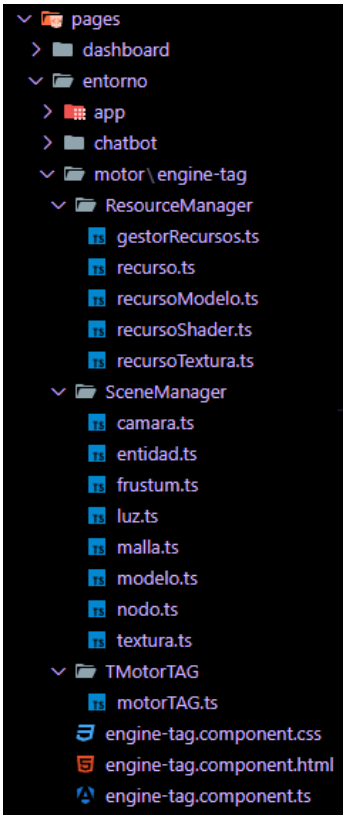
TAG

Documento creado por TDT.

Índice de contenidos

1. Estructura y lógica

La estructura de archivos para nuestro motor está montada con carpetas de esta manera:



Hemos dividido nuestro motor en 3 carpetas:

- **/ResourceManager:** esta carpeta contiene todo lo relacionado con el gestor de recursos, que se explicará más adelante
- **/SceneManager:** esta carpeta contiene todo lo relacionado con la gestión de la escena: la cámara, las clases de nodo y entidad, la iluminación, los modelos con sus mallas y texturas y además, el código necesario para hacer el frustum culling, que también se explicará más adelante
- **/TMotorTAG:** en esta carpeta únicamente se incluye la clase de nuestro motor gráfico, para tenerlo bien diferenciado del resto de elementos

Por último, mencionar que todo lo relacionado con la interfaz se encuentra en los archivos engine-component, generados con Angular, que es el “main” de nuestra aplicación web.

En el archivo html únicamente está el canvas, que ocupa toda la pantalla. En el archivo .ts, está todo el funcionamiento general, como inicializar el motor y obtener el webglcontext necesario para el uso del motor. Esto se explicará en el apartado de interfaz.

Para el árbol de escena, se ha seguido el diagrama explicado en teoría, de forma que creamos un árbol de escena y éste cuenta con los nodos de iluminación, cámara y un los nodos modelo que vayan creándose en la escena. Todas las entidades cuentan con el método de dibujado, que es abstracto y cada tipo de nodo tiene un override, puesto que no es lo mismo dibujar una luz que una cámara o un modelo con sus mallas.

```
export abstract class TEntidad {

    abstract dibujar(
        gl: WebGL2RenderingContext,
        texture?:any,
        textureColor?:any,
        camara?:any,
        matrizTransformacion?: any
    ): void;
}
```

Utilizando el '?' conseguimos que se use o no un elemento en el constructor. Por ejemplo, una luz no tiene texturas, obviamente.

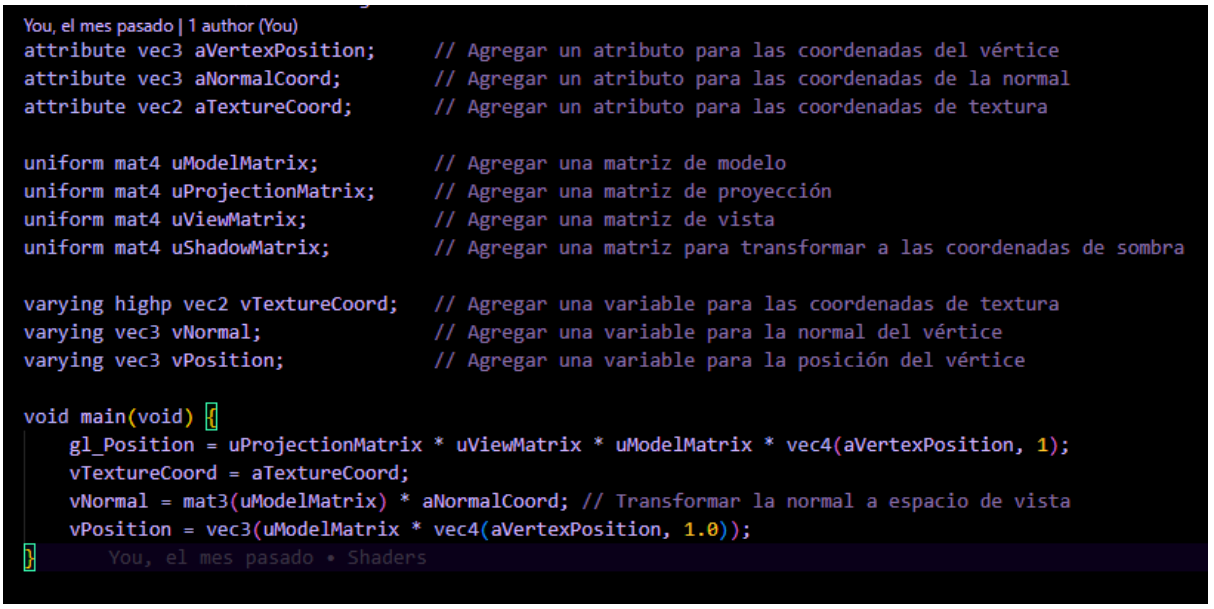
Para el dibujado del motor, se ha usado el método recursivo recorrer tal y como se ha explicado en teoría. Cada vez que se dibuja el motor, simplemente se llama a recorrer y dibujará todos sus elementos, así:

```
public dibujar() {
    if (!this.camara) {
        console.error("La cámara no está definida, no se puede dibujar la escena.");
        return;
    }

    const projectionMatrix = this.camara.getProjectionMatrix();
    const viewMatrix = this.camara.getViewMatrix();
    const frustum = new Frustum(projectionMatrix, viewMatrix);

    this.webglCTX.clear(this.webglCTX.COLOR_BUFFER_BIT | this.webglCTX.DEPTH_BUFFER_BIT);

    this.arbolEscena.recorrer(this.webglCTX, (nodo, gl) => {
        const entidad = nodo.getEntidad();
        if (entidad instanceof TModelo) {
            const min = vec3.fromValues(-1.0, -1.0, -1.0); // Ajusta estos valores
            const max = vec3.fromValues(1.0, 1.0, 1.0);    // Ajusta estos valores
            const worldMatrix = entidad.getWorldMatrix();
            vec3.transformMat4(min, min, worldMatrix);
            vec3.transformMat4(max, max, worldMatrix);

            if (frustum.isBoxInFrustum(min, max)) {
                entidad.dibujar(gl);
            }
        }
    });
}
```

2. Shaders de nuestro motor

Para poder hacer la selección de modelos hemos implementado el color picking, y para ello, hemos creado dos shaders: uno principal (que incluye las texturas e iluminación) y otro simplemente con un uniform para el color. El código de los vertex y fragment de cada shader es el siguiente:

- **VertexShader principal**



- **FragmentShader principal**

```
precision highp float;                  // Precisión de punto flotante

varying highp vec2 vTextureCoord;       // Coordenadas de textura
varying vec3 vNormal;                   // Normal del fragmento
varying vec3 vPosition;                 // Posición del fragmento

uniform sampler2D uSampler;             // Textura
uniform vec3 uLightDirection;           // Dirección de la luz
uniform vec3 uLightColor;               // Color de la luz
uniform float uLightIntensity;          // Intensidad de la luz
uniform vec3 uCameraPosition;           // Posición de la cámara

varying vec2 vDepthUv;                  // Coordenadas de profundidad
uniform sampler2D depthColorTexture;    // Textura de profundidad

uniform bool uIsWall;

void main(void) {
    vec4 textureColor = texture2D(uSampler, vTextureCoord);

    if (uIsWall) {
        gl_FragColor = textureColor;
    } else {
        vec3 lightDirection = normalize(uLightDirection);
        float diffuse = max(dot(vNormal, lightDirection), 0.0);

        vec3 viewDirection = normalize(uCameraPosition - vPosition);
        vec3 reflectionDirection = reflect(-lightDirection, vNormal);
        float specular = pow(max(dot(viewDirection, reflectionDirection), 0.0), 5.0); // El segundo argumento es el brillo especular

        vec3 lightColor = vec3(uLightColor);
        vec3 diffuseColor = lightColor * diffuse * uLightIntensity;
        vec3 specularColor = lightColor * specular * uLightIntensity; // Aplica la intensidad a la luz especular

        vec4 finalColor = textureColor * vec4(diffuseColor + specularColor, 1.0) ; // Agrega la luz especular a la final

        gl_FragColor = finalColor;
    }
}
```

- **VertexShader de color**

```
#version 300 es
layout(location = 0) in vec4 VertexPosition;

uniform mat4 uModelMatrix;
uniform mat4 uViewMatrix;
uniform mat4 uProjectionMatrix;

void main()
{
    mat4 mvp = uProjectionMatrix * uViewMatrix * uModelMatrix;
    gl_Position = mvp * VertexPosition;
}
```

- **FragmentShader de color**

```
#version 300 es
precision highp float;                  // Precisión de punto flotante

uniform vec4 uColor; // Color de la malla
out vec4 fragColor;

void main(void) {
    fragColor = uColor;
}
```

Para crear los programas (necesitaremos dos gl programs) desde el motor creamos un recurso shader que cargará el contenido de los ficheros glsl correspondientes. El orden es este:

1. Creación del recurso shader: Extiende de la clase TRecurso, que es abstracta. Tiene los siguientes atributos:

```
export class TRecursoShader extends TRecurso {

    // Vertex shader y Fragment shader como strings
    private vertexShaderMainSource: string = '';
    private fragmentShaderMainSource: string = '';

    private vertexShaderColorSource: string = '';
    private fragmentShaderColorSource: string = '';

    private gl!: WebGL2RenderingContext;
    public loadedPromise: Promise<void>;

    // Programas
    private shaderProgramMain!: WebGLProgram | null;
    private shaderProgramColorSelect!: WebGLProgram | null;

    constructor(
        rutaVertexMainShader: string,
        rutaFragmentMainShader: string,
        rutaVertexColorShader: string,
        rutaFragmentColorShader: string,
        context: WebGL2RenderingContext
    ) {
```

2. Carga de los ficheros glsl en el constructor: Tenemos un método asíncrono para la lectura de ficheros en typescript. En el constructor se cargan los source usando este método:

```
override cargarFichero(filename: string): Promise<string> {
    return new Promise((resolve, reject) => {
        fetch(`/assets/Shaders/${filename}`)
            .then(response => {
                if (!response.ok) {
                    throw new Error('Network response was not ok');
                }
                return response.text();
            })
            .then(data => {
                resolve(data);
            })
            .catch(error => {
                console.error("Error al cargar el archivo:", error);
                reject(error);
            });
    });
}
```

3. Carga de shaders. Ahora que los source han sido leídos correctamente, se generan los shaders.

```
// Cargar shaders y devolverlo
private CargarShader(type: any, sourceGLSL: string): any {
    const shader = this.gl.createShader(type);
    if (shader) {
        this.gl.shaderSource(shader, sourceGLSL);
        this.gl.compileShader(shader);
        if (!this.gl.getShaderParameter(shader, this.gl.COMPILE_STATUS)) {
            console.log("Error al compilar shader: " + this.gl.getShaderInfoLog(shader))
            this.gl.deleteShader(shader);
            return null;
        }

    }
    return shader;
}
```

4. Crear los programas. Creamos dos programas, uno principal y otro para dibujar las mallas con color de la siguiente manera:

```
private async InitShaders(gl: WebGL2RenderingContext): Promise<any>  {
    // Codigo GLSL
    await this.loadedPromise;
    const vertexShader = this.CargarShader(this.gl.VERTEX_SHADER, this.vertexShaderMainSource);
    const fragmentShader = this.CargarShader(this.gl.FRAGMENT_SHADER, this.fragmentShaderMainSource);

    const vertexShaderColor = this.CargarShader(this.gl.VERTEX_SHADER, this.vertexShaderColorSource);
    const fragmentShaderColor = this.CargarShader(this.gl.FRAGMENT_SHADER, this.fragmentShaderColorSource);

    // Crear shader program main
    this.shaderProgramMain = gl.createProgram();
    if (this.shaderProgramMain !== null && vertexShader && fragmentShader) {
        gl.attachShader(this.shaderProgramMain, vertexShader);
        gl.attachShader(this.shaderProgramMain, fragmentShader);
        // Continúa con el resto del código aquí
        gl.linkProgram(this.shaderProgramMain);

        if (!gl.getProgramParameter(this.shaderProgramMain, gl.LINK_STATUS)) {
            alert(
                "Unable to initialize the shader program: " +
                gl.getProgramInfoLog(this.shaderProgramMain)
            );
            return null;
        }
    } else {
        throw new Error('No se pudo crear el programa de shaders');
    }
```
Ver ejemplos reales de GitHub

Dado que necesitamos tener acceso a los programas en todo momento, se han creado métodos aquí para poder acceder a un programa u otro o establecerlo, necesario para cambiar entre shader principal o de color.

```
public getShaderProgramMain(): WebGLProgram | null {
    return this.shaderProgramMain;
}

public getShaderProgramColorSelect(): WebGLProgram | null {
    return this.shaderProgramColorSelect;
}

public useProgramMain(gl: WebGL2RenderingContext): void {
    if (gl && this.shaderProgramMain) {
        gl.useProgram(this.shaderProgramMain);
    } else {
        console.error('No se puede usar el programa principal de shaders porque no se ha inicializado.');
    }
}

public useProgramColorSelect(gl: WebGL2RenderingContext): void {
    if (gl && this.shaderProgramColorSelect) {
        gl.useProgram(this.shaderProgramColorSelect);
    } else {
        console.error('No se puede usar el programa de selección de color porque no se ha inicializado.');
    }
}
```

Además, hemos hecho uso de un servicio con una instancia usando el patrón Singleton para acceder a este recurso shader.

```
export class ShaderService {
    private static instance: ShaderService | null = null;
    recursoShader: TRecursoShader | null = null;

    private constructor() { }

    static getInstance(): ShaderService {
        if (!this.instance) {
            this.instance = new ShaderService();
        }
        return this.instance;
    }

    setRecursoShader(       You, hace 2 semanas • Casi color picking
        rutaVertexMainShader: string,
        rutaFragmentMainShader: string,
        rutaVertexColorShader: string,
        rutaFragmentColorShader: string,
        context: WebGL2RenderingContext
    ) {
        this.recursoShader = new TRecursoShader(
            rutaVertexMainShader,
            rutaFragmentMainShader,
            rutaVertexColorShader,
            rutaFragmentColorShader,
            context
        );
    }

    getRecursoShader(): TRecursoShader | null {
        return this.recursoShader;
    }
}
```

Finalmente, desde el motor,

```
async init() {
    ShaderService.getInstance().setRecursoShader(
        "vertexMain.glsl",
        "fragmentMain.glsl",
        "vertexColorSelect.glsl",
        "fragmentColorSelect.glsl",
        this.webglCTX
    );
    this.recursoShader = ShaderService.getInstance().getRecursoShader();
    if (this.recursoShader !== null) {
        await this.recursoShader.loadedPromise;
    } else {
        console.error('recursoShader es null');
    }

    if (!this.webglCTX) {
        console.error("No se pudo inicializar el contexto WebGL.");
        return;
    }
}
```

2.1. Iluminación

Para crear iluminación en la escena, hemos seguido el cálculo de iluminación de Phong, utilizando una luz ambiental, difusa y especular (con brillo especular establecido directamente en el shader). Esto se puede apreciar en el fragment shader principal de la imagen de arriba. Después, en la clase Luz, creamos un nodo con los parámetros necesarios. En la siguiente imagen se muestra el código de dibujado de la luz en la escena.

```
override dibujar(gl: WebGL2RenderingContext): void {
    // Obtén el programa de shaders actualmente en uso
    let shaderProgram = gl.getParameter(gl.CURRENT_PROGRAM);
    // Obtén las ubicaciones de las variables uniformes
    let uLightDirectionLocation = gl.getUniformLocation(shaderProgram, "uLightDirection");
    let uLightColorLocation = gl.getUniformLocation(shaderProgram, "uLightColor");
    let uLightIntensityLocation = gl.getUniformLocation(shaderProgram, "uLightIntensity");

    // Establece los valores de las variables uniformes
    gl.uniform3fv(uLightDirectionLocation, this.getLightDirection());
    gl.uniform3fv(uLightColorLocation, this.getLightColor());
    gl.uniform1f(uLightIntensityLocation, this.getLightIntensity());
}
```

Para crear e iniciar la luz, desde el motor tenemos un método que se encarga de crear la luz y asignar los parámetros necesarios. Nuestra luz es blanca, direccional desde arriba:

```
public InitLights(gl: WebGL2RenderingContext){
    let lightDirection = [0.0, 1.0, 0.0]; // Luz desde arriba
    let lightColor = [1.0, 1.0, 1.0]; // Color de la luz
    let lightIntensity = 0.9; // Intensidad de la luz (0.0 a 1.0)

    this.luzEscena = new TLuz(lightDirection, lightColor, lightIntensity);
    this.luzEscena.dibujar(gl);
}
```

2.2. Texturas

Para las texturas, se ha creado la clase TRecursoTextura con funcionamiento muy parecido al del TRecursoShader. Cada modelo tiene una textura única asignada, y ésta se cargará en las mallas. El funcionamiento correcto se logra con estos pasos:

1. Carga asíncrona de una imagen: como cada textura se carga con una imagen jpg, basándonos en la carga de ficheros ya creada para los shaders, la aplicamos para un objeto de tipo blob. El código es este:

```
// Cargar la textura como fichero con fetch
override cargarFichero(texture: string): Promise<Blob> {    // Para la
    // Atributos
    return new Promise((resolve, reject) => {
        fetch(`/assets/3D_Models/Texturas/${texture}.jpg`)
            .then(response => {
                if (!response.ok) {
                    throw new Error('Network response was not ok');
                }
                return response.blob();
            })
            .then(blob => {
                resolve(blob);
            })
            .catch(error => {
                console.error("Error al cargar el archivo:", error);
                reject(error);
            });
    });
}
```

