Trabajo Práctico Final

Juego de Basketball Booteable

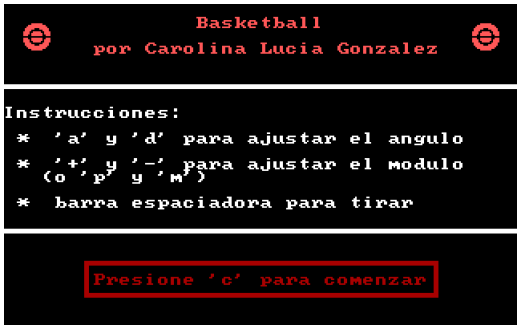
R-222 Arquitectura del Computador

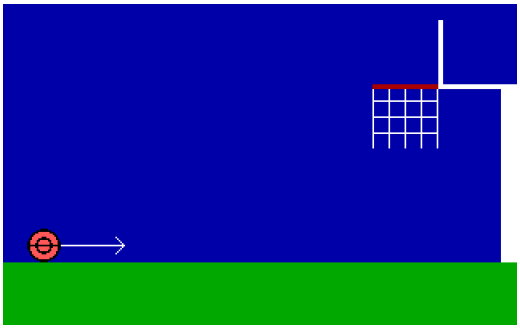
Carolina Lucía González (G–4850/1)

Índice

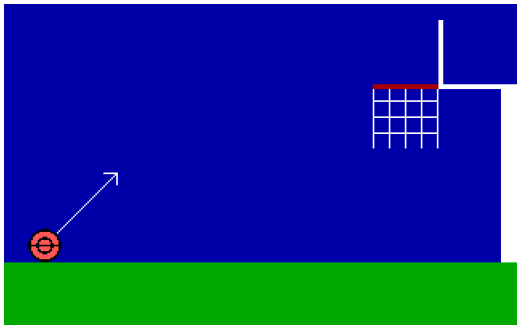
1. Introducción

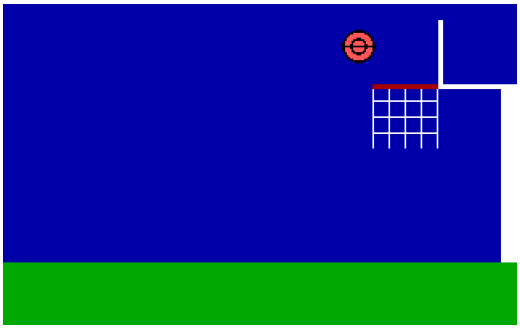
Para este trabajo decidí mostrar cómo generar código booteable y trabajar con el modo de video, aplicándolo en un juego sencillo.





Se trata de un juego en el cual el objetivo es encestar la pelota. Cuenta con dos niveles. Una flecha indica la velocidad de la pelota. Se puede ajustar el ángulo (con 'a' y 'd') y el módulo (con '+' y '-', o con 'p' y 'm'), y para lanzarla basta con presionar la barra espaciadora.





El código está escrito en NASM 16 bits y funciona para la arquitectura x86.

2. Metodología

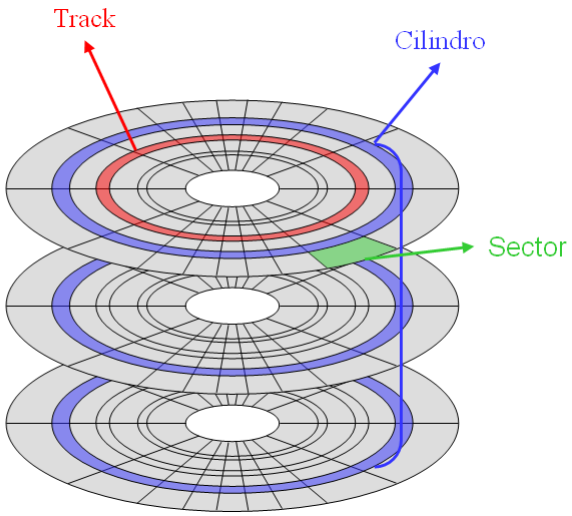
Dividí el trabajo en varios módulos (los detallados en el índice). Cada uno de ellos es crucial para el programa y, salvo por la parte del booteo, no dependen uno de otro. Para algunos de ellos primero tuve que hacer un esquema en pseudocódigo con el fin de facilitarme la escritura de los mismos en assembler

(por este motivo encontrarán partes del código muy comentadas, aunque la idea en general era escribir un código prolijo y legible por cualquiera).
En todo momento trabajo en modo real.

