3.4. Simulación Física

En prácticamente cualquier videojuego (tanto 2D como 3D) es necesaria la detección de colisiones y, en muchos casos, la simulación realista de dinámica de cuerpo rígido. En esta sección estudiaremos la relación existente entre sistemas de detección de colisiones y sistemas de simulación física, y veremos algunos ejemplos de uso del motor de simulación física *Physaxe*.

La mayoría de los videojuegos requieren en mayor o menor medida el uso de técnicas de detección de colisiones y simulación física. Desde un videojuego clásico como *Arkanoid*, hasta modernos juegos automovilísticos como *Gran Turismo* requieren definir la interacción de los elementos en el mundo físico.

Definimos un **cuerpo rígido** como un objeto sólido ideal, infinitamente duro y no deformable.

El motor de *simulación física* puede abarcar una amplia gama de características y funcionalidades, aunque la mayor parte de las veces el término se refiere a un tipo concreto de simulación de la **dinámica de cuerpos rígidos**. Esta dinámica se encarga de determinar el movimiento de estos cuerpos rígidos y su interacción ante la influencia de fuerzas.

En el mundo real, los objetos no pueden pasar a través de otros objetos [21]. En nuestro videojuego, a menos que tengamos en cuenta las colisiones de los cuerpos, tendremos el mismo efecto. El *sistema de detección de colisiones*, que habitualmente es un módulo del motor de simulación física, se encarga de calcular estas relaciones, determinando la relación espacial existente entre cuerpos rígidos.

La mayor parte de los videojuegos actuales incorporan ciertos elementos de simulación física básicos. Algunos títulos se animan a incorporar ciertos elementos complejos como simulación de telas, cuerdas, pelo o fluidos. Algunos elementos de simulación física son precalculados y almacenados como animaciones estáticas (y pueden desplegarse en videojuegos 2D como sprites animados), mientras que otros necesitan ser calculados en tiempo real para conseguir una integración adecuada.

Como hemos indicado anteriormente, las tres tareas principales que deben estar soportadas por un motor de simulación física son la detección de colisiones, su resolución (junto con otras restricciones de los

[21] salvo casos específicos convenientemente documentados en la revista *Más Allá*

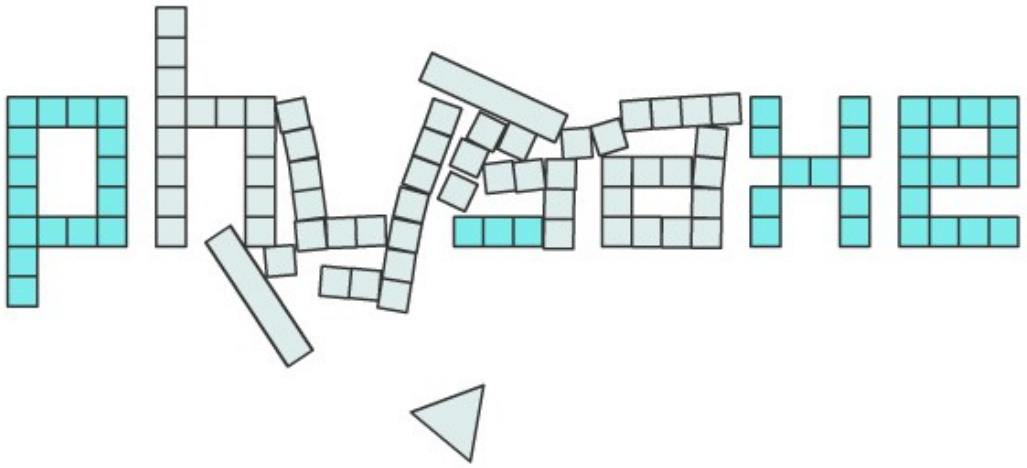

Figura 3.18: Ejemplo de mundo físico dinámico creado con Physaxe. El logotipo de la biblioteca se destruye mediante diversas formas convexas dinámicas.

objetos) y calcular la actualización del mundo tras esas interacciones. De forma general, las características que suelen estar presentes en motores de simulación física son:

- Detección de colisiones entre objetos dinámicos de la escena. Esta detección podrá ser utilizada posteriormente por el módulo de simulación dinámica.
- Cálculo de lineas de visión y tiro parabólico, para la simulación del lanzamiento de proyectiles en el juego.
- Definición de geometría estática de la escena (cuerpos de colisión) que formen el escenario del videojuego. Este tipo de geometría puede ser más compleja que la geometría de cuerpos dinámicos.
- Especificación de fuerzas (tales como viento, rozamiento, gravedad, etc...), que añadirán realismo al videjuego.
- Simulación de destrucción de objetos: paredes y objetos del escenario.
- Definición de diversos tipos de articulaciones, tanto en elementos del escenario (bisagras en puertas, raíles...) como en la descripción de las articulaciones de personajes.
- Especificación de diversos tipos de motores y elementos generadores de fuerzas, así como simulación de elementos de suspensión y muelles.

