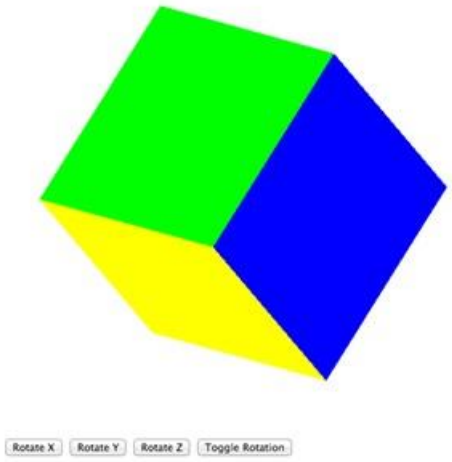
Introducción a WebGL

Dra. Irene Olaya Ayaquica Martínez
Facultad de Ciencias de la Computación, BUAP
irene.ayaquica@correo.buap.mx

Ejemplos



rotating cube with buttons

cube with lighting

texture mapped cube

¿Qué es OpenGL?

- OpenGL es una interfaz de programación de aplicaciones de renderizado de gráficos por computadora (API).
  - Con ella, se pueden generar imágenes en color de alta calidad al renderizar con primitivas geométricas y de imagen.
  - Constituye la base de muchas aplicaciones interactivas que incluyen gráficos 3D.
  - Al usar OpenGL, la parte gráfica de su aplicación puede ser:
    - Independiente del sistema operativo
    - Independiente del sistema de ventanas

¿Qué es WebGL?

- WebGL es una implementación en JavaScript de OpenGL ES 2.0
  - se ejecuta en todos los navegadores recientes (Chrome, Firefox, IE, Safari)
    - independiente del sistema operativo
    - Independiente del sistema de ventanas
  - la aplicación se puede ubicar en un servidor remoto

¿Qué es WebGL?

- la representación se realiza dentro del navegador utilizando hardware local
- utiliza el elemento canvas de HTML5
- se integra con aplicaciones y paquetes web estándar
  - CSS
  - jQuery

¿Qué necesitamos conocer?

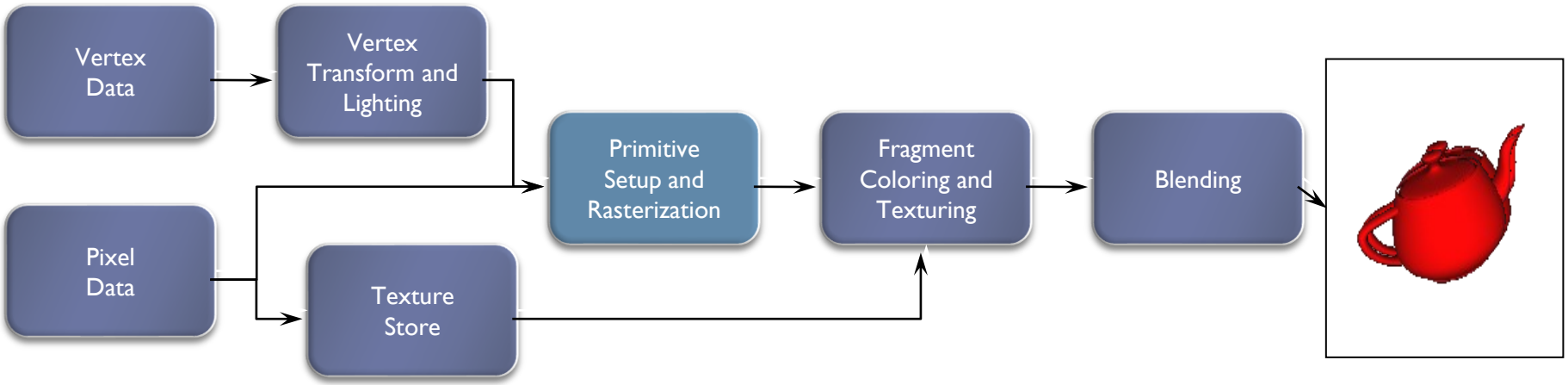
- Entorno y ejecución web
- Conceptos básicos de OpenGL moderno
- Arquitectura pipeline
- OpenGL basado en sombreadores
- Lenguaje de sombreado OpenGL (GLSL)
- JavaScript

