Motores de Física (introducción)

Sistemas Gráficos

Grado en Ingeniería Informática

Curso 2019/2020

1. Introducción

En una escena gráfica que represente una escena real, se quiere poder representar las características que tiene el mundo real. Los objetos caen por la fuerza de la gravedad, tienen masa que afecta a sus reacciones, si dos objetos colisionan entre sí, el de menos masa tiene una reacción mayor que el de más pesado, cuando un objeto se desliza por una superficie se ve frenado en mayor o menor medida según su coeficiente de rozamiento, etc.

Para implementar todos esos fenónmenos sería necesario representar las magnitudes físicas involucradas con dichos fenómenos, incorporarlas en los objetos y en cada frame, aplicar las ecuaciones físicas correspondientes para determinar la nueva posición y orientación de cada objeto según las fuerzas que actúan sobre él y su inercia lineal y angular.

Eso es precisamente lo que hace un motor de física. Es una biblioteca que representa y gestiona diferentes magnitudes físicas permitiendo dotar a la escena de esos efectos del mundo real.

Como es de imaginar, se requiere bastante cálculo para ello. Así que se suelen separar los cálculos de la gestión de la física y los cálculos de la gestión de la visualización en hebras distintas.

En prácticas se usará la biblioteca `Physijs`, que permite incorporar física a una escena modelada con Three. Dado que son bibliotecas desarrolladas por distinta gente y cada una

tiene su evolución, en las aplicaciones que usen Physijs, **es obligatorio** usar la versión de Three.js que viene con la propia biblioteca Physijs. Dicha biblioteca se puede descargar de `github.com/chandlerprall/Physijs`.

De todos modos, hay que decir que Phisijs no es en sí mismo un motor de física, sino que es un *wrapper* que hace más fácil el uso de `ammo.js`, que sí implementa un motor de física.

Una escena básica con Physijs

Además de descargarse la biblioteca y añadir el archivo `physi.js` en el `html` y modificar el `html` para usar la versión de `three.js` que trae `Physijs`, es necesario indicarle a Physijs cuál es el *web worker* que se va a usar para gestionar las hebras e indicar igualmente que se va a usar `ammo` como motor de física.

Por lo demás, una escena Physijs es similar a una escena Three pero usando versiones físicas de algunos elementos. En particular:

- La escena debe ser física.
  ```
  escenaFisica = new Physijs.Scene().
  ```
- Hay que establecer el valor de la gravedad. Es un vector, se puede tener gravedad hacia cualquier dirección y de cualquier valor.
  ```
  escenaFisica.setGravity (new THREE.Vector3 (0,-10,0))
  ```
- Los materiales deben ser físicos, aunque se basan en un material Three.
  ```
  mat = new THREE... // El que se quiera
  matFisico = Physijs.createMaterial (mat, 0.9, 0.3)
  ```
- Los mesh deben ser físicos. Se crean a partir de una geometría Three.
  ```
  geom = new THREE.BoxGeometry ( . . .)
  meshFisico = new Physijs.BoxMesh (geom, matFisico, 25)
  ```
- Los elementos que se desee sean tenidos en cuenta por el motor de física deben colgar **DIRECTAMENTE** del nodo raiz de la escena.
  ```
  escenaFisica.add (meshFisico)
  ```
  Hay alguna excepción a esta regla pero por lo general hay que cumplirla.
- Indicar, en el update principal, que la escena realice la simulación física. Que aplique toda la formulación física que corresponda y actualice los objetos en consecuencia.
  ```
  escenaFisica.simulate()
  ```

No todos los elementos en una escena física deben ser físicos, solo aquellos que se desee que sean tenidos en cuenta por el motor de física.

Veamos cómo quedaría un posible clase `MiEscenaFisica`.

```
// La clase deriva de la escena física
class MyPhysiScene extends Physijs.Scene {

  constructor (myCanvas) {

    // El gestor de hebras
    Physijs.scripts.worker = './physijs/physijs_worker.js'
    // El motor de física de bajo nivel, en el cual se apoya Physijs
    Physijs.scripts.ammo   = './ammo.js'

    // Las dos líneas anteriores DEBEN EJECUTARSE ANTES de inicializar Physijs.Scene.
    //   En este caso, antes de llamar a super
    super();

    // Se crea el visualizador,
    //   pasándole el lienzo sobre el que realizar los renderizados. Esto no cambia.
    this.renderer = this.createRenderer(myCanvas);

    // Se establece el valor de la gravedad, negativo en la Y, los objetos caen hacia abajo
    this.setGravity (new THREE.Vector3 (0, -10, 0));

    // Se construye una figura física

    // Se crea un material físico en base a un material Three
    var mat = new THREE.MeshPhongMaterial ({color: 0xff0000});
    var matFisico = Physijs.createMaterial (mat, 0.9, 0.3);

    // Se crea una geometría Three
    var geom = new THREE.BoxGeometry (1, 3, 2);

    // Se crea un mesh físico
    this.figuraFisica = new Physijs.BoxMesh (geom, matFisico, 25);

    // IMPORTANTE: Los elementos que se desee sean tenidos en cuenta en la FISICA
    //   deben colgar DIRECTAMENTE de la escena. NO deben colgar de otros nodos.

    this.add (this.figuraFisica);
  }

  update() {
    // Entre otras cosas

    this.simulate();
  }
}
```

2. Magnitudes físicas

Ya se ha visto que una de las magnitudes que se representa es la fuerza de la gravedad, configurándola en la escena física.

