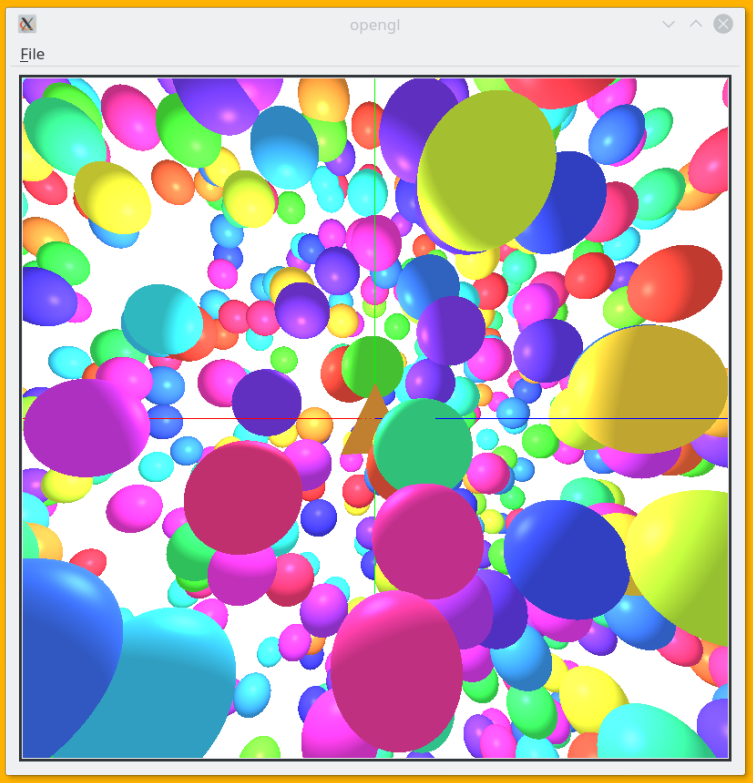


Informática Gráfica con OpenGL 4

Domingo Martín Perandrés

2018

Índice general

Índice General III

1 Una esfera por revolución 1

Índice de figuras

1.1 Barrido por revolución. 1
1.2 Perfil para obtener un cilindro 2
1.3 Despliegue de los puntos en un plano 2
1.4 Posicionado lineal de los puntos 3
1.5 Posicionado 2D de los puntos 3
1.6 Triángulos degenerados 4
1.7 Versión optimizada 4
1.8 Resultado del ejemplo 1 con los diferentes modo de visualización 6

Introducción

Informática Gráfica es una asignatura de tercer curso del grado en informática de la Universidad de Granada. Con temática y nombres similares se encuentra en los grados de los estudios de informática en diferentes universidades españolas. La informática gráfica trata de cómo se pueden producir imágenes usando un ordenador. Con tal objetivo es fácil entender la importancia que tiene dado que los humanos, la mayoría, captamos la mayor parte de la información de nuestro entorno (e internamente) mediante información visual y que el uso de los ordenadores está ampliamente extendido en la sociedad actual.

Dado que la asignatura es el primer contacto del alumno con la informática gráfica, su impartición comienza desde lo más básico, avanzando hasta llegar a un nivel en el que se pueden programar aplicaciones 3D interactivas. Los alumnos disponen de cuantioso material, además de las clases teóricas y prácticas, para adquirir los conocimientos necesarios sobre la materia. Dados los avances que se han producido tanto en el software como en el hardware, y sobre todo, dada la amplitud del área, los contenidos se centran en cuatro temáticas básicas: aprender qué es un modelo, aprender qué es un modelo jerárquico, aprender a cómo se consigue realismo añadiendo iluminación y texturas, e interacción. No sólo se aprenden los conceptos sino que sobre todo hay que implementarlos y visualizarlos. Esto implica entender los conceptos de objetos, cámara, iluminación, etc., los cuales conforman la escena.