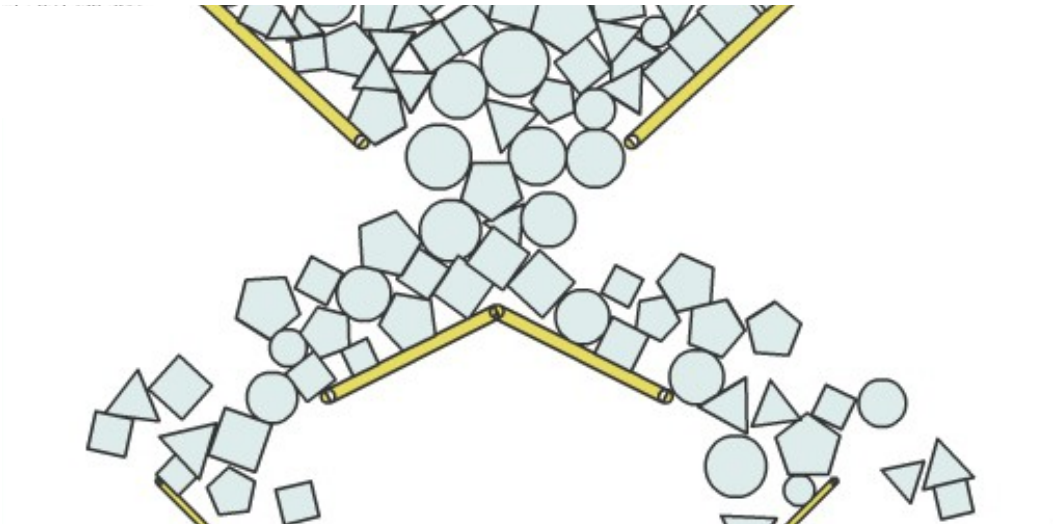

Figura 3.19: La utilización de formas estáticas permiten definir fácilmente obstáculos en el mundo sobre el que interaccionarán los elementos dinámicos. Esta imagen forma parte de los ejemplos oficiales de Physaxe.

3.4.1. Algunos Motores de Simulación

El desarrollo de un motor de simulación física desde cero es una tarea compleja y que requiere gran cantidad de tiempo. Afortunadamente existen gran variedad de motores de simulación física muy robustos, tanto basados en licencias libres como comerciales. A continuación se describirán brevemente algunas de las bibliotecas más utilizadas:

- **Bullet**. Bullet es una biblioteca de simulación física ampliamente utilizada tanto en la industria del videojuego como en la síntesis de imagen realista (Blender, Houdini, Cinema 4D y LightWave las utilizan internamente). Bullet es multiplataforma, y se distribuye bajo una licencia libre zlib compatible con GPL.
- **ODE**. ODE (*Open Dynamics Engine*) `www.ode.org` es un motor de simulación física desarrollado en C++ bajo doble licencias BSD y LGPL. El desarrollo de ODE comenzó en el 2001, y ha sido utilizado como motor de simulación física en multitud de éxitos mundiales, como el aclamado videojuego multiplataforma *World of Goo*, *BloodRayne 2* (PlayStation 2 y Xbox), y *TitanQuest* (Windows).
- **PhysX**. Este motor privativo, es actualmente mantenido por NVidia con aceleración basada en hardware (mediante unidades específicas de procesamiento físico PPUs *Physics Processing Units* o mediante núcleos CUDA. Las tarjetas gráficas con soporte de CUDA

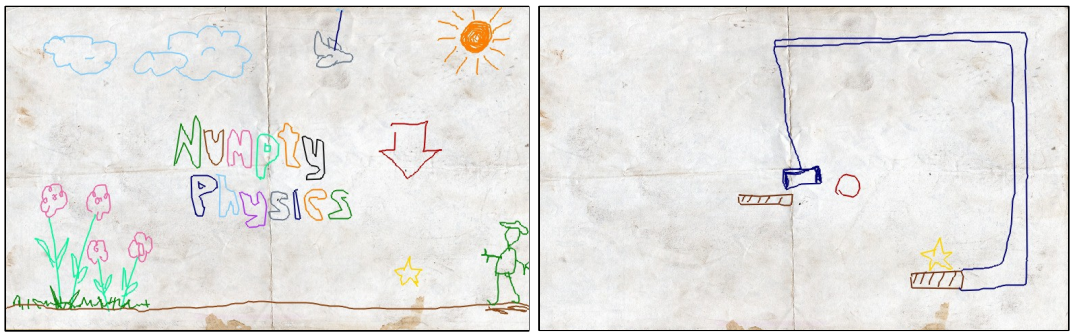

Figura 3.20: Capturas de pantalla de *Numpty Physics*, un clon libre del videojuego *Crayon Physics Deluxe*, igualmente basado en Box2D, cuyo objetivo es dirigir la bola al destino dibujando elementos de interacción física sobre la pantalla.

(siempre que tengan al menos 32 núcleos CUDA) pueden realizar la simulación física en GPU. Este motor puede ejecutarse en multitud de plataformas como PC (GNU/Linux, Windows y Mac), PlayStation 3, Xbox y Wii. El SDK es gratuito, tanto para proyectos comerciales como no comerciales. Existen multitud de videojuegos comerciales que utilizan este motor de simulación.