En cuanto a magnitudes físicas aplicables a los objetos se tiene la masa del objeto, un coeficiente de rozamiento y un coeficiente de restitución (rebote).

La **masa** se indica al instanciar un Mesh físico, que tiene tres parámetros: la geometría Three, el material físico y la masa. Si se pone un valor de masa 0 a dicho objeto no le afectará la fuerza de la gravedad.

Será necesario ponerle masa 0 a lo que haga de suelo, las paredes si éstas no deben caerse y en general a cualquier cosa que no se quiera que sea afectada por la gravedad, colisiones, impulsos, etc. Estos objetos no se va a caer, volcarse ni salir rebotados en una colisión.

```
var suelo = new Physijs.BoxMesh(geometriaThree, materialFisico, 0); // masa 0
```

Si no se indica ningún valor, por defecto se asignará un valor de masa directamente proporcional al volumen del objeto. Los objetos más voluminosos por defecto son más pesados.

Los coeficientes de **rozamiento** y **rebote** se indican al instanciar un material físico. Son valores entre 0 y 1 con el significado que se intuye. Un valor alto de rozamiento indicará que el objeto físico que tenga dicho material costará mucho arrastrarlo. Un valor alto de rebote indicará que el objeto físico que tenga dicho material se verá muy afectado en una colisión con otro objeto.

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

También hay que tener en cuenta la masa en las colisiones, si un objeto muy pesado colisiona con uno muy ligero (poca masa), el segundo se verá más afectado que el primero.

Los valores concretos que, para cada magnitud física, se le dé a un objeto deben darse teniendo en cuenta las características que tendrían esos objetos en el mundo real, figura 1.

Figura 1: Los objetos ligeros son más *dinámicos* que los pesados.

3. Tipos de figuras físicas

La biblioteca Physijs dispone de varias clases para crear Mesh físicos. La geometría concreta que se visualizará de la figura será la geometría Three que se haya usado para construir el Mesh físico. Pero el motor de física realizará los cálculos considerando que la geometría de la figura es la que se corresponde con la clase Physijs que se ha usado para construirla.

Es decir, si construimos un Mesh físico usando la clase `Physijs.BoxMesh`, esa figura física no rodará, y no lo hará aunque la construyamos pasándole una esfera como geometría Three. Se vería una esfera un poco rara que se desliza (arrastrando) pero no rueda.

```
var matFisico = Physijs.createMaterial ( . . . );
var geometria = new THREE.SphereGeometry (5);
var figuraFisica = new Physijs.BoxMesh (geometria, matFisico, 150);

// Se tiene una esfera que físicamente se comporta como una caja.
// No rueda, se desliza.
```

Se deben construir las figuras físicas usando la clase que mejor se adapte a la geometría que se use para construir el Mesh físico. Los Mesh físicos disponibles son:

- Physijs.BoxMesh
- Physijs.SphereMesh
- Physijs.CylinderMesh
- Physijs.ConeMesh
- Physijs.ConvexMesh

La clase `ConvexMesh` está pensada para aquellas geometrías que no encajan en las anteriores. Pero su procesamiento es más lento, por lo que se debería evitar su uso en la medida de lo posible, intentando construir las figuras físicas con alguna de las otras clases. Si no hay ninguna que se pueda considerar la mejor, elegir la menos mala.

Objetos compuestos

En el caso de que se necesite crear un objeto compuesto por varias figuras, y que todas sean consideradas por el motor de física:

- Debe haber una jerarquía entre ellas.
- La raiz de la jerarquía debe ser una figura física.

- La jerarquía debe estar hecha **antes** de añadir la raiz de la jerarquía a la raíz de la escena.

El siguiente ejemplo muestra un fragmento de cómo un suelo y unas paredes (objetos físicos) se relacionan jerárquicamente y se añaden a la escena.

```
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;

// Primero se monta la jerarquía
suelo.add (paredIzq);

// Luego se cuelga la raíz de la jerarquía en la raíz de la escena
scene.add (suelo);
```

Formas especiales

Hay otro Mesh físico que sirve para crear terrenos o suelos ondulados. Se trata de la clase `Physijs.HeightfieldMesh`, que parte de una geometría de Three que es un plano con una determinada resolución, y donde la coordenada Z de cada vértice representa la altura en dicho punto. Se ponen en dichas Z las alturas deseadas y finalmente se construye el Mesh físico que representa ese terreno con ondulaciones.

