Motores de físicas para videojuegos

Índice

Un tipo de juegos que ha tenido una gran proliferación en el mercado de aplicaciones para móviles son aquellos juegos basados en físicas. Estos juegos son aquellos en los que el motor realiza una simulación física de los objetos en pantalla, siguiendo las leyes de la cinemática y la dinámica. Es decir, los objetos de la pantalla están sujetos a gravedad, cada uno de ellos tiene una masa, y cuando se produce una colisión entre ellos se produce una fuerza que dependerá de su velocidad y su masa. El motor físico se encarga de realizar toda esta simulación, y nosotros sólo deberemos encargarnos de proporcionar las propiedades de los objetos en pantalla. Uno de los motores físicos más utilizados es Box2D, originalmente implementado en C++. Se ha utilizado para implementar juegos tan conocidos y exitosos como Angry Birds. Podemos encontrar ports de este motor para las distintas plataformas móviles. Tanto libgdx como Cocos2D incluyen una implementación del mismo.

Angry Birds, implementado con Box2D

Antes de comenzar a estudiar el motor Box2D, vamos a repasar los fundamentos de la librería libgdx para Android. De esta forma podremos crear juegos que utilicen físicas tanto para iOS como para Android, utilizando Cocos2D y libgdx respectivamente. El motor de físicas es idéntico en ambos casos, tiene la misma API, con la salvedad de que en libgdx utilizamos una implementación Java de la misma, en lugar de la implementación C++ original. Utilizaremos en los ejemplos la implementación Java de Box2D incluida en libgdx, pero sería inmediado trasladar el código a C++ para utilizarlo dentro de Cocos2D.

1. Juegos en Android con libgdx

El motor libgdx cuenta con la ventaja de que soporta tanto la plataforma Android como la plataforma Java SE. Esto significa que los juegos que desarrollemos con este motor se podrán ejecutar tanto en un ordenador con máquina virtual Java, como en un móvil Android. Esto supone una ventaja importante a la hora de probar y depurar el juego, ya que el emulador de Android resulta demasiado lento como para poder probar un videojuego en condiciones. El poder ejecutar el juego como aplicación de escritorio nos permitirá probar el juego sin necesidad del emulador, aunque siempre será imprescindible hacer también prueba en un móvil real ya que el comportamiento del dispositivo puede

diferir mucho del que tenemos en el ordenador con Java SE.

1.1. Estructura del proyecto libgdx

Para conseguir un juego multiplataforma, podemos dividir la implementación en dos proyectos:

- **Proyecto Java genérico.** Contiene el código Java del juego utilizando libgdx. Podemos incluir una clase principal Java (con un método `main`) que nos permita ejecutar el juego en modo escritorio.
- **Proyecto Android.** Dependerá del proyecto anterior. Contendrá únicamente la actividad principal cuyo cometido será mostrar el contenido del juego utilizando las clases del proyecto del que depende.

El primer proyecto se creará como proyecto Java, mientras que el segundo se creará como proyecto Android que soporte como SDK mínima la versión 1.5 (API de nivel 3). En ambos proyectos crearemos un directorio `libs` en el que copiaremos todo el contenido de la librería libgdx, pero no será necesario añadir todas las librerías al *build path*.

En el caso del proyecto Java, añadiremos al *build path* las librerías:

- `gdx-backend-jogl-natives.jar`
- `gdx-backend-jogl.jar`
- `gdx-natives.jar`
- `gdx.jar`

En el caso de la aplicación Android añadiremos al *build path*:

- `gdx-backend-android.jar`
- `gdx.jar`
- Proyecto Java. Añadimos el proyecto anterior como dependencia al *build path* para tener acceso a todas sus clases.

Tenemos que editar también el `AndroidManifest.xml` para que su actividad principal soporte los siguientes cambios de configuración:

```
android:configChanges="keyboard|keyboardHidden|orientation"
```

En el proyecto Java crearemos la clase principal del juego. Esta clase deberá implementar la interfaz `ApplicationListener` y definirá los siguientes métodos:

```
public class MiJuego implements ApplicationListener {

    @Override
    public void create() {
    }

    @Override
    public void pause() {
    }

    @Override
```

```
    public void resume() {
    }

    @Override
    public void dispose() {
    }

    @Override
    public void resize(int width, int height) {
    }

    @Override
    public void render() {
    }

}
```

Este será el punto de entrada de nuestro juego. A continuación veremos con detalle cómo implementar esta clase. Ahora vamos a ver cómo terminar de configurar el proyecto.

