# Práctica 2: Introducción a WebGL y tranformaciones en 3D

En esta práctica trabajaremos lo siguiente:

- La estructura básica de un programa que utiliza WebGL para mostrar una escena (geometría) en tiempo real
- Uso de transformaciones para mover objetos de la escena
- Manejo interactivo de la cámara, que requerirá uso de transformaciones y gestión de eventos

Pra ello utilizaremos WebGL, una API para renderizar gráficos (2D o 3D) en el navegador, soportada por la mayoría de navegadores modernos. Está basado en OpenGL, y premite utilizar la GPU para renderizar el contenido (siempre que esta soporte OpenGL).

## 1. Conceptos básicos de OpenGL

OpenGL es una API multiplataforma para generar y manipular gráficos e imágenes, tanto en 2D como en 3D. En realidad, no es una implementación, sino una especificación, esto es, dice cuál debe ser el resultado de cada una de las funciones, y la implementación concreta corre a cargo del desarrollador (usualmente, los fabricantes de tarjetas gráficas u otro hardware relacionado). Por tanto, puede haber distintas implementaciones, pero esto es generalmente transparente para el programador.

A la vez que el pipeline gráfico ha ido evolucionando desde el fixed function pipeline hacia una mayor flexibilidad, OpenGL ha ido adaptándose a esta evolución. Desde OpenGL 3 en adelante las versiones soportan este nuevo escenario en que se ha pasado de que el desarrollador provea la configuración de ciertas etapas del pipeline a que programe dichas etapas mediante shaders.

OpenGL funciona como una máquina de estados, esto es, en cada momento hay una serie de variables con valores que definen cómo se debe operar. El estado de OpenGL se denomina contexto, y las funciones se podrían dividir en: las que cambian el estado, o añaden/eliminan elementos del mismo, y las que leen/utilizan el estado; dentro de estas últimas se incluyen las que utilizan el estado actual para renderizar y generar la imagen.

## 2. WebGL y las herramientas de esta práctica

Las aplicaciones que utilizan WebGL usan Javascript para el código de control, y GLSL (OpenGL Shading Language, el lenguaje para programar shaders de OpenGL) para los shaders. Así, tendremos un fichero con código HTML que hará referencia a uno o varios ficheros en Javascript, así como a código en GLSL para los shaders.

WebGL es una API de bastante bajo nivel, y con frecuencia se utiliza en combinación con librerías o motores gráficos adicionales que facilitan la mayoría de las tareas (e.g., three.js, o PlayCanvas).

En esta práctica, cuyo objetivo es pedagógico, y no de eficiencia o desempeño, utilizaremos simplemente unas librerías sencillas para el manejo de matrices y otras funcionalidades base, así como un programa base con funcionalidad mínima del que partir y sobre el que construir, y cuyo objetivo es la simplicidad y la claridad.

Así pues, como herramientas sólo necesitaremos un editor para el código HTML y Javascript, y un navegador que soporte WebGL (la mayoría de los navegadores modernos lo hacen, si tenéis dudas sobre un cierto navegador, podéis usar este enlace: https://get.webgl.org/). Junto con este guión se os proporciona:

- Un código base sobre el que trabajaremos (practica_02_base.html y practica_02_base.js)
- Una serie de ficheros con código para funcionalidades básicas, como crear el contexto, preparar los shaders, o el manejo de matrices. [1]
    - Basta con incluir la carpeta `Common` proporcionada con el resto del código y referenciarla adecuadamente en el fichero HTML (ya está hecho en el que se os proporciona de base).

# 3. Estructura básica de un programa con WebGL

Consta, como mínimo, de un fichero HTML (que además incluye los shaders, que podrían alternativamente estar en ficheros aparte) y un fichero Javascript. Se muestra a continuación su estructura básica.

## 3.1. Fichero HTML

El fichero tiene varios scripts, para incluir los ficheros Javascript necesarios, y para los shaders (utilizaremos sólo dos shaders, vertex y fragment). Además, tiene un canvas, que es el elemento de HTML que permite renderizar con WebGL.

```
<!DOCTYPE html>
<html>
    <head>
    <title>Practica 02</title>
    <!-- Vertex shader -->
    <script id="vertex-shader" type="x-shader/x-vertex">
        // Declaraciones de variables
        // Código del shader
        void main()
        {
            gl_Position = ...
        }
    </script>
    <!-- Fragment shader -->
    <script id="fragment-shader" type="x-shader/x-fragment">
        // Declaraciones de variables
        // Código del shader
        void main()
        {
            gl_FragColor = ...
        }
    </script>
    !-- Ficheros Javascript externos.
    Los tres primeros son utilidades comunes.
    El cuarto es nuestro código Javascript que editaremos para la práctica.
    -->
    <script type="text/javascript" src="../Common/webgl-utils.js"></script>
    <script type="text/javascript" src="../Common/initShaders.js"></script>
    <script type="text/javascript" src="../Common/MVnew.js"></script>
    <script type="text/javascript" src="practica_02_base.js"></script>
    </head>
    <body>
        <!-- El canvas, que permite renderizar con WebGL -->
        <canvas id="gl-canvas" width="1024" height="512">
            Oops ... your browser doesn't support the HTML5 canvas element
        </canvas>
    </body>
</html>
```

