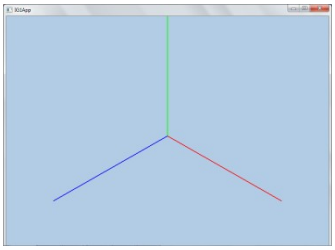
Facultad de Informática
Curso 20-21. Práctica 2

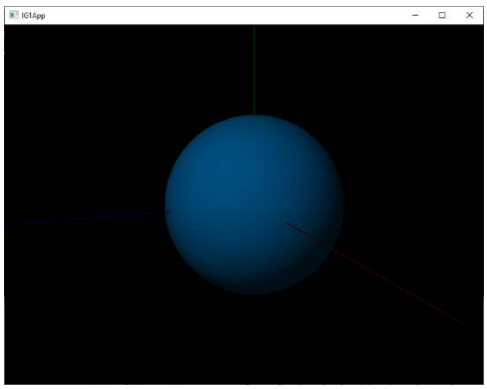


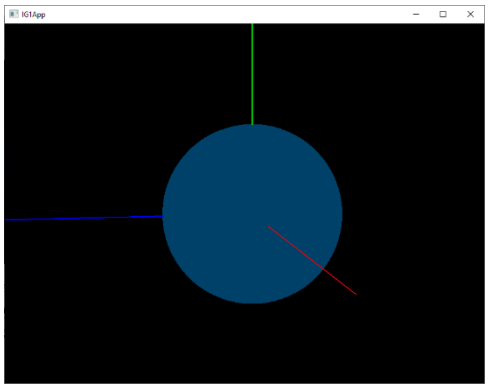
1. Prepara tu proyecto creando una escena vacía que tiene, de momento, solo los ejes coordenados y cuyo color de fondo es (0.7, 0.8, 0.9) (o negro), tal como se muestra en la figura adjunta.

2. Vamos a introducir una luz en la escena. El significado de los comandos que se utilizan a continuación se explicará más adelante. De momento limítate a añadir el siguiente método a la clase **`Scene`**:

```
void Scene::sceneDirLight(Camera const&cam) const {
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glm::fvec4 posDir = { 1, 1, 1, 0 };
    glMatrixMode(GL_MODELVIEW);
    glLoadMatrixd(value_ptr(cam.viewMat()));
    glLightfv(GL_LIGHT0, GL_POSITION, value_ptr(posDir));
    glm::fvec4 ambient = { 0, 0, 0, 1 };
    glm::fvec4 diffuse = { 1, 1, 1, 1 };
    glm::fvec4 specular = { 0.5, 0.5, 0.5, 1 };
    glLightfv(GL_LIGHT0, GL_AMBIENT, value_ptr(ambient));
    glLightfv(GL_LIGHT0, GL_DIFFUSE, value_ptr(diffuse));
    glLightfv(GL_LIGHT0, GL_SPECULAR, value_ptr(specular));
}
```

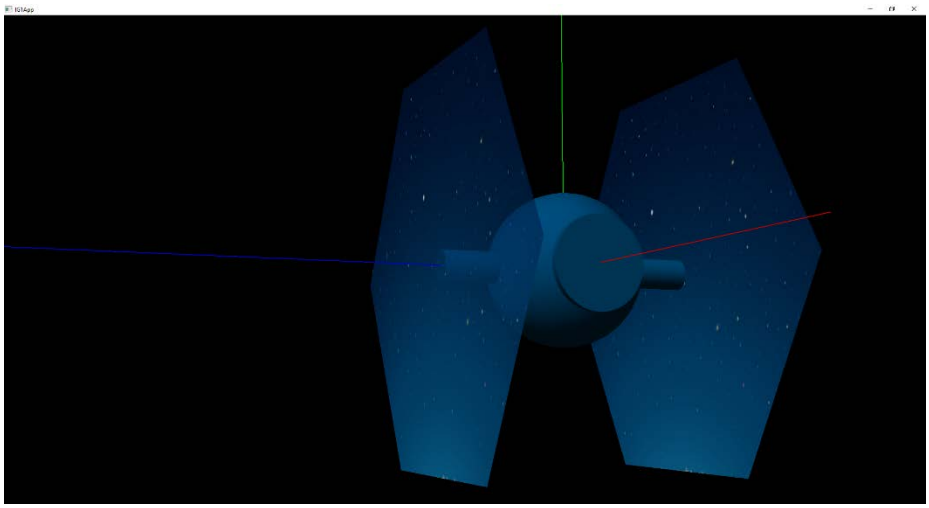
Este método debe ser llamado (salvo que se diga lo contrario) al principio del método **`render(cam)`** de **`Scene`** para que la luz tenga efecto antes de renderizar los objetos de la escena. Esta luz no sería necesaria para que se viera la escena, pero mientras con ella, una esfera (entidad cuádrica) de color añil (usando **`color material`**) se renderiza como abajo a la izquierda (con sensación de volumen, como se puede ver), sin ella se renderiza como una mancha añil redonda, tal como se ve a la derecha:





3. Añade a tu proyecto la clase **`QuadricEntity`**, que hereda de **`Abs_Entity`**, y las clases **`Sphere`**, **`Cylinder`**, **`Disk`** y **`PartialDisk`**, que heredan de **`QuadricEntity`**, y que permiten dibujar esferas, cilindros, discos y discos parciales como entidades cuádricas de la biblioteca GLU.

4. Crea una nueva escena con un caza estelar imperial TIE. Está compuesto por dos alas hexagonales **`wingL`** y **`wingR`** con textura traslúcida **`noche.bmp`**; un cilindro **`shaft`** que va de un ala a la otra; una esfera **`core`** en medio que tiene delante un **`front`** formado por un corto y ancho cilindro con un disco que lo tapa. Todos estos elementos son de color añil (0, 65, 106). Las alas hexagonales las puedes hacer con la malla que genera polígonos o como discos cuádricos de base hexagonal. **Esta será la escena 1 de la Práctica 2**.

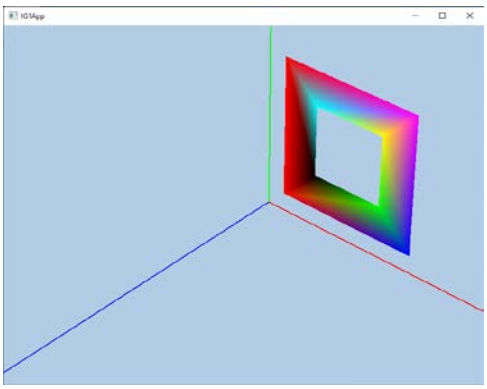


5. Define la clase **`IndexMesh`** que hereda de **`Mesh`**, añadiendo el array **`vIndices`**, modificando el método **`render()`**, tal como se explica en las transparencias, y usando el comando **`glDrawElements(...)`** en **`draw()`**.

6. Añade a la clase **`IndexMesh`** el método:

`static IndexMesh* generaAnilloCuadradoIndexado();`

que construye la malla del anillo cuadrado que se presenta en las transparencias, usando índices, esto es, la malla tiene 8 vértices, 8 colores y 10 índices. Define la clase **`AnilloCuadrado`** que hereda de **`Abs_Entity`** y que da valor al atributo **`Mesh* mMesh`** usando el método anterior. Crea una nueva escena que contenga los ejes coordenados y un objeto de esta clase. Al renderizar esta escena se verá la imagen de abajo. No uses aquí **`sceneDirLight(cam)`** ni actives **`color material`**.



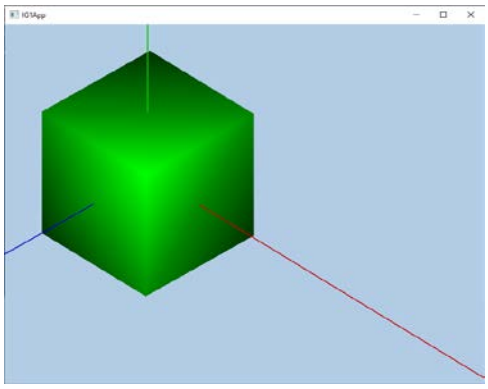
7. Añade el vector de vectores normales **`vNormals`** a la clase **`Mesh`** y modifica el método **`render()`** de esa clase tal como se explica en las transparencias. Haz lo mismo con este método en la clase **`IndexMesh`**.

