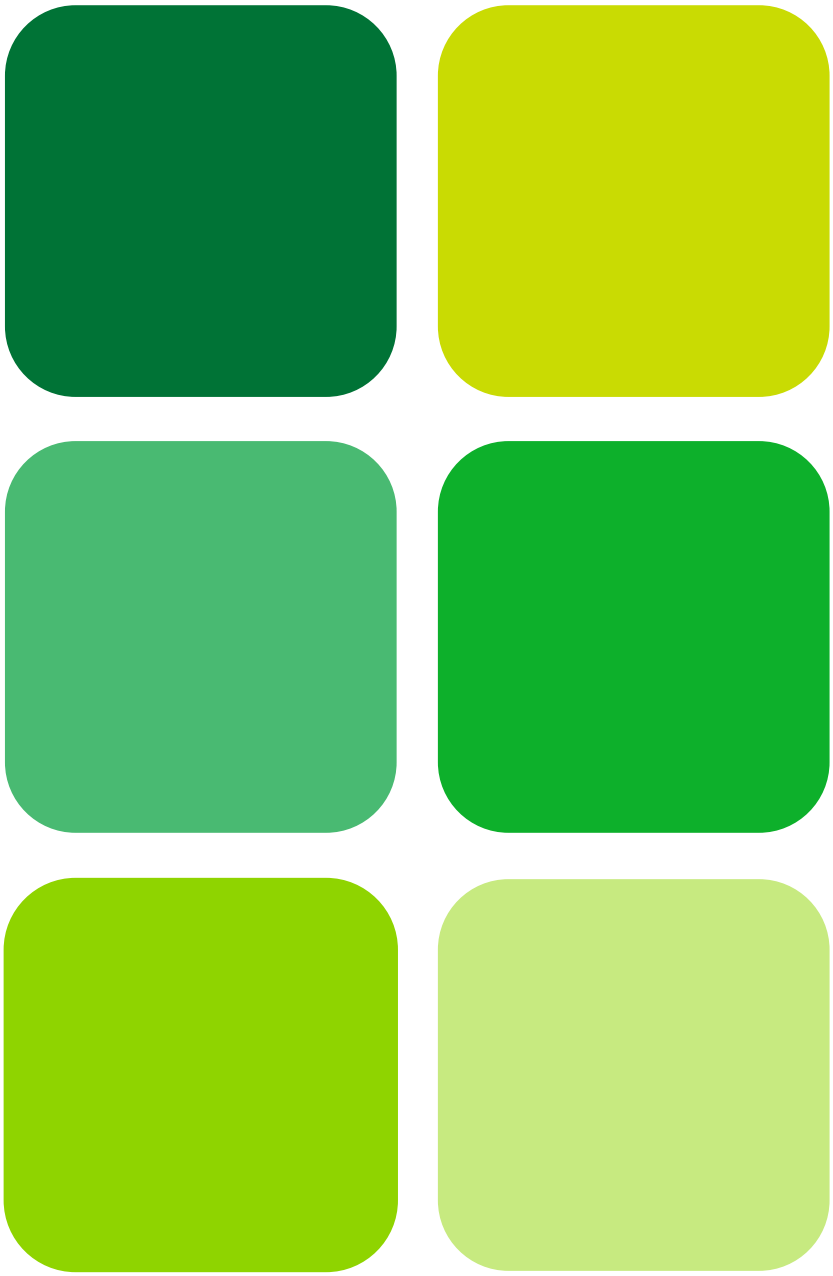


www.sapientia.uji.es | 71

UNIVERSITAT JAUME·I

Síntesis de imagen y animación

José Ribelles

Síntesis de imagen y animación

José Ribelles

Departament de Llenguatges i Sistemes Informàtics

Codi d'assignatura SIE020

Síntesis de imagen y animación - UJI - DOI: http://dx.doi.org/10.6035/Sapientia71

Edita: Publicacions de la Universitat Jaume I. Servei de Comunicació i Publicacions
Campus del Riu Sec. Edifici Rectorat i Serveis Centrals. 12071 Castelló de la Plana
http://www.tenda.uji.es e-mail: publicacions@uji.es

Col·lecció Sapientia, 71
www.sapientia.uji.es
Primera edició, 2012

ISBN: 978-84-695-5223-0

Publicacions de la Universitat Jaume I és una editorial membre de l'UNE, cosa que en garanteix la difusió de les obres en els àmbits nacional i internacional. www.une.es

Índice general

Índice de figuras

Índice de listados

Prefacio

La asignatura *Síntesis de Imagen y Animación* es una asignatura optativa de 4 créditos ECTS y pertenece al plan de estudios del *Máster Universitario en Sistemas Inteligentes* de la Universitat Jaume I. Este máster cuenta principalmente con alumnos titulados en Ingeniería Informática. En el plan de estudios de la titulación de Ingeniería Informática que se imparte en dicha universidad, la materia Informática Gráfica es también una materia optativa. Y en el nuevo título de grado cuya implantación se inició en el curso 2010-11 también continua como una materia optativa. En consecuencia, la asignatura *Síntesis de Imagen y Animación* se enfoca teniendo en cuenta que los alumnos, con bastante probabilidad, no han adquirido conocimientos en la materia. Además, dado que también es una asignatura optativa en el máster, la asignatura se plantea de manera muy práctica con el objetivo de facilitar al alumnado el uso de los gráficos por ordenador en su área de interés, esté o no relacionada con la informática gráfica.

El temario de la asignatura se ha organizado en tres partes:

1. CONCEPTOS. Este primer bloque introduce al alumno en la programación moderna de gráficos por computador a través del estándar OpenGL. Trata los fundamentos del proceso de obtención de imágenes sintéticas centrándose en las etapas que el programador debe realizar teniendo en cuenta el *pipeline* de los procesadores gráficos actuales.

2. TÉCNICAS BÁSICAS. El objetivo de este bloque es introducir un conjunto de técnicas que ayuden a reforzar los conocimientos adquiridos en el primer bloque. Respecto a la diversidad de métodos que existen se ha optado por impartir aquellos que puedan ser más didácticos y que, tal vez con menor esfuerzo de programación, permitan mejorar de manera importante la calidad visual de la imagen sintética.

3. TÉCNICAS AVANZADAS. Este último bloque introduce técnicas más complejas y está más relacionado con el desarrollo de aplicaciones gráficas tratando, por ejemplo, técnicas de interacción y de animación por computador.

El objetivo de este libro es aportar suficiente material teórico y práctico para apoyar la docencia, tanto presencial, desarrollada en clases de teoría o en laboratorio, como no presencial, proporcionando al estudiante un material que facilite el

estudio de la materia, de un nivel y contenido adecuado a la asignatura. Este libro pretende ser el complemento ideal a las explicaciones que el profesor imparta en sus clases, no su sustituto, y en el que el alumno deberá con sus anotaciones mejorar su contenido. Así, el libro cuenta con la misma organización que la asignatura y trata los siguientes aspectos:

1. CONCEPTOS. Introduce la programación de *Shaders* con OpenGL, el modelado poligonal, las transformaciones geométricas, la transformación de la cámara, proyecciones, el modelo de iluminación de Phong y la aplicación de texturas 2D.

2. TÉCNICAS BÁSICAS. Se ocupa de introducir técnicas para el procesado de imágenes, como la convolución o el *antialiasing*, técnicas para aumentar el realismo visual, como trasparencias, reflejos y sombras, y métodos de aplicación de texturas más avanzados como el *environment mapping* o las texturas 3D.

3. TÉCNICAS AVANZADAS. Introduce la programación de la interacción, manejo de eventos y selección de objetos 3D, el modelado de curvas cúbicas y superficies bicúbicas, y diferentes técnicas de animación con *shaders*.

Recursos en línea

Se ha creado la página web `http://sie020.uji.es` como apoyo a este material, para mantenerlo actualizado incluyendo más ejercicios, programas de ejemplo, *shaders*, páginas de ayuda, fe de erratas, etc.

Agradecimientos

Quiero expresar mi agradecimiento a los profesores Miguel Chover Sellés y M. Ángeles López Malo, ambos de la Universitat Jaume I, que han querido colaborar con la revisión de este material y que sin duda han contribuido a aumentar su calidad.

Aunque existen muchas obras que han resultado muy útiles para preparar este material, sólo unas pocas han sido las que más han influenciado en su contenido. En concreto: *Computer Graphics: Principles & Practice* (Foley et al., 1990), *Fundamentals of Computer Graphics* (Shirley et al., 2009), *Real-Time Rendering* (Akenine-Möller et al., 2008), *OpenGL Shading Language* (Rost et al., 2010) y *OpenGL Programming Guide* (Dave Shreiner, 2009).

Junio, 2012

PARTE I
CONCEPTOS

Capítulo 1

Introducción a OpenGL

OpenGL es una interfaz de programación de aplicaciones (API) para generar imágenes por ordenador. Permite desarrollar aplicaciones interactivas que producen imágenes en color de alta calidad formadas por objetos tridimensionales (ver figura 1.1). Además, OpenGL es independiente tanto del sistema operativo como del sistema gráfico de ventanas. En este capítulo se introduce la programación con OpenGL a través de un pequeño programa y se presenta el lenguaje GLSL para la programación de *shaders*.

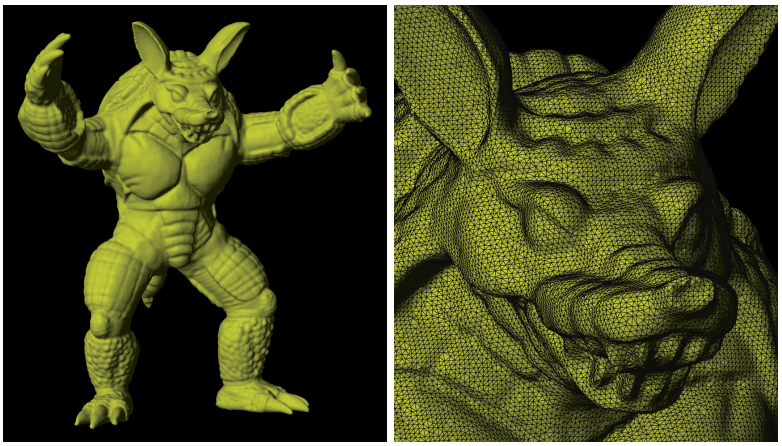


Figura 1.1: Ejemplo de objeto tridimensional dibujado con OpenGL

1.1. OpenGL

OpenGL se presentó en 1992. Su predecesor fue Iris GL, un API diseñado y soportado por la empresa Silicon Graphics. Desde entonces, la OpenGL Architecture Review Board (ARB) conduce la evolución de OpenGL controlando la especificación y los tests de conformidad.

En sus orígenes, OpenGL se basó en un *pipeline* configurable de funcionamiento fijo. El usuario podía especificar algunos parámetros pero el funcionamiento y el orden de procesamiento era siempre el mismo. Con el paso del tiempo, los fabricantes de hardware gráfico necesitaron dotarla de mayor funcionalidad que la inicialmente concebida. Así, se creó un mecanismo para definir extensiones que, por un lado, permitía a los fabricantes proporcionar hardware gráfico con mayores posibilidades, al mismo tiempo que ofrecían la capacidad de no realizar siempre el mismo *pipeline* de funcionalidad fija.

En el año 2004 aparece OpenGL 2.0, el cual incluiría el OpenGL Shading Language, GLSL 1.1, e iba a permitir a los programadores la posibilidad de escribir código que fuese ejecutado por el procesador gráfico. Para entonces, las principales empresas fabricantes de hardware gráfico ya ofrecían procesadores gráficos programables. A estos programas se les denominó *shaders* y permitieron incrementar las prestaciones y el rendimiento de los sistemas gráficos de manera espectacular, al generar además una amplia gama de efectos: iluminación más realista, fenómenos naturales, texturas procedurales, procesamiento de imágenes, efectos de animación, etc. (ver figura 1.2).

Dos años más tarde, el consorcio ARB pasó a ser parte del grupo Khronos (http://www.khronos.org/). Entre sus miembros activos, promotores y contribuidores se encuentran empresas de prestigio internacional como AMD (ATI), Apple, Nvidia, S3 Graphics, Intel, IBM, ARM, Sun, Nokia, etc.

Es en el año 2008 cuando OpenGL, con la aparición de OpenGL 3.0 y GLSL 1.3, adopta el modelo de obsolescencia pero manteniendo compatibilidad con las versiones anteriores. Sin embargo, en el año 2009, con las veriones de OpenGL 3.1 y GLSL 1.4 es cuando probablemente se realiza el cambio más significativo, el *pipeline* de funcionalidad fija y sus funciones asociadas son eliminadas, aunque disponibles aún a través de extensiones que están soportadas por la mayor parte de las implementaciones. En este libro, todos los ejemplos de código se han realizado de acuerdo a esta última versión de OpenGL evitando siempre utilizar las extensiones de compatibilidad.

1.2. El *pipeline*

El funcionamiento básico del *pipeline* se representa en el diagrama simplificado que se muestra en la figura 1.3.

Las etapas de procesamiento de vértices y de fragmentos son programables. El programador debe escribir los *shaders* que desea que sean ejecutados en cada una de ellas.

El procesador de vértices acepta vértices como entrada, los procesa utilizando el *vertex shader* y envía el resultado a la etapa denominada *Construcción de la primitiva y conversión al raster*. En ella, los vértices se agrupan dependiendo de qué primitiva geométrica se está procesando (puntos, líneas o polígonos). De forma opcional también se pueden utilizar para generar nuevas primitivas. A continuación

Figura 1.2: Ejemplos de objetos dibujados mediante *shaders*

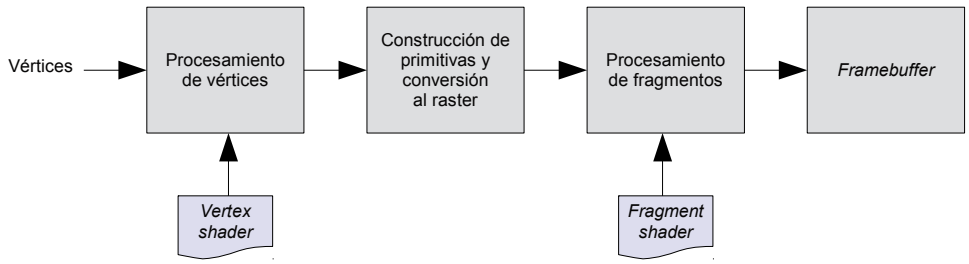


Figura 1.3: Secuencia básica de operaciones del *pipeline* de OpenGL

se determinan los píxeles que se deben de colorear produciendo como salida lo que se denominan *fragmentos*, que forman a su vez la entrada de la siguiente etapa.

El procesador de fragmentos determina el color definitivo utlizando el *fragment shader*, que se ejecuta para cada fragmento recibido. Su resultado se envía al *framebuffer*, pero antes de actualizar el píxel correspondiente aún se ha de determinar si debe ser escrito o no. Finalmente, el contenido del *framebuffer* también puede ser recuperado y reutilizado en alguna de las etapas anteriores.

1.3. Otras API

1.3.1. FreeGLUT, The Free OpenGL Utility Toolkit

Sitio web: `http://freeglut.sourceforge.net/`

Esta API permite crear ventanas que contienen contextos OpenGL en cualquiera de los sistemas operativos más populares como Microsoft Windows, Apple Mac OS X y Linux. Además permite especificar rutinas de *callback* para atender eventos de ventana, ratón o teclado. Como herramienta para la creación de interfaces gráficas de usuario no proporciona más que la posibilidad de crear menús desplegables, y sin interés en el caso de necesitar un entorno de usuario con mayor complejidad. Sin embargo, su simplicidad y portabilidad la mantienen entre las favoritas.

Para suplir la escasez de controles que permitan crear interfaces que incluyan elementos tan habituales como botones, barras de desplazamiento o listas desplegables, por ejemplo, al mismo tiempo que asegurar la portabilidad entre los sistemas citados, la compatibilidad con la GLUT, y que requieran poco esfuerzo de aprendizaje, se recomiendan:

- GLUI, The User Interface Library. `http://glui.sourceforge.net/`
- FLTK, The Fast Light Toolkit. `http://www.fltk.org/`

1.3.2. GLEW, The OpenGL Extension Wrangler Library

Sitio web: `http://glew.sourceforge.net/`

La librería GLEW permite determinar de manera eficiente y en tiempo de ejecución qué extensiones OpenGL están soportadas en la plataforma en la que se está ejecutando la aplicación gráfica. También existe para los sistemas operativos más populares.

1.3.3. GLM, OpenGL Mathematics

Sitio web: `http://glm.g-truc.net/`

La librería GLM es una librería matemática para aplicaciones gráficas basada en la especificación del lenguaje GLSL. Está hecha en C++ y consta únicamente de ficheros de cabecera. Es independiente de la plataforma y soporta una amplia variedad de compiladores.

Ejercicios

► **1.1** Comprueba que tienes instalado el último *driver* del fabricante del hardware gráfico de tu ordenador personal. Comprueba la existencia de las librerías FreeGLUT, GLEW y GLM. Actualiza las que tengas ya instaladas e instala las demás.

1.4. El primer programa con OpenGL

El flujo general de una aplicación OpenGL consta de cuatro pasos:

1. Crear una ventana.
2. Inicializar los estados de OpenGL.
3. Procesar los eventos generados por el usuario.
4. Dibujar la escena (y volver al paso 3).

El listado 1.1 incluye la rutina principal de un programa en C que utiliza FreeGLUT y OpenGL. Esta rutina crea una ventana, inicializa los estados de OpenGL a través de la rutina *init()*, establece las funciones de *callback* y, finalmente, entra en un bucle a la espera de que se produzcan eventos. Sin embargo, la salida gráfica de este primer programa no es más que un fondo de ventana de color amarillo. Aunque simple, este primer programa ya consta de la estructura general de una aplicación OpenGL.

Ejercicios

► **1.2** Consulta en las guías de programación de OpenGL y FreeGLUT el significado de cada una de las funciones utilizadas en las rutinas del listado 1.1, y contesta a las siguientes cuestiones:

- ¿Qué estados de OpenGL inicializarás preferentemente en la rutina *init()*?
- ¿Qué rutina es la encargada de dibujar la escena?
- ¿Cuál es el objetivo de la llamada a *glutSwapBuffers()* al final de la rutina *display()*?

Listado 1.1: El primer programa en C que utiliza OpenGL y FreeGLUT

```
#include <GL/freeglut.h>

void init (void)
{
  glClearColor (1.0, 1.0, 0.0, 1.0);
}

void reshape (int ancho, int alto)
{
  glViewport (0, 0, ancho, alto);
}

void display (void)
{
  glClear (GL_COLOR_BUFFER_BIT);
  glutSwapBuffers ();
}

int main (int argc, char **argv)
{
  glutInit               (&argc, argv);
  glutInitDisplayMode    (GLUT_DOUBLE | GLUT_RGBA);
  glutInitWindowSize     (1024, 768);
  glutInitWindowPosition (100, 100);
  glutCreateWindow       (argv[0]);

  init ();

  glutDisplayFunc (display);
  glutReshapeFunc (reshape);

  glutMainLoop ();

  return 0;
}
```

1.5. GLSL

El lenguaje GLSL forma parte de OpenGL desde su versión 2.0, y permite al programador escribir el código que desea ejecutar en los procesadores programables de la GPU. En la actualidad hay cinco tipos de procesadores: vértices, control de teselación, evaluación de teselación, geometría y fragmentos; por lo que también decimos que hay cinco tipos de *shaders*, uno por cada tipo de procesador.

GLSL es un lenguaje de alto nivel, parecido al C, aunque también toma prestadas algunas características del C++. Su sintaxis se basa en el ANSI C. Constantes, identificadores, operadores, expresiones y sentencias son básicamente las mismas que en C. El control de flujo con bucles, la sentencias condicionales *if-then-else* y las llamadas a funciones son idénticas al C.

Pero GLSL también añade características no disponibles en C, entre otras se destacan las siguientes:

- Tipos vector: vec2, vec3, vec4
- Tipos matriz: mat2, mat3, mat4
- Tipos *sampler* para el acceso a texturas: sampler1D, sampler2D, sampler3D, samplerCube
- Tipos para comunicarse entre *shaders* y con la aplicación: uniform, in, out
- Acceso a componentes de un vector mediante: .xyzw .rgba .stpq
- Operaciones vector-matriz, por ejemplo: $vec4\ a = b * c$, siendo b de tipo $vec4$ y c de tipo $mat4$
- Variables predefinidas que almacenan estados de OpenGL

GLSL también dispone de funciones propias como, por ejemplo, trigonométricas (*sin*, *cos*, *tan*, etc.), exponenciales (*pow*, *exp*, *sqrt*, etc.), comunes (*abs*, *floor*, *mod*, etc.), geométricas (*length*, *cross*, *normalize*, etc.), matriciales (*transpose*, *inverse*, etc.) y operaciones relacionales con vectores (*equal*, *lessThan*, *any*, etc). Consulta la especificación del lenguaje para conocer el listado completo.

Y también hay características del C no soportadas en OpenGL como es el uso de punteros, los tipos: *byte*, *char*, *short*, *long int*; y la conversión implícita de tipos está muy limitada.

Del C++, GLSL copia la sobrecarga, el concepto de constructor y el que las variables se pueden declarar en el momento de ser utilizadas.

En el listado 1.2 se muestra incluido en un programa en C un ejemplo de *shader*, el más simple posible. Las variables *vertexShaderSource* y *fragmentShaderSource* contienen los fuentes de los *shaders* de vértices y de fragmentos respectivamente. Cuando desde la aplicación se ordene dibujar, cada vértice producirá la ejecución del *shader* de vértices, el cual a su vez produce como salida la posición

Listado 1.2: Un *shader* muy básico

```
const char *vertexShaderSource =
{
  "#version 140\n"
  ""
  "in vec4 posicion;"
  ""
  "void main ()"
  "{"
  "    gl_Position = posicion;"
  "}"
};

const char *fragmentShaderSource =
{
  "#version 140\n"
  ""
  "out vec4 color;"
  ""
  "void main (void)"
  "{"
  "    color = vec4 (1.0,0.0,0.0,1.0);"
  "}"
};
```

del vértice (y también, en el caso de haberlos, de otros atributos como el color, la normal, etc.). En el ejemplo del listado 1.2, la posición se almacena en una de las variables predefinidas en GLSL. El resultado atraviesa el *pipeline*, los vértices se agrupan dependiendo del tipo de primitiva a dibujar, y en la etapa de conversión al *raster* la posición del vértice (y también de sus atributos en el caso de haberlos) es interpolada generando los fragmentos y produciendo, cada uno de ellos, la ejecución del *shader* de fragmentos en el procesador correspondiente. El objetivo de este último *shader* es determinar el color definitivo del fragmento. Siguiendo con el ejemplo, todos los fragmentos son puestos a color rojo (en formato RGBA).

1.6. Trabajando con *shaders*

El compilador de GLSL está integrado en el propio *driver* de OpenGL. Esto implica que la aplicación en tiempo de ejecución será quien envíe el código fuente del *shader* al *driver* para que sea compilado y enlazado, creando un ejecutable que puede ser instalado en los procesadores correspondientes. Tres son los pasos a realizar:

1. Crear y compilar los objetos *shader*.
2. Crear un programa y añadirle los objetos compilados.
3. Enlazar el programa creando un ejecutable.

