

GRADO EN INGENIERÍA INFORMÁTICA
Sistemas Gráficos



**UNIVERSIDAD
DE GRANADA**

P2 : ROBO RACER : Desarrollo de juego en Three.js.

Juan Miguel Acosta Ortega (*acostaojuanmi@correo.ugr.es*)

David Serrano Domínguez (*davidserrano07@correo.ugr.es*)

31 de mayo de 2024

Índice

1 Primera parte de la práctica : Descripción inicial del juego, revisiones y mejoras.	4
2 Diseño de la aplicación	4
2.1 Diagrama de clases	5
2.1.1 Clase RoboRacer	5
2.1.2 Clase Circuito	6
2.1.3 Clase Item de Investigación	6
2.1.4 Clase Ensamblado	7
2.1.5 Clase Componente Placa	8
2.1.6 Clase Componente Tornillos	8
2.1.7 Clase Plancton	9
2.1.8 Clase Ovni	10
2.1.9 Clase Funciones HUD	11
2.1.10 Relaciones entre clases	11
2.2 Modelos simples y jerárquicos	12
2.2.1 Coche lunar (El protagonista)	12
2.2.2 Items	15
2.2.3 Enemigos	18
2.3 Algoritmos y funcionalidades relevantes	20
2.3.1 Circuito	20
2.3.2 Movimiento del personaje por el tubo	21
2.3.3 Colisiones y sus efectos	24
2.3.4 Animaciones	25
2.3.5 Picking	27
2.3.6 Cámaras	29
2.3.7 Iluminación	31
2.3.8 Materiales	33
2.3.9 Bibliotecas y modelos externos	37

Objetivos

En esta práctica se detallarán los aspectos del juego ROBO RACER (entrega de la práctica 2) relevantes para la comprensión de la contrucción de todos los aspectos relacionados con la asignatura Sistemas Gráficos (Modelado, luces, cámaras, materiales, animaciones ...) y la conexión de todas las clases y componentes.

1. Primera parte de la práctica : Descripción inicial del juego, revisiones y mejoras.

Descripción del anteproyecto:

Nuestro juego, al igual que el de los demás, va a ser un juego de avanzar con un vehículo al estilo Mario Kart. La distinción que proponemos es tanto la ambientación como el objetivo principal del juego. Al igual que en el famoso juego de carreras se han de dar 3 vueltas al circuito, el cuál será un “Torus Knot” nativo de ThreeJS. Sin embargo para añadirle dificultad se proponen diferentes retos :

1. Al terminar las tres vueltas se ha de tener una “barra de investigación” llena. Esta se llenará recogiendo “puntos de investigación” distribuidos por el mapa.
2. Además el protagonista ha de esquivar los diferentes alienígenas que intentan despedazar nuestra máquina. Estos al hacer contacto con ella le arrebatará “piezas”, lo que significa que bajará la “barra de vida”.
3. Para recuperar vida y no perder antes de completar la carrera deberá ir recogiendo componentes que le darán más oportunidad de acabar la partida.

Revisiones y mejoras:

La idea esencial del juego es la misma, básicamente hay dos tipos de enemigos, voladores y terrestres, y tres tipos de items, tuercas y tornillos (vida), puntos de investigación (se ha de llenar la barra de investigación antes de acabar 3 vueltas al circuito para ganar), y los componentes (una pcb que da un pequeño impulso).

A esto se le añade una interfaz gráfica que indique al usuario el daño y el progreso en el juego, animaciones que indicarán la marcha del personaje, la colisión con enemigos y el patrón de movimiento de los enemigos voladores y el giro del personaje al rotar por el tubo, la cámara general y en tercera persona, el picking que nos permite derrotar enemigos, y distintas configuraciones de iluminaciones y materiales que se detallarán más adelante. Además en este documento también se detallan los algoritmos utilizados para la colisión, picking, cambio de cámara, interfaz, animaciones ...

2. Diseño de la aplicación

La aplicación trata de una entidad principal (RoboRacer.js) que es el encargado de importar los modelos y cargarlos, tener la lógica principal del juego y renderizar la escena. Además usa las funciones de la clase funcionesHUD para hacer cambios en la interfaz gráfica. En los siguientes apartados detallaremos la composición del juego y la relación de sus diferentes componentes.

2.1. Diagrama de clases

2.1.1. Clase RoboRacer

RoboRacer	
- VUELTA : int - NUM_ENEMIGOS : int - NUM_PREMIOS : int - VELOCIDAD : float - SUBE_VEL : float - VELOCIDAD_MAX : float - fondo : Mesh - sol : Mesh - toro : Mesh - mouse: Vector2 - raycaster : Raycaster - camera : PerspectiveCamera - camera2 : PerspectiveCamera - iluminacionProta : DirectionalLight - iluminacionProta2 : DirectionalLight - ambientLight : AmbientLight - luzPuntual : PointLight - enemigosAColisionar : Map<Box3, Mesh> - ovnis : Vector<Ovni> - tornillosAColisionar : Map<Box3, Mesh> - placasAColisionar : Map<Box3, Mesh> - investigacionesAColisionar : Map<Box3, Mesh> - recienColisionadoEnemigo : Mesh - teclas : Map<key, boolean> - renderer : WebGL	+ colocarEnemigos() : void + colocarPremios() : void + picking (event : Event) : void + createCamera() : void + createCameraThirdPerson(): void + cambiaCamara() : void + createGUI() : GUI + createLights() : void + colisionaEnemigo() : boolean + colisionaTornillos() : boolean + colisionaPlacas() : boolean + colisionaInvestigaciones() : boolean + moverProta(): void + comprouebaVuelta() : void +onWindowResize() : void +restaurarJuego(): void +update() : void

Figura 1: Clase principal del juego : RoboRacer.

2.1.2. Clase Circuito

Circuito
- tubo : tubeGeometry
+getPathFromTorusKnot(torus: torusKnotGeometry) : catMullRomCurve3

Figura 2: Clase circuito

2.1.3. Clase Item de Investigación

Investigacion
- t : float - alfa : float - ensamblado : Object3D - padreTraslacion : Object3D - padreRotacion : Object3D - padrisimo : Object3D - tubo : tubeGeometry - path : Vector3D - radio : float - segmentos : int - esfera : Object3D - text : Object3D
+update(t,alfa) : void

Figura 3: Clase Investigación

2.1.4. Clase Ensamblado

Ensamblado
<pre>- MAX_GIRO :float - giro_personaje : float - colision_animacion : boolean - andando_animacion : boolean - giroHombro : float - giroCodo : float - giroMano : float - t : float - alfa : float - ensamblado : Object3D - padreTraslacion : Object3D - padreRotacion : Object3D - padrisimo : Object3D - tubo : tubeGeometry - path : Vector3D - radio : float - segmentos : int - brazo : Object3D - brazol : Object3D - ruedalt : Object3D - ruedalb : Object3D - ruedalm : Object3D - ruedarm : Object3D - ruedart : Object3D - ruedarb : Object3D - chasis : Object3D - cabeza : Object3D - cajaProta : Box3 - objetoAlejado : Object3D - spotProta : SpotLight</pre> <pre>+ colision() : void + animacionAndar(): void + giro_derecha(): void + giro_izquierda : void + enderezar : void +update(t,alfa) : void</pre>

Figura 4: Clase ensamblado

2.1.5. Clase Componente Placa

C_Placa
- t : float - alfa : float - ensamblado : Object3D - padreTraslacion : Object3D - padreRotacion : Object3D - padrisimo : Object3D - tubo : tubeGeometry - path : Vector3D - radio : float - segmentos : int - esfera : Object3D - placa : Object3D - padreCondensadores : Object3D - padreCondensadores2 : Object3D - padreResistencias : Object3D - padreResistencias2 : Object3D - microchip : Object3D
+update(t,alfa) : void

Figura 5: Clase C_{Placa}

2.1.6. Clase Componente Tornillos

C_Tornillos
- t : float - alfa : float - ensamblado : Object3D - padreTraslacion : Object3D - padreRotacion : Object3D - padrisimo : Object3D - tubo : tubeGeometry - path : Vector3D - radio : float - segmentos : int - esfera : Object3D - tornillo : Object3D - tuerca : Object3D
+update(t,alfa) : void

Figura 6: Clase C_{Tornillos}

2.1.7. Clase Plancton

Plancton
<ul style="list-style-type: none"> - VIDAS: int - t: float - alfa: float - gemTubo: TubeGeometry - path: Vector3D - radio: float - segmentos: int - eyeBrow: Mesh - eyePlancton: Mesh - eyePupilePlancton: Mesh - bodyPlancton: Mesh - mouthPlancton: Mesh - antena1: Mesh - antena2: Mesh - leg1: Mesh - leg2: Mesh - arm1: Mesh - arm2: Mesh - ensamblado: Object3D - padrePupilda: Object3D - padreOjoCompleto: Object3D - padre: Object3D - padrePierna1: Object3D - padrePierna2: Object3D - padreAntena1: Object3D - padreAntena2: Object3D - padreBrazo1: Object3D - padreBrazo2: Object3D - padreTraslation: Object3D - padreRotation: Object3D - padrisimo: Object3D
<ul style="list-style-type: none"> + restarVida(): void + restaurarVida(): void - createEyeBrow(): void - createEyePlancton(): void - createEyePupilePlancton(): void - createBodyPlancton(): void - createMouthPlancton(): void - createAntennasPlancton(): void - createLegsPlancton(): void - createArmsPlancton(): void + update(): void

Figura 7: Clase enemigo terrestres: Plancton.

2.1.8. Clase Ovni

Ovni
- VIDAS: int - MAX_ -ALTURA: int - MIN_ ALTURA: int - SUBIENDO: boolean - t: float - alfa: float - gemTubo: TubeGeometry - path: Vector3D - radio: float - segmentos: int - headFinal: Mesh - ovni: Mesh - cannon1: Mesh - cannon2: Mesh - light: Vector<Mesh> - ensamblado: Object3D - padreCannon1: Object3D - padreCannon2: Object3D - plancton: Plancton - padreTraslation: Object3D - padreRotation: Object3D - padrisimo: Object3D
+ restarVida(): void + restaurarVida(): void
- createOvni(): void - createCannons(): void - createHeadOvni(): void - createLights(): Vector<Mesh> + update(): void

Figura 8: Clase enemigo volador: Ovni.

