Programación de Videojuegos con OpenGL®

Segunda Edición

Luke Benstead
con Dave Astle y Kevin Hawkins

PROGRAMACIÓN DE VIDEOJUEGOS CON OPEN GL

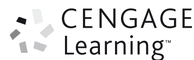
SEGUNDA EDICIÓN

LUKE BENSTEAD

CON
DAVE ASTLE Y KEVIN HAWKINS

TRADUCCIÓN:
MARÍA LORENA CAMPA ROJAS
TRADUCTORA PROFESIONAL

REVISIÓN TÉCNICA:
HÉCTOR MANUEL GÓMEZ GUTIÉRREZ



Australia • Brasil • Corea • España • Estados Unidos • Japón • México • Reino Unido • Singapur

Programación de videojuegos con OPENGL
Segunda edición.
Luke Benstead con Dave Astle
y Kevin Hawkins

Director de producto y desarrollo Latinoamérica:
Daniel Oti Yvonnet

Director editorial y de producción Latinoamérica:
Raúl D. Zendejas Espejel

Editor:
Pablo Miguel Guerrero Rosas

Coordinadora de producción editorial:
Abril Vega Orozco

Editor de producción:
Omar A. Ramírez Rosas

Coordinador de manufactura:
Rafael Pérez González

Diseño de portada:
Mike Tanamachi

Composición tipográfica y diseño de interiores:
Patricia Delgado Trujillo

© D.R. 2011 por Cengage Learning Editores, S.A. de C.V.,
una Compañía de Cengage Learning, Inc.
Corporativo Santa Fe
Av. Santa Fe 505, piso 12,
Col. Cruz Manca, Santa Fe,
C.P. 05349, México, D.F.
© Cengage Learning es una marca registrada
usada bajo permiso.

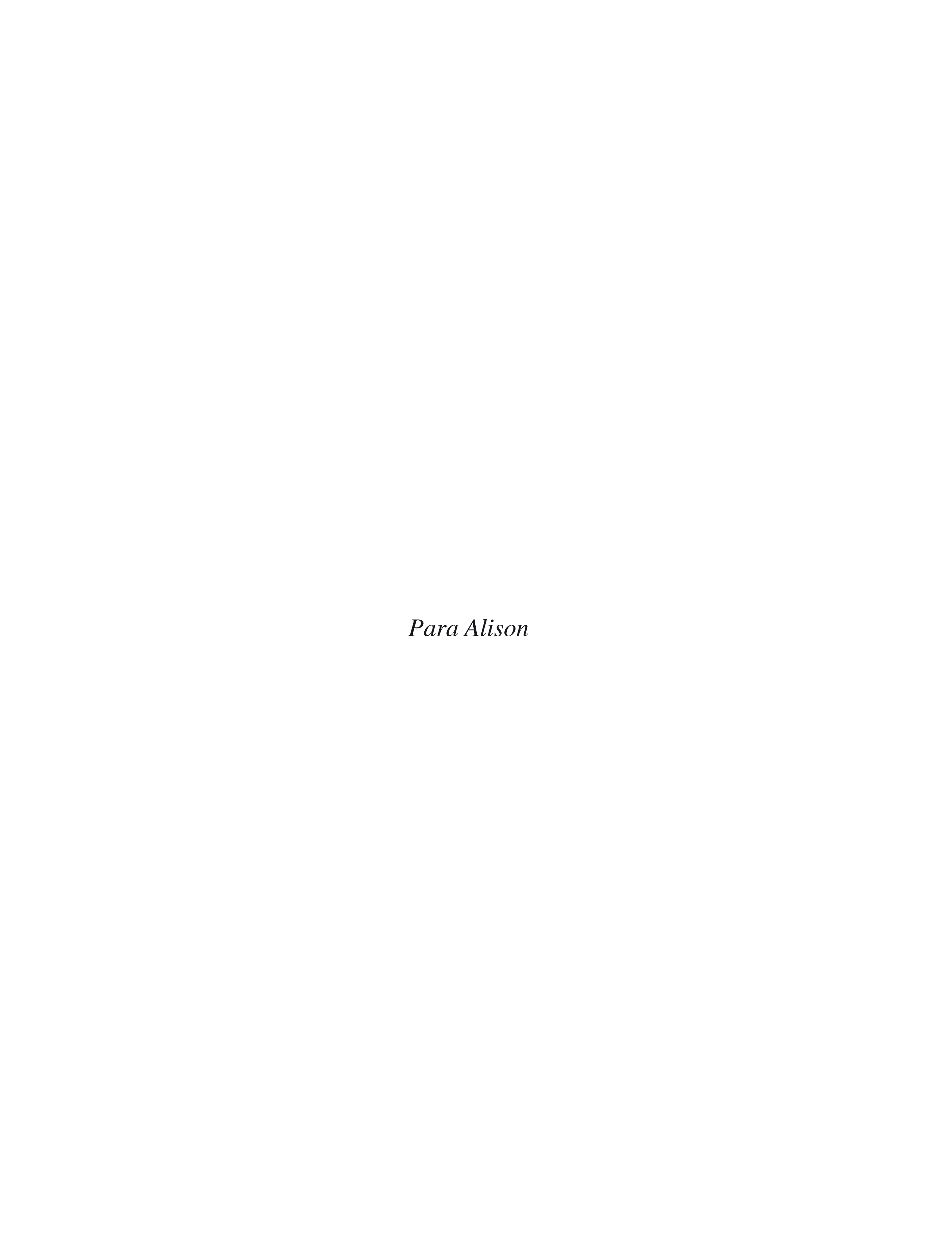
DERECHOS RESERVADOS. Ninguna parte de este trabajo, amparado por la Ley Federal del Derecho de Autor, podrá ser reproducida, transmitida, almacenada o utilizada en cualquier forma o por cualquier medio, ya sea gráfico, electrónico o mecánico, incluyendo, pero sin limitarse a lo siguiente; fotocopiado, reproducción, escaneo, digitalización, grabación en audio, distribución en internet, distribución en redes de información o almacenamiento y recopilación en sistemas de información, a excepción de lo permitido en el Capítulo III, Artículo 27 de la Ley Federal del Derecho de Autor, sin el consentimiento por escrito de la Editorial.

Traducido del libro *Beginning OPENGL game programming. Second edition.*
Luke Benstead with Dave Astle and Kevin Hawkins
Publicado en inglés por Course Technology,
una compañia de Cengage Learning © 2009
ISBN: 978-1-59863-528-7

Datos para catalogación bibliográfica:
Benstead Luke , *et.al.*
Programación de videojuegos con OPENGL
Segunda edición.
ISBN-13: 978-607-481-762-1
ISBN-10: 607-481-762-6

Visite nuestro sitio en:
http://latinoamerica.cengage.com

Impreso en México
1 2 3 4 5 6 7 14 13 12 11



Para Alison

SOBRE LOS AUTORES

Luke Benstead es coadministrador de http://nehe.gamedev.net/ y ha sido programador en OpenGL y C++ durante 7 años. En la actualidad trabaja como desarrollador de software en Londres, Inglaterra. Tiene una licenciatura en Programación Multimedia por la Universidad de Portsmouth.

Kevin Hawkins recibió el grado de licenciatura en Ciencias de la Computación y el grado de maestría en Ingeniería de Software por la Universidad Embry-Riddle. Actualmente, es el Director de Ingeniería de Software en Raydon Corporation. Junto con Dave, Kevin es cofundador de GameDev.net y coautor de la primera edición de Beggining OpenGL Game Programming [Programación de juegos con OpenGL para principantes], y More OpenGL Game Programming [Más programación de juegos con OpenGL].

Dave Astle ha estado involucrado en el mundo del desarrollo de juegos durante más de una década. En la actualidad, forma parte del equipo de ingenieros, y es desarrollador de tecnología, en Advanced Content Group de QUALCOMM, Inc. Fue cofundador de GameDev.net, donde actualmente se desempeña como presidente y director ejecutivo de la compañía. Es coautor de la primera edición de Programación de juegos con OpenGL para principantes, OpenGL Game Programming, More OpenGL Game Programming, y OpenGL ES Game Development; contribuyó en varios otros libros sobre desarrollo de juegos y frecuentemente dicta conferencias en la industria, entre ellas la Conferencia de Desarrolladores de Juegos. Tiene una licenciatura en Ciencias de la Computación por la Universidad de Utah.

AGRADECIMIENTOS

En primer lugar, quisiera agradecer a mi novia Alison, quien me apoyó mientras escribía este libro y a lo largo del proceso me sirvió una cantidad infinita de tazas de té. Prometo que ahora pasaré menos tiempo frente a la computadora aunque sea por una temporada.

También me gustaría agradecer a Carsten Haubold por ser un excelente editor técnico, en especial por su colaboración en las aplicaciones de muestra; sin su ayuda no se verían tan bien, no serían tan estables o no serían tan numerosas. Ha sido formidable trabajar contigo, Carsten. Gracias también a Jenny, Heather, Brandon y a todos los que han participado en la elaboración de este libro; ¡todos son fabulosos!

Jeff Molofee merece una mención especial. Si no hubiera iniciado el sitio web NeHe, nunca me hubiera interesado en OpenGL ni en la programación en general.

Me gustaría agradecer a mi familia: Gayna y Nigel, Stephen y Terry, Josh, Lee, Abigail y George ¡y a todos los demás que la falta de espacio no me permite mencionar! Por último, quisiera dar las gracias a mis amigos: Sean, Jayne, Rob, Hayley, Natalie y Wayne. Gracias por las muy merecidas distracciones.

CONTENIDO

Prefacio xvii

Introducción xix

PARTE 1

OpenGL básico 1

Capítulo 1

La exploración comienza... de nuevo 3

¿Por qué hacer juegos? 3
- El mundo de los juegos en 3D 4
- Los elementos de un juego 4

¿Qué es OpenGL? 7
- Historia de OpenGL 7
- Arquitectura de OpenGL 8
- Función fija contra Programabilidad 8
- El modelo de obsolescencia 9
- Características obsoletas en este libro 10
- Bibliotecas relacionadas 10
- Un vistazo rápido 11

Resumen 14
Lo que se aprendió 14
Preguntas de repaso 15
Por su cuenta 15

Capítulo 2

Creación de una aplicación sencilla de OpenGL 17

Acerca de la plataforma 17

Introducción a WGL 18
- El contexto de render 18

Formatos de pixel 23
- nSize 24
- dwFlags 25
- iPixelType 25
- cColorBits 25
- Configuración del formato de pixel 25

Una aplicación de OpenGL 26
- OpenGL en pantalla completa 36
- La clase Example 37

Resumen 42
Lo que se aprendió 42
Preguntas de repaso 42
Por su cuenta 42

Capítulo 3

Estados y primitivas de OpenGL 43

Funciones de estado 43
Consulta de estados numéricos 44
Activación y desactivación de estados 44
- glIsEnabled() 45
- Consulta de valores de cadena 45
- glGetStringi() 46
- Localización de errores 46

Colores en OpenGL 47
Manejo de primitivas 49
- Modo inmediato (Immediate Mode) 49
- Vertex arrays 52
- Vertex buffer objects 60
- Dibujo de puntos en 3D 64
- Dibujo de líneas en 3D 66
- Dibujo de triángulos en 3D 69

Resumen 72
Lo que se aprendió 72
Preguntas de repaso 73
Por su cuenta 73

Capítulo 4

Transformaciones y matrices 75

Cómo entender las transformaciones de coordenadas 75
- Coordenadas de visualización 77
- Transformaciones de visualización 78
- Transformaciones de modelado 79
- Transformaciones de proyección 81

Transformaciones de ventana de visualización 82
Funciones fijas de OpenGL y matrices 82
Matriz modelview 82
Traslación 83
Rotación 86
Escalas 90
Pilas de matrices 92
Ejemplo del robot 95
Proyecciones 98
Ortográfica 98
Perspectiva 99
Configuración de la ventana de visualización 100
Ejemplo de proyección 101
Manipulación del punto de visualización 103
Uso de gluLookAt() 103
Uso de glRotate() y glTranslate() 104
Creación de sus propias rutinas personalizadas 105
Uso de sus propias matrices 106
Carga de su matriz 106
Multiplicación de matrices 107
Resumen 108
Lo que se aprendió 108
Preguntas de repaso 109
Por su cuenta 109

Capítulo 5

Extensiones de OpenGL 111

¿Qué es una extensión? 111
Nomenclatura de la extensión 112
Cadenas de nombre 112
Funciones y fichas 113
Obtención de un punto de entrada a las funciones 114
Extensiones en Windows 115
Búsqueda de extensiones compatibles 115
Extensiones de WGL 118
Definición de fichas 118
Introducción a GLee 118
Configuración de GLee 119
Uso de GLee 119
Uso de GLee con las extensiones básicas 120
Extensiones en acción 121
Resumen 121
Lo que se aprendió 121
Preguntas de repaso 121
Por su cuenta 122

Capítulo 6

Cambiando a un diagrama (pipeline) programable 123

El futuro de OpenGL 123
¿Qué es GLSL? 125
Vertex shaders 125
Fragment shaders 126
El lenguaje GLSL 127
Estructura del shader 127
Pre-procesador 128
Variables 128
Entradas del shader 135
Declaraciones 136
Constructores 136
Reducción 138
Definición de funciones 139
Funciones integradas 139
Funciones obsoletas de GLSL 140
Uso de shaders 141
Creación de objetos en GLSL 142
Consulta de los registros de información 146
Envío de datos al shader 147
La clase GLSLProgram 150
Sustitución del diagrama de función fija 150
Manejo de sus propias matrices .. 153
La Biblioteca Kazmath 153
Ejemplo modificado del robot 153
Resumen 154
Lo que se aprendió 154
Preguntas de repaso 154
Por su cuenta 155

Capítulo 7

Mapeo de texturas 157

Una perspectiva del mapeo de texturas 157
Uso del mapa de textura 158
Objetos de textura 159
Creación de objetos de textura 159
Eliminación de objetos de textura 160
Especificación de texturas 160
Texturas en 2D 161
Texturas en 1D 165
Texturas en 3D 165
Texturas de mapa de cubos 166
Filtrado de texturas 166
Coordenadas de textura 168
Aplicación de las coordenadas de textura 169
Parámetros de textura 171
Modos de envoltura de textura 171
Mipmaps 175
Mipmaps y la biblioteca de utilidades de OpenGL 176
Carga de archivos de imagen Targa 177
El formato de archivo Targa 177
La clase TargaImage 178
Resumen 181
Lo que se aprendió 181
Preguntas de repaso 181
Por su cuenta 181

PARTE 2

Más allá de lo básico 183

Capítulo 8

Iluminación, mezclado y niebla 185

Iluminación 185
Normales 187
El modelo de iluminación 189
Materiales 190
Iluminación en GLSL 194
Mezclado 202
Funciones de mezclado separadas 206
Color constante de mezclado 206
Niebla 207
Ejemplo de niebla 208
Resumen 209
Lo que se aprendió 209
Preguntas de repaso 209
Por su cuenta 209

Capítulo 9

Más sobre el mapeo de texturas .. 211

Subimágenes 211
Copiado desde el búfer de color... 212
Mapeo del entorno 213
Mapeo de esfera 213
Mapeo reflectante de cubos 214
Pruebas alfa 215
Texturizado múltiple 217
Unidades de textura 218
Texturizado múltiple en GLSL 218
Resumen 220
Lo que se aprendió 220
Preguntas de repaso 220
Por su cuenta 220

Capítulo 10

Mejora del rendimiento 221

Frustum culling 221
Ecuación del plano 222
Definición del frustum 223
Prueba de un punto 225
Prueba de una esfera 225
Aplicación del frustum culling 226
Resumen 227
Lo que se aprendió 227
Preguntas de repaso 227
Por su cuenta 227

Capítulo 11

Despliegue de texto 229

Fuentes asignadas por textura en 2D 229
- Generación de las coordenadas de textura 230
- Ejemplo de fuentes asignadas por textura 231

Fuentes en 2D con FreeType 231
- La biblioteca FreeType 232
- Inicialización de FreeType y carga de una fuente 233
- Configuración del tamaño de fuente 234
- Generación de texturas de glifo 234
- Liberación de recursos en FreeType 236
- Ejemplo de FreeType 236

Una nota sobre fuentes en 3D 237
Resumen 238
Lo que se aprendió 238
Preguntas de repaso 238
Por su cuenta 238

Capítulo 12

Búferes de OpenGL 239

¿Qué es un búfer de OpenGL? 239
- Limpieza de búferes 240

Prueba de tijera 240
Los búferes de color 241
- Enmascaramiento de color 241
- Establecimiento del color de limpieza 242

El búfer de profundidad 242
- Control de la prueba de profundidad 242
- Desactivación de la escritura en el búfer de profundidad 243
- Problemas potenciales 244

Búfer de plantilla 245
Resumen 249
Lo que se aprendió 249
Preguntas de repaso 250
Por su cuenta 250

Capítulo 13

El juego final 251

El formato del modelo MD2 251
- El encabezado de MD2 252
- Carga de los datos del modelo 253
- Animación del modelo MD2 255
- Render del modelo 257

Creación de explosiones 258
Puntos Sprite 259
- Uso de puntos Sprite 260

Ogro Invasion! 261
Una nota sobre la detección de colisiones 264
Resumen 264
Preguntas de repaso 265
Por su cuenta 265

Apéndice A

Respuestas a las preguntas de repaso y ejercicios 267

Apéndice B

Lecturas adicionales 275

Apéndice C

Contenido del cd 279

Índice 281

PREFACIO

El libro que tiene en sus manos cuenta con mucha historia detrás. En 2001 Dave Astle y Kevin Hawkins, cofundadores de GameDev.net, escribieron *OpenGL Game Programming* —un excelente libro que abarca OpenGL 1.2 y que contiene más de 780 páginas. Incluye una amplia gama de temas, desde superficies curvas hasta la física de los juegos y desde la simulación de sombras hasta la generación de sonido usando DirectX API. Fue el primer libro de OpenGL que compré, y mi ejemplar ha sido usado y consultado tantas veces que su cubierta se mantiene unida ¡con cinta adhesiva! En aquel momento, éste era el libro que debías comprar si querías aprender OpenGL.

Para 2004 ya se había lanzado OpenGL 1.5 y la dinámica industria de los gráficos había avanzado. Kevin y Dave unieron fuerzas una vez más, no sólo para actualizar el libro, sino también para ampliarlo a fin de cubrir características nuevas y más avanzadas. Se tomó la decisión de crear dos volúmenes. El primero consistió en una versión revisada de la parte central del libro, se eliminó cierto material y se creó la primera edición de Beggining OpenGL Game Programming [Programación de juegos con OpenGL para principiantes]; mientras que los temas más avanzados se conviertieron en el segundo volumen: More OpenGL Game Programming.

A finales de 2007, estaba próxima la emisión de la segunda edición de Programación de juegos con OpenGL para principiantes. En ese momento, OpenGL 2.1 era la versión más reciente, pero se había anunciado el inminente lanzamiento de OpenGL 3.0. Los cambios originales propuestos para OpenGL 3.0 hubieran hecho que cualquier libro sobre OpenGL 2.1 se volviera obsoleto

de inmediato, así que se tomó la decisión de esperar. Finalmente, OpenGL 3.0 fue lanzado en agosto de 2008 y la producción de este libro se inició poco tiempo después.

Espero que disfruten esta segunda edición de *Programación de juegos con OpenGL para principiantes y* ¡que empiece el aprendizaje!

—Luke Benstead

INTRODUCCIÓN

Cambios a la primera edición

En general, la idea de esta edición consiste en enseñar cómo hacer render de *forma rápida* y a prueba de cambios futuros. Los métodos tradicionales de render con OpenGL que el lector debe conocer, como el modo inmediato, las arreglos de vértices o vertex arrays y los despliegues de listas que están predestinados a ser eliminados (por obsoletos) en una futura versión de OpenGL. Estos métodos aún se estudian de manera breve en la presente edición, pero sólo como un trampolín de inicio para después pasar a métodos de render más rápidos y un poco más complejos, con vertex buffer objects y el lenguaje de shaders (sombreado) de OpenGL (OpenGL Shading Language, GLSL).

El cambio más importante a la primera edición es probablemente la inclusión de GLSL, que no aparecía en absoluto en ella. En la actualidad, los lenguajes de shaders son muy comunes y hacer el render sin ellos (usando el diagrama de función fija) se considera obsoleta en la especificación de OpenGL 3.0. Por desgracia, esto hace que la curva de aprendizaje sea mucho más empinada de lo que solía ser, pero, por lo general resulta una gran idea aprender las buenas prácticas desde el principio.

En esta edición ya no se cubren los siguientes temas (o se estudian de manera breve) por considerarse obsoletos:

- Patrones de punteado (Stipple patterns)

- Cuadrantes y polígonos
- Color secundario
- Texturas residentes y prioridad de la textura
- La pila de matrices de textura
- Generación de coordenadas de textura
- Combinaciones de texturas
- Listas de despliegue
- El búfer de acumulación
- Contorno de fuentes y fuentes de mapa de bits usando "wgl"
- Pruebas Alpha
- Niebla en OpenGL

En la presente edición se incluyen los siguientes temas nuevos:

- El lenguaje de shaders de OpenGL (GLSL 1.30)
- El modelo de obsolescencia
- Modelo de carga y animación MD2
- Sprites de punto
- Fuentes usando FreeType
- Creación de contexto en OpenGL 3.0
- Vertex buffer objects (Objetos del búfer de vértice)
- Pruebas Alpha con GLSL
- Niebla con GLSL

¿Quién debe leer este libro?

Este libro está dirigido a profesionales que se iniciaron recientemente en la programación 3D o que están en proceso de migrar desde otro API 3D (como Direct3D) hacia OpenGL. El lector debe tener alguna experiencia programando en C++ y al menos una comprensión básica de los gráficos y las matemáticas en 3D. Después de leer el libro, usted deberá ser capaz de aplicar sus nuevos conocimientos en OpenGL para crear sus propios juegos.

¿Qué se incluye y qué no?

El enfoque de este libro es que usted comience a programar gráficos en 3D usando OpenGL. Para que libro se mantenga conciso, se ha asumido que el lector cuenta con ciertos conocimientos básicos.

El primer supuesto es que usted sabe programar C++ en la plataforma de su elección. C++ es un lenguaje masivo cuyo dominio implica años de práctica; por lo tanto, no se espera que usted sea un experto, pero debe tener un conocimiento básico de los siguientes temas:

- Compilar programas y vincularlos con bibliotecas externas

- Clases, herencia y funciones virtuales

- Arreglos y punteros
- Los contenedores de biblioteca con plantilla estándar, STL (vectores, listas, etcétera).

En el Apéndice B, “Lecturas complementarias”, se presenta una lista de excelentes referencias sobre C++. Incluso si se tiene un buen conocimiento de C++, vale la pena revisar este material.

El segundo supuesto es que usted comprende en cierto nivel las matemáticas en 3D. En este libro sólo se cubren de manera muy breve las matemáticas en 3D en relación con OpenGL. No hace mucho tiempo, era posible usar OpenGL durante un largo periodo antes de tener necesidad de conocimientos sólidos en matemáticas en 3D. Sin embargo, con el cambio que implica render basado en shaders, desde un principio se requiere por lo menos una comprensión básica de las matrices.

Por último, en este libro sólo se tratarán temas de desarrollo de juegos en relación directa con los gráficos. Para la mayoría de los juegos se requieren temas como la arquitectura, la física y el audio de los juegos, así como la inteligencia artificial, pero son temas tan amplios que ¡todos merecen que se les dedique un libro entero!

Acerca de la plataforma base

La ventaja principal de OpenGL sobre otras APIs gráficas es que funciona en muchas plataformas. A pesar de la API de OpenGL funciona en múltiples plataformas, el código fuente necesario para crear una ventana de OpenGL accesible y para manejar entradas y eventos del sistema es algo muy específico de una plataforma a otra. Planear la creación de un código para cada plataforma sería un objetivo poco alcanzable. Por esta razón, se tomó la decisión de enfocarse en el sistema operativo más comúnmente utilizado, Microsoft Windows. Pero, para mostrar lo fácil que es transferir el código OpenGL a otro sistema operativo, en el CD se incluyen versiones de todos los ejemplos escritos para GNU/Linux. Las versiones para Linux del código fuente se escribieron, probaron y elaboraron bajo Ubuntu 8.10. Las versiones para Windows del código fuente se probaron con Windows Vista y Windows XP.

Los usuarios de Linux y otros sistemas operativos alternativos (como OSX), estarán complacidos de saber que la mayor parte del libro se aplica a todas las plataformas; la excepción a esta regla es el capítulo 2, “Creación de una aplicación simple de OpenGL,” que se enfoca en la plataforma Microsoft Windows.

OpenGL 2.1 y OpenGL 3.0

Este libro se dirige principalmente a OpenGL 3.0, puesto que es la versión más reciente del programa. OpenGL 3.0 difiere de las anteriores versiones en que establece un nivel mínimo de apoyo en la tarjeta de gráficos para crear un contexto. Por esta razón, el texto supone que tanto el hardware como los controladores gráficos son compatibles con OpenGL 3.0.

OpenGL 3.0 todavía puede considerarse una versión muy nueva y, en el momento de escribir este libro, no todos los fabricantes de tarjetas gráficas habían lanzado controladores completamente compatibles con OpenGL 3.0. Sería una lástima que muchas personas no pudieran

utilizar el código fuente del libro por estar en espera de que sus fabricantes lanzaran nuevos controladores. Así que, en el CD, se incluyen dos versiones de código para cada plataforma; una versión está diseñada para OpenGL 3.0 (y su correspondiente lenguaje de shaders GLSL 1.30) y la otra para OpenGL 2.1 (y GLSL 1.20).

Las diferencias entre estas dos versiones del código son mínimas:

- Capítulos 1-4—El código fuente es el mismo para ambas versiones, excepto por la creación de contexto en OpenGL, que baja al contexto de OpenGL 2.1 para la versión 2.1.
- Capítulo 5—El código es el mismo a excepción del ejemplo de extensiones manuales, que utiliza `glGetString()` para 2.1 en vez de `glGetStringi()`, que sólo es compatible con OpenGL 3.0.
- Capítulos 6-12—El código fuente en C++ es el mismo, pero difieren en los shaders GLSL.
- Capítulo 13—El único código fuente para este capítulo es el juego final. Hay una sola versión del juego que baja a OpenGL 2.1 y GLSL 1.20 si no es compatible con OpenGL 3.0.

Por supuesto, esto todavía supone un controlador de gráficos para OpenGL 2.1. Si tiene problemas para ejecutar cualesquiera de las muestras de código, es probable que el problema se resuelva al actualizar los controladores de gráficos.

Uso del libro

El CD

El CD contiene el código fuente para todas las aplicaciones de muestra que acompañan al libro. Es deseable tener acceso a estos archivos de código fuente para utilizarlos en conjunto con el texto.

Extensiones

Las extensiones se analizan a detalle en el capítulo 5, “Extensiones de OpenGL.” Las extensiones son necesarias para tener acceso en la plataforma de Windows a todas las características de OpenGL en las versiones superiores a 1.1. En vez de enumerar todas las extensiones requeridas, se asume que el controlador al menos es compatible para la funcionalidad básica de OpenGL 2.1.

Lenguaje y Herramientas

Para compilar los ejemplos incluidos en el CD, primero es necesario adquirir un compilador IDE para C++. La versión para Windows del código fuente se compila utilizando el Visual C++ 2008 Express Edition de uso libre, mientras que la versión para el código GNU/Linux se compila usando Code::Blocks y el compilador GNU G++ Compiler.

El CD incluye el IDE Code::Blocks para las plataformas Windows, Linux y Mac OSX, con los cuales podrá iniciar. Si está usando Code::Blocks en la plataforma Windows, una función de importación de Visual C++ Project convertirá el proyecto de Visual C++ a Code::Blocks.

Encabezados y bibliotecas

Al compilar aplicaciones de OpenGL, es necesario tener varias bibliotecas vinculadas y diversos archivos de encabezado incluidos. Los archivos de encabezado se almacenan en forma convencional en un directorio de inclusión llamado GL. Los siguientes archivos de encabezado pueden incluirse en un proyecto, dependiendo de la plataforma y características requeridas:

- **gl.h** Éste es el archivo de encabezado principal que define la mayor parte de las funciones de OpenGL.
- **glu.h** Es el encabezado de la biblioteca de herramientas de OpenGL (OpenGL Utility Library).
- **glext.h** Es el archivo de encabezado de las extensiones OpenGL. Este archivo de encabezado se actualiza de manera regular y está disponible en opengl.org. Incluye las constantes y las definiciones para las extensiones OpenGL más recientes.
- **wglext.h** Es el archivo de encabezado de las extensiones Windows. Similar a glext.h pero para extensiones exclusivas de Windows.
- **glxext.h** Es el archivo de encabezado de las extensiones GLX; contiene constantes para las extensiones de GLX.

Todas las aplicaciones de OpenGL deben vincularse al menos con `opengl32.lib` en Windows, o `libGL.a` en Linux. Si la aplicación hace uso de la biblioteca OpenGL Utility, entonces también deben vincularse a glu32.lib (en Windows) o `libGLU.a` (en Linux).

Uso de C++

En todo el código fuente, se ha hecho uso de un subconjunto limitado de la Standard Template Library (STL). El uso de recipientes y algoritmos de la STL es una buena práctica y su utilización hace que el código sea más conciso, más seguro, más simple y normalmente más rápido. Si usted no tiene ningún conocimiento de la STL, revise los recursos de C++ en el Apéndice B. En el código fuente, se han utilizado los siguientes componentes de la STL:

- `std::vector` —Un arreglo dinámico de tamaño variable. El vector maneja su propia memoria y almacena sus elementos de manera consecutiva en la memoria, del mismo modo que un arreglo C-style. Por esta razón, los vectores pueden transferirse a funciones de C (por ejemplo, OpenGL) al pasar un puntero al primer elemento (por ejemplo, &myArray[0]).

- `std::string` —Una clase de cadena. `string` sustituye casi por completo el uso de arreglos de caracteres. La clase `string` tiene muchos métodos integrados útiles. Las cadenas pueden transferirse a funciones de C utilizando el método `c_str()`, que devuelve una const char*.

- `std::ifstream` —Una vía para la entrada a archivos. `ifstream` se utiliza en el libro para leer datos de los archivos. Se usa para cargar shaders desde archivos de texto, texturas desde imágenes TGA y modelos desde archivos MD2. `ifstream` sustituye al C FILE y a sus funciones asociadas.

- `std::map` —Un recipiente asociativo. Un `map` almacena un conjunto ordenado de los principales pares de valores.>

- `std::list` —Un recipiente que se comporta como una lista doblemente enlazada. Este recipiente sólo se utiliza en el juego final para almacenar una lista de entidades.

Todas estas clases se explican a detalle en cualquier libro de consulta sobre C++, y en las referencias listadas en el Apéndice B.

Sitio web de apoyo

El sitio web que acompaña al libro se puede encontrar en http://glbook.gamedev.net/. Ahí se publicarán las actualizaciones del programa y las erratas detectadas, según sea necesario. Por favor, revise este sitio si tiene algún problema.

PARTE 1

Open GL básico

La exploración comienza ... de nuevo

Antes de profundizar en el desarrollo de juegos es necesario que usted tenga una comprensión básica del medio en el que trabajará. Utilizará la API de OpenGL para gráficos, por lo que revisaremos el origen, el diseño y la evolución de OpenGL. También proporcionaremos una visión general de la industria de los videojuegos, así como un vistazo a los elementos básicos que participan en un juego.

En este capítulo, usted aprenderá:

- Qué es un juego
- Acerca de OpenGL y su historia
- Sobre el futuro de OpenGL
- Acerca de las bibliotecas que se pueden utilizar para ampliar la funcionalidad de OpenGL

¿Por qué hacer juegos?

En la última década, el entretenimiento interactivo ha crecido a pasos agigantados. Los juegos de computadora, que solían constituir un nicho de mercado, ahora han crecido para convertirse en una industria multimillonaria. En los últimos años se ha visto que este crecimiento tiende a acelerarse y aún no se vislumbra el final. La industria del entretenimiento interactivo es un mercado explosivo que impulsa las tecnologías informáticas más recientes hasta el límite y que ayuda a conducir la investigación en áreas como los gráficos y la inteligencia artificial. Este

implacable impulso y crecimiento atrae a muchas personas hacia la industria, pero en realidad ¿por qué la gente crea juegos?

Hay miles de personas en todo el mundo que están aprendiendo a escribir juegos, y cada uno de ellos es impulsado por una sola cosa: la diversión. El desarrollo de juegos reúne muchas habilidades diferentes, ésta es la razón por la que resulta tan atractivo para tantas y tan diversas personas. ¡Los artistas y músicos pueden aplicar sus talentos creativos y los programadores pueden utilizar sus habilidades para resolver problemas!

El mundo de los juegos en 3D

Aunque muchas compañías han contribuido al crecimiento de los juegos en 3D, debe hacerse mención especial a id Software, que fue un catalizador importante en el surgimiento de este tipo de juegos. Hace más de 15 años, John Carmack y compañía lanzaron internacionalmente un pequeño juego llamado *Wolfenstein 3D*. *Wolf3D* arrodilló al mundo de los juegos al presentar gráficos tridimensionales basados en ray-casting de gráficos en 3D en tiempo real y un mundo inmersivo que dejó a los jugadores sentados frente a sus computadoras durante horas y horas. El juego fue un nuevo comienzo para la industria, y nunca volvió atrás. En 1993, el mundo de *Doom* causó un gran alboroto e impulsó a la tecnología de gráficos 3D mas allá de sus límites con su motor 2.5D. El mundo de los juegos se deleitaba en la realización técnica interpuesta por id en su juego *Doom*, pero no se detuvo ahí. Varios años más tarde, *Quake* mejoró los juegos en 3D para siempre. Ya no se tenían enemigos en 3D "falsos", sino que había muchas entidades tridimensionales que podían moverse en un espacio 3D totalmente poligonal. Ahora, las posibilidades estaban limitadas sólo por la cantidad de polígonos que el CPU (y después el GPU) podía procesar y visualizar en pantalla. *Quake* también puso en práctica los juegos con múltiples jugadores en red, con lo que las hordas de usuarios de internet se unieron a la diversión que proporcionan las luchas a muerte contra otras 30 personas.

Desde el lanzamiento de *Quake*, la industria se ha beneficiado con los nuevos avances tecnológicos que ocurren casi todos los meses. El sector de los juegos en 3D ha creado un hardware acelerador de 3D que realiza las matemáticas tridimensionales en el mismo silicio. Ahora, cada seis meses se lanza un nuevo hardware que parece superar a su antecesor con el doble de potencia, velocidad y flexibilidad. Con todos estos avances, no podría haber un momento más emocionante que ahora para el desarrollo de juegos en 3D.

Los elementos de un juego

Ahora usted debe estarse preguntando, "¿cómo se hace un juego?" Antes de poder responder esta pregunta debe entender que los juegos, en su nivel más elemental, son software. En la actualidad, los programas de software se desarrollan en equipo y cada miembro de éste trabaja en su especialidad hasta que la labor de todos se integra para crear un producto único y coherente. Los juegos se desarrollan de manera muy similar, sólo que la programación no es la única área de especialización. Se requieren artistas que generen las imágenes y los bellos escenarios

que hoy en día prevalecen en la mayoría de los juegos. Los diseñadores de nivel le dan vida al mundo virtual y utilizan el trabajo realizado por los artistas para crear mundos más allá de lo creíble.

Los programadores ensamblan cada elemento y se aseguran de que todo funcione como un conjunto. Los técnicos de sonido y los músicos crean el audio necesario para proporcionarle al jugador una experiencia virtual rica, multimedia y creíble. Los diseñadores integran el concepto del juego y los productores coordinan el esfuerzo de todos los demás.

Como hay personas que trabajan en diferentes áreas de especialización, el juego debe dividirse en diversos elementos que se unirán al final del proceso. En general, los juegos se dividen en las siguientes áreas:

- Gráficos
- Entrada
- Música y sonido
- Lógica del juego e inteligencia artificial
- Redes
- Interfaz del usuario y sistema de menús

Cada una de estas áreas puede dividirse en sistemas aún más específicos. Por ejemplo, la lógica del juego podría consistir de física y sistemas de partículas, mientras que los gráficos podrían tener un sistema de render 2D y/o 3D. En la figura 1.1 se muestra un ejemplo simple de la arquitectura de un juego.

Como puede verse, cada elemento de un juego se divide en sus propias piezas y se comunica con los otros componentes. El elemento lógico tiende a ser el eje del juego, es donde se toman las decisiones para procesar entradas y enviar salidas. Sin embargo, la arquitectura mostrada en la figura 1.1 es muy simple, a diferencia de la figura 1.2 donde se muestra cómo podría verse la arquitectura de un juego más avanzado.

Como se puede observar en la figura 1.2, un juego más complejo requiere un diseño arquitectónico más complicado. Se desarrollan y utilizan componentes más detallados para implementar características específicas o funcionalidades que el software del juego necesita para operar sin problemas. Debe tenerse en cuenta que los juegos disponen de las combinaciones más complejas de tecnología y diseño de software, por ello el desarrollo de juegos requiere un pensamiento abstracto y que se implemente en un nivel superior al desarrollo tradicional de software. Cuando usted está desarrollando un juego, está creando una obra de arte que debe ser tratada como tal. Prepárese para intentar cosas nuevas por usted mismo y rediseñar las tecnologías existentes a fin de satisfacer sus necesidades. No existe ninguna manera establecida para desarrollar juegos, del mismo modo que no existe un método para pintar un cuadro. ¡Debemos esforzarnos por ser innovadores y establecer nuevos estándares!

Figura 1.1
Un juego se compone de varios subsistemas.

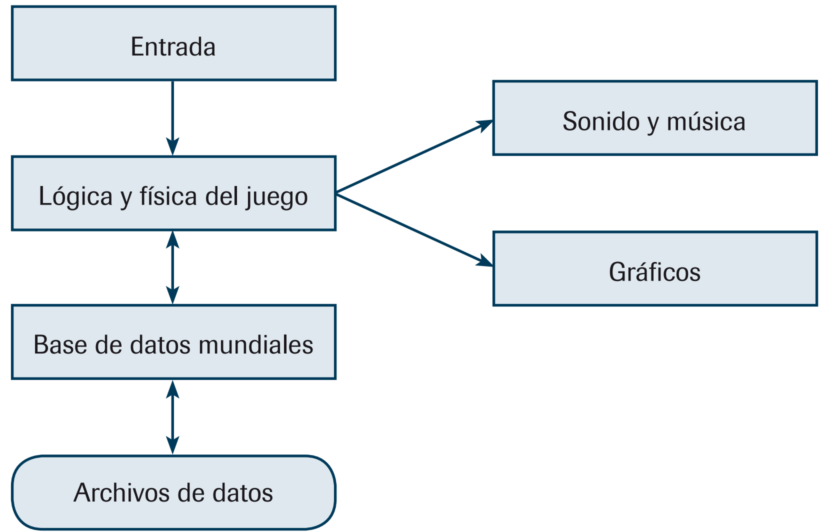
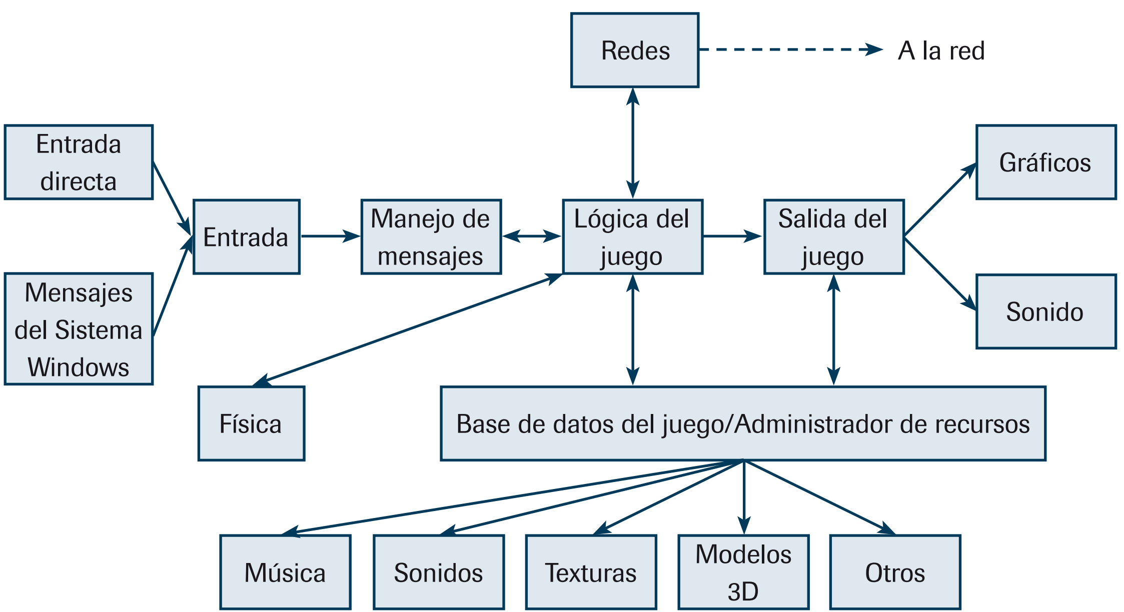


Figura 1.2
Diseño arquitectónico de un juego más avanzado.



¿Qué es OpenGL?

OpenGL es una API (Interfaz de Programación de Aplicaciones, por sus siglas en inglés para Application Programming Interface) de bajo nivel que le proporciona al programador una interfaz con el hardware de gráficos. OpenGL no proporciona una funcionalidad de alto nivel como las funciones matemáticas o una interfaz con cualquier otro hardware. OpenGL sólo se refiere a gráficos.

La principal ventaja que tiene OpenGL sobre otras API de gráficos es que se ejecuta en diferentes sistemas operativos o plataformas. OpenGL se puede ejecutar en Windows, Linux, Mac OSX y dispositivos portátiles como el proyecto abierto Pandora. Su hermano menor OpenGL ES puede ejecutarse en muchos dispositivos portátiles.

OpenGL se utiliza en muchos tipos de aplicaciones, desde programas CAD hasta juegos como *Doom 3*, y desde simulaciones científicas hasta aplicaciones de modelado en 3D.

Sugerencia

OpenGL significa "Open Graphics Library." "Open" se usa porque OpenGL es un estándar abierto, lo que implica que muchas compañías pueden contribuir a su desarrollo. No significa que OpenGL sea un código fuente abierto.

Historia de OpenGL

OpenGL fue desarrollado originalmente por Silicon Graphics, Inc. (SGI) como una API de gráficos con propósitos múltiples e independiente de la plataforma. Desde 1992, el desarrollo de OpenGL ha sido supervisado por el OpenGL Architecture Review Board (ARB), que estaba compuesto por importantes distribuidores de la industria de los gráficos y otros líderes del sector como 3DLabs, ATI, Dell, Evans & Sutherland, Hewlett-Packard, IBM, Intel, Matrox, NVIDIA, SGI, Sun Microsystems, Silicon Graphics y, a partir de 2003, Microsoft. El papel de ARB fue establecer y mantener la especificación de OpenGL, la cual indica qué características deben incluirse al desarrollar una distribución de OpenGL.

En 2006 el control de la especificación de OpenGL se transfirió al grupo Khronos, quien mantiene los estándares abiertos y está compuesto por la mayoría de los miembros originales de ARB. Esto implicó que el cambio de control hacia el nuevo grupo fuera sencillo. Por razones históricas, el grupo de trabajo de OpenGL en Khronos todavía se conoce como el ARB. Khronos ha seguido desarrollando la especificación de OpenGL y lanzó OpenGL 3.0 a finales de 2008, con la promesa de una pronta liberación de OpenGL 3.1.

Los diseñadores de OpenGL sabían que los fabricantes de hardware desearían agregar características que no estaban expuestas en las interfaces básicas de OpenGL. Para solucionar este problema, incluyeron un método para extender OpenGL. En ocasiones otros proveedores de hardware adoptan estas extensiones, y si el apoyo a una extensión es lo suficientemente amplio

—o ARB considera que la extensión es importante y necesaria— ésta puede promoverse a la especificación básica de OpenGL. Casi todas las adiciones más recientes a OpenGL comenzaron como extensiones —muchas de ellas relacionadas directamente con los videojuegos. Las extensiones se estudian a detalle en el capítulo 5, "extensiones de OpenGL."

Arquitectura de OpenGL

La arquitectura original de OpenGL se basaba en una máquina de estado interna. El programador debía usar una función de OpenGL para cambiar un estado en particular y OpenGL dibujaría usando este estado hasta que fuera cambiado de nuevo. Por ejemplo, si usted quería dibujar en rojo, debía establecer el estado de color en rojo y trazar algunos objetos, y después quizá cambiar a blanco. En la función fija de OpenGL estos estados pueden afectar la iluminación, los colores, el culling, etcétera.

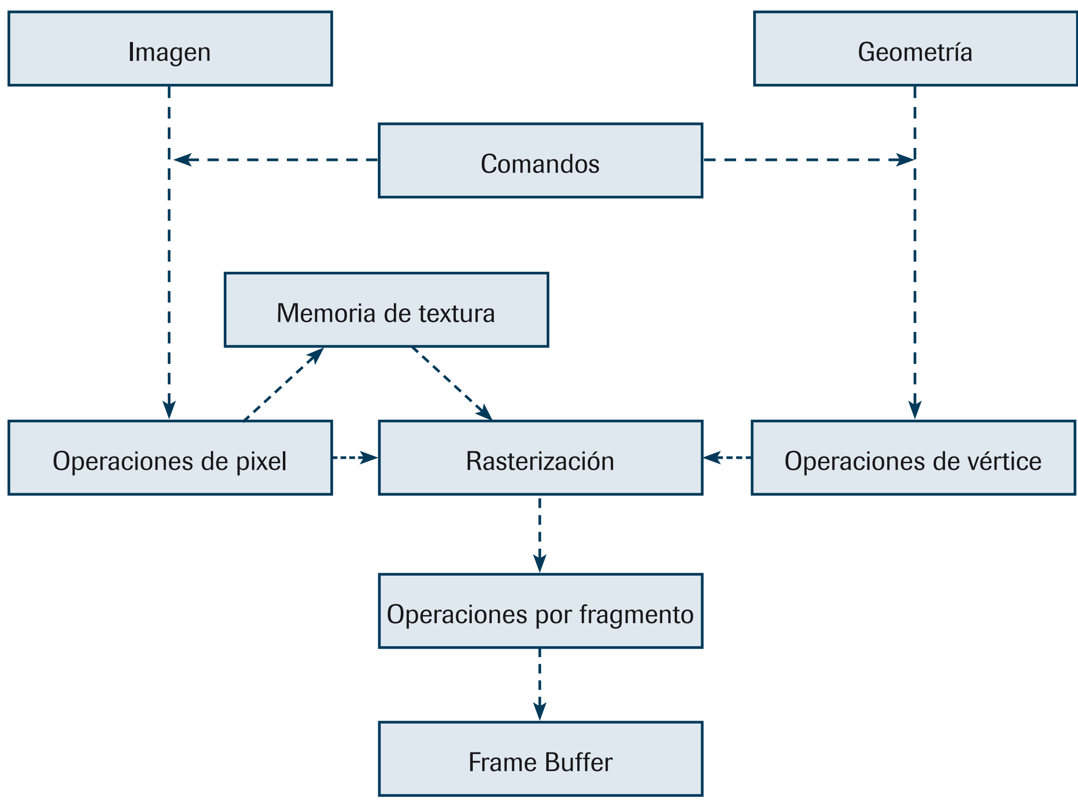
En OpenGL 3.0, empezamos a ver un cambio hacia API menos orientadas al estado. Las funciones de estado para el color, las normales, la iluminación, y otras se han vuelto obsoletas porque no tienen sentido en un diagrama programable. Cuando se usan shaders, queda a criterio del programador no sólo pasar la información correcta (por ejemplo, la del color del vértice), sino también aplicar esta información al vértice en el shader.

Función fija contra Programabilidad

Cuando se inventó OpenGL la potencia del procesamiento en computadora estaba muy lejos de ser lo que es hoy en día. Normalmente una PC tenía un solo procesador (CPU), que realizaba todo el procesamiento de sistemas y gráficos. El diagrama de función fija se diseñó para obtener el mayor provecho del hardware usando una sola ruta de render. A finales de la década de 1990 comenzaron a aparecer en el mercado las tarjetas de gráficos en 3D. Estas tarjetas contenían procesadores gráficos específicos que podían realizar el render en forma independiente al CPU principal. De manera súbita, las PC tenían el poder de dibujar escenas mucho más complejas en tiempo real. Pronto se hizo evidente para los proveedores de gráficos que la capacidad de ejecutar programas compilados personalizados en los procesadores de gráficos (GPU) proporcionaría a los programadores mucho más control, flexibilidad y poder que la utilización del modelo estándar de función fija.

En los últimos años, el uso de estos programas de shaders (o sombreado) GPU se ha adoptado como el método de render preferido. En el diagrama programable, los shaders acogen diferentes partes del proceso de render. Al momento de escribir en OpenGL, es posible proporcionar tres tipos de shaders: Los *vertex shaders*, (sombreadores o de vértice) que operan sobre cualquier vértice enviado al diagrama, los *fragment shaders*, (sombreadores de fragmento) que operan sobre cualquier pixel representado en la pantalla después del culling y, más recientemente, los *geometry shaders*, (sombreadores de geometría) que en realidad permiten al programador generar vértices en la tarjeta de gráficos. En la actualidad los shaders de geometría no son parte del OpenGL básico pero se proporcionan como extensiones específicas del proveedor. Es probable

Figura 1.3
El diagrama de render de OpenGL



que esto cambie en la próxima versión de OpenGL y los shaders de geometría se incluyan en la API básica. Los shaders de vértice y de fragmento ya forman parte del OpenGL básico.

El modelo de obsolescencia

En 2007 el grupo Khronos anunció que la API OpenGL se sometería a una limpieza profunda, que se realizaría originalmente en dos etapas. La primera, denominada "Longs Peak", destinada a reducir la API, rompiendo compatibilidades hacia atrás por primera vez en la historia de OpenGL y presentando un nuevo modelo de objetos. Un poco después, Longs Peak estaría seguida por "Mt. Evans", en esta etapa se instauraría una funcionalidad avanzada y moderna (incluyendo los shaders de geometría) en el OpenGL básico.

Por desgracia, las cosas no ocurrieron completamente de acuerdo con lo planeado. Después de un año de retrasos, Khronos anunció OpenGL 3.0. A pesar de que no contenía todo lo que Longs Peak había prometido en un inicio, contaba con una característica nueva que preparó el camino para una API nueva, limpia y esbelta: el modelo de obsolescencia.

El modelo de obsolescencia se introdujo con el fin de proporcionar un proceso para eliminar partes de la API. La eliminación de una característica de OpenGL puede seguir varias etapas. Primero, una función o ficha se marca como obsoleta. Una característica obsoleta no debe usarse en ningún código nuevo. Luego, en una versión futura, la característica obsoleta se retirará de la base o núcleo. La función retirada puede implementarse como una extensión para que el código heredado pueda seguir utilizando la característica con algunos cambios menores. Eventualmente la función dejará de ser compatible.

Cada implementación puede proporcionar un método para crear un contexto de compatibilidad futura durante la inicialización. Si se usa una característica obsoleta en esta clase de contexto se producirá un error del tipo `INVALID_OPERATION`.

Características obsoletas en este libro

En este libro no estudiaremos ninguna funcionalidad obsoleta, excepto cuando contribuya al aprendizaje. Cualquier funcionalidad obsoleta que se mencione será identificada como tal. Al momento de escribir este texto no todos los controladores de OpenGL 3.0 funcionaban en el contexto de compatibilidad futura, por lo que todo el código fuente usará un contexto compatible con versiones anteriores, pero no se usarán funciones obsoletas. En el capítulo 2, "Creación de una aplicación simple de OpenGL" usted aprenderá a crear un contexto de compatibilidad futura en la plataforma Windows.

Bibliotecas relacionadas

Existen muchas bibliotecas disponibles que se construyen sobre y alrededor de OpenGL, con el propósito de dar mayor soporte y funcionalidad más allá del apoyo al render de bajo nivel que lo caracteriza. No tenemos espacio para cubrir todas las bibliotecas relacionadas de OpenGL, y siempre hay nuevas surgiendo, por lo que aquí limitaremos nuestra cobertura a dos de las más importantes: GLUT y SDL. Un poco más adelante, cuando analicemos las extensiones, estudiaremos una biblioteca adicional, GLee.

GLUT

GLUT, por las siglas de OpenGL Utility Toolkit, es un conjunto de bibliotecas de apoyo disponible en todas las plataformas principales. OpenGL no admite directamente la creación de alguna forma de ventana, menú o entrada. Ahí es donde interviene GLUT, que proporciona una funcionalidad básica en todas estas áreas, sin dejar de ser independiente de la plataforma. De este modo, es posible transferir con facilidad las aplicaciones basadas en GLUT, por ejemplo, de Windows a UNIX con pocos o ningún cambio.

GLUT es fácil de usar y aprender, y aunque no proporciona toda la funcionalidad que ofrece el sistema operativo, se desempeña bastante bien para demos y aplicaciones sencillas.

Debido a que su objetivo final será la creación de un juego bastante complejo, usted necesitará

mayor flexibilidad que la ofrecida por GLUT. Por esta razón, no se utiliza en el código del libro. Sin embargo, si desea saber más acerca de GLUT, visite su sitio web oficial en http://www.opengl.org/resources/libraries/glut.html.

SDL

La Simple Direct Media Layer (SDL), es una biblioteca multimedia de plataforma cruzada que incluye soporte para audio, entrada, gráficos en 2D, y muchas otras características. También proporciona soporte directo para gráficos en 3D a través de OpenGL, así que es una opción popular para el desarrollo de juegos multiplataforma. Usted puede encontrar más información sobre SDL en www.libsdl.org.

Un vistazo rápido

Ahora nos adelantaremos un poco y daremos un vistazo a una sección de código en OpenGL. En este momento no tendrá mucho sentido, pero después de leer algunos capítulos lo entenderá por completo. En el CD, abra el proyecto llamado "Simple", almacenado en la carpeta Chapter 1. En este programa de ejemplo se muestran dos polígonos superpuestos.

Advertencia

En el siguiente ejemplo se utilizarán las funciones matriciales ahora obsoletas `glMatrixMode()`, `gluLookAt()`, `glLoadIdentity()` y `gluPerspective()`, así como el modo de render inmediato que ya es obsoleto para que el código no sea muy complicado. En el capítulo 6 aprenderá cómo manejar sus propias matrices en lugar de éstas.

Observe que este código utiliza SDL para los aspectos específicos del sistema operativo. Lo anterior hace que el código se mantenga simple y así usted pueda centrarse en las instrucciones de OpenGL. En este ejemplo se utiliza la forma original o el estilo antiguo de render, llamada *modo inmediato*. Con este método, las primitivas se forman al enviar a OpenGL una serie de vértices de uno en uno. Posteriormente se estudiarán otros métodos más eficientes para el render, pero primero conoceremos el método `initialize()` que se utiliza después de tener un contexto en OpenGL.

```
bool SimpleApp::initialize()
{
  //Permita la prueba
  glEnable (GL_DEPTH_TEST);
```

```
  //Fije la matriz de proyección
  resize(WINDOW_WIDTH, WINDOW_HEIGHT);
  return true;
}
```

Esto no es particularmente emocionante (¡por ahora!). El primer comando de OpenGL activa el búfer en z, lo que garantiza que los objetos más cercanos al espectador se vean por encima de los objetos que se encuentran más lejos. La línea final en `initialization()` llama al método `resize()`. Este método configura lo que se conoce como la *matriz de proyección*. Lo anterior es necesario para que las primitivas se desplieguen de manera correcta y los objetos más alejados de la cámara se vean más pequeños. El método `resize()` se solicita automáticamente cada vez que la ventana cambia de tamaño. A continuación se ve el método en detalle:

```
void SimpleApp::resize(int w, int h)
{
  //Prevenga dividir un cero por error
  if (h <= 0)
  {

      h = 1;
  }

  //Al cambiar el tamaño de la ventana, informamos a OpenGL el nuevo
  tamaño de visualización;
  glViewport (0, 0, (GLsizei)w, (GLsizei) h);

  glMatrixMode (GL_PROJECTION); //obsoleto
  glLoadIdentity ();
  //Después configuramos nuestra matriz de proyección con la relación de
  aspecto correcta
  gluPerspective(60.0f, float(w)/float(h), 1.0f, 100.0f);//obsoleto

  glMatrixMode(GL_MODELVIEW); //obsoleto
  glLoadIdentity(); //obsoleto
}
```

Esto configura el modo en que los objetos del mundo se transforman en píxeles sobre la pantalla.

Por último, daremos un vistazo al código que en realidad hace el render, que de manera poco sorprendente se llama método `render()`. Este método se solicita con cada fotograma para actualizar la pantalla.

```
glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glLoadIdentity();

gluLookAt(0.0, 1.0, 6.0, //Posición
          0.0, 0.0, 0.0, //Hacia donde estamos mirando
          0.0, 1.0, 0.0); // Vector superior
```

Estas primeras líneas limpian la pantalla y restablecen la vista. `gluLookAt()` configura una cámara que mira hacia abajo del eje z hacia nuestros polígonos.

```
glBegin (GL_TRIANGLES);
  // Envía los vértices y colores para el triángulo
  glColor4f(1.0f, 0.0f, 0.0f, 1.0f);
  glVertex3f(2.0f, 2.5f,-1.0f);
  glColor4f(0.0f, 1.0f, 0.0f, 1.0f);
  glVertex3f(-3.5f, -2.5f, -1.0f);
  glColor4f(0.0f, 0.0f, 1.0f, 1.0f);
  glVertex3f(2.0f, -4.0f, -1.0f);
glEnd();
```

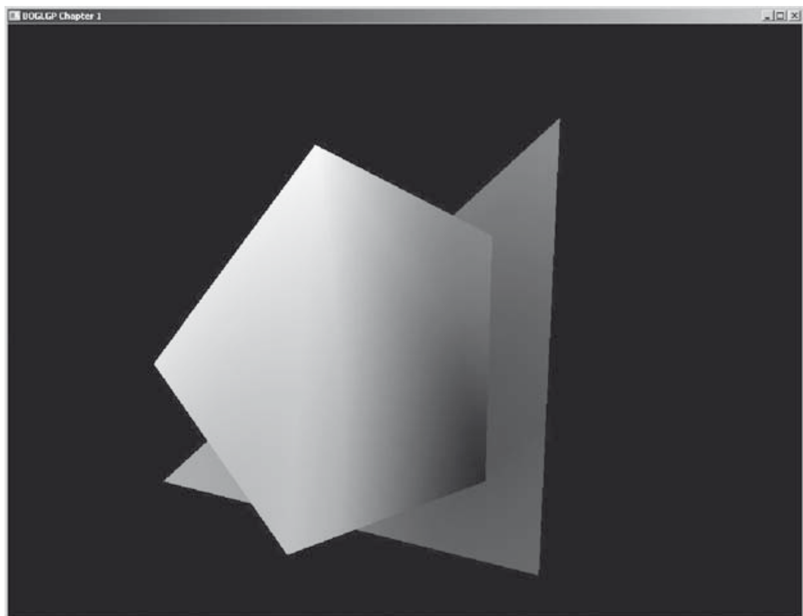
Estas líneas dibujan un triángulo al transmitir a OpenGL una lista de *vértices* (puntos en el espacio tridimensional). La primera línea le dice a OpenGL que estamos a punto de representar un triángulo. Antes de cada declaración de vértice, cambiamos el color actual.

```
glBegin(GL_TRIANGLE_FAN);
  //Envía los vértices y colores para el pentágono
  glColor4f(1.0f, 1.0f, 1.0f, 1.0f);
  glVertex3f(-1.0f, 2.0f, 0.0f);
  glColor4f(1.0f, 1.0f, 0.0f, 1.0f);
  glVertex3f(-3.0f, -0.5f, 0.0f);
  glColor4f (0.0f, 1.0f, 1.0f, 1.0f);
  glVertex3f(-1.5f, -3.0f, 0.0f);
  glColor4f(0.0f, 0.0f, 0.0f, 1.0f);
  glVertex3f(1.0f,-2.0f, 0.0f);
  glColor4f (1.0f, 0.0f, 1.0f, 1.0f);
  glVertex3f(1.0f, 1.0f, 0.0f);
glEnd ();
```

Aquí hacemos casi lo mismo con un pentágono, sólo que esta vez representamos un *abanico de triángulos*. Éste consiste en una serie de triángulos que comparten un solo vértice. El primer triángulo se especifica usando tres vértices y después, para cada otro vértice enviado a Open-

GL, se forma un triángulo diferente al combinarlo con el vértice anterior y el primer vértice del conjunto. Si esto suena complicado, no se preocupe, los abanicos de triángulos se explicarán a detalle en el capítulo siguiente.

Figura 1.4
Un ejemplo sencillo en OpenGL



Resumen

En este capítulo, usted dio un primer vistazo a OpenGL, la aplicación que usará por el resto de este libro para las demostraciones de gráficos y juegos. Ahora que tiene una visión general de la API que va a utilizar, puede seguir con la parte divertida que es ¡el desarrollo real!

Lo que se aprendió

- Los juegos en 3D son un campo emocionante que crece con rapidez.
- OpenGL es una biblioteca de gráficos que se usa en muchas aplicaciones.
- OpenGL ha existido durante más de 16 años. Su desarrollo está supervisado por el grupo Khronos.
- OpenGL ha tenido algunos cambios radicales en su versión más reciente.
- Las bibliotecas como SDL pueden usarse en conjunto con OpenGL para acelerar su desarrollo y agregarle funcionalidad.

Preguntas de repaso

1. ¿Cuándo se presentó OpenGL?
2. ¿Cuál es la versión más reciente de OpenGL?
3. ¿Quién decide qué cambios se realizan en OpenGL?

Por su cuenta

1. Modifique el programa de ejemplo para que el pentágono sea rojo y el triángulo sea azúl.
2. Añada otro triángulo al programa.

Creación de una aplicación sencilla de OpenGL

Antes de poder iniciar con el render de gráficos con OpenGL, necesitamos crear una ventana compatible con OpenGL.

En este capítulo, usted aprenderá sobre:

- WGL y otras funciones relacionadas con Windows compatibles con OpenGL
- Formatos de pixel
- Uso de OpenGL con Windows
- OpenGL a pantalla completa

Acerca de la plataforma

Antes de comenzar a implementar nuestra aplicación de OpenGL, primero debemos analizar algunos aspectos sobre la plataforma. Aunque OpenGL es una API multiplataforma, hay muchos otros factores en el desarrollo de un juego que dependerán de la plataforma sobre la que se ejecutará el programa. Cada sistema operativo tiene una manera diferente de crear una ventana, generar un contexto para OpenGL y procesar eventos.

Como es lógico, existen métodos para crear aplicaciones multiplataforma (uno de ellos se estudió en el capítulo anterior al utilizar la biblioteca SDL), pero las bibliotecas SDL y muchos otros métodos introducen dependencias con el programa y podrían no ser compatibles con la funcionalidad que requiere la plataforma de su elección.

Resulta obvio que no podemos cubrir todas las plataformas a detalle en este libro, así que debe hacerse una concesión. Debido a que Windows es la plataforma más extendida (especialmente cuando se trata del desarrollo de juegos), en este capítulo se mostrará cómo crear una clase de ventana de OpenGL que utiliza la API Win32. Para aquellos que emplean GNU/Linux, en el CD se puede encontrar una clase equivalente escrita usando GLX, la cual representa un reemplazo de la versión en Win32.

Éste es el único capítulo del libro que es específico para la plataforma Windows. Todo lo demás en el libro funciona perfectamente bien en cualquier plataforma con un contexto útil para OpenGL.

Introducción a WGL

El conjunto de API utilizadas para configurar OpenGL en Windows es conocido colectivamente como WGL, en ocasiones pronunciado "wiggle". Algunas de la cosas que WGL le permite hacer son:

- Crear y seleccionar un contexto de render
- Usar el soporte de las fuentes de Windows en aplicaciones de OpenGL
- Cargar extensiones de OpenGL

Las fuentes y extensiones se estudiarán en el capítulo 11, "Visualización de texto" y en el capítulo 5, "Extensiones de OpenGL", respectivamente. En este capítulo se analiza el renderizado de contextos.

Nota

WGL proporciona una funcionalidad considerable además de lo mencionado en esta lista. Sin embargo, las características adicionales son muy avanzadas (y requieren extensiones) o bien son muy especializadas; por consiguiente, no se estudian en el presente volumen.

El contexto de render

Para que un sistema operativo pueda funcionar con OpenGL, se necesita un medio que conecte a OpenGL con una ventana. Si está permitida la ejecución de varias aplicaciones a la vez, también se requiere una forma de evitar que las múltiples aplicaciones de OpenGL interfieran entre sí. Esto se hace a través del uso de un *contexto de render*. En Windows, la interfaz del dispositivo gráfico (o GDI) utiliza un contexto de dispositivo para recordar las especificaciones de modos de dibujo y comandos. El contexto de rendering tiene el mismo propósito para OpenGL. Sin embargo, tenga en cuenta que en Windows, un contexto de render no sustituye a un contexto de dispositivo. Los dos interactúan para asegurar que su aplicación se comporte

adecuadamente. De hecho, es necesario establecer el contexto de dispositivo primero y luego crear el contexto de render con un formato de pixel correspondiente. Los detalles de esto se analizarán en breve.

En realidad usted puede crear varios contextos de render para una sola aplicación. Esto es útil para aplicaciones tales como herramientas de modelado en 3D, donde se tienen varias ventanas o puertos de visualización, y en cada uno es necesario dar seguimiento a su configuración de manera independiente. También podría utilizarse para tener un contexto de rendering que administre su pantalla principal mientras que otro administra los componentes de la interfaz del usuario. La limitación es que sólo puede haber un contexto de render activo por subproceso en un momento dado, aunque se pueden tener varios subprocesos —cada uno con su propio contexto— haciendo el render en una sola ventana a la vez.

A continuación se dará un vistazo a las funciones WGL más importantes para el manejo de contextos.

wglCreateContext

Antes de poder utilizar un contexto de render, es necesario crearlo. Esto se hace a través de:

```
HGLRC wglCreateContext (HDC hdc);
```

`hDC` es el manejador o handle del contexto de dispositivo que haya creado previamente para su aplicación en Windows. Usted debe llamar esta función sólo después de haber establecido el formato de pixeles del contexto de dispositivo, de manera que los formatos de pixel coincidan. (En breve se estudiarán los formatos de pixel.) En vez de devolver el contexto de render real, se obtiene un manejador, que se puede utilizar para pasar el contexto de rendering a otras funciones. No es posible utilizar `wglCreateContext()` para crear un contexto en la versión 3.0 o posterior, en su lugar debe usar `wglCreateContextAttribsARB()`, que se analizará en un momento.

wglDeleteContext

Cuando haya terminado con su contexto de render, debe hacer saber al sistema que puede liberar los recursos establecidos para éste. Lo anterior puede hacerse usando:

```
BOOL wglDeleteContext(HGLRC hRC);
```

wglMakeCurrent

Para llamar a cualquier función de OpenGL, debe tener un contexto de render activo en ese momento. Esto tiene sentido porque un contexto contiene todo el estado necesario para que OpenGL opere. La función que necesita usar para hacer que un contexto sea el actual en el subproceso activo es:

```
BOOL wglMakeCurrent(HDC hDC, HGLRC hRC);
```

Para que la función opere, el `hDC` y el `hRC` deben compartir el mismo formato de pixel. Si desea anular la selección del contexto actual, debe pasar `NULL` para el segundo parámetro.

Para que OpenGL funcione de manera correcta, `wglCreateContext()` y `wglMakeCurrent()` deben llamarse durante la etapa de inicialización. Esto se puede hacer si se les llama cuando se maneje el mensaje `WM_CREATE`. Del mismo modo, puede llamar a `wglDestroyContext()` cuando la aplicación maneje el mensaje `WM_DESTROY`. Es recomendable anular la selección del contexto de render actual antes de llamar a `wglDestroyContext()`.

