Figura 3.26: *Office Basket* en acción. El objetivo del juego es encestar la pelota de papel. Cada vez el origen se situará en una nueva posición del espacio.

3.4.7. Más allá del Hola Mundo

En esta sección estudiaremos el ejemplo del Mini-Juego *Office Basket* (ver Figura 3.26). En el desarrollo de este ejemplo hemos utilizado, además de la clase *PhSprite* convenientemente rediseñada, una clase auxiliar para el dibujado de la flecha (llamada *VectorArrow*). En el siguiente listado se muestran los aspectos más relevantes de la clase principal del ejemplo.

Listado 3.52: Clase principal de Office Basket (Fragmento).

```
1  class OfficeBasket extends Sprite{
2    var _previousTime:Int;            // Tiempo desde lanzamiento
3    var _sw:Int; var _sh:Int;         // Ancho y alto de la pantalla
4    var _spx:Float; var _spy:Float;   // Coordenadas de la bola
5    var _ball:PhPaperBall = null;     // Objeto para la bola
6    var _bin:PhPaperBin = null;       // Objeto para la papelera
7    var _arrow:VectorArrow;           // Clase para dibujar flecha
8
9    // Métodos Auxiliares ===========================================
10   function generateRandomPaperPosition():Void {
11     _spx =_sw/2 - 50 + Std.random(Std.int(_sw/2));
12     _spy = 50 + Std.random(Std.int(_sh/3));  }
13   function createArrow(px:Float, py:Float):Void {
14     _arrow = new VectorArrow(_layerArrow, "ball", "dot", "arrow",
15                  px, py, 10, 100);  }
```

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

```
16    function createBasket(w:Float, h:Float):Void {
17      var xaux = 50 + Std.random(Std.int(w/2));
18      _bin = new PhPaperBin("basket",_layerBin,_world, xaux, h-52); }
19    // Physic World ================================================
20    function createPhysicWorld():Void {
21      var size = new AABB(-1000, -1000, 1000, 1000);
22      var bp = new SortedList();
23      _world = new World(size, bp);
24      createLimits(_sw, _sh);
25      _world.gravity = new phx.Vector(0,0.9);
26    }
27    function createLimits(w:Float, h:Float):Void {
28      // Creamos los límites del mundo: makeBox(Ancho, Alto, X, Y)
29      // La X e Y de la caja tiene origen en esquina superior izqda.
30      var s = Shape.makeBox(w,200,0,h-20);
31      s.material.restitution = 0.2;  // Establecemos propiedades
32      s.material.density = 9;        // en el material...
33      _world.addStaticShape(s);
34      _world.addStaticShape(Shape.makeBox(w,200,0,-200));
35      _world.addStaticShape(Shape.makeBox(200,h,-200,0));
36      _world.addStaticShape(Shape.makeBox(200,h,w,0));
37    }
38    // Update ======================================================
39    function onEnterFrame(e:Event):Void {
40      var now = Lib.getTimer();
41      var secondsSinceClick = (now - _previousTime)/1000.0;
42      _world.step(1,20);
43      if (_ball != null) {       // Si hay bola (usuario hizo click)
44        _ball.update(_sw, _sh);  // Actualizamos la bola!
45        if (secondsSinceClick > 2) {   // Si han pasado más de 2 seg
46      // Comprobamos si la bola está tocando con el objeto
47      // que define la base de la papelera...
48      if (_ball.checkCollision(_bin.getIdBase()))
49        _scoreBoard.update(1);
50      _ball.destroy(); _ball = null;
51      generateRandomPaperPosition();  // Nueva posición!!
52      _arrow.setBasePos(_spx, _spy);  // Actualizar flecha
53        }
54      }
55      // Solo actualizamos la flecha si no hay bola activa...
56      else { _arrow.update();  _layerArrow.render(); }
57      _layerSky.render(); _layerPaper.render(); _layerBin.render();
58      _fpsText.update();
59    }
60    // Events ======================================================
61    function onMouseClick(e:MouseEvent):Void {
62      if (_ball == null) // Si no hay pelota activa, creamos una
63        _ball = new PhPaperBall("ball", _layerPaper, _world,
64                   _spx, _spy, _arrow.getVector());
65      _previousTime = Lib.getTimer();
66    }
67    function onMouseMove(event:MouseEvent):Void {
68      _arrow.setArrowPos(event.localX, event.localY);   }
69  }
```

tegnix

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

David Vallejo Fernández · Carlos González Morcillo
david.vallejo@tegnix.com · carlos.gonzalez@tegnix.com