2.1.9. Clase Funciones HUD

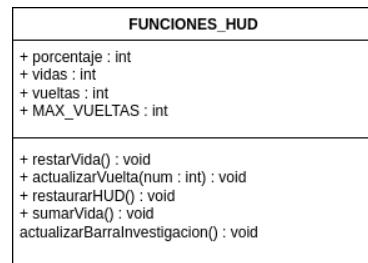


Figura 9: Funciones HUD

2.1.10. Relaciones entre clases

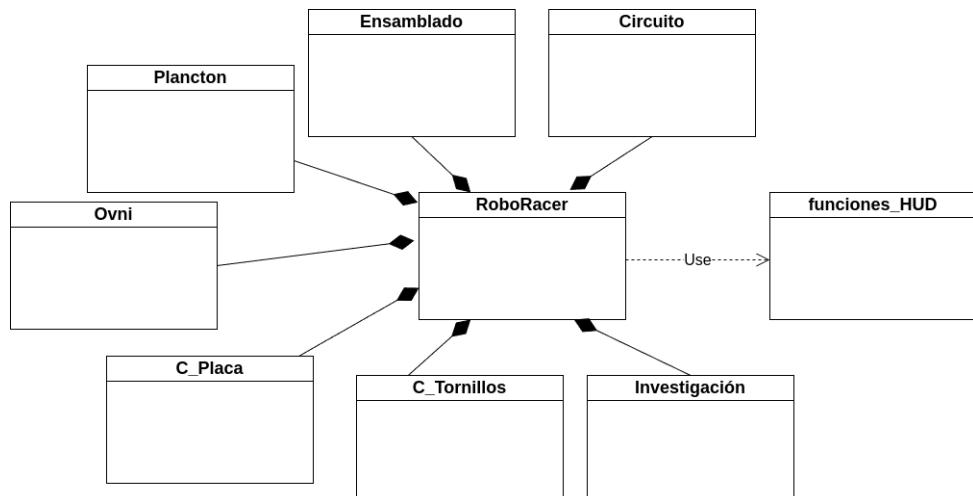


Figura 10: Relaciones entre clases

2.2. Modelos simples y jerárquicos

En este apartado detallaremos los grafos de escena de la fabricación de los modelos principales:

2.2.1. Coche lunar (El protagonista)

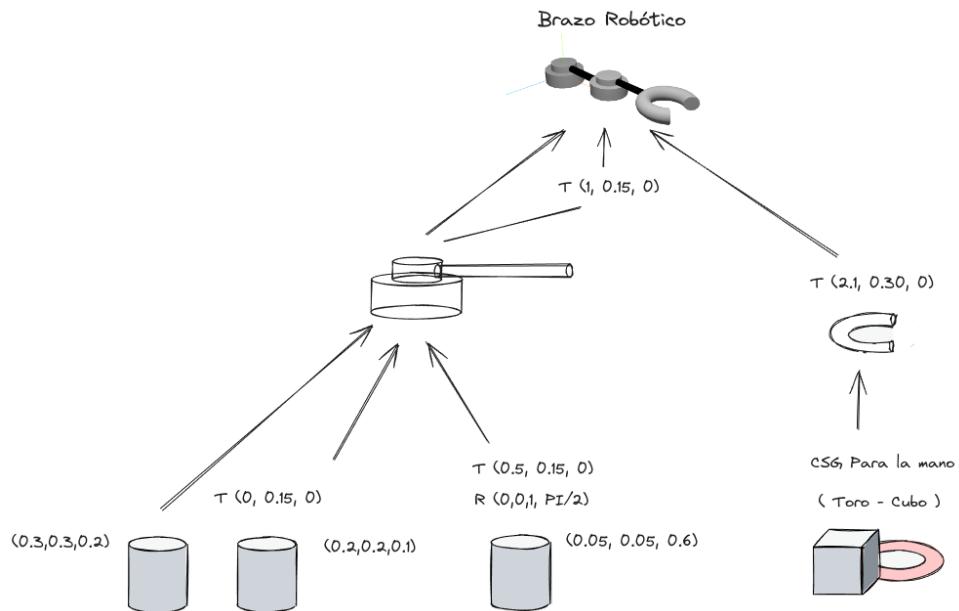


Figura 11: Brazo del coche lunar.

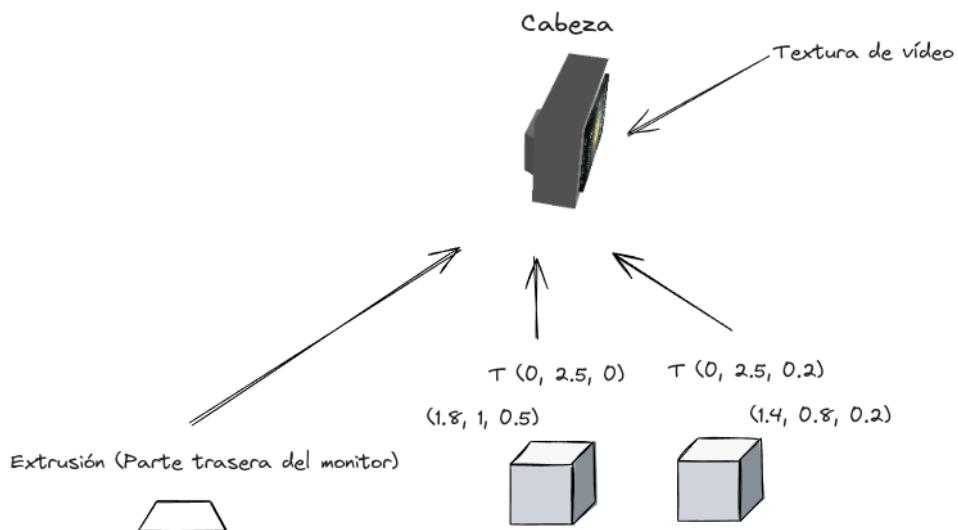


Figura 12: Cabeza del coche lunar.

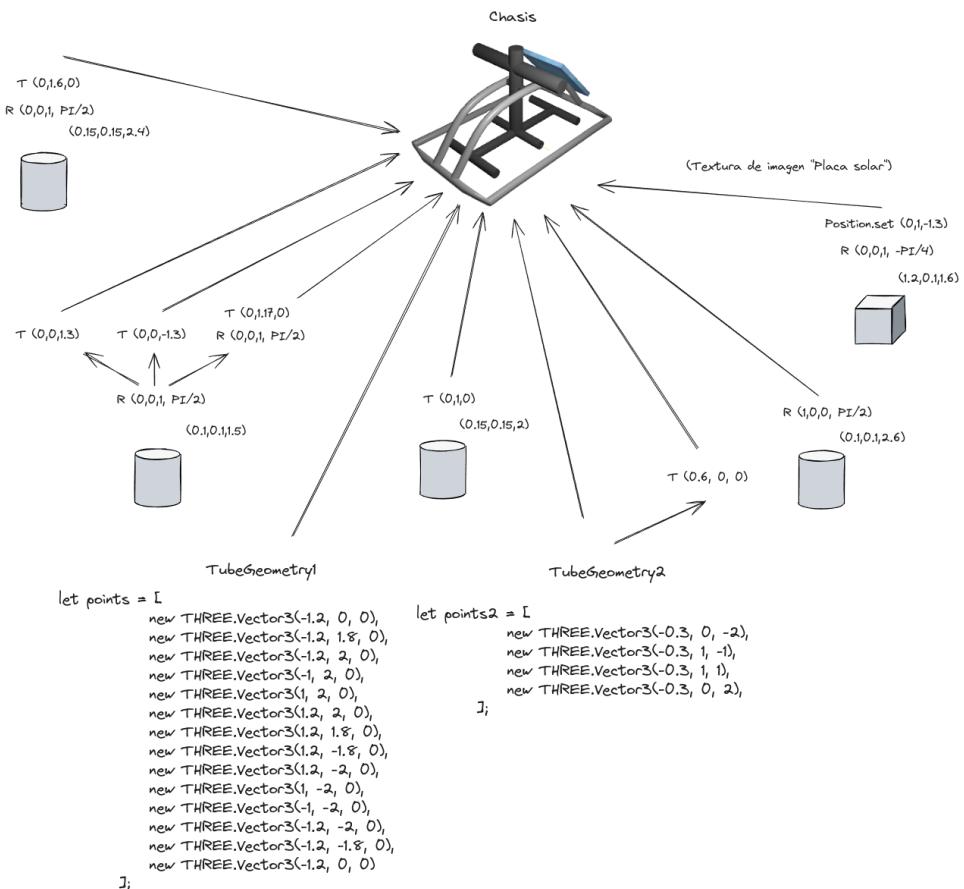


Figura 13: Chasis del coche.



Figura 14: Ruedas usadas en el coche lunar.

