

Prof. José Ribelles
Departamento de Lenguajes y Sistemas Informáticos
Universitat Jaume I

En la práctica anterior conociste lo que va a ser tu entorno de trabajo durante el curso e incluso llegaste a hacer pequeñas modificaciones a los Shaders de vértices y fragmentos. Aunque lo que hiciste fuera nuevo para ti, es realmente en esta práctica donde se comienza a trabajar la materia propiamente dicha de esta asignatura. La Práctica 1 que comienzas justo ahora está ligada al Bloque 1 de las clases de teoría y, en consecuencia, vas a trabajar en lo mismo que se te ha explicado en clase, es decir, modelado poligonal, transformaciones geométricas, y transformaciones de proyección. Como ya se explicó en clase el día de la presentación, esta práctica forma parte del bloque fundamental de la asignatura. Te va a exigir un esfuerzo importante al mismo tiempo que te va a resultar muy poco agradecida, y es que no verás más que líneas, líneas de colores o, en el mejor de los casos, superficies coloreadas mediante colores planos.

El trabajo que has de realizar está organizado en cuatro partes para las que también dispones de cuatro sesiones presenciales de dos horas cada una de ellas. Se utilizará una sesión para cada parte.

Recuerda que:

- *El boletín es autoguiado, ve a tu ritmo, no tengas prisa por acabar.*
- *Realiza todos los ejercicios que se te indican y contesta a las preguntas que se te plantean.*
- *Entiende lo que estás haciendo y asimílalo, compréndelo.*
- *Aclara todas las dudas que te surjan.*

Parte I

Modelado Poligonal

WebGL admite modelos definidos mediante las dos estructuras de datos vistas en clase, es decir, la de vértices compartidos (ver Listado 1) y la de caras independientes (ver Listado 2). Antes de comenzar con los ejercicios, es necesario que repases **ambos tipos de estructuras** y que resuelvas las dudas.

Listado 1: Modelo poligonal de un cuadrado con la estructura de vértices compartidos

```
var unCuadradoVC = {           // estructura de vértices compartidos

  "vertices" : [               // vector de vértices, atributo de posición
    -0.5, -0.5, 0.0,          // v0
    0.5, -0.5, 0.0,           // v1
    0.5, 0.5, 0.0,            // v2
    -0.5, 0.5, 0.0],          // v3

  "indices" : [                // vector de índices
    0, 1, 2,                  // t0
    0, 2, 3],                 // t1

};
```

Listado 2: Modelo poligonal de un cuadrado con la estructura de caras independientes

```
var unCuadradoCI = {           // estructura de caras independientes

  "vertices" : [               // hay un único vector, atributo de posición
    -0.5, -0.5, 0.0,          // t0 v0
    0.5, -0.5, 0.0,           // t0 v1
    0.5, 0.5, 0.0,            // t0 v2
    -0.5, -0.5, 0.0,          // t1 v0
    0.5, 0.5, 0.0,            // t1 v2
    -0.5, 0.5, 0.0],          // t1 v3

};
```

En esta sesión nuestro objetivo es modelar el poliedro de [Johnson](#) número 10, al que vamos a llamar **j10** de forma abreviada (ver Figura 1). Procede de esta forma:

1. Enumera, comenzando por 0, los 9 vértices del poliedro j10 y obtén los 14 triángulos que lo forman.
Define todos los triángulos en sentido antihorario.
2. A partir del sistema de coordenadas que se muestra en la propia Figura 1, obtén las coordenadas de los 9 vértices del poliedro j10 teniendo en cuenta que:
 - la base es un cuadrado de lado 1: $(\pm 0.5, -0.7, \pm 0.5)$,
 - los vértices que están en el plano $Y = 0$: $(\pm 0.7, 0, 0)$ y $(0, 0, \pm 0.7)$,
 - y el vértice superior es: $(0, 0.7, 0)$.

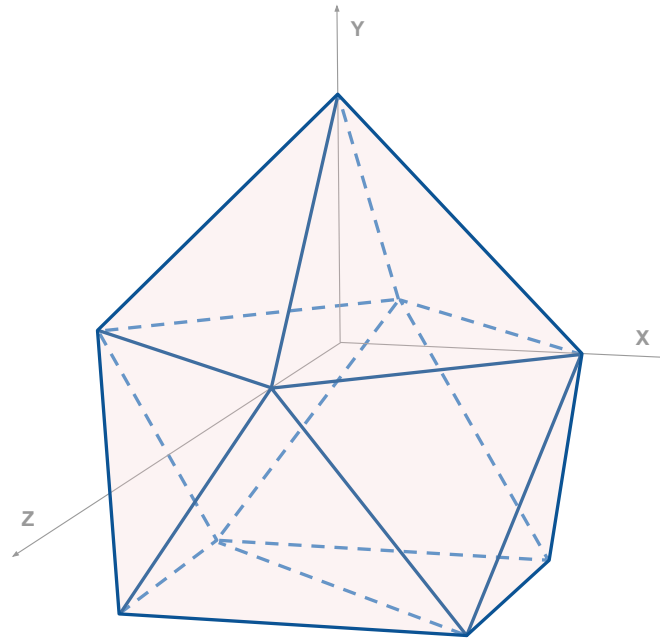


Figura 1: Poliedro j10

Descarga del *Aula Virtual* los ficheros `poligonos.js`, `dibuja.html`, `dibuja.js`, `gl-matrix-min.js` y `comun.js`. Abre `dibuja.html` en el navegador y observa que se obtiene un cuadrado como el que se muestra en la Figura 2 (izquierda). Ese cuadrado se corresponde con el modelo `unCuadradoVC` que se muestra en el Listado 1 y se encuentra en `poligonos.js`. Compruébalo¹.

3. Completa en `poligonos.js` el modelo `j10VC` con los vértices y triángulos obtenidos. Después cambia en la línea 6 de `dibuja.js` el valor de la variable `modeloSeleccionado` así:

```
var modeloSeleccionado = j10VC;
```

Recarga `dibuja.html`, debes obtener un resultado muy similar al de la Figura 2 (derecha).

4. Completa en `poligonos.js` el modelo `j10CI` que debe almacenar al poliedro `j10` utilizando la estructura de caras independientes. Asigna el nuevo modelo a la variable `modeloSeleccionado` y comprueba que el resultado es correcto.
5. Obtén una tira de triángulos para el poliedro `j10`. Después, completa el modelo `j10VCstrip` para que codifique al poliedro `j10` utilizando la tira de triángulos que has obtenido y la estructura de vértices compartidos. Comprueba el resultado.

Observa que el campo `primitiva` cambia a `TRIANGLE_STRIP`. Este campo lo utilizamos para decirle a la GPU cómo ha de interpretar la información que le vamos a enviar. Consulta el Anexo A si quieres saber más sobre qué otras primitivas de dibujo existen en WebGL.

6. Completa el modelo `j10CIstrip` que debe almacenar al poliedro `j10` utilizando la tira de triángulos obtenida en el ejercicio anterior y la estructura de caras independientes.