A continuación se presenta un pequeño fragmento de código para demostrar la inicialización y la destrucción de un contexto en OpenGL mediante el manejo de mensajes de Windows:

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM
lParam)
{
  static HRC hRC;
  static HDC hDC;

  switch (uMsg)
  {
    case WM_CREATE:          // En el mensaje de creación
         hDc = GetDC (hWnd);   // Obtiene el contexto de dispositivo para la
                                 ventana
         hRC = wglCreateContext(hDC); // Se crea un nuevo contexto de render
         wglMakeCurrent(hDC, hRC); // Se actualiza para el subproceso activo
    break;
    case WM_DESTROY:                // En el mensaje de destruir
         wglMakeCurrent(hDC, NULL); // Anula la selección del contexto actual
         wglDeleteContext(hRC); // Borra el contexto de render
         PostQuitMessage(0); // Envía el mensaje WM_QUIT
    break;
  }
}
```

wglCreateContextAttribsARB

Antes de OpenGL 3.0, lo anterior hubiera sido todo lo que necesitaba saber acerca de la creación y manejo de contextos de render. Pero como se mencionó en el capítulo anterior, en la versión 3.0 se introdujo un contexto especial de compatibilidad futura (o “compatible con futuras versiones”) que sólo permite la funcionalidad no obsoleta. No hay manera de pasar ninguna información adicional en `wglCreateContext` para especificar que desea un contexto de compa-

tibilidad futura (o "compatible con futuras versiones"), o escoger para qué versión de OpenGL desea crear el contexto de compatibilidad futura (o "compatible con futuras versiones") (por ejemplo, OpenGL 3.1 puede volver obsoletas más funcionalidades que la versión 3.0). Así que OpenGL 3.0 introdujo una nueva función llamada `wglCreateContextAttribsARB`. En la actualidad, `wglCreateContextAttribsARB` se ofrece como una extensión y se accede a ella utilizando el mecanismo de extensión que se describe a detalle en el capítulo 5.

La definición de la función es la siguiente:

```
HGLRC wglCreateContextAttribsARB(HDC hDC, HGLRC hShareContext, const
int *attribList)
```

Al igual que `wglCreateContext()`, la función toma el contexto de dispositivo como un parámetro, pero con la adición de dos nuevas órdenes. La primera, `hShared Context`, es un contexto de OpenGL creado previamente con el que se desea compartir datos (texturas, búferes, etcétera). Normalmente, usted pasaría NULL para este parámetro si no desea compartir recursos. El último parámetro toma una serie de pares de datos clave-valor en un arreglo que establece las opciones para el nuevo contexto. En este arreglo es posible especificar la versión de OpenGL que desee (`WGL_CONTEXT_MAJOR_VERSION_ARB` y `WGL_CONTEXT_MINOR_VERSION_ARB`) así como banderas de contexto especiales usando `WGL_CONTEXT_FLAGS_ARB`. Este elemento en el arreglo es una máscara de bits que se puede utilizar para especificar si desea habilitar un contexto de compatibilidad futura (o "compatible con futuras versiones") (`WGL_CONTEXT_FORWARD_COMPATIBLE_BIT_ARB`) y/o un contexto de depuración (`WGL_CONTEXT_DEBUG_BIT_ARB`).

Es interesante señalar que, como esta función es una extensión de OpenGL, se requiere tener un contexto OpenGL para acceder a ella ... y poder usarla para crear el contexto OpenGL. Esto parece imposible, pero la clave es utilizar el antiguo `wglCreateContext()` para crear un contexto temporal y entonces, mientras ese contexto es el actual, se toma un puntero para la nueva función, se crea el nuevo contexto utilizando el puntero de función y finalmente se destruye el contexto temporal. Una sección de código dice más que mil palabras:

```
//Establecimiento de la versión que queremos, en este caso 3.0
int attribs[] = {

    WGL_CONTEXT_MAJOR_VERSION_ARB, 3,
    WGL_CONTEXT_MINOR_VERSION_ARB, 0,
    WGL_CONTEXT_FLAGS_ARB,
0}; // el cero indica el final del arreglo

//Crea un contexto temporal a fin de obtener un puntero para la función
HGLRC tmpContext = wglCreateContext (m_hdc);
//Lo vuelve el actual
wglMakeCurrent(m_hdc, tmpContext);
```

```
//Obtiene el puntero a la función
wglCreateContextAttribsARB = (PFNWGLCREATECONTEXTATTRIBSARBPROC)
wglGetProcAddress("wglCreateContextAttribsARB");

//Si es NULL, entonces OpenGL 3.0 no es compatible
if(!wglCreateContextAttribsARB)
{
    MessageBox(NULL, "OpenGL 3.0 no es compatible", "Se ha producido un
    error", MB_ICONERROR | MB_OK);
    DestroyWindow(hWnd);
    Return 0;
}

//Crea un contexto de OpenGL 3.0 con la nueva función y el arreglo de atributos
m_hglrc=wglCreateContextAttribsARB(m_hdc, 0, attribs);

//Elimina el contexto temporal
wglDeleteContext(tmpContext);

//Hace que el contexto actual sea GL3
wglMakeCurrent (m_hdc, m_hglrc);
```

Obtención del contexto actual

La mayor parte de las veces usted almacenará el manejador de su contexto de render en una variable global o de elemento, pero en ocasiones no tendrá disponible esa información. Esto ocurrirá a menudo cuando se estén usando múltiples contextos de render en una aplicación multiproceso. Para llevar el manejador al contexto actual puede utilizar lo siguiente:

```
HGLRC wglGetCurrentContext();
```

Si no existe un contexto de render actual, esto devolverá `NULL` como resultado. De manera similar, se puede obtener un manejador o handle para el contexto actual de dispositivo:

```
HDC wglGetCurrentDC();
```

Ahora que usted conoce los fundamentos para manejar contextos de render, es necesario estudiar los formatos de pixeles y la estructura `PIXELFORMATDESCRIPTOR`, así como la manera de utilizarla para configurar su ventana.

Formatos de pixel

OpenGL proporciona un número finito de *formatos de pixel* que incluyen características tales como el modo de color, el búfer de profundidad, número de bits por pixel, y si la ventana tiene búfer doble. El formato de pixel está asociado con la ventana de render y el contexto de dispositivo, los cuales describen qué tipos de datos son compatibles. Antes de crear un contexto de render, es necesario seleccionar el formato de pixel adecuado a utilizar.

Lo primero que debe hacerse es utilizar la estructura `PIXELFORMATDESCRIPTOR` para definir las características y el comportamiento que desea para la ventana. Esta estructura se define como:

```
typedef struct tagPIXELFORMATDESCRIPTOR {
  WORD nSize;           // tamaño de la estructura
  WORD nVersion;        // siempre inicializar a 1
  DWORD dwFlags;        // banderas de las propiedades del búfer de pixel
  BYTE iPixelType;      // tipo de datos de pixel
  BYTE cColorBits;      // número de bits por pixel
  BYTE cRedBits;        // número de bits de color rojo
  BYTE cRedShift;       // conteo de cambios para los bits de color rojo
  BYTE cGreenBits;      // número de bits de color verde
  BYTE cGreenShift;     // conteo de cambios para los bits de color verde
  BYTE cBlueBits;       // número de bits de color azul
  BYTE cBlueShift;      // conteo de cambios para los bits de color azul
  BYTE cAlphaBits;      // número de bits alpha
  BYTE cAlphaShift;     // conteo de cambios para los bits alfa
  BYTE cAccumBits;      // número de bits del búfer de acumulación
  BYTE cAccumRedBits;   // número de bits de acumulación rojos
  BYTE cAccumGreenBits; // número de bits de acumulación verdes
  BYTE cAccumBlueBits;  // número de bits de acumulación azules
  BYTE cAccumAlphaBits; // número de bits de acumulación alfa
  BYTE cDepthBits;      // número de bits del búfer de profundidad
  BYTE cStencilBits;    // número de bits del búfer de plantilla
  BYTE cAuxBuffers;     // número de búfers auxiliares. No compatible
  BYTE iLayerType;      // ya no se utiliza
  BYTE bReserved;       // número de planos superpuestos y subyacentes
  DWORD dwLayerMask;    // ya no de utiliza
  DWORD dwVisibleMask;  // transparencia del plano subyacente
  DWORD dwDamageMask;   // ya no se utiliza
}PIXELFORMATDESCRIPTOR;
```

Ahora se revisarán brevemente los campos más importantes en esta estructura.

Tabla 2.1 Banderas de formato de pixel

Valor	Significado
PFD_DRAW_TO_WINDOW	El búfer puede dibujar en una ventana o en la superficie del dispositivo.
PFD_SUPPORT_OPENGL	El búfer es compatible con el dibujado de OpenGL.
PFD_DOUBLEBUFFER	El búfer doble es compatible. Esta bandera y PFD_SUPPORT_GDI son mutuamente excluyentes.
PFD_DEPTH_DONTCARE	El formato de píxel requerido puede tener o no un búfer de profundidad. Para seleccionar un formato de pixeles sin un búfer de profundidad, se debe especificar esta bandera. De lo contrario, sólo se considerarán formatos de pixel con un búfer de profundidad.
PFD_DOUBLEBUFFER_DONTCARE	El formato de pixel requerido puede ser tanto de búfer simple como de búfer doble.
PFD_GENERIC_ACCELERATED	El formato de pixel requerido es acelerado por el controlador de dispositivo.
PFD_GENERIC_FORMAT	El formato de pixel requerido sólo se admite en el software. (Verifique esta bandera si su aplicación se está ejecutando más lentamente de lo esperado.)

nSize

El primero de los campos más importantes en la estructura es nSize. Este campo siempre debe establecerse igual al tamaño de la estructura, como sigue:

```
pfd.nSize=sizeof(PIXELFORMATDESCRIPTOR);
```

Esto es directo y es un requisito común para estructuras de datos que pasan como punteros. A menudo, una estructura necesita conocer su tamaño y la cantidad de memoria que se le ha asignado al momento de realizar diversas operaciones. Un campo para el tamaño permite un acceso fácil y preciso a esta información.

dwFlags

El siguiente campo, dwFlags, especifica las propiedades del búfer de pixel. En la tabla 2.1 se muestran los valores más comunes que se requieren para dwFlags.

iPixelType

El campo iPixelType especifica el tipo de datos de pixel. Este campo puede configurarse con alguno de los siguientes valores:

- PFD_TYPE_RGBA. Pixeles RGBA. Cada pixel tiene cuatro componentes en este orden: rojo, verde, azul y alfa.
- PFD_TYPE_COLORINDEX. Modo de Paleta. Cada píxel utiliza un valor de color indexado.

Para nuestros propósitos, el campo iPixelType siempre se establecerá en PFD_TYPE_RGBA. Esto le permite utilizar el modelo estándar de colores RGB con un componente alfa para efectos tales como la transparencia.

cColorBits

El campo cColorBits especifica los bits por pixel disponibles en cada búfer de color. En la actualidad, este valor puede establecerse en 8, 16, 24 o 32. Si los bits de color requeridos no están disponibles en el hardware de la máquina, se usará el valor más alto cercano al que se haya seleccionado. Por ejemplo, si usted estableció cColorBits en 24 y el hardware de gráficos no es compatible con 24-bits de renderizado, pero es compatible con 16-bits, el contexto de dispositivo que se creará será de 16 bits.

Configuración del formato de pixel

Después de que los campos de la estructura PIXELFORMATDESCRIPTOR hayan sido establecidos en su valores deseados, la siguiente etapa consiste en pasar la estructura a la función ChoosePixelFormat():

```
int ChoosePixelFormat(HDC hdc, CONST PIXELFORMATDESCRIPTOR*ppfd);
```

Esta función intenta encontrar un formato de pixel predefinido que coincida con el que se especificó mediante PIXELFORMATDESCRIPTOR. Si no puede encontrar una coincidencia exacta, localizará la más cercana posible y cambiará los campos del descriptor del formato de pixel para que coincida con lo que realmente se genera. El formato de pixel se devuelve como un entero que representa una ID. Este valor se puede usar con la función SetPixelFormat():

```
BOOL SetPixelFormat(HDC hdc, int pixelFormat, const PIXELFORMATDESCRIPTOR*ppfd);
```

Lo anterior establece el formato de pixel para el contexto de dispositivo y la ventana asociada con éste. Observe que el formato de pixel puede establecerse sólo una vez para una ventana, por lo que si usted decide cambiarlo, debe destruir la ventana y volver a crearla.

El siguiente listado muestra un ejemplo de la creación de un formato de pixel:

```
PIXELFORMATDESCRIPTOR pfd;
memset(&pfd, 0, sizeof(PIXELFORMATDESCRIPTOR));
pfd.nSize = sizeof(PIXELFORMATDESCRIPTOR);// tamaño
pfd.dwFlags = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL | PFD.DOUBLEBUFFER;
pfd.nVersion = 1; // versión
pfd.iPixelType = PFD_TYPE_RGBA; // tipo de color
pfd.cColorBits = 32; // profundidad de color preferida
pfd.cDepthBits = 24; // búfer de profundidad
pfd.iLayerType = PFD_MAIN_PLANE; // capa principal

// elige el mejor formato de pixel coincidente, regresa al índice
int pixelFormat = ChoosePixelFormat (hDC, &pfd);

// configura el formato de pixel al dispositivo de contexto
SetPixelFormat(hDC, pixelFormat, &pfd);
```

Una de las primeras cosas que podrían notarse acerca de este fragmento es que el descriptor del formato de píxel se inicializa por primera vez como igual a cero, y sólo se establecen algunos de los campos. Esto significa simplemente que para establecer el formato de pixel existen varios campos que usted ni siquiera necesita. En ocasiones, estos otros campos pueden requerirse, pero por ahora pueden establecerse con un valor igual a cero.

Una aplicación de OpenGL

Usted ya tiene las herramientas, así que ahora vamos a aplicarlas. En esta sección del capítulo, se unirán los conocimientos adquiridos en las secciones anteriores para crear una ventana de OpenGL habilitada. El código se ha diseñado para proporcionarle un marco básico que usted puede construir para sus aplicaciones de OpenGL.

El código fuente se divide en dos clases: `GLWindow` y `Example`. `GLWindow` maneja la creación y la destrucción de la ventana, así como el procesamiento de eventos; además mantiene todo el código específico para la plataforma. La clase `Example` encapsula todo el código de OpenGL independiente de la plataforma para cada muestra de libro, por lo que es completamente portátil.

Antes de introducirnos por completo en los detalles de la creación de ventanas, daremos un vistazo al método `WinMain` —el punto de entrada de la aplicación— para obtener una visión general de cómo funciona el programa:

```
#define WIN32_LEAN_AND_MEAN
#define WIN32_EXTRA_LEAN

#include <windows.h>
#include “glwindow.h”
#inc1ude “example.h”

int WINAPI WinMain(HINSTANCE hinstance,
                             HINSTANCE hPrevInstance,
                             LPSTR cmdLine,
                             int cmdShow)
{
    //Establece la configuración de nuestra ventana
    const int windowWidth = 1024;
    const int windowHeight = 768;
    const int windowBPP = 32;
    const int windowFullscreen = false;

    //Ésta es nuestra ventana
    GLWindow programWindow(hInstance);

    //El ejemplo de código OpenGL
    Example example;

    //Adjunta el ejemplo a nuestra ventana
    programWindow.attachExamp1e(&examp1e);
```

El código anterior define nuestros `#includes` y establece los atributos de nuestra ventana (tamaño, color, profundidad y si queremos una ventana a pantalla completa o no). La variable `programWindow` se inicializa al pasar en el argumento `hInstance` de WinMain; esto se requiere más adelante durante la creación de la ventana. La variable `example` maneja nuestra escena en OpenGL y debe adjuntarse a la ventana antes de entrar al ciclo principal.

```
    //Intenta crear la ventana
    if (!programWindow.create(windowWidth, windowHeight, windowBPP,
    windowFullscreen))
    {
      MessageBox(NULL, “No se puede crear la ventana de OpenGL”, “Ha ocu-
      rrido un error”, MB_ICONERROR | MB_OK);
      programWindow.destroy();
      return 1;
  }
```

```
    if(!example.init())//Inicializa nuestro ejemplo
    {
      MessageBox(NULL, “No se puede inicializar la aplicación”,
      “Ha ocurrido un error”, MB_ICONERROR | MB_OK);

      programWindow.destroy();
      return 1;
    }
```

La ventana se crea usando la configuración definida anteriormente; si la creación de la ventana falla por alguna razón, entonces se muestra un mensaje de error antes de destruir la ventana. Una vez que la ventana ha sido creada debe inicializarse el ejemplo. Si `init()` devuelve `false`, el programa se cerrará.

```
    //Éste es el ciclo principal, renderizamos fotogramas hasta
    que isRunning devuelva falso
    while(programWindow.isRunning())
    {
      programWindow.processEvents(); //Procesa cualquier evento de ventana

      //Obtenemos el tiempo que transcurrió desde el último fotograma
      float elapsedTime = programWindow.getElapsedSeconds();

      example.prepare(elapsedTime); //Realiza cualquier lógica de pre-render
      example.render(); // Hace render a la escena

      programWindow.swapBuffers();
    }

    example.shutdown(); //Liberar cualquier recurso
    programWindow.destroy(); //Destruye la ventana del programa

    return 0; //Devuelve verdadero
}
```

¡Aquí es donde sucede toda la acción! Hasta que el sistema de manejo de eventos recibe un mensaje de `WM_DESTROY`, los ciclos del programa crean un solo fotograma en cada periodo. `processEvents()` maneja los mensajes de Windows, tales como cambiar el tamaño de los eventos, y después realiza la acción apropiada. A continuación, `getElapsedSeconds()` devuelve el número de segundos que han transcurrido desde que se le llamó por última vez, así que en este caso devuelve el tiempo necesario para representar un solo fotograma. Esta información

se envía al método prepare(); dicho método se utiliza para realizar cualquier acción necesaria antes de efectuar el render. Si usted está escribiendo un juego, aquí es donde manejaría las pulsaciones de teclas o procesaría los movimientos del jugador. El siguiente paso es render() que, como habrá adivinado, es donde están las llamadas a OpenGL para dibujar en la pantalla. El método swapBuffers() es sólo una envoltura alrededor de la función SwapBuffers de Windows que intercambia el búfer frontal (que es lo que se dibuja en la pantalla) con el búfer posterior (que es donde se realiza el render); este proceso se denomina búfer doble.

Una vez que el método ProcessEvents() recibe un mensaje WM_DESTROY, la bandera running se establece como falsa y se pone fin al ciclo del juego. En este punto, el example se apagará liberando todos los recursos asignados, después se llama al método destroy() para restaurar la configuración de la visualización si fuésemos a pantalla completa.

Éstas son entonces las etapas requeridas para ejecutar la aplicación. Ahora, daremos un vistazo al código que crea la ventana:

```
bool GLWindow::create(int width, int heigth, int bpp, bool fullscreen)
{
    DWORDdwExStyle; // Estilo de ventana extendido
    DWORDdwStyle;   // Estilo de ventana

    m_isFullscreen = fullscreen; //Almacena la bandera de pantalla completa

    m_windowRect.left = (long)0; //Establece el valor de la izquierda en 0
    m_windowRect.right = (long)width; //Establece el valor de la derecha
                                        en el ancho requerido
    m_windowRect.top = (long)0; //Establece el valor superior en 0
    m_windowRect.bottom = (long)heigth; //Establece el valor inferior en
                                          la altura requerida

      //Llena la estructura de la clase de ventana
    m_windowClass.cbSize       = sizeof(WNDCLASSEX);
    m_windowClass.style        = CS_HREDRAW | CS_VREDRAW;
    m_windowClass.lpfnWndProc = GLWindow::StaticWndProc; // Establecemos
    nuestro método estático como el manejador de eventos
    m_windowClass.cbClsExtra  = 0;
    m_windowClass.cbWndExtra  = 0;
    m_windowClass.hInstance   = m_hinstance;
    m_windowClass.hIcon       = LoadIcon(NULL, IDI_APPLICATION);//Icono
    preestablecido
    m_windowClass.hCursor     = LoadCursor(NULL, IDC_ARROW); // puntero
    del mouse preestablecido
    m_windowClass.hbrBackground      = NULL;   //no necesita fondo
```

```
    m_windowClass.lpszMenuName       = NULL    // No hay menú
    m_windowClass.lpszClassName      = "GLClass";
    m_windowClass.hIconSm     = LoadIcon(NULL, IDI_WINLOGO); // Icono
    pequeño del logo de Windows

  //Registra la clase de ventana
    if (!RegisterClassEx(&m_windowClass))
    {
      MessageBox(NULL, "Falla al registrar la clase de ventana", NULL, MB_OK);
      return false;
  }

  if (m_isFullscreen) //si estamos en pantalla completa, debemos cam-
biar el modo de visualización
 {
    DEVMODE dmScreenSettings;   //modo de dispositivo

    memset(&dmScreenSettings, 0, sizeof(dmScreenSettings));
    dmScreenSettings.dmSize = sizeof(dmScreenSettings);

    dmScreenSettings.dmPelsWidth = width;    //ancho de pantalla
    dmScreenSettings.dmPelsHeight = height;  //altura de pantalla
    dmScreenSettings.dmBitsPerPel = bpp;     //bits por píxel
    dmScreenSettings.dmFields = DM_BITSPERPEL | DM_PELSWIDTH | DM_PELSHEIGHT;

    if
    (ChangeDisplaySettings(&dmScreenSettings, CDS_FULLSCREEN) != DISP_
CHANGE_SUCCESSFUL)
    {
      //Establece que el modo de visualización falló, cambio a modo de
      ventana
      MessageBox(NULL, "El modo de visualización falló", NULL, MB_OK);
      m_isFullscreen = false;
    }
  }

  if(m_isFullscreen)   //¿Seguimos en el modo de pantalla completa?
  {
    dwExStyle = WS_EX_APPWINDOW;   //Estilo de ventana extendido
    dwStyle = WS_POPUP;            //Estilo de ventana
```

```
    ShowCursor(False);       //Oculta el puntero del mouse
  }
  else
  {
    dwExStyle = WS_EX_APPWINDOW | WS_EX_WINDOWEDGE;//Estilo de ventana extendida
    dwStyle = WS_OVERLAPPEDWINDOW;            //Estilo de ventana
  }

  AdjustWindowRectEx(&m_windowRect, dwStyle, false, dwExStyle);
  //Ajusta la ventana al tamaño real solicitado

    //Clase registrada, por lo que ahora se crea nuestra ventana
    m_hwnd = CreateWindowEx(NULL,              //estilo extendido

      "GLClass",                               //nombre de la clase
      "BOGLGP - Capítulo 2 - Aplicación de OpenGL", //nombre de la
      aplicación
      dwStyle | WS_CLIPCHILDREN |
      WS_CLIPSIBLINGS,
      0, 0,                                    //coordenada x, y
      m_windowRect.right - m_windowRect.left,
      m_windowRect.bottom - m_windowRect.top,  //anchura, altura
      NULL,                                    // referencia al padre
      NULL,                                    // referencia al menu
      m_hinstance,                             // instancia de aplicación
      this);                   // pasamos un puntero a GLWindow aquí

    //verifica si la creación de la ventana  falló (hwnd sería igual a NULL)
    if(!m_hwnd)
      return 0;

    m_hdc = GetDC(m_hwnd);
    ShowWindow(m_hwnd, SM_SHOW);           //visualización de la ventana
    UpdateWindow(m_hwnd);                  //actualiza la ventana

    m_lastTime = GetTickCount()/1000.0f;   //Inicializa el cronómetro
    return true;
}
```

Los puntos más importantes en esta función son la creación y el registro de la clase de ventana y la llamada a `CreateWindowEx()` para crear la ventana. Ponga atención al último parámetro

de `CreateWindowEx()`. Este parámetro le permite adjuntar algunos datos adicionales a una ventana. En nuestro caso, el puntero `this` se almacena como el dato adicional ¡para que se pueda realizar el truco! En Windows, el manejo de mensajes se realiza mediante una función que pasa como el miembro `lpfnWndProc` de la clase de ventana. Por desgracia, para este parámetro no es posible pasar una función miembro; se debe utilizar una función global que no sea miembro o un método estático de una clase. La desventaja de esto es que no es posible acceder a las variables miembro o funciones miembro desde el procedimiento para el manejo de mensajes. Observe que en el código anterior pasamos `GLWindow::StaticWndProc` como el miembro `lpfnWndProc` que es un método estático. A continuación se analizará ese método:

```
LRESULT CALLBACK GLWindow::StaticWndProc(HWND hWnd, UINT uMsg, WPARAM
wParam, LPARAM lParam)
{
  GLWindow*window = NULL;

    //Si éste es el mensaje de crear
    if(uMsg == WM_CREATE)
  {
    //Obtiene el puntero que almacenamos durante la creación
      window = (GLWindow*)((LPCREATESTRUCT)lParam)->lpCreateParams;

    //Asocia el puntero de la ventana con el hwnd de los otros eventos a
acceder
    SetWindowLongPtr(hWnd, GWL_USERDATA, (LONG_PTR)window);
    }
    else
    {
    //Si éste no es un evento de creación, entonces debimos haber al-
macenado un puntero para la ventana
      window = (GLWindow*)GetWindowLongPtr(hWnd, GWL_USERDATA);

      if(!window)
      {
          //Convierte al manejo en el evento predeterminado
          return DefWindowProc(hWnd, uMsg, wParam, lParam);
      }
    }

  //Llama al participante de nuestra ventana (nos permite acceder a las va-
riables participante)
```

```
  return window->WndProc(hWnd, uMsg, wParam, lParam);
}
```

Cuando se crea una ventana usando `CreateWindowEx()`, Windows envía un mensaje `WM_CREATE`. Los mensajes son procesados por el método `StaticWndProc()` y, como usted puede ver, el mensaje `WM_CREATE` se trata como un caso especial. Cuando se recibe un mensaje `WM_CREATE`, el puntero que fue almacenado por la llamada `CreateWindowEx()` se pasa a `SetWindowLongPtr()` para ser almacenado permanentemente a fin de que otros mensajes lo usen. La próxima vez que llega un mensaje (por ejemplo, `WM_SIZE`) el código en la declaración else lo operará. Esto recuperará el puntero almacenado, y después llamará al método no estático `WndProc` para la ventana. En el método `WndProc`, es posible acceder a variables miembro. El código está totalmente envuelto en la clase `GLWindow` por lo que no hay necesidad de ninguna función global y hace que reemplazar una ventana Win32 (digamos, por una SDL) sea muy fácil.

Ahora veamos el método `WndProc`:

```
LRESULT GLWindow::WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM
lParam)
{

    switch(uMsg)
    {
    case WM_CREATE: //creación de la ventana
    {
      m_hdc = GetDC(hWnd);
      setupPixelFormat();

    // Establece la versión deseada, en este caso 3.0
    int attribs[] = {
      WGL_CONTEXT_MAJOR_VERSION_ARB, 3,
      WGL_CONTEXT_MINOR_VERSION_ARB, 0,
    0}; //cero indica el final del arreglo

    //Crea el contexto temporal a fin de que podamos obtener un puntero
para la función
    HGLRC tmpContext = wglCreateContext(m_hdc);
    //Lo actualiza
    wglMakeCurrent(m_hdc, tmpContext);

    //Obtiene el puntero de la función
    wglCreateContextAttribsARB = (PFNWGLCREATECONTEXTATTRIBSARBPROC)
```

```
wglGetProcAddress("wglCreateContextAttribsARB");

    //Si esto es NULL, entonces OpenGL 3.0 no es compatible
    if(!wglCreateContextAttribsARB)
    {
      MessageBox(NULL, "OpenGL 3.0 no es compatible", "Ha ocurrido un
      error", MB_ICONERROR | MB_OK);
      DestroyWindow (hWnd);
      return 0;
    }

    //Crea un contexto de OpenGL 3.0 con la nueva función
    m_hglrc = wglCreateContextAttribsARB(m_hdc, 0, attribs);
    //Elimina el contexto temporal
    wglDeleteContext(tmpContext);
    //Hacer que el contexto actual sea GL3
    wglMakeCurrent(m_hdc, m_hglrc);

    m_isRunning = true; //Marca nuestra ventana en funcionamiento
  }
  break;
  case WM_DESTROY: //destruye la ventana
  case WM_CLOSE: //la ventana se está cerrando

    wglMakeCurrent(m_hdc, NULL);
    wglDeleteContext(m_hglrc);
    m_isRunning = false; //Finaliza el ciclo principal
    PostQuitMessage(0); //Envía un mensaje WM_QUIT
    return 0;
  break;
  case WM_SIZE:
  {
    int height = HIWORD(lParam); //recupera el ancho y la altura
    int width = LOWORD(lParam);
    getAttachedExample()->onResize(width, height); //Llama al método
de cambio de tamaño o resize del ejemplo
  }
  break;
  case WM_KEYDOWN:
    if(wParam == VK_ESCAPE) //Si se pulsó la tecla escape
    {
```

```
            DestroyWindow(m_hwnd); //Envía un mensaje WM_DESTROY
        }
    break;
    default:
        break;
    }

    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
```

Este método es llamado por Windows cada vez que recibe un mensaje del propio Windows. No entraremos en detalles sobre el sistema de mensajes de Windows, debido a que cualquier buen libro sobre dicha plataforma puede funcionar para ello, pero por lo general usted sólo necesita preocuparse por el manejo de mensajes durante las operaciones de creación, destrucción, y cambio de tamaño de la ventana. Esperamos los siguientes mensajes:

- `WM_CREATE`: Este mensaje es enviado cuando se crea la ventana. Aquí establecemos el formato de píxel, recuperamos el contexto de la ventana de dispositivos, y creamos el contexto de render en OpenGL.

- `WM_DESTROY`, `WM_CLOSE`: Estos mensajes son enviados cuando la ventana se destruye o el usuario la cierra. Aquí destruimos el contexto de render y luego enviamos a Windows el mensaje `WM_QUIT` con la función `PostQuitMessage()`.

- `WM_SIZE`: Este mensaje es enviado cada vez que se cambia el tamaño de la ventana. También se envía durante una parte de la secuencia de creación de la ventana, debido a que el sistema operativo redimensiona y ajusta la ventana de acuerdo con los parámetros definidos en la función `CreateWindowEx()`. Aquí llamamos al método `OnResize` de nuestro ejemplo para poder configurar el visor de OpenGL y los parámetros de la proyección.

- `WM_KEYDOWN`: Este mensaje es enviado cada vez que se pulsa alguna tecla. En este código de mensaje particular, sólo estamos interesados en la recuperación del código clave y en si éste es igual al código virtual de la tecla `ESC`, `VK_ESCAPE`. Si es así, salimos de la aplicación mediante una llamada a la función `DestroyWindow()`.

El método `processEvents()` es responsable de agregar estos mensajes a la cola de los que están listos para su procesamiento:

```
void GLWindow::ProcessEvents()
{
    MSG msg;
```

```
    //Mientras haya mensajes en la cola, los guarda en msg
    while(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
    {
      //Procesa los mensajes de uno en uno
      TranslateMessage(&msg);
      DispatchMessage(&msg);
    }
}
```

El ciclo recorre todos los mensajes de la cola y realiza un procesamiento básico de cada mensaje en la llamada `TranslateMessage()`. El llamado a `DispatchMessage()` es lo que dispara la función `StaticWndProc`.

Puede aprender más acerca de los mensajes de Windows y sobre cómo manejarlos a través del Microsoft Developer Network, MSDN; visite el sitio web de MSDN en http://msdn.microsoft.com.

OpenGL en pantalla completa

En el método `create` descrito antes, es probable que haya notado una sección de código que permite un modo de pantalla completa para nuestra aplicación si la variable `m_isFullscreen` es verdadera. Ahora es el momento de explicar esto con detalle.

Tabla 2.2 Campos DEVMODE importantes

Campo	Descripción
`dmSize`	Tamaño de la estructura en bytes, utilizado para el control de versiones.
`dmBitsPerPixel`	Número de bits por pixel.
`dmPelsWidth`	Ancho de la pantalla.
`dmPelsHeight`	Altura de la pantalla.
`dmFields`	Conjunto de banderas de bit que indican cuáles campos son válidos. Las banderas para los campos en esta tabla son `DM_BITSPERPIXEL`, `DM_PELSWIDTH` y `DM_PELSHEIGHT`.

Para cambiar al modo de pantalla completa, debe utilizarse la estructura de datos `DEVMODE`, que contiene información sobre un dispositivo de visualización. En realidad la estructura es bastante grande, pero por fortuna sólo hay que preocuparse por unos pocos elementos, los cuales se presentan en la tabla 2.2.

Después de haber inicializado la estructura `DEVMODE` es necesario pasarla a `ChangeDisplay-Settings()`:

```
LONG ChangeDisplaySettings(LPDEVMODE pDevMode, DWORD dwFlags);
```

Esto toma a un puntero como el primer parámetro de una estructura `DEVMODE` y un conjunto de banderas que describe exactamente lo que usted desea hacer. En este caso, se pasará `CDS_FULLSCREEN` para remover la barra de tareas de la pantalla y forzar a Windows para que deje solo el resto de la pantalla al cambiar el tamaño y mover ventanas en el nuevo modo de visualización. Si la función es verdadera, devuelve `DISP_CHANGE_SUCCESFUL`. Puede cambiar el modo de visualización de nuevo al estado predeterminado si pasa `NULL` y `0` como los parámetros `pDevMode` y `dwFlags`.

Hay algunas cosas que debe tener en cuenta al momento de cambiar al modo de pantalla completa. La primera es que debe asegurarse de que la anchura y la altura especificadas en la estructura `DEVMODE` coincidan con el ancho y la altura que se utiliza para crear la ventana. La forma más sencilla de asegurar esto consiste en usar las mismas variables de ancho y alto para ambas operaciones. Además, debe asegurarse de cambiar la configuración de la pantalla *antes* de crear la ventana.

La configuración de estilo para el modo de pantalla completa difiere de la de las ventanas regulares, por lo que debe ser capaz de manejar ambos casos. Si no está en el modo de pantalla completa, utilizará el mismo tipo de configuración descrito en el programa de ejemplo para las ventanas regulares. Si está en modo de pantalla completa, necesitará usar la bandera `WS_EX_APPWINDOW` para el estilo extendido y la bandera `WS_POPUP` para el estilo de ventana normal. La bandera `WS_EX_APPWINDOW` lleva a una ventana de nivel superior a la barra de tareas una vez que su propia ventana sea visible. La bandera `WS_POPUP` crea una ventana sin borde, que es exactamente lo que desea con una aplicación a pantalla completa. Otra cosa que probablemente querrá hacer en el modo de pantalla completa es remover de la pantalla el cursor del ratón, lo cual puede hacer empleando la siguiente función:

```
int ShowCursor(BOOL bShow);
```

La clase Example

Ahora que sabe cómo crear una ventana de OpenGL, daremos un vistazo a la definición de la clase `Example`:

```
class Example
{
public:
    Example();
```

```
    bool init ();
    void prepare(float dt);
    void render();
    void shut down();

    void onResize(int width, int height);

private:
    float m_rotationAngle;
};
```

Es necesario mencionar que la clase `Example` no es de ningún modo la única forma de diseñar su aplicación de OpenGL, pero para los propósitos de este libro proporciona una forma muy sencilla y flexible de aislar el código que cambiará para cada muestra, de una manera amistosa entre múltiples plataformas.

Como usted podrá observar, además de los cinco métodos públicos también existe un atributo privado llamado `m_rotationAngle`; éste es sólo para el ejemplo del presente capítulo y será reemplazado por otras variables en los capítulos posteriores, dependiendo de qué es lo que se desea dibujar.

A continuación se muestra la aplicación de la clase `Example` para este capítulo, la cual despliega un triángulo multicolor giratorio:

```
#ifdef_WIN32
#include<windows.h>
#endif

#include<GL/gl.h>
#include<GL/glu.h>
#include“example.h”

Example::Example()
{
    m_rotationAngle = 0. 0f;
}

bool Example::init()
{
    glEnable(GL_DEPTH_TEST);
    glClearColor(0.5f, 0.5f, 0.5f, 0.5f);
```

```
    //Devuelve exitoso
    return true;
  }

  void Example::prepare(float dt)
  {
    const float SPEED = 15.0f;
    m_rotationAngle += SPEED*dt;
    if(m_rotationAngle > 360.0f)
    {
      m_rotationAngle -= 360.0f;
    }
}

void Example::render()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();

    glRotatef(m_rotationAngle, 0.0f, 0.0f, 1.0f);

      glBegin(GL_TRIANGLES);
        glColor3f(1.0f, 0.0f, 0.0f);
        glVertex3f(-0.5f, -0.5f, -2.0f);
        glColor3f(1.0f, 1.0f, 0.0f);
        glVertex3f(0.5f, -0.5f, -2.0f);
        glColor3f(0.0f, 0.0f, 1.0f);
        glVertex3f(0.0f, 0.5f, -2.0f);
      glEnd();
  }
  void Example::shutdown()
  {
    //No hay nada que hacer aquí todavía
  }

  void Example::onResize(int width, int height)
  {
    glViewport(0, 0, width, height);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
```

```
    gluPerspective(45.0f, float(width)/float(height), 1.0f, 100.0f);

    glMatrixMode(GL_MODELVIEM);
    glLoadIdentity();
}
```

El método `init()` permite probar la profundidad y usa `glClearCol()` para cambiar el color de fondo a gris. El método `prepare()` incrementa el ángulo de rotación con base en el tiempo de render de un fotograma para que sea utilizado por `glRotatef()` en el método render. En este ejemplo, el método `OnResize` establece el punto de visualización para la proyección en perspectiva, la cual se describe a detalle en el capítulo 4, "Transformaciones y matrices."

El método `render()` es donde colocamos todas las llamadas de dibujado (o dibujo) de OpenGL. En este método, primero limpiamos los búfers de color y profundidad, los cuales se describen en el capítulo 12, "Búfers de OpenGL." Enseguida, restablecemos la matriz del modelo al cargar la matriz identidad con `glLoadIdentity()`, que se detalla en el capítulo 4.

A continuación, el triángulo se dibuja mediante las funciones `glBegin()`, `glVertex3f()`, y `glEnd()`. Antes de cada vértice cambiamos a un color diferente con `glColor3f()`, que toma un valor entre 0.0 y 1.0 para los parámetros de color rojo, verde y azul. El primer vértice se dibuja en rojo, el segundo en amarillo y el tercero en azul.

Nota

El método que hemos utilizado para el envío de colores y vértices a OpenGL se llama *modo inmediato*. Éste es el método más lento de renderizado y se considera obsoleto en OpenGL 3.0. Sin embargo, es la forma más fácil y comprensible de dibujar, por eso se eligió para este ejemplo. En el próximo capítulo, aprenderá acerca de los vertex buffer objects (o VBO), que constituyen el método preferido y no obsoleto para enviar datos de primitivas en OpenGL 3.0.

Actualizaciones en función del tiempo

Más arriba pasamos por alto el hecho de que un valor de tiempo pasa al método `prepare()`. Este valor se usa para mantener las actualizaciones de movimiento independientes de la rapidez del fotograma. Utilicemos un ejemplo; digamos que en el clásico tirador de primera persona, su personaje dispara un cohete. Si la posición del cohete se actualiza cada fotograma, éste viajaría más rápido en una PC con una velocidad mayor que otra con una rapidez de fotograma más baja.

Para evitar esto, es necesario multiplicar la velocidad constante por los segundos transcurridos en cada fotograma. Lo anterior mantiene todo funcionando a la misma velocidad, sin importar qué tan rápida o lenta sea la PC. Eso es lo que hacemos con el ángulo de rotación en el ejemplo de este capítulo.

Esto es todo lo que necesita saber sobre la creación de una ventana de OpenGL en la plataforma Windows. Ahora, daremos un vistazo a la excitante presentación en pantalla:

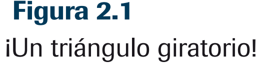
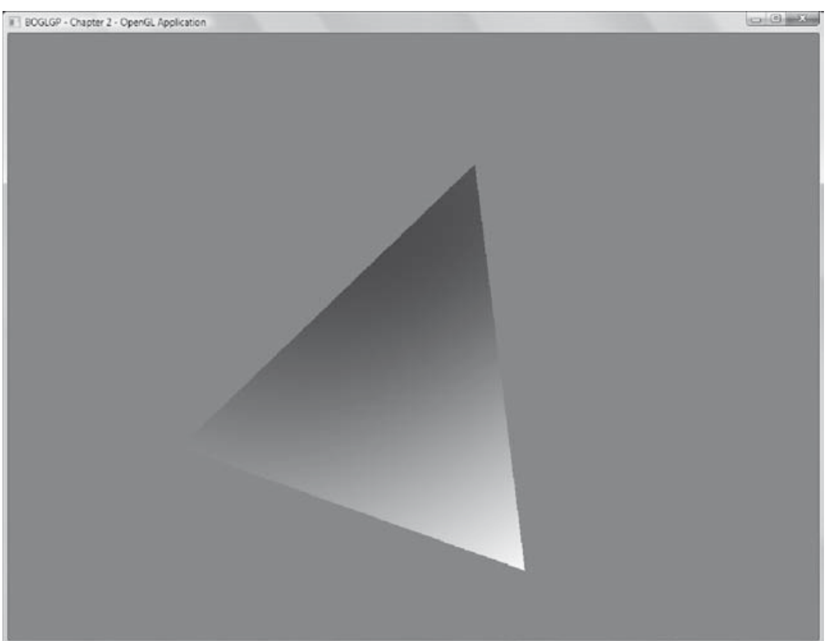


Figura 2.1

¡Un triángulo giratorio!



Resumen

En este capítulo, usted aprendió a crear una aplicación simple de OpenGL en la plataforma Windows. También aprendió sobre el contexto de render y su correspondencia con las funciones "wiggle", `wglCreateContext()`, `wglDeleteContext()`, `wglMakeCurrent()`, `wglGetCurrentContext()` y la nueva `wglCreateContextAttribsARB()`.

También se estudiaron los formatos de pixel y aprendió a configurarlos para OpenGL en el sistema operativo Windows. Por último, proporcionamos el código fuente completo para una aplicación básica de OpenGL y analizamos cómo configurar la ventana para el modo de pantalla completa en OpenGL.

Lo que se aprendió

- Las funciones WGL, o *wiggle*, son un conjunto de extensiones a la API Win32 que se crearon específicamente para OpenGL. Varias de las funciones principales implican el contexto de render, que se usa para recordar la configuración y los comandos de OpenGL. Es posible utilizar varios contextos de render a la vez.
- La estructura `PIXELFORMATDESCRIPTOR` se emplea para describir un contexto de dispositivo usado para hacer un render con OpenGL. Esta estructura debe especificarse y definirse antes de que cualquier código de OpenGL pueda funcionar en una ventana.
- El modo de pantalla completa en OpenGL se usa en la mayoría de los juegos 3D que se están desarrollando en la actualidad. En este capítulo se estudió cómo implementar el modo de pantalla completa en una aplicación de OpenGL y se aprendió cómo realizar actualizaciones que sean independientes de la velocidad de los fotogramas.

Preguntas de repaso

1. ¿Qué es el contexto de render?
2. ¿Cómo se recupera el contexto de render actual?
3. ¿Qué es un `PIXELFORMATDESCRIPTOR`?
4. ¿Qué hace la función de OpenGL `glClearColor()`?
5. ¿Qué estructura se requiere para configurar una aplicación a pantalla completa?

Por su cuenta

1. Modifique la aplicación para mostrar otro triángulo, esta vez en rojo; además, cambie el color de fondo a blanco.

CAPÍTULO 3

Estados y primitivas de OpenGL

¡Por fin llegó el momento de entrar de lleno en OpenGL! Para desencadenar el poder de OpenGL, primero es necesario comenzar con lo básico, y esto implica comprender las primitivas. Para iniciar, debemos analizar algo que surgirá durante el estudio de las primitivas y a lo largo de casi todo el libro a partir de este momento: la máquina de estados de OpenGL.

Después de leer este capítulo, usted aprenderá lo siguiente:

- Cómo acceder a los valores en la máquina de estados de OpenGL
- Los tipos de primitivas disponibles en OpenGL
- Cómo funcionan el modo inmediato y los arreglos de vértice o vertex arrays
- Cómo dibujar primitivas usando vertex buffer objects

Funciones de estado

La máquina de estados de OpenGL consiste en cientos de configuraciones que tienen un número finito de valores posibles (estados). Estas configuraciones son aspectos como el color del render actual, o si está habilitado el texturizado. Cada configuración puede cambiarse de forma individual y localizarse usando la API de OpenGL. Dicha aplicación proporciona una serie de funciones que permiten consultar la máquina de estados para un contexto particular, y la mayor parte de dichas funciones comienzan con `glGet`. Las versiones más genéricas de estas funciones se estudiarán en la presente sección y las más específicas se cubrirán a lo largo del libro, dependiendo de las características con las que están relacionadas.

Nota

Todas las funciones en esta sección requieren de la existencia de un contexto válido de render. De lo contrario, los valores que devuelvan serán indefinidos.

Consulta de estados numéricos

Existen cuatro funciones de uso general que le permiten recuperar valores numéricos (o Booleanos) almacenados en los estados de OpenGL. Son las siguientes:

```
void glGetBooleanv(GLenum pname, GLboolean*params);
void glGetDoublev(GLenum pname, GLdouble*params);
void glGetFloatv(GLenum pname, GLfloat*params);
void glGetIntegerv(GLenum pname, GLint*params);
```

En cada uno de estos prototipos, el parámetro `pname` especifica la configuración de estado que está consultando, y `params` es un arreglo lo suficientemente grande para contener todos los valores asociados con la configuración en cuestión. El número de estados posibles es grande, así que en vez de hacer una lista de todos los estados en este capítulo, analizaremos el significado específico de muchos de los valores `pname` aceptados por estas funciones a medida que surjan. De cualquier forma, la mayor parte de ellos no tendrá mucho sentido en este momento (a menos que usted ya sea un gurú de OpenGL, en cuyo caso, ¿qué hace leyendo esto?).

Por supuesto, la determinación de la configuración de la máquina de estado actual es interesante, pero no tanto como la capacidad de cambiar la configuración. Contra lo que podría esperarse, no hay `glSet()` o alguna función genérica similar para el establecimiento de valores de la máquina de estado. En cambio, existe una variedad de funciones más específicas, de las que hablaremos a medida que se vuelvan más importantes.

Activación y desactivación de estados

Ahora ya sabe cómo encontrar los estados en la máquina de estados de OpenGL, así que ¿cómo activar o desactivar los estados? Introduzca las funciones `glEnable()` y `glDisable()`:

```
void glEnable(Glenum cap);
void glDisable(Glenum cap);
```

El parámetro `cap` representa la capacidad de OpenGL que usted desea habilitar o deshabilitar. `glEnable()` la activa y `glDisable()` la desactiva. ¡Fácil! OpenGL incluye más de 40 funciones que se pueden habilitar y deshabilitar. Entre éstas se incluyen `GL_BLEND` (para operaciones de mezcla), `GL_TEXTURE_2D` (para texturas bidimensionales) y, como ha visto en ejemplos anteriores, `GL_DEPTH_TEST` (para la ordenación de profundidad en el búfer z). A medida que

avance en el libro, aprenderá más capacidades que se pueden activar y desactivar mediante estas funciones.

glIsEnabled()

En ocasiones, sólo se desea saber si una determinada capacidad de OpenGL está activada o desactivada. Aunque esto se puede hacer con `glGetBooleanv()`, es más sencillo utilizar `glIsEnabled()`, que tiene el prototipo siguiente:

```
Glboolean glIsEnabled(Glenum cap);
```

`glIsEnabled()` puede llamarse con cualquiera de los valores aceptados por `glEnable()`/`glDisable()`. Si la capacidad está habilitada, devuelve `GL_TRUE` y en caso contrario, devuelve `GL_FALSE`. Una vez más explicaremos el significado de los diversos valores a medida que surjan en el libro.

Consulta de valores de cadena

Para conocer los detalles de la implementación de OpenGL que está en uso, al momento de la ejecución, puede usar la siguiente función:

```
const GLubyte *glGetString(GLenum name);
```

La cadena terminada en null que se devuelve depende del valor que se pasa como `name`, el cual puede ser cualquiera de los valores de la tabla 3.1.

Tabla 3.1 Parámetros glGetString()

Parámetro	Definición
GL_VENDOR	La cadena devuelta indica el nombre de la compañía, cuya aplicación OpenGL se está utilizando. Por ejemplo, la cadena de proveedor para los controladores ATI es `ATI Technologies Inc`. Por lo general, este valor será siempre el mismo para una empresa determinada.
GL_RENDERER	La cadena contiene información que suele reflejar el hardware utilizado. Por ejemplo, el del autor devuelve GeForce 8400M GS/PCI/SSE2. De nuevo, este valor no cambiará de versión a versión.

Tabla 3.1 Parámetros glGetString() (continuación)

Parámetro	Definición
GL_VERSION	La cadena contiene un número de versión en la forma de `major_number.minor_number` o `major_number.minor_number.release_number`, posiblemente seguida por información adicional proporcionada por el proveedor. Los controladores actuales del autor devuelven `3.0 NVIDIA 177.89`.
GL_EXTENSIONS	La cadena devuelta contiene una lista delimitada por espacios de todas las extensiones de OpenGL disponibles. Esto se tratará en mayor detalle en el capítulo 5, "extensiones de OpenGL". Este parámetro ha quedado obsoleto en favor del uso de glGetString() que se analiza a continuación.

Sugerencia

`glGetString()` proporciona información útil acerca de la implementación de OpenGL, pero tenga cuidado con la forma en la que lo usa. Algunos nuevos programadores lo utilizan para decidir qué opciones de render emplear. Por ejemplo, si saben que una característica es compatible con el hardware de las tarjetas Nvidia GeForce, pero sólo con el software de tarjetas anteriores, pueden comprobar la cadena de render para `geforce` y, si no está allí, deshabilitar la funcionalidad. Ésta es una mala idea. La mejor manera de saber cuáles características son lo suficientemente rápidas como para ser usadas, es determinar sus perfiles la primera vez que se ejecuta el juego y hacerlo de nuevo cada vez que se detecta un cambio en el hardware.

glGetStringi()

`gLGetStringi()` se añadió en OpenGL 3.0 y le permite captar las cadenas de OpenGL usando un índice en lugar de devolver todas las cadenas juntas con espacios. Al momento de escribir este texto, el único parámetro válido es `GL_EXTENSIONS`. El formato de la llamada es:

```
GLubyte*glGetStringi(GLenum name, GLuint index);
```

`index` puede ser cualquier valor entre cero y `NUM_EXTENSIONS-1`. `glGetStringi()` se abordará con mayor detalle en el capítulo 5, "Extensiones de OpenGL."

Localización de errores

Al pasar valores incorrectos en las funciones de OpenGL, se produce y se establece una

bandera de error. Cuando esto sucede, la función regresa sin hacer nada, así que si usted no está consiguiendo los resultados que espera, la consulta de las banderas de error puede ayudarle a tener un seguimiento más sencillo de los problemas en el código. Esto puede hacerse a mediante lo siguiente:

```
GLenum glGetError();
```

Esto devuelve uno de los valores en la tabla 3.2. El valor devuelto indica el primer error que se produjo desde el inicio o desde la última llamada a glGetError(). En otras palabras, una vez que se genera un error, su bandera no se modifica hasta que se haga una llamada a glGetError(); después de hecha la llamada, la bandera de error se restablecerá a GL_NO_ERROR.

Colores en OpenGL

Antes de entrar en detalles sobre el render de primitivas, analizaremos el color de manera breve. En OpenGL (y en general en los gráficos de computadora), un color está formado por una combinación de los tres colores primarios de la luz: rojo, verde y azul. Los diferentes colores se pueden crear mediante la variación de la intensidad de cada componente de color. Además de los tres componentes visibles del color, OpenGL también da seguimiento a otro componente llamado alfa. El canal alfa se utiliza como un factor de contribución en la transparencia y otros efectos.

Tabla 3.2 Códigos de error de OpenGL

Valor	Significado
GL_NO_ERROR	Se explica por sí solo. Esto es lo que queremos que ocurra todo el tiempo.
GL_INVALID_ENUM	Este error se genera cuando se pasa un valor enumerado en OpenGL que la función no acepta normalmente.
GL_INVALID_VALUE	Este error se genera al utilizar un valor numérico que se encuentra fuera del rango aceptado.
GL_INVALID_OPERATION	Este error puede ser más difícil de localizar que los dos anteriores. Ocurre cuando la combinación de valores que se pasa a una función no trabaja en conjunto o bien no funciona con la configuración de estado existente.
GL_INVALID_FRAMEBUFFER_OPERATION	El objeto en el búfer de fotograma no está completo de alguna manera (es decir, hay un accesorio necesario que hace falta).

Tabla 3.2 Códigos de error de OpenGL (*continuación*)

Valor	Significado
GL_STACK_OVERFLOW	OpenGL contiene varios bloques que pueden manipularse directamente, el más común es la pila de matrices o matrix stack. Este error se produce cuando la llamada a la función podría haber causado el sobreflujo de la pila.
GL_STACK_UNDERFLOW	Es como el error anterior, excepto que ocurre cuando la función podría haber causado un subflujo. Esto sucede generalmente cuando se tienen más fotogramas que pulsaciones.
GL_OUT_OF_MEMORY	Este error se genera cuando la operación hace que el sistema se quede sin memoria. A diferencia de las otras condiciones de error, cuando éste ocurre, es posible modificar el estado actual de OpenGL. De hecho, todo el estado de OpenGL, con excepción de la bandera de error en sí, se vuelve indefinido. Si se le presenta este error, debe tratar de salir de la aplicación con tanta gracia como le sea posible.
GL_TABLE_TOO_LARGE	Este error es poco frecuente, ya que sólo pueden generarlo las funciones del subconjunto de imágenes de OpenGL, que no se usa con frecuencia en los juegos. Ocurre como resultado de utilizar una tabla que es demasiado grande para que la aplicación la maneje.

Por lo general, cada componente de color se expresa como un valor de punto flotante entre 0.0 y 1.0, donde 1.0 representa la máxima intensidad y 0.0 indica que no existe contribución del canal de color. Por ejemplo, el negro estaría representado por el establecimiento de los canales rojo, verde y azul en 0.0, mientras que el blanco se especificará estableciendo los tres componentes en 1.0.

En la siguiente sección, se verá que el color de una primitiva puede especificarse de diversas maneras dependiendo del método de render que se esté utilizando. El modo inmediato utiliza el conjunto de funciones `glColor()`:

```
void glColor{34}{bsifd ubusui}(T components);
void glColor{34}{bsifd ubusui}v(T components);
```

El primer conjunto de funciones toma cada valor para el canal de color de manera individual. Las variaciones que terminan en "v" toman un arreglo de valores. Las versiones byte, short e integer de las funciones asignan los valores entre 0.0 y 1.0, donde el máximo valor entero posible se asigna a 1.0.

Cuando se usan vertex arrays y vertex buffer objects, los colores de las primitivas se especifican como arreglos de datos. En la siguiente sección aprenderá más sobre esto.

Manejo de primitivas

Entonces, ¿qué son las primitivas? El diccionario de Merriam-Webster define primitivo como "una persona poco sofisticada". Bueno, eso no ayuda mucho, así que lo definiremos de otra manera; en palabras simples, las primitivas son entidades geométricas básicas tales como puntos, líneas y triángulos.

Para crear sus juegos, usted utilizará miles de estas primitivas, así que es importante saber cómo funcionan. Sin embargo, antes de entrar en los tipos específicos de primitivas, tenemos que hablar acerca de cómo se dibujan en OpenGL.

A través de los años, OpenGL ha obtenido varios métodos para dibujar primitivas. Cada nuevo método se ha diseñado para mejorar el desempeño del render. OpenGL 1.0 era compatible con el modo inmediato (el único método de dibujo que hemos estudiado hasta ahora). Poco después, en OpenGL 1.1, se introdujeron los vertex arrays y, luego, unas cuantas versiones más tarde, en OpenGL 1.5, los vertex buffer objects (VBO) pasaron a formar parte de la base. En los siguientes apartados obtendremos una visión general rápida del modo inmediato y los vertex arrays, antes de pasar a una explicación más detallada de los VBO, que ahora son la forma recomendada para hacer primitivas. El modo inmediato (immediate mode) y los vertex arrays están programados para retirarse en una versión futura de OpenGL; sin embargo, siguen estando disponibles en la versión 3.0 y se utilizan ampliamente en el código existente, por lo que todavía es una buena idea aprender a utilizarlos.

Modo inmediato (Immediate Mode)

Para hacer un render con el modo inmediato, primero debe informar a OpenGL que está a punto de dibujar una primitiva, luego enviar una serie de puntos que forman esa primitiva y, por último, decirle a OpenGL que ha finalizado el dibujo.

Para notificar a OpenGL que está a punto de comenzar el render de primitivas, es necesario usar la siguiente función:

```
void glBegin(GLenum mode);
```

`glBegin()` dice a OpenGL dos cosas: 1) que usted está listo para empezar a dibujar y, 2) el tipo de primitiva que desea dibujar. El tipo de primitiva se especifica con el parámetro mode, que puede tomar cualquiera de los valores de la tabla 3.3.

En la figura 3.1 se ilustran ejemplos de cada uno de los tipos de primitivas que puede dibujar con OpenGL mediante la función `glBegin()`.

Cada llamada a `glBegin()` debe ir acompañada de una llamada a `glEnd()`, que tiene la forma siguiente:

```
void glEnd();
```

Tabla 3.3 Parámetros válidos de glBegin()

Parámetro	Definición
GL_POINTS	Puntos individuales.
GL_LINES	Segmentos de línea individuales compuestos por pares de vértices.
GL_LINE_STRIP	Serie de líneas conectadas.
GL_LINE_LOOP	Ciclo cerrado de líneas conectada, el último segmento se crea automáticamente.
GL_TRIANGLES	Triángulos individuales como tripletes de vértices.
GL_TRIANGLE_STRIP	Serie de triángulos conectados.
GL_TRIANGLE_FAN	Conjunto de triángulos que contiene un vértice central común (el vértice central es el primero que se especifica en el conjunto).
GL_QUADS	Cuadriláteros (polígonos de 4 vértices).
GL_QUAD_STRIP	Serie de cuadriláteros conectados.
GL_POLYGON	Polígono convexo con un número arbitrario de vértices.

Figura 3.1
Tipos de primitivas en OpenGL.

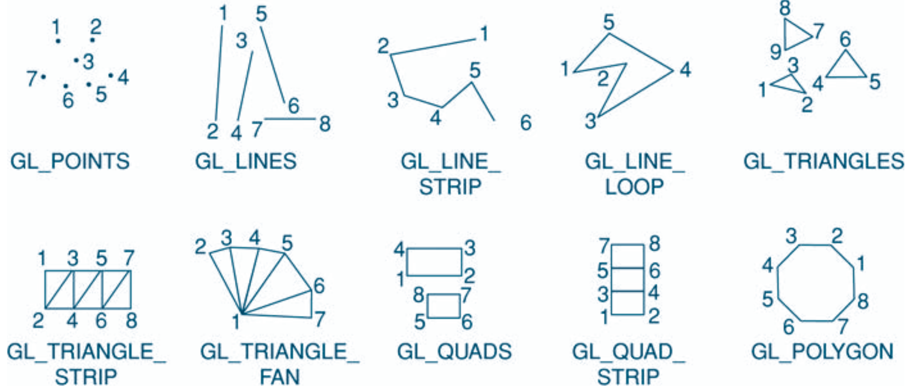


Tabla 3.4 Funciones válidas glBegin()/glEnd()

Función	Descripción
glVertex*()	Establece coordenadas de vértice
glColor*()	Establece el color actual
glSecondaryColor*()	Establece el color secundario
glIndex*()	Establece el índice del color actual
glNormal*()	Establece el vector de coordenadas normales
glTexCoord*()	Establece las coordenadas de textura
glMultiTexCoord*()	Establece las coordenadas de textura para texturizado múltiple
glFogCoord*()	Establece las coordenadas de niebla
glArrayElement()	Especifica los atributos de un vértice único basado en los elementos de un vertex array
glEvalCoord*()	Genera coordenadas al renderizar curvas y superficies de Bezier
glEvalPoint*()	Genera puntos al renderizar curvas y superficies de Bezier
glMaterial*()	Establece las propiedades del material (afecta el sombreado cuando se utiliza la iluminación de OpenGL)
glEdgeFlag*()	Controla el dibujo de bordes
glCallList*()	Ejecuta un listado en pantalla
glCallLists*()	Ejecuta listados en pantalla

Como puede observarse, `glEnd()` no toma ningún parámetro. En realidad no hay mucho que decir sobre `glEnd()`, salvo que le indica a OpenGL que usted ha terminado de hacer el render del tipo de primitiva que especificó en `glBegin()`. Tenga en cuenta que los bloques de `glBegin()/glEnd()` no pueden anidarse.

No todas las funciones de OpenGL pueden utilizarse dentro de un bloque `glBegin()/glEnd()`. De hecho, sólo se pueden usar variaciones de las funciones enlistadas en la Tabla 3.4. Si se emplea cualquier otra llamada de OpenGL se generará un error del tipo `GL_INVALID_OPERATION`.

En medio de las llamadas a `glBegin()` y `glEnd()` deben especificarse los puntos que forman la primitiva. Para ello, se utiliza la familia de funciones `glVertex*()` que toman la forma:

```
void glVertex{234}{dfis}();
```

o bien

```
void glVertex{234}{dfis}v();
```

La versión de `glVertex()` que se utiliza con mayor frecuencia es `glVertex3f()`, la cual tiene tres valores de punto flotante que representan las coordenadas x, y, z del vértice. Las versiones de la función que terminan en "v" adoptan un arreglo de valores como el único parámetro.

Como un ejemplo, el siguiente código dibujará un triángulo usando el modo inmediato:

```
glBegin(GL_TRIANGLES);
glVertex3f(-1.0f, -0.5f, 0.0f);
glVertex3f(1.0f, -0.5f, 0.0f);
glVertex3f(0.0f, 1.0f, 0.0f);
glEnd;
```

Puede dibujar más de un triángulo si agrega más vértices en múltiplos de tres entre `glBegin()` y `glEnd()`.

Vertex arrays

Aunque es útil para primitivas simples, el modo inmediato no es muy eficiente cuando se trata de describir los vértices que conforman un modelo complejo. Lo más conveniente es cargar este tipo de modelos desde un archivo en disco, o bien generarlo mediante un procedimiento que emplea código. De cualquier manera, será necesario manejar una gran cantidad de datos de vértices que deberán enviarse a OpenGL. Una vez que los vértices han sido cargados/generados, se almacenan en una especie de recipiente llamado arreglo (array). La utilización del modo inmediato para enviar estos vértices a OpenGL significará iterar sobre el arreglo y enviar los vértices de uno en uno. Los vertex arrays alivian este problema al especificar el formato y la ubicación en un arreglo para que OpenGL pueda usar los datos de manera directa. Lo anterior evita la necesidad de escribir un recorrido general para procesar los vértices, un método que podría implicar millones de llamadas a funciones como `glVertex*()` y `glColor*()`. No sólo eso, sino que OpenGL también es capaz de optimizar el proceso de render si conoce todos los datos a la vez.

Los vertex arrays tienen otras ventajas. Imagine el render de un cubo empleando triángulos. Un cubo está formado por ocho vértices y 12 triángulos. Cada uno de los vértices será utilizado por varios triángulos, pero si se usa el modo directo será necesario especificar el mismo vértice varias veces, una por cada triángulo empleado. Los vertex arrays permiten que los triángulos compartan los vértices, lo que significa menos datos que almacenar y un menor procesamiento en OpenGL.

Nota

En los ejemplos que se presentarán en el resto del libro, usaremos el recipiente `std::vector` para almacenar datos de vértices. `vector` es parte de la biblioteca de plantillas estándar (STL) de C++ y proporciona un arreglo con cambios de tamaño dinámicos que maneja

Nota (continuación)

su propia memoria. Un vector garantiza que los datos se almacenen de manera contigua en memoria, por lo que es posible acceder a los datos de la misma manera que en una matriz tradicional en C. Como puede observarse, al pasar un puntero a las funciones de OpenGL que requieren un arreglo, en su lugar usamos otro puntero para el primer elemento del vector.

Tabla 3.5 Banderas tipo array

Bandera	Significado
GL_COLOR_ARRAY	Habilita un arreglo que contiene información del color primario para cada vértice
GL_EDGE_FLAG_ARRAY	Habilita un arreglo que contiene las banderas de borde para cada vértice
GL_INDEX_ARRAY	Habilita un arreglo que contiene índices de una paleta de colores para cada vértice

Habilitación de arrays

Antes de poder usar los vertex arrays, es necesario habilitarlos. En este caso, no se utiliza `glEnable()` como podría esperarse; sino que se usa la función `glEnableClientState()`, la cual tiene la siguiente definición:

```
void glEnableClientState(Glenum cap);
```

`glEnableClientState()` toma un parámetro que especifica el tipo de arreglo que queremos activar. Éste puede ser cualquiera de los valores listados en la tabla 3.5. Cuando haya terminado con un arreglo, puede deshabilitarlo utilizando `glDisableClientState()`, que tiene la siguiente definición:

```
void glDisableClientState(Glenum cap);
```

Los vertex arrays se utilizan para enviar atributos de vértices diferentes a los de su posición en 3D. También puede usarse para enviar colores, normales (vectores direccionales utilizados en la iluminación) y coordenadas de textura (utilizadas para posicionar una textura sobre una superficie). Cada propiedad que se desee especificar debe habilitarse de manera individual usando `glEnableClientState()`.

Nota

El término vertex array puede ser un poco confuso. Los vertex arrays (o arreglos de vértice) son una serie de arreglos que declaramos a OpenGL y que almacenan diferentes atributos por vértice (como los colores, las normales, etcétera). Uno de estos arreglos debe almacenar posiciones del vértice y este tipo se denomina vertex array, de ahí la confusión.

Trabajo con arreglos

Una vez que ha habilitado el tipo de arreglo que desea utilizar, debe decirle a OpenGL dónde se encuentran los datos para este arreglo, y en qué formato están almacenados. Para esto se usa el conjunto de funciones `gl*Pointer()`—la función que utilice dependerá de la propiedad del vértice que está especificando, por supuesto, los datos más importante son los de posición:

```
void glVertexPointer(GLint size, GLenum type, GLsize stride, const GLvoid*array);
```

Este arreglo contiene la posición de cada vértice. size debe ser 2, 3, o 4 y `type` puede establecerse en `GL_SHORT, GL_INT, GL_FLOAT o GL_DOUBLE.`

Por ejemplo, digamos que usted ha almacenado su asombroso modelo de alto poligonaje en un arreglo de flotantes, y cada tres flotantes en el arreglo representan un solo vértice (x, y, z). Usted declararía este arreglo a OpenGL usando la siguiente llamada:
`glVertexPointer()` (suponiendo que `myArray` es el arreglo que almacena los datos):

```
glVertexPointer(3, GL_FLOAT, 0, &myArray[0]);
```

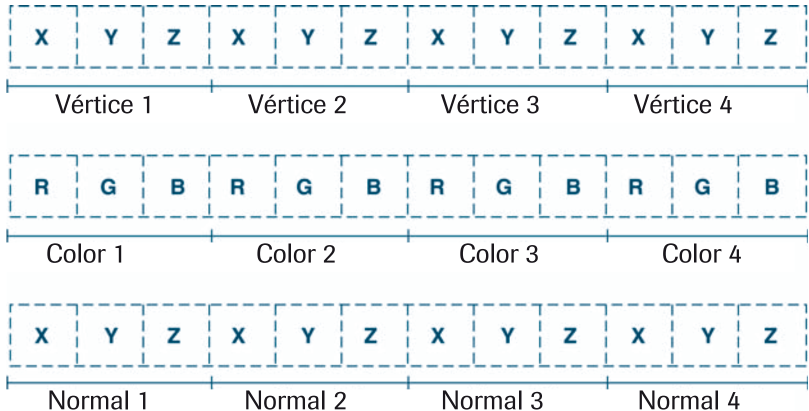
En este ejemplo, se establece que cada vértice tiene tres elementos (x, y, z) y cada elemento es un tipo de punto flotante. El parámetro de paso (cero en este caso) se refiere a la cantidad de relleno en bytes entre cada vértice, lo cual es útil si en el arreglo se almacenan otros datos además de los vértices. No tenemos otros datos, por lo que no se necesita relleno. Por último, se pasa el puntero al inicio del arreglo. Si se desea especificar otras propiedades, podrían almacenarse de la misma manera en arreglos (vea la figura 3.2) y utilizar las funciones apropiadas gl`*Pointer()` para describirlas a OpenGL. Ahora daremos un vistazo a las otras funciones `gl*Pointer()`.

```
void glColorPointer(GLint size, GLenum type, GLsizei stride, const
GLvoid*array);
```

Este arreglo contiene los datos del color primario para cada vértice, `size` es el número de componentes de color que pueden ser 3 (rojo, verde, azul) o 4 (rojo, verde, azul, alpha). `type` puede ser `GL_BYTE`, `GL_UNSIGNED_BYTE`, `GL_SHORT`, `GL_UNSIGNED_SHORT`, `GL_INT`, `GL_UNSIGNED_INT`, `GL_FLOAT` o `GL_DOUBLE`.

```
void glEdgeFlagPointer(GLsizei stride, const GLboolean*array);
```

Figura 3.2
Posible distribución del arreglo para posiciones, colores y normales.



Las banderas edge le permiten ocultar ciertos bordes de los polígonos para hacer un render en modo wireframe (por ejemplo, si un cuadrado se formó con dos triángulos, es posible que en wireframe se desee ocultar el borde diagonal). En este caso, no hay ningún parámetro `type` o `size`, sólo un arreglo de valores Booleanos y el `stride`.

```
void glIndexPointer(GLenum type, GLsizei stride, const GLvoid*array);
```

La nomenclatura de esta función causa cierta confusión. Como era de esperarse, `glIndexPointer()` no especifica un arreglo de índices de primitivas, sino que proporciona una lista de índices de color para los modos de visualización en paleta. Es poco probable que usted vaya a utilizar este tipo de visualización en sus aplicaciones, `type` puede establecerse en `GL_SHORT`, `GL_INT`, `GL_FLOAT` o `GL_DOUBLE`.

```
void glNormalPointer(GLenumtype, GLsizei stride, const GLvoid*array);
```

Esto especifica un arreglo de vectores normales de dirección, en esencia la dirección en que se "observa" cada vértice. Las normales se usan para calcular la iluminación y siempre constan de tres componentes (x, y, z), así que no hay necesidad de un parámetro `size`. `type` puede ser `GL_BYTE`, `GL_SHORT`, `GL_INT`, `GL_FLOAT` o `GL_DOUBLE`.

```
void glTexCoordPointer(GLint size, GLenum type, GLsizei stride, const
GLvoid*array);
```

Este arreglo proporciona una coordenada de textura para cada vértice, `size` es el número de coordenadas por vértice y debe ser 1, 2, 3 o 4. `type` se puede establecer en `GL_SHORT`, `GL_INT`, `GL_FLOAT` o `GL_DOUBLE`.

```
void glSecondaryColorPointer(GLint size, GLenum type, GLsizei stride, const
GLvoid*pointer);
```

Por último, este arreglo contiene una lista de los colores secundarios. `size` es el número de componentes por color, que siempre es tres (rojo, verde y azul). Los tipos permitidos son los mismos que en `glColorPointer()`.

Renderizado usando vertex arrays

Una vez que le ha dicho OpenGL dónde encontrar los datos del vértice utilizando las funciones `gl*Pointer()`,está listo para empezar a hacer el render. Preste atención especial a estas funciones, ya que son muy importantes para el render con vertex buffer objects, que se estudia más adelante en este capítulo.

glDrawArrays() Una vez que ha habilitado los arreglos y le ha indicado a OpenGL dónde se encuentran, está listo para dibujar las primitivas. Existen varios métodos que pueden utilizarse para dibujar vertex arrays, el más utilizado es `glDrawArrays()`, que tiene la siguiente definición:

```
void glDrawArrays(GLenum mode, GLint first, GLsizei count);
```

`glDrawArrays()` dibuja una serie de primitivas usando las cotas actuales de los vertex arrays. `mode` es el tipo de primitiva a dibujar. Los argumentos válidos para mode son los mismos que puede pasar a `glBegin()`. `first` es el índice inicial de los elementos a dibujar (en los arreglos), y `count` es el número de elementos a dibujar. Enseguida daremos un vistazo a un ejemplo sencillo que dibuja un triángulo. En primer lugar, en la inicialización de la aplicación se almacenan los tres vértices que forman el triángulo en un arreglo; para un modelo más complejo, es probable que las posiciones del vértice se carguen desde un archivo.

```
m_vertices.push_back(-2.0f); //X
m_vertices.push_back(-2.0f); //Y
m_vertices.push_back(0.0f); //Z

m_vertices.push_back(2.0f); //X
m_vertices.push_back(-2.0f); //Y
m_vertices.push_back(0.0f); //Z

m_vertices.push_back(0.0f); //X
m_vertices.push_back(2.0f); //etc...
m_vertices.push_back(0.0f);
```

Después, al hacer el render, habilitamos el arreglo de las posiciones del vértice y le indicamos a OpenGL dónde pueden encontrarse los datos. A continuación, usamos `glDrawArrays()` para dibujar los tres vértices y por último desactivamos el vertex array:

```
glEnableClientState(GL_VERTEX_ARRAY); //Habilita el vertex array
//Le dice a OpenGL dónde están los vértices
glVertexPointer(3, GL_FLOAT, 0, &m_vertices[0]);
//Dibuja el triángulo, empezando en el vértice con índice cero
glDrawArrays(GL_TRIANGLES, 0, 3);
//Por último desactiva el vertex array
glDisableClientState(GL_VERTEX_ARRAY);
```

glDrawElements() `glDrawArrays()` es adecuado si cada vértice para cada primitiva está enlistado secuencialmente en el arreglo, pero en ocasiones dos o más primitivas pueden compartir vértices. En esta situación podemos utilizar otro método, `glDrawElements()`, el cual toma una lista de índices en el vertex array que se almacenan en el orden que desee para el render. La función `glDrawElements()` tiene la siguiente especificación:

```
void glDrawElements(GLenum mode, GLsizei count, GLenum type, const GLvoid*indi-
ces);
```

Al igual que en `glDrawArrays()`, `mode` es el tipo de primitivas que desea dibujar. Esta vez `count` es el número de índices que desea dibujar, `type` es el tipo de datos del arreglo de índices y, por último, `indices` es un puntero para el inicio del arreglo.

Ahora veremos un ejemplo: la siguiente sección de código dibuja dos triángulos que forman un cuadrado; dos de los vértices son compartidos por los triángulos. En el ejemplo, `m_vertices` es un `std::vector` de `floats` y `m_indices` es un `std::vector` de enteros sin signo.

Primero, en la inicialización, almacenamos nuestros cuatro vértices, cada uno compuesto por tres números de punto flotante (x, y, z):

```
m_vertices. push_back (-2.0f); //X
m_vertices. push_back (-2.0f); //Y
m_vertices. push_back (0.0f);  //Z

m_vertices. push_back (2.0f);  //X
m_vertices. push_back (-2.0f); //Y
m_vertices. push_back (0.0f);  //Z

m_vertices. push_back (2.0f);  //X
m_vertices. push_back (2.0f);  //etc...
m_vertices. push_back (0.0f);

m_vertices. push_back (-2.0f);
m_vertices. push_back (2.0f);
m_vertices. push_back (0.0f);
```

Después, todavía en la inicialización, se especifican los índices del triángulo en el vertex array:

```
//Primer triángulo, formado por los vértices 1ro., 2do. y 4to.
//(Arreglo de base cero)
m_indices.push_back(0);
m_indices.push_back(1);
m_indices.push_back(3);
//Segundo triángulo, formado por los vértices 2do., 3ro. y 4to.
m_indices.push_back(1);
m_indices.push_back(2);
m_indices.push_back(3);
```

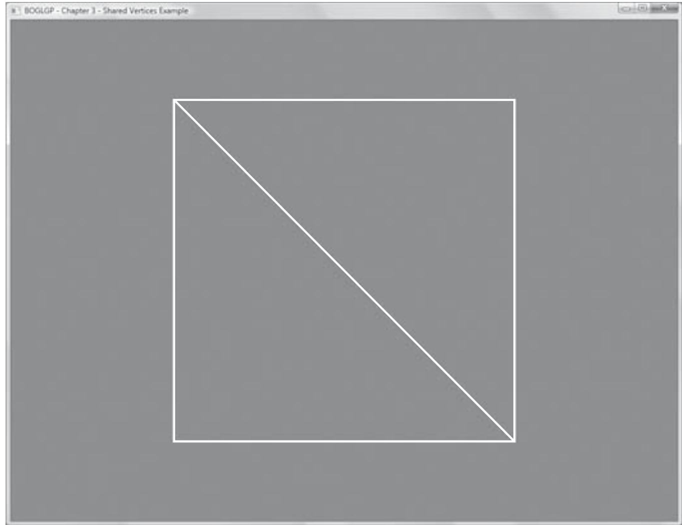
Así que ahora tenemos una serie de cuatro vértices y seis índices. En el render, podemos hacer lo siguiente para representar los triángulos:

```
glEnableClientState(GL_VERTEX_ARRAY); //Habilita el vertex array
//Le dice a OpenGL dónde están los vértices
glVertexPointer(3, GL_FLOAT, 0, &m_vertices[0]);
//Dibuja los triángulos, se pasa el número de índices, el tipo de datos
del arreglo de índices (GL_UNSIGNED_INT) y después se envía el puntero
al inicio del arreglo
```

```
glDrawElements(GL_TRIANGLES, m_indices.size(), GL_UNSIGNED_INT, &m_indi-
ces [0]);

//Finalmente, se desactiva el vertex array
glDisableClientState(GL_VERTEX_ARRAY);
```

El código fuente completo para este ejemplo puede encontrarse en el CD dentro de la carpeta "shared vertices". En la figura 3.3 se muestra una captura de pantalla de la aplicación en ejecución.

Figura 3.3
Vértices compartidos entre triángulos.



glDrawRangeElements() Esta función es casi una extensión de `glDrawElements()`. La diferencia es que `glDrawRangeElements()` le permite especificar un rango de vértices a usar. Por ejemplo, si tiene un vertex array que contiene 1,000 vértices, pero usted sabe que el objeto que está a punto de dibujar accede sólo a los primeros 100 vértices, puede utilizar `glDrawRangeElements()` para decirle a OpenGL que por el momento no utilizará todo el arreglo. Esto puede permitir a OpenGL una transferencia y un almacenamiento más eficiente de los datos de vértice. El prototipo es el siguiente:

```
void glDrawRangeElements (GLenum mode, GLuint start, GLuint end, GLsizei
count, GLenum type, const GLvoid*indices);
```

`mode`, `count`, `type` e `indices` tienen la misma finalidad que los parámetros correspondientes en `glDrawElements()`. `start` y `end` corresponden a las cotas inferior y superior de los índices de vértice contenidos en indices.

glMultiDrawArrays() El último método de dibujo que analizaremos es `glMultiDrawArrays()`. En realidad, ésta es sólo una función de conveniencia que le permite dibujar varios arreglos en una sola llamada. El prototipo de la función es el siguiente:

```
void glMultiDrawArrays(GLenum mode, GLint*first, GLsizei*count, GLsizei prim-
count)
```

Como puede observarse, ahora `first` y `count` son arreglos y el parámetro final indica cuántos elementos hay en cada uno de ellos. Esta función es equivalente al siguiente código fuente:

```
for (int i = 0; i < primcount; ++i)
{
    if (count [i] > 0)
      glDrawArrays(mode, first[i], count[i]);
}
```

Vertex buffer objects

Los vertex arrays son mucho más eficientes para la descripción de grandes cantidades de datos de vértices que el modo inmediato; sin embargo, los datos que usted especifica siguen siendo leídos desde variables del programa que están almacenadas en la memoria de su sistema (RAM) y no en la memoria de su tarjeta de gráficos(VRAM). Estos datos deben enviarse de manera continua a la GPU. Dicho proceso podría ser mucho más rápido si sólo pudiéramos almacenar los datos en búferes en la tarjeta gráfica. Los VBO proporcionan esta funcionalidad, ya que le permiten crear búferes de memoria donde se pueden almacenar y actualizar los datos de vértice y después usarlos para un render más rápido de las primitivas.

Para utilizar un VBO, es necesario realizar los siguientes pasos:

1. Generar un nombre para el búfer.
2. Vincular (activar) el búfer.
3. Almacenar datos en el búfer.
4. Utilizar el búfer para dibujar los datos.
5. Destruir el búfer.

Generación de un nombre

Para generar un nombre para el vertex buffer object, es necesario utilizar la función `glGenBuffers()` que tiene el siguiente formato:

```
void glGenBuffers(GLsizei n, GLuint*buffers);
```

n representa el número de nombres de búfer que queremos generar. buffers es un puntero hacia una variable o arreglo que puede almacenar n nombres de búfer. glGenBuffers() devuelve una serie de nombres enteros de los cuales está garantizado que no se han generado antes mediante una llamada previa a glGenBuffers(), a menos que los nombres hayan sido eliminados previamente por medio de glDeleteBuffers(), que se escribe de la manera siguiente:

```
void glDeleteBuffers(GLsizei n, const GLuint*buffers);
```

Al pasar los mismos parámetros a glDeleteBuffers() que a glGenBuffers() se liberarán todos los nombres que fueron generados. A continuación se presenta un ejemplo que muestra cómo generar y borrar un nombre de búfer único:

```
GLuint bufferID;
glGenBuffers(1, &bufferID); //Genera el nombre y lo guarda en bufferID

//Hace cierta inicialización y algún render con el búfer

glDeleteBuffers(1, &bufferID); //Libera el nombre
```

Resulta valioso señalar que lo único que hace glGenBuffers()es generar un nombre e identificarlo como en uso, el búfer real no se genera hasta que se encuentra vinculado.

Vinculación del búfer

Una vez que haya generado un nombre para el búfer, es necesario vincularlo para poder trabajar con él. Esta vinculación lo convierte en el búfer actual; todas las llamadas y el render de OpenGL relacionados con los búferes se efectuará sobre el búfer actualmente vinculado. Para activar un búfer, debe utilizarse la función glBindBuffer que consta de dos argumentos:

```
void glBindBuffer(GLenum target, GLuint buffer);
```

target puede ser GL_ARRAY_BUFFER, GL_ELEMENT_ARRAY_BUFFER, GL_PIXEL_ PACK_ BUFFER o GL_PIXEL_UNPACK_BUFFER, pero las dos opciones relevantes para los VBO son GL_ARRAY_BUFFER o GL_ELEMENT_ARRAY_BUFFER (las otras opciones se relacionan con pixel buffer objects u objetos del búfer de pixel, el cual permite almacenar datos de píxeles, en vez de datos de vértices). GL_ARRAY_BUFFER se utiliza cuando el búfer contiene datos para cada uno de los vértices (posiciones, colores, normales, etcétera); GL_ELEMENT_ARRAY_BUFFER se utiliza cuando el búfer almacena índices.

buffer es el nombre del búfer generado con anterioridad por glGenBuffers(). Un valor de cero para buffer es especial, ya que en ese caso la llamada desenlazará cualquier búfer que actualmente se encuentre vinculado.

Llenado del búfer

En este punto usted tiene un búfer generado y vinculado listo para recibir algunos datos. En el momento de su creación, el búfer tiene un tamaño cero y, para llenarlo de datos, se llama a glBufferData().

