Prácticas de Informática Gráfica

(Versión 11)

Dr. Arno Formella

http://www.ei.uvigo.es/~formella

Dra. Mª Victoria Luzón García

Universidad de Vigo

Departamento de Informática

E-32004 Ourense

17 de marzo de 2004

Índice

1. Revisiones del documento

Durante el transcurso del tiempo se ha realizado los siguientes cambios en el documento:

Versión 10 a Versión 11

- Existe un apartado breve sobre el uso de STL.
- Se ha puesto el `Makefile` para la compilación por defecto para un entorno sin Leda.

Versión 9 a Versión 10

- Se ha mejorado la versión en formato PDF.
- Unos fallos ortográficos menos.

Versión 8 a Versión 9

- Los ejemplos de programación (Section **??**) existen ahora también con estructuras de datos de STL (*standard template library*) como sustituto de Leda para estructuras simples.
- Unos fallos ortográficos menos.
- Se ha eliminado el fallo en la parte del doble búfer (Section 14.2).

Se ha actualizado las bases para la entrega de trabajos (Section 19.2).

Versión 7 a Versión 8

Se ha actualizado las bases para la entrega de trabajos (Section 19.2).

Versión 6 a Versión 7

- Mª Victoria Luzón García ya es doctora.
- Se ha añadido las bases para la entrega de trabajos (Section 19.2).

Versión 5 a Versión 6

- Se ha reorganizado las tareas para el curso 2001/2002.
- Se ha clasificado la sección como calcular el plano de un polígono tridimensional (Section 13) de una forma robusta y tolerante como extensión para dejar más tiempo para el desarrollo de trabajos creativos.

- Se ha eliminado el fallo en el dibujo de un polígono no-estrellado.
- Faltaba el factor de un medio en la fórmula para calcular el área de un polígono. (El fallo no afectaba el cálculo del vector normal porque se normalizaba de todos modos.)

Versión 4 a Versión 5

- Se ha cambiado un poco las tareas a partir de la semana 9.

Versión 3 a Versión 4

- Se ha cambiado un poco las tareas a partir de la semana 7.

Versión 2 a Versión 3

- Manejo de eventos (Section 7) y transformaciones (Section 8) se introduce ahora antes
- `Leda` (Section 9) se introduce ahora más tarde

Versión 1 a Versión 2

- Añadido (Section 4.3) una descripción del `make clean`
- Se usa (Section 5.5) `glOrtho` en vez de `gluOrtho2D`

2. Objetivo

nuestro objetivo es:

- visualizar entornos tridimensionales
- con herramientas avanzadas

Debido a la complejidad del problema, las prácticas no siguen un camino simplemente derecho, sino que muchas veces hay que tener en cuenta los los aspectos siguientes:

- se usan cosas sin saber cómo están relacionadas con la tarea actual, en un determinado momento
- se entiende algunos pasos de la programación sólo superficialmente; se aclararán posteriormente
- hay que hacer muchas veces marcha atrás y rehacer varios ejercicios para captar los detalles

3. Organización

Cada alumn@ es responsable de hacer sus propias copias de los datos generados durante las prácticas. No se puede esperar que algún fichero quede guardado hasta la semana siguiente.
Cada sesión de prácticas está dividida en tres partes:

preparación *Antes* de venir a las prácticas se lee el material con las tareas pendientes y se desarrolla un plan para realizar durante las dos horas delante del ordenador.

prácticas Se dedican a programar las tareas, escuchar las instrucciones y resolver los posible problemas o dudas.

trabajo después *Después* de las prácticas se lee el material obtenido, sin olvidarse de repetir, en caso que sea necesario el material de las prácticas anteriores, ya que muchas veces podemos aprovechar del contenido de clases anteriores.

La nota final de las prácticas se obtiene mediante un examen a final de curso (junto con el examen de teoría), es decir, no es necesario hacer ningún proyecto o trabajo adicional.
Vale mucho más un pequeño, pero constante, esfuerzo semana a semana, que intentar aprender todo un dia antes del examen.
Recomendamos: usar un pequeño librito (llamado *Mis conocimientos de informática gráfica*) que se llena durante el curso con apuntes *propios*. Nuestra observación es: cuanto menos contiene el librito al final del curso y cuantas menos veces se ha releído su contenido tanto peor será la nota a final de curso.
Bastante información (sobre todo los manuales) está escrita en inglés, sin embargo, el nivel necesario para entenderlo no es muy alto y siempre existe el camino fácil: ¡se pregunta! si queda algo sin entender.
No está prohibido, sino explícitamente deseado, que se trabaje en pequeños grupos ayudándose mutuamente. (Aunque el examen final cada un@ tiene que hacerlo por sí mism@).

4. Software

Usamos los siguientes paquetes de software:

Linux sistema operativo

KDE (o bién Gnome) sistema de entorno gráfico bajo XFree con ventanas

C++ compilador de la gama Gnu

OpenGL librería en C de visualización gráfica

Leda librería en C++ de estructuras de datos y algoritmos avanzados

STL (Como sustituto de Leda, se puede usar la librería estándar de plantillas (*standard template library*) llamado STL.)

Además usamos diferentes herramientas de desarrollo como:

- editores, según gusto personal (p.e. `emacs` o `vi`)
- una herramienta para actualizar ficheros según un conjunto de reglas implícitas y explícitas (`make`). Las reglas explícitas ya están dadas de antemano en un fichero llamado `Makefile`.

(Nota: si se quiere usar en vez de STL el Leda se puede usar el fichero `MakefileLEDA`, pero hay que renombrarlo a `Makefile`.)

4.1. Manuales

Todo los manuales importantes de software usados durante el curso están disponibles en los ordenadores en formato HTML, PostScript o PDF.
Este documento se entiende como guía adicional. No sustituye de ninguna manera a los manuales de los diferentes paquetes. Es imprescindible consultar a menudo los manuales y visitar la página acompañante en la güeb durante el curso. Por ejemplo, no se repiten las descripciones de las funciones y sus parámetros dadas en los manuales.
Para facilitar el uso de este documento, usamos los siguientes modos de escribir con colores:

- `comandos de linux`
- `secuencias de código fuente`
- `nombres o contenidos de ficheros`

4.2. Comienzo

Habilidades por aprender:

- entrar en Linux
- familizarse un poco con el entorno gráfico
- abrir el navegador y localizar los manuales
- abrir una “shell”
- experimentar con los siguientes comandos de Linux:
  `man, ls, cd, mcopy, cp, mv, mkdir`

- crearse un subdirectorio propio
- abrir el editor favorito
- familizarse con el editor
- escribir un programa (p.e., el famoso “hola mundo”)
- compilar el programa
- ejecutar el programa

4.3. `make`

La herramienta `make` nos facilita mucho la tarea de compilar un programa, sobre todo si el proyecto consiste en varios ficheros de código fuente y si se tiene que trabajar con librerías adicionales.
Hemos preparado un fichero `Makefile` que vamos a usar siempre junto con el `make` (es decir, aprovechamos la situación de que `make` usa por defecto un fichero llamado `Makefile` para encontrar sus reglas). No hay por qué entender el contenido de este fichero, pero tampoco está prohibido intentarlo.
Lo único que tenemos que hacer es:

> decir a `make` cuáles son los ficheros que queremos compilar para formar el programa. Listamos esos ficheros en un fichero llamado `objs.make` tal cual está: si hace falta añadimos líneas adecuadas.

Una vez actualizado el fichero `objs.make` se crea el ejecutable simplemente con el comando: `make`
Nuestro ejecutable siempre se llamará `ig` y se ejecuta con `./ig`
El `make` podemos usar también para borrar todos los ficheros que no necesitamos salvar, p.e., el ejecutable. Con un simple `make clean` nos borra todo los ficheros temporales y el ejecutable.

