95.11 – Trabajo Práctico N°1
Lunar Lander 40° aniversario

1. Objetivo del TP

El objetivo del presente trabajo consiste en la implementación de una reversión del juego Lunar Lander en lenguaje ISO-C99 utilizando la biblioteca gráfica SDL2. El trabajo consiste en la integración de los 3 ejercicios obligatorios ya realizados durante el curso además del desarrollo de algunas funcionalidades adicionales.

2. Alcance del TP

Mediante el presente TP se busca que el estudiante adquiera y aplique conocimientos sobre los siguientes temas:

- Programas en modo consola,
- Interacción asincrónica con el usuario,
- Funciones,
- Punteros,
- Memoria dinámica,
- Juego de caracteres ASCII,
- Directivas al preprocesador C,
- Modularización.

3. Introducción

3.1. Atari Lunar Lander

El arcade Lunar Lander fue desarrollado por Atari en el año 1979. El juego consistía en la misión de hacer aterrizar un módulo en la superficie de la luna.

El Lunar Lander fue el primer videojuego en utilizar la plataforma de hardware que luego fue popularizada por el Asteroids. Ésta tenía la particularidad de implementar una unidad de gráficos vectoriales que controlaba los movimientos del haz del monitor. El CPU del arcade era un Motorola 6502 de 8 bits y 1 MHz de velocidad.

Figura 1: Atari Lunar Lander.

Se trata de un juego sencillo en el cual se presenta en la pantalla un terreno escarpado y una nave que vuela sobre él. El objetivo es lograr hacer descender a la nave suavemente sobre determinadas áreas marcada en el terreno.

La nave se controla haciéndola rotar a izquierda y derecha y entregándole potencia a su tobera. Al introducir fichas en el juego se carga combustible en la nave y tanto rotar la nave como tener encendida la tobera consumen unidades de combustible. El juego se pierde al quedarse sin combustible.

Mientras haya combustible se tiene la misión de alunizar la nave. Si el alunizaje es suave, sin velocidad horizontal y con la nave vertical se ganan puntos. Dependiendo del inclumplimiento de las condiciones anteriores se podrá perder combustible o directamente destruir la nave. Cuando la nave aterriza o se destruye, el juego recomienza con la cantidad de combustible restante.

En el juego original había cuatro diferentes misiones con distintas dificultades según la variación de la fricción, el modo de control de la nave o la aceleración de la gravedad.

3.2. SDL2

SDL2 (Simple DirectMedia Layer versión 2) es una biblioteca gráfica multiplataforma desarrollada para C que provee una interfaz muy sencilla para la realización de programas gráficos.

No es objetivo de este curso interiorizarse en el funcionamiento de dicha biblioteca sino que se proveen plantillas para que el alumno desarrolle su implementación dentro de ella. Todo el trabajo se resuelve utilizando únicamente la primitiva de SDL2 de dibujado de una línea dados dos pares de coordenadas.

Las coordenadas en la pantalla se referencian por el número de pixel y algo importante a señalar es que suele ser convención de la computación gráfica (y SDL2 no es la excepción) ubicar la coordenada $(0, 0)$ arriba a la izquierda.

Si bien la coordenada de la pantalla se encuentra arriba a la izquierda, se pide implementar todas las funcionalidades referidas a la esquina de abajo a la izquierda, que es la coordenada natural que utilizamos habitualmente. Todos los archivos provistos por la cátedra definen el eje y positivo hacia arriba. Para realizar esta transformación teniendo una coordenada y natural

hay que computar `VENTANA_ALTO - y` y enviarle ese valor a SDL2.

La biblioteca SDL2 se provee en las distribuciones de GNU/Linux derivadas de Debian en el paquete `libsdl2-dev`. Para instalar el mismo basta con ingresar en la terminal la instrucción:

```
sudo apt-get install libsdl2-dev
```

Al compilar hay que indicarle al enlazador del GCC que enlace contra la biblioteca en cuestión, esto se hace mediante el *flag* `-lSDL2`.