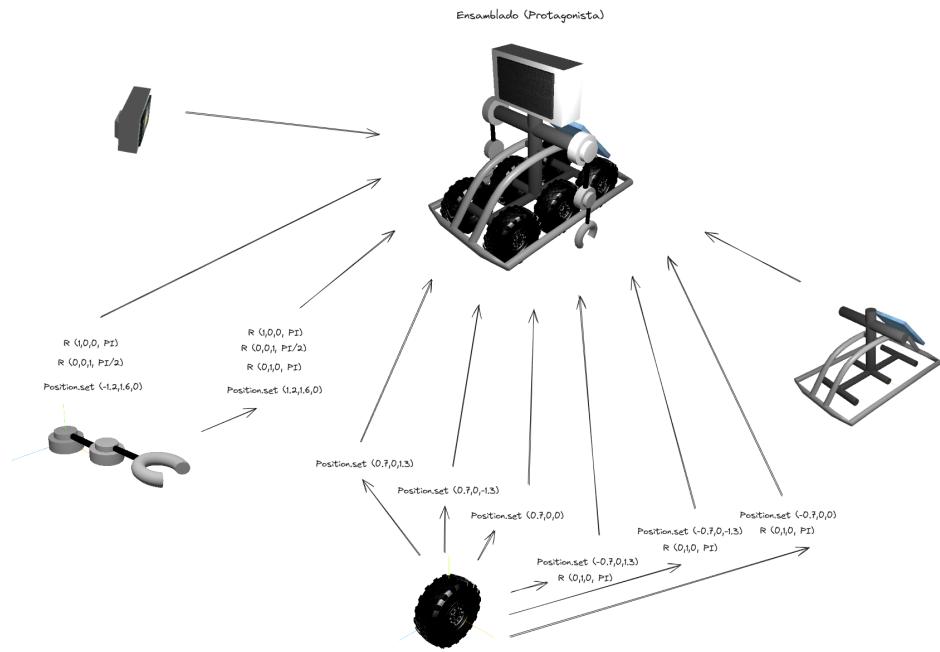


Figura 15: Coche lunar.

2.2.2. Items

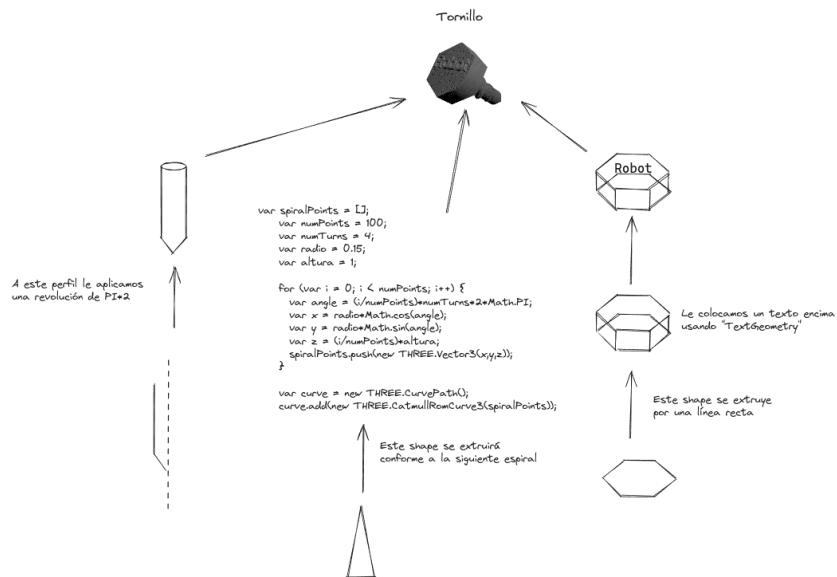


Figura 16: Grafo de tornillo.

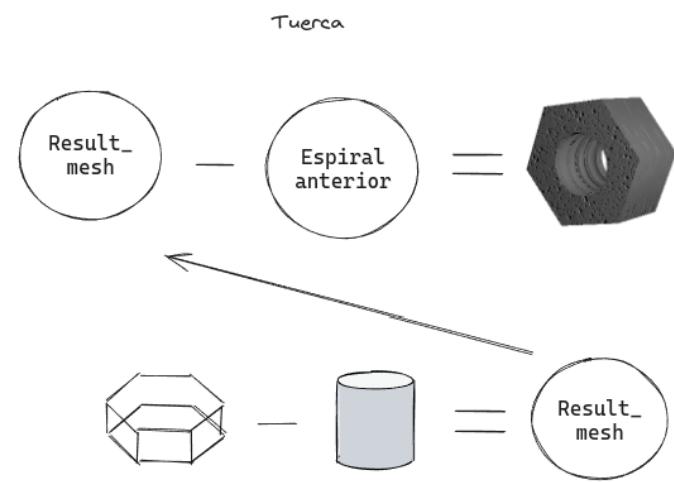


Figura 17: Grafo de tuerca.

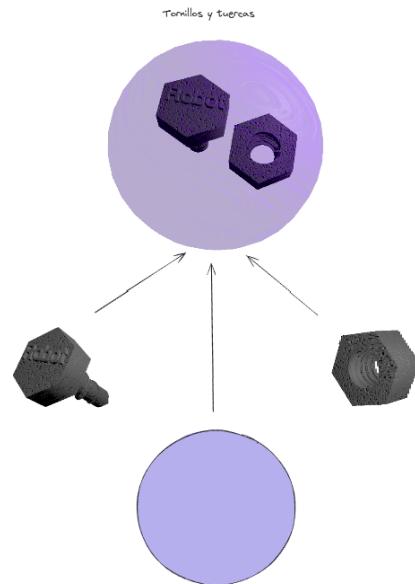


Figura 18: Grafo del ítem que aporta vida.

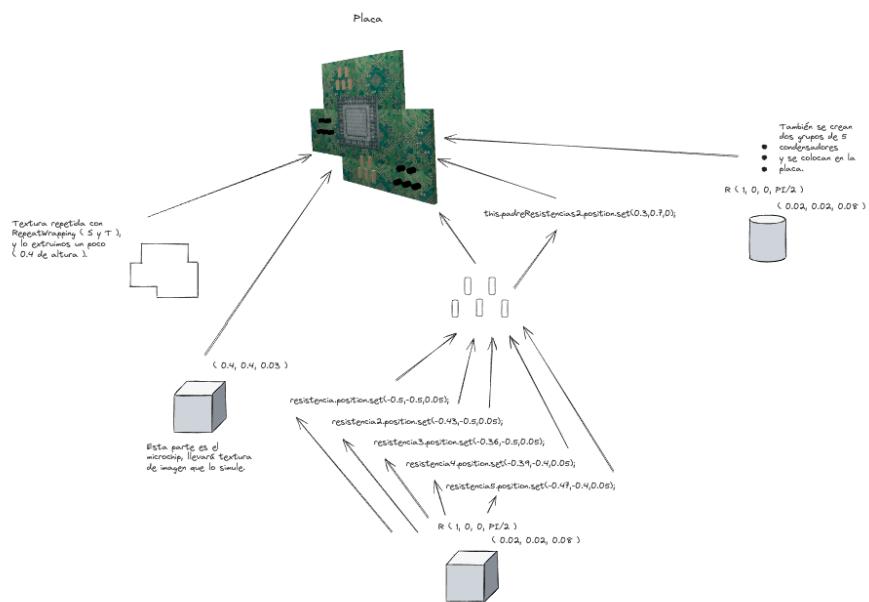


Figura 19: Grafo de la placa.

Item placa

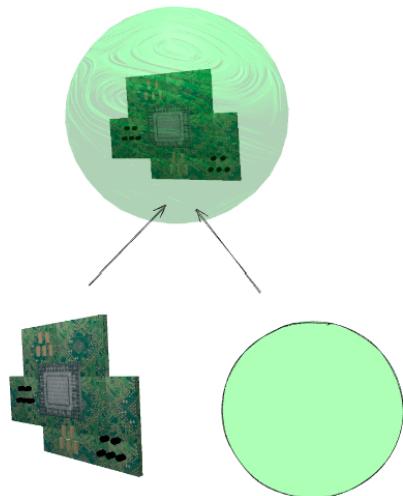


Figura 20: Grafo que aporta impulso.

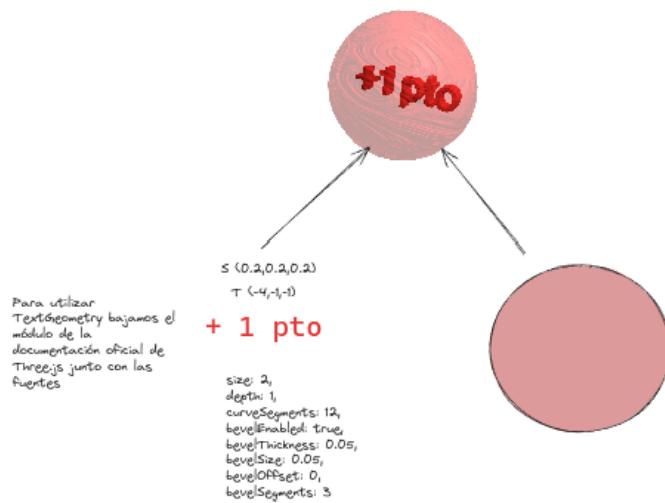


Figura 21: Grafoitem de investigación.

2.2.3. Enemigos

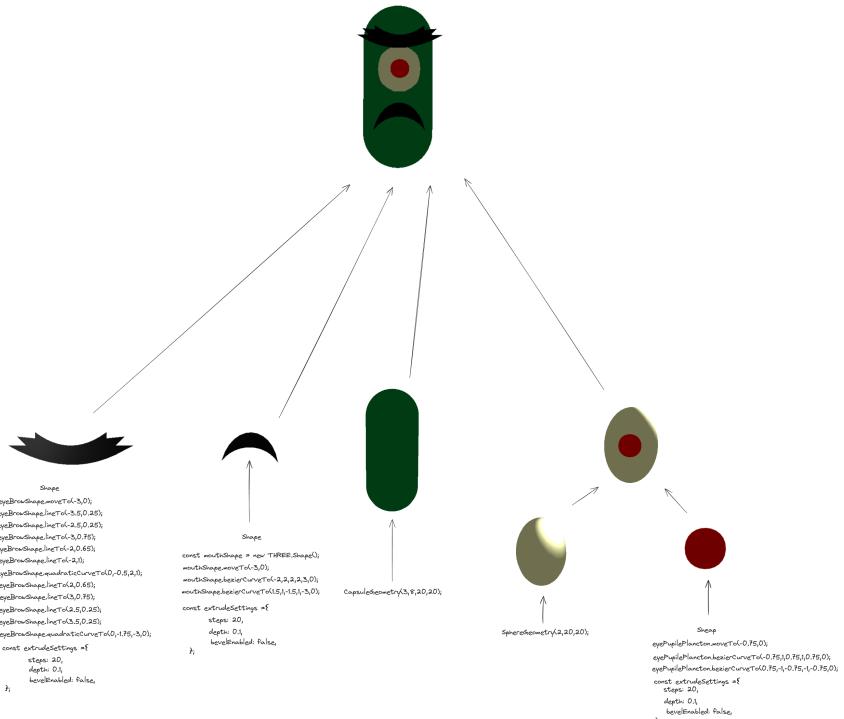


Figura 22: GRafo Cuerpo Plancton.