- **Havok**. El motor Havok se ha convertido en el estándar de facto en el mundo del software privativo, con una amplia gama de características soportadas y plataformas de publicación (PC, Videoconsolas y Smartphones). Desde que en 2007 Intel comprara la compañía que originalmente lo desarrolló, Havok ha sido el sistema elegido por más de 150 videojuegos comerciales de primera línea. Títulos como *Age of Empires*, *Killzone 2 & 3*, *Portal 2* o *Uncharted 3* avalan la calidad del motor.

- **Box2D**. Es un motor de simulación física 2D[22] libre que utilizaremos en este curso. Ha sido el corazón del fenómeno multiplataforma *Angry Birds*, que supuso más de 5 millones de ingresos a los propietarios sólo en concepto de publicidad. Entre otros títulos comerciales se encuentran *Crayon Physics Deluxe* (ver Figura 3.20), *Indredibots* y *Tiny Wings* (que se convirtió en la aplicación más vendidad del App Stores en Febrero de 2011).

Existen algunas bibliotecas específicas para el cálculo de colisiones (la mayoría distribuidas bajo licencias libres). Por ejemplo, *I-Collide*, desarrollada en la Universidad de Carolina del Norte permite calcular

[22] Web oficial en: http://box2d.org/

intersecciones entre volúmenes convexos. Existen versiones menos eficientes para el tratamiento de formas no convexas, llamadas *V-Collide* y *RAPID*.

Estas bibliotecas pueden utilizarse como base para la construcción de nuestro propio conjunto de funciones de colisión para videojuegos que no requieran funcionalidades físicas complejas.

La biblioteca **Physaxe** que utilizaremos en esta sección es, en realidad, una capa de recubrimiento sobre Box2D. La funcionalidad de Box2D está directamente accesible a través de los objetos de bajo nivel de la biblioteca.

3.4.2. Aspectos destacables

El uso de un motor de simulación física en el desarrollo de un videojuego conlleva una serie de aspectos que deben tenerse en cuenta relativos al diseño del juego, tanto a nivel de jugabilidad como de módulos arquitectónicos:

- **Predictibilidad**. El uso de un motor de simulación física afecta a la predictibilidad del comportamiento de sus elementos. Además, el ajuste de los parámetros relativos a la definición de las características físicas de los objetos (coeficientes, constantes, etc...) son difíciles de visualizar.
- **Realización de pruebas**. La propia naturaleza caótica de las simulaciones (en muchos casos no determinista) dificulta la realización de pruebas en el videojuego.
- **Integración**. La integración con otros módulos del juego puede ser compleja. Por ejemplo, ¿qué impacto tendrá en la búsqueda de caminos de Inteligencia Artificial el uso de simulaciones físicas? ¿cómo garantizar el determinismo en un videojuego multijugador?.
- **Realismo gráfico**. El uso de un motor de simulación puede dificultar el uso de ciertas técnicas de representación realista (como el uso de sprites con objetos que pueden ser destruidos). Además, el uso de cajas límite puede producir ciertos resultados poco realistas en el cálculo de colisiones.
- **Exportación**. La definición de objetos con propiedades físicas añade nuevas variables y constantes que deben ser tratadas por las herramientas de exportación de los datos del juego.

Figura 3.21: Gracias al uso de PhysX, las baldosas del suelo en *Batman Arkham Asylum* pueden ser destruidas (derecha). En la imagen de la izquierda, sin usar PhysX el suelo permanece inalterado, restando realismo y espectacularidad a la dinámica del juego.

- **Interfaz de Usuario**. Es necesario diseñar interfaces de usuario adaptados a las capacidades físicas del motor (¿cómo se especifica la fuerza y la dirección de lanzamiento de una granada?, ¿de qué forma se interactúa con objetos que pueden recogerse del suelo?).

3.4.3. Conceptos Básicos

A principios del siglo XVII, Isaac Netwon publicó las tres **leyes fundamentales del movimiento**. A partir de estos tres principios se explican la mayor parte de los problemas de dinámica relativos al movimiento de cuerpos y forman la base de la mecánica clásica. En el estudio de la dinámica resulta especialmente interesante la segunda ley de Newton, que puede ser escrita como $F = m \times a$, donde F es la fuerza resultante que actúa sobre un cuerpo de masa m, y con una aceleración lineal a aplicada sobre el centro de gravedad del cuerpo.

Desde el punto de vista de la mecánica en videojuegos, la **masa** puede verse como una medida de la resistencia de un cuerpo al movimiento (o al cambio en su movimiento). A mayor masa, mayor resistencia para el cambio en el movimiento. Según la segunda ley de Newton que hemos visto anteriormente, podemos expresar que $a = F/m$, lo que nos da una impresión de cómo la masa aplica resistencia al movimiento. Así, si aplicamos una **fuerza** constante e incrementamos la masa, la aceleración resultante será cada vez menor.

El **centro de masas** (o de **gravedad**) de un cuerpo es el punto espacial donde, si se aplica una fuerza, el cuerpo se desplazaría sin aplicar ninguna rotación.

Un **sistema dinámico** puede ser definido como cualquier colección de elementos que cambian sus propiedades a lo largo del tiempo. En el caso particular de las simulaciones de cuerpo rígido nos centraremos en el cambio de posición y rotación.