Para facilitar el desarrollo de las aplicaciones hago uso de una biblioteca (*library*[1]) llamada OpenGL[2]. OpenGL contiene toda la funcionalidad para poder visualizar objetos 3D de una manera relativamente sencilla de tal manera que el usuario sólo tiene que hacer unas pocas llamadas para obtener resultados. Docentemente esta es una característica muy deseable pues los alumnos comienzan a ver, no sólo figurada sino realmente, los resultados de su trabajo desde el principio.

OpenGL tiene numerosas ventajas, aunque las principales son que existen interfaces para numerosos lenguajes y, sobre todo, que es multiplataforma. Esta última característica es especialmente importante en la docencia e investigación pues no obliga al usuario a trabajar con un solo sistema operativo. Desde que apareció en el año 1992 hasta nuestros días se ha producido, como no podía ser de otra manera, una gran evolución, siempre encaminada a mejorar el rendimiento, que en la mayoría de los casos hay que entenderlo como mayor

[1] Una vez establecido mediante el uso de la itálica que una palabra original es en inglés y lo que significa en español, en el resto del documento aparecerá de forma normal

[2] https://www.opengl.org/

velocidad con modelos más complejos. Probablemente el cambio más significativo fue el cambio de un cauce (*pipeline*) estático a otro dinámico. Esto es, de una funcionalidad que no se podía cambiar a otra que se podía programar. Pongamos un ejemplo para entender el cambio usando una batidora: la batidora estática bate muy bien pero solo tiene un cabezal y una velocidad, la batidora dinámica tiene varias velocidades y se le pueden cambiar los cabezales para adaptarse a distinto tipo de alimentos o productos. Para usar la primera sólo había que pulsar el interruptor. Para la segunda hay que saber elegir la cuchilla apropiada y ajustar la velocidad. Como se puede observar, se produce un cambio de simplicidad por flexibilidad.

Si tomamos en cuenta estas dos posibilidades en cuanto a su efecto sobre la docencia, y en particular, guiados por la interfaz de programación, para el caso de un curso de introducción a la informática gráfica en la que se exponen ideas sencillas y el rendimiento no es una componente clave, la decisión, en mi lugar, se inclina por usar el cauce estático y su sencillo paradigma de programación. Por ejemplo, dado el código de inicialización (similar aunque más simple que el mostrado en el tema **??**, el código necesario para dibujar un punto es el siguiente:

```
void _gl_widget::draw_object()
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
  glBegin(GL_POINTS);
  glVertex3f(0,0,0);
  glEnd();
}
```

Si comparamos este código con todo el código que hay que escribir para dibujar un sólo punto (ver los temas **??** y **??**), es fácil ver por qué es más conveniente cuando se está empezando a aprender los conceptos básicos de la informática gráfica. Esta es la aproximación que llevo a cabo en la asignatura Informática Gráfica, centrando la atención en la comprensión más que en la velocidad o el rendimiento.

Dicho esto, también existe la posibilidad de comenzar a usar OpenGL utilizando la versión programable y su interfaz de programación correspondiente para estudiar los conceptos básicos de informática gráfica: la única condición es que se necesita de más tiempo para la comprensión de cómo funciona, más tiempo para programar, y más tiempo para depurar. La recompensa será que se está trabajando con una interfaz moderna en vez de la obsoleta, que las aplicaciones irán mucho más rápidas, y que se tendrá la posibilidad de hacer cosas que no se pueden hacer con el cauce estático, y que en principio fue el motivo para el cambio de una aproximación a la otra.

Para aquellos alumnos que les interese y les guste el mundo de los gráficos, aunque implique un mayor esfuerzo —una inversión no un gasto— he creado este manual en el que se expone el temario que vemos en la asignatura pero realizado con la interfaz de programación de OpenGL en la versión 4.5.

