CLASE 4 IG: Perspectiva

Práctica 4

Partimos de la plantilla clase 4-1 en Glitch, que ya trae incorporado el tema de los manejadores de eventos, la librería gl-matrix incorporada, los métodos para la pila de matrices, etc.

Creando una model matrix inicial

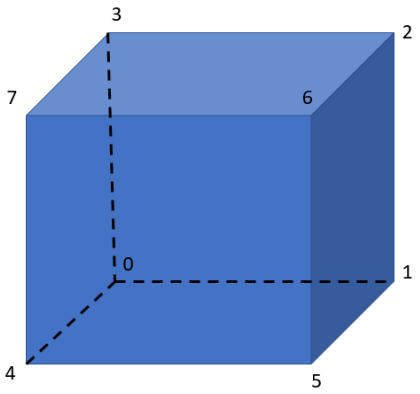
Lo primero es ir al render, entre las dos llamadas a gl.bindBuffer, y meter el siguiente código para inicializar la matriz de modelado

```
// Set the model Matrix.
modelMatrix = mat4.create();
mat4.identity(modelMatrix);
```

Creando un cubo inalámbrico usando índices

Usando índices (leer antes el documento IG4-3: primitivas WebGL)

Si queremos pintar un cubo 3D no basta con dar el vector de vértices. Lo más cómodo es indicar un vector de índices. Como por ahora vamos a pintar solo sus aristas, lo que haremos será indicar un vector de 8 vértices por un lado, y otro vector para los índices de las aristas, por parejas. Un cubo tiene 12 aristas, y por tanto habrá que indicar 24 índices.



Para meter los índices habrá que crear un nuevo buffer esta vez de índices. Lo primero es declarar var index_buffer como variable global, y luego crear el buffer al lado de donde creábamos el buffer de vértices

```
var index_buffer;
…
// create index buffer
index_buffer = gl.createBuffer();
```

Pintando el cubo con el vector de índices

Nos vamos a encontrar una función glRenderCubeIBO(), que nos pinta un cubo de tamaño unidad centrado en el origen. Eso significa que las coordenadas tendrían que ser del tipo (0.5, -0.5, 0.5). Pero para no estar arrastrando con tanto decimal, es más fácil definirlo entre 0 y 1, y

luego trasladamos -0.5 en cada dimensión. Eso sí, entonces habrá que usar un bloque push/pop para borrar la traslación temporal de la ModelMatrix

```
function glRenderCubeIBO() {
 glPushMatrix();
   mat4.translate(modelMatrix, modelMatrix, [-0.5, -0.5, -0.5]);
   gl.uniformMatrix4fv(modelMatrixLoc, false, modelMatrix);

   // create vertices
   arrayV = new Float32Array([0,0,0, 1,0,0, 1,1,0, 0,1,0,
                              0,0,1, 1,0,1, 1,1,1, 0,1,1]);
   gl.bufferData(gl.ARRAY_BUFFER, arrayV, gl.STATIC_DRAW);

   // draw edges
   arrayI = new Uint16Array([0,1,1,2,2,3,3,0,
                             4,5,5,6,6,7,7,4,
                             0,4,1,5,2,6,3,7]);
   gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, arrayI,
gl.STATIC_DRAW);
   gl.uniform4fv(colorLocation, [0,0,0,1]);
   gl.drawElements(gl.LINES, 24, gl.UNSIGNED_SHORT, 0);
   delete arrayV;
   delete arrayI;
 glPopMatrix();
}
```

Aparte de esta función, hay que hacer otros dos cambios:

- En la función init, al llamar a gl.vertexAttribPointer, hay que poner 3 valores por vértice en lugar de 2

```
gl.vertexAttribPointer(coordLocation, 3, gl.FLOAT, false, 0, 0);
```

- En el vertex shader el vector de coordenadas pasa a ser de tipo vec3 en lugar de vec2, y cuando creemos el vec4 hay que usar las tres coordenadas

```
in vec3 aCoordinates;
gl_Position = uModelMatrix * vec4(aCoordinates, 1.0);
```

Y por último llamamos a la función para pintar el cubo. Recordemos también que antes y después habrá que activar y desactivar el buffer de índices

```
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, index_buffer);
glRenderCubeIBO();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, null);
```