El listado 1.3 muestra un ejemplo de todo el proceso. Observa detenidamente la función *initShader* e identifica en el código cada una de las tres etapas. Utiliza la especificación de OpenGL para conocer más a fondo cada una de las órdenes que aparecen en el ejemplo. Cabe señalar también que en el listado se han utilizado *glGetShaderiv* y *glGetProgramiv* para conocer el resultado de la compilación y el enlazado e informar de posibles errores.

Ahora que ya tenemos el ejecutable es el momento de obtener los índices de las variables del *shader* para poder envíar datos desde la aplicación. Como hay dos tipos de variables, las uniformes y los atributos de los vértices, se utiliza una función diferente para cada tipo. Respectivamente son: *glGetUniformLocation* y *glGetAttribLocation*. En el listado 1.3, al final de la función *initShader*, se muestra cómo se averigua el índice de la variable *posicion* correspondiente al *shader* de vértices del listado 1.2.

Por último, para que el programa ejecutable sea instalado en los procesadores correspondientes, es necesario indicarlo con la orden *glUseProgram* que como parámetro debe recibir el identificador del programa que se desea utilizar. La carga de un ejecutable siempre supone el desalojo del que hubiera con anterioridad.

Listado 1.3: Ejemplo de creación de un *shader*

```
void chequeaEnlazado (GLuint program)
{
  GLuint estado;

  glGetProgramiv (program, GL_LINK_STATUS, &estado);
  if (estado == FALSE)
  {
    GLint len;

    glGetProgramiv (program, GL_INFO_LOG_LENGTH, &len);
    std::string msgs(' ',len);
    glGetProgramInfoLog (program, len, &len, &msgs[0]);
    std::cerr << msgs << std::endl;
  }
}

void chequeaCompilacion (GLuint shader)
{
  GLuint estado;

  glGetShaderiv (shader, GL_COMPILE_STATUS, &estado);
  if (estado == FALSE)
  {
    GLint len;

    glGetShaderiv (shader, GL_INFO_LOG_LENGTH, &len);
    std::string msgs(' ',len);
    glGetShaderInfoLog (shader, len, &len, &msgs[0]);
```

```
    std::cerr << msgs << std::endl;
  }
}

void initShader (void)
{
  GLuint vertexShader = glCreateShader (GL_VERTEX_SHADER);
  glShaderSource  (vertexShader, 1, &vertexShaderSource, NULL);
  glCompileShader (vertexShader);
  chequeaCompilacion (vertexShader);

  GLuint fragmentShader = glCreateShader (GL_FRAGMENT_SHADER);
  glShaderSource  (fragmentShader, 1, &fragmentShaderSource,NULL);
  glCompileShader (fragmentShader);
  chequeaCompilacion (fragmentShader);

  program = glCreateProgram ();

  glAttachShader  (program, vertexShader);
  glAttachShader  (program, fragmentShader);

  glLinkProgram   (program);
  chequeaEnlazado (program);

  coordenadas = glGetAttribLocation (program, "posicion");
}
```

Capítulo 2

Modelado

Se denomina modelo al conjunto de datos que describe a un objeto y que puede ser utilizado por un sistema gráfico para ser visualizado. Hablamos de modelo poligonal cuando se utilizan polígonos para describir la geometría. En general, el triángulo es la primitiva más utilizada aunque también el cuadrilátero se emplea en ocasiones (ver figura 2.1).

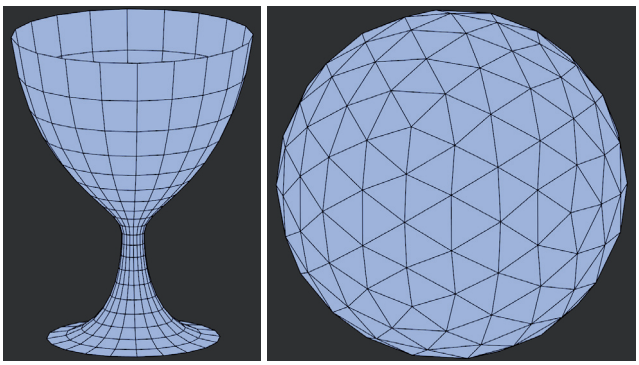


Figura 2.1: A la izquierda objeto representado mediante cuadriláteros, y a la derecha objeto representado mediante triángulos

2.1. Modelado poligonal

Un modelo polígonal, además de vértices y caras, suele almacenar otra información a modo de atributos como el color, la normal o las coordenadas de textura. Estos atributos son necesarios para mejorar significativamente el realismo visual. Por ejemplo, en la figura 2.2 se muestra el resultado de la visualización del modelo poligonal de una tetera obtenido gracias a que, además de la geometría, se ha proporcionado la normal de la superficie para cada vértice.

La normal es un vector perpendicular a la superficie en un punto. Si la superficie es plana, la normal es la misma para todos sus puntos. Para un triángulo es

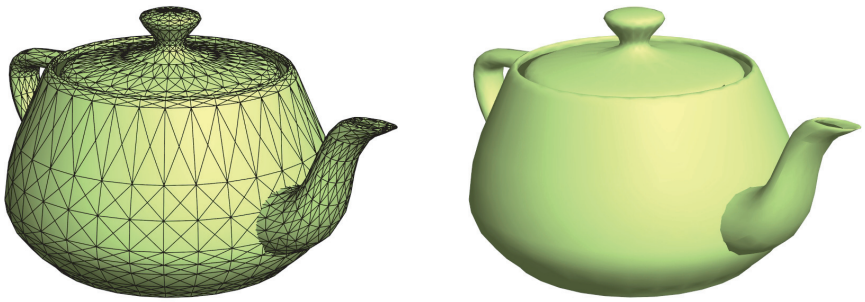


Figura 2.2: Visualización del modelo poligonal de una tetera. En la imagen de la izquierda se pueden obervar los polígonos utilizados para representarla

fácil obtenerla realizando el producto vectorial de dos de los vectores directores de sus aristas. Ya que el producto vectorial no es commutativo, es muy importante establecer cómo se va a realizar el cálculo y también que los vértices que forman las caras se especifiquen siempre en el mismo orden, para así obtener todas las normales de manera consistente.

Ejercicios

► **2.1** Observa la siguiente descripción poligonal de un objeto. Las líneas que comienzan por v se corresponden con los vértices e indican sus coordenadas. El primero es el número 1 y los demás se enumeran de forma consecutiva. Las líneas que comienzan por f se corresponden con las caras e indican qué vertices lo forman.

```
v 0 0 0
v 0 0 1
v 1 0 1
v 1 0 0
v 0 1 0
v 0 1 1
v 1 1 1
v 1 1 0
f 1 3 2
f 1 4 3
f 1 2 5
f 2 6 5
f 3 2 6
f 3 6 7
f 3 4 7
f 4 8 7
f 4 1 8
f 1 5 8
```

- Dibújalo en papel, ¿qué objeto representa?
- ¿Están todas sus caras definidas en el mismo orden?
- ¿En qué sentido están definidas, horario o antihorario?

2.2. Polígonos y OpenGL

2.2.1. Primitivas geométricas

Las primitivas básicas de dibujo son el punto, el segmento de línea y el triángulo. Cada primitiva se define especificando sus respectivos vértices. Un vértice consta de cuatro valores ya que se trabaja con coordenadas homogéneas. Las primitivas geométricas se forman agrupando vértices, y estas se agrupan a su vez para definir objetos de mayor complejidad.

En OpenGL, la primitiva geométrica se utiliza para especificar cómo se han de agrupar los vértices tras ser operados en el procesador de vértices y así poder ser visualizada. Son las siguientes:

- Dibujo de puntos:
  - GL_POINTS
- Dibujo de líneas:
  - Segmentos sueltos: GL_LINES
  - Secuencia o tira de segmentos: GL_LINE_STRIP
  - Secuencia cerrada de segmentos: GL_LINE_LOOP
- Triángulos
  - Triángulos sueltos: GL_TRIANGLES
  - Tira de triángulos: GL_TRIANGLE_STRIP
  - Abanico de triángulos: GL_TRIANGLE_FAN

Una tira de triángulos es una serie de triángulos conectados a través de aristas compartidas. Se define mediante una secuencia de vértices, donde los primeros tres vértices definen el primer triángulo. Cada nuevo vértice define un nuevo triángulo utilizando ese vértice y los dos últimos del triángulo anterior. El abanico de triángulos es igual que la tira excepto que cada vértice nuevo sustituye siempre al segundo del triángulo anterior.

Ejercicios

► **2.2** Obtén las tiras de triángulos que representan el objeto definido en el ejercicio anterior, ¿puedes conseguirlo con solo una tira?

2.2.2. Modelado

Habitualmente solemos asociar el concepto de vértice con las coordenadas que definen la posición de un punto en el espacio. En OpenGL, el concepto de vértice

es más general, entendiéndose como una agrupación de datos a los que llamamos atributos. Estos pueden ser de cualquier tipo: reales, enteros, vectores, etc. Los más utilizados son la posición, la normal y el color. Pero OpenGL permite que el programador pueda incluir como atributo cualquier información que para él tenga sentido y que necesite tener en el *shader*.

OpenGL no proporciona mecanismos para describir o modelar objetos geométricos complejos, sino que proporciona mecanismos para especificar cómo dichos objetos deben ser dibujados. Es responsabilidad del programador definir las estructuras de datos adecuadas para almacenar la descripción del objeto. Sin embargo, como OpenGL requiere que la información que vaya a visualizarse se disponga en vectores, lo habitual es utilizar también vectores para almacenar los vértices así como sus atributos y utilizar índices a dichos vectores para definir las primitivas geométricas. El listado 2.1 muestra un ejemplo de estructura de datos.

Listado 2.1: Ejemplo de estructura de datos para un modelo poligonal

```
struct vertice
{
  GLfloat coordenadas[3];
  GLfloat color[3];
}
Vertices[nVertices];

struct triangulo
{
  GLuint indices[3];
}
Triangulos[nTriangulos];
```

2.2.3. Visualización

En primer lugar, el modelo poligonal se ha de almacenar en *buffer objects*. Un *buffer object* no es mas que una porción de memoria reservada dinámicamente y controlada por el propio procesador gráfico. Siguiendo con el ejemplo de la estructura de datos que se muestra en el listado 2.1 necesitaremos dos *buffers*, uno para el vector de vértices y otro para el de triángulos. Después hay que asignar a cada *buffer* sus datos correspondientes. El listado 2.2 recoge estas operaciones, exáminalo y acude a la especificación del lenguaje para conocer más detalles de las funciones utilizadas.

En segundo lugar, hay que obtener los índices de las variables del *shader* que representan los atributos de los vértices, que en nuestro ejemplo son las coordenadas y el color. El listado 2.3 muestra el *shader* que se va a utilizar y las instrucciones que permiten obtener los índices correspondientes.

Ahora que ya tenemos el modelo almacenado en la memoria controlada por la GPU, el *shader* compilado y enlazado, y obtenidos los índices de los atribu-

Listado 2.2: Creación de *buffer objects*

```
enum {bufferVertices, bufferTriangulos, nBuffers}
GLuint buffers[nBuffers];

glGenBuffers (nBuffers, buffers);

glBindBuffer (GL_ARRAY_BUFFER, buffers[bufferVertices]);
glBufferData (GL_ARRAY_BUFFER,
              nVertices*sizeof(vertice),
              Vertices, GL_STATIC_DRAW);

glBindBuffer (GL_ELEMENT_ARRAY_BUFFER, buffers[bufferTriangulos]);
glBufferData (GL_ELEMENT_ARRAY_BUFFER,
              nTriangulos*sizeof(triangulo),
              Triangulos, GL_STATIC_DRAW);
```

Listado 2.3: *Shader* y obtención de los índices de los atributos de los vértices

```
const char *vertexShaderSource =
{
  "#version 140\n"
  ""
  "in  vec3 posicion;"
  "in  vec3 color;"
  "out vec4 nuevoColor;"
  ""
  "void main()"
  "{"
  "   nuevoColor  = vec4 (color, 1.0);"
  "   gl_Position = vec4 (posicion, 1.0);"
  "}"
};

const char *fragmentShaderSource =
{
  "#version 140\n"
  ""
  "in  vec4 nuevoColor;"
  "out vec4 colorFragmento;"
  ""
  "void main(void)"
  "{"
  "   colorFragmento = nuevoColor;"
  "}"
};

GLuint iPosicion = glGetAttribLocation (program, "posicion");
GLuint iColor    = glGetAttribLocation (program, "color");
```

tos de los vértices, ya sólo nos queda el ultimo paso, su visualización. Primero, para cada atributo hay que especificar dónde y cómo se encuentran almacenados así como habilitar los vectores correspondientes. Después ya se puede ordenar el dibujado, indicando tipo de primitiva y número de elementos. Los vértices se procesarán de manera independiente, pero siempre en el orden en el que son enviados al procesador gráfico. El listado 2.4 muestra esta operación. De nuevo, acude a la especificación del lenguaje para conocer más detalles de las órdenes utilizadas.

Listado 2.4: Ejemplo de visualización de un modelo poligonal

```
void display (void)
{
  glClear (GL_COLOR_BUFFER_BIT);

  glUseProgram (program);

  glBindBuffer (GL_ARRAY_BUFFER, buffers[bufferVertices]);
  glVertexAttribPointer (iPosicion, 3, GL_FLOAT, GL_FALSE,
                         sizeof(vertice), (GLvoid*)0);
  glVertexAttribPointer (iColor,    3, GL_FLOAT, GL_FALSE,
                         sizeof(vertice),
                         (GLvoid*)(sizeof(GLfloat)*3));
  glEnableVertexAttribArray (iPosicion);
  glEnableVertexAttribArray (iColor);

  glBindBuffer    (GL_ELEMENT_ARRAY_BUFFER,
                   buffers[bufferTriangulos]);
  glDrawElements (GL_TRIANGLES, nTriangulos*3,
                  GL_UNSIGNED_INT, 0);

  glBindBuffer (GL_ARRAY_BUFFER, 0);
  glBindBuffer (GL_ELEMENT_ARRAY_BUFFER, 0);

  glutSwapBuffers();
}
```

Ejercicios

► **2.3** Escribe un programa que dibuje la figura que se muestra a continuación. No importa si la degradación de color que obtengas no coincide exactamente. Ayúdate de los listados de código que aparecen en este capítulo y también en el anterior.

Capítulo 3

Transformaciones geométricas

En la etapa de modelado los objetos se definen bajo un sistema de coordenadas propio. A la hora de crear una escena, estos objetos se incorporan bajo un nuevo sistema de coordenadas conocido como sistema de coordenadas del mundo. Este cambio de sistema de coordenadas es necesario y se realiza mediante transformaciones geométricas.

3.1. Transformaciones básicas

3.1.1. Traslación

La transformación de traslación consiste en desplazar el punto $p = (p_x, p_y, p_z)$ mediante un vector $t = (t_x, t_y, t_z)$ de manera que el nuevo punto $q = (q_x, q_y, q_z)$ se obtiene así:

$$q_x = p_x + t_x, \qquad q_y = p_y + t_y, \qquad q_z = p_z + t_z \tag{3.1}$$

La representación matricial con coordenadas homogéneas de esta transformación es:

$$T(t) = T(t_x, t_y, t_z) = \begin{pmatrix} 0 & 0 & 0 & t_x \\ 0 & 0 & 0 & t_y \\ 0 & 0 & 0 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.2}$$

Utilizando esta representación, el nuevo punto se obtiene así:

$$\tilde{q} = T(t) \cdot \tilde{p} \tag{3.3}$$

donde $\tilde{p} = (p_x, p_y, p_z, 1)$ y $\tilde{q} = (q_x, q_y, q_z, 1)$, es decir, los puntos p y q en coordenadas homogéneas.

3.1.2. Escalado

La transformación de escalado consiste en multiplicar el punto $p = (p_x, p_y, p_z)$ con los factores de escala s_x, s_y y s_z de tal manera que el nuevo punto $q = (q_x, q_y, q_z)$ se obtiene así:

$$q_x = p_x \cdot s_x, \qquad q_y = p_y \cdot s_y, \qquad q_z = p_z \cdot s_z \tag{3.4}$$

La representación matricial con coordenadas homogéneas de esta transformación es:

$$S(s) = S(s_x, s_y, s_z) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.5}$$

Utilizando esta representación, el nuevo punto se obtiene así: $\tilde{q} = S(s) \cdot \tilde{p}$.

Ejercicios

► **3.1** Cuando los tres factores de escala son iguales se denomina escalado uniforme. Ahora, lee y contesta las siguientes cuestiones:

- ¿Qué ocurre si los factores de escala son diferentes entre sí?
- ¿Y si algún factor de escala es cero?
- ¿Qué ocurre si uno o varios factores de escala son negativos?
- ¿Y si el factor de escala está entre cero y uno?

3.1.3. Rotación

La transformación de rotación gira un punto un ángulo ϕ alrededor de un eje, y las representaciones matriciales con coordenadas homogéneas para los casos en los que el eje de giro coincida con uno de los ejes del sistema de coordenadas son las siguientes:

$$R_x(\phi) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.6}$$

$$R_y(\phi) = \begin{pmatrix} \cos\phi & 0 & \sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.7}$$

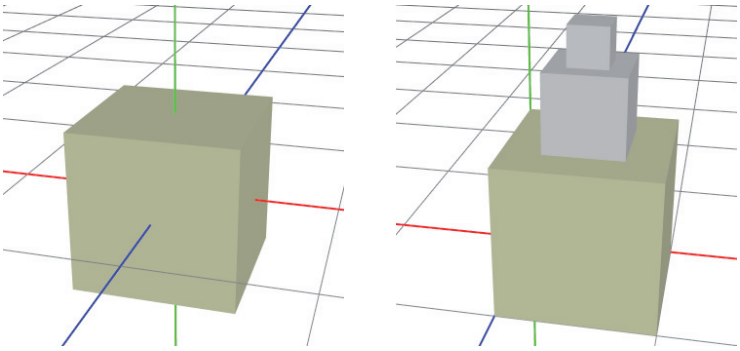
$$R_z(\phi) = \begin{pmatrix} \cos\phi & -\sin\phi & 0 & 0 \\ \sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.8}$$

Utilizando cualquiera de estas representaciones, el nuevo punto siempre se obtiene así: $\tilde{q} = R(\phi) \cdot \tilde{p}$.

Ejercicios

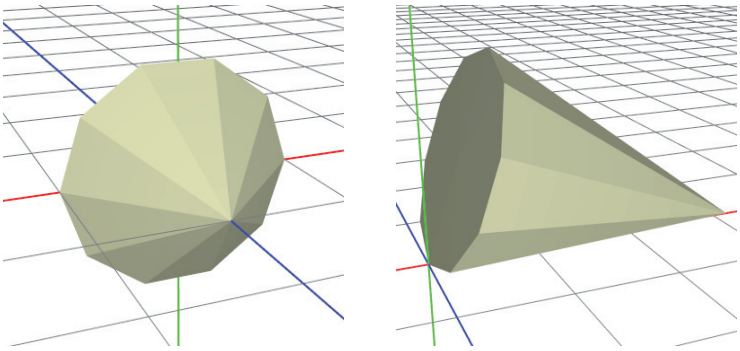
En los siguientes ejercicios el eje X es el de color rojo, el Y es el de color verde y el Z es el de color azul.

▶ **3.2** Comienza con un cubo de lado uno centrado en el origen de coordenadas tal y como se muestra en la figura (a). Usa dos cubos más como este y obtén el modelo que se muestra en la figura (b), donde cada nuevo cubo tiene una longitud del lado la mitad del cubo anterior. Detalla las transformaciones utilizadas.



(a) (b)

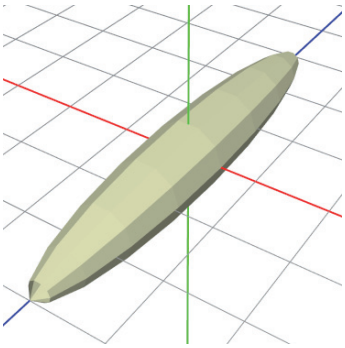
▶ **3.3** Determina las transformaciones que situan al cono que se muestra en la figura (c) (radio de la base y altura uno) en la posición que se muestra en la figura (d) (radio de la base uno y altura tres).



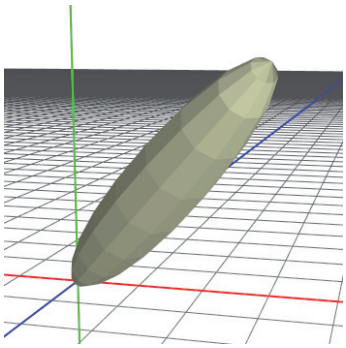
(c) (d)

▶ **3.4** Comienza con una esfera de radio uno centrada en el origen de coordenadas. La figura (e) muestra la esfera escalada con factores de escala $s_x = s_y = 0{,}5$ y $s_z = 3$. Obtén

las transformaciones que situan a la esfera tal y como se muestra en la figura (f) donde un punto final está en el origen y el otro en la recta $x = y = z$.



(e)



(f)

3.2. Concatenación de transformaciones

Una gran ventaja del uso de las transformaciones geométricas en su forma matricial con coordenadas homogéneas es que se pueden concatenar. De esta manera, una sola matriz puede representar a toda una secuencia de matrices de transformación.

Cuando se realiza la concatenación de transformaciones, es muy importante operar la secuencia de transformaciones en el orden correcto ya que el producto de matrices no posee la propiedad conmutativa. Por ejemplo, piensa en una esfera con radio unidad centrada en el origen de coordenadas y en las dos siguientes matrices de transformación T y R. $T(5, 0, 0)$ desplaza la componente x cinco unidades. $S(5, 5, 5)$ escala las tres componentes con un factor de cinco. Ahora, dibuja en el papel cómo quedaría la esfera después de aplicarle la matriz de transformación M si las matrices se multiplican de las dos formas posibles, es decir, $M = T \cdot S$ y $M = S \cdot T$. Como verás los resultados son bastante diferentes.

Por otra parte, el producto de matrices sí que posee la propiedad asociativa. Esto se puede aprovechar para reducir el número de operaciones aumentando así la eficiencia.

3.3. Matriz de transformación de la normal

La matriz de transformación es consistente para geometría y para vectores tangentes a la superficie. Sin embargo, dicha matriz no siempre es válida para los vectores normales a la superficie. Esto ocurre cuando se utilizan transformaciones de escalado no uniforme (ver figura 3.1). En este caso, lo habitual es que la matriz de transformación de la normal N sea la traspuesta de la inversa de la matriz de transformación. Sin embargo, la matriz inversa no siempre existe por lo que se

recomienda que la matriz N sea la traspuesta de la matriz adjunta. Además, como la normal es un vector y la traslación no le afecta, y el escalado y la rotación son transformaciones afines, solo hay que calcular la adjunta de los 3×3 componentes superior izquierda.

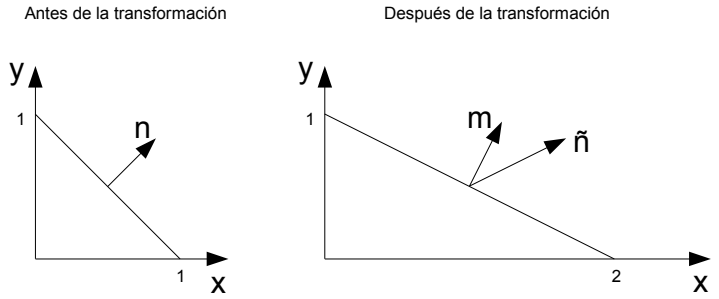


Figura 3.1: En la imagen de la izquierda se representa un polígono y su normal n. En la imagen de la derecha se muestra el mismo polígono tras aplicar una transformación de escalado no uniforme $S(2, 1)$. Si se aplica esta transformación a la normal n, se obtiene $\tilde{n}$ como vector normal en lugar de m que es la normal correcta.