Así, nuestra **simulación** consistirá en la ejecución de un modelo matemático que describe un sistema dinámico en un ordenador. Al utilizar modelos, se simplifica el sistema real, por lo que la simulación no describe con total exactitud el sistema simulado.

Habitualmente se emplean los términos de interactividad y tiempo real de modo equivalente, aunque no lo son. Una **simulación interactiva** es aquella que consigue una tasa de actualización suficiente para el control por medio de una persona. Por su parte, una **simulación en tiepo real** garantiza la actualización del sistema a un número fijo de frames por segundo. Habitualmente los motores de simulación física proporcionan tasas de frames para la simulación interactiva, pero no son capaces de garantizar Tiempo Real.

3.4.4. Formas de Colisión

Como hemos comentado anteriormente, para calcular la colisión entre objetos, es necesario proporcionar una representación geométrica del cuerpo que se utilizará para calcular la colisión. Esta representación interna se calculará para determinar la posición y orientación del objeto en el mundo. Estos datos, con una descripción matemática mucho más simple y eficiente, son diferentes de los que se emplean en la representación visual del objeto (como hemos visto en la sección 3.2, basados en sprites) que cuentan con un mayor nivel de detalle).

Habitualmente se trata de simplificar al máximo la forma de colisión. Aunque el SDC (*Sistema de Detección de Colisiones*) soporte objetos complejos, será preferible emplear tipos de datos simples, siempre que el resultado sea aceptable. La Figura 3.22 muestra algunos ejemplos de aproximación de formas de colisión para ciertos objetos del juego.

Multitud de motores de simulación física separan la forma de colisión de la transformación interna que se aplica al objeto. De esta forma, como muchos de los objetos que intervienen en el juego son dinámicos, basta con aplicar la transformación a la forma de un modo computacionalmente muy poco costoso. Además, separando la transformación de la forma de colisión es posible que varias entidades del juego compartan la misma forma de colisión.

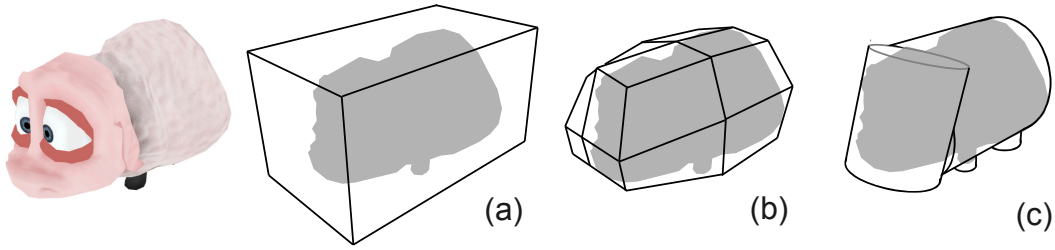

Figura 3.22: Diferentes formas de colisión para el objeto de la imagen. (a) Aproximación mediante una caja. (b) Aproximación mediante un volumen convexo. (c) Aproximación basada en la combinación de varias primitivas de tipo cilíndrico.

Algunos motores de simulación física permiten compartir la misma descripción de la forma de colisión entre entidades. Esto resulta especialmente útil en juegos donde la forma de colisión es compleja, como en simuladores de carreras de coches.

Como se muestra en la Figura 3.22, las entidades del juego pueden tener diferentes formas de colisión, o incluso pueden compartir varias primitivas básicas (para representar por ejemplo cada parte de la articulación de un brazo robótico).

El **Mundo Físico** sobre el que se ejecuta el SDC mantiene una lista de todas las entidades que pueden colisionar empleando habitualmente una estructura global *Singleton*. Este *Mundo Físico* es una representación del mundo del juego que mantiene la información necesaria para la detección de las colisiones. Esta separación evita que el SDC tenga que acceder a estructuras de datos que no son necesarias para el cálculo de la colisión.

Los SDC mantienen estructuras de datos específicas para manejar las colisiones, proporcionando información sobre la *naturaleza* del contacto, que contiene la lista de las formas que están intersectando, su velocidad, etc...

Para gestionar de un modo más eficiente las colisiones, las formas que suelen utilizarse con convexas. Una **forma convexa** es aquella en la que un rayo que surja desde su interior atravesará la superfice una única vez. Las superficies convexas son mucho más simples y requieren menor capacidad computacional para calcular colisiones que las formas cóncavas. Algunas de las primitivas soportadas habitualmente en SDC son:

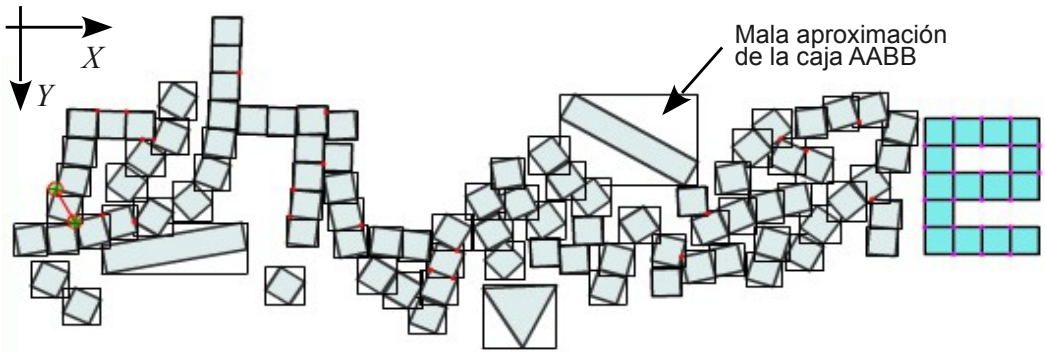

Figura 3.23: Gestión de la forma de un objeto empleando cajas límite alineadas con el sistema de referencia universal AABBs. Como se muestra en la figura, dependiendo de la rotación de la figura, puede ser que la aproximación mediante una caja AABB sea bastante pobre.

