FACULTAD DE INFORMÁTICA
Curso 2020-2021
Práctica 1

Entrega 1 (Ejercicios 1-7). Todas las mallas de los ejercicios de la escena 2D tienen que estar en Z = 0 y centradas. La escena de esta entrega se renderiza con la tecla 0.

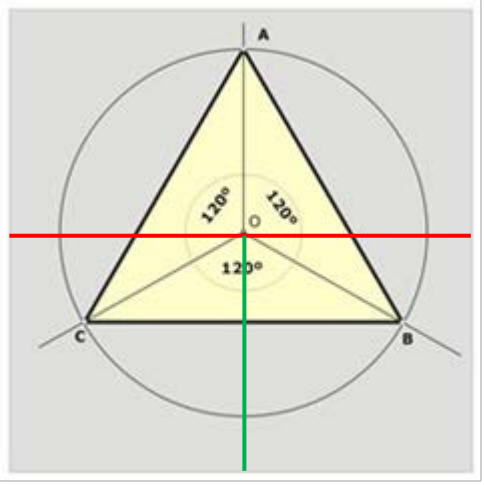
1. Polígono regular (Dibujo de líneas)

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:

$$x = Cx + R\cos(ang)$$
$$y = Cy + R\sin(ang)$$

Para C=(0, 0) y R=rd.



Para rd=3, por ejemplo, el triángulo de la imagen, se generan 3 vértices, el primero (A) con un ángulo inicial de 90º, y los siguientes (C y B) incrementando el ángulo en 360º/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.

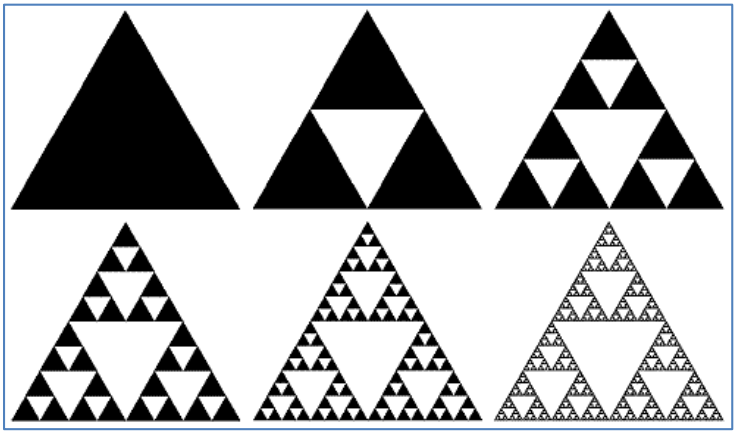
Añade a la clase Abs_Entity un atributo mColor (dvec4) para el color, inícialo a blanco en la constructora (mColor(1)), y define un método (setColor) para modificarlo. Si quieres define también la destructora.

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 las líneas con glColor3d(r,g,b) y glLineWidth(2). Para acceder a las components del color: mColor.r, mColor.g, mColor.b. Después de renderizar la malla restablece, en OpenGL, los atributos a sus valores por defecto (blanco y 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. Triangulo de Sierpinski (Dibujo de puntos)



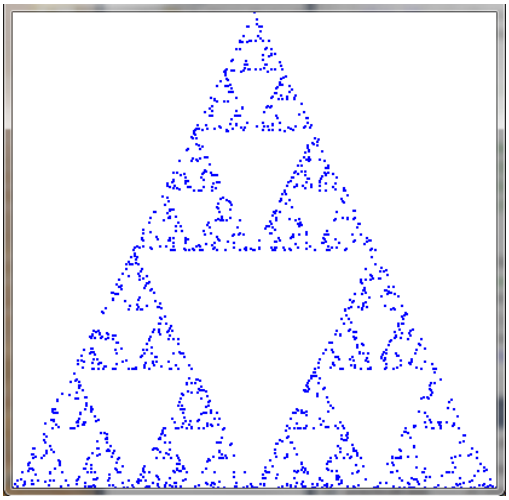
El matemático polaco Waclaw Sierpinski introdujo este fractal en 1919. El fractal de Sierpinski se puede construir tomando un triángulo cualquiera, eliminando el triángulo central que se obtiene uniendo los puntos medios de cada lado, y repitiendo hasta el infinito el mismo proceso en los tres triángulos 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 T0, T1 y T2 de un triángulo equilátero inicial. Para ello:

- Se parte de un punto P0 elegido al azar de entre los tres vértices Ti.
- En cada iteración (k>0) se genera otro punto Pk calculando el punto medio entre el punto anterior (Pk-1) y uno de los tres vértices iniciales Ti 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 T0, T1 y T2, 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 Pm de dos puntos A=(Ax, Ay, Az) y B=(Bx, By, Bz) es la semisuma de las coordenadas de los puntos: Pm = (A + B) / 2.0

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

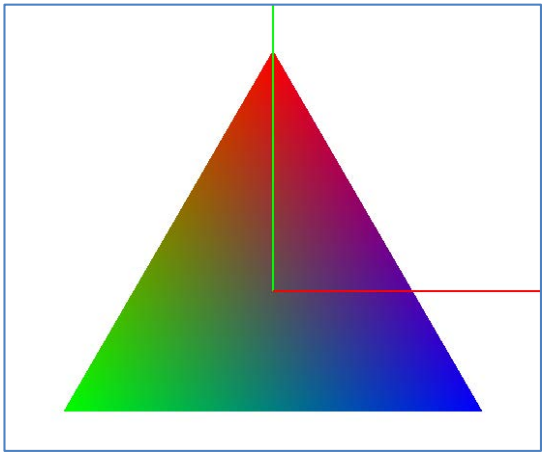
Define la clase Sierpinski heredando de Abs_Entity, redefine el método render(...) para establecer el grosor de los puntos con glPointSize(2) y el color con glColor4dv(value_ptr(mColor)). Recuerda restablecer, en OpenGL, los atributos a sus valores por defecto después de renderizar la malla.