Una vez definida la clase principal del juego, podemos modificar la actividad de Android para que ejecute dicha clase. Para hacer esto, haremos que en lugar de heredar de `Activity` herede de `AndroidApplication`, y dentro de `onCreate` instanciaremos la clase principal del juego definida anteriormente, y llamaremos a `initialice` proporcinando dicha instancia:

```
public class MiJuegoAndroid extends AndroidApplication {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        initialize(new MiJuego(), false);
    }
}
```

Con esto se pondrá en marcha el juego dentro de la actividad Android. Podemos también crearnos un programa principal que ejecute el juego en modo escritorio. Esto podemos hacerlo en el proyecto Java. En este caso debemos implementar el método `main` de la aplicación Java *standalone*, y dentro de ella instanciar la clase principal de nuestro juego y mostrarla en un objeto `JoglApplication` (Aplicación OpenGL Java). En este caso deberemos indicar también el título de la ventana donde se va a mostrar, y sus dimensiones:

```
public class MiJuegoDesktop {

    public static void main(String[] args) {
        new JoglApplication(new MiJuego(), "Ejemplo Especialista",
                            480, 320, false);
    }

}
```

Con esto hemos terminado de configurar el proyecto. Ahora podemos centrarnos en el código del juego dentro del proyecto Java. Ya no necesitaremos modificar el proyecto Android, salvo para añadir *assets*, ya que estos *assets* deberán estar replicados en ambos proyectos para que pueda localizarlos de forma correcta tanto la aplicación Android como Java.

1.2. Ciclo del juego

Hemos visto que nuestra actividad principal de Android, en lugar de heredar de `Activity`, como se suele hacer normalmente, hereda de `AndroidApplication`. Este tipo de actividad de la librería libgdx se encargará, entre otras cosas, de inicializar el contexto gráfico, por lo que no tendremos que realizar la inicialización de OpenGL manualmente, ni tendremos que crear una vista de tipo `SurfaceView` ya que todo esto vendrá resuelto por la librería.

Simplemente deberemos proporcionar una clase creada por nosotros que implemente la interfaz `ApplicationListener`. Dicha interfaz nos obligará a definir un método `render` (entre otros) que se invocará en cada *tick* del ciclo del juego. Dentro de él deberemos realizar la actualización y el renderizado de la escena.

Es decir, libgdx se encarga de gestionar la vista OpenGL (`GLSurfaceView`) y dentro de ella el ciclo del juego, y nosotros simplemente deberemos definir un método `render` que se encargue de actualizar y dibujar la escena en cada iteración de dicho ciclo.

Además podemos observar en `ApplicationListener` otros métodos que controlan el ciclo de vida de la aplicación: `create`, `pause`, `resume` y `dispose`. Por ejemplo en `create` deberemos inicializar todos los recursos necesarios para el juego, y el `dispose` liberaremos la memoria de todos los recursos que lo requieran.

De forma alternativa, en lugar de implementar `ApplicationListener` podemos heredar de `Game`. Esta clase implementa la interfaz anterior, y delega en objetos de tipo `Screen` para controlar el ciclo del juego. De esta forma podríamos separar los distintos estados del juego (pantallas) en diferentes clases que implementen la interfaz `Screen`. Al inicializar el juego mostraríamos la pantalla inicial:

```
public class MiJuego extends Game {

    @Override
    public void create() {
        this.setScreen(new MenuScreen(this));
    }
}
```

Cada vez que necesitemos cambiar de estado (de pantalla) llamaremos al método `setScreen` del objeto `Game`.

La interfaz `Screen` nos obliga a definir un conjunto de métodos similar al de `ApplicationListener`:

```
public class MenuScreen implements Screen {
    Game game;

    public MenuScreen(Game game) {
        this.game = game;
    }

    public void show() { }
```

```
    public void pause() { }
    public void resume() { }
    public void hide() { }
    public void dispose() { }
    public resize(int width, int height) { }
    public render(float delta) { }
}
```

1.3. Módulos de libgdx

En libgdx encontramos diferentes módulos accesibles como miembros estáticos de la clase `Gdx`. Estos módulos son:

- `graphics`: Acceso al contexto gráfico de OpenGL y utilidades para dibujar gráficos en dicho contexto.
- `audio`: Reproducción de música y efectos de sonido (WAV, MP3 y OGG).
- `input`: Entrada del usuario (pantalla táctil y acelerómetro).
- `files`: Acceso a los recursos de la aplicación (*assets*).

