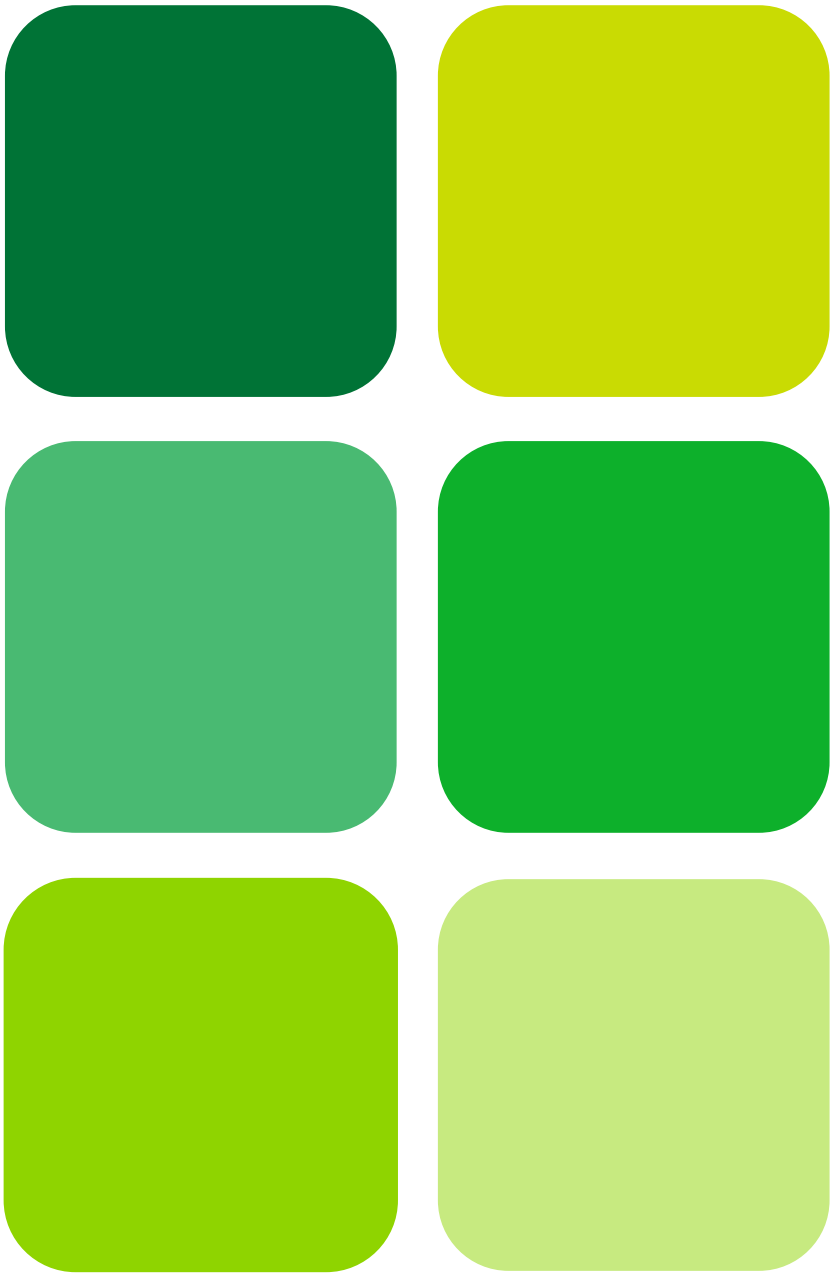


www.sapientia.uji.es | 107

UNIVERSITAT JAUME·I

Informática Gráfica

José Ribelles
Ángeles López

Informática Gráfica

José Ribelles
Ángeles López

Departament de Llenguatges i Sistemes Informàtics

Codi d'assignatura VJ1221

Informática Gráfica - UJI - DOI: http://dx.doi.org/10.6035/Sapientia107

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 107
www.sapientia.uji.es
Primera edició, 2015

ISBN: 978-84-16356-29-4

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

Aquest llibre, de contingut científic, ha estat avaluat per persones expertes externes a la Universitat Jaume I, mitjançant el mètode denominat revisió per iguals, doble cec.

Índice general

Índice de figuras

Índice de listados

Prefacio

La materia Informática Gráfica forma parte de los grados en Ingeniería Informática, Diseño y Desarrollo de Videojuegos y también del máster universitario en Sistemas Inteligentes, todos ellos de la Universitat Jaume I. El objetivo de este libro es proporcionar 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 adecuados a las asignaturas en las que se imparte. Este libro pretende ser el complemento ideal a las explicaciones que el profesor imparta en sus clases, no su sustituto, y del cual el alumno deberá mejorar el contenido con sus anotaciones.

El libro introduce al alumno en la programación moderna de gráficos por computador a través de la interfaz de programación de hardware gráfico WebGL 1.0. 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. Así, se introduce la programación de *shaders* con WebGL, el modelado poligonal, las transformaciones geométricas, la transformación de la cámara, las proyecciones, el modelo de iluminación de Phong y la aplicación de texturas 2D.

Este libro también introduce técnicas para el procesado de imágenes, como la convolución o el *antialising*, técnicas para aumentar el realismo visual, como trasparencias, reflejos y sombras, y métodos de aplicación de texuras más avanzados como el *environment mapping* o texturas procedurales. Respecto a la diversidad de métodos que existen, se ha optado por incluir 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.

Para terminar, este trabajo no se olvida de introducir técnicas relacionadas con el desarrollo de aplicaciones gráficas, tratando, por ejemplo, técnicas de interacción y de animación por computador con *shaders*.

Recursos en línea

Se ha creado la página web `http://cphoto.uji.es/grafica` como apoyo a este material, donde el lector puede descargar los programas de ejemplo que se incluyen en los diferentes capítulos.

Obras relacionadas

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

Capítulo 1

Introducción a WebGL

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

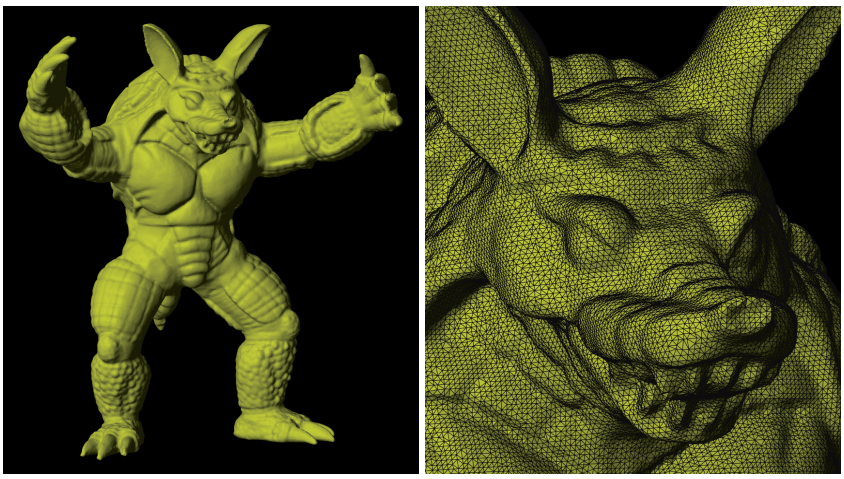


Figura 1.1: Ejemplo de objeto tridimensional dibujado con WebGL

1.1. Antecedentes

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 un 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).

Figura 1.2: Ejemplos de objetos dibujados mediante *shaders*

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, aunque 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.

OpenGL ES 1.1 aparece en el 2007. Se trata de la versión de OpenGL para sistemas empotrados incluyendo teléfonos móviles, tabletas, consolas, vehículos, etc. En solo un año evoluciona y se presenta la versión 2.0 creada a partir de la especificación de OpenGL 2.0. Desde el 2014 se encuentra disponible la versión 3.1, pero todavía está poco extendida, siendo la versión 2.0 la más soportada en la actualidad, al menos en lo que a dispositivos móviles se refiere.

WebGL 1.0 aparece en el año 2011. Se crea a partir de la especificación de OpenGL ES 2.0. En la actualidad está soportada por los navegadores Safari, Chrome, Firefox, Opera e Internet Explorer. Respecto al futuro cercano de WebGL, la versión 2.0 se basa en la especificación de OpenGL ES 3.0 y ya se encuentra en fase de pruebas.

1.2. Prueba de WebGL

Averiguar si se dispone de soporte para WebGL es muy simple. Abre un navegador y accede a cualquiera de las muchas páginas que informan de si el navegador soporta o no WebGL. Por ejemplo, la página `http://get.webgl.org` es una de ellas. Si funciona, se mostrará una página con un cubo en alambre dando vueltas sobre sí mismo como el que aparece en la figura 1.3.

Figura 1.3: Resultado con éxito del test de soporte de WebGL en un navegador proporcionado por la página `http://get.webgl.org`

Ejercicios

► **1.1** Comprueba la disponibilidad de WebGL en los diferentes navegadores que tengas instalados en tu equipo. Si tienes varios sistemas operativos repite las pruebas en cada uno de ellos. Si tienes dispositivos móviles, teléfonos o tabletas a tu alcance, prueba también el soporte con los diferentes navegadores. Después de las distintas pruebas:

- ¿Cuál es tu opinión respecto al estado de soporte de WebGL en los diferentes navegadores?

- ¿Crees que es suficiente o que por contra tendremos que esperar aún más a que aparezcan nuevas versiones de los navegadores?
- ¿Piensas que lo que desarrolles en WebGL vas a a tener que probarlo en cada navegador y sistema con el fin de comprobar, no solo su funcionamiento, sino también si se obtiene o no el mismo resultado?

▶ 1.2 Accede a la siguiente web: `http://webglreport.com/`. Obtendrás una página con un contenido similar al que se muestra en la figura 1.4. Aunque muchos términos te resulten extraños, trata de contestar a las siguientes preguntas:

- ¿Cuál es la versión de WebGL que soporta tu navegador?
- ¿Cuántos bits se utilizan para codificar el color de un píxel?

▶ 1.3 Si realizas una búsqueda en internet encontrarás bastantes páginas que ofrecen una selección de ejemplos y de páginas donde los desarrolladores cuelgan sus propios trabajos. A continuación figuran tres de ellos, prueba algunos de los ejemplos que ahí puedes encontrar:

- Chrome Experiments: `http://www.chromeexperiments.com/`
- 22 Experimental WebGL Demo Examples: `http://www.awwwards.com/22-experimental-webgl-demo-examples.html`
- WebGL Samples: `http://webglsamples.org/`

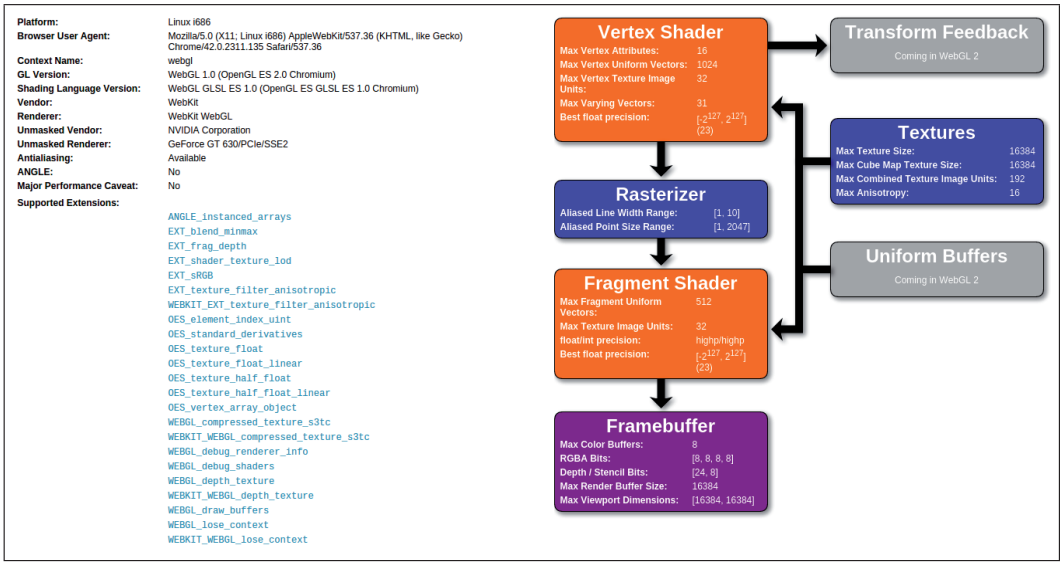


Figura 1.4: Contenido de la página `http://webglreport.org`

1.3. Aplicación WebGL

Para crear páginas web dinámicas es necesario combinar HTML y JAVASCRIPT. Ahora, con WebGL se añade un tercer elemento, el lenguaje GLSL ES, que

es el que se utiliza para escribir los *shaders*. Sin embargo, la estructura general de una aplicación WebGL no es diferente, es decir, sigue siendo la misma que cuando se crean aplicaciones web utilizando únicamente HTML y JAVASCRIPT.

1.3.1. HTML5 y canvas

Un canvas es un elemento rectangular que define el espacio de la página web donde se desea visualizar la escena 3D. El código del listado 1.1 muestra cómo crear un canvas de tamaño 800 por 600.

Listado 1.1: Ejemplo de creación de un canvas con HTML5

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title> Informática Gráfica </title>
    <style type="text/css">
      canvas {border: 1px solid black;}
    </style>
  </head>

  <body>
    <canvas id="myCanvas" width="800" height="600">
      El Navegador no soporta HTML5
    </canvas>
  </body>
</html>
```

Ejercicios

► **1.4** Examina el listado 1.1, utiliza un editor para escribirlo y guárdalo con el nombre *miCanvas.html*. Ahora ábrelo con el navegador que hayas seleccionado para trabajar. Prueba a cambiar algunos de los parámetros como el tamaño, el tipo de borde, o su color. Realmente, que el marco sea visible no es necesario, pero de momento facilita ver claramente cuál es el área de dibujo establecida.

1.3.2. Contexto WebGL

Un contexto WebGL es un objeto JAVASCRIPT a través del cual se accede a toda la funcionalidad de WebGL. Es necesario crear primero el canvas y entonces obtener el contexto a partir de este. Observa el código del listado 1.2 que chequea la disponibilidad de WebGL en el navegador.

Listado 1.2: Obtención de un contexto WebGL

```
function getWebGLContext() {

  var canvas = document.getElementById("myCanvas");

  var names = ["webgl", "experimental-webgl", "webkit-3d",
      "moz-webgl"];

  for (var i = 0; i < names.length; ++i) {
    try {
      return canvas.getContext(names[i]);
    }
    catch(e) {
    }
  }

  return null;

}

function initWebGL() {

  var gl = getWebGLContext();

  if (!gl) {
      alert("WebGL no está disponible");
  } else {
      alert("WebGL disponible");
  }
}

initWebGL();
```

Ejercicios

► **1.5** Examina el listado 1.2, utiliza un editor para escribirlo y guárdalo como *contexto.js*. Ahora recupera *miCanvas.html* y añade el *script* justo antes de cerrar el cuerpo de la página web:

- `<script src="contexto.js"></script>`

Refresca la página en el navegador y comprueba el resultado. ¿Tienes soporte para WebGL?

1.4. El mínimo programa

Observa la nueva función *initWebGL* del listado 1.3. Esta función especifica un color de borrado, o color de fondo, utilizando el método *clearColor*, y or-

dena que borre el contenido del canvas con la orden *clear* y el parámetro *COLOR_BUFFER_BIT*. Ambas instrucciones son ya órdenes de WebGL. Cabe señalar que aunque este programa contiene la mínima expresión de código que utiliza WebGL, la estructura habitual de un programa que utilice WebGL se corresponde con el que se muestra más adelante en el listado 2.4.

Listado 1.3: Inclusión de órdenes WebGL

```
function initWebGL() {

  var gl = getWebGLContext();

  if (!gl) {
    alert("WebGL no está disponible");
    return;
  }

  // especifica en RGBA el color de fondo
  gl.clearColor(1.0,0.0,0.0,1.0);

  // borra el canvas utilizando el color
  // especificado en la línea anterior
  gl.clear(gl.COLOR_BUFFER_BIT);

}
```

Ejercicios

► **1.6** Ejecuta el programa *c01/minimoPrograma.html* que implementa la función *initWebGL* del listado 1.3. Consulta en la guía de programación de WebGL las órdenes *clear* y *clearColor* y contesta a las siguientes cuestiones:

- ¿Qué has de cambiar para que el color de fondo sea amarillo?
- ¿Qué ocurre si intercambias el orden de *clear* y *clearColor*? ¿Por qué?

1.5. El *pipeline*

El funcionamiento básico del *pipeline* se representa en el diagrama simplificado que se muestra en la figura 1.5. Las etapas de procesado del vértice y del fragmento son programables, y es el programador el responsable de escribir los *shaders* que se han de ejecutar en cada una de ellas.

El procesador de vértices acepta vértices como entrada, los procesa utilizando el *shader de vértices* y envía el resultado a la etapa denominada «procesado de la primitiva». En esta etapa, los vértices se reagrupan dependiendo de qué primitiva

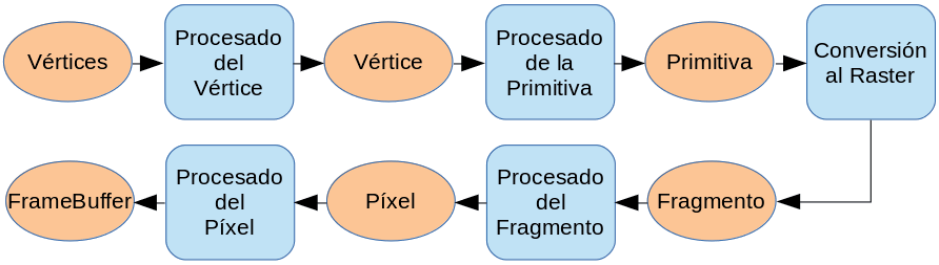


Figura 1.5: Secuencia básica de operaciones del *pipeline* de OpenGL

geométrica se está procesando (puntos, líneas o triángulos). También se realizan otras operaciones que de momento se van a omitir. La primitiva pasa por una etapa de «conversión al *raster*», que básicamente consiste en generar pequeños trozos denominados fragmentos que todos juntos cubren la superficie de la primitiva.

El procesador de fragmentos determina el color definitivo de cada fragmento utilizando el *shader de fragmentos*. El resultado se envía al *framebuffer*, aunque no sin antes atravesar algunas etapas que de momento también se omiten.

1.6. GLSL

El lenguaje GLSL forma parte de WebGL 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. Sin embargo, WebGL 1.0 solo soporta dos tipos de *shaders*: el de vértices y el de fragmentos, por lo que solo es posible escribir códigos para sus dos respectivos procesadores.

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: sampler2D, samplerCube
- Tipos para comunicarse entre *shaders* y con la aplicación: uniform, varying
- 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 WebGL

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. También hay características del C no soportadas en OpenGL, como es el uso de punteros, de 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 puedan declarar en el momento de ser utilizadas. En el listado 1.4 se muestra el código HTML, que incluye un ejemplo de *shader*, el más simple posible.

Los *scripts* identificados como *myVertexShader* y *myFragmentShader* contienen los códigos fuente del *shader* de vértices y del *shader* de fragmentos respectivamente. Estos *scripts* se deben incluir en el cuerpo de la página HTML (ver por ejemplo el listado 2.5).

Cuando desde la aplicación se ordene dibujar un modelo poligonal, cada vértice producirá la ejecución del *shader* de vértices, el cual a su vez produce como salida la posición del vértice que se almacena en la variable predefinida *gl_Position*. El resultado del procesado de cada vértice 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 propósito de este último *shader* es determinar el color definitivo del fragmento. Siguiendo con el ejemplo, todos los fragmentos son puestos a color verde (especificado en formato RGBA) utilizando la variable predefinida *gl_FragColor*.

1.7. Compilación y enlazado de un shader

Los *shaders* han de ser compilados y enlazados antes de poder ser ejecutados en una GPU. El compilador de GLSL está integrado en el propio *driver* de OpenGL instalado en la máquina (ordenador, teléfono, tableta, etc.). 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. Los pasos a realizar son tres:

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

Ejercicios

► **1.7** El listado 1.5 muestra un ejemplo de todo el proceso. Observa detenidamente la función *initShader* e identifica en el código cada una de las tres etapas. Consulta la especificación de WebGL para conocer más a fondo cada una de las órdenes que aparecen en el ejemplo.

► **1.8** Edita el fichero *c01/miPrimerTrianguloConWebGL.js*. Observa cómo queda incluida la función *initShader* dentro de un código más completo y en qué momento se le llama desde la función *initWebGL*.

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 1.4: Un *shader* muy básico

```
<script id="myVertexShader" type="x-shader/x-vertex">

    // Shader de vértices

    // Declaración del atributo posición
    attribute vec3 VertexPosition;

    void main() {

      // se asigna la posición del vértice a
      // la variable predefinida gl_Position
      gl_Position = vec4(VertexPosition,1.0);

    }

</script>

<script id="myFragmentShader" type="x-shader/x-fragment">

    // Shader de fragmentos

    void main() {

      // se asigna el color verde a cada fragmento
      gl_FragColor = vec4(0.0,1.0,0.0,1.0);

    }

</script>
```

Listado 1.5: Compilación y enlazado de un *shader*

```
function initShader() {

    // paso 1
    var vertexShader = gl.createShader(gl.VERTEX_SHADER);
    gl.shaderSource(vertexShader,
            document.getElementById('myVertexShader').text);
    gl.compileShader(vertexShader);

    var fragmentShader = gl.createShader(gl.FRAGMENT_SHADER);
    gl.shaderSource(fragmentShader,
            document.getElementById('myFragmentShader').text);
    gl.compileShader(fragmentShader);

    // paso 2
    var program = gl.createProgram();
    gl.attachShader(program, vertexShader);
    gl.attachShader(program, fragmentShader);

    // paso 3
    gl.linkProgram(program);
    gl.useProgram(program);

    return program;
}
```

Capítulo 2

Modelado poligonal

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

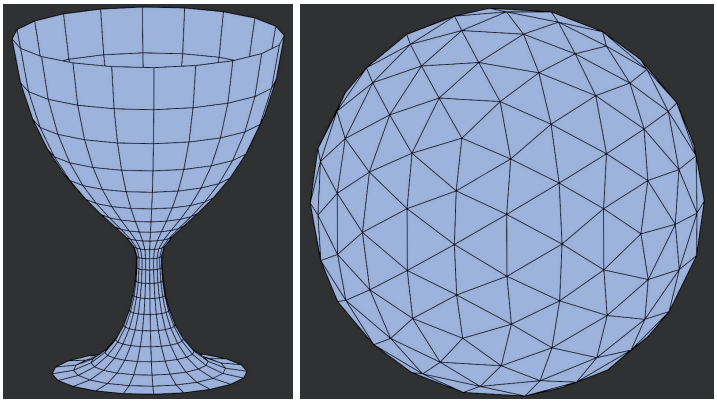


Figura 2.1: A la izquierda, objeto representado mediante cuadriláteros y a la derecha, objeto representado mediante triángulos

El *hardware* gráfico actual se caracteriza por su velocidad a la hora de pintar polígonos. Los fabricantes anuncian desde hace años tasas de dibujado de varios millones de polígonos por segundo. Por esto, el uso de modelos poligonales para visualizar modelos 3D en aplicaciones interactivas es prácticamente una obligación. Por contra, los modelos poligonales representan las superficies curvas de manera aproximada, como se puede observar en la figura 2.1. Sin embargo, hay métodos que permiten visualizar un modelo poligonal de modo que este sea visualmente exacto. Por ejemplo, la figura 2.2 muestra la representación poligonal del modelo de una copa y donde se puede observar que el hecho de que la superficie curva se represente mediante un conjunto de polígonos planos no impide que la observemos como si de una superficie curva se tratara.

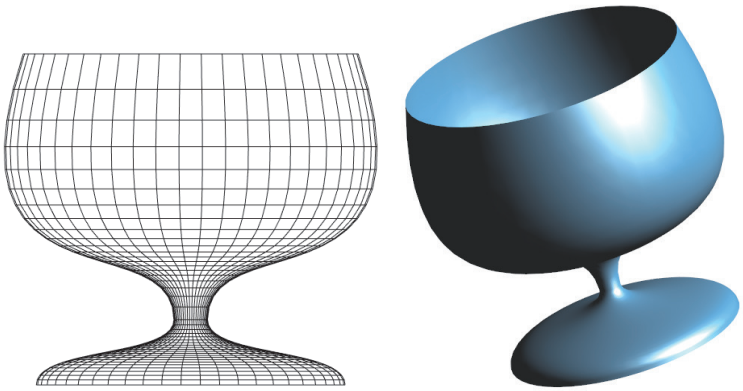


Figura 2.2: Representación poligonal de una copa y resultado de su visualización

2.1. Representación

Normalmente los modelos poligonales representan objetos donde aristas y vértices se comparten entre diferentes polígonos. A este tipo de modelos se les denomina mallas poligonales. La figura 2.3 muestra varios ejemplos. A la hora de definir una estructura para la representación de mallas poligonales es importante tener en cuenta esta característica para tratar de reducir el espacio de almacenamiento, el consumo de ancho de banda y el tiempo de dibujado.

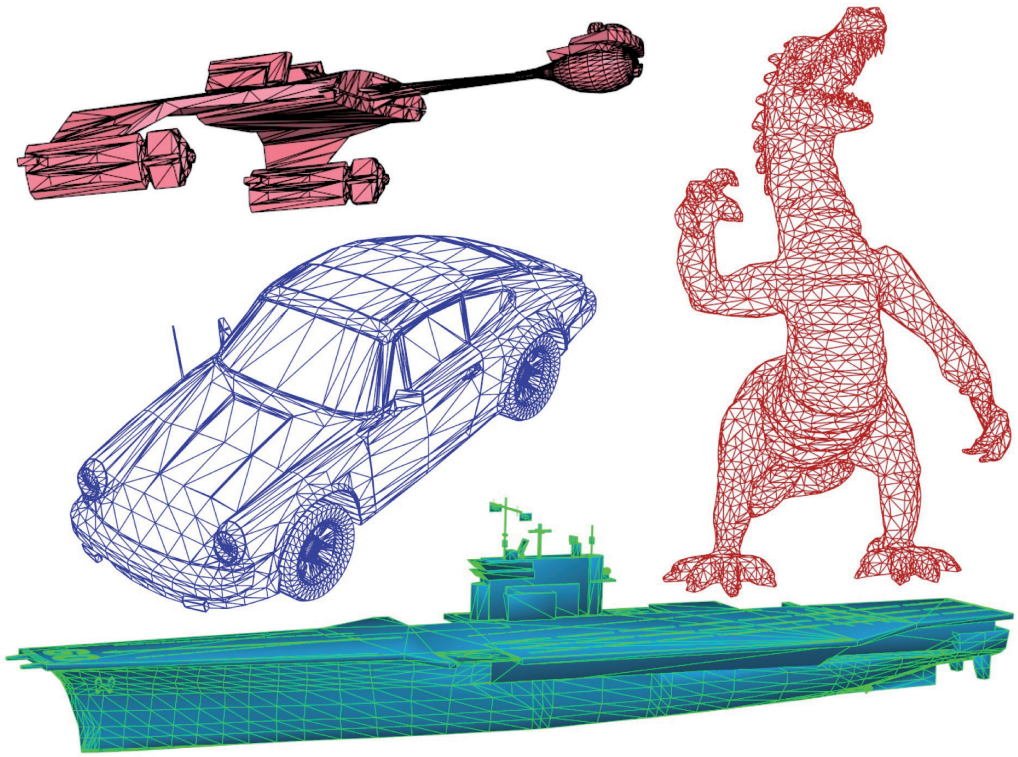


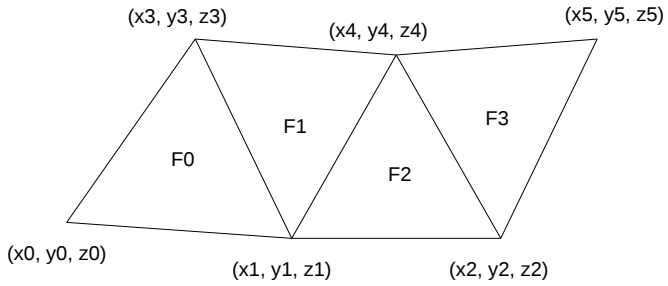
Figura 2.3: Ejemplos de mallas poligonales

2.1.1. Caras independientes

Este tipo de representación se caracteriza por almacenar cada triángulo de manera independiente, como si estos no compartieran información alguna. Aunque es un tipo de estructura muy poco práctica, WebGL la soporta y se utiliza de referencia para realizar comparaciones con otros tipos de representación. Su estructura de datos (ver listado 2.1) se correspondería con una única lista de triángulos, donde para cada triángulo se almacenan las coordenadas de cada uno de sus vértices, tal y como se muestra en la figura 2.4.

Listado 2.1: Estructura correspondiente a la representación de caras independientes

```
struct triangulo {

  vector3 coordenadas[3];

} Triangulos[nTriangulos];
```




```
Triangulos[] = { { x0, y0, z0, x1, y1, z1, x3, y3, z3 },
                 { x1, y1, z1, x4, y4, z4, x3, y3, z3 },
                 { x1, y1, z1, x2, y2, z2, x4, y4, z4 },
                 { x2, y2, z2, x5, y5, z5, x4, y4, z4 } }
```

Figura 2.4: Esquema de almacenamiento de una malla poligonal mediante la estructura de caras independientes

2.1.2. Vértices compartidos

En este caso se separa la información de los vértices y la de los triángulos en dos listas (ver listado 2.2). De esta manera, cada vértice compartido se almacena una única vez y cada triángulo se representa mediante tres índices en la lista de vértices (ver figura 2.5).

Listado 2.2: Estructura correspondiente a la representación de vértices compartidos