2. Crear la textura con la imagen jpg: En este método se devuelve la textura que se usará en las mallas. A continuación se presenta el código.

```
// Cargar la textura en el contexto WebGL
override async cargarTextura(gl: WebGL2RenderingContext, filename: string){
    // Cargar textura
    const texture = gl.createTexture();
    gl.bindTexture(gl.TEXTURE_2D, texture);

    // Atributos de la textura
    const level = 0;                    // Nivel de detalle de la imagen. 0 es el nivel base
    const internalFormat = gl.RGBA;     // Formato de los componentes de textura
    const width = 1;                    // Anchura de la textura
    const height = 1;                   // Altura de la textura
    const border = 0;                   // Siempre 0
    const srcFormat = gl.RGBA;          // Formato de los datos de textura
    const srcType = gl.UNSIGNED_BYTE;   // Tipo de los datos de textura
    const pixel = new Uint8Array([10, 10, 10, 10]);     // Color azul

    gl.texImage2D(gl.TEXTURE_2D, level, internalFormat, width, height, border, srcFormat, srcType, pixel);

    function isPowerOf2(value:any) {      You, hace 2 semanas • Casi color picking
        return (value & (value - 1)) === 0;
    }

    const image = new Image();
    this.cargarFichero(filename).then((blob) => {

        image.crossOrigin = "anonymous";
        image.onload = function () {
            gl.bindTexture(gl.TEXTURE_2D, texture);
            gl.texImage2D(gl.TEXTURE_2D, level, internalFormat, srcFormat, srcType, image);

            // Verificar si las dimensiones son potencias de 2
            if (isPowerOf2(image.width) && isPowerOf2(image.height)) {
                gl.generateMipmap(gl.TEXTURE_2D);
            } else {
                // No es potencia de 2, desactivar mipmapping y establecer el env de textura a CLAMP_TO_EDGE
                gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE);
                gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.CLAMP_TO_EDGE);
                gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
            }
        };
        image.src = URL.createObjectURL(blob);
    });
    return texture;
}
```

El funcionamiento de todo el método es el siguiente: Primero, crea una nueva textura en WebGL y la vincula al objetivo TEXTURE_2D.

3. Define los atributos de la textura, incluyendo el formato interno (RGBA), las dimensiones (1x1), el formato de origen (RGBA) y el tipo de origen (UNSIGNED_BYTE). Luego, carga un pixel azul como textura por defecto utilizando texImage2D.

4. Define una función isPowerOf2 que verifica si un número es potencia de 2. Esta función se utilizará más tarde para verificar las dimensiones de la imagen.

5. Crea una nueva imagen y utiliza la función cargarFichero para cargar el archivo especificado en filename como un Blob.

6. Cuando la imagen se carga (en el evento onload), la vincula a la textura y la carga en WebGL utilizando texImage2D.

7. Verifica si las dimensiones de la imagen son potencias de 2 utilizando la función isPowerOf2. Si es así, genera un mipmap para la textura utilizando generateMipmap. Los mipmaps son versiones de la textura a diferentes resoluciones que se utilizan para mejorar el rendimiento y la apariencia de las texturas cuando se visualizan a diferentes distancias.

8. Si las dimensiones de la imagen no son potencias de 2, ajusta los parámetros de la textura para desactivar el mapeo mip y establecer el envoltorio de textura a CLAMP_TO_EDGE, y el filtro minificador a LINEAR. Esto evita artefactos visuales que pueden ocurrir cuando se utiliza una textura cuyas dimensiones no son potencias de 2.

Por último, en el dibujado de las mallas, se agregó un condicional para controlar con GL_CURRENT_PROGRAM qué programa de los dos (color o principal) se está utilizando. Como las texturas se aplican en el shader principal, usamos este código para que la textura generada en el código previo se aplique correctamente:

```
if (this.shaderProgramMain === shaderProgram) {
    this.InitBuffersMain(gl, this.buffers);

    gl.activeTexture(gl.TEXTURE0);
    gl.bindTexture(gl.TEXTURE_2D, texture);
    gl.uniform1i(gl.getUniformLocation(shaderProgram, "uSampler"), 0);

} else if (this.shaderProgramColorSelect === shaderProgram) {

    this.InitBuffersColor(gl, this.buffers);
    let colorLocation = gl.getUniformLocation(shaderProgram, 'uColor');
    gl.uniform4fv(colorLocation, colorModelo);
}
```

Lean Canvas permite definir y visualizar los aspectos fundamentales del proyecto de una manera clara y concisa. Se proporciona a continuación el esquema realizado donde se explican de forma breve los elementos del mismo y posteriormente se detallarán.

2.3. Cámara

Existe una única cámara que toma como parámetro movimiento orbital o movimiento libre. Está implementado pero sin utilizar (ya que las teclas finalmente no se usan) que con el teclado se pueda cambiar entre un movimiento u otro. Con la tecla ‘R’ se podría resetear la posición de la cámara para que vuelva a su estado original:

```
resetear_mov_camara(): void {
    this.camaraMotor.setPosition(vec3.fromValues(12, 5, -12));
    this.camaraMotor.setLookAt([0, 0, 0]);
    this.camaraMotor.setUp([0, 1, 0]);
}
```

La cámara se encuentra en el archivo camara.ts y contiene toda la funcionalidad necesaria para la misma en la escena. Su constructor y parámetros necesarios es este:

```
You, hace 2 semanas | 2 authors (You and others)
export class TCamara extends TEntidad {
    private esPerspectiva: boolean;
    private cercano: number;
    private lejano: number;
    private angle: number = 0;
    private fov: number; // Campo de visión para la cámara perspectiva
    private aspectRatio: number; // Relación de aspecto para la cámara perspectiva

    private _projectionMatrix: mat4 = mat4.create();
    private _viewMatrix: mat4 = mat4.create();
    private _position: vec3 = vec3.create(); // Posición de la cámara
    private _lookAt: vec3 = vec3.create(); // Punto al que mira la cámara
    private _up: vec3 = vec3.fromValues(0, 1, 0); // Vector 'up'

    constructor(esPerspectiva: boolean, cercano: number, lejano: number, fov: number, aspectRatio: number) {
        super(); // Llama al constructor de TEntidad si es necesario
        this.esPerspectiva = esPerspectiva;
        this.cercano = cercano;
        this.lejano = lejano;
        this.fov = fov;
        this.aspectRatio = aspectRatio;

        this.calculateProjectionMatrix();
        this.calculateViewMatrix();
    }
```

Para inicializar la cámara, se crea desde el motor al inicializarlo:

```
public crearCamara(esPerspectiva: boolean, cercano: number, lejano: number, fov: number, aspectRatio: number): TCamara {
    this.camara = new TCamara(esPerspectiva, cercano, lejano, fov, aspectRatio); // Asignación a la propiedad de la clase

    this.camara.setPosition(vec3.fromValues(13, 6.5, -13));
    this.camara.setLookAt([0, 2, 0]);
    this.camara.setUp([0, 1, 0]);

    this.camara.calculateViewMatrix();

    return this.camara;
}
```

1. Crea una nueva instancia de TCamara con los parámetros proporcionados (esPerspectiva, cercano, lejano, fov, aspectRatio) y la asigna a this.camara.
2. Configura la posición de la cámara con this.camara.setPosition(vec3.fromValues(13, 6.5, -13)). Esto sitúa la cámara en las coordenadas (13, 6.5, -13).
3. Configura el punto al que la cámara está mirando con this.camara.setLookAt([0, 2, 0]). Esto hace que la cámara mire hacia el punto (0, 2, 0).
4. Configura el vector "up" de la cámara con this.camara.setUp([0, 1, 0]). Esto establece que el eje Y es "arriba" para la cámara.

5. Llama a this.camara.calculateViewMatrix() para calcular la matriz de vista de la cámara. Esta matriz se utiliza para transformar las coordenadas del mundo a las coordenadas de la cámara.
6. Finalmente, retorna la cámara creada y configurada.

3. Clase TMotor

Aunque ya se han mencionado en los apartados previos el dibujado mediante el método recorrer, la inicialización de luces y de cámara y del recurso shader con un service para tener acceso a los programas en todo momento, se puede destacar la cabecera y constructor del motor:

```
export class TMotorTAG {
    private arbolEscena!: TNodo; // Arbol de escena
    private gestorRecursos: TGestorRecursos = new TGestorRecursos(); // Gestor de recursos
    private webglCTX!: WebGL2RenderingContext; // Contexto WebGL
    private recursoShader!: TRecursoShader | null; // Recurso de shader
    private camara!: TCamara;
    private luzEscena!: TLuz;
    constructor(canvas: HTMLCanvasElement) {
        this.arbolEscena = new TNodo();
        this.gestorRecursos = new TGestorRecursos();
        this.webglCTX = canvas.getContext('webgl2', { preserveDrawingBuffer: true })!;
    }

    getWebglCTX(): WebGL2RenderingContext {
        return this.webglCTX;
    }
```

Como mención destacada, es importante hablar de que desde la interfaz se pasa el canvas HTML al motor y desde aquí se crea el webglcontext que será imprescindible en todas las operaciones relacionadas con WebGL. Además, para el funcionamiento correcto de colorPicking, el contexto WebGL se configura con la opción preserveDrawingBuffer establecida en true. Esto es esencial para permitir que el buffer de dibujo se conserve después de que se presenta, lo que es necesario para implementar esta técnica, que se explicará en el apartado de la interfaz.

