

# **Creación de geometría**

Francisco Velasco Anguita

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

Sistemas Gráficos

Grado en Ingeniería Informática  
Curso 2019-2020

# Contenidos

## 1 Objetivos

## 2 Introducción

## 3 Modelado de sólidos

- Definición de sólido
- Esquemas de representación
- Representación B-Rep

## 4 Modelos jerárquicos

- Repaso a las transformaciones geométricas
- Transformaciones aplicadas a una Geometry
- Transformaciones aplicadas a un Mesh
- Transformaciones aplicadas a un Object3D

# Objetivos

- Diferenciar modelo geométrico y volumétrico
- Saber representar sólidos
- Saber modelar un sólido combinando distintos métodos
- Saber diseñar e implementar modelos jerárquicos  
(de objetos articulados)

# Introducción

## Modelos geométricos vs. volumétricos

### • **Modelo**

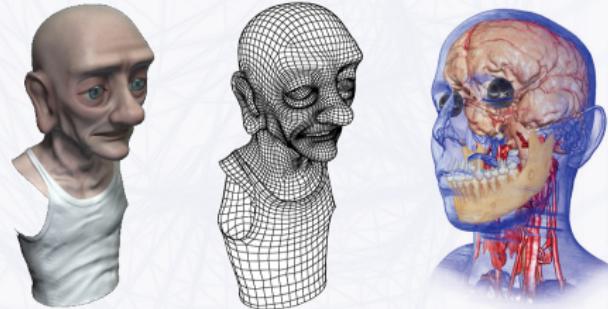
- ▶ Representación de lo más relevante de una entidad

### • **Modelos geométricos**

- ▶ Representan la geometría de un objeto, la *forma* de su frontera
- ▶ Si la frontera encierra un espacio, éste no es relevante

### • **Modelos volumétricos**

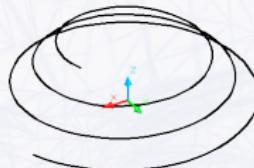
- ▶ Representan propiedades espacialmente localizadas
- ▶ El interior es heterogéneo en cuanto a su material o propiedades



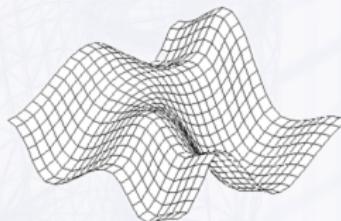
# Modelos Geométricos

- Consideraremos elementos con coordenadas 3D
- Aún así, se puede hablar de elementos:

- Unidimensionales



- Bidimensionales



- Tridimensionales

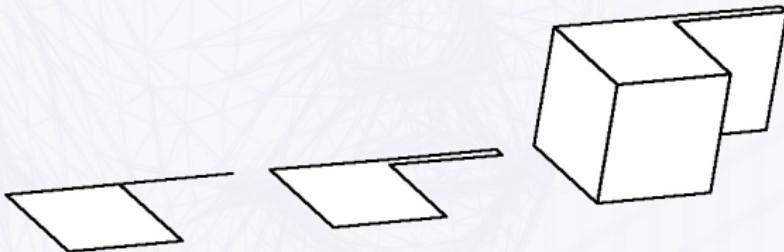


# Elementos 3D tridimensionales: Sólidos

- **Definición de Sólido:** Desde dos puntos de vista.
- Desde un punto de vista **Topológico**

Una *porción* del espacio.  $S \subset \mathbb{R}^3$

- ▶ Acotado
- ▶ Cerrado
- ▶ Rígido
- ▶ Regular

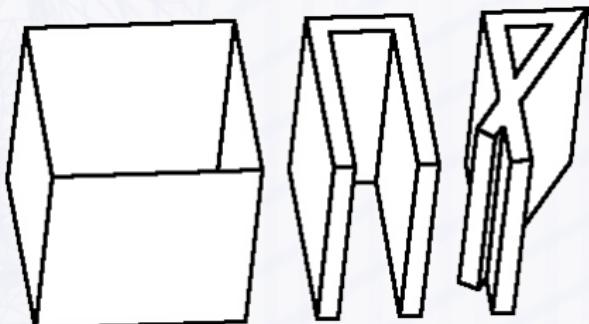
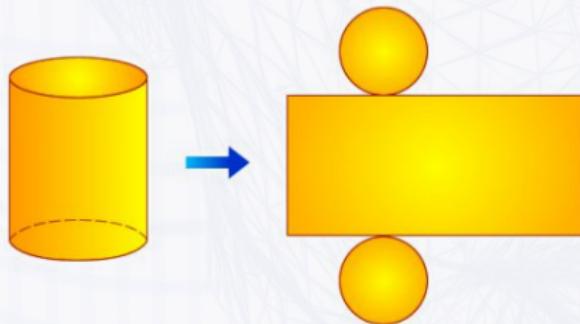


# Sólidos

- Desde un punto de vista **Algebraico**

El sólido se describe mediante su superficie frontera, que es:

- ▶ Cerrada
- ▶ Orientada
- ▶ Completa
- ▶ Que no se corta a sí misma



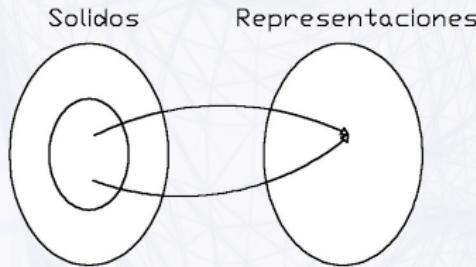
# Modelado de Sólidos

- **Representación de un Sólido:**

- ▶ Una colección finita de símbolos (de un alfabeto finito) que describen un sólido.
- ▶ Ejemplo:  $(x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2 \leq r^2$

- **Esquema de Representación:**

- ▶ La relación entre los sólidos y las descripciones de estos.



- Propiedades deseables: **dominio**, **no-ambigüedad**, poca ocupación de memoria, facilidad de creación y edición, etc.