1.4. Gráficos con libgdx

Dentro del método `render` podremos acceder al contexto gráfico de OpenGL mediante la propiedad `Gdx.graphics`.

Del contexto gráfico podemos obtener el contexto OpenGL. Por ejemplo podemos vaciar el fondo de la pantalla con:

```
int width = Gdx.graphics.getWidth();
int height = Gdx.graphics.getHeight();

GL10 gl = Gdx.app.getGraphics().getGL10();
gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
gl.glViewport(0, 0, width, height);
```

Podemos utilizar además las siguientes clases de la librería como ayuda para dibujar gráficos:

- `Texture`: Define una textura 2D, normalmente cargada de un fichero (podemos utilizar `Gdx.files.getFileHandle` para acceder a los recursos de la aplicación, que estarán ubicados en el directorio `assets` del proyecto). Sus dimensiones (alto y ancho) deben ser una potencia de 2. Cuando no se vaya a utilizar más, deberemos liberar la memoria que ocupa llamando a su método `dispose` (esto es así en en todos los objetos de la librería que representan recursos que ocupan un espacio en memoria).
- `TextureAtlas`: Se trata de una textura igual que en el caso anterior, pero que además incluye información sobre distintas regiones que contiene. Cuando tenemos diferentes items para mostrar (por ejemplo diferentes fotogramas de un *sprite*), será conveniente empaquetarlos dentro de una misma textura para aprovechar al máximo la memoria. Esta clase incluye información del área que ocupa cada item, y nos permite obtener por separado diferentes regiones de la imagen. Esta clase lee el formato generado por

la herramienta *TexturePacker*.

- `TextureRegion`: Define una región dentro de una textura que tenemos cargada en memoria. Estos son los elementos que obtenemos de un *atlas*, y que podemos dibujar de forma independiente.
- `Sprite`: Es como una región, pero además incluye información sobre su posición en pantalla y su orientación.
- `BitmapFont`: Representa una fuente de tipo *bitmap*. Lee el formato *BMFont* (`.fnt`), que podemos generar con la herramienta *Hiero bitmap font tool*.
- `SpriteBatch`: Cuando vayamos a dibujar varios *sprites* 2D y texto, deberemos dibujarlos todos dentro de un mismo *batch*. Esto hará que todas las caras necesarias se dibujen en una sola operación, lo cual mejorará la eficiencia de nuestra aplicación. Deberemos llamar a la operación `begin` del *batch* cuando vayamos a empezar a dibujar, y a `end` cuando hayamos finalizado. Entre estas dos operaciones, podremos llamar varias veces a sus métodos `draw` para dibujar diferentes texturas, regiones de textura, *sprites* o cadenas de texto utilizando fuentes *bitmap*.
- `TiledMap`, `TileAtlas` y `TileLoader`: Nos permiten crear un mosaico para el fondo, y así poder tener fondos extensos. Soporta el formato TMX.

1.4.1. Sprites

Por ejemplo, podemos crear *sprites* a partir de una región de un *sprite sheet* (o *atlas*) de la siguiente forma:

```
TextureAtlas atlas = new TextureAtlas(Gdx.files.getFileHandle("sheet",
                                      FileType.Internal));
TextureRegion regionPersonaje = atlas.findRegion("frame01");
TextureRegion regionEnemigo = atlas.findRegion("enemigo");

Sprite spritePersonaje = new Sprite(regionPersonaje);
Sprite spriteEnemigo = new Sprite(regionEnemigo);
```

Donde `"frame01"` y `"enemigo"` son los nombres que tienen las regiones dentro del fichero de regiones de textura. Podemos dibujar estos *sprites* utilizando un *batch* dentro del método `render`. Para ello, será recomendable instanciar el `batch` al crear el juego (`create`), y liberarlo al destruirlo (`dispose`). También deberemos liberar el *atlas* cuando no lo necesitemos utilizar, ya que es el objeto que representa la textura en la memoria de vídeo:

```
public class MiJuego implements ApplicationListener {

    SpriteBatch batch;

    TextureAtlas atlas;

    Sprite spritePersonaje;
    Sprite spriteEnemigo;

    @Override
    public void create() {
        atlas = new TextureAtlas(Gdx.files.getFileHandle("sheet",
                                 FileType.Internal));
        TextureRegion regionPersonaje = atlas.findRegion("frame01");
```

```
        TextureRegion regionEnemigo = atlas.findRegion("enemigo");

        spritePersonaje = new Sprite(regionPersonaje);
        spriteEnemigo = new Sprite(regionEnemigo);

        batch = new SpriteBatch();
    }

    @Override
    public void dispose() {
        batch.dispose();
        atlas.dispose();
    }

    @Override
    public void render() {
        batch.begin();
        spritePersonaje.draw(batch);
        spriteEnemigo.draw(batch);
        batch.end();
    }

}
```

