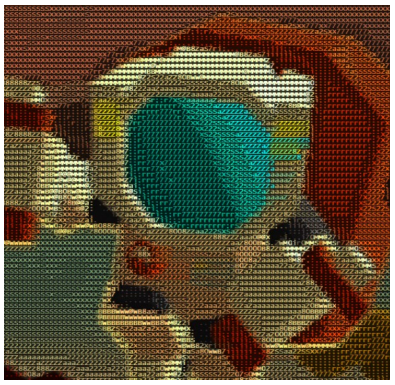
Proyecto FPII de Enrique Juan Gamboa

MARS: MARS - CMD Ed.



¿Qué es Mars:Mars?

Mars: Mars es un pequeño juego creado por Pomelo Games para Android e iOS en el que controlas un pequeño astronauta cuyo objetivo es moverse de una plataforma a otra. Para ello se vale de una pequeña mochila propulsora con la cual maniobrará por el aire, teniendo cuidado con aterrizar en una plataforma sin ir demasiado rápido contra el suelo. Eso sí, la mochila tiene una cantidad de combustible limitada, con que tendrá que tener en cuenta su consumo.

Gameplay trailer

Objetivos de esta versión

1. El nivel: la idea es utilizar la generación procedural que utilizamos en la Práctica 1 y editarla de manera que se pueda generar progresivamente el nivel, e implementar los métodos necesarios para que se pueda generar zonas llanas donde aterrizar.

2. Las físicas: se harán de trasfondo, cada frame, y se buscará una jugabilidad divertida antes de que sean realistas.

3. El renderizado: será muy simple, mostrará con qué cohete se está propulsando, el terreno y las plataformas. Debajo del renderizado del juego

se mostrará el número de la última plataforma a la que se llegó y el número máximo al que se ha llegado.

4. Puntuación y guardado: el juego guardará las puntuaciones junto al nombre de usuario que se use, y leerá todos los usuarios para averiguar quién tiene la puntuación más alta y enseñarla. Si un usuario que ya existe juega, comenzará donde se quedó la anterior partida.

Partida habitual

El juego empezará en el menú principal, donde se permitirá introducir un nombre con el que se cargará la partida (si existe) o creará una nueva.

Ya en el gameplay el jugador pulsará, desde el suelo, A o D para propulsarse a la izquierda o derecha. También podrá pulsar ambos botones para propulsarse rápidamente hacia arriba.
Tras el despegue, cada vez que el jugador pulse una de las teclas, su nivel de gasolina, representada debajo de la pantalla de juego, disminuirá.
Si el jugador aterriza con un descenso limpio y sin chocarse con nada, el mapa se desplazará hacia la izquierda para que el jugador pueda ver más mapa. En cambio, si el jugador aterrizara demasiado rápido o se chocara por el lateral, morirá y reaparecerá en su punto de partida anterior. Se sabrá si la velocidad de descenso es mortal si el color del astronauta cambia.

Una vez que el jugador quiera salir del juego, podrá entrar en el menú de pausa (pulsando P) para salir del juego, donde se guardará su progreso hasta la última llanura en la que aterrizó.

Representación interna

El juego a realizar contará, en principio, con 4 clases:

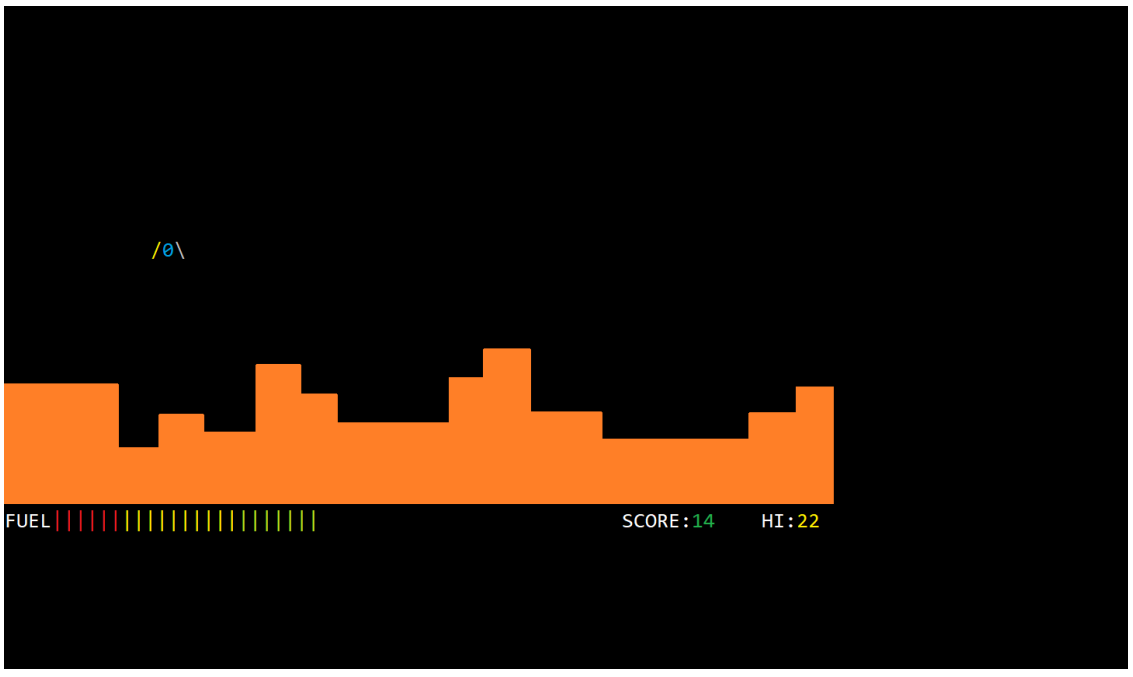
1. **Clase principal** (Program.cs): aquí se realizará todo el bucle de juego, la llamada a diferentes clases, el renderizado, etc. También se encargará del nivel de gasolina y de leer y guardar la puntuación.
   Esta clase utilizará variables simples como int y hará uso de arrays y structs para ordenar información y que quede un código más limpio.