Evolución del pipeline OpenGL

Al principio…

- OpenGL 1.0 fue lanzado el 1 de julio de 1994
- Este pipeline era completamente de función fija
- Las únicas operaciones disponibles fueron corregidas por la implementación
- El pipeline evolucionó pero se mantuvo basado en la operación de función fija a través de las versiones 1.1 a 2.0 de OpenGL (septiembre de 2004)

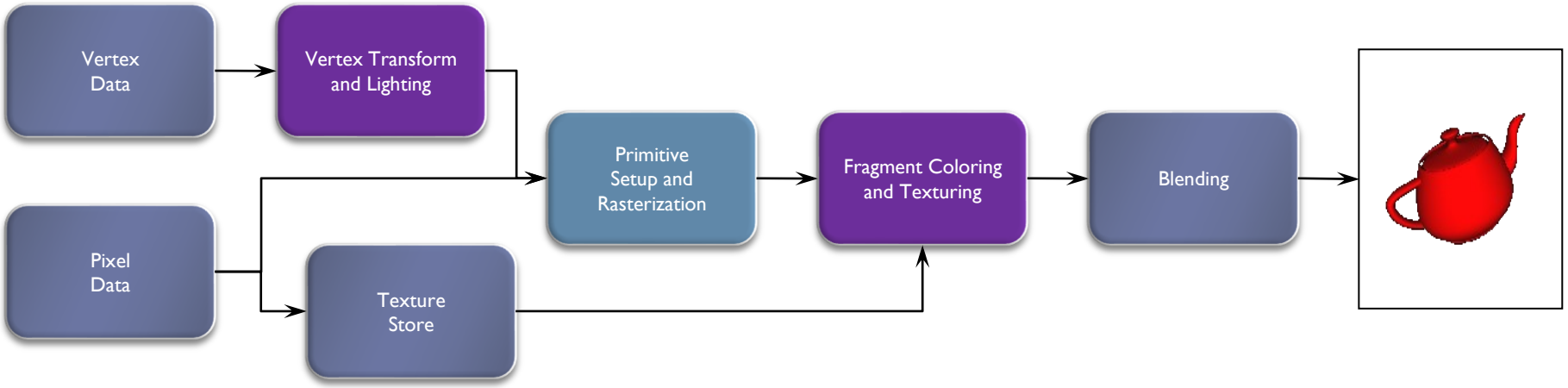
Al principio…



Inicios del pipeline programable

- OpenGL 2.0 (oficialmente) agregó sombreadores programables
  - *Vertex shading* aumentó la transformación de función fija y el escenario de iluminación
  - *Fragment shading* aumentó la etapa de coloración del fragmento
- Sin embargo, el pipeline de función fija todavía estaba disponible.

Inicios del pipeline programable



Un cambio evolutivo

- OpenGL 3.0 introdujo el modelo de obsolescencia
  - el método utilizado para eliminar funciones de OpenGL
- El proceso siguió siendo el mismo hasta OpenGL 3.1 (lanzado el 24 de marzo de 2009)
- Se introdujo un cambio en la forma en que se utilizan los contextos OpenGL.

Un cambio evolutivo

Context Type	Description
Full	Includes all features (including those marked deprecated) available in the current version of OpenGL
Forward Compatible	Includes all non-deprecated features (i.e., creates a context that would be similar to the next version of OpenGL)