Cuando dibujemos en el *batch* deberemos intentar dibujar siempre de forma consecutiva los *sprites* que utilicen la misma textura. Si dibujamos un *sprite* con diferente textura provocaremos que se envíe a la GPU toda la geometría almacenada hasta el momento para la anterior textura.

1.4.2. Animaciones y delta time

Podemos también definir los fotogramas de la animación con un objeto `Animation`:

```
Animation animacion = new Animation(0.25f,
    atlas.findRegion("frame01"),
    atlas.findRegion("frame02"),
    atlas.findRegion("frame03"),
    atlas.findRegion("frame04"));
```

Como primer parámetro indicamos la periodicidad, y a continuación las regiones de textura que forman la animación. En este caso no tendremos ningún mecanismo para que la animación se ejecute de forma automática, tendremos que hacerlo de forma manual con ayuda del objeto anterior proporcionando el número de segundos transcurridos desde el inicio de la animación

```
spritePersonaje.setRegion(animacion.getKeyFrame(tiempo, true));
```

Podemos obtener este tiempo a partir del tiempo transcurrido desde la anterior iteración (*delta time*). Podemos obtener este valor a partir del módulo de gráficos:

```
tiempo += Gdx.app.getGraphics().getDeltaTime();
```

La variable tiempo anterior puede ser inicializada a `0` en el momento en el que comienza la animación. El *delta time* será muy útil para cualquier animación, para saber cuánto debemos avanzar en función del tiempo transcurrido.

1.4.3. Fondos

Podemos crear fondos basados en mosaicos con las clases `TiledMap`, `TileAtlas` y `TileLoader`.

```
TiledMap fondoMap = TiledLoader.createMap(
      Gdx.files.getFileHandle("fondo.tmx",
      FileType.Internal));

TileAtlas fondoAtlas = new TileAtlas(fondoMap,
      Gdx.files.getFileHandle(".", FileType.Internal));
```

Al crear el *atlas* se debe proporcionar el directorio en el que están los ficheros que componen el mapa (las imágenes). Es importante recordar que el *atlas* representa la textura en memoria, y cuando ya no vaya a ser utilizada deberemos liberar su memoria con `dispose()`.

Podemos dibujar el mapa en pantalla con la clase `TileMapRenderer`. Este objeto se deberá inicializar al crear el juego de la siguiente forma, proporcionando las dimensiones de cada *tile*:

```
tileRenderer = new TiledMapRenderer(fondoMap, fondoAtlas, 40, 40);
```

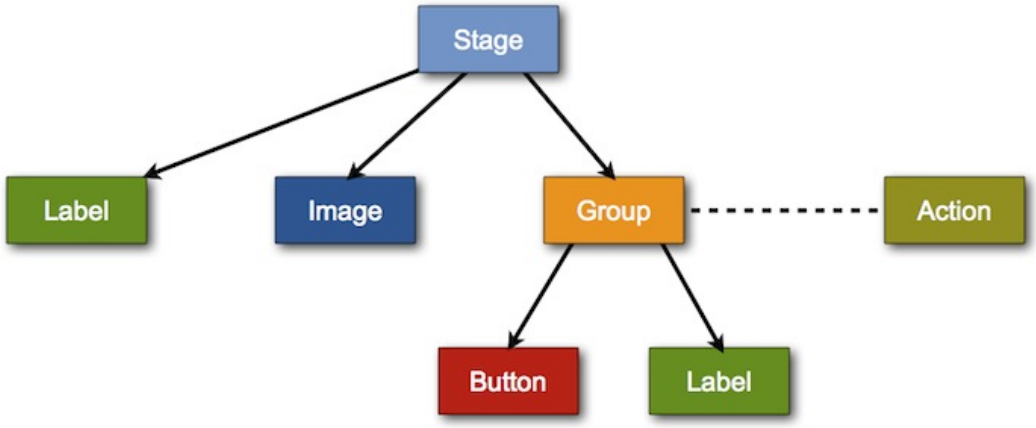
Dentro de `render`, podremos dibujarlo en pantalla con:

```
tileRenderer.render();
```

Cuando no vaya a ser utilizado, lo liberaremos con `dispose()`.