Este pretende ser un manual que permita la creación de programas usando OpenGL y programas para GPU (*shaders*) de la forma más rápida, planteando ejemplos prácticos y resolviendo los conceptos teóricos sobre la marcha. El desarrollo se irá realizando de forma gradual, empezando desde lo más básico y añadiendo complejidad en cada paso. En algunos de estos pasos tendremos que incluir contenidos teóricos. Para todos los ejemplos existe

el código correspondiente, aunque hay que incidir en que facilita el aprendizaje el intentar partir del esqueleto inicial e intentar codificar cada ejemplo una vez que se ha entendido. Si se estudia el código de cada ejemplo, se podrá comprobar que una misma funcionalidad puede ir cambiando conforme se avanza. Esto es así para reflejar el hecho de que normalmente producimos una primera solución que funciona pero que conforme la vamos usando podemos apreciar lo que va bien y lo que va mal, permitiendo una mejora. Esta es la forma de trabajar que se propone con estos ejemplos.

A partir de este momento, comienza la aventura de ver y comprender cómo unos cuantos (muchos) números se pueden convertir en imágenes que nos sorprendan.

1

Una esfera por revolución

Una vez hemos visto como se dibuja una escena con los ejes y un cubo vamos a visualizar un objeto más complejo: una esfera. Hay que observar que lo que vamos a conseguir es una aproximación a la esfera pues la vamos a construir con triángulos, la única primitiva con superficie que disponemos. Es fácil ver que nuestro modelo de la esfera será mejor cuantos más triángulos usemos.

En el caso de los ejes y el cubo nos hemos podido permitir el crearlos directamente introduciendo las posiciones de los vértices y las composiciones de los triángulos, pues su número era muy reducido. En general vamos a tener que buscar otros procedimientos que permitan la creación de objetos de forma automática. Uno de dichos procedimientos es el barrido por revolución. La idea es muy sencilla: dada una curva, se hace girar alrededor de un eje un numero de veces, y a partir de dichos perfiles se crea la superficie. En la figura 1.1 se puede ver una curva generatriz para crear una especie de copa.

Los puntos de la curva o perfil generatriz se giran alrededor del eje y obteniendo una serie de perfiles a partir de los cuales se pueden crear las caras. Para realizar el giro de cada punto del perfil, usamos las funciones seno y coseno, pues se puede observar que realmente lo que estamos haciendo es calcular las posiciones en una circunferencia. Por ejemplo, dado un punto con las siguiente coordenadas $P(x,y,0)$ (obsérvese que la coordenada z es igual a 0, pues nuestra generatriz está en dicho plano), lo que queremos es rotarlo alrededor del eje y. Esto implica que la coordenada y no va a cambiar. Por tanto, sólo nos interesa el valor de la coordenada x, la cual la podemos

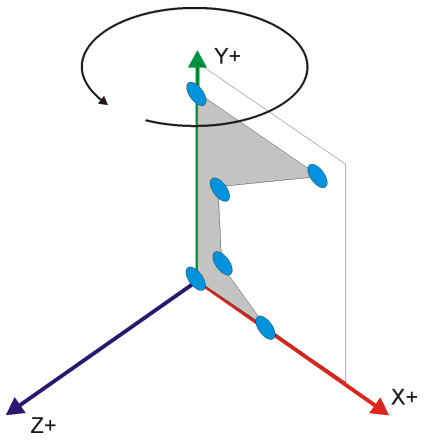


Figura 1.1: Barrido por revolución.