4.4. Salvar los datos

Para hacer la copia de seguridad de nuestros datos hace falta salvar todos los ficheros con código fuente (es decir, `*.cpp` y `*.h`), el `Makefile` y la lista de nombres de ficheros (es decir, `objs.make`). La lista es fácil de volver a crear.

5. OpenGL (primeros pasos)

5.1. Características

- OpenGL es una implementación de la cadena de reproducción (“rendering pipeline”) que realiza la tarea de visualizar un modelo de un mundo virtual en una pantalla
- OpenGL trabaja en el espacio tridimensional
- OpenGL es independiente de la plataforma
- aunque originalmente fué desarrollado para el lenguaje de programación C, existe OpenGL para varios lenguajes
- OpenGL está diseñado para que diferentes partes estén realizadas directamente en hardware (especialmente en sistemas de Silicon Graphics)
- se puede modificar el estado y el flujo de datos en la cadena de reproducción (“rendering pipeline”) de OpenGL
- OpenGL dispone de diferentes tipos de objetos simples (p.e., puntos, líneas y polígonos), los cuales pueden tener varias aparencias (p.e., líneas interrumpidas o polígonos rellenados)
- OpenGL permite el uso de texturas
- OpenGL usa diferentes modos de visualizar entornos tridimensionales (p.e., alámbrico, “flat shading”, “Gouraud shading”)
- para usar OpenGL se necesita normalmente, además del compilador, unas librerías con funciones auxiliares que conectan OpenGL al sistema gráfico realmente usado
- dichas librerías ayudan en la implementación de animaciones simples

5.2. Uso de C con Mesa

Las funciones de OpenGL como las usamos en C bajo Linux con su implementación Mesa, y de las librerías auxiliares siguen una nomenclatura bastante intuitiva:

- las funciones de OpenGL tienen el prefijo `gl` (viene de graphics library)
- las funciones de las librerías auxiliares tienen el prefijo `glu` y `glut` (viene de graphics library utilities y graphics library utility tools)
- el sufijo de muchas funciones de OpenGL indica el número de parámetros esperados, así como sus tipos (en el siguiente texto, el sufijo `*` indica que no está especificada exactamente con qué tipo de datos se está trabajando. En un programa dicho asterisco tiene que ser sustituido por el sufijo adecuado)

- las constantes definidas en los ficheros de cabecera (“header files”) suelen tener el prefijo `GL_`

Ejemplo:

```
glVertex3d(parametro0, parametro1, parametro2)
```

observando que:

- `gl` indica que es una función de OpenGL
- `Vertex` es el nombre de la función, que en este caso tiene que ver algo con vertices o puntos
- `3` indica que hay que pasar tres parámetros a la función
- `d` indica que los parametros deben ser del tipo `double` (es decir, números flotantes de doble precisión)

5.3. Máquina de estados

- OpenGL tiene un comportamiento similar al de una máquina de estados.
- En todo momento, OpenGL mantiene un conjunto de variables que representan el estado actual.
- Podemos cambiar el valor de cada una de estas variables mediante llamadas a funciones de OpenGL.
- Podemos “obtener” los valores de estas variables mediante llamadas a funciones (en la mayoría de los casos con prefijo `glGet`).

Ejemplo: El color, p.e., es una variable de estado. Cuando se especifica un color todos los objetos son dibujados utilizando ese color hasta que se indique un color distinto.
Existen también variables de estado que hacen referencia a modos que pueden ser habilitados (función `glEnable()`) o deshabilitados (función `glDisable()`).
Todas las variables de estado tienen valores actuales que podemos consultar con diferentes funciones simples (p.e., `glGetBooleanv()`, `glGetDoublev()`, `glGetFloatv()` o funciones mas específicas (p.e., `glGetLight()` o `glGetError()`).
Para facilitar el uso de estados, OpenGL dispone de varias pilas que se puede usar para salvar estados (poner en la pila (“push”)) y restaurar estados anteriores (sacar de la pila (“pop”)).

5.4. Manejo de la ventana

El primer programa `ig01.cpp` usa casi el mínimo número de funciones para realizar un simple programa con OpenGL: se abre una ventana y se llena con un color.
De la librería auxiliar se usa:

`glutInit()`
`glutInitDisplayMode()`
`glutCreateWindow()`
`glutDisplayFunc()`
`glutMainLoop()`

El fondo de la ventana se dibuja en un color (Section 5.7) usando las funciones de OpenGL:

`glClearColor()`
`glClear()`

Se puede usar las funciones

`glutInitWindowPosition()`
`glutInitWindowSize()`

para fijar la ventana con cierto tamaño en la pantalla.

5.5. Colocar un sistema de coordenadas

Al principio vamos a usar una proyección muy sencilla para visualizar nuestro mundo virtual: la proyección ortogonal que simplemente ignora la coordenada *z* de un punto *(x,y,z)* para obtener el punto *(x,y)* en el plano de visualización (es decir, en el plano de la ventana).

`gluOrtho()`

Más adelante colocamos una cámara virtual (Section 15) con perspectiva.

5.6. Puntos

OpenGL es capaz de visualizar puntos, incluso puntos que, matemáticamente no tienen ninguna extensión en ninguna dimensión. El problema se resuelve dibujando los puntos directamente como un conjunto de pixeles en la ventana.
Siempre que se quiera dibujar un objeto de OpenGL (en este caso un conjunto de puntos, después veremos segmentos (Section 5.8) y polígonos (Section 6.3)) hay que decírselo de antemano con la función

`glBegin()`

Un conjunto de puntos se define entonces con llamadas a

`glVertex*()`

y se termina el trabajo con

`glEnd()`

Anota que, entre una llamada a `glBegin()` y su correspondiente llamada a `glEnd()`, se puede llamar sólamente a un subconjunto de todas las funciones de OpenGL.
Se puede cambiar el tamaño, medido en pixeles, de los puntos posteriormente dibujados con

`glPointSize()`

Para que nos dibuje (el mago dentro del ordenador) el contenido de la ventana tenemos que llamar a la función

```
glFlush()
```

y en ciertas circunstancias también a

```
glFinish()
```

5.7. Color

OpenGL trabaja con el espacio de colores llamado RGB (Red Green Blue, o bién rojo verde azul). Así lo especificamos (Section 5.4) al haber llamado a `glutInitDisplayMode()`. Cada color está definido por tres valores entre 0 y 1, ambos incluídos. La siguiente tabla contiene las definiciones de los colores mas llamativos:

color	componente **rojo** (red)	componente **verde** (green)	componente **azul** (blue)
rojo	1.0	0.0	0.0
verde	0.0	1.0	0.0
azul	0.0	0.0	1.0
blanco	1.0	1.0	1.0
negro	0.0	0.0	0.0
amarillo	1.0	1.0	0.0
magenta	1.0	0.0	1.0
cyan	0.0	1.0	1.0

Se cambia el estado de OpenGL respeto al color que se va a usar con:

```
glColor*()
```

5.8. Segmentos

OpenGL es capaz de visualizar segmentos, o cunjuntos de segmentos, incluso aquellos que, matemáticamente, no tienen ninguna extensión respeto a su anchura. El problema se resuelve dibujando los segmentos directamente como un conjunto de pixeles en la ventana.
Siempre cuando se quiere dibujar un objeto de OpenGL (en este caso unos segmentos, antes vimos puntos (Section 5.6) y después veremos polígonos (Section 6.3)) hay que decírselo de antemano con la función

```
glBegin()
```

Existen tres tipos de conjuntos de segmentos: segmentos separados, segmentos unidos (es decir, una polilínea) y segmentos cíclicamente unidos (es decir, una polilínea cerrada). El conjunto de segmentos se define mediante un conjunto de puntos que definen las esquinas de los segmentos con llamadas a

```
glVertex*()
```

y se termina el trabajo con

```
glEnd()
```

Igual que vimos con los puntos (Section 5.6), se puede variar la anchura, medida en pixeles, de las líneas con

`glLineWidth()`

Además se pueden dibujar las líneas con patrones. Los patrones permitidos se basan en cadenas de 16 bits. Siempre que un bit del patrón sea igual a 1 se dibuja dicho pixel, si es igual a 0 no se dibuja. Las polilíneas se dibujan desde su primer hasta su último punto, utilizando los bits del patrón desde el menos significativo hasta el más significativo y después se empieza de nuevo.