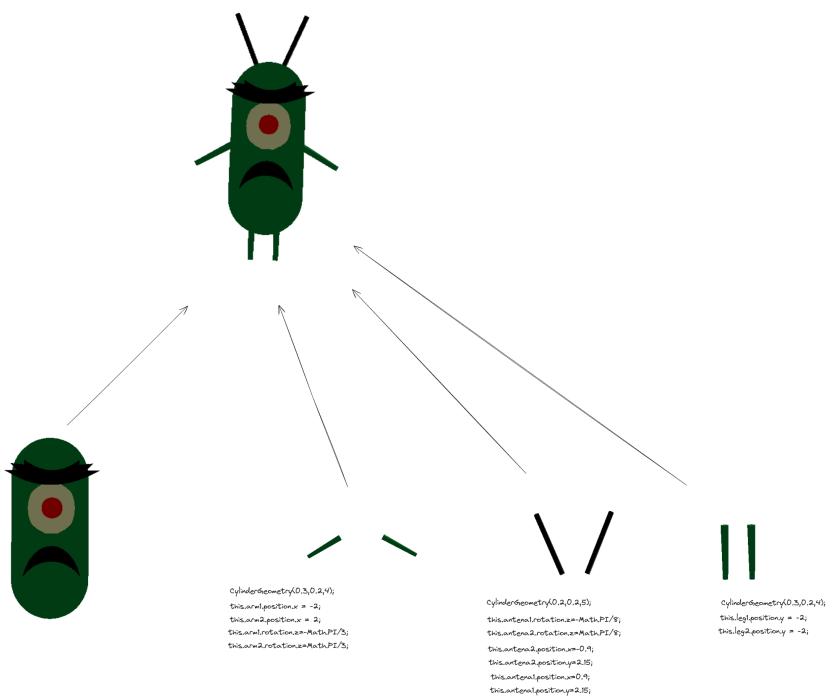


Figura 23: Grafo Plancton.

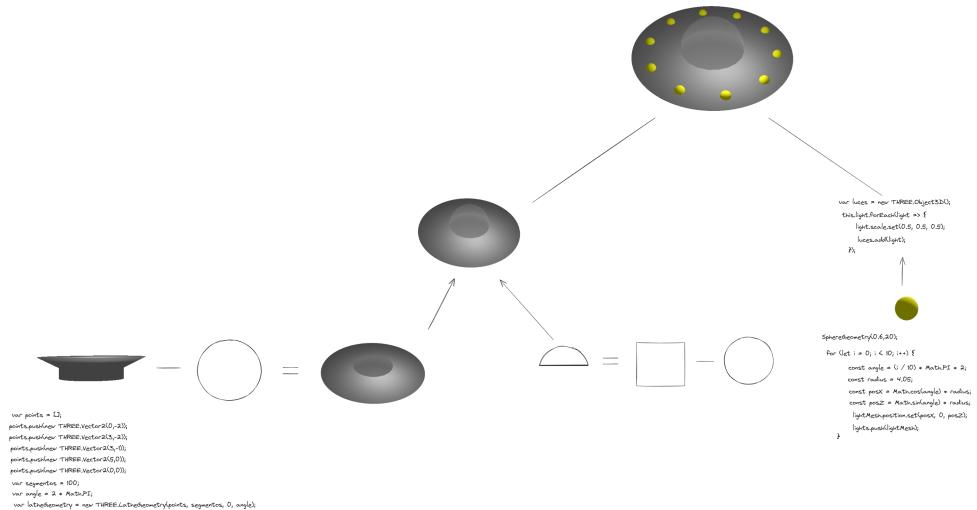


Figura 24: Grafo Ovni Base.

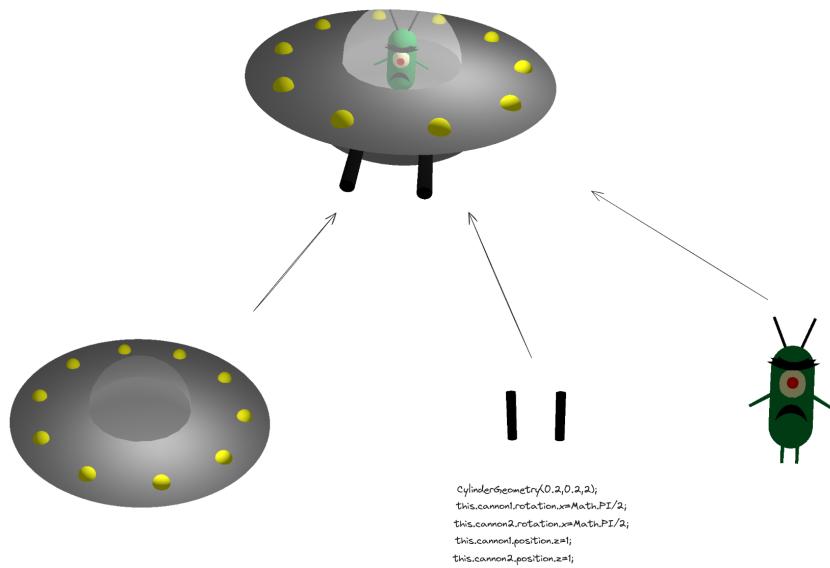


Figura 25: Grafo Ovni.

2.3. Algoritmos y funcionalidades relevantes

2.3.1. Circuito

La forma del circuito está basada en la geometría nativa de Three.js "TorusKnot" con los parámetros **100 (radio)**, **10 (tubo)**, **80(tubularSegments)**, **20(radialSegments)**, **1(p)**, **5(q)**.

A este le añadimos una textura para el canal del color y un relieve con un bumpmap para simular una superficie tecnológica (esto se detalla más adelante).

A partir de esta geometría se extrae el "path" mediante el siguiente método:

```
1  getPathFromTorusKnot (torusKnot) {
2
3      const p = torusKnot.parameters.p;
4      const q = torusKnot.parameters.q;
5      const radius = torusKnot.parameters.radius;
6      const resolution = torusKnot.parameters.tubularSegments;
7      var u, cu, su, quOverP, cs;
8      var x, y, z;
9      // En points se almacenan los puntos que extraemos del torusKnot
10     const points = [];
11     for (let i = 0; i < resolution; ++i) {
12         u = i / resolution * p * Math.PI * 2;
13         cu = Math.cos(u);
14         su = Math.sin(u);
15         quOverP = q / p * u;
16         cs = Math.cos(quOverP);
17
18         x = radius * (2 + cs) * 0.5 * cu;
19         y = radius * (2 + cs) * su * 0.5;
20         z = radius * Math.sin(quOverP) * 0.5;
21
22         points.push(new THREE.Vector3(x, y, z));
23     }
24     // Una vez tenemos el array de puntos 3D construimos y devolvemos el CatmullRomCurve3
25     return new THREE.CatmullRomCurve3(points, true);
26 }
```

Así a partir de este podemos construir un tubeGeometry. Esto lo hacemos con la intención de conocer todos los parámetros de este para poder colocar correctamente nuestros items y personajes alrededor del circuito.

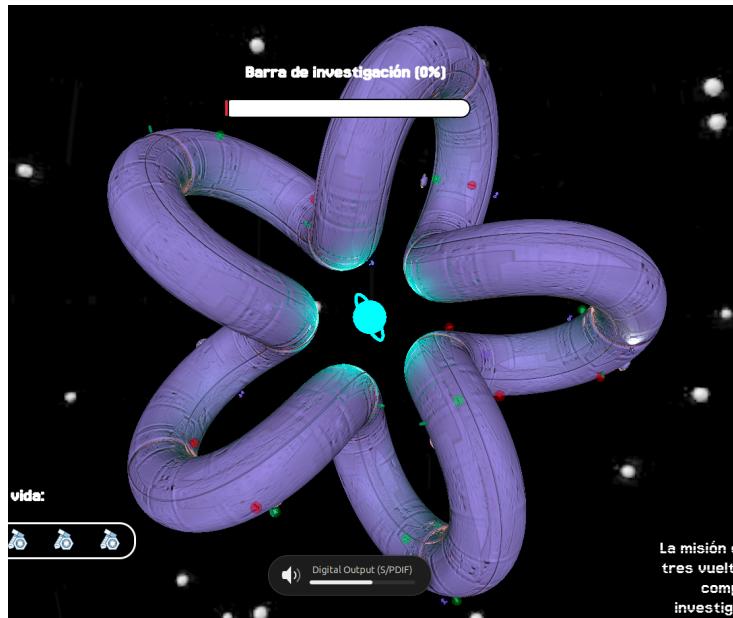


Figura 26: Plano general, Cámara alejada.

2.3.2. Movimiento del personaje por el tubo

Para colocar y actualizar la posición de cualquier objeto alrededor del circuito se necesita, como se indica anteriormente, la geometría del tubeGeometry y ciertos parámetros clave. Para documentar esto pondremos el ejemplo de la colocación del personaje principal, los demás componentes de colocan de manera similar.

El personaje principal tendrá a la hora de su creación un **parámetro t** (de 0 a 1 el avance en el mapa), y **su alfa** (de 0 a Π^*2 su rotación por este).