4. Gestor de recursos

Para el gestor de recursos se ha utilizado lo visto en teoría. Desde el motor se crea e inicializada el gestor de recursos y se almacenan en él los recursos que se vayan creando. Se realiza de esta manera:

```
public getRecursoModelo(ID: string, filename: string) {
    let recModelo;
    recModelo = this.recursosModelo.find(recurso => recurso.getID() === ID);
    if (!recModelo) {
        recModelo = new TRecursoModelo();
        recModelo.setID(ID);
        recModelo.cargarFichero(filename);
        this.agregarRecurso(recModelo, "Modelo");
    } else {
        //console.log("Ya existe " + ID);
    }
    return recModelo;
}

public async getRecursoTextura(ID: string, filename: string) {
    let recTextura;
    recTextura = this.recursosTextura.find(recurso => recurso.getID() === ID);
    if (!recTextura) {
        recTextura = new TRecursoTextura();
        recTextura.setID(ID);
        await recTextura.cargarFichero(filename);
        this.agregarRecurso(recTextura, "Textura");
    }
    return recTextura;
}
```

Estas dos funciones, getRecursoModelo y getRecursoTextura, se utilizan para obtener un modelo o una textura respectivamente, dado un ID y un nombre de archivo.getRecursoModelo(ID: string, filename: string): Esta función busca en la lista de recursos de modelos (recursosModelo) un recurso con el ID proporcionado. Si no encuentra el recurso, crea uno nuevo, establece su ID, carga el archivo correspondiente y lo agrega a la lista de recursos de modelos. Finalmente, devuelve el recurso de modelo.

getRecursoTextura: Esta función es similar a getRecursoModelo, pero trabaja con recursos de textura. Busca en la lista de recursos de textura (recursosTextura) un recurso con el ID proporcionado. Si no

encuentra el recurso, crea uno nuevo, establece su ID, carga de forma asíncrona el archivo correspondiente y lo agrega a la lista de recursos de textura. Finalmente, devuelve el recurso de textura.

5. Dibujado

En este apartado se va a explicar todo lo relacionado con el dibujado de mallas en la escena

5.1 Carga de mallas en el recurso modelo

Una vez se ha cargado correctamente el recurso modelo, dibujamos el recurso con lo siguiente:

```
override dibujarRecurso(
    gl: WebGL2RenderingContext,
    texture: WebGLTexture,
    camara: any,
    colorModelo: any,
    matrizTransformacion?: mat4,
    isWall: boolean = false // Añadido
): void {
    if (this.mallas.length === 0) {
        console.log("Aún no se han cargado las mallas.");
        return;
    }

    if (!gl) {
        console.error('No se pudo obtener el contexto WebGL.');
        return;
    }

    // Renderizado normal
    for (let i: number = 0; i < this.mallas.length; i++) {
        this.mallas[i].dibujar(
            gl,
            texture,
            camara,
            colorModelo,
            matrizTransformacion,
            isWall // Añadido
        );
    }
}
```

Hay dos cosas destacables aquí. Por un lado, el boolean isWall para comprobar si el elemento a dibujar es una pared o no. En caso de que sí sea una pared, se aplicará frustum culling:

Por otro lado, dado que un modelo puede contener varias mallas, en un bucle for se recorren y se dibujan por individual.

5.2 Inicialización de buffers con webgl-obj-loader

Para empezar, una malla como tal, en su constructor solamente cuenta con el contenido del fichero .obj (modelo 3D).

```
export class TMalla extends TEntidad {
    private OBJfile: string;
    private buffers: any;

    private rotationAngleX: number = 0;
    private rotationAngleY: number = 0;

    private ID_malla: string = "";
    private color_malla: any;

    private recursoShader!: any;
    private shaderProgramMain!: WebGLProgram;
    private shaderProgramColorSelect!: WebGLProgram;

    constructor(OBJfile_param: string) {
        super();
        this.OBJfile = OBJfile_param;
        this.buffers = new OBJ.Mesh(this.OBJfile);
        this.recursoShader = ShaderService.getInstance().getRecursoShader();
        this.shaderProgramMain = this.recursoShader.getShaderProgramMain();
        this.shaderProgramColorSelect = this.recursoShader.getShaderProgramColorSelect();
    }
```

Sin embargo, cuenta con el objeto buffers, que se inicializan con la librería webgl-obj-loader (descargada con npm. Su página oficial es la siguiente: https://www.npmjs.com/package/webgl-obj-loader

Se ha seguido un tutorial para su uso. Lo primero que se hace es inicializar los buffers con la clase OBJ.Mesh con el contenido del fichero .obj cargado de forma asíncrona en el recurso del modelo. También accedemos Inicializa los buffers principales de un objeto 3D. Dado que existen dos programas, son necesarios dos métodos para la inicialización de buffers: uno para el programa principal y otro para el programa del color. La diferencia está en los atributos que utiliza cada uno de ellos.

Se ha hecho todo el proceso haciendo uso del tutorial de esta librería, y los métodos actúan de la siguiente manera:

1. **InitBuffersMain:**

- Comprueba si los buffers son nulos. Si lo son, el método se detiene y retorna.
- Llama a OBJ.initMeshBuffers para inicializar los buffers de la malla.
- Obtiene el programa actualmente en uso.
- Obtiene las ubicaciones de los atributos aVertexPosition, aTextureCoord y aNormalCoord en el programa principal.
- Habilita el atributo de posición de vértice y lo vincula al buffer de vértices. Luego, especifica cómo se deben interpretar los datos en el buffer de vértices.
- Hace lo mismo para el atributo de coordenadas de textura y el buffer de texturas.
- Hace lo mismo para el atributo de coordenadas normales y el buffer de normales.
- Finalmente, vincula el buffer de índices al contexto WebGL.
- Retorna los buffers.

2. **InitBuffersColor:**

- Comprueba si los buffers son nulos. Si lo son, el método se detiene y retorna.
- Llama a OBJ.initMeshBuffers para inicializar los buffers de la malla.
- Define la ubicación del atributo de posición de vértice como 0. Habilita el atributo de posición de vértice y lo vincula al buffer de vértices. Luego, especifica cómo se deben interpretar los datos en el buffer de vértices.
- Vincula el buffer de índices al contexto WebGL
- Retorna los buffers.

Como se puede apreciar, se hace uso de esta librería para este proceso en concreto. Aquí se ve el principio del método de inicialización principal en código. El de color funciona igual pero para los

atributos en específico del color.

```
private InitBuffersMain(gl: WebGL2RenderingContext, buffers: any) {
    if (!buffers) return; // Verificar que los buffers no sean null
    OBJ.initMeshBuffers(gl, buffers);

    let shaderProgram = gl.getParameter(gl.CURRENT_PROGRAM);
    let vertexPositionAttributeLocation = gl.getAttribLocation(shaderProgram, "aVertexPosition");
    let textureCoordAttributeLocation = gl.getAttribLocation(shaderProgram, "aTextureCoord");
    let normalCoordAttributeLocation = gl.getAttribLocation(shaderProgram, "aNormalCoord");

    gl.enableVertexAttribArray(vertexPositionAttributeLocation);
    gl.bindBuffer(gl.ARRAY_BUFFER, buffers.vertexBuffer);
    gl.vertexAttribPointer(
        vertexPositionAttributeLocation,
        buffers.vertexBuffer.itemSize,
        gl.FLOAT,
        false,
        0,
        0
    );      You, hace 2 meses • Modificando malla
```

5.3 Dibujado de las mallas

Una vez los buffers han sido inicializados, para el dibujado se divide en 2 métodos. Por un lado, dado que una malla es un objeto del tipo TEntidad y la TEntidad es una clase abstracta con un método de dibujado (tal y como se ha explicado en el primer apartado) es necesario sobreescribirlo para el dibujado de la malla (no es lo mismo el de la luz o el de la cámara como ya se ha explicado antes).

```
dibujar(
    gl: WebGL2RenderingContext,
    texture: any,
    camara: any,
    colorModelo: any,
    matrizTransformacion: any,
    isWall: boolean // Añadido
): void {
    if (!this.buffers) {
        return; // Salir temprano si los buffers no están inicializados
    }

    const draw = () => {
        if (!this.buffers) return; // Verificar nuevamente antes de dibujar

        this.dibujarMalla(
            gl,
            this.buffers,
            texture,
            camara,
            colorModelo,
            matrizTransformacion,
            isWall // Añadido
        );

        requestAnimationFrame(draw);
    };

    draw();
}
```

Dibujar necesita contar para el funcionado correcto el context gl creado en el motor, la textura del recurso textura, la cámara, el color del modelo (pues para el funcionamiento del color picking, cada malla tiene que tener un color diferente), la matriz de transformación y por último, comprobar si es una pared o no para el frustum culling.

Para el dibujado como tal, necesitamos llamar a otro método de dibujar malla, privado en esta clase. Se añade en un requestAnimationFrame, que crea un bucle de renderizado que continuará dibujando la malla en la pantalla antes de cada repintado del navegador, siempre que this.buffers sea verdadero. El dibujado que irá en el bucle tiene esta funcionalidad:

1. Verifica si los buffers están inicializados. Habilita el culling de caras y la prueba de profundidad en WebGL. Esto es para mejorar el rendimiento y la apariencia de la malla.