2.1. Código booteable

Los bootloaders tienen un límite de 512 bytes, y esto (en principio) no permite hacer demasiadas cosas.
Para programas más extensos se necesitan dos partes, donde la primera (de 512 bytes) se encargue de cargar la segunda (en la cual estará el código largo del programa).

Antes de entrar en detalles del booteo, precisamos conocer la organización del disco.
Un disco duro es un grupo de *platters* (platos/discos), normalmente con 2 lados (o cabezas) cada uno. Esos lados (*heads*) están numerados del 0 al n. Cada head está dividido en varios *tracks*, y a su vez estos están divididos en *sectores*. Un sector es, comunmente, de 512 bytes. Un *cilindro* es un conjunto de tracks con igual radio en el disco, comprende todos los tracks con el mismo número a lo largo de todos los heads.



CHS (*cylinder-head-sector*, en español: cilindro-cabeza-sector) es una forma antigua de dar direcciones a los bloques físicos de datos en discos duros de menos de 8GB. Actualmente se usa el sistema LBA (direccionamiento lógico de bloques), que consiste en dividir el disco entero en sectores y asignar a cada uno un único número. Si bien los valores CHS ya no tienen una relación

física directa con los datos almacenados, algunos programas siguen utilizando valores virtuales de los mismos.

Volviendo al bootloader, para cargar la segunda parte con la primera necesito indicar cuánto espacio ocupa (medido en cantidad de sectores de 512 bytes), las direcciones CHS y de memoria en las que comienza el código de esa segunda parte, y de dónde bootear.

En assembler:

- **int 13h:** interrupción de BIOS que se encarga de las operaciones de acceso a disco. (Nota: aunque lo mencione antes, debe ir luego de asignarle los valores a los registros y antes del `jmp`.)
- **ah:** 02h es la función para leer sectores.
  `mov ah,02h`
- **al:** número de sectores a leer. En mi caso necesito un poco menos de 5,5KiB, por lo tanto cargo 11 sectores.
  `mov al,11`
- **ch:** número de cilindro.
  `mov ch,0`
- **cl:** número de sector.
  `mov cl,2`
- **dh:** head number (número de cabeza).
  `mov dh,0`
- **dl:** drive number (número de unidad).
  Al bootear, `dl` es inicializado con el debido número de drive. Luego, el valor de `dl` será el correcto en tanto no se modifique ese registro.
- **es:bx :** (segment:offset) dirección de memoria para la cual cargar los sectores. Como el archivo de la segunda parte empieza en 0x00007e00[1], corresponde que `es:bx` tenga ese valor.
- **cf:** luego de ejecutar `int 13h`, `cf` se encontrará activada si se produjo un error.
- **jmp 0x00007e00 :** para ir a la segunda parte.

[1] En la siguiente página explico el porqué.

Además:
En la primera parte:

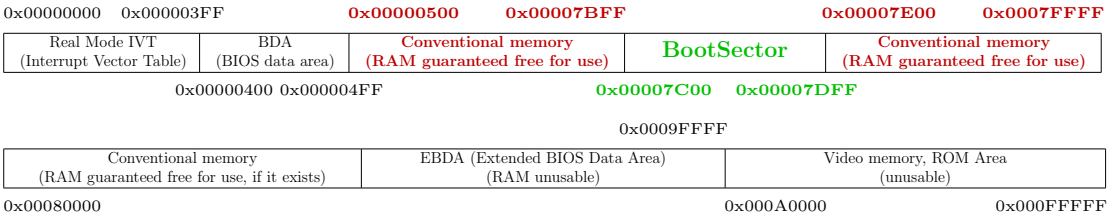
- ▷ `[ORG 0x7c00]`: dirección de memoria donde comienza el archivo.
- ▷ `times 510-($-$$) db 0`: completa los últimos bytes del archivo con 0 (salvo los últimos dos).
- ▷ `dw 0xAA55`: últimos dos bytes que corresponden a la firma del booteo.