¹El campo `primitiva` se explica en un ejercicio más adelante

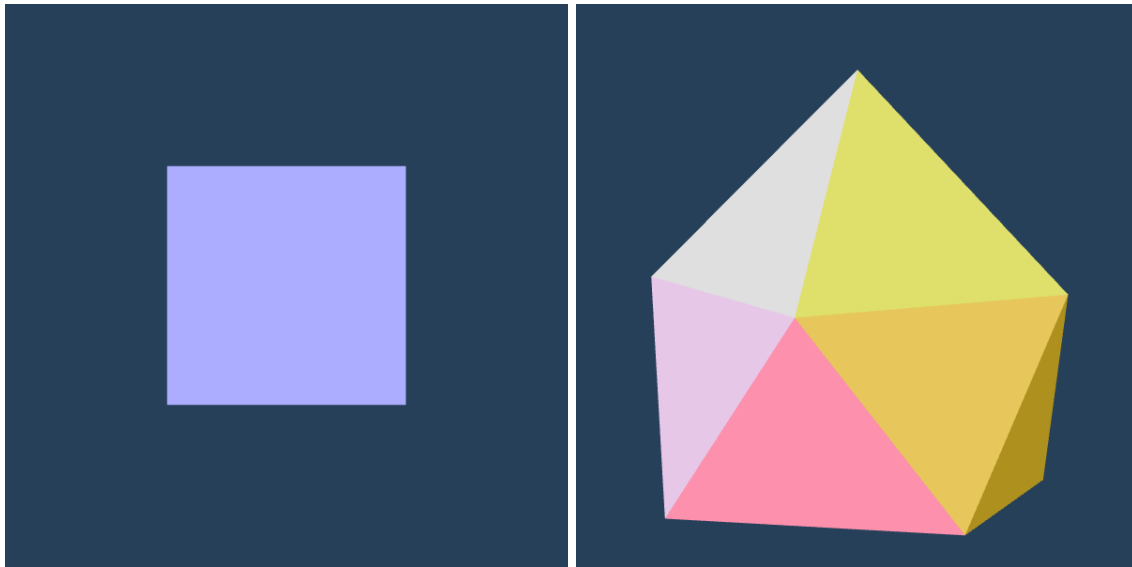


Figura 2: Izquierda: resultado al dibujar el modelo `unCuadradoVC` del Listado 1 y, derecha, poliedro `j10` coloreado

Extensión opcional: si consideras que necesitas practicar más, los poliedros de [Johnson](#) son una buena fuente de donde poder elegir más modelos. Por ejemplo, crear el modelo de un [Icosaedro](#) representa un importante desafío.

Llegados a este punto, has creado modelos poligonales utilizando las dos estructuras de datos vistas en la asignatura, la de vértices compartidos y la de caras independientes. También has experimentado con algunas de las primitivas gráficas soportadas por la GPU, y has podido comprobar que utilizar una primitiva u otra obliga a realizar cambios en el propio modelo poligonal. Consulta el Anexo B si quieres saber más sobre la diferencia entre vértice y atributo en WebGL. Y también el Anexo C si quieres conocer un poco mejor el proceso de dibujado de un modelo poligonal mediante WebGL.

Cuestiones

1. ¿Qué es un abanico de triángulos?, y ¿qué es una tira de triángulos?
2. Compara los costes de almacenamiento de los modelos `j10VC` y `j10CI`. Asume que el coste de almacenar tanto un real de simple precisión como un entero es de 4 bytes.
3. Obtén también el coste de almacenamiento del modelo `j10VCstrip` y compáralo con los dos anteriores. ¿Cuál te ofrece un menor coste de almacenamiento?
4. ¿Cuántos triángulos degenerados contiene la tira de triángulos utilizada en `j10VCstrip`?

Parte II

Transformaciones Geométricas

Descarga del *Aula Virtual* el fichero `primitivasG.js` que contiene los modelos de las primitivas geométricas plano, cubo, tapa, esfera, cono y cilindro. Estas son las primitivas que vas a utilizar en el resto de ejercicios de este boletín. La Tabla 1 recoge la información que necesitas para conocer cómo se han definido cada una de ellas, es decir, dimensión, posición y orientación.

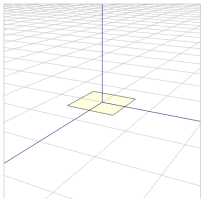
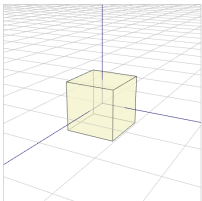
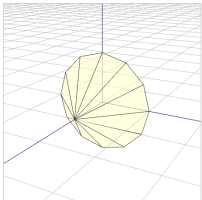
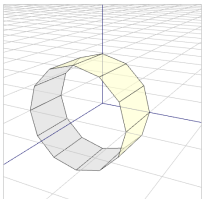
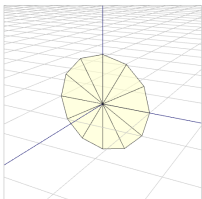
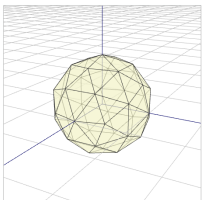
Vista 3D	Descripción
	Plano: es un cuadrado sobre el plano XZ centrado en el origen de coordenadas y lado uno.
	Cubo: está centrado en el origen de coordenadas, con sus caras paralelas a los planos de referencia XY, XZ e YZ, y lado uno.
	Cono: la base reside en el plano XY, tiene radio uno y está centrada en el origen de coordenadas, el eje del cono coincide con el eje Z y la altura es uno.
	Cilindro: igual que el cono.
	Tapa: igual que el cono pero la altura es cero.
	Esfera: está centrada en el origen de coordenadas y su radio es uno.

Tabla 1: Primitivas geométricas en el fichero `primitivasG.js`

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 giros alrededor de los ejes de coordenadas:

- `mat4.fromTranslation (out, v)`
- `mat4.fromScaling (out, v)`
- `mat4.fromXRotation (out, rad)`
- `mat4.fromYRotation (out, rad)`
- `mat4.fromZRotation (out, rad)`

En todas estas funciones:

- el parámetro *out* es una matriz de tipo `mat4` en la que vas a obtener la matriz solicitada,
- el parámetro *v* es un vector de tipo `vec3` donde has de proporcionar los correspondientes valores de traslación o escalado según corresponda,
- y el parámetro *rad* es el ángulo de giro en radianes.

Por último, para realizar la concatenación de transformaciones la biblioteca *glMatrix* solo proporciona la función:

- `mat4.multiply(out, a, b)`, que siempre realiza la siguiente operación: $out = a * b$

Para facilitar la concatenación de transformaciones, en `comun.js` se implementa la función `concat(...M)` que de izquierda a derecha multiplica las matrices proporcionadas como parámetros de entrada y devuelve el correspondiente resultado. **Recuerda que el producto de matrices no es conmutativo.** Por ejemplo, si deseas obtener $M = M_1 * M_2 * M_3$, debes hacer:

- `var M = concat (M1, M2, M3);`

En esta sesión, nuestro objetivo es modelar el tanque que puedes ver en la Figura 3.

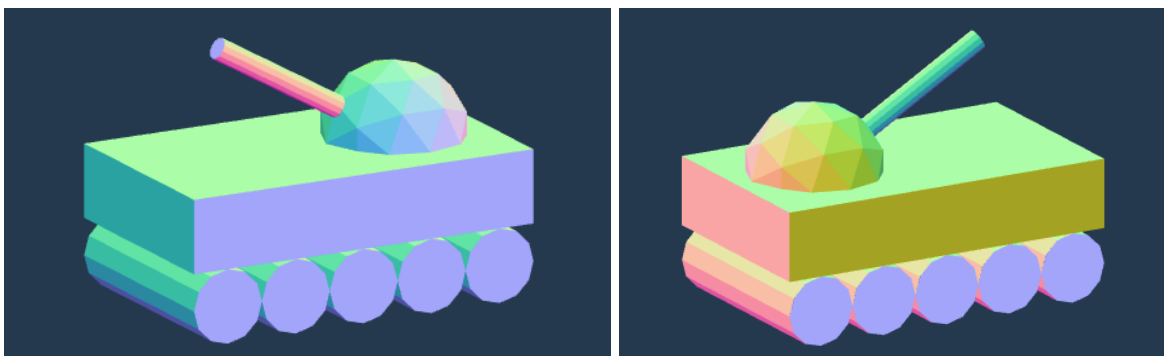


Figura 3: Tanque modelado a partir de primitivas geométricas básicas

Descarga del *Aula Virtual* los ficheros `primitivasG.js`, `comun.js`, `gl-matrix-min.js`, `tanque.html` y `tanque.js`. En este último fichero, la función `drawScene()` (que se reproduce en el Listado 3) escala un cubo para obtener el cuerpo del tanque. Comprueba en tu navegador que obtienes el mismo resultado que el que se muestra en la Figura 4.