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

3. TrianguloRGB

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
```

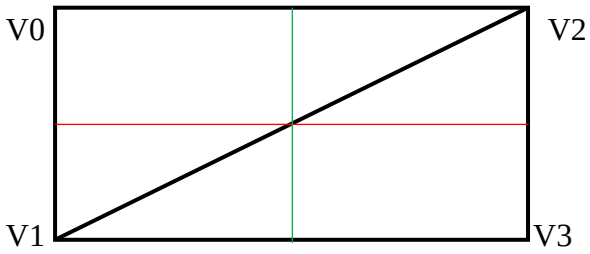
4. Rectángulo

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 por la cara BACK en modo líneas.

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(dmat4(1), ...);
g->setModelMat(rotate(g->modelMat(), ...);

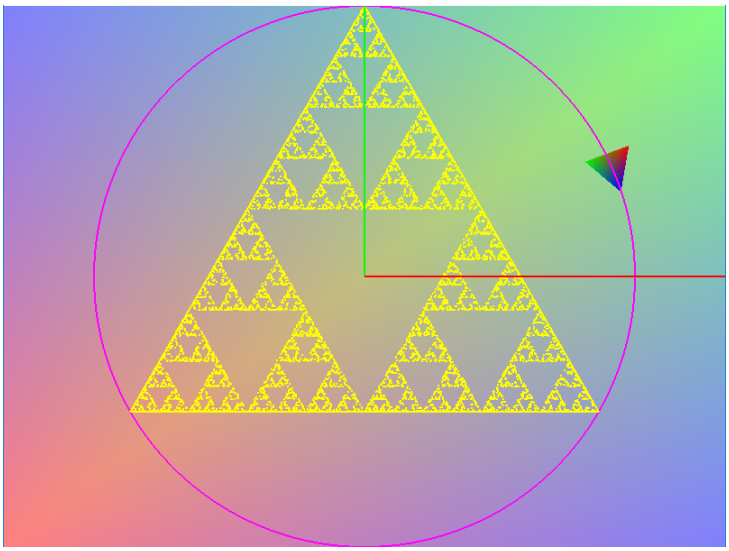
Aplica al rectángulo una traslación en el eje Z de -100.

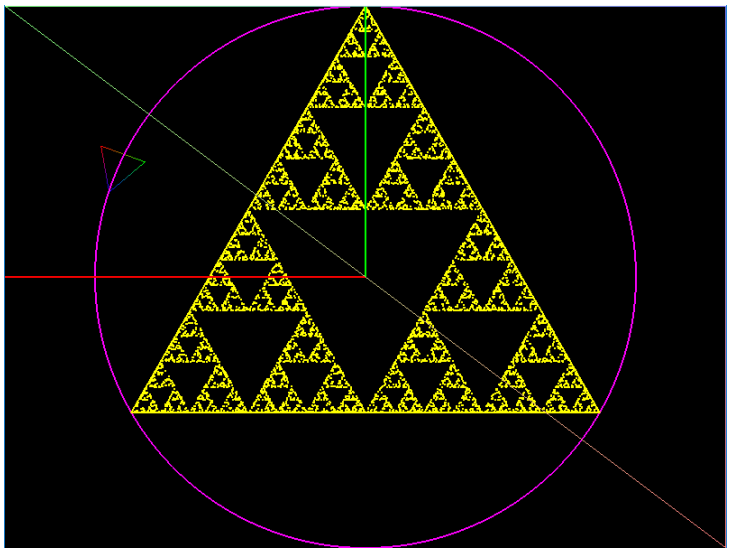
Aplica al triángulo RGB un giro en Z (prueba con 25º y -25º) y una translación en X e Y (prueba a cambiar el orden de las transformaciones).

Animación:

Añade a la clase Abs_Entity el método update() con implementación vacía para que las subclases puedan redefinirlo: virtual void update() {}. Añade a la clase Scene un método void update() que indique a las entidades que se actualicen. En IG1App define la tecla u para indicar a la escena que se actualice.

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





6. 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 changeScene(int id) que elimina la escena actual y genera la nueva. La escena 2D es la escena 0, y con la tecla 1 cambiamos a la escena 1 (la primera escena 3D) que de momento contiene los ejes RGB. Modifica el método init para que, en función de mId inicie una escena u otra. Puede que más adelante tengas que adaptar otros métodos, como free, resetGL, initGL.

En IG1App define las teclas 0 y 1 para indicar a la escena que cambie a la escena 1/0.

7. Animación (opcional) (Entrega: 11 de marzo)

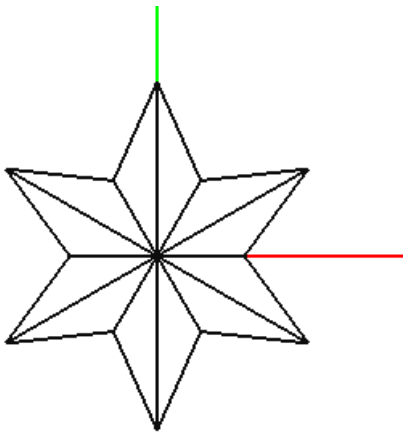
Añade, a la clase IG1App, el método update() (sin argumentos) y la función estática s_update() (que invoca al método update()) para el callback de glutIdleFunc. Está función será llamada cuando la aplicación esté desocupada y la utilizamos para actualizar los valores de animación. El método update() 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.

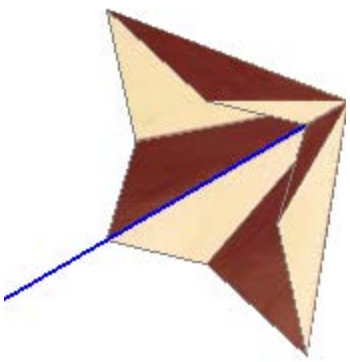
Entrega 2 (Ejercicios 8-20). La escena de esta entrega se renderiza con la tecla 1.

8. Estrella 3D

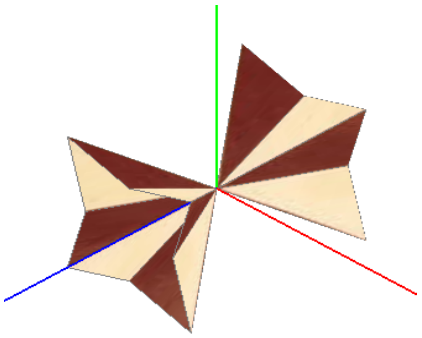
