DONKEY KONG

Memoria de la proyecto final de la asignatura "Programación"

Grado: Ingeniería Informática

Miguel Hernández Cassel & Ana Tian Villanueva Conde
NIA de los alumnos: 100405956 & 100405817

Universidad Carlos III Madrid

Índice:

Resumen del documento

El proyecto final propuesto para esta asignatura de Programación de 1º del grado en Ingeniería Informática es trata del un juego clásico arcade “Donkey Kong”, en el lenguaje de programación Python, utilizando la librería de Pyxel. En esta memoria se recoge y analiza las clases utilizadas, el diseño de dichas clases, incluyendo los métodos y algoritmos más relevantes implementados en el código del juego. Se irá explicando el proceso de trabajo en el proyecto, finalizando la memoria con una breve conclusión de nuestras opiniones acerca del proyecto.

Diseño de las clases

Damos paso al análisis de las clases del código. En el código encontramos una clase principal denominada “App”; en la que se llama al resto de clases y se usa de forma mayoritaria la librería Pyxel que disponemos para realizar la práctica. El resto de las clases del código sirven de fundamento para los personajes (Donkey Kong, Mario y Pauline) y elementos del tablero (Plataformas, escaleras y barriles)

- *Class App:*

Como ya ha sido enunciado, esta es la clase principal o main de todo el código. En esta clase encontramos los métodos esenciales para hacer que el juego aparezca; `reset(self)`, `update(self)` y `draw(self)`, así como el constructor de la clase que tiene los criterios esenciales de la clase:

```
def __init__(self):
     pyxel.init(224, 256, caption=”Donkey Kong”)
     pyxel.load(“Sprites.pyxres”)
     self.x = 0
     self.reset()
     pyxel.run(self.update, self.draw)
```

Este método constructor se puede resumir como una llamada a la pantalla de juego, empleando métodos que proporciona Pyxel.

Con “`pyxel.init`” se determinan las dimensiones del eje x (224) y eje y (256) que tiene el tablero y posteriormente se le da nombre “Donkey Kong”; `pyxel.load` nos permite cargar los Sprites que usamos recurrentemente durante el código. Por último, `reset` y `run` finalizan el constructor haciendo visible la pantalla de juego. Pasando a un mejor análisis de estos métodos:

```
def reset(self):
     self.DonkeyKong = DK(20, 56)
     self.mario = Mario(56, 232, 3)
     self.pauline = Pauline(88, 33)
     self.EsRampas = Rampas(0, 0)
     self.EsRampas.crearRampas()
     self.EsEscaleras = Escaleras(0, 0, 0)
     self.EsEscaleras.crearEscaleras()
     self.barriles = Barriles(0, 0)
```

Este método lo utilizamos cada vez que se abra el juego por primera vez. Carga todos los datos en su posición por defecto u original. Aquí creamos todo aquello que conforma el tablero, incluyendo los personajes, pasando los datos correspondientes de sus constructores. Añadir que se invoca a los métodos `crearRampas()` y `crearEscaleras()`; que serán utilizados de nuevo en el método `draw`.

```
def update(self):
     self.array_rampa = []
     self.vidas = []
     self.mario.mover(self.EsRampas, self.EsEscaleras)
     self.lista_escaleras = []
     self.barriles.rodar(self.EsRampas, self.EsEscaleras)
```

Update es utilizado para determinar las principales listas del código y los métodos de movimiento de Mario y los barriles

```
def draw(self):
     pyxel.cls(0)
     self.pauline.aparecer()
     self.mario.aparecer()
     self.DonkeyKong.draw()
     for      self.pos_escalera_x,      self.pos_escalera_y,      self.escalera_h      in
self.EsEscaleras.lista_escalera:
        pyxel.blt(self.pos_escalera_x,    self.pos_escalera_y,    0,    121,    237,    10,
self.pos_escalera_h, 3)

     for self.x,self.y in self.EsRampas.array_rampa:
         pyxel.blt(self.x, self.y, 0, 236, 103, 16, 8, 3)
      self.barriles.aparecerFijos()
      if(rango.frame_count > 65):
         self.barriles.aparecer()
     for self.vidas_x, self.vidas_y in self.mario.numVidas():
         pyxel.blt(self.vidas_x, self.vidas_y, 0, 241, 201, 8, 8, 3)
     pyxel.text(8, 8, "000000", 7)
     pyxel.text(88, 0, "HIGH SCORE:", 8)
     pyxel.text(96, 8, "000000", 7)
     pyxel.text(168, 24, "L = 01", 1)
```

El ultimo método de esta clase es el método draw. Este es el encargado, como dice su nombre, de dibujar todo lo que aparece en la pantalla. Utilizando la recursividad, este llama a otros métodos, como es el ejemplo para Mario, DK y Pauline, o utiliza bucles y dos instrucciones determinadas de pyxel.

