Documentación Juego Retro LBreakOut2

Simón Sloan García Villa Alejandro Tirado Ramirez

October 31, 2025

Contents

1 Introducción

En esta oportunidad estamos trabajando un juego retro clásico llamado *LBreakOut2*, mediante el lenguaje de programación *Jack* del programa educativo *Nand2Tetris*, usando por supuesto el compilador correspondiente *JackCompiler* para pasarlo a *Aritmética de Stack* y luego ejecutarlo en *VMSimulator*. En las próximas secciones se ahondará más sobre cada uno de los aspectos particulares tanto del juego, como su solución y por supuesto los aprendizajes adquiridos.

2 Ejecución del juego

A continuación se describen los pasos necesarios para ejecutar el juego desde la terminal en un sistema operativo Linux:

1. Clone el repositorio con el siguiente comando: *git clone git@github.com:SSloan07/LBreckOut.git*.
2. Ingrese al directorio del proyecto: *cd LBreckOut*.
3. Acceda al subdirectorio *Square*: *cd Square*.
4. Compile el código utilizando: *JackCompiler*.
5. Diríjase al directorio donde tenga instalado *Nand2Tetris*.
6. Ingrese a las carpetas correspondientes: *cd Nand2Tetris/nand2tetris/tools*.
7. Ejecute el emulador de la máquina virtual: *bash VMEmulator.sh*.

Una vez abierto el *VMEmulator*, seleccione la ruta donde clonó el repositorio y cargue el archivo *Square.vm*. Finalmente, ejecute el programa con la opción *No animation* ¡y disfrute del juego!

3 Sobre el juego

Como se dijo en la introducción, en esta oportunidad nos propusimos la creación del clásico juego retro *LBreakOut2* donde tenemos una plataforma que se mueve, de izquierda a derecha (o viceversa) por medio de las teclas ←, →. Además tenemos cierto número de obstáculos, en el caso particular de nosotros, 4 filas y 8 columnas de rectángulos, que tenemos que destruir sin que la pelota se escape por la parte inferior de la pantalla. Por otra parte, tenemos un sistema de vidas, donde solo se pierde si se te escapa la pelota 3 veces (cada vez que pierdas una vida, aparecerá un contador con una vida menos) y se gana si destruyes todos los obstáculos.

4 Sobre el diseño general escogido

En términos generales tenemos *6 archivos*, estos son:

- `Ball.jack` - Control del movimiento de la pelota
- `Gameboard.jack` - Gestión del tablero de juego
- `Main.jack` - Punto de entrada principal
- `Obstacle.jack` - Manejo de obstáculos/bloques
- `Square.jack` - Elementos gráficos básicos
- `SquareGame.jack` - Lógica principal del juego

A continuación se explicará de manera un poco más precisa, cada uno de los documentos.

4.1 Square.jack

Nuestro juego, al igual que la gran mayoría de los juegos retro usa la figura geométrica *Cuadrado* como base. En esta línea de ideas, este archivo es bastante importante en nuestro juego, porque los otros documentos usarán sus funciones para ejecutar ciertas acciones como mostrar las paredes o incluso la propia barra. Además en este archivo están las acciones fundamentales de movimiento de la barra con funciones como *moveLeft* o *moveRight*.

Consultar el archivo Square.jack para más detalle.

4.2 Ball.jack

Este archivo es bastante corto pero tiene todo el movimiento de la pelota (incluyendo la lógica necesaria para que rebote exactamente en un ángulo de 45 grados).

4.2.1 Movimiento básico

Para lograr el movimiento, tenemos el siguiente método principal:

```
1  method void move() {
2    do erase();
3    let x = x + dx;
4    let y = y + dy;
5    do draw();
6    return;
7  }
```

Listing 1: Método move() de la pelota

Aquí está la magia, y es que como podemos ver, tenemos un movimiento tanto en x como en y, donde **y** y **x** son las posiciones iniciales en **y** y en **x**, y donde **dy** y **dx** son la posición deseada (después del movimiento).

4.2.2 Física del movimiento a 45°

Para garantizar el movimiento a 45 grados, inicializamos `dx` y `dy` con valores absolutos iguales (generalmente 1 o -1). Esto asegura que:

$$\text{velocidad} = \sqrt{dx^2 + dy^2} = \sqrt{1^2 + 1^2} = \sqrt{2}$$

Y el ángulo se mantiene en:

$$\theta = \arctan\left(\frac{dy}{dx}\right) = \arctan(1) = 45^\circ$$

4.2.3 Sistema de rebotes

Los métodos de rebote mantienen este ángulo cambiando solo una componente a la vez:

```
    // Rebote vertical - invierte direcci n
        Y
    method void bounceY() {
      let dy = -dy;
      return;
    }

    // Rebote horizontal - invierte
        direcci n X
    method void bounceX() {
      let dx = -dx;
      return;
    }
```

Listing 2: Métodos de rebote

Este diseño garantiza que la pelota siempre se mueva en diagonales de 45 grados, simplificando la física del juego mientras se mantiene un comportamiento predecible y jugable.

