Tetera de Martin Newell, diseñado en 1975 empleando curvas de Bézier (en la Universidad de Utah).

Laboratorio 01

Introducción a la programación grafica con OpenGL

Objetivo: El objetivo de esta práctica[1], es que el alumno reconozca algunas bibliotecas gráficas, además de familiarizarse con **OpenGL**[2] y sus primitivas gráficas; también que reconozca el entorno de desarrollo de este laboratorio (Windows, CodeBlocks[3], GLUT[4] u otros). Puede emplearse Visual C++ (modo consola) o cualquier otra versión de C/C++ de forma indiferente, para el cual se tratará que los ejemplos sean independientes del entorno o IDE.

Esta sección servirá como introducción de los conceptos fundamentales para el desarrollo de programas basados en **OpenGL**, y nos permitirá obtener un marco de trabajo útil para las sesiones posteriores.

Duración de la Práctica: 2 Horas.
Lugar de realización: Laboratorio de cómputo.

El conocimiento requerido para realizar esta sesión práctica, es de haber asimilado los conceptos básicos de programación en C/C++.

El desarrollo tendrá la siguiente estructura de temas:

1. Introducción a las librerías gráficas
2. FLTK: Fast Light Toolkit
3. Maya (de Autodesk)
4. OpenGL: Open Graphics Library
5. Principales características de OpenGL
6. Ventajas de OpenGL
7. Preparando CodeBlocks para programar con OpenGL
8. Programa ejemplo con OpenGL
9. Breve explicación acerca del código usado como ejemplo
10. Manejo de primitivas gráficas para polilíneas
11. Algunas primitivas bidimensionales
12. Ejercicios propuestos
13. Referencias

[1] Estos apuntes de laboratorio han sido redactados y modificados en el transcurso de los semestres anteriores buscando mostrar la información necesaria para motivar al alumno, a su vez sirve como guía en cada sesión; no obstante el alumno debe ampliar los temas desarrollados con la ayuda de la bibliografía sugerida.

[2] Veáse **www.opengl.org**.

[3] Véase **www.codeblocks.org/** para su respectiva descarga.

[4] Veáse **www.opengl.org/resources/libraries/glut/**.

1. INTRODUCCIÓN A LAS LIBRERÍAS GRÁFICAS

Un sistema gráfico se compone principalmente de modelos a graficar (datos u objetos) y su visualización en un dispositivo adecuado. Estos dispositivos (Hardware) son de uso específico ya sea en 2D o 3D. La implementación de los modelos a visualizar se realizan con un nivel de abstracción tal que el usuario hace uso de bibliotecas (también llamadas librerías) graficas, las cuales encapsulan gran parte de las funciones y procedimientos necesarios en el Pipeline (tubería de procesos).

"Las librerías gráficas permiten conformar un software que puede generar imágenes en base a unos modelos matemáticos y unos patrones de iluminación, texturas, etc."

A las librerías gráficas también se les denomina **API** (del inglés **Aplicación Programming Interface - Interfaz de Programación de Aplicaciones**) que viene a ser un conjunto de **especificaciones** de comunicación entre componentes del software. Representan un método para conseguir **abstracción** en la programación, generalmente (aunque no necesariamente) entre los niveles o capas inferiores y los superiores del software. Uno de los principales propósitos de una API consiste en proporcionar un conjunto de **funciones** de uso general, por ejemplo, para dibujar **ventanas** o **iconos** en la pantalla. De esta forma, los **programadores** se benefician de las ventajas de la API haciendo uso de su funcionalidad, evitándose el trabajo de programar todo desde el principio.

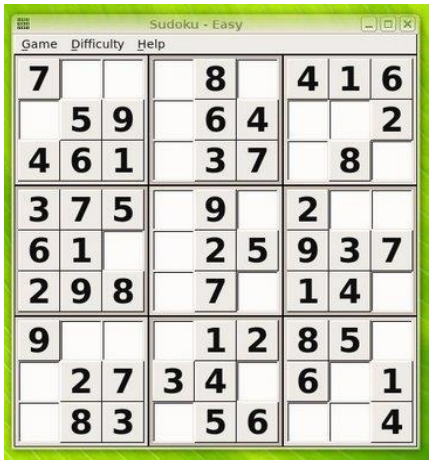
Los objetivos de las librerías gráficas es, en primer lugar que exista independencia del hardware (tanto en los dispositivos de entrada como de salida) que se usa, en segundo lugar debe ser independiente de la aplicación. La librería es accedida a través de una interface única (al menos para cada lenguaje de programación) para cualquier aplicación.

Las librerías gráficas realizan la gestión de imágenes 3D en dos niveles, en **bajo nivel** donde las tareas de gestión de los elementos de la escena se hacen en serie y estas están compuestos por primitivas gráficas y atributos, podemos destacar en este nivel a OpenGL, Direct 3D, Java 3D, etc. en **alto nivel** donde las tareas a realizar son: la gestión global de los elementos de la escena, carga/descarga de memoria, gestión elementos no visibles, elección del modelo geométrico, nivel detalle, textura y elección de la técnica de presentación (rendering); podemos destacar en este nivel el Inventor, Performer, Hewlett Packard, etc.

2. FLTK[5]

La **FLTK - Fast Light Toolkit**" (Herramientas de Software Rápidas y Livianas) es una biblioteca del tipo **GUI** (Graphic User Interface) multiplataforma, desarrollada inicialmente por Bill Spitzak y luego mantenida por un grupo de desarrolladores alrededor del mundo. La biblioteca fue desarrollada teniendo en cuenta la programación de gráficos 3D, para esto, provee una interfaz en OpenGL, pero también permite el desarrollo de aplicaciones de propósito general.

Utilizando sus propios **widgets** (Los widgets de escritorio también se conocen como **gadgets**, como una nueva categoría de mini aplicaciones), trazado de gráficos 2D y sistema de eventos (aunque FLTK ha ganado experiencia y soporte para usar opcionalmente la biblioteca gráfica **Cairo**[6]) desarrollados sobre un código abstracto independiente del sistema, lo cual permite escribir programas que lucen idénticos cuando son compilados en cualquiera de los sistemas operativos soportados.



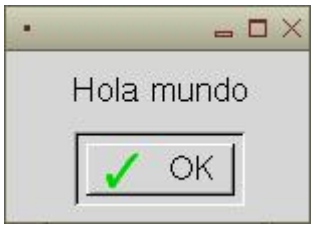
FLTK Sudoku
www.easvsw.com/~mike/sudo

FLTK es software libre, licenciado bajo **LGPL** (GNU Lesser General Public License, antes GNU **Library General**

[5] Véase **http://www.fltk.org/**
[6] Véase **http://cairographics.org/**

Public License o Licencia Pública General para Bibliotecas de GNU) con la cláusula adicional que permite el enlazado estático para aplicaciones con licencias incompatibles. Además incluye **FLUID** (del Inglés, FLTK User Interface Designer), que es una interfaz de diseño gráfico de FLTK que genera los archivos de cabecera y código para una GUI desarrollada. Por ejemplo:

```
int main(int argc, char *argv[]) {
  Fl_Window* w = new Fl_Window(330, 190);
  new Fl_Button(110, 130, 100, 35, "Hola, Mundo!");
  w->end();
  w->show(argc, argv);
  return Fl::run();
}
```



3. Maya[7]

Es un programa informático dedicado al desarrollo de gráficos en tres dimensiones, efectos especiales y animación. Surgió a partir de la evolución de **Power Animator** y de la fusión de **Alias** y **Wavefront**, dos empresas canadienses dedicadas a los gráficos generados por ordenador. Más tarde **Silicon Graphics** (ahora SGI), el gigante informático, absorbió a **Alias-Wavefront**, que finalmente ha sido absorbida por **Autodesk**.

Maya se caracteriza por su potencia y las posibilidades de expansión y personalización de su interfaz y herramientas. **MEL** (Maya Embedded Language) es el código que forma el núcleo de Maya, y gracias al cual se pueden crear scripts y personalizar el paquete.

Maya posee numerosas herramientas para modelado, animación, render, simulación de ropa y cabello, dinámicas (simulación de fluidos), etc.

Además, **Maya** es el único software de 3D acreditado con un **Oscar** gracias al enorme impacto que ha tenido en la industria cinematográfica como herramienta de efectos visuales, con un uso muy extendido debido a su gran capacidad de ampliación y personalización.

Maya trabaja con cualquier tipo de superficie **Nurbs**, Polygons y Subdivision Surfaces, incluye la posibilidad de interactuar entre todos los diferentes tipos de geometría.

"Industrial, Light & Magic (ILM) [8], la casa de efectos visuales galardonada con el Premio a la Academia, confió en el software de entretenimiento digital de Autodesk, Inc. para crear miles de tomas de efectos visuales para las películas Transformers: Revenge of the Fallen, Harry Potter y The Half-Blood Prince, Terminator Salvation y Star Trek. ILM creó efectos visuales impresionantes utilizando Autodesk Maya y Autodesk Inferno".

Transformers renderizado con Maya Autodesk

[7] Véase **autodesk.com/products/maya/overview**
[8] Texto extraído de http://www.3dprofesional.com/noticias/noticia2009071502.html

4. OpenGL[9]

OpenGL es una librería gráfica que provee una interfaz entre el software y el hardware gráfico (un ejemplo de hardware gráfico es la combinación tarjeta de video + monitor de un computador personal). Utilizando esta librería, se crean programas interactivos que generan imágenes en color de objetos tridimensionales.

Desde el punto de vista del programador OpenGL es una API para interactuar con dispositivos gráficos y aceleradoras 3D. Es una biblioteca de trazado de gráficos de alto rendimiento, fácil de usar, intuitivo, portable, en el que el programador describe las llamadas necesarias a funciones y comandos para conseguir una escena, apariencia o efecto.