`pyxel.blt` es un comando que funciona de la siguiente forma: `pyxel.blt(x,y,img,u,v,w,h,col)` en el que `x` e `y` son las coordenadas en la pantalla de pyxel en las que se muestra el Sprite, `img` la imagen del pyxeleditor a la que pertenece (puede ser 0, 1 o 2), `u` y `v` las coordenadas en el Sprites.pyxres del Sprite, con `w` siendo su anchura, y `h` su altura. El último dato es el color, de la paleta de pyxel, que tiene el fondo que rodea al Sprite al recordarlo, 3 siendo un color verdoso, transparente en nuestro caso. Las coordenadas corresponden al píxel superior izquierdo del Sprite. Por ejemplo; `pyxel.blt(self.x, self.y, 0, 236, 103, 16, 8, 3)` para las rampas.

`pyxel.text` tiene una estructura más sencilla. `Pyxel.text(x, y, s, col)` Se pasan las coordenadas `x` e `y` a la pantalla en la que pondrá determinado String `s` que se pase, en un color determinado (0-15)

Pasando al resto de clases, esenciales para la estructura del tablero y jugabilidad, cabe detallar algunas cosas. Como norma general, en todas las clases coinciden campos referidos a la posición en x y posición en y de los diferentes elementos referidos. Además, los métodos constructores de las clases siguen la misma estructura:

```
def __init__(self, x, y):
      self.x = x
      self.y = y
```

En los que van variando los nombres y/o valores de los campos; o aparecendo otros necesarios para cada clase

- *Class Rampa:*

Para esta clase encontramos 3 parámetros parámetros: “x, “y”, asignados a `self.x`, `self.y`, y `array_rampa`, una lista que contiene todas las rampas; que son incluidos en el constructor.

```
def __init__(self, x, y):
    self.x = x
    self.y = y
    self.array_rampa = []
```

En esta clase encontramos un solo método más: `def crearRampas(self)`

Este método, como su nombre indica, sirve para crear las rampas que guardamos en `array_rampa`. Esta lista tiene una longitud “`len(array_rampa)`” de 7, comienza con la rampa inferior, `array_rampa[0]` hasta la rampa superior, `array_rampa[6]`, donde se encuentra Pauline.

`Array_rampa[0]` y `array_rampas[5]` tienen una peculiaridad que las diferencian del resto de plataformas, pues tienen una parte plana (sin pendiente) y a partir de octavo elemento de la plataforma comienza con inclinación. Para el resto listas en `array_rampa` hemos tenido que calcular la posición en la que empieza la primera plataforma y a partir de ahí le sumamos + 1 a la posición del siguiente; mediante la utilización de bucles

- *Class Escalera:*

En esta clase encontramos en el método constructor el campo de la altura, puesto a que dependiendo de la posición en la que se encuentra la escalera tiene una altura distinta, además de funcionar para las partes en las que tenemos escaleras partidas.

```
def __init__(self, x, y, h):
    self.pos_escalera_x = x
    self.pos_escalera_y = y
    self.escalera_h = h
    self.lista_escalera = []
```

Otra vez, como en la clase Rampa, el único método que nos encontramos es `crearEscaleras(self)`. Este método nos sirve, como el propio nombre indica, crea las escaleras que son guardadas en `lista_escalera`. Dicha lista tiene una longitud de 4, cada uno de los elementos correspondiendo a una lista determinada de escaleras:

`lista_escalera[0]` guarda las escaleras que encontramos al lado de Donkey Kong y que suben hasta la plataforma en la que se encuentra Pauline.

`lista_escalera[1]` corresponde a una serie de escaleras que se encuentran en la misma posición del eje x, todas con la misma altura de 16 píxeles. Estas escaleras son, además, las escleras con mayor y menor posición en el eje x (32 y 182).

`Lista_escalera[2]` contiene el resto de las escaleras que no están en la pantalla ya o son aquellas partidas. Estas escaleras tienen un tamaño superior al de un solo Sprite, con lo que su tamaño total se consigue con la suma de dos Sprites por tramo.