2. **Listas:** a pesar de que se podría realizar el juego de manera sencilla sin usar listas, al requerirse en la práctica las utilizaremos para guardar la última plataforma a la que ha llegado el jugador y permitir retroceder.

3. **PhySick:** esta clase será el respaldo de todo el juego: va a ser el motor físico. Este motor contará con funciones para simular un campo gravitatorio dependiendo de la masa, aceleración, gravedad e incluso resistencia al aire de un objeto, una vez inicializado con los parámetros deseados.
   Va a contar con una amplia gama de funciones para simular movimiento, variables get/set de elementos como la masa y posición de objetos y funciones de propulsión física. Para todo ello hará un gran uso de structs y arrays.

4. **LvlGen:** esta clase podría introducirse en la clase principal, pero por orden y POO he optado por mantenerlo en una clase aparte.
   Esta clase está basada en una función que nos fue aportada para la práctica 1 de esta asignatura, con la que se generaba un nivel proceduralmente.
   En esta iteración será modificado para que sólo genere el suelo (ya que está programada para que genere una cueva) con parámetros como altura máxima e irregularidad de terreno y que sea capaz de generar terreno llano en intervalos para permitir que el astronauta aterrice.

Concepto de renderizado



Concepto de juego realizado en PhotoShop

FIN DE DESARROLLO

Mecánicas y controles:

Este juego ha acabado consistiendo en una mecánica casi idéntica a la propuesta original, variando únicamente en los siguientes aspectos:

1. El astronauta puede aterrizar donde sea, siempre que no vaya demasiado rápido
2. El astronauta no puede salir de los límites horizontales del mapa, pero sí del vertical
3. Salir del juego desde el juego (no pulsando detener compilación en VS) guarda automáticamente el juego
4. La representación del juego ha cambiado: si el astronauta va a impactar con el suelo fatídicamente, su fondo de renderizado se volverá rojo. También indica su nivel de gasolina a través del color de sus caracteres.

Los controles siguen la misma filosofía que los de la propuesta y del juego original:

- *A/flecha izquierda* controla el propulsor izquierdo (impulsando hacia la derecha), y
- *D/flecha derecha* el derecho (impulsando a la izquierda.)
- Si se pulsan ambas el juego lo registrará como *espacio/W/flecha arriba* (también habilitadas), que activará ambos propulsores.

En cuanto a interacciones con menús, permanecen los clásicos ESC para salir del juego y P para pausarlo.

Bucle de juego:

Al iniciar la partida, el menú principal aparece y se pide al jugador que introduzca su nombre para guardar su progreso. Tras pulsar enter, aparece directamente en el juego. El jugador deberá maniobrar el astronauta con WASD/flechas para poder avanzar por el nivel y aterrizar sin chocarse con los límites horizontales si demasiado rápido contra el suelo. Al aterrizar, el mapa avanzará junto con el astronauta para mostrar más parte del mundo por el que avanzar. Si el jugador muere, un mensaje de muerte aparecerá con su puntuación y permitirá reintentar desde su último aterrizaje efectivo. Si el jugador quisiera pausar la partida, dándole a la tecla P detendrá el juego y

mostrará un mensaje de pausa. Si le vuelve a dar a la misma tecla, se reanuda la partida.
Si el jugador decidiera dejar de jugar, pulsará la tecla ESC, que mostrará un mensaje de despedida con su puntuación, y se guardará la partida.

Descripción del código, representaciones, etc.

Querría empezar diciendo que la representación interna ha variado con respecto a la propuesta inicial, en concreto con el uso de listas(o la falta del mismo).
Las listas son representaciones muy potentes que permiten enlazar gran cantidad de datos concretos entre sí, son capaces de actuar como arrays sin necesidad de inicializar un tamaño y ordenan muchos elementos si el programa lo pidiera. Además permiten insertar, reemplazar, cambiar y combinar posiciones en sí mismas, lo que viene muy bien para una gran cantidad de programas y juegos.
El problema es que éste juego no es uno de ellos. Al ser este juego uno que cuenta muchísimo con datos "analógicos" (con muchos decimales) y que pide arrays limitados, no he sido capaz de poner en uso tan potente herramienta.