Define la función static Mesh* generaEstrella3D(GLdouble re, GLuint np, GLdouble h) que genera los vértices de una estrella de np puntas, centrada en el plano Z=h. Utiliza la primitiva GL_TRIANGLE_FAN con primer vértice V0 = (0, 0, 0). El número de vértices es 2*np + 2.

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 (CCW).





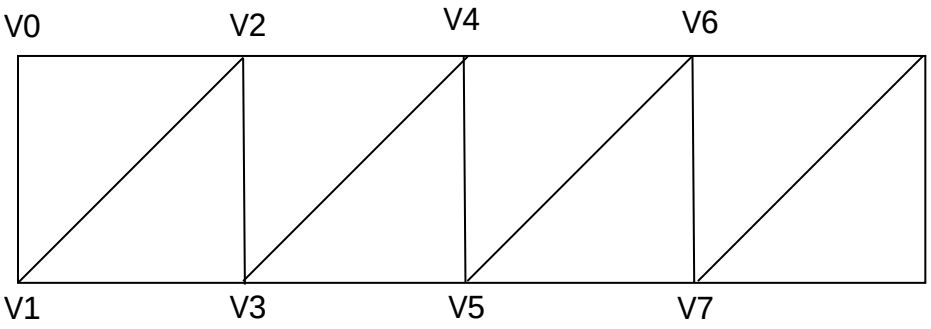
Define la clase Estrella3D heredando de Abs_Entity, y añade una entidad de esta clase a la escena 1 (renderiza en modo líneas). Modifica el método render de la clase Estrella3D para dibujar dos veces la estrella según aparece en la imagen. La misma malla se renderiza dos veces con distinta matriz de modelado.

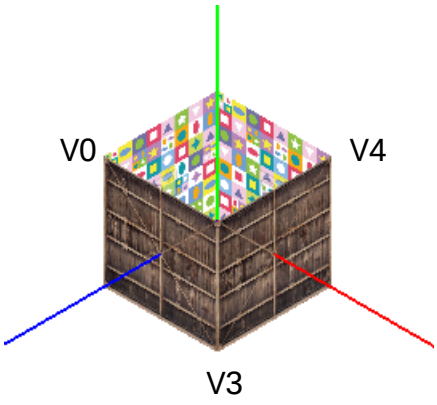


Animación: Añade a la clase Estrella3D atributos (los ángulos de giro) para que gire sobre su eje Z y sobre su eje Y. Redefine el método update para actualizar los ángulos y la matriz mModelMat. La matriz se genera a partir de la matriz identidad, aplicando las transformaciones de glm con los ángulos.

9. 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, ..., V7) más 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).

10. Caja con fondo (opcional)

Define la clase CajaConFondo para renderizar cajas que disponen de un rectángulo para el fondo. Para ello añade otro atributo de tipo Mesh* para la malla del fondo y otro atributo de tipo matriz para colocarla en el fondo. Renderiza también en modo líneas.

11. Texturas

Descarga del campus BmpsP1 (archivos de imágenes .bmp) y Texturas.zip (clases para el manejo de texturas). Copia los archivos .bmp en el directorio Bmps del proyecto y los archivos de código (.h y .cpp) en el directorio IG1App. Y añade al proyecto estos nuevos archivos de código (Explorador de soluciones -> Agregar -> Elemento existente).

Modifica la clase Mesh para incorporar el vector de coordenadas de textura, que es un nuevo atributo (vector<dvec2> vTexCoords), y adapta el método render.

En la clase Scene, añade un atributo para las texturas de los objetos (vector<Texture*> gTextures). En el método init, crea y carga las texturas (con el método load de Texture) para los objetos de la escena. Adapta los métodos free, setGL (activa las texturas en OpenGL), y resetGL (las desactiva).