8. Añade la información de los vectores normales a la malla de la clase **`AnilloCuadrado`**. Observa que todos los vectores normales de esta malla son iguales y perpendiculares al plano **`XY`**. Renderiza el anillo cuadrado indexado, pero ahora con el vector de normales activado e invocando, ahora sí, **`sceneDirLight(cam)`**, con **`color material`** activo. Se tiene que mostrar la misma imagen que en el apartado 6. **La renderización de un anillo cuadrado indexado y con normales será la escena 2 de la Práctica 2**.

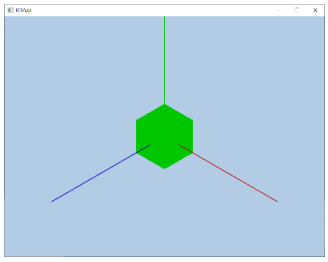
9. En la clase **`IndexMesh`** define el método:

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

que construye la malla indexada de un cubo centrado en el origen de arista de longitud **`l`**, con tapa superior e inferior. La primitiva de esta malla es **`GL_TRIANGLES`**. Hay 8 vértices y el color de todos ellos será verde. 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 **`(±1, 0, 0), (0, ±1, 0), (0, 0, ±1)`**. Por ejemplo, la esquina del cubo en el octante positivo del espacio es el vértice de coordenadas **`dvec3(l/2, l/2, l/2)`**, que corresponde al tercer vértice, es decir, al de la componente 2 del array de vértices. Su vector normal es **`glm::normalize(dvec3(1, 1, 2))`**. La captura adjunta es la renderización de uno de estos cubos, con la luz **`sceneDirLight(cam)`** dada y el **`color material`** activado.



10. Crea la clase **`Cubo`** que hereda de **`Abs_Entity`** y que da valor al atributo **`mMesh`** usando el método del apartado anterior de generación de la malla de un cubo indexado y con normales. Para comprobar la importancia de los vectores normales, renderiza el cubo sin tener en cuenta **`vNormals`** en **`render()`** y tendrás que obtener la mancha verde de la figura adjunta.



11. 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, tal como se ha explicado. Evidentemente, cuando utilices este método para obtener los vectores normales, la renderización del cubo debe ser la misma que la del correspondiente apartado anterior.

12. Define una escena que contenga unos ejes ordenados y un objeto de la clase **`Cubo`** como el descrito más arriba. **La renderización de uno de estos cubos verdes a partir de una malla indexada y con normales calculadas por el método del apartado anterior será la escena 3 de la Práctica 2**.

13. Crea la clase **`CompoundEntity`** que hereda de la clase **`Abs_Entity`** y que dispone de un atributo nuevo:

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

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

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

14. Define, en esta clase, la destructora **`~CompoundEntity()`** y reescribe el método **`render()`**.

15. Define el caza estelar imperial como una entidad compuesta. Es decir, define la clase **`TIE`** que hereda de **`CompoundEntity`** de forma que un objeto de esta clase está compuesto por dos alas (que son, según prefieras, dos discos cuádricos, o dos entidades con un hexágono regular como malla, cada una), más un núcleo central (que es una esfera cuádrica), un eje (que es el cilindro cuádrico que va de ala a ala), y un morro (que es una entidad compuesta formada por un cilindro y un disco cuádricos dispuestos tal como se explicó más arriba). Observa que el elemento **`i`**-ésimo de un **`TIE`** construido como una **`CompoundEntity`** no tiene nombre particular y nos podemos referir a él, cuando sea necesario, como **`gObjects.at(i)`**. En estas condiciones, la escena 1 pasa a ser entonces una entidad compuesta de la clase **`TIE`** que se ha de ver como el caza imperial se veía antes.

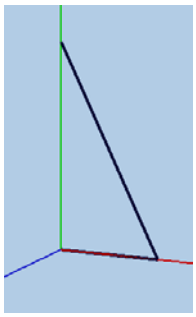
16. Define la clase de las mallas por revolución **`MbR`** que hereda de **`IndexMesh`** y que dispone de tres atributos: **`int n`**, para el número de muestras que se toman al girar el perfil alrededor del eje **`Y`**; **`dvec3* perfil`**, para el array de vértices que define el perfil que va a hacerse girar alrededor del eje **`Y`**, e **`int m`**, para el número de puntos del perfil. Define una constructora con tres argumentos para esta clase, que da valor a los atributos de la forma obvia.