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

2. Crea una matriz de proyección y la configura para una perspectiva con un campo de visión de 45 grados, una relación de aspecto basada en el lienzo de WebGL, y un plano cercano y lejano de 0.1 y 100.0 respectivamente.
3. Crea una matriz de vista del modelo y aplica la rotación en los ejes X y Y.
4. Si existe una matriz de transformación, la multiplica con la matriz de vista del modelo.
5. Obtiene la matriz de vista de la cámara y la redondea.

```
const projectionMatrix = mat4.create();
mat4.perspective(
    projectionMatrix,
    (45 * Math.PI) / 180,
    gl.canvas.width / gl.canvas.height,
    0.1,
    100.0
);

const modelViewMatrix = mat4.create();

// Aplicar la rotación primero
mat4.rotate(modelViewMatrix, modelViewMatrix, this.rotationAngleX, [1, 0, 0]);
mat4.rotate(modelViewMatrix, modelViewMatrix, this.rotationAngleY, [0, 1, 0]);

// Aplicar la matriz de transformación si existe
if (matrizTransformacion) {
    mat4.multiply(modelViewMatrix, matrizTransformacion, modelViewMatrix);
}

const roundedViewMatrix = this.roundMatrix(camara.getViewMatrix());
```

```
private roundMatrix(matrix: mat4): mat4 {
    let roundedMatrix = mat4.create();
    for (let i = 0; i < matrix.length; i++) {
        roundedMatrix[i] = parseFloat(matrix[i].toFixed(2));
    }
    return roundedMatrix;
}
```

6. Obtiene el shader program que se esté usando y establece las matrices de modelo, vista y proyección en el program.

```
let shaderProgram = gl.getParameter(gl.CURRENT_PROGRAM);
gl.uniformMatrix4fv(
    gl.getUniformLocation(shaderProgram, "uModelMatrix"),
    false,
    modelViewMatrix
);
gl.uniformMatrix4fv(
    gl.getUniformLocation(shaderProgram, "uViewMatrix"),
    false,
    roundedViewMatrix
);
gl.uniformMatrix4fv(
    gl.getUniformLocation(shaderProgram, "uProjectionMatrix"),
    false,
    projectionMatrix
);
```

7. Establece el uniforme uIsWall en el program que se esté usando, necesario para el frustum culling

```
// Establecer el uniforme uIsWall antes de dibujar
const uIsWallLocation = gl.getUniformLocation(shaderProgram, "uIsWall");
gl.uniform1i(uIsWallLocation, isWall ? 1 : 0);
```

(Consultar apartado de shaders)

8. Si el programa de sombreado actual es el programa principal, inicializa los buffers principales, activa la textura 0, vincula la textura y establece el uniforme del sampler en el programa de sombreado. Si el programa de sombreado es el programa de selección de color, inicializa los buffers de color y establece el uniforme de color en el programa de sombreado.

```
if (this.shaderProgramMain === shaderProgram) {
    this.InitBuffersMain(gl, this.buffers);

    gl.activeTexture(gl.TEXTURE0);
    gl.bindTexture(gl.TEXTURE_2D, texture);
    gl.uniform1i(gl.getUniformLocation(shaderProgram, "uSampler"), 0);
} else if (this.shaderProgramColorSelect === shaderProgram) {
    this.InitBuffersColor(gl, this.buffers);
    let colorLocation = gl.getUniformLocation(shaderProgram, 'uColor');
    gl.uniform4fv(colorLocation, colorModelo);
}
```

9. Finalmente, dibuja los elementos de la malla usando los índices en el buffer de índices

```
gl.drawElements(
    gl.TRIANGLES,
    buffers.indexBuffer.numItems,
    gl.UNSIGNED_SHORT,
    0
);
```

5.3 Borrado de las mallas

Como en nuestra escena el usuario puede o bien borrar todos los muebles de la escena o el mueble seleccionado con el color picking, necesitamos un método para limpiar los buffers de la malla:

```
public limpiarBuffers(gl: WebGL2RenderingContext): void {
    if (this.buffers) {
        if (this.buffers.vertexBuffer) {
            gl.deleteBuffer(this.buffers.vertexBuffer);
        }
        if (this.buffers.textureBuffer) {
            gl.deleteBuffer(this.buffers.textureBuffer);
        }
        if (this.buffers.normalBuffer) {
            gl.deleteBuffer(this.buffers.normalBuffer);
        }
        if (this.buffers.indexBuffer) {
            gl.deleteBuffer(this.buffers.indexBuffer);
        }
        this.buffers = null; // Establecer buffers a null después de limpiarlos
    }
}
```

Su funcionamiento es el siguiente:

1. Comprueba si this.buffers existe. Si no existe, el método se detiene y retorna.
2. Si this.buffers.vertexBuffer existe, lo elimina utilizando gl.deleteBuffer.
3. Hace lo mismo para this.buffers.textureBuffer, this.buffers.normalBuffer y this.buffers.indexBuffer.
4. Finalmente, establece this.buffers a null.

5.3 Frustum culling

Para las paredes hemos utilizado frustum culling, permitiendo que según vaya girando la cámara se descarten las paredes que no están en el campo de visión de la cámara. Esto se logra utilizando la clase Frustum, que define un volumen de visión en forma de pirámide truncada.

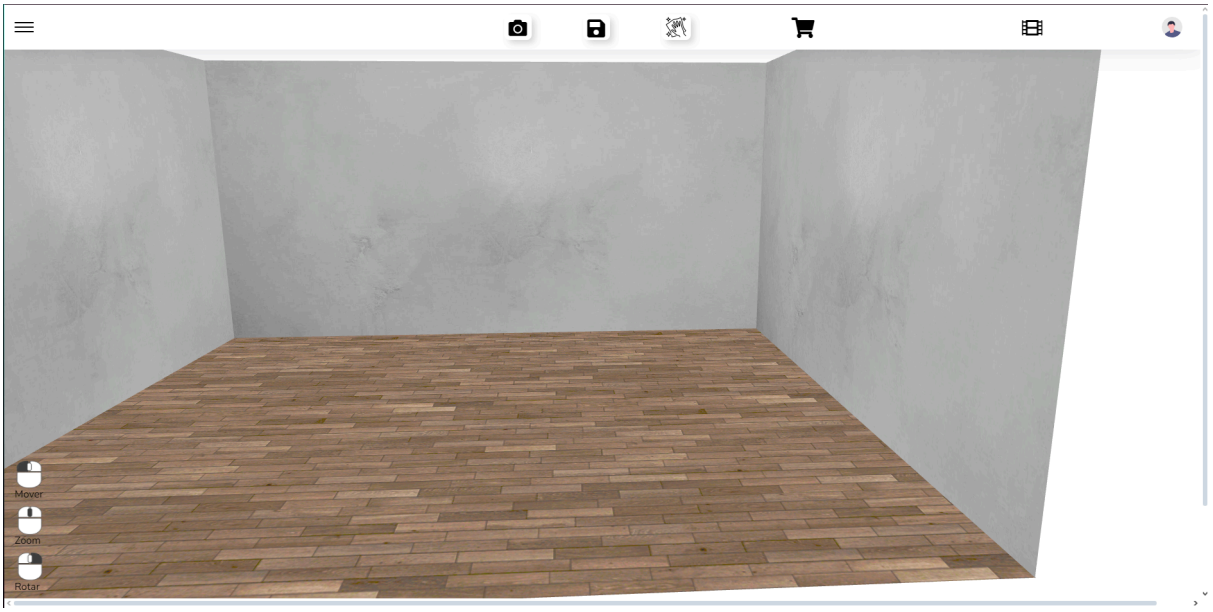
```
import { mat4, vec3, vec4 } from 'gl-matrix';

You, hace 3 segundos | 2 authors (DiegoLIUA and others)
export class Frustum {
    private planes: vec4[] = [];

    constructor(projectionMatrix: mat4, viewMatrix: mat4) {
        const pvMatrix = mat4.multiply(mat4.create(), projectionMatrix, viewMatrix);

        // Extract frustum planes
        this.planes = [
            vec4.fromValues(pvMatrix[3] - pvMatrix[0], pvMatrix[7] - pvMatrix[4], pvMatrix[11] - pvMatrix[8],
            vec4.fromValues(pvMatrix[3] + pvMatrix[0], pvMatrix[7] + pvMatrix[4], pvMatrix[11] + pvMatrix[8],
            vec4.fromValues(pvMatrix[3] - pvMatrix[1], pvMatrix[7] - pvMatrix[5], pvMatrix[11] - pvMatrix[9],
            vec4.fromValues(pvMatrix[3] + pvMatrix[1], pvMatrix[7] + pvMatrix[5], pvMatrix[11] + pvMatrix[9],
            vec4.fromValues(pvMatrix[3] - pvMatrix[2], pvMatrix[7] - pvMatrix[6], pvMatrix[11] - pvMatrix[10],
            vec4.fromValues(pvMatrix[3] + pvMatrix[2], pvMatrix[7] + pvMatrix[6], pvMatrix[11] + pvMatrix[10],
        ];

        // Normalize the planes
        this.planes.forEach(plane => {
            const length = Math.sqrt(plane[0] * plane[0] + plane[1] * plane[1] + plane[2] * plane[2]);
            plane[0] /= length;
            plane[1] /= length;
            plane[2] /= length;
            plane[3] /= length;
        });
    }

    isBoxInFrustum(min: vec3, max: vec3): boolean {
        for (const plane of this.planes) {
            if (
                plane[0] * (plane[0] < 0 ? min[0] : max[0]) +
                plane[1] * (plane[1] < 0 ? min[1] : max[1]) +
                plane[2] * (plane[2] < 0 ? min[2] : max[2]) +
                plane[3] < 0
            ) {
                return false;
            }
        }
        return true;
    }
}
```

En el constructor de Frustum, se calculan los seis planos (izquierdo, derecho, inferior, superior, cercano y lejano) que definen este volumen a partir de la matriz de proyección y la matriz de vista. Estos planos se normalizan para facilitar los cálculos posteriores.