Tras ejecutar el ejemplo, vemos que el juego genera posiciones aleatorias de la pelota de papel. Tras el lanzamiento de la pelota, dejamos que la simulación física actúe durante un intervalo de tiempo y generamos una nueva posición. El intervalo de tiempo transcurrido se calcula en base a la variable miembro *_previousTime* (ver línea 2). Las posiciones aleatorias de la bola se almacenan en las variables *_spx* y *_spy* mediante el método auxiliar *generateRandomPaperPosition* (definido en las líneas 10–12).

Los objetos de tipo *PhPaperBall* y *PhPaperBin* son clases hijas de la nueva implementación de la clase *PhSprite* (ver líneas 5–6), y serán descritas en las siguientes subsecciones. La creación de la papelera se realiza en la función *CreateBasket*, que instancia el objeto de tipo *PhPaperBin* en una posición aleatoria de la pantalla.

La variable *_arrow* utiliza la clase auxiliar *VectorArrow* para dibujar la flecha (ver Figura 3.27). El objeto se instancia en el método auxiliar *createArrow* (ver líneas 13–15).

El bucle principal (descrito en las líneas 39–54) se encarga de comprobar si el objeto *_ball* existe (línea 43). Se crea una instancia de la bola en la función de *callback* *onMouseClick* (definida en las líneas 61–66), empleando el constructor de la clase *PhPaperBall*.

Si la bola existe, se actualiza su posición, empleando el método *update* de la clase (línea 44). Si han pasado más de 2 segundos desde su lanzamiento (ver líneas 45 y 40–41), comprobamos si la pelota está colisionando con la base de la papelera. Si es así, (ver línea 49), actualizamos la puntuación. Tanto si hubo colisión como si no fue así, destruimos la bola actual y creamos una nueva (líneas 50–52).

Hemos introducido una modificación en la creación de los límites del mundo con respecto del ejemplo anterior. En este caso, estamos definiendo propiedades específicos del material (ver líneas 30–32).

La clase **Shape** mantiene una variable *material* de la clase *Material*. En esta variable se pueden establecer ciertas propiedades básicas de los materiales:

- **restitution**. El *Coeficiente de Restitución* mide la cantidad de velocidad que mantiene un cuerpo cuando colisiona con otro cuerpo (rebote). Es un valor entre 1 y 0. Valores cercanos a 1 implican mayor rebote.
- **density**. La densidad se mide en kg/m^3, y se utiliza para calcular la masa del cuerpo.
- **friction**. La fricción mide cuánto se desliza una forma sobre una superficie. Este valor habitualmente se especifica entre 0 (no hay fricción) y 1.

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

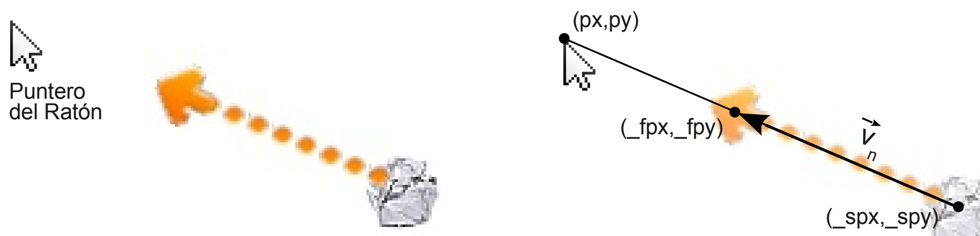

Figura 3.27: Las variables miembro *_fpx*, *_fpy* se calculan en base a la posición del ratón *px,py*. Obtenemos el vector v normalizado a una determinada longitud, especificado en la variable miembro *_maxLength*.