1.4.4. Escena 2D

En libgdx tenemos también una API para crear un grafo de la escena 2D, de forma similar a Cocos2D. Sin embargo, en este caso esta API está limitada a la creación de la interfaz de usuario (etiquetas, botones, etc). Será útil para crear los menús, pero no para el propio juego.



Grafo de la escena 2D en libgdx

El elemento principal de esta API es `Stage`, que representa el escenario al que añadiremos los distintos actores (nodos). Podemos crear un escenario con:

```
stage = new Stage(width, height, false);
```

Podremos añadir diferentes actores al escenario, como por ejemplo una etiqueta de texto:

```
Label label = new Label("gameover", fuente, "Game Over");
stage.addActor(label);
```

También podemos añadir acciones a los actores de la escena:

```
FadeIn fadeIn = FadeIn.$(1);
FadeOut fadeOut = FadeOut.$(1);
Delay delay = Delay.$(fadeOut, 1);
Sequence seq = Sequence.$(fadeIn, delay);
Forever forever = Forever.$(seq);
label.action(forever);
```

Para que la escena se muestra y ejecute las acciones, deberemos programarlo de forma manual en `render`:

```
@Override
public void render() {
    stage.act(Gdx.app.getGraphics().getDeltaTime());
    stage.draw();
}
```

1.5. Entrada en libgdx

La librería *libgdx* simplifica el acceso a los datos de entrada, proporcionándonos en la propiedad `Gdx.input` toda la información que necesitaremos en la mayoría de los casos sobre el estado de los dispositivos de entrada. De esta forma podremos acceder a estos datos de forma síncrona dentro del ciclo del juego, sin tener que definir *listeners* independientes.

A continuación veremos los métodos que nos proporciona este objeto para acceder a los diferentes dispositivos de entrada.

1.5.1. Pantalla táctil

Para saber si se está pulsando actualmente la pantalla táctil tenemos el método `isTouched`. Si queremos saber si la pantalla acaba de tocarse en este momento (es decir, que en la iteración anterior no hubiese ninguna pulsación y ahora si) podremos utilizar el método `justTouched`.

En caso de que haya alguna pulsación, podremos leerla con los métodos `getX` y `getY`. Deberemos llevar cuidado con este último, ya que nos proporciona la información en coordenadas de Android, en las que la *y* es positiva hacia abajo, y tiene su origen en la parte superior de la pantalla, mientras que las coordenadas que utilizamos en *libgdx* tiene

el origen de la coordenada *y* en la parte inferior y son positivas hacia arriba.

```
public void render() {
    if(Gdx.input.isTouched()) {
        int x = Gdx.input.getX()
        int y = height - Gdx.input.getY();

        // Se acaba de pulsar en (x,y)
        ...
    }
    ...
}
```

Para tratar las pantallas multitáctiles, los métodos `isTouched`, `getX`, y `getY` pueden tomar un índice como parámetro, que indica el puntero que queremos leer. Los índices son los identificadores de cada contacto. El primer contacto tendrá índice `0`. Si en ese momento ponemos un segundo dedo sobre la pantalla, a ese segundo contacto se le asignará el índice `1`. Ahora, si levantamos el primer contacto, dejando el segundo en la pantalla, el segundo seguirá ocupando el índice `1`, y el índice `0` quedará vacío.

Si queremos programar la entrada mediante eventos, tal como se hace normalmente en Android, podemos implementar la interfaz `InputProcessor`, y registrar dicho objeto mediante el método `setInputProcessor` de la propiedad `Gdx.input`.

1.5.2. Posición y aceleración

Podemos detectar si tenemos disponible un acelerómetro llamando a `isAccelerometerAvailable`. En caso de contar con él, podremos leer los valores de aceleración en *x*, *y*, y *z* con los metodos `getAccelerometerX`, `getAccelerometerY`, y `getAccelerometerZ` respectivamente.

También podemos acceder a la información de orientación con `getAzimuth`, `getPitch`, y `getRoll`.

2. Motor de físicas Box2D

Vamos ahora a estudiar el motor de físicas Box2D. Es importante destacar que este motor sólo se encargará de simular la física de los objetos, no de dibujarlos. Será nuestra responsabilidad mostrar los objetos en la escena de forma adecuada según los datos obtenidos de la simulación física. Comenzaremos viendo los principales componentes de esta librería.