Ejemplo:

El patrón `0111111011101101` = `0x7EED` genera una línea discontinua cuyos trocitos crecen en longitud.

Se puede dilatar el patrón con un factor entero, así cada bit del patrón controla más de un pixel de la línea.

`glLineStipple()`

Antes de poder ver líneas controladas por patrones hay que habilitar el modo adecuado:

`glEnable()`

que siempre se puede deshabilitar de nuevo con

`glDisable()`

6. Polígonos

A primera vista, parece que los polígonos no deberían provocar ninguna dificultad, ya que cada uno sabe intuitivamente lo que es un polígono. Sin embargo, una vez que se intenta programar una aplicación gráfica, se notará sin duda que no es tan sencillo, sobre todo, cuando se intentan escribir programas correctos y robustos.

6.1. Definición

Un polígono es

1. una lista de puntos
2. que se encuentran en un solo plano
3. cada dos puntos consecutivos de la lista forman los extremos de un segmento (el último punto de la lista lo hace con el primero).

Es decir, el borde de un polígono implícitamente define una polilínea cerrada en un plano.
Es conveniente separar bién las diferentes partes de un polígono:

1. los puntos que forman las esquinas

2. los segmentos sin los puntos extremos
3. el interior del polígono, es decir, todos los puntos que pertenecen al polígono pero que son ni esquinas ni pertenecen a los segmentos del borde

Para trabajar matemáticamente bién con polígonos como conjuntos de puntos a los cuales se les pueda aplicar operaciones de conjuntos como intersección, diferencia, unión o complemento, hay que introducir el concepto de regularización. Pero no vamos a profundizar en este tema.
Lo que sí necesitamos es aclarar el asunto para poder programar correctamente con OpenGL.
La definición del polígono (Section 6.1) tiene sus problemillas cuando la aplicamos a diferentes dimensiones:

bidimensional:
- Todos los puntos de la lista se encuentran obviamente en un plano.
- Entonces, si la lista contiene sólo puntos colineales, sí, definen un polígono, aún su área debe ser cero (vamos a ver después lo que es).
- Así también un solo punto se podría ver como polígono, aunque por razones de las operaciones de conjuntos mencionadas arriba no es conveniente tratar un solo punto como polígono.
- ¿Cómo calcular el área de un polígono?

tridimensional:
- La lista tiene que contener por lo menos tres puntos que no son colineales.
- ¿Existen polígonos tridimensionales con área cero?
- ¿Cómo calcular el plano en el cual se encuentran los puntos?
- ¿Cómo calcular el área de un polígono?

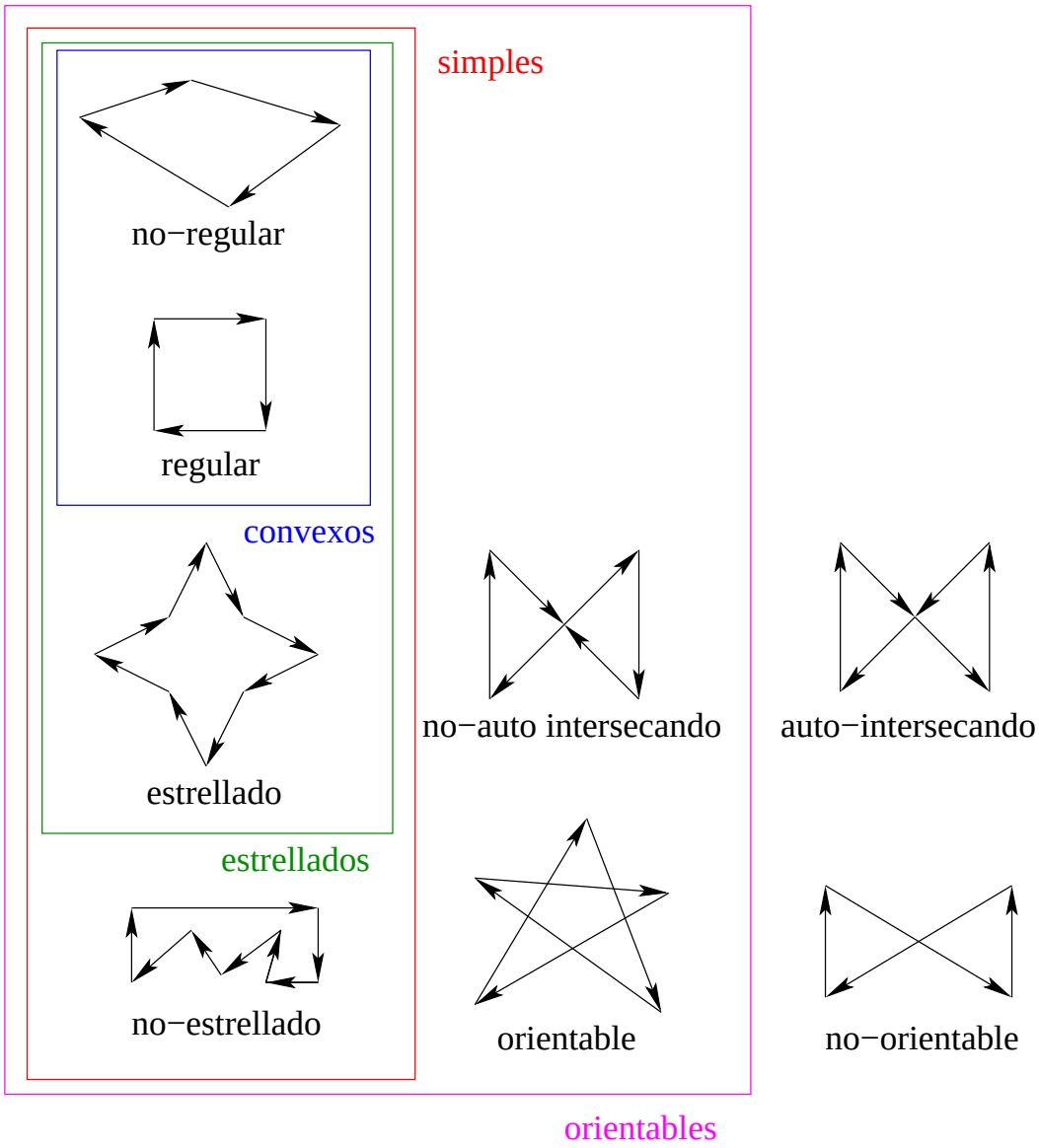
Una diferencia más entre polígonos bi- y tridimensionales es: en el espacio los polígonos tienen dos caras.