Señalar por último que, después de aplicar la transformación, y únicamente en el caso de incluir escalado, las longitudes de las normales no se preservan, por lo que es necesario normalizarlas.

3.4. Giro alrededor de un eje arbitrario

Sean d y ϕ el vector unitario del eje de giro y el ángulo de giro respectivamente. Para realizar la rotación hay que calcular en primer lugar una base ortogonal que contenga d. La idea es hacer un cambio de base entre la base que forman los ejes de coordenadas y la nueva base, haciendo coincidir el vector d con, por ejemplo, el eje X, para entonces rotar ϕ grados alrededor de X y finalmente deshacer el cambio de base.

La matriz que representa el cambio de base es esta:

$$R = \begin{pmatrix} d_x & d_y & d_z \\ e_x & e_y & e_z \\ f_x & f_y & f_z \end{pmatrix} \tag{3.9}$$

donde e es un vector unitario normal a d, y f es el producto vectorial de los otros dos vectores $f = d \times e$. Esta matriz deja al vector d en el eje X, al vector e en el eje Y y al vector f en el eje Z. El vector e se puede obtener de la siguiente manera: partiendo del vector d haz cero su componente de menor magnitud (el más pequeño en valor absoluto), intercambia los otros dos componentes, niega uno de ellos y normalízalo.

Así, teniendo en cuenta que R es ortogonal y que por lo tanto su inversa coincide con la traspuesta, la matriz de rotación final es:

$$R_d(\phi) = R^T R_x(\phi) R \tag{3.10}$$

3.5. Transformaciones en OpenGL

Como ayuda a la programación, la librería GLM proporciona funciones tanto para la construcción de las matrices de transformación como para operar con ellas. En concreto, las siguientes funciones permiten construir, respectivamente, las matrices de traslación, escalado y giro alrededor de un eje arbitrario que pasa por el origen:

- mat4 translate (float t_x, float t_y, float t_z)
- mat4 scale (float s_x, float s_y, float s_z)
- mat4 rotate (float α, float x, float y, float z)

Así, por ejemplo, la matriz de transformación M para que un objeto se escale al doble de su tamaño y después se traslade en dirección del eje X un total de diez unidades se obtendría de la siguiente forma:

```
mat4 T, S, M;
T = translate (10.0f, 0.0f, 0.0f);
S = scale (2.0f, 2.0f, 2.0f);
M = T*S;
```

O también así:

```
mat4 M (translate (10.0f, 0.0f, 0.0f)*scale (2.0f, 2.0f, 2.0f));
```

Para el cálulo de la matriz normal N, utilizando GLM se puede hacer por ejemplo lo siguiente:

```
mat3 N (mat3(transpose(inverse(M))));
```

La construcción de ambas matrices, la matriz de transformación del modelo y la matriz de transformación de la normal, es conveniente que tenga lugar en la aplicación y que ambas se suministren al procesador de vértices para que en el *shader*

sólo se tenga que multiplicar cada vértice y cada normal por su correspondiente matriz. También en dicho procesador hay que normalizar la normal resultante de la operación de transformación. El listado 3.1 muestra cómo realizar estas operaciones en el *vertex shader*.

Listado 3.1: *Shader* para transformar la posición y la normal de cada vértice

```
uniform mat4 M; // matriz de transformación del modelo
uniform mat3 N; // matriz de transformación de la normal

in  vec3 posicion, vNormal;
out vec3 normal;

void main ()
{
  normal = normalize (N * vNormal);
  gl_Position = M * vec4(posicion, 1.0);
}
```

Algunos ejemplos de resultados obtenidos mediante la aplicación de transformaciones geométricas a primitivas geométricas simples como el cubo, la esfera o el toro, se muestran en la figura 3.2.

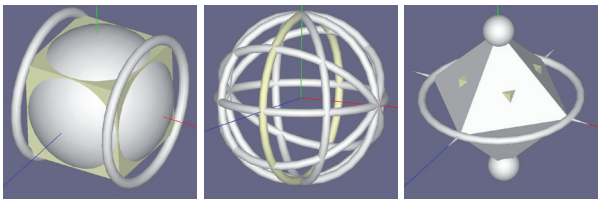


Figura 3.2: Ejemplos de objetos creados utilizando transformaciones geométricas

Ejercicios

► **3.5** Modela la típica grúa de obra cuyo esquema se muestra en la figura 3.3 utilizando como única primitiva cubos de lado uno centrados en el origen de coordenadas. La carga es de lado 1, el contrapeso es de lado 1,4, y tanto el pie como el brazo tienen una longitud de 10. Incluye las transformaciones que giran la grúa sobre su pie, que desplazan la carga a lo largo del brazo, y que levantan y descienden la carga.

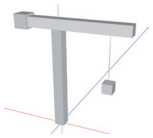


Figura 3.3: Esquema de una grúa de obra

Capítulo 4

Viendo en 3D

Al igual que en el mundo real se utiliza una cámara para conseguir fotografías, en nuestro mundo virtual también es necesario definir un modelo de cámara que permita obtener vistas 2D de nuestro mundo 3D. El proceso por el que la cámara sintética obtiene una fotografía se implementa como una secuencia de tres transformaciones:

- Transformación de la cámara: ubica la cámara virtual en el origen del sistema de coordenadas orientada de manera conveniente.
- Transformación de proyección: determina cuánto del mundo 3D es visible, a este espacio limitado se le denomina *volumen de la vista*, y proyecta el contenido del volumen en un plano 2D.
- Transformación al área de dibujo: consiste en mover el resultado de la transformación de proyección al espacio de la ventana destinado a mostrar la vista 2D.

4.1. Transformación de la cámara

La posición de una cámara, el lugar desde el que se va a tomar la fotografía, se establece especificando un punto p del espacio 3D. Una vez posicionada, la cámara se orienta de manera que su objetivo quede apuntando a un punto específico de la escena. A este punto i se le conoce con el nombre de *punto de interés*. En general, los fotográfos utilizan la cámara para hacer fotos apaisadas u orientadas en vertical, aunque tampoco resulta extraño ver fotografías realizadas con otras inclinaciones. Esta inclinación se establece mediante el vector UP denominado *vector de inclinación*. Con estos tres datos queda perfectamente posicionada y orientada la cámara tal y como se muestra en la figura 4.1.

Algunas de las operaciones que los sistemas gráficos realizan requieren que la cámara esté situada en el origen de coordenadas apuntando en la dirección del eje Z negativo y coincidiendo el vector de inclinación con el eje Y positivo. Por esta

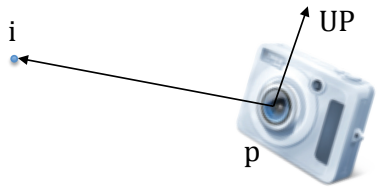


Figura 4.1: Parámetros para ubicar y orientar una cámara: p, posición de la cámara; UP, vector de inclinación; i, punto de interés

razón, es necesario aplicar una transformación al mundo 3D de manera que, desde la posición y orientación requerida por el sistema gráfico, se observe lo mismo que desde donde el usuario estableció su cámara. A esta transformación se le denomina *transformación de la cámara*.

Si F es el vector normalizado que desde la posición de la cámara apunta al punto de interés, UP' es el vector de inclinación normalizado, $S = F \times UP'$ y $U = S \times F$, entonces el resultado de la siguiente operación crea la matriz de transformación M_C que sitúa la cámara en la posición y orientación requerida por el sistema gráfico:

$$M_C = \begin{pmatrix} S_x & S_y & S_z & 0 \\ U_x & U_y & U_z & 0 \\ -F_x & -F_y & -F_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & -p_x \\ 0 & 1 & 0 & -p_y \\ 0 & 0 & 1 & -p_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.1}$$

4.2. Transformación de proyección

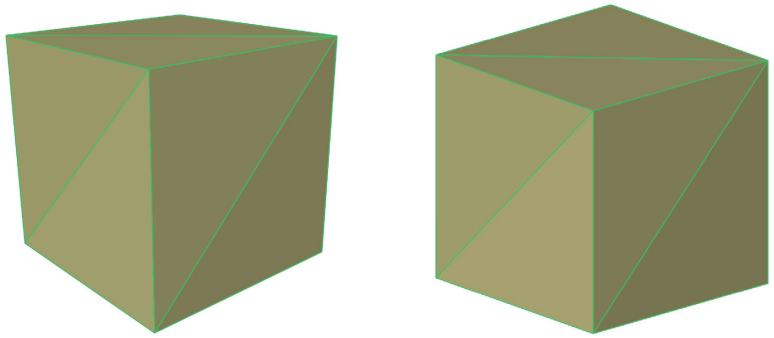
El volumen de la vista determina la parte del mundo 3D que puede ser vista por el observador. La forma y dimensión de este volumen depende del tipo de proyección. Así, hay dos tipos de vistas:

- Vista perspectiva: similar a como funciona nuestra vista y se utiliza para generar imágenes más fieles a la realidad en aplicaciones como videojuegos, simulaciones o, en general, la mayor parte de aplicaciones gráficas.
- Vista paralela: utilizada principalmente en ingeniería o arquitectura, y se caracteriza por preservar longitudes y ángulos.

La figura 4.2 muestra un ejemplo de un cubo dibujado con ambos tipos de vistas.

4.2.1. Proyección paralela

Este tipo de proyección se caracteriza por que los rayos de proyección son paralelos entre sí e intersectan de forma perpendicular con el plano de proyección.



(a) (b)

Figura 4.2: Vista de un cubo obtenida con: (a) vista perspectiva y (b) vista paralela

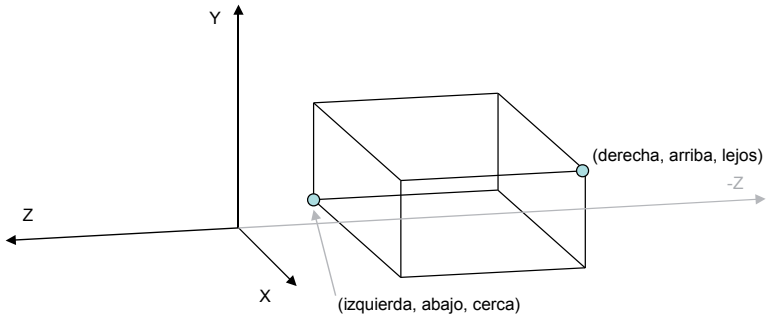


Figura 4.3: Esquema del volumen de la vista de una proyección paralela

El volumen de la vista tiene forma de caja que se alinea con los ejes de coordenadas tal y como se muestra en la figura 4.3, donde también se han indicado los nombres de los seis parámetros que definen dicho volumen.

En general, los sistemas gráficos trasladan y escalan esa caja de manera que la convierten en un cubo centrado en el origen de coordenadas. A este cubo se le denomina *volumen canónico de la vista* y a las coordenadas en este volumen *coordenadas normalizadas del dispositivo*. La matriz de transformación correspondiente para un cubo de lado dos es la siguiente:

$$M_{par} = \begin{pmatrix} \frac{2}{derecha-izquierda} & 0 & 0 & -\frac{derecha+izquierda}{derecha-izquierda} \\ 0 & \frac{2}{arriba-abajo} & 0 & -\frac{arriba+abajo}{arriba-abajo} \\ 0 & 0 & \frac{2}{lejos-cerca} & -\frac{lejos+cerca}{lejos-cerca} \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.2}$$

4.2.2. Proyección perspectiva

Este tipo de proyección se caracteriza por que los rayos de proyección parten todos ellos desde la posición del observador. El volumen de la vista tiene forma de pirámide, que queda definida mediante cuatro parámetros: los planos cerca y lejos (los mismos que en la proyección paralela), el ángulo θ en la dirección Y y la relación de aspecto de la base de la pirámide $ancho/alto$. En la figura 4.4 se detallan estos parámetros.

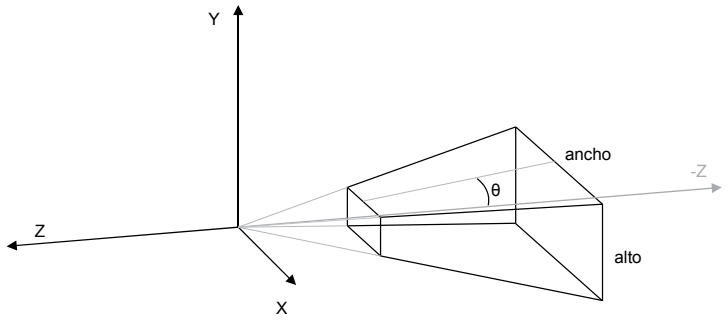


Figura 4.4: Esquema del volumen de la vista de una proyección perspectiva

En general, los sistemas gráficos convierten ese volumen con forma de pirámide en el volumen canónico de la vista. La matriz de transformación correspondiente para un cubo de lado dos es la siguiente:

$$M_{per} = \begin{pmatrix} \frac{aspect}{tan(\theta)} & 0 & 0 & 0 \\ 0 & \frac{1}{tan(\theta)} & 0 & 0 \\ 0 & 0 & \frac{lejos+cerca}{cerca-lejos} & \frac{2 \cdot lejos \cdot cerca}{cerca-lejos} \\ 0 & 0 & -1 & 0 \end{pmatrix} \tag{4.3}$$

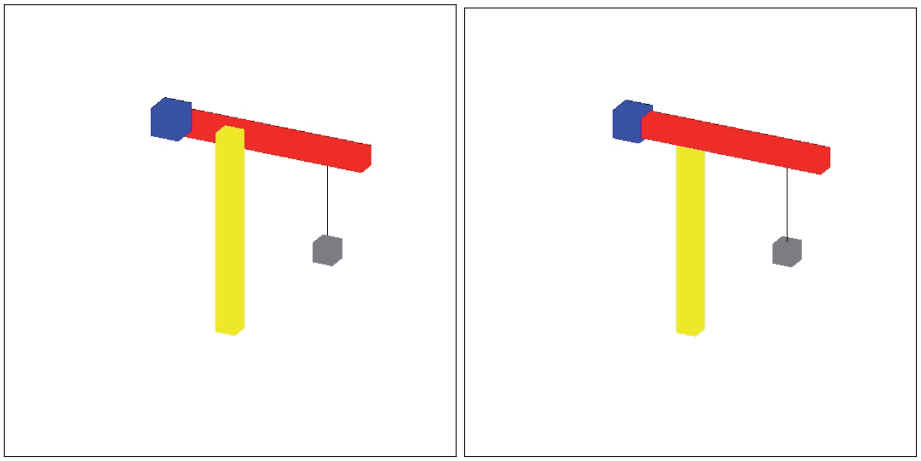
4.3. Transformación al área de dibujo

El área de dibujo, también conocida por su término en inglés *viewport*, es la parte de la ventana de la aplicación donde se muestra la vista 2D. La transformación al *viewport* consiste en mover el resultado de la proyección a dicha área. Se asume que la geometría a visualizar reside en el volumen canónico de la vista, es decir, se cumple que las coordenadas de todos los puntos $(x, y, z) \in [-1, 1]^3$. Entonces, si n_x y n_y son respectivamente el ancho y el alto del *viewport* en píxeles, y o_x y o_y son el píxel de la esquina inferior izquierda del *viewport* en coordenadas de ventana, para cualquier punto que resida en el volumen canónico de la vista, sus coordenadas de ventana se obtienen con la siguiente transformación:

$$M_{vp} = \begin{pmatrix} \frac{n_x}{2} & 0 & 0 & \frac{n_x-1}{2} + o_x \\ 0 & \frac{n_y}{2} & 0 & \frac{n_y-1}{2} + o_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.4}$$

4.4. Eliminación de partes ocultas

La eliminación de partes ocultas consiste en determinar qué primitivas de la escena son tapadas por otras primitivas desde el punto de vista del observador (ver figura 4.5). Aunque para resolver este problema se han desarrollado diversos algoritmos, en la práctica el más utilizado es el algoritmo conocido como *Z-Buffer*.



(a) (b)

Figura 4.5: Ejemplo de escena visualizada: (a) sin resolver el problema de la visibilidad y (b) con el problema resuelto

Este algoritmo se caracteriza por su simplicidad. Para cada píxel de la primitiva que se está dibujando, su valor de profundidad (su coordenada z) se compara con el valor almacenado en un *buffer* denominado *buffer de profundidad*. Si la profundidad de la primitiva para dicho píxel es menor que el valor almacenado en el *buffer* para ese mismo píxel, tanto el *buffer* de color como el de profundidad se actualizan con los valores de la nueva primitiva, siendo eliminado en cualquier otro caso.

El algoritmo se muestra en el listado 4.1. En este algoritmo no importa el orden en que se pinten las primitivas, pero sí es muy importante que el *buffer* de profundidad se inicialice siempre al valor de profundidad máxima antes de pintar la primera primitiva.

Listado 4.1: Algoritmo del *Z-Buffer*

```
if (pixel.z < bufferProfundidad(x,y).z)
{
  bufferProfundidad(x,y).z = pixel.z;
  bufferColor(x,y).color   = pixel.color;
}
```

4.5. Viendo en OpenGL

En OpenGL es responsabilidad del programador construir la matriz de transformación de la cámara y la matriz de proyección. Ambas matrices se han de suministrar al procesador de vértices donde cada vértice v debe ser multiplicado por dichas matrices. Si M_M es la matriz de transformación del modelo, el nuevo vértice v' se obtiene así: $v' = M_{Proy} \cdot M_C \cdot M_M \cdot v$; donde M_{Proy} será M_{Par} o M_{Per}.

La librería GLM proporciona diversas funciones para construir las matrices vistas en este capítulo. Así, la función *lookAt* construye la matriz resultado de la ecuación 4.1 a partir de la posición de la cámara p, el punto de interés i y el vector de inclinación UP. Además, las funciones *ortho* y *perspective* construyen las matrices de proyección paralela y perspectiva respectivamente. Son estas:

- mat4 lookAt (vec3 p, vec3 i, vec3 UP)
- mat4 ortho (float izquierda, float derecha, float abajo, float arriba, float cerca, float lejos)
- mat4 perspective (float θ, float ancho/alto, float cerca, float lejos)

Independientemente del tipo de proyección utilizada, los elementos que quedan fuera del volumen de la vista son eliminados en la etapa conocida con el nombre de recortado. Esta etapa se situa entre el procesador de vértices y el de fragmentos, y forma parte de la funcionalidad fija de la GPU. Si una primitiva intersecta con el volumen de la vista de manera que parte de él queda dentro y parte de él queda fuera, se recorta de manera que al procesador de fragmentos le llegan únicamente los trozos situados dentro del volumen de la vista.

Como el resultado de la proyección perspectiva puede hacer que la coordenada w (la cuarta coordenada del vértice) sea distinta de uno, cada vértice debe ser dividido por w, proceso conocido con el nombre de *división perspectiva*. Esta tarea también la realiza la GPU de forma fija después de la etapa de recortado.

Respecto a la transformación al área de dibujo, esta la realiza la GPU de forma fija después de la división perspectiva, en una etapa también previa al procesador de fragmentos. El programador sólo debe especificar la ubicación del *viewport* en la ventana mediante la orden *glViewport*, indicando la esquina inferior izquierda, el ancho y el alto en coordenadas de ventana:

- void glViewport (int x, int y, int $ancho$, int $alto$)

La llamada a la función *glViewport* debe realizarse cada vez que se produzca un cambio en el tamaño de la ventana de la aplicación con el fin de mantener el *viewport* actualizado. Además, es importante que la relación de aspecto del *viewport* sea igual a la relación de aspecto utilizada al definir el volumen de la vista para no deformar el resultado de la proyección.

Respecto a la eliminación de partes ocultas, OpenGL implementa el algoritmo del *Z-Buffer* y la GPU comprueba la profundidad de cada fragmento de forma fija en una etapa posterior al procesador de fragmentos. A esta comprobación se le denomina test de profundidad. Sin embargo, el programador aún debe realizar tres tareas:

1. Solicitar la creación del *buffer* de profundidad al crear el contexto en el entorno gráfico. Esta tarea la facilita la librería FreeGLUT y se solicita de la siguiente manera:
   - glutInitDisplayMode (... | GLUT_DEPTH)
2. Habilitar la operación del test de profunidad:
   - glEnable (GL_DEPTH_TEST)
3. Inicializar el *buffer* a la profundidad máxima antes de comenzar a dibujar:
   - glClear (... | GL_DEPTH_BUFFER_BIT)

Ejercicios

► 4.1 Dibuja un esquema del *pipeline* de OpenGL en el que se incluyan las etapas comentadas en esta sección: recortado, división perspectiva, transformación al área de dibujo y test de profundidad.

► 4.2 Modifica el programa de la grúa de obra realizado en el capítulo anterior para que el usuario pueda obtener cuatro vistas con proyección paralela, tres de ellas con dirección paralela a los tres ejes de coordenadas y la cuarta en dirección (1,1,1). En todas ellas la grúa debe observarse en su totalidad.

► 4.3 Amplía la solución del ejercicio anterior para que se pueda cambiar de proyección paralela a perspectiva, y viceversa, y que sin modificar la matriz de transformación de la cámara se siga observando la grúa completa.

► 4.4 Amplía la solución del ejercicio anterior para que se realice la eliminación de las partes ocultas.

Capítulo 5

Modelos de iluminación y sombreado

Un modelo de iluminación determina el color de la superficie en un punto. Un modelo de sombreado utiliza un modelo de iluminación y especifica cuándo usarlo.

5.1. Modelo de iluminación de Phong

En esta sección se describe el modelo de iluminación de Phong. Este modelo tiene en cuenta los tres aspectos siguientes:

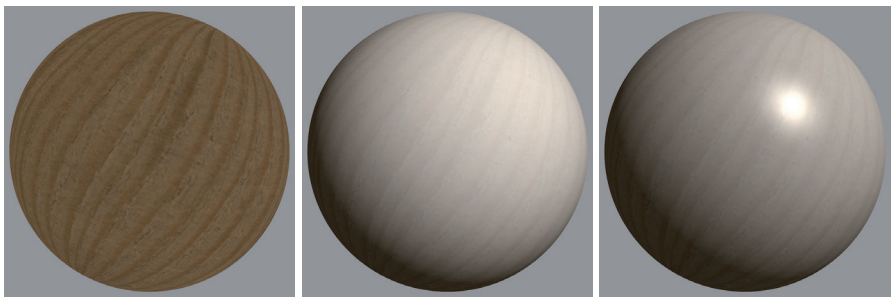
- Luz ambiente: luz que proporciona iluminación uniforme a lo largo de la escena.
- Reflexión difusa: luz reflejada por la superficie en todas las direcciones.
- Reflexión especular: luz reflejada por la superficie en una sola dirección o en un rango de ángulos muy cercano al ángulo de reflexión perfecta.

5.1.1. Luz ambiente

La luz ambiente I_a que se observa en cualquier punto de una superficie es siempre la misma. Parte de la luz que llega a un objeto es absorbida por este, y parte es reflejada, la cual se modela con el coeficiente k_a, $0 \leq k_a \leq 1$. Si L_a es la luz ambiente, entonces:

$$I_a = k_a L_a \tag{5.1}$$

La figura 5.1(a) muestra un ejemplo en el que el modelo de iluminación únicamente incluye luz ambiente.