[1] Fuente de dicho código: "Interactive Computer Graphics: A Top-Down Approach with WebGL", Ed Angel and Dave Shreiner. Ed. Addison-Wesley, 7th Edition, 2015.

 

## 3.2. Fichero Javascript

El fichero tiene una serie de variables globales, y dos funciones principales (y puede tener tantas funciones adicionales como se requiera), que son:

- `init()`: Su ejecución se asocia al evento `onload`, que ocurre al cargarse la ventana (`window.onload = init`). Prepara e inicializa todo lo necesario, incluyendo, entre otros:
  - Crear el contexto,
  - Configurarlo (e.g., si queremos que haya depth test o no),
  - Preparar los shaders (recordad que los shaders son programas independientes, y que por tanto hay que compilarlos y enlazarlos),
  - Preparar los datos de la escena, al menos su estado inicial (esto incluye geometría, cámara, etc.)
  - Llamar a la función que renderiza (render) - si la escena es estática, bastará con una sola llamada; si es dinámica (si hay animación), se recomienda llamarla mediante `requestAnimationFrame` (el código base proporcionado utiliza una versión multiplataforma, `requestAnimFrame`)
- `render()`: Se encarga de implementar aquello que se necesite en el bucle de render. Como mínimo, dibujará la escena. Si la escena es dinámica, también actualizará la posición o cualquier otra propiedad de los objetos de la escena o de la cámara como proceda.

## 3.3. Programa proporcionado: funcionamiento

En este punto, podemos abrir el fichero `practica_02_base.html` en el navegador para ver qué hace. El resultado debería ser algo similar a la Figura 1. Los tres ejes (x en verde, y en rojo, z en azul) están situados en el origen del sistema de coordenadas del mundo, y el cubo blanco en wireframe está centrado en el origen. Hay además un cubo sólido desplazado respecto al origen que rota sobre sí mismo, y otro cubo sólido que rota en torno al origen del mundo en el plano definido por los ejes x-z.

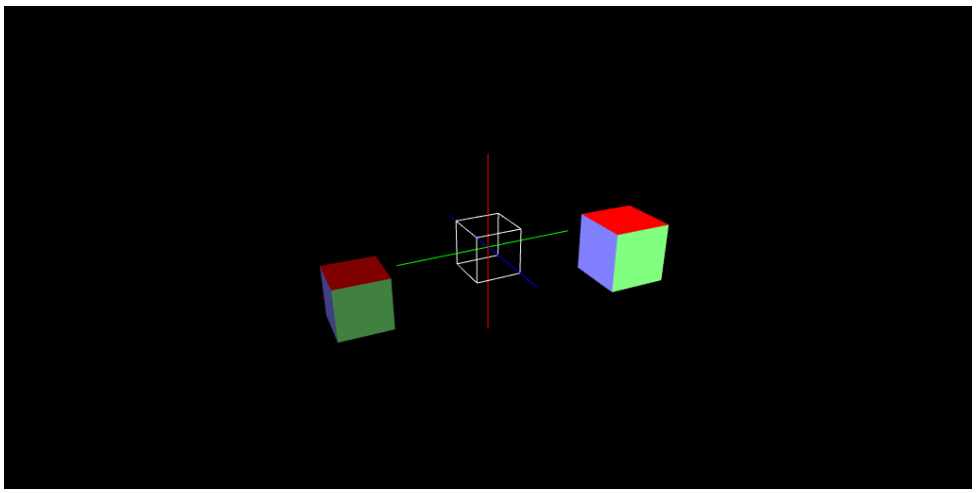

Figura 1: Screenshot de `practica_02_base.html`.

A continuación, inspeccionad los ficheros HTML y Javascript proporcionados (`practica_02_base.html/js`). Aunque el código está muy comentado, se proveen aquí explicaciones adicionales para cada una de las

 

partes - es recomendable consultar el código en paralelo a estas explicaciones, que ayudarán de cara a los cambios que habrá que realizar.