OpenGL contiene más de un centenar de comandos que nos ayudan a definir objetos, aplicar transformaciones a esos objetos, cambiar sus propiedades (color, textura, luz, etc.), posición de la cámara, entre otros. OpenGL fue creada con el objetivo de ser estándar y disponible en distintos sistemas y entornos de ejecución (plataformas de software). Su antecesora fue **Iris GL** primera API de programación para estaciones gráficas de alto rendimiento desarrollada en 1982 por Silicon Graphics, Inc (líder mundial en gráficos y animaciones por ordenador).

En el año de 1992 muchas empresas del hardware y software se pusieron de acuerdo para desarrollar conjuntamente una librería gráfica libre: OpenGL. Entre estas empresas destacan, Silicon Graphics Inc., Microsoft, IBM Corporation, Sun Microsystems, Digital Equipment Corporation (DEC), Hewlett-Packard Corporation, Intel e Intergraph Corporation. Así nació OpenGL (Open Graphics Library).

Existen versiones OpenGL para XWindows bajo UNIX (Linux, SGI, SUN, Dec, etc.), Microsoft Windows, MacOS y para distintos sistemas embebidos (OpenGL corre en los display CRT de los aviones espía U2, por ejemplo). Puesto que cada una de estas plataformas maneja de distinta manera un display, las componentes de interacción (botones, ventanas, recepción de input del usuario, etc), no forman parte de OpenGL. Un programa que utilice OpenGL necesita de un componente adicional, que realice la tarea de hacer de puente entre OpenGL y la plataforma particular donde corre. En una plataforma con un sistema de ventanas como Microsoft Windows, por ejemplo, esta componente se encargará entre otras cosas de abrir una ventana y prepararlo todo para que OpenGL pueda dibujar sobre ella.

Cada plataforma tiene su propia librería para este efecto. Como ejemplos, en XWindows de UNIX la librería es GLX, y en Microsoft Windows es wGL. Cada una de estas librerías maneja los detalles y particularidades de esos sistemas de ventanas. Para resolver el problema de manejo de ventanas, **Mark J. Kilgard** de SGI creó la librería **GLUT-library (OpenGL Utility Toolkit)**. GLUT hace el trabajo de puente entre OpenGL y el sistema de ventanas, de una manera portable. Puesto que GLUT debe funcionar en distintas plataformas, sólo provee una funcionalidad mínima desde el punto de vista de manejo de ventanas, menús e input.

5. PRINCIPALES CARACTERISTICAS DE OpenGL

- **Portabilidad:** *OpenGL* es por diseño independiente del hardware, sistema operativo o sistema de ventanas. Las funciones de *OpenGL* trabajan de la misma forma en cualquier plataforma, es decir, se pueden llamar a las funciones de *OpenGL* desde un programa escrito y obtener los mismos resultados independientemente del lugar donde el programa se está ejecutando. Esto supone que sea portable y que la programación mediante *OpenGL* sea más fácil de realizar que con un API integrado en un sistema de ventanas. La independencia del hardware se basa en no incluir funciones para la gestión de ventanas, interactividad con el usuario, ni manejo de ficheros. Esto se debe a que dichas funciones están definidas en cada sistema de operativo, para obtener la portabilidad, en *OpenGL* los programas desarrollados con esta API interactúan con el sistema de ventanas de la plataforma donde los gráficos son visualizados (normalmente en una ventana). Hay varias herramientas y bibliotecas para el manejo de ventanas desarrolladas para trabajar con *OpenGL*.

[9] Véase **http://www.opengl.org/**, actualmente en la versión 4.3 (Agosto, 2012) compite con Direct3D auspiciada por Microsoft. Vea también **http://es.wikipedia.org/wiki/OpenGL** para una descripción resumida de las diferentes versiones.

- **Perceptiva a la red:** Es perceptiva a la red, de manera que es posible separar nuestra aplicación *OpenGL* en un servidor y un cliente que verdaderamente produzca los gráficos. Existe un protocolo para mover por la red los comandos *OpenGL* entre el servidor y el cliente. Gracias a su independencia del sistema operativo, el servidor y el cliente no tiene porque ejecutarse en el mismo tipo de plataforma, muy a menudo el servidor será un supercomputador ejecutando una compleja simulación y el cliente una simple estación de trabajo mayormente dedicada a la visualización gráfica. *OpenGL* permite al desarrollador escribir aplicaciones que se puedan desplegar en varias plataformas fácilmente.

- **Funciones para crear gráficos 2D y 3D:** Las funciones de *OpenGL* como ya se comentó anteriormente, están diseñadas para permitir crear gráficos 2D y 3D con especial énfasis en 3D. *OpenGL* tiene funciones o primitivas con las que se podrá realizar modelado 3D, transformaciones, utilización de color e iluminación, sombras, mapeado de texturas, animación, movimiento borroso que se describen brevemente a continuación:

  - ✓ **Primitivas geométricas:** Permiten construir descripciones matemáticas de objetos. Las actuales primitivas son: puntos, líneas, polígonos, imágenes y bitmaps.
  - ✓ **Codificación del Color:** en modos RGBA (Rojo-Verde-Azul-Alfa) o de color indexado.
  - ✓ **Visualización y Modelado**: que permite disponer objetos en una escena tridimensional, mover nuestra cámara por el espacio y seleccionar la posición ventajosa deseada para visualizar la escena de composición.
  - ✓ **Mapeado de texturas:** que ayuda a traer realismo a nuestros modelos por medio del dibujo de superficies realistas en las caras de nuestros modelos poligonales
  - ✓ I**luminación de materiales:** es una parte indispensable de cualquier gráfico 3D. *OpenGL* provee de comandos para calcular el color de cualquier punto dado las propiedades del material y las fuentes de luz en la habitación.
  - ✓ **Doble buffering:** ayuda a eliminar el parpadeo de las animaciones, cada fotograma consecutivo en una animación se construye en un buffer separado de memoria y mostrado solo cuando está completo.
  - ✓ **Anti-alizado:** reduce los bordes escalonados en las líneas dibujadas sobre una pantalla de ordenador. Los bordes escalonados aparecen a menudo cuando las líneas se dibujan con baja resolución. El anti-alizado es una técnica común en gráficos de ordenador que modifica el color y la intensidad de los píxeles cercanos a la línea para reducir el zig-zag artificial.
  - ✓ **Sombreado Gouraud:** es una técnica usada para aplicar sombreados suaves a un objeto 3D y producir una sutil diferencia de color por sus superficies.
  - ✓ **Z-buffering [10]:** mantiene registros de la coordenada Z de un objeto 3D. El Z-buffer se usa para registrar la proximidad de un objeto al observador, y es también crucial para el eliminado de superficies ocultas.
  - ✓ **Efectos atmosféricos:** como la niebla, el humo y las neblinas hacen que las imágenes producidas por ordenador sean más realistas. Sin efectos atmosféricos las imágenes aparecen a veces irrealmente nítidas y bien definidas. *Niebla* es un término que en realidad describe un algoritmo que simula neblinas, brumas, humo o polución o simplemente el efecto del aire, añadiendo profundidad a nuestras imágenes.
  - ✓ **Alpha blending:** usa el valor Alfa (valor de material difuso) del código RGBA, y permite combinar el color del fragmento que se procesa con el del píxel que ya está en el buffer. Imagine por ejemplo dibujar una ventana transparente de color azul claro enfrente de una caja roja. El Alpha Blending permite simular la transparencia de la ventana, de manera que la caja vista a través del cristal aparezca con un tono magenta.
  - ✓ **Listas de Display:** permiten almacenar comandos de dibujo en una lista para un trazado posterior, cuando las listas de display (listas de visualización) se usan apropiadamente puedan mejorar mucho el rendimiento de nuestras aplicaciones.
  - ✓ **Evaluadores Polinómicos:** sirven para soportar B-Splines racionales no uniformes, esto es para ayudar a dibujar curvas suaves a través de unos cuantos puntos de referencia, ahorrándose la necesidad de acumular grandes cantidades de puntos intermedios.
  - ✓ **Características de Feedback, Selección y Elección** que ayudan a crear aplicaciones que permiten al usuario seleccionar una región de la pantalla o elegir un objeto dibujado en la

[10] Buffer de datos, conceptos diseñados por Edwin Catmull y Wolfgag Straßer. Actualmente las tarjetas de video la traen implementada a través de un algoritmo.

misma. El modo de feedback permite al desarrollador obtener los resultados de los cálculos de trazado.
- ✓ **Primitivas de Raster** (bitmaps y rectángulos de pixeles)
- ✓ **Operaciones con Pixeles**
- ✓ **Transformaciones:** rotación, escalado, perspectivas en 3D.

- **Ausencia de comandos para modelos complejos:** Para conseguir rendimiento no tiene comandos para describir modelos complejos (mapas, pájaros, moléculas, etc.) sino que tiene primitivas que permiten dibujar puntos, líneas y polígonos, y es el programador el que tendrá que construir objetos más complejos.

- **Ejecución inmediata:** Con **OpenGL** cualquier comando es ejecutado inmediatamente. Cuando en un programa se especifica que dibuje algo, lo hace inmediatamente. Pero existe la opción de poner comandos en Display Lists. Una **display list** es una lista no editable de comandos almacenados para una ejecución posterior y se puede ejecutar más de una vez. Por ejemplo, se pueden usar para redibujar el gráfico cuando el usuario cambia el tamaño de la ventana, también se puede utilizar para dibujar la misma forma más de una vez si esta se repite como un elemento de un gráfico.

- **Sintaxis Común:** Todos los comandos de **OpenGL** utilizan la misma sintaxis. Todos ellos usan el prefijo **gl** y después la palabra que forma el comando con la primera letra en mayúscula. También en algunos comandos va seguido como sufijo un número y una letra, que indican el número de parámetros del comando y el tipo de sus argumentos. Por ejemplo:

  glColor3f(1.0,1.0,1.0)

  Éste es un comando que permite cambiar el color activo (Color) y tiene 3 parámetros de tipo **float**. Además, algunos comandos pueden llevar la letra *v* al final que indica que el parámetro de la función es un puntero a un array o vector.

