

PURTADA

TABLA DE CONTENIDOS

CAPITULO 1	5
1. INTRODUCCIÓN	5
<i>¿Por qué este curso?</i>	5
<i>¿Qué aprenderás?</i>	5
<i>¿Cómo aprender este curso?</i>	6
<i>Las lecciones</i>	6
<i>El código</i>	6
<i>Requisito previo</i>	7
2. QUE ES WEBGL Y PORQUE USAR THREE.JS.....	7
<i>Introducción</i>	7
<i>¿Qué es WebGL?</i>	7
<i>Three.js al rescate</i>	8
<i>¿Qué pasa con otras bibliotecas?</i>	8
3. ESCENA BÁSICA.....	9
<i>Introducción</i>	9
<i>Archivos base</i>	9
<i>Cómo cargar Three.js</i>	10
<i>Cómo usar Three.js</i>	11
<i>Primera escena</i>	12
<i>Primer render</i>	15
4. SERVIDOR LOCAL.....	18
<i>Introducción</i>	18
<i>El estado de las herramientas de compilación</i>	18
<i>Vite</i>	18
<i>Node.js</i>	19
<i>Archivos Zip</i>	19
<i>Levantar el servidor</i>	20
<i>Solución de problemas</i>	20
<i>Mas acerca de la plantilla de Vite</i>	21
<i>Recupera nuestra escena</i>	21
5. TRANSFORMACIÓN DE OBJETOS	24
<i>Introducción</i>	24
<i>Configuración</i>	24
<i>Mover objetos</i>	25
<i>Ayudante de ejes</i>	26
<i>Escalar objetos</i>	27
<i>Rotar objetos</i>	28
<i>Combinando transformaciones</i>	31
<i>Gráfico de escena</i>	32
6. ANIMACIONES.....	33
<i>Introducción</i>	33
<i>Configuracion</i>	33
<i>Usando requestAnimationFrame</i>	34
<i>Usando una biblioteca</i>	38
<i>Elegir la solución adecuada</i>	39
7. CAMARAS.....	40
<i>Introducción</i>	40
<i>Cámara</i>	40
<i>ArrayCamera</i>	40
<i>StereoCamera</i>	40
<i>CubeCamera</i>	40
<i>OrthographicCamera</i>	40

<i>PerspectiveCamera</i>	40
PERSPECTIVECAMERA.....	40
<i>Campo de visión</i>	41
<i>Relación de aspecto</i>	41
<i>Cerca y lejos</i>	41
CÁMARA ORTOGRÁFICA	42
CONTROLES PERSONALIZADOS.....	42
CONTROLES INCORPORADOS	46
<i>DeviceOrientationControls</i>	46
<i>FlyControls</i>	46
<i>FirstPersonControls</i>	46
<i>PointerLockControls</i>	46
<i>OrbitControls</i>	47
<i>TrackballControls</i>	47
<i>TransformControls</i>	47
<i>DragControls</i>	47
ORBITCONTROLS	47
<i>Instanciar</i>	47
<i>Objetivo</i>	48
<i>Cuando usar los controles integrados</i>	49
8. PANTALLA COMPLETA Y REDIMENSIONAMIENTO.....	49
<i>Introducción</i>	49
<i>Configuración</i>	49
<i>Encajar en la ventana gráfica</i>	49
<i>Manejar cambio de tamaño</i>	52
<i>Manejar proporción de píxeles</i>	54
<i>Manejar pantalla completa</i>	57
9. GEOMETRIAS	60
10. DEBUG UI.....	65
11. TEXTURAS	66
12. MATERIALES.....	75
13. TEXTO 3D.....	85
14. GO LIVE	91
CAPITULO 2	92
15. LUCES	92
16. SOMBRAS	97
17. CASA ENCANTADA	105
18. PARTICULAS	106
19. GENERADOR DE GALAXIAS	114
20. ANIMACION BASADA EN SCROLL	121
CAPITULO 3	122
21. FISICA	122
22. MODELOS IMPORTADOS	150
23. RAYCASTER Y EVENTOS DE MOUSE.....	160
24. MODELOS CUSTOMIZADOS CON BLENDER	178
<i>Introduccion</i>	178
<i>Eligiendo el software</i>	178
<i>Descargando Blender</i>	178
<i>Interface</i>	178
<i>Shortcuts</i>	184
<i>Vista</i>	184
<i>Seleccionando</i>	190

<i>Creando objetos</i>	190
<i>Eliminando objetos</i>	192
<i>Ocultando objetos</i>	193
<i>Transformando Objetos</i>	193
<i>Modos</i>	193
<i>Sombreado</i>	194
<i>Propiedades</i>	195
<i>Motores de renderizado</i>	199
<i>Búsqueda</i>	200
<i>Guardar la configuración</i>	201
<i>Tiempo de hamburguesa</i>	204
<i>Exportar</i>	205
<i>Probar en Three.js</i>	205
<i>Ir mas allá</i>	205
25. MAPAS DE ENTORNO	205
26. RENDERIZADOS REALISTAS	205
27. ESTRUCTURA DE CODIGO PARA PROYECTOS GRANDES	205
CAPITULO 4	206
28. SOMBREADOS.....	206
29. PATRONES DE SOMBREADO	206
30. MAR FURIOSO	206
31. GALAXIA ANIMADA	206
32. MATERIALES MODIFICADOS	206
CAPITULO 5	207
33. POSTPROCESADO.....	207
34. TIPS DE DESEMPEÑO	207
35. INTRODUCCION Y PROCESO DE CARGA.....	207
36. MEZCLANDO HTML y WebGL	207
CAPITULO 6	208
37. CREANDO UNA ESCENA EN BLENDER.....	208
38. COCINANDO Y EXPORTANDO LA ESCENA	208
39. IMPORTANDO Y OPTIMIZANDO LA ESCENA	208
40. AGREGANDO DETALLES A LA ESCENA.....	208
CAPITULO 7	209
41. QUE ES REACT Y REACT THREE FIBER.....	209
42. PRIMERA APPLICACION EN REACT	209
43. PRIMERA APPLICACION EN R3F	209
44. DREI	209
45. DEBUG	209
46. AMBIENTE Y PUESTA EN ESCENA	209
47. CARGAR MODELOS.....	209
48. TEXTO 3D	209
49. ESCENA DE PORTAL	210
50. EVENTOS DE MOUSE.....	210
51. POSTPROCESADO.....	210
52. PORTAFOLIO DIVERTIDO Y SIMPLE	210
53. FISICA	210
54. CREAR UN JUEGO.....	234
55. EL FINAL	237

CAPITULO 1

1. Introducción

¿Por qué este curso?

Cuando comencé a aprender Three.js, no teníamos muchos recursos. La biblioteca estaba floreciendo, pero sin duda era nueva y parecía haber considerables lagunas en la documentación. En ese momento, tuvimos que avanzar y profundizar en el código y los ejemplos, encontrar hilos sobre problemas frecuentes y seguir intentándolo hasta que funcionó.

Hoy en día, la documentación es excelente. Tenemos tutoriales bastante decentes; la biblioteca es estable, aunque todavía está evolucionando gracias a la comunidad. Sin embargo, todavía falta un curso definitivo que te lleve de la mano. Sería útil si tuviera orientación para comprender completamente este tema y obtener suficiente experiencia para hacer sus proyectos sin luchar ni perderse.

Después de lanzar mi portafolio, muchas personas me preguntaron si podía darles tutoría o dónde aprender esas técnicas que uso. De hecho, he tenido la idea de crear un curso para mis alumnos durante mucho tiempo. Cuando recibí todas esas solicitudes, pensé que era hora de crear ese curso en línea para que todos lo exploraran.

¿Qué aprenderás?

Three.js es enorme y puede hacer un número infinito de cosas con él. Eso explica por qué el curso es tan largo.

Primero, comenzaremos aprendiendo todos los conceptos básicos, como crear nuestra primera escena, agregar objetos, elegir los materiales correctos, agregar texturas y animar todo.

Luego, pasaremos por muchas habilidades tradicionales: por ejemplo, crear nuestras propias geometrías, agregar luces y sombras, interactuar con los objetos 3D, agregar partículas.

Finalmente, terminaremos con técnicas más avanzadas como la física, produciendo renders realistas, escribiendo sombreadores personalizados, agregando pos-procesamiento e incluso creando nuestros propios modelos de Blender. ¡Así es! También aprenderá a usar Blender durante este curso.

Es bastante bueno crear experiencias impresionantes de WebGL, pero si necesita una PC de gama alta que pueda ejecutar juegos AAA a 140 fps para disfrutarla, no vale la pena. Aprenderemos cómo monitorear el desempeño y aplicar diferentes consejos para que nuestra experiencia WebGL funcione en tantos dispositivos como sea posible.

Algunas cosas pueden parecer desafiantes y no se preocupe, es perfectamente normal. Nadie puede entender todo en el primer intento. Cada lección se enfoca en una habilidad. Estas lecciones le harán probar dicha habilidad repetidamente en diferentes situaciones hasta que se sienta lo suficientemente cómodo.

Como soy un desarrollador creativo, también haré todo lo posible para ayudarlo a crear experiencias elegantes. Le daré muchos consejos y trucos para encontrar los ajustes perfectos para sus creaciones WebGL.

¿Cómo aprender este curso?

Descubra cómo puede aprender mejor Cada individuo tiene una forma única de aprender. Es una lección valiosa para que descubra cómo aprovechar al máximo este curso.

Mi recomendación es que la clase se ejecute en una segunda pantalla desde su computadora en funcionamiento, o tal vez una parte de su pantalla si esta es lo suficientemente grande. Aprenda siempre por la mañana. Ese es el momento del día en que tu cerebro es más eficiente. No intentes apresurar las cosas y completar todo el curso en solo una semana. El cerebro necesita tiempo para procesar elementos y lo hace mejor por la noche. Tal vez intente una o dos (tres como máximo) lecciones al día. Coma bien, duerma lo suficiente, beba y manténgase saludable. Y no olvide apagar cualquier distracción.

Por supuesto, no es necesario terminar toda la clase para comenzar a trabajar en proyectos, ya sean personales o profesionales. Si tiene ganas de abandonar y quiere empezar a crear por su cuenta, ¡adelante! El curso estará allí esperándote si lo necesitas.

Las lecciones

Cada lección cubre un tema, y es mejor seguirlos en orden, pero si ya se siente cómodo con los conceptos básicos, puede intentar saltar directamente a técnicas más avanzadas como los sombreadores.

También encontrarás un archivo zip final por lección. Es como el archivo zip inicial, pero muestra la tarea terminada si la necesita.

Cada lección se graba en video, pero recuerde, también tiene acceso a una versión de texto en la misma página. El código formateado implica la coloración de la sintaxis, y en cada paso del camino, también tiene acceso a vistas previas del resultado que se supone que debe lograr.

Durante cada clase, revisaremos varias páginas de documentación, herramientas en línea y ejemplos. Todos los enlaces estarán disponibles en "Enlaces de lecciones" en la parte inferior derecha de cada página. Continúe y haga clic en ellos cuando desee profundizar en cualquiera de los temas que cubrimos.

El código

Cada desarrollador tiene su manera de codificar y yo tengo la mía. Durante la lección, en el archivo zip inicial y en el archivo zip final, el código seguirá mis preferencias, pero haz lo que deseas. Si desea refactorizar todo el código, adelante.

Requisito previo

Obviamente, sería mejor si tuviera una computadora. Algunas partes del curso pueden variar según el sistema operativo, como los comandos de la terminal o algunos breves de Blender. Siempre proporcionaré las versiones de macOS y Windows.

Este curso es amigable para principiantes, pero necesita conocer los conceptos básicos de JavaScript como variables, bucles, funciones, eventos y, quizás, módulos. No se preocupe por este último punto; aprenderá los conceptos básicos del módulo siguiendo las siguientes lecciones.

También necesitará un editor de código. Recomiendo Visual Studio Code porque es un editor bien engrasado con una amplia comunidad y actualizaciones extendidas cada mes, pero use el editor que más le convenga. Habrá algunos casos en los que instalaremos complementos, pero nuevamente, no será un gran problema y probablemente encontrará los complementos correspondientes para su editor de código.

Otro requisito es un navegador moderno con herramientas útiles para desarrolladores. Usaré Chrome, pero también puedes usar Firefox. No recomiendo otros navegadores porque necesitamos que nuestras herramientas sean robustas y eficientes. En algún momento, instalaremos una extensión de Chrome para ayudarnos a monitorear el rendimiento de WebGL, pero esta parte es opcional.

También aprenderemos a usar Blender. Es un software 3D gratuito que funciona en todos los sistemas operativos principales, y es simplemente genial. Si tienes curiosidad, pruébalo ya.

No necesitas ser bueno en matemáticas. Soy terrible en eso, y aun así, me las arreglo para crear experiencias impresionantes. Sí, ser bueno en eso sería una ventaja, pero puedes prescindir de él, y te explicaré las fórmulas.

Finalmente, si no sabe nada sobre Three.js, está en el lugar correcto.

2. Que es WebGL y porque usar Three.js

Introducción

Si está aquí, probablemente ya sepa qué es Three.js, pero hablemos de ello y veamos por qué necesitamos esta biblioteca.

Three.js es una biblioteca de JavaScript 3D que permite a los desarrolladores crear experiencias 3D para la web. Funciona con WebGL, pero también puede funcionar con SVG y CSS. Esos dos son bastante limitados y no los cubriremos en este curso.

¿Qué es WebGL?

WebGL es una API de JavaScript que procesa triángulos en un lienzo a una velocidad notable. Es compatible con la mayoría de los navegadores modernos y es rápido porque utiliza la Unidad de procesamiento gráfico (GPU) del visitante.

WebGL puede dibujar más que triángulos y también se puede utilizar para crear experiencias en 2D, pero nos centraremos en las experiencias en 3D utilizando triángulos por el bien del curso.

La GPU puede realizar miles de cálculos en paralelo. Imagina que quieras renderizar un modelo 3D y este modelo está constituido por 1000 triángulos, que, pensándolo bien, no son tantos. Cada triángulo incluye 3 puntos. Cuando queramos renderizar nuestro modelo, la GPU tendrá que calcular la posición de estos 3000 puntos. Debido a que la GPU puede hacer cálculos paralelos, manejará todos los puntos de los triángulos de una sola vez.

Una vez que los puntos del modelo están bien ubicados, la GPU debe dibujar cada píxel visible de esos triángulos. Una vez más, la GPU manejará los cálculos de miles y miles de píxeles de una sola vez.

Las instrucciones para colocar los puntos y dibujar los píxeles están escritas en lo que llamamos "shaders". Y déjame decirte que los shaders son difíciles de dominar. También necesitamos

proporcionar datos a estos shaders. Por ejemplo: cómo colocar los puntos según las transformaciones del modelo y las propiedades de la cámara. Estos se llaman matrices. También necesitamos proporcionar datos para ayudar a colorear los píxeles. Si hay una luz y el triángulo está frente a esa luz, debería ser más brillante que si el triángulo no lo estuviera.

Y esta es la razón por la que WebGL nativo es tan difícil. Dibujar un solo triángulo en el lienzo requeriría al menos 100 líneas de código. Buena suerte si quieras agregar perspectiva, luces, modelos y animar todo en ese caso.

Pero el WebGL nativo se beneficia de existir en un nivel bajo, muy cerca de la GPU. Esto permite excelentes optimizaciones y más control.

Three.js al rescate

Three.js es una biblioteca de JavaScript con licencia MIT que funciona justo encima de WebGL. El objetivo de la biblioteca es simplificar drásticamente el proceso de manejo de todo lo que acabamos de decir. Tendrá una escena animada en 3D en solo unas pocas líneas de código y no tendrá que proporcionar sombreadores ni matrices.

Debido a que Three.js está justo encima de WebGL, aún podemos interactuar con él de alguna manera. En algún momento, llegaremos a escribir sombreadores y crear matrices.

Ricardo Cabello, también conocido como Mr. doob (sitio web, Twitter), es el desarrollador que creó Three.js. Todavía está trabajando en ello, pero ahora cuenta con la ayuda de una gran comunidad. Puede consultar la lista de colaboradores aquí:

<https://github.com/mrdoob/three.js/graphs/contributors>

Actualmente, la biblioteca recibe una actualización cada mes y puede ver qué ha cambiado en la página de lanzamientos aquí: <https://github.com/mrdoob/three.js/releases>

Puede descubrir muchos proyectos excepcionales utilizando Three.js en la página de inicio del sitio web:

<https://threejs.org/>

También hay documentación bien mantenida que usaremos mucho:

<https://threejs.org/docs/index.html#manual/en/introduction/Creating-a-scene>

Y puede encontrar cientos de ejemplos con código público aquí:

https://threejs.org/examples/#webgl_tonemapping

Si quieres seguir las actualizaciones y descubrir proyectos excepcionales, te aconsejo que sigas las cuentas de Twitter de Mr. doob y Three.js:

<https://twitter.com/mrdoob>

https://twitter.com/threejs_org

¿Qué pasa con otras bibliotecas?

Three.js es la biblioteca WebGL más popular por buenas razones que ya cubrimos. Es muy estable, proporciona muchas funciones, la documentación es notable, la comunidad está trabajando duro en las actualizaciones y todavía está lo suficientemente cerca del WebGL nativo.

Es por eso que probablemente sea mejor aprender Three.js.

Pero hay muchas otras bibliotecas, y algunas de ellas también son increíbles. Sea curioso, pruébelos y experimente su fuerza por sí mismo. Incluso puede aprender cosas que serán útiles para sus proyectos Three.js.

3. Escena básica

Introducción

Para nuestra primera lección, haremos que Three.js funcione de la manera más sencilla: sin paquete, sin dependencia, sin módulos, solo un archivo HTML y algo de JavaScript.

Archivos base

comenzar, cree un archivo index.html simple:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>03 - Basic Scene</title>
</head>
<body>
    <script src=".//script.js"></script>
</body>
</html>
```

Y un simple console.log ()



```
console.log('Hello Three.js')
```

Abra index.html en su navegador. Para hacer eso, puede intentar hacer doble clic en el archivo. Si no funciona o se abre con el navegador incorrecto, puede arrastrar y soltar el archivo en el navegador correcto. Incluso si Three.js funciona en la mayoría de los navegadores, recomiendo usar un navegador compatible con desarrolladores como Chrome o Firefox.

Ahora abra las herramientas de desarrollo.

Para hacer eso, puede hacer clic derecho en cualquier lugar de la página y elegir Inspeccionar o puede presionar F12 en Windows y CMD + OPCIÓN + I en MacOS.

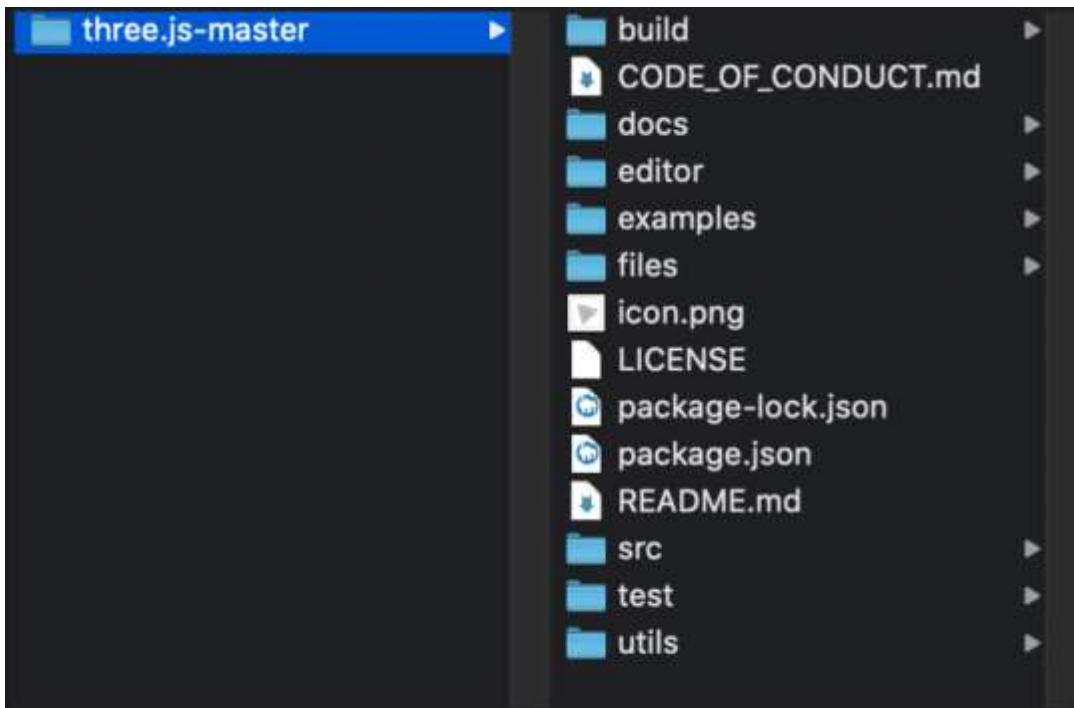
Luego navegue a la pestaña Consola en la parte superior de Herramientas para desarrolladores.

Debe mantener la consola abierta en todo momento para ver posibles errores y advertencias.

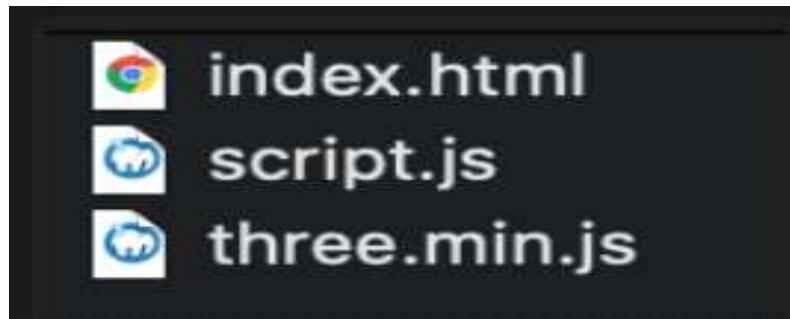
Cómo cargar Three.js

Ahora necesitamos cargar la biblioteca Three.js. Hay muchas formas de hacerlo. Por ahora, simplemente descargaremos la biblioteca y la cargaremos usando <script>.

Vaya a <https://threejs.org/> y haga clic en el botón de descarga para descargar un archivo zip y descomprimirlo. El archivo es bastante pesado, pero no se preocupe, solo necesitamos un archivo. Debería obtener una carpeta que se parece a esto:



Vaya a la carpeta build / y copie el archivo three.min.js a su proyecto. Debería obtener algo como esto:



Ahora podemos cargar la biblioteca Three.js al final del <body>, justo antes de cerrarla:

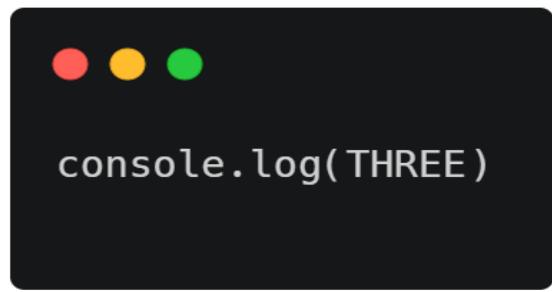


Asegúrese de cargar three.min.js antes de su script.js; de lo contrario, tu script no conocerá lo que hay dentro del archivo three.min.js.

Cómo usar Three.js

Dentro de nuestro archivo script.js, ahora tenemos acceso a una variable llamada THREE. Tenga cuidado y escríbalo siempre en mayúsculas.

Si muestras esta variable mediante un console.log(), veras que están sucediendo muchas cosas dentro:



La variable THREE contiene la mayoría de las clases y propiedades que puede necesitar en un proyecto clásico de Three.js. Desafortunadamente, no todas las clases están dentro de esta variable, pero veremos más adelante cómo acceder a ellas.

Para usar una de esas clases, debes primero instanciarla.

Por ejemplo, si quieras crear una escena, debes escribir `const scene = new THREE.Scene()`. Si desea crear una geometría de esfera, debe escribir `const sphereGeometry = new THREE.SphereGeometry(1.5, 32, 32)`. Profundizaremos en esto más adelante.

Primera escena

Es hora de crear nuestra escena y producir algo en la pantalla.

Necesitamos 4 elementos para comenzar:

- Una escena que contendrá objetos.
- Algunos objetos
- Una cámara
- Un renderizador

Escena

La escena es como un contenedor. Coloca sus objetos, modelos, partículas, luces, etc. en él, y en algún momento, le pide a Three.js que renderice esa escena.

Para crear una escena, use la clase Scene:



Objetos

Los objetos pueden ser muchas cosas. Puede tener geometrías primitivas, modelos importados, partículas, luces, etc. Empezaremos con un simple cubo rojo. Para crear ese cubo rojo, necesitamos crear un tipo de objeto llamado Malla también conocido como "*mesh*". Una malla es la combinación de una geometría (la forma) y un material (cómo se ve).

Hay muchas geometrías y muchos materiales, pero mantendremos las cosas simples por ahora y crearemos un BoxGeometry y un MeshBasicMaterial.

Para crear la geometría, usamos la clase BoxGeometry con los primeros 3 parámetros que corresponden al tamaño de la caja.

```
// Object  
const geometry = new THREE.BoxGeometry(1, 1, 1)
```

Para crear el material, usamos la clase `MeshBasicMaterial` con un parámetro: un objeto {} que contiene todas las opciones. Todo lo que necesitamos es especificar su propiedad de color.

Hay muchas formas de especificar un color en Three.js. Puede enviarlo como un JS hexadecimal `0xff0000`, puede enviarlo como una cadena hexadecimal '# ff0000', puede usar nombres de colores como 'rojo', o puede enviar una instancia de la clase `Color`; cubriremos más sobre eso más tarde.

```
// Object  
const geometry = new THREE.BoxGeometry(1, 1, 1)  
const material = new THREE.MeshBasicMaterial({ color: 0xff0000 })
```

Para crear la malla final, usamos la clase `Mesh` y enviamos la geometría y material como parámetros.

```
// Object  
const geometry = new THREE.BoxGeometry(1, 1, 1)  
const material = new THREE.MeshBasicMaterial({ color: 0xff0000 })  
const mesh = new THREE.Mesh(geometry, material)
```

Ahora puede agregar su malla a la escena usando el método `add (...)`:

```
scene.add(mesh)
```

Si no agrega un objeto a la escena, no podrá verlo.

Cámara

La cámara no es visible. Es más, como un punto de vista teórico. Cuando hacemos un render de la escena, será desde el punto de vista de esa cámara.

Puede tener varias cámaras como en un set de película, y puede cambiar entre esas cámaras como desee. Por lo general, solo usamos una cámara.

Hay diferentes tipos de cámaras y hablaremos de ellas en una lección futura. Por ahora, simplemente necesitamos una cámara que maneje la perspectiva (haciendo que los objetos cercanos parezcan más prominentes que los lejanos).

Para crear la cámara, usamos la clase PerspectiveCamera. Hay dos parámetros esenciales que debemos proporcionar.

El campo de visión

El campo de visión es lo grande que es su ángulo de visión. Si usa un ángulo muy grande, podrá ver en todas las direcciones a la vez, pero con mucha distorsión, porque el resultado se dibujará en un pequeño rectángulo. Si usa un ángulo pequeño, las cosas se verán ampliadas. El campo de visión(*fov*) se expresa en grados y corresponde al ángulo de visión vertical. Para este ejercicio usaremos un ángulo de 75 grados.

La relación de aspecto

En la mayoría de los casos, la relación de aspecto es el ancho del lienzo dividido por su altura. No hemos especificado ningún ancho o alto por ahora, pero lo necesitaremos más adelante. Mientras tanto, crearemos un objeto con valores temporales que podremos reutilizar.

No olvide agregar su cámara a la escena. Todo debería funcionar sin agregar la cámara a la escena, pero podría resultar en errores más adelante:

```
// Sizes
const sizes = {
  width: 800,
  height: 600
}

// Camera
const camera = new THREE.PerspectiveCamera(75, sizes.width / sizes.height)
scene.add(camera)
```

Renderizador

El trabajo del renderizador es hacer el render. ¿Apuesto a que no lo viste venir?

Simplemente le pediremos al renderizador que renderice nuestra escena desde el punto de vista de la cámara, y el resultado se dibujará en un lienzo. Puede crear el lienzo usted mismo o dejar que el renderizador lo genere y luego agregarlo a su página. Para este ejercicio, agregaremos el lienzo al html y lo enviaremos al renderizador.

Cree el elemento <canvas> antes de cargar los scripts y dele una clase:



```
<canvas class="webgl"></canvas>
```

Para crear el renderizador, usamos la clase `WebGLRenderer` con un parámetro: un objeto {} que contiene todas las opciones. Necesitamos especificar la propiedad del lienzo correspondiente al `<canvas>` que agregamos a la página.

Cree una variable de lienzo al comienzo del código, luego busque y almacene en ella el elemento que creamos en el HTML usando `document.querySelector(...)`.

Es mejor asignar el lienzo a una variable porque lo usaremos para otros propósitos en las próximas clases.

También necesitamos actualizar el tamaño de su renderizador con el método `setSize(...)` usando el objeto de tamaños que creamos anteriormente. El método `setSize(...)` cambiará automáticamente el tamaño de nuestro `<canvas>` de acuerdo a:

```
// Canvas
const canvas = document.querySelector('canvas.webgl')

// ...

// Renderer
const renderer = new THREE.WebGLRenderer({
  canvas: canvas
})
renderer.setSize(sizes.width, sizes.height)
```

De momento, no verás nada, pero tu lienzo está allí y su tamaño se ha ajustado acordemente. Puedes usar las herramientas de desarrollador para inspeccionar el elemento `<canvas>` si tienes curiosidad.

Primer render

Es hora de trabajar en nuestro primer render. Llame al método `render(...)` en el renderizador y envíe la escena y la cámara como parámetros:



```
renderer.render(scene, camera)
```

¿Todavía nada? Aquí está el problema: no hemos especificado la posición de nuestro objeto, ni la de nuestra cámara. Ambos están en la posición predeterminada, que es el centro de la escena y no podemos ver un objeto desde su interior (por defecto).

Necesitamos mover cosas.

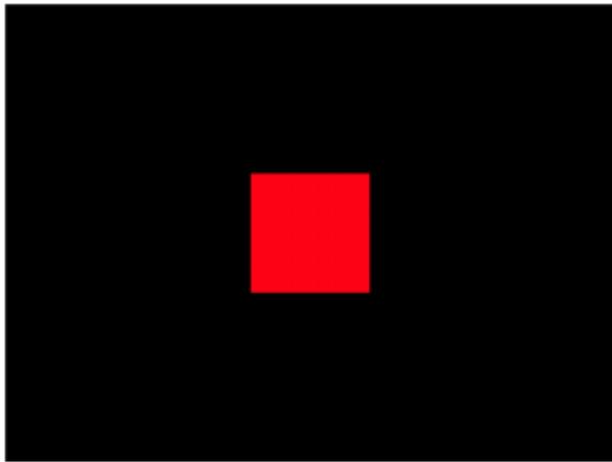
Para hacer eso, tenemos acceso a múltiples propiedades en cada objeto, como posición, rotación y escala. Por ahora, use la propiedad de posición para mover la cámara hacia atrás.

La propiedad de posición es un objeto con tres propiedades relevantes: x, y z. De forma predeterminada, Three.js considera que el eje de avance / retroceso es z.

Para mover la cámara hacia atrás, debemos proporcionar un valor positivo a esa propiedad. Puede hacerlo en cualquier lugar una vez que haya creado la variable de la cámara, pero tiene que suceder antes de hacer el renderizado:



```
const camera = new THREE.PerspectiveCamera(75, sizes.width / sizes.height)
camera.position.z = 3
scene.add(camera)
```



Felicitaciones, debería ver su primer renderizado. Parece un cuadrado, y eso se debe a que la cámara se alinea perfectamente con el cubo y solo se puede ver un lado.

No se preocupe por el tamaño del render; aprenderemos cómo hacer que el lienzo se ajuste a la ventana gráfica más adelante.

En las próximas clases, aprenderá más sobre las propiedades de posición, rotación y escala, cómo cambiarlas y animar la escena.

4. Servidor local

Introducción

La forma en que cargamos Three.js en la lección anterior es la más sencilla. Desafortunadamente, tiene algunas limitaciones.

La primera es que solo tenemos acceso a las clases principales. Hay docenas de clases en este núcleo y podemos hacer muchas cosas con ellas, pero no tenemos acceso a todos los recursos disponibles. Como ejemplo, en una futura lección, necesitaremos acceso a la clase OrbitControls, la cual no está disponible en las clases principales.

La segunda es que al abrir un archivo HTML de esta manera, el navegador no dejará que Javascript ejecute ninguna instrucción. Como un ejemplo, no serás capaz de cargar archivos locales como texturas o modelos. De hecho, esto es algo bueno por razones de seguridad. No es recomendable que un script pueda cargar archivos desde tu computador solo porque abriste un archivo HTML que pensabas era seguro.

Pero aun necesitamos ser capaces de ejecutar código _JavaScript como si fuera un sitio web y para hacer eso, necesitamos ejecutar un servidor local.

Hay muchas maneras de manipular estos problemas, pero la mas simple solución es usar la herramienta “build tool” o “bunder”.

El estado de las herramientas de compilación

Hay muchas herramientas de compilación (tambien conocidas como "*build tools*") y probablemente has escuchado hablar de alguna de ellas como Webpack, Vite, Gulp, Parcel, etc.

Todas ellas tienen variadas características con sus respectivas ventajas y desventajas, pero estaremos utilizando una muy específica en las siguientes lecciones.

Hoy en día, la herramienta de compilación más popular es Webpack. Es ampliamente utilizada, tiene una gran comunidad y hay una gran cantidad de cosas que puedes lograr con ella. Sin embargo, aunque Webpack sea la más popular, no es precisamente la más apreciada.

De hecho, la herramienta más apreciada en estos días es "*Vite*". Más rápida al instalarse, más rápida al ejecutarse y menos propensa a errores. En fin, la experiencia para el desarrollador es mucho mejor.

Inicialmente todos los ejercicios corrían en Webpack y casi todas las lecciones lo usaban. Pero ahora los ejercicios se ejecutan en Vite. No te preocupes, hemos configurado Vite de manera que los archivos y el comportamiento de Vite luzca muy similar a Webpack.

Vite

As mentioned earlier, Vite is a build tool. The idea is that we are going to write web code like HTML/CSS/JS and Vite will build the final website. It'll also do a bunch of things like optimizations, cache breaking, source mapping, running a local server, etc.

While Vite handles the most basic needs, we can also add plugins in order to handle more features like exotic languages, or special files. We are actually going to add a plugin later in the course, which will be able to handle GLSL files in order to create custom shaders.

Vite was created by Evan You, the creator of Vue.js, and is highly maintained by hundreds of developers.

Node.js

Primero necesitas tener Node.js instalado en tu computadora.

Node.js permite ejecutar Javascript en tu computadora fuera del explorador. Es muy recomendable para correr herramientas como Vite. Lleva existiendo mucho tiempo y es bastante popular.

Si no sabes si ya tienes Node.js instalado en tu computadora o que versión está instalada, abre tu terminal (MacOs) o símbolo del sistema (Windows) y ejecuta el siguiente comando: node -v.

Si la respuesta te indica que node no se reconoce como un comando entonces no está instalado.

Si está instalado, entonces la respuesta contendrá la versión actual. Asegurate que este actualizada con la última versión. Al momento en el que estoy escribiendo esto Vite funciona con la versión 14.18 y mayores, sin embargo, recomiendo que siempre tengas instalada la última versión LTS.

Para instalar Node.js, visite <https://nodejs.org/en/>, descargue la versión "LTS" e instálelo con las configuraciones predeterminadas.

Cierre su terminal (MacOs) o símbolo de sistema (Windows), ábralo de nuevo, y ejecute nuevamente el comando: node -v nuevamente para verificar la versión.

Archivos Zip

Ahora que hemos instalado Node.js en nuestros computadores, Podemos correr el iniciador.

Descarga y descomprime el archive iniciador con extensión zip.

Nuevamente, nos apoyaremos de algunos commandos en la terminal. Podrías usar la terminal (MacOs) o el símbolo de Sistema (Windows) como hicimos previamente, pero te recomiendo utilizar VSCode y hacer uso de la terminal integrada.

Abre VSCode y presiona CMD + J (MacOs) o CTRL + J (Windows) para abrir la terminal integrada.

La terminal funciona un poco como tu explorador de archivos y necesitarás estar en la carpeta correcta para correr los commandos. En nuestro caso, necesitamos mover nuestra terminal dentro de la carpeta del Proyecto que acabamos de descomprimir.

Puedes usar cd seguido del nombre de la carpeta para navegar dentro de ella. También puedes usar cd (con un espacio al final), y arrastrar la carpeta a la que quieras acceder. Puedes probar en donde te encuentras con pwd y enlistar los archivos en la carpeta actual con ls.

Dependencias

Ahora que estamos en la carpeta del Proyecto, necesitamos instalar dependencias. Que dependencias? Bueno, tenemos dos dependencias en este proyecto: Vite y Three.js.

Para instalarlos, ejecuta el comando npm install desde tu terminal.

Espera un poco y deberías ver una carpeta denominada node_modules creada en la carpeta del proyecto.

Cuando instalamos Node.js, automáticamente instalamos NPM. NPM es un administrador de dependencias que buscará las dependencias enlazadas en el archivo package.json y las instalará en la carpeta node_modules.

Levantar el servidor

Ya casi estamos listos y ahora podemos pedirle a Vite que corra el servidor.

Para hacerlo, estando aun en la terminal y en la carpeta del proyecto, ejecutamos el commando npm run dev.

Espere un segundo o dos y el sitio web deberia abrirse en tu explorador por defecto con un mensaje que lee "*Soon to be a Three.js experience*" escrito en la pagina.

Si la pagina no abre, la terminal deberia mostrar dos URLs que se ven algo como <http://localhost:5173/> y <http://192.168.1.25:5173/>. Intenta acceder a ellas en tu navegador.

Solución de problemas

Si todo salio bien, te puedes saltar esta sección.

Si te encontraste con algun problema durante el proceso de instalacion, enumerare algunos problemas clasicos y como resolverlos.

Carpeta en terminal

Asegurate de que tu terminal este abierto en la carpeta del proyecto.

Utiliza el commando pwd para mostrar la carpeta en la que la terminal se encuentra actualmente.

Utilice cd (con un espacio al final) y arrastre la carpeta en cuestion, posteriormente presione enter para mover la terminal a esa carpeta.

Ruta larga

Si la carpeta de tu proyecto se encuentra anidada muy profundamente dentro de otras carpetas, puede que hayas terminado en una ruta tan larga que Node.js no pueda manejarla.

Mueve la carpeta algunos niveles arriba y asurate de mover la terminal a la par antes de intentar nuevamente.

Carpeta versionada

Ten cuidado con herramientas como OneDrive, Google Drive, Dropbox, etc. Las cuales guardan tus archivos en linea.

Pueden crear conflictos con tus dependencias de Node.js.

Intenta mover tu proyecto fuera de esas carpetas guardadas.

Caracteres especiales en la ruta

Evita cualquier carácter especial in la ruta de la raíz de tu computador hasta el proyecto.

Idealmente, deberías solo tener letras minúsculas, números, guiones y guiones bajos.

MacOs y Xcode

MacOs puede pedirte que instales "las herramientas de línea de comando de Xcode".

Generalmente es una advertencia inofensiva, pero en ocasiones crea problemas de ejecución para los proyectos de Node.js

A la fecha que escribo esto, este link parece contener más información al respecto

<https://medium.com/@mrjohnkilonzi/how-to-resolve-no-xcode-or-clt-version-detected-d0cf2b10a750>. Para resumirlo, corre xcode-select – install desde tu terminal y sigue las instrucciones.

Permisos

Algunas veces, los permisos no estan definidos correctamente y NPM no puede instalar las dependencias.

Si sabes lo que estas haciendo, intenta arreglar los permisos. Si no es así, borra la carpeta, vuelve a descargar el iniciador y sigue las instrucciones de nuevo.

Mas acerca de la plantilla de Vite.

Hay algunas cosas que necesitas saber acerca de la plantilla de Vite antes de que continuemos:

- La configuración de Vite esta definida en el archivo vite.config.js. Si tienes curiosidad de como configurar un proyecto de Vite, revisa este link <https://vitejs.dev>.
- Necesitas correr npm install solamente una vez despues de descargar el proyecto.
- Necesitas correr npm run dev cada vez que quieras correr el servidor y empezar a codificar. Tu terminal puede parecer congelada, Pero es perfectamente normal, y significa que el servidor esta corriendo.
- Presiona CTRL + C para detener el servidor. Puede que necesites presionar este atajo multiples veces en sistemas Windows o confirmar con la letra o.
- Las únicas carpetas que necesitas explorar son las carpetas src/ y static/
- En la carpeta src/ puedes encontrar los archivos tradicionales index.html, script.js y style.css.
- Puedes meter los “archivos estáticos” dentro de la carpeta static/. Esos archivos serán servidos como si estuvieran disponibles en la carpeta raíz (sin tener que escribir static/). Puedes probarlo al agregar /door.jpg al final de la url (<http://localhost:5173/door.jpg>). Usaremos esto para cargar texturas y modelos posteriormente.
- La pagina automáticamente se recargara cuando guardes cualquiera de estos archivos.
- Puedes acceder a tu servidor local desde cualquier otro dispositivo en la misma red al escribir la misma URL que se ve algo como esto: <http://192.168.1.25:5173>. Es muy útil debuggear en otros dispositivos como teléfonos móviles. Si no esta funcionando, usualmente es gracias al firewall.

Recupera nuestra escena

Aquí está la parte fácil. Queremos recuperar nuestra escena en esta plantilla de Vite.

Primero, debe agregar el <canvas> al archivo src / index.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>04 - Local Server</title>
    <link rel="stylesheet" href="./style.css">
</head>
<body>
    <canvas class="webgl"></canvas>
    <script type="module" src="./script.js"></script>
</body>
</html>
```

Ahora necesitamos agregar el mismo código JavaScript de la lección anterior en el archivo /src/script.js. La única diferencia es que necesitamos importar Three.js primero.

Para importar Three.js, necesitamos escribir la siguiente línea al principio de el archivo /src/script.js:

```
import * as THREE from 'three'
```

Esto importará todas las clases núcleo de Three.js dentro de la variable THREE. El módulo three esta en la carpeta /node_modules/, pero no necesitas tocarla.

Despues podemos seguir con el mismo código que antes:

```
import * as THREE from 'three'

// Scene
const scene = new THREE.Scene()

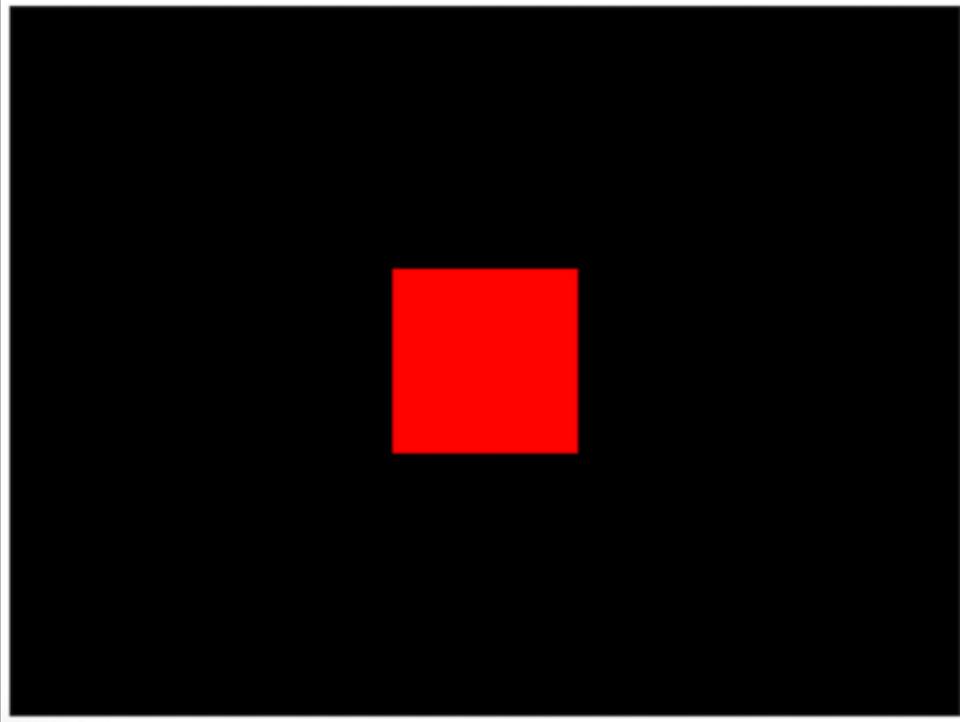
// Object
const geometry = new THREE.BoxGeometry(1, 1, 1)
const material = new THREE.MeshBasicMaterial({ color: 0xff0000 })
const mesh = new THREE.Mesh(geometry, material)
scene.add(mesh)

// Sizes
const sizes = {
    width: 800,
    height: 600
}

// Camera
const camera = new THREE.PerspectiveCamera(75, sizes.width /
sizes.height)
camera.position.z = 3
scene.add(camera)

// Renderer
const renderer = new THREE.WebGLRenderer({
    canvas: document.querySelector('canvas.webgl')
})
renderer.setSize(sizes.width, sizes.height)
renderer.render(scene, camera)
```

Si el servidor ya estaba corriendo, abre la página (no hay necesidad de recargar)
Si no, corre npm run dev desde la terminal y la pagina debería abrirse.



Y eso es todo. Tenemos el mismo código que en la lección anterior, pero esta vez con una herramienta de compilación que se ocupe del servidor local y haga un montón de otras optimizaciones por nosotros.

Si son te sientes confiado de hacerlo tu mismo, no te preocupes, lo haremos juntos la siguiente lección.

5. Transformación de objetos

Introducción

Ahora que tenemos todo en su lugar, podemos explorar las funcionalidades de Three.js.

Antes de animar nuestra escena, necesitamos saber cómo transformar objetos en nuestra escena.

Ya lo hemos hecho con la cámara moviéndola hacia atrás usando `camera.position.z = 3`.

Hay 4 propiedades para transformar objetos en nuestra escena.

- `position` (mover el objeto)
- `scale` (para cambiar el tamaño del objeto)
- `rotation` (para rotar el objeto)
- `quaternion` (para rotar también el objeto; más sobre eso más adelante)

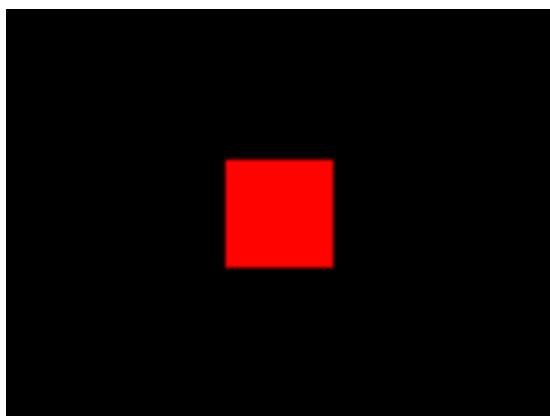
Todas las clases que heredan de la clase `Object3D` poseen esas propiedades como `PerspectiveCamera` o `Mesh` y clases que aún no hemos cubierto.

Puede ver de qué clases heredan cada clase en la parte superior de la documentación de Three.js.

Esas propiedades se compilarán en lo que llamamos matrices. Las matrices son utilizadas internamente por Three.js, por WebGL y por la GPU para transformar cosas. Afortunadamente, no tiene que manejar las matrices usted mismo y puede modificar las propiedades mencionadas anteriormente.

Configuración

En el iniciador, todo lo que tenemos es el proyecto como lo dejamos en la lección anterior con un cubo en el centro de la vista.



Mover objetos

La posición posee 3 propiedades esenciales, que son x, y z. Esos son los 3 ejes necesarios para posicionar algo en un espacio 3D.

La dirección de cada eje es puramente arbitraria y puede variar según el entorno. En Three.js, generalmente consideramos que el eje y va hacia arriba, el eje z va hacia atrás y el eje x va hacia la derecha.

En cuanto al significado de 1 unidad, depende de ti. 1 puede ser de 1 centímetro, 1 metro o incluso 1 kilómetro. Te recomiendo que adaptes la unidad a lo que quieras construir. Si va a crear una casa, probablemente debería pensar en 1 unidad como 1 metro.

Puedes jugar con la propiedad de posición de su malla e intentar adivinar dónde llegará el cubo (intente mantenerlo en la pantalla).

Asegúrese de hacer eso antes de llamar al método render (...) o renderizará la malla antes de moverla.



```
mesh.position.x = 0.7  
mesh.position.y = - 0.6  
mesh.position.z = 1
```

La propiedad de posición no es cualquier objeto. Es una instancia de la clase Vector3. Si bien esta clase tiene una propiedad x, y y z, también tiene muchos métodos útiles.

Puede obtener la longitud de un vector:



```
console.log(mesh.position.length())
```

Puede obtener la distancia de otro Vector3 (asegúrese de usar este código después de crear la cámara):

```
console.log(mesh.position.distanceTo(camera.position))
```

Puedes normalizar sus valores (o sea que reducirás la longitud del vector a 1 unidad para preservar su dirección):

```
console.log(mesh.position.normalize())
```

Para cambiar los valores, en lugar de cambiar x, y z por separado, también puede usar el método set (...):

```
mesh.position.set(0.7, - 0.6, 1)
```

Ayudante de ejes

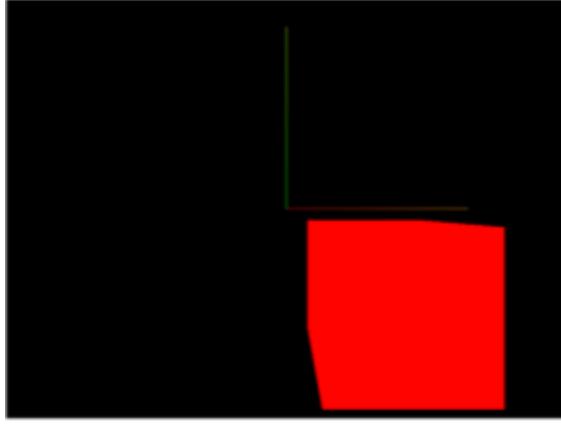
Antes de continuar, como puede ver, colocar cosas en el espacio puede ser un verdadero desafío. Saber dónde está orientado cada eje es complicado sobre todo cuando empezamos a mover la cámara.

Una buena solución es utilizar Three.js AxesHelper.

El AxesHelper mostrará 3 líneas correspondientes a los ejes x, y, z, cada una comenzando en el centro de la escena y yendo en la dirección correspondiente.

Para crear AxesHelper, cree una instancia y agréguelo a la escena justo después de crear una instancia de esa escena. Puede especificar la longitud de las líneas como único parámetro. Vamos a utilizar 2:

```
/**  
 * Axes Helper  
 */  
const axesHelper = new THREE.AxesHelper(2)  
scene.add(axesHelper)
```



Debería ver una línea verde y una roja.

La línea verde corresponde al eje y. La línea roja corresponde al eje x y hay una línea azul correspondiente al eje z pero no podemos verla porque está perfectamente alineada con la cámara.

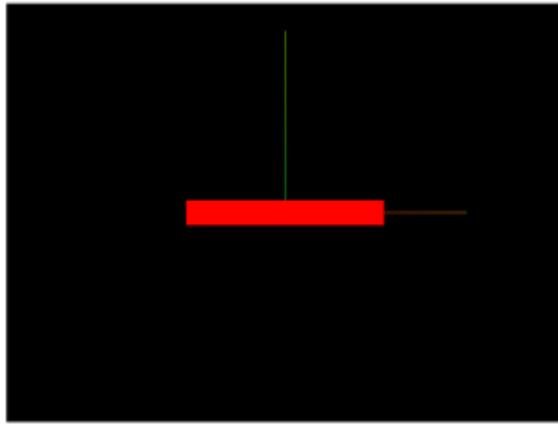
No usaremos este AxesHelper en las próximas elecciones, pero siéntase libre de agregarlo si necesita una referencia visual.

Escalar objetos

scale también es un Vector3. Por defecto, x, y y z son iguales a 1, lo que significa que el objeto no tiene ninguna escala aplicada. Si pones 0.5 como valor, el objeto tendrá la mitad de su tamaño en este eje, y si pones 2 como valor, será el doble de su tamaño original en este eje.

Si cambia esos valores, el objeto comenzará a escalar en consecuencia. Comenta la posición y agrega estas escalas:





Claramente, no podemos ver la escala z porque nuestra malla está frente a la cámara.

Si bien puede usar valores negativos, puede generar errores más adelante porque los ejes no estarán orientados en la dirección lógica. Trate de evitar hacerlo.

Debido a que es un `Vector3`, podemos usar todos los métodos mencionados anteriormente.

Rotar objetos

La rotación es un poco más problemática que la posición y la escala. Hay dos formas de manejar una rotación.

Evidentemente puedes usar la propiedad de rotación, sin embargo, una opción menos obvia sería usar la propiedad de cuaternión. Three.js admite ambos, y la actualización de uno actualizará automáticamente el otro. Es solo una cuestión de cuál prefieres.

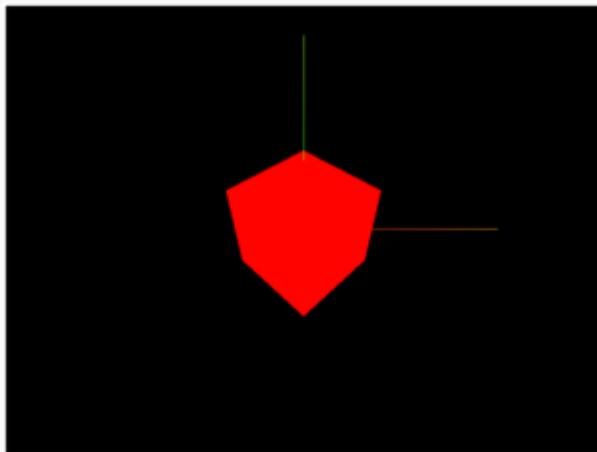
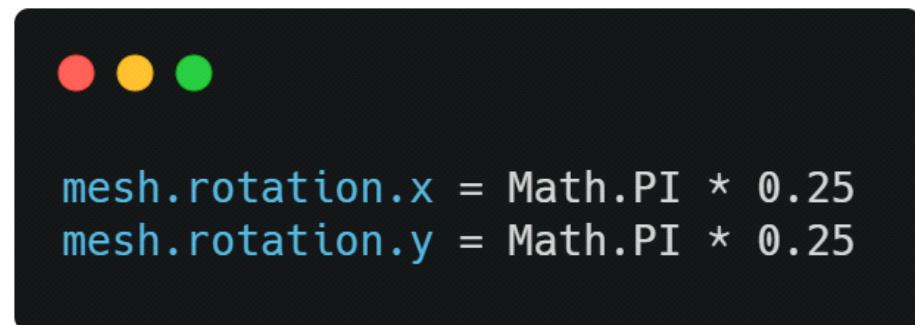
Rotación

La propiedad de rotación también tiene propiedades x, y y z, pero en lugar de un `Vector3`, es un `Euler`. Cuando cambia las propiedades x, y, z de un `Euler`, puede imaginarse poner un palo a través del centro de su objeto en la dirección del eje y luego rotar ese objeto en ese palo.

- Si gira sobre el eje y, puede imaginarlo como un carrusel.
- Si gira sobre el eje x, puede imaginar que está girando las ruedas de un automóvil en el que estarías.
- Y si gira en el eje z, puede imaginar que está girando las hélices frente a un avión en el que estaría.

El valor de estos ejes se expresa en radianes. Si quieras lograr la mitad de una rotación, tendrás que escribir algo como 3,14159 ... Probablemente reconozcas ese número como π . En JavaScript nativo, puede terminar con una aproximación de π usando Math.PI.

Comente la escala y agregue un octavo de rotación completa en los ejes x e y:



¿Es fácil? Sí, pero cuando combinás esas rotaciones, puede terminar con resultados extraños. ¿Por qué? Porque, mientras gira el eje x, también cambia la orientación de los otros ejes. La rotación se aplica en el siguiente orden: x, y y luego z. Eso puede resultar en comportamientos extraños como un bloqueo de cardán llamado cuando un eje no tiene más efecto, todo debido a los anteriores.

Podemos cambiar este orden usando el método reorder (...) object.rotation.reorder ('yxz')

Si bien Euler es más fácil de entender, este problema de pedido puede causar problemas. Y esta es la razón por la que la mayoría de los motores y software 3D utilizan otra solución llamada Quaternion.

Quaternion

La propiedad del cuaternion también expresa una rotación, pero de una forma más matemática, que resuelve el problema del orden.

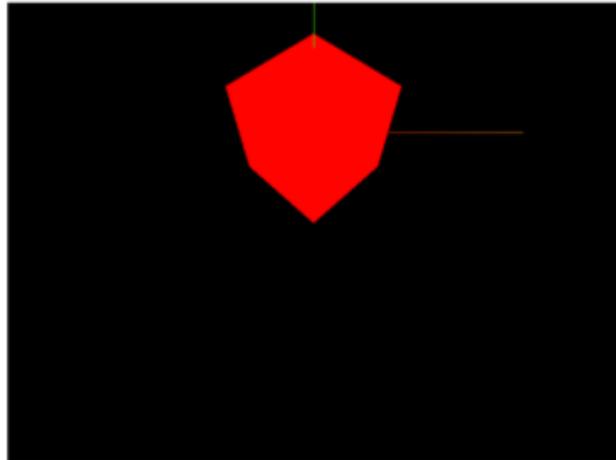
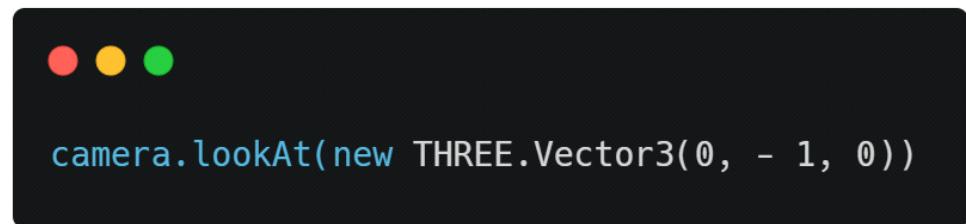
No cubriremos cómo funcionan los cuaterniones en esta lección, pero tenga en cuenta que el cuaternion se actualiza cuando cambia la rotación. Esto significa que puede utilizar cualquiera de los dos como desee.

¡Mira esto!

Las instancias de Object3D tienen un método excelente llamado lookAt (...) que le permite pedirle a un objeto que mire algo. El objeto rotará automáticamente su eje -z hacia el objetivo que proporcionaste. No se necesitan matemáticas complicadas.

Puedes usarlo para girar la cámara hacia un objeto, orientar un cañón para enfrentar a un enemigo o mover los ojos del personaje hacia un objeto.

El parámetro es el objetivo y debe ser un Vector3. Puedes crear uno solo para probarlo:



El cubo parece estar más alto, pero, de hecho, la cámara mira debajo del cubo.

También podemos usar cualquier Vector3 existente, como la posición de la malla, pero eso dará como resultado la posición predeterminada de la cámara porque nuestra malla está en el centro de la escena.



```
camera.lookAt(mesh.position)
```

Combinando transformaciones

Puede combinar la posición, la rotación (o cuaternion) y la escala en cualquier orden. El resultado será el mismo. Es equivalente al estado del objeto.

Combinemos todas las transformaciones que probamos antes:

```
mesh.position.x = 0.7  
mesh.position.y = - 0.6  
mesh.position.z = 1  
mesh.scale.x = 2  
mesh.scale.y = 0.25  
mesh.scale.z = 0.5  
mesh.rotation.x = Math.PI * 0.25  
mesh.rotation.y = Math.PI * 0.25
```

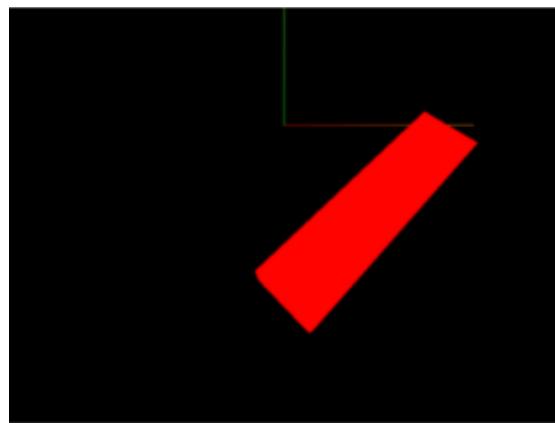


Gráfico de escena

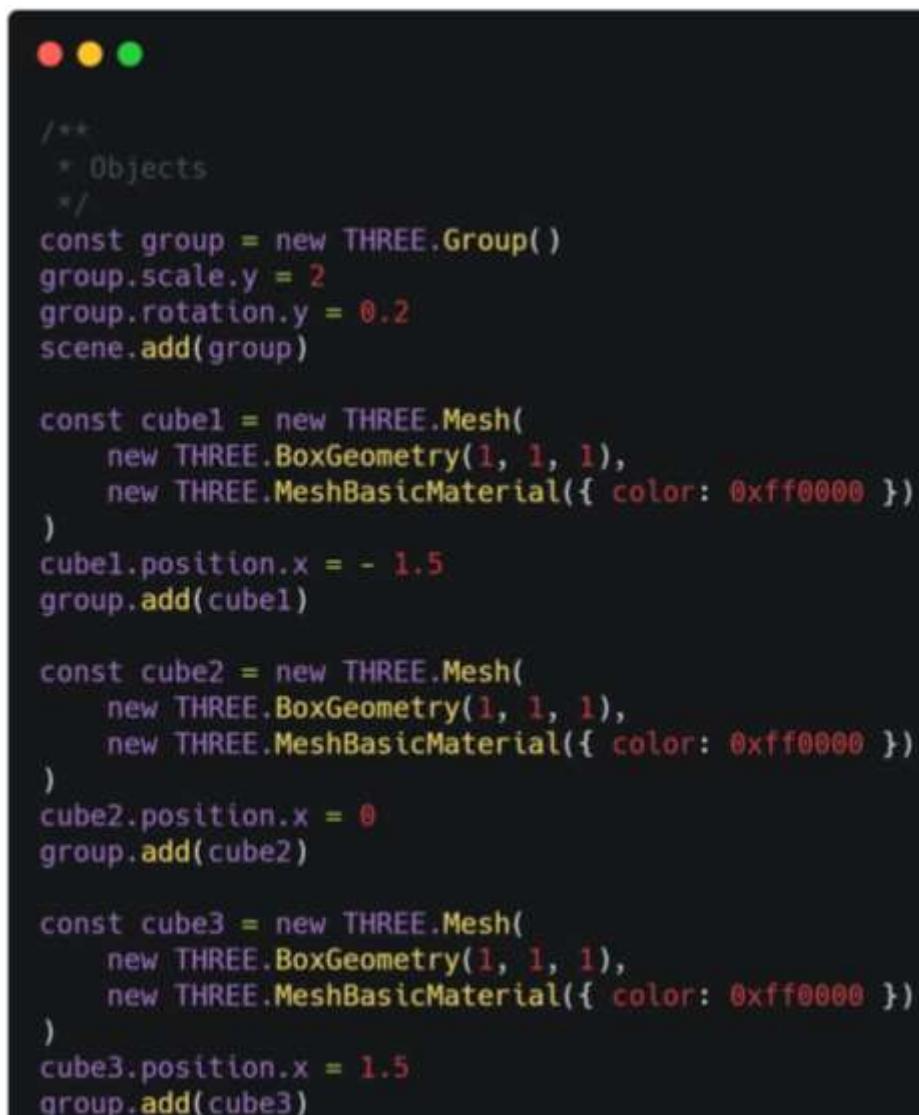
En algún momento, es posible que desees agrupar las cosas. Digamos que estás construyendo una casa con paredes, puertas, ventanas, techo, arbustos, etc. Cuando crees que has terminado, te das cuenta de que la casa es demasiado pequeña y tienes que volver a escalar cada objeto y actualizar sus posiciones.

Una buena alternativa sería agrupar todos esos objetos en un contenedor y escalar ese contenedor. Puedes hacerlo con la clase grupal.

Cree una instancia de un grupo y agréguelo a la escena. Ahora, cuando desee crear un nuevo objeto, puede agregarlo al Grupo que acaba de crear utilizando el método add (...) en lugar de agregarlo directamente a la escena.

Debido a que la clase Group hereda de la clase Object3D, tiene acceso a las propiedades y métodos mencionados anteriormente, como *position*, *scale*, *rotation*, *quaternion* y *lookAt*.

Comente la llamada *lookAt* (...) y, en lugar de nuestro cubo creado previamente, cree 3 cubos y agréguelos a un grupo. Luego aplique transformaciones en el grupo:

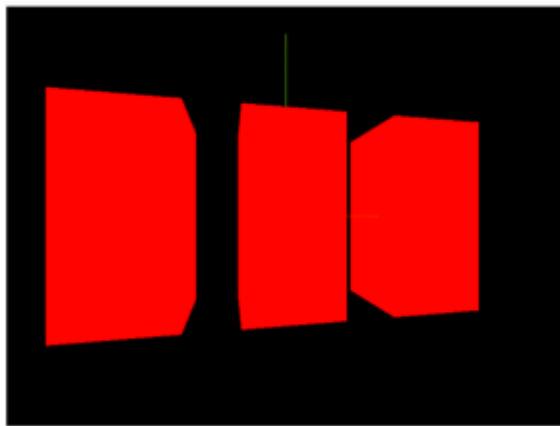


```
/* Objects */
const group = new THREE.Group()
group.scale.y = 2
group.rotation.y = 0.2
scene.add(group)

const cube1 = new THREE.Mesh(
    new THREE.BoxGeometry(1, 1, 1),
    new THREE.MeshBasicMaterial({ color: 0xffff00 })
)
cube1.position.x = - 1.5
group.add(cube1)

const cube2 = new THREE.Mesh(
    new THREE.BoxGeometry(1, 1, 1),
    new THREE.MeshBasicMaterial({ color: 0xffff00 })
)
cube2.position.x = 0
group.add(cube2)

const cube3 = new THREE.Mesh(
    new THREE.BoxGeometry(1, 1, 1),
    new THREE.MeshBasicMaterial({ color: 0xffff00 })
)
cube3.position.x = 1.5
group.add(cube3)
```



El orden realmente no importa, siempre que sea JavaScript válido.

Ahora que sabemos cómo transformar objetos, es hora de crear algunas animaciones.

6. Animaciones

Introducción

Creamos una escena que renderizamos una vez al final de nuestro código. Eso ya es un buen progreso, pero la mayoría de las veces querrás animar tus creaciones.

Las animaciones, cuando se usa Three.js, funcionan como stop motion. Mueves los objetos y haces un render. Luego mueves los objetos un poco más y haces otro render. Etc. Cuanto más mueva los objetos entre renderizados, más rápido parecerán moverse.

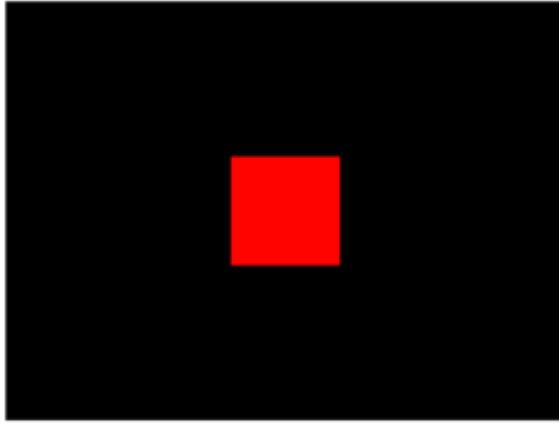
La pantalla que está viendo funciona con una frecuencia específica. A eso lo llamamos velocidad de fotogramas. La velocidad de fotogramas depende principalmente de la pantalla, pero la computadora en sí tiene limitaciones. La mayoría de las pantallas funcionan a 60 fotogramas por segundo. Si haces los cálculos, eso significa aproximadamente un fotograma cada 16 ms. Pero algunas pantallas pueden funcionar mucho más rápido, y cuando la computadora tiene dificultades para procesar cosas, funcionará más lentamente.

Queremos ejecutar una función que mueva objetos y haga el render en cada fotograma independientemente de la velocidad de fotogramas.

La forma nativa de JavaScript de hacerlo es usando el método `window.requestAnimationFrame (...)`.

Configuracion

Como lo teníamos anteriormente, tenemos nuestro cubo en el centro de nuestra escena.



Usando requestAnimationFrame

El propósito principal de `requestAnimationFrame` no es ejecutar código en cada marco. `requestAnimationFrame` ejecutará la función que proporciones en el siguiente marco. Pero luego, si esta función también usa `requestAnimationFrame` para ejecutar esa misma función en el siguiente marco, terminará con su función ejecutándose en cada marco para siempre, que es exactamente lo que queremos.