```
struct vertice {
  float coordenadas[3];
} Vertices[nVertices];
struct triangulo {
  unsigned int indices[3];
} Triangulos[nTriangulos];
```

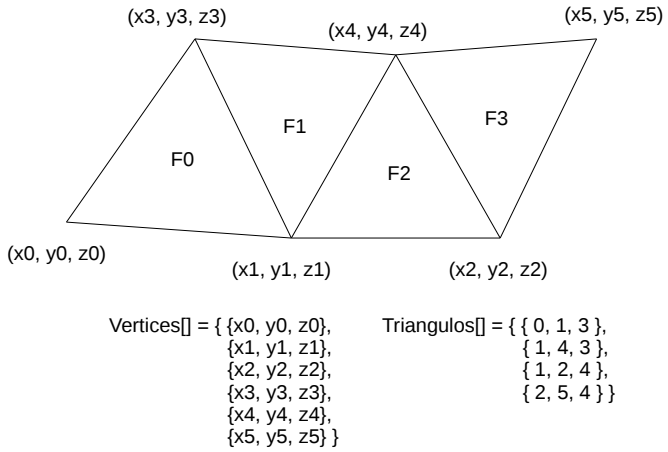


Figura 2.5: Esquema de almacenamiento de una malla poligonal mediante la estructura de vértices compartidos

2.1.3. Tiras y abanicos de triángulos

Estos tipos de representación tratan de aumentar las prestaciones del sistema gráfico creando grupos de triángulos que comparten vértices. En el caso de un grupo de tipo tira de triángulos, 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 (ver figura 2.6). En el caso de un grupo de tipo abanico de triángulos, su primer vértice representa a un vértice común a todos los triángulos del mismo grupo. De nuevo, los primeros tres vértices definen el primero y después cada vértice nuevo sustituye siempre al segundo del triángulo anterior (ver figura 2.6).

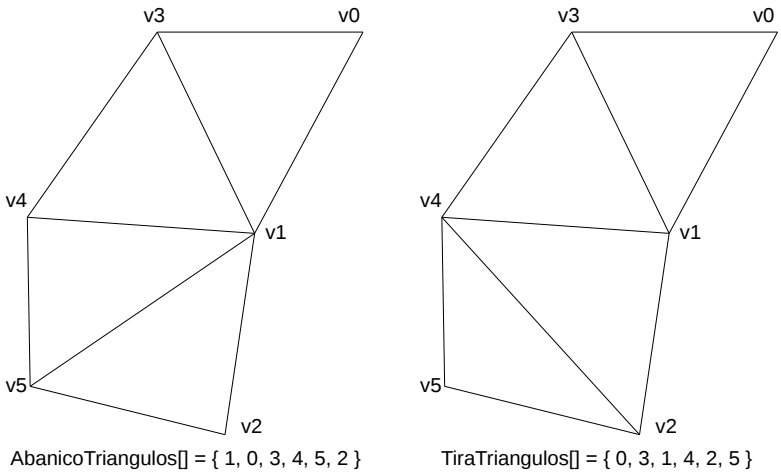


Figura 2.6: Ejemplo de abanico y tira de triángulos

2.2. La normal

Un modelo poligonal, 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 2.7 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.

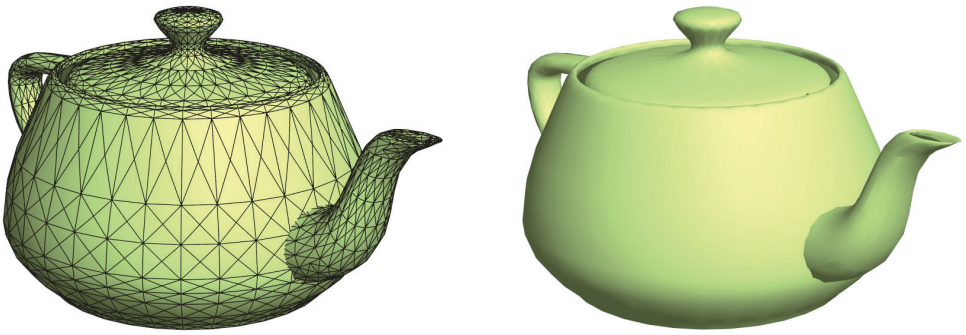


Figura 2.7: Visualización del modelo poligonal de una tetera. En la imagen de la izquierda se pueden obervar los polígonos utilizados para representarla

La normal es un vector perpendicular a la superficie en un punto. Si la superficie es plana, la normal es la misma para todos los puntos de la superficie. Para un triángulo es fácil obtenerla realizando el producto vectorial de dos de los vectores directores de sus aristas. Ya que el producto vectorial no es conmutativo, 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 se referencia con 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?
- Calcula la normal para cada vértice, ¿qué problema encuentras?
- Obtén las tiras de triángulos que representan el objeto, ¿puedes conseguirlo con solo una tira?

2.3. Mallas y WebGL

2.3.1. Tipos de primitivas geométricas

Las primitivas básicas de dibujo en WebGL son el punto, el segmento de línea y el triángulo. Cada primitiva se define especificando sus respectivos vértices, y estas se agrupan a su vez para definir objetos de mayor complejidad. En WebGL, la primitiva geométrica se utiliza para especificar cómo han de ser agrupados 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 independientes: gl.LINES
  - Secuencia o tira de segmentos: gl.LINE_STRIP
  - Secuencia cerrada de segmentos: gl.LINE_LOOP
- Triángulos:
  - Triángulos independientes: gl.TRIANGLES
  - Tira de triángulos: gl.TRIANGLE_STRIP
  - Abanico de triángulos: gl.TRIANGLE_FAN

2.3.2. Descripción de la geometría

Habitualmente se suele asociar el concepto de vértice con las coordenadas que definen la posición de un punto en el espacio. En WebGL, el concepto de vértice es más general, entendiéndose como una agrupación de datos a los que se denominan atributos. Estos pueden ser de cualquier tipo: reales, enteros, vectores, etc. Los más utilizados son la posición, la normal y el color, pero WebGL permite que el programador pueda incluir como atributo cualquier información que para él tenga sentido y que necesite tener disponible en el *shader*.

WebGL 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 WebGL 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 (ver listado 2.3).

2.3.3. Visualización

En primer lugar, el modelo poligonal se ha de almacenar en *buffer objects*. Un *buffer object* no es más que una porción de memoria reservada dinámicamente y controlada por el propio procesador gráfico. Siguiendo con el ejemplo del listado 2.3 se necesitan dos *buffers*, uno para el vector de vértices y otro para el de índices. Después hay que asignar a cada *buffer* sus datos correspondientes. El listado 2.4 recoge estas operaciones en la función *initBuffers*, examínalo y acude a la especificación del lenguaje para conocer más detalles de las funciones utilizadas.

Listado 2.3: Ejemplo de modelo de un cubo definido con triángulos preparado para ser utilizado con WebGL

```
var exampleCube = {

    "vertices" : [-0.5, -0.5,  0.5,
                   0.5, -0.5,  0.5,
                   0.5,  0.5,  0.5,
                  -0.5,  0.5,  0.5,
                  -0.5, -0.5, -0.5,
                   0.5, -0.5, -0.5,
                   0.5,  0.5, -0.5,
                  -0.5,  0.5, -0.5],

    "indices" : [ 0, 1, 2, 0, 2, 3,
                  1, 5, 6, 1, 6, 2,
                  3, 2, 5, 3, 6, 7,
                  4, 6, 5, 4, 7, 6,
                  0, 7, 4, 0, 3, 7,
                  0, 5, 1, 0, 4, 5]

};
```

El listado 2.5 muestra el código HTML que incluye dos *scripts*, uno para cada *shader*, el de vértices y el de fragmentos. Observa que el *shader* de vértices define un único atributo para cada vértice, su posición. El segundo paso consiste en obtener los índices de las variables del *shader* que representan los atributos de los vértices (que de momento es solo la posición) y habilitar el vector correspondiente. Estas operaciones se muestran en el listado 2.4 y se han incluido al final de la función *initShaders*, que es la encargada de compilar, enlazar y crear un ejecutable del *shader*.

Ahora que el modelo ya se encuentra almacenado en la memoria controlada por la GPU, el *shader* ya está compilado y enlazado, y los índices de los atributos de los vértices ya se han obtenido, sólo queda el último paso: su visualización.

Primero, hay que indicar los *buffers* que contienen los vértices y los índices correspondientes al modelo que se va a visualizar. También hay que especificar cómo se encuentran dispuestos cada uno de los atributos de los vértices en el *buffer* correspondiente. 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. Estas operaciones se muestran en el listado 2.4, en la función *draw*. De nuevo, acude a la especificación del lenguaje para conocer más detalles de las órdenes utilizadas.

Ejercicios

► **2.2** Abre con el navegador el archivo *c02/visualiza.html*, que incluye el código del listado 2.5, y produce a su vez la carga de *c02/visualiza.js*, que contiene el código del listado 2.4. Ahora responde a la siguiente pregunta:

- ¿Qué objeto se muestra?

► **2.3** Realiza las modificaciones oportunas en el código de *visualiza.js* para pintar un pentágono. A continuación tienes el modelo del pentágono:

```
var examplePentagon = {

  "vertices" : [ 0.0,  0.9, 0.0,
                -0.95, 0.2, 0.0,
                -0.6, -0.9, 0.0,
                 0.6, -0.9, 0.0,
                 0.95, 0.2, 0.0],

  "indices" : [ 0, 1, 2, 3, 4]

};
```

Ahora dibújalo utilizando puntos primero y después líneas:

- Puntos: `gl.drawElements(gl.POINTS, 5, gl.UNSIGNED_SHORT, 0);`
- Líneas: `gl.drawElements(gl.LINE_LOOP, 5,gl.UNSIGNED_SHORT,0);`

► **2.4** Continúa utilizando los mismos vértices del modelo del pentágono y realiza las modificaciones necesarias para obtener una estrella como la de la figura 2.8. Prueba también a cambiar el grosor del trazo mediante la orden `gl.lineWidth(5.0)` justo antes de ordenar el dibujado de la geometría.

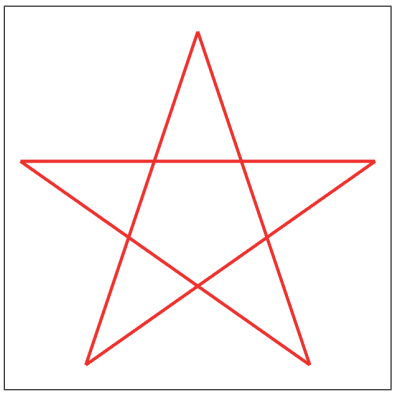


Figura 2.8: Ejemplo de estrella

► **2.5** Utiliza ahora la descripción de los vértices que figura a continuación. Crea una función que se llame *drawGeometryLines* y que utilizando dichos vértices dibuje la estrella con líneas de manera que se obtenga el resultado de la figura 2.9 (imagen de la izquierda). Crea otra función que se llame *drawGeometryTriangles* y que dibuje la estrella mediante triángulos independientes de manera que se obtenga el resultado que se muestra en la figura 2.9 (imagen de la derecha).

```
"vertices" : [0.0,   0.9,   0.0,
              -0.95,  0.2,   0.0,
              -0.6,  -0.9,   0.0,
               0.6,  -0.9,   0.0,
               0.95,  0.2,   0.0,
               0.0,  -0.48,  0.0,
               0.37, -0.22,  0.0,
               0.23,  0.2,   0.0,
              -0.23,  0.2,   0.0,
              -0.37, -0.22,  0.0],
```

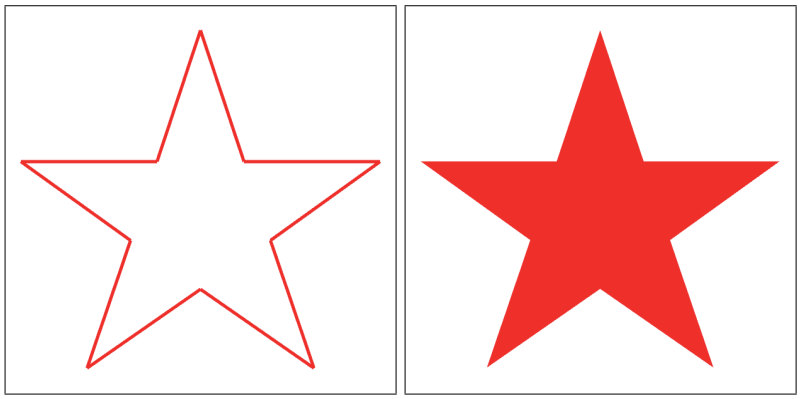


Figura 2.9: Estrella dibujada con líneas (izquierda) y con triángulos (derecha)

Listado 2.4: Código mínimo que se corresponde con la estructura básica de un programa que utiliza WebGL (disponible en *c02/visualiza.js*)

```
var gl, program;

var exampleTriangle = {

  "vertices" : [-0.7, -0.7, 0.0,
                 0.7, -0.7, 0.0,
                 0.0,  0.7, 0.0],
```

```
  "indices" : [ 0, 1, 2]

};

function getWebGLContext() {

  var canvas = document.getElementById("myCanvas");

  var names = ["webgl", "experimental-webgl", "webkit-3d",
      "moz-webgl"];

  for (var i = 0; i < names.length; ++i) {
    try {
      return canvas.getContext(names[i]);
    }
    catch(e) {
    }
  }

  return null;

}

function initShaders() {

  var vertexShader = gl.createShader(gl.VERTEX_SHADER);
  gl.shaderSource(vertexShader,
      document.getElementById("myVertexShader").text);
  gl.compileShader(vertexShader);

  var fragmentShader = gl.createShader(gl.FRAGMENT_SHADER);
  gl.shaderSource(fragmentShader,
      document.getElementById("myFragmentShader").text);
  gl.compileShader(fragmentShader);

  program = gl.createProgram();
  gl.attachShader(program, vertexShader);
  gl.attachShader(program, fragmentShader);

  gl.linkProgram(program);

  gl.useProgram(program);

  // Obtener la referencia del atributo posición
  program.vertexPositionAttribute =
       gl.getAttribLocation( program, "VertexPosition");

  // Habilitar el atributo posición
  gl.enableVertexAttribArray(program.vertexPositionAttribute);

}
```

```
function initBuffers(model) {

  // Buffer de vértices
  model.idBufferVertices = gl.createBuffer ();
  gl.bindBuffer (gl.ARRAY_BUFFER, model.idBufferVertices);
  gl.bufferData (gl.ARRAY_BUFFER,
      new Float32Array(model.vertices), gl.STATIC_DRAW);

  // Buffer de índices
  model.idBufferIndices = gl.createBuffer ();
  gl.bindBuffer (gl.ELEMENT_ARRAY_BUFFER, model.idBufferIndices);
  gl.bufferData (gl.ELEMENT_ARRAY_BUFFER,
      new Uint16Array(model.indices), gl.STATIC_DRAW);
}

function initRendering() {
  gl.clearColor(0.15,0.15,0.15,1.0);
}

function draw(model) {

  gl.bindBuffer(gl.ARRAY_BUFFER, model.idBufferVertices);
  gl.vertexAttribPointer(program.vertexPositionAttribute, 3,
      gl.FLOAT, false, 0, 0);

  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, model.idBufferIndices);
  gl.drawElements(gl.TRIANGLES, 3, gl.UNSIGNED_SHORT, 0);
}

function drawScene() {

  gl.clear(gl.COLOR_BUFFER_BIT);
  draw(exampleTriangle);
}

function initWebGL() {

  gl = getWebGLContext();
  if (!gl) {
    alert("WebGL no está disponible");
    return;
  }

  initShaders();
  initBuffers(exampleTriangle);
  initRendering();
  requestAnimationFrame(drawScene);
}

initWebGL();
```

Listado 2.5: Código HTML que incluye un canvas y los dos *shaders* básicos (disponible en *c02/visualiza.html*)

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title> Informática Gráfica </title>
    <style type="text/css">
      canvas {border: 1px solid black;}
    </style>
  </head>
  <body>

    <canvas id="myCanvas" width="800" height="600">
      El Navegador no soporta HTML5
    </canvas>

    <script id="myVertexShader" type="x-shader/x-vertex">

      attribute vec3 VertexPosition;

      void main() {

        gl_Position = vec4 (VertexPosition,1.0);

      }

    </script>

    <script id="myFragmentShader" type="x-shader/x-fragment">

      void main() {

        gl_FragColor = vec4(0.0,1.0,0.0,1.0);

      }

    </script>

    <script src = "visualiza.js"></script>

  </body>
</html>
```

2.3.4. Variables uniform

Imagina que por ejemplo se desea dibujar la estrella de los ejercicios anteriores cada vez de un color diferente. Para lograrlo es necesario que el color que figura en el *shader* de fragmentos sea una variable, tal y como se muestra en el listado 2.6.

Listado 2.6: Ejemplo de variable *uniform* en el *shader* de fragmentos

```
precision mediump float;

uniform vec4 myColor;

void main() {

  gl_FragColor = myColor;

}
```

La variable *myColor* es de tipo *uniform* porque su valor será constante para todos los fragmentos que reciba procedentes de procesar la estrella, pero se puede hacer que sea diferente para cada estrella cambiando su valor antes de dibujarla. Observa el fragmento de código del listado 2.7. Esas líneas son las encargadas de obtener el índice de la variable *myColor* en el *shader* y de especificar su valor. Mientras que la primera línea solo es necesario ejecutarla una vez, la segunda habrá que utilizarla cada vez que se necesite asignar un nuevo valor a la variable *myColor*. Así, la primera línea se podría añadir al final de la función *initShaders*, mientras que la segunda línea habrá que añadirla justo antes de ordenar el dibujado del modelo.

Listado 2.7: Ejemplo de asignación de una variable *uniform*

```
var idMyColor = gl.getUniformLocation(program, "myColor");
gl.uniform4f(idMyColor, 1, 0, 1, 1);
```

Ejercicios

► **2.6** Dibuja dos estrellas, una para pintar el interior de un color y la otra para pintar el borde de un color diferente (ver figura 2.10). Elije colores que contrasten entre sí. Ten en cuenta también que el orden de dibujado es importante, piensa qué estrella deberías dibujar en primer lugar. Nota: puedes definir dos vectores de índices con diferente nombre y crear un *buffer* adicional en la función *initBuffers*.

2.3.5. Variables varying

Hasta el momento, cada vértice consta únicamente de un único atributo: su posición. Ahora se va a añadir un segundo atributo, por ejemplo, un valor de color para cada vértice. La primera modificación necesaria consiste en ampliar la información de cada vértice para contener su valor de color. El listado 2.8 muestra el

modelo de un triángulo tras añadir los cuatro componentes de color RGBA a cada vértice.

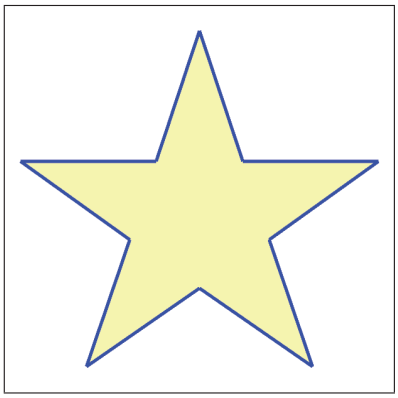


Figura 2.10: Ejemplo de dos estrellas: una aporta el color interior y la otra el color del borde

Listado 2.8: Ejemplo de modelo con dos atributos por vértice: posición y color

```
var exampleTriangle = {

  "vertices" : [
      -0.7, -0.7, 0.0,    1.0, 0.0, 0.0, 1.0,  // rojo
       0.7, -0.7, 0.0,    0.0, 1.0, 0.0, 1.0,  // verde
       0.0,  0.7, 0.0,    0.0, 0.0, 1.0, 1.0], // azul

  "indices" : [ 0, 1, 2]

};
```

Los cambios correspondientes al *shader* se recogen en el listado 2.9. Se puede observar que ahora están declarados los dos atributos en el *shader* de vértices. La variable *colorOut* se ha declarado con el identificador *varying* tanto en el *shader* de vértices como en el *shader* de fragmentos. A una variable de tipo *varying* se le asigna un valor en el *shader* de vértices para cada vértice del modelo. En el *shader* de fragmentos, la variable *colorOut* tendrá un valor convenientemente interpolado para cada fragmento de la primitiva que se esté dibujando. La interpolación la realiza automáticamente la GPU a partir de los valores asignados a *colorOut* en cada vértice. La figura 2.11 muestra el resultado.

Por supuesto, también es necesario habilitar los dos atributos correspondientes, tal y como se muestra en el listado 2.10. Estúdialo y contesta a esta pregunta, ¿dónde colocarías este código en el listado 2.4?

Listado 2.9: Ejemplo de *shader* con dos atributos por vértice: posición y color

```
// Shader de vértices

attribute vec3 VertexPosition;
attribute vec4 VertexColor;

varying vec4 colorOut;

void main()  {

  colorOut    = VertexColor;
  gl_Position = vec4(VertexPosition, 1);

}


// Shader de fragmentos

precision mediump float;

varying vec4 colorOut;

void main() {

  gl_FragColor = colorOut;

}
```

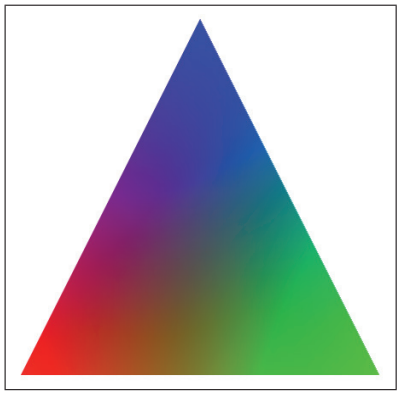


Figura 2.11: Resultado de visualizar un triángulo con un valor de color diferente para cada vértice

Por último, hay que modificar la función de dibujo. Observa la nueva función en el listado 2.11, que recoge los cambios necesarios. Presta atención a los dos últimos parámetros de las dos llamadas a *gl.vertexAttribPointer*. Consulta la documentación para entender el por qué de esos valores.

Listado 2.10: Localización y habilitación de los dos atributos: posición y color

```
vertexPositionAttribute =
    gl.getAttribLocation(program, "VertexPosition");
gl.enableVertexAttribArray(vertexPositionAttribute);

vertexColorAttribute =
    gl.getAttribLocation(program, "VertexColor");
gl.enableVertexAttribArray(vertexColorAttribute);
```

Listado 2.11: Dibujo de un modelo con dos atributos por vértice

```
function draw() {

    gl.bindBuffer(gl.ARRAY_BUFFER, idVerticesBuffer);
    gl.vertexAttribPointer(vertexPositionAttribute, 3, gl.FLOAT,
        false, 7*4, 0);
    gl.vertexAttribPointer(vertexColorAttribute,    4, gl.FLOAT,
        false, 7*4, 3*4);

    gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, idIndicesBuffer);
    gl.drawElements(gl.TRIANGLES, 3, gl.UNSIGNED_SHORT, 0);
}
```

Ejercicios

► **2.7** Observa la imagen de la figura 2.12, ¿qué colores se han asignado a los vértices? Edita *c02/trianguloVarying.html*, que ya incluye los fragmentos de código que se han explicado en esta sección, y modifica los valores de color para comprobar si has acertado.

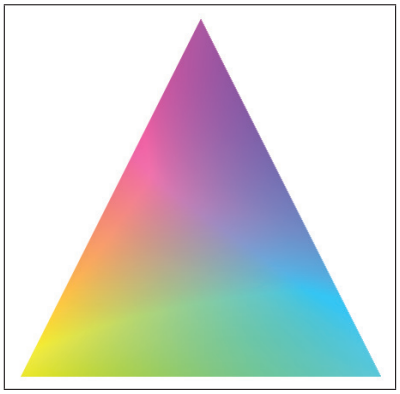


Figura 2.12: Resultado de visualizar un triángulo con un valor de color diferente para cada vértice

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. La figura 3.1 muestra algunos ejemplos de objetos obtenidos mediante la aplicación de transformaciones a primitivas geométricas simples como el cubo, la esfera o el toro.

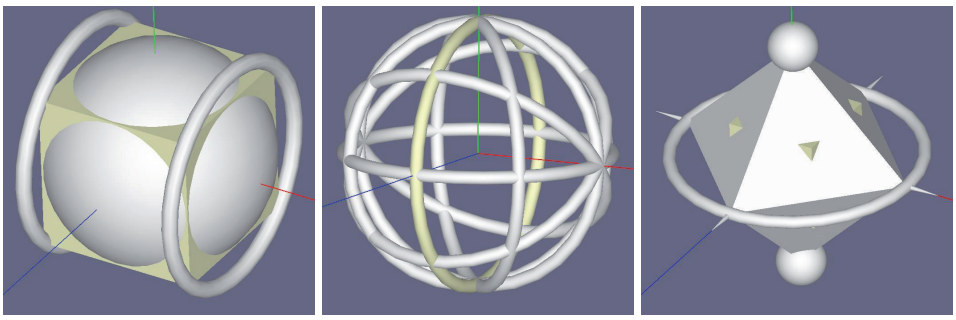


Figura 3.1: Ejemplos de objetos creados utilizando transformaciones geométricas

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:

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

Utilizando esta representación, el nuevo punto se obtiene así:

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

donde $\tilde{p} = (p_x, p_y, p_z, 1)^T$ y $\tilde{q} = (q_x, q_y, q_z, 1)^T$, 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: $\tilde{q} = R(\phi) \cdot \tilde{p}$.

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 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.

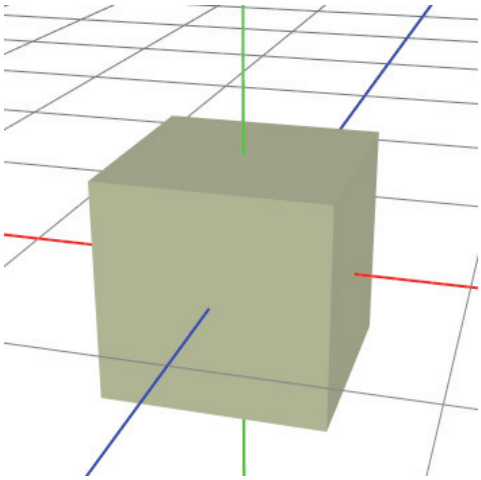
Por ejemplo, piensa en una esfera con radio una unidad centrada en el origen de coordenadas y en las dos siguientes matrices de transformación T y S: $T(5, 0, 0)$ desplaza la componente x cinco unidades; $S(5, 5, 5)$ escala las tres componentes con un factor de cinco. Ahora, dibuja en el papel cómo quedaría 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.

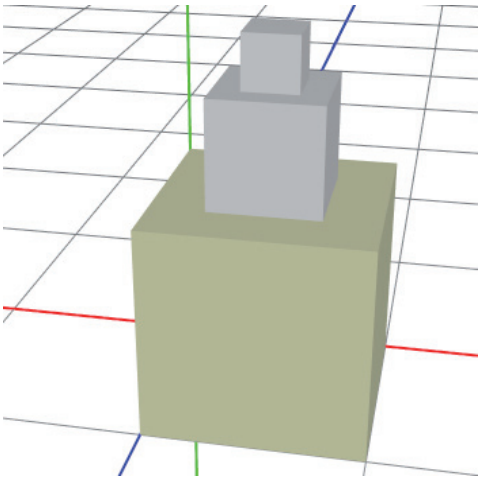
Ejercicios

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 de la del cubo anterior. Detalla las transformaciones utilizadas.

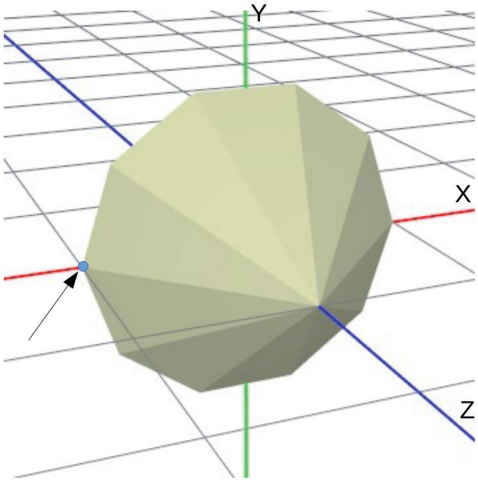


(a)

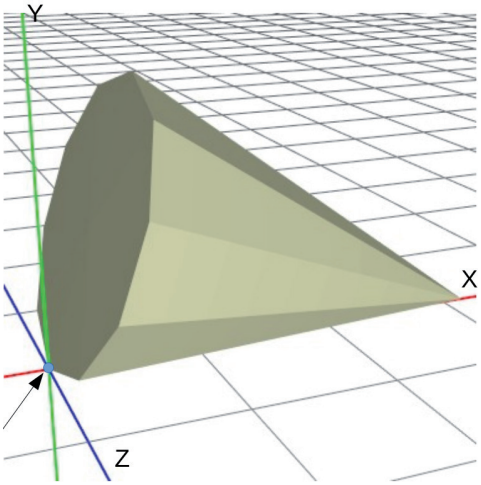


(b)

► **3.3** Determina las transformaciones que sitúan el cono que se muestra en la figura (*c*) (radio de la base y altura de valor 1) en la posición que se muestra en la figura (*d*) (radio de la base de valor 1 y altura de valor 3). Ten en cuenta el punto de referencia señalado con una flecha, de manera que quede ubicado tal y como se observa en la figura.