- **Esferas/Círculos**. Son las primitivas más simples y eficientes; basta con definir su centro y radio.
- **Cajas/Cuadrados**. Por cuestiones de eficiencia, se suelen emplear cajas límite alineadas con los ejes del sistema de coordenadas (AABB o ***Axis Aligned Bounding Box***). Las cajas AABB se definen mediante las coordenadas de dos extremos opuestos. El principal problema de las cajas AABB es que, para resultar eficientes, requieren estar alineadas con los ejes del sistema de coordenas global. Esto implica que si el objeto rota, como se muestra en la Figura 3.23, la aproximación de forma puede resultar de baja calidad. Por su eficiencia, este tipo de cajas suelen emplearse para realizar una primera aproximación a la intersección de objetos para, posteriormente, emplear formas más precisas en el cálculo de la colisión.
- **Cilindros/Cápsulas**. Los cilindros son ampliamente utilizados. Se definen mediante dos puntos y un radio. Las cápsulas pueden verse como el volumen resultante de desplazar una esfera entre dos puntos. El cálculo de la intersección con cápsulas es más eficiente que con esferas o cajas, por lo que se emplean para el modelo de formas de colisión en formas que son aproximadamente cilíndricas (como las extremidades del cuerpo humano).
- **Volúmenes/Áreas convexas**. La mayoría de los SDC permiten trabajar con volúmenes y áreas convexas (ver Figura 3.22). La forma del objeto suele representarse internamente mediante un conjunto de n planos (en el espacio 3D) o un conjunto de vértices (si se trabaja en 2D). Aunque este tipo de formas es menos eficiente que las primitivas estudiadas anteriormente, flexibilizan mucho el trabajo y son ampliamente utilizadas en desarrollo de videojuegos.

3.4.5. Optimizaciones

La detección de colisiones es, en general, una tarea que requiere el uso intensivo de la CPU. Por un lado, los cálculos necesarios para determianr si dos formas intersecan no son triviales. Por otro lado, muchos juegos requiren un alto número de objetos en la escena, de modo que el número de test de intersección a calcular rápidamente crece. En el caso de n objetos, si empleamos un algoritmo de fuerza bruta tendríamos una complejidad $O(n^2)$. Es posible utilizar ciertos tipos de optimizaciones que mejoran esta complejidad inicial:

- **Coherencia Temporal**. Este tipo de técnica de optimización (también llamada *coherencia entre frames*), evita recalcular cierto tipo de información en cada frame, ya que entre pequeños intervalos de tiempo los objetos mantienen las posiciones y orientaciones en valores muy similares.
- **Particionamiento Espacial**. El uso de estructuras de datos de particionamiento especial permite comprobar rápidamente si dos objetos podrían estar intersecando si comparten la misma celda de la estructura de datos. Algunos esquemas de particionamiento jerárquico, como árboles octales, BSPs o árboles-kd permiten optimizar la detección de colisiones en el espacio. Estos esquemas tienen en común que el esquema de particionamiento comienza realizando una subdivisión general en la raíz, llegando a divisiones más finas y específicas en las hojas. Los objetos que se encuentran en una determinada rama de la estructura no pueden estar colisionando con los objetos que se encuentran en otra rama distinta.
- **Barrido y Poda (SAP)**. En la mayoría de los motores de simulación física se emplea un algoritmo Barrido y Poda (*Sweep and Prune*). Esta técnica ordena las cajas AABBs de los objetos de la escena y comprueba si hay intersecciones entre ellos. El algoritmo *Sweep and Prune* hace uso de la *Coherencia temporal frame a frame* para reducir la etapa de ordenación de $O(n \times log(n))$ a $O(n)$.

En muchos motores, como en Box2D, se utilizan varias capas o pasadas para detectar las colisiones. Primero suelen emplearse cajas AABB para comprobar si los objetos pueden estar potencialmente en colisión (detección de la colisión amplia o en Inglés *Broadphase*). A continuación, en una segunda capa se hacen pruebas con volúmenes generales que engloban los objetos (por ejemplo, en un objeto compuesto por varios subobjetos, se calcula una esfera que agrupe a todos los subobjetos). Si esta segunda capa de colisión da un resultado positivo, en una tercera pasada se calculan las colisiones empleando las formas finales (colisión de granularidad fina).