En la segunda parte:

- ▷ `[ORG 0x7e00]`: dirección de memoria donde comienza el archivo.
- ▷ `times 5632-($-$$) db 0`: completa los últimos bytes con 0.

Por supuesto que en las dos partes también hay que inicializar los registros de segmento como corresponda.

Las direcciones de los archivos y stacks no pueden ser cualesquiera, deben respetar el mapa de memoria (x86)[6], que es algo así:[2]



Entonces la dirección de la primera parte del bootloader será 0x7C00, la de la segunda parte 0x7E00, y las direcciones de los stacks pueden estar entre 0x500 y 0x7BFF o entre el fin de la segunda parte y 0x7FFF.

2.2. Gráficos

Para empezar el juego necesito activar el modo de video. Esto lo hago con la interrupción 10h (interrupción de BIOS que controla los servicios de pantalla), con `ah` = 00h (función para activar el modo de video) y `al` = 13h (modo 13h: resolución de 320 × 200 y 256 colores[5]).

```
xor ax,ax
mov al,13h
int 10h
```

[2]Muestro sólo el primer MiB ya que en modo real las direcciones son de 20 bits. Para acceder a direcciones superiores hay que cambiar a modo protegido, y no me pareció necesario hacer esto dado que utilizo poca memoria.

2.2.1. Pintar pixeles

Una vez activado el modo de video, para pintar un pixel necesito:

- En `cx` y `dx`: coordenadas (x, y) (respectivamente) del pixel.
- En `al`: color.
- En `ah`: 0ch, función para escribir un pixel gráfico.
- `int 10h`

También es importante que el pixel se encuentre dentro de los límites de la pantalla, ya que de lo contrario se puede pintar un pixel no deseado.
Es por esto que creo una función que se fija si las coordenadas están dentro de la pantalla, y de ser así lo pinta. Requiere: x en `cx`, y en `dx`, c en `al` y `ah` = 0ch (siendo (x, y) las coordenadas del pixel y c el color).

```
pset:
        cmp cx,0
        jl pset.fin
        cmp dx,0
        jl pset.fin
        cmp cx,319
        jg pset.fin
        cmp dx,199
        jg pset.fin

        int 10h

        .fin:
        ret
```

Para las figuras que aparecen en el programa son necesarias unas funciones base para dibujar rectángulos, líneas rectas, círculos y circunferencias. En todas estas ellas, para pintar un pixel, utilizo la función anterior.

Rectángulo
La siguiente función recibe[3] 5 parámetros: x_1, y_1, x_2, y_2 y c, donde (x_1, y_1) y (x_2, y_2) son las coordenadas (en pixeles) de dos esquinas opuestas del rectángulo y c es el número del color.

[3]Cada vez que diga que una función "recibe parámetros" me refiero a que esos valores se encuentran en el stack (y en ese orden).

El algoritmo es muy sencillo: primero intercambia los valores de las coordenadas de modo que $x_1 \leq x_2$ y $y_1 \leq y_2$, y luego simplemente recorre todos los pixeles con coordenadas (x, y) tales que $x_1 \leq x \leq x_2$ y $y_1 \leq y \leq y_2$, y los pinta del color c.

Línea recta
Una recta (no vertical) está determinada por la siguiente ecuación:

$$y = mx + h$$

Con dos puntos de una recta se pueden determinar m y h. En efecto, supongamos que tenemos (x_0, y_0) y (x_1, y_1) entonces:

$$y_0 = mx_0 + h \qquad\qquad y_1 = mx_1 + h$$

$$y_0 - y_1 = m(x_0 - x_1)$$

$$h = y_0 - mx_0$$

Por lo tanto:

$$m = \frac{y_0 - y_1}{x_0 - x_1} \qquad\qquad h = y_0 - \frac{y_0 - y_1}{x_0 - x_1} x_0$$

Utilizaré esto para graficar todos los puntos de un segmento.