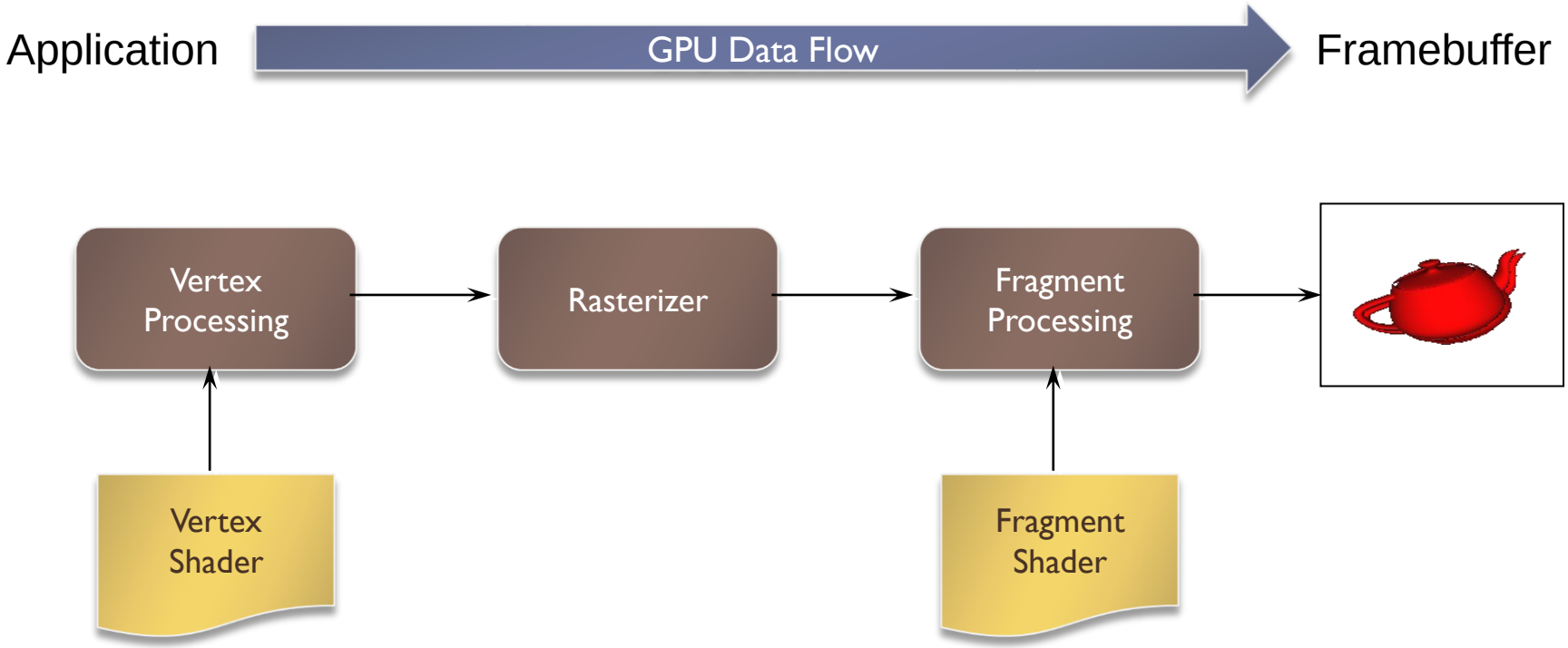
OpenGL ES y WebGL

- OpenGL ES 2.0
  - Diseñado para dispositivos integrados y portátiles como teléfonos móviles
  - Basado en OpenGL 3.1
  - Basado en sombreadores

- WebGL
  - Implementación en JavaScript de ES 2.0
  - Se ejecuta en los navegadores más recientes.

Desarrollo de aplicaciones en WebGL

Modelo pipeline simplificado



Programación WebGL en resumen

- Todos los programas WebGL deben hacer lo siguiente:
  - Configurar el canvas para renderizar
  - Generar datos en la aplicación
  - Crear programas de sombreado
  - Crear objetos de búfer y cargar datos en ellos
  - "Conectar" ubicaciones de datos con variables de sombreado
  - Renderizar

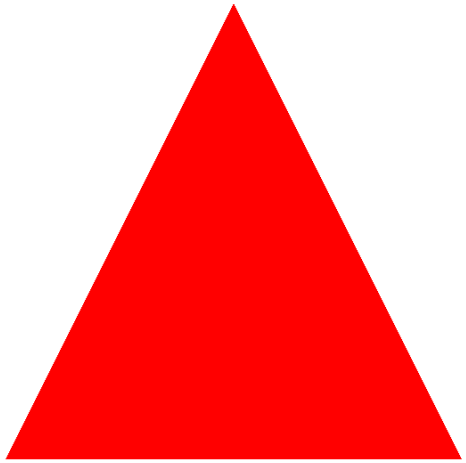
Marco de aplicación

- Las aplicaciones WebGL necesitan un lugar para renderizar
  - Elemento Canvas HTML5
- Podemos poner todo el código en un solo archivo HTML
- Preferimos poner la configuración en un archivo HTML y la aplicación en un archivo JavaScript separado
- El archivo HTML incluye sombreadores
- El archivo HTML se lee en utilidades y aplicaciones.

