



Hola Mundo Three.js

Francisco Velasco Anguita

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Granada

Sistemas Gráficos

Grado en Ingeniería Informática
Curso 2023-2024

Contenidos

1 Ejemplo de aplicación Three.js

2 La aplicación

3 Diseño

- Modelo jerárquico
- Grafo de escena
- Diagrama de clases

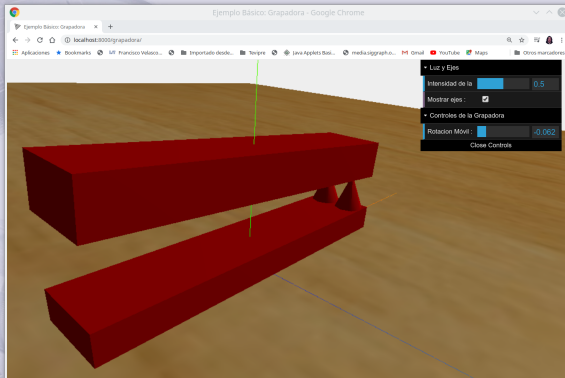
4 Interfaz gráfica de usuario

5 Implementación

- Estructura de la aplicación
- La clase `MyScene`
- La clase `Grapadora`

6 Consejos sobre rendimiento

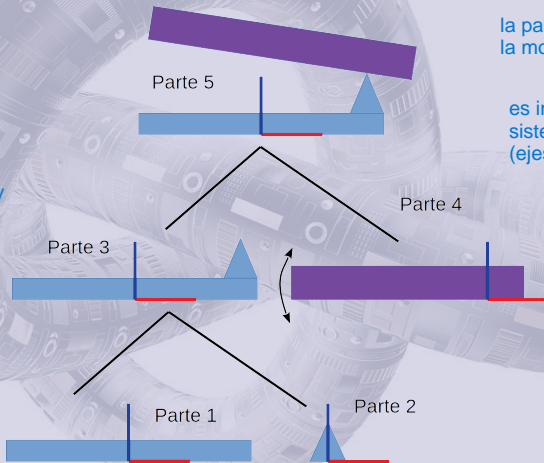
La grapadora



Diseñando el modelo jerárquico

- Descomposición (proceso descendente)

modelo jerárquico
primero hacemos
una etapa divide y
vencerás
descendente:
descomponemos el
modelo que
queremos realizar, y
lo vamos
descomponiendo



la parte morada es móvil,
la morada es la fija

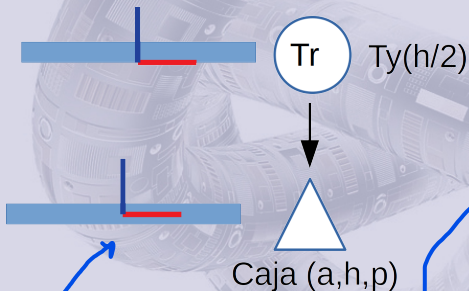
es importante poner los
sistemas de referencia
(ejes)

Diseñando el modelo jerárquico

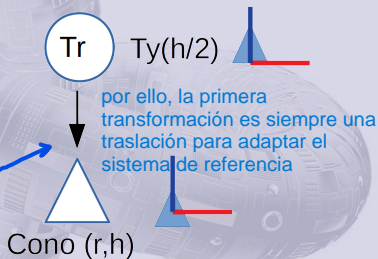
- Composición (proceso ascendente)