Cree una función llamada `tick` y llame a esta función una vez. En esta función, use `window.requestAnimationFrame (...)` para llamar a esta misma función en el siguiente marco:

```
/** * Animate */ const tick = () => { console.log('tick') window.requestAnimationFrame(tick) } tick()
```

Eso es. Tienes tu bucle infinito.

Como puede ver en la consola, se llama al 'tick' en cada cuadro. Si prueba este código en una computadora con una alta velocidad de fotogramas, el 'tick' aparecerá con una frecuencia más alta.

Ahora puede mover la llamada `renderer.render(...)` dentro de esa función y aumentar la rotación del cubo:



```
/** * Animate */
const tick = () =>
{
    // Update objects
    mesh.rotation.y += 0.01

    // Render
    renderer.render(scene, camera)

    // Call tick again on the next frame
    window.requestAnimationFrame(tick)
}

tick()
```

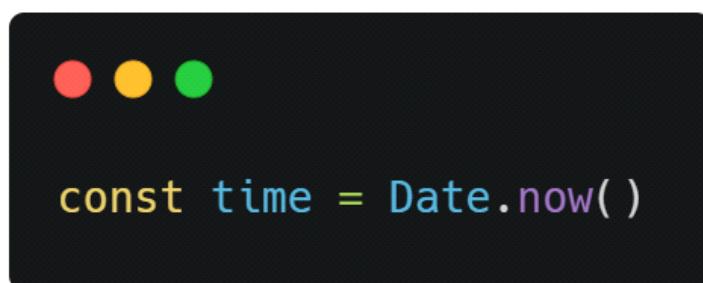
Felicitaciones, ahora tiene una animación de Three.js.

El problema es que si prueba este código en una computadora con alta velocidad de fotogramas, el cubo girará más rápido y si prueba con una velocidad de fotogramas más baja, el cubo girará más lento.

Adaptación a la velocidad de fotogramas

Para adaptar la animación a la velocidad de fotogramas, necesitamos saber cuánto tiempo ha pasado desde el último tick.

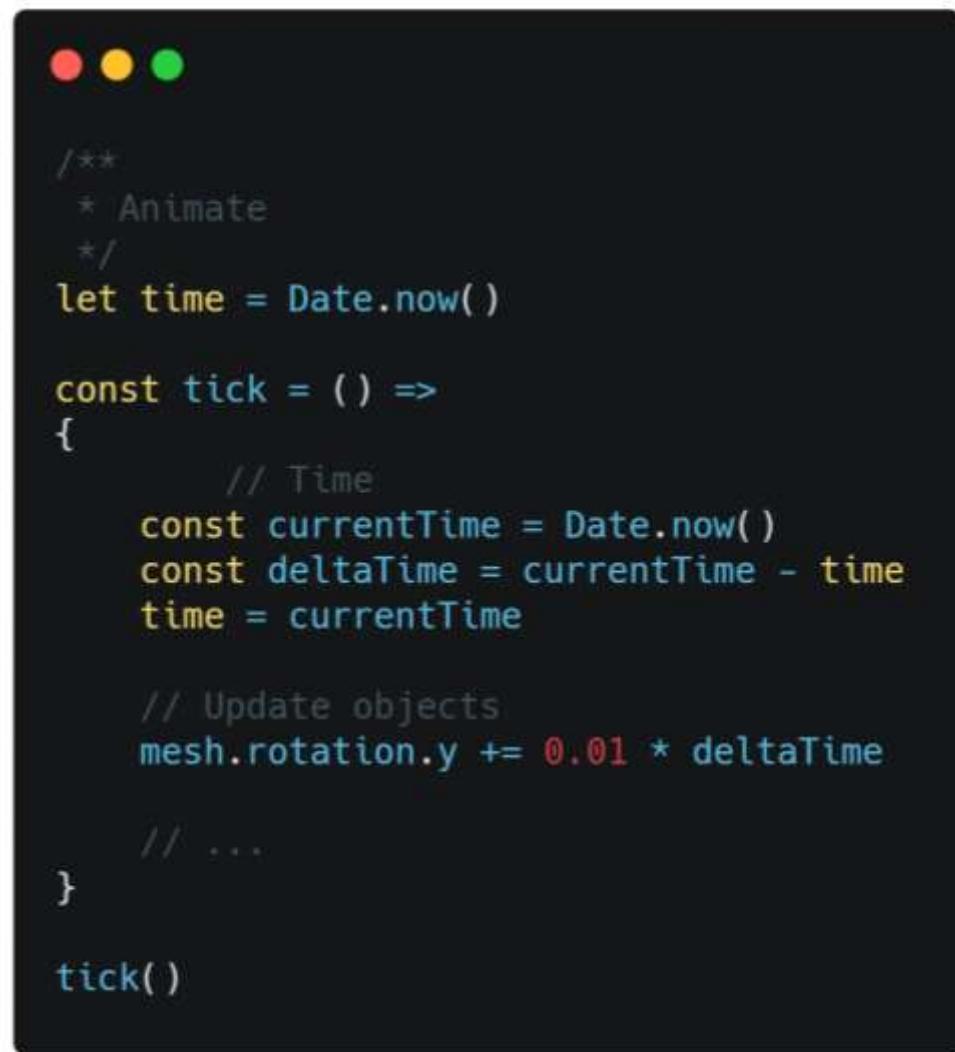
Primero, necesitamos una forma de medir el tiempo. En JavaScript nativo, puede usar `Date.now()` para obtener la marca de tiempo actual:



```
const time = Date.now()
```

La marca de tiempo corresponde a cuánto tiempo ha pasado desde el 1 de enero de 1970 (el comienzo del tiempo para Unix). En JavaScript, su unidad está en milisegundos.

Lo que necesita ahora es restar la marca de tiempo actual a la del marco anterior para obtener lo que podemos llamar deltaTime y usar este valor al animar objetos:



```
/*  
 * Animate  
 */  
let time = Date.now()  
  
const tick = () =>  
{  
    // Time  
    const currentTime = Date.now()  
    const deltaTime = currentTime - time  
    time = currentTime  
  
    // Update objects  
    mesh.rotation.y += 0.01 * deltaTime  
  
    // ...  
}  
  
tick()
```

El cubo debería girar más rápido porque el deltaTime debería ser de alrededor de 16 si su pantalla funciona a 60 fps, así que no dude en reducirlo multiplicando el valor.

Ahora que basamos nuestra rotación en la cantidad de tiempo transcurrida desde el último fotograma, esta velocidad de rotación será la misma en todas las pantallas y en todas las computadoras, independientemente de la velocidad de fotogramas.

Usando el reloj

Si bien este código no es tan complicado, hay una solución incorporada en Three.js llamada Clock que manejará los cálculos de tiempo.

Simplemente tiene que crear una instancia de una variable Clock y usar los métodos integrados como getElapsedTime (). Este método devolverá cuántos segundos han pasado desde que se creó el Reloj.

Puede utilizar este valor para rotar el objeto:

```
/**  
 * Animate  
 */  
const clock = new THREE.Clock()  
  
const tick = () =>  
{  
    const elapsedTime = clock.getElapsedTime()  
  
    // Update objects  
    mesh.rotation.y = elapsedTime  
  
    // ...  
}  
  
tick()
```

También puede usarlo para mover cosas con la propiedad de posición. Si lo combina con Math.sin (...) puede obtener un resultado bastante bueno:

```
/**  
 * Animate  
 */  
const clock = new THREE.Clock()  
  
const tick = () =>  
{  
    const elapsedTime = clock.getElapsedTime()  
  
    // Update objects  
    mesh.position.x = Math.cos(elapsedTime)  
    mesh.position.y = Math.sin(elapsedTime)  
  
    // ...  
}  
  
tick()
```

Y obviamente, puedes usar esas técnicas para animar cualquier Object3D como la cámara:

```
/**  
 * Animate  
 */  
const clock = new THREE.Clock()  
  
const tick = () =>  
{  
    const elapsedTime = clock.getElapsedTime()  
  
    // Update objects  
    camera.position.x = Math.cos(elapsedTime)  
    camera.position.y = Math.sin(elapsedTime)  
    camera.lookAt(mesh.position)  
  
    // ...  
}  
  
tick()
```

Otro método disponible es getDelta (...), pero no debe usarlo a menos que sepa exactamente lo que está sucediendo en el código de la clase Clock. Usarlo puede alterar tu animación y obtendrás resultados no deseados.

Usando una biblioteca

A veces querrá animar su escena de una manera muy específica que requerirá el uso de otra biblioteca. Hay toneladas de bibliotecas de animación, pero una muy famosa es GSAP.

Para agregar GSAP a nuestro proyecto Webpack, podemos usar el administrador de dependencias provisto con Node.js llamado npm.

En su terminal (mientras el servidor no se está ejecutando o usando otra ventana de terminal en la misma carpeta), ejecute `npm install --save gsap@3.5.1`

El argumento `--save` guarda la dependencia en `package.json` para que el módulo se pueda recuperar si hacemos una instalación npm.

El `@ 3.5.1` fuerza la versión. Usamos esta versión porque fue la que se usó al escribir la lección, pero puede probar la última versión si lo desea eliminando `@ 3.5.1`.

GSAP ahora está disponible en la carpeta `node_modules /`, y podemos importarlo en nuestro `script.js`:

```
import './style.css'  
import * as THREE from 'three'  
import gsap from 'gsap'  
  
// ...
```

Hay muchas formas de utilizar GSAP y podríamos dedicarle un curso completo, pero no es el objetivo de este curso. Simplemente crearemos una interpolación para probar las cosas. Si ya sabe cómo usar GSAP, funciona igual con Three.js.

Comenta el código relacionado con las animaciones anteriores pero mantén la función tick con el render. Luego puede crear lo que llamamos una interpolación (una animación de A B) usando gsap.to(...):

```
/**  
 * Animate  
 */  
gsap.to(mesh.position, { duration: 1, delay: 1, x: 2 })  
  
const tick = () =>  
{  
    // Render  
    renderer.render(scene, camera)  
  
    // Call tick again on the next frame  
    window.requestAnimationFrame(tick)  
}  
  
tick()
```

GSAP tiene un requestAnimationFrame incorporado, por lo que no necesita actualizar la animación usted mismo, pero aún así, si desea ver el cubo en movimiento, debe seguir haciendo los renderizados de su escena en cada cuadro.

Elegir la solución adecuada

En cuanto a elegir entre JS nativo y una biblioteca de animación, es una cuestión de lo que desea lograr. Si va a crear un carrusel que gira para siempre, no necesita ninguna biblioteca para eso. Pero si desea animar, por ejemplo, el movimiento de una espada, es posible que prefiera utilizar una biblioteca.

7. Camaras

Introducción

Ya creamos una PerspectiveCamera, pero existen otros tipos de cámaras, como puedes ver en la documentación.

Cámara

La clase Camera es lo que llamamos una clase abstracta. Se supone que no debe usarlo directamente, pero puede heredar de la clase para tener acceso a propiedades y métodos en común. Algunas de las siguientes clases heredan de la clase Camera.

ArrayCamera

ArrayCamera se utiliza para renderizar su escena varias veces mediante el uso de varias cámaras. Cada cámara renderizará un área específica del lienzo. Puedes imaginar que esto parece un juego multijugador de consola de la vieja escuela en el que teníamos que compartir una pantalla dividida.

StereoCamera

StereoCamera se usa para renderizar la escena a través de dos cámaras que imitan los ojos para crear lo que llamamos un efecto de paralax que atraerá a su cerebro a pensar que hay profundidad. Debe tener el equipo adecuado como un casco de realidad virtual o gafas rojas y azules para ver el resultado.

CubeCamera

La CubeCamera se utiliza para obtener un render en cada dirección (hacia adelante, hacia atrás, hacia la izquierda, hacia la derecha, hacia arriba y hacia abajo) para crear un render del entorno. Puede utilizarlo para crear un mapa de entorno para la reflexión o un mapa de sombras. Hablaremos de eso más tarde.

OrthographicCamera

OrthographicCamera se utiliza para crear representaciones ortográficas de su escena sin perspectiva. Es útil si creas un juego de estrategia en tiempo real como Age of Empire. Los elementos tendrán el mismo tamaño en la pantalla independientemente de su distancia a la cámara.

PerspectiveCamera

La PerspectiveCamera es la que ya usamos y simulamos una cámara de la vida real con perspectiva. Nos centraremos en OrthographicCamera y PerspectiveCamera.

PerspectiveCamera

Como vimos anteriormente, la clase PerspectiveCamera necesita algunos parámetros para ser instanciados, pero no usamos todos los parámetros posibles. Agregue el tercer y cuarto parámetro:

```
const camera = new THREE.PerspectiveCamera(75, sizes.width / sizes.height, 1, 100)
```

Debería obtener el mismo resultado, pero hablemos de esos parámetros en detalle.

Campo de visión

El primer parámetro llamado campo de visión corresponde al ángulo de amplitud vertical de la vista de la cámara en grados. Si usa un ángulo pequeño, terminará con un efecto de alcance largo, y si usa un gran angular, terminará con un efecto de ojo de pez porque, al final, lo que ve la cámara se estirará o exprimido para que se ajuste al lienzo.

En cuanto a elegir el campo de visión correcto, tendrás que probar cosas. Normalmente uso un campo de visión entre 45 y 75.

Relación de aspecto

El segundo parámetro se llama relación de aspecto y corresponde al ancho dividido por la altura. Si bien puede pensar que obviamente es el ancho del lienzo por la altura del lienzo y Three.js debería calcularlo por sí mismo, no siempre es el caso si comienza a usar Three.js de maneras muy específicas. Pero en nuestro caso, simplemente puede usar el ancho del lienzo y la altura del lienzo.

Recomiendo guardar esos valores en un objeto porque los vamos a necesitar varias veces:



```
const sizes = {  
    width: 800,  
    height: 600  
}
```

Cerca y lejos

Los parámetros tercero y cuarto llamados cerca y lejos, corresponden a qué tan cerca y qué tan lejos puede ver la cámara. Cualquier objeto o parte del objeto más cercano a la cámara que el valor cercano o más alejado de la cámara que el valor lejano no se mostrará en el render.

Puedes ver eso como en esos viejos juegos de carreras donde podías ver los árboles aparecer en la distancia.

Si bien puede tener la tentación de usar valores muy pequeños y muy grandes como 0.0001 y 9999999, puede terminar con un error llamado z-fighting en el que dos caras parecen pelear por cuál se representará por encima de la otra.

<https://twitter.com/FreyaHolmer/status/799602767081848832>

https://twitter.com/Snapman_I_Am/status/800567120765616128

Intente utilizar valores razonables y aumente esos valores solo si los necesita. En nuestro caso, podemos usar 0.1 y 100.

Cámara ortográfica

Si bien no usaremos este tipo de cámara durante el resto del curso, puede ser útil para proyectos específicos.

La OrthographicCamera se diferencia de la PerspectiveCamera por su falta de perspectiva, lo que significa que los objetos tendrán el mismo tamaño independientemente de su distancia a la cámara.

Los parámetros que debe proporcionar son muy diferentes de los de PerspectiveCamera.

En lugar de un campo de visión, debe proporcionar qué tan lejos puede ver la cámara en cada dirección (izquierda, derecha, arriba y abajo). Luego, puede proporcionar los valores cercanos y lejanos tal como lo hicimos para PerspectiveCamera.

Comente la PerspectiveCamera y agregue OrthographicCamera. Mantenga la actualización de la posición y llame a lookAt (...):

```
const camera = new THREE.OrthographicCamera(- 1, 1, 1, - 1, 0.1, 100)
```

Como puede ver, no hay perspectiva y los lados de nuestro cubo parecen paralelos. El problema es que nuestro cubo no parece cúbico.

Eso se debe a los valores que proporcionamos para los lados izquierdo, derecho, superior e inferior que son 1 o - 1, lo que significa que representamos un área cuadrada, pero esa área cuadrada se estirará para ajustarse a nuestro lienzo rectangular y nuestro lienzo no es 't un cuadrado.

Necesitamos usar la proporción del lienzo (ancho por alto). Creamos una variable denominada AspectRatio (al igual que PerspectiveCamera) y almacenemos esa proporción en ella:

```
const aspectRatio = sizes.width / sizes.height  
const camera = new THREE.OrthographicCamera(- 1 * aspectRatio, 1 * aspectRatio, 1, - 1, 0.1, 100)
```

Esto da como resultado un ancho del área de renderizado mayor que la altura del área de renderizado porque el ancho de nuestro lienzo es mayor que su altura.

Ahora tenemos un cubo que parece un cubo.

Controles personalizados

Volvamos a nuestra PerspectiveCamera. Comente la OrthographicCamera, descomente la PerspectiveCamera, mueva la cámara para que mire hacia el cubo y elimine la rotación de la malla en la función de marca:

```
// Camera
const camera = new THREE.PerspectiveCamera(75, sizes.width / sizes.height, 1, 1000)
// const aspectRatio = sizes.width / sizes.height
// const camera = new THREE.OrthographicCamera(-1 * aspectRatio, 1 * aspectRatio, 1, -1, 0.1, 1000)

// camera.position.x = 2
// camera.position.y = 2
camera.position.z = 3
camera.lookAt(mesh.position)
scene.add(camera)
```

Lo que queremos hacer ahora es controlar la cámara con nuestro mouse. En primer lugar, queremos saber las coordenadas del mouse. Podemos hacer eso usando JavaScript nativo escuchando el evento `mousemove` con `addEventListener`:

```
// Cursor
window.addEventListener('mousemove', (event) =>
{
    console.log(event.clientX, event.clientY)
})
```

Podríamos usar esos valores, pero recomiendo ajustarlos. Al ajustar, me refiero a tener una amplitud de 1 y que el valor puede ser tanto negativo como positivo.

Si solo nos enfocamos en el valor x, eso significaría que:

- si su cursor está en el extremo izquierdo del lienzo, debería obtener - 0.5
- si su cursor está en el centro del lienzo, debería obtener 0
- si su cursor está en el extremo derecho del lienzo, debería obtener 0.5
- Si bien esto no es obligatorio, ayuda tener valores limpios como ese.

Al igual que la variable de tamaño, crearemos una variable de cursor con propiedades x y y predeterminadas y luego actualizaremos esas propiedades en la devolución de llamada `mousemove`:

```
// Cursor
const cursor = {
  x: 0,
  y: 0
}

window.addEventListener('mousemove', (event) =>
{
  cursor.x = event.clientX / sizes.width - 0.5
  cursor.y = event.clientY / sizes.height - 0.5

  console.log(cursor.x, cursor.y)
})
```

Dividir `event.clientX` por `sizes.width` nos dará un valor entre 0 y 1 (si mantenemos el cursor sobre el lienzo) mientras que restar 0.5 le dará un valor entre - 0.5 y 0.5.

Ahora tiene la posición del mouse almacenada en la variable de objeto del cursor y puede actualizar la posición de la cámara en la función de marca:

```
const tick = () =>
{
  // ...

  // Update camera
  camera.position.x = cursor.x
  camera.position.y = cursor.y

  // ...
}
```

Como puede ver, está funcionando, pero los movimientos de los ejes parecen algo incorrectos. Esto se debe a que el eje `position.y` es positivo cuando va hacia arriba en Three.js, pero el eje `clientY` es positivo cuando va hacia abajo en la página web.

Simplemente puede invertir el cursor.y mientras lo actualiza agregando un - delante de toda la fórmula (no olvide los paréntesis):

```
window.addEventListener('mousemove', (event) =>
{
    cursor.x = event.clientX / sizes.width - 0.5
    cursor.y = - (event.clientY / sizes.height - 0.5)
})
```

Finalmente, puede aumentar la amplitud multiplicando cursor.xy cursor.y y pedirle a la cámara que mire la malla usando el método lookAt (...):

```
const tick = () =>
{
    // ...

    // Update camera
    camera.position.x = cursor.x * 5
    camera.position.y = cursor.y * 5
    camera.lookAt(mesh.position)

    // ...
}
```

Podemos ir aún más lejos haciendo una rotación completa de la cámara alrededor de la malla usando Math.sin (...) y Math.cos (...).

sin y cos, cuando se combinan y se usan con el mismo ángulo, nos permiten colocar las cosas en un círculo. Para hacer una rotación completa, ese ángulo debe tener una amplitud de 2 veces π (llamado "pi"). Para que lo sepas, una rotación completa se llama "tau" pero no tenemos acceso a este valor en JavaScript y tenemos que usar π en su lugar.

Puede acceder a una aproximación de π en JavaScript nativo utilizando Math.PI.

Para aumentar el radio de ese círculo, simplemente puede multiplicar el resultado de Math.sin (...) y Math.cos (...):

```
const tick = () =>
{
    // ...

    // Update camera
    camera.position.x = Math.sin(cursor.x * Math.PI * 2) * 2
    camera.position.z = Math.cos(cursor.x * Math.PI * 2) * 2
    camera.position.y = cursor.y * 3
    camera.lookAt(mesh.position)

    // ...
}

tick()
```

Si bien este es un buen comienzo para controlar la cámara, Three.js ha integrado varias clases llamadas **controles** para ayudarlo a hacer lo mismo y mucho más.

Controles incorporados

Si escribe "controles" en la documentación de Three.js, verá que hay muchos controles prefabricados. Solo usaremos uno de ellos durante el resto del curso, pero puede ser interesante conocer su función.

DeviceOrientationControls

DeviceOrientationControls recuperará automáticamente la orientación del dispositivo si su dispositivo, sistema operativo y navegador lo permiten y rotará la cámara en consecuencia. Puede usarlo para crear universos inmersivos o experiencias de realidad virtual si tiene el equipo adecuado.

FlyControls

FlyControls permite mover la cámara como si estuviera en una nave espacial. Puede rotar en los 3 ejes, avanzar y retroceder.

FirstPersonControls

FirstPersonControls es como FlyControls, pero con un eje ascendente fijo. Puede ver eso como una vista de pájaro volando donde el pájaro no puede hacer un barril. Si bien FirstPersonControls contiene "FirstPerson", no funciona como en los juegos FPS.

PointerLockControls

PointerLockControls utiliza la API de JavaScript de bloqueo de puntero. Esta API oculta el cursor, lo mantiene centrado y sigue enviando los movimientos en la devolución de llamada del evento mousemove. Con esta API, puede crear juegos FPS directamente dentro del navegador. Si bien esta clase suena muy prometedora si desea crear ese tipo de interacción, solo manejará la rotación de la cámara cuando el puntero esté bloqueado. Tendrás que manejar la posición de la cámara y la física del juego tú mismo.

OrbitControls

OrbitControls es muy similar a los controles que hicimos en la lección anterior. Puede rotar alrededor de un punto con el mouse izquierdo, trasladar lateralmente con el mouse derecho y acercar o alejar la imagen con la rueda.

TrackballControls

TrackballControls es como OrbitControls pero no hay límites en términos de ángulo vertical. Puede seguir girando y hacer giros con la cámara incluso si la escena se pone al revés.

TransformControls

TransformControls no tiene nada que ver con la cámara. Puede usarlo para agregar un gizmo a un objeto para mover ese objeto.

DragControls

Al igual que TransformControls, DragControls no tiene nada que ver con la cámara. Puede usarlo para mover objetos en un plano frente a la cámara arrastrándolos y soltándolos. Solo usaremos OrbitControls, pero siéntase libre de probar las otras clases.

OrbitControls

Comentemos la parte donde actualizamos la cámara en la función tick.

Instanciar

Primero, necesitamos crear una instancia de una variable usando la clase OrbitControls. Si bien puede pensar que puede usar THREE.OrbitControls aquí, desafortunadamente está equivocado.

La clase OrbitControls es parte de aquellas clases que no están disponibles por defecto en la variable THREE. Esa decisión ayuda a reducir el peso de la biblioteca. Y aquí es donde entra nuestra plantilla Webpack.

Es posible que la clase OrbitControls no esté disponible en la variable TRES; todavía se encuentra en la carpeta de dependencias. Para importarlo, debe proporcionar la ruta desde dentro de la carpeta / node_modules /, que es /three/examples/jsm/controls/OrbitControls.js:

```
importar {OrbitControls} de 'three / examples / jsm / controls / OrbitControls.js'
```

JavaScript

Ahora puede crear una instancia de una variable usando la clase OrbitControls (sin los TRES.) Y asegúrese de hacerlo después de crear la cámara.

Para que funcione, debe proporcionar la cámara y el elemento en la página que manejará los eventos del mouse como parámetros:

```
// Controls
const controls = new OrbitControls(camera, canvas)
```

Ahora puede arrastrar y soltar con el mouse izquierdo o el derecho para mover la cámara, y puede desplazarse hacia arriba o hacia abajo para acercar o alejar.

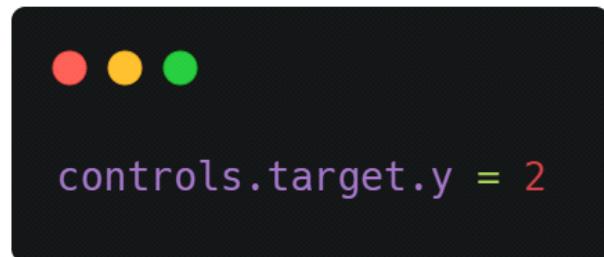
Es mucho más fácil que nuestro código personalizado y viene con más controles. Pero vayamos un poco más lejos.

Objetivo

De forma predeterminada, la cámara está mirando al centro de la escena. Podemos cambiar eso con la propiedad objetivo.

Esta propiedad es un Vector3, lo que significa que podemos cambiar sus propiedades x, y z.

Si queremos que OrbitControls mire por encima del cubo de forma predeterminada, solo tenemos que aumentar la propiedad y:



Esto no es muy útil en nuestro caso así que comentemos esta parte.

Mojadura

Si lee la documentación de OrbitControls hay menciones de amortiguación. La amortiguación suavizará la animación agregando algún tipo de fórmulas de aceleración y fricción. Para habilitar la amortiguación, cambie la propiedad enableDamping de los controles a true.

Para que funcionen correctamente, los controles también deben actualizarse en cada marco llamando a controls.update(). Puede hacer eso en la función de tick:



Verá que los controles ahora son mucho más suaves.

Puede usar muchos otros métodos y propiedades para personalizar sus controles, como la velocidad de rotación, la velocidad del zoom, el límite del zoom, el límite del ángulo, la fuerza de amortiguación y las combinaciones de teclas (porque sí, también puede usar su teclado).

Cuando usar los controles integrados

Si bien esos controles son útiles, tienen limitaciones. Si confía demasiado en ellos, podría terminar teniendo que cambiar el funcionamiento de la clase de una manera inesperada.

Primero, asegúrese de enumerar todas las características que necesita de esos controles, luego verifique si la clase que está a punto de usar puede manejar todas esas características.

Si no es así, tendrá que hacerlo por su cuenta.

8. Pantalla completa y redimensionamiento

Introducción

Nuestro lienzo tiene actualmente una resolución fija de 800x600. No es necesario que su WebGL se adapte a toda la pantalla, pero si desea una experiencia inmersiva, podría ser mejor.

Primero, nos gustaría que el lienzo ocupara todo el espacio disponible. Luego, debemos asegurarnos de que aún se ajuste si el usuario cambia el tamaño de su ventana. Finalmente, debemos brindarle al usuario una forma de experimentar con la experiencia en pantalla completa.

Configuración

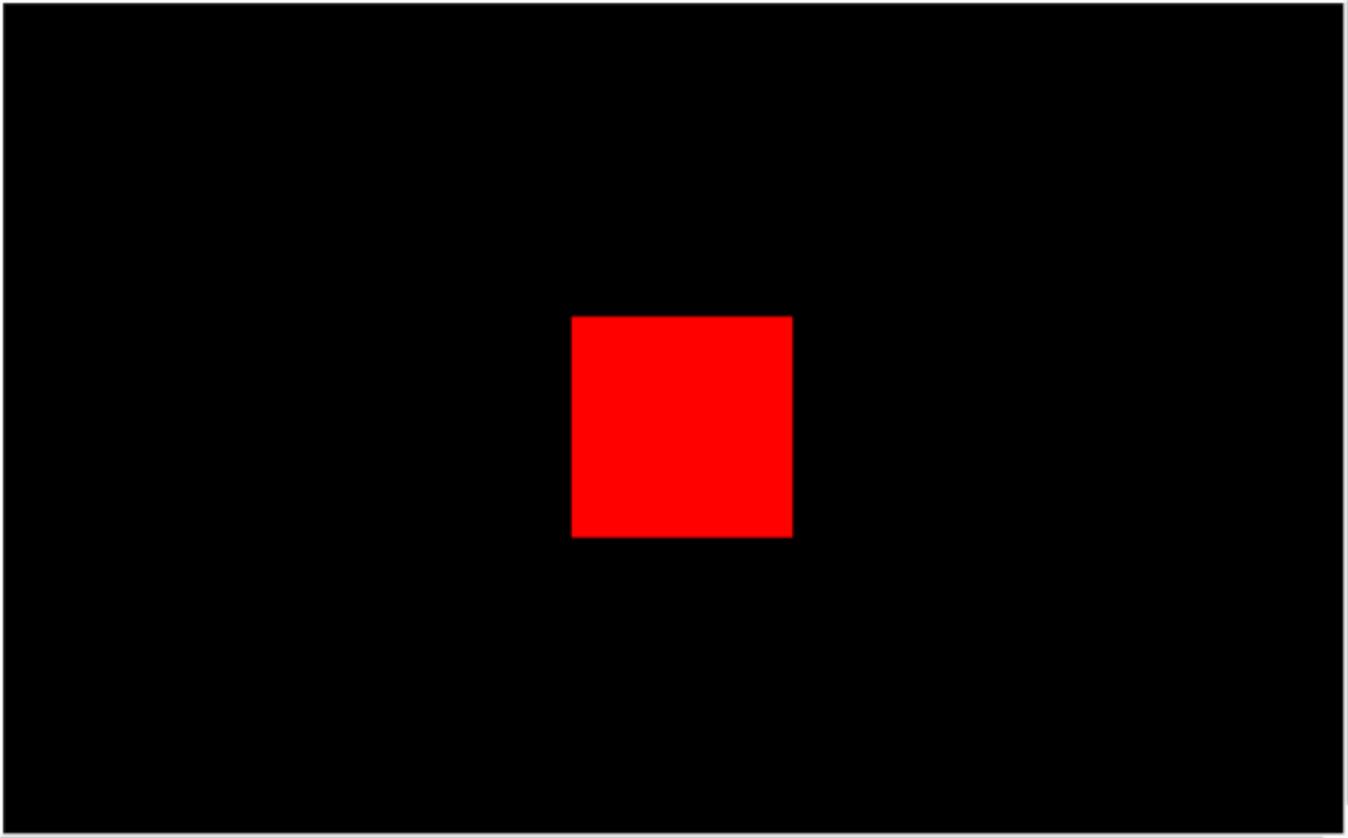
El iniciador contiene lo que terminamos en la lección anterior. Tenemos nuestro cubo en el centro, y podemos arrastrar y soltar para mover la cámara.

Encajar en la ventana gráfica

Para que el lienzo encaje perfectamente en la ventana gráfica, en lugar de usar números fijos en la variable de tamaños, use `window.innerWidth` y `window.innerHeight`:



```
// ...
// Sizes
const sizes = {
  width: window.innerWidth,
  height: window.innerHeight
}
// ...
```



Puede ver que el lienzo ahora tiene el ancho y el alto de la ventana gráfica. Desafortunadamente, hay un margen blanco y una barra de desplazamiento (intente desplazarse si no ve ninguna barra de desplazamiento).

El problema es que todos los navegadores tienen estilos predeterminados como títulos más significativos, enlaces subrayados, espacio entre párrafos y rellenos en la página. Hay muchas formas de solucionarlo, y puede depender del resto de su sitio web. Si tiene otro contenido, intente no romper ninguno de ellos mientras hace esto.

Mantendremos las cosas simples y arreglaremos la posición del lienzo usando CSS.

Nuestra plantilla ya está vinculada a un archivo `src/style.css`.

```
<link rel="stylesheet" href=".style.css">
```

Puede escribir CSS estándar como está acostumbrado y la página se recargará automáticamente.

Una buena cosa que puede hacer primero sería eliminar cualquier tipo de margen o relleno en todos los elementos usando un comodín *:

```
*  
{  
    margin: 0;  
    padding: 0;  
}
```

Podemos arreglar el lienzo en la parte superior izquierda usando su clase webgl para seleccionarlo:

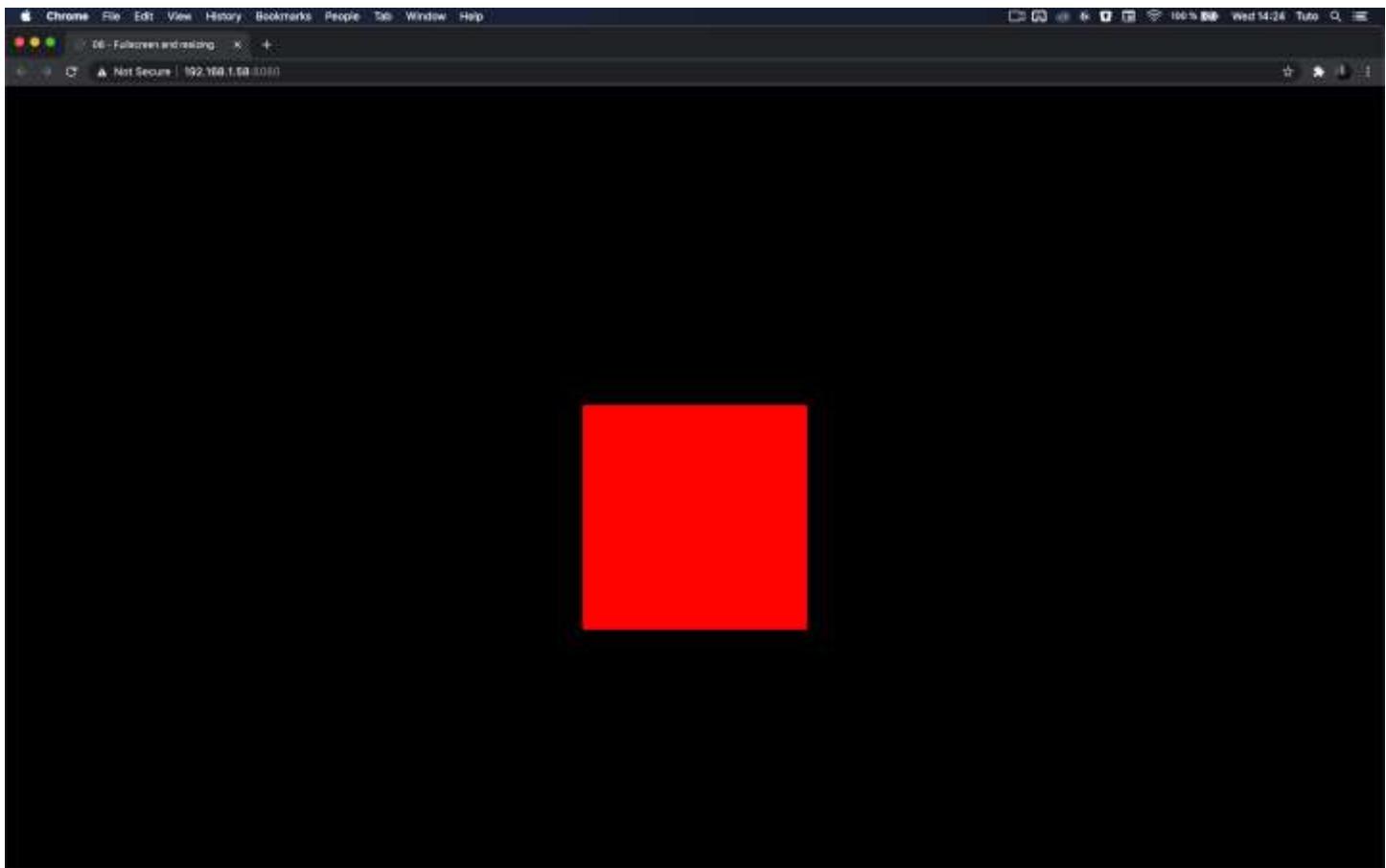
```
.webgl  
{  
    position: fixed;  
    top: 0;  
    left: 0;  
}
```

No necesita especificar el ancho o alto en el lienzo porque Three.js ya se está ocupando de eso cuando llama al método renderer.setSize (...). Esta es una buena oportunidad para solucionar un pequeño problema en nuestro lienzo. Tal vez hayas notado un contorno azul al arrastrar y soltar. Esto ocurre principalmente en las últimas versiones de Chrome. Para solucionarlo, simplemente podemos agregar un esquema: ninguno; en el .webgl:

```
.webgl  
{  
    position: fixed;  
    top: 0;  
    left: 0;  
    outline: none;  
}
```

Si desea eliminar cualquier tipo de desplazamiento incluso en pantallas táctiles, puede agregar un desbordamiento: oculto tanto en html como en el cuerpo:

```
html,  
body  
{  
    overflow: hidden;  
}
```



Ahora puede disfrutar de su WebGL en todo su esplendor. Desafortunadamente, si cambia el tamaño de la ventana, el lienzo no lo seguirá.

Tenemos que lidiar con el cambio de tamaño.

Manejar cambio de tamaño

Para cambiar el tamaño del lienzo, primero debemos saber cuándo se está cambiando el tamaño de la ventana. Para hacerlo, puede escuchar el evento de cambio de tamaño en la ventana.

Agregue el oyente de cambio de tamaño justo después de la variable de tamaños:

```
window.addEventListener('resize', () =>
{
    console.log('window has been resized')
})
```

Ahora que activamos una función cuando se cambia el tamaño de la ventana, necesitamos actualizar algunas cosas en nuestro código.

Primero, debemos actualizar la variable de tamaños:

```
window.addEventListener('resize', () =>
{
    // Update sizes
    sizes.width = window.innerWidth
    sizes.height = window.innerHeight
})
```

En segundo lugar, debemos actualizar la relación de aspecto de la cámara cambiando su propiedad de aspecto:

```
window.addEventListener('resize', () =>
{
    // ...
    // Update camera
    camera.aspect = sizes.width / sizes.height
})
```

Cuando cambia las propiedades de la cámara, como el aspecto, también necesita actualizar la matriz de proyección usando `camera.updateProjectionMatrix()`. Hablaremos de matrices más adelante:

```
window.addEventListener('resize', () =>
{
    // ...
    camera.updateProjectionMatrix()
})
```

Finalmente, debemos actualizar el renderizador. La actualización del renderizador actualizará automáticamente el ancho y la altura del lienzo:

```
window.addEventListener('resize', () =>
{
    // ...
    // Update renderer
    renderer.setSize(sizes.width, sizes.height)
})
```

Todo junto

```
window.addEventListener('resize', () =>
{
    // Update sizes
    sizes.width = window.innerWidth
    sizes.height = window.innerHeight

    // Update camera
    camera.aspect = sizes.width / sizes.height
    camera.updateProjectionMatrix()

    // Update renderer
    renderer.setSize(sizes.width, sizes.height)
})
```

Puede cambiar el tamaño de la ventana como desee, el lienzo debe cubrir la ventana gráfica sin ninguna barra de desplazamiento ni desbordamiento.

Manejar proporción de píxeles

Algunos de ustedes pueden ver una especie de renderizado borroso y artefactos con forma de escaleras en los bordes (llamado aliasing), pero no todos ustedes. Si lo hace, es porque está probando en una pantalla con una proporción de píxeles superior a 1.

La proporción de píxeles corresponde a la cantidad de píxeles físicos que tiene en la pantalla para una unidad de pixel en la parte del software.

Algo de historia

Hace unos años, todas las pantallas tenían una proporción de píxeles de 1 y todo estaba bien. Pero cuando miraba de cerca su pantalla, podía ver esos píxeles, y era una limitación para la precisión de las imágenes y la delgadez de las fuentes.

La empresa que más hizo al respecto fue Apple. Apple vio una oportunidad y comenzó a construir pantallas con una proporción de píxeles de 2 llamadas retina. Ahora, muchos constructores lo están haciendo y puedes ver pantallas con proporciones de píxeles aún más altas.

Si bien esto es bueno para la calidad de la imagen, una proporción de píxeles de 2 significa 4 veces más píxeles para renderizar. Y una proporción de píxeles de 3 significa 9 veces más píxeles para renderizar.

¿Y adivina qué? Las proporciones de píxeles más altas generalmente se encuentran en los dispositivos más débiles: los móviles.

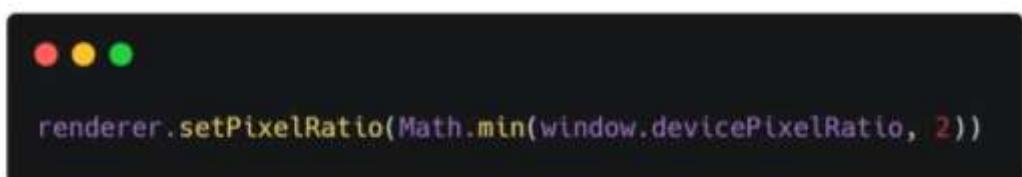
Ahí va tu velocidad de fotogramas.

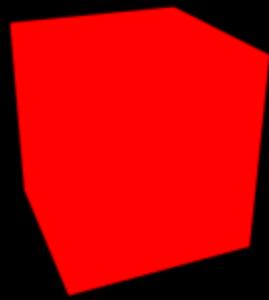
Manejar la proporción de píxeles

Para obtener la proporción de píxeles de la pantalla, puede usar `window.devicePixelRatio`, y para actualizar la proporción de píxeles de su renderizador, simplemente necesita llamar a `renderer.setPixelRatio (...)`

Es posible que tenga la tentación de enviar simplemente la proporción de píxeles del dispositivo a ese método, pero terminará con problemas de rendimiento en dispositivos con una proporción de píxeles alta.

Tener una proporción de píxeles superior a 2 es principalmente marketing. Sus ojos no verán casi ninguna diferencia entre 2 y 3, pero creará problemas de rendimiento y agotará la batería más rápido. Lo que puede hacer es limitar la proporción de píxeles a 2. Para hacerlo, puede usar `Math.min ()`:





Existen técnicas para recibir notificaciones cuando cambia la proporción de píxeles, pero solo se trata de usuarios que tienen varias pantallas con diferentes proporciones de píxeles y, por lo general, cambian el tamaño de la ventana cuando cambian de una pantalla a otra. Es por eso que simplemente agregaremos este método a la devolución de llamada de cambio de tamaño también:

```
window.addEventListener('resize', () =>
{
    // Update sizes
    sizes.width = window.innerWidth
    sizes.height = window.innerHeight

    // Update camera
    camera.aspect = sizes.width / sizes.height

    // Update renderer
    renderer.setSize(sizes.width, sizes.height)
    renderer.setPixelRatio(Math.min(window.devicePixelRatio, 2))
})
```

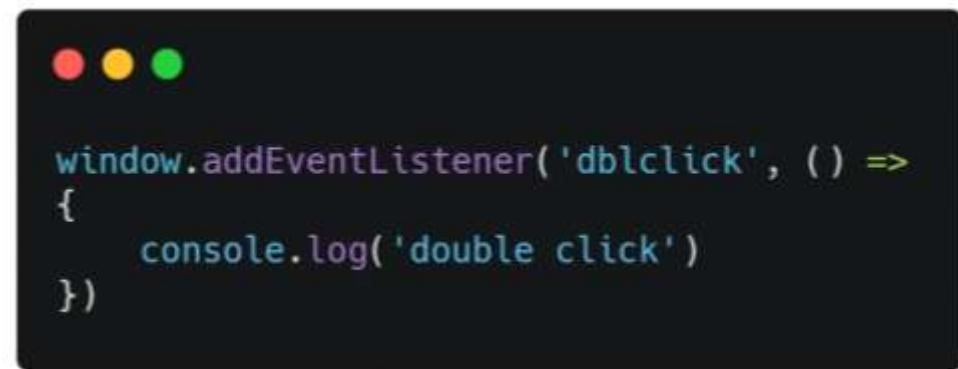
Manejar pantalla completa

Ahora que tenemos el lienzo ocupando todo el espacio disponible con la proporción de píxeles correcta, es hora de agregar soporte a la pantalla completa.

Primero, debemos decidir qué acción activará el modo de pantalla completa. Podría ser un botón HTML, pero en su lugar usaremos un doble clic.

Cuando ocurre el doble clic, cambiaremos la pantalla completa, lo que significa que si la ventana no está en pantalla completa, un doble clic habilitará el modo de pantalla completa, y si la ventana ya está en pantalla completa, un doble clic saldrá del modo de pantalla completa.

Primero, necesitamos escuchar el evento de doble clic, y podemos hacerlo con el evento dblclick:



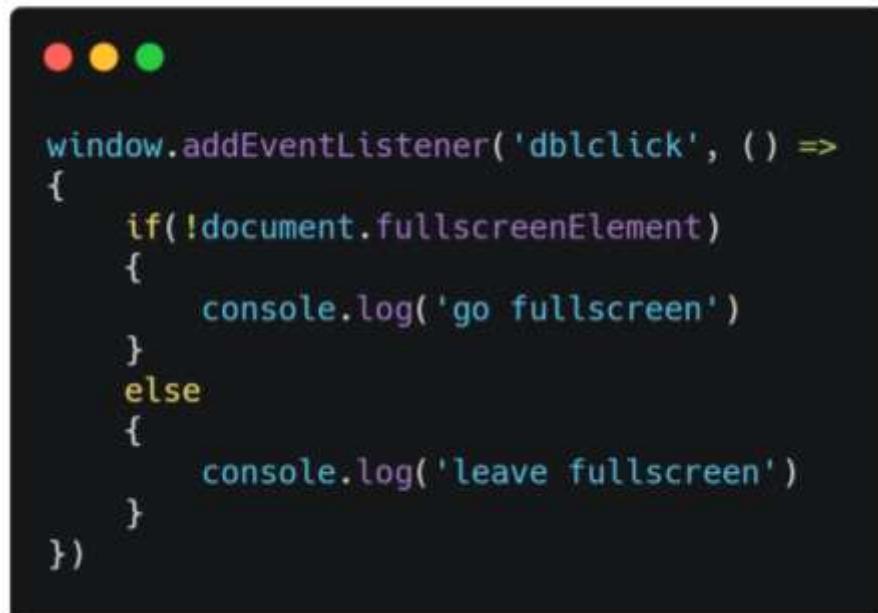
```
window.addEventListener('dblclick', () =>
{
    console.log('double click')
})
```

Este evento funcionará en la mayoría de los navegadores modernos, excepto en Chrome Android: https://developer.mozilla.org/docs/Web/API/Element/dblclick_event

Ahora que tenemos nuestro evento, necesitamos 3 cosas:

- Una forma de saber si ya está en pantalla completa
- Un método para ir al modo de pantalla completa
- Un método para salir del modo de pantalla completa

Para saber si ya estamos en pantalla completa o no, podemos usar document.fullscreenElement:



```
window.addEventListener('dblclick', () =>
{
    if(!document.fullscreenElement)
    {
        console.log('go fullscreen')
    }
    else
    {
        console.log('leave fullscreen')
    }
})
```

El método para solicitar la pantalla completa está asociado con el elemento. Es porque puedes elegir lo que estará en pantalla completa. Puede ser la página completa, cualquier elemento DOM o el <canvas>.

Usaremos <canvas> y llamaremos al método requestFullscreen () en él:

```
window.addEventListener('dblclick', () =>
{
  if(!document.fullscreenElement)
  {
    canvas.requestFullscreen()
  }
  else
  {
    console.log('leave fullscreen')
  }
})
```

El método para salir del modo de pantalla completa está disponible directamente en el documento:

```
window.addEventListener('dblclick', () =>
{
  if(!document.fullscreenElement)
  {
    canvas.requestFullscreen()
  }
  else
  {
    document.exitFullscreen()
  }
})
```

Puede probar el resultado haciendo doble clic en cualquier lugar para alternar el modo de pantalla completa. Desafortunadamente, esto no funcionará en Safari.

Este navegador se está tomando su tiempo para admitir funciones oficialmente simples como la pantalla completa, y necesitamos usar versiones con prefijos para que funcione para `document.fullscreenElement`, `canvas.requestFullscreen` y `document.exitFullscreen`:



```
window.addEventListener('dblclick', () =>
{
  const fullscreenElement = document.fullscreenElement || document.webkitFullscreenElement

  if(!fullscreenElement)
  {
    if(canvas.requestFullscreen)
    {
      canvas.requestFullscreen()
    }
    else if(canvas.webkitRequestFullscreen)
    {
      canvas.webkitRequestFullscreen()
    }
  }
  else
  {
    if(document.exitFullscreen)
    {
      document.exitFullscreen()
    }
    else if(document.webkitExitFullscreen)
    {
      document.webkitExitFullscreen()
    }
  }
})
```

Todo debería funcionar bien en todos los navegadores modernos.

9. Geometrias

Introducción

Hasta ahora, solo usábamos la clase `BoxGeometry` para crear nuestro cubo. En esta lección, descubriremos varias otras geometrías, pero primero, debemos comprender qué es realmente una geometría.

¿Qué es una geometría?

En Three.js, las geometrías se componen de vértices (coordenadas de puntos en espacios 3D) y caras (triángulos que unen esos vértices para crear una superficie).

Usamos geometrías para crear mallas, pero también puedes usar geometrías para formar partículas. Cada vértice (singular de vértices) corresponderá a una partícula, pero esto es para una lección futura.

Podemos almacenar más datos que la posición en los vértices. Un buen ejemplo sería hablar de las coordenadas UV o las normales. Como verá, aprenderemos más sobre ellos más adelante.

Las diferentes geometrías integradas

Three.js tiene muchas geometrías integradas. Si bien no necesita saber con precisión cómo crear una instancia de cada uno, es bueno saber que existen.

Todas las geometrías integradas que veremos heredarán de la clase `BufferGeometry`. Esta clase tiene muchos métodos integrados como `translate (...)`, `rotateX (...)`, `normalize ()`, etc. pero no los usaremos en esta lección.

La mayoría de las páginas de documentación de geometrías tienen ejemplos.

`BoxGeometry` Para crear una caja.

`PlaneGeometry` Para crear un plano rectangular.

`CircleGeometry` Para crear un disco o una parte de un disco (como un gráfico circular).

`ConeGeometry` Para crear un cono o una parte de un cono. Puede abrir o cerrar la base del cono.

`CylinderGeometry` Para crear un cilindro. Puede abrir o cerrar los extremos del cilindro y puede cambiar el radio de cada extremo.

`RingGeometry` Para crear un anillo plano o una parte de un círculo plano.

`TorusGeometry` Para crear un anillo que tenga un grosor (como una dona) o una parte de un anillo.

`TorusKnotGeometry` Para crear algún tipo de geometría de nudos.

`DodecahedronGeometry` Para crear una esfera de 12 caras. Puede agregar detalles para una esfera más redonda.

`OctahedronGeometry` Para crear una esfera de 8 caras. Puede agregar detalles para una esfera más redonda.

`TetrahedronGeometry` Para crear una esfera de 4 caras (no será una gran esfera si no aumenta los detalles). Puede agregar detalles para una esfera más redonda.

`IcosahedronGeometry` Para crear una esfera compuesta por triángulos que tienen aproximadamente el mismo tamaño.

`SphereGeometry` Para crear el tipo más popular de esfera donde las caras parecen cuadrículas (las cuadrículas son solo una combinación de dos triángulos).

`ShapeGeometry` Para crear una forma basada en una ruta.

`TubeGeometry` Para crear un tubo siguiendo una ruta.

`ExtrudeGeometry` Para crear una extrusión basada en una ruta. Puede agregar y controlar el bisel.

LatheGeometry Para crear un jarrón o parte de un jarrón (más como una revolución).
TextGeometry Para crear un texto 3D. Deberá proporcionar la fuente en formato json de tipografía. Si necesita una geometría en particular que no es compatible con Three.js, puede crear su propia geometría en JavaScript, o puede crearla en un software 3D, exportarla e importarla a su proyecto. Aprenderemos más sobre eso más adelante.

Caja de ejemplo

Ya hicimos un cubo pero no hablamos mucho sobre los parámetros. La mayoría de las geometrías tienen parámetros, y siempre debe echar un vistazo a la documentación antes de usarla.

BoxGeometry tiene 6 parámetros:

ancho: el tamaño en el eje x

altura: el tamaño en el eje y

profundidad: el tamaño en el eje z

widthSegments: cuántas subdivisiones hay en el eje x

heightSegments: cuántas subdivisiones hay en el eje y

depthSegments: cuántas subdivisiones hay en el eje z

Las subdivisiones corresponden a la cantidad de triángulos que deben componer la cara. Por defecto es 1, lo que significa que solo habrá 2 triángulos por cara. Si establece la subdivisión en 2, terminará con 8 triángulos por cara:

```
const geometry = new THREE.BoxGeometry(1, 1, 1, 2, 2, 2)
```

El problema es que no podemos ver estos triángulos.

Una buena solución es agregar wireframe: fiel a nuestro material. El wireframe mostrará las líneas que delimitan cada triángulo:

```
const material = new THREE.MeshBasicMaterial({ color: 0xffff00, wireframe: true })
```

Como puede ver, hay 8 triángulos por cara.

Si bien esto no es relevante para un cubo de cara plana, se vuelve más interesante cuando se usa un **SphereGeometry**:

```
const geometry = new THREE.SphereGeometry(1, 32, 32)
```

Cuanta más subdivisiones agreguemos, menos podemos distinguir las caras. Pero tenga en cuenta que demasiados vértices y caras afectarán el rendimiento.

Creando su propia geometría de búfer

A veces, necesitamos crear nuestras propias geometrías. Si la geometría es muy compleja o con una forma precisa, es mejor crearla en un software 3D (y lo cubriremos en una lección futura), pero si la geometría no es demasiado compleja, podemos construirla nosotros mismos usando BufferGeometry.

Para crear su propia geometría de búfer, comience creando una instancia de un BufferGeometry vacío. Crearemos un triángulo simple:

```
// Create an empty BufferGeometry
const geometry = new THREE.BufferGeometry()
```

Para agregar vértices a BufferGeometry, debe comenzar con Float32Array.

Float32Array son matrices nativas de JavaScript. Solo puede almacenar flotantes en el interior, y la longitud de esa matriz es fija.

Para crear un Float32Array, puede especificar su longitud y luego llenarlo más tarde:

```
const positionsArray = new Float32Array(9)

// First vertice
positionsArray[0] = 0
positionsArray[1] = 0
positionsArray[2] = 0

// Second vertice
positionsArray[3] = 0
positionsArray[4] = 1
positionsArray[5] = 0

// Third vertice
positionsArray[6] = 1
positionsArray[7] = 0
positionsArray[8] = 0
```

O puedes pasar un arreglo

```
const positionsArray = new Float32Array([
  0, 0, 0, // First vertex
  0, 1, 0, // Second vertex
  1, 0, 0 // Third vertex
])
```

Como puede ver, las coordenadas de los vértices se especifican linealmente. La matriz es una matriz unidimensional en la que se especifican las x, y z del primer vértice, seguidas de las x, y z del segundo vértice, y así sucesivamente.

Antes de poder enviar esa matriz a BufferGeometry, debe transformarla en un BufferAttribute.

El primer parámetro corresponde a su matriz escrita y el segundo parámetro corresponde a la cantidad de valores que forman un atributo de vértice. Como vimos anteriormente, para leer esta matriz, tenemos que ir 3 por 3 porque una posición de vértice se compone de 3 valores (x, y z):

```
const positionsAttribute = new THREE.BufferAttribute(positionsArray, 3)
```

Luego podemos agregar este atributo a nuestro BufferGeometry usando el método setAttribute (...). El primer parámetro es el nombre de este atributo y el segundo parámetro es el valor:

```
geometry.setAttribute('position', positionsAttribute)
```

Elegimos 'posición' como nombre porque los sombreadores internos de Three.js buscarán ese valor para colocar los vértices. Veremos más sobre eso en las lecciones de sombreadores.

Las caras se crearán automáticamente siguiendo el orden de los vértices.

Todos juntos:

```
// Create an empty BufferGeometry
const geometry = new THREE.BufferGeometry()

// Create a Float32Array containing the vertices position (3 by 3)
const positionsArray = new Float32Array([
    0, 0, 0, // First vertex
    0, 1, 0, // Second vertex
    1, 0, 0 // Third vertex
])

// Create the attribute and name it 'position'
const positionsAttribute = new THREE.BufferAttribute(positionsArray, 3)
geometry.setAttribute('position', positionsAttribute)
```

También podemos crear un montón de triángulos aleatorios:

```
// Create an empty BufferGeometry
const geometry = new THREE.BufferGeometry()

// Create 50 triangles (450 values)
const count = 50
const positionsArray = new Float32Array(count * 3 * 3)
for(let i = 0; i < count * 3 * 3; i++)
{
    positionsArray[i] = (Math.random() - 0.5) * 4
}

// Create the attribute and name it 'position'
const positionsAttribute = new THREE.BufferAttribute(positionsArray, 3)
geometry.setAttribute('position', positionsAttribute)
```

La única dificultad puede ser la parte de contar $* 3 * 3$, pero es bastante simple de explicar: necesitamos 50 triángulos. Cada triángulo está compuesto por 3 vértices y cada vértice está compuesto por 3 valores (x, y z).

Índice

Una cosa interesante con BufferGeometry es que puede mutualizar vértices usando la propiedad index. Considere un cubo. Varias caras pueden usar algunos vértices como los de las esquinas. Y si miras de cerca, cada vértice puede ser utilizado por varios triángulos vecinos. Eso dará como resultado una matriz de atributos más pequeña y una mejora del rendimiento. Pero no cubriremos esta parte en esa lección.

10. Debug UI

11. Texturas

Introducción

¿Estás aburrido de tu cubo rojo? Es hora de agregar algunas texturas.

Pero primero, ¿qué son las texturas y qué podemos hacer realmente con ellas?

¿Qué son las texturas?

Las texturas, como probablemente sepas, son imágenes que cubrirán la superficie de sus geometrías. Muchos tipos de texturas pueden tener diferentes efectos en la apariencia de su geometría. No se trata solo del color.

Estos son los tipos de texturas más comunes que utilizan una famosa textura de puerta de João Paulo. Cómprale un Ko-fi o conviértete en un Patreon si te gusta su trabajo.

Color (o albedo) La textura del albedo es la más simple. Solo tomará los píxeles de la textura y los aplicará a la geometría.

Alfa La textura alfa es una imagen en escala de grises donde el blanco será visible y el negro no.

Altura La textura de altura es una imagen en escala de grises que moverá los vértices para crear cierto relieve. Deberá agregar una subdivisión si desea verla.

Normal La textura normal agregará pequeños detalles. No moverá los vértices, pero atraerá a la luz para que piense que la cara está orientada de manera diferente. Las texturas normales son muy útiles para agregar detalles con buen rendimiento porque no es necesario subdividir la geometría.

Oclusión ambiental La textura de oclusión ambiental es una imagen en escala de grises que simula sombras en las grietas de la superficie. Si bien no es físicamente preciso, ciertamente ayuda a crear contraste.

Metalidad La textura de metal es una imagen en escala de grises que especificará qué parte es metálica (blanca) y no metálica (negra). Esta información ayudará a crear una reflexión.

Rugosidad La rugosidad es una imagen en escala de grises que viene con metalidad y que especificará qué parte es rugosa (blanca) y qué parte es lisa (negra). Esta información ayudará a disipar la luz. Una alfombra es muy resistente y no verá el reflejo de la luz en ella, mientras que la superficie del agua es muy suave y puede ver la luz reflejada en ella. Aquí, la madera es uniforme porque tiene una capa transparente.

PBR

Esas texturas (especialmente el metal y la rugosidad) siguen lo que llamamos principios PBR. PBR son las siglas de Physically Based Rendering (Representación basada en la física). Reagrupa muchas técnicas que tienden a seguir instrucciones de la vida real para obtener resultados realistas.

Si bien existen muchas otras técnicas, PBR se está convirtiendo en el estándar para renderizados realistas y muchos software, motores y bibliotecas lo están utilizando.

Por ahora, simplemente nos centraremos en cómo cargar texturas, cómo usarlas, qué transformaciones podemos aplicar y cómo optimizarlas. Veremos más sobre PBR en lecciones posteriores.

Cómo cargar texturas

Obtener la URL de la imagen

Para cargar la textura, necesitamos la URL del archivo de imagen.

Debido a que usamos Webpack, hay dos formas de obtenerlo.

Puede poner la textura de la imagen en la carpeta / src / e importarla como importaría una dependencia de JavaScript:

Usando JavaScript nativo

Con JavaScript nativo, primero debe crear una instancia de imagen, escuchar el evento de carga y luego cambiar su propiedad src para comenzar a cargar la imagen:

```
const image = new Image()
image.onload = () =>
{
    console.log('image loaded')
}
image.src = '/textures/door/color.jpg'
```

Debería ver que aparece 'imagen cargada' en la consola. Como puede ver, configuramos la fuente en '/textures/door/color.jpg' sin la carpeta / static en la ruta.

No podemos usar esa imagen directamente. Primero debemos crear una textura a partir de esa imagen.

Esto se debe a que WebGL necesita un formato muy específico al que pueda acceder la GPU y también porque se aplicarán algunos cambios a las texturas como el mipmapping, pero veremos más sobre eso un poco más adelante.

Crea la textura con la clase Texture:

```
const image = new Image()
image.addEventListener('load', () =>
{
    const texture = new THREE.Texture(image)
})
image.src = '/textures/door/color.jpg'
```

Lo que tenemos que hacer ahora es usar esa textura en el material. Desafortunadamente, la variable de textura ha sido declarada en una función y no podemos acceder a ella fuera de esta función. Esta es una limitación de JavaScript llamada alcance.

Podríamos crear la malla dentro de la función, pero hay una mejor solución que consiste en crear la textura fuera de la función y luego actualizarla una vez que se carga la imagen estableciendo la propiedad de textura `needUpdate` en `true`:

```
const image = new Image()
const texture = new THREE.Texture(image)
image.addEventListener('load', () =>
{
    texture.needsUpdate = true
})
image.src = '/textures/door/color.jpg'
```

Mientras hace esto, puede usar inmediatamente la variable de textura inmediatamente, y la imagen será transparente hasta que se cargue.

Para ver la textura en el cubo, reemplace la propiedad de color por mapa y use la textura como valor:

```
const material = new THREE.MeshBasicMaterial({ map: texture })
```

Debería ver la textura de la puerta a cada lado de su cubo.

Usando TextureLoader

La técnica de JavaScript nativa no es tan complicada, pero existe una forma aún más sencilla con `TextureLoader`.

Cree una instancia de una variable usando la clase `TextureLoader` y use su método `.load(...)` para crear una textura:

```
const textureLoader = new THREE.TextureLoader()
const texture = textureLoader.load('/textures/door/color.jpg')
```

Internamente, Three.js hará lo que hizo antes para cargar la imagen y actualizar la textura una vez que esté lista.

Puede cargar tantas texturas como desee con una sola instancia de TextureLoader.

Puede enviar 3 funciones después de la ruta. Serán convocados para los siguientes eventos:

- cargar cuando la imagen se cargó correctamente
- progreso cuando la carga está progresando
- error si algo salió mal



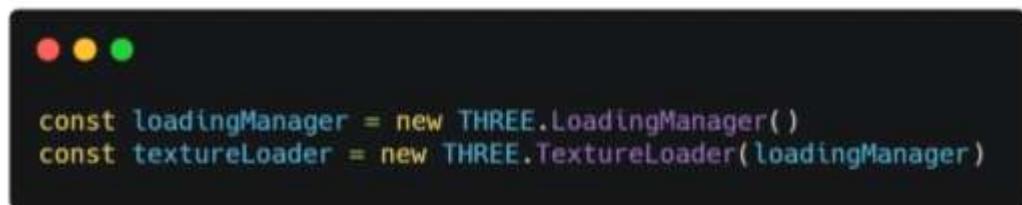
```
const textureLoader = new THREE.TextureLoader()
const texture = textureLoader.load(
  '/textures/door/color.jpg',
  () =>
{
  console.log('loading finished')
},
() =>
{
  console.log('loading progressing')
},
() =>
{
  console.log('loading error')
}
)
```

Si la textura no funciona, podría ser útil agregar esas funciones de devolución de llamada para ver qué está sucediendo y detectar errores.

Usando el LoadingManager

Finalmente, si tiene varias imágenes para cargar y desea mutualizar los eventos, como recibir una notificación cuando se cargan todas las imágenes, puede usar un LoadingManager.

Crea una instancia de la clase LoadingManager y pásala al TextureLoader:



```
const loadingManager = new THREE.LoadingManager()
const textureLoader = new THREE.TextureLoader(loadingManager)
```

Puede escuchar los diversos eventos reemplazando las siguientes propiedades por sus propias funciones onStart, onLoad, onProgress y onError:

```
const loadingManager = new THREE.LoadingManager()
loadingManager.onStart = () =>
{
    console.log('loading started')
}
loadingManager.onLoad = () =>
{
    console.log('loading finished')
}
loadingManager.onProgress = () =>
{
    console.log('loading progressing')
}
loadingManager.onError = () =>
{
    console.log('loading error')
}

const textureLoader = new THREE.TextureLoader(loadingManager)
```

Ahora puede comenzar a cargar todas las imágenes que necesita:

```
const colorTexture = textureLoader.load('/textures/door/color.jpg')
const alphaTexture = textureLoader.load('/textures/door/alpha.jpg')
const heightTexture = textureLoader.load('/textures/door/height.jpg')
const normalTexture = textureLoader.load('/textures/door/normal.jpg')
const ambientOcclusionTexture = textureLoader.load('/textures/door/ambientOcclusion.jpg')
const metalnessTexture = textureLoader.load('/textures/door/metalness.jpg')
const roughnessTexture = textureLoader.load('/textures/door/roughness.jpg')
```

Como puede ver aquí, cambiamos el nombre de la variable de textura `colorTexture`, así que no olvide cambiarlo también en el material:

```
const material = new THREE.MeshBasicMaterial({ map: colorTexture })
```

El `LoadingManager` es muy útil si desea mostrar un cargador y ocultarlo solo cuando todos los activos están cargados. Como veremos en una lección futura, también puede usarlo con otros tipos de cargadores.

Desenvolvimiento UV

Si bien es bastante lógico cómo colocar una textura en un cubo, las cosas pueden ser un poco más complicadas para otras geometrías.

Intente reemplazar su BoxGeometry con otras geometrías:

```
const geometry = new THREE.BoxGeometry(1, 1, 1)

// Or
const geometry = new THREE.SphereGeometry(1, 32, 32)

// Or
const geometry = new THREE.ConeGeometry(1, 1, 32)

// Or
const geometry = new THREE.TorusGeometry(1, 0.35, 32, 100)
```

Como puede ver, la textura se estira o aprieta de diferentes formas para cubrir la geometría.

Eso se llama desenvolvimiento UV. Puede imaginarlo como desenvolver un origami o un envoltorio de caramelo para que quede plano. Cada vértice tendrá una coordenada 2D en un plano (generalmente cuadrado).

De hecho, puede ver esas coordenadas UV 2D en la propiedad `geometry.attributes.uv`:
`console.log(geometry.attributes.uv)`

JavaScript

Three.js genera esas coordenadas UV cuando usa las primitivas. Si crea su propia geometría y desea aplicarle una textura, deberá especificar las coordenadas UV.

Si está haciendo la geometría con un software 3D, también tendrá que desenvolver UV.

No te preocupes; la mayoría del software 3D también tiene desenvolvimiento automático que debería funcionar.

Transformando la textura

Volvamos a nuestro cubo con una textura y veamos qué tipo de transformaciones podemos aplicar a esa textura.

Repetir

Puede repetir la textura usando la propiedad `repeat`, que es un `Vector2`, lo que significa que tiene propiedades `x` y `y`.

Intente cambiar estas propiedades:

```
const colorTexture = textureLoader.load('/textures/door/color.jpg')
colorTexture.repeat.x = 2
colorTexture.repeat.y = 3
```

Como puede ver, la textura no se repite, pero es más pequeña y el último píxel parece estirado. Eso se debe a que la textura no está configurada para repetirse por defecto. Para cambiar eso, debe actualizar las propiedades `wrapS` y `wrapT` usando la constante `THREE.RepeatWrapping`.

`wrapS` es para el eje x

`wrapT` es para el eje y

`colorTexture.wrapS = THREE.RepeatWrapping`

`colorTexture.wrapT = THREE.RepeatWrapping`

También puede alternar la dirección con `TRES`.

