

FACULTAD DE INFORMÁTICA
Curso 2019-2020
Ejercicios P1

1. **Polígono regular** (Dibujo de líneas)

Define, en la clase **Mesh**, la función **static Mesh* generaPoligono(GLuint numL, GLdouble rd)** que genera los **numL** vértices del polígono regular inscrito en la circunferencia de radio **rd** centrada en el plano $Z=0$. Utiliza la primitiva **GL_LINE_LOOP**.

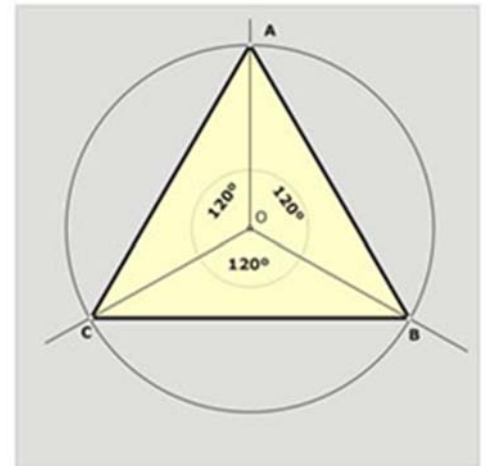
Para generar los vértices utiliza la ecuación de la circunferencia con centro **C** y radio **R**:

$$x = C_x + R \cos(\text{ang})$$
$$y = C_y + R \sin(\text{ang})$$

Para $C=(0, 0)$ y $R=\text{rd}$.

Para el triángulo de la imagen generamos tres vértices, el primero con un ángulo inicial de 90° , y los siguientes incrementando el ángulo en $360^\circ/\text{numL}$ (cuidado con la división). Recuerda pasar los grados a radianes:

```
using namespace glm;
cos(a) y sin(a) para ángulos en radianes.
Para transforma grados a radianes:
radians(degrees).
Por ejemplo: cos(radians(90))
```



Un triángulo tiene dos caras (**FRONT** y **BACK**), y para identificarlas se usa el orden de los vértices en la malla. En OpenGL los vértices de la cara exterior (**FRONT**) se dan en **orden contrario a las agujas del reloj (CCW)**. En la imagen: A, C, B.

Añade a la clase **Abs_Entity** un atributo **mColor** para el color (**dvec4**), inícialo a 1 en la constructora (**mColor(1)**), y define un método para modificarlo. Define también la destructora.

Define la clase **Poligono** heredando de **Abs_Entity**, y redefine el método **render(...)** para establecer, antes de renderizar la malla, el color y el grosor de la líneas con **glColor3d(r,g,b)** y **glLineWidth(2)**. Para acceder a las componentes del color: **mColor.r**, **mColor.g**, **mColor.b**. Después de renderizar la malla restablece, en OpenGL, los atributos a sus valores por defecto (1).

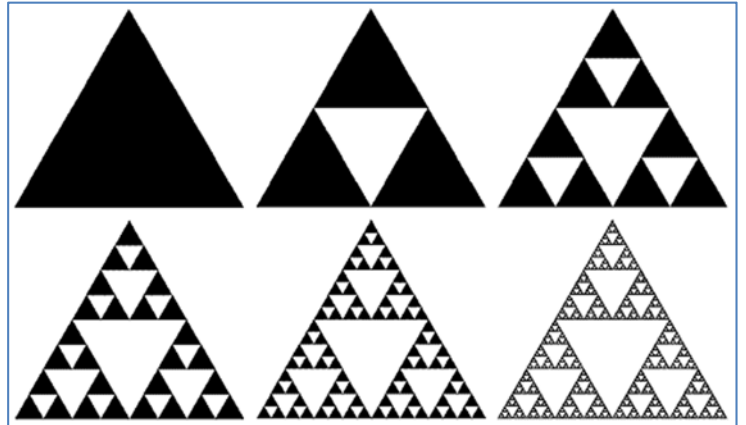
Añade a la escena (en el método **init()**) dos entidades de esta clase: un **triangulo** (amarillo) y una **circunferencia** (magenta).

Modifica el tamaño de la ventana, utiliza las teclas + y - para cambiar la escala, y las flechas para cambiar la vista.