Por último, `lista_escalera[3]` es una lista de todos los tramos de escaleras partidas que se encuentran a lo largo de la pantalla. Cada uno tiene una altura menor que 16, con lo que no solo variamos su posición, sino la longitud del Sprite también.

- *Class DK:*

En esta clase se pueden observar dos métodos, el método constructor recibe las posiciones x e y de Donkey Kong en la pantalla, junto a una variable `contador` para el siguiente método, denominado método `draw`.

En este método se utiliza el contador para cambiar el Sprite con ayuda de pyxel.frame_count. Se inicializa una variable que se iguala al módulo que resulta de dividir en número de frames que han aparecido en la pantalla entre 120; con este número y si es mayor o igual que 60, el `contador` del constructor suma una unidad. Cuando `contador` se iguala a uno; el Sprite de Donkey Kong que anteriormente aparecía estático, se modifica por otra imagen en la que aparece lanzando un barril.

- *Class Pauline:*

Estas clases es la más simple. Con tan solo dos métodos, el método constructor recibe la posición x e y; que se llama con posterioridad desde la clase principal, junto a otro método, `aparecer`. En el método se cogen las coordenadas introducidas por en la clase App y se devolverá su Sprite correspondiente, con ayuda de la instrucción de pyxel `pyxel.btl`. Pauline, aparecerá como consecuencia con un Sprite estático en la pantalla.

- *Class Mario:*

En esta clase encontramos el mayor número de métodos. El constructor ahora tiene un campo especial “vidas”, para llevar la cuenta de las vidas restantes que tiene Mario, parámetro que nos servirá de utilidad para otro método en esta clase.

```
def __init__(self, x, y, vidas):
    self.mario_x = x
    self.mario_y = y
    self.vidasMario = vidas
    self.array_rampa = []
```

En cuanto el resto de los métodos:
`def aparecer(self):`

Método que nos sirve para que Mario aparezca en pantalla y para ello utilizamos `self.mario_x`, `self.mario_y` para determinar su posición y cuatro Sprites distintos en función del movimiento que se le asigna a Mario desde el teclado; todos con altura 16 y anchura 12

`def numVidas(self)` en este método se lleva la cuenta del número de vidas que le restan a Mario, siendo el número de vidas que tiene Mario al inicio de la partida 3. Con este número en cuenta, se crea una lista “vidas” que guarda en 3 posiciones un Sprite correspondiente a cada vida, que luego se muestra por pantalla.

`def toparEscalera`. Este método es utilizado para comprobar si Mario está en el rango de una escalera. Para ello, utilizamos la variable booleana `HayEscalera`. El método busca en la lista de escaleras, y en caso de que la posición actual de Mario coincida o se encuentre en el rango de alguna de las escaleras, entonces devuelve `True`. En caso contrario, la respuesta es `False`. El método devuelve el valor que tome la variable `HayEscalera`.

`def hayRampa` es un método muy similar al que le precede, pero en este caso es utilizado para comprobar si Mario se encuentra con una rampa, utilizando en este caso la variable `HayRampa`, sin embargo, la utilidad de este método vendrá a la hora de comprobar de que Mario esté o no subiendo una escalera, puesto a que la respuesta de este método será `True` para cualquier instante en el que no haya una escalera cerca del rango de Mario

`def mover(self, rampas, escaleras)`

Este método servirá para que Mario pueda desplazarse a lo largo de la pantalla. Está compuesto por 4 principales opciones: Moverse a derecha, izquierda, arriba o saltar y caer. Para esto, utilizamos dos métodos definidos anteriormente (`toparEscalera` y `hayRampa`), creando las variables `HayRampa` y `HayEscalera` a partir de ellos.

Movimiento a la derecha `pyxel.btn(pyxel.KEY_RIGHT)`

Cuando pulsamos la tecla flecha derecha del teclado Mario se desplazará a la derecha, siempre y cuando no se pase del límite de la pantalla de 224 píxeles. Hay que tener en cuenta que Mario tiene una anchura, por lo que tenemos que restarle a este límite dicha anchura por tanto tenemos un límite de 212 píxeles.

Hay que tener en cuenta si Mario sube o baja la rampa. Si está bajando tenemos que comprobar que Mario está más alto que la siguiente rampa y por tanto tenemos que sumarle 1 a su posición actual. En caso contrario, cuando Mario está subiendo, tenemos que ver si la rampa está más alta que Mario y por tanto a la posición de Mario tenemos que restarle 1.