Los parámetros necesarios del circuito son :

1. **this.tubo = geomTubo;**
2. **this.path = geomTubo.parameters.path;**
3. **this.radio = geomTubo.parameters.radius;**
4. **this.segmentos = geomTubo.parameters.tubularSegments;**

Además se ha de crear un patrón de jerarquía de objetos que se repetirá en todos los componentes a colocar por el circuito. Este se crea con la intención de asegurar que las transformaciones se hagan en el orden esperado:

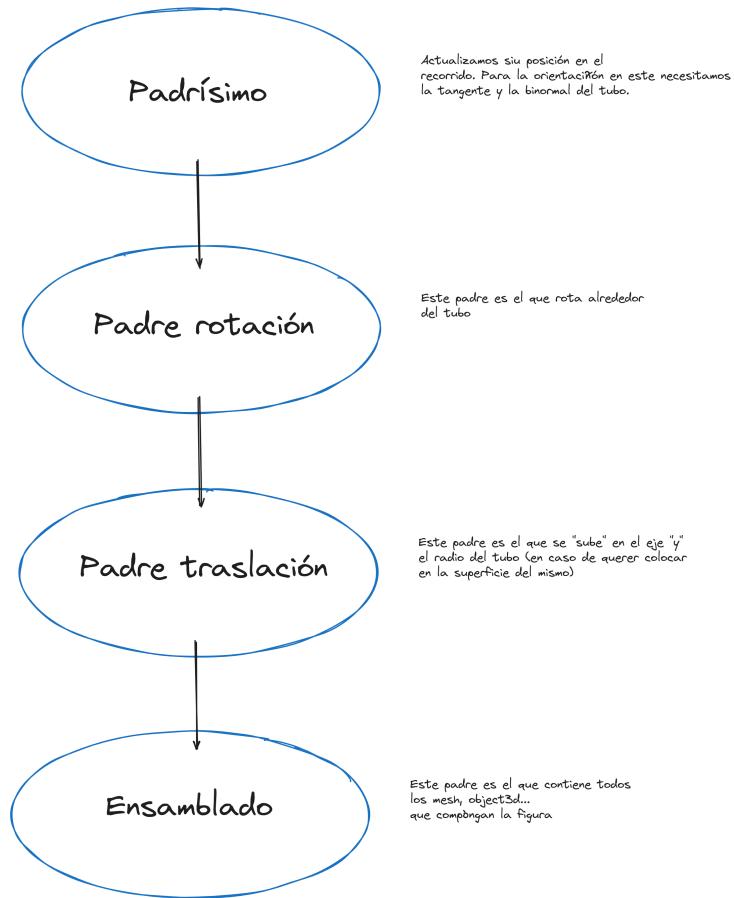


Figura 27: Jerarquía de padres.

Es en el método `actualiza` (update) cuando se reposiciona al personaje según los nuevos valores de sus componentes **t** y **alfa**.

A la hora de testear esta funcionalidad cambiábamos los valores mediante la GUI sencilla que proporciona Three.js, pero a la hora de realizar un juego funcional y atractivo este método no es muy cómodo.

Para actualizar sus componentes **t** y **alfa** se utilizan las teclas **w, a, s, y d**. A estas teclas se les establece un "eventListener" que procura que al ser pulsadas o no cambian su estado (true or false) en un map (tecla ->pulsada) **teclas** que utilizaremos para mover al personaje.

EL método **moverProta()** es el que realmente posee la funcionalidad que hace esto posible.

Para que el movimiento no sea lineal tenemos una variable **VELOCIDAD**, **VELOCIDAD MAX** y **SUBE VEL**. **SUBE VEL** es una componente que se le agrega a velocidad conforme vamos dejando pulsada la tecla W, y que deja de agregarse al llegar a una velocidad máxima. De este modo simulamos una aceleración.

Del mismo modo al soltar la W no se frena automáticamente, sino que la velocidad va decrementándose. Además la velocidad máxima aumenta un 10 por ciento con cada vuelta.

Para más detalle dejamos el método reflejado a continuación:

```

1     moverProta() {
2         var atras;
3         if (this.teclas.get('w')) {
4             if (!this.teclas.get('a') && !this.teclas.get('d')) {
5                 this.prota.enderezar();
6             }
7
8             atras = false;
9             this.VELOCIDAD = this.VELOCIDAD + this.SUBE_VEL;
10            if (this.VELOCIDAD >= this.VELOCIDAD_MAX) {
11                this.VELOCIDAD = this.VELOCIDAD_MAX;
12            }
13            this.prota.update((this.prota.t + this.VELOCIDAD) % 1, this.prota.alfa, this.
14 VELOCIDAD);
15            this.compruebaVuelta();
16
17        } else {
18            if (!this.teclas.get('s') && !atras) {
19                //aqui se frena
20                this.VELOCIDAD = this.VELOCIDAD - 0.000005;
21                if (this.VELOCIDAD < 0) {
22                    this.VELOCIDAD = 0;
23                }
24                this.prota.update((this.prota.t + this.VELOCIDAD) % 1, this.prota.alfa, this.
25 VELOCIDAD);
26            }
27            if (this.teclas.get('a')) {
28                this.prota.update(this.prota.t, (this.prota.alfa - 0.01) % (Math.PI * 2), this.
29 VELOCIDAD);
30                this.prota.giro_izquierda();
31            }
32            if (this.teclas.get('s')) {
33                atras = true;
34                this.VELOCIDAD = 0;
35                if (this.prota.t - 0.0002 < 0) {
36                    this.prota.t = 1;
37                }
38                this.prota.update((this.prota.t - 0.0002) % 1, this.prota.alfa, this.VELOCIDAD);
39            }
40            if (this.teclas.get('d')) {
41                this.prota.update(this.prota.t, (this.prota.alfa + 0.01) % (Math.PI * 2), this.
42 VELOCIDAD);
43                this.prota.giro_derecha();
44        }
}

```

2.3.3. Colisiones y sus efectos

Pasamos a documentar nuestra propuesta a la detección de colisión de los enemigos y los premios. Para ello en primer lugar hemos colocado estos componentes a lo largo del mapa.

En un principio se comenzó añadiendo componentes aleatorias, pero en la versión final se han colocado de manera fija para evitar inconveniencias y poca variedad a lo largo de la carrera.

Una vez hecho esto se obtienen arrays que contienen tanto los mesh de los enemigos, placas ... como sus cajas englobantes, y al actualizar la posición del protagonista se comprueba colisión comprobando si nuestra caja englobante interseca con alguna otra de las colocadas en el circuito (ejemplo de colisión con el enemigo):

```
1  colisionaEnemigo() {
2      var colision = false;
3      for (var i = 0; i < this.enemigosAColisionar.length; i++) {
4          let [caja, mesh] = this.enemigosAColisionar[i];
5          if (this.cajaProta.intersectsBox(caja)) {
6              if (this.recienColisionadoEnemigo != caja) {
7                  this.recienColisionadoEnemigo = caja;
8                  colision = true;
9                  this.VELOCIDAD -= this.VELOCIDAD * 0.2;
10                 if (this.VELOCIDAD < 0) {
11                     this.VELOCIDAD = 0;
12                 }
13                 //Llamamos a animación del prota
14                 this.prota.colision();
15                 console.log(mesh);
16             }
17             return colision;
18         }
19     }
20     return colision;
21 }
```

De esta manera además de devolver un booleano en caso de querer hacer otras acciones, hacemos una acción. En concreto al colisionar con enemigos nos **bajaremos velocidad, desencadenaremos una animación de colisión y nos restaremos una vida**.

En el caso de la placa nos daremos un **pequeño impulso**, los tornillos nos **otorgarán vida** en caso de faltarnos y los puntos de investigación **llenarán la barra de investigación**.

2.3.4. Animaciones

A lo largo del juego se pueden visualizar varias animaciones:

1. **La animación del objeto volador (ovnis).** Esta es una animación sencilla en la que se trasladan estos objetos de arriba a abajo en un rango específico para simular una levitación.
2. **Colisión con los enemigos.** Esta animación se activa cuando ocurre una colisión con un enemigo. Hace que el padre "ensamblado" del protagonista haga dos vueltas completas en torno a su eje y. Esta al activarse mienstras se avanza hacia delante simula "vueltas de campana" como se hace en el videojuego "Mario kart". Además en esta secuencia también se hace un efecto de parpadeo de la luz "spotLight" del personaje, y así cumplimos el requisito de existencia de luz cambiante en el juego.

```
1   if (this.colisionAnimacion){  
2       this.andandoAnimacion = false;  
3       this.ensamblado.rotation.y += 0.1;  
4       this.girado += 0.1;  
5       if (this.girado >= 4*Math.PI){  
6           this.colisionAnimacion = false;  
7           this.andandoAnimacion = true;  
8           this.girado = 0;  
9           this.ensamblado.rotation.y = 0;  
10      }  
11  
12      //Hacer que parpadee la luz del prota 3 veces  
13  
14      for (let i = 0; i < 3; i++) {  
15          setTimeout(() => {  
16              this.spotProta.intensity = 0;  
17              setTimeout(() => {  
18                  this.spotProta.intensity = 5000;  
19                  }, 300);  
20              }, i * 1000);  
21      }  
22  }  
23
```

3. **Animación de andar del personaje.** Esta animación se va intercalando con la de colisión, y básicamente simula un movimiento de brazos que va más rápido conforme aumenta la velocidad del personaje por el circuito.

```
1   animacionAndar(velocidad){  
2       this.giroHombro += velocidad *30;  
3       this.brazol.hombro.rotation.y = Math.sin(this.giroHombro );  
4       this.brazor.hombro.rotation.y = Math.sin(this.giroHombro );  
5  
6       this.giroCodo += velocidad *30;  
7       this.brazol.codo.rotation.y = Math.sin(this.giroCodo);  
8       this.brazor.codo.rotation.y = Math.sin(this.giroCodo );  
9  
10      this.giroMano += velocidad *30;  
11      this.brazol.resultMesh1.rotation.x = Math.sin(this.giroMano);  
12      this.brazor.resultMesh1.rotation.x = Math.sin(this.giroMano);  
13  
14      this.ruedalb.rotation.x += velocidad * 30;  
15      this.ruedalt.rotation.x += velocidad * 30;  
16      this.ruedarb.rotation.x += velocidad * 30;  
17      this.ruedart.rotation.x += velocidad * 30;  
18      this.ruedalm.rotation.x += velocidad * 30;  
19      this.ruedarm.rotation.x += velocidad * 30;  
20  
21  }  
22
```

4. **Animación de giro del personaje.** Esta animación hace que el ensamblado rote (con un límite prudente) en el eje y para que el giro quede menos artificial.