- **Máquina de estados: OpenGL** se puede considerar como una **máquina de estados**. Las aplicaciones se pueden poner en varios estados que permanecen en efecto hasta que se cambian. Por ejemplo el color, se puede poner un color y dibujar un objeto, y ese color seguirá activo hasta que se ponga otro color. **OpenGL** tiene las siguientes variables de estado:

  - Color
  - Vista actual
  - Transformaciones de proyecciones
  - Posiciones
  - Luces
  - Propiedades de material.

  Algunas de estas variables de estado se pueden habilitar o deshabilitar con los comandos *glEnable()* o *glDisable()*.

  Cada variable de estado tiene un valor por defecto disponible en todo momento a la consulta por parte del programador. También es posible el almacenamiento del valor de estas variables de estado para posteriormente recuperarlo. Las funciones que permiten hacer esto son *glPushAttrib()* y *glPopAttrib()*.

- **Variedad de lenguajes: OpenGL** puede ser invocado desde cualquiera de los siguientes lenguajes C, C++, Fortran, Ada, Java. Aunque se puede considerar C el más popular, hecho demostrado en que todos los manuales y documentación de **Opengl** se desarrollan con C. Otros lenguajes de programación como Visual Basic que pueden invocar a funciones de C, también podrán hacer uso de **OpenGL.**

- **Pipeline de renderizado:** Con **OpenGL** se debe seguir un orden para la realización de las operaciones graficas necesarias para renderizar una imagen en la pantalla. La mayor parte de las implementaciones de **OpenGL** siguen un mismo orden en sus operaciones, una serie de

plataformas de proceso, que en su conjunto crean lo que se suele llamar el "OpenGL Rendering Pipeline"

- **Tipos de datos propios: OpenGL** tiene sus propios tipos de datos para así hacer el código más fácilmente portable, aunque estos tipos de datos corresponden con los tipos de datos de C, y por tanto se podrán utilizar unos u otros indistintamente. Es necesario tener en cuenta que si se utilizan los tipos de datos de C, dependiendo del compilador y entorno habrá unas reglas para planificar el espacio de memoria de varias variables. Por tanto se recomienda el uso de los tipos de **Opengl**. Todos los tipos de **Opengl** tienen el sufijo **gl** y a continuación el tipo de C correspondiente.

6. VENTAJAS DE OpenGL

- **El estándar de la industria:** Un consorcio independiente, el *OpenGL Architecture Review Borrad (Junta de Revisión de Arquitecturas),* guía la especificación **OpenGL**, acepta o rechaza cambios y propone pruebas de conformidad. Con amplio soporte de la industria, **OpenGL** es el único verdaderamente abierto, vendedor neutral, estándar de gráficos multiplataforma.
- **Estable:** Las implementaciones **OpenGL** han estado disponibles por más de **siete años** en una amplia variedad de plataformas. Las adiciones para la especificación están adecuadamente controladas, y las actualizaciones propuestas salen a la luz con el tiempo para que los desarrolladores adopten cambios. Los requisitos retrasados de compatibilidad aseguran que las aplicaciones existentes no se conviertan obsoletas.
- **Confiabilidad y portabilidad:** Todas las aplicaciones **OpenGL** producen resultados visuales uniformes de despliegue en cualquier hardware condescendiente a API **OpenGL**, a pesar del sistema operativo o sistema de ventanas.
- **Constante evolución:** Por su diseño perfeccionista y de enfoque al futuro, **OpenGL** permite que las nuevas innovaciones de hardware ser accesible a través del API por el mecanismo de la extensión **OpenGL**. De este modo, las innovaciones aparecen en el API en un momento oportuno, permiten a los desarrolladores de aplicaciones y vendedores de hardware incorporar nuevas características en su producto normal puesto en circulación.
- **Escalable:** Las aplicaciones basadas a API **OpenGL** pueden funcionar con sistemas de órdenes subsiguientes de usuarios para PCs, las estaciones de trabajo, y las supercomputadoras. Como resultado, las aplicaciones pueden escalar a cualquier clase de máquina que el desarrollador pueda escoger para sus objetivos.
- **Fácil para usar: OpenGL** está adecuadamente estructurado con un diseño intuitivo y órdenes lógicas. Las rutinas eficientes **OpenGL** típicamente dan como resultado aplicaciones con menos líneas de código que aquellos que generan programas utilizando otras bibliotecas gráficas u otros paquetes. Además, los drivers **OpenGL** encapsulan información acerca del hardware subyacente, liberando el desarrollo de aplicaciones de diseño para características específicas de hardware.
- **Adecuadamente documentada:** Numerosos libros han sido publicados acerca de **OpenGL**, y una gran cantidad de código de muestra está fácilmente disponible, generando información acerca de **OpenGL** económica y fácil para obtener.

OpenGL es una **API** (*Applications Programming Interface*), o sea una librería**.** Llamando sucesivamente a sus funciones se consigue un comportamiento adecuado sin importarnos en un primer momento que es lo que esta API está haciendo en realidad. Contamos pues con diversos tipos de funciones:

- **Funciones primitivas,** que definen todos los objetos a bajo nivel como pueden ser puntos, líneas o polígonos más o menos complejos.

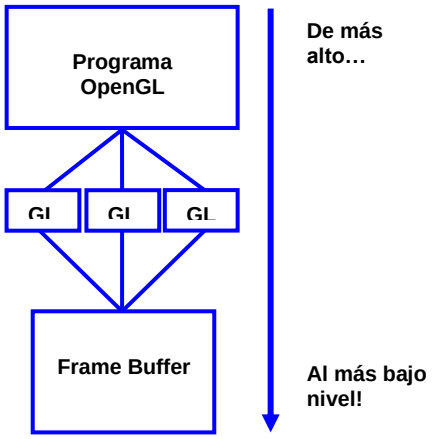
- **Funciones atributivas,** que nos permitirán definir las características de todo aquello que dibujemos, por ejemplo "glColor".

- **Funciones de visualización,** para posicionar la cámara, proyectar la geometría a la pantalla o eliminar (Clipping) aquellos objetos fuera de nuestro ángulo de visión.

- **Funciones de transformación,** para girar, rotar, trasladar, escalar la geometría.

- **Funciones de entrada,** para poder generar aplicaciones interactivas en las que el usuario participe. Comúnmente relativas al uso del teclado y del ratón.

Todas las llamadas a OpenGL empiezan siempre con las letras **gl**, pero no solo se cuenta con estas funciones, observe el siguiente diagrama adjunto.



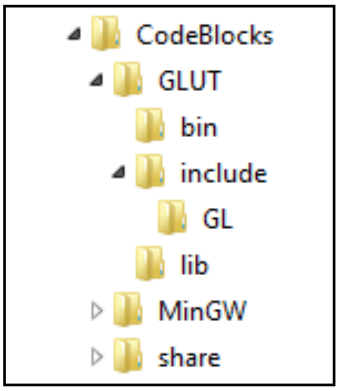
Las librerías que forman la API OpenGL son:

Biblioteca	Archivos	Descripción
Gl	Gl.h Opengl32.lib	**OpenGL Library (GL):** Funciones del núcleo de **OpenGL**. Se limitan a facilitar las operaciones que podríamos denominar gráficas puras, no existiendo procedimientos para el uso de matrices, funciones cuadráticas y similares o funciones de manejo de ventanas y eventos. Por ello, se utiliza unas bibliotecas complementarias que son una extensión a OpenGL para facilitar dichas operaciones.
Glu	Glu.h Glu32.lib	**OpenGL Utility Library (GLU):** Librería de utilidades más comunes. Totalmente portable. Esta librería proporciona una serie de funciones que sirven para tareas tales como actualizar matrices para obtener orientaciones y proyecciones del punto de vista adecuado, generar superficies etc... Todas las funciones de esta librería comienzan por las letras "**glu**", y se irán describiendo a medida que sean útiles.
Aux	Glaux.h Glaux.dll Glaux.lib	**OpenGL Auxiliary Library (AUX):** Librería de recursos. Son muy útiles y cubren una gran variedad de funciones como gestionar las ventanas independientes de la plataforma, gestionar los eventos de entrada a la aplicación o dibujar objetos complejos en 3D.
Glut	Glut.h Glut32.dll Glut32.lib	**OpenGL Utility Toolkit (GLUT):** Librería de recursos posterior a la AUX. Añade nuevas prestaciones. GLUT ofrece una interfaz común para múltiples plataformas para el manejo de ventanas, buffers, renderización de texto, entrada por teclado y menús, por lo que, los desarrolladores podrán usar un interface común para el sistema de ventanas independiente de la plataforma, es decir, las aplicaciones de **OpenGL** que usan GLUT se pueden portar entre plataformas sin tener que introducir muchos cambios en el código fuente.