```
void glBufferData(GLenum target, GLsizeiptr size, const GLvoid*data, GLenum
usage);
```

target puede ser GL_ARRAY_BUFFER o GL_ELEMENT_ARRAY_BUFFER (al igual que glBindBuffer()). size es el tamaño en bytes del arreglo de vértice, y data es un puntero hacia el arreglo de datos que deben copiarse (es decir, su arreglo de posiciones de vértice). Por último, usage es un aviso para OpenGL que le dice de qué manera va a utilizar este búfer. Puede ser GL_STREAM_DRAW, GL_STREAM_READ, GL_STREAM_COPY, GL_STATIC_DRAW, GL_STATIC_READ, GL_STATIC_COPY, GL_DYNAMIC_DRAW, GL_DYNAMIC_READ o GL_DYNAMIC_COPY. Cada constante es una combinación de dos partes: la primera es una palabra que describe con qué frecuencia se accede a la memoria intermedia, y la segunda es el tipo esperado de acceso. La explicación de las posibles palabras clave se da en las tablas 3.6 y 3.7.

Recuerde que el usage es sólo una sugerencia para el rendimiento; usted puede seguir utilizando el búfer como quiera, pero si especifica una sugerencia de uso incorrecta es posible que el rendimiento no sea el óptimo.

Al llamar a glBufferData() sobre un objeto de búfer (buffer object) que ya contiene datos, la información antigua será destruida y reemplazada con los nuevos datos. Si al intentar la creación de un almacenamiento se termina la memoria de video, entonces la llamada fallará con un error GL_OUT_OF_MEMORY, que puede verificarse usando glGetError().

Tabla 3.6 Valores de frecuencia del búfer

Valor	Significado
STREAM	Los datos se modificarán sólo una vez, y se accederá a ellos sólo algunas veces.
STATIC	Los datos se alterarán una vez y se accederá a ellos varias veces (esta sugerencia es adecuada para la geometría estática).
DYNAMIC	El búfer se modificará a menudo y será visitado muchas veces (esto es adecuado para modelos animados).

Tabla 3.7 Valores de acceso del búfer

Valor	Significado
DRAW	El contenido del búfer será alterado por la aplicación y se usará para el render con OpenGL.
READ	El contenido será llenado por OpenGL y después será leído por la aplicación.
COPY	El contenido será modificado por OpenGL y luego lo utilizará como fuente para el render.

Render con búferes y vertex arrays

Al hacer un render con VBO, se sigue un proceso muy similar al de los vertex arrays regulares. Sin embargo, cuando un vertex buffer object está vinculado, el parámetro array de las funciones `gl*Pointer()` se convierte en un desplazamiento (en bytes) en el búfer actualmente vinculado, en vez de un puntero hacia una variable del arreglo. Veamos el ejemplo que utilizamos antes; la siguiente línea utiliza `glVertexPointer()` para describir un arreglo de posiciones de vértices para el render con vertex arrays:

```
glVertexPointer(3, GL_FLOAT, 0, &myArray[0]);
```

Al utilizar vertex buffer objects, esto se convierte en:

```
//Vincula el búfer que almacena los datos del vértice
glBindBuffer(GL_ARRAY_BUFFER, bufferObject);
//Le dice a OpenGL que los vértices comienzan a partir del inicio de estos
datos
glVertexPointer(3, GL_FLOAT, 0, BUFFER_OFFSET(0));
```

Lo anterior supone que `myArray` se cargó en un búfer válido con el nombre almacenado en `bufferObject`. Para regresar de nuevo a los vertex arrays estándar, usted puede llamar a `glBindBuffer()` con un cero como el parámetro `buffer`, entonces el parámetro final de las funciones `gl*Pointer()` volverá a ser tratado como un puntero hacia una variable de arreglo.

El desplazamiento es útil si está almacenando más de un tipo de datos en el búfer. Por ejemplo, puede almacenar coordenadas de vértice en la primera mitad de un búfer y colores de los vértices en la segunda mitad. En este caso, se utiliza el desplazamiento en `glColorPointer()` para indicar a OpenGL en qué punto del arreglo inician los datos de color.

Un búfer también puede utilizarse para almacenar índices. Si un búfer se vincula utilizando `GL_ELEMENT_ARRAY_BUFFER` como el target, las funciones de render que toman un arreglo de

índices como `glDrawElements()` o `glDrawRangeElements()`, adoptan un desplazamiento (en bytes) en el búfer vinculado.

Ahora que hemos cubierto los conceptos básicos del render con vertex buffer objects, nos enfocaremos en algunos ejemplos sólidos donde los utilizaremos para dibujar primitivas.

Sugerencia

La especificación de un vertex buffer object define una macro llamada BUFFER_OFFSET para evitar advertencias del compilador cuando se pasa un desplazamiento entero como parámetro del arreglo. Se define de la siguiente manera:

```
#define BUFFER_OFFSET(i) ((char*) NULL + (i))
```

`i` es el desplazamiento en bytes.

Dibujo de puntos en 3D

No hay nada más primitivo de un punto, así que eso será lo que dibujaremos en primer lugar. Cuando se utiliza el modo de render GL_POINTS, cada posición de vértice que envíe a OpenGL se representa con un punto en la pantalla. Veamos un ejemplo que dibuja un solo punto. En primer lugar, antes de dibujar, inicializamos un arreglo con un solo punto en el centro de la pantalla dos unidades hacia atrás, y después crearemos un vertex buffer object:

```
GLfloat vertex[] = {0.0f, 0.0f, -2.0f};
glGenBuffers(1, &m_vertexBuffer); //Genera un búfer para los vértices
glBindBuffer(GL_ARRAY_BUFFER, m_vertexBuffer); //Vincula el búfer de vér-
tices
glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat)*3, &vertex[0], GL_STATIC_DRAW);
//Envía los datos a OpenGL
```

Para dibujar el punto, vinculamos el búfer y establecemos el puntero del vértice (observe que el desplazamiento es cero ya que queremos apuntar al principio del búfer vinculado). Luego, una vez que hemos habilitado el arreglo de vértices, usamos `glDrawArrays()` para dibujar un solo punto.

```
glBindBuffer(GL_ARRAY_BUFFER, m_vertexBuffer);
glVertexPointer(3, GL_FLOAT, 0, BUFFER_OFFSET(0));

glEnableClientState(GL_VERTEX_ARRAY);
glDrawArrays(GL_POINTS, 0, 1);
glDisableClientState(GL_VERTEX_ARRAY);
```

Modificación del tamaño de un punto

De forma predeterminada, los puntos tienen un tamaño de 1.0. Si observa el programa de ejemplo para la sección anterior, se dará cuenta de que el punto apenas puede verse en el centro de la pantalla. El tamaño del punto puede aumentarse utilizando `glPointSize()`, que tiene el prototipo siguiente:

```
void glPointSize(GLfloat size)
```

El resultado es un punto cuadrado con un ancho `size`, con centro en la coordenada de vértice que haya especificado. Si se inhabilita el anti-aliasing del punto (que es el comportamiento por defecto), entonces el tamaño del punto se redondea al entero más cercano, que es el tamaño en pixeles del punto. El tamaño de punto nunca se redondeará a menos de 1.

Anti-aliasing de puntos

Aunque es posible especificar objetos con alta precisión, hay un número finito de pixeles en la pantalla. Esto puede causar que los bordes de las primitivas tengan un aspecto dentado. El proceso de suavizar estos bordes se conoce como anti-aliasing. Puede habilitar el anti-aliasing de puntos al permitir el suavizamiento de puntos, lo cual se hace al pasar `GL_POINT_SMOOTH` a `glEnable()`. El suavizamiento de puntos puede desactivarse de nuevo pasando el mismo parámetro a `glDisable()`.

Cuando está habilitado el suavizamiento de puntos, el rango posible de tamaños de punto puede ser limitado. La especificación a OpenGL sólo requiere que el suavizamiento de puntos sea posible en puntos con un `size` de 1.0, pero algunas implementaciones pueden permitir otros tamaños. Si se utiliza un tamaño no admitido, entonces el tamaño de punto se redondeará al valor compatible más cercano. Con el anti-aliasing activado, el tamaño actual del punto se utiliza como el diámetro de un círculo centrado en las coordenadas x, y del punto especificado.

Nota

Vale la pena señalar que para que el anti-aliasing funcione, el mezclado o blending debe estar habilitado, éste se discute en el capítulo 8, “Iluminación, mezclado y niebla.”

Un ejemplo de puntos

El CD incluye una aplicación de ejemplo llamada “Pointy Example”, que pueden encontrar en la carpeta correspondiente a este capítulo. El programa muestra una serie de puntos que aumentan gradualmente de tamaño. Daremos un vistazo a las partes más importantes del código. En primer lugar, durante la inicialización generamos una fila de 15 puntos espaciados a 0.5 unidades de distancia en el eje *x*.

```
for(float point = -4.0f; point < 5.0; point+=0.5f)
{
    m_vertices.push_back(point); //X
    m_vertices.push_back(0.0f); //Y
    m_vertices.push_back(0.0f); //Z
}

glGenBuffers(1, &m_vertexBuffer); //Genera un búfer para los vértices
glBindBuffer(GL_ARRAY_BUFFER, m_vertexBuffer); //Vincula el búfer de vértice
glBufferData(GL_ARRAY_BUFFER, sizeof(float)*m_vertices.size(),
            &m_vertices[0], GL_STATIC_DRAW); //Envía los datos a OpenGL
```

Después, durante el render, se dibujan los puntos uno a la vez, aumentando el tamaño del punto antes de dibujar cada uno.

```
float pointSize = 0.5f;
for(unsigned int i = 0; i < m_vertices.size()/3; ++i)
{
    glPointSize(pointSize);
    glDrawArrays(GL_POINTS, i, 1); //Dibuja el punto en i
    pointSize += 1.0f;
}
```

En la figura 3.4 se muestra una captura de pantalla del ejemplo de puntos.

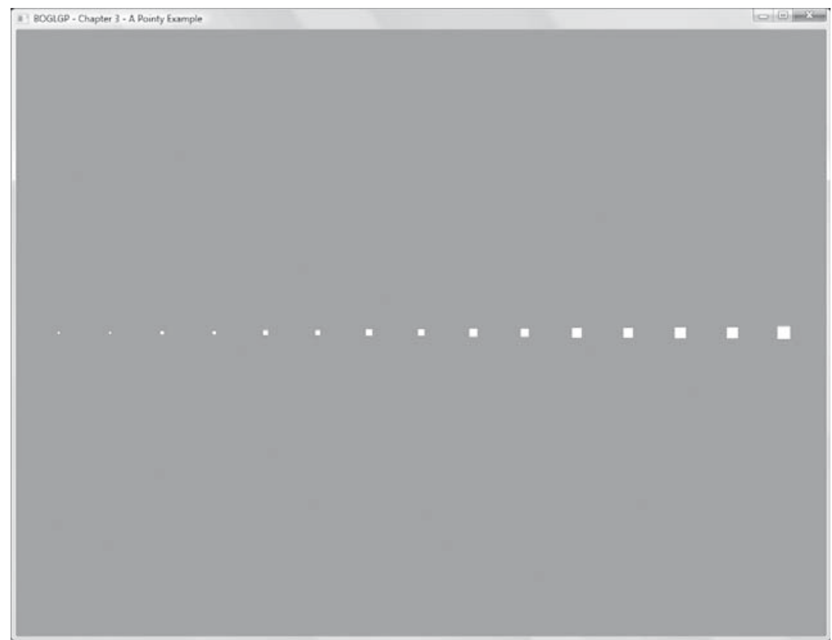
Dibujo de líneas en 3D

El dibujo de líneas en 3D no es muy diferente del dibujo de puntos, excepto que esta vez enviamos los dos vértices finales para cada línea. Veamos cómo se dibuja una sola línea, comenzando con el código de inicialización:

```
GLfloat vertex[] = {-1.0f, 0.0f, -2.0f,
                    1.0f, 0.0f, -2.0f};
glGenBuffers(1, &m_vertexBuffer); //Genera un búfer para los vértices
glBindBuffer(GL_ARRAY_BUFFER, m_vertexBuffer); //Vincula el búfer de vértice
glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat)*6, &vertex[0],
GL_STATIC_DRAW); //Envía los datos a OpenGL
```

Observe que hemos añadido un vértice adicional al arreglo, y aumentó el tamaño del búfer que pasó a `glBufferData()`. Durante el render, los únicos cambios respecto al dibujo de puntos son el modo que se pasa a `glDrawArrays()`, y el número de vértices que se incrementa a 2:

Figura 3.4
Captura de pantalla del ejemplo de puntos.




```
glBindBuffer(GL_ARRAY_BUFFER, m_vertexBuffer);
glVertexPointer(3, GL_FLOAT, 0, 0);

glEnableClientState(GL_VERTEX_ARRAY);
glDrawArrays(GL_LINES, 0, 2); //Ahora estamos dibujando líneas
glDisableClientState(GL_VERTEX_ARRAY);
```

Modificación del ancho de línea

El valor predeterminado para el ancho de una línea es de 1.0. Puede cambiar este valor usando la función que acertadamente se llama `glLineWidth()`.

```
void glLineWidth(GLfloat width);
```

Vale la pena señalar que OpenGL 3.0 considera obsoletos los anchos de línea superiores a 1.0, pero las anchuras mayores todavía están disponibles en un contexto compatible con versiones anteriores.

Anti-aliasing de líneas

El anti-aliasing de líneas funciona de una forma bastante parecida al de puntos. Puede habilitarlo al pasar `GL_LINE_SMOOTH` a `glEnable()` y deshabilitarlo usando `glDisable()`. Al igual que el suavizado de puntos, sólo se garantiza que el suavizado de líneas esté disponible en las líneas con un ancho de 1.0.

Ejemplo del ancho de línea

En el CD se incluye una aplicación llamada "Lines". Esta aplicación simplemente dibuja una columna de líneas, cada una con un ancho diferente. Veamos de nuevo las partes más importantes del código. Durante la inicialización, es necesario generar los vértices que conforman las líneas:

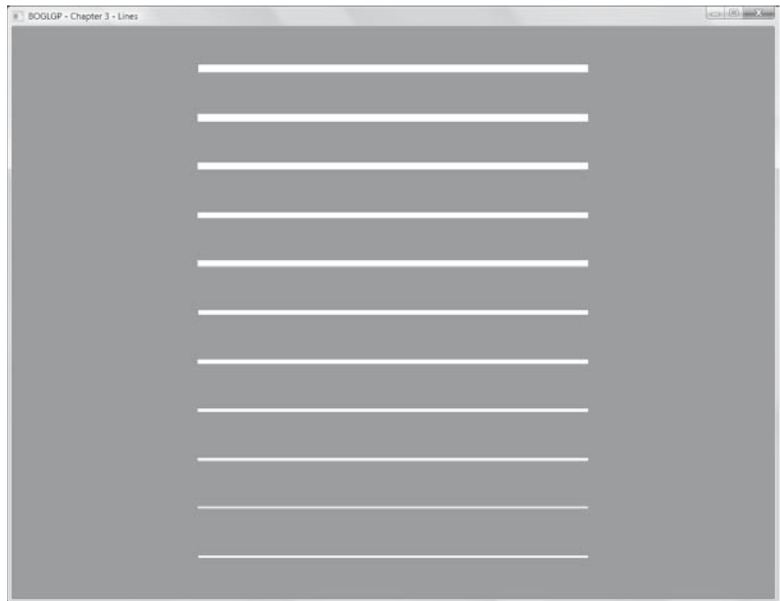
```
for(float line = -3.0f; line < 3.0f; line+=0.5f)
{
    m_vertices.push_back(-2.0f); //X
    m_vertices.push_back(line);  //Y
    m_vertices.push_back(-6.0f);  //Z

    m_vertices.push_back(2.0f); //X
    m_vertices.push_back(line); //Y
    m_vertices.push_back(-6.0f); //Z
}

glGenBuffers(1, &m_vertexBuffer); //Genera un búfer para los vértices
glBindBuffer(GL_ARRAY_BUFFER, m_vertexBuffer); //Vincula el búfer de vértice
glBufferData(GL_ARRAY_BUFFER, sizeof(float)* m_vertices.size(),
            &m_vertices[0], GL_STATIC_DRAW); //Envía los datos a OpenGL
```

De nuevo, el código de render es similar al de la represtación de puntos, excepto que esta vez cambiamos el ancho de las líneas usando `glLineWidth()` en vez de `glPointSize()`. Observe también que en cada iteración `i` se incrementa en dos, para que cada periodo podamos trazar la línea siguiente en el arreglo.

```
float lineWidth = 0.5f;
for(unsigned int i = 0; i < m_vertices.size()/3; i+=2)
{
    glLineWidth(lineWidth);
    glDrawArrays(GL_LINES, i, 2); //Dibuja la línea en i
    lineWidth +=1.0f;
}
```

El resultado de este código puede verse en la figura 3.5.

Figura 3.5
Captura de pantalla del ejemplo de líneas.



Dibujo de triángulos en 3D

Aunque puede hacer algunas cosas interesantes armadas con puntos y líneas, la mayoría de los escenarios de un juego están compuestos por polígonos. OpenGL ofrece modos para el render de cuadriláteros y polígonos con lados arbitrarios; sin embargo, el uso de triángulos se considera una mejor idea. El render de triángulos tiene varias ventajas, la mayor parte del hardware 3D funciona con triángulos internos, ya que son más fáciles y más rápidos para rasterizar (la interpolación de colores y otras acciones se realizan con mayor facilidad a través de un triángulo que de un cuadro) y los puntos de un triángulo se encuentran siempre en el mismo plano en el espacio 3D. Los triángulos siempre son convexos y hacen que los cálculos para tareas distintas al render, como la detección de colisiones, sean más sencillos. Incluso, ARB ha previsto que en una versión futura de OpenGL se retiren los modos de render con polígonos y cuadriláteros. Si tiene una lista de vértices que desea convertir en un polígono de lados arbitrarios, ¡no tenga miedo! Si observa la figura 3.1, se dará cuenta de que los abanicos de triángulos le permiten hacer un polígono complejo construyéndolo a partir de varios triángulos. Es interesante ¿no? Además, la representación de cuatro puntos con una tira de triángulos producirá un cuadrilátero formado por dos triángulos. Por esta razón, no estudiaremos la representación con los modos `GL_QUADS`, `GL_QUAD_STRIP` o `GL_POLYGONS`. Si desea utilizarlos, el render ocurre de la misma manera; simplemente debe cambiar el modo pasado a `glDrawArrays()` o `glDrawElements()`.

Al igual que se hizo con los puntos y las líneas, daremos un vistazo a la manera en que puede hacerse un triángulo único. Construimos la lista de vértices en una forma muy similar:

```
GLfloat vertex[] = {-1.0f, -0.5f, -2.0f,
                    1.0f, -0.5f, -2.0f,
                    0.0f, 0.5f, -2.0f};
glGenBuffers(1, &m_vertexBuffer); //Genera un búfer para los vértices
glBindBuffer(GL_ARRAY_BUFFER, m_vertexBuffer); //Vincula el búfer de vértice
glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat)*9, &vertex[0],
GL_STATIC_DRAW); //Envía los datos a OpenGL
```

Los únicos cambios respecto al render de líneas son el modo que pasamos a `glDrawArrays()` y por supuesto el número de vértices que queremos renderizar.

```
glBindBuffer(GL_ARRAY_BUFFER, m_vertexBuffer);
glVertexPointer(3, GL_FLOAT, 0, 0);

glEnableClientState(GL_VERTEX_ARRAY);
glDrawArrays(GL_TRIANGLES, 0, 3);
glDisableClientState(GL_VERTEX_ARRAY);
```

Como en el caso de los puntos y las líneas, para dibujar más de un triángulo es necesario especificar más vértices en el arreglo, aumentar el tamaño del búfer pasado a `glBufferData()` en forma correspondiente, y cambiar el conteo de vértices pasado a `glDrawArrays()`.

Ejemplo con el modo de polígono

En el CD encontrará un ejemplo llamado “Polygon Mode”, que muestra tres métodos diferentes para el render de polígonos. Puede cambiar la forma en que se representan los polígonos utilizando `glPolygonMode()`, que tiene la siguiente definición:

```
void glPolygonMode(GLenum face, GLenum mode);
```

`face` indica cuál de los lados del polígono cambiará su tipo de render. Éste puede ser `GL_FRONT`, `GL_BACK` o `GL_FRONT_AND_BACK` (en la siguiente sección se estudiarán los lados delanteros y traseros). `mode` puede ser `GL_POINT` (la cara del polígono se representa mediante puntos en cada vértice), `GL_LINE` (la cara se dibuja con líneas), o `GL_FILL` (renderizado normal).

En el ejemplo, se muestran tres cuadros giratorios, cada uno hecho con una franja de triángulo y girando en sentido de las manecillas del reloj. A cada cuadro se le da un modo de polígono diferente; empezando desde la izquierda, se dibujan con la siguiente configuración:

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
glPolygonMode(GL_FRONT_AND_BACK, GL_POINT);
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
```

Nota

En el ejemplo, hemos establecido el modo de dibujo de la parte delantera y trasera de los polígonos al mismo tiempo. En la actualidad, es posible establecer los modos de las caras frontal y posterior de manera individual al pasar `GL_FRONT` o `GL_BACK` como el primer parámetro a `glPolygonMode()`. Siéntase libre de utilizar esta funcionalidad, sólo tenga en cuenta que se ha programado su eliminación para una versión futura de OpenGL, así que no trabajará en un contexto compatible hacia adelante.

Culling de caras de polígonos

A pesar de ser infinitamente delgados, los polígonos tienen dos caras: delantera y trasera, algunas funciones en OpenGL (como `glPolygonMode()`) pueden cambiar el comportamiento de render para uno o ambos lados.

En ocasiones, usted sabrá que el espectador puede ver sólo un lado de un polígono; por ejemplo, los polígonos de una caja sólida y opaca sólo alguna vez tendrán visible su cara frontal. En esta situación, es posible evitar que OpenGL renderice y procese la parte posterior de la primitiva. OpenGL puede hacer esto automáticamente a través del proceso conocido como *culling*. Para utilizar el culling, deberá habilitarlo al pasar `GL_CULL_FACE` a `glEnable()`. Después, puede especificar cuál cara descartar mediante `glCullFace()`, que tiene la siguiente definición:

```
void glCullFace(GLenum mode);
```

`mode` puede ser `GL_FRONT`, `GL_BACK` o `GL_FRONT_AND_BACK`, aunque obviamente el culling de ambas caras ¡no dibujará nada en absoluto! El valor predeterminado es `GL_BACK`.

Usted debe haber notado que el culling sólo es útil si puede determinar qué lado del polígono es el frontal, y cuál es el posterior. Las caras delantera y trasera están determinadas por el *acomodo del polígono* —el orden en que se especifican los vértices. Si mira el polígono de frente, puede elegir cualquier vértice con el cual iniciar su descripción. Para terminar de describirlo, debe proceder en sentido de las manecillas del reloj o al contrario alrededor de sus vértices, OpenGL puede utilizar el acomodo para determinar automáticamente si un polígono es la cara delantera o trasera. Por defecto, OpenGL trata a los polígonos con ordenamiento contrario a las manecillas del reloj

como la cara delantera y a aquellos con ordenamiento a las manecillas del reloj como la vista trasera. El comportamiento predeterminado se puede cambiar con `glFrontFace()`:

```
void glFrontFace(GLenum mode);
```

Si desea utilizar la orientación en sentido antihorario para la cara frontal de los polígonos, `mode` debe ser `GL_CCW` y si quiere usar la orientación horaria, debe ser `GL_CW`.

Anti-aliasing de polígonos

Al igual que con los puntos y las líneas, se puede optar por el anti-aliasing de polígonos; éste se controla pasando `GL_POLYGON_SMOOTH` a `glEnable()` y `glDisable()`. Como era de esperarse, la función se encuentra desactivada por defecto.

Resumen

En este capítulo, usted aprendió un poco más acerca de la máquina de estados de OpenGL. Ya sabe cómo utilizar `glGet()` y `glIsEnabled()` para consultar los valores de los parámetros de OpenGL. También aprendió a obtener información sobre la aplicación OpenGL utilizando `glGetString()` y `glGetStringi()`, así como a encontrar errores utilizando `glGetError()`.

Se estudiaron los diferentes tipos de primitivas que pueden dibujarse usando OpenGL y se vieron tres diferentes métodos para hacerlo: el modo inmediato, los vertex arrays y los vertex buffer objects. Ahora que entiende las bases del render, es momento de avanzar hacia temas más interesantes.

Lo que se aprendió

- Puede consultar la máquina de estados de OpenGL utilizando `glGet()` y `glIsEnabled()`. Las primitivas se dibujan pasando una serie de vértices a OpenGL, ya sea uno a la vez (mediante el modo inmediato) o en un arreglo (vertex arrays y vertex buffer objects). Sabe generar vertex buffer objects, llenarlos con los datos de sus primitivas y usar los búferes en el render con vertex arrays. Ya sabe cómo hacer puntos, líneas y triángulos en OpenGL, así cómo habilitar el anti-aliasing para cada tipo. También aprendió a variar el tamaño de puntos y líneas mediante `glPointSize()` y `glLineWidth()`, respectivamente.
- Puede cambiar la manera en que OpenGL dibuja los polígonos usando `glPolygonMode()`, asimismo puede descartar las caras frontal o trasera de un polígono al pasar `GL_CULL_FACE` a `glEnable()` y elegir el lado a descartar utilizando `glCullFace()`. También sabe que las caras frontales o traseras de los polígonos están determinadas por el acomodo de sus vértices; por último, aprendió a cambiar el hecho de que las caras frontales se especifican con la ordenación de vértices en sentido de las manecillas del reloj o al contrario utilizando `glFrontFace()`.

Preguntas de repaso

1. ¿Cómo se habilita el culling?
2. ¿Cómo se encuentra la versión actual de OpenGL?
3. De forma predeterminada, ¿la cara frontal de un polígono se dibuja con los vértices acomodados en sentido de las manecillas del reloj o al contrario?
4. ¿Qué se pasa a `glEnable()` para habilitar el suavizado de polígonos?

Por su cuenta

1. Escriba una aplicación que represente una pirámide de cuatro lados (sin incluir la parte inferior). Los lados de la pirámide deben formarse usando un abanico de triángulos y la parte inferior debe hacerse con una franja de triángulos. El render de todos los polígonos debe hacerse utilizando vertex buffer objects y cada vértice debe tener un color distinto.

Transformaciones y matrices

Ahora es el momento de tomar un breve descanso, dejaremos un momento de aprender cómo crear objetos *en* el mundo para enfocarnos en el aprendizaje de cómo mover los objetos *alrededor* del mundo. Éste es un ingrediente vital para la generación de mundos realistas en juegos en 3D; sin él, las escenas tridimensionales que usted crea serían estáticas, aburridas y no-interactivas en su totalidad. OpenGL hace que sea fácil para el programador mover objetos mediante una serie de transformaciones de coordenadas que se estudian en este capítulo. Asimismo, usted encontrará la forma de utilizar sus propias matrices con OpenGL, lo que le da el poder de manipular los objetos de muchas maneras diferentes.

En este capítulo, usted aprenderá acerca de:

- Los fundamentos de las transformaciones de coordenadas
- Las transformaciones de cámara y visualización
- Las matrices y pilas de matrices en OpenGL
- Las proyecciones
- El uso de sus propias matrices con OpenGL

Cómo entender las transformaciones de coordenadas

Haga a un lado este libro y deje de leer por un momento. Mire a su alrededor. Ahora, imagine que tiene una cámara en sus manos y está tomando fotografías de su entorno. Puede ser, por

ejemplo, que se encuentre en una oficina y tenga unas paredes, este libro, su escritorio, y quizá su computadora cerca de usted. Cada uno de estos objetos tiene una forma y una geometría descritas dentro de un *sistema de coordenadas locales*, que es único para cada objeto, está centrado en él y no depende de ninguna otra entidad. También tienen algún tipo de posición y orientación en el espacio del mundo. Del mismo modo, usted tiene una posición y una orientación en ese espacio. La relación entre las posiciones de los objetos a su alrededor y su posición y orientación determinan si los objetos están detrás o delante de usted. Mientras toma fotografías de los objetos, la lente de la cámara también tiene algún efecto en el resultado final de las imágenes que está fotografiando. Una lente de zoom hace que los objetos aparezcan más cerca o más lejos de su posición. Usted apunta y hace clic, y la imagen se "representa" en la película de la cámara (o en su tarjeta de memoria si tiene una cámara digital). La cámara y su película tienen configuraciones como el tamaño y la resolución, que ayudan a definir la forma en que se representa la imagen final. Dicha imagen que se ve en una foto, es un producto de la forma en que interactúan la posición de los objetos, la posición de usted, la lente de su cámara y la configuración de ésta para captar las características de los objetos tridimensionales que lo rodean en una imagen bidimensional.

Las transformaciones funcionan de la misma manera. Le permiten mover, girar y manipular objetos en un mundo tridimensional, al tiempo que le permiten proyectar coordenadas tridimensionales en una pantalla bidimensional. A pesar de que las transformaciones parecen modificar un objeto de manera directa, en realidad son una simple transformación del sistema de coordenadas locales del objeto a otro sistema coordenado. Al dibujar escenas en 3D, los vértices pasan a través de cuatro tipos de transformaciones antes de ser finalmente desplegadas en pantalla:

- **Transformación de modelado.** La transformación de modelado mueve los objetos alrededor de la escena y traslada los objetos de las coordenadas locales a las coordenadas del mundo.
- **Transformación de visualización.** La transformación de visualización especifica la ubicación de la cámara y mueve los objetos de las coordenadas del mundo a las coordenadas de cámara o de visualización.
- **Transformación de proyección.** La transformación de proyección define el volumen de visualización y los planos agregados, asimismo transporta los objetos de las coordenadas de visualización a las coordenadas agregadas.
- **Transformación de la ventana de visualización.** Esta transformación traslada las coordenadas agregadas a la ventana de visualización en dos dimensiones, es decir la ventana en su pantalla.

Si bien estas cuatro transformaciones son las estándar en gráficos 3D, OpenGL incluye y combina las transformaciones del modelado y la visualización en una sola transformación del modelo de visualización, la cual se analizará en la sección "Matriz modelview" de este capítulo.

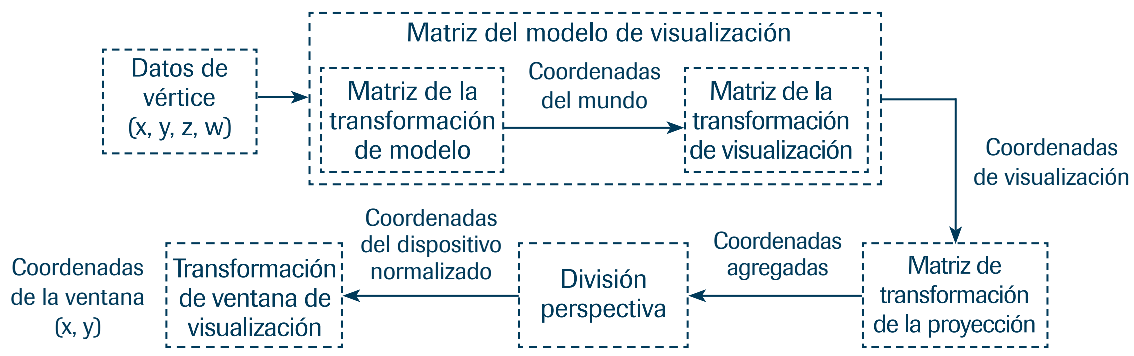
En la tabla 4.1 se muestra un resumen de todas estas transformaciones.

Al escribir sus programas en 3D, recuerde que las transformaciones se ejecutan en un orden

Tabla 4.1 Transformaciones en OpenGL

Transformación	Descripción
Visualización	En los gráficos 3D, especifica la ubicación de la cámara (no es una verdadera transformación en OpenGL)
Modelado	En los gráficos 3D, maneja los objetos que se mueven alrededor de la escena (no es una verdadera transformación en OpenGL)
Proyección	Define el volumen de visualización y los planos agregados
Ventana de visualización	Traslada la proyección de la escena a la ventana de render
Modelo de visualización	Es una combinación de las transformaciones de visualización y modelado

Figura 4.1
Diagrama de la transformación de vértices.

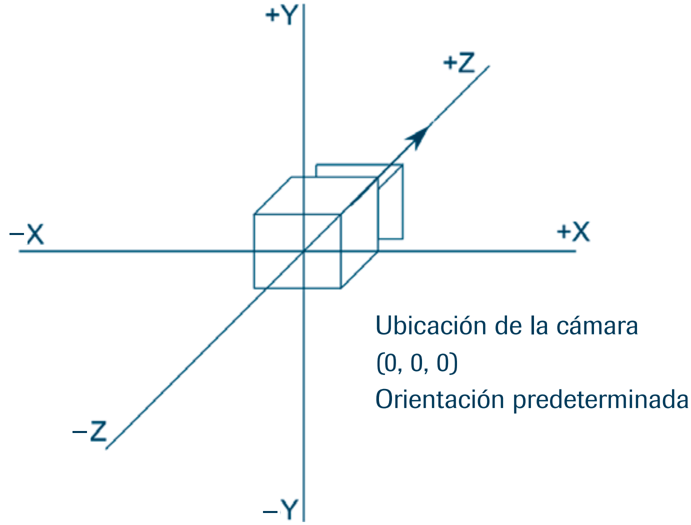


específico; por ejemplo, las transformaciones del modelo de visualización o modelview se ejecutan antes que las de proyección. En la figura 4.1 se muestra el orden general en que se realizan las transformaciones de vértice.

Coordenadas de visualización

Uno de los conceptos más importantes para las transformaciones y la visualización en OpenGL es el concepto de la cámara, o coordenadas de visualización. En los gráficos 3D, la matriz de transformación de la visualización actual, que convierte las coordenadas del mundo a coordenadas de visualización, define la posición y la orientación de la cámara. Por su parte, OpenGL convierte las coordenadas del mundo a coordenadas de visualización con la matriz modelview.

Figura 4.2
La matriz de visualización predeterminada en OpenGL mira hacia el eje z negativo.



Cuando un objeto está en coordenadas de visualización, se conoce la relación geométrica entre el objeto y la cámara, lo que significa que nuestros objetos se posicionan en relación con la posición de la cámara y están listos para ser dibujados adecuadamente. En esencia, puede utilizar la transformación de visualización para mover una cámara sobre el mundo en 3D, mientras que la transformación de modelado mueve los objetos alrededor del mundo. En OpenGL, la cámara predeterminada (o transformación de la matriz de visualización) siempre está orientada para mirar hacia el eje z negativo, como se muestra en la figura 4.2.

Para darle una idea de esta orientación, imagine que está situado en el origen y gira 90 grados hacia la izquierda (alrededor del eje y); ahora estaría de frente al eje x negativo. De manera similar, si usted se colocara en la orientación predeterminada de la cámara y girara 180 grados, quedaría de frente a la dirección positiva de z.

Transformaciones de visualización

La transformación de visualización se utiliza para posicionar y apuntar la cámara. Como ya se indicó, la orientación predeterminada de ésta es la que apunta hacia el eje z negativo y está colocada en el origen (0, 0, 0). Puede mover y cambiar la orientación de la cámara gracias a los comandos de traslación y rotación, lo que, de hecho, manipula la transformación de visualización.

Recuerde que la transformación de visualización debe especificarse antes de cualquier otra transformación de modelado. Esto se debe a que las transformaciones en OpenGL se aplican en orden inverso. Al especificar la transformación de visualización en primer lugar, usted asegura que se aplique después de las transformaciones de modelado.

¿Cómo se crea la transformación de visualización? Primero, es necesario borrar la matriz actual. Esto se logra mediante la función `glLoadIdentity()`, especificada como

```
void glLoadIdentity();
```

Esto hace que la matriz actual sea igual a la *matriz identidad* y es análogo a borrar la pantalla antes de comenzar el render.

Sugerencia

La matriz identidad es aquella en la que los valores de sus elementos diagonales son todos iguales a 1 y todos los demás valores (que no están en la diagonal) son iguales a 0, por lo que, dada la matriz de 4 x 4 M: M(0,0) = M(1,1) = M(2,2) = M(3,3) = 1. Al multiplicar la matriz identidad I por una matriz M se obtiene una matriz igual a M, de modo que I x M = M.

Después de inicializar la matriz actual, puede crear la matriz de visualización de varias formas distintas. Un método consiste en dejar que la matriz de visualización sea igual a la matriz identidad. Esto se traduce en la ubicación y orientación predeterminadas de la cámara, es decir que estaría en el origen y mirando hacia el eje z negativo. Entre los demás métodos se incluyen los siguientes:

- Uso de la función `gluLookAt()` para especificar una línea de visión que se extiende desde la cámara. Ésta es una función que encapsula un conjunto de comandos de traslación y rotación, y que se analizará más adelante en este capítulo dentro de la sección "Uso de `gluLookAt()`".
- Uso de los comandos de modelado para la traslación y la rotación `glTranslate()` y `glRotate()`. Estos comandos se describen con más detalle en la sección "Uso de `glRotate()` y `glTranslate()`" de este mismo capítulo; por ahora, basta con saber que este método mueve los objetos en el mundo con relación a una cámara fija.
- Creación de sus propias rutinas que utilizan las funciones de traslación y rotación para su propio sistema de coordenadas (por ejemplo, las coordenadas polares de una cámara que se encuentra en órbita alrededor de un objeto). En el presente capítulo se analizará este concepto en la sección "Creación de sus propias rutinas personalizadas".

Transformaciones de modelado

Las transformaciones de modelado permiten colocar y orientar un modelo al moverlo, rotarlo y escalarlo. Puede realizar estas operaciones una a la vez o como una combinación de eventos.

En la figura 4.3 se ilustra una integración de las tres operaciones que pueden utilizarse en objetos:

Figura 4.3
Las tres transformaciones de modelado.

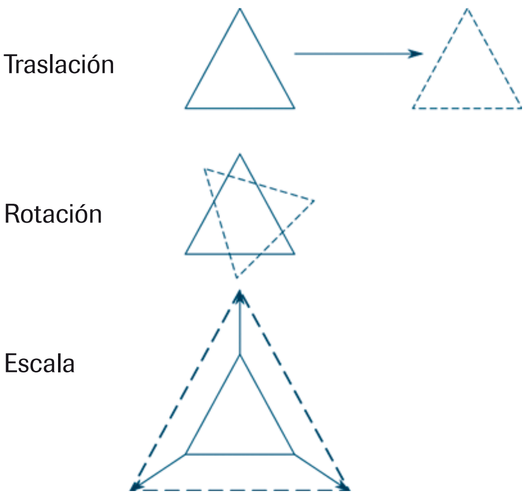
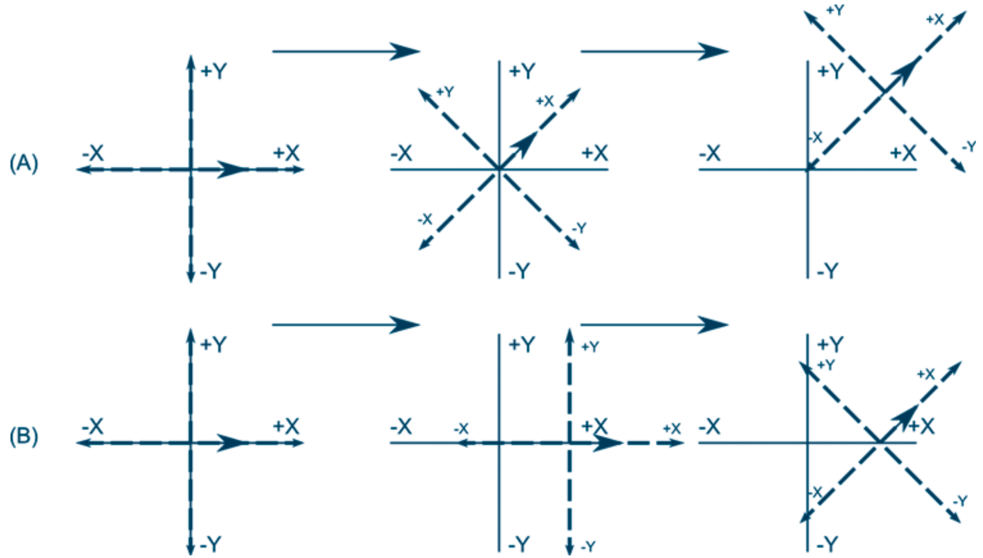


Figura 4.4
a) Una rotación seguida de una traslación. b) Una traslación seguida de una rotación.



- **Traslación.** Esta operación es el acto de mover un objeto a lo largo de un vector específico.
- **Rotación.** Sucede cuando un objeto se gira alrededor de un vector.
- **Escala.** Esta operación se utiliza al aumentar o disminuir el tamaño de un objeto. Con la escala es posible especificar valores diferentes para ejes distintos. Lo anterior le proporciona la capacidad de estirar y encoger objetos de una manera no uniforme.

El orden en el que se especifican las transformaciones de modelado es muy importante para la representación final de la escena. Por ejemplo, como se muestra en la figura 4.4, la rotación seguida de la traslación de un objeto tiene un efecto completamente diferente a la traslación seguida por la rotación. Digamos que usted tiene una flecha situada en el origen que descansa sobre el plano x-y, y la primera transformación que aplica es una rotación de 30 grados alrededor del eje z. A continuación, aplica una transformación de traslación de 5 unidades a lo largo del eje x. La posición final del triángulo sería (5, 4.33) con la flecha apuntando a un ángulo de 30 grados desde el eje x positivo. Ahora, intercambiemos el orden y digamos que primero la flecha se traslada cinco unidades a lo largo del eje x. A continuación, la flecha gira 30 grados alrededor del eje z. Después de la traslación, la flecha se encuentra en (5, 0). Cuando se aplica la transformación de rotación, la flecha todavía estaría ubicada en (5, 0), pero estaría apuntando a un ángulo de 30 grados desde el eje x.

Transformaciones de proyección

La transformación de proyección define el volumen de visualización y los planos agregados. Se realiza después de las transformaciones de modelado y visualización. Se puede pensar en la transformación de proyección como en la determinación de qué objetos pertenecen a un volumen de visualización y cómo deberían verse. Es muy parecido a elegir una lente de cámara que se utilizará para ver el mundo. El campo de visión que usted elige al crear la transformación de proyección determina el tipo de lente que tiene. Por ejemplo, un campo de visión muy amplio sería como tener una lente gran angular, donde se podría ver una enorme área de la escena pero sin mucho detalle. Con un campo de visión más pequeño, que sería similar a la lente de un telefoto, usted podría mirar los objetos como si estuvieran más cerca de lo que realmente están.

OpenGL ofrece dos tipos de proyecciones:

- **Proyección en perspectiva.** Este tipo de proyección muestra mundos en 3D exactamente como usted ve las cosas en la vida real. Con una proyección en perspectiva, los objetos que están más lejos parecen más pequeños que aquellos que están más cerca de la cámara.
- **Proyección ortográfica.** Este tipo de proyección presenta objetos en pantalla a tamaño real, sin importar su distancia de la cámara. Esta proyección es útil para el software de CAD, donde los objetos se dibujan con vistas específicas para mostrar sus dimensiones (es decir, vistas frontal, izquierda y superior); también puede usarse para juegos isométricos.

Transformaciones de ventana de visualización

La última transformación es la de *ventana de visualización*. Esta transformación representa las coordenadas agregadas, que se crearon mediante la transformación de perspectiva sobre la superficie de render en su ventana. Se puede pensar en la transformación de la ventana de visualización como una forma de determinar si la imagen final debe ser ampliada o reducida, dependiendo del tamaño de la superficie de render.

Funciones fijas de OpenGL y matrices

Ahora que ha aprendido acerca de las diversas transformaciones que participan en OpenGL, daremos un vistazo a la forma en que se pueden aprovechar. Para todos los cálculos matemáticos, las transformaciones en OpenGL se basan en la *matriz*. Como pronto se verá, OpenGL tiene algo llamado *pila de matrices*, que es útil para la construcción de modelos complicados formados con muchos objetos simples. En esta sección se estudiarán cada una de las transformaciones y se profundizará en la pila de matrices.

Antes de comenzar, vale la pena señalar que la pila de matrices se considera obsoleta en OpenGL 3.0, sobre todo por que es una funcionalidad que puede proporcionar una biblioteca de terceros y cuando se migra hacia un diagrama programable (vea el capítulo 6, "Cambiando a un diagrama (pipeline) programable") debe tener cuidado con sus propias matrices. Sin embargo, la actual funcionalidad de matriz se seguirá usando por un tiempo y se emplea en la mayor parte del código que está disponible en el momento en que se escribió este texto. Además, los conceptos de la pila de matrices y las diferentes matrices que se analizarán son vitales para los gráficos por computadora en 3D. El desarrollador sólo tiene la responsabilidad de su manejo. Ahora, comenzamos por dar un vistazo a la *matriz modelview*.

Matriz modelview

La *matriz modelview* define el sistema de coordenadas que se utiliza para colocar y orientar los objetos. Esta matriz de 4 x 4 puede transformar vértices o bien puede ser transformada por otras matrices. Los vértices se transforman al multiplicar un vector de vértices por la matriz modelview, lo que resulta en un nuevo vector de vértices que ha sido transformado. La matriz modelview en sí misma puede ser transformada al multiplicarla por otra matriz de 4x4.

Antes de llamar a cualquier comando de transformación, debe especificar si desea modificar la matriz modelview o la matriz de proyección. La modificación de cualquiera de ellas se realiza mediante la función OpenGL `glMatrixMode()`, que se define como

```
void glMatrixMode(GLenum mode);
```

Para modificar la matriz modelview, utilice el argumento `GL_MODELVIEW` que establece a dicha matriz como la matriz actual. Lo anterior significa que será modificada con los comandos de transformación posteriores. Para hacer esto se escribe:

```
void glMatrixMode(GL_MODELVIEW);
```

Entre los argumentos restantes para glMatrixMode() se incluyen GL_PROJECTION, GL_COLOR o GL_TEXTURE. GL_PROJECTION se usa para especificar la matriz de proyección; GL_COLOR se utiliza para indicar el color de la matriz, que no se estudiará en este libro, y GL_TEXTURE se emplea para indicar la textura de la matriz, que se analizará en el capítulo 7, "Texturas".

Por lo general, al principio del ciclo de render, usted tendrá que reiniciar la matriz modelview a la posición (0, 0, 0) y orientación predeterminadas (mirando hacia el eje z negativo). Para ello, se llama la función glLoadIdentity(), que llama a la matriz identidad como si fuera la matriz modelview actual; con esto se colocará la cámara en el origen del mundo y con la orientación por defecto. A continuación se muestra un fragmento de cómo puede restablecerse la matriz modelview:

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity(); //Restablece la matriz modelview
// ... hace otras transformaciones
```

Traslación

La traslación le permite mover un objeto de una posición a otra dentro del mundo. La función de OpenGL glTranslate() realiza esta función y se define de la siguiente manera:

```
void glTranslate{fd}(TYPE x, TYPE y, TYPE z);
```

Los parámetros x, y, z especifican la extensión de la traslación a lo largo de los ejes x, y, z, respectivamente. Por ejemplo, si se ejecuta el comando

```
glTranslatef(3.0f, 1.0f, 8.0f);
```

cualesquiera objetos especificados posteriormente se trasladarán tres unidades a lo largo del eje x positivo, una unidad a lo largo del eje y positivo, y ocho unidades a lo largo del eje z positivo, hasta una posición final de (3, 1, 8).

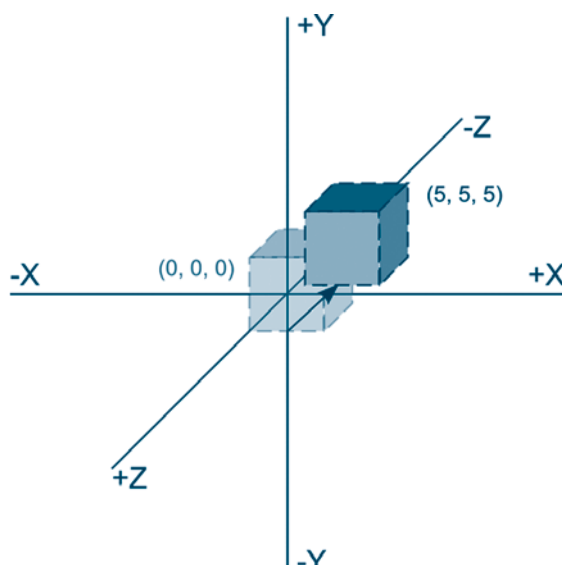
Suponga que usted desea mover un cubo desde el origen hasta la posición (5, 5, 5). En primer lugar, debe llamar a la matriz modelview y restablecerla a la matriz identidad, para iniciar en el origen (0, 0, 0). A continuación, realiza la transformación de traslación sobre la matriz actual hasta la posición (5, 5, 5) antes de llamar a la función renderCube(). En código, esto se escribe así

```
glMatrixMode(GL_MODELVIEW);      //Establece como matriz actual a modelview
glLoadIdentity();                //Restablece modelview a la matriz identidad
```

```
glTranslatef(5.0f, 5.0f, 5.0f); //Mueve a (5, 5, 5)
renderCube ();                  //Dibuja el cubo
```

En la figura 4.5 se ilustra la manera en que se ejecuta este código.

Figura 4.5
Traslación de un cubo desde el origen hasta (5, 5, 5).



¿Qué tal un ejemplo de traslación? En el CD, dentro de la carpeta del capítulo 4, encontrará un ejemplo llamado Translation que ilustra una traslación oscilatoria muy simple a lo largo del eje z. El ejemplo dibuja un plano cuadrado en el origen, pero como el sistema de coordenadas del mundo se está trasladando, el plano cuadrado parece acercarse y alejarse de la vista. A continuación se presenta el código de la función `prepare()`, que realiza la lógica de oscilación:

```
//Si nos movemos en la dirección -z, se disminuye la posición
if(m_currentDirection == FORWARD) {
    m_zPosition -= speed*dt;
}else {//en caso contrario nos movemos hacia atrás por lo que la posición se
incrementa
    m_zPosition += speed*dt;
}

//Si llegamos al límite cercano o lejano, se invierte la dirección
if(m_zPosition >= nearLimit) {
```

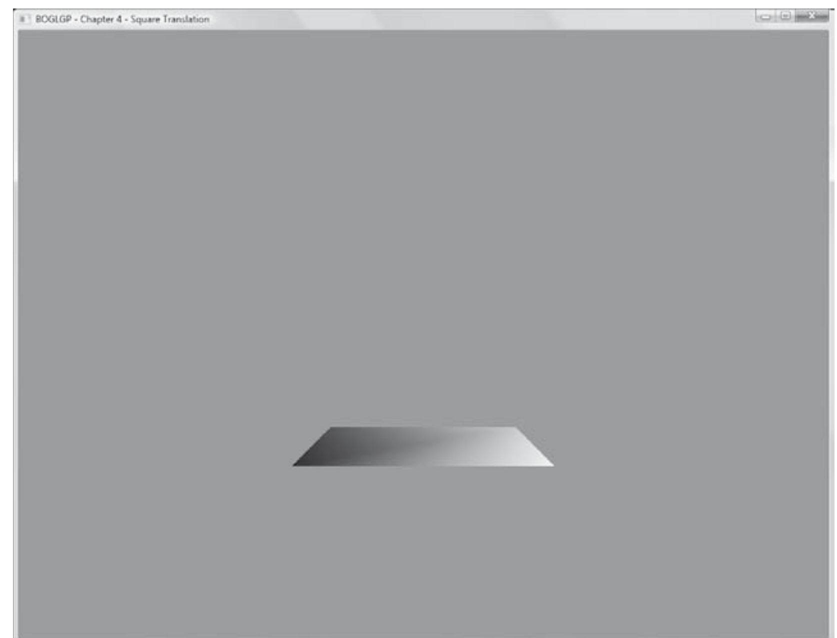
```
    m_currentDirection = FORWARD;
    m_zPosition = nearLimit;
}else if(m_zPosition <= farLimit) {
    m_currentDirection = BACKWARD;
    m_zPosition = farLimit;
}
```

El valor de `dt` en el código anterior es el tiempo transcurrido desde el último fotograma. Si se hacen transformaciones dependientes del tiempo, todo se mantiene funcionando a la misma velocidad sin importar la potencia de la computadora. El código aumenta o disminuye el valor utilizado para trasladar el mundo a lo largo del eje z, en función de la "dirección" que se está desplazando. Cuando el valor de la traslación llega a un extremo (definido como las variables `nearLimit` y `farLimit`), entonces se cambia la "dirección" de la traslación. Este código en la función `prepare()` se llama antes de la función `render()`, que se especifica de la manera siguiente:

```
void Example:: render()
{
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
    //Llama a la matriz identidad (restablece a la posición y orientación pre-
determinadas)
    glLoadIdentity();
    //Vincula el arreglo de color y establece el puntero de color apuntando
hacia éste
    glBindBuffer(GL_ARRAY_BUFFER, m_colorBuffer);
    glColorPointer(3, GL_FLOAT, 0, 0);
    //Vincula el arreglo de vértices y establece el puntero del vértice apun-
tando hacia éste
    glBindBuffer(GL_ARRAY_BUFFER, m_vertexBuffer);
    glVertexPointer(3, GL_FLOAT, 0, 0);
    //Traslada utilizando nuestra variable zPosition
    glTranslatef(0.  0, 0. 0, m_zPosition);
    //Dibuja el cuadrado como una franja de triángulo
    glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
}
```

La función `render()` es muy simple. Después de borrar los búferes de color y profundidad, llamamos a la matriz identidad para inicializar el mundo con la posición y orientación predeterminadas; después vinculamos nuestros búferes de vértice y trasladamos a lo largo del eje z usando el valor determinado en la función `prepare()`, y por último dibujamos el cuadro. La

Figura 4.6
Captura de pantalla en el ejemplo de traslación de un cuadro.



ejecución resultante muestra un plano que se mueve hacia delante y hacia atrás a lo largo del eje z. En la figura 4.6 se presenta una captura de pantalla para este ejemplo.

Rotación

La rotación en OpenGL se logra a través de la función `glRotate()`, que se define como

```
void glRotate{fd}(TYPE angle, TYPE x, TYPE y, TYPE z);
```

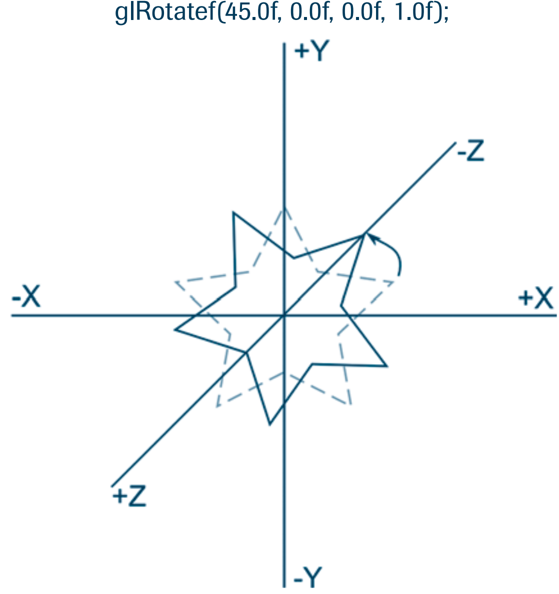
Con esta función, se realiza una rotación alrededor del vector especificado por los parámetros `x, y, z` El ángulo de rotación se especifica mediante `angle` y se mide en grados en sentido contrario a las manecillas del reloj.

Por ejemplo, si desea girar 135 grados alrededor del eje y en sentido contrario a las manecillas del reloj, debe utilizar lo siguiente:

```
glRotatef(135.0f, 0.0f, 1.0f, 0.0f);
```

El valor de `1.0f` para el argumento `y` especifica un vector que apunta en la dirección del eje y positivo. En la figura 4.7 se ilustra cómo funciona `glRotate()`.

Figura 4.7
La función glRotatef() toma el ángulo de rotación y un vector para el eje de rotación como parámetros.



Si se desea girar en el sentido de las manecillas del reloj, se puede establecer el ángulo de rotación como un número negativo. Para girar 135 grados alrededor del eje y en sentido de las manecillas del reloj, se utiliza el siguiente código:

```
glRotatef(-135.0f, 0.0f, 1.0f, 0.0f);
```

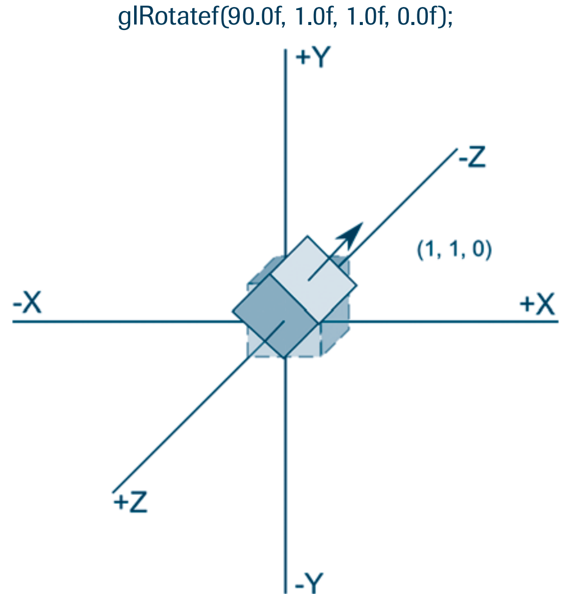
¿Qué pasa si desea girar alrededor de un eje arbitrario? Esto puede lograrse al especificar el vector del eje arbitrario en los parámetros x, y, z. Si se dibuja una línea desde el origen relativo hasta el punto representado por (x, y, z), es posible ver el eje arbitrario alrededor del cual girará. Por ejemplo, si gira 90 grados alrededor del eje especificado por el vector (1, 1, 0), usted rotará en torno al eje relativo que va desde el origen hasta el punto (1, 1, 0). En código, esto se ve de la manera siguiente:

```
glRotatef (90.0f, 1.0f, 1.0f, 0.0f);
```

En la figura 4.8 se ilustra cómo funciona esto.

El giro alrededor de un eje único está bien, pero la mayoría de las aplicaciones giran sus objetos alrededor de varios ejes. Al hacer esto, el orden en el que se especifican las rotaciones es muy importante porque cada rotación aplicada cambia el sistema de coordenadas locales del

Figura 4.8
Rotación alrededor de un eje arbitrario.



objeto. Por ejemplo, si gira un objeto 60 grados sobre el eje x y luego lo rota 45 grados sobre el eje y en llamadas subsecuentes a `glRotate()`, entonces la orientación resultante del objeto será producto de las dos rotaciones que ocurren una tras otra dentro del contexto del sistema de coordenadas locales del objeto. La primera rotación se aplicará de la manera esperada, y el objeto girará 60 grados alrededor del eje x. Sin embargo, la segunda rotación sobre el eje y no será en el contexto del sistema de coordenadas del mundo. En cambio, la rotación sobre el eje y se produce en el contexto del sistema de coordenadas locales del objeto. Como éste ya ha sido girado 60 grados respecto al eje x, el nuevo eje y también habrá girado 60 grados en sentido contrario. Su segunda rotación respecto al eje y en realidad será dentro de esta nueva configuración. Veamos un ejemplo; tal vez así esta explicación tendrá más sentido.

En el CD, dentro de la carpeta del capítulo 4, se incluye un ejemplo titulado Rotation. Asimismo, en la figura 4.9, se muestra una captura de pantalla de este ejemplo. Si usted construye y ejecuta el ejemplo, verá el mismo plano que hemos creado en el ejemplo de traslación, excepto que esta vez está girando sobre el origen en vez de trasladarse a lo largo del eje z. Además, en este ejemplo se pueden observar dos conjuntos de líneas que representan los ejes x, y del sistema de coordenadas.

Las líneas blancas representan los ejes x, y del sistema de coordenadas del mundo, mientras que las líneas amarillas representan los mismos ejes pero en el sistema de coordenadas locales del objeto. La parte importante de este ejemplo está representada por las siguientes líneas en el método `render()`:

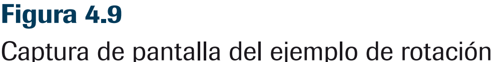
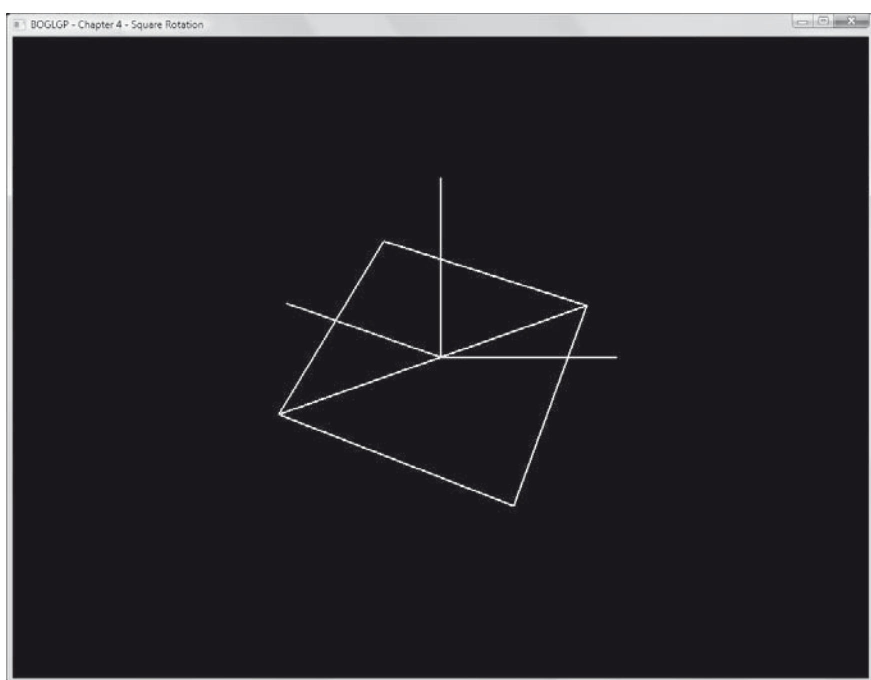


Figura 4.9
Captura de pantalla del ejemplo de rotación.