# Modelado de Sólidos

## Esquemas de representación

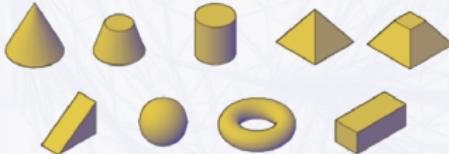
- **Barrido**

- ▶ El sólido se representa mediante una superficie plana y una trayectoria



- **Instanciación de primitivas**

- ▶ Se tiene un conjunto de sólidos básicos
- ▶ Un sólido se representa mediante la instanciación concreta de uno de esos sólidos básicos



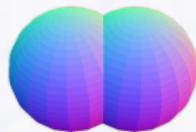
# Modelado de Sólidos

## Esquemas de representación

- **CSG: Constructive Solid Geometry** (Geometría de Sólidos Constructiva)

- ▶ Se tiene un conjunto de primitivas básicas (normalmente formado por semiplanos)
- ▶ Se tienen operaciones booleanas entre sólidos (se operan 2 sólidos para obtener 1 sólido resultado)
  - ★ Unión  
Se toma TODA la materia de los operandos
  - ★ Intersección  
Se toma SOLO la materia COMÚN de los operandos
  - ★ Diferencia (no es conmutativa)  
Se toma la materia del primer operando MENOS la que ocupa el segundo operando

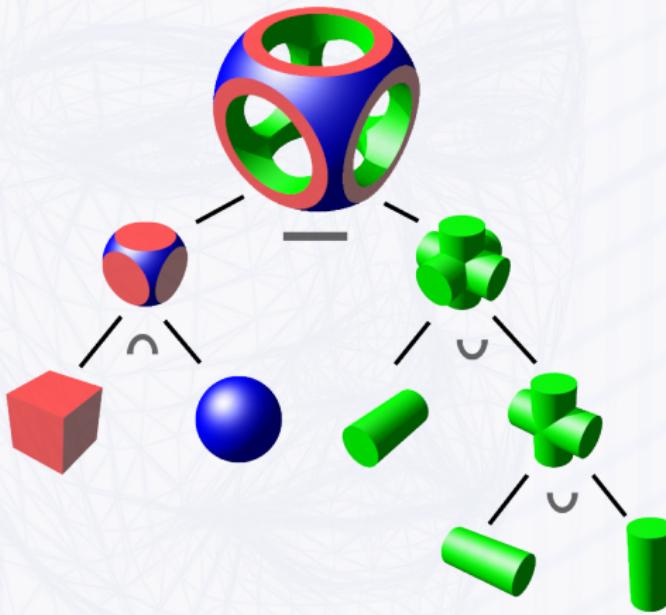
- Un sólido se representa mediante el árbol de operaciones que lo definen



# Modelado de Sólidos

## Sólidos CSG

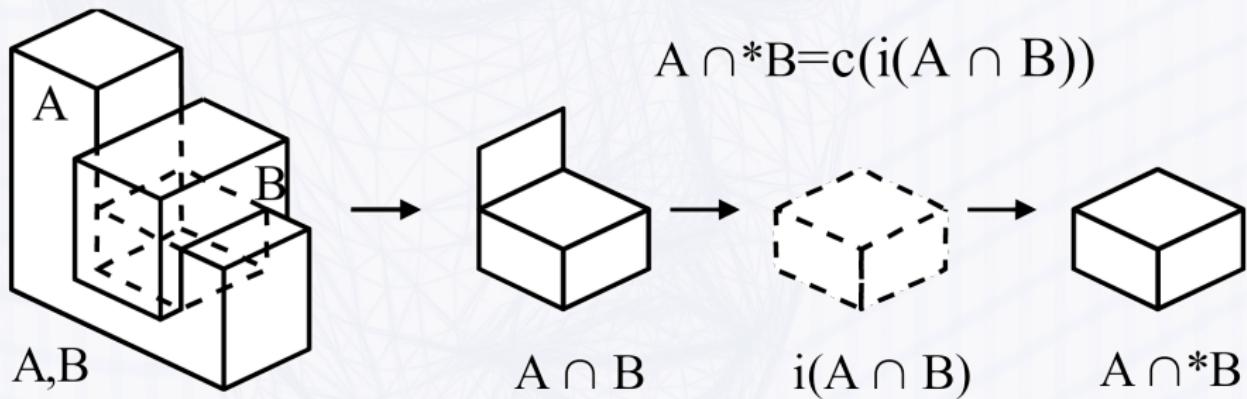
- Un sólido CSG se representa mediante el árbol de operaciones que lo definen



# Modelado de Sólidos

## Sólidos CSG

- Las operaciones booleanas son regularizadas para que el resultado siga siendo un sólido válido.
- La regularización se define como la clausura del interior de un cuerpo
- Se denotan con un arterisco



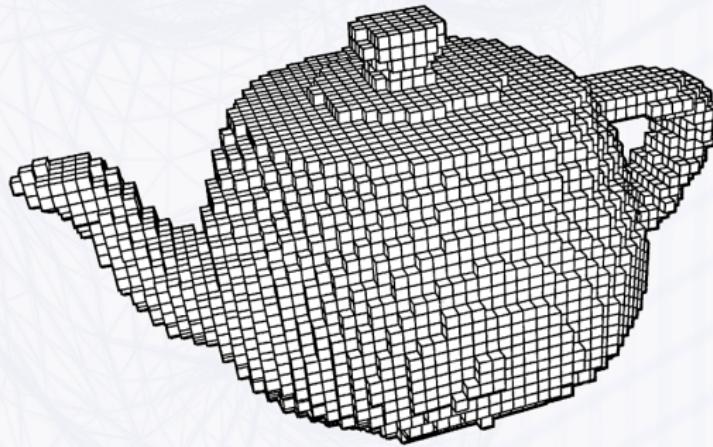
(Imagen realizada por Juan Carlos Torres)

# Modelado de Sólidos

## Esquemas de representación

### ● Descomposición

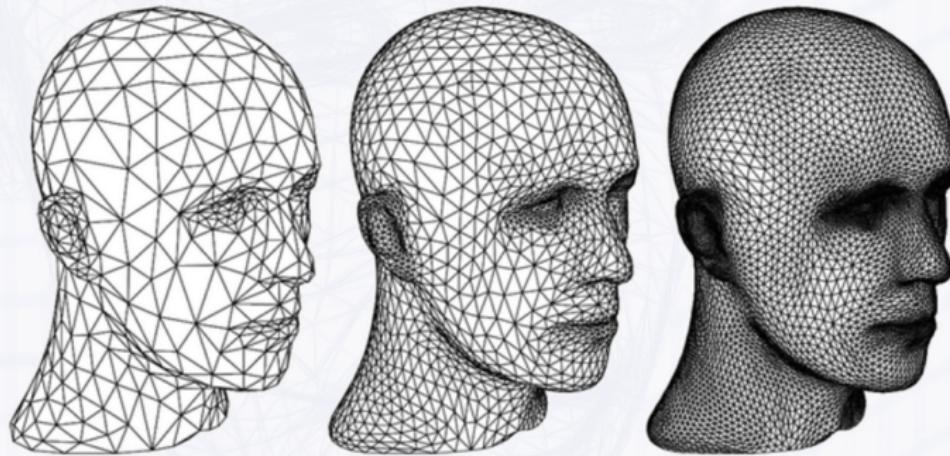
- ▶ El espacio se descompone en elementos disjuntos, normalmente cubos
- ▶ Es sólido se representa enumerando cuáles de esos elementos están ocupados