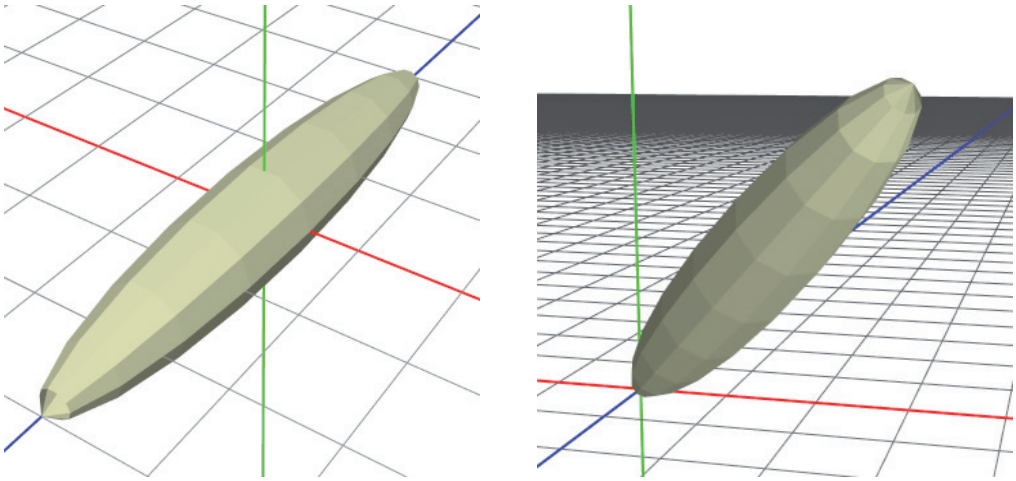


(c)



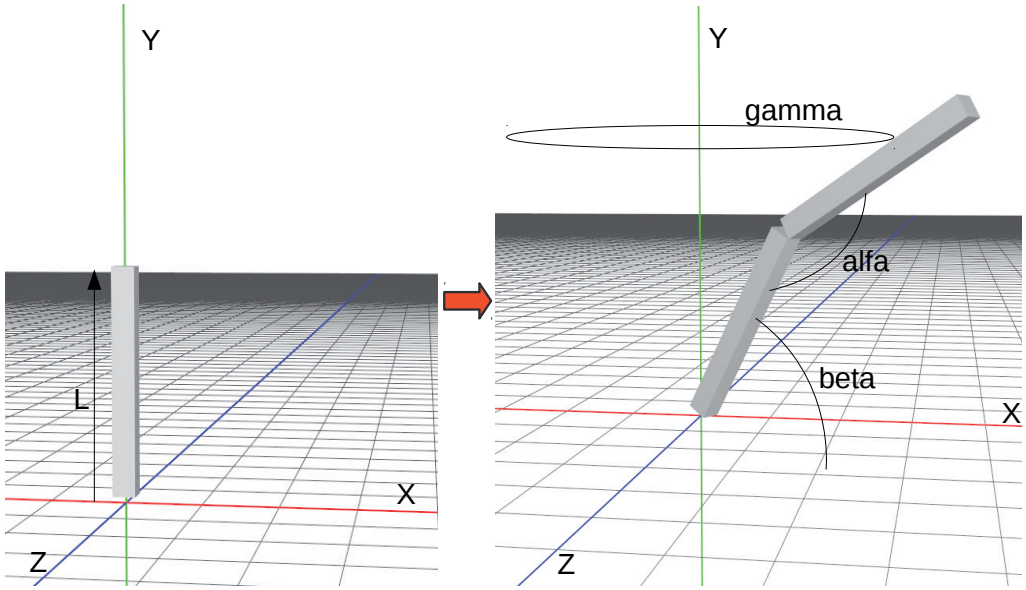
(d)

► **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)

► **3.5** Observa la posición inicial del objeto en la figura de la izquierda. Este objeto lo vas a utilizar dos veces para crear el brazo articulado que se muestra en la figura de la derecha. Determina las transformaciones que has de aplicarle. La posición final del brazo ha de depender de los ángulos $alfa$, $beta$ y $gamma$.



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. En concreto, esto ocurre cuando se utilizan transformaciones de escalado no uniforme (ver figura 3.2). 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.

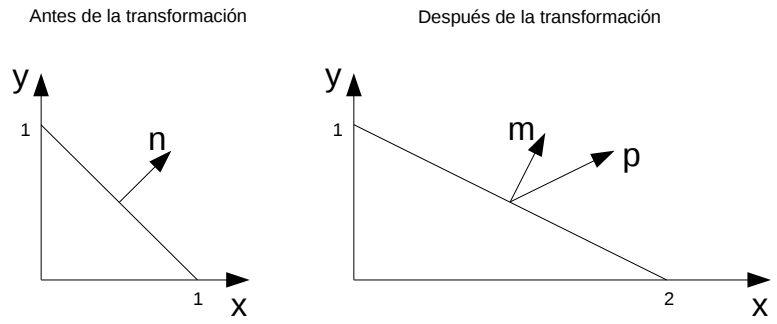


Figura 3.2: En la imagen de la izquierda se representa un polígono y su normal n. En la imagen de la derecha se muestra el mismo polígono tras aplicar una transformación de escalado no uniforme $S(2, 1)$. Si se aplica esta transformación a la normal n, se obtiene p como vector normal en lugar de m, que 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. Como esta operación es cara computacionalmente, ¿se te ocurre alguna manera de evitarla?

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 (ver figura 3.3). 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.

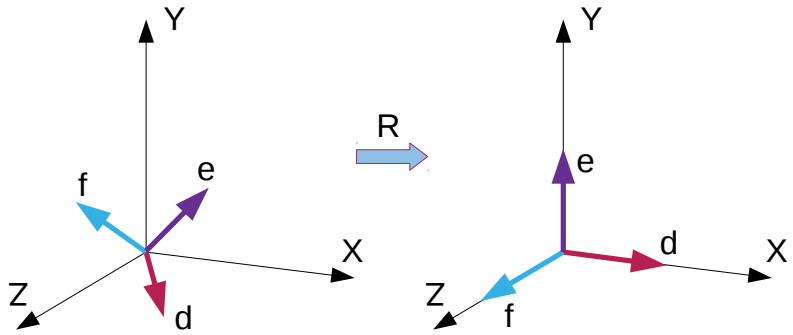


Figura 3.3: La nueva base formada por los vectores d, e y f se transforma para coincidir con los ejes de coordenadas

Así, teniendo en cuenta que R es ortogonal, es decir, que 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. La biblioteca GLMATRIX

Como ayuda a la programación, la biblioteca GLMATRIX 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 (out, in, v)
- mat4.scale (out, in, v)
- mat4.rotate (out, in, rad, $axis$)

```
var M = mat4.create();
var T = mat4.create();
var S = mat4.create();
mat4.translate (T, T, [10,0,0]);
mat4.scale     (S, S, [2,2,2]);
mat4.multiply  (M, T, S);
```

Así, por ejemplo, la matriz de transformación M que escala un objeto al doble de su tamaño y después lo traslada en dirección del eje X un total de diez unidades, se obtendría de lasiguiente forma:

¿Qué ocurriría si en el ejemplo, en vez de $mat4.multiply(M, T, S)$, apareciera $mat4.multiply(M, S, T)$?

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

```
var N = mat3.create();
mat3.fromMat4 (N,M);
mat3.invert   (N,N);
mat3.transpose(N,N);
```

3.6. Transformaciones en WebGL

WebGL no proporciona modelos de primitivas geométricas 3D. Por lo tanto, y en primer lugar, es necesario obtener una descripción geométrica de primitivas básicas como el cubo, la esfera, el cono, etc. Por ejemplo, el listado 2.3 mostraba el fragmento de código que define un cubo centrado en el origen de coordenadas, con sus caras paralelas a los planos de referencia XY, XZ e YZ, y lado de valor 1. Observa que este modelo consta de dos vectores, uno contiene las coordenadas de los vértices y el otro contiene los índices al vector de vértices que de tres en tres describen los triángulos. El índice del primer vértice es 0. De momento se utilizan modelos que solo constan de geometría, es decir, no contienen otros atributos (normal, color, etc.), por lo que se visualizan en alambre, es decir, solo las aristas (ver listado 3.1).

Observa ahora la nueva función $drawScene()$ en el listado 3.2. Esta función incluye ahora los tres pasos necesarios para dibujar una primitiva con matrices de transformación. En primer lugar, se obtiene la matriz de transformación del modelo M, que en este caso se trata de un escalado a la mitad de su tamaño. Esta matriz se ha de multiplicar con cada vértice del objeto para que este quede ubicado en su posición, tamaño y orientación definitiva. Esta operación, como se repite para cada

Listado 3.1: Visualización en alambre de un modelo

```
function draw(model) {

  gl.bindBuffer (gl.ARRAY_BUFFER, model.idBufferVertices);
  gl.vertexAttribPointer (vertexPositionAttribute, 3,
                          gl.FLOAT, false, 3*4, 0);

  gl.bindBuffer (gl.ELEMENT_ARRAY_BUFFER, model.idBufferIndices);
  for (var i = 0; i < model.indices.length; i += 3)
    gl.drawElements (gl.LINE_LOOP, 3, gl.UNSIGNED_SHORT, i*2);

}
```

vértice del objeto, se ha de realizar en el procesador de vértices. Así, en el segundo paso, se establece el valor de la matriz M en el *shader* de vértices. Finalmente, en el paso 3, se llama a la función $draw$, que es la que produce el dibujado de la primitiva que se recibe como parámetro.

Listado 3.2: Ejemplo de los pasos necesarios para dibujar un objeto transformado

```
function drawScene() {

  gl.clear(gl.COLOR_BUFFER_BIT);

  var M  = mat4.create();
  var iM = gl.getUniformLocation(program,"M");

  gl.lineWidth(1.5); // ancho de línea

  // 1. calcula la matriz de transformación
  mat4.identity(M);
  mat4.scale    (M, M, [0.5,0.5,0.5]);

  // 2. establece la matriz M en el shader de vértices
  gl.uniformMatrix4fv(iM, false, M);

  // 3. dibuja la primitiva
  draw(exampleCube);

}
```

En el *shader* de vértices se han de incluir las operaciones que multiplican cada vértice por su correspondiente matriz de transformación (ver listado 3.3). Observa que las líneas comentadas corresponden al código que haría falta en el caso de que, además, se suministrara el atributo de la normal para cada vértice.

Listado 3.3: *Shader* de vértices para transformar la posición de cada vértice

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

attribute vec3 VertexPosition;
// attribute vec3 VertexNormal;
// varying   vec3 VertexNormalT;

void main ()
{
  // VertexNormalT = normalize (N * VertexNormal);
  gl_Position = M * vec4(VertexPosition, 1.0);
}
```

Ejercicios

► **3.6** Edita *c03/transforma.html* y *c03/transforma.js*. Comprueba que se han incluido todos los cambios explicados en esta sección. Observa también *c03/primitivas.js*, que incluye los modelos de algunas primitivas geométricas simples. Echa un vistazo a la descripción de los modelos. Prueba a visualizar las diferentes primitivas.

► **3.7** Modela la típica grúa de obra cuyo esquema se muestra en la figura 3.4 utilizando como única primitiva cubos de lado 1 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.

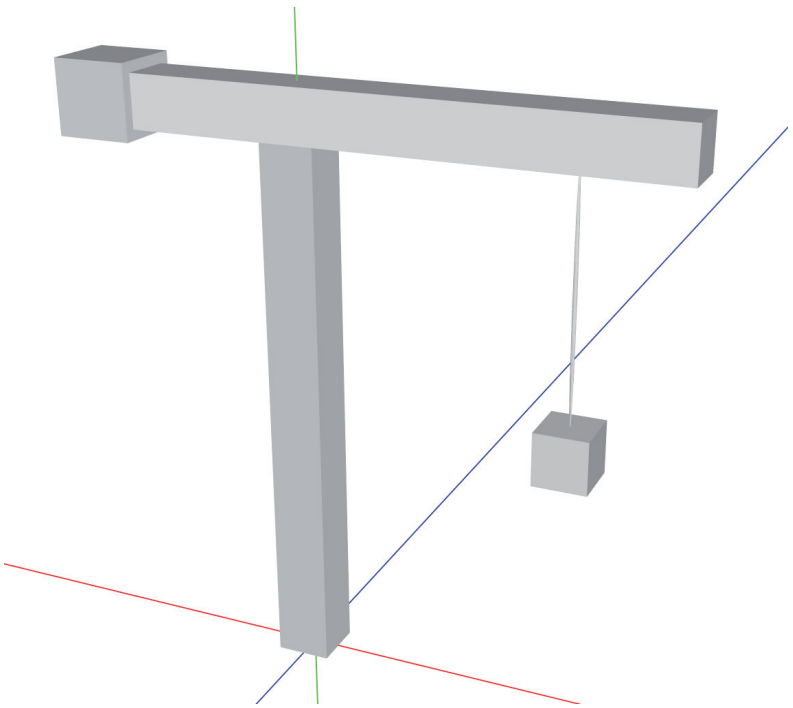


Figura 3.4: Ejemplo de una grúa de obra

► **3.8** Observa la escena que se muestra en la figura 3.5, inspirada en la figura de bloques de madera que se muestra abajo. Crea una escena que produzca la misma salida utilizando únicamente la primitiva cubo. El cubo central tiene de lado $0{,}1$, y los que están a su lado tienen la misma base pero una altura que va incrementándose en $0{,}1$ a medida que se alejan del central. Utiliza un bucle para pintar los trece cubos. Ambas imágenes muestran la misma escena, pero en la imagen de la derecha se ha girado toda la escena para apreciar mejor que se trata de primitivas 3D.

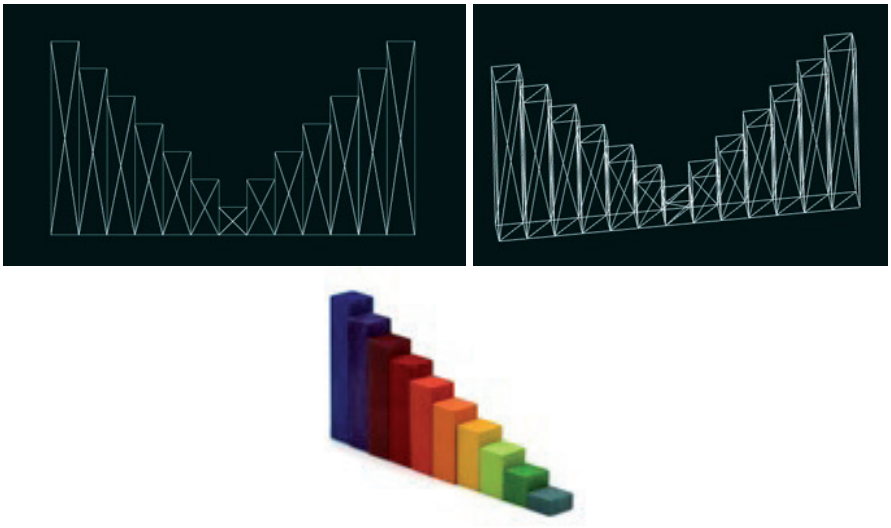


Figura 3.5: Juego de bloques de madera coloreados

► **3.9** Internet es una fuente de inspiración excepcional, utilízala para buscar ejemplos que te sirvan de muestra y practicar con el uso de las transformaciones geométricas. Fíjate en los que aparecen en la figura 3.6.

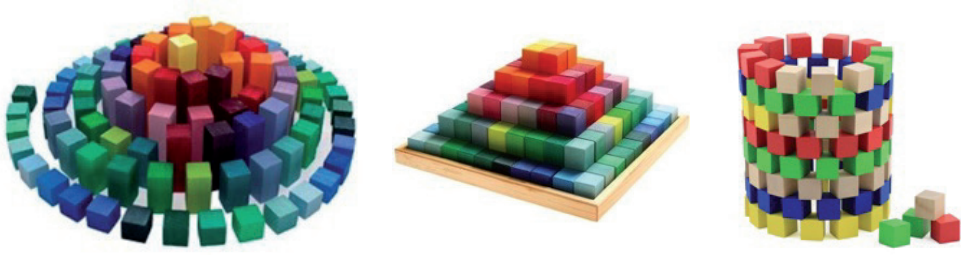


Figura 3.6: Otros ejemplos de juegos de bloques de madera coloreados.

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 transforma el contenido de este volumen al volumen canónico de la vista.
- Transformación al área de dibujo: el contenido del volumen canónico de la vista se transforma para ubicarlo en el espacio de la ventana destinado a mostrar el resultado de 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

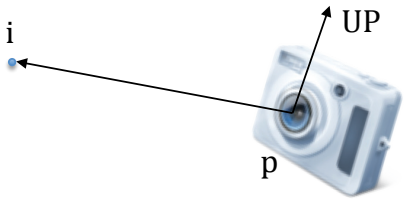


Figura 4.1: Parámetros para ubicar y orientar una cámara: p, posición de la cámara; UP, vector de inclinación; i, punto de interés

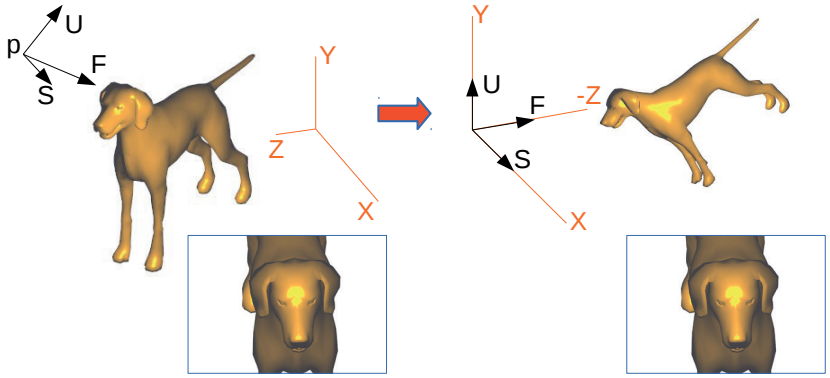


Figura 4.2: Transformación de la cámara. La cámara situada en el punto p en la imagen de la izquierda se transforma para quedar como se observa en la imagen de la derecha. Dicha transformación se aplica al objeto de tal manera que lo que se observa sea lo mismo en ambas situaciones

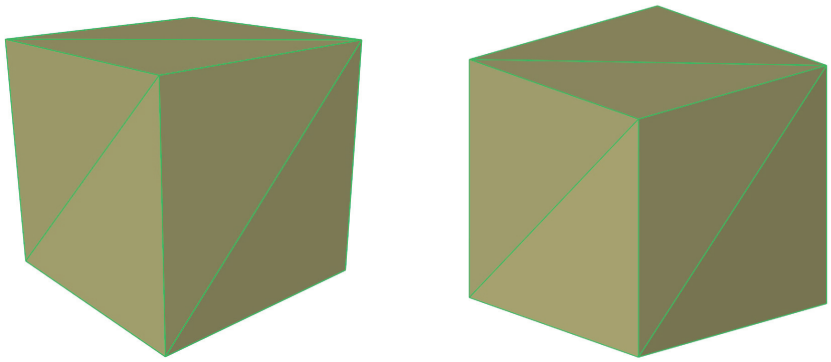
razón, es necesario aplicar una transformación al mundo 3D de manera que, desde la posición y orientación requeridas por el sistema gráfico, se observe lo mismo que desde donde el usuario estableció su cámara (ver figura 4.2). A esta transformación se le denomina *transformación de la cámara*.

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 requeridas 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



(a) (b)

Figura 4.3: Vista de un cubo obtenida con: (a) vista perspectiva y (b) vista paralela

proyección. Así, hay dos tipos de vistas:

- Vista perspectiva. Es 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. Es la utilizada principalmente en ingeniería o arquitectura. Se caracteriza por preservar longitudes y ángulos.

La figura 4.3 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. El volumen de la vista tiene forma de caja, la cual, se alinea con los ejes de coordenadas tal y como se muestra en la figura 4.4, donde también se han indicado los nombres de los seis parámetros que definen dicho volumen.

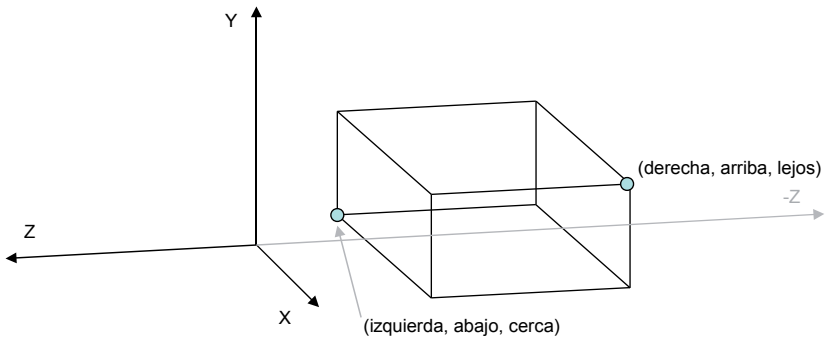


Figura 4.4: Esquema del volumen de la vista de una proyección paralela

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* (ver figura 4.5) y a las coordenadas en este volumen *coordenadas normalizadas del dispositivo*. La matriz de transformación correspondiente para un cubo de lado 2 es la siguiente:

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

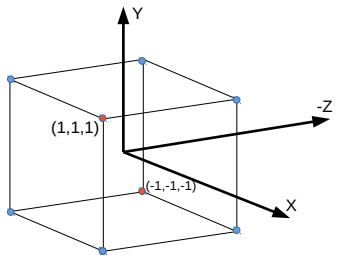


Figura 4.5: Volumen canónico de la vista, cubo de lado 2 centrado en el origen de coordenadas

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 truncada, 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 4.6 se detallan estos parámetros.

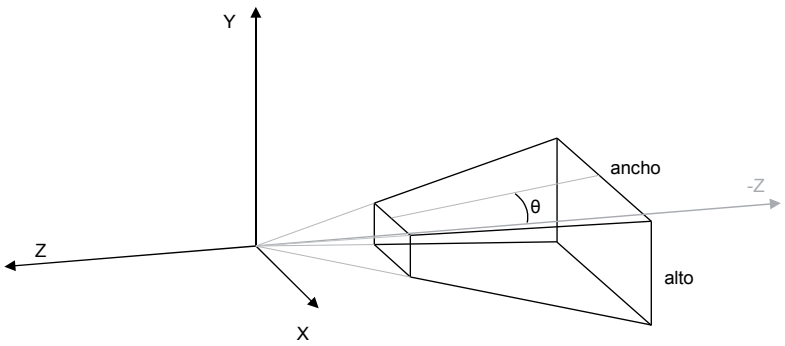


Figura 4.6: Esquema del volumen de la vista de una proyección perspectiva

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 2 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 área de dibujo en píxeles, y o_x y o_y son el píxel de la esquina inferior izquierda del área de dibujo 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:

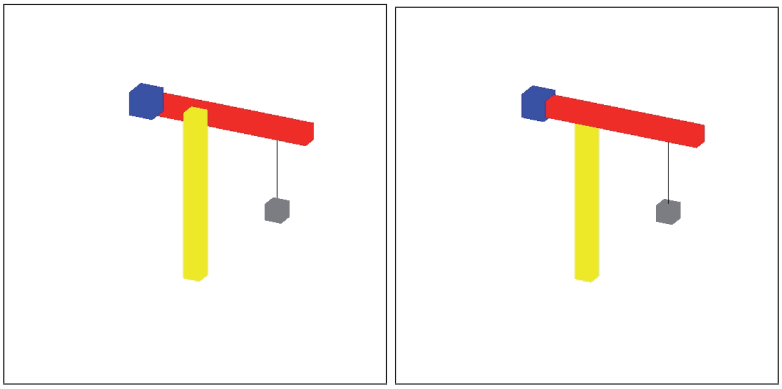
$$M_{vp} = \begin{pmatrix} \frac{n_x}{2} & 0 & 0 & \frac{n_x-1}{2} + o_x \\ 0 & \frac{n_y}{2} & 0 & \frac{n_y-1}{2} + o_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.4}$$

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 4.7). Aunque para resolver este problema se han desarrollado diversos algoritmos, el más utilizado en la práctica es el algoritmo conocido como *z-buffer*.

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.



(a) (b)

Figura 4.7: Ejemplo de escena visualizada: (a) sin resolver el problema de la visibilidad y (b) con el problema resuelto

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 3D con WebGL

Hasta ahora, en los ejercicios realizados en los temas anteriores, se ha conseguido visualizar modelos sin tener que realizar ni la transformación de la cámara ni establecer un tipo de proyección. Esto ocurre porque el modelo, o la escena, a visualizar se ha definido de manera que hábilmente quedase dentro del volumen canónico de la vista. De esta manera se obtiene una proyección paralela del contenido del volumen observando la escena en dirección del eje $-Z$.

En este capítulo se ha visto cómo construir la matriz de transformación de la cámara para poder observar la escena desde cualquier punto de vista, y la matriz de proyección para poder elegir entre vista paralela y vista perspectiva. En WebGL es responsabilidad del programador calcular estas matrices y operarlas con cada uno de los vértices del modelo.

Habitualmente, la matriz de la cámara se opera con la de transformación del modelo, creando la transformación conocida con el nombre de *modelo-vista*. Esta matriz y la de proyección se han de suministrar al procesador de vértices, donde cada vértice v debe ser multiplicado por ambas matrices. Si M_C es la matriz de transformación de la cámara, M_M es la matriz de transformación del modelo, y M_{MV} es la matriz modelo-vista, el nuevo vértice v' se obtiene como resultado de la siguiente operación: $v' = M_{Proy} \cdot M_{MV} \cdot v$; donde $M_{MV} = M_C \cdot M_M$; y M_{Proy} será M_{Par} o M_{Per}. El listado 4.2 recoge estas operaciones.

Listado 4.2: *Shader* de vértices para transformar la posición de cada vértice

```
uniform    mat4 projectionMatrix;   // perspectiva o paralela
uniform    mat4 modelViewMatrix;    // cameraMatrix * modelMatrix

attribute vec3 VertexPosition;

void main()  {

  gl_Position = projectionMatrix *
                modelViewMatrix *
                vec4(VertexPosition,1.0);