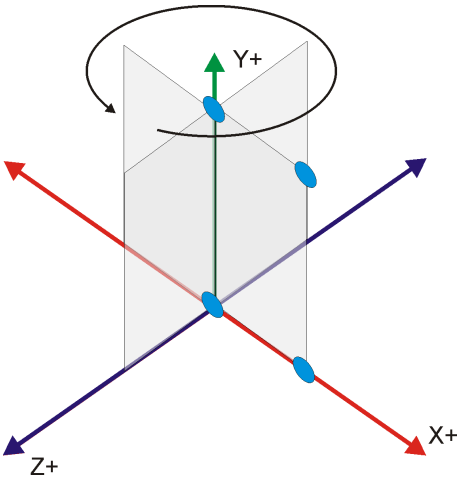


Figura 1.2: Perfil para obtener un cilindro

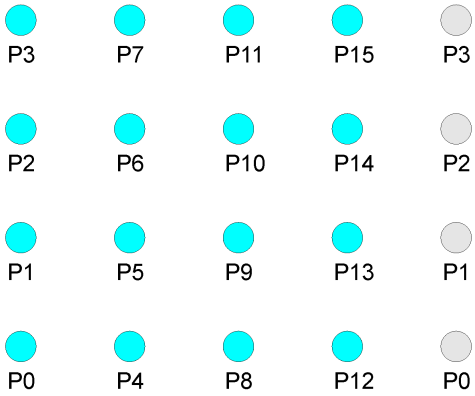


Figura 1.3: Despliegue de los puntos en un plano

tomar como el radio R de la circunferencia. Para calcular los puntos de una circunferencia dado el radio usamos una fórmula muy conocida:

$$x = R \cdot \cos(\alpha)$$
$$z = -R \cdot \sin(\alpha)$$
$$0 \leq \alpha \leq 2\pi$$

Por ejemplo, si queremos obtener 3 perfiles sólo tenemos que dividir el rango entre 3, obteniendo los ángulos 0°, 120° y 240° (los cálculos se hacen en radianes).

Vamos a ver un ejemplo sencillo para entender cómo se hace la construcción del objeto. Vamos a usa un perfil que nos permita construir un cilindro. Para ello vamos a utilizar el perfil de 4 puntos que se muestra en la figura 1.2. Además para simplificar el proceso sólo vamos a producir 4 divisiones.

El proceso de girado del perfil original permite crear los otros tres. Todos los puntos nuevos formarán la geometría del objeto. Un detalle importante a notar es que los puntos de los extremos del perfil que se encuentran en el eje y, al rotarlos producen nuevos puntos que tienen las mismas coordenadas. Esto es, son puntos diferentes que ocupan la misma posición. El almacenamiento de los puntos se realiza de forma secuencial, de tal manera que primero se meten los puntos del primer perfil, luego los del segundo, etc. Además, para cada perfil mantenemos el orden de los puntos, de tal manera que primero almacenaremos P_0, luego P_1, hasta P_3, y a continuación vendrán P'_0, P'_1, etc.

Lo que queremos hacer, una vez que hemos guardado la geometría, es crear los triángulos que representan a la superficie. La forma más sencilla es olvidarse de la forma del objeto en 3 dimensiones y crear una forma en la que lo importante sea la distribución de los puntos unos

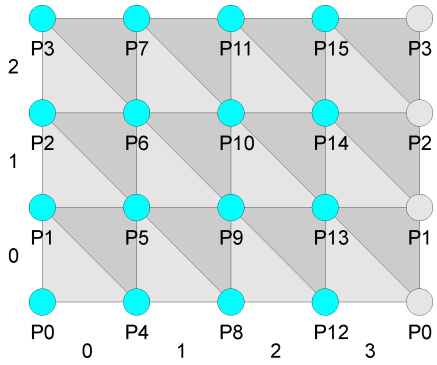


Figura 1.4: Posicionado lineal de los puntos

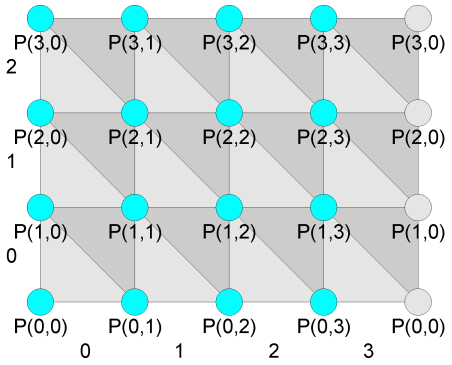


Figura 1.5: Posicionado 2D de los puntos

con respecto a otros. Para ello hacemos como que aplanamos el perfil original y el resto de perfiles calculados sobre un plano, de tal manera que la localización de los vértices mantenga la posición relativa. Este esquema lo podemos ver en la figura 1.3. Para poder cerrar el objeto es necesario crear las caras que unen los puntos del último perfil con los puntos del primer perfil. Esto se indica con la columna de puntos de color gris.