(a) (b) (c)

Figura 5.1: Ejemplos de iluminación: (a) Solo luz ambiente; (b) Luz ambiente y reflexión difusa; (c) Luz ambiente, reflexión difusa y especular

5.1.2. Reflexión difusa

La reflexión difusa es característica de superficies rugosas, mates, sin brillo. Este tipo de superficies se puede modelar fácilmente con la Ley de Lambert. Así, el brillo observado en un punto depende sólo del ángulo θ, $0 \leq \theta \leq 90$, entre la dirección a la fuente de luz L y la normal N de la superficie en dicho punto (ver figura 5.2). Si L y N son vectores unitarios y k_d, $0 \leq k_d \leq 1$, representa la parte de luz difusa reflejada por la superficie, la ecuación que modela la reflexión difusa es la siguiente:

$$I_d = k_d L_d \cos\theta = k_d L_d (L \cdot N) \tag{5.2}$$

Para tener en cuenta la atenuación que sufre la luz al viajar desde su fuente de origen hasta la superficie del objeto situado a una distancia d, se propone utilizar la siguiente ecuación donde los coeficientes a, b y c son constantes características de la fuente de luz:

$$I_d = \frac{k_d}{a + bd + cd^2} L_d (L \cdot N) \tag{5.3}$$

La figura 5.1(b) muestra un ejemplo en el que el modelo de iluminación incluye luz ambiente y reflexión difusa.

5.1.3. Reflexión especular

Este tipo de reflexión es propia de superficies brillantes, pulidas, y responsable de los brillos que suelen observarse en esos tipos de superficies. El color del brillo suele ser diferente del color de la superficie y muy parecido al color de la fuente de luz. Además, la posición de los brillos depende de la posición del observador.

Phong propone que la luz que llega al observador dependa únicamente del ángulo Φ entre el vector de reflexión perfecta R y el vector dirección del observador V (ver figura 5.2). Si R y V son vectores unitarios, k_s, $0 \leq k_s \leq 1$, representa

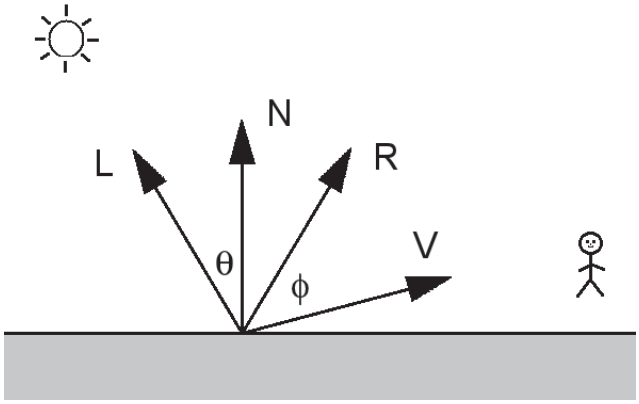


Figura 5.2: Geometría del modelo de iluminación de Phong

la parte de luz especular reflejada por la superficie y α modela el brillo característico del material de la superficie, la ecuación que modela la reflexión especular es la siguiente:

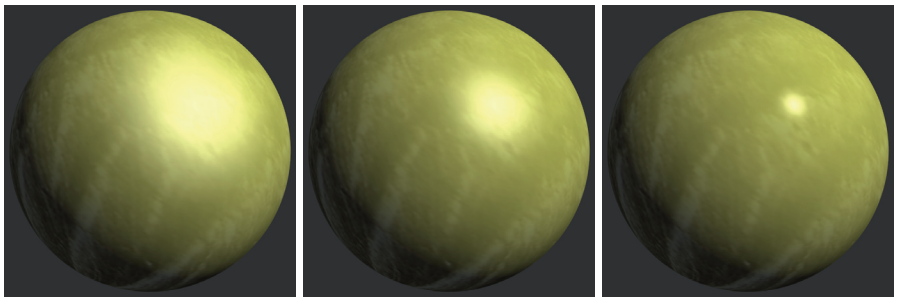
$$I_s = k_s L_s \cos^{\alpha} \Phi = k_s L_s (R \cdot V)^{\alpha} \tag{5.4}$$

donde R se obtiene de la siguiente manera:

$$R = 2N(N \cdot L) - L \tag{5.5}$$

La figura 5.1(c) muestra un ejemplo en que el modelo de iluminación incluye luz ambiente, reflexión difusa y especular.

Respecto al valor de α, un valor igual a 1 modela un brillo grande, mientras que valores mucho mayores, por ejemplo entre 100 y 500, modelan brillos más pequeños propios de materiales, por ejemplo, metálicos. La figura 5.3 muestra varios ejemplos obtenidos con distintos valores de α.



(a) $\alpha = 3$ (b) $\alpha = 10$ (c) $\alpha = 100$

Figura 5.3: Ejemplos de iluminación con diferentes valores de α para el cálculo de la reflexión especular

5.1.4. Materiales

El modelo de iluminación de Phong tiene en cuenta las propiedades del material del objeto al calcular la iluminación y así proporcionar mayor realismo. En concreto son cuatro: ambiente k_a, difusa k_d, especular k_s y brillo α. La tabla 5.1 muestra una lista de materiales con los valores de ejemplo para estas constantes.

Esmeralda	k_a[]= { 0.022, 0.17, 0.02, 1.0 } k_d[]= { 0.08, 0.61, 0.08 ,1.0 } k_s[]= { 0.63, 0.73, 0.63, 1.0 } α[]= { 0.6 }	Jade	k_a[]= { 0.14, 0.22, 0.16 , 1.0 } k_d[]= { 0.54, 0.89, 0.63, 1.0 } k_s[]= { 0.32, 0.32, 0.32 , 1.0 } α[]= { 0.10 }
Obsidiana	k_a[]= { 0.05, 0.05, 0.07, 1.0 } k_d[]= { 0.18, 0.17, 0.23, 1.0 } k_s[]= { 0.33, 0.33, 0.35, 1.0 } α[]= { 0.30 }	Perla	k_a[]= { 0.25, 0.21, 0.21, 1.0 } k_d[]= { 1.0, 0.83, 0.83, 1.0 } k_s[]= { 0.30, 0.30, 0.30, 1.0 } α[]= { 0.09 }
Rubí	k_a[]= { 0.18, 0.01, 0.01, 1.0 } k_d[]= { 0.61, 0.04, 0.04, 1.0 } k_s[]= { 0.73, 0.63, 0.63, 1.0 } α[]= { 0.60 }	Turquesa	k_a[]= { 0.10, 0.19, 0.17, 1.0 } k_d[]= { 0.39, 0.74, 0.69, 1.0 } k_s[]= { 0.29, 0.31, 0.31, 1.0 } α[]= { 0.10 }
Bronce	k_a[]= { 0.21, 0.13, 0.05, 1.0 } k_d[]= { 0.71, 0.43, 0.18, 1.0 } k_s[]= { 0.39, 0.27, 0.17, 1.0 } α[]= { 0.20 }	Cobre	k_a[]= { 0.19, 0.07, 0.02, 1.0 } k_d[]= { 0.71, 0.27, 0.08, 1.0 } k_s[]= { 0.26, 0.14, 0.09, 1.0 } α[]= { 0.10 }
Oro	k_a[]= { 0.25, 0.20, 0.07, 1.0 } k_d[]= { 0.75, 0.61, 0.23, 1.0 } k_s[]= { 0.63, 0.56, 0.37, 1.0 } α[]= { 0.40 }	Plata	k_a[]= { 0.20, 0.20, 0.20, 1.0 } k_d[]= { 0.51, 0.51, 0.51, 1.0 } k_s[]= { 0.51, 0.51, 0.51, 1.0 } α[]= { 0.40 }
Plástico	k_a[]= { 0.0, 0.0, 0.0 , 1.0 } k_d[]= { 0.55, 0.55, 0.55 ,1.0 } k_s[]= { 0.70, 0.70, 0.70 , 1.0 } α[]= { 0.25 }	Goma	k_a[]= { 0.05, 0.05, 0.05, 1.0 } k_d[]= { 0.50, 0.50, 0.50, 1.0 } k_s[]= { 0.70, 0.70, 0.70, 1.0 } α[]= { 0.08 }

Tabla 5.1: Ejemplos de propiedades de material para el modelo de Phong

5.1.5. El modelo de Phong

A partir de las ecuaciones 5.1, 5.3 y 5.4, se define el modelo de iluminación de Phong como:

$$I = k_a L_a + \frac{1}{a + bd + cd^2}(k_d L_d (L \cdot N) + k_s L_s (R \cdot V)^{\alpha}) \tag{5.6}$$

En el listado 5.1 se muestra la función que calcula la iluminación en un punto sin incluir el factor de atenuación. En el caso de tener múltiples fuentes de luz, hay que sumar los términos de cada una de ellas:

$$I = k_a L_a + \sum_{1 \leq i \leq m} \frac{1}{a_i + b_i d + c_i d^2}(k_d L_{di} (L_i \cdot N) + k_s L_{si} (R_i \cdot V)^{\alpha_i}) \tag{5.7}$$

Listado 5.1: Función que implementa para una fuente de luz el modelo de iluminación de Phong sin incluir el factor de atenuación

```
// Ka, Kd, Ks, alfa, La, Ld y Ls son variables uniformes
// N, L y V se asumen normalizados
vec4 phong (vec3 N, vec3 L, vec3 V)
{
  float NdotL = dot(N,L);
  vec4  color = Ka * La;
  if (NdotL > 0.0)
  {
    vec3  R       = normalize(2 * N * NdotL - L);
    float RdotV_n = pow(max(0.0, dot(R,V)), alfa);

    color = color + ((NdotL * (Ld * Kd)) + (RdotV_n * (Ls * Ks)));
  }
  return color;
}
```

5.2. Tipos de fuentes de luz

En general, siempre se han considerado dos tipos de fuentes de luz dependiendo de su posición:

- Posicional: la fuente emite luz en todas las direcciones desde un punto dado, muy parecido a como por ejemplo ilumina una bombilla.
- Direccional: la fuente está ubicada en el infinito, todos los rayos de luz son paralelos y viajan en la misma dirección. En este caso el vector L en el modelo de iluminación de Phong es constante.

En ocasiones se desea restringir los efectos de una fuente de luz posicional a un área limitada de la escena, tal y como haría por ejemplo una linterna. A este tipo de fuente posicional se le denomina foco (ver figura 5.4). A diferencia de una luz posicional general, un foco viene dado además de por su posición, por la dirección S y el ángulo δ que determinan la forma del cono tal y como se muestra en la figura 5.5.

Así, un fragmento es iluminado por el foco sólo si está dentro del cono de luz. Esto se averigua calculando el ángulo entre el vector L y el vector S. Si el resultado es mayor que el ángulo δ es que ese fragmento está fuera del cono siendo, en concecuencia, afectado únicamente por la luz ambiente.

Además, se puede considerar que la intensidad de la luz decae a medida que los rayos se separan del eje del cono. Esta atenuación se calcula mediante el coseno del ángulo entre los vectores L y S elevado a un exponente. Cuanto mayor sea este exponente, mayor será la concentración de luz alrededor del eje del cono. Por ejemplo, en la figura 5.4 se observa perfectamente en la imagen iluminada con el foco esta atenuación.

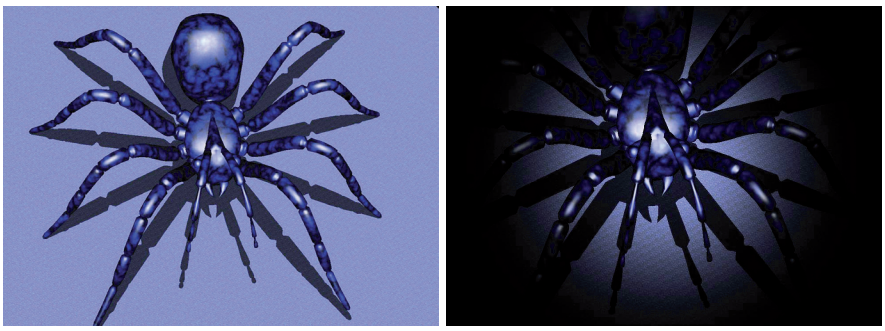


Figura 5.4: Ejemplo de escena iluminada: a la izquierda, con una luz posicional, y a la derecha, con la fuente convertida en foco

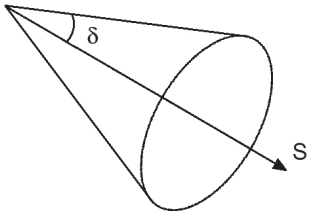


Figura 5.5: Parámetros característicos de un foco de luz

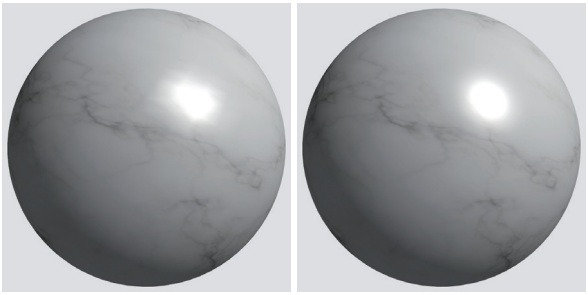
Finalmente, el factor de atenuación calculado se incorpora al modelo de iluminación de Phong multiplicando al factor de atenuación que ya existía.

5.3. Modelos de sombreado

Dado un polígono, y dado un modelo de iluminación, hay tres métodos para determinar el color de cada fragmento:

- Plano: el modelo de iluminación se aplica una sola vez y su resultado se aplica a toda la superficie del polígono. Este método requiere la normal de cada polígono.
- Gouraud: el modelo de iluminación se aplica en cada vértice del polígono y los resultados se interpolan sobre su superficie. Este método requiere la normal en cada uno de los vértices del polígono. Un ejemplo del resultado obtenido se muestra en la figura 5.6(a). El listado 5.2 muestra el *shader* correspondiente a este modelo de sombreado.
- Phong: el modelo de iluminación se aplica para cada fragmento. Este método requiere la normal en el fragmento, que se puede obtener por interpolación de las normales de los vértices. Un ejemplo del resultado obtenido se muestra

en la figura 5.6(b). El listado 5.3 muestra el *shader* correspondiente a este modelo de sombreado.



(a) Gouraud (b) Phong

Figura 5.6: Ejemplos de modelos de sombreado: (a) Gouraud; (b) Phong

Listado 5.2: *Shader* para realizar un sombreado de Gouraud

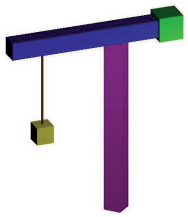
```
// Vertex shader ------------------------------------------------------
uniform mat4  projection, camera, transf; // matrices
uniform mat3  normal;
uniform vec4  Ka, Kd, Ks;                 // material
uniform float alfa;
uniform vec4  Lp, La, Ld, Ls;             // fuente de luz

in  vec3 posicion, vNormal;      // recibe vértice y normal
out vec4 nuevoColor;             // color del vértice

void main()
{
  vec4 ecPosition = camera*transf*vec4(posicion,1.0);
  vec3 N          = normalize(normal * vNormal);
  vec3 L          = normalize(vec3(Lp - ecPosition));
  vec3 V          = normalize(vec3(-ecPosition));

  nuevoColor  = phong(N, L, V);
  gl_Position = projection * ecPosition;
}

// Fragment shader ----------------------------------------------------
in  vec4 nuevoColor;
out vec4 colorFragmento;

void main()
{
  colorFragmento = nuevoColor;
}
```

Listado 5.3: *Shader* para realizar un sombreado de Phong

```
// Vertex shader ------------------------------------------------------
uniform mat4 projection, camera, transf;
uniform mat3 normal;
uniform vec4 Lp;                    // fuente de luz

in  vec3 posicion, vNormal;         // recibe vértice y normal
out vec3 N, L, V;                   // los vectores

void main()
{
  vec4 ecPosition = camera*transf*vec4(posicion,1.0);

  N = normalize (normal * vNormal);
  L = vec3 (normalize(Lp - ecPosition));
  V = normalize (vec3(-ecPosition));

  gl_Position = projection * ecPosition;
}

// Fragment shader ----------------------------------------------------
uniform vec4  Ka, Kd, Ks;           // material
uniform float alfa;
uniform vec4  La, Ld, Ls;           // fuente de luz

in  vec3 N, L, V;
out vec4 colorFragmento;

void main()
{
  colorFragmento = phong (N, L, V);
}
```

Ejercicios

► **5.1** Añade al menos una fuente de luz y distintas propiedades de material a los objetos de la escena de la grúa de obra realizada en ejercicios anteriores. Utiliza el modelo de sombreado de Gouraud y el de Phong para observar las diferencias en los resultados.



Capítulo 6

Aplicación de texturas 2D

En el capítulo anterior se mostró cómo un modelo de iluminación aumenta el realismo visual de las imágenes generadas por computador. Ahora, se va a mostrar cómo se puede utilizar una imagen 2D a modo de mapa de color de manera que el valor definitivo de un determinado píxel dependa de ambos, es decir, de la iluminación de la escena y de la textura. El uso de texturas para aumentar el realismo visual de las aplicaciones es muy frecuente, por lo que el número de técnicas en la literatura es también muy elevado. En concreto, en este capítulo se presenta cómo utilizar OpenGL para aplicar una textura 2D sobre una superficie (ver figura 6.1).

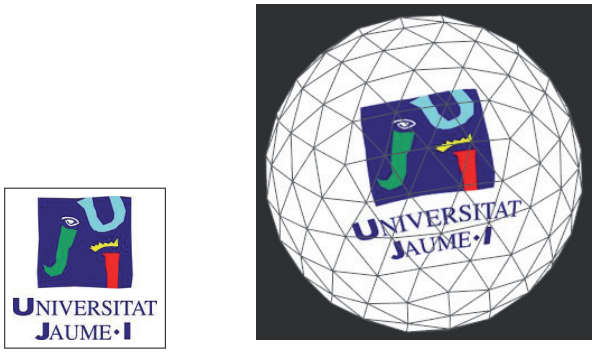


Figura 6.1: Resultado de aplicación de una textura 2D a un objeto 3D

6.1. Introducción

Los pasos para aplicar una textura sobre una superficie son los siguientes:

1. Crear un objeto textura.
2. Asignar la unidad de textura.
3. Especificar para cada vértice de la superficie sus coordenadas de textura.

OpenGL no proporciona ningún método para cargar una imagen en memoria. El programador debe utilizar las librerías adecuadas dependiendo del formato de la imagen. Una forma sencilla de leer un fichero de imagen es convertirlo primero a formato RGB y utilizar la rutina que se muestra en el listado 6.1.

Listado 6.1: Lectura de un archivo de imagen en formato RGB

```
GLubyte *leeTextura (char *nombreFichero, int ancho, int alto) {

  GLubyte *textura = (GLubyte*) malloc (
                     sizeof(GLubyte)*3*ancho*alto);
  FILE    *fichero;

  fichero = fopen (nombreFichero, "rb");

  fread  (textura, sizeof(GLubyte), ancho*alto*3, fichero);
  fclose (fichero);
  return textura;
}
```

6.2. Crear un objeto textura

En OpenGL las texturas se representan mediante objetos con nombre. El nombre no es más que un entero sin signo donde el valor cero está reservado. Para crear objetos textura es necesario obtener en primer lugar nombres que no se estén utilizando. Esta solicitud se realiza con la orden *glGenTextures*. Se han de solicitar tantos nombres como objetos textura necesitemos, teniendo en cuenta que un objeto textura sólo almacena una única textura.

Una vez obtenidos los nombres, se crean uno a uno los objetos textura asignando el nombre obtenido mediante la orden *glBindTexture*. Para eliminar un objeto textura existe la orden *glDeleteTextures*, que permite eliminar varios objetos textura con una única llamada. En el listado 6.2 se muestra un ejemplo en el que se crea un objeto textura.

6.2.1. Especificar una textura

A cada objeto textura hay que especificarle tanto la textura, la imagen 2D en nuestro caso, como los diferentes parámetros de aplicación. Para especificar la textura se utiliza la siguiente orden:

- glTexImage2D (GLenum $objetivo$, GLint $nivel$, GLint $formatoInterno$, GLsizei $ancho$, GLsizei $alto$, GLint, $borde$, GLenum $formato$, GLenum $tipo$, const void *$datos$);

donde objetivo será *GL_TEXTURE_2D*. Los parámetros *formato*, *tipo* y *datos* especifican, respectivamente, el formato de los datos de la imagen, el tipo de esos

datos y una referencia a los datos de la imagen. El parámetro *nivel* se utiliza sólo en el caso de usar diferentes resoluciones de la textura, siendo cero en cualquier otro caso. El parámetro *formatoInterno* se utiliza para indicar cuáles de las componentes R, G, B y A se emplean como texeles de la imagen. Por último, *ancho* y *alto* son las dimensiones de la textura, y *borde* ha de ser cero desde la versión 3.1 de OpenGL. Observa de nuevo el listado 6.2 para ver un ejemplo de uso de esta orden.

6.2.2. Especificar parámetros

Los parámetros permiten controlar el modo en que se aplica la textura sobre la superficie. En concreto se tratan dos aspectos: la repetición y el filtrado. Para ambos se utiliza la misma orden, *glTexParamenteri*. El primer parámetro de esta orden es en ambos casos *GL_TEXTURE_2D*. El segundo es el que se desea especificar, y su valor se indica a continuación como tercer y último parámetro.

Para especificar la repetición de la textura o, dicho de otra manera, qué ocurre cuando las coordenadas de textura exceden el rango $[0, 1]$, se emplean los valores *GL_CLAMP* y *GL_REPEAT*. En el caso de utilizar *GL_CLAMP*, las coordenadas de textura se modifican al valor más cercano al del rango válido cuando se salen de este. En el caso de utilizar *GL_REPEAT*, sólo la parte decimal de las coordenadas de textura se emplearán, produciendo la repetición de la textura a lo largo de la superficie. Observa la figura 6.2 donde se muestran ejemplos de repetición.

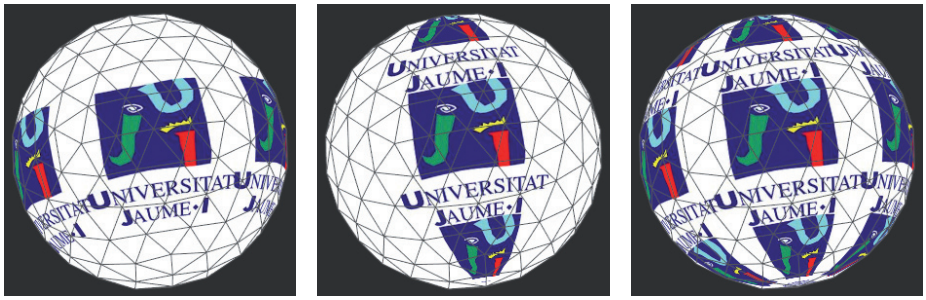


Figura 6.2: Ejemplo de repetición de textura. A la izquierda se ha repetido la textura en sentido s, en el centro en sentido t, y a la derecha en ambos sentidos

Respecto al filtrado de la textura, OpenGL trata tanto el problema de magnificación –cuando la porción de la textura a utilizar tiene un tamaño inferior a un píxel–, como el de la minimización –cuando la porción de la textura a utilizar abarca un conjunto de píxeles–. Así, se utilizará el valor *GL_LINEAR* si se requiere una interpolación lineal de 2x2 o *GL_NEAREST* si no se requiere filtrado. En el listado 6.2 se muestra el uso de estos parámetros.

