

# Grafos de Escena

Francisco Velasco Anguita

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

Sistemas Gráficos

Grado en Ingeniería Informática  
Curso 2017-2018

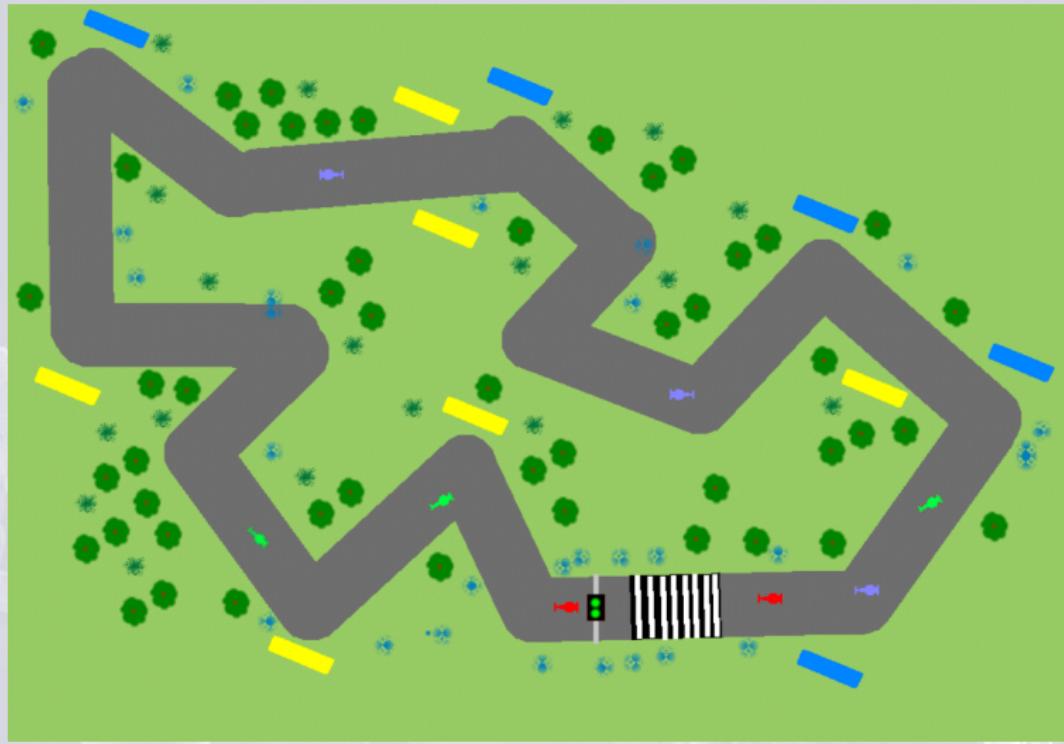
# Contenidos

- 
- 1 Introducción
  - 2 Concepto y estructura de un grafo de escena
  - 3 Modelos jerárquicos
  - 4 Animación
  - 5 Interacción
  - 6 Detección de colisiones
  - 7 Física

# Objetivos

- Comprender lo que es un grafo de escena y su estructura
- Saber organizar un sistema gráfico 3D mediante grafos de escena
- Dotar de movimiento a los elementos de la escena
- Permitir la interacción del usuario con la aplicación
- Detectar colisiones entre objetos
- Programar un motor de física

# Introducción



# Introducción



# Introducción

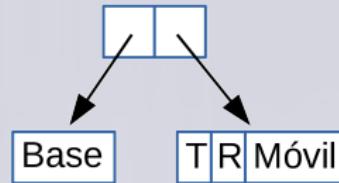


## Ejemplo: Grapadora sin estructura

```
void dibujaGrapadora () {  
    glBegin (GL_TRIANGLES);  
        // vertices para la base  
    glEnd();  
    glTranslatef ( . . . );  
    glRotatef ( . . . );  
    glBegin (GL_TRIANGLES);  
        // vertices para la parte móvil  
    glEnd();  
}
```

# Introducción

- Se organiza mediante un modelo jerárquico

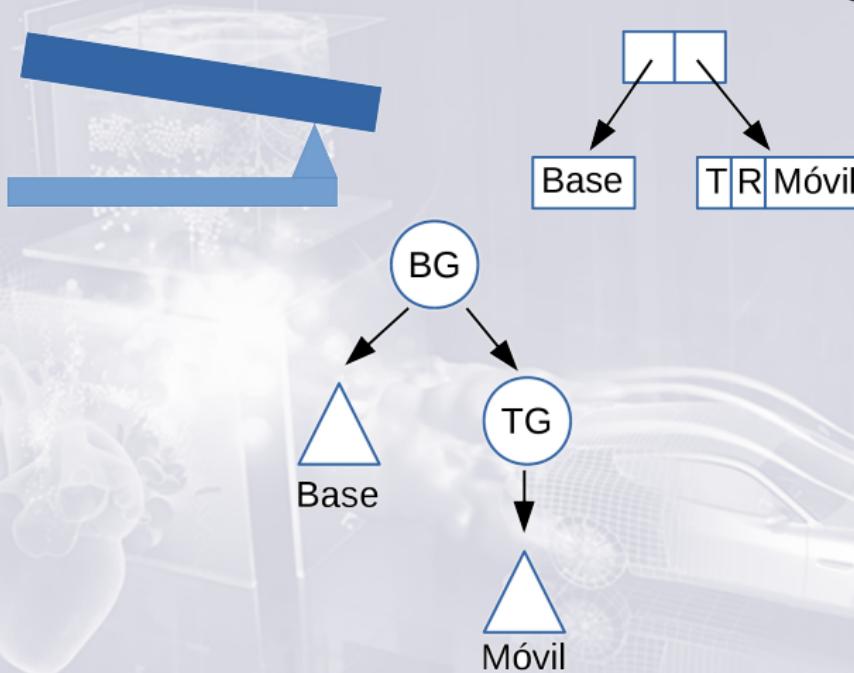


## Ejemplo: Modelo Jerárquico implementado mediante código

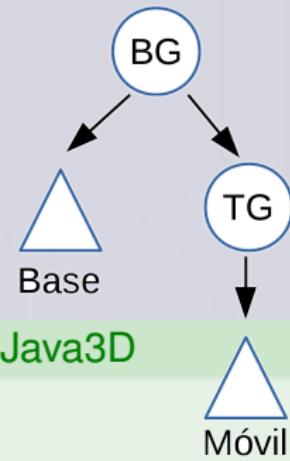
```
void dibujaGrapadora () { dibujaBase ();      dibujaMovil () ; }
void dibujaBase () {
    glBegin (GL_TRIANGLES);
    // vertices para la base
    glEnd ();
}
void dibujaMovil {
    glTranslatef ( . . . );
    glRotatef ( . . . );
    glBegin (GL_TRIANGLES);
    // vertices para la parte móvil
    glEnd ();
}
```

# Introducción

- Se organiza mediante un modelo jerárquico
- Se implementa mediante una estructura de datos: un grafo



# Introducción



## Ejemplo: Implementación del grafo de escena en Java3D

```
class Grapadora extends BranchGroup {  
    public Grapadora () {  
        Base base = new Base ();  
        Movil movil = new Movil ();  
        Transform3D transf = new Transform3D ();  
        // aquí iría el código para definir transf  
        TransformGroup movilTransformado = new TransformGroup (transf);  
        movilTransformado.addChild (movil);  
        this.addChild (base);  
        this.addChild (movilTransformado);  
    }  
}
```

# Introducción

## VRML

```
Material {  
    ambientColor 0.200 0.200 0.200  
    diffuseColor 0.800 0.400 0.500  
    shininess 0.000  
}  
Cube {  
    width 2.000  
    height 2.000  
    depth 2.000  
}
```

## Java 3D Programming

```
universe = createVirtualUniverse();  
Locale locale = createLocale( universe );  
BranchGroup sceneBranchGroup = createSceneBranchGroup();  
Background background = createBackground();
```

## OpenGL Programming

```
glClearColor(1.0, 1.0, 1.0, 1.0);  
glClear(GL_COLOR_BUFFER_BIT);  
	glColor3f(0.0, 0.0, 0.0);  
	glMatrixMode(GL_PROJECTION);  
	glLoadIdentity();
```

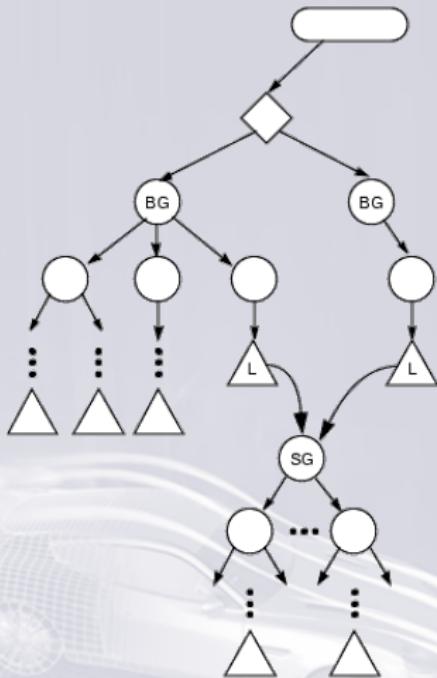
## Vendor Specific Programming

# Concepto de grafo de escena

- Grafo de escena

- ▶ Grafo dirigido acíclico
- ▶ Representa los elementos de una escena
- ▶ Y las relaciones entre ellos

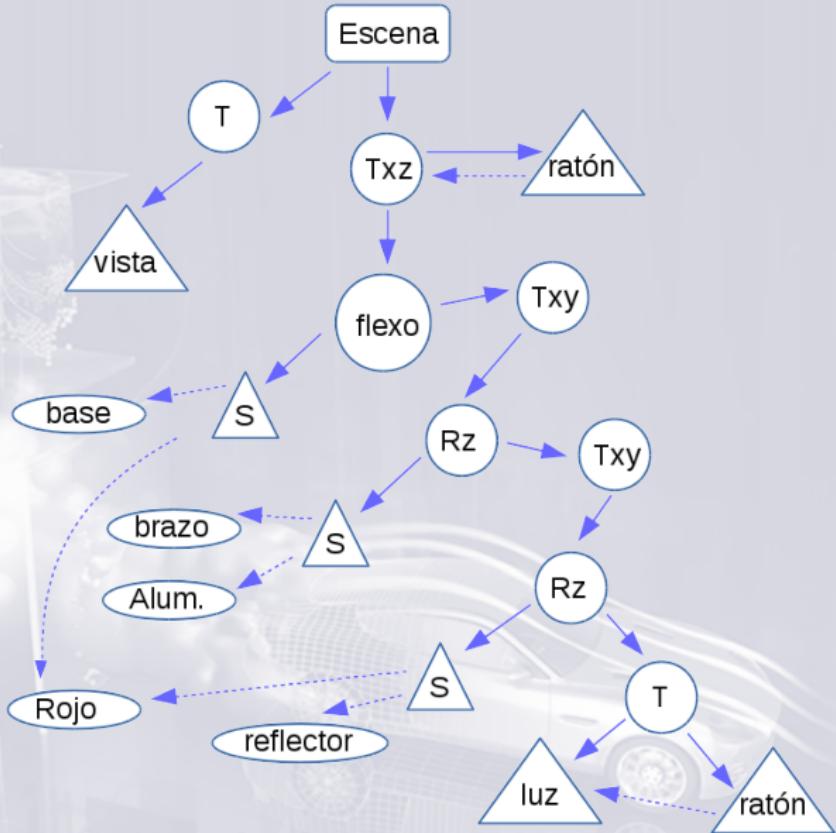
- Usa otras API para visualizar



# Tipos de nodos

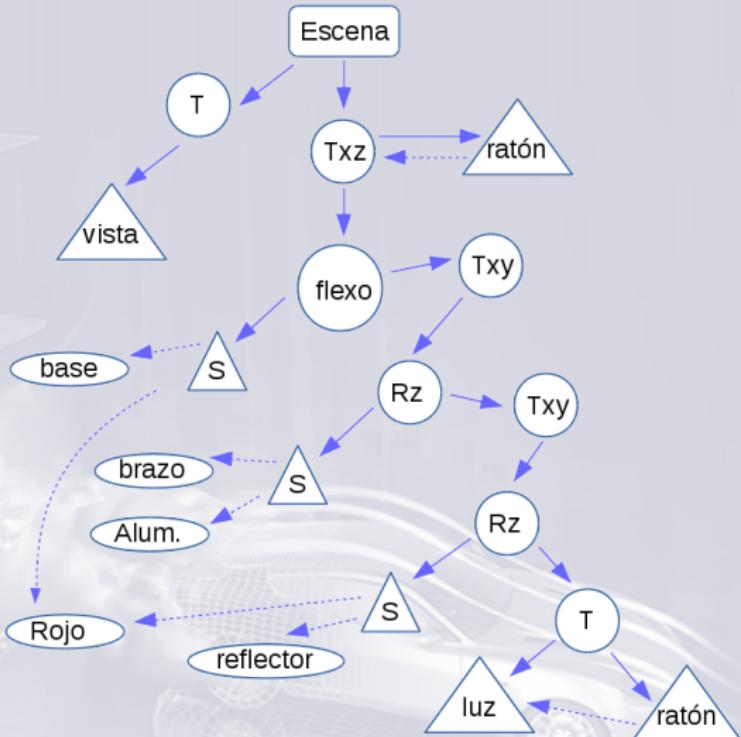
- Nodos de geometría
  - ▶ Objetos geométricos
  - ▶ El fondo de la escena, etc.
- Nodos que *influyen* en otros elementos
  - ▶ Materiales
  - ▶ Luces
  - ▶ Comportamientos, etc.
- Nodos gestión
  - ▶ Vista
  - ▶ Grupos
  - ▶ Transformaciones, etc.