El método isBoxInFrustum se utiliza para determinar si un objeto (en este caso, una pared) está dentro del frustum. Para ello, se comprueba si el objeto está en el lado positivo de todos los planos del frustum. Si el objeto está en el lado negativo de al menos uno de los planos, se considera que está fuera del frustum y se descarta, es decir, no se renderiza.

De esta manera, el frustum culling permite optimizar el rendimiento de la renderización 3D al evitar el procesamiento de objetos que no están en el campo de visión de la cámara.

Durante todo el dibujado y en las capturas que se han visto, desde el motor, modelo (con su TRecursoModelo), mallas, necesitamos el boolean de isWall (explicado aquí y en el apartado de shaders) ya que las paredes se crean desde la interfaz y éstas si cuentan con dicho boolean como true, lo que permite que únicamente a las paredes se les aplique el frustum. El resultado visual es este:



Como se puede observar, desde la cámara hay una pared que impediría la vista. Según el usuario vaya rotando la cámara, se irá escondiendo la pared correspondiente para que la vista esté en todo momento sobre los muebles de la escena.

6. Interfaz

La interfaz se encuentra en nuestro “main”, que es el archivo engine-tag.component.ts (En el HTML solo tenemos el canvas html). Contiene todas las funciones del usuario que necesitan del motor: cargar un mueble y mostrarlo, duplicarlo, borrarlo, limpiar toda la escena, guardar una escena en la BBDD y cargarla (una escena es lo que se muestra en pantalla. por ejemplo, un usuario ha cargado 3 sillas y 2 mesas, y quiere guardarla. pues se guarda la posición y rotación de cada malla, y si quiere cargar una escena guardada, se crean modelos con esa información almacenada) y hacer una captura de pantalla.

6.1 Inicialización del motor

Primero necesitamos todos estos elementos:

```
@ViewChild('canvas') private canvasRef!: ElementRef<HTMLCanvasElement>;

private motorTAG!: TMotorTAG;
private camaraMotor!: any;
private muebles_escena: any[] = [];
private nombreMueble: string = "";
private nombreMuebleDuplicado: string = "";
private clickStartX: number = 0;
private clickStartY: number = 0;
private isDragging: boolean = false;
private lastPosX: number = 0;
private lastPosY: number = 0;
private totalRotation: number = 0;
private totalMovementX: number = 0;
private totalMovementY: number = 0;
private radius: number = 20;
private minRadius: number = 5;
private maxRadius: number = 30;

private recursoShader!: any;
private shaderProgramMain!: WebGLProgram;
private shaderProgramColorSelect!: WebGLProgram;

private selected_object_id = -1;

private gl!: WebGL2RenderingContext;
private selected_mueble_index: number = 0;
```

Nada más iniciar la página, con el método de Angular ngAfterViewInit se llama al método de iniciar motor, que contiene esta información:

```
async ngAfterViewInit(): Promise<void> {
    this.initMotor();
    this.recursoShader = ShaderService.getInstance().getRecursoShader();
    this.shaderProgramMain = this.recursoShader.getShaderProgramMain();
    this.shaderProgramColorSelect = this.recursoShader.getShaderProgramColorSelect();
}

private nombres_muebles: any[] = [];

async initMotor(): Promise<void> {
    const canvas = this.canvasRef.nativeElement;

    // Crear el motor
    this.motorTAG = new TMotorTAG(canvas);  // Asignar al motorTAG de la clase
    await this.motorTAG.init();

    let glContext = canvas.getContext('webgl2', { preserveDrawingBuffer: true });
    if (!glContext) {
        throw new Error('WebGL 2.0 no es soportado por tu navegador');
    }
    this.gl = glContext;
    if (!this.gl) {
        console.error('Unable to initialize WebGL. Your browser may not support it.');
        return;
    }

    this.motorTAG.InitLights(this.gl);

    // Crear la cámara
    this.camaraMotor = this.motorTAG.crearCamara(true, 0.1, 1000.0, 45 * Math.PI / 180, canvas.width / canvas.height);

    // Crear el suelo y agregarlo al array
    const idSuelo = 0; // Puedes asignar un ID específico para el suelo si es necesario
    const colorSuelo = this.createColor(idSuelo); // Generar un color único para el suelo
    const suelo: TModelo = this.motorTAG.crearNodoModelo("suelo", "suelo", colorSuelo);
    this.muebles_escena.push(suelo);

    this.crearParedes();

    this.motorTAG.dibujar();

    this.setupEventListeners(canvas);
}
```

Se usa ngAfterViewInit para que siempre funcione cada vez que cargue la vista en la web. En este método llamamos al canvas del HTML. Iniciamos el motor pasándole dicho canvas, y él se encarga de crear el webglcontext que se usará en todo el motor. Se inicializan las luces y se crea la cámara, creamos el suelo y las paredes (con un método aparte):

```
private crearParedes(): void {
    if (this.muebles_escena.length === 0) {
        console.error("No hay suelo en la escena para duplicar.");
        return;
    }

    const posiciones = [
        vec3.fromValues(0, 4.9, 9.8),
        vec3.fromValues(-9.8, 4.9, 0),
        vec3.fromValues(0, 4.9, -9.8),
        vec3.fromValues(9.8, 4.9, 0)
    ];

    const rotaciones = [
        vec3.fromValues(-Math.PI / 2, Math.PI, 0), //va
        vec3.fromValues(Math.PI / 2, 0, -Math.PI / 2),
        vec3.fromValues(Math.PI / 2, Math.PI , 0), //va
        vec3.fromValues(Math.PI / 2, 0, Math.PI / 2)
    ];

    posiciones.forEach((posicion, index) => {
        const idPared = index + 1;
        const colorPared = this.createColor(idPared); // Genera un color único basado en la longitud actual del array
        const nuevaPared: TModelo = this.motorTAG.crearNodoModelo("pared", "pared", colorPared);

        nuevaPared.setPosition(posicion);
        nuevaPared.setRotation(rotaciones[index]);

        this.muebles_escena.push(nuevaPared);
        //// console.log(this.muebles_escena);
    });
}
```

Y finalmente, recorriendo el árbol de escena dibujamos todo.

6.2 Creación de un modelo y duplicarlo en escena

```
public crearModelo(nombreModelo: string) {
    this.nombreMueble = nombreModelo;
    // Crear el mueble
    const colorMueble = this.createColor(this.muebles_escena.length + 1); // Genera un color único basado en la longitud actual del array
    const nuevoMueble: TModelo = this.motorTAG.crearNodoModelo(`${this.nombreMueble}_${this.muebles_escena.length + 1}`, this.nombreMueble, colorMueble);
    this.nombres_muebles.push(nombreModelo);
    this.muebles_escena.push(nuevoMueble);
    // console.log(this.muebles_escena);
    this.motorTAG.dibujar();
}
private contadorDuplicados: { [nombre: string]: number } = {};

public crearModeloDuplicado() {
    const nombreBase = this.nombreMuebleDuplicado;
    const index = this.contadorDuplicados[nombreBase] || 1;
    const nombreMueble = `${nombreBase}_${index}`;

    // Incrementar el contador de duplicados para este nombre base
    this.contadorDuplicados[nombreBase] = index + 1;

    // Crear el mueble
    const colorMueble = this.createColor(this.muebles_escena.length + index); // Genera un color único basado en la longitud actual del array
    const nuevoMueble: TModelo = this.motorTAG.crearNodoModelo(nombreMueble, nombreBase, colorMueble);
    this.nombres_muebles.push(nombreBase);
    this.muebles_escena.push(nuevoMueble);
    // console.log(this.muebles_escena);
    this.motorTAG.dibujar();
}
```

crearModelo(nombreModelo: string): Este método crea un nuevo modelo con un nombre y color únicos. Primero, asigna el nombre del modelo a this.nombreMueble. Luego, genera un color único para el modelo basado en la longitud actual del array this.muebles_escena. Después, crea un nuevo modelo utilizando el método this.motorTAG.crearNodoModelo y lo añade a los arrays

this.nombres_muebles y this.muebles_escena. Finalmente, llama a this.motorTAG.dibujar() para renderizar la escena con el nuevo modelo.

contadorDuplicados: Esta es una propiedad de la clase que se utiliza para llevar un registro de cuántas veces se ha duplicado un modelo. Es un objeto donde las claves son los nombres de los modelos y los valores son el número de veces que se ha duplicado cada modelo.

crearModeloDuplicado(): Este método crea una copia de un modelo existente. Primero, obtiene el nombre base del modelo a duplicar y el número de veces que se ha duplicado hasta ahora. Luego, genera un nombre único para el modelo duplicado y incrementa el contador de duplicados para el nombre base. Después, genera un color único para el modelo duplicado, crea el modelo duplicado y lo añade a los arrays this.nombres_muebles y this.muebles_escena. Finalmente, llama a this.motorTAG.dibujar() para renderizar la escena con el modelo duplicado.

