

# Informática Gráfica I

Curso 2024/2025

Facultad de Informática  
Universidad Complutense de Madrid

## Entrega I Apartados del 1 al 17

**Fecha de entrega: 13 de febrero de 2025**

### Instrucciones de la práctica

Para llevar a cabo este trabajo se tendrá en cuenta lo siguiente:

- Esta práctica se compone de varias escenas.
- Para realizar cada escena es necesario implementar una serie de pasos, en forma de apartados.
- El código que se desarrolle para cada escena deberá ser reutilizable por las escenas posteriores.
- Se tendrá especial atención a la calidad del código: inexistencia de fugas de memoria, código bien organizado, comentado y legible, se siguen los principios de la programación orientada a objetos, entre otros.

### Rúbrica de evaluación

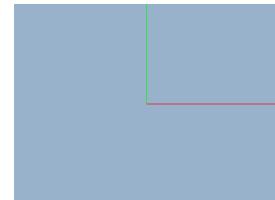
Criterio 1	Pesos
Escena 1	2.5
Escena 2	2.5
Escena 3	2.5
Calidad del código	1.5
Ausencia de fugas	1
<b>Total</b>	<b>10.0</b>
Opcional	+1
Defensa	-1.5

## Escena 1: Polígonos sin relleno

---

### 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:

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

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

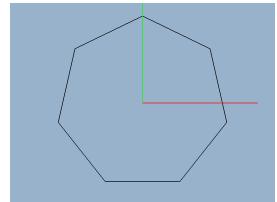
Genera los vértices empezando por el que se encuentra en el eje Y ( $\alpha=90^\circ$ ) y, para los siguientes, aumenta el ángulo en  $360^\circ/\text{num}$  (ojo con la división). Usa las funciones trigonométricas `cos(alpha)` y `sin(alpha)` de `glm`, que requieren que el ángulo `alpha` esté en radianes,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

### Apartado 3 .....

Define una clase `SingleColorEntity` que extienda `Abs_Entity` con un atributo `mColor` de tipo `glm::vec4` para dotar de color a una entidad sin tener que dar color a los vértices de su malla. Este atributo se podrá consultar y modificar con sendos métodos `color` y `setColor`. El constructor recibirá también el color inicial como argumento, que tendrá como valor por defecto el blanco (`glm::vec4 color = 1`). Además, este constructor seleccionará el `shader simple`. Por último, sobrescribe el método `render()` con una definición similar a la de `EntityWithColors`, pero que que cargue el color en la GPU con `mShader->setUniform("color", mColor)`.

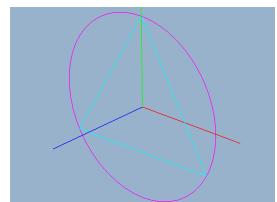
#### Apartado 4 .....

Define la clase RegularPolygon que hereda de SingleColorEntity 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, a modo de ejemplo, un heptágono regular; pero debería ser válido para cualquier polígono.



#### 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.



## Escena 2: Polígonos con relleno

---

### Apartado 6 .....

Define la clase RGBTriangle que hereda de EntityWithColors 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 .....

Core Profile no admite `glPolygonMode` diferenciado para la cara delantera y trasera porque es un caso particular del *culling*.

Utilizando *culling*, redefine el método `render()` para 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:

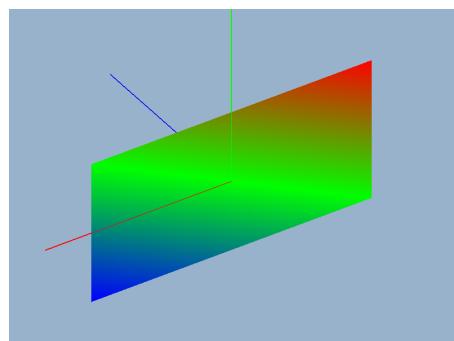
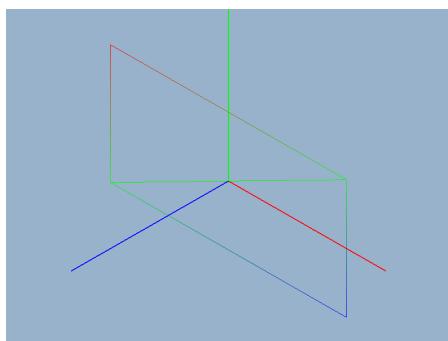
```
static Mesh* generateRectangle(GLdouble w, GLdouble h)
```

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 EntityWithColors, y añade una entidad de esta clase a la escena. Utilizando *culling*, redefine su método `render()` para establecer que los triángulos se rellenen por la cara **BACK** y se muestren con líneas, por la cara **FRONT**.