2. Triángulo de Sierpinski (Dibujo de puntos)

El matemático polaco Waclaw Sierpinski introdujo este fractal en 1919. El fractal de Sierpinski se puede construir tomando un triángulo cualquiera, eliminando el triángulo central que se obtiene uniendo los puntos medios de cada lado, y repitiendo hasta el infinito el mismo proceso en los tres triángulos restantes.

En la figura observamos hasta cinco iteraciones sucesivas, para el caso de un triángulo equilátero.



Otra forma de construir la figura del triángulo de Sierpinski es generando puntos a partir de los tres vértices T_0 , T_1 y T_2 de un triángulo equilátero inicial. Para ello:

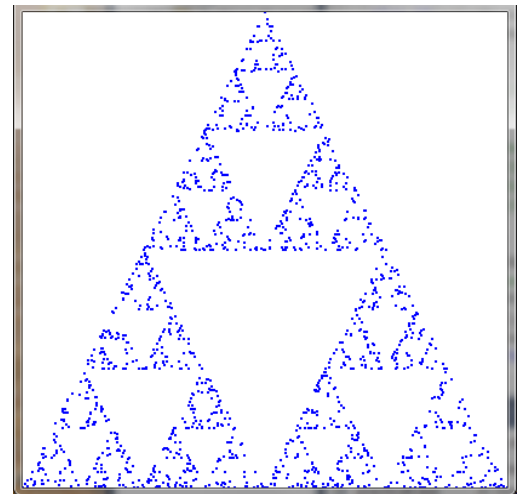
- Se parte de un punto P_0 elegido al azar de entre los tres vértices T_i .
- En cada iteración ($k > 0$) se genera otro punto P_k calculando el punto medio entre el punto anterior (P_{k-1}) y uno de los tres vértices T_i elegido aleatoriamente.

Define la función `static Mesh* generaSierpinski(GLdouble rd, GLuint numP)` que genera la malla formada por `numP` vértices del triángulo equilátero de Sierpinski inscrito en la circunferencia de `radio rd` centrada en el origen. Utiliza la primitiva `GL_POINTS`.

Los tres vértices del triángulo inicial T_0 , T_1 y T_2 , forman parte del triángulo de Sierpinski y serán los primeros vértices de la malla. Por tanto, para elegir aleatoriamente uno de ellos: `vertices[rand()%3]`

El punto medio P_m de dos puntos $A=(A_x, A_y, A_z)$ y $B=(B_x, B_y, B_z)$ es la semisuma de las coordenadas de los puntos: $P_m=((A_x + B_x) / 2, (A_y + B_y) / 2, (A_z + B_z) / 2)$

```
Mesh * generaSierpinski (...) {  
    Mesh * triangulo = generaPoligono(3, rd);  
    Mesh * mesh = new Mesh();  
    ... // crea la malla de Sierpinski  
    delete triangulo; triangulo = nullptr;  
    return mesh;  
}
```



Define la clase `Sierpinski` heredando de `Abs_Entity`, redefine el método `render(...)` para establecer el grosor de los puntos con `glPointSize(2)` y el color con `glColor4dv(value_ptr(mColor))`. Recuerda restablecer, en OpenGL, los atributos a sus valores por defecto (1) después de renderizar la malla.

Añade a la escena una entidad de esta clase de color amarillo.

3. TriánguloRGB

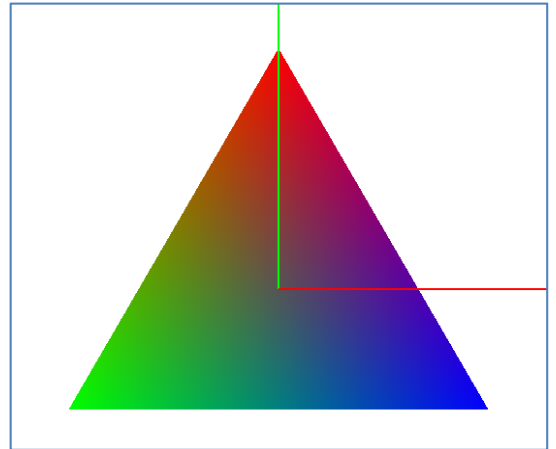
Define la función `static Mesh* generaTrianguloRGB(GLdouble rd)` que añade al triángulo un color primario en cada vértice. Utiliza la primitiva `GL_TRIANGLES`.

```
Mesh * generaTrianguloRGB(GLdouble rd) {  
    Mesh * mesh = generaPoligono(3, rd);  
    ... // añade el vector de colores  
    return mesh;  
}
```

Define la clase `TrianguloRGB` heredando de `Abs_Entity`, y añade una entidad de esta clase a la escena. Redefine el método `render(...)` para establecer que el triángulo se rellene por la cara `FRONT` y no por la cara `BACK`.