6.3 Limpiar toda la escena

Para limpiar toda la escena tenemos el siguiente código:

```
public borrarModelos(): void {
    if (this.muebles_escena.length > 5) {
        for (let i = 5; i < this.muebles_escena.length; i++) {
            const modeloAEliminar = this.muebles_escena[i];
            if (modeloAEliminar instanceof TModelo) {
                // Limpiar buffers de las mallas dentro de cada modelo
                const mallas = modeloAEliminar.getMallas();
                for (let malla of mallas) {
                    if (malla instanceof TMalla) {
                        malla.limpiarBuffers(this.gl);
                    }
                }

                // Eliminar el nodo del árbol de escena a través de motorTAG
                this.motorTAG.eliminarNodoPorEntidad(modeloAEliminar);
            }
        }

        const elementosFijos = this.muebles_escena.slice(0, 5);
        this.muebles_escena = elementosFijos;
        this.nombres_muebles = [];

        // Limpiar recursos del gestor de recursos
        this.motorTAG.getGestorRecursos().limpiarRecursosModelos();

        this.motorTAG.dibujar();
    } else {
        // console.log("No hay muebles para borrar.");
    }
}
```

1. Primero, verifica si hay más de 5 modelos en la escena con this.muebles_escena.length > 5. Si hay menos de 6 modelos, no hace nada ya que estos 5 modelos son el suelo y las 4 paredes y no queremos que se borren.
2. Para cada modelo, verifica si es una instancia de TModelo. Si el modelo es una instancia de TModelo, obtiene las mallas del modelo con modeloAEliminar.getMallas() y recorre cada malla con otro bucle for. Para cada malla, verifica si es una instancia de TMalla. Si lo es, llama a malla.limpiarBuffers(this.gl) para limpiar los buffers de la malla.
3. Después de limpiar los buffers de las mallas del modelo, llama a this.motorTAG.eliminarNodoPorEntidad(modeloAEliminar) para eliminar el modelo del árbol de escena.
4. Una vez que ha recorrido todos los modelos y ha eliminado los necesarios, guarda los primeros 5 modelos en this.muebles_escena y vacía el array this.nombres_muebles.
5. Luego, llama a this.motorTAG.getGestorRecursos().limpiarRecursosModelos() para limpiar los recursos de los modelos en el gestor de recursos.
6. Finalmente, llama a this.motorTAG.dibujar() para renderizar la escena sin los modelos eliminados.

6.4 Color picking

Para el color picking se ha seguido el tutorial proporcionado por el profesor. Ya se ha explicado anteriormente lo necesario para su funcionamiento en lo relativo a los shaders y las mallas, pero el núcleo del funcionamiento está en el click del canvas:

```
canvas.addEventListener('click', (event: MouseEvent) => {
    this.gl.clear(this.gl.COLOR_BUFFER_BIT | this.gl.DEPTH_BUFFER_BIT);
    this.recursoShader.useProgramColorSelect(this.gl);

    const rect = canvas.getBoundingClientRect();
    const x = event.clientX - rect.left;
    const y = event.clientY - rect.top;
    let mouse_y = canvas.clientHeight - y;
    let mouse_x = x;

    requestAnimationFrame(() => {
        const pixel = new Uint8Array(4);
        this.gl.readPixels(mouse_x, mouse_y, 1, 1, this.gl.RGBA, this.gl.UNSIGNED_BYTE, pixel);
        this.selected_object_id = this.convertColorToId(pixel[0], pixel[1], pixel[2], pixel[3]);

        this.selected_mueble_index = -1;
        for (let i = 0; i < this.muebles_escena.length; i++) {
            const mueble = this.muebles_escena[i];
            const muebleColor = mueble.getColor();
            if (this.colorsAreSimilar(pixel, muebleColor)) {
                this.selected_mueble_index = i;
                this.nombreMuebleDuplicado = mueble.getName();
                break;
            }
        }
        if (this.selected_mueble_index > 4) {
            this.threeService.controlesMueble("nombreObjetoClickeado");
        } else {
            this.threeService.controlesMueble(null);
        }
        this.recursoShader.useProgramMain(this.gl);
        this.motorTAG.dibujar();
    });
});
```

Muy resumidamente, se limpia el buffer de color y profundiad, se dibuja con el shader de colores la escena, se guarda el color que se haya clicado (cada malla tiene un color diferente) y volvemos al shader principal. Dibujamos la escena con el shader principal. Con esto implementado se tiene un control del mueble clicado. Para crear un color para cada malla, que debe ser aleatorio:

```
public createColor(id: number): Float32Array {
    id = id * 1000000;
    var r, g, b, a;
    var red_bits = this.gl.getParameter(this.gl.RED_BITS);
    var green_bits = this.gl.getParameter(this.gl.GREEN_BITS);
    var blue_bits = this.gl.getParameter(this.gl.BLUE_BITS);
    var alpha_bits = this.gl.getParameter(this.gl.ALPHA_BITS);
    var total_bits = red_bits + green_bits + blue_bits + alpha_bits;
    var red_scale = Math.pow(2, red_bits);
    var green_scale = Math.pow(2, green_bits);
    var blue_scale = Math.pow(2, blue_bits);
    var alpha_scale = Math.pow(2, alpha_bits);
    var red_shift = Math.pow(2, green_bits + blue_bits + alpha_bits);
    var green_shift = Math.pow(2, blue_bits + alpha_bits);
    var blue_shift = Math.pow(2, alpha_bits);
    var color = new Float32Array(4);
    r = Math.floor(id / red_shift);
    id = id - (r * red_shift);
    g = Math.floor(id / green_shift);
    id = id - (g * green_shift);
    b = Math.floor(id / blue_shift);
    id = id - (b * blue_shift);
    a = id;
    color[0] = r / (red_scale - 1);
    color[1] = g / (green_scale - 1);
    color[2] = b / (blue_scale - 1);
    color[3] = a / (alpha_scale - 1);
    return color;
}
```

Este método crea un color según el ID de cada modelo. También se comprueba a la hora de crear un modelo que su color no esté ya para otro modelo ya existente.

6.5 Guardar una escena

Para guardar la escena de los muebles en pantalla se crea un JSON con toda la información de las mallas en ella.

```
public generarJSON(): string {
    const mueblesEscenaSimplificados = this.muebles_escena.slice(5).map(mueble => {
        const radianes = mueble.rotation[1];
        const grados = radianes * (180 / Math.PI); // Convertir radianes a grados
        return {
            posicion: mueble.position,
            colorModelo: mueble.colorModelo,
            fichero: mueble.fichero,
            ID: mueble.recursoTextura?.ID,
            rotacion: grados // Usar grados en lugar de radianes
        };
    });
    return JSON.stringify(mueblesEscenaSimplificados);
}
```

1. Primero, toma todos los modelos en this.muebles_escena excepto los primeros cinco con this.muebles_escena.slice(5). Luego, transforma cada modelo en un objeto más simple con

el método map. Para cada modelo, obtiene la rotación en radianes, la convierte a grados y devuelve un objeto con las propiedades posicion, colorModelo, fichero, ID y rotacion.

2. posicion es la posición del modelo en la escena, colorModelo es el color del modelo, fichero es el nombre del archivo del modelo, ID es el ID del recurso de textura del modelo, si existe, rotacion es la rotación del modelo en grados.
3. Finalmente, convierte el array de objetos simplificados a una cadena JSON con JSON.stringify y la retorna.

6.6 Cargar una escena

Para cargar una escena en base al JSON guardado en el método previo, usamos este método:

```
public cargarEscenaJSON(json_file: string): void {

    this.borrarModelos();

    let mueblesEscena = JSON.parse(json_file);

    for (let [index, mueble] of mueblesEscena.entries()) {
        const colorMueble = Object.values(mueble.colorModelo) as vec3;

        const nuevoMueble: TModelo = this.motorTAG.crearNodoModelo(`${mueble.fichero}_${index + 1}`, mueble.fichero, colorMueble);
        const position = Object.values(mueble.posicion) as vec3;
        const rotation = mueble.rotacion;
        // console.log("Rotacionm: ", rotation);
        nuevoMueble.setPosition(position);
        nuevoMueble.setRotationY(rotation);
        this.muebles_escena.push(nuevoMueble);
        this.nombres_muebles.push(mueble.fichero);

    }
    this.motorTAG.getGestorRecursos().limpiarRecursosModelos();
    this.motorTAG.dibujar();
}
```

1. Primero, llama a this.borrarModelos() para eliminar todos los modelos existentes en la escena.
2. Luego, parsea la cadena JSON con JSON.parse(json_file) para obtener un array de objetos que representan los modelos.
3. Recorre cada modelo en el array con un bucle for. Para cada modelo, obtiene el color, la posición y la rotación del modelo.
4. Crea un nuevo modelo con this.motorTAG.crearNodoModelo, utilizando el nombre del archivo del modelo y el color. El nombre del nuevo modelo se genera añadiendo el índice (incrementado en 1) al nombre del archivo del modelo.
5. Establece la posición y la rotación del nuevo modelo con nuevoMueble.setPosition(position) y nuevoMueble.setRotationY(rotation).

6. Añade el nuevo modelo al array this.muebles_escena y el nombre del archivo del modelo al array this.nombres_muebles.
7. Una vez que ha recorrido todos los modelos, llama a this.motorTAG.getGestorRecursos().limpiarRecursosModelos() para limpiar los recursos de los modelos y a this.motorTAG.dibujar() para renderizar la escena con los nuevos modelos.