Necesito partir (en principio) en 2 casos: recta vertical, recta no vertical.
El caso de la recta vertical ($x_0 = x_1$) lo resuelvo directamente pintando todos los vértices (x, y) con $x = x_0$ y $y_0 \leq y \leq y_1$ (o $y_1 \leq y \leq y_0$).
El otro caso es más complicado:

$$y = \left(\frac{y_0 - y_1}{x_0 - x_1}\right) x + \left(y_0 - \frac{y_0 - y_1}{x_0 - x_1} x_0\right)$$

Lo que tiene de complicado es que las coordenadas deben ser enteras, y esto hace que en la mayoría de los casos no haya la misma cantidad de puntos en el eje x que en el eje y. ¿Qué tiene de malo esto? Que no da lo mismo iterar sobre x que sobre y, es necesario recorrer el eje más largo (de lo contario quedarían huecos en la línea).
Así que de vuelta hay que partir en dos casos: $|x_0 - x_1| > |y_0 - y_1|$ y $|x_0 - x_1| \leq |y_0 - y_1|$.

Caso $|x_0 - x_1| > |y_0 - y_1|$:
(pseudocódigo)

```
for xi = x0 to x1
        yi = xi*(y1-y0)/(x1-x0) + h
        pset(xi,yi,line_color)
next xi
```

(x_i aumenta o decrece según sea $x_0 \leq x_1$ o $x_0 > x_1$)

Caso $|x_0 - x_1| \leq |y_0 - y_1|$:
(pseudocódigo)

```
for yi = y0 to y1
        xi = (yi-h)*(x1-x0)/(y1-y0)
        pset(xi,yi,line_color)
next yi
```

(y_i aumenta o decrece según sea $y_0 \leq y_1$ o $y_0 > y_1$)

Círculo
La función recibe 4 parámetros: x, y, r y c, siendo (x, y) el centro del círculo, r el radio y c el color.
Matemáticamente, un círculo con centro (x, y) es el conjunto de todos los puntos del plano que distan a lo sumo r de (x, y). Es decir, son todos los puntos (a, b) con $(a - x)^2 + (b - y)^2 \leq r^2$.
La función hace lo siguiente: dentro del cuadrado de lado $2r$ y centro (x, y), se fija qué pixeles (a, b) cumplen con la ecuación anterior (porque me pareció más sencillo que estar calculando $\sqrt{r^2 - (a - x)^2}$ por cada a posible).
Como a y b son enteros, en realidad lo que se forma no tiene mucha forma de círculo, por lo tanto hay que "suavizarlo" pintando algunos puntos que aproximan la figura. Es así que decido pintar los pixeles (a, b) tales que:
$(a - x)^2 + (b - y)^2 \leq r^2 + \dfrac{(r + 1)^2 - r^2}{4}$.

Circunferencia
Similar al círculo, pero pinto los pixeles (a, b) tales que:
$(r - 1)^2 + \dfrac{r^2 - (r - 1)^2}{4} \leq (a - x)^2 + (b - y)^2 \leq r^2 + \dfrac{(r + 1)^2 - r^2}{4}$.

2.2.2. Escritura en pantalla

Dentro de las funciones que ofrece la interrupción 10h se encuentra una que permite escribir una cadena de caracteres con ciertos atributos.
En assembler se precisa:

- **ah:** 13h
- **al:** modo de escritura (número entre 0 y 3 inclusive).
- **bl:** color (sólo para los modos de escritura 0 y 1).
- **cx:** longitud de la cadena.
- **dh:** fila en la cual se escribirá la cadena.
- **dl:** columna en la cual se escribirá la cadena.
- **es:bp :** dirección de memoria del comienzo de la cadena.

En el programa la utilizo para mostrar las instrucciones y los mensajes del resultado del tiro.

2.3. Lectura del teclado

Para modificar la velocidad y efectuar el tiro necesito saber si se presionó alguna tecla y cuál.
La interrupción 16h se encarga de controlar el teclado. Con `ah` = 01h se obtiene el estado del buffer del teclado (básicamente dice si se presionó alguna tecla o no). Con `ah` = 00h se lee la tecla presionada, devuelve en `al` el caracter ASCII de la misma.