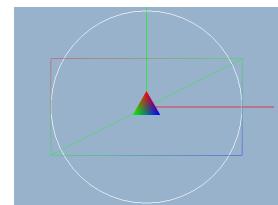


### Apartado 9

Refactoriza el código para que cada escena sea una subclase de la clase genérica Scene. Para ello, convierte en virtual el método `init` de Scene y define una clase `Scene0` como subclase de Scene que defina su método `init` para la escena inicial. Ajusta la inicialización de `IG1App` para que construya una escena de tipo `Scene0`.

### Apartado 10

Construye una escena 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.



### Apartado 11

Coloca el triángulo RGB de la escena 1 en el punto  $(R, 0)$ , siendo  $R$  el radio de la circunferencia de esa escena.

### Apartado 12

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 virtual `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 13

Sobrescribe el método update() en la clase RGBTriangle de forma que el triángulo de esta clase de la escena 1, rote en horario sobre sí mismo a la par que lo hace en anti horario sobre la circunferencia.

## Apartado 14 .....

Implementa la actualización continua de la escena invocando periódicamente el método update de IG1App. Para ello, declara en dicha clase una constante FRAME\_DURATION, una variable booleana mUpdateEnabled y una variable mNextUpdate de tipo double. En el bucle del método run, cuando mUpdateEnabled sea cierto, utiliza la función `double glfwGetTime()` para obtener el tiempo actual y llamar al método update cada FRAME\_DURATION segundos. En ese caso, en lugar de glfwWaitEvents habrá que usar

```
void glfwWaitEventsTimeout(double timeout);
```

con el tiempo restante para llegar a mNextUpdate, que se irá actualizando oportunamente.

Haz que mUpdateEnabled se active y desactive con el evento de teclado **U**.

## Escena 3: Cubo con color

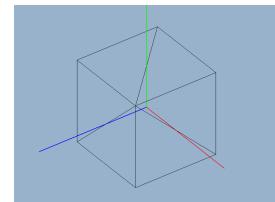
---

### Apartado 15 .....

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 `SingleColorEntity`, 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.

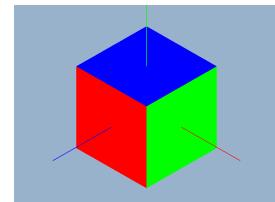


### Apartado 16 .....

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 `EntityWithColors`, y añade una entidad de esta clase a la escena.



### Apartado 17 .....

*(Opcional)* Programa el método `update()` de la clase `RGBCube` tal como se muestra en la grabación “*demonstración de la escena 2*”.

**Entrega II Apartados del 18 al 37**  
**Fecha de entrega: 13 de marzo de 2025**

### Instrucciones de la práctica

Para llevar a cabo este trabajo se tendrá en cuenta lo siguiente:

- Esta práctica se compone de una única escena con diversos elementos.
- Para realizar cada escena es necesario implementar una serie de pasos, en forma de apartados.
- El código que se desarrolle para cada escena deberá ser reutilizable por las escenas posteriores, incluidas en esta u otra práctica.
- Se tendrá especial atención a la calidad del código: inexistencia de fugas de memoria, código bien organizado, comentado y legible, se siguen los principios de la programación orientada a objetos, entre otros.

### Rúbrica de evaluación

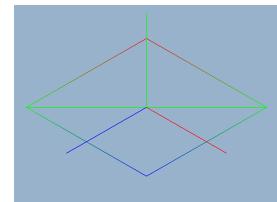
Criterio 1	Pesos
Suelo con textura	1.5
Caja con textura	1.5
Estrella doble con textura	1.5
Cristalera	1.5
Foto	1.5
Calidad del código	1.5
Ausencia de fugas	1
<b>Total</b>	<b>10.0</b>
Opcionales	+1
Defensa	-1.5

## Escena 4: Texturas

---

### Apartado 18 .....

Define la entidad `Ground` como subclase de `EntityWithColors` 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.