6.7 Hacer una captura de pantalla

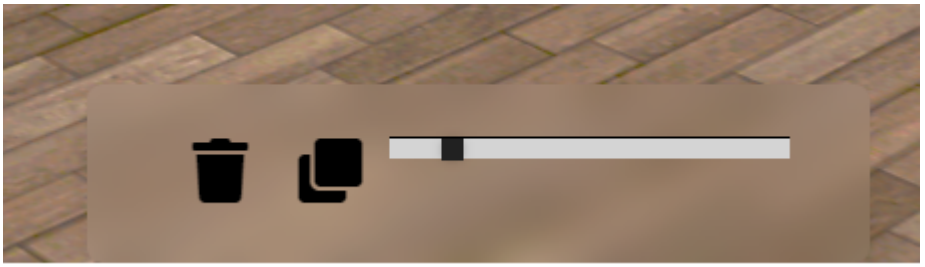
El usuario puede hacer una captura de pantalla y guardarla como png. Hay dos métodos. Uno para mostrar en el desplegable de las escenas del usuario una miniatura (donde es necesaria la captura como base64) y otro para descargar la captura de pantalla como png:

```
public capturarCanvas(): void {
    const canvas = this.canvasRef.nativeElement;
    if (!canvas) {
        return;
    }

    window.requestAnimationFrame(() => {
        canvas.toBlob((blob) => {
            if (blob) {
                // console.log('Blob creado correctamente.');
                const url = URL.createObjectURL(blob);
                const link = document.createElement('a');
                link.href = url;
                link.download = 'captura.png'; // Nombre del archivo de salida
                link.click();
                URL.revokeObjectURL(url); // Limpia la memoria después de descargar
            } else {
                // console.log('No se pudo crear el blob.');
                // console.log(blob);
            }
        });
    });
}

public capturarCanvasEscena(): Promise<string> {
    return new Promise((resolve, reject) => {
        const canvas = this.canvasRef.nativeElement;
        if (!canvas) {
            reject('No se encontró el elemento canvas.');
            return;
        }
        const base64data = canvas.toDataURL('image/jpeg', 0.5); // Reducir la calidad a 0.5
        if (base64data) {
            resolve(base64data);
        } else {
            reject('No se pudo capturar la imagen como Base64.');
        }
    });
}
```

6.7 Movimiento y rotación del mueble seleccionado

```
if (this.isDragging) {
    const deltaX = event.clientX - this.lastPosX;
    const deltaY = event.clientY - this.lastPosY;

    if (this.selected_mueble_index > 4) {
        const mueble = this.muebles_escena[this.selected_mueble_index];
        const sensitivityX = 0.015;
        const sensitivityY = 0.02;

        // Obtener las direcciones de la cámara
        const cameraDirection = vec3.subtract(vec3.create(), this.camaraMotor.getLookAt(), this.camaraMotor.getPosition());
        const upVector = vec3.fromValues(0, 1, 0);
        const rightVector = vec3.cross(vec3.create(), cameraDirection, upVector);
        vec3.normalize(rightVector, rightVector);

        // Vector perpendicular a rightVector en el plano XZ (utilizando la componente Z del cameraDirection)
        const forwardVector = vec3.fromValues(cameraDirection[0], 0, cameraDirection[2]);
        vec3.normalize(forwardVector, forwardVector);

        const movementVector = vec3.create();
        vec3.scaleAndAdd(movementVector, movementVector, rightVector, deltaX * sensitivityX);
        vec3.scaleAndAdd(movementVector, movementVector, forwardVector, -deltaY * sensitivityY);
        movementVector[1] = 0; // Asegurarse de no mover en el eje Y

        let position = mueble.getPosition();
        vec3.add(position, position, movementVector);

        // Limitar el movimiento del mueble dentro de un rango específico
        position[0] = Math.max(-9, Math.min(9, position[0]));
        position[2] = Math.max(-9, Math.min(9, position[2]));

        mueble.setPosition(position);
        this.gl.clear(this.gl.COLOR_BUFFER_BIT | this.gl.DEPTH_BUFFER_BIT);
    } else {
        if (event.buttons === 2) {
            this.rotacion_libre(deltaX, deltaY);
        } else if (event.buttons === 1) {
            this.movimiento_libre(deltaX, deltaY);
        }
        this.lastPosX = event.clientX;
        this.lastPosY = event.clientY;
        this.gl.clear(this.gl.COLOR_BUFFER_BIT | this.gl.DEPTH_BUFFER_BIT);
    }
}
```

Dentro del mousemove del canvas con el ratón, si el usuario ha seleccionado un mueble (gracias al color picking) y éste se arrastra, se moverá, tal y como se muestra en el código. También puede rotarlo con una barra de desplazamiento que aparecerá si se ha seleccionado un mueble:



```
public rotarMuebleSeleccionado(angle: number): void {
    if (this.selected_mueble_index === -1) {
        return;
    }
    const modeloARotar = this.muebles_escena[this.selected_mueble_index];
    if (modeloARotar instanceof TModelo) {
        modeloARotar.setRotationY(angle); // Ajusta la rotación en Y del modelo
        // Luego de rotar el modelo, necesitas redibujar la escena
        this.gl.clear(this.gl.COLOR_BUFFER_BIT | this.gl.DEPTH_BUFFER_BIT);
    }
}
```

6.8 Eventos de cámara

Existen 3 tipos de movimiento con la cámara. Si se usa el click izquierdo, el movimiento es libre, tal y como funciona en Blender el SHIFT+Wheel Click.

```
movimiento_libre(deltaX: number, deltaY: number): void {
    const sensitivity = 0.02;
    const cameraDirection = vec3.subtract(vec3.create(), this.camaraMotor.getLookAt(), this.camaraMotor.getPosition());
    const upVector = vec3.fromValues(0, 1, 0);
    const rightVector = vec3.cross(vec3.create(), cameraDirection, upVector);
    const upVectorAdjusted = vec3.cross(vec3.create(), rightVector, cameraDirection);
    vec3.normalize(rightVector, rightVector);
    vec3.normalize(upVectorAdjusted, upVectorAdjusted);

    const movementVector = vec3.create();
    vec3.scaleAndAdd(movementVector, movementVector, rightVector, -deltaX * sensitivity);
    vec3.scaleAndAdd(movementVector, movementVector, upVectorAdjusted, deltaY * sensitivity);

    const newPosition = vec3.create();
    vec3.add(newPosition, this.camaraMotor.getPosition(), movementVector);
    if (newPosition[1] < 1) {
        newPosition[1] = 1
    }

    this.camaraMotor.setPosition(newPosition);

    const newLookAt = vec3.create();
    vec3.add(newLookAt, this.camaraMotor.getLookAt(), movementVector);
    if (newLookAt[1] < 1) {
        newLookAt[1] = 1
    }
    this.camaraMotor.setLookAt(newLookAt);
}
```

1. Primero, define una constante sensitivity que determina la sensibilidad del movimiento de la cámara.
2. Luego, calcula la dirección de la cámara restando la posición de la cámara del punto al que está mirando (getLookAt()).
3. Define un vector upVector que representa la dirección hacia arriba en la escena 3D.
4. Calcula el vector rightVector que es perpendicular tanto a la dirección de la cámara como al vector hacia arriba. Este vector representa la dirección hacia la derecha en la vista de la cámara.
5. Ajusta el vector hacia arriba para que sea perpendicular tanto a la dirección de la cámara como al vector hacia la derecha.
6. Normaliza los vectores rightVector y upVectorAdjusted para que tengan una longitud de 1.
7. Calcula el vector de movimiento escalando y sumando los vectores rightVector y upVectorAdjusted por -deltaX * sensitivity y deltaY * sensitivity respectivamente.
8. Calcula la nueva posición de la cámara sumando el vector de movimiento a la posición actual de la cámara. Si la componente y de la nueva posición es menor que 1, la establece a 1 para evitar que la cámara se mueva por debajo del plano y = 1.
9. Establece la nueva posición de la cámara con this.camaraMotor.setPosition(newPosition).
10. Calcula el nuevo punto al que la cámara está mirando sumando el vector de movimiento al punto actual al que está mirando. Si la componente y del nuevo punto al que está mirando es menor que 1, la establece a 1 para evitar que la cámara mire por debajo del plano y = 1.
11. Establece el nuevo punto al que la cámara está mirando con this.camaraMotor.setLookAt(newLookAt).

Si se usa el click derecho, el movimiento será orbital:

```
movimiento_orbital(deltaX: number): void {
    const sensitivity = 0.005;
    const angle = deltaX * sensitivity;
    //this.camaraMotor.rotateY(angle);
    this.totalRotation += angle * 57;
    if (this.totalRotation > 360 || this.totalRotation < -360) {
        this.totalRotation = 0;
    }
    this.camaraMotor.updateOrbitPosition(angle, this.radius, this.radius / 2);
}
```

Además, el usuario cuenta con un zoom con unos límites para no hacer zoom muy pronunciado o irse muy lejos de la escena:

```
zoom_camara(delta: number): void {
    this.gl.clear(this.gl.COLOR_BUFFER_BIT | this.gl.DEPTH_BUFFER_BIT);

    this.radius += delta;
    this.radius = Math.max(this.minRadius, Math.min(this.radius, this.maxRadius));

    const cameraDirection = vec3.subtract(vec3.create(), this.camaraMotor.getLookAt(), this.camaraMotor.getPosition());
    vec3.normalize(cameraDirection, cameraDirection);
    const newPosition = vec3.scaleAndAdd(vec3.create(), this.camaraMotor.getLookAt(), cameraDirection, -this.radius);
    this.camaraMotor.setPosition(newPosition);
}
```

Cabe destacar que todos los métodos de cámara cuentan con el limpiado de buffers de color y profundidad. Esto es necesario porque cada movimiento de cámara irá actualizando la escena, y de no usarlo, se “sobreescribirán” todos los objetos hasta que se vuelva a dibujar correctamente la escena con motor.dibujar();