Estas animaciones se activan según la combinación de teclas que se estén pulsando (solo w ->enderezar, a ->giro izquierda, d ->giro derecha):

```
1      giro_derecha() {
2          if (this.ensamblado.rotation.y >= -this.MAX_GIRO) {
3              this.giro_personaje += 0.01;
4              this.ensamblado.rotation.y -= 0.01 ;
5          }
6      }
7
8
9      giro_izquierda() {
10         if (this.ensamblado.rotation.y <= this.MAX_GIRO) {
11             this.giro_personaje -= 0.01;
12             this.ensamblado.rotation.y += 0.01 ;
13         }
14     }
15
16     enderezar() {
17
18         if (this.ensamblado.rotation.y > 0){
19             this.ensamblado.rotation.y -= 0.01;
20             this.giro_personaje -= 0.01;
21         }
22         else if (this.ensamblado.rotation.y < 0){
23             this.ensamblado.rotation.y += 0.01;
24             this.giro_personaje += 0.01;
25         }
26     }
27 }
```

2.3.5. Picking

Para entender el funcionamiento y cómo hemos solucionado la propuesta del picking en nuestro juego, para así poder atacar tanto a enemigos voladores en nuestro caso los ovnis, como a enemigos terrestres que serán los plancton se detalla lo siguiente.

Para poder hacer uso de esta funcionalidad hemos declarado dos variables que serán necesarias para su uso, en este caso son:

```
1 //VARIABLES PICKING
2 this.mouse = new THREE.Vector2();
3 this.raycaster = new THREE.Raycaster();
```

Una vez conocidas estas lo siguiente para el correcto funcionamiento del picking es agregar un "eventListener" para cuando el usuario haga click.

```
1 document.addEventListener('click', (event) => {
2     if (event.button === 0) {
3         this.picking(event);
4     }
5});
```

Ahora reflejamos que hace la función picking

```
1 picking(event) {
2     try {
3         event.preventDefault();
4
5         this.mouse.x = (event.clientX / window.innerWidth) * 2 - 1;
6         this.mouse.y = 1 - 2 * (event.clientY / window.innerHeight);
7
8         this.raycaster.setFromCamera(this.mouse, this.camera2);
9
10        var arrayMesh = this.enemigosAColisionar.map((value) => value[1]);
11
12        var pickedObjects = this.raycaster.intersectObjects(arrayMesh, true);
13
14        if (pickedObjects.length > 0) {
15            var object = pickedObjects[0].object;
16            // Buscar en la jerarquía de padres para encontrar el objeto Plancton o Ovni
17            while (object && !(object instanceof Plancton) && !(object instanceof Ovni)) {
18                object = object.parent;
19            }
20            if (object) {
21                console.log(object);
22                object.restarVida();
23                if (object.VIDAS == 0) {
24                    object.update((object.t + 0.5) % 1, object.alfa);
25                    object.restaurarVida();
26                }
27                // Buscar su caja correspondiente en enemigosAColisionar
28                var caja = this.enemigosAColisionar.find((value) => value[1] === object)[0];
29                caja.setFromObject(object);
30            }
31        }
32
33    } catch (e) {
34        console.log("Se ha disparado al aire" + e);
35    }
36}
```

Lo primero que faremos en la función es meterlo todo dentro de un **try catch** para cuando el usuario dispare a un objeto que no sea "esperado" avisar por consola que ha disparado al aire.

En caso de que haya disparado a alguno de nuestros enemigos, los cuales se encuentran en el array de **pickedObjects** mostrado anteriormente la función hará lo siguiente: Si se ha pickeado algún objeto entonces la función de **intersectObjects** de raycaster nos devolverá un array con los objetos pickeados, lo que haremos a continuación será sacar el último objeto y obtener mediante un while en el que accedemos a el padre del objeto actual, el objeto plancton o ovni en función del que se haya seleccionado.

Una vez hecho esto le restamos vida al objeto, ya que para eliminar un plancton hay que clickarlo dos veces y a un ovni cuatro .

En caso de que después de restarle la vida le queden cero, lo recolocamos usando el update del objeto en el tubo de nuevo y además debemos también recolocar la caja de colisión y esto lo hacemos usando de nuevo la función **setFromObject** y pasándole el objeto. Además le restauramos la vida.

2.3.6. Cámaras

En este punto vamos a explicar las dos cámaras que tenemos en nuestro juego, así como cambiar entre estas usando el botón espacio del teclado.

Uno de los requisitos iniciales de la práctica era que nuestro juego tuviera dos cámaras una con un plano general del circuito y otra en tercera persona que estuviera detrás del personaje principal para poder ver el circuito y los posibles enemigos que nos pudiéramos encontrar. Ambas son en perspectiva , no ortográficas.

Comenzaremos explicando la cámara de plano general.

```
1 createCamera() {
2     this.camera = new THREE.PerspectiveCamera(45, window.innerWidth / window.innerHeight,
3                                                 0.01, 2000);
4
5     this.camera.position.set(0, 0, 450);
6     var look = new THREE.Vector3(0, 0, 0);
7     this.camera.lookAt(look);
8     this.add(this.camera);
9
10    this.cameraControl = new TrackballControls(this.camera, this.renderer.domElement);
11
12    this.cameraControl.rotateSpeed = 5;
13    this.cameraControl.zoomSpeed = -2;
14    this.cameraControl.panSpeed = 0.5;
15
16    this.cameraControl.target = look;
17 }
```

Para esta cámara creamos una **PerspectiveCamera** y le pasamos el fov, aspect, near y far.Una vez hecho esto posicionamos la cámara en el (0,0,450) para tener una vista completa del circuito. Luego creamos un vector con posición en el origen de coordenada y le pasamos como lookat a la camara este vector llamada look, haciendo así que nuestra cámara apunte hacia el origen de coordenadas.

Una vez creada la creamos los controllos de la cámara creando una instancia de **TrackballControls**. Los parámetros para dicho control es la velocidad de giro que la establecemos en cinco, la velocidad de zoom en dos y la velocidad de panorámica(velocidad horizontal y vertical de la cámara) que la ponemos en 0.5, por último establecemos como objetivo de la cámara el origen de coordenadas haciendo que la cámara siempre mire a dicho origen de coordenadas aun moviendo dicha cámara.

Vamos a pasar ahora a explicar la creación de la cámara en tercera persona.

```
1 createCameraThirdPerson() {
2     this.cameraController = new THREE.Object3D();
3     this.cameraController.position.set(0, 30, -22);
4     this.cameraController.rotateY(Math.PI);
5     this.cameraController.rotateX(-Math.PI / 12);
6
7     this.camera2 = new THREE.PerspectiveCamera(60, window.innerWidth / window.innerHeight,
8                                                 0.01, 1000);
9     this.cameraController.add(this.camera2);
10 }
```

Lo primero que hacemos es crear el control de la cámara como un **Object3D** una vez hecho esto lo posicionamos y rotamos para que esta encima de la cabeza de nuestro personaje.

Cuando ya tenemos posicionado dicho control creamos al igual que con la cámara general un **PerspectiveCamera** al que le pasamos distintos parámetros de fov y far. Y una vez creada le añadimos al **cameraController** nuestra cámara de 3 persona.

Vamos por último en este apartado a explicar como cambiar entre ambas cámaras usando el espacio. Primero hemos añadido un event listener para cuando pulsamos el espacio.

```
1 document.addEventListener('keydown', (event) => {
2     //Barra espaciadora ->
3     if (event.code === 'Space') {
4         this.guiControls.cambia = !this.guiControls.cambia;
5         this.cambiaCamara();
6     }
7});
```

Cuando ocurre este evento llamamos a la función **cambiaCamara** que se encarga de cambiar entre dichas cámaras. Esta función hace lo siguiente:

```
1 cambiaCamara() {
2     if (this.guiControls.cambia) {
3         this.remove(this.camera);
4         this.padreCamara.add(this.cameraController);
5     } else {
6         this.padreCamara.remove(this.cameraController);
7         this.add(this.camera);
8
9         this.cameraControl = new TrackballControls(this.camera, this.renderer.domElement);
10
11        this.cameraControl.rotateSpeed = 5;
12        this.cameraControl.zoomSpeed = -2;
13        this.cameraControl.panSpeed = 0.5;
14
15        this.cameraControl.target = look;
16    }
17}
```

Lo que hace es comprobar el estado del booleano de la **gui** el cual si es true entonces eliminamos la cámara general y añadimos a **padreCamara** el **cameraController** que contenía la cámara de tercera persona. En caso contrario eliminamos el **cameraController** y agregamos a la escena la cámara de plano general.

Destacar que **padreCamara** es una variable que contiene el **padreTranslacion** del objeto ensamblado, ya que aquí es donde debemos agregar la cámara para que esta se vea correctamente.

2.3.7. Iluminación

Vamos a explicar ahora que luces tenemos en nuestro juego, como las hemos creado y donde se encuentran estas.

En nuestro juego tenemos principalmente cinco luces, concretamente tenemos una en el centro posicionado en el planeta azul, luego direccionales posicionadas cada una a un lado del circuito otra en el robot la cual varia la intensidad que era uno de los requisitos y por último una luz ambiental.