4. Preliminares

El presente trabajo práctico debe estructurarse sobre las implementaciones ya realizadas hasta el momento. Por esto previo a enfocarnos en el desarrollo del trabajo propiamente dicho se mencionarán las implementaciones previas con las que ya se cuenta.

4.1. Ejercicios obligatorios

En alguno de los casos hay sutiles variaciones en los prototipos de las funciones que deberían ser completamente compatibles con las implementaciones ya desarrolladas. Se utilizarán las interfases detalladas en este enunciado.

Se cuenta con las siguientes funciones:

4.1.1. Ejercicio Obligatorio Nº1

Se tienen las funciones de evolución de velocidad y movimiento:

```
double computar_velocidad(double vi, double a, double dt);
double computar_posicion(double pi, double vi, double dt);
```

4.1.2. Ejercicio Obligatorio Nº2

Se tienen las funciones de manipulación de vectores de coordenadas:

```
bool vector_esta_arriba(float **v, size_t n, float x, float y);
void vector_trasladar(float **v, size_t n, float dx, float dy);
void vector_rotar(float **v, size_t n, double rad);
```

Notar que se modificó la interfaz para trabajar sobre vectores dinámicos.

4.1.3. Ejercicio Obligatorio Nº3

Se tienen las funciones de manejo de vectores dinámicos:

```
float **vector_desde_matriz(const float m[][2], size_t n);
void vector_destruir(float **v, size_t n, size_t m);
float **vector_densificar(float **v, size_t nv, size_t nn, float margen);
```

4.2. Ejercicio de diccionarios

Adicionalmente a los ejercicios obligatorios se indicó la confección de un diccionario capaz de traducir un carácter ASCII en un vector de coordenadas (enteras) y su longitud.

A tal fin se proveen los archivos `caracteres.c` y `caracteres.h` que definen un vector de determinado tamaño para cada uno de los caracteres. En los mismos se encuentran descriptas las letras de la A a la Z, los números del 0 al 9, flechas hacia izquierda, derecha, arriba, y abajo y el espacio.

El siguiente es un extracto del archivo `caracteres.h`:

```
extern const int caracter_a[7][2];
extern const int caracter_b[12][2];
extern const int caracter_c[4][2];
extern const int caracter_d[7][2];
extern const int caracter_e[7][2];
extern const int caracter_f[6][2];
```

Deben mapearse un diccionario que permita convertir los ASCIIs `'A'...'Z'`, `'0'...'9'`, `'<'`, `'>'`, `'^'`, `'v'`, `'␣'` en su correspondiente vector y longitud. Es decir, de alguna manera debe mapearse la `'F'` en un puntero a `caracter_f` y en la longitud `6`.

No se permite bajo ningún punto de vista replicar el contenido de los archivos en otro archivo. La traducción y definición del diccionario debe realizarse exclusivamente mediante la inclusión de `caracteres.h`.

5. Desarrollo del TP

5.1. Funcionamiento del juego

Se cuenta con un archivo `config.h` provisto por la cátedra, en el mismo se encuentran las siguientes definiciones:

```
#define VENTANA_ANCHO 1000
#define VENTANA_ALTO 800

#define JUEGO_FPS 100
#define JUEGO_COMBUSTIBLE_INICIAL 9999
#define JUEGO_COMBUSTIBLE_RADIANES 1
#define JUEGO_COMBUSTIBLE_POT_X_SEG 1

#define G 5

#define NAVE_ROTACION_PASO (PI/20)
#define NAVE_MAX_POTENCIA (3*G)

#define NAVE_X_INICIAL 100
#define NAVE_Y_INICIAL (VENTANA_ALTO-100)
#define NAVE_VX_INICIAL 100
#define NAVE_VY_INICIAL 0
#define NAVE_ANGULO_INICIAL (PI/2)
#define NAVE_POTENCIA_INICIAL 0
```

El juego se inicial con una cantidad `JUEGO_COMBUSTIBLE_INICIAL` de combustible (la cual no puede ser nunca superior a 9999) y se termina cuando el combustible llega a cero.