6.2. Clasificación

Asumimos primero que dos puntos consecutivos de la lista de puntos del polígono no son iguales, así tampoco el primero igual que el último. Tales polígonos con puntos múltiples son fáciles de detectar y de eliminar (si es necesario).
Luego podemos clasificar los polígonos (como cualquier otro bicho), por lo menos, según la tabla:

clase	**especie**	**variante**
simple	convexo	regulares
		no-regulares
	no-convexo	estrellado
		no-estrellado
débilmente simple	auto-intersecando	
	no-auto-intersecando	
no-simple	orientable	
	no-orientable	

La gráfica da ejemplos de diferentes polígonos:



¿Por qué tanto rollo?

6.3. Polígonos de OpenGL

OpenGL maneja polígonos correctamente siempre y cuando sean *simples y convexos*. Si ese no es el caso, OpenGL dibuja cosas raras.
Además en algunas ocasiónes se quiere especificar el vector normal del plano en el cual se encuentra el polígono. Dicho vector normal se necesita p.e. para algoritmos de visualización avanzada (“Phong shading”).
Cuando se escriben aplicaciones gráficas nos enfrentamos con el problema siguiente: desde alguna fuente “vienen” listas de puntos (o bién de otro programa, o bién de un fichero, o bién de

modo interactivo) y hay que interpretar las listas como polígonos, es decir, hay que verificar si las listas cumplen la definición de especificar polígonos simples y convexos. Si ese no es el caso, a lo mejor se pueden “corregir” las listas. (Programas con tal propriedades se llama robustos y tolerantes.)
Entonces (en el caso de mera visualización con OpenGL):

1. Se puede eliminar puntos múltiples consecutivos en la lista.
2. Se puede intentar calcular *el* plano (Section 13) en el cual se encuentran más o menos los puntos del polígono (eso no es tam trivial).
3. En dicho plano, es decir, una vez corregidos los puntos hasta que se encuentren exactamente en el plano, se puede verificar si el polígono es simple y convexo (Section 6.6) (eso es algo bastante fácil).
4. Si no es así, se podría subdividir el polígono en partes simples y convexos para seguir trabajando después con las partes sin problemas (este paso no lo vamos a realizar en estas prácticas).

Antes de dedicarnos a los detalles, dibujamos polígonos con OpenGL asumiendo que la lista de puntos cumple la definición (Section 6.1).
Siempre que se quiera dibujar un objeto de OpenGL (en este caso unos polígonos, antes vimos puntos (Section 5.6) y segmentos (Section 5.8)), hay que decírselo de antemano con la función

```
glBegin()
```

La lista de puntos se define con consecutivas llamadas a

```
glVertex*()
```

y se termina el trabajo con

```
glEnd()
```

En lugar de dibujar polígonos rellenados, OpenGL puede dibujar, o bién solo las esquinas o bién solo los segmentos del borde. Eso se realiza con la función

```
glPolygonMode()
```

a la cual hay que pasar también cuál de las dos posibles caras (Section 11) del polígono se quiere pintar.
Además se pueden llenar los polígonos con patrones que no detallamos por el momento.

```
glEnable()
glPolygonStipple()
```

6.4. Simple

Asumimos un polígono bidimensional sin puntos múltiples consecutivos.
Un polígono es *simple* siempre y cuando el grado de cada vértice del grafo inducido por los segmentos del borde del polígono sea igual a dos.
Dicho grafo se obtiene así:

1. Se calcula todos los puntos de intersección entre dos segementos del borde (entonces todas las esquinas estarán entre ellos).
2. Estos puntos representan los vértices del grafo.
3. Se añade al grafo una arista entre dos nodos siempre y cuando los puntos correspondientes estén unidos directamente por un trozo de segmento.

En consecuencia, si permitiésemos que un punto sólo forme un polígono, no formaría un polígono simple.
Una explicación intuitiva de simple es:

- imagínate el borde del polígono como una cuerda colocada encima de la mesa (o vete a buscar una ...)
- pincha con un clavo (o un bolí) en un punto interior del polígono
- tira a la cuerda y observa *cuantas veces* le da la vuelta al clavo (bolí)
- si independientemente de la selección del punto interior siempre observas solo una vuelta, entonces, el polígono es simple.

Para comprobar si un polígono es simple o no, podemos aplicar p.e. el siguiente algoritmo sencillo:

1. Calculamos todos los puntos de intersección entre todas las parejas posibles de dos segmentos.
2. Construímos el grafo inducido.
3. Comprobamos si cada vértice tiene grado igual a dos.

Debido al paso uno, dicho algoritmo tendría un tiempo de ejecución inevitablemente cuadrático en el número de esquinas del polígono.
Existen algoritmos (basados en el paradigma de “barrer el plano”) que realizan el test en un tiempo de ejecución del orden n log n, siendo n el número de esquinas del polígono. (Para recordar: si el algoritmo sencillo necesitaría una hora para comprobar un polígono con un millon de puntos, el algoritmo avanzado lo haría en menos de una décima de segundo.)
Como veremos en breve, no necesitamos ninguno de ellos...

6.5. Convexo

Asumimos un polígono bidimensional sin puntos múltiples consecutivos.
Un polígono es *convexo* siempre y cuando cumpla una de las siguientes condiciones:

- Para cualesquiera dos puntos del interior del polígono, todos los puntos del segmento definido por los dos puntos también se encuentren en el interior.

- Para cualquier segmento del borde del polígono, todas las esquinas qué no son aquellas que definen el segmento están siempre a la derecha (o siempre a la izquierda) de la recta definida por el segmento.
- El polígono es simple y todos los ángulos entre dos segmentos adyacentes del borde del polígono son menor o igual (o son mayor o igual) de 180 grados (es decir, *pi* si medimos en radianes).

Se puede comprobar formalmente que las tres condiciones son equivalentes.
La primera condición es difícil de comprobar programando, porque hay un número infinito de puntos en el interior de un polígono. La condición sirve más bién en comprobaciones matemáticas.
La segunda condición sí se puede programar. Resultaría en un algoritmo con tiempo de ejecución cuadrático en el número de esquinas, porque hay que hacer comprobaciones para cada segmento con casi todas las demás esquinas.
La tercera condición también se puede programar. La comprobación de los ángulos nos llevaría un tiempo lineal en el número n de esquinas. Sin embargo, necesitamos un tiempo en el orden n log n para comprobar si el polígono adicionalmente es simple (o cuadrático si nos conformamos con el algoritmo simple) como vimos arriba (Section 6.4).
Pues, ¿hay algo mejor? (¿e incluso más sencillo de programar?)

6.6. Simple y convexo

Asumimos un polígono bidimensional sin puntos múltiples consecutivos.
Fijémonos en la esquina con la coordenada *x* mínima (si hay más de una, en aquella que también tiene la coordenada *y* mínima).
Si miramos ahora todos los demás puntos en el orden dado por la lista vemos que en un polígono simple y convexo: La secuencia de las coordenadas *x* de todos los puntos tiene como mucho *un* máximo. La misma condición tiene que cumplirse para la coordenada *y*.
¡Vaya que fácil! Sólamente hay que tener en cuenta: si todos los puntos son colineales, también se cumplen esas condiciones, pero no consideramos el polígono simple en todos los casos.
Entonces el algoritmo de comprobación de si un polígono es simple y convexo es sencillo de programar y tiene un tiempo de ejecución lineal en el número de esquinas. Lo vamos a hacer una vez conociendo un poco Leda (Section 9).
Para recordar: si se usan los algoritmos (sencillos) que comprueban las condiciónes simple y convexo por separado y si se necesitaría una hora para comprobar un polígono con un millon de puntos, este último algoritmo lo haría en unos pocos milisegundos.
Además, el último algoritmo se comporta bién con los números flotantes que se usa en los procesadores. Calcular puntos de intersección o comparar ángulos entre segmentos es una verdadera aventura, si se usan dichos números flotantes.