Movimiento a la izquierda `pyxel.btn(pyxel.KEY_LEFT)`

Cuando pulsamos la techa de la flecha izquierda, Mario se desplaza a la izquierda. Ocurre lo mismo que en el caso anterior, si encuentra un bloque de rampa que está por encima de él entonces sube, y si el bloque está por debajo entonces baja.

Movimiento hacia arriba `pyxel.btn(pyxel.KEY_UP) pyxel.btn(pyxel.KEY_SPACE)`

Mario puede moverse hacia arriba y saltar siempre y cuando no choque con una rampa. En caso de que choque, Mario no puede avanzar más y por la acción de gravedad baja hasta la rampa que tenga debajo. En caso de que haya una escalera, Mario sube la escalera en el tramo que exista. Para ello, utilizamos la variable booleana `HayEscalera`, que en caso de ser `False`, significa que Mario no puede atravesar la rampa, sin embargo, si es `True` entonces da "permiso" a Mario para atravesar la rampa.

Para que no atraviese los tramos de la plataforma, y parecido a lo que hacemos con las escaleras, creamos un bucle que compruebe que la posición de Mario coincide con la de la rampa + 16 y la altura de Mario +16 coincide con la y de la rampa.

Mover abajo (Gravedad)

Mario se desplazará hacia abajo (cuando no pulsamos ningún botón) siempre y cuando no esté encima de una plataforma (en tal caso se queda parado encima de la plataforma) y cuando no estemos pulsando el botón up (puesto que cuando sube, aumenta su posición -2 y cuando baja disminuye su posición +1, lo que hace que suba con velocidad 1). En caso de que Mario se encuentre subiendo una escalera, no descenderá.

- *Class Lista_barriles:*

El método constructor de esta clase inicializa una lista de barriles, que contiene a los barriles que se encuentran en la pantalla y un contador utilizado para otro método de la misma clase.

```
def __init__(self):
        self.barriles = []
        self.contador = 0
```

Junto a este método encontramos otros tres, que llaman a la clase posterior a esta, la clase barriles:

`def crearBarril(self)`, que vuelve a utilizar el contador de frames que se disponible con pyxel, jugando con la variable `contador`. A cada intervalo de 120 frames, en el caso de que 10 barriles no estén en la pantalla y por tanto la longitud de la lista no supera los 10 elementos, se crea un barril en su posición inicial (70, 78) que se añade a la lista de barriles, haciendo una llamada de recursividad a la clase Barril

`def aparecer(self)`. Este método hace aparecer a los barriles uno a uno, recorriendo la lista de barriles y haciendo pasar sus elementos, con otro uso de la clase Barril, que expone la aparición del barril; todo ello teniendo en cuenta la creación del barril, con una llamada al método anterior

`def rodar(self, rampas, escaleras):` En este último método de la clase se vuelve a recorrer la lista de los barriles, y para cada uno de los barriles se llama otra vez a la siguiente clase

- *Class Barril:*

El constructor contiene las posiciones del barril, una lista de barriles y una variable booleana `HayBarril` para determinar cuando aparece el barril.

```
def __init__(self, x, y):
     self.pos_barril_x = 70
     self.pos_barril_y = 78
```

`def aparecerFijos(self)`. Con este método aparecen los barriles situados en la parte izquierda de Donkey Kong. Estos barriles son estáticos.

`def aparecer(self)`. Este método corresponde a los barriles son aquellos que descienden por la pantalla, con el mismo Sprite todo el tiempo. Donkey Kong tiene que coger los barriles y lanzárselos a Mario. Mario tendrá que esquivar estos barriles. En el momento que el barril llega hasta el final del la última plataforma; el barril “desaparece”, séase, dejamos de enseñar el Sprite

`def hayRampa(self, rampas)`. Este método devuelve el valor de una variable booleana `HayRampa`. Con el uso de un bucle, recorremos la lista de las rampas, y si la posición + 12 píxeles en el eje y del barril coincide con el del tramo de la plataforma, `HayRampa` equivaldrá `True`, en caso contrario `HayRampa` será `False`. El método devuelve el valor que tome `HayRampa`

`def hayEscalera(self, escaleras)` Este método casi idéntico al método que le precede, pero en este caso la variable booleana es `HayEscalera`. Los valores respuesta dependen esta vez de que si el barril coincide en cierto rango dónde se encuentra una escalera.