ahora creamos el árbol de abajo hacia arriba

► Parte 1



► Parte 2

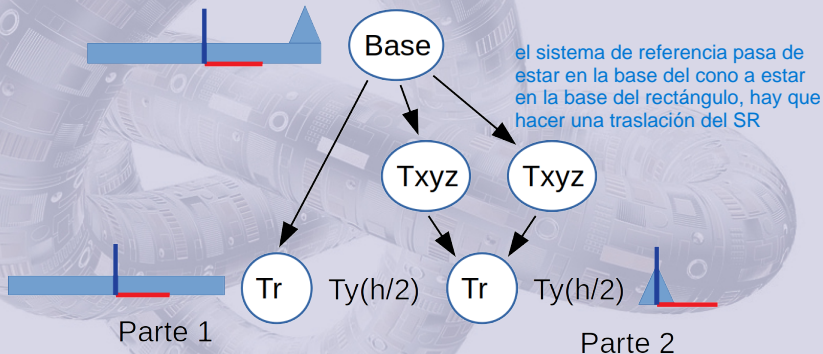


siempre se pone el sistema de referencia en el centro de la figura

Diseñando el modelo jerárquico

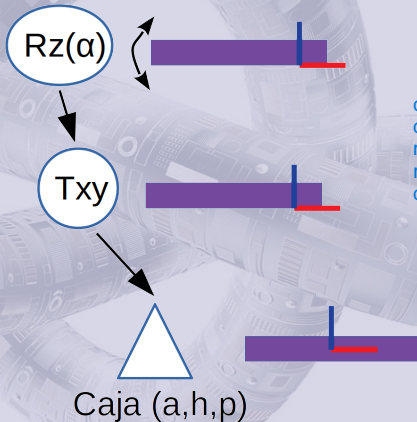
- Composición (proceso ascendente)

- Parte 3



Diseñando el modelo jerárquico

- Composición (proceso ascendente)
 - Parte 4

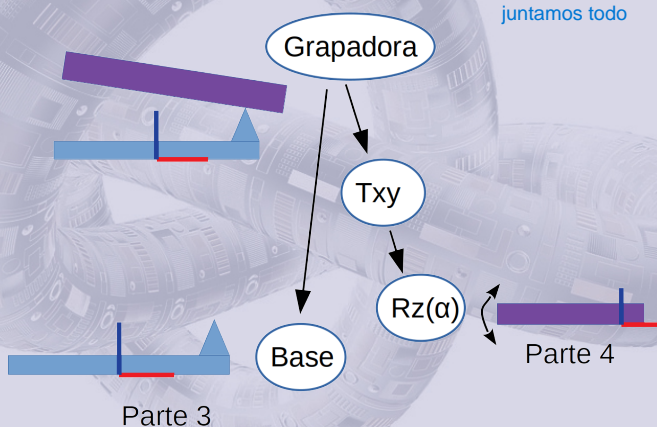


como se hace una traslación desde un punto en concreto, movemos el sistema de referencia hasta el punto donde queremos que se rote

Diseñando el modelo jerárquico

- Composición (proceso ascendente)

- Parte 5



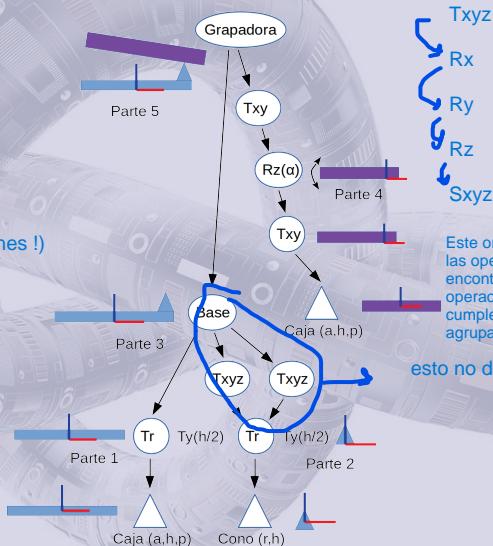
Modelo jerárquico

Diseño general

diseño teórico

```
position.x = 7
position.set(1,2,3);
rotation.x = pi/2 (en radianes !)
```

scale.x °



Orden de las operaciones:

Txyz
Rx
Ry
Rz
Sxyz

Este orden nos sirve para agrupar las operaciones, cuando nos encontremos más de una operación consecutiva, si cumplen esta jerarquía podemos agruparla