Podemos configurar el modo en que se rellenan los triángulos con el comando `glPolygonMode(...)`:

```
glPolygonMode(GL_BACK, GL_LINE)  
glPolygonMode(GL_BACK, GL_POINT)  
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL) // por defecto
```



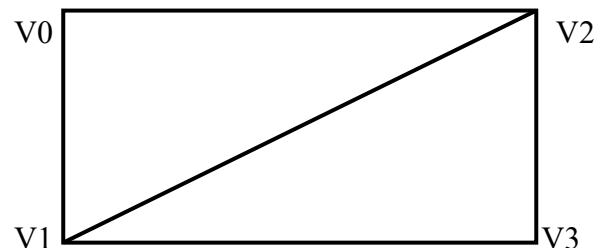
4. Rectángulo

Define la función `static Mesh* generaRectangulo(GLdouble w, GLdouble h)` que genera los cuatro vértices del rectángulo centrado en el plano $Z=0$, de ancho `w` y alto `h`. Utiliza la primitiva `GL_TRIANGLE_STRIP`.

Recuerda formar los triángulos en el orden contrario a las agujas del reloj.

En el ejemplo: `V0`, `V1`, `V2`, `V3`

Define los triángulos: `V0`, `V1`, `V2` y `V2`, `V1`, `V3`



Define la función `static Mesh* generaRectanguloRGB(GLdouble w, GLdouble h)` que añade un color a cada vértice.

Define la clase `RectanguloRGB` heredando de `Abs_Entity`, y añade una entidad de esta clase a la escena. Redefine el método `render(...)` para establecer que los triángulos se rellenen por la cara `FRONT` y no por la cara `BACK`.

Todas las mallas de los ejercicios anteriores tienen que estar en $Z = 0$ y centradas.

5. Escena 2D

Compón una escena, sobre fondo negro, con todas las entidades anteriores, utilizando las matrices de modelado para disponerlas en la escena.

Utiliza, en `Scene::init()`, las funciones de `glm` (en `gtc/matrix_transform.hpp`):

- `translate(mat, dvec3(dx, dy, dz))`: devuelve la matriz (`dmat4`) `mat*translationMatrix`, resultante de aplicar la translación (`dx, dy, dz`) a la matriz `mat` (`dmat4`).
- `scale(mat, dvec3(sx, sy, sz))`: devuelve la matriz (`dmat4`) `mat*scaleMatrix`, resultante de aplicar la escala (`sx, sy, sz`) a la matriz `mat` (`dmat4`).
- `rotate(mat, radians(ang), dvec3(eje de rotación))`: devuelve la matriz (`dmat4`) `mat*rotationMatrix`, resultante de aplicar la rotación a la matriz `mat` (`dmat4`).

Por ejemplo: `auto g = new Poligono(...); gObjects.push_back(g); g->setColor(...);`
`g->setModelMat(translate(g->modelMat(), ...));`
`g->setModelMat(rotate(g->modelMat(), ...));`