La tabla 3.2 muestra los valores que toman estas variables para algunos materiales.

Material	**Densidad**	**Fricción**	**Restitución**
Metal	7.85	0.2	0.2
Piedra	2.4	0.5	0.1
Madera	0.53	0.4	0.15
Cristal	2.5	0.1	0.2
Goma	1.5	0.8	0.4
Hielo	0.92	0.01	0.1
Tela	0.03	0.6	0.1
Esponja	0.018	0.9	0.05
Aire	0.001	0.9	0.0

Cuadro 3.2: Definición de las propiedades de algunos materiales.

A continuación estudiaremos los aspectos más relevantes de la implementación de la clase de utilidad *VectorArrow*. Como muestra la Figura 3.27, las variables miembro (*_spx*, *_spy*) y (*_fpx*, *_fpy*) definen las coordenadas iniciales y finales del vector (ver líneas 5-6). La posición incial vendrá determinada por la posición de la bola (calculada aleatoriamente en la clase principal del juego). Como veremos, la posición final vendrá determinada por la línea que forma el puntero del ratón y la posición de la bola. La clase, además, permite especificar el número de puntos que se dibujarán como parte del vector *_nPoints* o la longitud máxima de dibujado *_maxLength* (líneas 3-4).

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

Listado 3.53: Clase auxiliar de Vector Arrow (Fragmento).

```
1  class VectorArrow {
2    var _vSprite:Array<TileClip>;      // Sprites a dibujar...
3    var _nPoints:Int;                  // Número de puntos a desplegar
4    var _maxLength:Int;                // Longitud máxima del vector
5    var _spx:Float; var _spy:Float;    // Posición inicial del vector
6    var _fpx:Float; var _fpy:Float;    // Posición final del vector
7    // Constructor ==================================================
8    public function new(l: TileLayer, idBase:String, idDot:String,
9                 idArrow, px:Float, py:Float, nPoints:Int,
10                length:Int):Void {
11     _root = l; _nPoints = nPoints; _spx = px; _spy = py;
12     _maxLength = length;  // Longitud máxima del vector
13     _vSprite = new Array<TileClip>();
14     var s:TileClip;
15     for (i in 1..._nPoints+1) {  // Añadir elementos
16       if (i==1) s = new TileClip(idBase);
17       else if (i==nPoints) {s = new TileClip(idArrow); s.scale = 2;}
18       else s = new TileClip(idDot);
19       s.x = _spx; s.y = _spy; // Inicialmente todos en esta posición
20       _vSprite.push(s); _root.addChild(s);
21     }
22   }
23   // Clases auxiliares ============================================
24   public function getVector():phx.Vector {
25     var v = new Point(_fpx - _spx, _fpy - _spy);
26     v.normalize(v.length * 0.25);
27     return new phx.Vector(v.x, v.y);
28   }
29   public inline static function radToDeg(rad:Float):Float
30     { return 180 / Math.PI * rad;  }
31   // Update =======================================================
32   public function update():Void {
33     var v = new Point(_fpx - _spx, _fpy - _spy);
34     v.normalize(v.length / _nPoints);
35     for (i in 0..._nPoints) {          // Calculamos nueva posición
36       _vSprite[i].x = _spx + v.x * i; // escalando el vector v
37       _vSprite[i].y = _spy + v.y * i;
38       if (i == _nPoints -1) {  // Actualizar rotación de la flecha
39         v.normalize(1);
40         var angle = Math.acos(v.x);
41         if (v.y < 0) angle = 2*Math.PI - angle;
42         _vSprite[i].rotation = angle;
43       }
44     }
45   }
46   public function setArrowPos(px:Float, py:Float):Void {
47     var v = new Point(px - _spx, py - _spy);
48     if (v.length > _maxLength) {
49       v.normalize(_maxLength);
50       _fpx = _spx + v.x; _fpy = _spy + v.y;   }
51     else { _fpx = px; _fpy = py; }
52   }
53   public function setBasePos(px:Float, py:Float):Void
54     { _spx = px; _spy = py;   }
55 }
```