```
glRotatef(m_xRotation, 1.0f, 0.0f, 0.0f);
glRotatef(m_yRotation, 0.0f, 1.0f, 0.0f);
//Dibuja el cuadrado como una franja de un triángulo
glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
```

Durante la ejecución del ejemplo notará que el plano siempre está girando en forma adecuada alrededor del mismo eje x del mundo, lo que también parece alterar la ubicación del eje y. Esto se debe a que la rotación alrededor del eje x se especifica en primer lugar, mientras todavía estamos en la orientación original del sistema de coordenadas del mundo. Sin embargo, una vez que giramos a lo largo del eje x, la orientación del sistema de coordenadas del mundo cambia para reflejar la rotación, y como se puede ver en el ejemplo, la orientación del eje y cambia (la línea amarilla perpendicular al plano). Entonces, cuando giramos alrededor del eje y, la rotación se produce en la nueva orientación que se ha creado como consecuencia de la rotación del eje x. Esperemos que, a través de este ejemplo, se pueda ver hasta qué punto es importante el orden de rotación en torno a ejes diferentes. Tómese algún tiempo para modificar el ejemplo de la rotación de un cuadrado para ver cómo los diferentes órdenes de rotación pueden afectar el resultado rotacional final de un objeto.

Escalas

Una escala, en su definición más simple, aumenta o disminuye el tamaño de un objeto o sistema de coordenadas. En otras palabras, al utilizar las operaciones de escalado, las coordenadas de los vértices de un objeto se multiplican por un factor de escala para cada eje. Esto significa que si normalmente usted coloca un vértice en la ubicación (1, 1, 1) sin escala, entonces al aplicar un factor de escala de 2.0 a lo largo de cada eje se colocaría al vértice en la ubicación (2, 2, 2). La escala se lleva a cabo en OpenGL a través de la función `glScale()`, que se define como

```
void glScale{fd}(TYPE x, TYPE y, TYPE z);
```

Los valores pasados a los parámetros x, y, z especifican el factor de escala a lo largo de cada eje. Por ejemplo, esta línea aplica un factor de escala de 2.0 a lo largo de cada eje:

```
glScalef(2.0f, 2.0f, 2.0f);
```

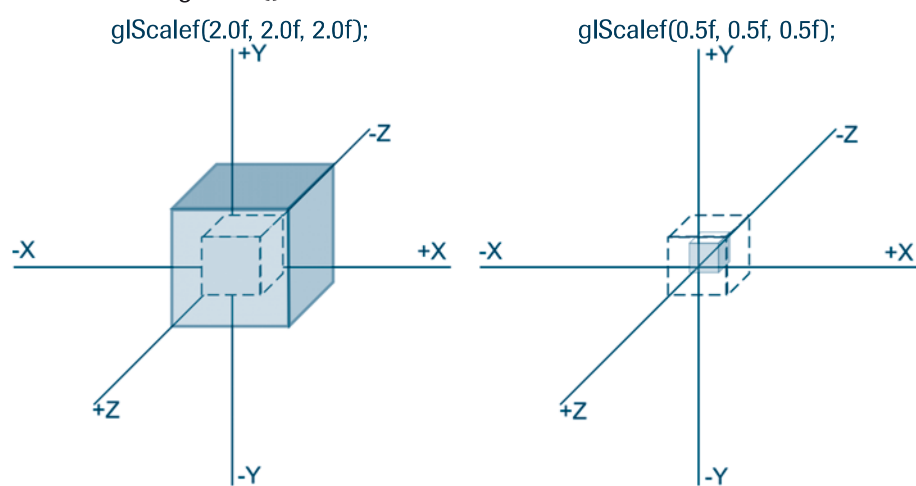
Si dibujara un cubo unitario de 1 x 1 x 1 después de ejecutar la línea anterior, entonces el cubo en realidad se dibujaría de 2 x 2 x 2. Ahora, digamos que toma ese cubo y quiere el doble de su anchura (eje x), sin modificar su altura (eje y) y profundidad (eje z). Entonces, debería utilizar lo siguiente:

```
glScalef(2.0f, 1.0f, 1.0f);
```

¿Qué pasa si quisiera reducir el tamaño de un objeto? Bueno, como los factores de escala se multiplican por los vértices, basta con elegir un valor inferior a uno, así:

```
glScalef(0.5f, 0.5f, 0.5f);
```

Figura 4.10
El efecto de llamar a glScale().



Esta línea reducirá un objeto a la mitad de su tamaño original. Un valor de 0.2 lo reduciría en una quinta parte, 0.1 en una décima parte y así sucesivamente. Si usted asigna un valor negativo a `glScale()` entonces, el objeto se volteará, lo cual es útil para simular reflexiones (en el capítulo 12 puede ver un ejemplo de esto). Si se establece un factor de escala de 1.0, entonces el eje al que pertenece el factor de escala no sufrirá ninguna modificación. Como ya habrá deducido, la escala es equivalente a multiplicar por el factor de la escala. Los valores entre 0.0 y 1.0 reducirán el objeto, y los valores superiores a 1.0 aumentarán el tamaño del objeto. En la figura 4.10 se ilustra la función `glScale()`.

En los ejemplos que se han presentado hasta ahora en este capítulo, usted ha visto un objeto moverse por la traslación, y a un objeto girar para la rotación. Así que, naturalmente, ahora verá un ejemplo de escala donde un objeto se contraerá y se expandirá. Dentro de la carpeta del capítulo 4 en el CD encontrará un ejemplo titulado Scaling. Al observar la función `prepare()` en la clase `Example`, verá lo siguiente:

```
//Si estamos aumentando la escala, entonces incrementamos la variable de es-
cala
if(m_increasing) {
    m_scale += speed * dt;
} else { // si no es así, se disminuye
    m_scale -= speed * dt;
}
//Si llegamos al límite mínimo o máximo, se invierte el sentido de la escala
if(m_scale>= maxLimit) {
    m_increasing = false;
} else if(m_scale <= minLimit) {
    m_increasing = true;
}
```

Antes de hacer un render de cada fotograma, la función `prepare()` aumenta o disminuye nuestro factor de escala dentro de un rango de 0.1 a 2.0 unidades. A continuación, pasamos el factor de escala a la función `glScale()` en la función `render()`:

```
void Example:: render ()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity ();

    //Vincula la matriz de color y establece el puntero de color para direc-
cionarlo a éste
    glBindBuffer(GL_ARRAY_BUFFER, m_colorBuffer);
    glColorPointer(3, GL_FLOAT, 0, 0);
```

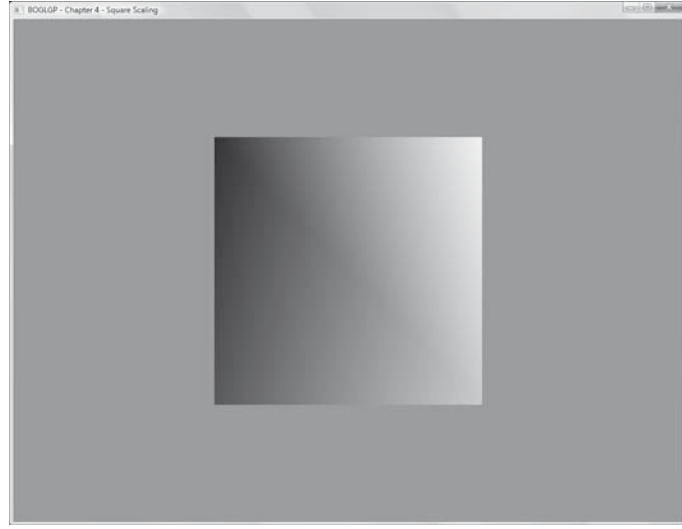
```
    //Vincula el conjunto de vértices y establece el puntero del vértice para
direccionarlo a éste
    glBindBuffer(GL_ARRAY_BUFFER, m_vertexBuffer);
    glVertexPointer(3, GL_FLOAT, 0, 0);

    glTranslatef(0.0f, 0.0f,-10.0f); //Mueve 10 unidades hacia atrás
    glScalef(m_scale, m_scale, m_scale); // Escalado usando nuestra variable
de escala
    //Dibuja el cuadro como una franja de triángulo
    glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
}
```

La función `render()` configura la cámara 10 unidades atrás y rotada sobre el eje z de modo que podamos ver el plano desde arriba. Después se llama a la función `glScale()`, pasando el factor de escala a los tres parámetros del eje. El resultado es un plano que aumenta y disminuye de tamaño según el valor del factor de escala. Aunque no pueda ver cómo el plano va cambiando de forma, la figura 4.11 es una captura de pantalla del ejemplo en el que se escala un cuadro.

Pilas de matrices

La matriz modelview con la que hemos estado practicando hasta ahora, en realidad es sólo una matriz en la parte superior de una pila de matrices, que naturalmente se denomina pila de matrices de OpenGL. Hay cuatro tipos de pilas de matrices en OpenGL:

Figura 4.11
Captura de pantalla del ejemplo para modificar la escala de un cuadro. ¡Emocionante!

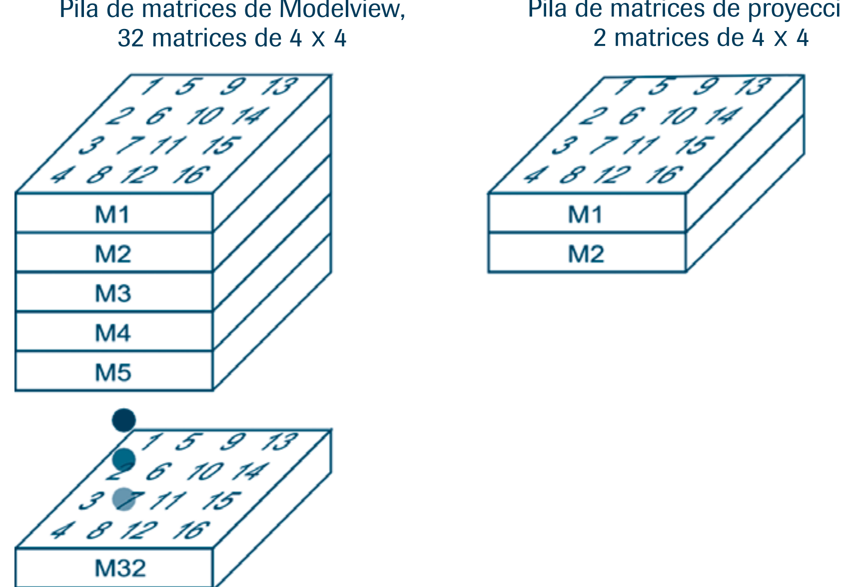


- La pila de matrices modelview
- La pila de matrices de proyección
- La pila de matrices de color
- La pila de matrices de textura

La matriz modelview es aquella que está en la parte superior de la pila de matrices modelview, asimismo la matriz de proyección es la que se encuentra en la parte superior de su pila de matrices correspondiente. En la figura 4.12 se proporciona más información sobre estas pilas de matrices.

Figura 4.12

Las pilas de matrices modelview y de proyección se componen de 32 matrices de 4 x 4 y 2 matrices de 4x4, respectivamente, para la implementación de Microsoft OpenGL.



La pila de matrices de textura se utiliza para la transformación de coordenadas de textura, y la matriz de color puede utilizarse para modificar los colores.

La pila de matrices modelview le permite guardar el estado actual de la matriz de transformación, realizar otras transformaciones, y luego regresar a la matriz de transformación guardada sin tener que almacenar o calcular la matriz de transformación por su cuenta. Las pilas de matrices de proyección, textura y color le permiten hacer lo mismo.

Al usar la pila de matrices modelview, en esencia usted es capaz de transformar objetos de un sistema de coordenadas a otro, aunque conserva la capacidad de regresar al sistema de coordenadas original. Por ejemplo, si nos situamos en el punto (10, 5, 7) y luego introducimos la matriz modelview en la pila, nuestra matriz de transformación actual se restablece en el sistema de

coordenadas locales alrededor del punto (10, 5, 7). Esto significa que cualquier transformación que hagamos ahora se basará en el sistema de coordenadas en (10, 5, 7). Entonces, si nos trasladamos 10 unidades sobre el eje x positivo con `glTranslate(10.0f, 0.0f, 0.0f)`, estamos en la posición (10, 0, 0) en la matriz de transformación actual, pero en el mundo estamos posicionados en (20, 5, 7). Cuando la pila de matrices se desactiva, volvemos a la transformación de la matriz original y por lo tanto al sistema de coordenadas original, es decir que estaremos otra vez colocados en (10, 5, 7).

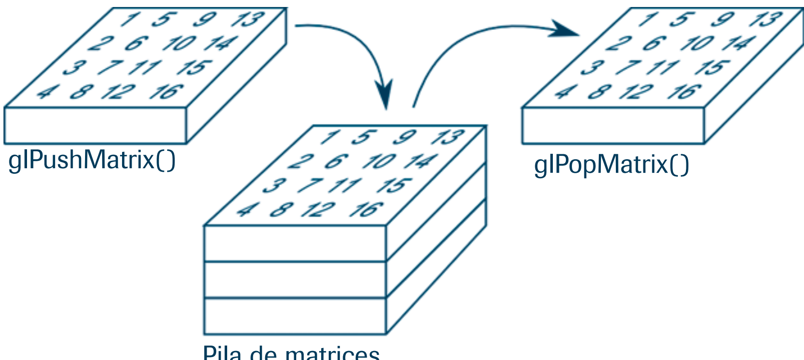
Las dos funciones que le permiten introducir y extraer de las pilas de matrices son `glPushMatrix()` y `glPopMatrix ()`. La función `glPushMatrix()` copia la matriz actual y la introduce en la pila, y se define como:

```
void glPushMatrix();
```

Si se introducen demasiadas matrices en la pila, OpenGL genera un error del tipo `GL_STACK_OVERFLOW`. Se garantiza que la pila de matrices modelview tenga al menos una profundidad de 32 matrices, mientras que todas las demás pilas de matrices tienen una profundidad garantizada de 2. Para saber si su aplicación soporta pilas más grandes, es necesario llamar a las funciones

```
glGet() con GL_MAX_MODELVIEW_STACK_DEPTH, GL_MAX_PROJECTION_STACK_
DEPTH, GL_MAX_COLOR_STACK_DEPTH o GL_MAX_TEXTURE_STACK_DEPTH.
```

Figura 4.13
Introducción y extracción de la pila de matrices.



La función `glPopMatrix()` extrae la matriz superior de la pila y descarta su contenido. Todas las demás matrices en la pila suben una posición. `glPopMatrix()` se define como

```
void glPopMatrix();
```

Si intenta utilizar esta función cuando sólo hay una matriz en la pila, OpenGL generará un error del tipo `GL_STACK_UNDERFLOW`.

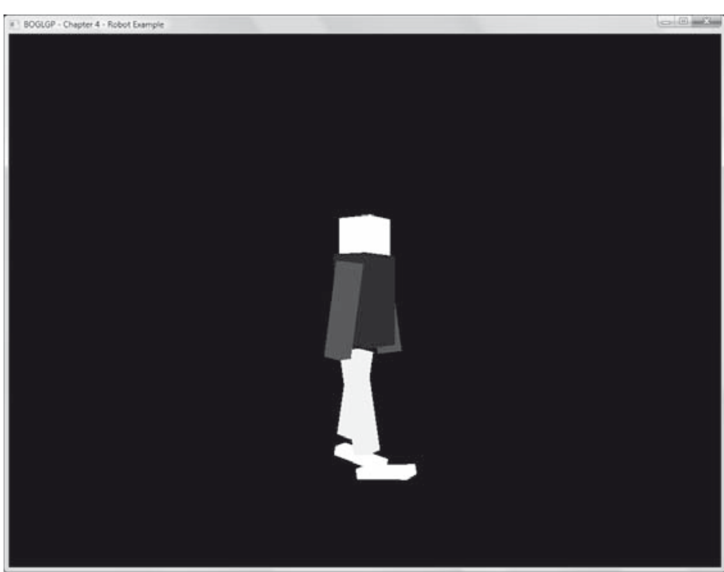
En la figura 4.13 se muestra cómo operan las funciones `glPushMatrix()` y `glPopMatrix()` sobre la pila de matrices.

Ejemplo del robot

En el CD se encuentra el código fuente de un demo de OpenGL llamado "Robot Example", el cual muestra a un robot animado caminando, alrededor del cual gira la cámara. El robot está construido a partir de cubos que se escalan a diferentes formas y tamaños para representar los brazos, las piernas, los pies, el torso y la cabeza del robot. Las funciones `glPushMatrix()` y `glPopMatrix()`se utilizan para colocar las partes del cuerpo del robot en coordenadas relativas al centro del mismo. Preste atención especial a estas funciones a medida que avanza a través del código fuente.

En la figura 4.14 se muestra una captura de pantalla del demo del robot.

Figura 4.14
Una captura de pantalla del demo del Robot.



Mientras navega a través del código fuente, hay dos funciones en las que debe enfocarse. La primera corresponde al método `animate()` en la clase `Robot`:

```
void Robot:: animate(float dt)
```

```
{
    const float armRotationSpeed = 20.0f;
    const float maxArmAngle = 15.0f;
    const float legRotationSpeed = 30.0f;
    const float maxLegAngle = 15.0f;
    for (unsigned int side = 0; side < MAX_SIDES; ++side){
    //brazos
      if (m_armStates [side] == FORWARD_STATE) {
          m_armAng1es [side] += armRotationSpeed*dt;
      } else {
          m_armAngles[side] -= armRotationSpeed*dt;
      }
      //cambio de estado si se exceden los ángulos
      if (m_armAngles[side] >= maxArmAngle) {
          m_armStates[side] = BACKWARD_STATE;
      } else if (m_armAngles[side] <= -maxArmAngle) {
      m_armStates[side] = FORWARD_STATE;
      }

      //Piernas
      if(m_legStates[side] == FORWARD_STATE) {
          m_legAngles[side] += legRotationSpeed*dt;
      } else {
          m_legAngles [side] -= legRotationSpeed*dt;
      }
    //Cambia el estado si los ángulos se exceden
      if(m_legAngles[side]>= maxLegAngle){
          m_legStates[side] = BACKWARD_STATE;
      } else if (m_legAngles[side] <=-maxLegAngle){
          m_legStates[side] = FORWARD_STATE;
      }
    }
}
```

El método `animate()` modifica el ángulo de rotación de las piernas y los brazos del robot, asimismo determina la dirección de cada brazo y pierna que está en movimiento. Como se ha visto en los ejemplos anteriores, utilizamos el parámetro de tiempo delta (dt) para mantener la velocidad independiente de los fotogramas en movimiento. Los arreglos `m_armAngles` y `m_legAngles` almacenan los ángulos de cada brazo y pierna, respectivamente (el arreglo de índice 0 es para el miembro izquierdo, y el de índice 1 para el derecho); los arreglos `m_armStates` y `m_legStates` almacenan, de manera respectiva, la dirección actual de cada brazo y pierna que están en movimiento.

El método que realiza el render principal es, probablemente, el más interesante. Éste es el método `render()` de la clase Robot:

```
void Robot::render(float xPos, float yPos, float zPos)
{
    glPushMatrix();
      glTranslatef(xPos, yPos, zPos); //mueve a (0, 0, -30)
      //Dibuja las partes de la cabeza y el torso
      renderHead (0.0f, 3.5f, 0.0f);
      renderTorso(0.0f, 0.0f, 0.0f);
    //Mueve el brazo izquierdo lejos del torso y gíralo para obtener el efec-
to de “caminando”

    glPushMatrix();
      glTranslatef(0.0f,-0.5f, 0.0f);
      glRotatef(m_armAngles[LEFT], 1.0f, 0.0f, 0.0f);
      renderArm (2.0f, 0.0f, 0.0f);
    glPopMatrix();
    //Mueve el brazo derecho lejos del torso y gíralo para obtener el efecto
de “caminando”
    glPushMatrix();
      glTranslatef(0.0f, -0.5f, 0.0f);
      glRotatef(m_armAng1es[RIGHT], 1.0f, 0.0f, 0.0f);
      renderArm(-2.0f, 0.0f,-0.0f);
    glPopMatrix();
    //Mueve la pierna izquierda lejos del torso y gírala para obtener el
efecto de “caminando”
    glPushMatrix();
      glTranslatef(0.0f, -0.5f, 0.0f);
      glRotatef(m_legAngles[left], 1.0f, 0.0f, 0.0f);
      renderLeg(-1.0f, -5.0f, -0.5f);
    glPopMatrix();
      //Mueve la pierna derecha lejos del torso y gírala para obtener el
      efecto de “caminando”
    glPushMatrix();
      glTranslatef(0.0f, -0.5f, 0.0f);
      glRotatef(m_legAngles[RIGHT], 1.0f, 0.0f, 0.0f);
      renderLeg(1.0f, -5.0f, -0.5f);
    glPopMatrix();
  glPopMatrix(); //Regresa al sistema coordenado original
}
```

El método `renderRobot()` dibuja todo el robot en la posición especificada. Para simplificar el código, el método llama a otros diversos métodos donde cada uno hace cierta parte del cuerpo del robot. Estos métodos son `renderHead()`, `renderTorso()`, `renderArm()` y `renderLeg()`. Éste último a su vez llama a otro método, `renderFoot()`. A su vez, cada uno de estos métodos dibuja su parte respectiva en la posición especificada, en relación con la posición del robot en sí porque usamos las funciones `glPush Matrix()` y `glPopMatrix()` para posicionar y girar cada parte del robot.

Proyecciones

Hemos mencionado varias veces las transformaciones de proyección e incluso las hemos utilizado en el código, así que es momento de analizar su funcionamiento. Como hemos señalado, hay dos clases generales de transformaciones de proyección disponibles en OpenGL: la ortográfica (o paralela) y la de perspectiva. A continuación nos ocuparemos a detalle de ambas.

Al establecer una transformación de proyección, en realidad se crea un volumen de visualización, que sirve para dos propósitos. El primero es que el volumen de visualización define una serie de planos de corte, que determinan la parte de su mundo en 3D que es visible en un momento dado. Los objetos que están fuera de este volumen no se dibujan.

El segundo propósito del volumen de visualización es determinar cómo se dibujan los objetos. Esto depende de la forma del volumen, que es la principal diferencia entre las proyecciones ortográfica y de perspectiva.

Sin embargo, antes de especificar cualquier tipo de transformación de proyección es necesario que se asegure de que la matriz de proyección sea la pila de matrices actualmente seleccionada. Como se vio anteriormente con la matriz modelview, esto se hace mediante una llamada a `glMatrixMode()`:

```
glMatrixMode(GL_PROJECTION);
```

En la mayoría de los casos, usted querrá agregar a esto una llamada a `glLoadIdentity()` para limpiar cualquier cosa que pueda estar almacenada en la matriz de proyección, de modo que las transformaciones anteriores no se acumulen. A diferencia de la matriz modelview, es raro que se hagan muchos cambios a la matriz de proyección.

Una vez que se ha seleccionado la pila de matrices de proyección, usted está listo para especificar su proyección. Primero estudiaremos las proyecciones ortográficas y después las transformaciones de perspectiva más comunes.

Ortográfica

Como se mencionó antes, las proyecciones ortográficas, o paralelas, son aquellas que no implican corrección de perspectiva. En otras palabras, no se hace ajuste por la distancia que existe con respecto a la cámara; los objetos que se representan en pantalla aparecerán al mismo tamaño

sin importar si están cerca o lejos. Aunque esto puede no parecer tan real como las proyecciones de perspectiva, tiene una serie de usos. De manera tradicional, las proyecciones ortográficas se incluyen en OpenGL para aplicaciones como CAD, pero también pueden utilizarse para juegos en 2D o juegos isométricos.

Para configurar proyecciones ortográficas, OpenGL proporciona la función `glOrtho()`:

```
glOrtho (GLdouble left, GLdouble right, GLdouble bottom, GLdouble top,
GLdouble near, GLdouble far);
```

`left` y `right` especifican la coordenada x de los planos agregados, `bottom` y `top` especifican la coordenada y de los planos agregados, y `near` y `far` especifican la distancia a la coordenada z de los planos agregados. En conjunto, estas coordenadas especifican un volumen de visualización en forma de caja. De manera más precisa, los planos opuestos son paralelos entre sí, y los planos adyacentes son perpendiculares.

Como las proyecciones ortográficas se utilizan normalmente para crear escenas en 2D, la biblioteca de utilidades de OpenGL proporciona una rutina adicional para configurar proyecciones ortográficas en escenas en las que en realidad no se utilizará la coordenada z:

```
gluOrtho2D(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top);
```

`left`, `right`, `bottom` y `top` especifican lo mismo que en `glOrtho()`. El uso de `gluOrtho2D()` es equivalente a llamar `glOrtho()` con una configuración `near` de –1.0 y `far` de 1.0. Cuando se utiliza `gluOrtho2D()`, normalmente se desea enviar vértices que contienen dos componentes (coordenadas x, y), porque la coordenada z no es necesaria.

En este caso, es muy común utilizar coordenadas enteras, así como establecer el volumen para que coincida con las coordenadas x, y de la ventana de visualización.

Perspectiva

Aunque las proyecciones ortográficas pueden ser interesantes, las proyecciones de perspectiva crean escenas que lucen más realistas, y eso es lo que probablemente utilizará con mayor frecuencia. En las proyecciones de perspectiva, a medida que un objeto se aleja del observador, aparece más pequeño en la pantalla, un efecto conocido comúnmente como escorzo. El volumen de visión para una proyección de perspectiva es un frustum, que se parece a una pirámide con la parte superior cortada, donde el extremo más estrecho mira hacia el espectador. Como el otro extremo lejano del frustum es más grande que el extremo cercano, se crea el efecto de escorzo. La forma en que esto funciona es que OpenGL transforma el frustum para que sea un cubo. Esta transformación afecta a los objetos en el interior del frustum, así que los objetos en el extremo ancho del volumen se comprimen más que los que están en el extremo estrecho. Cuanto mayor sea la relación entre los extremos ancho y estrecho, más se reduce un objeto. Si los extremos del frustum tienen un tamaño parecido, no habrá mucha corrección de perspectiva (si son iguales,

no habrá ninguna corrección, que es lo que ocurre con las proyecciones ortográficas).

Hay un par de formas en las que usted puede configurar el frustum de visualización y, por consiguiente, la proyección de perspectiva. La primera que estudiaremos es la siguiente:

```
void glFrustum(GLdouble left, GLdouble right, GLdouble bottom, GLdouble
top, GLdouble near, GLdouble far);
```

`left`, `right`, `top` y `bottom` en conjunto especifican las coordendas x, y en el plano agregado cercano, y `near` y `far` especifican la distancia existente entre los planos agregados cercano y lejano. Así, la esquina superior izquierda del plano agregado cercano está en (left, top, —near), y la esquina inferior derecha se encuentra en (right, bottom, —near). Las esquinas del plano agregado lejano se determinan mediante la emisión de rayos de visión desde el espectador a través de las esquinas del plano agregado cercano hasta intersecar el plano agregado lejano. Así, cuanto más cerca del espectador esté el plano agregado cercano, más grande será el plano agregado lejano, y el escorzo será más evidente.

El uso de `glFrustum()` le permite especificar un tronco asimétrico, que puede ser útil en algunos casos, pero generalmente no se utiliza. Además, pensar en lo que el espectador puede ver en términos de un tronco no es muy intuitivo. En cambio, es más fácil pensar en el campo de visión, es decir, ¿qué amplitud de ángulo puede ver el espectador? La biblioteca de utilidades de OpenGL proporciona una función que le permite especificar directamente el campo de visión, y luego calcula el frustum para usted. Esta función es la siguiente:

```
void gluPerspective(GLdouble fov, GLdouble aspect, GLdouble near, GLdouble far);
```

`fov` (field of view) especifica en grados el ángulo alrededor del eje y que es visible para el usuario. `aspect` (aspect ratio) es la relación de aspecto de la pantalla, esto es el ancho dividido entre la altura. Lo anterior determina el campo de visión alrededor del eje x. `near` y `far` tienen los mismos significados que han tenido en las otras funciones de proyección incluidas en esta sección.

Una característica que no hemos mencionado en nuestro análisis de la creación de un frustum es cómo determinar una relación adecuada entre la anchura de los extremos cercano y lejano (es decir, la amplitud del campo de visión). El campo de visión adecuado depende mucho de la aplicación. Si se desea crear un efecto de "ojo de pescado", un campo de visión muy amplio puede ser apropiado. Para una perspectiva realista, un campo entre 45 y 90 grados suele funcionar bien. En general, es necesario que experimente para observar lo que parece adecuado para su aplicación particular.

Configuración de la ventana de visualización

Algunas de las funciones de proyección que acabamos de comentar están estrechamente relacionadas con el tamaño de la ventana de visualización (por ejemplo, la relación de aspecto en

`gluPerspective()`). Usted sabe que la transformación de la ventana de visualización ocurre después de la transformación de proyección, por lo que ahora es un momento muy adecuado para analizarla.

En esencia, la ventana de visualización especifica las dimensiones y la orientación de la ventana 2D en la que se realizará el render. Se establece mediante `glViewport()`:

```
void glViewport(GLint x, GLint y, GLsizei width, GLsizei height);
```

`x, y` especifican las coordenadas de la esquina inferior izquierda de la ventana de visualización, y `width` y `height` especifican el tamaño de la ventana en pixeles.

Cuando se crea un contexto de render en primer lugar y se adjunta a su ventana, la visualización se ajusta automáticamente para que coincida con las dimensiones de la ventana. Eso puede ser suficiente para algunas aplicaciones, pero en la mayoría de los casos será necesario actualizar su visualización cada vez que la ventana cambie de tamaño. Aunque generalmente la visualización coincide con el tamaño de su ventana, no hay nada que las obligue a ser del mismo tamaño. Habrá ocasiones en las que quiera limitar el render a una sub-región de su ventana, y una forma de hacer esto consiste en establecer una ventana de visualización más pequeña.

Ejemplo de proyección

Para tener una mejor idea de las diferencias entre los dos tipos principales de proyección, hemos incluido una demostración simple que le permite ver la misma escena en cada modo. La demostración comienza con una proyección perspectiva; al presionar la barra espaciadora es posible alternar

Figura 4.15
Proyección ortográfica.

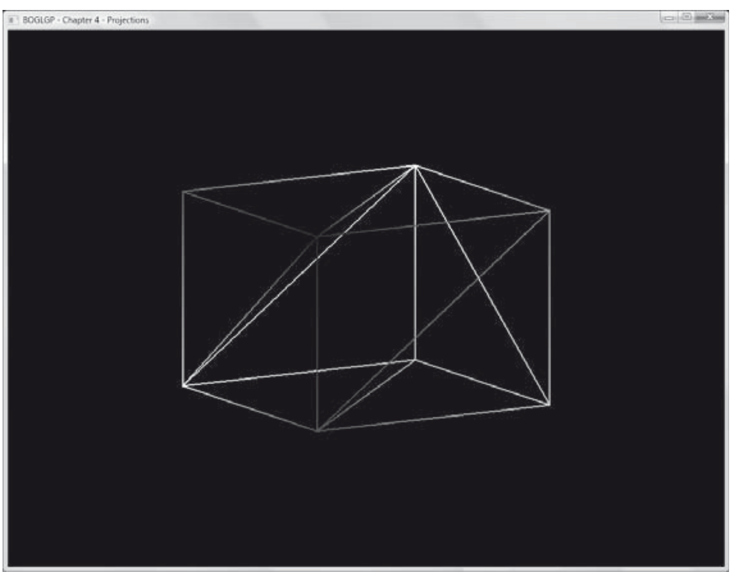
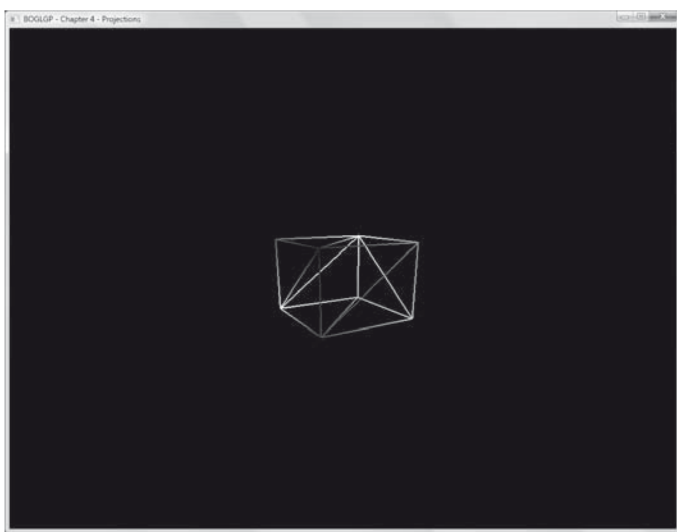


Figura 4.16
Proyección de perspectiva.



entre las proyecciones ortográfica (que se muestra en la figura 4.15) y perspectiva (mostrada en la figura 4.16).

La parte más relevante de esta demostración está en los métodos `updateProjection()` y `OnResize()` de la clase `Example`, que se reproducen a continuación para su comodidad:

```
void Example::updateProjection()
{
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();

    if(m_perspective) {
       //Establece la perspectiva con la relación de aspecto apropiado
       glFrustum(-1.0, 1.0, -1.0, 1.0, 1.0, 1000.0);
    } else {
      //Establece una proyección ortográfica con el mismo plano cercano
      glOrtho(-1.0, 1.0, -1.0, 1.0, 1.0, 1000.0);
    }
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
void Example::onResize(int width, int height)
{
```

```
    if(height = = 0) //Evita una división entre cero
    {
      height = 1;
    }

    //Establece la visualización al nuevo tamaño de ventana
    glViewport(0, 0, width, height);

    //Establece la proyección con base en la bandera m_perspective flag
    updateProjection();
}
```

Manipulación del punto de visualización

En esta sección, se presentarán varias opciones para manipular el punto de visualización o "cámara". La primera alternativa es utilizar la función `gluLookAt()`, que le permite especificar la posición del punto de visualización, un vector direccional desde el punto de visualización y otro vector ascendente para orientar y posicionar la visión. La segunda opción consiste en emplear una combinación de las funciones `glTranslate()` y `glRotate()` para orientar y colocar el punto de visualización. Por último, puede utilizar sus propias rutinas personalizadas para definir el comportamiento de la visualización. Por ejemplo, usted podría desear que el punto de visualización se oriente a través del sistema de coordenadas polares.

A continuación se describe cada una de estas opciones.

Uso de gluLookAt()

La función `gluLookAt()` se define como

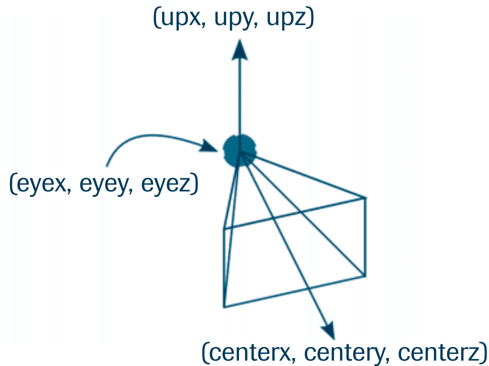
```
void gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble centerx,
GLdouble centery, GLdouble centerz, GLdouble upx, GLdouble upy, GLdouble upz);
```

Esta función puede utilizarse para definir la ubicación y la orientación de la cámara en vez de las transformaciones de modelado `glTranslate()` y `glRotate()`. El primer conjunto de tres parámetros (`eyex, eyey, eyez`) especifica la ubicación de la cámara. El valor (0, 0, 0), naturalmente, especifica el origen. El siguiente conjunto de parámetros (`centerx, centery, centerz`) especifica hacia dónde se dirige la cámara. A partir de éstos, OpenGL puede determinar cuál dirección es la que está frente a la cámara.

El último conjunto de parámetros (`upx, upy, upz`) es un vector que indica cuál es la dirección que está hacia arriba. En la figura 4.17 se muestra cómo operan todos estos parámetros sobre la cámara con la función `gluLookAt()`.

A continuación se presenta un pequeño fragmento de código que utiliza la función `gluLookAt()`.

Figura 4.17
Los parámetros de gluLookAt() especifican la ubicación y la orientación de la cámara.



gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble centerx, GLdouble centery, GLdouble centerz, GLdouble upx, GLdouble upy, GLdouble upz);

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glLoadIdentity();
/*Establecemos la cámara en la posición (0, 0, 10) mirando hacia el eje z
negativo (0, 0, -100). La orientación de la cámara se determina mediante el
vector (0, 1, 0)
*/
gluLookAt(0.0, 0.0, 10.0, //Posición
          0.0, 0.0, -100.0, //Punto hacia donde mira la cámara
          0.0, 1.0, 0.0); //Vector de orientación

glEnableClientState(GL_VERTEX_ARRAY);
glBindBuffer(GL_ARRAY_BUFFER, m_vertexBuffer);
glVertexPointer(3, GL_FLOAT, 0, 0);
glDrawArrays(GL_TRIANGLES, 0, 3);
```

Como puede verse, la función `gluLookAt()` es bastante fácil de usar. Es posible mover la cámara a cualquier posición y orientación que se desee mediante la manipulación de los parámetros.

Uso de glRotate() y glTranslate()

Una desventaja de `gluLookAt()` es que se debe vincular la librería GLU con su aplicación. ¿Qué pasa si no quiere utilizar la biblioteca GLU, pero desea obtener la misma funcionalidad?

Una solución consiste simplemente en usar las funciones de transformación en el modelado glRotate() y glTranslate(), como se explicó anteriormente en este capítulo. El código que se presenta enseguida utiliza las funciones de modelado para producir el mismo efecto en la cámara que el código anterior de gluLookAt().

```
glCLear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glLoadIdentity();
/* Movemos la transformación de modelado a (0.0, 0.0, -10.0). Al mover el mundo
10 unidades a lo largo del eje z negativo, efectivamente la cámara se mueve a
la posición (0.0, 0.0, 10.0)
*/
glTranslatef(0.0f, 0.0f, -10.0f);

//Dibuja un triángulo
glEnableClientState(GL_VERTEX_ARRAY);
glBindBuffer(GL_ARRAY_BUFFER, m_vertexBuffer);
glVertexPointer(3, GL_FLOAT, 0, 0);
glDrawArrays(GL_TRIANGLES, 0, 3);
```

En este caso, no hay una gran diferencia en el código con respecto a la función gluLookAt() porque todo lo que estamos haciendo es mover la cámara a lo largo del eje z. Pero si se estuviera orientando la cámara en un ángulo extraño, también habría necesidad de utilizar la función glRotate(), lo cual nos conduce a la siguiente manera de manipular la cámara: sus rutinas personalizadas.

Creación de sus propias rutinas personalizadas

Suponga que desea crear su propio simulador de vuelo. En un simulador de vuelo típico, la cámara está colocada en el asiento del piloto, por lo que se mueve y se orienta en la misma forma que el avión. La orientación del avión se define mediante los ángulos pitch, yaw y roll, que son los ángulos de rotación con respecto al centro de gravedad del avión (en su caso, el piloto y la posición de la cámara). Usando las funciones de transformación del modelado, usted puede establecer la siguiente función para crear la transformación de la imagen:

```
void planeView(GLfloat planeX, GLfloat planeY, GLfloat planeZ, GLfloat roll,
              GLfloat pitch, Glfloat yaw)
{
  //roll es la rotación alrededor del eje z
  glRotatef(roll, 0.0f, 0.0f, 1.0f);
  //yaw, o heading, es la rotación alrededor del eje y
  glRotatef(yaw, 0.0f, 1.0f, 0.0f);
```

```
  //pitch es la rotación alrededor del eje x
  glRotatef(pitch, 1.0f, 0.0f, 0.0f);
  // mueve a las coordenadas del mundo del avión
  glTranslatef(-planeX, -planeY, -planeZ);
}
```

Con esta función, usted coloca la cámara en el asiento del piloto de su avión sin importar la orientación o la ubicación del mismo. Éste es sólo uno de los usos de sus rutinas personalizadas. Otros usos incluyen la aplicación de coordenadas polares, como la rotación alrededor de un punto fijo y la utilización de funciones de transformación de modelado para crear lo que se conoce como "movimiento tipo Quake", donde el ratón y el teclado pueden utilizarse para controlar la cámara.

El máximo grado de control de la cámara puede obtenerse al construir y cargar de manera manual sus propias matrices, lo cual se estudiará en la siguiente sección.

Uso de sus propias matrices

Hasta ahora, hemos analizado las funciones que le permiten modificar las pilas de matrices sin realmente tener que preocuparse por las matrices en sí. Esto es grandioso porque le permite hacer muchas cosas sin tener que entender las matemáticas matriciales, y las funciones que le proporciona OpenGL son bastante potentes y flexibles. Sin embargo, es posible que en algún momento usted desee crear efectos avanzados que son posibles sólo si se afecta de manera directa a las matrices. Para ello será necesario que usted conozca el entorno de las matemáticas matriciales, que se adoptó como requisito previo para la lectura de este libro. Sin embargo, al menos mostraremos cómo cargar su propia matriz, la forma de multiplicar la parte superior de la pila de matrices por una matriz personalizada, así como un ejemplo de la utilización de una matriz personalizada.

Carga de su matriz

Antes de que pueda cargar una matriz, es necesario especificarla. Las matrices de OpenGL son matrices de columna mayor 4 x 4, con números de punto flotante y dispuestas como en la figura 4.18.

Figura 4.18
Formato de la matriz de columna mayor utilizada por OpenGL.

$$\begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix}$$

Como las matrices son de 4 x 4, es posible que se sienta tentado a declararlas como arreglos bidimensionales, pero hay un gran problema con esto. En C y C++, los arreglos de dos dimensiones son de fila mayor. Por ejemplo, para acceder al elemento inferior izquierdo de la matriz mostrada en la figura 4.18, se podría pensar en el uso de matrix[3][0], que es la manera en la que se puede acceder a la esquina inferior izquierda de un arreglo bidimensional de 4 x 4 en C/ C++, pero como las matrices de OpenGL son de columna mayor, usted realmente tendría acceso al elemento superior derecho de la matriz. Para obtener el elemento inferior izquierdo, habría necesidad de utilizar matrix[0][3]. Esto es opuesto a lo que usted ha usado en C/C++, por lo que es contrario a la intuición y propenso al error. En lugar de utilizar arreglos de dos dimensiones, es recomendable emplear un arreglo unidimensional de 16 elementos. El enésimo elemento del arreglo corresponde al elemento mn de la figura 4.18.

A manera de ejemplo, si desea especificar la matriz identidad puede usar:

```
GLfloat identity[16] = {1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0,
0.0, 0.0, 0.0, 0.0, 1.0);
```

Esto es bastante fácil. Así que, ahora que se ha especificado una matriz, el siguiente paso es cargarla. Lo anterior se hace llamando a `glLoadMatrix()`:

```
void glLoadMatrix{fd}(const TYPE matrix[16]);
```

Cuando se llama a `glLoadMatrix()`, lo que esté en la parte superior de la pila de matrices seleccionada actualmente se sustituye por los valores del arreglo matricial, que es un arreglo de 16 elementos como se especificó anteriormente.

Multiplicación de matrices

Además de cargar nuevas matrices en la pila (perdiendo con esto toda la información que se encontraba previamente en ella), es posible multiplicar el contenido de la matriz activa por una nueva matriz. De nuevo, usted debe especificar la matriz personalizada como lo hizo antes y luego hacer la siguiente llamada:

```
void glMultMatrix{fd}(const TYPE matrix[16]);
```

Una vez más, la matriz es un arreglo de 16 elementos. `glMultMatrix()` usa la post-multiplicación; en otras palabras, si la matriz activa antes de la llamada a `glMultMatrix()` es `Mold` y la nueva matriz es `Mnew`, entonces la nueva matriz será `Mold x Mnew`. Tenga en cuenta que el orden es importante, porque la multiplicación de matrices no es conmutativa; en la mayoría de los casos `Mold x Mnew` no tendrá el mismo resultado que `Mnew x Mold`.

Resumen

En este capítulo, usted aprendió a manipular objetos en su escena mediante el uso de transformaciones. También examinó cómo cambiar la forma en que se ve la propia escena a través de la creación de proyecciones. En el proceso, aprendió acerca de las matrices de proyección y del modelo de visualización, así como acerca de su manipulación empleando las funciones integradas y las matrices definidas por su cuenta. Ahora tiene los medios para colocar, mover y animar objetos en un mundo tridimensional; también puede trasladar los objetos dentro de ese mundo.

Lo que se aprendió

- Las transformaciones le permiten mover, girar y manipular objetos en un mundo en 3D, al mismo tiempo que le permite proyectar coordenadas tridimensionales en una pantalla bidimensional.
- La transformación de la imagen especifica la ubicación de la cámara.
- La transformación de modelado mueve objetos en todo el mundo 3D.
- La transformación de proyección define la visualización, el volumen y planos agregados.
- La transformación de la ventana de visualización dibuja la proyección de la escena en el visor, o la ventana, de su pantalla.
- La transformación modelview de OpenGL es una combinación de las transformaciones de modelado y visualización.
- El punto de vista también se denomina "cámara" o "coordenadas de visualización".
- La traslación es el acto de mover un objeto a lo largo de un vector.
- La rotación es el acto de girar un objeto alrededor de un eje definido por vectores.
- El escalado es el acto de aumentar o disminuir el tamaño de un objeto.
- La proyección de perspectiva muestra mundos en 3D exactamente como usted ve las cosas en la vida real. Los objetos más alejados parecen más pequeños que los objetos que están más próximos a la cámara.
- La proyección ortográfica muestra objetos en pantalla a su tamaño verdadero, sin importar su distancia a la cámara.
- La matriz modelview define el sistema de coordenadas que se utiliza para colocar y orientar objetos. La matriz modelview se configura como la matriz actual mediante la función `glMatrixMode()` con `GL_MODELVIEW` como parámetro. Por otra parte, si se usa `GL_PROJECTION` como parámetro, la matriz actual será la de proyección.
- `glLoadIdentity()` restablece a la matriz identidad como la matriz actual.
- La traslación en OpenGL se realiza mediante la función `glTranslate()`.
- La rotación en OpenGL se realiza mediante la función `glRotate()`
- El escalado en OpenGL se realiza mediante la función `glScale()`.
- Para guardar y restablecer la matriz actual se utilizan las funciones `glPushMatrix()` y `glPopMatrix()`.
- Las funciones `glOrtho()` y `gluOrtho2D()` se utilizan para configurar las proyecciones ortográficas.

- Las funciones `glFrustum()` y `gluPerspective()` se emplean para configurar las proyecciones perspectivas.
- `gluLookAt()` puede utilizarse para ubicar y orientar el punto de vista de OpenGL.
- La función `glLoadMatrix()` se usa para cargar una matriz definida por el usuario como la matriz actual en OpenGL.
- La función `glMultMatrix()` se emplea para multiplicar la matriz actual de OpenGL por una matriz definida por el usuario.

Preguntas de repaso

1. ¿Cómo se almacena la matriz actual de modo que pueda restaurarse después?
2. Escriba la línea de código que colocará a un objeto en (10, 5, 0).
3. Mencione tres diferentes pilas de matrices en OpenGL.
4. Escriba la línea de código que reducirá a la mitad el tamaño de los objetos que se dibujarán después.
5. ¿Qué comando le permite rotar un objeto?
6. ¿Qué comando le permite cargar su propia matriz en la pila de matrices actual?
7. ¿Cómo se restaura una matriz que fue introducida previamente en la pila de matrices?

Por su cuenta

1. Escriba un programa que dibuje una pirámide girando de manera constante alrededor del eje y, y que se mueva hacia atrás y hacia adelante a lo largo del eje z.

Extensiones de OpenGL

En el capítulo 2, mencionamos que para obtener un contexto en OpenGL 3.0 era necesario utilizar extensiones de OpenGL. Ahora es el momento de ver el mecanismo de extensión con mayor detalle. En este capítulo, usted aprenderá:

- Qué es una extensión de OpenGL
- Cómo descubrir cuáles extensiones son compatibles
- Por qué son necesarias las extensiones en la plataforma Windows
- Cómo se tiene acceso a las extensiones
- Cómo se utiliza la biblioteca GLee a fin de aplicar en forma clara la mayoría de las características de vanguardia.

¿Qué es una extensión?

Una extensión de OpenGL es en realidad sólo eso, una extensión del OpenGL estándar. De hecho, al igual que en OpenGL, cada extensión se registra en un documento de especificación. Este documento describe a detalle algunas funcionalidades adicionales que le permiten hacer los render con mayor rapidez, con una mejor calidad o con mayor facilidad. Las extensiones le permiten a los fabricantes de tarjetas gráficas dar un acceso rápido a las características más avanzadas de sus conjuntos de chips, sin necesidada de esperar las actualizaciones de la especificación de OpenGL.

Las especificaciones de cada extensión se alojan en el registro de extensiones de OpenGL, que puede encontrarse en http://www.opengl.org/registry/.

Por lo general, la especificación de una extensión de OpenGL contiene la siguiente información:

- **Nombre**—El nombre de la extensión
- **Cadenas de nombre**—Identificadores únicos para la extensión
- **Descripción**—En esta sección se refiere brevemente el propósito de la extensión, por qué existe la necesidad de extender una funcionalidad y qué es lo que logra la extensión
- **Nuevos procedimientos y funciones**—Los prototipos de cualquier función nueva de la API que se haya agregado como parte de esta especificación
- **Nuevas fichas**—Cualquier constante nueva que esté definida por la extensión
- **Dependencias**—Cualquier otra extensión sobre la que se base la presente extensión

Nomenclatura de la extensión

Toda extensión de OpenGL debe tener un nombre único para poder identificarla. Hay una convención de nomenclatura estándar para las extensiones; todos los nombres deben comenzar con un prefijo, las palabras estarán separadas con guiones bajos en vez de espacios, y todo lo que se encuentre después del prefijo suele escribirse en minúsculas. Por ejemplo:

```
PREFIX_extension_name
```

Cuando un proveedor crea una extensión, la nombra con su propio prefijo único. Por ejemplo, NVIDIA utiliza el prefijo NV, mientras que Apple usa, como es lógico, Apple. Si una extensión es adoptada por más de un proveedor, el prefijo cambia a EXT para reflejar este hecho. En algún momento, ARB podría aprobar oficialmente la extensión, en cuyo caso el prefijo se convierte en ARB. En la tabla 5.1 se muestran algunos de los prefijos utilizados.

Cadenas de nombre

Como se mencionó con anterioridad, en cada documento de especificación existe una sección llamada “Name Strings” o cadenas de nombre. Estas cadenas son valores únicos para la extensión que se enlistan como parte de la(s) cadena(s) GL_EXTENSIONS que pueden obtenerse utilizando `glGetString()` o `glGetStringi()` si la extensión es compatible con la implementación de OpenGL.

Tabla 5.1 Prefijos en las extensiones de OpenGL

Prefijo	Significado/Proveedor/Categoría
ARB	Extensión aprobada por el Architecture Review Board
EXT	Extensión adoptada por dos o más proveedores
APPLE	Apple Inc.
ATI	AMD/ATI
MESA	Implementación de código abierto de la especificación OpenGL
NV	NVIDIA
SGI	Silicon Graphics Inc.
SGIS	Silicon Graphics Inc. (especializado)
SGIX	Silicon Graphics Inc. (experimental)
SUN	Sun Microsystems
WIN	Microsoft

Por lo general, la cadena de nombre para una extensión es el nombre de la extensión acompañado por el prefijo `GL_`, `WGL_` (para extensiones de Windows), `GLX_` (para extensiones X), o `GLU_` (para las extensiones a la librería GLU).

Aunque cada extensión suele tener sólo una cadena de nombre, de vez en cuando alguna puede tener varias cadenas, especialmente si la extensión añade una funcionalidad específica para el sistema de manejo de ventanas.

Funciones y fichas

Las extensiones pueden agregar nuevas funciones, nuevas fichas, o ambas cosas a OpenGL. Si la extensión proporciona nuevas funciones, tendrá que obtener un puntero hacia ellas antes de poder utilizarlas. Esto se explica un poco más adelante en la sección “Obtención de un punto de entrada a una función”.

Cualquier función nueva está obligada a seguir las convenciones de nomenclatura definidas por OpenGL, es decir, cada función está precedida por “`gl`” en minúsculas y después el resto del nombre de la función utiliza una letra mayúscula al principio de cada palabra. Por último, cada nombre de función está seguido del prefijo usado en el nombre de la extensión (como ocurre con ARB).

Por ejemplo:

```
glFunctionNameARB()
```

Si posteriormente la extensión se traslada a la base de OpenGL, el sufijo se retira de los nombres de la función que realiza.

Algunas extensiones pueden no especificar ninguna función nueva para OpenGL, sino que sólo proporcionan una nueva ficha para pasar a una función existente. Una ficha es una constante o enumeración como `GL_TEXTURE_2D` o `GL_FLOAT`.

Todas las nuevas fichas deben seguir una nomenclatura consistente con el resto de OpenGL. Esto significa que deberán tener nombres con letras mayúsculas, con espacios sustituidos por guiones bajos y deberán tener el prefijo `GL_`. Una vez más la ficha tendrá adjunto el prefijo del nombre de la extensión. Por ejemplo:

```
GL_SOME_NEW_TOKEN_ARB
```

Si una extensión sólo proporciona nuevas fichas (y no funciones), es mucho más fácil de usar ya que no es necesario obtener punteros hacia la función. En su lugar, es posible simplemente descargar el archivo de encabezado `glext.h` más reciente desde el registro de extensiones de OpenGL, el cual proporciona las fichas de extensión y las definiciones de función más actuales. Para las fichas de extensión en plataformas específicas también existen `wglext.h` y `glxext.h`, para Windows y X, respectivamente.

Obtención de un punto de entrada a las funciones

Los archivos de encabezado `glext.h`, `wglext.h` y `glxext.h` le proporcionan las definiciones de las funciones de extensión más recientes, ¡pero las definiciones no son de mucha utilidad sin las implementaciones de la función real! Como las implementaciones no están disponibles cuando se compila el código, es necesario vincularse con éstas de manera dinámica al momento de la ejecución. Lo anterior implica consultar los controladores de gráficos en busca de un puntero hacia la función que se desea utilizar. Esto es específico para cada plataforma, pero la idea general es la misma en todas ellas.

En primer lugar, debe declararse un puntero hacia una función. Cada función tiene sus propios parámetros y tipo de respuesta, lo cual hace que definir punteros hacia las funciones sea complicado. Por fortuna, los encabezados mencionados anteriormente nos proporcionan algunos typedef (tipos de definición) para que los punteros de función sean un poco más legibles. A continuación, se llama a una función específica de la plataforma para encontrar la dirección de la función deseada y asignarle el puntero de la función. Si después de la llamada descrita anteriormente el puntero que se obtiene es NULL, entonces la extensión que proporciona la función no está disponible. Ahora es posible utilizar el puntero hacia la función de la misma manera que una función normal.

La información anterior puede desglosarse a través de un ejemplo. Para ello se obtendrá un puntero hacia la función `glGetStringi()`. Más adelante, usted tendrá que utilizar esto para saber cuáles otras extensiones son compatibles. Después de incluir `glext.h` debemos declarar el puntero de una función del modo siguiente:

```
PFNGLGETSTRINGIPROC glGetStringi = NULL;
```

Eso es bastante sencillo. PFNGLGETSTRINGIPROC es el typedef para el tipo puntero de función, que se define en glext.h. El paso siguiente es conseguir la dirección de la función glGetStringi() y asignársela a nuestro puntero. En Windows, para realizar tal procediminto se usa la función wglGetProcAddress() (en X que usaría glXGetProcAddress()) que tiene la siguiente definición:

```
PROC wglGetProcAddress (LPCSTR lpszProcName);
```

Veamos cómo se utiliza para conseguir nuestro puntero hacia la función glGetStringi():

```
glGetStringi = (PFNGLGETSTRINGIPROC) wglGetProcAddress(“glGetStringi”);
```

Debemos vaciar el valor devuelto hacia nuestro tipo de puntero antes de asignárselo a nuestro puntero de función. Si la función no estaba disponible, wglGetProcAddress() devolverá NULL. Es necesario que verifique esto antes de intentar usar el puntero de función, de lo contrario, el programa puede fallar. Por ejemplo:

```
if(glGetStringi != NULL) {
      //Puede usarse glGetStringi
}
```

Extensiones en Windows

La intención original del mecanismo de extensión fue proporcionar funcionalidades de vanguardia que no formaban parte del OpenGL básico. Éste sigue siendo el caso en las plataformas distintas a Windows. Sin embargo, para tener acceso a cualquier funcionalidad posterior a OpenGL 1.1 en Windows es necesario utilizar extensiones. Para entender por qué, debemos observar qué se requiere para utilizar OpenGL en sus aplicaciones.

Al desarrollar una aplicación de OpenGL, se deben incluir los archivos de encabezado de OpenGL, que le dan acceso a las definiciones de funciones y fichas disponibles. También se precisa vincular la aplicación (en Windows) con OPENGL32.lib y posiblemente con GLU32.lib. Estos archivos de biblioteca se requieren para acceder a las funcionalidades de las bibliotecas de vínculos dinámicos OPENGL32.dll y GLU32.dll, respectivamente.

Aquí es donde radica el problema, Microsoft no se ha mantenido al día con los lanzamientos de OpenGL, y los encabezados, bibliotecas y archivos DLL están muy desfasados. Para utilizar cualquier funcionalidad nueva es necesario emplear el mecanismo de extensión.

Búsqueda de extensiones compatibles

Antes de usar una extensión de OpenGL, es una buena idea primero verificar si ésta se encuentra presente en el sistema en funcionamiento. Como se mencionó anteriormente, las extensiones

disponibles pueden obtenerse mediante el uso de glGetStringi() y pasando GL_EXTENSIONS como primer parámetro. glGetStringi() toma un índice entero como segundo parámetro. Para obtener una lista de todas las extensiones compatibles con la implementación de OpenGL, es necesario iterar desde 0 hasta NUM_EXTENSIONS- 1 llamando a glGetStringi() cada vez con el nuevo índice. Pero ¿qué tan grande es NUM_EXTENSIONS? Para averiguarlo, necesita pasar GL_NUM_EXTENSIONS a glGetIntegerv(), de la manera siguiente:

```
GLint numExtensions = 0;
glGetIntegerv(GL_NUM_EXTENSIONS, &numExtensions);
```

Después de esta llamada, numExtensions conservará el número de extensiones compatibles con la implementación actual de OpenGL. Lo único que queda por hacer es tomar las cadenas de extensión una por una y guardarlas en un arreglo:

```
  for (int i = 0; i < numExtensions; i++)
  {
    //Obtiene la extensión en i
    string extension = (const char*) glGetStringi(GL_EXTENSIONS, i);
    //Añade la extensión a la lista de cadenas
    extensions.push_back(extension);
  }
```

Como se describe en la sección “Obtención de un punto de entrada a las funciones”, antes de poder utilizar la función glGetStringi(), tendrá que obtener un puntero hacia ella. Después, puede utilizar la función para consultar las extensiones disponibles. Todos estos pasos pueden unirse para crear una función que devuelve un arreglo de todas las extensiones compatibles:

```
vector <string> getSupportedExtensions()
{
  PFNGLGETSTRINGIPROC glGetStringi = NULL;
  glGetStringi = (PFNGLGETSTRINGIPROC)wglGetProcAddress(“glGetStringi”);

  vector <string> extensions;
  if(glGetStringi == NULL)
  {
    //Devuelve un arreglo vacío
    return extensions;
  }

  GLint numExtensions = 0;
  //Obtiene el número de extensiones compatibles
```

```
  glGetIntegerv(GL_NUM_EXTENSIONS, &numExtensions);
  for (int i = 0; i < numExtensions; i++)
  {
    //Obtiene la extensión en i
    string extension = (const char*) glGetStringi (GL_EXTENSIONS, i);
    //Añade la extensión a la lista de cadenas
    extensions.push_back(extension);
  }
  return extensions;
}
```

Nota

Podrá observar que vaciamos el resultado de `glGetStringi()` hacia `const char*`. Esto se debe a que `glGetStringi()` devuelve una cadena formada por caracteres sin signo `(const Glubyte*)`, pero nosotros requerimos caracteres con signo para asignarlos a la instancia de cadena.

Para determinar la compatibilidad de una extensión sólo es necesario iterar sobre el arreglo en busca de la cadena de extensión que se requiere. La función `isExtensionSupported()` que se presenta a continuación sólo hace eso.

```
bool isExtensionSupported(const string & ext)
{
    //Obtiene una lista de extensiones
    vector<string> extensions = getSupportedExtensions();

    /*Recorre todas las extensiones y si la actual coincide con la que se
    está buscando entonces devuelve true*/
    for(vector<string>::iterator it = extensions.begin(); it! = exten-
sions.end (); ++it)
  {
    if (*it == ext)
    {
      return true;
    }
  }

  /*Si recorremos todas las extensiones y ninguna coincide, entonces
devuelve false*/
```

```
    return false;
}
```

Extensiones de WGL

Además de las extensiones estándar para OpenGL, hay algunas que son específicas para el sistema Windows. Estas extensiones proporcionan adiciones muy específicas para el sistema de manejo de ventanas y la forma en que interactúa con OpenGL. Para conseguir estas extensiones específicas de Windows, debe utilizar la función wglGetExtensions-StringARB() que (como glGetStringi()) es en sí misma una extensión (ARB_extension_string). Para obtener acceso a esta función, tendrá que utilizar wglGetProcAddress() y comprobar antes de usarla que el puntero de función devuelto no es NULL. El formato de esta función es el siguiente:

```
const char*wglGetExtensionsStringARB(HDC hdc);
```

El único parámetro es el handle o manejador para el contexto de render. Esta función se diferencia de glGetStringi() en que devuelve las cadenas de extensión como una lista delimitada por espacios; en ese sentido, es igual que el antiguo glGetString(GL_EXTENSIONS).

Definición de fichas

Si usted está utilizando una versión actualizada de glext.h y wglext.h o glxext.h todas sus fichas de extensión ya estarán definidas, pero también puede definir fichas de extensión por su cuenta. Si una extensión especifica fichas nuevas cualquiera, éstas se enlistarán en el documento de especificación asociado dentro del registro de OpenGL. Por ejemplo, en la especificación NV_half_float bajo el encabezado de "New tokens" se encuentra una ficha llamada HALF_FLOAT_NV con el valor 0 x 140B. Para definir esto, sólo tiene que utilizar:

```
#define GL_HALF_FLOAT_NV 0 x 140B
```

Observe que colocamos el prefijo GL_ a la constante definida en la especificación, para que coincida con la convención de nombres adoptada para las otras fichas.

Introducción a GLee

Si utiliza sólo unas pocas funciones de extensión, la obtención y manejo de los punteros de función no es demasiado difícil. Sin embargo, cuando se empiezan a utilizar muchas funciones de extensión, éstas se vuelven difíciles de manejar y entonces la obtención de los punteros de función se complica rápidamente, ya que el proceso abarrota el código de inicialización. Por fortuna, hay bibliotecas existentes que de manera transparente inicializan punteros de función, especifican fichas y proporcionan un modo fácil de comprobar si una extensión es compatible.

Las dos bibliotecas más comunes que ofrecen esta funcionalidad son GLee (OpenGL Easy Extension Library) y GLEW (OpenGL Extension Wrangler Library). En este texto se abordará GLee, pero el uso de GLEW es muy similar. GLee es desarrollado y mantenido por Ben Woodhouse, y está disponible para Linux y Windows, aunque también es compatible con OSX y FreeBSD. Se lanza bajo una licencia BSD modificada sin restricciones. En el CD que se incluye con el libro puede encontrar la última versión de GLee con el soporte de OpenGL 3.0; sin embargo, puede revisar por su cuenta si hay actualizaciones disponibles en http://elf-stone.com/glee.php.

Configuración de GLee

El paquete de fuentes de GLee proporciona un archivo de encabezado (para incluirlo en su código), un archivo fuente (glee.c) y una biblioteca (glee.lib). Existen dos opciones para utilizar GLee en sus aplicaciones: usted puede agregar el archivo fuente a su proyecto, el cual se compilará junto con el resto del código, o bien puede decirle al compilador que se enlace con la biblioteca de GLee. De cualquier manera, sólo tiene que extraer el archivo zip que contiene a GLee y copiar los archivos a un lugar donde pueda incluirlos en su proyecto. Esto puede hacerse en una carpeta local para el proyecto o se puede incluir en el directorio donde se encuentran el compilador y las bibliotecas.

Uso de GLee

Antes de poder utilizar GLee, es necesario incluir el archivo de encabezado `glee.h. glee.h` que reemplaza la necesidad de incluir `gl.h`; de hecho, su código producirá un error descriptivo del compilador si se incluye a `gl.h` antes de `glee.h`. Las últimas versiones de GLee no requieren ninguna inicialización especial, por lo que una vez que se haya incluido el archivo de encabezado, le será posible comenzar a revisar las extensiones.

De manera interna, GLee mantiene una variable booleana para cada extensión, la cual almacena si dicha extensión está disponible o no. El nombre de cada una de estas variables es el mismo que la cadena de nombre, pero con el prefijo `GLEE_` en vez de `GL_`. Por ejemplo, cuando se desee saber si la extensión para objetos del búfer de vértices está disponible (que siempre será cierto en un contexto GL 3.0), se puede utilizar el texto siguiente:

```
if(GLEE_ARB_vertex_buffer_object)
{
   glBindBufferARB(...);
   ...
 }
```

Las extensiones específicas para una plataforma (WGL/GLX) se tratan de forma un poco diferente; en este caso, toda la cadena de nombre lleva el prefijo `GLEE_` e incluye la parte `WGL_` o `GLX_` del nombre. Por ejemplo `GLEE_WGL_ARB_pbuffer`.

Puede comprobar cuál versión de OpenGL es compatible si verifica el valor de la variable `GL_VERSION_x_y`, en donde x y y son los números de versión mayor y menor, respectivamente.

Uso de GLee con las extensiones básicas

Durante la existencia de OpenGL, muchas características que empezaron como extensiones pasaron a formar parte de la especificación básica. GLee permite utilizar estas funciones sin necesidad de agregar el sufijo de extensión. Por ejemplo, los vertex buffer objects (objetos del búfer de vértices) fueron trasladados a la especificación base, así que puede utilizar:

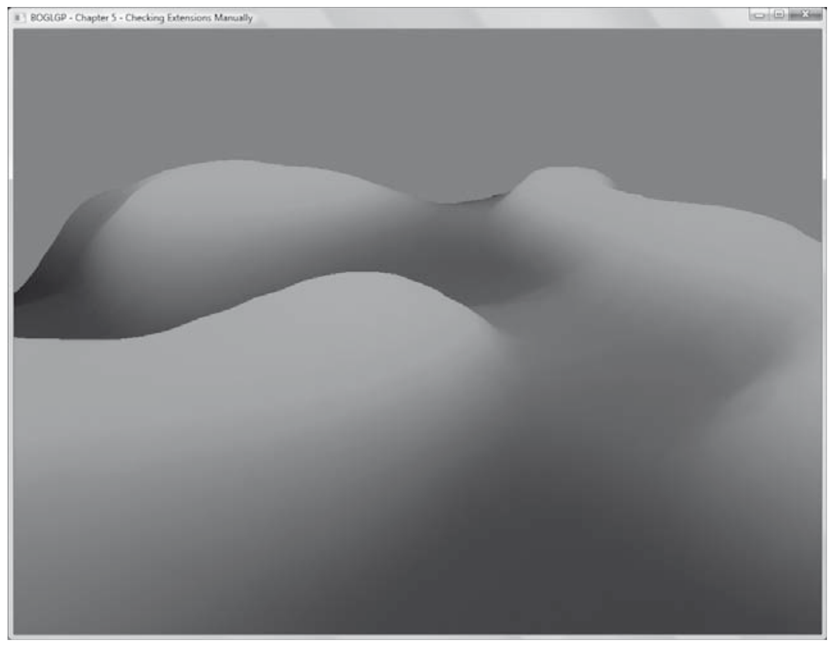
```
glBindBuffer(...);
```

en vez de:

```
glBindBufferARB(...);
```

Si está utilizando la versión anterior, asegúrese de verificar la bandera `GLEE_VERSION_x_y` para ver si el controlador es compatible con la versión de OpenGL en la que la extensión se promovió al núcleo.

Figura 5.1
Captura de pantalla del ejemplo del terreno.



Extensiones en acción

Como los ejemplos de este libro están escritos para OpenGL 3.0, todos ellos usan una extensión de algún tipo. Sin embargo, hasta ahora hemos hecho la verificación de extensiones de forma manual. Dentro de la carpeta para el capítulo 5 en el CD, se incluyen dos versiones de la misma aplicación. El programa hace el render de un terreno simple basado en un mapa de elevaciones, el cual se seguirá desarrollando en capítulos posteriores para hacerlo más realista. Una versión de la aplicación comprueba si hay extensiones disponibles y las carga de forma manual; la otra utiliza a GLee para manejar las extensiones.

Resumen

Las extensiones de OpenGL son esenciales en la plataforma Windows y son necesarias para crear y utilizar un contexto de OpenGL 3.0, así como para tener acceso a todas las funciones avanzadas. Las extensiones también permiten utilizar las funcionalidades de vanguardia en las tarjetas gráficas más recientes en cualquier plataforma. Ahora ya podrá comprender cómo se comprueba la disponibilidad de una extensión y la forma de tener acceso a las funciones que ésta proporciona. En este capítulo se vio la manera en que las bibliotecas como GLee pueden dar acceso fácil a las extensiones, y al uso de las funcionalidades posteriores a OpenGL 1.1 para la plataforma Windows.

Lo que se aprendió

- Las extensiones existen para que los proveedores de hardware puedan innovar y agregar nuevas funciones con mayor rapidez
- El registro de OpenGL mantiene una base de datos con todas las especificaciones de extensión a OpenGL
- Durante la programación, los elementos clave para una extensión son la cadena, la función, los punteros y las fichas de la extensión.
- Las bibliotecas como GLee hacen que el uso y el manejo de las extensiones sea más sencillo.

Preguntas de repaso

1. ¿Qué función se utiliza para devolver una lista de extensiones compatibles, y cuál debe ser su primer argumento?
2. ¿Qué indica el prefijo ARB?
3. ¿En qué versión de OpenGL está disponible `glGetStringi()`?
4. ¿Para qué sirven las extensiones?

Por su cuenta

1. Escriba un programa que lea las extensiones disponibles en su versión de OpenGL y escriba la lista en un archivo de texto.

Cambiando a un diagrama (pipeline) programable

En los capítulos anteriores se han utilizado varios métodos de render —comenzando con el modo inmediato para después pasar a los vertex arrays (arreglos de vértices) y, por último, utilizar vertex arrays con VBO. Todos estos métodos emplean el diagrama de función fija, el cual se encuentra disponible en OpenGL y lo estará por algún tiempo; sin embargo, las técnicas de función fija serán eliminadas de manera gradual de la especificación OpenGL en favor del render basado en shaders. Las técnicas de shader y GLSL representan un tema muy amplio, por lo que más que ser una referencia completa, este capítulo proporciona una introducción breve.

En el presente capítulo, usted aprenderá:

- El lenguaje de sombreado (Shading Language) de OpenGL
- Cómo usar los shaders GLSL para hacer render de primitivas
- Cómo manejar sus propias matrices de transformación

El futuro de OpenGL

Durante la existencia de OpenGL, se han agregado muchas características para mejorar su desempeño al hacer render de escenas complejas. Por desgracia, la API se ha vuelto muy grande, con toda una gama de opciones para elegir en cada tarea de render específica. Esa gran cantidad de opciones hace que sea muy difícil determinar cuál técnica se desempeña de manera más eficiente; en otras palabras, se ha vuelto difícil encontrar lo que se conoce como la “vía rápida”.

La introducción del modelo de obsolescencia en OpenGL 3.0 promete corregir esta situación. La mayor parte de la API (específicamente la sección de función fija) ha sido marcada como obsoleta; las funciones que quedan proporcionan los métodos más rápidos para el render.

A continuación se presenta una lista de las principales funciones obsoletas en la versión 3.0. Si posee algún conocimiento previo de OpenGL (o leyó la primera edición de este libro), la siguiente lista puede resultarle interesante.

- Modo de Color Index
- Versiones 1.10 y 1.20 (ahora reemplazadas por la versión 1.30) del shading language en OpenGL
- Modo inmediato
- Procesamiento de vértices de función fija
- Pilas de matrices
- Vertex arrays del cliente
- Rectángulos
- Posición de raster
- Puntos non-sprite
- Líneas anchas y punteado de líneas
- Primitivas de cuadrilátero y polígono
- Modo de dibujo de polígonos separados
- Punteado de polígonos
- Dibujo de pixeles
- Mapas de bits
- Modo de textura de envoltura, `GL_CLAMP`
- Listas de visualización
- Búfer de selección
- Búfer de acumulación
- Pruebas alfa
- Pilas de atributos
- Evaluadores
- Cadena de extensión unificada

Esto es bastante ¿no es cierto? En la actualidad, la mayor parte de las funcionalidades anteriores se implementan mediante shaders, incluso algunas secciones (como la pila de matrices) pueden implementarse en bibliotecas independientes. Algunas de las características mencionadas en la lista ya no son importantes, debido a que han sido sustituidas por métodos más eficientes (por ejemplo, las listas de visualización o display lists) y otras han sido eliminadas debido a que en realidad no pertenecen a una representación de la API (por ejemplo, el búfer de selección). Al final de este capítulo, usted aprenderá cómo hacer render con funcionalidades actualizadas, a prueba de obsolescencia, reemplazando las características de función fija con las que se ha trabajado hasta ahora (los vertex arrays y la pila de matrices) por técnicas nuevas que emplean shaders.

¿Qué es GLSL?

El GLSL o Shading Language (lenguaje de sombreado) de OpenGL se utiliza para escribir programas que se ejecutan en la GPU. El lenguaje se define en un documento de especificación desarrollado por el ARB que puede encontrarse en el registro de opengl.org junto con la especificación OpenGL. GLSL es un lenguaje de alto nivel que toma muchos elementos de C y C++, por lo que la sintaxis debe resultarle muy familiar.

Existen dos tipos principales de shaders: los vertex shaders (sombreadores de vértice), que operan en cada vértice enviado a la tarjeta gráfica, y los fragment shaders (sombreadores de fragmento, también conocidos como pixel shaders o sombreadores de pixel), que operan sobre cada pixel que se va a representar.

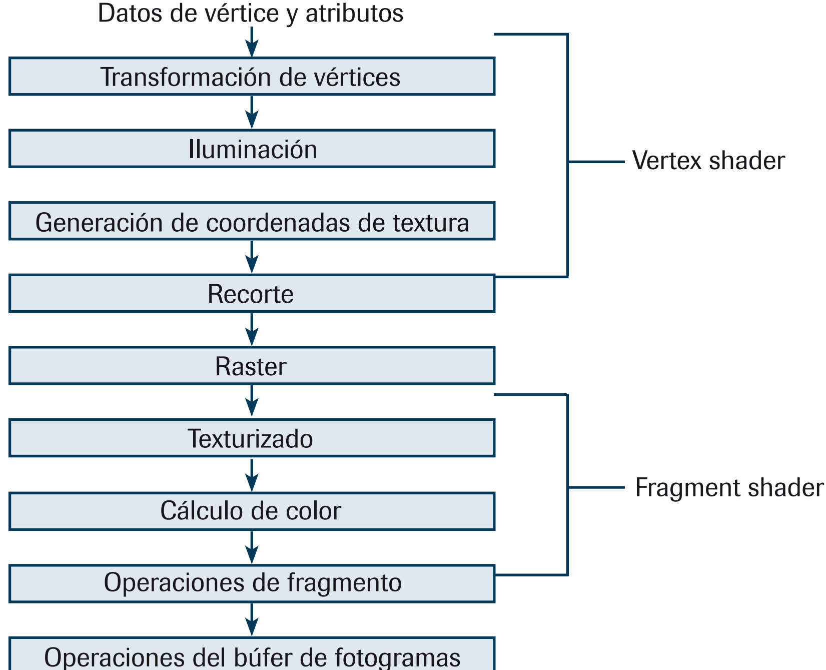
Al utilizar shaders, se sustituyen de manera efectiva todas las secciones de la segmentación, lo que proporciona una gran flexibilidad y control, pero eso quiere decir que, al usarlos, parte de la funcionalidad que operaba de manera automática mediante el render de función fija, ahora se convertirá en responsabilidad del usuario. En la figura 6.1 se muestran las etapas del diagrama que pueden reemplazarse con los shaders GLSL.

Vertex shaders

Como ya se estableció, OpenGL toma una serie de vértices para construir primitivas geométri-

Figura 6.1

Etapas del diagrama que pueden reemplazarse mediante shaders GLSL.



cas. Éstas después se transforman, recortan y dibujan para producir como salida los pixeles en el búfer del fotograma. Los vertex shaders son programas que operan en cada vértice que se envía a la tarjeta gráfica mediante un comando de render como `glDrawArrays()`. Este shader tiene la responsabilidad de calcular la posición final del vértice (normalmente, emplea las matrices de proyección y modelview), así como la de calcular los atributos para cada vértice, como el color. Un aspecto importante que debe señalarse es que un vertex shader funciona solamente para un vértice a la vez; no es posible extraer información sobre los vértices circundantes. La única salida que requiere un vertex shader es la posición. Ésta debe almacenarse en una variable integrada, que es un vector de punto flotante con cuatro elementos (`vec4`) llamado `gl_Position`. El shader puede dar salida a otras variables, que luego pueden utilizarse como entradas en el fragment shader. Por lo general, estas salidas se interpolan sobre la superficie de la primitiva y cada fragmento recibe el valor interpolado que corresponde a su posición sobre la superficie. Así, por ejemplo, imagine un vertex shader del que sale un solo valor de punto flotante. Si el shader opera sobre dos vértices vecinos y genera 0.0 para el primero y 1.0 para el segundo, entonces (suponiendo una interpolación lineal), el fragmento que se encuentra exactamente en el punto medio entre los dos vértices recibirá el valor 0.5 como entrada. Ésta es la forma en que los colores, las coordenadas de textura, etcétera, se interpolan suavemente a lo largo de la superficie. Después de completar la etapa de procesamiento de vértices, la tarjeta gráfica vuelve a tomar el control hasta la etapa de procesamiento de fragmentos. Cuando utilice un vertex shader, tendrá que manejar manualmente lo siguiente:

- Transformación de vértices (usando las matrices de proyección y modelview)
- Generación de coordenadas de textura
- Cálculos de iluminación
- Aplicación de color
- Transformación normal

La transformación de vértices y la aplicación de color se explican más adelante en este capítulo. Las coordenadas de textura se estudian en el capítulo 7 "Diseño de texturas" y la iluminación y las normales se tratan en el capítulo 8, "Mezcla, iluminación y niebla."

Nota

Como ya se analizó en el capítulo 1, hay un tercer tipo de shader llamado geometry shader (shader de geometría), éste es relativamente nuevo y en la actualidad sólo está disponible como una extensión. Por lo general, se utiliza para efectos avanzados que están fuera del alcance de este libro.

Fragment shaders

El fragment shader realiza la tarea de calcular el color de salida final de un pixel que se almacenará en el búfer de fotogramas. Los fragment shaders toman las salidas del vertex shader (que

pueden haber sido interpoladas) como variables de entrada. Estas entradas pueden usarse para dar color o textura al fragmento o para lograr efectos más avanzados, como la representación de vaporización y la iluminación por pixel. Los fragment shaders necesitan generar una única salida definida por el usuario, el vector de cuatro elementos que conforman el color del pixel. Cuando se utiliza un fragment shader es necesario controlar las siguientes partes de la tubería:

- Cálculo de colores por pixel
- Aplicación de texturas
- Cálculo de niebla por pixel
- Aplicación de iluminación por pixel

El lenguaje GLSL

Ahora que tiene una idea general de para qué sirve el GLSL, es momento abordar cómo usarlo. Antes de pasar a los detalles de cómo se cargan, compilan y utilizan los shaders en OpenGL Primero se estudiará el lenguaje en sí. Ha habido tres versiones de GLSL desde su introducción; la primera de ellas se denominó 1.10, la segunda 1.20 y finalmente, coincidiendo con el lanzamiento de OpenGL 3.0, se presentó la versión 1.30. En este capítulo se cubrirá la versión 1.30 ya que las versiones 1.10 y 1.20 se han marcado como obsoletas.

Estructura del shader

Un shader de GLSL debe contener al menos una sola función llamada `main()`. Ésta tiene el mismo objetivo que la función `WinMain()` o `main()` en un programa C o C++; se ejecuta cuando se inicia el shader, y éste termina cuando finaliza la función. La definición del método main es como sigue:

```
void main (void)
```

Como puede verse, la función `main()` no devuelve nada, cualquier salida que genera el shader se pasa a través de variables que tienen un calificativo de `out` (los calificadores de variables se estudiarán más adelante). Las variables que se pasan dentro o fuera de un shader deben declararse en el ámbito global. Es normal que se declare una lista de estas variables en la parte superior del shader antes de la función `main()`. A continuación se presenta un ejemplo de un vertex shader simple; aquí pueden observarse las variables mencionadas antes de la función `main()`. La primera línea es una directiva del pre-procesador.

```
#version 130

uniform mat4 projection_matrix;
uniform mat4 modelview_matrix;
```

```
in vec3 a_Vertex;
in vec3 a_Color;
out vec4 color;
void main (void)
{
    vec4 pos = modelview_matrix * vec4(a_Vertex, 1.0);
    gl_Position = projection_matrix*pos;
    color = vec4 (a_Color, 1.0);
}
```

Pre-procesador

GLSL toma la idea de un pre-procesador de C. Las directivas del pre-procesador se evalúan antes de la compilación del código fuente del shader y pueden alterar el código final que se compila. En la tabla 6.1 se muestra la lista completa de las directivas del pre-procesador y sus funciones.

Variables

Para cualquier lenguaje de programación, las variables son fundamentales y GLSL no es la excepción. Al igual que en C++, las variables de GLSL obedecen reglas de alcance (scope, en inglés). Un bloque de alcance se define dentro de un par de llaves ({...}), y las variables que se declaran dentro de esas llaves no tienen presencia fuera de dicho alcance. Una variable puede existir en un alcance local (entre dos llaves), en un ámbito global (en la parte superior del shader) o puede existir en más de un shader (ámbito del programa). Para que una variable pueda existir en dos shaders, ésta se debe declarar con los calificadores correctos en ambos shaders (por ejemplo, dentro y fuera).

Tabla 6.1 Directivas del pre-procesador

Directiva	Función
#	No tomar en cuenta
#define	Definir una constante o una macro
#undef	Eliminar la definición de una constante definida previamente
#if / #ifdef / #ifndef / #else / #elif / #endif	Compilación condicional
#error	Insertar un error en el registro de salida del shader
#pragma	Declarar opciones de implementación específica

Tabla 6.1 Directivas del pre-procesador (continuación)

Directiva	Función
`#line`	Anular la línea de numeración reconocida por el compilador
`#version`	Elegir la versión de GLSL a utilizar (la versión actual es 1.30)
`#extension`	Habilitar las funcionalidades extendidas disponibles para la versión de GLSL

Tipos de datos

Al igual que C y C++, GLSL es un lenguaje de tipos estáticos. Cada variable que se declara debe especificar un tipo de datos que conservará durante todo el periodo de vida de la variable. GLSL proporciona algunos tipos de datos conocidos como “float” para números de punto flotante e “int” para los números enteros, pero como vimos en el ejemplo del vertex shader, GLSL también tiene algunos tipos de base específicos para la programación de shaders. Estos tipos de datos incluyen vectores y matrices con dimensiones distintas. En la tabla 6.2 se presenta una lista de todos los tipos de datos en GLSL.

Tabla 6.2 Tipos de datos en GLSL

Tipo de datos	Descripción
`void`	Se usa como el tipo de resultado para funciones que no devuelven un valor
`bool`	Almacena un valor verdadero o falso
`int`	Un entero con signo
`uint`	Un entero sin signo
`float`	Un número de punto flotante
`vec2, vec3, vec4`	vector con dos, tres o cuatro elementos de punto flotante
`bvec2, bvec3, bvec4`	vector con dos, tres o cuatro elementos Booleanos
`ivec2, ivec3, ivec4`	vector con dos, tres o cuatro elementos enteros
`uvec2, uvec3, uvec4`	vector con dos, tres o cuatro elementos enteros sin signo
`mat2, mat3, mat4`	matriz de 2x2, 3x3 o 4x4 elementos de punto flotante
`mat2x2, mat3x3, mat4x4`	Equivalente a `mat2`, `mat3` y `mat4`

Tabla 6.2 Tipos de datos en GLSL (*continuación*)

Tipo de datos	Descripción
`mat2x3, mat2x4`	Matrices no cuadradas de punto flotante, con dos columnas
`mat3x2, mat3x4`	Matrices no cuadradas de punto flotante, con tres columnas
`mat4x2, mat4x3`	Matrices no cuadradas de punto flotante, con cuatro columnas
`sampler1D, sampler2D, sampler3D, usampler1D, usampler2D, usampler3D, isampler1D, isampler2D, isampler3D`	Handle o manejador para acceder a texturas de una, dos o tres dimensiones (acceso a punto flotante, entero sin signo, entero)
`samplerCube, usamplerCube, isamplerCube`	Handle o manejador para acceder a una textura representada mediante cubos
`sampler1DShadow, sampler2DShadow, samplerCubeShadow`	Handle o manejador para acceder a texturas de profundidad de una o dos dimensiones o a texturas de profundidad representadas mediante cubos
`sampler1DArray, sampler2DArray, usampler1DArray, usampler2DArray, isampler1DArray, isampler2DArray`	Handle o manejador para acceder a arreglos de texturas de una o dos dimensiones
`sampler1DArrayShadow, sampler2DArrayShadow`	Handle o manejador para acceder a arreglos de textura de profundidad con una o dos dimensiones

Nota

Los `samplers` representan un acceso hacia un mapa de textura, permiten la entrada a los datos de los píxeles de la textura y son necesarios para muchos efectos diferentes. Las texturas y los samplers se analizarán a detalle en el capítulo siguiente.