# Ejemplo



# Estructura

- Un nodo raíz, solo uno.
- Nodos grupo
  - ▶ Transformación geométrica
- Nodos hoja
  - ▶ Figuras
  - ▶ Luces
  - ▶ Vistas
  - ▶ Comportamientos
- Nodos componente
  - ▶ Geometría
  - ▶ Material



# Grafos de escena

## Características

- Se separa la representación de la visualización
  - ▶ De la visualización se encarga el motor de rendering
- Orientado a objetos
  - ▶ Cada rama define una componente independiente, adaptable y reusable
- Permite ser modificado fácilmente
  - ▶ En tiempo de desarrollo / ampliación / mantenimiento
  - ▶ También en tiempo de ejecución
  - ▶ En el diseño hay que buscar un compromiso entre la eficiencia y el mantenimiento

# Grafos de escena

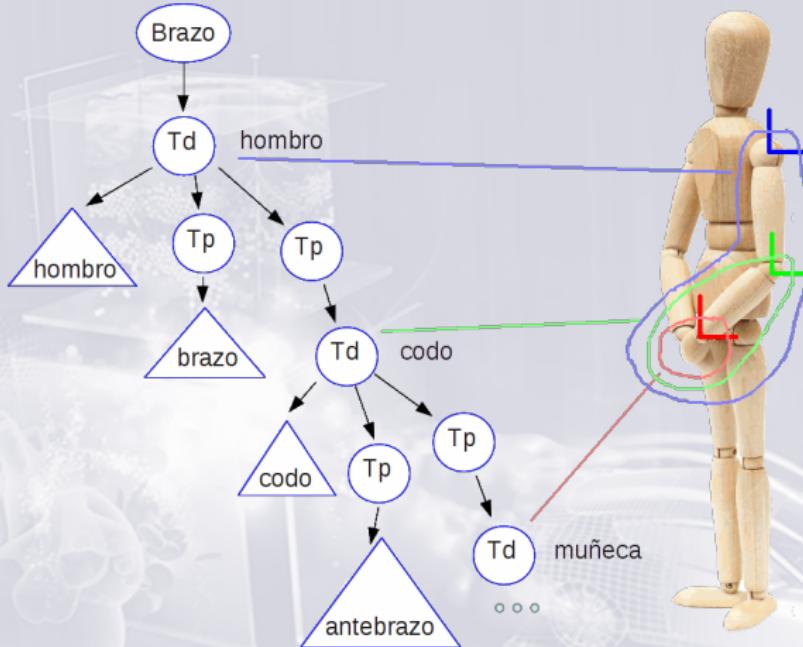
## Características

- Modela estructuras jerárquicas fácilmente
- Ejemplo: Hagamos el siguiente objeto



## Ejemplo 2

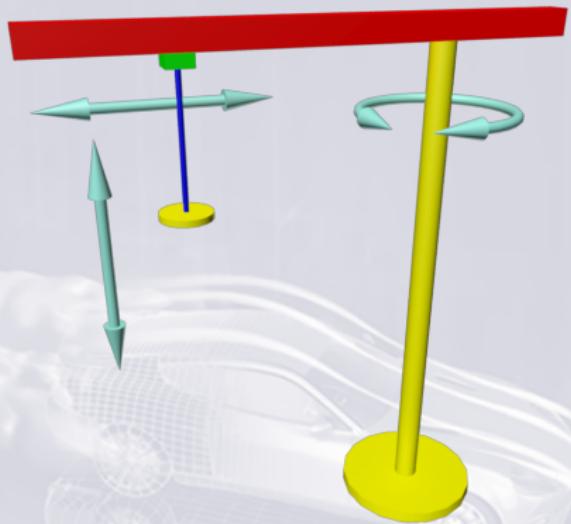
- Grafo de uno de los brazos



- Ejercicio:** Hacer el grafo completo del maniquí

## Ejemplo 3

- Una grúa de obra



# API's de grafos de escena: X3D

eXtensible 3D graphics

[www.web3d.org/x3d/what-x3d](http://www.web3d.org/x3d/what-x3d)



- Desarrollado por el Web3D Consortium
- Oficialmente incorporado al estándar multimedia MPEG-4
- Añade a VRML características XML para facilitar su integración con las tecnologías Web

► *Possible tema de ampliación de conocimientos*



Imagen cortesía de

<http://www.web3d.org/example/historic-vienna>

# OpenSceneGraph

[www.openscenegraph.org](http://www.openscenegraph.org)



- Biblioteca gráfica multiplataforma de software libre
- Completamente escrita en C++ y OpenGL
- Hace un uso intensivo de STL y patrones de diseño.
  - ▶ *Possible tema de ampliación de conocimientos*



Imagen cortesía de

<http://www.openscenegraph.org/index.php/gallery/screenshots>

# Java 3D

[java3d.java.net](http://java3d.java.net)



- API que permite la inclusión de gráficos 3D en Aplicaciones Java de escritorio y Applets
  - Objetivos de diseño: Proporcionar
    - ▶ Un paradigma PDO para el desarrollo de aplicaciones gráficas
    - ▶ Soporte para formatos estándar de archivo 3D (VRML, OBJ, etc.)
- ▶ *Possible tema de ampliación de conocimientos*



# Grafos de escena

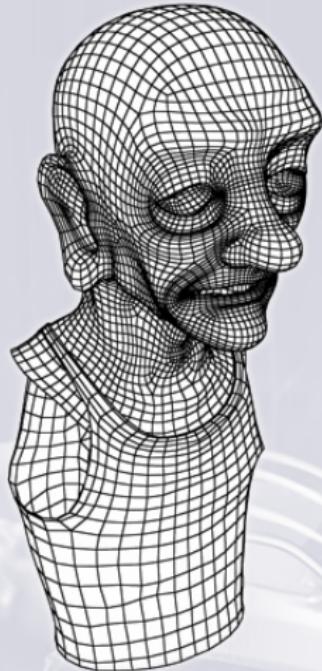
- Elementos en un grafo de escena
    - ▶ Modelos jerárquicos
      - ★ Nodos de geometría
      - ★ Nodos de transformaciones y otros nodos internos
    - ▶ Animación
    - ▶ Interacción
    - ▶ Detección de colisiones
  - En el próximo tema: Visualización
    - ▶ Vistas
    - ▶ Luces
    - ▶ Materiales
- ⇒ La aplicación práctica de la teoría la veremos en **Three.js**

# Representación de las Figuras

Geometría + Aspecto



Geometría



Esquema de Representación: **Boundary Representation (B-Rep)**

# Esquemas de Representación

## • Representación de una figura

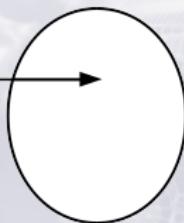
- ▶ Una colección finita de símbolos (de un alfabeto finito) que la describen
- ▶ Ejemplos:
  - ★  $(x - 1)^2 + (y - 2)^2 + (z - 3)^2 \leq 4^2$
  - ★ *Caja*  $((0, 0, 0), (2, 2, 2)) \cup$  *Esfera*  $((2, 2, 2), 1)$
  - ★ *Vertices* =  $\{(0, 0, 0), (1, 0, 0), (0, 1, 0), (0, 0, 1)\}$   
*Caras* =  $\{(0, 1, 3), (0, 2, 1), (0, 3, 2), (1, 2, 3)\}$

## • Esquema de Representación

Figuras

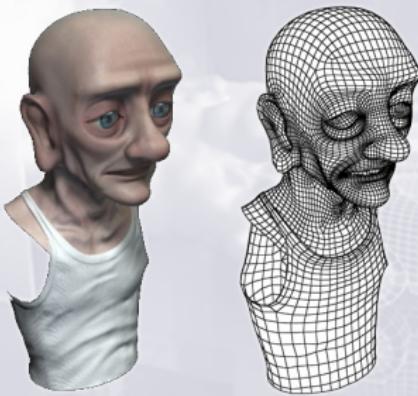


Representaciones



# Representación B-Rep

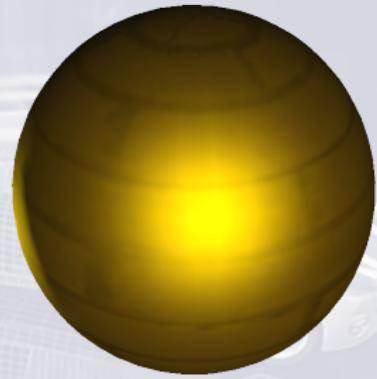
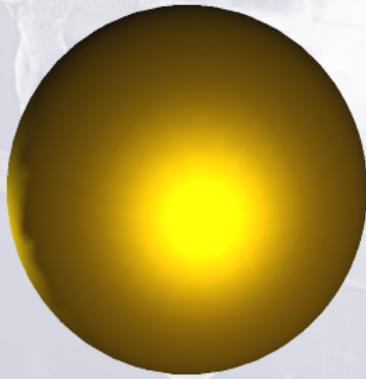
- La figura es representada describiendo la frontera que la delimita
- La frontera es representada mediante un malla de polígonos
- Características de la representación
  - ▶ Es exacta para figuras poliédricas
  - ▶ Es aproximada en las fronteras curvas
    - ★ Mediante técnicas de shading se suple *visualmente* esa desventaja
  - ▶ Es robusta e independiente de su complejidad geométrica



# Representación B-Rep

## Información a almacenar

- Obligatorio
  - ▶ Geometría: Los vértices, sus coordenadas
  - ▶ Topología: Conexiones entre vértices para formar las caras
- Adicionalmente en cada vértice
  - ▶ Vector normal: Necesario para el cálculo de iluminación
  - ▶ Coordenadas de textura: Necesario para la aplicación de texturas



# Representación B-Rep

## Geometría indexada y no indexada

- Geometría no indexada

: Información en una geometría no indexada

```
// Cada vértice aparece varias veces, una vez por cara
// Se dan los vértices en orden antihorario para formar las caras
```

```
Vértices_y_Caras = {C, B, A,     C, A, V,     C, V, B,     A, B, V}
```

- Geometría indexada

: Información en una geometría indexada

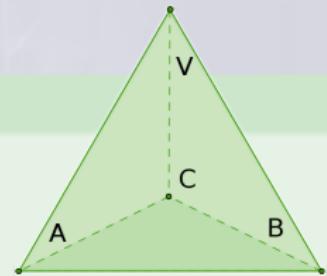
```
// Cada vértice se define una sola vez
```

```
Vértices = {C, A, B, V}
```

```
// Las caras se definen con índices a los vértices
```

```
// Los índices se dan en sentido antihorario
```

```
Caras = {0, 2, 1,     0, 1, 3,     0, 3, 2,     1, 2, 3}
```



# Geometría indexada y no indexada

## Ventajas e inconvenientes

- Ventajas de la geometría indexada
  - ▶ Se ahorra espacio de almacenamiento
  - ▶ La modificación de un vértice es más rápido
  - ▶ Operaciones como saber si dos caras comparten un mismo vértice son más rápidas
- Ventajas de la geometría no indexada
  - ▶ La visualización es más rápida
  - ▶ Permite tener un mismo vértice con varias normales y coordenadas de textura, según la cara desde la que se use dicho vértice
- Las ventajas de una son los inconvenientes de la otra

# Creación de geometría

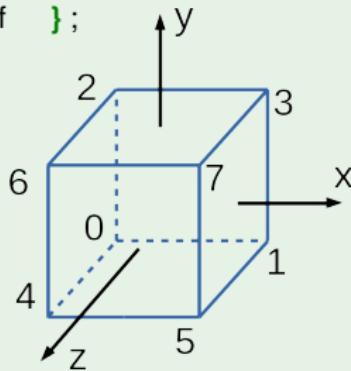
- Manualmente
  - ▶ Se define en el código toda la información
- Proceduralmente
  - ▶ A partir de unos parámetros, un método genera la información
- Operando con primitivas básicas
  - ▶ Se dispone de una geometría básica ya generada
  - ▶ Se construye una geometría compleja operando con la que se tiene en cada momento
- Interactivamente
  - ▶ Al usuario se le dan las herramientas para construir sus figuras “*a golpe de ratón*”
- Cargándola de un archivo
  - ▶ Se genera la geometría leyendo la información de archivos

# Creación manual de geometría

**Ejemplo:** Creación manual de un cubo mediante índices

```
// Coordenadas de los vértices
float[] vertices = {
    -1.0f, -1.0f, -1.0f, 1.0f, -1.0f, -1.0f,
    -1.0f, 1.0f, -1.0f, 1.0f, 1.0f, -1.0f,
    -1.0f, -1.0f, 1.0f, 1.0f, -1.0f, 1.0f,
    -1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f};

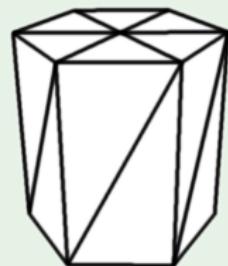
// Índices para formar las caras
int[] indices = {
    0, 2, 3, 1,      // Cara trasera
    4, 5, 7, 6,      // Frontal
    1, 3, 7, 5,      // Derecha
    0, 4, 6, 2,      // Izquierda
    0, 1, 5, 4,      // Inferior
    2, 6, 7, 3 };   // Superior
```



# Creación procedural de geometría

## Ejemplo: Creación procedural de un cilindro

```
Cylinder ( float radius , float height , int resolution ) {
    int nv = (resolution+1)*2;
    float *vertices = new float[nv*3]; // 3 coordenadas/vertice
    vertices[0] = vertices[1] = vertices[2] = 0.0;
    vertices[(resolution+1)*3+0] = 0.0;
    vertices[(resolution+1)*3+1] = height;
    vertices[(resolution+1)*3+2] = 0.0;
    float x, z, a;
    for (int i=0, i < resolution , i++) {
        a = (2.0*PI*i)/resolution;
        x = radius * cos (a); z = radius * sin (a);
        vertices[(i+1)*3+0] = vertices[(i+resolution+2)*3+0] = x;
        vertices[(i+1)*3+1] = 0.0;
        vertices[(i+resolution+2)*3+1] = height;
        vertices[(i+1)*3+2] = vertices[(i+resolution+2)*3+2] = z;
    }
    int ni = resolution * 4 * 3; // 4 triángulos * 3 índices/trí
    int *indices = new int[ni];
    . . .
}
```



# Creación de Geometría

## Three.js

### Creación manual y procedural

- Se usa la clase **PolyhedronGeometry**
- Se definen o calculan los arrays de coordenadas e índices y se le pasan como parámetros al constructor
- **PolyhedronGeometry** (`vertices`, `indices`)

### Ejemplo: Pirámide de base cuadrada

```
var vertices = [ -1,-1,-1, -1,-1,1, 1,-1,1, 1,-1,-1, 0,1,0 ];  
  
var indices = [ 0,3,2, 2,1,0, 0,1,4, 1,2,4, 2,3,4, 3,0,4 ];  
  
var piramide = new THREE.PolyhedronGeometry (vertices ,indices);
```

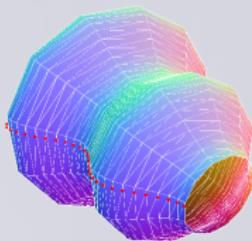
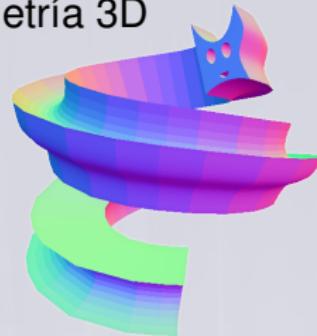
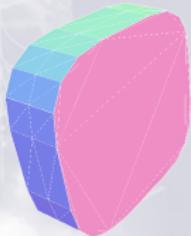
- En el array de vértices, cada 3 valores es un vértice
- En el array de caras, cada 3 valores es un triángulo

# Creación de Geometría

## Operando con geometrías

- Geometría 2D + operación = Geometría 3D

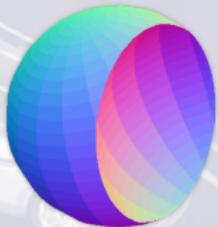
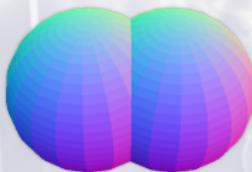
- Extrusión
- Barrido
- Revolución



- Geometrías 3D + operación = Geometría 3D

- Operaciones booleanas

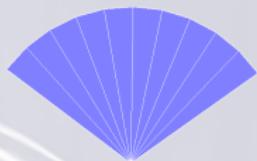
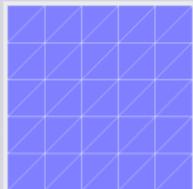
- Unión
- Intersección
- Diferencia



# Creación de geometría 2D

## Three.js

- **PlaneGeometry**: Plano con coordenadas (x, y, 0)
  - ▶ width, height: Anchura y altura del plano
  - ▶ widthSegments, heightSegments: Número de segmentos en cada dimensión. Por defecto, 1
- **CircleGeometry**:
  - ▶ radius: Radio
  - ▶ segments: Mínimo, 3. Por defecto, 8
  - ▶ thetaStart: Ángulo de inicio. Por defecto, 0
  - ▶ thetaLength: Ángulo abarcado. Por defecto,  $2\pi$
- No se usan para construir geometría 3D, no derivan de Shape

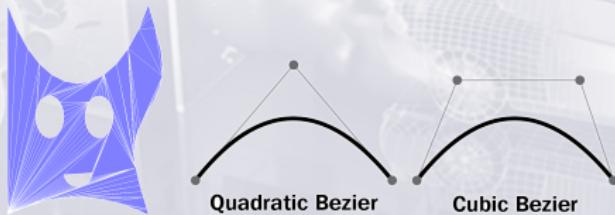


# Creación de geometría 2D

Three.js

## Clase Shape

- Se usa para crear geometría 2D  
También para geometría 3D por extrusión y barrido
- Se va dibujando el contorno exterior mediante diversas órdenes
  - ▶ `moveTo (x, y)`: Movimiento sin dibujar hasta la posición (x,y)
  - ▶ `lineTo (x, y)`: Línea recta desde la posición actual hasta la posición (x,y)
  - ▶ `quadraticCurveTo (aCPx, aCPy, x, y)`: Bezier cuadrática
  - ▶ `bezierCurveTo (aCPx1, aCPy1, aCPx2, aCPy2, x, y)`: Bezier cúbica
  - ▶ `splineThru (pts)`: Spline por los puntos indicados.  
pts es un array de THREE.Vector2