Listado 6.2: Ejemplo de creación de una textura

```
GLubyte *textura= leeTextura("metal.rgb", 256, 256);

// Solicita un nombre
GLuint nombre;
glGenTextures (1, &nombre);

// Crea un objeto textura
glBindTexture (GL_TEXTURE_2D, nombre);

// Especifica la textura RGB de talla 256x256
glTexImage2D (GL_TEXTURE_2D, 0, GL_RGB, 256, 256, 0, GL_RGB,
              GL_UNSIGNED_BYTE, textura);

// Repite la textura tanto en s como en t
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

// Filtrado lineal de la textura
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);

// Activa la unidad de textura 0
glActiveTexture (GL_TEXTURE0);

// Asigna el objeto textura a dicha unidad de textura
glBindTexture (GL_TEXTURE_2D, nombre);

// Obtén el índice de la variable del shader de tipo sampler2D
GLuint loc= glGetUniformLocation (program, "Img");

// Indica que Img del shader use la unidad de textura 0
glUniform1i (loc,0);
```

6.3. Asignar la unidad de textura

Una vez creado el objeto textura, es necesario asignarlo a una unidad de textura. Las unidades de texturas son finitas y su número depende del hardware. El consorcio ARB fija que al menos 4 unidades de textura deben existir, pero es posible que nuestro procesador gráfico disponga de más.

La orden *glActiveTexture* especifica el selector de unidad de textura activa. Así, por ejemplo, la orden *glActiveTexture(GL_TEXTURE0)* selecciona la unidad de textura cero. A continuación hay que especificar el objeto textura a utilizar por dicha unidad con la orden *glBindTexture* (ver el listado 6.2). A cada unidad de textura sólo se le puede asignar un objeto textura pero, durante la ejecución, podemos cambiar tantas veces como queramos el objeto textura asignado a una unidad.

6.4. Especificar coordenadas

Las coordenadas de textura son un atributo más de los vértices, como lo son también el color o la normal. Es responsabilidad del programador suministrar estas coordenadas para cada vértice del modelo al igual que con el resto de los atributos. El rango de coordenadas válido en el espacio de la textura es entre 0 y 1, independientemente del tamaño en píxeles de la textura. Observa el ejemplo de la figura 6.3.

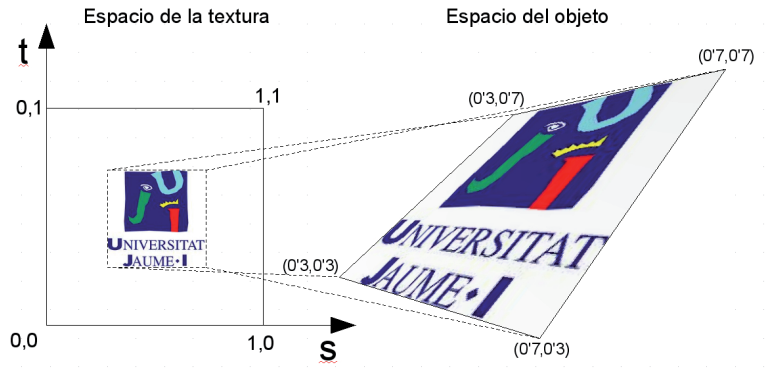


Figura 6.3: Asignación de coordenadas de textura a un objeto

6.5. Muestreo de la textura

Las coordenadas de textura se proporcionan para cada vértice y son interpoladas en el *pipeline* del procesador gráfico. En el procesador de fragmentos se utiliza la orden *texture* para acceder al texel correspondiente a dicho fragmento. Esta función devuelve el color, cuatro componentes, y ya ha sido procesado según los parámetros de repetición y filtrado especificados por el programador. Observa el *fragment shader* del listado 6.3.

Listado 6.3: Acceso a la textura desde el *fragment shader*

```
uniform sampler2D Img;

in  vec2 coordTextura;
out vec4 colorFragmento;

void main()
{
  colorFragmento = texture (Img, coordTextura);
}
```

El acceso a la unidad de textura se realiza a través de un *sampler*, en el caso de una imagen 2D el tipo correcto es *sampler2D*. Indicar a qué unidad de textura debe acceder el *sampler* se realiza desde la aplicación con la orden *glUniformi* tal y como se puede observar al final del listado 6.2. Por último señalar que desde el *Fragment shader* se puede acceder a múltiples unidades de textura.

PARTE II
TÉCNICAS BÁSICAS

Capítulo 7

Realismo visual

Este capítulo presenta tres tareas básicas en la búsqueda del realismo visual en imágenes sintéticas: trasparencia, reflejos y sombras. En la literatura se han presentado numerosos métodos para cada una de ellas. Aquí, se muestra una manera simple de realizarlas, consiguiendo una mejora importante en la calidad visual con poco esfuerzo de programación.

7.1. Trasparencia

Cuando una escena incluye un objeto trasparente, el color de los píxeles cubiertos por dicho objeto depende, además de las propiedades del objeto trasparente, de los objetos que hayan detrás de él. Un método sencillo para incluir objetos trasparentes en nuestra escena consiste en dibujar, en primer lugar, todos los objetos que sean opacos para después dibujar los objetos trasparentes. El grado de trasparencia se suministra al procesador gráfico como una cuarta componente en las propiedades de material del objeto, conocida como componente *alfa*. Si *alfa* es uno, el objeto es totalmente opaco, y cero significa que es totalmente trasparente (ver imagen 7.1). Así, el color final se calcula a partir del color del *framebuffer* y del color del fragmento de esta manera:

$$C_{final} = alfa \cdot C_{fragmento} + (1 - alfa) \cdot C_{framebuffer} \tag{7.1}$$

Al dibujar un objeto trasparente, el test de profundidad se ha de realizar de igual manera que al dibujar un objeto opaco y así asegurar que el problema de la visibilidad se resuelve correctamente. Sin embargo, ya que un objeto trasparente deja ver a través de él, el valor de profundidad de cada fragmento suyo que supere el test deberá actualizar el *buffer* de color pero no el de profundidad, ya que de hacerlo evitaría que otros objetos trasparentes situados detrás fuesen visibles. El listado 7.1 recoge la secuencia de órdenes de OpenGL necesaria para poder incluir objetos trasparentes en la escena. Consulta la especificación de OpenGL para conocer las órdenes involucradas.

Figura 7.1: Cilindro trasparente con valores, de izquierda a derecha, $alfa = 0,1$, $alfa = 0,4$ y $alfa = 0,7$

En el caso de que hayan varios objetos trasparentes y que estos se solapen en la proyección, el color final en la zona solapada es diferente dependiendo del orden en el que se hayan dibujado. Una forma de evitar este problema es establecer la operación de cálculo de la trasparencia como un incremento sobre el color acumulado en el *framebuffer*:

$$C_{final} = alfa \cdot C_{fragmento} + C_{framebuffer} \tag{7.2}$$

En OpenGL, esto se conigue especificando como función de cálculo *GL_ONE* en lugar de *GL_ONE_MINUS_SRC_ALPHA*. De esta manera, el orden en el que se dibujen los objetos trasparentes ya no influye en el color final de las zonas solapadas. Por contra, las zonas visibles a través de los objetos trasparentes son más brillantes que en el resto, produciendo una diferencia que en general resulta demasiado notable.

7.2. Reflejos

Los objetos reflejantes, al igual que los trasparentes, son también muy habituales en cualquier escenario. Desde espejos puros a objetos que por sus propiedades de material, y también de su proceso de fabricación, como el mármol por ejemplo, reflejan la luz y actúan casi como espejos. Sin embargo, el cálculo del reflejo es realmente complejo, por lo que se han desarrollado métodos alternativos consiguiendo muy buenos resultados visuales.

Esta sección muestra como añadir superficies planas reflejantes a nuestra escena (ver figura 7.2). El método consiste en dibujar la escena de forma simétrica respecto al plano que contiene al objeto reflejante (el objeto en el que se vaya a observar el reflejo). Hay dos tareas principales. La primera es obtener la transformación de simetría. La segunda es evitar que la escena simétrica se observe fuera de los límites del objeto reflejante.

Para dibujar la escena simétrica respecto a un plano, hay que trasladar el plano al origen, girarlo para hacerlo coincidir con, por ejemplo, el plano $Z = 0$, escalar con un factor de -1 en la dirección Z, deshacer el giro y la traslación. Dado un

Listado 7.1: Secuencia de operaciones para dibujar objetos trasparentes

```
// dibuja en primer lugar los objetos opacos
...

// activa el cálculo de la trasparencia
glEnable (GL_BLEND);

// especifica la funcion de cálculo
glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

// impide la actualización del buffer de profundidad
glDepthMask (GL_FALSE);

// dibuja los objetos trasparentes
...

// inhabilita la trasparencia y
// permite actualizar el buffer de profundidad
glDisable   (GL_BLEND);
glDepthMask (GL_TRUE);
```

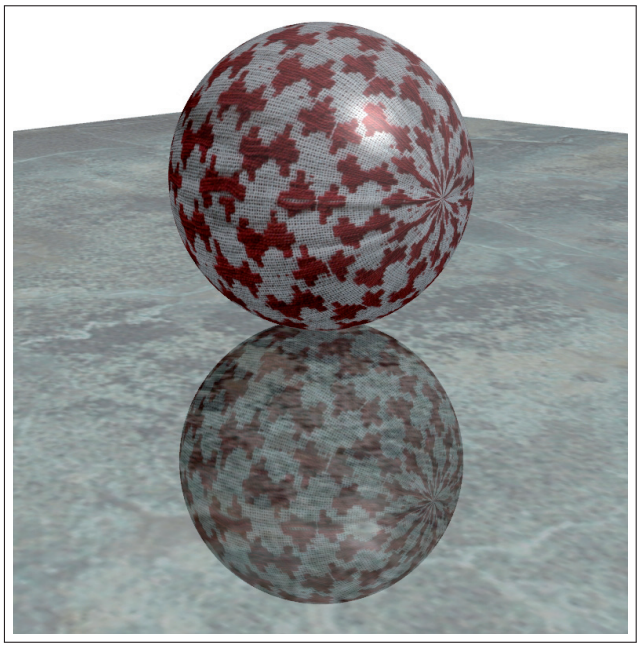


Figura 7.2: Ejemplo de objeto reflejado en una superficie plana

punto P del objeto plano reflejante y un vector V perpendicular al plano, la matriz de trasformación M es la siguiente:

$$M = \begin{pmatrix} 1-2V_x^2 & -2V_xV_y & -2V_xV_z & 2(P\cdot V)V_x \\ -2V_xV_y & 1-2V_y^2 & -2V_yV_z & 2(P\cdot V)V_y \\ -2V_xV_z & -2V_yV_z & 1-2V_z^2 & 2(P\cdot V)V_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{7.3}$$

Para la segunda tarea, no dibujar fuera de los límites del objeto reflejante (ver figura 7.3), hay varios métodos. Uno de ellos consiste en utilizar el *buffer* de plantilla de la siguiente forma. En primer lugar se dibuja el objeto reflejante habiendo previamente deshabilitado los *buffers* de color y de profundidad, y también habiendo configurado el *buffer* de plantilla para que se pongan a 1 los píxeles de dicho *buffer* que correspondan con la proyección del objeto. Después, se habilitan los *buffers* de profundidad y de color y se configura el *buffer* de plantilla para rechazar los píxeles que en el *buffer* de plantilla no estén a 1. Entonces se dibuja la escena simétrica. Después se deshabilita el *buffer* de plantilla y se dibuja la escena normal. Por último, opcionalmente, dibujar el objeto reflejante utilizando trasparencia. El listado 7.2 muestra cómo se realizan estos pasos con OpenGL. La creación del *buffer* de plantilla hay que solicitarla en el proceso de inicialización junto a los otros *buffers*:

```
glutInitDisplayMode (... |GLUT_STENCIL);
```

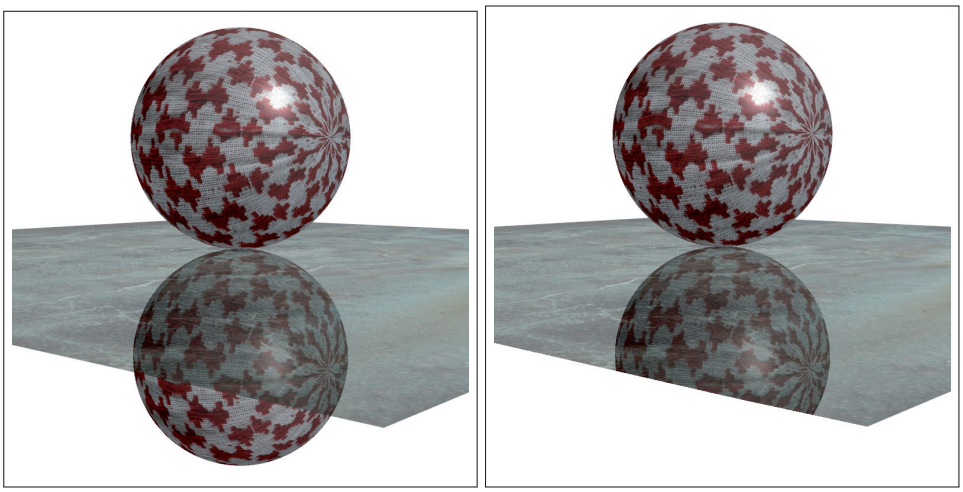


Figura 7.3: Al dibujar la escena simétrica es posible observarla fuera de los límites del objeto reflejante (izquierda). El *buffer* de plantilla se puede utilizar para resolver el problema (derecha)

Listado 7.2: Secuencia de operaciones para dibujar objetos reflejantes

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

// Desactiva los buffers de color y profundidad
glDisable   (GL_DEPTH_TEST);
glColorMask (GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);

// Establece como valor de referencia el 1
glEnable      (GL_STENCIL_TEST);
glStencilOp   (GL_REPLACE, GL_REPLACE, GL_REPLACE);
glStencilFunc (GL_ALWAYS,1,0xffffffff);

// Dibuja el objeto reflejante
...

// Activa de nuevo los buffers de profundidad y de color
glColorMask (GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
glEnable    (GL_DEPTH_TEST);

// Configura el buffer de plantilla
glStencilFunc (GL_EQUAL,1,0xffffffff);
glStencilOp   (GL_KEEP, GL_KEEP, GL_KEEP);

// Dibuja la escena reflejada
...

// Desactiva el test de plantilla
glDisable(GL_STENCIL_TEST);

// Dibuja la escena normal
...

// Dibuja el objeto reflejante con trasparencia
...
```

7.3. Sombras

En el mundo real, si hay fuentes de luz, habrán sombras. Sin embargo, en el mundo de la informática gŕafica podemos crear escenarios con fuentes de luz y sin sombras. Por desgracia, la ausencia de sombras en la escena es algo que, además de incidir negativamente en el realismo visual de la imagen sintética, nos dificulta de manera importante su comprensión, sobre todo en escenarios tridimensionales. Esto ha hecho que en la literatura encontremos mumerosos y muy diversos métodos que tratan de aportar soluciones. Por suerte, prácticamente cualquier método que nos permita añadir sombras, por sencillo que sea, puede ser más que suficiente para aumentar el realismo y que el usuario se sienta cómodo al observar el mundo 3D.

Un método muy simple para el cálculo de sombras sobre superficies planas es el conocido con el nombre de sombras proyectivas. Consiste en obtener la proyección del objeto situando la cámara en el punto de luz y estableciendo como plano de proyección aquel en el que queramos que aparezca su sombra. El resultado de la proyección, al que llamamos objeto sombra, se dibuja como un objeto más de la escena pero sin propiedades de material ni iluminación, simplemente de color oscuro. Dada una fuente de luz L y un plano de proyección $N \cdot x + d = 0$ la matriz de proyección M es la siguiente:

$$M = \begin{pmatrix} N \cdot L + d - L_x N_x & -L_x N_y & -L_x N_z & -L_x d \\ -L_y N_x & N \cdot L + d - L_y N_y & -L_y N_z & -L_y d \\ -L_z N_x & -L_z N_y & N \cdot L + d - L_z N_z & -L_z d \\ -N_x & -N_y & -N_z & N \cdot L \end{pmatrix} \tag{7.4}$$

Por contra, este método presenta una serie de problemas:

- Como el objeto sombra es coplanar con el plano que se ha utilizado para el cálculo de la proyección, habría que añadir un pequeño desplazamiento a uno de ellos para evitar el efecto conocido como *stitching*. OpenGL proporciona la orden *glPolygonOffset* para especificar el desplazamiento que se sumará a la profundidad de cada fagmento siempre y cuando se haya habilitado con *glEnable (GL_POLYGON_OFFSET_FILL)*.

- Hay que controlar que el objeto sombra no vaya más allá de la superficie sobre la que recae. Al igual que en la representación de reflejos, el *buffer* de plantilla se puede utilizar para asegurar el correcto dibujado de la escena.

- Las sombras son muy oscuras, pero utilizando trasparencia se puede conseguir un resultado mucho más agradable al dejar entrever el plano sobre el que se asientan (ver figura 7.4).

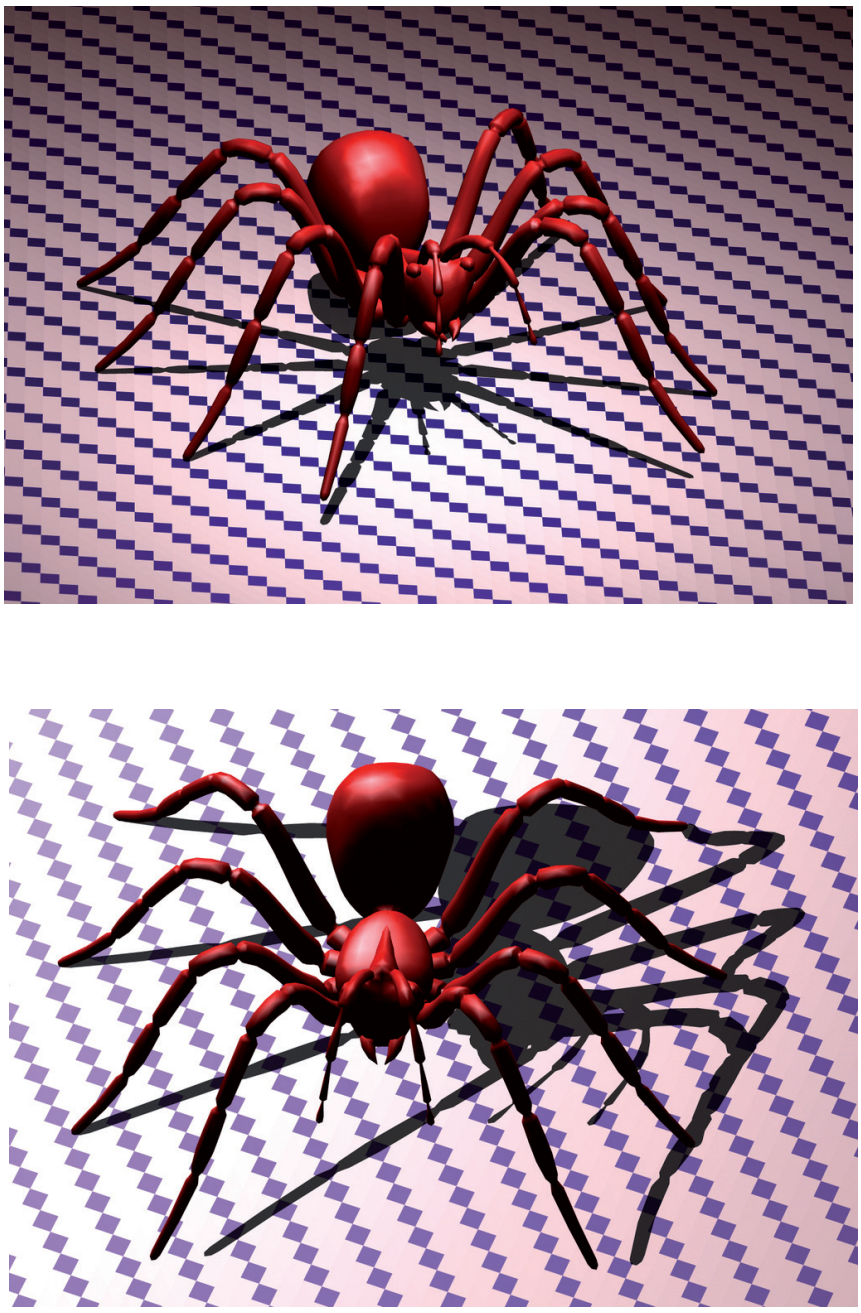


Figura 7.4: Sombras proyectivas trasparentes

Capítulo 8

Texturas avanzadas

En el capítulo 6 se muestra cómo aplicar una textura, que consiste en una imagen 2D, sobre un objeto 3D. Ahora, el objetivo es mostrar otras técnicas de aplicación de texturas donde probablemente la principal diferencia reside en un nuevo concepto de textura más general. Donde antes una textura se utilizaba únicamente para obtener un valor de color para cada fragmento, ahora una textura se va a utilizar para modificar el aspecto general de un objeto, incluyendo por ejemplo las normales y su geometría.

8.1. *Environment mapping*

Esta técnica de aplicación de textura se utiliza para simular objetos que reflejan su entorno. El objetivo es utilizar una textura que contenga la escena que rodea al objeto y, en tiempo de ejecución, determinar las coordenadas de textura que van a depender del vector dirección del observador o, mejor dicho, de su reflexión en cada punto de la superficie. De esta manera, las coordenadas de textura cambian al moverse el observador consiguiendo que parezca que el objeto refleja su entorno.

A la textura o conjunto de texturas que se utilizan para almacenar el entorno de un objeto se le denomina mapa de entorno. Este mapa puede estar formado por sólo una textura, al cual se accede utilizando coordenadas esféricas, o por un conjunto de seis texturas cuadradas formando un cubo. Este último es el que se describe en esta sección.

Un mapa de cubo son seis texturas cuadradas que se disponen para formar un cubo de lado dos centrado en el origen de coordenadas. Para cada punto de la superficie del objeto reflejante se obtiene el vector de reflexión respecto a la normal en ese punto de la superficie. Las coordenadas de este vector se van a utilizar para acceder al mapa de cubo y obtener el color. En primer lugar, hay que determinar cuál de las seis texturas se ha de utlizar. Para ello, se elige la coordenada de mayor magnitud. Si es la coordenada x, se utiliza la cara derecha o izquierda del cubo, dependiendo del signo. De igual forma, si es la y se utiliza la de arriba o la de abajo, o la de delante o de detrás si es la z. Después, hay que obtener las coordenadas u y

v para acceder a la textura seleccionada. Por ejemplo, si la coordenada x del vector de reflexión es la de mayor magnitud, las coordenadas de textura se pueden obtener así:

$$u = \frac{y+x}{2x} \qquad v = \frac{z+x}{2x} \tag{8.1}$$

Este cálculo lo realiza en OpenGL la función *texture* cuando accede a una textura de tipo *samplerCube*. Esta operación se realiza en el *fragment shader* igual que cuando se accede a una textura 2D. El vector de reflexión se calcula con la función *reflect* para cada vértice en el *vertex shader* y el *pipeline* de OpenGL hará que cada fragmento reciba el vector convenientemente interpolado. En el listado 8.1 se muestra el uso de estas funciones en ambos *shaders*. La figura 8.1 muestra un ejemplo de mapa y del resultado obtenido.

Figura 8.1: En la imagen de la izquierda se muestra el mapa de cubo con las seis texturas en forma de cubo desplegado. En la imagen de la derecha, el mapa de cubo se ha utilizado para simular que el objeto central está reflejando su entorno

Listado 8.1: *Shader* para *environment cube mapping*

