FÍSICAS EN PHASER:

ARCADE:

Diapositivas de clase: https://gjimenezucm.github.io/pvli2122/tema5/arcade.html#/arcade-physics

- Pensado para tratar colisiones AABB (axis-aligned bounded rectangles): para manejar objetos **sin rotaciones**.
- Sólo se comprueba si existen **colisiones** (solapamiento o overlapping) entre dos rectángulos. (Como hacemos en SDL)
- Problemas con las áreas **transparentes**.
- Pensado para juegos **sencillos**.

Configuración del motor arcade:
https://newdocs.phaser.io/docs/3.52.0/Phaser.Types.Physics.Arcade.ArcadeWorldConfig

Usar Arcade:

```
this.player = this.add.sprite(100, 200, 'dude');

this.physics.add.existing(this);
```

o

```
this.player = this.physics.add.sprite(100, 450, 'dude');
```

Así añadimos al sprite la propiedad *body*. (<- enlace para saber todo lo que puedo hacer sobre un body).

Varias propiedades interesantes de body son:

```
this.body.setCollideWorldBounds()
this.body.onFloor() // útil para no saltar infinitamente
this.body.setBounce(1,1) // rebotar al colisionar
this.body.setVelocity(1,0) // mover
```

Pequeño ejemplo de física básica aquí.

Grupos en Arcade

Los **Group** se usan para manejar grupos de colisiones (entre otras cosas). Una entidad creada por un grupo físico tendrá física. Enlace a la documentación sobre groups.

Podemos utilizar add.group() o add.staticGroup():

```
this.platforms = this.physics.add.group();
```

Los grupos creados con `physics.add.group()` son dinámicos, y los grupos creados con `physics.add.staticGroup()` son estáticos (entidades que no se mueven, pero que tienen colisión).

Los creamos sobre physics cuando sean static. Si no son estáticos, se puede hacer this.add.

Crear elementos en un grupo

```
preload -> this.load.image('platform', 'platform.png');

create -> this.platforms = this.physics.add.staticGroup();

          this.platforms.create(500, 150, 'platform');

          this.platforms.create(-200, 300, 'platform');

          this.platforms.create(400, 450, 'platform');
```

También podemos usar `add()` y `addMultiple()` para añadir nuestros propios GO. -> Ver en la api.

Add -> `add(child, [addToScene])` Inserta en el grupo el GO child y, si queremos, en la escena.

AddMultiple hace lo mismo, solo que en vez de pasarle un GO, le paso un array de GO.

Colisiones

Para que se puedan detectar colisiones, el objeto debe tener un collider creado:

```
this.physics.add.collider(player, group);
```

Para que se nos avise cuando haya una colisión, hay que incluir un callback.

```
// create
this.physics.add.collider(player, group, onCollision);
// el método recibe dos parámetros, son los objetos que han colisionado
function onCollision(obj1, obj2) { // hacer algo }
```

También puede hacerse con una función anónima:

```
this.physics.add.collider(player, group, (o1, o2) => { // hacer algo });
```

Otro método que nos puede interesar es `collide()` (¡no add.collider()!), que devuelve un booleano que indica si ha habido colisión:

```
// en update, donde this es una Scene
if(this.physics.collide(this.player, this.platform)) { console.log("Hay colisión"); }
```

Solapamiento

En caso de que queramos saber si hay solapamientos, sin efectos físicos, usaremos *overlap*, en lugar de collider. En create:

```
this.physics.add.overlap(player, group, (o1, o2) => {  // o1 y o2 se están tocando });
```

En update donde this es una Scene:

```
if(this.physics.overlap(this.player, this.platform)) { textInfo.text = "Hay solape"; }
```

Objetos invisibles o triggers

Se pueden crear con add.zone(), de la escena, y luego lo añadimos a las físicas:

```
let trigger = this.add.zone(300, 200, 200, 200); // x, y, width, height
// Añade un body
this.physics.world.enable(trigger);
trigger.body.setAllowGravity(false);
trigger.body.setImmovable(false);
```

MATTER.JS:

- Admite **rotaciones** y formas más complejas (rampas).
- Tiene más **precisión**, pero es más **lento**.
- Tiene un modelo de física mucho más **avanzado**, springs (muelles), polígonos, fuerzas, restricciones…

Uso de Matter:

```
physics: {
        default: 'matter',
        matter: { . . .
```

Con esta configuración, todas las Scene tendrán una propiedad matter a través de la cual podemos acceder al motor físico. update -> `this.matter.add.image(100, -100, 'roca');` donde roca es una imagen añadida en el create.

Para agregar entidades físicas hacemos:

`nave = this.matter.add.image(100, 100, 'nave');` // ¡No sólo se pueden añadir imágenes! -> Doc.

o `this.matter.add.gameObject(nave);` // nave ha sido creado aparte, como una imagen, un sprite...

Matter inyecta propiedades y métodos físicos complejos a este objeto que creamos:

```
nave.setFrictionAir(0.1); //  fricción
nave.setMass(50);  // masa
nave.setFixedRotation(); // inercia infinita
nave.setAngularVelocity(-0.1); // velocidad angular
nave.thrust(0.1); // empuje
nave.setVelocity(16, 0); // x e y
nave.applyForce({15, 24}); // un vector
```

Aplicar fuerzas

```
sprite.applyForce({x: 100, y: 0});
```

Hay funciones más cómodas, que tienen en cuenta la orientación del GameObject:

```
sprite.thrust(19);
sprite.thrustLeft(14);
sprite.thrustBack(5);
```

Aplicar impulsos

Un impulso es una fuerza que se ejerce en un momento puntual, pero que para y su efecto (la aceleración) sólo ocurre por un instante. Para hacer esto con Matter, vamos a usar `applyForce()` durante sólo un instante.

Gravedad

Es una fuerza constante hacia una dirección (hacia abajo por lo general). Esta se define en la config inicial cuando añadimos el motor de físicas.

Un objeto puede tener una bounding box más elaborada que influya y afecte a la gravedad:

```
let rect = this.matter.add.image(200, 50, 'blue');

rect.setBody({ type: 'rectangle',

               width: 128,

               height: 128 });
```

Formas

Cada forma tiene unas propiedades particulares (por ejemplo, 'fromVertices' tiene una cadena verts="x y x y ... x y" o un array). Podemos verlas aquí. Algunas de estas formas son: 'rectangle', 'circle', 'trapezoid', 'polygon', 'fromVertices'…

Detección de colisiones

Para detectar colisiones y solapamientos, utilizamos eventos.

```
        // cuando se inicia la colisión

this.matter.world.on('collisionstart',

    (evento, cuerpo1, cuerpo2) => {// hacer algo });

        // cuando termina la colisión

this.matter.world.on('collisionend',

    (evento, cuerpo1, cuerpo2) => {// hacer algo });
```

Restricciones / Constraints

Un constraint es una restricción sobre la distancia a la que tienen que estar dos objetos. (a.k.a. Muelles / springs). Siguen la Ley de Hooke, $F = -k * X$, donde k es la *stiffness* (rigidez) y X la longitud.