Un ejemplo simple

- Genera un triángulo rojo

- Tiene todos los elementos de una aplicación más compleja.
  - vertex shaders
  - fragment shaders
  - HTML canvas



triangle.html

```
<!DOCTYPE html>
<html>
<head>
<script id="vertex-shader" type="x-shader/x-vertex">
attribute vec4 vPosition;
void main()
{
    gl_Position = vPosition;
}
</script>
<script id="fragment-shader" type="x-shader/x-fragment">
precision mediump float;
void main()
{
    gl_FragColor = vec4( 1.0, 0.0, 0.0, 1.0 );
}
</script>
```

triangle.html

- <script type="text/javascript" src="../Common/webgl-utils.js"></script>
- <script type="text/javascript" src="../Common/initShaders.js"></script>
- <script type="text/javascript" src="triangle.js"></script>
- </head>
- <body>
- <canvas id="gl-canvas" width="512" height="512">
- Oops ... your browser doesn't support the HTML5 canvas element
- </canvas>
- </body>
- </html>

triangle.js

```
var gl;
var points;

window.onload = function init()
{
    var canvas = document.getElementById( "gl-canvas" );
    gl = WebGLUtils.setupWebGL( canvas );
    if ( !gl ) { alert( "WebGL isn't available" );
 }

var vertices = new Float32Array([-1, -1, 0, 1, 1, -1]);

 //  Configure WebGL

gl.viewport( 0, 0, canvas.width, canvas.height );
gl.clearColor( 1.0, 1.0, 1.0, 1.0 );
```

triangle.js

```
// Load shaders and initialize attribute buffers

var program = initShaders( gl, "vertex-shader", "fragment-shader" );

gl.useProgram( program );

// Load the data into the GPU

var bufferId = gl.createBuffer();
gl.bindBuffer( gl.ARRAY_BUFFER, bufferId );
gl.bufferData( gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW );
```

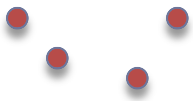
triangle.js

```
// Associate out shader variables with our data buffer

    var vPosition = gl.getAttribLocation( program, "vPosition" );
    gl.vertexAttribPointer( vPosition, 2, gl.FLOAT, false, 0, 0 );
    gl.enableVertexAttribArray( vPosition );
   render();
};

function render()
{
    gl.clear( gl.COLOR_BUFFER_BIT );
    gl.drawArrays( gl.TRIANGLES, 0, 3 );
}
```

Representar objetos geométricos

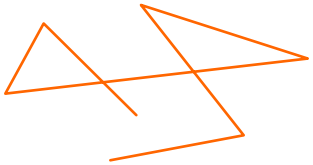
- Los objetos geométricos se representan mediante vértices
- Un vértice es una colección de atributos genéricos
  - coordenadas posicionales
  - colores
  - coordenadas de textura
  - cualquier otro dato asociado con ese punto en el espacio
- Posición almacenada en coordenadas homogéneas de 4 dimensiones
- Los datos de vértice deben almacenarse en objetos de búfer de vértice (VBO)