Rotando la escena con el ratón

Vamos a usar los eventos de ratón para rotar

Añadiendo los handlers del ratón para obtener las coordenadas del puntero

Añadimos el evento mouse move y mouse down

```
// add mouse handlers
document.onmousedown = onMouseDown;
document.onmousemove = onMouseMove;
```

Y mostramos por pantalla la posición del puntero, sólo si el botón está apretado

```
  function onMouseDown(e) {
         if (e.buttons==1)
         console.log("down = ("+e.pageX+","+e.pageY+")");
      }
  function onMouseMove(e) {
         if (e.buttons==1)
         console.log("move = ("+e.pageX+","+e.pageY+")");
      }
```

Rotando la escena

Creamos un par de variables para almacenar el puntero cuando hacemos mousedown, y otras dos para ir calculalando el ángulo acumulado de rotación

```
var rotateX=0, rotateY=0;
var mouseX, mouseY;
```

Y ahora hacemos los evetos mousedown y mouse move

```
function onMouseDown(e) {
         if (e.buttons==1 && e.srcElement==canvas) {
           mouseX = e.pageX;
           mouseY = e.pageY;
         }
      }

function onMouseMove(e) {
         if (e.buttons==1 && e.srcElement==canvas) {
           rotateY = rotateY + (e.pageY - mouseY) * 0.01;
           //rotateX = ??
           mouseX = e.pageX;
           mouseY = e.pageY;
           //console.log("move = ("+mouseX+","+mouseY+")");
         }
      }
```

Por último, la rotación en el render queda así, justo antes de pintar el cubo

```
// rotate scene
mat4.rotateY(modelMatrix, modelMatrix, rotateY);
```

Mejor usar desplazamiento en X para rotar en Y

El movimiento vertical del ratón no parece lo más adecuado para rotar en Y. Mejor cambiamos el cálculo y usamos la diferencia del movimiento en X para calcular el ángulo a rotar en Y.

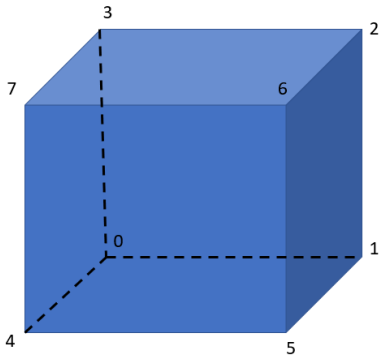
```
rotateY = rotateY + (e.pageX - mouseX) * 0.01;
```

Terminar la rotación completa en XY

Sólo nos falta añadir el cálculo de la variable rotateY, y multiplicar por la matrix de rotación en Y

Creando un cubo sólido + depth test

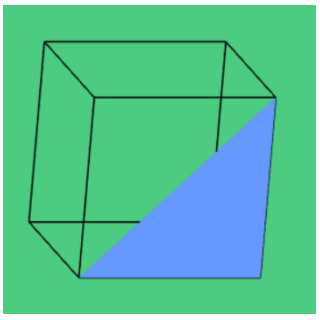
El anterior objeto era inalámbrico. Para crear un objeto sólido con sus caras pintadas, es mejor usar otra esctructura, que represente vértices y caras, y cada cara como uno o varios triángulos. De esta forma, como ya tenemos los vértices guardados, habrá que añadir más adelante los índices de las caras, que serían:




```
1,0,3,     1,3,2,   // cara trasera
4,5,6,     4,6,7,   // cara delantera
7,6,2,     7,2,3,   // cara superior
0,1,5,     0,5,4,   // cara inferior
5,1,2,     5,2,6,   // cara derecha
0,4,7,     0,7,3,   // cara izquierda
```

Añadimos nuevo vector de caras

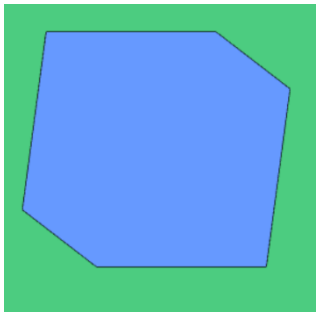
En nuestra función glRenderCubeIBO vamos a añadir un segundo vector para los índices de las caras. Este nuevo vector de índices que representarán las caras se pintarán con triángulos. También añadimos un color para las líneas y otro para los tríangulos



```
// draw faces
arrayF = new Uint16Array([4,5,6]);
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER,
   arrayF, gl.STATIC_DRAW);
gl.uniform4fv(colorLocation, [0.3,0.5,1,1]);
gl.drawElements(gl.TRIANGLES, 3,
   gl.UNSIGNED_SHORT, 0);
```

