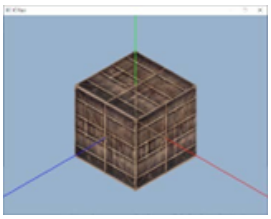
```
static Mesh* generateStar3DTexCor(GLdouble re, GLuint np, GLdouble h)
```



que genera coordenadas de textura para la malla de una estrella. Modifica la constructora de Estrella3D y el método `render()` para renderizar la estrella con textura tal como se muestra en la captura, con una estrella de 8 puntas.

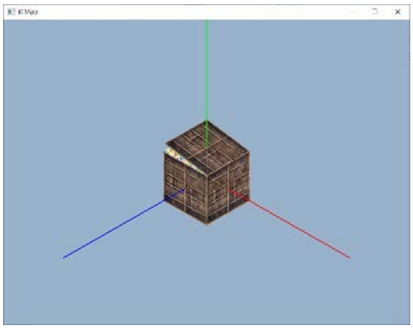
Apartado 29

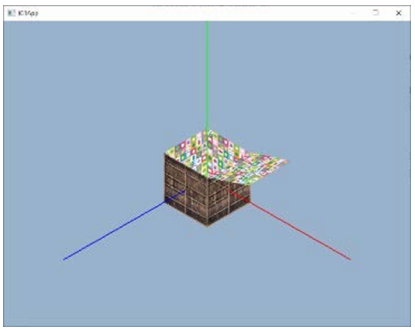
(Opcional) Define la clase `Box` mediante la malla de un contorno de caja junto con dos mallas de rectángulo más, una para la tapa y otra para el fondo. La renderización de una caja se muestra en la captura. Aunque no se ven, las caras interiores de la caja tienen todas, la textura interior del contorno de una caja.



Apartado 30

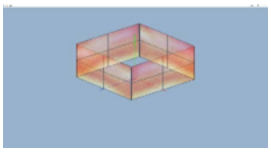
(Opcional) Define el método `update()` de la clase `Box` de forma que la tapa se abra 180^{o} para después volver a cerrase y así sucesivamente.





Apartado 31

Define la clase `GlassParapet` cuyos objetos se renderizan en contornos de caja con una textura traslúcida en todas sus caras, tal como se muestra en la captura adjunta.



Apartado 32

(Opcional) En la clase `Texture`, programa un método

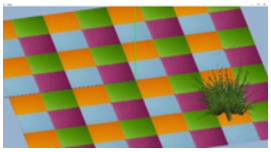
```
void load(const string& BMP_Name, u8vec3 color, GLubyte alpha)
```

que cargue una textura localizada en el path dado por el primer argumento, cambiando la componente alfa de los texeles cuyo color sea el dado en el segundo argumento, por el valor especificado por el tercer argumento. El resto de los *texeles* mantiene su valor alfa original.

Apartado 33

(Opcional) Utilizando el método del apartado anterior, carga la textura `grass.bmp`, con el color de fondo (negro) transparente, manteniendo la opacidad en el resto.

Apartado 34



(Opcional) Define la clase `Grass` cuyos objetos se renderizan encima del suelo, en una esquina, mostrando la textura anterior (con fondo transparente), rotada y renderizada tres veces.

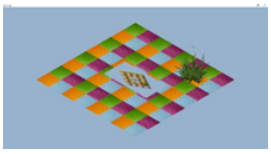
Apartado 35

En la clase `Texture`, define un método

```
void loadColorBuffer(GLsizei width, GLsizei height, GLuint buffer=GL_FRONT)
```

que cargue el buffer de color (frontal o trasero) dado por el tercer argumento, como una textura de dimensiones dadas por los parámetros primero y segundo.

Apartado 36



Define la clase `Photo` que hereda de `Abs_Entity` y cuyos objetos se renderizan en un rectángulo centrado sobre el suelo, tal como se muestra en la captura adjunta. El rectángulo tiene adosada una textura obtenida de la imagen que carga el método del apartado anterior. Define el método `update()` de esta clase de forma que su atributo `mTexture` se actualice a la textura de este rectángulo.

Apartado 37

(Opcional) Implementa el evento de teclado **F** que guarda la imagen de la foto hecha como en el apartado anterior, como fichero `.bmp`.