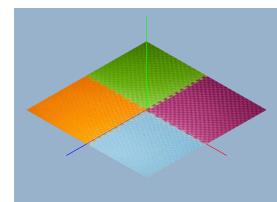


### Apartado 19 .....

Define la entidad `EntityWithTexture` para renderizar objetos con textura. Ha de guardar un atributo protegido `mTexture` de tipo `Texture*` con su textura y `mModulate` de tipo `bool` para indicar si se modulará la imagen con el color de los vértices (por defecto a `false`). El constructor seleccionará el shader `texture` y su método `render` tomará como referencia el de `EntityWithColors`, fijará el atributo uniforme `modulate` del shader y llamará a `bind` y `unbind` antes y después de renderizar la malla (solo si la textura es no nula).

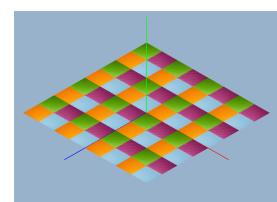
### Apartado 20 .....

Define en la clase `Mesh` el método `static Mesh* generateRectangleTexCor(GLdouble w, GLdouble h)` que añada coordenadas de textura al rectángulo para mostrar la imagen completa sobre él. Cambia la clase madre de `Ground` a `EntityWithTexture`, adapta su constructora y modifica su método `render` para que se muestre el suelo con la textura de la captura adjunta.



### Apartado 21 .....

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

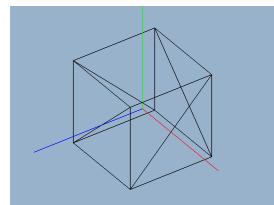


## Apartado 22

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.

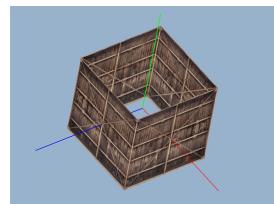
## Apartado 23

Define la clase `BoxOutline` que hereda de `SingleColorEntity` 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.



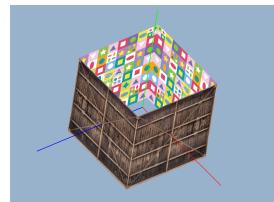
## Apartado 24

Añade coordenadas de textura a la malla del contorno de una caja, definiendo el método `static Mesh* generateBoxOutlineTexCor(GLdouble length)`, y cambia la clase madre de `BoxOutline` a `EntityWithTexture`. 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 25

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 un atributo de tipo `Texture*` (además del heredado) y utiliza apropiadamente el *culling*.

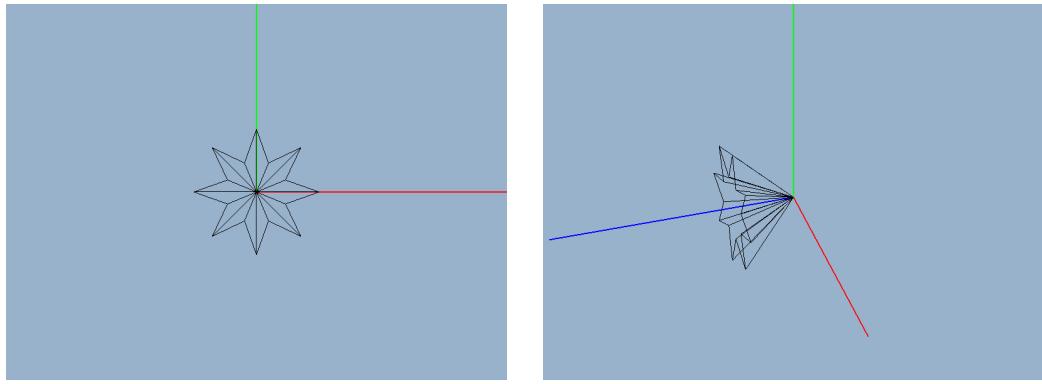


## Apartado 26

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 27 .....

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



### Apartado 28 .....

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 29 .....

Define el método:

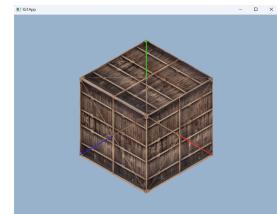
```
static Mesh* generateStar3DTexCor(GLdouble re, GLuint np, GLdouble h)
```

que genera coordenadas de textura para la malla de una estrella. Cambia la clase madre de Star3D a EntityWithTexture y adapta el método render() para renderizar la estrella con textura tal como se muestra en la captura, con una estrella de 8 puntas.