Listado 3: Ejemplo de transformación geométrica de un modelo

```
function drawScene() {  
  
    gl.clear(gl.COLOR_BUFFER_BIT|gl.DEPTH_BUFFER_BIT);  
  
    5    // Crea una matriz de tipo mat4 y devuelve la matriz identidad  
    var matR = mat4.create();  
    var matS = mat4.create();  
  
    // Obtiene la transformación de rotación de acuerdo a la interacción del usuario  
    10    matR = getRotationMatrix();  
  
    // Crea una transformación de escalado  
    mat4.fromScaling (matS, [1.0, 0.2, 0.6]);  
  
    15    // Establece como matriz de transformación del modelo: matR * matS  
    setUniform ("modelMatrix", concat(matR, matS));  
  
    // Dibuja la primitiva cubo  
    draw(cubo);  
    20  
}
```

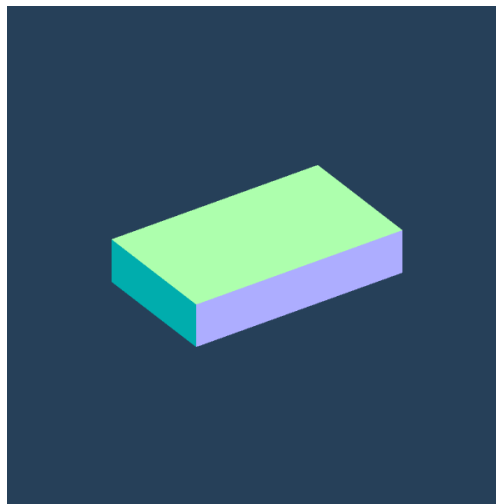


Figura 4: Cuerpo del tanque

Comprueba también que el *shader* de vértices (en `tanque.html`) contiene una variable *uniform* (ver Anexo D) de tipo `mat4` llamada `modelMatrix`. Esta es la matriz de transformación del modelo y se multiplica por cada vértice para obtener su posición definitiva.

1. Procede con las ruedas del tanque (ver Figura 5). Tienen un radio de 0.1 y una longitud de 0.6. Primero utiliza cinco cilindros para las cinco ruedas. Después coloca tapas a ambos lados de cada cilindro.



Figura 5: Cuerpo y ruedas del tanque

2. Coloca la esfera que hace de torreta del tanque, radio 0.2 (ver Figura 6).

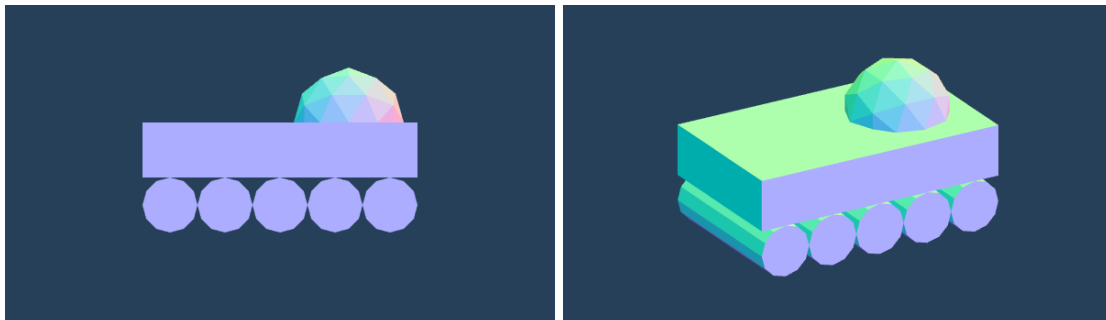


Figura 6: Cuerpo, ruedas y torreta del tanque

3. Coloca el cañón del tanque, es un cilindro de radio 0.03 y longitud 0.6. Un extremo del cilindro está en el centro de la torreta, y el otro extremo fíjate cómo queda en la Figura 7. Colócale una tapa al extremo visible del cañón.

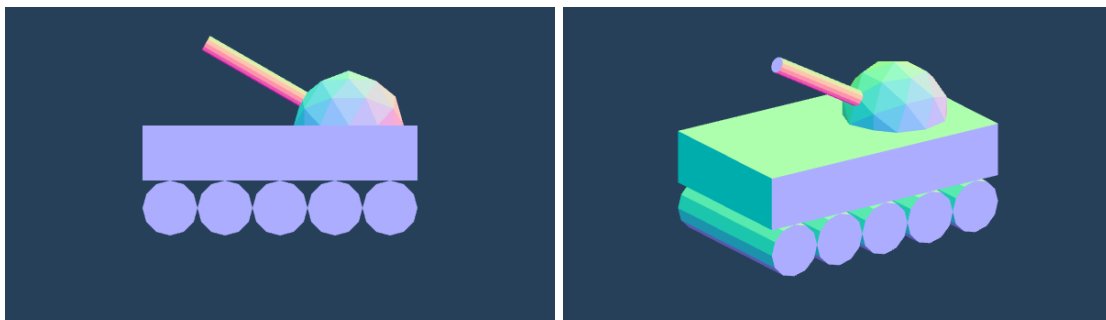


Figura 7: Tanque terminado

Extensión opcional: añade 12 pequeños conos alrededor de la torreta y otros 10 en las correspondientes tapas de las ruedas (ver Figura 8). Para colocar los conitos alrededor de la torreta, piensa cómo puedes resolverlo sin utilizar trigonometría.

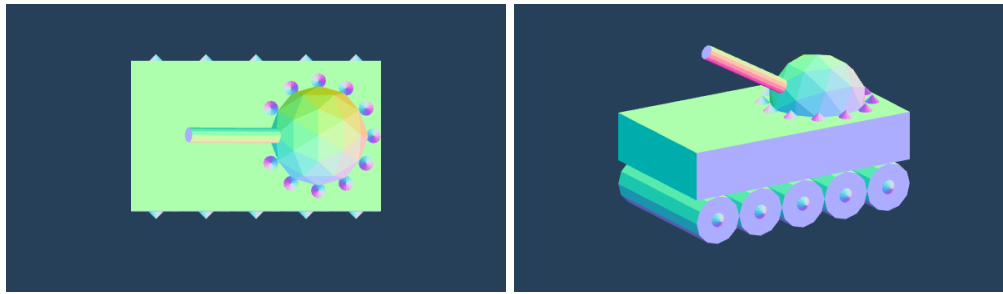


Figura 8: Tanque con pequeños conos

Si has llegado a este punto significa que entiendes cómo están definidas las distintas primitivas geométricas en el fichero `primitivasG.js`, que para cada modelo que necesites incluir en tu escena habrás de obtener su matriz de transformación, y que en el *shader* de vértices deberás operar cada vértice del modelo por su matriz de transformación correspondiente.

Los ejercicios que has realizado hasta el momento no son especialmente fáciles. Cada ejercicio encierra alguna complicación que lo hace diferente del resto. Aunque te sea inevitable resolverlos a base de realizar pruebas, y corregir cuando no te sale lo que quieres, esta no es manera de proceder, trata de resolverlo primero en papel, dibújate cada pieza cómo evoluciona a medida que la vas transformando, y si al implementarlo no te sale lo que buscas, averigua dónde tienes el error y trata de entenderlo. Y si tienes dudas no las dejes pasar, acláralas con tu profesor en clase o en sus tutorías.

Cuestiones

5. ¿Qué es la matriz de transformación del modelo?
6. ¿En qué tipo de *shader*, vértices o fragmentos, utilizas la matriz de transformación del modelo? Justifica tu respuesta.
7. Si tienes tres transformaciones representadas cada una de ellas por las matrices mA , mB y mC , ¿cómo debes operar las matrices para obtener la matriz de transformación del modelo cuando la primera transformación que deseas aplicar es mB , después mA y por último mC ?
8. ¿Puedes demostrar con un ejemplo que el producto de matrices no es conmutativo? Dibújalo sobre el papel.
9. Plantea qué transformaciones añadirías al cañón del tanque para que se pueda tanto rotar como subir y bajar. Detalla dónde las incluirías entre las transformaciones que ya le aplicas al cañón.

Parte III

Transformaciones de Cámara y Proyección

En los ejercicios anteriores podías cambiar el punto de vista de manera interactiva. Sin embargo, tu escena debía estar en el rango $[-1,1]$ y solo podías obtener como resultado una proyección paralela. Como vimos en clase, para disponer de un modelo de cámara completo necesitas de dos nuevas transformaciones: transformación de la cámara y transformación de proyección. Antes de comenzar con los ejercicios **repasa ambos tipos de transformaciones** y resuelve las dudas.