}
```

La librería GLMATRIX proporciona diversas funciones para construir las matrices vistas en este capítulo. Así, la función *mat4.lookAt* construye la matriz de transformación de la cámara, 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 *mat4.ortho* y *mat4.perspective* construyen las matrices que transforman el volumen de proyección paralela y perspectiva, respectivamente, al volumen canónico de la vista. Son estas:

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

Tras el procesamiento de los vértices, estos se reagrupan dependiendo del tipo de primitiva que se esté dibujando. Esto ocurre en la etapa procesado de la primitiva (ver figura 4.8). A continuación se realiza la operación conocida con el nombre de *división perspectiva*, que consiste en que cada vértice sea dividido por su propia w (la cuarta coordenada del vértice). Esto es necesario, ya que el resultado de la proyección perspectiva puede hacer que la coordenada w sea distinta de 1.

La transformación al área de dibujo se realiza a continuación, aún en la misma etapa. El programador sólo debe especificar la ubicación del *viewport* en el canvas mediante la orden *gl.viewport*, indicando la esquina inferior izquierda, el ancho y el alto en coordenadas de ventana:

- `gl.viewport (`x`, `y`, `$ancho$`, `$alto$`)`

La llamada a la función *gl.viewport* debe realizarse cada vez que se produzca un cambio en el tamaño del canvas con el fin de mantener el *viewport* actualizado. Además, es importante que la relación de aspecto del *viewport* sea igual a la

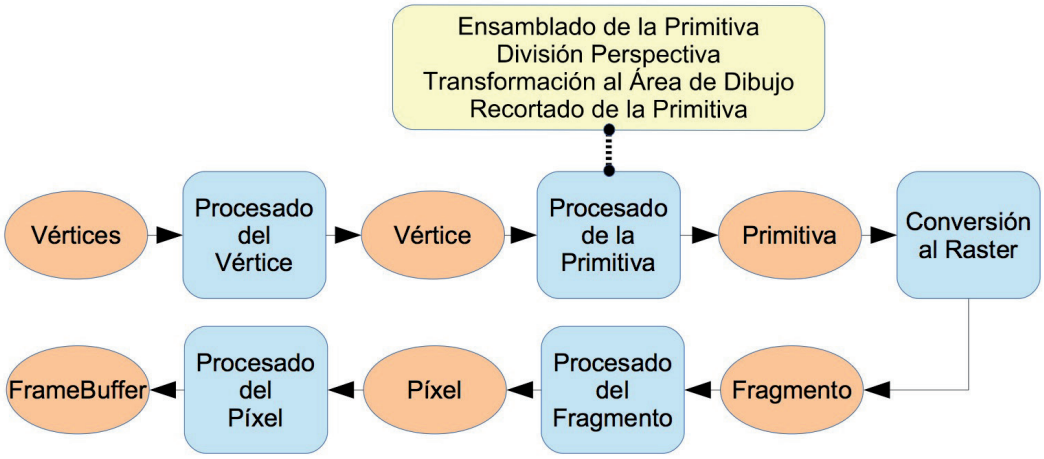


Figura 4.8: En color amarillo, las tareas principales que se realizan en la etapa correspondiente al procesado de la primitiva

relación de aspecto utilizada al definir el volumen de la vista para no deformar el resultado de la proyección.

La última tarea dentro de la etapa de procesado de la primitiva es la operación de recortado, que consiste en eliminar los elementos que quedan fuera de lps límites establecidos por el volumen de la vista. Si una primitiva intersecta con el volumen de la vista, se recorta de manera que por el *pipeline* únicamente continúan los trozos que han quedado dentro del volumen de la vista.

Respecto a la eliminación de partes ocultas, WebGL implementa el algoritmo del *z-buffer* y la GPU comprueba la profundidad de cada fragmento de forma fija en la etapa de procesado del fragmento, pero después de ejecutar el *shader* de fragmentos (ver figura 4.9). A esta comprobación se le denomina test de profundidad.

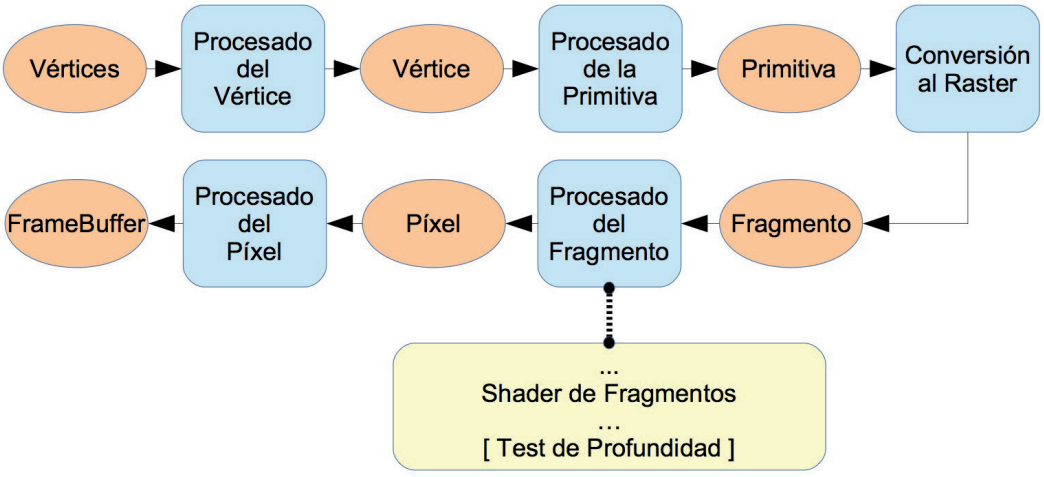


Figura 4.9: En color amarillo, las tareas principales que se realizan en la etapa correspondiente al procesado del fragmento donde se indica que el test de profundidad se realiza con posterioridad a la ejecución del *shader* de fragmentos

Sin embargo, el programador aún debe realizar dos tareas:

1. Habilitar la operación del test de profundidad:
   - `gl.enable (gl.DEPTH_TEST)`
2. Inicializar el *buffer* a la profundidad máxima antes de comenzar a dibujar:
   - `gl.clear (... | gl.DEPTH_BUFFER_BIT)`

Ejercicios

► **4.1** Ejecuta el programa *c04/mueveLaCamara.html*. Comprueba que se ha añadido una cámara interactiva que puedes modificar utilizando el ratón y las teclas *shift* y *alt*. Edita *c04/mueveLaCamara.js* y estudia la función *getCameraMatrix*, que devuelve la matriz de transformación de la cámara, ¿cuál es el punto de interés establecido? Estudia también la función *setProjection*, ¿qué tipo de proyección se utiliza?, ¿qué cambios harías para utilizar el otro tipo de proyección visto en este capítulo?

► **4.2** Modifica cualquiera de las soluciones realizadas 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 el modelo 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 el modelo completo.

► **4.4** Amplía la solución del ejercicio anterior para que se realice la eliminación de las partes ocultas. Para observarlo mejor, utiliza un color diferente para cada primitiva y dibújalas como triángulos, no como líneas.

Capítulo 5

Modelos de iluminación y sombreado

«In trying to improve the quality of the synthetic images, we do not expect to be able to display the object exactly as it would appear in reality, with texture, overcast shadows, etc. We hope only to display an image that approximates the real object closely enough to provide a certain degree of realism».

Bui Tuong Phong, 1975

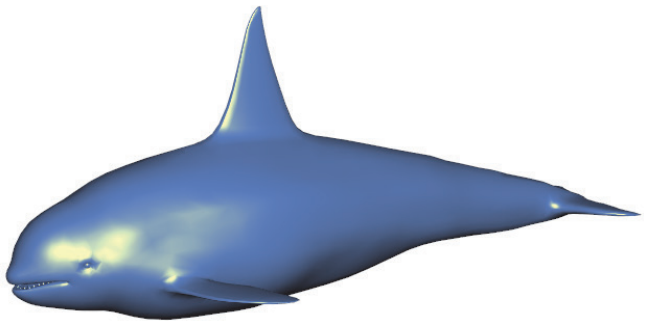


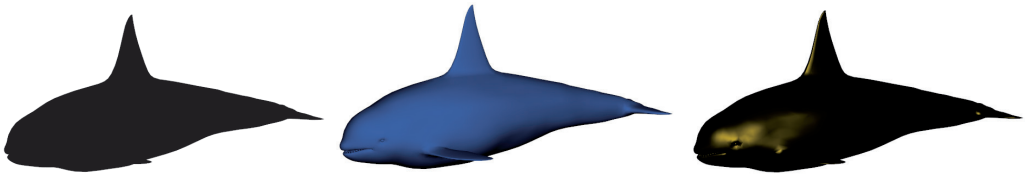
Figura 5.1: Ejemplo obtenido utilizando el modelo de iluminación de Phong

5.1. Modelo de iluminación de Phong

En esta sección se describe el modelo de iluminación de Phong (ver figura 5.1). Este modelo tiene en cuenta los tres aspectos siguientes:

- Luz ambiente: luz que proporciona iluminación uniforme a lo largo de la escena (ver figura 5.2(a)).
- Reflexión difusa: luz reflejada por la superficie en todas las direcciones (ver figura 5.2(b)).

- Reflexión especular: luz reflejada por la superficie en una sola dirección o en un rango de ángulos muy cercano al ángulo de reflexión perfecta (ver figura 5.2(c)).



(a) (b) (c)

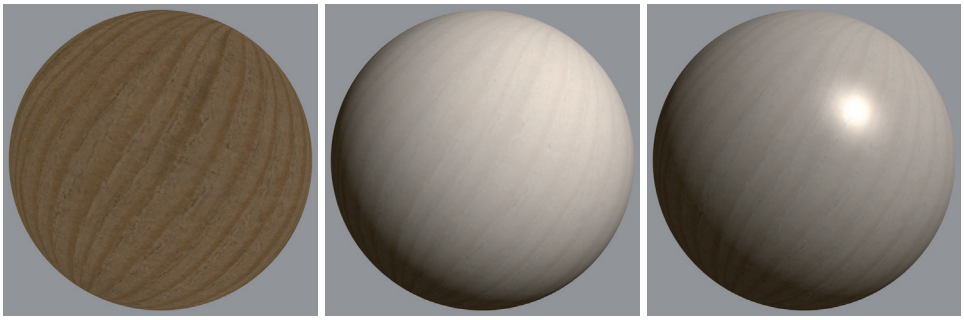
Figura 5.2: Ejemplo de las componentes del modelo de iluminación de Phong: (a) Luz ambiente; (b) Reflexión difusa; (c) Reflexión especular. La combinación de estos tres aspectos produce el resultado que se muestra en la figura 5.1

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



(a) (b) (c)

Figura 5.3: 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 solo 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 5.4). 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:

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

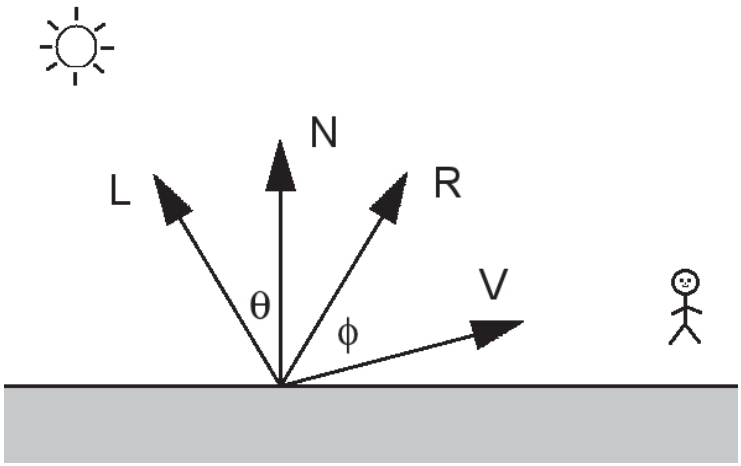


Figura 5.4: Geometría del modelo de iluminación de Phong

La figura 5.3(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 de dirección al observador V (ver figura 5.4). Si R y V son vectores unitarios, k_s, $0 \leq k_s \leq 1$

representa 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:

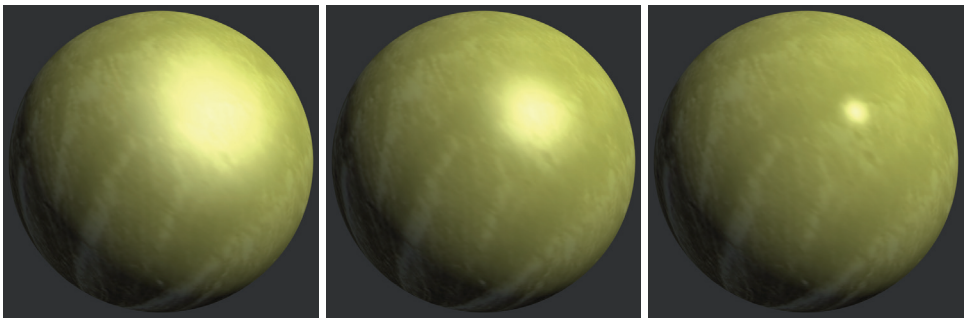
$$I_s = k_s L_s \cos^{\alpha} \Phi = k_s L_s (R \cdot V)^{\alpha} \tag{5.4}$$

donde R se obtiene de la siguiente manera:

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

La figura 5.3(c) muestra un ejemplo en el que el modelo de iluminación incluye luz ambiente, reflexión difusa y reflexión 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 5.5 muestra varios ejemplos obtenidos con distintos valores de α.



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

Figura 5.5: 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.

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:

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

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

Tabla 5.1: Ejemplos de propiedades de algunos materiales para el modelo de Phong

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:

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

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 ilumina una bombilla, por ejemplo.
- 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.

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

```
struct LightData {
    vec3 Position; // Posición en coordenadas del ojo
    vec3 La;       // Ambiente
    vec3 Ld;       // Difusa
    vec3 Ls;       // Especular
};
uniform LightData Light;

struct MaterialData {
    vec3 Ka;       // Ambiente
    vec3 Kd;       // Difusa
    vec3 Ks;       // Especular
    float alpha;   // Brillo especular
};
uniform MaterialData Material;

// N, L y V se asumen normalizados
vec3 phong (vec3 N, vec3 L, vec3 V) {

    vec3  ambient  = Material.Ka * Light.La;
    vec3  diffuse  = vec3(0.0);
    vec3  specular = vec3(0.0);

    float NdotL    = dot (N,L);

    if (NdotL > 0.0) {
      vec3  R       = reflect(-L, N);
      float RdotV_n = pow(max(0.0, dot(R,V)), Material.alpha);

      diffuse  = NdotL   * (Light.Ld * Material.Kd);
      specular = RdotV_n * (Light.Ls * Material.Ks);
    }

    return (ambient + diffuse + specular);
}
```

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 5.6). A diferencia de una luz posicional, 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 5.7.

Así, un fragmento es iluminado por el foco solo 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 consecuencia, afectado únicamente por la luz ambiente.

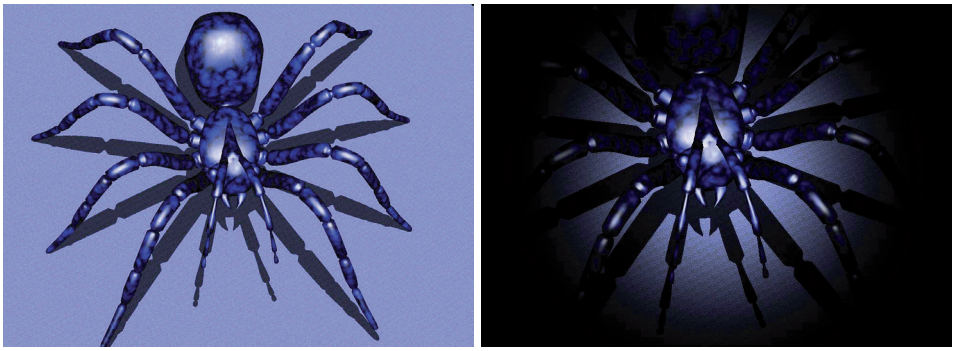


Figura 5.6: Ejemplo de escena iluminada: a la izquierda, con una luz posicional y a la derecha, con la fuente convertida en foco

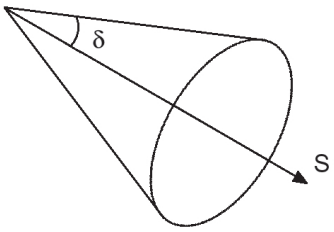


Figura 5.7: Parámetros característicos de un foco de luz

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 (ver figura 5.6, imagen de la derecha). Finalmente, el factor de atenuación calculado se incorpora al modelo de iluminación de Phong, multiplicando el factor de atenuación que ya existía. El listado 5.2 muestra el nuevo *shader* que implementa el foco de luz (la estructura *LightData* muestra solo los campos nuevos).

5.3. Modelos de 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. Dados un polígono y 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 5.8(a). El listado 5.3 muestra el *shader* correspondiente a este modelo de sombreado.

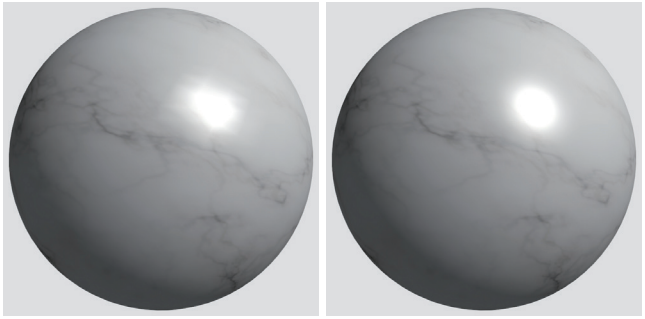
- Phong: el modelo de iluminación se aplica para cada fragmento. Este método requiere la normal en el fragmento, que se puede obtener por interpolación de las normales de los vértices. Un ejemplo del resultado obtenido se muestra en la figura 5.8(b). El listado 5.4 muestra el *shader* correspondiente a este modelo de sombreado.

Listado 5.2: *Shader* para el foco de luz

```
struct LightData {     // Solo figuran los campos nuevos
    ....
    vec3  Direction; // Dirección de la luz en coordenadas del ojo
    float Exponent;  // Atenuación
    float Cutoff;    // Angulo de corte en grados
}
uniform LightData Light;

vec3 phong (vec3 N, vec3 L, vec3 V) {

    vec3  ambient  = Material.Ka * Light.La;
    vec3  diffuse  = vec3(0.0);
    vec3  specular = vec3(0.0);

    float NdotL      = dot (N,L);
    float spotFactor = 1.0;

    if (NdotL > 0.0) {

      vec3  s      = normalize (Light.Position - ec);
      float angle  = acos(dot(-s, Light.Direction));
      float cutoff = radians(clamp(Light.Cutoff, 0.0, 90.0));

      if (angle < cutoff) {

        spotFactor= pow (dot(-s,Light.Direction),Light.Exponent);

        vec3  R       = reflect(-L, N);
        float RdotV_n = pow(max(0.0, dot(R,V)), Material.alpha);

        diffuse  = NdotL   * (Light.Ld * Material.Kd);
        specular = RdotV_n * (Light.Ls * Material.Ks);
      }
    }

    return (ambient + spotFactor * (diffuse + specular));

}
```

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

```
// Shader de vértices ------------------------------------------------------
uniform    mat4 projectionMatrix;
uniform    mat4 modelViewMatrix;
uniform    mat3 normalMatrix;

attribute  vec3 VertexPosition;
attribute  vec3 VertexNormal;

varying    vec3 colorOut;

// ... aquí va el código del listado 5.1

void main()  {

    vec3 N          = normalize(normalMatrix * VertexNormal);
    vec4 ecPosition= modelViewMatrix * vec4(VertexPosition,1.0);
    vec3 ec         = vec3(ecPosition);
    vec3 L          = normalize(Light.Position - ec);
    vec3 V          = normalize(-ec);

    colorOut        = phong(N,L,V);

    gl_Position     = projectionMatrix * ecPosition;

}

// Shader de fragmentos ----------------------------------------------------
precision mediump float;

varying vec3 colorOut;

void main() {

    gl_FragColor = vec4(colorOut,1);

}
```



(a) Gouraud (b) Phong

Figura 5.8: Ejemplos de modelos de sombreado: (a) Gouraud; (b) Phong

Listado 5.4: *Shader* para realizar un sombreado de Phong

```
// Shader de vértices ------------------------------------------------------
uniform   mat4 projectionMatrix;
uniform   mat4 modelViewMatrix;
uniform   mat3 normalMatrix;

attribute vec3 VertexPosition;
attribute vec3 VertexNormal;

varying vec3 N, ec;

void main()  {

    N  = normalize(normalMatrix * VertexNormal);
    vec4 ecPosition= modelViewMatrix * vec4(VertexPosition,1.0);
    ec = vec3(ecPosition);

    gl_Position    = projectionMatrix * ecPosition;

}

// Shader de fragmentos ----------------------------------------------------
precision mediump float;

// ... aquí va el código del listado 5.1

varying vec3 N, ec;

void main() {

    vec3 n = normalize(N);
    vec3 L = normalize(Light.Position - ec);
    vec3 V = normalize(-ec);

    gl_FragColor = vec4(phong(n,L,V),1.0);

}
```

5.4. Implementa Phong con WebGL

5.4.1. Normales en los vértices

Hasta ahora las primitivas geométricas contenían únicamente el atributo de posición. Para poder aplicar el modelo de iluminación de Phong, es necesario que además se proporcionen las normales para cada vértice. Por ejemplo, el listado 5.5 define el modelo de un cubo donde para cada vértice se proporcionan la posición y la normal. Observa que en el modelo del cubo, la normal de un vértice es diferente según la cara que lo utilice. Por este motivo, la descripción del cubo ha pasado de 8 vértices a 24.

Listado 5.5: Modelo de un cubo con la normal definida para cada vértice

```
var exampleCube = {

    "vertices" : [-0.5,-0.5, 0.5, 0.0, 0.0, 1.0,
                   0.5,-0.5, 0.5, 0.0, 0.0, 1.0,
                   0.5, 0.5, 0.5, 0.0, 0.0, 1.0,
                  -0.5, 0.5, 0.5, 0.0, 0.0, 1.0,
                   0.5,-0.5, 0.5, 1.0, 0.0, 0.0,
                   0.5,-0.5,-0.5, 1.0, 0.0, 0.0,
                   0.5, 0.5,-0.5, 1.0, 0.0, 0.0,
                   0.5, 0.5, 0.5, 1.0, 0.0, 0.0,
                   0.5,-0.5,-0.5, 0.0, 0.0,-1.0,
                  -0.5,-0.5,-0.5, 0.0, 0.0,-1.0,
                  -0.5, 0.5,-0.5, 0.0, 0.0,-1.0,
                   0.5, 0.5,-0.5, 0.0, 0.0,-1.0,
                  -0.5,-0.5,-0.5,-1.0, 0.0, 0.0,
                  -0.5,-0.5, 0.5,-1.0, 0.0, 0.0,
                  -0.5, 0.5, 0.5,-1.0, 0.0, 0.0,
                  -0.5, 0.5,-0.5,-1.0, 0.0, 0.0,
                  -0.5, 0.5, 0.5, 0.0, 1.0, 0.0,
                   0.5, 0.5, 0.5, 0.0, 1.0, 0.0,
                   0.5, 0.5,-0.5, 0.0, 1.0, 0.0,
                  -0.5, 0.5,-0.5, 0.0, 1.0, 0.0,
                  -0.5,-0.5,-0.5, 0.0,-1.0, 0.0,
                   0.5,-0.5,-0.5, 0.0,-1.0, 0.0,
                   0.5,-0.5, 0.5, 0.0,-1.0, 0.0,
                  -0.5,-0.5, 0.5, 0.0,-1.0, 0.0],

    "indices" :  [ 0, 1, 2, 0, 2, 3, 4, 5, 6, 4, 6, 7,
                   8, 9,10, 8,10,11,12,13,14,12,14,15,
                  16,17,18,16,18,19,20,21,22,20,22,23]
};
```

La función *initShaders* es un buen sitio donde obtener la referencia al nuevo atributo y habilitarlo, así como para obtener la referencia a la matriz de transformación de la normal (ver listado 5.6).

Listado 5.6: Obtención de referencias para el uso de las normales

```
program.vertexNormalAttribute =
    gl.getAttribLocation ( program, "VertexNormal");
program.normalMatrixIndex      =
    gl.getUniformLocation( program, "normalMatrix");
gl.enableVertexAttribArray(program.vertexNormalAttribute);
```

Por supuesto, la matriz de la normal es propia de cada modelo, ya que se calcula a partir de la matriz modelo-vista y se obtiene, como se explicó en el capítulo 3,

a partir de la traspuesta de su inversa. Como esta operación hay que realizarla para cada modelo, es conveniente crear una función específica y llamarla antes de ordenar su dibujado para obtener así la matriz de la normal del *shader*, tal y como se muestra en el listado 5.7.

Listado 5.7: Funciones para el cálculo y la inicialización de la matriz de la normal en el *shader* a partir de la matriz modelo-vista

```
function getNormalMatrix(modelViewMatrix) {

  var normalMatrix = mat3.create();

  mat3.fromMat4   (normalMatrix, modelViewMatrix);
  mat3.invert     (normalMatrix, normalMatrix);
  mat3.transpose  (normalMatrix, normalMatrix);

  return normalMatrix;

}

function setShaderNormalMatrix(normalMatrix) {

  gl.uniformMatrix3fv(program.normalMatrixIndex, false,
      normalMatrix);

}
```

Por último, a la hora de dibujar el modelo, hay que especificar cómo se encuentra almacenado dicho atributo en el vector de vértices (ver listado 5.8).

Listado 5.8: Nueva función de dibujo que incluye dos atributos: posición y normal

```
function drawSolid(model) {

  gl.bindBuffer (gl.ARRAY_BUFFER, model.idBufferVertices);
  gl.vertexAttribPointer (program.vertexPositionAttribute, 3,
                          gl.FLOAT, false, 2*3*4,   0);
  gl.vertexAttribPointer (program.vertexNormalAttribute,   3,
                          gl.FLOAT, false, 2*3*4, 3*4);

  gl.bindBuffer    (gl.ELEMENT_ARRAY_BUFFER,
                    model.idBufferIndices);
  gl.drawElements (gl.TRIANGLES, model.indices.length,
                    gl.UNSIGNED_SHORT, 0);

}
```

5.4.2. Materiales

Para especificar un material es necesario, en primer lugar, obtener las referencias de las variables que van a contener el valor del material en el *shader*. Examina el listado 5.1 para recordarlas. El listado 5.9 muestra el código correspondiente a este paso. Un buen sitio para colocarlo sería en la función *initShaders*. En segundo lugar, hay que especificar el material para cada modelo justo antes de ordenar su dibujado. Para esto, es buena idea definir una función que se encargue de inicializar las variables del *shader* correspondientes al material. El listado 5.10 muestra ambas acciones.

Listado 5.9: Obtención de las referencias a las variables del *shader* que contendrán el material

```
program.KaIndex    = gl.getUniformLocation(program, "Material.Ka");
program.KdIndex    = gl.getUniformLocation(program, "Material.Kd");
program.KsIndex    = gl.getUniformLocation(program, "Material.Ks");
program.alphaIndex= gl.getUniformLocation(program,
                     "Material.alpha");
```

Listado 5.10: La función *setShaderMaterial* recibe un material como parámetro e inicializa las variables del *shader* correspondientes. En la función *drawScene* se establece un valor de material antes de dibujar el objeto

```
var Silver = {
  "mat_ambient" : [ 0.19225, 0.19225, 0.19225 ],
  "mat_diffuse" : [ 0.50754, 0.50754, 0.50754 ],
  "mat_specular": [ 0.508273, 0.508273, 0.508273 ],
  "alpha"       : [ 51.2 ]
};

function setShaderMaterial(material) {

  gl.uniform3fv(program.KaIndex,     material.mat_ambient);
  gl.uniform3fv(program.KdIndex,     material.mat_diffuse);
  gl.uniform3fv(program.KsIndex,     material.mat_specular);
  gl.uniform1f (program.alphaIndex, material.alpha);
}

function drawScene () {

  ....
  // establece un material y dibuja el cubo
  setShaderMaterial(Silver);
  drawSolid(exampleCube);
  ....
}
```

5.4.3. Fuente de luz

Respecto a la fuente de luz, por una parte hay que obtener las referencias a las variables del *shader* (ver el listado 5.11) que definen los parámetros de la fuente de luz, y que de manera similar al material podemos colocar en la función *initShaders*. Por otra parte, hay que inicializar las variables del *shader* antes de ordenar el dibujado del primer objeto de la escena. Es interesante agrupar la inicialización en una sola función (ver listado 5.12).

Listado 5.11: Obtención de las referencias a las variables del *shader* que contendrán los valores de la fuente de luz

```
program.LaIndex       = gl.getUniformLocation(program, "Light.La");
program.LdIndex       = gl.getUniformLocation(program, "Light.Ld");
program.LsIndex       = gl.getUniformLocation(program, "Light.Ls");
program.PositionIndex= gl.getUniformLocation(program,
                       "Light.Position");
```

Listado 5.12: La función *setShaderLight* inicializa las variables del *shader* correspondientes

```
function setShaderLight() {

  gl.uniform3f(program.LaIndex,         1.0,1.0,1.0);
  gl.uniform3f(program.LdIndex,         1.0,1.0,1.0);
  gl.uniform3f(program.LsIndex,         1.0,1.0,1.0);
  gl.uniform3f(program.PositionIndex, 10.0,10.0,0.0);

}
```

Ejercicios

► **5.1** Ejecuta el programa *c05/phongConSombreadoGouraud.html*, que implementa el modelo de iluminación de Phong y el modelo de sombreado de Gouraud. Como resultado obtendrás imágenes similares a las de la figura 5.9. Examina el *shader* en el HTML y comprueba que los fragmentos de código comentados en esta sección se incluyen en *c05/iluminacion.js*. Comprueba también cómo el fichero *primitivas.js* contiene las normales de cada vértice para cada una de las primitivas definidas.

► **5.2** A partir del ejemplo anterior realiza los cambios necesarios para implementar el modelo de sombreado de Phong. La figura 5.10 muestra dos ejemplos de resultados obtenidos utilizando este modelo de sombreado.

Figura 5.9: Ejemplos obtenidos con el modelo de iluminación de Phong y el modelo de sombreado de Gouraud

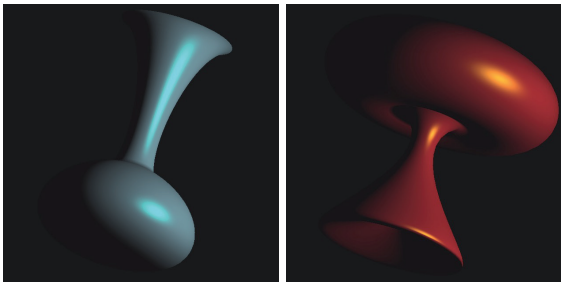


Figura 5.10: Ejemplos obtenidos con el modelo de sombreado de Phong

5.5. Iluminación por ambas caras

Cuando la escena incorpora modelos abiertos, es posible observar los polígonos que se encuentran en la parte trasera de los objetos, como por ejemplo ocurre en la copa de la figura 5.11. Para una correcta visualización es necesario utilizar la normal contraria en los polígonos que forman parte de la cara trasera. Esto es una operación muy sencilla en WebGL y se muestra en el listado 5.13.

Listado 5.13: Iluminación en ambas caras, modificación en el *shader* de fragmentos

```
if (gl_FrontFacing)
    gl_FragColor = vec4(phong(n,L,V),1.0);
else
    gl_FragColor = vec4(phong(-n,L,V),1.0);
```

5.6. Sombreado cómic

El objetivo es simular el sombreado típico en cómics. Este efecto se consigue haciendo que la componente difusa del color de un fragmento se restrinja a solo un número determinado de posibles valores. La función *toonShading* que se muestra en el listado 5.14 realiza esta operación para cada fragmento. La variable *levels* determina el número máximo de colores distintos (ver figura 5.12).

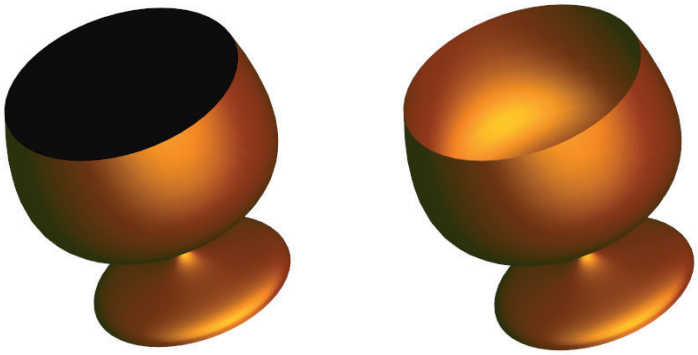


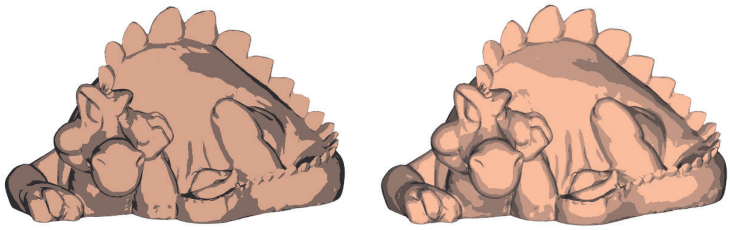
Figura 5.11: Ejemplo de modelo en el que hay que aplicar iluminación en ambas caras para una correcta visualización

Listado 5.14: Iluminación en ambas caras, modificación en el *shader* de fragmentos

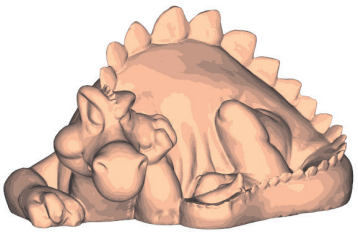
```
vec3 toonShading (vec3 N, vec3 L) {

    vec3  ambient     = Material.Ka * Light.La;
    float NdotL       = max(0.0, dot (N,L));
    float levels      = 3.0;
    float scaleFactor = 1.0 / levels;

    vec3 diffuse  = ceil(NdotL * levels) * scaleFactor *
                    (Light.Ld * Material.Kd);

    return (ambient + diffuse);

