

## 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.

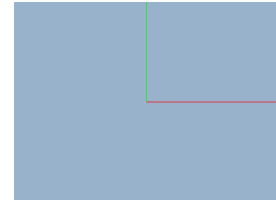
### Rúbrica de evaluación

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:

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

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

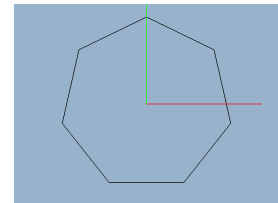
Genera los vértices empezando por el que se encuentra en el eje Y ( $\alpha=90^\circ$ ) y, para los siguientes, aumenta el ángulo en  $360^\circ/\text{num}$  (ojo con la división). Usa las funciones trigonométricas `cos(alpha)` y `sin(alpha)` de **glm**, que requieren que el ángulo alpha esté en radianes, 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 cargue el color en la GPU con `mShader->setUniform("color", mColor)`.

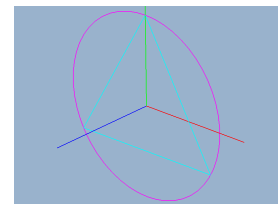
#### Apartado 4 .....

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.

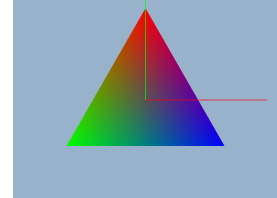


## Escena 2: Polígonos con relleno

---

### 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:

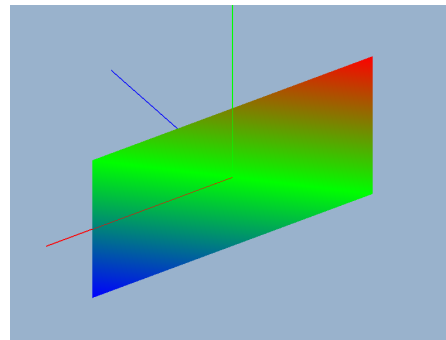
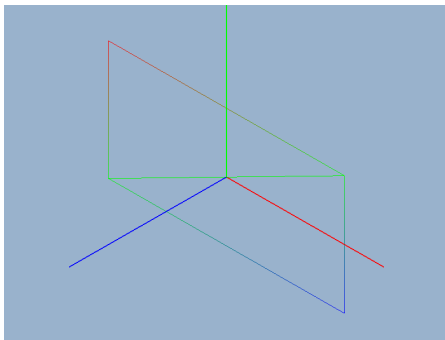
```
static Mesh* generateRectangle(GLdouble w, GLdouble h)
```

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**.

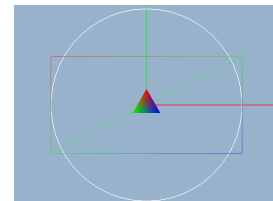


## Apartado 9 .....

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 “*demo de la escena 2*”.