Mientras haya combustible se jugarán "partidas" del juego. En cada partida se iniciará con la nave rotada `NAVE_ANGULO_INICIAL` estando la nave en la posición (`NAVE_X_INICIAL`, `NAVE_Y_INICIAL`) de

la pantalla e impulsándose a una velocidad de (`NAVE_VX_INICIAL`, `NAVE_VY_INICIAL`) con su pontencia en `NAVE_POTENCIA_INICIAL`. Cada partida se iniciará con un terreno aleatorio nuevo.

Cuando el usuario presione las teclas izquierda o derecha la nave rotará `NAVE_ROTACION_PASO` antihorarios u horarios respectivamente hasta los valores máximos de $\pm\frac{\pi}{2}$ (es decir, la nave nunca podrá estar cabeza abajo). Por cada radián que la nave rotara se consumirá una cantidad `JUEGO_COMBUSTIBLE_RADIANES` de combustible. Cuando el usuario presione las teclas arriba o abajo la potencia deberá incrementarse o decrementarse en una unidad respectivamente hasta alcanzar el valor máximo de `NAVE_MAX_POTENCIA` o el mínimo de 0. En cada paso de tiempo se consumirá combustible a razón de `JUEGO_COMBUSTIBLE_POT_X_SEG` multiplicadas por la potencia unidades de combustible por segundo.

Durante el desarrollo de la partida la nave se desplazará libre por la pantalla con una gravedad de `G` en el eje $-y$. Además habrá una aceleración igual a la potencia que se aplicará impulsando a la nave según su sentido de rotación (eg. cuando la nave esté vertical, con 0 grados, la fuerza será hacia arriba). La coordenada x deberá estar siempre entre 0 y `VENTANA_ANCHO`, si sobrepasara estos valores deberá reaparecer del otro lado (ie. el mundo es cilíndrico).

La partida se terminará cuando la nave pase por debajo del terreno (restarle a la coordenada y de la nave la distancia a la tobera para considerar –imperfectamente– el diámetro de la nave). Hay tres posibilidades al llegar al terreno:

Buen aterrizaje: La nave desciende con $|v_x| < 1$, con $|v_y| < 10$ y $|\varphi| < 0{,}01$. Se le deben sumar 50 puntos al usuario e imprimir la leyenda `"YOU␣HAVE␣LANDED"` en la pantalla.

Aterrizaje violento: La nave desciende con $|v_x| < 2$, $|v_y| < 20$ y $|\varphi| < 0{,}01$. Se le deben sumar 15 puntos al usuario e imprimir la leyenda `"YOU␣LANDED␣HARD"`.

Destrucción: Si no es una de las anteriores. Se le suman 5 puntos al usuario, se le quitan 250 unidades de combustible y se imprime la leyenda `"DESTROYED"`.

Al terminar el juego informar al usuario por `stdout` del puntaje alcanzado.

5.2. Visualización del juego

A lo largo del juego se debe dibujar a la nave en la posición de la pantalla que le corresponda, con su ángulo correspondiente y el chorro de la misma proporcional a la potencia suministrada en ese instante y con su ángulo correspondiente.

Durante todo el desarrollo del juego debe aparecer en la parte superior de la pantalla la información del juego la siguiente información alineada de manera similar a la del juego original (ver la figura 1):

Puntaje: La leyenda `"SCORE"` seguida del puntaje entero, con 4 dígitos, completando con ceros.

Tiempo: La leyenda `"TIME"` seguida del tiempo total (de la partida) como un entero de 4 dígitos completando con ceros.

Combustible: La leyenda `"FUEL"` seguida del combustible como un entero de 4 dígitos completando con ceros.

Altura: La leyenda `"ALTITUDE"` seguida de la distancia vertical al terreno como un entero de 4 dígitos alineados a derecha.