2.1. Componentes de Box2D

Los componentes básicos que nos permiten realizar la simulación física con Box2D son:

- `Body`: Representa un cuerpo rígido. Estos son los tipos de objetos que tendremos en el mundo 2D simulado. Cada cuerpo tendrá una posición y velocidad. Los cuerpos se

verán afectados por la gravedad del mundo, y por la interacción con los otros cuerpos. Cada cuerpo tendrá una serie de propiedades físicas, como su masa o su centro de gravedad.

- `Fixture`: Es el objeto que se encarga de fijar las propiedades de un cuerpo, como por ejemplo su forma, coeficiente de rozamiento o densidad.
- `Shape`: Sirve para especificar la forma de un cuerpo. Hay distintos tipos de formas (subclases de `Shape`), como por ejemplo `CircleShape` y `PolygonShape`, para crear cuerpos con formar circulares o poligonales respectivamente.
- `Constraint`: Nos permite limitar la libertad de un cuerpo. Por ejemplo podemos utilizar una restricción que impida que el cuerpo pueda rotar, o para que se mueva siguiendo sólo una línea (por ejemplo un objeto montado en un rail).
- `Joint`: Nos permite definir uniones entre diferentes cuerpos.
- `World`: Representa el mundo 2D en el que tendrá lugar la simulación. Podemos añadir una serie de cuerpos al mundo. Una de las principales propiedades del mundo es la gravedad.

Lo primero que deberemos hacer es crear el mundo en el que se realizará la simulación física. Como parámetro deberemos proporcionar un vector 2D con la gravedad del mundo:

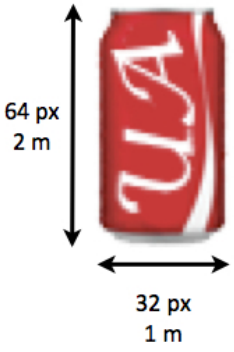
```
World world = new World(new Vector2(0, -10), true);
```

2.2. Unidades de medida

Antes de crear cuerpos en el mundo, debemos entender el sistema de coordenadas de Box2D y sus unidades de medida. Los objetos de Box2D se miden en metros, y la librería está optimizada para objetos de 1m, por lo que deberemos hacer que los objetos que aparezcan con más frecuencia tengan esta medida.

Sin embargo, los gráficos en pantalla se miden en píxeles (o puntos). Deberemos por lo tanto fijar el ratio de conversión entre pixeles y metros. Por ejemplo, si los objetos con los que trabajamos normalmente miden 32 pixeles, haremos que 32 pixeles equivalgan a un metro. Definimos el siguiente ratio de conversión:

```
public final static float PTM_RATIO = 32;
```



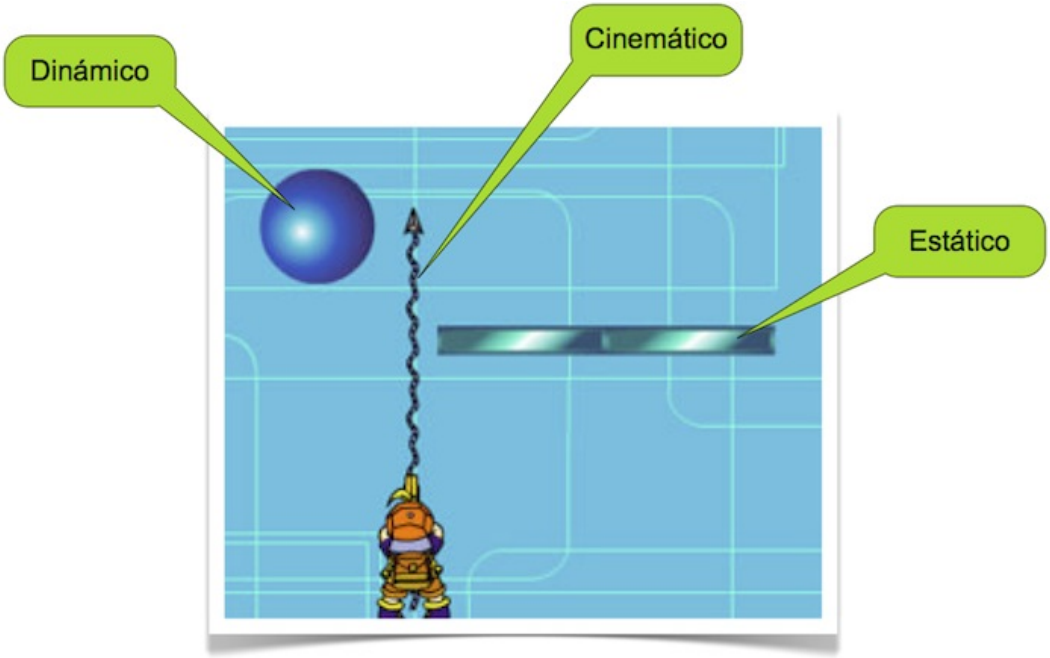
Métricas de Box2D

Para todas las unidades de medida Box2D utiliza el sistema métrico. Por ejemplo, para la masa de los objetos utiliza Kg.

