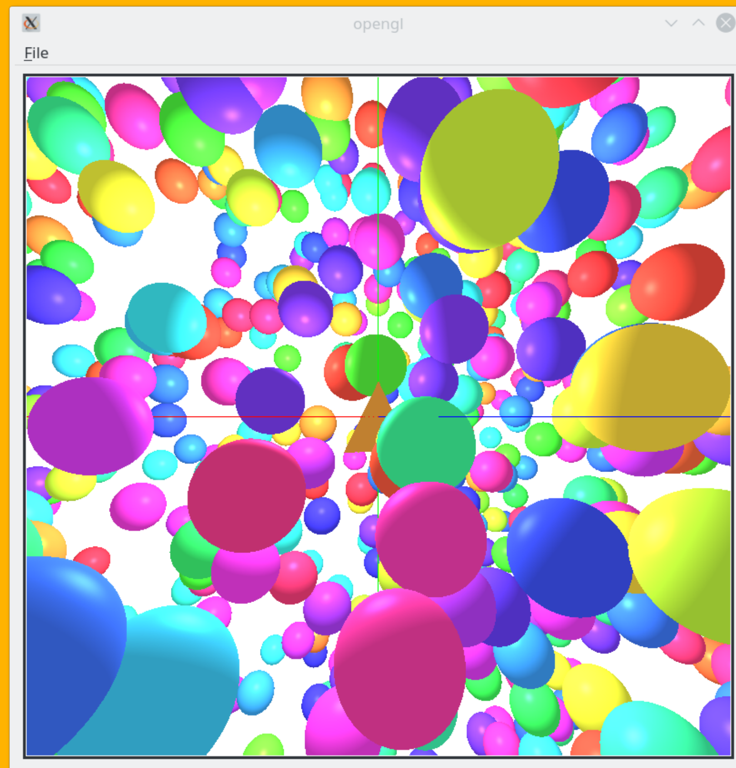


Informática Gráfica con OpenGL 4

Domingo Martín Perandrés

2018

Copyright ©Domingo Martín Perandrés 2018
Reservados todos los derechos. Queda rigurosamente prohibida, sin la autorización de los titulares del “Copyright”, bajo las sanciones establecidas en la leyes, la reproducción parcial o total de esta obra por cualquier medio o procedimiento, incluidos la reprografía y el tratamiento informático, asi como la distribuvión de ejemplares mediante alquiler o prestamo públicos.

Índice general

Índice General v

1 Sólo compilar 1

2 Un punto 11

3 Movemos el punto 15

4 Cambiamos el color 19

5 Proyección paralela 23

6 ¡Tres dimensiones! 27

7 Un triángulo 39

7.0.1 Solución 1 . . . 40

7.0.2 Solución 2 . . . 40

7.0.3 Solución 3 . . . 41

7.0.4 Solución 4 . . . 42

7.0.5 Solución 5 . . . 43

7.0.6 Solución 6 . . . 43

8 Interpolando colores 47

9 Un cubo y diversos modos de dibujado 49

10 Una esfera por revolución 57

11 Transformaciones geométricas 63

12 Modelado jerárquico sin movimiento 75

13 Modelado jerárquico con movimiento 87

14 Iluminación 95

15 Iluminación: Phong 107

16 Texturas 113

17 Texturas e iluminación 125

18 Selección 135

Índice de figuras

1.1 Cauce gráfico de OpenGL . . . 3
1.2 Punto en un espacio cartesiano derecho 3D . . . 4
1.3 Espacio de color RGB . . . 4
1.4 Discretización de un triángulo en los píxeles correspondientes. . . . 6
1.5 Resultado del ejemplo 1 . . . 10

2.1 Resultado del ejemplo 2 . . . 14

3.1 Resultado del ejemplo 3 . . . 17

4.1 Resultado del ejemplo 4 . . . 20

5.1 Proyección paralela y proyección de perspectiva . . . 24
5.2 Volumen de visión de una proyección paralela . . . 25

6.1 Proyección de perspectiva . . . 28
6.2 Frustum . . . 29
6.3 Dos sistemas de coordenadas. . . . 30
6.4 Proceso para obtener la transformación de vista. . . . 31
6.5 Relatividad en el posicionamiento: da lo mismo mover el ojo y dejar el objeto fijo, que dejar el ojo fijo y mover el objeto. . . . 33
6.6 Resultado del ejemplo 6 con la cámara en su posición original y después de rotar alrededor del origen . . . 38

7.1 Resultado del ejemplo 7 . . . 45

8.1 Resultado del ejemplo 8 . . . 48

9.1 Resultado del ejemplo 9 . . . 56

10.1 Barrido por revolución. . . . 57
10.2 Perfil para obtener un cilindro . . . 58
10.3 Despliegue de los puntos en un plano . . . 58
10.4 Posicionado lineal de los puntos . . . 59
10.5 Posicionado 2D de los puntos . . . 59
10.6 Triángulos degenerados . . . 60
10.7 Versión optimizada . . . 60
10.8 Resultado del ejemplo 10 con los diferentes modo de visualización . . . 62

11.1 Traslación 64
11.2 Escalado 64
11.3 Rotación en el eje x 65
11.4 Rotación en el eje y 65
11.5 Rotación en el eje z 65
11.6 Combinación de transformaciones para rotar y escalar con respecto a un punto que no es el origen 68
11.7 Secuenciación de objectos en un VBO. 70
11.8 Dos vistas del resultado del ejemplo 11 73

12.1 Resultado del ejemplo 12 con los diferentes modo de visualización 85

13.1 Pinza 88
13.2 Mano 88
13.3 Brazo1 90
13.4 Brazo2 90
13.5 Base 92
13.6 Ejemplo completo con movimiento 92

14.1 Reflexión difusa: cómo afecta la orientación 96
14.2 Cálculo de las normales 97
14.3 Tipos de reflexión 98
14.4 Resultados del ejemplo 14 106

15.1 Resultados del ejemplo 15 111

16.1 Pasos en la aplicación de una textura 114
16.2 Ajustes sobre las texturas 116
16.3 Resultados del ejemplo 16 123

17.1 Localización de los puntos con respecto a las coordenadas de textura con distribución más regular pero errónea en el último perfil 126
17.2 Localización de los puntos con respecto a las coordenadas de textura con distribución menos regular y errónea en el último perfil 126
17.3 Localización de los puntos con respecto a las coordenadas de textura con distribución más regular y el último perfil correcto 127
17.4 Localización de los puntos con respecto a las coordenadas de textura con distribución menos regular y el último perfil correcto 127
17.5 Resultados del ejemplo 17 133

18.1 Modelo PLY de Beethoven 136
18.2 Resultados del ejemplo 18 141

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 1, 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 1 y 2), 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

Domingo Martín Perandrés©

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

Sólo compilar

Vamos a empezar preparando la infraestructura necesaria para poder utilizar OpenGL y los shaders. Esto es, dado que OpenGL sólo se encarga de dibujar, es necesario poder abrir una ventana, y en su caso, poder capturar los eventos para poder crear una aplicación interactiva.

Para estas tareas existe varias bibliotecas, desde las más sencillas, como glut[1], hasta las más sofisticadas, como Qt[2], GTK[3], etc. En nuestros ejemplos vamos a usar Qt, lo cual nos permitirá crear interfaces de usuario más complejos, cuando así lo necesitemos.

Otra característica de OpenGL es que es extensible, y el mecanismo que usa es el de las llamadas a funciones. Por tanto, lo primero que se tiene que hacer es el enlace entre los punteros y las funciones. Podemos hacer esta correspondencia manualmente, pero es mejor usar algunos de los programas que lo permiten como es el caso de GLEW[4]. Es importante tener en cuenta que hasta que no se hace la inicialización de GLEW no es posible (produce errores) llamar a funciones de OpenGL. Hay que también tener en cuenta que si se hace un include de GLEW no se puede hacer de OpenGL.

Qt tiene una capa de software para utilizar OpenGL que teóricamente oculta y/o simplifica el uso de OpenGL. Dado que estamos aprendiendo, vamos a utilizar la forma nativa de hacerlo, lo que además permite su utilización con otras bibliotecas.

Con respecto a OpenGL, vamos a empezar con el ejemplo más básico: un programa que use shaders, se compile correctamente y al ejecutarse no de errores, aunque no haga nada más.

[1] http://freeglut.sourceforge.net/
[2] https://www.qt.io/
[3] https://www.gtk.org/
[4] http://glew.sourceforge.net/

Para ello es necesario que tengamos muy claro que por un lado vamos a tener que programar código que se ejecutará en la CPU, como hacemos normalmente, pero sobre todo, que vamos a crear pequeños programas, los shaders, que van a correr en la GPU. La diferencia más grande entre la CPU y la GPU es el grado de paralelismo: en las CPUs actuales pueden estar corriendo decenas de procesos mientras que en las GPUs pueden correr miles de procesos. Si el algoritmo es paralelizable, puede suponer uno o dos ordenes de magnitud de incremento de la velocidad de cómputo.

Otra cosa importante a tener en cuenta, es que el Sistema Operativo "normal" corre en la CPU. Por tanto, todo el tema de control y gestión del software se realiza en la CPU. Por ello, los shaders deben ser cargados, compilados y enlazados mediante el código que escribimos en la CPU, aunque en algún momento, una vez el programa y los datos se han descargado en la GPU, la misma podrá funcionar de manera autónoma y ejecutar sus propios programas, los shaders, accediendo a sus propios datos, y también intercambiar datos con la CPU. Debemos entender que vamos a tener dos procesadores funcionando en paralelo, y cada uno corriendo sus propias tareas.

Esta descripción explica el que tengamos que escribir código para la CPU por un lado y los shaders por otro.

Podemos observar que en el código para la CPU, el que hemos escrito para QT, tenemos 4 partes:

- shaders: se encarga de leer los shaders y comprobar que no hay errores.
- glwidget: se encarga de dibujar mediante OpenGL.
- window: crea la interfaz de usuario.
- main: programa principal que inicializa la aplicación.

Antes de ver cómo funciona el programa tenemos que entender que son los shaders. La definición más simple es que son programas, sólo que corren en la GPU.

En la evolución de OpenGL se pasó de un cauce gráfico fijo a un cauce programable mediante shaders. Esto aporta más flexibilidad a costa de una mayor complicación. Quizás nos estemos preguntando que es el cauce gráfico (*graphics pipeline*).

La creación de gráficos mediante un ordenador es un proceso costoso y complejo, el cual se puede dividir en tareas más sencillas que se van enlazando una detrás de otra para producir el resultado final. Es el equivalente a una cadena de montaje, donde cada sección realiza una o unas tareas muy específicas, de forma acumulativa de tal manera que con cada paso se está más cerca del final. La secuencia de estas etapas está definida y no se puede cambiar. Esto es, si usamos el símil de una cadena de montaje de coches, no podemos empezar poniendo las ruedas sino que tendremos que hacerlo construyendo el chasis.

Domingo Martín Perandrés©

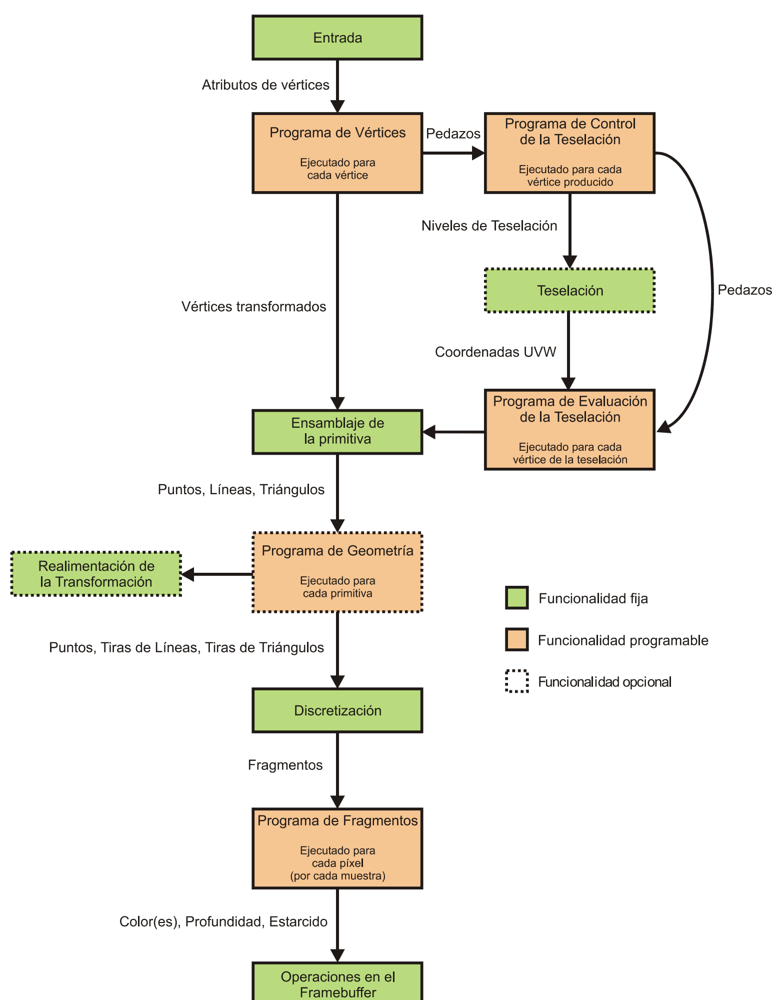


Figura 1.1: Cauce gráfico de OpenGL

Las etapas de OpenGL se pueden ver en la figura 1.1. Lo que se ha hecho al pasar del cauce estático al dinámico es que ciertas partes ya no son fijas sino que se pueden programar. Pero si con la funcionalidad fija, por ejemplo, entraban vértices y salían vértices, lo mismo debe ocurrir con la versión programable.

Actualmente tenemos los siguientes tipos:

- Vertex Shaders: GL_VERTEX_SHADER
- Tessellation Control y Evaluation Shaders: GL_TESS_CONTROL_SHADER y GL_TESS-_EVALUATION_SHADER
- Geometry Shaders: GL_GEOMETRY_SHADER
- Fragment Shaders: GL_FRAGMENT_SHADER

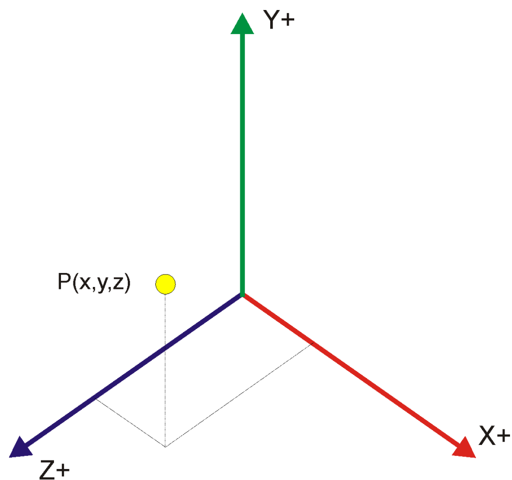


Figura 1.2: Punto en un espacio cartesiano derecho 3D

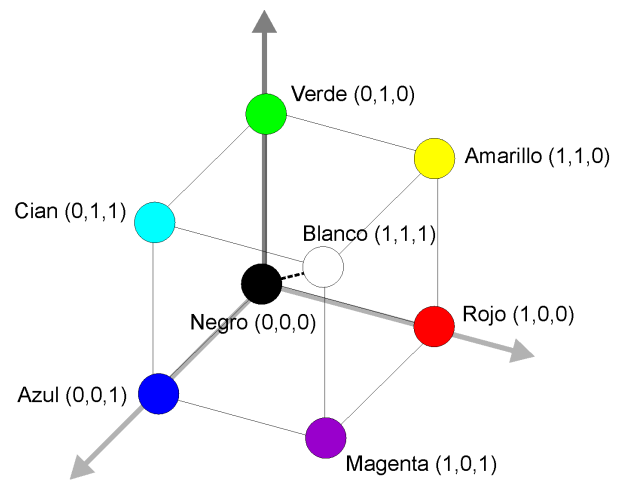


Figura 1.3: Espacio de color RGB

- Compute Shaders: GL_COMPUTE_SHADER

Los mismos se distinguen por la posición que ocupan en el cauce visual de OpenGL y por la funcionalidad que aportan. Nosotros nos vamos a centrar en el uso de los programas de vértices (*vertex shaders*) y de los programas de fragmentos (*fragment shaders*), encontrándose el primero al comienzo del cauce y el segundo al final del cauce. Indicar que la presencia de los shaders es totalmente optativa, salvo el caso de los vertex shaders.

Los vertex shaders están al comienzo del cauce gráfico. Reciben vértices y entregan vértices, más concretamente las coordenadas de los vértices (también pasan información a otras etapas). Hay que tener en cuenta que OpenGL se creó para la generación de gráficos tridimensionales, por lo que es fácil entender que se necesiten tres coordenadas para definir cada vértice o punto en el espacio (la primitiva más simple que se nos puede ocurrir), que en un sistema de coordenadas cartesianas vienen definidas por la x, y y z (figura 1.2). Por ahora no vamos a profundizar más en los sistemas de coordenadas que usa OpenGL, sólo indicar que los mismos se pueden definir en el dominio de los número reales, $\mathbb{R}$, o en el de los enteros $\mathbb{N}$. Es fácil entender que OpenGL puede trabajar con información 2D, como por ejemplo una imagen. Lo que necesita una mayor explicación es que internamente no usa 3 sino 4 coordenadas, nombrándose la última, normalmente, como w. Esto se debe a que OpenGL realiza los cálculos en coordenadas homogéneas. En principio podemos seguir adelante sin entrar en mayor detalle. Bastará con indicar que el valor de w es 1 normalmente.

Los fragment shaders trabajan con los fragmentos. Al principio parece que un fragment es un píxel, pero no, veamos en qué se diferencian. Los primero que tenemos que tener en cuenta es que cualquier escena que queramos producir finalmente se representará como una imagen, esto es una disposición rectangular de píxeles, siendo un píxel un cuadrado que

tiene como atributos una posición, (x,y) y un color, que puede venir descrito por 3 (RGB) o 4 componentes (RGBA) (figura 1.3).

Se puede observar que las posiciones sólo pueden ser enteros sin signo. El color viene indicado por las intensidades de los colores primarios rojo, verde y azul. Además se puede añadir una componente A (la A viene de canal alfa), que se encarga de indicar el valor de la transparencia. Ya lo veremos con más detalle más adelante. Lo que sí debemos comprender es que los modelos que definimos para dibujar usando primitivas definidas como objetos en coordenadas reales, finalmente son transformadas en un conjunto de píxeles en coordenadas enteras, la imagen. Esto es lo que hace el cauce gráfico. Siguiendo con la explicación de lo que es un fragmento, pensemos en un triángulo definido por 3 vértices. Finalmente el triángulo se transformará en un conjunto de fragmentos que se relacionan uno a uno con un píxel (figura 1.4). La diferencia semántica de fragmento y píxel para OpenGL es que un píxel es el cuadrado final que tiene un color y ocupa una posición, mientras que un fragmento es el mismo cuadrado, también con un color, pero que se encuentra en la etapa de ser procesado. Por ejemplo, en algunos casos el fragmento puede ser eliminado por lo que no producirá un píxel.

Una vez que tenemos una primera definición de los vertex y fragment shaders, vamos a crear las versiones más básicas de ambos tipos de shaders. Este es el código del vertex shader:

```
#version 450 core

void main(void)
{
}
```

Y este es el código del fragment shader:

```
#version 450 core

void main(void)
{
}
```

Se puede observar que el código es muy parecido a C o C++, con una función principal, en la que no se hace nada, pues sólo queremos que se compile, enlace y ejecute correctamente. Sólo apuntar que estamos indicando que el código que vamos a usar debe ser compatible con la versión 4.5 del lenguaje de programación GLSL, que es el que se utiliza para programar los shaders. También indicamos que queremos la funcionalidad *core*, lo que implica que queremos usar sólo las instrucciones más modernas evitando la compatibilidad con las antiguas (para ello hay que usar el perfil de compatibilidad).

Pasemos ahora a ver el código principal, en concreto, empecemos por la clase _gl_widget. Esta clase deriva de QOpenGLWidget que es la que permite dibujar con OpenGL. Son im-

Domingo Martín Perandrés©

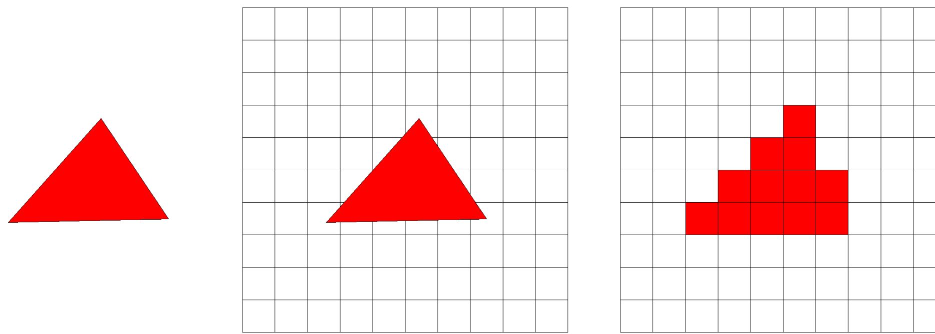


Figura 1.4: Discretización de un triángulo en los píxeles correspondientes.

portantes las funciones initializeGL, resizeGL y paintGL. Estas tres funciones son llamadas automáticamente al crearse una instancia de la clase.

La función initializeGL la usamos para inicializar OpenGL, tal y como su nombre indica. En concreto la vamos a usar para mostrar la versión e inicializar GLEW para poder acceder al resto de funcionalidad. Además, definimos el color con el que se va a limpiar la imagen.

```
void _gl_widget::initializeGL()
{
  const GLubyte* strm;

  strm = glGetString(GL_VENDOR);
  std::cerr << "Vendor: " << strm << "\n";
  strm = glGetString(GL_RENDERER);
  std::cerr << "Renderer: " << strm << "\n";
  strm = glGetString(GL_VERSION);
  std::cerr << "OpenGL Version: " << strm << "\n";

  if (strm[0] == '1'){
    std::cerr << "Only OpenGL 1.X supported!\n";
    exit(-1);
  }

  strm = glGetString(GL_SHADING_LANGUAGE_VERSION);
  std::cerr << "GLSL Version: " << strm << "\n";

  glewExperimental = GL_TRUE;
  int err = glewInit();
  if (GLEW_OK != err){
    std::cerr << "Error: " << glewGetErrorString(err) << "\n";
    exit (-1);
  }

  int Max_texture_size=0;
  glGetIntegerv(GL_MAX_TEXTURE_SIZE, &Max_texture_size);
  std::cout << "Max texture size: " << Max_texture_size << "\n";

  glClearColor(1.0,0.0,0.0,1.0);

  initialize_object();
}
```

La función initialize_object permite cargar los shaders y crear un Objeto de Atributos de Vértices (*Vertex Attribute Object*) o VAO. Es necesario crear al menos un VAO para que se pueda ejecutar cualquier shader pues el VAO es responsable de proporcionar la entrada al vertex shader.

```
void _gl_widget::initialize_object()
{
  _shaders Shader;
  Program=Shader.load_shaders("shaders/example01.vert","shaders/example01.frag");
  if (Program==0){
    exit(-1);
  }

  glCreateVertexArrays(1,&VAO);
  glBindVertexArray(VAO);

  glBindVertexArray(0);
}
```

Obsérvese como se usa una instancia de la clase _shader parar realizar la lectura, compilación y enlazado de los shaders, devolviendo un identificador del programa que se ha creado. Para la creación del VAO se observa un patrón que es común en la forma de operar de OpenGL: se crea un elemento y luego se activa mediante el enlazado (*binding*). La función glCreateVertexArrays nos devuelve un entero que identifica al VAO. La variable VAO está definida como un GLuint. Mediante la función del binding, lo que se está haciendo, para que se pueda entender, es que hay una variable global que maneja internamente OpenGL que indica en cada momento que VAO está activo, y en la misma se está poniendo el valor que se indica. De esta manera, si se hacen otras operaciones que hacen referencia al VAO ya no hay que indicarlo explícitamente sino que lo indica la variable global. La manera de indicar que no hay ningún VAO activo es poner la variable global a 0 con la instrucción del binding. En nuestro caso sólo hemos creado el VAO, no hemos hecho nada más, pero es importante insistir en que al menos debe haber un VAO definido y activo para que se pueda dibujar.

La función resizeGL es llamada cuando cambia la ventana de tamaño y también la primera vez para conocer el tamaño inicial de la ventana.

La función paintGL es la encargada de dibujar cada vez que sea necesario. Por tanto, es la que tiene que tener el código que nos permitirá dibujar usando los shaders.

Antes de ver en detalle la función de dibujado, veamos como se cargan los shaders. Para ello usamos la función load_shaders.

La función intenta cargar los shaders indicados, compilarlos y enlazarlos, devolviendo el identificador del programa que los contiene. Vayamos por pasos.

Lo primero es tener el código de los programas. Esto lo podemos hacer bien leyendo un fichero de texto o bien como una cadena de texto en el propio programa principal. Por ejemplo, si queremos el mismo código de los dos programas que hemos visto anteriormente, podríamos hacer los siguiente:

```
char *Vertex_shader_source="#version 450 core\nvoid main(void){}";
char *Fragment_shader_source="#version 450 core\nvoid main(void){}";
```

En el caso que leamos el fichero, tendremos que generar unas cadenas de caracteres a partir de los caracteres del fichero, por ejemplo, Vertex_shader_source y Fragment_shader_source, que son del tipo char *.

Ahora creamos las variables que definirán los shaders para OpenGL:

```
GLuint Vertex_shader=glCreateShader(GL_VERTEX_SHADER);
GLuint Fragment_shader=glCreateShader(GL_FRAGMENT_SHADER);
```

Como ya hemos comentado, en OpenGL es normal usar enteros sin signo como identificadores de las variables y la funcionalidad. Con el código anterior hemos creado dos identificadores, uno para un vertex shader y otro para un fragment shader.

A continuación le asignamos la cadena de caracteres que representa el código del programa:

```
glShaderSource(Vertex_shader,1,(const GLchar **) &Vertex_shader_source,NULL);
glShaderSource(Fragment_shader,1,(const GLchar **) &Fragment_shader_source,NULL);
```

A partir de aquí, podemos eliminar las cadenas de caracteres pues se ha realizado una copia. Una vez se tiene el código hay que compilarlo. Puede ocurrir que la compilación de error, igual que los programa de CPU. Por ello hay que detectarlo. El código muestra la compilación y la forma de indicar si hay algún error:

```
GLuint Vertex_shader_compiled;
GLuint Fragment_shader_compiled;

// compile
glCompileShader(Vertex_shader);
glGetShaderiv(Vertex_shader, GL_COMPILE_STATUS, &Vertex_shader_compiled);
if (Vertex_shader_compiled==GL_FALSE){
  cout << "Error compiling the vertex shader" << endl;
  return(0);
}

// compile
glCompileShader(Fragment_shader);
glGetShaderiv(Fragment_shader, GL_COMPILE_STATUS, &Fragment_shader_compiled);
if (Fragment_shader_compiled==GL_FALSE){
  cout << "Error compiling the fragment shader" << endl;
  return(0);
}
```

Si no hay ningún error, el siguiente paso es crear el programa que se conformará con el vertex shader y el fragment shader. El código compilado se adjunta al programa y después se enlaza. También es conveniente comprobar si hay algún error:

```
GLuint Program=glCreateProgram();
```

```
// Attach
glAttachShader(Program,Vertex_shader);
glAttachShader(Program,Fragment_shader);

// Link
GLuint Shaders_compiled;

glLinkProgram(Program);
glGetProgramiv(Program, GL_LINK_STATUS, &Shaders_compiled);
if (Shaders_compiled==GL_FALSE){
  cout << "Error linking" << endl;
  return(0);
}

glUseProgram(0);
```

Si no hay ningún error, ya es posible usar el programa. Para ello hay que activarlo mediante la instrucción glUseProgram indicando el identificador del programa que se quiere usar. Si se indica el valor 0 significa que se está desactivando el último programa que estuviera activo.

Una vez que ya tenemos disponible el programa con los shaders, vamos a ver en detalle los que hace la función paintGL:

```
void _gl_widget::paintGL()
{
  draw_object();
}
```

Para hacer el código más extensible declaramos una función adicional encargada de dibujar los objetos: draw_object.

```
void _gl_widget::draw_object()
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
  glUseProgram(Program);
  glBindVertexArray(VAO);

  glBindVertexArray(0);
  glUseProgram(0);
}
```

Lo que se hace es limpiar la zona de memoria (*buffer*) que contendrá la imagen, usando el color que previamente se indicó, y también el z-buffer al valor que tiene por defecto (1.0). Después se activa el programa que se va a usar, y también el VAO para a continuación desactivar el VAO y el programa. Como es fácil ver, no hay ninguna orden de dibujado, y por tanto, el programa no se va a usar. Sólo vamos a conseguir que la pantalla que veamos tenga el color que hemos indicado en el initializeGL, en este caso, el color rojo $(1,0,0,1)$. Aunque nuestro ejemplo inicial no utiliza los shaders ni dibuja nada, lo importante es que se ha compilado todo correctamente y que funciona. Además tenemos preparada la infraestructura para empezar a dibujar cosas más interesantes, lo cual haremos en el siguiente ejemplo. En la figura 1.5 se ve lo que se obtiene.

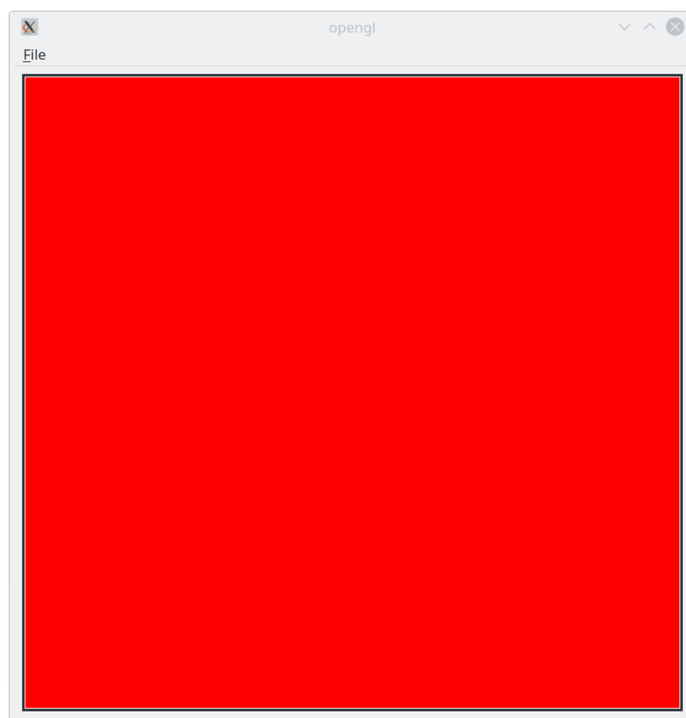


Figura 1.5: Resultado del ejemplo 1

Domingo Martín Perandrés©

2

Un punto

Una vez que hemos visto cómo crear el código y compilarlo usando shaders que no hacen nada, vamos a ver un ejemplo en el que vamos a dibujar un punto. Además, para hacerlo más sencillo, el programa en la CPU no va mandar ningún valor sino que va a ser el propio shader el que tenga la posición del punto que se va a dibujar.

OpenGL trabaja con diferentes sistemas de coordenadas, algunos definidos en 3 dimensiones y otros en 2, algunos usan valores reales y otros enteras. Los mismos se usan para pasar desde un modelo 3D a una imagen 2D.

Los sistemas de coordenadas que se pueden definir son los siguientes:

- Sistema de coordenadas del modelo: permite definir el objeto en aquella unidades que le sean más propias. Normalmente los objetos se definen en el centro para poder realizar fácilmente las rotaciones y escalados. Es un sistema de coordenadas 3D que utiliza flotantes. Es un S.C. derecho.
- Sistema de coordenadas del mundo: todos los objetos de una escena, incluidas las fuentes de luz y la cámara, se definen con respecto a este sistema de coordenadas. Normalmente los objetos tienen que ser transformados (escalados, rotados y trasladados) para que se posicionen correctamente. Define las unidades más apropiadas para la escena. Es un sistema de coordenadas 3D que utiliza flotantes. Es un S.C. derecho.
- Sistema de coordenadas de la cámara u ojo: para poder realizar la proyección de 3 dimensiones a 2 dimensiones, los objetos se tienen que definir con respecto a este sistema de coordenadas. Es un sistema de coordenadas 3D que utiliza flotantes. Es un S.C. derecho.
- Sistema de coordenadas de dispositivo normalizado: el volumen de visión se transforma a este sistema de coordenada para facilitar ciertas operaciones. Es un sistema de coordenadas 3D que utiliza flotantes. Es un S.C. derecho.

Domingo Martín Perandrés©

- Sistema de coordenadas de dispositivo: una vez se realiza la proyección y se convierten los distintos elementos en píxeles, los mismos se dibujan en este sistema de coordenadas, dependiendo de las dimensiones del *viewport*. Es un sistema de coordenadas 2D que utiliza enteros. Es un S.C. derecho.

En este ejemplo, y dado que vamos a usar los valores por defecto, nos vamos a centrar en el sistema de coordenadas del dispositivo normalizado, un paralelepípedo, volumen 3D, que tiene los siguientes valores extremos: $-1 \leq x \leq +1$, $-1 \leq y \leq +1$ y $0 \leq z \leq +1$. Esto quiere decir que los elementos que no se encuentren dentro del volumen no serán visibles.

Vamos a dibujar el punto en el centro del volumen, pero pegado al plano delantero: $(0,0,0)$. Para ello modificamos el vertex shader:

```
#version 450 core

void main(void)
{
  gl_Position=vec4(0,0,0,1);
}
```

En este código se puede ver cómo se hace uso de los tipos que tiene definidos GLSL, en particular un vector de 4 flotantes, vec4. Nos puede extrañar que hayamos dicho que vamos a dibujar un punto y que diéramos 3 coordenadas mientras que ahora incluimos una mas, cuyo valor es 1. Esto se debe, como hemos comentado anteriormente, a que internamente OpenGL usa coordenadas homogéneas que se definen con 4 valores. Normalmente este valor debe ser 1.

Además vemos como el punto en coordenadas homogéneas se copia en la variable gl_Position, la cual es una variable interna que sirve para almacenar la posición que se quiere visualizar.

Vamos a ver cómo modificamos el fragment shader:

```
#version 450 core

out vec4 frag_color;

void main(void)
{
  frag_color=vec4(0,0,0,1);
}
```

En el código se indica que el shader va a producir una salida, que se compone de 4 flotantes y que hemos llamado frag_color. Por supuesto el nombre indica lo que va a guardar, el color del fragmento. Para ello copiamos el valor $(0,0,0,1)$, el cual indica, que vamos a dibujar en color negro y con opacidad completa. La pregunta que uno se puede hacer es cual es la posición donde se usará el color indicado en el fragment shader. La respuesta está en que OpenGL, a partir de la información de la posición transformada del vértice, la cual se

guardó en gl_Position, y en función de la primitiva que se esté dibujando (puntos, líneas, triángulos), calcula el fragmento o fragmentos correspondientes a dicha primitiva, así como la posición del fragmento o fragmentos, que es lo que llega a un fragment shader, donde se va calcular el color del/los mismo/s o eliminarlo/s.

Veamos ahora como hay que modificar el código de dibujado.

```
void _gl_widget::draw_object()
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
  glUseProgram(Program);
  glBindVertexArray(VAO);

  glPointSize(10);

  glDrawArrays(GL_POINTS,0,1);

  glBindVertexArray(0);
  glUseProgram(0);
}
```

La instrucción glPointSize permite cambiar el tamaño del punto. Si no la ponemos se dibujara un punto de tamaño 1, un píxel. Para que se vea más claramente lo hemos agrandado hasta 10.

La instrucción importante es glDrawArrays(TIPO_PRIMITIVA, DESPLAZAMIENTO, NUMERO_ELEMENTOS). No vamos a entrar en todos los detalles de la misma, solo indicar que permite dibujar el tipo de primitiva que se ponga, usando los valores que comienzan con el índice del desplazamiento, y dibujando tantos elementos como se indiquen en el número de elementos. Para entenderlo, imaginemos que tenemos un vector de 25 puntos 3D y queremos dibujarlos todos, tendríamos que usar la siguiente llamada: glDrawArrays(G_POINTS,0,25). Si quisiéramos dibujar del 5 al 15 tendríamos que usar la siguiente llamada: glDrawArrays(G_POINTS,4,10).

Lo curioso es que no tenemos un vector de puntos que mandarle al shader sino que es el propio shader el que tiene las coordenadas del punto que se va a dibujar. Entonces, ¿para qué necesitamos usar la función glDrawArrays? Pues porque necesitamos que se ejecute al menos una instancia de nuestro shader. Esto es, glDrawArrays se puede usar para indicar lo que se quiere dibujar, pero también se puede ver como la forma de ejecutar el número de instancias del programa que se quiera, indicado por el número de elementos.

En nuestro caso, necesitamos que se ejecute una vez. En este caso el tipo de primitiva no es relevante.

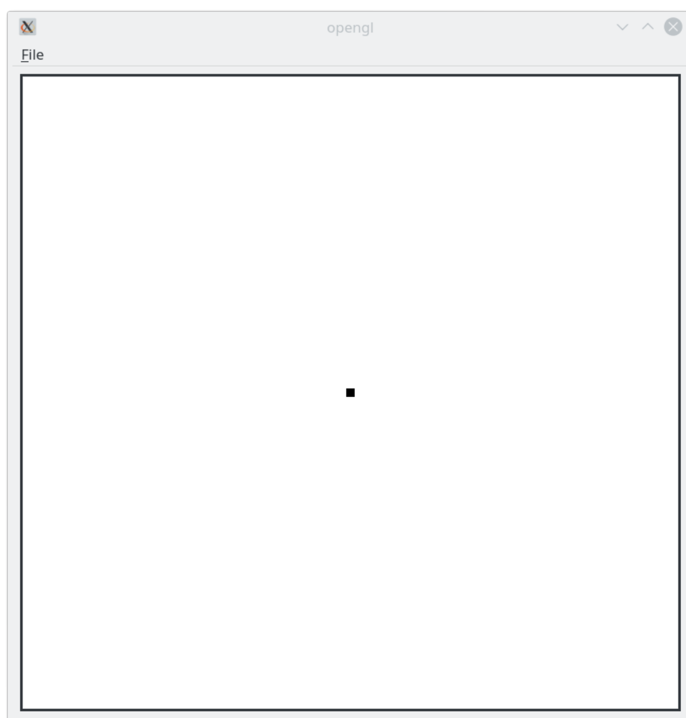


Figura 2.1: Resultado del ejemplo 2

Domingo Martín Perandrés©

3

Movemos el punto

En este ejemplo vamos a cambiar la posición del punto, pero lo vamos a hacer para que sea interactivo, respondiendo a las pulsaciones de las teclas de cursor. Esto implica otro importante cambio: es necesario mandarle la nueva posición al vertex shader, ya no puede estar fijado en el código.

Veamos el código:

```
#version 450 core

uniform vec3 position;

void main(void)
{
  gl_Position=vec4(position,1);
}
```

Hemos creado una variable de tipo vector de 3 flotantes (vec3), que se llama position. En esta variable es donde debemos meter los nuevos valores del punto. La variable tiene el modificador uniforme (uniform), el cual indica que es una variable compartida por todas las instancias del shader. Esto es, si se ejecutan 1000 instancias, todas tendrán la variable position con el mismo valor.

Obsérvese cómo se usa la variable que tiene 3 coordenadas, x, y y z, para construir la que tiene 4 y generar la posición.

El código principal hay que cambiarlo para que podamos modificar la posición. Por un lado es necesario que incluyamos el código que captura los eventos del teclado y en función de los valores de las teclas cambiar las coordenadas:

```
void _gl_widget::keyPressEvent(QKeyEvent *Keyevent)
{
```

```
  switch(Keyevent->key()){
    case Qt::Key_Left:Position.x-=0.1;break;
    case Qt::Key_Right:Position.x+=0.1;break;
    case Qt::Key_Up:Position.y+=0.1;break;
    case Qt::Key_Down:Position.y-=0.1;break;
  }

  update();
}
```

La función update le dice a Qt que cuando pueda (debe atender a numerosos eventos) actualice el contenido de la imagen, llamando a paintGL.

Para poder cambiar las coordenadas de la posición definimos una variable Position, compuesta por tres coordenadas flotantes que se direccionan con los nombres x, y y z.