Para dar diferente altura de cada punto se puede hacer aleatoriamente o de acuerdo con una función donde reciba la (x, y) del punto y devuelva la altura en ese punto.

```
// Se va a hacer un suelo ondulado de tamaño 60 x 50 con una resolución de 100 x 100
// Se comienza con una geometría plana de Three, con ese tamaño y esa resolución
var tamaX = 60;   var tamaY = 50;   var resolucion = 100;
var sueloGeometria = new THREE.PlaneGeometry (tamaX, tamaY, resolucion, resolucion);

// Ahora se recorren todos los vértices para modificar su altura
// La altura se calcula con una función, calculaAltura, que recibe dos parámetros (x,y)
//   con valores entre 0 y 1 cada uno que indica en qué parte del suelo se está
var x = 0.0;
var y = 0.0;
for (var i = 0; i < sueloGeometria.vertices.length; i++) {
  x = sueloGeometria.vertices[i].x / tamaX;
  y = sueloGeometria.vertices[i].y / tamaY;
  sueloGeometria.vertices[i].z = calculaAltura (x,y);
}

// Al haber modificado los vértices hay que recalcular las normales
sueloGeometria.computeFaceNormals();
sueloGeometria.computeVertexNormals();

// Por último, se construye el suelo físico ondulado a partir de dicha geometría
// Hay que volver a darle la resolución y debe coincidir.
var sueloFisico = new Physijs.HeightfieldMesh (sueloGeometria, sueloMaterial,
    0,   // masa 0 para que no se caiga por la gravedad
    resolucion, resolución);

// Como el suelo se ha creado en el plano X,Y con las alturas en Z
// Y el suelo se querrá en el plano X,Z con las alturas en Y, se gira 90º
sueloFisico.rotation.x = -Math.PI / 2;
```

Un posible resultado se observa en la figura 2.

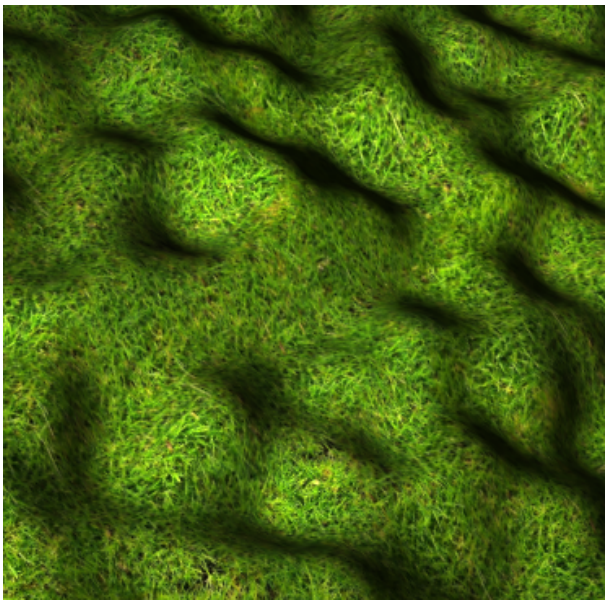

Figura 2: Para realizar un terreno ondulado se recurre a la clase Physijs.HeightfieldMesh

4. Interacción con las figuras

Cuando es un motor de física el que gestiona la escena, la posición, orientación y velocidad de cada objeto viene determinado por lo que está ocurriendo en la escena en ese momento: la fuerza de la gravedad, si ha colisionado con otro objeto, el rozamiento que tenga con otros objetos, etc.

No obstante, se sigue pudiendo interactuar manualmente con la escena.

Poner una posicion o una orientación manualmente

Como en cualquier objeto, se puede hacer modificando los atributos `position` y/o `rotation`. Pero hay que indicarle al motor de física que se han modificado manualmente esos atributos para que los tenga en cuenta y no reescriba los valores que se le han dado manualmente con los valores que el motor de física calcule. Para ello, cada vez que se modifiquen manualmente esos atributos hay que poner a `true` los atributos `__dirtyPosition` y/o `__dirtyRotation` respectivamente[1].

[1]No se aprecia bien pero esos atributos comienzan con 2 guiones bajos.

```
// Se modifica la posición de una figura manualmente
this.unaFigura.position.set (0,25,0);

// Hay que indicarlo para que el motor de física tenga en cuenta la nueva posición
this.unaFigura.__dirtyPosition = true;
```

Establecer manualmente una velocidad

Las figuras físicas disponen de dos métodos para establecerles velocidad, un método para la velocidad lineal, y otro para la velocidad angular. Ambos métodos reciben como parámetro un vector tridimensional, al igual que la magnitud física velocidad, que también es un vector.

Dicho vector nos indica la dirección en la que actúa la velocidad y el módulo de dicho vector nos indica y la cantidad de velocidad.

Los métodos son `setLinearVelocity(vector3)` y `setAngularVelocity(vector3)` respectivamente. También se tienen los getter correspondientes para leer las respectivas velocidades.

Ejemplos típicos en los que se usan dichos métodos serían.

Imaginad que una caja se está deslizando y se quiere frenarla en seco. Bastaría modificarle la velocidad lineal a dicha caja en el método que se ejecute cuando se desee realizar ese frenado. Y como velocidad se le pondría 0 en cada dimensión.

```
var velocidadCero = new THREE.Vector3(); // Por defecto, (0,0,0)
this.unaCaja.setLinearVelocity (velocidadCero);
```

Cuando una esfera está rodando sobre un suelo plano, en Physijs no se detiene, lo cual no es natural. En el mundo real, cualquier esfera que está rodando sobre un suelo plano acaba parándose más pronto que tarde.