Velocidad en x: La leyenda `"HORIZONTAL␣SPEED"` seguida del módulo de v_x con 4 dígitos enteros alineados a derecha, seguido de un espacio, seguido de una flecha hacia la derecha si v_x es positiva y a la izquierda si es negativa.

Velocidad en y: La leyenda `"VERTICAL␣SPEED"` seguida del módulo de v_y con 4 dígitos enteros alineado a derecha, seguido de un espacio, seguido de una flecha hacia arriba si v_y es positiva y hacia abajo si es negativa.

5.3. Bloque principal

Se provee un archivo `main.c` el cual debe ser usado de base para la elaboración del trabajo práctico. En el mismo se delimitan los bloques dentro de la función `main()` en los cuales el alumno debería introducir código. Los códigos que se encuentran dentro de esos bloques son códigos de ejemplo y pueden (deben) ser borrados por el alumno al desarrollar su implementación.

Es importante destacar que no hacen faltan conocimientos de programación gráfica para la realización del presente trabajo. Todas las rutinas que tienen que ver con la inicialización y manejo de cosas gráficas se proveen ya implementadas. El alumno sólo deberá manejar la primitiva de SDL2 `SDL_RenderDrawLine(renderer, x0, x1, y0, y1);` la cual dibuja una recta entre esas coordenadas.

El código provisto se encarga de inicializar todo lo referido a SDL2. Luego se entra en un bucle en el cual en cada iteración se chequea si el usuario interactuó con la ventana (cerrándola o mediante el teclado) para luego dar paso a las rutinas de dibujado de la pantalla, antes de la siguiente iteración se hace una pausa para garantizar que el juego funcione a la velocidad predefinida. Al terminarse el bucle principal se liberan los recursos asociados a SDL2[1].

Dentro del `main()` provisto se considera que el alumno debe intervenir sólo dentro de las zonas delimitadas:

- Una zona al comienzo, para declarar y definir todas sus variables de estado.
- Una zona dentro de la rutina de chequeo de teclado para modificar variables de estado en función de la entrada del usuario.
- Una zona dentro de la rutina de dibujado de ventana para evolucionar las variables de estado y dibujar todo lo referido a la parte gráfica.
- Una zona al finalizar el bucle principal para liberar los recursos asociados a sus variables de estado.

En caso de necesitar realizar una pausa durante el juego (por ejemplo, para imprimir un mensaje y que esté presente más de un `DT`) se encuentra disponible la variable `dormir`. Si se define a la misma con una cantidad de milisegundos el programa hará esa pausa.

Está prohibido realizar modificaciones en `main()` por fuera de los bloques señalizados pero está permitido agregar en el archivo inclusiones, definiciones, implementación de funciones y todo lo que haga falta para completar el trabajo.

Figura 2: Tipografía suministrada.

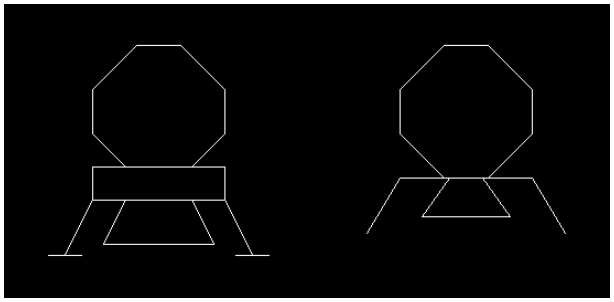

Figura 3: Naves suministradas.

5.4. Tipografías

Como se explicó en la sección 4.2 se proveen tipografías representadas como `const int[][2]`. Los caracteres ocupan un espacio de `CARACTER_ANCHO` por `CARACTER_ALTO` unidades. La coordenada cero se encuentra abajo a la izquierda del carácter.

Las tipografías provistas representan los caracteres vectoriales originales del Lunar Lander (figura 2) además de las flechas ya mencionadas. Para obtener una tipografía similar a la del juego original (figura 1) se recomienda escalar las coordenadas por un factor de 2.