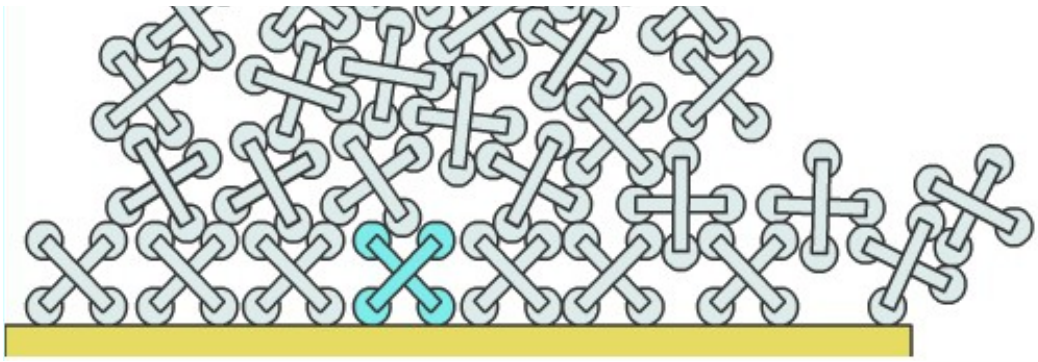

Figura 3.24: Physaxe soporta segmentos, polígonos con esquinas redondeadas, como forma de colisión básica.

3.4.6. Hola Mundo con Physaxe

Una vez estudiados los aspectos generales de los motores de simulación física, en esta sección presentaremos un ejemplo totalmente funcional con la biblioteca Physaxe, un wrapper escrito en Haxe del motor Box2D y de Glaze. Su autor destaca, entre otras, las siguientes características:

- Soporte de cuerpos rígidos basados en diversas formas de colisión: cajas (`phx.Shape.makeBox`), círculos (`phx.Circle`), Polígonos convexos de cualquier tipo (`phx.Polygon`) y segmentos (polígonos con esquinas redondeadas `phx.Segment`, ver Figura 3.24).
- Diversas fases de detección *Broadphase*.
- Activación y desactivación de objetos.
- Multitud de propiedades de simulación asociadas a los cuerpos rígidos (fricción, límites de movimiento, restricciones, etc...).

Antes de comenzar con los ejemplos, debes instalar Physaxe. Como vimos en las secciones anteriores, basta con ejecutar `sudo haxelib install physaxe`.

Como se ha comentado al inicio de la sección, el motor de simulación física es independiente del motor de representación gráfica. El caso de Physaxe no es una excepción, y será necesario desarrollar clases de utilidad que conecten ambos motores. Veamos a continuación el primer ejemplo básico tipo *Hola Mundo* con Physaxe.

Listado 3.50: Clase principal de HelloPhysic (Fragmento).

```
1  class HelloPhysic extends Sprite{
2    var _layer:TileLayer;              // Capa principal de dibujado
3    var _world:World;                  // Mundo de simulación física
4    var _vPhSprite:Array<PhSprite>;    // Array de Physical Sprite
5    // Constructor ========================================================
6    function init():Void {
7      loadResources();        // Carga los recursos gráficos (omitido)
8      _vPhSprite = new Array<PhSprite>();
9      createPhysicWorld();    // Crea el mundo de simulación física
10     addListeners();         // Añade los Listeners (omitido)
11   }
12   // Physic World =======================================================
13   function createPhysicWorld():Void {
14     var size = new AABB(-1000, -1000, 1000, 1000);  // Límites
15     var bp = new SortedList();          // Método de Broadphase
16     _world = new World(size, bp);       // Creación del mundo
17     createLimits(_sw, _sh);             // Crea cajas estáticas
18     _world.gravity = new phx.Vector(0,0.9);   // Gravedad
19   }
20   function createLimits(w:Float, h:Float):Void {
21     // Creamos los límites del mundo: makeBox(Ancho, Alto, X, Y)
22     // La X e Y de la caja tiene origen en esquina superior izqda.
23     _world.addStaticShape(Shape.makeBox(w,40,0,h));
24     _world.addStaticShape(Shape.makeBox(w,40,0,-40));
25     _world.addStaticShape(Shape.makeBox(40,h,-40,0));
26     _world.addStaticShape(Shape.makeBox(40,h,w,0));
27   }
28   // Update =============================================================
29   function onEnterFrame(e:Event):Void {
30     _world.step(1,10);             // Avanzamos un paso de simulación
31     for (p in _vPhSprite) p.update(_sw, _sh);
32     _layer.render(); _fpsText.update();
33   }
34   // Events =============================================================
35   function onMouseClick(e:MouseEvent):Void {
36     var p = new PhSprite("box", _layer, _world, e.localX, e.localY);
37     _vPhSprite.push(p);
38   }
39 }
```

La clase cuenta con un objeto de tipo *World* (ver línea 3). Esta clase forma parte de la distribución de *Physaxe* y contendrá los elementos que intervendrán en la simulación física (cuerpos, restricciones, propiedades, etc...). Aunque el motor interno de *Physaxe* (*Box2D*) soporta la creación de múltiples mundos, habitualmente no es necesario (ni siquiera deseable) tener varias instancias de esta clase en ejecución.