Se podía pensar en aumentarle el rozamiento al material, pero no se consigue nada, ya que las esferas se mueven rodando, no deslizando. La solución está en añadir en el método `update` líneas para ir atenuando poco a poco la velocidad angular de la esfera. En el ejemplo, en cada frame la velocidad se reduce en un 5 % por lo que termina deteniéndose.

```
// Se lee la velocidad actual
var velocidadAngular = this.unaEsfera.getAngularVelocity();

// Se le vuelve a poner la misma velocidad pero un poco atenuada
var atenuacion = 0.95;
this.unaEsfera.setAngularVelocity (velocidadAngular.multiplyScalar(atenuacion));
```

Empujar objetos

Otra manera de interactuar con las figuras físicas es aplicarles impulsos. Para aplicar un impulso, es decir una fuerza, se necesita otro vector tridimensional, donde el vector indica la dirección, y el módulo del vector indica la cantidad de fuerza.

El método que le aplica el impulso a una figura física se denomina `applyCentralImpulse`.

Para considerar de manera independiente la dirección y la cantidad de la fuerza, se puede normalizar la dirección para a continuación multiplicar dicho vector normalizado por la cantidad de fuerza deseada. Obteniendo así un vector que combina tanto la dirección como la cantidad de fuerza que se quiere aplicar.

```
// La cantidad de fuerza que se desea aplicar
var fuerza = 10;

// La dirección
var offset = new THREE.Vector3 (1,2,3);

// Se calcula un vector que respete esa dirección pero que su módulo sea
//   el de la cantidad de fuerza que se desea aplicar.
var effect = offset.normalize().multiplyScalar(fuerza);

// Se aplica el impulso al objeto físico
this.objetoFisico.applyCentralImpulse (effect);

// Si se quiere aplicar el impulso en la dirección opuesta solo hay que negar el vector
this.objetoFisico.applyCentralImpulse (effect.negate());
```

5. Procesado de colisiones

Una de las ventajas de usar un motor de física es que se detectan y procesan las colisiones. Se evalúa cuándo dos figuras colisionan, sus respectivas velocidades, geometrías, masas, coeficientes de restitución (rebote), etc. y calcula las nuevas velocidades para cada figura.

Cada figura sale despedida de esa colisión en la dirección calculada, y a la velocidad lineal y angular que se haya calculado.

Observar el ejemplo que se ha subido a Prado, *física*, la gestión que realiza el motor de física detectando las colisiones y procesando el resultado de cada colisión produce un resultado bastante natural. Se muestra una captura en la figura 3.

Se puede programar un listener asociado a una figura física que será llamado cuando colisione con otra figura física. Dicha función recibe diversa información relacionada con la

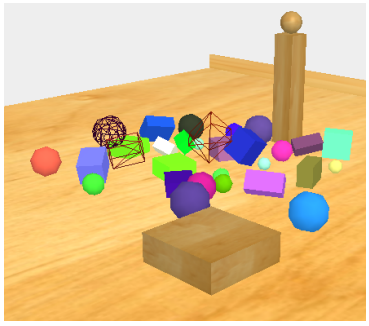

Figura 3: La detección y procesamiento de las colisiones es una de las grandes ventajas de un motor de física.

colisión, como quién es la otra figura, a qué velocidad relativa linear y angular se ha producido la colisión y la normal de contacto.

No obstante, la programación más habitual que se realiza de esta función, sobre todo en un video juego es anotar puntos, restar vidas, hacer desaparecer a alguna de las figuras, etc. Ya que el resultado natural de que ambas figuras salgan rebotadas según las condiciones en las que se produjo la colisión se realiza siempre, automáticamente, sin que haya que programar nada extra.

Las figuras físicas que tengan asociado un listener de colisiones **deben** colgar directamente de la raiz de la escena.

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

6. Restricciones

Con lo visto hasta el momento, hay bastante libertad de movimientos en los objetos en una escena física. Se mueven por el efecto de la fuerza de gravedad, por los impulsos que se le den, por el resultado de las colisiones entre ellos, etc.

Pero qué pasaría si, por ejemplo, se intentara abrir una puerta y se le aplicara un impulso para hacerlo. Pues que la puerta se caería al suelo. Igual que si se pone una tabla vertical y se le aplica un impulso. No hay nada que obligue a esa tabla vertical a comportarse como una puerta que gira por sus bisagras.

Un motor físico físico aumenta su usabilidad si se le pueden poner limitaciones en los movimientos (restricciones) a las figuras físicas. Así, en el ejemplo anterior, la tabla vertical se

comportará como una puerta si se le puede poner una restricción que solo le permita girar por un eje vertical que esté en un extremo, simulando la restricción que unas bisagras le imponen a una puerta real.

En esta sección se explican diferentes restricciones que se pueden poner a las figuras física. En todas ellas **el orden en el que se realizan las siguientes acciones es importante**. Si las diferentes instrucciones no se realizan en el orden descrito es muy probable que las restricciones no surtan el efecto esperado.

1. Los objetos que intervienen en una restricción se añaden a la escena
2. Se define la restricción referenciando los objetos involucrados
3. Se añade la restricción a la escena
   El método para añadir restricciones a la escena no es add,
   se denomina `addConstraint`
4. Se configura la restricción si hay algo que configurar

6.1. Restricción punto a punto

Es la restricción más básica. Se restringe la posición de un objeto con respecto a otro. Es como si los dos objetos estuvieran conectados con un elemento invisible recto y rígido.