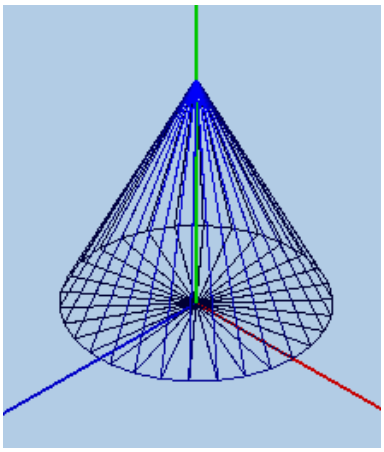
17. En la clase **MbR**, define el método estático:

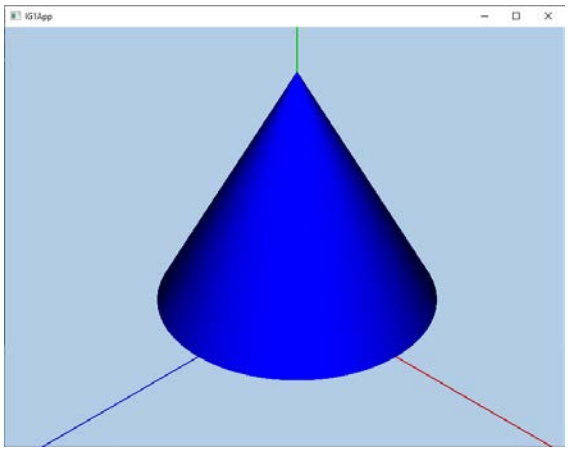
```
static MbR* generaIndexMeshByRevolution(int mm,
                                        int nn, glm::dvec3* perfil)
```

que construye, tal como se ha explicado, la malla indexada por revolución que se obtiene al hacer girar **perfil**, que tiene **mm** puntos, alrededor del eje **Y**, tomando **nn** muestras.

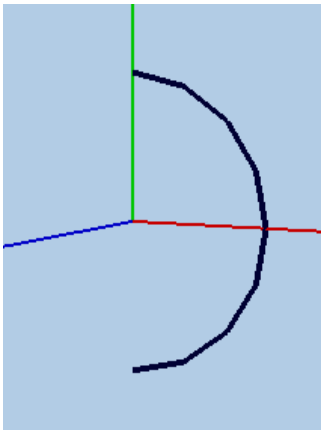
18. Define la clase **Cono** que hereda de **Abs_Entity** y cuya constructora es de la forma **Cono(h, r, n)**, donde **h** es la altura del cono, **r** es el radio de la base, y **n** es el número de muestras que se toman. La malla de esta clase está construida por revolución. El perfil tiene únicamente tres puntos en forma de 7 invertido, como se ve en la captura de la derecha y se ha explicado en clase. La malla por revolución que se obtiene, dibujada en modo línea y en modo relleno, es la que se ve en las capturas de abajo.