  }

void main() {

    vec3 n = normalize(N);
    vec3 L = normalize(Light.Position - ec);

    gl_FragColor = vec4(toonShading(n,L),1.0);

}
```

Ejercicios

► **5.3** Añade la función *toonShading* a tu *shader* de fragmentos y haz que se llame a esta en lugar de a la función *phong*. Observa que la componente especular se ha eliminado de la ecuación, por lo que la función *toonShading* solo necesita los vectores N y L.



(a) $levels = 3$ (b) $levels = 5$



(c) $levels = 10$

Figura 5.12: Resultado de la función *toonShading* con diferentes valores de la variable *levels*

5.7. Niebla

El efecto de niebla se consigue mezclando el color de cada fragmento con el color de la niebla. A mayor distancia de un fragmento respecto a la cámara, mayor peso tiene el color de la niebla y al contrario, a menor distancia, mayor peso tiene el color del fragmento.

En primer lugar hay que obtener el color del fragmento y después, a modo de post-proceso, se mezcla con el color de la niebla dependiendo de la distancia a la cámara. Para implementar este efecto se definen tres variables: distancia mínima, distancia máxima y color de la niebla. Así, dado un fragmento, tenemos tres posibilidades:

- Si el fragmento está a una distancia menor que la distancia mínima, no se ve afectado por la niebla, el color definitivo es el color del fragmento.
- Si el fragmento está a una distancia mayor que la máxima, el color definitivo es el color de la niebla (es importante que el color de fondo coincida con el de la niebla).
- Si el fragmento está a una distancia entre la mínima y la máxima, su color definitivo depende del color del fragmento y del de la niebla. Esta variación puede ser lineal con la distancia, pero suele producir mucho mejor resultado utilizar una función exponencial.

El listado 5.15 muestra la estructura *FogData* y el cálculo del valor de niebla de manera lineal y, con comentarios, de manera exponencial. La figura 5.13 muestra algunos resultados.

Listado 5.15: *Shader* de niebla

```
struct FogData {
    float maxDist;
    float minDist;
    vec3  color;
};
FogData Fog;

void main() {

    Fog.minDist = 10.0;
    Fog.maxDist = 20.0;
    Fog.color   = vec3(0.15, 0.15, 0.15);

    vec3 n = normalize(N);
    vec3 L = normalize(Light.Position - ec);
    vec3 V = normalize(-ec);

    float dist      = abs (ec.z);

    // lineal
    loat fogFactor = (Fog.maxDist - dist) /
                     (Fog. maxDist - Fog.minDist);

    // exponencial
    // float fogFactor = exp(-pow(dist,2.0));

    fogFactor       = clamp (fogFactor, 0.0, 1.0);
    vec3 phongColor = phong(n,L,V);
    vec3 myColor    = mix (Fog.color, phongColor, fogFactor);

    gl_FragColor    = vec4(myColor,1.0);

}
```

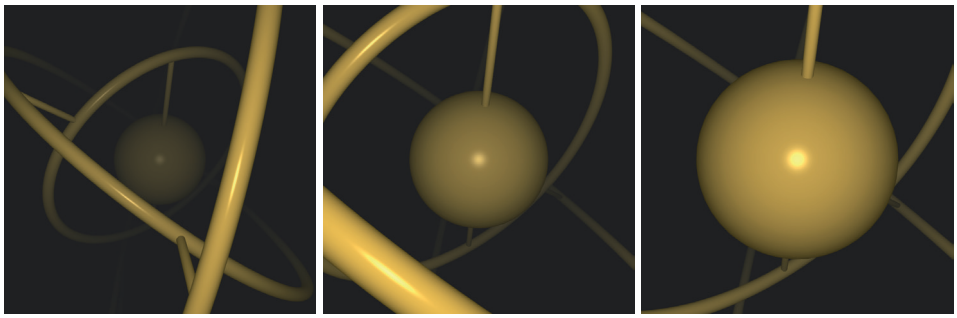


Figura 5.13: Resultados obtenidos utilizando niebla en el *shader*

Capítulo 6

Texturas

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 (ver figura 6.1). 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 van a revisar técnicas que resultan especialmente idóneas para gráficos en tiempo real.

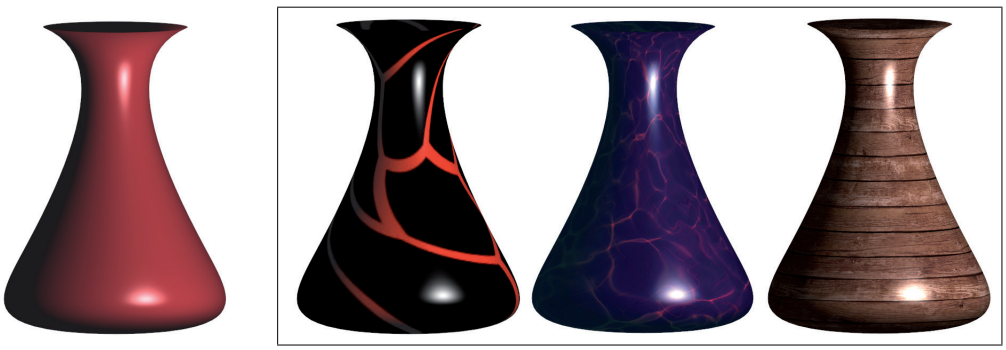


Figura 6.1: Resultados obtenidos al aplicar diferentes texturas 2D sobre el mismo objeto 3D

6.1. Coordenadas de textura

Las coordenadas de textura son un atributo más de los vértices, como lo es también la normal. El rango de coordenadas válido en el espacio de la textura es $[0..1]$, y es independiente del tamaño en píxeles de la textura (ver figura 6.2). La aplicación ha de suministrar estas coordenadas para cada vértice y la GPU las interpolará para cada fragmento (ver figura 6.3).

El listado 6.1 muestra el nuevo atributo y cómo se asignan las coordenadas de textura a una variable de tipo *varying* para que cada fragmento reciba sus coordenadas de textura interpoladas. En dicho listado se utiliza la función *texture2D* para, a partir de las coordenadas y una textura, obtener un valor de color. La textura está declarada de tipo *sampler2D*, que es el tipo que GLSL establece para una textura 2D. La figura 6.4 muestra dos resultados de la aplicación de textura.

Por supuesto, desde el *shader* de fragmentos es posible acceder a diferentes texturas y realizar cualquier combinación con los valores obtenidos para determinar el color definitivo. La figura 6.5 muestra un ejemplo donde se han combinado dos texturas y el listado 6.2 muestra los cambios necesarios en el *shader* de fragmentos para acceder a varias texturas (el *shader* de vértices no cambia). Observa cómo se utilizan las mismas coordenadas de textura para acceder a las diferentes texturas. En el ejemplo se ha utilizado la operación de multiplicación para combinar los colores obtenidos de cada textura, pero evidentemente se podría utilizar cualquier otra operación.

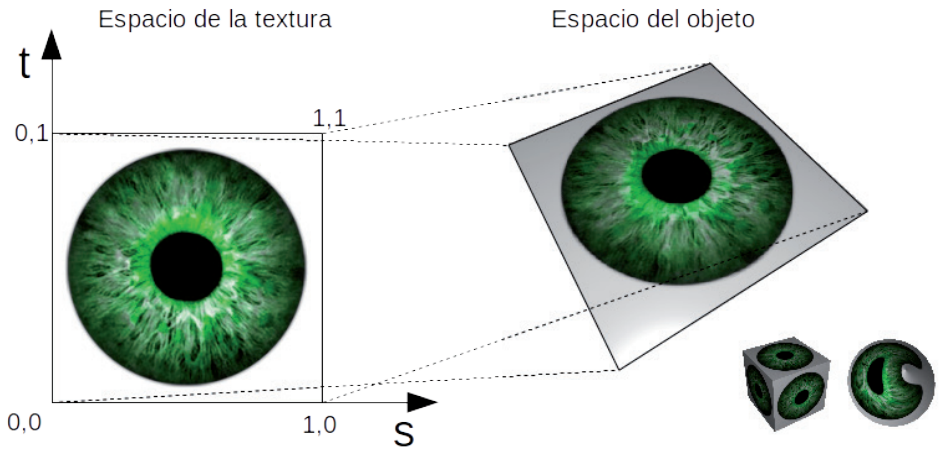


Figura 6.2: Correspondencia entre coordenadas de textura y coordenadas de un objeto

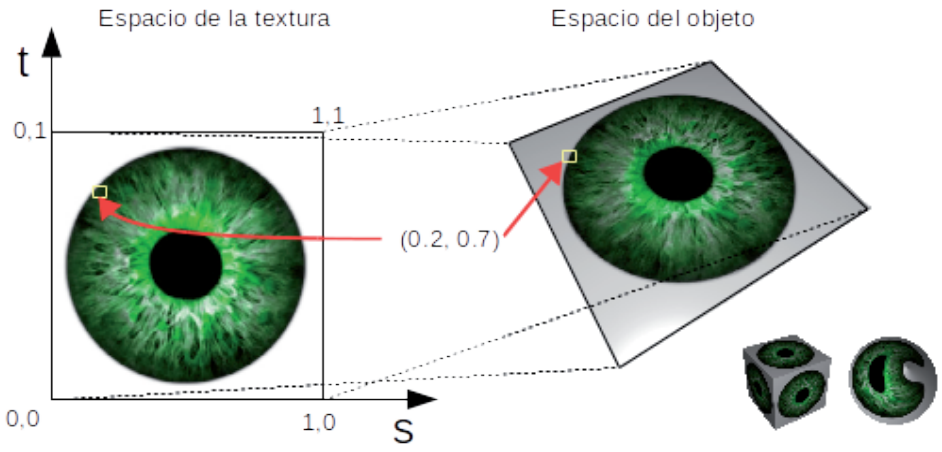


Figura 6.3: En el espacio del objeto, cada fragmento recibe las coordenadas de textura interpoladas

Listado 6.1: Cambios necesarios para que un *shader* utilice una textura 2D

```
// Shader de vértices
...
attribute vec2 VertexTexcoords;  // nuevo atributo
varying   vec2 texCoords;

void main()  {
  ...
  // se asignan las coordenadas de textura del vértice
  // a la variable texCoords
  texCoords = VertexTexcoords;

}

// Shader de fragmentos
...
uniform sampler2D myTexture;     // la textura
varying vec2      texCoords;     // coords de textura interpoladas

void main() {
  ...
  // acceso a la textura para obtener un valor de color RGBA
  gl_FragColor = texture2D(myTexture, texCoords);

}
```

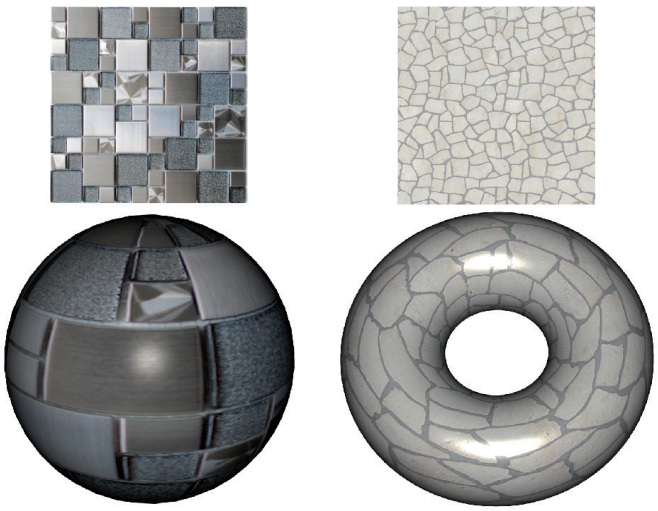


Figura 6.4: Ejemplos de aplicación de textura 2D. En estos casos el color definitivo de un fragmento se ha obtenido a partir de la textura y del modelo de iluminación de Phong

Listado 6.2: Cambios en el *shader* de fragmentos para utilizar varias texturas 2D

```
// Shader de fragmentos
...
uniform sampler2D myTexture1, myTexture2, ...; // las texturas
varying vec2      texCoords;    // coords de textura interpoladas

void main() {
  ...
  // acceso a las texturas para obtener los valores de color RGBA
  gl_FragColor = texture2D(myTexture1, texCoords) *
                 texture2D(myTexture2, texCoords) * ...;

}
```

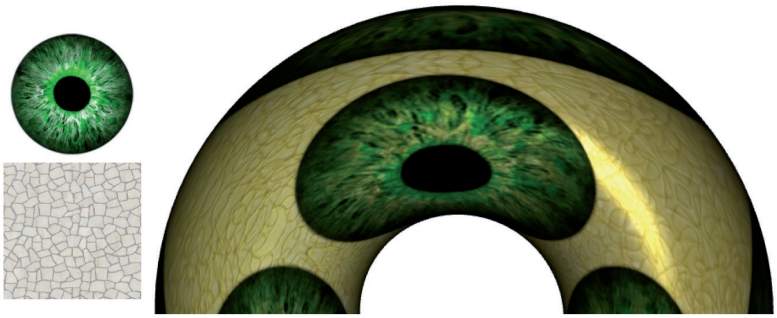


Figura 6.5: Resultado de combinar dos texturas

Por último, en el caso de proporcionar valores mayores que 1 en las coordenadas de textura, es posible gobernar el resultado de la visualización utilizando la función *gl.texParameter*. Tres son los modos posibles, y pueden combinarse con valores distintos en cada dirección S y T. Estos son los casos posibles:

- Repite la textura (ver figura 6.6)
  - `gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_[S|T], gl.REPEAT);`
- Extiende el borde de la textura (ver figura 6.7)
  - `gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_[S|T], gl.CLAMP_TO_EDGE);`
- Repite la textura de manera simétrica (ver figura 6.8)
  - `gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_[S|T], gl.MIRRORED_REPEAT);`

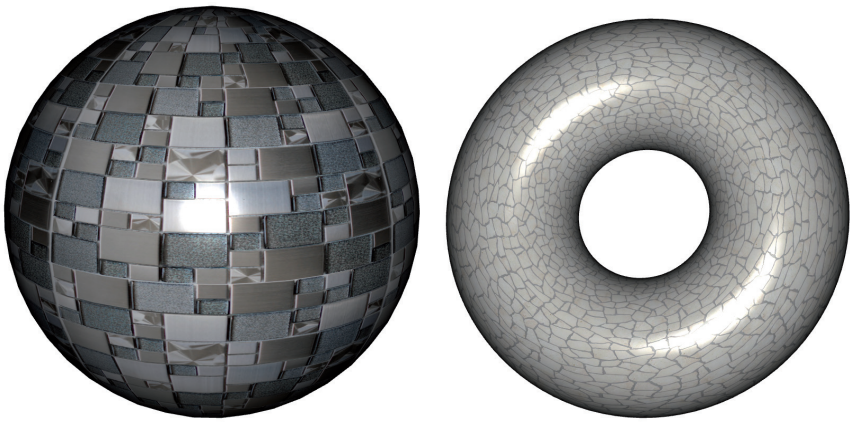


Figura 6.6: Ejemplos de repetición de textura

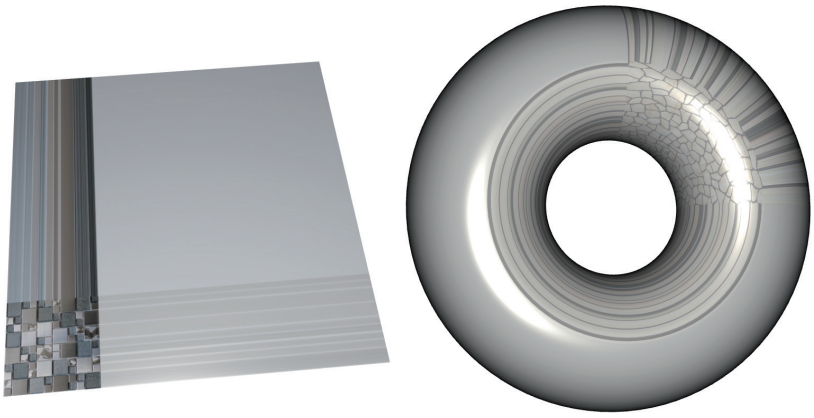


Figura 6.7: Ejemplos de extensión del borde de la textura

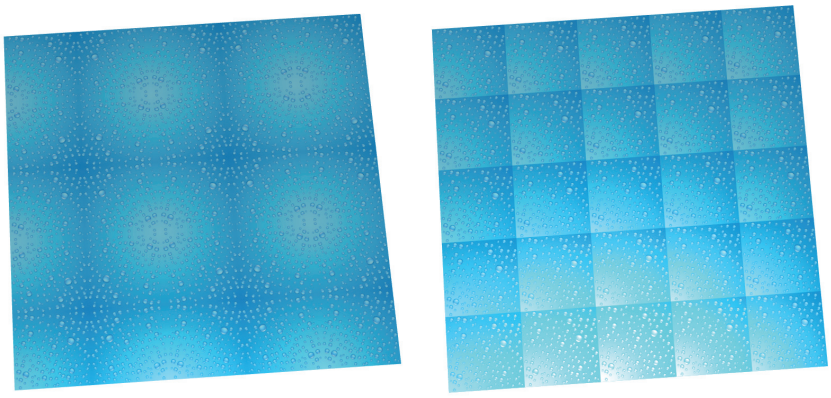


Figura 6.8: Comparación entre la repetición de la textura de manera simétrica (imagen de la izquierda) y repetición normal como la de la figura 6.6 (imagen de la derecha)

6.2. Leyendo téxeles

Un téxel es un píxel de la textura. En los orígenes, el acceso a los téxeles de una textura solo era posible realizarlo desde el *shader* de fragmentos. Sin embargo, los procesadores gráficos que aparecían a partir del 2008 eliminaban esa limitación y era posible acceder a una textura también desde el *shader* de vértices. En cualquier caso, el acceso a la textura conlleva dos problemas, debido a que rara vez el tamaño de un fragmento se corresponde con el de un téxel de la textura:

- Magnificación: cuando un fragmento se corresponde con un trocito de un téxel de la textura.
- Minimización: cuando un fragmento se corresponde con varios téxeles de la textura.

6.2.1. Magnificación

Para tratar este problema, WebGL proporciona dos tipos de filtrado:

- Filtro caja: se utiliza el téxel más cercano. Por desgracia, aunque es muy rápido, produce el típico efecto de pixelado que obtenemos al ampliar una imagen (ver figura 6.9).
  - `gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);`
- Filtro bilineal: utiliza cuatro téxeles cuyos valores se interpolan linealmente. Este filtro produce un efecto de borrosidad (ver figura 6.10).
  - `gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);`

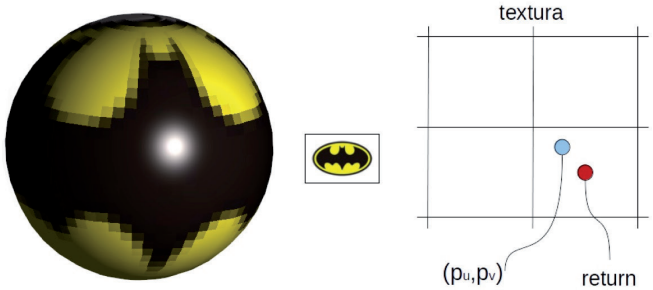


Figura 6.9: Filtro caja de WebGL, devuelve el valor del téxel más cercano y produce el efecto de pixelado

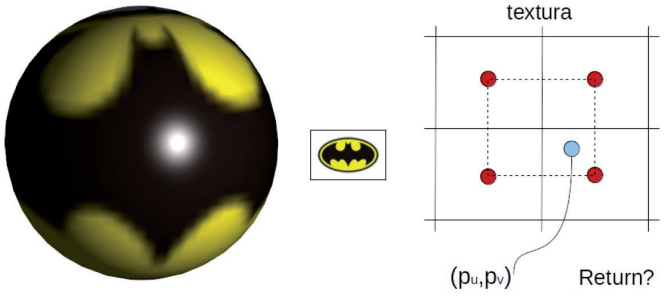


Figura 6.10: Filtro bilineal de WebGL, devuelve la interpolación lineal de cuatro téxeles y produce el efecto de borrosidad

6.2.2. Minimización

WebGL proporciona un método más de filtrado para el problema de la minimización, además del filtro caja y del bilineal comentados en la sección anterior. Se denomina *mipmapping* y consiste en proporcionar, además de la textura original, un conjunto de versiones más pequeñas de la textura, cada una de ellas un cuarto más pequeña que la anterior (ver figura 6.11).

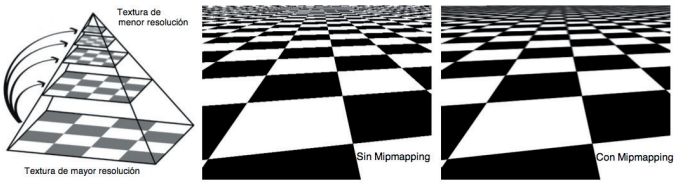


Figura 6.11: *Mipmapping* de WebGL, se construye un conjunto de texturas, cada una de ellas un cuarto más pequeña que la anterior. Observa la diferencia entre aplicar o no este tipo de filtro

- Filtro caja y bilineal, respectivamente:
  - `gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);`
  - `gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);`
- Mipmapping: en tiempo de ejecución, la GPU selecciona la textura cuyo tamaño se acerca más al de la textura en la pantalla. WebGL puede generar la pirámide de texturas de manera automática:
  - `gl.generateMipmap(gl.TEXTURE_2D);`

  Utilizando este método de filtrado, WebGL puede realizar un filtrado trilineal seleccionando dos texturas, muestreando cada una utilizando un filtro bilineal, e interpolando los dos valores obtenidos:
  - `gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR_MIPMAP_LINEAR);`

6.2.3. Texturas 3D

Una textura 3D define un valor para cada punto del espacio. Este tipo de texturas resulta perfecta para objetos que son creados a partir de un medio sólido como una talla de madera o un bloque de piedra. Son una extensión directa de las texturas 2D en las que ahora se utilizan tres coordenadas (s, t, r), y donde en lugar de téxeles tenemos vóxeles.

La principal ventaja de este tipo de texturas es que el propio atributo de posición del vértice se puede utilizar como coordenada de textura. Sin embargo, son caras de almacenar y también de filtrar. También son ineficientes cuando se utilizan con modelos de superficies, ya que la mayor parte de la información que la textura almacena nunca se utiliza. Por último, aunque OpenGL soporta este tipo de texturas, WebGL 1.0 no lo hace, aunque es cierto que una textura 3D siempre podría implementarse como un vector de texturas 2D.

6.2.4. Mapas de cubo

Un mapa de cubo son seis texturas cuadradas donde cada una de ellas se asocia a una cara del cubo distinta y que juntas capturan un determinado entorno (ver figura 6.12). Los mapas de cubo se utilizan principalmente para tres aplicaciones:

- *Reflection mapping*
- *Refraction mapping*
- *Skybox*

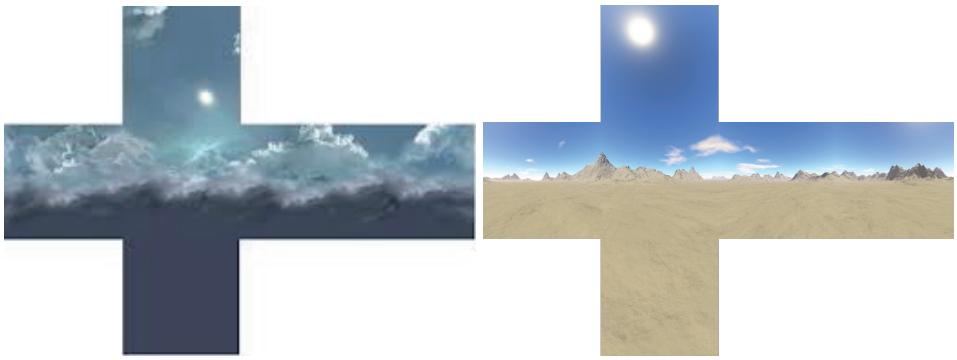


Figura 6.12: Ejemplos de texturas de mapa de cubo

Reflection mapping

Esta técnica de aplicación de textura sirve para simular objetos que reflejan su entorno. En la literatura también aparece con el nombre de *environment mapping*. 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, a la cual se accede utilizando coordenadas esféricas, o por un conjunto de seis texturas cuadradas formando un mapa de cubo. Este último es el que se describe en esta sección.

El mapa de cubo se dispone para formar un cubo de lado 2 centrado en el origen de coordenadas. Para cada punto de la superficie del objeto reflejante se obtiene el vector de reflexión R respecto a la normal N en ese punto de la superficie (ver figura 6.13). 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 utilizan la cara derecha o izquierda del cubo, dependiendo del signo. De igual forma, si es la y se utilizan 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:

$$u = \frac{y + x}{2x} \qquad v = \frac{z + x}{2x} \tag{6.1}$$

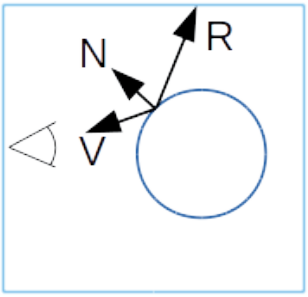


Figura 6.13: Vectores involucrados en *reflection mapping*

En WebGL una textura mapa de cubo se declara de tipo *samplerCube*. El cálculo de las coordenadas de textura se realiza de manera automática al acceder a la textura mediante la función *textureCube*, que se realiza habitualmente en el *shader* de fragmentos, igual que cuando se accedía a una textura 2D. El vector de reflexión R se calcula para cada vértice mediante la función *reflect*, y el procesador gráfico hará que cada fragmento reciba el vector R convenientemente interpolado. En el listado 6.3 se muestra el uso de estas funciones en el *shader*. La figura 6.14 muestra un ejemplo de mapa y del resultado obtenido.

Listado 6.3: Cambios en el *shader* para *reflection mapping*

```
// Shader de vértices
...
attribute vec3 VertexNormal
varying   vec3 R;  // vector de reflexión en el vértice

void main() {
  ...
  vec4 ecPosition = modelViewMatrix * vec4(vertexPosition,1.0);
  vec3 N          = normalize(normalMatrix * vertexNormal);
  vec3 V          = normalize(vec3(-ecPosition));
  ...
  R = reflect(V, N);
}

// Shader de fragmentos
...
uniform samplerCube myCubeMapTexture;
varying vec3 R;    // vector de reflexión interpolado

void main() {
  ...
  gl_FragColor = textureCube(myCubeMapTexture, R);
}
```

Figura 6.14: 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

Refraction mapping

Esta técnica de aplicación de textura se utiliza para simular objetos que la luz atraviesa como hielo, agua, vidrio, diamante, etc. (ver figura 6.15). Las coordenadas de textura se corresponden con el vector de refracción para cada punto de la superficie. El vector de refracción se calcula a partir de los vectores V y N uti-

lizando la ley de Snell. Esta ley establece que la luz se desvía al cambiar de un medio a otro (del aire al agua, por ejemplo) en base a un índice de refracción. El vector de refracción se calcula con la función *refract* para cada vértice en el *shader* de vértices y el procesador gráfico hará que cada fragmento reciba el vector convenientemente interpolado. En el listado 6.4 se muestra el uso de estas funciones en ambos *shaders*. Además, se combina el resultado de la reflexión y la refracción, ya que es habitual que los objetos refractantes también sean reflejantes. El peso se balancea utilizando la variable *refractionFactor*.

Listado 6.4: Cambios en el *shader* para *refraction mapping*

```
// Shader de vértices
...
attribute vec3 VertexNormal;
varying   vec3 RefractDir, ReflectDir;

void main() {
  ...
  ReflectDir = reflect(V, N);
  RefractDir = refract(V, N, material.refractionIndex);
}

// Shader de fragmentos
...
uniform samplerCube myCubeMapTexture;
varying vec3 RefractDir, ReflectDir;

void main() {
  ...
  gl_FragColor = mix (textureCube(myCubeMapTexture, RefractDir),
                      textureCube(myCubeMapTexture, ReflectDir),
                      material.refractionFactor);
}
```

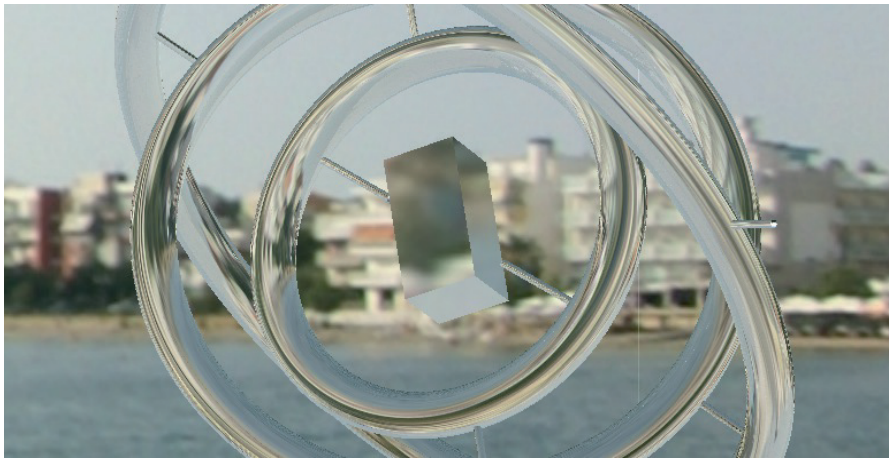


Figura 6.15: Ejemplo de *refraction mapping*

Skybox

Un *skybox* es un cubo muy grande situado alrededor del observador. El objetivo de este cubo es representar el fondo de una escena que habitualmente contiene elementos muy distantes al observador, como por ejemplo el sol, las montañas, las nubes, etc. Como textura se utiliza un mapa de cubo. Las coordenadas de textura se establecen en el *shader* de vértices, simplemente a partir del atributo de posición de cada vértice del *Skybox*. El procesador gráfico hará que cada fragmento reciba las coordenadas de textura convenientemente interpoladas. En el listado 6.5 se muestran los principales cambios para dibujar el *skybox*.

Listado 6.5: Cambios en el *shader* para *skybox*