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

David Vallejo Fernández · Carlos González Morcillo
david.vallejo@tegnix.com · carlos.gonzalez@tegnix.com

El constructor de la clase (líneas 8-22) crea un array de *Sprites* que serán dibujados para representar el vector. En la base se posiciona una instancia de la bola de papel (línea 16), en el otro extremo una flecha (línea 17) y como elementos intermedios Sprites de puntos (línea 18). Estos sprites serán reposicionados cada vez que el usuario mueva el ratón. La función de *callback* que se ejecuta cuando hay un cambio en la posición del ratón es *setArrowPos*, definida en las líneas 46-52.

Esta función calcula la posición final del array de Sprites en base a la línea que une la posición del ratón con la posición inicial de la bola de papel (calculada aleatoriamente). El vector v (línea 47) se normaliza a la dimensión máxima especificada en el constructor de la clase (ver Figura 3.27). El punto final se calcula simplemente sumando ese vector a la posición inicial de la bola de papel (línea 50).

El método *update* (definido en las líneas 32-45) reposiciona los sprites del array, obteniendo posiciones intermedias mediante una sencilla interpolación lineal. El ángulo del sprite final se obtiene empleando el producto escalar del vector V con la rotación incial de la flecha (especificada mediante el vector (1,0)).

Como hemos comentado anteriormente, la clase auxiliar *PhSprite* ha sido convenientemente rediseñada en este ejemplo para facilitar la incorporación de nuevas formas de colisión. En concreto, el método *createShape* (definido en la línea 8 del siguiente listado) debe ser sobreescrito por cada subclase de *PhSprite*. Este método puede ser empleado tanto por subclases que implementen objetos dinámicos (como el caso de *PhPaperBall*) como por objetos estáticos (la papelera, implementada en *PhPaperBin* es un objeto de colisión estático). Esta clase de utilidad emplea un atributo *_isDynamic* (línea 5) que indica el tipo de comportamiento del objeto.

Únicamente en el caso de ser un objeto dinámico, será necesario actualizar su posición y rotación en el método *update* (ver líneas 37-39).

Listado 3.54: Clase auxiliar de PhSprite V.2 (Fragmento).

```
 1 class PhSprite extends TileSprite {
 2   var _root:TileLayer;      // Capa de dibujado del objeto
 3   var _world:World;         // Mundo de simulación en Physaxe
 4   var _body:Body;           // Cuerpo físico de simulación
 5   var _isDynamic:Bool;      // El objeto es dinámico o estático?
 6   // Este método debe ser sobreescrito...
 7   // Si es una StaticShape, puede añadirlas al mundo y devolver null
 8   function createShape():Shape { return null;  }
 9   // Constructor ==================================================
10   public function new(id:String, l:TileLayer, w:World, px:Float,
11                py:Float, isDynamic:Bool, ?vect:phx.Vector):Void {
12     super(id); _root = l; _world = w; _isDynamic = isDynamic;
13     x = px; y = py;
14     _root.addChild(this);
```

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

```
15      if (_isDynamic) {
16        _body = new Body(x,y);
17        _body.addShape(createShape());
18        _body.w = Std.random(100)/1000.0;    // Veloc. Angular
19        _body.v = vect;
20        _body.updatePhysics();
21        _world.addBody(_body);
22      }
23      else {
24        // El objeto es estático, puede añadir la forma al mundo
25        // en el propio método 'createShape'
26        var s = createShape();
27        if (s!=null) _world.addStaticShape(s);
28      }
29    }
30    // Destructor ===================================================
31    public function destroy():Void {
32      if (_isDynamic) _world.removeBody(_body);
33      _root.removeChild(this);
34    }
35    // Update =======================================================
36    public function update(w:Int, h:Int):Bool {
37      if (_isDynamic) {
38        x = _body.x;  y = _body.y;  rotation = _body.a;
39        _body.updatePhysics();
40      }
41      return true;
42    }
43  }
```

En base a esta implementación de la clase *PhSprite* se definen dos clases hijas específicas para el objeto dinámico de la bola de papel (clase *PhPaperBall*) y para la papelera (clase *PhPaperBin*). La implementación de estas clases hijas de *PhSprite* es muy sencilla, y básicamente se encargan de implementar los métodos específicos que definen su geometría de colisión y su comportamiento.