La figura muestra con un solo triángulo. Habría que terminar de meter los 12 triángulos que forman el cubo completo (36 vértices en la llamada a gl.drawElement)

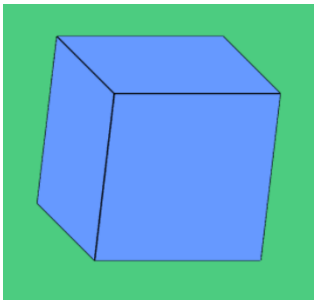
Sin embargo, se aprecia una cosa rara, y es porque las caras se están pintando sobre el frame buffer en el orden que se mandan en el código. Y por tanto, las caras azules se pintan después de las aristas



Añadiendo depth test

Para arreglar esto, simplemente habilitaremos el test de profundidad, con la llamada siguiente en cualquier parte del init

```
gl.enable(gl.DEPTH_TEST);
```



Y luego resetear el depth buffer en cada render, al igual que lo hacemos con el frame_buffer:

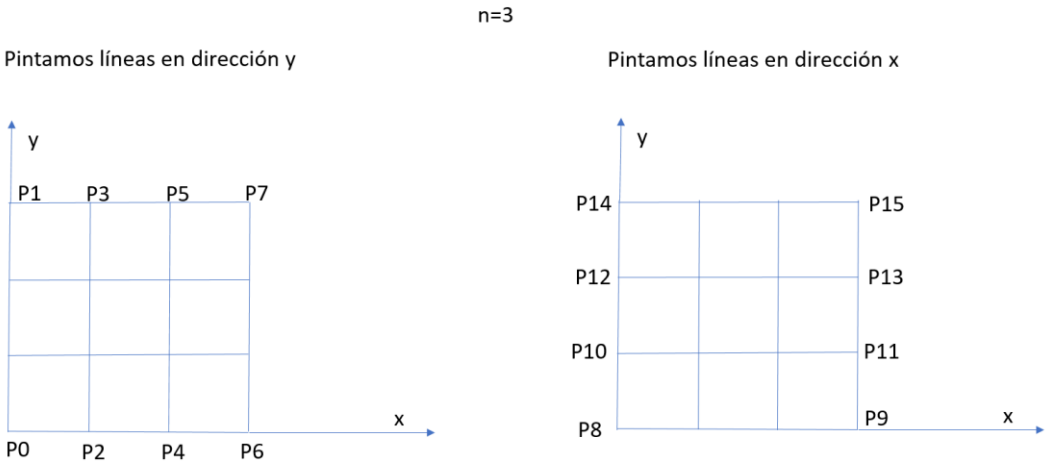
```
gl.clear(gl.COLOR_BUFFER_BIT |
gl.DEPTH_BUFFER_BIT);
```

Otra cosa adicional es invertir el signo de la z en el vertex shader, después de calcular la posición. Ya lo explicaremos luego

```
gl_Position.z *= -1.0;
```

Añadimos un suelo

Vamos a añadir un cuadrado a modo de suelo, para comprobar que estamos en una proyección paralela. Le pasaremos el tamaño del suelo (size) y el número de líneas (n); ambos valores iguales para X y para Y. Usaremos la primitiva gl_lines, y no necesitaremos índices. Simplemente haremos líneas largas en las dirección x e y



El código de la función glRenderGround(size, n) ya va incluido en el ejemplo.