```
// Shader de vértices
...
attribute vec3 VertexPosition;
varying   vec3 tcSkybox;  // coordenadas de textura del Skybox

void main() {

  // simplemente asigna a tcSkybox las coordenadas del vértice
  tcSkybox = VertexPosition;
  ...
}

// Shader de fragmentos
...
uniform samplerCube myCubeMapTexture;
varying vec3 tcSkybox;  // coordenadas de textura interpoladas

void main() {
  ...
  gl_FragColor = textureCube(myCubeMapTexture, tcSkybox);
}
```

6.3. Técnicas avanzadas

6.3.1. Normal 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. A esta técnica se le conoce también como *bump mapping*. El cálculo de la variación de la normal se puede realizar en el propio *shader* de manera procedural (ver imagen 6.16). Además, una función de ruido se puede utilizar para generar la perturbación (ver imagen 6.17). En esta sección, el mapa se va a precalcular y se proporcionará al procesador gráfico como una textura, conocida con el nombre de mapa de normales o *bump map* (ver figura 6.18).

El cálculo de la iluminación se ha de realizar en el espacio de la tangente. Este espacio se construye para cada vértice a partir de su normal, el vector tangente a

la superficie en dicho punto y el vector resultante del producto vectorial de esos dos vectores, que se denomina vector binormal. Con los tres vectores se crea la matriz que permite realizar el cambio de base. Como resultado, la normal coincide con el eje Z, la tangente con el eje X y la binormal con el eje Y. Entonces se operan los vectores L y V con esta matriz y hacemos que el procesador gráfico los interpole para cada fragmento. En el *shader* de fragmentos se accede a la textura que almacena el mapa de normales y a partir de esta normal y los vectores L y V interpolados se aplica el modelo de iluminación.

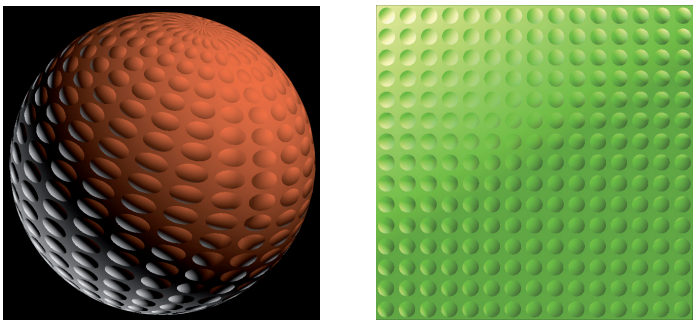


Figura 6.16: Objetos texturados con la técnica de *bump mapping*. La modificación de la normal produce que aparentemente la superficie tenga bultos

Figura 6.17: La normal del plano se perturba utilizando una función de ruido, haciendo que parezca que tenga pequeñas ondulaciones

Figura 6.18: Mapa de normales y su resultado aplicado sobre un modelo

6.3.2. Displacement mapping

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. En el *shader* de vértices se accede a la textura que almacena el mapa de desplazamiento y se modifica la posición del vértice, sumándole el resultado de multiplicar el desplazamiento por la normal en el vértice (ver figura 6.19).

Figura 6.19: Ejemplo de desplazamiento de la geometría

6.3.3. Alpha mapping

Esta técnica consiste en utilizar una textura para determinar qué partes de un objeto son visibles y qué partes no lo son. Esto permite representar objetos que geométricamente pueden tener cierta complejidad de una manera bastante sencilla a partir de una base geométrica simple y de la textura que hace la función de máscara. Es en el *shader* de fragmentos donde se accede a la textura para tomar esta decisión. Por ejemplo, la figura 6.20 muestra una textura y dos resultados de cómo se ha utilizado sobre el mismo objeto. En el primer caso, se ha utilizado para eliminar fragmentos, produciendo un objeto agujereado cuyo modelado sería bastante más complejo de obtener utilizando métodos tradicionales. En el segundo caso, el valor del *alpha map* se ha utilizado para decidir el color definitivo del fragmento. Con el mismo modelo y cambiando únicamente la textura es muy sencillo obtener acabados muy diferentes (ver figura 6.21).

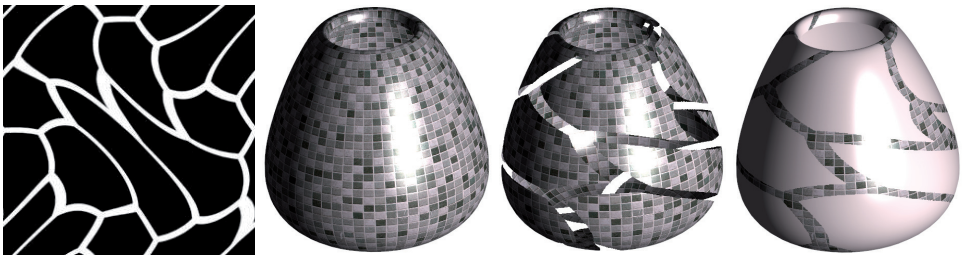


Figura 6.20: Ejemplos de aplicación de la técnica *alpha mapping*

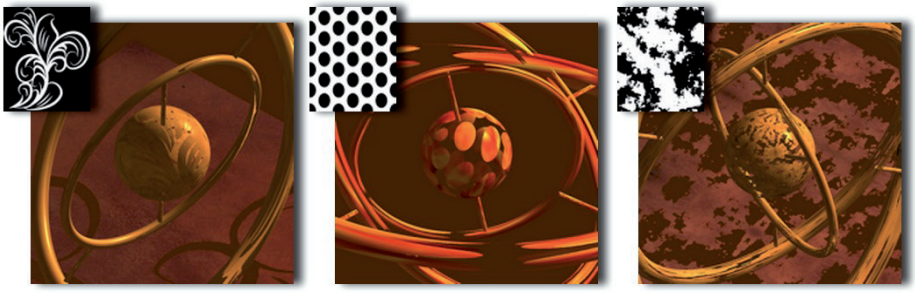


Figura 6.21: Ejemplos de aplicación de diferentes *alpha maps* sobre el mismo objeto

6.4. Texturas en WebGL

Para poder aplicar una textura en WebGL son necesarios tres pasos: crear un objeto textura, establecer los datos de la textura y asignar una unidad de textura.

En WebGL las texturas se representan mediante objetos con nombre. El nombre no es más que un entero sin signo donde el valor 0 está reservado. Para crear un objeto textura es necesario obtener en primer lugar un nombre que no se esté utilizando. Esta solicitud se realiza con la orden *gl.createTexture*. Se han de solicitar tantos nombres como objetos textura necesitemos, teniendo en cuenta que un objeto textura almacena una única textura.

Una vez creado este objeto, hay que especificar tanto la textura (la imagen 2D en nuestro caso) como los diferentes parámetros de repetición y filtrado. Para especificar la textura se utiliza la siguiente orden:

- `gl.texImage2D (GLenum` *objetivo*`, GLint` *nivel*`,`
  `GLenum` *formatoInterno*`, GLenum` *formato*`,`
  `GLenum` *tipo*`, TexImageSource` *datos*`);`

donde el 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 solo en el caso de usar diferentes resoluciones de la textura, siendo 0 en cualquier otro caso. El parámetro *formatoInterno* se utiliza para indicar cuáles de las componentes R, G, B y A se emplean como téxeles de la imagen.

Ya solo queda asignar el objeto textura a una unidad de textura. Estas unidades 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 *gl.activeTexture* especifica el selector de unidad de textura activa. Así, por ejemplo, la orden *gl.activeTexture(gl.TEXTURE0)* selecciona la unidad de textura 0. A continuación, hay que especificar el objeto textura a utilizar por dicha unidad con la orden *gl.bindTexture*. A cada unidad de textura solo se le puede asignar un objeto textura pero, durante la ejecución, podemos cambiar tantas veces como queramos el objeto textura asignado a una determinada unidad. El listado 6.6 muestra un ejemplo que incluye todos los pasos.

Listado 6.6: Creación de una textura en WebGL

```
// Crea un objeto textura
var texture = gl.createTexture();
gl.bindTexture(gl.TEXTURE_2D, texture);

// Especifica la textura RGB
gl.texImage2D (gl.TEXTURE_2D, 0, gl.RGB, gl.RGB,
    gl.UNSIGNED_BYTE, image);

// Repite la textura tanto en s como en t
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.REPEAT);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.REPEAT);

// Filtrado
gl.texParameteri (gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
    gl.LINEAR_MIPMAP_LINEAR);
gl.texParameteri (gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER,
    gl.LINEAR);
gl.generateMipmap(gl.TEXTURE_2D);

// Activa la unidad de textura 0
gl.activeTexture(gl.TEXTURE0);

// Asigna el objeto textura a dicha unidad de textura
gl.bindTexture  (gl.TEXTURE_2D, texture);
```

Una vez creada la textura no hay que olvidar asignar a la variable de tipo *sampler* del *shader* la unidad de textura que ha de utilizar (ver listado 6.7).

Listado 6.7: Asignación de unidad a un *sampler2D*

```
// Obtiene el índice de la variable del shader de tipo sampler2D
program.textureIndex =
  gl.getUniformLocation(program, 'myTexture');

// Indica que el sampler myTexture del shader use
// la unidad de textura 0
gl.uniform1i(program.textureIndex, 0);
```

Como se explicó en el primer punto de este capítulo, los vértices han de proporcionar un nuevo atributo: las coordenadas de textura. En consecuencia, hay que habilitar el atributo (ver listado 6.8) y también especificar cómo se encuentra almacenado (ver listado 6.9).

Listado 6.8: Habilitación del atributo de coordenada de textura

```
// se obtiene la referencia al atributo
program.vertexTexcoordsAttribute =
  gl.getAttribLocation ( program, "VertexTexcoords");

// se habilita el atributo
gl.enableVertexAttribArray(program.vertexTexcoordsAttribute);
```

Listado 6.9: Especificación de tres atributos: posición, normal y coordenadas de textura

```
gl.bindBuffer (gl.ARRAY_BUFFER, model.idBufferVertices);
gl.vertexAttribPointer (program.vertexPositionAttribute,   3,
    gl.FLOAT, false, 8*4,   0);
gl.vertexAttribPointer (program.vertexNormalAttribute,     3,
    gl.FLOAT, false, 8*4, 3*4);
gl.vertexAttribPointer (program.vertexTexcoordsAttribute, 2,
    gl.FLOAT, false, 8*4, 6*4);
```

Ejercicios

► **6.1** Ejecuta el programa *c06/textura2D.html*. Experimenta y prueba a cargar diferentes texturas. Como resultado, obtendrás imágenes similares a las de la figura 6.22. Después edita los códigos *textura2D.html* y *textura2D.js*. Comprueba cómo se han incorporado en el código todos los cambios que se han explicado en esta sección, necesarios para poder utilizar texturas 2D.

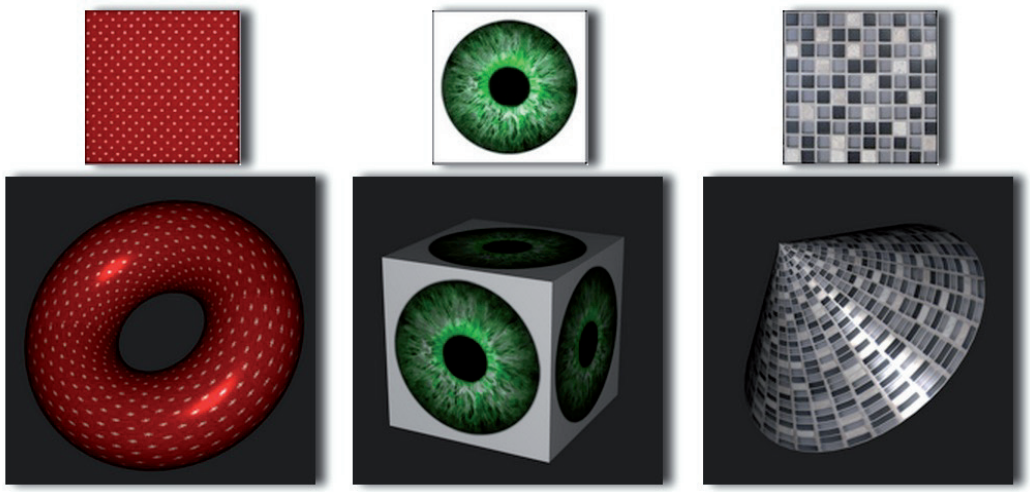


Figura 6.22: Ejemplos obtenidos utilizando texturas en WebGL

► **6.2** Con la ayuda de un editor gráfico, crea un *alpha map* o búscalo en internet. Cárgalo como una nueva textura y utilízalo para obtener resultados como los que se mostraban en las figuras 6.20 y 6.21.

► **6.3** Añade una textura de mapa de cubo. Utilízalo como *skybox* y para que el modelo refleje su entorno. Observa las imágenes de la figura 6.23 para ver los resultados que podrías obtener. Interesantes, ¿verdad? Sin duda, un ejemplo inspirado en *Fiat Lux* de Paul Debevec. El listado 6.10 contiene el *shader* que se ha utilizado. Observa que con el mismo *shader* se pinta el cubo y el modelo. La variable `skybox` se utiliza para saber si se está pintando el cubo o el modelo. Por otra parte, la matriz *invT* es la inversa de la matriz modelo-vista del *skybox*. Recuerda que el *skybox* ha de estar centrado en la posición de la cámara. El listado 6.11 muestra la función *loadCubeMap*, encargada de crear la textura de mapa de cubo.

Figura 6.23: Ejemplos obtenidos utilizando *reflection mapping* en WebGL

Listado 6.10: *Shader* para *skybox* y *reflection mapping*

```
// Shader de vértices
...
uniform   bool skybox;
uniform   mat3 invT;
varying   vec3 texCoords;

void main()  {

  vec3 N          = normalize(normalMatrix * VertexNormal);
  vec4 ecPosition = modelViewMatrix * vec4(VertexPosition,1.0);
  vec3 ec         = vec3(ecPosition);
  gl_Position     = projectionMatrix * ecPosition;

  if (skybox)
    texCoords = vec3(VertexPosition);
  else
    texCoords = invT * reflect (normalize(ec),N);
}

// Shader de fragmentos
precision mediump float;

uniform samplerCube myTexture;
varying vec3 texCoords;

void main() {

  gl_FragColor = textureCube(myTexture, texCoords);

}
```

► **6.4** Añade refracción al ejercicio anterior (ver figura 6.24). Utiliza índices de refracción menores que 1 (esto es debido a la implementación de la función *refract* en GLSL). Para cada fragmento, haz que el color final sea un 15 % el valor del reflejo y un 85 % el de refracción. Utiliza la función *mix*.

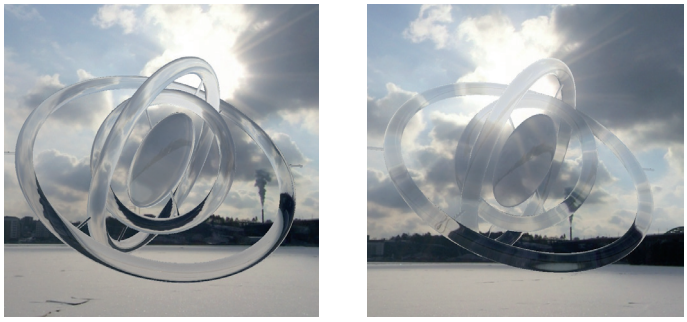


Figura 6.24: Ejemplos obtenidos utilizando *refraction* y *reflection mapping* en WebGL. El índice de refracción es, de izquierda a derecha, de 0,95 y 0,99

Listado 6.11: Función para crear la textura de mapa de cubo

```
function loadCubeMap() {

  var texture = gl.createTexture();
  gl.activeTexture(gl.TEXTURE0);
  gl.bindTexture  (gl.TEXTURE_CUBE_MAP, texture);
  gl.texParameteri(gl.TEXTURE_CUBE_MAP, gl.TEXTURE_WRAP_S,
      gl.CLAMP_TO_EDGE);
  gl.texParameteri(gl.TEXTURE_CUBE_MAP, gl.TEXTURE_WRAP_T,
      gl.CLAMP_TO_EDGE);
  gl.texParameteri(gl.TEXTURE_CUBE_MAP, gl.TEXTURE_MIN_FILTER,
      gl.LINEAR);
  gl.texParameteri(gl.TEXTURE_CUBE_MAP, gl.TEXTURE_MAG_FILTER,
      gl.LINEAR);

  var faces = [["posx.jpg", gl.TEXTURE_CUBE_MAP_POSITIVE_X],
               ["negx.jpg", gl.TEXTURE_CUBE_MAP_NEGATIVE_X],
               ["posy.jpg", gl.TEXTURE_CUBE_MAP_POSITIVE_Y],
               ["negy.jpg", gl.TEXTURE_CUBE_MAP_NEGATIVE_Y],
               ["posz.jpg", gl.TEXTURE_CUBE_MAP_POSITIVE_Z],
               ["negz.jpg", gl.TEXTURE_CUBE_MAP_NEGATIVE_Z]];

  for (var i = 0; i < faces.length; i++) {
    var face = faces[i][1];
    var image = new Image();
    image.onload = function(texture,face, image) {
      return function() {
        gl.texImage2D(face, 0, gl.RGBA, gl.RGBA,
            gl.UNSIGNED_BYTE, image);
      }
    } (texture,face, image);
    image.src = faces[i][0];
  }

  program.textureIndex = gl.getUniformLocation(program,
      'myTexture');
  gl.uniform1i(program.textureIndex, 0);
}
```

Capítulo 7

Realismo visual

Este capítulo presenta tres tareas básicas en la búsqueda del realismo visual en imágenes sintéticas: transparencia, reflejos y sombras. En la literatura se han presentado numerosos métodos para cada una de ellas. Al igual que en capítulos anteriores, se van a presentar aquellos métodos que, aun siendo básicos, consiguen una mejora importante en la calidad visual con poco esfuerzo de programación.

7.1. Transparencia

Los objetos transparentes son muy habituales en el mundo que nos rodea. Suele ocurrir que estos objetos produzcan un efecto de refracción de la luz, o que la luz cambie alguna de sus propiedades al atravesarlos. Todo esto hace que la inclusión de objetos transparentes en un mundo virtual sea un problema complejo de resolver. En esta sección, se va a abordar el caso más sencillo, es decir, suponer que el objeto transparente es muy fino y no va a producir el efecto de refracción de la luz ni va a modificar las propiedades de las fuentes de luz. Quizá pueda parecer que se simplifica mucho el problema, lo cual es cierto, pero aún así la ganancia visual que se va a obtener es muy alta.

Cuando una escena incluye un objeto transparente, el color de los píxeles cubiertos por dicho objeto depende, además de las propiedades del objeto transparente, de los objetos que hayan detrás de él. Un método sencillo para incluir objetos transparentes en nuestra escena consiste en dibujar primero todos los objetos que sean opacos y dibujar después los objetos transparentes. El grado de transparencia se suministra al procesador gráfico como una cuarta componente en las propiedades de material del objeto, conocida como componente *alfa*. Si *alfa* es 1, el objeto es totalmente opaco, y si es 0 significa que el objeto es totalmente transparente (ver imagen 7.1). Así, el color final se calcula a partir del color del *framebuffer* y del color del fragmento de esta manera:

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

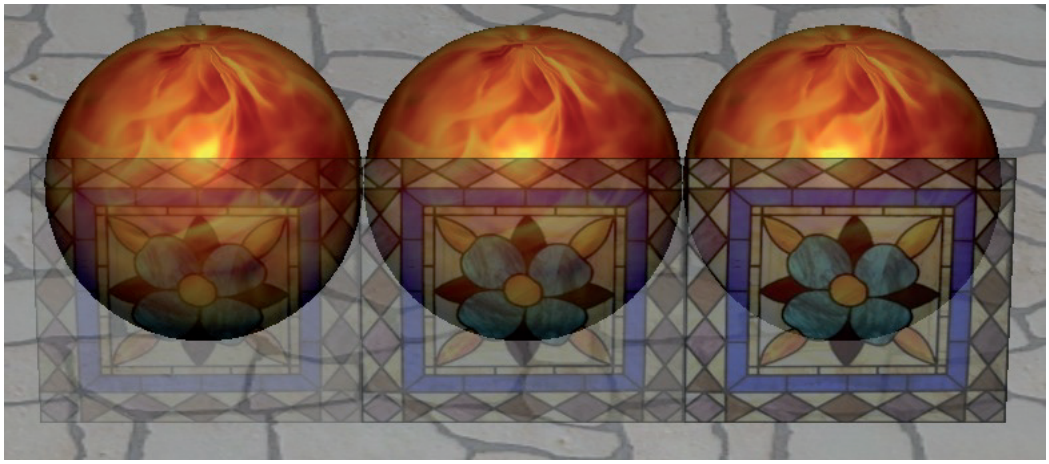


Figura 7.1: Tres ejemplos de transparencia con, de izquierda a derecha, $alfa = 0{,}3$, $0{,}5$ y $0{,}7$

Al dibujar un objeto transparente, el test de profundidad se tiene que 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 transparente deja ver a través de él, para cada uno de los fragmentos que supere el test deberá actualizarse el *buffer* de color, pero no el de profundidad, ya que de hacerlo evitaría que otros objetos transparentes situados detrás fuesen visibles. El listado 7.1 recoge la secuencia de órdenes de WebGL necesaria para poder incluir objetos transparentes en la escena.

Listado 7.1: Secuencia de operaciones para dibujar objetos transparentes

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

// activa el cálculo de la transparencia
gl.enable (gl.BLEND);

// especifica la funcion de cálculo
gl.blendFunc (gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA);

// impide la actualización del buffer de profundidad
gl.depthMask (gl.FALSE);

// dibuja los objetos transparentes
...

// inhabilita la transparencia
gl.disable    (gl.BLEND);

// permite actualizar el buffer de profundidad
gl.depthMask (gl.TRUE);
```

En el caso de que haya varios objetos transparentes 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 (ver figura 7.2). En este caso, habría que dibujar los objetos transparentes de manera ordenada, pintando en primer lugar el más lejano y, en último lugar, el más cercano al observador. Sin embargo, en ocasiones la ordenación no es trivial, como por ejemplo cuando un objeto atraviesa otro. En estos casos, una forma de evitar este problema es establecer la operación de cálculo de la transparencia como un incremento sobre el color acumulado en el *framebuffer*:

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

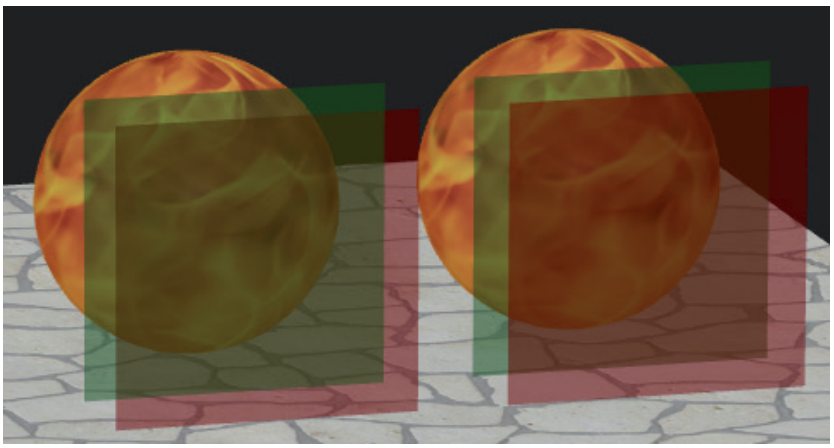


Figura 7.2: Dos resultados diferentes en los que únicamente se ha variado el orden en el dibujado de los objetos transparentes

En WebGL, 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 transparentes ya no influye en el color final de las zonas solapadas. Por contra, las zonas visibles a través de los objetos transparentes son más brillantes que en el resto, produciendo una diferencia que en general resulta demasiado notable.

Ejercicios

► **7.1** Observa los dos ejemplos de la figura 7.2 y contesta, ¿cuál de las dos figuras te parece más correcta visualmente? ¿En qué orden crees que se han pintado los objetos transparentes que aparecen delante de cada esfera?

Hasta el momento se han utilizado polígonos individuales como objetos transparentes, lo que podría ser suficiente para simular por ejemplo una ventana. Pero,

¿qué ocurre si es un objeto más complejo en el que lo que se ve a través de él es a él mismo? En esos casos, se debe prestar atención especial al orden de dibujado. Dado un objeto transparente, una solución muy sencilla consiste en dibujar primero las caras traseras del objeto (que dependen de la posición del observador) y después las caras de delante del objeto. Por suerte, el procesador gráfico implementa en su *pipeline* la eliminación de caras de manera automática. El programador debe habilitarlo (`gl.enable(gl.CULL_FACE)`) e indicar si quiere eliminar las caras de la parte trasera (`gl.cullFace(gl.BACK)`) o de la delantera (`gl.cullFace(gl.FRONT)`). El listado 7.2 recoge la secuencia de órdenes de WebGL necesaria para poder visualizar de manera correcta este tipo de objetos transparentes. Además, suele ser conveniente aplicar la iluminación por ambas caras, ya que la parte trasera ahora se ilumina gracias a la transparencia del propio objeto. La figura 7.2 muestra algunos resultados obtenidos con esta técnica.

Listado 7.2: Objetos transparentes

```
// Primero pinta los objetos opacos
....
// Después los transparentes
gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA);
gl.enable (gl.BLEND);        // habilita la transparencia
gl.enable (gl.CULL_FACE);    // habilita el face culling
gl.depthMask(false);

gl.cullFace(gl.FRONT);   // se eliminan los de cara al observador
drawSolid(exampleCube); // se dibuja el objeto transparente

gl.cullFace(gl.BACK);    // se eliminan los de la parte trasera
drawSolid(exampleCube); // se vuelve a dibujar el objeto

gl.disable(gl.CULL_FACE);
gl.disable(gl.BLEND);
gl.depthMask(true);
```

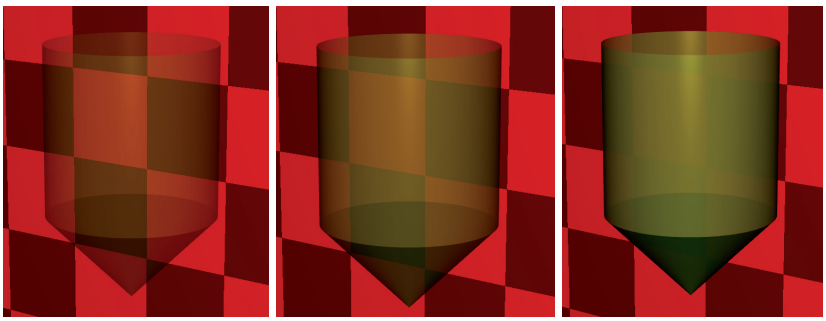


Figura 7.3: Ejemplo de objeto transparente con, de izquierda a derecha, $alfa = 0,3$, $0,5$ y $0,7$

Ejercicios

► **7.2** Consulta la información del *pipeline* de WebGL y averigua en qué etapas tienen lugar las operaciones de *face culling* y *blending*.

► **7.3** Ejecuta el programa *c07/transparencia/transparencia.html*. Comprueba que en el código figura todo lo necesario para dibujar el objeto de manera transparente. Observa la escena moviendo la cámara y contesta, ¿crees que el objeto transparente se observa de forma correcta desde cualquier punto de vista? Si no es así, ¿por qué crees que ocurre?

7.2. Sombras

En el mundo real, si hay fuentes de luz, habrá 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, 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.

7.2.1. Sombras proyectivas

Un método 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 objeto sombra, es decir, el resultado de la proyección, 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:

$$M = \begin{pmatrix} N \cdot L + d - L_x N_x & -L_x N_y & -L_x N_z & -L_x d \\ -L_y N_x & N \cdot L + d - L_y N_y & -L_y N_z & -L_y d \\ -L_z N_x & -L_z N_y & N \cdot L + d - L_z N_z & -L_z d \\ -N_x & -N_y & -N_z & N \cdot L \end{pmatrix} \tag{7.3}$$

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*. WebGL proporciona la orden *gl.polygonOffset* para especificar el desplazamiento, que se sumará al valor de profundidad de cada fragmento siempre y cuando se haya habilitado con *gl.enable (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 transparencia se puede conseguir un resultado mucho más agradable, ya que deja entrever el plano sobre el que se asientan (ver figura 7.4).
- Muy complejo para superficies curvas o para representar las sombras que caen sobre el propio objeto.