# Modelado de Sólidos

## Esquemas de representación

- **B-Rep: Boundary Representation** (Representación de Fronteras)
  - ▶ Un sólido es representado describiendo la frontera que lo delimita
  - ▶ La frontera es representada mediante un malla de polígonos, normalmente triángulos



# Modelado de Sólidos

## Comparando esquemas

- No hay un esquema que sea bueno para todo
  - ▶ Barrido e Instanciación de primitivas
    - ✗ Tienen un dominio reducido
    - ✓ En cambio, son cómodos para crear sólidos
  - ▶ CSG
    - ✓ Tiene un dominio más amplio, dependiendo del conjunto de primitivas
    - ✓ Permite crear sólidos complejos de una manera intuitiva
    - ✗ Sin embargo, para visualización:
      - Requiere ser convertido a B-Rep o ...
      - Usar Ray Casting y evaluar el árbol en cada rayo

# Modelado de Sólidos

## Comparando esquemas

### ► Descomposición

- ✗ El dominio de objetos representables de manera exacta es mínimo
- ✓ De manera aproximada, puede representar cualquier sólido
- ✓ Las operaciones booleanas se implementan muy fácilmente
- ✓ El cálculo de propiedades volumétricas es muy sencillo

### ► B-Rep

- ✗ El dominio de representaciones exactas se reduce a los poliedros
- ✓ De manera aproximada, puede representar cualquier sólido
- ✓ La visualización es sencilla, simulando representaciones exactas de cualquier sólido

Sin embargo, son operaciones complejas:

- ✗ La creación
- ✗ Las operaciones booleanas
- ✗ El cálculo de propiedades volumétricas

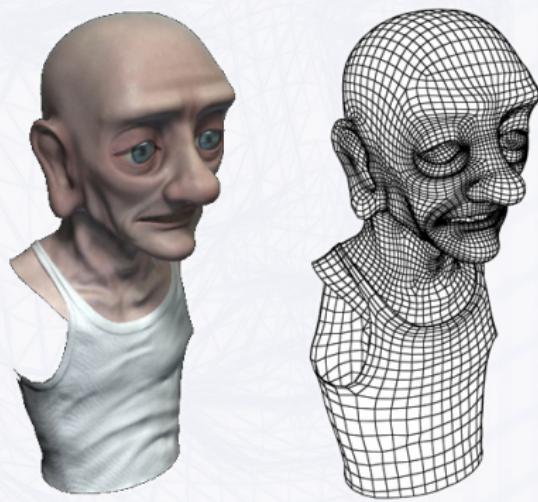
# Modelado de Sólidos

## Esquemas híbridos

- Un modelador de sólidos suele usar varios esquemas
  - ▶ Uno primario y otros secundarios
  - ▶ Se beneficia de las ventajas de ellos
- Por ejemplo:
  - ▶ Primario: B-Rep, usado para
    - ★ Visualización
    - ★ Edición de la malla
  - ▶ Secundarios de entrada:
    - ★ Barrido
    - ★ Instanciación de primitivas
    - ★ CSG
  - ▶ Secundario de consulta:
    - ★ Descomposición, para cálculo de propiedades volumétricas

# Representación B-Rep

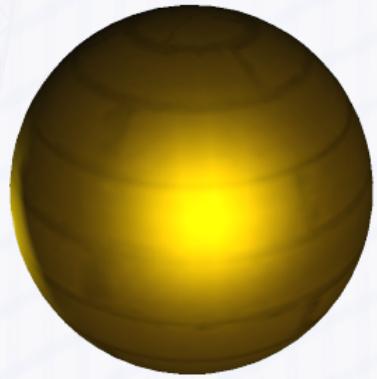
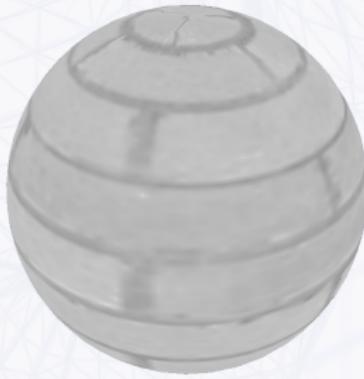
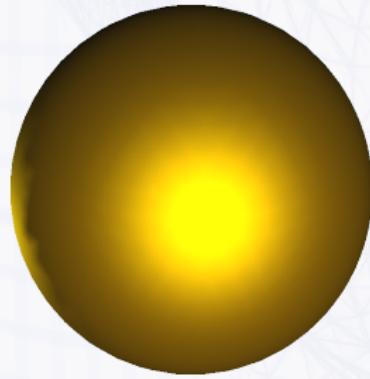
- Características de la representación
  - ▶ Es exacta para figuras poliédricas
  - ▶ Es aproximada en las fronteras curvas
    - ★ Mediante técnicas de shading se suple *visualmente* esa desventaja
  - ▶ Es robusta e independiente de su complejidad geométrica



# 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

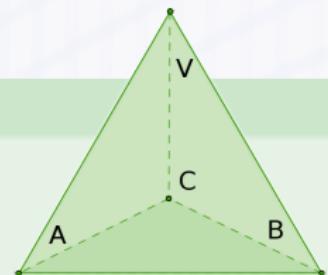
```
// Cada vértice se define una sola vez
```

```
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

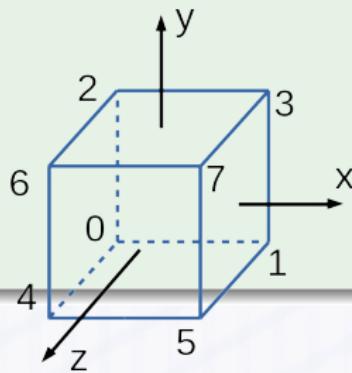
- Manualmente
  - ▶ Se define en el código toda la información
- Proceduralmente
  - ▶ A partir de unos parámetros, un método genera la información
- Operando con primitivas básicas
  - ▶ Se dispone de una geometría básica ya generada
  - ▶ Se construye una geometría compleja operando con la que se tiene 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};

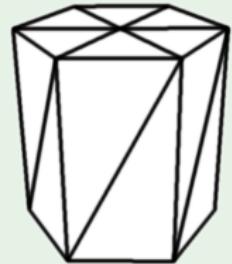
// Indices para formar las caras
int[] indices = {
    0, 2, 3, 1,      // Cara trasera
    4, 5, 7, 6,      // Frontal
    1, 3, 7, 5,      // Derecha
    0, 4, 6, 2,      // Izquierda
    0, 1, 5, 4,      // Inferior
    2, 6, 7, 3 };   // Superior
```