Primitivas geométricas de OpenGL

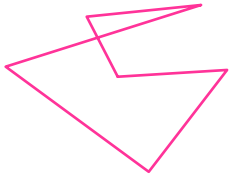


GL_POINTS

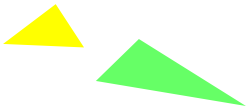
GL_LINES



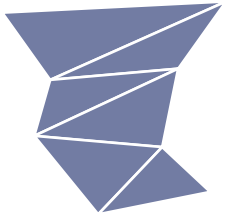
GL_LINE_STRIP



GL_LINE_LOOP



GL_TRIANGLES



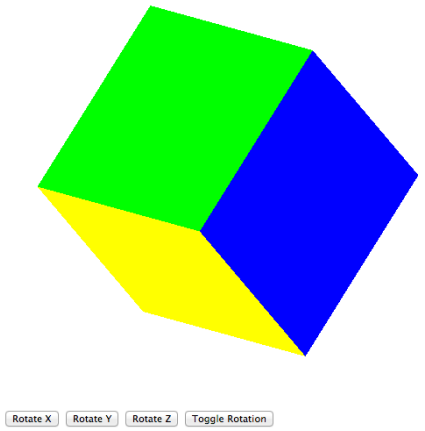
GL_TRIANGLE_STRIP

GL_TRIANGLE_FAN

Segundo ejemplo

- Renderiza un cubo con un color diferente para cada cara

- Nuestro ejemplo demuestra:
  - modelado de objetos simple
    - construir objetos 3D a partir de primitivas geométricas
    - Construir primitivas geométricas a partir de vértices
  - inicializando datos de vértice
  - organizar datos para renderizar
  - interactividad
  - animación



Inicializando los datos del cubo

- Construiremos cada cara de cubo a partir de triángulos individuales

- Necesitamos determinar cuánto almacenamiento se requiere
  - (6 caras) (2 triángulos / cara) (3 vértices / triángulo)
  - var numVertices = 36;

- Para simplificar la comunicación con GLSL, usaremos un paquete MV.js que contiene un objeto vec3 similar al tipo vec3 de GLSL

Inicializando los datos del cubo

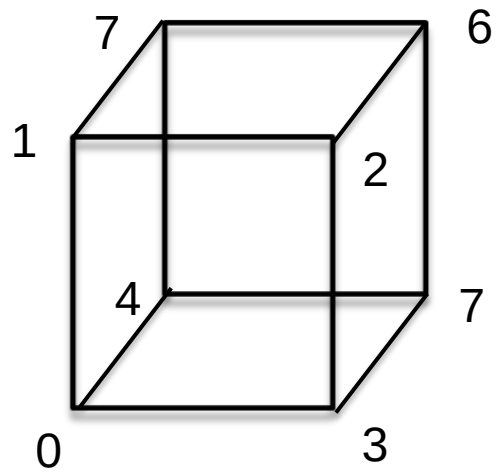
- Antes de que podamos inicializar nuestro VBO, debemos preparar los datos

- Nuestro cubo tiene dos atributos por vértice
  - posición
  - color

- Creamos dos matrices para contener los datos de VBO
  - var puntos = [];
  - var colores = [];

Datos del cubo

- Vértices de un cubo unitario centrado en el origen
  - lados alineados con los ejes
  - var vertices = [
    - vec4(-0.5, -0.5, 0.5, 1.0),
    - vec4(-0.5, 0.5, 0.5, 1.0),
    - vec4(0.5, 0.5, 0.5, 1.0),
    - vec4(0.5, -0.5, 0.5, 1.0),
    - vec4(-0.5, -0.5, -0.5, 1.0),
    - vec4(-0.5, 0.5, -0.5, 1.0),
    - vec4(0.5, 0.5, -0.5, 1.0),
    - vec4(0.5, -0.5, -0.5, 1.0)
  -];



Datos del cubo

- También configuraremos una variedad de colores RGBA
- Podemos usar vec3 o vec4 o simplemente arreglos JS

```
var vertexColors = [
        [ 0.0, 0.0, 0.0, 1.0 ],  // black
        [ 1.0, 0.0, 0.0, 1.0 ],  // red
        [ 1.0, 1.0, 0.0, 1.0 ],  // yellow
        [ 0.0, 1.0, 0.0, 1.0 ],  // green
        [ 0.0, 0.0, 1.0, 1.0 ],  // blue
        [ 1.0, 0.0, 1.0, 1.0 ],  // magenta
        [ 0.0, 1.0, 1.0, 1.0 ],  // cyan
        [ 1.0, 1.0, 1.0, 1.0 ]   // white
    ];
```