El modelo de cámara implementado sigue siendo un modelo sencillo pero suficiente para la asignatura. Te permite tanto obtener una proyección perspectiva como modificar la posición de la cámara desplazando el ratón mientras se mantiene presionado su botón izquierdo, acercar o alejar la cámara de la escena con la rueda central del ratón, y esto mismo pero con la tecla *Shift* pulsada te permite modificar el ángulo de visión conocido por el nombre de *fovy*. Ve al Anexo [E](#) para saber más sobre este modelo.

La biblioteca *glmMatrix* nos proporciona las funciones que necesitamos para construir las correspondientes matrices de transformación:

- Matriz de transformación de la cámara: `mat4.lookAt (out, eye, center, up);`
donde *eye* es la posición de la cámara, *center* es el punto de interés, y *up* es el vector de inclinación.
- Matriz de transformación de la proyección:
 - Perspectiva: `mat4.perspective (out, fovy, aspect, near, far);`
donde *fovy* es el ángulo de apertura del campo de vista, *aspect* es la relación de aspecto de la base de la pirámide truncada, y *near* y *far* son respectivamente las distancias a los planos de recorte frontal y trasero que hacen finito el volumen de la vista.
 - Paralela: `mat4.ortho (out, left, right, bottom, top, near, far);`
donde (*left*, *bottom*, *near*) y (*right*, *top*, *far*) se corresponden con las coordenadas de los dos puntos que forman la diagonal del volumen de la vista en forma de caja.

A partir de la matriz del modelo y la de la cámara se obtiene la **matriz del modelo-vista**:

- `mat4.multiply (Mmv, Mc, Mm); // $M_{modeloVista} = M_{camara} * M_{modelo}$`

La matriz modelo-vista y la de proyección son las que operaremos en el *shader* de vértices con los vértices del modelo. De esta manera, el *shader* nos quedaría tal y como se muestra en el Listado 4 (fíjate bien en el orden de las operaciones).

Listado 4: *Shader* de vértices que permite obtener diferentes vistas de nuestra escena

```
in      vec3 VertexPosition;    // atributo de posición del vértice
uniform mat4 modelViewMatrix;   // resultado de Mcamara * Mmodelo
uniform mat4 projectionMatrix;  // perspectiva o paralela

5 void main()  {

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

}
```

En esta sesión, nuestro objetivo es modelar un globo terráqueo de escritorio como el de la Figura 9. Pero antes de continuar trata de responder a las siguientes preguntas. En los ejercicios anteriores, como es lógico, la matriz de transformación del modelo es diferente para cada una de las primitivas que forman la escena, ¿ocurre lo mismo con las matrices de transformación de la cámara y de proyección? ¿son las mismas o diferentes para cada primitiva?

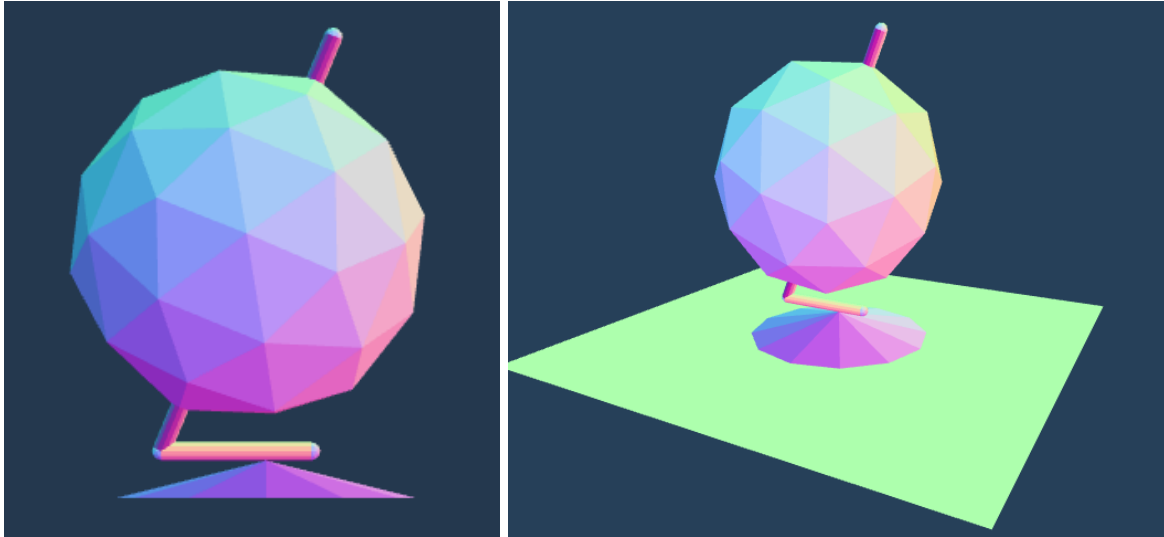


Figura 9: Modelo de un globo terráqueo de escritorio

Descarga `globo.html` y `globo.js` del *Aula Virtual* (junto con `gl-matrix-min.js`, `comun.js` y `primitivasG.js`).

Carga en el navegador `globo.html` y comprueba que aparece un cono que se corresponde con la base del globo terráqueo. No lo modifiques y contesta a las siguientes preguntas:

- ¿qué dimensiones tiene el cono tras aplicarle la transformación del modelo?,
- ¿dónde se encuentra ubicado y orientado respecto al sistema de coordenadas global?

1. Crea el cilindro horizontal, tiene un radio de 0.0625 y su longitud no cambia, sigue siendo 1. Se apoya sobre el vértice del cono, sin atravesarlo, a 0.3 de un extremo y a 0.7 del otro. Observa el resultado en la Figura 10.

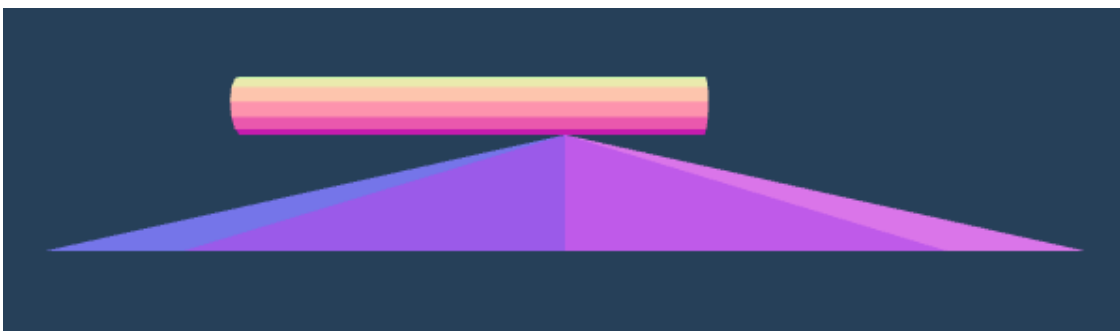


Figura 10: Cilindro corto del modelo del globo

2. Crea el cilindro que atraviesa el globo, que tiene el mismo radio que el horizontal pero una longitud de 3.0. La inclinación es de -23 grados respecto a la vertical. Y el centro de su base inferior coincide con el centro de la base del otro cilindro. Obsérvalo en la Figura 11.

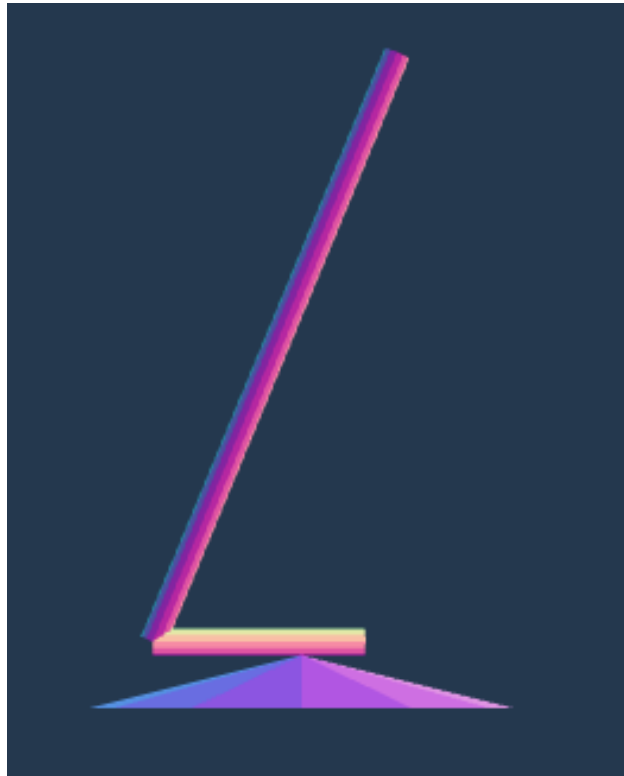


Figura 11: Cilindro largo del modelo del globo

3. Coloca la esfera que hace de globo. Su radio es 1.2 y está centrada sobre el cilindro largo. La Figura 9 te muestra cómo ha de quedar. Observa que también queda inclinada el mismo ángulo que el cilindro que la atraviesa.
4. Para terminar pon un plano que haga de soporte y añade tres esferitas que se colocan en los extremos de los cilindros (ver Figura 12).