## Clase Shape (2)

## Three.js

- Contornos cerrados con una sola orden
  - ▶ absarc (x,y, radius, aStartAngle, aEndAngle):  
Dibuja un segmento de círculo
  - ▶ absellipse (x,y, xRadius, yRadius, aStartAngle, aEndAngle):  
Dibuja un segmento de elipse



- Añadido de agujeros
  - ▶ Se crean los shapes de los agujeros
  - ▶ Se añaden al contorno exterior con el método holes.push

# Clase Shape (3)

## Three.js

### Ejemplo: Forma libre 2D

```

var shape = new THREE.Shape ();
// Se crea el contorno exterior
shape.moveTo(10, 10);
shape.lineTo(10, 40);
shape.bezierCurveTo(15, 25, 25, 25, 30, 40);
shape.splineThru( [new THREE.Vector2(32, 30),
    new THREE.Vector2(28, 20), new THREE.Vector2(30, 10)] );
shape.quadraticCurveTo(20, 15, 10, 10);
// Agujeros de ojos y boca
var hole = new THREE.Shape ();
hole.absellipse(16, 24, 2, 3, 0, Math.PI * 2);
shape.holes.push(hole);
// El otro ojo con otra elipse de manera similar, ahora la boca
hole = new THREE.Shape ();
hole.absarc(20, 16, 2, Math.PI*2, Math.PI);
shape.holes.push(hole);

```



# Clase Shape (y 4)

# Three.js

- A partir de un Shape se puede construir:

- ▶ Una Geometría 2D

Clase **ShapeGeometry** (unShape)

- ▶ Una Geometría 3D por extrusión o barrido

Clase **ExtrudeGeometry** (unShape, unasOpciones)

# Geometría 3D por extrusión y barrido Three.js

## ● ExtrudeGeometry

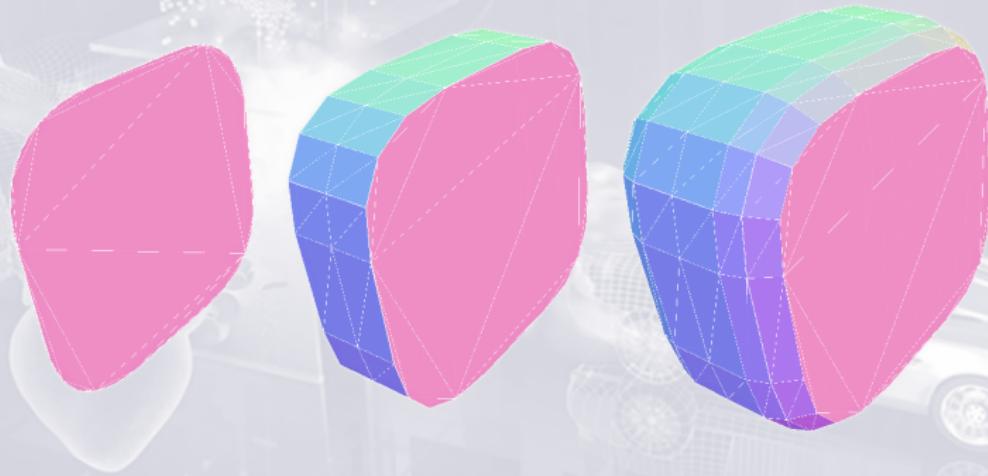
- ▶ shape: Un objeto Shape con el contorno
- ▶ options: Puede tener los siguientes parámetros opcionales
  - ★ amount: La cantidad de estrusión. Por defecto, 100
  - ★ bevelEnabled: Añadido del bisel. Por defecto, true
  - ★ bevelThickness: En la dirección de extrusión. Por defecto, 6
  - ★ bevelSize: En el plano del Shape. Por defecto, bevelThickness - 2
  - ★ bevelSegments: Segmentos para suavizar el bisel. Por defecto, 3
  - ★ curveSegments: Segmentos para las curvas del Shape
  - ★ steps: Segmentos de la parte extruída. Por defecto, 1
  - ★ extrudePath: En caso de que se quiera seguir un camino libre.  
Si no se especifica, se extruye por el eje Z.

# Geometría por extrusión

Three.js

## Ejemplo: Geometría por extrusión

```
var shape = new THREE.Shape();    shape.moveTo (10,10);  
shape.lineTo (20,10); shape.quadraticCurveTo (30,10, 30,20); ...  
  
var options = { amount: 8, steps: 2, curveSegments: 4  
    bevelThickness: 4, bevelSize: 2, bevelSegments: 2 };  
  
var geometry = new THREE.ExtrudeGeometry (shape, options);
```

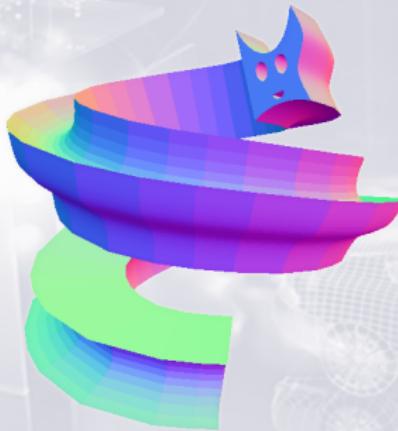


# Geometría por barrido

# Three.js

## Ejemplo: Geometría por barrido

```
var shape = createShape () ; // Método que crea y devuelve un Shape  
// En pts se guarda un array de THREE.Vector3 que define el camino  
var path = new THREE.CatmullRomCurve3 (pts) ;  
var options = { steps: 50, curveSegments: 4, extrudePath: path } ;  
var geometry = new THREE.ExtrudeGeometry (shape, options) ;
```



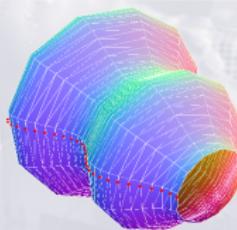
# Geometría por revolución

- **LatheGeometry**

- ▶ points:  
Un array de THREE.Vector2 con puntos para construir el perfil
- ▶ segments, phiStart, phiLength: Parámetros opcionales.  
Valores por defecto: 12, 0,  $2\pi$
- ▶ La revolución se realiza por el eje Y

## Ejemplo: Geometría por revolución

```
var points = [ new THREE.Vector2 (1,-2), ... ];  
var shapeGeometry = new THREE.LatheGeometry (points, 24, 0, Math.PI);
```



# Geometría 3D

## Three.js

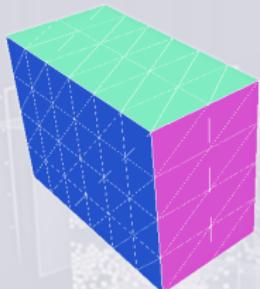
### Primitivas básicas: Cubo, Esfera, Cilindro y Toro

- **CubeGeometry** (width, height, depth,  
widthSegments, heightSegments, depthSegments)
  - ▶ El valor por defecto para el número de segmentos es 1
- **SphereGeometry** (radius, widthSegments, heightSegments,  
phiStart, phiLength, thetaStart, thetaLength)
  - ▶ Valores por defecto: (50, 8, 6, 0,  $2\pi$ , 0,  $\pi$ )
- **CylinderGeometry** (radiusTop, radiusBottom, height,  
segmentsX, segmentsY, openEnded)
  - ▶ Valores por defecto: (20, 20, 100, 8, 1, false)
- **TorusGeometry**  
(radius, tube, radialSegments, tubularSegments, arc)
  - ▶ Valores por defecto: (100, 40, 8, 6,  $2\pi$ )

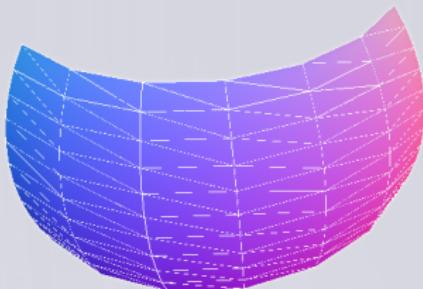
# Cubo, esfera, cilindro y toro

## Ejemplos

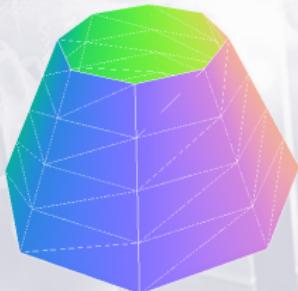
Three.js



CubeGeometry (10,15,20, 2,4,6)



SphereGeometry (15,6,10, 0,2,1,1)



CylinderGeometry (10,20,20, 8,4,true)



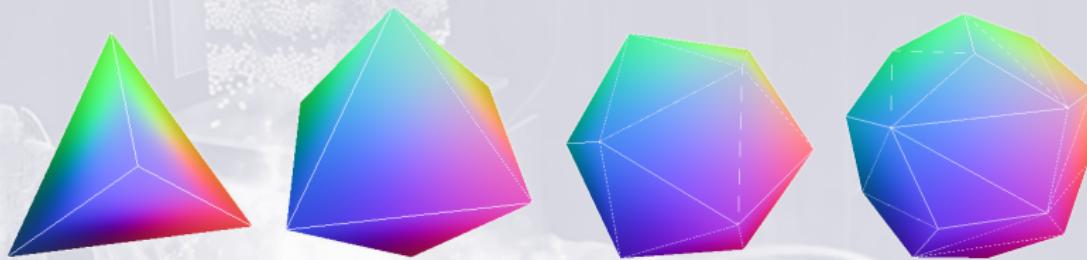
TorusGeometry (10,5, 6,12, 3)

# Geometría 3D

## Algunos poliedros regulares

### Three.js

- TetrahedronGeometry, OctahedronGeometry, IcosahedronGeometry, DodecahedronGeometry
  - ▶ radius: El radio de la figura. Por defecto, 1



- Tanto las primitivas básicas como los poliedros regulares se crean con el origen de coordenadas en su centro

# Geometría mediante operaciones booleanas

- Se parte de 2 operandos
- Se obtiene 1 resultado
- Operaciones:
  - ▶ Unión
    - ★ Se toma TODA la materia de los operandos para construir el resultado
  - ▶ Intersección
    - ★ Se toma SOLO la materia COMÚN de los operandos para construir el resultado
  - ▶ Diferencia
    - ★ No es commutativa
    - ★ El resultado es el primer operando MENOS la materia que ocupa el segundo operando



# Operaciones booleanas

# Three.js

- Se tiene que usar la extensión ThreeBSP
  - ▶ <https://github.com/skalnik/ThreeBSP>
  - ▶ Escrita en CoffeeScript, hay que compilarla hacia JavaScript
- Procedimiento
  - ① Los operandos primitivos se convierten a nodos ThreeBSP
  - ② Se opera entre nodos ThreeBSP,  
el resultado es un nodo ThreeBSP
    - ★ Unión, método `union`
    - ★ Intersección, método `intersect`
    - ★ Diferencia, método `subtract`
  - ③ Con el resultado final
    - ① Se convierte a un Mesh de Three
    - ② Se calculan los vectores normales

# Operaciones booleanas (ejemplo)

Three.js

- ( unaEsfera — otraEsfera ) — unaCaja

## Operaciones booleanas: Código HTML

```
<script type="text/javascript" src="ThreeBSP.js"></script>
```

## Operaciones booleanas: Código JavaScript

```
// Previamente se crean las geometrías
var sphere1 = new THREE.SphereGeometry ( ... );
// Se construyen nodos ThreeBSP
var sphere1bsp = new ThreeBSP (sphere1);
// Se construye el arbol binario con las operaciones
var partialResult = sphere1bsp.subtract (sphere2bsp);
var finalResult = partialResult.subtract (boxBsp);
// Y finalmente
var result = finalResult.toMesh (material);
result.geometry.computeFaceNormals ();
result.geometry.computeVertexNormals ();
```



# Creación de una geometría por partes Three.js

- Se puede crear una única geometría juntando varias componentes
- Se deja de tener acceso a los distintos componentes

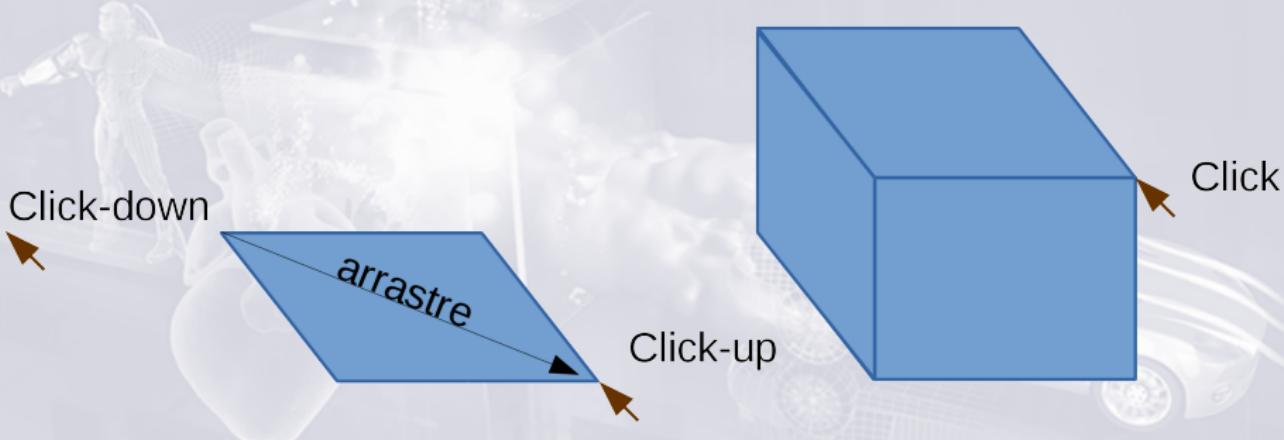
## Ejemplo: Creación de una geometría por partes

```
var geometry = new THREE.Geometry();  
var component1 = createFirstComponent();  
var component2 = createSecondComponent();
```

```
THREE.GeometryUtils.merge(geometry, component1);  
THREE.GeometryUtils.merge(geometry, component2);
```

# Creación de geometría interactivamente

- Requiere procesar adecuadamente los eventos del ratón según el contexto
- Se van completando las listas de vértices e índices necesarias para construir la geometría
- Hay que ir mostrando el resultado parcial mientras se construye, el usuario necesita una realimentación



# Geometría eficiente

## Geometría en GPU

- Las instrucciones vistas crean la geometría en CPU
- Es más eficiente tenerla en Buffer Objects en la GPU
- Algunas instrucciones tienen versión para GPU, por ejemplo:
  - ▶ BoxGeometry (CPU) tiene versión para GPU, `BoxBufferGeometry`
  - ▶ Consultar las disponibles en [threejs.org](http://threejs.org)
- Se puede crear una geometría en GPU  
a partir de cualquier geometría de CPU

### Ejemplo: Geometría en GPU

```
var geometria = . . .
// Construcción de una geometría por cualquiera de los medios vistos

var geometriaGPU = new THREE.BufferGeometry();
geometriaGPU.fromGeometry (geometria);
```

# Construcción de una figura

# Three.js

- Una figura, un Mesh en Three, se compone de:
  - ▶ Una geometría
  - ▶ Un material
- Se usa la clase **Mesh**