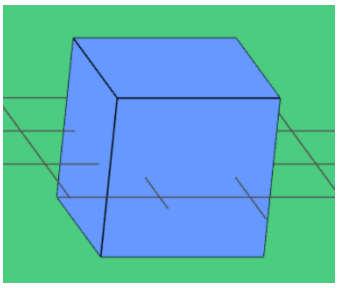
```
// draw squared floor
 function glRenderGround(size, n) {
   glPushMatrix();
     mat4.scale(modelMatrix, modelMatrix, [size, size, size]);
     mat4.translate(modelMatrix, modelMatrix, [-0.5, 0, -0.5]);
     gl.uniformMatrix4fv(modelMatrixLoc, false, modelMatrix);

     // creamos vector vértices
     k = 0;
     arrayV = new Float32Array(12*n);
     for (i = 0; i < n; i++) {
```

```
      arrayV[k++] = i/(n-1);
      arrayV[k++] = 0;
      arrayV[k++] = 0;
      arrayV[k++] = i/(n-1);
      arrayV[k++] = 0;
      arrayV[k++] = 1;
    }
    for (i = 0; i <= n; i++) {
      arrayV[k++] = 0;
      arrayV[k++] = 0;
      arrayV[k++] = i/(n-1);
      arrayV[k++] = 1;
      arrayV[k++] = 0;
      arrayV[k++] = i/(n-1);
    }
    gl.bufferData(gl.ARRAY_BUFFER, arrayV, gl.STATIC_DRAW);
    gl.drawArrays(gl.LINES, 0, 4*n);
    delete arrayV;
  glPopMatrix();
 }
```

Y luego llamamos a esta función después de pintar el cubo.

```
// drawGround
glRenderGround(5, 10);
```



Ahora mismo se ve todo muy grande. Y además el suelo se corta por delante y por detrás debido al recorte con el volumen normalizado –(1,1) de la escena. Lo mejor es hacer más pequeño toda la escena, y además poder hacer zoom con el ratón.

Añadiendo zoom

Para poder hacer zoom en la escena con la rueda del ratón, vamos a añadir el listener, y habrá una variable zoomFactor que iremos escalando en función del sentido de la rueda del ratón, y haciendo un scale de toda la escena junto con las rotaciones del ratón, al principio de cada render.

Primero nos declaramos una variable global que guarde el zoom a aplicar:

```
var zoomFactor = 1;
```

A continuación escribimos la función zoom:

```
function zoom(e) {
  if (e.deltaY<0)
    zoomFactor *= 1.1;
  else
    zoomFactor *= 0.9;
}
```

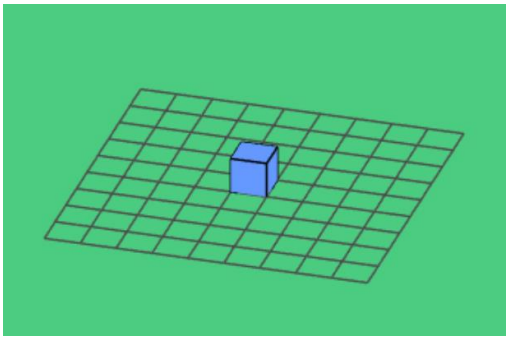
Y la añadimos al listener:

```
document.onwheel = zoom;
```

Y luego hacemos el escalado:

```
// transformations from GUI and mouse
mat4.scale(modelMatrix, modelMatrix, [zoomFactor, zoomFactor,
zoomFactor]);
mat4.rotateX(modelMatrix, modelMatrix, rotateX);
mat4.rotateY(modelMatrix, modelMatrix, rotateY);
```

Ahora se ve guay. También podemos subir un pizco el cubo para que esté apoyado sobre el suelo. Es decir, en el método renderCubeIBO, en la función de trasladar que ya teníamos, dejamos la Y sin trasladar (recordemos que las coordenadas de los vértices estaban entre 0 y 1, y luego trasladábamos -0.5 en todas las dimensiones). La nueva llamada quedaría así:



```
mat4.translate(modelMatrix,
modelMatrix, [-0.5, 0, -0.5]);
```

La figura muestra con size 9 y n=10

Añadiendo perspectiva

Para que se note el efecto de la perspectiva, vamos a aplicar la transformación que vimos en teoría.

Perspectiva básica

Ya la librería gl-matrix.js incluye la función mat4.perspective que nos multiplica por la matriz de perspectiva. Los parámetros son ángulo de visión, ratio del canvas, plano znear y plano zfar.