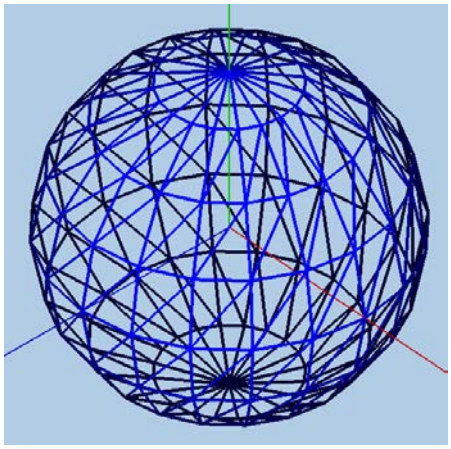




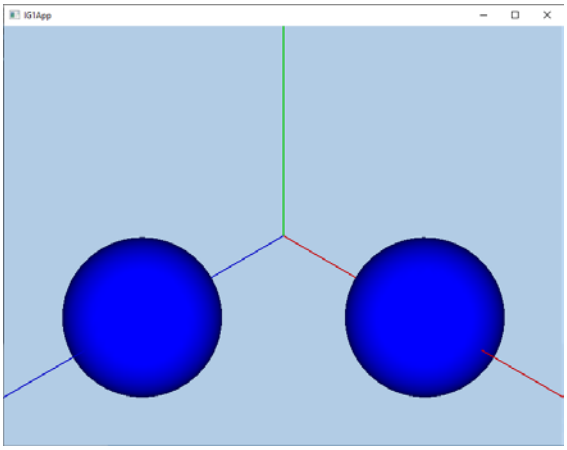


19. Define la clase **Esfera** que hereda de **Abs_Entity** y cuya constructora es de la forma **Esfera(r, p, m)**, donde **r** es el radio de la esfera y **m** -inicial de la palabra meridiano- es el número de muestras que se toman. La malla de esta clase está construida por revolución. El perfil es como el que se muestra en la captura de abajo a la izquierda y tiene **p** – inicial de la palabra paralelo- puntos. La esfera que se obtiene, en modo línea, es la que se muestra a la derecha. Observa cómo todas las caras cuadrangulares dan lugar a dos caras triangulares.

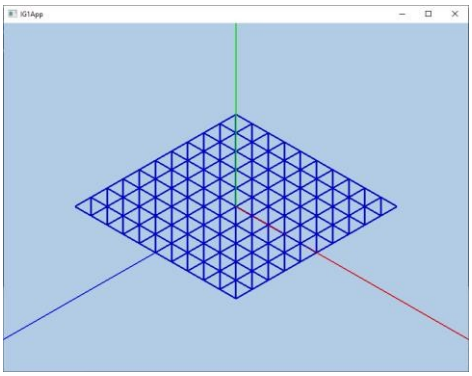


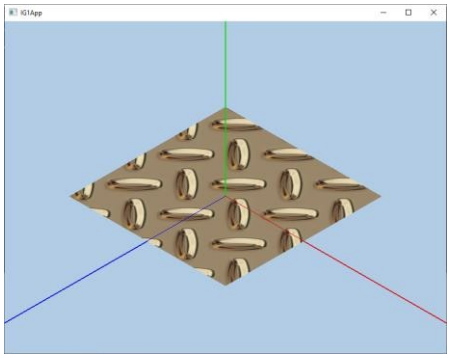


20. Muestra una escena como la de abajo, que contiene dos esferas con el mismo radio y color. La de la izquierda ha sido construida con la clase **`Esfera`** y la de la derecha con la clase de esferas cuádricas **`Sphere`**. Como ves, deben resultar dos esferas iguales, iluminadas de la misma forma. **La renderización de estas dos esferas como se muestra en la captura adjunta será la escena 4 de la Práctica 2**.



21. Define la clase **`Grid`** que hereda de **`Abs_Entity`** y cuyos objetos son mallas de triángulos indexados que forman cuadrículas o rejillas. La constructora de la clase **`Grid`** tiene dos parámetros: **`lado`** es la longitud del cuadrado de la rejilla, y **`nDiv`** es el número de divisiones que se toman en cada lado. Por ejemplo, en modo línea, **`new Grid(200, 10)`** se renderiza en la captura de abajo a la izquierda. El método **`render()`** de la clase **`Grid`** debe contemplar, por supuesto, la posibilidad de renderizar rejillas con textura, como en la captura de abajo a la derecha.





Recuerda que tienes que definir el método:

```
static IndexMesh* generateGrid(GLdouble lado, GLuint nDiv)
```

en la clase **`IndexMesh`**, que genera la malla indexada de la rejilla tal como se ha explicado, con vértices, coordenadas de textura, índices de caras triangulares y vectores normales, por supuesto.

22. Define la clase **`GridCube`** que hereda de **`CompoundEntity`** y cuyos objetos están formados por seis objetos de la clase **`Grid`** dispuestos de manera que formen un cubo. Define una escena que contenga los ejes más un objeto de esta nueva clase. **Esta será la escena 5 de la Práctica 2.** Cuando dispongas las seis cuadrículas del cubo, ten cuidado de dejar siempre las caras frontales mirando al exterior. Esto es importante.

Añade texturas a las rejillas de estos cubos. Pon una textura a las caras superior e inferior y otra al resto. La renderización mostrará entonces un cubo con texturas en todas las caras como el de la captura de abajo. En el cubo de este ejemplo, las caras laterales tienen la textura **`stones.bmp`** mientras que las caras superior e inferior tienen la textura **`checker.bmp`**. Estas texturas las puedes descargar del Campus Virtual.