# 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);
```

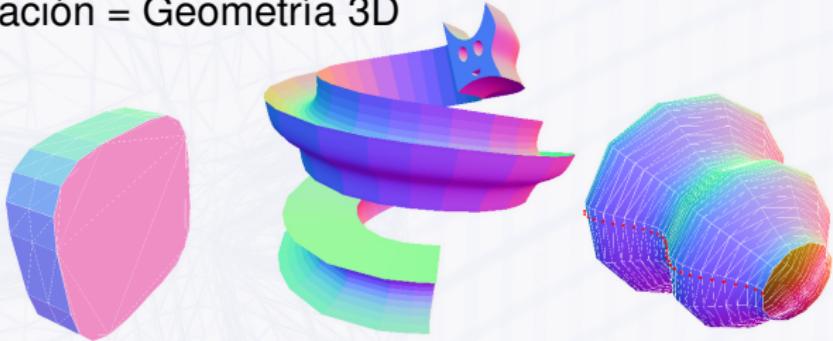
- En el array de vértices, cada 3 valores es un vértice
- En el array de caras, cada 3 valores es un triángulo

# Creación de Geometría

## Operando con geometrías

- Geometría 2D + operación = Geometría 3D

- ▶ Extrusión
- ▶ Barrido
- ▶ Revolución



- Geometrías 3D + operación = Geometría 3D

- ▶ Operaciones booleanas

- ★ Unión
- ★ Intersección
- ★ Diferencia

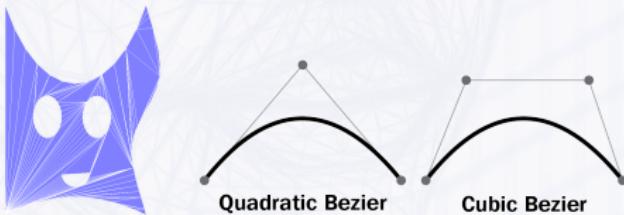


# Creación de geometría 2D

# Three.js

## 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



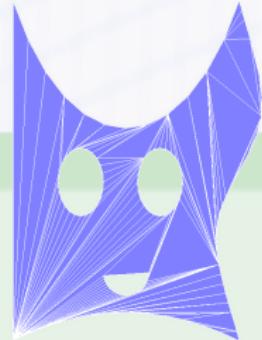
- Añadido de agujeros
  - ▶ Se crean los shapes de los agujeros
  - ▶ Se añaden al contorno exterior con el método `holes.push`

# 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 ellipse 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

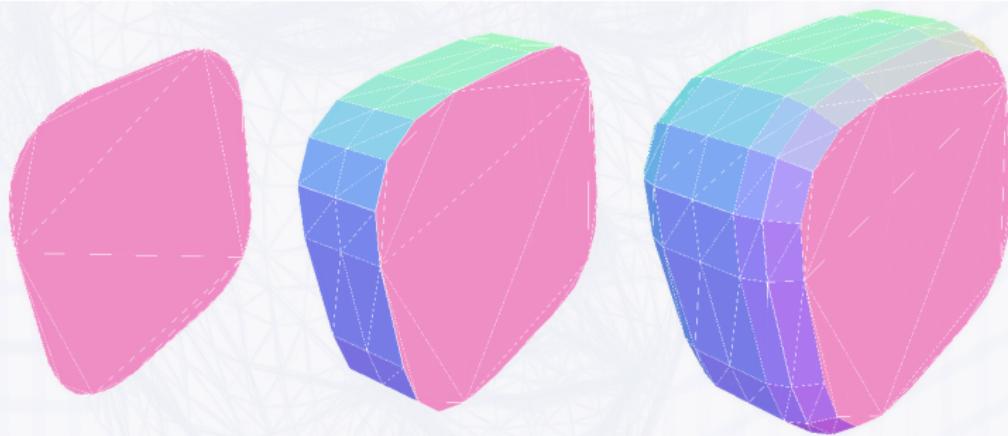
- ▶ shape: Un objeto Shape con el contorno
- ▶ options: Puede tener los siguientes parámetros opcionales
  - ★ amount: La cantidad de estrusión. Por defecto, 100
  - ★ bevelEnabled: Añadido del bisel. Por defecto, true
  - ★ bevelThickness: En la dirección de extrusión. Por defecto, 6
  - ★ bevelSize: 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);
```