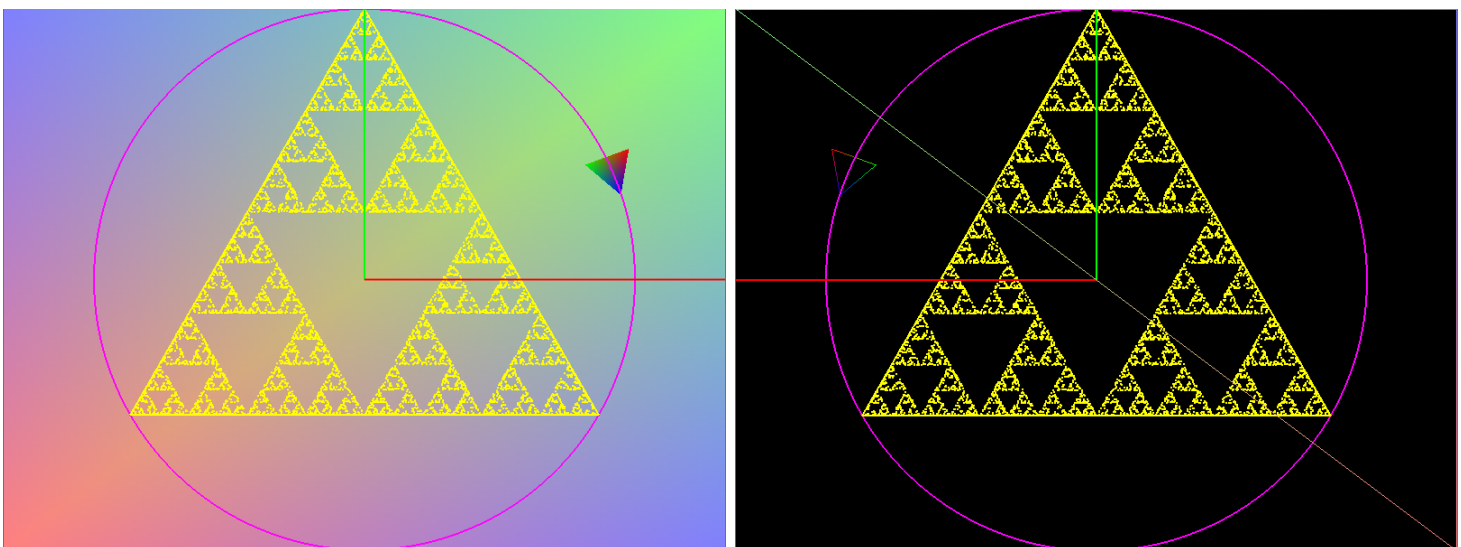
Aplica al rectángulo una traslación en el eje Z de -100.

Aplica al triángulo RGB un giro en Z (prueba con 25° y -25°) y una translación en X e Y (prueba a cambiar el orden de las transformaciones).

Animación:

Añade a la clase `Scene` un método `void update()` que indique a las entidades que se actualicen. Añade a la clase `Abs_Entity` el método `update()` con implementación vacía para que las subclases puedan redefinirlo. Define la tecla `u` para indicar a la escena que se actualice.

Añade a la clase `TrianguloRGB` atributos para generar un desplazamiento del objeto describiendo una circunferencia a la vez que gira sobre su centro: los ángulos de giro. Redefine el método `update` para actualizar los ángulos y la matriz `mModelMat`. La matriz se obtiene a partir de la matriz identidad, aplicando las transformaciones de `glm` con los ángulos.



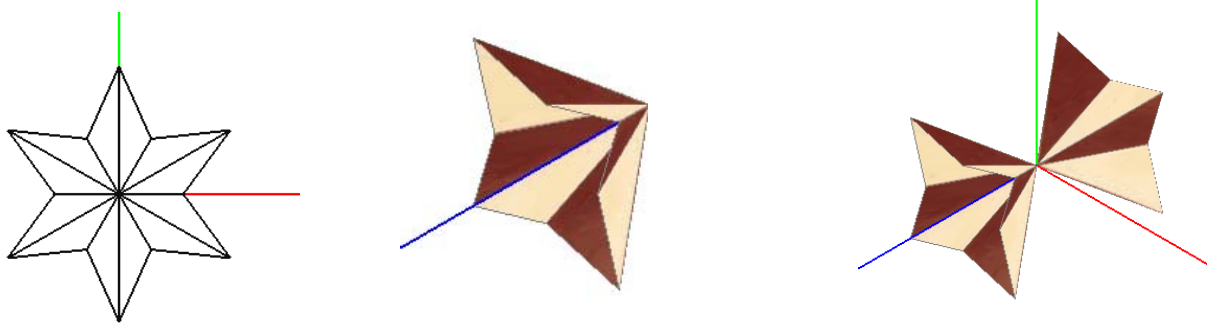
6. Estrella 3D

Define la función `static Mesh* generaEstrella3D(GLdouble re, GLdouble np, GLdouble h)` que genera los vértices de una estrella de `np` puntas, centrada en el plano $Z=h$. Utiliza la primitiva `GL_TRIANGLE_FAN` con primer vértice $V0 = (0, 0, 0)$.

El número de vértices es $2*np + 2$.