## 3.4. Fichero HTML (`practica_02_base.html`)

Lo más relevante para esta práctica es el código de los shaders. No es el objetivo principal de esta práctica trabajar con shaders, pero sí debemos entender qué está sucediendo a alto nivel. Como se ha visto en teoría, las entradas al shader pueden ser de distintos tipos, que en el código se indican con calificadores. En concreto aquí tenemos:

- `attribute`: Variables con un valor por vértice, y de sólo lectura, que se pasan como entrada al vertex shader
- `uniform`: Valor constante para el *draw call* en curso, para la primitiva que se está procesando; pueden estar tanto en el vertex como en el fragment shader.
- `varying`: Variable de salida en el *vertex shader*, y de entrada en el *fragment shader* (son por tanto datos cuyo valor se interpola entre el *vertex* y el *fragment shader*).

El funcionamiento de los shaders proporcionados es el siguiente:

- *Vertex shader*: Se trata de un shader muy sencillo que únicamente:
  - Transforma los vértices al espacio homogéneo: `gl_Position = projection * view * model * vec4(vPosition,1);`
  - Pasa el color de los vértices a la siguiente etapa: `varColor = vColor;`
- *Fragment shader*: También muy sencillo, obtiene el color de cada fragment como el producto: `gl_FragColor = varColor * colorMult;` siendo:
  - `varColor`: el color asignado a los vértices (e interpolado para los fragments).
  - `colorMult`: multiplicador (se puede ver como un peso o coeficiente aplicado al color), que es un `uniform`, y por tanto constante para cada primitiva renderizada, i.e., para cada objeto, en el caso del ejemplo.

## 3.5. Fichero HTML (`practica_02_base.js`)

Primero tenemos la definición de la escena, donde:

- Para cada objeto se especificará la posición de sus vértices, y el color de los mismos. Se definen una serie de vértices (`cubeVerts`), que luego se indexarán para crear la geometría de los distintos objetos presentes: `pointsAxes`, `pointsWireCube` y `pointsCube` contienen los vectores de posición de los vértices de los objetos correspondientes; `colorsAxes`, `colorsWireCube` y `colorsCube` hacen lo propio con los colores.
- Se crean variables para almacenar la información necesaria de los shaders (programInfo, en el ejemplo). Ésta incluye:
  - `program`: una combinación de un vertex y un fragment shader compilados
  - `uniformLocations` y `attribLocations`: serán variables que se usen para establecer la conexión entre las variables del código Javascript y las variables correspondientes del shader.

 

- Se crea una estructura que almacene los objetos a dibujar. Para cada objeto, contendrá cualquier información necesaria para renderizarlo. En nuestro ejemplo de partida, ésta incluye los *shaders* (`programInfo`), los vértices y su color, el tipo de primitiva, y los `uniforms` (matriz del modelo y multiplicador del color). Se puede añadir cualquier otra información que se considere necesaria.
- Variables para almacenar otra información necesaria (matriz de proyección, matriz de la vista, etc.).

A continuación, tenemos la definición de la función `init`, que se ejecuta al cargar la ventana:

- Crear y configurar el contexto
- Compilar de los shaders (usando `initShaders`)
- Establecer las localizaciones de atributos y uniforms en los shaders, y asignarlas a `programInfo`
- Establecer la cámara:
  - Se crean las matrices de proyección y de la vista iniciales (se podrán cambiar luego en la función de render si es necesario).
  - Se pasan dichas matrices a los shaders (mediante `uniformMatrix4fv`).
- Llamada a `requestAnimFrame(render)`, explicado en la sección anterior, para iniciar el renderizado

La función encargada de renderizar, render, es llamada en bucle hasta que se para la ejecución (al final de la misma hay una llamada a `requestAnimFrame(render)`). Tiene dos partes:

- Actualización de la escena. En el ejemplo, hay dos cubos que se mueven; por tanto, aquí se actualiza su posición cambiando su matriz del modelo. También se podría aquí, por ejemplo, actualizar la cámara, cambiando la matriz de la vista, o la de proyección.
- Renderizado. En el ejemplo, un bucle itera sobre los objetos a renderizar. Para cada uno, establece los shaders a utilizar (`useProgram`), pasa la información del objeto a los buffers y variables correspondientes (`setBuffersAndAttributes`), pasa los uniforms a los shaders (setUniforms), y da la orden de dibujar, especificando el tipo de primitiva (`drawArrays`).
  - `useProgram` y `drawArrays` son funciones propias de WebGL. `setBuffersAndAttributes` y `setUniforms` se han creado para abstraer y facilitar la comprensión, no es necesario editarlas para esta práctica (aunque no se impide hacerlo).

# 4. Trabajo a realizar por el alumno

El trabajo a realizar en esta práctica tiene dos partes.

## 4.1. Creación y animación de objetos

Se pide añadir a la base proporcionada una serie de cubos sólidos (no wireframe) que (a) roten en torno a sí mismos, y (b) orbiten en torno al origen del sistema de coordenadas del mundo.