esto no debería de hacerse

Modelo jerárquico

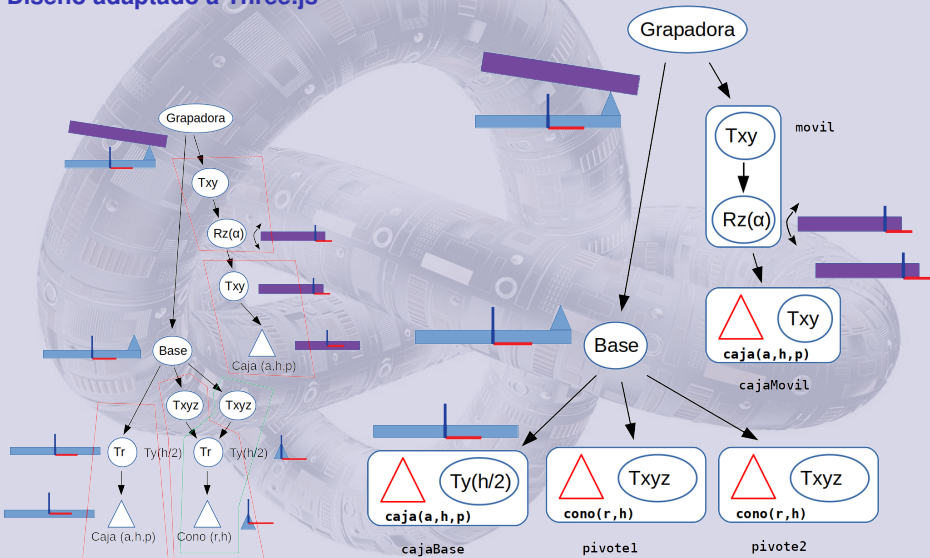
importante

Diseño adaptado a Three.js

- Un nodo interno puede realizar varias transformaciones, pero en un determinado orden
 - 1 Los escalados, el orden de los distintos ejes no es importante
 - 2 Las rotaciones, primero sobre Z, luego sobre Y, por último sobre X
 - 3 Las traslaciones, el orden de los distintos ejes no es importante
 - Un nodo con geometría, también puede realizar transformaciones (con las mismas restricciones)
 - Un nodo no puede tener más de un padre
- Un modelo jerárquico adaptado a Three
- ▶ Es un árbol, no un grafo
 - ▶ Puede tener menos nodos si se agrupan varias transformaciones en un único nodo (importante, cumpliendo los requisitos)

Modelo jerárquico

Diseño adaptado a Three.js



Grafo de escena

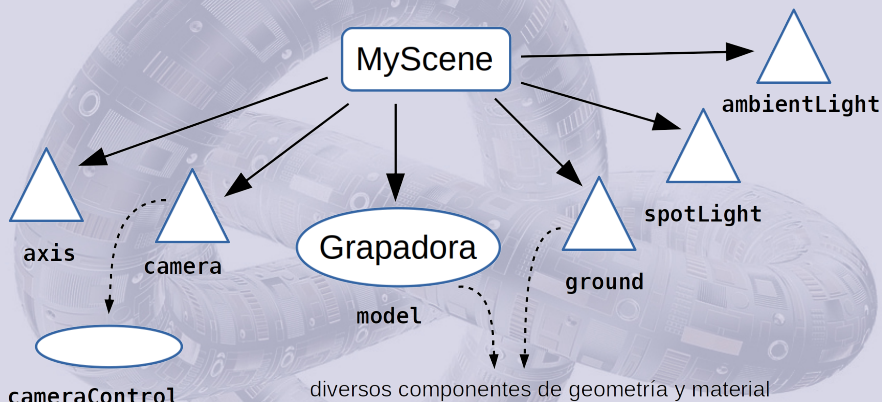
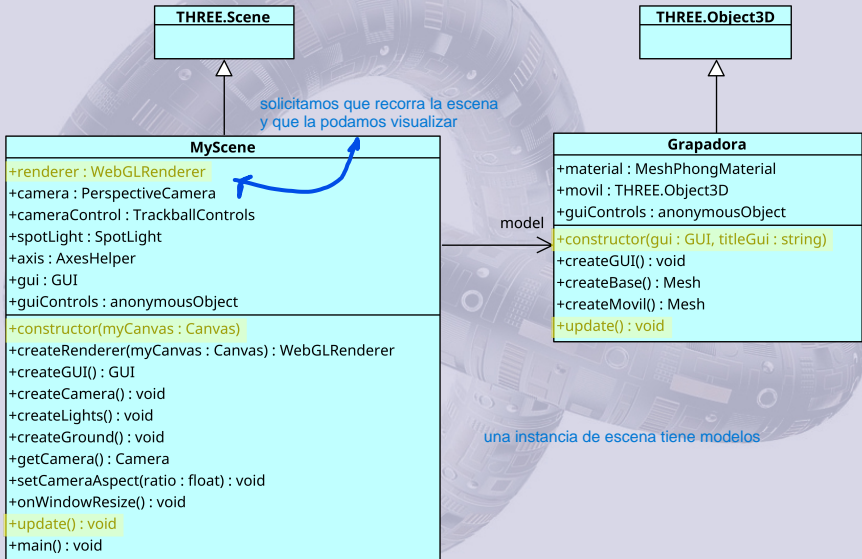