Antes de usar las variables por primera vez es necesario declararlas, lo cual se hace de la misma forma que en C, especificando el tipo antes del nombre de la variable. Una variable puede inicializarse al declararla de la siguiente manera:

```
int i = 0;
```

Los tipos de variable mas complejos (vectores y matrices) están formados de varios componentes; por ejemplo, un `vec2` se compone de dos elementos de punto flotante. Para los vectores, estos elementos se pueden abordar de forma individual mediante la adición de un punto (`.`) al nombre de variable para después usar el nombre de componente para seleccionar dicho componente. Para los vectores, los nombres de componente disponibles son los siguientes:

- `(x, y, z, w)`—Cuando se accede a vectores que representan puntos
- `(r, g, b, a)`—Para acceder a vectores que representan colores
- `(s, t, p, q)`—Para acceder a vectores que representan coordenadas de textura

La asignación de diferentes nombres de componente sólo tiene propósitos estéticos; `x`,`r` y `s` se refieren al mismo componente. Se puede acceder a los componentes de una matriz mediante una notación tipo arreglo:

```
mat4 [column][row]
```

`column` y `row` tienen que estar dentro de los límites de la matriz. Si se intenta acceder a algún punto fuera de los límites de la matriz, se obtiene un comportamiento indefinido.

Al igual que en otros lenguajes, en GLSL es posible definir arrays de variables. Estos arrays deben indexarse con una constante entera de base cero. Los índices negativos no están permitidos. Una declaración de array en GLSL es similar a su contraparte en C:

```
vec3 array[3]; //Un array de 3 vectores
```

Los arrays en GLSL pueden declararse inicialmente sin especificar un tamaño, aunque es necesario de render declararlos de nuevo con un tamaño determinado antes de usarlos por primera vez, por ejemplo:

```
vec3 array[];  //Declaración de un array sin tamaño específico
vec3 array[3]; //Declarado de nuevo incluyendo el tamaño del array antes de
su primer uso
```

Una vez que se ha especificado el tamaño, el array no puede declararse de nuevo. Los arrays de GLSL pueden inicializarse con los datos en la misma línea en que fueron definidos; la sintaxis en GLSL difiere ligeramente de C, ya que los tipos de GLSL usan constructores en vez de una lista de valores de inicialización:

```
float array[3] = float[3](0.0, 1.0, 2,0); //Inicializa los 3 flotantes
```

```
float array[3] = float[](0.0, 1.0, 2.0); //Hace lo mismo, el segundo 3 es op-
cional
```

Una vez declarada, la longitud de un array se puede determinar en el shader más adelante utilizando el método integrado `length()`:

```
float array[3];
int arrayLength = array.length(); //Después de ejecutar esto, arrayLength
será igual a 3
```

Estructuras

Los tipos de datos personalizados pueden definirse mediante la agrupación de variables en estructuras similares a las de C++. Una estructura debe contener al menos un miembro de datos. Las estructuras se definen mediante la palabra clave `struct` y un identificador, seguidos de las variables encerradas entre llaves:

```
struct Light {
    vec3 color;
    vec3 position
};
Light firstLight;
firstLight.color = vec3(0.0, 1.0, 2.0);
```

No está permitido el anidamiento de una declaración de estructura dentro de otra. Sin embargo, es perfectamente válido que una estructura contenga una instancia de otra estructura diferente:

```
struct B {
      struct A {//¡No es válido!
         int a;
    };
};

struct A {int a;}
struct B {
    A a; //OK.
};
```

Tabla 6.3 Operadores de GLSL

Precedencia	Clase de operador	Operadores	Asociatividad
1 (más alto)	agrupación entre paréntesis	( )	N/A
2	subíndice de array, llamada a la función y estructura del constructor, campo o método del selector, swizzler, incremento fijo de puesto y decremento fijo	[] ( ) . + + - -	De izquierda a derecha
3	incremento prefijo y decremento unitario	+ + - - + - ˜ !	De derecha a izquierda
4	multiplicativo	* / %	De izquierda a derecha
5	aditivo	+ -	De izquierda a derecha
6	cambio a nivel de bits	<<>>	De izquierda a derecha
7	relacional	<><=>=	De izquierda a derecha
8	igualdad	==!=	De izquierda a derecha
9	a nivel de bits y	&	De izquierda a derecha
10	a nivel de bits exclusivo o	^	De izquierda a derecha
11	a nivel de bits inclusivo o	|	De izquierda a derecha
12	lógico y	&&	De izquierda a derecha
13	lógico exclusivo o	^ ^	De izquierda a derecha
14	lógico inclusivo o	||	De izquierda a derecha
15	selección	? :	De derecha a izquierda
16	asignación, asignación aritmética	= += -= *= /= %= <<= >>= &= ^ = |=	De derecha a izquierda
17 (más bajo)	secuencia	,	De izquierda a derecha

Operadores

Los tipos de datos de GLSL son compatibles con los operadores básicos como suma, resta, multiplicación, división, asignación, igualdad y comparación (menor que, mayor que). En la tabla 6.3 se muestra la lista completa de todos los operadores GLSL disponibles:

Tal como cabría esperar, los operadores funcionan con números enteros, enteros sin signo y números de punto flotante. En los tipos de vectores que están formados por varios componentes (por ejemplo x, y, z), los operadores funcionan de componente en componente. La única excepción es la multiplicación que involucra matrices, la cual funciona con reglas estándar de álgebra lineal.

Al utilizar los operadores es necesario asegurarse de que los tipos en la expresión coincidan; por ejemplo, se puede multiplicar un número entero por otro entero, pero no por un número de tipo flotante. Si se desea mezclar los tipos, es necesario utilizar constructores para imitar el comportamiento de la fusión. Esto se analizará posteriormente en la sección "Constructores".

Tabla 6.4 Calificadores del tipo de almacenamiento

Calificador	Efecto
`None`	Una variable local que puede leerse y escribirse
`const`	Una variable que no se puede escribir
`in`	Una variable que se ha copiado de una etapa anterior (por ejemplo, una variable pasada desde el vertex shader)
`out`	Una variable que se copia a una fase posterior del procesamiento (por ejemplo, una variable pasada del vertex shader al fragment shader)
`uniform`	Una variable pasada desde la aplicación que no cambia durante el render de la primitiva actual
`centroid in`	Igual que in, pero usando la interpolación del centroide
`centroid out`	Igual que out, pero usando la interpolación del centroide

Calificadores de variable

Los calificadores de variable son aquellos que modifican de alguna manera el comportamiento de una variable. Cuando la variable no tiene un calificador se comportará de la manera esperada; la variable podría leerse y escribirse y respetaría el ámbito en el cual fue declarada. En la tabla 6.4 se muestra la lista completa de los calificadores del tipo no obsoleto y el efecto que tienen sobre la variable. La interpolación del centroide, que aparece en la tabla, a veces se utiliza para evitar los artificios que se presentan al utilizar el muestreo múltiple. Como éste es un tema avanzado, no se estudiará a detalle en el presente libro.

Es posible especificar calificadores adicionales para las variables de salida de los vertex shaders y las variables de entrada en un fragment shader. Estos calificadores adicionales se enlistan en la tabla 6.5 y afectan la interpolación de la variable.

El último tipo de calificador se utiliza en las variables que se pasan a las funciones. Estos calificadores pueden verse en la tabla 6.6.

Tabla 6.5 Calificadores de interpolación

Calificador	Significado
`smooth`	Interpolación correcta de perspectiva
`flat`	Sin interpolación
`noperspective`	Interpolación lineal

Tabla 6.6 Calificadores de parámetro

Calificador	Significado
`None`	El mismo que in
`in`	La variable es una entrada a la función
`out`	La variable pasada será el destino de la salida de una función
`inout`	La variable puede ser a la vez la entrada y la salida de la función

Los diferentes calificadores de variables pueden utilizarse en las siguientes situaciones:

- Las variables globales pueden usar `const`, `in`, `out` y `uniform`
- Las variables locales sólo pueden utilizar `const`
- Los argumentos de las funciones puede utilizar `const`, `in`, `out` e `inout`

Entradas del shader

Existen dos métodos para pasar valores de variables desde una aplicación de OpenGL: uniformes y atributos.

Uniformes

Una variable con un calificador uniforme obtiene su valor pasado en el shader desde la aplicación y permanece constante en todas las etapas del shader; los uniformes no cambian a cada vértice o fragmento y no pueden ser el destino de una asignación dentro del shader. Su valor sólo puede ser establecido por la aplicación y se puede tener acceso a ellos mediante todas las fases del programa del shader si la variable se declara de forma idéntica en cada shader. Los valores uniformes se transmiten en el programa del shader utilizando las funciones `glUniform*()` de la API de OpenGL, las cuales se describen más adelante.

Atributos de vértice

Un atributo de vértice (vertex attribute) es una variable global normal marcada con el calificador `in` en un vertex shader. La aplicación especifica el valor de un atributo de vértice empleando la función `glVertexAttribPointer()`. Esta función se analizará a detalle en la sección “Envío de datos a los shaders.”

Declaraciones

GLSL contiene las mismas declaraciones de control de flujo que se encuentran en C y C++. La lógica de ramificación se puede lograr usando las declaraciones `if` e `if-else`, las cuales se comportan de la misma manera que en C, con una pequeña diferencia, que es que no está permitido definir una variable dentro de una declaración `if`:

```
if (condition) //Esto está permitido
{
      //Hacer algo
}
if (bool someVariable = condition) //Esto no está permitido
{
      //hacer algo
}
```

Es posible lograr la lógica del ciclo si se utilizan las construcciones `for`, `while` y `do-while`, que se comportan de forma idéntica a sus homólogos en C++. Las variables pueden declararse dentro de los enunciados `for` o `while` y, por consiguiente, serán locales al ciclo.

Constructores

Los tipos de datos en GLSL tienen constructores integrados que pueden utilizarse para crear nuevas variables inicializadas con datos. Ya hemos visto un ejemplo del uso de los constructores en la sección de “Arreglos” al inicializar un arreglo con datos. Sin duda, usted usará una gran cantidad de constructores en el código de GLSL, no sólo para inicializar nuevas variables, sino también para copiar datos de una variable de un tipo a otro. Por ejemplo, suponga que tenemos un valor de color almacenado como un vector de tres elementos de punto flotante (`vec3`). Sin embargo, la salida de nuestro fragment shader espera un vector de cuatro elementos. Podemos copiar los datos a nuestra variable de salida mediante el uso de un constructor `vec4`, que toma dos argumentos, una variable `vec3` y un valor de punto flotante para el cuarto componente:

```
out vec4 ourColorOutput;
void main(void)
{
```

```
    vec3 color = vec3(1.0, 0.0, 0.0); //Un vector de 3
    elementos inicializado con un constructor

    //Un constructor se usa para copiar los datos a un
    vector de 4 elementos
    ourColorOutput = vec4(color, 1.0);
}
```

Tabla 6.7 Constructores de GLSL

Constructor	Propósito
`int(bool)`	Convierte un valor booleano en un número entero (el resultado es 1 si el valor es verdadero o 0 si es falso)
`int(float)`	Convierte un valor flotante en un entero (la parte decimal se elimina)
`int(uint)`	Convierte un entero sin signo en un entero con signo
`float(bool)`	Convierte un valor booleano en uno flotante (el resultado es de 1.0 si el valor es verdadero, o 0.0 si es falso)
`float(int)`	Convierte un valor entero en uno flotante
`float(uint)`	Convierte un valor entero sin signo en uno flotante
`bool(float)`	Convierte un valor flotante en uno booleano (distinto de cero es verdadero)
`bool(int)`	Convierte un valor entero en uno booleano (distinto de cero es verdadero)
`bool(uint)`	Convierte un valor entero sin signo en uno booleano (distinto de cero es verdadero)
`uint(bool)`	Convierte un valor booleano en un entero sin signo (el resultado es 1 si el valor es verdadero o 0 si es falso)
`uint(float)`	Convierte un valor flotante en un entero sin signo (la parte decimal se elimina, si el flotante es negativo entonces el comportamiento es indefinido)
`uint(int)`	Convierte un valor entero con signo en un entero sin signo
`vec2(float)`	Inicializa ambos componentes del vector para el valor pasado
`vec2(float, float)`	Inicializa el vector con los dos flotantes
`vec2(vec3)`	Elimina el último componente del `vec3` y construye un `vec2` con los componentes restantes
`vec3(float)`	Inicializa todos los componentes con el flotante pasado
`vec3(float, float, float)`	Inicializa el `vec3` con los tres flotantes
`vec3(vec4)`	Elimina el último componente del `vec4` y construye un `vec3` con los componentes restantes

Tabla 6.7 Constructores de GLSL (*continuación*)

Constructor	Propósito
`vec3(vec2, float)`	Construye un `vec3` usando el `vec2` como los dos primeros componentes y el flotante como el componente final
`bvec3(int, float, uint)`	Usa conversiones booleanas sobre cada parámetro
`mat2(float)`	Inicializa la diagonal de la matriz como flotante; todos los demás elementos se establecen en cero
`mat2(vec2, vec2)`	Inicializa las dos columnas usando los dos vectores
`mat2(float, float, float, float)`	Inicializa la matriz con los dos elementos de la primera columna, y después los dos elementos de la segunda columna

Existen muchos constructores diferentes dentro de cada clase, de manera que los objetos nuevos se pueden inicializar con diferentes tipos de datos. En la tabla 6.7 se enumeran los principales constructores que se utilizan de manera regular.

Por razones de espacio, la lista de la tabla 6.7 no es exhaustiva; los constructores de vectores pueden extenderse hasta `vec4` siguiendo las mismas reglas y los constructores de matriz se extienden a los diferentes tamaños de matriz (`mat3`, `mat4`, `mat3x2`, etcétera) siguiendo los mismos patrones de los parámetros disponibles.

Reducción

Algunos constructores le permiten pasar más de un tipo de argumento para construir un objeto a partir de la suma de sus componentes, por ejemplo, para construir una `vec4` pueden utilizarse un `vec3` y un flotante (un total de cuatro componentes). Sin embargo, en ocasiones podría desear construir un vector de tres componentes a partir de un vector de cuatro componentes, pero no necesariamente usando los tres primeros componentes. Por ejemplo, es posible que quiera construir un `vec3` con los componentes `y`, `z` y `w`. Tal vector puede construirse de la manera siguiente:

```
vec4 fourVec (1.0, 2.0, 3.0, 4.0);
vec3 threeVec = vec3 (fourVec.y, fourVec.z, fourVec.w);
```

GLSL proporciona un método abreviado para hacer esto, el cual se denomina “reducción” o “swizzling”. Mediante la reducción, usted puede hacer la misma conversión de esta manera:

```
vec4 fourVec (1.0, 2.0, 3.0, 4.0);
vec3 threeVec = fourVec.yzw;
```

La reducción funciona en todos los tipos de vectores, y se puede emplear cualquier combinación de nombres de los componentes provenientes del mismo conjunto de nombres (`xyzw`, `rgba` o `stpq`). A continuación se presentan algunos ejemplos:

```
vec4 vector;
vector.x: //Devuelve un flotante
vector.xyz: //Devuelve un vec3
vector.rg: //Devuelve un vec2
vector.xyza; //No se permite, no es parte del mismo conjunto de nombres
```

También se pueden asignar valores a algunos elementos utilizando la misma sintaxis para los componentes en el lado izquierdo de una asignación:

```
vec4 vector = vec4 (1.0, 2.0, 3.0, 4.0);
vector.xw vec2 = (1.0, 2.0); // ahora el vector es 1.0, 2.0, 3.0, 2.0
vector.xy vec3 = (0.0, 1.0); // ahora el vector es 0.0, 1.0, 3.0, 2,0
vector.xx vec2 = (1.0, 0.0); //No está permitido, no se puede utilizar
dos veces el mismo componente
```

Definición de funciones

Las funciones en GLSL pueden parecer muy conocidas ya que se declaran del mismo modo que en C, con la siguiente sintaxis:

```
returnType functionName(typeA arg1, typeB arg2, ... typeZ argn)
{
    return returnValue;
}
```

Las declaraciones de función difieren de C en que cada parámetro puede incluir uno de los siguientes calificadores: `in`, `out`, `inout` o `const` (mientras que C sólo tiene `const`). Las funciones en GLSL pueden sobrecargarse (como los métodos de C++); dos funciones pueden tener el mismo nombre pero diferentes parámetros y la versión correcta será llamada dependiendo de los parámetros que se le hayan asignado.

Funciones integradas

GLSL proporciona un gran número de funciones integradas que están disponibles automáticamente para usarse en un shader. Algunas de éstas son funciones simples de conveniencia que usted podría escribir por su cuenta (un ejemplo es `max()`). Otras proporcionan acceso a una funcionalidad de hardware que es imposible recrear en forma manual (por ejemplo, la fun-

ción `texture()`, que permite el acceso a una textura) y algunas están diseñadas para acelerarse mediante el hardware (por ejemplo, las funciones trigonométricas). Hay demasiadas funciones integradas como para cubrirlas en este capítulo, sin embargo en la tabla 6.8 se describen algunas de las más utilizadas. En la sección 8 de la especificación de GLSL se puede encontrar una lista completa de todas las funciones integradas.

Funciones obsoletas de GLSL

Antes de estudiar cómo se utilizan los shaders de GLSL en las aplicaciones OpenGL, tendremos que hablar brevemente de las funcionalidades obsoletas. A pesar de haber sido promovido hace muy poco tiempo al núcleo del programa, GLSL no escapó por completo a la ira del nuevo modelo de desaprobación u obsoleto. Las siguientes funciones se consideran obsoletas o fueron rediseñadas:

- Las palabras clave de atributo y variación fueron sustituidas por in y out.
- `gl_ClipVertex` fue sustituido por `gl_ClipDistance`.
- `gl_FragData` y `gl_FragColor` son obsoletos.
- Los atributos integrados se consideran obsoletos en favor de aquellos definidos por el usuario.
- La mezcla de funciones fijas de vértice o etapas de fragmento con programas de shaders es obsoleta. Los vertex shaders y los fragment shaders siempre deben usarse juntos.
- Todos los nombres de funciones de textura integradas han cambiado.
- `gl_FogFragCoord` y `gl_TexCoord` han sido sustituidas a favor de variables definidas por el usuario.
- La función integrada `ftransform()` se considera obsoleta.
- `gl_MaxVaryingFloats` ha sido sustituida por `gl_MaxVaryingComponents`.

Tabla 6.8 Funciones integradas en GLSL

Sintaxis	Descripción
`TYPE radians(TYPE degrees)`	Convierte grados en radianes
`TYPE degrees(TYPE radians)`	Convierte radianes en grados
`TYPE sin(TYPE angle)`	La función seno estándar
`TYPE cos(TYPE angle)`	La función coseno estándar
`TYPE tan(TYPE angle)`	La función tangente estándar
`TYPE acos(TYPE x)`	Función arco coseno
`TYPE asin(TYPE x)`	Función arco seno
`TYPE atan(TYPE y, TYPE x)`	Función arco tangente
`TYPE pow(TYPE x, TYPE y)`	Devuelve x elevada a la potencia y
`TYPE exp(TYPE x)`	Devuelve la potencia natural de x

Tabla 6.8 Funciones integradas en GLSL (*continuación*)

`TYPE log(TYPE x)`	Devuelve el logaritmo natural de x
`TYPE sqrt(TYPE x)`	Devuelve la raíz cuadrada de x
`TYPE abs(TYPE x)`	Devuelve x si x >= 0, en otro caso devuelve –x
`TYPE floor(TYPE x)`	Devuelve un valor igual al entero más próximo menor que x
`TYPE ceil(TYPE x)`	Devuelve un valor igual al entero más próximo mayor que x
`TYPE mod (TYPE x, float y)`	Devuelve x – y * `float(x/y)`
`TYPE min(TYPE x, TYPE y)`	Devuelve el valor más bajo entre x o y
`TYPE max(TYPE x, TYPE y)`	Devuelve el valor más alto entre x o y
`float length(TYPE x)`	Devuelve la longitud del vector x
`float distance(TYPE p0, TYPE p1)`	Devuelve la distancia entre p0 y p1
`float dot(TYPE x, TYPE y)`	Devuelve el producto escalar de x por y
`vec3 cross(vec3 x, vec3 y)`	Devuelve el producto cruz de dos vectores
`TYPE normalize(TYPE x)`	Devuelve un vector con una longitud de 1 y que tiene la misma dirección que x
`TYPE texture(SAMPLER sampler, TYPE p)`	Realiza una búsqueda de textura en el límite de la textura con el muestreador usando la coordenada de textura p

Uso de shaders

Para utilizar los programas de GLSL en su código, es necesario utilizar las funciones de la API C que fueron promovidos al núcleo de OpenGL 2.0. Para usar shaders de GLSL en su aplicación, es necesario llevar a cabo varios pasos, que son los siguientes:

1. Cree los shader objects (objetos de shader) —Esto consiste normalmente en crear un objeto de programa y dos shader objects (fragment y vertex).
2. Envíe la fuente a OpenGL—La fuente para cada shader se asocia con los shader objects correspondientes.
3. Compile los shaders.
4. Una los shaders al objeto de programa.
5. Vincule el programa.
6. Enlace el programa listo para usarse.
7. Envíe cualquier variable uniforme y los atributos de vértice.
8. Haga un render de los objetos que utilizan el programa.

Existen muchas formas diferentes de cargar el código fuente en su aplicación, pero en los ejemplos para este capítulo se supondrá que la fuente se almacena en una instancia `std::string` lista para funcionar.

Nota

Cuando la fuente de GLSL se carga desde el disco, tenga cuidado de preservar los caracteres de nueva línea ya que GLSL se basa en ellos durante la compilación. Si los caracteres de nueva línea se eliminan durante la carga de archivos, el shader presentará un error de compilación.

Creación de objetos en GLSL

A fin de preparar el programa de GLSL para su uso, lo primero que debe hacerse es generar los objetos que mantienen el estado del programa en OpenGL. Existen dos tipos de objetos que deberán usarse: los shader objects, que mantienen el código fuente, y los datos pertenecientes a los vertex shaders o fragment shaders y los objetos de programa contienen información relacionada con el programa GLSL en su conjunto.

Para crear los shader objects, se debe utilizar la función `glCreateShader()`, que tiene el siguiente prototipo:

```
GLuint glCreateShader(GLenum type);
```

La función `glCreateShader()` creará un nuevo shader object y devolverá un handle del mismo. En la actualidad, el tipo de parámetro puede ser `GL_VERTEX_SHADER` o `GL_FRAGMENT_SHADER` (aunque es probable que en el futuro haya otros tipos de parámetros disponibles). Lo más probable es que haya necesidad de hacer dos llamadas a esta función, una para cada tipo de shader. En algún momento, los shader objects tendrán que anexarse a un objeto de programa, que se crea de una manera similar usando la función `glCreateProgram()`:

```
GLuint glCreateProgram(void);
```

`glCreateProgram()` no toma ningún argumento y devuelve un handle para un objeto de programa. Ahora estamos listos para enviar el código fuente del shader a OpenGL. Esto se hace utilizando la función `glShaderSource()`:

```
void glShaderSource(GLuint shader, GLsizei count, const GLchar **string,
const GLint*length);
```

El primer parámetro es el shader object en el que se desea cargar el código fuente. `string` es un arreglo de cadenas al estilo C, y `count` es el número de cadenas en este arreglo. `length` es un arreglo que almacena la longitud en caracteres de las cadenas incluidas en el arreglo `string`. Si la longitud es `NULL`, se supone que todas las cadenas del arreglo terminan en cero (y por lo tanto no se requiere una longitud). Si los shaders se almacenan en una sola cadena de C++ (en vez de en un arreglo de cadenas), la fuente puede enviarse a OpenGL usando el siguiente código:

```
//Se crea un puntero temporal hacia la cadena
const GLchar*tmp = static_cast<const GLchar*> (m_vertexShader.source.c_
str());

//Envía la fuente a OpenGL, NULL indica que la cadena termina en cero
glShaderSource(m_vertexShader.id, 1, &tmp, NULL);
```

Una vez que el código fuente ha sido enviado a los shader objects, estamos listos para compilar los shaders; esto se hace mediante el comando `glCompileShader()`, que tiene la siguiente definición:

```
void glCompileShader(GLuint shader);
```

El shader object se pasa como el único argumento. `glCompileShader()` no devuelve un valor para indicar el éxito o el fracaso, por lo que para averiguar si la compilación del shader se realizó correctamente, es necesario consultar el estado de compilación usando la siguiente función:

```
void glGetShaderiv(GLuint shader, GLenum pname, GLint*params);
```

`glGetShaderiv()` toma el shader object como el primer parámetro. `pname` es una enumeración para especificar los datos que se desean recuperar en relación con el shader; éstos pueden ser cualesquiera de los valores de la tabla 6.9. El resultado se almacena en la variable hacia la que apunta `params`.

Por ejemplo, para comprobar si un shader se compiló con éxito, podría hacerse lo siguiente:

```
GLint result;
glGetShaderiv(shaderObject, GL_COMPILE_STATUS, &result);
```

Tabla 6.9 Valores pname de glGetShaderiv()

GLenum	Resultado
GL_COMPILE_STATUS	GL_TRUE si el shader se compila con éxito, o GL_FALSE en caso contrario
GL_SHADER_TYPE	GL_VERTEX_SHADER si el shader es un objeto de vertex shader o GL_FRAGMENT_SHADER si es un objeto de fragment shader
GL_DELETE_STATUS	GL_TRUE si el shader fue marcado para su eliminación; GL_FALSE en caso contrario
GL_INFO_LOG_LENGTH	La longitud en caracteres del registro de información incluyendo el caracter con terminación nula
GL_SHADER_SOURCE_LENGTH	La longitud en caracteres del código fuente de este shader

```
if(result == GL_TRUE)
{
    //El shader se compiló con éxito
}
```

Si la compilación falla por alguna razón, puede obtener información más detallada sobre este hecho al recuperar el registro de información adjunto al shader. Lo anterior se analizará más adelante en este capítulo.

Una vez que se hayan compilado los shaders, es posible añadirlos al objeto de programa. La función con la que se hace esto se llama `glAttachShader()`:

```
void glAttachShader(GLuint program, GLuint shader);
```

Si se intenta anexar dos veces el mismo shader al programa, OpenGL generará un error `GL_INVALID_OPERATION`. Es posible adjuntar los shader objects a un objeto de programa antes de haber sido compilados, o incluso antes de haber cargado el código fuente. `glAttachShader()` tiene una función contraparte llamada `glDetachShader()` la cual toma los mismos parámetros:

```
void glDetachShader(GLuint program GLuint shader);
```

Una vez que los shaders se hayan anexado al objeto de programa, será posible vincular el programa de GLSL. La fase de vinculación realiza comprobaciones de validez en el programa y lo prepara para ser usado. La vinculación puede fallar por varias razones:

- Uno de los objetos de shader no se ha compilado con éxito.
- El número de variables de atributo activas ha superado la cantidad máxima soportada por la versión de OpenGL.
- Se ha superado el número de variables uniformes activas o soportadas.
- La función principal no se encuentra en uno de los shaders anexados.
- Una variable de salida del vertex shader no se declaró correctamente en el fragment shader.
- Una función o variable de referencia no se puede resolver.
- Una variable global compartida entre etapas se declaró con diferentes tipos o valores iniciales.

Puede vincular un programa utilizando la siguiente función:

```
void glLinkProgram(GLuint program);
```

De nuevo, la función no devuelve información de si fue exitosa o no (porque el trabajo se realiza de forma asíncrona), pero se puede recuperar el estado del vínculo de una manera similar a la comprobación del estado de compilación de un shader. Para recuperar información sobre un objeto de programa, se utiliza `glGetProgramiv()`:

```
void glGetProgramiv(GLuint program, GLenum pname, GLint*params);
```

El primer parámetro es el objeto de programa que se desea consultar. `pname` puede ser cualquiera de los parámetros de la tabla 6.10. Cuando se completa la llamada a la función, el resultado se almacena en `params`.

Tabla 6.10 Valores pname para glGetProgramiv()

GLenum	Resultado
GL_DELETE_STATUS	Devuelve GL_TRUE si el programa está marcado para su eliminación, GL_FALSE en caso contrario
GL_LINK_STATUS	Devuelve GL_TRUE si el programa se vincula con éxito, GL_FALSE en caso contrario
GL_VALIDATE_STATUS	Devuelve GL_TRUE si la última operación de validación se realizó correctamente, GL_FALSE en caso contrario
GL_INFO_LOG_LENGTH	Devuelve la longitud en caracteres del registro de información del programa, incluyendo el caracter con terminación nula
GL_ATTACHED_SHADERS	Devuelve el número de shaders adjuntos al programa

Tabla 6.10 Valores pname para glGetProgramiv() (*continuación*)

GL_ACTIVE_ATTRIBUTES	Devuelve el número de atributos activos
GL_ACTIVE_ATTRIBUTE_ MAX_LENGTH	Devuelve la longitud del nombre del atributo activo más largo, incluyendo el caracter con terminación nula
GL_ACTIVE_UNIFORMS	Devuelve el número de uniformes activos
GL_ACTIVE_UNIFORM_MAX_LENGTH	Devuelve la longitud del nombre del uniforme activo más largo, incluyendo el caracter con terminación nula

Una vez que el programa ha pasado la etapa de vinculación, está listo para usarse. Un programa de GLSL puede activarse mediante la función `glUseProgram()`:

```
void glUseProgram(GLuint program);
```

Lo anterior vinculará y habilitará el programa. Cualquier primitiva que haya sido enviada a OpenGL mientras el programa está habilitado usará los shaders adjuntos para el render. Si el parámetro de programa que se pasa es 0, entonces el shader se desactivará.

Consulta de los registros de información

Como se ha mencionado, hay ocasiones en las que una compilación de shaders o una vinculación de un programa puede fallar por alguna razón. Para ayudar a diagnosticar el problema, OpenGL almacena un registro del error. Existe un registro de información adjunto a los shader objects y objetos de programa, el cual puede recuperarse mediante una de las siguientes funciones, dependiendo del tipo de objeto:

```
void glGetProgramInfoLog(GLuint program, GLsizei maxLength, GLsizei
*length, GLchar *infoLog);
void glGetShaderInfoLog(GLuint shader, GLsizei maxLength, GLsizei *length,
GLchar *infoLog);
```

El primer parámetro de cada función es el handle del objeto para el cual se está tratando de recuperar el registro. `maxLength` es el tamaño del búfer al que se desea copiar la información del registro; OpenGL copiará tanto del registro como pueda hasta llegar a `maxLength`. La longitud total de la cadena devuelta (excluyendo el terminador nulo) se almacena en `length`. El registro se copia en el búfer hacia el que apunta `infoLog`.

Envío de datos al shader

Hasta ahora se ha cubierto toda la información que se debe cargar, compilar y enlazar, el uso de shaders y la recuperación de información cuando algo sale mal. Sin embargo, los shaders no son útiles si no se les envía algún dato. Se ha mencionado antes que hay dos maneras de pasar datos desde una aplicación a un programa de GLSL: variables uniformes y atributos de vértice.

Paso de datos a variables uniformes

El envío de datos a variables uniformes consiste en un par de pasos. Cada implementación de GLSL tiene un número limitado de ubicaciones para almacenar variables uniformes. Al vincular un programa de GLSL, cada uniforme se adjunta a una de estas ubicaciones (la implementación de GLSL determina cuál uniforme va a cada ubicación). Antes de que sea posible enviar datos a un uniforme, primero es necesario conocer su ubicación. `glGetUniformLocation()` hace esto tomando el nombre de una variable uniforme como parámetro y devolviendo la ubicación como un entero sin signo. El prototipo es el siguiente:

```
GLuint glGetUniformLocation(GLuint program, const GLchar* name);
```

El primer parámetro es el objeto de programa, el segundo es el nombre de la variable según fue definida en los shaders.

Sugerencia

La obtención de la posición de un uniforme puede ser un proceso bastante lento, por lo que es una buena idea almacenar en caché el resultado de la búsqueda de la ubicación la primera vez que se realiza. Esto se puede hacer fácilmente mediante el uso de `std::map`. La clase `GLSLProgram` usada en los ejemplos demuestra cómo hacer esto.

Una vez que se haya recuperado la ubicación de una variable uniforme, entonces será posible enviar datos a la misma. Para enviar datos que comienzan con `glUniform`, existe una familia de funciones disponibles. Éstas tienen los siguientes prototipos:

```
void glUniform{1|2|3|4}{f|i}(GLint location, TYPE v);
void glUniform{1|2|3|4}ui(GLint location, TYPE v);

void glUniform{1|2|3|4}{f|i}v(GLint location, GLuint count, const TYPE *v);
void glUniform{1|2|3|4}uiv(GLint location, GLuint count, const TYPE *v);

void glUniformMatrix{2|3|4}fv(GLint location, GLuint count, GLboolean trans-
pose, const GLfloat *v);
```

```
void glUniformMatrix{2x3|3x2|2x4|4x2|3x4|4x3}fv(GLint location, GLuint
count, GLboolean transpose, const GLfloat *v);
```

`location` es la ubicación obtenida utilizando `glGetUniformLocation()`. En el caso de los flotantes, enteros, enteros sin signo y vectores, sólo se debe pasar la ubicación y los valores elegidos en una de las primeras parejas de funciones. A continuación se muestran algunos ejemplos:

```
glUniform1f(floatLocation, 1.0f);
glUniform3f(vectorLocation, 1.0f, 2.0f, 3.0f);
glUniform1i(integerLocation, 1);
glUniformui(unsignedIntLocation, 2);
```

Al pasar datos en arreglos uniformes, debe usarse una de las segundas parejas de funciones anteriores. En este caso `count` es el número de valores en el arreglo, y v es un puntero hacia un arreglo que contiene los datos. Así, por ejemplo, si su shader tenía definido el siguiente uniforme:

```
vec3 vecArray[4];
```

Éste podría recibir los datos del siguiente modo:

```
float data [] = { 1.0, 1.0, 1.0,
                  2.0, 2.0, 2.0,
                  3.0, 3.0, 3.0,
                  4.0, 4.0, 4.0 };
glUniform3fv(vecArrayLocation, 4, data);
```

El último par de funciones para enviar datos uniformes (las que comienzan con `glUniformMatrix`) se comportan de manera similar a la última serie, pero contienen un parámetro adicional denominado `transpose`. Si usted ha almacenado sus datos en el orden de columna mayor, entonces será necesario pasar `GL_FALSE` a `transpose`; en caso contrario, se debe pasar `GL_TRUE`.

Paso de datos a los atributos de vértice

Los atributos en GLSL son variables que se definen en el vertex shader con el calificador `in`. El paso de datos a los atributos de vértice es similar en algunas formas a pasar datos uniformes. Al igual que en los uniformes, GLSL proporciona un número de espacios para los atributos de vértice. Sin embargo, con los atributos de vértices, es posible dejar que OpenGL determine en qué ubicación almacenar el atributo, o bien especificar dicha localización en forma manual. Si usted prefiere dejar que OpenGL determine las ubicaciones de manera automática, puede recuperar el lugar de una variable utilizando la siguiente función:

```
GLint glGetAttribLocation (GLuint program, const GLchar*name);
```

Los argumentos son los mismos que los de glGetUniformLocation(). Al igual que glUniformLocation(), para que la función opere el programa tiene que haber sido vinculado.

Si prefiere especificar la ubicación de los atributos por su cuenta, puede hacerlo con glBind AttribLocation(). Esta función toma tres argumentos: el primero es el objeto de programa, después index es la ubicación que desea dar a este atributo y por último name es el nombre de la variable en el shader. El prototipo es el siguiente:

```
void glBindAttribLocation(GLuint program, GLuint index, const GLchar*name);
```

Las llamadas a glBindAttribLocation() deben hacerse antes de vincular el programa de GLSL, hasta entonces, las ubicaciones de los atributos no tendrán efecto. El atributo cero es especial y siempre debe utilizarse para la posición del vértice.

Una vez que se tiene la ubicación del atributo (ya sea generada automáticamente y después recuperada mediante una consulta o especificada manualmente), se pueden enviar los datos al atributo mediante glVertexAttribPointer():

```
void glVertexAttribPointer(GLuint index, GLint size, GLenum type, GLboolean
normalized, GLsizei stride, const GLvoid *pointer);
```

glVertexAttribPointer() es muy similar a las funciones del vertex array glVertexPointer(), glColorPointer(), etcétera. Esto es porque realizan un trabajo muy similar; glVertexPointer() establece el atributo integrado gl_Vertex (ahora obsoleto), mientras que glVertexAttribPointer() puede establecer cualquier atributo. index es la ubicación del atributo que se desea establecer, size indica el número de componentes por elemento (puede estar entre uno y cuatro). Type puede ser cualquiera de los siguientes: GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, GL_UNSIGNED_INT, GL_FLOAT o GL_DOUBLE. Si la bandera normalized es verdadera, entonces los datos se convierten en un valor de punto flotante entre —1.0 y 1.0 (para valores con signo) o 0.0 y 1.0 para los valores sin signo. stride especifica el desplazamiento en bytes entre los atributos del arreglo, y un valor de cero indica que los atributos se almacenan de manera consecutiva en el arreglo. Por último, pointer es un puntero hacia el arreglo de datos que se enviarán al atributo. Si se están empleando VBO (¡lo cual debería ser así!), el puntero será un desplazamiento entero en bytes hacia el búfer enlazado actualmente.

A fin de que los atributos del vértice tengan efecto, éstos deben estar habilitados antes del render; para hacer esto, debe utilizarse glEnableVertexAttribArray():

```
void glEnableVertexAttribArray(GLuint index);
```

El único argumento es la ubicación del atributo. Los atributos se puede desactivar con la correspondiente llamada a `glDisableVertexAttribArray()`:

```
void glDisableVertexAttribArray(GLuint index);
```

Hasta ahora se han cubierto los aspectos básicos que necesitan conocerse para usar shaders. En los capítulos siguientes, se utilizará GLSL casi exclusivamente para conseguir una serie de efectos como texturas, niebla, transparencia e iluminación.

La clase GLSLProgram

En el CD se incluye una clase llamada `GLSLProgram`, que facilita la carga y el uso de shaders. A partir de ahora, esta clase se utilizará en los ejemplos, por lo que a continuación se proporciona una breve explicación sobre su uso:

```
GLSLProgram* shaderProgram = new GLSLProgram("path/to/vertex/shader",
"path/to/fragment/shader");
//Carga los archivos del shader
if(!shaderProgram->initialize())
{
      //Algo salió mal
}

//Enlaza las ubicaciones del atributo
shaderProgram->bindAttrib(0, "a_Vertex");
shaderProgram->bindAttrib(1, "a_Color");

//Vincula de nuevo el programa
shaderProgram->linkProgram();
shaderProgram->bindShader(); //Habilita nuestro programa

//Envía algunos datos uniformes
shaderProgram->sendUniform("modelview_matrix", modelviewMatrix);
shaderProgram->sendUniform("projection_matrix", projectionMatrix);

//Cuando queda hecho:
delete shaderProgram;
```

Sustitución del diafragma de función fija

Ahora que se han cubierto los aspectos básicos del shading lenguaje (lenguaje de sombreado)

GLSL, es hora de saber cómo se usa para reemplazar el vértice de función fija y las etapas de fragmento en el diagrama. En la siguiente sección, estudiaremos cómo transformar las posiciones de vértice en el shader, y cómo pasar los colores a través del vertex shader y hacia el fragment shader.

Cálculo de las transformaciones de vértice

Como el vertex shader sustituye a toda la lógica de transformación del diagrama con funciones fijas, es responsabilidad del usuario hacer esto de manera manual en GLSL. Esto no es tan complicado como parece. En el ejemplo anterior se vio que pasamos el modelview y las matrices de proyección a nuestro shader como variables uniformes. Para encontrar la posición final del vértice, sólo es necesario multiplicar la posición local de éste por la matriz modelview y después multiplicar el resultado de esa transformación por la matriz de proyección:

```
//Primero se multiplica el vértice actual por la matriz modelview
vec4 pos = modelview_matrix*vec4(a_Vertex, 1,0);

//Después se multiplica el resultado por la matriz de proyección
gl_Position = projection_matrix * pos;
```

Al multiplicar matrices, el orden de la multiplicación es importante, pues si se hacen las multiplicaciones en un orden equivocado, se obtendrá un resultado incorrecto.

Sugerencia

En los ejemplos de GLSL, pasamos las matrices modelview y de proyección de manera individual para obtener una mayor claridad. Sin embargo, resulta más eficiente multiplicar las matrices modelview y de proyección una sola vez para crear una matriz modelview-proyección y después pasarla al shader, para luego multiplicarla por la posición del vértice en el shader. Esto nos ahorra una multiplicación de matrices en cada vértice procesado.

Aplicación de colores

La aplicación de colores a sus primitivas en GLSL es un proceso de dos pasos. En el vertex shader, debe leerse el atributo de entrada que contiene el color de los vértices. Usted tiene que pasar esto a su fragment shader utilizando una variable `out`. El color se interpolará antes de la entrada al fragment shader a menos que haya especificado el calificador `flat`. En éste, puede pasarse directamente este color como el final, o bien puede realizarse alguna lógica para cambiar el color (por ejemplo, para efectos de niebla, o para modularlo con una textura) antes de enviar el nuevo color a la salida. A continuación daremos un vistazo rápido a un ejemplo. En su aplicación OpenGL, debe enviar colores usando `glVertexAttribPointer()`; si tiene enlazado el atributo de color a la ranura 1 y los datos se almacenan en un VBO, el código será como el siguiente:

```
//Enlaza el arreglo de colores
glBindBuffer(GL_ARRAY_BUFFER, m_colorBuffer);
glVertexAttribPointer((GLint)1, 3, GL_FLOAT, GL_FALSE, 0, 0);
```

Cuando se hace el render de las primitivas, el color de cada vértice se almacenará en la variable del atributo de color, la cual se lee en el vertex shader:

```
#versión 130

uniform mat4 projection_matrix ;
uniform mat4 modelview_matrix;

in vec3 a_Vertex;
in vec3 a_Color; //El atributo de color que se pasó desde el programa
out vec4 color; //El color de salida que se pasará al fragment shader
void main(void)
{
    vec4 pos = modelview_matrix * vec4(a_Vertex, 1.0);
    gl_Position = projection_matrix * pos;
    color = vec4(a_Color, 1.0);
}
```

El color se asigna a la variable de salida (color) donde se utiliza como el color de fragmento en el fragment shader. Éste requiere dar salida a un vector de cuatro componentes que se usará como el color del píxel al final de la etapa de procesamiento del fragmento.

```
#versión 130

in vec4 color; //El color pasa desde el vertex shader (interpolado)
out vec4 outColor; //Define la salida de nuestro fragment shader

void main(void)
{
      outColor = color; //Copia el color a la salida
}
```

En la carpeta de recursos para este capítulo, hay una aplicación llamada "GLSL Terrain." Esta es una adaptación de la aplicación del terreno mostrada en el capítulo 5, que en vez de usar el procesamiento de función fija, hace todo su render a través de shaders simples. Los shaders se almacenan en archivos de texto en el directorio de datos, el vertex shader tiene la extensión de archivo .vert, y el fragment shader tiene la extensión de archivo .frag. Estos archivos pueden

editarse en cualquier editor de texto y ver el resultado inmediato al volver a ejecutar la aplicación. En el próximo capítulo, mejoraremos el programa aún más al añadir ¡textura al terreno!

Manejo de sus propias matrices

Ahora que se ha cubierto el render empleando una tubería programable, casi hemos dejado atrás la función fija de OpenGL (obsoleta). Sin embargo, hay un último tema que debemos abordar antes de que sus aplicaciones de OpenGL sean plenamente compatibles con futuras versiones. Hasta ahora hemos estado confiando en la funciones integradas para manejar la matriz modelview y las matrices de proyección. Si se observa el código fuente del ejemplo del terreno en GLSL podrá verse que pasamos las matrices modelview y de proyección a los shaders en forma manual, así:

```
//Obtiene las matrices actuales de OpenGL
glGetFloatv(GL_MODELVIEW_MATRIX, modelviewMatrix);
glGetFloatv(GL_PROJECTION_MATRIX, projectionMatrix);

//Envía las matrices modelview y de proyección a los shaders
m_GLSLProgram->sendUniform(“modelview_matrix”, modelviewMatrix);
m_GLSLProgram->sendUniform(“projection_matrix”, projectionMatrix);
```

Como puede verse, las matrices se recuperan de OpenGL. Si desea utilizar un contexto compatible con futuras versiones, necesita manejar las matrices por su cuenta, o usar una biblioteca de terceros que lo haga por usted.

La Biblioteca Kazmath

Kazmath es una biblioteca matemática de código abierto en 3D, que fue desarrollado por los encargados de NeHe (http://nehe.gamedev.net/): Carsten Haubold y Luke Benstead (¡uno de los co-autores de este libro!). Ofrece más de 100 funciones relacionadas con matemáticas que manipulan las estructuras básicas, como vectores y matrices. Una característica que ofrece la biblioteca es su propia pila de matrices, la cual puede utilizarse casi como un reemplazo exacto para las funciones de matriz en OpenGL. En el CD se incluye una versión de Kazmath, pero las versiones más recientes siempre pueden encontrarse en http://www.kazade.co.uk/kazmath/.

Ejemplo modificado del robot

En el CD puede encontrarse una versión del ejemplo del robot del capítulo 4, que sólo utiliza funcionalidades no obsoletas. GLSL reemplaza al render de función fija y la pila de matrices se sustituye con la biblioteca Kazmath. Por simplicidad, en los ejemplos restantes del libro se seguirán utilizando las pilas de matrices integradas en OpenGL, pero si desea emplear un contexto

compatible con versiones futuras, entonces el uso de la biblioteca Kazmath es una manera posible de manejar sus matrices.

Resumen

En este capítulo se ha cubierto una gran cantidad de material y algunos aspectos pueden resultar bastante abrumadores en este momento. Pero no tenga miedo, las cosas se volverán más claras en los capítulos siguientes a medida que pongamos en práctica el GLSL. En este capítulo, ha aprendido que GLSL es un lenguaje de programación utilizado para escribir pequeños programas que se ejecutan en la GPU de su tarjeta gráfica. Ahora debe entender que (en la actualidad) hay dos tipos principales de shaders que pueden utilizarse para sustituir las etapas del diagrama (o pipeline) con funciones fijas: vertex shaders y fragment shaders. Ya ha aprendido acerca de todos los componentes principales del shading language GLSL, incluidas las variables, sus tipos y calificadores, así como las funciones, declaraciones y constructores. Ahora debe ser capaz de cargar, compilar y utilizar sus propios shaders en OpenGL empleando las funciones de la API C relacionadas con GLSL. También se hizo una breve referencia a cómo pueden sustituirse totalmente las funcionalidades obsoletas restantes que se han utilizado hasta ahora, mediante una biblioteca de terceros.

Lo que se aprendió

- Los shaders son programas que se ejecutan en la GPU
- Los vertex shaders y fragment shaders son los dos tipos de shaders disponibles para su uso en OpenGL sin extensiones
- Los shaders de GLSL proporcionan una gran flexibilidad en comparación con el diagrama de función fija
- Para dar salida a la posición del vértice hacia la variable `gl_Position` se requiere un vertex shader, mientras que un fragment shader debe dar salida a un solo color de cuatro componentes
- Las variables de GLSL pueden pasarse entre etapas empleando los calificadores `in` y `out`
- Las variables uniformes pueden pasarse al programa de shader desde la aplicación
- Los atributos de vértice son variables que se pasan para cada vértice al programa de shader, usando la función `glVertexAttribPointer()`
- La pila de matrices de OpenGL puede reemplazarse mediante una biblioteca de terceros o por medio del manejo propio de las matrices modelview y de proyección

Preguntas de repaso

1. ¿Qué significa GLSL?
2. ¿Qué es un shader?
3. ¿Cómo se especifica la versión de GLSL necesaria para un shader?

4. ¿Qué es un uniforme?
5. ¿Cuál es la diferencia entre un uniforme y un atributo?
6. ¿Qué comando agrega un shader a un programa?
7. ¿Cómo se vincula un programa de GLSL?

Por su cuenta

1. Modifique el vertex shader en el programa del terreno, de manera que todo éste se dibuje en rojo, sin tomar en cuenta el color que se haya pasado.

Mapeo de texturas

Las escenas dibujadas en los capítulos anteriores sólo han usado colores sólidos para decorar las primitivas. Los colores son divertidos, ¡pero son poco realistas! El uso del mapeo de texturas puede proporcionar instantáneamente un gran salto hacia el realismo en nuestras escenas de render.

En este capítulo, usted aprenderá:

- Los fundamentos del mapeo de texturas
- Cómo crear, utilizar y eliminar objetos de textura
- La aplicación de texturas en GLSL
- Cómo usar mipmaps
- Filtrado de texturas
- Modos de envoltura en la textura
- Cómo cargar imágenes Targa (TGA)

Una perspectiva del mapeo de texturas

El mapeo de textura es un proceso en el cual se aplica una imagen sobre la superficie de una primitiva en vez de dibujar utilizando los colores básicos. Por ejemplo, si desea hacer un render de las paredes de una casa, la representación de éstas en un color liso sería muy poco atractivo y nada realista. El acto de aplicar una textura con patrón de ladrillos a la pared mejoraría mucho

la escena. En el ejemplo del terreno utilizado en los capítulos anteriores, el paisaje se hizo empleando diferentes tonos de verde en función de la altura de la colina. La escena se vería mucho más real si el terreno siguiera el modelo de hierba. El mapeo de textura es tan esencial para proporcionar realismo, que sería muy difícil encontrar un juego creado en los últimos 10 años en donde no se use.

Un mapa de textura es un arreglo rectangular de datos de color, y cada elemento de color se conoce como un *texel*. Aunque un mapa de textura es rectangular, puede asignarse a cualquier superficie mediante el uso de coordenadas de textura. La forma más común del mapa de textura es una imagen bidimensional, como una foto, la cual tiene un ancho y una altura determinadas. Algunos efectos requieren el uso de una textura unidimensional (con una anchura arbitraria y una altura de un texel) o incluso una textura tridimensional (con una anchura, cierta altura y determinada profundidad).

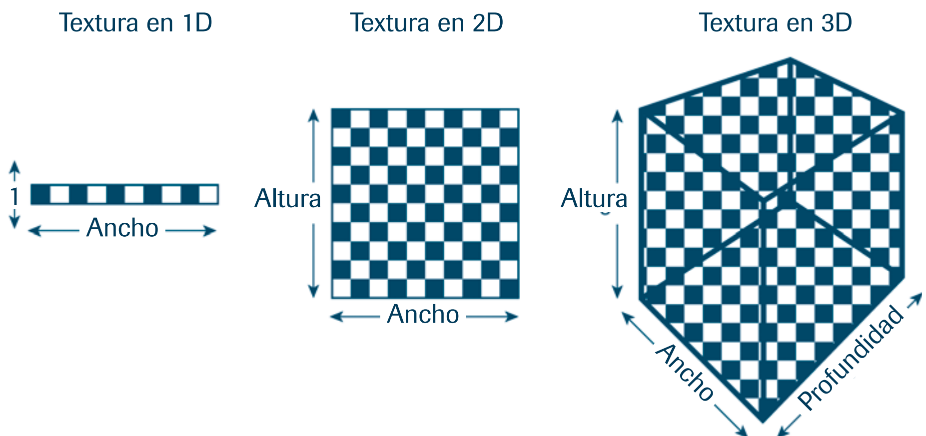
La aplicación de una textura sobre una superficie puede compararse con la impresión de una imagen en una hoja de papel; no importa en qué dirección se gire o se mueva el papel, la imagen fija se mantendrá en el mismo lugar y en la misma orientación que el papel.

Uso del mapa de textura

Existen varios pasos que deben seguirse para utilizar el mapeo de texturas:

1. Cargar la textura en la memoria.
2. Generar un objeto de textura en OpenGL.
3. Enlazar el objeto de textura (convertirlo en la textura actualmente activa).
4. Cargar los datos de imagen al objeto textura.
5. Especificar cualquier modo de filtrado/envoltura.

Figura 7.1
Texturas en una, dos y tres dimensiones.



6. Enviar los datos de coordenadas de textura a OpenGL.
7. Aplicar la textura en el fragment shader.

El primer paso consiste en almacenar los datos de imagen (los colores del texel, la anchura, la altura y la profundidad del color) en la memoria. Esta información puede cargarse desde un archivo de imagen o puede generarse mediante procedimientos de código. Antes de abordar la carga de datos de imagen desde un archivo, primero estudiaremos cómo se manejan las texturas en OpenGL.

Nota

Hay muchos formatos de imagen diferentes por ahí que puede utilizar para almacenar la textura en disco. Más adelante en este mismo capítulo, aprenderá cómo cargar el formato de imagen Targa en la sección "Carga de archivos de imagen Targa". Las imágenes Targa son muy adecuadas para el almacenamiento de texturas ya que, como se pueden comprimir, soportan un gran número de colores y cuentan con el apoyo del canal alfa (algo que es muy útil para las técnicas de mezclado como la transparencia), además tienen un formato fácil de leer.

Objetos de textura

Las texturas tienen una gran cantidad de información asociada que OpenGL necesita tomar en cuenta. Estos datos incluyen el tamaño de la textura, los datos de color, las opciones de filtrado, etcétera. OpenGL enlaza de manera interna estos datos entre sí como objetos de textura. OpenGL oculta el acceso directo a dichos objetos, pero es posible controlarlos con handles enteros asociados (también conocidos como "nombres de textura") de manera muy similar a los objetos de shader y de programa que se vieron en el capítulo anterior.

Creación de objetos de textura

Una vez que haya cargado los datos de imagen (ya sea desde un archivo o mediante un procedimiento de generación) en su aplicación, es necesario decirle a OpenGL que cree un objeto de textura donde se podrán cargar los datos. Un objeto de textura se crea de manera automática la primera vez que se enlaza un nombre de textura único. Puede generar nombres de textura únicos usando `glGenTextures()`:

```
void glGenTextures(GLsizei n, GLuint *textures);
```

`n` especifica el número de nombres de textura única que desea generar. Los nuevos nombres se almacenan en la variable a la que se apunta mediante textures. OpenGL etiqueta de manera

interna a cada nombre generado por glGenTextures() como si estuviera en uso. Esto sirve para que cada vez que se llame a la función, se pueda garantizar que los nombres generados son únicos. A continuación se presentan un par de ejemplos que muestran cómo generar nombres de textura:

```
GLuint firstTexture = 0;            //Variable de salida
glGenTextures (1, &firstTexture);   //Genera un nombre de textura único

GLuint textureNameArray[3];         //Un arreglo que contendrá 3 nombres
                                    de textura
glGenTextures(3, textureNameArray); //Genera nombres y los guarda en el
                                    arreglo
```

Una vez que haya generado un nombre único para su textura, debe enlazarlo antes de que OpenGL cree el objeto de textura asociado. Esto se hace utilizando la función glBindTexture():

```
void glBindTexture(GLenum target, GLuint texture);
```

target puede ser una de las siguientes constantes: GL_TEXTURE_1D, GL_TEXTURE_2D, GL_TEXTURE_3D, GL_TEXTURE_CUBE_MAP, TEXTURE_1D_ARRAY o TEXTURE_2D_ARRAY. Cada target corresponde a un tipo de textura diferente. OpenGL utiliza esta información para calcular la dimensionalidad de la textura. glBindTexture() le permite cambiar entre objetos de textura, de modo que en cada ocasión su estado correspondiente se considera el estado actual; ésta es la forma de hacer el render de un objeto con una textura y después cambiar a otra textura para un objeto diferente.

Eliminación de objetos de textura

Cuando haya concluido por completo el uso de un objeto de textura, éste debe ser eliminado. OpenGL asigna memoria para cada objeto de textura, y si no se eliminan, pueden ocurrir pérdidas de recursos. Los objetos de textura pueden eliminarse utilizando la función glDeleteTextures():

```
void glDeleteTextures(GLsizei n, GLuint *textures);
```

Una vez que un objeto de textura se ha eliminado, su nombre asociado está libre para ser reutilizado y puede devolverse en una llamada posterior a glGenTextures().

Especificación de texturas

Una vez creado un objeto de textura, puede copiar en él los datos de imagen. OpenGL proporciona una familia de tres funciones para hacer esto, la función utilizada dependerá de la dimensionalidad de la textura. Las funciones se llaman glTexImage1D(), glTexImage2D() y

glTexImage3D() para las texturas unidimensionales, bidimensionales y tridimensionales, respectivamente.

Nota

Hay una ocasión en la que la función glTexImage*() no coincide con la dimensionalidad de la textura que se suministra con los datos. Este caso especial se presenta al utilizar una característica conocida como *arreglos de textura* (texture arrays). Éstos proporcionan un medio para llenar con datos un arreglo de texturas en una sola llamada y acceder a ellos como un arreglo en el fragment shader. Los arreglos de textura están fuera del alcance de este libro por lo que no se estudiarán a detalle.

Texturas en 2D

Para especificar los datos de imagen a fin de obtener una textura bidimensional (que es por mucho el target de textura más común), se emplea glTexImage2D():

```
void glTexImage2D(GLenum target, GLint level, GLint internalformat, GLsizei width, GLsizei height, GLint border, GLenum format, GLenum type, const GLvoid *pixels)
```

target debe ser GL_TEXTURE_2D, GL_PROXY_TEXTURE_2D, o una de las constantes relacionadas con el mapeo de cubos: GL_TEXTURE_CUBE_MAP_POSITIVE_X, GL_TEXTURE_CUBE_MAP_POSITIVE_Y, GL_TEXTURE_CUBE_MAP_POSITIVE_Z, GL_TEXTURE_CUBE_MAP_NEGATIVE_X, GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, GL_TEXTURE_CUBE_MAP_NEGATIVE_Z o GL_PROXY_TEXTURE_CUBE_MAP. Los targets que comienzan con GL_PROXY se utilizan para probar si un determinado formato de textura es compatible y no se analizarán a detalle. Las constantes relacionadas con el mapa de cubos se estudiarán un poco más adelante. GL_TEXTURE_2D indica una textura bidimensional; éste es el parámetro target que se utilizará con más frecuencia.

El parámetro level se utiliza para generar mipmaps a diferentes niveles de detalle. Este parámetro se cubrirá en la sección llamada "mipmaps." El nivel básico es 0, que es lo que debería pasarse cuando no se están empleando mipmaps.

El parámetro internalformat especifica el número y tipo de componentes que forman la textura. Hay muchos formatos posibles para este parámetro, pero los más utilizados son: GL_RGB, GL_RGBA y GL_DEPTH_COMPONENT. También hay "formatos de tamaño de color," que extienden estos formatos básicos por tener un bit adicional para la profundidad deseada (por ejemplo, GL_RGBA8) y algunos formatos que incluyen una letra como sufijo a fin de representar un tipo de datos para los canales (por ejemplo, GL_RGBA32F). En la tabla 7.1 se muestran algunos valores comunes para internalformat.

Tabla 7.1 Formatos internos de textura más comunes

Formato	Descripción
GL_DEPTH_COMPONENT	Valores de profundidad
GL_RGB	Valores de rojo, verde y azul
GL_RGBA	Valores de rojo, verde, azul y alfa
GL_RGB8	Valores de rojo, verde y azul con 8 bits requeridos por canal
GL_RGBA8	Valores de rojo, verde, azul y alfa con 8 bits requeridos por canal
GL_RGBA32F	Valores de rojo, verde, azul y alfa con un almacenamiento de punto flotante requerido de 32 bits por canal

Tabla 7.2 Formatos de pixeles de textura

Formato	Descripción
GL_DEPTH_COMPONENT	Valores de profundidad
GL_RED	Valores de pixel rojo (R)
GL_GREEN	Valores de pixel verde (G)
GL_BLUE	Valores de pixel azul (B)
GL_ALPHA	Valores alfa (A)
GL_RGB	Valores de rojo, verde y azul (RGB)
GL_RGBA	Valores de rojo, verde, azul y alfa (RGBA)
GL_BGR	Valores de azul, verde y rojo (BGR)
GL_BGRA	Valores de azul, verde, rojo y alfa (BGRA)

Nota

El uso de formatos internos que especifiquen una profundidad de bits de forma predeterminada se considera una buena idea, porque algunas implementaciones de OpenGL pueden usar menos de 8 bits por canal. Observe que los formatos que especifican una profundidad de bits son solicitudes; OpenGL pueden pasar por alto el valor de la profundidad de bits.

`width` y `height` son las dimensiones del mapa de textura especificado.

El parámetro `border` se considera obsoleto. Si se establece en 1, OpenGL dibujará un borde alrededor de la textura. Para obtener compatibilidad con versiones futuras, es necesario establecerlo en cero. Las versiones futuras de OpenGL, generarán un error `INVALID_VALUE` si este parámetro es distinto de cero.

El parámetro `format` indica el formato de los datos de imagen que se pasarán como el último parámetro de esta función. Los valores más comunes se enlistan en la tabla 7.2.

Tabla 7.3 Tipos más comunes de datos de textura

Formato	Descripción
GL_UNSIGNED_BYTE	Entero de 8 bits sin signo
GL_BITMAP	Un solo bit (0 o 1)
GL_BYTE	Entero de 8 bits con signo
GL_UNSIGNED_SHORT	Entero de 16 bits sin signo (2 bytes)
GL_SHORT	Entero de 16 bits con signo (2 bytes)
GL_UNSIGNED_INT	Entero de 32 bits sin signo (4 bytes)
GL_INT	Entero de 32 bits con signo (4 bytes)
GL_HALF_FLOAT	Tipo de punto flotante de 2 bytes
GL_FLOAT	Punto flotante de precisión simple (4 bytes)
GL_UNSIGNED_BYTE_3_3_2	Empacado en un entero sin signo de 8-bits. R3, G3, B2
GL_UNSIGNED_BYTE_2_3_3_REV	Empacado en un entero sin signo de 8-bits. B2, G3, R3
GL_UNSIGNED_SHORT_5_6_5	Empacado en un entero sin signo de 16-bits. R5, G6, B5
GL_UNSIGNED_SHORT_5_6_5_REV	Empacado en un entero sin signo de 16-bits. B5, G6, R5
GL_UNSIGNED_SHORT_4_4_4_4	Empacado en un entero sin signo de 16-bits. R4, G4, B4, A4
GL_UNSIGNED_SHORT_4_4_4_4_REV	Empacado en un entero sin signo de 16-bits. A4, B4, G4, R4
GL_UNSIGNED_SHORT_5_5_5_1	Empacado en un entero sin signo de 16-bits. R5, G5, B5, A1
GL_UNSIGNED_SHORT_1_5_5_5_REV	Empacado en un entero sin signo de 16-bits. A1, B5, G5, R5
GL_UNSIGNED_INT_8_8_8_8	Empacado en un entero sin signo de 32-bits. R8, G8, B8, A8
GL_UNSIGNED_INT_8_8_8_8_REV	Empacado en un entero sin signo de 32-bits. A8, B8, G8, R8
GL_UNSIGNED_INT_10_10_10_2	Empacado en un entero sin signo de 32-bits. R10, G10, B10, A2

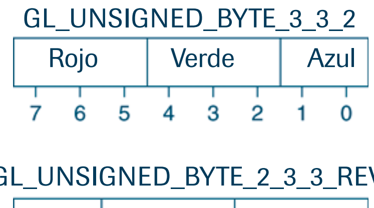
Tabla 7.3 Tipos más comunes de datos de textura (continuación)

Formato	Descripción
`GL_UNSIGNED_INT_2_10_10_10_REV`	Empacado en un entero sin signo de 32-bits. A2, B10, G10, R10
`GL_UNSIGNED_INT_24_8`	Empacado en un entero sin signo de 32-bits. D24, S8*

El parámetro `type` define el tipo de datos de imagen a utilizar. Éste puede tomar cualquiera de los valores de la tabla 7.3.

Los valores empacados de la tabla 7.3 son los formatos donde los canales individuales de color se empacan en un solo tipo de datos. Por ejemplo, cuando se utiliza el formato GL_UNSIGNED_BYTE_3_3_2, tres canales de color se almacenan en un solo byte. El canal rojo ocupa los tres bits más significativos, seguido por el canal verde, que también tiene tres bits, y finalmente el canal azul, que llena los otros dos bits (los bits menos significativos del byte). Si el formato termina en _REV entonces el orden de los canales de color se invierte. GL_UNSIGNED_BYTE_2_3_3_REV almacena los canales de modo que el azul toma los dos bits más significativos, seguido por el verde y el rojo, que tienen los tres bits menos significativos. En la figura 7.2 se muestra cómo funciona el empacado de datos.

Figura 7.2
Tipo de datos empacados para la configuración de bits.



Nota

En realidad existen tres formatos de empacado que no se incluyen en la tabla 7.3. Éstos son GL_UNSIGNED_INT_10F_11F_11F_REV, GL_UNSIGNED_INT_5_9_9_9_REV y GL_FLOAT_32_UNSIGNED_INT_24_8_REV. Estos tipos de empacado conducen a un proceso de conversión más complicado que los de la tabla 7.3, el cual no se estudiará por encontrarse fuera del alcance de este libro.

El parámetro final para glTexImage2D() es pixels, que es un puntero hacia los datos de imagen almacenados en memoria. Los datos de imagen se leerán usando el formato indicado por type.

A modo de ejemplo, si usted tiene una imagen con una anchura y altura de 128 y sus datos de imagen se almacenan en un arreglo de bytes sin signo (ImageData), donde a cada canal de color (*r, g, b, a*) se le asignan 8 bits, puede especificar la imagen para OpenGL con la siguiente llamada:

```
glTexImage2D(GL_TEXTURE_2D, GL_RGBA8, 128, 128, 0, GL_RGBA, GL_UNSIGNED_
BYTE, imageData);
```

Después de esta llamada, la textura se cargará en el objeto de textura actualmente enlazado y estará listo para su uso.

Texturas en 1D

La especificación de texturas en 1D es muy similar a la de las texturas en 2D. La única diferencia entre los dos tipos es que las texturas en 1D siempre tienen una altura de 1, éstas pueden utilizarse para producir efectos de shaders, como Cel-Shading (un estilo de render para dibujos animados). Una textura en 1D se especifica con glTexImage1D():

```
void glTexImage1D(GLenum target, GLint level, GLint internalformat, GL-
sizei width, GLint border, GLenum format, GLenum type, const GLvoid
*pixels)
```

Las únicas diferencias entre esta función y glTexImage2D() son las siguientes:

- No existe un parámetro de altura
- Se debe especificar GL_TEXTURE_1D como el parámetro target

Texturas en 3D

Se puede pensar en las texturas en 3D como una serie de texturas en 2D sobrepuestas una a la otra. Por lo general, las texturas en 3D se generan mediante un procedimiento y se accede a

ellas usando coordenadas 3D. Los datos de las texturas en 3D se especifican mediante la función `glTexImage3D()`:

```
void glTexImage3DEXT(GLenum target,Glint level, GLenum internalformat, GLsi-
zei width, GLsizei height, GLsizei depth, Glint border, GLenum format, GLe-
num type, const GLvoid *pixels)
```

Una vez más, los parámetros son los mismos que en `glTexImage2D()`, con la excepción del parámetro adicional `depth`, que especifica la tercera dimensión de la textura.

Texturas de mapa de cubos

Una textura de mapa de cubos es un tipo especial de target de textura. Los mapas de cubos se componen de seis texturas individuales en 2D. Por lo general se utilizan con una coordenada de textura tridimensional, la cual forma un vector de dirección que apunta desde el centro de un cubo al texel requerido. La búsqueda del texel se realiza en dos etapas. Primero, dada la coordenada de textura en 3D (`s, t, r`), la magnitud más alta de los tres componentes de la coordenada de textura se emplea para determinar cuál de las seis texturas del cubo se utilizará. Después, una vez determinada la textura en 2D, los componentes de la coordenada de textura en 3D se utilizan para calcular una coordenada de textura en 2D (`s, t`). Cada una de las texturas laterales en el mapa de cubo se especifica mediante `glTexImage2D()` junto con uno de los valores de `GL_TEXTURE_CUBE_MAP *target`. Las texturas que componen el mapa de cubo deben ser cuadradas (es decir, su ancho y altura deben ser iguales). Para acceder a un mapa de cubo en un shader, es necesario usar uno de los samplers de textura en mapas de cubos (vea la tabla 6.2).

Filtrado de texturas

Cuando se asigna una textura a un polígono, es muy poco probable que un solo pixel asigne a la imagen los texeles de uno en uno. Si la imagen se observa cerca de la ventana de visualización, un pixel puede tomar sólo una pequeña parte del texel asignado (la situación se conoce como *magnificación*). Por el contrario, si la textura está muy lejos de la ventana de visualización entonces un solo píxel puede contener varios texeles (esto se conoce como *minificación*). En estas situaciones, OpenGL debe calcular el color del pixel; el comportamiento de este cálculo se controla mediante el *filtrado de textura*.

Es posible indicarle a OpenGL cómo manejar el filtrado de texturas mediante las funciones `glTexParameter()`:

```
void glTexParameter{if}(GLenum target, GLenum pname, TYPE param)
void glTexParameter{if}v(GLenum target, GLenum pname, TYPE param)
```

`target` debe ser `GL_TEXTURE_1D`, `GL_TEXTURE_2D`, `GL_TEXTURE_1D_ARRAY`, `GL_`

TEXTURE_2D_ARRAY, GL_TEXTURE_3D o GL_TEXTURE_CUBE_MAP. Para el filtrado de texturas pname debe ser GL_TEXTURE_MAG_FILTER o GL_TEXTURE_MIN_FILTER dependiendo de la situación de filtrado para la que se desea establecer el modo. Al configurar el filtro de magnificación, los posibles valores de param son GL_LINEAR o GL_NEAREST. Si se usa GL_NEAREST como el filtro de magnificación se le dice a OpenGL que utilice el texel más cercano al centro del pixel para el color final. En ocasiones, esto se denomina *muestreo de punto* (point sampling). Es el método de filtrado más económico y puede resultar en texturas que aparentan pertenecer a un solo bloque. Al establecer GL_LINEAR se le dice a OpenGL que use el promedio ponderado de los cuatro texeles más cercanos al centro del pixel. Este tipo de filtrado da como resultado texturas suaves y también se conoce como *filtrado bilineal*.

Cuando se establece el valor del filtrado de minificación, existen más valores posibles, los cuales se enlistan en la tabla 7.4 y se ordenan según su calidad de render.

Tabla 7.4 Valores para el filtrado de texturas de minificación

Filtro	Descripción
GL_NEAREST	Usa el texel más cercano al centro del pixel que se dibujará.
GL_LINEAR	Usa interpolación bilineal.
GL_NEAREST_MIPMAP_NEAREST	Usa el nivel de mipmap más cercano a la resolución del polígono y utiliza el filtrado GL_NEAREST en ese nivel.
GL_NEAREST_MIPMAP_LINEAR	Usa el nivel de mipmap más cercano a la resolución del polígono y utiliza el filtrado GL_LINEAR en ese nivel.
GL_LINEAR_MIPMAP_NEAREST	Usa el muestreo GL_NEAREST en los dos niveles más cercanos a la resolución del polígono, y después interpola linealmente entre los dos valores.
GL_LINEAR_MIPMAP_LINEAR	Usa el filtrado bilineal para obtener muestras de los dos niveles más cercanos a la resolución del polígono, y después hace una interpolación lineal entre ambos valores. Esto también se conoce como *filtrado trilineal*.

Los cuatro filtros relacionados con mipmaps tendrán más sentido después de haber estudiado la sección “Mipmaps” que se incluye más adelante en este capítulo.

La configuración predeterminada del filtrado para una textura se establece con GL_LINEAR para el filtro de magnificación y con GL_NEAREST_MIPMAP_LINEAR para el filtro de minificación. Si no se están utilizando mipmaps con la textura, es necesario cambiar el filtro de minificación

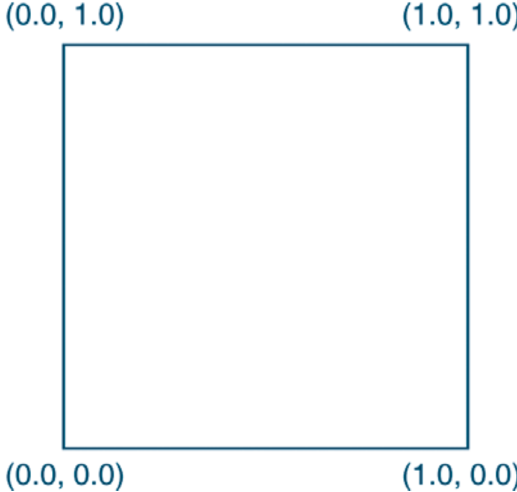
a GL_LINEAR o GL_NEAREST; de no hacerse esto, el texturizado no funcionará correctamente porque el modo de filtrado predeterminado requiere que se hayan generado todos los niveles de mipmap.

Coordenadas de textura

En los capítulos anteriores se ha hecho referencia varias veces a las coordenadas de textura y éste es el momento de abordarlas en detalle. Las texturas (a diferencia de la mayoría de las superficies a las que se aplica el render) tienen una forma rectangular por lo que debe haber algún método de mapeo que permita transferirlas a un polígono arbitrario. Las coordenadas de textura se usan para determinar si cada parte de la textura debe aplicarse a una cara del polígono. A cada esquina de una textura se le asigna una coordenada en 2D con (0.0, 0.0) para la esquina inferior izquierda y (1.0, 1.0) para la esquina superior derecha. Al hacer el render de una primitiva, las coordenadas de textura se especifican para cada vértice y después se interpolan para el procesamiento de fragmentos. Durante dicho procesamiento, las coordenadas interpoladas se utilizan para buscar el color del pixel en el mapa de textura enlazado actualmente. Mientras los componentes de vértices y vectores suelen etiquetarse como x, y, z y w, las coordenadas de textura se refieren generalmente como s, t, r y q (la excepción a esta regla es la nomenclatura de componentes vectoriales en los shaders de GLSL, donde r se sustituye por p para evitar un conflicto con la nomenclatura de componentes rgba). En la figura 7.3 se muestra la asignación de coordenadas para una textura en un polígono simple.