Lo que se hace es copiar el valor de esta variable que está en la memoria de la CPU a la memoria de la GPU para que la pueda usar el shader. Para ello hay que conseguir la posición en la que se encuentra la variable position en el shader mediante la función glGetUniformLocation, usando como parámetros el identificador del programa que posee el shader y el nombre de la variable. La función nos devuelve un entero que indica la posición, la cual se puede usar para copiar los valores.

Veamos el código:

```
void _gl_widget::draw_object()
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
  glUseProgram(Program);
  glBindVertexArray(VAO);

  glPointSize(10);

  glUniform3fv(glGetUniformLocation(Program, "position"),1,(GLfloat *) &Position);

  glDrawArrays(GL_POINTS,0,1);

  glBindVertexArray(0);
  glUseProgram(0);
}
```

Usamos la función glUniform3fv para indicar que pasamos un puntero a un vector de 3 flotantes. El segundo parámetro indica cuantos elementos se quieren copiar. En este caso solo 1.

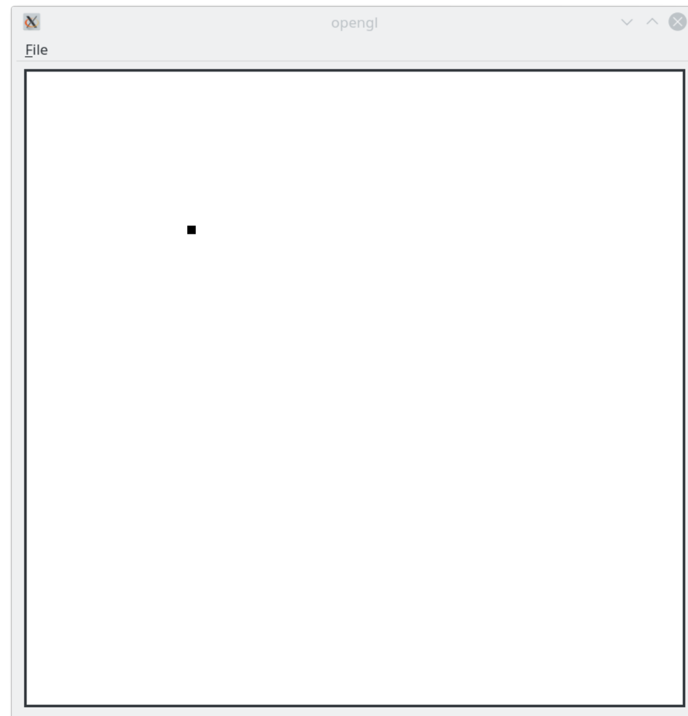


Figura 3.1: Resultado del ejemplo 3

4

Cambiamos el color

Seguimos avanzando y en este ejemplo vamos a hacer que se pueda cambiar el color del punto, eligiendo entre 8 posibilidades con los números entre 0 y 7. Vamos a pasar el color al vertex shader usando el mismo mecanismo que en el ejemplo anterior, mediante una variable de tipo uniform, y dado que el color se usa en el fragment shader, vamos a ver como se conectan las variables.

El código del vertex shader es el siguiente:

```
#version 450 core

uniform vec3 position;
uniform vec3 color;

out vec3 color_out;

void main(void)
{
  color_out=color;
  gl_Position=vec4(position,1);
}
```

Podemos ver que aparece la variable color que se compone de tres flotantes. Lo importante es que estamos declarando una variable que se llama color_out que también se compone de tres flotantes, pero que añade el atributo out. Esto significa que su valor se va a pasar a las siguientes etapas del cauce, y que en concreto, llegará a la que nos interesa, el fragment shader. La condición para que ocurra esto es que que el tipo y el nombre coincida, y tenga el atributo in en la variable del fragment shader, como ahora veremos. Para poder pasar el color le asignamos a la variable de salida el valor de la variable de entrada.

Veamos ahora el código del fragment shader:

```
#version 450 core
```

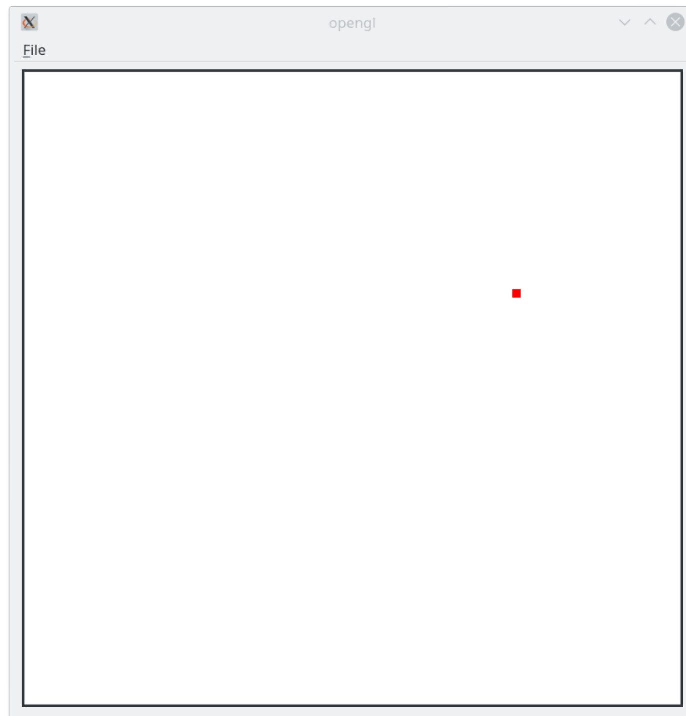


Figura 4.1: Resultado del ejemplo 4

```
in vec3 color_out;
out vec4 frag_color;

void main(void)
{
  frag_color=vec4(color_out,1);
}
```

Como se puede apreciar, llega el valor mandado por el vertex shader y lo usamos para cambiar el color. Como son tres componentes y la salida tiene cuatro, usamos el mismo mecanismo de ajuste que hemos usado con la posición.

El código para cambiar el color hace uso de un vector que se tiene predefinido, asignándolo a la variable de tres flotantes Color:

```
case Qt::Key_0:Color=_gl_widget_ne::COLORS[0];break;
case Qt::Key_1:Color=_gl_widget_ne::COLORS[1];break;
case Qt::Key_2:Color=_gl_widget_ne::COLORS[2];break;
case Qt::Key_3:Color=_gl_widget_ne::COLORS[3];break;
case Qt::Key_4:Color=_gl_widget_ne::COLORS[4];break;
case Qt::Key_5:Color=_gl_widget_ne::COLORS[5];break;
case Qt::Key_6:Color=_gl_widget_ne::COLORS[6];break;
case Qt::Key_7:Color=_gl_widget_ne::COLORS[7];break;
```

Domingo Martín Perandrés©

En el programa de dibujado tenemos que pasar el valor del color usando el mecanismo de obtener la posición como en el ejemplo anterior. Todo lo demás sigue igual.

```
glUniform3fv(glGetUniformLocation(Program, "color"),1,(GLfloat *) &Color);
```

Domingo Martín Perandrés©

Domingo Martín Perandrés©

5

Proyección paralela

En este ejemplo vamos a introducir la transformación de proyección, en concreto de una proyección paralela. Además, vamos a ver cómo se puede usar el atributo location para identificar la posición de las variables uniformes.

Hasta ahora hemos podido cambiar las coordenadas del punto que dibujamos entre unos límites muy concretos, los del dispositivo normalizado: $-1 \leq x \leq +1$, $-1 \leq y \leq +1$ y $0 \leq z \leq +1$. ¿Cómo podemos hacer para que esto cambie y tener mayor flexibilidad? Dado que queremos seguir trabajando con un ejemplo sencillo, seguiremos en dos dimensiones y sólo vamos a ver la transformación de proyección, no la transformación de la cámara, que dejamos para más adelante.

La idea básica es que nuestros modelos, nuestros objetos, nuestras escenas se definen en tres dimensiones pero los sistemas que permiten mostrar el resultado, la pantalla del ordenador, la impresora, etc, son bidimensionales. El procedimiento que pasa de tres dimensiones a dos dimensiones se llama proyección.

¡Un momento!, acabamos de decir que queremos trabajar en 2 dimensiones pero ahora resulta que el modelo es tridimensional; parece una contradicción. El problema se soluciona si pensamos que la información de un plano bidimensional está imbuida en un volumen tridimensional. La idea es que nuestros componentes de la escena están todos en un mismo plano, con la z constante. De esta manera se puede hacer la proyección, pasando de tres dimensiones a dos.

Existen dos tipos de proyección: perspectiva y paralela (figura 5.1). La proyección de perspectiva es la que nos resulta más natural pues es con la que funciona nuestro ojo o las cámaras fotográficas. Una de sus principales características es el acortamiento perspectivo: lo que está más lejos se ve más pequeño.

Domingo Martín Perandrés©

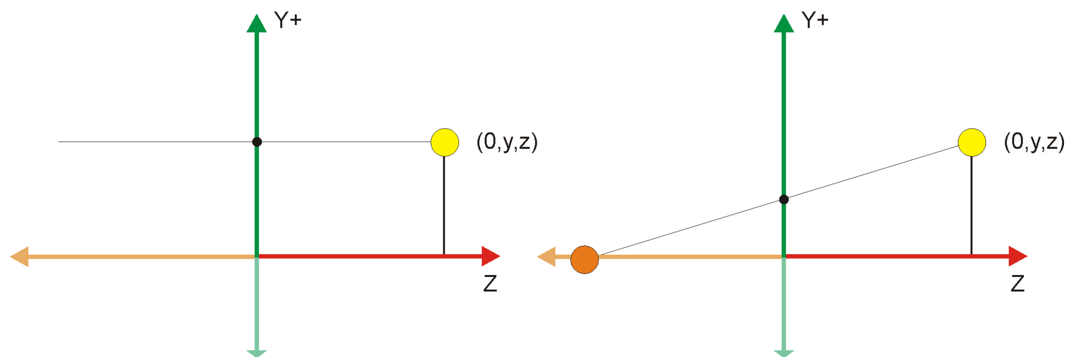


Figura 5.1: Proyección paralela y proyección de perspectiva

La proyección paralela no existe en la realidad pero es conveniente para el dibujo técnico y otros tipos de reproducciones cuando no queremos que se produzca cambio en las medidas debido a la distancia.

Una cosa muy importante de OpenGL, y otros paquetes gráficos, es que numerosas operaciones que en principio no parece que tengan relación con los elementos gráficos de la escena, finalmente vemos que el efecto se lleva a cabo mediante una transformación que se aplica sobre dichos elementos gráficos. Un ejemplo es la proyección. En ambos tipos de proyección, para poder definirlas debemos indicar el volumen de visión: todo el espacio tridimensional que podemos ver. Imaginémonos que miramos a través de una ventana. La misma limita lo que podemos ver. Ese volumen que podemos ver es el volumen de visión. Hay que fijarse que en el caso humano además de los límites definidos por los cuatro lados de la ventana, además tenemos el límite de no poder ver por detrás de nuestras cabezas y también el límite en cuanto a lo que somos capaces de ver en la distancia. En el caso de usar un ordenador, no tenemos ventana física pero la podemos representar por un rectángulo. Además tenemos que definir un plano que nos limite lo que es visible por delante, y otro plano que nos limite lo que ya no es visible por detrás. Para definir el rectángulo necesitamos 4 valores, las coordenadas de las esquinas inferior izquierda y la superior derecha; el plano delantero y el trasero son 2 valores más. Por tanto necesitamos 6 valores para definir el volumen de visión. En la figura 5.2 se puede ver un ejemplo de una proyección paralela.

Bien, eso es lo que vamos a hacer en este ejemplo, definir una proyección paralela, mediante un volumen de visión. En concreto vamos a definir una ventana de coordenadas $x_{min} = -1$, $x_{max} = 1$, $y_{min} = -1$, $y_{max} = 1$. Y para el plano delantero le daremos el valor 0 y para el plano trasero el valor 1. El efecto es que cualquier punto que tenga como coordenadas $-1 \leq x \leq +1$, $-1 \leq y \leq +1$ y $0 \leq z \leq +1$ se podrá ver. En otro caso no. Además, tal y como hemos definido la proyección, al punto (x,y,z) le corresponderá el punto proyectado (x,y).

¡Pero con esto no hemos conseguido nada, son las mismas dimensiones que las del dispositivo normalizado! Sí, pero sólo para el ejemplo. Ahora ya podemos poner las dimensiones que más nos interesen. Lo veremos más adelante con un ejemplo.

Domingo Martín Perandrés©

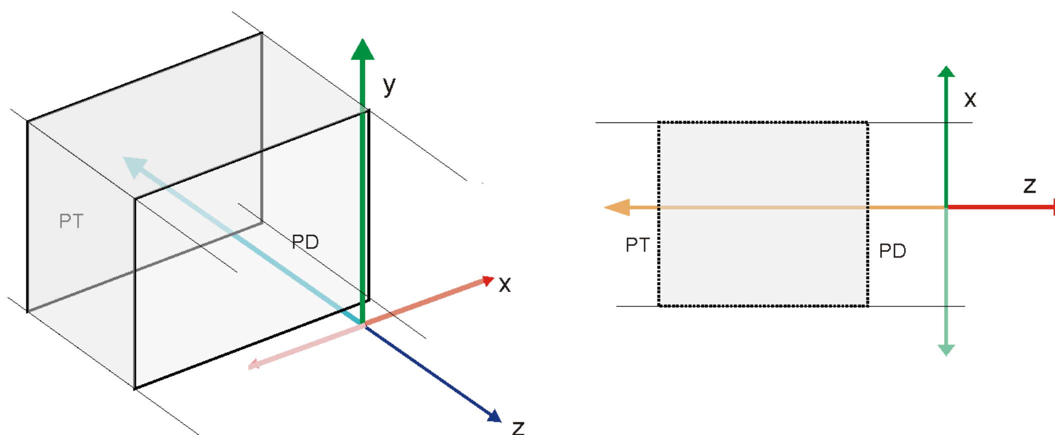


Figura 5.2: Volumen de visión de una proyección paralela

Probablemente todavía quede pendiente la pregunta de cómo afecta la definición del volumen de visión para que se realice la proyección de los objetos que hay en la escena. La respuesta es que los parámetros que definen el volumen de visión sirven para calcular una trasformación que al aplicarse sobre las posiciones de los elementos de la escena produce la proyección. Las transformaciones en OpenGL se representan mediante matrices de 4x4 (recuérdese que trabaja en coordenadas homogéneas). Las matrices representan las transformaciones, y la transformación es la multiplicación de la matriz por el vector que representan la posición de cada vértice del modelo. En nuestro caso, nuestro modelo es un solo punto, con coordenadas $(0,0,0)$ que se convierte en $(0,0,0,1)$ al pasarlo a coordenadas homogéneas. Si ahora tenemos una matriz de 4x4, sólo nos hace falta saber multiplicar vectores por matrices. OpenGL sabe hacerlo. El resultado es otro vector que representa la posición transformada. Matemáticamente nos queda $\vec{v}' = \vec{v} \cdot M$. Cuidado que en OpenGL se usan matrices definidas por columnas (*column major*) y por tanto se multiplican las matrices por vectores columna: $\vec{v}'^T = M \cdot \vec{v}^T$.

Empecemos por ver el código del vertex shader:

```
#version 450 core

layout (location=0) uniform mat4 matrix;
layout (location=1) uniform vec3 position;
layout (location=2) uniform vec3 color;

out vec3 color_out;

void main(void)
{
  color_out=color;
  gl_Position=matrix*vec4(position,1);
}
```

Podemos ver que se ha añadido el identificador layout que indica que se van definir características de la variable. En nuestro caso concreto usamos el modificador position para indicar el número de índice que queremos para cada variable. De esta forma nos ahorramos tener que preguntar cada vez que se quiera cambiar el valor como se hacia en los ejemplos anteriores.

Además podemos ver cómo se define una matriz de 4x4 de flotantes mediante mat4 a la que llamamos matrix. Aquí se puede ver cómo la matriz postmultiplica el punto, tal y como hemos indicado. El resultado es el punto transformado, en este caso, proyectado.

Para calcular la matriz usamos una clase de matrices de Qt (existen otras bibliotecas). El código que calcula la matriz y que copia los valores es el siguiente. Obsérvese que como ya conocemos los índices de las variables sólo hace falta que los usemos.

```
void _gl_widget::draw_object()
{
  QMatrix4x4 Parallel_projection;
  Parallel_projection.ortho(-1,1,-1,1,0,1);

  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
  glUseProgram(Program);
  glBindVertexArray(VAO);

  glPointSize(10);

  glUniformMatrix4fv(0,1,GL_FALSE,Parallel_projection.data());
  glUniform3fv(1,1,(GLfloat *) &Position);
  glUniform3fv(2,1,(GLfloat *) &Color);

  glDrawArrays(GL_POINTS,0,1);

  glBindVertexArray(0);
  glUseProgram(0);
}
```

La función que crea la configuración de una matriz de proyección paralela se llama ortho.

Domingo Martín Perandrés©

6

¡Tres dimensiones!

Una vez que hemos visto los mecanismos básicos para dibujar, vamos a pasar a las 3 dimensiones, espacio para el que se diseñó OpenGL. El principal cambio es que vamos a tener que introducir la cámara que permita observa la escena. Para poder apreciar las tres dimensiones vamos a definir unos ejes de coordenadas que se dibujarán con tres líneas de color rojo, verde y azul para los ejes x, y y z.

Lo primero que tenemos que tener en cuenta, y es muy importante, es que la cámara virtual realiza dos tareas bien diferenciadas: el proceso de proyección por el que se pasa de información tridimensional a información bidimensional, y la colocación de los objetos para ser observados por la cámara. Veámoslo con una comparación con el mundo real.

Imaginemos que un conocido ha visitado un lugar muy bonito y ha hecho fotografías. Quiere que nosotros compartamos su experiencia y podamos hacer fotografías similares: ¿Qué información nos tendrá que dar? Si lo pensamos un poco deduciremos que es la siguiente:

- La posición donde se colocó para sacar la fotografía
- La dirección en la cual apuntó la cámara
- La orientación de la cámara, vertical, apaisada o cualquier otro ángulo
- Si tiene una lente zoom, la apertura de la misma

Al hacer la foto, la luz que entra por la lente llegará al sensor y se almacenará como una imagen de píxeles RGB (realmente tanto el proceso como el formato son más complejos).

En el mundo virtual, nuestra cámara debe hacer lo mismo, pero de una forma simulada, con todos los objetos y entidades representados como números y ecuaciones: un poco más complicado que coger la cámara, apuntar y disparar.

Domingo Martín Perandrés©

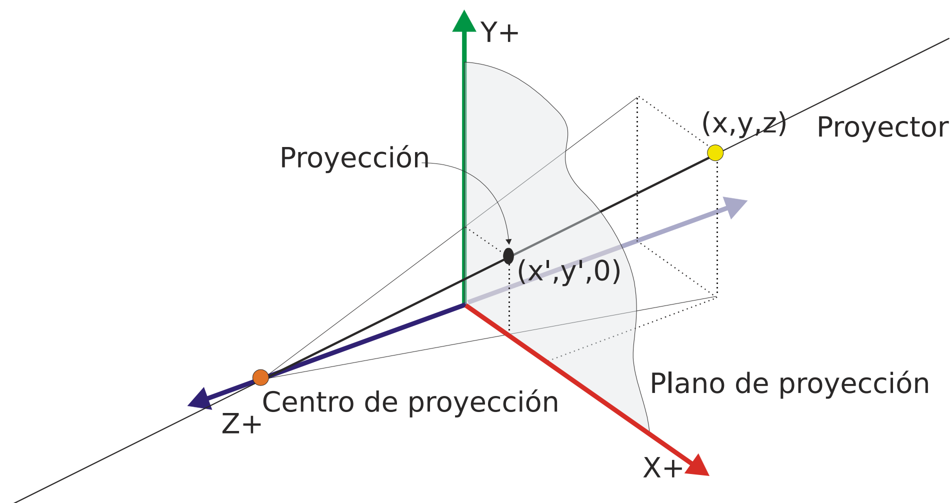


Figura 6.1: Proyección de perspectiva

Dado que queremos obtener resultados con nuestra cámara virtual lo antes posible, vamos a empezar por explicar en qué consiste y como se realiza el proceso de proyección, cuando la cámara no se mueve. Después, una vez nuestra cámara pueda hacer fotografías, veremos cómo podemos moverla a cualquier sitio y orientarla de cualquier manera. El primer paso es la proyección y el segundo el posicionamiento.

- Proyección de perspectiva
  La proyección es el proceso para pasar de 3 dimensiones a 2 dimensiones (en general, de n dimensiones a $n-1$ dimensiones). En el tema anterior hemos visto la proyección paralelela, cuya característica principal es que todos los proyectores, las líneas rectas que emanan de los vértices, son parelelos y por tanto, intersecan en el infinito. Ahora estamos interesados en ver la proyección de perspectiva. En este caso, los proyectores son líneas rectas que unen cada vértice con otro punto llamado centro de proyección, CP. Dada una superficie, en nuestro caso un plano llamado plano de proyección, la proyección es la intersección de los proyectores con el plano de proyección (figura 6.1).

  Obsérvese que la cámara virtual que estamos modelando es del tipo *pin hole*, esto es, que no tiene lente para enfocar. Al tener que pasar todos los proyectores por un punto, todos los elementos de la escena, desde los más cercanos a los más lejanos están enfocados. Una de las características de la cámara de OpenGL es que la misma es fija y su centro de proyección está colocado en el origen del sistema de coordenadas de la cámara. Para poder terminar de definir el volumen de visión, el frustum, necesitamos dar los valores de la ventana, la distancia con respecto al origen del plano delantero y la distancias con respecto al origen del plano trasero. Los planos de corte deben cumplir que estén delante del centro de proyección, el origen, y para poder generar un volumen válido, que la distancia del plano trasero sea mayor o igual que la del plano delantero. Los planos de corte, como indica su nombre, se encargan de eliminar la información que está por delante del plano delantero y por detrás del plano trasero. Para terminar de

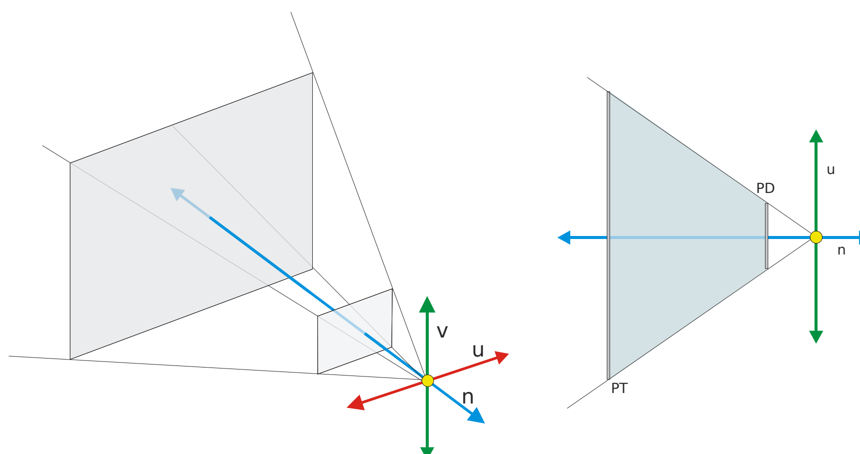


Figura 6.2: Frustum

delimitar el volumen de visión nos hace falta definir la ventana, una zona rectangular que se inscribe en el plano delantero. Se define mediante dos puntos en 2 dimensiones: la posición inferior izquierda y la posición superior derecha. Si proyectamos cada uno de los lados de la ventana usando el centro de proyección, nos permite terminar de definir el volumen de visión de la proyección de perspectiva también llamado frustum (figura 6.2).

Por tanto, al igual que el caso de la proyección paralela, para la definición de la proyección de perspectiva se utilizan 6 parámetros y, normalmente, la función se llama frustum.

Tal y como hemos dicho, la proyección es el conjunto de puntos que resultan de la intersección de los proyectores con el plano de proyección. Si hemos entendido correctamente la forma de hacer la proyección paralela, habremos comprendido que consiste en modificar las coordenadas de los vértices mediante una transformación representada mediante una ecuación. La transformación la hemos definido mediante una matriz de 4×4 que multiplica al vector creado con las coordenadas del vértice, y la aplicación de la trasformación consiste en multiplicar la matriz por el vector. Por tanto, para proyectar se multiplican las coordenadas de los vértices por una matriz que realiza la proyección.

Si quisiéramos implementar la proyección tendríamos que crear el siguiente código:

```
QMatrix4x4 Projection;

Projection.frustum(X_MIN,X_MAX,Y_MIN,Y_MAX,FRONT_PLANE_PERSPECTIVE,↩
    BACK_PLANE_PERSPECTIVE);
```

y pasar los valores de la matriz al vertex shader para que calcule la proyección a cada vértice. Podemos probar a visualizar el punto que hemos usado en los ejemplos

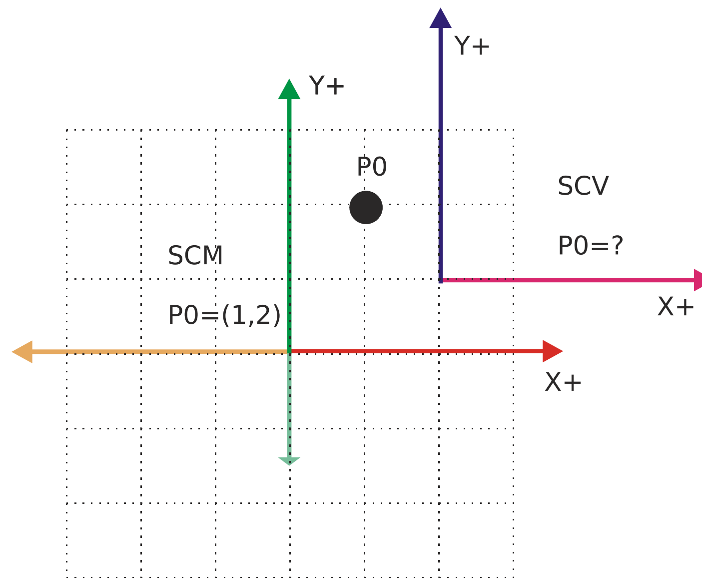


Figura 6.3: Dos sistemas de coordenadas.

anteriores. Es probable que no se vea nada. El problema es que el punto, al estar en el plano del origen, $z = 0$, no está dentro del volumen de visión. Esto nos lleva al siguiente problema que es el posicionamiento de la cámara.

- Posicionamiento
  La idea de posicionar correctamente la cámara para que realice la captura de la zona deseada es sencilla en el caso real. En el caso virtual es un poco más complejo. La idea más importante para realizar el posicionamiento es que hay que realizar un cambio de sistema de coordenadas para pasar los vértices de su definición en el Sistema de Coordenadas de Mundo al Sistema de Coordenadas de Vista.

  Para poder pasar de un sistema de coordenadas a otro hay que realizar transformaciones geométricas. Para una mejor comprensión de lo que viene a continuación es conveniente entender lo que son, aunque sea de manera superficial. Para ello revisa el tema 11.

  Veamos un ejemplo sencillo mediante un caso en dos dimensiones (figura 6.3). El punto P_0 está dado en coordenadas del Sistema de Coordenadas de Mundo, SCM. Los parámetros que definen a la vista (también llamada cámara u ojo), permiten crear el Sistema de Coordenadas de Vista, SCV. La pregunta es, si las coordenadas de P_0 con respecto al SCM son $(1,2)$, ¿cuales serán las coordenadas desde el SCV? La respuesta es fácil de ver: $(-1,1)$. La solución implica un cambio del sistema de referencia para el punto P_0, esto es, un cambio de sistema de coordenadas.

  La idea del cambio de sistema de coordenadas es simple: colocamos uno de los sistemas de coordenadas en la misma posición y con la misma orientación que el otro. Para ello tenemos que aplicar una serie de transformaciones, que normalmente consisten en rotaciones y traslaciones. En nuestro ejemplo, la transformación para alinear el

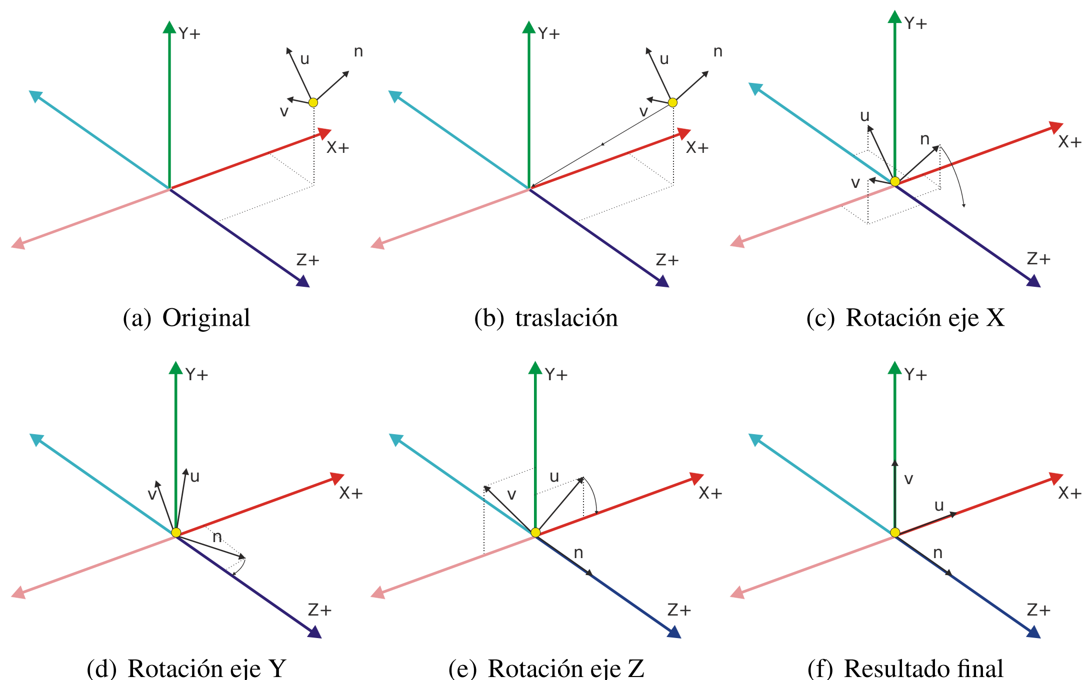


Figura 6.4: Proceso para obtener la transformación de vista.

SCV con el SCM es una traslación de $(-2,-1)$. Esto hace que los orígenes estén en la misma posición y dado que los ejes se alinean perfectamente no es necesaria ninguna rotación.

Una vez que sabemos el conjunto de transformaciones que superpone ambos sistemas de coordenadas, ya tenemos la transformación que permite pasar de un sistema a otro, la llamada transformación de vista, TV (en OpenGL, VIEW). Si aplicamos la TV a un punto en definido en el SCM nos pasa sus coordenadas al SCV. Vamos a comprobarlo. Apliquemos la traslación $(-2,-1)$ al punto $(1,2)$, obteniendo $(-2+1,-1+2) = (-1,1)$. Justo lo que habíamos deducido visualmente.

En nuestro caso estamos trabajando en 3 dimensiones y mediante los parámetros que definen a la cámara se crea un sistema de coordenadas tridimensional, el SCV. Pero ahora tenemos que superponer un sistema en 3 dimensiones, el SCV, con otro en 3 dimensiones, el SCM. Para ello son necesarias, en el caso más general, una traslación que haga coincidir ambos orígenes seguida de tres rotaciones sobre los ejes x, y y z, en el orden que el usuario quiera pero que sea coherente para conseguir el resultado deseado.

Los pasos para crear la TV se pueden ver en la figura 6.4.

Una vez que hemos visto las dos componentes de la cámara, vamos a implementar una muy sencilla: siempre mira al origen del SCM, y se mueve en la superficie de una esfera, con la dirección hacia arriba siendo la vertical, aplicando una proyección de perspectiva.

Domingo Martín Perandrés©

La transformación que representa a la cámara es otra matriz de 4×4. En este ejemplo vamos a implementar una cámara que se mueve en la superficie de una esfera, definida por el *radio*, y dos ángulos: α para el giro sobre el eje y y β, para el giro sobre el eje x (estamos utilizando coordenada esféricas). Para poder modificar dichos parámetros vamos a usar las teclas de cursor para los ángulos y avanza y retrocede página para aumentar y disminuir el radio.

Para definir la transformación completa que implementa la cámara con el posicionamiento y la proyección usamos el siguiente código:

```
QMatrix4x4 Projection;
QMatrix4x4 Rotation_x;
QMatrix4x4 Rotation_y;
QMatrix4x4 Translation;

Projection.frustum(X_MIN,X_MAX,Y_MIN,Y_MAX,FRONT_PLANE_PERSPECTIVE,BACK_PLANE_PERSPECTIVE↩
    );

Rotation_x.rotate(Angle_x,1,0,0);
Rotation_y.rotate(Angle_y,0,1,0);
Translation.translate(0,0,-Distance);

Projection*=Translation;
Projection*=Rotation_x;
Projection*=Rotation_y;
```

Para ver más claro el proceso hemos dejado los pasos individuales de la transformación de la cámara: una rotación con respecto al eje x, una rotación con respecto al eje y y una traslación. Primero definimos las transformaciones y luego las combinamos, o concatenamos, multiplicándolas. Es muy importante el orden porque, en general, el producto de matrices no es conmutativo. Y otra cosa que debemos recordar, es que OpenGL pos-multiplica: dados la matriz M y vértice v, el nuevo vértice se calcula como :$v' = M \cdot v^T$. Por tanto, y si tenemos en cuenta el orden temporal en el que se deben aplicar, primero será la rotación con respecto a y, luego con respecto a x, después la traslación y finalmente la proyección.

Podemos ver que dada la forma en la que se multiplican las matrices, la codificación implica invertir el orden. Al final, la matriz Projection contiene lo siguiente $P = P \cdot T \cdot R_x \cdot R_y$. Esto es lo que mandamos al shader.

Otro aspecto interesante a tener en cuenta es observar que la cámara virtual es una transformación que se aplica a los vértices de los objetos, haciendo que se coloquen y orienten de la manera esperada. En la realidad, normalmente movemos la cámara con respecto a los objetos (en muchos casos es la única opción, pensemos en un edificio, por ejemplo), en nuestro caso, movemos los objetos con respecto a la cámara. El resultado es el mismo (figura 6.5).

El código que permite cambiar los valores de los parámetros de la cámara es siguiente:

```
void _gl_widget::keyPressEvent(QKeyEvent *Keyevent)
{
switch(Keyevent->key()){
```

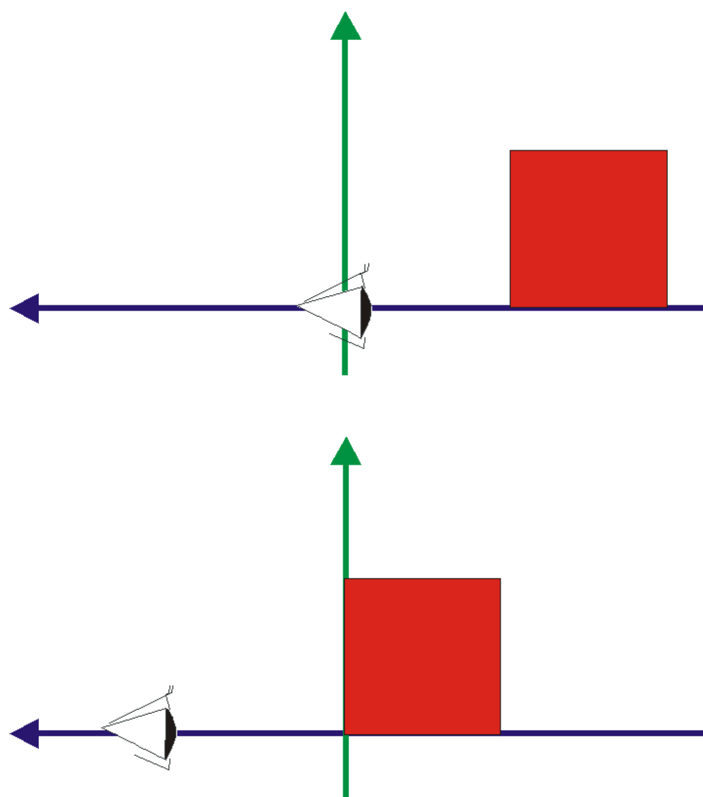


Figura 6.5: Relatividad en el posicionamiento: da lo mismo mover el ojo y dejar el objeto fijo, que dejar el ojo fijo y mover el objeto.

```
case Qt::Key_Left:Angle_y-=ANGLE_STEP;break;
case Qt::Key_Right:Angle_y+=ANGLE_STEP;break;
case Qt::Key_Up:Angle_x-=ANGLE_STEP;break;
case Qt::Key_Down:Angle_x+=ANGLE_STEP;break;
case Qt::Key_PageUp:Distance+=DISTANCE_STEP;break;
case Qt::Key_PageDown:
Distance-=DISTANCE_STEP;
if (Distance<DISTANCE_STEP) Distance=DISTANCE_STEP;
break;
}

update();
}
```

Con esto hemos resuelto el problema de la cámara pero nos encontramos con otro. Hasta ahora sólo hemos dibujado un punto, pero para dibujar los ejes necesitamos como mínimo 6 puntos, pues usamos dos puntos extremos para cada eje. ¿Cómo hacemos para mandar varios puntos a los shaders, en nuestro caso, al vertex shader? La respuesta está en el uso de los *Vertex Buffer Objects*, VBOs. ¿Qué es un VBO? Básicamente lo podemos ver como un array o vector, sólo que para ser usado en la GPU.

En nuestro caso, y como primera aproximación, se nos ocurre que necesitamos 2 vectores: uno para almacenar las posiciones de los puntos extremos, y otro para almacenar los

colores de los ejes. Para el primer vector está claro que necesitamos que pueda almacenar 6 posiciones en el espacio. Para los colores, podríamos pensar que con 3 elementos sería suficiente. Dado que vamos a usar la función glDrawArrays, esto implica que tiene que haber una correspondencia uno a uno entre los distintos datos. En este caso, si definimos 6 vértices necesitaremos 6 colores, aunque se repitan por parejas. Por ejemplo, si queremos dibujar el eje de las equis en color rojo, habrá que definir las dos posiciones para los vértices de los extremos, $(-x,0,0)$ y $(x,0,0)$, los colores serán $(1,0,0)$ y $(1,0,0)$. Ya veremos el por qué de esto más adelante (Tema 8).

Centrémonos ahora en ver cómo vamos a crear los vectores y rellenarlos de datos. Lo primero es la creación de los datos de los ejes. Para ello usaremos esta función:

```
void _gl_widget::initialize_axis_data(vector<_vertex3f> &Axis_vertices, vector<_vertex3f>↩
     &Axis_colors)
{
  Axis_vertices.resize(6);
  Axis_vertices[0]=_vertex3f(MAX_AXIS_SIZE,0,0); // x
  Axis_vertices[1]=_vertex3f(-MAX_AXIS_SIZE,0,0);
  Axis_vertices[2]=_vertex3f(0,MAX_AXIS_SIZE,0); // y
  Axis_vertices[3]=_vertex3f(0,-MAX_AXIS_SIZE,0);
  Axis_vertices[4]=_vertex3f(0,0,MAX_AXIS_SIZE); // z
  Axis_vertices[5]=_vertex3f(0,0,-MAX_AXIS_SIZE);

  Axis_colors.resize(6);
  Axis_colors[0]=_vertex3f(1,0,0);// red
  Axis_colors[1]=_vertex3f(1,0,0);
  Axis_colors[2]=_vertex3f(0,1,0);// green
  Axis_colors[3]=_vertex3f(0,1,0);
  Axis_colors[4]=_vertex3f(0,0,1);// blue
  Axis_colors[5]=_vertex3f(0,0,1);
}
```

Se puede ver que mandamos dos vectores por referencia para que sean inicializados en su tamaño y rellenados con los datos que necesitamos. Una vez que tenemos los datos veamos cóm se usan:

```
void _gl_widget::initialize_object()
{
  _shaders Shader;

  Program=Shader.load_shaders("shaders/example_tres_dimensiones.vert","shaders/↩
      example_tres_dimensiones.frag");
  if (Program==0){
    exit(-1);
  }

  vector<_vertex3f> Axis_vertices;
  vector<_vertex3f> Axis_colors;

  initialize_axis_data(Axis_vertices,Axis_colors);

  glCreateVertexArrays(1,&VAO);
  glBindVertexArray(VAO);

  // posiciones
  glCreateBuffers(1,&VBO_vertices);
```

```
  glNamedBufferStorage(VBO_vertices,Axis_vertices.size()*3*sizeof(float),&Axis_vertices↩
      [0],GL_MAP_WRITE_BIT);
  glVertexArrayVertexBuffer(VAO,1,VBO_vertices,0,3*sizeof(float));
  glVertexArrayAttribFormat(VAO,1,3,GL_FLOAT,GL_FALSE,0);
  glEnableVertexArrayAttrib(VAO,1);

  // colores
  glCreateBuffers(1,&VBO_colors);
  glNamedBufferStorage(VBO_colors,Axis_colors.size()*3*sizeof(float),&Axis_colors[0],↩
      GL_MAP_WRITE_BIT);
  glVertexArrayVertexBuffer(VAO,2,VBO_colors,0,3*sizeof(float));
  glVertexArrayAttribFormat(VAO,2,3,GL_FLOAT,GL_FALSE,0);
  glEnableVertexArrayAttrib(VAO,2);

  glBindVertexArray(0);
}
```

Para entender lo que estamos haciendo debemos ir paso a paso. En primer lugar creamos un identificador para un VBO. Esto lo hacemos con la instrucción glCreateVertexArrays. Los parámetros que mandamos son el número de identificadores que queremos obtener, y la dirección de la variable o vector de variables donde se van a devolver los identificadores. Estos identificadores, tal y como hemos visto ya, son enteros sin signo, GLuint. Como medida de control deberíamos controlar que se nos devuelve un valor válido. En nuestro ejemplo lo asumimos.