7. Eventos

Sistemas operativos modernos, especialmente en acciones relacionados con la interfaz de usuario, usan el concepto de eventos en la planificación y organización de tareas.
El sistema operativo Windows por ejemplo incorpora más de 800 eventos (repecticamente mensajes asociados) a los cuales un programa de aplicación puede reaccionar.
Nosotros consideramos sólamente muy pocos eventos y la reacción de nuestra aplicación a dichos eventos la realizamos con funciones llamadas “callback functions”.

7.1. Concepto de “callbacks”

En el primer programita (Section 5.4) ya definimos con una llamada a `glutDisplayFunc()` que OpenGL debería usar nuestra función `Display()` cuando hay que redibujar el contenido de la ventana.
Es decir, podemos registrar unas funciones que se preocupan de la realización del trabajo necesario siempre y cuando “alguien” (de nuevo, ¿el mago dentro del ordenador?) manda un mensaje a nuestro programa porque haya occurrido algún evento.
Con la función

`glutPostRedisplay()`

podemos mandar un mensaje a nuestra aplicación para indicar que hay que redibujar el contenido de la ventana, es decir, en un futuro próximo se llamará automáticamente a nuestra función `Display()`.
Además de los eventos “redibujar” usaremos los eventos “pulsado-una-tecla” y despues “terminado-un-temporizador” (Section 14).

7.2. Eventos del teclado

Con la función

`glutKeyboardFunc()`

podemos registrar una función que reaccione a eventos del teclado. Esa función tiene que aceptar tres parámetros, es decir, su prototipo en C es:

```
void KeyPress(unsigned char key_code, int xpos, int ypos);
```

Una posible reacción a una tecla es: terminar el programa si el usuario pulsa la tecla ‘q’:

```
static void KeyPress(
  unsigned char key_code,
  int xpos,
  int ypos
) {
  switch(key_code) {
```

```
      case 'q':
      case 'Q':
        glFinish(); // terminamos lo que queda de OpenGL
        exit(0);    // salimos del programa con exito
        break;
      default:
        break;
    }
    // mandamos un mensaje a nuestro programa
    // que redibuja el contenido de la ventana
    glutPostRedisplay();
  }
```

8. Transformaciones

8.1. Transformaciones afines

OpenGL permite con el uso de las tres funciones

`glTranslate*()`
`glScale*()`
`glRotate*()`

la realización de transformaciones afines simples, es decir, la traslación, el redimensionalmiento o escalado, y la rotación alrededor de un vector.
Si se quiere realizar otro tipo de transformaciones afines, p.e., la transformación de sesgo, hay que recurrir a la función

`glMultMatrix*()`

que realiza directamente una multiplicación con una matriz con coordenadas homogéneas. Claro, para que sea una transformación afine la matriz no debe ser singular.

8.2. Transformaciones de OpenGL

Como ya dijimos en la descripción de características (Section 5.1) OpenGL implementa la cadena de reproducción (“rendering pipeline”), es decir, todo el camino desde el modelo del mundo virtual hasta su representación en la ventana en la pantalla.
Este camino implica varias transformaciones desde un sistema de coordenadas al siguiente. OpenGL usa diferentes matrices en los diferentes pasos:

1. sistema de coordenadas (tridimensional) del objeto

- se aplican unas transformaciones usando la matriz modelado, que se selecciona con `glMatrixMode(GL_MODELVIEW)`

2. sistema de coordenadas (tridimensional) del mundo virtual
   - se aplican unas transformaciones usando la matriz de perspectiva, que se selecciona con `glMatrixMode(GL_PERSPECTIVE)`
3. sistema de coordenadas (tridimensional) de la vista
   - se aplica una transformación que genera coordenadas bidimensionales normalizadas. Tales características no hace falta cambiarlas, ya que están dadas implícitamente por la proyección aplicada antes
4. sistema de coordenadas (bidimensional) de visualización
   - se aplica una transformación que genera coordenadas que corresponden a unidades del dispositivo (en nuestro caso: pixeles en una ventana de la pantalla)
5. sistema de coordenadas (bidimensional) del dispositivo
   -

¿Uno se puede preguntar por qué hay dos matrices, la del modelado y la de la perspectiva?
Unas razones son:

- la matriz del modelado se aplica también a los vectores normales
- la iluminación, es decir, el cálculo del color del cual se debe visualizar un punto del mundo virtual, se realiza antes de aplicar la proyección
- OpenGL permita especificar unos planos de recorte adicionales (usando la función `glClipPlane()`) que se aplica convenientemente antes de distorsionar el mundo virtual debido a la proyección

8.3. Matrices

Entonces, OpenGL maneja una matriz de modelado y una matriz de perspectiva. (Hay dos matrices más: la de las texturas y la de los colores, que no vamos a mirar más detenidamente.)
Una vez que hemos seleccionado con la cual de las matrices queremos trabajar

`glMatrixMode()`

se puede manipular la matriz escogida con

`glLoadIdentity()`
`glLoadMatrix*()`
`glMultMatrix*()`

Para facilitar el trabajo con las matrices, OpenGL dispone de una pila para cada tipo de matriz que podemos usar para restaurar estados anteriores sin la costosa aplicación de una inversa:

```
glPushMatrix()
glPopMatrix()
```

9. Leda

9.1. Características

- Leda contiene un amplio conjunto de estructuras de datos y varios algoritmos que trabajan sobre ellos
- Leda está desarrollada en C++ utilizando el concepto de clases y templates

Usamos Leda de forma muy restringida, ya que sólamente queremos evitar trabajar con punteros e índices. Es mucho más fácil aprovechar los objetos y sus métodos ya definidos.

9.2. Puntos

Usamos los puntos bidimensionales `point` y los puntos tridimensionales `d3_point`.

9.3. Vectores

No se pueden realizar ciertas operaciones directamente con puntos, por eso usamos el tipo de datos `vector` que nos proporciona las operaciones necesarias.

9.4. Listas

Las listas de Leda tienen (casi) todas las operaciones implementadas. Sobre todo sus iteradores son muy fáciles de usar. Existen dos tipos de ellos: iteradores sobre los contenidos e iteradores sobre los contenedores (los llamados `list_item`).
Ejemplo:
Si almacenamos las esquinas de un polígono tridimensional en una lista de puntos del tipo `d3_point` podemos dibujar dicho polígono con OpenGL así:

```
list<d3_point> L;
...
glBegin(GL_POLYGON);
forall(p,L) {
  glVertex3d(p.xcoord(),p.ycoord(),p.zcoord());
}
glEnd();
```

Ejemplo:
Para calcular un área ponderada (Section 13.2) aplicando la fórmula usamos el iterador de contenedores porque necesitamos acceso al sucesor cíclico:

```
list<d3_point> L;
...
double    A_xy=0.0;
if(L.size()>2) {
  list_item it,jt;
  forall_items(it,L) {
    jt   =L.cyclic_succ(it);
    A_xy+=jt.contents().xcoord()*it.contents().ycoord()-
          it.contents().xcoord()*jt.contents().ycoord();
  }
  A_xy*=0.5;
}
```

9.5. Polígonos

Leda no dispone de polígonos tridimensionales, y para los bidimensionales (`polygon`) no están todas las funciones disponibles que apuntaremos que sean necesarias (Section 13.4).
Por éso desarrollamos nuestras propias clases para puntos `IGpoint` y polígonos `IGpolygon`.
Los ficheros `IGpoint.h` e `IGpolygon.h` ya están preparados parcialmente y contienen las declaraciones de las clases.
Los ficheros `IGpoint.cpp` e `IGpolygon.cpp` también están preparados parcialmente y contienen las definiciones de los métodos.