Arreglos en JS

- Un arreglo JS es un objeto con atributos y métodos como length, push () y pop ()
  - fundamentalmente diferente del arreglo de C
  - no se puede enviar directamente a las funciones de WebGL
  - usa la función flatten () para extraer datos del arreglo JS

  - gl.bufferData(gl.ARRAY_BUFFER, flatten(colors),
  - gl.STATIC_DRAW);

Generar una cara del cubo a partir de vértices

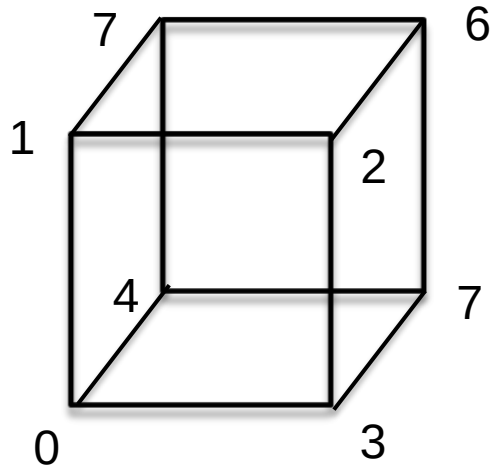
- Para simplificar la generación de la geometría, usamos una función de conveniencia quad ()
  - crea dos triángulos para cada cara y asigna colores a los vértices

```
function quad(a, b, c, d) {
var indices = [ a, b, c, a, c, d ];
    for ( var i = 0; i < indices.length; ++i ) {
        points.push( vertices[indices[i]] );
        // for vertex colors use
        //colors.push( vertexColors[indices[i]] );
        // for solid colored faces use
        colors.push(vertexColors[a]);
}
```

Generar una cara del cubo a partir de vértices

- Genera 12 triángulos para el cubo
  - 36 vértices con 36 colores

```
function colorCube() {
    quad( 1, 0, 3, 2 );
    quad( 2, 3, 7, 6 );
    quad( 3, 0, 4, 7 );
    quad( 6, 5, 1, 2 );
    quad( 4, 5, 6, 7 );
    quad( 5, 4, 0, 1 );
}
```



Almacenamiento de atributos de vértices

- Los datos de vértice deben almacenarse en un objeto de búfer de vértice (VBO)
- Para configurar una VBO debemos
  - crea un vacío llamando a gl.createBuffer ();
  - enlazar un VBO específico para la inicialización llamando a gl.bindBuffer (gl.ARRAY_BUFFER, vBuffer);
  - cargar datos en VBO usando (para nuestros puntos)
  - gl.bufferData (gl.ARRAY_BUFFER, flatten (points),
  - gl.STATIC_DRAW);

Código del arreglo de vértices

- Asociar variables de sombreado con arreglos de vértices
  - var cBuffer = gl.createBuffer();
  - gl.bindBuffer(gl.ARRAY_BUFFER, cBuffer);
  - gl.bufferData(gl.ARRAY_BUFFER, flatten(colors), gl.STATIC_DRAW);

  - var vColor = gl.getAttribLocation(program, "vColor");
  - gl.vertexAttribPointer(vColor, 4, gl.FLOAT, false, 0, 0);
  - gl.enableVertexAttribArray(vColor);

Código del arreglo de vértices

- var vBuffer = gl.createBuffer();
- gl.bindBuffer(gl.ARRAY_BUFFER, vBuffer);
- gl.bufferData(gl.ARRAY_BUFFER, flatten(points), gl.STATIC_DRAW);

- var vPosition = gl.getAttribLocation(program, "vPosition");
- gl.vertexAttribPointer(vPosition, 3, gl.FLOAT, false, 0, 0);
- gl.enableVertexAttribArray(vPosition);

Dibujar primitivas geométricas

- Para grupos contiguos de vértices, podemos usar la función de renderizado simple

```
function render()
{
    gl.clear( gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
    gl.drawArrays( gl.TRIANGLES, 0, numVertices );
    requestAnimFrame( render );
}
```