`def rodar(self, rampas, escaleras)`. El método principal de toda la clase. Este método es esencial para simular el movimiento de los barriles por las plataformas y en ocasiones, hasta por las escaleras. Para esto usamos la recursividad y llamamos a los métodos `hayRampa` y `hayEscalera` con las variables `HayRampa` y `HayEscalera`

Si el barril se encuentra con una escalera tiene un 25% de posibilidades de bajar por ella, por lo que inicializamos una variable (`esBajar`) que nos da un número aleatorio que nos sirve para determinar si el barril baja la escalera o no. Primero tenemos que limitar el espacio en el que se puede mover el barril, conseguido con la siguiente principio de un bucle `if(self.pos_barril_x > 0 and self.pos_barril_x < 222):`

Una vez se ha definido este límite, entramos en las diferentes posibilidades de movimiento que tiene el barril:

```
if esBajar > 25:
      HayEscalera = False
```

En estas dos últimas líneas de código se expresa como el movimiento por las escaleras depende del booleano definido con anterioridad, que, si es mayor de 25, resultará en un negativo a la “existencia” de una escalera.

```
if HayEscalera == True:
        self.pos_barril_y +=1
```

Como se puede observar en esta parte de código, ante la existencia de una escalera, la posición del eje y del barril aumenta, simulando un movimiento de bajar por ella.

```
if HayRampa == True and HayEscalera == False:
            self.pos_barril_x += 1
            for i in range(0, len(rampas.array_rampa)):
                rampa = rampas.array_rampa[i]

                if (self.pos_barril_x + 12) in range(rampa[0], rampa[0] + 16) and
rampa[1] in range(self.pos_barril_y + 10, self.pos_barril_y+ 18):
                    self.pos_barril_y = rampa[1] – 12

                    if self.pos_barril_y in range(101, 134) or self.pos_barril_y in
range(165, 180) or self.pos_barril_y in range(228, 248):
                        self.pos_barril_x -= 2
```

Saliendo de la parte de las escaleras, si el barril se encuentra en una plataforma, rueda por ella variando su posición en x, aunque hay que tener en cuenta que algunas rampas están inclinadas. Este caso es muy similar al de Mario cuando el que baja o sube por la rampa.

Hay un momento en el que el barril llega al tope de una rampa y tiene que cambiar de dirección. Estos son el caso de la rampa 2, 4, y 6, en los cuales cambiaremos su velocidad. Si para bajar las primeras rampas tenía una velocidad de +1, para bajar al lado contrario la velocidad será de -2, y así compensa con una velocidad total de -1 al sentido de contrario.

```
elif HayRampa == False:
        self.pos_barril_y += 2
```

En caso de que no exista ningún trozo de rampa se encuentre debajo del barril, el desciende simulando un efecto de gravedad, haciendo que toque la plataforma que encuentra directamente debajo.

Este método terminará cuando el barril llegue al final de la última rampa.

Algoritmos utilizados

Hay una serie de recursos que utilizamos que importamos de la librería pyxel:

`pyxel.btn(key);` Utilizamos este para comprobar cuál es la techa que está pulsando el jugador en el teclado, y con ello desplazamos a Mario acorde a la acción que es pulsada, la tecla apareciendo en la parte entre paréntesis.

`pyxel.text y pyxel.blt`: Estas dos instrucciones explicadas con anterioridad en la memoria, nos permiten que aparezcan las imágenes en la pantalla y el texto, en posición y color que quiera el usuario.

`pyxel.frame_count`: Esto es un campo que siempre está cambiando, que sirve como medida del tiempo, puesto que cuenta los frames que han pasado desde el inicio de la aplicación. Se utiliza para calcular cuando se deben de soltar los barriles.

También importamos la librería random:

que utilizamos a la utilizamos para que cada barril tenga posibilidades de bajar por una escalera, con `random.randint(0, 99)`. Este genera un número aleatorio que, si es menor de 25 para nuestro código, significa que el barril descenderá por la escalera.

Dos algoritmos esenciales en el trabajo son los bucles for e if, elif y else. He aquí un ejemplo de su uso:

```
for i in rango(0, len(rampas.array_rampa)):
            rampa = rampas.array_rampa[i]
            if self.pos_barril_x in range(rampa[0]- 12, rampa[0] + 16) and
self.pos_barril_y +12 in range(rampa[1] – 4, rampa[1] +4):
                HayRampa = True
```

El bucle for utiliza una estructura de control de flujo en la que se controlan de antemano la cantidad de iteraciones mínimas que va a tener, pudiendo este número llegar a infinito.