Lo siguiente que hacemos es activar el VAO haciendo su correspondiente binding. Esto implica que todo lo que hagamos a continuación va a "pertenecer" al VAO que hemos activado. En concreto, los VBOs que vamos a crear, rellenar y activar, van a pertenecer al VBO activo. Lo interesante de esta forma de operar, y por eso se creó, es que cuando se activa un VAO implica la activación automática de todos los VBOs asociados. En las primeras versiones esta activación había que hacerla manualmente, mediante código, lo cual era muy tedioso. Obsérvese como al final desactivamos el VAO y sus datos asociados mediante un binding al valor 0.

Lo siguiente que podemos ver se aplica tanto para el VBO de las posiciones como el de los colores. Dado que son iguales sólo comentaremos las llamadas del primero.

La primera llamada, y siguiendo el patrón que ya hemos visto en numerosas ocasiones, consiste en crear un buffer mediante glCreateBuffers. Los parámetros son los mismos: número de variables que se quieren crear y la dirección donde almacenar los valores.

La llamada glVertexArrayVertexBuffer se encarga de reservar el espacio, y también es posible inicializarlo, tal y como hacemos en este caso. Como se ve, el primer parámetro es el identificador del VBO. El segundo indica las necesidades de espacio ¡en bytes! En nuestro caso tenemos 6 puntos por 3 coordenadas por el tamaño que ocupe cada coordenada que están definidas como flotantes. El tercer parámetro es la dirección donde comienzan los datos que se quieren copiar. Si el valor es NULL implica que no se copia nada. Por último le damos una indicación a OpenGL de cual va a ser el uso del VBO. En nuestro caso estamos indicando que vamos a poder escribir. Todas las posibilidades se pueden enconrar en el manual de la función.

A continuación vamos a hacer una asignación del VBO que hemos creado a un VAO, y además identificamos la posición o slot en la que se va a colocar dentro del VAO mediante la llamada glVertexArrayVertexBuffer. El primer parámetro es el VAO. El segundo el índice o slot dentro del VAO. El tercero el identificador del VBO. El cuarto parámetro nos indica que posición dentro del buffer debemos considerar como la primera. En nuestro caso la posición inicial, 0. El último parámetro indica la distancia entre elementos en el buffer. Estamos indicando que las coordenadas de un punto se componen de 3 flotantes.

La llamada glVertexArrayAttribFormat se encarga de indicar el formato de los datos. El primer parámetro es el VAO al que pertenece. El segundo el índice o slot. El tercero nos indica el número de componentes de cada objeto o componente por vértice. Hemos indicado que hay 3, correspondiente a las tres coordenadas. Después indicamos el tipo de las componentes, en este caso GL_FLOAT. Y por último indicamos la distancia entre elementos en el buffer.

Por último habilitamos el VBO mediante la llamada glEnableVertexArrayAttrib, indicando el VAO y el índice del VAO.

Con esto hemos conseguido crear el VAO y sus VBOs asociados. Además hemos metido los datos de las posiciones y de los colores. Sólo nos hace falta ver cómo se puede usar estos buffers dentro de los shader. El vertex shader es el siguiente:

```
#version 450 core

layout (location=0) uniform mat4 matrix;
layout (location=1) in vec3 vertex;
layout (location=2) in vec3 color;

out vec3 color_out;

void main(void)
{
  color_out=color;
  gl_Position=matrix*vec4(vertex,1);
}
```

Como se puede apreciar, recibe la matriz que permite transformar a los vértices, los vértices y los colores. Es importante observar que la matriz es la misma para todos los vértices y por eso se declara de tipo uniform.

Lo que aparece nuevo es que hemos indicado que tenemos variables de entrada, vertex y color, las cuales, además tienen asociada una posición mediante el uso de la instrución layout en combinación con location. Aquí viene un concepto importante para entender el funcionamiento del vertex shader: un vertex shader sólo trabaja con los datos de un vértice, si necesito operar con varios vértices necesitaré que se ejecuten tantas instancias del shader como vértices tenga. ¿Quién se encarga de hacer que se ejecuten todas las instancias necesaria? La llamada glDrawArrays con los parámetros adecuados.

Domingo Martín Perandrés©

Para dejarlo más claro, con el código que hemos expuesto, lo que hemos hecho es inicializar y rellenar con datos los VBOs en la GPU. En concreto tenemos las coordenadas de 6 vértices y las componentes de 6 colores. Necesitamos que se dibujen las 3 líneas de los ejes. Para ello la llamada es la siguiente:

```
void _gl_widget::draw_object()
{
  QMatrix4x4 Projection;
  QMatrix4x4 Rotation_x;
  QMatrix4x4 Rotation_y;
  QMatrix4x4 Translation;

  Projection.frustum(X_MIN,X_MAX,Y_MIN,Y_MAX,FRONT_PLANE_PERSPECTIVE,↩
      BACK_PLANE_PERSPECTIVE);

  Rotation_x.rotate(Angle_x,1,0,0);
  Rotation_y.rotate(Angle_y,0,1,0);
  Translation.translate(0,0,-Distance);

  Projection*=Translation;
  Projection*=Rotation_x;
  Projection*=Rotation_y;

  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
  glUseProgram(Program);
  glBindVertexArray(VAO);

  glUniformMatrix4fv(0,1,GL_FALSE,Projection.data());

  glDrawArrays(GL_LINES,0,6);

  glBindVertexArray(0);
  glUseProgram(0);
}
```

La llamada importante (las demás ya sabemos qué hacen) es glDrawArrays(GL_LINES,0,6): estamos pidiendo que se interpreten los 6 puntos (comenzando en la posición 0) como líneas, lo cual implica que cada dos vértices conformarán un eje. Al decir que se van a usar 6 vértices estamos diciendo que se ejecuten 6 instancias del vertex shader.

Como hemos activado los VBOs y además, en el código del vertex shader hemos indicado que vamos a recibir dicha información, ahora se puede entender que OpenGL lo que va a hacer es mandar los datos de un vértice a cada instancia del vertex shader. Si numeráramos los vertex shaders, el número 0 recibirá los datos de vértice 0 y del color 0. El vertex shader 1 recibirá los datos del vértice 1 y del color 1, y así con todos los demás. ¿Donde estarán dichos valores? Pues en las variables vertex y color, por supuesto.

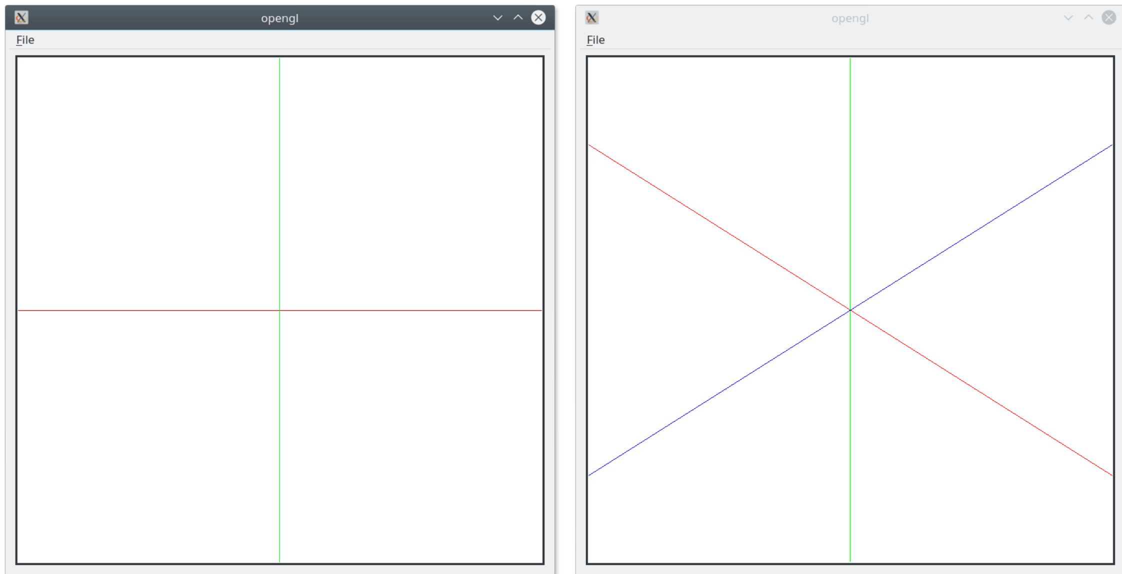


Figura 6.6: Resultado del ejemplo 6 con la cámara en su posición original y después de rotar alrededor del origen

7

Un triángulo

Una vez que tenemos nuestra escena 3D funcionando y mediante los ejes podemos posicionarnos vamos a introducir un nuevo e importante tipo de primitiva: el triángulo. De hecho, en OpenGL casi todo se construye con triángulos. Esto es así porque es el polígono más sencillo, tiene superficie, y por sus propiedades para ser visualizado, que comentaremos más adelante. Si queremos representar un objeto que tiene superficie debemos hacerlo mediante triángulos.

Su definición es muy sencilla: mediante 3 vértices, y por supuesto, habrá que decirle a OpenGL que interprete los 3 vértices como un triángulo, no como 3 vértices independientes, o una línea (el tercer vértice no se usaría).

En este ejemplo se plantea otro problema pues se van a dibujar dos objetos: los ejes y el triángulo. ¿Cómo lo hacemos teniendo encuenta lo que hemos aprendido respecto a los VAOs y los VBOs?

En principio, se me ocurren las siguiente posibilidades:

- Solución 1: usar 1 VAO, 1 programa y 2 VBO con reescritura.
- Solución 2: usar 2 VAOs, 2 programas y 4 VBO.
- Solución 3: usar 2 VAOs, 1 programa y 4 VBO.
- Solución 4: usar 1 VAO, 2 programas y 4 VBO.
- Solución 5: usar 1 VAO, 1 programa y 4 VBO con intercambio.
- Solución 6: usar 1 VAO, 1 programa y 2 VBO con intercalado.

Vamos a estudiar las ventajas y desventajas de cada método.

Domingo Martín Perandrés©

Solución 1

Se usan los 2 VBO para posiciones y colores, y antes de dibujar los ejes se copian los valores de los ejes, y antes de dibujar el triángulo se copian los valores del triángulo. Esto es, cada vez que se dibuja la escena completa, se cambian los contenidos de los VBOs. Esto es sencillo de hacer y para nuestro ejemplo no notaremos ningún problema, pero pensemos que vamos a dibujar 2 objetos y que cada unos está formado por un millón de vértices. Es fácil ver lo poco óptimo que es, y mucho más si en vez de 2 tenemos cientos o miles de objetos de un gran tamaño.

Solución 2

Para esta segunda solución pensé que dado que eran dos objetos lo mejor era tener 2 programas y 2 VAOs, con la idea de poder usar el mismo procedimiento para los distintos objetos que se pudieran crear. Dada mi creencia original de que con cada VAO iban asociados los VBOs que se definían y que no podían ocupar los mismos puertos (*bindings*) pensé que la solución era que cada shader tuviera los VBOs en posiciones diferentes.

Pongo el vertex shader de los ejes:

```
#version 450 core

layout (location=0) uniform mat4 matrix;
layout (location=1) in vec3 vertex;
layout (location=2) in vec3 color;

out vec3 color_out;

void main(void)
{
  color_out=color;
  gl_Position=matrix*vec4(vertex,1);
}
```

Y del triángulo:

```
#version 450 core

layout (location=0) uniform mat4 matrix;
layout (location=3) in vec3 vertex;
layout (location=4) in vec3 color;

out vec3 color_out;

void main(void)
{
  color_out=color;
  gl_Position=matrix*vec4(vertex,1);
}
```

Domingo Martín Perandrés©

Se puede ver que los nombre de los atributos son iguales pero no los puntos de enlace. Esto implica que hay 4 VBO que deben ser inicializados con los datos correspondientes de los ejes y del triángulo.

Para dibujar hay que hacer lo siguiente:

```
// axis
glUseProgram(Program1);
glBindVertexArray(VAO1);

glUniformMatrix4fv(0,1,GL_FALSE,Projection.data());

glDrawArrays(GL_LINES,0,6);

glBindVertexArray(0);
glUseProgram(0);

// triangle
glPolygonMode(GL_FRONT_AND_BACK,GL_FILL);

glUseProgram(Program2);
glBindVertexArray(VAO2);

glUniformMatrix4fv(0,1,GL_FALSE,Projection.data());

glDrawArrays(GL_TRIANGLES,0,3);

glBindVertexArray(0);
glUseProgram(0);
```

Es importante observar que al activar cada VAO hace que se activen a su vez los atributos correspondientes. Otra cosa muy importante es que la matriz que representa la transformación de la cámara y la proyección, como vimos en el ejemplo anterior, hay que ”copiarla” en ambos programas. Esto es, **las variables de tipo uniform no son gestionadas por el VAO** y por tanto hay que inicializarlas cada vez que se activa un programa.

Solución 3

La siguiente solución consiste en mantener los 2 VAOs y los 4 VBO, pero con un solo programa. Eso implica que las posiciones de los atributos son las mismas. En principio pensé que no se podía pero resulta que sí, lo cual implica que por un lado existen los VBOs, pero que es la activación del VAO la que hace que se produzcan los enlaces que se van a usar con el programa que se active. Por tanto, al crear los VBOs de los ejes, les asignamos las posiciones 1 y 2, para el VAO de los ejes. Hacemos lo mismo para el tríangulo.

El vertex shader es el siguiente:

```
#version 450 core

layout (location=0) uniform mat4 matrix;
layout (location=1) in vec3 vertex;
layout (location=2) in vec3 color;
```

```
out vec3 color_out;

void main(void)
{
  color_out=color;
  gl_Position=matrix*vec4(vertex,1);
}
```

Y el programa principal queda de la siguiente manera:

```
// axis
glUseProgram(Program1);
glBindVertexArray(VAO1);

glUniformMatrix4fv(0,1,GL_FALSE,Projection.data());

glDrawArrays(GL_LINES,0,6);

glBindVertexArray(0);

// triangle
glPolygonMode(GL_FRONT_AND_BACK,GL_FILL);

glBindVertexArray(VAO2);

glUniformMatrix4fv(0,1,GL_FALSE,Projection.data());

glDrawArrays(GL_TRIANGLES,0,3);

glBindVertexArray(0);
glUseProgram(0);
```

Solución 4

Existe también la posibilidad de usar 1 VAO y 2 programas. Esto implica que los programas deben usar diferentes posiciones para los atributos, para los ejes 1 y 2, y para el triángulo 3 y 4, pero todo se activa con un solo VAO.

El código de los vertex shaders ya lo hemos visto, veamos cómo queda el código principal:

```
glBindVertexArray(VAO1);

// axis
glUseProgram(Program1);
glUniformMatrix4fv(0,1,GL_FALSE,Projection.data());
glDrawArrays(GL_LINES,0,6);
glUseProgram(0);

// triangle
glUseProgram(Program2);
glUniformMatrix4fv(0,1,GL_FALSE,Projection.data());
glDrawArrays(GL_TRIANGLES,0,3);
glUseProgram(0);
```

```
glBindVertexArray(0);
```

Solución 5

En esta solución vamos a usar 1 VAO y 1 program. Esto implica que las posiciones del shader no cambian, son 1 y 2. Pero si tenemos un sólo VAO no nos va a hacer automáticamente el cambio de los VBOs. Por tanto, hay que intercambiar los enlaces de los 4 VBO dos a dos. Esto es, cuando vamos a dibujar los ejes, hacemos que los VBOs 1 y 2 se enlacen con las posiciones 1 y 2, y cuando vamos a dibujar el triángulo hacemos que los VBOs 3 y 4 se enlacen con las posiciones 1 y 2. Lo demás no cambia. Dado que sólo tenemos un solo programa la matriz solo se inicializa una vez.

Veamos el código principal:

```
glBindVertexArray(VAO1);
glUseProgram(Program1);

glUniformMatrix4fv(0,1,GL_FALSE,Projection.data());

// axis
glVertexArrayAttribBinding(VAO1,1,1);
glVertexArrayAttribBinding(VAO1,2,2);
glDrawArrays(GL_LINES,0,6);

// triangle
glVertexArrayAttribBinding(VAO1,1,3);
glVertexArrayAttribBinding(VAO1,2,4);
glDrawArrays(GL_TRIANGLES,0,3);

glUseProgram(0);
glBindVertexArray(0);
```

Solución 6

Tras las distintas pruebas podemos llegar a ciertas conclusiones que nos pueden ayudar a crear programas más eficientes. La dos ideas principales son

- **Hay que intentar que el máximo de datos se encuentren en la GPU**
- **Hay que intentar evitar los cambios**

Estas ideas van en la dirección del *Zero Driver Overhead*.

En la solución que proponemos ahora, sólo se definen 1 VAO y 1 program, y **sólo 2 VBO**. La idea es eliminar el cambio de programa, evitando la copia de variables de tipo uniform, y evitar los cambios de VAO, que pueden ser costosos. Esto implica que en el VBO

de posiciones vamos a meter las posiciones de los ejes y del triángulo, y lo mismo se va a hace con el VBO de los colores.

La idea es my sencilla: primero ponemos los vértices de los ejes y a continuación los vértices del triángulo. Lo mismo para los colores.

Veamos de nuevo el código de la inicialización de los VBOs:

```
glCreateVertexArrays(1,&VAO1);
glBindVertexArray(VAO1);

glCreateBuffers(1,&VBO_vertices1);
glNamedBufferStorage(VBO_vertices1,(Axis_vertices.size()+Triangle_vertices.size())*3*↩
    sizeof(float),nullptr,GL_DYNAMIC_STORAGE_BIT | GL_MAP_WRITE_BIT);
glVertexArrayVertexBuffer(VAO1,1,VBO_vertices1,0,3*sizeof(float)); // 0,VBO
glVertexArrayAttribFormat(VAO1,1,3,GL_FLOAT,GL_FALSE,0);
glEnableVertexArrayAttrib(VAO1,1);

glCreateBuffers(1,&VBO_colors1);
glNamedBufferStorage(VBO_colors1,(Axis_colors.size()+Triangle_colors.size())*3*sizeof(↩
    float),nullptr,GL_DYNAMIC_STORAGE_BIT | GL_MAP_WRITE_BIT);
glVertexArrayVertexBuffer(VAO1,2,VBO_colors1,0,3*sizeof(float)); // 1,VBO
glVertexArrayAttribFormat(VAO1,2,3,GL_FLOAT,GL_FALSE,0);
glEnableVertexArrayAttrib(VAO1,2);

// Put data
// vertices
glNamedBufferSubData(VBO_vertices1,0,Axis_vertices.size()*3*sizeof(GLfloat),&↩
    Axis_vertices[0]);
glNamedBufferSubData(VBO_vertices1,Axis_vertices.size()*3*sizeof(GLfloat),↩
    Triangle_vertices.size()*3*sizeof(GLfloat),&Triangle_vertices[0]);
// colors
glNamedBufferSubData(VBO_colors1,0,Axis_colors.size()*3*sizeof(GLfloat),&Axis_colors[0]);
glNamedBufferSubData(VBO_colors1,Axis_colors.size()*3*sizeof(GLfloat),Triangle_colors.↩
    size()*3*sizeof(GLfloat),&Triangle_colors[0]);

glBindVertexArray(0);
```

El código mostrado es casi igual al que vimos en el tema anterior, pero hay un par de cambios importantes. El primero es que en este caso con la instrucción glNamedBufferStorage sólo estamos creando el buffer de un tamaño determinado (las suma de vertices de los ejes y el triángulo), ya que hemos puesto el puntero a los datos a nullptr. Otro detalle es que hemos incluido la directiva GL_DYNAMIC_STORAGE_BIT para que nos permita acceder al buffer en otro momento, que es lo que hacemos después con la función glNamedBufferSubData. La misma nos sirve para escribir datos en el buffer. Como se puede ver, los parámetros son el identificador del VBO, la posición inicial desde la que va a empezar a guardarse los datos, el tamaño de los datos que se quieren copiar, y finalmente, la dirección donde comienzan los datos que se van a copiar al VBO.

En el ejemplo se puede ver como los parámetros de inicio y tamaño se usan para meter los datos de los ejes, desde la posicion 0 usando $6 \times 3 \times sizeof(float)$ bytes, y del triángulo, comenzando donde terminan los datos de los ejes, y ocupando $3 \times 3 \times sizeof(float)$ bytes.

Como se ve, hemos secuenciado la información de los dos objetos que se desean dibujar.

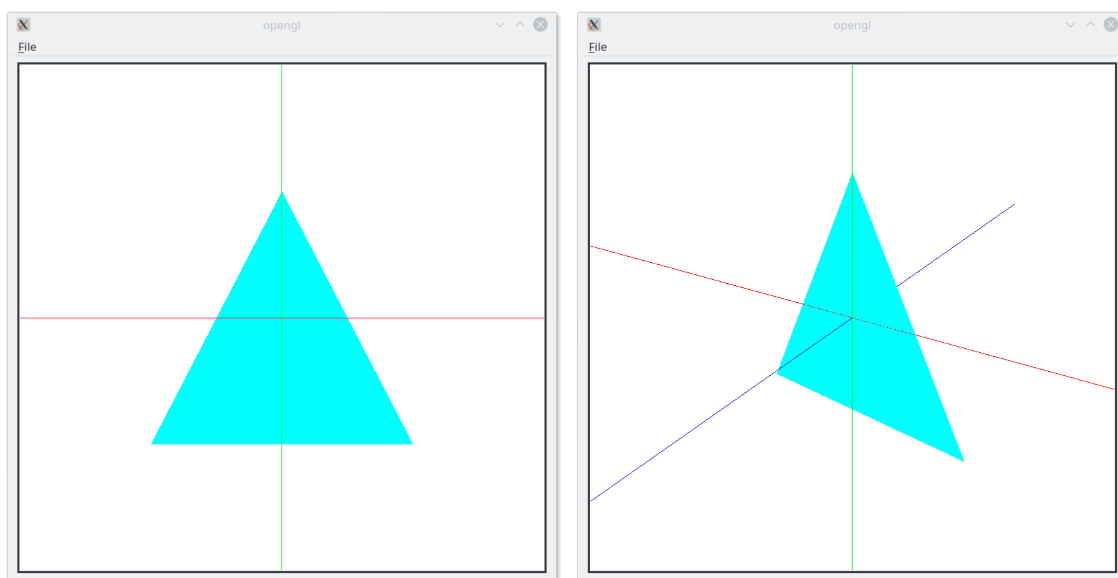


Figura 7.1: Resultado del ejemplo 7

Para dibujar ambos objetos usamos el siguiente código:

```
glBindVertexArray(VAO1);
glUseProgram(Program1);
glUniformMatrix4fv(0,1,GL_FALSE,Projection.data());

// axis
glDrawArrays(GL_LINES,0,6);
// triangle
glDrawArrays(GL_TRIANGLES,6,3);

glUseProgram(0);
glBindVertexArray(0);
```

El detalle está en que la instrucción glDrawArrays permite indicar desde donde empieza a coger los datos: para los ejes empezamos en 0 y uso 6 posiciones que interpreta como líneas (por tanto, desde el vértice 0 al 5), y para el triángulo empiezo en la posición 6 y uso 3 vértices que son interpretados como un triángulo (por tanto, se usan desde el vértice 6 al 8).

Como se puede ver, de esta manera se optimiza el número de llamadas y no hay cambios de contexto.

8

Interpolando colores

Con el anterior ejemplo hemos conseguido hacer lo que en la mayoría de los tutoriales sobre OpenGL es el primer ejemplo: dibujar un triángulo. ¡Pero nosotros hemos añadido unos ejes y movimiento en 3D! Así es más fácil entender el espacio y la localización de los objetos. Antes de continuar hay que entender una de las características principales del paso de variables entre el vertex shader y el fragment shader: la interpolación.

Hasta este momento, lo que hemos hecho en el vertex shader es producir la posición que le corresponde al vértice de entrada, la cual se guarda en gl_Position, una variable interna de OpenGL que se usa en otras etapas, incluido el fragment shader, y hemos pasado el color, tal y como nos llegaba.

Esto parece inútil: ¿por que no ternerlo directamente en el fragment shader sin tener que pasar por el vertex shader? La respuesta está en que normalmente la forma de obtener el color no va a ser una asignación directa como la que estamos haciendo sino que se va a producir una interpolación, si la primitiva tiene 2 o 3 vértices (líneas y triángulos). Esto es, imaginemos una línea recta compuesta por 2 vértices, uno de color blanco y otro negro. Esta línea se convertirá en un conjunto de pixeles, con un extremo blanco y el otro negro. ¿Cual debe ser el color de los píxeles intermedios? Pues una interpolación entre el blanco y el negro que dependerá de la posición del pixel. Por tanto, en el cauce gráfico, además de la tarea de convertir la primitiva gráfica en fragmentos, debe calcular el color para cada uno de estos fragmentos usando una interpolación líneal. En el caso de los triángulos también se lleva a cabo una interpolación, por línea de barrido, que asigna un color a cada uno de los píxeles.

Debemos entender que a cada variable que pasamos del vertex shader al fragment shader se le aplica una interpolación (es posible modificar este comportamiento). En los ejemplos que hemos implementado la interpolación se ha hecho entre extremos iguales, con lo cual no se ha producido ningún cambio. En este ejercicio vamos a hacer que se interpolen los colores

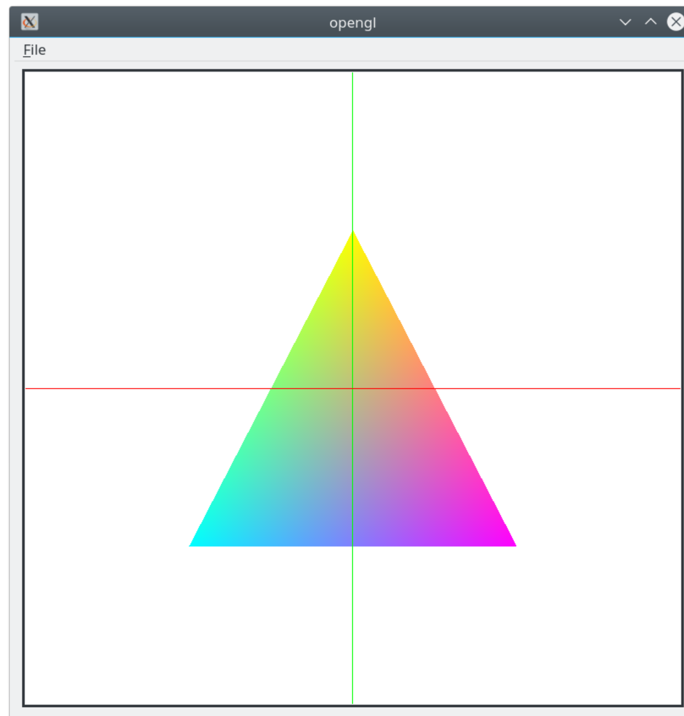


Figura 8.1: Resultado del ejemplo 8

de los vértices del tríangulo. Para ello solo tenemos que cambiar el color. Y ya está, OpenGL se encarga de realizar la interpolación.

```
Triangle_vertices.resize(3);
Triangle_vertices[0]=_vertex3f(-TRIANGLE_SIZE,-TRIANGLE_SIZE,0); // x
Triangle_vertices[1]=_vertex3f(TRIANGLE_SIZE,-TRIANGLE_SIZE,0);
Triangle_vertices[2]=_vertex3f(0,TRIANGLE_SIZE,0); // y

Triangle_colors.resize(3);
Triangle_colors[0]=_vertex3f(0,1,1); // cian
Triangle_colors[1]=_vertex3f(1,0,1); // magenta
Triangle_colors[2]=_vertex3f(1,1,0); // yellow
```

Como veremos, la interpolación es necesaria para los cálculos de iluminación y para la implementación de otros procedimientos que necesitan valores intermedios dados sólo los extremos.

9

Un cubo y diversos modos de dibujado

Una vez hemos visto cómo se dibuja un triángulo vamos a visualizar un objeto más complejo compuesto de triángulos. Es importante tener en cuenta que, a efectos de dibujado, no hay diferencia entre los distintos objetos, lo único que va a cambiar es el número de triángulos, siendo la posición y orientación de los mismos la que nos parezca como un coche o un dragón. En nuestro caso, vamos a usar un cubo. Vamos a usar el mismo procedimiento de secuenciar los datos de los distintos objetos, en nuestro caso los ejes y el cubo.

Otro de los problemas que tenemos que resolver es que queremos dibujar los objetos de distintas maneras, mientras que los ejes siempre se mantienen iguales. Los modos de dibujado que queremos implementar son puntos, líneas y relleno. En el modo puntos sólo se dibujaran los puntos de las primitivas, en el modo líneas solo las líneas y en el modo relleno solo el interior en un color sólido o constante. Vamos a usar una variable que indique el modo de dibujado que deseamos obtener, y otra variable que controle cómo se genera el color. Por tanto, lo que se va a hacer es crear el cubo y meter los datos de posición y color junto a los de los ejes en los correspondientes VBOs. Para facilitar la codificación vamos a hacer uso de una capacidad de C++ (y otros lenguajes orientados a objetos): la derivación. Hay que indicar que las aplicaciones gráficas se adaptan especialmente bien a este paradigma.

Para ello vamos a crear una clase genérica que defina los datos que se necesitan almacenar. Para nuestro ejemplo, vamos a definir los vectores de vértices y de colores para cada vértice, de triángulos, y por supuesto, como vamos a usar la instrucción glDrawArrays, los vectores correspondientes para los vértices y los colores para que puedan ser usados directamente al rellenar los VBOs. Es verdad que se podría prescindir de los mismos, pero hemos optado porque sea más visible el proceso y permitir alguna mejora en la velocidad a cambio de incrementar el espacio ocupado.

La clase sería la siguiente:

Domingo Martín Perandrés©

```
class _basic_object3d{
public:
  vector<_vertex3f> Vertices;
  vector<_vertex3f> Vertices_colors;

  vector<_vertex3i> Triangles;

  vector<_vertex3f> Vertices_drawarray;
  vector<_vertex3f> Vertices_colors_drawarray;
};
```

Hay que observar que las versiones de los vectores que acaban en drawarray, se componen de los datos de los otros vectores pero con la información dispuesta de tal manera que se adapte a la condición de que para cada objeto que se va a dibujar deben estar todos los vértices y para cada vértice todos sus atributos adicionales. Por ejemplo, para el caso de nuestro cubo, los vamos a crear con 8 vértices con sus correspondientes colores, y 12 triángulos. Estos datos se van a almacenar en los vectores Vertices, Vertices_colors y Triangles, respectivamente. En Vertices_drawarray tendremos que almacena 12×3 vértices. Lo mismo para los colores.

Las clases para los ejes y el cubo lo que hacen es inicializar los vectores. Veamos el caso de los ejes. Primero la definición:

```
namespace _axis_ne {
const float MAX_AXIS_SIZE=1000;
}

class _axis: public _basic_object3d
{
public:

  _axis();
};
```

Sólo tenemos el constructor que es el que se encarga de inicializar los datos:

```
using namespace _axis_ne;

_axis::_axis()
{
  // vertices
  Vertices.resize(6);
  Vertices[0]=_vertex3f(-MAX_AXIS_SIZE,0,0);
  Vertices[1]=_vertex3f(MAX_AXIS_SIZE,0,0);
  Vertices[2]=_vertex3f(0,-MAX_AXIS_SIZE,0);
  Vertices[3]=_vertex3f(0,MAX_AXIS_SIZE,0);
  Vertices[4]=_vertex3f(0,0,-MAX_AXIS_SIZE);
  Vertices[5]=_vertex3f(0,0,MAX_AXIS_SIZE);

  // vertices colors
  Vertices_colors.resize(6);
  Vertices_colors[0]=RED;
  Vertices_colors[1]=RED;
  Vertices_colors[2]=GREEN;
  Vertices_colors[3]=GREEN;
  Vertices_colors[4]=BLUE;
  Vertices_colors[5]=BLUE;
```

```
  // create drawarrays
  // vertices and color
  Vertices_drawarray.resize(Vertices.size());
  Vertices_colors_drawarray.resize(Vertices_colors.size());

  for (unsigned int i=0;i<Vertices.size();i++){
    Vertices_drawarray[i]=Vertices[i];
    Vertices_colors_drawarray[i]=Vertices_colors[i];
  }
}
```

El código para el cubo es similar, aunque se puede observar que la forma de rellenar los vectores drawarray difiere un poco:

```
_cube::_cube()
{
  // vertices
  Vertices.resize(8);
  Vertices[0]=_vertex3f(-0.5,-0.5,0.5);
  Vertices[1]=_vertex3f(0.5,-0.5,0.5);
  Vertices[2]=_vertex3f(-0.5,0.5,0.5);
  Vertices[3]=_vertex3f(0.5,0.5,0.5);
  Vertices[4]=_vertex3f(-0.5,-0.5,-0.5);
  Vertices[5]=_vertex3f(0.5,-0.5,-0.5);
  Vertices[6]=_vertex3f(-0.5,0.5,-0.5);
  Vertices[7]=_vertex3f(0.5,0.5,-0.5);

  // triangles
  Triangles.resize(12);
  Triangles[0]=_vertex3i(2,0,1);
  Triangles[1]=_vertex3i(1,3,2);
  Triangles[2]=_vertex3i(3,1,5);
  Triangles[3]=_vertex3i(5,7,3);
  Triangles[4]=_vertex3i(7,5,4);
  Triangles[5]=_vertex3i(4,6,7);
  Triangles[6]=_vertex3i(6,4,0);
  Triangles[7]=_vertex3i(0,2,6);
  Triangles[8]=_vertex3i(0,4,5);
  Triangles[9]=_vertex3i(5,1,0);
  Triangles[10]=_vertex3i(6,2,3);
  Triangles[11]=_vertex3i(3,7,6);

  // vertices colors
  Vertices_colors.resize(8);
  Vertices_colors[0]=_vertex3f(1,0,0);
  Vertices_colors[1]=_vertex3f(0,1,0);
  Vertices_colors[2]=_vertex3f(0,0,1);
  Vertices_colors[3]=_vertex3f(0,1,1);
  Vertices_colors[4]=_vertex3f(1,0,1);
  Vertices_colors[5]=_vertex3f(1,1,0);
  Vertices_colors[6]=_vertex3f(0.5,0.5,0.5);
  Vertices_colors[7]=_vertex3f(0.4,0.5,0.6);

  // create drawarrays
  // vertices
  Vertices_drawarray.resize(Triangles.size()*3);
  for (unsigned int i=0;i<Triangles.size();i++){
    Vertices_drawarray[i*3]=Vertices[Triangles[i]._0];
    Vertices_drawarray[i*3+1]=Vertices[Triangles[i]._1];
    Vertices_drawarray[i*3+2]=Vertices[Triangles[i]._2];
  }

  // vertices colors
```

```
  Vertices_colors_drawarray.resize(Triangles.size()*3);
  for (unsigned int i=0;i<Triangles.size();i++){
    Vertices_colors_drawarray[i*3]=Vertices_colors[Triangles[i]._0];
    Vertices_colors_drawarray[i*3+1]=Vertices_colors[Triangles[i]._1];
    Vertices_colors_drawarray[i*3+2]=Vertices_colors[Triangles[i]._2];
  }
}
```

Para poder dibujar los ejes y el cubo debemos meter los datos en los VBOs. En la clase que se encarga del dibujado, _gl_widget, añadimos dos instancias de las clases que necesitamos. De esta manera al crear el objeto principal, creará los datos de los objetos que se van a dibujar. En la función initializeGL, que recordamos que se llama automáticamente al crear la instancia de la clase _gl_widget, no sólo inicializaremos OpenGL sino que llamaremos a la función initialize_objects que se va a encargar de crear las variables y VBOs necesarios para visualizar los objetos. El código es el siguiente:

```
void _gl_widget::initialize_objects()
{
  _shaders Shader;

  Program1=Shader.load_shaders("shaders/cubo.vert","shaders/cubo.frag");
  if (Program1==0){
    exit(-1);
  }

  glCreateVertexArrays(1,&VAO1);
  glBindVertexArray(VAO1);

  // vertices
  int Num_vertices_total=Axis.Vertices_drawarray.size()+Cube.Vertices_drawarray.size();
  glCreateBuffers(1,&VBO_vertices1);
  glNamedBufferStorage(VBO_vertices1,Num_vertices_total*3*sizeof(GLfloat),nullptr,↩
      GL_DYNAMIC_STORAGE_BIT | GL_MAP_WRITE_BIT);
  glVertexArrayVertexBuffer(VAO1,0,VBO_vertices1,0,3*sizeof(GLfloat));
  glVertexArrayAttribFormat(VAO1,0,3,GL_FLOAT,GL_FALSE,0);
  glEnableVertexArrayAttrib(VAO1,0);

  // colors
  glCreateBuffers(1,&VBO_colors1);
  glNamedBufferStorage(VBO_colors1,Num_vertices_total*3*sizeof(GLfloat),nullptr,↩
      GL_DYNAMIC_STORAGE_BIT | GL_MAP_WRITE_BIT);
  glVertexArrayVertexBuffer(VAO1,1,VBO_colors1,0,3*sizeof(GLfloat));
  glVertexArrayAttribFormat(VAO1,1,3,GL_FLOAT,GL_FALSE,0);
  glEnableVertexArrayAttrib(VAO1,1);

  // Put data
  // vertices axis
  glNamedBufferSubData(VBO_vertices1,0,Axis.Vertices_drawarray.size()*3*sizeof(GLfloat),&↩
      Axis.Vertices_drawarray[0]);
  // vertices cube
  glNamedBufferSubData(VBO_vertices1,Axis.Vertices_drawarray.size()*3*sizeof(GLfloat),↩
      Cube.Vertices_drawarray.size()*3*sizeof(GLfloat),&Cube.Vertices_drawarray[0]);

  // colors
  // color axis
  glNamedBufferSubData(VBO_colors1,0,Axis.Vertices_drawarray.size()*3*sizeof(GLfloat),&↩
      Axis.Vertices_colors_drawarray[0]);
  // colors cube
  glNamedBufferSubData(VBO_colors1,Axis.Vertices_drawarray.size()*3*sizeof(GLfloat),Cube.↩
      Vertices_drawarray.size()*3*sizeof(GLfloat),&Cube.Vertices_colors_drawarray[0]);

  glBindVertexArray(0);
```

```
}
```

Para poder dibujar los objetos, modificamos el código de la función draw_objects. No sólo necesitamos el código para dibujar los ejes sino que además queremos poder dibujar el cubo en los modos que hemos indicado previamente. Además también tenemos que incluir las matrices con las transformaciones para simular la cámara, permitiendo el cambio de un proyección de perspectiva a una paralela y viceversa. Como ya sabemos, la visualización se lleva a cabo por los shaders. Por tanto, no sólo debemos pasar la información de donde encontrar los vértices y los colores, sino que debemos incluir las matrices y añadimos las variables que van a controlar como se visualiza. En concreto, creamos una variable que se encarga del modo de visualización y otra de la forma de usar los colores. El código completo de la función es el siguiente:

```
void _gl_widget::draw_objects()
{
  QMatrix4x4 Projection;
  QMatrix4x4 View;
  QMatrix4x4 Model;

  float Aspect=(float)Window_height/(float)Window_width;

  if (Projection_type==PERSPECTIVE_PROJECTION){
    Projection.frustum(X_MIN,X_MAX,Y_MIN*Aspect,Y_MAX*Aspect,FRONT_PLANE_PERSPECTIVE,↩
        BACK_PLANE_PERSPECTIVE);
  }
  else{
    Projection.ortho(X_MIN*Scale_factor,X_MAX*Scale_factor,Y_MIN*Aspect*Scale_factor,↩
        Y_MAX*Aspect*Scale_factor,FRONT_PLANE_PARALLEL,BACK_PLANE_PARALLEL);
  }

  View.translate(0,0,-Distance);
  View.rotate(Angle_camera_x,1,0,0);
  View.rotate(Angle_camera_y,0,1,0);

  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
  glBindVertexArray(VAO1);
  glUseProgram(Program1);

  glUniformMatrix4fv(10,1,GL_FALSE,Model.data());
  glUniformMatrix4fv(11,1,GL_FALSE,View.data());
  glUniformMatrix4fv(12,1,GL_FALSE,Projection.data());

  // axis
  glUniform1i(20,MODE_FILL);
  glUniform1i(21,(int)MODE_COLORS_VARIABLE);
  glDrawArrays(GL_LINES,0,6);

  if (Draw_points){
    glPolygonMode(GL_FRONT_AND_BACK,GL_POINT);
    glPointSize(3);
    glUniform1i(20,MODE_POINT);
    glUniform1i(21,(int)MODE_COLORS_FIXED);/
    glUniform3fv(25,1,(GLfloat*) &COLORS[COLOR_POINT]);
    glDrawArrays(GL_TRIANGLES,6,36);
  }

  if (Draw_lines){
    glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
    glPolygonOffset(-1,1);
    glUniform1i(20,MODE_LINE);
```

```
    glUniform1i(21,(int)MODE_COLORS_FIXED);
    glUniform3fv(26,1,(GLfloat*) &COLORS[COLOR_LINE]);
    glDrawArrays(GL_TRIANGLES,6,36);
  }

  if (Draw_fill){
    glPolygonMode(GL_FRONT_AND_BACK,GL_FILL);
    glUniform1i(20,MODE_FILL);
    glUniform1i(21,Mode);
    glUniform3fv(27,1,(GLfloat*) &COLORS[COLOR_FILL]);
    glDrawArrays(GL_TRIANGLES,6,36);
  }

  glUseProgram(0);
  glBindVertexArray(0);
}
```

En este código hay que señalar los siguientes elementos. Por un lado, tenemos tres matrices de transformación: Model, View y Projection. La funcionalidad de las dos últimas ya la hemos visto y es la que permite implementar la cámara. La matriz Model se encarga de aplicar las transformaciones al modelo. Ya que sólo la inicializamos, su valor será la identidad, esto es, no se aplica ninguna transformación geométrica a los modelos (las transformaciones las veremos en el siguiente tema).