10. STL

10.1. Características

- STL contiene un conjunto de estructuras de datos básico en forma de plantillas (*templates*) y los algoritmos simples que trabajan sobre ellos
- STL está desarrollada en C++ utilizando el concepto de plantillas para que el usuario pueda trabajar con sus classes propias dentro de dichos contenedores.

Usamos STL de forma muy restringida, ya que sólamente queremos evitar trabajar con punteros e índices. Es mucho más fácil aprovechar los objetos y sus métodos ya definidos.

10.2. Puntos y Vectores

STL no da soporte para objetos geométricos, por eso, tenemos que implementar las clases que representan puntos o vectores y que contengan los métodos apropriados para trabajar con ellos. Por ejemplo en siguiente trozo de código implementa un punto `d3_point` que simula el comportamiento de un punto tridimensional de Leda tal como lo usamos en el £chero `IGPoint.h`. El código se usa solamente si la variable `HAVE_LEDA` no está de£nida durante la compilación (en el otro caso se ve que se incluye el punto `d3_point` de Leda).

```
#if defined(HAVE_LEDA)
#include <LEDA/d3_point.h>
#else
class d3_point {
protected:
  double x,y,z;
public:
  d3_point(
  ) :
    x(0.0),
    y(0.0),
    z(0.0)
  {
  }
  d3_point(
    double x_,
    double y_,
    double z_
  ) :
    x(x_),
    y(y_),
    z(z_)
  {
  }
  d3_point(
    const d3_point& p
  ) :
    x(p.x),
    y(p.y),
    z(p.z)
  {
  }
  d3_point& operator=(
    const d3_point& P
```

```
  ) {
    x=P.x;
    y=P.y;
    z=P.z;
    return *this;
  }
  d3_point operator*(
    double d
  ) const {
    return d3_point(x*d,y*d,z*d);
  }
  d3_point operator+(
    const d3_point& P
  ) const {
    return d3_point(x+P.x,y+P.y,z+P.z);
  }
  const d3_point& to_vector(void) const { return *this; }
  double xcoord(void) const { return x; }
  double ycoord(void) const { return y; }
  double zcoord(void) const { return z; }
};
#endif
```

10.3. Listas

Las listas de STL se usa (casi) igual como las listas de leda (ya que ambos están implementadas con plantillas). ¡Pero sus iteradores son diferentes! Existen dos tipos de iteradores: iteradores constantes que no permiten manipular el objeto contenido en la lista, y iteradores no constantes que sí permiten modi£caciones de los objetos.

Ejemplo:

Si almacenamos las esquinas de un polígono tridimensional en una lista de puntos del tipo `d3_point` podemos dibujar dicho polígono con OpenGL así:

```
  list<d3_point> L;
  ...
  glBegin(GL_POLYGON);
  for(
    list<IGPoint>::const_iterator it=L.begin();
    it!=L.end();
    it++
  ) {
    glVertex3d(it->xcoord(),it->ycoord(),it->zcoord());
```

```
}
glEnd();
```

Es decir, los iteradores de STL se comportan parecido a *punteros inteligentes* que se inicializa para que *apunten* al principio de la lista, y se incrementa simplemente para avanzar sobre la lista hasta que se llegue a su £nal.

11. Orientación

El concepto de orientación es muy importante en el ámbito de la informática gráfica (y de la geometría algorítmica en general).
Miramos de nuevo los ejemplos de diferentes polígonos (Section 6.2) bidimensionales. Algunos están clasificados como orientables. ¿Qué significa?
Miramos de nuevo el juego con la cuerda:

- imagínate el borde del polígono como una cuerda colocada encima de la mesa (los puntos están marcados en la cuerda)
- pincha con un clavo (o un bolí) en un punto interior del polígono
- tira a la cuerda y observa en qué dirección los puntos marcados en la cuerda le dan la(s) vuelta(s) al clavo (bolí)

Obviamente, todos los puntos en una región del polígono tienen las mismas propriedades respeto a las vueltas dadas.
Apuntamos las siguientes características:

- el *número de vueltas* que da la cuerda
- si todas las vueltas en todas las regiones *giran en la misma dirección* o no.

Si todas las vueltas de una región giran en sentido *antihorario*, se dice que la orientación de la región es *positiva*.
Si todas las vueltas de todas las regiones giran en el mismo sentido, se define que el polígono es *orientable*.
Los polígonos simples tienen sólamente una región, por lo tanto, siempre son orientables. Si el polígono no es simple, no es tan fácil de comprobar si es orientable o no (no lo vamos a ver en esas prácticas). Sin embargo, calcular el número de vueltas de una región es bastante fácil (pero tampoco lo vamos a ver).
Asumimos que tenemos un polígono orientable. ¿Cómo se detecta el sentido de la orientación?
El truco de la esquina: Fijémonos en la esquina con la coordenada *x* mínima (si hay más de una, en aquella que también tiene la coordenada *y* mínima). Basta con determinar la orientación justo

en esa esquina, es decir, el triángulo formado por dicha esquina, la esquina anterior y la esquina posterior tienen la misma orientación que todo el polígono, claro, respetando el orden dado en la lista del polígono.
¿Cómo se calcula en un programa? Esperamos la respuesta que se va a dar una vez que sabemos cómo se calcula el área (Section 13.2) y dedicamos un momento a cómo OpenGL aprovecha la orientación.

11.1. Eliminar caras

Ya mencionamos al final de la definición de un polígono (Section 6.1) que un polígono tridimensional tiene dos caras.
Si modelamos un poliedro cerrado con caras no-transparentes (se trata entonces de un volumen), es obvio que no podemos ver nunca una cara que se encuentra a espaldas del poliedro, es decir, en la visualización de nuestro mundo virtual no hace falta dibujar esas caras.
Eliminar dichos polígonos significa que no estén tratados más en siguientes fases del “pipeline” y éso resulta normalmente en una ejecución más rápida del programa.
¡Pero tenemos que tener cuidado modelando el objeto! Por defecto, OpenGL trata la cara con orientación positiva (antihorario) como cara anversa. Eso significa que tenemos que modelar el poliedro así que todas las listas de puntos de sus polígonos vistas desde fuera del poliedro estén orientadas en sentido antihorario.
El resultado de esa forma de modelar el poliedro es que en la proyección al plano de visualización los polígonos cuyas caras anversas se ven están orientados en sentido positivo y los polígonos cuyas caras reversas se ven están orientados en sentido negativo (y así OpenGL los puede distinguir, p.e., con el truco de la esquina (Section 11)).
Antes hay que habilitar este modo de OpenGL:

```
glEnable()
```

y hay que decir cuáles de las caras no se quieren dibujar:

```
glCullFace()
```

La definición de cuál es la cara anversa y cuál la cara reversa se pueda cambiar globalmente con

```
glFrontFace()
```

12. Fichero de entrada

Para facilitar la tarea de modelado definimos un formato de un fichero que va a contener la descripción de polígonos tridimensionales:

```
N [ t n [ x y z r g b]_n ]_N
```

con la siguiente explicación:

`N`	número de objetos en el fichero
`t`	tipo del objeto (por el momento siempre 0)
`n`	número de puntos de un polígono
`x y z`	coordenadas de un punto
`r g b`	color de un punto
`[...]_k`	significa que se repite la cosa entre las corchetas `k` veces

Ejemplo:
Un fichero que define un solo triángulo con las tres esquinas coloreadas diferente sería:

```
1
0
3
0 0 0 1 0 0
0 1 0 0 1 0
1 0 0 0 0 1
```

13. Extensión: El plano de un polígono tridimensional

La definición de un polígono (Section 6.1) exige que todos los puntos se encuentren en un solo plano. Entonces, nos enfrentamos a dos problemas:

1. ¿Cómo calcular el plano dados los puntos?
2. ¿Cómo verificar si los puntos, verdaderamente, están en un solo plano?