Para crear todas estas luces tenemos una función que se encarga de esto.

```
1 createLights() {
2     //ILUMINACION
3     //luz direccional 1
4     this.iluminacionProta = new THREE.DirectionalLight(0xffaaaa, 6.5);
5     this.iluminacionProta.position.set(0, 0, -50);
6
7     //luz ambiental
8     this.ambientLight = new THREE.AmbientLight(0x404040, 1);
9
10    //luz direccional 2
11    this.iluminacionProta2 = new THREE.DirectionalLight(0xaaafff, 6.5);
12    this.iluminacionProta2.position.set(0, 0, 50);
13    this.iluminacionProta2.rotateY(Math.PI);
14
15    //Luz puntual del planeta
16    this.luzPuntual = new THREE.PointLight(0x00ffff);
17    this.luzPuntual.intensity = 1;
18    this.luzPuntual.power = 100000;
19    this.luzPuntual.position.set(0, 0, 0);
20    this.luzPuntual.visible = true;
21    this.luzPuntual.castShadow = true;
22
23    this.add(this.iluminacionProta);
24    this.add(this.iluminacionProta2);
25    this.add(this.ambientLight);
26    this.add(this.luzPuntual);
27 }
```

Primero creamos la primera luz direccional **"directionalLight"** con un color rojizo, luego creamos la luz ambiental **"ambientalLight"**, posteriormente creamos la otra luz direccional cuyo color es un azul claro. Una vez creadas estas luces la siguiente que creamos es la del planeta cuyo color es un azul claro parecido al cian **"pointLight"**. Una vez creadas estas cuatro luces las añadimos a nuestra escena.

Ahora vamos a ver como hemos creado la luz de nuestro personaje **"spotLight"** y como hacemos que cuando este se choque con un enemigo parpadee durante unos segundos.

Primero para crearlo hemos hecho lo siguiente:

```
1 this.objetoAlejado = new THREE.Object3D();
2 this.objetoAlejado.position.set(0, 0, 10);
3 this.ensamblado.add(this.objetoAlejado);
4 //Luz que sigue al protagonista
5 this.spotProta = new THREE.SpotLight(0xffffffff);
6 this.spotProta.power = 5000;
7 this.spotProta.angle = Math.PI / 4;
8 this.spotProta.penumbra = 1;
9 this.spotProta.decay = 2;
10 this.spotProta.position.set(0, 2.5, 0);
11 this.spotProta.target = this.objetoAlejado;
12 this.spotProta.rotateOnAxis(new THREE.Vector3(0,1,0), Math.PI/2 );
13
14 this.ensamblado.add(this.spotProta);
```

En la clase ensamblado creamos un **object3D** que posicionamos y luego añadimos a nuestro ensamblado. Este objeto será luego el target de la luz de nuestro personaje que será hacia adonde apunte dicha luz.

Una vez hecho esto creamos una luz **SpotLight** de color blanco, le asignamos de ppterencia cinco mil y un ángulo de PI/4, penumbra de uno, decay de dos y la posicionamos en (0,2.5,0). Una vez hecho todo esto le agregamos como target el objeto alejado creado anteriormente y rotamos la luz en el eje Y PI/2.

Cuando tenemos todo esto le agregamos a ensamblado la luz.

Una vez tenemos la luz vamos a ver como funciona el parpadeo.

```
1 if (this.colisionAnimacion){  
2     this.andandoAnimacion = false;  
3     this.ensamblado.rotation.y += 0.1;  
4     this.girado += 0.1;  
5     if (this.girado >= 4*Math.PI){  
6         this.colisionAnimacion = false;  
7         this.andandoAnimacion = true;  
8         this.girado = 0;  
9         this.ensamblado.rotation.y = 0;  
10    }  
11  
12 //Hacer que parpadee la luz del prota 3 veces  
13 for (let i = 0; i < 3; i++) {  
14     setTimeout(() => {  
15         this.spotProta.intensity = 0;  
16         setTimeout(() => {  
17             this.spotProta.intensity = 5000;  
18             }, 300);  
19         }, i * 1000);  
20    }  
21 }
```

Este if se va a encontrar en el **update** de nuestra clase ensamblado y se va a ejecutar cuando nos colisionemos con un enemigo y se ponga a true la variable **colisionAnimación**, que tras ejecutar las distintas líneas de código encargadas de la animación llegara a un **for** el cual se va a ejecutar tres veces, la cantidad de veces que parpadeará la luz. Dentro de este for tendremos un **timeout** el cual hará que la intensidad de la luz se ponga a cero y posteriormente hacer otro **timeout** que pondrá la luz del personaje al máximo(5000) a los trescientos milisegundos y a su vez el parpadeo durará el valor de i actual por mil milisegundos.

2.3.8. Materiales

En esta sección explicaremos los materiales que nos encontramos en nuestro juego.

Primero mostramos las texturas usadas para nuestro circuito que tiene una textura normal y otra de relieve. A continuación se muestra la linea de código donde se incluyen ambas texturas y posteriormente una imagen de cada una.

```
1 var texture = new THREE.TextureLoader().load("./textures/circuitoT.jpg");
2 texture.wrapS=THREE.RepeatWrapping;
3 texture.wrapT=THREE.RepeatWrapping;
4 texture.repeat.set(10,3);
5
6 //Aniadimos bumpMap al circuito
7 const bumpTexture = new THREE.TextureLoader().load('./textures/circuito_bump_map.jpg');
8 bumpTexture.wrapS=THREE.RepeatWrapping;
9 bumpTexture.wrapT=THREE.RepeatWrapping;
10 bumpTexture.repeat.set(10,3);
```

Se cargan ambas texturas y hacemos que se repitan en ambos sentidos siendo repetido a lo largo de circuito diez veces y alrededor del mismo tres.

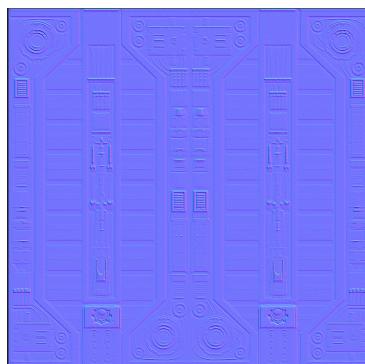


Figura 28: Textura de relieve.

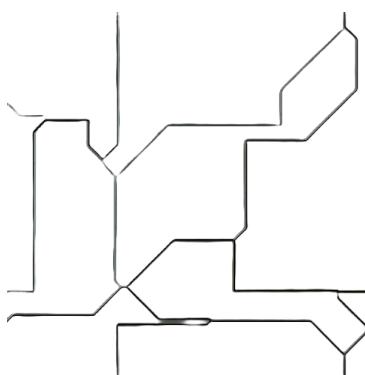


Figura 29: Textura.

A continuación pasamos a ver las texturas que hemos añadido a nuestros objetos. Vamos a empezar por el objeto de la placa base.

```
1 let texture = new THREE.TextureLoader().load('../imgs/placa.jpg');
2 texture.wrapS = THREE.RepeatWrapping;
3 texture.wrapT = THREE.RepeatWrapping;
4 //Configurar repeticion de la textura con THREE.RepeatWrapping
5 let materialPlacaSolar = new THREE.MeshPhongMaterial({ map: texture });
```

```

6
7 //Material esfera
8 const bumpTexture = new THREE.TextureLoader().load('../textures/normalmapitem2.jpg')
9 materialEsfera.bumpMap = bumpTexture
10 materialEsfera.bumpScale = 10

```

Primero se le añade una textura de una placa base a nuestro objeto placa y una vez hecho esto le añadimos una textura de relieve a la esfera que envuelve a la placa.

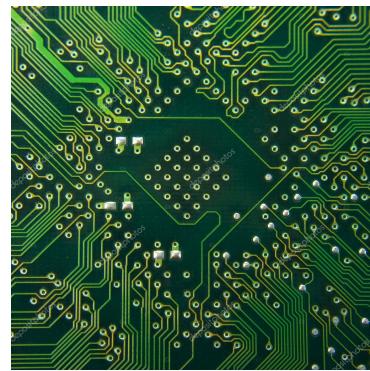


Figura 30: Textura Placa.

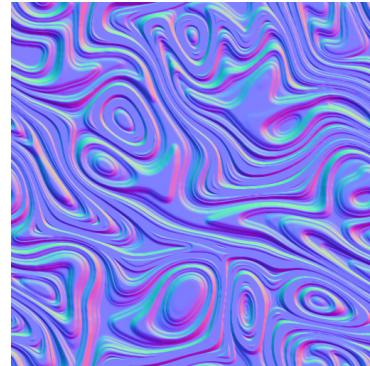


Figura 31: Textura Relieve esfera.

La siguiente figura será el tornillo y la tuerca ambas tendrán la misma textura de relieve y la esfera que los envuelve tendrá la misma textura que el objeto anterior variando solo el color de dicha esfera. Para dar además esa sensación de dureza hemos usado las propiedades **roughness** y **metalness** en **MeshStandardMaterial**.

```

1 const bumpTexture = new THREE.TextureLoader().load('../textures/bumpmapmetal.jpg')
2 material.bumpMap = bumpTexture
3 material.bumpScale = 10

```

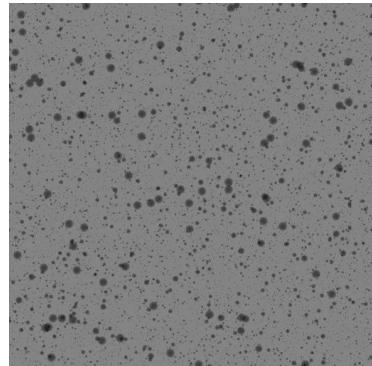


Figura 32: Textura Tornillo y Tuerca.

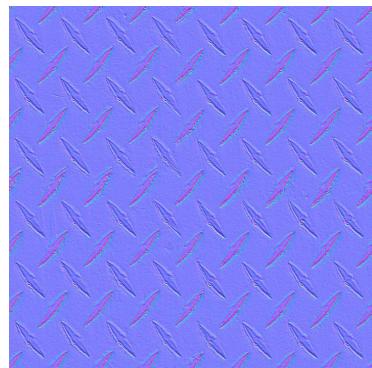


Figura 33: Textura relieve Tornillo y Tuerca.

El último objeto que nos faltaría sería el objeto de investigación pero la textura que tiene es para la esfera que lo envuelve y es igual que para los objetos anteriores sólo que en este caso de color rojo.