```
// Vertex shader ------------------------------------------------------
uniform mat4  projection, camera, transf; // matrices
uniform mat3  normal;

in  vec3 posicion, vNormal;      // recibe vértice y normal
out vec3 reflexion;              // vector de reflexión

void main()
{
  vec4 ecPosition = camera * transf * vec4(posicion,1.0);
  vec3 N          = normalize(normal * vNormal);
  reflexion       = reflect(ecPosition.xyz, N);
  gl_Position     = projection * ecPosition;
}

// Fragment shader ----------------------------------------------------
uniform samplerCube mapa;

in  vec3 reflexion;
out vec4 colorFragmento;
```

```
void main()
{
  vec3 color = vec3(texture(mapa, reflexion));
  colorFragmento = vec4 (color, 1.0);
}
```

Por último, para crear el mapa de cubo hay que proporcionar las seis texturas al mismo objeto textura y establecer las opciones de repetición para las tres coordenadas de textura. El listado 8.2 muestra la secuencia de órdenes.

Listado 8.2: Creación de un mapa de cubo

```
GLubyte *textura1= leeTextura("iquierda.rgb", 512, 512);
GLubyte *textura2= leeTextura("derecha.rgb",  512, 512);
GLubyte *textura3= leeTextura("arriba.rgb",   512, 512);
GLubyte *textura4= leeTextura("abajo.rgb",    512, 512);
GLubyte *textura5= leeTextura("delante.rgb",  512, 512);
GLubyte *textura6= leeTextura("detras.rgb",   512, 512);

GLuint nombre;
glGenTextures (1, &nombre);

// Crea un objeto textura
glBindTexture (GL_TEXTURE_CUBE_MAP, nombre);

// Especifica las texturas
glTexImage2D (GL_TEXTURE_CUBE_MAP_POSITIVE_X, 0, GL_RGB, 512, 512,
              0, GL_RGB, GL_UNSIGNED_BYTE, textura1);
glTexImage2D (GL_TEXTURE_CUBE_MAP_NEGATIVE_X, ..., textura2);
glTexImage2D (GL_TEXTURE_CUBE_MAP_POSITIVE_Y, ..., textura3);
glTexImage2D (GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, ..., textura4);
glTexImage2D (GL_TEXTURE_CUBE_MAP_POSITIVE_Z, ..., textura5);
glTexImage2D (GL_TEXTURE_CUBE_MAP_NEGATIVE_Z, ..., textura6);

glTexParameteri (GL_TEXTURE_CUBE_MAP, ..._WRAP_S, GL_REPEAT);
glTexParameteri (GL_TEXTURE_CUBE_MAP, ..._WRAP_T, GL_REPEAT);
glTexParameteri (GL_TEXTURE_CUBE_MAP, ..._WRAP_R, GL_REPEAT);
glTexParameteri (GL_TEXTURE_CUBE_MAP, ..._MAG_FILTER, GL_LINEAR);
glTexParameteri (GL_TEXTURE_CUBE_MAP, ..._MIN_FILTER, GL_LINEAR);

// Activa la unidad de textura 0 y le asigna la textura
glActiveTexture (GL_TEXTURE0);
glBindTexture   (GL_TEXTURE_CUBE_MAP, nombre);
```

8.2. Enrejado

Enrejado es un tipo de textura que se califica como procedural. Una textura procedural es aquella que se obtiene mediante la ejecución de un procedimiento. Dicho de otra manera, en lugar de acceder a una imagen para obtener el color de un fragmento, ahora habrá un algoritmo que al ejecutarse nos proporcionará dicho valor.

Un ejemplo muy simple de textura procedural es el enrejado. Consiste en utilizar la orden *discard* para eliminar fragmentos, produciendo en consecuencia agujeros en la superficie del objeto. Las coordenadas de textura se utilizan para controlar el tamaño del agujero y la repetición. En concreto, es la parte fraccional de las coordenadas de textura las que se utilizan. Observa el código del *fragment shader* del listado 8.3. La figura 8.2 muestra algunos resultados.

Listado 8.3: *Shader* para textura de enrejado

```
in  vec3 color;
in  vec2 coordTextura;
out vec4 colorFragmento;

uniform vec2 nVeces;
uniform vec2 talla;

void main()
{
  float s = fract (coordTextura.s * nVeces.s);
  float t = fract (coordTextura.t * nVeces.t);

  if ((s > talla.s) && (t > talla.t)) discard;

  colorFragmento = vec4(color, 1.0);
}
```

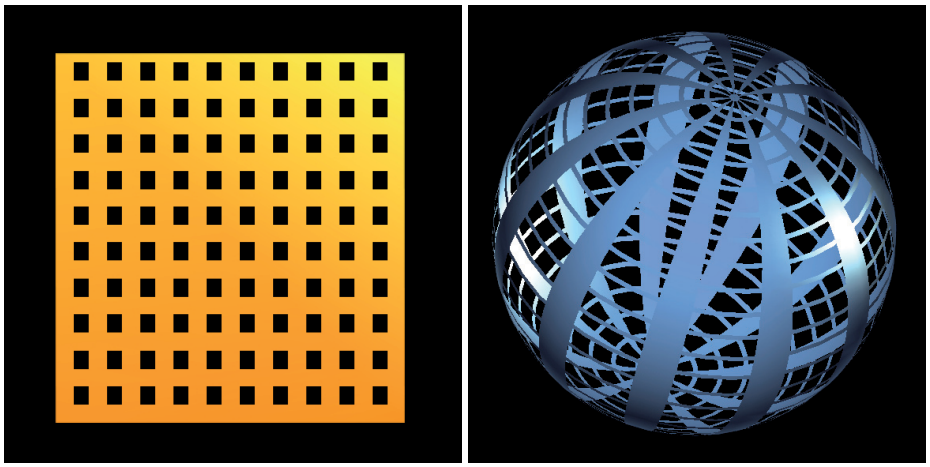


Figura 8.2: Ejemplos de enrejados

8.3. Texturas 3D

Una textura 3D define un valor para cada punto del espacio. Este tipo de texturas resulta perfecto para objetos que son creados a partir de un medio sólido como una talla de madera o un bloque de piedra. Hay diferentes métodos para generar

texturas 3D. En esta sección se muestra cómo utilizar una función de ruido para crear dicha textura (ver figura 8.3). En términos generales, el uso de ruido en informática gráfica resulta muy útil para simular efectos atmosféricos, materiales o simplemente imperfecciones. A diferencia de otras áreas, la función de ruido que nos interesa ha de producir siempre la misma salida para el mismo valor de entrada, y al mismo tiempo aparentar aleatoriedad, es decir, que no muestre patrones regulares.

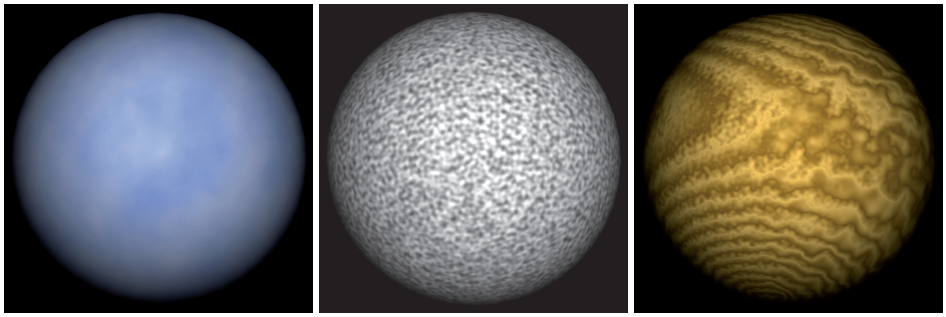


Figura 8.3: Ejemplos de resultados obtenidos utilizando una textura 3D creada con una función de ruido

El uso de ruido en OpenGL se puede realizar de tres maneras. Una es utilizar la familia de funciones *noise* de GLSL. Otra es implementar una función de ruido propia en el *shader*. Y la última es almacenar el resultado de la función de ruido y proporcionarlo en forma de textura al procesador gráfico para que en lugar de calcularlo lo lea directamente. Esta útima opción es sin duda la más eficiente. Además, algunos fabricantes de hardware gráfico no implementan las funciones *noise* obligándonos a implementar la nuestra propia. Por otra parte, el ruido se puede utilizar en el *vertex shader* para animar o simplemente modificar la geometría. En esta sección, el objetivo es utilizar el ruido como método para generar una textura, por lo que su uso recae exclusivamente en el *fragment shader*. La figura 8.4 muestra otro ejemplo de objeto que utiliza una textura 3D.

La textura 3D se especifica utilizando la función *glTexImage3D*. Además, al igual que para una textura 2D, hay que crear un objeto textura, asignarlo a una unidad y activarla. El listado 8.4 muestra estos pasos.

En el *shader* se accede a la textura a través de un *sampler3D* utilizando tres coordenadas de textura. Estas pueden ser las propias coordenadas geométricas del vértice, que han de ser enviadas por el *vertex shader* y recibidas para cada fragmento debidamente interpoladas. De esta forma se asegura que para cada vértice siempre se obtiene el mismo valor de textura. Normalmente, es habitual añadir dos factores de escala para controlar la amplitud de la textura:

```
vec4 valor = texture(madera, coordVertice * S1) * S2;
```

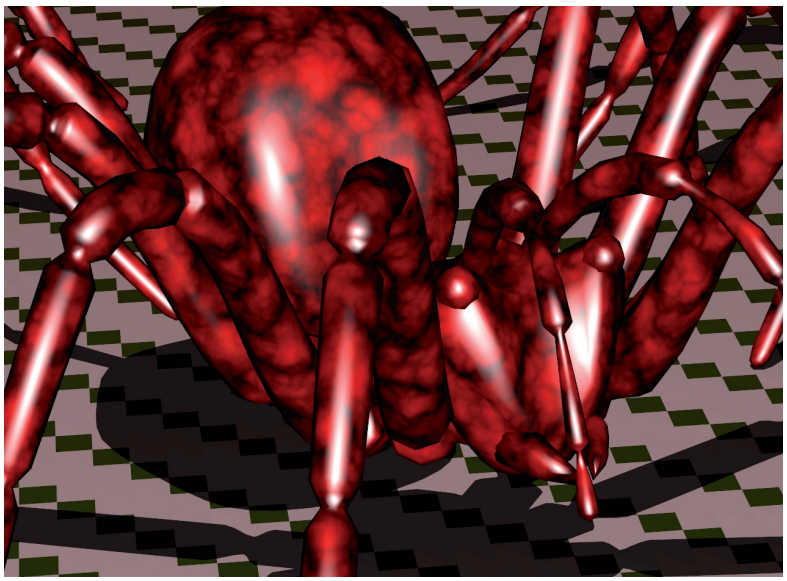


Figura 8.4: Objeto que hace uso de una textura 3D creada con una función de ruido

Listado 8.4: Creación de una textura 3D

```
GLubyte *textura= leeTextura3D("datos.rgb", 512, 512, 512);

GLuint nombre;
glGenTextures (1, &nombre);

// Crea un objeto textura
glBindTexture (GL_TEXTURE_3D, nombre);

// Especifica la textura y sus parámetros
glTexImage3D (GL_TEXTURE_3D, 0, GL_RGB, 512, 512, 512,
              0, GL_RGB, GL_UNSIGNED_BYTE, textura);
glTexParameteri (GL_TEXTURE_3D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri (GL_TEXTURE_3D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri (GL_TEXTURE_3D, GL_TEXTURE_WRAP_R, GL_REPEAT);
glTexParameteri (GL_TEXTURE_3D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri (GL_TEXTURE_3D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);

// Activa la unidad de textura 0 y le asigna la textura
glActiveTexture (GL_TEXTURE0);
glBindTexture   (GL_TEXTURE_3D, nombre);
```

8.4. *Bump mapping*

Esta técnica consiste en modificar la normal de la superficie para dar la ilusión de rugosidad o simplemente de modificación de la geometría a muy pequeña escala (ver imagen 8.5). El cálculo de la variación de la normal se puede realizar en el propio *shader* utilizando un algoritmo, o precalcularlo y proporcionárselo al procesador gráfico como una textura, conocida con el nombre de mapa de normales o *bump map*. También, una función de ruido se puede utilizar para generar la perturbación (ver imagen 8.6).

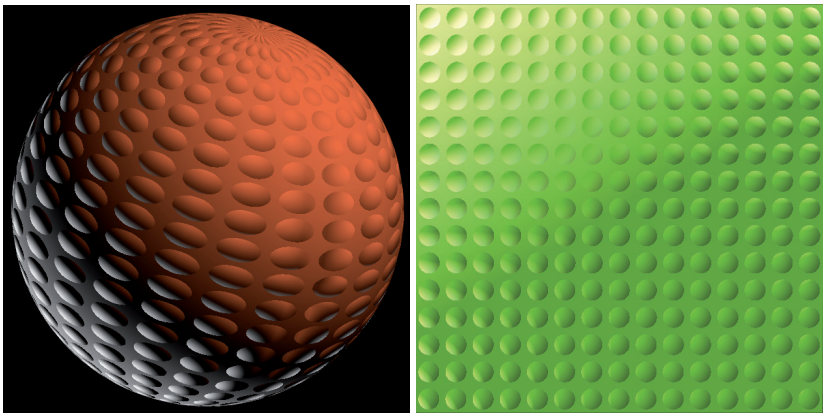


Figura 8.5: Objetos texturados con la técnica de *bump mapping*. La modificación de la normal produce que aparentemente la superficie tenga bultos

Figura 8.6: La normal del plano se perturba utilizando una función de ruido haciendo que parezca que tenga pequeñas ondulaciones

8.5. Desplazamiento

Esta técnica consiste en aplicar un desplazamiento en cada vértice de la superficie del objeto. El caso más sencillo es aplicar el desplazamiento en la dirección de la normal de la superficie en dicho punto. El desplazamiento se puede almacenar en una textura a la que se le conoce como mapa de desplazamiento. El *vertex shader* también puede acceder al mapa de desplazamiento y así modificar la posición del vértice (ver figura 8.7).

Figura 8.7: Ejemplo de desplazamiento de la geometría produciendo una ondulación de la superficie

Capítulo 9

Proceso de imágenes

Desde sus orígenes, OpenGL ha tenido en cuenta en el diseño de su *pipeline* la posibilidad de manipular imágenes sin asociarle geometría alguna. Sin embargo, no es hasta que se produce la aparición de los procesadores gráficos programables cuando de verdad OpenGL se puede utilizar como una herramienta para procesado de imágenes, consiguiendo aumentar de manera drástica la capacidad de analizar y modificar imágenes, así como de generar una amplia variedad de efectos (ver imagen 9.1).

Figura 9.1: Ejemplo de procesado de imagen. A la imagen de la izquierda se le ha aplicado un efecto de remolino generando la imagen de la derecha

9.1. Efectos de imagen

Conseguir diversos efectos de imagen a través del procesador gráfico es una tarea bastante sencilla utilizando OpenGL. El procesador de fragmentos recibe un valor de color de cuatro componentes (RGBA) que podemos modificar como más nos interese. En el caso de que nuestro efecto involucre utilizar sólo una imagen, esta puede ser enviada al procesador gráfico utilizando la orden *glDrawPixels*. Sin embargo, si para calcular el color definitivo de un fragmento necesitamos conocer los valores de color de otros píxeles de la imagen, entonces será necesario proporcionar la imagen como textura. También, en el caso de utilizar más de una imagen, el resto de imágenes tendrán que ser proporcionadas como texturas.

9.1.1. Brillo

La modificación del brillo de una imagen es un efecto muy sencillo. Sólo hay que multiplicar el color de cada fragmento por un valor $alpha$. Si dicho valor es 1, la imagen no se altera, si es mayor que uno se aumentará el brillo, y si es menor que uno se disminuirá. El listado 9.1 muestra el código del *fragment shader* correspondiente. La figura 9.2(b) muestra un ejemplo donde $alpha = 1,6$.

Listado 9.1: *Fragment shader* para modificar el brillo de una imagen

```
uniform float alfa;

in  vec4 color;
out vec4 colorFragmento;

void main()
{
  colorFragmento = color * alfa;
}
```

9.1.2. Negativo

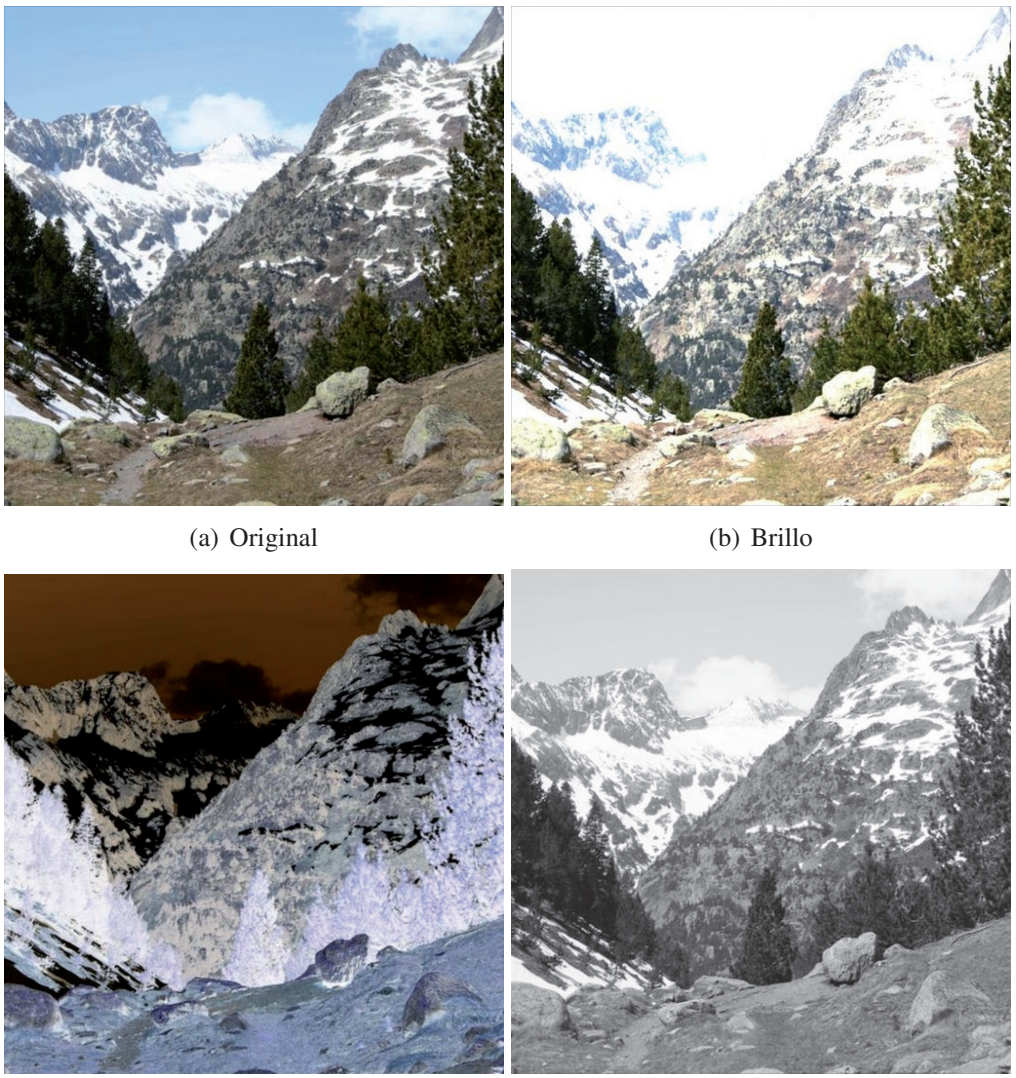
Otro ejemplo muy sencillo es la obtención del negativo de una imagen (ver imagen 9.2(c)). Únicamente hay que asignar como color final del fragmento el resultado de restarle a uno su valor de color original. En el listado 9.1 habría que sustituir el cálculo del color del fragmento por la siguiente línea:

```
  colorFragmento = 1.0 - color;
```

9.1.3. Escala de grises

También muy fácil es la obtención de la imagen en escala de grises (ver imagen 9.2(d)). Únicamente hay que asignar como color final del fragmento el resultado de la media de sus tres componentes. En el listado 9.1 habría que sustituir el cálculo del color del fragmento por las siguientes líneas:

```
  float media    = (color[0] + color[1] + color[2]) / 3.0;
  colorFragmento = vec4 (media, media, media, 1.0);
```



(a) Original (b) Brillo

(c) Negativo (d) Escala de grises

Figura 9.2: Diversos efectos realizados sobre la misma imagen

9.1.4. Mezcla de imágenes

Mezclar dos imágenes supone acceder en el *fragment shader* a un píxel de cada una de las imágenes para calcular el color final. Al menos una de ella se ha de cargar como textura, mientras que la otra puede ser proporcionada a través de la orden *glDrawPixels* (o por supuesto también almacenando ambas como texturas).

En su forma más sencilla, sólo hay que utilizar la función *mix* que realiza una mezcla lineal de dos valores x e y usando un valor real α así: $x * (1 - \alpha) + y * \alpha$.

El listado 9.2 muestra un ejemplo del correspondiente *fragment shader* en el que ambas imágenes son accedidas como texturas. En la figura 9.3 se muestra el resultado de mezclar la imagen de la figura 9.2(a) y de la figura 9.1 utilizando dos valores distintos de *alfa*.

Listado 9.2: *Fragment shader* para combinar dos imágenes

```
uniform float alfa;
uniform sampler2D imagenA, imagenB;

in  vec2 coordenadaTextura;
out vec4 colorFragmento;

void main()
{
  vec4 colorA = texture(imagenA, coordenadaTextura.st);
  vec4 colorB = texture(imagenB, coordenadaTextura.st);

  colorFragmento = mix (colorA, colorB, alfa) ;
}
```

(a) alfa = 0,33 (b) alfa = 0,66

Figura 9.3: Mezcla de dos imágenes

Ejercicios

► **9.1** Escribe el *fragment shader* que realice la diferencia de dos imágenes.

9.2. Convolución

La convolución es una operación matemática fundamental en procesamiento de imágenes. Consiste en calcular para cada píxel la suma de productos entre la imagen fuente y una matriz mucho más pequeña a la que se le denomina filtro de convolución. Lo que la operación de convolución realice depende de los valores de dicho filtro. Para un filtro de dimensión $m \times n$ la operación es:

$$Res(x,y) = \sum_{j=0}^{n-1} \sum_{i=0}^{m-1} Img(x+(i-\frac{m-1}{2}), y+(j-\frac{n-1}{2})) \cdot Filtro(i,j) \quad (9.1)$$

Realizar esta operación con OpenGL requiere que la imagen sea cargada como textura, ya que es la única manera de que desde el *fragment shader* se pueda acceder a cuaquier píxel de la imagen. Por otra parte, si la operación de la convolución sobrepasa los límites de la imagen, los mismos parámetros que se utilizaron para especificar el comportamiento de la aplicación de texturas fuera del rango $[0,1]$ se utilizarán ahora también (ver sección 6.2.2).

El listado 9.3 muestra el *fragment shader* para el cálculo de la convolución de un filtro de tamaño 3×3. Hay que tener en cuenta que el *shader* se ejecutará para cada píxel, por lo que es conveniente que por eficiencia sea la aplicación quien calcule una sola vez los desplazamientos y los remita al *shader*. Para el cálculo de los desplazamientos hay que recordar que el rango válido de una textura es $[0,1]$, por lo que el desplazamiento para acceder al siguiente píxel, por ejemplo, en horizontal será $\frac{1}{anchoImagen-1}$, De igual forma, cada valor del filtro es dividido previamente para así evitar la división dentro del *shader*, que en el listado implicaría dividir por 9 el valor de color asignado al fragmento en la última línea del código.

Las operaciones más habituales son el *blurring*, el *sharpening* y la detección de aristas entre otras. La tabla 9.1 muestra ejemplos de filtros de suavizado o *blurring*, nitidez o *sharpening* y detección de bordes.

1	1	1
1	1	1
1	1	1

(a)

0	-1	0
-1	5	-1
0	-1	0

(b)

0	1	0
1	-4	1
0	1	0

(c)

Tabla 9.1: Filtros de convolución: (a) suavizado, (b) nitidez y (c) detección de bordes

Listado 9.3: *Fragment shader* para el cálculo de la convolución