## Ejemplo: Creación de un Mesh

```
var geometry = new THREE.SphereGeometry ( 5, 8, 8 );
var material = new THREE.MeshBasicMaterial ( { color: 0x34AAE6 } );
var sphere = new THREE.Mesh ( geometry, material );
```



# Cargar el modelo desde un archivo

- Preferible cuando se desea modelar objetos complejos
- Three puede importar el formato de Wavefront Technologies
- Los modelos se buscan en internet usando los términos `wavefront model`
  - ▶ En la página [tf3dm.com](http://tf3dm.com) hay bastantes



Modelo realizado por Turn 10 Studios descargado de [tf3dm.com](http://tf3dm.com)

# Descripción del formato

- El formato se basa en archivos de texto
  - ▶ El archivo `.obj` puede contener:
    - ★ Vértices
    - ★ Coordenadas de textura
    - ★ Vectores normales
    - ★ Definición de polígonos, normalmente triángulos
  - ▶ El archivo `.mtl` contiene definiciones de materiales
- Desde Blender se puede exportar a `.obj`

# Ejemplo de archivo .obj

## Ejemplo: Parte de un archivo .obj

```
# Wavefront OBJ file
# Archivo de materiales
mtllib california.mtl
# Vértices El primero es el 1 y así sucesivamente
v 0.80341 -0.22432 0.07015
v 0.83425 -0.22218 0.11258
.
.
#
# Coordenadas de textura
vt 0.02089 0.93665
vt 0.02412 0.92041
.
.
#
# Vectores normales
vn -0.51776 -0.85330 -0.06165
vn -0.59393 -0.72737 -0.34377
.
.
#
# Triángulos vértice / coord.textura / normal
usemtl redColor # Los siguientes triángulos tienen este material
f 4529/1/1 4530/2/2 4531/3/3
f 4532/4/4 4529/1/1 4531/3/3
.
.
```

# Ejemplo de archivo .mtl

## Ejemplo: Parte de un archivo .mtl

```
# Blender MTL file

newmtl redColor
Ns 1000.000000
Ka 0.019482 0.019482 0.019482
Kd 0.800000 0.062764 0.079847
Ks 0.386484 0.780355 0.700265
Ni 1.000000
d 1.000000
illum 2

newmtl tire
Ka 0.8 0.8 0.8
Kd 1 1 1
Ks 0.376471 0.376471 0.376471
illum 2
Ns 27.8576
map_Kd tireA0.png
```

## Carga de modelos .obj

## Three.js

- Se requiere incorporar las siguientes bibliotecas
  - ▶ OBJLoader.js
  - ▶ MTLLoader.js
- Se cargan los materiales y el modelo (en ese orden)



# Carga de modelos .obj (ejemplo)

# Three.js

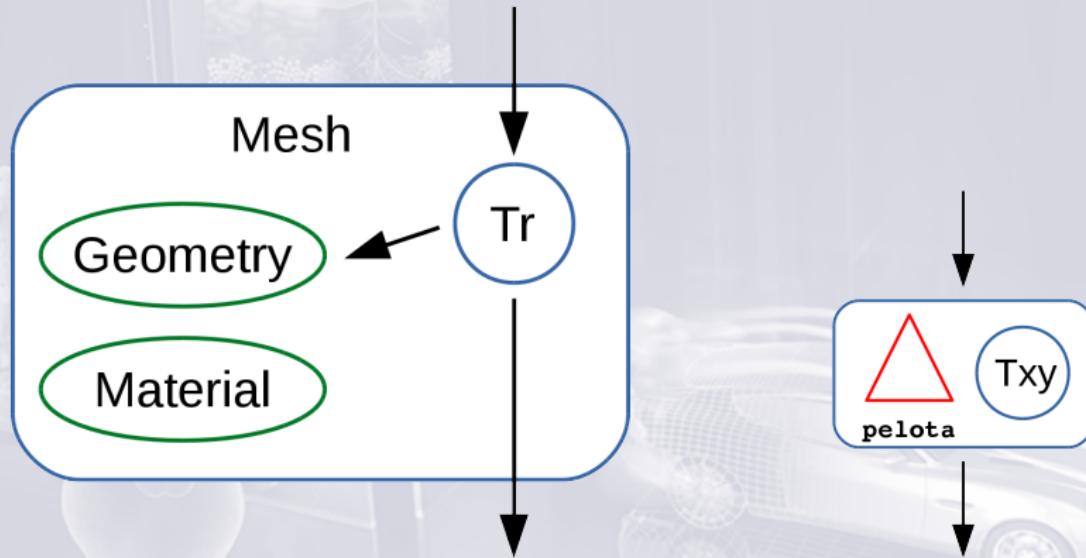
## Ejemplo: Carga de modelos .obj

```
// Se cargan los materiales
var mtlLoader = new THREE.MTLLoader();
mtlLoader.setBaseUrl ('porsche911/');
mtlLoader.setPath ('porsche911/');
mtlLoader.load ('911.mtl', function (materials) {
    materials.preload();

    // Ahora se carga el modelo
    var objLoader = new THREE.OBJLoader();
    objLoader.setMaterials (materials);
    objLoader.setPath ('porsche911/');
    objLoader.load ('Porsche_911_GT2.obj', function (object) {
        // Se configura (si es necesario) y se usa
        object.rotation.y = 2;
        scene.add (object);
    });
});
```

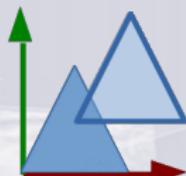
# La clase Mesh

- La clase **Mesh**, además de Geometría y Material
- Incorpora un nodo de Transformación



# Repasso a las transformaciones geométricas

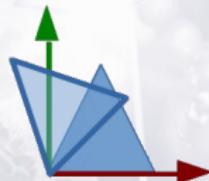
- Traslación



- Escalado uniforme



- Rotación



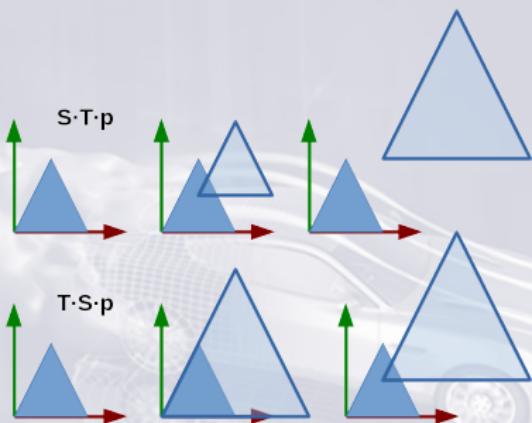
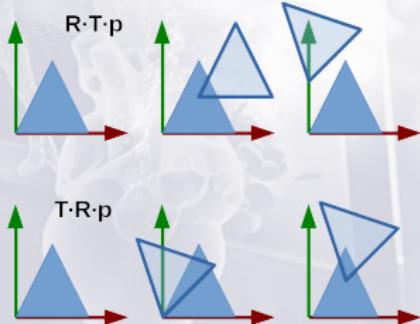
¡Cuidado!

El escalado y la rotación se realizan respecto al origen de coordenadas

# Repaso a las transformaciones geométricas

## Composición no comutativa de transformaciones

- Composiciones conmutativas
  - ▶ Cualquier combinación de escalados uniformes y rotaciones
  - ▶ Cualquier combinación de traslaciones
- Composiciones no conmutativas
  - ▶ Las traslaciones con las rotaciones o los escalados



# Transformaciones geométricas

Three.js

## Su gestión en la clase Mesh

- Se configuran fácilmente mediante sus atributos:
  - ▶ `position`, `rotation`, `scale`
  - ▶ Traslación, rotación y escalado, respectivamente
  - ▶ Son atributos de la clase `THREE.Vector3`

## Ejemplo: Gestión de la transformación de un Mesh

```
var pelota = new THREE.Mesh (unaGeometria, unMaterial);  
pelota.rotation.y = 1; // Las rotaciones son en radianes  
pelota.position.x = 5;  
pelota.scale.z = 2;
```

- ¿En qué orden se aplican las transformaciones?
  - 1 Escalado
  - 2 Rotaciones (1º Rotación en Z, 2º en Y, 3º en X)
  - 3 Traslación

# Transformaciones de un Mesh

¿Cuándo se aplican?

## Three.js

- Al recorrer el grafo para visualizar un frame, en cada nodo Mesh

- 1 `this.matrix = position · rotation · scale`
- 2 `this.matrixWorld = parent.matrixWorld · this.matrix`
- 3 `this.matrixWorld · this.geometry`

- Si se modifican los atributos de transformación entre Frames
  - Animación
- Si nunca se modifican dichos atributos entre Frames
  - Se pierde tiempo recalculando `matrix`
  - Se puede evitar poniendo el atributo `matrixAutoUpdate` a `false`
  - Hay que llamar a `updateMatrix()` para que `matrix` se calcule la primera vez

# Otros métodos de transformación

Three.js

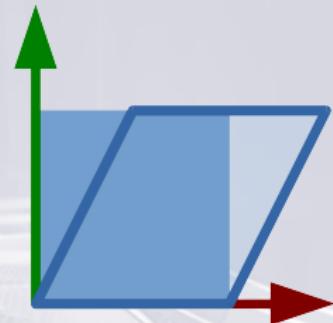
(Métodos de Object3D, heredados en Mesh)

- `translateOnAxis (dirección, distancia)`
  - ▶ Se traslada el Mesh una distancia en una dirección concreta
  - ▶ dirección es un Vector3 que debe estar normalizado
  - ▶ Es acumulativo a lo que ya tenga position
- `rotateOnAxis (eje, ángulo)`
  - ▶ Se rota el Mesh un ángulo respecto a un eje concreto
  - ▶ eje es un Vector3 que debe estar normalizado
  - ▶ Es acumulativo a lo que ya tenga rotation
- `applyMatrix (matriz)`
  - ▶ `this.matrix = matriz · this.matrix`
  - ▶ Se actualizan position, rotation y scale para mantener la consistencia entre ambos modos de representar la transformación

# Transformaciones mediante matrices Three.js

## Clase Matrix4

- El constructor, que no tiene parámetros, construye la identidad
- Métodos habituales:
  - ▶ makeTranslation (x,y,z)
  - ▶ makeRotationX (ángulo)
  - ▶ makeRotationY (ángulo)
  - ▶ makeRotationZ (ángulo)
  - ▶ makeRotationAxis (eje, ángulo)
  - ▶ makeScale (x,y,z)
  - ▶ makeShear (x,y,z) → deslizamiento
  - ▶ multiply (otraMatriz)
  - ▶ multiplyMatrices (unaMatriz, otraMatriz)
  - ▶ set ( $v_{11}$ ,  $v_{12}$ , ...,  $v_{43}$ ,  $v_{44}$ ) →  $v_{fila,columna}$



# Sistema de coordenadas local

## Three.js

- Las figuras predefinidas se crean centradas con respecto a su sistema de coordenadas local
- Se puede modificar su posición y/u orientación aplicando una transformación **permanente** a la geometría, usando el método:  
`applyMatrix (unaTransformación)`
- Esta transformación **no** la heredan sus hijos, se ha transformado la geometría, no el nodo mesh

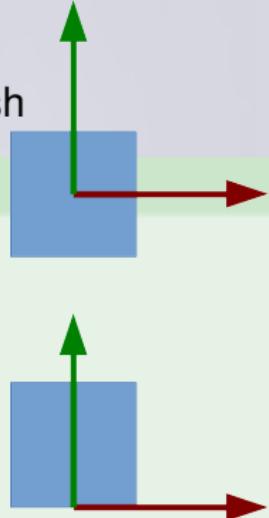
### Ejemplo:

```
var c1 = new THREE.CylinderGeometry (1,1,2);
```

```
var c2 = new THREE.CylinderGeometry (1,1,2);
```

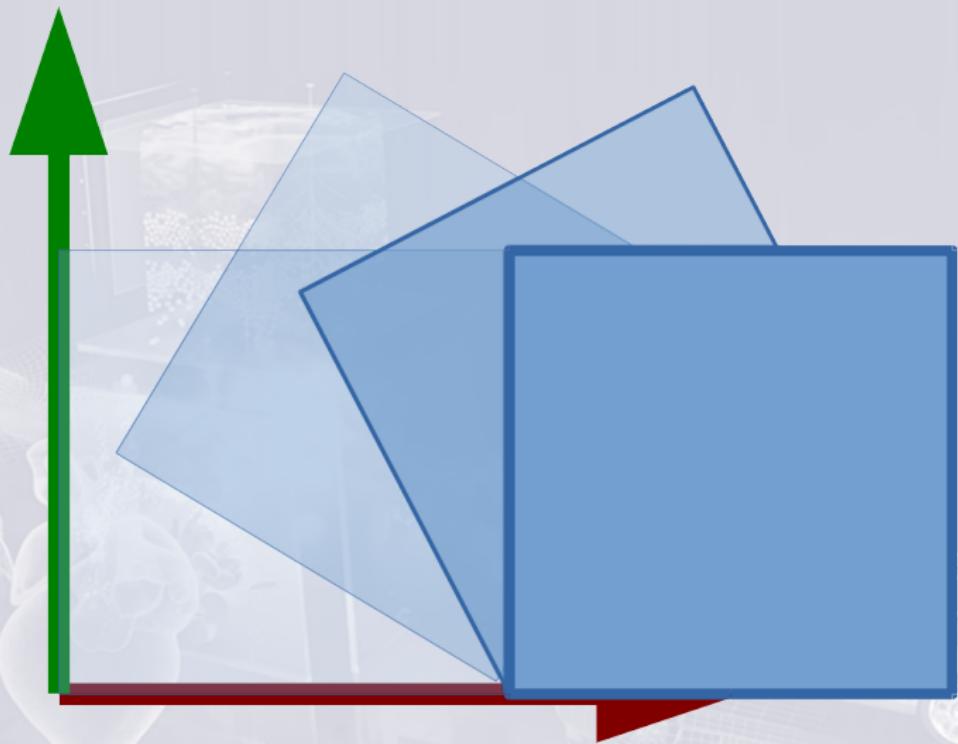
```
var t = new THREE.Matrix4().makeTranslation (0,1,0);
c2.applyMatrix (t);
```