La clase *PhPaperBall* define un polígono convexo en la función *createShape* (líneas 3-10). El convenio requerido por *Physaxe* en la definición de los vértices es que éstos deben ser indicados por orden en el sentido contrario al giro de las agujas del reloj (ver Figura 3.28). Las coordenadas pueden especificarse además según un desplazamiento inicial. Si no se indica nada, se tienen que indicar respecto del centro del Sprite. Esto resulta molesto, ya que la mayoría de los paquetes gráficos miden el desplazamiento con respecto de la esquina superior izquierda. Si conocemos el ancho y el alto del sprite, podemos indicar ese desplazamiento como último parámetro al constructor de la clase *Polygon* (ver línea 9). En el caso de la bola de papel, el Sprite tiene un tamaño de 32x32 píxeles. Como el (0,0) se indicaría en el centro del Sprite, para reposicionar el origen en la esquina superior izquierda, hay que indicar (-16,-16) como vector de *offset*.

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

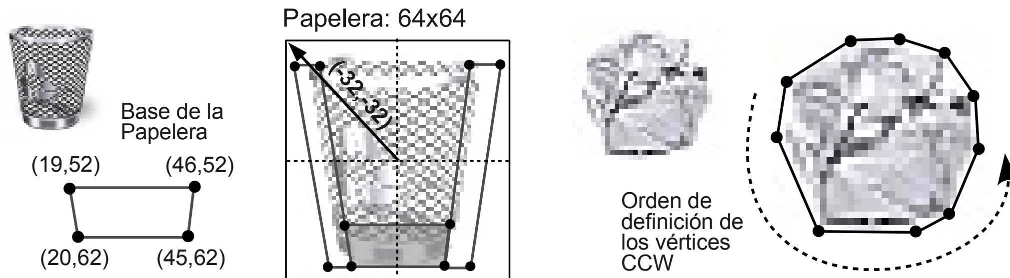

Figura 3.28: Definición de los vértices de los objetos del Mini-Juego, descritos en el método *createShape* de *PhPaperBin* y *PhPaperBall*. En la figura de la izquierda se describen las coordenadas relativas de una de las partes que definen la papelera, así como el vector de *Offset* (-32,-32) que indica el nuevo origen para especificar las coordenadas. A la derecha se representan los vértices empleados en la bola de papel, así como el orden de definción de los mismos.

De forma análoga, en la Figura 3.28 (izquierda) se indica que el offset en el caso de la papelera debe ser de (-32,-32) porque el tamaño del Sprite es de 64x64 píxeles.

Listado 3.55: Clase PhPaperBall (Fragmento).

```
1  class PhPaperBall extends PhSprite {
2    // Create Shape ================================================
3    override function createShape():Shape {
4      return (new phx.Polygon ([new phx.Vector(1,12),
5         new phx.Vector(1,17),  new phx.Vector(7,30),
6         new phx.Vector(20,30), new phx.Vector(25,29),
7         new phx.Vector(29,19), new phx.Vector(28,12),
8         new phx.Vector(17,2),  new phx.Vector(7,6)],
9         new phx.Vector(-16,-16)));
10   }
11   // Constructor =================================================
12   public function new(id:String, l:TileLayer, w:World, px:Float,
13                 py:Float, vect:phx.Vector):Void {
14     super(id, l, w, px, py, true, vect);    }
15   // Check Collision ==============================================
16   public function checkCollision(idBase:Int):Bool {
17     for (a in _body.arbiters.iterator())
18       if ((a.s1.id == idBase) || (a.s2.id == idBase)) return true;
19     return false;
20   }
21   // Update =======================================================
22   override public function update(w:Int, h:Int):Bool {
23     x = _body.x;  y = _body.y;  rotation = _body.a;
24     _body.updatePhysics();
25     return true;   }
26 }
```