Para poder dibujar en los tres modos que hemos indicado, punto, línea y relleno, cuando se trata del cubo, es necesario crear el código para cada tipo. Obsérvese que en los tres casos se dibuja lo mismo, triángulos, pero lo importante está en que indicamos a OpenGL como debe realizar el dibujado de los triángulos mediante la llamada glPolygonMode(MODO_CARA, MODO_VISUALIZACION). La variable importante es MODO_VISUALIZACION pues indica cómo queremos dibujar. Las opciones son GL_POINT, GL_LINE y GL_FILL para dibujar sólo los puntos, las líneas o el relleno interior de los triángulos, respectivamente. La variable MODO_CARA indica a que lados de la cara debe afectar.

Cuando dibujamos el cubo en modo puntos o líneas, se fuerza a que el color sea fijo. Cuando se dibuja en modo relleno, dependiendo de la variable Modo se dibujará en el color sólido o realizará la interpolación entre colores. Dado que tenemos distintas variables para la visualización, es posible combinar los distintos efectos.

Los shaders son muy sencillos. El vertex shader se encarga de seleccionar el color apropiado para cada vértice dependiendo de las variables de control. Además, como ya hemos visto anteriormente, se encarga de calcular la posición aplicando las transformaciones. Obsérvese el orden de aplicación de las mismas:

```
#version 450 core

layout (location=0) in vec3 vertex;
layout (location=1) in vec3 color;

layout (location=10) uniform mat4 model_matrix;
layout (location=11) uniform mat4 view_matrix;
layout (location=12) uniform mat4 projection_matrix;

layout (location=20) uniform int mode_rendering;
```

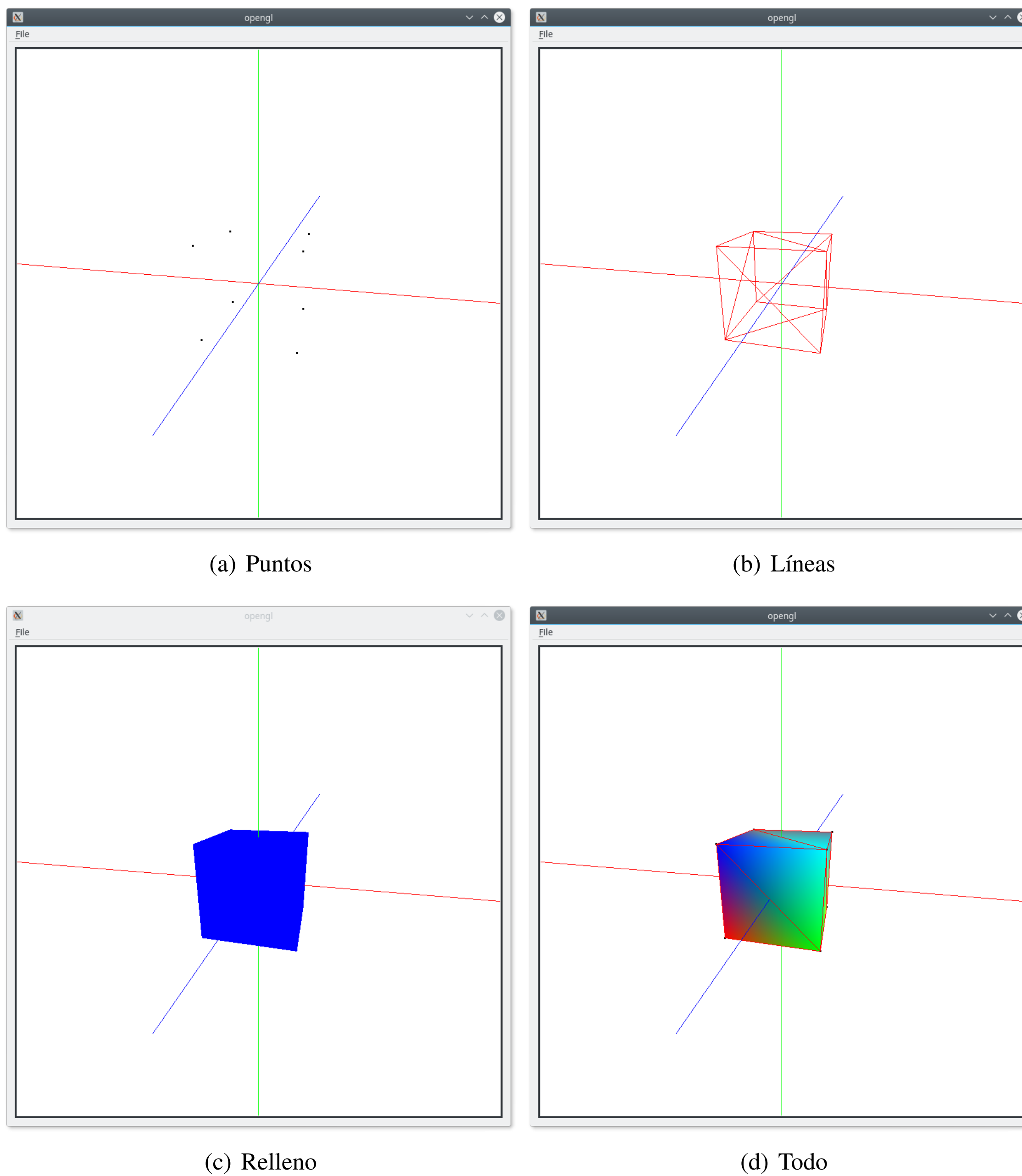
```
layout (location=21) uniform int mode_color;

layout (location=25) uniform vec3 color_point;
layout (location=26) uniform vec3 color_line;
layout (location=27) uniform vec3 color_fill;

out vec3 color_out;
out vec3 vertex_out;

void main(void)
{
  if (mode_color==0){ // fixed
    switch (mode_rendering){
      case 0: color_out=color_point;break;
      case 1: color_out=color_line;break;
      case 2: color_out=color_fill;break;
    }
  }
  else{
    color_out=color;
  }
  vertex_out=vec3(model_matrix*vec4(vertex,1));
  gl_Position=projection_matrix*view_matrix*model_matrix*vec4(vertex,1);
}
```

El fragment shader es trivial:

```
#version 450 core

in vec3 color_out;
in vec3 vertex_out;

out vec4 frag_color;

void main(void)
{
  frag_color=vec4(color_out,1);
}
```

El código que permite cambiar la activación de cada modo es muy sencillo:

```
case Qt::Key_P:Draw_points=!Draw_points;break;
case Qt::Key_L:Draw_lines=!Draw_lines;break;
case Qt::Key_F:Draw_fill=!Draw_fill;break;
```



(a) Puntos
(b) Líneas
(c) Relleno
(d) Todo

Figura 9.1: Resultado del ejemplo 9

10

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 10.1 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

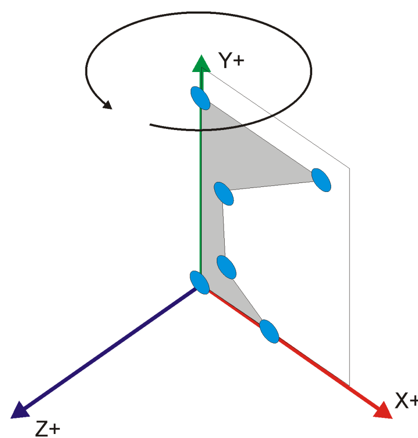


Figura 10.1: Barrido por revolución.

Domingo Martín Perandrés©

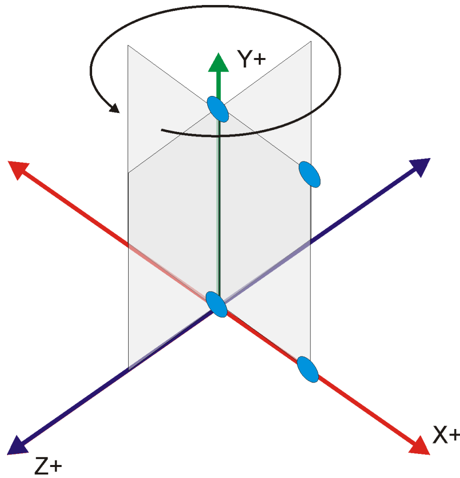


Figura 10.2: Perfil para obtener un cilindro

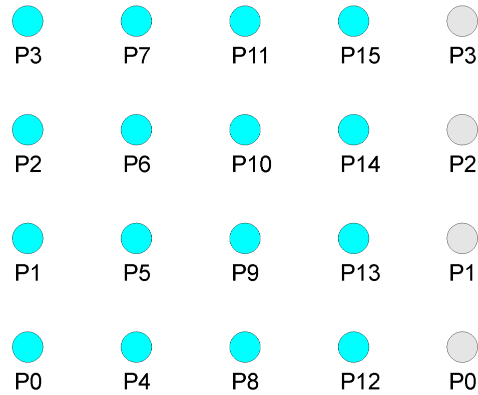


Figura 10.3: Despliegue de los puntos en un plano

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 10.2. 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

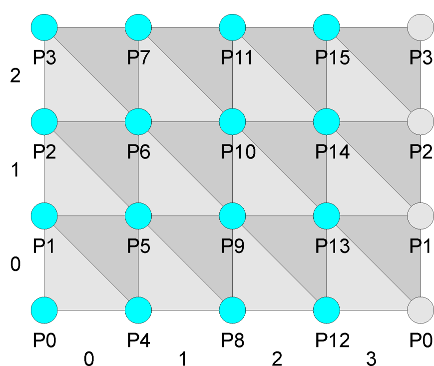


Figura 10.4: Posicionado lineal de los puntos

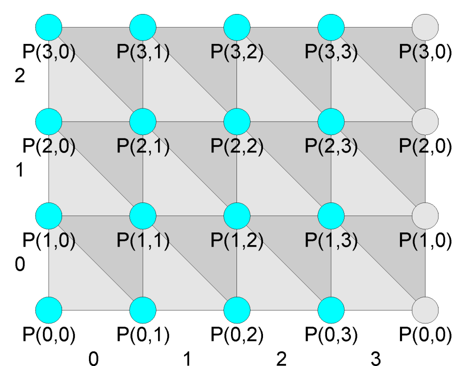


Figura 10.5: Posicionado 2D de los puntos

unos 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 10.3. 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 10.4. 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-

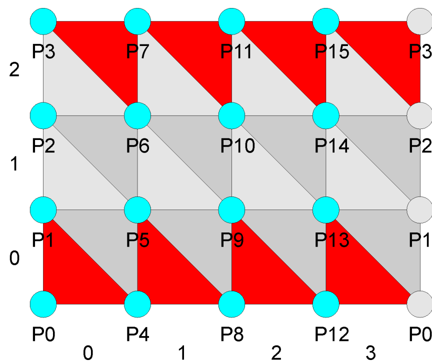


Figura 10.6: Triángulos degenerados

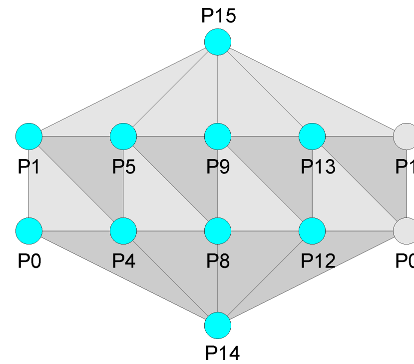


Figura 10.7: Versión optimizada

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

Domingo Martín Perandrés©

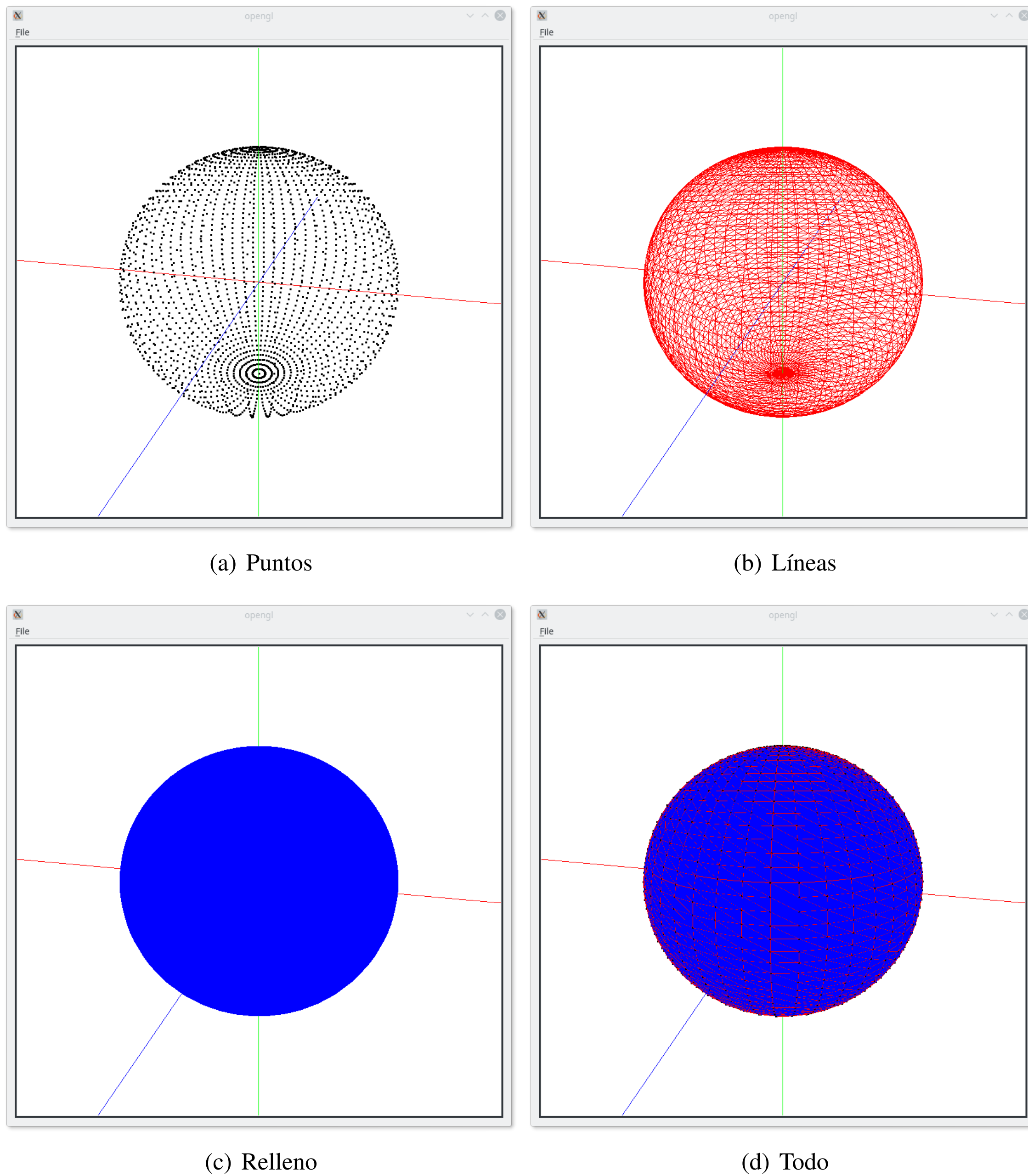
Sólo nos quedaría el hacer que se aplicara el operador módulo para obtener el algoritmo completo. En vez de mostrar dicho código, vamos a plantear otra forma de tratar el direccionamiento de los puntos dentro de la matriz que hará que nuestro código sea más fácil de leer. Para ello sólo tenemos que direccionar los puntos en forma matricial en vez de lineal, tal y como se muestra en la figura 10.5. 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 10.6 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 10.8: Resultado del ejemplo 10 con los diferentes modo de visualización

11

Transformaciones geométricas

Ya hemos visto cómo crear objetos 3D sencillos. Ahora vamos a ver cómo se pueden crear objetos más complejos mediante el uso de objetos más sencillos y la aplicación sobre los mismos de transformaciones geométricas, lo que se denomina modelado jerárquico. Por tanto, lo primero que tendremos que comprender son las transformaciones geométricas, el procedimiento matemático para hacer que el modelo pueda cambiar de tamaño, orientación y posición, entre otras posibilidades. Para poder observar el efecto de las transformaciones vamos a usar el cubo.

Dado el cubo, que se ha creado de tal forma que tiene un tamaño de arista unidad y que se encuentra centrado con respecto al origen, lo que nos estamos planteando es cómo hacer para que el cubo lo podamos colocar en otra posición, le podamos cambiar el tamaño (y al hacerlo también incluso la forma), y podamos orientarlo como deseemos. Veamos esto tres tipos de transformaciones con más detalle.

La transformación que permite mover un objeto de una posición a otra se llama **traslación** (ver figura 11.1). Dado que queremos que nuestro objeto, el cubo, se comporte como si fuera rígido, el movimiento que se aplica a un vértice hay que aplicarlo a todos. Por tanto, sólo tenemos que ver cómo se puede hacer para trasladar un punto de una posición a otra y aplicar la misma operación a todos los puntos del modelo.

Si tenemos el punto P_0 y el valor de las traslaciones $T = (T_x, T_y, T_z)$, la posición trasladada se obtiene sumando el desplazamiento a las coordenadas del punto: $P'_0 = P_0 + T$:

$$x' = x + T_x$$
$$y' = y + T_y$$
$$z' = z + T_z$$

Domingo Martín Perandrés©

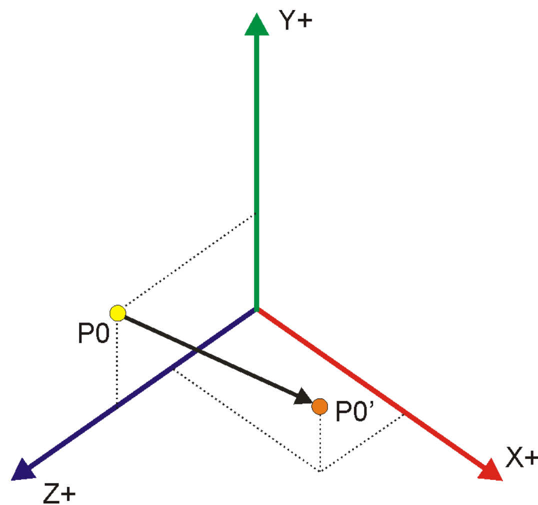


Figura 11.1: Traslación

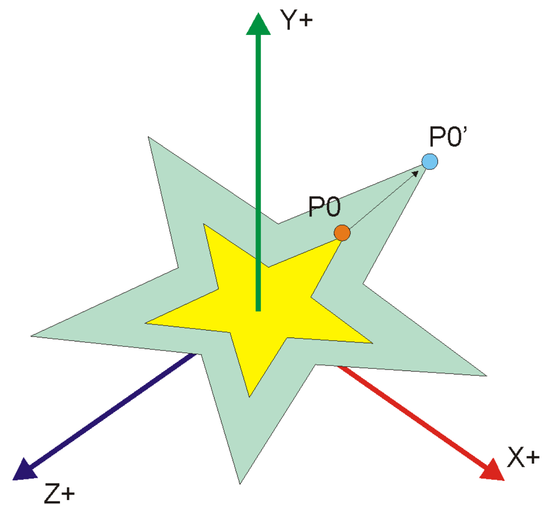


Figura 11.2: Escalado

La transformación que permite cambiar el tamaño se llama **escalado** (ver figura 11.2). La operación que escala es la multiplicación. Si tenemos el punto P_0 y tenemos los factores de escalado $E = (E_x, E_y, E_z)$, la posición escalada se obtiene al multiplicar los factores de escala por las coordenadas del punto: $P_0' = P_0 \cdot E$:

$$x' = x \cdot E_x$$
$$y' = y \cdot E_y$$
$$z' = z \cdot E_z$$

Es importante hacer notar lo siguiente:

- Si los factores de escala son mayores que 1 el objeto se agranda.
- Si los factores de escala son menores que 1 y mayores que 0, el objeto se empequeñece.
- Si los factores de escala son 0, el objeto colapsa al punto $(0,0,0)$.
- Si los factores son negativos se produce un reflejo, pudiendo invertir el objeto.
- Si los factores son todos iguales se produce un escalado homogéneo.
- Si los factores son diferentes se produce un escalado heterogéneo.
- El escalado se realiza en relación al origen de coordenadas. Esto es, al escalar el punto, el resultado estará en la línea que une el punto con el origen. Si la posición con respecto a la que se desea realizar el escalado no es el origen habrá que previamente convertir dicha posición en el origen. Veremos el proceso más adelante.

Domingo Martín Perandrés©

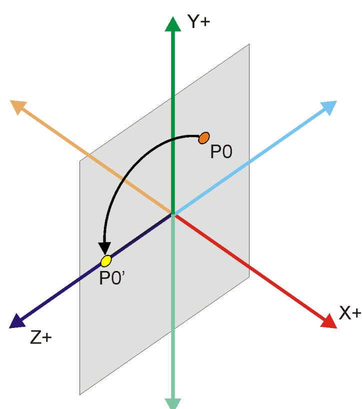


Figura 11.3: Rotación en el eje x

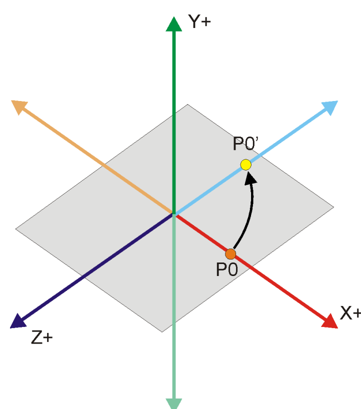


Figura 11.4: Rotación en el eje y

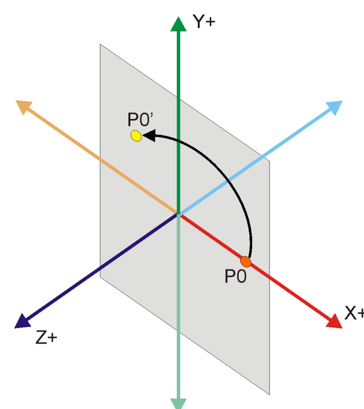


Figura 11.5: Rotación en el eje z

La transformación que permite cambiar la orientación se llama **rotación**. Dado que estamos en 3 dimensiones, tenemos la posibilidad de girar el objeto con respecto a los tres ejes principales, teniendo una formulación diferente para la rotación con respecto al eje x (ver figura 11.3), el eje y (ver figura 11.4), y al eje z (ver figura 11.5). En este caso, el parámetro que define una rotación es el ángulo que deseamos rotar. Las fórmulas para realizar las rotaciones son las siguientes:

$$
\begin{array}{lll}
x' = x & x' = x \cdot \cos(\alpha) + z \cdot \sin(\alpha) & x' = x \cdot \cos(\alpha) - y \cdot \sin(\alpha) \\
y' = y \cdot \cos(\alpha) - z \cdot \sin(\alpha) & y' = y & y' = x \cdot \sin(\alpha) + y \cdot \cos(\alpha) \\
z' = y \cdot \sin(\alpha) + z \cdot \cos(\alpha) & z' = -x \cdot \sin(\alpha) + z \cdot \cos(\alpha) & z' = z
\end{array}
$$

Es importante hacer notar que la coordenada que es igual al eje sobre el que se está rotando no cambia. También, que como el escalado, las rotaciones se realizan teniendo como pivote al origen. En caso de que se desee rotar con respecto a otro pivote primero habrá que convertirlo en el origen.

Por tanto, ya tenemos las fórmulas que nos permiten hacer las distintas transformaciones. Ahora vamos a ver que, en general, los objetos sufren varías transformaciones para conseguir el resultado final.Por ejemplo, es muy normal tener que cambiar de tamaño o escala, rotarlo y trasladarlo (transformación de instanciación).

Otros casos son los escalados y rotaciones con respecto a puntos que no son el origen: el procedimiento consiste en convertir el punto con respecto al que se escala o rota en el origen con una traslación, a continuación se aplica el escalado o rotación, y por último, se desharía la traslación (ver figura 11.6).

De estos ejemplos que hemos comentado, lo importante es ver que tenemos que producir una secuencia de transformaciones para obtener el resultado. Por ejemplo, en la transformación de instanciación tendríamos que aplicar los siguientes pasos en el siguiente orden:

1. $P' = E(P)$
2. $P'' = R(P')$
3. $P''' = T(P'')$

El problema reside en que cada vez que aplicamos una transformación, la misma debe ser aplicada a todos los puntos del modelo: si nuestro modelo consta de 1 millón de vértices, y hay n transformaciones, entonces tendremos que aplicar n millones de transformaciones.

La solución pasa por intentar construir una transformación que sea capaz de realizar lo mismo pero con una sola aplicación. Esto es, estamos buscando una trasformación U tal que $P''' = U(P)$. ¿Es esto posible? Sí, sólo tenemos que ver las fórmulas con las que hemos definido las transformaciones y hacer una sustitución. Vamos a plantear un caso más sencillo para verlo: un escalado seguido de una traslación. Para el escalado tenemos $P' = E(P)$ y para la traslación tenemos $P'' = T(P')$. Vamos a usar las formulas: $P' = P \cdot E$ y $P'' = P' + T$. Por tanto, sustituyendo P' en la segunda ecuación nos quedará $P'' = P \cdot E + T$. Si en vez de hacer primero el escalado y luego la traslación invirtiéramos el orden nos quedaría $P'' = (P + T) \cdot E$.

Ahora pensemos en que tenemos que crear el código que debe permitir la combinación de cualquier tipo y número de transformaciones en cualquier orden y nos daremos cuenta de que tenemos un gran problema. Tenemos que buscar una solución que permita realizar dicha tarea de una manera sencilla y directa. La solución pasa por reescribir las fórmulas de las transformaciones de tal manera que facilite el proceso. Dicha reescritura es la matricial: vamos a buscar matrices que realicen la misma transformación que la definida por la ecuación explícita.

El primer paso consiste en definir cómo se transforma un punto al utilizar matrices. La idea es que la matriz define la transformación y que el punto sufre dicha transformación al operar con la matriz. Si hacemos que un punto represente a un vector unidimensional, podemos definir la transformación como el producto del punto por la matriz. Esto es, $P' = P \cdot M$. En mátemáticas y en OpenGL, se usa la versión de definición por columnas (*column major*) en vez de la definición por filas (*row major*) que normalmente se usa en los lenguajes de programación como C, C++, etc. Por tanto, si usamos la definición column major, nos queda $P'^T = M \cdot P^T$. Esto es, trabajamos con vectores columna en vez de fila.

Dado que vamos a realizar las transformaciones en el espacio 3D y que los puntos se definen en el espacio 3D, vamos a buscar nuestras matrices en el espacio 3D. Por tanto, lo que queremos es una matrix 3×3 que al multiplicar por el vector columna de un punto nos de el resultado:

Domingo Martín Perandrés©

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad \rightarrow \quad \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} a \cdot x + d \cdot y + g \cdot z \\ b \cdot x + e \cdot y + h \cdot z \\ c \cdot x + f \cdot y + i \cdot z \end{bmatrix}$$

Ahora tenemos que plantear las trasformaciones e igualar para poder encontrar los valores de los coeficientes: Vamos a ver cómo se hace con el escalado:

$$\begin{matrix} x' = x \cdot E_x \\ y' = y \cdot E_y \\ z' = z \cdot E_z \end{matrix} \quad = \quad \begin{matrix} x' = a \cdot x + d \cdot y + g \cdot z \\ y' = b \cdot x + e \cdot y + h \cdot z \\ z' = c \cdot x + f \cdot y + i \cdot z \end{matrix} \quad \rightarrow \quad \begin{matrix} x \cdot E_x = a \cdot x + d \cdot y + g \cdot z \\ y \cdot E_y = b \cdot x + e \cdot y + h \cdot z \\ z \cdot E_z = c \cdot x + f \cdot y + i \cdot z \end{matrix}$$

Es fácil ver que los coeficientes que igualan ambos lados de la ecuación son los siguientes:

$$\begin{bmatrix} E_x & 0 & 0 \\ 0 & E_y & 0 \\ 0 & 0 & E_z \end{bmatrix}$$

Podemos aplicar el mismo proceso para las rotaciones y obtener los coeficientes adecuado, pero con el caso de las traslaciones nos encontramos un problema ya que la matriz nos queda de la siguiente manera:

$$\begin{bmatrix} 1 + \frac{T_x}{x} & 0 & 0 \\ 0 & 1 + \frac{T_y}{y} & 0 \\ 0 & 0 & 1 + \frac{T_z}{z} \end{bmatrix}$$

El problema consiste en que los coeficientes de la matriz dependen de las coordenadas del punto que se quieren transformar con la matriz. Esto implica que si tenemos 1 millón de puntos tenemos que crear 1 millón de transformaciones lo cual no tiene sentido en nuestro caso.

Para resolverlo vamos a ampliar la dimensión del espacio en el que vamos a realizar las transformaciones pasando de uno de 3×3 a otro de 4×4 o espacio proyectivo. También se dice que vamos a realizar los cálculos en coordenadas homogéneas. ¿Cómo pasamos un punto en coordenadas 3D a otro 4D? Vamos a usar la siguiente formulación: $x' = x \cdot w$, $y' = y \cdot w$ y $z' = z \cdot w$, para cualquier $w \neq 0$. En tal caso, nos queda lo siguiente: $(x,y,z) \rightarrow (x \cdot w, y \cdot w, z \cdot w, w)$. El paso de 4D a 3D se realiza con la operación inversa: $(x,y,z,w) \rightarrow (\frac{x}{w}, \frac{y}{w}, \frac{z}{w}, 1)$.

Hay que tener en cuenta que a cada punto en coordenadas 3D le corresponden infinitos puntos (una recta) en 4D, al variar el valor de w. Podemos simplificar las operaciones a realizar si elegimos $w = 1$. En tal caso el paso de 3D a 4D es trivial: $(x,y,z) \rightarrow (x,y,z,1)$.

Domingo Martín Perandrés©

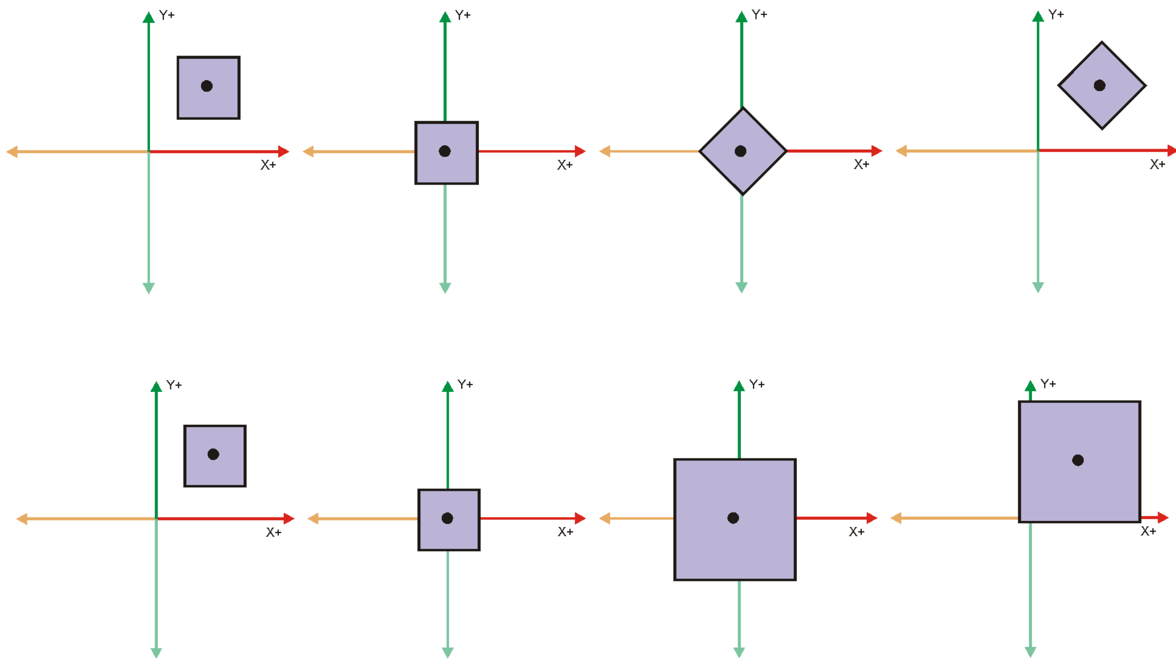


Figura 11.6: Combinación de transformaciones para rotar y escalar con respecto a un punto que no es el origen

Si hemos pasado los puntos a 4D, las matrices que representan a las transformaciones también. En este caso, la traslación se puede integrar fácilmente. Veamos como quedan las distintas matrices de transformación para la traslación y el escalado:

$$\begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} E_x & 0 & 0 & 0 \\ 0 & E_y & 0 & 0 \\ 0 & 0 & E_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Y las de las rotaciones:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) & 0 \\ 0 & \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} cos(\alpha) & 0 & \sin(\alpha) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\alpha) & 0 & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Por tanto, ya hemos visto como es posible crear las matrices que representan a las transformaciones. Y lo más importante, esta forma de escribir las transformaciones tiene las gran ventaja de que el proceso de combinar transformaciones se convierte en una simple multiplicación de matrices. Esto es, si tenemos las siguientes matrices que realizan un escalado M_e, una rotación M_r y una traslación M_t, la matriz que realiza las tres transformaciones es

el producto de las tres: $M' = M_e \cdot M_r \cdot M_t$. Además, se cumple que la inversa de la matriz es la transformación inversa, esto es, la matriz que deshace la transformación, la cual puede ser difícil de calcular. Matemáticamente, $M'^{-1} = M_t^{-1} \cdot M_r^{-1} \cdot M_e^{-1}$.

La gran ventaja que aportan estas dos capacidades hacen que el uso de matrices 4x4 sea la norma en todas las bibliotecas gráficas y en las GPUs, a pesar de que haya que realizar más operaciones.

Ahora podemos comprender el que la transformación de proyección se defina también con una matriz 4x4 y que los vértices se generalicen a 4 dimensiones.

Pasemos por tanto a ver un ejemplo en el que se aplican las transformaciones al cubo, haciendo que se dibuje en una posición distinta de la original y con otra orientación y tamaño. Esto implica que vamos a aplicarle un escalado, una o varias rotaciones y una traslación al cubo original para obtener el resultado final. Nuestro código es muy sencillo, pues ya hemos visto como se dibuja. Lo único que hemos cambiado es que ahora tenemos que definir la matriz de transformación en el programa principal y mandarla al vertex shader.

En los ejemplos anteriores teníamos definidas las tres transformaciones para realizar el modelado, la transformación a la posición de la cámara y la proyección. Las dos últimas las hemos modificado para crear nuestra cámara pero la de modelado, Model, era igual a la identidad y por tanto no producía ningún cambio. Ahora vamos a modificar dicha matriz para cambiar el objeto. También podremos ver ahora que para la creación de la transformación de vista hemos recurrido a las transformaciones geométricas.

Lo que hay que tener en cuenta es que en el vertex shader aparecen las tres matrices que representan a las transformaciones principales: la transformación de modelado, la transformación de la cámara y la transformación de proyección. Esta separación es importante pues nos permite transformar el punto a cualquiera de los sistemas de coordenadas que se definen con las transformaciones. Esto es, con la transformación de modelado pasamos de coordenadas de modelado o maestras a coordenadas de mundo. Con la transformación de cámara, ojo o visión, pasamos de coordenadas de mundo a coordenadas de cámara. Y con la transformación de proyección pasamos de coordenadas de cámara a coordenadas de dispositivo normalizado. Es importante entender que se puede pasar de un sistema de coordenadas a otro mediante la transformación, o ir en la dirección inversa aplicando la transformación inversa.

Una vez que se han explicado las transformaciones geométricas vamos a pasar a ver la infraestructura necesaria para poder aplicarlas a uno o varios objetos. En este caso, tenemos dos objetos, los ejes de referencia y un cubo al que le vamos a aplicar una transformación de modelado, pero la idea es poder añadir más objetos de una forma sencilla. También que mediante los mecanismos de herencia de C++, permita crear objetos más complejos a partir de objetos más sencillos.

Para ello vamos a extender la infraestructura que creamos para el ejemplo del cubo de tal manera que permita ser generalizada a cualquier objeto y cualquier número. Esto implica

Figura 11.7: Secuenciación de objectos en un VBO.

modificar las clases que ya hemos visto y añadir una clase que permita manejar el conjunto de objetos.

Veamos el código de la clase:

```
class _basic_object3d{
public:
  vector<_vertex3f> Vertices;
  vector<_vertex3f> Vertices_colors;

  vector<_vertex3i> Triangles;
  vector<_vertex3f> Triangles_colors;

  vector<_vertex3f> Vertices_drawarray;
  vector<_vertex3f> Vertices_colors_drawarray;

  _basic_object3d();

  virtual void draw(){};
  int num_vertices_drawarray(){return Vertices_drawarray.size();};
  void update_drawing_data(int Initial_position1, int Num_vertices1);

protected:
  int Initial_position;
  int Num_vertices;
};
```

De esta definición, es importante señalar que la función de dibujado se ha definido como virtual, lo cual implica que será una clase derivada la que la implementará. La función num_vertices_drawarray devuelve el número de vértices necesarios para dibujar el modelo si se usa la función glDrawArrays. Como ya hemos visto, en ese caso, para el resto de atributos son necesarios el mismo número de elementos.

La función update_drawing_data(int Initial_position1, int Num_vertices1) recibe los valores y los guarda localmente. Dado que se van a guardar los datos de varios objetos en un mismo buffer, es necesario saber la posición en la que comienzan y hasta donde llegan los datos propios, pues estos son los parámetros que se usan con glDrawArrays. La idea se muesta en la figura 11.7.

Como tenemos varios objetos, y el orden en el que se inserten puede variar, no podemos saber cuales serán estos valores hasta que no tengamos la información del tamaño de todos los Vertices_drawarray para todos los objetos. Para gestionar todos los objetos que pueden ser dibujados y encargarse del cálculo de las posiciones de los mismo se ha creado una clase que se llama _object_management:

```
class _object_management
{
public:

  _object_management();
  void add_object(_basic_object3d* Object1);
  _basic_object3d* object(unsigned int Num_object1);
  void update_objects();
  unsigned int num_objects(){return Num_objects;};
  unsigned int num_vertices_object(unsigned int Num_object1);
  unsigned int num_vertices_total(){return Num_vertices_total;};

protected:
  std::vector<_basic_object3d*> Vec_objects;
  std::vector<int> Vec_num_vertices_drawarray_objects;

  unsigned int Num_objects;
  unsigned int Num_vertices_total;
};
```

Obsérvese cómo mantiene un vector de punteros a los objetos que se pueden crear (ya que derivarán de _basic_object3d).