// OJO: Se ha transformado la Geometry, no el Mesh

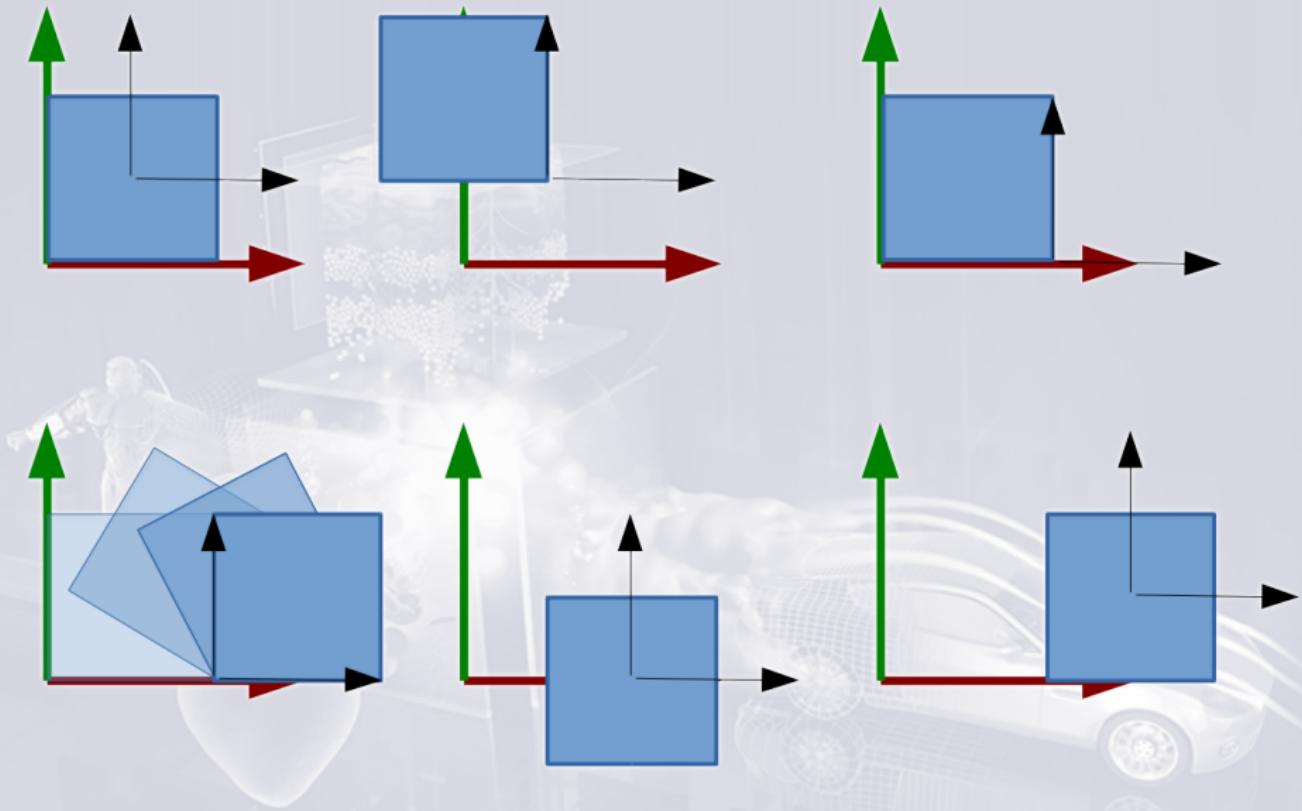


# Ejemplo

## Rodamiento de un cubo



# Rodamiento de un cubo



# Rodamiento del cubo

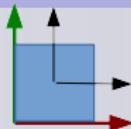
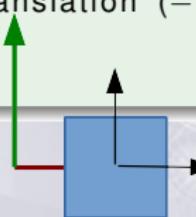
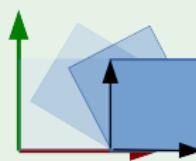
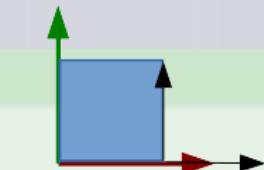
Three.js

Ejemplo: Rodamiento de un cubo

```
// Pre - rotación
cube.geometry.applyMatrix (
    new THREE.Matrix4().makeTranslation (-widht/2, widht/2, 0));
cube.position.x += widht/2;
cube.position.y += -widht/2;

// Rotación, varios Frames modificando amount
cube.geometry.applyMatrix (
    new THREE.Matrix4().makeRotationZ (amount));

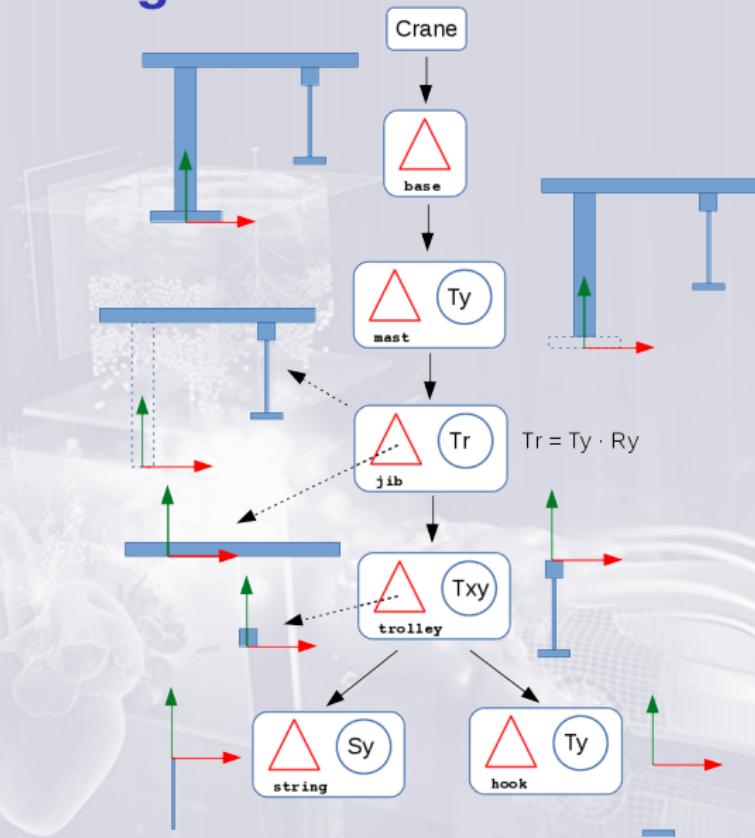
// Post - rotación
cube.geometry.applyMatrix (
    new THREE.Matrix4().makeTranslation (-widht/2, -widht/2, 0));
cube.position.x += widht/2;
cube.position.y += widht/2;
```



# Nodos internos

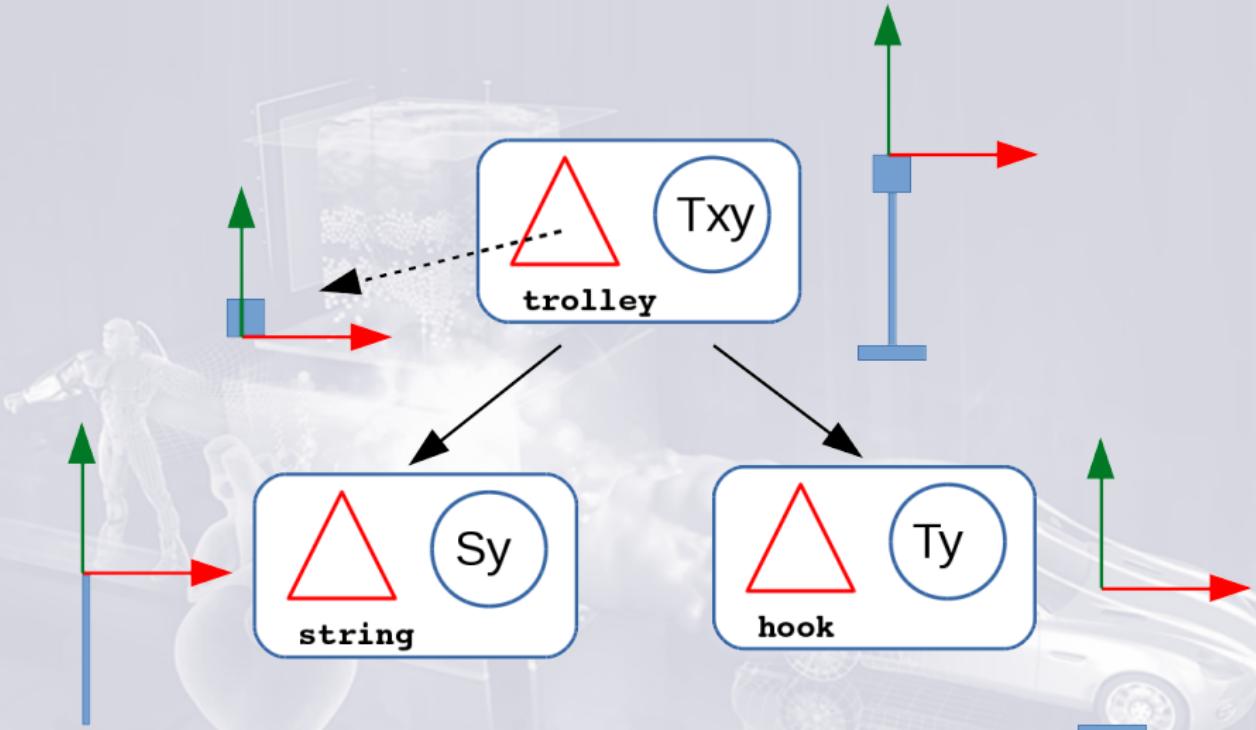
- Se usa la clase `THREE.Object3D` como nodo
- Atributos
  - ▶ `name`: Un nombre, opcional y no necesariamente único
  - ▶ `parent`: Referencia a su nodo padre, solo uno
  - ▶ `children`: Un array de hijos
  - ▶ Atributos para representar la transformación
    - ★ Los mismos que se han explicado para Mesh
      - `position`, `rotation`, etc.
    - ★ De hecho Mesh deriva de Object3D
      - Un Mesh también actúa como nodo interno
- Métodos para montar la jerarquía
  - ▶ `add (object)`: Se añade `object` como hijo
  - ▶ `remove (object)`: Se elimina `object` como hijo
  - ▶ `getObjectByName (string)`: Devuelve el objeto con dicho nombre

# Ejemplo: Una grúa



# Grúa

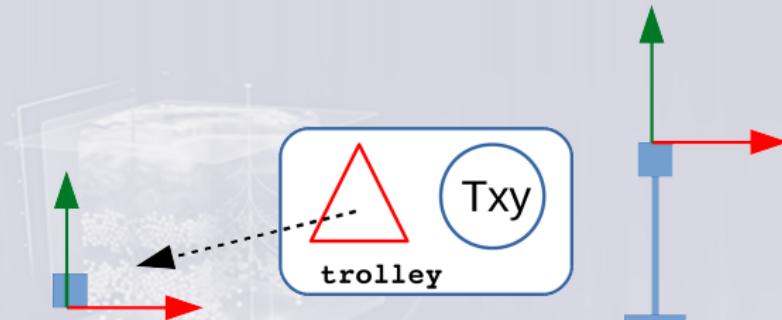
## Conjunto Pluma-Cuerda-Gancho



# Conjunto Pluma-Cuerda-Gancho

Three.js

Pluma



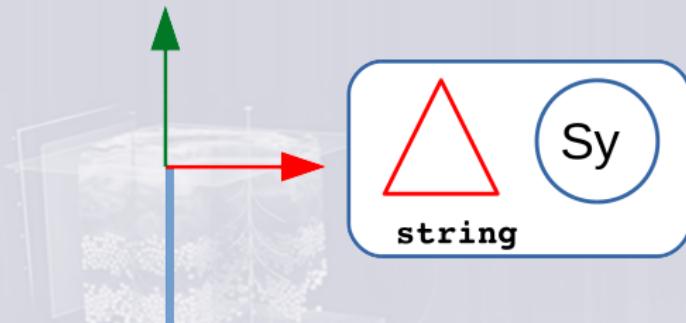
## Conjunto Pluma-Cuerda-Gancho: Pluma

```
var createTrolleyStringHook = function () {
    trolley = new THREE.Mesh (new THREE.BoxGeometry ( . . . ), material);
    trolley.geometry.applyMatrix (
        new THREE.Matrix4().makeTranslation (0, trolleyHeight/2, 0));
    trolley.castShadow = true;
    trolley.position.y = -trolleyHeight; // Baja la pluma y sus hijos
    trolley.position.x = distanceMin; // Desplaza la pluma y sus hijos
    ...
}
```

# Conjunto Pluma-Cuerda-Gancho

Three.js

Cuerda



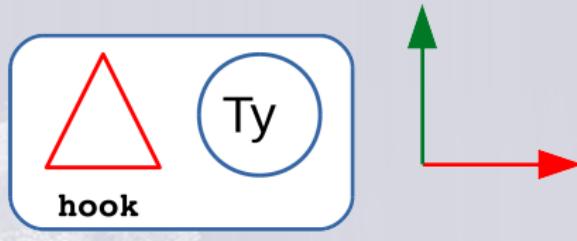
## Conjunto Pluma-Cuerda-Gancho: Cuerda

```
    . . .
    string = new THREE.Mesh (
        new THREE.CylinderGeometry (0.1, 0.1, 1.0), mat);
    string.geometry.applyMatrix (
        new THREE.Matrix4().makeTranslation (0, -0.5, 0));
    string.castShadow = true;
    stringLength = computeStringLength ();
    string.scale.y = stringLength;
    trolley.add (string);
    . . .
```

# Conjunto Pluma-Cuerda-Gancho

Gancho

Three.js

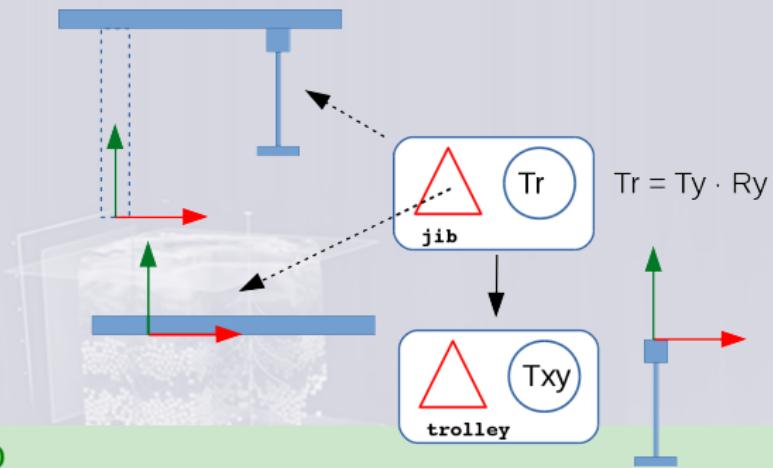


## Conjunto Pluma-Cuerda-Gancho: Gancho

```
hook = new THREE.Mesh ( new THREE.CylinderGeometry ( . . . ) , mat );
hook.geometry.applyMatrix (
  new THREE.Matrix4 () . makeTranslation ( 0 , -baseHookHeight / 2 , 0 ) );
hook.castShadow = true ;
hook . position . y = -stringLength ;
trolley . add ( hook );
return trolley ;
}
```

## Grúa

Brazo



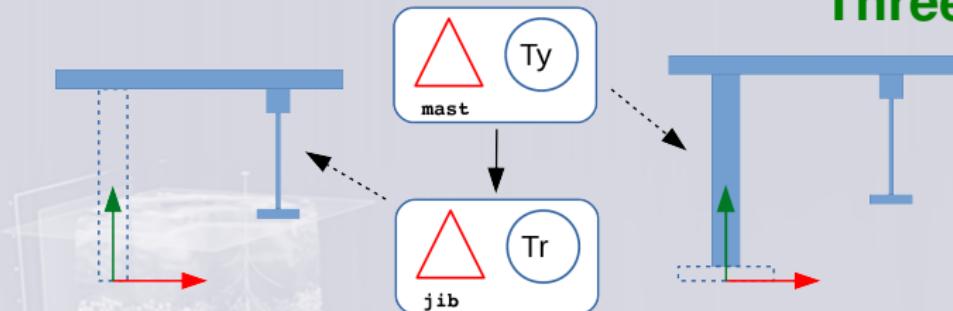
Three.js

## Grúa: Brazo

```
var createJib = function () {
    jib = new THREE.Mesh (new THREE.BoxGeometry ( . . . ), material);
    jib.geometry.applyMatrix (
        new THREE.Matrix4().makeTranslation ( . . . ));
    jib.castShadow = true;
    jib.position.y = craneHeight;
    jib.rotation.y = angle;
    jib.add (createTrolleyStringHook());
    return jib;
}
```

# Grúa

## Mástil



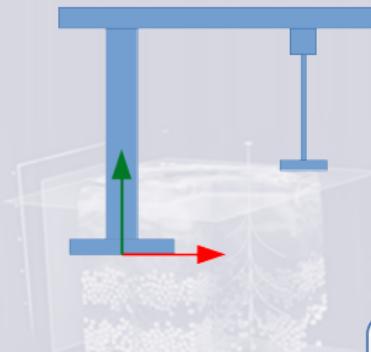
# Three.js

## Grúa: Mástil

```
var createMast = function () {
    var mast = new THREE.Mesh (new THREE.CylinderGeometry ( . . . ) , mat);
    mast.geometry.applyMatrix (
        new THREE.Matrix4 () . makeTranslation ( . . . ) );
    mast.castShadow = true ;
    mast.position.y = baseHookHeight;
    mast.autoUpdateMatrix = false ;
    mast.updateMatrix ();
    mast.add (createJib ());
    return mast;
}
```

# Grúa

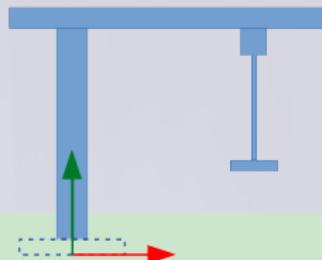
Base



Crane



# Three.js



Grúa: Base

```
var createBase = function () {
    var base = new THREE.Mesh (new THREE.CylinderGeometry ( . . . ) , mat) ;
    base . geometry . applyMatrix (
        new THREE.Matrix4 () . makeTranslation (0 , baseHookHeight/2 , 0));
    base . castShadow = true ;
    base . autoUpdateMatrix = false ;
    base . add (createMast ()) ;
    return base ;
}
```

# Animación

## Introducción

- **Animación:** Creación de la ilusión de que las cosas cambian.
- Se basa en el fenómeno de la persistencia de la visión.
- Percepción de movimiento: 24 imágenes por segundo.