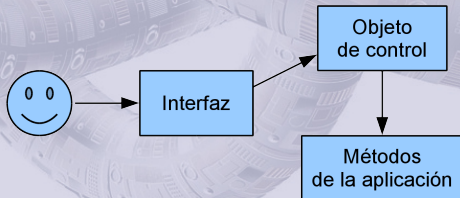
Diagrama de clases



Interfaz Gráfica de Usuario

dat.gui

- Se puede descargar de:
<https://github.com/dataarts/dat.gui>
- La ayuda se encuentra en:
<https://github.com/dataarts/dat.gui/blob/master/API.md>
- Ejemplos de su uso en:
<http://workshop.chromeexperiments.com/examples/gui>
- Estructura básica



dat.gui

Ejemplo

GUI: Objeto de control

```
GUIcontrols = {  
  axis : true,  
  power : 0.5  
  addBox : () => { . . . }  
}
```

▼ Axis and Lights

Axis on/off : ☒Light intensity : 0.5

▼ Actions

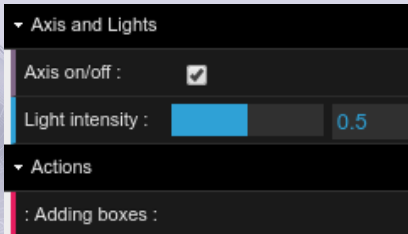
: Adding boxes :

- El valor asignado a cada atributo determina el tipo de control
 - ▶ Booleano, muestra un checkbox
 - ▶ Numérico, muestra un deslizador
 - ▶ Función, muestra un “botón”
 - ★ Al pulsarlo se ejecuta el código de la función

dat.gui

Composición de la interfaz

- Se crea el objeto único de la interfaz
- Se le pueden añadir *carpetas* y controles



GUI: Composición de la interfaz

```
var gui = new GUI();

var axisLights = gui.addFolder ('Axis and Lights');
// obj. control atributo Texto en pantalla
axisLights.add(GUIcontrols, 'axis').name('Axis on/off :');
// En los deslizadores, min, máx, incremento
axisLights.add(GUIcontrols, 'power', 0, 1.0, 0.1)
.name('Light intensity :');

var actions = gui.addFolder ('Actions');
var addingBoxes = actions.add(GUIcontrols, 'addBox')
.name (': Adding boxes :');
```

hay que añadir caracteres que indiquen que es un botón

dat.gui

Actualización de la escena

- Cuando es necesario se leen los valores del objeto de control
- Con ellos se modifican los objetos de la escena
- Si se realiza en los métodos update (para cada frame), los objetos siempre están actualizados según la interfaz

▼ Axis and Lights

Axis on/off : ☒

Light intensity : 0.5

▼ Actions

: Adding boxes :

GUI: Lectura de valores desde la aplicación

```
// Desde algún método update  
this.spotLight.power = GUIcontrols.power;
```

dat.gui

Actuación sobre la escena

- Desde la interfaz se puede modificar directamente la escena
- Un método se ejecuta cada vez que se cambia un valor

GUI: Modificación de la escena desde la interfaz

```
// Se tiene un método que modifica el modelo
setIntensidad (valor) {
  this.spotLight.power = valor;
}

// Se crea y configura el componente de la interfaz
var axisLights = gui.addFolder ( 'Axis and Lights' );
axisLights.add(GUIcontrols, 'lightIntensity', 0, 1.0, 0.1)
  .name('Light intensity :')
  .onChange ( (value) => this.setIntensidad(value) );
```

añadimos el método onchange para que cada vez que se actualice llamemos al método para que modifique el valor

dat.gui

Actualizar la interfaz desde el código

- Si desde el código se modifica el objeto de control, la interfaz no mostrará los nuevos valores al usuario
- Salvo los componentes de la interfaz que estén en modo *escucha*