Deberá haber un mínimo de 20 cubos, y cada uno tendrá su propia órbita (pueden colisionar entre ellos, esto es, solaparse, y no hay que reaccionar a dichas colisiones, simplemente se ignoran). Cada cubo debe tener un color único en todas sus caras (no distinto en cada cara, como los dos cubos sólidos del ejemplo, que están hechos así para ilustrar de forma clara su movimiento), y diferente al del resto de cubos.

A tener en cuenta:

 

- Los dos cubos que se mueven en el ejemplo base sirven de ayuda para la implementación. Para crear las matrices de transformación necesarias, utilizan funciones de `MVnew.js` (disponible en `Common`). Ahí encontraréis funciones para crear y operar con matrices y vectores.
- Recordad lo que vimos en clase de teoría sobre la combinación de transformaciones y el orden en que se aplican.

## 4.2. Movimiento interactivo de la cámara

Se pide añadir a lo anterior una cámara controlable mediante el ratón y el teclado. La cámara será al estilo de la que puede haber, por ejemplo, en un first person shooter (sin traslación en y, esto es, no hay saltos): la traslación será posible en dos ejes, y la rotación en otros dos (sin roll o rotación en torno al eje z, esto es, no se puede inclinar la cabeza hacia los lados).

Además, se podrá cambiar el modo de proyección (perspectiva/ortográfica), y el field of view (en la cámara perspectiva). En concreto, dicho control de la cámara debe ser como sigue:

- Teclas de dirección (`arriba/abajo/izq/dcha`): arriba/abajo controlarán la traslación de la cámara en z, e izq/dcha la traslación de la cámara en x (en ejes locales a la cámara).
- Ratón: controlará la rotación de la cámara, en concreto los ángulos denominados *pitch* (rotación en torno al eje x, controlado por la posición horizontal del ratón en la pantalla) y *yaw* (rotación en torno al eje y, controlado por la posición vertical del ratón en la pantalla). El movimiento del ratón sólo modificará la rotación mientras esté pulsado el ratón.
- Tecla `p`: selecciona proyección perspectiva (la que está por defecto).
- Tecla `o`: selecciona proyección ortográfica.
- Teclas `+`/`-`: incrementan/decrementan el *field of view* de la cámara (sólo en el modo de proyección perspectiva).

A tener en cuenta:

- Como muestra el código base proporcionado, la matriz de la vista se puede crear con la función `lookAt`, a la que se le proporciona, en espacio del mundo: el lugar donde está la cámara (`eye`), hacia dónde mira (`target`), y un vector que indica la dirección hacia arriba de la misma (`up`). A partir de ahí, si además se quiere mover la cámara, una forma de hacerlo es aplicar las matrices de transformación que corresponda. Al hacerlo, recordad de clase de teoría que mover la cámara en una dirección equivale a mover toda la escena en la dirección opuesta.
- Para la creación de la matriz de proyección basta con utilizar las funciones `perspective` u `ortho` con los parámetros adecuados.
- Para gestionar eventos de teclado o ratón os pueden ser útiles:
  - `EventTarget.addEventListener` asociado a la ventana (e.g., `window.addEventListener("keydown", keyPressedHandler);`).
  - `KeyboardEvent.code` para detectar la tecla pulsada.
  - `MouseEvent.clientX`, `MouseEvent.clientY` para obtener la posición del ratón.
- Se recomienda establecer un valor máximo y mínimo de *pitch* y *yaw* (+/-90º puede ser razonable). También se recomienda establecer un máximo y mínimo para el *field of view*.

 

# 5. Entrega

Se entregarán los ficheros .html y .js creados o editados para la resolución de la práctica (no hay que entregar los ficheros Javascript proporcionados en `Common`). El código añadido a la base proporcionada debe estar claramente indicado y apropiadamente comentado. Se entregará también un readme que contenga los nombres de los autores, el tiempo invertido por cada autor, y una descripción de los ficheros incluidos (si hay más aparte de los proporcionados). Si habéis tomado alguna decisión de diseño ante algún aspecto no especificado en el guión, indicadla también en el readme, de forma razonada. Todo ello se entregará comprimido en un único fichero .zip de nombre: `VID_practica02_GrupoXX.zip`, con XX el número de vuestro grupo.

No se utilizarán librerías adicionales, ni código de terceros adicional al proporcionado, para la resolución. En el caso de consultar alguna fuente para la resolución (más allá de la documentación de la API), se debe incluir en el *readme*.

La entrega se realizará vía Moodle, a través del enlace habilitado al efecto. En dicho enlace de Moodle se indica la fecha límite de entrega.

 