Veamos cómo son las clases de los dos objetos que vamos a crear. Empezamos con los ejes:

```
class _axis: public _basic_object3d
{
public:

  _axis();
  void draw();
};
```

Solo necesitamos el constructor y la función para dibujar. El constructor de los ejes es el mismo que ya hemos visto. Veamos la función de dibujado:

```
void _axis::draw()
{
  glDrawArrays(GL_LINES,Initial_position,Num_vertices);
}
```

Se puede ver como se usan las variables que describimos previamente para indicar la posición de comienzo dentro del buffer y el número de vértices.

Ahora veamos cómo se puede hacer para crear los objetos y meter la información en los buffers:

```
...
Object_management.add_object(new _axis());
Object_management.add_object(new _cube());
Object_management.update_objects();
...
glCreateBuffers(1,&VBO_vertices1);
glNamedBufferStorage(VBO_vertices1,Object_management.num_vertices_total()*3*sizeof(↩
    GLfloat),nullptr,GL_DYNAMIC_STORAGE_BIT | GL_MAP_WRITE_BIT);
```

```
glVertexArrayVertexBuffer(VAO1,0,VBO_vertices1,0,3*sizeof(GLfloat));
glVertexArrayAttribFormat(VAO1,0,3,GL_FLOAT,GL_FALSE,0);
glVertexArrayAttribBinding(VAO1,0,0);
glEnableVertexArrayAttrib(VAO1,0);
...
// Put data
// vertices
int Counter=0;
for (unsigned int i=0;i<Object_management.num_objects();i++){
  glNamedBufferSubData(VBO_vertices1,Counter*3*sizeof(GLfloat),Object_management.↩
      num_vertices_object(i)*3*sizeof(GLfloat),&Object_management.object(i)->↩
      Vertices_drawarray[0]);
  Counter+=Object_management.num_vertices_object(i);
}
...
```

Podemos ver cómo se crean los objetos _axis y _cube y se añaden al manejador de objetos. Mediante la función update_objects se actualizan las posiciones para cada objeto dentro de los buffers. A continuación, con los datos calculados, podemos crear los VBOs conociendo el número total de elementos que se necesitan y actualizar su contenido.

Sólo nos queda la parte relacionada con el dibujado. Una de las variaciones que se incluyen es que ya no se tiene una sola matriz de transformación sino tres: la transformación de modelado, la transformación de la cámara y la transformación de proyección.

Hasta ahora, la transformación de modelado ha sido la identidad. En este ejemplo vamos a aplicarle al cubo dos rotaciones y una traslación.

Para dibujar los ejes tenemos que hacer lo siguiente:

```
//
glUniformMatrix4fv(10,1,GL_FALSE,Model.data());
glUniformMatrix4fv(11,1,GL_FALSE,View.data());
glUniformMatrix4fv(12,1,GL_FALSE,Projection.data());

// axis
glUniform1i(20,MODE_FILL);
glUniform1i(21,(int)MODE_COLORS_VARIABLE);

Object_management.object(0)->draw();
```

Y para dibujar el cubo, en modo relleno, se hace lo siguiente:

```
Model.translate(0.5,0.25,0.25);
Model.rotate(35,1,0,0);
Model.rotate(45,0,1,0);
Model.scale(1,2,1);

glUniformMatrix4fv(10,1,GL_FALSE,Model.data());
...
if (Draw_fill){
  glPolygonMode(GL_FRONT_AND_BACK,GL_FILL);
  glUniform1i(20,MODE_FILL);
  glUniform1i(21,Mode);
  glUniform3fv(27,1,(GLfloat*) &COLORS[COLOR_FILL]);

  for (unsigned int i=1;i<Object_management.num_objects();i++){
```

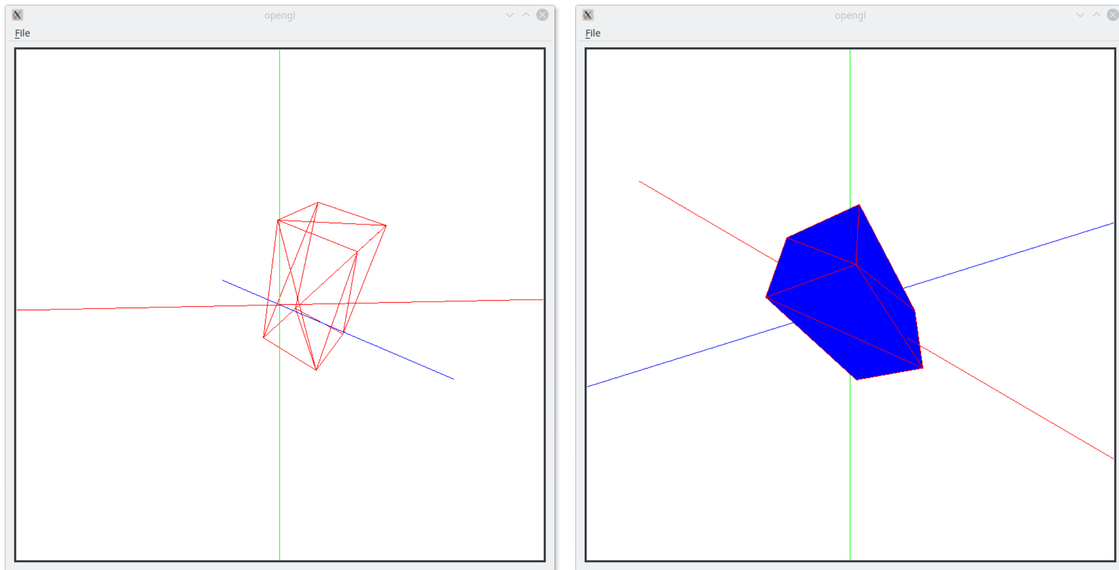


Figura 11.8: Dos vistas del resultado del ejemplo 11

```
    Object_management.object(i)->draw();
  }
}
```

Los shaders son los mismos que los usados en el tema anterior.

12

Modelado jerárquico sin movimiento

En el ejemplo anterior hemos planteado los pilares para poder construir objetos mediante lo que se llama modelado jerárquico. La idea es muy sencilla: construir objetos más complejos mediante la utilización de objetos más simples. En nuestro ejemplo queremos construir un coche de apariencia muy simple a partir de un cubo, un cono y un cilindro. A partir del cubo crearemos el chasis mediante dos piezas, las cuatro ruedas se van a crear a partir del cilindro y los dos faros a partir del cono. Se puede ver el resultado en la figura 12.

El problema que tenemos que resolver es entender cómo a partir de un sólo objeto definido de cierta manera y posición podemos generar otros que pueden tener distinto tamaño, orientación y posición. La respuesta está en el uso de las transformaciones.

Como hemos visto en el ejemplo anterior, a partir de un cubo unidad centrado en el origen hemos conseguido visualizar otro cubo que tenía diferente tamaño, orientación y posición. ¿Qué ocurriría si se cambia la transformación de modelado y se vuelve a dibujar el cubo sin borrar la escena? Pues que se obtendría un segundo cubo. ¿Y si añado una tercera transformación y se vuelve a dibujar el cubo? Se añadiría un tercer cubo a la imagen. Es fácil ver que se pueden dibujar todos los cubos transformados que se deseen. Y lo que es más importante, sólo se ha necesitado un objeto cubo para poder hacerlo.

Esta es la idea de los símbolos y las instancias: un símbolo se define una sola vez pero se puede dibujar, instanciar, todas las que se desee. La instanciación se lleva a cabo mediante la aplicación de una transformación que, normalmente y en su forma más general, se compone de un escalado, una o varias rotaciones y una traslación. Por tanto ya conocemos el mecanismo básico para modelar: se crea una matriz de transformación y se dibuja el objeto, al cual se le aplicará la transformación.

Ahora tenemos que entender cómo podemos usar objetos más sencillos para construir objetos más complejos. La idea clave es la de jerarquización, esto es, establecer dependencias. Veámoslo con un ejemplo real: el movimiento de uno de nuestros brazos. Si nos fijamos, en

Domingo Martín Perandrés©

un brazo existe una relación de dependencia en el movimiento de las articulaciones: puedo mover independientemente un dedo, pero si muevo la mano implica que el dedo se moverá junto con la mano. Igualmente, si muevo el radio y el cúbito se moverá la mano, y si muevo el húmero se moverán el radio y el cúbito. Existe una dependencia del movimiento de tal manera que el movimiento de una parte se transmite a aquellas de las que se compone. Esta idea se puede extender al modelado.

Por ejemplo, a partir del cilindro puedo crear el objeto rueda, y a partir del objeto rueda puedo crear las cuatro ruedas del coche. Con el cono podría crear el objeto faro y usarlo dos veces. Con el cubo crearía el chasis. Con todas esas piezas podría crear el objeto coche, y a partir de un coche dibujar muchos coches para crear una carretera llena de ellos.

Hay que observar que el modelado jerárquico busca la reutilización de los elementos que se tienen haciendo que el modelado sea más sencillo.

Para poder implementar nuestro coche necesitamos crear dos tipos nuevos de objetos: un cilindro y un cono. Ambos son objetos que se pueden generar por revolución, tal y como vimos con la esfera. La diferencia está en el perfil que se rota. Por tanto vamos a crear una clase _revolution_object3d que deriva de _basic_object3d. El código es el siguiente:

```
class _revolution_object3d: public _basic_object3d
{
public:

void create(unsigned int Num_horizontal_divisions1, vector<_vertex3f> &Profile);

protected:
int linear_position(int Column,int Row){return (Column*(Num_vertical_divisions)+Row);};

int Num_horizontal_divisions;
int Num_vertical_divisions;
};
```

Simplemente tiene una función que dados el perfil generador y el número de divisiones horizontales crea el objeto por revolución. A partir de esta clase se puede construir los objetos cilindro y cono, y también, por ejemplo, la esfera.

Dado que estos objetos derivan de _basic_object3d se pueden usar con el manejador de objetos, y meter toda la información en los buffers tal y como vimos en el ejemplo anterior.

Vamos a intentar construir la rueda y luego usarla para ver qué problemas surgen y cómo resolverlos. Lo primero que tenemos que apreciar es que la rueda es un cilindro al que le hemos cambiado la dimensiones y la orientación. En concreto, vamos a construir la rueda aplicándole un escalado con los valores $(1,0.4,1)$ y una rotación de 90 grados con respecto al eje x. Para hacer que la rueda esté apoyada sobre el suelo tenemos que incluir una traslación con los valores $(0,0.5,0)$.

Para ver cómo se tiene que llamar al cilindro para crear la rueda, veamos previamente como sería la carga de los objetos que vamos a utilizar, esto es, los ejes, el cubo, el cilindro y el cono. Usaremos el procedimiento expuesto en el ejemplo anterior:

```
_axis *Axis=new _axis;
_cube *Cube=new _cube;
_cylinder *Cylinder=new _cylinder;
_cone *Cone=new _cone

// each drawable object must be added
Object_management.add_object(Axis);
Object_management.add_object(Cube);
Object_management.add_object(Cylinder);
Object_management.add_object(Cone);

// update the positions in the buffer
Object_management.update_objects();
```

De esta manera sabemos que el cilindro es el tercer objeto y se direcciona con la posición 2. Por tanto, para dibujar la rueda habría que hacer lo siguiente:

```
Model.traslate(0,0.5,0);
Model.rotate(90,1,0,0);
Model.scale(1,0.4,1);
..
glUniformMatrix4fv(10,1,GL_FALSE,Model.data());
...
Object_management.object(2)->draw();
```

Con esto dibujamos una rueda, pero que está localizada en el origen y apoyada en el suelo, el plano $y = 0$. Para crear las ruedas del coche es necesario mover la rueda a cuatro posiciones diferentes lo cual se realiza mediante traslaciones. Imaginemos que las ruedas se colocan simetricamente con respecto al plano $x = 0$. Para dibujarlas tendría que utilizar las transformaciones que he creado para obtener la rueda y añadir la traslación. Sería algo parecido a esto:

```
// wheel 1
Model.traslate(-3,0,2);
Model.traslate(0,0.5,0);
Model.rotate(90,1,0,0);
Model.scale(1,0.4,1);
..
glUniformMatrix4fv(10,1,GL_FALSE,Model.data());
...
Object_management.object(2)->draw();

// wheel 2
Model.traslate(-3,0,-2);
Model.traslate(0,0.5,0);
Model.rotate(90,1,0,0);
Model.scale(1,0.4,1);
..
glUniformMatrix4fv(10,1,GL_FALSE,Model.data());
...
Object_management.object(2)->draw();

// wheel 3
```

```
Model.traslate(3,0,2);
Model.traslate(0,0.5,0);
Model.rotate(90,1,0,0);
Model.scale(1,0.4,1);
..
glUniformMatrix4fv(10,1,GL_FALSE,Model.data());
...
Object_management.object(2)->draw();

// wheel 4
Model.traslate(3,0,-2);
Model.traslate(0,0.5,0);
Model.rotate(90,1,0,0);
Model.scale(1,0.4,1);
..
glUniformMatrix4fv(10,1,GL_FALSE,Model.data());
...
Object_management.object(2)->draw();
```

Se puede probar que el código funciona pero también que hay que repetir mucho código que además es siempre el mismo. En concreto, el código que permite obtener la rueda a partir del cilindro. Si se consiguiera, por un lado, encapsular el código de la rueda como si fuera un nuevo objeto y además encontrara la forma de que las transformaciones que se definieran en un nivel superior se pudieran pasar a los objetos que se encontraran en un nivel superior, el código que acabo de escribir sería más sencillo:

```
// wheel 1
Model.traslate(-3,0,2);
draw_wheel();

// wheel 2
Model.traslate(-3,0,-2);
draw_wheel();

// wheel 3
Model.traslate(3,0,2);
draw_wheel();

// wheel 4
Model.traslate(3,0,-2);
draw_wheel();
```

En este pseudocódigo suponemos que la función draw_wheel() dibuja la rueda tal y como la hemos definido, y también que la transformación que se ha definido previamente a la llamada le afecta, pero de tal manera que se mantiene el orden correcto y se obtiene el resultado deseado. En este caso, la traslación se aplicaría a la rueda. Para desarrollar la solución correcta hay que tener en cuenta que el mecanismo que se diseñe debe permitir que se pueda usar en cualquier número de niveles. Esto es, que una vez creado un objeto, se puede crear uno nuevo, que puede ser usado para crear uno nuevo, etc., etc.

Una posible idea sería la de definir cada objeto como una función o una clase que pudiera ser reutilizada, y además, que se pudiera pasar la transformación a la función de dibujado. Por ejemplo, la función para dibujar la rueda podría ser la siguiente:

```
void draw_wheel(QMatrix4x4 M)
```

```
{
  QMatrix4x4 Model;

  Model=Model*M;
  Model.traslate(0,0.5,0);
  Model.rotate(90,1,0,0);
  Model.scale(1,0.4,1);
  ..
  glUniformMatrix4fv(10,1,GL_FALSE,Model.data());
  glUniformMatrix3fv(13,1,GL_FALSE,Model.normalMatrix().data());
  ...
  Object_management.object(2)->draw();
}
```

Y la forma de llamarla sería:

```
// wheel 1
QMatrix4x4 Model;
Model.traslate(-3,0,2);
draw_wheel(Model);
```

Obviando el problema de que para dibujar la rueda se tiene que conocer la posición del objeto cilindro, el esquema funciona pues permite crear funciones que llamen a funciones, que llamen a otras funciones, etc. Es importante observar que lo que importa es que las trasformaciones que se definen en un nivel superior lleguen al o los niveles inferiores. Si nos fijamos bien, en el ejemplo que hemos planteado para dibujar las 4 ruedas, cada vez que se va a dibujar una rueda se define su transformación que es pasada a la función. Pero si queremos generalizar tenemos que considerar que a la función que dibuja las 4 ruedas también le ha llegado una transformación de una función que se encuentra en un nivel superior. Otro detalle a tener en cuenta es que si desde una función de un nivel se llama a otra de un nivel inferior, enviando la transformación, cuando se regresa del nivel inferior se debe recuperar la transformación que hubiera. Esto es, si el nivel n tiene la transformación T, que proviene del nivel m, y se llama una función de un nivel inferior o, pasándole T más las transformaciones propias del nivel n, por ejemplo T', esto es, pasando $T \cdot T'$, del desarrollo que hemos visto está claro que cuando se regrese de la ejecución del nivel o, se deberá recuperar la transformación T para que pueda volver a ser usada en otras llamadas a funciones de nivel inferior.

El mecanismo que permite hacer este paso y recuperación es la pila. En los ejemplos que hemos mostrado había un mecanismo de pila que no hemos programado nosotros pero que ha servido para nuestros intereses: las llamadas a función, en las que sabemos que el paso y recuperación de los parámetros se implementa mediante una pila. Para nuestra implementación, aunque sea posible usar el envio de las transformaciones en combinación con la pila de las funciones, vamos a hacerlo mucho más explicito mediante la construcción y uso de una pila de transformaciones. Además de esta manera seguimos la forma de operar que tienen las versiones antiguas de OpenGL.

Para ello sólo tenemos que crear una clase que permita usar las funciones de push y pop típicas de las pilas, y además le hemos añadido la posibilidad de modificar el top de la

pila combinándolo con las diferentes transformaciones que hemos visto. El código sería el siguiente:

```
class _pile
{
public:
  _pile();
  void push();
  void pop();
  void traslate(float x,float y,float z);
  void rotate(float Angle,float x,float y,float z);
  void scale(float x,float y,float z);
  float *top(){
    if (Top>=0) return Vec_matrices[Top].data();
    else return nullptr;
  };
  QMatrix4x4 top_matrix(){return Vec_matrices[Top];};

protected:
  std::vector<QMatrix4x4> Vec_matrices;
  int Top;
};
```

Las únicas funciones que hay que explicar son el push, el pop y top. La función push aumentan en 1 el Top de la pila (Top con mayúscula es la variable), y copia el contenido de la posición anterior. Esta es la manera de poder preservar la transformación que se tiene localmente para poder recuperarla cuando se haga pop, y también, al copiar el contenido, es la forma de enviar la transformación a niveles inferiores. La función pop simplemente decrementa en 1 el Top. Por último, la función top devuelve los valores de la matriz de transformación que se encuentre en el Top para que pueda ser usada por OpenGL. Un detalle a tener en cuenta es que al crear la pila se mete la transformación identidad por defecto.

Si ahora hacemos que la instancia de la clase pila sea global para todos los objetos que se van a usar y las funciones que se van a crear, la forma de utilizarla es muy sencilla. En su forma más genérica, y como ejemplo, sería así:

```
void draw_m()
{
Pile.push();
Pile.traslate(1,2,3);
draw_n();
Pile.pop();

Pile.push();
Pile.scale(2,2,1);
draw_o();
Pile.pop();
}
...
void draw_n()
{
Pile.push();
Pile.rotate(45,1,0,0);
draw_p();
Pile.pop();
}
...
```

La idea está clara pero nos falta por resolver un par de pequeños problemas. El primero es que en las hojas de la estructura jerárquica se van a encontrar las funciones que de verdad producen el dibujado de un objeto, que ahora deben sufrir la transformación que se encuentre en el Top de la pila. Por tanto, debemos modificar las funciones de dibujado para que se actualicen las matrices con los valores apropiados. Por ejemplo, para el caso del cilindro sería asi:

```
void _cylinder::draw()
{
  glUniformMatrix4fv(10,1,GL_FALSE,Pile.top());
  glDrawArrays(GL_TRIANGLES,Initial_position,Num_vertices);
}
```

Obtenemos los valores de la matriz que está en el top de la pila, los pasamos al vertex shader y por último dibujamos el objeto, el cual conoce sus parámetros, posición y tamaño, con respecto al VBO.

El otro problema a resolver es que hasta ahora hemos usado como clase base para crear cualquier tipo de objeto o función la clase _basic_object3d. Esta clase va bien si el objeto que vamos a crear funciona como un símbolo que puede ser dibujado, pero si en vez de eso funciona como una clase que simplemente cambia la pila y hace llamada a otras funciones, resulta excesiva. La solución pasa por crear una clase base mucho más sencilla que permita hacer las llamadas de una manera eficiente, pero que admita la inclusión de símbolos. El código para la clase básica es:

```
class _basic_object
{
public:
  void add_symbol(_basic_object* Basic_object1){Vec_objects.push_back(Basic_object1);}
  virtual void draw(){};

protected:
  vector<_basic_object*> Vec_objects;
};
```

Lo importante es que ahora una instancia del tipo _basic_object puede llamar a otras instancias del mismo tipo. Y además, la función para dibujar, draw, es virtual, con lo cual cada objeto implementará la suya. Si hacemos que la clase _basic_object3d derive de _basic_object, gracias a la herencia y el polimorfismo tendremos todos los elementos para poder implementar la jerarquía del coche, la cual se muestra a continuación en detalle. Empezamos por el foco:

```
class _focus:public _basic_object
{
public:
  void draw();
};
...
void _focus::draw()
{
  Pile.push();
```

```
  Pile.rotate(-90,0,0,1);
  Pile.scale(.4,.3,.4);
  Vec_objects[0]->draw();
  Pile.pop();
}
```

Ahora la rueda:

```
class _wheel:public _basic_object
{
public:
  void draw();
};
...
void _wheel::draw()
{
  Pile.push();
  Pile.rotate(90,1,0,0);
  Pile.scale(1,.4,1);
  Vec_objects[0]->draw();
  Pile.pop();
}
```

El chasis:

```
class _chasis:public _basic_object
{
public:
  void draw();
};
...
void _chasis::draw()
{
  glUniform3fv(27,1,(GLfloat*) &COLORS[2]); // green
  Pile.push();
  Pile.traslate(0,1,0);
  Pile.scale(5,2,3);
  Vec_objects[0]->draw();
  Pile.pop();

  glUniform3fv(27,1,(GLfloat*) &COLORS[3]); // blue
  Pile.push();
  Pile.traslate(0.5,2.5,0);
  Pile.scale(2,1,3);
  Vec_objects[0]->draw();
  Pile.pop();
}
```

Por último el coche completo:

```
class _car:public _basic_object
{
public:
  void draw();
};
...
void _car::draw()
{
  // chasis
  Pile.push();
```

```
  Pile.traslate(0,.5,0);
  Vec_objects[0]->draw();
  Pile.pop();

  // wheels
  glUniform3fv(27,1,(GLfloat*) &COLORS[1]); //red

  Pile.push();
  Pile.traslate(-1.5,.5,1.5);
  Vec_objects[1]->draw();
  Pile.pop();

  Pile.push();
  Pile.traslate(-1.5,.5,-1.5);
  Vec_objects[1]->draw();
  Pile.pop();

  Pile.push();
  Pile.traslate(1.5,.5,1.5);
  Vec_objects[1]->draw();
  Pile.pop();

  Pile.push();
  Pile.traslate(1.5,.5,-1.5);
  Vec_objects[1]->draw();
  Pile.pop();

  // focus
  glUniform3fv(27,1,(GLfloat*) &COLORS[5]); // magenta

  Pile.push();
  Pile.traslate(-2.4,2,1);
  Vec_objects[2]->draw();
  Pile.pop();

  Pile.push();
  Pile.traslate(-2.4,2,-1);
  Vec_objects[2]->draw();
  Pile.pop();
}
```

Es importante entender que cada objeto de la jerarquía puede hacer uso de otros objetos, pero para ello debe poder acceder a los mismos mediante un puntero. Por ejemplo, el objeto rueda se construye a partir del objeto cilindro. Por tanto, el objeto rueda tiene un puntero al objeto cilindro para poder llamarlo. De esta manera estamos implementado realmente la idea de que hay un solo símbolo que se reusa numerosas veces.

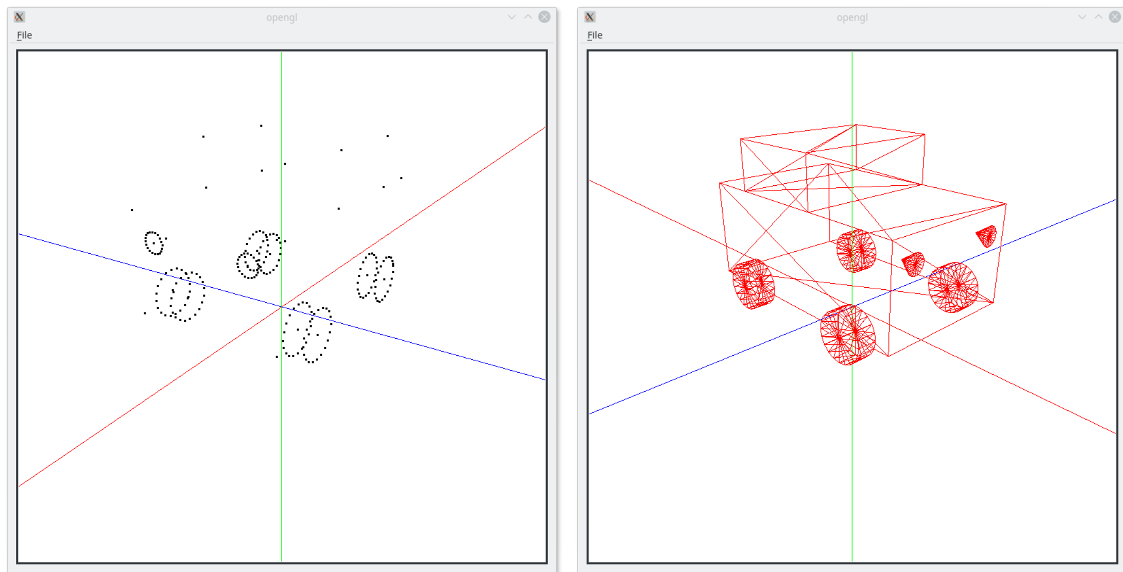
Por tanto, el proceso de inicialización sería de la siguiente manera:

```
_axis *Axis=new _axis();
_cube *Cube=new _cube();
_cylinder *Cylinder=new _cylinder(0.5,0.5,20);
_cone *Cone=new _cone(0.5,0.5,20);

// each drawable object must be added
Object_management.add_object(Axis);
Object_management.add_object(Cube);
Object_management.add_object(Cylinder);
Object_management.add_object(Cone);

// update the positions in the buffer
Object_management.update_objects();
```

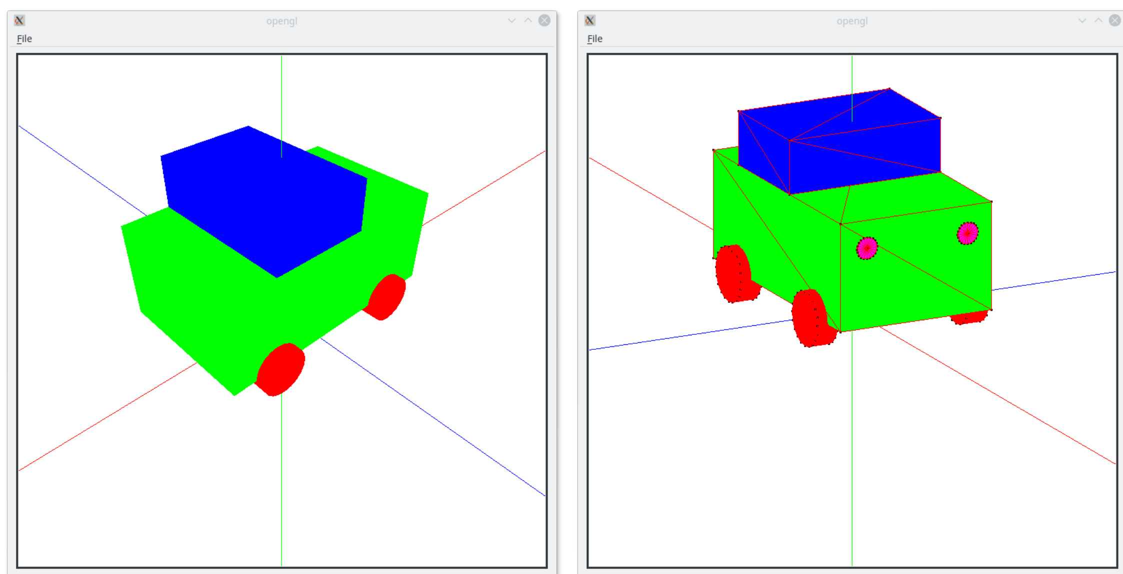
```
// now is possible to draw
_wheel *Wheel=new _wheel();
Wheel->add_symbol(Cylinder);

_focus *Focus=new _focus();
Focus->add_symbol(Cone);

_chasis *Chasis=new _chasis();
Chasis->add_symbol(Cube);

_car *Car=new _car();
Car->add_symbol(Chasis);
Car->add_symbol(Wheel);
Car->add_symbol(Focus);

// each object that want to be drawn is added
Vec_objects.push_back(Axis);
Vec_objects.push_back(Car);
```

Las instrucciones para dibujar son las siguientes:

```
for (unsigned int i=1;i<Vec_objects.size();i++){
  Vec_objects[i]->draw();
}
```



(a) Puntos (b) Líneas



(c) Relleno (d) Todo y proyección paralela

Figura 12.1: Resultado del ejemplo 12 con los diferentes modo de visualización

Domingo Martín Perandrés©

13

Modelado jerárquico con movimiento

Una vez que hemos visto como crear una estructura jerárquica estática, vamos a incluir lo necesario para que el modelo se pueda mover. Para ello vamos a construir un brazo de robot con 4 grados de libertad. El movimiento de cada componente se realizará pulsando las teclas apropiadas.

Dado que lo que queremos es producir el movimiento del modelo, la primera pregunta que tenemos que resolver es cómo se consigue producir dicho movimiento. La respuesta es mediante la aplicación de transformaciones. Por ejemplo, si queremos que el modelo se mueva de un sitio a otro aplicaremos una traslación. Si queremos que gire aplicaremos una rotación. La diferencia con las transformaciones que hemos usado en el ejemplo anterior está en que los valores de los parámetros de dichas transformaciones cambian con el tiempo. En el caso que queramos que el coche vaya del punto a al punto b, tenemos que hacer que la traslación vaya cambiando de tal manera que al principio esté en a y mediante, por ejemplo, un movimiento lineal, llegue al punto b. Por tanto, al pulsar las teclas de control de movimiento lo que vamos a hacer es cambiar el valor de los parámetros, en la línea de lo que hemos hecho para controlar la posición de la cámara.

Dado que la idea es muy sencilla, vamos a aprovechar para repasar el proceso de modelado construyendo paso a paso el brazo de robot, y comentado las transformaciones que son necesarias para producir el movimiento. El robot se muestra en la figura 13.6. Como se puede observar es muy sencillo y sólo necesita de un cubo para su construcción. Las piezas de la jerarquía son las siguientes:

- La pinza
- La mano
- El brazo1

Domingo Martín Perandrés©

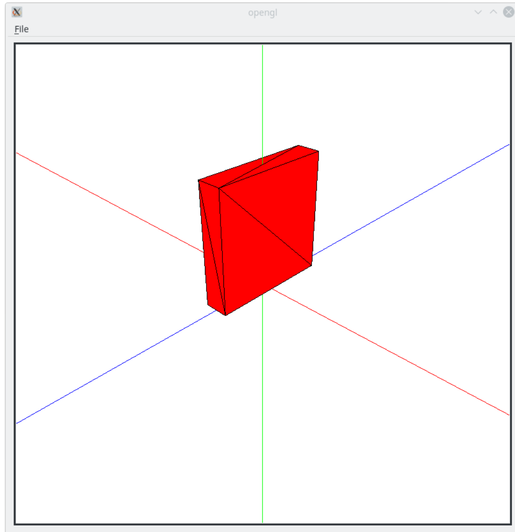


Figura 13.1: Pinza

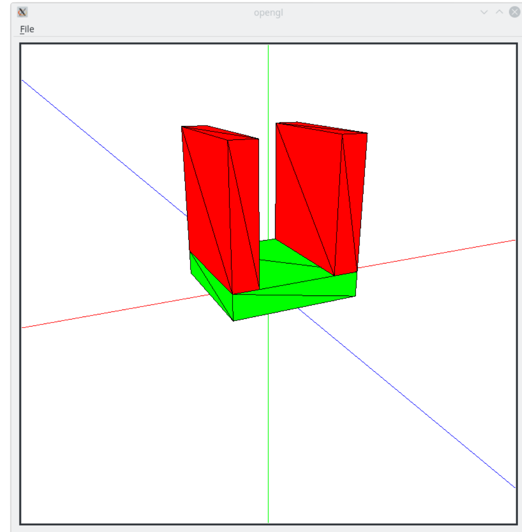


Figura 13.2: Mano

- El brazo2
- La base

La jerarquía de dependencia es muy sencilla: pinza $\leftarrow$ mano $\leftarrow$ brazo1 $\leftarrow$ brazo2 $\leftarrow$ base.

Uno de los detalles a tener en cuenta para poder introducir el movimiento es que ahora las transformaciones deben cambiar. En el código del ejemplo del coche, las clases de cada pieza tenían transformaciones fijas, con sus valores conocidos en tiempo de compilación. Estas eran las transformaciones de modelado. El problema que nos encontramos es que queremos cambiar los valores de los parámetros concretos de cada clase. Esto implica que ahora las clases deben poder cambiar dichos valores. En nuestra implementación vamos a añadir variables globales a cada clase. Esta variables se usarán para calcular las transformaciones en cada momento. Para la creación de la jerarquía usaremos las mismas estructuras que anteriormente.

Para entenderlo mejor, y como forma de trabajo recomendada, vamos a crear las piezas desde la más sencilla, la hoja de la jerarquía, hasta la más compleja, la raíz de la jerarquía.

La primera pieza que vamos a construir es la pinza. Veamos la definición:

```
class _clamp: public _basic_object3d
{
public:
  void draw();
};
```

Y la implementación de la función de dibujado.

```
void _clamp::draw()
{
  glUniform3fv(27,1,(GLfloat*) &COLORS[1]);
  Pile.push();
  Pile.traslate(0,0.5,0);
  Pile.scale(0.2,1,1);
  glUniformMatrix4fv(10,1,GL_FALSE,Pile.top());
  Vec_objects[0]->draw();
  Pile.pop();
}
```

El objeto que se obtiene se puede ver en la figura 13.1. Una vez que tenemos la pieza de la pinza vamos a crear la mano que se compone de una palma y dos pinzas. Estás pinzas son móviles y se pueden acercar o alejar. El movimiento lo realizamos en el eje x. Dado que la mano se diseña de tal manera que está centrada, la forma de obtener el movimiento es muy fácil. Veamos el código de la mano:

```
class _hand: public _basic_object3d
{
public:
  void draw();
  float Translation=0;
};
```

La primera diferencia es que hemos declarado una variable que se encargará de mantener el valor de la traslación de cada pinza. Al ser simétrico el movimiento basta con una sola variable. Veamos ahora la implementación del dibujado.

```
void _hand::draw()
{
  // palma
  glUniform3fv(27,1,(GLfloat*) &COLORS[2]);
  Pile.push();
  Pile.traslate(0,0.1,0);
  Pile.scale(1,0.2,1);
  glUniformMatrix4fv(10,1,GL_FALSE,Pile.top());
  Vec_objects[0]->draw();
  Pile.pop();

  // pinza 1
  Pile.push();
  Pile.traslate(-Translation,0,0);
  Pile.traslate(0.4,0.2,0);
  Vec_objects[1]->draw();
  Pile.pop();

  // pinza2
  Pile.push();
  Pile.traslate(Translation,0,0);
  Pile.traslate(-0.4,0.2,0);
  Vec_objects[1]->draw();
  Pile.pop();
}
```

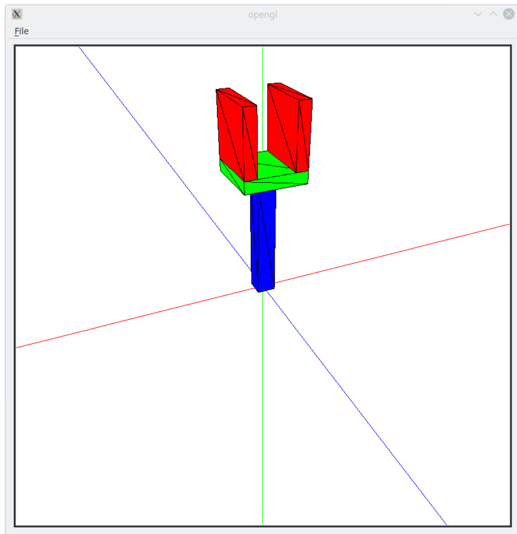


Figura 13.3: Brazo1

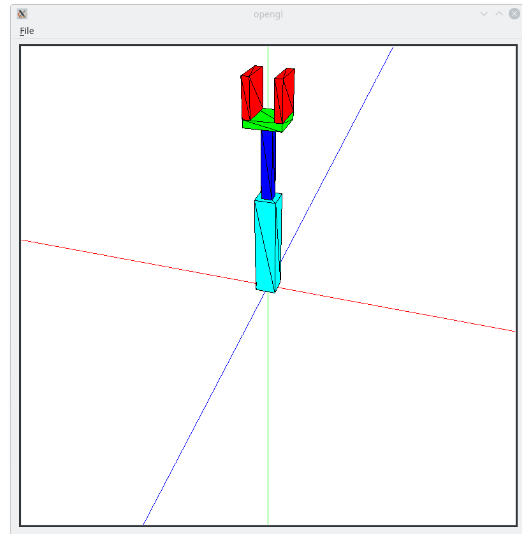


Figura 13.4: Brazo2

Obsérvese como se hace uso de la variable Translation para modificar la posición de las pinzas haciendo que se trasladen con respecto al eje x. No hay ningún tipo de control sobre el valor de dicha variable por lo que puede ocurrir que se salgan de la palma o se crucen. En una implementación más elaborada estos detalles se deben tener en cuenta. El resultado se puede ver en la figura 13.2.

Sigamos ascendiendo en la jerarquía. La siguiente pieza es el brazo1.

```
class _arm1: public _basic_object3d
{
public:
  void draw();
  float Angle=0;
};
```

Y el dibujado:

```
void _arm1::draw()
{
  glUniform3fv(27,1,(GLfloat*) &COLORS[3]);
  Pile.push();
  Pile.traslate(0,1,0);
  Pile.scale(.3,2,0.3);
  glUniformMatrix4fv(10,1,GL_FALSE,Pile.top());
  Vec_objects[0]->draw();
  Pile.pop();

  Pile.push();
  Pile.traslate(0,2,0);
  Pile.rotate(Angle,0,1,0);
  Vec_objects[1]->draw();
  Pile.pop();
}
```

La forma de operar es ya siempre la misma. En este caso el movimiento que queremos es un giro de la mano con respecto al eje y. El resultado se puede ver en la figura 13.3. También a tener en cuenta es el hecho de que el movimiento de una pieza se suele hacer en la pieza que la referencia o padre. En este caso, brazo1 es quien gira a mano.

Pasamos al brazo2:

```
class _arm2: public _basic_object3d
{
public:
  void draw();
  float Angle=0;
};
```

Y como se dibuja:

```
void _arm2::draw()
{
  glUniform3fv(27,1,(GLfloat*) &COLORS[4]);
  Pile.push();
  Pile.traslate(0,1.5,0);
  Pile.scale(.6,3,0.6);
  glUniformMatrix4fv(10,1,GL_FALSE,Pile.top());
  Vec_objects[0]->draw();
  Pile.pop();

  Pile.push();
  Pile.traslate(0,3,0);
  Pile.rotate(Angle,1,0,0);
  Vec_objects[1]->draw();
  Pile.pop();
}
```

El resultado se puede ver en la figura 13.4.