# Clasificación

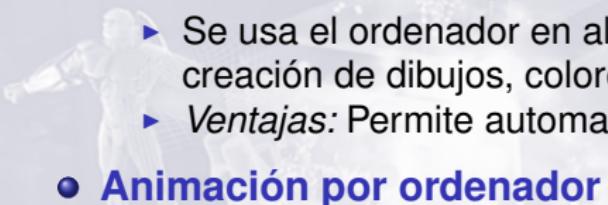
## ● Animación convencional

- ▶ Orientada principalmente a animación 2D con apariencia plana
- ▶ *Ventajas:* Mayor flexibilidad y expresividad en los personajes
- ▶ *Desventajas:* Creación de todos los dibujos a mano



## ● Animación asistida por ordenador

- ▶ Se usa el ordenador en algunas fases del proceso: creación de dibujos, coloreado, ...
- ▶ *Ventajas:* Permite automatizar ciertos procesos reiterativos



## ● Animación por ordenador

- ▶ Orientada principalmente a animación 3D con entornos complejos
- ▶ *Ventajas:* Automatismo y manejo de grandes cantidades de información
- ▶ *Desventajas:* Falta de expresividad

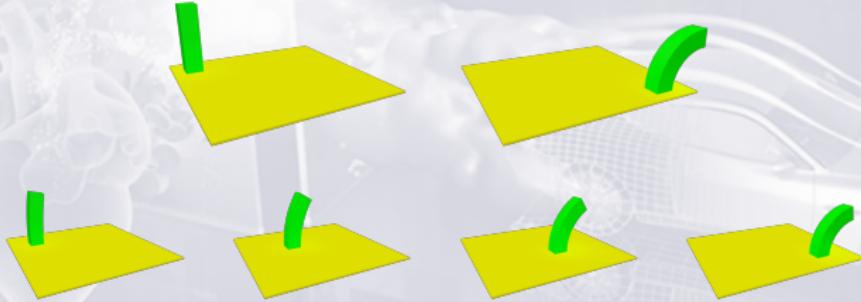


# Animación Convencional vs. por Ordenador

- Animación Convencional (o Asistida)



- Animación por Ordenador



# Animación por ordenador

El problema de la falta de expresividad

## ● Motion capture

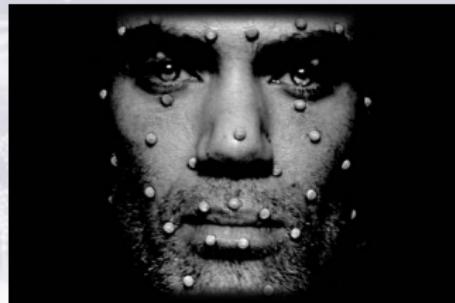
- ▶ Se busca dotar de expresividad humana a los personajes ...



# Animación por ordenador

## Motion capture

- ... también a nivel expresión facial



# Etapas en una animación por ordenador

- Un corto de animación por ordenador requiere varias etapas
  - ▶ Guion, Storyboard, grabación de los diálogos, etc.  
(los detalles en la asignatura de 4º)
- **Guion (Script)**

Uno de los aspectos más importantes de la animación.

¿Qué se quiere contar?

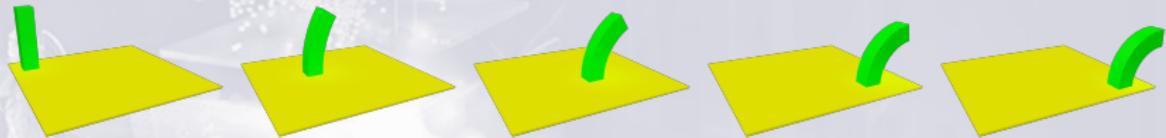
- **Esquema de la historia (Storyboard)**

Resumen gráfico (en viñetas) de la historia



# Modos de implementar la animación

- Animación procedural
  - ▶ Modificando valores de parámetros
- Mediante escenas clave
  - ▶ Se indican valores concretos de parámetros en frames concretos
  - ▶ El ordenador calcula los valores en los frames intermedios



- Mediante caminos
  - ▶ La posición de un objeto viene determinada por una línea



# Animación procedural

## Three.js

- Cada objeto animable dispone de un método que:
  - ▶ Lee el tiempo
  - ▶ Modifica los parámetros que correspondan
- Dicho método es llamado para cada frame
- Permite una animación muy personalizada

### Algoritmo: Método `animate` de la clase `Game`

```
Game.deltaTime = this.clock.getDelta();  
  
this.gameObjects.forEach (function (gameObject) {  
    gameObject.update();  
});
```

La clase Game sería una clase que puede controlar diversos aspectos del juego.

En el ejemplo, tiene la lista de objetos que hay que animar en cada frame.

# Animación procedural

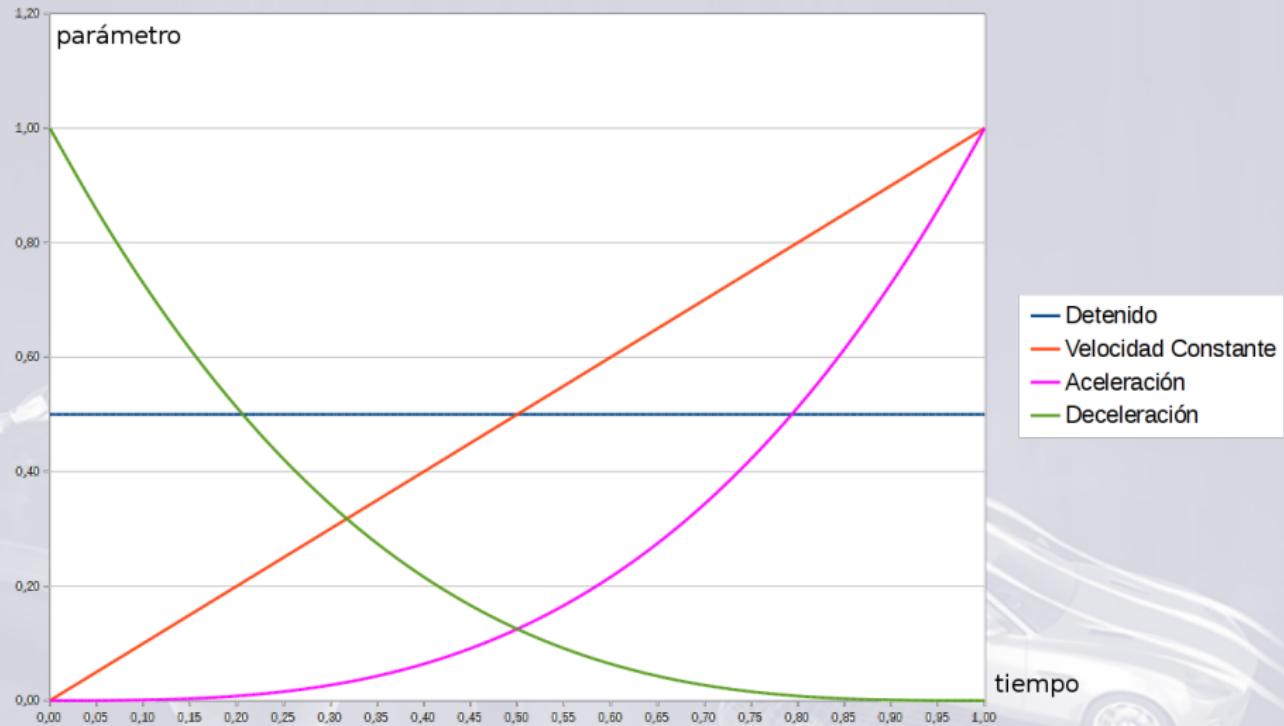
## Método update

- Tiene toda la responsabilidad de la animación
- Debe conocer qué movimientos puede hacer la figura
- Saber en qué estado se encuentra cada movimiento
- Saber y controlar en qué momento ocurre cada cosa y a qué velocidad



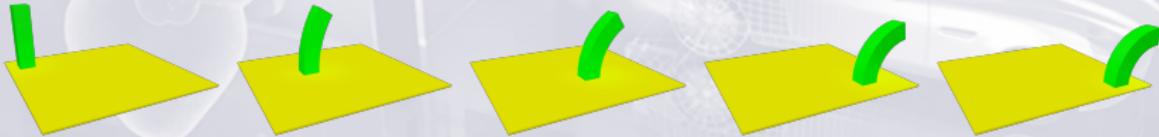
Curvas de función: Controlan la velocidad de cambio de los parámetros

# Curvas de función



# Animación mediante escenas clave

- La animación se analiza y descompone en movimientos sencillos
- En cada movimiento se eligen momentos importantes
  - ▶ El principio, el final, tal vez algún momento intermedio
- Esos momentos importantes son las **Escenas clave**
- Para cada escena clave se definen los valores concretos de los parámetros que determinan la posición de la figura.
- La animación queda configurada mediante:
  - ▶ La definición de cada escena clave
  - ▶ El tiempo que transcurre entre cada 2 escenas clave
  - ▶ La definición de la gráfica que controla el ritmo de ese movimiento
  - ▶ La definición de cómo se encadenan los diferentes movimientos



# Animación mediante escenas clave

Three.js

- Se usa la biblioteca **Tween.js**
  - ▶ <https://github.com/tweenjs/tween.js/>
- Cada animación Tween es un movimiento entre un origen y un destino
- Se definen 2 variables locales con los parámetros a usar en el movimiento
  - ▶ Cada variable local contiene los valores para los parámetros
  - ▶ Una variable con los valores para el origen y otra para el destino
- La animación se completa indicando:
  - ▶ El tiempo, en ms, que transcurre entre el origen y el destino
  - ▶ Cómo se modifican los parámetros de las figuras según los parámetros de las variables locales usadas en la animación

# Animación con Tween

# Three.js

## Ejemplo: Uso de Tween.js

```
// Inclusión de la biblioteca
<script src="../libs/tween.js"></script>

// Variables locales con los parámetros y valores a usar
var origen = { x: 0, y: 300 };
var destino = { x: 400, y: 50 };

// Definición de la animación: Variables origen, destino y tiempo
var movimiento = new TWEEN.Tween(origen).to(destino, 2000); // 2 seg

// Qué hacer con esos parámetros
movimiento.onUpdate (function(){
    mesh.position.x = origen.x;
    mesh.position.y = origen.y;
});

// La animación comienza cuando se le indique
movimiento.start();

// Hay que actualizar los movimientos Tween en la función de render
TWEEN.update();
```

# Biblioteca Tween.js

## Control de la velocidad (1)

- Se realiza con el método `easing(param)`

- Donde `param` puede ser

- ▶ Velocidad constante  
`TWEEN.Easing.Linear.None`

- ▶ Aceleración al empezar y/o deceleración al acabar  
`TWEEN.Easing.Quadratic.InOut`

- ★ Cambiando `InOut` por `In` o `Out`, hace que sea solo aceleración o deceleración

- ★ Cambiando `Quadratic` por `Cubic`, `Quartic`, `Quintic`, `Exponential` se consigue una mayor aceleración/deceleración

# Biblioteca Tween.js

## Control de la velocidad (y 2)

- ▶ Con retroceso  
TWEEN.Easing.Back.InOut
- ▶ Elástico  
TWEEN.Easing.Elastic.InOut
- ▶ Rebote  
TWEEN.Easing.Bounce.InOut
  - ★ En los tres, cambiando InOut por In o Out, hace que el efecto se produzca solo al principio o al final

# Biblioteca Tween.js

## Ajuste de otros aspectos de la animación

- Número de repeticiones
  - ▶ Método `repeat (n)`  
Se puede indicar `Infinity` para repeticiones infinitas
- Movimiento de vaivén
  - ▶ Método `yoyo (true)`
- Acciones a realizar antes y después de la animación
  - ▶ Método `onStart (function () { ... })`
  - ▶ Método `onComplete (function () { ... })`
- Encadenamiento de animaciones
  - ▶ Método `chain (otraAnimacion)`
- Detención de una animación
  - ▶ Método `stop ()`

# Biblioteca Tween.js

# Three.js

- Cada método devuelve el propio objeto
  - ▶ Se pueden encadenar los mensajes, definiendo la animación de una manera muy compacta

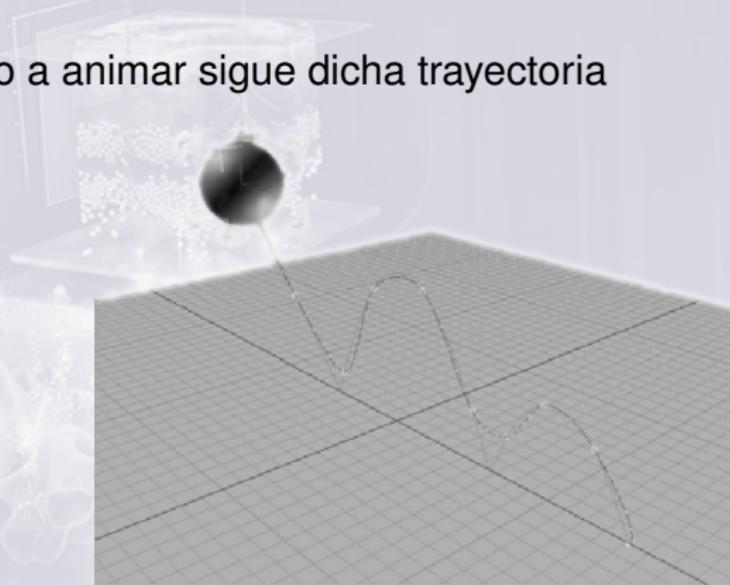
## Ejemplo: Definición de una animación

```
var origen = { p : 0 };
var destino = { p : 100 };

var movimiento = new TWEEN.Tween( origen )
    .to( destino, 1000 )
    .easing( TWEEN.Easing.Linear.None )
    .onStart( function () { mesh.position.y += 10; })
    .onUpdate( function () { mesh.position.x = origen.x })
    .onComplete( function () { mesh.position.y -= 10; })
    .repeat( Infinity )
    .yoyo( true )
    .start();
```

# Animación mediante caminos

- Se define una trayectoria
- El objeto a animar sigue dicha trayectoria



# Animación mediante caminos

## Procedimiento

- Se define el camino mediante un Spline



- La posición y orientación del objeto a animar se toman del spline



La cámara siguel el camino marcado por el spline

# Animación mediante caminos

Three.js

## Definición de Splines

- Se definen indicando sus puntos de paso
- Pueden crearse abiertos o cerrados

### Ejemplo: Definición de Splines

```
// Spline abierto
var sp1 = new THREE.SplineCurve3([
    new THREE.Vector3(0, 0, 0), new THREE.Vector3(0, 1, 0), ... ]);

// Spline cerrado
var sp2 = new THREE.ClosedSplineCurve3([
    new THREE.Vector3(0, 0, 0), new THREE.Vector3(0, 1, 0), ...
    ... , new THREE.Vector3(0, 0, 0)]);
```