En cuanto al resto de sistemas:

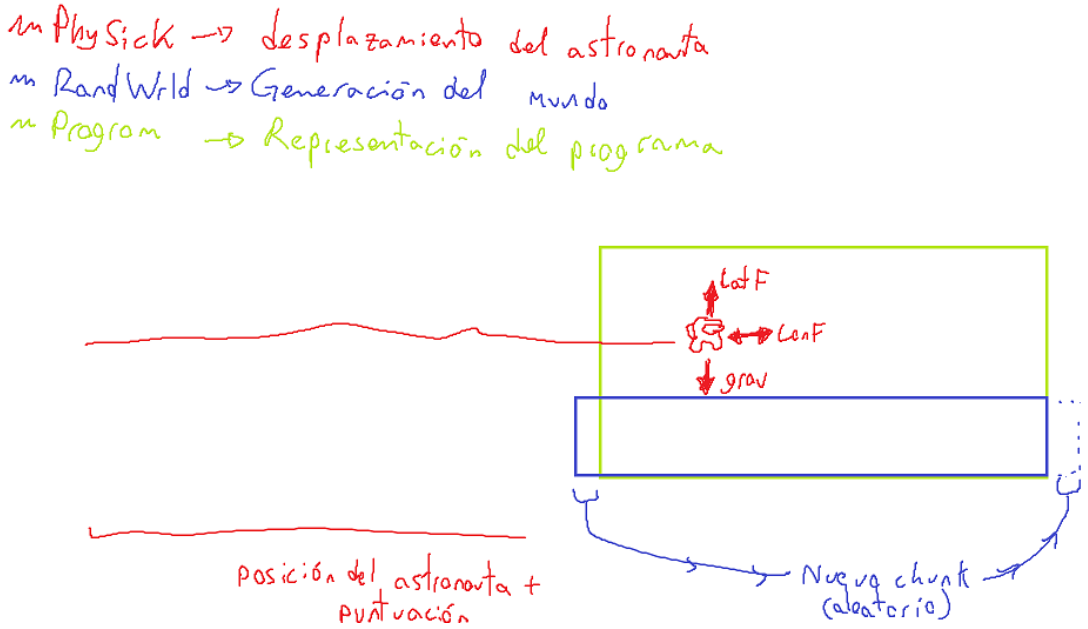
- **PhySick**, el motor de físicas que he creado para este juego (y sin duda el mayor quebradero de cabeza), cuenta con lo siguiente:
    - El "Mundo": con datos como la gravedad, resistencia del aire, etc. y los métodos que este sistema implementa, es muy sencillo aplicar las fuerzas del mundo al objeto que se describa en el sistema.
    - El "Objeto": también con sus propios datos, es afectado por el "Mundo", pero tiene métodos que le permiten recibir fuerzas vectoriales para contrarrestar efectos, cancelar todas las fuerzas que recibe e incluso se le puede modificar la posición a nuestro antojo.

  Este sistema está compuesto especialmente por una gran cantidad de datos tipo *float*, ya que se buscaba un cálculo preciso y movimiento (por lo menos interno) suave. Tiene varios métodos, y uno crucial que cuenta el tiempo para aplicar fuerzas progresivas.

- **RandWrld**, el generador del mundo (en la propuesta inicial, *LvlGen*) al final también se ha ocupado de su propio renderizado. Esto ha permitido, desde luego, una mayor limpieza del código. Además se ha creado un método que permite pedir el mundo representado en pantalla en el momento, principalmente para el cálculo de posiciones y colisiones.

- **Program**, el más cargado, es el que se encarga del bucle de juego principal, el renderizado y colisión del astronauta y de la lectura y escritura de datos. En cuanto a esto último, los métodos que lo realizan son capaces de crear el propio archivo y escribirlo, y si ya existe, sobreescribir datos si es el mismo jugador o añadir nuevos. Además puede detectar si se han corrompido, reseteando el progreso del jugador solicitante para impedir futuros problemas.

La representación interna es un poco compleja, ya que cada sistema que he creado usa la suya propia para no depender las unas de las otras y permitir su uso en otras aplicaciones en el futuro.
Debido a la complejidad de las mismas, creo que la mejor manera de explicarlas es mediante un esquema.



Como se puede comprobar, PhySick lleva la cuenta de la distancia total que ha recorrido el astronauta. Mientras tanto, a Program sólo le importa la distancia para obtener la puntuación, y para RandWrld no tiene ninguna relevancia. En cambio, Program y RandWrld cooperan mucho más, compartiendo dimensiones y datos de guardado mucho más que con PhySick.