4.3 Obstacle.jack

Este archivo es el que le da proposito al juego, pues es el que se encarga, tanto de la generación de los obstaculos, como de su activación o desactivación, permitiendo por supuesto, pintarlos o no pintarlos (cabe aclarar, aunque este archivo no existe por si solo, si permite que el archivo Gameboard.jack haga la magia, gestionando así los metodos que este documento ofrece). Como metodos destacados tenemos *draw()* y *destroy()*. El primero es el que se encarga de pintar los obstaculos (si y solo si su atributo **active** es igual a True). Por otra parte, el segundo, **borra** los obstaculos solo si están **activos** y después de ejecutada esta acción, cambia el estado del atributo **active** a **false**. A continuación el codigo mencionado:

```
  method void draw() {
    if (active) {
      do obstacle.draw();
    }
    return;
  }
```

Listing 3: Método draw de los obstaculos

```
    method void destroy() {
      if (active) {
        do erase();
        let active = false;
      }
      return;
    }
```

Listing 4: Método destroy de los obstaculos

4.4 Gameboard.jack

Este archivo cumple un papel fundamental dentro del diseño general del juego, ya que actúa como el **núcleo del tablero de juego**. Es el encargado de inicializar los elementos visibles del entorno (paredes y obstáculos), gestionar las colisiones de la pelota con los distintos componentes y mantener el control sobre los elementos activos del juego.

A diferencia de los demás módulos, `Gameboard.jack` no solo administra objetos, sino que también ejecuta parte de la lógica central del juego, garantizando la coherencia espacial y la correcta interacción entre la pelota, la barra y los obstáculos.

4.4.1 Estructura general y atributos

El archivo define la clase `Gameboard` con los siguientes campos principales:

- `topWall`, `leftWall`, `rightWall`: objetos de tipo `Square` que representan las tres paredes visibles del tablero (superior, izquierda y derecha).
- `screenWidth`, `screenHeight`: dimensiones del área visible del juego, normalmente 512×256 píxeles.
- `obstacles`: arreglo que contiene todos los obstáculos del juego, creados en una disposición matricial.
- `activeObstacles` y `obstacleCount`: contadores de obstáculos activos y del total inicial respectivamente.
- `type`: valor auxiliar usado para determinar el tipo de colisión o interacción.

Esta estructura permite mantener una representación lógica clara del entorno físico donde se desarrolla el juego.

4.4.2 Inicialización del tablero

El constructor `new()` prepara todo el escenario. En primer lugar, ajusta las coordenadas de las paredes para adaptarse a los límites de la pantalla. Posteriormente, se crea el arreglo de obstáculos y se llama al método `createObstacles()` para distribuirlos ordenadamente. Finalmente, se define el número inicial de obstáculos activos y se retorna la instancia del tablero.

```
    constructor Gameboard new(int width, int
        height) {
      let screenWidth = width;
      let screenHeight = height;

      // Inicializaci n de paredes
      let topWall = Square.new(0, 0, 1, 1);
      let leftWall = Square.new(0, 0, 1, 1);
      let rightWall = Square.new(width - 11,
          0, 1, 1);

      // Inicializaci n de obst culos
      let obstacleCount = 12;
      let obstacles =
          Array.new(obstacleCount);
      do createObstacles();
      let activeObstacles = 12;
      return this;
    }
```

Listing 5: Constructor del tablero de juego

4.4.3 Creación de obstáculos

El método `createObstacles()` se encarga de generar la disposición inicial de los obstáculos en una cuadrícula de tres filas por cuatro columnas. Cada obstáculo se crea con una posición inicial (x, y) que se actualiza dinámicamente a medida que avanza el ciclo anidado, garantizando un espaciado uniforme entre filas y columnas.

```
    method void createObstacles() {
      var int i, j, x, y, index;
      let y = 50; // Fila inicial

      while (j < 3) {
        let x = 80; // Columna inicial
        while (i < 4) {
          let index = (j * 4) + i;
          let obstacles[index] =
              Obstacle.new(x, y, 8);
          let x = x + 100; // Espacio entre
              obst culos
          let i = i + 1;
        }
        let y = y + 30; // Espacio entre
            filas
        let j = j + 1;
      }
      return;
    }
```

Listing 6: Método de creación de obstáculos

Esta modularidad facilita escalar la dificultad del juego simplemente ajustando el número de filas, columnas o la separación entre los obstáculos.