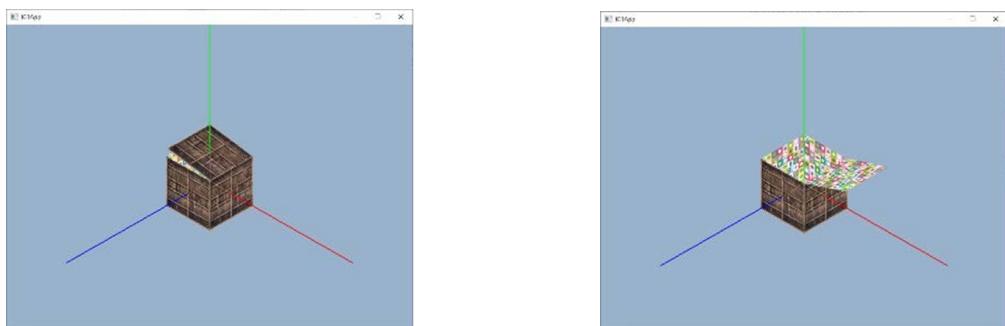
## Apartado 30 .....

**(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.



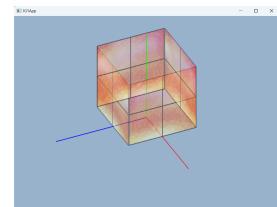
## Apartado 31 .....

**(Opcional)** Define el método update() de la clase Box de forma que la tapa se abra 180° para después volver a cerrarse y así sucesivamente.



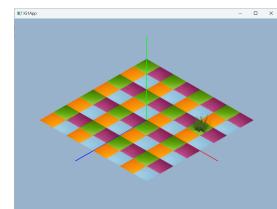
## Apartado 32 .....

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



## Apartado 33 .....

**(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. Para tener en cuenta la transparencia, utiliza el shader de fragmentos texture\_alpha, que puedes cargar con `Shader::get("texture:texture_alpha")`.