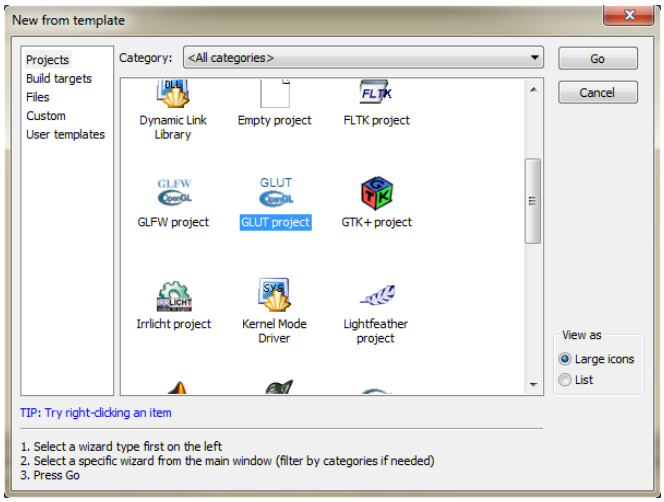
7. PREPARANDO CODEBLOCKS PARA PROGRAMAR CON OpenGL

Se sugiere descargar la versión que incluye el compilador GNU GCC

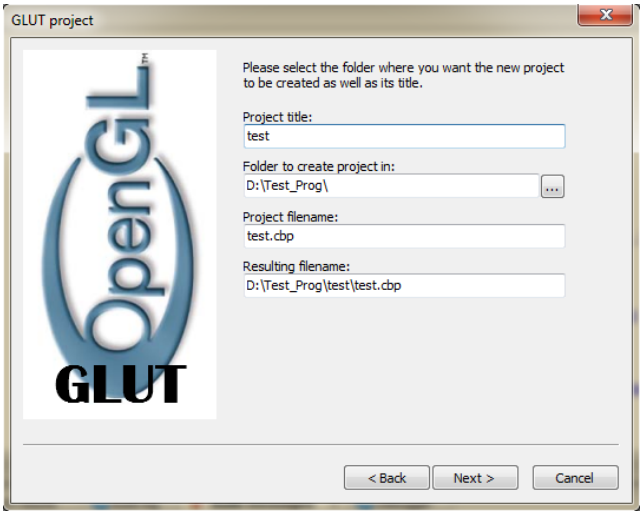
- **Paso 1:** Desempaquetar el **OPENGL95.ZIP (o la que obtuviese: GLUT)** y ubicarlo en un directorio (elección libre) de tal forma que la estructura final de los archivos debe ser como sigue:



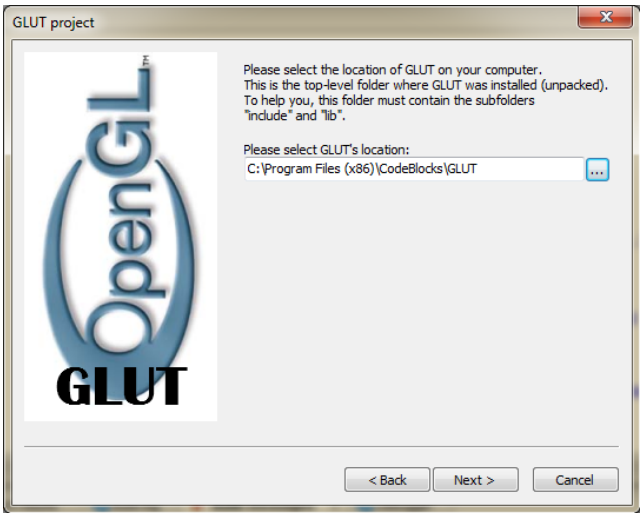
- **Paso 2:** Crear un nuevo proyecto: al menú **Archivo – Nuevo- Proyecto.** Seleccionar **GLUT - Project**.



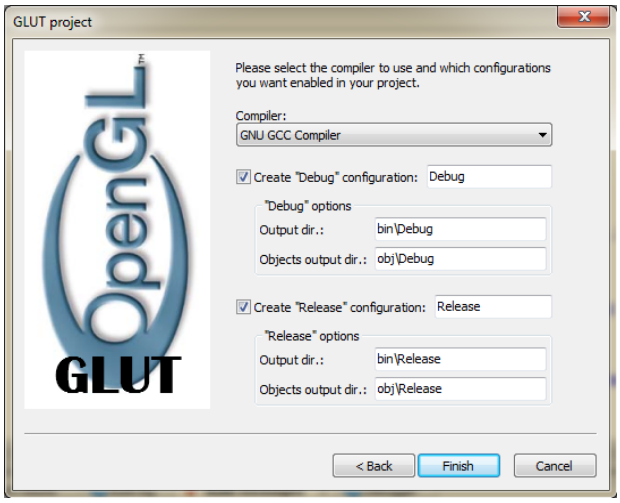
- **Paso 3:** En Asistente, indicar el nombre del proyecto.



- **Paso 4:** Especificar la carpeta en donde se ubica la librería GLUT:



- **Paso 5:** Finalmente seleccione GNU GCC Compiler y Finish.



8. PROGRAMA EJEMPLO CON OpenGL

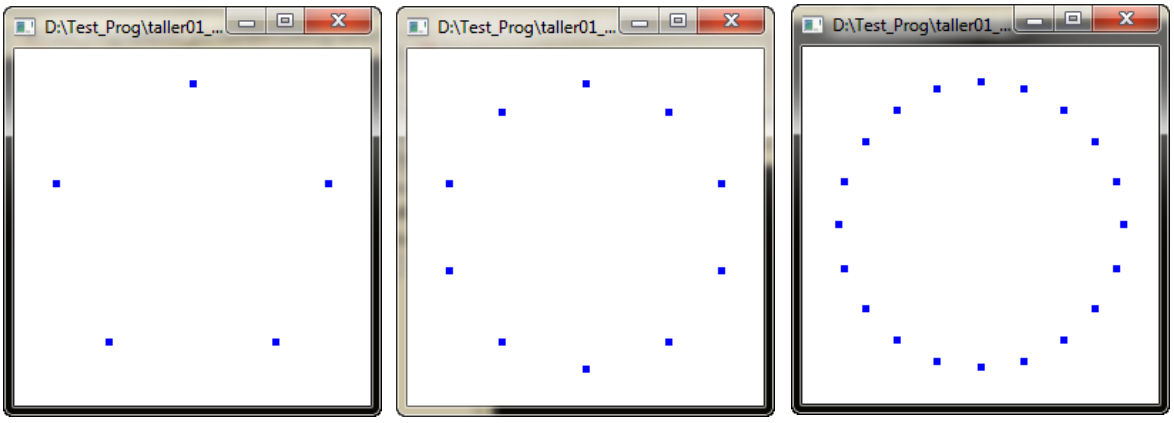
Motivación:

Nos gustaría generar los cinco vértices de un pentágono sobre una circunferencia circunscrita. Para esto podemos usar las siguientes ecuaciones paramétricas para describir la circunferencia (donde $0 \leq \theta \leq 2\pi$):

$$x = \text{radio sen } \theta$$
$$y = \text{radio cos } \theta$$

Luego iremos incrementando el número de coordenadas (5, 10, 20 a más) para obtener una mejor representación gráfica de la circunferencia.

Un programa realizado con el lenguaje C para resolver el problema planteado, se describe a continuación:




```
1. #include <stdlib.h>
2. #include <math.h>
3. #include <GL/glut.h>
4.
5. void init(void);
6. void display(void);
7. void reshape(int,int);
8. int main(int argc, char** argv)
9. {
10.glutInit(&argc, argv);
11.glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
12.glutInitWindowSize(250,250);
13.glutInitWindowPosition(100,100);
14.glutCreateWindow(argv[0]);
```

```
15.init();
16.glutDisplayFunc(display);
17.glutReshapeFunc(reshape);
18.glutMainLoop();
19.return 0;
20.}
21.void init(void)
22.{
23. glClearColor(0.0,0.0,0.0,0.0);  //parametros: rojo, amarillo, azul, el cuarto es el parametro alpha
24.glShadeModel(GL_FLAT);
25.}

26.void display(void)
27.{
28.GLfloat ang, radio = 8.0f, x, y;
29.glClear(GL_COLOR_BUFFER_BIT);
30.glPushMatrix();                              // salva el estado actual de la matriz
31.glBegin(GL_POINTS);
            for (ang = 0.0f; ang < 2 * GL_PI; ang += 2*GL_PI/5)
            {
                  x = radio * sin(ang);
                  y = radio * cos(ang);
                  glVertex2f(x,y);
            }
32.glEnd();
33.glPopMatrix();   // reecupera el estado del matriz
34.glFlush();
35.}
36.void reshape(int w, int h)
37.{
38.glViewport(0,0,(GLsizei)w, (GLsizei)h);
39.glMatrixMode(GL_PROJECTION);
40.glLoadIdentity();
41.glOrtho(-10.0,10.0,-10.0,10,-10.0,10.0);
42.glMatrixMode(GL_MODELVIEW);
43.glLoadIdentity();
44.}
```

9. BREVE EXPLICACIÓN ACERCA DEL CÓDIGO USADO EN EL EJEMPLO

Para comprender mejor el lenguaje, iremos línea a línea analizando qué significa y cómo se utilizó el OpenGL en el problema anterior.

```
1.  #include <stdlib.h>
2.  #include <math.h>
3.  #include <GL/glut.h>
4.
```

Las líneas del 1 al 4 es lo que se conoce como directivas del preprocesador, todos los compiladores de C/C++ disponen de un preprocesador, un programa que examina el código antes de compilarlo. En el se incluyen los archivos de cabecera de las librerías del C/C++ y las librerías del OpenGL que contienen las funciones utilizadas en el programa.

```
5.  void init(void);
6.  void display(void);
7.  void reshape(int,int);
```

Las líneas del 5 al 7 son las funciones prototipo usadas en el programa.

```
8.  int main(int argc, char** argv)
9.  {
```

```
10.glutInit(&argc, argv);
11.glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
12.glutInitWindowSize(250,250);
13.glutInitWindowPosition(100,100);
14.glutCreateWindow(argv[0]); //  pruebe con glutCreateWindow("Taller 01");
15.init();
16.glutDisplayFunc(display);
17.glutReshapeFunc(reshape);
18.glutMainLoop();
19.return 0;
20.}
```

Cualquier programa en C se caracteriza por su función main() o principal, función que se llama automáticamente al iniciar la ejecución del programa. Debemos recoger los parámetros de la línea de comandos, argc y argv, mediante **glutInit**(), como sigue:

glutInit (&argc, argv);