En la línea 5 se define el *Array* de objetos de tipo *PhSprite* (*Physical Sprite*). Esta clase de utilidad ha sido desarrollada para los ejemplos a modo de conector entre el motor de dibujado (basado en Sprites gráficos) y el motor de simulación física. En este ejemplo, esta clase será utilizada para añadir cajas de tamaño aleatorio en la escena.

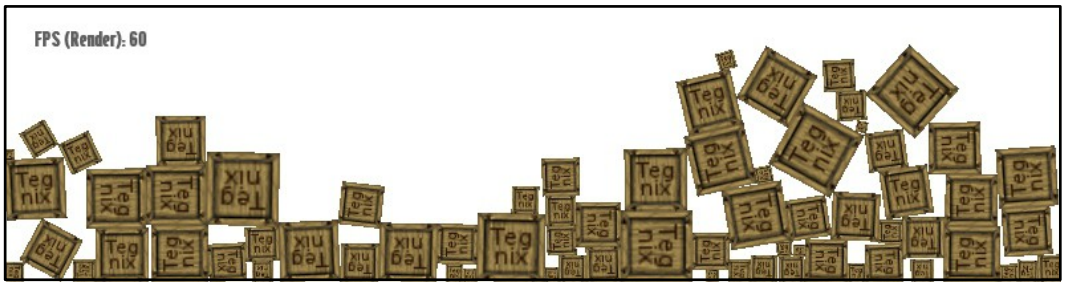

Figura 3.25: Resultado de la ejecución del *Hello Physics*. Cada vez que se pincha con el ratón sobre la ventana se añade una caja con dimensión y rotación aleatoria.

En el ejemplo de la siguiente sección, la clase *PhSprite* será rediseñada para soportar otras formas de colisión.

Las llamadas relativas a la creación del mundo de simulación física se han encapsulado en la llamada al método *createPhysicWorld* (ver líneas 13–19). La primera llamada en este método (línea 14) define los límites del mundo como una caja AABB. Estas dimensiones se especifican mediante dos vértices; el superior izquierdo y el inferior derecho. En este caso, únicamente se tendrán en cuenta los objetos físicos situados entre las coordenadas (-1000,-1000) y la (1000,1000).

Límites del Mundo. Internamente, *Physaxe* llama al método *boundCheck*, para determinar si los cuerpos se encuentran dentro de los límites del mundo. En otro caso, serán eliminados (ejecutando su método *onDestroy*).

A continuación se define el método de detección de colisión *BroadPhase* (línea 15). Esta primera etapa de detección de colisiones evita el tener que comparar todos los pares de formas de colisión, reduciendo el número de comparaciones a realizar. En *Physaxe* hay disponibles tres métodos de deteción *BroadPhase*:

- **BruteForce**. Este método almacena las formas de colisión en una lista y realiza la comprobación entre todos los pares. Este método es el más lento y no es aconsejable su uso salvo con fines de depuración.
- **SortedList**. Almacenando las formas de colisión en una lista ordenada, sólo se realizan las comprobaciones entre las formas que colisionan en el eje Y.

- **Quantize**. Es el método más avanzado de los tres. Divide el espacio del mundo en cuadrados de 2^n unidades. Cada forma se almacena en los cuadrados que intersecan con su caja límite (*bounding box*), comparando con el resto de formas que comparten el mismo cuadrado. Este método es recomendable si tenemos mundos muy grandes con una densidad pequeña de objetos dinámicos.

Si te encuentras con fuerza, puedes implementar tu propio método de *Broadphase* teniendo en cuenta el interfaz general definido en `phx.col.Broadphase`. Como recomienda el autor, puedes partir del método *Bruteforce* para orientar tu implementación.

Para la creación del mundo de simulación física (ver línea 16) es necesario especificar únicamente el tamaño del mundo y el método de optimización de *Broadphase* definidos en las líneas anteriores.

En la línea 18 se especifica la fuerza de la gravedad del mundo. Este vector puede cambiarse en cualquier momento en tiempo de ejecución.

La llamada al método auxiliar de la línea 17 sirve para crear los límites de colisión del mundo mediante llamadas a *addStaticShape*. Los cuerpos estáticos no colisionan con otros objetos estáticos y no pueden desplazarse (se comportan como si tuvieran masa infinita; en realidad *Box2D* almacena un valor de cero para la masa).

Mediante la llamada al método *makeBox* se crea una caja de colisión. Es importante señalar que las coordenadas de los objetos estáticos deben ser universales (coordenadas globales). De este modo, sabiendo que el (0,0) está situado en la esquina superior izquierda y que las coordenadas *X* e *Y* de la caja se indican con respecto de la esquina superior izquierda, las líneas 23–26 definen cuatro cajas situadas en los bordes de la ventana. Por ejemplo, la línea 23 define la caja que sirve como suelo (la caja queda *fuera* de la ventana, el borde superior de la caja coincide con el borde inferior de la ventana).