Las caras que queremos construir se muestran en la figura 1.4. Es importante ver que hemos convertido nuestro conjunto de puntos en una matriz, y que por tanto, podemos tratar esta configuración de una manera más sencilla utilizando filas y columnas. En nuestro caso tenemos 3 filas, de 0 a 2, y 4 columnas, de 0 a 3. Si nos fijamos, para cada configuración de fila y columna tenemos que crear 2 caras, que llamaremos par e impar (pues coinciden con su paridad posicional como ahora veremos).

Lo que estamos intentando es automatizar el proceso de construcción de las caras, haciendo que el mismo sea un código que se repite una y otra vez. Por tanto, nos falta obtener el patrón de construcción. Vamos a intentar deducirlo llevando a cabo, como ejemplo, la generación de las caras de un par de posiciones en la matriz. Como se sabe, una matriz la podemos recorrer por filas y para cada fila por columnas, o al revés, por columnas y para cada columna por filas. Para nuestro caso nos da igual, sólo hay que se coherentes.

Por ejemplo, nuestro código puede ser el siguiente:

```
for (int Fila=0;Fila<Num_filas;Fila++){
  for (int Columna=0;Columna<Num_columnas;Columna++){
    // crear cara par
    // crear cara impar
  }
}
```

Vamos a crear las 2 caras cuando Fila=0 y Columna=0. La cara par, $Cara_0$, estará compuesta por los puntos P_1, P_0 y P_4, mientras que la impar, $Cara_1$, estará compuesta por los puntos P_4, P_5 y P_1. Un detalle importante es que tenemos que indicar los puntos siempre si-

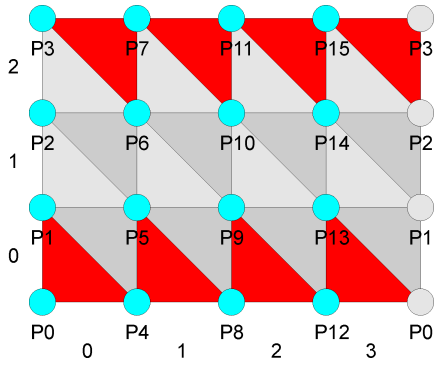


Figura 1.6: Triángulos degenerados

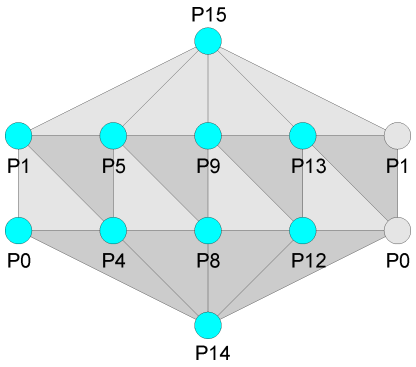


Figura 1.7: Versión optimizada

guiendo el mismo sentido, en este caso, contrario a las agujas del reloj. También podríamos definir todas las caras siguiendo el sentido de las agujas del reloj. Lo que no podemos hacer es mezclar ambos sentidos, pues como veremos, a partir de los vértices calcularemos la visibilidad de las caras y podría ocurrir que siendo visibles, al estar más definidas, se vieran incorrectamente.

Con sólo una cara no podemos ver ningún patrón. Vamos, por tanto, a crear las 2 caras cuando Fila=0 y Columna=1. En este caso, la cara par, $Cara_2$, estará compuesta por los puntos P_5, P_4 y P_8, mientras que la impar, $Cara_3$, estará compuesta por los puntos P_8, P_9 y P_5. Ahora sí se pueden empezar a ver los patrones.