`colorTexture.wrapS = THREE.MirroredRepeatWrapping`

`colorTexture.wrapT = THREE.MirroredRepeatWrapping`

Offset

Puede compensar la textura utilizando la propiedad de desplazamiento que también es un `Vector2` con propiedades `x` y `y`. Cambiar estos simplemente compensará las coordenadas UV:



```
colorTexture.offset.x = 0.5  
colorTexture.offset.y = 0.5
```

Rotación

Puede rotar la textura usando la propiedad `rotación`, que es un número que corresponde al ángulo en radianes:



```
colorTexture.rotation = Math.PI * 0.25
```

Si elimina las propiedades de desplazamiento y repetición, verá que la rotación se produce alrededor de la esquina inferior izquierda de las caras del cubo:

Es decir, de hecho, las coordenadas UV 0, 0. Si desea cambiar el pivote de esa rotación, puede hacerlo usando la propiedad `center` que también es un `Vector2`:



```
colorTexture.rotation = Math.PI * 0.25  
colorTexture.center.x = 0.5  
colorTexture.center.y = 0.5
```

La textura ahora rotará en su centro.

Filtrado y mapeo Mip

Si miras la cara superior del cubo mientras esta cara está casi oculta, verás una textura muy borrosa. Eso se debe al filtrado y al mipmapping.

Mipmapping (o "mapeo mip" con un espacio) es una técnica que consiste en crear la mitad de una versión más pequeña de una textura una y otra vez hasta obtener una textura 1x1. Todas esas variaciones de textura se envían a la GPU, y la GPU elegirá la versión más adecuada de la textura.

Three.js y la GPU ya manejan todo esto, y puede configurar qué algoritmo de filtro usar. Hay dos tipos de algoritmos de filtro: el filtro de minificación y el filtro de aumento.

Filtro de minificación

El filtro de minificación ocurre cuando los píxeles de textura son más pequeños que los píxeles del render. En otras palabras, la textura es demasiado grande para la superficie, cubre. Puede cambiar el filtro de minificación de la textura utilizando la propiedad `minFilter`. Hay 6 valores posibles:

TRES. Filtro más cercano

TRES.LinearFilter

TRES. Más cercanoMipmapNearestFilter

THREE.NearestMipmapLinearFilter

TRES.LinearMipmapNearestFilter

TRES.LinearMipmapLinearFilter

El valor predeterminado es THREE.LinearMipmapLinearFilter. Si no está satisfecho con el aspecto de su textura, debe probar los otros filtros.

No veremos cada uno, pero probaremos el THREE.NearestFilter, que tiene un resultado muy diferente:



```
colorTexture.minFilter = THREE.NearestFilter
```

Si está utilizando un dispositivo con una proporción de píxeles superior a uno, no verá mucha diferencia. De lo contrario, coloque la cámara donde esta cara esté casi oculta, y debería obtener más detalles y artefactos extraños.

Si prueba con la textura `checkerboard-1024x1024.png` ubicada en la carpeta `/static/textures/`, verá esos artefactos con mayor claridad:



```
const colorTexture = textureLoader.load('/textures/checkerboard-1024x1024.png')
```

Los artefactos que ve se denominan patrones de mumaré y, por lo general, desea evitarlos.

Filtro de aumento

El filtro de ampliación funciona igual que el filtro de minificación, pero cuando los píxeles de la textura son más grandes que los píxeles del render. En otras palabras, la textura es demasiado pequeña para la superficie que cubre.

Puedes ver el resultado usando la textura checkerboard-8x8.png que también se encuentra en la carpeta static / textures /:



Formato de textura y optimización

Cuando estés preparando tus texturas, debes tener en cuenta 3 elementos cruciales:

- El tamaño (o la resolución)
- Los datos
- El peso

No olvide que los usuarios que accedan a su sitio web deberán descargar esas texturas. Puede usar la mayoría de los tipos de imágenes que usamos en la web, como .jpg (compresión con pérdida pero generalmente más ligera) o .png (compresión sin pérdida pero generalmente más pesada).

Intente aplicar los métodos habituales para obtener una imagen aceptable pero lo más clara posible. Puede utilizar sitios web de compresión como TinyPNG (también funciona con jpg) o cualquier software.

El tamaño

Cada píxel de las texturas que esté utilizando deberá almacenarse en la GPU independientemente del peso de la imagen. Y al igual que su disco duro, la GPU tiene limitaciones de almacenamiento. Es incluso peor porque el mipmapping generado automáticamente aumenta la cantidad de píxeles que deben almacenarse.

Intente reducir el tamaño de sus imágenes tanto como sea posible.

Si recuerda lo que dijimos sobre el mapeo mip, Three.js producirá una versión la mitad más pequeña de la textura repetidamente hasta que obtenga una textura 1x1. Por eso, el ancho y alto de la textura deben ser una potencia de 2. Eso es obligatorio para que Three.js pueda dividir el tamaño de la textura por 2.

Algunos ejemplos: 512x512, 1024x1024 o 512x2048

512, 1024 y 2048 se pueden dividir por 2 hasta llegar a 1.

Si está utilizando una textura con un ancho o alto diferente a un valor de potencia de 2, Three.js intentará estirarlo a la potencia más cercana del número 2, lo que puede tener resultados visualmente pobres, y también recibirá una advertencia en la consola.

Los datos

Todavía no lo hemos probado, porque tenemos otras cosas que ver primero, pero las texturas admiten la transparencia. Como sabrá, los archivos jpg no tienen un canal alfa, por lo que es posible que prefiera usar un png.

O puede usar un mapa alfa, como veremos en una lección futura.

Si está usando una textura normal (la púrpura), probablemente querrá tener los valores exactos para los canales rojo, verde y azul de cada píxel, o podría terminar con fallas visuales. Para eso, necesitará usar un png porque su compresión sin pérdida preservará los valores.

Dónde encontrar texturas

Desafortunadamente, siempre es difícil encontrar las texturas perfectas. Hay muchos sitios web, pero las texturas no siempre son correctas y es posible que deba pagar.

Probablemente sea una buena idea comenzar buscando en la web. Aquí hay algunos sitios web en los que termino con frecuencia.

poliigon.com

3dtextures.me

arroway-textures.ch

Siempre asegúrese de tener derecho a usar la textura si no es para uso personal.

También puede crear las suyas propias usando fotos y software 2D como Photoshop o incluso texturas procedimentales con software como Substance Designer.

12. Materiales

Introducción

Los materiales se utilizan para poner un color en cada píxel visible de las geometrías.

Los algoritmos que deciden el color de cada píxel se escriben en programas llamados sombreadores.

Escribir sombreadores es una de las partes más desafiantes de WebGL y Three.js, pero no se preocupe; Three.js tiene muchos materiales incorporados con sombreadores prefabricados.

Descubriremos cómo crear nuestros propios sombreadores en una lección futura. Por ahora, usemos los materiales de Three.js.

Configuración

El motor de arranque no contiene ningún objeto. Esta es una excelente ocasión para revisar los conceptos básicos de la creación de mallas.

Prepara nuestra escena

Para probar los materiales, debemos preparar una hermosa escena y cargar algunas texturas.

Cree 3 mallas compuestas por 3 geometrías diferentes (una esfera, un plano y un toro) y use el mismo MeshBasicMaterial en todos los 3. Sí, puede usar un material en varias mallas. Mueve la esfera de la izquierda y el toro de la derecha para separarlos.

El método add (...) admite la adición de varios objetos a la vez:

```
/**  
 * Objects  
 */  
const material = new THREE.MeshBasicMaterial()  
  
const sphere = new THREE.Mesh(  
    new THREE.SphereGeometry(0.5, 16, 16),  
    material  
)  
sphere.position.x = - 1.5  
  
const plane = new THREE.Mesh(  
    new THREE.PlaneGeometry(1, 1),  
    material  
)  
  
const torus = new THREE.Mesh(  
    new THREE.TorusGeometry(0.3, 0.2, 16, 32),  
    material  
)  
torus.position.x = 1.5  
  
scene.add(sphere, plane, torus)
```

Ahora podemos rotar nuestros objetos en nuestra función tick como hicimos en la lección de Animación:

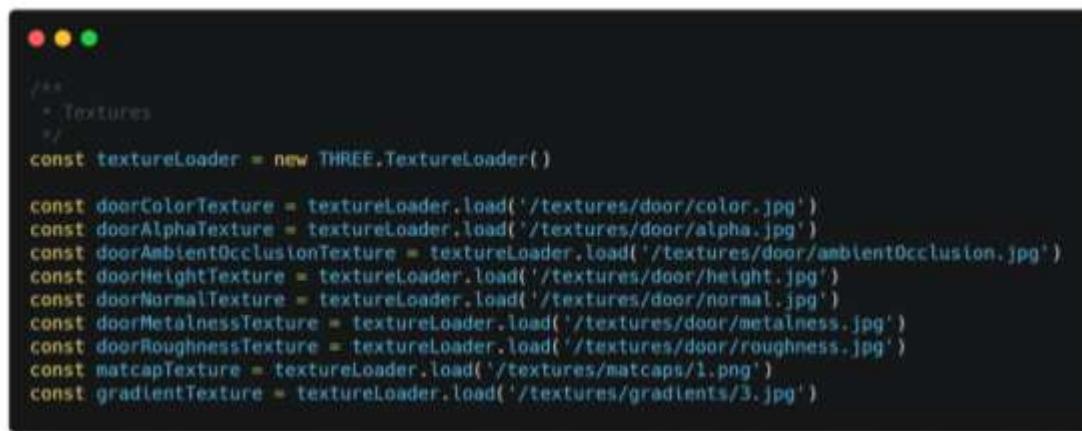
```
/*  
 * Animate  
 */  
const clock = We can now rotate our objects on our tick function as we did on the Animation lesson: new  
THREE.Clock()  
  
const tick = () =>  
{  
    const elapsedTime = clock.getElapsedTime()  
  
    // Update objects  
    sphere.rotation.y += 0.1 * elapsedTime  
    plane.rotation.y += 0.1 * elapsedTime  
    torus.rotation.y += 0.1 * elapsedTime  
  
    sphere.rotation.x += 0.15 * elapsedTime  
    plane.rotation.x += 0.15 * elapsedTime  
    torus.rotation.x += 0.15 * elapsedTime  
  
    // ...  
}  
  
tick()
```

Deberías ver tus 3 objetos girando lentamente.

Los materiales que vamos a descubrir utilizan texturas de muchas formas diferentes. Carguemos algunas texturas usando TextureLoader como hicimos en la lección de Texturas.

Todas las imágenes de textura se encuentran en la carpeta / static / textures /. Por ahora, cargaremos todas las texturas de puerta ubicadas en la carpeta / static / textures / door /, la primera textura matcap ubicada en la carpeta / static / textures / matcaps / y la primera textura degradada ubicada en / static / textures / degradados / carpeta.

Asegúrese de hacer eso antes de crear una instancia del material:



```
const textureLoader = new THREE.TextureLoader()

const doorColorTexture = textureLoader.load('/textures/door/color.jpg')
const doorAlphaTexture = textureLoader.load('/textures/door/alpha.jpg')
const doorAmbientOcclusionTexture = textureLoader.load('/textures/door/ambientOcclusion.jpg')
const doorHeightTexture = textureLoader.load('/textures/door/height.jpg')
const doorNormalTexture = textureLoader.load('/textures/door/normal.jpg')
const doorMetalnessTexture = textureLoader.load('/textures/door/metalness.jpg')
const doorRoughnessTexture = textureLoader.load('/textures/door/roughness.jpg')
const matcapTexture = textureLoader.load('/textures/matcaps/1.png')
const gradientTexture = textureLoader.load('/textures/gradients/3.jpg')
```

Para asegurarse de que todas las texturas estén bien cargadas, puede usarlas en su material con la propiedad del mapa, como vimos en la lección Texturas.

```
const material = new THREE.MeshBasicMaterial({ map: doorColorTexture })
```

Hasta ahora, solo usábamos MeshBasicMaterial, que aplica un color uniforme o una textura en nuestra geometría.

Si busca "material" en la documentación de Three.js, verá que hay muchas clases que podemos usar. Probémoslos.

MallaBásicoMaterial

MeshBasicMaterial es probablemente el material más "básico" ... Pero hay múltiples propiedades que aún no hemos cubierto.

Puede establecer la mayoría de esas propiedades mientras crea una instancia del material en el objeto que enviamos como parámetro, pero también puede cambiar esas propiedades en la instancia directamente:



```
const material = new THREE.MeshBasicMaterial({
  map: doorColorTexture
})

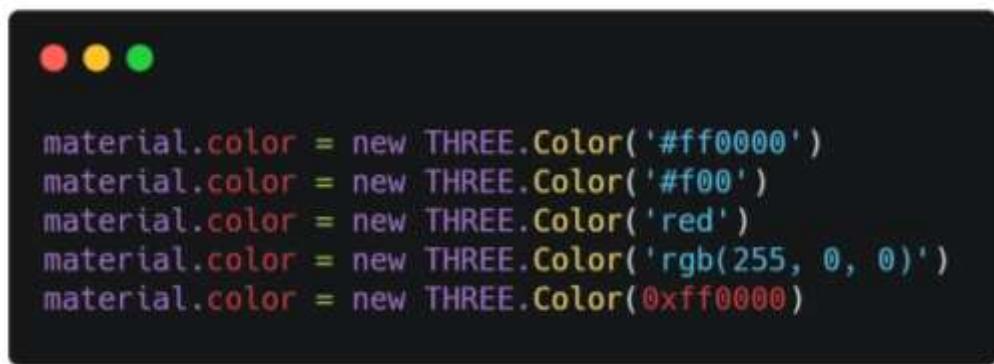
// Equals
const material = new THREE.MeshBasicMaterial()
material.map = doorColorTexture
```

Usaremos el segundo método, pero siéntase libre de hacer lo que quiera.

La propiedad del mapa aplicará una textura en la superficie de la geometría:

```
material.map = doorColorTexture  
material.map = doorColorTexture
```

La propiedad de color aplicará un color uniforme en la superficie de la geometría. Cuando cambia la propiedad de color directamente, debe crear una instancia de una clase de Color. Puede utilizar muchos formatos diferentes:



```
material.color = new THREE.Color('#ff0000')  
material.color = new THREE.Color('#f00')  
material.color = new THREE.Color('red')  
material.color = new THREE.Color('rgb(255, 0, 0)')  
material.color = new THREE.Color(0xff0000)
```

La combinación de color y mapa teñirá la textura con el color:

```
material.map = doorColorTexture  
material.color = new THREE.Color('#ff0000')
```

La propiedad wireframe mostrará los triángulos que componen su geometría con una línea delgada de 1px independientemente de la distancia de la cámara:

```
material.wireframe = true
```

La propiedad de opacidad controla la transparencia, pero, para que funcione, debe establecer la propiedad de transparencia en verdadero para informar a Three.js que este material ahora admite transparencia:

```
material.transparent = true  
material.opacity = 0.5
```

Ahora que la transparencia está funcionando, podemos usar la propiedad alphaMap para controlar la transparencia con una textura:

```
material.transparent = true  
material.alphaMap = doorAlphaTexture
```

La propiedad lateral le permite decidir qué lado de una cara es visible. De forma predeterminada, la parte frontal es visible (TRES.Lado frontal), pero puede mostrar la parte posterior en su lugar (TRES.Lado trasero) o ambos (TRES.DOBLE LADO):

```
material.side = THREE.DoubleSide
```

Debería ver tanto la parte delantera como la trasera del avión.

Trate de evitar el uso de THREE.DoubleSide porque renderizar ambos lados significa tener el doble de triángulos para renderizar.

Algunas de estas propiedades, como la estructura alámbrica o la opacidad, se pueden utilizar con otros tipos de materiales. No los repetiremos todo el tiempo.

MallaNormalMaterial

MeshNormalMaterial muestra un bonito color violeta, azulado y verdoso que se parece a la textura normal que vimos en las lecciones de Texturas. Eso no es casualidad porque ambos están relacionados con lo que llamamos normales:

```
const material = new THREE.MeshNormalMaterial()
```

Las normales son información codificada en cada vértice que contiene la dirección del exterior de la cara. Si mostrara esas normales como flechas, obtendría líneas rectas que salen de cada vértice que compone su geometría.

Puede usar Normales para muchas cosas, como calcular cómo iluminar la cara o cómo el entorno debe reflejarse o refractarse en la superficie de las geometrías.

Al usar MeshNormalMaterial, el color solo mostrará la orientación del pariente normal a la cámara. Si gira alrededor de la esfera, verá que el color es siempre el mismo, independientemente de la parte de la esfera que esté mirando.

Si bien puede usar algunas de las propiedades que descubrimos con MeshBasicMaterial como estructura alámbrica, transparente, opacidad y lateral, también hay una nueva propiedad que puede usar, que se llama flatShading:

```
material.flatShading = true
```

flatShading aplanará las caras, lo que significa que las normales no se interpolarán entre los vértices.

MeshNormalMaterial puede ser útil para depurar los normales, pero también se ve muy bien, y puede usarlo tal como lo hizo ilithya en su portafolio <https://www.ilithya.rocks>.

MallaMatcapMaterial

MeshMatcapMaterial es un material fantástico debido a lo bien que puede verse a la vez que tiene un gran rendimiento.

Para que funcione, MeshMatcapMaterial necesita una textura de referencia que parezca una esfera.



```
const material = new THREE.MeshMatcapMaterial()
material.matcap = matcapTexture
```

Las mallas aparecerán iluminadas, pero es solo una textura que se parece a ella.

El único problema es que la ilusión es la misma independientemente de la orientación de la cámara. Además, no puede actualizar las luces porque no hay ninguna.

Pruebe diferentes texturas disponibles en la carpeta / static / textures / matcaps / (solo una de las líneas a continuación):

```
const matcapTexture = textureLoader.load('/textures/matcaps/2.png')
const matcapTexture = textureLoader.load('/textures/matcaps/3.png')
const matcapTexture = textureLoader.load('/textures/matcaps/4.png')
const matcapTexture = textureLoader.load('/textures/matcaps/5.png')
const matcapTexture = textureLoader.load('/textures/matcaps/6.png')
const matcapTexture = textureLoader.load('/textures/matcaps/7.png')
const matcapTexture = textureLoader.load('/textures/matcaps/8.png')
```

En cuanto a dónde encontrar texturas matcaps, puedes hacer una simple búsqueda en la web como cualquier tipo de texturas. Asegúrate de tener derecho a usar la textura si no es para uso personal. También está esta amplia lista de matcaps: <https://github.com/nidoxr/matcaps>

También puede crear sus propios matcaps usando un software 3D renderizando una esfera frente a la cámara en una imagen cuadrada. Finalmente, puede intentar hacer un matcap en un software 2D como Photoshop.

Malla ProfundidadMaterial

MeshDepthMaterial simplemente coloreará la geometría en blanco si está cerca del valor cercano de la cámara y en negro si está cerca del lejano

```
const material = new THREE.MeshDepthMaterial()
```

Puede utilizar este material para efectos especiales en los que necesite saber qué tan lejos está el píxel de la cámara. Lo usaremos en una lección futura.

Añadiendo algunas luces

Los siguientes materiales necesitan luces para ser vistos. Agreguemos dos luces simples a nuestra escena.

Crea una AmbientLight y agrégala a la escena:

```
/*
 * Lights
 */
const ambientLight = new THREE.AmbientLight(0xffffff, 0.5)
scene.add(ambientLight)
```

Cree un PointLight y agréguelo a la escena:



```
// ...
const pointLight = new THREE.PointLight(0xffffff, 0.5)
pointLight.position.x = 2
pointLight.position.y = 3
pointLight.position.z = 4
scene.add(pointLight)
```

Veremos más sobre las luces, cómo funcionan y cómo modificarlas en una lección futura.

MallaLambertMaterial

El MeshLambertMaterial es el primer material que reacciona a la luz que vamos a utilizar:

```
const material = new THREE.MeshLambertMaterial()
```

Las cosas se están poniendo realistas, como puede ver. Si bien la iluminación no es muy convincente, es un buen comienzo.

MeshLambertMaterial admite las mismas propiedades que MeshBasicMaterial, pero también algunas propiedades relacionadas con las luces. Veremos esas propiedades más adelante en la lección con materiales más adecuados.

El MeshLambertMaterial es el material de mayor rendimiento que utiliza luces. Desafortunadamente, los parámetros no son convenientes y puede ver patrones extraños en la geometría si observa de cerca las geometrías redondeadas como la esfera.

MallaPhongMaterial

El MeshPhongMaterial es muy similar al MeshLambertMaterial, pero los patrones extraños son menos visibles, y también puede ver el reflejo de la luz en la superficie de la geometría:

```
const material = new THREE.MeshPhongMaterial()
```

MeshPhongMaterial es menos eficaz que MeshLambertMaterial. Sin embargo, realmente no importa a este nivel.

Puede controlar el reflejo de la luz con la propiedad de brillo. Cuanto mayor sea el valor, más brillante será la superficie. También puede cambiar el color del reflejo utilizando la propiedad especular:

```
material.shininess = 100
```

```
material.specular = new THREE.Color(0x1188ff)
```

MallaEstándarMaterial

MeshStandardMaterial utiliza principios de representación basados en la física. Sí, estamos hablando del PBR que vimos en la lección de Texturas. Al igual que MeshLambertMaterial y MeshPhongMaterial, admite luces pero con un algoritmo más realista y mejores parámetros como rugosidad y metalidad.

Se llama "estándar" porque el PBR se está convirtiendo en un estándar en muchos software, motores y bibliotecas. La idea es tener un resultado realista con parámetros realistas, y debería tener un resultado muy similar independientemente de la tecnología que esté utilizando:

```
const material = new THREE.MeshStandardMaterial()
```

Puede cambiar las propiedades de rugosidad y metalidad directamente:

```
material.metalness = 0.45
```

```
material.roughness = 0.65
```

Agregar una interfaz de usuario de depuración

Si bien esto no es necesario, ahora sería un excelente momento para agregar una interfaz de usuario de depuración. Eso será muy útil para probar las diferentes propiedades.

Primero, debemos agregar la dependencia Dat.GUI a nuestro proyecto. En la terminal, en la carpeta del proyecto (donde el servidor debería estar ejecutándose actualmente), use el siguiente comando:

```
npm install --save dat.gui
```

Luego, en la parte superior de su código, importe dat.gui (no olvide volver a iniciar el servidor con

```
npm run dev si lo detuvo):
```

```
import * as dat from 'dat.gui'  
/**  
 * Debug  
 */  
const gui = new dat.GUI()
```

Y agregue los ajustes (después de crear el material):

```
gui.add(material, 'metalness').min(0).max(1).step(0.0001)
```

```
gui.add(material, 'roughness').min(0).max(1).step(0.0001)
```

Y eso es. Ahora puede cambiar la metalidad y la rugosidad a su gusto.

Continuemos con las otras propiedades de MeshStandardMaterial.

La propiedad del mapa le permite aplicar una textura simple. Puede utilizar doorColorTexture:

```
material.map = doorColorTexture
```

La propiedad aoMap (literalmente "mapa de oclusión ambiental") agregará sombras donde la textura es oscura. Para que funcione, debes agregar lo que llamamos un segundo conjunto de UV (las coordenadas que ayudan a posicionar las texturas en las geometrías).

Simplemente podemos agregar nuevos atributos como hicimos en la lección de Geometrías y usar el atributo uv predeterminado. En términos más simples, duplicamos el atributo uv.

```
sphere.geometry.setAttribute('uv2', new THREE.BufferAttribute(sphere.geometry.attributes.uv.array, 2))  
plane.geometry.setAttribute('uv2', new THREE.BufferAttribute(plane.geometry.attributes.uv.array, 2))  
torus.geometry.setAttribute('uv2', new THREE.BufferAttribute(torus.geometry.attributes.uv.array, 2))
```

Ahora puede agregar el aoMap usando la textura doorAmbientOcclusionTexture y controlar la intensidad usando la propiedad aoMapIntensity:

```
material.aoMap = doorAmbientOcclusionTexture
```

```
material.aoMapIntensity = 1
```

Las grietas deben verse más oscuras, lo que crea contraste y agrega dimensión.

La propiedad displacementMap moverá los vértices para crear un verdadero relieve:

```
material.displacementMap = doorHeightTexture
```

Debería verse terrible. Eso se debe a la falta de vértices en nuestras geometrías (necesitamos más subdivisiones) y el desplazamiento es demasiado fuerte:

```
material.displacementScale = 0.05  
// ...  
new THREE.SphereGeometry(0.5, 64, 64),  
// ...  
new THREE.PlaneGeometry(1, 1, 100, 100),  
// ...  
new THREE.TorusGeometry(0.3, 0.2, 64, 128),
```

En lugar de especificar una metalidad y rugosidad uniformes para toda la geometría, podemos usar `metalnessMap` y `roughnessMap`:

```
material.metalnessMap = doorMetalnessTexture  
material.roughnessMap = doorRoughnessTexture
```

El reflejo parece extraño porque las propiedades de metalidad y rugosidad aún afectan a cada mapa respectivamente. Deberíamos comentarlos o utilizar sus valores originales:

```
material.metalness = 0  
material.roughness = 1
```

El mapa normal simulará la orientación normal y agregará detalles en la superficie independientemente de la subdivisión:

```
material.normalMap = doorNormalTexture
```

Puede cambiar la intensidad normal con la propiedad `normalScale`. Cuidado, es un `Vector2`:

```
material.normalScale.set(0.5, 0.5)
```

Y finalmente, puede controlar el alfa usando la propiedad `alphaMap`. No olvide establecer la propiedad `transparent` en verdadero:

```
material.transparent = true  
material.alphaMap = doorAlphaTexture
```

Aquí tienes una hermosa puerta. Siéntase libre de modificar las propiedades y probar cosas.

MallaFísicoMaterial

El `MeshPhysicalMaterial` es el mismo que el `MeshStandardMaterial` pero con el apoyo de un efecto de capa transparente. Puede controlar las propiedades de esa capa transparente e incluso usar una textura como en este ejemplo de `Three.js`, pero no probaremos esta aquí.

PuntosMaterial

Puede utilizar PointsMaterial con partículas. Veremos más sobre eso en una lección dedicada.

ShaderMaterial y RawShaderMaterial

ShaderMaterial y RawShaderMaterial se pueden usar para crear sus propios materiales, pero veremos más sobre eso en una lección dedicada.

Mapa del entorno

El mapa del entorno es como una imagen de lo que rodea la escena. Puede usarlo para agregar reflexión o refracción a sus objetos. También se puede utilizar como información de iluminación.

Aún no lo hemos cubierto, pero puede usarlo con muchos de los materiales que vimos.

Primero, configuremos un MeshStandardMaterial muy simple con la interfaz de usuario de depuración como lo hicimos anteriormente:

```
const material = new THREE.MeshStandardMaterial()
material.metalness = 0.7
material.roughness = 0.2
gui.add(material, 'metalness').min(0).max(1).step(0.0001)
gui.add(material, 'roughness').min(0).max(1).step(0.0001)
```

Para agregar el mapa de entorno a nuestro material, debemos usar la propiedad envMap. Three.js solo admite mapas de entorno de cubo. Los mapas de entorno de cubo son 6 imágenes y cada una corresponde a un lado del entorno.

Puede encontrar varios mapas de entorno en la carpeta / static / textures / environmentMap /.

Para cargar una textura de cubo, debe usar CubeTextureLoader en lugar de TextureLoader.

Cree una instancia de CubeTextureLoader antes de instanciar el material y llame a su método load (...) pero use una matriz de rutas en lugar de una ruta:

```
const cubeTextureLoader = new THREE.CubeTextureLoader()

const environmentMapTexture = cubeTextureLoader.load([
    '/textures/environmentMaps/0/px.jpg',
    '/textures/environmentMaps/0/nx.jpg',
    '/textures/environmentMaps/0/py.jpg',
    '/textures/environmentMaps/0/ny.jpg',
    '/textures/environmentMaps/0/pz.jpg',
    '/textures/environmentMaps/0/nz.jpg'
])
```

Ahora puede usar environmentMapTexture en la propiedad envMap de su material:

material.envMap = environmentMapTexture

Debería ver que el entorno se refleja en la superficie de la geometría. Intente modificar la metalidad y la rugosidad para obtener diferentes resultados.

También puede probar otros mapas de entorno en la carpeta / static / textures / environmentMap /.

Dónde encontrar mapas ambientales

Para encontrar mapas de entorno interesantes, siempre puede hacer una búsqueda simple en la web y asegurarse de tener derecho a usar el mapa de entorno si no es para uso personal.

Una de las mejores fuentes es HDRIHaven. Este sitio web tiene cientos de increíbles HDRI. HDRI son las siglas de High Dynamic Range Imaging. Se componen de una imagen (no un mapa de cubos) y contienen más datos que una simple imagen, mejorando así la información de iluminación para un resultado más realista. Las imágenes HDRIHaven son gratuitas y están bajo licencia CC0, lo que significa que puede hacer lo que quiera con ellas sin tener que dar crédito a los autores. Pero si aprecias su trabajo, puedes agradecerles suscribiéndote a su Patreon.

Pero tenemos un problema. Como dijimos, Three.js solo admite mapas de cubos. Para convertir un HDRI en un mapa de cubos, puede utilizar esta herramienta en línea: <https://matheowis.github.io/HDRI-to-CubeMap/>

Cargue un HDRI, gírelo como desee y descargue una versión de mapa de cubos compuesta por 6 imágenes. El formato predeterminado es .png, y tendrá que convertirlos a .jpg si lo desea.

13. Texto 3D

Introducción

Ya conocemos suficientes conceptos básicos para crear contenido. Para nuestro primer proyecto, recrearemos lo que hizo ilithya con su genial portafolio <https://www.ilithya.rocks/> y tendremos un gran texto en 3D en el medio de la escena con objetos flotando alrededor.

Este portafolio es un excelente ejemplo de lo que puede hacer bastante temprano cuando aprende Three.js. Es simple, eficiente y se ve genial.

Three.js ya admite geometrías de texto 3D con la clase TextGeometry. El problema es que debe especificar una fuente, y esta fuente debe estar en un formato json particular llamado tipografía.

No hablaremos de licencias, pero debe tener derecho a usar la fuente a menos que sea para uso personal.

Cómo obtener una fuente tipográfica

Hay muchas formas de obtener fuentes en ese formato. Primero, puede convertir su fuente con convertidores como este: <https://gero3.github.io/facetype.js/>. Debe proporcionar un archivo y hacer clic en el botón convertir.

También puede encontrar fuentes en los ejemplos de Three.js ubicados en la carpeta / node_modules / three / examples / fonts /. Puede tomar esas fuentes y ponerlas en la carpeta / static /, o puede importarlas directamente en su archivo JavaScript porque son json y los archivos .json son compatibles al igual que los archivos .js en Webpack:

```
import typefaceFont from 'three/examples/fonts/helvetiker_regular.typeface.json'  
Mezclaremos esas dos técnicas abriendo / node_modules / three / examples / fonts /, tomando los archivos helvetiker_regular.typeface.json y LICENSE, y colocándolos en la carpeta / static / fonts / (que necesita crear).
```

Ahora se puede acceder a la fuente simplemente escribiendo /fonts/helvetiker_regular.typeface.json al final de la URL base.

Cargar la fuente

Para cargar la fuente, debemos usar una nueva clase de cargador llamada FontLoader. Este cargador funciona igual que el TextureLoader. Agregue el siguiente código después de la parte de textureLoader (si está usando otra fuente, no olvide cambiar la ruta):



```
/**  
 * Fonts  
 */  
  
const fontLoader = new THREE.FontLoader()  
  
fontLoader.load(  
    '/fonts/helvetiker_regular.typeface.json',  
    (font) =>  
    {  
        console.log('loaded')  
    }  
)
```

Debería "cargarse" en su consola. Si no es así, consulte los pasos anteriores y busque posibles errores en la consola.

Ahora tenemos acceso a la fuente usando la variable de fuente dentro de la función. A diferencia de TextureLoader, tenemos que escribir el resto de nuestro código dentro de esa función de éxito.

Crea la geometría

Como dijimos anteriormente, usaremos TextGeometry. Tenga cuidado con el código de ejemplo en la página de documentación; los valores son mucho más grandes que los de nuestra escena.

Asegúrese de escribir su código dentro de la función de éxito:

```
fontLoader.load(
  '/fonts/helvetiker_regular.typeface.json',
  (font) =>
{
  const textGeometry = new THREE.TextGeometry(
    'Hello Three.js',
    {
      font: font,
      size: 0.5,
      height: 0.2,
      curveSegments: 12,
      bevelEnabled: true,
      bevelThickness: 0.03,
      bevelSize: 0.02,
      bevelOffset: 0,
      bevelSegments: 5
    }
  )
  const textMaterial = new THREE.MeshBasicMaterial()
  const text = new THREE.Mesh(textGeometry, textMaterial)
  scene.add(text)
}
)
```

Debería obtener un texto blanco en 3D que necesita mejoras.

Primero, deshazte del cubo. Su propósito era asegurarse de que todo funcionara.

Si quieres ver algo genial, agrega wireframe: fiel a tu material.

```
const textMaterial = new THREE.MeshBasicMaterial({ wireframe: true })
```

Ahora puede ver cómo se genera la geometría y hay muchos triángulos. Crear una geometría de texto es largo y difícil para la computadora. Evite hacerlo demasiadas veces y mantenga la geometría lo más baja posible reduciendo las propiedades curveSegments y bevelSegments.

Elimine la estructura alámbrica una vez que esté satisfecho con el nivel de detalles.

Centrar el texto

Hay varias formas de centrar el texto. Una forma de hacerlo es mediante delimitación. El límite es la información asociada con la geometría que indica qué espacio ocupa esa geometría. Puede ser una caja o una esfera.

En realidad, no puede ver esos límites, pero ayuda a Three.js a calcular fácilmente si el objeto está en la pantalla y, de lo contrario, el objeto ni siquiera se renderizará. Eso se llama sacrificio frustum, pero no es el tema de esta lección.

Lo que queremos es usar este límite para conocer el tamaño de la geometría y volver a centrarlo. De forma predeterminada, Three.js usa el límite de esfera. Lo que queremos es un cuadro delimitador, para ser más precisos. Para hacerlo, podemos pedirle a Three.js que calcule el límite de este cuadro llamando a `computeBoundingBox()` en la geometría:

```
textGeometry.computeBoundingBox()
```

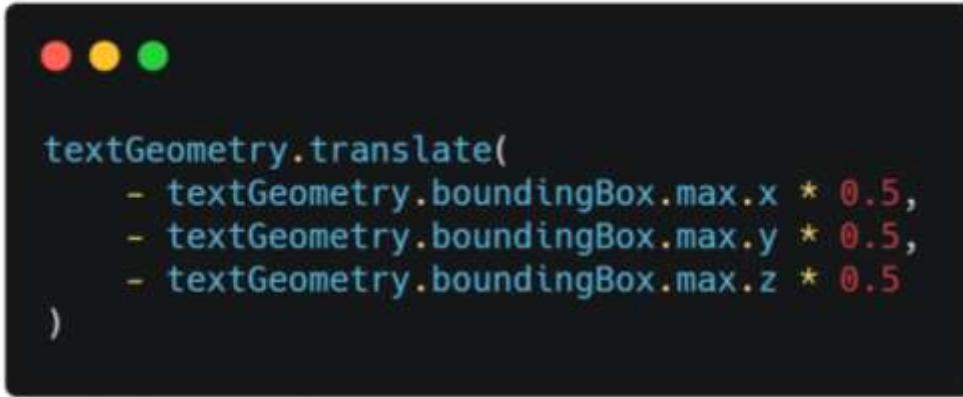
Y podemos marcar esta casilla con la propiedad `boundingBox` en la geometría.

```
console.log(textGeometry.boundingBox)
```

El resultado es un objeto llamado Box3 que tiene una propiedad mínima y una propiedad máxima. La propiedad mínima no está en 0 como podríamos haber esperado. Eso se debe a bevelThickness y bevelSize, pero podemos ignorar esto por ahora.

Ahora que tenemos las medidas, podemos mover el objeto. En lugar de mover la malla, vamos a mover toda la geometría. De esta forma, la malla seguirá estando en el centro de la escena, pero la geometría del texto también estará centrada dentro de nuestra malla.

Para hacer esto, podemos usar el método translate (...) en nuestra geometría justo después del método computeBoundingBox ():



El texto debe estar centrado, pero si desea ser muy preciso, también debe restar el tamaño de bisel que es 0.02:

```
textGeometry.translate(
  - (textGeometry.boundingBox.max.x - 0.02) * 0.5, // Subtract bevel size
  - (textGeometry.boundingBox.max.y - 0.02) * 0.5, // Subtract bevel size
  - (textGeometry.boundingBox.max.z - 0.03) * 0.5 // Subtract bevel thickness
)
```

Lo que hicimos aquí en realidad se puede hacer mucho más rápido llamando al método center () en la geometría:

```
textGeometry.center()
```

Mucho más fácil, ¿no? El objetivo de hacerlo nosotros mismos fue aprender acerca de los límites y la eliminación truncada.

Agregue un material matcap

Es hora de agregar un material interesante a nuestro texto. Vamos a utilizar un MeshMatcapMaterial porque se ve genial y tiene un gran rendimiento.

Primero, elijamos una textura matcap. Vamos a usar los matcaps ubicados en la carpeta / static / textures / matcaps / pero síntase libre de usar sus propios matcaps.

También puede descargar uno de este repositorio <https://github.com/nidox/matcaps>. ¡No pierdas mucho tiempo eligiéndolo! Si no es para uso personal, asegúrese de tener derecho a usarlo. No necesita una textura de alta resolución y 256x256 debería ser más que suficiente.

Ahora podemos cargar la textura usando TextureLoader que ya está en el código:

```
const matcapTexture = textureLoader.load('/textures/matcaps/1.png')
```

Ahora podemos reemplazar nuestro feo MeshBasicMaterial por un hermoso MeshMatcapMaterial y usar nuestra variable matcapTexture con la propiedad matcap:

```
const textMaterial = new THREE.MeshMatcapMaterial({ matcap: matcapTexture })
```

Debería tener un texto bonito con un material atractivo.

Agregar objetos

Agreguemos objetos flotando alrededor. Para hacer eso, crearemos una dona pero dentro de una función de bucle.

En la función de éxito, justo después de la parte de texto, agregue la función de bucle:

```
for(let i = 0; i < 100; i++)
```

```
{
```

```
}
```

Podríamos haber hecho esto fuera de la función de éxito, pero vamos a necesitar que el texto y los objetos se creen juntos por una buena razón que verá más adelante.

En este bucle, cree un TorusGeometry (un nombre técnico para una dona), el mismo material que para el texto y la malla:

```
for(let i = 0; i < 100; i++)
{
    const donutGeometry = new THREE.TorusGeometry(0.3, 0.2, 20, 45)
    const donutMaterial = new THREE.MeshMatcapMaterial({ matcap: matcapTexture })
    const donut = new THREE.Mesh(donutGeometry, donutMaterial)
    scene.add(donut)
}
```

Deberías conseguir 100 donas en el mismo lugar.

Agreguemos algo de aleatoriedad para su posición:

```
donut.position.x = (Math.random() - 0.5) * 10
donut.position.y = (Math.random() - 0.5) * 10
donut.position.z = (Math.random() - 0.5) * 10
```

Las rosquillas deberían haber girado en todas las direcciones.

Finalmente, podemos agregar aleatoriedad a la escala. Pero ten cuidado; necesitamos usar el mismo valor para los 3 ejes (x, y, z):

```
const scale = Math.random()
donut.scale.set(scale, scale, scale)
```

Optimizar

Nuestro código no está muy optimizado. Como vimos en una lección anterior, podemos usar el mismo material en múltiples mallas, pero también podemos usar la misma geometría.

Mueva el donutGeometry y el donutMaterial fuera del bucle:

```
const donutGeometry = new THREE.TorusGeometry(0.3, 0.2, 20, 45)
const donutMaterial = new THREE.MeshMatcapMaterial({ matcap: matcapTexture })

for(let i = 0; i < 100; i++)
{
    // ...
}
```

Debería obtener el mismo resultado, pero podemos ir aún más lejos. El material del texto es el mismo que el de la dona.

Eliminemos donutMaterial, cambiemos el nombre del textMaterial por material y usémoslo tanto para el texto como para el donut:

```
const material = new THREE.MeshMatcapMaterial({ matcap: matcapTexture })

// ...

const text = new THREE.Mesh(textGeometry, material)

// ...

for(let i = 0; i < 100; i++)
{
    const donut = new THREE.Mesh(donutGeometry, material)

    // ...
}
```

Podríamos ir aún más lejos, pero hay una lección dedicada a las optimizaciones.

Ir más lejos

Si lo desea, puede agregar más formas, animarlas e incluso probar otros matcaps.

14. Go live

CAPITULO 2

15. Luces

Introducción

Como vimos en la lección anterior, agregar luces es tan simple como agregar mallas. Crea una instancia de una luz usando la clase adecuada y la agrega a la escena.

Hay varios tipos de luces y ya descubrimos AmbientLight y PointLight.

En esta lección, veremos todas las diferentes clases en detalle y cómo usarlas.

Configuración

Una escena ya está configurada en el iniciador (completa con una esfera, un cubo, un toro y un plano como piso), pero siéntete libre de probar esto tú mismo si quieras practicar.

Debido a que vamos a usar luces, debemos usar un material que reaccione a las luces. Podríamos haber usado MeshLambertMaterial, MeshPhongMaterial o MeshToonMaterial, pero en su lugar usaremos MeshStandardMaterial porque es el más realista como vimos en la lección anterior. También redujimos la rugosidad del material a 0.4 para ver los reflejos de las luces.

Una vez que el motor de arranque esté funcionando, retire AmbientLight y PointLight para comenzar desde cero. Debería obtener un renderizado negro sin nada visible en él.

Luz ambiental

AmbientLight aplica iluminación omnidireccional en todas las geometrías de la escena. El primer parámetro es el color y el segundo parámetro es la intensidad. En cuanto a los materiales, puede establecer las propiedades directamente al crear una instancia o puede cambiarlas después:



```
const ambientLight = new THREE.AmbientLight(0xffffff, 0.5)
scene.add(ambientLight)

// Equals
const ambientLight = new THREE.AmbientLight()
ambientLight.color = new THREE.Color(0xffffff)
ambientLight.intensity = 0.5
scene.add(ambientLight)
```

Y como hicimos con los materiales, puede agregar las propiedades a la interfaz de usuario de depuración. No haremos eso en el resto de la lección, pero siéntase libre de agregar ajustes si desea facilitar la prueba:

```
gui.add(ambientLight, 'intensity').min(0).max(1).step(0.001)
```

Si todo lo que tiene es un AmbientLight, tendrá el mismo efecto que para un MeshBasicMaterial porque todas las caras de las geometrías se iluminarán por igual.

En la vida real, cuando iluminas un objeto, los lados de los objetos en el lado opuesto de la luz no serán totalmente negros porque la luz rebota en las paredes y otros objetos. El rebote de luz no es compatible con Three.js por razones de rendimiento, pero puede usar un AmbientLight tenue para simular este rebote de luz.

Luz direccional

DirectionalLight tendrá un efecto similar al del sol, como si los rayos del sol viajaran en paralelo. El primer parámetro es el color y el segundo parámetro es la intensidad:

```
● ● ●  
const directionalLight = new THREE.DirectionalLight(0x00ffff, 0.3)  
scene.add(directionalLight)
```

De forma predeterminada, la luz parece provenir de arriba. Para cambiar eso, debes mover toda la luz usando la propiedad de posición como si fuera un objeto normal.

directionalLight.position.set(1, 0.25, 0)

La distancia de la luz no importa por ahora. Los rayos provienen de un espacio infinito y viajan en paralelo al infinito opuesto.

HemisferioLuz

El HemisphereLight es similar al AmbientLight pero con un color diferente del cielo que el color que viene del suelo. Los rostros que miran al cielo se iluminarán con un color mientras que otro color iluminará los rostros que miran al suelo.

El primer parámetro es el color correspondiente al color del cielo, el segundo parámetro es el color del suelo y el tercer parámetro es la intensidad:

```
● ● ●  
const hemisphereLight = new THREE.HemisphereLight(0xffff00, 0x0000ff, 0.3)  
scene.add(hemisphereLight)
```

PointLight

El PointLight es casi como un encendedor. La fuente de luz es infinitamente pequeña y la luz se esparce uniformemente en todas direcciones. El primer parámetro es el color y el segundo parámetro es la intensidad:

```
● ● ●  
const pointLight = new THREE.PointLight(0xff9000, 0.5)  
scene.add(pointLight)
```

Lo podemos mover como cualquier objeto:

pointLight.position.set(1, - 0.5, 1)

De forma predeterminada, la intensidad de la luz no se desvanece. Pero puede controlar esa distancia de desvanecimiento y qué tan rápido se desvanece utilizando las propiedades de distancia y desvanecimiento. Puede establecerlos en los parámetros de la clase como el tercer y cuarto parámetro, o en las propiedades de la instancia:

```
const pointLight = new THREE.PointLight(0xff9000, 0.5, 10, 2)
```

RectAreaLight

RectAreaLight funciona como las grandes luces rectangulares que puedes ver en el set de fotos. Es una mezcla entre una luz direccional y una luz difusa. El primer parámetro es el color, el segundo parámetro es la intensidad, el tercer parámetro es el ancho del rectángulo y el cuarto parámetro es su altura:



```
const rectAreaLight = new THREE.RectAreaLight(0x4e00ff, 2, 1, 1)
scene.add(rectAreaLight)
```

RectAreaLight solo funciona con MeshStandardMaterial y MeshPhysicalMaterial.

Luego puede mover la luz y rotarla. Para facilitar la rotación, puede utilizar el método lookAt (...) que vimos en una lección anterior:

```
rectAreaLight.position.set(- 1.5, 0, 1.5)
```

```
rectAreaLight.lookAt(new THREE.Vector3())
```

Un Vector3 sin ningún parámetro tendrá su x, y, yz a 0 (el centro de la escena).

SpotLight

El SpotLight funciona como una linterna. Es un cono de luz que comienza en un punto y se orienta en una dirección. Aquí la lista de sus parámetros:

color: el color

intensidad: la fuerza

distancia: la distancia a la que la intensidad cae a 0

ángulo: qué tan grande es el haz

penumbra: qué tan difuso es el contorno del rayo

decaimiento: qué tan rápido se atenúa la luz



```
const spotLight = new THREE.SpotLight(0x78ff00, 0.5, 10, Math.PI * 0.1, 0.25, 1)
spotLight.position.set(0, 2, 3)
scene.add(spotLight)
```

Girar nuestro SpotLight es un poco más difícil. La instancia tiene una propiedad denominada target, que es un Object3D. El SpotLight siempre está mirando ese objeto de destino. Pero si intenta cambiar su posición, el SpotLight no se moverá:

```
spotLight.target.position.x = - 0.75
```

Eso se debe a que nuestro objetivo no está en la escena. Simplemente agregue el objetivo a la escena y debería funcionar:

```
scene.add(spotLight.target)
```

Rendimiento

Las luces son geniales y pueden ser realistas si se usan bien. El problema es que las luces pueden costar mucho cuando se trata de rendimiento. La GPU tendrá que hacer muchos cálculos como la distancia desde la cara a la luz, cuánto esa cara está mirando hacia la luz, si la cara está en el cono de luz puntual, etc.

Intente agregar la menor cantidad de luces posible y trate de usar las luces que cuestan menos.

Costo mínimo:

Luz ambiental

HemisferioLuz

Costo moderado:

Luz direccional

PointLight

Alto costo:

Destacar

RectAreaLight

Horneando

Una buena técnica de iluminación se llama hornear. La idea es que hornees la luz en la textura. Esto se puede hacer en un software 3D. Desafortunadamente, no podrá mover las luces, porque no hay ninguna y probablemente necesitará muchas texturas.

Un buen ejemplo es la página de inicio de Three.js Journey

Ayudantes

Colocar y orientar las luces es difícil. Para ayudarnos, podemos utilizar ayudantes. Solo se admiten los siguientes ayudantes:

HemisferioLuzAyudante

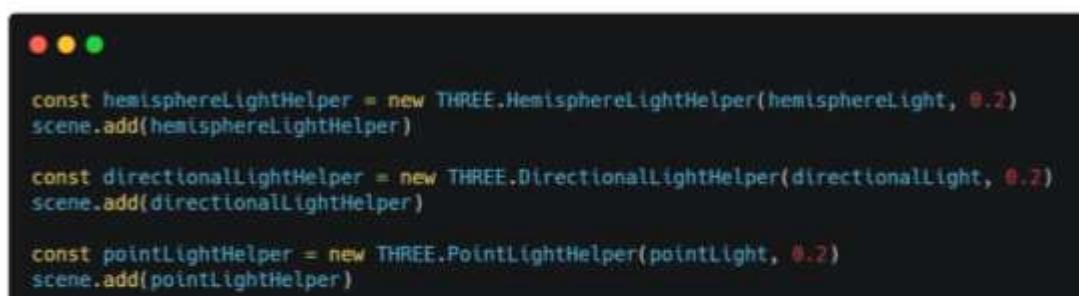
DirectionalLightHelper

PointLightHelper

RectAreaLightHelper

SpotLightHelper

Para usarlos, simplemente cree una instancia de esas clases. Utilice la luz correspondiente como parámetro y agréguelos a la escena. El segundo parámetro le permite cambiar el tamaño del ayudante:



```
const hemisphereLightHelper = new THREE.HemisphereLightHelper(hemisphereLight, 0.2)
scene.add(hemisphereLightHelper)

const directionalLightHelper = new THREE.DirectionalLightHelper(directionalLight, 0.2)
scene.add(directionalLightHelper)

const pointLightHelper = new THREE.PointLightHelper(pointLight, 0.2)
scene.add(pointLightHelper)
```

Para SpotLightHelper, no hay ningún parámetro de tamaño. Además, después de mover el objetivo, debe llamar al método update (...) pero en el siguiente cuadro:

```
const spotLightHelper = new THREE.SpotLightHelper(spotLight)
scene.add(spotLightHelper)
window.requestAnimationFrame( () =>
{
    spotLightHelper.update()
})
```

RectAreaLightHelper es aún más difícil de usar. En este momento, la clase no es parte de la variable TRES. Debe importarlo desde las dependencias de los ejemplos como hicimos con OrbitControls:

import { RectAreaLightHelper } from 'three/examples/jsm/helpers/RectAreaLightHelper.js'

Entonces puedes usarlo:

```
const rectAreaLightHelper = new RectAreaLightHelper(rectAreaLight)
```

```
scene.add(rectAreaLightHelper)
```

16. Sombras

Introducción

Ahora que tenemos luces, queremos sombras. La parte posterior de los objetos está de hecho en la oscuridad, y esto se llama la sombra del núcleo. Lo que nos falta es la sombra paralela, donde los objetos crean sombras sobre los otros objetos.

Las sombras siempre han sido un desafío para el renderizado 3D en tiempo real, y los desarrolladores deben encontrar trucos para mostrar sombras realistas a una velocidad de fotogramas razonable.

Hay muchas formas de implementarlos y Three.js tiene una solución incorporada. Tenga en cuenta que esta solución es conveniente, pero está lejos de ser perfecta.

Cómo funciona

No detallaremos cómo funcionan las sombras internamente, pero intentaremos comprender los conceptos básicos.

Cuando haces un render, Three.js primero hará un render para cada luz que se supone que proyecta sombras. Esos renders simularán lo que ve la luz como si fuera una cámara. Durante estos renders de luces, MeshDepthMaterial reemplaza todos los materiales de mallas.

Los resultados se almacenan como texturas y mapas de sombras con nombre.

No verá esos mapas de sombras directamente, pero se utilizan en todos los materiales que se supone que reciben sombras y se proyectan en la geometría.

Aquí hay un excelente ejemplo de lo que ven la luz direccional y el foco:

https://threejs.org/examples/webgl_shadowmap_viewer.html

Configuración

Nuestro motor de arranque está compuesto por una esfera simple en un plano con una luz direccional y una luz ambiental.

Puede controlar estas luces y la metalización y rugosidad del material en Dat.GUI.

Cómo activar sombras

Primero, necesitamos activar los mapas de sombras en el renderizador:

`renderer.shadowMap.enabled = true`

Luego, debemos revisar cada objeto de la escena y decidir si el objeto puede proyectar una sombra con la propiedad `castShadow` y si el objeto puede recibir sombra con la propiedad `receiveShadow`.

Intente activarlos en la menor cantidad de objetos posible:



```
sphere.castShadow = true  
  
// ...  
plane.receiveShadow = true
```

Finalmente, active las sombras en la luz con la propiedad `castShadow`.

Solo los siguientes tipos de luces admiten sombras:

PointLight

Luz direccional

Destacar

Y nuevamente, intente activar sombras en la menor cantidad de luces posible:

`directionalLight.castShadow = true`

Deberías tener una sombra de la esfera en el avión.

Lamentablemente, esa sombra se ve terrible. Intentemos mejorarlo.

Optimizaciones de mapas de sombras

Tamaño de renderizado

Como dijimos al comienzo de la lección, Three.js está haciendo renderizados llamados mapas de sombras para cada luz. Puede acceder a este mapa de sombras (y muchas otras cosas) utilizando la propiedad de sombra en la luz:

`console.log(directionalLight.shadow)`

En cuanto a nuestro render, necesitamos especificar un tamaño. De forma predeterminada, el tamaño del mapa de sombras es de solo 512 x 512 por motivos de rendimiento. Podemos mejorarlo, pero tenga en cuenta que necesita un valor de potencia 2 para el mipmapping:



```
directionalLight.shadow.mapSize.width = 1024  
directionalLight.shadow.mapSize.height = 1024
```

La sombra ya debería verse mejor.

Cerca y lejos

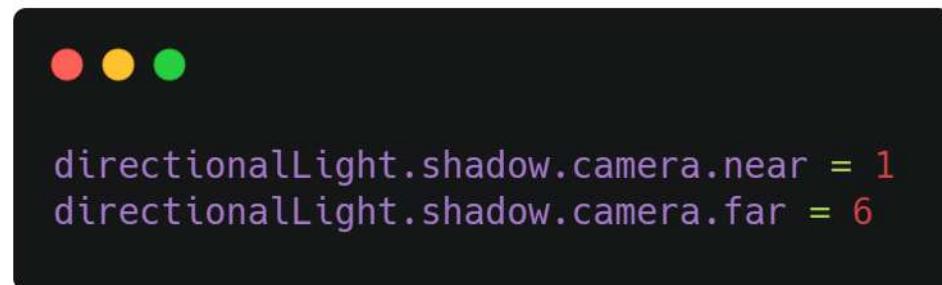
Three.js está usando cámaras para hacer los renderizados de mapas de sombras. Esas cámaras tienen las mismas propiedades que las cámaras que ya usamos. Esto significa que debemos definir un cerca y un lejos. Realmente no mejorará la calidad de la sombra, pero podría corregir errores en los que no puede ver la sombra o donde la sombra aparece recortada repentinamente.

Para ayudarnos a depurar la cámara y obtener una vista previa de cerca y de lejos, podemos usar un CameraHelper con la cámara utilizada para el mapa de sombras ubicado en la propiedad directionalLight.shadow.camera:



```
const directionalLightCameraHelper = new THREE.CameraHelper(directionalLight.shadow.camera)
scene.add(directionalLightCameraHelper)
```

Ahora puede ver visualmente el cerca y el lejos de la cámara. Intente encontrar un valor que se ajuste a la escena:

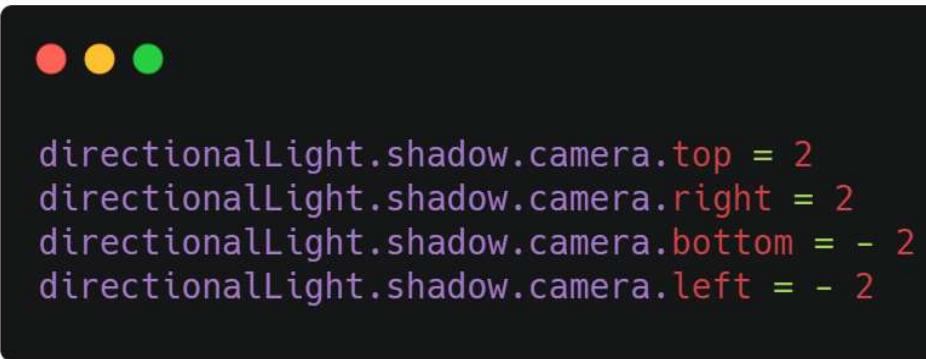


```
directionalLight.shadow.camera.near = 1
directionalLight.shadow.camera.far = 6
```

Amplitud

Con el ayudante de cámara que acabamos de agregar, podemos ver que la amplitud de la cámara es demasiado grande.

Debido a que estamos usando una DirectionalLight, Three.js está usando una OrthographicCamera. Si recuerda la lección Cámaras, podemos controlar qué tan lejos en cada lado puede ver la cámara con las propiedades superior, derecha, inferior e izquierda. Reduzcamos esas propiedades:



```
directionalLight.shadow.camera.top = 2
directionalLight.shadow.camera.right = 2
directionalLight.shadow.camera.bottom = - 2
directionalLight.shadow.camera.left = - 2
```

Cuanto menor sean los valores, más precisa será la sombra. Pero si es demasiado pequeño, las sombras simplemente se recortarán.

Puede ocultar el asistente de cámara una vez que haya terminado:

```
directionalLightCameraHelper.visible = false
```

Difuminar

Puede controlar el desenfoque de la sombra con la propiedad radius:

```
directionalLight.shadow.radius = 10
```

Esta técnica no utiliza la proximidad de la cámara con el objeto. Es solo un desenfoque general y barato.

Algoritmo de mapa de sombras

Se pueden aplicar diferentes tipos de algoritmos a los mapas de sombras:

TRES.BasicShadowMap Muy eficaz pero de mala calidad

TRES.PCFShadowMap Menos rendimiento pero bordes más suaves

THREE.PCFSOFTShadowMap Menos rendimiento pero bordes aún más suaves

THREE.VSMShadowMap Menos rendimiento, más restricciones, puede tener resultados inesperados

Para cambiarlo, actualice la propiedad renderer.shadowMap.type. El valor predeterminado es THREE.PCFShadowMap, pero puede utilizar THREE.PCFSOFTShadowMap para obtener una mejor calidad.

```
renderer.shadowMap.type = THREE.PCFSOFTShadowMap
```

Tenga en cuenta que la propiedad radius no funciona con THREE.PCFSOFTShadowMap. Tendrás que elegir.

Destacar

Intentemos agregar un SpotLight como hicimos en la lección Lights y agreguemos la propiedad castShadow a true. No olvide agregar la propiedad de destino a la escena.

También agregaremos un asistente de cámara:

```
// Destacar
const spotLight = nuevo THREE.SpotLight (0xffffffff, 0.4, 10, Math.PI * 0.3)

spotLight.castShadow = true

spotLight.position.set (0, 2, 2)
scene.add (spotLight)
scene.add (spotLight.target)

const spotlightCameraHelper = nuevo THREE.CameraHelper (spotLight.shadow.camera)
scene.add (spotlightCameraHelper)
```

Puede reducir la intensidad de otras luces si la escena es demasiado brillante:

```
const ambientLight = new THREE.AmbientLight(0xffffff, 0.4)

// ...

const directionalLight = new THREE.DirectionalLight(0xffffff, 0.4)
```

Como puede ver, las sombras no se combinan bien. Se manejan de forma independiente y, lamentablemente, no hay mucho que hacer al respecto.

Pero podemos mejorar la calidad de la sombra utilizando las mismas técnicas que usamos para la luz direccional.

Cambie el shadow.mapSize:

```
spotLight.shadow.mapSize.width = 1024  
spotLight.shadow.mapSize.height = 1024
```

Debido a que ahora estamos usando un SpotLight, internamente, Three.js está usando un PerspectiveCamera. Eso significa que en lugar de las propiedades superior, derecha, inferior e izquierda, debemos cambiar la propiedad fov. Intente encontrar un ángulo lo más pequeño posible sin tener que recortar las sombras:

```
spotLight.shadow.camera.fov = 30
```

```
spotLight.shadow.camera.near = 1  
spotLight.shadow.camera.far = 6
```

Puede ocultar el asistente de cámara una vez que haya terminado:

```
spotLightCameraHelper.visible = false
```

PointLight

Probemos la última luz que soporta sombras, la PointLight:

```
// Point Light  
const pointLight = new THREE.PointLight(0xffffff, 0.3)  
pointLight.castShadow = true  
pointLight.position.set(- 1, 1, 0)  
scene.add(pointLight)  
  
const pointLightCameraHelper = new THREE.CameraHelper(pointLight.shadow.camera)  
scene.add(pointLightCameraHelper)
```

Puede reducir la intensidad de otras luces si la escena es demasiado brillante:

```
const ambientLight = new THREE.AmbientLight(0xffffff, 0.3)  
// ...  
const directionalLight = new THREE.DirectionalLight(0xffffff, 0.3)  
// ...  
const spotLight = new THREE.SpotLight(0xffffff, 0.3, 10, Math.PI * 0.3)
```

Como puede ver, el asistente de cámara es una PerspectiveCamera (como para SpotLight) pero mirando hacia abajo. Eso se debe a cómo Three.js maneja los mapas de sombras para PointLight.

Debido a que la luz puntual se ilumina en todas las direcciones, Three.js tendrá que representar cada una de las 6 direcciones para crear un mapa de sombra de cubo. El ayudante de cámara que ves es la posición de la cámara en el último de esos 6 renders (que es hacia abajo).

Hacer todos esos renders puede generar problemas de rendimiento. Intente evitar tener demasiado PointLight con las sombras habilitadas.

Las únicas propiedades que puede modificar aquí son mapSize, near y far:

```
● ● ●

const ambientLight = new THREE.AmbientLight(0xffffff, 0.3)

// ...

const directionalLight = new THREE.DirectionalLight(0xffffff, 0.3)

// ...

const spotLight = new THREE.SpotLight(0xffffff, 0.3, 10, Math.PI * 0.3)
```

Puede ocultar el asistente de cámara una vez que haya terminado:

```
pointLightCameraHelper.visible = false
```

Hornear sombras

Las sombras de Three.js pueden ser muy útiles si la escena es simple, pero de lo contrario podría volverse desordenada.

Una buena alternativa son las sombras horneadas. Hablamos de luces horneadas en la lección anterior y es exactamente lo mismo. Las sombras se integran en las texturas que aplicamos sobre los materiales.

En lugar de comentar todas las líneas de código relacionadas con las sombras, podemos simplemente desactivarlas en el renderizador:

```
renderer.shadowMap.enabled = false
```

Ahora podemos cargar una textura de sombra ubicada en /static/textures/bakedShadow.jpg usando el clásico TextureLoader.

Agregue el siguiente código antes de crear los objetos y las luces:

```
● ● ●

/*
 * Textures
 */
const textureLoader = new THREE.TextureLoader()
const bakedShadow = textureLoader.load('/textures/bakedShadow.jpg')
```

Y finalmente, en lugar de usar un MeshStandardMaterial en el plano, usaremos un MeshBasicMaterial simple con el valor de la sombra al horno como mapa:

```
● ● ●  
const plane = new THREE.Mesh(  
    new THREE.PlaneGeometry(5, 5),  
    new THREE.MeshBasicMaterial({  
        map: bakedShadow  
    })  
)
```

Debería ver una bonita sombra falsa borrosa y realista. El principal problema es que no es dinámico, y si la esfera o las luces se mueven, las sombras no lo harán.

Alternativa para hornejar sombras

Una solución menos realista pero más dinámica sería utilizar una sombra más simple debajo de la esfera y ligeramente por encima del plano.

La textura es un halo simple. La parte blanca será visible y la parte negra será invisible.

Luego, movemos esa sombra con la esfera.

Primero, eliminemos la sombra horneada anterior volviendo a colocar el MeshStandardMaterial en el plano:

```
● ● ●  
const plane = new THREE.Mesh(  
    new THREE.PlaneGeometry(5, 5),  
    material  
)
```

Luego, podemos cargar una textura de sombra básica ubicada en /static/textures/backedShadow.jpg.

```
const simpleShadow = textureLoader.load('/textures/simpleShadow.jpg')
```

Podemos crear la sombra usando un plano simple que rotamos y colocamos ligeramente por encima del suelo. El material debe ser negro pero con la textura de la sombra como alphaMap. No olvide cambiar transparente a verdadero y agregar la malla a la escena:

```
const sphereShadow = new THREE.Mesh(  
    new THREE.PlaneGeometry(1.5, 1.5),  
    new THREE.MeshBasicMaterial({  
        color: 0x000000,  
        transparent: true,  
        alphaMap: simpleShadow  
    })  
)  
sphereShadow.rotation.x = - Math.PI * 0.5  
sphereShadow.position.y = plane.position.y + 0.01  
  
scene.add(sphere, sphereShadow, plane)
```

Ahí tienes, una sombra no tan realista pero muy eficaz.

Si va a animar la esfera, simplemente puede animar la sombra en consecuencia y cambiar su opacidad según la elevación de la esfera:

```
const clock = new THREE.Clock()  
  
const tick = () =>  
{  
    const elapsedTime = clock.getElapsedTime()  
  
    // Update the sphere  
    sphere.position.x = Math.cos(elapsedTime) * 1.5  
    sphere.position.z = Math.sin(elapsedTime) * 1.5  
    sphere.position.y = Math.abs(Math.sin(elapsedTime * 3))  
  
    // Update the shadow  
    sphereShadow.position.x = sphere.position.x  
    sphereShadow.position.z = sphere.position.z  
    sphereShadow.material.opacity = (1 - sphere.position.y) * 0.3  
  
    // ...  
}  
  
tick()
```

Que técnica usar

Encontrar la solución adecuada para manejar las sombras depende de usted. Depende del proyecto, las actuaciones y las técnicas que conozcas. También puedes combinarlos.

17. Casa encantada

18. Partículas

Introducción

Las partículas son precisamente lo que esperas de ese nombre. Son muy populares y se pueden usar para lograr varios efectos como estrellas, humo, lluvia, polvo, fuego y muchas otras cosas.

Lo bueno de las partículas es que puedes tener cientos de miles de ellas en la pantalla con una velocidad de fotogramas razonable. La desventaja es que cada partícula está compuesta por un plano (dos triángulos) siempre mirando hacia la cámara.

Crear partículas es tan simple como hacer una malla. Necesitamos un BufferGeometry, un material que pueda manejar partículas (PointsMaterial), y en lugar de producir una malla, necesitamos crear un Points.

Configuración

El motor de arranque solo está compuesto por un cubo en el medio de la escena. Ese cubo asegura que todo funcione.

Primeras partículas

Deshagámonos de nuestro cubo y creemos una esfera compuesta de partículas para empezar.

Geometría

Puede utilizar cualquiera de las geometrías básicas de Three.js. Por las mismas razones que para Mesh, es preferible usar BufferGeometries. Cada vértice de la geometría se convertirá en una partícula:

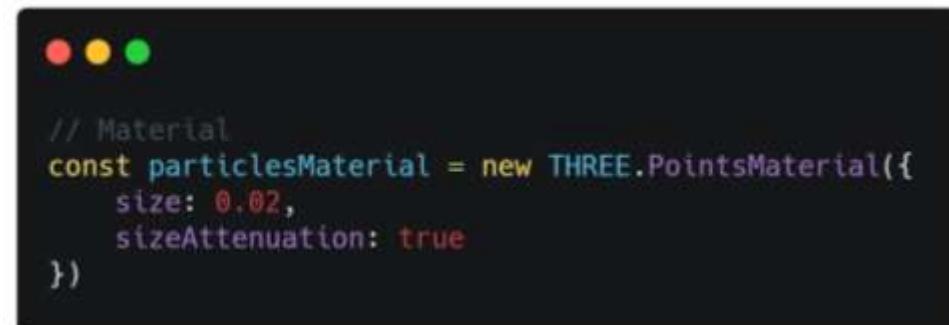


```
/*  
 * Particles  
 */  
// Geometry.  
const particlesGeometry = new THREE.SphereGeometry(1, 32, 32)
```

PointsMaterial

Necesitamos un tipo especial de material llamado PointsMaterial. Este material ya puede hacer mucho, pero descubriremos cómo crear nuestro propio material de partículas para ir aún más lejos en una lección futura.

PointsMaterial tiene múltiples propiedades específicas de las partículas, como el tamaño para controlar el tamaño de todas las partículas y la atenuación de tamaño para especificar si las partículas distantes deben ser más pequeñas que las partículas cercanas:



```
// Material  
const particlesMaterial = new THREE.PointsMaterial({  
    size: 0.02,  
    sizeAttenuation: true  
})
```

Como siempre, también podemos cambiar esas propiedades después de crear el material:

```
const particlesMaterial = new THREE.PointsMaterial()
particlesMaterial.size = 0.02
particlesMaterial.sizeAttenuation = true
```

Puntos

Finalmente, podemos crear las partículas finales de la misma manera que creamos una Malla, pero esta vez usando la clase Points. No olvides agregarlo a la escena:

```
// Points
const particles = new THREE.Points(particlesGeometry, particlesMaterial)
scene.add(particles)
```

Eso fue fácil. Personalicemos esas partículas.