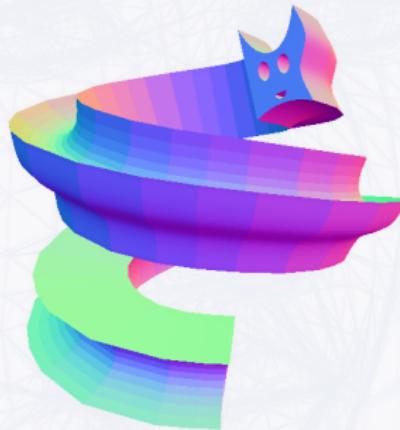


# Geometría por barrido

# 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);
```



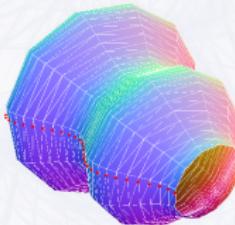
# Geometría por revolución

- **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

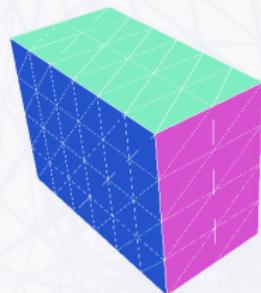
### Primitivas básicas: Cubo, Esfera, Cilindro y Toro

- **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,  $2\pi$ )

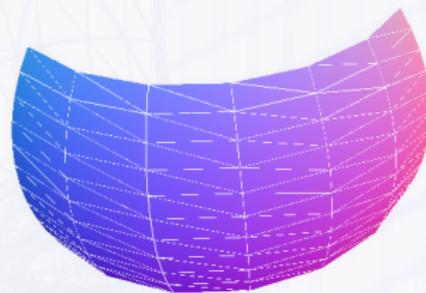
# Cubo, esfera, cilindro y toro

## Ejemplos

Three.js



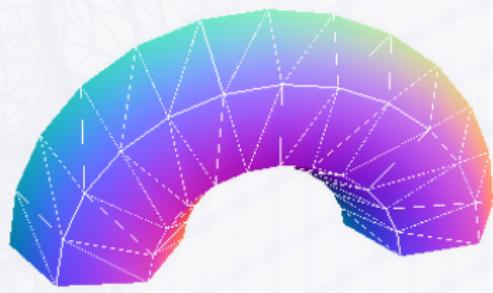
CubeGeometry (10,15,20, 2,4,6)



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



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



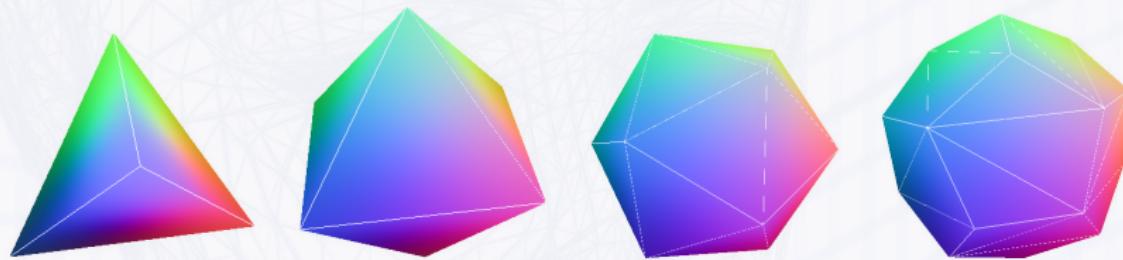
TorusGeometry (10,5, 6,12, 3)

# Geometría 3D

## Algunos poliedros regulares

## Three.js

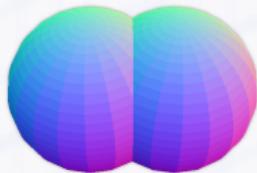
- `TetrahedronGeometry`, `OctahedronGeometry`,  
`IcosahedronGeometry`, `DodecahedronGeometry`
  - ▶ `radius`: El radio de la figura. Por defecto, 1



- 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 conmutativa
    - ★ El resultado es el primer operando MENOS la materia que ocupa el segundo operando



# Operaciones booleanas

# Three.js

- Se tiene que usar la biblioteca ThreeBSP
  - ▶ <https://github.com/skalnik/ThreeBSP>
  - ▶ Escrita en CoffeeScript, hay que compilarla hacia JavaScript  
coffee -compile ThreeBSP.coffee
- Procedimiento
  - ➊ Se crean las geometrías por los medios vistos
  - ➋ Se posicionan y orientan donde deban estar
    - ★ Se usan los siguientes métodos:  
`translate (x,y,z)`, `scale (x,y,z)`,  
`rotateX (angulo)`, `rotateY (angulo)`, `rotateZ (angulo)`
    - ★ ¡Cuidado! Esas transformaciones Sí se aplican a la geometría en el mismo orden en el que se escriben en el código

# Operaciones booleanas

# Three.js

- Procedimiento (continuación)
  - ③ Las geometrías se convierten a nodos ThreeBSP
  - ④ Se opera entre nodos ThreeBSP,  
el resultado es un nodo ThreeBSP
    - ★ Unión, método `union`
    - ★ Intersección, método `intersect`
    - ★ Diferencia, método `subtract`
  - ⑤ 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 JavaScript

```
// Previamente se crean las geometrías
var sphere1 = new THREE.SphereGeometry ( ... );

// Se posicionan y orientan
sphere1.translate (1, 0.5, 0.25);

// Se construyen nodos ThreeBSP
var sphere1bsp = new ThreeBSP (sphere1);

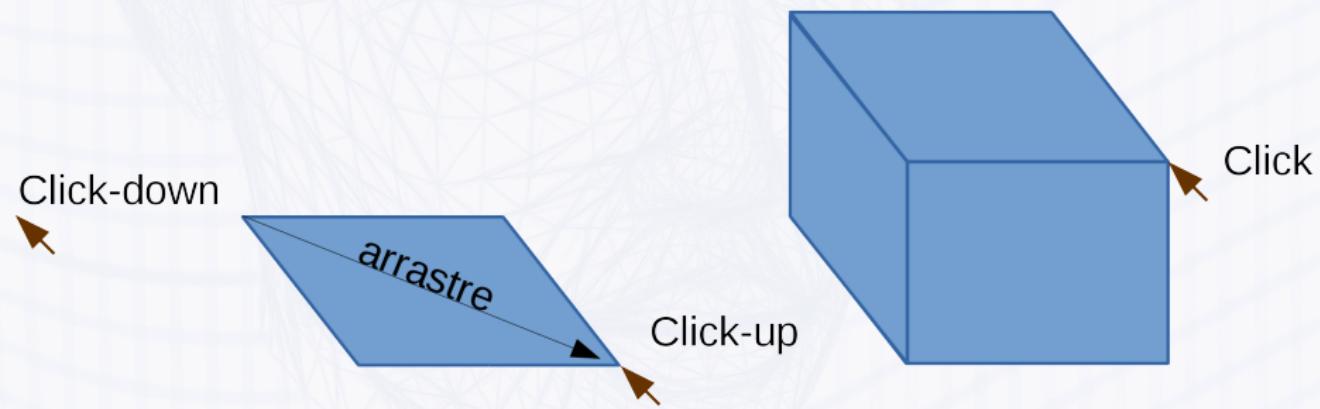
// Se construye el arbol binario con las operaciones
var partialResult = sphere1bsp.subtract (sphere2bsp);
var finalResult = partialResult.subtract (boxBsp);