Cada vez que *renderizamos* un frame en el motor de representación gráfico tenemos que avanzar igualmente un paso en la simulación física. La actualización de la simulación física se realiza mediante la llamada al método *step* (línea 30). El primer argumento de la función es la cantidad de tiempo transcurrida desde la última actualización (especificado en 1/60 Segundos). Como el despliegue gráfico se realiza a 60fps, llamaremos al método con un valor de 1. El segundo argumento indica el número de subpasos de simulación para el cálculo de colisiones. Este valor permite garantizar que las colisiones se calculan de forma correcta

(evitando el denominado efecto *túnel*[23]). Valores mayores del número de subpasos obtienen mejores resultados, pero con un coste computacional mayor. En el caso de este ejemplo se han utilizado 10 subpasos.

De igual forma, en cada paso de simulación llamaremos al método *update* (ver línea 31) de la clase auxiliar *PhSprite*. Cada vez que el usuario haga *click* con el ratón, crearemos una nueva instancia de objeto *PhSprite* (ver línea 36), indicando como primer parámetro el nombre del *Sprite* que queremos utilizar ("*box*" en este caso), la capa de dibujado, el objeto del mundo físico y las coordenadas donde se añadirá el objeto. Este nuevo objeto se incluirá en el array *vPhSprite* (línea 37).

A continuación estudiaremos la implementación de los aspectos más relevantes de la clase *PhSprite*. Como se ha comentado anteriormente, esta clase será adaptada en el siguiente ejemplo para permitir cualquier forma de colisión (la implementación actual únicamnete soporta cajas).

La clase mantiene un cuerpo de simulación física (de la clase *Body*, ver línea 4) asociado a cada Sprite. El método update *update* (ver líneas 18-22) se encarga de obtener las propiedades de posición y rotación del cuerpo de simulación física y copiarlas al *Sprite*.

Listado 3.51: Clase PhSprite (Verión 1).

```
1  class PhSprite extends TileSprite {
2    var _root:TileLayer;    // Capa de dibujado de este Sprite
3    var _world:World;       // Mundo de simulación física
4    var _body:Body;         // Cuerpo de simulación asociado al Sprite
5    // Constructor ========================================================
6    public function new(id:String, l:TileLayer, w:World, px:Float,
7                py:Float):Void {
8      super(id); _root = l; _world = w; x = px; y = py;
9      _root.addChild(this);    // Añadir Sprite a la capa de dibujado
10     _body = new Body(x,y);   // Crear cuerpo de simulación física
11     scale = 0.25 + Std.random(75) / 50;    // Tamaño aleatorio
12     _body.addShape(Shape.makeBox(width, height));
13     _body.a = (Math.PI / 180.0) * Std.random(360);  // Rotación!
14     _body.updatePhysics();  // Actualización del cuerpo
15     _world.addBody(_body);  // Añadimos el cuerpo al mundo
16   }
17   // Update =============================================================
18   public function update(w:Int, h:Int):Void {
19     _body.updatePhysics();    // Actualizamos el cuerpo en físicas
20     x =_body.x; y =_body.y;   // Copiamos coordenadas al Sprite
21     rotation = _body.a;       // Copiamos rotación en Sprite
22   }
23 }
```

[23] El efecto túnel ocurre cuando los objetos no colisionan con otros objetos. Imaginemos que tenemos un objeto de colisión muy estrecho. Si otro objeto se mueve a una alta velocidad en su dirección y las colisiones se calculan de forma discreta es posible que, en los pasos de simulación concretos, ambos objetos no lleguen nunca a colisionar. De este modo, es como si el objeto hubiera pasado por un *túnel*, evitando así la colisión.

Es posible mover manualmente un cuerpo durante la simulación. Debes tener en cuenta en este caso que es necesario llamar al método `world.sync(body)` para notificar el cambio realizado al motor de simulación física.

La implementación del constructor (líneas 8–15) es muy sencilla. La línea 10 crea un cuerpo genérico, en las coordenas *X*,*Y* pasadas al constructor de *PhSprite*. En la línea 11 se obtiene un valor aleatorio para la escala del objeto. Mediante la llamada a *addShape* (línea 12) se especifica la forma de colisión de ese objeto. En este caso utilizamos una caja con las mismas dimensiones del Sprite. El atributo *a* del cuerpo permite indicar la rotación en radianes (ver línea 13). En este caso asignamos un valor aleatorio entre 0 y 360. La línea 15 añade el objeto al mundo de simulación física.

Dulces Sueños... Cuando un cuerpo permanece estático y su energía cinética disminuye a un valor menor que la definida en `world.sleepEpsilon`, el cuerpo pasa al estado de *dormido*. Un cuerpo dormido no puede moverse, salvo que lo despertemos manualmente. En este caso, basta con llamar a `world.activate(body)` para desprenderlo de los brazos de Morfeo.

Como veremos en la siguiente sección "*Más Allá del Hola Mundo*", los cuerpos y formas de colisión en *Physaxe* cuentan con multitud de parámetros que pueden definirse. En el caso del siguiente Mini-Juego definiremos materiales con propiedades físicas específicas y definiremos manualmente formas de colisión convexas. Además, definiremos un vector de velocidad lineal según la posición del puntero en tiempo de ejecución.

Identificadores únicos. Cada objeto de tipo `Body` y `Shape` cuentan con identificadores (campo *id* únicos). Aunque en la implementación son atributos públicos, no es conveniente modificarlos (aunque sí es muy útil consultarlos para comprobar si ha ocurrido una colisión entre objetos).