GUI: Modo escucha

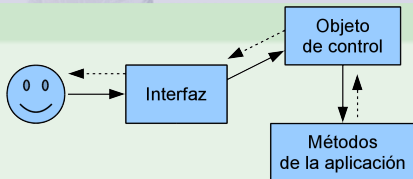
```
// Creación del objeto de control
```

```
control = {
  valor1 : 3,
  valor2 : 5
}
```

```
// Composición de la interfaz
```

```
interfaz = new GUI();
interfaz.add (control, 'valor1', 1, 10, 1).listen();
interfaz.add (control, 'valor2', 1, 10, 1);
```

```
// En algún punto de la aplicación se modifica el objeto de control
control.valor1 = 7; // El usuario verá este valor en la interfaz
control.valor2 = 7; // El componente de este valor NO se actualiza
```



cuando ponemos `listen`, si la aplicación modifica valores de los objetos de control, éstos se modificarán para el usuario

Implementación: Estructura de la aplicación

- La aplicación es un `html` que referencia a otros archivos

Aplicación: Archivo `index.html`

Sirve para todos los ejercicios, simplemente cambiamos el nombre de la página y de los archivos

```
<!DOCTYPE html>
<html>
<head>
  <title>Ejemplo Básico : Grapadora</title>
  <meta charset="utf-8">
  <script type="text/javascript" src="../libs/jquery.js"></script>
  <script type="importmap">
    { "imports": { "three": "../libs/three.module.js" } }
  </script>
  <script type="module" src="MyScene.js"></script>
  <style>
    body { margin: 0; overflow: hidden; }
  </style>
</head>
<body>
<!-- Div que muestra la imagen, el lienzo -->
<div id="WebGL-output">
</div>
</body>
</html>
```

En el canvas se ponen los gráficos

La clase `MyScene`

- Su responsabilidad principal es:
 - ▶ Crear un `renderer`
 - ▶ Crear el grafo de escena
 - ▶ Actualizar y visualizar el grafo (método `update()`)
 - ★ La visualización se realiza solicitándosela al `renderer`
 - ★ La actualización del grafo que es accesible directamente desde esta clase se realiza desde el propio método `update()`
 - ★ La actualización del grafo no accesible directamente, se le solicita a otros objetos
 - ★ **Importante:**
El método `update()` de la escena se encarga también de que vuelva a ser llamado cada vez que haya que “refrescar” la pantalla
- Veamos las partes del código más significativas

imports y exports

- En cada archivo se deben importar las clases que se van a necesitar
- Y exportar las clases que puedan necesitar otros

MyScene: imports

// Clases de la biblioteca

```
import * as THREE from '../libs/three.module.js'
import { GUI } from '../libs/dat.gui.module.js'
import { TrackballControls } from '../libs/TrackballControls.js'
```

cargamos todo

// Clases de mi proyecto

```
import { Grapadora } from './Grapadora.js'
```

cada clase dentro de un archivo

// En este caso no se exporta nada

El main

MyScene: El main

```
/// La función main
$(function () {

    // Se instancia la escena pasándole el div que se ha creado en el html para visualizar
    // Al instanciar la escena, se construye el renderer y el grafo
    var scene = new MyScene("#WebGL-output");

    // Se añaden los listener de la aplicación. En este caso, el que va a comprobar cuándo se
    // modifica el tamaño de la ventana de la aplicación.
    window.addEventListener("resize", () => scene.onWindowResize());

    // Que no se nos olvide, la primera visualización.
    scene.update();
});
```

Los listener controlan eventos. Aquí hemos puesto que se renderice todo si cambia el tamaño de la pantalla

La clase MyScene

MyScene: Clase y constructor

```
class MyScene extends THREE.Scene {  
  // Recibe el div que se ha creado en el html que va a ser el lienzo en el que mostrar  
  // la visualización de la escena  
  constructor (myCanvas) {  
    super();  
    // Se crea el visualizador, pasándole el lienzo sobre el que realizar los renderizados  
    this.renderer = this.createRenderer(myCanvas);  
    // Se crea la interfaz gráfica de usuario  
    this.gui = this.createGUI ();  
  
    // Construimos los distintos elementos que tendremos en la escena  
    // No basta con construirlos, deben añadirse al grafo con el método add  
    // this aquí es el nodo raíz del grafo  
  
    // Se crean unas luces. El propio método las añade al grafo  
    this.createLights ();  
    // Tendremos una cámara con un control de movimiento con el ratón  
    this.createCamera ();  
    // Un suelo  
    this.createGround ();  
    // Y unos ejes. Imprescindibles para orientarnos sobre dónde están las cosas  
    this.axis = new THREE.AxesHelper (5);  
    this.add (this.axis);  
    // Por último creamos el modelo.  
    this.model = new Grapadora(this.gui, "Controles de la Grapadora");  
    this.add (this.model);  
  }  
  . . .  
}
```