13.1. El vector normal del plano

Si sabemos que todos los puntos están en un plano, un vector normal al plano se podría calcular sencillamente con:

1. Se escoge tres esquinas no-colineales del polígono, las llamamos, p.e., *p*, *q* y *r*, respetando su orden en la lista.
2. Se calcula un vector normal *N* al plano como el producto vectorial de los dos vectores de diferencias, es decir, $N=(r-q)\times(p-q)$

¿En qué dirección apunta dicho vector?
Usamos una regla de la mano *derecha* para interpretar el producto vectorial:

> Si el pulgar apunta en dirección del primer vector y el dedo índice apunta en dirección del segundo vector, entonces, el dedo medio apunta en direccion del vector del producto.

Por otro lado, si se puede seguir a tres vectores de esta manera con los dedos de la mano derecha, los vectores forman un *sistema de coordenadas derecho*.
Matemáticamente, eso se expresa de la siguiente forma:

> Tres vectores (tridimensionales) *a*, *b* y *c* forman un sistema de coordenadas derecho si el producto mixto *a*(*b* x *c*) (es decir, el producto escalar entre *a* y el producto vectorial entre *b* y *c*) es mayor que cero (es decir, estrictamente positivo).

Sin embargo, calcular el vector normal de un polígono a base del producto vectorial no es recomendable por dos razones:

- calcular el producto vectorial no es muy estable si los vectores están casi colineales y
- hay que saber de antemano que todos los puntos están en un plano, que todavía no sabemos

Miramos otra forma un poquito más compleja:
Dado un polígono tridimensional (por el momento asumimos que esté colocado en un solo plano), se puede calcular un vector normal *N* con la siguiente fórmula:

$$N = \begin{pmatrix} A_{yz} \\ -A_{xz} \\ A_{xy} \end{pmatrix}$$

donde A_{xy} (respectivamente A_{xz} y A_{yz}) es el área de la proyección del polígono al plano *xy* (respectivamente *xz* e *yz*) del sistema de coordenadas.
Bueno, hemos pasado la pelota: en vez de calcular el vector normal tenemos que calcular áreas...

13.2. El área del polígono

Existen varias formas de calcular el área de un polígono. Pero la mayoría de la literatura (incluyendo Leda) se limita a calcular el área de un polígono simple.
Una de las formulas (para un polígono simple) es la siguiente:

$$A = \frac{1}{2} \sum_{i=0}^{n-1} (x_i y_{i+1} - x_{i+1} y_i)$$

donde n es el número de esquinas, x_i e y_i son las coordenadas de la esquina i. Está claro: en la suma se calculan los índices módulo n, es decir, x_n es igual a x_0 e y_n es igual a y_0.
Se observa que con la fórmula se calcula un valor negativo si la orientación del polígono es negativa (es decir, sentido horario). O con otras palabras: calculando el área de un polígono simple con esa fórmula, se puede derivar de su signo, si la orientación del polígono es positiva o

negativa (que nos resuelva el problema todavía abierto del apartado sobre la orientación (Section 11)).
¿Qué pasa si aplicamos la fórmula a polígonos no-simples?
Pues, no se calcula el área tal cual se entiende normalmente sino el *área ponderada*.
Recordamos el juego con la cuerda (Section 11). Hemos observado cuantas veces el polígono da la vuelta alrededor de algún punto en una región del polígono. Si contamos las vueltas en sentido antihorario positivo y las vueltas en sentido horario negativo, podemos asignar a cada región un número entero que llamamos *número de vueltas* (se llama "winding number" en inglés).
Asumimos que tenemos un polígono que tenga k regiones (en el caso de un polígono simple k es igual a 1). Cada región tiene su área, p.e., A_j en la región j.
Entonces definimos el área ponderada del polígono como:

$$A = \sum_{j=0}^{k} w_j A_j$$

donde los w_j son justamente el número de vueltas.
Lo divertido viene ahora:
¡La fórmula dada arriba para calcular el área de un polígono simple, calcula para un polígono no-simple el área ponderada!
¿Quién puede comprobarlo?
Queríamos calcular las áreas de los polígonos en los planos del sistema de coordenadas con el fin de obtener un vector normal, y ahí no hablamos de polígonos simples.
Más concreto: la fórmula para el vector normal sigue siendo correcta si usamos el área ponderada en lugar de el área normal y corriente.

13.3. El centro del polígono

Además del vector normal del plano del polígono necesitamos también un punto, el cual estamos seguros que está en el plano.
Podríamos escoger simplemente un punto al azar de las esquinas.
Sin embargo, es más robusto usar algún punto céntrico. Una posibilidad es usar el centro, que es la media aritmética de todas las esquinas y se calcula como:

$$c = 1/n \sum_{i=0}^{n-1} p_i$$

donde los p_i son las esquinas del polígono. (La suma entre puntos normalmente no está definida, pero se entiende aquí como suma de vectores tridimensionales.)

13.4. Conclusión

Para que nuestra aplicación basada en OpenGL funcione de una forma correcta, robusta, tolerante y eficiente tenemos que tratar listas de puntos que supuestamente definen polígonos tridimensionales de la siguiente manera:

1. Eliminamos puntos múltiples consecutivos de la lista.
2. Calculamos las tres proyecciones en los planos del sistema de coordenadas.
3. Calculamos las áreas ponderadas (Section 13.2) de los tres polígonos.
4. Construímos un vector normal (Section 13.1). Si no es posible rechazamos el polígono.
5. Calculamos el centro de los puntos (Section 13.3).
6. Proyectamos los puntos al plano definido por el centro y el vector normal y
   a) aceptamos el polígono “corregido” o
   b) rechazamos el polígono si la distancia de algún punto a este plano es demasiado grande.
7. Proyectamos el polígono (ahora es seguro que es plano) al plano del sistema de coordenadas dado por la componente dominante del vector normal.
8. Comprobamos si el polígono es simple y convexo (Section 6.6). Si no es asi:
   a) subdividimos el polígono en partes simples y convexos o
   b) rechazamos el polígono.

Salvo la última, todas las operaciones se pueden realizar en un tiempo de ejecución lineal en el número de esquinas del polígono.
Para l@s interesad@s: se ha desarrollado también un algoritmo que realiza la subdivisión de un polígono simple pero no-convexo en triángulos que se ejecuta en tiempo lineal en el número de esquinas del polígono. Los polígonos no-simples siempre necesitan más tiempo de cálculo, porque pueden tener muchas regiones que haya que tratar.
Además: las operaciones no salen del espacio de los números racionales, que es una propiedad muy importante en el ámbito de la geometría algorítmica, ya dijimos antes: si se trabaja con los flotantes de los procesadores, muchas veces la tarea se convierte en una verdadera aventura, o mejor dicho, una verdadera pesadilla si la meta es hacer algo bién hecho.
Antes de inventar la rueda por quinta vez, usamos la librería Leda que nos proporciona las estructuras de datos básicos.

14. Animaciones

Es muy fácil realizar animaciones sencillas con OpenGL.

14.1. Temporizador

Cada vez que se ha acabado el tiempo de un temporizador ejecutamos tres funciones

1. transformamos (Section 8) nuestro mundo virtual,
2. mandamos un mensaje a nuestro programa que redibuje el contenido de la ventana,
3. y reponemos el temporizador de nuevo.