2.3. Tipos de cuerpos

Encontramos tres tipos diferentes de cuerpos en Box2D según la forma en la que queremos que se realice la simulación con ellos:

- **Dinámicos**: Están sometidos a las leyes físicas, y tienen una masa concreta y finita. Estos cuerpos se ven afectados por la gravedad y por la interacción con los demás cuerpos.
- **Estáticos**: Son cuerpos que permanecen siempre en la misma posición. Equivalen a cuerpos con masa infinita. Por ejemplo, podemos hacer que el escenario sea estático.
- **Cinemáticos**: Al igual que los cuerpos estáticos tienen masa infinita y no se ven afectados por otros cuerpos ni por la gravedad. Sin embargo, en esta caso no tienen una posición fija, sino que tienen una velocidad constante. Nos son útiles por ejemplo para proyectiles.



Tipos de cuerpos en Box2D

2.4. Creación de cuerpos

Con todo lo visto anteriormente ya podemos crear distintos cuerpos. Para crear un cuerpo primero debemos crear un objeto de tipo `BodyDef` con las propiedades del cuerpo a crear,

como por ejemplo su posición en el mundo, su velocidad, o su tipo. Una vez hecho esto, crearemos el cuerpo a partir del mundo (`World`) y de la definición del cuerpo que acabamos de crear. Una vez creado el cuerpo, podremos asignarle una forma y densidad mediante *fixtures*. Por ejemplo, en el siguiente caso creamos un cuerpo dinámico con forma rectangular:

```
BodyDef bodyDef = new BodyDef();
bodyDef.type = BodyType.DynamicBody;
bodyDef.position.x = x / PTM_RATIO;
bodyDef.position.y = y / PTM_RATIO;

Body body = world.createBody(bodyDef);

PolygonShape bodyShape = new PolygonShape();
bodyShape.setAsBox((width/2) / PTM_RATIO, (height/2) / PTM_RATIO);
body.createFixture(bodyShape, 1.0f);
bodyShape.dispose();
```

Podemos también crear un cuerpo de forma circular con:

```
BodyDef bodyDef = new BodyDef();
bodyDef.type = BodyType.DynamicBody;
bodyDef.position.x = x / PTM_RATIO;
bodyDef.position.y = y / PTM_RATIO;

Body body = world.createBody(bodyDef);

Shape bodyShape = new CircleShape();
bodyShape.setRadius(radius / PTM_RATIO);
Fixture bodyFixture = body.createFixture(bodyShape, 1.0f);
bodyShape.dispose();
```

También podemos crear los límites del escenario mediante cuerpos de tipo estático y con forma de arista (*edge*):

```
BodyDef limitesBodyDef = new BodyDef();
limitesBodyDef.position.x = x;
limitesBodyDef.position.y = y;

Body limitesBody = world.createBody(limitesBodyDef);
EdgeShape limitesShape = new EdgeShape();
limitesShape.set(new Vector2(0.0f / PTM_RATIO, 0.0f / PTM_RATIO),
                 new Vector2(width / PTM_RATIO, 0.0f / PTM_RATIO));
limitesBody.createFixture(limitesShape,0).setFriction(2.0f);

limitesShape.set(new Vector2(width / PTM_RATIO, 0.0f / PTM_RATIO),
                 new Vector2(width / PTM_RATIO, height / PTM_RATIO));
limitesBody.createFixture(limitesShape,0);

limitesShape.set(new Vector2(width / PTM_RATIO, height / PTM_RATIO),
                 new Vector2(0.0f / PTM_RATIO, height / PTM_RATIO));
limitesBody.createFixture(limitesShape,0);

limitesShape.set(new Vector2(0.0f / PTM_RATIO, height / PTM_RATIO),
                 new Vector2(0.0f / PTM_RATIO, 0.0f / PTM_RATIO));
limitesBody.createFixture(limitesShape,0);
```

Los cuerpos tienen una propiedad `userData` que nos permite vincular cualquier objeto con el cuerpo. Por ejemplo, podríamos vincular a un cuerpo físico el `Sprite` que queremos utilizar para mostrarlo en pantalla:

```
body.setUserData(sprite);
```

De esta forma, cuando realicemos la simulación podemos obtener el *sprite* vinculado al cuerpo físico y mostrarlo en pantalla en la posición que corresponda.