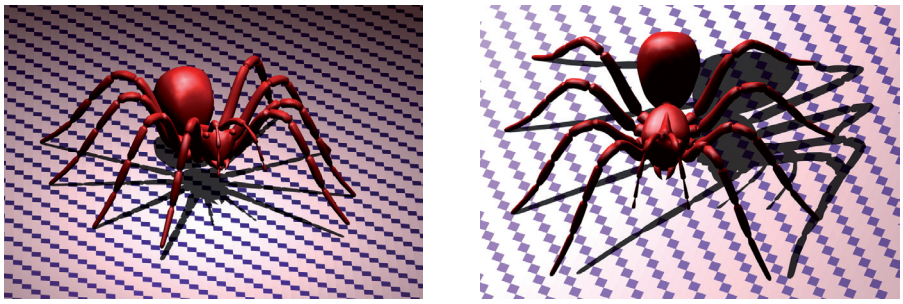


Figura 7.4: Ejemplo de sombras proyectivas transparentes

7.2.2. Shadow mapping

Si la escena se observa desde la posición donde se ubica la fuente de luz, lo que se consigue es ver justo lo que la fuente de luz ilumina, por lo tanto se cumple también que estará en sombra lo que la luz no ve. Este método se basa en dibujar primero la escena vista desde la fuente de luz con el objetivo de crear un mapa de profundidad y almacenarlo como textura. Después, se dibuja la escena vista desde la posición del observador pero consultando la textura de profundidad para saber si un fragmento está en sombra y pintarlo de manera acorde.

Para obtener dicho mapa de profundidad con WebGL, es necesario dibujar la escena contra un *framebuffer object* (FBO). El procesador gráfico puede dibujar en un FBO diferente del creado por defecto, y en ese caso su contenido no es visible al usuario. Tras crear el FBO, se dibuja la escena y en el *shader* de fragmentos se establece como color final del fragmento su valor de profundidad (ver imagen de la izquierda en la figura 7.5):

- `gl_FragColor = vec4(vec3(gl_FragCoord.z),1.0);`

El contenido del FBO se almacena como un objeto textura de manera que pueda ser consultado más tarde. Al dibujar la escena vista desde la posición del observador, cada vértice del modelo se ha de operar también con la matriz de transformación de la cámara situada en la fuente de luz. De esta manera, para cada fragmento se puede comparar su profundidad con la almacenada en la textura y saber si el

fragmento está en sombra o no. La figura 7.5 muestra un ejemplo de esta técnica, donde se puede observar el mapa de profundidad obtenido desde la posición de la fuente de luz y la vista con sombras desde la posición de la cámara.

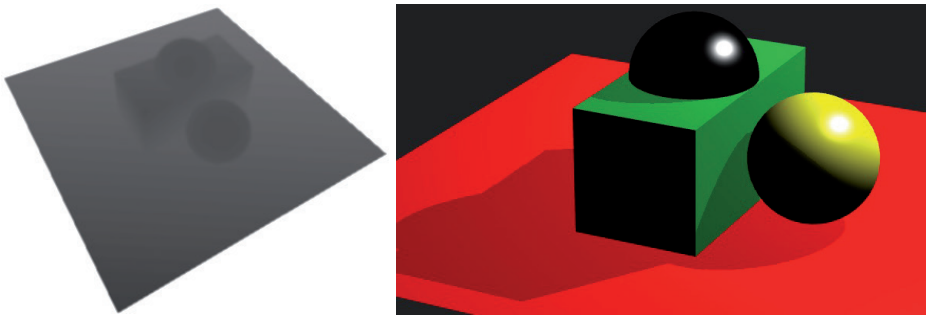


Figura 7.5: Ejemplo de *shadow mapping*. A la izquierda se observa el mapa de profundidad obtenido desde la fuente de luz; a la derecha se muestra la escena con sus sombras

Ejercicios

► **7.4** Ejecuta el programa *c07/sombras/sombras.html*. Observa la escena mientras mueves la cámara y contesta, ¿detectas algún tipo de problema en el dibujado de las sombras?, ¿a qué crees que se debe? Modifica el código para que la dimensión de la textura de profundidad sea de un tamaño mucho más pequeño, ¿qué ocurre ahora?, ¿por qué?

7.3. Reflejos

Los objetos reflejantes, al igual que los transparentes, son también muy habituales en cualquier escenario. Es fácil encontrar 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 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 cómo añadir superficies planas reflejantes a nuestra escena (ver figura 7.6). El método consiste en dibujar la escena de forma simétrica respecto al plano que contiene el 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 y deshacer el giro y la traslación. Dado un punto P del objeto plano reflejante y un vector V perpendicular al plano, la matriz de trasformación M es la siguiente:

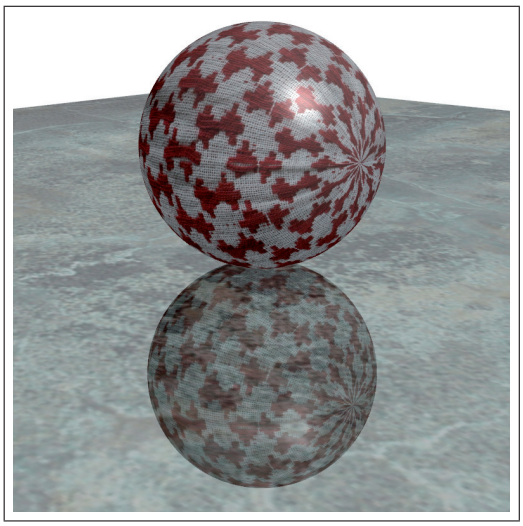


Figura 7.6: Ejemplo de objeto reflejado en una superficie plana

$$M = \begin{pmatrix} 1-2V_x^2 & -2V_xV_y & -2V_xV_z & 2(P\cdot V)V_x \\ -2V_xV_y & 1-2V_y^2 & -2V_yV_z & 2(P\cdot V)V_y \\ -2V_xV_z & -2V_yV_z & 1-2V_z^2 & 2(P\cdot V)V_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{7.4}$$

Para la segunda tarea, la de no dibujar fuera de los límites del objeto reflejante (ver figuras 7.7 y 7.8), 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, de manera opcional, hay que dibujar el objeto reflejante utilizando transparencia. El listado 7.3 muestra cómo se realizan estos pasos con WebGL. El ejemplo *c07/reflejos/reflejos.html* recoge todas las operaciones descritas, produciendo imágenes como la de la figura 7.8.

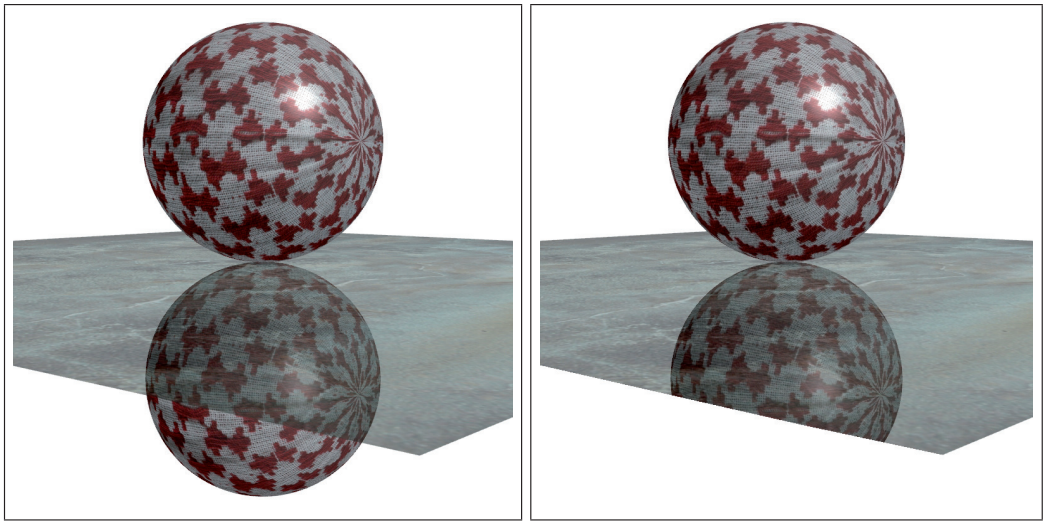


Figura 7.7: Al dibujar la escena simétrica es posible observarla fuera de los límites del objeto reflejante (izquierda). El *buffer* de plantilla se puede utilizar para resolver el problema (derecha)

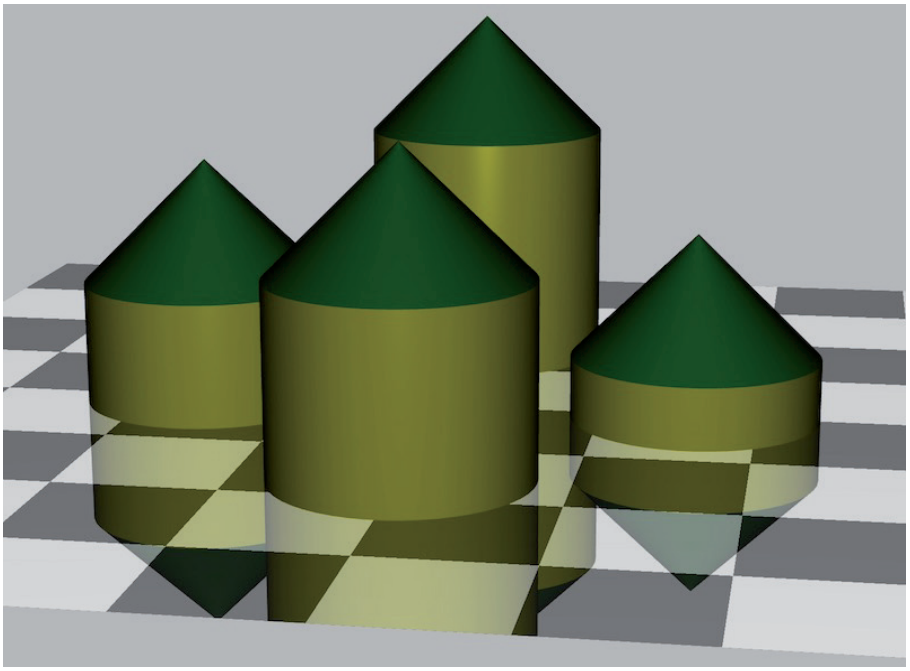


Figura 7.8: Ejemplo de reflejo plano

Listado 7.3: Secuencia de operaciones para dibujar objetos reflejantes

```
gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT |
    gl.STENCIL_BUFFER_BIT);

// Desactiva los buffers de color y profundidad
gl.disable(gl.DEPTH_TEST);
gl.colorMask(false,false,false,false);

// Establece como valor de referencia el 1
gl.enable(gl.STENCIL_TEST);
gl.stencilOp(gl.REPLACE,gl.REPLACE,gl.REPLACE);
gl.stencilFunc(gl.ALWAYS,1,0xFFFFFFFF);

// Dibuja el objeto reflejante
...

// Activa de nuevo los buffers de profundidad y de color
gl.enable(gl.DEPTH_TEST);
gl.colorMask(true,true,true,true);

// Configura el buffer de plantilla
gl.stencilOp(gl.KEEP,gl.KEEP,gl.KEEP);
gl.stencilFunc(gl.EQUAL,1,0xFFFFFFFF);

// Dibuja la escena reflejada
...

// Desactiva el test de plantilla
gl.disable(gl.STENCIL_TEST);

// Dibuja la escena normal
...

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

Capítulo 8

Texturas procedurales

Una textura procedural se caracteriza porque la información propia de la textura la genera un procedimiento en tiempo de ejecución, ss decir, la textura no se encuentra previamente almacenada, tal y como se explicó en el capítulo 6. Ahora, existe una función que implementa la textura y a esta función se le llama desde el *shader* para, a partir de unas coordenadas de textura, computar y devolver un valor. La función puede devolver tanto un valor de color como cualquier otro valor que se pueda utilizar para determinar el aspecto final del modelo (ver figura 8.1). Sin duda, el uso de texturas procedurales es una de las grandes ventajas que ofrecen los actuales procesadores gráficos programables.

Figura 8.1: Ejemplo de objeto dibujado con una textura procedural. En este caso, el valor devuelto por la función de textura se utiliza para determinar si hay que eliminar un determinado fragmento

Son varias las ventajas que ofrece el uso de las texturas procedurales. Por ejemplo, una textura procedural, en general, va a consumir menos memoria, ya que al no estar discretizada no presenta los problemas derivados de tener una resolución fija y cualquier aspecto clave de la textura se puede parametrizar para obtener efectos muy diversos utilizando la misma función. Por ejemplo, las copas de la figura 8.1 se han obtenido utilizando exactamente la misma textura procedural, pero para cada una se han asignado valores diferentes a las variables que gobiernan el número y tamaño de los agujeros.

También es cierto que no todo son ventajas. Por ejemplo, computacionalmente son más caras y los algoritmos que hay que implementar a veces son muy com-

plejos, incluso también podría ocurrir que los resultados fueran diferentes según el procesador gráfico que se utilice.

En definitiva, combinar texturas basadas en imágenes junto con texturas procedurales será casi siempre lo ideal. Por ejemplo, para una pelota de golf como la de la figura 8.2 se pueden utlizar imágenes para el logo, manchas o arañazos, y una función procedural para los hoyuelos.

Figura 8.2: Ejemplo de combinación de texturas 2D y textura procedural

8.1. Rayado

El objetivo de este tipo de textura procedural es mostrar un rayado sobre la superficie del objeto (ver figura 8.3). Se utiliza una coordenada de textura y el rayado que se obtiene siempre será en la dirección de la coordenada elegida. Se aplica un factor de escala sobre la coordenada de textura para establecer el número de rayas. La parte real de la coordenada de textura se compara con una variable que controla el ancho de la raya para saber si el fragmento se ha de colorear con el color de la raya o con el del material del modelo. El listado 8.1 muestra estas operaciones.

Ejercicios

► **8.1** Ejecuta el ejemplo *c08/rayado/rayado.html*. Prueba a modificar los parámetros que gobiernan el rayado. Ahora, edítalo y prueba a utilizar la otra coordenada de textura.

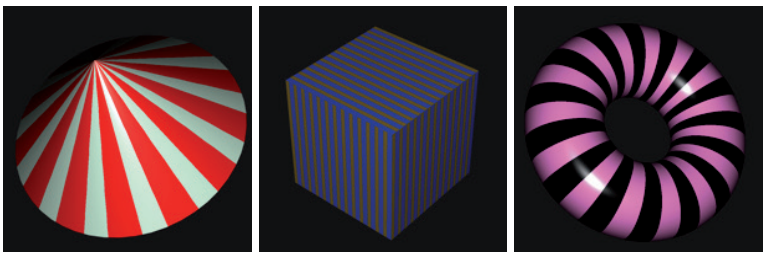


Figura 8.3: Ejemplos del *shader* de rayado

Listado 8.1: *Shader* de rayado

```
// Shader de vértices
...
varying float TexCoord;

void main()  {
  ...
  TexCoord = VertexTexcoords.t;

}

// Shader de fragmentos
...
uniform vec3  StripeColor;     // color de la raya
uniform float Scale;           // número de rayas
uniform float Width;           // ancho de la raya

varying float TexCoord;

void main() {

  float scaledT = fract(TexCoord * Scale);
  float       s = step (Width, scaledT);
  vec3    newKd = mix  (StripeColor, Kd, s);
  ...
  gl_FragColor  = vec4 (phong(newKd,N,L,V), 1.0);

}
```

8.2. Damas

Este tipo de textura procedural consiste en presentar la superficie de un objeto como la del tablero del juego de las damas (ver figura 8.4). Se utiliza un factor de escala sobre las coordenadas de textura para establecer el número de cuadrados, en principio el mismo en ambas direcciones. La parte entera de las coordenadas de textura escaladas se utiliza para saber a qué fila y columna pertenece cada fragmento. Sumando ambas y obteniendo el módulo dos se averigua si el resultado es par o impar y, en consecuencia, se sabe si el fragmento pertenece a una casilla blanca o negra. El listado 8.2 recoge los cambios necesarios.

Ejercicios

► **8.2** Ejecuta el ejemplo *c08/damas/damas.html*. Prueba a modificar el parámetro que gobierna el número de cuadrados. Ahora, edita el código y modifícalo para que se pueda establecer un número diferente de cuadrados en cada dirección.

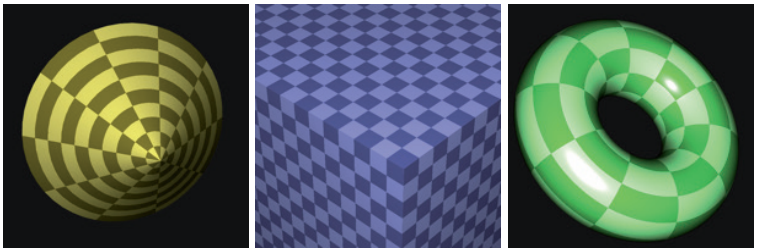


Figura 8.4: Ejemplos del *shader* de damas

Listado 8.2: *Shader* de damas

```
// Shader de vértices
...
varying vec2 TexCoord;

void main()  {
  ...
  TexCoord = VertexTexcoords;

}

// Shader de fragmentos
...
uniform float Scale;    // número de cuadrados

varying vec2 TexCoord;

void main() {

  float row = floor( TexCoord.s * Scale );
  float col = floor( TexCoord.t * Scale );

  float res   = mod( row + col , 2.0);
  vec3  newKd = Kd + (res * 0.4);
  ...
  gl_FragColor = vec4(phong(newKd,N,L,V), 1.0);

}
```

8.3. Enrejado

El enrejado consiste en utilizar la orden *discard* para eliminar fragmentos, produciendo en consecuencia agujeros en la superficie del objeto (ver figuras 8.5 y 8.1). Las coordenadas de textura se escalan para determinar el número de agujeros, utilizando un factor de escala distinto para cada dirección. La parte real se utiliza para controlar el tamaño del agujero. El listado 8.3 recoge los cambios necesarios.

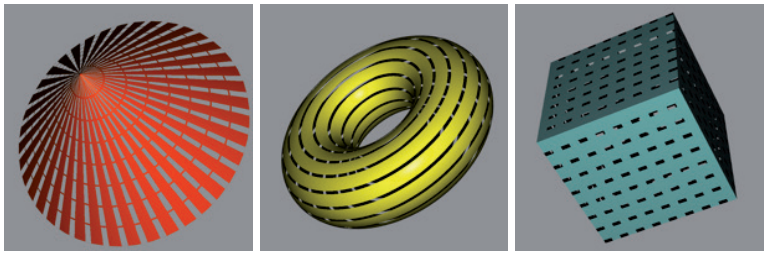


Figura 8.5: Ejemplos del *shader* de enrejado

Listado 8.3: *Shader* de enrejado

```
// Shader de vértices
...
varying vec2 TexCoord;

void main() {
  ...
  TexCoord = VertexTexcoords;

}

// Shader de fragmentos
...
uniform vec2 Scale;
uniform vec2 Threshold;

varying vec2 TexCoord;

void main() {

  float ss = fract(TexCoord.s * Scale.s);
  float tt = fract(TexCoord.t * Scale.t);

  if ((ss > Threshold.s) && (tt > Threshold.t))
    discard;
  ...
  gl_FragColor = vec4(phong(N,L,V), 1.0);

}
```

Ejercicios

► **8.3** Ejecuta el ejemplo *c08/enrejado/enrejado.html*. Prueba a modificar los parámetros que gobiernan el número de agujeros y su tamaño. Ahora, edita el código y modifícalo para que se eliminen los fragmentos que no pertenecen a la superficie de un círculo. Fíjate en los resultados que se muestran en las imágenes de la figura 8.6.

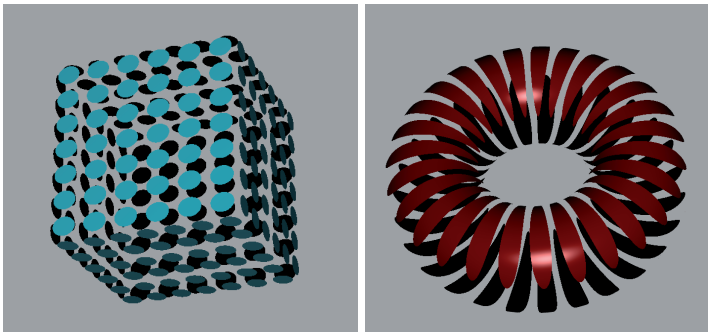


Figura 8.6: Ejemplos de enrejados circulares

8.4. Ruido

En términos generales, el uso de ruido en informática gráfica resulta muy útil para simular efectos atmosféricos, materiales o simplemente imperfecciones en los objetos. La figura 8.7 muestra algunos ejemplos en los que se ha utilizado una función de ruido como método de textura procedural. A diferencia de otras áreas, la función de ruido que nos interesa ha de ser repetible, es decir, que produzca siempre la misma salida para el mismo valor de entrada y que al mismo tiempo aparente aleatoriedad, es decir, que no muestre patrones regulares.

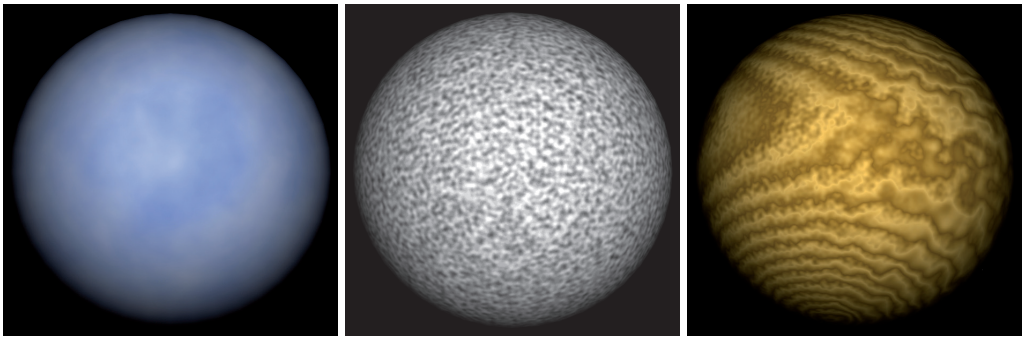


Figura 8.7: Ejemplos obtenidos utilizando una función de ruido como textura procedural

El uso de ruido en WebGL se puede realizar de tres maneras. Una es utilizar la familia de funciones *noise* de GLSL. Por desgracia, algunos fabricantes de *hardware* gráfico no implementan las funciones *noise*. Otra manera es implementar la función de ruido en un programa externo, ejecutarlo y almacenar el resultado para proporcionarlo en forma de textura al procesador gráfico, pero en este caso ya no podemos hablar de textura procedural. Por último, está la alternativa de implementar una función de ruido propia en el *shader*. Por ejemplo, las figuras 8.8 y 8.9 muestran diversos ejemplos de objetos en los que se ha utilizado una función de ruido de Perlin en el *shader* de fragmentos para a partir de la posición interpolada de cada fragmento, obtener un valor de ruido y combinarlo de diferentes formas con los valores de material.

Ejercicios

► **8.4** Ejecuta los ejemplos *c08/ruido/nubes.html* y *c08/ruido/sol.html*. Prueba a modificar los parámetros que gobiernan el aspecto final de los modelos. Ahora, edita el código y prueba a realizar modificaciones a partir de los valores de ruido obtenidos, seguro que consigues resultados interesantes.

Figura 8.8: Objetos que hacen uso de una función de ruido para colorear su superficie

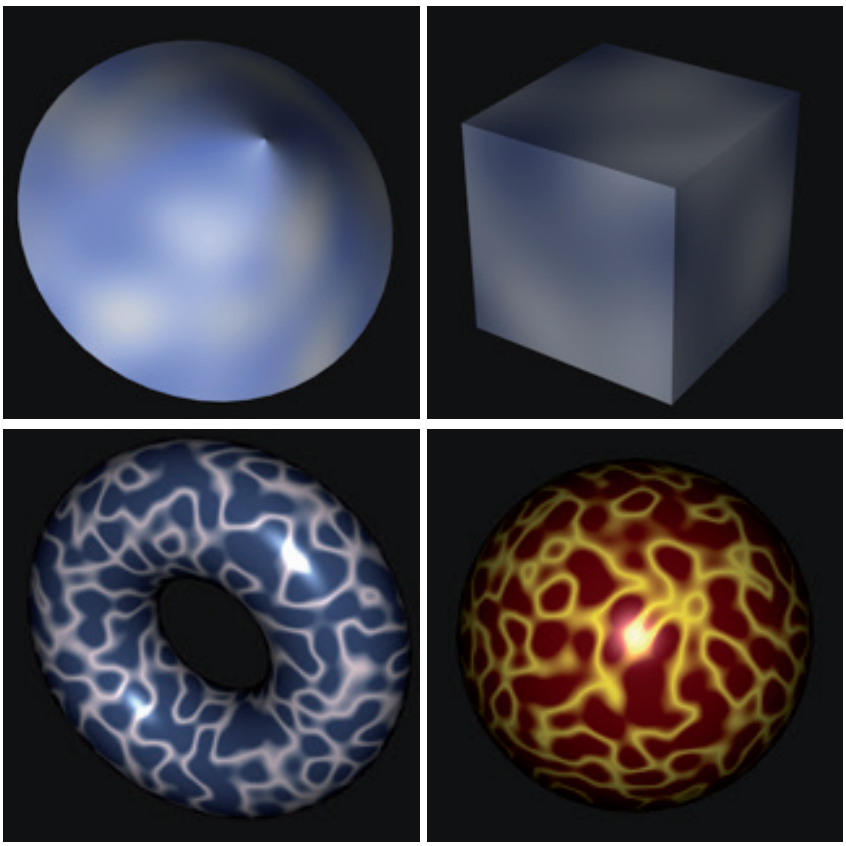


Figura 8.9: Ejemplos obtenidos con la función de ruido de Perlin

Capítulo 9

Interacción y animación con *shaders*

Este capítulo junta técnicas de interacción con métodos básicos de animación con *shaders*. Es un capítulo de índole muy práctica que se acompaña de ejemplos que complementan los métodos explicados.

9.1. Selección

En una aplicación interactiva es fácil que el usuario pueda señalar objetos de la escena y que, por tanto, la aplicación necesite saber de qué objeto se trata. Habitualmente, el usuario utiliza el ratón para mover el puntero y mediante el botón izquierdo realiza la selección al presionarlo, pero también puede hacerlo con el dedo en el caso de utilizar dispositivos móviles con pantalla táctil. En cualquier caso, como resultado de la interacción se produce un evento que es necesario atender para averiguar las coordenadas del píxel sobre el que se hizo el clic.

El método que se propone parte de ese evento averigua las coordenadas de dicho píxel y borra el canvas para pintar cada objeto seleccionable de un color diferente y plano. De esta manera, si ahora se accede al canvas en las coordenadas elegidas, se puede averiguar el color del píxel correspondiente, y sabiendo el color se sabe a qué objeto pertenece. Ya solo queda borrar el canvas y pintar la escena que se ha de mostrar al usuario.

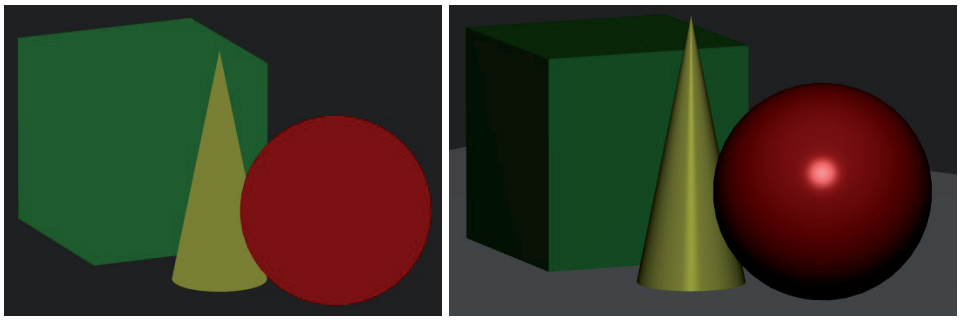
Para averiguar las coordenadas del píxel seleccionado hay que tener en cuenta que el origen de coordenadas en el navegador está en la esquina superior izquierda de la página. Por lo tanto, al hacer el clic, el manejador de eventos nos proporciona las coordenadas respecto a dicho origen. Sin embargo, lo que necesitamos conocer son las coordenadas respecto al origen de WebGL que se corresponde con la esquina inferior izquierda del canvas. El listado 9.1 muestra cómo obtener las coordendas correctas para ser utilizadas en WebGL.

Tras el clic producido por el usuario, hay que dibujar la escena utilizando colores planos, tal y como se muestra en la figura 9.1(a). Por supuesto, este resultado

Listado 9.1: Conversión de coordenadas para ser utilizadas en WebGL

```
rectangle    = event.target.getBoundingClientRect();
x_in_canvas = (event.clientX - rectangle.left);
y_in_canvas = (rectangle.bottom - event.clientY);
```

intermedio no se hace visible al usuario, simplemente se dibuja para después acceder al *framebuffer* y conocer el color del píxel. El listado 9.2 muestra la operación de acceso al color de un píxel que se realiza mediante la función *gl.readPixels*. La variable *pixels* contendrá en consecuencia el valor de color buscado. Después, ya solo resta comparar el valor leído con los utilizados al dibujar los objetos, borrar el *framebuffer* con la orden *gl.clear* y dibujar la escena, esta vez con las técnicas habituales, ya que este nuevo dibujado sí que será el que finalmente termine mostrándose al usuario.



(a) Colores planos (b) Aspecto final

Figura 9.1: Las dos escenas pintadas para la selección de objetos

Listado 9.2: Acceso al color de un píxel en el *framebuffer*