Figura 7.3

Valores de las coordenadas de textura en un polígono.



Nota

Aunque la mayor parte de las veces las coordenadas de textura oscilarán entre 0.0 y 1.0, hay ocasiones en las que los valores pueden ser mayores que eso. Estos valores más altos se analizarán con detalle más adelante en este capítulo, dentro de la sección "Modos envoltura de textura".

Las texturas pueden esparcirse a través de varios polígonos, indicando las coordenadas de textura para que sólo una parte se represente en cada polígono. A modo de ejemplo, normalmente un cuadrilátero se forma mediante dos triángulos. Si desea lograr una textura cuadrada sin divisiones, tendrá que especificar las coordenadas de textura correctas para los dos triángulos. Vea la figura 7.3 e imagine la diagonal que se formaría si el cuadro estuviera formado por dos triángulos. La textura se asignaría correctamente siempre y cuando cada triángulo especificara las mismas coordenadas de textura para los mismos vértices.

Aplicación de las coordenadas de textura

Las coordenadas de textura son un atributo de un vértice, por lo que si usted está usando GLSL, debe especificar las coordenadas de textura usando `glVertexAttribPointer()`. Debe enviar a GLSL los datos de la coordenada de textura como un arreglo, en la misma forma que los datos de color. Para aplicar las coordenadas de textura en GLSL es necesario hacer lo siguiente:

1. En el vertex shader, es necesario declarar una variable `out` para las coordenadas de vértice actuales, las cuales se interpolan y se utilizan como entrada en el fragment shader.
2. El vertex shader debe leer la coordenada de entrada y asignársela a la variable de salida.
3. Después, el fragment shader utilizará esta coordenada para realizar una búsqueda de textura para el fragmento.

A continuación se verá un ejemplo concreto en GLSL. Primero, se presenta el código de un vertex shader que realiza el texturizado:

```
#version 130

uniform mat4 projection_matrix;
uniform mat4 modelview_matrix;

in vec3 a_Vertex;
in vec3 a_Color;
in vec2 a_TexCoord0;
```

```
out vec4 color;
out vec2 texCoord0;

void main(void)
{
    texCoord0 = a_TexCoord0;
    color = vec4(a_Color, 1.0);
    vec4 pos = modelview_matrix * vec4(a_Vertex, 1.0);
    gl_Position = projection_matrix * pos;
}
```

El vertex shader lee el atributo de entrada (`a_TexCoord`) y lo asigna a la variable de salida (`texCoord0`). El trabajo real de aplicación de la textura se lleva a cabo en el fragment shader:

```
#version 130

uniform sampler2D texture0;

in vec4 color;
in vec2 texCoord0;

out vec4 outColor;

void main(void) {
  outColor = color * texture(texture0, texCoord0.st);
}
```

Aquí puede observarse el tipo de `sampler2D` que se ha utilizado. Los samplers proporcionan acceso a una unidad de textura. Al aplicarlos, usted requiere establecer el uniforme del sampler (en este caso texture0) en la unidad a la que está enlazada la textura. Las unidades de textura se analizarán con más detalle al estudiar las texturas múltiples en el capítulo 9, "Más sobre el mapeo de texturas." Por ahora, todo lo que debe saber es que la unidad de textura predeterminada es 0; de modo que en su aplicación puede establecer el sampler utilizando la clase GLSLProgram del modo siguiente:

```
m_GLSLProgram->sendUniform("texture0", 0);
```

La coordenada de textura que pasa al fragment shader habrá sido interpolada para el fragmento actual. A fin de obtener el color del texel con base en la textura y usando la coordenada de textura para este fragmento, puede utilizar la función `texture()`:

```
vec4 texture(sampler1D sampler, float P)
vec4 texture(sampler2D sampler, vec2 P)
vec4 texture(sampler3D sampler, vec3 P)
vec4 texture(samplerCube sampler, vec3 P)
float texture(sampler1DShadow sampler, vec3 P)
float texture(sampler2DShadow sampler, vec3 P)
float texture(samplerCubeShadow sampler, vec4 P)
vec4 texture(sampler1DArray sampler, vec2 P)
vec4 texture(sampler2DArray sampler, vec3 P)
float texture(sampler1DArrayShadow sampler, vec3 P)
float texture(sampler2DArrayShadow sampler, vec4 P)
```

`sampler` es el sampler de textura que se usa para buscar el texel, y `P` es la coordenada de textura empleada para localizar el texel. El color de texel devuelto toma en cuenta los modos de filtrado de texturas. El color de la textura puede ser devuelto directamente por el fragment shader o bien usted puede combinar el color con otras variables. En el ejemplo anterior, se multiplica el color del texel por el color del fragmento interpolado; el resultado es una combinación de ambos.

Parámetros de textura

Cuando se analizaron los modos de filtrado de texturas, se utilizó `glTexParameter()` para establecer los filtros de magnificación y minificación. Pero éste no es el único uso para esa función. Veamos de nuevo su definición:

```
void glTexParameter{if}(GLenum target, GLenum pname, TYPE param)
void glTexParameter{if}v(GLenum target, GLenum pname, TYPE param)
```

Existen otros valores posibles para `pname` y `param` que no están relacionados con el filtrado de texturas sino con la alteración del modo en que se aplica la textura actualmente enlazada. En la tabla 7.5 se muestra una lista de valores posibles para `pname` y los valores no obsoletos que pueden establecerse para `param`.

Modos de envoltura de textura

Los modos de envoltura de textura le permiten modificar la forma en que OpenGL interpreta las coordenadas de textura fuera del rango [0, 1]. Al usar la función `glTexParameter()` con `GL_TEXTURE_WRAP_S`, `GL_TEXTURE_WRAP_T` o `GL_TEXTURE_WRAP_R`, puede especificar la manera en la que OpenGL interpreta las coordenadas `s`, `t` y `r`, respectivamente.

Tabla 7.5 Parámetros de textura

Nombre	Tipo	Valores
GL_TEXTURE_WRAP_S	entero	GL_CLAMP_TO_EDGE, GL_REPEAT, GL_MIRRORED_REPEAT
GL_TEXTURE_WRAP_T	entero	GL_CLAMP_TO_EDGE, GL_REPEAT, GL_MIRRORED_REPEAT
GL_TEXTURE_WRAP_R	entero	GL_CLAMP_TO_EDGE, GL_REPEAT, GL_MIRRORED_REPEAT
GL_TEXTURE_MIN_FILTER	entero	GL_NEAREST, GL_LINEAR, GL_NEAREST_MIPMAP_NEAREST, GL_NEAREST_MIPMAP_LINEAR, GL_LINEAR_MIPMAP_NEAREST, GL_LINEAR_MIPMAP_LINEAR
GL_TEXTURE_MAG_FILTER	entero	GL_NEAREST, GL_LINEAR
GL_TEXTURE_MIN_LOD	flotante	cualquier valor
GL_TEXTURE_MAX_LOD	flotante	cualquier valor
GL_TEXTURE_BASE_LEVEL	entero	cualquier entero no negativo
GL_TEXTURE_MAX_LEVEL	entero	cualquier entero no negativo
GL_TEXTURE_LOD_BIAS	flotante	cualquier valor
GL_TEXTURE_COMPARE_MODE	enumeración	GL_NONE, GL_COMPARE_R_TO_TEXTURE
GL_TEXTURE_COMPARE_FUNC	enumeración	GL_LEQUAL, GL_GEQUAL, GL_LESS, GL_GREATER, GL_EQUAL, GL_NOTEQUAL, GL_ALWAYS, GL_NEVER

Nota

Hay dos modos de textura que todavía están disponibles en OpenGL, pero que han sido marcados como obsoletos desde OpenGL 3.0, por lo que no se estudiarán a detalle. Estos modos son GL_CLAMP y GL_CLAMP_TO_BORDER. Ambos modos son muy similares a GL_CLAMP_TO_EDGE y sólo difieren en la forma en que se muestrean los texeles en el borde de la textura.

Modo de envoltura GL_REPEAT

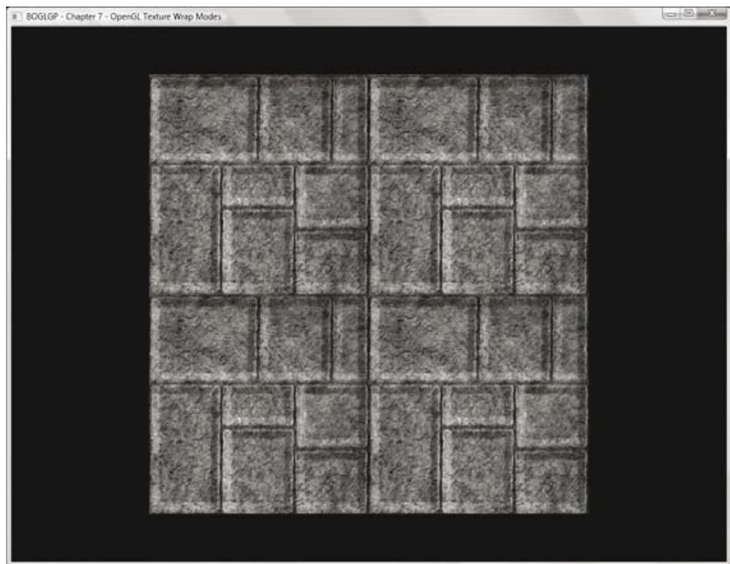
El modo de envoltura predeterminado es GL_REPEAT. En este modo, las texturas se apilan si la coordenada va fuera del rango [0, 1]. Por ejemplo, si usted especifica las coordenadas de textura (2.0, 2.0), entonces la textura se repetirá dos veces, tanto en la dirección s como en la dirección t. En la figura 7.4 se muestra el efecto de GL_REPEAT.

Aunque el modo de envoltura predeterminado es GL_REPEAT, puede revertirlo y especificar otro modo mediante el uso de los siguientes comandos:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

Esto restablecerá el modo de envoltura tanto en la dirección `s` como en la dirección `t`.

Figura 7.4
Modo de envoltura `GL_REPEAT`.



Modo de envoltura GL_CLAMP_TO_EDGE

El modo de envoltura `GL_CLAMP_TO_EDGE` funciona mediante la restricción de las coordenadas de textura en el rango de 0.0 a 1.0. Si usted especifica coordenadas de textura fuera de este rango, OpenGL tomará el borde de la textura y lo extenderá por el resto de la superficie. En la figura 7.5 se muestra a `GL_CLAMP_TO_EDGE` en acción.

Para establecer el modo de envoltura `GL_CLAMP_TO_EDGE` tanto en la dirección `s` como en la dirección `t`, se utiliza el siguiente código:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

Modo de envoltura GL_MIRRORED_REPEAT

El último modo de envoltura que se analizará es `GL_MIRRORED_REPEAT`. Este modo es similar a `GL_REPEAT`, pero en lugar de repetir la misma textura una y otra vez, la imagen se refleja en varias ocasiones a lo largo de las direcciones `s` y `t`. Lo anterior se muestra en la figura 7.6.

Las siguientes líneas de código establecen el modo de envoltura GL_MIRRORED_REPEAT:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_MIRRORED_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_MIRRORED_REPEAT);
```

Figura 7.5
Modo de envoltura GL_CLAMP_TO_EDGE.

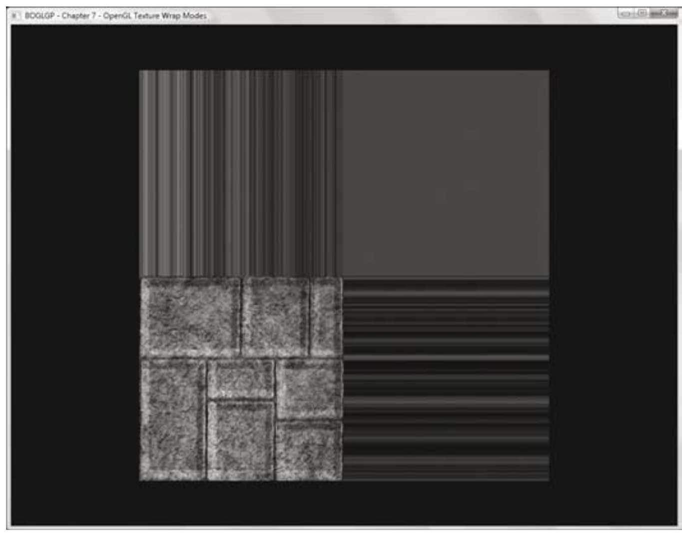


Figura 7.6
Modo de envoltura GL_MIRRORED_REPEAT.

Mipmaps

Los mipmaps son una serie de versiones precalculadas de una textura, cada una de las cuales tiene la mitad del tamaño de la anterior. Por ejemplo, si la imagen de textura original tiene dimensiones de 64 x 64, entonces la serie de imágenes a diferentes niveles de mipmap se generaría con dimensiones de 32 x 32, 16 x 16, 8 x 8, 4 x 4, 2 x 2 y, por último, 1 x 1, lo que resulta en siete niveles de mipmap.

Los mipmaps ayudan a contrarrestar un artificio visual que se denomina *swimming*. Éste se produce cuando dos pixeles adyacentes muestran la misma textura, pero desde texeles bastante separados. Esto tiende a ocurrir cuando la superficie texturizada está muy lejos de la ventana de visualización. Cuando el visor se mueve, las porciones de la textura muestreada cambian, dando lugar a la aparición de diferentes colores. Los mipmaps reducen este problema porque los niveles con resoluciones más bajas se utilizan para polígonos distantes, dando lugar a un muestreo más consistente. Los mipmaps tienen el beneficio adicional de reducir la memoria caché usada en texturas, ya que los niveles más pequeños tienen más probabilidad de permanecer en la memoria de video de alta velocidad por el tiempo que sea necesario. En la figura 7.7 se muestra una serie de mipmaps generados a partir de una imagen base.

OpenGL realiza los procesos con mipmaps determinando qué imagen de textura debe utilizar de acuerdo con la relación existente entre el tamaño del fragmento y el tamaño de los texeles que se aplican a éste. OpenGL elige el nivel de mipmap que permita acercarse tanto como sea posible a un mapeo uno a uno. Cada nivel se define empleando las funciones `glTexImage*()`. El parámetro level de estas funciones especifica el nivel de detalle, o resolución, de la imagen especificada.

De forma predeterminada, es necesario especificar todos los niveles desde 0 hasta donde la textura se reduce a una dimensión 1 x 1 (que es el equivalente a $\log_2$ de la dimensión máxima de la textura base). Estos límites pueden cambiarse mediante la función `glTexParameter()` y al especificar `pname` como `GL_TEXTURE_BASE_LEVEL` o `GL_TEXTURE_MAX_LEVEL`, respectivamente.

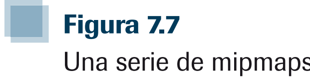
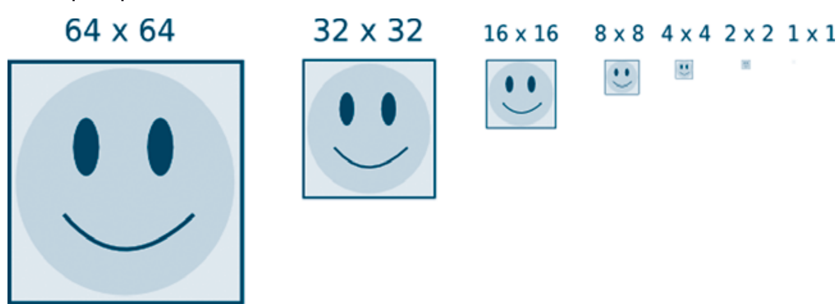


Figura 7.7
Una serie de mipmaps.



El uso de mipmapping se habilita por primera vez mediante la especificación de uno de los valores de mipmapping para el filtrado de texturas de minificación. Después, deberán especificarse

los niveles de mipmaps de textura utilizando las funciones glTexImage*(). El siguiente código crea un mipmap de siete niveles con un filtro de minificación GL_NEAREST_MIPMAP_LINEAR, comenzando con una imagen base de 64 x 64:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST_MIPMAP_LINEAR);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 64,64,0, GL_RGB, GL_UNSIGNED_BYTE,
texImage0);
glTexImage2D(GL_TEXTURE_2D, 1, GL_RGB, 32,32,0, GL_RGB, GL_UNSIGNED_BYTE,
texImage1);
glTexImage2D(GL_TEXTURE_2D, 2, GL_RGB, 16,16,0, GL_RGB, GL_UNSIGNED_BYTE,
texImage2);
glTexImage2D(GL_TEXTURE_2D, 3, GL_RGB, 8, 8, 0, GL_RGB, GL_UNSIGNED_BYTE,
texImage3);
glTexImage2D(GL_TEXTURE_2D, 4, GL_RGB, 4, 4, 0, GL_RGB, GL_UNSIGNED_BYTE,
texImage4);
glTexImage2D(GL_TEXTURE_2D, 5, GL_RGB, 2, 2, 0, GL_RGB, GL_UNSIGNED_BYTE,
texImage5);
glTexImage2D(GL_TEXTURE_2D, 6, GL_RGB, 1, 1, 0, GL_RGB, GL_UNSIGNED_BYTE,
texImage6);
```

Mipmaps y la biblioteca de utilidades de OpenGL

La biblioteca GLU permite que las funciones gluBuild1DMipmaps() y gluBuild2DMipmaps() construyan mipmaps de manera automática para texturas en una y dos dimensiones, respectivamente. Estas funciones sustituyen a la serie de llamadas que normalmente se hacen a las funciones glTexImage2D() y glTexImage1D() para especificar mipmaps.

```
int gluBuild2DMipmaps(GLenum target, GLint components, GLint width, GLint
height, GLenum format, GLenum type, const void *data);
int gluBuild1DMipmaps(GLenum target, GLint components GLint width, GLenum
format, GLenum type, const void *data);
```

El siguiente código utiliza la función gluBuild2DMipmaps() para especificar mipmaps de la misma manera que se hizo en el ejemplo anterior empleando glTexImage2D():

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST_MIPMAP_LINEAR);
gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGB, 64, 64, GL_RGB, GL_UNSIGNED_BYTE,
texImage0);
```

Carga de archivos de imagen Targa

Ahora que se han cubierto los aspectos básicos del uso de texturas con OpenGL, ¡es hora de saber cómo cargarlas en su aplicación! Aquí se analizará el formato de imagen Targa, que es muy flexible y adecuado para las texturas utilizadas en juegos.

El formato de archivo Targa

El formato Targa se divide en dos partes: el encabezado, que almacena información sobre el resto del archivo, y los datos. El encabezado consiste en una serie de campos, que se enlistan en la estructura siguiente:

```
struct TargaHeader
{
  unsigned char idLength;
  unsigned char colorMapType;
  unsigned char imageTypeCode;
  unsigned char colorMapSpec[5];
  unsigned short xOrigin;
  unsigned short yOrigin;
  unsigned short width;
  unsigned short height;
  unsigned char bpp;
  unsigned char imageDesc;
};
```

El encabezado proporciona información importante necesaria para cargar el resto del archivo; los campos `idLength`, `imageTypeCode`, `width`, `height`, `bpp` e `imageDesc` tienen una importancia especial.

`idLength` contiene la longitud en bytes de una cadena de identificación que se encuentra más adelante en el archivo. Quizás desee saltarse la cadena de identificación en vez de cargarla; en ese caso, el campo `idLength` indica la cantidad de datos que deben saltarse. `imageTypeCode` almacena un valor que indica el tipo de imagen, éste puede ser cualquiera de los valores de la tabla 7.6.

`width` y `height` son las dimensiones de la imagen en texeles. `bpp` es la profundidad del color de la imagen, de manera más concreta, indica los bits necesarios para almacenar cada `texel.` `ImageDesc` es un byte de datos cuyos bits almacenan información sobre los pixeles. Los cuatro bits menos significativos en el byte almacenan el número de bits por pixel destinados al canal alfa. Los dos que nos interesan son los bits 4 y 5, que almacenan la esquina de la imagen donde inician los datos de pixel. Algunos archivos Targa almacenan los datos de imagen en forma inversa (los datos inician desde la parte inferior de la imagen); estos bits nos dicen si hay necesidad de voltear la imagen después de haberla cargado. Los valores posibles para estos bits se muestran en la tabla 7.7.

Tabla 7.6 Códigos del tipo de imagen

Código	Descripción
0	No se incluyen datos de imagen
1	Imagen con color asignado sin comprimir
2	Imagen RGB sin comprimir
3	Imagen en blanco y negro sin comprimir
9	Imagen con color asignado comprimida (RLE)
10	Imagen RGB comprimida (RLE)
11	Imagen en blanco y negro comprimida

Tabla 7.7 Origen de las imágenes Targa

Primera posición de pixel	Bit 5	Bit 4	Valor hexadecimal
Inferior izquierda	0	0	0x00
Inferior derecha	0	1	0x10
Superior izquierda	1	0	0x20
Superior derecha	1	1	0x30

Después del encabezado sigue la cadena del identificador que se mencionó anteriormente. Es posible saltarse dicha cadena si se usa el valor `idLength`. Después de esto se encuentran los datos de pixel de la imagen. Estos datos se almacenan en un formato de raíz para un Targa sin comprimir o en un formato RLE comprimido para la versión comprimida. Aquí no se analizará el algoritmo de descompresión; si usted desea aprender más sobre esto, puede consultar el código fuente en el CD o buscar la especificación de la imagen Targa en internet, donde se describe la compresión con detalle.

La clase TargaImage

Dentro de las carpetas de recursos para las dos aplicaciones de ejemplo de este capítulo que se encuentran en el CD, puede encontrarse un encabezado y un archivo fuente denominados targa.h y targa.cpp, respectivamente.

Esta clase cargará una imagen Targa y almacenará los datos listos para su uso en OpenGL. Veamos la definición de la clase:

```
class TargaImage
{
public:
    TargaImage();
    virtual ~TargaImage();
    //funciones de carga y descarga
    bool load(const string& filename);
    void unload();

    unsigned int getWidth() const;
    unsigned int getHeight() const;
    unsigned int getBitsPerPixel() const;
    const unsigned char* getImageData() const;

private:
    TargaHeader m_header;
    unsigned int m_width;
    unsigned int m_height;
    unsigned int m_bitsPerPixel;
    unsigned int m_bytesPerPixel;

    vector<unsigned char> m_imageData;

    //Load() llama a una de estas funciones, dependiendo de su tipo
    bool loadUncompressedTarga(istream& fileIn);
    bool loadCompressedTarga(istream& fileIn);

    bool isImageTypeSupported(const TargaHeader& header);
    bool isCompressedTarga(const TargaHeader& header);
    bool isUncompressedTarga(const TargaHeader& header);

    void flipImageVertically();
};
```

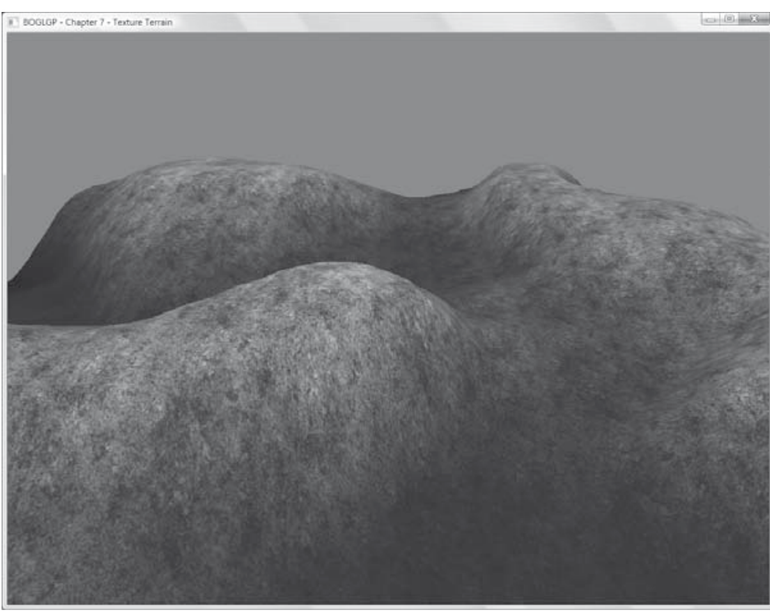
La clase `TargaImage` está diseñada para que se pueda reutilizar con facilidad. No llama a ninguna función de OpenGL en forma automática, sino que permite el acceso a los datos internos para que el usuario pueda llamar a estas funciones cuando así lo desee. A continuación se verá un ejemplo del uso de esta clase:

```
//Declara la instancia TargaImage
TargaImage texture;
```

```
//Y asigna espacio para el nombre de textura generado
GLuint texID;

if (!texture.load("data/rock.tga"))
{
    std::cerr << "Could not load the texture" << std::endl;
    return false;
}

glGenTextures(1, &texID);
glBindTexture(GL_TEXTURE_2D, texID);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexImage2D(GL_TEXTURE_2D,0, GL_RGB8, texture.getWidth(),
    texture.getHeight(), 0, GL_RGB, GL_UNSIGNED_BYTE,
    texture.getImageData());
```

En la figura 7.8 se muestra la aplicación de ejemplo para este capítulo, una versión texturizada del ejemplo del terreno que se ha presentado en capítulos anteriores.

Figura 7.8
Ejemplo del terreno con texturas.



Resumen

En este capítulo, usted aprendió acerca de los objetos de textura y la forma en que OpenGL los utiliza para almacenar el estado de una textura. También conoció la manera de vincular texturas y utilizarlas en GLSL al enlazar samplers uniformes. Además aprendió a enviar coordenadas de textura como un atributo de vértices y a emplearlas en el fragment shader para buscar el texel correspondiente al fragmento. En este momento usted es capaz de entender las razones por las que se usan mipmaps y cómo controlar la forma en que OpenGL realiza los procesos con mipmaps a fin de modificar la calidad de la salida. Por último, aprendió sobre el formato de imagen Targa y cómo utilizar la clase `TargaImage` para cargar archivos de imagen Targa en sus aplicaciones.

Lo que se aprendió

- El mapeo de texturas le permite adjuntar una imagen a una superficie para crear objetos con aspecto realista.
- Las coordenadas de textura se utilizan para asignar la textura al objeto.
- OpenGL incluye texturas de una, dos y tres dimensiones.
- La textura es la información almacenada en los *objetos de textura*, los cuales pueden administrarse mediante handles o manejadores denominados *nombres de textura*.
- Los modos de filtrado alteran la manera en que las texturas se muestrean para su exposición.
- Los datos de textura se especifican mediante los comandos `glTexImage*()`.
- Un mipmap es una textura que consiste en diferentes versiones para distintas resoluciones. Estos niveles se utilizan con el fin de mejorar la calidad del muestreo de texturas.

Preguntas de repaso

1. ¿Qué es un objeto de textura?
2. Si el nivel de textura básico es 128 x 128, ¿cuáles son las dimensiones de los mipmaps?
3. ¿Cuál es el valor predeterminado para el modo de filtrado de texturas en OpenGL?

Por su cuenta

1. Dado un puntero hacia los datos de imágenes en 2D, `ImageData`, cuyas dimensiones son 256 x 256 y cuyo tipo es RGBA, escriba el código para crear un objeto de textura y especifique la textura en 2D con mipmaps.

PARTE 2

Más allá de lo básico

Iluminación, mezclado y niebla

Hasta ahora se ha estudiado lo suficiente en OpenGL como para generar una escena bastante realista mediante el mapeo de texturas. En este capítulo se estudiarán tres temas que agregan realismo a la escena y de las que ningún juego debería prescindir: *iluminación*, *mezclado* (o *blending*) y niebla.

En este capítulo, usted aprenderá sobre:

- Normales de superficie
- Iluminación mediante GLSL
- Fuentes de luz y materiales
- Mezclado (o blending) y transparencias
- Niebla

Iluminación

Después del texturizado, probablemente la iluminación es el aspecto más importante en la creación de una escena realista. Si mira a su alrededor en este momento, verá luz en todas partes (¡a menos que esté leyendo esto en la oscuridad!). Algunas áreas serán más brillantes que otras, dependiendo de qué tan cerca de la luz esté cada superficie, y por supuesto en función de lo brillante que sea la fuente de luz. Lo anterior es importante para simular superficies en las aplicaciones de OpenGL. En los gráficos por computadora, la iluminación suele tratarse de manera independiente al sombreado; la iluminación, simplemente determina qué tan brillante debe ser

cualquier fragmento sin importar si existe algún objeto entre el fragmento y la fuente de luz.

Antes de comenzar, es necesario examinar primero cómo se comporta la luz en el mundo real. Las fuentes de luz producen fotones con diferentes longitudes de onda que cubren todo el espectro de colores. Estos fotones golpean las superficies y algunos de ellos se absorberán mientras que otros serán reflejados, dependiendo de las propiedades reflexivas de la superficie. Una superficie del tipo de un espejo reflejará los fotones de una manera bastante uniforme, mientras que una superficie rugosa los esparcirá. En algún momento, los fotones reflejados ingresan a nuestros ojos (tal vez después de haber sido reflejados por otras superficies), que es como podemos ver el objeto. El color que observamos depende de cuáles longitudes de onda de luz se absorben y cuáles se reflejan.

Modelar todo este proceso de manera exacta en gráficos por computadora no es imposible, pero requiere mucha potencia de procesamiento. En la actualidad, el uso de un modelo de iluminación totalmente exacto es bastante caro para los juegos, ya que éstos necesitarían realizar los cálculos en tiempo real. En vez de proporcionar un modelo perfectamente exacto, es común el empleo de un modelo de iluminación simplificado que es más rápido pero menos preciso. Hay una amplia selección de modelos de iluminación entre los cuales se puede elegir, cada uno con sus propias fortalezas y debilidades.

Nota

OpenGL tiene su propias funciones de iluminación integradas con base en el modelo de iluminación Blinn-Phong. Sin embargo, estas funciones de la API se consideran obsoletas porque forman parte del diagrama de función fija. Aquí se producirá una salida similar a la del modelo de iluminación de función fija, pero en su lugar de emplearán shaders de GLSL.

Los cálculos de iluminación descomponen la luz en cuatro tipos diferentes, que se combinan para generar la salida del color para un fragmento. Estos términos son los siguientes:

Luz ambiental —La luz ambiental simula la luz reflejada tantas veces en las superficies que la fuente de la luz ya no es evidente. La luz ambiental no se ve afectada por las posiciones de la luz o la posición del espectador.

Luz difusa —Esta luz proviene de una dirección determinada, pero cuando choca con la superficie, se refleja igualmente en todas direcciones. Este tipo de iluminación se ve afectada por la posición de la fuente de luz, pero no por la posición del espectador.

Luz especular —Esta luz proviene de una dirección determinada y se refleja en la superficie en una dirección particular. La luz especular se ve afectada por la posición de la fuente de luz y por la posición del espectador.

Luz de emisión —Ésta es la luz que emite un objeto. No es fácil representar el efecto de esta luz sobre otros objetos (un modelo de iluminación muy realista debería tenerlo en cuenta) por lo que normalmente el objeto sólo se representa con una iluminación más intensa.

El resultado final de la iluminación de un objeto depende de varios factores:

El número de fuentes de luz —Cada luz tiene su propia contribución ambiental, difusa, especular y de emisión. Esto debe tenerse en cuenta siempre que una superficie se vea afectada por una luz.

La orientación de la superficie —Si la superficie está orientada en dirección opuesta a la luz, no tiene mucho sentido iluminarla. El cálculo de la iluminación está afectado por el ángulo con el que la luz golpea a la superficie.

El material del objeto —El usuario puede proporcionar una serie de propiedades materiales a una superficie, las cuales determinan la cantidad de luz que debe ser reflejada y qué tan brillante es la superficie.

El modelo de iluminación —Los distintos modelos de iluminación producen salidas diferentes.

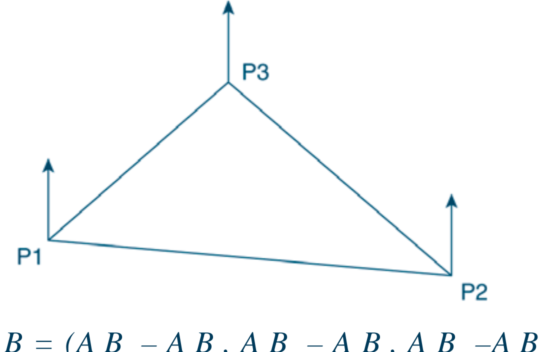
Normales

Antes de empezar a analizar la forma en que se aplica la iluminación, es necesario cubrir algunos requisitos previos para calcularla. Una normal es un vector tridimensional que representa la dirección hacia la que apunta una superficie. Una cosa importante a observar sobre una normal es que es un vector con *longitud unitaria*, es decir que la longitud del vector es 1. En la lista anterior, se mencionó que la orientación de la superficie es importante para la iluminación. La normal de la superficie es la forma en que se especifica la dirección de dicha superficie para el cálculo de la iluminación. Por lo general, las normales se especifican vértice por vértice (como un atributo de vértice), pero cuando se utilizan técnicas de iluminación más precisas (por ejemplo, bump mapping), es posible especificar una normal por cada fragmento. En la figura 8.1 se muestran las normales de un triángulo, los tres vértices tienen normales idénticas para que coincidan con la normal de la superficie.

Cálculo de normales

Calcular la normal de una superficie es bastante sencillo si se emplean matemáticas vectoriales básicas, en particular si se usa el producto cruz. Dados dos vectores, A y B, el producto cruz devolverá el vector perpendicular a ambos. La ecuación para calcular el producto cruz es la siguiente:

Figura 8.1
Un triángulo con sus normales asociadas por vértice.



$$A \times B = (A_y B_z - A_z B_y, A_z B_x - A_x B_z, A_x B_y - A_y B_x)$$

Así, para calcular la normal de una superficie, se necesitan dos vectores de entrada. Estos vectores son dos bordes del polígono. Dados los puntos de un triángulo, P1, P2 y P3, es posible calcular estos vectores empleando lo siguiente:

```
A = P2 – P1
B = P3 – P1
```

La normal puede determinarse mediante el cálculo del producto cruz. El orden en que los vectores se introducen en la ecuación es importante. Si A y B se intercambian, entonces la normal calculada apuntará en la dirección opuesta. El encadenamiento de los vértices también puede afectar esto. En el ejemplo anterior se supone que el triángulo está usando un encadenamiento en sentido antihorario. En la figura 8.2 se muestra cómo se determinan los dos vectores de entrada usando los tres vértices.

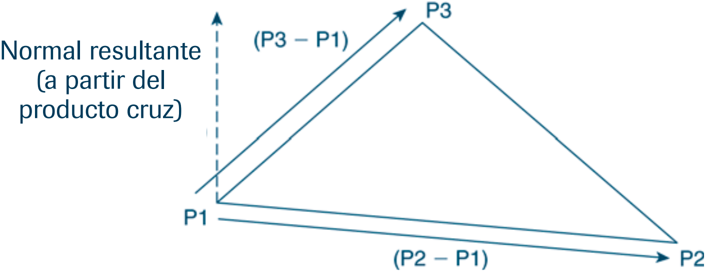
Nota

Una de las razones principales por las que deben usarse triángulos para el render es que los vértices que forman esta figura siempre comparten el mismo plano. Si se hace el render de una superficie con cuatro o más vértices, no es posible garantizar que esto se cumpla. Dado que la normal de la superficie se aplicará a todos los vértices, si éstos no se encuentran en el mismo plano, la iluminación no funcionará correctamente.

El método utilizado para calcular normales de vértice varía ligeramente dependiendo del tipo de objeto del que se pretende hacer el render. Si se trata de un polígono único, como un triángulo o un objeto de múltiples caras con lados planos y esquinas en ángulo, entonces la determinación de una normal para cada cara (con todos los vértices compartiendo la normal) funciona de

forma adecuada. Sin embargo, si se hace el render de un objeto liso como una esfera (en la que dos superficies pueden compartir el mismo vértice), entonces la normal del vértice debe ser el promedio de las normales de las caras que comparten dicho vértice. Esto da por resultado una superficie lisa. Cuando se desea aplicar a conjuntos complejos que incluyen bordes duros y curvas suaves, la iluminación se vuelve más difícil. Hay soluciones para este problema, como los *grupos de suavizado* (smoothing groups) (los cuales identifican los triángulos que pueden usarse junto al promedio de las normales).

Figura 8.2
Localización de los dos vectores de entrada para el producto cruz.



Asignación de la longitud unitaria a la normal

Aunque ahora ya podemos calcular el vector perpendicular a una superficie, no se puede usar de manera confiable en cálculos de iluminación hasta que se haya normalizado. La normalización es el proceso de acortamiento o alargamiento de un vector para darle una longitud de 1. El primer paso en la normalización es encontrar la longitud actual del vector; esto se puede lograr al determinar la raíz cuadrada de la suma de las componentes individuales al cuadrado, que no es tan complicado como suena:

$$|A| = sqrt(A_x^2 + A_y^2 + A_z^2)$$

Una vez que se tiene la longitud del vector, éste se normaliza dividiendo cada uno de sus componentes entre el valor de la longitud.

El modelo de iluminación

Los cálculos de iluminación que se estudiarán aquí están basados en el modelo Blinn-Phong, el cual a su vez se basa en el modelo Phong que es un poco más costoso desde el punto de vista computacional. La iluminación se calculará para cada vértice y el color resultante se interpolará sobre la superficie de un polígono. Este método de iluminación prefiere la velocidad sobre la precisión y la calidad, pero el concepto puede extenderse hasta el nivel de los pixeles para un mayor realismo.

Materiales

Las propiedades de una superficie tienen un efecto en el cálculo de la iluminación. Por ejemplo, una superficie metálica de plata aparecerá brillante, mientras que un muro de ladrillos no. Es necesario especificar estas propiedades materiales para obtener un contraste realista entre las diferentes superficies.

El cálculo de la iluminación usa las propiedades del material para simular la forma en que una superficie refleja la luz roja, verde y azul. Por ejemplo, si usted tiene un objeto rojo puro, éste reflejará toda la luz roja pero absorberá toda la luz verde y azul. Si hace brillar una luz azul pura en el objeto, aparecerá negro (porque no hay luz roja que reflejar y la luz azul se absorbe). Si coloca el objeto bajo una luz blanca, éste aparecerá de color rojo —reflejaría la luz roja, pero absorbería la verde y la azul.

Existen cinco diferentes propiedades de los materiales que es necesario especificar para la ecuación de iluminación en el vertex shader:

Color difuso —El color y la intensidad que refleja la luz difusa.
Color ambiental—El color y la intensidad que refleja la luz ambiental.
Color especular —El color y la intensidad que refleja la luz especular.
Color de emisión —Color de luz que emite ese material.
Brillantez —Tamaño del reflejo especular.

La diferencia entre la luz difusa y especular se puede entender como la diferencia entre las pinturas mate y brillante. Imagine una pared pintada con pintura mate, ésta refleja la luz uniformemente en todas direcciones; la intensidad aparente de la luz reflejada no se modifica aunque usted cambie su punto de visión. En los gráficos por computadora, una superficie mate tendría una alta proporción de luz difusa. Si pintara la misma pared con pintura brillante, vería mayores intensidades de luz en función del ángulo entre su posición y la pared, y de acuerdo con el ángulo entre la pared y la luz. Esta superficie puede simularse con una alta proporción de luz especular.

La propiedad material ambiental define qué tan bien se refleja la luz ambiental en la superficie. Un valor ambiental alto significa que la superficie refleja bien la luz ambiental del entorno. La propiedad material final, la brillantez, funciona con el valor especular. Cuanto mayor sea el valor de la brillantez, mayor será el reflejo especular.

Atenuación

Para ser realistas, las luces deben iluminar los objetos con menos intensidad cuando éstos se encuentran lejos de la fuente luminosa. En algún momento, cuando el objeto está lo suficientemente lejos, la luz debe dejar de iluminarlo por completo. A esta disminución dependiente de la distancia se le conoce como *atenuación*. Hay ocasiones en las que no se desea que alguna luz sea atenuada; por ejemplo, la luz del Sol está tan lejos y representa una fuente luminosa tan fuerte, que debe representarse sin atenuación. No obstante, la mayoría de las luces en la escena necesitarán un factor de atenuación de la iluminación para dar una impresión realista.

La atenuación se calcula utilizando tres valores modificables: los factores de atenuación constante, lineal y cuadrático. Los tres parámetros se utilizan para calcular un valor de atenuación de punto flotante en cada vértice. Los colores específicos de luz difusa, especular, y ambiental se multiplican por este factor para reducir la intensidad de la luz dependiendo de la distancia de la fuente luminosa. El factor de atenuación se calcula de la siguiente manera:

$$\frac{1}{k_c + k_1 d + k_q d^2}$$

K_c, K_1, y K_q son la atenuación constante, lineal y cuadrática, respectivamente. `D` es la distancia desde la fuente de luz. Los valores globales de la luz ambiental (vea la sección "Contribución ambiental") y la de emisión no se ven afectados por la atenuación.

Reflexión lambertiana

Ahora es el momento de abordar los cálculos de iluminación. Éstos se dividirán en sus partes componentes antes de reunirlos en los vertex shaders, comenzando con la luz difusa.

Como ya se mencionó, la iluminación difusa no toma en cuenta la posición del espectador, sino sólo la dirección de la luz y la normal a la superficie. Si una superficie está de espaldas a una fuente de luz, entonces no recibirá ningún tipo de luz difusa, por lo que la intensidad de la luz difusa sobre una superficie depende del ángulo de la superficie respecto a la fuente de luz. Este efecto se conoce como reflexión lambertiana. La intensidad de la luz difusa puede calcularse mediante la siguiente ecuación:

$$\max(L \bullet N,\ 0.0) \times C \times I$$

`L` es la dirección de la luz, `N` es la normal a la superficie, `C` es el color difuso del material e `I` es la propiedad difusa de la luz; la multiplicación de dos vectores (como los mencionados `L` y `N`) por lo general significa realizar el `producto punto`. La multiplicación de colores implica el cálculo de un producto de sus componentes.

Nota

El *producto punto* es un cálculo simple que puede realizarse sobre dos vectores. Siempre que los vectores tengan una longitud unitaria, el resultado del producto punto es el coseno del ángulo entre los dos vectores de entrada. El producto punto se calcula como:

$$A \bullet B = A_x B_x + A_y B_y + A_z B_z$$

El término especular

El valor especular se calcula utilizando una ecuación de iluminación derivada del modelo de reflexión Blinn-Phong. Para determinar el color especular, primero debe calcularse lo que se conoce como *vector medio*. El vector medio se encuentra a mitad del camino entre el vector de visualización y el vector de luz (vea la figura 8.3); se obtiene al sumar dichos vectores.

$$H = L + E$$

`L` es la dirección desde el vértice hasta la fuente de luz, y `E` es el vector desde el vértice hasta la posición de visualización.

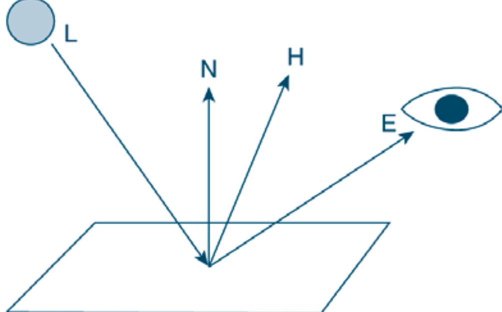
Una vez que se tiene el vector medio, la contribución especular al color final se puede encontrar con:

$$\max(N \bullet H,\ 0)^s \times S_m \times S_l$$

Donde `N` es la normal del vértice, `H` es el vector medio y `S` es el valor de la brillantez del material. S_m y S_l son los colores especulares del material y la luz, respectivamente. Este valor se añade al color final del fragmento (quizá después de multiplicarlo por un factor de atenuación). Posteriormente en este capítulo se verá un ejemplo del valor especular calculado en los shaders de GLSL.

Figura 8.3

Vectores utilizados en la iluminación de Blinn-Phong.



Contribución ambiental

El término ambiental representa la luz que ilumina de manera indirecta una superficie al reflejarse en otras superficies. Se pueden modelar dos tipos de luz ambiental: la luz ambiental específica de la fuente, que se encuentra usando los valores ambientales de una fuente luminosa, y la luz ambiental global, la cual es una constante que afecta a toda la escena. Los términos ambientales simplemente se multiplican por la reflectancia ambiental del material y luego se añaden al color final. La luz ambiental global no experimenta atenuación, por lo que se puede añadir de manera sencilla al final del shader. La luz ambiental específica se ve afectada por los

valores de atenuación de la fuente luminosa y antes de aplicarla es necesario multiplicarla por dichos valores de atenuación. En los ejemplos de este capítulo no se utiliza un valor ambiental global; la aplicación de éste se deja como ejercicio para el lector.

La matriz normal

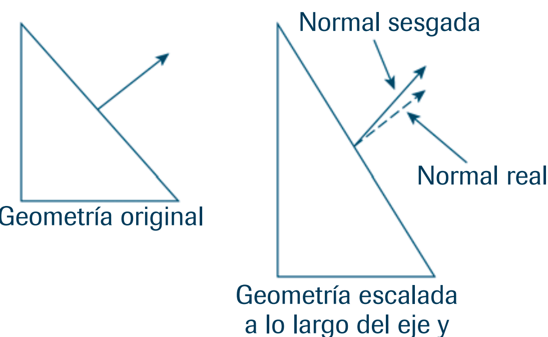
Todos los cálculos de iluminación se realizan en el espacio de visualización. Como ya se vio en el capítulo 4, para transformar un vértice del espacio del objeto al espacio de visualización se emplea la matriz modelview. Por desgracia, las cosas no son tan sencillas cuando se trata de transformar las normales de superficie al espacio de visualización. Las normales requieren una matriz especial de 3 x 3 conocida como la *matriz normal* para poderlas colocar en el espacio de visualización. La matriz normal se basa en la parte de rotación de la matriz modelview. Casi siempre, las siguientes líneas de código son suficientes para transformar un vector normal al espacio de visualización:

```
//a_Normal es el atributo de vértice normal,
//normal contiene la normal transformada final
mat3x3 normalMatrix = mat3x3 (modelview_matrix);
vec3 normal = normalize(normalMatrix*a_Normal);
```

Esta transformación funciona bien si su matriz modelview es *ortogonal*. En una matriz ortogonal, la *transpuesta* de la matriz (la matriz que resulta de intercambiar todas las filas y columnas) es igual a la *inversa* de la matriz (al multiplicar una matriz por su inversa se obtiene la matriz identidad). Si sólo se utilizan rotaciones y traslaciones, entonces la matriz modelview será ortogonal y podrá usarse el código anterior para calcular la matriz normal. Si se emplea cualquier escala no uniforme al hacer el render de la escena, entonces la matriz normal debe calcularse como:

```
N = transpose(inverse(M))
```

Figura 8.4
Normal transformada incorrectamente al usar una matriz modelview no ortogonal.



Donde `N` es la matriz normal resultante, y `M` es la matriz de 3x3 formada por los elementos de la parte superior izquierda de la matriz modelview. Si se usan matrices no ortogonales para la transformación, pueden obtenerse normales sesgadas como la de la figura 8.4.

Nota

Los 3 x 3 elementos de la parte superior izquierda de la matriz modelview pueden usarse si la matriz es ortogonal, ya que la transpuesta e inversa de una matriz ortogonal son iguales. En este caso, la transpuesta e inversa se anulan mutuamente, dando lugar a la matriz original.

El cálculo de la inversa de una matriz es un proceso complicado, que no se cubrirá a detalle aquí. Si desea tener más información sobre cómo se obtiene la inversa de una matriz, puede consultar los ejemplos de código fuente para este capítulo, en donde se comenta un método para realizar dicho cálculo.

Iluminación en GLSL

Ahora que se han cubierto los aspectos básicos de la iluminación, es el momento de poner ese conocimiento en práctica mediante la aplicación de los conceptos en GLSL. En las secciones siguientes aprenderá a iluminar una escena con tres diferentes tipos de fuente de luz.

Iluminación direccional

Comenzaremos con la forma más simple de iluminación: la luz direccional. Ésta proviene de cierta dirección, pero no tiene una verdadera posición de la fuente, sino que proviene de un lugar infinitamente lejano. Las luces direccionales no existen en la vida real, pero algunas fuentes de luz como el Sol están lo suficientemente lejos en la escena como para poder utilizar una luz direccional al ser modeladas. Es más económico calcular luces direccionales que luces posicionales, ya que no experimentan atenuación y el cálculo de la iluminación no se basa en la distancia a la fuente luminosa (que requiere un costoso cálculo de la raíz cuadrada cuando se usa una luz posicional).

Ahora se dará un vistazo a cómo modelar una luz direccional con GLSL. Es necesario calcular los términos de iluminación: difusa, ambiental y especular. El término ambiental es simplemente el valor ambiental del material multiplicado por el valor ambiental de la luz. Como el término ambiental no se ve afectado por la distancia o la dirección, sólo se añade al color final del vértice. En el caso de las contribuciones difusa y especular se emplea la dirección de la luz para calcular su valor. En una luz direccional, la posición de la luz representa su dirección, además siempre se debe establecer el componente de posición final (w) en cero a fin de reflejar que dicha luz direccional no se ve afectada por la traslación (ya que es una dirección y no un vértice). A

continuación se presenta un vertex shader en GLSL, el cual calcula el efecto de una luz direccional. Por razones de brevedad, no se han incluido las variables uniforme y global.

```
void main(void)
{
    //Transforma la normal en el espacio de visualización
    empleando la matriz normal
    vec3 N = normalize(normal_matrix*a_Normal);

    //Calcula la dirección de la luz, en una luz direccional la
    //posición en realidad es un vector de dirección
    //Transformamos la posición en el espacio de visualización
    antes de normalizar el vector
    vec3 L = normalize (modelview_matrix*light0.position).xyz;

    //Calculamos el ángulo entre la normal y la dirección de la luz
    float NdotL = max(dot(N, L), 0.0);

    //Se fija el color ambiental, de modo que se agrega como el
    color inicial y luego se construye sobre éste
    //con las otras contribuciones de iluminación
    vec4 finalColor = material_ambient*light0.ambient;

    //Hace la transformación del vértice estándar en el espacio
    de visualización
    vec4 pos = modelview_matrix*vec4(a_Vertex, 1.0);

    //Como estamos en el espacio de visualización (todo es relativo a la cámara)
    //El vector de visualización es simplemente la posición negada
    vec3 E = -pos.xyz;

    //Si la normal a la superficie está completamente orientada hacia la luz
    if(NdotL> 0.0)

  {
    //Añade el color difuso utilizando Reflexión Lambertiana
    finalColor += material_diffuse*light0.diffuse*NdotL;

    //Calcula el vector medio y lo hace de longitud unitaria
    vec3 HV = normalize(L + E);
```

```
    //Encuentra el ángulo entre la normal y el vector medio
    float NdotHV = max(dot(N, HV), 0.0);

    //Calcula el color especular con Blinn-Phong
    finalColor += material_specular*light0.specular*pow(NdotHV, material_shini-
ness);
  }

  //Salida del color final, las coordenadas de textura y la posición
  color = finalColor;
  texCoord0 = a_TexCoord0;
  gl_Position = projection_matrix*pos;
}
```

El principal cálculo de iluminación se realiza dentro del bloque `NdotL if`. Aquí es donde se calculan los términos difuso y especular. Observe que el término ambiental se calcula cerca del principio, porque éste no se ve afectado por la dirección de la fuente de luz. La figura 8.5 es una captura de pantalla con la aplicación de *iluminación* al ejemplo *del terreno*.

Luces puntuales

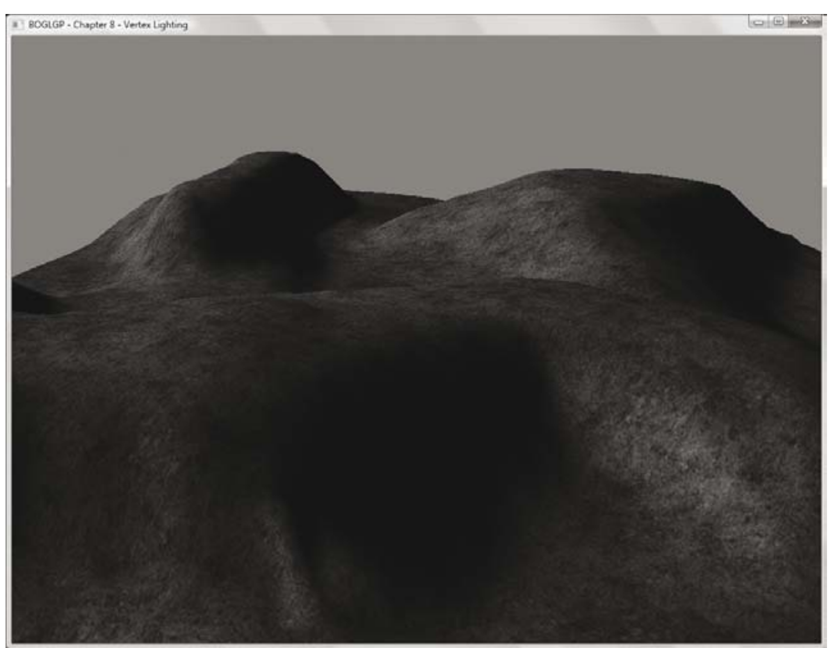
Hay dos diferencias principales entre las luces puntuales y las luces direccionales:

- Las luces puntuales tienen una posición en vez de una dirección.
- Las luces puntuales experimentan atenuación.

Aparte de eso, el algoritmo general es el mismo. Al especificar la posición de una luz puntual, la componente w debe establecerse en 1.0, porque las luces puntuales se ven afectadas por la traslación. A continuación se presenta un shader en GLSL que calcula el efecto de una luz puntual:

```
void main(void)
{
    vec3 N = normalize(normal_matrix*a_Normal);
    vec4 pos = modelview_matrix*vec4(a_Vertex, 1.0);
    //Calcula la posición de la luz en el espacio de visualización
    //al multiplicar por la matriz modelview
    vec3 lightPos = (modelview_matrix*light0.position).xyz;
    //Obtiene el vector de dirección de la luz mediante la
    //localización del vector entre el vértice y la posición de la luz
    vec3 lightDir = (lightPos - pos.xyz).xyz;
```

Figura 8.5
Captura de pantalla del ejemplo de luz direccional.



```
//Encuentra el ángulo entre la normal y la dirección
de la luz
float NdotL = max(dot(N, lightDir.xyz), 0.0);

//La distancia entre el vértice y la luz es la longitud
//del vector direccional de la luz
float dist = length(lightDir);

//Encuentra el vector de visualización (igual que una luz
direccional)
vec3 E = -(pos.xyz);

//Asigna el color ambiental
vec4 finalColor = material_ambient*light0.ambient;

//El cálculo de la iluminación es igual que para la luz direccional
if(NdotL > 0.0)
{
```

```
      vec3 HV = normalize(lightPos + E);
      finalColor += material_diffuse*light0.diffuse*NdotL;
      float NdotHV = max(dot(N, HV), 0.0);
      finalColor += material_specular*light0.specular *
                                       pow (NdotHV, material_shininess);
  }

  //Calcula el factor de atenuación
  float attenuation = 1.0/( light0.constant_attenuation +
                            light0.linear_attenuation*dist + light0.quadra-
tic_attenuation*dist*dist);

  //El valor de emisión del material no está afectado por
  //la atenuación, debido a esto se agrega por separado
  color = material_emissive + (finalColor*attenuation);

  texCoord0 = a_TexCoord0;
  gl_Position = projection_matrix *pos;
}
```

Las únicas diferencias con el shader de luz direccional son los cálculos de la dirección de la luz y la adición de la atenuación. Observe que el valor de emisión del material no experimenta atenuación, ya que la propia superficie "emite" la luz. En la figura 8.6 se muestra este shader en acción.

Focos

Los focos pueden entenderse como una luz puntual especializada. Pero a diferencia de una luz puntual que irradia luz en todas direcciones, su influencia se limita a un cono direccional. Un foco tiene unas cuantas propiedades adicionales que deben tenerse en cuenta. En primer lugar, y quizás de manera más obvia, el foco tiene un vector de dirección que determina el sentido del cono de luz. La segunda propiedad adicional que se requiere es el ángulo de corte. Éste es el ángulo que tiene el cono de influencia del foco. La propiedad adicional final se denomina el *exponente del foco*. Este valor determina la rapidez con la que disminuye la intensidad de luz desde el centro del cono hasta sus paredes. En efecto, esto es una forma de atenuación; así que en el shader de GLSL, el foco forma parte del cálculo de la atenuación.

Para determinar si un vértice está dentro de la influencia de un foco, primero debe encontrarse el ángulo entre la normal a la superficie y el vector dirigido desde el vértice hasta la fuente de luz (igual que en las ecuaciones de iluminación). Si el vértice está de frente a la fuente de luz, entonces éste se encuentra dentro de los límites del cono.

Para determinar si el vértice debe estar iluminado, es necesario tomar el ángulo entre la dirección del foco y el vector dirigido desde la luz hasta el vértice. Si el ángulo es mayor que el coseno

Figura 8.6
Captura de pantalla del ejemplo de luz puntual.

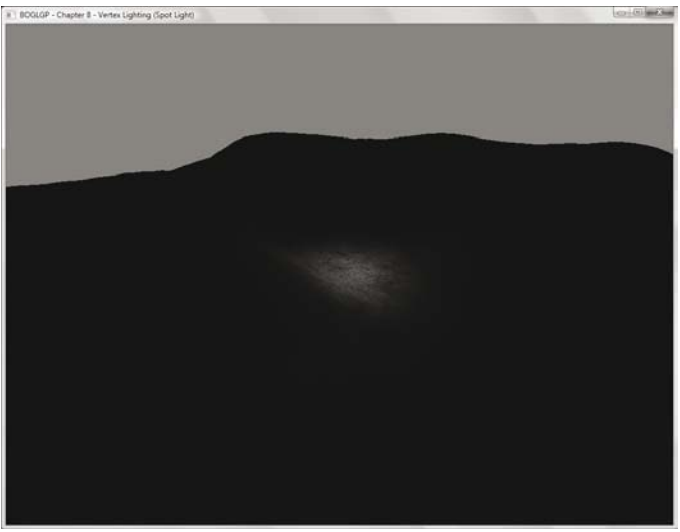
del ángulo de corte del foco, entonces puede estar iluminado. El cálculo de la iluminación es igual que para la luz puntual, sólo que aquí la atenuación toma en cuenta el exponente del foco. A continuación se muestra el vertex shader para un solo foco.

```
void main(void)
{
    vec3 N = normalize(normal_matrix*a_Normal);
    vec3 lightPos = (modelview_matrix*light0.position).xyz;
    vec4 pos = modelview_matrix*vec4(a_Vertex, 1.0);
    vec3 lightDir = (lightPos - pos.xyz).xyz;
    float NdotL = max(dot(N, lightDir.xyz), 0.0);
    float dist = length(lightDir);
    vec3 E = -(pos.xyz);
    vec4 finalColor = material_ambient * light0.ambient;
    float attenuation = 1.0;

    //Si la superficie está de frente a la fuente de luz
    if(NdotL > 0.0)
    {
      //Encuentra el ángulo entre la dirección de la luz y la
      dirección del foco
      float spotEffect = dot(normalize (light0.spotDirection), norma-
      lize (-lightDir));
```

```
        //Si es mayor que el coseno del corte del foco,
        entonces debe iluminarse
        if(spotEffect > cos (light0.spotCutOff))
        {
        vec3 HV = normalize(lightPos + E);
        float NdotHV = max(dot(N, HV), 0.0);
        finalColor += material_specular *light0.specular *  pow(NdotHV,
        material_shininess);

          //Calcula la atenuación empleando el exponente del foco
          spotEffect = pow(spotEffect, light0.spotExponent);
          attenuation = spotEffect/
    (light0.constant_attenuation +
    light0.linear_attenuation * dist +
    light0.quadratic_attenuation * dist * dist);

          finalColor += material_diffuse * light0.diffuse
    * NdotL;
         }
      }

      color = material_emissive + (finalColor * attenuation);
      texCoord0 = a_TexCoord0;
      gl_Position = projection_matrix * pos;
}
```

Observe cómo se usa el valor de corte del foco para determinar si un vértice debe estar iluminado. En la figura 8.7 se muestra una captura de pantalla del ejemplo del foco, el cual puede encontrarse en el CD.

Figura 8.7
Ejemplo del foco.



Luces múltiples

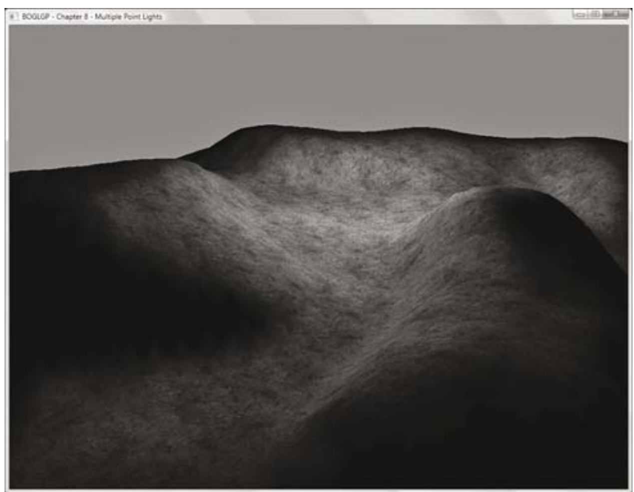
En este capítulo, se ha analizado la iluminación de una superficie con una sola luz. En la mayoría de las escenas habrá varias luces y la mayor parte de las superficies estarán iluminadas por más de una de estas luces a la vez. Hay un par de formas de agregar varias luces en las aplicaciones de OpenGL. Es posible hacer un render de la escena varias veces, una vez para cada luz. Cada pasada de render combinará la escena usando la mezcla aditiva. Este método es simple, pero resulta lento para la representación de escenas grandes y complejas. Otro método consiste en escribir un shader de GLSL que recorra cierta cantidad de fuentes luminosas pasadas a través de variables uniformes.

En el CD puede encontrarse un ejemplo llamado *multiple lights* (luces múltiples), donde el shader de GLSL usa las propiedades de dos luces puntuales (pasadas como uniformes) para iluminar la escena. El shader puede ampliarse con facilidad para incluir más luces alterando el tamaño del arreglo `lights` y pasando los datos de las propiedades (difusa, especular, etcétera) de las nuevas luces.

Mejora de la calidad

Los cálculos de iluminación cubiertos en este capítulo se han aplicado dentro del vertex shader. La iluminación de vértices es económica, y en las superficies ordenadas a manera de mosaico puede proporcionar una calidad de iluminación bastante aceptable. Sin embargo, muchas de las superficies dibujadas en una escena 3D no tienen forma de mosaico y la iluminación resultante puede no ser muy buena, en especial para el caso de los focos. La calidad de la iluminación

Figura 8.8
Ejemplo de las luces puntuales múltiples.



puede mejorar dramáticamente si los cálculos de iluminación se trasladan al fragment shader. Para ello, basta con calcular cualquier elemento que no dependa de la distancia o la dirección (por ejemplo, la contribución ambiental o el vector medio) en el vertex shader y luego transmitir estos valores al fragment shader donde se lleva a cabo el cálculo de la iluminación. Recuerde normalizar de nuevo los vectores que deben tener una longitud unitaria en el fragment shader. Esto se debe a que un vector no necesariamente mantendrá la longitud unitaria cuando es interpolado para las operaciones por fragmento.

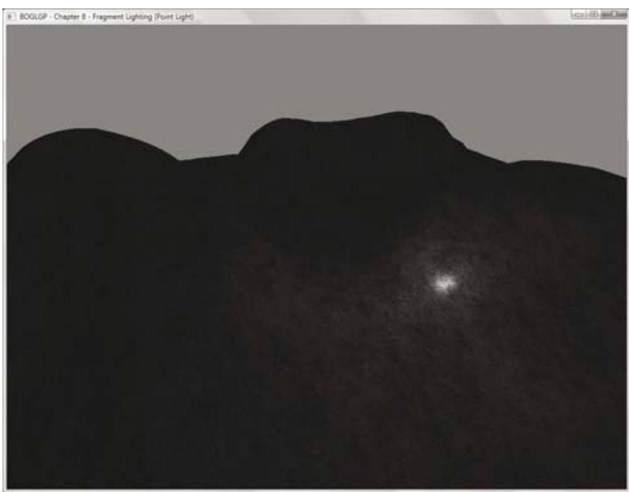
Dentro del CD puede encontrarse una versión por pixel del ejemplo de la luz puntual. Ahí puede observarse que el vertex shader se ha acortado bastante porque la mayor parte del código se ha trasladado al fragment shader.

Mezclado

El mezclado (o blending) es una característica muy poderosa, representa la clave para muchos de los efectos gráficos que se ven en los juegos de computadora modernos. Éste se produce después de la etapa de procesamiento de fragmentos en el diagrama y se usa para mezclar el color del fragmento de salida con el color del fragmento al que previamente se hizo render en el búfer de fotogramas. Esta técnica se utiliza normalmente para simular superficies translúcidas como el agua, el cristal, etcétera.

Aquí es donde entra en juego el canal alfa, que hasta ahora había sido ignorado. Normalmente, el valor alfa se usa para representar la opacidad del fragmento que se pretende dibujar. Por lo general, el fragmento que se está representando se conoce como el *origen* y el que ya existía en la memoria de video se denomina el *destino*.

Figura 8.9
Ejemplo de la luz puntual realizado por fragmento.



El mezclado en OpenGL se habilita usando `glEnable()` con el parámetro `GL_BLEND`; después de esto permanecerá habilitado hasta que se desactive mediante `glDisable()`.

Mientras el mezclado esté disponible, sus efectos se pueden controlar mediante la función `glBlendFunc`:

```
void glBlendFunc(GLenum sfactor, GLenum dfactor);
```

`glBlendFunc()` especifica los factores de mezclado del origen y el destino, los cuales deben estar entre 0.0 y 1.0. `sfactor` es el factor de mezcla del origen, y `dfactor` es el del destino. Los factores de mezcla se multiplican por los colores del origen y el destino antes de combinarse para crear el color final del fragmento.

En la tabla 8.1 se muestran los valores posibles para `glBlendFunc`:

Tabla 8.1 Factores de mezcla

Factor	Descripción
GL_ZERO	Cada componente se multiplica por cero (es decir, se vuelve negro).

Tabla 8.1 Factores de mezcla (continuación)

Factor	Descripción
GL_ONE	El color no se modifica (multiplicado por 1.0).
GL_SRC_COLOR	Cada componente del color se multiplica por el correspondiente.
GL_ONE_MINUS_SRC_COLOR	Cada componente se multiplica por 1.0 – el componente de color del origen.
GL_DST_COLOR	Cada componente de color se multiplica por el correspondiente componente de color del destino.
GL_ONE_MINUS_SRC_ALPHA	Cada componente de color se multiplica por 1.0 – el componente de color del destino.
GL_SRC_ALPHA	Cada componente de color se multiplica por el valor alfa del origen.
GL_ONE_MINUS_SRC_ALPHA	Cada componente se multiplica por 1.0 – el valor alfa del origen.
GL_DST_ALPHA	Cada componente de color se multiplica por el valor alfa del destino.
GL_ONE_MINUS_DST_ALPHA	Cada componente se multiplica por 1.0 – el valor alfa del destino.
GL_CONSTANT_COLOR	Cada componente de color se multiplica por el componente correspondiente del color constante actual (vea “Color de mezcla constante”).
GL_ONE_MINUS_CONSTANT_COLOR	Cada componente de color se multiplica por 1.0 – el componente de color constante.
GL_CONSTANT_ALPHA	Cada componente de color se multiplica por el componente alfa del color constante actual.
GL_ONE_MINUS_CONSTANT_ALPHA	Cada componente de color se multiplica por 1.0 - el valor alfa del color constante.
GL_SRC_ALPHA_SATURATE	Cada componente de color se multiplica por el menor valor entre el valor alfa del origen o 1.0 – el valor alfa del destino. El valor alfa no se modifica. Esto es válido solamente como un factor del origen.

Los factores de mezcla predeterminados son GL_ONE para el origen y GL_ZERO para el destino, lo que básicamente significa que no existe mezclado porque el color del destino está multiplicado por cero (no hay influencia) y el color de origen mantiene su intensidad.

Los factores de mezcla pueden combinarse de distintas maneras para crear diferentes efectos. El más común de éstos es la traslucidez, que se conoce más comúnmente como transparencia. Por lo general, en los efectos de transparencia se usa GL_SRC_ALPHA como el factor de mezcla

del origen y GL_ONE_MINUS_SRC_ALPHA como el factor de mezcla del destino. Esta combinación permite que la contribución de un fragmento de color esté determinada por su canal alfa. Un valor de alfa elevado implicará una mayor influencia dada al color de origen, lo que resulta en una superficie con aspecto opaco. Un valor de alfa bajo significa que el color de destino tendrá más influencia, haciendo que la superficie aparezca más transparente. El código siguiente sirve para configurar el mezclado para la transparencia:

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

A continuación se verá un ejemplo concreto. Digamos que usted dibuja un triángulo color rojo (1.0, 0.0, 0.0, 1.0) y luego justo enfrente hace el render de un triángulo azul (0.0, 0.0, 1.0, 0.5). Como se está empleando el canal alfa de origen como un factor de combinación, el triángulo azul se representará con el 50% de transparencia. Los colores resultantes antes de la combinación se calcularían de la siguiente manera:

$$
\begin{aligned}
Source.R &= 0.0 \times 0.5 = 0.0 \\
Source.G &= 0.0 \times 0.5 = 0.0 \\
Source.B &= 1.0 \times 0.5 = 0.5 \\
Source.A &= 0.5 \times 0.5 = 0.25
\end{aligned}
$$

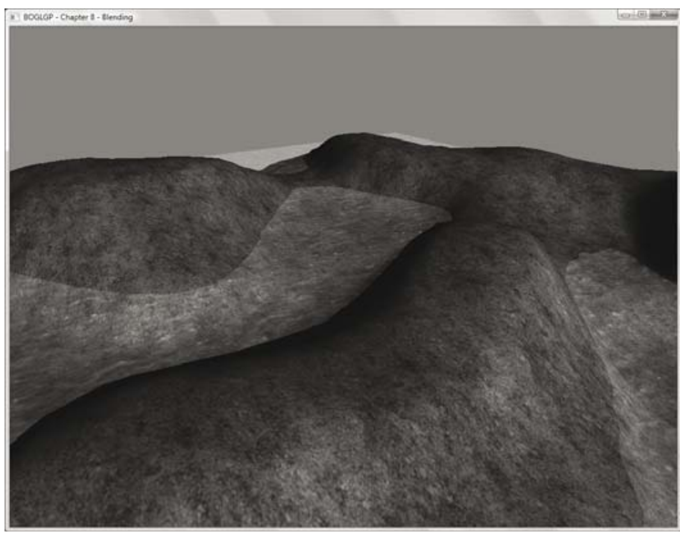
$$
\begin{aligned}
Dest.R &= 1.0 \times 0.5 = 0.5 \\
Dest.G &= 0.0 \times 0.5 = 0.0 \\
Dest.B &= 0.0 \times 0.5 = 0.0 \\
Dest.A &= 1.0 \times 0.5 = 0.5
\end{aligned}
$$

El color final del fragmento se determina mediante la suma de estos dos colores para obtener el color resultante (0.5, 0.0, 0.5, 0.75).

Por desgracia, a medida que las escenas se vuelven más complejas, las cosas no son tan simples. Al hacer el render sin ningún tipo de transparencia, puede representar los polígonos que componen la escena y el búfer z (buffer de profundidad) se encargará de determinar si un fragmento se ve o está oculto detrás de otra superficie. En realidad el orden de render no importa cuando se trabaja con superficies opacas, pero al representar superficies transparentes ese orden se vuelve importante. Al hacer el render de una escena que contiene superficies transparentes, debe representar primero los objetos opacos, y después los objetos transparentes, de acuerdo con la profundidad medida desde la cámara, de atrás hacia adelante. Esto es porque si usted dibuja un polígono transparente y luego hace otro polígono detrás de él, el segundo no pasará la prueba de profundidad y no se representará, por lo que no se verá detrás del polígono transparente.

La aplicación de ejemplo para este capítulo agrega agua translúcida a nuestra aplicación del terreno.

Figura 8.10
Captura de pantalla del ejemplo del mezclado.



Funciones de mezclado separadas

Como se vio en el ejemplo para el cálculo de color en la sección anterior, el canal alfa de un fragmento combinado también se ve afectado por los factores de mezcla que se especifiquen al utilizar `glBlendFunc()`. En el ejemplo, el valor alfa suministrado para el fragmento de origen fue de 0.5, pero después de mezclarlo con el fragmento de destino, el alfa final fue de 0.75. En ocasiones, no se desea que el componente alfa resulte afectado por los mismos factores de mezcla que los componentes de color rojo, verde y azul. Por fortuna, OpenGL proporciona una función que es similar a `glBlendFunc()` pero que permite que los factores para el canal alfa del origen y el destino se especifiquen por separado. Esta función se llama `glBlendFuncSeparate()`:

```
void glBlendFuncSeparate(GLenum sfactorRGB, GLenum dfactorRGB, GLenum
sfactor-Alpha, GLenum dfactorAlpha);
```

`sfactorRGB` y `dfactorRGB` especifican los factores del origen y el destino para los componentes de color rojo, verde y azul. `sfactorAlpha` y `dfactorAlpha` especifican los factores de mezcla para el componente alfa.

Color constante de mezclado

Algunos de los factores de mezclado enumerados en la tabla 8.1 se refieren a un color de mezcla constante. Éste se usa normalmente para mezclar imágenes RGB que no especifican un valor alfa. El color de mezcla constante puede especificarse mediante la función `glBlendColor()`:

```
void glBlendColor(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha)
```

Los parámetros se explican por sí mismos; definen los componentes rojo, verde, azul y alfa del color constante. El tipo `GLclampf` indica un valor de punto flotante entre 0.0 y 1.0. El color de mezcla predeterminado es (0, 0, 0, 0). El valor actual del color de mezcla se puede recuperar al pasar `GL_BLEND_COLOR a glGet()`.

Niebla

La niebla es un efecto que se añade con facilidad a una escena. No sólo aumenta el realismo, sino que también puede usarse para evitar que la geometría configurada a la distancia surja repentinamente a medida que el espectador se acerca. La niebla puede especificarse con cualquier color, aunque la mayor parte de las veces tendrá un color gris. Suele simularse mediante la mezcla del color de la niebla con el color del fragmento al que se hace render. La contribución del color de la niebla en comparación con el color del fragmento se determina comúnmente mediante alguno de los tres cálculos de niebla diferentes: *lineal*, *exponencial* y *exponencial cuadrado*.

Si se usa el cálculo lineal, la intensidad de la niebla aumenta linealmente con la distancia que separa al fragmento de la cámara; entre más lejos esté algo, mayor será la influencia que el color de la niebla tendrá sobre el color de salida final. El área afectada por la niebla puede controlarse mediante el uso de dos valores que limitan su alcance. Estos parámetros se llamarán *inicio de la niebla* y *final de la niebla*, los cuales definen la distancia z del comienzo y el término de la niebla, respectivamente. El *factor de mezcla* (la contribución del color de la niebla al color final del fragmento) puede calcularse utilizando la siguiente fórmula:

```
blendFactor = (fogEnd - fragDistance)/(fogEnd – fogStart)
```

Para calcular el color final en GLSL, puede usarse la función `mix` en el fragment shader. La función `mix` tiene el siguiente prototipo:

```
genType mix(genType x, genType y, genType a)
genType mix(genType x, genType y, float a)
```

La función `mix` realiza el siguiente cálculo de `x` y `y` que devuelve el resultado:

$$Result = x \times (1 - a) + (y \times a)$$

Al pasar el color de la niebla como primer parámetro, el color del fragmento como segundo parámetro y el factor de mezcla como parámetro final se obtiene el color de la niebla, de este modo:

```
outColor = mix(fogColor, fragColor, blendFactor);
```

Los otros dos modos de niebla no están limitados por valores de inicio y fin. En vez de esto, todos los fragmentos se ven afectados por la niebla, la distancia de la cámara determina cuánta influencia tiene el color de la niebla en el fragmento. En la niebla exponencial, la intensidad aumenta exponencialmente de acuerdo con la distancia que hay de la niebla a la cámara. Aunque con estos modos de niebla no existe un rango definido, puede usarse un factor de densidad para controlar la intensidad de la niebla. El factor de mezcla para la niebla exponencial se calcula como:

```
blendFactor = exp(-fogDensity * fragDistance);
```

La función `exp` devuelve la potencia natural de su único parámetro. La niebla exponencial cuadrada proporciona una mejor calidad que la niebla exponencial regular. Los factores de mezcla pueden calcularse mediante:

```
blendFactor = exp2(-fogDensity * fragDistance);
```

Ejemplo de niebla

En el ejemplo de niebla incluido en el CD se muestran los tres modos de niebla en acción. Al presionar la barra espaciadora es posible cambiar entre los distintos modos.

Figura 8.11
El terreno con presencia de niebla.

Resumen

En este capítulo ha habido mucho que aprender. Se analizó un modelo básico de iluminación basado en el modelo de Blinn-Phong y usted aprendió a implementarlo en un nivel de vértice a vértice. También aprendió a simular una mezcla de superficies translúcidas. Por último, se estudió la aplicación de niebla en la escena y el uso de diferentes modos para conseguir distintos efectos de niebla.

Lo que se aprendió

- Las normales de superficie se utilizan para definir la dirección hacia donde está dirigida una superficie y se requieren para hacer los cálculos de iluminación.
- La iluminación se calcula al combinar las contribuciones de los diferentes tipos de luz: *difusa*, *ambiental*, *especular* y de *emisión*.
- Al calcular la iluminación se toman en cuenta las propiedades de los materiales para determinar el color final.
- La atenuación es el desvanecimiento de la luz con la distancia.
- En OpenGL se pueden modelar tres diferentes tipos de luces: *direccionales*, *puntuales* y de *foco*.
- Las luces direccionales no experimentan atenuación.
- La iluminación puede calcularse al nivel de los vértices o de pixel en pixel para una mayor calidad.
- El mezclado combina el color de un nuevo fragmento con el de uno que ya se encuentra en el búfer de fotogramas.
- El mezclado puede controlarse a través de los *factores de mezcla*, que se especifican mediante `glBlendFunc()`.
- La niebla se simula mediante la combinación de un color de niebla con el de un fragmento en función de su distancia a la cámara.

Preguntas de repaso

1. ¿Qué representa la contribución ambiental a la iluminación?
2. ¿Cuáles son las dos principales diferencias entre las luces puntuales y direccionales?
3. ¿Cuál es la función de la propiedad de brillantez de un material?

Por su cuenta

1. Escriba una aplicación de luz direccional al nivel de pixeles.
2. Extienda el ejemplo de las luces múltiples y añada una tercera luz puntual con un color amarillo difuso.

Más sobre el mapeo de texturas

El mapeo de texturas es un tema muy extenso, y la parte que se analizó en el capítulo 7 es sólo la punta del iceberg en la asignación de texturas.

En este capítulo, se estudiará:

- El uso de OpenGL para actualizar porciones de una textura existente
- El copiado de datos desde el búfer de fotogramas hacia una textura
- El uso de las pruebas alfa para hacer completamente transparentes ciertas partes de una superficie texturizada
- El uso del texturizado múltiple para aplicar más de una textura a una superficie

Subimágenes

La creación de una nueva textura es un proceso en OpenGL que utiliza muchos recursos. Cada vez que se llama a `glTexImage()`, OpenGL debe asignar memoria para contener la textura y realizar otras operaciones a fin de obtener la textura en un estado utilizable. Esto no es realmente un problema si ocurre una sola vez por textura durante la fase de carga de la aplicación. Pero en ocasiones es deseable actualizar una textura (¡o incluso varias!) en cada fotograma. Esta creación repetida de texturas puede acabar fácilmente con la velocidad del render.

Por fortuna, OpenGL proporciona una manera de actualizar una textura ya existente con nuevos datos, ya sea total o parcialmente. Esto es mucho más eficiente que reasignar memoria en

cada ciclo. Dada una textura existente, usted puede actualizar los datos de la imagen con una de las siguientes funciones:

```
void glTexSubImage1D(GLenum target, GLint level, GLint xoffset, GLsizei wi-
dth, GLenum format, GLenum type, const GLvoid* pixels);
void glTexSubImage2D(GLenum target, GLint level, GLint xoffset, GLint yoff-
set, GLsizei width, GLsizei height, GLenum format, GLenum type, const GLvoid
*pixels);
void glTexSubImage3D(GLenum target, GLint level, GLint xoffset, GLint yoff-
set, GLint zoffset, GLsizei width, GLsizei height, GLsizei depth, GLenum
format, GLenum type, const GLvoid* pixels);
```

La mayoría de los parámetros son iguales que para las funciones `glTexImage*()`. `xoffset`, `yoffset` y `zoffset`, que corresponden a las coordenadas izquierda, inferior y frontal (para texturas 3D) del área que se sustituirá por los nuevos datos. `width`, `height` y `depth` definen el tamaño del área. La zona delimitada debe ajustarse dentro de los límites de la textura.

Copiado desde el búfer de color

Imagine que en un juego su personaje visita un cuarto con una cámara para el control de seguridad. Dentro de la habitación hay una pantalla de televisión que muestra la vista de la cámara en el pasillo exterior. Para lograr este tipo de efecto, la imagen en la pantalla debe ser una textura asignada que se actualice en forma dinámica. Hay varias maneras de hacer esto, incluyendo las siguientes:

- Hacer un render de la escena en el búfer de fotogramas desde el punto de vista de la cámara y leer de nuevo los pixeles dibujados para almacenarlos en una textura.
- Hacer un render de la escena directamente desde la cámara a una textura, utilizando objetos del búfer de fotogramas.

La última de estas dos opciones es más eficiente, pero también más compleja y se encuentra fuera del alcance de este libro. El método más sencillo, que se estudiará aquí, es igual de eficaz aunque un poco más lento. OpenGL proporciona varias funciones para la lectura del búfer de fotogramas hacia una textura, en función de las dimensiones de la textura de destino:

```
void glCopyTexImage1D(GLenum target, GLint level, GLint internalformat,
GLint x, GLint y, GLsizei width, GLint border);
void glCopyTexImage2D(GLenum target, GLint level, GLint internalformat,
GLint x, GLint y, GLsizei width, GLsizei height, GLint border);
```

Estas funciones crean una textura completamente nueva a partir del área designada como dato de origen en el búfer de fotogramas. Los parámetros `target`, `level` e `internalformat` son los mismos que para las funciones `glTexImage()`. `border` se considera obsoleta y siempre debe ser igual a cero. `x`, `y`, `width` y `height` definen un rectángulo en el búfer de fotogramas de donde se copiarán los datos de textura, aquí `x` y `y` especifican la esquina inferior izquierda del rectángulo. No existe una versión en 3D de la función `glCopyTexImage()`, esto se debe a que no es posible crear una textura en 3D a partir de un búfer de fotogramas en 2D.

En ocasiones resulta útil actualizar sólo una parte de una textura empleando el búfer de fotogramas. Para este fin, OpenGL cuenta con las funciones `glCopyTexSubImage()`:

```
void glCopyTexSubImage1D(GLenum target, GLint level, GLint xoffset, GLint x,
GLint y, GLsizei width);
void glCopyTexSubImage2D(GLenum target, GLint level, GLint xoffset, GLint
yoffset, GLint x, GLint y, GLsizei width, GLsizei height);
void glCopyTexSubImage3D(GLenum target, GLint level, GLint xoffset, GLint
yoffset, GLint zoffset, GLint x, GLint y, GLsizei width, GLsizei height);
```

A diferencia de las funciones que crean una nueva textura, hay una versión en 3D de `glCopyTexSubImage()` la cual puede copiar los datos desde el búfer de fotogramas hacia una textura en 3D existente. Los parámetros adicionales para estas funciones, `xoffset`, `yoffset` y `zoffset` se usan para especificar la esquina inferior izquierda frontal de la región rectangular de destino donde se actualizará la textura. Podrá ver el uso de estas funciones en la demostración del mapeo del entorno que se aborda en la siguiente sección.

Mapeo del entorno

El mapeo del entorno es un proceso que le permite simular superficies reflectantes asignando el entorno circundante a una textura (de ahí se deriva el nombre). Después, el objeto aplica en forma dinámica las coordenadas de textura calculadas para dar una ilusión de espejo. Se estudiarán dos tipos de mapeo del entorno: el mapeo de esfera y el mapeo de cubos.

Mapeo de esfera

El *mapeo de esfera* es un método muy simple por medio del cual se obtiene el efecto de un reflejo. Cuando se usa el mapeo de esfera, las coordenadas de textura se generan al tomar el vector desde la visualización de una superficie y reflejarlo a través de la normal de la superficie. Este vector reflejado se utiliza para generar las coordenadas `s` y `t` y así buscar el color de texel en una textura especial. Una textura generada por mapeo de esfera es una imagen que ha sido pasada por un filtro del tipo “ojo de pez o gran angular”. La imagen resultante se ve como una esfera con el entorno circundante deformado. Los mapas de esferas tienen varias desventajas. En primer lugar, dependen de la vista, así que a menos que se esté observando el objeto en un

ángulo específico, la reflexión no parece realista. En segundo lugar, como siguen el modelo de una esfera, la aplicación de un mapa de este tipo a un objeto que no es esférico no se muestra correctamente. Por último (y probablemente lo más importante), casi siempre la imagen debe generarse en forma manual y mantenerse estática para que el reflejo no muestre objetos en movimiento por toda la escena. Afortunadamente, el mapeo de cubos corrige todos estos problemas y en realidad es mucho más fácil de aplicar en GLSL que el mapeo de esferas.

Mapeo reflectante de cubos

El mapeo reflectante de cubos puede proporcionar un reflejo más realista que un mapa de esferas. En el método de mapeo de cubos, la escena se dibuja con seis texturas en ángulos de 90 grados (norte, este, sur, oeste, arriba, abajo). Estas texturas se usan para formar un mapa de cubo. Las coordenadas de textura se generan en forma similar al método de mapeo de esferas, pero en lugar de generar las coordenadas de textura con sólo dos componentes (s y t), se genera una coordenada de textura con tres componentes (s, t y r). Después, estas coordenadas dan acceso a la imagen correcta del mapa de cubo para el color del texel; lo anterior permite que el reflejo dibuje con precisión el entorno. Observe la figura 9.1 para ver cómo funciona la búsqueda de la textura.

Figura 9.1
Las coordenadas del mapa de cubo se generan al reflejar el vector de visualización sobre la normal.

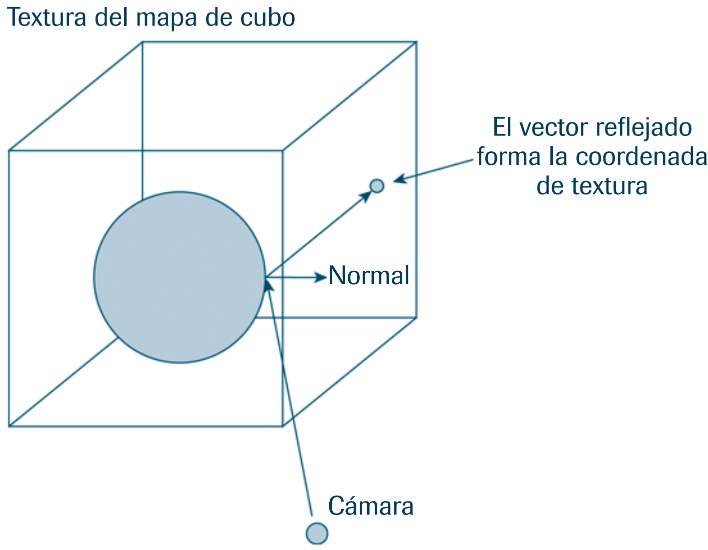
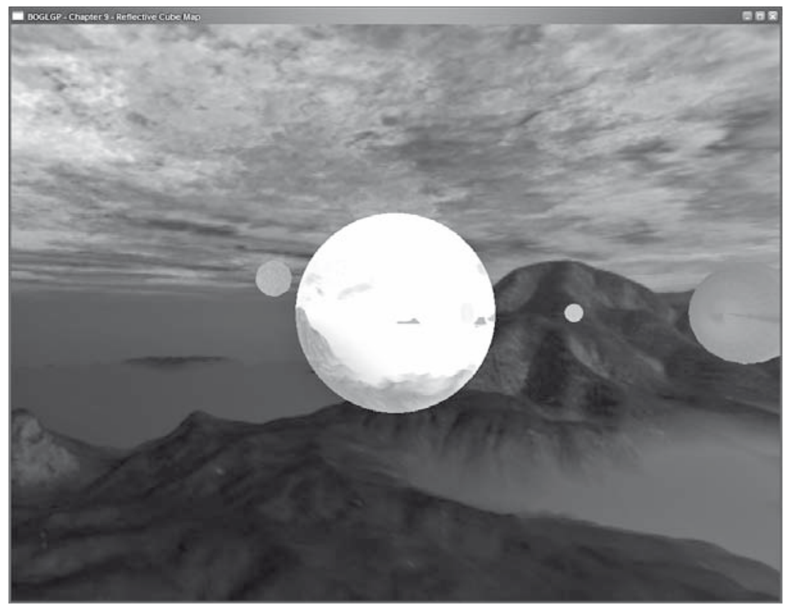


Figura 9.2
Captura de pantalla del ejemplo de mapeo de cubos.



La aplicación de muestra para el mapeo de cubos contiene un shader GLSL que genera las coordenadas de textura para el mapeo de cubos y las usa con el fin de reflejar órbitas giratorias en la superficie de una esfera.

Pruebas alfa

En el capítulo anterior se vio que, cuando se quiere representar opacidad, puede usarse el mezclado o blending para simular superficies translúcidas mediante el canal alfa. Pero, ¿y si se desea que una parte de cierta superficie sea completamente transparente, como los espacios en una cerca de malla de alambre? En este caso, si el canal alfa es un valor determinado se pueden descartar ciertos fragmentos, dando lugar a áreas transparentes con la forma de un polígono. La especificación del componente alfa de un fragmento puede hacerse usando un mapa de textura con un canal alfa, el mapa de textura puede muestrearse en el fragment shader y después compararse con un valor límite a fin de determinar si el fragmento debe desecharse.

Ahora se verá un ejemplo. Una manera simple y económica de hacer el render de los árboles en una escena es hacer una textura de árbol en 2D usando dos cuadrantes que se cruzan como en la figura 9.3.

Figura 9.3
Árbol sencillo con un render de dos cuadrantes.

Obviamente, no se desea que el fondo de la textura se visualice sobre los polígonos; sólo el árbol en sí. Si la imagen de textura, por ejemplo, almacena un valor de 0 en el canal alfa para todas las partes transparentes de la imagen, y un valor alfa de 1 para todas las partes visibles de ésta, es posible descartar los “fragmentos invisibles” en el fragment shader del modo siguiente:

```
#version 130
uniform sampler2D texture0;
in vec2 texCoord0;
out vec4 outColor;
void main(void) {
    //Sampling de la textura
    outColor = texture(texture0, texCoord0.st);

    //Si el componente alfa es demasiado bajo
    //este fragmento se descarta
    if (outColor.a < 0.1) {
      discard;
    }
}
```

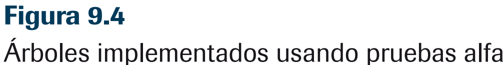
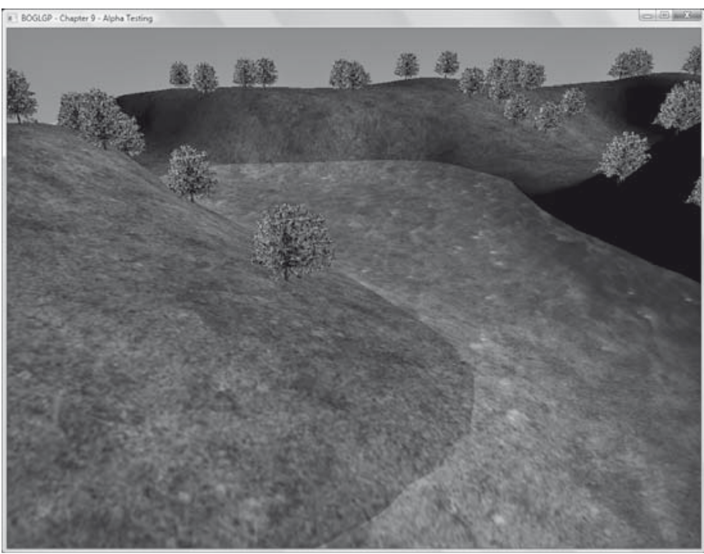


Figura 9.4
Árboles implementados usando pruebas alfa.



La palabra `discard` simplemente evita que el fragmento se siga procesando. En la figura 9.4 se muestra una versión actualizada de la aplicación del terreno, la cual presenta a los árboles mapeados con una textura que tiene áreas transparentes.

Texturizado múltiple

Dicho de forma simple, el texturizado múltiple (multitexturing) es el proceso de utilizar más de un mapa de textura sobre una sola superficie. En los ejemplos que se han visto hasta ahora, utilizamos una textura única para establecer el color difuso de cada fragmento. Este tipo de textura se conoce como *mapeo difuso*; sin embargo, las texturas tienen diversos usos, además de establecer el color difuso. Muchos juegos utilizan un mapa de textura en escala de grises, que se combina con el mapa difuso para dar la ilusión de una iluminación estática sobre una superficie. Esta técnica se conoce como *mapeo de luz*. Las texturas pueden contener información distinta a la del color. Como una textura RGB se forma de tres componentes por texel, es posible usar el mapa de textura para almacenar un vector normal comprimido de tres componentes para cada texel. Esto permite el empleo de una técnica conocida como *mapeo de relieve* (height map en inglés), el cual permite modelar superficies desiguales por medio de la iluminación por pixel que usa la normal que pasa desde el mapa de relieve. Éstas son sólo algunas de las formas más comunes en las que se utiliza el texturizado múltiple pero hay muchas otras, y las texturas se emplean de maneras innovadoras a cada instante.

Hasta ahora, al trabajar con texturas se ha empleado un uniforme de muestreo (o de sampling) en el fragment shader para acceder a los datos de textura. En la aplicación, a este uniforme se le ha dado el valor de 0. Dicho valor representa la unidad de textura a la que el sampler permite el acceso.

Unidades de textura

Cuando se hace una llamada a `glBindTexture()`, el objeto de textura se une a la "unidad de textura activa actual", que hasta ahora ha sido la unidad cero. Es posible cambiar la unidad de textura activa mediante la función `glActiveTexture()`, que tiene el prototipo siguiente:

```
void glActiveTexture(GLenum texture)
```

El argumento `texture` toma la forma de `GL_TEXTURE`*n* donde *n* es un valor entre 0 y `GL_MAX_TEXTURE_UNITS - 1`. Usted puede encontrar el número máximo de unidades de textura admitidas usando la función `glGetIntegerv()`:

```
int maxTexUnits; //Guarda el número máximo de unidades
glGetIntegerv (GL_MAX_TEXTURE_UNITS, &maxTexUnits);
```

Si el valor pasado a `glActiveTexture()` está fuera del rango válido se genera un error del tipo `GL_INVALID_ENUM`. Cada unidad de textura tiene su propia textura de entorno válida y sus propios parámetros de filtrado y datos de imagen actuales; además, los destinos de la textura (`GL_TEXTURE_2D`, `GL_TEXTURE_3D`, etcétera) están habilitados para cada unidad de textura.

El siguiente ejemplo muestra cómo vincular diferentes texturas a unidades de textura 0 y 1:

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texture1);
glActiveTexture (GL_TEXTURE1);
glBindTexture (GL_TEXTURE_2D, texture2);
```

Nota

Si está usando la segmentación de función fija, es necesario que llame a `glEnable()` con la textura de destino que se requiere para cada unidad de textura. Así, en el ejemplo anterior, sería necesario llamar a `glEnable(GL_TEXTURE_2D)` después de cada llamada a `glActiveTexture()`. Cuando se utiliza GLSL no es necesario hacer esto.

Texturizado múltiple en GLSL

Una vez que se han enlazado los objetos de textura con las unidades de textura que se desea utilizar, es necesario asociar un sampler con cada una de ellas, mediante establecer el valor de los uniformes del sampler para las unidades de textura respectivas. Entonces, será posible acceder a las texturas enlazadas usando la función de `GLSL texture()`. Los valores muestreados pueden usarse de la manera que más convenga, en función de sus propósitos. Los casos más

comunes de texturizado múltiple combinan los colores de texel, lo cual puede hacerse multiplicando los colores entre sí, o utilizando la función de `GLSL mix()`, que toma un valor de mezcla para determinar la cantidad en que cada color contribuye al valor final.

Coordenadas de texturas múltiples

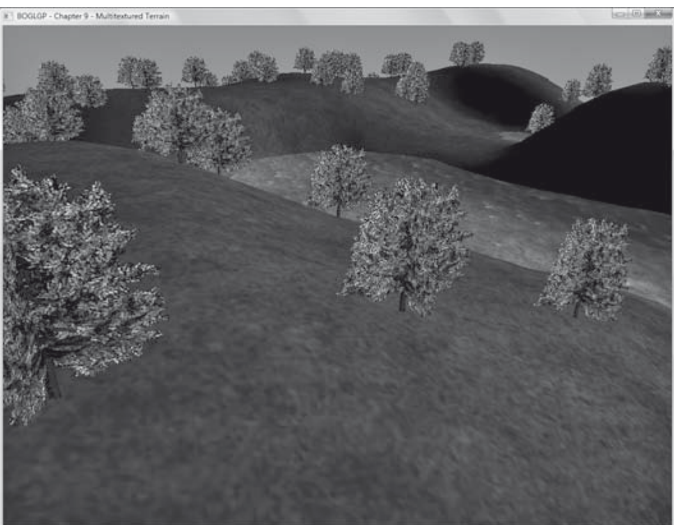
Las coordenadas de textura para las diferentes unidades de textura pueden especificarse al pasarlas como atributos de vértice, de la misma manera que lo hace con las coordenadas de textura para la primera unidad de textura. Después, se utiliza el atributo de coordenada respectivo como el segundo parámetro para la función `texture()` a fin de localizar el color de texel. Al igual que el primer conjunto de coordenadas de textura, el atributo debe enviarse a una variable `out` en el vertex shader para que sea interpolado por el fragment shader.

Ejemplo de texturizado múltiple

En el CD se encuentra una versión actualizada del ejemplo del terreno. Aquí se utilizan dos texturas; una textura 1D, que se emplea para colorear el terreno con base en la altura, y una textura 2D en escala de grises, que se apila para agregar los detalles de la hierba. Cuando estas texturas se combinan, el resultado es un terreno mucho más variado y realista. La coordenada de textura única para la textura en 1D se genera al tomar la altura del terreno y normalizarla en el rango 0.0–1.0. El punto más alto en el terreno obtiene una coordenada de 1.0 y el punto más bajo recibe una coordenada de 0.0. El terreno resultante puede verse en la figura 9.5.

Figura 9.5

Captura de pantalla del terreno con texturas múltiples.



Resumen

En este capítulo, usted aprendió que las partes de una textura existente pueden actualizarse de manera dinámica, lo cual resulta mucho más eficiente que sustituir la textura con una nueva. Aprendió que el conjunto de funciones `glTexSubImage*()` proporciona esta funcionalidad. Además, descubrió que el mapeo del entorno ofrece una manera sencilla de simular superficies reflejantes y que las imágenes reflejadas se pueden actualizar en forma dinámica mediante el copiado de datos desde el búfer de color. Por otra parte, aprendió que ciertas partes de un polígono pueden hacerse totalmente transparentes mediante el uso de las pruebas alfa para descartar los fragmentos no deseados en el fragment shader. También supo que el texturizado múltiple puede permitir la creación de efectos, como el mapeo de luz.

Lo que se aprendió

- Puede actualizar una textura, o parte de ella, con la familia de funciones `glTexSubImage*()`.
- Los datos pueden leerse desde el búfer de color después del render para luego utilizarlos en efectos como el mapeo del entorno.
- El mapeo del entorno le permite simular reflejos mediante el uso de coordenadas de textura generadas en forma dinámica.
- Al verificar el valor alfa de un fragmento y usar la palabra `discard`, es posible hacer que ciertas partes de un polígono sean transparentes.
- Es posible aplicar más de una textura a una sola superficie mediante el texturizado múltiple.

Preguntas de repaso

1. ¿Cuál es la función de la palabra `discard` del fragment shader?
2. ¿Cuáles son las desventajas del mapeo de esferas?
3. ¿Qué comando de OpenGL establece la unidad de textura activa?

Por su cuenta

1. Adapte el ejemplo de texturizado múltiple usando una tercera textura que le agregue sombreado al terreno.

Mejora del rendimiento

Los ejemplos que se han visto hasta ahora no son demasiado intensos para la GPU (de acuerdo con los estándares modernos), ya que el número de polígonos ha sido muy bajo. Sin embargo, los juegos complejos requieren cientos de miles de polígonos para representar cada fotograma, en ocasiones compuesto por millones de vértices. El procesamiento, la transformación, la iluminación y el render de todos estos polígonos sin ningún tipo de exclusión (culling) afectarán de forma significativa el rendimiento. La forma más importante de mejorar el rendimiento consiste en no hacer render de polígonos que el jugador no verá. Existen muchos métodos para excluir la geometría no visible. En este capítulo, se estudiará un solo método de exclusión de grupos de polígonos conocido como *frustum culling*.

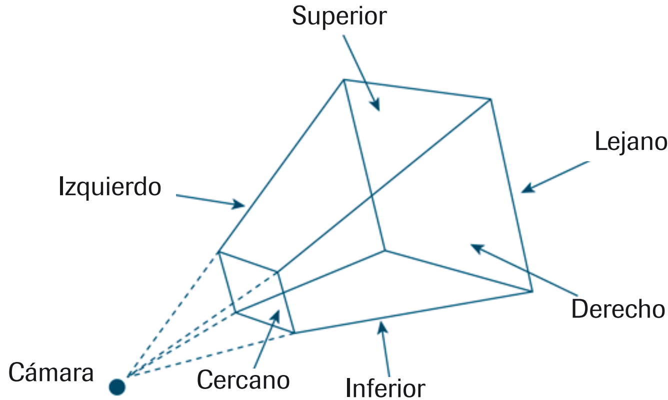
Frustum culling

El frustum culling es una manera rápida y eficaz de evitar el render de un gran número de polígonos que no se ven porque están fuera de la vista de la cámara. Como se aprendió en el capítulo 4 “Transformaciones y matrices”, la escena que se ve en OpenGL está contenida en un espacio geométrico conocido como *frustum*. El frustum de visualización para una proyección en perspectiva tiene la forma de una pirámide girada hasta una posición horizontal, con la cámara colocada en la punta (vea la figura 10.1).

Sólo los objetos que se encuentran total o parcialmente dentro del frustum de visualización se dibujarán en el búfer de fotogramas; cualquier otra cosa es recortada por OpenGL, pero no antes

de hacer una gran cantidad de procesamiento sobre objetos que nunca llegarán a la pantalla. El frustum culling es un método para verificar si los objetos están contenidos dentro del frustum antes de hacer el render. Si un objeto se encuentra fuera del frustum puede omitirse por completo.

Figura 10.1
Frustum de visualización para una escena proyectada en perspectiva.



Podría pensarse que la expresión "objeto" es un poco vaga. El frustum culling funciona mejor cuando se aplica a grupos asociados de polígonos. Por ejemplo, un modelo en 3D de un personaje sería adecuado para ser excluido por completo; si todo el modelo estuviera fuera del frustum entonces no se haría el render de ninguno de sus polígonos. Por lo tanto, la agrupación lógica de polígonos para que puedan excluirse en conjunto depende del usuario. Por lo general, esto implicará darle una primitiva de frontera a un conjunto de polígonos (por ejemplo, una esfera), para que se pruebe contra el frustum. La primitiva de frontera debe abarcar todos los polígonos que conforman el objeto.

Ecuación del plano

Un *plano* puede visualizarse como si fuera un pedazo de papel que se extiende infinitamente en todas direcciones, se define mediante la *ecuación del plano*, que es:

```
ax + by + cz + d = 0
```

`a, b, c` son las tres componentes de la normal del plano. `d` es la distancia del plano desde el origen. `x` y, `z` definen cualquier punto en el plano. Cualquier coordenada en el mundo 3D puede utilizarse para los argumentos `x, y, z` y si el resultado es 0, entonces el punto se encuentra sobre el plano. Si el resultado es positivo, el punto está frente al plano y si el resultado es negativo, el punto está detrás de éste. Si la normal del plano tiene una longitud unitaria, entonces el resultado

es la distancia en unidades a la que está el punto desde el plano. Esto es muy útil para la detección de colisiones, así como para el frustum culling, ya que el frustum de visualización puede definirse mediante seis planos (superior, inferior, izquierdo, derecho, cercano y lejano).

Definición del frustum

El primer paso para el frustum culling es el cálculo de los planos que componen el frustum de visualización, los cuales se almacenan en la matriz de proyección modelview. Si revisa el capítulo 4, recordará que esta matriz transforma los datos de vértice para recortar el espacio y se encuentra al multiplicar la matriz modelview por la matriz de proyección (algo que se ha estado haciendo con los vertex shaders en todos los ejemplos). La multiplicación manual de matrices involucra algo de código, pero afortunadamente hay un truco que hace que la pila de matrices de OpenGL realice todo el trabajo. Sin embargo, recuerde que la pila de matrices de OpenGL se considera obsoleta, por lo que las versiones futuras de este sistema pueden requerir la multiplicación manual de matrices. En vez de multiplicar la matriz en nuestro propio código, podemos realizar los siguientes pasos:

1. Tome las matrices modelview y proyección actuales (`glGetFloatv()`)
2. Introduzca la matriz modelview actual (`glPushMatrix()`)
3. Cargue la matriz de proyección almacenada (`glLoadMatrixf()`)
4. Multiplique la matriz modelview almacenada (`glMultMatrixf()`)
5. Tome el estado actual de la matriz modelview (`glGetFloatv()`)
6. Restaure la matriz modelview original (`glPopMatrix ()`)

El siguiente código hace exactamente eso:

```
GLfloat projection[16];
GLfloat modelview[16];
GLfloat mvp[16];
/* Toma de OpenGL las matrices MODELVIEW y PROYECCIÓN actuales */
glGetFloatv(GL_PROJECTION_MATRIX, projection);
glGetFloatv(GL_MODELVIEW_MATRIX, modelview);

glPushMatrix();
    //Carga la matriz de proyección almacenada
    glLoadMatrixf(projection);
    //Multiplica la matriz MODELVIEW almacenada por la matriz de
    proyección
    glMultMatrixf(modelview);
    //Lee el resultado de la multiplicación
    glGetFloatv(GL_MODELVIEW_MATRIX, mvp);
```

```
    //y restaura la antigua MODELVIEW_MATRIX
glPopMatrix();
```

Después de que este código se ha ejecutado, `mvp` contendrá la matriz modelview-proyección. A partir de esta matriz pueden obtenerse los seis planos que componen el frustum, ya sea sumando una de las tres primeras filas de la matriz a la cuarta fila, o restando una de las tres primeras filas de la cuarta. En la tabla 10.1 se muestran las filas que se suman o restan a la cuarta fila para obtener cada plano.

A modo de ejemplo, considere que para obtener los valores `a`, `b`, `c` y `d` para el plano cercano, debe hacerse lo siguiente:

```
a = mvp[3] + mvp[2];
b = mvp[7] + mvp[6];
c = mvp[11] + mvp[10];
d = mvp[15] + mvp[14];
```

Lo mismo puede hacerse para los otros planos, cambiando los índices de los elementos que se suman o restan y usando la operación correcta de la tabla 10.1. Una vez que se han obtenido los valores de los planos, es necesario normalizar el plano (no sólo la parte normal) al dividir a, b, c y d entre la longitud de la normal al (`a, b, c`). El código siguiente normalizará un plano cuando se tienen los valores de `a`, `b`, `c` y `d`:

```
Plane p; //Una estructura simple para contener nuestro resultado
float t = sqrt(a * a + b * b + c * c);
p.a = a / t;
p.b = b / t;
p.c = c / t;
p.d = d / t;
```

Tabla 10.1 Filas de origen para extraer los planos del frustum

Plano	Fila	Suma/Resta
Izquierdo	1ra.	Sumar
Derecho	1ra.	Restar
Inferior	2da.	Sumar
Superior	2da.	Restar
Cercano	3ra.	Sumar
Lejano	3ra.	Restar

Una vez que esto se repite para los seis planos, se tendrá la representación de un frustum válido contra el cual es posible comenzar a probar objetos.

Prueba de un punto

La verificación de la existencia de un punto dentro del frustum de visualización es una operación simple y representa la base para probar objetos más complejos como esferas. Si el punto está detrás de cualquiera de los planos que forman el frustum, tal punto se encuentra fuera del mismo. Como ya se ha dicho, al introducir un punto en la ecuación del plano se sabrá si éste se encuentra frente, detrás o sobre el plano. Simplemente se deben recorrer los seis planos, comprobando el punto contra cada uno de ellos. Si el punto está detrás de cualquiera de los planos, entonces se considera fuera del frustum. Si la matriz `m_planes` almacena los seis planos que componen el frustum, la siguiente función devolverá `true` si el punto especificado está dentro del tronco, o `false` en caso contrario.

```
bool PointInFrustum(float x, float y, float z)
{
  for (int p = 0; p < 6; p++ )
  {
    if (m_planes[p].a * x + m_planes[p].b * y +
        m_planes[p].c * z + m_planes[p].d < 0)
    {
      return false;
    }
  }
  return true;
}
```

Prueba de una esfera

La verificación de la existencia de una esfera dentro del frustum es sólo una extensión de la prueba de punto. Se sabe que el resultado de introducir un punto en la ecuación del plano devuelve la distancia entre el punto y el plano. Dada una esfera con un radio `R` y un punto central `P`, se puede determinar si una esfera está contenida en un frustum si la distancia desde el plano hasta `P` es mayor o igual a `R`. El código siguiente devolverá `true` si la esfera se encuentra al menos parcialmente dentro del frustum, o `false` en caso contrario.

```
bool sphereInFrustum(float x, float y, float z, float radius)
{
  for (int p = 0; p < 6; p++)
  {
```

```
    if (m_planes[p].a * x + m_planes[p].b * y +
      m_planes[p].c * z + m_planes[p].d <= -radius)
    {
      return false;
    }
  }
  return true;
}
```

Aplicación del frustum culling

En el CD puede encontrarse una aplicación de ejemplo que representa una serie de modelos de Quake 2 en el formato MD2 sobre un terreno. El frustum culling puede activarse y desactivarse mediante la barra espaciadora, y la cámara puede hacerse girar sobre el terreno usando las teclas de flecha izquierda y flecha derecha. Al cargarse cada modelo MD2 tiene asociada una esfera de frontera, la cual se prueba contra el frustum. Si la esfera está totalmente fuera del frustum de visualización, entonces no se hace el render del modelo. En la barra de títulos de la ventana se muestra un contador de "fotogramas por segundo", para que usted pueda observar el cambio en la velocidad de representación cuando el frustum culling está activado y desactivado. Preste atención especial a la clase `Frustum`, que contiene el código para calcular el frustum de visualización y verificar si los objetos son visibles. En la figura 10.2 se muestra una captura de pantalla de la aplicación del frustum culling en acción.

Figura 10.2

Captura de pantalla de la aplicación del frustum culling.

Resumen

En este capítulo se estudió un método simple para evitar el render de polígonos que se encuentran fuera del frustum de visualización. La exclusión de lo que no puede verse es una parte vital de la escritura de un dispositivo de juegos eficaz. Usted ha aprendido que la agrupación de la geometría para la exclusión puede aumentar enormemente la velocidad de los fotogramas en sus aplicaciones. También aprendió que un plano puede representarse mediante la ecuación del plano, y el frustum de visualización puede definirse mediante seis planos. Asimismo, se vio que al introducir un punto en la ecuación del plano se puede determinar la distancia entre el punto y el plano, este conocimiento puede servir para determinar si un punto está dentro o fuera del frustum de visualización.

Lo que se aprendió

- La exclusión de la geometría que no puede verse es de vital importancia para alcanzar una alta velocidad de fotogramas.
- El frustum culling es sólo una de las formas en la que se puede excluir la geometría invisible.
- Un plano se representa mediante la ecuación del plano.
- La ecuación del plano puede decirnos a qué distancia está un punto desde un plano.
- El frustum de visualización se compone de seis planos.
- El frustum de visualización puede extraerse de la matriz de proyección modelview.

Preguntas de repaso

1. ¿Cómo se extrae el plano izquierdo de la matriz de proyección modelview?
2. ¿Cómo se puede normalizar un plano?
3. ¿Qué es la ecuación del plano?

Por su cuenta

1. Investigue el algoritmo de exclusión Octree que hace uso del frustum culling.

11

Despliegue de texto

La mayoría de los juegos deben desplegar textos en pantalla para el usuario, por ejemplo, para mostrar la puntuación actual, el tiempo restante o incluso mensajes de otros miembros del equipo en los juegos con jugadores múltiples. Por desgracia, OpenGL no se ocupa por sí mismo de la funcionalidad de presentación de textos en su API, de manera que el despliegue de texto no es una tarea sencilla.

En este capítulo se estudiará:

- El uso de fuentes asignadas por textura en 2D
- La biblioteca FreeType
- El uso de la biblioteca FreeType para generar texto suavizado

Fuentes asignadas por textura en 2D

Las fuentes de mapa de textura en 2D representan una de las formas más simples para mostrar texto en OpenGL. Una *textura de una fuente* se crea por separado (normalmente se genera usando una aplicación para la creación de fuentes), suele contener una cuadrícula de 256 caracteres (16 líneas de 16 caracteres) y se carga como cualquier otra textura en su aplicación. En la figura 11.1 se muestra un ejemplo de textura de fuente. Cada carácter en una cadena se despliega como un cuadrilátero (formado por dos triángulos) con la parte de la textura que contiene el carácter asignado a éste. Para representar una cadena completa, es necesario manipular las coordenadas de textura para cada carácter.

Figura 11.1
Ejemplo de una textura de fuente.

Generación de las coordenadas de textura

La generación de las coordenadas de textura con base en el carácter es la parte más complicada al usar fuentes asignadas por textura. Como la textura tiene 16 caracteres en cada dirección, el ancho de cada carácter es 1/16vo. de la anchura total. Como las coordenadas de textura por lo general oscilan entre 0.0 y 1.0, el ancho y la altura de un carácter en coordenadas de textura es 1.0/16.0.

Las dimensiones de todos los caracteres son iguales, pero la posición de cada uno de ellos debe determinarse en forma dinámica. La posición X del carácter puede encontrarse utilizando el residuo de la división entera del valor ASCII del carácter entre 16. Del mismo modo, la posición Y se encuentra dividiendo el valor del carácter entre 16. El resultado de las posiciones X y Y debe multiplicarse por el ancho/la altura del carácter a fin de encontrar la posición final. El código resultante para determinar la posición de un carácter `(ch)` es el siguiente:

```
const float oneOverSixteen = 1.0f / 16.0f;
float xPos = float(ch % 16) * oneOverSixteen;
float yPos = float(ch / 16) * oneOverSixteen;
```

Mediante el uso de estas posiciones y las dimensiones de los caracteres, es posible encontrar las coordenadas de textura para las cuatro esquinas del carácter del modo siguiente:

```
texCoords[0] = xPos;
texCoords[1] = 1.0f - yPos - oneOverSixteen;

texCoords[2] = xPos + oneOverSixteen;
texCoords[3] = 1.0f - yPos - oneOverSixteen;

texCoords[4] = xPos + oneOverSixteen;
texCoords[5] = 1.0f - yPos;

texCoords[6] = xPos;
texCoords[7] = 1.0f - yPos;
```

Observe que la posición del eje y se resta de 1.0. Esto es para que las coordenadas de textura se encuentren entre 0.0 en la parte inferior y 1.0 en la parte superior. Si la sustracción no se hiciera, las texturas se invertirían.

Ejemplo de fuentes asignadas por textura

En el CD se puede encontrar una aplicación de ejemplo que carga una textura de fuente e imprime una cadena en la pantalla. El fragment shader en el ejemplo descarta cualquier pixel negro, lo cual quiere decir que si el texto se superpone a otros objetos, éstos se verán a través de las partes de la fuente que tienen un color distinto al blanco.

Fuentes en 2D con FreeType

Aunque las fuentes asignadas por textura son simples, tienen muchas desventajas. Una textura de fuente debe generarse por separado, lo cual puede llevar mucho tiempo si se están usando varias texturas. Las fuentes asignadas por textura también sufren de bordes irregulares o aliasing, y pierden su nitidez al escalarlas. Por último, cada caracter asignado en la textura se muestra del mismo ancho. A menos que utilice una fuente mono-espaciada, el espacio entre caracteres puede ser demasiado amplio. A pesar de estos inconvenientes, pueden ser útiles para pequeñas demostraciones independientes en las que el uso de una biblioteca no es lo ideal. Algunas de las desventajas de las fuentes asignadas por textura pueden solucionarse (por ejemplo, mediante el almacenamiento de la anchura de cada caracter), pero no es posible eliminarlas todas.

Figura 11.2
Captura de pantalla del ejemplo para las fuentes asignadas por textura en 2D.

La biblioteca FreeType

La biblioteca FreeType es un generador de fuentes de código abierto que puede cargar muchos formatos de archivo diferentes y proporcionarle a las aplicaciones una API para acceder a su contenido. La versión más reciente de la biblioteca FreeType puede encontrarse en http://www.freetype.org/, pero la versión actual en el momento de escribir este libro se incluye en el CD. Lo que la biblioteca FreeType puede proporcionar es un medio para cargar un archivo de fuente y hacer el render de cada carácter a una textura completa con anti-aliasing. Estas texturas pueden utilizarse para hacer el render de un texto. FreeType también da acceso a información de las fuentes como el ancho de un carácter o la posición de éste para letras como la "p", que puede extenderse por debajo de los otros caracteres en la línea.

Para utilizar FreeType durante la impresión de fuentes 2D en OpenGL, deben realizarse los siguientes pasos:

- Inicializar la biblioteca FreeType
- Cargar el archivo de fuente elegido
- Configurar el tamaño de fuente elegido usando la API de FreeType
- Cargar un glifo (forma del carácter) para cada carácter
- Generar una textura con base en el glifo cargado
- Descargar la fuente y la biblioteca
- Usar las texturas generadas para el render de los caracteres

Inicialización de FreeType y carga de una fuente

Para inicializar la biblioteca FreeType, debe utilizar la función `FT_Init_FreeType()` que tiene el siguiente prototipo:

```
FT_Error FT_Init_FreeType(FT_Library* alibrary);
```

El parámetro `alibrary` es un puntero hacia un objeto `FT_Library`, que se utiliza más adelante para cargar la fuente. `FT_Error` es una definición para un entero. `FT_Init_FreeType`, como la mayoría de las funciones en la API de FreeType, devolverá 0 si tiene éxito.

Una vez inicializada la biblioteca `FreeType`, es posible cargar una fuente (también conocida como una *face*). Un archivo de fuente se carga al pasar su ruta de acceso a la función `FT_New_Face()`:

```
FT_Error FT_New_Face(FT_Library library, const char*
filepathname, FT_Long face_index, FT_Face* aface);
```

`library` es el objeto `FT_Library`, que se inicializó previamente. `filepathname` es la ubicación del archivo de la fuente en el sistema de archivos. Algunos de éstos pueden contener varias caras o faces; `face_index` se utiliza para especificar cuál se desea cargar. El parámetro final es un puntero hacia un objeto `FT_Face`, que se emplea para proporcionar acceso a toda la información relacionada con la cara cargada. De nuevo, si hay una falla al cargar el tipo de fuente, esta función devolverá un valor distinto de cero.

Una vez que estas dos funciones se hayan completado correctamente, la biblioteca se habrá inicializado y el tipo de fuente habrá sido cargado, listo para acceder a la información de las fuentes. Esto se muestra en el código siguiente:

```
FT_Library library; // Crea una instancia de la biblioteca freetype

if (FT_Init_FreeType(&library))
{
  std::cerr <<“Could not initialize the freetype library” <<std::endl;
  return false;
}

FT_Face fontInfo; //Almacena información sobre la fuente cargada

//Ahora se intenta cargar información de la fuentes
if(FT_New_Face(library, fontName.c_str(), 0, &fontInfo))
{
  std::cerr << “Could not load the specified font” << std::endl;
```

```
  return false;
}
```

En el ejemplo anterior, FontName es un objeto std::string, que contiene la ruta al archivo de fuentes.

Configuración del tamaño de fuente

Una vez que se ha cargado la fuente (face), el siguiente paso consiste en especificar el tamaño que debe usarse. El tamaño de fuente se establece mediante la función FT_Set_Char_Size():

```
FT_Error FT_Set_Char_Size(FT_Face face, FT_F26Dot6
char_width, FT_F26Dot6 char_height, FT_UInt
horizontal_resolution, FT_UInt vertical_resolution);
```

face es el objeto FT_Face que almacena la información de la fuente. char_width y char_height son, respectivamente, el ancho y la altura requeridos para el carácter. FreeType almacena las dimensiones en unidades, que son el equivalente un 64vo. de un pixel. Esto significa que los valores pasados a char_width y char_height deben ser la altura y el ancho requeridos del pixel multiplicados por 64. Los dos últimos parámetros son la resolución horizontal y vertical en puntos por pulgada (DPI). Cuando se pasa un valor de cero en estos parámetros, se utiliza el resultado predeterminado que es de 72 dpi.

Generación de texturas de glifo

Después de haber especificado el tamaño de la fuente, puede comenzar a generar texturas a partir de la cara de la fuente. El primer paso es obtener un índice del glifo para el que se desea generar una textura. Para hacer esto, existe una función de la API de FreeType llamada FT_Get_Char_Index():

```
FT_Uint FT_Get_Char_Index(FT_Face face, FT_ULong charcode);
```

Esta función devuelve un índice para el carácter especificado mediante el *charmap* (mapa de carácter) seleccionado. Cada tipo de fuente contiene charmaps, que asignan caracteres a los glifos. FreeType usa de manera predeterminada el charmap Unicode contenido en la fuente. Si la fuente no contiene un charmap Unicode, FreeType volverá atrás para tratar de emular un charmap mediante la realización de comparaciones sobre los nombres de glifo.

Después, el índice devuelto se pasa a la función FT_Load_Glyph() que carga el glifo desde la cara de la fuente.

```
FT_Error FT_Load_Glyph(FT_Face face, FT_UInt glyph_index,
FT_Int32 load_flags);
```

Una vez más, face es el objeto Face_Font, glyph_index es el índice devuelto desde FT_Get_Char_Index(), y load_flags es una máscara de bits de opciones que se utiliza para cargar el glifo. La opción predeterminada es FT_LOAD_DEFAULT. En la tabla 11.1 se muestran algunas banderas de carga posibles que son de utilidad. La lista completa se puede encontrar en la documentación de FreeType. El glifo cargado se almacena en el atributo “glyph” de la face de la fuente.

Una vez que el glifo se carga, podemos recuperarlo y almacenarlo en una variable local mediante la función FT_Get_Glyph():

```
FT_Error FT_Get_Glyph (FT_GlyphSlot slot, FT_Glyph* aglyph);
```

slot es el atributo del glifo del objeto FT_Face y aglyph es un puntero hacia el objeto FT_Glyph, en el que se desea almacenar el glifo. El siguiente paso es decirle a FreeType que haga el render del glifo en un mapa de bits para que después sea posible cargarlo en una textura. Esto se hace mediante una función que lleva el nombre adecuado de FT_Glyph_To_Bitmap():

```
FT_Error FT_Glyph_To_Bitmap(FT_Glyph* the_glyph,
FT_Render_Mode* render_mode, FT_Vector* origin, FT_Bool destroy);
```

the_glyph es el glifo al que se va a hacer render. render_mode altera la manera en que se hace el render del glifo y puede tomar cualquiera de los parámetros de la tabla 11.2. El parámetro origin es un puntero hacia un vector que trasladará el glifo antes de dibujarlo; al pasar 0 (o NULL) no ocurrirá ninguna traslación. El parámetro final es una bandera que indica si el glifo original debe ser destruido después de convertirlo en un mapa de bits. Una vez que esta llamada se completa, el glifo debe enviarse a un objeto FT_BitmapGlyph.

Tabla 11.1 Banderas de carga útiles en FreeType

Opción	Comportamiento
FT_LOAD_DEFAULT	Devuelve un mapa de bits para el glifo; si no se encuentra un mapa de bits, entonces FreeType utiliza en su lugar el contorno de fuente escalable para el glifo.
FT_LOAD_NO_HINTING	Desactiva las sugerencias de fuentes.
FT_LOAD_RENDER	Después de cargar el glifo, se hace una llamada automática a la función FT_Render_Glyph.
FT_LOAD_NO_AUTOHINT	Deshabilita la generación automática de sugerencias.

Tabla 11.2 Modos de render en FreeType

Modo	Comportamiento
FT_RENDER_MODE_NORMAL	Genera mapas de 8 bits en escala de grises con anti-aliasing.
FT_RENDER_MODE_LIGHT	Igual que FT_RENDER_MODE_NORMAL.
FT_RENDER_MODE_MONO	Genera mapas de 1 bit.
FT_RENDER_MODE_LCD	Genera mapas de 8 bits 3x el ancho del glifo original.
FT_RENDER_MODE_LCD_V	Genera mapas de 8 bits 3x la altura del glifo original.

La generación de una textura a partir del mapa de bits cargado consiste sólo en iterar a través de los pixeles del mapa de bits y asignar éstos a un arreglo de datos de imagen, que después se cargan mediante `glTexImage2D()`. Se puede tener acceso a los datos de imagen a través del atributo `bitmap` del glifo, que a su vez se almacena en el atributo `buffer` de `bitmap`.

Al cargar cada glifo, tenga cuidado de almacenar los atributos adicionales de éste que puedan ser necesarios para su render. Esto incluye las dimensiones del glifo en píxeles, su posición (para las letras que se extienden debajo del renglón como “p” o “y”), así como el parámetro `advance` del glifo (es decir, qué tanto debe trasladarse para el siguiente carácter). El uso de estos atributos puede verse en la aplicación de ejemplo incluida en el CD.

Liberación de recursos en FreeType

Una vez que ha hecho el render de sus caracteres de glifo en texturas, ya no necesita la biblioteca FreeType ni el tipo de letra cargado. Estos recursos pueden liberarse mediante las funciones `FT_Done_Face` y `FT_Done_FreeType()`, que son las funciones para la cara y la biblioteca, respectivamente:

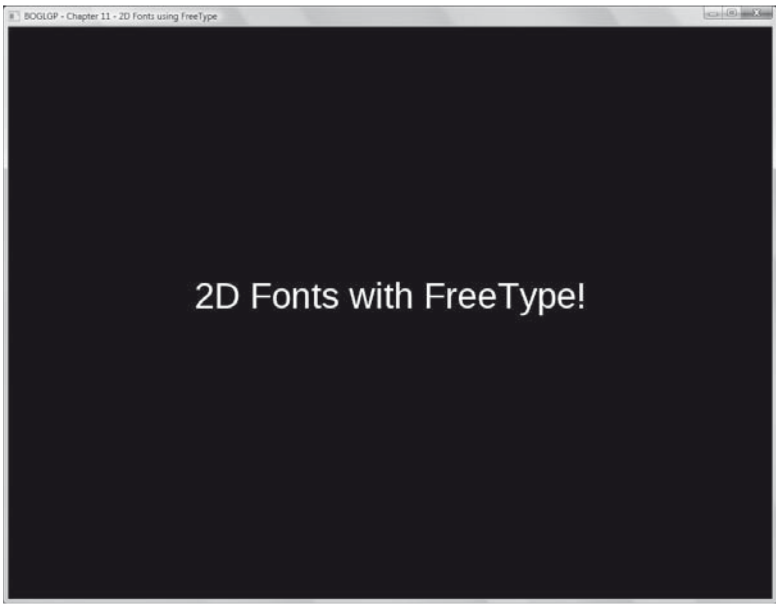
```
FT_Error FT_Done_Face(FT_Face face);
FT_Error FT_Done_FreeType(FT_Library library);
```

Ejemplo de FreeType

En el CD puede encontrarse una aplicación de muestra con una clase FreeTypeFont que realiza la carga de un archivo de fuente, como puede ser una fuente TrueType (.ttf) y genera texturas para el conjunto de caracteres ASCII. La extensión de la aplicación para caracteres Unicode se deja como ejercicio para el lector.

Generar una textura para cada carácter es mucho menos eficiente que generar una textura única que contiene muchos caracteres y manipular sus coordenadas de textura para mostrarlos. La acción de vincular de nuevo la textura para cada carácter puede ser bastante lenta cuando se

Figura 11.3
Captura de pantalla del ejemplo FreeType.



hace el render de una gran cantidad de texto. Por motivos de simplicidad, en el ejemplo se utiliza el enfoque de texturas múltiples.

Una nota sobre fuentes en 3D

En este capítulo se han estudiado dos métodos para el render de fuentes en 2D. También es posible hacer el render de texto que se ha proyectado en tres dimensiones. Este texto puede rotarse y trasladarse de la misma forma que cualquier otro objeto en 3D. En el presente libro no se cubrirán las fuentes tridimensionales por varias razones, de manera específica:

- Los métodos más comunes para el render de texto en 3D son específicos de una plataforma.
- Muchos métodos hacen uso intensivo de las listas de visualización, que ahora se consideran obsoletas.
- Al momento de escribir este libro, las bibliotecas de fuentes en 3D disponibles no se habían actualizado para eliminar el uso de funcionalidades obsoletas.
- Generar fuentes en 3D mediante la información de FreeType es un proceso complicado que está fuera del alcance de este libro.

Si usted utiliza la plataforma Windows y desea emplear fuentes en 3D (y está conforme con usar las listas de visualización obsoletas) la función `wglUseFontOutlines()` genera caracteres

en 3D y las guarda como listas de visualización. En la web hay muchos tutoriales y artículos sobre el uso de *fuentes en curvas*. Un método más práctico para el render de fuentes en 3D es usar las medidas proporcionadas por la biblioteca FreeType para generar manualmente polígonos en 3D que representan carácteres. Aunque éste es un procedimiento complicado, hay muchos ejemplos disponibles en línea.

Resumen

En este capítulo, usted aprendió dos métodos diferentes para imprimir texto en pantalla. Se estudió cómo utilizar fuentes en 2D asignadas por textura como un medio práctico para el render de texto básico. También aprendió sobre la biblioteca FreeType y cómo usarla para cargar caracteres desde un archivo de fuentes. Por último, descubrió un método para usar la biblioteca FreeType a fin de generar texturas y hacer el render de ellas a manera de polígonos que se imprimen como texto.

Lo que se aprendió

- OpenGL no tiene una funcionalidad integrada para el procesamiento de texto.
- Las fuentes en 2D asignadas por textura proporcionan una forma rápida y práctica para el render de texto.
- Las fuentes en 2D asignadas por textura tienen varias desventajas, como la mala calidad y el aliasing.
- La biblioteca FreeType es un generador práctico de fuente abierta.
- FreeType puede cargar muchos formatos de fuente y proporcionar acceso a la información de las fuentes.
- FreeType puede utilizarse para generar texturas con anti-aliasing durante el tiempo de ejecución, lo que proporciona una salida con mejor calidad que las fuentes asignadas por textura.

Preguntas de repaso

1. ¿Cuáles son los problemas inherentes a las fuentes asignadas por textura?
2. ¿Qué es la biblioteca FreeType?
3. ¿Cómo se establece el tamaño de la fuente con FreeType?

Por su cuenta

1. Extienda la aplicación de FreeType para que pueda trabajar con caracteres Unicode.
2. Modifique la aplicación de FreeType para que el render de las texturas de carácter se haga hacia una textura única en vez de hacia texturas múltiples.

Búferes de OpenGL

A lo largo del libro se han usado varios búferes de OpenGL sin tener que observarlos con detalle.

En este capítulo se hará un estudio más detallado de los siguientes temas:

- Operaciones generales de búfer
- El búfer de color
- El búfer de profundidad
- El búfer de plantilla

¿Qué es un búfer de OpenGL?

Un búfer en OpenGL es un área de memoria consecutiva que se usa para guardar de manera temporal los datos relacionados con pixeles o vértices. Existen varios tipos diferentes de búferes que almacenan datos relacionados con los pixeles, que al combinarse forman un solo *búfer de fotogramas*. Los principales tipos de búferes que se combinan para crear el búfer de fotogramas son los de profundidad, de plantilla y de color. En las versiones actuales de OpenGL, hay un cuarto búfer conocido como búfer de acumulación; sin embargo, el uso de éste se considera obsoleto, por lo que no se estudiará en el presente capítulo.

Tabla 12.1 Valores de máscara válidos

Valor	Búfer
`GL_COLOR_BUFFER_BIT`	Borra los búferes de color actualmente habilitados
`GL_DEPTH_BUFFER_BIT`	Borra el búfer de profundidad
`GL_STENCIL_BUFFER_BIT`	Borra el búfer de plantilla
`GL_ACCUM_BUFFER_BIT`	Borra el búfer de acumulación (obsoleto)

A lo largo de este libro, se han usado los búferes de vértices (vertex buffers) en la forma de vertex buffer objects. Este capítulo se centrará en los búferes de pixeles (pixel buffers) en OpenGL que están relacionados con el búfer de fotogramas (frame-buffer).

Limpieza de búferes

Antes de usar un búfer, éste debe limpiarse a fin de tenerlo listo para el almacenamiento de una nueva serie de datos. Por lo general, los búferes se deben limpiar al inicio de cada fotograma. Si no se limpian, entonces cualquier parte del búfer que no haga un render en un fotograma aún contendrá los datos del fotograma anterior, lo cual puede ocasionar problemas con el render. Los búferes de OpenGL se limpian usando la función `glClear()`:

```
void glClear (GLbitfield mask)
```

El parámetro `mask` es una OR de valores por bit que determina los búferes que serán limpiados. Hasta ahora, en los ejemplos se ha usado `glClear()` para limpiar búferes de profundidad y de color al mismo tiempo, con el siguiente comando:

```
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
```

Puede usarse cualquier combinación de los valores mostrados en la tabla 12.1 para seleccionar los búferes que se desea limpiar.

Prueba de tijera

La prueba de tijera es una característica muy poderosa que en ocasiones se pasa por alto. OpenGL permite definir un área rectangular de la pantalla llamada cuadro de tijera. Una vez definido, todos los comandos de render sólo pueden afectar a la zona del búfer de fotogramas dentro del cuadro de tijera. Si intenta dibujar cualquier cosa fuera de este cuadro, la imagen no se desplegará. La prueba de tijera puede habilitarse al pasar `GL_SCISSOR_TEST` a `glEnable()`

y se puede deshabilitar con una llamada correspondiente a `glDisable()`. En un inicio, el cuadro de tijera se define del mismo tamaño que la ventana establecida durante la creación del contexto de OpenGL. Si el tamaño de la ventana no ha vuelto a cambiarse, entonces la habilitación de la prueba de tijera no tendrá ningún efecto. Para establecer las dimensiones del cuadro de tijera, se llama a `glScissor()`:

```
void glScissor(GLint x, GLint y, GLsizei width, GLsizei height);
```

`x` y `y` especifican la esquina inferior izquierda del cuadro de tijera en las coordenadas de ventana. `width` y `height` son las dimensiones del cuadro también en las coordenadas de ventana. Recuerde que todos los comandos de render sólo afectarán a la zona que se encuentra dentro del cuadro de tijera, lo que incluye a `glClear()`.

Los búferes de color

Un búfer de color almacena un color para cada pixel de la pantalla. De hecho, un contexto de OpenGL puede contener varios búferes de color, pero la mayoría de las aplicaciones hacen uso de dos búferes: el frontal y el posterior. Hasta ahora, en todos los ejemplos se ha utilizado un contexto de doble búfer. En un contexto de este tipo, toda operación de render que no se recorta de alguna manera se procesa en el búfer posterior fuera de la pantalla. Al final del fotograma los búferes se intercambian, de manera que el búfer de reserva se vuelve visible en pantalla en el búfer frontal. El render para el siguiente fotograma afecta al nuevo búfer posterior hasta que los búferes se intercambian de nuevo. Este proceso impide un artificio llamado *desgarro*, que puede ocurrir en un contexto de búfer único. En términos generales, cualquier mención del *búfer de color* se refiere al búfer que es el objetivo actual del render (por lo general el búfer posterior).

Enmascaramiento de color

Es posible que para algunos efectos de render no quiera representar todos los canales de color. Por ejemplo, puede optar por sólo hacer el render del canal de color verde para obtener un efecto de visión al estilo de unas gafas nocturnas. En algunos casos, es posible que desee deshabilitar por completo la escritura al búfer de color. Esto resulta útil cuando sólo va a escribir en los búferes de profundidad o de plantilla. OpenGL proporciona una función, `glColorMask()`:, que puede activar o desactivar los canales de color de manera individual.

```
void glColorMask(GLboolean r, GLboolean g, GLboolean b, GLboolean a);
void glColorMask(GLuint buf, GLboolean r, GLboolean g, GLboolean b, GL-
boolean a);
```

La primera versión de esta función activa o desactiva los canales individuales del búfer de color en el del búfer de fotogramas predeterminado. Al pasar `GL_FALSE` como uno de los

argumentos, se desactivará la escritura de color a ese canal; si se pasa `GL_TRUE` entonces se activa la escritura de color.

Todos los canales de color están habilitados por defecto. La segunda versión de `glColorMask()` se usa junto con objetos del búfer de fotogramas, que se analizarán en forma breve más adelante en este capítulo. Los canales habilitados afectarán a todas las operaciones que alteren el búfer de color, incluyendo `glClear()`.

Establecimiento del color de limpieza

Cuando se llama a `glClear()` con `GL_COLOR_BUFFER_BIT` como uno de los valores de la máscara, el búfer de color se limpia tomando el color de limpieza. Este color se establece mediante la función `glClearColor()`:

```
void glClearColor(GLclampf r, GLclampf g, GLclampf b, GLclampf a);
```

Los argumentos `r`, `g`, `b` y `a` son, respectivamente, los componentes rojo, verde, azul y alfa del color de limpieza. El valor por defecto del color de limpieza es negro con alfa cero (todos los componentes son 0.0).

El búfer de profundidad

El búfer de profundidad (que también se conoce como búfer z) registra el valor de la profundidad de los píxeles que se dibujan en pantalla. Cuando la prueba de profundidad está habilitada, la distancia entre el render de un pixel y el punto de visualización (el componente z) se compara con el valor correspondiente (es decir, el que se encuentra en la misma ubicación x/y) actualmente almacenado en el búfer de profundidad. Si la distancia es mayor (es decir, el pixel está más lejos) que el valor almacenado actualmente, entonces (tradicionalmente) el pixel no se dibuja. De lo contrario, el pixel se dibuja y el valor en el búfer de profundidad se actualiza. Esta *prueba de profundidad* evita el render de los objetos que están lejos de la cámara sobre otros que estén más cerca. Es necesario habilitar la comparación de la profundidad al pasar `GL_DEPTH_TEST` a `glEnable()` y puede desactivarse mediante `glDisable()`. Las pruebas de profundidad están deshabilitadas de manera predeterminada.

Control de la prueba de profundidad

Aunque el comportamiento habitual de la prueba de profundidad para un pixel sólo consiste en que se hace un render de él si pasa la prueba (es decir, el valor de z entrante es menor o igual al valor correspondiente en el búfer de profundidad), el comportamiento puede controlarse al cambiar la función de comparación que determina si se pasa la prueba de profundidad o no. La función de comparación se puede controlar mediante la función `glDepthFunc()`:

```
void glDepthFunc(GLenum func);
```

Tabla 12.2 Funciones de comparación de la profundidad

Valor	Comportamiento
GL_NEVER	El fragmento entrante no pasa la prueba de profundidad y se descarta.
GL_ALWAYS	El fragmento entrante siempre pasa la prueba de profundidad.
GL_LESS	Si el valor entrante de z es menor que el valor almacenado, el fragmento pasa.
GL_EQUAL	Si el valor entrante de z es igual al valor almacenado, el fragmento pasa.
GL_LEQUAL	Si el valor entrante de z es menor o igual que el valor almacenado, el fragmento pasa.
GL_GREATER	Si el valor entrante de z es mayor que el valor almacenado, el fragmento pasa.
GL_GEQUAL	Si el valor entrante de z es mayor o igual que el valor almacenado, el fragmento pasa.
GL_NOTEQUAL	Si el valor entrante de z no es igual al valor almacenado, el fragmento pasa.

El parámetro `func` puede ser cualquiera de los valores de la tabla 12.2.

Aunque al pasar `GL_ALWAYS` parece obtenerse el mismo comportamiento que al desactivar por completo las pruebas de profundidad, hay una diferencia. Cuando se utiliza `GL_ALWAYS`, el búfer de profundidad se actualiza con los valores de profundidad entrantes. Si la prueba de profundidad está desactivada, entonces el búfer de profundidad no se actualiza de ninguna manera.

Desactivación de la escritura en el búfer de profundidad

De manera similar a la forma en que se desactiva la escritura en el búfer de color con `glColorMask()`, la escritura en el búfer de profundidad puede deshabilitarse mediante `glDepthMask()`:

```
void glDepthMask(GLboolean mask);
```

Al pasar `GL_FALSE` se desactivará la escritura en el búfer de profundidad y con `GL_TRUE` puede volverse a activar. La deshabilitación de la escritura en el búfer de profundidad es especialmente

útil cuando se utiliza un sistema de partículas (vea el capítulo 13 "El juego final"). Un sistema de partículas se compone de muchos polígonos individuales cuya profundidad debe probarse contra la geometría existente, pero no entre sí. Un enfoque consistiría en ordenar las partículas de atrás hacia delante (de la más alejada a la más cercana) antes del render, lo que sería increíblemente ineficiente, en especial para los sistemas con un gran número de partículas. Un enfoque mejor y más eficaz consiste en hacer el render del sistema de partículas al final y deshabilitar la escritura al búfer de profundidad antes de representar las partículas. Así es como se manejan las explosiones en el juego que se estudia en el próximo capítulo.

Problemas potenciales

Hay un par de cosas relacionadas con el búfer de profundidad que es necesario vigilar durante el render. Estos problemas comunes son el z-fighting (conflicto en Z) y el render erróneo de superficies transparentes.

Z-fighting

El búfer de profundidad tiene una precisión limitada. Cuando se pretende determinar el formato de pixel para una ventana de OpenGL, una de las opciones es el número de bits por pixel reservado para el búfer de profundidad; entre más bits haya disponibles, mayor será la precisión. El Z-fighting se produce cuando dos polígonos se superponen entre sí a una profundidad similar. Debido a la falta de precisión, algunos fragmentos que deberían fallar la prueba de profundidad pueden pasarla, lo cual puede causar artificios visuales. Por lo general, el aumento del número de bits asignados al búfer de profundidad ayudará a resolver el problema. La otra (y mejor) opción consiste en acortar la distancia entre los planos cercano y lejano que hacen que el búfer z sea más preciso. Otro factor es que el búfer z tiene una precisión más alta en la proximidad del plano cercano, que se degrada al avanzar hacia el plano lejano. Esto significa que si se aleja el plano cercano la precisión mejorará mucho más que si se acerca el plano lejano.

Éstas son las maneras posibles de reducir la probabilidad de que se presenten Z-fightings, sin embargo, hay ocasiones en las que dichos conflictos son inevitables y la única solución es evitar las situaciones en las que los polígonos superpuestos compartan el mismo plano.

Representación de superficies transparentes

En el capítulo 8 se cubrió en forma breve este problema al estudiar las mezclas, pero ahora que se ha tratado en detalle el búfer de profundidad, probablemente es necesario hacer una recapitulación.

Las pruebas de profundidad representan una excelente manera de evitar el render de fragmentos que se ocultan detrás de otros que ya se han representado previamente. Esto se convierte en un problema con los polígonos transparentes, porque en este caso el espectador *necesita* ver los polígonos que están detrás de otros objetos. Si se hace el render de un polígono transparente

antes de los polígonos que están detrás de él, éstos no pasarán la prueba de profundidad y no se dibujarán. Por supuesto, el enmascaramiento de profundidad permite deshabilitar la escritura en el búfer de profundidad, por lo que un enfoque común (pero erróneo) consiste en deshabilitar la escritura de profundidad para los polígonos transparemtes. Esto permitirá el render de los fragmentos que están detrás de la superficie transparente, pero la transparencia no se mostrará correctamente. Como ya se vio en el capítulo 8, esta última se logra mezclando el fragmento de entrada con el valor en el búfer de color. Por supuesto, lo anterior supone el render previo de los polígonos que van a aparecer detrás de la superficie transparente; la desactivación de la escritura de profundidad no ayudará, porque el orden de render de los polígonos sí importa, y se debe hacer el render de las superficies transparentes de las que se encuentran detrás de ella.

El enfoque habitual consiste en efectuar primero el render de todas las superficies opacas de la escena y después hacerlo para las superficies transparentes de atrás hacia adelante (de la más lejana a la más cercana). La mayoría de las veces, esto dará muy buenos resultados, pero hay situaciones en las que no es posible determinar el orden correcto de los polígonos de atrás hacia adelante.

Búfer de plantilla

El búfer de plantilla es un búfer de propósito general que, al combinarse con la prueba de plantilla, puede usarse para lograr diversos efectos. Se emplea comúnmente para limitar el render en ciertas partes de la pantalla y para efectos tales como el reflejo, donde sólo se hace el render de los objetos en la superficie reflejante. El búfer de plantilla también se usa en las técnicas de sombreado o shading.

Durante la creación del contexto es necesario habilitar el búfer de plantilla. En Windows esto se hace al establecer el parámetro `cStencilBits` del `PIXELFORMATDESCRIPTOR` de la siguiente manera:

```
pfd.cStencilBits = 8;
```

Esto asigna ocho bits para cada pixel en el búfer de plantilla. Para habilitar la prueba de plantilla, se pasa `GL_STENCIL_TEST` a `glEnable()`. Cuando la prueba de plantilla está activada, su comportamiento se controla mediante la función de plantilla y la operación de plantilla. La primera de ellas define una función de comparación que determina cuáles fragmentos pasan la prueba y cuáles no. La operación de plantilla especifica lo que ocurre cuando un fragmento pasa o no la prueba. La función de prueba de plantilla requiere un valor de referencia y una máscara. Cuando se procesa un fragmento, el parámetro AND del valor de referencia se compara con el parámetro AND del valor de plantilla para el pixel y la máscara. La función de plantilla, el valor de referencia y la máscara se especifican mediante la función `glStencilFunc()`:

```
void glStencilFunc(GLenum func, GLint ref, GLuint mask);
```

Los valores posibles para `func` se enlistan en la tabla 12.3.

Si un fragmento se procesa, lo que ocurre con el búfer de plantilla está determinado por la operación de plantilla, que puede especificarse mediante la función `glStencilOp()`.

Tabla 12.3 Funciones de plantilla

Función	Descripción
GL_NEVER	Siempre falla.
GL_ALWAYS	Siempre pasa.
GL_LESS	El fragmento pasa si (referencia y máscara) < (plantilla y máscara).
GL_LEQUAL	El fragmento pasa si (referencia y máscara) <= (plantilla y máscara).
GL_GREATER	El fragmento pasa si (referencia y máscara) > (plantilla y máscara).
GL_GEQUAL	El fragmento pasa si (referencia y máscara) >= (plantilla y máscara).
GL_EQUAL	El fragmento pasa si (referencia y máscara) == (plantilla y máscara).
GL_NOTEQUAL	El fragmento pasa si (referencia y máscara) / (plantilla y máscara).

Al llamar a `glStencilOp()`, se especifica lo que ocurre en tres situaciones diferentes:

1. La prueba de plantilla falla (`fail`)
2. La prueba de plantilla pasa, pero la prueba de profundidad falla (`zfail`).
3. La prueba de plantilla y la prueba de profundidad pasan (o la prueba de plantilla pasa pero la de profundidad está desactivada) (`zpass`).

La función `glStencilOp()` tiene la siguiente definición:

```
void glStencilOp(GLenum fail, GLenum zfail, GLenum zpass);
```

Los tres parámetros determinan lo que ocurre en cada una de las tres situaciones mencionadas anteriormente. Cada argumento puede tomar cualquier valor de la tabla 12.4.

Tabla 12.4 Operaciones de plantilla

Función	Descripción
GL_KEEP	Mantiene el valor actual de plantilla.