Apartado 38

Define una escena que contenga un suelo, una caja con su tapa que se abre y se cierra, situada en una esquina del suelo, una estrella encima de la caja, una cristalera que rodea el suelo, unas hierbas y una foto. Evidentemente, no es obligatorio que aparezcan los apartados opcionales.

Entrega III Apartados del 39 al 55
Fecha de entrega: 4 de abril de 2024

Apartado 39

Añade a la clase `Camera` los atributos `glm::dvec3 mRight, mUpward` y `mFront`, más el método protegido `void setAxes()` que les da valor usando la función `row()`.

Apartado 40

Modifica aquellos métodos de la clase `Camera` que deban invocar el método `setAxes()` para mantener el valor de estos atributos coherentemente con cualquier cambio en la matriz de vista.

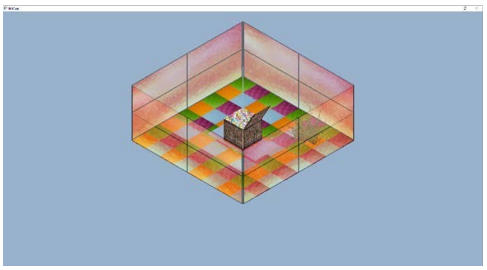
Apartado 41

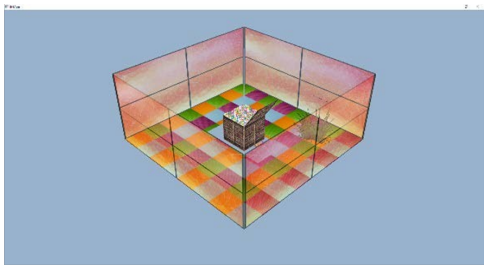
Añade a la clase `Camera`, métodos para desplazar la cámara en cada uno de sus ejes, sin cambiar la dirección de vista:

```
void moveLR(GLdouble cs); // A izquierda/A derecha
void moveFB(GLdouble cs); // Adelante/Atrás
void moveUD(GLdouble cs); // Arriba/Abajo
```

Apartado 42

Añade a la clase `Camera` un método público `void changePrj()` para cambiar de proyección ortogonal (izquierda) a perspectiva (derecha, en las capturas de más abajo)), y viceversa. El cambio de proyección está mediado por el atributo booleano `bOrto` de la clase `Camera`.





Apartado 43

Modifica aquellos métodos de la clase `Camera` afectados por el cambio de proyección (por ejemplo, `setPM()` para que el zoom siga funcionando correctamente).

Apartado 44

Define el evento de la tecla **'p'** para cambiar entre proyección ortogonal y perspectiva.

Apartado 45

Comenta las llamadas a los métodos `pitch()`, `yaw()` y `roll()` de las teclas especiales y prueba allí que funcionan correctamente los métodos del apartado 41, tanto con proyección ortogonal como con proyección perspectiva.

Apartado 46

Añade a la clase `Camera`, métodos para rotar la cámara en cada uno de sus ejes u, v y n:

```
void pitchReal(GLdouble cs);
void yawReal(GLdouble cs);
void rollReal(GLdouble cs);
```

Como con las traslaciones, prueba en las teclas especiales que estas rotaciones funcionan correctamente. Hay una demo en el Campus Virtual que muestra cuál debe ser el comportamiento esperado de cada una de ellas.

Apartado 47

(Opcional) Sitúa una cámara encima del triángulo que rota alrededor del origen (escena de la primera entrega), mirando al centro de este, y haz que la cámara lo acompañe tanto en su rotación sobre sí mismo como en su movimiento por la circunferencia. Para ello define en la clase `Camera` un método `update()` que hace que la cámara imite los dos movimientos del triángulo. Invoca este método dentro del método `update()` de la clase `IG1App` apropiadamente para que se muevan, de forma coordinada, triángulo y cámara, cuando la ejecución se encuentra en stand-by. Hay una demo en el Campus Virtual.

Apartado 48

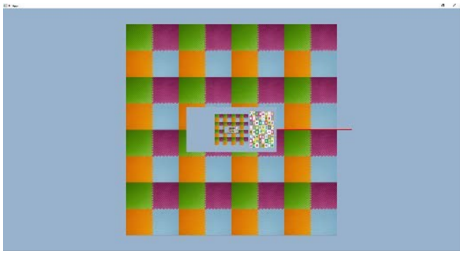
Incorpora a la clase `Camera` el método `orbit(incAng, incY)` para desplazar la cámara a lo largo de una circunferencia situada sobre el plano **XZ**, a una determinada altura sobre el eje Y, alrededor de `mLook`, tal como aparece definido este método en las transparencias.