// Y finalmente
var result = finalResult.toMesh (material);
result.geometry.computeFaceNormals ();
result.geometry.computeVertexNormals ();
```



# Creación de geometría interactivamente

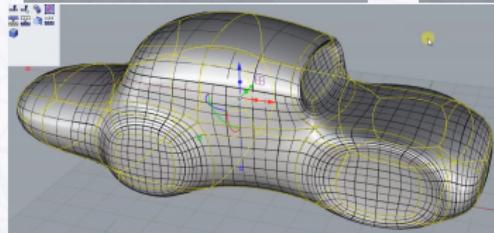
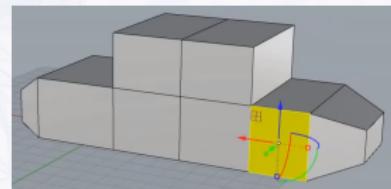
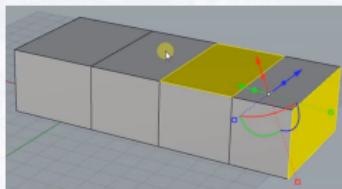
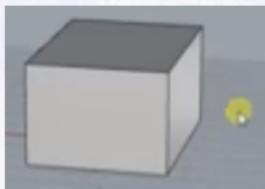
- Requiere procesar adecuadamente los eventos del ratón según el contexto
- Se van completando las listas de vértices e índices necesarias para construir la geometría
- Hay que ir mostrando el resultado parcial mientras se construye, el usuario necesita una realimentación



# Creación de geometría interactivamente

## Refinamiento progresivo de la malla

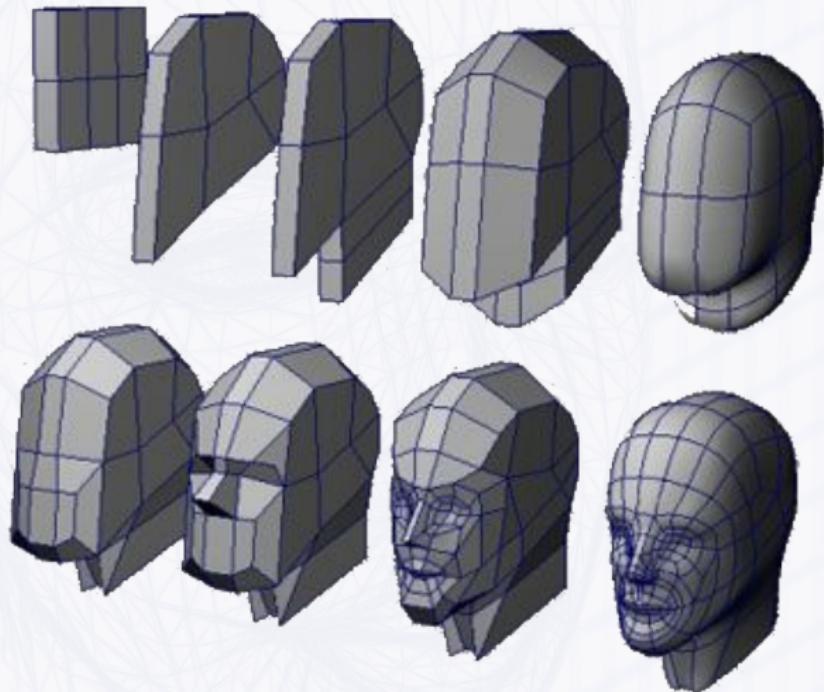
- Se parte de un sólido simple (puede ser una caja)
- Se operan las siguientes:
  - ▶ Subdivisión de polígonos
  - ▶ Desplazamientos de polígonos, aristas y/o vértices
  - ▶ Refinamiento de malla
- Se puede llegar a modelar geometrías complejas



Podéis ver el proceso completo en <https://youtu.be/Qhp0MGtMfro>

# Refinamiento progresivo de la malla

Otro ejemplo



# 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)
- **Sin embargo**, no hay operadores para geometría en GPU
  - ▶ Por ejemplo, no se tienen operaciones booleanas
- **Afortunadamente**, se puede crear una geometría en GPU a partir de cualquier geometría de CPU

### Ejemplo: Geometría en GPU

```
var geometria = // Construida 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
  - ▶ MTLLoader.js
  - ▶ OBJLoader.js
- Se cargan los materiales y el modelo (en ese orden)



# Carga de modelos .obj (ejemplo)

# Three.js

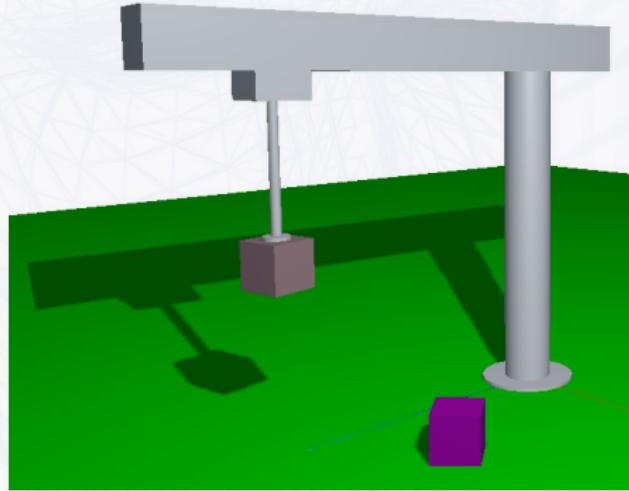
## Ejemplo: Carga de modelos .obj

```
class Modelo extends THREE.Object3D {  
    constructor() {  
        super();  
  
        var that = this;  
        var materialLoader = new THREE.MTLLoader();  
        var objectLoader = new THREE.OBJLoader();  
        materialLoader.load ('porsche911/911.mtl',  
            function (materials) {  
                objectLoader.setMaterials (materials);  
                objectLoader.load ('porsche911/Porsche_911_GT2.obj',  
                    function (object) {  
                        var modelo = object;  
                        that.add (modelo);  
                    }, null, null);  
            });  
        ...  
    }  
}
```

- El modelo se *manipula* operando con `this`
  - ▶ Por ejemplo: `this.position.x = 5;`

# Modelos jerárquicos

- Se combinan geometrías y transformaciones. Objetivos:
  - ▶ Crear figuras compuestas a partir de figuras simples, fáciles de editar y mantener
  - ▶ Tener objetos articulados fácil de animar y mantener

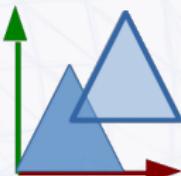


# Grafo dirigido acíclico

- Representa un modelo jerárquico
  - ▶ La raíz representa la figura completa
  - ▶ Los nodos internos son transformaciones (una por nodo)
    - ★ La transformación de un nodo afecta a todos sus hijos
  - ▶ Los nodos hoja representan geometría
    - ★ Una geometría se ve transformada por su padre, y por su abuelo, y así sucesivamente hasta llegar a la raíz (y en ese orden)
- En un buen diseño ...
  - ▶ Un movimiento en un grado de libertad del objeto debería implicar cambios en un solo nodo
  - ▶ A veces (con los escalados) esto no es posible

# Repasso a las transformaciones geométricas

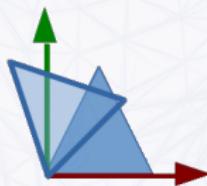
- Traslación



- Escalado uniforme



- Rotación



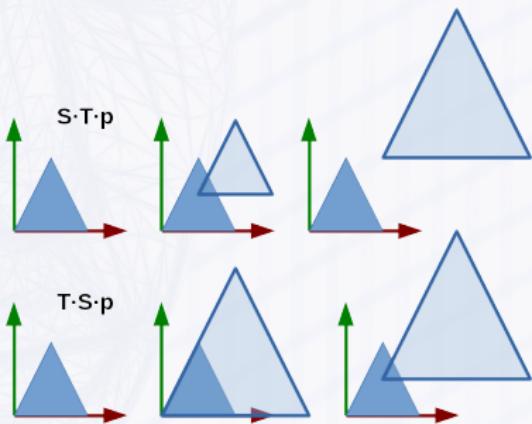
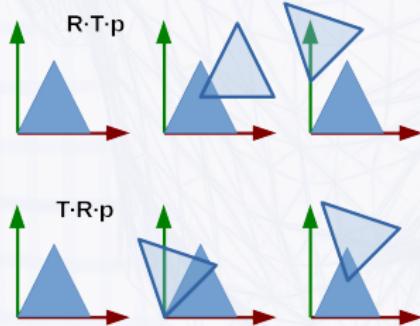
¡Cuidado!