Se debe implementar una función que reciba (al menos) una cadena de caracteres, la posición y la escala y dibuje dicha cadena en la pantalla.

Siendo que las rutinas de impresión deben ser rápidas, no realizar copias de los caracteres en memoria no constante, y realizar las transformaciones necesarias para escalar y posicionar en el mismo acto de dibujarlos.

5.5. Naves

Se suministran dos vectores que representan las dos naves originales del Lunar Lander contenidos en los archivos `naves.c` y `naves.h`.

Si bien en el juego original se realizaba un zoom al área de interés al realizar el alunizaje, por lo que se pasaba de la `nave_chica` a la `nave_grande` que poseía más detalle, en la simplificación de este trabajo se trabajará siempre en la misma escala y se puede usar la nave de preferencia.

5.6. Generación del terreno

Para generar los terrenos aleatorios debe usarse la función de densificación ya desarrollada. La idea es partir de un terreno estático, fijo y constante y agregar puntos intermedios hasta lograr una forma arbitraria.

[1] Si bien se liberan los recursos, la biblioteca tiene detalles de implementación interna por lo que herramientas como Valgrind alertarán de fugas de memoria. Estas fugas son responsabilidad de los desarrolladores de SDL2 y está fuera de nuestro alcance corregirlas.

Siendo que la generación de buenos terrenos no es trivial, se provee una implementación funcional que genera terrenos más o menos buenos, totalmente aleatorios y sin la necesidad de definir plantillas estáticas complicadas.

Esta implementación es sucia y tiene muchos números mágicos dado que fue calibrada hasta obtener un resultado satisfactorio:

```c
// Devuelve un vector que representa el terreno y su tamaño n.
float **crear_terreno(size_t *n) {
  *n = 0;

  const float terreno_estatico[][2] = {
    {0, 100},                       // izquierda
    {-1, VENTANA_ALTO * 2.0 / 3},   // "medio"
    {VENTANA_ANCHO, 100}            // derecha
  };

  size_t nt = 3;
  float **terreno = vector_desde_matriz(terreno_estatico, nt);
  if(terreno == NULL) return NULL;

  // Asignamos la coordenada del medio aleatoriamente:
  terreno[1][0] = rand() % VENTANA_ANCHO;

  // Iterativamente densificamos 30 veces achicando el margen cada vez:
  for(size_t i = 1; i < 30; i++) {
    float **aux = vector_densificar(terreno, nt, 2 * (i + 5), 100 / i);
    vector_destruir(terreno, nt, 2);
    if(aux == NULL) return NULL;
    terreno = aux;
    nt = 2 * (i + 5);
  }

  *n = nt;
  return terreno;
}
```

Si el alumno quiere puede implementar una estrategia propia, pero es libre de usar esta implementación.

6. Consideraciones generales

- Si bien parte importante del trabajo se encuentra resuelta por trabajos anteriores, el mismo presenta una extensión considerablemente mayor a la de los enunciados precedentes. Se recomienda releer el enunciado las veces que sean necesarias para extraer toda la información contenida en el mismo.
- El programa debe estar modularizado, las funciones tienen que estar agrupadas por funcionalidad y además debe haber un diseño de funciones correcto (funciones sencillas, claras, genéricas, que reciban los parámetros justos, etc.)
- Se recomienda sistematizar el trabajo para poder implementar pequeñas partes que puedan ser probadas. El trabajo tiene muchísimas partes independientes entre sí que pueden abordarse de a una por vez.

Se deben entregar:

- Los códigos fuentes del proyecto completo.
- El archivo Makefile para compilarlo.
- Un pequeño informe que explique sintéticamente cómo se diseñó la solución, de qué forma se diseñó, consideraciones que se tomaron, etc.

7. Entrega

Fecha de entrega: Domingo 19 de mayo.

La entrega se realiza por correo a la dirección `algoritmos9511entregas` en `gmail.com` (reemplazar en por arroba).

El trabajo práctico es de elaboración individual.