2.5. Simulación

Ya hemos visto cómo crear el mundo 2D y los cuerpos rígidos. Vamos a ver ahora cómo realizar la simulación física dentro de este mundo. Para realizar la simulación deberemos llamar al método `step` sobre el mundo, proporcionando el *delta time* transcurrido desde la última actualización del mismo:

```
world.step(delta, 6, 2);
world.clearForces();
```

Además, los algoritmos de simulación física son iterativos. Cuantas más iteraciones se realicen mayor precisión se obtendrá en los resultados, pero mayor coste tendrán. El segundo y el tercer parámetro de `step` nos permiten establecer el número de veces que debe iterar el algoritmo para resolver la posición y la velocidad de los cuerpos respectivamente. Tras hacer la simulación, deberemos limpiar las fuerzas acumuladas sobre los objetos, para que no se arrastren estos resultados a próximas simulaciones.

Tras hacer la simulación deberemos actualizar las posiciones de los *sprites* en pantalla y mostrarlos. Por ejemplo, si hemos vinculado el `Sprite` al cuerpo mediante la propiedad `userData`, podemos recuperarlo y actualizarlo de la siguiente forma:

```
Sprite sprite = (Sprite)body.getUserData();
Vector2 pos = body.getPosition();
float rot = (float)Math.toDegrees(body.getAngle());

sprite.setPosition((int)(pos.x * PTM_RATIO), (int)(pos.y * PTM_RATIO));
sprite.setRotation(rot);

batch.begin();
sprite.draw(batch);
batch.end();
```

2.6. Detección de colisiones

Hemos comentado que dentro de la simulación física existen interacciones entre los diferentes objetos del mundo. Podemos recibir notificaciones cada vez que se produzca un contacto entre objetos, para así por ejemplo aumentar el daño recibido.

Podremos recibir notificaciones mediante un objeto que implemente la interfaz `ContactListener`. Esta interfaz nos forzará a definir los siguientes métodos:

```
@Override
public void beginContact(Contact c) {
    // Se produce un contacto entre dos cuerpos
}

@Override
```

```
public void endContact(Contact c) {
    // El contacto entre los cuerpos ha finalizado
}

@Override
public void preSolve(Contact c, Manifold m) {
    // Se ejecuta antes de resolver el contacto.
    // Podemos evitar que se procese
}

@Override
public void postSolve(Contact c, ContactImpulse ci) {
    // Podemos obtener el impulso aplicado sobre los cuerpos en contacto
}
```

Podemos obtener los cuerpos implicados en el contacto a partir del parámetro `Contact`. También podemos obtener información sobre los puntos de contacto mediante la información proporcionada por `WorldManifold`:

```
public void beginContact(Contact c) {
    Body bodyA = c.getFixtureA().getBody();
    Body bodyB = c.getFixtureB().getBody();

    // Obtiene el punto de contacto
    Vector2 point = c.getWorldManifold().getPoints()[0];

    // Calcula la velocidad a la que se produce el impacto
    Vector2 velA = bodyA.getLinearVelocityFromWorldPoint(point);
    Vector2 velB = bodyB.getLinearVelocityFromWorldPoint(point);

    float vel = c.getWorldManifold().getNormal().dot(velA.sub(velB));

    ...
}
```

De esta forma, además de detectar colisiones podemos también saber la velocidad a la que han chocado, para así poder aplicar un diferente nivel de daño según la fuerza del impacto.

También podemos utilizar `postSolve` para obtener el impulso ejercido sobre los cuerpos en contacto en cada instante:

```
public void postSolve(Contact c, ContactImpulse ci) {

    Body bodyA = c.getFixtureA().getBody();
    Body bodyB = c.getFixtureB().getBody();

    float impulso = ci.getNormalImpulses()[0];
}
```

Debemos tener en cuenta que `beginContact` sólo será llamado una vez, al comienzo del contacto, mientras que `postSolve` nos informa en cada iteración de las fuerzas ejercidas entre los cuerpos en contacto.