Ahora hay que decirle al motor gráfico como queremos "renderizar"; es decir, si hay que refrescar la pantalla o no, que "buffers" hay que activar/desactivar y que modalidad de colores queremos usar. Para ello empleamos las siguientes funciones:

glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);

Se puede definir también el doble buffer (GLUT_DOUBLE) al igual que GLUT_SINGLE. Hay dos posibilidades en el momento de colorear, podemos usar colores indexados, es decir referirnos a ellos por un identificador y que el sistema los busque en una lista, o bien podemos usar la convención RGB. En nuestro caso le decimos al subsistema gráfico que cada color a aplicar será definido por tres valores numéricos, uno para el rojo (Red), otro para el verde (Green) y otro para el azul (Blue). Recordar la teoría aditiva del color, para esto usamos la constante GLUT_RGB. Con un buffer simple, contamos tan solo con un área de memoria que se redibuja constantemente (esto es con GLUT_SINGLE). Esto no es factible para aplicaciones donde la velocidad de render es muy alta o el contenido gráfico es elevado. En nuestro caso sólo definimos un polígono y por lo tanto nos conformaremos con el buffer simple. De todas formas casi siempre utilizaremos el buffer doble, dos zonas de memoria que se alternan de manera que se dibuja primero una y en la siguiente iteración se dibuja la otra.

glutInitWindowSize (ANCHO, ALTO);

Con esta orden GLUT nos permite definir las medidas de nuestra ventana de visualización. Estamos definiendo literalmente el ANCHO y ALTO de nuestra ventana (en píxeles).

glutInitWindowPosition (ORIGENX, ORIGENY);

Nos permite colocar la ventana en algún punto determinado de la pantalla. Después podremos moverla con el ratón, si no lo impedimos por código; pero siempre hay que indicar un punto de origen para la primera vez que ejecutamos la aplicación.

glutCreateWindow ("Mi Grafico");

Una vez ya definido como "renderizar", con qué medidas de ventana y en qué posición física de la pantalla, creamos la ventana. Para ello le damos un nombre cualquiera que será el título que aparecerá en esta. De hecho a la función se le pasa un array de caracteres, ya sea explícito o sea el nombre entre comillas, o bien implícito, es decir, una variable que contiene el titulo.

glutDisplayFunc(funcion_Dibujar);
Con esto le decimos a la librería que cada vez que note que se debe redibujar, llame a una función que hemos llamado dibujar (en la mayoría de ejemplos aparece con DISPLAY).

glutReshapeFunc(funcion_ControlDeVentana);
Controla el cambio de tamaño de la ventana de visualización.

glutMainLoop();

Usualmente se suele entrar en un bucle infinito que domine cualquier aplicación OpenGL. Con la función MainLoop, que siempre se pone al final del main, le decimos a la librería que espere eternamente a que se produzcan "eventos", es decir, que hasta que no ocurra algo se mantenga a la expectativa. En nuestro caso el único evento posible es el propio "render" dado que aún no sabemos interactuar con el ratón, ni sabemos controlar que pasa cuando se mueve la pantalla o se redimensiona (luego veremos ese tema).

```
21.void init(void)
22.{
23.glClearColor(0.0,0.0,0.0,0.0);  //parametros: rojo, amarillo, azul, el cuarto es el parametro alpha
24.glShadeModel(GL_FLAT);
25.}
```

Con esta función inicializamos ciertos parámetros como el color de la ventana a usar.

glClearColor(0.0, 0.0, 0.0, 0.0)

Con esto se define el color con el que se borrara el buffer al hacer un glClear(). Los 3 primeros parámetros son las componentes R, G y B, siguiendo un rango de [0,1]. La última es el valor **alpha**, del que hablaremos más adelante.

24.glShadeModel(GL_FLAT);

Las primitivas de **OpenGL** están siempre sombreadas, pero el modelo de sombreado puede ser plano (GL_FLAT) o suave (GL_SMOOTH). Si especificamos un color diferente para cada vértice, la imagen resultante variará con cada modelo de sombreado. Con el sombreado suave, el color de los puntos interiores del polígono estará interpolado entre los colores especificados para los vértices. En el sombreado plano, el color especificado para el último vértice se usa en toda la región de la primitiva. La única excepción es GL_POLYGON, en cuyo caso el color usado en toda la región es el especificado para el primer vértice.

```
25.void display(void)
26.{
27.GLfloat ang, radio = 8.0f, x, y;
28.glClear(GL_COLOR_BUFFER_BIT);
29.glPushMatrix();                          // salva el estado actual de la matriz
30.glBegin(GL_POINTS);
31.for (ang = 0.0f; ang < 2 * GL_PI; ang += 2*GL_PI/5)
   {
     x = radio * sin(ang);
     y = radio * cos(ang);
    glVertex2f(x,y);
   }
32.glEnd();
27.GLfloat ang, radio = 8.0f, x, y;
```

Define los tipos de datos con la que va a trabajar el OpenGL. Las declaraciones de variables usualmente definidas como enteras (int), reales (float) y de doble precisión (double) son definidas mediante: Glint, Glfloat y Gldouble respectivamente. Los tipos de datos que usa el OpenGL son:

Tipo en OpenGL	Tipo en C	Representación interna
GLbyte	char en signo	Entero de 8 bits
GLshort	short	Entero de 16 bits
GLint, GLsizei	int	Entero de 32 bits

GLfloat, GLclampf	float	Coma flotante de 32bits
GLdouble, GLclampd	double	Coma flotante de 64 bits
GLubyte, GLboolean	unsigned char	Entero de 8 bits sin signo
GLushort	unsigned short	Entero de 16 bits sin signo
GLuint, GLenum, GLbitfield	unsigned long	Entero de 32 bits sin signo

28.glClear(GL_COLOR_BUFFER_BIT)

Borra un buffer, o una combinación de varios, definidos por flags. En este caso, el buffer de los colores lo borra (en realidad, cada componente R G y B tienen un buffer distinto, pero aquí los trata como el mismo). Para borrarlos utiliza el color que ha sido previamente definido en init() mediante glClearColor(), en este caso, el (0,0,0,0) es decir, negro.

29.glPushMatrix();

Esta función se emplea para colocar la matriz actual en la pila de matrices actual. Esto se emplea a menudo para almacenar la matriz de transformación actual de manera que pueda recuperarse más tarde con una llamada a glPopMatrix.

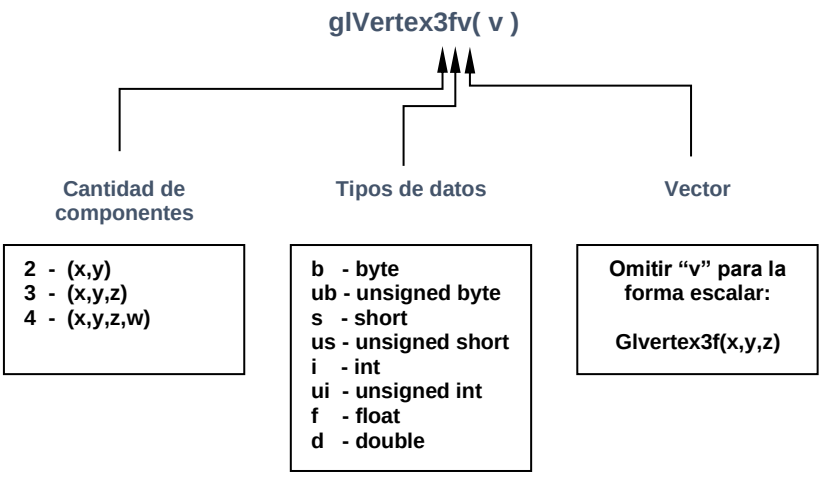
```
30.glBegin(GL_POINTS);
for (ang = 0.0f; ang < 2 * GL_PI; ang += 2*GL_PI/5)
{
x = radio * sin(ang);
y = radio * cos(ang);
glVertex2f(x,y);
}
31.glEnd();
```

Objetos Begin y End: A partir de objetos básicos podemos construir objetos cada vez más complejos (por ejemplo cualquier polígono), esto se consigue usando las funciones **glBegin** y **glEnd** como sigue:

```
void glBegin(modo)
...// objetos
...// a encapsular
void glEnd(void)
```

El parámetro **modo** se encapsula entre las funciones **glBegin y glEnd**. El parámetro modo que recibe la primera, sirve para decirle a OpenGL que tipo de objeto deseamos crear, para esto hemos optado por indicarle que será una nube puntos lo que se encapsulará.

El formato de los comandos de OpenGL para especificar vértices (o coordenadas) es el siguiente:



```
36.void reshape(int w, int h)
37.{
38.glViewport(0,0,(GLsizei)w, (GLsizei)h);
39.glMatrixMode(GL_PROJECTION);
40.glLoadIdentity();
41.glOrtho(-10.0,10.0,-10.0,10,-10.0,10.0);
42.glMatrixMode(GL_MODELVIEW);
43.glLoadIdentity();
44.}
```

La función **void reshape()** controla el cambio de tamaño de la ventana de visualización

38.glViewport(0,0,(GLsizei)w, (GLsizei)h);

glViewport(Glint x,Glint y,GLsizei w,Glsizei h);

Define un área rectangular de ventana de visualización, donde (x,y) es la esquina inferior izquierda, w y h son el ancho y la altura correspondientemente del **Viewport**, después de activarla, es ahí en donde se dibuja.

39.glMatrixMode(GL_PROJECTION);

Ahora parametrizamos nuestra proyección; es decir, le decimos a OpenGL como debe proyectar nuestros gráficos en pantalla. Por ello y para empezar le decimos que active la matriz de proyección que es la que vamos a retocar.

40.glLoadIdentity();

Limpia la matriz de proyección inicializándola con la matriz identidad.