De modo que cuando uno se mueve el otro le sigue manteniendo la misma distancia y orientación relativas.

Se realiza con la clase `Physijs.PointConstraint`.

```
// Los objetos obj1 y obj2 deben estar en la escena ANTES de definir la restricción
var restric = new Physijs.PointConstraint (obj1, obj2, obj2.position);

// Las restricciones también deben añadirse a la escena. Pero con otro método
scene.addConstraint (restric);
```

En el ejemplo, los objetos 1 y 2 quedan *enlazados* y a partir de ese momento se mueven solidariamente.

6.2. Restricción tipo bisagra

Con esta restricción se consigue que un objeto solo pueda girar con respecto a un eje determinado.

En la figura 4 la caja solo gira alrededor del cilindro, no puede separarse del cilindro, ni caerse, ni ningún otro movimiento.

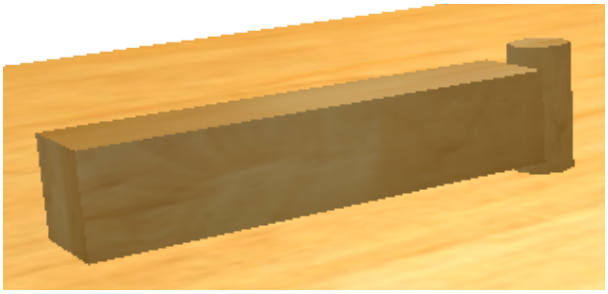

Figura 4: La caja solo puede girar como si el cilindro fuese una bisagra.

Se realiza con la clase `Physijs.HingeConstraint`.

```
1 var restric = new Physijs.HingeConstraint (
2       objMovil, objFijo, objFijo.position,
3       new THREE.Vector3 (0,1,0));    // el eje de la bisagra
4 scene.addConstraint (restric);
5
6 // PRIMERO: Se añade la restricción a la escena, LUEGO se configura
7 // Configuración de la restricción:
8 // - Angulo mínimo y máximo
9 // - Con qué fuerza intenta evitar salirse de límites < 0.5
10 // - Rebote al llegar a límites
11
12 retric.setLimits (-Math.PI/2, Math.PI/2, 0.1, 0.1);
13
14 // Para moverlo intencionadamente
15 // Velocidad positiva mueve en un sentido, negativa en el opuesto
16
17 restric.enableAngularMotor (velocidadMaxima, aceleracion);
18
19 // Para desactivar el motor
20 //    (solo se movería por gravedad o colisiones)
21
22 restric.disableMotor();
```

En la línea 2 se indica cuál es el objeto móvil, el que se mueve como una puerta, el objeto fijo con respecto al cual se va a vincular el móvil, y qué punto concreto es el que se usa para la restricción, en este caso el atributo position, que normalmente indica el centro de dicha figura.

En la línea 3 es donde se indica cual es el eje de giro, en este caso el eje vertical. Por tanto, el eje de giro es el eje vertical que pasa por el punto `objFijo.position`.

En la línea 4 se añade a la escena con `addConstraint` y **después** se configura la restricción, si es que es necesario.

La configuración se hace en la línea 12. En este caso se han puesto unos límites de giro de 90^{o} para cada lado. Los siguientes dos parámetros a $0{,}1$ son menos importantes, e indican una resistencia de la puerta a sobrepasar el límite establecido o un efecto rebote cuando se llega a dicho límite.

Con la restricción tal cuál está hasta la línea 12 sería suficiente y la puerta giraría por impulsos, gravedad, colisiones, etc.

Pero se le puede poner a la puerta una especie de *motor* para forzar su movimiento. Un ejemplo de activación del motor se realiza en la línea 17. Se trata de un motor que, desde parado, mueve la puerta con la aceleración indicada hasta que llega a la velocidad máxima y a partir de ahí, la puerta sigue girando a esa velocidad. Siempre dentro de los límites de giro que se hayan configurado. Tanto la velocidad como la aceleración son `float`. Una velocidad positiva la mueve en un sentido y una velocidad negativa la mueve en el otro.

Ese motor tiene bastante fuerza, mientras está activo es difícil que la puerta se mueva porque un objeto colisione con ella. Es más, si en su movimiento colisiona con un objeto que tenga libertad de movimientos es posible que lo impulse.

Por ejemplo, las típicas barras que maneja un jugador en un pinball para golpear la bola se harían con una técnica como esta. La figura 4 puede ser una versión un poco tosca en cuanto geometría de esas barras, pero funcional.

Cuando se quiera desactivar dicho *motor* se ejecutaría una instrucción como la mostrada en la línea 22.

6.3. Restricción de deslizamiento

Un objeto con restricción de deslizamiento solo puede moverse a lo largo de una recta.

La restricción se crea indicándole la dirección por la que puede desplazarse. También se le pueden establecer límites en ese desplazamiento y al igual que en la restricción de bisagra se le puede activar y desactivar un motor.

Como podría esperarse, la dirección de desplazamiento se indica con un `Vector3`. Sin embargo, **las coordenadas de este vector no son nada intuitivas**.