Geometría personalizada

Para crear una geometría personalizada, podemos comenzar desde un BufferGeometry y agregar un atributo de posición como hicimos en la lección de Geometrías. Reemplace SphereGeometry con geometría personalizada y agregue el atributo 'posición' como lo hicimos antes:

```
// Geometry
const particlesGeometry = new THREE.BufferGeometry()
const count = 500

const positions = new Float32Array(count * 3) // Multiply by 3 because each position is composed of 3 values (x, y, z)

for(let i = 0; i < count * 3; i++) // Multiply by 3 for same reason
{
    positions[i] = (Math.random() - 0.5) * 10 // Math.random() - 0.5 to have a random value between -0.5 and +0.5
}

particlesGeometry.setAttribute('position', new THREE.BufferAttribute(positions, 3)) // Create the Three.js BufferAttribute and specify that each information is composed of 3 values
```

No se frustre si no puede extraer este código usted mismo. Es un poco complejo y las variables utilizan formatos extraños.

Deberías tener un montón de partículas por toda la escena. Ahora es un excelente momento para divertirse y probar los límites de su computadora. Pruebe 5000, 50000, 500000 tal vez. Puede tener millones de partículas y aún tener una velocidad de fotogramas razonable.

Puedes imaginar que hay límites. En una computadora inferior o un teléfono inteligente, no podrá tener una experiencia de 60 fps con millones de partículas. También vamos a agregar efectos que reducirán drásticamente la velocidad de fotogramas. Pero aún así, eso es bastante impresionante.

Por ahora, mantengamos el recuento en 5000 y cambiemos el tamaño a 0,1:

```
const count = 5000  
// ...  
particlesMaterial.size = 0.1  
// ...
```

Color, mapa y mapa alfa

Podemos cambiar el color de todas las partículas con la propiedad de color en PointsMaterial. No olvide que debe usar la clase Color si está cambiando esta propiedad después de crear una instancia del material:

```
particlesMaterial.color = new THREE.Color('#ff88cc')
```

También podemos usar la propiedad del mapa para poner una textura en esas partículas. Use TextureLoader que ya está en el código para cargar una de las texturas ubicadas en / static / textures / partículas /:

```
/*  
 * Textures  
 */  
const textureLoader = new THREE.TextureLoader()  
const particleTexture = textureLoader.load('/textures/particles/2.png')  
// ...  
particlesMaterial.map = particleTexture
```

Estas texturas son versiones redimensionadas del paquete proporcionado por Kenney y puede encontrar el paquete completo aquí: <https://www.kenney.nl/assets/particle-pack>. Pero también puedes crear el tuyo propio.

Como puede ver, la propiedad de color está cambiando el mapa, al igual que con los otros materiales. Si miras de cerca, verás que las partículas frontales ocultan las partículas traseras.

Necesitamos activar la transparencia con transparente y usar la textura en la propiedad alphaMap en lugar del mapa:

```
// particlesMaterial.map = particleTexture  
particlesMaterial.transparent = true  
particlesMaterial.alphaMap = particleTexture
```

Eso es mejor, pero aún podemos ver algunos bordes de las partículas al azar.

Esto se debe a que las partículas se dibujan en el mismo orden en que se crean, y WebGL realmente no sabe cuál está frente a la otra.

Hay varias formas de solucionar este problema.

Usando alphaTest

AlphaTest es un valor entre 0 y 1 que permite al WebGL saber cuándo no representar el píxel de acuerdo con la transparencia de ese píxel. De forma predeterminada, el valor es 0, lo que significa que el píxel se renderizará de todos modos. Si usamos un valor pequeño como 0.001, el píxel no se renderizará si el alfa es 0:

```
particlesMaterial.alphaTest = 0.001
```

Esta solución no es perfecta y si observa de cerca, aún puede ver fallas, pero ya es más satisfactorio.

Uso de depthTest

Al dibujar, WebGL prueba si lo que se dibuja está más cerca de lo que ya se dibujó. Eso se llama prueba de profundidad y se puede desactivar (puede comentar el alphaTest):

```
// particlesMaterial.alphaTest = 0.001  
particlesMaterial.depthTest = false
```

Si bien esta solución parece solucionar completamente nuestro problema, desactivar la prueba de profundidad podría crear errores si tiene otros objetos en su escena o partículas con diferentes colores. Las partículas pueden dibujarse como si estuvieran por encima del resto de la escena.

Agrega un cubo a la escena para ver que:



```
const cube = new THREE.Mesh(  
    new THREE.BoxGeometry(),  
    new THREE.MeshBasicMaterial()  
)  
scene.add(cube)
```

Uso de depthWrite

Como dijimos, WebGL está probando si lo que se dibuja está más cerca de lo que ya se dibuja. La profundidad de lo que se dibuja se almacena en lo que llamamos un búfer de profundidad. En lugar de no probar si la partícula está más cerca de lo que está en este búfer de profundidad, podemos decirle al WebGL que no escriba partículas en ese búfer de profundidad (puede comentar la prueba de profundidad):



```
// particlesMaterial.alphaTest = 0.001  
// particlesMaterial.depthTest = false  
particlesMaterial.depthWrite = false
```

En nuestro caso, esta solución solucionará el problema casi sin inconvenientes. A veces, otros objetos pueden dibujarse detrás o delante de las partículas dependiendo de muchos factores, como la transparencia, el orden en el que agregaste los objetos a tu escena, etc.

Vimos múltiples técnicas y no existe una solución perfecta. Tendrás que adaptarte y encontrar la mejor combinación según el proyecto.

Mezcla

Actualmente, WebGL dibuja los píxeles uno encima del otro.

Al cambiar la propiedad de fusión, podemos decirle al WebGL no solo que dibuje el píxel, sino que también agregue el color de ese píxel al color del píxel ya dibujado. Eso tendrá un efecto de saturación que puede verse increíble.

Para probar eso, simplemente cambie la propiedad de fusión a THREE.AdditiveBlending (mantenga la propiedad depthWrite):

```
// particlesMaterial.alphaTest = 0.001  
// particlesMaterial.depthTest = false  
particlesMaterial.depthWrite = false  
particlesMaterial.blending = THREE.AdditiveBlending
```

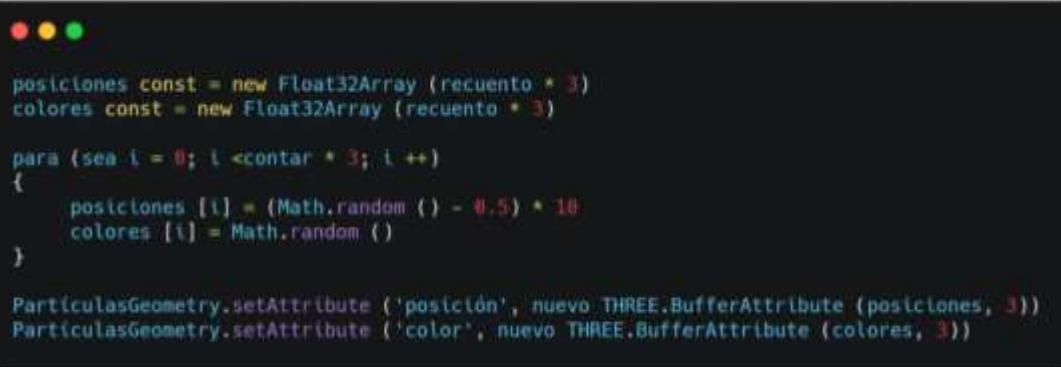
Agregue más partículas (digamos 20000) para disfrutar mejor de este efecto.

Pero tenga cuidado, este efecto afectará el rendimiento y no podrá tener tantas partículas como antes a 60 fps.

Ahora, podemos quitar el cubo.

Colores diferentes

Podemos tener un color diferente para cada partícula. Primero necesitamos agregar un nuevo atributo llamado color como hicimos para la posición. Un color se compone de rojo, verde y azul (3 valores), por lo que el código será muy similar al atributo de posición. De hecho, podemos usar el mismo bucle para estos dos atributos:



```
posiciones const = new Float32Array (recuento * 3)  
colores const = new Float32Array (recuento * 3)  
  
para (sea i = 0; i < contar * 3; i ++)  
{  
    posiciones [i] = (Math.random () - 0.5) * 10  
    colores [i] = Math.random ()  
}  
  
ParticulasGeometry.setAttribute ('posición', nuevo THREE.BufferAttribute (posiciones, 3))  
ParticulasGeometry.setAttribute ('color', nuevo THREE.BufferAttribute (colores, 3))
```

Tenga cuidado con los singulares y los plurales.

Para activar esos colores de vértice, simplemente cambie la propiedad vertexColors a true:

```
particlesMaterial.vertexColors = true
```

El color principal del material aún afecta a estos colores de vértice. Siéntete libre de cambiar ese color o incluso comentarlo.

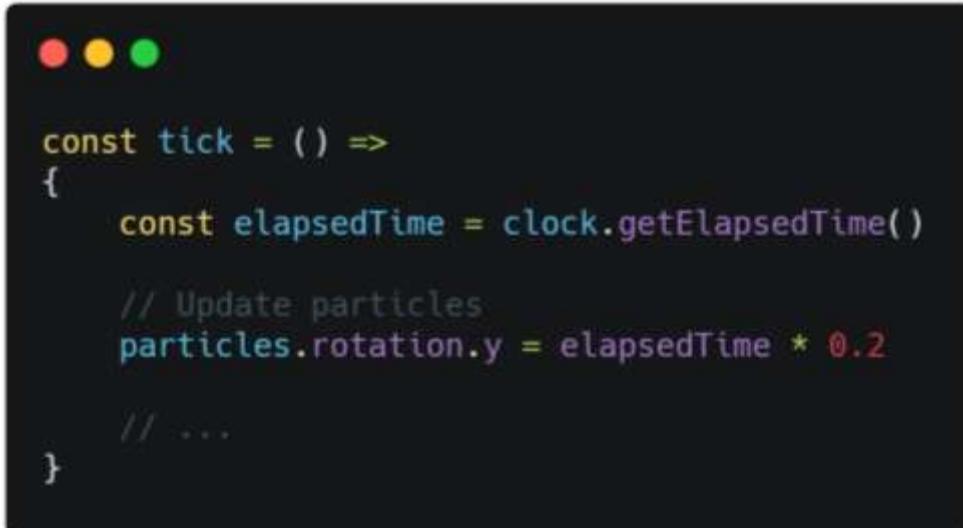
Animar

Hay varias formas de animar partículas.

Usando los puntos como un objeto

Debido a que la clase Points hereda de la clase Object3D, puede mover, rotar y escalar los puntos como desee.

Gire las partículas en la función tick:



```
const tick = () =>
{
    const elapsedTime = clock.getElapsedTime()

    // Update particles
    particles.rotation.y = elapsedTime * 0.2

    // ...
}
```

Si bien esto ya es genial, queremos más control sobre cada partícula.

Cambiando los atributos

Otra solución sería actualizar cada posición de vértice por separado. De esta forma, los vértices pueden tener diferentes trayectorias. Vamos a animar las partículas como si estuvieran flotando en ondas, pero primero veamos cómo podemos actualizar los vértices.

Comienza comentando la rotación anterior que hicimos sobre las partículas completas:

Para actualizar cada vértice, tenemos que actualizar la parte derecha en el atributo de posición porque todos los vértices se almacenan en esta matriz de una dimensión donde los primeros 3 valores corresponden a las coordenadas x, y y z del primer vértice, luego los 3 siguientes los valores corresponden a la x, y y z del segundo vértice, etc.

Solo queremos que los vértices se muevan hacia arriba y hacia abajo, lo que significa que solo actualizaremos el eje y. Debido a que el atributo de posición es una matriz de una dimensión, tenemos que revisarlo 3 por 3 y solo actualizar el segundo valor, que es la coordenada y.

Aquí, elegimos tener un bucle for simple que va de 0 a contar y creamos una variable i3 en el interior que va de 3 por 3 simplemente multiplicando i por 3.

La forma más sencilla de simular el movimiento de las ondas es utilizar un seno simple. Primero, actualizaremos todos los vértices para que suban y bajen en la misma frecuencia.

Se puede acceder a la coordenada y en la matriz en el índice i3 + 1:

```
const tick = () =>
{
    // ...

    for(let i = 0; i < count; i++)
    {
        const i3 = i * 3

        particlesGeometry.attributes.position.array[i3 + 1] = Math.sin(elapsedTime)
    }

    // ...
}
```

Desafortunadamente, nada se mueve. El problema es que se debe notificar a Three.js que la geometría cambió. Para hacer eso, tenemos que establecer `needUpdate` en verdadero en el atributo de posición una vez que hayamos terminado de actualizar los vértices:

```
const tick = () =>
{
    // ...

    for(let i = 0; i < count; i++)
    {
        const i3 = i * 3

        particlesGeometry.attributes.position.array[i3 + 1] = Math.sin(elapsedTime)
    }
    particlesGeometry.attributes.position.needsUpdate = true

    // ...
}
```

Todas las partículas deberían moverse hacia arriba y hacia abajo como un avión.

Es un buen comienzo y ya casi llegamos. Todo lo que tenemos que hacer ahora es aplicar un desplazamiento al seno entre las partículas para que obtengamos esa forma de onda.

Para hacer eso, podemos usar la coordenada x. Y para obtener este valor, podemos usar la misma técnica que usamos para la coordenada y, pero en lugar de `i3 + 1`, es solo `i3`:

```
const tick = () =>
{
    // ...

    for(let i = 0; i < count; i++)
    {
        let i3 = i * 3

        const x = particlesGeometry.attributes.position.array[i3]
        particlesGeometry.attributes.position.array[i3 + 1] = Math.sin(elapsedTime + x)
    }
    particlesGeometry.attributes.position.needsUpdate = true

    // ...
}
```

Deberías obtener hermosas ondas de partículas. Desafortunadamente, debes evitar esta técnica. Si tenemos 20000 partículas, estamos revisando cada una, calculando una nueva posición y actualizando todo el atributo en cada cuadro. Eso puede funcionar con una pequeña cantidad de partículas, pero queremos millones de partículas.

Mediante el uso de un sombreador personalizado

Para actualizar estos millones de partículas en cada fotograma con una buena velocidad de fotogramas, necesitamos crear nuestro propio material con nuestros propios sombreadores. Pero los sombreadores son para una lección posterior.

19. Generador de galaxias

Introducción

Ahora que sabemos como usar las partículas, podemos crear algo cool parecido a una galaxia. Pero en lugar de producir solo una galaxia, vamos a crear un generador de galaxias.

Para eso vamos a usar Dat.GUI para permitirle al usuario modificar los parámetros y generar una nueva galaxia en cada cambio.

Configuración

El inicio esta compuesto únicamente por un cubo en el centro de la escena. Eso asegura que todo esta funcionando correctamente.



Partículas base

Primero, remueve el cubo y crea una función “generateGalaxy”. Cada vez que llamemos a la función, vamos a remover la galaxia previa (si existe alguna) y crear una nueva.

Podemos llamar a esa función inmediatamente:

```
/**  
 * Galaxy  
 */  
const generateGalaxy = () =>  
{  
}  
  
generateGalaxy()
```

Podemos crear un objeto que contendrá todos los parámetros de nuestra galaxia. Crea este objeto antes de la función “generateGalaxy”. La completaremos progresivamente y también agregaremos cada parámetro a Dat.GUI:

```
const parameters = {}
```

En nuestra función “generateGalaxy”, vamos a crear algunas partículas solo para asegurarnos que todo esta funcionando. Podemos iniciar con la geometría y agregar el conteo de partículas a los parámetros:

```
const parameters = {}
parameters.count = 1000

const generateGalaxy = () =>
{
    /**
     * Geometry
     */
    const geometry = new THREE.BufferGeometry()

    const positions = new Float32Array(parameters.count * 3)

    for(let i = 0; i < parameters.count; i++)
    {
        const i3 = i * 3

        positions[i3] = (Math.random() - 0.5) * 3
        positions[i3 + 1] = (Math.random() - 0.5) * 3
        positions[i3 + 2] = (Math.random() - 0.5) * 3
    }

    geometry.setAttribute('position', new THREE.BufferAttribute(positions, 3))
}
generateGalaxy()
```

Ese es el mismo código como el anterior, pero estamos manejando el ciclo un poco diferente. Ahora podemos crear el material usando la clase “PointsMaterial”. Esta vez, de nuevo podemos agregar ajustes a los “parameters” del objeto:

```
parameters.size = 0.02

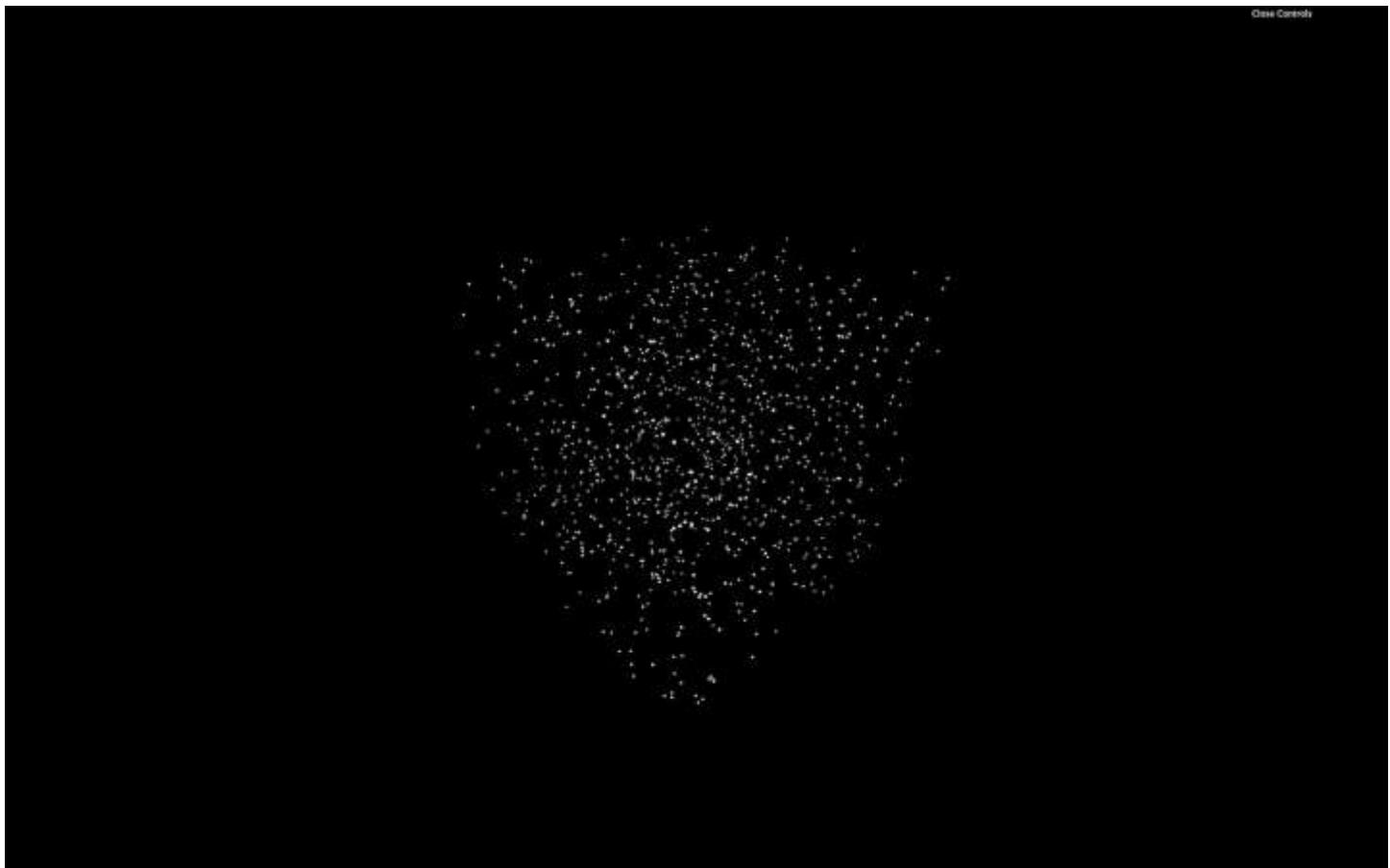
const generateGalaxy = () =>
{
    // ...

    /**
     * Material
     */
    const material = new THREE.PointsMaterial({
        size: parameters.size,
        sizeAttenuation: true,
        depthWrite: false,
        blending: THREE.AdditiveBlending
    })
}
```

Finalmente, podemos crear los puntos usando la clase “Points” y agregarlos a la escena:

```
const generateGalaxy = () =>
{
    // ...

    /**
     * Points
     */
    const points = new THREE.Points(geometry, material)
    scene.add(points)
}
```



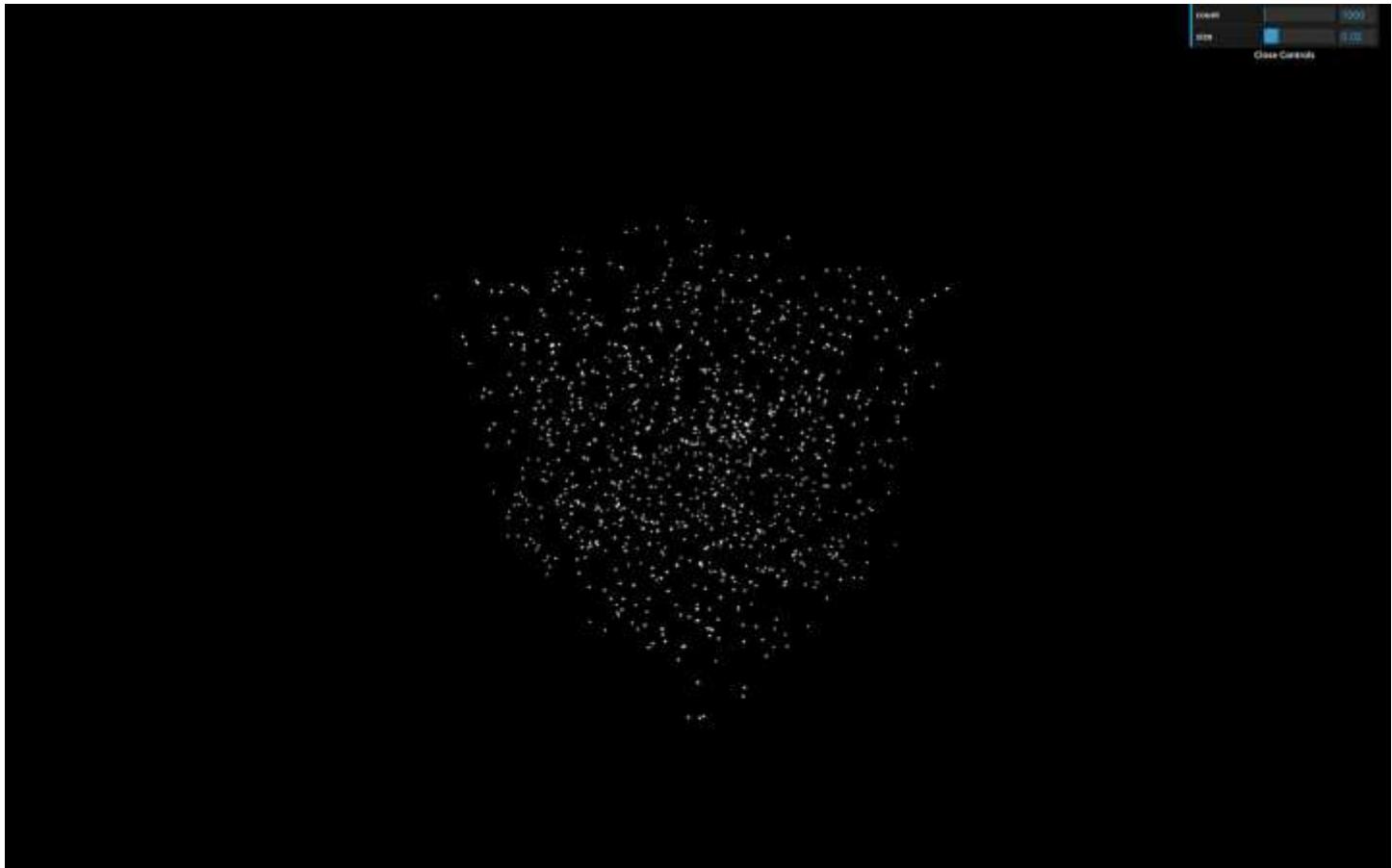
Podrías ser capaz de ver algunos puntos flotando alrededor.

Ajustes

Al momento ya tenemos dos parámetros, "count" y "size". Vamo a agregarlos a la instancia de Dat.GUI que ya tenemos creada al inicio del código. Como puedes imaginar, debemos agregar esos ajustes después de crear los parámetros:

```
parameters.count = 1000
parameters.size = 0.02

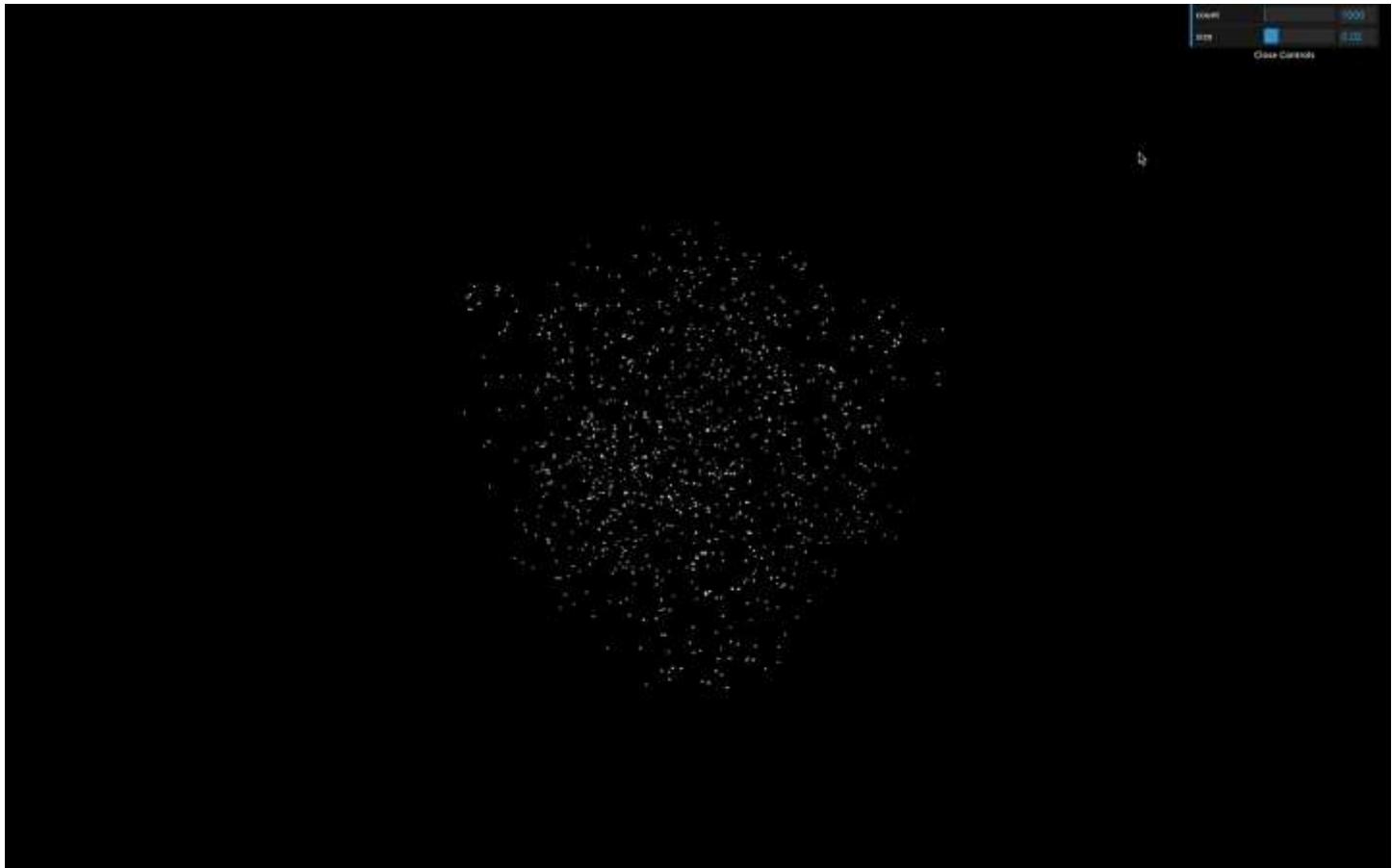
gui.add(parameters, 'count').min(100).max(1000000).step(100)
gui.add(parameters, 'size').min(0.001).max(0.1).step(0.001)
```



Deberías tener dos nuevos rangos en los ajustes pero cambiándolos no generara una galaxia nueva. Para generar una galaxia nueva, deberás llamar al evento de cambio. Mas precisamente al evento "finishChange" para prevenir generar galaxias mientras estas arrastrando y soltando el rango de valores:

```
gui.add(parameters,  
'count').min(100).max(1000000).step(100).onFinishChange(generateGalaxy)  
gui.add(parameters,  
'size').min(0.001).max(0.1).step(0.001).onFinishChange(generateGalaxy)
```

Ese código no funcionará porque la función "generateGalaxy" no existe aún. Deberías mover esos ajustes después de la función "generateGalaxy".



20. Animacion basada en scroll

CAPITULO 3

21. Fisica

Introducción

La física puede ser una de las características más interesantes que puede agregar a una experiencia WebGL. La gente disfruta jugando con objetos, los ve chocar, colapsar, caer y rebotar.

Hay muchas formas de agregar física a su proyecto, y depende de lo que desee lograr. Puede crear su propia física con algunas matemáticas y soluciones como Raycaster, pero si desea obtener una física realista con tensión, fricción, rebote, restricciones, pivotes, etc. y todo eso en el espacio 3D, es mejor que use una biblioteca.

Teoría

La idea es sencilla. Vamos a crear un mundo de física. Este mundo de la física es puramente teórico y no podemos verlo. Pero en este mundo, las cosas se caen, chocan, se frotan, se deslizan, etc.

Cuando creamos una malla Three.js, también crearemos una versión de esa malla dentro del mundo de la física. Si hacemos una caja en Three.js, también creamos una caja en el mundo de la física.

Luego, en cada fotograma, antes de renderizar nada, le decimos al mundo de la física que se actualice; tomamos las coordenadas (posición y rotación) de los objetos físicos y las aplicamos a la malla Three.js correspondiente.

Y eso es todo. Lo más difícil es organizar nuestro código en una estructura decente. Esa es una parte donde los caminos se separan. Cada desarrollador tendrá sus hábitos, y también depende de lo que quieras hacer y de lo compleja que pueda llegar a ser la física.

Para empezar, simplemente crearemos esferas y cajas.

Bibliotecas

Hay varias bibliotecas disponibles. Primero, debe decidir si necesita una biblioteca 3D o una biblioteca 2D. Si bien puede pensar que tiene que ser una biblioteca 3D porque Three.js tiene que ver con 3D, puede estar equivocado. Las bibliotecas 2D suelen ser mucho más eficaces, y si puede resumir su experiencia física hasta las colisiones 2D, es mejor que utilice una biblioteca 2D.

Un ejemplo es si desea crear un juego de billar. Las bolas pueden chocar y rebotar en las paredes, pero puedes proyectar todo en un plano 2D. Puedes diseñar bolas como círculos en el mundo de la física y las paredes son simples rectángulos. De hecho, no podrás hacer trucos golpeando la parte inferior de la pelota para que pueda saltar sobre las otras bolas.

Un excelente ejemplo de un proyecto hecho como este es Ouigo Let's play de Merci Michel. Utilizaron una biblioteca de física 2D porque cada colisión y animación se puede representar en un espacio 2D porque cada colisión y animación se puede representar en un espacio 2D.

Físicas 3D

Para la física 3D, hay tres bibliotecas principales:

Ammo.js

- Sitio web: <http://schteppe.github.io/ammo.js-demos/>
- Repositorio de Git: <https://github.com/kripken/ammo.js/>
- Documentación: Sin documentación
- Puerto JavaScript directo de Bullet (un motor de física escrito en C++)
- Un poco pesado
- Aún actualizado por una comunidad

Cannon.js

- Sitio web: <https://schteppe.github.io/cannon.js/>
- Repositorio de Git: <https://github.com/schteppe/cannon.js>
- Documentación: <http://schteppe.github.io/cannon.js/docs/>
- Más ligero que Ammo.js
- Más cómodo de implementar que Ammo.js
- Principalmente mantenido por un desarrollador
- No se ha actualizado durante muchos años.
- Hay una bifurcación mantenida

Oimo.js

- Sitio web: <https://lo-th.github.io/Oimo.js/>
- Repositorio de Git: <https://github.com/lo-th/Oimo.js>
- Documentación: <http://lo-th.github.io/Oimo.js/docs.html>
- Más ligero que Ammo.js
- Más fácil de implementar que Ammo.js
- Principalmente mantenido por un desarrollador
- No se ha actualizado durante 2 años.

Físicas 2D

Para la física 2D, hay muchas bibliotecas, pero esta es la más popular:

Matter.js

- Sitio web: <https://brm.io/matter-js/>
- Repositorio de Git: <https://github.com/liabru/matter-js>
- Documentación: <https://brm.io/matter-js/docs/>
- Principalmente mantenido por un desarrollador
- Todavía un poco actualizado

P2.js

- Sitio web: <https://schteppe.github.io/p2.js/>

- Repositorio de Git: <https://github.com/scheppe/p2.js>
- Documentación: <http://scheppe.github.io/p2.js/docs/>
- Principalmente mantenido por un desarrollador (igual que Cannon.js)
- No se ha actualizado durante 2 años.

Planck.js

- Sitio web: <https://piqnt.com/planck.js/>
- Repositorio de Git: <https://github.com/shakiba/planck.js>
- Documentación: <https://github.com/shakiba/planck.js/tree/master/docs>
- Principalmente mantenido por un desarrollador
- Todavía actualizado hoy en día.

Box2D.js

- Sitio web: <http://kripken.github.io/box2d.js/demo/webgl/box2d.html>
- Repositorio de Git: [https://github.com/kripken/box2d.js/](https://github.com/kripken/box2d.js)
- Documentación: Sin documentación
- Principalmente mantenido por un desarrollador (igual que Ammo.js)
- Todavía actualizado hoy en día

No usaremos una biblioteca 2D en esta lección, pero el código de la biblioteca 2D sería muy similar a un código de biblioteca 3D. La principal diferencia son los ejes que tienes que actualizar.

Ya existen soluciones que intentan combinar Three.js con bibliotecas como Physijs. Aún así, no usaremos ninguna de esas soluciones para obtener una mejor experiencia de aprendizaje y comprender mejor lo que está sucediendo.

Si bien Ammo.js es la biblioteca más utilizada y particularmente con Three.js, como puede ver en los ejemplos, optaremos por Cannon.js. La biblioteca es más cómoda de implementar en nuestro proyecto y más fácil de usar.

Importar Cannon.js

Para agregar Cannon.js a nuestro proyecto, primero debemos agregar la dependencia.

En su terminal, en la carpeta del proyecto, ejecute este comando `npm install --save cannon`.

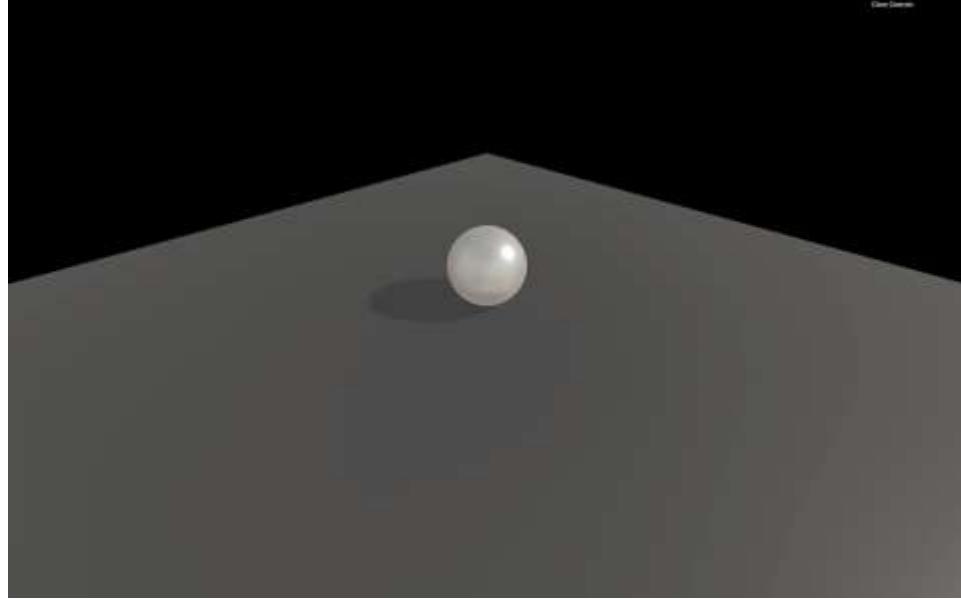
Ahora podemos importar Cannon.js en nuestro JavaScript con una importación clásica:

```
import CANNON from 'cannon'
```

Todo lo que necesitamos está disponible en la variable CANNON.

Configuración

Nuestro motor de arranque está compuesto por una esfera en un plano, y las sombras ya están habilitadas por razones estéticas.



Base Mundo

Primero, necesitamos crear un mundo Cannon.js:

```
      
  
    /**
     * Physics
     */
    const world = new CANNON.World()
```

Podríamos hacer una experiencia WebGL en el espacio donde no hay gravedad, pero mantengamos los pies en la Tierra y agreguemos gravedad. Puede cambiar este valor con la propiedad de gravedad, que es un Cannon.js Vec3.

Cannon.js Vec3 es como Three.js Vector3. Tiene propiedades x, y, z, pero también un método set (...):

```
world.gravity.set(0, - 9.82, 0)
```

Usamos - 9,82 como valor porque es la constante de gravedad en la Tierra, pero puedes usar cualquier otro valor si quieres que las cosas caigan más lentamente o si tu escena ocurre en Marte.

Objeto

Como ya tenemos una esfera en nuestra escena, creamos una esfera dentro de nuestro mundo Cannon.js.

Para hacer eso, debemos crear un Cuerpo. Los cuerpos son simplemente objetos que caerán y chocarán con otros cuerpos.

Antes de que podamos crear un cuerpo, debemos decidir una forma. Hay muchas formas primitivas disponibles como Caja, Cilindro, Plano, etc. Buscaremos una Esfera con el mismo radio que nuestra esfera Three.js:



```
const sphereShape = new CANNON.Sphere(0.5)
```

Luego podemos crear nuestro Cuerpo y especificar una masa y una posición:



```
const sphereBody = new CANNON.Body({
  mass: 1,
  position: new CANNON.Vec3(0, 3, 0),
  shape: sphereShape
})
```

Finalmente, podemos agregar el Body al mundo con addBody (...):



```
world.addBody(sphereBody)
```

No sucede nada porque todavía necesitamos actualizar nuestro mundo Cannon.js y actualizar nuestra esfera Three.js en consecuencia.

Actualiza el mundo de Cannon.js y la escena Three.js

Para actualizar nuestro mundo, debemos usar el paso (...). El código detrás de este método es difícil de entender y no lo explicaremos en esta lección, pero puede encontrar más información al respecto en este artículo.

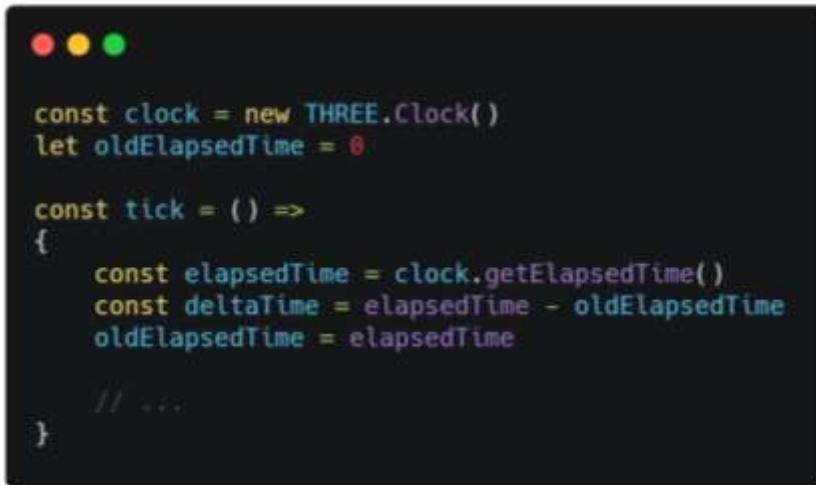
Para que funcione, debe proporcionar un paso de tiempo fijo, cuánto tiempo pasó desde el último paso y cuántas iteraciones puede aplicar el mundo para ponerse al día con un posible retraso.

No explicaremos qué es un paso de tiempo, pero como queremos que nuestra experiencia se ejecute a 60 fps, usaremos 1/60. No se preocupe, la experiencia funcionará a la misma velocidad en dispositivos con mayor y menor velocidades de cuadro.

Depende de usted el número de iteraciones, pero no es tan importante si la experiencia se desarrolla sin problemas. Usemos 3.

Para el tiempo delta, es un poco más difícil. Necesitamos calcular cuánto tiempo ha pasado desde el último fotograma. No utilice el método `getDelta()` de la clase `Clock`. No obtendrá el resultado deseado y se equivocará con la lógica interna de la clase.

Para obtener el tiempo delta correcto, debemos restar el tiempo transcurrido del fotograma anterior al tiempo transcurrido actual:

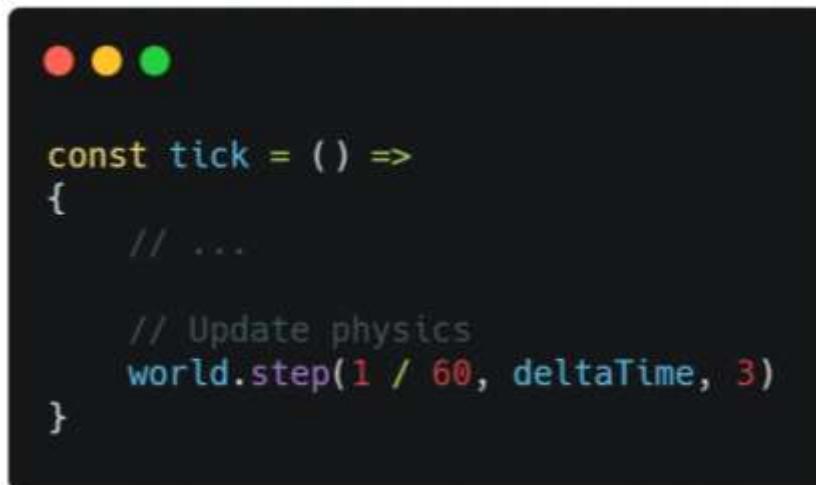


```
const clock = new THREE.Clock()
let oldElapsedTime = 0

const tick = () =>
{
    const elapsedTime = clock.getElapsedTime()
    const deltaTime = elapsedTime - oldElapsedTime
    oldElapsedTime = elapsedTime

    // ...
}
```

Finalmente podemos actualizar nuestro mundo:



```
const tick = () =>
{
    // ...

    // Update physics
    world.step(1 / 60, deltaTime, 3)
}
```

Aun así, nada parece moverse. Pero nuestro cuerpo de esfera está cayendo, y puedes verlo registrando su posición después de actualizar el mundo:



```
world.step(1 / 60, deltaTime, 3)  
console.log(sphereBody.position.y)
```

Lo que tenemos que hacer ahora es actualizar nuestra esfera Three.js usando las coordenadas del cuerpo de la esfera. Hay dos formas de hacerlo. Puede actualizar cada propiedad de posición por separado:

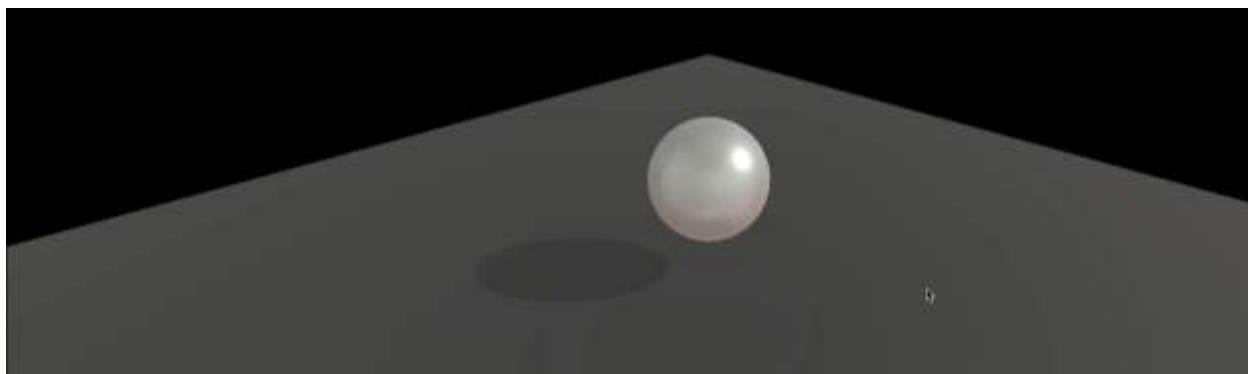


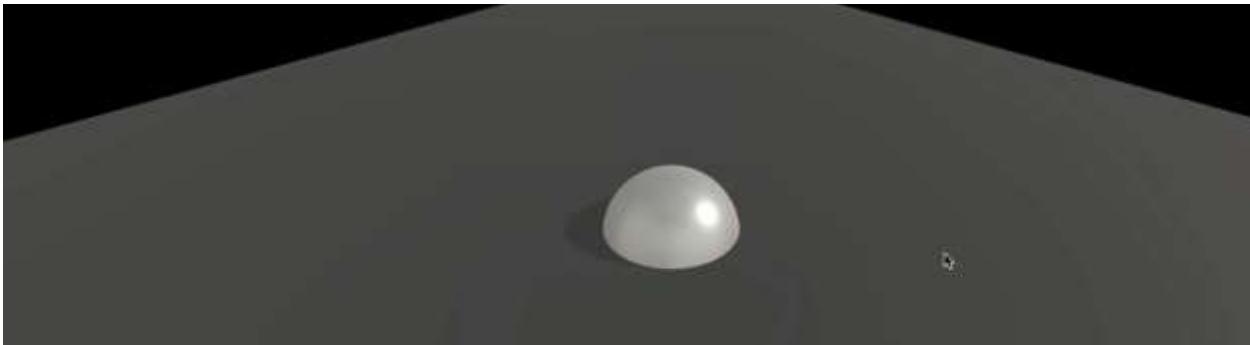
```
sphere.position.x = sphereBody.position.x  
sphere.position.y = sphereBody.position.y  
sphere.position.z = sphereBody.position.z
```

O puede copiar todas las propiedades como una con el método `copy (...)`:

```
sphere.position.copy(sphereBody.position)
```

`copy (...)` está disponible en muchas clases como `Vector2`, `Vector3`, `Euler`, `Quaternion` e incluso clases como `Material`, `Object3D`, `Geometry`, etc.





Eventualmente deberías ver caer tu esfera. El problema es que nuestra esfera parece caer por el suelo. Es porque ese piso existe en la escena Three.js pero no en el mundo Cannon.js.

Simplemente podemos agregar un nuevo cuerpo usando una forma de plano, pero no queremos que nuestro piso se vea afectado por la gravedad y la caída. En otras palabras, queremos que nuestro piso sea estático. Para hacer que un cuerpo esté estático, establezca su masa en 0:

```
const floorShape = new CANNON.Plane()
const floorBody = new CANNON.Body()
floorBody.mass = 0
floorBody.addShape(floorShape)
world.addBody(floorBody)
```

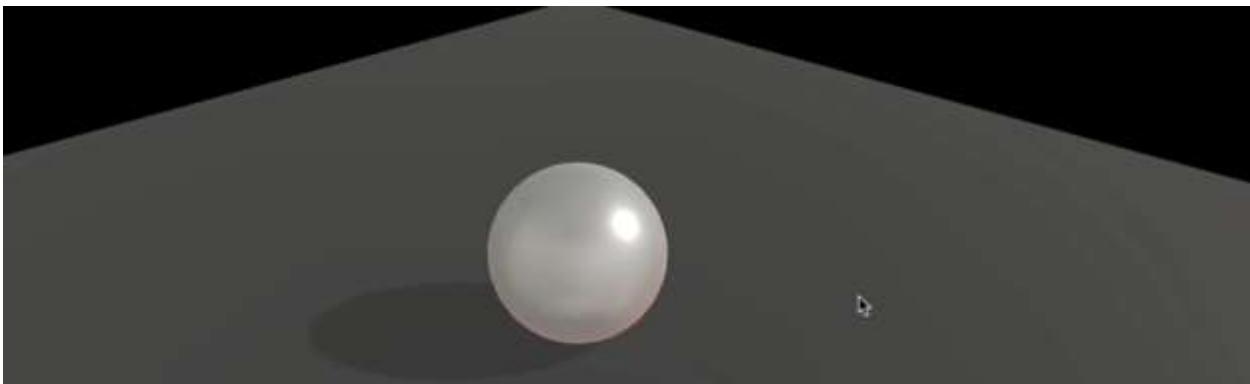
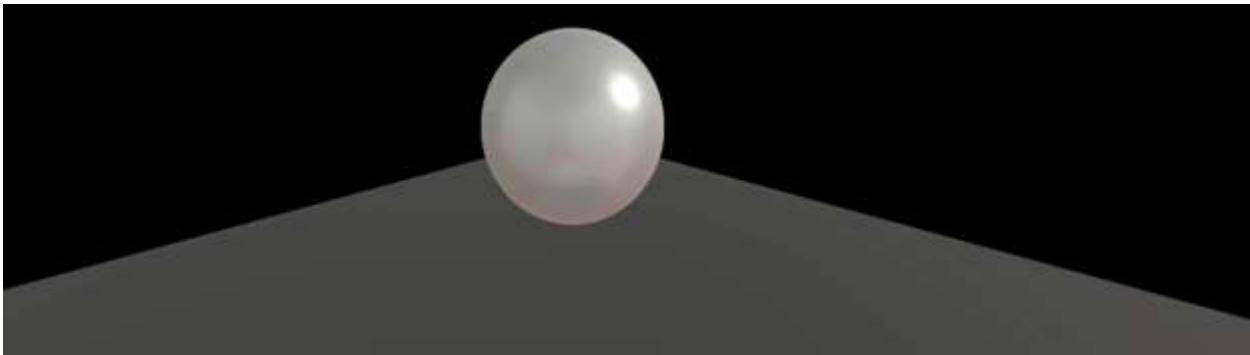
Como puede ver, esta vez hicimos las cosas de manera bastante diferente. Creamos un cuerpo sin parámetro, y luego configuramos esos parámetros. El resultado es el mismo, y la única razón por la que hicimos esto es por el bien de la lección. Algo interesante es que puede crear un cuerpo compuesto por varias formas. Puede resultar útil para objetos complejos pero sólidos.

Debería ver la esfera saltando en una dirección (probablemente hacia la cámara). No es el resultado esperado. La razón es que nuestro avión está de frente a la cámara por defecto. Necesitamos rotarlo como hicimos rotar el piso en Three.js.

La rotación con Cannon.js es un poco más difícil que con Three.js porque tienes que usar Quaternion. Hay varias formas de rotar el cuerpo, pero tiene que ser con su propiedad de cuaternión. Vamos a utilizar `setFromAxisAngle(...)`.

El primer parámetro es un eje. Puedes imaginarlo como una espiga atravesando el cuerpo. El segundo parámetro es el ángulo. Es cuánto estás rotando el cuerpo alrededor de ese pico.

```
floorBody.quaternion.setFromAxisAngle(new CANNON.Vec3(-1, 0, 0), Math.PI
* 0.5)
```



Configuramos el eje como si fuera un pico que atraviesa el cuerpo en el eje x negativo (hacia la izquierda en relación con la cámara) y configuramos el ángulo en $\text{Math.PI} * 0.5$ (un cuarto de círculo).

Ahora debería ver la esfera caer y luego detenerse en el suelo.

No necesitamos actualizar el piso Three.js con el piso Cannon.js porque este objeto no se mueve.

Material de contacto

Como puede ver, la pelota no rebota mucho. Ese es el comportamiento predeterminado, y podemos cambiarlo con Material (no el Material de Three.js) y ContactMaterial.

Un material es solo una referencia. Puedes darle un nombre y asociarlo con un Cuerpo. La idea es crear un Material para cada tipo de material que tengas en tu escena.

Suponga que todo en su mundo está hecho de plástico. En ese caso, solo tiene que crear un material y nombrarlo 'predeterminado' o 'plástico'. Si tiene varios tipos de materiales en su escena, digamos un material para el piso y otro para la pelota. Luego, debe crear varios materiales y darles nombres como 'concreto' y 'plástico'.

Podrías haberlos llamado "suelo" y "pelota". Aún así, si desea usar los mismos materiales para paredes y otros objetos como cubos, podría ser inconveniente usar un material llamado 'suelo'.

Antes de crear la esfera y el suelo, cree estos dos materiales:

```
const concreteMaterial = new CANNON.Material('concrete')
const plasticMaterial = new CANNON.Material('plastic')
```

Ahora que tenemos nuestro Material, debemos crear un *ContactMaterial*. Es la combinación de los dos materiales y contiene propiedades para cuando los objetos chocan.

Los dos primeros parámetros son los materiales. El tercer parámetro es un objeto {} que contiene dos propiedades importantes: el coeficiente de fricción (cuánto frota) y el coeficiente de restitución (cuánto rebota); ambos tienen valores predeterminados de 0.3.

Una vez creado, agregue ContactMaterial al mundo con el método addContactMaterial (...):

```
const concretePlasticContactMaterial = new CANNON.ContactMaterial(
    concreteMaterial,
    plasticMaterial,
    {
        friction: 0.1,
        restitution: 0.7
    }
)
world.addContactMaterial(concretePlasticContactMaterial)
```

No hay mucha fricción entre el concreto y el plástico, pero si dejas caer una bola de plástico sobre un piso de concreto, verás que rebota bastante.

Ahora podemos usar nuestro Material en el cuerpo. Puede pasar el Material directamente al crear una instancia del Cuerpo o después con la propiedad del material. Hagamos ambas cosas por aprender:

```
const sphereBody = new CANNON.Body({
    // ...
    material: plasticMaterial
})

// ...

const floorBody = new CANNON.Body()
floorBody.material = concreteMaterial
```

Debería ver que la pelota rebota muchas veces antes de detenerse. No podemos ver la fricción en acción porque nuestra pelota cae perfectamente recta sobre nuestro piso y pasa la mayor parte del tiempo en el aire.

Tener diferentes materiales y crear un material de contacto para cada combinación puede resultar desconcertante. Para simplificar todo, reemplazamos nuestros dos Materiales por uno predeterminado y usémoslo en todos los Cuerpos:

```
const defaultMaterial = new CANNON.Material('default')
const defaultContactMaterial = new CANNON.ContactMaterial(
    defaultMaterial,
    defaultMaterial,
    {
        friction: 0.1,
        restitution: 0.7
    }
)
world.addContactMaterial(defaultContactMaterial)

// ...

const sphereBody = new CANNON.Body({
    // ...
    material: defaultMaterial
})

// ...

floorBody.material = defaultMaterial
```

Obtenemos el mismo resultado.

Podemos ir aún más lejos configurando nuestro material como el predeterminado de nuestro mundo. Para hacer eso, simplemente asigne el defaultContactMaterial a la propiedad defaultContactMaterial del mundo:

```
world.defaultContactMaterial = defaultContactMaterial
```

Ahora podemos eliminar o comentar la asignación de material de SphereBody y floorBody.

Aplicar fuerzas

Hay muchas formas de aplicar fuerzas a un cuerpo:

applyForce para aplicar una fuerza al cuerpo desde un punto específico en el espacio (no necesariamente en la superficie del cuerpo) como el viento que empuja todo un poco todo el tiempo, un empujón pequeño pero repentino en un dominó o una fuerza repentina mayor para hacer un pájaro enojado saltar hacia el castillo enemigo.

applyImpulse es como applyForce pero en lugar de agregar a la fuerza que resultará en cambios de velocidad, se aplica directamente a la velocidad.

`applyLocalForce` es lo mismo que `applyForce` pero las coordenadas son locales para el cuerpo (lo que significa que 0, 0, 0 sería el centro del cuerpo).

`applyLocalImpulse` es lo mismo que `applyImpulse` pero las coordenadas son locales del cuerpo.

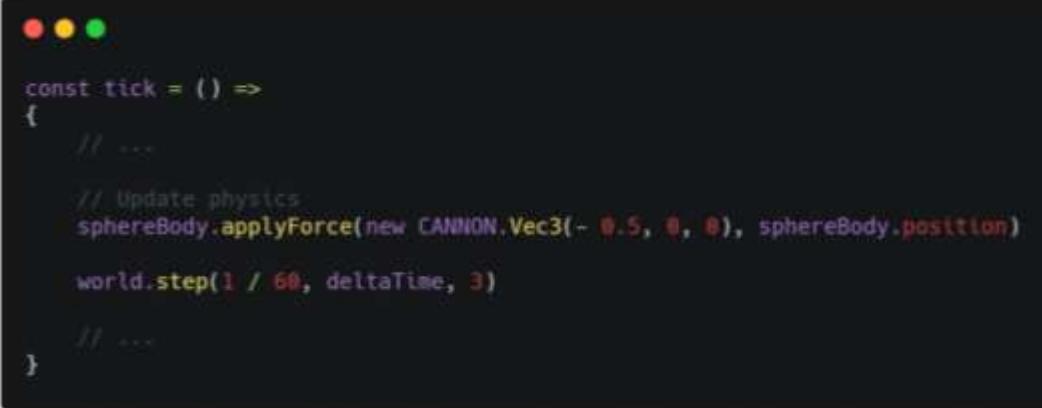
Debido a que el uso de métodos de "fuerza" dará como resultado cambios de velocidad, no usemos métodos de "impulso".

Usemos `applyLocalForce (...)` para aplicar un pequeño impulso en nuestra `sphereBody` al principio:

```
sphereBody.applyLocalForce(new CANNON.Vec3(150, 0, 0), new CANNON.Vec3(0, 0, 0))
```

Puedes ver la pelota rebotar hacia la derecha y rodar.

Ahora usemos `applyForce (...)` para aplicar algo de viento. Debido a que el viento es permanente, debemos aplicar esta fuerza a cada cuadro antes de actualizar el mundo. Para aplicar correctamente esta fuerza, el punto debe ser la esfera `Cuerpo.posición`:



```
const tick = () =>
{
    // ...
    // Update physics
    sphereBody.applyForce(new CANNON.Vec3(- 0.5, 0, 0), sphereBody.position)
    world.step(1 / 60, deltaTime, 3)
    // ...
}
```

Manejar múltiples objetos

Manejar uno o dos objetos es fácil, pero manejar docenas de objetos puede ser un desastre. Necesitamos automatizar un poco las cosas.

Primero, elimine o comente la esfera, la forma de esfera y el cuerpo de esfera.

Automatizar con funciones

Para empezar, mejoraremos la forma en que creamos esferas con una función que agregará las versiones Three.js y Cannon.js.

Como parámetros de esta función, solo pasaremos el radio y la posición, pero siéntase libre de agregar otros parámetros como masa, material, subdivisiones, etc.

```
/**  
 * Utils  
 */  
const createSphere = (radius, position) =>  
{  
}
```

Ahora podemos crear la malla Three.js:

```
const createSphere = (radius, position) =>  
{  
    // Three.js mesh  
    const mesh = new THREE.Mesh(  
        new THREE.SphereGeometry(radius, 20, 20),  
        new THREE.MeshStandardMaterial({  
            metalness: 0.3,  
            roughness: 0.4,  
            envMap: environmentMapTexture  
        })  
    )  
    mesh.castShadow = true  
    mesh.position.copy(position)  
    scene.add(mesh)  
}
```

Y el cuerpo de Cannon.js:

```
const createSphere = (radius, position) =>
{
    // ...

    // Cannon.js body
    const shape = new CANNON.Sphere(radius)

    const body = new CANNON.Body({
        mass: 1,
        position: new CANNON.Vec3(0, 3, 0),
        shape: shape,
        material: defaultMaterial
    })
    body.position.copy(position)
    world.addBody(body)
}
```

Podemos eliminar la esfera creada anteriormente y llamar a `createSphere (...)` (después de crear el mundo Cannon.js y la escena Three.js). No olvide eliminar la actualización de la esfera en la función `tick ()`:

```
createSphere(0.5, { x: 0, y: 3, z: 0 })
```

Como puede ver, la posición no tiene que ser Three.js Vector3 o Cannon.js Vec3 y simplemente podemos usar un objeto con propiedades x, y z (afortunadamente para nosotros).

Debería ver la esfera flotando sobre el suelo, pero desafortunadamente, ya no se mueve. Y esto es perfectamente normal porque eliminamos el código que estaba tomando la posición del cuerpo de Cannon.js para aplicarlo a la posición de malla de Three.js.

Usa una variedad de objetos

Para manejar esta parte, crearemos una matriz de todos los objetos que deben actualizarse. Luego agregaremos la malla y el cuerpo recién creados dentro de un objeto a esa matriz:

```
const objectsToUpdate = []

const createSphere = (radius, position) =>
{
    // ...

    // Save in objects to update
    objectsToUpdate.push({
        mesh: mesh,
        body: body
    })
}
```

Puede escribir esta última parte así (no es necesario especificar la propiedad cuando el nombre de la variable es el mismo en JavaScript):

```
objectsToUpdate.push({ mesh, body })
```

Ahora podemos recorrer esa matriz dentro de la función tick () (justo después de actualizar el mundo) y copiar cada body.position a mesh.position:

```
const tick = () =>
{
    // ...

    world.step(1 / 60, deltaTime, 3)

    for(const object of objectsToUpdate)
    {
        object.mesh.position.copy(object.body.position)
    }
}
```

La esfera debería empezar a caer de nuevo.

Agregar a Dat.GUI

Divirtámonos y agreguemos un botón `createSphere` a nuestro Dat.GUI. El problema es que el primer parámetro cuando se usa el método `gui.add (...)` debe ser un objeto y el segundo parámetro debe ser un nombre de propiedad. Desafortunadamente, nuestro método `createSphere` no está en un objeto y también necesita pasárle parámetros. Este tipo de situación puede ocurrir con regularidad. Una solución no tan mala sería crear un objeto cuyo único propósito sería tener esas funciones perdidas como propiedades:

```
const gui = new dat.GUI()
const debugObject = {}
```

Y luego agregue funciones cuando sea necesario (después de crear la función `createSphere`):

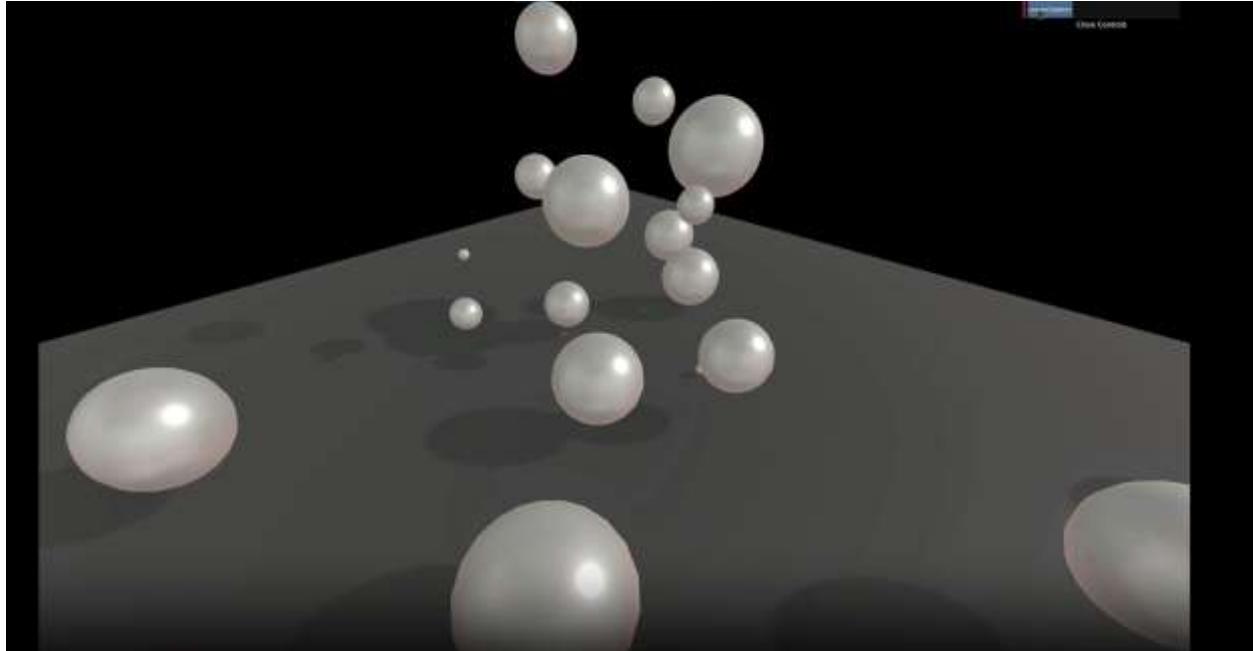
```
debugObject.createSphere = () =>
{
  createSphere(0.5, { x: 0, y: 3, z: 0 })
}
```

Finalmente, podemos agregar esta nueva propiedad `createSphere` a Dat.GUI:

```
gui.add(debugObject, 'createSphere')
```

Si hace clic en el botón `createSphere` recién creado, debería ver esferas apuntándose una sobre la otra. Eso se debe a que la esfera aparece exactamente en la misma posición. Agreguemos algo de aleatoriedad:

```
debugObject.createSphere = () =>
{
  createSphere(
    Math.random() * 0.5,
    {
      x: (Math.random() - 0.5) * 3,
      y: 3,
      z: (Math.random() - 0.5) * 3
    }
  )
}
```



¡Está lloviendo esferas!

Trate de no quemar su computadora; este código necesita optimización.

Optimizar

Debido a que la geometría y el material de la malla Three.js son los mismos, deberíamos sacarlos de la función createSphere. El problema es que estamos usando el radio para crear nuestra geometría. Una solución fácil sería fijar el radio de SphereGeometry en 1 y luego escalar la malla:

```
const sphereGeometry = new THREE.SphereGeometry(1, 20, 20)
const sphereMaterial = new THREE.MeshStandardMaterial({
    metalness: 0.3,
    roughness: 0.4,
    envMap: environmentMapTexture
})
const createSphere = (radius, position) =>
{
    // Three.js mesh
    const mesh = new THREE.Mesh(sphereGeometry, sphereMaterial)
    mesh.castShadow = true
    mesh.scale.set(radius, radius, radius)
    mesh.position.copy(position)
    scene.add(mesh)

    // ...
}
```

Debería obtener el mismo resultado.

Agregar cajas

Ahora que nuestras esferas están funcionando bien, hagamos el mismo proceso pero para las cajas.