Mientras tanto, el bucle if es un condicional, con el que se debe cumplir cierta condición para entrar en el bucle, como es que la posición x del barril coincida con la de la rampa. Si no se cumple dicho condición, se sale del bucle, cabiendo la posibilidad de entrar en uno nuevo, este denominándose “elif” o “else”; dependiendo si hay más o menos condiciones que cumplir.

Descripción del trabajo

El código que se le ha presentado corresponde a los primeros 3 Sprints a realizar de la práctica final.

El primer Sprint de la práctica sirve de base para el resto, ya que este corresponde a la creación de todas las clases necesarias para los personajes y objetos. Esto incluye los campos y métodos de cada una de ellas. En esta parte lo más difícil a la hora de programar fueron las escaleras, y principalmente las rampas, ya que había que ir calculando las posiciones concretas en las que se encuentra cada trozo de rampa o cada escalera, además de que son la base para colocar el resto de los personajes en sus posiciones.

El segundo Sprint gira alrededor del movimiento de nuestro héroe y protagonista, Mario. Hacerle moverse a izquierda y derecha no era tan difícil como cuando tenía que desplazarse rampas con inclinación y subir las escaleras o saltar. Puesto a que son dos cosas diferentes, y en una se puede caer por efeto de la gravedad, y en el otro no. Para simular mejor una visión del movimiento, se va cambiando el Sprite que se imprime en la pantalla en función del movimiento que sigue Mario. Al caer de una plataforma, Mario cae a la plataforma que tiene directamente debajo, simulándose un efecto similar al de la gravedad. A la hora de saltar, Mario “choca” con un trozo de rampa que tenga encima suya.

El tercer Sprint trata los barriles. Los barriles estáticos situados a la izquierda de Donkey Kong son los más sencillos de mostrar. En cuanto a los barriles móviles, la parte difícil es el movimiento que ellos tienen, defendiendo por la pantalla, y en ocasiones por las escaleras. Para evitar que siempre que se encuentra con una escalera el barril descienda, generamos un número aleatorio que nos ayuda a determinar si realmente puede o no bajar la escalera. En este sprint la parte más compleja es que apareciesen 10 barriles en la pantalla, generando la situación más difícil a la hora de codificar.

En este trabajo no se ha llegado a codificar de forma completa todos los Sprints que se pedían en la práctica. Falta, por tanto, la simulación del juego en la que Mario es capaz de sumar puntos si salta por encima de los barriles, animaciones para los personajes y los barriles, la muerte y pérdida de una vida de Mario en caso de colisionar con un barril, así como de los extras de añadir la música, efectos de sonido y el martillo.

A la hora de terminar el trabajo, se han simplificado al máximo los métodos para hacerlos más fáciles de leer.

Conclusión

Para esta práctica se ha usado el lenguaje de programación de Python, impartido en la asignatura, junto al uso de pyxel, una herramienta que se puede instalar como una librería. La herramienta es fácil de utilizar una vez has comprendido como trabajar con ella de forma eficiente. A la hora de usar el pyxeleditor, si no fuera por que se nos proporcionara con el archivo de Sprites.pyxres, hubiera sido bastante difícil el poder sacar los sprites para el juego; puesto a que las imágenes que se nos daban junto a la práctica se cortaban y no teníamos mucha idea al principio de como trabajar con ellas.

Ambos miembros de este equipo, tanto Ana como Miguel, hemos trabajado activamente con el tiempo que poseíamos para realizar la práctica, ya que es cierto que el límite de entrega ha sido más reducido este año, dando mayor inseguridad al ver el límite más cerca de lo planteado.

El tiempo ha contribuido en parte a que nuestro trabajo no salga del todo perfecto, puesto a que en estas últimas semanas organizar todas las asignaturas para hacer hueco a la práctica ha sido muy difícil, especialmente porque nuestros horarios no coincidían la mayoría del tiempo. El hecho de que en mitad del tiempo para realizar la práctica se incluyesen dos semanas de ejercicios para la misma semana no ayuda.

El resultado de nuestro trabajo no es perfecto, puesto a que tiene sus fallos, como los movimientos erráticos que tiene el barril, y los anti bajos que hemos sufrido, ha habido momentos de mucho entusiasmo cuando parte del código en la que llevábamos perfilando bastante tiempo funcionaba y tenemos una buena sensación de lo que ha salido de nuestra recreación de “Donkey Kong”. Ante todo, estamos contentos de como ha salido nuestro proyecto.