# Animación mediante caminos

## Uso del Spline en la animación

Three.js

- Se puede obtener una posición y una dirección tangente

### Ejemplo: Uso de Splines para modificar parámetros

```
// Se necesita un parámetro entre 0 y 1
//     Representa la posición en el spline
//     0 es el principio
//     1 es el final
var time = Date.now();
var looptime = 20000;    // 20 segundos
var t = ( time % looptime ) / looptime;

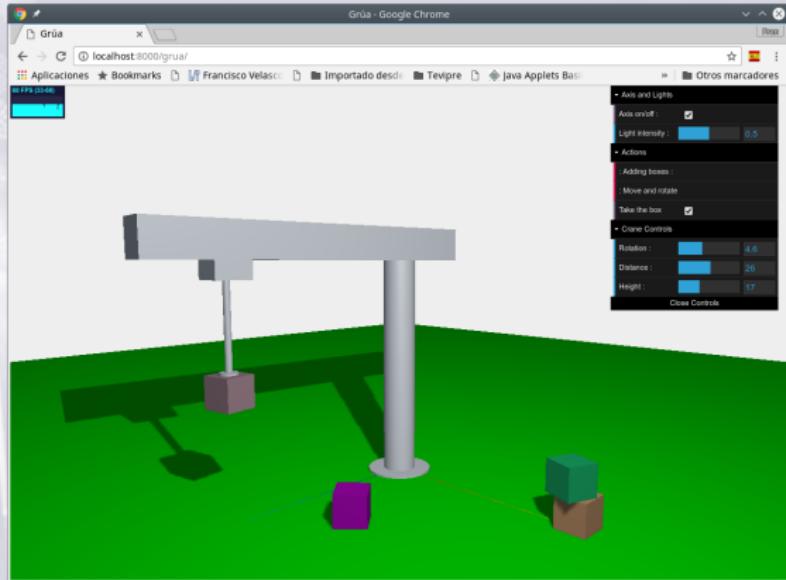
// Se coloca y orienta el objeto a animar
object.position.copy (spline.getPointAt (t));
object.rotation.copy (spline.getTangentAt (t));
```

# Animación combinando técnicas

- El método onUpdate de TWEEN es como un método update personalizado pero
  - ▶ Es llamado solo cuando esa animación está activa
- Se puede usar Tween para interpolar un parámetro entre 0 y 1 con una determinada curva de velocidades
  - ▶ Usar dicho parámetro para obtener la posición y orientación de una trayectoria (spline)
  - ▶ Y posicionar un objeto en dicho camino
- Se puede usar TWEEN para interpolar un parámetro entre 0 y 1
  - ▶ Nos indica que porcentaje ha transcurrido de un movimiento mayor
  - ▶ E iniciar y detener animaciones en función de eso

# Interacción

- Con la aplicación
  - ▶ Mediante una GUI (`dat.gui.js`)
- Con la escena
  - ▶ Selección de objetos  
(Picking)
  - ▶ Edición
  - ▶ Movimientos de cámara



# Interfaz Gráfica de Usuario

- Más información en <https://github.com/dataarts/dat.gui>
- Se define una función-objeto con un atributo por cada parámetro a controlar, el tipo de dato determina el tipo del control en la GUI
  - ▶ Booleano, para un control tipo *checkbox*
  - ▶ Número, para un control tipo *slider*
  - ▶ Función, para un control de tipo *button*
    - ★ La función es el código que se ejecutará al pulsar el botón
- Se conforma la interfaz, agrupando opciones, poniendo etiquetas, límites, etc.
- Cuando es necesario, se consultan los atributos de dicha función-objeto

dat.gui

**Ejemplo****GUI: Ejemplo de definición**

```

GUIControls = new function() {
    this.axis = true;
    this.lightIntensity = 0.5;
    this.addBox = function () { . . . },
}

var gui = new dat.GUI();

var axisLights = gui.addFolder ('Axis and Lights');
axisLights.add(GUIControls, 'axis').name('Axis on/off :');
axisLights.add(GUIControls, 'lightIntensity', 0, 1.0)
    .name('Light intensity :');

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

```

**GUI: Ejemplo de uso**

```
spotLight.intensity = GUIControls.lightIntensity;
```

# Interacción con el ratón en la escena

- Se definen **funciones** asociadas a determinados **eventos** del ratón
- Se definen **estados** que indican **qué se está haciendo** con la aplicación en cada momento
- Cada función que procese un evento del ratón
  - ▶ Debe consultar el estado actual de la aplicación para ...
  - ▶ Realizar el procesamiento correcto

Hacer un clic y arrastrar el ratón puede ser:

- ★ Realizar un movimiento de cámara
- ★ Añadir un objeto a la escena en una posición
- ★ Seleccionar y mover un objeto existente
- ★ *Cualquier otra cosa ...*

# Eventos del ratón que pueden escucharse

- Entre otros, se pueden escuchar los siguientes eventos
  - ▶ mousedown      mouseup      mousemove      wheel
- En el *main* se añaden los *listener* y se indican las funciones que se ejecutarán cuando se produzca cada evento

## Listener: Ejemplo

```
window.addEventListener ("mousemove" , onMouseMove);  
                              evento                     función
```

- Valores asociados al evento que se pueden consultar
  - ▶ clientX: La coordenada X del ratón (relativa a la ventana)
  - ▶ clientY: La coordenada Y
  - ▶ which: El botón concreto que se ha pulsado
    - ★ 0 (ninguno), 1 (izquierdo), 2 (central), 3 (derecho)

# Ratón junto a una tecla modificadora

- Los eventos del ratón pueden realizar distintas acciones si se producen estando pulsada una o varias teclas modificadoras
- En la función que procesa un evento del ratón se puede consultar el estado de dichas teclas para realizar un procesamiento u otro
- Teclas modificadoras que pueden consultarse
  - ctrlKey
  - altKey
  - shiftKey

**Ejemplo:** Movimiento de cámara solo con `Ctrl` pulsado

```
function onMouseDown (event) {  
    if (event.ctrlKey) {  
        scene.getCameraControls().enabled = true;  
    } else {  
        scene.getCameraControls().enabled = false;  
        // Se desabilitan los movimientos de cámara  
        // Se realiza otro procesamiento  
    }  
}
```

# Estados de la aplicación

- Una variable global que indica qué se está haciendo
- Se puede establecer al elegir una opción del menú
- Se consulta desde las funciones que procesan los eventos del ratón para determinar el procesamiento a realizar

## Ejemplo: Definición y usos de estados de aplicación

```
// Se definen (normalmente) como constantes numéricas
TheScene.NO_ACTION = 0;
TheScene.ADDING_BOXES = 1;
TheScene.MOVING_BOXES = 2;

// Se usan para darle valor a la variable de estado de la aplicación
applicationMode = TheScene.NO_ACTION;

// Se consulta al procesar un evento del ratón
function onMouseDown (event) {
    switch (applicationMode) {
        case TheScene.ADDING_BOXES :
            // procesamiento para mouseDown y ADDING_BOXES
```

# Lectura del teclado

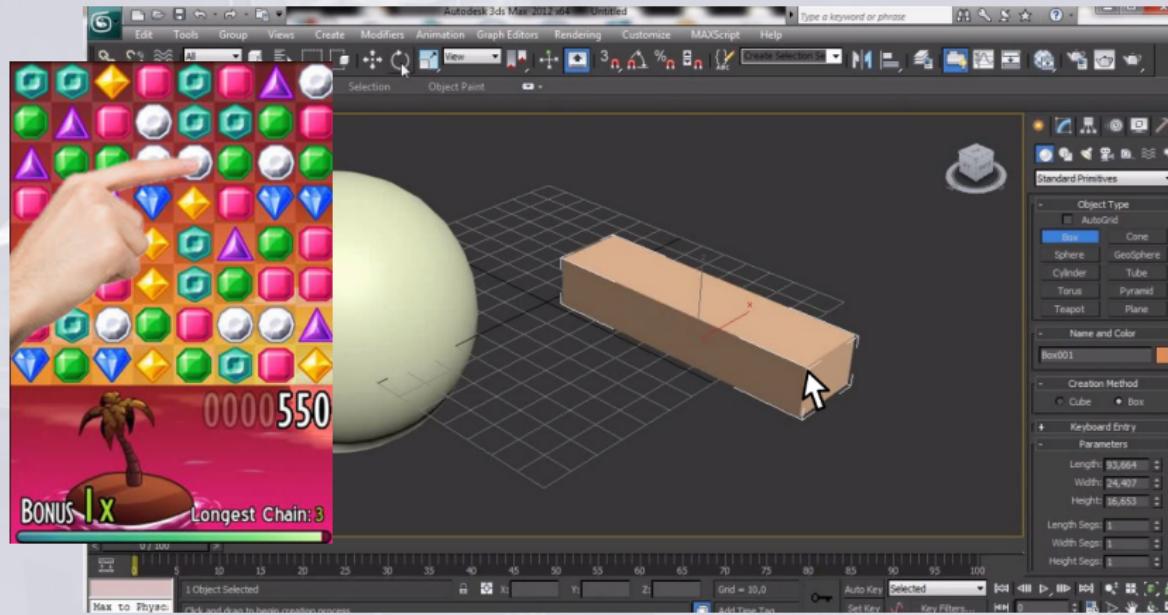
- Se definen **funciones** asociadas a **eventos del teclado**
  - ▶ keydown: Se pulsa una tecla
  - ▶ keyup: Se suelta
  - ▶ keypress: Pulsación y suelta.  
Este evento no lo generan las teclas modificadoras.
- En dichas funciones se lee la tecla que produjo el evento
  - ▶ var x = event.which || event.keyCode
    - ★ Así, la lectura del código asociado a dicha tecla funciona en todos los navegadores
  - ▶ La lista completa de códigos se puede consultar en  
[https://www.w3schools.com/charsets/ref\\_html\\_utf8.asp](https://www.w3schools.com/charsets/ref_html_utf8.asp)
  - ▶ Para saber, de manera más cómoda,  
si se ha pulsado un carácter imprimible se puede usar  
`if (String.fromCharCode (x) == "A")`

# Selección de objetos

## • Picking

Seleccionar un elemento de la escena con un puntero

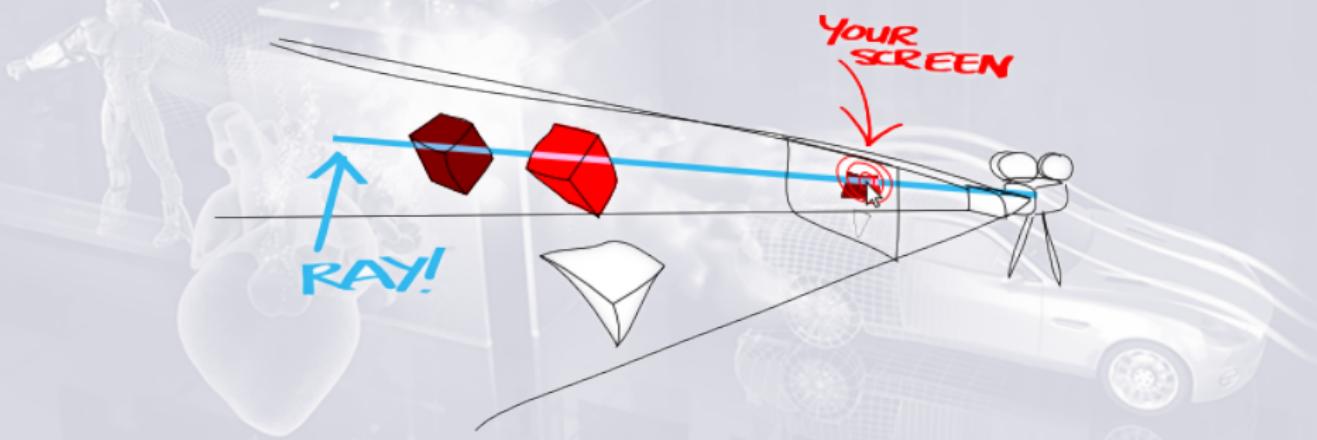
- ▶ Suele ser una operación habitual en muchas aplicaciones gráficas



# Selección de objetos (Picking)

## Proceso a realizar

- ① Saber en qué píxel se ha hecho clic
- ② Lanzar un rayo
  - ▶ Desde la cámara
  - ▶ Que pase por dicho píxel
- ③ Obtener los objetos alcanzados por ese rayo



# Picking

# Three.js

## Ejemplo: Picking

```
function onDocumentMouseDown (event) {
    // Posición del clic en coordenadas de dispositivo normalizado
    var mouse = new THREE.Vector2 ();
    mouse.x = (event.clientX / window.innerWidth) * 2 - 1;
    mouse.y = 1 - 2 * (event.clientY / window.innerHeight);
    // Se construye el rayo
    var raycaster = new THREE.Raycaster ();
    raycaster.setFromCamera (mouse, camera);
    // pickableObjects es un Array de objetos seleccionables
    // Lista de objetos alcanzados, entre los seleccionables
    var pickedObjects = raycaster.intersectObjects
        (pickableObjets, true);
    // pickedObjects es un vector ordenado desde el objeto más cercano
    // Se cambia el aspecto del más cercano
    if (pickedObjects.length > 0) {
        selectedObject = pickedObjects[0].object;
        selectedPoint = new THREE.Vector3 (pickedObjects[0].point);
        ...
    }
}
```

# Selección del objeto global

Uso del atributo `userData`

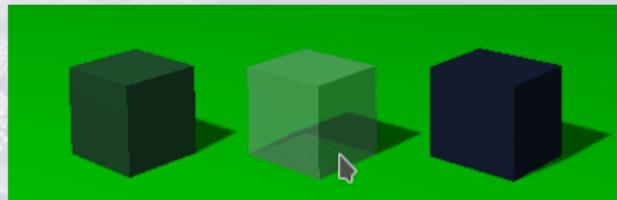
- Pick devuelve el Mesh 'clicado'
- Se puede desear acceder a la raíz del árbol de la figura
- Se usa el atributo `setData` de Mesh
  - ▶ En cada Mesh se hace que `userData` apunte a la raíz
  - ▶ Tras el Pick, se accede a la raíz mediante `userData`

En vez de `setData -> userData`

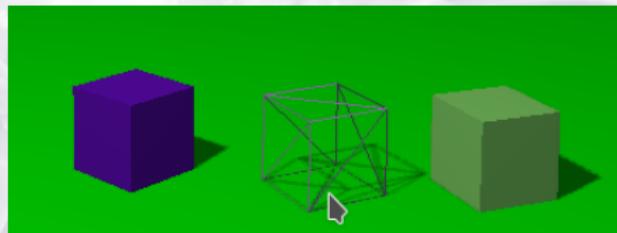


# Feedback

- El objeto concreto seleccionado debe indicarse al usuario
- Se realiza con un cambio en su aspecto
  - ▶ Transparencias, cambio de color, modo alambre, etc.