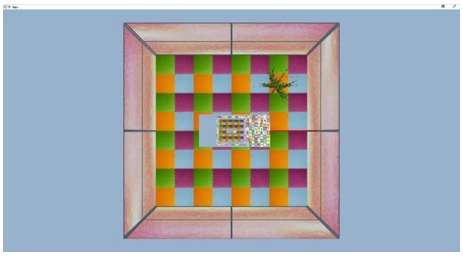
Apartado 49

Modifica los métodos `set2D()` y `set3D()` de manera que se inicialicen de forma coherente los atributos de la cámara mEye, `mLook`, `mUp`, `mAng` y `mRadio`.

Apartado 50

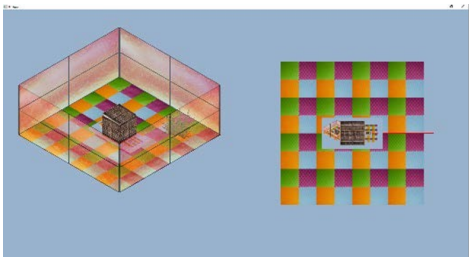
Añade a la clase `Camera` un método `setCenital()` que muestra la escena desde una posición cenital. Abajo tienes dos capturas que muestran una escena cenitalmente en proyección ortogonal (izquierda) y perspectiva (derecha).

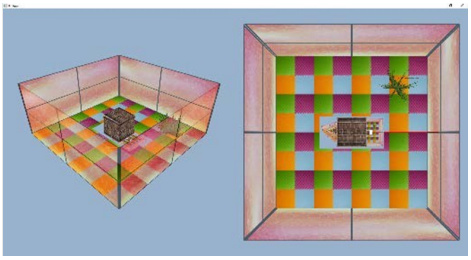




Apartado 51

Define el evento de la tecla **'k'** para renderizar dos vistas de forma simultánea, tal como se muestra en las capturas de abajo. En cada una de ellas, a la izquierda se ve la escena con la cámara en su posición normal 3D, y a la derecha con la cámara en posición cenital. La captura de la izquierda está hecha con proyección ortogonal y la de la derecha con proyección perspectiva. Añade a la aplicación un atributo bool `m2Vistas` para mediar, en la tecla **'k'**, entre la renderización de una vista o de dos.





Apartado 52

Añade a `IG1App` dos nuevos atributos: `dvec2 mMouseCoord`, para guardar las coordenadas del ratón, e `int mMouseButt`, para guardar el botón pulsado.

Apartado 53

Programa los siguientes callbacks de `IG1App`, definidos tal como se han explicado en clase:

- `void mouse(int button, int state, int x, int y)`: captura en `mMouseCoord` las coordenadas del ratón (x, y), y en `mMouseButt`, el botón pulsado.
- `void motion(int x, int y)`: captura las coordenadas del ratón, obtiene el desplazamiento con respecto a las anteriores coordenadas y, si el botón pulsado es el derecho, mueve la cámara en sus ejes `mRight` (horizontal) y `mUpward` (vertical) el correspondiente desplazamiento, mientras que si es el botón izquierdo, rota la cámara alrededor de la escena.
- `void mouseWheel(int n, int d, int x, int y)`: si no está pulsada ninguna tecla modificadora, desplaza la cámara en su dirección de vista (eje `mFront`), hacia delante/atrás según sea d positivo/negativo; si se pulsa la tecla **Ctrl**, escala la escena, de nuevo según el valor de d.

Apartado 54

(Opcional) Renderiza dos escenas diferentes dividiendo la ventana en dos puertos de vista. En el de la izquierda se mostrará la escena 1, es decir, la de la cristalera, y en el de la derecha, la escena 0, es decir, la bidimensional de la primera entrega.

Apartado 55

(Opcional) Modifica la programación de los eventos de ratón de forma que este pueda actuar independientemente en cada puerto de vista, dependiendo de en cual se encuentre el cursor. Modifica el evento de la tecla **'u'** de forma que se actualice la escena del puerto de vista que contiene el cursor del ratón. Modifica el evento de la tecla **'p'** de forma que se cambie el tipo de proyección de la escena del puerto de vista que contiene el cursor del ratón. La foto puede seguir mostrando la ventana entera.