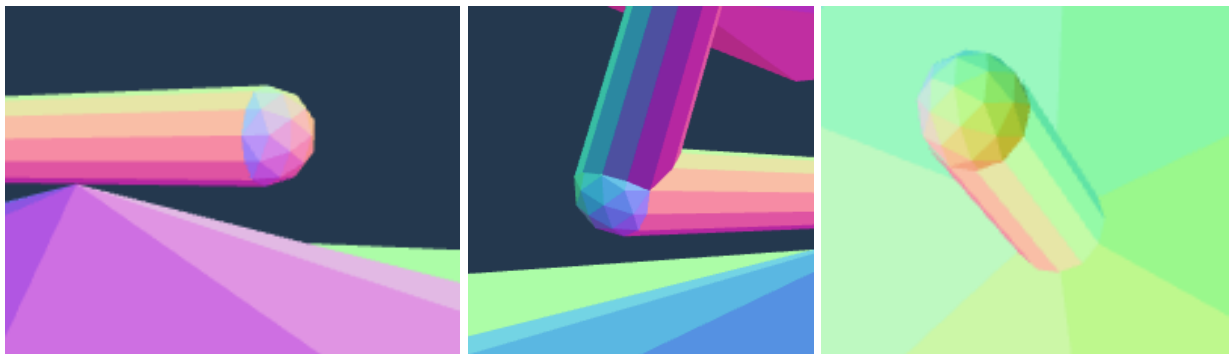


Figura 12: Detalle de las esferas al final de los cilindros

Extensión opcional: haz que la función `drawGlobo` tenga como parámetros de entrada una posición y una orientación para aplicarlas a modo de traslación y rotación alrededor del eje Y a todas las primitivas que componen el globo. De esta manera, cuando llames a dicha función podrás hacer que el globo se coloque y oriente donde y como tú quieras. Por ejemplo así: `drawGlobo ([1, 0, -0.5], Math.PI);`. Y repite esto mismo para el modelo del tanque y junta a ambos modelos en la misma escena. Podrás conseguir resultados como el que se muestra en la Figura 13.

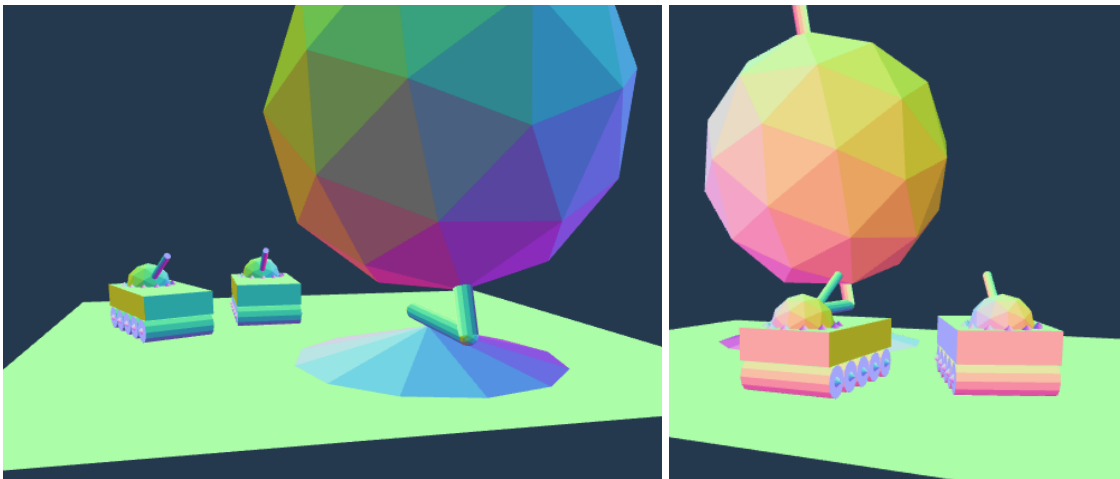


Figura 13: Cambio de la posición y orientación de los objetos en la escena

Cuestiones

10. ¿Qué es la matriz de transformación de proyección? ¿y de qué tipos conoces?
11. ¿Qué significa cada uno de los parámetros de la función `perspective`? Dibújalo en papel.
12. ¿Qué función de la biblioteca *glMatrix* utilizarás si quieres trabajar con proyección paralela?
13. ¿Qué función de la biblioteca *glMatrix* te permite obtener fácilmente la matriz de transformación de la cámara?
14. ¿Qué dos matrices se operan habitualmente entre ellas para crear la matriz modelo vista?
15. En el *shader* de vértices, ¿en qué orden debes operar cada vértice con las matrices de proyección, cámara y modelo?
16. Si el usuario modifica de manera interactiva la posición de la cámara, está claro que para obtener la nueva vista debes obtener la nueva matriz de transformación de la cámara pero, ¿te sigue valiendo la matriz de transformación de proyección que ya tenías o por el contrario también debes obtenerla?

Parte IV

Modelos articulados

Los ejercicios de transformaciones con modelos articulados conllevan algo más de dificultad que los que has realizado hasta el momento. En un modelo articulado, la posición y orientación de las diferentes partes del modelo dependen ya no solo de la posición y orientación inicial de cada parte, sino también de la posición y orientación de las otras partes del mismo modelo. Piensa por ejemplo en el modelo de un brazo, en el que la mano la sitúas al final del antebrazo, y al mover el antebrazo se mueve la mano también pero no porque la hayas movido sino porque está unida al antebrazo mediante la articulación de la muñeca. El objetivo de esta sesión de prácticas es crear el modelo de un flexo cuya principal diferencia con el resto de modelos del boletín es que precisamente se trata de un modelo articulado (ver Figura 14). Primero vas a crear el modelo como si se tratara de un objeto estático, y después le añadirás las transformaciones necesarias para darle su movimiento característico.