Dibujar primitivas geométricas

- gl.drawArrays inicializa el sombreador de vértices
- requestAnimationFrame necesario para volver a dibujar si algo está cambiando
- Tenga en cuenta que debemos borrar tanto el búfer de fotogramas como el búfer de profundidad
- Buffer de profundidad utilizado para la eliminación de superficies ocultas
- Habilitar HSR con gl.enable (gl.GL_DEPTH) en init ()

Shaders y GLSL

Sombreado de vértices

- Un sombreador que se ejecuta para cada vértice
  - Cada instanciación puede generar un vértice
  - Las salidas se pasan al rasterizador donde se interpolan y están disponibles para los sombreadores de fragmentos.
  - Posicionar la salida en coordenadas de clip

- Hay muchos efectos que podemos hacer en los sombreadores de vértices
  - Cambiar sistemas de coordenadas
  - Vértices en movimiento
  - Iluminación por vértice: campos de altura

Sombreado de fragmentos

- Un sombreador que se ejecuta para cada píxel "potencial"
  - los fragmentos aún deben pasar varias pruebas antes de llegar al framebuffer

- Hay muchos efectos que podemos hacer en los sombreadores de fragmentos.
  - Iluminación por fragmento
  - Mapeo de texturas y protuberancias
  - Mapas de entorno (reflexión)

GLSL

- Lenguaje de sombreado OpenGL
  - Lenguaje similar a C con algunas características de C ++
  - Matriz de 2-4 dimensiones y tipos de vectores
  - Tanto los sombreadores de vértices como los de fragmentos están escritos en GLSL
- Cada sombreador tiene un main ()

Tipos de datos en GLSL

- Tipos escalares: float, int, bool
- Tipos de vectores: vec2, vec3, vec4
- ivec2, ivec3, ivec4
- bvec2, bvec3, bvec4
- Tipos de matriz: mat2, mat3, mat4
- Muestreo de texturas: sampler1D, sampler2D, sampler3D,
- samplerCube
- Constructores de estilo C ++
- vec3 a = vec3 (1.0, 2.0, 3.0);

Operadores

- Operadores aritméticos y lógicos estándar C / C ++
- Operadores sobrecargados para operaciones matriciales y vectoriales

  - mat4 m;
  - vec4 a, b, c;

  - b = a*m;
  - c = m*a;

Calificadores

- attribute
  - atributos de vértice de la aplicación
- varying
  - copiar atributos de vértice y otras variables de los sombreadores de vértices a los sombreadores de fragmentos
  - los valores son interpolados por rasterizador
    - varying vec2 texCoord;
    - varying vec4 color;
- uniform
  - variable constante de sombreado de la aplicación
    - uniform float time;
    - uniform vec4 rotation;

Funciones

- Integradas
  - Aritmética: sqrt, power, abs
  - Trigonométrica: sin, asin
  - Gráfica: length, reflect
- Definidas por el usuario

Variables integradas

- gl_Position
  - (obligatorio) posición de salida del sombreador de vértices
- gl_FragColor
  - (obligatorio) color de salida del sombreador de fragmentos
- gl_FragCoord
  - posición del fragmento de entrada
- gl_FragDepth
  - valor de profundidad de entrada en el sombreador de fragmentos