Pero normalmente la matriz de proyección se guarda en una matriz diferente a la modelMatrix, llamada viewMatrix. Así que debemos modificar el vertex shader para que lea dos matrices:

```
uniform mat4 uModelMatrix;
uniform mat4 uViewMatrix;

void main(void) {
  gl_Position = uViewMatrix * uModelMatrix *
        vec4(aCoordinates, 1.0);
```

Y luego tenemos que crear la variable viewMatrixLoc, y asociarla con la variable del shader, usando esta instrucción:

```
var viewMatrixLoc;
…
viewMatrixLoc = gl.getUniformLocation(shaderProgram,
"uViewMatrix");
```

Por último, crearemos una matriz vacía, y usaremos el método perspective del objeto mat4 para crear una matriz de perspectiva. Esto lo hacemos con el siguiente código, añadiéndolo después de haber metido la identidad en la matriz modelmatrix, pero antes de las rotaciones del ratón

```
// perspective
viewMatrix = mat4.create();
mat4.perspective(viewMatrix,
      Math.PI/4, // vertical opening angle
      1,         // ratio width-height
      1,         // z-near
      30         // z-far
);
gl.uniformMatrix4fv(viewMatrixLoc, false, viewMatrix);
```

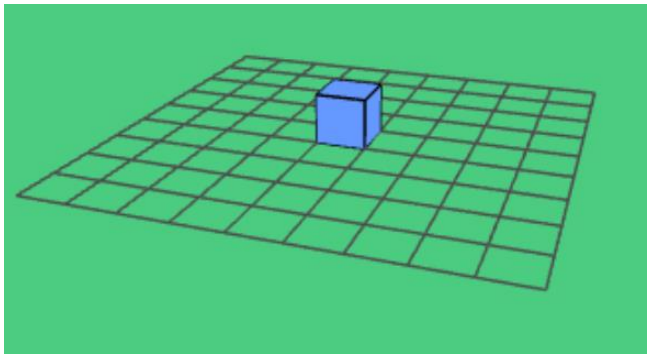
Sin embargo, ahora deberíamos ver sólo la línea del suelo, y nada del cubo. El problema es que el observador está en el (0,0,0), justo dentro del cubo!!!

Separar al observador de la escena

Para poder arreglar esto debemos alejar al observador de la escena. Que es equivalente a trasladar toda la escena hacia atrás en la dirección z. Para ello añadimos este translate a la modelMatrix, antes del resto de transformaciones

```
mat4.translate(modelMatrix, modelMatrix, [0,0,-10]);
```

Ahora sí se aprecia la diferencia entre las proyecciones paralela y perspectiva



OJO: al usar la perspectiva la vista vuelve a ser hacia la z negativa. Ya no hará falta invertir la z de los vértices en el shader.

Caminando por la escena

Hasta ahora la proyección siempre es en el origen. Ahora vamos a hacer que podamos mover la cámara libremente por la escena.

La transformación de vista

Lo que hicimos antes de trasladar la escena 10 unidades en z para probar la perspectiva fue algo temporal. Lo correcto es que la escena esté donde sea, y nosotros digamos dónde va el observador, y hacia dónde está mirando. Lo correcto sería hacer toda la transformación de vista:

- Trasladar el observador al origen (y con él toda la escena)

- Reorientar el sistema de coordenadas para que la dirección de vista pase a ser el eje z, y la dirección vertical el eje y
- Hacer la perspectiva

Por tanto, si sustituimos la llamada a trasladar 10 en z por un LookAt como éste:

```
mat4.lookAt(modelMatrix,
      [0,0,10], // camera position
      [0,0,0],  // Point at which the camera is looking
      [0,1,0]   // up vector
);
```

sería todo exactamente igual. Podemos ir variando los parámetros (eye, center, up), y veríamos cómo va cambiando la vista. Con esto ya podríamos caminar con teclas por la escena. Vamos a hacerlo

Preparando la escena para movernos por ella

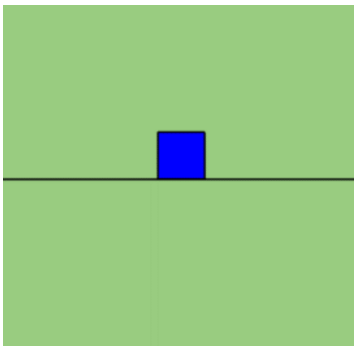
Vamos a hacer varias cosas previas

- Pondremos el suelo aún más grande, con size 15 y n=15.
- Deshabilitamos los movimientos del ratón y su rueda. Ahora sólo nos moveremos con las flechas (se comenta la inicialización del callback y mousemove)
- Creamos un objeto player de la siguiente manera

```
var player = {
  x: 0, y: 0, z: 10,
};
```