La clase MyScene

MyScene: Construyendo el renderer

```
createRenderer (myCanvas) {  
  // Se recibe el lienzo sobre el que se van a hacer los renderizados.  
  // Un div definido en el html.  
  
  // Se instancia un Renderer WebGL  
  var renderer = new THREE.WebGLRenderer();  
  
  // Se establece un color de fondo en las imágenes que genera el render  
  renderer.setClearColor(new THREE.Color(0xEEEEEE), 1.0);  
  
  // Se establece el tamaño, se aprovecha la totalidad de la ventana del navegador  
  renderer.setSize(window.innerWidth, window.innerHeight);  
  
  // La visualización se muestra en el lienzo recibido  
  $(myCanvas).append(renderer.domElement);  
  
  return renderer;  
}
```

La clase MyScene

MyScene: La interfaz de usuario

```
createGUI () {  
    // Se crea la interfaz gráfica de usuario  
    var gui = new GUI();  
  
    // La escena le va a añadir sus propios controles.  
    // Se definen mediante un objeto de control  
    // En este caso la intensidad de la luz y si se muestran o no los ejes  
    this.guiControls = {  
        lightPower : 500,  
        axisOnOff : true  
    }  
  
    // Se crea una sección para los controles de esta clase  
    var folder = gui.addFolder ( 'Luz y Ejes' );  
  
    // Se le añade un control para la intensidad de la luz  
    folder.add (this.guiControls, 'lightPower', 0, 1000, 20)  
        .name('Intensidad de la Luz : ')  
        .onChange ( (value) => this.setLightPower (value) );  
  
    // Y otro para mostrar u ocultar los ejes  
    folder.add (this.guiControls, 'axisOnOff')  
        .name ( 'Mostrar ejes : ' )  
        .onChange ( (value) => this.setAxisVisible (value) );  
  
    return gui;  
}
```

La clase MyScene

MyScene: El método update

```
update () {  
    // Le decimos al renderizador  
    // "visualiza la escena que te indico usando la cámara que te estoy pasando"  
    this.renderer.render (this, this.getCamera());  
  
    // Se actualizan los elementos del grafo para cada frame  
  
    // Los nodos accesibles directamente desde esta clase se actualizan aquí  
  
    // Se actualiza la posición de la cámara según su controlador  
    this.cameraControl.update();  
  
    // Para la actualización del resto de nodos se le pide a los objetos que correspondan  
    this.model.update();  
  
    // Este método debe ser llamado cada vez que queramos visualizar la escena de nuevo  
    // Se consigue con la siguiente línea  
    // Le decimos al navegador: "La próxima vez que haya que refrescar la pantalla,  
    // llama al método que te indico".  
    // Si no existiera esta línea, update() se ejecutaría solo la primera vez  
    requestAnimationFrame(() => this.update())  
}
```

La clase Grapadora

- Se encarga de construir y actualizar “su parte del grafo”
- Añade su parte de interfaz gráfica de usuario a la aplicación

Grapadora: imports y exports

```
import * as THREE from '../libs/three.module.js'

class Grapadora extends THREE.Object3D {
  . . .
}

export { Grapadora }
```

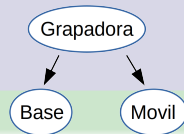
importante



La clase Grapadora

Grapadora: Clase y constructor

```
constructor(gui, titleGui) {  
    super();  
    // Se crea la parte de la interfaz que corresponde a la grapadora  
    // Se crea primero porque otros métodos usan las variables que se definen para la interfaz  
    this.createGUI(gui, titleGui);  
    // El material se usa desde varios métodos. Por eso se almacena en un atributo  
    this.material = new THREE.MeshStandardMaterial({color: 0xCF0000});  
    // A la base no se accede desde ningún método. Se almacena en una variable local del constructor  
    var tamano = 0.15; // 15 cm de largo. Las unidades son metros  
    var base = this.createBase(tamano);  
    // Al nodo que contiene la transformación interactiva que abre y cierra la grapadora se  
    // accede desde el método update, se almacena en un atributo.  
    this.movil = this.createMovil(tamano);  
    // Al nodo this, la grapadora, se le cuelgan como hijos la base y la parte móvil  
    this.add (base);  
    this.add (this.movil);  
}
```