Por último vamos a crear la base que es la raíz de la jerarquía:

```
class _base: public _basic_object3d
{
public:
  void draw();
  float Angle_base;
  float Angle_arm;
};
```

La principal diferencia es que hay dos variables para girar el brazo2, con respecto al eje x y respecto al eje y. El código del dibujado es el siguiente:

```
void _base::draw()
{
  glUniform3fv(27,1,(GLfloat*) &COLORS[5]);
  Pile.push();
  Pile.traslate(0,.2,0);
  Pile.scale(2,.4,2);
  glUniformMatrix4fv(10,1,GL_FALSE,Pile.top());
```

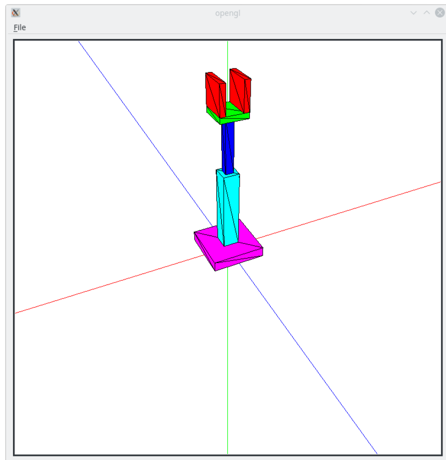


Figura 13.5: Base

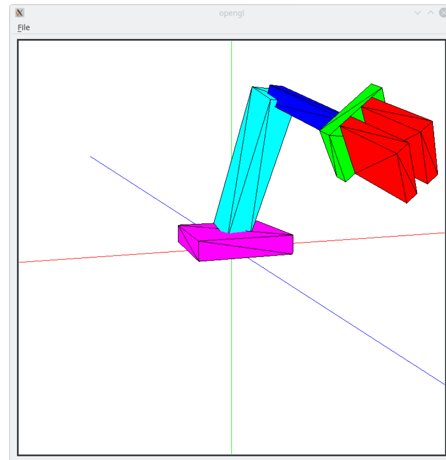


Figura 13.6: Ejemplo completo con movimiento

```
  Vec_objects[0]->draw();
  Pile.pop();

  Pile.push();
  Pile.traslate(0,.4,0);
  Pile.rotate(Angle_base,0,1,0);
  Pile.rotate(Angle_arm,1,0,0);
  Vec_objects[1]->draw();
  Pile.pop();
}
```

La jerarquía completa se puede ver en la figura 13.5, y aplicando el movimiento en figura 13.6

Veamos ahora cómo hacemos para crear la estructura. Para ello hacemos uso de las herramientas creadas en el ejemplo anterior, pero para añadir más flexibilidad, además de tener un vector con todos los objetos que se han creado, Vec_objects, vamos a tener un vector de objetos que se van a dibujar, Vec_render_objects.

El código que crea los objetos y la jerarquía es el siguiente:

```
void _gl_widget::initialize_objects()
{
  _axis *Axis=new _axis();
  _cube *Cube=new _cube();

 // add the real objects
  Object_management.add_object(Axis);
  Object_management.add_object(Cube);

  // update the positions in the buffer
  Object_management.update_objects();

  // now create the hierarchy
  _clamp *Clamp=new _clamp();
```

```
Clamp->add_symbol(Cube);

_hand *Hand=new _hand();
Hand->add_symbol(Cube);
Hand->add_symbol(Clamp);

_arm1 *Arm1=new _arm1();
Arm1->add_symbol(Cube);
Arm1->add_symbol(Hand);

_arm2 *Arm2=new _arm2();
Arm2->add_symbol(Cube);
Arm2->add_symbol(Arm1);

_base *Base=new _base();
Base->add_symbol(Cube);
Base->add_symbol(Arm2);

// each object is added
Vec_objects.push_back(Axis); // 0
Vec_objects.push_back(Clamp); // 1
Vec_objects.push_back(Hand); // 2
Vec_objects.push_back(Arm1); // 3
Vec_objects.push_back(Arm2); // 4
Vec_objects.push_back(Base); // 5

// now add the objects that must be rendered
Vec_render_objects.push_back(Axis);
Vec_render_objects.push_back(Base);

//
_shaders Shader;

Program1=Shader.load_shaders("shaders/example36.vert","shaders/example36.frag");
if (Program1==0){
exit(-1);
}

glCreateVertexArrays(1,&VAO1);
glBindVertexArray(VAO1);

glCreateBuffers(1,&VBO_vertices1);
glNamedBufferStorage(VBO_vertices1,Object_management.num_vertices_total()*3*sizeof(↩
    GLfloat),nullptr,GL_DYNAMIC_STORAGE_BIT | GL_MAP_WRITE_BIT);
glVertexArrayVertexBuffer(VAO1,0,VBO_vertices1,0,3*sizeof(GLfloat));
glVertexArrayAttribFormat(VAO1,0,3,GL_FLOAT,GL_FALSE,0);
glVertexArrayAttribBinding(VAO1,0,0);
glEnableVertexArrayAttrib(VAO1,0);

glCreateBuffers(1,&VBO_colors1);
glNamedBufferStorage(VBO_colors1,Object_management.num_vertices_total()*3*sizeof(↩
    GLfloat),nullptr,GL_DYNAMIC_STORAGE_BIT | GL_MAP_WRITE_BIT);
glVertexArrayVertexBuffer(VAO1,1,VBO_colors1,0,3*sizeof(GLfloat));
glVertexArrayAttribFormat(VAO1,1,3,GL_FLOAT,GL_FALSE,0);
glVertexArrayAttribBinding(VAO1,1,1);
glEnableVertexArrayAttrib(VAO1,1);

// Put data
// vertices
int Counter=0;
for (unsigned int i=0;i<Object_management.num_objects();i++){
glNamedBufferSubData(VBO_vertices1,Counter*3*sizeof(GLfloat),Object_management.↩
    num_vertices_object(i)*3*sizeof(GLfloat),&Object_management.object(i)->↩
    Vertices_drawarray[0]);
Counter+=Object_management.num_vertices_object(i);
}
```

```
  // colors
  Counter=0;
  for (unsigned int i=0;i<Object_management.num_objects();i++){
  glNamedBufferSubData(VBO_colors1,Counter*3*sizeof(GLfloat),Object_management.↩
      num_vertices_object(i)*3*sizeof(GLfloat),&Object_management.object(i)->↩
      Vertices_colors_drawarray[0]);
  Counter+=Object_management.num_vertices_object(i);
  }

  glBindVertexArray(0);
}
```

Para dibujar mantenemos el código del ejemplo anterior, solo cambiando lo siguiente:

```
...
for (unsigned int i=1;i<Vec_render_objects.size();i++){
  Vec_render_objects[i]->draw();
}
...
```

¿Cómo hacemos para cambiar los valores de las variables que controlan el movimiento? Pues en la función que se encarga de los eventos de teclado:

```
...
case Qt::Key_Q:static_cast<_hand *>(Vec_objects[2])->Translation+=0.05;break;
case Qt::Key_W:static_cast<_hand *>(Vec_objects[2])->Translation-=0.05;break;
case Qt::Key_A:static_cast<_arm1 *>(Vec_objects[3])->Angle+=2;break;
case Qt::Key_S:static_cast<_arm1 *>(Vec_objects[3])->Angle-=2;break;
case Qt::Key_Z:static_cast<_arm2 *>(Vec_objects[4])->Angle+=1;break;
case Qt::Key_X:static_cast<_arm2 *>(Vec_objects[4])->Angle-=1;break;
case Qt::Key_C:static_cast<_base *>(Vec_objects[5])->Angle_arm+=1;break;
case Qt::Key_V:static_cast<_base *>(Vec_objects[5])->Angle_arm-=1;break;
case Qt::Key_B:static_cast<_base *>(Vec_objects[5])->Angle_base+=2;break;
case Qt::Key_N:static_cast<_base *>(Vec_objects[5])->Angle_base-=2;break;
...
```

Aunque se dispone de los punteros a los objetos, hay que hacer un ajuste (*casting*) para que se adapte a la clase apropiada.

Es importante hacer notar que la implementación que se muestra en este ejemplo está simulando la funcionalidad de la interfaz obsoleta de OpenGL y que no está optimizada para hacer un uso correcto de la arquitectura y funcionamiento de la GPU. La intención es que sirva para entender las transformaciones y su aplicación así como la idea de jerarquía. El alumno deberá investigar formas más óptimas para su uso con la GPU

14

Iluminación

En este ejemplo vamos a ver cómo se simula la iluminación de los objetos dándoles realismo. Tenemos tres aproximaciones al realismo que van de menos a más. Son el sombreado plano, el sombreado de Gouraud, y el sombreado de Phong. En este tema vamos a ver cómo se calculan el sombreado plano y el de Gouraud, dejando el de Phong para el siguiente. Esta división se debe a que veremos en primer lugar cómo se pueden hacer los cálculos del sombreado plano y de Gouraud en el vertex shader, pero para realizar el sombreado de Phong nos obligará a hacerlo en el fragment shader.

Para poder simular la iluminación nos hace falta un modelo de reflexión. El que vamos a usar es un modelo sencillo que tiene tres componentes: ambiental, difusa y especular.

Empecemos por la componente más sencilla de entender, y en muchos casos la que mayor importancia tiene, la componente difusa. La idea básica para entender su comportamiento consiste en entender que la orientación del objeto respecto a la fuente de luz modifica la cantidad de luz reflejada: cuanto más perpendicular esté a los rayos de luz más reflejará. La figura 14.1 muestra la idea.

Dado que las superficies con las que estamos trabajando son triángulos, esto implica que cuanto mas perpendicular sea el plano en el que se encuentra inscrito el triángulo mayor será la cantidad de luz reflejada. ¿De qué manera queda definido el plano? ¡Mediante la normal! Por tanto, lo primero que tenemos que hacer es ver cómo se obtienen las normales para nuestros modelos. Podemos ver que se pueden calcular dos tipos de normales: las normales de los triángulos y las normales de los vértices. Veamos cómo se calcula cada uno de los dos tipos.

El cálculo de la normal de un triángulo, es muy sencillo. La normal del plano que incluye al triángulo se obtiene de la siguiente manera. Dados los puntos P_0, P_1 y P_2, podemos calcular los vectores $\vec{A} = P_1 - P_0$ y $\vec{B} = P_2 - P_0$. Si aplicamos el producto vectorial $\vec{A} \times \vec{B}$ obtenemos

Domingo Martín Perandrés©

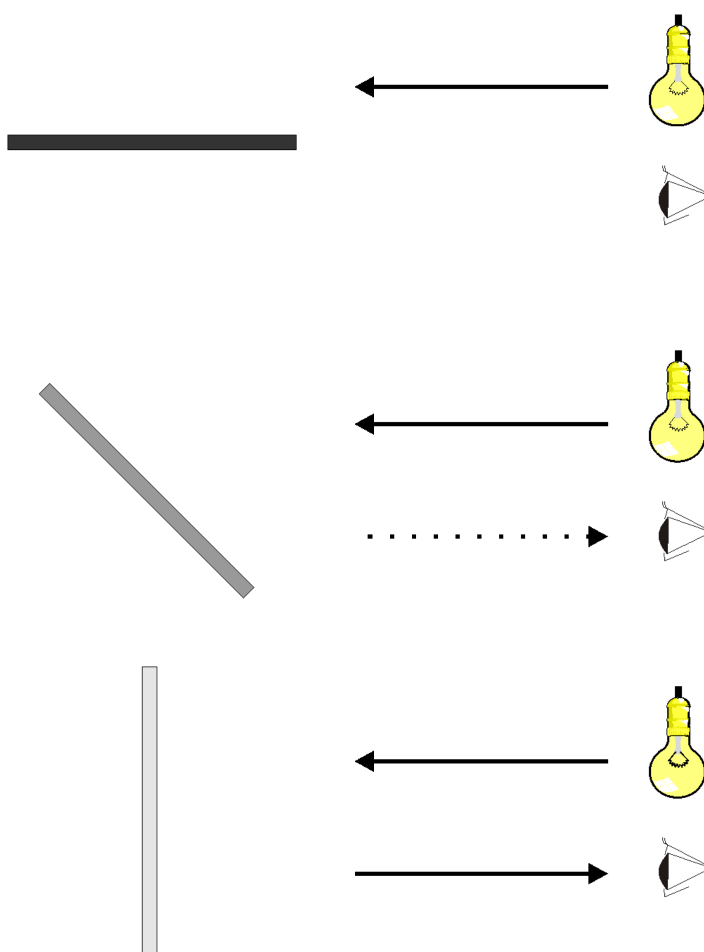
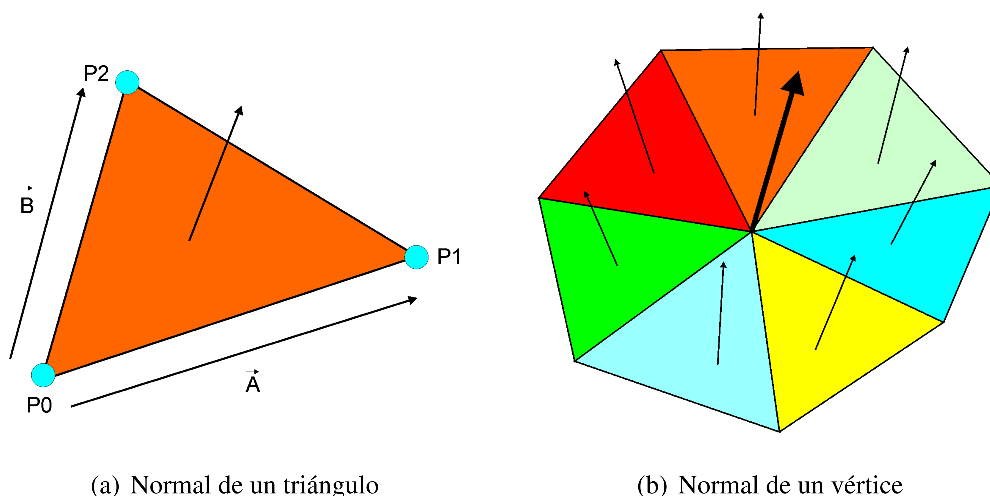


Figura 14.1: Reflexión difusa: cómo afecta la orientación

el vector normal, $\vec{N}$, cuyo sentido viene dado por la regla de la mano derecha (en general, se entiende que la normal apunta hacia el lado exterior del polígono) (ver figura 14.2(a)).

La normal de la cara se usa no sólo para cálculos de iluminación sino que también sirve para determinar la orientación de la cara, si la misma mira hacia adentro del objeto o hacia afuera. Es importante que mire hacia afuera para que la cara pueda ser visualizada, en el caso de que se indique, mediante la instrucción glPolygonMode, que sólo se muestren las caras orientadas hacia adelante, mediante el valor GL_FRONT. Una solución consiste en visualizar las caras que miran hacia adelante y hacia atrás, mediante el valor GL_FRONT_AND_BACK, pero salvo que se quiera hacer porque se planee el introducirse en el interior del objeto, dicha opción evitará que se pueda mejorar el rendimiento, pues si así se indica, OpenGL puede eliminar de los cálculos todas aquellas caras que no son visibles en relación al observador. Para determinar si una cara es visible o no desde la posición del observador, basta con hacer el producto escalar entre la normal y el vector que se forma entre la posición del observador y una posición de la cara.



(a) Normal de un triángulo

(b) Normal de un vértice

Figura 14.2: Cálculo de las normales

Para el cálculo de la normal de un vértice en un modelos de triángulos, se parte de las normales de los triángulos que confluyen en dicho vértice y se calcula el valor medio de las normales (ver figura 14.2(b)). Esto es:

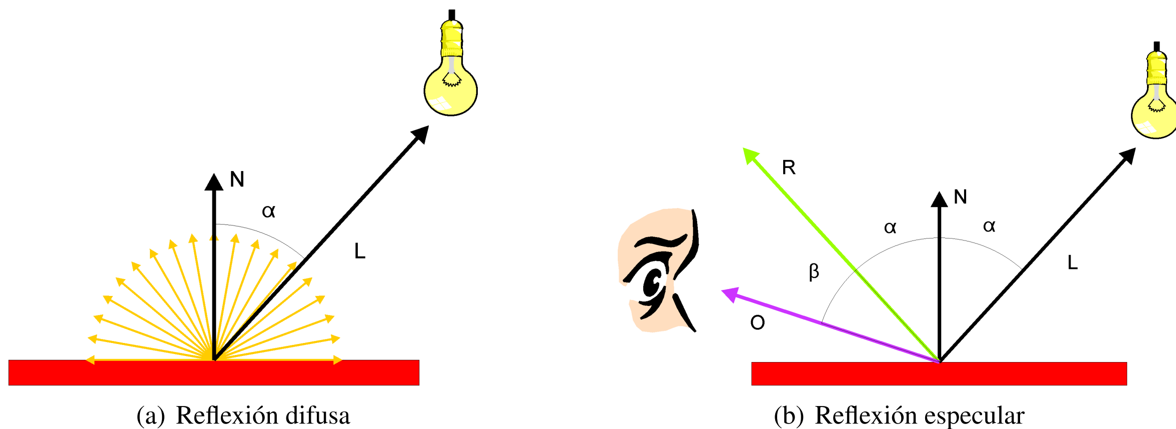
$$\vec{N} = \frac{\sum_{i=1}^{n} \vec{N_i}}{n}$$

Obsérvese que la normal es una aproximación, pero lo mismo se puede decir de un modelo poligonal que representa a una superficie curva. Otro detalle importante que quedará claro en breve es que las normales deben estar normalizadas.

Existen objetos en los que el cálculo de las normales es más sencillo o directo. Por ejemplo, en el caso de la esfera las normales en los vértices se obtienen de los propios vértices: $\vec{N} = \vec{P} - \vec{C}$, siendo $\vec{C}$ el centro de la esfera.

Para guardar las normales, y dado que estamos usando la función glDrawArrays para dibujar, tenemos que crear 2 VBO más, uno que guardara una normal por cada vértice, siendo la misma normal para los tres vértices de un triángulo, y otro que guardará una normal por cada vértice, pero teniendo cada vértice su propia normal. Los datos se secuenciarán de la misma manera que se ha hecho con los vértices y los colores. Los ejes no tienen normales con lo cual no se meten datos.

Dado que estamos usando la primitiva glDrawArrays, la misma permite que vértices que son compartidos por varias caras puedan tener varias normales, ya que hemos metido los vértices de cada cara, aunque ocurran repeticiones. Esta redundancia se puede evitar si usamos un vector de vértices y un vector de índices a dichos vértices, y se usa la función glDrawElements, pero en tal caso habrá compartición de valores a costa de no producir repeticiones.



(a) Reflexión difusa (b) Reflexión especular

Figura 14.3: Tipos de reflexión

Una vez que hemos visto cómo se calculan las normales sigamos viendo cómo se utilizan para los cálculos de iluminación.

Empezamos con el cálculo de la reflexión **difusa**, la cual se muestra en la figura 14.3(a). Dada la normal que define la orientación del triángulo, la reflexión será 0 cuando sea perpendicular al vector que indica la dirección de la luz, y será máxima cuando los dos vectores sean paralelos. O sea, cuando los vectores son perpendiculares el valor debe ser 0 y cuando son paralelos debe ser máximo. O si tenemos en cuenta el ángulo que se forma entre ambos vectores, cuando el ángulo es 0 el valor debe ser máximo y cuando el ángulo es 90 grados, el valor debe ser 0. La función coseno cumple con la propiedad, si entendemos que el valor que devuelve servirá de modulador. Así, $\cos(0°) = 1$ y $\cos(90°) = 0$. Para calcular el coseno vamos a usar el producto escalar: $\vec{N} \cdot \vec{L} = |\vec{N}| \cdot |\vec{L}| \cdot \cos(\alpha)$. Si tenemos los vectores normalizados, entonces $|\vec{N}| = 1$ y $|\vec{L}| = 1$ y por tanto, la ecuación queda así: $\vec{N} \cdot \vec{L} = \cos(\alpha)$. Es importante hacer notar que para esta componente la posición del observador no afecta el resultado. Esto es, cuando llega un rayo de luz el objeto refleja en todas las direcciones. La mayor parte de los objetos que vemos suelen tener una componente mayoritariamente difusa.

También es importante tener en cuenta que el producto escalar puede dar valores negativos, en cuyo caso implica que la parte delantera de la cara está orientada hacia atrás. Este método también se puede usar para evitar cálculos de caras que no son visibles, es el llamado *culling* que puede ser habilitado en OpenGL.

Existen objetos que reflejan la luz de otra manera, que vamos a llamar reflexión **especular**. Un ejemplo son los espejos, casi un reflector especular ideal. En este tipo de objetos, un rayo de entrada es reflejado produciendo un rayo de salida con el mismo ángulo que el rayo de entrada. La idea se muestra en la figura 14.3(b). En este caso sí que es importante la posición del observador pues dependiendo de la misma es posible que vea el rayo reflejado o no. En un reflecto especular ideal, el ángulo entre el rayo reflejado y la dirección del observador tendría que ser 0. Lo que ocurre normalmente es que los reflectores especulares no son ideales y se produce una cierta dispersión alrededor del rayo reflejado. Para modelar

Domingo Martín Perandrés©

este comportamiento de nuevo recurrimos al coseno, pero esta vez, para el ángulo β que se forma entre $\vec{R}$ y $\vec{0}$: cuando β es 0° se obtiene la máxima reflexión y cuando es 90° se obtiene 0. Dado que podemos encontrar objetos más o menos especulares, es necesario añadir un modificador de la función coseno: elevar el coseno a una potencia, con valores entre 0 e infinito. En las implementaciones, se usan valores finitos. Por tanto, la reflexión especular nos quedaría de la siguiente manera $\vec{R} \cdot \vec{O} = \cos^n(\beta)$, estando los vectores $\vec{R}$ y $\vec{O}$ normalizados y siendo n el exponente.

La última componente que nos falta es la reflexión **ambiental**. Con esta componente se pretende modelar una especie de flujo de luz constante que viene de todas direcciones. Este flujo no is imaginado sino que tiene un fundamento físico. Imaginemos una habitación pequeña sin ninguna iluminación salvo la luz que entra por un pequeño agujero. Un observador que esté en el interior podrá ver las distintas partes de la habitación. El motivo es que la luz se empieza a reflejar en las paredes una y otra vez haciendo visible lo que en principio no estaba iluminado directamente. Por tanto, es una componente que forma a partir de las numerosas reflexiones de los rayos de luz en los distintos objetos. Es fácil ver que conforme los rayos se reflejan irán adquiriendo la tonalidad del material del objeto que produce la reflexión. En un modelo de iluminación global, todos estos reflejos se tienen en cuenta. En el modelo sencillo de iluminación que estamos explicando, un modelo local, esta componente se simplifica mediante el uso de un valor constante de baja intensidad. Además, evita el efecto de que las caras que no están iluminadas directamente se vean de color negro, lo cual resulta poco natural.

Recapitulemos. Nuestro modelo de reflexión de la luz es el siguiente: $I_r = Ambiental + Difusa + Especular$. Esto es, la intensidad de luz reflejada por un objeto es la suma de las componentes ambiental, difusa y especular. Hemos visto que el valor de las reflexiones difusa y especular depende de la orientación del objeto con respecto a la luz en el caso de la reflexión difusa, y del observador con respecto al rayo reflejado en la reflexión especular. Ahora tenemos que incluir la intensidad de la fuente de luz y el material: si no hay luz no podemos iluminar nada; el material del objeto modula la reflexión, un color oscuro refleja menos luz que un color claro.

Si la intensidad de la fuente de luz es I_l, si la capacidad del material para reflejar la componente difusa es K_d, la capacidad del material para reflejar la componente especular es K_e, y la reflexión ambiental se modela con K_a, la fórmula nos queda: $I_r = I_l \cdot K_a + I_l \cdot K_d \cdot cos(\alpha) + I_l \cdot K_e \cdot \cos^n(\beta)$

Dado que tanto la luz como los materiales se definen como colores RGB, la formula anterior se debe extender a las tres componentes.

Un detalle importante a tener en cuenta en la ecuación que hemos explicado es la posibilidad de obtener valores de intensidad reflejada mayor que 1, cuando en OpenGL, la máxima intensidad es 1. Veamos un ejemplo. Si suponemos que el valor de la luz es $I_l = 1$, luz blanca, la componente ambiental es $K_a = 0.25$, la componente difusa es $K_d = 0.8$, y la componente especular es $K_d = 0.5$, un gris medio, podemos ver que como mínimo la intensidad reflejada

será $I_r = 1 \cdot 0.25 \rightarrow 0.25$, pero el valor máximo se dará cuando α y β sean 0. En tal caso el valor será: $I_r = 1 \cdot 0.25 + 1 \cdot 0.8 \cdot 1 + 1 \cdot 0.5 \cdot 1 \rightarrow 1.55$. Este valor tiene que ser recortado a 1. Lo que veremos es que muchas partes del objeto se ven con una intensidad máxima. Por tanto, es importante modular correctamente los valores de los materiales, ya que será bastante normal usar fuentes de luz de color blanco $I_l(1,1,1)$.

Nos centramos ahora en cómo se aplica el modelo de iluminación a un objeto. Tal y como hemos visto, las componentes difusa y especular dependen de la normal del objeto y de la posición de la fuente de luz. Además, la componente especular depende de la posición del observador. Por tanto, es necesario pasar dichos parámetros a los shaders, en concreto, al vertex shader.

Todos los cálculos del modelo de iluminación se pueden hacer en coordenadas del modelo, en coordenadas de mundo o en coordenadas de cámara, implicando, en cada caso, los ajustes pertinentes. En nuestro caso, los vamos a hacer en coordenadas de mundo. Obsérvese que no hay transformación de modelado, o dicho de otro modo, la matriz de transformación es la identidad.

Para el cálculo de la posición de la cámara usamos la inversa de la transformación de la cámara o transformación de vista que ya hemos visto (ver tema 6).

La transformación de las normales es diferente de la transformación de los vértices. Si la transformación para los vértices es M, la de las normales es $(M^{-1})^T$. Se deja para el lector el que compruebe por qué es así.

Veamos el código necesario para poder implementar la iluminación. El objeto que vamos a usar es la esfera, pues permite apreciar muy fácilmente las variaciones de color, sobre todo en los modos de sombreado.

En primer lugar tenemos que incluir nuevos vectores:

```
vector<_vertex3f> Vertices;
vector<_vertex3f> Vertices_colors;
vector<_vertex3f> Vertices_normals;

vector<_vertex3i> Triangles;
vector<_vertex3f> Triangles_colors;
vector<_vertex3f> Triangles_normals;

vector<_vertex3f> Vertices_drawarray;
vector<_vertex3f> Vertices_colors_drawarray;
vector<_vertex3f> Vertices_normals_drawarray;
vector<_vertex3f> Vertices_triangles_normals_drawarray;
```

Al construir la esfera deberemos rellenar todos los datos tal y como hecho anteriormente, sólo que ahora, además, hay que calcular las normales para los triángulos y los vértices y meterlos en los correspondientes vectores.

Además tendremos que crear y rellenar con los datos correspondientes los VBOs de las normales de los triángulos y vértices. El código es el mismo que hemos usado anteriormente pero cambiando los datos a usar.

Para controlar el color hasta ahora teníamos la posibilidad de utilizar un color fijo dependiendo del tipo visualización (puntos, líneas y relleno), y en tal caso todos los elementos se dibujaban con el color especificado, y teníamos la posibilidad de usar los colores mandados en los vectores y realizar una interpolación entre los mismos: si usábamos los colores definidos para cada triángulo el color no cambiaba, pero en el caso de usar los colores definidos para cada vértice el color sí cambiaba. Ahora debemos añadir las componentes de los materiales que no son otra cosa que colores. Tenemos que definir las cuatro componentes que hemos explicado en el modelo de iluminación: el material para las componentes ambiental, la difusa y especular, y el exponente para modular la componente especular. Cada objeto tendrá su material. Para integrarlo todo, crearemos una estructura que contenga el color único cuando no hay iluminación y las componentes del material:

```
class _material {
public:
  _vertex3f Constant;
  _vertex3f Ambient;
  _vertex3f Diffuse;
  _vertex3f Specular;
   float Specular_exponent;
};
```

La función que se encarga de dibujar es la siguiente:

```
void _gl_widget::draw_objects()
{
  QMatrix4x4 Modelview;
  QMatrix4x4 Projection;
  QMatrix4x4 Modelview_normals;
  QMatrix4x4 Camera;

  float Aspect=(float)Window_height/(float)Window_width;

  if (Projection_type==PERSPECTIVE_PROJECTION){
    Projection.frustum(X_MIN,X_MAX,Y_MIN*Aspect,Y_MAX*Aspect,FRONT_PLANE_PERSPECTIVE,↩
        BACK_PLANE_PERSPECTIVE);
  }
  else{
    Projection.ortho(X_MIN*Scale_factor,X_MAX*Scale_factor,Y_MIN*Aspect*Scale_factor,↩
        Y_MAX*Aspect*Scale_factor,FRONT_PLANE_PARALLEL,BACK_PLANE_PARALLEL);
  }

  // Modelview
  Modelview.translate(0,0,-Distance);
  Modelview.rotate(Angle_x,1,0,0);
  Modelview.rotate(Angle_y,0,1,0);

  // transformacion para las normales
  Modelview_normals=Modelview;
  Modelview_normals.inverted();
  Modelview_normals.transposed();

  // calculo de la posicion de la camara
```

```
  Camera.rotate(-Angle_y,0,1,0);
  Camera.rotate(-Angle_x,1,0,0);
  QVector3D Camera_position=(QVector3D)(Camera*QVector4D(0,0,Distance,1));

  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
  glBindVertexArray(VAO1);
  glUseProgram(Program1);

  glUniformMatrix4fv(10,1,GL_FALSE,Modelview.data());
  glUniformMatrix4fv(11,1,GL_FALSE,Projection.data());
  glUniformMatrix4fv(12,1,GL_FALSE,Modelview_normals.data());

  // axis
  glUniform1i(20,(int)MODE_LINE);
  glUniform1i(21,(int)MODE_COLORS_VARIABLE);
  glDrawArrays(GL_LINES,0,Axis_vertices_size);

  if (Draw_points){
    glPolygonMode(GL_FRONT_AND_BACK,GL_POINT);
    glPointSize(3);
    glUniform1i(20,(int)MODE_POINT);
    glUniform1i(21,(int)MODE_COLORS_FIXED);
    glUniform3fv(25,1,(GLfloat*) &COLORS[COLOR_POINT]);

    glDrawArrays(GL_TRIANGLES,Axis_vertices_size,Sphere_vertices_size);
  }

  if (Draw_lines){
  glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
    glUniform1i(20,(int)MODE_LINE);
    glUniform1i(21,(int)MODE_COLORS_FIXED);
    glUniform3fv(26,1,(GLfloat*) &COLORS[COLOR_LINE]);

    glDrawArrays(GL_TRIANGLES,Axis_vertices_size,Sphere_vertices_size);
  }

  if (Draw_fill){
    glPolygonMode(GL_FRONT_AND_BACK,GL_FILL);
    glUniform1i(20,(int)MODE_FILL);
    glUniform1i(21,Mode_color);
    glUniform3fv(27,1,(GLfloat*) &COLORS[COLOR_FILL]);

    if (Illumination_active){
      glUniform1i(22,(int)Mode_shading);
      glUniform1i(23,(int)Illumination_active);
      glUniform4fv(30,1,(GLfloat*) &Light.Position);
      glUniform3fv(31,1,(GLfloat*) &Light.Color);
      glUniform3fv(32,1,(GLfloat*) &Camera_position);
      glUniform3fv(40,1,(GLfloat*) &Sphere_material.Ambient);
      glUniform3fv(41,1,(GLfloat*) &Sphere_material.Diffuse);
      glUniform3fv(42,1,(GLfloat*) &Sphere_material.Specular);
      glUniform1f(43,Sphere_material.Specular_exponent);
      glUniform3fv(44,1,(GLfloat*) &Ambient_coeff);
    }
    else{
      glUniform1i(23,(int)Illumination_active);
    }
    glDrawArrays(GL_TRIANGLES,Axis_vertices_size,Sphere_vertices_size);
  }

  glUseProgram(0);
  glBindVertexArray(0);
}
```

Es importante resaltar lo siguiente en el código. Lo primero es que hemos juntado las transformaciones de modelado y de vista en la modelview, que es la que tenemos en la

versión antigua de OpenGL. También se muestra el código que calcula la tranformación que se debe aplicar a las normales, siguiendo lo explicado. Para calcular la posición de la cámara se puede ver que calculamos la inversa de la transformación de vista. Por último, en el caso de que esté activa la iluminación habrá que mandar los valores de la posición de la luz, el color de la misma, la posición de la cámara y el material.

Para poder ver el efecto de la iluminación tenemos que tener activo el modo en el que los colores son variables. Con la iluminación activa, podemos elegir entre el sombreado plano y el de Gouraud. El código para cambiar las variables de control es el siguiente:

```
case Qt::Key_P:Draw_points=!Draw_points;break;
case Qt::Key_L:Draw_lines=!Draw_lines;break;
case Qt::Key_F:Draw_fill=!Draw_fill;break;

case Qt::Key_1:Mode_shading=MODE_SHADING_FLAT;break;
case Qt::Key_2:Mode_shading=MODE_SHADING_GOURAUD;break;

case Qt::Key_I:Illumination_active=!Illumination_active;break;

case Qt::Key_C:
  if (Mode_color==MODE_COLORS_FIXED) Mode_color=MODE_COLORS_VARIABLE;
  else Mode_color=MODE_COLORS_FIXED;
  break;
```

El código del vertex shader es el siguiente. Para el sombreado plano y el sombreado de Gouraud es el mismo salvo que en un caso se usan las normales de los triángulos y en el otro las normales de los vértices. Por tanto, el código se podría compactar, pero se ha dejado en la forma actual para facilitar su entendimiento. Como se puede observar, simplemente se realizan los cálculos expuestos al explicar cada una de las componentes:

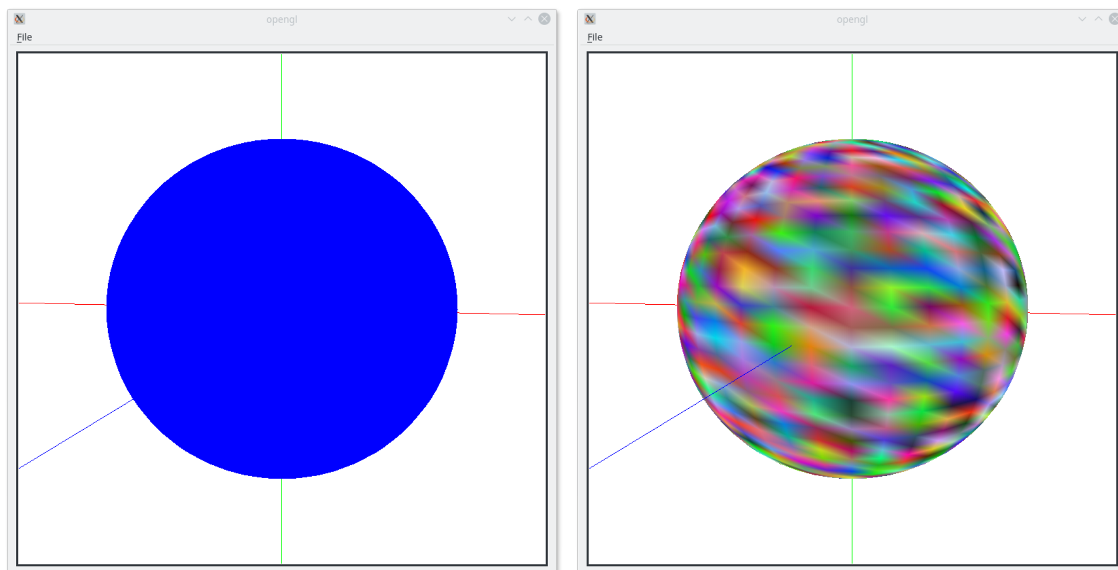
```
#version 450 core
layout (location=0) in vec3 vertex;
layout (location=1) in vec3 color;
layout (location=2) in vec3 triangle_normal;
layout (location=3) in vec3 vertex_normal;
// layout (location=4) in vec2 texture_coordinates;

layout (location=10) uniform mat4 modelview;
layout (location=11) uniform mat4 projection;
layout (location=12) uniform mat4 modelview_normals;

layout (location=20) uniform int mode_rendering;
layout (location=21) uniform int mode_color;
layout (location=22) uniform int mode_shading;
layout (location=23) uniform int illumination_active;
// layout (location=23) uniform int texture_active;

layout (location=25) uniform vec3 color_point;
layout (location=26) uniform vec3 color_line;
layout (location=27) uniform vec3 color_fill;

layout (location=30) uniform vec4 light_position;
layout (location=31) uniform vec3 light_color;
layout (location=32) uniform vec3 camera_position;

layout (location=40) uniform vec3 material_ambient;
layout (location=41) uniform vec3 material_diffuse;
```

```
layout (location=42) uniform vec3 material_specular;
layout (location=43) uniform float material_specular_exponent;
layout (location=44) uniform vec3 ambient_coeff;

out vec3 color_out;

void main(void)
{
  if (mode_color==0){ // fixed
    switch (mode_rendering){
      case 0: color_out=color_point;break;
      case 1: color_out=color_line;break;
      case 2: color_out=color_fill;break;
    }
  }
  else{ // using vectors
    if (mode_rendering==2){ // fill
      if (illumination_active==1){
        if (mode_shading==0){ // flat
          // ambient
          vec3 ambient_component=ambient_coeff*light_color*material_ambient;

          // diffuse
          vec3 light_direction;
          if (light_position.w==0){
          light_direction=vec3(light_position);
          }
          else{
          light_direction=vec3(light_position)-vertex;
          }
          light_direction=normalize(light_direction);
          float diffuse_intensity = max(dot(light_direction,triangle_normal), 0.0);
          vec3 diffuse_component = diffuse_intensity*material_diffuse;

          // specular
          vec3 view_direction=camera_position-vertex;
          view_direction=normalize(view_direction);
          vec3 reflected_direction = reflect(-light_direction,triangle_normal);
          float specular_intensity = pow(max(dot(view_direction,reflected_direction), ↩
              0.0), material_specular_exponent);
          vec3 specular_component = specular_intensity*material_specular;

          color_out=ambient_component+diffuse_component+specular_component;
        }
        else{ // Gouraud
          // ambient
          vec3 ambient_component=ambient_coeff*light_color*material_ambient;

          // diffuse
          vec3 light_direction;
          if (light_position.w==0){
          light_direction=vec3(light_position);
          }
          else{
          light_direction=vec3(light_position)-vertex;
          }
          light_direction=normalize(light_direction);
          float diffuse_intensity = max(dot(light_direction,vertex_normal), 0.0);
          vec3 diffuse_component = diffuse_intensity*material_diffuse;

          // specular
          vec3 view_direction=camera_position-vertex;
          view_direction=normalize(view_direction);
          vec3 reflected_direction = reflect(-light_direction,vertex_normal);
          float specular_intensity = pow(max(dot(view_direction,reflected_direction), ↩
              0.0), material_specular_exponent);
          vec3 specular_component = specular_intensity*material_specular;
```

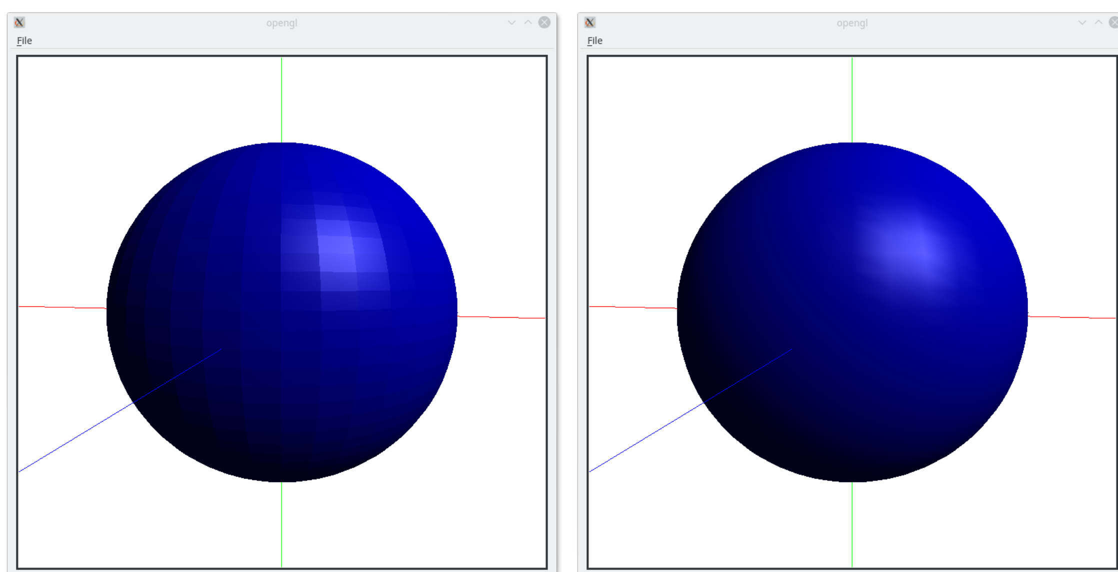
Domingo Martín Perandrés©

```
          color_out=ambient_component+diffuse_component+specular_component;
        }
      }
      else{ // no illumination
        color_out=color;
      }
    }
    else{ // no fill
      color_out=color;
    }
  }
  gl_Position=projection*modelview*vec4(vertex,1);
}
```

Domingo Martín Perandrés©



(a) Color constante (b) Color variable sin iluminación



(c) Color variable, iluminación y sombreado plano (d) Color variable, iluminación y sombreado de Gouraud

Figura 14.4: Resultados del ejemplo 14

15

Iluminación: Phong

En el ejemplo anterior hemos implementado el sombreado plano y el de Gouraud, pero no el de Phong. Esto se debe a que los cálculos que se necesitan se pueden hacer fácilmente en un fragment shader pero es muy complicado en el vertex shader o implicaría un número de triángulos muy grande. La idea para entender la diferencia entre Gouraud y Phong consiste en que Gouraud es una versión simplificada de Phong: con Gouraud se obtienen los valores de reflexión en los vértices, se calculan los colores correspondientes y los mismos se interpolan para obtener el resto de posiciones (fragmentos/píxeles), interpolación realizada por el fragment shader. Con Phong se quiere obtener un resultado más exacto haciendo que se realicen los cálculos de reflexión en todos los puntos del objeto. El problema está en que en el vertex shader tenemos una definición continua de los objetos, líneas y triángulos, con lo cual el número de posiciones es infinito. Pero los objetos continuos son discretizados en fragmentos, que es lo que llega a los fragment shaders, y aunque puedan ser muchos, son finitos, con lo cual podemos hacer los cálculos, consistentes en obtener la reflexión para cada fragmento. Por eso pasamos los cálculos al fragment shader.