## Apartado 34 .....

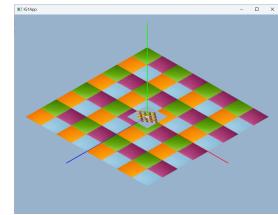
En la clase Texture, define un método

```
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 35 .....

Define la clase Photo que hereda de EntityWithTexture 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 36 .....

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

## Apartado 37 .....

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 cristalería que rodea el suelo, unas hierbas y una foto. Evidentemente, no es obligatorio que aparezcan los apartados opcionales.

**Entrega III Apartados del 38 al 53**  
**Fecha de entrega: 3 de abril de 2025**

### **Instrucciones de la práctica**

Para llevar a cabo este trabajo se tendrá en cuenta lo siguiente:

- Esta práctica se compone de movimientos de cámara y varias escenasopcionales.
- Es necesario implementar una serie de pasos, en forma de apartados.
- El código que se desarrolle deberá ser reutilizable por las escenas posteriores.
- Se tendrá especial atención a la calidad del código: inexistencia de fugas de memoria, variables con tipo correctamente definido, código bien organizado, comentado y legible, siguiendo los principios de la programación orientada a objetos.

### **Rúbrica de evaluación**

Criterio	Pesos
Movimiento LR-FB-UD	1
Cambio de Proyección	1
Movimiento pitch/roll/yaw	1
Movimiento orbital	1
Cambio de vista 2D - 3D	1
Vista cenital	1
Dos vistas	1
Movimientos de ratón	1
Calidad del código	1
Ausencia de fugas	1
<b>Total</b>	<b>10.0</b>
Defensa (individual)	-1.5

Opcionales	Pesos
Puerto de vista con dos escenas	+0.5
Ratón y puertos de vista	+0.5

## Apartado 38 .....

Añade a la clase Camera los atributos `glm::vec3 mRight, mUpward` y `mFront`, más el método protegido `void setAxes()` que les da valor usando la función `row()`.

## Apartado 39 .....

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 40 .....

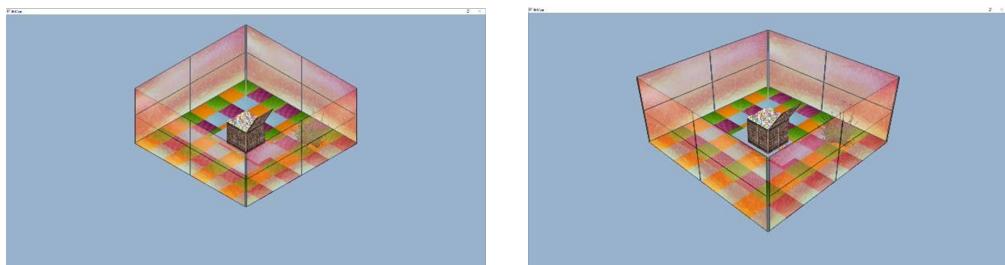
Añade a la clase Camera, métodos para desplazar la cámara en cada uno de sus ejes, sin cambiar la dirección de vista:

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

Asigna las pulsaciones `a` y `d` a `moveLR`, `w` y `s` a `moveUD`, y `W` y `S` a `moveFB` (son las mismas teclas, pero con las mayúsculas activas).

## Apartado 41 .....

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 `bOrtho` de la clase Camera.



## Apartado 42 .....

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

## Apartado 43 .....

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

## Apartado 44 .....

Comprueba que funcionan correctamente los métodos del apartado 40, tanto con proyección ortogonal como con proyección perspectiva.

## Apartado 45 .....

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

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

Asocia las teclas **←** y **→** a yawReal si no está pulsada la tecla **Ctrl** y a rollReal si lo está; y las teclas **↑** y **↓** a pitchReal. Hay una demo en el Campus Virtual que muestra cuál debe ser el comportamiento esperado de cada una de ellas.

## Apartado 46 .....

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 47 .....

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 48 .....

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 49 .....

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 booleano `m2Vistas` para mediar, en la tecla `[k]`, entre la renderización de una vista o de dos.

## Apartado 50 .....

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

## Apartado 51 .....

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

- `void mouse(int button, int state, int mods)`: captura en `mMouseCoord` las coordenadas del ratón ( $x, y$ ), y en `mMouseButt`, el botón pulsado.
- `void motion(double x, double 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(double dx, double dy)`: 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 52 .....

**(Opcional)** Renderiza dos escenas diferentes en la ventana dividida en dos puertos de vista del apartado 49. En el de la izquierda se mostrará la Escena 4, es decir, la de la cristalera, y en el de la derecha, la Escena 2, es decir, la bidimensional de la primera entrega.

## Apartado 53 .....

**(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.

**Entrega IV Apartados del 54 al 68**  
**Fecha de entrega: 24 de abril de 2025**

### **Instrucciones de la práctica**

Para llevar a cabo este trabajo se tendrá en cuenta lo siguiente:

- Esta práctica se compone de varias escenas.
- Para realizar cada escena es necesario implementar una serie de pasos, en forma de apartados.
- El código que se desarrolle deberá ser reutilizable por las escenas posteriores.
- Se tendrá especial atención a la calidad del código: inexistencia de fugas de memoria, variables con tipo correctamente definido, código bien organizado, comentado y legible, siguiendo los principios de la programación orientada a objetos.

### **Rúbrica de evaluación**

Criterio	Pesos
Escena 5 (Torus)	1
Escena 6 (Cubo indexado)	2
Escena 7 (Advanced TIE X-1)	2
Escena 8 (Planeta Tatooine)	1
Movimiento de la nave	1
Calidad del código	1
Ausencia de fugas	1
<b>Total</b>	<b>10.0</b>
Defensa (individual)	-1.5

Opcionales	Pesos
Escena 9 (Farmer)	+1

## Escena 5: Torus

### Apartado 54

Define la clase `IndexMesh` que hereda de `Mesh`, añadiendo atributos para el array de índices `std::vector<GLuint> vIndexes` y para el identificador de su *vertex object buffer* asociado `GLuint mIBO`. Sobrescribe los métodos `draw` `load` y `unload`, tal como se explica en las transparencias. Asegúrate de que dichos métodos estén declarados como virtuales en `Mesh`.

### Apartado 55

En la clase `IndexMesh`, define el método estático:

```
static IndexMesh* generateByRevolution(  
    const std::vector<glm::vec2>& profile, GLuint nSamples,  
    GLfloat angleMax = 2 * std::numbers::pi)
```

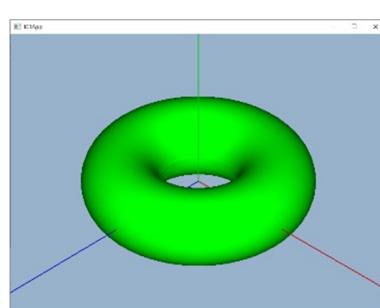
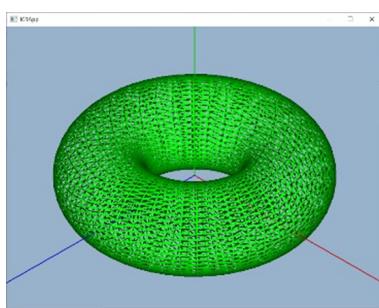
que construye, tal como se ha explicado, la malla por revolución que se obtiene al hacer girar `profile` `angleMax` radianes alrededor del eje Y, tomando `nSamples` muestras.

### Apartado 56

Define la clase `Torus` que hereda de `SingleColorEntity` y cuya constructora es de la forma:

```
Torus(GLdouble R, GLdouble r, GLuint nPoints = 40, GLuint nSamples = 40)
```

donde `r` es el grosor de la “rosquilla”, `R` es el radio de la “rosquilla”, `nSamples` es el número de muestras y `nPoints` es el número de puntos con que se aproxima la circunferencia. La malla está construida por revolución con el perfil de la captura adjunta (en ese caso, con `nPoints` igual a 12). En las siguientes capturas se muestra el toro en modo línea y en modo relleno. No obstante, para que el modo relleno se muestre como en la imagen será necesario esperar a los apartados siguientes.



Muestra este objeto en la escena con la tecla `5`.

## Escena 6: IndexedBox

---

### Apartado 57 .....

Añade a la clase `Mesh` el atributo `std::vector<glm::vec3> vNormals` para que los vértices de las mallas puedan disponer también de vector normal y el atributo `GLuint mNBO` para guardar el identificador de su *vertex array object* asociado. Modifica los métodos `load` y `unload` de `Mesh` para que contemplen el caso en que la malla tenga array de vectores normales, que se ha de registrar como el atributo número 3 del *vertex array object*. Comprueba que todas las mallas que tenías siguen renderizándose correctamente después de estos cambios.

### Apartado 58 .....

Para introducir luz en la escena, define una clase `ColorMaterialEntity` como subclase de `SingleColorEntity` que utilice el shader `simple_light`. Además, modifica el método `updateVM` de `Camera` para que fije el atributo uniforme `"lightDir"` del shader a las coordenadas uniformes de vista que se corresponden con la dirección  $(-1, -1, -1)$  en coordenadas mundiales (para ello tendrás que multiplicarla por la matriz de vista).

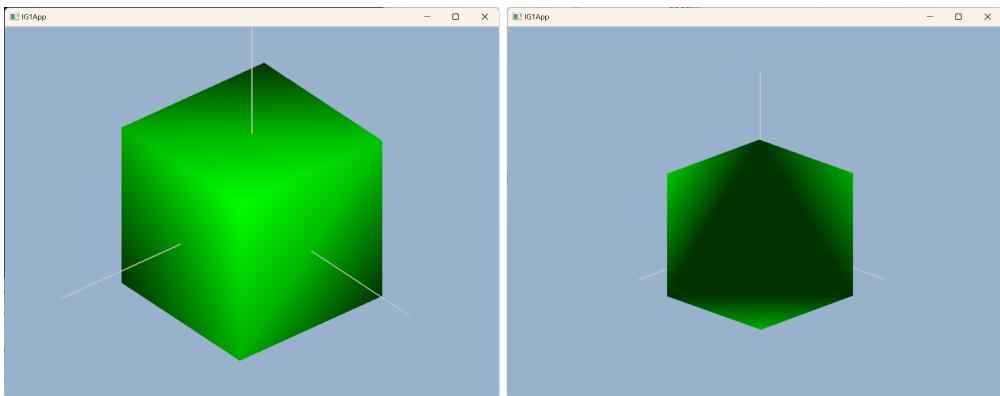
### Apartado 59 .....

En la clase `IndexMesh` define el método:

```
static IndexMesh* generateIndexedBox(GLdouble l);
```

que construye la malla indexada de un cubo centrado en el origen de arista de longitud 1, con tapa superior e inferior. La primitiva de esta malla es `GL_TRIANGLES`. Hay 8 vértices sin color asociado. Define cuidadosamente los 36 índices que, de 3 en 3, determinan las 12 caras triangulares de la malla. Recuerda que los índices de estas caras deben darse en sentido anti-horario según se mira el cubo desde su exterior.

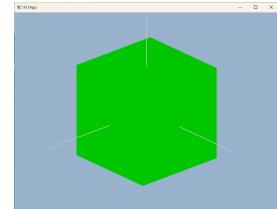
Los vectores normales puedes calcularlos a mano. Observa que el vector normal de cualquier vértice es una suma de varios vectores de los siguientes seis tipos posibles  $(\pm 1, 0, 0)$ ,  $(0, \pm 1, 0)$ ,  $(0, 0, \pm 1)$ . Por ejemplo, la esquina del cubo en el octante positivo del espacio es el vértice de coordenadas `vec3(1 / 2, 1 / 2, 1 / 2)`. Su vector normal es el vector `glm::normalize(vec3(1, 1, 2))` porque este vértice participa en 4 triángulos.



## Apartado 60 .....

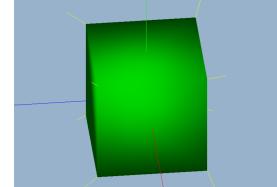
Define la clase `IndexedBox` como subclase de `ColorMaterialEntity`, utilizando la malla anterior y el color verde. Comprueba por un momento que si se cambia su clase madre a `SingleColorEntity` (es decir, si se ignoran las normales) saldrá dibujado como en la figura adjunta.

Muestra este objeto en la escena con la tecla **[6]**.



## Apartado 61 .....

Como forma de depuración, sobrescribe el método `render` de `ColorMaterialEntity` y añade una segunda renderización con el shader `normals`. Esto dibujará segmentos amarillos desde cada vértice en dirección de sus vectores normales. Comprueba que las normales tienen la dirección esperada.



Introduce en la clase `ColorMaterialEntity`, para evitar que las normales estén siempre visibles, un atributo privado `mShowNormals` y un método público `toggleShowNormals`, ambos estáticos, de tal forma que la renderización de las normales se active y desactive con la tecla **[N]**.

## Apartado 62 .....

En la clase `IndexMesh` define el método:

```
void buildNormalVectors();
```

que construye los vectores normales de una malla indexada a partir de los índices de las caras con el método de Newell, tal como se explica en las transparencias. Evidentemente, cuando utilices este método para obtener los vectores normales, la renderización del

cubo debe ser la misma que la del correspondiente apartado anterior donde los vectores normales se calculaban a mano.

### Apartado 63 .....

Añade una llamada al método `buildNormalVectors` del apartado anterior al final del método `generateByRevolution` de `IndexMesh`.

Haz que la clase `Torus` descienda `ColorMaterialEntity` y comprueba que su apariencia ya coincide con la de la imagen del Apartado 56.

## Escena 7: AdvancedTIE

### Apartado 64 .....

Crea, utilizando el método `generateByRevolution` y definiendo los perfiles correspondientes, las siguientes figuras de revolución como subclases de `ColorMaterialEntity`:

1. `Sphere(GLdouble radius, GLuint nParallels, GLuint nMeridians)` como una esfera de radio `radius` con `nParallels` paralelos y `nMeridians` meridianos.
2. `Disk(GLdouble R, GLdouble r, GLuint nRings, GLuint nSamples)` como un disco ( $r = 0$ ) o corona circular ( $r > 0$ ) de radio exterior `R` y radio interior `r` con `nRings` vértices en el perfil y `nSamples` muestras de revolución.
3. `Cone(GLdouble h, GLdouble r, GLdouble R, GLuint nRings, GLuint nSamples)` como un cono ( $r = 0$  o  $R = 0$ ), cilindro ( $R = r$ ) o tronco de cono de altura `h`, radio inferior `r`, radio superior `R`, `nRings` vértices en el perfil y `nSamples` muestras de revolución.

### Apartado 65 .....

Crea la clase `CompoundEntity` que hereda de la clase `Abs_Entity` y que dispone de un atributo protegido nuevo:

```
std::vector<Abs_Entity*> gObjects;
```

y de un método público para añadir una entidad a este vector de la entidad compuesta:

```
void addEntity(Abs_Entity* ae);
```

Define la destructora:

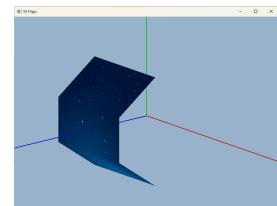
```
~CompoundEntity()
```

y, sobre todo, reescribe convenientemente los métodos `render`, `update`, `load` y `unload`.

### Apartado 66 .....

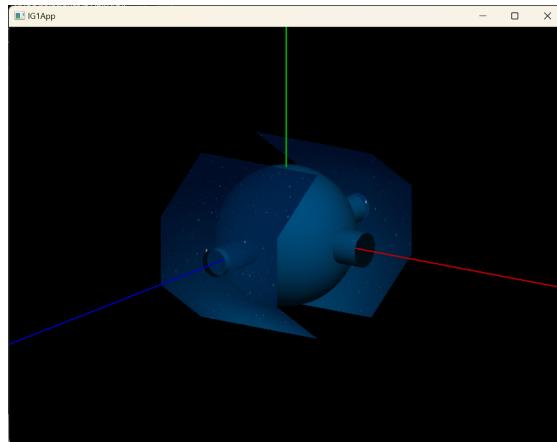
Construye el caza estelar imperial **Advanced TIE X-1**, es decir, la nave de Darth Vader. Define pues una nueva clase `AdvancedTIE` que hereda de la clase `CompoundEntity` y cuyos objetos se renderizan como la nave de la captura adjunta. El fondo negro es opcional.

Cada una de estas naves está construida como sigue: dos alas que son elementos de la nueva clase de entidad que tienes que definir `WingAdvancedTIE` y que se generan mediante un nuevo método de la clase `Mesh`, utilizando la primitiva que quieras. Las alas se renderizan con textura translúcida `noche.jpg`.



Además, tienen un eje, que es un cilindro que va de ala a ala; un núcleo central, que es una esfera cuádrica; y, por último, un morro, que es una entidad compuesta por un cilindro y un disco cuádricos, este último tapando uno de los lados del cilindro, tal como se muestra en la imagen. El color de todos estos elementos es el añil (0, 65, 106). Debes ajustarte a esta construcción.

Muestra esta escena con la tecla 7.

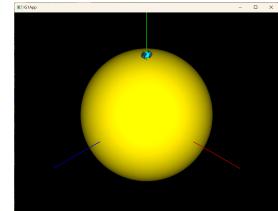


## Escena 8: Planeta Tatooine

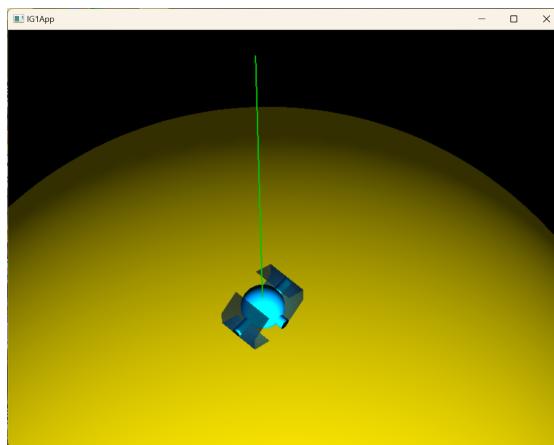
---

### Apartado 67 .....

Crea una escena sobre fondo negro, formada por la esfera del desierto planeta Tatooine (de color amarillo estándar (255, 233, 0)) y el caza imperial de Darth Vader en su polo norte, tal como se muestra en la captura adjunta.



Muestra esta escena con la tecla [8].



### Apartado 68 .....

Usando la técnica del nodo ficticio programa dos métodos en la clase Scene:

- `rotate()`: permite rotar el caza sobre sí mismo haciendo que varíe la dirección en la que apunta el morro de la nave.
- `orbit()`: permite mover el caza por el círculo máximo de la esfera que pasa por la nave, en la dirección en la que apunta el morro.

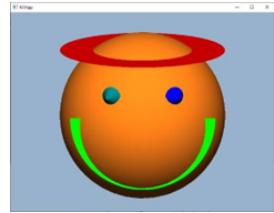
Programa los eventos de teclado de forma que estos dos métodos sean invocados, respectivamente, por las teclas `f` y `g`.

## Escena 9: Farmer

---

### Apartado 69 .....

**(Opcional)** Crea una escena que tiene, como único objeto, la cabeza del granjero que aparece en la captura adjunta y que está formada por los siguientes elementos: el sombrero, que es un disco rojo; la barba, que es un disco parcial verde; los ojos, que son conos de distinto color, azul marino y gris marengo; y la propia cabeza, que es una esfera naranja.



Reúne todos estos objetos en una entidad compuesta Farmer que mire inicialmente a la dirección positiva el eje Z, pero que se colocará en la escena con la orientación de la imagen (rotada 45 grados).

Muestra este objeto en la escena con la tecla **9**.