Para crear una caja, debemos usar una BoxGeometry y una forma de Caja. Ten cuidado; los parámetros no son los mismos. Un BoxGeometry necesita un ancho, un alto y una profundidad. Mientras tanto, una forma de Caja necesita la mitad de Extensiones. Está representado por un Vec3 correspondiente a un segmento que comienza en el centro de la caja y une una de las esquinas de esa caja:

```

// Create box
const boxGeometry = new THREE.BoxGeometry(1, 1, 1)
const boxMaterial = new THREE.MeshStandardMaterial({
  metalness: 0.3,
  roughness: 0.4,
  envMap: environmentMapTexture
})
const createBox = (width, height, depth, position) =>
{
  // Three.js mesh
  const mesh = new THREE.Mesh(boxGeometry, boxMaterial)
  mesh.scale.set(width, height, depth)
  mesh.castShadow = true
  mesh.position.copy(position)
  scene.add(mesh)

  // Cannon.js body
  const shape = new CANNON.Box(new CANNON.Vec3(width * 0.5, height * 0.5, depth * 0.5))

  const body = new CANNON.Body({
    mass: 1,
    position: new CANNON.Vec3(0, 3, 0),
    shape: shape,
    material: defaultMaterial
  })
  body.position.copy(position)
  world.addBody(body)

  // Save in objects
  objectsToUpdate.push({ mesh, body })
}

createBox(1, 1.5, 2, { x: 0, y: 3, z: 0 })

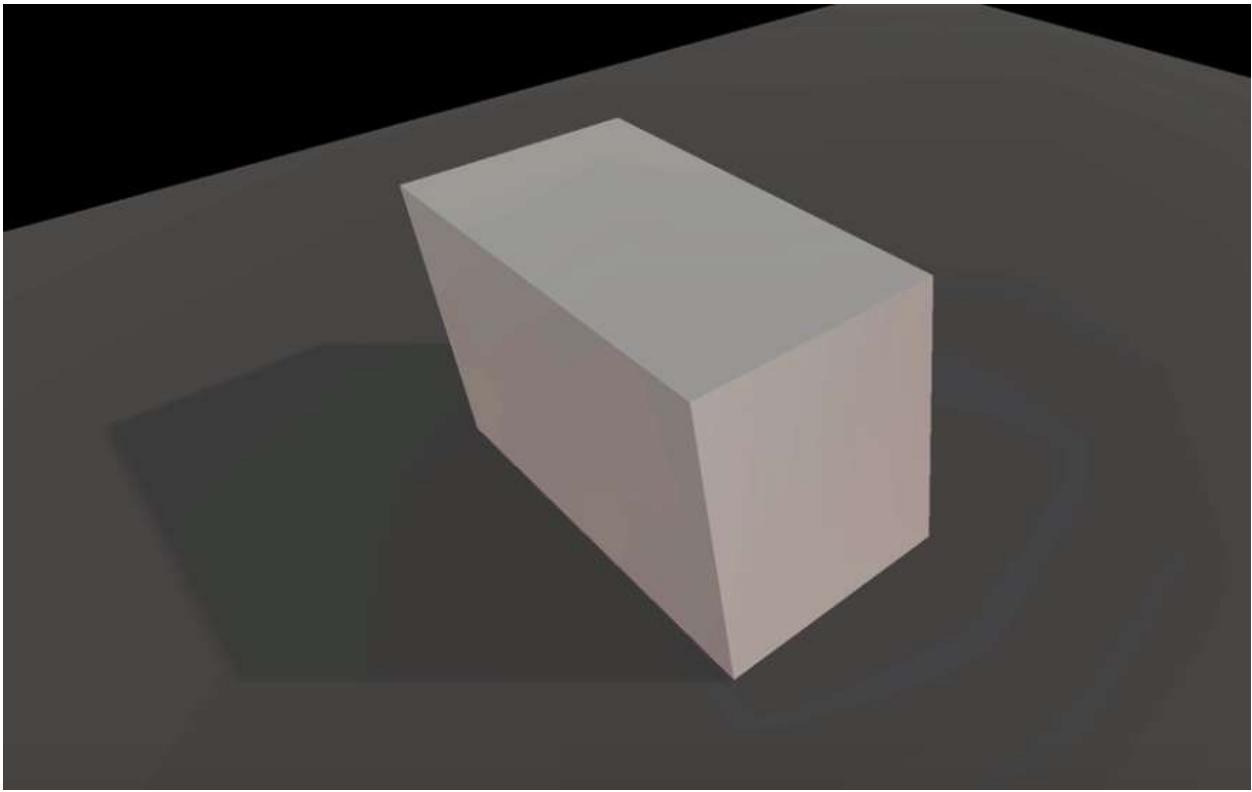
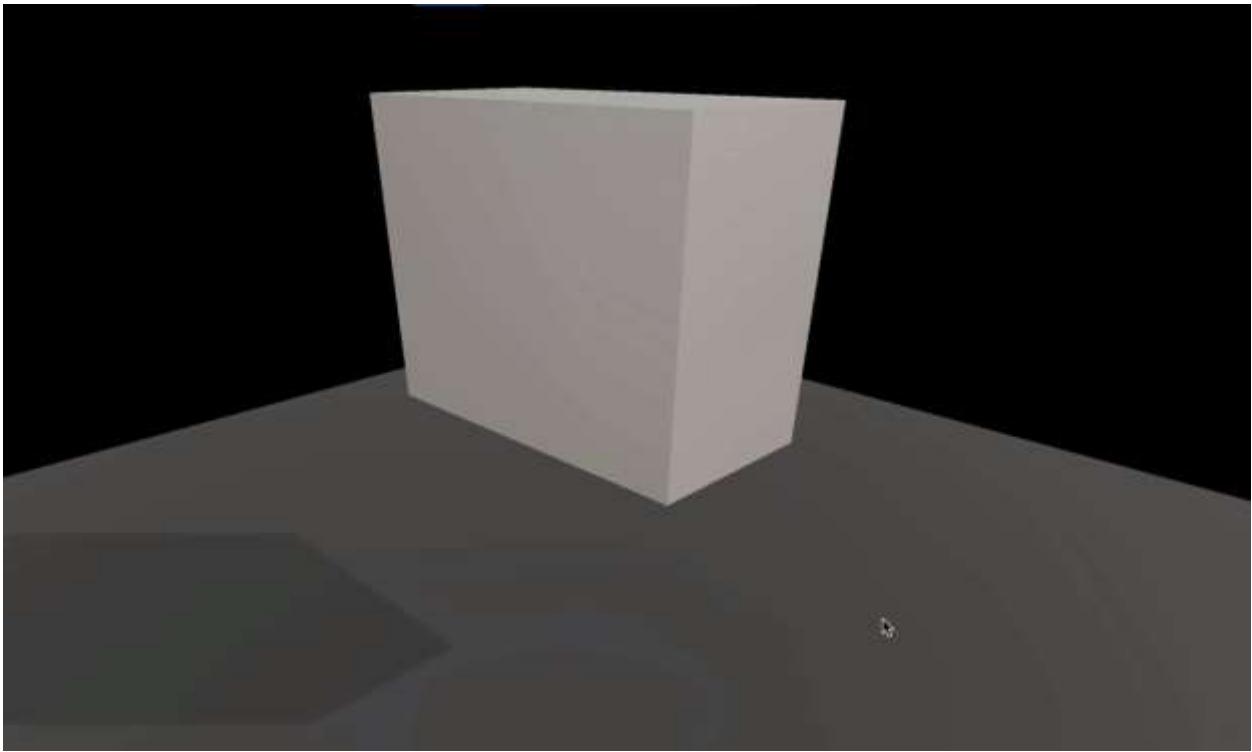
debugObject.createBox = () =>
{
  createBox(
    Math.random(),
    Math.random(),
    Math.random(),
    {
      x: (Math.random() - 0.5) * 3,
      y: 3,
      z: (Math.random() - 0.5) * 3
    }
  )
}
gui.add(debugObject, 'createBox')

```

No olvide eliminar la primera llamada a `createSphere (...)`, o tendrá la esfera y la caja creadas en la misma posición simultáneamente, lo que podría complicarse.

Debería ver una caja caer y repentinamente atravesar el piso. Si hace clic en el botón `createBox` de `Dat.GUI`, debería ser aún más agudo.

Olvidamos una cosa importante: nuestras mallas no giran. Lo que sucede aquí es que la caja rebota en el suelo y cae de lado. Pero todo lo que podemos ver es la caja erguida y atravesando el piso porque la malla `Three.js` no gira como lo hace el cuerpo `Cannon.js`.



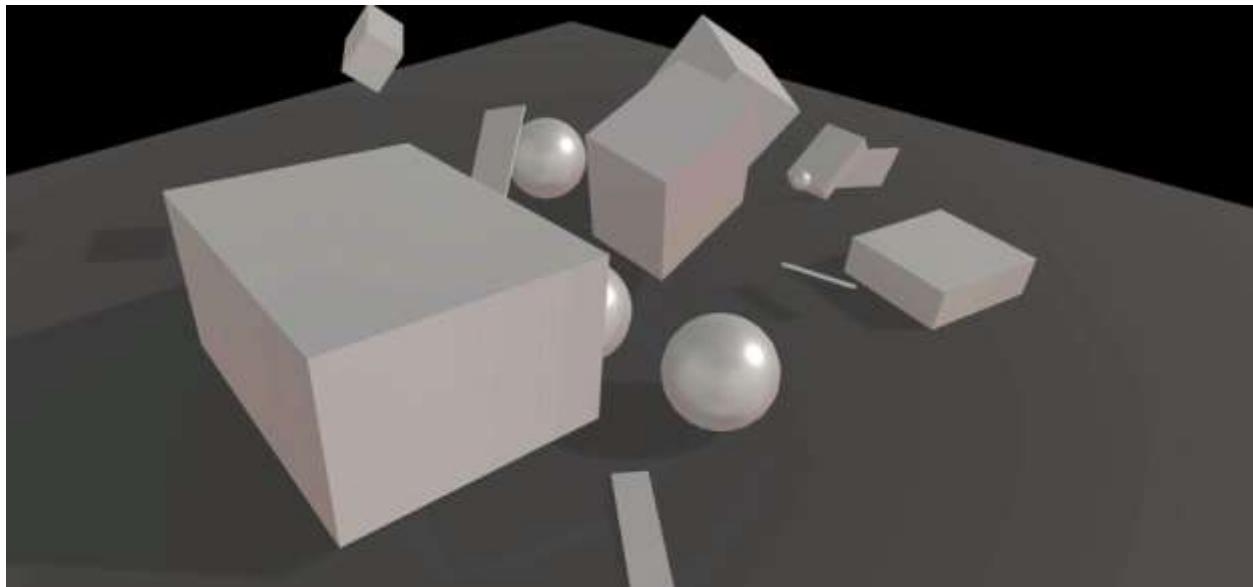
No vimos el problema antes porque estábamos usando esferas y se veían igual si las rotamos o no.

Podemos solucionar este problema copiando el cuaternión del cuerpo en el cuaternión de la malla tal como hicimos con la posición:

```
const tick = () =>
{
    // ...

    for(const object of objectsToUpdate)
    {
        object.mesh.position.copy(object.body.position)
        object.mesh.quaternion.copy(object.body.quaternion)
    }

    // ...
}
```



Las cajas deberían caer adecuadamente ahora. Puedes crear esferas y cajas como quieras. Como siempre, trate de no quemar su computadora.

Rendimiento

Broadphase

Al probar las colisiones entre objetos, un enfoque ingenuo es probar cada cuerpo contra todos los demás cuerpos. Si bien esto es fácil de hacer, es costoso en términos de rendimiento.

Ahí es donde surge broadphase. La fase ancha está haciendo una clasificación aproximada de los cuerpos antes de probarlos. Imagínese tener dos pilas de cajas alejadas una de la otra. ¿Por qué probarías las cajas de una pila con las cajas de la otra pila? Están demasiado lejos para chocar.

Hay 3 algoritmos de fase ancha disponibles en Cannon.js:

NaiveBroadphase: prueba todos los cuerpos contra todos los demás cuerpos

GridBroadphase: cuadra el mundo y solo prueba los cuerpos contra otros cuerpos en el mismo cuadro de cuadrícula o los cuadros de cuadrícula de los vecinos.

SAPBroadphase (Sweep and podar broadphase): prueba cuerpos en ejes arbitrarios durante múltiples pasos.

La fase ancha predeterminada es NaiveBroadphase y le recomiendo que cambie a SAPBroadphase. El uso de esta fase ancha puede generar errores en los que no se produce una colisión, pero es raro e implica hacer cosas como mover cuerpos muy rápido.

Para cambiar a SAPBroadphase, simplemente cree una instancia en la propiedad world.broadphase y también use este mismo mundo como parámetro:

```
world.broadphase = new CANNON.SAPBroadphase(world)
```

```
world.broadphase = new CANNON.SAPBroadphase(world)
```

Dormir

Incluso si usamos un algoritmo de fase ancha mejorado, se prueban todos los cuerpos, incluso los que ya no se mueven. Podemos usar una función llamada dormir.

Cuando la velocidad del cuerpo se vuelve increíblemente lenta (en un punto en el que no puede verlo moverse), el cuerpo puede quedarse dormido y no se probará a menos que se le aplique una fuerza suficiente por código o si otro cuerpo lo golpea.

Para activar esta función, simplemente establezca la propiedad allowSleep en true en el mundo:

```
world.allowSleep = true
```

También puede controlar la probabilidad de que el cuerpo se duerma con las propiedades sleepSpeedLimit y sleepTimeLimit, pero no las cambiaremos.

Eventos

Puede escuchar eventos en el cuerpo. Eso puede ser útil si desea hacer cosas como reproducir un sonido cuando los objetos chocan o si desea saber si un proyectil ha tocado a un enemigo.

Puede escuchar eventos en Body como 'collide', 'sleep' o 'wakeup'.

Reproducimos un sonido de golpe cuando nuestras esferas y cajas chocan con algo. Primero, cree ese sonido en JavaScript nativo y cree una función que debería reproducir el sonido.

Algunos navegadores como Chrome impiden que se reproduzcan sonidos a menos que el usuario haya interactuado con la página, como hacer clic en cualquier lugar, así que no se preocupe si no escucha los primeros sonidos.

```
/**  
 * Sounds  
 */  
const hitSound = new Audio('/sounds/hit.mp3')  
  
const playHitSound = () =>  
{  
    hitSound.play()  
}
```

Un poco exagerado reproducir un sonido, pero agregaremos más a esa función más adelante.

Ahora, escuchemos el evento 'collide' en nuestros cuerpos. Solo nos centraremos en la función `createBox` y la agregaremos a la función `createSphere` una vez que hayamos terminado.

Ahora, escuche el evento `collide` y use la función `playHitSound` como devolución de llamada:

```
const createBox = (width, height, depth, position) =>  
{  
    // ...  
  
    body.addEventListener('collide', playHitSound)  
  
    // ...
}
```

Debería escuchar el sonido del golpe cuando el cubo toca el suelo o cuando los cubos chocan. No olvide hacer clic en la página antes de que el cuadro llegue al suelo si está utilizando Chrome porque Chrome se niega a reproducir sonidos si aún no se ha producido ninguna interacción con el usuario.

El sonido parece bastante bueno. Desafortunadamente, las cosas se vuelven realmente extrañas cuando agregamos múltiples cajas.

El primer problema es que cuando llamamos a `hitSound.play()` mientras se reproduce el sonido, no sucede nada porque ya se está reproduciendo. Podemos solucionarlo restableciendo el sonido a 0 con la propiedad `currentTime`:

```
const playHitSound = () =>
{
    hitSound.currentTime = 0
    hitSound.play()
}
```

Si bien esto es mejor al principio, escuchamos demasiados sonidos de golpe incluso cuando un cubo toca ligeramente a otro. Necesitamos saber qué tan fuerte fue el impacto y no jugar nada si no fue lo suficientemente fuerte.

Para obtener la fuerza del impacto, primero necesitamos obtener información sobre la colisión. Podemos hacer eso agregando un parámetro a la devolución de llamada 'collide' (que es nuestra función playHitSound):



```
const playHitSound = (collision) =>
{
    console.log(collision)

    // ...
}
```

La variable de colisión ahora contiene mucha información. La fuerza del impacto se puede encontrar llamando al método `getImpactVelocityAlongNormal()` en la propiedad de contacto:

```
const playHitSound = (collision) =>
{
    console.log(collision.contact.getImpactVelocityAlongNormal())

    // ...
}
```

Si observa los registros, debería ver un número. Cuanto más fuerte sea el impacto, mayor será el número.

Probamos ese valor y solo reproducimos el sonido del impacto La fuerza es lo suficientemente fuerte:

```
const playHitSound = (collision) =>
{
    const impactStrength = collision.contact.getImpactVelocityAlongNormal()

    if(impactStrength > 1.5)
    {
        hitSound.currentTime = 0
        hitSound.play()
    }
}
```

Para aún más realismo, podemos agregar algo de aleatoriedad al volumen del sonido:

```
const playHitSound = (collision) =>
{
    const impactStrength = collision.contact.getImpactVelocityAlongNormal()

    if(impactStrength > 1.5)
    {
        hitSound.volume = Math.random()
        hitSound.currentTime = 0
        hitSound.play()
    }
}
```

Si quisiéramos ir aún más lejos, podríamos tener múltiples sonidos de golpe ligeramente diferentes. Y para evitar que se reproduzcan demasiados sonidos simultáneamente, podríamos agregar un retardo muy corto en el que el sonido no se pueda reproducir nuevamente después de reproducirse una vez.

No los haremos en esta lección, pero siéntase libre de probar cosas.

Copiamos el código que usamos en la función `createBox` a la función `createSphere`:

```
const createSphere = (radius, position) =>
{
```

```
// ...
```

```
    body.addEventListener('collide', playHitSound)
```

```
// ...
```

Los sonidos también deberían funcionar para las esferas.

```
// Reset
debugObject.reset = () =>
{
    console.log('reset')
}
gui.add(debugObject, 'reset')
```

Ahora, recorramos cada objeto dentro de nuestra matriz objectsToUpdate. Luego retire tanto el object.body del mundo como el object.mesh de la escena. Además, no olvide eliminar eventListener como lo hubiera hecho en JavaScript nativo:

```
debugObject.reset = () =>
{
    for(const object of objectsToUpdate)
    {
        // Remove body
        object.body.removeEventListener('collide', playHitSound)
        world.removeBody(object.body)

        // Remove mesh
        scene.remove(object.mesh)
    }
}
```

Quitar cosas

Agreguemos un botón de reinicio.

Cree una función de reinicio y agréguela a su Dat.GUI como hicimos para createBox y createSphere: Y eso es. Puede hacer clic en el botón de reinicio para eliminar todo.

Vaya más allá con Cannon.js

Si bien cubrimos los conceptos básicos y ya puede hacer muchas cosas, aquí hay algunas áreas de mejora.

Restricciones

Las restricciones, como sugiere el nombre, habilitan restricciones entre dos cuerpos. No los cubriremos en esta lección, pero aquí está la lista de restricciones:

HingeConstraint: actúa como la bisagra de una puerta.

DistanceConstraint: obliga a los cuerpos a mantener una distancia entre ellos.

LockConstraint: fusiona los cuerpos como si fueran una sola pieza.

PointToPointConstraint: pega los cuerpos a un punto específico.

Clases, métodos, propiedades y eventos

Hay muchas clases y cada una con diferentes métodos, propiedades y eventos. Intente navegar por todos ellos al menos una vez para saber si existen. Puede que le ahorre algo de tiempo en sus proyectos futuros.

Ejemplos de

La documentación no es perfecta. Sería útil si pasara algún tiempo en las demostraciones e investigando para descubrir cómo hacer las cosas. Es probable que muchas personas hayan tenido los problemas que pueden surgir. No dude en confiar en la comunidad.

Trabajadores

Ejecutar la simulación física lleva tiempo. El componente de su computadora que hace el trabajo es la CPU. Cuando ejecuta Three.js, Cannon.js, la lógica de su código, etc., todo lo hace el mismo hilo en su CPU. Ese hilo puede sobrecargarse rápidamente si hay demasiado que hacer (como demasiados objetos en la simulación física), lo que da como resultado una caída de la velocidad de fotogramas.

La solución correcta es utilizar trabajadores. Los trabajadores le permiten poner una parte de su código en un hilo diferente para distribuir la carga. A continuación, puede enviar y recibir datos de ese código. Puede resultar en una mejora considerable del rendimiento.

El problema es que el código debe estar claramente separado. Puede encontrar un buen y sencillo ejemplo aquí en el código fuente de la página.

Cannon-es

Como dijimos anteriormente, Cannon.js no se ha actualizado durante años. Afortunadamente, algunos chicos bifurcaron el repositorio y comenzaron a trabajar en las actualizaciones. Gracias a ellos, tenemos acceso a una versión mejor y mantenida de Cannon.js:

Repositorio de Git: <https://github.com/pmndrs/cannon-es>

Página de NPM: <https://www.npmjs.com/package/cannon-es>

Para usar esta versión en lugar de la original, abra la terminal en la carpeta del proyecto (o apague el servidor), elimine la dependencia anterior de cannon.js con npm uninstall --save cannon, instale la nueva versión con npm install --save cannon -es@0.15.1, y cambie la forma en que importa Cannon.js en el código:

```
import * as CANNON from 'cannon-es'
```

Todo debería funcionar como antes. Puede consultar los cambios en la página del repositorio de Git.

Ammo.js

Usamos Cannon.js porque la biblioteca es fácil de implementar y comprender. Uno de sus mayores competidores es Ammo.js. Si bien es más difícil de usar e implementar en su proyecto, aquí hay algunas características que pueden ser de su interés:

Es un portage de Bullet, un motor de física bien conocido y bien engrasado escrito en C++.

Tiene soporte para WebAssembly (wasm). WebAssembly es un lenguaje de bajo nivel compatible con los navegadores más recientes. Debido a que es de bajo nivel, tiene un mejor rendimiento.

Es más popular y puede encontrar más ejemplos de Three.js.

Admite más funciones.

Si necesita el mejor rendimiento o tiene características particulares en su proyecto, probablemente debería optar por Ammo.js en lugar de Cannon.js.

Physijs

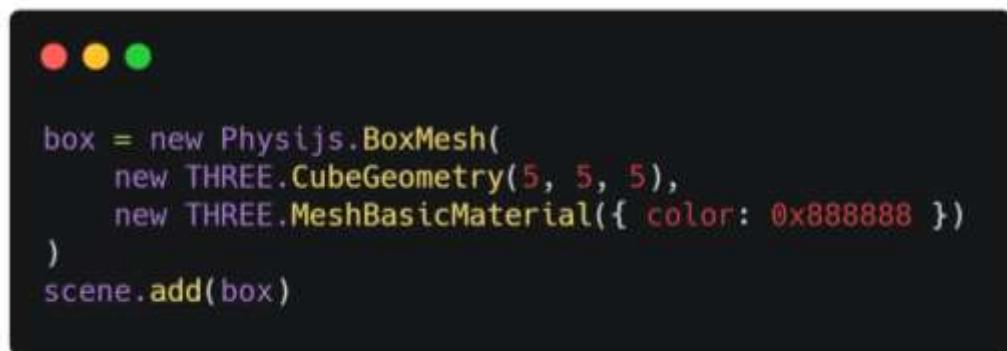
Physijs facilita la implementación de la física en un proyecto Three.js. Utiliza Ammo.js y es compatible con los trabajadores de forma nativa.

Sitio web: <https://chandlerprall.github.io/Physijs/>

Repositorio de Git: <https://github.com/chandlerprall/Physijs>

Documentación: <https://github.com/chandlerprall/Physijs/wiki>

En lugar de crear el objeto Three.js y el objeto físico, crea ambos simultáneamente:



```
box = new Physijs.BoxMesh(
    new THREE.CubeGeometry(5, 5, 5),
    new THREE.MeshBasicMaterial({ color: 0x888888 })
)
scene.add(box)
```

Physijs se encargará del resto.

Si bien es fascinante, especialmente para los principiantes, las cosas se complican cuando intentas hacer algo que no es compatible con la biblioteca. Encontrar de dónde viene un error también puede ser complicado.

Al igual que para Ammo.js, tómate tu tiempo y piensa cuál es la mejor solución para tu proyecto.

22. Modelos importados

Introducción

Three.js te permite crear muchas geometrías primitivas, pero cuando se trata de formas más complejas, es mejor que usemos un software 3D dedicado.

En esta lección, usaremos modelos ya hechos, pero aprenderemos cómo crear un modelo completamente dentro de un software 3D en una lección futura.

Formatos

A lo largo del tiempo, se han utilizado muchos formatos de modelos 3D. Cada uno respondió a un problema, como qué datos están incrustados en el modelo, el peso, su compresión, compatibilidad, derechos de autor, etc.

Por eso, hoy tenemos acceso a cientos de formatos de modelos:
https://en.wikipedia.org/wiki/List_of_file_formats#3D_graphics.

Algunos formatos están dedicados a un software. Se sabe que algunos son muy ligeros, pero a veces carecen de datos específicos. Se sabe que algunos tienen casi todos los datos que podría necesitar, pero son pesados. Algunos formatos son de código abierto, algunos formatos no lo son, algunos son binarios, algunos son ASCII, y sigue y sigue.

Si necesita datos precisos y no puede encontrar el formato adecuado compatible con su software, incluso puede crear el suyo con bastante facilidad.

Aquí hay una lista de formatos populares que puede encontrar:

OBJ
FBX
STL
CAPA
COLLADA
3DS
GLTF

No cubriremos todos estos formatos. Sería aburrido y no es necesario porque un formato se está convirtiendo en un estándar y debería cubrir la mayoría de sus necesidades.

GLTF

GLTF son las siglas de GL Transmission Format. Está hecho por Khronos Group (los chicos detrás de OpenGL, WebGL, Vulkan, Collada y con muchos miembros como AMD / ATI, Nvidia, Apple, id Software, Google, Nintendo, etc.)

GLTF se ha vuelto muy popular en los últimos años.

Admite conjuntos de datos muy diferentes. Obviamente, puede tener datos como las geometrías y los materiales, pero también puede tener datos como cámaras, luces, gráficos de escena, animaciones, esqueletos, morphing e incluso múltiples escenas.

También admite varios formatos de archivo como json, binario, texturas incrustadas.

GLTF se ha convertido en el estándar cuando se trata de tiempo real. Y debido a que se está convirtiendo en un estándar, la mayoría de los softwares 3D, motores de juegos y bibliotecas lo admiten. Eso significa que puede obtener fácilmente un resultado similar en diferentes entornos.

Eso no significa que deba usar GLTF en todos los casos. Si solo necesita una geometría, es mejor que use otro formato como OBJ, FBX, STL o PLY. Debe probar diferentes formatos en cada proyecto para ver si tiene todos los datos que necesita, si el archivo no es demasiado pesado, cuánto tiempo lleva descomprimir la información si está comprimida, etc.

Encuentra un modelo

Primero, necesitamos un modelo. Como dijimos anteriormente, aprenderemos cómo crear nuestro propio modelo en un software 3D más adelante, pero por ahora, usemos uno prefabricado.

El equipo de GLTF también proporciona varios modelos, desde un simple triángulo hasta modelos realistas y cosas como animaciones, transformaciones, materiales de capa transparente, etc.

Puede encontrarlos en este repositorio: <https://github.com/KhronosGroup/glTF-Sample-Models>

Si desea probar esos modelos, tendrá que descargar o clonar todo el repositorio y tomar los archivos que necesita. Pero comenzaremos con un pato simple que ya puede encontrar en la carpeta / static / models / en el iniciador.

Formatos GLTF

Si bien GLTF es un formato en sí mismo, también puede tener diferentes formatos de archivo. Es un poco complejo, pero por buenas razones.

Si abre la carpeta / static / models / Duck /, verá 4 carpetas diferentes. Cada uno contiene el pato, pero en diferentes formatos GLTF:

glTF
glTF-binario
glTF-Draco
glTF incrustado

Incluso puede encontrar otros formatos, pero esos 4 son los más importantes y cubren lo que necesitamos aprender.

Ten cuidado; su sistema operativo puede ocultar la extensión de algunos de estos archivos. Consulte los nombres de archivo de su editor de código que deberían mostrar la extensión.

glTF

Este formato es una especie de formato predeterminado. El archivo Duck.gltf es un JSON que puede abrir en su editor. Contiene información diversa como cámaras, luces, escenas, materiales, transformaciones de objetos, pero ni las geometrías ni las texturas. El archivo Duck0.bin es un binario que no se puede leer así. Por lo general, contiene datos como las geometrías y toda la información asociada con los vértices como coordenadas UV, normales, colores de vértices, etc. DuckCM.png es simplemente la textura del pato.

Cuando cargamos este formato, solo cargamos el Duck.gltf que contiene referencias a los otros archivos que luego se cargarán automáticamente.

glTF-binario

Este formato está compuesto por un solo archivo. Contiene todos los datos de los que hablamos en el formato predeterminado glTF. Ese es un archivo binario, y no puede simplemente abrirlo en su editor de código para ver qué hay dentro.

Este formato puede ser un poco más liviano y más cómodo de cargar porque solo hay un archivo, pero no podrá modificar fácilmente sus datos. Por ejemplo, si desea cambiar el tamaño o comprimir las texturas, simplemente no puede porque está dentro de ese archivo binario, fusionar con el resto.

glTF-Draco

Este formato es como el formato predeterminado de glTF, pero los datos del búfer (normalmente la geometría) se comprimen mediante el algoritmo de Draco. Si compara el tamaño del archivo .bin, verá que es mucho más ligero.

Si bien hay una carpeta separada para este formato, puede aplicar la compresión Draco a los otros formatos.

Dejemos esto a un lado, y hablaremos más de eso más adelante.

glTF incrustado

Este formato es como el formato glTF-Binary porque es solo un archivo, pero este archivo es en realidad un JSON que puede abrir en su editor.

El único beneficio de este formato es tener un solo archivo fácilmente editable.

Elegir

Elegir el formato correcto depende de cómo quiera manejar los activos.

Si desea poder alterar las texturas o las coordenadas de las luces después de exportar, es mejor que elija el glTF predeterminado. También presenta la ventaja de cargar los diferentes archivos por separado, lo que resulta en una mejora de la velocidad de carga.

Si solo desea un archivo por modelo y no le importa modificar los activos, es mejor que elija glTF-Binary.

En ambos casos, debe decidir si desea utilizar la compresión Draco o no, pero cubriremos esta parte más adelante.

Configuración

El motor de arranque se compone de un avión vacío.

Debido a que GLTF es un estándar, claramente admite luces. Por lo general, cuando importa un GLTF en su proyecto Three.js, terminará con Meshes que tienen MeshStandardMaterial y, como probablemente recuerde, si no tiene luces en su escena, no verá muchas de ellas. materiales.

Ya hay AmbientLight y DirectionalLight en el motor de arranque.

Cargue el modelo en Three.js

Para cargar archivos GLTF en Three.js, debemos usar GLTFLoader. Esta clase no está disponible de forma predeterminada en la variable TRES. Necesitamos importarlo desde la carpeta examples / ubicada en las tres dependencias:

```
import { GLTFLoader } from 'three/examples/jsm/loaders/GLTFLoader.js'
```

luego podemos instanciarlo como hicimos para TextureLoader:

```
const gltfLoader = new GLTFLoader()
```

Y si lo necesitamos, también podemos usar un LoadingManager como hicimos en la lección de Texturas.

Cargar modelos, buenas noticias, es casi tan fácil como cargar una textura. Llamamos al método load (...) y usamos los parámetros correctos:

La ruta al archivo

La función de devolución de llamada exitosa

La función de devolución de llamada de progreso

La función de devolución de llamada de error



```
gltfLoader.load(
  '/models/Duck/glTF/Duck.gltf',
  (gltf) =>
  {
    console.log('success')
    console.log(gltf)
  },
  (progress) =>
  {
    console.log('progress')
    console.log(progress)
  },
  (error) =>
  {
    console.log('error')
    console.log(error)
  }
)
```

Debería ver el progreso y la función de éxito que se está llamando. Si no se pudo cargar el archivo, es posible que se invoquen las funciones de error. Verifique la ruta y no olvide que no debemos agregar la parte / static.

Supongamos que sabemos lo que estamos haciendo y eliminemos la devolución de llamada de progreso y error:



```
gltfLoader.load(
  '/models/Duck/glTF/Duck.gltf',
  (gltf) =>
  {
    console.log(gltf)
  }
)
```

Agrega el modelo cargado a nuestra escena

Si observa el objeto registrado en la consola, encontrará muchos elementos. La parte más importante es la propiedad de la escena porque solo tenemos una escena en el modelo exportado.

Esta escena contiene todo lo que necesitamos. Pero también incluye más. Comience siempre por estudiar lo que está disponible en él y observe la propiedad de escala de los diferentes Grupos, Object3D y Mesh.

La malla debe ser nuestro pato. Realmente no nos importa la PerspectiveCamera. Tanto la cámara como el pato parecen estar en el primer y único Object3D en la matriz de niños de la escena. Peor aún, Object3D tiene una escala establecida en un valor mínimo.

Como puede ver, es un poco complejo incluso conseguir nuestro pato, y es donde la mayoría de los principiantes se pierden.

Todo lo que queremos es que nuestro pato entre en escena. Tenemos múltiples formas de hacerlo:

Agregue toda la escena en nuestra escena. Podemos hacer eso porque incluso si su nombre es escena, de hecho es un Grupo.

Agregue los elementos secundarios de la escena a nuestra escena e ignore la PerspectiveCamera no utilizada.

Filtre los niños antes de agregarlos a la escena para eliminar los objetos no deseados como PerspectiveCamera.

Agregue solo la malla pero termine con un pato que podría escalar, posicionarse o rotarse incorrectamente.

Abra el archivo en un software 3D y elimine PerspectiveCamera y luego expórtelo nuevamente.

Debido a que la estructura de nuestro modelo es simple, agregaremos Object3D a nuestra escena e ignoraremos la PerspectiveCamera no utilizada dentro. En lecciones futuras, agregaremos toda la escena como un objeto:

```
gltfLoader.load(  
    '/models/Duck/glTF/Duck.gltf',  
    (gltf) =>  
    {  
        scene.add(gltf.scene.children[0])  
    }  
)
```

Deberías ver el pato.

Puede probar con otros formatos, pero no el Draco que aún no funcionará:

```
gltfLoader.load(  
    '/models/Duck/glTF/Duck.gltf', // Default glTF  
  
    // Or  
    gltfLoader.load(  
        '/models/Duck/glTF-Binary/Duck.glb', // glTF-Binary  
  
    // Or  
    gltfLoader.load(  
        '/models/Duck/glTF-Embedded/Duck.gltf', // glTF-Embedded
```

Otro modelo llamado FlightHelmet (también tomado de las muestras del modelo glTF) está disponible en la carpeta / static / models /. Este modelo viene en un solo formato, que es el glTF predeterminado. En lugar de un hermoso casco, solo obtenemos algunas piezas.

El problema es que agregamos solo el primer hijo de la escena cargada a nuestra escena.

Lo que podemos intentar es hacer un bucle en los niños y agregarlos a la escena:

```
for(const child of gltf.scene.children)  
{  
    scene.add(child)  
}
```

Eso dará como resultado más elementos, pero no todos. Peor aún, al refrescar, puede obtener diferentes partes.

El problema es que cuando agregamos un niño de una escena a otra, se elimina automáticamente de la primera escena. Eso significa que la primera escena ahora tiene menos niños.

Cuando agregamos el primer objeto, se elimina de la primera escena y el segundo elemento simplemente se mueve al primer lugar. Pero su ciclo ahora toma el segundo elemento de la matriz. Siempre te quedarán elementos en la matriz de niños.

Existen múltiples soluciones a este problema. La primera solución es tomar los primeros hijos de la escena cargada y agregarlos a nuestra escena hasta que no quede ninguno:

```
while(gltf.scene.children.length)
{
    scene.add(gltf.scene.children[0])
}
```

Ahora tenemos todo el casco.

Otra solución sería duplicar la matriz de niños para tener una matriz independiente inalterada. Para hacer eso, podemos usar el operador de propagación ... y poner el resultado en una nueva matriz []:

```
const children = [...gltf.scene.children]
for(const child of children)
{
    scene.add(child)
}
```

Esta es una técnica de JavaScript nativa para duplicar una matriz sin tocar la original.

Finalmente, una solución buena y simple que mencionamos anteriormente es agregar la propiedad de escena:

Compresión Draco

Volvamos a nuestro pato, pero esta vez usaremos la versión de Draco:

```
gltfLoader.load(
```

```
    '/models/Duck/glTF-Draco/Duck.gltf',
```

Lamentablemente, no tenemos ningún pato. Si observa los registros, debería ver una advertencia similar a esta No se proporcionó una instancia de DRACOLoader. Necesitamos proporcionar una instancia de DRACOLoader a nuestro GLTFLoader para que pueda cargar archivos comprimidos.

Como vimos al navegar por los archivos, la versión de Draco puede ser mucho más ligera que la versión predeterminada. La compresión se aplica a los datos de la zona de influencia (normalmente la geometría). No importa si está utilizando el glTF predeterminado, el glTF binario o el glTF incrustado.

Ni siquiera es exclusivo de glTF y puede usarlo con otros formatos. Pero tanto glTF como Draco se hicieron populares simultáneamente, por lo que la implementación fue más rápida con los exportadores glTF.

Google desarrolla el algoritmo bajo la licencia Apache de código abierto:

Sitio web: <https://google.github.io/draco/>

Repositorio de Git: <https://github.com/google/draco>

Agregar el DRACOLoader

Three.js ya es compatible con Draco. Debemos comenzar importando el DRACOLoader:

```
import { DRACOLoader } from 'three/examples/jsm/loaders/DRACOLoader.js'
```

Luego podemos crear una instancia del cargador (antes del gltfLoader):

```
const dracoLoader = new DRACOLoader()
```

El decodificador está disponible en JavaScript nativo pero también en Web Assembly (wasm), y puede ejecutarse en un trabajador (otro hilo como vimos al final de la lección de Física). Esas dos características mejoran significativamente el rendimiento, pero implican tener un código completamente separado.

Three.js ya proporcionó este código separado. Para encontrarlo, debemos buscar en la dependencia Three.js y copiar la carpeta del decodificador Draco en nuestra carpeta / static /.

Esta carpeta de Draco se encuentra en / node_modules / three / examples / js / libs /. Tome la carpeta / draco / completa y cópiela en su carpeta / static /. Ahora podemos proporcionar la ruta a esta carpeta a nuestro dracoLoader:

```
dracoLoader.setDecoderPath('/draco/')
```

Finalmente, podemos proporcionar la instancia de DRACOLoader a la instancia de GLTFLoader con el método setDRACOLoader (...):

```
gltfLoader.setDRACOLoader(dracoLoader)
```

Tu pato debería estar de regreso, pero esta vez es una versión comprimida de Draco.

Aún puede cargar un archivo glTF no comprimido con GLTFLoader y el decodificador Draco solo se carga cuando es necesario.

Cuando usar la compresión Draco

Si bien podría pensar que la compresión Draco es una situación en la que todos ganan, no lo es. Sí, las geometrías son más ligeras, pero primero hay que cargar la clase DRACOLoader y el decodificador. En segundo lugar, su computadora necesita tiempo y recursos para decodificar un archivo comprimido que puede resultar en una breve congelación al comienzo de la experiencia, incluso si estamos usando un trabajador y un código de ensamblador web.

Tendrás que adaptarte y decidir cuál es la mejor solución. Si solo tiene un modelo con una geometría de 100 kB, probablemente no necesite Draco. Pero si tiene muchos MB de modelos para cargar y no

le importan algunos bloqueos al comienzo de la experiencia, es posible que necesite la compresión Draco.

Animaciones

Como dijimos anteriormente, glTF también admite animaciones. Y Three.js puede manejar esas animaciones.

Cargar un modelo animado

Primero, necesitamos un modelo animado. Podemos usar el fox ubicado en la carpeta / static / models / Fox / (también tomado de las muestras del modelo glTF).

Cambia la ruta para cargar ese zorro:

```
gltfLoader.load(  
  '/models/Fox/glTF/Fox.gltf',
```

Tenemos un problema; el zorro es demasiado grande. Si no puede verlo, mire arriba o aleje la imagen.

Antes de manejar la animación, arreglemos la escala. Si observa la composición de la escena importada, el zorro está compuesto por un Object3D, a su vez hecho de un hueso y un SkinnedMesh. No explicaremos cuáles son, pero la idea es que no deberíamos simplemente escalar el Object3D. Incluso si funcionara en este caso, probablemente no funcionaría con modelos más complejos.

Lo que podemos hacer aquí es escalar la escena cargada y agregarla directamente a nuestra escena:



```
gltfLoader.load(  
  '/models/Fox/glTF/Fox.gltf',  
  (gltf) =>  
  {  
    gltf.scene.scale.set(0.025, 0.025, 0.025)  
    scene.add(gltf.scene)  
  }  
)
```

Maneja la animación

Si observa el objeto gltf cargado, puede ver una propiedad denominada `animaciones` que contiene varios `AnimationClip`.

Estos `AnimationClip` no se pueden utilizar fácilmente. Primero necesitamos crear un `AnimationMixer`. Un `AnimationMixer` es como un reproductor asociado a un objeto que puede contener uno o varios `AnimationClips`. La idea es crear uno para cada objeto que necesite ser animado.

Dentro de la función de éxito, cree un mezclador y envíe el `gltf.scene` como parámetro:

```
const mixer = new THREE.AnimationMixer(gltf.scene)
```

Ahora podemos agregar `AnimationClips` al mezclador con el método `clipAction (...)`. Comencemos con la primera animación:

```
const action = mixer.clipAction(gltf.animations[0])
```

Este método devuelve una `AnimationAction`, y finalmente podemos llamar al método `play ()` en él:

```
action.play()
```

Lamentablemente, todavía no hay animación.

Para reproducir la animación, debemos decirle al mezclador que se actualice en cada cuadro. El problema es que nuestra variable de mezclador ha sido declarada en la función de devolución de llamada de carga, y no tenemos acceso a ella en la función tick. Para solucionarlo, podemos declarar la variable del mezclador con un valor nulo fuera de la función de devolución de llamada de carga y actualizarla cuando se carga el modelo:

```
let mixer = null

gltfLoader.load(
  '/models/Fox/glTF/Fox.gltf',
  (gltf) =>
{
  gltf.scene.scale.set(0.03, 0.03, 0.03)
  scene.add(gltf.scene)

  mixer = new THREE.AnimationMixer(gltf.scene)
  const action = mixer.clipAction(gltf.animations[0])
  action.play()
}
)
```

Y finalmente, podemos actualizar el mezclador en la función tick con el deltaTime ya calculado.

Pero antes de actualizarlo, debemos probar si la variable del mezclador es diferente de nula. De esta forma, actualizamos el mezclador si el modelo está cargado, lo que significa que la animación no está lista:

```
const tick = () =>
{
  // ...

  if(mixer)
  {
    mixer.update(deltaTime)
  }

  // ...
}
```

La animación debería estar ejecutándose. Puede probar las otras animaciones cambiando el valor en el método clipAction (...).

```
const action = mixer.clipAction(gltf.animations[2])
```

23. Raycaster y eventos de mouse

Introducción

Como sugiere el nombre, un Raycaster puede lanzar (o disparar) un rayo en una dirección específica y probar qué objetos se cruzan con él.

Puede usar esa técnica para detectar si hay una pared frente al jugador, probar si la pistola láser golpea algo, probar si hay algo debajo del mouse para simular eventos del mouse y muchas otras cosas.

Configuración

En nuestro iniciador, tenemos 3 esferas rojas, y vamos a disparar un rayo y ver si esas esferas se cruzan.

Crea el Raycaster

Crea una instancia de un Raycaster:

```
/*  
 * Raycaster  
 */  
const raycaster = new THREE.Raycaster()
```

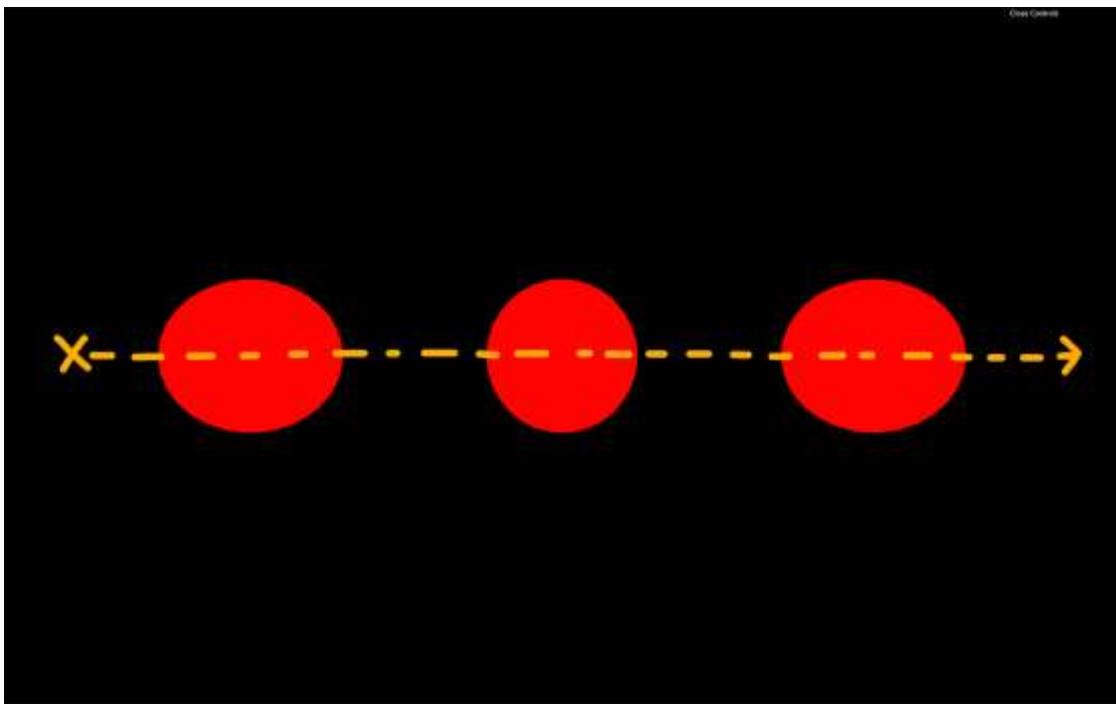
Para cambiar la posición y la dirección donde se lanzará el rayo, podemos usar el método set (...). El primer parámetro es la posición y el segundo parámetro es la dirección.

Ambos son Vector3, pero la dirección debe normalizarse. Un vector normalizado tiene una longitud de 1. No se preocupe, no tiene que hacer las matemáticas usted mismo y puede llamar al método normalize () en el vector:

```
const rayOrigin = new THREE.Vector3(- 3, 0, 0)  
const rayDirection = new THREE.Vector3(10, 0, 0)  
rayDirection.normalize()  
  
raycaster.set(rayOrigin, rayDirection)
```

Este ejemplo de un vector normalizado no es muy relevante porque podríamos haber establecido 1 en lugar de 10, pero si cambiamos los valores, todavía tendremos el método normalize () asegurándonos de que el vector tenga 1 unidad de largo.

Aquí, la posición del rayo supuestamente comienza un poco a la izquierda en nuestra escena, y la dirección parece ir hacia la derecha. Nuestro rayo debe atravesar todas las esferas.



Lanzar un rayo

Para lanzar un rayo y obtener los objetos que se cruzan, podemos usar dos métodos, `intersectObject (...)` (singular) e `intersectObjects (...)` (plural).

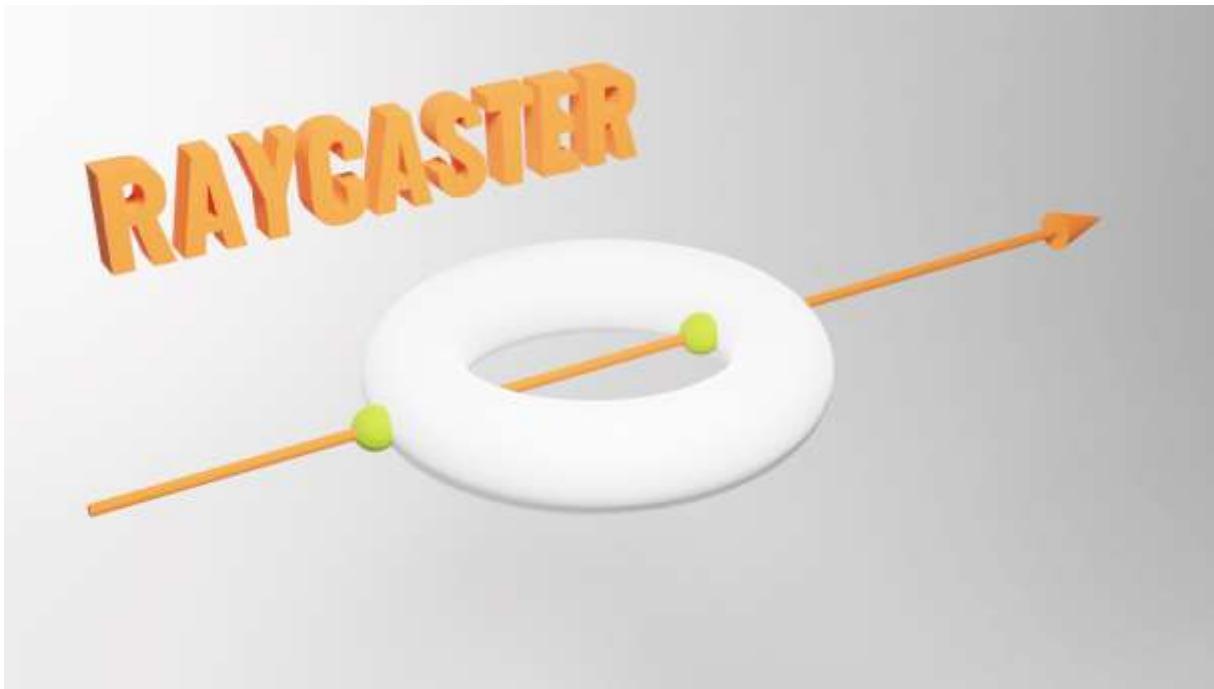
`intersectObject (...)` probará un objeto e `intersectObjects (...)` probará una matriz de objetos:

```
● ● ●  
const intersect = raycaster.intersectObject(object2)  
console.log(intersect)  
  
const intersects = raycaster.intersectObjects([object1, object2, object3])  
console.log(intersects)
```

Si observa los registros, verá que `intersectObject (...)` devolvió una matriz de un elemento (probablemente la segunda esfera) e `intersectObjects (...)`, devolvió una matriz de tres elementos (probablemente las 3 esferas).

Resultado de una intersección

El resultado de una intersección es siempre una matriz, incluso si está probando solo un objeto. Eso se debe a que un rayo puede atravesar el mismo objeto varias veces. Imagina una dona. El rayo atravesará la primera parte del anillo, luego el orificio del medio y luego nuevamente la segunda parte del anillo.



Cada elemento de esa matriz devuelta contiene mucha información útil:

distancia: la distancia entre el origen del rayo y el punto de colisión.

cara: qué cara de la geometría fue golpeada por el rayo.

faceIndex: el índice de esa cara.

objeto: qué objeto está afectado por la colisión.

punto: un Vector3 de la posición exacta en el espacio 3D de la colisión.

uv: las coordenadas UV en esa geometría.

Depende de usted utilizar esos datos. Si desea probar si hay una pared frente al jugador, puede probar la distancia. Si va a cambiar el color del objeto, puede actualizar el material del objeto. Si desea mostrar una explosión en el punto de impacto, puede crear esta explosión en la posición del punto.

Prueba en cada cuadro

Actualmente, solo lanzamos un rayo al principio. Si queremos probar cosas mientras se mueven, tenemos que hacer la prueba en cada cuadro. Animemos las esferas y las convertiremos en azules cuando el rayo se cruce con ellas.

Elimina el código que hicimos anteriormente y solo conserva la instanciación de raycaster:

```
const raycaster = new THREE.Raycaster()
```

Anime las esferas usando el tiempo transcurrido y el clásico Math.sin (...) en la función tick:

```
const clock = new THREE.Clock()

const tick = () =>
{
    const elapsedTime = clock.getElapsedTime()

    // Animate objects
    object1.position.y = Math.sin(elapsedTime * 0.3) * 1.5
    object2.position.y = Math.sin(elapsedTime * 0.8) * 1.5
    object3.position.y = Math.sin(elapsedTime * 1.4) * 1.5

    // ...
}
```

Debería ver las esferas ondeando hacia arriba y hacia abajo en diferentes frecuencias. Ahora actualice nuestro raycaster como lo hicimos antes, pero en la función tick:

```
const clock = new THREE.Clock()

const tick = () =>
{
    // ...

    // Cast a ray
    const rayOrigin = new THREE.Vector3(- 3, 0, 0)
    const rayDirection = new THREE.Vector3(1, 0, 0)
    rayDirection.normalize()

    raycaster.set(rayOrigin, rayDirection)

    const objectsToTest = [object1, object2, object3]
    const intersects = raycaster.intersectObjects(objectsToTest)
    console.log(intersects)

    // ...
}
```

Una vez más, no necesitamos normalizar rayDirection porque su longitud ya es 1. Pero es una buena práctica mantener normalize () en caso de que cambiemos la dirección.

También ponemos la matriz de objetos a probar en una variable objectsToTest. Eso será útil más tarde.

Si miras la consola, deberías obtener una matriz con intersecciones, y esas intersecciones siguen cambiando dependiendo de las posiciones de las esferas.

Ahora podemos actualizar el material de la propiedad del objeto para cada elemento de la matriz de intersecciones:

```
for(const intersect of intersects)
{
    intersect.object.material.color.set('#0000ff')
}
```

Desafortunadamente, todos se vuelven azules, pero nunca vuelven al rojo. Hay muchas formas de volver a poner en rojo los objetos que no se cruzan. Lo que podemos hacer es poner todas las esferas en rojo y luego convertir en azul las que se cruzan:

```
for(const object of objectsToTest)
{
    object.material.color.set('#ff0000')
}

for(const intersect of intersects)
{
    intersect.object.material.color.set('#0000ff')
}
```

Usa el raycaster con el mouse

Como dijimos anteriormente, también podemos usar el raycaster para probar si un objeto está detrás del mouse. En otras palabras, si está desplazando un objeto.

Matemáticamente hablando, es un poco complejo porque necesitamos lanzar un rayo de la cámara en la dirección del mouse, pero, afortunadamente, Three.js está haciendo todo el trabajo pesado. Por ahora, comentemos el código relacionado con el raycaster en la función tick.

Flotando

Primero, manejemos el flotar.

Para empezar, necesitamos las coordenadas del mouse. No podemos utilizar las coordenadas JavaScript nativas básicas, que están en píxeles. Necesitamos un valor que vaya de -1 a +1 tanto en el eje horizontal como en el vertical, siendo la coordenada vertical positiva al mover el ratón hacia arriba.

Así es como funciona WebGL y está relacionado con cosas como el espacio de clip, pero no es necesario que comprendamos esos conceptos complejos.

Ejemplos:

El mouse está en la parte superior izquierda de la página: -1 / 1

El mouse está en la parte inferior izquierda de la página: -1 / -1

El mouse está en el medio verticalmente y a la derecha horizontalmente: 1/0

El mouse está en el centro de la página: 0/0

Primero, creamos una variable de mouse con un Vector2 y actualicemos esa variable cuando el mouse se esté moviendo:

```
/**  
 * Mouse  
 */  
const mouse = new THREE.Vector2()  
  
window.addEventListener('mousemove', (event) =>  
{  
    mouse.x = event.clientX / sizes.width * 2 - 1  
    mouse.y = - (event.clientY / sizes.height) * 2 + 1  
  
    console.log(mouse)  
})
```

Mire los registros y asegúrese de que los valores coincidan con los ejemplos anteriores.

Podríamos lanzar el rayo en la devolución de llamada del evento mousemove, pero no se recomienda porque el evento mousemove podría activarse más que la velocidad de fotogramas de algunos navegadores. En su lugar, lanzaremos el rayo en la función tick como lo hicimos antes.

Para orientar el rayo en la dirección correcta, podemos usar el método setFromCamera () en el Raycaster. El resto del código es el mismo que antes. Simplemente actualizamos los materiales de los objetos a rojo o azul si se cruzan o no:

```
const tick = () =>  
{  
    // ...  
  
    raycaster.setFromCamera(mouse, camera)  
  
    const objectsToTest = [object1, object2, object3]  
    const intersects = raycaster.intersectObjects(objectsToTest)  
  
    for(const intersect of intersects)  
    {  
        intersect.object.material.color.set('#0000ff')  
    }  
  
    for(const object of objectsToTest)  
    {  
        if(!intersects.find(intersect => intersect.object === object))  
        {  
            object.material.color.set('#ff0000')  
        }  
    }  
  
    // ...
}
```

Las esferas deben volverse azules si el cursor está encima de ellas.

Eventos de entrada y salida del ratón

Los eventos de mouse como 'mouseenter', 'mouseleave', etc. tampoco son compatibles. Si quieres ser informado cuando el ratón "entra" en un objeto o "sale" de ese objeto, tendrás que hacerlo tú mismo. Lo que podemos hacer para reproducir los eventos mouseenter y mouseleave es tener una variable que contenga el objeto actualmente suspendido.

Si hay un objeto que se cruza, pero no había uno antes, significa que se ha producido una entrada del mouse en ese objeto.

Si ningún objeto se cruza, pero había uno antes, significa que ha ocurrido una salida del ratón.

Solo necesitamos guardar el objeto que se cruza actualmente:

```
let currentIntersect = null
```

Y luego, pruebe y actualice la variable currentIntersect:

```
const tick = () =>
{
    // ...
    raycaster.setFromCamera(mouse, camera)
    const objectsToTest = [object1, object2, object3]
    const intersects = raycaster.intersectObjects(objectsToTest)

    if(intersects.length)
    {
        if(!currentIntersect)
        {
            console.log('mouse enter')
        }

        currentIntersect = intersects[0]
    }
    else
    {
        if(currentIntersect)
        {
            console.log('mouse leave')
        }

        currentIntersect = null
    }
    // ...
}
```

Evento de clic del mouse

Ahora que tenemos una variable que contiene el objeto actualmente suspendido, podemos implementar fácilmente un evento de clic.

Primero, debemos escuchar el evento de clic independientemente de dónde suceda:

```
window.addEventListener('click', () =>
{
```

```
)
```

Luego, podemos probar si hay algo en la variable currentIntersect:

```
window.addEventListener('click', () =>
{
  if(currentIntersect)
  {
    console.log('click')
  }
})
```

También podemos probar que objeto fue afectado por el click:



```
window.addEventListener('click', () =>
{
  if(currentIntersect)
  {
    switch(currentIntersect.object)
    {
      case object1:
        console.log('click on object 1')
        break

      case object2:
        console.log('click on object 2')
        break

      case object3:
        console.log('click on object 3')
        break
    }
  }
})
```

Reproducir eventos nativos toma tiempo, pero una vez que los entiendes son realmente fáciles de realizar.

Lanzando Rayos con modelos

This is all great, but can we apply ray casting to imported models?

The answer is yes, and it's actually quite easy. But we are going to do it together because there are a few interesting things that we can learn along the way.

First, we need a model.

Load the model

The Duck model that we used in a previous lesson is located in the static/models/Duck/ folder.

Now would be a good time to try to load that model on your own and add it to the scene.

First, we are going to use the [GLTFLoader](#).

```
import { GLTFLoader } from 'three/examples/jsm/loaders/GLTFLoader.js'
```

Next, we need to instantiate it.

You can put that code anywhere after instantiating the scene and before the tick function:

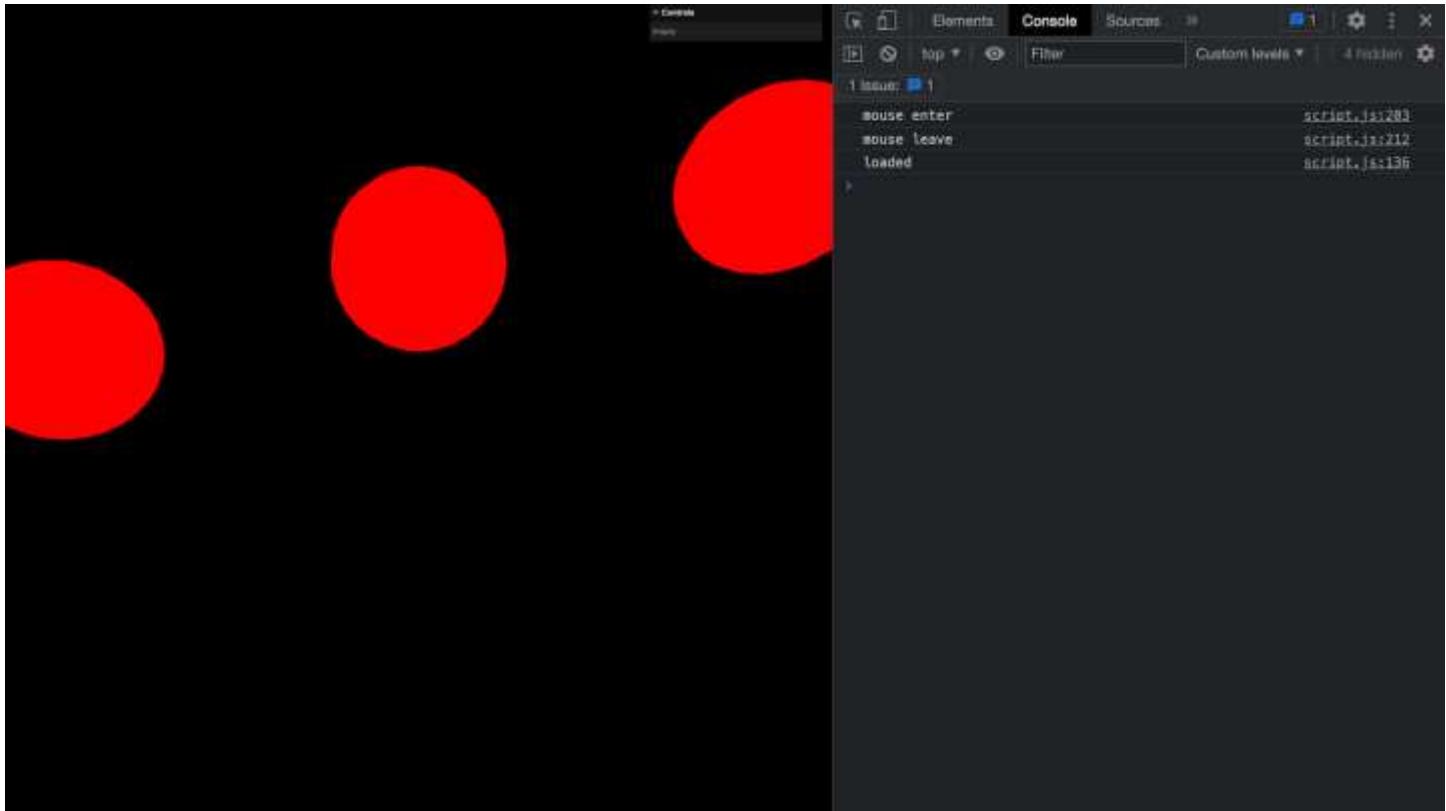
```
/**  
 * Model  
 */  
const gltfLoader = new GLTFLoader()
```

We can now call the load method. The two parameters are the path to the file and the function that should be called when the model is loaded.

We are going to use the glTF-Binary, but feel free to use another version. In addition, don't forget that you need to add a DracoLoader instance to the GLTFLoader instance if you want to use the Draco compressed version.

Call the load method and send './models/Duck/glTF-Binary/Duck.glb' as the path (without the static/ path) and a function with a console log in it:

```
gltfLoader.load(  
  './models/Duck/glTF-Binary/Duck.glb',  
  () =>  
  {  
    console.log('loaded')  
  }  
)
```



You should see the 'loaded' in the console.

We can now add the model to the scene. First, add a gltf argument to the function:

```
gltfLoader.load(  
  './models/Duck/glTF-Binary/Duck.glb',  
  (gltf) =>  
 {  
   console.log('loaded')  
 }  
)
```

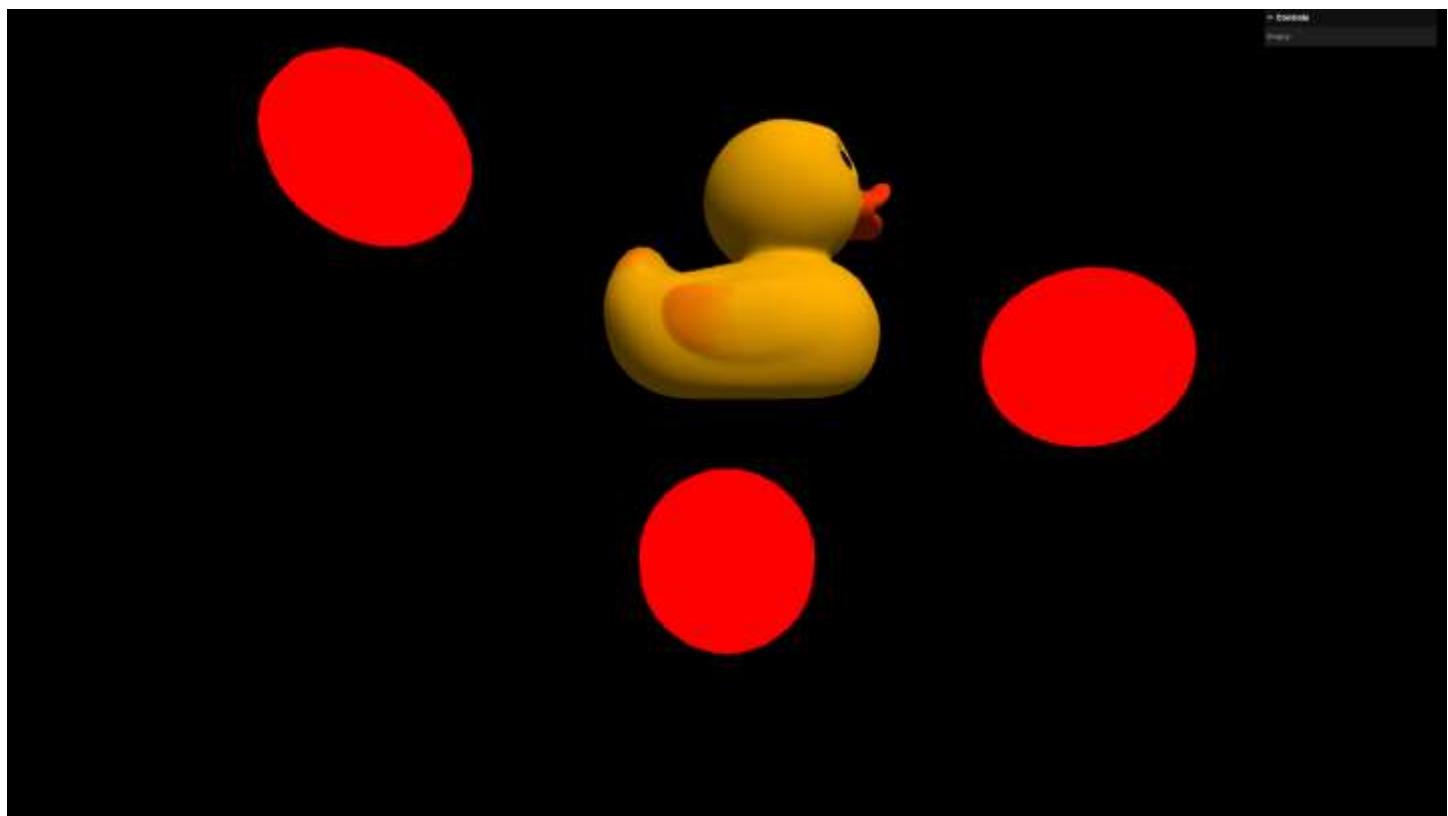
Now, add the whole loaded scene contained in the gltf.scene property to your own scene:

```
gltfLoader.load(  
  './models/Duck/glTF-Binary/Duck.glb',  
  (gltf) =>  
 {  
   scene.add(gltf.scene)  
 }  
)
```

Lights

If you've tried doing it on your own, you've probably struggled a little executing this step. It seems that something has been added to the scene, but it's all black. And the reason is that our Duck material is a [MeshStandardMaterial](#) and this kind of material can only be seen with lights. Let's add an [AmbientLight](#) and a [DirectionalLight](#):

```
/**  
 * Lights  
 */  
// Ambient light  
const ambientLight = new THREE.AmbientLight('#ffffff', 0.3)  
scene.add(ambientLight)  
  
// Directional light  
const directionalLight = new THREE.DirectionalLight('#ffffff', 0.7)  
directionalLight.position.set(1, 2, 3)  
scene.add(directionalLight)
```



Now that we can see the Duck, move it down a little:

```
gltfLoader.load(
  './models/Duck/glTF-Binary/Duck.glb',
  (gltf) =>
{
  gltf.scene.position.y = - 1.2
  scene.add(gltf.scene)
}
)
```

Intersect the model

Let's try the raycaster on the model.

The exercise will be quite simple. We want the Duck to get bigger when the cursor enters it and revert to its normal size when the cursor leaves it.

We are going to test if the cursor is in the Duck or not on each frame, meaning we need to configure the tick function. The raycaster is already set from the mouse and we can do our intersect test right after the code related to the test we did with the spheres.

Previously, we used raycaster.intersectObjects to test the raycaster against an array of meshes. But right now, we are only testing one object which is going to be the gltf.scene. Yes, this object might have multiple children and even worse, children inside children, but you'll see that it's not a problem and we are still testing one object.