Hay que observar que si usamos los fragmentos implica que debemos obtener el valor de la posición y de la normal para cada fragmento pues sólo tenemos los valores en los vértices. De nuevo el proceso de interpolación es el encargado de hacer posible ese cálculo. Para ello debemos crear dos variables de salida del vertex shader, la posición y la normal, y ponerlas como variables de entrada en el fragment shader. OpenGL se encargará de hacer la interpolación entre los valores dependiendo del tipo de primitiva que se esté dibujando. De esta forma obtenemos la posición y la normal para cada fragmento y podemos realizar el cálculo de iluminación, que es el mismo que hemos visto en el ejemplo anterior.

El código del vertex shader es el siguiente:

```
#version 450 core
layout (location=0) in vec3 vertex;
layout (location=1) in vec3 color;
```

Domingo Martín Perandrés©

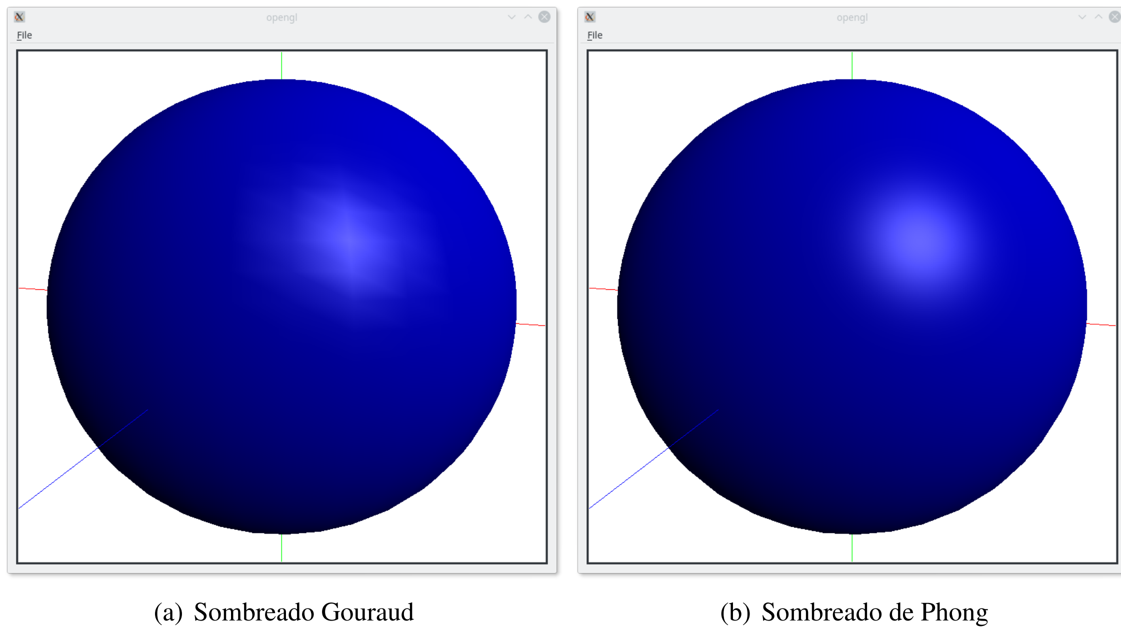
```
layout (location=2) in vec3 triangle_normal;
layout (location=3) in vec3 vertex_normal;

layout (location=10) uniform mat4 modelview;
layout (location=11) uniform mat4 projection;
layout (location=12) uniform mat4 modelview_normals;

layout (location=20) uniform int mode_rendering;
layout (location=21) uniform int mode_color;
layout (location=22) uniform int mode_shading;
layout (location=23) uniform int illumination_active;

layout (location=25) uniform vec3 color_point;
layout (location=26) uniform vec3 color_line;
layout (location=27) uniform vec3 color_fill;

layout (location=30) uniform vec4 light_position;
layout (location=31) uniform vec3 light_color;
layout (location=32) uniform vec3 camera_position;

layout (location=40) uniform vec3 material_ambient;
layout (location=41) uniform vec3 material_diffuse;
layout (location=42) uniform vec3 material_specular;
layout (location=43) uniform float material_specular_exponent;
layout (location=44) uniform vec3 ambient_coeff;

out vec3 color_out;
out vec3 vertex_out;
out vec3 normal_out;

void main(void)
{
  if (mode_color==1){ // color variable
    if (mode_rendering==2){ // fill
      if (illumination_active==1){
        switch(mode_shading){
        case 0: // flat
          {
          // ambient
          vec3 ambient_component=ambient_coeff*light_color*material_ambient;

          // diffuse
          vec3 light_direction;
          if (light_position.w==0){
          light_direction=vec3(light_position);
          }
          else{
          light_direction=vec3(light_position)-vertex;
          }
          light_direction=normalize(light_direction);
          float diffuse_intensity = max(dot(light_direction,triangle_normal), 0.0);
          vec3 diffuse_component = diffuse_intensity*material_diffuse;

          // specular
          vec3 view_direction=camera_position-vertex;
          view_direction=normalize(view_direction);
          vec3 reflected_direction = reflect(-light_direction,triangle_normal);
          float specular_intensity = pow(max(dot(view_direction,reflected_direction), ↩
              0.0), material_specular_exponent);
          vec3 specular_component = specular_intensity*material_specular;

          color_out=ambient_component+diffuse_component+specular_component;
          }
          break;
        case 1: // Gouraud
          {
          // ambient
```

```
          vec3 ambient_component=ambient_coeff*light_color*material_ambient;

          // diffuse
          vec3 light_direction;
          if (light_position.w==0){
          light_direction=vec3(light_position);
          }
          else{
          light_direction=vec3(light_position)-vertex;
          }
          light_direction=normalize(light_direction);
          float diffuse_intensity = max(dot(light_direction,vertex_normal), 0.0);
          vec3 diffuse_component = diffuse_intensity*material_diffuse;

          // specular
          vec3 view_direction=camera_position-vertex;
          view_direction=normalize(view_direction);
          vec3 reflected_direction = reflect(-light_direction,vertex_normal);
          float specular_intensity = pow(max(dot(view_direction,reflected_direction), ↩
              0.0), material_specular_exponent);
          vec3 specular_component = specular_intensity*material_specular;

          color_out=ambient_component+diffuse_component+specular_component;
          }
          break;
        case 2: // Phong
          vertex_out=vertex;
          normal_out=vertex_normal;
          break;
        }
      }
      else{ // no illumination
        color_out=color;
      }
    }
    else{ // no fill
      color_out=color;
    }
  }
  else{ // color fixed
    switch (mode_rendering){
      case 0: color_out=color_point;break;
      case 1: color_out=color_line;break;
      case 2: color_out=color_fill;break;
    }
  }
  gl_Position=projection*modelview*vec4(vertex,1);
}
```

Mantenemos un código muy parecido al del ejemplo anterior, y sólo cambiamos el que en caso de querer un sombreado de Phong pasamos los valores de las posiciones de los vértices y de las normales de los mismos al fragment shader. Esto hará que ambos datos sean interpolados obteniendo un valor para cada fragmento. No transformamos ni la posición ni la normal porque los cálculos se están realizando en coordenadas de mundo y la transformación de modelado es la identidad. Hay que tener en cuenta que una cosa son los cálculos de iluminación, realizados en coordenadas de mundo, y otra cosa es que los objetos tienen que convertirse en fragmentos, que finalmente acaban como píxeles, en coordenadas de dispositivo. Esto es, con gl_Position OpenGL va a calcular los fragmentos, mientras que con las variables vertex_out y normal_out va a calcular la reflexión de los puntos del objeto en coordenadas de mundo que se corresponden con los fragmentos.

El código del fragment shader es el siguiente:

```
#version 450 core
layout (location=20) uniform int mode_rendering;
layout (location=21) uniform int mode_color;
layout (location=22) uniform int mode_shading;
layout (location=23) uniform int illumination_active;

layout (location=30) uniform vec4 light_position;
layout (location=31) uniform vec3 light_color;
layout (location=32) uniform vec3 camera_position;

layout (location=40) uniform vec3 material_ambient;
layout (location=41) uniform vec3 material_diffuse;
layout (location=42) uniform vec3 material_specular;
layout (location=43) uniform float material_specular_exponent;
layout (location=44) uniform vec3 ambient_coeff;

in vec3 color_out;
in vec3 vertex_out;
in vec3 normal_out;

out vec4 frag_color;

void main(void)
{
  if (mode_color==1){ // fixed
    if (mode_rendering==2){ // fill
      if (illumination_active==1){
        if (mode_shading==2){ // Phong
          // ambient
          vec3 ambient_component=ambient_coeff*light_color*material_ambient;

          // diffuse
          vec3 light_direction;
          if (light_position.w==0){
          light_direction=vec3(light_position);
          }
          else{
          light_direction=vec3(light_position)-vertex_out;
          }
          light_direction=normalize(light_direction);
          vec3 normal_normalized=normalize(normal_out);
          float diffuse_intensity = max(dot(light_direction,normal_normalized), 0.0);
          vec3 diffuse_component = diffuse_intensity*material_diffuse;

          // specular
          vec3 view_direction=camera_position-vertex_out;
          view_direction=normalize(view_direction);
          vec3 reflected_direction = reflect(-light_direction,normal_normalized);
          float specular_intensity = pow(max(dot(view_direction,reflected_direction), ↩
              0.0), material_specular_exponent);
          vec3 specular_component = specular_intensity*material_specular;

          frag_color=vec4(ambient_component+diffuse_component+specular_component,1);
        }
        else{ // no Phong
          frag_color=vec4(color_out,1);
        }
      }
      else{ // no illumination
        frag_color=vec4(color_out,1);
      }
    }
    else{ // no fill
      frag_color=vec4(color_out,1);
```



(a) Sombreado Gouraud (b) Sombreado de Phong

Figura 15.1: Resultados del ejemplo 15

```
    }
  }
  else{
    frag_color=vec4(color_out,1);
  }
}
```

El código es muy simple. Solo para el caso del sombreado de Phong realiza los cálculos de iluminación con los valores propios de posición y normal para cada fragmento. En el resto de los casos usa el color que se ha calculado en el vertex shader.

De nuevo, es importante hacer notar que no se pretende que el código esté optimizado sino que sea lo más descriptivo y fácil de entender posible.

En la figura 15.1 se pueden comparar los resultados de ambos sombreados. Es fácil ver que la calidad del sombreado de Phong (figura 15.1(b)) es mucho mejor que el sombreado de Gouraud (figura 15.1(a)). Si el número de triángulos es muy grande, llegando a que como mucho ocupen un pixel, el resultado será similar.

Domingo Martín Perandrés©

16

Texturas

En este ejemplo vamos a hacer uso de una de las funcionalidades que permite obtener mayor realismo: las texturas.

Imaginemos que queremos producir una imagen 2D realista: un buen pintor conseguirá un buen resultado con mucho esfuerzo, pero nada será mejor que una fotografía. Esta idea se puede extender a la visualización de modelos realistas: sólo con vértices y colores es muy difícil que consigamos un alto grado de realismo, y si se consigue sólo es posible con un gran número de vértices.

Pongamos un ejemplo. Imaginemos que queremos modelar un tablón de madera. La forma del mismo es muy fácil de modelar con un paralelepípedo, un cubo estirado. Con 8 vértices y 8 colores, poco se puede conseguir para simular la madera. La solución consistiría en incrementar el número de vértices hasta que consiguiéramos el nivel de detalle deseado.

La solución que se propone con las texturas es la siguiente: hacemos una foto de cada lado del tablón y cada foto es pegada a un lado del modelo. La geometría en sencilla pero la visualización es realista. No vamos a entrar en cómo se puede pegar la imagen en el modelo, pero ayuda el pensar que la fotografía se ha imprimido en una tela que es deformable, de tal manera que la ajustamos al objeto.

Los pasos para aplicar una textura se muestran en la figura 16.1. El primer paso consiste en pasar una imagen matricial a un sistema de coordenadas normalizado, con coordenadas u y v, cumpliendo que $0 \leq u \leq 1$ y $0 \leq v \leq 1$. Esta normalización permite independizar el tamaño real de la imagen de su aplicación al modelo.

El siguiente paso consiste en asignarle a cada punto del modelo las coodenadas de textura correspondientes. En el ejemplo de la imagen, para representar un rectángulo necesitamos 2 triángulos. Si tenemos 4 puntos, P_0, P_1, P_2 P_3, el triángulo T_0 podría estar compuesto por los

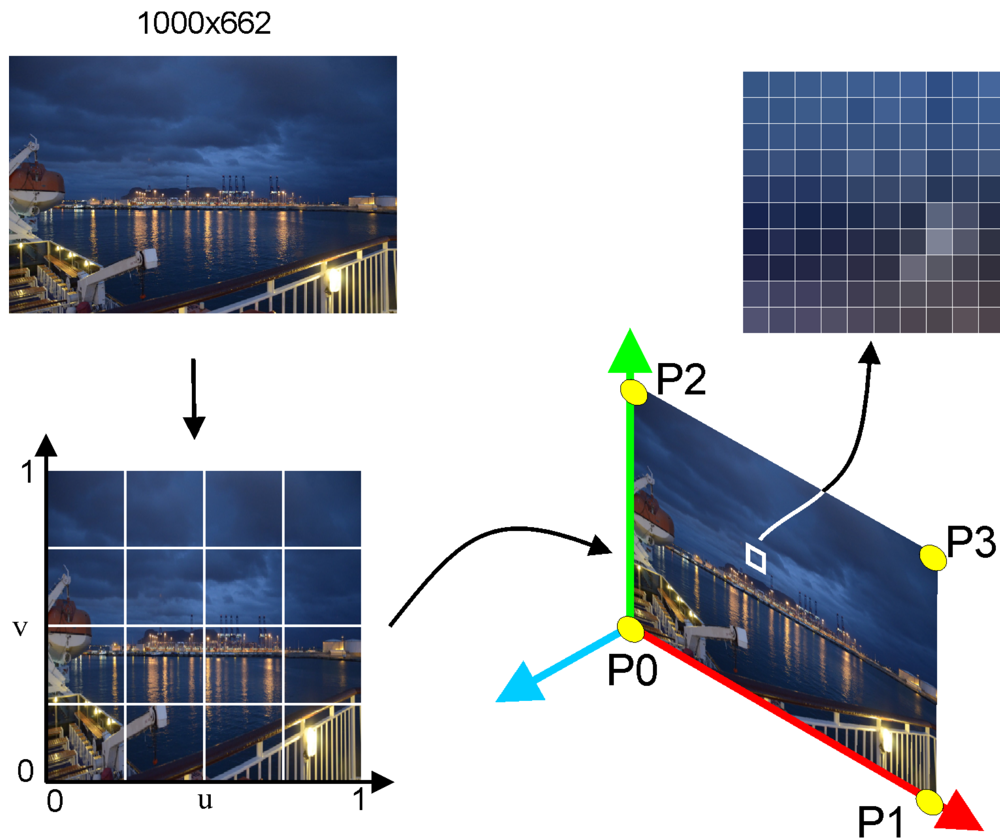


Figura 16.1: Pasos en la aplicación de una textura

puntos $(P_2, P_0, P1)$ y el triángulo T_1 por los puntos $(P_1, P_3, P2)$. Dado que queremos mostrar toda la textura, eso implica la siguiente asignación de coordenadas de textura:

- $(0,0) \rightarrow P_0$
- $(1,0) \rightarrow P_1$
- $(0,1) \rightarrow P_2$
- $(1,1) \rightarrow P_3$

Si solo quisiéramos mostrar la parte central, las coordenadas podrían ser las siguientes:

- $(0.25, 0.25) \rightarrow P_0$
- $(0,75, 0.25) \rightarrow P_1$
- $(0.25, 0.75) \rightarrow P_2$
- $(0.75, 0.75) \rightarrow P_3$

Domingo Martín Perandrés©

Es importante observar con estos dos ejemplos que la diferencia en lo que se muestra se ha conseguido simplemente cambiado las coordenada de textura, no las coordenadas de los puntos.

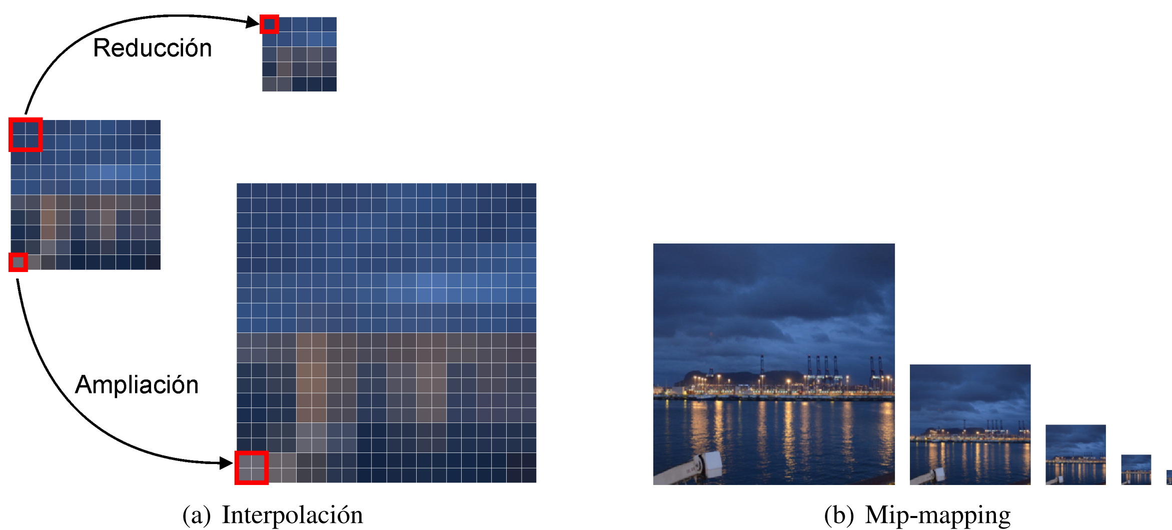
Una vez que se ha hecho la correspondencia, OpenGL se encarga del trabajo duro, realizando la correspondencia entre lo píxeles de la imagen de entrada y los píxeles de la imagen de salida, resolviendo los distintos problemas de escala que hay, realizando la interpolación, ajustando la perspectiva, si la hay, etc.

Uno de los factores importantes a tener en cuenta cuando se usa una textura es su naturaleza matricial, frente a la naturaleza continua de la geometría. Esto es, las imágenes tienen un tamaño finito y no se puede extraer más detalle del que hay. Veámoslo con un ejemplo. Si tengo la textura aplicada a un objeto, dependiendo de la posición de la cámara y el tipo de proyección me puedo encontrar con tres posibilidades:

- Hay una relacion 1:1 entre los píxeles de la textura y los píxeles de salida. En este caso no se hace nada.
- Hay mas píxeles en la textura que píxeles de salida. En este caso, estamos lejos del objeto. OpenGL tiene (si así se le pide) que combinar varios píxeles para producir uno de salida. En general, esto no es un problema pues estamos reduciendo la cantidad de información.
- Hay menos píxeles en la textura que píxeles de salida. En este caso, nos hemos acercado al objeto para ver los detalles. La imágen no tiene tanto nivel de detalle, pero para mejorar un poco la salida, OpenGL (si así se le pide) intenta generar los píxeles que faltan mediante una interpolación. En general, esto es un problema pues se intenta generar nueva información. La interpolación puede producir un resultado satisfactorio en objetos con cambios suaves pero no en otros casos.

La idea de la interpolación se puede ver en la figura 16.2(a). Si tenemos clara la idea de la interpolación, es fácil entender en qué consiste el *mip-mapping*. Dado que nos podemos alejar del objeto, parece evidente que cuanto más nos alejemos la textura que sería necesaria para visualizar correctamente debería ser cada vez más pequeña. En otro caso, si sólo tuviéramos la textura original, OpenGL tendría que encargarse de hacer la reducción. Imaginemos un caso extremo: el objeto está tan lejos que sólo hace falta un píxel para representarlo. El trabajo de reducir la textura puede ser muy costoso. Para resolver este problema aparece el mip-mapping: no sólo se tiene la textura al tamaño original sino que se crean versiones cada vez más pequeñas. Esto es, hacemos la reducción una vez y dejamos almacenado el resultado. De esta manera OpenGL, en función de la distancia puede elegir la textura cuyo tamaño mejor se ajuste. La idea del mip-mapping se puede ver en la figura 16.2(b)

Pasemos ahora a ver cómo se puede hacer uso de las texturas con OpenGL. Para que resulte más sencillo, nos vamos a centrar sólo en el código necesario para poder mostrar una textura de la manera más sencilla. Eso significa que no vamos a empezar directamente con



(a) Interpolación (b) Mip-mapping

Figura 16.2: Ajustes sobre las texturas

el código del ejemplo anterior sino que nos basaremos en uno con menos capacidades, especialmente en los shaders, donde no vamos a meter el cálculo de la iluminación. También el objeto al que le vamos a pegar la textura se simplifica, siendo un sencillo quad: dos triángulos que forman un rectángulo. Veamos los pasos que hay que dar para dibujar la textura.

Para poder meter las coordenadas de textura modificamos el objeto básico:

```
class _basic_object3d{
public:
  vector<_vertex3f> Vertices;
  vector<_vertex3f> Vertices_colors;
  vector<_vertex2f> Vertices_texture_coordinates;

  vector<_vertex3i> Triangles;

  vector<_vertex3f> Vertices_drawarray;
  vector<_vertex3f> Vertices_colors_drawarray;
  vector<_vertex2f> Vertices_texture_coordinates_drawarray;
};
```

Veamos ahora el código de creación del quad:

```
_quad::_quad()
{
  // vertices
  Vertices.resize(4);
  Vertices[0]=_vertex3f(-QUAD_SIZE,-QUAD_SIZE,0);
  Vertices[1]=_vertex3f(QUAD_SIZE,-QUAD_SIZE,0);
  Vertices[2]=_vertex3f(QUAD_SIZE,QUAD_SIZE,0);
  Vertices[3]=_vertex3f(-QUAD_SIZE,QUAD_SIZE,0);

  // triangles
  Triangles.resize(2);
  Triangles[0]=_vertex3i(0,1,3);
```

```
  Triangles[1]=_vertex3i(1,2,3);

  // vertices colors
  Vertices_colors.resize(4);
  Vertices_colors[0]=COLORS[1];
  Vertices_colors[1]=COLORS[2];
  Vertices_colors[2]=COLORS[3];
  Vertices_colors[3]=COLORS[4];

  // vertices texture coordinates
  Vertices_texture_coordinates.resize(4);
  Vertices_texture_coordinates[0]=_vertex2f(0,0);
  Vertices_texture_coordinates[1]=_vertex2f(1,0);
  Vertices_texture_coordinates[2]=_vertex2f(1,1);
  Vertices_texture_coordinates[3]=_vertex2f(0,1);

  // create drawarrays
  // vertices
  Vertices_drawarray.resize(Triangles.size()*3);
  for (unsigned int i=0;i<Triangles.size();i++){
    Vertices_drawarray[i*3]=Vertices[Triangles[i]._0];
    Vertices_drawarray[i*3+1]=Vertices[Triangles[i]._1];
    Vertices_drawarray[i*3+2]=Vertices[Triangles[i]._2];
  }
  // vertices colors
  Vertices_colors_drawarray.resize(Triangles.size()*3);
  for (unsigned int i=0;i<Triangles.size();i++){
    Vertices_colors_drawarray[i*3]=Vertices_colors[Triangles[i]._0];
    Vertices_colors_drawarray[i*3+1]=Vertices_colors[Triangles[i]._1];
    Vertices_colors_drawarray[i*3+2]=Vertices_colors[Triangles[i]._2];
  }

  // vertices texture coordinates
  Vertices_texture_coordinates_drawarray.resize(Triangles.size()*3);
  for (unsigned int i=0;i<Triangles.size();i++){
    Vertices_texture_coordinates_drawarray[i*3]=Vertices_texture_coordinates[Triangles[i↩
        ]._0];
    Vertices_texture_coordinates_drawarray[i*3+1]=Vertices_texture_coordinates[Triangles[↩
        i]._1];
    Vertices_texture_coordinates_drawarray[i*3+2]=Vertices_texture_coordinates[Triangles[↩
        i]._2];
  }
}
```

Se puede ver que es similar a los ejemplos anteriores sólo que tenemos que añadir la información de las coordenadas de textura. Por supuesto, tendremos que crear un VBO que contenga dicha información. Para mantener la coherencia con los ejemplos anteriores, usamos la siguiente posición libre. Veamos la creación y el rellenado de los datos:

```
...
glCreateBuffers(1,&VBO_texture_coordinates1);
glNamedBufferStorage(VBO_texture_coordinates1,(Axis.Vertices_drawarray.size()+Quad.↩
    Vertices_texture_coordinates_drawarray.size())*2*sizeof(float),nullptr,↩
    GL_DYNAMIC_STORAGE_BIT | GL_MAP_WRITE_BIT);
glVertexArrayVertexBuffer(VAO1,4,VBO_texture_coordinates1,0,2*sizeof(float)); // 1,VBO
glVertexArrayAttribFormat(VAO1,4,2,GL_FLOAT,GL_FALSE,0);
glEnableVertexArrayAttrib(VAO1,4);
...
glNamedBufferSubData(VBO_texture_coordinates1,Axis.Vertices_drawarray.size()*2*sizeof(↩
    GLfloat),Quad.Vertices_texture_coordinates_drawarray.size()*2*sizeof(GLfloat),&Quad.↩
    Vertices_texture_coordinates_drawarray[0]);
...
```

Sólo hay que tener en cuenta que no es necesario meter datos de coordenadas de textura para los ejes.

Para terminar la inicialiación tenemos que hacer que haya una textura por defecto. Una posibilidad es crear una imagen y otra es crearla. Veamos cómo se hace en el segundo caso:

```
QImage Image(DEFAULT_WINDOW_WIDTH,DEFAULT_WINDOW_HEIGHT,QImage::Format_RGB888);
Image.fill(QColor(DEFAULT_TONE,DEFAULT_TONE,DEFAULT_TONE,255));

glCreateTextures(GL_TEXTURE_2D,1,&Texture);
glTextureStorage2D(Texture,1,GL_RGB8,DEFAULT_WINDOW_WIDTH,DEFAULT_WINDOW_HEIGHT);

glBindTexture(GL_TEXTURE_2D,Texture);
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
// fill with data
glTextureSubImage2D(Texture,0,0,0,DEFAULT_WINDOW_WIDTH,DEFAULT_WINDOW_HEIGHT,GL_RGB,↩
    GL_UNSIGNED_BYTE,Image.constBits());
```

Usando la funcionalidad de Qt creamos una imagen del tamaño indicado, con el formato indicado, y la rellenamos de un color, en nuestro caso un gris claro.

Siguiendo la forma de operar de OpenGL, lo primero que hacemos es crear un identificador de textura. Es importante hacer notar que una vez creada una textura, incluidos sus parámetros, la misma no puede ser cambiada, obligando a borrarla y crear una nueva si se desea modificar el tamaño. Esto quiere decir que si creamos una textura de 1024×1024 podremos cambiar su contenido, los píxeles, cargando otra imagen, pero si la imagen que se carga es de 512×512 no se podrá usar. Lo mismo se puede decir del formato y otros atributos.

Una vez que tenemos el identificador definimos el formato y la necesidades de espacio. Para poder cambiar los parámetros para rodear (WRAP_S, WRAP_T) y de filtrado del tamaño (MAG_FILTER,MIN_FILTER), tenemos que hacer un enlace (*binding*), en este caso, del tipo GL_TEXTURE_2D. Entre las características de las texturas está la posibilidad de crear versiones a menor tamaño (*mip-mapping*), y que se realiza interpolación. En nuestro caso hemos elegido el caso más sencillo de interpolación: no interpolación (GL_NEAREST). La última instrucción se encarga de cargar los datos en la GPU. Una vez hecho esto se puede visualizar la textura.

Ahora sólo nos falta el código que se encarga de la visualización:

```
void _gl_widget::draw_objects()
{
  QMatrix4x4 Modelviewprojection;

  float Aspect=(float)Window_height/(float)Window_width;

  if (Projection_type==PERSPECTIVE_PROJECTION){
    Modelviewprojection.frustum(X_MIN,X_MAX,Y_MIN*Aspect,Y_MAX*Aspect,↩
        FRONT_PLANE_PERSPECTIVE,BACK_PLANE_PERSPECTIVE);
```

```
  }
  else{
    Modelviewprojection.ortho(X_MIN*Scale_factor,X_MAX*Scale_factor,Y_MIN*Aspect*↩
        Scale_factor,Y_MAX*Aspect*Scale_factor,FRONT_PLANE_PARALLEL,BACK_PLANE_PARALLEL);
  }

  Modelviewprojection.translate(0,0,-Distance);
  Modelviewprojection.rotate(Angle_x,1,0,0);
  Modelviewprojection.rotate(Angle_y,0,1,0);

  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

  glBindVertexArray(VAO1);

  glUseProgram(Program1);
  glUniformMatrix4fv(10,1,GL_FALSE,Modelviewprojection.data());

  // axis
  glUniform1i(20,(int)MODE_LINE);
  glDrawArrays(GL_LINES,0,Axis.Vertices_drawarray.size());

  // quad
  glPolygonMode(GL_FRONT_AND_BACK,GL_FILL);
  glUniform1i(20,(int)MODE_FILL);
  glUniform1i(24,(int)Texture_active);
  glBindTexture(GL_TEXTURE_2D,Texture);
  glDrawArrays(GL_TRIANGLES,Axis.Vertices_drawarray.size(),Quad.Vertices_drawarray.size()↩
      );

  glUseProgram(0);
  glBindVertexArray(0);
}
```

En este código simplificado, hay que destacar que hemos compactado todas las transformaciones en una sola, y que hemos añadido un variable para controlar que se active o no la textura. El detalle a tener en cuenta es que para poder hacer uso de la textura que ya habíamos cargado previamente, hay que hacer un binding indicando que es del tipo 2D.

Los shaders que se han implementado para este ejemplo son muy sencillos. El vertex shader es el siguiente:

```
#version 450 core

layout (location=0) in vec3 vertex;
layout (location=1) in vec3 color;
layout (location=4) in vec2 texture_coordinates;

layout (location=10) uniform mat4 modelviewprojection;

layout (location=20) uniform int mode_rendering;
layout (location=24) uniform int texture_active;

out vec3 color_out;
out vec2 texture_coordinates_out;

void main(void)
{
  if (mode_rendering==1){ // line
    color_out=color;
  }
  else{ // no line
    if (texture_active==0) color_out=color;
```

```
    else texture_coordinates_out=texture_coordinates;
  }
  gl_Position=modelviewprojection*vec4(vertex,1);
}
```

Se puede ver como el vertex shader pasa las coordenadas de textura al fragment shader.

Veamos ahora el fragment shader:

```
#version 450 core
layout (location=20) uniform int mode_rendering;
layout (location=24) uniform int texture_active;

uniform sampler2D texture_image;

in vec3 color_out;
in vec2 texture_coordinates_out;

out vec4 frag_color;

void main(void)
{
  if (mode_rendering==1){ // line
    frag_color=vec4(color_out,1);
  }
  else{ // no line
    if (texture_active==0) frag_color=vec4(color_out,1);
    else frag_color = texture(texture_image,texture_coordinates_out);
  }
}
```

En el fragment shader, además de las coordenadas de textura, tenemos una variable de tipo uniform que indica que es una textura 2D mediante la instrucción sampler2D. Y para muestrear la textura tenemos la función texture a la que se le pasa la textura y las coordenadas de textura y devuelve un color, encargándose de todo el trabajo de interpolación, ajuste de perspectiva, etc. ¡Sencillo!

Dado que nuestra geometría representa a un cuadrado, si la imagen que se carga no es cuadrada se producirá una deformación.

Con otras variables uniformes hemos podido asignarle una posición que luego servía para referenciarla y poder almacenar un valor mediante el modificador location. ¿Se puede hacer lo mismo con las variables que representan a las texturas? Sí, pero en este caso no se usa el modificador location sino el modificador binding. Existen puertos de enlace (*binding*) para la texturas y también se identifican con enteros. En OpenGL se usan los alias GL_TEXTUREx siendo "x" un numero entre 0 y el número máximo de puertos de enlace que tenga la implementación.

Para activar un puerto en el código principal se modificaría el código de la siguiente manera:

```
glPolygonMode(GL_FRONT_AND_BACK,GL_FILL);
glUniform1i(20,(int)MODE_FILL);
```

```
glUniform1i(24,(int)Texture_active);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D,Texture);
glDrawArrays(GL_TRIANGLES,Axis.Vertices_drawarray.size(),Quad.Vertices_drawarray.size());
```

Y el frament shader debería incluir la siguiente instrucción:

```
layout (binding=0) uniform sampler2D texture_image;
```

Este procedimiento se puede usar para poder usar más de una textura.

Antes de terminar, veamos el código que permite cargar cualquier imagen y convertirla en textura.

Lo primero que tenemos que hacer es cargar la imagen. En nuestro caso vamos a usar la infraestructura de Qt para hacerlo.

Mediante el siguiente código cargamos la imagen en una QImage

```
bool _window::load_file(const QString &fileName,QImage &Image)
{
  QImageReader Reader(fileName);
  Reader.setAutoTransform(true);
  Image = Reader.read();
  if (Image.isNull()) {
    QMessageBox::information(this, QGuiApplication::applicationDisplayName(), tr("Cannot ↩
        load %1.").arg(QDir::toNativeSeparators(fileName)));
    return false;
  }
  Image=Image.mirrored();
  Image=Image.convertToFormat(QImage::Format_RGB888);
  return true;
}
```

Un detalle a tener en cuenta es que los formatos de imágenes suelen usar un sistema de coordenadas izquierdo, con el origen en la esquina superior izquierda mientras que OpenGL usa un sistema de coordenadas derecho con el origen en la esquina inferior izquierda. Por ello aplicamos una operación de reflejo horizontal.

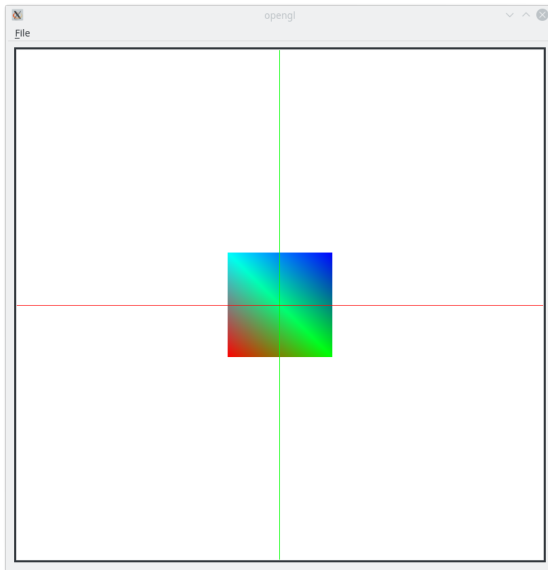
Ahora necesitamos cargar la imagen en la GPU:

```
void _gl_widget::load_image(QImage &Image1)
{
  glDeleteTextures(1,&Texture);

  glCreateTextures(GL_TEXTURE_2D,1,&Texture);
  glTextureStorage2D(Texture,1,GL_RGB8,Image1.width(),Image1.height());

  glBindTexture(GL_TEXTURE_2D,Texture);
  glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
  glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
  glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
  glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
  // fill with data
```

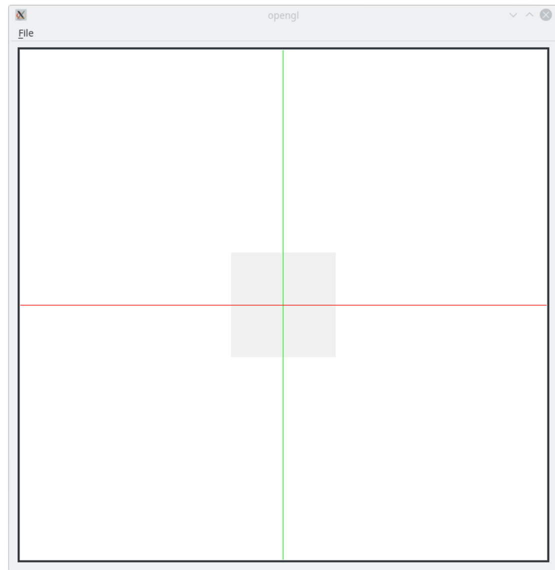
```
  glTextureSubImage2D(Texture,0,0,0,Image1.width(),Image1.height(),GL_RGB,↩
      GL_UNSIGNED_BYTE,Image1.constBits());

  update();
}
```

El único cambio a destacar con respecto a la creación de la textura inicial es que se borra la textura anterior. Esto se pone para permitir el poder leer distintas imágenes sin tener que reiniciar el programa, y, sobre todo, y tal como hemos comentado anteriormente, porque si hay algún cambio en los parámetros de la textura que se quiere usar con respecto a la que hay,el proceso implica destruir la anterior y crear una nueva.

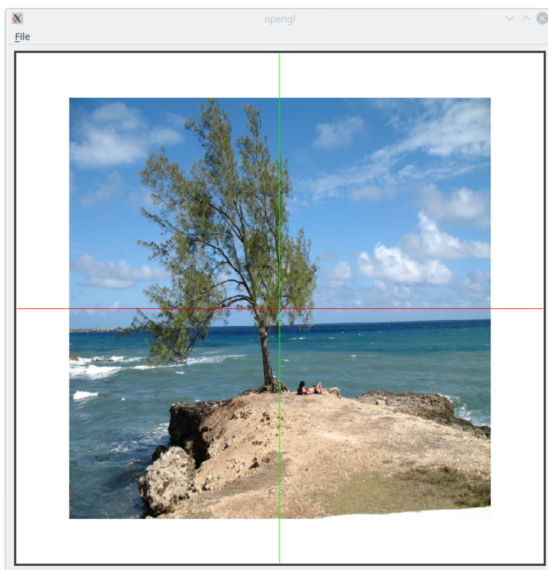
Domingo Martín Perandrés©



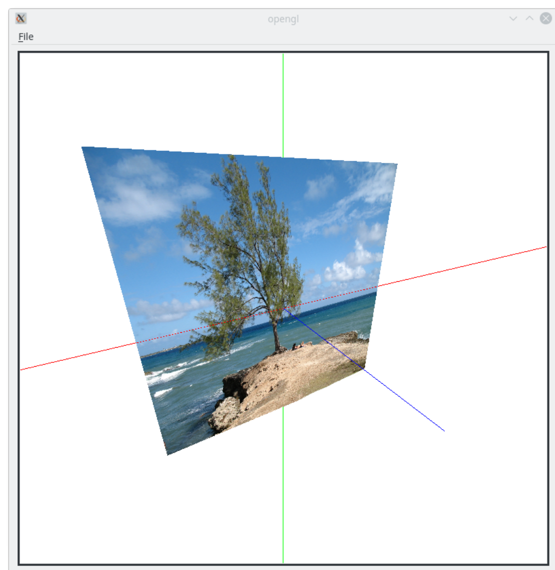
(a) Interpolación del color sin textura



(b) Textura inicial



(c) Textura de una imagen



(d) Textura de una imagen con cambio de posición

Figura 16.3: Resultados del ejemplo 16

Domingo Martín Perandrés©

17

Texturas e iluminación

Una vez hemos visto como se puede visualizar una textura vamos a desarrollar un ejemplo un poco más complejo, añadiendo la aplicación de la iluminación a la textura. Para ello vamos a crear un modelo de la Tierra utilizando una esfera. Esto nos va permitir reincidir sobre el cálculo de las coordenadas de textura, en este caso aplicadas a un objeto cerrado.

