

## Práctica 0

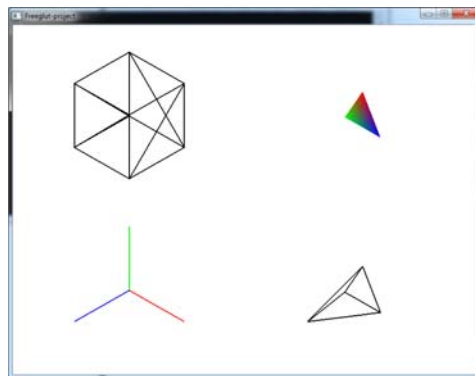
- Define la función `static Mesh* generateTriangle(GLdouble r)` que genera los vértices del triángulo equilátero, de radio  $r$ , centrado en el plano  $Z=0$  (Utiliza la primitiva `TRIANGLES`). Define la clase `Triangulo` que dibuja las líneas del perímetro, y añade un triángulo a la escena.

Recuerda utilizar `glPolygonMode(GL_FRONT_AND_BACK, GL_LINE)` para que solo se rendericen las líneas de los triángulos.

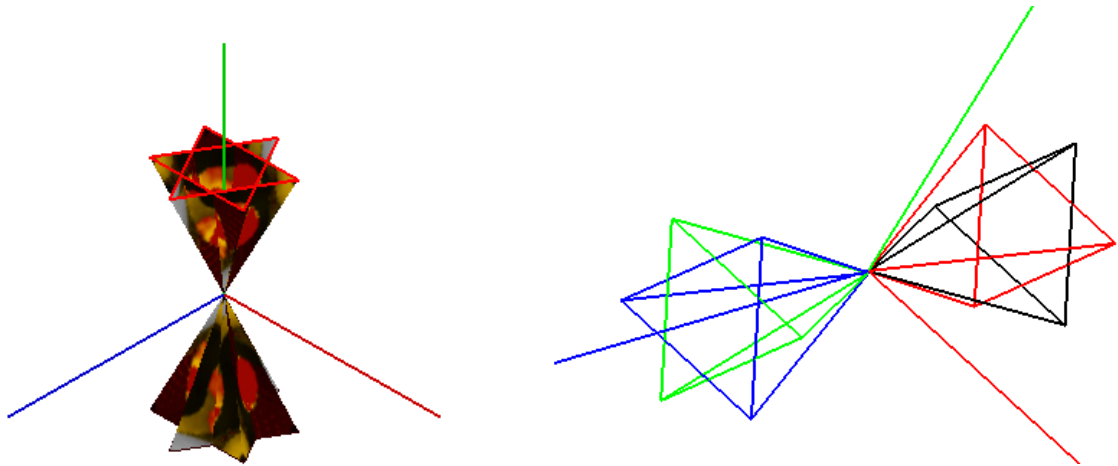
Utiliza la ecuación de la circunferencia, con centro  $C=(0, 0)$  y radio  $R=r$ .

$$\begin{aligned}x &= C_x + R \cos \text{ang} \\y &= C_y + R \sin \text{ang}\end{aligned}$$

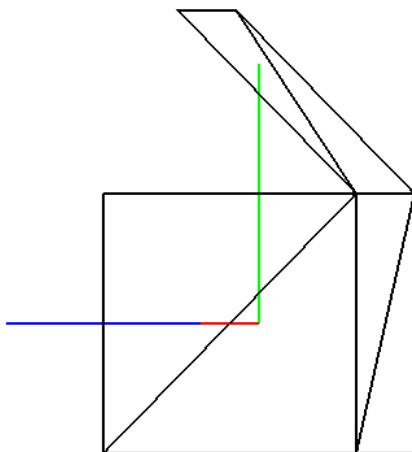
- Define la función `static Mesh* generateTriangleRGB(GLdouble r)` que genera los vértices del triángulo equilátero, de radio  $r$ , centrado en el plano  $Z=0$ , con un color primario en cada vértice. Define la clase `TrianguloRGB` que dibuja el triángulo relleno, y añade una entidad de esta clase a la escena.
- Define la función `static Mesh* generateTriPyramid(GLdouble r, GLdouble h)` que genera los vértices de la pirámide de altura  $h$  en el eje  $Z$ , y base triangular en el plano  $Z=0$  de radio  $r$  (Utiliza la primitiva `TRIANGLE_FAN`). Define la clase `TriPyramid` que dibuja las aristas de la pirámide y añade una entidad de esta clase a la escena.
- Define la función `static Mesh* generateContCubo(GLdouble l)` que genera los vértices del contorno, alrededor del eje  $Y$ , del cubo de lado  $l$ , centrado en el origen de coordenadas (Utiliza la primitiva `TRIANGLE_STRIP`). Define la clase `ContCubo` que dibuja las líneas de los triángulos y añade una entidad de esta clase a la escena.
- Modifica el método `scene::render()` para que dibuje cada entidad en un puerto de vista. Los cuatro puertos de vista juntos tienen que ocupar toda la ventana. Añade a la clase `Viewport` un método para modificar su posición.



- Define la clase `Diabolo` con una malla para una pirámide triangular. El método `render(glm::dmat4 const& modelViewMat)` dibuja cuatro pirámides triangulares formando la siguiente composición (debe aparecer en el eje Z).