El temporizador tiene que aceptar un parámetro que es la identificación de la ventana. Nosotros usamos por el momento dicho parámetro solamente para pasarlo de nuevo registrando el temporizador otra vez.
Para no complicar la cosa mucho, usamos dos variables globales que nos indican si estamos en modo de animación y cuantos milisegundos dejamos pasar con el temporizador.

```
static bool animate       =false;
static int  timer_interval=500;

static void Timer(
  int ident
) {
  if(animate) {
    TransformWorld();
    glutPostRedisplay();
    glutTimerFunc(timer_interval,Timer,ident);
  }
}
```

Dedicamos la tecla ‘t’ para comenzar y terminar la animación, es decir, si no estamos en modo animación (`animate==false`) activamos el temporizador la primera vez, si estamos en modo animación solamente lo desactivamos. Un fragmento de código que se coloca en la función `KeyPress()` sería:

```
...
case 'a':
  if(animate==false) {
    animate=true;
    // para lanzar el temporizador la primera vez
    // usamos la identificaci'on que devuelve la
    // funci'on glutCreateWindow() guardada en una
    // variable global llamado ident
    glutTimerFunc(timer_interval,Timer,ident);
  }
  else animate=false;
  break;
...
```

Podemos usar otras teclas para cortar o ampliar el intervalo del temporizador.

14.2. Doble búfer

Para mejorar la aparencia de las animaciones es conveniente usar un doble búfer para la memoria de la ventana. La idea es sencilla: cuando estamos contemplando el contenido de uno de los búfers, se dibuja el nuevo mundo virtual en el otro búfer, una vez terminado el dibujo se intercambia los dos búfers en un milisegundo. Así siempre veremos una imagen estable.
El uso de un doble búfer se tiene que avisar en la función

```
glutInitDisplayMode()
```

que llamamos al principio del programa y se intercambia los búfers con

```
glutSwapBuffers()
```

que será la última acción en la función `Display()`.

15. Cámara virtual

Para no tener que trabajar con matrices de proyección genéricas, OpenGL dispone de dos tipos de “cámaras virtuales” predefinidos, es decir, de tipos de proyecciones desde el mundo virtual al plano de visualización. Además existen funciones de la librería auxiliar que facilitan la especificación de cámaras.

15.1. Proyección ortogonal

Ya usamos la proyección ortogonal desde el principio (Section 5.5) del curso

```
gluOrtho()
```

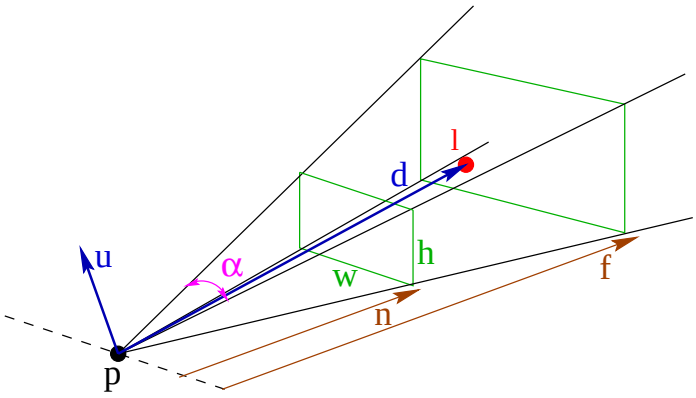
Si no se quiere incluir la coordenada z (que se usa para el recorte de los objetos) se puede usar

```
gluOrtho2D()
```

que pone los planos de z a 1 y -1.

15.2. Proyección de perspectiva

Para especificar una cámara virtual con perspectiva se necesita los siguientes parámetros:



p	posición	(position)
d	dirección de mirada	(direction)
l	centro de enfoque	(look-at point)
u	dirección hacia arriba	(up-vector)
n	distancia al plano anterior	(near distance)
f	distancia al plano posterior	(far distance)
a=w/h	aspecto	(aspect)
α	ángulo de abertura	(aperture angle)

Claro, hace falta solamente uno de los dos: o bién el centro de enfoque *l* o bién la dirección *d* desde la posición de la cámara hasta en centro de enfoque dado que $d = l - p$.
OpenGL dispone de la función

`glFrustum()`

para especificar una cámara virtual asumiendo que está colocada en el origen (es decir, $p = (0, 0, 0)$).
Otra posibilidad es usar directamente los parámetros dados en el dibujo y la tabla aprovechando la librería auxiliar. Los cuatro parámetros α, *a*, *n*, y *f* se pueden especificar mediante una llamada a la función

`gluPerspective()`

y los tres parámetros restantes, es decir, *p*, *c*, y *u* con

`gluLookat()`

16. Visualización correcta con profundidad

OpenGL dispone de un z-búfer (memoria de profundidad) que se puede configurar de diferentes maneras.
Antes de poder usar el búfer hay que indicar su posible uso durante la inicialización:

`glutInitDisplayMode()`

La configuración del z-búfer se realiza con los siguientes funciones

`glDepthFunc()`

que determina el tipo de comparación por hacer,

`glDepthMask()`

que decide si se puede escribir en el búfer (normalmente se activa, aún en el uso avanzado. Cuando se quieren generar sombras se desactiva),

`glDepthRange()`

que establece el rango (normalmente entre 0 y 1, así es por defecto; otros valores se usa en visualización avanzada, p.e., para dibujar el mundo virtual en capas respeto a la coordenada z),

`glClearDepth()`

que define el valor que se usa para inicializar el z-búfer.
Finalmente se activa su uso con

`glEnable()`

y se borra antes de dibujar un nuevo mundo virtual con

`glClear()`

17. Materias avanzadas

Hay muchas capacidades y materias de OpenGL que no podemos tocar en estás prácticas por falta de tiempo.
Una lista (no completa) es:

- modos de iluminación y sombreado
- uso de materiales para superficies
- especificación de fuentes de luz
- modelado avanzado
- operaciones directas con pixeles
- operaciones con texturas
- introducción de niebla

18. Enlaces

- la página principal de Leda:
  `http://www.algorithmic-solutions.de/as_html/products/products.html`

- la página principal de OpenGL de Silicon Graphics:
  `http://www.sgi.com/software/opengl/index.html`
- la página principal de usuarios de OpenGL:
  `http://www.opengl.org`
- un cursillo bueno de técnicas avanzadas con OpenGL:
  `http://www.sgi.com/software/opengl/advanced98/notes/notes.html`

19. Tareas

19.1. Soluciones

Parte de las soluciones se encuentra en el código documentado (`http://www.ei.uvigo.es/~formella/ig00/html/index.html`).

19.2. Bases para la entrega de trabajos

Las bases para la realización y entrega del trabajo de Prácticas de Informática Gráfica:

1. Los grupos de trabajo serán, como máximo, de **2 personas**.
2. Requerimientos mínimos de la prática (6 puntos).
   - Dibujar objetos (polígonos, líneas, puntos) de diferentes modos, colores, patrones etc.
   - Utilizar eventos de teclado.
   - Utilizar matrices de transformación.
   - Utilizar animaciones.
3. Aportaciones propias (2.5 puntos).
   - Creatividad y complejidad.
   - Uso avanzado de Leda o de STL.
   - Uso avanzado de las librerías `glu` y `glut`.
4. Desarrollo de software (1.5 puntos).
   - Documentación mínima que incluya breve manual de usuario.
   - Código documentado.

Plazos:

- Cada alumno deberá entregar **el último día de prácticas (3 de Junio)** el trabajo con su documentación correspondiente.
- **8 de Junio por la mañana.** Se publicará una lista con aquellos alumnos que, aún habiendo entregado el trabajo, deberán realizar la parte práctica del examen para poder superar la asignatura.
- **21 de Junio a las 10:00 h.** Todos aquellos que no hayan desarrollado el trabajo cubriendo los requerimientos mínimos, tendrán que realizar la parte práctica del examen (junto con el examen de teoría).