Vamos a empezar por el cálculo de las coordenadas de textura para la esfera que, como recordaremos, se creaba por barrido circular. Si repasamos lo visto en el tema 10, veremos que fuimos haciendo evolucionar nuestro modelo desde el más simple en el que se repetían los puntos de los extremos (polos), produciendo triángulos degenerados, hasta el más optimizado en el que se eliminaban las repeticiones de vértices y los triángulos que no aportaban nada.

Para poder aplicarle una textura a nuestro modelo por revolución tenemos que, de alguna manera, retroceder a la versióno más simple en la que se repetían los vértices de los extremos, pero sin añadir los triángulos degenerados. El motivo de este cambio es hacer que la deformación que se produzca en los triángulos de las tapas sea similar para cada uno de ellos.

Si hacemos corresponder la estructura de la figura 17.1 con el espacio paramétrico en el que se define la textura, podemos ver que al vértice P_3 le corresponden las coordenadas $(0,1)$, y en correspondencia, al vértice P_{11} le corresponden las coordenadas $(0.5,1)$. Estos puntos se han puesto de ejemplo. El resto es fácil de calcular.

Si en vez de hacer la correspondencia anterior intentamos usar la distribución de puntos mostrada en la figura 17.2, es fácil que el vértice P_{15} tendrá, por ejemplo, las coordenadas $(0.5,1)$ y que cada triángulo de la tapa cubrirá partes de distinto tamaño y forma de la imagen. Esta opción es posible pero menos regular que la anterior.

El otro cambio importante se puede visualizar mostrando el problema que tenemos con la estructura de la figura 17.1. El problema está en que hemos hecho que el último perfil sea

Domingo Martín Perandrés©

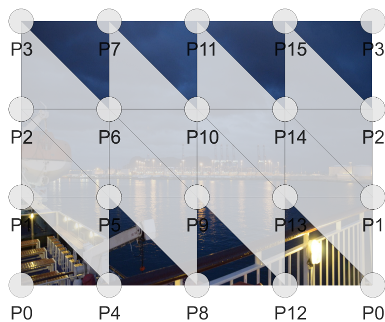


Figura 17.1: Localización de los puntos con respecto a las coordenadas de textura con distribución más regular pero errónea en el último perfil

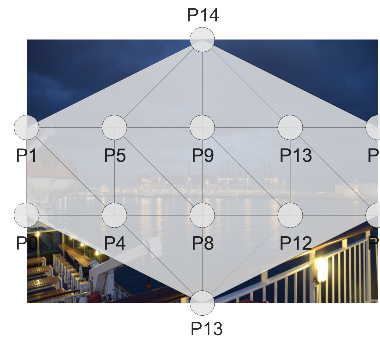


Figura 17.2: Localización de los puntos con respecto a las coordenadas de textura con distribución menos regular y errónea en el último perfil

el mismo que el primero. Para el modelado que hemos hecho hasta ahora, la solución era correcta, pero no ocurre lo mismo si deseamos usar texturas. El problema es muy fácil de ver. Si nos fijamos, el triángulo formado por P_1, P_4 y P_5, con las siguientes coordenadas de textura $(0,0.33)$, $(0.25,0)$ y $(0.25,0.33)$, respectivamente, se dibujará correctamente, pero el triángulo formado por P_{13}, P_0 y P_1, con las siguientes coordenadas de textura $(0.75,0.33)$, $(0,0)$ y $(0,0.33)$, respectivamente, no se dibujará bien. El problema está en intentar reusar las coordenadas de textura de los puntos P_0 y P_1. La solución pasa por repetir los puntos del primer perfil de tal manera que tengamos los puntos P_{16}, P_{17} y P_{18}, que ocupan las mismas posiciones que los puntos P_0, P_1 y P_2, pero que ahora sí podrán tener las coordenadas de textura correctas, $(1,0)$, $(1,0.33)$ y $(1,0.66)$. De esta manera el triángulo P_{13}, P_{16} y P_{17} se dibujará correctamente. (IMAGEN)

Para poder aplicar la iluminación a las texturas lo que se hace es combinar lo que se ha hecho en el ejemplo de iluminación y en el de texturas.

Veamos el código de visualización:

```
void _gl_widget::draw_objects()
{
  QMatrix4x4 Modelview;
  QMatrix4x4 Projection;
  QMatrix4x4 Modelview_normals;
  QMatrix4x4 Camera;

  float Aspect=(float)Window_height/(float)Window_width;

  if (Projection_type==PERSPECTIVE_PROJECTION){
    Projection.frustum(X_MIN,X_MAX,Y_MIN*Aspect,Y_MAX*Aspect,FRONT_PLANE_PERSPECTIVE,↩
        BACK_PLANE_PERSPECTIVE);
  }
  else{
    Projection.ortho(X_MIN*Scale_factor,X_MAX*Scale_factor,Y_MIN*Aspect*Scale_factor,↩
        Y_MAX*Aspect*Scale_factor,FRONT_PLANE_PARALLEL,BACK_PLANE_PARALLEL);
  }
```

Domingo Martín Perandrés©

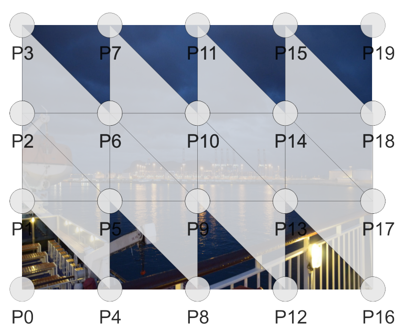


Figura 17.3: Localización de los puntos con respecto a las coordenadas de textura con distribución más regular y el último perfil correcto

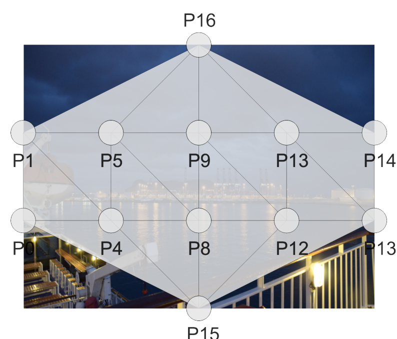


Figura 17.4: Localización de los puntos con respecto a las coordenadas de textura con distribución menos regular y el último perfil correcto

```
Modelview.translate(0,0,-Distance);
Modelview.rotate(Angle_x,1,0,0);
Modelview.rotate(Angle_y,0,1,0);

Modelview_normals=Modelview;
Modelview_normals.inverted();
Modelview_normals.transposed();

Camera.rotate(-Angle_y,0,1,0);
Camera.rotate(-Angle_x,1,0,0);
QVector3D Camera_position=(QVector3D)(Camera*QVector4D(0,0,Distance,1));

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

glBindVertexArray(VAO1);

glUseProgram(Program1);
glUniformMatrix4fv(10,1,GL_FALSE,Modelview.data());
glUniformMatrix4fv(11,1,GL_FALSE,Projection.data());
glUniformMatrix4fv(12,1,GL_FALSE,Modelview_normals.data());


// axis
glUniform1i(20,(int)MODE_LINE);
glUniform1i(21,(int)MODE_COLORS_VARIABLE);
glUniform1i(23,(int)false); // no illumination
glUniform1i(24,(int)false); // no texture

glDrawArrays(GL_LINES,0,Axis.Vertices_drawarray.size());

if (Draw_points){
  glPolygonMode(GL_FRONT_AND_BACK,GL_POINT);
  glPointSize(3);
  glUniform1i(20,(int)MODE_POINT);
  glUniform1i(21,(int)MODE_COLORS_FIXED);
  glUniform3fv(25,1,(GLfloat*) &COLORS[COLOR_POINT]);
  glDrawArrays(GL_TRIANGLES,Axis.Vertices_drawarray.size(),Sphere.Vertices_drawarray.↩
      size());
}

if (Draw_lines){
  glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
```

```
    glUniform1i(20,(int)MODE_LINE);
    glUniform1i(21,(int)MODE_COLORS_FIXED);
    glUniform3fv(26,1,(GLfloat*) &COLORS[COLOR_LINE]);
    glDrawArrays(GL_TRIANGLES,Axis.Vertices_drawarray.size(),Sphere.Vertices_drawarray.↩
        size());
  }

  if (Draw_fill){
    glPolygonMode(GL_FRONT_AND_BACK,GL_FILL);
    glUniform1i(20,(int)MODE_FILL);
    glUniform1i(21,Mode_color);
    glUniform3fv(27,1,(GLfloat*) &COLORS[COLOR_FILL]);

    if (Illumination_active){
      glUniform1i(22,(int)Mode_shading);
      glUniform1i(23,(int)Illumination_active);
      glUniform4fv(30,1,(GLfloat*) &Light.Position);
      glUniform3fv(31,1,(GLfloat*) &Light.Color);
      glUniform3fv(32,1,(GLfloat*) &Camera_position);
      glUniform3fv(40,1,(GLfloat*) &Sphere_material.Ambient);
      glUniform3fv(41,1,(GLfloat*) &Sphere_material.Diffuse);
      glUniform3fv(42,1,(GLfloat*) &Sphere_material.Specular);
      glUniform1f(43,Sphere_material.Specular_exponent);
      glUniform3fv(44,1,(GLfloat*) &Ambient_coeff);
    }
    else{
      glUniform1i(23,(int)Illumination_active);
    }

    if (Texture_active){
      glUniform1i(24,(int)Texture_active);
      glActiveTexture(GL_TEXTURE0);
      glBindTexture(GL_TEXTURE_2D,Texture);
    }
    else{
      glUniform1i(24,(int)Texture_active);
    }

    glDrawArrays(GL_TRIANGLES,Axis.Vertices_drawarray.size(),Sphere.Vertices_drawarray.↩
        size());
  }

  glUseProgram(0);
  glBindVertexArray(0);
}
```

Lo más destacable es que hemos vuelto a recuperar la Modelview y la Projection como matrices individuales. Esto se hace para poder calcular la matriz de transformación de las normales.

El trabajo duro es realizado en los shaders. El vertex shader es el siguiente:

```
#version 450 core
layout (location=0) in vec3 vertex;
layout (location=1) in vec3 color;
layout (location=2) in vec3 triangle_normal;
layout (location=3) in vec3 vertex_normal;
layout (location=4) in vec2 texture_coordinates;

layout (location=10) uniform mat4 modelview;
layout (location=11) uniform mat4 projection;
layout (location=12) uniform mat4 modelview_normals;
```

```
layout (location=20) uniform int mode_rendering;
layout (location=21) uniform int mode_color;
layout (location=22) uniform int mode_shading;
layout (location=23) uniform int illumination_active;
layout (location=24) uniform int texture_active;

layout (location=25) uniform vec3 color_point;
layout (location=26) uniform vec3 color_line;
layout (location=27) uniform vec3 color_fill;

layout (location=30) uniform vec4 light_position;
layout (location=31) uniform vec3 light_color;
layout (location=32) uniform vec3 camera_position;

layout (location=40) uniform vec3 material_ambient;
layout (location=41) uniform vec3 material_diffuse;
layout (location=42) uniform vec3 material_specular;
layout (location=43) uniform float material_specular_exponent;
layout (location=44) uniform vec3 ambient_coeff;

out vec3 color_out;
out vec3 vertex_out;
out vec3 normal_out;
out vec2 texture_coordinates_out;

void main(void)
{
  if (mode_color==1){ // color variable
    if (mode_rendering==2){ // fill
      if (texture_active==1) texture_coordinates_out=texture_coordinates;
      if (illumination_active==1){
        switch(mode_shading){
          case 0: // flat
            {
            // ambient
            vec3 ambient_component=ambient_coeff*light_color*material_ambient;

            // diffuse
            vec3 light_direction;
            if (light_position.w==0){
            light_direction=vec3(light_position);
            }
            else{
            light_direction=vec3(light_position)-vertex;
            }
            light_direction=normalize(light_direction);
            float diffuse_intensity = max(dot(light_direction,triangle_normal), 0.0);
            vec3 diffuse_component = diffuse_intensity*material_diffuse;

            // specular
            vec3 view_direction=camera_position-vertex;
            view_direction=normalize(view_direction);
            vec3 reflected_direction = reflect(-light_direction,triangle_normal);
            float specular_intensity = pow(max(dot(view_direction,reflected_direction), ↩
                 0.0), material_specular_exponent);
            vec3 specular_component = specular_intensity*material_specular;

            if (texture_active==1) color_out=vec3(ambient_coeff+diffuse_intensity+↩
                 specular_intensity);
            else color_out=ambient_component+diffuse_component+specular_component;
            }
            break;
          case 1: // Gouraud
            {
            // ambient
            vec3 ambient_component=ambient_coeff*light_color*material_ambient;
```

```
            // diffuse
            vec3 light_direction;
            if (light_position.w==0){
            light_direction=vec3(light_position);
            }
            else{
            light_direction=vec3(light_position)-vertex;
            }
            light_direction=normalize(light_direction);
            float diffuse_intensity = max(dot(light_direction,vertex_normal), 0.0);
            vec3 diffuse_component = diffuse_intensity*material_diffuse;

            // specular
            vec3 view_direction=camera_position-vertex;
            view_direction=normalize(view_direction);
            vec3 reflected_direction = reflect(-light_direction,vertex_normal);
            float specular_intensity = pow(max(dot(view_direction,reflected_direction), ↩
                0.0), material_specular_exponent);
            vec3 specular_component = specular_intensity*material_specular;

            if (texture_active==1) color_out=vec3(ambient_coeff+diffuse_intensity+↩
                specular_intensity);
            else color_out=ambient_component+diffuse_component+specular_component;
            }
            break;
          case 2: // Phong
            vertex_out=vertex;
            normal_out=vertex_normal;
            break;
        }
      }
      else{ // no illumination
        color_out=color;
      }
    }
    else{ // no fill
      color_out=color;
    }
  }
  else{ // color fixed
    if (texture_active==1){ // texture active
      texture_coordinates_out=texture_coordinates;
    }
    else{
      switch (mode_rendering){
        case 0: color_out=color_point;break;
        case 1: color_out=color_line;break;
        case 2: color_out=color_fill;break;
      }
    }
  }

  gl_Position=projection*modelview*vec4(vertex,1);
}
```

El código no es muy diferente del que vimos en el tema anterior.

Veamos ahora el fragment shader:

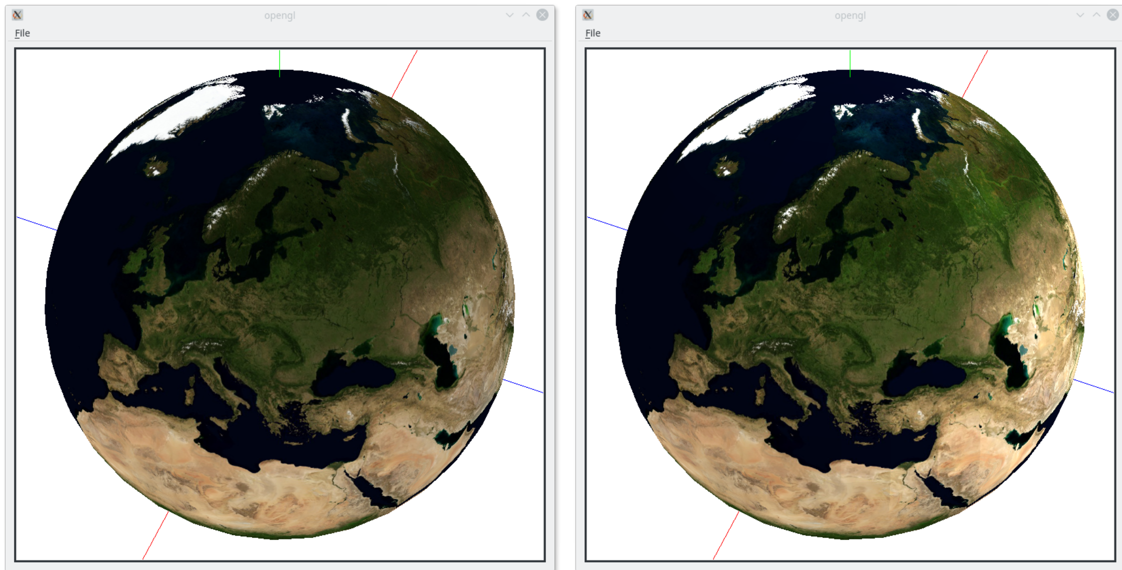
```
#version 450 core
layout (location=20) uniform int mode_rendering;
layout (location=21) uniform int mode_color;
layout (location=22) uniform int mode_shading;
layout (location=23) uniform int illumination_active;
```

```
layout (location=24) uniform int texture_active;

layout (location=30) uniform vec4 light_position;
layout (location=31) uniform vec3 light_color;
layout (location=32) uniform vec3 camera_position;

layout (location=40) uniform vec3 material_ambient;
layout (location=41) uniform vec3 material_diffuse;
layout (location=42) uniform vec3 material_specular;
layout (location=43) uniform float material_specular_exponent;
layout (location=44) uniform vec3 ambient_coeff;

layout (binding=0) uniform sampler2D texture_image0;

in vec3 color_out;
in vec3 vertex_out;
in vec3 normal_out;
in vec2 texture_coordinates_out;

out vec4 frag_color;

void main(void)
{
  if (mode_color==1){ // interpolated
    if (mode_rendering==2){ // fill
      if (illumination_active==1){ //
        if (mode_shading==2){ // Phong
          // diffuse
          vec3 light_direction;
          if (light_position.w==0){
            light_direction=vec3(light_position);
          }
          else{
            light_direction=vec3(light_position)-vertex_out;
          }
          light_direction=normalize(light_direction);
          vec3 normal_normalized=normalize(normal_out);
          float diffuse_intensity = max(dot(light_direction,normal_normalized), 0.0);

          // specular
          vec3 view_direction=camera_position-vertex_out;
          view_direction=normalize(view_direction);
          vec3 reflected_direction = reflect(-light_direction,normal_normalized);
          float specular_intensity = pow(max(dot(view_direction,reflected_direction), ↩
              0.0), material_specular_exponent);

          if (texture_active==0){ // only illumination
            // ambient
            vec3 ambient_component=ambient_coeff*light_color*material_ambient;
            vec3 diffuse_component = diffuse_intensity*material_diffuse;
            vec3 specular_component = specular_intensity*material_specular;
            frag_color=vec4(ambient_component+diffuse_component+specular_component,1);
          }
          else{ // texuture+illumination
            vec3 color_texture0=vec3(texture(texture_image0,texture_coordinates_out));

            vec3 ambient_component=ambient_coeff*light_color*color_texture0;
            vec3 diffuse_component = diffuse_intensity*color_texture0;
            vec3 specular_component = specular_intensity*material_specular;
            frag_color=vec4(ambient_component+diffuse_component+specular_component,1);
          }
        }
        else{ // no Phong
          if (texture_active==1) frag_color=texture(texture_image0,↩
              texture_coordinates_out)*vec4(color_out,1);
          else frag_color=vec4(color_out,1);
        }
```

```
      }
      else{ // no illumination
        frag_color=vec4(color_out,1);
      }
    }
    else{ // no fill
      frag_color=vec4(color_out,1);
    }
  }
  else{
    if (texture_active==1) frag_color = texture(texture_image0,texture_coordinates_out);
    else frag_color=vec4(color_out,1);
  }
}
```

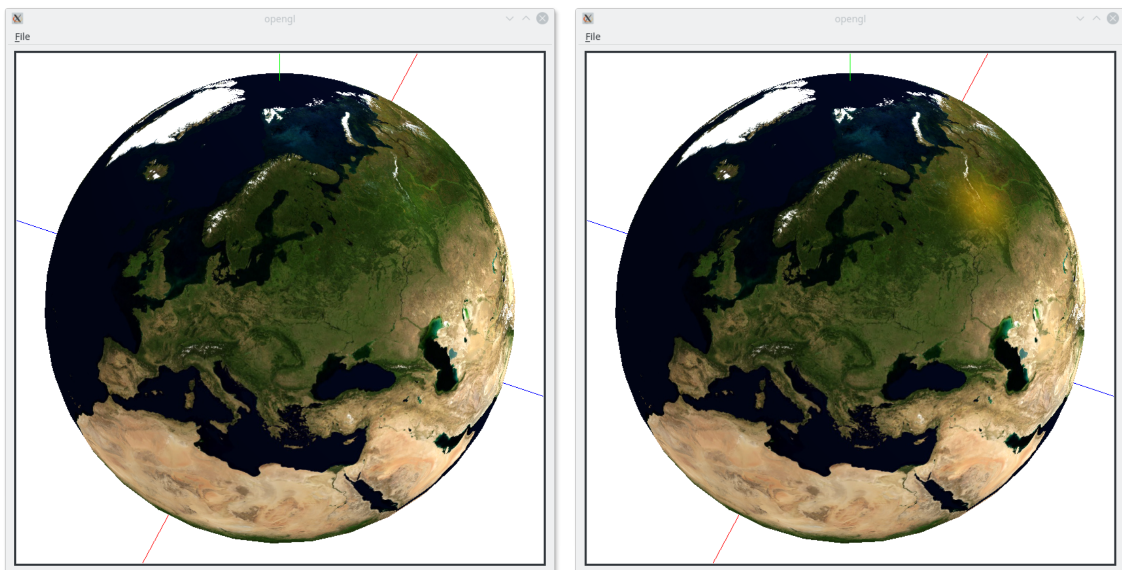
Se puede ver que la gran diferencia es que ahora el color del material sobre el que se aplica la iluminación es el color resultante de muestrear la textura.

Domingo Martín Perandrés©



(a) Textura sin iluminación

(b) Textura con iluminación y sombreado plano



(c) Textura con iluminación y sombreado de Gouraud

(d) Textura con iluminación y sombreado de Phong

Figura 17.5: Resultados del ejemplo 17

Domingo Martín Perandrés©

18

Selección

En este tema vamos a desarrollar el código para hacer una selección por color. En primer lugar tenemos que entender qué es la selección. La idea es muy sencilla: dados el conjunto de elementos de la escena queremos tener la capacidad para poder seleccionar individualmente cada uno de ellos, o alguna de sus partes, para posteriormente aplicarle algún tipo de proceso. Un ejemplo sencillo sería el poder seleccionar un objeto para borrarlo, moverlo o realizar cualquier otro tipo de operación.

La selección se podría hacer de distintas maneras como por ejemplo podría ser el tener asignado un identificador, numérico o nominal, para cada objeto o parte, y mediante el teclado realizar la selección. Esto sería bastante engorroso. En vez de ello, lo que queremos es que la selección se base en un proceso visual e interactivo, queremos poder realizar la selección mediante el ratón, mediante una correlación visual. Esto es, queremos que el objeto más cercano sobre el que se pone el cursor del ratón pueda ser seleccionado.

Existen varias formas de implementar esta idea, desde el método de selección implementado en las primeras versiones de OpenGL, lanzar un rayo a través de la posición que ha sido marcada y ver si se produce una intersección con algún objeto, a la selección por color que es la que vamos a implementar.

El método es relativamente sencillo. Cada objeto o parte que queramos identificar debe tener asociado un identificador, un valor numérico en nuestro caso. La idea clave consiste en convertir cada identificador en un color de tal manera que hay una relación única entre ambos, sin posibilidad de repetición. De esta manera podemos asegurar que podemos pasar de un número a un color y también el paso inverso de un color a un identificador. Si este proceso es posible, y lo es, basta con dibujar los objetos de la escena con los colores correspondientes a sus identificadores. El dibujado con la eliminación de partes activa hace que sólo los píxeles visibles sean dibujados, esto es los que están más cerca del observador. Si ahora colocamos el cursor en la posición deseada y hacemos una selección (*pick*) al pulsar algunos de

Domingo Martín Perandrés©

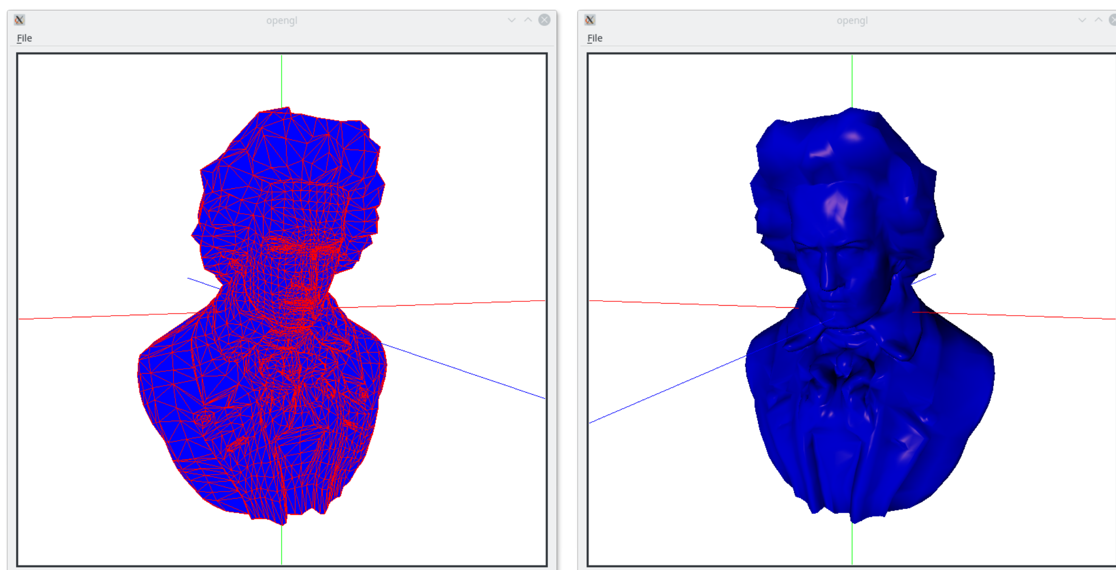


Figura 18.1: Modelo PLY de Beethoven

los botones del ratón (o cualquier tecla), sólo tenemos que recuperar el color de la posición seleccionada y convertirlo en un identificador, que nos indicará el objeto seleccionado.

La conversión de identificador a color y viceversa es muy fácil de hacer. Sólo tenemos que recordar que cuando dibujamos con un color, el mismo esta codificado mediante el modelo de color RGB, de tal manera que cada componente tiene un byte para poder representar 256 tonalidades. Los 24 bits para el color nos dan 16777216 millones de combinaciones. Esto es, con 24 bits podríamos distinguir 16777215 objetos distintos, uno menos que el total pues el color blanco sirve para identificar que no se ha hecho ninguna selección.

Antes de ver en detalle el código que hace la selección vamos a comentar el objeto que vamos a usar: en vez de la esfera vamos a cargar un objeto definido con el formato PLY. Este formato es ampliamente utilizado por su sencillez. El modelo más sencillo se describe con un conjunto de vértices y un conjunto de polígonos que se forman con dichos vértices. Los vértices se definen con coordenadas reales y los polígonos se definen con las posiciones de los vértices. En nuestro caso estamos interesados en objetos que estén sólo descritos por triángulos.

Dado el código lector de ficheros PLY, un objeto PLY es fácilmente asimilable a cualesquiera de los objetos gráficos que hemos visto con anterioridad, incluyendo el cálculo de las normales de los triángulos y de los vértices. Se puede ver un ejemplo de un modelo PLY, Beethoven, en la figura 18.1.

Como queremos distinguir los triángulos que componen el modelo, tendremos que asociar un identificador a cada triángulo. La forma más sencilla de hacerlo es asociar a cada triángulo su propia posición, desde 0 hasta 5029, pues Beethoven está compuesto de 5030

triángulos. Por tanto cada entero debe ser convertido a un color RGB. Para ello aplicamos la siguiente idea: todos los identificadores desde 0 a 255 irán en la componente azul, *blue*, desde 256 hasta 65535 corresponderán al verde, *green*, y desde 65536 hasta 16777215 corresponden al rojo, *red*. Por tanto, para saber que valor corresponde a cada parte sólo tenemos que dividir el número entre 65536, siendo el cociente la componente roja. El resto lo dividimos entre 256, siendo el cociente la parte del verde, y el resto la parte del azul. Pero como sabemos que el número esta codificado en binario y sabemos que cada byte corresponde una componente de color, las operaciones se pueden simplificar con máscaras y rotaciones binarias (multiplicaciones y divisiones por 2).

Sólo nos queda ver cómo pasamos el identificador de cada triángulo. Una posibilidad sería crear una estructura de datos que se pasara a la GPU de tal manera que para cada triángulo se pudiera leer su identificador. En vez de eso vamos a utilizar una variable interna de GLSL que está presente en los fragment shaders: gl_PrimitiveID. Esta variable indica la posición de la primitiva a la que pertenece el fragmento que se está dibujando. Esto es, si estoy dibujando triángulos, me dará la posición del triángulo al que pertenece el fragmento. ¡Eso es justo lo que queremos! Por tanto, sólo tendremos que convertir el valor de la variable gl_PrimitiveID a un color RGB. Esto es lo que hacemos con el siguiente código del fragment shader:

```
#version 450 core

in vec3 vertex_out;

out vec4 frag_color;

void main(void)
{
  vec3 Color;
  Color.r= (gl_PrimitiveID & 0x00FF0000) >> 16;
  Color.g= (gl_PrimitiveID & 0x0000FF00) >> 8;
  Color.b= gl_PrimitiveID & 0x000000FF;
  Color=Color/255.0;
  frag_color=vec4(Color,1);
}
}
```

El único detalle a tener en cuenta es que tenemos que normalizar los valores de las componentes de color.

El vertex shader es muy sencillo y sólo deber realizar el cálculo de la posición:

```
layout (location=0) in vec3 vertex;

layout (location=10) uniform mat4 modelview;
layout (location=11) uniform mat4 projection;

out vec3 color_out;

void main(void)
{
  gl_Position=projection*modelview*vec4(vertex,1);
}
```

Debemos tener en cuenta que estos shaders sólo se van a encargar de la conversión de enteros a colores. Como se puede comprobar, son muy diferentes de los shaders que hemos visto en los anteriores ejemplos para visualizar. Por tanto, nos encontramos que vamos a tener dos programas en la GPU: uno que se encarga, cuando sea necesario, de hacer la conversión de enteros a colores para poder hacer la selección, y un segundo que se encarga de la visualización normal y también visualizar el objeto seleccionado. ¿Se podrían haber hecho todo con un solo programa? Sí, pero de esta manera resulta más clara la división de la funcionalidad, y además es más fácil de mantener.

Si hemos entendido correctamente el proceso, se verá que estamos usando el framebuffer, la zona de memoria donde se almacena la imagen final, no sólo para realizar la visualización final sino también para realizar la selección. Además, al realizar el proceso de selección vamos obtener una imagen del modelo pero con unos colores que no son los que representan las características visuales del objeto sino la codificación por posición, tal como se puede apreciar en la figura 18.2(a). Podríamos hacer nuestra implementación con el framebuffer pero esto implicará que vamos a ver en pantalla la visualización de la codificación, sólo un instante, y después la visualización normal. Esto va a producir un parpadeo desagradable a la visión. La solución está en utilizar otro framebuffer.

La idea de tener varios framebuffers es muy sencilla: un framebuffer son un conjunto de una o más zonas de memoria donde se dibuja y/o realizan otras operaciones. Hasta ahora hemos utilizado el framebuffer por defecto, el cual tiene al menos dos zonas de dibujado para poder llevar a cabo la técnica del doble buffer, y otra zona de memoria para poder hacer la eliminación de partes ocultas mediante el algoritmo del z-buffer. Es importante tener en cuenta que sólo puede haber un framebuffer principal que es el que permite mostrar la información, normalmente en pantalla. El resto de framebuffers que se puedan crear no están conectados a la pantalla y por eso se les llama *off-screen*.

Para nuestro algoritmo, esta condición nos viene muy bien pues sólo deseamos hacer unos cálculos y que no se vea el resultado de los mismos. Esto es, queremos dibujar pero no queremos ver el resultado del dibujo sino sólo usarlo. Para ello tenemos que crear un framebuffer que tendrá asociado una zona de memoria para dibujar y otra zona de memoria para usarla para el z-buffer. Una vez que se define y se activa un framebuffer, todo funciona como hemos visto hasta ahora y, por tanto, todo lo que se dibuje lo hará en el framebuffer activo (en la memoria de dibujado).

Para implementar la selección hemos definido una función que se encargue de la tarea de crear el framebuffer nuevo y sus memorias asociadas, dibuje el objeto con los shaders que hemos visto anteriormente, y finalmente, a partir de la posición indicada nos devuelva el identificador del triángulo que se haya seleccionado. Una vez ha terminado desactiva el programa y el VAO y borra las estructuras de datos. Veamos el código:

```
void _gl_widget::pick()
{
  QMatrix4x4 Modelview;
  QMatrix4x4 Projection;
```

```
QMatrix4x4 Modelview_normals;
QMatrix4x4 Camera;
QMatrix4x4 Light;

float Aspect=(float)Window_height/(float)Window_width;

if (Projection_type==PERSPECTIVE_PROJECTION){
  Projection.frustum(-Camera_width,Camera_width,-Camera_width*Aspect,Camera_width*↩
      Aspect,FRONT_PLANE_PERSPECTIVE,BACK_PLANE_PERSPECTIVE);
}
else{
  Projection.ortho(X_MIN*Scale_factor,X_MAX*Scale_factor,Y_MIN*Aspect*Scale_factor,↩
      Y_MAX*Aspect*Scale_factor,FRONT_PLANE_PARALLEL,BACK_PLANE_PARALLEL);
}

Modelview.translate(0,0,-Distance);
Modelview.rotate(Angle_camera_x,1,0,0);
Modelview.rotate(Angle_camera_y,0,1,0);

makeCurrent();

// Frame Buffer Object to do the off-screen rendering
glGenFramebuffers(1,&FBO);
glBindFramebuffer(GL_FRAMEBUFFER,FBO);

// Texture for drawing
glGenTextures(1,&Color_texture);
glBindTexture(GL_TEXTURE_2D,Color_texture);
// RGBA8
glTexStorage2D(GL_TEXTURE_2D,1,GL_RGBA8, Window_width,Window_height);
// this implies that there is not mip mapping
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST);

// Texure for computing the depth
glGenTextures(1,&Depth_texture);
glBindTexture(GL_TEXTURE_2D,Depth_texture);
// Float
glTexStorage2D(GL_TEXTURE_2D,1,GL_DEPTH_COMPONENT24, Window_width,Window_height);

// Attatchment of the textures to the FBO
glFramebufferTexture(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,Color_texture,0);
glFramebufferTexture(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,Depth_texture,0);

// OpenGL will draw to these buffers (only one in this case)
static const GLenum Draw_buffers[]={GL_COLOR_ATTACHMENT0};
glDrawBuffers(1,Draw_buffers);

glClearColor(1,1,1,1);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

glBindVertexArray(VAO1);
glUseProgram(Program2);

glUniformMatrix4fv(10,1,GL_FALSE,Modelview.data());
glUniformMatrix4fv(11,1,GL_FALSE,Projection.data());

// draw the model
glDrawArrays(GL_TRIANGLES,Axis.Vertices_drawarray.size(),Ply_object.Vertices_drawarray.↩
    size());

// get the pixel
int Color;
glReadBuffer(GL_FRONT);
glPixelStorei(GL_PACK_ALIGNMENT,1);
glReadPixels(Selection_position_x,Selection_position_y,1,1,GL_RGBA,GL_UNSIGNED_BYTE,&↩
    Color);
```

```
  uint B=uint((Color & 0x00FF0000) >> 16);
  uint G=uint((Color & 0x0000FF00) >> 8);
  uint R=uint((Color & 0x000000FF));

  Selected_triangle= (R << 16) + (G << 8) + B;

  if (Selected_triangle==16777215) Selected_triangle=-1;

  glUseProgram(0);
  glBindVertexArray(0);

  glDeleteTextures(1,&Color_texture);
  glDeleteTextures(1,&Depth_texture);
  glDeleteFramebuffers(1,&FBO);

  // the normal framebuffer take the control of drawing
  glBindFramebuffer(GL_DRAW_FRAMEBUFFER,defaultFramebufferObject());
}
```

Como se puede apreciar, es una versión simplificada de la función que dibuja, y que usamos otro programa, aunque los datos del modelo son los mismos. La mayor diferencia es que realiza la lectura de un píxel en la posición indicada mediante la función glReadPixels. Una vez que hemos realizado la lectura del color hay que realizar la conversión del mismo a un entero. Llamará la atención las posiciones que ocupan las componentes RGB. Para entenderlo hay que tener en cuenta si la arquitectura de la máquina es *big-endian* o *little-endian*. El resultado final es el identificador del objeto o -1 si no hemos pinchado en ningún objeto o si lo hemos hecho sobre el fondo (color blanco, valor 16777215).

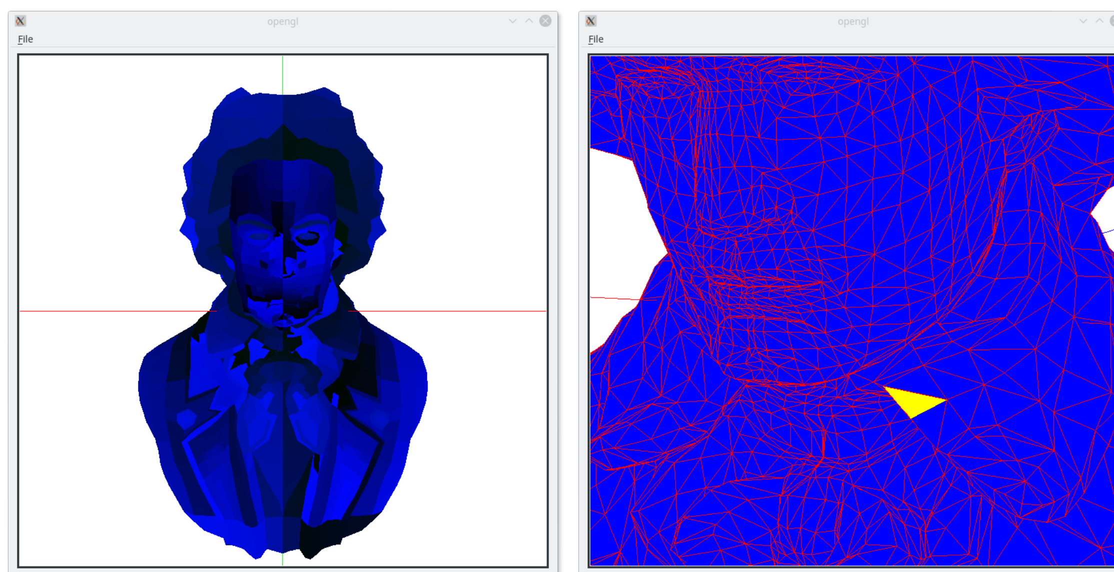
Hay que fijarse que la función pick es independiente de la visualización: la llamamos y nos devuelve el objeto seleccionado. Para hacer la selección vamos a utilizar el botón derecho del ratón. Esto permite seguir haciendo movimientos de la cámara sin interferir. Para ello cambiamos los eventos de ratón. El que controla cuando se aprieta una tecla:

```
void _gl_widget::mousePressEvent(QMouseEvent *Event)
{
  if (Event->buttons() & Qt::LeftButton) {
    Change_position=true;
    Initial_position_x=Event->x();
    Initial_position_y=Event->y();
    }
  else{
    if (Event->buttons() & Qt::RightButton) {
      Selection_position_x=Event->x();
      Selection_position_y=height()-Event->y();
    }
  }
}
```

y cuando se suelta la tecla:

```
void _gl_widget::mouseReleaseEvent(QMouseEvent *Event)
{
  Change_position=false;
  if (Event->button() & Qt::RightButton){
    pick();
```

Domingo Martín Perandrés©



(a) Asignación de colores por identifcador

(b) Visualización de la selección

Figura 18.2: Resultados del ejemplo 18

```
    update();
  }
}
```

Estas dos funciones hacen que cuando se apriete la tecla derecha del ratón se guarde la posición del cursor, invirtiendo la coordenada *y*, y cuando se suelta la tecla derecha se realiza el pick y se redibuja.

Para que la función de dibujado incluya la posibilidad de mostrar la selección sólo falta incluir una variable booleana que controle la visualización del objeto seleccionado, y también una variable que indique la posición del triángulo seleccionado. Cuando esté active la visualización del triangulo seleccionado y el identificador sea distinto de -1, se dibujará el triángulo de color amarillo. Las modificaciones en el programa principal y en los shaders son muy sencillas.