4.4.4 Dibujo del tablero

El método `draw()` se encarga de renderizar las paredes del juego. Dado que la pantalla tiene límites de 0–511 píxeles en el eje **x** y 0–255 en el eje **y**, se realizan pequeños ajustes para que las paredes queden perfectamente visibles dentro del área. Entre cada dibujo se incluye una pausa temporal mediante `Sys.wait()` para permitir una visualización progresiva.

```
    method void draw() {
      var int adjustedWidth, adjustedHeight;
      let adjustedWidth = screenWidth - 1;
      do topWall.drawWall(adjustedWidth, 10);
      do Sys.wait(1000);

      let adjustedHeight = screenHeight - 1;
      do leftWall.drawWall(10,
          adjustedHeight);
      do Sys.wait(1000);

      do rightWall.drawWall(10,
          adjustedHeight);
      do Sys.wait(1000);
      return;
    }
```

Listing 7: Método de dibujo del tablero

4.4.5 Sistema de colisiones

El corazón de la jugabilidad se encuentra en el método `checkBallCollisions()`, que determina cuándo y cómo reacciona la pelota ante los distintos elementos del entorno. El método calcula los bordes de la pelota y de la barra, y evalúa superposiciones para determinar si ocurrió una colisión. En caso afirmativo, se invoca el rebote correspondiente. También se verifican los límites del tablero y la pérdida de la pelota.

```
method boolean checkBallCollisions(Ball
    ball, Square paddle) {
  var int ballLeft, ballRight, ballTop,
      ballBottom;
  var int paddleLeft, paddleRight,
      paddleTop, paddleBottom;

  // Coordenadas de la pelota y la barra
  let ballLeft = ball.getLeft();
  let ballRight = ball.getRight();
  let ballTop = ball.getTop();
  let ballBottom = ball.getBottom();
  let paddleLeft = paddle.getX();
  let paddleRight = paddle.getX() +
      (paddle.getSize() * 4);
  let paddleTop = paddle.getY();
  let paddleBottom = paddle.getY() +
      (paddle.getSize() / 2);

  // Colisiones con la barra
  if (~(ballBottom < paddleTop)) {
    if (~(ballTop > paddleBottom)) {
      if (~(ballRight < paddleLeft)) {
        if (~(ballLeft > paddleRight)) {
          do ball.bounceY();
          return false;
        }
      }
    }
  }

  // Colisiones con obst culos y paredes
  do checkObstacleCollisions(ball);
  if (~(ballTop > 10)) { do
      ball.bounceY(); }
  if (~(ballLeft > 10)) { do
      ball.bounceX(); }
  if (~(ballRight < 500)) { do
      ball.bounceX(); }

  // Verificar si la pelota se perdi
  if (~(ballBottom < 245)) {
    return true;
  }

  return false;
}
```

Listing 8: Verificación de colisiones principales

4.4.6 Gestión de colisiones con obstáculos

Cuando la pelota impacta un obstáculo, se destruye visualmente y se reduce el contador de obstáculos activos. Este comportamiento se implementa en el método `checkObstacleCollisions()`, que recorre el arreglo de obstáculos y compara sus coordenadas con las de la pelota.

```
method void checkObstacleCollisions(Ball
    ball) {
  var int i;
  var boolean isActive;
  var int obsLeft, obsRight, obsTop,
      obsBottom;
  var int ballLeft, ballRight, ballTop,
      ballBottom;
  var Obstacle currentObstacle;
  var Square obstacleSquare;

  let i = 0;
  while (i < obstacleCount) {
    let currentObstacle = obstacles[i];
    let isActive =
        currentObstacle.isActive();

    if (isActive) {
      let obstacleSquare =
          currentObstacle.getSquare();
      let ballLeft = ball.getLeft();
      let ballRight = ball.getRight();
      let ballTop = ball.getTop();
      let ballBottom = ball.getBottom();
      let obsLeft =
          obstacleSquare.getX();
      let obsRight =
          obstacleSquare.getX() +
          (obstacleSquare.getSize() * 4);
      let obsTop = obstacleSquare.getY();
      let obsBottom =
          obstacleSquare.getY() +
          (obstacleSquare.getSize() / 2);

      if (ballBottom > obsTop) {
        if (ballTop < obsBottom) {
          if (ballRight > obsLeft) {
            if (ballLeft < obsRight) {
              do
                  currentObstacle.destroy();
```

```
30                  let activeObstacles =
                        activeObstacles - 1;
31                  do ball.bounceY();
32                }
33              }
34            }
35          }
36        }
37        let i = i + 1;
38      }
39      return;
40    }
```

Listing 9: Gestión de colisiones con los obstáculos

Este método contribuye al dinamismo del juego, manteniendo la sensación de progreso y recompensa visual cada vez que el jugador destruye un bloque.