Vertex shaders simple para el ejemplo del cubo

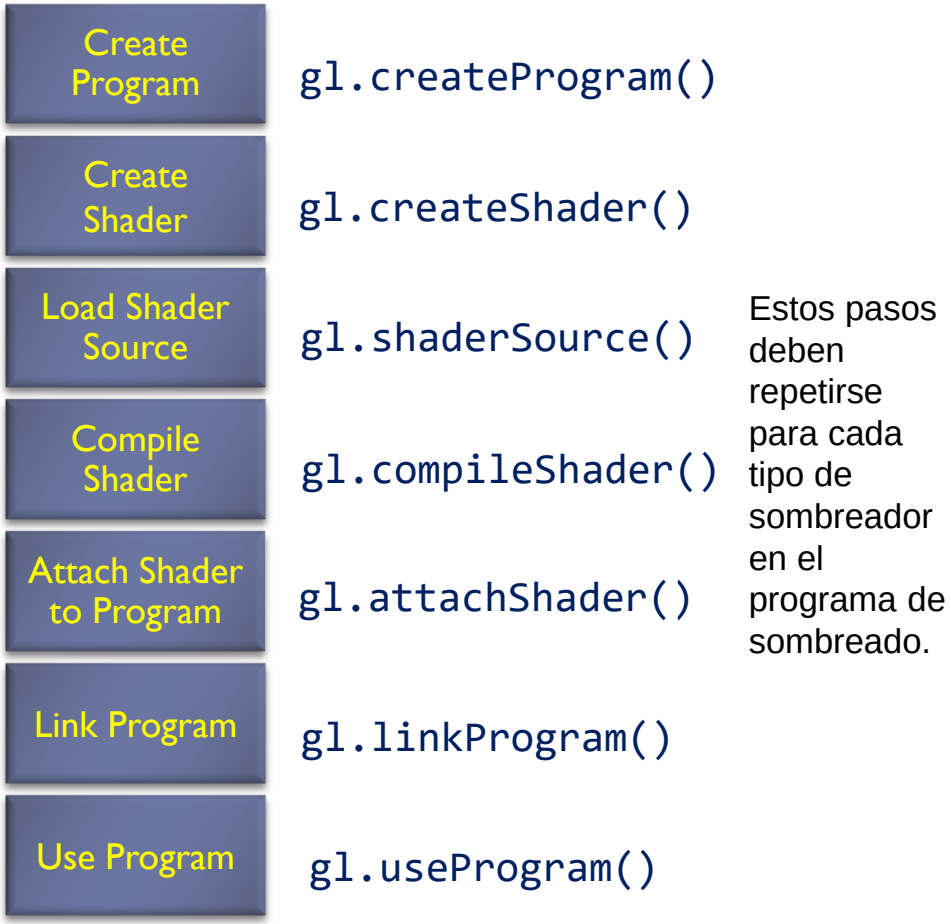
```
attribute  vec4 vPosition;
attribute  vec4 vColor;
varying vec4 fColor;

void main()
{
    fColor = vColor;
    gl_Position = vPosition;
}
```

Fragment shaders simple para el ejemplo del cubo

```
precision mediump float;

varying vec4 fColor;
void main()
{
    gl_FragColor = fColor;
}
```

Incorporar sombreadores a WebGL

- Los sombreadores deben compilarse y vincularse para formar un programa de sombreado ejecutable.
- WebGL proporciona el compilador y el enlazador.
- Un programa WebGL debe contener sombreadores de vértices y fragmentos.



Una forma más sencilla

- Usar una función creada para facilitar la carga de sus sombreadores.
- initShaders (vFile, fFile);
  - initShaders toma dos nombres de archivo
  - vFile ruta al archivo de sombreado de vértices
  - fFile para el archivo de sombreado de fragmentos

- Falla si los sombreadores no se compilan o el programa no se vincula

Asociación de datos y variables de sombreado

- Se necesita asociar una variable de sombreado con una fuente de datos OpenGL
  - atributos de sombreado de vértices → atributos de vértice de la aplicación
  - sombreadores uniformes → la aplicación proporcionó valores uniformes
- OpenGL relaciona las variables de sombreado con los índices para que la aplicación las configure
- Dos métodos para determinar la asociación variable / índice
  - especificar la asociación antes de la vinculación del programa
  - asociación de consultas después de la vinculación del programa

Determinación de ubicaciones después de la vinculación

- Asume que ya se conocen los nombres de las variables
- loc = gl.getAttribLocation(program,“name”);
- loc = gl.getUniformLocation(program,“name”);

Inicialización de valores de variable uniforme

- Variables uniformes
- gl.uniform4f(index, x, y, z, w);
- var transpose = gl.GL_TRUE;
- // Como programadores en C
- gl.uniformMatrix4fv(index, 3, transpose, mat);

Organización de la aplicación

- Archivo HTML:
  - contiene sombreadores
  - trae utilidades y archivo JS de aplicación
  - describe los elementos de la página: botones, menús
  - contiene el elemento canvas

Organización de la aplicación

- Archivo JS
  - init()
    - configura VBO
    - contiene oyentes para la interacción
    - configura las matrices de transformación necesarias
    - lee, compila y enlaza sombreadores
  - render()