Desplazamiento por	Vector a usar
eje x	$(0, 1, 0)$
eje y	$(0, 0, \pi/2)$
eje z	$(\pi/2, 0, 0)$

Desconozco la razón de esta peculiar forma de definir la dirección de deslizamiento.

Se define con la clase `Physijs.SliderConstraint`.

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

// Límites al movimiento, distancia mínima y máxima
retric.setLimits (-10, 10);

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

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

Como puede verse, con la salvedad de cómo se indica la dirección de desplazamiento, la definición y configuración de esta restricción es similar a la restricción de bisagra.

6.4. Restricción tipo péndulo

Con esta restricción, la figura solo puede oscilar como lo haría un péndulo, figura 5. Se pueden establecer ángulos límite en cada eje, activar y desactivar un motor.

Se define con la clase `Physijs.ConeTwistConstraint`.

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

// Límites al movimiento, 3 ángulos para los 3 ejes
retric.setLimits (Math.PI/2, 0, Math.PI/2);

// Para moverlo intencionadamente unos determinados ángulos por eje
restric.enableMotor ();
restric.setMotorTarget (new THREE.Vector3 (1, 0, 1.5));

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

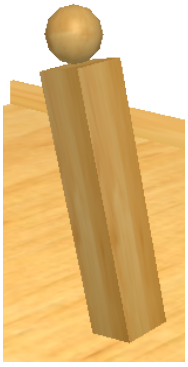

Figura 5: Con esta restricción la caja oscila como un péndulo.

En la línea 6 se establecen los límites de esa oscilación. Desde la vertical, el péndulo puede oscilar en cada eje (o combinación de ellos) hasta el límite indicado. Tanto en ángulos positivos como negativos.

Las líneas 9 y 10 activan un motor. En este caso el péndulo quedaría con la orientación combinada que resulte de inclinarlo 1 radian con respecto al eje x y 1.5 radianes con respecto al eje z.

6.5. Restricciones mediante otros objetos

Otra forma de restringir los movimientos de unas figuras es colocar otras figuras que impidan los movimientos no deseados.

Por ejemplo, imaginar que se está haciendo una juego de billar y se desea evitar que las bolas salten y puedan caerse de la mesa. Podría hacerse poniendo una tapa por encima de la mesa de manera que si debido a una colisión entre bolas alguna saltara, tropezara con esa tapa que le está haciendo de límite vertical.

El problema ahora estaría en que dicha tapa lo ocultaría todo y no se verían las bolas. Se soluciona poniendo esa tapa totalmente transparente configurando adecuadamente los atributos de transparencia de su material `{opacity: 0.0, transparent: true}`.

6.6. Restricción genérica de grados de libertad

Se trata de una restricción que es bastante configurable permitiendo controlar de manera exacta los movimientos de un objeto. Se pueden definir límites en los grados de libertad, mo-

dificar los límites cuando sea necesario, mover los elementos a conveniencia activando algún motor, y por supuesto, desactivar los motores que se hayan activado.

Esta restricción se define con la clase `Physijs.DOFConstraint` (Degree Of Freedom Constraint). Se explica su uso mediante un ejemplo, un coche como el de la figura 6.

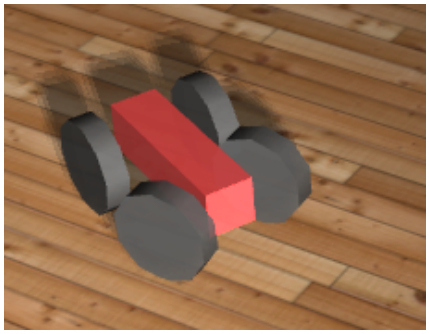

Figura 6: El movimiento de este Fórmula 1, tanto en lo relativo a la propulsión como a la dirección se gestiona con restricciones DOFConstraint.

En los ejemplos de código que se incluyen relativos al coche no se indica sin las variables son `var variable` o `this.variable` por claridad de escritura y porque tampoco eso es relevante para comprender el ejemplo. Cuando se implementen estas técnicas en una aplicación concreta hay que definir y usar las variables como corresponda a cada caso.

La construccion del F1

La construcción del F1 comienza con la construcción y colocación de sus 5 piezas, carrocería y ruedas. Al hacer el material físico de las ruedas, darle mayor o menor coeficiente de rozamiento en función de cómo se quiera que *agarren* los neumáticos.

```
1 // El coche se construye de la manera habitual, pero usando figuras físicas
2
3 var carroceria = new Physijs.BoxMesh ( . . . );
4 scene.add (carroceria);
5
6 // Rueda Front Right
7 var ruedaFR = new Physijs.CylinderMesh ( . . . );
8
9 // Se gira porque el cilindro sale 'tumbado'
10 ruedaFR.rotation.x = Math.PI/2;
11
12 // Se posiciona correctamente con respecto a la carrocería
13 ruedaFR.position.set ( . . . );
14
15 // Las ruedas se añaden a la escena, no a la carrocería
16 scene.add (ruedaFR);
17
18 // Y así con las otras 3 ruedas
```

Prestar atención a la línea 16. Las ruedas se cuelgan directamente de la raiz de la escena, no del nodo de la carrocería.