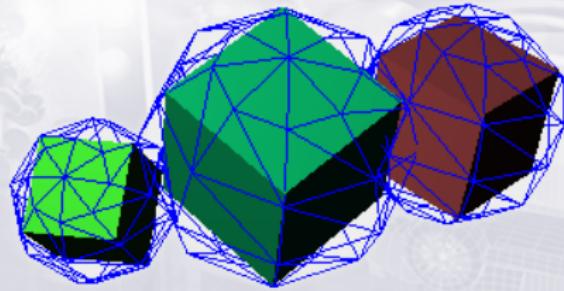
Atributo del material `opacity = 0.5` y `transparent = true`



Atributo del material `wireframe = true`

# Detección de colisiones

- Normalmente es necesario saber cuándo 2 objetos colisionan
- La detección de colisiones se realiza en dos fases
  - ▶ Fase gruesa:  
Se descartan rápidamente los elementos que no colisionan
  - ▶ Fase fina:  
Se determina con exactitud si 2 elementos están colisionando
- En la fase gruesa se usa:
  - ▶ Indexación espacial
  - ▶ Cajas o esferas englobantes



# Indexación espacial de la escena

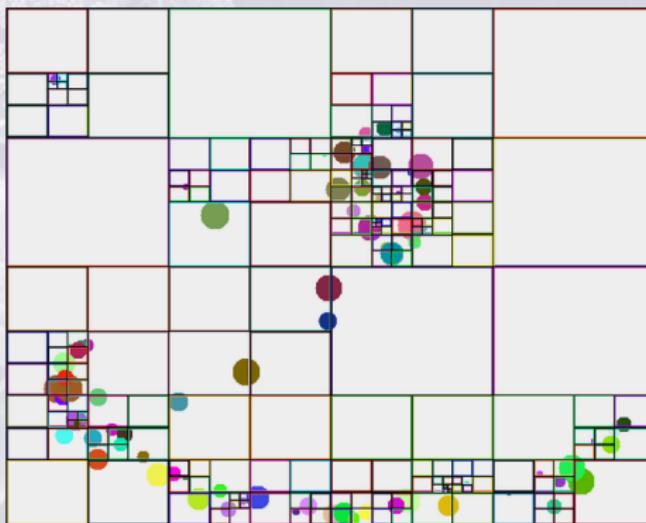
- Cuando se tienen muchos objetos en la escena es importante saber indentificarlos con rapidez
  - ▶ Cuando se lanzan rayos para Ray Tracing o Picking
  - ▶ Cuando se buscan colisiones entre objetos
- Para ello se usan estructuras de descomposición espacial



# Estructuras para indexación espacial

## Octrees

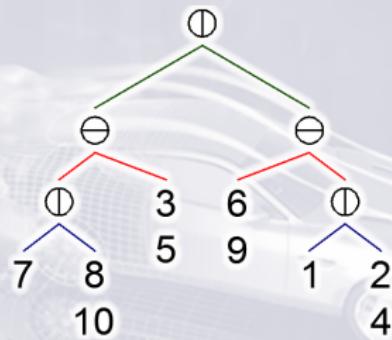
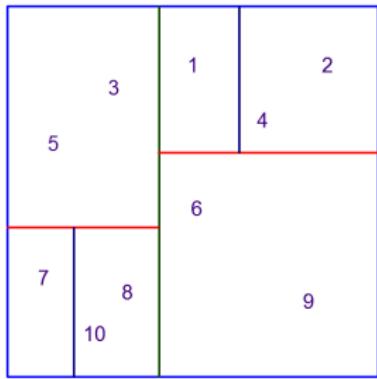
- El espacio se subdivide jerárquicamente en octantes
- Un octante solo se subdivide si contiene más objetos que el límite establecido



# Estructuras para indexación espacial

## KD-Trees

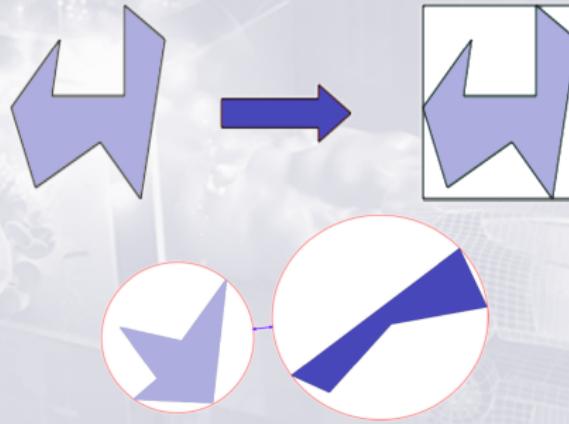
- El espacio se subdivide en 2 semiespacios por un plano
  - ▶ Solo si tiene más elementos que un límite
  - ▶ El plano está alineado con los ejes
  - ▶ Situado de manera que quede un árbol balanceado
  - ▶ En cada nivel se cambia el eje de división, alternativamente  $X \rightarrow Y \rightarrow Z \rightarrow X \dots$



# Cajas y esferas englobantes

- Formas sencillas que engloban completamente al objeto
- Permiten saber rápidamente cuando 2 objetos no colisionan
- Según la precisión exigida, se usan para determinar la colisión

**Ejercicio:** Diseñar sendos algoritmos para calcular la caja y la esfera englobante de un objeto cualquiera a partir de su lista de vértices



# Física

- Un motor de física permite:
  - ▶ Dotar de gravedad a la escena
  - ▶ Masa a los objetos
  - ▶ Atributos a los materiales como:
    - ★ Rozamiento
    - ★ Efecto rebote
  - ▶ Detectar y procesar colisiones entre objetos
- Requiere bastante cálculo
  - ▶ Se suele separar la Física y el Rendering en hebras distintas
- Usaremos la biblioteca **Physijs**
  - ▶ Descargable de [github.com/chandlerprall/Physijs](https://github.com/chandlerprall/Physijs)
  - ▶ Requiere usar la versión de Three.js que viene con Physijs

# Una escena básica con Physijs

## Three.js

### Physijs: Una escena básica

```
// Se establece qué web worker gestiona las hebras
Physijs.scripts.worker = '../libs/physijs_worker.js';

// Se establece el motor de física
// Physijs solo es un wrapper que hace más fácil el uso de ammo
Physijs.scripts.ammo = '../libs/ammo.js';

// Se crea una escena física y se indica el valor de la gravedad
var scene = new Physijs.Scene();
scene.setGravity (new THREE.Vector3 (0, -10, 0));

// Se crea una geometría 'normal' y a partir de ésta
// se crea un Mesh físico y se añade a la escena
var stoneGeom = new THREE.BoxGeometry (1, 3, 2);
var stone = new Physijs.BoxMesh (stoneGeom,
    new THREE.MeshPhongMaterial ({color: 0xff0000}));
scene.add (stone);

// En la función de render hay que añadir una línea
// para realizar la simulación física
scene.simulate();
```

# Propiedades de los objetos

## • Rozamiento y rebote

- ▶ Con valores entre 0.0 y 1.0
- ▶ Se indican al definir un material físico

```
var mat = Physijs.createMaterial (  
    new THREE.MeshPhongMaterial ({color: 0xff0000}),  
    0.9, // rozamiento  
    0.3); // rebote
```

## • Masa

- ▶ Se indica al crear el Mesh físico

```
var suelo = new Physijs.BoxMesh (  
    new THREE.BoxGeometry (60,1,60), mat,  
    0); // masa
```

- ▶ El valor 0 hace que no le afecte la gravedad, necesario en aquellos objetos como el suelo, paredes que no se caen, etc.

# Objetos compuestos

- Para que un objeto compuesto por varios elementos sea tratado como un objeto único por Physijs
  - Debe haber una relación jerárquica entre los elementos

## Physijs: Objetos compuestos tratados como un todo

```
var suelo = new Physijs.BoxMesh (new THREE.BoxGeometry (60,1,60) , materialSuelo , 0);  
  
var paredIzq = new Physijs.BoxMesh (new THREE.BoxGeometry (2,6,60) , materialParedes , 0);  
paredIzq.position.x = -30;  
paredIzq.position.y = 2.5;  
  
suelo.add (paredIzq);  
  
scene.add (suelo);
```

# Modificaciones manuales de posición y orientación

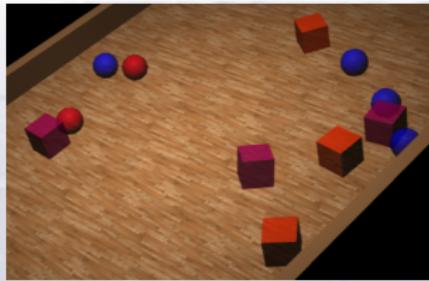
- La posición y orientación de un objeto viene determinada por:
  - ▶ El efecto de la gravedad
  - ▶ La interacción con otros objetos
    - ★ Colisiones, rozamientos, rebotes, etc.
- Si se desea modificar la posición y/u orientación manualmente
  - ▶ Se modifican los atributos `position` y/o `rotation`
  - ▶ Se le indica al motor de física para que lo tenga en cuenta
    - ★ Atributo `__dirtyPosition = true`
    - ★ Atributo `__dirtyRotation = true`

# Colisiones

- El motor detecta y procesa las colisiones
  - ▶ Rebotes, cambios de dirección, etc.
- Se puede programar una función
  - ▶ Emitir un sonido, quitar una vida, etc.

## Physijs: Listener de colisiones

```
elMesh.addEventListener ('collision',  
    function (elOtroObjeto , velocidad , rotacion , normal) {  
        // el procesamiento a realizar  
    });
```

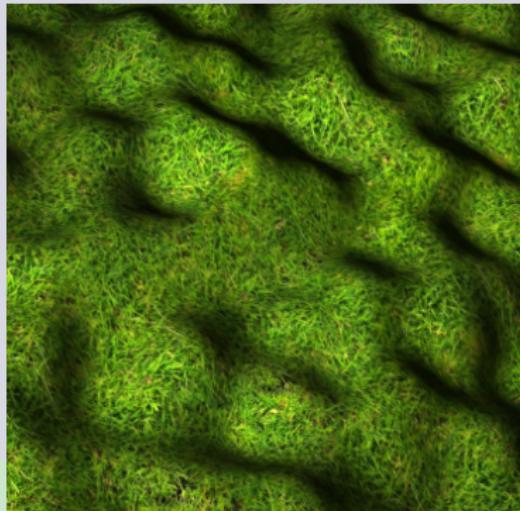


# Formas disponibles

- Se debe usar el la forma Physijs que mejor se adapte a la geometría creada
- Las formas más usuales de Physijs son:
  - ▶ Physijs.BoxMesh
  - ▶ Physijs.SphereMesh
  - ▶ Physijs.CylinderMesh
  - ▶ Physijs.ConeMesh
  - ▶ Physijs.ConvexMesh : Para aquellas geometrías que no encajen bien en las otras formas

# Formas especiales

- Physijs.PlaneMesh
  - ▶ Se crea con masa 0
  - ▶ No le afecta la gravedad
  - ▶ No es movido por otros objetos
  
- Physijs.HeightfieldMesh
  - ▶ Crea un plano con alturas



## Physijs: Plano con alturas

```
var sueloGeometria = new THREE.PlaneGeometry (60, 50, 100, 100);
for (var i = 0; i < sueloGeometria.vertices.length; i++) {
    sueloGeometria.vertices[i].z = // se le da la altura deseada
}
sueloGeometria.computeFaceNormals();           // Necesario
sueloGeometria.computeVertexNormals();         // al cambiar las Z
var suelo = new Physijs.HeightfieldMesh (sueloGeometria,
    sueloMaterial, 0, // masa
    100, 100);
```

# Restricciones a los movimientos

- Se pueden añadir restricciones a los movimientos que tiene un objeto por la gravedad y las colisiones
- PointConstraint
  - ▶ Se fija la posición de un objeto respecto a otro
  - ▶ Si uno se mueve, el otro también lo hará
  - ▶ Manteniendo la distancia y la orientación

## Physijs: Restricción objeto a objeto

```
var restric = new Physijs.PointConstraint (
    obj1, obj2, obj2.position);

// Las restricciones deben añadirse a la escena
scene.addConstraint (restric);
```

# Restricción tipo bisagra

- HingeConstraint

## Physijs: Restricción tipo bisagra

```
var restric = new Physijs.HingeConstraint (
    objMovil, objFijo, objFijo.position,
    new THREE.Vector3 (0,1,0)); // el eje de la bisagra
scene.addConstraint (restric);

// Límites al movimiento, principalmente ángulo mínimo y máximo
retric.setLimits (-2.2, -0.6, 0.1, 0);

// Para moverlo intencionadamente
restric.enableAngularMotor (velocidad, aceleracion);

// Para desactivar el motor
// (solo se movería por gravedad o colisiones)
restric.disableMotor();
```

# Restricción de deslizamiento

- SliderConstraint

## Physijs: Restricción tipo deslizamiento

```
var restric = new Physijs.SliderConstraint (
    objMovil, objFijo, objFijo.position,
    new THREE.Vector3 (1,0,0)); // el eje del deslizamiento
scene.addConstraint (restric);

// Límites al movimiento, distancias mínimas y máximas
retric.setLimits (-10, 10);

// Para moverlo intencionadamente
restric.enableLinearMotor (velocidad, aceleracion);

// Para desactivar el motor
// (solo se movería por gravedad o colisiones)
restric.disableMotor();
```

# Restricción de péndulo

- ConeTwistConstraint

## Physijs: Restricción de péndulo

```
var restric = new Physijs.ConeTwistConstraint (
    objMovil, objFijo, objFijo.position);
scene.addConstraint (restric);

// Límites al movimiento, 3 ángulos para los 3 ejes
retric.setLimits (0.5*Math.PI, 0.5*Math.PI, 0.5*Math.PI);

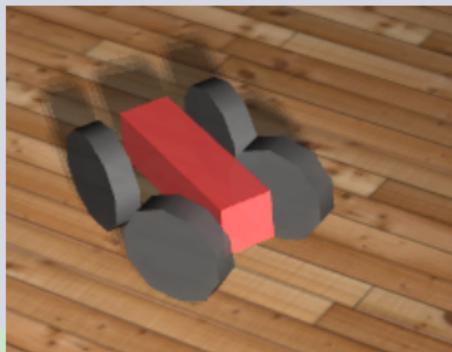
// Para moverlo intencionadamente unos determinados ángulos por eje
restric.enableMotor ();
restric.setMotorTarget (new THREE.Vector3 (1, -0.5, 1.5));

// Para desactivar el motor
// (solo se movería por gravedad o colisiones)
restric.disableMotor();
```

# Restricción de grado de libertad

- **DOFConstraint**

- ▶ Permite controlar de manera exacta los movimientos angulares y lineales de un objeto



## Physijs: Ejemplo: Un coche

```
// Se hace y se posiciona la carrocería y las ruedas
var carroceria = new Physijs.BoxMesh ( . . . );
.
.
.
scene.add ( carroceria );
var ruedaMotrizIzq = new Physijs.CylinderMesh ( . . . );
.
.
.
scene.add ( ruedaMotrizIzq );

// se restringen las ruedas a la carrocería y se añaden a la escena
var rml = new Physijs.DOFConstraint ( ruedaMotrizIzq , carroceria , new
    THREE.Vector3 ( el punto de restricción));
scene.addConstraint ( rml );
```

# Restricción de grado de libertad

## Continuación

### Phyjs: Ejemplo: Un coche (continuación)

```
// Se configuran las ruedas motrices (similar para la derecha)
// Se bloquea todo, es sobreescrito con el motor
rml.setAngularLowerLimit ({x:0, y:0, z:0});
rml.setAngularUpperLimit ({x:0, y:0, z:0});

// Se configuran las ruedas directrices (similar para la derecha)
// Giran en Y según el ángulo indicado
// Pueden girar en Z libremente,
// el límite inferior es mayor que el superior
rdl.setAngularLowerLimit ({x:0, y:anguloGiro, z:0.1});
rdl.setAngularUpperLimit ({x:0, y:anguloGiro, z:0});

// Se hacen girar las ruedas motrices (similar para la derecha)
// El 2 indica que se está configurando el eje Z
// Límite inferior y superior (giros completos sin límites)
rml.configureAngularMotor (2, 0.1, 0, velocidad, fuerza);
```

# Grafos de Escena

Francisco Velasco Anguita

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

Sistemas Gráficos

Grado en Ingeniería Informática  
Curso 2017-2018