Añade a las entidades un atributo Texture* mTexture = nullptr y un método setTexture(Texture* tex) { mTexture = tex; }. En el método render de las entidades, activa (método bind de Texture) la textura antes de renderizar la malla, y desactívala después (unbind).

12. Suelo con textura

Añade la clase Suelo: Entidad que renderiza un rectángulo centrado en el plano Y=0, embaldosado con una textura que se repite. En la constructora, utiliza la malla generaRectangulo() y establece la matriz de modelado para posicionarla horizontal.

Define la función static Mesh* generaRectanguloTexCor(GLdouble w, GLdouble h, GLuint rw, GLuint rh) que añade coordenadas de textura para cubrir el rectángulo con una imagen que se repite rw veces a lo ancho y rh veces a lo alto.

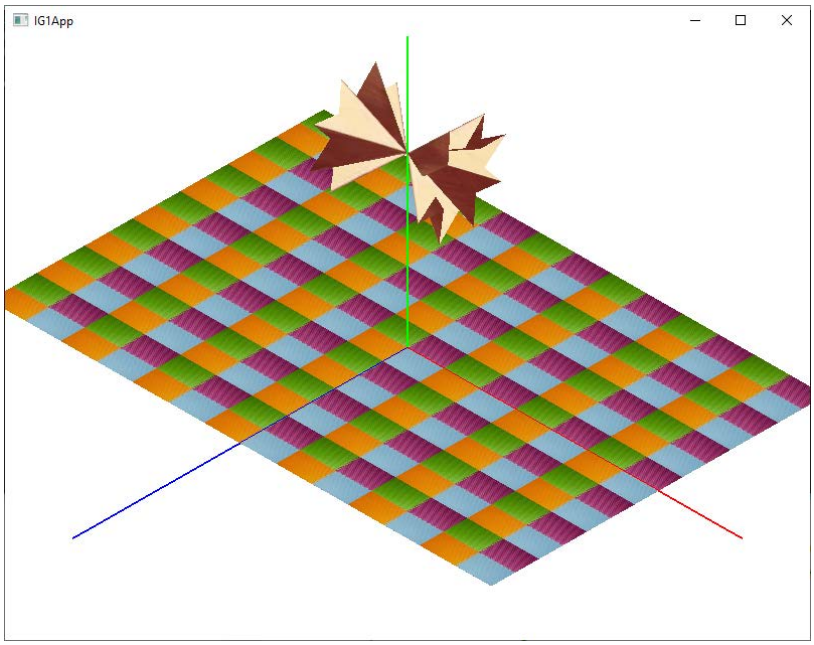
Modifica la constructora de Suelo y el método render para renderizar el suelo con textura.

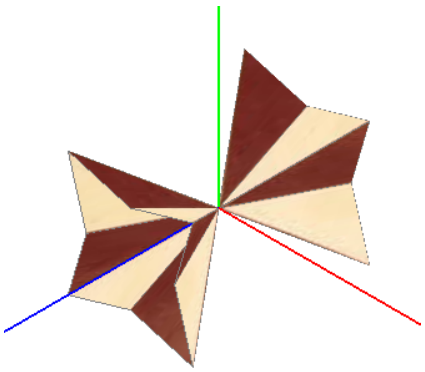
Ajusta el número de repeticiones de la textura según las dimensiones del rectángulo y de la textura, de forma que la imagen guarde sus proporciones y no pierda calidad.

Establece un color para modularlo con la textura.

13. Estrella con textura

Define la función static Mesh* generaEstrellaTexCor(GLdouble re, GLuint np, GLdouble h) que añade coordenadas de textura a la estrella, centrando la imagen en su vértice (0,0,0).





Modifica la constructora de Estrella3D y el método render para que se renderice la estrella con textura. Utiliza la textura de las figuras adjuntas donde aparece una estrella con un número de puntas que es potencia de 2, arriba con 8 y a la izquierda con 4.

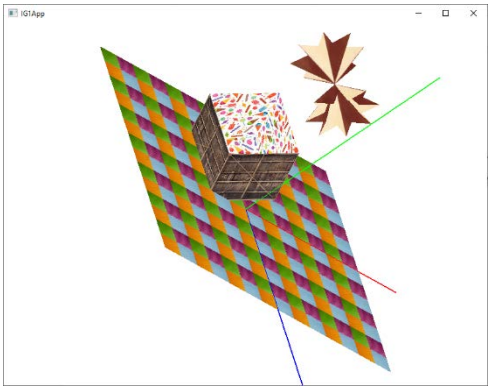
14. Caja con textura

Define la función static Mesh* generaContCuboTexCor(GLdouble nl) que añade coordenadas de textura a la caja de más arriba, repitiendo la imagen en cada cara de la caja.

Modifica el método render para renderizar la caja con dos texturas (añade otro atributo Texture*), una para el exterior de la caja (container.bmp) y otra para el interior (chuches.bmp). Para renderizar solo el exterior o el interior, utiliza los comandos:

```
glEnable(GL_CULL_FACE)
glCullFace(GL_FRONT / GL_BACK)
glDisable(GL_CULL_FACE)
```

15. Caja con fondo y con textura (opcional)



Añade texturas a la clase CajaConFondo de manera que el interior del fondo de la caja tenga la textura del interior de la caja, mientras que el exterior tenga la exterior.

Modifica el método update para que haga rotar la caja alrededor del eje Z.

16. Foto

Define una nueva entidad que renderiza un rectángulo sobre el suelo con una textura cuya imagen es la visualización de la escena en el renderizado anterior (front buffer). Puedes acceder a las dimensiones de la ventana añadiendo métodos de consulta a IG1App, por ejemplo: IG1App::s_ig1app.winWidth();

Añade a la clase Texture el método loadColorBuffer(width, height, buffer) con los comandos de OpenGL necesarios para copiar el color buffer en la textura: glCopyTexImage2D, glReadBuffer

Define el método update para que se actualice la textura.

17. Guardar en archivo bmp (opcional)

Define la tecla F para guardar una imagen resultante del renderizado en un archivo bmp.

Añade a la clase Texture el método save(const std::string & BMP_Name). Para implementarlo utiliza: una variable local de la clase PixMap32RGBA; el comando glGetTexImage(...) para obtener los datos de la textura en la variable (la variable tiene que tener memoria para la textura); y el método save_bmp24BGR() de la clase PixMap32RGBA para guardar la imagen en el archivo.

18. Planta (opcional)

Utiliza la imagen grass.bmp, con el color de fondo transparente, como textura de al menos tres rectángulos que se cruzan en el eje Y, y que aparecen sobre el suelo en una esquina. Define, en la clase Texture, el método

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

para que una vez cargada la imagen, los texels del color dado tomen el valor alpha dado y los demás texels queden opacos.

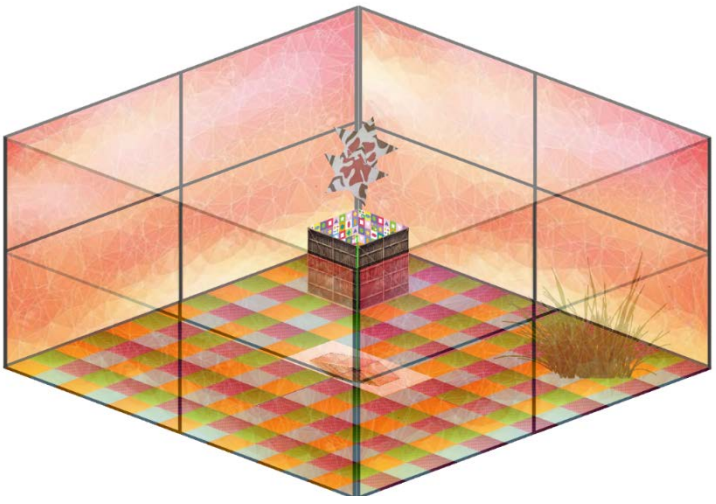
Crea la clase Planta cuyos objetos consisten en tres plantas cruzadas en el eje Y de forma perpendicular.

19. Cristalera traslúcida (blending)

Define el contorno de un cubo, es decir, una caja, que aparezca alrededor del suelo y que se renderice con la textura cristalTri.bmp en todas sus caras, con todos los colores translúcidos.

20. Escena 3D

Está formada por un suelo (que es un rectángulo en el plano Y=0, centrado en el origen), una caja sobre el suelo situada en el cuadrante <–X, -Z>, y una estrella por encima de la caja. Además, una foto que se renderiza en el plano XZ, centrada en el origen. Por último, una cristalera traslúcida rodeando el suelo. Recuerda que la estrella y el cubo deben rotar tal como se pidió en sus correspondientes apartados y que la foto debe actualizarse, todo ello al invocar el método update con la tecla u.



Entrega 3 (Ejercicios 21-). La escena de esta entrega se renderiza con la tecla 2.

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