La última textura que nos encontraremos en nuestro juego será en la cabeza del personaje principal la cual reproduce un vídeo en bucle y para añadirlo hemos usado **VideoTexture**

```
1 //VIDEO COMO TEXTURA on loop
2 let video = document.createElement('video');
3 video.addEventListener('canplaythrough', () => { video.play() });
4 video.src = 'imgs/HUD.mp4';
5 video.load();
6 video.play();
7 video.loop = true;
8 //silenciar
9 video.muted = true;
10 let videoTexture = new THREE.VideoTexture(video);
11 let panel = new THREE.Mesh(panelGeo, new THREE.MeshPhongMaterial({ map: videoTexture }));
```

Además el personaje principal tiene una textura de placa solar en la que lleva a la espalda:

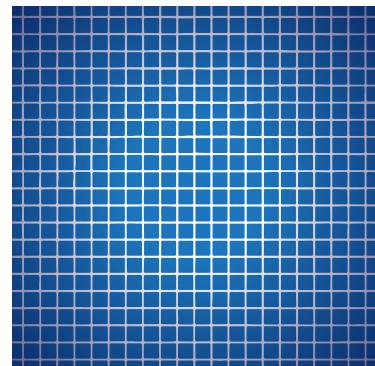


Figura 34: Textura Placa Solar.

Además y para finalizar el apartado recordamos el uso de **emissive** y **emissiveIntensity** para dar esa sensación de "bombilla" al planeta central del juego.

2.3.9. Bibliotecas y modelos externos

Vamos al último punto que es las bibliotecas y materiales externos, en nuestro caso no hemos usado muchas salvo para las ruedas del personaje principal y para los puntos de investigación hemos añadido una librería de texto para crear dicho objetos.

Vamos a empezar explicando las ruedas como las hemos importado y posteriormente poner la fuente de las hemos sacado.

```
1 const onProgress = function (xhr) {
2   if (xhr.lengthComputable) {
3     const percentComplete = xhr.loaded / xhr.total * 100;
4     console.log(percentComplete.toFixed(2) + '% downloaded');
5   }
6 };
//-----
7 new MTLLoader()
8   .setPath('../models/ruedas/')
9   .load('ruedas.mtl', function (materials) {
10     materials.preload();
11     new OBJLoader()
12       .setMaterials(materials)
13       .setPath('../models/ruedas/')
14       .load('ruedas.obj', function (object) {
15         self.add(object);
16       }, onProgress);
17   });
18});
```

En el constructor de las ruedas nos encontramos este trozo de código mediante el que cargamos la rueda.
La fuente de esta rueda es **la siguiente**.

Las siguientes librerías han sido usadas tanto **TextGeometry** como **FontLoader** que nos permiten crear una geometría a partir de un texto. En este caso para poner dentro del objeto de investigación el **+1pto**, a continuación el trozo de código donde hacemos uso de dicha librería, dicho trozo de código se encuentra en la creación del objeto de investigación.

```
1 import { TextGeometry } from '../../../../../libs/TextGeometry.js';
2 import { FontLoader } from '../../../../../libs/FontLoader.js';
3 const loader = new FontLoader();
4 var mesh = null;
5
6 loader.load('../fonts/helvetiker_regular.typeface.json', function (font) {
7
8   const geometry = new TextGeometry('+1 pto', {
9     font: font,
10    size: 2,
11    depth: 1,
12    curveSegments: 12,
13    bevelEnabled: true,
14    bevelThickness: 0.05,
15    bevelSize: 0.05,
16    bevelOffset: 0,
17    bevelSegments: 3
18  });
19});
```

3. Manual de usuario

Por último pasamos a hacer un pequeño repaso del uso del juego a modo de manual sencillo. Lo primero que nos encontramos al entrar al juego es una pantalla de carga, cuando se complete la barra será buen momento de empezar a jugar pulsando el botón **Lets go**. Adicionalmente para activar la música en caso de no haber comenzado se le podría dar antes al botón **Sonido**.

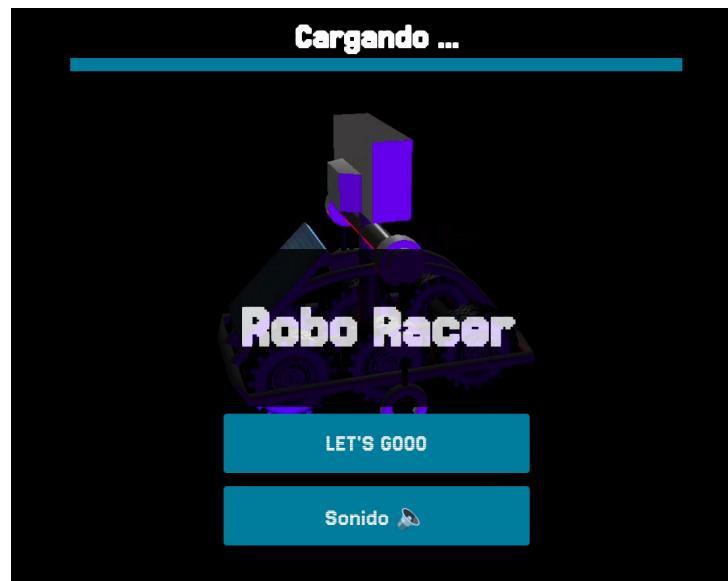


Figura 35: Pantalla de Carga.

Una vez cargado le daremos a el botón y nos saldrá el plano general del circuito, podremos jugar así pero será complicado por lo que debemos darle al espacio para cambiar a la cámara de tercera persona.

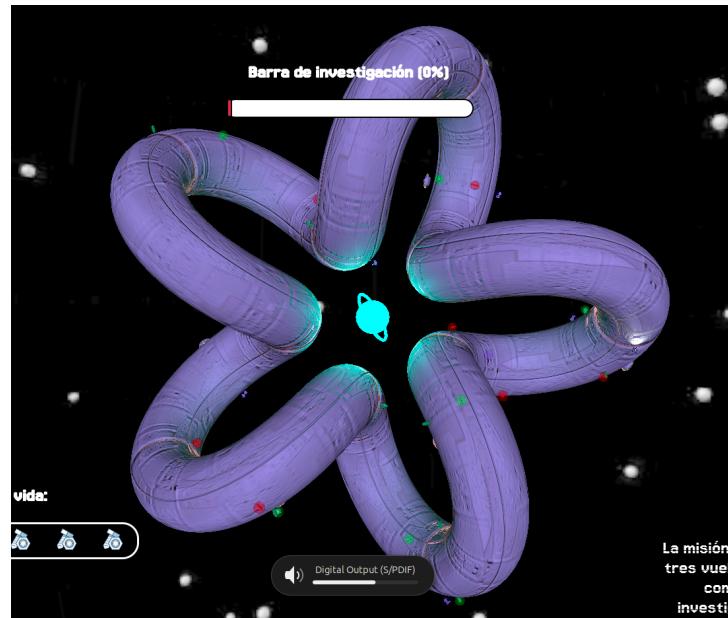


Figura 36: Plano general, Cámara alejada.

Una vez le hayamos dado estaremos listos para jugar. Simplemente nos faltara conocer los controles básicos:

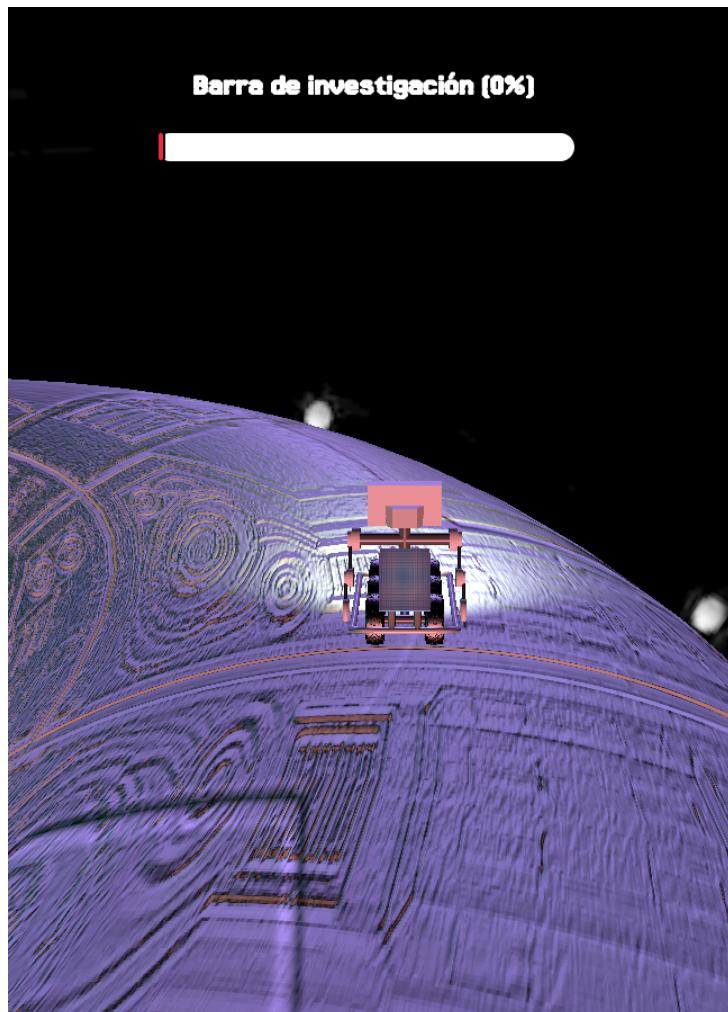


Figura 37: Salida Carrera.

CONTROLES BÁSICOS TECLADO

W: PARA AVANZAR
A: PARA IR A LA IZQUIERDA
D: PARA IR A LA DERECHA
S: PARA RETROCEDER
ESPACIO: PARA CAMBIAR DE CÁMARA

CONTROLES BÁSICOS RATÓN

CLICK IZQUIERDO: PARA ATACAR A LOS ENEMIGOS