Utiliza la ecuación de la circunferencia para generar los vértices. Puedes añadir un parámetro `ri` para el radio interior o utilizar `re/2`.

Recuerda generar los vértices en orden contrario a las agujas del reloj.



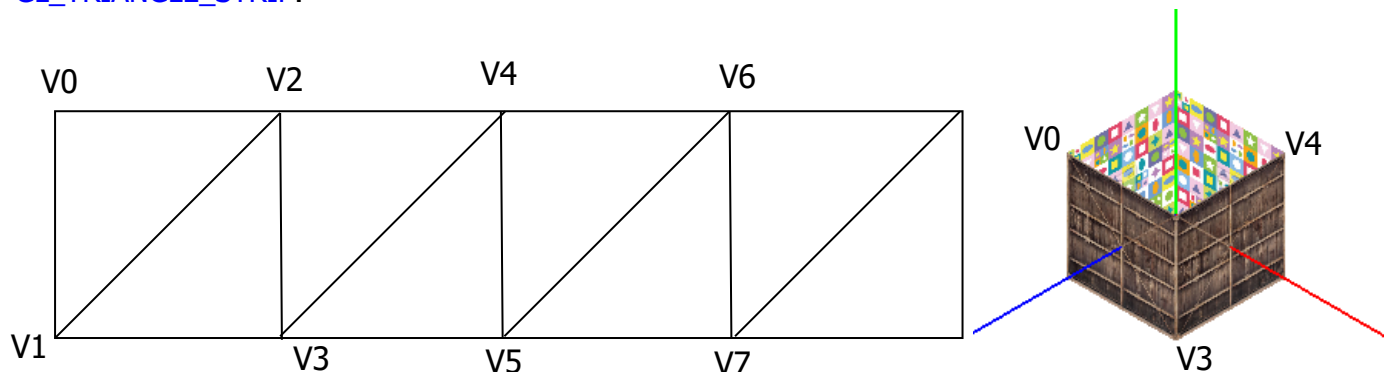
Define la clase `Estrella3D` heredando de `Abs_Entity`, y añade una entidad de esta clase a la escena (renderiza en modo líneas).

Modifica el método `render` de la clase `Estrella3D` para dibujar dos veces la estrella según aparece en la imagen. La misma malla se renderiza dos veces con distinta matriz de modelado.

Animación: Añade a la clase `Estrella3D` atributos para que gire sobre su eje Z y sobre su eje Y (los ángulos de giro). Redefine el método `update` para actualizar los ángulos y la matriz `mModelMat`.

7. Caja

Define la función `static Mesh* generaContCubo(GLdouble ld)` que genera los vértices del contorno de un cubo, centrado en los tres ejes, de lado `ld`. Utiliza la primitiva `GL_TRIANGLE_STRIP`.



El número de vértices es 10: 8 del cubo ($V0, \dots, V7$) + 2 para cerrar el contorno ($V0, V1$).

Define la clase `Caja` heredando de `Abs_Entity`, y añade una entidad de esta clase a la escena (renderiza en modo líneas).

8. Cambio de escena

Para cambiar entre escenas, añade a la clase `Scene` un atributo para el identificador de escena `int mId = 0` y un método para cambiarlo `void setState(int id)` (si es necesario, reinicia). Con la tecla `1` cambiamos a la escena `1` (escena 3D) y con la tecla `0` a la `0` (escena 2D). Modifica el método `init` para que, en función de `mId` inicie una escena u otra. Tendrás que adaptar otros métodos (`resetGL`, `initGL`, ...).

9. Animación (Opcional)

Añade, a la clase `IG1App`, el método `update()` (sin argumentos) y la función estática `s_update()` (que invoca al método `update()`) para el callback de `glutIdleFunc`. Esta función será llamada cuando la aplicación esté desocupada y la utilizamos para actualizar los valores de animación. El método `update()` debe indicar a la escena que se actualice cada cierto tiempo (no más de 60 veces por segundo). Para esto, añade una variable (`GLuint mLastUpdateTime`) para capturar el último instante en que se realizó una actualización y utiliza `glutGet(GLUT_ELAPSED_TIME)` (devuelve los milisegundos transcurridos desde que se inició) para actualizar la variable y controlar el tiempo que debe transcurrir entre actualizaciones. Añade también una variable `bool` para activar/desactivar la animación con la tecla `U`.