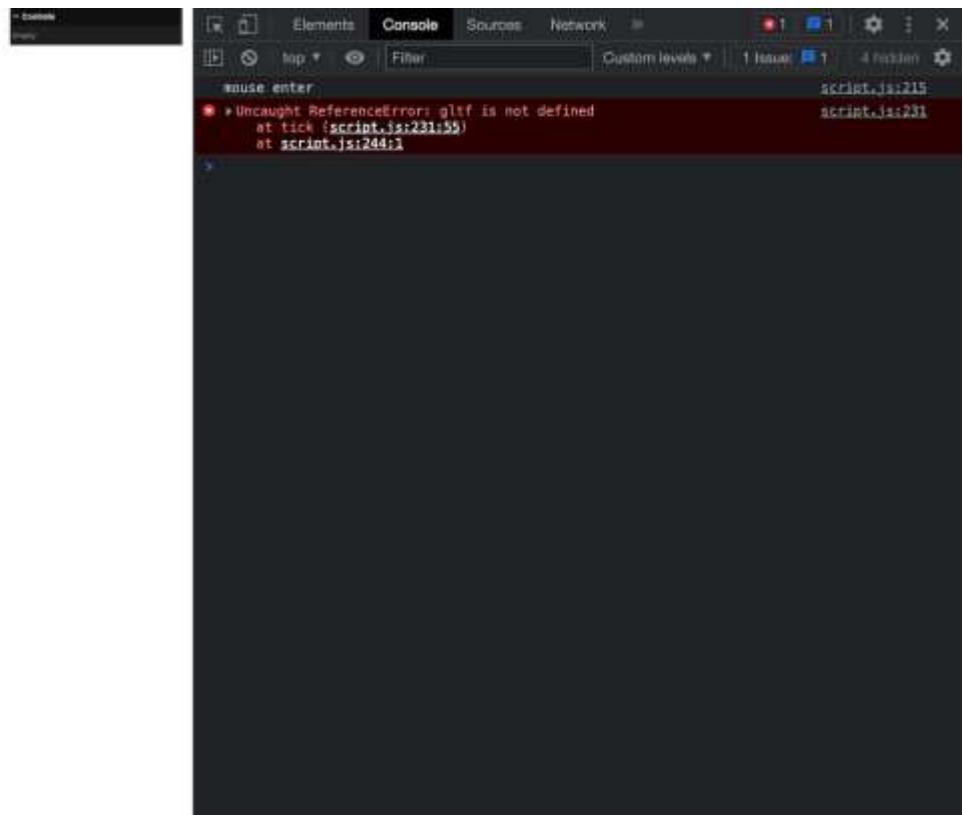
Instead of using intersectObjects (plural), we are going to use intersectObject (singular). It works just the same and will also return an array of intersections, but we have to send it an object instead of an array of objects.

So, what do you have to do? First, create a modelIntersects variable (so that it doesn't conflict with the intersects variable), then call the raycaster.intersectObject (singular) method and, finally, send it the gltf.scene (this code won't work):

```
const tick = () =>
{
  // ...

  // Test intersect with a model
  const modelIntersects = raycaster.intersectObject(gltf.scene)
  console.log(modelIntersects)

  // Update controls
  // ...
}
```



We made a mistake here. If you are comfortable with JS, you know that we can't access the `gltf` variable from outside of the loaded callback function. We call this the "scope" of a variable. Also, loading models takes time. Yes, we are testing in local with quite a simple model, but situations might vary and loading a complex object online will take time.

And those are classic issues that you will be confronted with when you try to interact with or animate loaded models.

To fix both of these issues, we are going to create a model variable using a `let` right before we load the model and set it to `null` (equivalent of "nothing" in JavaScript):

```
let model = null
gltfLoader.load(
    // ...
)
```

Since we created that `model` variable outside of any function, we will be able to use it in the `tick` function.

Next, when the model is loaded, we assign the `gltf.scene` to that `model`:

```
let model = null
gltfLoader.load(
  './models/Duck/glTF-Binary/Duck.glb',
  (gltf) =>
{
  model = gltf.scene
  gltf.scene.position.y = - 1.2
  scene.add(gltf.scene)
}
)
```

I'd also like to replace gltf.scene by model in that loaded function because it makes a bit more sense, although it's optional:

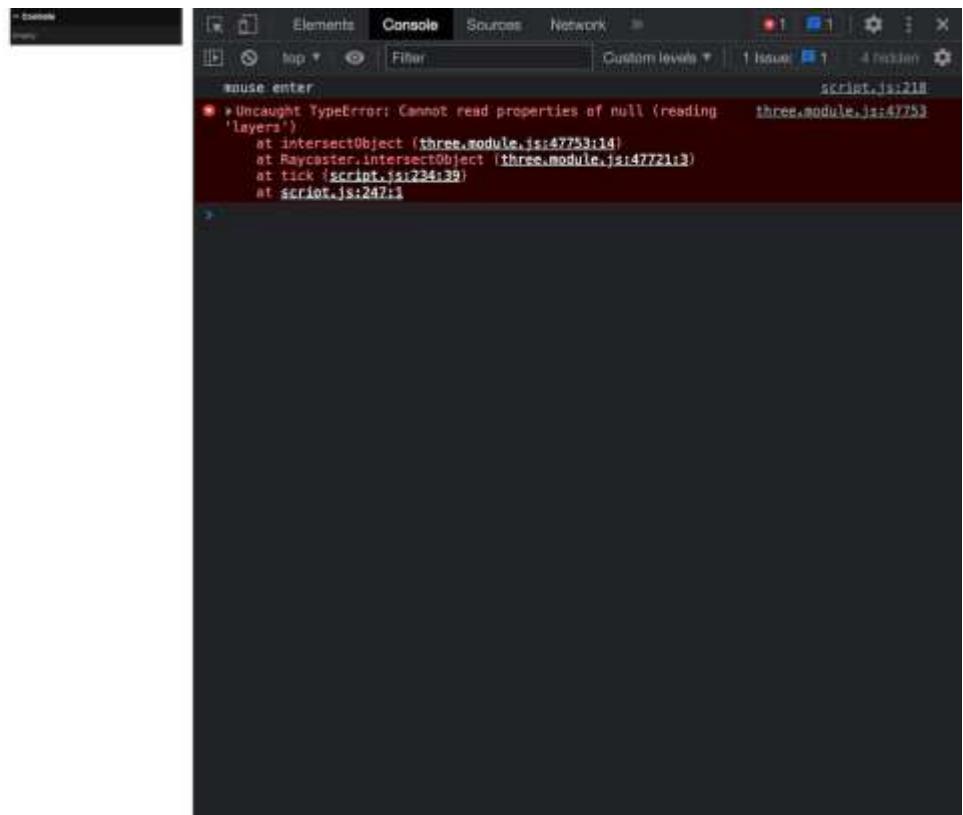
```
let model = null
gltfLoader.load(
  './models/Duck/glTF-Binary/Duck.glb',
  (gltf) =>
{
  model = gltf.scene
  model.position.y = - 1.2
  scene.add(model)
}
)
```

Back to the tick function and our intersectObject: we can now use the model variable instead of the gltf.scene (this code won't work just yet):

```
const tick = () =>
{
  // ...

  // Test intersect with a model
  const modelIntersects = raycaster.intersectObject(model)
  console.log(modelIntersects)

  // ...
}
```



And once again, we get an error. We've forgotten that loading a model takes time, meaning that the model variable will be null for a moment.

What we can do here is simply test if there is something in model with an if statement:

```
const tick = () =>
{
    // ...

    if(model)
    {
        const modelIntersects = raycaster.intersectObject(model)
        console.log(modelIntersects)
    }

    // ...
}
```

And now we get the array of intersects.

Notes

Before playing with the Duck size there are a few things to note.

Recursive

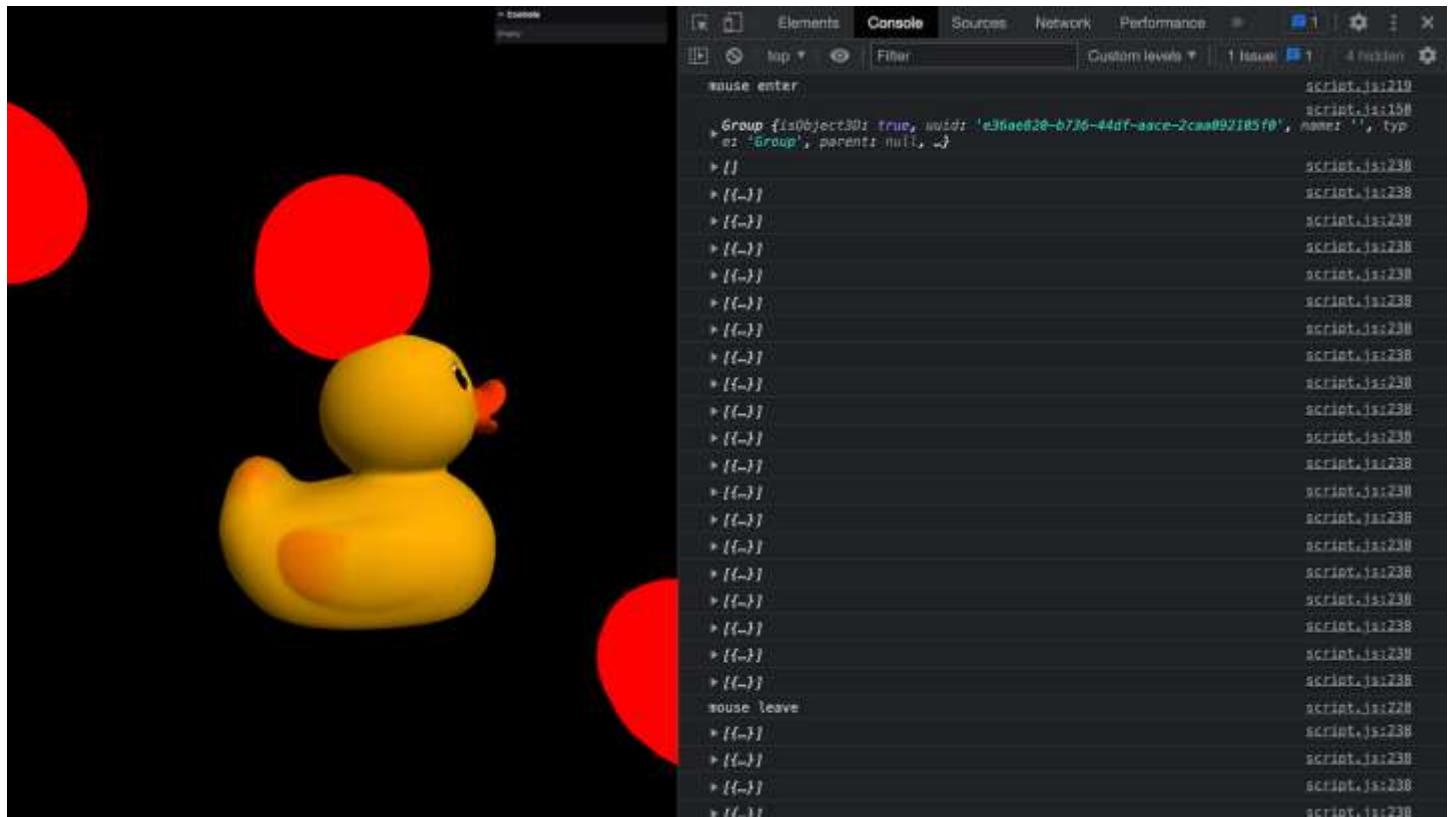
First, we are calling `intersectObject` on `model`, which is a [Group](#), not a [Mesh](#).

You can test that by logging `model` right before assigning it in the loaded callback function:

```

let model = null
gltfLoader.load(
    './models/Duck/glTF-Binary/Duck.glb',
    (gltf) =>
    {
        model = gltf.scene
        console.log(model)
        model.position.y = - 1.2
        scene.add(model)
    }
)

```



This shouldn't work since the Raycaster is supposed to be tested against Meshes. The reason why it's working is that, by default, the Raycaster will check the children of the object. Even better, it'll test children of children recursively.

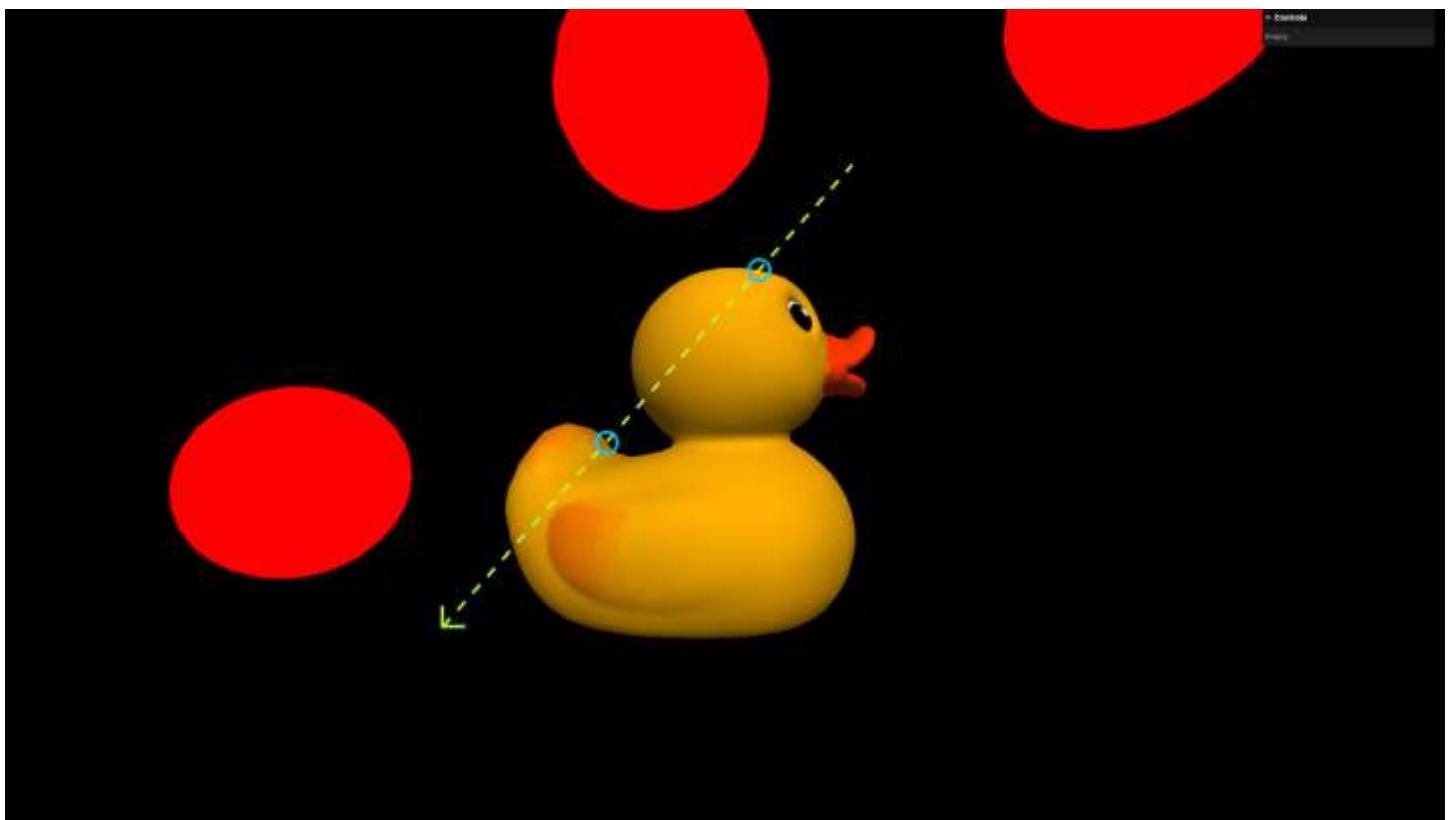
We can actually choose to deactivate that option by setting the second parameter of the `intersectObject` and `intersectObjects` methods to false, but we are fine with the default behaviour.

Array of intersects

The second thing to note is that we are receiving an array of intersects while we are testing only one object.

The first reason is that, since the Raycaster is testing children recursively, there can be multiple Meshes that intersect with the ray. It's not the case here since the Duck is made of only one Mesh, but we could have tested a more complex model.

The second reason is that, as we've seen earlier, even one Mesh can intersect multiple times with a ray and it's actually the case with our Duck. Tested from a very specific angle, you can have multiple intersects:



Update the scale

We are almost done. All we need to do now is update the scale of the model according to the array of intersects.

Right after calling the `intersectObject`, we can test the length of the array.

0 is considered as false, so we can just use `modelIntersects.length` as the condition.

If it's above 0, it'll be true, meaning that the mouse is hovering over the model and we should increase the scale. Otherwise, it will be false, meaning that the mouse isn't hovering over the model and we should set the scale to 1:

```
const tick = () =>
{
    // ...

    if(model)
    {
        const modelIntersects = raycaster.intersectObject(model)

        if(modelIntersects.length)
        {
            model.scale.set(1.2, 1.2, 1.2)
        }
        else
        {
            model.scale.set(1, 1, 1)
        }
    }
    // ...
}
```

24. Modelos customizados con Blender

Introducción

Ahora que sabemos como importar un modelo en nuestra escena; vamos a aprender como crear nuestro propio modelo usando un software de modelado 3d.

Eligiendo el software

Hay muchos softwares como cinema 4d, maya, 3ds max, blender, Zbrush, Marmoset Toolbag, Substance Painter, etc. Estos son geniales, y tienen diferentes tipos de drivers como e UX, el performance, las features, la compatibilidad, el precio, etc.

En esta lección, vamos a usar blender porque es gratuito, el desempeño es remarcable, trabaja con todas las principales OS, tiene muchas funciones, tiene una comunidad muy amplia, y se ha vuelto muy fácil desde la versión 2.8.

Se consiente que no serás un experto en blender al final de la lección. Tomaría un curso completo aprender todos sus aspectos, y hay muchos buenos recursos que aprender. La idea es entender lo básico y desmitificar el software para tener suficiente material para crear modelos simples.

Al inicio, vamos a descubrir los básicos. Eso será mucho por entender pero no te preocupes; vamos a repetir la mayoría de los atajos de teclado, mecanismos y las funcionalidades varias veces.

Si presionas un atajo de teclado incorrecto en algún punto, perderás la escena, o la interfaz se volverá completamente un desastre, solo cierra y abre de nuevo Blender.

Descargando Blender

Ve a la parte de descargas del sitio web de Blender y descarga la ultima versión:

<https://www.blender.org/download/>

El programa es bastante ligero, y no debería tomar mas de algunos minutos.

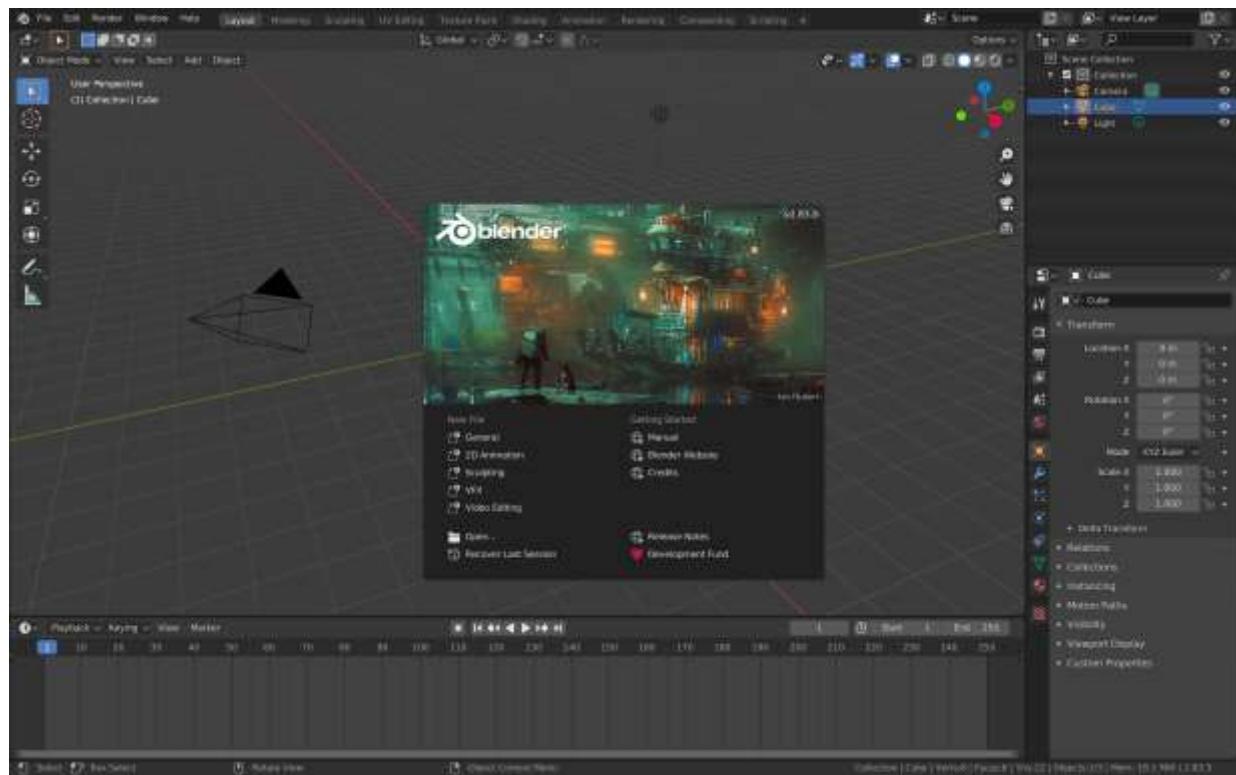
Una vez descargado simplemente instalalo.

La lección ha sido desarrollada con Blender 2.83.5. Aunque no deberían tenerse mayores diferencias, presta atención a potenciales cambios.

Interface

Pantalla de Bienvenida

La pantalla de bienvenida te da acceso a algunos links prácticos, plantillas y archivos abiertos recientemente.



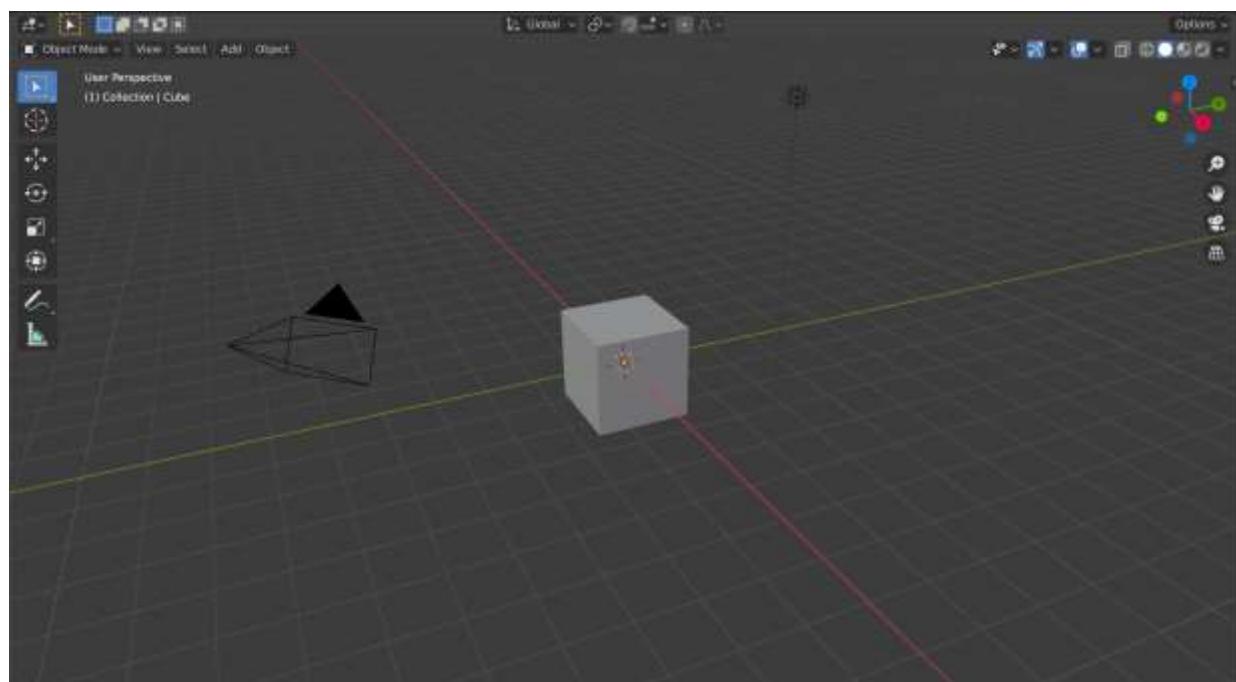
Esta imagen cambia con la versión de Blender, así que no te sorprendas si tienes una pantalla distinta. También puedes ver la versión exacta en la parte superior del recuadro. Da click en cualquier parte de la pantalla de bienvenida para cerrarla.

Areas

Las diferentes partes de la interfaz son llamadas áreas. Las áreas son muy flexibles, y tu puedes crear la disposición que tu quieras.

Areas por defecto

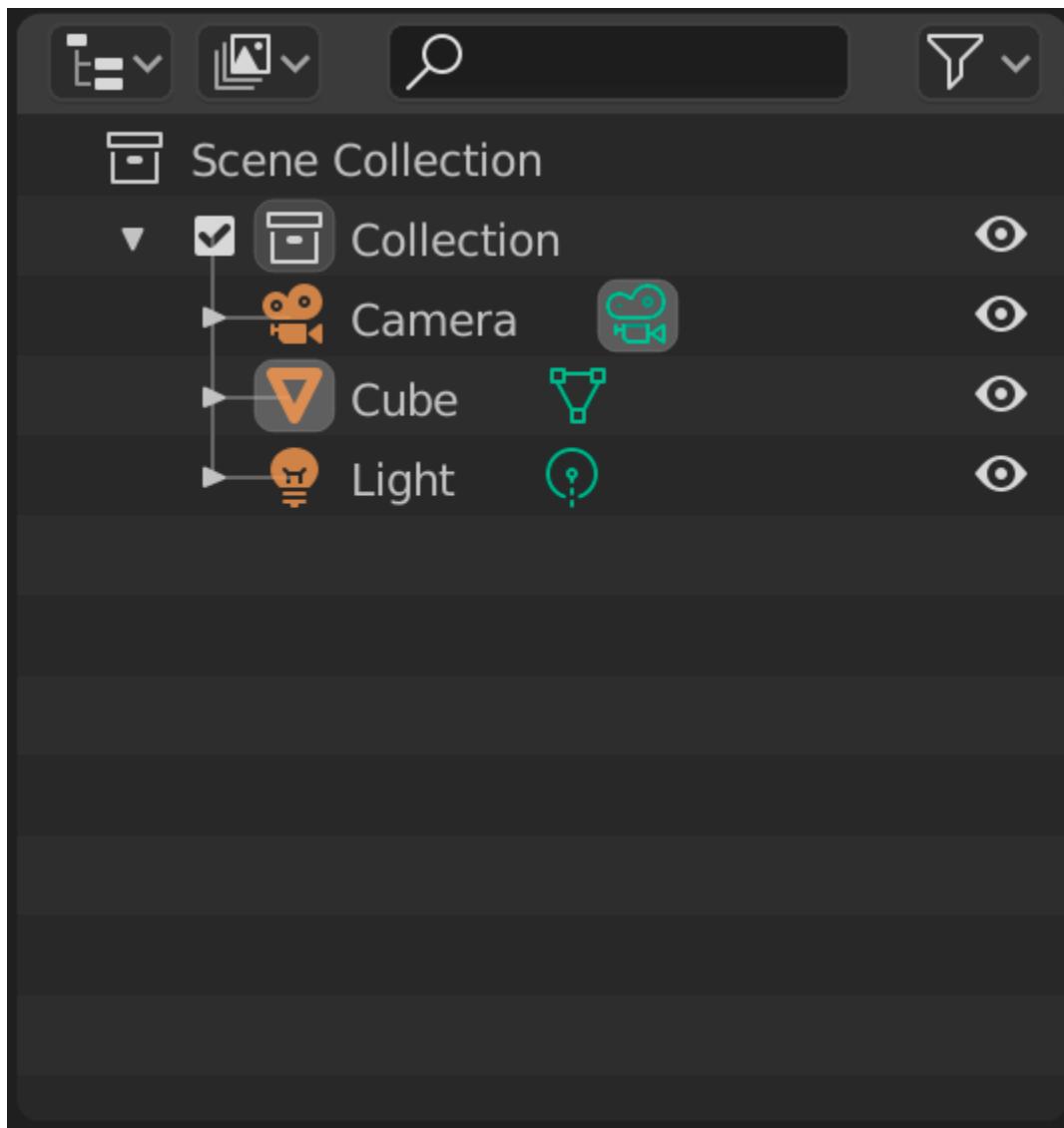
Por defecto, tienes el área principal llamada 3D Viewport



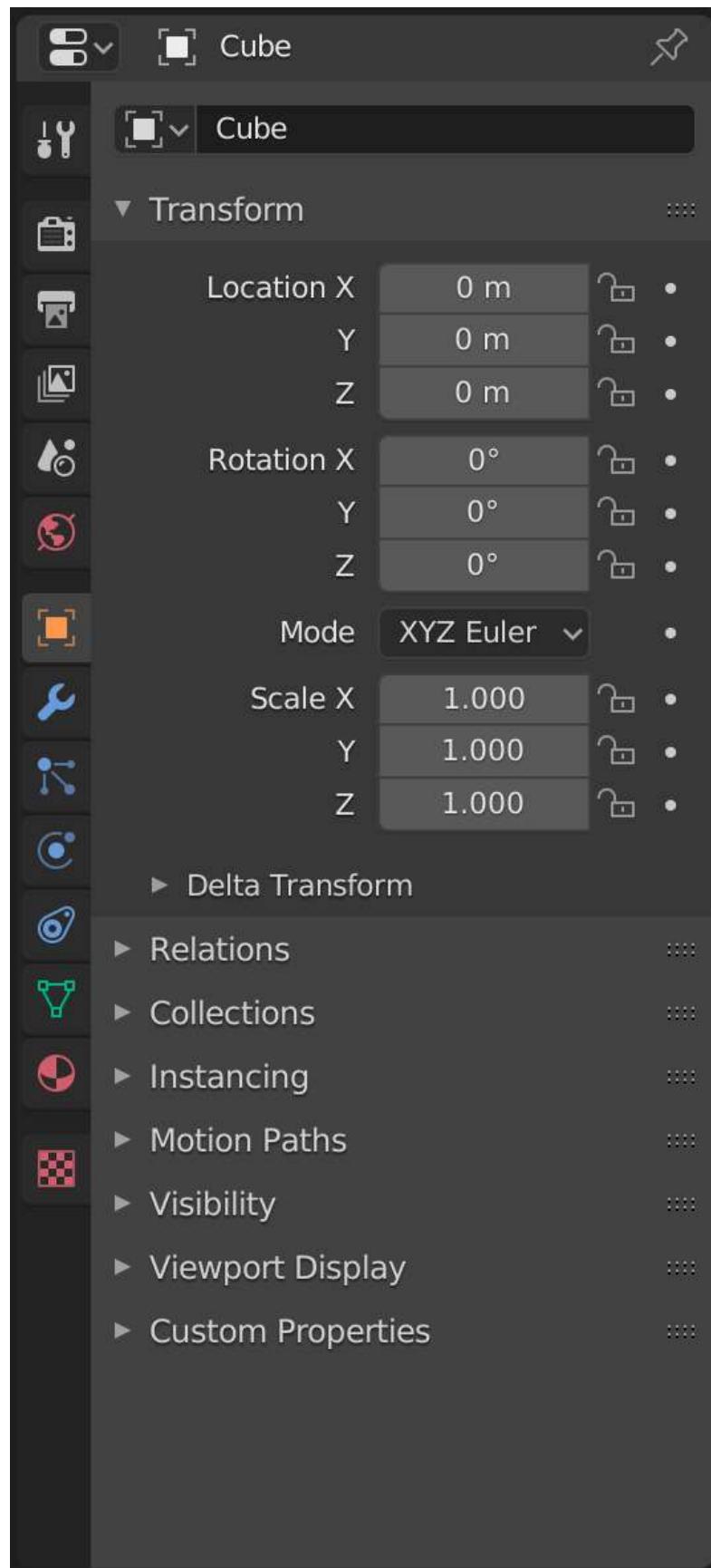
La línea de tiempo para crear animaciones:



El perfilador para ver y administrar los gráficos de escena (0 árbol de escena):

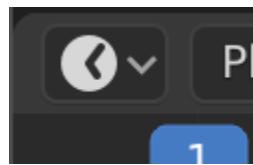


Las propiedades para administrar las propiedades del objeto activo (Seleccionado) y el entorno:

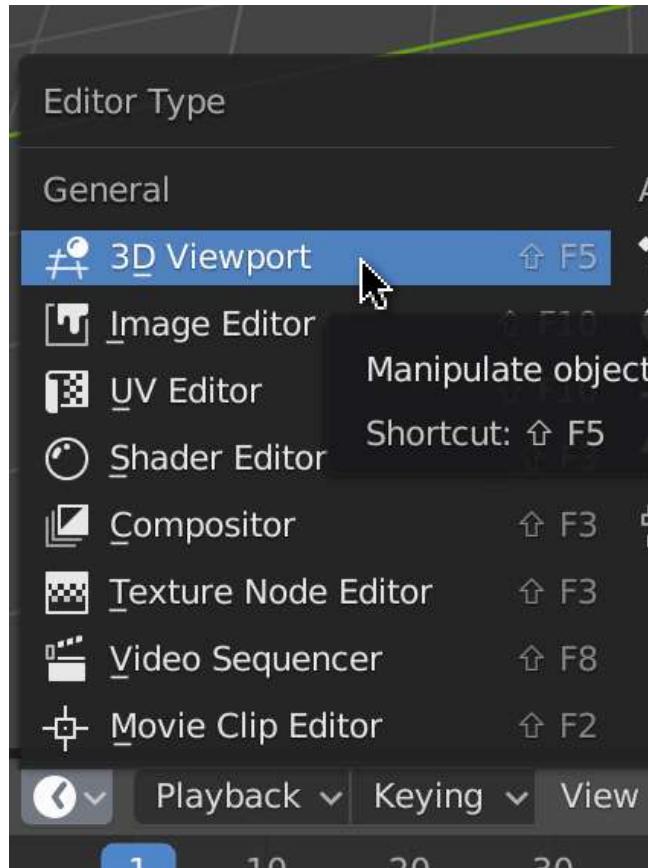


Cambiar un área

Para cambiar lo que se muestra en el área, da click en el botón izquierdo superior del área. Aquí, vamos a cambiar el área de la Línea de tiempo.



Vamos a cambiar la línea de tiempo por otra área del 3d viewport:



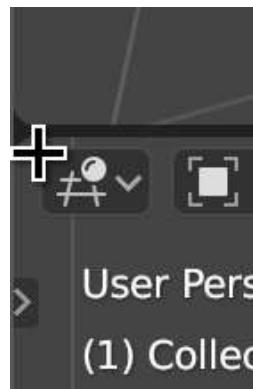
Redimensionar un área

Para redimensionar un área, posiciona el cursor entre las dos áreas, arrastra y suelta.

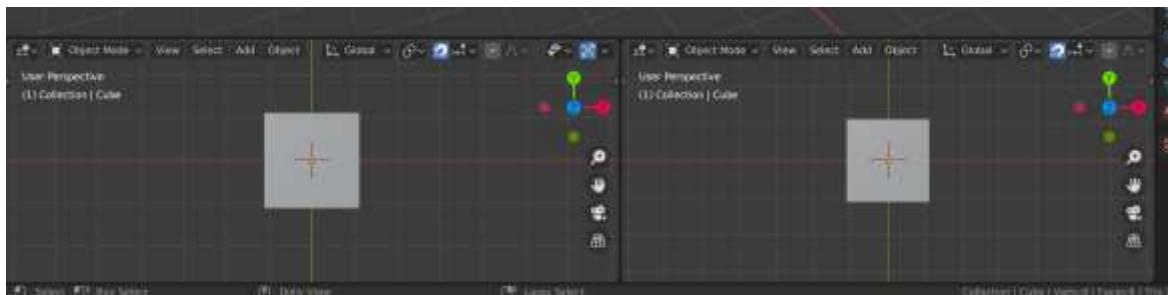


Crear nuevas áreas

Para crear un área nueva, primero, debemos decidir que área vamos a desplegar. Después, debemos posicionar el cursor en una de las cuatro esquinas de nuestra área (Algunos pixeles dentro del área? :



Finalmente, vamos a arrastrar y soltar en el área que queremos dividir:



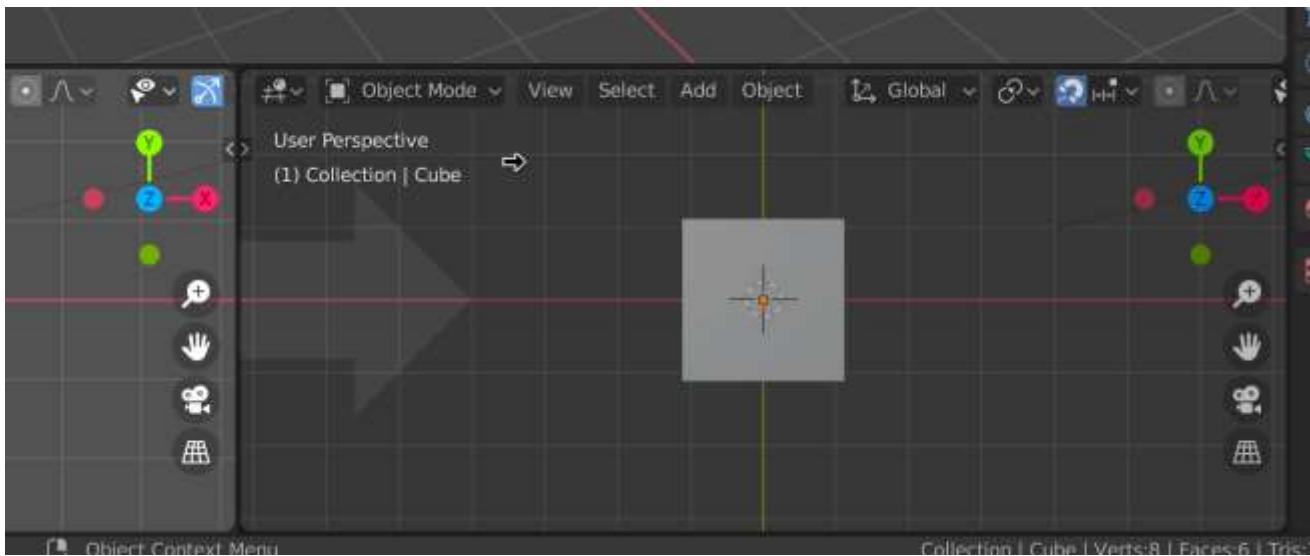
Remover un área

Remover algún área es algo confuso, y quizá termines con docenas de áreas indeseadas. No te preocunes, una vez que tengas la idea estará bien.

De alguna manera, no se remueve un área; desdividimos dos áreas. Lo primero, debes decidir cual de las dos áreas vamos a tomar sobre la otra. Si tu quieres remover el área de la derecha, inicia por el área de la izquierda. Entonces posiciona el cursor en una de las dos esquinas adyacentes al área que queremos remover (Algunos pixeles dentro del área que se supone que tomara el lugar de la otra):



Y después arrastra y suelta (justo como lo hicimos para crear un área) pero esta vez en la dirección opuesta (hacia el área que queremos remover):



podría tomar algunos intentos pero lo lograras.

Shortcuts

Una de las fortalezas de blender son los atajos de teclado. Hay toneladas de ellos, una vez que perfecciones los básicos, serás muy eficiente. No te preocupes, puedes usar todas las acciones de atajos de teclado atravez de la interfaz o con el panel de búsqueda que veremos después. A lo largo de esta lección, vamos a descubrir algunos de los atajos de teclado mas críticos.

Aquí esta una lista no exhaustiva de atajos de teclado:

<https://docs.google.com/document/d/1wZzJrEgNye2ZQqwe8oBh54AXwF5cYle56EGFe2bb0QU/edit?usp=sharing>

Una cosa importante a entender es que los atajos de teclado son sensibles a las áreas en las que se encuentra el cursor. Eso significa que el mismo shortcut tiene diferentes acciones dependiendo de lo que es detrás del cursor. La mayoría de los atajos de teclado que veremos en esta lección corresponden al 3D Viewport. Siempre deberías asegurarte de mantener el cursor sobre una de estas áreas cuando presiones los atajos de teclado.

Los atajos de teclado también son sensibles del modo, pero eso lo veremos en los modos después. Solo uno o dos atajos de teclado que veremos son diferentes en mac y Windows. Si hay alguna diferencia, ambas versiones serán citadas. Si el atajo de teclado incluye la tecla "CTRL" y estas usando mac, no asumas que es "CMD". Usa la tecla "CTRL" en su lugar.

Vista

Como puedes ver, puedes mover la vista en cada dirección posible. Podrías usar un panel táctil, pero yo recomiendo usar un mouse con botón de rueda que puedas presionar (o un tercer botón) por razones de productividad. Desde ahora, nos vamos a referir al botón de rueda (el que puedes presionar) como "MIDDLE MOUSE".

Si estas usando un panel táctil, puedes usar dos dedos.

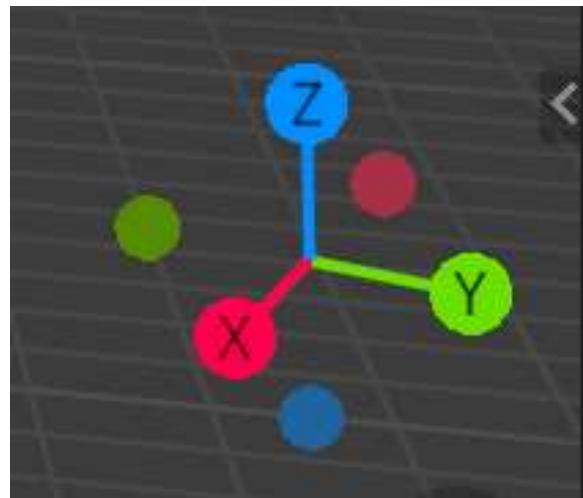
Si estas usando un Magic Mouse, puedes replicar el MIDDLE MOUSE. Ve a las preferencias atravez de "Editar-> Preferencias". Usando el menú de navegación a la izquierda, selecciona la sección "Input". Selecciona la opción "Emulate 3 Button Mouse".

También es mejor usar un teclado numérico.

Rotación de orbita



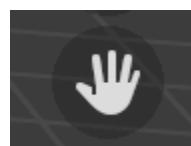
Podemos rotar la vista presionando el “MIDDLE MOUSE” y arrastrar y soltar en el “3d View port”. O podemos usar el gizmo en la parte superior derecha de cada “3D viewport”:



Truck y Pedestal



Truck es cuando la vista se mueve de izquierda a derecha, mientras el movimiento Pedestal es cuando la vista se mueve de arriba hacia abajo. Podemos realizar ambos movimientos simultáneamente presionando el “MIDDLE MOUSE” de nuevo, pero esta vez con la tecla “SHIFT” presionada. O podemos utilizar el icono de mano en la parte superior derecha:



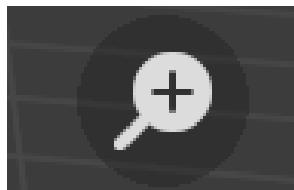
Truck también es llamado track.

Algunos podrían incorrectamente llamar estos movimientos como “pan”.

Dolly



El movimiento Dolly es cuando la vista se mueve adelante y hacia atrás. Podemos usar la "LLANTA" para lograrlo. O podemos usar el icono de magnificar en la parte superior derecha:

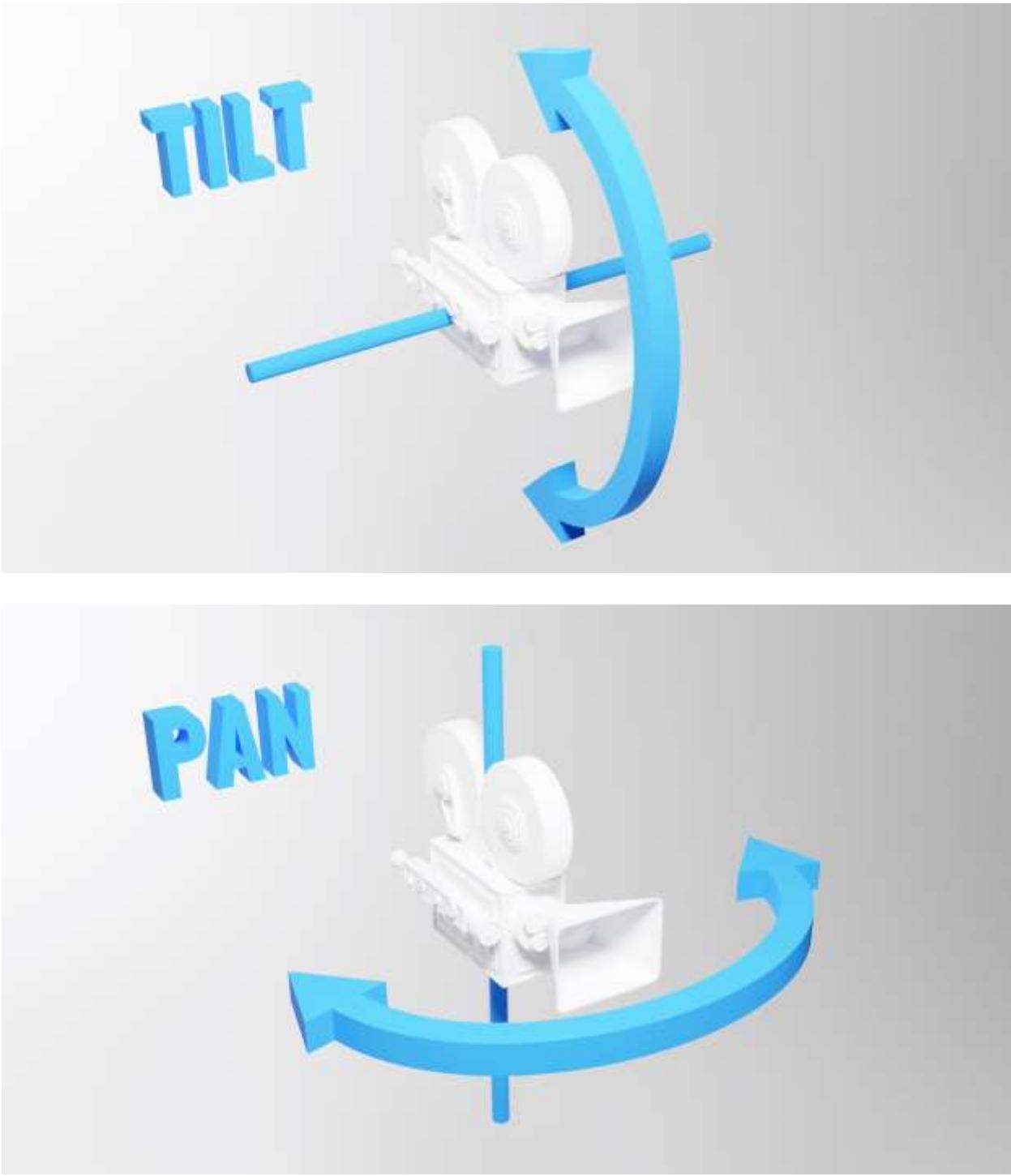


Se cuidadoso, hacer zoom es diferente de ir adelante y hacia atrás. Estamos alejándonos y acercándonos del punto de vista del que hablamos en la sección de Orbita, pero no podemos hacer zoom atravez de ese punto, y haciendo mucho zoom resultara en atascar la vista.

Para resolver este problema del limite del zoom, podemos movernos adelante y atrás presionando "SHIFT + CTRL + MIDDLE MOUSE" y arrastrar y soltar en el "3d Vlewport" (no funcionara con la técnica de los dos dedos del panel táctil). De esta manera, no vamos atascar el punto de vista.

No hay un icono en la interfaz para realizar esto.

Tilt y Pan



Tilt y pan son simples rotaciones en el punto de la cámara.

Tenemos que ir al modo de "Walk mode" también conocido como "Fly Mode" para usar estos movimientos.

Para hacer eso, si estas usando un teclado QWERTY, presiona "SHIFT + BACK QUOTE". La tecla "Back quote" is una comilla simple inclinada que puede ser usada para agregar un acento grave.

Encontrar dicho carácter podría ser un poco difícil porque la posición cambia mucho dependiendo de tu teclado. Podrías encontrar la en la esquina superior derecha, en la esquina inferior izquierda o muy cerca de la tecla "ENTER".

Si estas usando un teclado AZERTY, el atajo de teclado no funcionara. Necesitas cambiar el mapade teclas. Ve al menú de preferencias. "Edit > Preferences". Usando el menú de navegación a la izquierda,

selecciona la sección “Keymap”. En la sección de entradas, escribe “View navigation” y cambia el atajo de teclado a “View Navigation (walk/Fly) a “SHIFT + F”.

Eso es todo, ahora puedes usar el modo walk con “SHIFT + F”.

También puedes ir adelante o atrás, a los lados, usando el modo “Walk” con las flecha del teclado o “WASD” si usas un qwerty.

Perspectiva / Ortografica

La vista por defecto usa perspectiva. Podemos alternar con la versión ortográfica con “NUMPAD 5” o el icono de malla en la parte superior derecha:



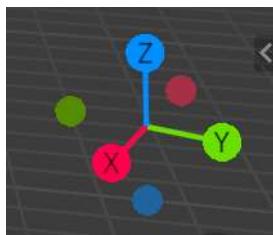
Ejes

Podemos alinear la cámara en el eje X, Y y Z presionando en el teclado numérico el “1”, “3”, y “7”.

Estamos hablando de pad numéricos, no los números en la parte superior del teclado.

Para posicionar de la cámara al inverso, presiona las mismas teclas pero con “CTRL”.

O podemos usar el gizmo y dar click en los ejes:



En Blender consideramos el eje vertical como el eje Z, a diferencia de Three.js donde es el eje Y.

Camara

Podrás haber visto la cámara en el “3D Viewport”. Para tener la camapra en “Viewpoint”, presiona “Numpad 0”:



La cámara se usara mientras hacemos renders. No se pueden realizar los renders de una escena sin una cámara.

Reset

Algunas veces nos perdemos, y no sabemos que escena tenemos seleccionada. Podemos enfocarnos en la escena anterior presionando “SHIFT + C”.

Enfocar

Para enfocar la cámara en un objeto, selecciona el objeto con el “MOUSE IZQUIERDO”, ahora presiona el “NUMPAD ,” (Estamos hablando de la coma en el pad numérico y podría ser un punto dependiendo del teclado que estés usando).

Podemos también enfocar un objeto y ocultar todo lo demás con “NUMPAD /”. Usa el mismo atajo de teclado para abandonar el modo de enfoque.

Seleccionando

Como vimos, podemos seleccionar objetos con el “MOUSE IZQUIERDO”. Podemos también seleccionar múltiples objetos con “SHIFT + MOUSE IZQUIERTO”.

Quizás habras notado que uno de los objetos siempre tiene un contorno mas luminoso. Este objeto no esta seleccionado; Es también uno activo. Veremos mas de eso adelante,

Para deshacer una selección, presiona “CMD + Z” (CTRL + Z en Windows). Si, seleccionando objetos es considerado como una action que se puede deshacer. Mientras esto podría parecer extraño, es realmente de ayuda cuando haces un click accidental.

Para deseleccionar un objeto, usa “SHIFT + MOUSE IZQUIERDO” de nuevo. Si no eta activo, se activara, y si esta activo, se deseleccionara.

Para seleccionar todo presiona “A”.

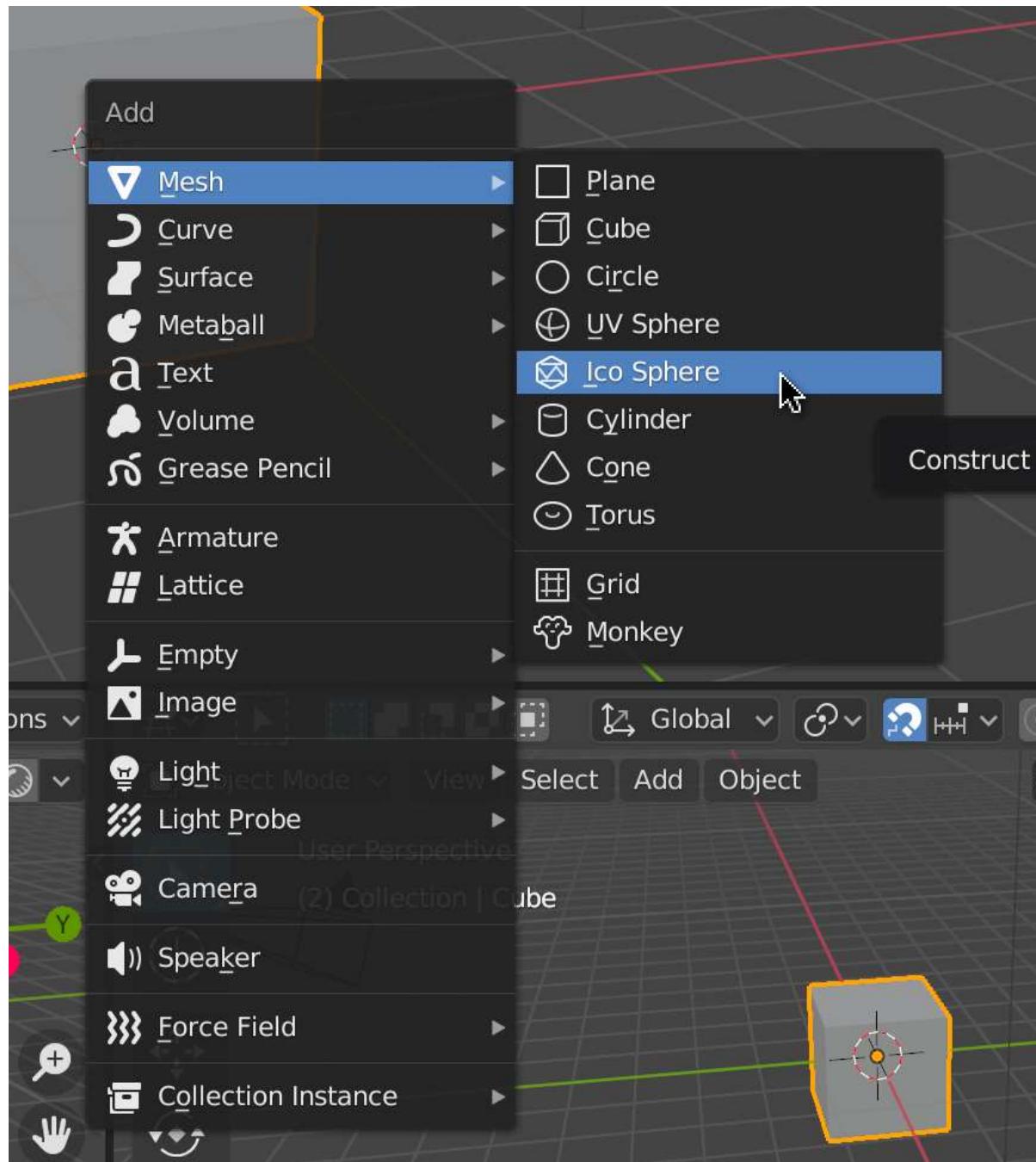
Para deseleccionar todo presiona dos veces “A”.

Para seleccionar un área rectangular presiona “B”.

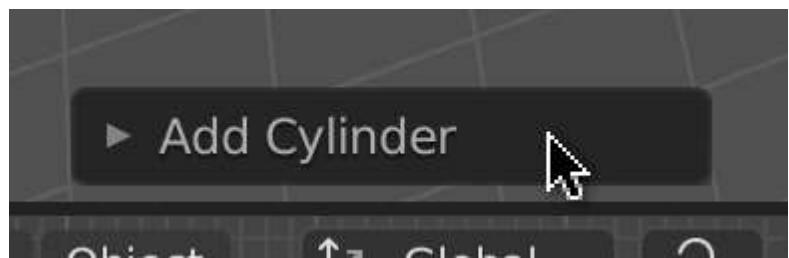
Para seleccionar como si estuvieras pintando, presiona “C”. Mientras estes en este modo, usa “WHEEL” para cambiar el radio.

Creando objetos

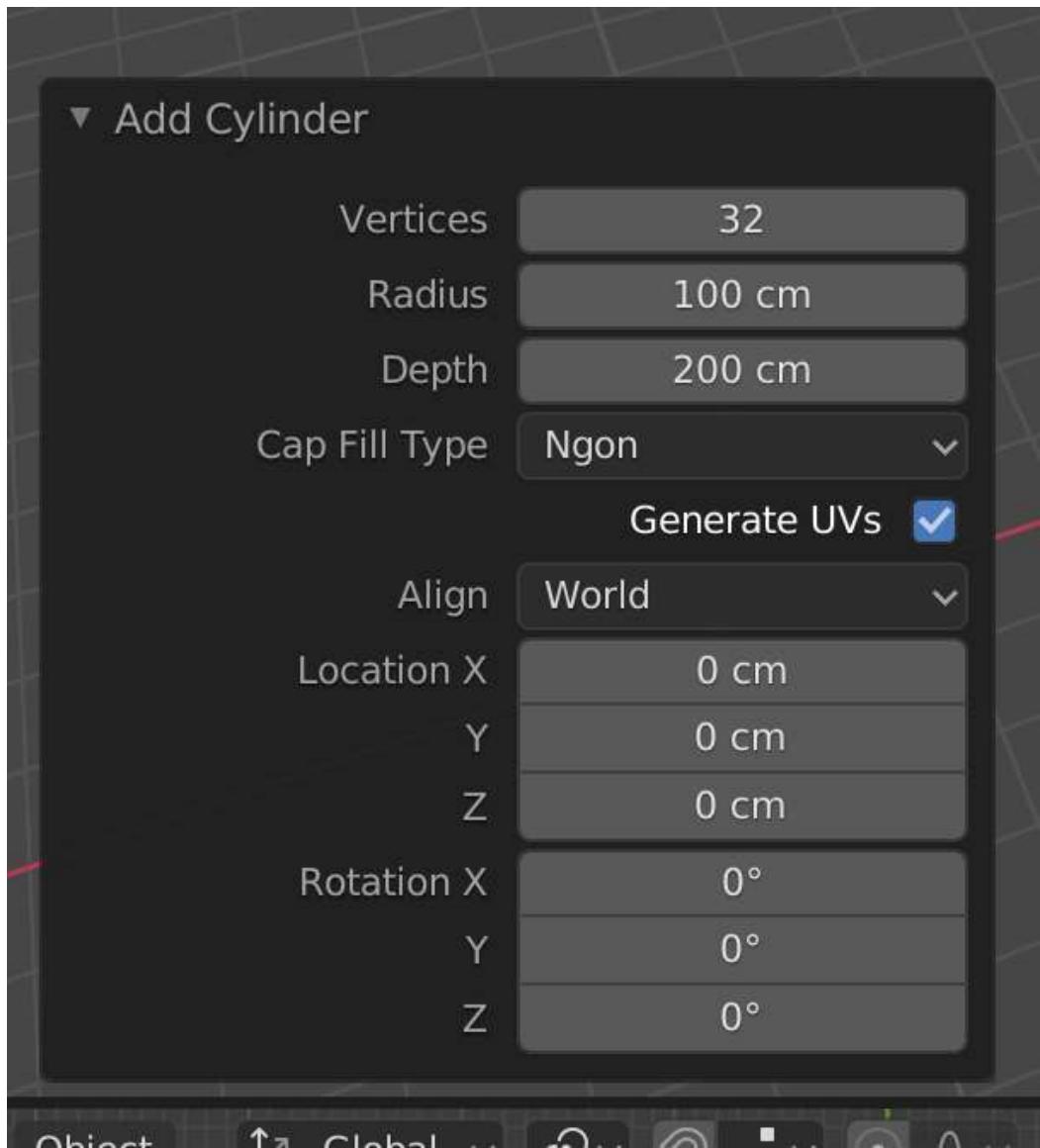
Para crear objetos, con el cursor sobre “3D Viewport”, presiona “SHIFT + A”. Un menú se desplegara detrás del cursor. Navegando atravez del menú para crear varios objetos. La mayoría de lo que vamos a ver en las lecciones esta en el sub-menu “Mesh”:



Cuando creas un objeto, un botón pequeño debería aparecer en la esquina inferior izquierda:



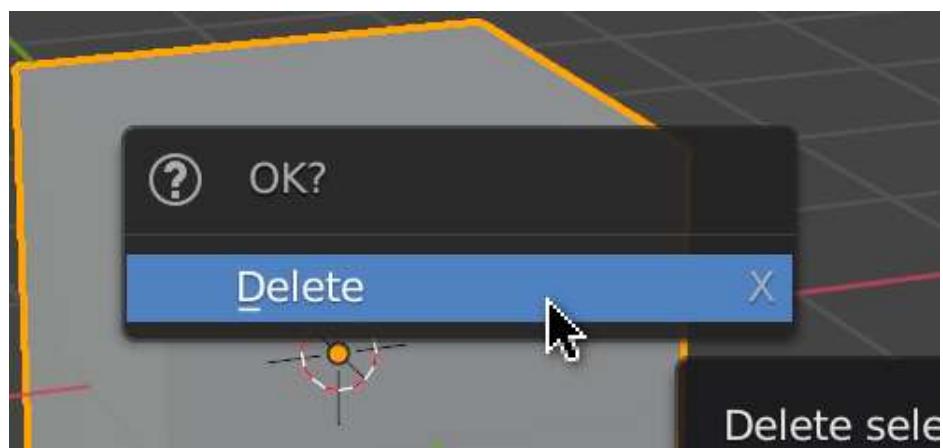
Da un click en la opción para cambiar varias de las propiedades como el tamaño, la subdivisión, y muchas otras propiedades relacionadas con el objeto que estamos tratando de hacer:



Si das click en cualquier otra parte, perderás el menú. Puedes seábrirlo con "F" pero si vas a iniciar modificaciones en la geometría, no serás capaz de reabrir este menú.

Eliminando objetos

Para eliminar un objeto, selecciónalo y, con el cursor sobre el "3D Viewport", presiona "X". Un menú de confirmación debería aparecer detrás del cursor. Da click en el para eliminar el objeto:



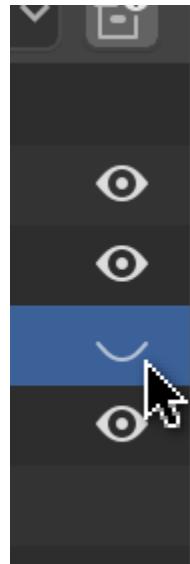
Ocultando objetos

Para ocultar los objetos seleccionados, presiona "H".

Para mostrar objetos ocultos, presiona "ALT + H".

Para ocultar los objetos no seleccionados, presiona "SHIFT + H".

Puedes también administrar esto en el "Outliner" con el icono de ojo.



Transformando Objetos

Hay 3 tipos de transformaciones, como en Three.js

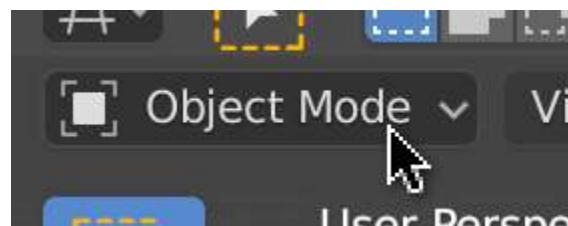
Modos

Estamos actualmente en el "Object Mode", donde podemos crear, eliminar y transformar objetos.

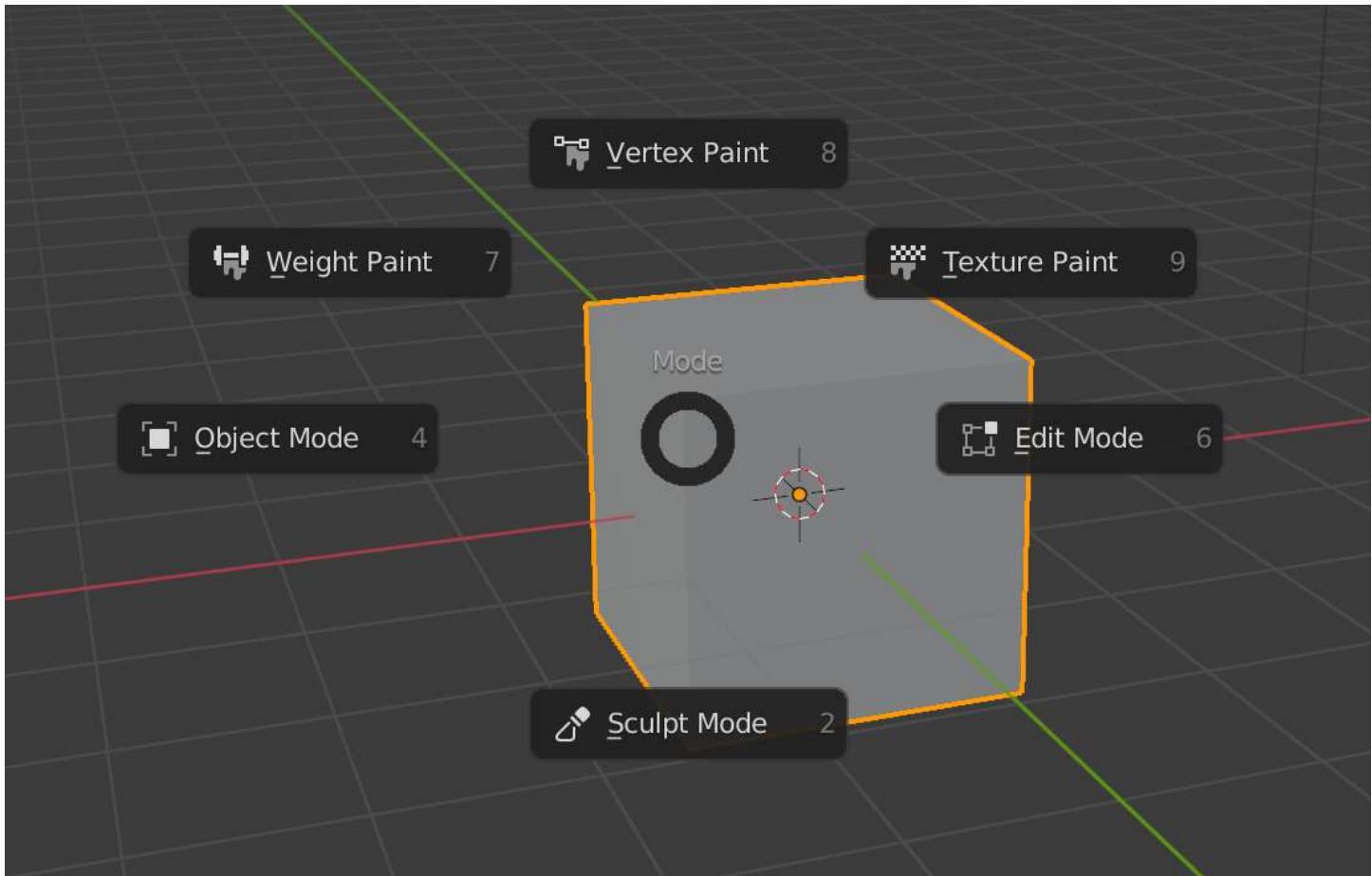
Hay muchos otros modos.

Cambiando el modo

Podemos cambiar el modo con el menú en la parte superior izquierda, en el botón de cualquier "3D viewport" área:



O podemos presionar "CTRL + TAB" para abrir el menú en rueda, (El tipo de menú mas cool):



No vamos a cubrir todos los modos, pero vamos a usar el “Edit Mode”. Selecciona una malla “Mesh” o crea una y cambia al modo de edición (Edit Mode).

Modo de edición

Existe un atajo de teclado para alternar el Edit Mode. Simplemente presiona la tecla “TAB”.

The Edit Mode es muy similar al “Object Mode” pero podemos editar los vértices, bordes y las caras. Por defecto, podemos cambiar el vertice. Intenta seleccionar los vértices, transformarlos con los atajos de teclado usuales – G para la posición, R para la rotación, S para la escala.

Para cambiar entre los bordes y las caras, podemos usar los botones en la parte superior derecha del “3D Viewport”:



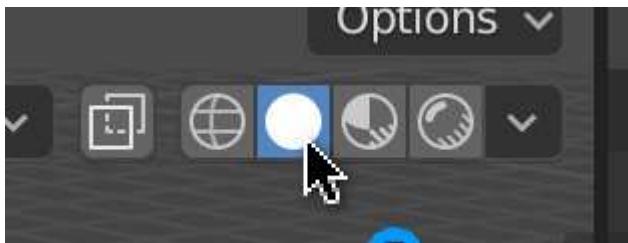
O podemos presionar los primeros tres números en la parte superior del teclado, 1, 2 y 3. Una vez que termines de editar el objeto, sal del “Edit Mode” con “TAB”.