Tabla 12.4 Operaciones de plantilla (continuación)

GL_ZERO	Establece el valor de plantilla en cero.
GL_REPLACE	Establece el valor de plantilla en el valor de referencia.
GL_INCR	Incrementa el valor de plantilla.
GL_DECR	Disminuye el valor de plantilla.
GL_INVERT	Al nivel de bits, invierte el valor de plantilla.
GL_INCR_WRAP	Igual que GL_INCR, pero ajusta el valor de plantilla a cero cuando se alcanza el valor representable máximo.
GL_DECR_WRAP	Igual que GL_DECR, pero el valor se establece alrededor del valor representable máximo si se aplica a un valor de plantilla igual a cero.

Como se mencionó anteriormente, los búferes de plantilla se utilizan regularmente para restringir el render de objetos reflejados en la superficie reflejante. Por ejemplo, imagine una escena con un espejo en el suelo. El reflejo puede simularse mediante la representación de la escena dos veces, una normal y la segunda vez volteada verticalmente con el render restringido al espejo. El búfer de plantilla puede utilizarse para restringir el render dentro del espejo de la siguiente manera:

```
glEnable(GL_STENCIL_TEST); //Habilita el uso de plantillas
glDepthMask(GL_FALSE); //Deshabilita la escritura de profundidad

//Siempre reemplace el valor de plantilla ya sea que pase o no
glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);

//Hacer que la prueba siempre pase
glStencilFunc(GL_ALWAYS, 1, OxFFFFFFFF);

drawMirror(); //Hace el render de la superficie reflejante

//Habilitar de nuevo la escritura de profundidad
glDepthMask(GL_TRUE);
//Sólo hacer el render cuando el valor de plantilla es 1
glStencilFunc(GL_EQUAL, 1, OxFFFFFFFF);
//No volver a cambiar nunca los valores de plantilla
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
```

```
drawFlippedScene(); //Hace el render de la escena volteada

//Deshabilita la prueba de plantilla
glDisable(GL_STENCIL_TEST);

//...

drawScene();
```

Ahora analizaremos paso a paso el semi-pseudocódigo anterior. Primero se desactivan las pruebas de profundidad. El espejo no debe afectar el búfer de profundidad o, de lo contrario, podría ocultar los objetos reflejados y sería posible que no se pudiera hacer el render de ellos. Después, se deshabilita la prueba de plantilla. La operación de plantilla está establecida para reemplazar siempre el valor de plantilla (ya sea que pase la prueba o no). La función de plantilla se establece para pasar siempre. Una vez que se hace el render del espejo, el búfer de plantilla estará lleno de ceros, a excepción de los pixeles que componen al espejo, los cuales se establecen en 1.

Después se vuelve a habilitar la escritura de profundidad para el render la escena reflejada; esta vez los fragmentos pasan la prueba *sólo* cuando el búfer de plantilla es igual a 1 (es decir, donde estaba el espejo). Una vez que se han elaborado los reflejos, la prueba de plantilla puede desactivarse a fin de hacer el render de la escena normal (no volteada).

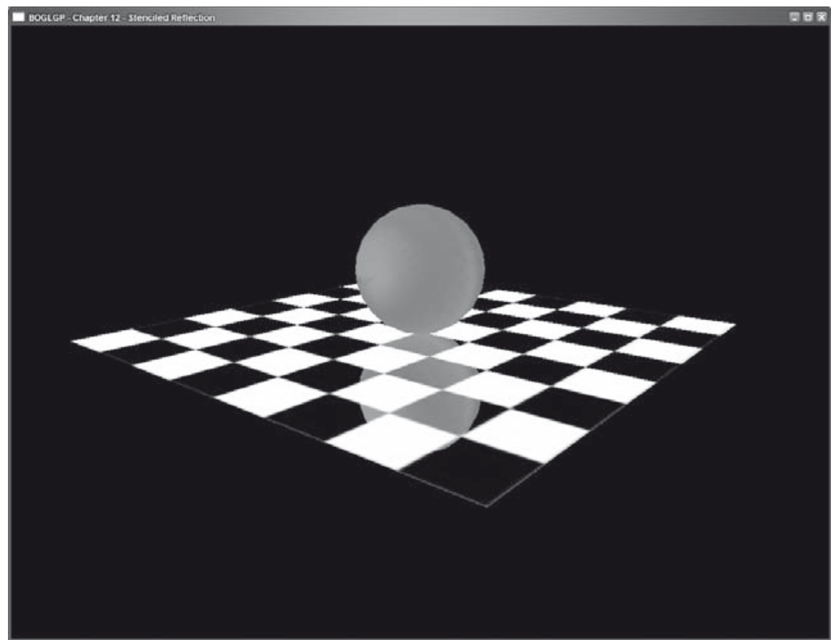
Esto es todo para el búfer de plantilla. Lo último que resta por mencionar es que cuando se use el búfer de plantilla, no debe olvidarse de limpiarlo en cada fotograma, junto con los otros búferes (de color y profundidad).

Una nota sobre los objetos del búfer de fotogramas

Los búferes mencionados en este capítulo se combinan para formar el búfer de fotogramas predeterminado. Sin embargo, OpenGL permite la creación de *framebuffer objects,* los cuales pueden considerarse como otros búferes de fotograma de los que puede hacer render para después leerlos de nuevo (en una textura, por ejemplo), a fin de mostrarlos en su aplicación. Los objetos del búfer de fotogramas representan un tema avanzado, por lo que no se van a estudiar con detalle; sin embargo, representan una característica muy eficaz que vale la pena investigar cuando domine el uso de OpenGL.

En el CD podrá encontrar una aplicación llamada *stencil testing*, donde se muestra una esfera reflejada sobre la superficie de un tablero de ajedrez. En la figura 12.1 se muestra una captura de pantalla de esta aplicación.

Figura 12.1
Esfera reflejada en la superficie de un tablero de ajedrez.



Resumen

El búfer de fotogramas se compone de cuatro búferes: de color, de profundidad, de plantilla y de acumulación. El búfer de acumulación se considera obsoleto y ya no debe utilizarse. De los tres búferes restantes, el más obvio es el búfer de color, que es el que finalmente aparece en pantalla. No obstante, los búferes de profundidad y de plantilla, contribuyen a lo que finalmente termina en el búfer de color.

Lo que se aprendió

- El búfer de fotogramas está formado por los búferes de color, de profundidad, de plantilla y de acumulación.
- `glClear()` establece todos los elementos de un búfer en un valor determinado. En el caso del búfer de color, éste es el color establecido mediante `glClearColor()`; en los otros búferes se establecen sus elementos en cero.
- Un cuadro de tijera define un área de la ventana que se verá afectada por los comandos de OpenGL. Si la prueba de tijera está habilitada, las demás partes de la ventana no estarán afectadas por los comandos de render.
- El búfer de profundidad se utiliza en la prueba de profundidad a fin de descartar fragmentos que se ocultan detrás de otros a los que se hizo render previamente.

- El búfer de plantilla permite controlar con precisión en qué partes de la pantalla se efectúa el render. Puede utilizarse para limitar el render de reflejos sobre superficies reflejantes.

Preguntas de repaso

1. ¿Qué función se usa para borrar un búfer de OpenGL?
2. ¿Qué hace el comando `glScissor()`?
3. ¿Cómo se desactiva la escritura en el búfer de profundidad?
4. ¿Cómo puede habilitarse la prueba de plantilla?

Por su cuenta

1. Usando como base el pseudocódigo de reflejo con plantilla, escriba una aplicación que efectúe el render de un cubo giratorio que se refleje en un espejo cuadrado sobre el suelo.

El juego final

Ahora es el momento de reunir toda la información sobre OpenGL que se ha cubierto hasta ahora para crear un juego llamado *Ogro Invasion!* En este capítulo, se describirá brevemente el diseño del juego y cualquier conocimiento adicional necesario para comprender el código.

De manera específica, se estudiará:

- Cómo hacer la carga y la animación de modelos MD2
- Partículas y puntos sprite
- Una visión general del diseño de juegos

El formato del modelo MD2

El almacenamiento de datos de los vértices del modelo en un archivo es mucho más flexible que guardarlos estáticamente en el código fuente. Los modelos pueden crearse en una aplicación de modelado 3D y guardarse en un archivo que se puede leer en la aplicación. Hay muchos formatos de modelo diferentes, pero el que se analiza aquí es el formato MD2, que es un formato de modelo animado muy popular entre los desarrolladores de juegos amateur, ya que es fácil de examinar y animar, muchas herramientas de modelado libre son compatibles con él y existe una gran cantidad de modelos gratuitos disponibles en internet.

El modelo de formato MD2 fue creado originalmente para almacenar los caracteres animados para el juego *Quake 2* de iD Software. Los archivos MD2 almacenan modelos animados como una serie de *fotogramas clave*. Cada fotograma clave almacena los datos de vértice de un solo fotograma de animación. Los fotogramas clave se organizan en el archivo en diferentes secuencias de animación; por ejemplo, los fotogramas del 40 al 45 definen la animación de un personaje corriendo, y los fotogramas del 46 al 53 definen la animación de un personaje al ataque. Los vértices de estos fotogramas clave se interpolan linealmente en función del tiempo de animación para crear marcos dinámicos entre los fotogramas clave, lo que crea una animación continua. El formato MD2 está dividido lógicamente en dos secciones distintas. La primera de ellas es el encabezado del archivo, el cual almacena información necesaria para leer el resto del archivo que contiene los datos del fotograma clave.

El encabezado de MD2

El encabezado de MD2 se puede leer en una estructura simple que tiene la siguiente definición:

```
 struct MD2Header {
    char ident[4];        //Debe ser igual a “IDP2”
    int versión;          //versión de MD2
    int skinWidth;        //ancho de la textura
    int skinHeight;       //altura de la textura
    int frameSize;        //tamaño de un fotograma en bytes
    int numSkins;         //número de texturas
    int numVertices;      //número de vértices
    int numTextureCoords; //número de coordenadas de textura
    int numTriangles;     //número de triángulos
    int numGLCmds;        //número de comandos de Opengl
    int numFrames;        //número total de fotogramas
    int skinOffset;       //desplazamiento a los nombres skin (64 bytes cada uno)
    int texCoordOffset;   //desplazamiento a las coordenadas de textura s-t
    int triangleOffset;   //desplazamiento a los triángulos
    int frameOffset;      //desplazamiento a los datos de fotograma
    int GLCmdOffset;      //desplazamiento a los comandos de OpenGL
    int eofOffset;        //desplazamiento al final del archivo
};
```

`ident` es una cadena de cuatro caracteres que identifica este archivo como un modelo MD2. El número `version` debería ser siempre igual a 8. `skinWidth` y `skinHeight` son las dimensiones del mapa o los mapas de textura del modelo. `frameSize` es el tamaño de cada fotograma en bytes.

Figura 13.1
Distribución del archivo MD2.

`numVertices`, `numTexCoords` y `numTriangles` se explican por sí mismas. Sin embargo, cabe señalar que `numVertices` es el número de vértices por fotograma, mientras que las coordenadas de los triángulos y las texturas son compartidas por todos los fotogramas. `numGLCmds` necesita una explicación un poco más larga, pues la última parte de un archivo MD2 almacena los datos índice del triángulo como una lista de abanicos hechos con triángulos y tiras de triángulos. Al hacer el render con el modo inmediato, el uso de la lista de comandos GL para el render del modelo es mucho más rápido que la representación sólo con triángulos. Sin embargo, cuando se usan VBO, es mucho más fácil y probablemente más eficiente enviar todos los triángulos indexados al mismo tiempo, en lugar de utilizar la lista de comandos GL para diferenciar entre las tiras de un triángulo y los abanicos de triángulos. `numGLCmds` es el número de comandos en la lista de comandos. `numFrames` es el número de fotogramas clave en el modelo. El resto del encabezado se compone de los desplazamientos. Cada desplazamiento representa el lugar en el archivo donde se encuentra cada conjunto de datos. En el cargador de MD2 del disco, estos desplazamientos se usan para trasladarse a la ubicación correcta en el archivo a fin de leer los datos. En la figura 13.1 se muestra la distribución del archivo en el modelo MD2.

Carga de los datos del modelo

El encabezado proporciona información suficiente para asignar memoria al almacenamiento de datos del modelo y a los desplazamientos necesarios para navegar por el archivo. El resto de

los datos del modelo se almacena en forma de bloques secuenciales. Cada uno de ellos almacena un determinado tipo de datos. El primer conjunto de datos en el archivo es la información skin (skin es una textura de modelo). Un modelo MD2 puede tener varias skins diferentes asociadas a él. Los datos de cada skin pueden leerse en la siguiente estructura:

```
struct Skin {
    char name[64]; //nombre del archivo de textura
};
```

Cada skin almacena la ruta del archivo de la textura. La ruta por lo general está relacionada con la estructura del directorio de *Quake 2*, de manera que para que pueda usarlo para sus propios juegos, quizás desee eliminar la ruta del directorio y dejar el nombre del archivo. Aunque nuestro modelo simple de manejo cargará los datos de la skin, el modelo de render sólo usa la textura enlazada actualmente.

Después de los datos de la skin está la lista de las coordenadas de textura. Las coordenadas de textura en MD2 se almacenan como `shorts` en lugar de `floats`:

```
struct TexCoord {
  short s;
  short t;
};
```

Antes de poder usar las texturas en nuestras aplicaciones, es necesario manipularlas un poco. En primer lugar, deben escalarse a un valor entre 0.0 y 1.0 mediante la división de las componentes `s` y `t` entre `skinWidth` y `skinHeight`, respectivamente. En segundo lugar, la coordenada t debe partirse de manera que se convierta en 1.0 –`t`. El siguiente código convierte las coordenadas de textura a una forma que puede manejarse:

```
float s = (float(md2TexCoords[i].s) / (float)header.skinWidth);
float t = 1.0f - (float(md2TexCoords[i].t) / (float)header.skinHeight);
```

donde `i` es un índice hacia la coordenada de textura actual, en el arreglo de coordenadas de textura.

Después de las coordenadas de textura están los datos de triángulo. Cada triángulo almacena tres índices en los vértices del fotograma clave y tres índices en las coordenadas de la textura que se cargaron previamente. La estructura de triángulo en MD2 es la siguiente:

```
struct Triangle {
    short vertexIndex[3];
    short texCoordIndex[3];
};
```

El siguiente paso consiste en cargar los datos reales del fotograma clave. Cada fotograma clave tiene la siguiente estructura:

```
struct KeyFrame {
    float scale[3];
    float translate[3];
    char name[16];
    Vertex vertices[numVertices];
};
```

El campo `name` se usa durante la creación del modelo; es tan sólo un identificador único para el fotograma clave. Los vértices en cada fotograma clave se almacenan en un formato comprimido. Para descomprimirlos es necesario multiplicar cada vértice por el factor `scale` y luego sumar el vector `translate`. La estructura del vértice en MD2 tiene la siguiente definición:

```
struct Vertex {
    unsigned char v[3];
    unsigned char lightNormalIndex;
};
```

Los componentes del vértice x, y, z se almacenan como un arreglo de caracteres. El índice `light-NormalIndex` es específico para *Quake 2* y se encuentra en un arreglo de normales de vértice. Un último aspecto a destacar acerca de los vértices es que en *Quake 2* los ejes z y y están intercambiados, por lo que al cargar el modelo MD2 deben cambiarse estas coordenadas de forma que el modelo tenga la orientación correcta. El siguiente código descomprime un vértice de un fotograma clave donde `frame` es un iterador que apunta hacia el fotograma clave actual, y `k` es un índice para el vértice que se va a descomprimir:

```
x = ((*frame).scale[0] * (*frame).vertices[k].v[0] + (*frame).translate[0]);
z = ((*frame).scale[1] * (*frame).vertices[k].v[1] + (*frame).translate[1]);
y = ((*frame).scale[2] * (*frame).vertices[k].v[2] + (*frame).translate[2]);
```

Animación del modelo MD2

La animación del modelo MD2 es bastante sencilla. Los fotogramas clave en un modelo así están separados a intervalos de tiempo distintos. Si simplemente se hace el render de uno después de otro, la animación se verá discontinua y poco realista. Los vértices simplemente saltarán de una posición a la siguiente. Para que la animación sea continua, es necesario generar cuadros entre los fotogramas clave mediante *interpolación*. Para ayudarle a visualizar la interpolación, imagine un pedazo de papel con dos puntos negros separados por cierta distancia. Estos puntos se denominarán v1 y v2 y cada uno de ellos representa un fotograma clave de la animación. Si

usted dibuja una línea de v1 a v2, estará trazando la trayectoria de un punto interpolado. Dado un valor de tiempo entre 0.0 y 1.0, podrá observar que es posible generar un punto a lo largo de la línea. Por ejemplo, en el momento 0.0 el punto interpolado será el mismo que v1, y en el momento 1.0 el punto interpolado será igual a v2. Así que en el momento 0.5, el punto interpolado estará exactamente a medio camino entre v1 y v2. Esta interpolación lineal entre dos puntos puede lograrse con el siguiente cálculo:

$$v_i = v_1 + t \times (v_2 - v_1)$$

donde v_i es el vértice interpolado. v_1 y v_2 son los vértices actual y siguiente, respectivamente, y t es el tiempo de interpolación (entre 0.0 y 1.0).

En la clase del modelo MD2 se tiene un seguimiento de cinco variables importantes:

- `m_startFrame` —Un entero. El número del fotograma que es el inicio de la secuencia de animación actual.
- `m_endFrame` —Un entero. El último fotograma de la animación actual.
- `m_currentFrame` —Un entero. El fotograma clave actual en la animación.
- `m_nextFrame` —Un entero. Normalmente es `(m_startFrame + 1)`, a excepción del momento cuando el modelo llega al final de su secuencia de animación, en cuyo caso `m_nextFrame` es igual a `m_startFrame`.
- `m_interpolation` —Un flotante. Es un valor entre 0.0 y 1.0 y se utiliza para calcular un fotograma interpolado entre el fotograma clave actual y el siguiente. La interpolación se incrementa con el tiempo y, cuando llega a 1.0, los fotogramas clave actual y siguiente se incrementan, y `m_interpolation` se restablece en 0.0.

En este momento usted debe ser capaz de visualizar cómo funciona la animación. Para cualquier punto en el modelo, se calcula un nuevo vértice a cada fotograma. Mediante el uso del factor de interpolación, este punto se mueve gradualmente desde la posición del fotograma clave actual hasta la posición del siguiente fotograma clave. Cuando el factor de interpolación es 0.0, entonces el vértice generado es el mismo que el fotograma clave *actual*. Cuando el factor de interpolación llega a 1.0, el vértice es igual al del *siguiente* fotograma clave. Cuando el factor de interpolación es de 1.0, los fotogramas actual y siguiente se incrementan, el factor de interpolación se restablece en 0.0 y todo el proceso se repite de nuevo.

El siguiente código genera un fotograma clave interpolado de manera lineal:

```
float t = m_interpolation;
int i = 0;
for (vector<Vertex>::iterator vertex = m_interpolatedFrame.vertices.begin();
    vertex != m_interpolatedFrame.vertices.end(); ++vertex) {
```

```
    float x1 = m_keyFrames[m_currentFrame].vertices[i].x;
    float x2 = m_keyFrames[m_nextFrame].vertices[i].x;
    (*vertex).x = x1 + t * (x2 - x1); //Calcula el componente X interpolado

    float y1 = m_keyFrames[m_currentFrame].vertices[i].y;
    float y2 = m_keyFrames[m_nextFrame].vertices[i].y;
    (*vertex).y = y1 + t * (y2 - y1); //Calcula el componente Y interpolado

    float z1 = m_keyFrames[m_currentFrame].vertices[i].z;
    float z2 = m_keyFrames[m_nextFrame].vertices[i].z;
    (*vertex).z = z1 + t * (z2 - z1); //Calcula el componente Z interpolado
    ++i;
}
```

La lógica de aumentar el tiempo de interpolación y desplazarse a través de la animación es bastante simple. El siguiente código se encarga de calcular la interpolación y los índices del fotograma actual y siguiente:

```
//Aumenta la interpolación con base en el tiempo transcurrido y
la velocidad del render
m_interpolation += dt * FRAMES_PER_SECOND;
if (m_interpolation >= 1.0f)
{
  //Hace que el fotograma actual sea igual al siguiente
  m_currentFrame = m_nextFrame;
  m_nextFrame++; //Incrementa el fotograma siguiente
  //Si se acaba de pasar el final de la animación
  if (m_nextFrame > m_endFrame)
  {
    //Establece el fotograma siguiente como el fotograma de
    inicio y reinicia la interpolación
    m_nextFrame = m_startFrame;
    m_interpolation = 0.0f;
  }
}
```

Render del modelo

El fotograma interpolado se calcula mejor antes de hacer el render como un todo para que los vértices del modelo puedan ser enviados al render en su conjunto. El render de un fotograma *sin textura* del modelo es simple porque el archivo MD2 proporciona una lista indexada de triángulos

que puede pasarse a `glDrawElements()`. Sin embargo, el render de un fotograma con textura no es tan simple. Si recordamos la estructura `Triangle` en el archivo MD2, se puede observar que contiene tres índices en el arreglo de vértices y tres índices en el arreglo de coordenadas de textura. Esto presenta un problema porque `glDrawElements()` sólo toma un conjunto de índices y usará el índice de vértices también para leer las coordenadas de textura, lo que probablemente dará lugar a una aplicación errónea de las coordenadas de textura. Para solucionar este problema se puede renunciar al uso de los datos de triángulo en MD2 durante el render y emplearlos para duplicar algunas de las coordenadas de vértice y textura a fin de disponerlas en forma de triángulos secuenciales, de modo que `glDrawArrays()` pueda manejarlas. En el código fuente del juego incluido en el CD, hay un método que realiza esta tarea y se llama `reorganizeVertices()`; este método puede encontrarse en el archivo fuente del cargador del modelo MD2 (md2model.cpp).

Una vez que los vértices se han dispuesto adecuadamente, para hacer el render del fotograma interpolado sólo es cuestión de llamar a `glDrawArrays()`.

```
//Activa los atributos de las coordenadas de vértice y textura
glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);

//Enlaza el búfer de vértices para el render
glBindBuffer(GL_ARRAY_BUFFER, m_vertexBuffer);
glVertexAttribPointer((GLint)0, 3, GL_FLOAT, GL_FALSE, 0, 0);

//Enlaza el búfer de coordenadas de textura para el render
glBindBuffer(GL_ARRAY_BUFFER, m_texCoordBuffer);
glVertexAttribPointer((GLint)1, 2, GL_FLOAT, GL_FALSE, 0, 0);

//Dibuja los triángulos que conforman el fotograma interpolado
glDrawArrays(GL_TRIANGLES, 0, m_interpolatedFrame.vertices.size());

//Desactiva los arreglos de atributos de vértices
glDisableVertexAttribArray(1);
glDisableVertexAttribArray(0);
```

La información que se ha estudiado hasta ahora debe proporcionar un entendimiento básico de cómo se cargan y animan los modelos MD2. La mejor manera de entender el formato en su totalidad es estudiar el código fuente que acompaña a este texto.

Creación de explosiones

El juego (si es del tipo de disparar y matar) requiere algunas explosiones. Un efecto de explosión se crea mediante el uso de lo que se conoce como un *sistema de partículas*. Éste se utiliza

para todo tipo de efectos, como humo, fuego, agua, chispas o cualquier entidad que se componga de muchas partículas de materia. Un sistema de partículas puede contener varios cientos o miles de partículas, cada una con su propia posición, tamaño, color y velocidad. Estas partículas se desplazan y animan en cada fotograma antes del render.

El sistema de partículas de una explosión es muy simple. Todas las partículas que componen la explosión comienzan en un punto central, y a cada una se le da una velocidad aleatoria al momento de crear la explosión. Cada partícula tiene una variable de *vida* asociada a ella; esta variable se inicializa en 1.0 y disminuye cada fotograma hasta llegar a 0.0. En ese momento, la partícula se clasifica como muerta; una vez que todas las partículas están muertas, la explosión se acaba y su memoria puede liberarse. La estructura de la partícula se compone de cuatro variables:

```
struct particle
{
  float life;
  Vector3 position;
  Vector3 velocity;
  Color color;
};
```

Ya se ha analizado cómo el valor de `life` determina cuando la partícula muere, pero también se usa como el componente alfa del color de la partícula. Al combinarla con la mezcla, se crea un efecto de atenuación mientras la partícula se aleja del centro de la explosión. `position` es la ubicación de la partícula en coordenadas del mundo; ésta se configura inicialmente en el centro de la explosión y en cada fotograma se le añade `velocity` para que la partícula se desplace hacia afuera. El color se fija en rojo, naranja o amarillo, de nuevo al azar.

En el juego, la clase `Explosion` contiene un arreglo de partículas. En cada fotograma el código actualiza el arreglo de partículas mediante el siguiente código:

```
particle.life -= 1.0f * dT; //Disminuye la vida de la partícula
                 (con base en el tiempo)
particle.position += particle.velocity * dT; //Actualiza la
                  posición de la partícula
```

Después, el render de las partículas se hace mediante *puntos sprite*, que se analizarán en la siguiente sección. Para ver en detalle cómo se hace el render de las explosiones, vea los archivos fuente `explosion.cpp` y `explosion.h` en la demo del juego *Ogro Invasion!* incluido en el CD.

Puntos Sprite

A menudo las partículas se texturizan durante el render y luego se mezclan de modo que aparezcan con una forma adecuada. A pesar de que se podría hacer el render de las partículas

como puntos, éstos no se pueden texturizar de manera correcta (porque no es posible especificar coordenadas de textura en esquinas) y es probable que no luzcan realistas. La forma tradicional en la que se evita esto consiste en hacer el render de cada partícula como un cuadro. Esto consume muchos más recursos de lo que debería ser, pues no sólo implica enviar cuatro vértices por partícula (en vez de 1) a la tarjeta gráfica, sino que también cada cuadro debe colocarse de frente a la cámara para que el espectador no vea la partícula de lado, lo que arruinaría el efecto. Por fortuna, los proveedores de tarjetas gráficas desarrollaron una extensión (`GL_ARB_POINT_SPRITE`), que combina las ventajas del render de puntos con la capacidad de asignarles una textura adecuada. La extensión `GL_ARB_POINT_SPRITE` se subió a la base en OpenGL 2.1 y de la misma forma estará disponible en un contexto de OpenGL 3.0.

Uso de puntos Sprite

El render de puntos sprite se habilita y deshabilita al pasar `GL_POINT_SPRITE` a `glEnable()` o `glDisable()`, respectivamente. Mientras esté habilitado el render de puntos sprite, cualquier render que se realice usando `GL_POINTS` se representará como puntos sprite. Las coordenadas de textura de los puntos sprite se generan automáticamente. De forma predeterminada, se especifica una coordenada de textura única para todo el sprite, que sólo en contadas ocasiones será lo que se desea. La generación de la coordenada de textura tradicional para puntos sprite puede habilitarse para cada unidad de textura si se llama a la siguiente función:

```
glTexEnvi(GL_POINT_SPRITE, GL_COORD_REPLACE, GL_TRUE);
```

Después de haber llamado a esta función, es posible especificar coordenadas de textura para cada esquina del sprite, con (0.0, 0.0) en la parte superior izquierda y (1.0, 1.0) en la parte inferior derecha. El acceso a estas coordenadas en el fragment shader puede obtenerse mediante la variable integrada `gl_PointCoord`, la cual puede usarse en el fragment shader para muestrear una textura como cualquier otra coordenada de textura en 2D —al pasarla como segundo parámetro en la función `texture()`:

```
gl_FragColor = texture(texture0, gl_PointCoord);
```

En el ejemplo anterior, `texture0` es el uniforme del sampler. Aunque ahora el render de puntos sprite texturizados es posible, quizá sean demasiado pequeños como para poder verlos. Para corregir esta situación, es necesario establecer el tamaño de los puntos, lo cual puede hacerse de dos formas. El primer método consiste simplemente en llamar a `glPointSize()` con el valor que se requiere. Si se utiliza este método con un vertex shader activado, no se llevará a cabo ninguna atenuación con la distancia (es decir, el tamaño del punto no se reducirá aunque esté más lejos de la cámara). El enfoque alternativo consiste en establecer la variable integrada `gl_PointSize` en el vertex shader, en donde puede calcular una distancia de atenuación propia puesto que tiene acceso a la posición del vértice. De forma predeterminada, en el vertex shader

no podrá establecer el tamaño del punto; para que esto sea posible es necesario llamar a la siguiente función:

```
glEnable (GL_VERTEX_PROGRAM_POINT_SIZE);
```

Esto le indicará a OpenGL que debe usar el valor establecido en el vertex shader y no el fijado por `glPointSize()`. Se puede regresar al comportamiento predeterminado pasando el mismo valor a `glDisable()`. Para ver un ejemplo de puntos sprite en acción, busque `explosion.cpp`, `particle.vert` y shaders `particle.frag` en los archivos fuente, que forman parte del código fuente del juego.

Ogro Invasion!

Ahora es el momento de analizar el diseño general del mini-juego incluido en el CD. El objetivo de *Ogro Invasion!* es detener a una horda de monstruos (llamados Ogros) que surgen continuamente invadiendo una isla idílica. El jugador tiene cinco minutos para matar a tantos monstruos como pueda. El juego ha sido diseñado para ampliarse, de hecho, hay muchas características que no están incluidas y que, obviamente, se dejan como ejercicio para que el lector las desarrolle (vea la sección "por su cuenta" donde se dan algunas sugerencias).

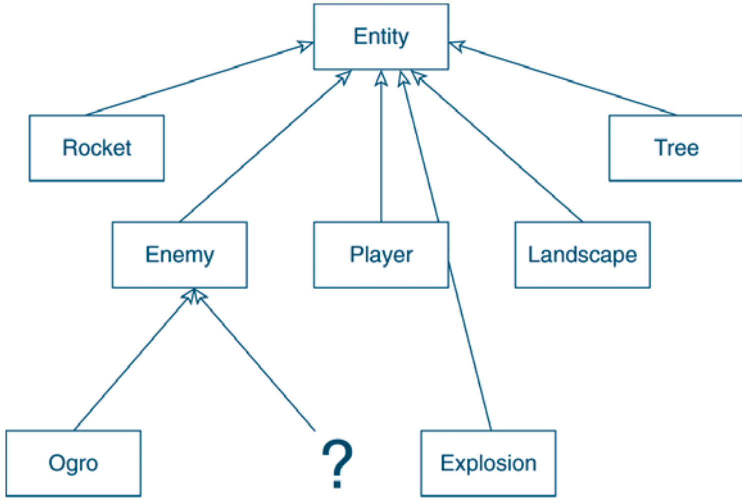
La mayoría de las características que componen el juego se han visto en capítulos anteriores. Se reutiliza la fiel representación de un terreno, los árboles procesados mediante pruebas alfa (capítulo 9), el frustum culling (capítulo 10) y las fuentes FreeType (capítulo 11). Los elementos restantes necesarios para completar el juego eran las explosiones y el modelo de render. Lo último que debe entenderse es el diseño general del juego. Todos los elementos visibles (sin incluir las fuentes) se clasifican como una entidad y representan una subclase de una clase madre llamada `Entity` que tiene la siguiente definición:

```
class Entity : private Uncopyable {
  private:
    virtual void onPrepare(float dt) = 0;
    virtual void onRender() const = 0;
    virtual void onPostRender() = 0;
    virtual bool onInitialize() = 0;
    virtual void onShutdown() = 0;
    virtual void onCollision(Entity* collider) = 0;

    bool m_canBeRemoved;

    GameWorld* m_world;
  public:
    Entity(GameWorld* const gameWorld);
```

```
    virtual ~Entity();

    void prepare(float dt);
    void render() const;
    void postRender();
    bool initialize();
    void shutdown();
    bool canBeRemoved() const;
    void destroy();

    void collide(Entity* collider);

    virtual Vector3 getPosition() const = 0;
    virtual void setPosition(const Vector3& position) = 0;
    virtual float getYaw() const = 0;
    virtual float getPitch() const = 0;
    virtual void setYaw(const float yaw) = 0;
    virtual void setPitch(const float pitch) = 0;
    virtual Collider* getCollider() = 0;
    virtual EntityType getType() const = 0;

    GameWorld* getWorld() {
      return m_world;
    }
};
```

Los métodos importantes que deben tenerse en cuenta son las funciones virtuales puras (las definiciones de métodos que terminan en = `0` ;). Si no está familiarizado con ellas, podemos describirlas brevemente como los métodos que *deben* ser anulados por una subclase. Por ejemplo, una de las subclases de `Entity` es la de `Player`. La clase `Player` implementa todas las funciones virtuales. En la figura 13.2 se muestra la jerarquía de clases del juego. Puede observarse que incluso el terreno es una `Entity` llamada `Landscape`. La función `getType()` puede usarse en cualquier puntero de `Entity` para determinar el tipo de entidad de que se trata.

Figura 13.2
Jerarquía de entidades en *Ogro Invasion*!



Todas las entidades del juego se almacenan en una lista única que es manejada por la clase `GameWorld`. Esta clase se encarga de generar nuevas entidades, dar seguimiento al número de enemigos, eliminar entidades muertas, etcétera. La lista de entidades se itera en cuadro y se realizan las siguientes acciones:

1. **Se preparan las entidades.** Cada función `Entity` `onPrepare()` se llama al sitio donde se realizan las actualizaciones basadas en el tiempo. Aquí es donde ocurre la animación del modelo de los monstruos y se calcula el movimiento de la entidad.
2. **Se realiza la detección de colisiones.** La mayor parte de la detección de colisiones se ejecuta mediante simples esferas limítrofes. Si la distancia entre dos entidades es menor que el radio de su ámbito de delimitación, entonces ocurre una colisión.
3. **Se eliminan las entidades muertas.** Cuando una entidad ya no está en uso (por ejemplo, un cohete que ha sido destruido) se establece su bandera `canBeRemoved`. Estas entidades muertas se eliminan antes del render.
4. **Se calcula el movimiento del ratón.** En cada fotograma se registra la posición del ratón y luego el cursor se mueve de nuevo al centro de la pantalla. Esto significa que en cada fotograma es posible encontrar la distancia que se trasladó el ratón desde el último cuadro, lo que puede usarse para actualizar la cámara.
5. **Se hace el render de las entidades.** Esto implica llamar al comando `OnRender()` y luego a `onPostRender()`. Estas funciones sólo se llaman si el ámbito de delimitación de la entidad es visible (frustum culling).

Una nota sobre la detección de colisiones

La detección de colisiones es un tema extenso y complejo sobre el cual puede haber (y hay) libros enteros. Por esta razón, la detección de colisiones que se utiliza en *Ogro Invasion*!, es muy simplista. A todas las entidades se les da un radio que define su ámbito de delimitación. En cada fotograma, todas las entidades se comparan contra las demás a fin de encontrar la distancia entre ellas. Si la distancia es menor que el radio combinado de las dos entidades, entonces se presenta una colisión. Si ocurre una colisión, ambas entidades llaman a su función `onCollision()`; el único parámetro de esta función es la entidad contra la que chocó. Lo anterior permite una lógica condicional, dependiendo del tipo de entidad que fue golpeada (por ejemplo, si un monstruo choca con un cohete, entonces el monstruo debe ser matado en su llamada a `onCollision()`). La única vez que no se usa la colisión basada en esferas es cuando una entidad choca con el terreno `Landscape`; en este caso, la entidad tiene su posición en el eje y modificada, de manera que se sitúe sobre el terreno.

Figura 13.3

Captura de pantalla del juego *Ogro Invasion!*

Resumen

Eso es todo. Explore el código y extiéndalo, agregue nuevas funciones, nuevos chicos malos y mejores gráficos. Si no, ¡simplemente trate una y otra vez de mejorar su puntuación más alta! ¡En la figura 13.3 se muestra el juego terminado!

Preguntas de repaso

No hay preguntas de repaso para este capítulo.

Por su cuenta

1. El juego tiene un cielo muy llano; aplique un skybox para ambientar el juego de mejor manera.
2. La jerarquía de clases está diseñada para permitir más de un tipo de enemigo. Usando la clase `Ogro` como base, añada un nuevo enemigo más inteligente, ¡que se defienda!
3. Escriba un código para guardar la puntuación más alta en un archivo de texto y modifique la pantalla de Game Over para mostrar si éste fue superado.

Respuestas a las preguntas de repaso y ejercicios

Capítulo 1
Preguntas de repaso

1. 1992.
2. OpenGL 3.O.
3. El grupo de trabajo de OpenGL, que forma parte del Grupo Khronos.

Por su cuenta

1. Cambie el código de dibujo por lo siguiente:

```
glColor4f(0.0, 0.0, 1.0, 1.0);
glBegin(GL_TRIANGLES);
  glVertex3f(2.0, 2.5, -1.0);
  glVertex3f(-3.5, -2.5, -1.0);
  glVertex3f(2.0, -4.0, -1.0);
glEnd();

glColor4f(1.0, 0.0, 0.0, 1.0);
glBegin(GL_TRIANGLE_FAN);
  glVertex3f(-1.0, 2.0, 0.0);
```

```
  glVertex3f(-3.0, -0.5, 0.0);
  glVertex3f(-1.5, -3.0, 0.0);
  glVertex3f(1.0, -2.0, 0.0);
  glVertex3f(1.0, 1.0, 0.0);
glEnd();
```

2. Las respuestas pueden variar.

Capítulo 2
Preguntas de repaso

1. El contexto de render conecta a OpenGL con una ventana.
2. En Windows, `wglGetCurrentContext()`.
3. Una estructura que define los atributos del contexto de OpenGL.
4. Establece el color con el que se limpia el búfer de color al llamar a `glClear()`.
5. La estructura `DEVMODE`.

Por su cuenta

1. Las respuestas pueden variar. En `example.cpp`, cambie el método `init()` para que contenga la línea `glClearColor(1.0f, 1.0f, 1.0f, 1.0f);`, la cual cambiará el color de fondo a blanco. También agregue lo siguiente a `render()` después dibujar el triángulo actual:

```
glColor4f(1.0f, 0.0f, 0.0f, 1.0f);
glBegin(GL_TRIANGLES);
  glVertex3f(2.0f, 2.5f, -1.0f);
  glVertex3f(-3.5f, -2.5f, -1.0f);
  glVertex3f(2.0f, -4.0f, -1.0f);
glEnd();
```

Capítulo 3
Preguntas de repaso

1. Al llamar a `glEnable(GL_CULL_FACE);`
2. `glGetString(GL_VERSION);`
3. En sentido contrario a las manecillas del reloj.
4. `GL_POLYGON_SMOOTH`

Por su cuenta

1. Si se supone que `m_vertices` y `m_colors` son vectores, entonces los datos de la pirámide pueden especificarse en OpenGL como:

```
//Crea las partes exteriores de la pirámide
m_vertices.push_back(Vertex(0.0,2.0,0.0)); //cima de la pirámide
m_vertices.push_back(Vertex(-1.0,-1.0,-1.0)); //sus 4 bordes
m_vertices.push_back(Vertex(1.0,-1.0,-1.0));
m_vertices.push_back(Vertex(1.0,-1.0,1.0));
m_vertices.push_back(Vertex(-1.0,-1.0,1.0));
m_vertices.push_back(Vertex(-1.0,-1.0,-1.0));

m_vertices.push_back(Vertex(1.0,-1.0,-1.0)); //La parte inferior
m_vertices.push_back(Vertex(1.0,-1.0,1.0));
m_vertices.push_back(Vertex(-1.0,-1.0,-1.0));
m_vertices.push_back(Vertex(-1.0,-1.0,1.0));

m_colors.push_back(Color(1.0,1.0,1.0)); //color de la parte superior
m_colors.push_back(Color(1.0,0.0,1.0));
m_colors.push_back(Color(1.0,1.0,0.0));
m_colors.push_back(Color(0.0,1.0,1.0));
m_colors.push_back(Color(0.0,0.0,1.0));
m_colors.push_back(Color(1.0,0.0,1.0));
m_colors.push_back(Color(1.0,1.0,0.0));
m_colors.push_back(Color(0.0,1.0,1.0));
m_colors.push_back(Color(1.0,0.0,1.0));
m_colors.push_back(Color(0.0,0.0,1.0));

glGenBuffers(1, &m_vertexVBO);
glBindBuffer(GL_ARRAY_BUFFER, m_vertexVBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat) * m_vertices.size() * 3,
&m_vertices[0], GL_STATIC_DRAW);

glGenBuffers(1, &m_colorVBO);
glBindBuffer(GL_ARRAY_BUFFER, m_colorVBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat) * m_colors.size() * 3,
&m_colors[0], GL_STATIC_DRAW);
```

Entonces es posible hacer dibujar la pirámide usando:

```
//Hace el render del abanico de triángulos para la cubierta
exterior de la pirámide
glDrawArrays(GL_TRIANGLE_FAN, 0, 6);
//Efectúa el render de la base de la pirámide -> usa los últimos
4 vértices
glDrawArrays(GL_TRIANGLE_STRIP, 6, 4);
```

Capítulo 4
Preguntas de repaso

1. `glPushMatrix()`
2. `glTranslatef(10.0f, 5.0f, 0.0f);`
3. Cualquier combinación entre las tres opciones siguientes: Proyección (`GL_PROJECTION`), Modelview (`GL_MODELVIEW`), o Textura (`GL_TEXTURE`) y Color (`GL_COLOR`).
4. `glScalef (0.5f, 0.5f, 0.5f);`
5. `glRotatef()`
6. `glLoadMatrix()`
7. `glPopMatrix()`

Por su cuenta

1. Las respuestas pueden variar. Para este ejercicio se puede adaptar el código para dibujar la pirámide del capítulo anterior.

Capítulo 5
Preguntas de repaso

1. `glGetStringi()` devuelve una lista de extensiones compatibles y el primer argumento debe ser `GL_EXTENSIONS`.
2. Un prefijo `ARB` significa que la extensión ha sido aprobada por la Architecture Review Board (Comisión para la revisión de la arquitectura).
3. OpenGL 3.O.
4. Para proporcionar las funcionalidades de vanguardia y facilitar el acceso a las características compatibles con un controlador, pero que aún no se incorporan a las bibliotecas incluidas con el compilador.

Por su cuenta

1. Las respuestas pueden variar. La siguiente función recupera las extensiones y las escribe a

un archivo en OpenGL 3.O:

```
void writeOutExtensions(const string& filename)
{
  //Se toma un puntero hacia la función glGetStringi
  PFNGLGETSTRINGIPROC glGetStringi = NULL;
  glGetStringi = (PFNGLGETSTRINGIPROC)wglGetProcAddress("glGetStringi");

  std::ofstream fileOut(filename.c_str()); //Crea un archivo
  de salida
  GLint numExtensions = 0;
  glGetIntegerv(GL_NUM_EXTENSIONS, &numExtensions); //Obtiene
  el conteo de extensiones

  for (int i = 0; i < numExtensions; ++i)
  { //Imprime cada extensión como una nueva línea en el archivo
    fileOut << glGetStringi(GL_EXTENSIONS, i) << std::endl;
  }
  fileOut.close(); //Cierra el archivo
}
```

Capítulo 6
Preguntas de repaso

1. El Shading Language de OpenGL
2. Un shader es una serie de instrucciones en código que sustituye a una determinada fase del diagrama de render. Por ejemplo, un shader puede realizar la lógica para el procesamiento de vértices.
3. Con la directiva de preprocesador `#version` (por ejemplo, `#version 130`).
4. Una variable que almacena datos pasados desde la aplicación, los cuales no varían de vértice a vértice.
5. Los atributos se especifican para cada vértice, mientras que los uniformes se pueden especificar a través de las primitivas.
6. `glAttachShader()`
7. `glLinkProgram()`

Por su cuenta

1. En el fragment shader, cambie la única línea dentro de la función `main()` a:

```
outColor = vec4 (1.0, 0.0, 0.0, 1.0);
```

Capítulo 7
Preguntas de repaso

1. Un objeto de textura es una estructura que contiene los atributos relacionados con una textura específica: el estado de la textura.
2. 128 x 128, 64 x 64, 32 x 32, 16 x 16, 8 x 8, 4 x 4, 2 x 2, 1 x 1
3. `GL_REPEAT`

Por su cuenta

1. Las respuestas pueden variar.

```
  GLuint textureObject;
  glGenTextures(1, &textureObject);
  glBindTexture(GL_TEXTURE_2D, textureObject);
  gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGBA, 256, 256, GL_RGBA, GL_UNSIGNED_
BYTE, imageData);
```

Capítulo 8
Preguntas de repaso

1. La luz que se ha reflejado tanto en las superficies que la fuente luminosa ya no es evidente.
2. Las luces puntuales tienen una posición en vez de una dirección; asimismo, en las luces puntuales también se aplica la atenuación a causa de la distancia.
3. Aumenta el tamaño y la intensidad del reflejo especular.

Por su cuenta

1. Las respuestas pueden variar.
2. Las respuestas pueden variar.

Capítulo 9
Preguntas de repaso

1. La palabra clave `discard` detiene el procesamiento del fragmento actual; no se hará ningún render en el búfer de fotogramas.
2. El mapeo de esferas sólo es eficaz cuando se observa desde cierta dirección. Además, el mapa de esferas debe crearse con anticipación, por lo que no reflejará una escena dinámica.
3. `glActiveTexture()`

Por su cuenta

1. Las respuestas pueden variar.

Capítulo 10

Preguntas de repaso

1. a = mvp[3] + mvp[O];
   b = mvp[7] + mvp[4];
   c = mvp[11] + mvp[8];
   d = mvp[15] + mvp[12];

2. Divide los cuatro componentes (a, b, c y d) entre la longitud de la normal.
3. ax + by + cz + d = O

Por su cuenta

1. Ésta es una tarea de investigación.

Capítulo 11

Preguntas de repaso

1. Las fuentes asignadas por textura sufren los efectos del aliasing y tienen una presentación pobre debido a que el render de cada carácter se hace con el mismo ancho.
2. La biblioteca FreeType proporciona acceso programático a la información de fuentes, la cual puede cargarse desde diferentes tipos de archivo de fuentes (por ejemplo, desde las fuentes TrueType).
3. El tamaño de fuente se establece mediante la función `FT_Set_Char_Size()`.

Por su cuenta

1. Las respuestas pueden variar.
2. Las respuestas pueden variar.

Capítulo 12

Preguntas de repaso

1. `glClear()`

2. `glScissor()` define una región de la pantalla que puede actualizarse mediante comandos de render, cuando están habilitadas las pruebas de tijera.
3. `glDepthMask(GL_FALSE);`
4. `glEnable(GL_STENCIL_TEST);`

Por su cuenta

1. Las respuestas pueden variar.

Capítulo 13

Preguntas de repaso

1. No hay preguntas de repaso para este capítulo.

Por su cuenta

1. Las respuestas pueden variar.
2. Las respuestas pueden variar.
3. Las respuestas pueden variar.

Lecturas adicionales

En este libro se ha cubierto una gran cantidad de información sobre gráficos por computadora con OpenGL. Sin embargo, el desarrollo de juegos es un campo muy complicado y diverso, por lo que para ser un programador más eficaz y rápido es necesario que cuente con la mayor cantidad posible de recursos. En este apéndice se proporciona una lista con estupendos sitios web y libros, a los que vale la pena echar un vistazo.

Recursos en la Red
OpenGL y desarrollo de juegos

GameDev.net

http://www.gamedev.net/ Los autores originales de este libro participaron en la fundación de GameDev.net. Es el principal recurso en línea para el desarrollo de juegos. GameDev.net cuenta con una comunidad activa con foros y revistas para desarrolladores, además de miles de artículos y noticias actualizadas de la industria de los juegos.

OpenGL

http://www.opengl.org/ El sitio de OpenGL. Contiene un foro activo, noticias constantes sobre OpenGL y, por supuesto, todas las especificaciones, archivos de encabezado e información de las extensiones que requiere para desarrollar aplicaciones en OpenGL.

Flipcode

http://www.flipcode.com/archives/ En su apogeo, Flipcode fue un sitio grande y popular para el desarrollo de juegos, con muchos artículos y tutoriales excelentes. Por desgracia, el sitio web ha sido cerrado y en la actualidad no recibe actualizaciones. Sin embargo, los archivos de los artículos aún permanecen en línea y contienen algunas joyas de conocimiento.

NeHe

http://nehe.gamedev.net/ Por largo tiempo, NeHe ha sido el sitio favorito de los principiantes en OpenGL. Después de haber sido fundado y administrado durante varios años de manera dedicada y desinteresada por Jeff Molofee, el sitio fue adquirido por GameDev.net y ahora recibe el mantenimiento de Luke Benstead y Haubold Carsten. En la actualidad experimenta una remodelación con nuevos tutoriales basados en OpenGL 3.0 y noticias que se actualizan de forma constante.

Ozone3D

http://www.ozone3d.net/ Ozone3D es el sitio donde se cuenta con un excelente conjunto de tutoriales para GLSL que cubren hasta GLSL 1.20.

Lighthouse3D

http://www.lighthouse3d.com/ Lighthouse3D contiene una serie de tutoriales para OpenGL en varios temas incluyendo GLSL, billboarding y selección de objetos.

Programación en C++

CPlusPlus.com

http://www.cplusplus.com/ El mejor sitio de referencia en línea para C++. El sitio contiene información sobre bibliotecas y funciones estándar de C y C++, así como información sobre los contenedores y algoritmos en STL.

Boost.org

http://www.boost.org/ Después de haber utilizado las bibliotecas estándar de C++, su próxima escala debe ser en Boost. Consiste en un conjunto de bibliotecas portátiles revisadas, que están diseñadas para trabajar de manera eficiente con las bibliotecas estándar de C++. Además, las bibliotecas de Boost son la base para la mayoría de las nuevas características incluidas en el próximo estándar de C++.

Herramientas y bibliotecas

Code:: Blocks

http://www.codeblocks.org/ Code:: Blocks es una IDE libre, de fuente abierta, publicada bajo la licencia GPL. Los ejemplos de Linux en el CD fueron compilados con esta IDE y tienen archivos de proyecto asociados.

Microsoft Visual C++ Express Edition

http://www.microsoft.com/express/windows/ En esta dirección URL está disponible la edición gratuita de Microsoft para la famosa IDE Visual C++ y su compilador. Si está desarrollando para Windows, debe tener en cuenta este sitio.

SDL

http://www.libsdl.org/ Como se mencionó en el primer capítulo, SDL es una biblioteca multiplataforma, de código abierto, que puede simplificar la creación de aplicaciones multiplataforma con OpenGL. SDL se encarga de la creación de ventanas, el manejo de eventos, la continuidad, la entrada del joystick, y el audio. Si desea que sus aplicaciones funcionen en más de una plataforma, definitivamente vale la pena revisar este sitio. Al momento de escribir este libro, SDL 1.2 era la versión más reciente, mientras que SDL 1.3 se encontraba en desarrollo. SDL 1.3 será compatible con la creación de contextos en OpenGL 3.0.

FreeType

http://www.freetype.org/ FreeType ya se presentó en el capítulo dedicado a las fuentes. En este sitio web, siempre puede encontrarse la versión más reciente.

Libros

Programación en C++

A first book of C++ 4th Edition

Gary Bronson, Cengage 2012

C++ Programming from Problem Analysis to Program Design, 5th Edition

D.S. Malik, Cengage 2011

C++ for engineers and scientists 3rd Edition
Gary Bronson, Cengage 2012

Effective C++ Third Edition
Scott Meyers, Addison-Wesley, 2005

Effective STL
Scott Meyers, Addison-Wesley, 2006

C++ Coding Standards
Herb Sutter y Andrei Alexandrescu, Addison-Wesley, 2005

OpenGL y Matemáticas 3D

More OpenGL Game Programming
Dave Astle, Cengage, 2006

Física para videojuegos
Erleben, Cengage 2011

Mathematics for 3D Game Programming & Computer Graphics
Eric Lengyel, Charles River Media, 2004

OpenGL Shading Language, Second Edition
Randi J. Rost, Addison Wesley, 2006

Apéndice C

Contenido del CD

El CD que acompaña a este libro incluye muchos recursos que pueden ser utilizados junto con el texto.

Código fuente

Por supuesto, el elemento más importante en el CD ¡es el código fuente! El CD contiene dos carpetas de código fuente: una con el código actualizado en OpenGL 3.0 y otra con las versiones de las mismas aplicaciones en OpenGL 2.1. Dentro de cada una de estas carpetas, las aplicaciones están organizadas por capítulos.

Cuando navegue por el código fuente, considere lo siguiente:

- Las aplicaciones correspondientes a los capítulos anteriores al 5 son idénticas para ambas versiones de OpenGL.
- Las diferencias entre las dos versiones del código son menores. La discrepancia más común se presenta en el código de GLSL, que está escrito en GLSL 1.20 para aquellas tarjetas que aún no son compatibles con GLSL 1.30. El único capítulo en el que también hay diferencia en el código fuente de C++ es el capítulo 5, donde se usa `glGetStringi()` para la versión en OpenGL 3.0 y `glGetString()` para la versión en OpenGL 2.1.
- Los archivos de proyecto en Visual C++ son para Windows, mientras que los archivos en Code::Blocks son para Linux.
- El juego del capítulo 13 puede bajarse a OpenGL 2.1 si OpenGL 3.0 no está disponible.

GLee

En el CD se incluye la biblioteca de extensión GLee para OpenGL. La versión incluida es compatible con OpenGL 3.0. La aplicación más reciente de Glee puede encontrarse en el sitio web de Ben Woodhouse: http://elf-stone.com/glee.php.

SDL 1.2

En el CD también se incluye la versión estable actual de la biblioteca SDL. Aunque no se incluye la posibilidad de crear un contexto en OpenGL 3.0 (lo cual está programado para añadirse en la versión 1.3), sigue siendo una biblioteca muy poderosa para crear juegos multiplataforma.

FreeType

FreeType es una biblioteca de fuentes de código abierto sorprendentemente potente. Como se vio en el capítulo 11, la capacidad de cargar fuentes de manera directa desde su archivo es realmente útil. En el CD se incluye la versión más reciente de FreeType 2. Las bibliotecas más recientes de FreeType pueden encontrarse en http://www.freetype.org/.

Code::Blocks

En el CD se incluye la IDE de código abierto Code:: Blocks con los instaladores para los siguientes sistemas operativos:

- Windows (con el compilador minGW)
- Ubuntu Linux
- Mac OSX

Al momento de escribir este libro, la última versión estable de Code:: Blocks era la 10.05. Para obtener la versión más reciente, consulte el sitio web de Code:: Blocks en http://www.codeblocks.org/.

ÍNDICE

3DLabs, 7

A

abanico de triángulos, 13
actualizaciones basadas en el tiempo, 40
Alexandrescu, Andrei, 280
ancho de líneas, 65-66
animación, modelo MD2, 255-257
antialiasing
líneas, 66
puntos, 63-64
triángulos, 70
arquitectura, 5-6, 8
arreglos de textura, 159
Astle, Dave, 280
atenuación, 187
ATI, 7
atributos de vértice, 147-148

B

banderas
formato de píxel, 24
vertex arrays, 52
biblioteca GLU, 173-174
biblioteca Kazmath, 152
bibliotecas, 10-11, 17
GLUT (OpenGL Utility Toolkit), 10-11
SDL (Simple Direct Medial Layer), 11
Boost, 279
brillantez, iluminación, 186
búfer de acumulación, 124
búfer de color, 208-209, 241-242
búfer de fotogramas, 239, 248
búfer de plantilla, 245-248
búfer de profundidad, 242-243
búfer de selección, 124
búfers, 239-249
de fotogramas, 239, 248

de color, 208-209, 241-242
de acumulación, 124
de profundidad, 242-243
enmascaramiento de color, 241-242
de limpieza, 240
llenado de, 60-61
macro BUFFER_OFFSET, 62
de plantilla, 245-248
pruebas de tijera, 240-241
selección, 124
superficies traslúcidas, 244-245
valores de frecuencia, 61
vinculación de, 60
z-fighting, 244

C

cadena de extensión unificada, 125
cadenas de nombre, extensiones, 112-113
cálculos de iluminación, 127
calificadores, GLSL, 134-135
campo cColorBits, 25
campo iPixelType, 24-25
campos DEVMODE, 36
carga o llamado de matrices, 105-106
Carmack, John, 4
clase FreeTypeFont, 234-235
clase GLSLProgram, 148-149
Code::Blocks, 279, 282
código fuente, 281
códigos de error, 47
color, 46-48, 127
aplicación de, 150-151
compatibilidad hacia adelante, 10
constructores, 136-137
contenedor std::vector, 51
contenido del CD, 281-282
contexto de render, 18-22
PIXELFORMATDESCRIPTOR, 22
wglCreateContextAttribsARB, 20-22
wglDeleteContext, 19
wglGetCurrentContext, 22
wglMakeCurrent, 19-20
wglCreateContext, 19, 21
coordenadas de textura, 127, 165-168, 228-229
coordenadas de visualización, 75-76
CPlusPlus.com, 278

D

datos, modelo MD2, 253-255
declaraciones, GLSL, 136
Dell, 7
desarrollo de juegos
juegos en 3D, 4
razones para el, 3-4
despliegue de listas, 124
detección de colisión, 264-265
diagrama de función fija, 8-9, 123-124, 151;
***véase también* modo inmediato, matrices, vertex arrays y VBO**
diagrama de transformación de vértice, 75,127
diagrama programable, 123-154
biblioteca Kazmath, 152
búfer de acumulación, 124
búfer de selección, 124
clase GLSLProgram, 148-149
colores, aplicación de, 150-151
dibujo de píxeles, 124
ejemplo del robot, 152-153
evaluadores, 125
fragment shaders, 127-128
geometry shaders, 127
glGetShaderiv( ) pname, 143-144
líneas, 124
listas de despliegue, 124
mapas de bits, 124
matrices personalizadas, 151-152
modelo de obsolescencia, 124
modo Color Index, 124
modo inmediato, 124
pilas de atributo, 125

pilas de matrices, 124
posición de raster, 124
prueba Alpha, 125
punteado de polígono, 124
puntos non-sprite, 124
rectángulos, 124
Diagrama de función fija, 124, 144-151
shader language (GLSL), 124-125, 128-141
atributos de vértice, 135, 147-148
calificadores, 134-135
constructores, 136-137
declaraciones, 136
entradas de shaders, 135
estructuras de shaders, 128-129
estructuras, 132
funciones obsoletas, 139-1411
funciones, definición de, 138-139
funciones incluidas, 139
objetos, 141-145
operadores, 133
pre-procesadores, 129
reducción, 138
samplers, 130
tipos de datos, 130-132
uniformes, 135, 145-147
shaders, 140-149
archivos de información, 145
atributos de vértice, 147-148
datos, envío de, 145-148
objetos, GLSL, 141-145
uniformes, 145-147
variables, 129-135
vertex arrays del cliente, 124
diagrama programable (*continuación*)
cadena de extensión unificada, 125
modo envoltura de textura, 124
vertex shaders, 125-127, 149-150
dibujo de píxel, 124
***Doom*, 4, 7**
dwFlags, 24

E

ecuación de plano, 220-221
ejemplificadores, GLSL, 131
ejemplo de proyección, 100-102
ejemplo del robot, 93-96, 152
elementos de un juego, 4-7
entradas, 5
gráficos, 5
inteligencia artificial, 5
interfaz de usuario, 5
lógica del juego, 5
música, 5
redes, 5
sistema de menús, 5
sonido, 5
encabezado, modelo MD2, 252-253
enmascaramiento del color, 241-242
entradas de shaders, GLSL, 135
entradas, 5
entretenimiento interactivo, antecedente, 3-4
errores, localización de, 46-47
escalamiento, 78-79, 88-90
esferas, pruebas de, 223-224
estados numéricos, 44
estructura del shader, GLSL, 128-129
atributos de vértice, 135, 147-148
evaluadores, 125
Evans & Sutherland, 7
explosiones, 259-260
extensiones permitidas, 115-117
extensiones, 111-122
definidas, 111-112
funciones de, 113-114
cadenas de nombres, 112-113
puntos de entrada, 114-115
fichas, 113-114
definición de, 118
GLee (OpenGL Easy Extension library), 118-120
extensiones principales, 120
instalación, 119

GLEW (OpenGL Extension Wrangler library), 118
nombrar, 112
permitidas, 115-117
prefijos, 113
WGL, 118
Windows, 115

F

fichas, extensiones, 113-114
definición de, 118
filtrado de textura, 163-165
Flipcode, 278
formato del modelo MD2, 251-258
animación, 255-257
datos, 253-255
encabezado, 252-253
fotogramas clave, 252
interpolación, 255
representación, 258
variable m_currentFrame, 256
variable m_endFrame, 256
variable m_interpolation, 256
variable m_nextFrame, 256
variable m_startFrame, 256
formatos de píxel, 23-26, 160
banderas, 24
campo cColorBits, 25
campo iPixelType, 24-25
dwFlags, 24
función ChoosePixelFormat, 25-26
nSize, 24
formatos internos, texturas, 160
fotogramas clave, modelo MD2, 252
fragment shaders, 8, 127-128
FreeType, 229-235, 282
biblioteca, 230-231
carga de banderas, 233
clase FreeTypeFont, 234-235
fuentes de carga, 231-232
inicialización, 231-232
recursos, 234
tamaños de fuente, 232
texturas de glifo, 232-234
frustum culling, 219-226
aplicación, 224-225
ecuación de un plano, 220-221
esfera, prueba de, 223-224
punto, prueba de, 223
fuentes
2D, 227-235
3D, 235-236
tamaños, 232
textura, 227-228
fuentes 2D, 227-235
fuentes 3D, 235-236
fuentes de luz, 183
función ChoosePixelFormat, 25-26
función glRotate( ), 103-104
función glTranslate( ), 103-104
función gluLookAT( ), 11-13, 102-103
funciones de estado, 43-44
estados numéricos, 44
glIsEnabled, 45
habilitación, 44-45
funciones gl*Pointer( ), 52-54
funciones obsoletas, GLSL, 139-140
funciones, extensiones de, 113-114
puntos de entrada, 114-115
funciones, GLSL, 138-139

G

GameDev.net, 277
generación de nombres, VBOs, 59
gIGetString( ), 45-46
glBegin( ), 49
funciones, 50
parámetros, 49
glColor( ), 47
glDrawArrays( ), 55

glDrawElements( ), 56-57
glDrawRangeElements( ), 58
GLee (OpenGL easy Extension Library), 118-120, 282
configuración, 119
extensiones básicas, 120
GLEW (OpenGL Extension Wrangler library), 118
glGetShaderiv( ) pname, 143-144
glIsEnabled, 45
glLoadIdentity( ), 11-13
glMatrixMode( ), 11-13
glMultiDrawArrays( ), 58
GLSL, 124-125, 128-141
atributos de vértice, 135, 147-148
calificadores, 134-135
constructores, 136-137
declaraciones, 136
entradas de shader, 135
estructura del shader, 128-129
estructuras, 132
funciones incluidas, 139
funciones obsoletas, 139-140
funciones, definición de, 138-139
iluminación, 190-198
calidad, 197-198
focos, 194-197
iluminación direccional, 190-192
luces múltiples, 197
luces puntuales, 192-194
objetos, 141-145
operadores, 133
preprocesadores, 129
reducción, 138
samplers, 131
texturas múltiples, 215-216
tipos de datos, 130-132
uniformes, 135, 145-147
variables, 129-135
glTexImage1D, 159
glTexImage2D, 159
glTexImage3D, 159
gluPerspective, 11-13
GLUT (OpenGL Utility Toolkit), 10-11
gráficas, 5
grupo Khronos, 7

H

Hewlett-Packard, 7

I

IBM, 7
iluminación ambiental, 182, 186, 189
iluminación difusa, 182, 186
iluminación emisiva, 183, 186
iluminación especular, 182, 186, 188-189
iluminación, 181-198
ambiental, 182, 186, 189
atenuación de, 187
brillantez, 186
emisiva, 183, 186
especular, 182, 186, 188-189
fuentes de, 183
GLSL, 190-198
calidad, 197-198
focos, 194-197
iluminación direccional, 190-192
luces múltiples, 197
luces puntuales, 192-194
luz difusa, 182, 186
material de la superficie, 183
materiales, 186-190
matriz normal, 189-190
modelo Blinn-Phong, 185, 188
modelos, 183, 185
normales, 183-185
orientación de la superficie, 183
reflexión lambertiana, 187-188
Intel, 7
inteligencia artificial, 5

interfaz de usuario, 5
Interfaz de Dispositivo Gráfico, 18
interpolación, modelo MD2, 255

J-K

juegos 3D, 4

L

Lengyel, Eric, 280
Lighthouse3D, 278
líneas, 64-67, 124
  ancho de, 65-66
  antialiasing, 66
lógica del juego, 5
"Longs Peak," 9

M

magnificación, 163
mapa difuso, 213
mapas de bits, 124
mapas de cubos, 163, 210-211
mapeo ambiental, 209-210
mapeo de esferas, 209-210
mapeo de luz, 214
mapeo de texturas, 155-178, 207-217
  arreglos de textura, 159
  biblioteca GLU, 173-174
  búfer de color, 208-209
  coordenadas de textura, 165-168
  filtrado de texturas, 163-165
  formatos de píxel, 160
  formatos internos, 160
  glTexImage1D, 159
  glTexImage2D, 159
  glTexImage3D, 159
  magnificación, 163
  mapa de cubos, 163
  mapeo ambiental, 209-210
  mapeo de esferas, 209—210
  mapeo reflexivo de cubos, 210-211
  mip maps, 172-174
  modo de envoltura GL_CLAMP_TO_EDGE, 170
  modo de envoltura GL_MIRRORED_REPEAT, 170-171
  modo de envoltura GL_REPEAT, 169-170
  modos de envoltura de textura, 168-171
  objetos de textura, 157-158
  parámetros de textura, 168—171
  pruebas alfa, 211-213
  sub-imágenes, 207-208
  Targa, 174-177
  texturas en 1D, 162-163
  texturas en 2D, 159-162
  texturas en 3D, 163
  texturizado múltiple, 213-216
  tipos de datos, 161
  valores de filtro, 164
mapeo reflexivo de cubos, 210-211
material de la superficie, 183
materiales, iluminación de, 186-190
matrices personalizadas, 105-106, 151-152
matrices, 80-81, 105-106
  carga de, 105-106
  multiplicación de, 106
  personalización de, 105-106, 151-152
matriz de color, 91-93
matriz de proyección, 12, 90-93
matriz de textura, 91-93
matriz model-view, 80-81, 90-93
matriz normal, 189-190
Matrox, 7
mensaje WM_CLOSE, 34
mensaje WM_CREATE, 34
mensaje WM_DESTROY, 34
mensaje WM_KEYDOWN, 35
mensaje WM_SIZE, 35
método initialize( ), 11-12
método prepare( ), 40

método render( ), 12
método resize( ), 12
Meyers, Scott, 280
mezclas, 64, 198-203
de color, 203
de destinos, 199
de factores, 200
de fuentes, 199
de funciones, 202
Microsoft Visual C++, 279
Microsoft, 7
mip map (patrón de reducción de texturas), 172-174
modelo Blinn-Phong, 185, 188
modelo obsoleto, 9-10, 124
modelos, iluminación de, 183, 185
modo Color Index, 124
modo de envoltura de textura, 124, 168-171
modo de envoltura GL_CLAMP_TO_EDGE, 170
modo de envoltura GL_MIRRORED_REPEAT, 170-171
modo de envoltura GL_REPEAT, 169-170
modo de pantalla completa, 35-37
campos DEVMODE, 36
modo inmediato, 11, 39, 48-51, 124
modo polígono, 68-69
Mt. Evans, 9
multiplicación de matrices, 106
música, 5

N

NeHe, 278
niebla, 203-204
nombres de extensiones, 112
normalizada, iluminación, 183-185
nSize, 24
NVIDIA, 7

O

objetos de textura, 157-158
objetos, GLSL, 141-145
***Ogro Invasion!,* 261-265**
Open GL Utility Toolkit (paquete de herramientas de OpenGL), *véase* GLUT
OpenGL
aplicación, muestra de, 26-35, 37-39
definición de, 7
historia, 7-8
sitio web, 277
operadores, GLSL, 133
orientación de la superficie, 183
Ozone3D, 278

P

parámetros de textura, 168-171
pilas de atributos, 125
pilas de matrices, 80-81, 90-93, 124
PIXELFORMATDESCRIPTOR, 22-26
plataforma, 17-18
polígonos, culling de, 69-70
posición de raster, 124
prefijos, extensiones, 113
pre-procesadores, GLSL, 129
primitivas, 48-49
programabilidad, 8-9
proyección de perspectivas, 79-80, 98-99
proyección ortográfica, 80, 97-98
pruebas alfa, 125, 211-213
pruebas de tijera, 240-241
punteado de un polígono, 124
punto de visualización, 102-105
función glRotate( ), 103-104
función glTranslate( ), 103-104
función gluLookAt( ), 102-104
rutinas personalizadas, creación de, 104-105
puntos non-sprite, 124
puntos, 62-64
antialiasing, 63-64

non-sprite, 124
pruebas de, 223
tamaño de, 63

Q

***Quake*, 4, 251**

R

rectángulos, 124
recursos de programación C++, 278-280
redes, 5
reducción, 138
reflexión lambertiana, 187-188
registros de información, 145
render, modelo MD2, 258
render, VBO, 61-62
render, vertex arrays, 54-58
Rost, Randi J., 280
rotación, 78-79, 84-88
rutinas personalizadas, creación de, 104-105

S

SDL (Simple Direct Media layer), 11, 17, 279, 282
shaders de geometría (geometry shaders), 9, 127
shaders, 140-149
atributos de vértice, 147-148
datos, envío de, 145-148
objetos, GLSL, 141-145
registros de información, 145
uniformes, 145-147
Silicon Graphics, Inc., 7
Simple Direct Medial Layer (Capa media directa y simple), *véase* SDL
sistema de menús, 5
sistemas de coordenadas locales, 74
sistemas operativos, 7
sitio web Ultimate Game Programming, 278
sonido, 5
sprites de punto, 260-261
Standard Template Library (STL), C++, 51
subimágenes, 207-208
Sun Microsystems, 7
superficies traslúcidas, 244-245
Sutter, Herb, 280

T

tamaño de puntos, 63
Targa, 174-177
texto, despliegue de, 227-237
coordenadas de textura, 228-229
FreeType, 229-235
banderas de carga, 233
biblioteca, 230-231
carga de fuentes, 231-232
clase FreeTypeFont, 234-235
inicialización, 231-232
recursos, 234
tamaños de fuente, 232
texturas de glifo, 232-234
fuentes en 2D, 227-235
fuentes en 3D, 235-236
texturas de fuentes, 227-228
texturas 1D, 162-163
texturas 2D, 159-162
texturas 3D, 163
texturas de glifo, 232-234
texturas múltiples, 213-216
GLSL, 215-216
mapa difuso, 213
mapeo de luz, 214
unidades de textura, 214-215
tipos de datos, 130-132, 161
transformación normal, 127
transformaciones de coordenadas, 73-75
coordenadas de visualización, 75-76
transformaciones de proyección, 74-75, 79-80, 96-99

ortográficas, 97-98
perspectiva, 98-99
transformaciones de ventana de visualización, 74-75, 80, 99-100
transformaciones de visualización, 74-77
transformaciones del modelado, 74-75, 78-79
traslación, 78-79, 81-84
triángulos, 67-70
antialiasing, 70
modo polígono, 68-69
polígonos, culling de, 69-70

U

unidades de textura, 214-215
uniformes, 135, 145-147

V

valores de acceso, búfers, 61
valores de cadena, búsqueda de, 45-46
glGetString( ), 45-46
valores de filtro, 164
variable m_currentFrame, 256
variable m_endFrame, 256
variable m_interpolation, 256
variable m_nextFrame, 256
variable m_startFrame, 256
variables, GLSL, 129-135
VBOs, véase vertex buffer objects (RT x objetos del búfer de vértice)
vertex arrays, 51-58
banderas del tipo, 52
definición, 52
funciones gl*Pointer( ), 52-54
glDrawArrays( ), 55
glDrawElements( ), 56-57
glDrawRangeElements( ), 58
glMultiDrawArrays( ), 58
habilitación, 52
render, 54-58
vertex arrays del cliente, 124
vertex buffer objects, 58-62
llenado de búfers, 60-61
macro BUFFER_OFFSET, 6:
nombre, generación de, 59
representación, 61-62
valores de acceso, 61
valores de frecuencia, 61
vinculación de búfers, 60
vertex shaders, 9, 125-127, 149-150
vértices, 13

W

WGL, 18, 118
wglCreateContext, 19, 21
wglCreateContextAttribsARB, 20-22
wglDeleteContext, 19
wglGetCurrentContext, 22
wglMakeCurrent, 19-20
Windows, extensiones de, 115
Wolfenstein 3D, 4

X-Z

Z-fighting, 244

Programación de Videojuegos con OpenGL®

Segunda Edición

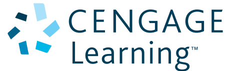
¿Es usted un programador principiante que apenas se está iniciando en la programación de gráficos en 3D? Si se siente cómodo programando en C++ y comprende de manera básica los conceptos de las matemáticas tridimensionales, la segunda edición de **Programación de videojuegos con OpenGL** le permitirá empezar a programar gráficos en 3D para juegos usando la API de OpenGL. Este libro ha sido revisado para operar con la versión de OpenGL 3.0, y es perfecto para programadores inexpertos en el desarrollo de videojuegos o que apenas se inician en OpenGL. A través de instrucciones paso a paso se enseñan nuevas habilidades y conceptos, además de presentar ejercicios de fin de capítulo para poner a prueba y reforzar lo aprendido. En este texto encontrará una cobertura completa pero concisa de todas las nuevas características de OpenGL y la forma en la que se aplican a los gráficos en 3D y el desarrollo de juegos, desde la creación de una aplicación simple con OpenGL hasta aplicar el mapeo de texturas e incluso desplegar fuentes en 2D; y a medida que se acerque al final del libro, será capaz de aplicar los conocimientos recién adquiridos de OpenGL para crear sus propios videojuegos.

Qué contiene el CD-ROM:
El código fuente para todas las aplicaciones de muestra que acompañan al libro.

Luke Benstead es coadministrador de http://nehe.gamedev.net/ y ha sido programador en OpenGL y C++ durante 7 años. En la actualidad trabaja como desarrollador de software en Londres, Inglaterra. Tiene una licenciatura en Programación Multimedia por la Universidad de Portsmouth.

Dave Astle fue cofundador de GameDev.net, en donde actualmente se desempeña como Presidente y Director Ejecutivo. Es coautor de la primera edición de Programación de videojuegos con OpenGL.

Kevin Hawkins es cofundador de GameDev.net y coautor de la primera edición de Programación de videojuegos con OpenGL.



ISBN-13: 978-607481762-1
ISBN-10: 607481762-6

http://latinoamerica.cengage.com