```
var pixels = new Uint8Array(4);
gl.readPixels(x_in_canvas, y_in_canvas, 1, 1, gl.RGBA,
              gl.UNSIGNED_BYTE, pixels);
```

Ejercicios

► **9.1** Ejecuta el ejemplo *c09/seleccion/seleccion.html* que implementa el método de selección descrito en esta sección. Comprueba su funcionamiento. Examina la atención del evento, la obtención de las coordenadas del píxel seleccionado y cómo se determina la primitiva seleccionada en base al color leído. Prueba a añadir un objeto más a la escena, por ejemplo una esfera de color azul, y que este sea también seleccionable.

9.1.1. Utiliza un FBO

El método anterior dibuja la escena de colores planos para averiguar el objeto seleccionado a partir del momento en que el usuario realiza la selección. Si la escena muestra objetos estáticos, esto no entraña mayor problema. Sin embargo, si los objetos seleccionables están animados, puede ocurrir que la selección del usuario y lo leído en el *framebuffer* no coincida debido al paso del tiempo y la correspondiente actualización de los objetos de la escena. Una manera de evitarlo es tener la escena de colores planos almacenada, de manera que al realizar el clic sea posible realizar la consulta sobre la escena que se está mostrando al usuario y no sobre una nueva.

Esto se puede conseguir utilizando un nuevo objeto *framebuffer* (FBO) para dibujar en él la escena con colores planos. La condición es que este FBO almacene color y profundidad y que tenga el mismo tamaño que el canvas. Cuando el usuario realiza la selección, el FBO ya contiene la imagen dibujada, puesto que en él se ha dibujado la misma escena que el usuario está viendo, solo que con colores planos. De esta manera, lo que se consigue es poder realizar la consulta sobre lo ya dibujado. El listado 9.3 muestra la operación de acceso al color de un píxel. Se puede observar que la única variación es que antes de llamar a la función *gl.readPixels* se activa el FBO creado a próposito y después se vuelve a establecer el *framebuffer* por defecto.

Listado 9.3: Acceso al color de un píxel en el FBO

```
gl.bindFramebuffer(gl.FRAMEBUFFER, myFbo); // selecciona el FBO
var pixels = new Uint8Array(4);
gl.readPixels(x_in_canvas, y_in_canvas, 1, 1, gl.RGBA,
              gl.UNSIGNED_BYTE, pixels);
gl.bindFramebuffer(gl.FRAMEBUFFER, null); // framebuffer normal
```

Ejercicios

► **9.2** Ejecuta el ejemplo *c09/seleccion/seleccionFbo.html* que implementa el método de selección que utiliza un objeto *framebuffer*. Comprueba su funcionamiento. Examina cómo se determina la primitiva seleccionada en base al color leído a partir de la escena ya dibujada. ¿De qué tamaño es el FBO?, ¿y el canvas?, ¿qué ocurriría si no coincidiesen ambos tamaños?

9.2. Animación

Realizar animación a través de *shaders* puede resultar muy sencillo. Solo necesitamos una variable uniforme en el *shader* y que esta se actualice desde la aplicación con el paso del tiempo. En el *shader* se utilizará dicha variable para modificar cualquiera de las propiedades 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; también modificar parámetros de las fuentes de luz haciendo que, por ejemplo, la intensidad de la luz aumente o decaiga de forma gradual o que los objetos cambien su color. Por supuesto, también podemos modificar la matriz de transformación del modelo cambiando la posición, el tamaño o la orientación de algún objeto de la escena o, por qué no, añadir dos materiales a un objeto y hacer que un objeto cambie de material con una simple interpolación lineal.

9.2.1. Eventos de tiempo

JAVASCRIPT 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 generar la animación. La función es la siguiente:

```
myVar = setTimeout(updateStuff, 40);
```

El valor numérico indica el número de milisegundos que han de transcurrir para que se llame a la función *updateStuff*. Una vez transcurrido dicho tiempo, esa función se ejecutará lo antes posible. Si se desea que la función se ejecute otra vez al cabo de un nuevo periodo de tiempo, ella misma puede establecerlo llamando a la función *setTimeout* antes de finalizar. Otra alternativa es utilizar la siguiente orden, que produce la ejecución de *updateStuff* cada 40 ms:

```
myVar = setInterval(updateStuff, 40);
```

Si por contra lo que se desea es que se suspenda la ejecución:

```
clearInterval(myVar);
```

9.2.2. Encendido / apagado

Un efecto muy simple de animación consiste en que una propiedad tome dos valores diferentes que van alternándose a lo largo del tiempo, como podrían ser la simulación del parpadeo de una luz al encenderse o el mal funcionamiento de un tubo fluorescente. En el *shader* necesitamos una variable para indicar el estado, es decir, si encendido o apagado (variable *modeOnOff* en el listado 9.4). En la aplicación es habitual que dicha variable esté gobernada por un simple proceso aleatorio. Por ejemplo, la función *setFlickering* en el listado 9.5 recibe un valor como parámetro que se compara con un número aleatorio. De esta manera, se puede controlar la preferencia hacia uno de los dos estados posibles.

Listado 9.4: *Shader* para encendido / apagado

```
// Shader de fragmentos
...
uniform bool modeOnOff; // almacena el estado
...
void main() {
  ...
  gl_FragColor = (modeOnOff == false) ?
                 vec4( phong(n,L,V), 1.0):
                 vec4( Material.Ka,  1.0);
}
```

Listado 9.5: Función que controla el encendido / apagado

```
function setFlickering(value) {

  if (Math.random()> value)  // Math.random en [0..1]
    gl.uniform1i (program.ModeOnOffIndex, false);
  else
    gl.uniform1i (program.ModeOnOffIndex, true);

}
```

Ejercicios

► **9.3** Ejecuta el ejemplo *c09/animacion/onOff.html* que implementa el método de animación de encendido / apagado. Comprueba su funcionamiento. Examina cómo se establece un valor diferente de parpadeo para cada primitiva, de manera que algunas se ven más tiempo encendidas y otras más tiempo apagadas. Realiza las modificaciones necesarias para que, en lugar de encendido / apagado, sea encendido / sobreiluminado.

9.2.3. Texturas

Modificar las coordenadas de textura con el tiempo es algo sencillo y de lo que se pueden obtener resultados muy interesantes. Por ejemplo, en el caso de ser una textura 2D, se podría simular un panel publicitario rotativo. En el caso de utilizar una textura procedural, por ejemplo el *shader* de nubes utilizado en el ejemplo *c08/ruido/nubes.html*, modificar las coordenadas de textura permite que su aspecto cambie suavemente con el tiempo. En cualquier caso, solo es necesario utilizar una variable que se incremente con el paso del tiempo (ver listado 9.6) y que a su vez se utilice para incrementar las coordenadas de textura en el *shader*, como se muestra en el listado 9.7. De esta manera, a cada instante de tiempo las coordenadas de textura de cada vértice son diferentes, produciéndose el efecto de animación.

Listado 9.6: Función que actualiza el desplazamiento de la textura con el tiempo

```
var texCoordsOffset = 0.0, Velocity = 0.01;

function updateTexCoordsOffset() {

  texCoordsOffset += Velocity;
  gl.uniform1f(program.texCoordsOffsetIndex, texCoordsOffset);

  requestAnimationFrame(drawScene);

}

function initWebGL() {
  ...
  setInterval(updateTexCoordsOffset,40);
  ...
}
```

Listado 9.7: *Shader* para actualizar las coordenadas de textura con el tiempo

```
// Shader de fragmentos
...
uniform float texCoordsOffset;
...
void main() {
  ...
  vec2 newTexCoords = texCoords;
  newTexCoords.s    += texCoordsOffset;
  gl_FragColor       = texture2D(myTexture, newTexCoords);
  ...
}
```

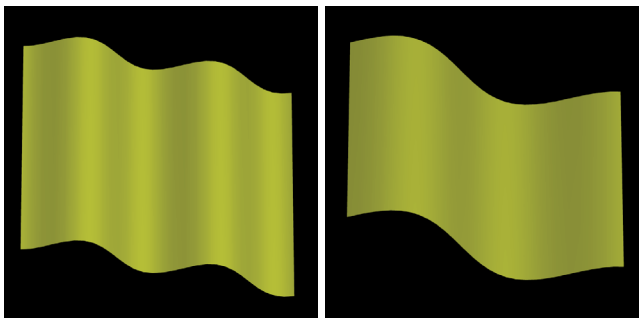
Ejercicios

► **9.4** Ejecuta el ejemplo *c09/animacion/nubes.html* que implementa el método de animación basado en modificar las coordenadas de textura con el tiempo. Comprueba su funcionamiento. Examina cómo se modifican las coordenadas de textura para acceder a la función de ruido. Ahora ponlo en práctica. Parte del ejemplo *c06/texturas2D.html* y modifícalo para que la textura se desplace sobre la superficie de las primitivas con el paso del tiempo.

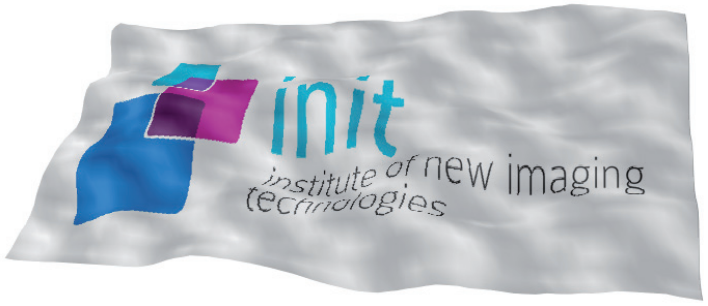
9.2.4. Desplazamiento

En la sección 6.3.2 se explicó el método que permite utilizar una textura como mapa de desplazamiento para que en tiempo de ejecución cada vértice se desplace a partir del valor leído del mapa. Ahora, lo que se persigue es animar la geometría,

haciendo que el desplazamiento de cada vértice no sea siempre el mismo, sino que cambie con el tiempo. La figura 9.2(a) muestra dos ejemplos en los que en el *shader* de vértices se ha utilizado la función seno con diferente amplitud de onda para calcular el desplazamiento en función del tiempo transcurrido. En la figura 9.2(b) se muestra el resultado de utilizar una función de ruido para determinar el desplazamiento. El valor de tiempo se utiliza como parámetro de entrada de la función de ruido, obteniéndose el efecto de una bandera al viento. El ejemplo *c09/animacion/vd.html* implementa este método de animación. Ejecútalo y experimenta con él para comprenderlo mejor.



(a) Desplazamiento producido por la función seno



(b) Desplazamiento producido por la función de ruido

Figura 9.2: Objetos animados con la técnica de desplazamiento

9.3. 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 espray. 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, transparencia, 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 9.3 y 9.4 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 diferentes. 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.

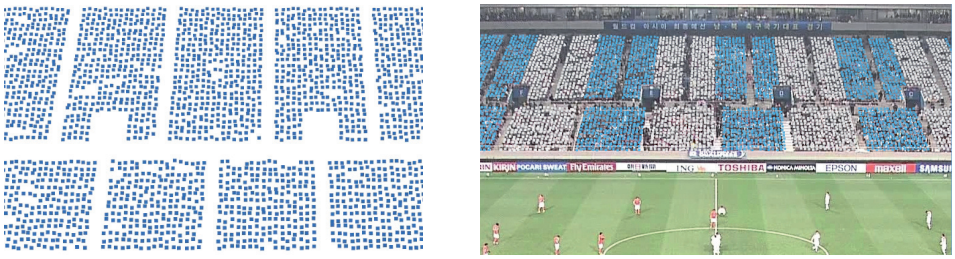


Figura 9.3: Animación de un mosaico implementado como sistema de partículas

Figura 9.4: 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, en el *shader* de vértices se determina el estado de la partícula, es decir, si aún no ha nacido, si está viva o si por el contrario 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. Para simplificar el dibujado de un sistema de partículas se asume que las partículas no colisionan entre sí, no reflejan luz y no producen sombras sobre otras partículas.

Un ejemplo de creación de un sistema de partículas se muestra en el listado 9.8. En concreto se crean diez mil partículas; a cada partícula se le asigna una velocidad y una posición a lo largo del eje X, ambas aleatorias, y un valor de nacimiento. Esta información se almacena en el vector *particlesInfo*, el cual se transfiere a un *buffer object*.

Listado 9.8: Cortina de partículas

```
var numParticles    = 10000;

function initParticleSystem() {

  var particlesInfo = [];

  for (var i= 0; i < numParticles; i++) {

    // velocidad
    var alpha    = Math.random();
    var velocity = (0.1 * alpha) + (0.5 * (1.0 - alpha));

    // posición
    var x = Math.random();
    var y = velocity;
    var z = 0.0;

    particlesInfo[i * 4 + 0] = x;
    particlesInfo[i * 4 + 1] = y;
    particlesInfo[i * 4 + 2] = z;
    particlesInfo[i * 4 + 3] = i * 0.00075; // nacimiento
  }

  program.idBufferVertices = gl.createBuffer ();
  gl.bindBuffer (gl.ARRAY_BUFFER, program.idBufferVertices);
  gl.bufferData (gl.ARRAY_BUFFER,
                 new Float32Array(particlesData),
                 gl.STATIC_DRAW);
}
```

El listado 9.9 muestra cómo se ordena el dibujado del sistema de partículas. En este ejemplo, cada partícula consta de dos atributos, posición e instante de nacimiento, y el sistema se dibuja como una colección de puntos.

Listado 9.9: Dibujado del sistema de partículas

```
function drawParticleSystem() {

  gl.bindBuffer (gl.ARRAY_BUFFER, program.idBufferVertices);
  gl.vertexAttribPointer (program.vertexPositionAttribute,
                          3, gl.FLOAT, false, 4*4,   0);
  gl.vertexAttribPointer (program.vertexStartAttribute,
                          1, gl.FLOAT, false, 4*4,   3*4);

  gl.drawArrays (gl.POINTS, 0, numParticles);

}
```

Por útltimo, en el *shader* de vértices se comprueba si la partícula ha nacido y, si es así, se calcula su posición a partir del valor de posición X y el valor de velocidad almacenado en la posición Y, junto con el valor del tiempo transcurrido. El listado 9.10 muestra este último paso. La figura 9.5 muestra dos ejemplos en los que úni-

camente cambia el tamaño del punto. El ejemplo *c09/animacion/particulas.html* implementa este método de animación. Ejecútalo y experimenta con él para comprenderlo mejor.

Listado 9.10: *Shader* de vértices para el sistema de partículas

```
...
attribute vec3  VertexPosition;
attribute float VertexStart;

void main()  {

  vec3 pos = vec3(0.0);

  if (Time > VertexStart) {       // si ha nacido
    float t = Time - VertexStart;
    if (t < 2.0) {                // si aún vive
      pos.x = VertexPosition.x;
      pos.y = VertexPosition.y * t;
      alpha = 1.0 - t / 2.0;
    }
  }

  vec4 ecPosition = modelViewMatrix * vec4(pos,1.0);

  gl_Position     = projectionMatrix * ecPosition;
  gl_PointSize    = 6.0;

}
```

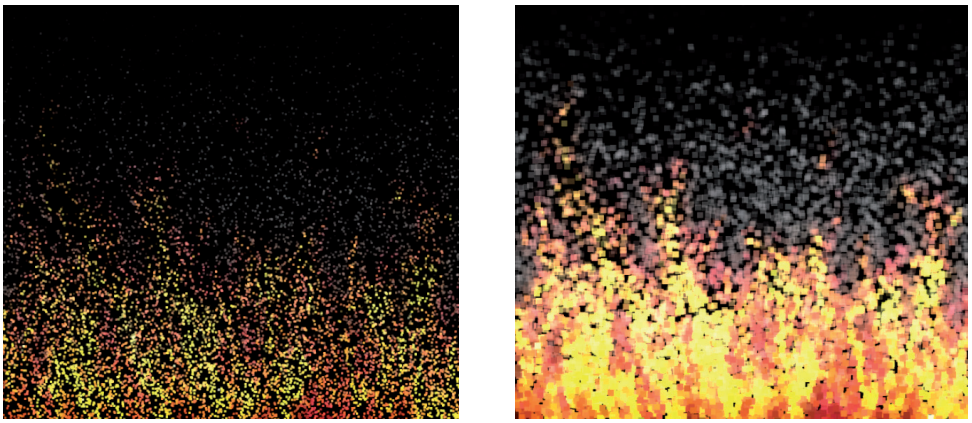


Figura 9.5: Ejemplo de sistema de partículas dibujado con tamaños de punto diferentes

Capítulo 10

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 se puede utilizar OpenGL 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 10.1).

Figura 10.1: 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

10.1. Apariencia visual

10.1.1. 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 10.2). Este problema es todavía más fácil de percibir, y también mucho más molesto, si los objetos están en movimiento.

Figura 10.2: 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 como una opción.

Supersampling

El método de *supersampling* se basa en tomar más muestras por cada píxel. De esta manera, el valor final de un píxel p se obtiene como resultado de la combinación de todas sus muestras. Hay que definir un patrón de muestreo y también se puede asignar un peso diferente a cada muestra.

$$p(x,y) = \sum_{i=1}^{n} w_i c(i,x,y) \tag{10.1}$$

La figura 10.3 muestra un ejemplo del funcionamiento del método donde, en lugar de utilizar una única muestra, se utilizan cuatro. En ese ejemplo, dos de las muestras quedan cubiertas por la proyección de la primitiva gráfica y el color final del píxel es el valor medio ponderado de los valores obtenidos para las cuatro muestras realizadas.

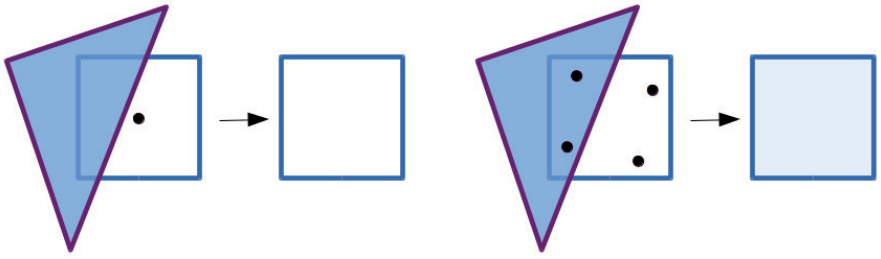


Figura 10.3: Ejemplo de funcionamiento del *supersampling*

La implementación más popular de este método se conoce con el nombre de *full scene anti-aliasing*, FSAA. Al utilizar esta técnica es necesario disponer de un *framebuffer* cuyo tamaño sea n veces mayor, donde n es el número de muestras, ya no solo para almacenar el color, sino también por ejemplo para guardar la profundidad de cada muestra. Este método procesa cada muestra de manera independiente,

por lo que es bastante costoso, dado que el número de píxeles se multiplica fácilmente por cuatro, ocho o incluso dieciséis.

Multisampling

El método conocido por *multi-sampling anti-aliasing*, MSAA, se basa en muestrear cada píxel n veces para averiguar el porcentaje del píxel cubierto por la primitiva. El *shader* de fragmentos solo se ejecuta una vez por fragmento, a diferencia del método *full scene anti-aliasing*, donde dicho *shader* se ejecutaba para cada muestra. OpenGL implementa este método y el programador solo necesita habilitarlo si lo desea. Sin embargo, WebGL 1.0 no lo porporciona, aunque algunos navegadores sí que lo hacen de forma experimental.

10.1.2. Corrección gamma

Los monitores no proporcionan una respuesta lineal respecto a los valores de intensidad de los píxeles. Esto produce que veamos las imágenes un poco más oscuras o no tan brillantes como realmente deberían observarse. En ciertas aplicaciones, es habitual que se ofrezca como opción al usuario poder realizar este ajuste de manera manual y así corregir el problema. En la figura 10.4, la curva CRT gamma muestra la respuesta del monitor para cada valor de intensidad de un píxel. La curva corrección gamma representa la intensidad del píxel necesaria para que el monitor presente la respuesta lineal, representada por la línea recta.

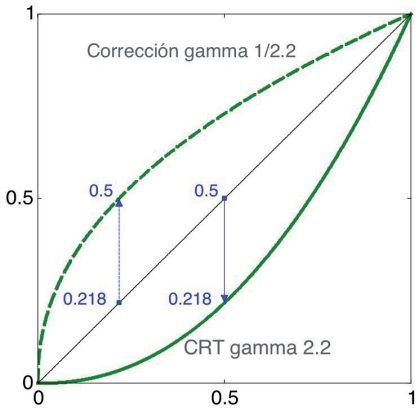


Figura 10.4: Esquema de funcionamiento de la correción gamma

Si la intensidad percibida es proporcional a la intensidad del píxel elevado a γ, $P = I^{\gamma}$, por lo que la correción gamma consiste en contrarrestar este efecto así:

$$P = (I^{\frac{1}{\gamma}})^{\gamma} \tag{10.2}$$

Esta operación se implementaría en el *shader* de fragmentos, tal y como figura en el listado 10.1 (ver ejemplo *c10/gamma.html*). La figura 10.6 muestra dos resultados obtenidos con valores de gamma $= 1{,}0$ y $2{,}2$.

Listado 10.1: *Shader* de fragmentos para la correción *gamma*

```
...
uniform float Gamma;

void main() {
  ...
  vec3  myColor     = phong(n,L,V);
  float gammaFactor = 1.0 / Gamma;

  myColor.r         = pow( myColor.r, gammaFactor);
  myColor.g         = pow( myColor.g, gammaFactor);
  myColor.b         = pow( myColor.b, gammaFactor);

  gl_FragColor      = vec4( myColor, 1.0);

}
```

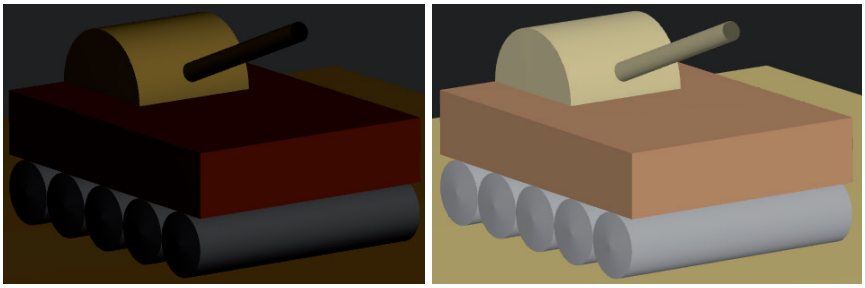


Figura 10.5: Ejemplos de corrección gamma: 1.0 (izquierda) y 2.2 (derecha)

10.2. Postproceso de imagen

En esta sección se consideran algunas técnicas de tratamiento de imágenes que se realizan a modo de postproceso del resultado de síntesis. Por ejemplo, las imágenes que se muestran en la figura 10.6 son el resultado del mismo proceso de síntesis y la diferencia es que, una vez obtenido el color del fragmento, se ha realizado alguna operación que se aplica por igual a todos los fragmentos de la imagen.

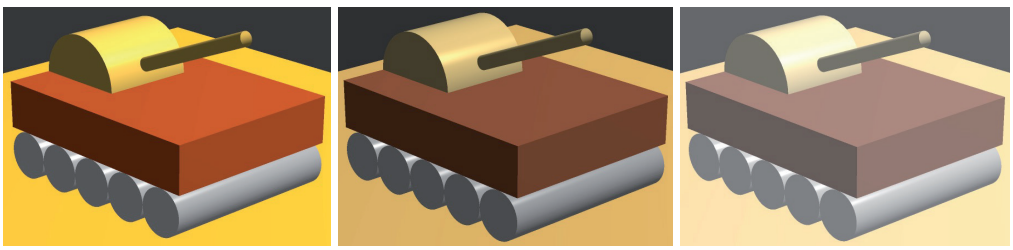


Figura 10.6: Ejemplos de postproceso de imagen

10.2.1. Brillo

La modificación del brillo de una imagen es un efecto muy sencillo. Consiste en escalar el color de cada fragmento por un valor *Brillo*. Si dicho valor es 1, la imagen no se altera, si es mayor que 1 se aumentará el brillo, y si es menor se disminuirá (ver figura 10.7). El listado 10.2 muestra el código que habría que incluir en el *shader* de fragmentos para modificar el brillo (ver ejemplo *c10/postproceso.html*).

Listado 10.2: Modificación del brillo de una imagen

```
gl_FragColor = vec4(miColor * Brillo, 1.0);
```

Figura 10.7: Ejemplos de modificación del brillo de la imagen con factores de escala 0,9, 1,2 y 1,5

10.2.2. Contraste

La alteración del contraste de la imagen se obtiene como resultado de mezclar dos colores, uno es el color obtenido como color del fragmento y el otro es el valor de luminancia media (ver figura 10.8). La variable *Contraste* se utiliza para dar más peso a un valor u otro (ver el listado 10.3 y el ejemplo *c10/postproceso.html*).

Listado 10.3: Modificación del contraste de una imagen

```
vec3 LuminanciaMedia = vec3(0.5, 0.5, 0.5);
gl_FragColor = vec4(mix(LuminanciaMedia, miColor, Contraste),1.0);
```

10.2.3. Saturación

La saturación es una mezcla del color del fragmento con un valor de intensidad específico de cada píxel (ver figura 10.9). Observa en el listado 10.4 las operaciones habituales para modificar la saturación que se pueden encontrar también en el ejemplo *c10/postproceso.html*.

Figura 10.8: Ejemplos de modificación del contraste de la imagen: 0,5, 0,75 y 1,0

Figura 10.9: Ejemplos de modificación de la saturación de la imagen: 0,2, 0,5 y 0,8

Listado 10.4: Modificación de la saturación de la imagen

```
vec3 lumCoef    = vec3(0.2125, 0.7154, 0.0721);
vec3 Intensidad = vec3(dot(miColor, lumCoef));
gl_FragColor    = vec4(mix(Intensidad, miColor, Saturacion), 1.0);
```

10.2.4. Negativo

Otro ejemplo muy sencillo es la obtención del negativo de una imagen (ver imagen 10.10). Únicamente hay que asignar como color final del fragmento el resultado de restarle a 1 su valor de color original. En el listado 10.5 se muestra el código correspondiente al cálculo del negativo del fragmento.

Listado 10.5: Negativo del fragmento

```
gl_FragColor = vec4(1.0 - miColor, 1.0);
```

10.2.5. Escala de grises

También es muy fácil la obtención de la imagen en escala de grises (ver imagen 10.11). Únicamente hay que asignar como color final del fragmento el resultado de

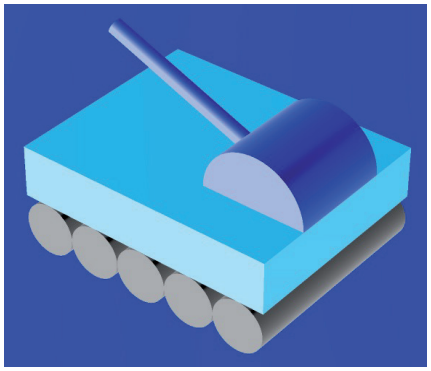


Figura 10.10: Ejemplo del resultado del negativo de la imagen

la media de sus tres componentes. En el listado 10.6 se muestra el código correspondiente al cálculo del color del fragmento.

Listado 10.6: Cálculo de la imagen en escala de grises

```
float media  = (miColor.r + miColor.g + miColor.b) / 3.0;
gl_FragColor = vec4(vec3(media), 1.0);
```

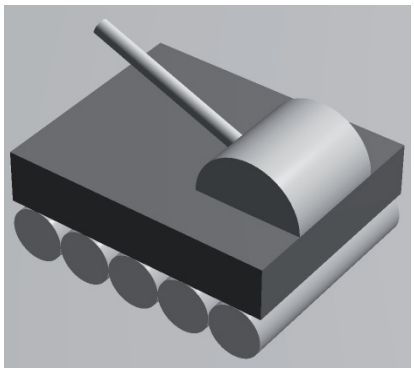


Figura 10.11: Ejemplo del resultado de la imagen en escala de grises

10.2.6. 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 denomina filtro de convolución. Lo que la operación de convolución realice depende de los valores del filtro. Para un filtro de dimensión $m \times n$ la operación es:

$$Res(x,y) = \sum_{j=0}^{n-1} \sum_{i=0}^{m-1} Img(x+(i-\frac{m-1}{2}), y+(j-\frac{n-1}{2})) \cdot Filtro(i,j) \quad (10.3)$$

Realizar esta operación con WebGL requiere que la imagen sintética se genere en primer lugar y se almacene después como textura para que en un segundo dibujado se pueda aplicar la operación de convolución sobre la imagen ya generada. 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.

Las operaciones más habituales son el *blurring*, el *sharpening* y la detección de bordes, entre otras. La tabla 10.1 muestra ejemplos de filtros de suavizado o *blurring*, nitidez o *sharpening* y detección de bordes (ver figura 10.12). El ejemplo *c10/bordes.html* incluye una implementación de este último filtro.

1	1	1
1	1	1
1	1	1

(a)

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

(b)

-1	-1	-1
-1	8	-1
-1	-1	-1

(c)

Tabla 10.1: Filtros de convolución: (a) suavizado, (b) nitidez y (c) detección de bordes

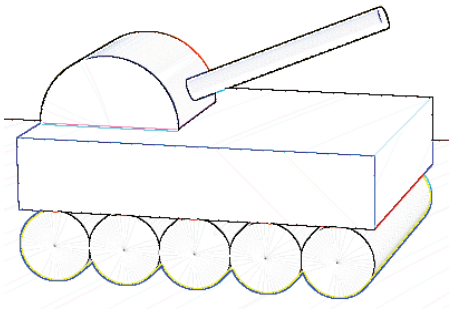


Figura 10.12: Ejemplo de resultado de la operación de convolución con el filtro de detección de bordes

10.3. 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 polígono sobre el que pegar la textura.
3. Escribir un *shader* de vértices que realice la operación geométrica deseada.

El paso 1 ha sido descrito en el capítulo 6. Para realizar el paso 2, un simple cuadrado de lado unidad es suficiente, no importa si la imagen no tiene esta proporción. En el paso 3, el *shader* de vértices debe transformar los vértices del

rectángulo de acuerdo a la transformación geométrica requerida. Por ejemplo, para ampliar la imagen solo 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 polígono de acuerdo a la proporción de la imagen.

Ejercicios

► **10.1** 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 *shader* de vértices 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 10.13).

Figura 10.13: *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 como resultado una 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.