```
void moveLR(GLdouble cs); // Left / Right
void moveFB(GLdouble cs); // Forward / Backward
void moveUD(GLdouble cs); // Up / Down
```

Añade los atributos mRight, mUpward y mFront, para cada uno de los ejes, más el método protegido void setAxes() que da valor a estos tres ejes. Tendrás que incluir <gtc/matrix_access.hpp> para poder usar la función row().

Todos los métodos que modifiquen algún elemento de la cámara tienen que actualizar todos los atributos necesarios para que sus valores sean coherentes. Por ello, modifica el método:

```
void setVM() { mViewMat = lookAt(mEye, mLook, mUp);  setAxes(); };
```

Quita (comenta) los métodos pitch, yaw y roll de la clase Camera.

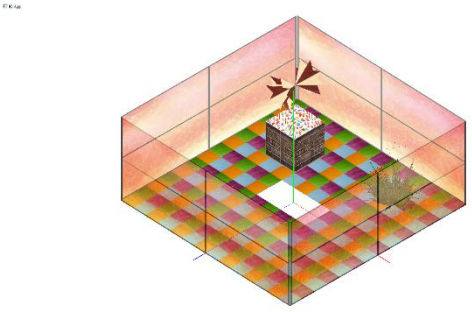
22. Añade a la clase Camera el método orbit para desplazar la cámara (mEye) siguiendo una circunferencia (en el plano determinado por los ejes mRight y mFront) a una determinada altura (eje mUpward) alrededor de mLook, modificando la posición y la dirección de vista de la cámara.

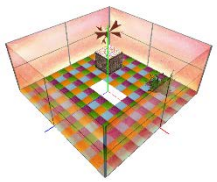
```
void orbit(GLdouble incAng, GLdouble incY)
```

Añade a la clase Camera los atributos para el ángulo y el radio (p. ej. 1000) de la circunferencia que recorrerá la cámara con este método. Tendrás que dar un valor inicial adecuado al ángulo en el método set3D.

23. Añade a la clase Camera un método changePrj() para cambiar de proyección ortogonal a perspectiva. Para establecer el cambio usa el booleano bOrto de la clase Camera. Modifica también aquellos métodos de la clase Camera afectados por el cambio de proyección (por ejemplo, para que el zoom siga funcionando correctamente con ambas proyecciones es necesario modificar setPM()).

Define la tecla p para cambiar entre proyección ortogonal y perspectiva.





24. Añade a la aplicación la opción (tecla k) de visualizar dos vistas simultáneamente. Añade a la aplicación un atributo bool m2Vistas para la opción, y utilízalo en el método display de la aplicación. Define el método display2Vistas() para dividir la ventana en dos puertos de vista y visualizar, en el lado izquierdo, la vista actual, y en el lado derecho, la vista cenital (utiliza una cámara auxiliar, la escala debe ser la misma). Puedes añadir a la clase Camera un método setCenital. Aquí se muestran capturas de lo que realmente se renderiza según que la proyección sea ortogonal o perspectiva.