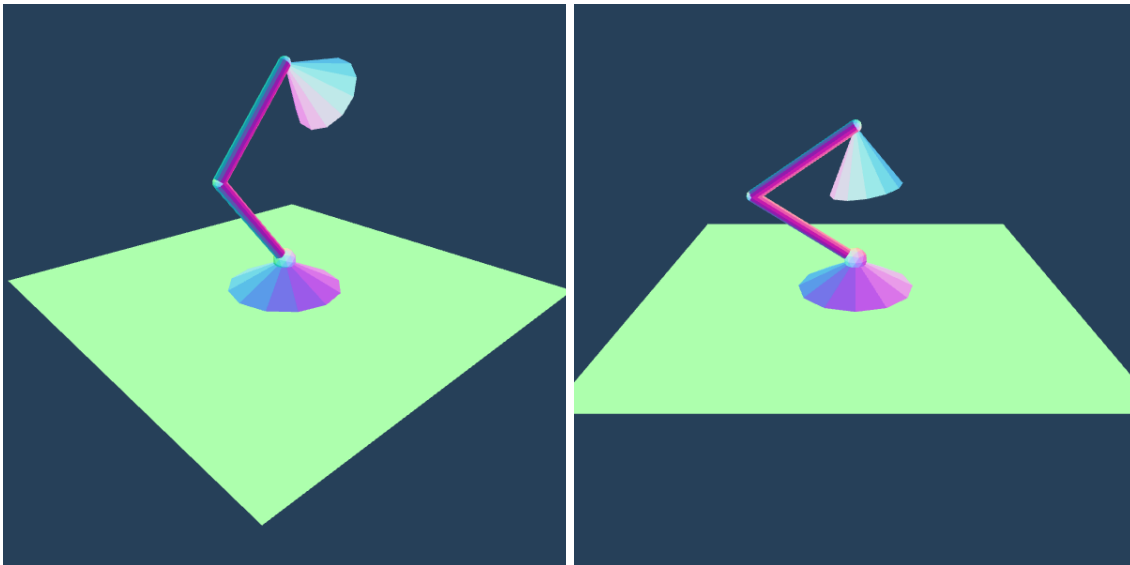


Figura 14: Modelo de un flexo

1. Modela un flexo similar al que se muestra en la Figura 14. Se original en lo que respecta al diseño (tanto en dimensiones como en qué primitivas utilizas), pero mantén el número de articulaciones, que en el ejemplo son tres, una por cada esfera (ver Figura 15). En primer lugar **modela la escena en el papel** (este punto no es negociable), da medidas a las primitivas y después impleméntalo. De momento olvídate de la posibilidad de que los brazos del flexo están articulados, eso ya lo harás después. **Nota: para poder atender dudas en este ejercicio es necesario que hayas realizado el correspondiente diseño en papel.**
2. Mejora tu modelo del flexo añadiendo algunos grados de libertad en cada una de las articulaciones. En la Figura 15 se muestra un esquema de cómo el modelo se articula utilizando solo dos variables: α y β . La variable α gobierna el plegado y desplegado del flexo, mientras que la variable β establece el giro del flexo sobre sí mismo.

El valor de estas variables lo vas a cambiar de forma interactiva a través de la pulsación de dos teclas. Con este fin, añade un manejador de eventos que dispare la ejecución de una función cada vez que se

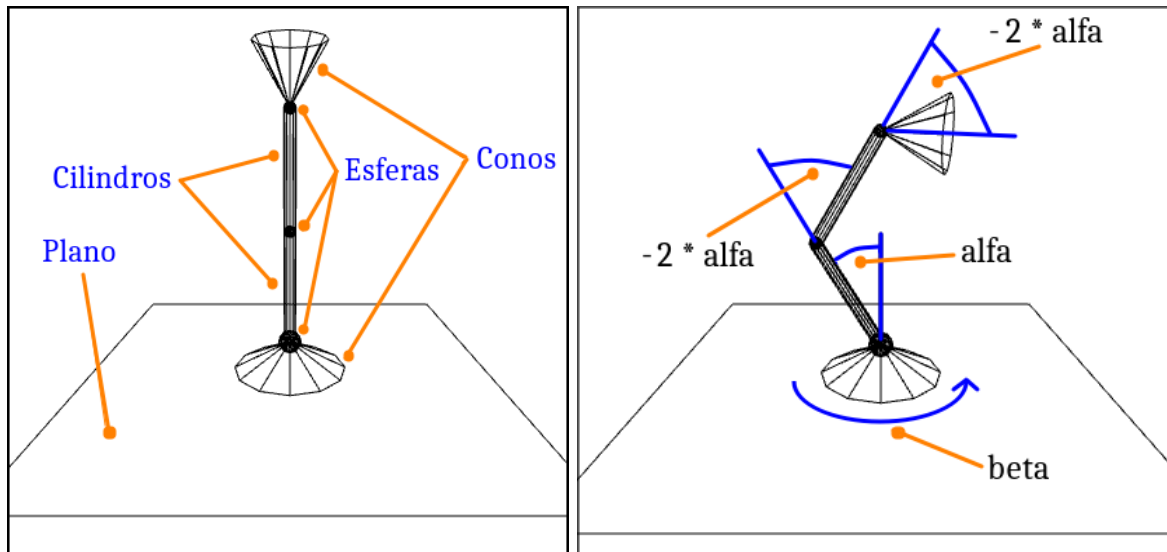


Figura 15: Esquema del flexo, izquierda, y grados de libertad, derecha

produzca una pulsación en el teclado. Dicha función será la encargada de comprobar qué acción realizó el usuario y actuar en consecuencia. El código del Listado 5 muestra cómo atender el evento producido al pulsar una tecla. Al producirse el evento, se ejecuta la función especificada como parámetro que, en este caso, se encarga de incrementar el ángulo correspondiente a la tecla que se haya pulsado. Las dos variables globales que almacenan los ángulos que gobiernan la orientación del flexo son *alfa* y *beta*. Realiza las siguientes modificaciones al código del ejercicio anterior:

- Añade interacción con el teclado utilizando el código del Listado 5.
- Llama a `initKeyboardHandler()` a continuación de `initHandlers()`;
- Añade las transformaciones de rotación a los diferentes elementos del flexo para que respondan a las acciones del usuario. Quizá esta sea la parte más complicada ya que has de conseguir que tu flexo no se autodestruya cuando comience a plegarse o a girar.

Listado 5: Atención de eventos de teclado

```

var alfa = beta = 0.0;

function initKeyboardHandler () {

5   document.addEventListener("keydown",
    function (event) {
        switch (event.key) {
            case 'a': alfa += 0.03; if (alfa > 1.05) alfa = 1.05; break;
            case 'A': alfa -= 0.03; if (alfa < -1.05) alfa = -1.05; break;
10         case 'd': beta += 0.05; break;
            case 'D': beta -= 0.05; break;
        }
        requestAnimationFrame(drawScene);
    }, false);
15 }

```

Extensión opcional: al igual que en los ejercicios opcionales de la sesión anterior, encapsula el código que dibuja al flexo en una función y crea una escena junto con el globo y el tanque. Podrás conseguir resultados como el que se muestra en la Figura 16.

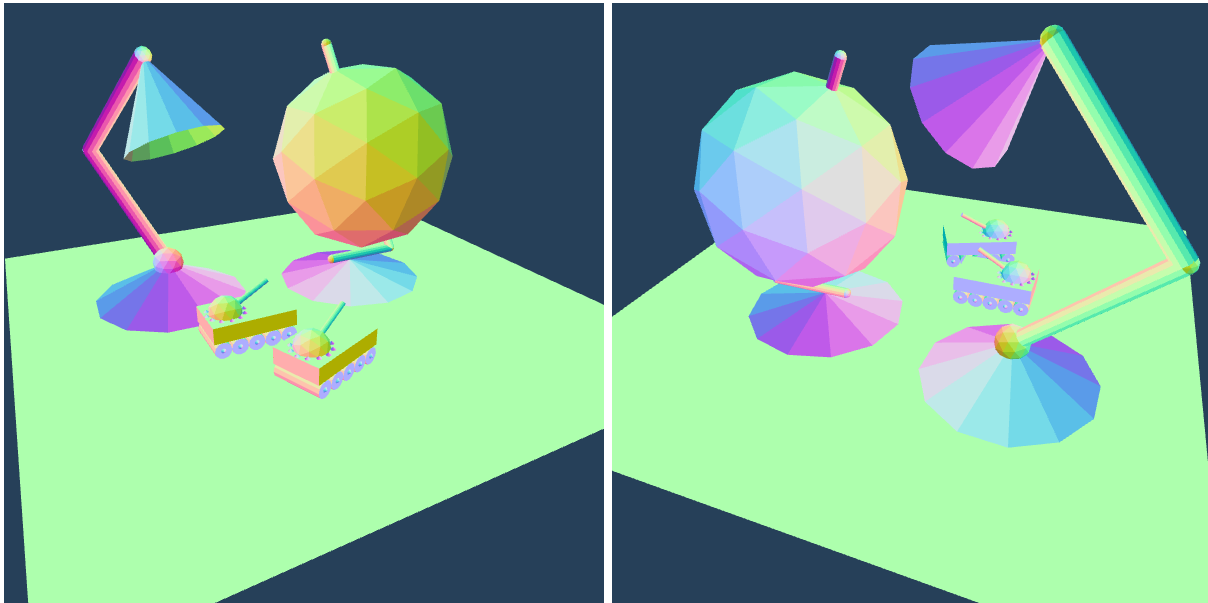


Figura 16: Escena con los modelos creados en esta práctica

Además, en toda esta práctica se han pintado los polígonos calculando y transformando sus respectivas normales en valores de color. En la siguiente práctica ya se estudia iluminación y se podrán obtener resultados mucho mejores. Pero antes, podemos quizá mejorar el aspecto visual algo más. Por ejemplo, envía al *shader* de fragmentos un valor de color antes de pintar cada objeto de la escena, de manera que este color se combine con el obtenido a partir de las normales a fin de al menos conseguir gamas de tonos diferentes para cada objeto (ver Figura 17). Procede de la siguiente manera:

- Declara `myColor` en el *shader* de fragmentos:
 - `uniform vec3 myColor;`
- También en el *shader* de fragmentos, suma o multiplica `myColor` al resultado de la función `pow`.
- En el lado de la aplicación, obtén una referencia a `myColor`. La referencia es todo lo que necesitas para poder modificar una variable de tipo `uniform`. Además, como esa referencia no cambia durante la ejecución de la aplicación solo es necesario obtenerla una vez. En tu código sustituye la llamada a `initUniformRefs` por:
 - `initUniformRefs("modelViewMatrix", "projectionMatrix", "myColor");`
- Y ahora ya solo te queda especificar un color de dibujo justo antes de dibujar un objeto, puedes hacerlo con la función:
 - `setUniform ("myColor", [1.0,0.5,0.5]);`

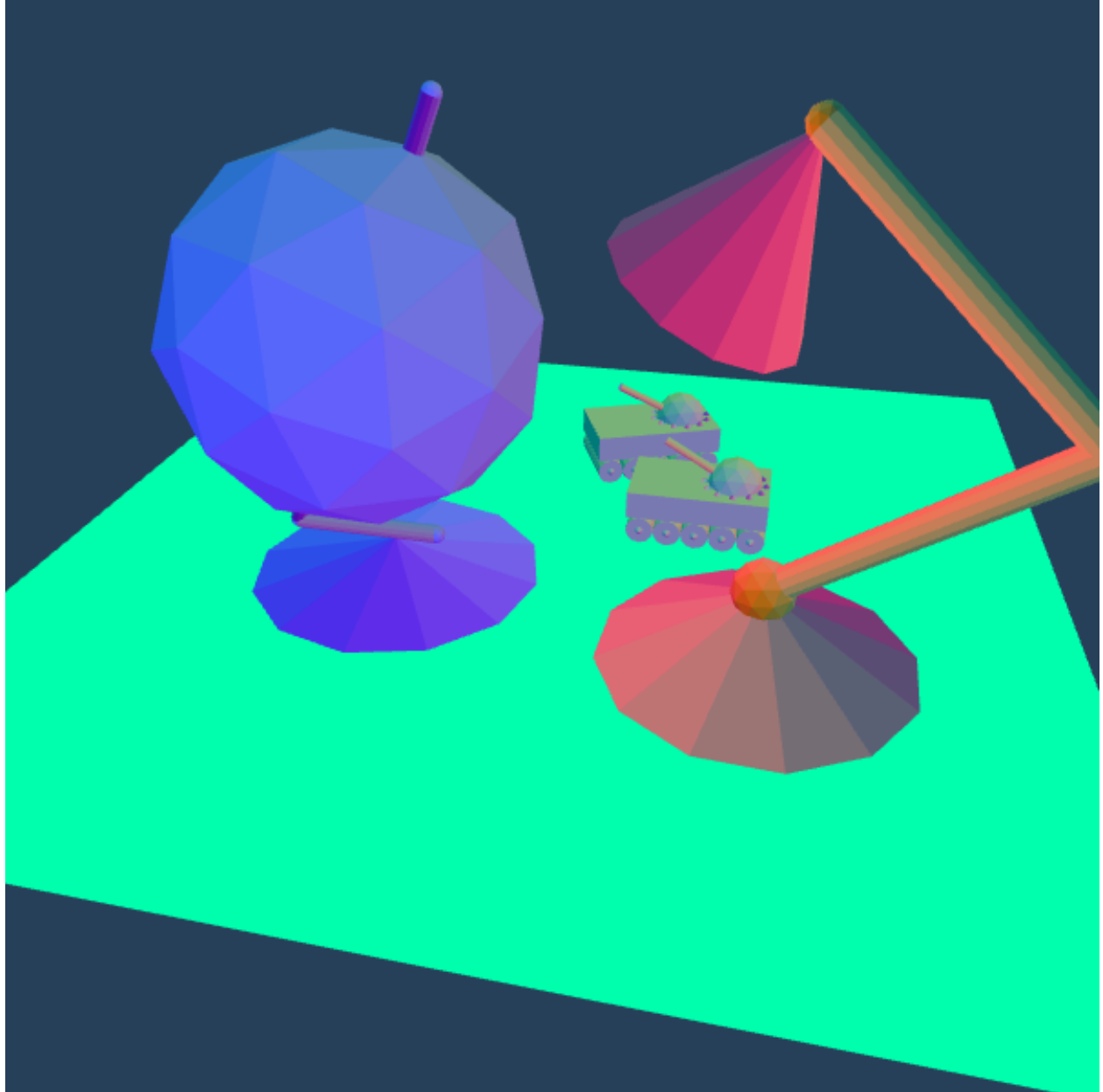


Figura 17: Escena resultado de añadir colores por objetos

Anexos

A. Primitivas gráficas

Las primitivas geométricas de dibujo que WebGL soporta son básicamente el punto, el segmento de línea y el triángulo. Cada primitiva se define especificando la secuencia de sus respectivos vértices. En WebGL, la primitiva geométrica establece cómo han de ser agrupados los vértices tras ser operados en el procesador de vértices. Son las siguientes:

- Dibujo de puntos:
 - `gl.POINTS`, dibuja un punto por vértice.
- Dibujo de líneas:
 - Segmentos independientes: `gl.LINES`, dibuja un segmento de línea cada dos vértices.
 - Secuencia o tira de segmentos: `gl.LINE_STRIP`, dibuja una polilínea que une en orden todos los vértices.
 - Secuencia cerrada de segmentos: `gl.LINE_LOOP`, igual que la anterior pero además une el primero con el último vértice con un segmento de línea.
- Triángulos:
 - Triángulos independientes: `gl.TRIANGLES`, dibuja un triángulo cada tres vértices.
 - Tira de triángulos: `gl.TRIANGLE_STRIP`
 - Abanico de triángulos: `gl.TRIANGLE_FAN`

Tanto la tira como el abanico de triángulos representan una serie de triángulos conectados a través de aristas compartidas. En concreto, la tira se define mediante una secuencia de vértices donde los tres primeros vértices forman el primer triángulo y, a partir de ahí, cada vértice se añade a los dos últimos vértices del triángulo anterior para formar un nuevo triángulo. Por su parte, el abanico de triángulos añade cada vértice al primer y tercer vértice del triángulo anterior.

Por ejemplo, el vector de índices para el modelo del cuadrado del Listado 1 depende del tipo de primitiva gráfica que utilicemos para pintarlo, tal y como se muestra en la Figura 18 (por supuesto hay más soluciones correctas además de las que ahí figuran).

TRIANGLES	[0, 1, 2, 0, 2, 3]
TRIANGLE_STRIP	[3, 0, 2, 1]
TRIANGLE_FAN	[0, 1, 2, 3]
LINES	[0, 1, 1, 2, 2, 3, 3, 0]
LINE_STRIP	[0, 1, 2, 3, 0]
LINE_LOOP	[0, 1, 2, 3]
POINTS	[0, 1, 2, 3]

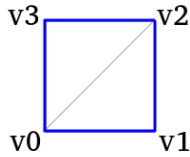


Figura 18: Vectores de índices por tipo de primitiva gráfica para el modelo del cuadrado

B. Vértices y atributos

Habitualmente solemos 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 llamamos **atributos**. Por ejemplo, la posición y la normal son dos de los atributos más comunes de un vértice. Este planteamiento permite al programador definir todos los atributos que considere oportunos dentro de un máximo que se puede consultar con la orden `gl.getParameter(gl.MAX_VERTEX_ATTRIBS)` (que habitualmente retorna 16, valor mínimo establecido por el estándar). En cuanto al tipo de los atributos, estos pueden ser reales, enteros, vectores, etc.