4.5 SquareGame.jack

El archivo `SquareGame.jack` constituye el **controlador principal del juego**. Es el encargado de coordinar la interacción entre los distintos componentes del sistema: la pelota (`Ball`), el tablero (`Gameboard`) y la barra (`Square`). A diferencia de los demás módulos, su función no es representar un elemento del juego, sino gestionar la dinámica completa, desde la inicialización hasta el final de la partida.

4.5.1 Propósito y estructura

La clase `SquareGame` implementa la lógica de ejecución general, manejando tanto el ciclo principal del juego como las entradas del usuario. Sus campos principales son:

- `square`: representa la barra controlada por el jugador.
- `board`: tablero donde se dibujan los elementos del juego.
- `ball`: la pelota principal del juego.
- `direction`: variable que indica la dirección actual del movimiento.

Esta combinación de objetos refleja una arquitectura modular (lo que nos brinda escalabilidad): cada clase se ocupa de un aspecto visual o funcional concreto, mientras que `SquareGame` se limita a orquestar su interacción.

4.5.2 Inicialización del juego

El constructor crea los objetos principales del juego: la barra, el tablero y la pelota. Luego llama al método `draw()` del tablero y de la pelota, presentando al jugador el estado inicial de la partida. En este punto, la dirección de movimiento se establece en cero, indicando que la barra aún no se mueve.

4.5.3 Gestión del movimiento

El método `moveSquare()` permite desplazar la barra horizontalmente según la dirección registrada. Este método se ejecuta de forma continua dentro del ciclo principal, con un pequeño retardo temporal (`Sys.wait(5)`) para suavizar el movimiento. Se omiten los desplazamientos verticales, ya que en este juego la barra solo se mueve de izquierda a derecha.

4.5.4 Ciclo principal de ejecución

La lógica principal se encuentra en el método `run()`, donde se gestionan los eventos, las colisiones y las condiciones de victoria o derrota. Durante cada iteración del bucle, se siguen los siguientes pasos:

1. Se lee la tecla presionada por el usuario.
2. Se actualizan las posiciones de la barra y de la pelota.
3. Se verifica si la pelota colisionó con la barra, las paredes o los obstáculos mediante el método `checkBallCollisions()` del tablero.
4. Si la pelota se pierde, se reduce el contador de vidas y se redibuja una nueva pelota en la posición inicial.
5. Si el número de vidas llega a cero, se muestra el mensaje "*Perdiste!!*" y el juego finaliza.
6. Si todos los obstáculos fueron destruidos, se declara la victoria mostrando "*Ganaste!!*".

4.5.5 Interacción con el jugador

El esquema de control mantiene una simplicidad clásica, propia de los juegos retro. Las teclas de flecha izquierda y derecha permiten mover la barra

4.5.6 Cierre del juego

Cuando el jugador gana o pierde, el método finaliza mostrando un mensaje en pantalla y limpiando los recursos utilizados mediante el método `dispose()`, que libera tanto la memoria del tablero como la de los objetos gráficos.

4.6 Main.jack

Este documento refleja el esqueleto principal del programa, teniendo como premisa fundamental, la inicialización de una instancia de **SquareGame** y llamar el metodo *run()* de este, de modo que cuando este finaliza su ejecución es porque el jugador o ganó o perdió, lo que naturalmente finaliza la ejecución del programa (después de limpiar la memoria con *dispose()* por supuesto).

5 Aprendizajes

Este proyecto resultó sumamente significativo, pues nos permitió enfrentarnos a un desafío nuevo: desarrollar un videojuego retro utilizando el lenguaje de programación `Jack`. A lo largo del proceso, logramos comprender los fundamentos físicos detrás del movimiento de la pelota y la barra, aspectos esenciales para la correcta dinámica del juego.

Además, este trabajo representó el cierre ideal del curso, ya que nos permitió apreciar la conexión entre los distintos niveles de abstracción de la computación. Desde la construcción lógica del hardware mediante compuertas, pasando por la elaboración de una ALU, hasta la traducción de instrucciones en lenguaje ensamblador, cada paso nos acercó a entender cómo la memoria, el manejo del stack y la aritmética se integran para formar la base de los lenguajes modernos.

Finalmente, al programar un videojuego funcional, comprendimos de forma tangible cómo todos esos conceptos convergen. Sin duda, fue un verdadero viaje en el tiempo: una experiencia que nos permitió reconocer la belleza de la computación y valorar todo el camino recorrido para alcanzar las tecnologías que hoy consideramos cotidianas.