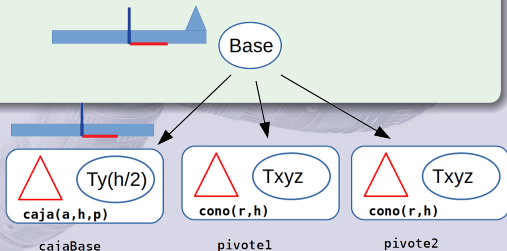
La clase Grapadora

Grapadora: Método createBase

```

createBase(tama) {
  // El nodo del que van a colgar la caja y los 2 conos y que se va a devolver
  var base = new THREE.Object3D();
  // Cada figura, un Mesh, está compuesto de una geometría y un material
  var cajaBase = new THREE.Mesh (new THREE.BoxGeometry (tama,tama*0.08,tama*0.2), this.
    material);
  cajaBase.position.y = tama*0.04;
  // La componente geometría se puede compartir entre varios meshes
  var geometriaPivote = new THREE.ConeGeometry (tama*0.05, tama*0.12);
  var pivote1 = new THREE.Mesh (geometriaPivote, this.material);
  var pivote2 = new THREE.Mesh (geometriaPivote, this.material);
  // Se posicionan los pivotes con respecto a la base
  pivote1.position.set (tama*0.45, tama*0.14, tama*0.05);
  pivote2.position.set (tama*0.45, tama*0.14, -tama*0.05);
  base.add(cajaBase);
  base.add(pivote1);
  base.add(pivote2);
  return base;
}

```



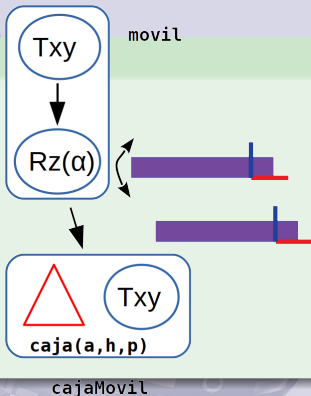
La clase Grapadora

Grapadora: Método createMovil

```
createMovil (tama) {
  // Se crea la parte móvil
  var cajaMovil = new THREE.Mesh (
    new THREE.BoxGeometry (tama, tama*0.12, tama*0.2),
    this.material
  );
  cajaMovil.position.set (-tama*0.45, tama*0.06, 0);

  var movil = new THREE.Object3D();

  [
    movil.rotation.z = this.guiControls.rotacion;
    movil.position.set (tama*0.45, tama*0.2, 0);
    movil.add(cajaMovil);
    return movil;
  ]
}
```



el orden de las líneas da igual, cuando el renderer recorra estas líneas ya ordenará las líneas

La clase Grapadora

Grapadora: Método createGUI

```
createGUI (gui, titleGui) {  
  // Controles para el movimiento de la parte móvil  
  this.guiControls = {  
    rotacion : 0  
  }  
  // Se crea una sección para los controles de la caja  
  var folder = gui.addFolder (titleGui);  
  // Estas líneas son las que añaden los componentes de la interfaz  
  // Las tres cifras indican un valor mínimo, un máximo y el incremento  
  folder.add (this.guiControls, 'rotacion', -0.125, 0.2, 0.001)  
    .name ('Apertura :')  
    .onChange ( (value) => this.setAngulo (-value) );  
}
```

Grapadora: Método update

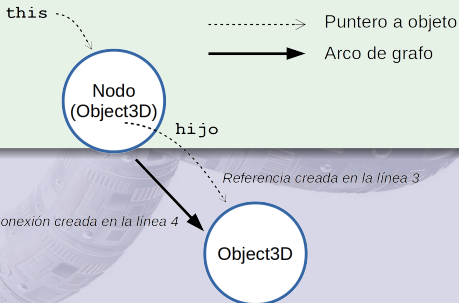
```
update () {  
  // En esta ocasión no hay nada que actualizar  
}
```