Sombreado

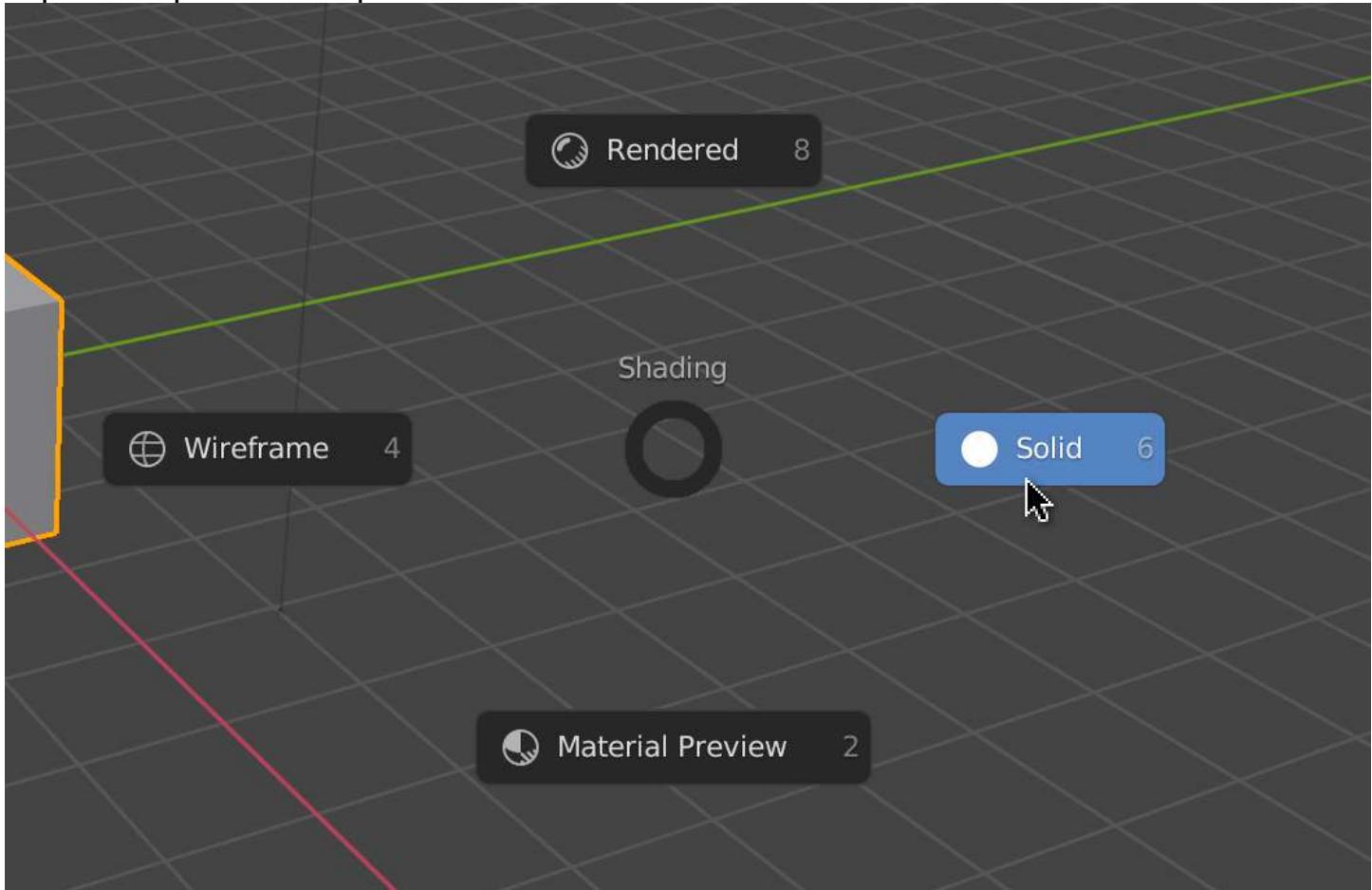
Sombreado es como ves los objetos en el “3D Viewport”.

Estamos actualmente en sombreado sólido o “Solid”. Este sombreado te permite ver los objetos con un material por defecto y sin soporte de luces. Es permanente y conveniente.

Podemos cambiar el sombreado con los botones en la esquina superior derecha del “3D viewport”.



O podemos presionar "Z" para abrir un menú de rueda:



Solid: El valor por defecto con el mismo material para todos los objetos.

Material: Como el “Solid shading”, pero con una previsualización de los materiales y sus texturas.

Wireframe: Todas las geometrías están en el wireframe.

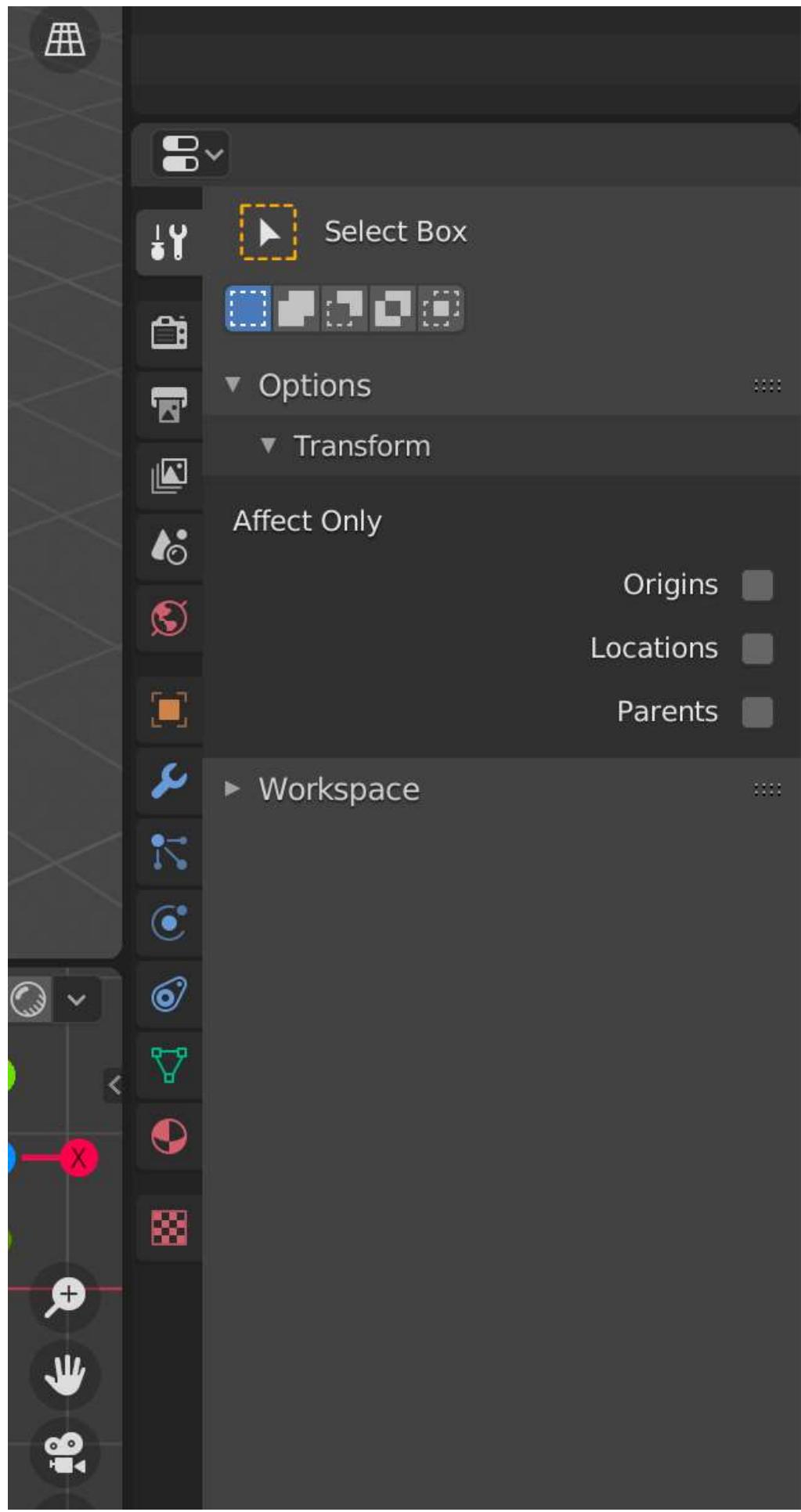
Renderizer: Renderizado de baja calidad, es el mas realista pero el de peor desempeño.

Si no puedes ver mucho con el “Render”, podría ser porque no tienes suficientes luces en la escena.

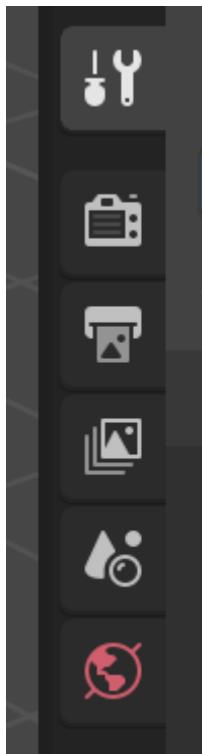
Solo agrega una o dos presionando “SHIFT + A” para abrir el menú de “Add”.

Propiedades

A menos que cambies la disposición, el área inferior derecha, denominada “Properties” o propiedades, vamos a jugar con las propiedades de render, propiedades de entorno y las propiedades de objeto activas. Recuerda que el objeto activo es el que tiene el contorno iluminado.

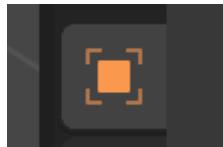


No vamos a cubrir todas las pestañas pero las superiores se utilizan para manejar los renders y los entornos:



Cuarto Propiedades de objeto

Las “Object properties” o propiedades de objeto te permiten modificar la precisión de las propiedades como transformaciones:



Cuarto Propiedades de modificador

Las “Modifiers Properties” o propiedades de modificadores te permiten agregar lo que llamamos modificadores. Estos son modificaciones no destructivas. Puedes subdividir, doblar, crecer, engrosar, etc. Y apagarlas como tu deseas:

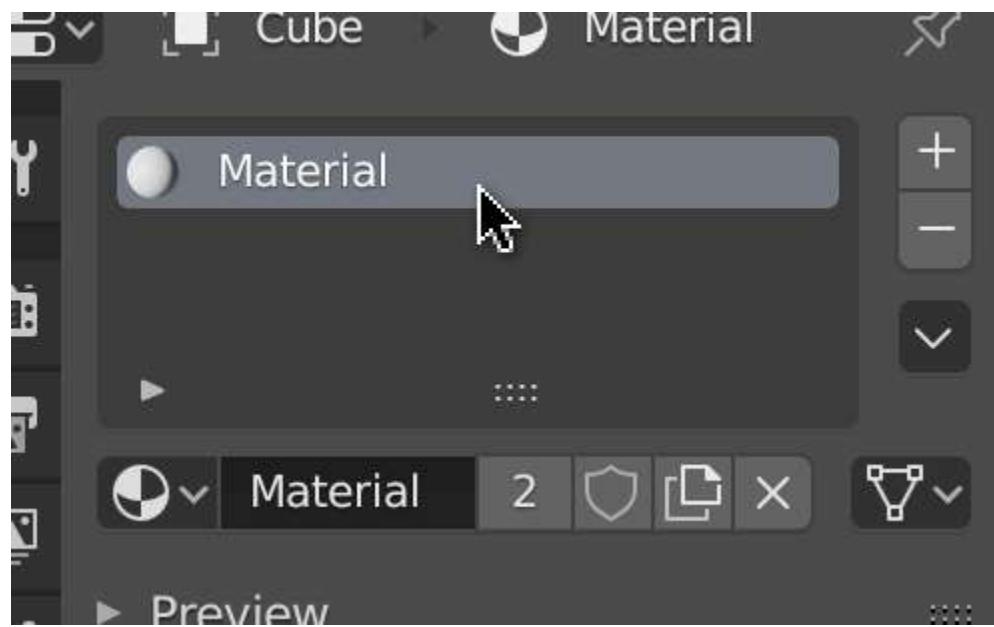


Cuarto propiedades de material

Las “Material Properties” o propiedades de material te permiten jugar con los materiales:



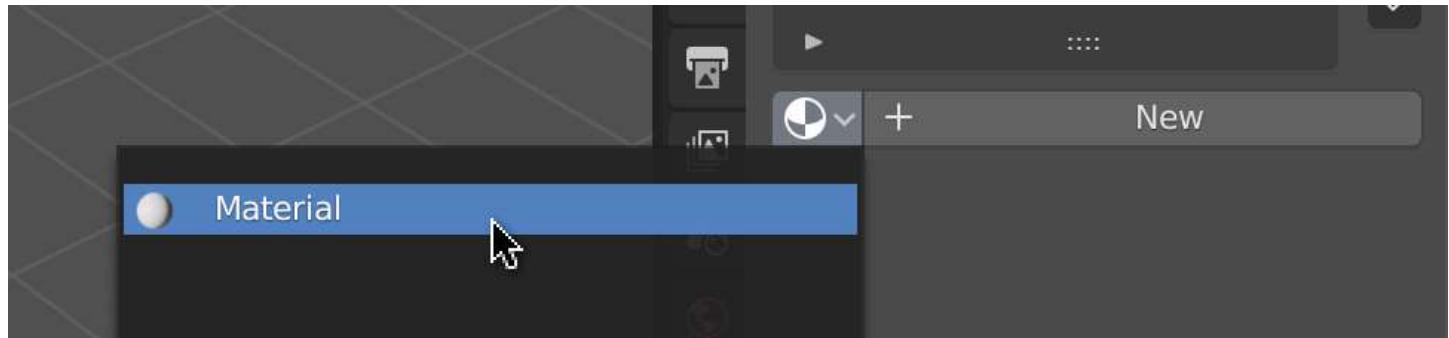
Por defecto, tienes acceso a un material denominado “Material”, y puede ser aplicado al cubo por defecto si no lo eliminás:



Puedes remover el material con el botón de “-” y combinarlo con el botón de “+”, pero usualmente usamos solo un material por cada malla.



Si no hay material en la malla, podemos seleccionar uno existente, o podemos crear uno nuevo con el botón de nuevo:



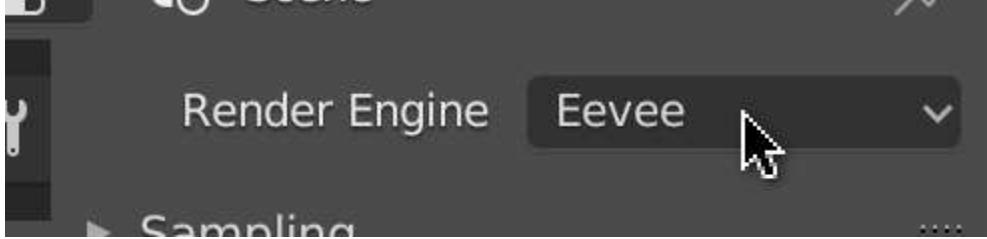
Podemos tener diferentes tipos de superficies para un material. El por defecto es llamado “Principled BSDF” y este tipo de superficie usa los principios de PBR como el “MeshStandardMaterial” lo hace en Three.js. Esto significa que deberíamos tener un resultado muy similar si exportamos este tipo de material a una escena de Three.js.
No veremos otros tipos de materiales en esta lección.

Motores de renderizado

Ve a la pestaña de “Render Properties”:



En este panel, podemos cambiar el motor de renderizado o “Render Engine”:



Existen tres tipos de motor de renderizado:

“Eevee”: Un motor de renderizado en tiempo real. Usa el GPU como lo hace Three.js, es muy performante, pero tiene limitaciones como realismo, rebote de luz, reflexión y refracción.

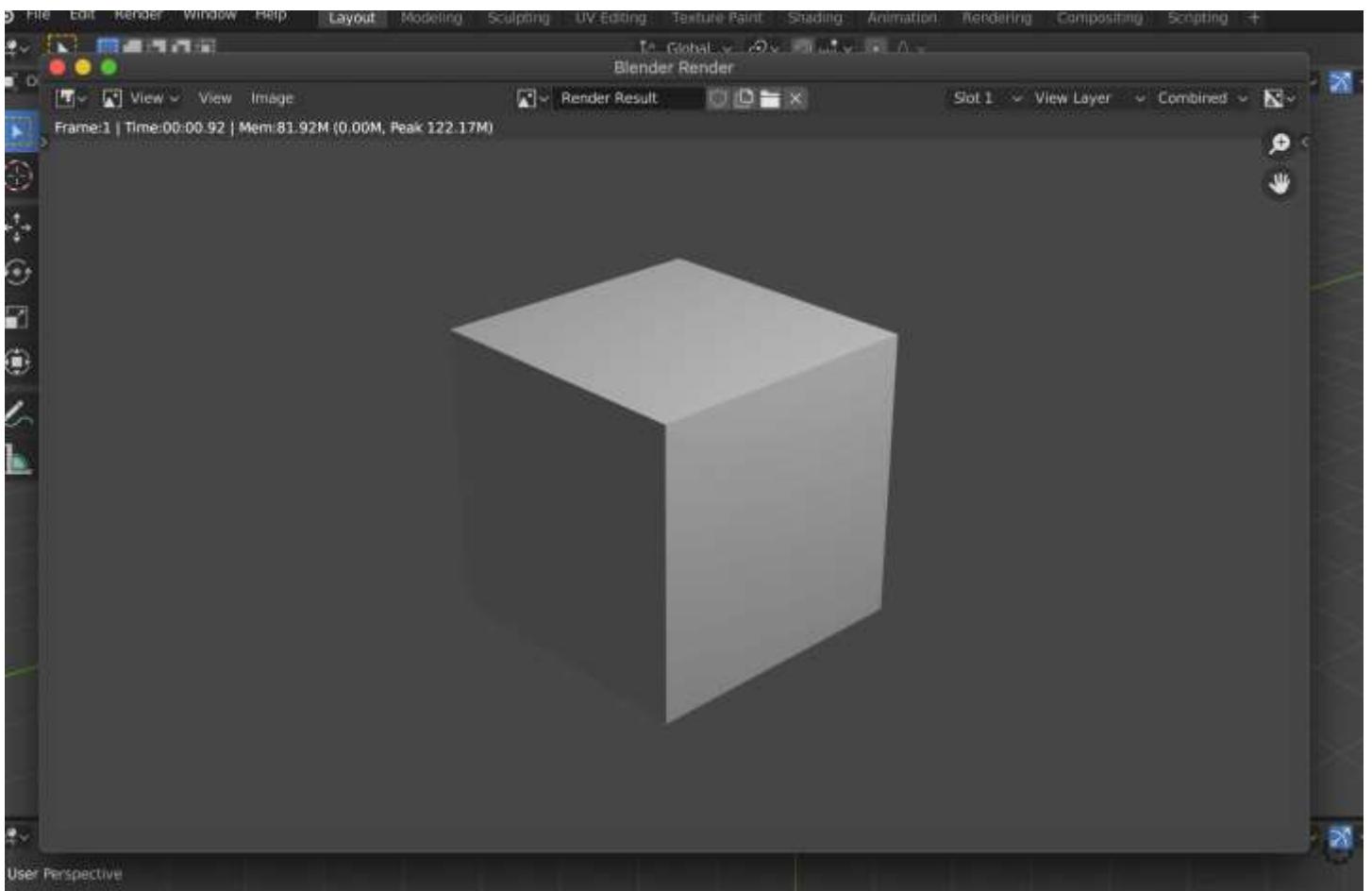
“Workbench”: Un motor antiguo que no se usa mucho ahora. Su desempeño es bastante bueno, pero los resultados no son muy realistas.

“Cycles”: Un motor de trazado de rayos. Es muy realista, Maneja las reflecciones de luz, reflexiones profundas, reflexiones profundas, y muchas más funcionalidades, pero es muy lento, y podrías terminar esperando por horas o incluso días para renderizar tu escena.

Puedes cambiar la propiedad y ver si algo cambia en la escena. Asegurate de usar el sombreado “Render” presionando la tecla Z para cambiar el sombreado. Los desarrolladores de Blender hicieron un excelente trabajo manteniendo un resultado muy similar entre los motores de renderizado.

El que se tiene por defecto es “Eevee” y es perfecto en nuestro caso porque es un renderizado en tiempo real, y Three.js es también renderizado en tiempo real.

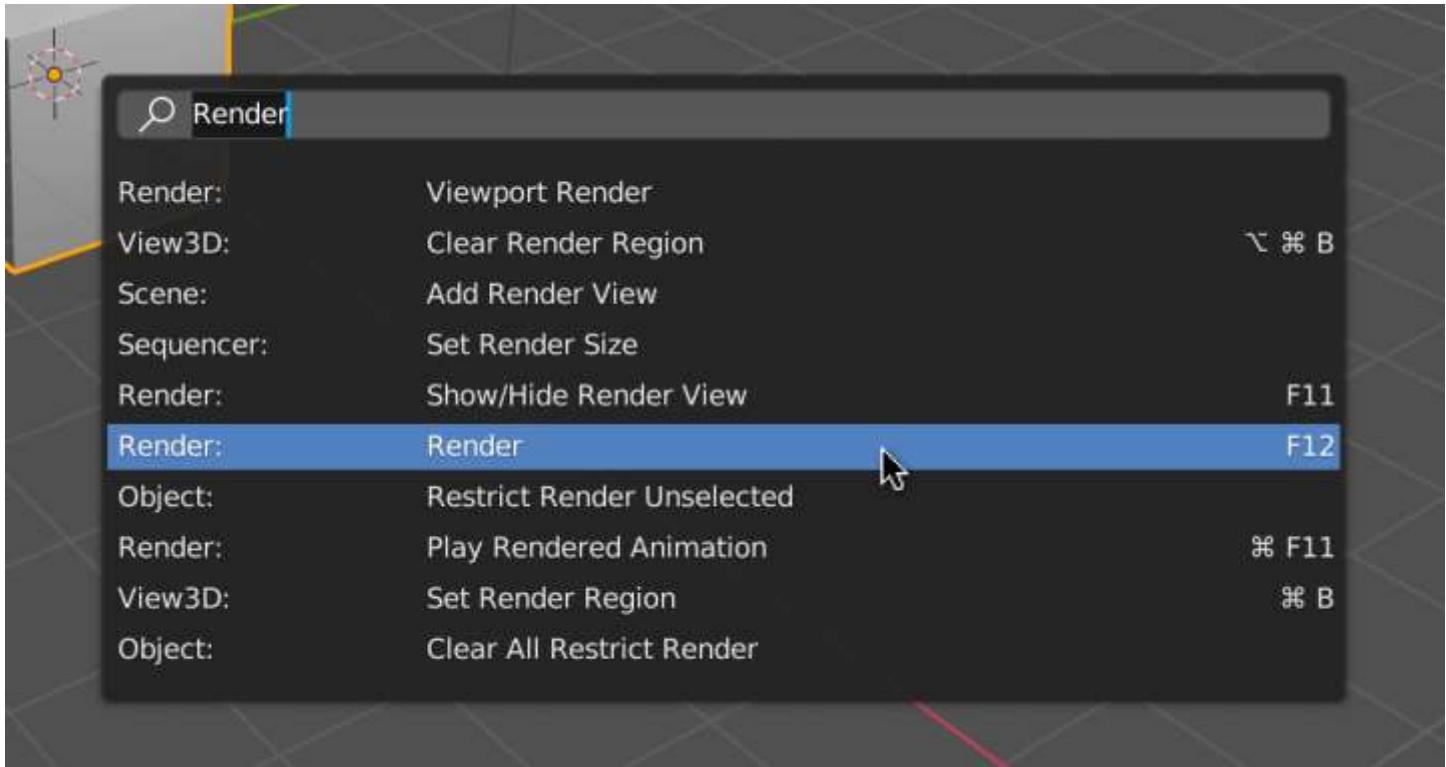
Si quieras renderizar la escena presiona “F12”. El render será visto desde la cámara, asegúrate de tener una cámara orientada a tu escena.



Búsqueda

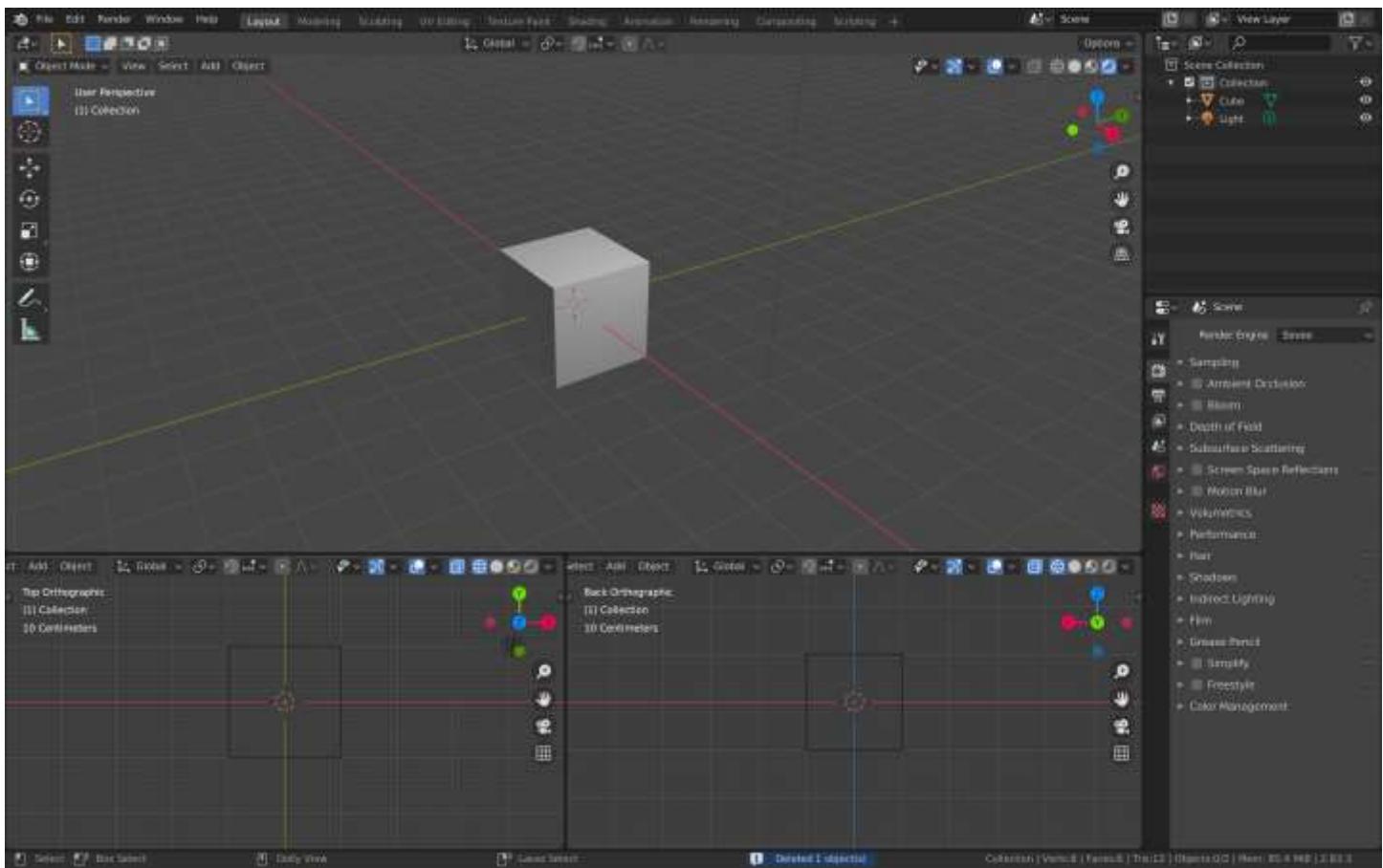
Hay muchas acciones posibles que no podemos recordar como lanzarlas todas. La solución correcta cuando no podemos encontrar donde se encuentra un botón o cual es el atajo de teclado, podemos usar el panel de búsqueda o “Search”.

Para abrir el panel de búsqueda, presiona la tecla “F3” (Podrías necesitar agregar la tecla “Fn” dependiendo de tu teclado y sistema operativo) y escribe el nombre de la acción:

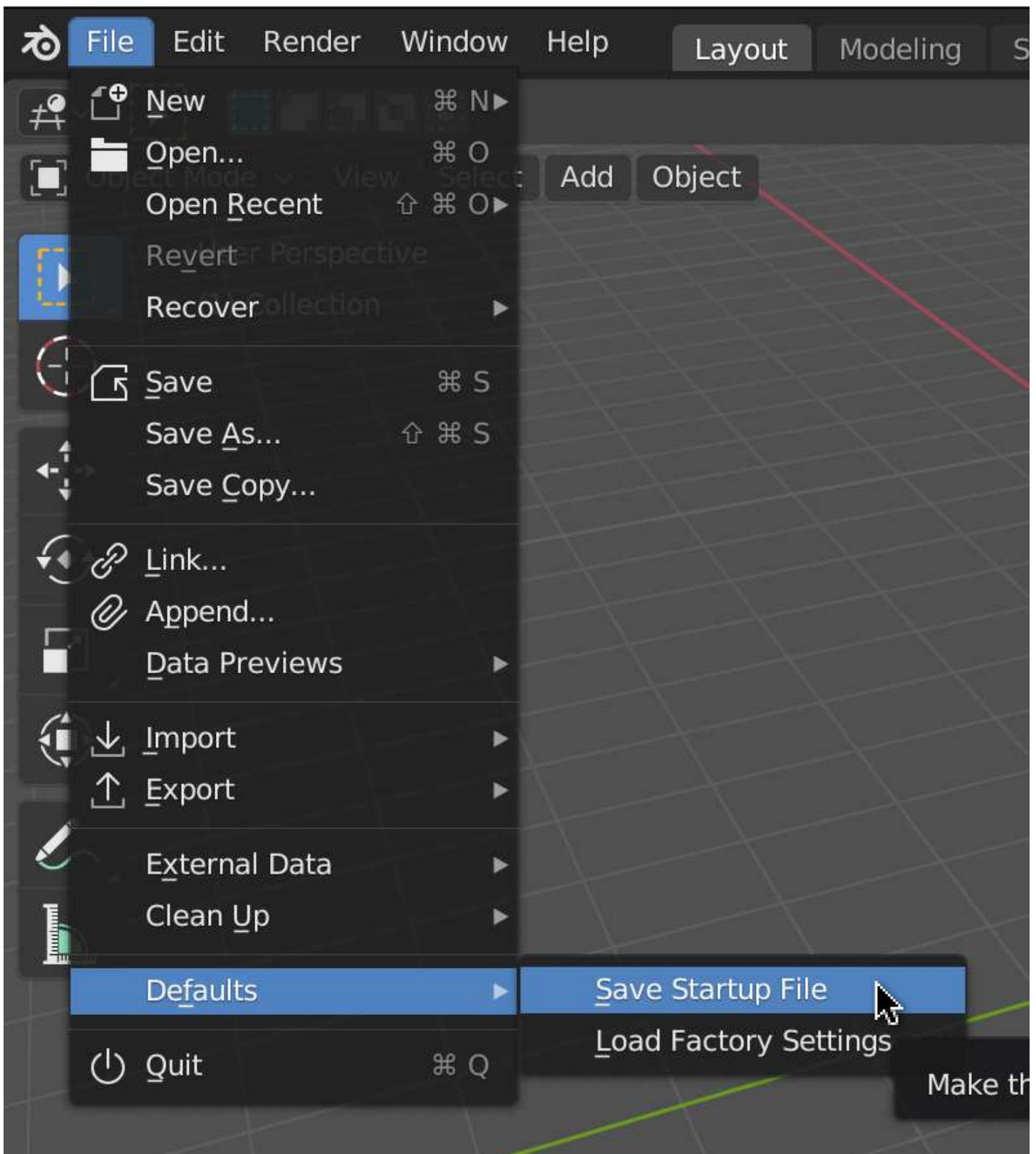


Guardar la configuración

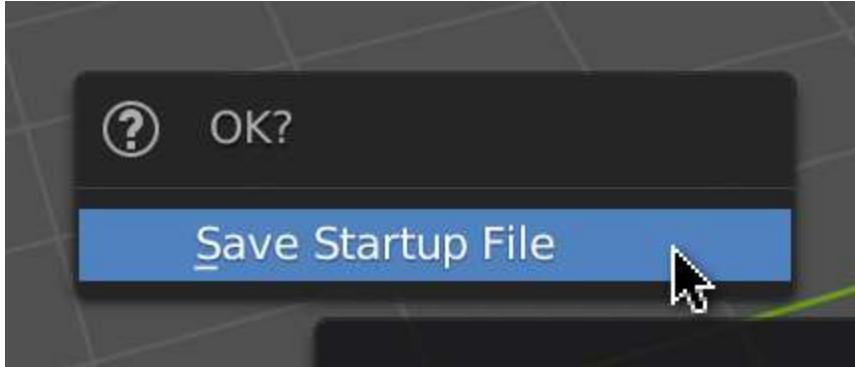
Crea una configuración sólida. Es libre a tu consideración, pero al usar Blender para exportar a Three.js no necesitamos una cámara. Podemos también crear dos vistas de lado debajo del principal "3D Viewport" en el eje "Z" y "Y" con el sombreado "Wireframe":



Una vez que estes conforme con tu configuración, ve a “File > Defaults > Save Startup File”. Esto va a guardar tu configuracion actual como la configuración por default cuando abras Blender:



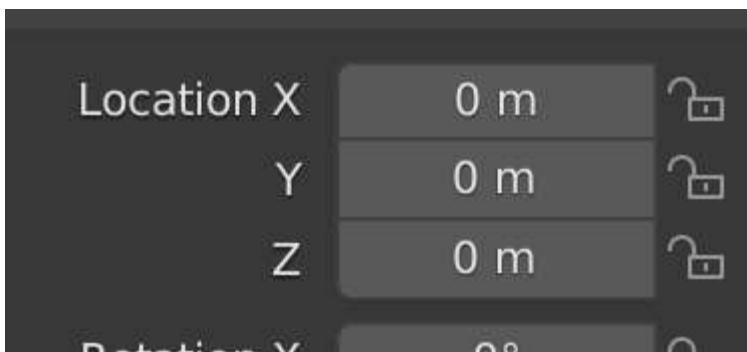
Se cuidadoso cuando des click en "Save Startup File"; un menú de confirmación debería abrirse y si mueves el mouse fuera de el lo perderás, da click de nuevo para confirmar:



Tiempo de hamburguesa

Es tiempo de crear nuestro propio modelo. En esta lección vamos a crear una hamburguesa e importarla en Three.js.

Como en Three.js, es una buena práctica decidir una unidad de escala. Si observas la cuadricula, un cubo ciertamente representa una unidad, y por defecto, Blender considera esa unidad como un metro:



Cuarto Botton bun

Cuarto Guardar

Cuarto Carne

Cuarto Queso

Cuarto Pan superior

Cuarto Toques finales

Cuarto Materiales

Exportar

Texto

Probar en Three.js

Texto

Ir mas allá

Texto

25. Mapas de Entorno

26. Renderizados realistas

27. Estructura de código para proyectos grandes

CAPITULO 4

28. Sombreados

29. Patrones de Sombreado

30. Mar furioso

31. Galaxia animada

32. Materiales modificados

CAPITULO 5

33. Postprocesado

34. Tips de desempeño

35. Introducción y proceso de carga

36. Mezclando HTML y WebGL

CAPITULO 6

37. Creando una escena en Blender

38. Cocinando y exportando la escena

39. Importando y optimizando la escena

40. Agregando detalles a la escena

CAPITULO 7

41. Que es React y React Three Fiber

42. Primera aplicacion en React

43. Primera aplicacion en R3F

44. Drei

45. Debug

46. Ambiente y puesta en escena

47. Cargar modelos

48. Texto 3D

49. Escena de portal

50. Eventos de mouse

51. Postprocesado

52. Portafolio divertido y simple

53. Fisica

INTRODUCCION

Ya aprendimos cómo agregar física a un proyecto Three.js y, como recordarás, fue bastante tedioso. Tuvimos que crear el mundo de la física por separado del mundo de Three.js y para cada objeto que queríamos agregar a la escena, teníamos que crear una versión en cada uno de los dos mundos.

En esta lección, veremos cuánto más fácil se vuelve este proceso cuando se usa R3F.

También ten en cuenta que esta lección es bastante extensa y puede resultar un poco aburrida porque tenemos muchos conceptos que cubrir. Pero no te preocupes, en la próxima lección vamos a poner este conocimiento en un ejercicio más divertido.

DEL CAÑOL A ESTOQUE

En la lección de Física anterior, usamos Cannon.js para manejar la parte de física. Cannon es una gran biblioteca de física, pero el código original no se ha actualizado en muchos años. Afortunadamente, el equipo de PMNDRS ha estado manteniendo una bifurcación del código, llamada `cannon-es` y también la han implementado en R3F (NPM, Github).

Si bien `cannon-es` es una solución perfectamente viable, hay un nuevo desafío y se llama Rapier.

RAPIER (ESTOQUE)

Rapier fue creado en 2019. Está escrito en Rust y funciona en JavaScript gracias a WebAssembly.

No vamos a entrar demasiado en detalles sobre WebAssembly, pero, en pocas palabras, WebAssembly permite ejecutar lenguajes como C/C++, C# y Rust en una página web con un rendimiento casi nativo.

En otras palabras, Rapier debería tener un gran rendimiento y no necesitamos aprender nada sobre WebAssembly o Rust. Simplemente podemos usarlo.

La biblioteca también es “determinismo”. De forma predeterminada, ejecutar la simulación con las mismas condiciones dará como resultado la misma animación, incluso en varios dispositivos (https://rapier.rs/docs/user_guides/javascript/determinism).

Rapier ha sido desarrollado por Dimforge:

<https://dimforge.com>

<https://twitter.com/dimforge>

Y la biblioteca de física funciona tanto para 2D como para 3D:

2D: <https://rapier.rs/demos2d/index.html>

3D: <https://rapier.rs/demos3d/index.html>

Rapier no está vinculado a Three.js y podemos usarlo con cualquier otra biblioteca.

ESTOQUE REACT THREE

Pero ¿qué pasa con Three.js y más específicamente con R3F?

La buena noticia es que R3F ya implementa Rapier como React Three Rapier gracias al equipo PMNDRS y más concretamente gracias a Hugo Wiledal (@etthugo).

React Three Rapier está funcionando a la perfección, pero tenga cuidado. Mientras escribo esta lección, todavía está en desarrollo y es posible que encuentres errores y funciones sin terminar. Estos deberían resolverse con el tiempo y es posible que ni siquiera los encuentre mientras sigue esta lección.

Aunque vamos a forzar una versión muy específica de la biblioteca para minimizar errores, esté atento a las posibles diferencias entre la lección y lo que tiene en su pantalla.

Si tiene dificultades, no dude en pedir ayuda en el servidor Three.js Journey Discord o incluso en el servidor PMNDRS.

COMO APRENDER

Vamos a aprender juntos Rapier con R3F, pero ya hay algunos recursos disponibles y usaremos la siguiente documentación como referencia.

Rapier (sin Three.js ni R3F)

- Documentación: https://rapier.rs/docs/user_guides/javascript/getting_started_js

- API: <https://rapier.rs/javascript3d/index.html>

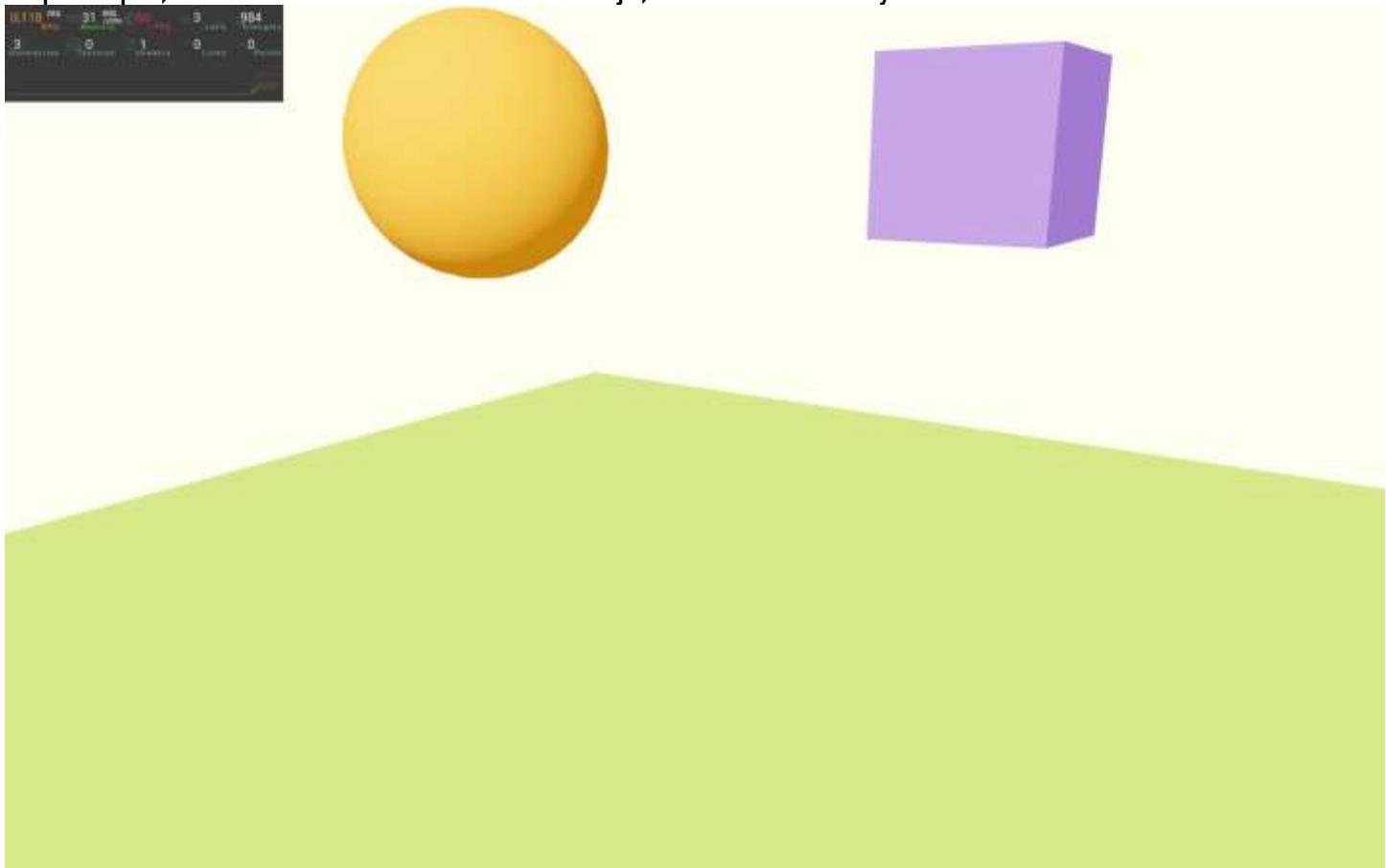
En cuanto a aprender Rapier en R3F, dado que la biblioteca está en desarrollo, aún no hay documentación completa disponible. Aún así, tenemos acceso a ejemplos:

- Léame: <https://github.com/pmnndrs/react-tres-rapier#readme>
- Ejemplos sobre React Three Fiber: <https://docs.pmnnd.rs/react-tres-fiber/getting-started/examples> (busque "rapier")

De todos modos, aprenderemos la mayor parte de lo que necesitaríamos en esta lección.

CONFIGURACION

Al principio, tenemos la clásica esfera naranja, el cubo morado y el suelo verde:



Como puedes ver, la esfera y el cubo están un poco más altos de lo habitual como si estuvieran a punto de caer.

El suelo también es más grueso. Los motores de física no son muy buenos cuando se trata de detectar colisiones con objetos delgados.

También disponemos de una fuente de luz direccional y una fuente de luz ambiental.

La dependencia `@react-tres/drei` ya está instalada dentro del proyecto y estamos usando el asistente `OrbitControls` para poder mover la cámara.

También tenemos `<Perf />` de `r3f-perf` para controlar el rendimiento.

IMPLEMENTANDO

Para agregar React Three Rapier al proyecto, desde la terminal, ejecute `npm install @react-tres/rapier@1.0` (forzamos la versión para evitar sorpresas; puede ignorar posibles advertencias de vulnerabilidad).

Haremos que la física funcione con solo unos pocos cambios en nuestro código. Será bastante rápido, pero entraremos en detalles justo después de esta parte.

FISICAS

Primero, en `Experience.jsx`, necesitamos importar `physics` desde `@react-tres/rapier`:

```
import { Physics } from '@react-three/rapier'
```

Ahora necesitamos agregarlo en nuestro JSX.

Dado que solo los objetos dentro de la etiqueta `<Physics>` se verán afectados por la física, la agregaremos alrededor de nuestras 3 `<mesh>`:

```
export default function Experience()
{
  return <>

    {/* ... */}

    <Physics>

      <mesh castShadow position={ [ - 2, 2, 0 ] }>
        {/* ... */}
      </mesh>

      <mesh castShadow position={ [ 2, 2, 0 ] }>
        {/* ... */}
      </mesh>

      <mesh receiveShadow position-y={ - 1.25 }>
        {/* ... */}
      </mesh>

    </Physics>

  </>
}
```

<RigidBody> (CUERPO RIGIDO)

Ahora, necesitamos especificar qué objetos se verán afectados por la física. Para hacer eso, necesitamos agregar un <RigidBody> alrededor de cada objeto.

Primero, importe `RigidBody` desde `@react-tres/rapier`:

```
import { RigidBody, Physics } from '@react-three/rapier'
```

Y ahora, podemos agregar el <RigidBody> alrededor de nuestros objetos que queremos que estén sujetos a la física.

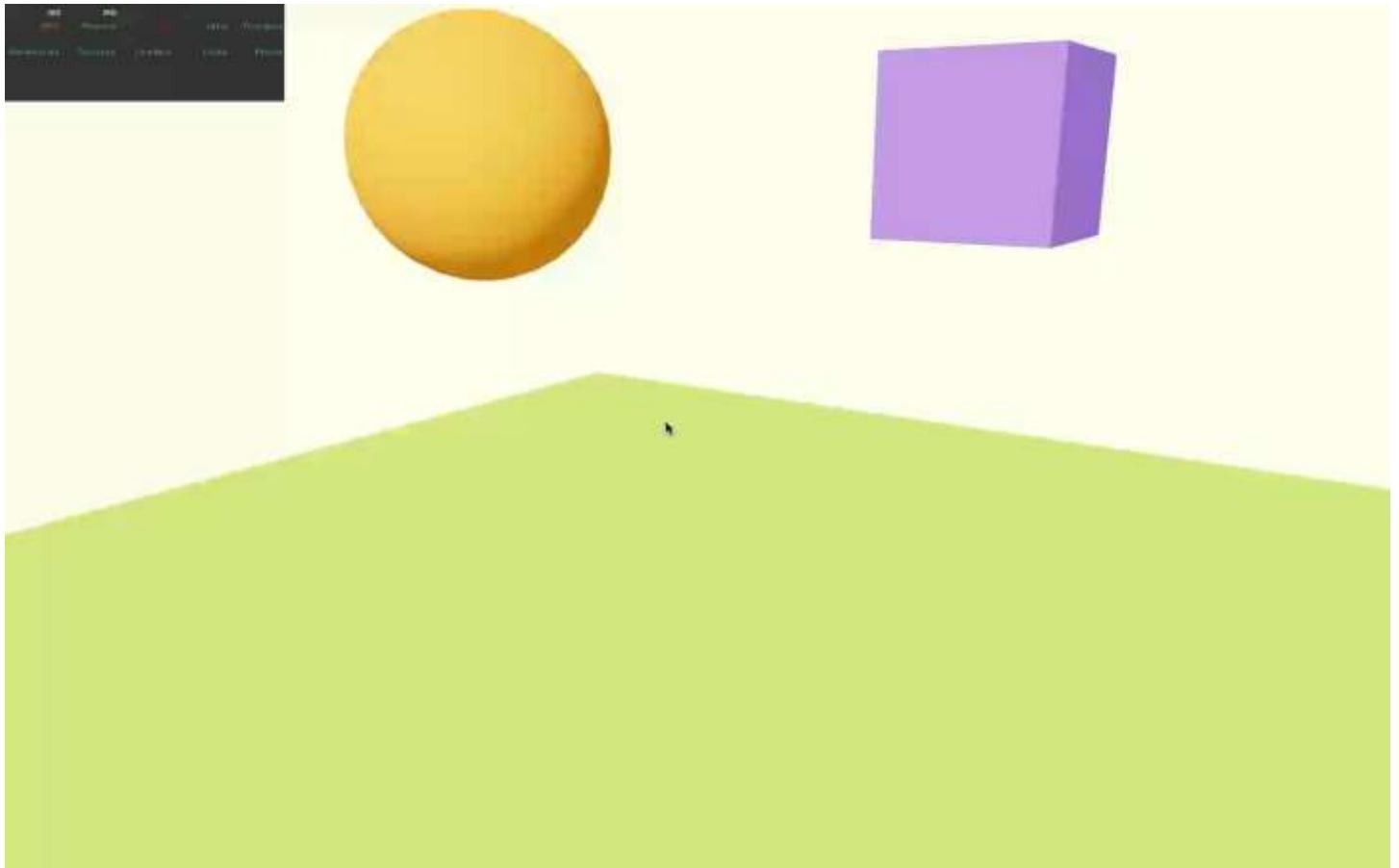
Empecemos por la esfera:

```
<Physics>

  <RigidBody>
    <mesh castShadow position={ [ - 2, 2, 0 ] }>
      <sphereGeometry />
      <meshStandardMaterial color="orange" />
    </mesh>
  </RigidBody>

  {/* ... */}

</Physics>
```



Y tada, la esfera está cayendo.

Pero obviamente, atraviesa el piso ya que no agregamos física a nuestro piso.

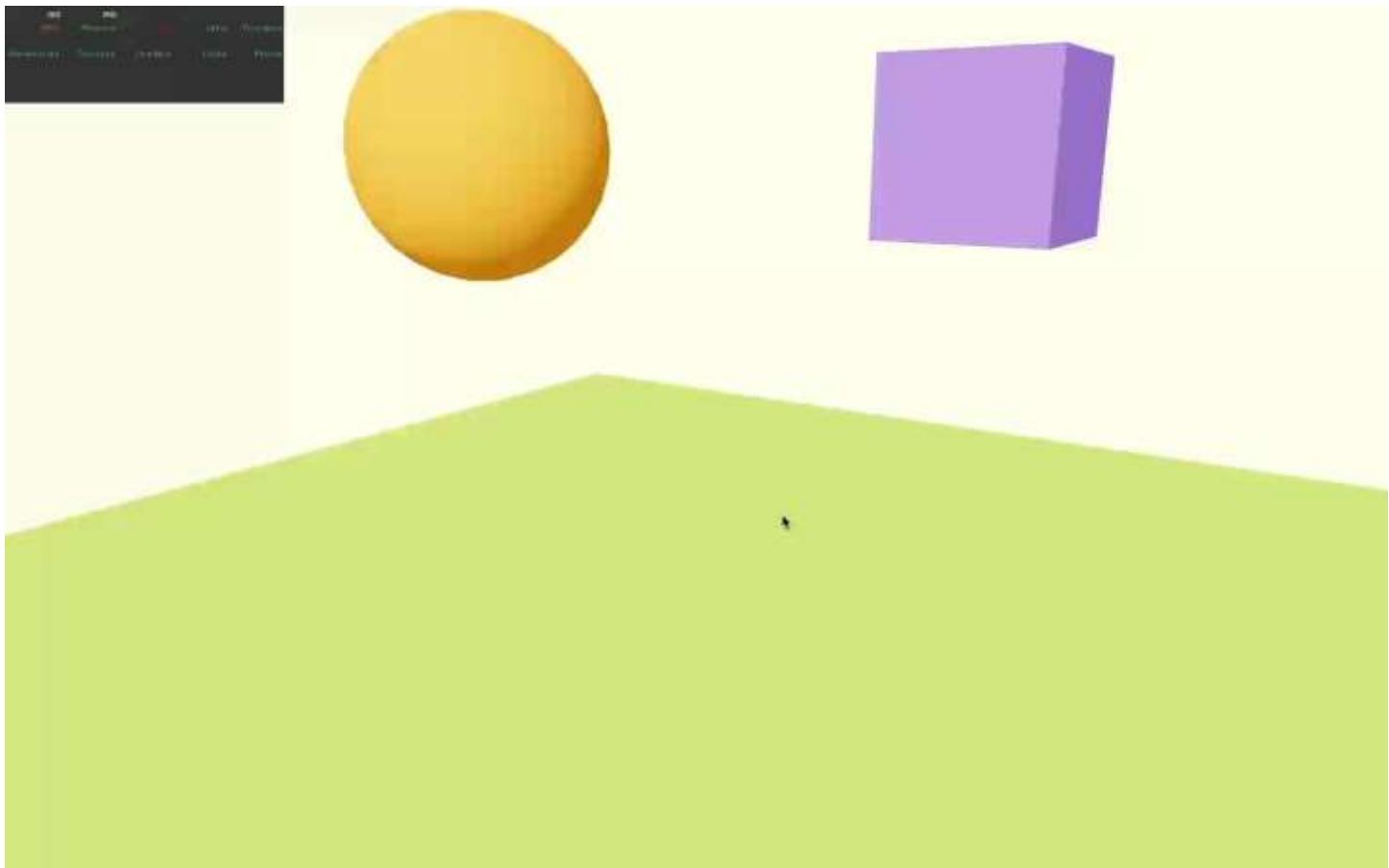
Arreglemos eso y agreguemos un `<RigidBody>` alrededor del piso, pero también agregaremos un atributo `type` con el valor `fixed` adjunto:

```
<Physics>

    /* ... */

    <RigidBody type="fixed">
        <mesh receiveShadow position-y={ - 1.25 }>
            <boxGeometry args={ [ 10, 0.5, 10 ] } />
            <meshStandardMaterial color="greenyellow" />
        </mesh>
    </RigidBody>

</Physics>
```



Tenga en cuenta que un `<RigidBody>` solo se puede agregar dentro del elemento `<Physics>`. De lo contrario se producirá un error.

DEBAJO DE LA CAPUCHA

Como puedes ver, agregar física fue realmente fácil.

Aquí hay algunas cosas a tener en cuenta:

No tenemos que actualizar la física en cada cuadro.

Los objetos Three.js se asocian automáticamente con el `RigidBody` correspondiente que los rodea.

React Three Rapier crea una forma física que parece coincidir con los objetos Three.js.

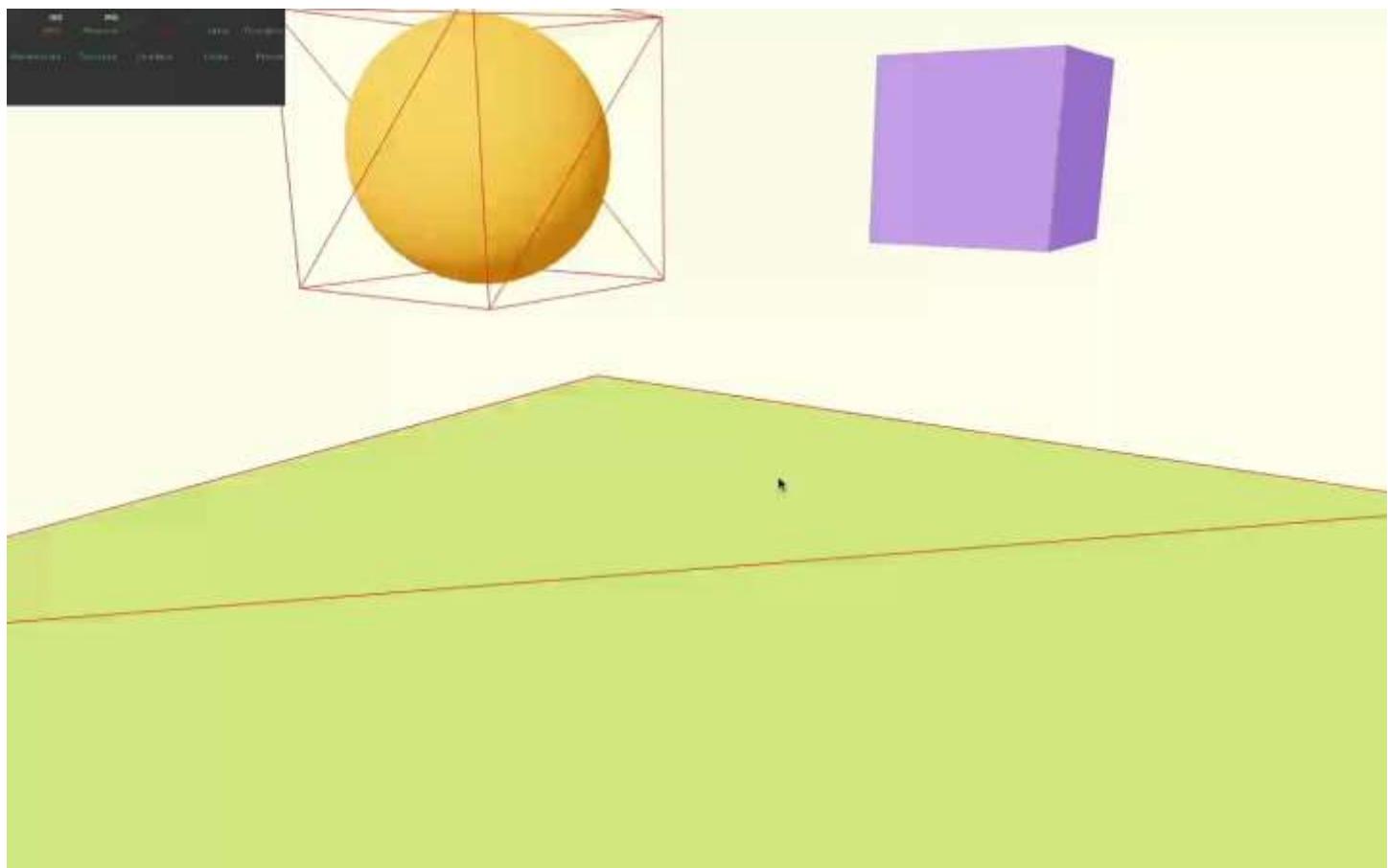
No tenemos que especificar cosas como propiedades de la superficie, masa del objeto, gravedad, etc.

De ahora en adelante, es posible que tengas que volver a cargar la página después de realizar cambios. Actualizar el mundo de la física sobre la marcha es una tarea bastante compleja y Hot Module Reload aún no es perfectamente compatible.

DEPURANDO

Para ver lo que realmente está sucediendo en el mundo de la Física, vamos a agregar un modo de depuración y es tan simple como agregar el atributo `debug` a `<Physics>`.

```
<Physics debug>  
  
 {/* ... */}  
  
</Physics>
```



Como puedes ver, la versión física de nuestra esfera es en realidad una caja. No te preocupes, vamos a cambiar eso.

Este modo de depuración suele ser sólo para desarrollo. Puede optar por eliminarlo cuando ya no lo necesite o puede hacerlo opcional si está utilizando una interfaz de usuario de depuración como [Leva](#).

Finalmente, esos wireframes parecen inofensivos, pero en realidad pueden tener un gran impacto en el rendimiento. Un poco más adelante en la lección, haremos una prueba de esfuerzo y agregaremos cientos de objetos físicos. Vamos a optimizar esos objetos para renderizarlos rápidamente, pero las versiones de depuración no se optimizarán.

COLISIONADORES

Los colisionadores son las formas que forman nuestros **RigidBodies**. En el caso de nuestra esfera, puedes ver que el colisionador parece ser una caja. En realidad, es más bien un cubo, ya que todos

los lados tienen el mismo tamaño y la palabra real utilizada por Rapier para esta forma es "cuboide".

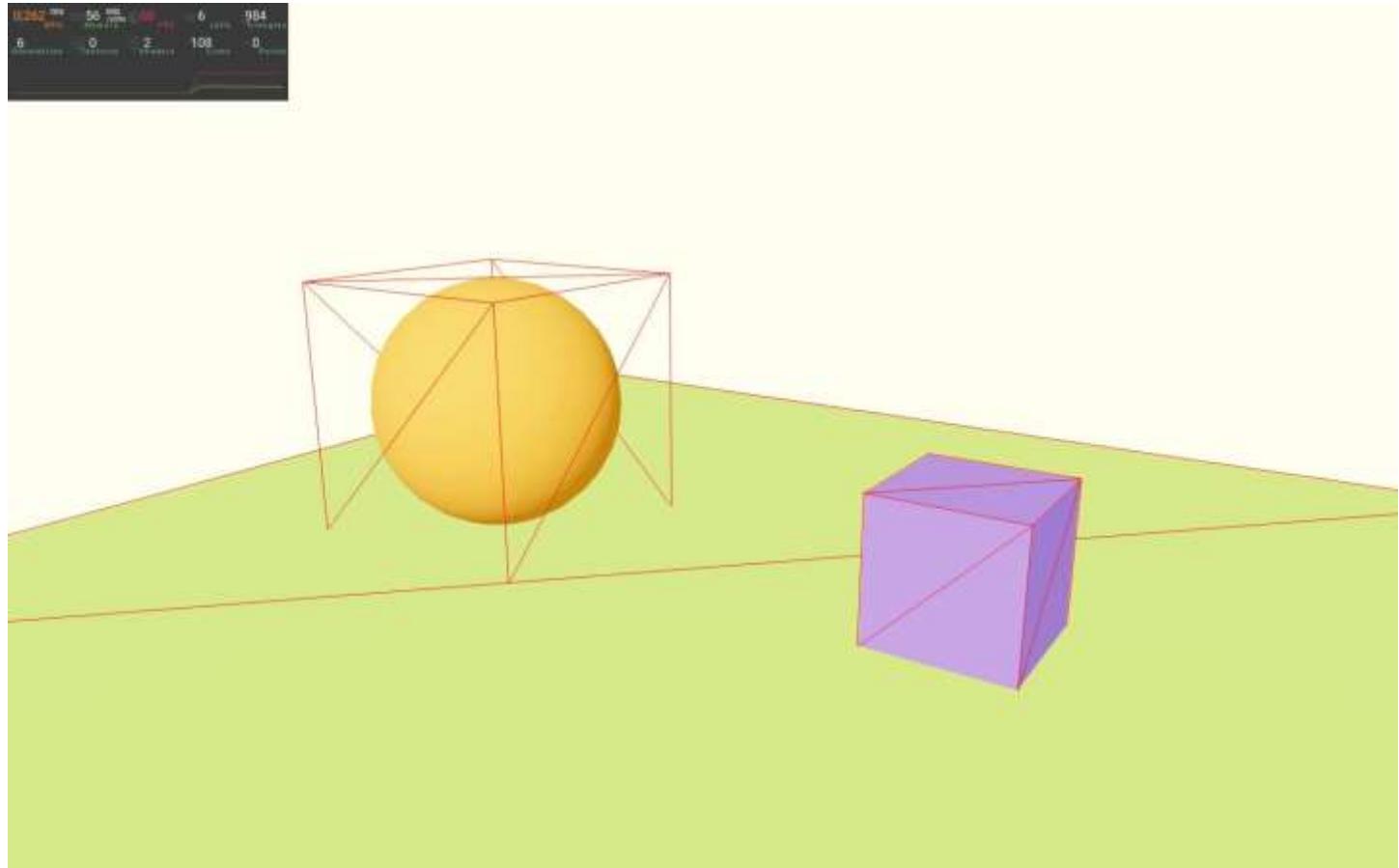
COLISIONADORES AUTOMATICOS

COLISIONADOR CUBOIDE

Este cuboide fue generado automáticamente por React Three Rapier.

Probemos con el cubo morado. Envuelva el cubo <mesh> en un <RigidBody>:

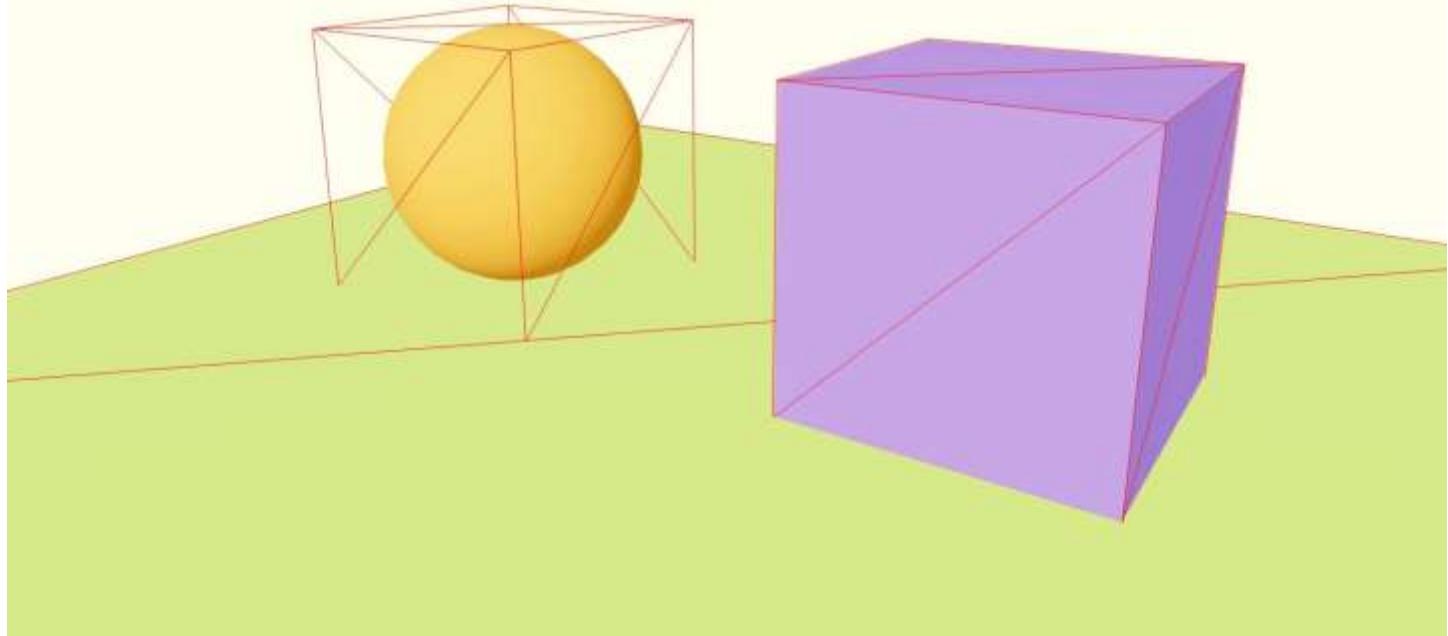
```
<RigidBody>
  <mesh castShadow position={ [ 2, 2, 0 ] }>
    <boxGeometry />
    <meshStandardMaterial color="mediumpurple" />
  </mesh>
</RigidBody>
```



Como puedes ver, este nuevo cuboide combina perfectamente con nuestro cubo original.