Entonces, para leer una tecla hago lo siguiente:

```
loop:
        mov ah,01h
        int 16h     ; Toma el estado del buffer del teclado
        jz loop     ; Si no se presiona ninguna tecla, volver a fijarse
                    ; Si se presiono alguna:
        xor ax,ax
        int 16h     ; Esto nos dice que tecla fue presionada
```

2.4. Wait

Para que el juego sea "visible" necesito que las imágenes permanezcan fijas durante cierto tiempo.
La interrupción 15h controla servicios extendidos de PC. Con `ah` = 86h se tiene una función que espera un intervalo de tiempo, medido en microsegundos (en `cx` la parte alta y en `dx` la parte baja del número deseado).
Por ejemplo, para esperar 1/4 s:

```
mov cx,0x0003
mov dx,0xd090
mov ah,86h
int 15h
```

(porque $3D090_{(16)} = 250000_{(10)}$)

2.5. Apagar sistema

La interrupción 15h también permite apagar la computadora, controla el APM (Advanced Power Management). Se necesita:

- `ax` = 5307h
- `cx` = 0003h
- `bx` = 0001h

Cuando termina el juego, la computadora se apagará gracias a esto.

2.6. Tiro

Una vez que se arroja, la pelota está en constante movimiento (describiendo una trayectoria parabólica). Hay que borrarla y redibujarla cada vez que cambia su posición, cada un cierto intervalo de tiempo $\Delta(t)$ definido de antemano.
Como en la física real, un vector modela su velocidad. En el programa guardo el módulo de la velocidad en x y el módulo de la velocidad en y. Además, elegí que la gravedad sea 8 para simplificar las cuentas.

2.6.1. Ecuaciones de tiro parabólico

La velocidad de un cuerpo que sigue una trayectoria parabólica se puede obtener con las fórmulas:

$$v_x(t) = v_{0x} \qquad\qquad v_y(t) = v_{0y} - gt$$

donde v_{0x} es el módulo de la velocidad inicial en x, v_{0y} es el módulo de la velocidad inicial en y, t es el tiempo transcurrido y g la gravedad.

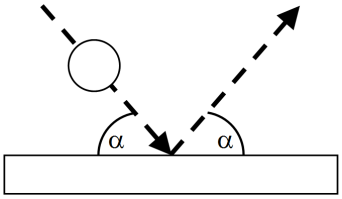
La posición del cuerpo en cada instante está dada por:

$$r_x(t) = x_0 + v_{0x}t \qquad\qquad r_y(t) = y_0 + v_{0y}t - \frac{1}{2}gt^2$$

donde x_0 y y_0 son las posiciones iniciales en x y en y respectivamente.[7]

2.6.2. Rebotes

La siguiente figura ilustra el comportamiento de la pelota cuando rebota:



Si es un rebote contra una superficie vertical (una pared por ejemplo) hay que modificar la componente x de la velocidad, multiplicándola por -1.
Si es un rebote contra una superficie horizontal, hay que modificar la componente y de la velocidad, multiplicándola por -1.

En el juego, cuando la pelota rebota hay que calcular la velocidad en ese instante y despúes aplicar el cambio necesario.

2.6.3. Fin del tiro

La pelota se detiene cuando se da alguna de las siguientes situaciones:

- ▷ La posición en x de la pelota no está en el intervalo $[0, 319]$ (se salió de la pantalla y no puede volver, ya sea porque rebotó o porque el tiro fue muy fuerte).
- ▷ La pelota toca el piso (su posición en y es mayor o igual a un cierto número).
- ▷ Se gana el juego. El juego se gana cuando se encesta la pelota, es decir, cuando $x + r$ y $x - r$ están dentro de los límites horizontales del aro, y cuando y está a la misma altura (siendo (x, y) la posición de la pelota y r su radio). Aunque así definido hace que se pueda encestar desde abajo, cosa que sería imposible o trampa en el basket real, pero como es un juego lo dejo así.

En ese momento se mostrará un cartel con el mensaje “GANASTE :)” o “PERDISTE :(” según corresponda.