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

David Vallejo Fernández · Carlos González Morcillo
david.vallejo@tegnix.com · carlos.gonzalez@tegnix.com

Para comprobar si la bola colisiona con la base de la papelera se ha creado un método auxiliar *checkCollision* (ver líneas 17–20). La clase *Body* incorpora una lista de *árbitros* con referencias a todos los objetos que colisionan entre sí (por parejas). En el caso de la línea 18 se recorren todos estos árbitros y se comprueba si alguno de los dos objetos que están colisionando tiene el mismo identificador que el objeto pasado como argumento a la función. Esto nos permite, desde la clase principal del juego pasarle como identificador el correspondiente a la base de la papelera (ver Figura 3.28).

Documentación de Physaxe. Desafortunadamente, no existe una documentación completa de las clases de la biblioteca Physaxe. Sin embargo, el código fuente de la misma es muy claro, y en la mayoría de los casos, basta con mirar la implementación de las clases. En tu distribución de GNU/Linux, la implementación de la versión 1.2 puedes encontrarla en `/usr/lib/haxe/lib/physaxe/1,2/phx/`.

Por último, la implementación de la clase *PhPaperBin* añade tres formas de colisión estáticas en el método *createShape* (ver líneas 4–21). Como desde el propio método se añaden las tres formas al mundo, es necesario devolver *null* en la línea 21 (de otra forma, el constructor de la clase *PhSprite* se encargaría de añadir la forma devuelta al invocar a ese método).

Coordenadas Globales. Recordemos que las coordenadas de los objetos de colisión estáticos deben ser especificadas de forma global. Por esta razón, a las coordenadas específicas de cada vértice de la papelera hay que añadirle las coordenadas globales de su centro (indicadas como variables miembro de la clase Sprite, en x e y).

Cuando se añade el objeto base de la papelera (líneas 16–20), se guarda el identificador como variable miembro de la clase *_idBase*. Este identificador puede consultarse mediante la llamada al método público *getIdBase*, definido en la línea 24. Esto nos permite proporcionar dicho identificador al objeto de la clase *PhPaperBall* y comprobar si existe colisión.

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

Listado 3.56: Clase PhPaperBin (Fragmento).

```
1  class PhPaperBin extends PhSprite {
2    var _idBase:Int;
3    // Create Shape ================================================
4    override function createShape():Shape {
5      // Borde izquierdo de la papelera
6      _world.addStaticShape(
7         new phx.Polygon ([new phx.Vector(6+x,9+y),
8            new phx.Vector(14+x,62+y), new phx.Vector(20+x,62+y),
9        new phx.Vector(12+x,9+y)], new phx.Vector(-32,-32)));
10     // Borde derecho de la papelera
11     _world.addStaticShape(
12        new phx.Polygon ([new phx.Vector(52+x,9+y),
13       new phx.Vector(45+x,62+y), new phx.Vector(51+x,62+y),
14       new phx.Vector(58+x,9+y)], new phx.Vector(-32,-32)));
15     // Base de la papelera
16     var saux = new phx.Polygon ([new phx.Vector(19+x,52+y),
17       new phx.Vector(20+x,62+y), new phx.Vector(45+x,62+y),
18       new phx.Vector(46+x,52+y)],new phx.Vector(-32,-32));
19     _idBase = saux.id;
20     _world.addStaticShape(saux);
21     return null;
22   }
23   // getIdBase ===================================================
24   public function getIdBase():Int {return _idBase;}
25   // Constructor =================================================
26   public function new(id:String, l:TileLayer, w:World, px:Float,
27               py:Float):Void {
28     super(id, l, w, px, py, false);
29   }
30 }
```

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es