Restricciones de las ruedas

Las ruedas tienen 2 restricciones principales.

Por un lado deben ir siempre con la carrocería, de modo que el coche, pase lo que pase, no pierda las ruedas. Si una rueda gira y por tanto se va moviendo por el suelo la carrocería siempre la acompaña. Y también al revés, si la carrocería se mueve, por el motivo que sea, las ruedas siempre la acompañan.

De hecho, se está haciendo un coche con propulsión trasera.

¿Esto que significa? Que para hacer que el coche se desplace se hacen girar las ruedas traseras. Estas al girar se desplazan por el suelo. La carrocería, cuya posición en el escenario está restringida a la posición de las ruedas, se desplazará con ellas.

Por otro lado, dado que la posición de las ruedas delanteras está restringida a la posición de la carrocería, cuando se mueva la carrocería las ruedas delanteras se desplazará con ella.

Igual que en un coche de propulsión trasera del mundo real.

Por otro lado, las ruedas tienen otra restricción. Solo pueden girar por su eje transversal. Como cualquier rueda del mundo real.

```
1 // Restricción para la Rueda Front Right
2 // La rueda se 'pega' a la carrocería en la posición indicada y serán inseparables
3 var restriccionFR = new Physijs.DOFConstraint(ruedaFR, carroceria, ruedaFR.position);
4
5 // Se añade la restricción a la escena
6 scene.addConstraint(restriccionFR);
7
8 // Se definen sus límites para los movimientos libres
9
10 // Solo se permite girar por el eje z,
11 // y como el límite inferior es mayor que el límite superior,
12 // se permite el giro completo, 2 PI radianes
13 restriccionFR.setAngularLowerLimit({ x: 0, y: 0, z: 0.1 });
14 restriccionFR.setAngularUpperLimit({ x: 0, y: 0, z: 0 });
15
16 // Igual para las otras 3 ruedas
```

Código 1: Definición y configuración de las restricciones de las ruedas

En la línea 3, para dar el punto de anclaje entre rueda y carrocería se toma la posición de la rueda. Solo hay que dar un punto que sea correcto, lo mismo da tomarlo en referencia a un objeto u otro.

Recordar que el orden en el que se hacen las cosas es importante. Primero se ha creado la restricción (línea 3), después se añade a la escena (línea 6) y por último se configura (líneas 13 y 14).

En cuanto a los límites establecidos en las líneas 13 y 14, para los ejes x y y, son 0 para el inferior y 0 para el superior. Es decir, no se permite ningún rango de giro y además las ruedas, con respecto a esos ejes tendrán la orientación de 0^{o} y no otra.

El límite para el eje z de la rueda se establece en $0{,}1$ para el límite inferior y en 0 para el límite superior. Como el límite inferior es mayor que el límite superior significa que se permite el giro completamente libre.

En este punto tenemos un coche que se puede desplazar solo por efecto de la gravedad, si lo colocamos en una rampa. O si algún otro objeto colisiona con el y lo empuja.

Motorizando el coche

Para hacer que el coche se mueva por sí mismo y no por la fuerza de la gravedad, solo hay que definir y activar los motores angulares de las ruedas traseras.

```
1 // Se configura un motor angular en la Restricción de la Rueda Rear Right
2 restriccionRR.configureAngularMotor (2, 0.1, 0, velocidad, fuerza)
3
4 // Y se activa cada vez que se quiera acelerar
5 restriccionRR.enableAngularMotor(2)
6
7 // Igual para la Restricción de la Rueda Rear Left
```

Los parámetros en la línea 2 significan lo siguiente.

El primer parámetro es el eje para el cual se está configurando el motor (0 para el eje x, 1 para el eje y, y 2 para el eje z). En este caso el motor que se está configurando es para el movimiento natural de las ruedas, su eje z.

Los dos siguientes parámetros son los límites inferior y superior que se establecen para el giro que provoque ese motor. Igual que antes, como se quiere un giro sin límites se pone el límite inferior mayor que el superior.

Los dos últimos parámetros indican la velocidad máxima angular a la que va a llegar la rueda en su giro (positiva hacia adelante y negativa hacia atrás) y con qué fuerza va a intentar alcanzar esa velocidad máxima. Si la fuerza es grande, el coche podría hacer *caballitos*.

Esos valores de velocidad y fuerza se pueden tomar de una GUI, o de un dispositivo de entrada.

Se puede tener un motor para cada eje x, y, y/o z de la figura que contiene esa restricción. La activación de un motor concreto se realiza indicando el eje que se quiere activar. En el ejemplo, línea 5, se activa el motor del eje 2, que como se acaba de comentar se corresponde con el eje z.

La frenada

Para hacer disminuir la velocidad de un coche en el mundo real se realizan normalmente dos acciones. Levantar el pie del acelerador, con lo que el coche va disminuyendo su velocidad poco a poco debido a diversos factores relacionados con el rozamiento. Pero si no es suficiente, se consigue una mayor deceleración pisando el pedal del freno.

En el coche del ejemplo se realiza lo mismo. En una primera instancia se desactiva el motor que está haciendo girar las ruedas, y en segundo lugar se actúa sobre la velocidad angular de las ruedas para frenarlas.