41.glOrtho(-10.0,10.0,-10.0,10,-10.0,10.0);

glOrtho (x, ANCHO, y, ALTO, cercano,lejano);

Permite indicar a OpenGL la manera en que proyectaremos los gráficos en la ventana que se ha creado. Usamos la función **glOrtho(),** creando el volumen de visualización. Todo lo que esté dentro de este volumen será proyectado de la forma más simple, eliminando su coordenada Z, o de profundidad. Esta proyección es la llamada de ortográfica.

42.glMatrixMode(GL_MODELVIEW);

Esta función se usa para determinar qué pila de matrices (GL_PROJECTION, GL_MODELVIEW o GL_TEXTURE) se usará con las operaciones de matrices.

Alguna funciones para el manejo de atributos:

- **glClearColor (float a,float b,float c, flota alpha)**

  Esta es algo genérica y se refiere al color con el cual debe de "resetearse" el frame buffer cada vez que redibujemos toda la escena de nuevo. En este caso el "**fondo**" de nuestra ventana será como el fijado por esta función en el frame buffer. El cuarto parámetro de la función se refiere al valor de alpha en cuanto al color. Veremos más adelante que el valor de alpha permite variar el **grado de transparencia** de un objeto.

- **glColor3f(float a,float b,float c)**

  En este caso definimos que todo lo que se dibuje desde este momento será del color especificado. Recordar que el orden de parámetros es Red, Green, Blue (modelo RGB).

- **glPointSize(grosor):**

Con esta llamada definimos que cada uno de nuestros puntos deberá tener un grosor de píxel en el momento de ser trasladado a pantalla.

- **glWidthLine(float grosor)**

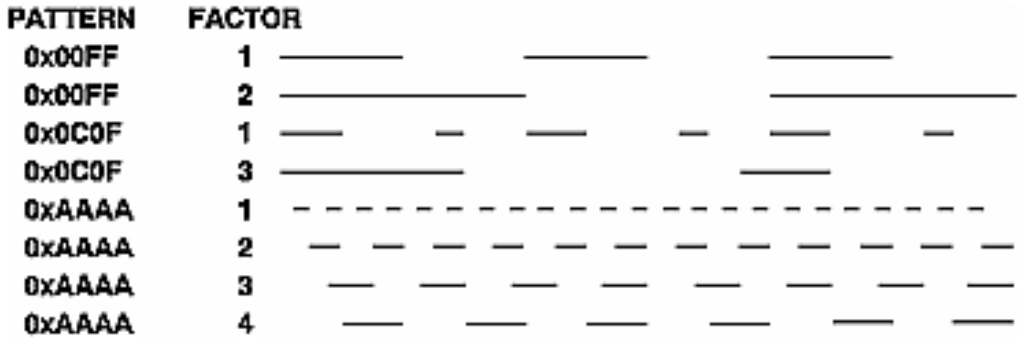
  Permite especificar líneas con el **grosor** deseado.

- **glLineStipple(int factor, GLushort patron)**

  Permite definir el **patron** de punteado (en las líneas); **patron** es una serie de 16 bits (0 y 1), el cual se repite constantemente. El patrón puede ser escalado por un **factor** deseado. Una vez definido el patrón, esta se activa con el comando **glEnable**(GL_LINE_STIPPLE) y se desactiva con **glDisable**().

  Por ejemplo:

  ```
  glLineStipple(FACTOR, PATTERN);
  glEnable(GL_LINE_STIPPLE);
  ```



- **glNormal3f(float a,float b,float c)**

  Cuando operemos con **luces** veremos que para cada cara de un polígono hay que asociar un vector o **normal**. Esta función define como normal a partir de este momento a un vector definido desde el origen con dirección positiva de las X. El orden de los parámetros es X, Y, Z.

- **glMaterialfv(GL_FRONT,GL_DIFFUSE,blanco)**

  Por último y también referido al tema de las **luces,** cada objeto será de un material diferente de manera que los reflejos que en el se produzcan debidos a las luces de nuestra escena variaran según su rugosidad, transparencia, capacidad de reflexión; en este caso definimos que todas las caras principales FRONT, son las caras que se ven del polígono) de los objetos dibujados a partir de ahora tendrán una componente difusa de color blanco (asumiendo que el parámetro blanco es un vector de reales que define este color). La componente difusa de un material define qué color tiene la luz que este propaga en todas las direcciones cuando sobre el incide un rayo luminoso. En este caso se vería luz blanca emanando del objeto (s) dibujado (s) a partir de ahora, antes hay que definir luces en nuestra escena y activarlas.

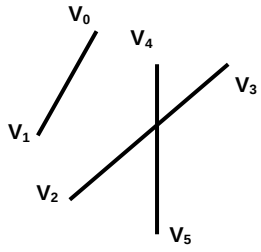
En cuanto a los colores, Podemos trabajar con la convención RGBA, es decir, grado de rojo, verde, azul y transparencia. Los valores para cada componente se encontraran siempre dentro del intervalo [0,1]. Si nos excedemos o nos quedamos cortos numéricamente hablando, **OpenGL** adoptara como valor 1 o 0 según sea el caso. Es decir, que redondea automáticamente, aunque hay que evitarle cálculos innecesarios y prever valores correctos. En el caso del valor para **alpha**, 0 significa transparencia total y de aquí podemos usar cualquier valor hasta 1, objeto totalmente opaco.

10. MANEJO DE PRIMITIVAS GRÁFICAS PARA POLILÍNEAS

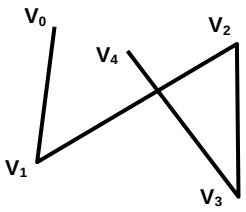
Una vez definidos los vértices con la que trabajaremos, estas pueden ser tratadas y representadas de muchas maneras, dependiendo de los que deseemos ver o mostrar; para ello OpenGL dispone de los siguientes parámetros para definir puntos, segmentos, triángulos, polígonos y mallados:

- **GL_POINTS**: para que todos los vértices indicados entre ambas funciones se dibujen por separado a modo de puntos libres.
- **GL_LINES**: cada dos vértices definidos, se traza automáticamente una línea que los une.
- **GL_POLYGON**: se unen todos los vértices formando un polígono.
- **GL_QUADS**: cada 4 vértices se unen para formar un cuadrilátero.
- **GL_TRIANGLES**: cada 3 vértices se unen para formar un triángulo.
- **GL_TRIANGLE_STRIP**: crea un triángulo con los 3 primeros vértices, entonces sucesivamente crea un nuevo triángulo unido al anterior usando los dos últimos vértices que se han definido y el actual.
- **GL_TRIANGLE_FAN**: Abanico de triángulos.
- **GL_LINE_STRIP**: Series de líneas conectadas.
- **GL_LINE_LOOP**: Mismo que el anterior, pero el último y el primer vértice cierran en bucle.
- **GL_QUAD_STRIP**: igual que en TRIANGLE_STRIP pero con cuadriláteros (véase la orientación de los vértices).

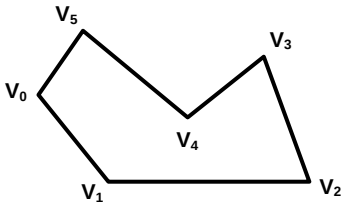
Cualquier otra representación gráfica, se obtiene combinando apropiadamente uno o más atributos de la lista de arriba. Consulte el manual de especificaciones del OpenGL (ver la bibliografía).



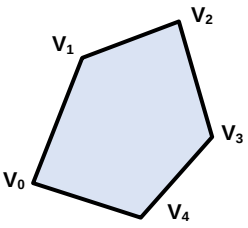
GL_LINES



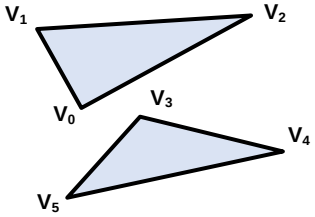
GL_LINE_STRIP



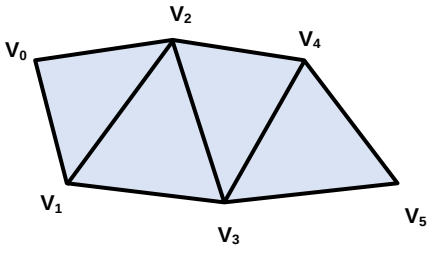
GL_LINE_LOOP



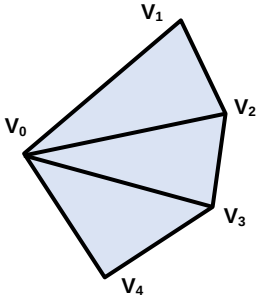
GL_POLYGON



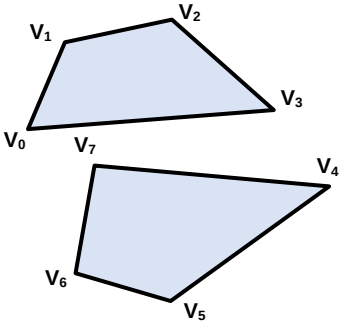
GL_TRIANGLES



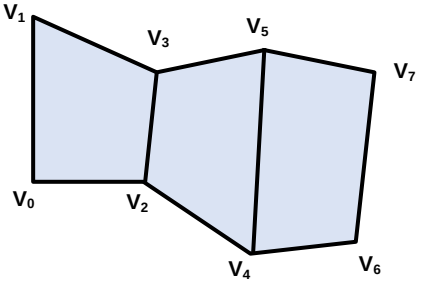
GL_TRIANGLE_STRIP



GL_TRIANGLE_FAN



GL_QUADS



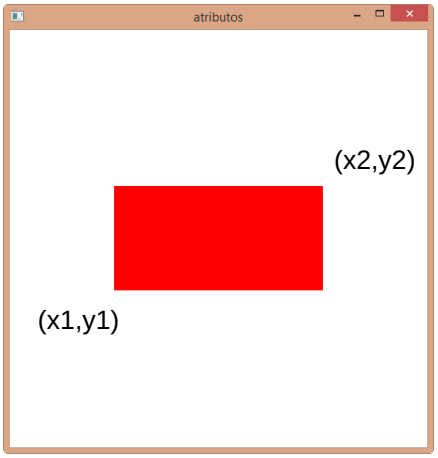
GL_QUAD_STRIP

11. ALGUNAS PRIMITIVAS BIDIMENSIONALES

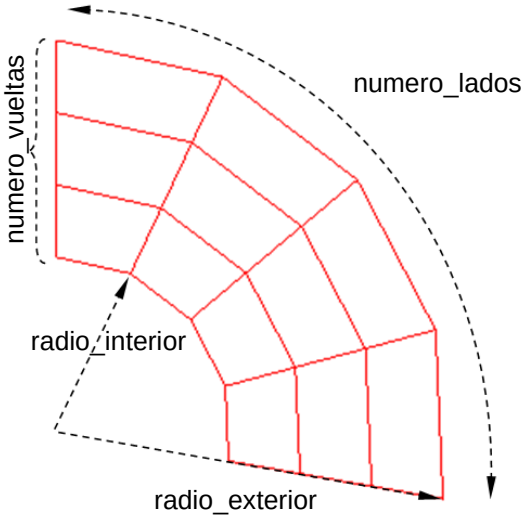
OpenGL dispone de funciones que permiten obtener figuras planares, por ejemplo:

- **void glRectf(GLfloat x1, GLfloat y1, GLfloat x2, GLfloat y2)**:

permite dibujar un rectángulo plano definido por el vértice inferior izquierdo (x1,y1) y el vértice superior derecho (x2,y2). También puede recibir variables enteras o un arreglo de vértices (véase las referencias).



Se puede dibujar discos, discos huecos y discos parciales, para esto OpenGL emplea los objetos cuádricos, las cuales emplean modeladores cuadráticos para una mejor visualización; estas funciones la provee la librería **glu**:



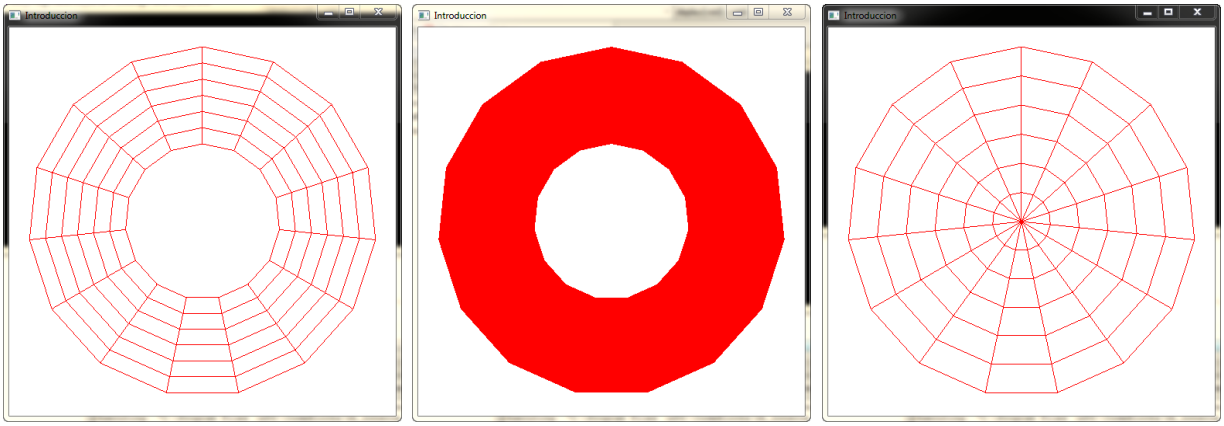
- **void gluDisk(GLUquadricObj *pt, GLdouble rinterior, GLdouble rexterior, GLint nlados, GLint nvueltas)**:

  permite dibujar un disco centrado sobre el plano XY, si **rinterior=0** entonces dibuja un disco sólido; el borde se representa con **nlados** (número de lados del disco), mientras que **nvueltas** proporciona el número de anillos que se genera radialmente; finalmente ***pu** es una variable de tipo **Quadric**.

  El siguiente código dibuja un disco hueco:

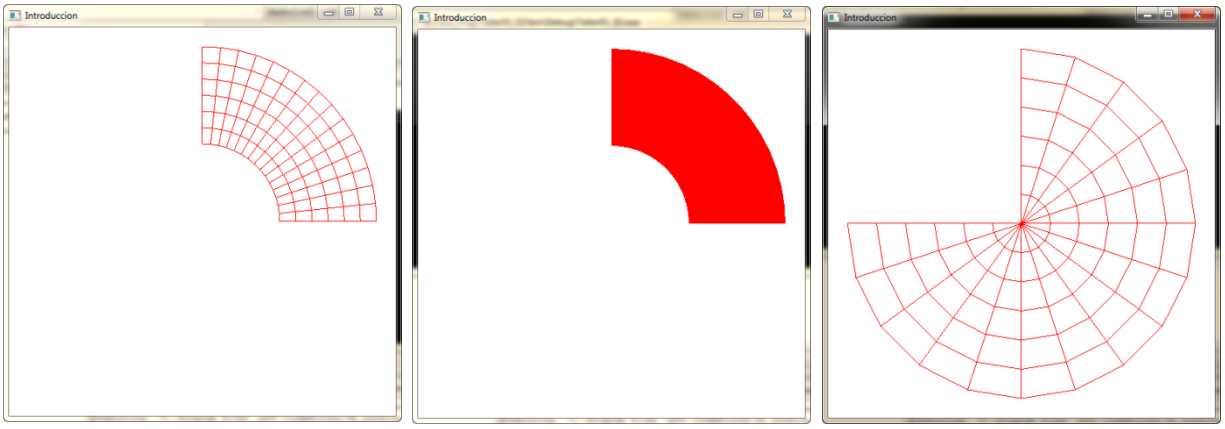
```
GLUquadricObj *pt;
pt=gluNewQuadric();

// esta orden especifica como va a ser renderizado:
gluQuadricDrawStyle(pt,GLU_LINE);
glColor3f (1.0, 0.0, 0.0);
// Dibuja Discos
gluDisk(pt, 4, 9, 15, 6);
```



- **void gluPartialDisk(GLUquadricObj *pt, GLdouble rinterior, GLdouble rexterior, GLint nlados, GLint nvueltas, GLdouble ang_ini, GLdouble ang_fin)**:
  dibuja un disco cuádrico parcial centrado sobre el plano XY, de manera similar a la anterior función; **ang_ini** es al ángulo de partida del disco, siendo 0º en la parte superior y 90º a la derecha, mientras que **ang_fin** especifica la porción final del disco.
  El siguiente código dibuja un sector de disco hueco:

```
GLUquadricObj *pt;
pt=gluNewQuadric();
// renderizar con lineas
gluQuadricDrawStyle(pt,GLU_LINE);
glColor3f (1.0, 0.0, 0.0);
// Dibuja Disco parcial
gluPartialDisk(pt, 4, 9, 15, 6,0,90);
```



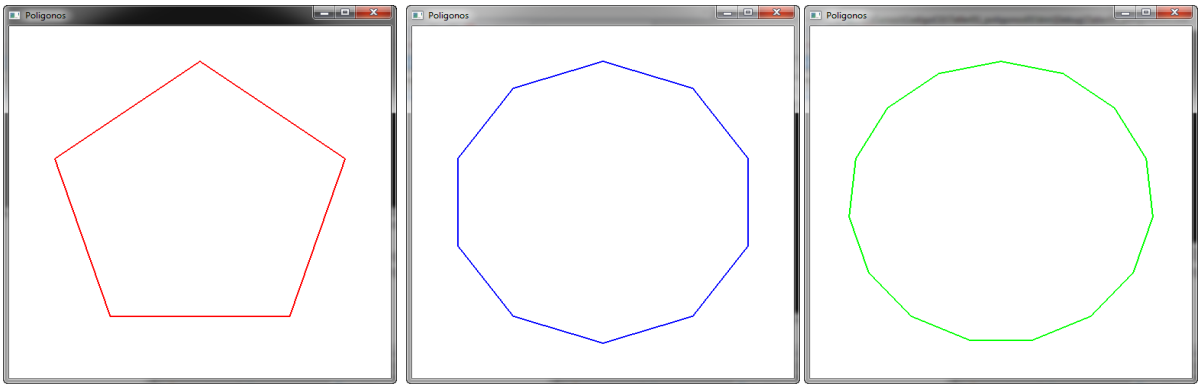
Más adelante, abordaremos otras primitivas del tipo 3D empleando los objetos cuádricos que hemos comentado.

12. EJERCICIOS PROPUESTOS

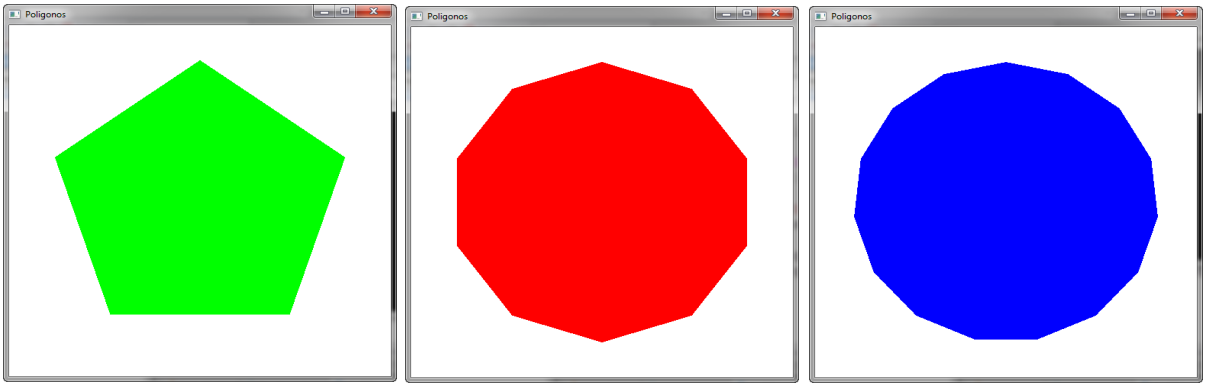
Ejercicio 01:

En el programa ejemplo realice las siguientes modificaciones:

1. En vez de puntos, ahora trabaje con líneas (GL_LINES), con polilíneas (GL_LINE_STRIP), con polilínea cerrada (GL_LINE_LOOP), y ejecute el programa.
2. Use diferentes colores para los puntos de la figura y ejecute el programa.
3. Modifique en glShadeModel para GL_SMOOTH y observe el resultado.
4. Emplee GL_POLYGON y observe las diferencias
5. Diseñe un objeto cualquiera, el que sea de su agrado.