```
const   int  tallaFiltro 9;

uniform vec2  desplazamiento[tallaFiltro];
uniform float filtro[tallaFiltro];

uniform sampler2D imagen;

in  vec2 coordenadaTextura;
out vec4 colorFragmento;

void main()
{
  int i;
  vec4 suma = vec4(0.0);

  for (i = 0; i < tallaFiltro; i++)
  {
    vec4 aux = textura(imagen,
                       coordenadaTextura.st + desplazamiento[i]);
    suma = suma + (aux * filtro[i]);
  }

  colorFragmento = suma;
}
```

Ejercicios

► **9.2** Modifica el código del listado 9.3 para operar con filtros de tamaño variable.

9.3. *Antialiasing*

Se conoce como esfecto escalera o dientes de sierra, o más comúnmente por su término en inglés *aliasing*, al artefacto gráfico derivado de la conversión de entidades continuas a discretas. Por ejemplo, al visualizar un segmento de línea se convierte a una secuencia de píxeles coloreados en el *framebuffer*, siendo claramente perceptible el problema, excepto si la línea es horizontal o vertical (ver imagen 9.4). Este problema es todavía más fácil de percibir, y también mucho más molesto, si los objetos están en movimiento.

Figura 9.4: En la imagen de la izquierda se observa claramente el efecto escalera que se hace más suave en la imagen de la derecha

Nos referimos con *antialiasing* a las técnicas destinadas a eliminar ese efecto escalera. Hoy en día, la potencia de los procesadores gráficos permite que desde el propio panel de control del controlador gráfico el usuario pueda solicitar la solución de este problema e incluso establecer el grado de calidad. Hay que tener en cuenta que a mayor calidad del resultado, mayor coste para la GPU, pudiendo llegar a producir cierta ralentización en la interacción con nuestro entorno gráfico. Por este motivo, las aplicaciones gráficas exigentes con el hardware gráfico suelen ofrecer al usuario la posibilidad de activarlo o no.

OpenGL implementa la técnica conocida como sobremuestreo, en inglés *multisampling*, también conocido por *Full Scene Anti-Aliasing*, FSAA. Consiste en obtener más de un valor para cada píxel de la imagen final de manera que el color definitivo sea el resultado de una operación de mezcla realizada sobre el conjunto de todas las muestras obtenidas. Tanto la operación como la localización de las muestras dependen del fabricante del hardware. Ahora, cada fragmento se extiende para incluir, para cada muestra, información adicional de color, profundidad, etc. Toda esta información se almacena en un nuevo *buffer* llamado *multisample buffer*. Cada etapa del *pipeline* se realiza sobre cada muestra. Sin embargo, para reducir el coste computacional, se suele utilizar el mismo valor de color y de coordenadas de textura para todas las muestras de un mismo píxel.

Ejercicios

► 9.3 Reflexiona sobre la necesidad de mantener un *buffer* de color, de profundidad y de plantilla, cuando se está usando la técnica de sobremuestreo. Es decir, si para cada muestra se almacena la información de color, de profunidad y de plantilla en el *buffer multisample*, ¿crees necesario mantener los otros *buffers* generales?

Su creación se solicita a través de la FreeGLUT así:

```
glutInitDisplayMode (... | GLUT_MULTISAMPLE);
```

Después, el sobremuestreo se habilita con:

```
glEnable (GL_MULTISAMPLE);
```

Para realmente saber si se está haciendo o no el sobremuestreo, hay que comprobar que el resultado de la variable *buffers* tras la ejecución de la siguiente línea es uno:

```
glGetIntegerv (GL_SAMPLES_BUFFERS, &buffers);
```

Y para conocer cuántas muestras se está calculando por cada píxel hay que consultar el valor de la variable *muestras* después de ejecutar la siguiente orden:

```
glGetIntegerv (GL_SAMPLES, &muestras);
```

9.4. Transformaciones

La transformación geométrica de una imagen se realiza en tres pasos:

1. Leer la imagen y crear un objeto textura con ella.
2. Definir un rectángulo sobre el que pegar la textura.
3. Escribir un *vertex shader* que realice la operación geométrica deseada.

El paso 1 ha sido descrito en el capítulo 6. Para realizar el paso 2 no es necesario que el rectángulo se defina de acuerdo a las proporciones de la imagen, simplemente hay que modelar un rectángulo y utilizar siempre el mismo. En el paso 3, el *vertex shader* debe transformar los vértices del rectángulo de acuerdo a la transformación geométrica requerida. Por ejemplo, para ampliar la imagen sólo hay que multiplicar los vértices por un factor de escala superior a 1, y para reducirla el factor de escala debe estar entre 0 y 1. Es en este paso donde también se debe dar la proporción adecuada al rectángulo para evitar la deformación de la imagen.

Ejercicios

► **9.4** Entre las operaciones típicas para el procesado de imágenes se encuentran las operaciones de volteo horizontal y vertical. ¿Cómo implementarías dichas operaciones?

Otra transformación a realizar en el *vertex shader* es el *warping*, es decir, la modificación de la imagen de manera que la distorsión sea perceptible. Esta técnica se realiza definiendo una malla de polígonos en lugar de un único rectángulo, y modificando los vértices de manera conveniente (ver imagen 9.5).

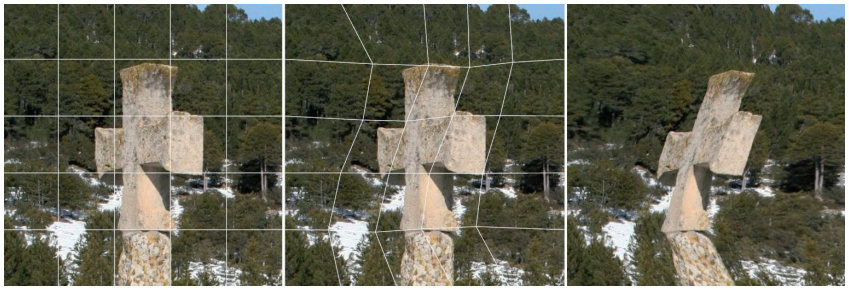


Figura 9.5: *Warping* de una imagen: imagen original en la izquierda, malla modificada en la imagen del centro y resultado en la imagen de la derecha

Como una extensión de la técnica anterior se podría realizar el *morphing* de dos imágenes. Dadas dos imágenes de entrada, hay que obtener la secuencia de imágenes que transforma una de las imágenes de entrada en la otra. Para esto se define una malla de polígonos sobre cada una de las imágenes, y el *morphing* se consigue mediante la transformación de los vértices de una malla a las posiciones de los vértices de la otra malla, al mismo tiempo que el color definitivo de cada píxel se obtiene con una función de mezcla.

9.5. Recuperación y almacenamiento

OpenGL permite llevar el contenido del *framebuffer* a la memoria principal pero no proporciona ninguna orden para almacenarlo en un fichero. Para transferir el contenido de la ventana de la aplicación a un fichero hay que hacerlo en dos pasos: llevarlo en primer lugar a memoria principal, haciendo previamente la correspondiente reserva de memoria, y después a un fichero, por ejemplo, como se muestra en el listado 9.4. La función que permite transferir el contenido del *framebuffer* a memoria principal es *glReadPixels*. Además es necesario utilizar la orden *glPixelStorei* para especificar la alineación de los datos en memoria para el comienzo de cada fila de píxeles. Un valor de 1 significa que la alineación es de tipo *byte*.

Listado 9.4: Almacenamiento del contenido de la ventana a un fichero

```
int      talla    = anchoVentana*altoVentana*3;
GLubyte *pixeles = (GLubyte *) calloc (sizeof(GLubyte), talla);

if (pixeles != NULL)
{
  // primer paso
  glPixelStorei (GL_PACK_ALIGNMENT, 1);
  glReadPixels  (0, 0, anchoVentana, altoVentana,
                 GL_RGB, GL_UNSIGNES_BYTE, pixeles);

  // segundo paso
  FILE *fichero = fopen (filename, "wb");
  fwrite (pixeles, sizeof(GLubyte), talla, fichero);
  fclose (fichero);

  free   (pixeles);
}
```

PARTE III
TÉCNICAS AVANZADAS

Capítulo 10

Interacción

10.1. Eventos

La captura y el manejo de eventos es esencial para crear aplicaciones interactivas. Sin embargo, OpenGL no proporciona ninguna herramienta que nos permita realizarlo, por lo que es necesario recurrir a otras librerías que nos proporcionen esta funcionalidad. Esta es precisamente una de las finalidades de la librería FreeGLUT. En esta sección se muestra el conjunto de órdenes que la librería FreeGLUT proporciona para el manejo de eventos.

La librería FreeGLUT permite asociar rutinas de código, conocidas como *callbacks*, ante eventos generados por el usuario al interactuar con el teclado y el ratón. La orden:

- void glutKeyboardFunc(void (*$funcion$)(unsigned char $tecla$,int x, int y));

permite al programador especificar la función de *callback* ante un evento de teclado. Dicha función recibirá el código ASCII de la tecla pulsada así como las coordenadas de ventana donde el puntero del ratón se encontraba cuando se produjo el evento. También la siguiente función:

- void glutSpecialFunc(void (*$funcion$)(int $tecla$, int x, int y));

permite especificar la función de *callback* para cuando el usuario presione una tecla especial, es decir, teclas de función, cursores, página arriba y abajo, teclas *Home* y *End*, y la tecla *Insert*. Para conocer qué tecla especial se ha pulsado hay constantes definidas del tipo: GLUT_KEY_F1, GLUT_KEY_PAGE_UP, GLUT_KEY_HOME, GLUT_KEY_LEFT, etc.

Respecto al ratón, la orden:

- void glutMouseFunc(void (*$funcion$)(int $boton$, int $estado$, int x, int y));

permite especificar la función de *callback* para cuando el usuario presione un botón del ratón. Dicha función recibirá qué botón se ha pulsado de entre tres posibles:

GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON o GLUT_RIGHT_BUTTON; y su estado, es decir, si se ha pulsado o soltado: GLUT_UP o GLUT_DOWN. Y también las coordenadas del cursor en coordenadas de ventana. Por otra parte, la función:

- void glutMotionFunc(void (*$funcion$)(int x, int y));

especifica la función de *callback* que será ejecutada cuando se desplace el ratón mientras el usuario mantenga algún botón pulsado. La función recibirá como parámetros las coordenadas del cursor en coordenadas de ventana.

Si el usuario mueve o modifica el tamaño de la ventana, con la función:

- void glutReshapeFunc(void (*$funcion$)(int $ancho$, int $alto$));

se especifica la función de *callback* que será ejecutada cuando ocurra cualquiera de las dos cosas. La función recibe como parámetros el nuevo ancho y el nuevo alto en píxeles de la ventana.

Por útltimo, indicar que con la orden *glutPostRedisplay()* el programador puede generar un evento para solicitar el redibujado de la ventana. Es decir, que en cuanto sea posible se ejecute la función de *callback* especificada con la orden *glutDisplayFunc*.

10.2. Menús

La librería FreeGLUT también proporciona la posibilidad de crear menús desplegables. Estos menús se crean y se asocian a un evento de ratón producido por la pulsación de uno de los botones. El menú siempre aparece allí donde el cursor se encuentre.

Para crear un menú se ha de llamar a la función:

- int glutCreateMenu(void (*$funcion$)(int $valor$));

la cual devuelve un identificador del menú creado y como parámetro se especifica la función de callback que será ejecutada cuando cualquiera de las opciones del menú sea seleccionada. La función de callback recibe como parámetro un identificador de la opción que el usuario ha seleccionado. Tras llamar a la función de creación de un menú, este se establece como menú actual. Para añadir una opción al menú se especifica con la función:

- void glutAddMenuEntry(char *$opcion$, int $valor$);

la cual además requiere un valor que será el que se envíe a la función de callback cuando la opción correspondiente sea seleccionada. Las opciones aparecerán en el menú en el mismo orden en el que se especifican en el código, y siempre se añaden al menú establecido como actual.

También es posible crear un submenú, es decir, una opción que despliega otro menú con más opciones. Un submenú se crea igual que un menú, usando las órdenes anteriores, y se añade a otro menú con la orden:

- void glutAddSubMenu(char *$opcion$, int $idMenu$);

la cual requiere como parámetros el nombre de la opción y el identificador del submenú. El nuevo submenú siempre se añade al menú establecido como actual. Si fuera necesario añadirlo a otro diferente, la orden *glutSetMenu(int idMenu)* nos permite marcar como actual el menú especficado como parámetro de la función.

Por último, sólo queda asociar el menú a un botón del ratón. Esta función es la encargada:

- void glutAttachMenu(int $boton$);

que como parámetro recibe el botón y le asocia el menú establecido como actual. El listado 10.1 recoge un ejemplo de menú desplegable y en la figura 10.1 se muestra el resultado.

Listado 10.1: Creación de un menú desplegable con FreeGLUT

```
// Crea un submenú
idSubmenu = glutCreateMenu(menu);

// Las opciones del submenú
glutAddMenuEntry("Alzado", 2);
glutAddMenuEntry("Planta", 3);
glutAddMenuEntry("Perfil", 4);

// Crea el menú principal
idMenu = glutCreateMenu(menu);

// Crea las entradas
glutAddMenuEntry("Perspectiva", 1);
glutAddSubMenu   ("Paralela", idSubmenu);
glutAddMenuEntry("Salir", 0);

// Asocia el menú al botón derecho
glutAttachMenu(GLUT_RIGHT_BUTTON);
```

Figura 10.1: Ejemplo de menú desplegable

10.3. Selección

Es fácil que una aplicación requiera que el usuario pueda señalar objetos de la escena y que, por tanto, la aplicación pueda saber qué objeto se ha señalado. Una forma de dar soporte a la selección de objetos con OpenGL es utilizar un objeto *framebuffer*. OpenGL permite crear, modificar y destruir objetos *framebuffer*, diferentes del creado inicialmente por la aplicación. Un objeto *framebuffer* encapsula la información necesaria para describir una colección de *buffers* de color, profundidad y plantilla. La capacidad de poder dibujar en un objeto *framebuffer* creado por la propia aplicación tiene distintos usos como la selección de objetos, el dibujado fuera de pantalla o el dibujado a textura. Esta sección describe cómo utilizarlo aplicado en concreto a la selección de objetos.

El usuario seleccionará un objeto utilizando el ratón y haciendo clic sobre un objeto de la escena. A continuación, la aplicación debe averiguar sobre qué objeto se ha hecho el clic. Entonces se dibuja la misma escena pero en el objeto *framebuffer* y con una salvedad, los objetos seleccionables se pintarán con colores planos y diferentes entre sí. Después, utlizando las coordenadas de ventana proporcionadas por la FreeGLUT tras el clic realizado por el usuario, se leerá el color de dicho píxel y así sabremos de qué objeto se trata. En consecuencia, es necesario que por una parte se cree un objeto *framebuffer* sobre el que dibujar la escena. Este *framebuffer* debe constar de *buffer* de color y de profundidad, y tendrá la misma dimensión que el *framebuffer* de la ventana de la aplicación. Por otra parte, hay que dibujar la escena dos veces, una sobre el *framebuffer* de la ventana y otra sobre el *framebuffer* propio utilizando colores planos, para entonces leer el color del píxel dado.

Para crear un objeto *framebuffer* hay que obtener un nombre utilizando la orden *glGenFramebuffers* y enlazarlo con la orden *glBindFramebuffer*, de forma muy similar a la creación de un objeto textura. Depués hay que asignarle, utilizando la orden *glFramebufferRenderbuffer*, los dos *buffers* que necesitamos, el de color y el de profundidad, y que previamente se deben haber creado. Para crear estos dos *buffers*, el de color y el de profundidad, y asociárselos al objeto *framebuffer*, también hay que solicitar dos nombres con la orden *glGenRenderbuffers* y enlazarles a cada uno un *buffer* mediante *glBindRenderbuffer* y *glRenderbufferStorage*. El listado 10.2 muestra un ejemplo con todos los pasos.

Cuando sea necesario hay que dibujar la escena en el *framebuffer* creado por la aplicación. En primer lugar, hay que activar el *framebuffer* para poder dibujar en él, dibujar la escena tal cual se ha mostrado al usuario (incluyendo el borrado de los *buffers* de color y profundidad) pero utilizando colores planos, y recuperar el color del píxel dado utilizando *glReadPixels*. El listado 10.3 muestra un ejemplo con todos los pasos.

Listado 10.2: Creación de un *framebuffer* con *buffer* de color y de profundidad

```
enum {Color, Depth, NumRenderbuffers};
GLuint framebuffer, renderbuffer[NumRenderbuffers];

// recibe como parámetros el ancho y el alto del framebuffer
void initFrameBuffer (int ancho, int alto)
{
  // se crean dos buffers
  glGenRenderbuffers(NumRenderbuffers, renderbuffer);

  // uno almacenará el color
  glBindRenderbuffer    (GL_RENDERBUFFER, renderbuffer[Color]);
  glRenderbufferStorage(GL_RENDERBUFFER, GL_RGBA, ancho, alto);

  // este otro almacenará la profundidad
  glBindRenderbuffer    (GL_RENDERBUFFER, renderbuffer[Depth]);
  glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT24,
                        ancho, alto);

  // ahora creamos el framebuffer
  glGenFramebuffers(1, &framebuffer);
  glBindFramebuffer(GL_DRAW_FRAMEBUFFER, framebuffer);

  // y le adjudicamos los dos buffers creados previamente
  glFramebufferRenderbuffer(GL_DRAW_FRAMEBUFFER,
                            GL_COLOR_ATTACHMENT0,
                            GL_RENDERBUFFER, renderbuffer[Color]);
  glFramebufferRenderbuffer(GL_DRAW_FRAMEBUFFER,
                            GL_DEPTH_ATTACHMENT,
                            GL_RENDERBUFFER, renderbuffer[Depth]);
  glEnable(GL_DEPTH_TEST);

  // de momento lo desactivamos
  glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0);
}
```

Listado 10.3: Dibujo en el *framebuffer*

```
// recibe como parámetros las coordenadas del píxel
void renderToFramebuffer (int x, int y)
{
  // activa el framebuffer para dibujar
  glBindFramebuffer(GL_DRAW_FRAMEBUFFER, framebuffer);

  // dibuja la escena utilizando colores planos
  ...

  // recupera el color del píxel seleccionado
  // dependiendo del color leído sabrás qué objeto
  // se ha seleccionado
  glBindFramebuffer(GL_READ_FRAMEBUFFER, framebuffer);
  GLfloat pixel[4];
  glReadPixels(x, alto-y, 1, 1, GL_RGBA, GL_FLOAT, pixel);