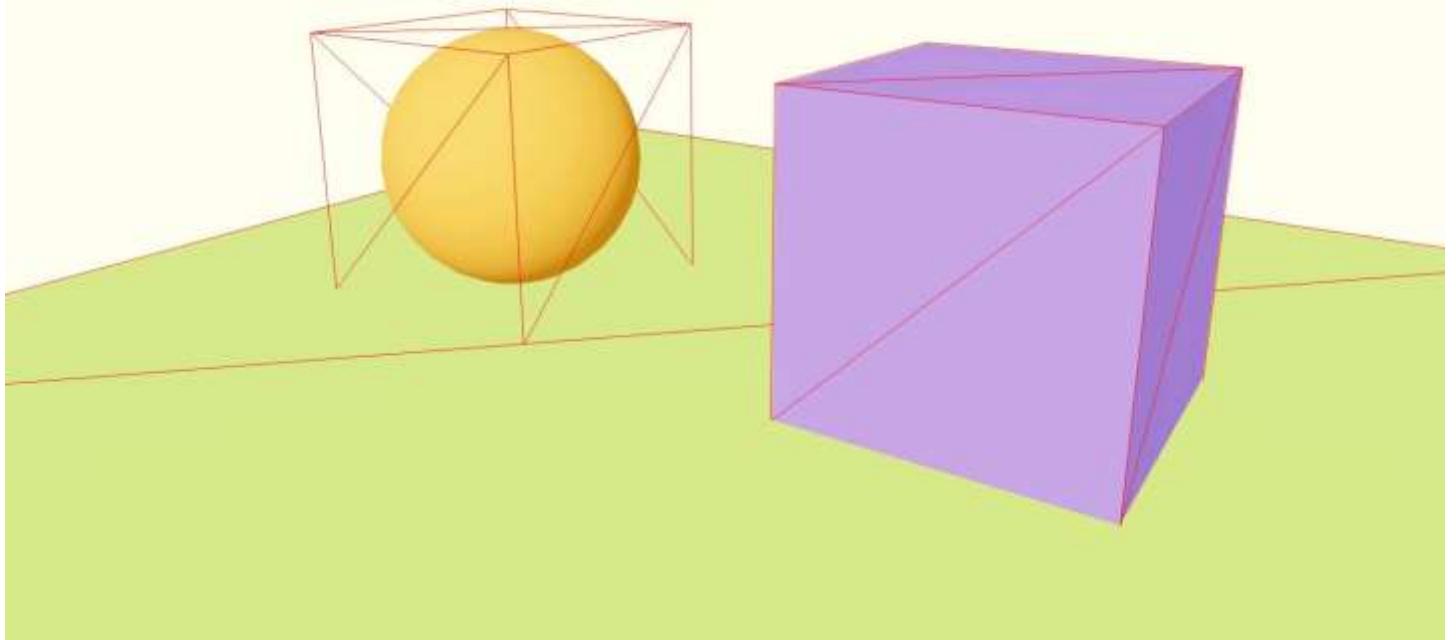
Cambie el tamaño de <mesh> usando el atributo de **scale** para ver si el colisionador coincide:

```
<RigidBody>
  <mesh castShadow position={ [ 2, 2, 0 ] } scale={ 2 }>
    <boxGeometry args={ [ 2, 2, 2 ] } />
    <meshStandardMaterial color="mediumpurple" />
  </mesh>
</RigidBody>
```



E incluso podemos probar `<boxGeometry>` en lugar de cambiar la `scale` en `<mesh>`:

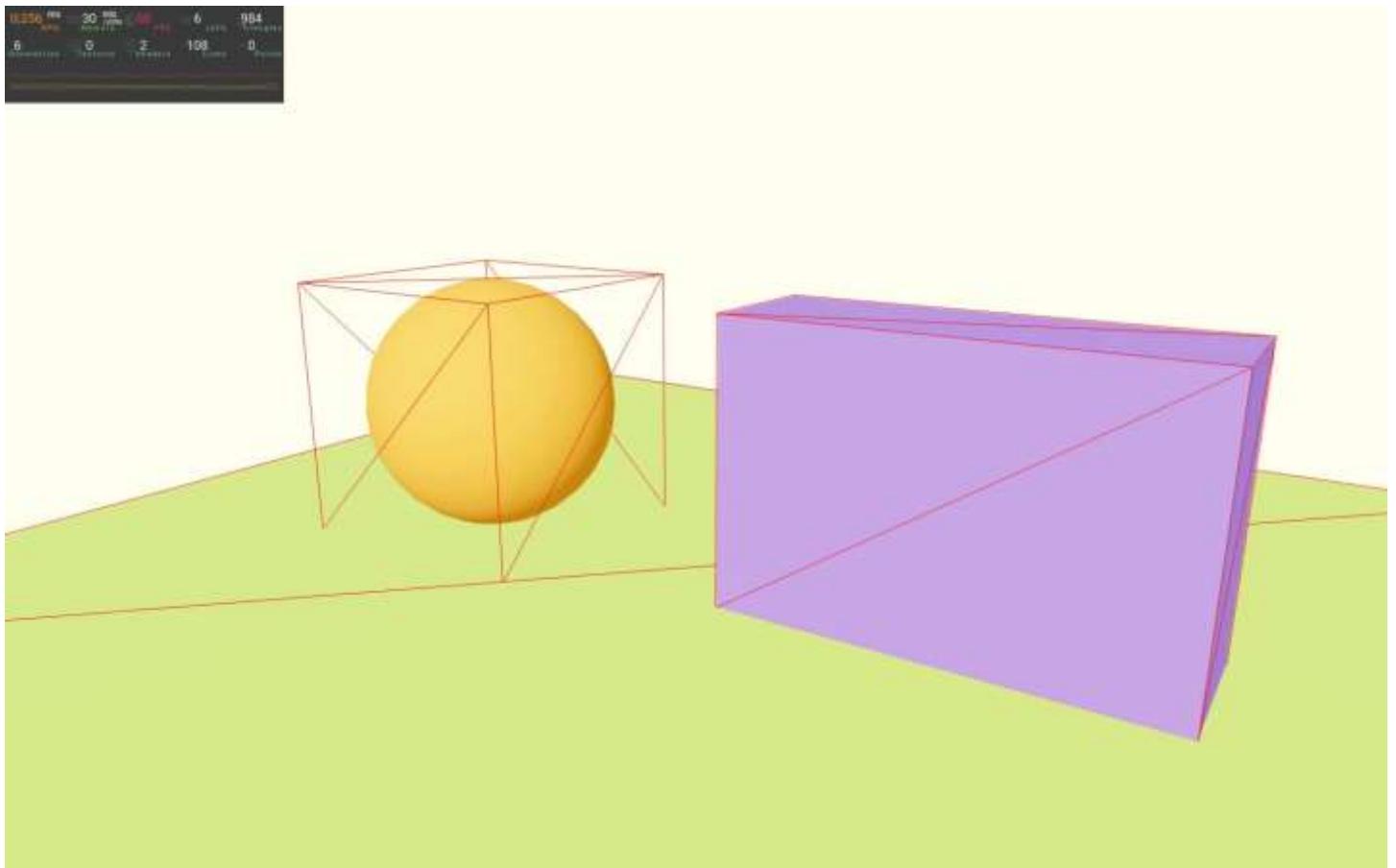
```
<RigidBody>
  <mesh castShadow position={ [ 2, 2, 0 ] }>
    <boxGeometry args={ [ 2, 2, 2 ] } />
    <meshStandardMaterial color="mediumpurple" />
  </mesh>
</RigidBody>
```



El colisionador parece encajar siempre.

¿Pero qué pasa si enviamos una forma no cúbica?

```
<RigidBody>
  <mesh castShadow position={ [ 2, 2, 0 ] }>
    <boxGeometry args={[ [ 3, 2, 1 ] ]} />
    <meshStandardMaterial color="mediumpurple" />
  </mesh>
</RigidBody>
```

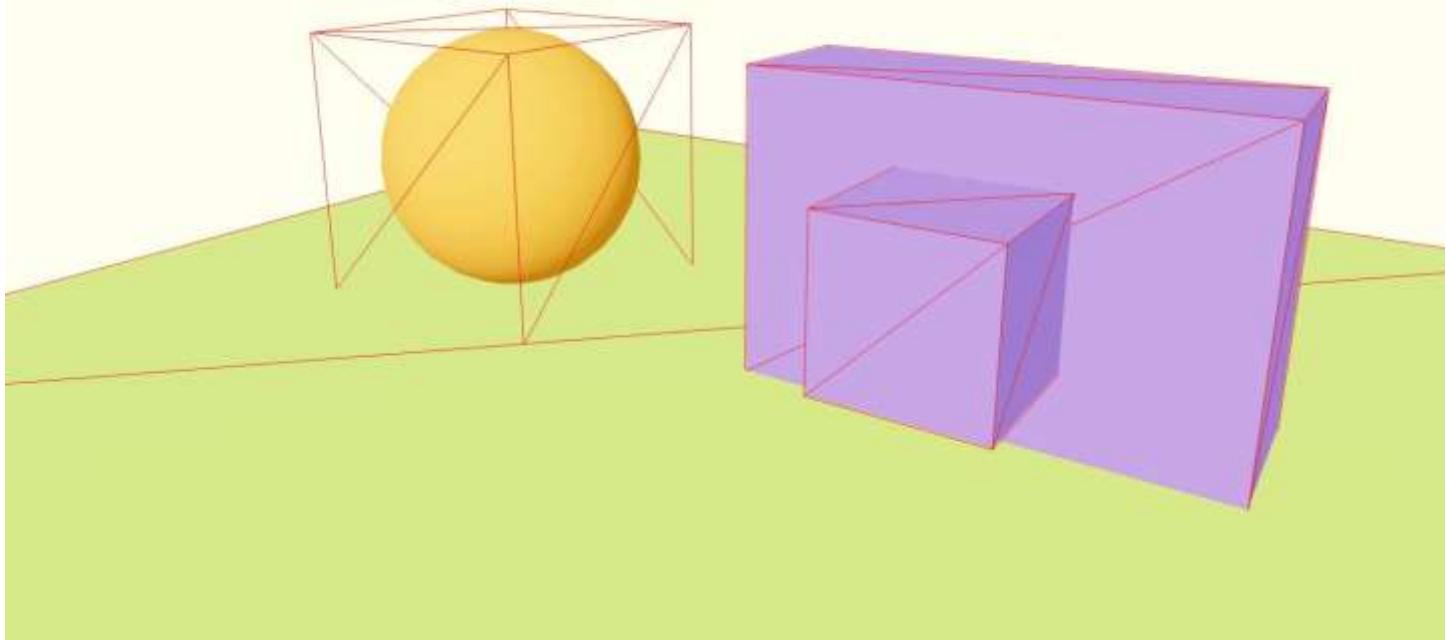
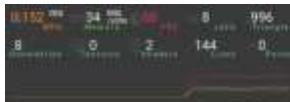


Sí, todavía está funcionando.

OBJETOS COMPUESTOS

Intentemos romper React Three Rapier y agregar una segunda `<mesh>` dentro del `<RigidBody>` del cuadro morado:

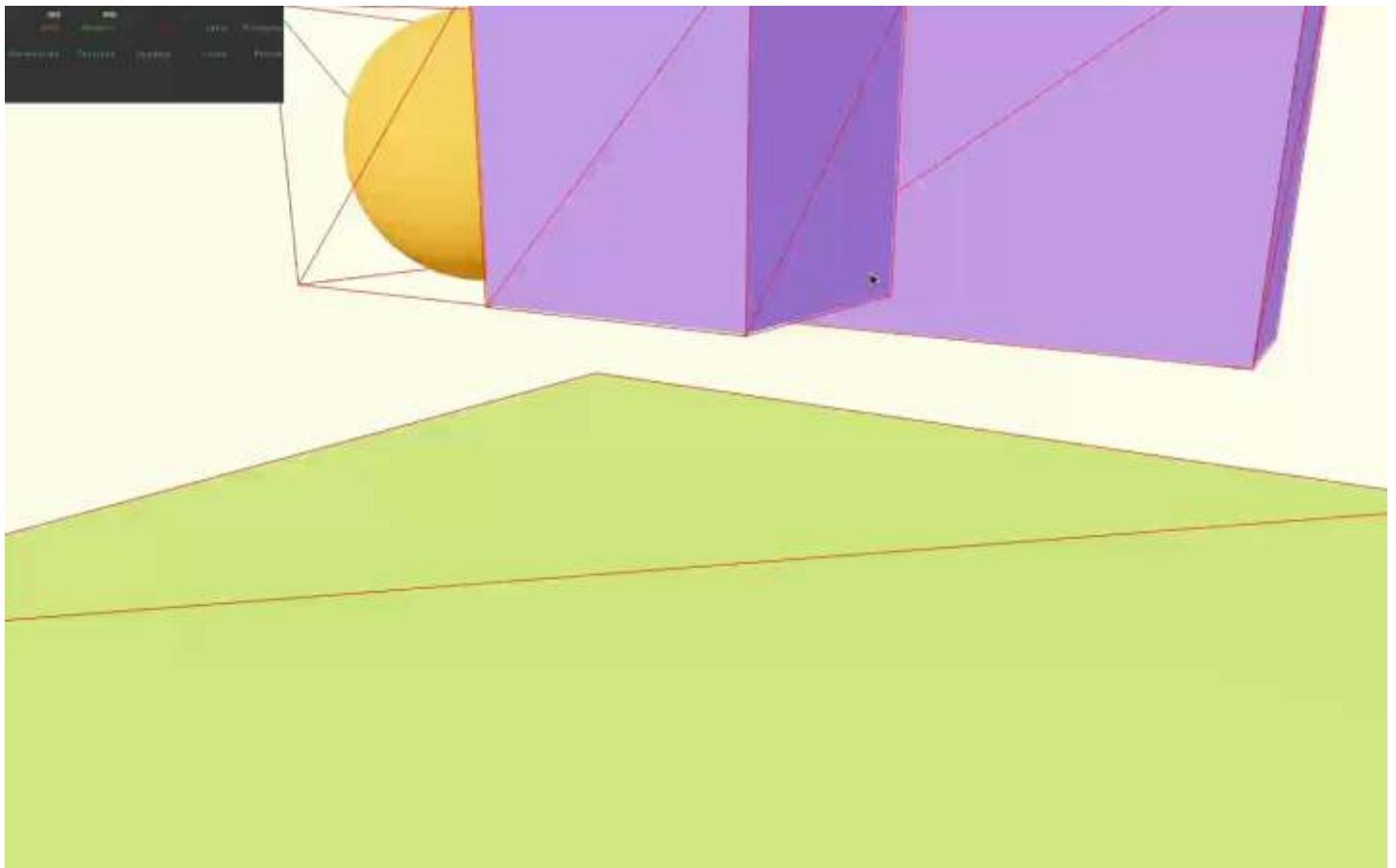
```
<RigidBody>
  <mesh castShadow position={ [ 2, 2, 0 ] }>
    <boxGeometry args={ [ 3, 2, 1 ] } />
    <meshStandardMaterial color="mediumpurple" />
  </mesh>
  <mesh castShadow position={ [ 2, 2, 1 ] }>
    <boxGeometry args={ [ 1, 1, 1 ] } />
    <meshStandardMaterial color="mediumpurple" />
  </mesh>
</RigidBody>
```



Como puede ver, React Three Rapier creó un segundo colisionador para la segunda `<mesh>`. Todavía tenemos un `RigidBody`, pero está compuesto por múltiples colisionadores.

Ahora intenta alejar un poco más este nuevo cubo pequeño del cuadro inicial:

```
<RigidBody>
  <mesh castShadow position={ [ 2, 2, 0 ] }>
    <boxGeometry args={ [ 3, 2, 1 ] } />
    <meshStandardMaterial color="mediumpurple" />
  </mesh>
  <mesh castShadow position={ [ 2, 2, 3 ] }>
    <boxGeometry args={ [ 1, 1, 1 ] } />
    <meshStandardMaterial color="mediumpurple" />
  </mesh>
</RigidBody>
```



Puede ver que los colisionadores no necesitan estar unidos físicamente para funcionar como un cuerpo complejo. Agregar algo de peso fuera del cuerpo también hace que el objeto caiga de costado porque el centro de masa cambia. Como en la vida real.

BALLON COLISIONADOR

Volvamos a nuestro ámbito.

Debido a que el colisionador predeterminado es un cuboide, no coincide con la geometría de nuestra esfera.

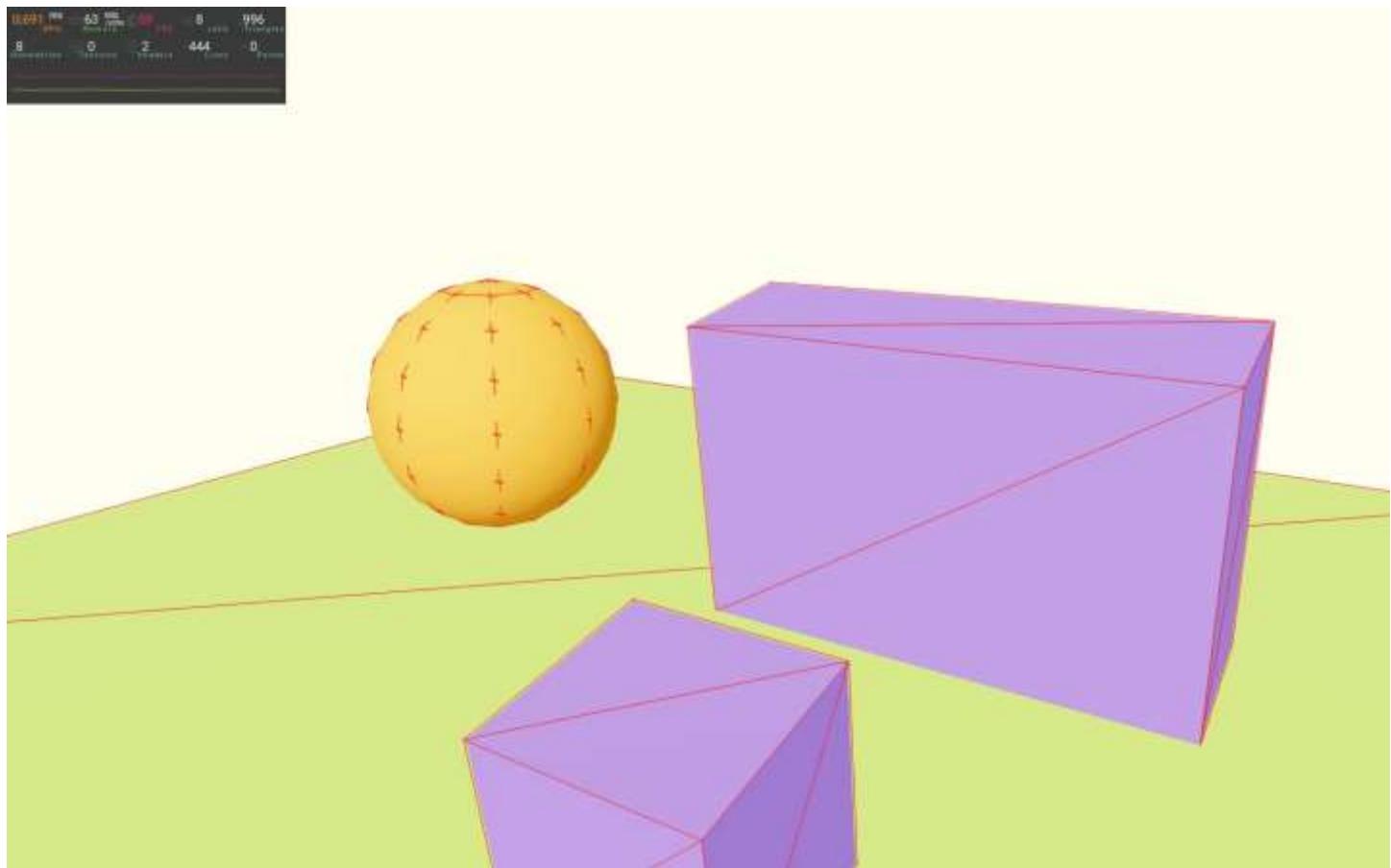
En realidad, no es necesariamente gran cosa y, a veces, no es necesario tener un colisionador preciso. La gente está acostumbrada a tener colisionadores que no coinciden exactamente con el modelo, especialmente los jugadores. La mayoría de los objetos en mi portafolio (<https://bruno-simon.com>) tienen una versión física que solo se compone de cajas y en realidad es mejor para el rendimiento que las formas precisas.

Aún así, intentemos solucionarlo.

Puede decidir qué colisionador automático desea utilizar en `<Rigidbody>` con el atributo `colliders`. El predeterminado es `cuboide`, pero podemos configurarlo como `ball`.

Agregue el atributo `colliders` (con una s) al `<Rigidbody>` de la esfera y configúrelo en `ball`:

```
<RigidBody colliders="ball">
  <mesh castShadow position={ [ - 2, 2, 0 ] }>
    <sphereGeometry />
    <meshStandardMaterial color="orange" />
  </mesh>
</RigidBody>
```



Y como puede ver en la versión de depuración, ahora tenemos una forma física que coincide mucho con nuestra esfera.

CASCARA COLISIONADORA

Primero, retira el cubo **<RigidBody>** y lo que hay dentro.

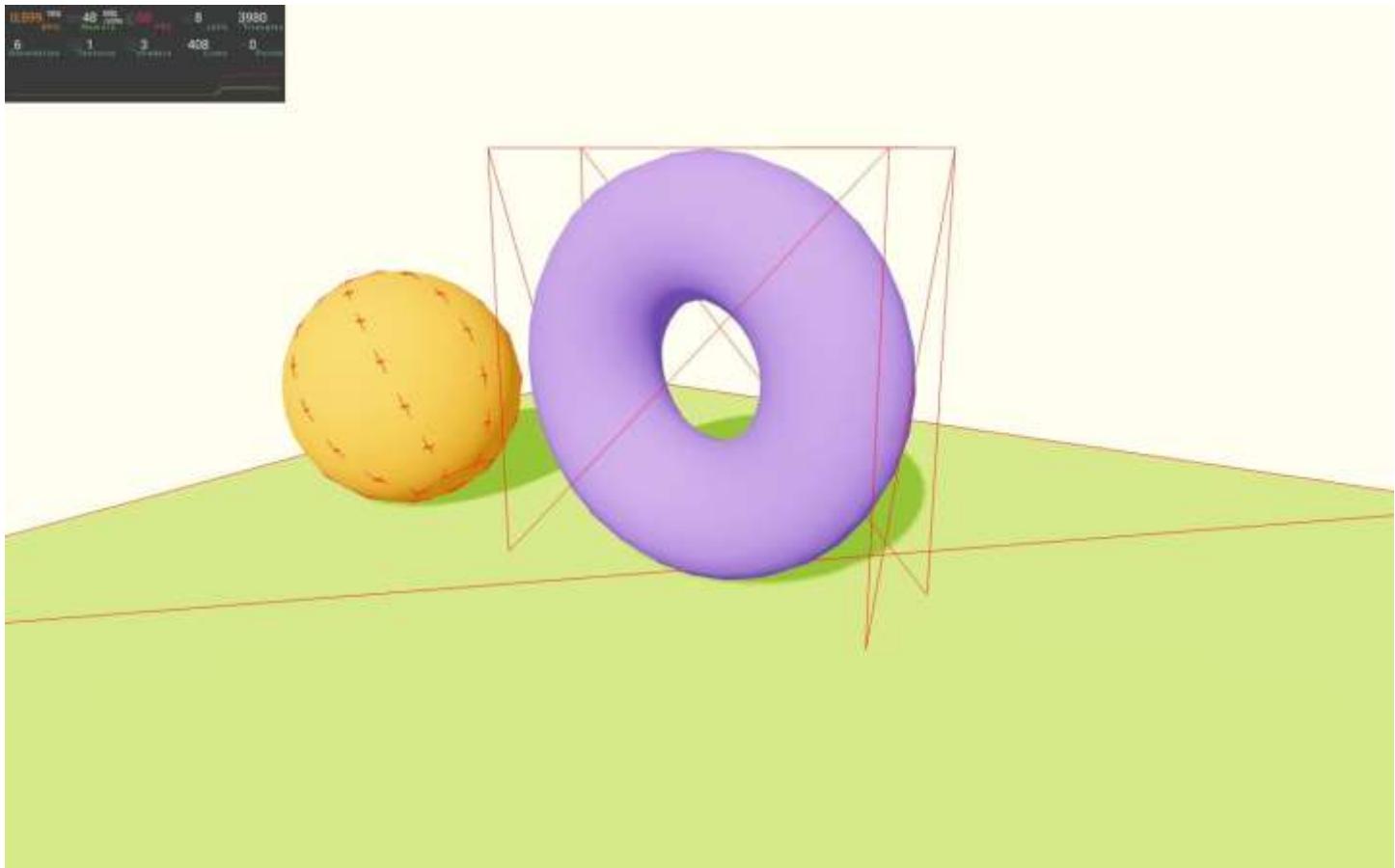
Hagamos las cosas un poco más complicadas. Ahora sería una buena oportunidad para intentar seguir las instrucciones usted mismo:

Agrega un toroide (o donut) con física. Coloca ese toroide en el centro de la escena, plano sobre el suelo y haz que la bola caiga sobre el toroide, desde arriba.

Hagámoslo juntos.

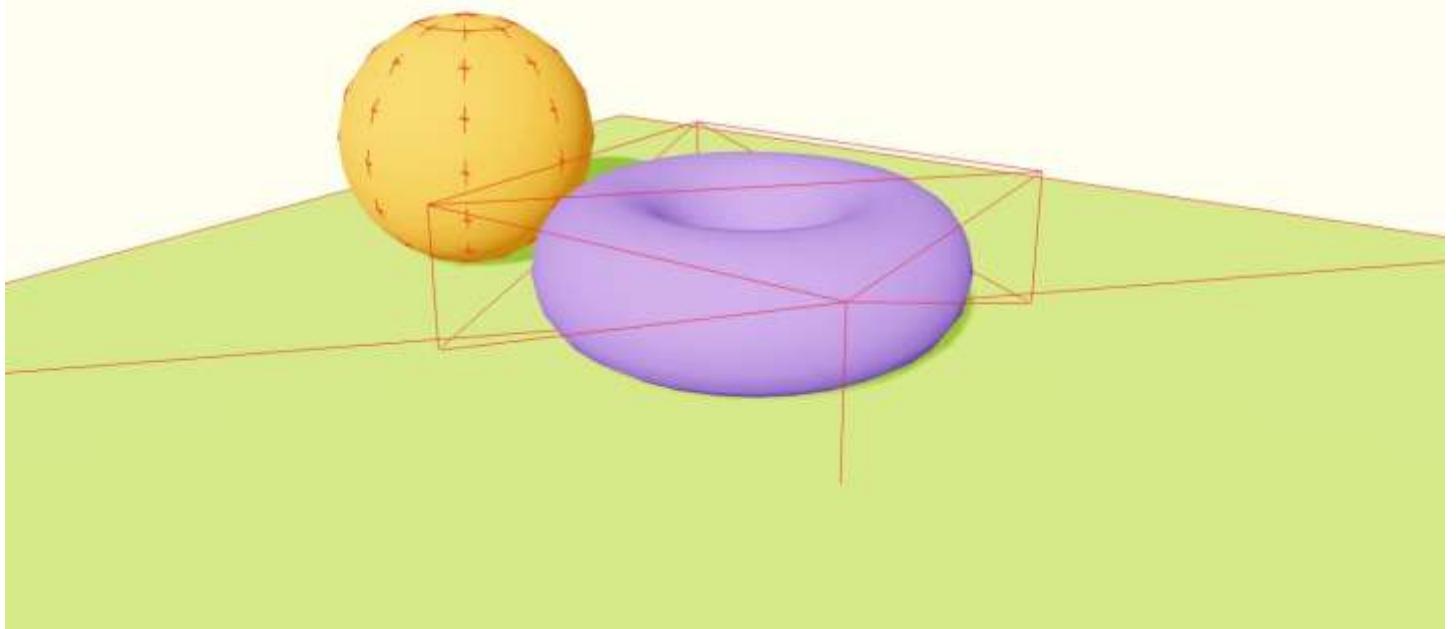
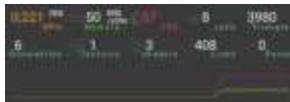
Primero, cree un **<RigidBody>** con el toroide dentro (puede encontrar los parámetros de geometría del toroide en la documentación de Three.js):

```
<RigidBody>
  <mesh castShadow position={ [ 0, 1, 0 ] }>
    <torusGeometry args={ [ 1, 0.5, 16, 32 ] } />
    <meshStandardMaterial color="mediumpurple" />
  </mesh>
</RigidBody>
```



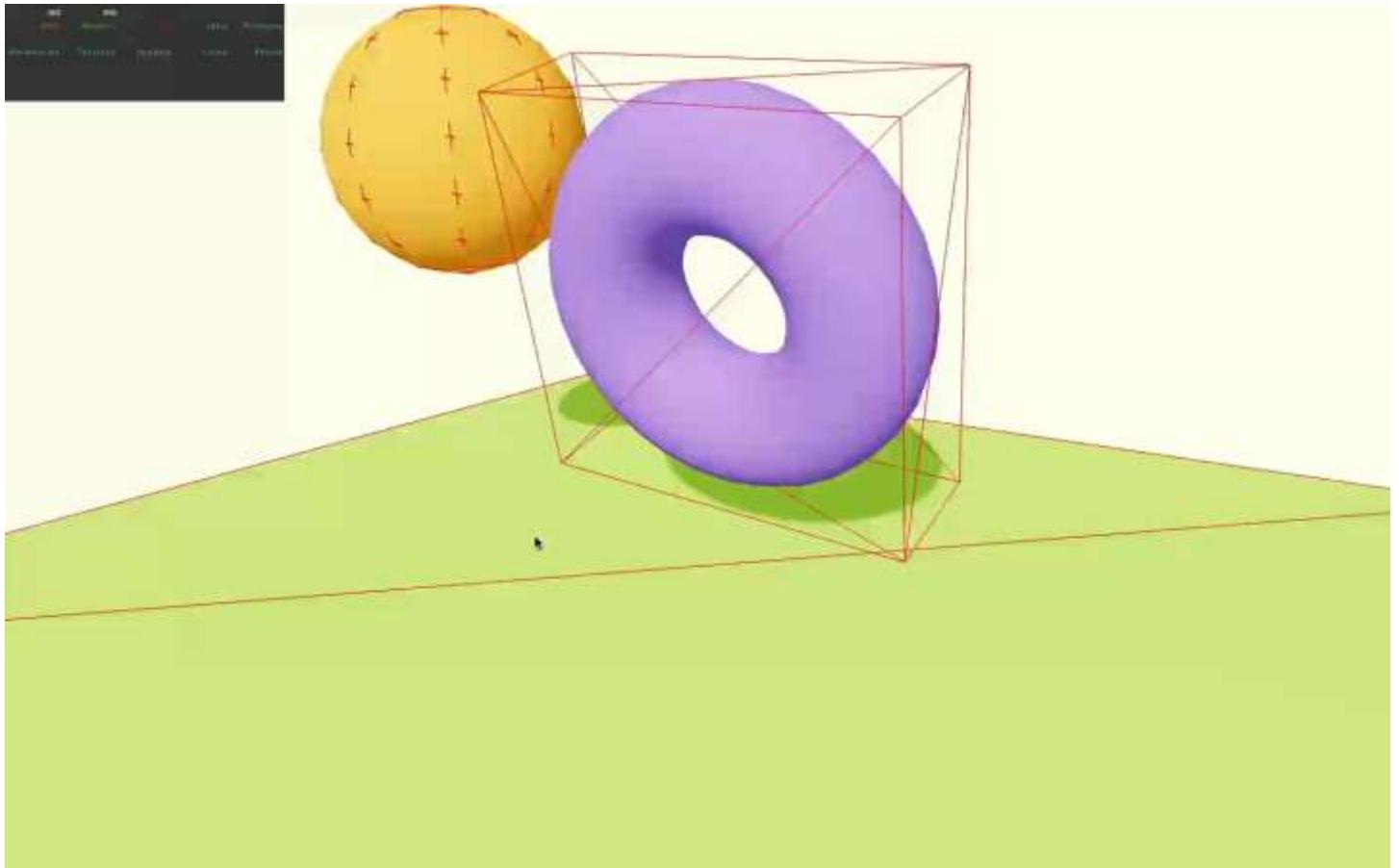
Gire la <mesh>:

```
<RigidBody>
  <mesh castShadow position={ [ 0, 1, 0 ] } rotation={ [ Math.PI * 0.5, 0, 0 ] }>
    <torusGeometry args={ [ 1, 0.5, 16, 32 ] } />
    <meshStandardMaterial color="mediumpurple" />
  </mesh>
</RigidBody>
```



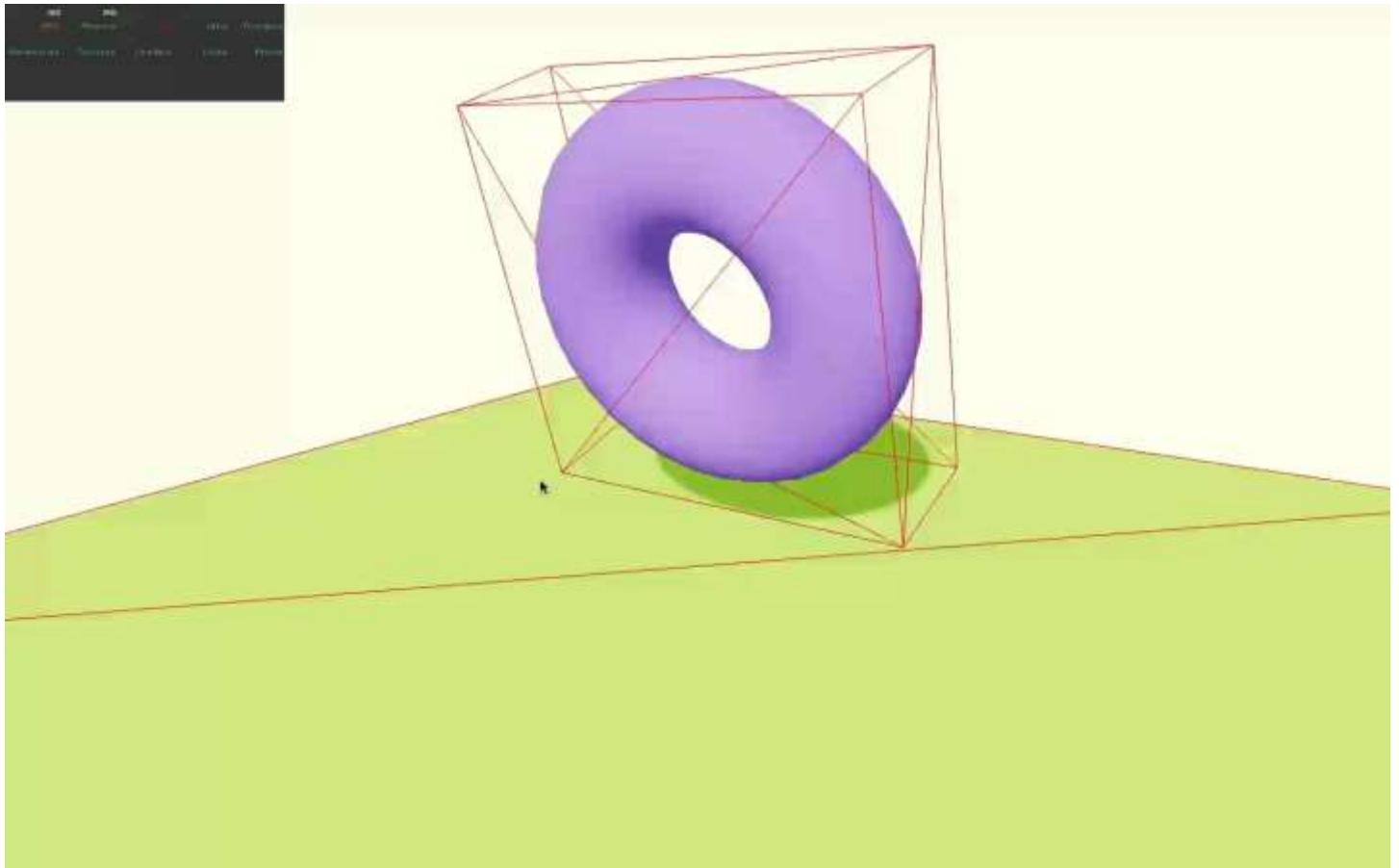
En lugar de dar media vuelta, reduzcamos el ángulo y movamos el toroide hacia el fondo. De esta manera, podemos probar si el toro cae de manera realista. Como resultado, la esfera debería caer justo encima de ella:

```
<RigidBody>
  <mesh castShadow position={ [ 0, 1, - 0.25 ] } rotation={ [ Math.PI * 0.1, 0, 0 ]
}>
  <torusGeometry args={ [ 1, 0.5, 16, 32 ] } />
  <meshStandardMaterial color="mediumpurple" />
</mesh>
</RigidBody>
```



Ahora, coloquemos la esfera encima cambiando su **position**:

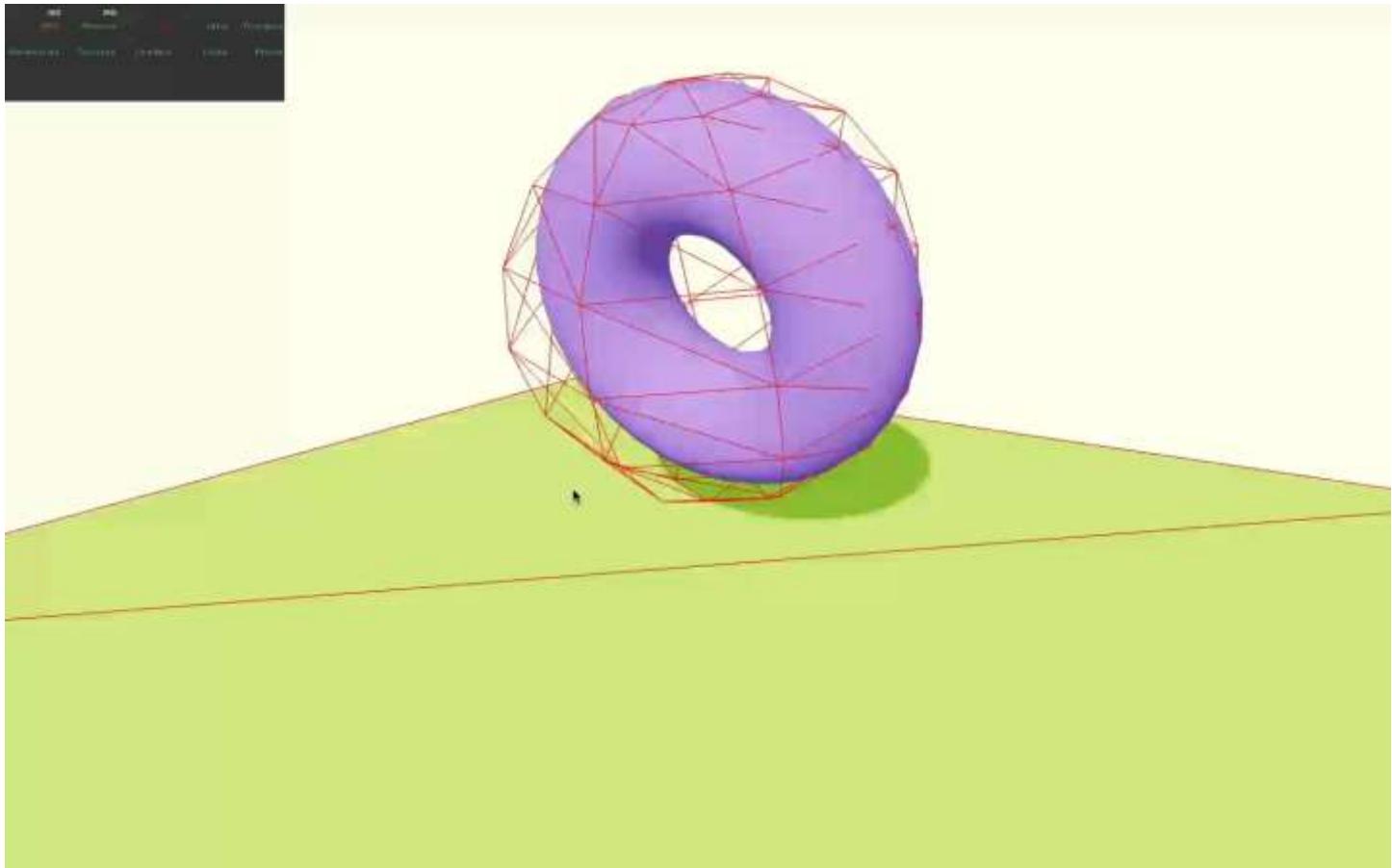
```
<RigidBody colliders="ball">
  <mesh castShadow position={ [ 0, 4, 0 ] }>
    <sphereGeometry />
    <meshStandardMaterial color="orange" />
  </mesh>
</RigidBody>
```



Estamos listos. El problema principal aquí, como puede ver, es que el colisionador de toros es un cuboide y evita que el toro caiga.

Probemos con una ball:

```
<RigidBody colliders="ball">
    <mesh castShadow position={ [ 0, 1, - 0.25 ] } rotation={ [ Math.PI * 0.1, 0, 0 ]
}>
    <torusGeometry args={ [ 1, 0.5, 16, 32 ] } />
    <meshStandardMaterial color="mediumpurple" />
</mesh>
</RigidBody>
```



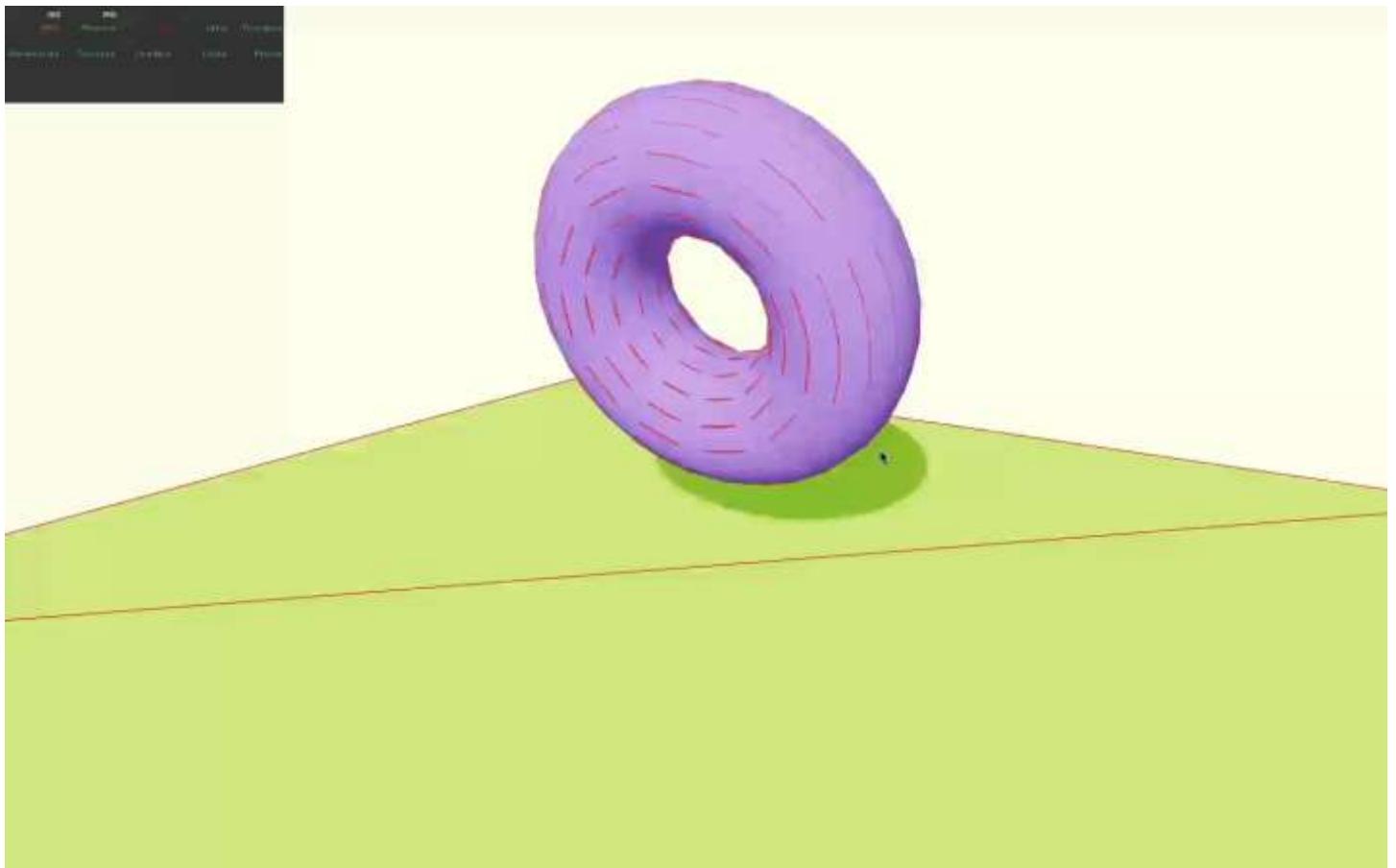
No muy convincente.

La forma del toroide es demasiado específica y necesitamos un colisionador personalizado.

Y ahí es donde el colisionador de casco viene al rescate.

Reemplace el valor del atributo `colliders` por "hull":

```
<RigidBody colliders="hull">
    <mesh castShadow position={ [ 0, 1, - 0.25 ] } rotation={ [ Math.PI * 0.1, 0, 0 ]
}>
    <torusGeometry args={ [ 1, 0.5, 16, 32 ] } />
    <meshStandardMaterial color="mediumpurple" />
</mesh>
</RigidBody>
```



Y ahora, las cosas parecen caer de una manera más precisa físicamente.

En realidad, el casco debería llamarse “casco convexo”. Imagina que colocas una membrana elástica alrededor del objeto. Se ajusta muy bien a la forma, pero se ignoran los agujeros en esa forma.

Y es por eso que la bola no cae dentro del agujero del toroide.

Por cierto, no te dejes engañar por la forma de depuración que puedes ver. Es la geometría base utilizada para crear el casco y no la geometría del casco real.

Apenas lleva tiempo crear cascós y funcionan como cualquier otra forma. También tenga en cuenta que tienen menos rendimiento que un cuboide o una bola.

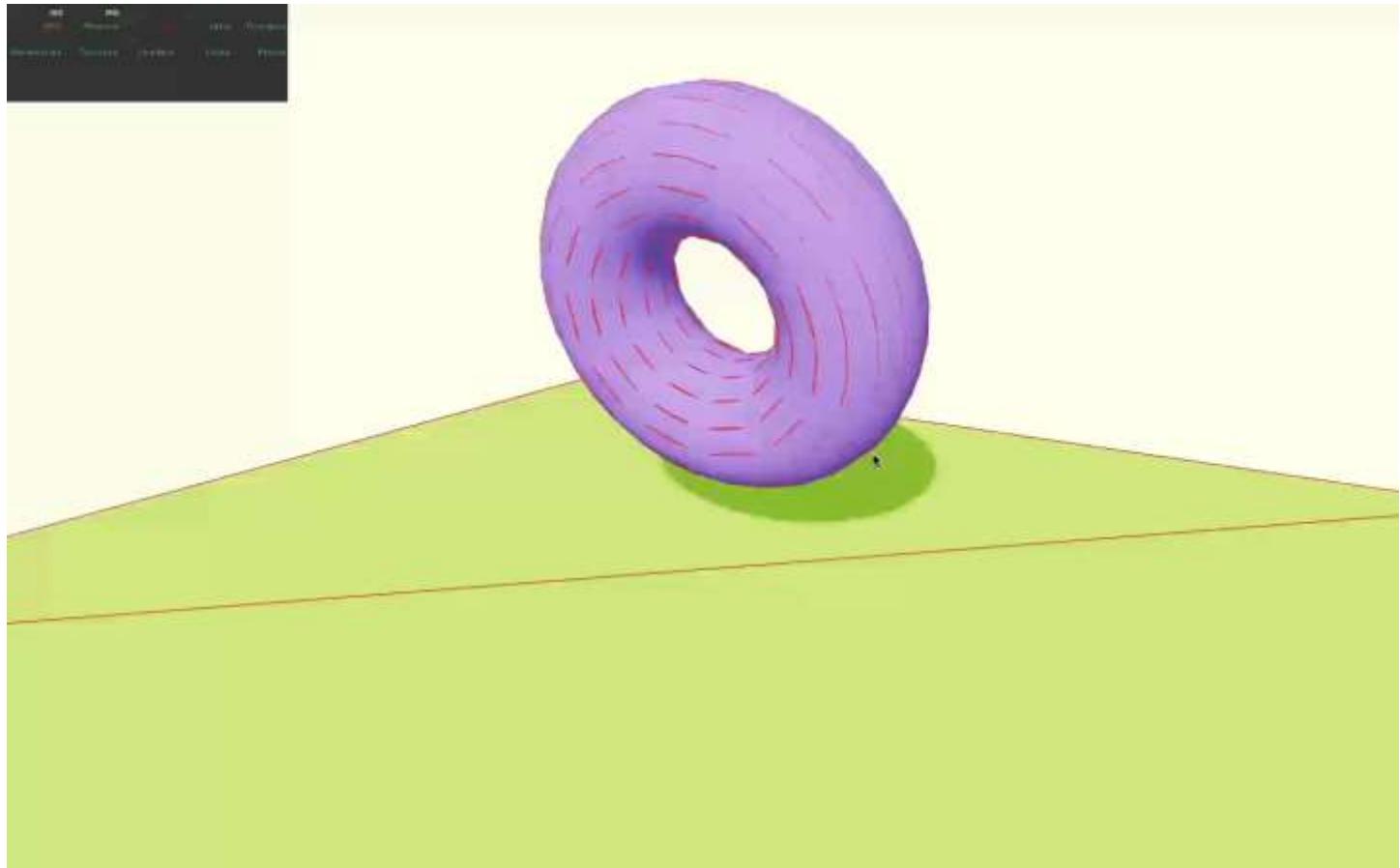
COLISIONADOR TRIMESH

Pero ¿y si realmente quisiéramos que nuestra bola cayera en nuestro agujero toroidal? Como si fuera la canasta de un poste de baloncesto.

En ese caso, una buena solución sería usar una malla triangular (o `trimesh` en React Three Rapier).

Cambie el atributo de `colliders` a “`trimesh`”:

```
<RigidBody colliders="trimesh">
    <mesh castShadow position={ [ 0, 1, -0.25 ] } rotation={ [ Math.PI * 0.1, 0, 0 ] }
}>
    <torusGeometry args={ [ 1, 0.5, 16, 32 ] } />
    <meshStandardMaterial color="mediumpurple" />
</mesh>
</RigidBody>
```



Y ahora la bola cae justo dentro del hoyo.

Entonces, ¿por qué no seguimos adelante y usamos trimesh?

Bueno, hay un problema. Debes evitar el uso de trimesh con **RigidBodies** dinámicos (los objetos dinámicos son los que caen como la esfera y el toroide).

La razón, como se explica en la documentación de Rapier, es que los colisionadores generados con un trimesh están vacíos por dentro y hace que la detección de colisiones sea más complicada y propensa a errores. Un objeto rápido podría atravesar el trimesh o acabar atascado en su superficie.

Esto no significa que no puedas usarlo, pero, preferiblemente, deberías usarlo en **RigidBodies** fijos. De lo contrario, deberías esperar algunos errores.

COLISIONADORES PERSONALIZADOS

Hemos visto que React Three Rapier maneja muy bien los colisionadores automáticos, pero a veces queremos crear nuestro propio colisionador o incluso una composición de colisionadores personalizados.

Para aprender, vamos a crear colisionadores en el toroide que no coinciden con la forma de la geometría.

Primero, debemos decirle a React Three Rapier que no genere el colisionador automático configurando el atributo `colliders` en `false` en el toro `<RigidBody>`:

```
<RigidBody colliders={ false }>
  <mesh castShadow position={ [ 0, 1, - 0.25 ] } rotation={ [ Math.PI * 0.1, 0, 0 ] }
}>
  <torusGeometry args={ [ 1, 0.5, 16, 32 ] } />
  <meshStandardMaterial color="mediumpurple" />
</mesh>
</RigidBody>
```

La pelota debe atravesar el toroide.

A continuación, enumeraremos los colisionadores disponibles. Dado que React Three Rapier implementa los colisionadores de Rapier, podemos usar la documentación de Rapier como referencia para los distintos parámetros:

- `BallCollider` <https://rapier.rs/javascript3d/classes/Ball.html>
- `CuboidCollider` <https://rapier.rs/javascript3d/classes/Cuboid.html>
- `RoundCuboidCollider` <https://rapier.rs/javascript3d/classes/RoundCuboid.html>
- `CapsuleCollider` <https://rapier.rs/javascript3d/classes/Capsule.html>
- `ConeCollider` <https://rapier.rs/javascript3d/classes/Cone.html>
- `CylinderCollider` <https://rapier.rs/javascript3d/classes/Cylinder.html>
- `ConvexHullCollider` <https://rapier.rs/javascript3d/classes/ConvexPolyhedron.html>
- `TrimeshCollider` <https://rapier.rs/javascript3d/classes/TriMesh.html>
- `HeightfieldCollider` <https://rapier.rs/javascript3d/classes/Heightfield.html>

COLISIONADOR CUBOIDE

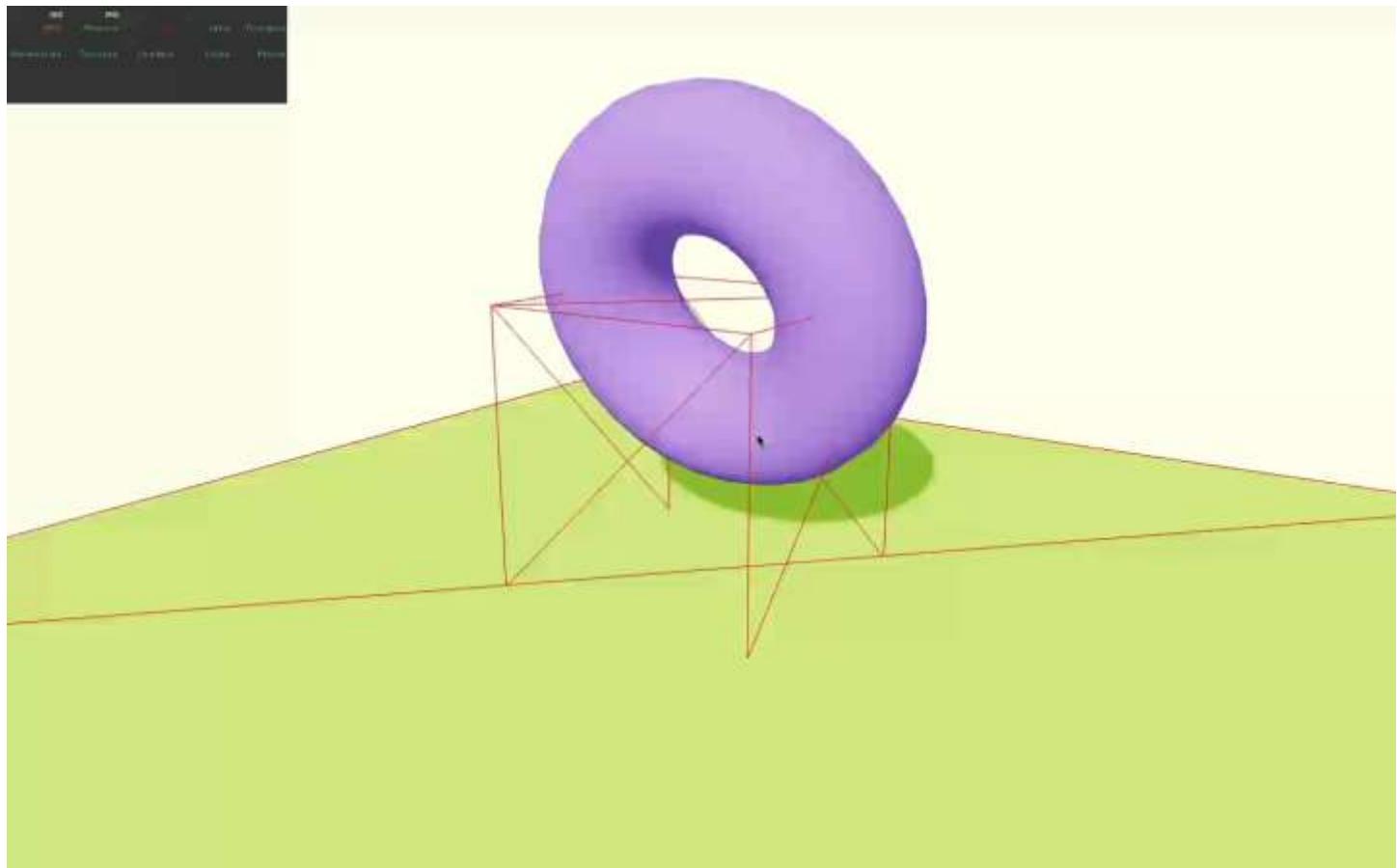
Probemos uno que ya conocemos.

Primero, importe `CuboidCollider` desde `@react-tres/rapier`:

```
import { CuboidCollider, RigidBody, Physics } from '@react-three/rapier'
```

Luego, agréguelo al `<RigidBody>` del toro con los siguientes parámetros (podemos agregarlo en cualquier parte del `<RigidBody>`):

```
<RigidBody colliders={ false }>
  <CuboidCollider args={ [ 1, 1, 1 ] } />
  /* ... */
</RigidBody>
```



Obtenemos el cuboide, pero está lejos de ser perfecto.

En primer lugar, configuramos los argumentos `arg` en `[1, 1, 1]`. Esos son los valores de lo que llamamos media extensión y podemos encontrar esos parámetros en la sección del constructor de la documentación de la clase `Cuboid`:

<https://rapier.rs/javascript3d/classes/Cuboid.html#constructor>

La mitad de la extensión es la mitad del ancho (en el eje x), la mitad de la altura (en el eje y) y la mitad de la profundidad (en el eje z).

Puedes intentar cambiarlos un poco para que coincidan aproximadamente con la forma del toroide:

COLISIONADOR MULTIPLE
BALON COLISIONADOR
OTROS COLISIONADORES
COLISIONADOR CUBOIDE REDONDEADO
COLISIONADOR CILINDRICO
COLISIONADOR CAPSULA
COLISIONADOR CONO

COLISIONAOR CASCARA CONVEXO
COLISIONADOR TRIMESH
COLISIONADOR CAMPO ALTURA

ACCEDE AL CUERPO Y APLICA FUERZAS
REFERENCIAS E IMPULSO

CONFIGURACIONES DE OBJETO

GRAVEDAD
RESTITUCION
FRICCION
MASA
 CONFIGURANDO LA MASA
 ACCEDIENDO A LA MASA
POSICION Y ROTACION
TIPO CINEMATICO

EVENTOS

ENTRAR DE COLISION
SALIR DE COLISION
ENCENDER REPOSO Y ENCENDER LEVANTARSE

DESDE UN MODELO

CARGAR EL MODELO
AGREGAR LAS FISICAS
USAR UN COLISIONADOR PERSONALIZADO
USAR UNA CASCARA
USAR EL TRIMESH

PRUEBAS DE ESTRÉS

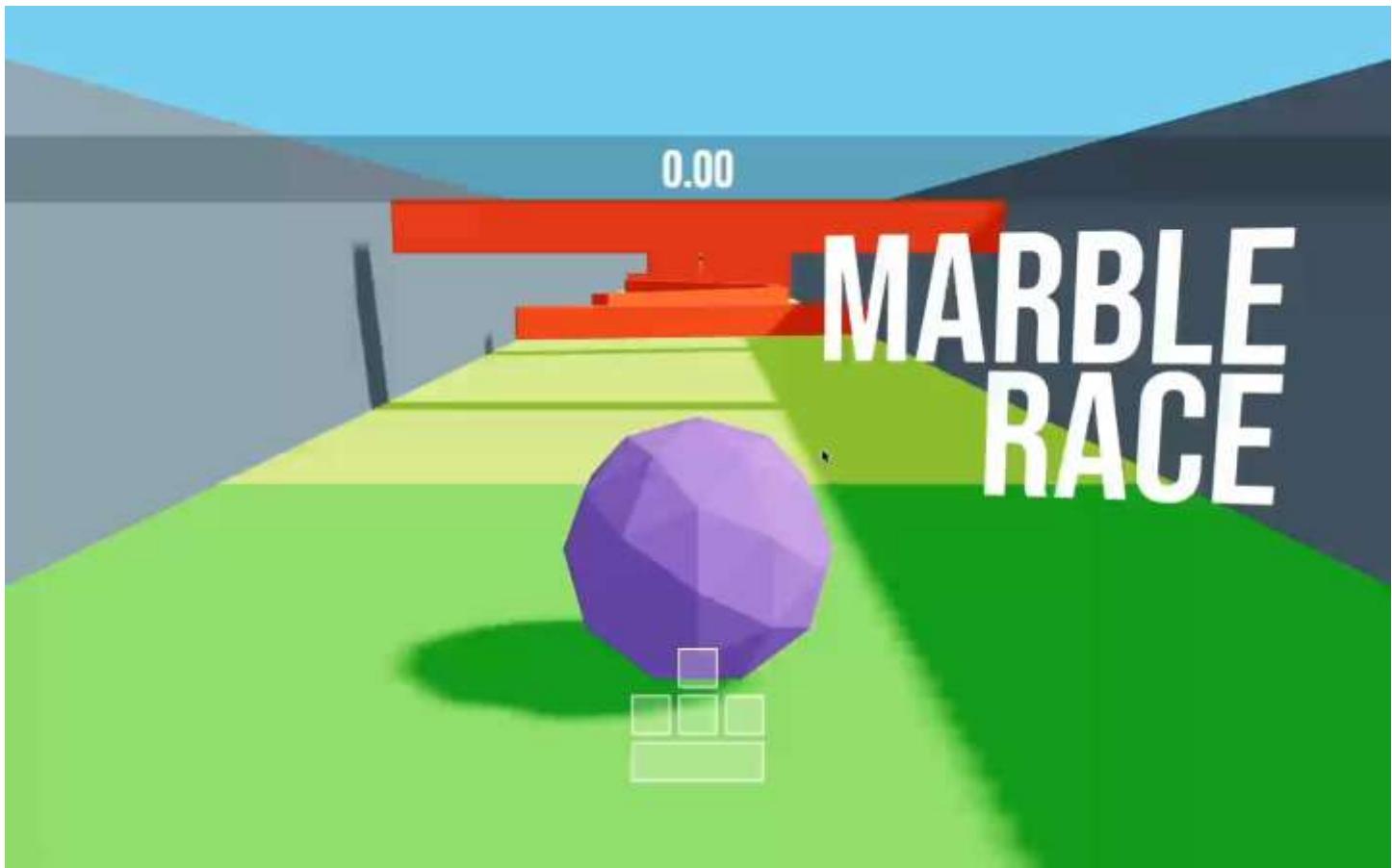
AGREGANDO PAREDES
MALLA INSTANCIADA
REGRESO A LAS FISICAS

YENDO MAS LEJOS

54. Crear un juego

INTRODUCCION

En esta lección amos a aprender como crear un minijuego.



Jugaremos con una canica y vamos a tener que ir atravez de un nivel lleno con varios obstáculos móviles para poder llegar a un destino.

Aquí algunas de las mecánicas:

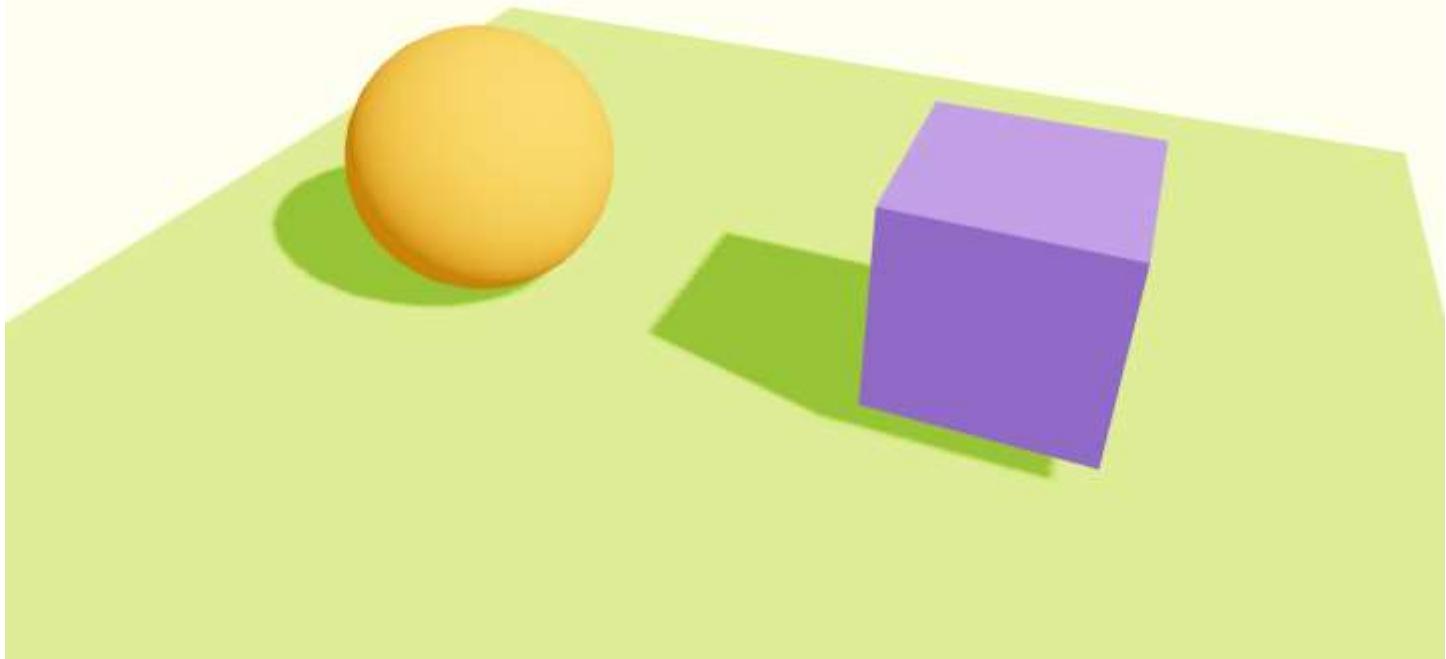
- El jugador puede mover la canica y saltar con el teclado.
- Tan pronto como la canica este en movimiento, un contador iniciara el conteo e indicara al jugador que tanto le tomo finalizar el mapa de obstáculos.
- Al final de la carrera, un botón de reinicio aparecerá y al dar click en el, va a reiniciar la canica a su posición inicial, reiniciara el contador y creara un nuevo set de obstáculos así que el nivel no será el mismo.

Esta lección es una buena oportunidad para poner en practica el conocimiento que obtuvimos en física, interfaces y componentes, pero también nuevos conceptos como un estado global y controles de teclado.

Y obviamente, vamos a crear y jugar un juego.

SETUP

En el inicio, vamos a tener la clásica esfera naranja, el cubo púrpura y el piso verde.



Ambos, la fuente de luz direccional y la fuente de luz ambiente están en el componente <Lights> porque vamos a crear algunos cambios relacionados con las luces un poco después y queremos mantener las cosas organizadas.

Las sombras ya están habilitadas y la luz direccional esta configurada para proyectar las sombras en un área bastante grande.

```
<directionalLight
  castShadow
  position={ [ 4, 4, 1 ] }
  intensity={ 1.5 }
  shadow-mapSize={ [ 1024, 1024 ] }
  shadow-camera-near={ 1 }
  shadow-camera-far={ 10 }
  shadow-camera-top={ 10 }
  shadow-camera-right={ 10 }
  shadow-camera-bottom={ - 10 }
  shadow-camera-left={ - 10 }>
</>
```

La dependencia `@react-three/drei` ya está instalada en el proyecto.

Estamos usando el ayudante `OrbitControls` para ser capaz de mover la cámara alrededor, pero vamos a removerlo después así que tendremos la cámara siguiendo a la canica.

No hemos agregado `<Perf />` desde `r3e-perf` pero podrías definitivamente usarlo si lo creas para tu propio juego. Teniendo un buen frame rate es muy importante y monitorear el performance te ayudara.

Ningún debug UI ha sido agregado porque todos los valores varios y los colores ya han sido cuidadosamente elegidos, pero tu deberías definitivamente agregar uno (como Leva) si tu vas a crear tu propio juego o si quieres mejorar este juego una vez que hayas terminado esta lección.

NIVEL

Cuando creamos un juego, es bueno tener las cosas en la pantalla lo mas rápido posible. Incluso si las mecánicas aun no funcionan, ayuda a tener una idea de que es lo que vendrá y es mucho mas interesante.

Es por lo que vamos a iniciar creando el nivel y varias de sus trampas.

Nuestro nivel se compondrá de lo que llamaremos bloques.

El primer bloque se compondrá de un piso simple sin nada mas. Es donde el jugador iniciara.

El ultimo bloque será la línea de meta.

En el medio, habran un grupo de bloques trampa con obstáculos en movimiento.

Vamos a crear 3 tipos diferentes de trampas y entonces los bloques en medio del inicio y el final con un conjunto aleatorio compuesto de bloques trampa.

COMPONENTE

En

AGREGAR FISICAS

55. El final

Outroduction

Felicidades, lo hiciste. Finalmente llegaste al final de este viaje. Ahora, como podrías imaginar, no es el final de todos los finales. Es tiempo para que continues por tu cuenta. Ahora tienes un gran entendimiento de Three.js y suficiente experiencia para aventurarte por tu cuenta.

Podrás tropezar con obstáculos, pero serán grandiosos para ti, y te sentirás orgulloso una vez que los superes. Cada impedimento es una manera de volverte mejor, y podrás enfrentarlos como retos interesantes.

Hay mucho que aprender aun. He usado Three.js por años, y aun sigo descubriendo nuevas técnicas. Mantente humilde, observa los últimos proyectos, intenta aprender de ellos, y siempre manten tus ojos en los premios mas altos. Siempre puedes ir mas lejos y mejorar. No te quedes en tu zona de confort; no progresaras.

Si tu creas cosas que quieres mostrar, muestralas en Twitter o en cualquier lugar que te guste. Incluso si piensas que es pequeño o inadecuado, no olvides que hubo un momento en el que no sabias absolutamente nada sobre Three.js. Las personas empezaran a notarte, darte consejo, y clientes o reclutadores van a contactarte pronto (si esto es lo que buscas).

Hablando de twitter, no dudes en compartir tus proyectos con el hastag #threejsJourney o directamente mencioname @bruno_simon

Si tienes alguna pregunta, únete al servidor de discord publico o el solo-miembros, la comunidad o yo vamos a hacer lo mejor para ayudarte.

Espero que este curso te gustara y que estes feliz con lo que has aprendido.

Gracias.