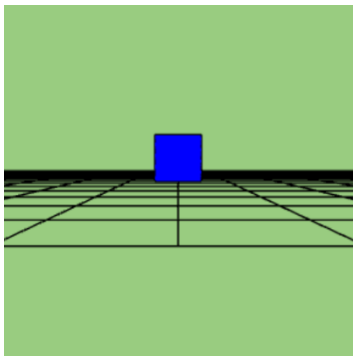
Donde guardamos su posición, que por defecto está en el (0,0,10). Y la llamada al lookAt será así:

```
eye = [player.x, player.y, player.z];
mat4.lookAt(modelMatrix, eye, [0,0,0],
[0,1,0]);
```



Es decir, fijamos el observador en la posición del player, y miramos al origen de coordenadas, a la misma altura que el player. En este caso, el suelo solo lo estaremos viendo de perfil.

Para levantarlo un poco, vamos a darle una altura de 0.5 metros, y que además el punto central al que mire esté también a la misma altura (para que mire en dirección horizontal al suelo). Ahora sí parece más normal



Moviendo la cámara por la escena

Ahora necesitamos añadir los callback de las teclas, como en la primera práctica. Vamos a hacer que al mover la tecla parriba (palante) avanzamos en z negativo, y patrás positivo. Y también izquierda y derecha movemos la x

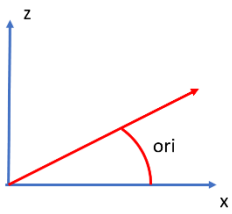
```
document.onkeydown = onKeyDown;

function onKeyDown(key) {
  switch (key.keyCode) {
    // up arrow
    case 38: {
      player.z -= 0.1;
      break;
    }
    // down arrow
    case 40: {
      player.z += 0.1;
      break;
    }
    // left arrow
    case 37: {
      player.x -= 0.1;
      break;
    }
    // down arrow
    case 39: {
      player.x += 0.1;
      break;
    }
  }
}
```

Ahora nos podemos mover libremente por el suelo, pero siempre tenemos la vista centrada en el cubo, porque hemos puesto el origen como punto a mirar en el lookAt. Para cambiar esto vamos a añadir un campo de orientación al player.

Girando la dirección de vista

Añadimos un campo ori al player, y lo inicializamos a –pi/2 (porque está mirando en la dirección z negativa). Y lo que queremos conseguir es que con las teclas izquierda y derecha rotes la vista, y con las teclas adelante y atrás camines en esa dirección.

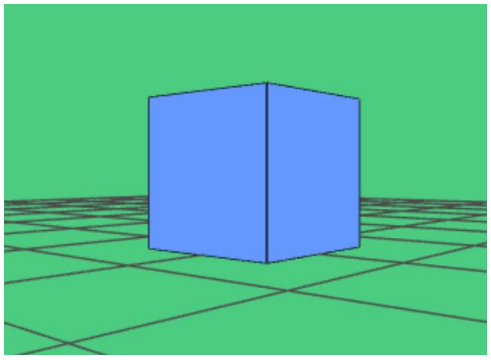


Entonces, el código a LookAt quedaría así (OJO que ahora el punto al que mira es v):

```
eye = [player.x, player.y, player.z];
v = [player.x + Math.cos(player.ori), player.y, player.z +
Math.sin(player.ori)];
mat4.lookAt(modelMatrix, eye, v, [0,1,0]);
```

Y el de las teclas sería así

```
function onKeyDown(key) {
  switch (key.keyCode) {
    // up arrow
    case 38: {
      player.x = player.x + 0.1*Math.cos(player.ori);
      player.z = player.z + 0.1*Math.sin(player.ori);
      break;
    }
    // down arrow
    case 40: {
      player.x = player.x - 0.1*Math.cos(player.ori);
      player.z = player.z - 0.1*Math.sin(player.ori);
      break;
    }
    // left arrow
    case 37: {
      player.ori -= 0.02;
      break;
    }
    // down arrow
    case 39: {
      player.ori += 0.02;
      break;
    }
  }
}
```



Ahora nos podemos mover libremente por el escenario.

Posibles ampliaciones

- Añadir más objetos al escenario
- Que alguno de esos objetos tenga algún movimiento prefijado
- Que alguno de los objetos te persiga
- Poder disparar balas
- Poder volar por el escenario
- Cambiar el modo de movimiento para acelerar y frenar (puedes derrapar con el vector up)