  // Desactiva de nuevo el framebuffer
  glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0);
}
```

Capítulo 11

Modelado de curvas y superficies

11.1. Curvas cúbicas paramétricas

Las curvas cúbicas son las que gozan de mayor popularidad. Hay varias razones por las que utilizar polinomios de orden cúbico para la representación de curvas. Las curvas de menor orden ofrecen poca flexibilidad, y las de mayor grado tienen un mayor coste computacional y sus propiedades son raramente requeridas. Por supuesto, hay que tener en cuenta que una curva compleja siempre se puede representar mediante una secuencia de segmentos de curvas cúbicas. Además, estas son las curvas de menor orden que permiten reresentar curvas 3D (ver figura 11.1).

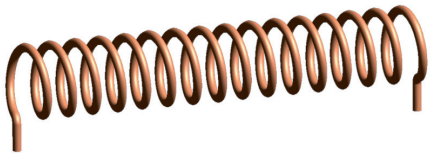


Figura 11.1: Este solenoide es un ejemplo de curva 3D

11.1.1. Representación

Se puede definir una curva cúbica paramétrica de la siguiente manera:

$$Q(u) = \begin{bmatrix} x(u) & y(u) & z(u) \end{bmatrix} \qquad u \in [0,1] \tag{11.1}$$

donde:

$$\begin{aligned} x(u) &= a_x u^3 + b_x u^2 + c_x u + d_x \\ y(u) &= a_y u^3 + b_y u^2 + c_y u + d_y \qquad u \in [0,1] \\ z(u) &= a_z u^3 + b_z u^2 + c_z u + d_z \end{aligned} \tag{11.2}$$

Si se define U así:

$$U = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \tag{11.3}$$

Y definimos la matriz de coeficientes C como:

$$C = \begin{bmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \\ d_x & d_y & d_z \end{bmatrix}$$

Entonces, la ecuación 11.1 se puede reescribir así:

$$Q(u) = UC$$

Sin embargo, la matriz de coeficientes C se puede formular como el producto de dos matrices MG, donde M es una matriz base característica del tipo de curva y G es un vector de geometría que establece las restricciones, que son puntos o tangentes, de una curva determinada:

$$Q(u) = UMG \tag{11.4}$$

11.1.2. Continuidad

En la sección anterior se dice que una curva compleja se puede representar como una secuencia de segmentos de curvas cúbicas. Esto obliga a prestar especial atención en los puntos en los que dos segmentos de curvas cúbicas se juntan. Se denomina continuidad de una curva al hecho de que todas las piezas que la forman vayan unidas unas a otras de tal manera que la curva pueda ser dibujada como una pieza continua. Existen varios tipos de continuidad:

- Paramétrica
  - C^0. Las posiciones de la curva coinciden.
  - C^1. Las posiciones y las primeras derivadas (tangentes) coinciden. En ciertas áreas se le denomina velocidad.
  - C^2. Las posiciones y primeras y segundas derivadas coinciden. En ciertas áreas se le denomina aceleración.
  - C^n. Coinciden posiciones y las n derivadas.
- Geométrica
  - G^0. Las posiciones de la curva coinciden.
  - G^1. Las posiciones coinciden y las tangentes lo hacen en dirección pero no en magnitud.

Ejercicios

► **11.5** Contesta a las siguientes cuestiones.

- Para que una cámara se desplace a lo largo de una trayectoria sin que el observador note efectos extraños, ¿qué tipo de continuidad debemos exigir en los puntos de unión?
- ¿Implica continuidad C^1 continuidad G^1?, ¿y al contrario?

11.2. Curvas de Hermite

La curva de Hermite se define mediante dos puntos finales, $P1$ y $P4$, y las tangentes de la curva en dichos puntos, $R1$ y $R4$. Por lo que su vector de geometría G_H es el siguiente:

$$G_H = \begin{bmatrix} P1 \\ P4 \\ R1 \\ R4 \end{bmatrix} \tag{11.5}$$

Si M_H es la matriz base de una curva de Hermite y, para la coordenada x, el vector de geometría es G_{Hx}, la ecuación 11.4 queda de la siguiente manera:

$$x(u) = UM_HG_{Hx} = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} M_HG_{Hx} \tag{11.6}$$

Entonces, para el punto inicial, $u = 0$, y el final, $u = 1$, se obtiene que:

$$\begin{aligned} x(0) = P1_x &= \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix} M_HG_{Hx} \\ x(1) = P4_x &= \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix} M_HG_{Hx} \end{aligned} \tag{11.7}$$

Y la tangente, que es la primera derivada de la curva, sería:

$$x'(u) = \begin{bmatrix} 3u^2 & 2u & 1 & 0 \end{bmatrix} M_HG_{Hx}$$

Entonces, las tangentes en los puntos inicial y final son:

$$\begin{aligned} x'(0) = R1_x &= \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix} M_HG_{Hx} \\ x'(1) = R4_x &= \begin{bmatrix} 3 & 2 & 1 & 0 \end{bmatrix} M_HG_{Hx} \end{aligned} \tag{11.8}$$

Por lo que sustituyendo los resultados obtenidos en 11.7 y 11.8 en el vector de geometría definido en 11.5, se obtiene:

$$G_{Hx} = \begin{bmatrix} P1_x \\ P4_x \\ R1_x \\ R4_x \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix} M_H G_{Hx} \tag{11.9}$$

Y para que esta igualdad se cumpla, la matriz base M_H ha de ser:

$$M_H = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \tag{11.10}$$

En resumen:

$$\begin{aligned} Q(u) &= \begin{bmatrix} x(u) & y(u) & z(u) \end{bmatrix} = U M_H G_H \\ &= \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} P1 \\ P4 \\ R1 \\ R4 \end{bmatrix} \end{aligned} \tag{11.11}$$

Ejercicios

► **11.6** Contesta a las siguientes cuestiones.

- Si una curva utiliza dos segmentos cúbicos de Hermite, ¿cuál será el vector de geometría del segundo segmento si se desea continuidad G^1?
- ¿Cómo harías una transformación geométrica de una curva de Hermite en el espacio 3D: aplicando primero la transformación a sus restricciones y entonces obtener los puntos de la curva para su dibujado, u obteniendo primero los puntos y transformándolos todos después?

11.3. Curvas de Bezier

La curva de Bezier se define mediante dos puntos finales, $P1$ y $P4$, y dos puntos de control, $P2$ y $P3$. Un punto de control se utiliza para definir la forma de la curva pero no pasa por él. En parte coincide con Hermite, en sus dos puntos finales, mientras que las tangentes vienen dadas por los vectores $P1P2$ y $P3P4$, que se relacionan con $R1$ y $R4$ así:

$$\begin{aligned} R1 &= Q'(0) = 3(P2 - P1) \\ R4 &= Q'(1) = 3(P4 - P3) \end{aligned} \tag{11.12}$$

El vector de geometría es el siguiente:

$$G_B = \begin{bmatrix} P1 \\ P2 \\ P3 \\ P4 \end{bmatrix} \tag{11.13}$$

Para calcular la matriz base M_B podemos partir de la relación entre el vector de geometría de Hermite y el de Bezier. Si llamamos M_{HB} a la matriz que nos relaciona uno con otro, y considerando la ecuación 11.12, podemos escribir que:

$$G_H = \begin{bmatrix} P1 \\ P4 \\ R1 \\ R4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -3 & 3 & 0 & 0 \\ 0 & 0 & -3 & 3 \end{bmatrix} \begin{bmatrix} P1 \\ P2 \\ P3 \\ P4 \end{bmatrix} = M_{HB} G_B \tag{11.14}$$

Recuerda que:

$$Q(u) = U M_H G_H \tag{11.15}$$

Si se sustituye 11.14 en la ecuación anterior se obtiene:

$$Q(u) = U M_H M_{HB} G_B = U M_B G_B \tag{11.16}$$

Luego la matriz base de Bezier M_B será $M_H M_{HB}$:

$$M_B = M_H M_{HB} = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \tag{11.17}$$

Si se desarrolla el producto de la ecuación 11.16 se obtiene que:

$$Q(u) = (1-u)^3 P1 + 3u(1-u)^2 P2 + 3u^2(1-u) P3 + u^3 P4 \tag{11.18}$$

O lo que es lo mismo:

$$Q(u) = \sum_{i=0}^{3} b_{i,3} P_i \tag{11.19}$$

donde los $b_{i,3}$ son conocidos como los polinomios de Bernstein de grado 3:

$$\begin{aligned} b_{0,3} &= (1-u)^3 \\ b_{1,3} &= 3u(1-u)^2 \\ b_{2,3} &= 3u^2(1-u) \\ b_{3,3} &= u^3 \end{aligned}$$

Y su forma general es:

$$b_{k,n}(u) = C(n,k)u^k(1-u)^{(n-k)} \tag{11.20}$$

donde n es el orden de la curva de Bezier, k es el número del polinomio entre 0 y n, y $C(n,k)$ son los coeficientes binomiales:

$$C(n,k) = \frac{n!}{k!(n-k)!} \tag{11.21}$$

Finalmente, dados k puntos, la función que calcula la curva de Bezier de orden n es:

$$Q(u) = \sum_{k=0}^{n} P_k C(n,k)u^k(1-u)^{(n-k)} \tag{11.22}$$

11.4. Curvas B-*Spline* uniformes no racionales

El origen de las *splines* son curvas antiguas realizadas con metal que garantizaban continuidad paramétrica C^2 y control global. Las B-*Splines* que se presentan en esta sección se caracterizan por mantener el mismo tipo de continuidad pero permiten control local. Estas curvas no pasan por los puntos de control, y estos a su vez se comparten entre los distintos segmentos.

Una B-*Spline* aproxima $m+1$ puntos de control $P_0..P_m$, con $m \geq 3$. Está formada por $m-2$ polinomios cúbicos y por los segmentos de curva $Q_3, Q_4, ..., Q_m$. Para cada segmento de curva Q_i, el parámetro u varía así: $u_i \leq u \leq u_{i+1}$ donde $3 \leq i \leq m$. Por ejemplo, para $m = 3$ hay cuatro puntos de control P0, P1, P2 y P3, y un único polinomio Q_3 definido en el intervalo $u_3 \leq u \leq u_4$.

Para cada $i \geq 4$ hay un punto de unión o nudo entre Q_i y Q_{i+1} y estará en u_i, al que se le conoce como valor del nudo. También u_3, inicial, y u_{m+1}, final, son valores de nudos por lo que hay un total de $m-1$ nudos.

Para hacer que una B-*Spline* sea cerrada simplemente hay que repetir los puntos de control P0, P1 y P2 al final, es decir, que la secuencia sería $P0$, $P1$, ..., P_m, $P0$, $P1$, $P2$.

Como la B-*Spline* es uniforme, los intervalos de u han de ser iguales, por lo que, $u_3 = 0$ y el intervalo $u_{i+1} - u_i = 1$.

Respecto al vector de geometría G_{BS} de una B-*Spline*, para un segmento Q_i definido por los puntos de control $P_{i-3}, P_{i-2}, P_{i-1}, P_i$ es el siguiente:

$$G_{BSi} = \begin{bmatrix} P_{i-3} \\ P_{i-2} \\ P_{i-1} \\ P_i \end{bmatrix}, 3 \leq i \leq m \tag{11.23}$$

Si definimos el vector U de la siguiente manera:

$$U = \begin{bmatrix}(u-u_i)^3 & (u-u_i)^2 & (u-u_i) & 1\end{bmatrix} \tag{11.24}$$

Entonces, el segmento Q_i con $u_i \leq u \leq u_{i+1}$ y $3 \leq i \leq m$ se puede escribir así:

$$Q_i(u) = UM_{BS}G_{BSi} \tag{11.25}$$

$$M_{BS} = \frac{1}{6}\begin{bmatrix}-1 & 3 & -3 & 1\\ 3 & -6 & 3 & 0\\ -3 & 0 & 3 & 0\\ 1 & 4 & 1 & 0\end{bmatrix} \tag{11.26}$$

Y desarrollándolo se obtiene:

$$\begin{aligned} Q_i(u-u_i) &= UM_{BS}G_{BSi} = \\ &= \frac{(1-u)^3}{6}P_{i-3} + \frac{3u^3-6u^2+4}{6}P_{i-2} + \\ &+ \frac{-3u^3+3u^2+3u+1}{6}P_{i-1} + \frac{u^3}{6}P_i, \qquad 0 \leq u < 1 \end{aligned} \tag{11.27}$$

Ejercicios

► **11.7** Observando el vector de geometría de una curva B-*Spline*, ¿hasta en cuántos segmentos se puede utilizar un punto de control?, ¿qué implica mover un punto de control en una B-*Spline*?

11.5. Superficies bicúbicas paramétricas

En la ecuación 11.4 se definía una curva cúbica paramétrica $Q(u) = UMG$ donde $U = [u^3 \;\; u^2 \;\; u \;\; 1]$, M es la matriz base 4×4 y G es el vector de geometría. Para una curva dada, el vector G es constante para todos los puntos $Q(u)$. Sin embargo, si permitimos que los puntos de dicho vector varíen a lo largo de una ruta 3D que dependa de un segundo parámetro v, obtenemos:

$$Q(u,v) = UMG(v) = UM\begin{bmatrix}G_1(v) & G_2(v) & G_3(v) & G_4(v)\end{bmatrix}^T \tag{11.28}$$

Para un valor fijo de $v = v_1$ el vector de geometría correspondiente $G(v_1)$ es constante por lo que $Q(u, v_1)$ será una curva. Si hacemos variar el parámetro v se obtiene el conjunto de todas las curvas definiendo en consecuencia una superficie. Si dichas curvas $G_i(v)$ son cúbicas, entonces a la superficie que definen se le denomina paramétrica bicúbica.

Cada curva cúbica $G_i(v)$ es:

$$G_i(v) = VMG_i = VM \begin{bmatrix} g_{i1} & g_{i2} & g_{i3} & g_{i4} \end{bmatrix}^T \tag{11.29}$$

sabiendo que $(ABC)^T = C^T B^T A^T$ se obtiene que:

$$G_i(v) = G_i^T M^T V^T = \begin{bmatrix} g_{i1} & g_{i2} & g_{i3} & g_{i4} \end{bmatrix} M^T V^T \tag{11.30}$$

y sustituyendo en 11.28 se obtiene que:

$$Q(u,v) = UMG(v) = UMGM^T V^T \qquad 0 \leq u, v \leq 1 \tag{11.31}$$

donde G es la matriz de coeficientes:

$$G = \begin{bmatrix} g_{11} & g_{12} & g_{13} & g_{14} \\ g_{21} & g_{22} & g_{23} & g_{24} \\ g_{31} & g_{32} & g_{33} & g_{34} \\ g_{41} & g_{42} & g_{43} & g_{44} \end{bmatrix} \tag{11.32}$$

11.6. Superficies de Hermite

A partir de la ecuación 11.31, se define una superficie de Hermite como:

$$Q(u,v) = UM_H G_H M_H^T V^T \qquad 0 \leq u, v \leq 1 \tag{11.33}$$

donde la matriz M_H es la misma matriz base que se obtuvo para representar una curva de Hermite. Para obtener la matriz de geometría G_H se procede de forma similiar a como se realizó para la curva de Hermite, es decir:

$$Q(u,v) = UM_H G_H(v) = UM_H \begin{bmatrix} P_1(v) & P_4(v) & R_1(v) & R_4(v) \end{bmatrix}^T \tag{11.34}$$

donde las funciones $P_1(v)$ y $P_4(v)$ definen respectivamente el punto inicial y final de la curva en el parámetro u, y las funciones $R_1(v)$ y $R_4(v)$ definen respectivamente las tangentes en dichos puntos.

Sabiendo que:

$$\begin{aligned} P_1(v) &= VM_H \begin{bmatrix} g_{11} & g_{12} & g_{13} & g_{14} \end{bmatrix}^T \\ P_4(v) &= VM_H \begin{bmatrix} g_{21} & g_{22} & g_{23} & g_{24} \end{bmatrix}^T \\ R_1(v) &= VM_H \begin{bmatrix} g_{31} & g_{32} & g_{33} & g_{34} \end{bmatrix}^T \\ R_4(v) &= VM_H \begin{bmatrix} g_{41} & g_{42} & g_{43} & g_{44} \end{bmatrix}^T \end{aligned} \tag{11.35}$$

podemos escribir que:

$$\begin{bmatrix} P_1(v) & P_4(v) & R_1(v) & R_4(v) \end{bmatrix} = VM_H G_H^T \tag{11.36}$$

donde G_H es la matriz de coeficientes:

$$G_H = \begin{bmatrix} g_{11} & g_{12} & g_{13} & g_{14} \\ g_{21} & g_{22} & g_{23} & g_{24} \\ g_{31} & g_{32} & g_{33} & g_{34} \\ g_{41} & g_{42} & g_{43} & g_{44} \end{bmatrix} \tag{11.37}$$

Calculando la traspuesta de la ecuación 11.36, sabiendo que $(VM_HG_H^T)^T = G_HM_H^TV^T$ se obtiene que:

$$\begin{bmatrix} P_1(v) \\ P_4(v) \\ R_1(v) \\ R_4(v) \end{bmatrix} = \begin{bmatrix} g_{11} & g_{12} & g_{13} & g_{14} \\ g_{21} & g_{22} & g_{23} & g_{24} \\ g_{31} & g_{32} & g_{33} & g_{34} \\ g_{41} & g_{42} & g_{43} & g_{44} \end{bmatrix} M_H^TV^T \tag{11.38}$$

y sustituyendo en la ecuación 11.34 obtenemos finalmente:

$$Q(u,v) = UM_H \begin{bmatrix} P_1(v) \\ P_4(v) \\ R_1(v) \\ R_4(v) \end{bmatrix} = UM_HG_HM_H^TV^T \tag{11.39}$$

Si relacionamos las ecuaciones 11.34 y 11.35 podemos comprender los coeficientes de la matriz de geometría G_H. La matrix 2×2 superior izquierda son los cuatro puntos de las esquinas de la superficie, las matrices 2×2 superior derecha e inferior izquierda son las tangentes en las esquinas en las direcciones paramétricas v y u respectivamente y, por último, la matriz 2×2 inferior derecha son los vectores de torsión también en las esquinas de la superficie.

Para garantizar continuidad paramétrica C^0 en la unión de dos superficies bicúbicas de Hermite, los puntos de control en la curva de unión han de coincidir y, si además se desea continuidad C^1, también han de coincidir las tangentes y los vectores de torsión a lo largo de dicha curva.

Ejercicios

► **11.8** Si una superficie necesita dos trozos bicúbicos de Hermite de manera que la unión se produce para $u = 1$ en un trozo y para $u = 0$ del siguiente trozo, ¿qué valores de las respectivas matrices de geometría coinciden si se desea continuidad G^1?

11.7. Superficies de Bezier

De forma equivalente a como se deriva la ecuación de la superficie de Hermite se obtiene la de la curva de Bezier que es:

$$Q(u,v) = UM_B G_B M_B^T V^T \tag{11.40}$$

La matriz de geometría de Bezier G_B está formada por los 16 puntos de control que se pueden observar en la figura 11.2.

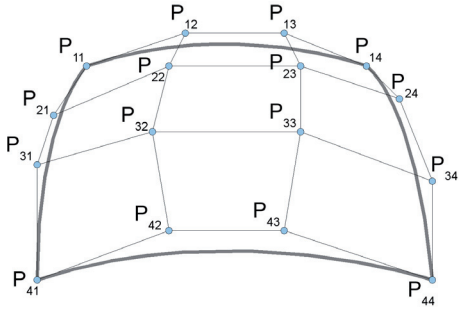


Figura 11.2: Los 16 puntos de control que definen la matriz de geometría de una superficie de Bezier

De forma general, para una matriz de $(n+1) \times (m+1)$ puntos de control la ecuación es la siguiente:

$$Q(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} P_{ij} b_{i,n}(u) b_{j,m}(v) \qquad 0 \leq u, v \leq 1 \tag{11.41}$$

Para garantizar continuidad paramétrica C^0 entre dos trozos de superficie de Bezier los cuatro puntos de control de la curva común a ambos deben coincidir. Si se desea continuidad G^1 los dos conjuntos de cuatro puntos de control a ambos lados del borde de unión han de ser colineales con los puntos del propio borde.

11.8. Superficies B-*Spline*

Un trozo de superficie bicúbica B-*Spline* se representa de la siguiente forma:

$$Q(u,v) = UM_{BS} G_{BS} M_{BS}^T V^T \tag{11.42}$$

La continuidad paramétrica C^2 está garantizada en los bordes siempre y cuando no se dupliquen puntos de control.

Capítulo 12

Animación

Los diferentes métodos y técnicas recogidos en este libro aparecen casi en cualquiera de los libros que trate los fundamentos de la informática gráfica. Con la animación por ordenador, lo que encontramos son libros que abordan de forma única y específica esta aplicación. Esto, sin lugar a duda, da una idea de la mangitud del campo de la animación por ordenador. Los orígenes de la animación, sus principios básicos, la introducción del ordenador en el proceso de producción, las diferentes técnicas de animación como la interpolación, la cinemática y la animación basada en física, la captura de movimiento, la simulación de fluidos, y un largo etcétera de métodos, técnicas y algoritmos los podemos encontrar en estas obras.

En este capítulo únicamente vamos a tratar la animación como el proceso de modificar los atributos de los elementos que componen la escena. Se va a mostrar lo fácil que resulta realizar animación utilizando *shaders*, cómo generar eventos de tiempo que nos ayuden a temporizar la animación y, por último, un tipo de modelado íntimamente ligado a la animación como son los sistemas de partículas.

12.1. *Shaders*

Realizar animación a través de *shaders* es sencillo. Sólo necesitamos una variable uniforme que vaya cambiando a lo largo del tiempo y que la aplicación sea quien la actualice. En el *shader*, se utiliza esta variable para modificar cualquiera de los atributos de los objetos.

Por ejemplo, si se modifica el valor *alfa* que controla la opacidad de un objeto, podemos conseguir que este aparezca o se desvanezca de forma gradual. Modificando las coordenadas de textura al utilizar, por ejemplo, una función seno podemos conseguir que la textura se ondule. O también aplicar una traslación a las coordenadas de una textura 3D permite obtener efectos muy interesantes en las superficies de los objetos. Si la textura almacena el resultado de una función de ruido que se utiliza para desplazar la geometría, modificar las coordenadas con el tiempo hará que toda la superficie se ondule. Un ejemplo de este último caso se puede observar en el listado 12.1.

Listado 12.1: Desplazamiento de un vértice utilizando una función de ruido y un valor de tiempo

```
uniform sampler2D ruido;
uniform float     tiempo, S1, S2;

in vec4 posicion;
in vec2 coordTextura;

void main ()
{
  ...
  vec3 valor = vec3(texture(ruido,(coordTextura*S1)+tiempo))*S2;
  vec3 nuevaPosicion = posicion.xyz + valor;
  ...
}
```

Otros ejemplo sería modificar parámetros de las fuentes de luz haciendo que, por ejemplo, la iluminación aumente o decaiga de forma gradual, o simplemente cambien sus colores mediante transiciones suaves. También por supuesto podemos modificar la matriz de transformación del modelo cambiando la posición, el tamaño o la orientación de algún objeto de la escena.

12.2. Eventos de tiempo

La librería FreeGLUT proporciona una orden para especificar que una determinada función sea ejecutada transcurrido un cierto tiempo. La disponibilidad de una función de este tipo es fundamental para actualizar la variable del *shader* que se utiliza para controlar la animación. La función es la siguiente:

- void glutTimerFunc (unsigned int $msegs$, void (*$funcion$) (int val), val);

El primer valor indica el número de milisegundos que han de transcurrir para que se llame a la función de *callback* especificada como segundo parámetro. Una vez transcurrido dicho tiempo, esa función de *callback* se ejecutará lo antes posible y recibirá como parámetro de entrada el valor indicado como tercer parámetro de la función *glutTimerFunc*. Si se desea que la función de *callback* se ejecute otra vez al cabo de un nuevo periodo de tiempo, ella misma puede establecerlo llamando a la función *glutTimerFunc* antes de finalizar. Por último, señalar que la librería FreeGLUT permite especificar varias funciones de *callback* de manera simultánea.

12.3. Sistemas de partículas

Los sistemas de partículas son una técnica de modelado para objetos que no tienen una frontera bien definida como, por ejemplo, humo, fuego o un spray. Estos objetos son dinámicos y se representan mediante una nube de partículas que,

en lugar de definir una superficie, definen un volumen. Cada partícula nace, vive y muere de manera independiente. En su periodo de vida, una partícula cambia de posición y de aspecto. Atributos como posición, color, trasparencia, velocidad, tamaño, forma o tiempo de vida se utilizan para definir una partícula. Durante la ejecución de un sistema de partículas, cada una de ellas se debe actualizar a partir de sus atributos y de un valor de tiempo global.

Las figuras 12.1 y 12.2 muestran ejemplos de un sistema en el que cada partícula es un cuadrado y pretenden modelar respectivamente un mosaico y pequeñas banderas en un escenario deportivo. En ambos casos, a cada partícula se le asigna un instante de nacimiento aleatorio de manera que las piezas del mosaico, o las banderas, aparecen en instantes de tiempo diferente. Mientras están vivas, para cada partícula se accede a una textura de ruido utilizando la variable que representa el paso del tiempo, y así no acceder siempre al mismo valor de la textura, con el fin de actualizar su posición. A cada partícula también se le asigna de forma aleatoria un valor de tiempo final que representa el instante en que la partícula debe desaparecer.

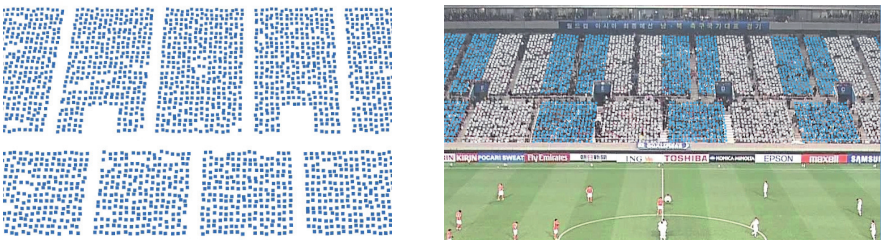


Figura 12.1: Animación de un mosaico implementado como sistema de partículas

Figura 12.2: Animación de banderas implementada como sistema de partículas

Todas las partículas junto con sus atributos pueden ser almacenadas en un *buffer object*. De esta manera, el *shader* de vértices es quien ha de determinar en cada ejecución el estado de la partícula, es decir, si aún no ha nacido, si está viva o si ya ha muerto. En el caso de estar viva, es en dicho *shader* donde se implementa su comportamiento. Por lo tanto, la visualización de un sistema de partículas se realiza totalmente en el procesador gráfico sin carga alguna para la CPU.