- Define la tecla 'a' de forma que el diábolo gire sobre su eje Z.
- Define la función `static Mesh* generateRectangle(GLdouble w, GLdouble h)` que genera los vértices del rectángulo de dimensiones WxH, centrado en el origen de coordenadas (Utiliza la primitiva `TRIANGLE_STRIP`).
- Define la clase `Cubo` con dos mallas: contorno de cubo y cuadrado. El método `render(glm::dmat4 const& modelViewMat)` dibuja un cubo con base y tapa formando la siguiente composición:



Utiliza las opciones:

```
glPolygonMode(GL_FRONT, GL_LINE);
glPolygonMode(GL_BACK, GL_POINT);
```

- Dragón (GL\_POINTS)

La generación de puntos basada en generar otro punto aplicando al anterior una transformación, se aplica a ciertas transformaciones dando lugar a figuras fractales.

Para obtener el Dragón se utilizan dos transformaciones T1 y T2, eligiendo aleatoriamente una de ellas en cada iteración utilizando las probabilidades PR1 y PR2 respectivamente.

Define la función `static Mesh* generaDragon(GLuint numVert)` que genera los vértices comenzando en el (0, 0) y obtiene el siguiente vértice aplicando al anterior:

- Con PR1= 0.787473

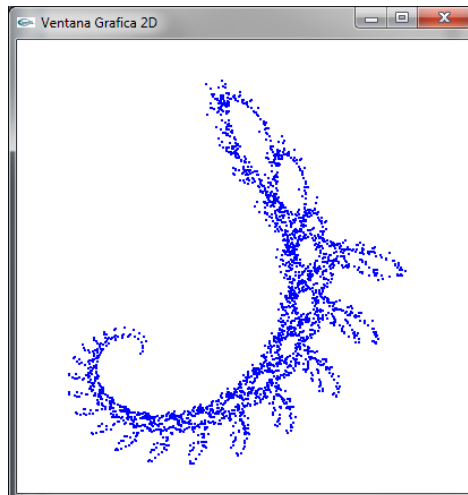
$$T1(x, y) = (0.824074 * x + 0.281482 * y - 0.882290, \\ -0.212346 * x + 0.864198 * y - 0.110607)$$

- Con PR2 = 1 - PR1 = 0.212527

$$T2(x, y) = 0.088272 * x + 0.520988 * y + 0.785360, \\ -0.463889 * x - 0.377778 * y + 8.095795$$

```
double azar= rand() / double(RAND_MAX);
if (azar < PR1) { ... }
else { ... }
```

Define la clase `Dragon` que dibuja los puntos con grosor 2, y añade una entidad de esta clase a la escena. Genera 3000 puntos y establece la matriz de modelado con una traslación de -40 en X y -170 en Y, y una escala de 40 en X e Y.



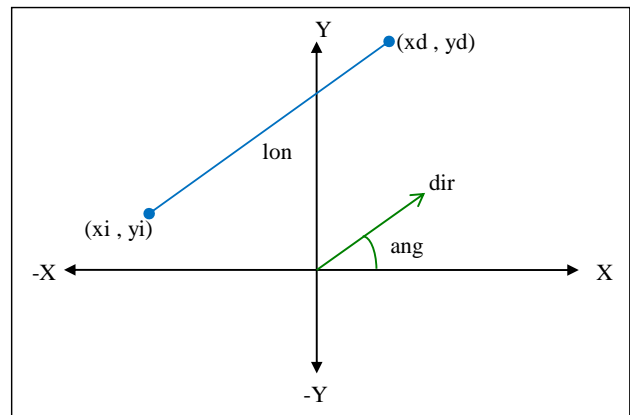
- Poliespirales (GL\_LINE\_STRIP)

Define la función `static Mesh* generaPoliespiral(dvec2 verIni, GLdouble angIni, GLdouble incrAng, GLdouble ladoIni, GLdouble incrLado, GLuint numVert)` que genera los vértices correspondientes a la poliespiral que comienza en el vértice `verIni` y obtiene el siguiente vértice aplicando al anterior `mover(...)`:

$$\text{mover}(x, y, \text{ang}, \text{lon}) = (x + \text{lon} * \cos(\text{ang}), y + \text{lon} * \sin(\text{ang}))$$

El ángulo y la longitud inicial son `angIni` y `ladoIni`, y se van incrementando en `incrAng` e `incrLado` respectivamente. Utiliza la primitiva `LINE_STRIP`.

```
#include <gtc/constants.hpp>
const double PI = glm::pi<double>();
radians(degrees)
// Para transforma grados a radianes
```



Define la clase `Poliespiral` que dibuja las líneas, y añade una entidad de esta clase a la escena.

Prueba con los siguientes datos: `verIni= (0, 0)`, `angIni= 0`

- `incrAng=160`, `lado= 1`, `incrLado= 1`, `numIter= 50`
- `incrAng=72`, `lado= 30`, `incrLado= 0.001`, `numIter= 5`
- `incrAng=60`, `lado= 0.5`, `incrLado= 0.5`, `numIter= 100`
- `incrAng=89.5`, `lado= 0.5`, `incrLado= 0.5`, `numIter= 100`
- `incrAng=45`, `lado= 1`, `incrLado= 1`, `numIter= 50`