Todos los atributos que por vértice se haya decidido incluir en la descripción de un modelo y que, por lo tanto, se van a enviar al sistema gráfico, se declararán siempre en el *shader* de vértices de tipo `in`. Por ejemplo, el Listado 6 muestra la declaración de tres atributos por vértice: posición, normal y coordenadas de textura.

Listado 6: Declaración de atributos en el *Shader* de vértices

```
#version 300 es

// Declaración de los atributos por vértice
in vec3 VertexPosition;      // atributo posición (x, y, z)
5 in vec3 VertexNormal;      // atributo normal (nx, ny, nz)
in vec2 VertexTexCoords;     // atributo coordenada de textura (s, t)

void main() {
    ...
10 }
```

Por otra parte, recalcar que 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.

C. Visualización de un modelo

La visualización de un modelo en WebGL requiere de tres tareas:

1. Alojar el modelo en la memoria del sistema gráfico. Idealmente cada modelo necesitará dos buffers en dicha memoria, uno para almacenar los vértices y otro para los índices (asumiendo que en la descripción del modelo se utiliza la estructura de vértices compartidos). Esta tarea no hay que hacerla cada vez que se requiera pintar el modelo, es decir, una vez que un modelo reside en la memoria del sistema gráfico ya puede ser dibujado todas las veces que sea necesario.
2. Para cada atributo de un vértice, establecer una relación entre el atributo declarado en el *Shader* de vértices y cómo la información correspondiente a dicho atributo se encuentra incluida en la descripción del modelo.
3. Ordenar el dibujo indicando qué *buffer* de vértices y qué *buffer* de índices se han de utilizar, tipo de primitiva gráfica que se tiene que utilizar para pintar y cantidad de índices.

A pesar de lo molesto o pesado que puede resultar este proceso, el lado bueno es que siempre es igual, siempre será el mismo código, no depende de lo que vayamos a pintar, es lo mismo pintar un solo triángulo que por ejemplo pintar un dragón de miles de triángulos, por lo que resulta un código fácilmente encapsulable en funciones y reutilizable.

- Para dibujar un modelo, este se ha de almacenar previamente en *buffer objects*.
 - Un *buffer object* no es mas que una porción de memoria reservada dinámicamente y controlada por el propio procesador gráfico.

Para un modelo poligonal con vértices e índices separados necesitaremos dos *buffer objects*, uno para el vector de vértices y otro para el de índices. Una vez creados, se copian los datos a cada uno de ellos. El Listado 7 recoge estas operaciones, examínalo y consulta la especificación del lenguaje si deseas conocer más detalles sobre las funciones utilizadas.

- Con el *shader* compilado y enlazado, hay que obtener referencias a las variables del *shader* que representan los distintos atributos de un vértice, que en nuestro ejemplo solo hay un atributo que es la posición del vértice (sus coordenadas), así como habilitar el atributo correspondiente (Listado 8).
- Por último, para cada atributo hay que especificar dónde y cómo se encuentra almacenado. Y después ya se puede ordenar el dibujado, indicando tipo de primitiva y número de elementos. El Listado 9 muestra estas operaciones. De nuevo, acude a la especificación del lenguaje si deseas conocer más detalles de las órdenes utilizadas.

Listado 7: Creación de los *Buffer objects* para vértices e índices

```
// Crea un buffer object para el vector de vértices del modelo
model.idBufferVertices = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, model.idBufferVertices);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(model.vertices), gl.STATIC_DRAW);
5
// Crea otro buffer object para el vector de índices del modelo
model.idBufferIndices = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, model.idBufferIndices);
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(model.indices),
10          gl.STATIC_DRAW);
```

Listado 8: Obtener la referencia de un atributo declarado en el *shader* de vértices

```
// Se obtiene una referencia al atributo VertexPosition que
// ha de estar declarado en el shader de vértices
program.vertexPositionAttribute = gl.getAttribLocation(program, "VertexPosition");
5
// Se habilita dicho atributo
gl.enableVertexAttribArray(program.vertexPositionAttribute);
```

Listado 9: Dibujado de un modelo

```
// Se establece el buffer de vértices a utilizar en el proceso de dibujado
gl.bindBuffer(gl.ARRAY_BUFFER, model.idBufferVertices);

// Se indica que hay un atributo por vértice que consta de tres floats
```

```
5 gl.vertexAttribPointer(program.vertexPositionAttribute, 3, gl.FLOAT, false, 0, 0);

// Se establece el buffer de índices
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, model.idBufferIndices);

10 // Se dibuja un único triángulo (3 índices)
gl.drawElements(gl.TRIANGLES, model.indices.length, gl.UNSIGNED_SHORT, 0);
```

D. Variables Uniform

Una variable de tipo uniform se comporta como una constante durante el dibujado de un modelo, pero como una variable entre distintos dibujados. Pueden existir tanto en el *shader* de vértices como en el de fragmentos, y su valor se ha de establecer desde la aplicación y nunca en el propio *shader*. Por ejemplo, el Listado 10 muestra un ejemplo de *shader* de fragmentos en el que en lugar de asignar un color fijo al fragmento, se le asigna la variable *myColor* que es de tipo uniform.

Listado 10: Ejemplo de variable uniform

```
#version 300 es

precision mediump float;

5 uniform vec4 myColor;          // variable que contiene el color de los fragmentos
out vec4 fragmentColor;        // color final del fragmento

void main() {

10 // fragmentColor = vec4 (1.0, 0.0, 0.0, 1.0);
   fragmentColor = myColor;

}
```

De esta manera todos los fragmentos de un modelo son coloreados utilizando el color que haya en la variable *myColor*. Sin embargo, entre dos dibujados se puede modificar el valor de dicha variable. Para desde la aplicación establecer el valor de la variable uniform hay que obtener en primer lugar una referencia a dicha variable, y ya después asignarle un valor, tal y como se muestra en el Listado 11.

Listado 11: Establece el valor de una variable uniform

```
// se obtiene la referencia a la variable 'myColor' de tipo uniform en el shader
idMyColor = gl.getUniformLocation(program, "myColor");

// se utiliza la referencia obtenida para modificar el valor de la variable
5 gl.uniform4f(idMyColor, 1.0, 0.0, 1.0, 1.0);
```

La referencia obtenida con la orden `getUniformLocation` no cambia durante la ejecución de la aplicación por lo que solo es necesario obtenerla una vez.

E. Modelo de cámara

El modelo de cámara se basa en colocar una cámara en la superficie de la esfera, punto P en la Figura 19, mirando hacia el origen de coordenadas, y siendo su vector de inclinación el vector $(0, 1, 0)$.

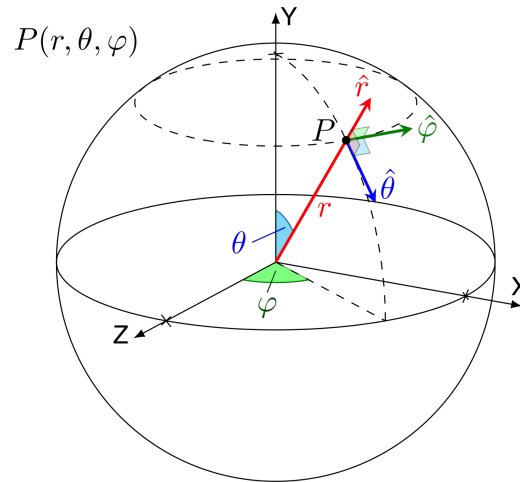


Figura 19: Modelo de cámara interactiva

El punto P se mueve en la superficie en base a dos ángulos. El ángulo ϕ , azimutal, se corresponde con el movimiento izquierda y derecha del ratón y por lo tanto varía en el intervalo $[0, 2\pi]$. Mientras que el ángulo θ , polar o colatitud, se corresponde con el movimiento arriba y abajo del ratón y varía en el intervalo $[0, \pi]$. Estos dos ángulos junto con radio r permiten describir al punto P mediante coordenadas esféricas. La [conversión a coordenadas rectangulares](#) se muestra en el Listado 12.

Listado 12: Conversión de coordenadas esféricas a rectangulares

```
// coordenadas esféricas a rectangulares
var x = radius * Math.sin(myPhi) * Math.sin(myZeta);
var y = radius * Math.cos(myPhi);
var z = radius * Math.sin(myPhi) * Math.cos(myZeta);
```