Veamos la composición de las dos caras par (sólo se indican los índices de los puntos): $Cara_0(1,0,4)$ y $Cara_2(5,4,8)$. Podemos apreciar que los indices de la $Cara_2$ se pueden obtener a partir de la $Cara_1$ simplemente sumando 4, que es el número de puntos que tiene el perfil. Es fácil comprobar que ocurre lo mismo cuando la Fila=0 y Columna=2, pero ¿qué sucede cuando la Columna vale 3? Vamos a comprobarlo.

En principio los puntos de las caras serian los siguiente: $Cara_6(13,12,16)$ y $Cara_7(16,-17,13)$. Pero no tenemos realmente los puntos 16 y 17 sino que debemos usar los puntos 0 y 1. Para ello debemos recurrir al operador módulo (%), de tal manera que el valor 16 nos devuelva 0, y el valor 17 nos devuelva 1. Es este caso, nos basta con utilizar como divisor el número total de puntos generado, que es 16, 4 puntos por 4 perfiles.

Por tanto, podemos generalizar la obtención de los caras de la primera fila, si hacemos lo siguiente (ND=número de divisiones; NP=número de puntos; F=Fila; C=Columna):

- Cara par:
  $Cara_{(F\cdot ND+C)\cdot 2} = ((C\cdot NP+F)+1,(C\cdot NP+F),((C+1)\cdot NP+F))$
- Cara impar:
  $Cara_{(F\cdot ND+C)\cdot 2+1} = (((C+1)\cdot NP+F),((C+1)\cdot NP+(F+1)),(C\cdot NP+F)+1)$

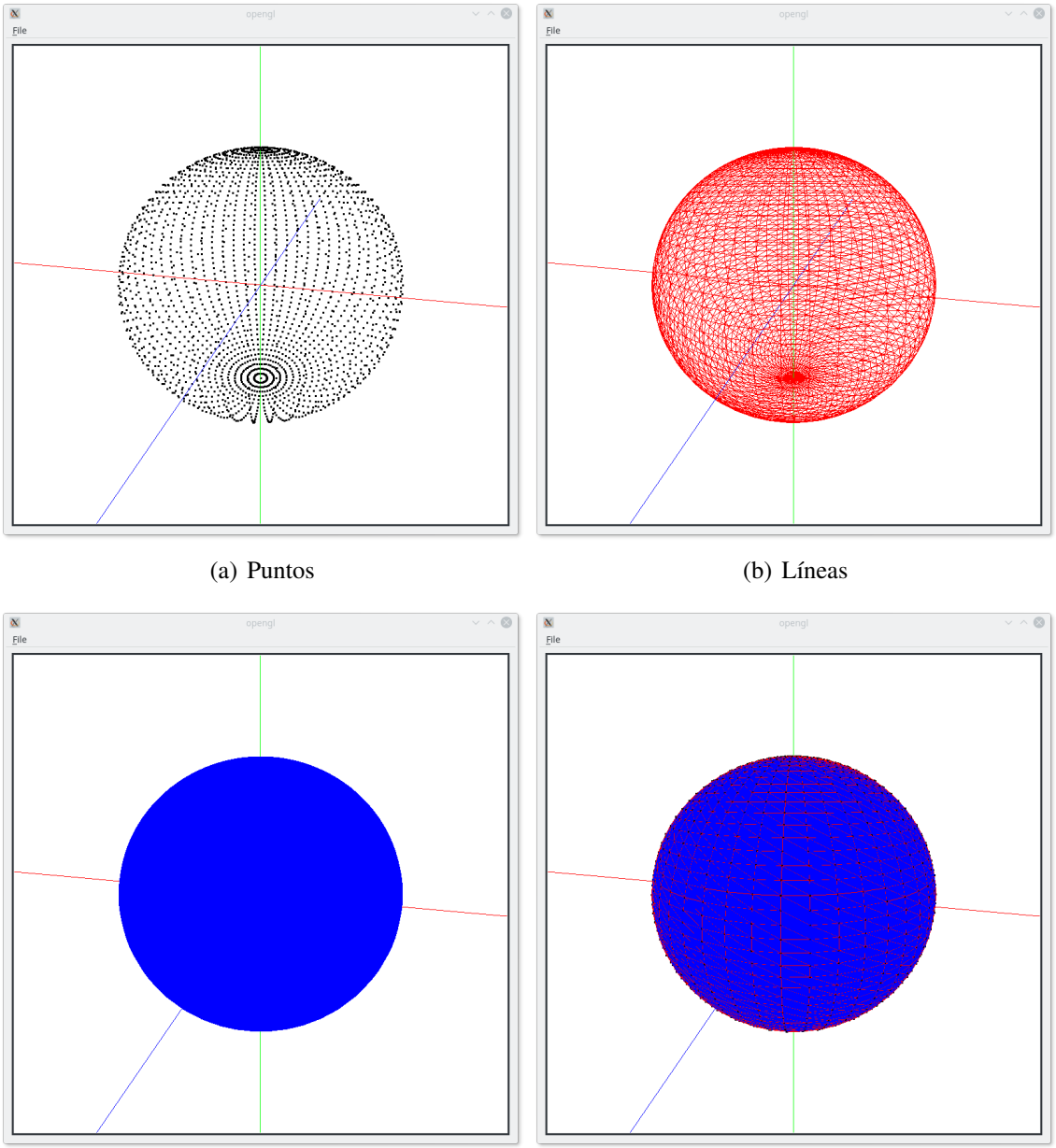
Sólo nos quedaría el hacer que se aplicara el operador módulo para obtener el algoritmo completo. En vez de mostrar dicho código, vamos a plantear otra forma de tratar el direccionamiento de los puntos dentro de la matriz que hará que nuestro código sea más fácil de leer. Para ello sólo tenemos que direccionar los puntos en forma matricial en vez de lineal, tal y como se muestra en la figura 1.5. Son los mismos puntos, almacenados de la misma manera, pero que por conveniencia los vamos a tratar como si ocuparan posiciones en una matriz. Así, vemos que al punto $P(0,0)$ le corresponde el punto P_0, al $P(0,1)$ le corresponde el punto P_4, o al $P(1,0)$ le corresponde el punto P_1. La conversión es muy sencilla: $P(Fila, Columa) = P(Columna \cdot Num_puntos + Fila)$. Ahora la creación de caras nos quedará así:

- Cara par:
  $Cara_{(F \cdot ND+C)\cdot 2} = (P(F+1,C), P(F,C), P(F,(C+1)\,\%ND))$
- Cara impar:
  $Cara_{(F \cdot ND+C)\cdot 2+1} = (P(F,(C+1)\,\%ND), P(F+1,(C+1)\,\%ND), P(F+1,C))$

Con esto conseguimos crear nuestro modelo por revolución. Pero sólo tenemos la versión más sencilla, la cual se visualiza correctamente pero no representa a un modelo correcto. Los motivos son dos: la duplicación de los puntos extremos y la creación de triángulos que no tienen área (triángulos degenerados). Esto triángulos se muestran en la figura 1.6 en color rojo.

Una versión más optimizada aún, elimina los puntos repetidos. Para ello la construcción del objeto debe distinguir entre 3 zonas: la tapa inferior, la zona central y la tapa superior. En nuestro algoritmo tenemos que reflejar esta distinción. Así, para crear los puntos de las zona central debemos quitar los puntos de los extremos, que se almacenarán al final. Después crearemos las caras de la zona central. Finalmente crearemos los triángulos de las tapas. En el caso de la tapas, podríamos crear triángulos tal y como hemos hecho hasta ahora, pero también se podría utilizar la primitiva abanico de triángulos (*triangle fan*).

Para la visualización de la esfera junto con los ejes sólo tendremos que coger el código del tema anterior y cambiar los valores del cubo por los de la esfera.



(a) Puntos (b) Líneas

(c) Relleno (d) Todo

Figura 1.8: Resultado del ejemplo 1 con los diferentes modo de visualización