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

# Resumen de las transformaciones geométricas

## Composición no comutativa de transformaciones

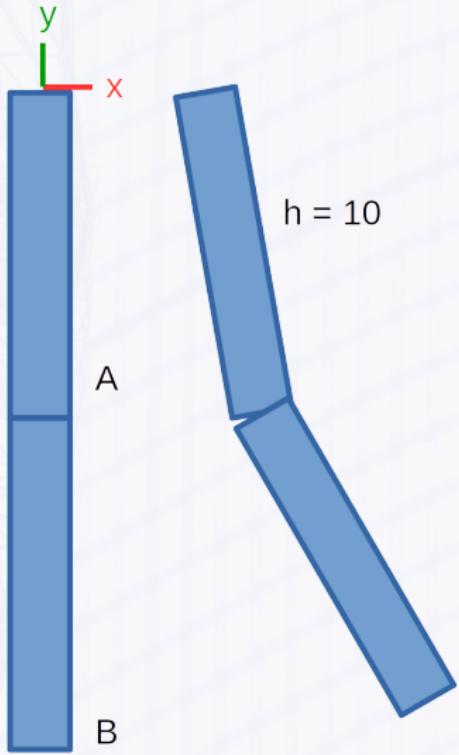
- Composiciones comutativas
  - ▶ Cualquier combinación de escalados uniformes y rotaciones
  - ▶ Cualquier combinación de traslaciones
- Composiciones no comutativas
  - ▶ Las traslaciones con las rotaciones o los escalados



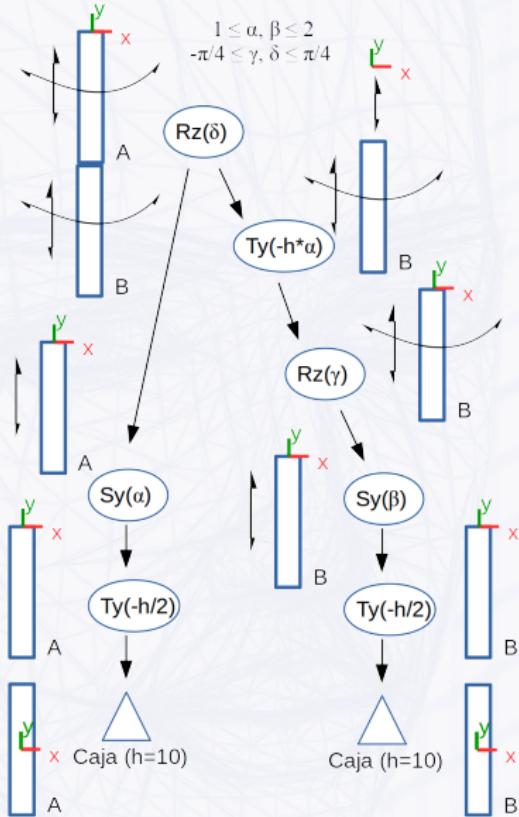
# Procedimiento para diseñar un grafo sin errores

## Ejemplo que combina traslaciones, rotaciones y escalados

- Grados de libertad
  - ▶ Las piezas A y B giran por su extremo superior
  - ▶ Las piezas A y B pueden aumentar su longitud al doble
  - ▶ Las piezas A y B siempre están conectadas como se muestra en la figura
- Hagámoslo en la pizarra



# Grafo que modela el objeto



# Transformaciones aplicadas DIRECTAMENTE a una Geometry

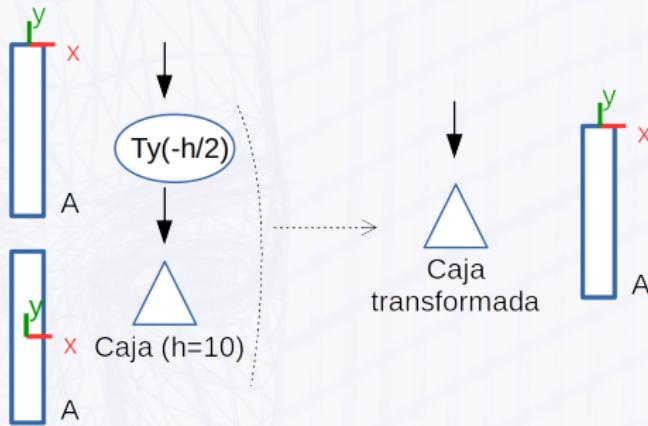
## Three.js

- Requisitos para poder aplicarlas
  - ▶ No pueden depender de variables que cambien su valor en el tiempo
  - ▶ Deben estar conectadas directamente a nodos de geometría (o a un nodo de transformación que también cumpla estos requisitos)
  - ▶ No pueden tener otros hijos
  - ▶ Es decir, una cadena de nodos de transformación
    - ★ Sin ramificaciones
    - ★ Que no dependen de variables
    - ★ Terminan en una geometría
- Se aplican una sola vez, en la construcción
- No se modifican a posteriori
- Deben aplicarse en el orden correcto, antes de construir el Mesh

# Transformaciones directas a una Geometry

## Three.js

- `scale (x, y, z)`
- `rotateX (radianes)`
- `rotateY (radianes)`
- `rotateZ (radianes)`
- `translate (x, y, z)`