Conexión de los nodos a sus respectivos padres

- No olvidarse de conectar los nodos a sus respectivos padres
- Los nodos que no estén conectados al grafo no son tenidos en cuenta en la visualización
- Las referencias se usarán para acceder cómodamente a los nodos que se deseen modificar

: Referencia (puntero) vs. Arco de grafo

```
1 class Nodo extends THREE.Object3D {  
2   constructor() {  
3     this.hijo = new THREE.Object3D ();  
4     this.add (this.hijo);  
5     ...  
6   }  
7   ...  
8 }
```



crear, y luego añadir logo

Consejos sobre rendimiento

Prestad especial atención a los métodos update

- La visualización se realiza varias veces por segundo
- Esto exige ser cuidadosos con el rendimiento
 - ▶ Optimizar los algoritmos empleados
 - ▶ **No construir objetos en los métodos update** sobre todo si se van a descartar después
 - ▶ **Reutilizar objetos ya existentes**
 - ★ Si los objetos necesarios son una cantidad fija, construirlos una sola vez y después solo cambiarles el estado con los setters correspondientes
 - ★ Si los objetos son en un número indeterminado (por ejemplo, proyectiles en un juego de naves) usar un gestor de objetos

Gestor de objetos

- Una clase que gestiona un array (almacen) de objetos
- Cuando se necesita un objeto, se toma del almacen.
Solo se construye un objeto nuevo cuando no hay en el almacen
- Cuando ya no se necesita, se retorna al almacen

Ejemplo: Gestor de objetos

```
class GestorBala {  
  constructor () {  
    this.almacen = [];  
  }  
  
  dameUnaBala () {  
    if (this.almacen.length) { // Hay elementos en el almacen  
      return this.almacen.pop();  
    } else { return new Bala(); } // Solo se construyen objetos cuando es necesario  
  }  
  
  teDevuelvoUnaBala (p) {  
    this.almacen.push (p);  
  }  
}
```

Uso del gestor de balas

Ejemplo: Constructor de la escena y método disparo

```
constructor() {  
    // Entre otras cosas  
    this.gestorBalas = new GestorBalas();  
    this.nodoBalas = new THREE.Object3D();  
    this.balasAdestruir = [];  
}  
  
disparo() {  
    // Se solicita una bala al gestor  
    var unaBala = this.gestorBalas.dameUnaBala();  
  
    // Se configura: posición, dirección, velocidad, potencia, etc.  
    unaBala.set ( . . . );  
  
    // Se añade al nodo de balas  
    this.nodoBalas.add (unaBala);  
}
```

Uso del gestor de balas

Ejemplo: update de la escena

```
update() {  
  // Se actualizan las balas  
  var unaBala;  
  for (var i = 0; i < this.nodoBalas.children.length; i++) {  
    unaBala = this.nodoBalas.children[i];  
    unaBala.update();  
    if (unaBala.debeDesaparecer()) {  
      this.balasAdestruir.push(unaBala);  
    }  
  }  
  
  // Se descartan las balas innecesarias, pero no se destruyen, se reciclan ;-)  
  for (var i = 0; i < this.balasAdestruir.length; i++) {  
    unaBala = balasAdestruir[i];  
    this.nodoBalas.remove(unaBala);  
    this.gestorBalas.teDevuelvoUnaBala(unaBala);  
  }  
  this.balasAdestruir.length = 0;  
}
```

Creación de geometría

NIGGER

- La creación de geometría es un proceso lento
 - ▶ **Las geometrías se crean en la construcción de la figura**
 - ▶ **No** se crean en los métodos que se ejecutan para cada frame
 - ▶ Si en algún caso, es totalmente necesario la creación de una geometría para descartar una anterior
 - ★ La que vamos a descartar debe destruirse
 - ★ Solo se construye y destruye la geometría, el Mesh será el mismo

: Substitución de la geometría

```
elMesh.geometry.dispose();  
elMesh.geometry = new THREE.BoxGeometry ( . . . );
```



Hola Mundo Three.js

Francisco Velasco Anguita

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Granada

Sistemas Gráficos

Grado en Ingeniería Informática
Curso 2023-2024