Para desactivar el motor angular de la restricción de las ruedas hay que indicarle cuál de los 3 motores posibles (eje x, y, o z) se quiere desactivar. Se desactivan los motores en las 2 ruedas motrices.

```
// Cada rueda tiene su propio motor, hay que desactivar ambos
restriccionRR.disableAngularMotor(2);    // El 2 se corresponde con el eje Z
restriccionRL.disableAngularMotor(2);
```

En ese momento el motor deja de empujar pero la inercia del coche hace que siga avanzando y además avanza bastante, como si estuviera en punto muerto.

Para decelerarlo más, se tienen dos opciones, ambas actuando sobre la velocidad angular de las ruedas, como se comentó en los ejemplos de la página 8. Para frenar se actúa sobre las 4 ruedas. Aunque también serviría actuar solamente sobre 2 ruedas del mismo eje, se hace como en un coche real, se frenan las 4 ruedas.

Una opción es modificar la velocidad angular de las ruedas en el `update` quitándole un porcentaje de velocidad en cada frame.

```
// En el método update
var velocidadActual = ruedaRR.getAngularVelocity();
var retencion = 0.02;  // En cada frame la velocidad se reduce en un 2%
velocidadActual.multiplyScalar (1.0 - retencion);
ruedaRR.setAngularVelocity (velocidadActual);
ruedaRL.setAngularVelocity (velocidadActual);
ruedaFR.setAngularVelocity (velocidadActual);
ruedaFL.setAngularVelocity (velocidadActual);
```

Otra opción es realizar un frenazo en seco, estableciendo la velocidad angular en 0.

```
var velocidadCero = new THREE.Vector3 ();
ruedaRR.setAngularVelocity (velocidadCero);
ruedaRL.setAngularVelocity (velocidadCero);
ruedaFR.setAngularVelocity (velocidadCero);
ruedaFL.setAngularVelocity (velocidadCero);
```

A pesar del frenazo de las ruedas en seco, como en el mundo real, el coche tampoco

se detiene inmediatamente. La inercia hace que siga deslizándose sobre la carretera una cierta distancia mayor o menor según el coeficiente de rozamiento de las ruedas con el asfalto.

Ya se tiene un coche al que no se le caen las ruedas, éstas solo giran por su eje, las ruedas traseras son capaces de propulsar el coche, hacia adelante y hacia atrás, se puede frenar el coche actuando sobre sus 4 ruedas.

Ya solo queda poder dirigirlo a derecha e izquierda, actuando sobre las ruedas delanteras, sus ruedas directrices.

Giros de volante

Los giros de volante se consiguen modificando los límites de las restricciones angulares aplicadas a las ruedas delanteras.

Se recuerda que los límites que se habían establecido al definir las restricciones de las ruedas, codigo 1 (pág. 17), se establecía un ángulo 0 en los ejes x e y, y un ángulo sin límites en el eje z.

```
restriccionFR.setAngularLowerLimit({ x: 0, y: 0, z: 0.1 });
restriccionFR.setAngularUpperLimit({ x: 0, y: 0, z: 0 });
// Similar en las otras 3 ruedas
```

Pues para hacer que las ruedas giren un determinado ángulo a derecha o izquierda solo hay que modificar los límites anteriores en las ruedas delanteras.

```
restriccionFR.setAngularLowerLimit({ x: 0, y: angulo, z: 0.1 });
restriccionFR.setAngularUpperLimit({ x: 0, y: angulo, z: 0 });
restriccionFL.setAngularLowerLimit({ x: 0, y: angulo, z: 0.1 });
restriccionFL.setAngularUpperLimit({ x: 0, y: angulo, z: 0 });
```

Se mantiene el ángulo 0 con respecto al eje x, se mantiene la ausencia de límites en los giros con respecto al eje z. Y en el eje y se han puesto los límites inferior y superior al mismo valor. No se les da ningún margen de giro más allá de ese valor, por tanto las ruedas solo pueden tener esa orientación.

Un valor de ángulo 0 conducirían al coche en línea recta, con un valor positivo las ruedas directrices estarían giradas hacia la izquierda y con un valor negativo estarían giradas hacia la derecha.

Cada vez que se modifique el ángulo hay que volver a establecer los límites de la restricción.

Últimos comentarios

Por último, solo decir que se han subido a Prado los fuentes de las dos aplicaciones que se han usado para explicar este tema.

En la aplicación *física* tenéis ejemplos de las restricciones de deslizamiento, bisagra y péndulo. También se han programado listener de colisiones para poner las figuras en modo alambre o en color sólido al colisionar con el suelo o las paredes del escenario.

En esta aplicación, haciendo clic con el ratón sobre las cajas y esferas se le aplica un empujón o un tirón en función del signo de la fuerza que se haya configurado en la GUI.

También incorpora la técnica explicada consistente hacer que las esferas detengan su rodar paulatinamente.

La otra aplicación, *fisica-coche*, implementa el coche que se ha visto al final del tema. El coche se maneja con las teclas del cursor. La aplicación puede verse como un juego consistente en manejar el coche de manera que se vayan empujando esferas y cajas hasta tirarlas por los lados del escenario sin que el coche se caiga al vacío en el intento.