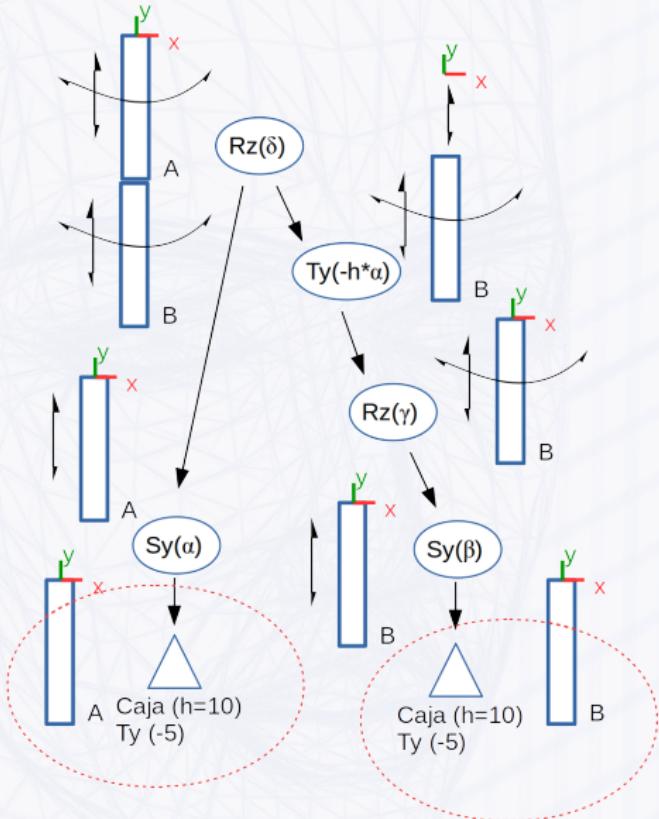


## Transformaciones

```
geometriaCaja = new THREE.BoxGeometry (1,10,1);
geometriaCaja.translate (0,-5,0); el orden es importante
materialCaja = new THREE.MeshPhongMaterial ({color: 0xCF0000});
caja = new THREE.Mesh (geometriaCaja, materialCaja);
```

## Three.js: Aplicadas a una Geometry

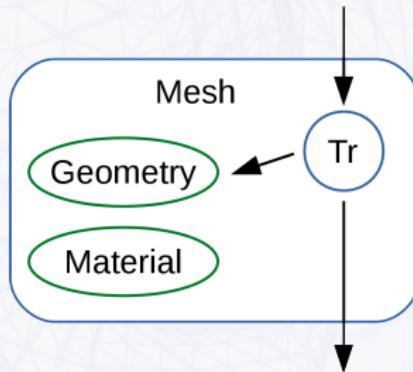
# Grafo con geometrías transformadas



# Transformaciones en un Mesh

Three.js

- La clase **Mesh**, además de Geometría y Material ...
- Incorpora un nodo de Transformación
- Dicha transformación se aplica en cada visualización
- Se usa para:
  - ▶ Mover el objeto ante una interacción del usuario
  - ▶ Realizar animación
- Los Mesh pueden tener hijos
  - ▶ La conexión con hijos y padre se realiza por dicha transformación



# Transformaciones en un Mesh

Three.js

- Se configuran fácilmente mediante sus atributos:
  - ▶ `position, rotation, scale`
  - ▶ Traslación, rotación y escalado, respectivamente

## Ejemplo: Gestión de la transformación de un Mesh

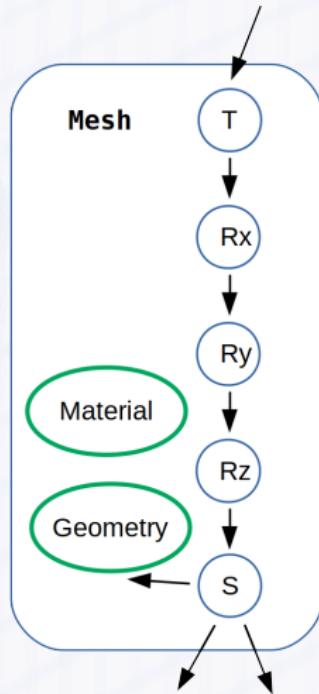
```
var pelota = new THREE.Mesh (unaGeometria, unMaterial);
pelota.rotation.y = 1;    // Las rotaciones son en radianes
pelota.position.x = 5;
pelota.scale.set (1,2,1); // Método para establecer las 3 dimensiones a la vez
```

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

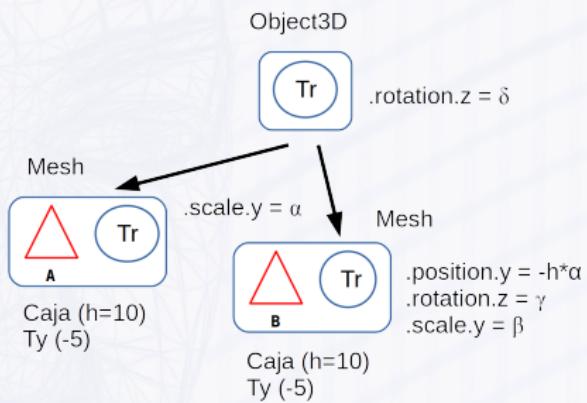
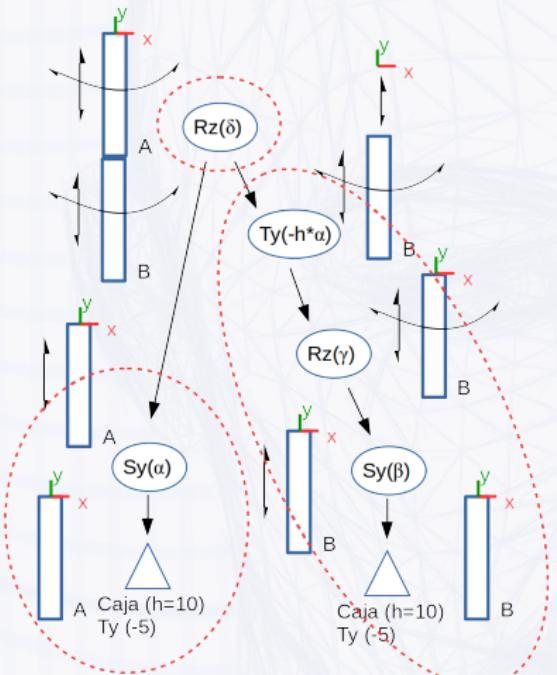
# Transformaciones en un Mesh

- Permiten compactar el grafo y obtener un grafo con menos nodos para implementarlo
- Deben cumplirse ciertas condiciones

## Three.js



# Grafo compactado para implementarlo en Three.js



## Nodos internos

## Three.js

- 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

# 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 con respecto a su orientación inicial
- ▶ 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

# **Creación de geometría**

Francisco Velasco Anguita

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

Sistemas Gráficos

Grado en Ingeniería Informática  
Curso 2019-2020