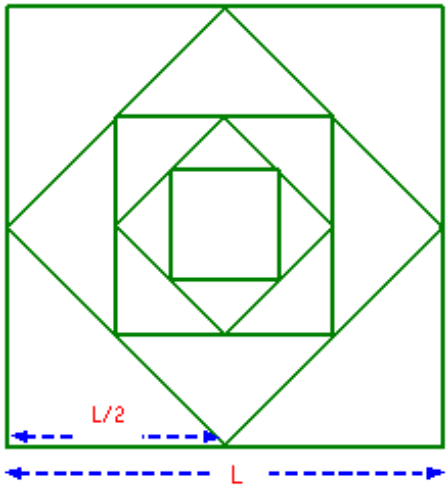
Con relleno



Ejercicio 02:

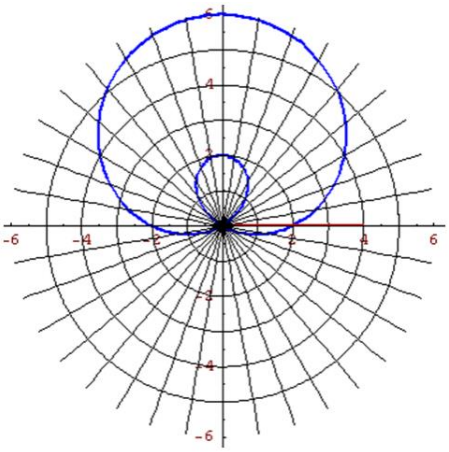
Diseñar un programa con OpenGL que permita leer por teclado:

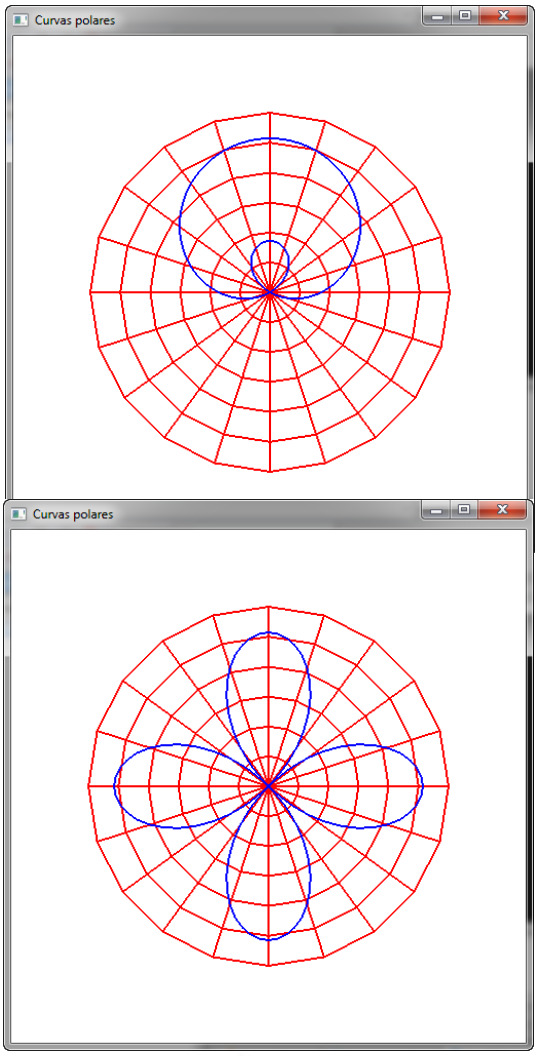
L: Longitud del lado del cuadrado mayor.
N: Número de cuadrados como se muestra en el gráfico.
Se deberá graficar empezando por el cuadrado más grande al más pequeño. Si gusta puede centrar sus gráficos en el origen de coordenadas; y por último tome sus precauciones definiendo las dimensiones de la ventana inicial de forma apropiada; maneje dimensiones relativamente grandes, tanto como permita la resolución de su monitor. El código deberá mostrar una estructura repetitiva para graficar los cuadrados.

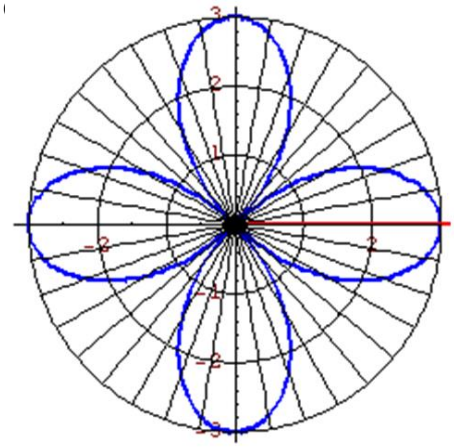


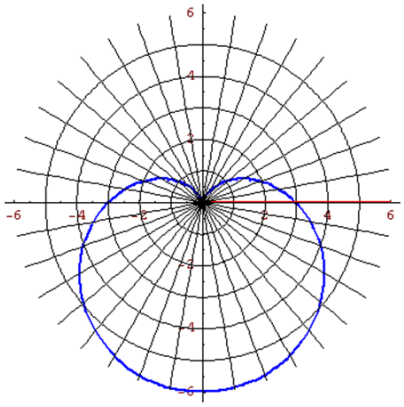
Ejercicio 03:

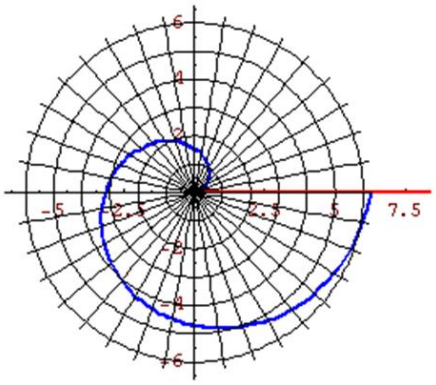
Hacer un programa usando OpenGL para generar las siguientes curvas cerradas (ignore o prescinda las etiquetas numéricas) según los gráficos sugeridos:











Cardioide: $f(\theta) = 3 - 3 \operatorname{sen} \theta$

Espiral (de Fermat): $f(\theta) = \theta$

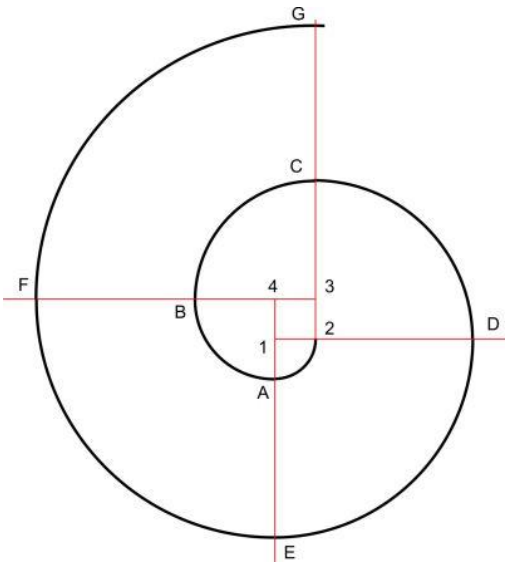
a. $f(\theta) = \operatorname{sen} \theta + \operatorname{sen}^3(2.5\,\theta)\ ;\ \theta \in [0; 4\pi]$

b. $f(\theta) = -N \cos\theta + \cos(5\,\theta)\ ;N \in \{\pm 1; \pm 2; \pm 3; \pm 4; \pm 5\}$

c. $f(\theta) = e^{\cos\theta} - 2\cos(4\,\theta)$

Ejercicio 04:

Diseñe una aplicación gráfica para dibujar la espiral de 4 centros, de la siguiente manera: trazamos el cuadrado de vértices 1,2,3,4. Procedemos prolongando sus lados, con el fin de limitar los arcos y dibujamos de la siguiente manera:
Empezamos tomando el arco con centro en 1 y radio 1-2 trazamos el arco 2A. Con centro en 4 trazamos el arco AB. Con centro en 3 trazamos el arco BC. Con centro en 2 trazamos el arco CD. Si deseamos más arcos repetiremos el ciclo.



13. REFERENCIAS

- Sellers G, Wright R, & Haemel N. (2016). **OpenGL Superbible** (7ma edición). Indiana. Addison-Wesley.
- Shreiner D, Sellers G, Kessenich J. (2013). **OpenGL Programming Guide: The Official Guide to Learning OpenGL**. Michigan. Addison-Wesley.
- Mark J. Kilgard. (1996). **The OpenGL Utility Toolkit (GLUT) Programming Interface** (API Version 3). Silicon Graphics Inc.
- Mark Segal, Kurt Akeley. (1998). **The OpenGL Graphics System: A Specification** (Version 1.2). Silicon Graphics Inc.
- **Fast Light Toolkit**

  https://www.fltk.org/
- http://www.taringa.net/posts/ebooks-tutoriales/2449209/Maya-the-fundamentals-tutoriales-profecionales-maya-2009.html