3. Posbiles mejoras

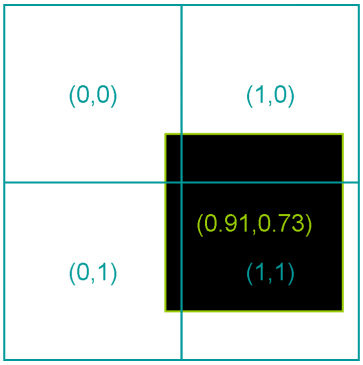
Desde algoritmos más eficientes hasta modelos físicos más reales. Se pueden enumerar algunas:

- ⋆ **Respecto al juego:** Quitar el inconveniente de encestar por abajo (que podría ser simplemente agregando un rebote en la red). Distintos efectos de rebote. Gráficos más elaborados (por ejemplo, agregar una persona tirando, o una tribuna de fondo). Efectos de festejo cuando se gana. Agregar obstáculos (como una pared, otro jugador, algo que vuela desde la tribuna), incluso algunos de ellos para destruir (habría que estudiar las ecuaciones físicas que determinan la rotura de los mismos). Más niveles.

- ⋆ **Respecto al código assembler:** Utilizar funciones y operaciones más eficientes. Tal vez algún truco matemático que ahorre operaciones/tiempo/recursos (como hacer `xor ax,ax` para tener 0 en `ax` en lugar de `mov ax,0`). Para las funciones de línea, círculo y circunferencia seguramente existen mejores implementaciones que las expuestas.

4. Observaciones

- ⊛ En lugar de usar `idiv` (que ejecuta la división entera entre un número en `dx:ax` y otro en el stack), utilizo una función propia que toma dos números del stack y redondea el resultado obtenido.
  Esto es porque al trabajar con enteros, necesito que los resultados sean lo más cercano posible a los verdaderos. Por ejemplo, si quiero pintar el pixel $\left(\frac{10}{11}, \frac{8}{11}\right)$, en el mapa tendría algo así:



Empleando `idiv` pintaría el pixel $(0, 0)$, cuando el que más aproxima es el $(1, 1)$.
La función que grafica segmentos calcula muchas divisiones, y no es conveniente acarrear tantos errores.
Además, cada vez que se usa `idiv` hay que guardar el contenido de los registros `ax` y `dx` si estos se están usando en el programa. Esto puede resultar muy molesto cuando se llama varias veces a esa función.

⊛ Tuve que crear una función que calcule la raíz cuadrada entera de un número natural (de hasta 16 bits). Para esto empleé el "algoritmo del ábaco"[8], que es muy práctico porque se vale de operaciones sencillas (shifts, sumas y restas). El único inconveniente es que devuelve la parte entera del resultado real, no redondea, y esto puede hacer que varíe ligeramente algún valor (pero para lo único que se emplea esta función es para calcular el módulo de la velocidad y graficar la flecha, por lo tanto no es tan grave).

Referencias

[1] NASM Documenation
`http://www.nasm.us/xdoc/2.10.07/nasmdoc.pdf`

[2] Booteo
`http://www.cs.cmu.edu/~410-s07/p4/p4-boot.pdf`

[3] Cylinder-head-sector
`http://en.wikipedia.org/wiki/Cylinder-head-sector`

[4] Interrupciones
`http://es.wikipedia.org/wiki/Llamada_de_interrupcin_del_BIOS`
`http://es.wikipedia.org/wiki/Int_13h`
`http://es.wikipedia.org/wiki/Int_10h`
`http://www.ctyme.com/intr/rb-0210.htm`
`http://es.wikipedia.org/wiki/Int_16h`
`http://es.wikipedia.org/wiki/Int_15h`
`http://www.cs.ubbcluj.ro/~dadi/ac/doc/ng87350.html`
`http://www.ctyme.com/intr/rb-1404.htm`

[5] Modo 13h
`http://es.wikipedia.org/wiki/Modo_13h`

[6] Memory Map (x86)
http://wiki.osdev.org/Memory_Map_(x86)

[7] Movimiento parabólico
http://es.wikipedia.org/wiki/Movimiento_parablico

[8] Square Root by Abacus Algorithm - Martin Guy, June 1985
http://medialab.freaknet.org/martin/src/sqrt/