Organización del Computador II

Departamento de Computación,
Facultad de Ciencias Exactas y Naturales,
Universidad de Buenos Aires

Trabajo Práctico 1: "Oportuncrisis"

Primer Cuatrimestre de 2009

Grupo "UNPCKHPD"

Pablo Antonio	290/08	pabloa@gmail.com
Pablo Herrero	332/07	pablodherrero@gmail.com
Estefanía Porta	451/04	estef.barbara@gmail.com

Índice

1. Introducción **1**
1.1. ¿Qué se muestra por pantalla? . . . 1
1.2. Los *sprites* . . . 2
1.3. Los fondos . . . 2
1.4. El *color-off* . . . 2
1.5. El *plasma* . . . 3
1.6. La lista y el iterador . . . 3

2. Desarrollo **3**
2.1. Funciones desarrolladas . . . 3
2.1.1. Función `recortar` . . . 3
2.1.2. Función `blit` . . . 4
2.1.3. Función `generarFondo` . . . 5
2.1.4. Función `generarPlasma` . . . 6
2.1.5. Lista . . . 7
2.1.6. Iterador . . . 9
2.2. Basura en los bitmaps . . . 10
2.3. Manejo de la pila (`entrada_funcion` y `salida_funcion`) . . . 10
2.4. Recepción de parámetros . . . 11
2.5. Reserva y liberación de memoria (`malloc` y `free`) . . . 11
2.6. Acceso a variables globales . . . 12

3. Resultados **12**
3.1. *Debugging* . . . 12
3.2. El problema de la basura en los bitmaps . . . 12

4. Conclusiones **12**

1. Introducción

El trabajo práctico consiste en programar, en lenguaje *assembly*[1] de la *arquitectura x86 de Intel*, ciertas funciones de un videojuego. La lógica del videojuego se halla programada en C/C++, y utiliza un número de funciones que, en principio, no se encuentran implementadas. Nuestra tarea fue implementar dichas funciones.

Las funciones faltantas se encargan de implementar:

- las funcionalidades básicas de una *lista doblemente enlazada*
- las funcionalidades de un *iterador* que sirva para recorrer dicha lista
- una función `recortar`, encargada de, a partir de un *sprite* que contiene varias instancias de un objeto visual, y dados los parámetros necesarios, quedarse con una instancia de este, y depositarla en un lugar específico con la orientación adecuada
- una función `blit` que deberá, dada una imagen con un fondo de un color dado (el *color-off*) y otra imagen, reemplazar en la primera los pixeles de dicho color por los pixeles correspondientes en la segunda imagen, a partir de un par de coordenadas especificado
- una función `generarFondo`, capaz de, a partir de una imagen para el fondo y una coordenada x en este, dibujarlo en pantalla tomando como inicio del fondo la coordenada especificada. De no contar con suficiente ancho en el fondo a partir de dicha coordenada como para llenar el ancho de la pantalla por completo, deberá tomar como valor de dicha coordenada, el valor máximo suficiente que permita llenar la pantalla.
- una función `generarPlasma`, encargada de crear un efecto visual llamado "plasma", y de reemplazar en la pantalla ciertos pixeles de un color, por los correspondientes a este efecto

Se utilizaron, a lo largo del trabajo, *bitmaps* de 24 bits de color. Tanto la pantalla como las imágenes cargadas se recorrieron como una matriz de pixeles, donde cada pixel estaba representado por tres bytes, uno para cada componente RGB del color a mostrar.

El juego hace uso de la *biblioteca multiplataforma libre SDL*.

1.1. ¿Qué se muestra por pantalla?

Muy básicamente, el juego consta de un ciclo que se repite todo el tiempo:

```
1 procesar eventos de entrada
2 actualizar posiciones, estados, etc.
3 chequear IA, colisiones, física, etc.
4 mostrar resultados por pantalla
5 ira a 1.
```

[1] Se optó por el ensamblador NASM, y se escribió el código en el lenguaje que este soporta.

En la etapa 4 ("mostrar resultados por pantalla"), que es la que nos atañe en este trabajo, la secuencia que se sigue es la siguiente:

```
1 generar el fondo actual (generarFondo)
2 generar el plasma, usando como color-off el color del
      cielo del fondo (generarPlasma)
3 para cada sprite de cada personaje:
4     recortar la instancia que se quiere del personaje
5     aplicar blit (cambiar el color-off del personaje
          por lo que haya en la pantalla)
```

Tener este esquema en mente nos ayudará a entender mejor las explicaciones que vienen a continuación. A su vez, las explicaciones que vienen a continuación, completarán y darán sentido al esquema.

1.2. Los *sprites*

Un *sprite*, para nosotros, es una imagen compuesta por una tira de figuras que se utilizan para dar movimiento a los personajes. En nuestro caso, tenemos un sprite para cada una de las animaciones de los movimientos de Wolverine (parado, caminando y atacando) y Gambit (parado).

Para simular el movimiento de los personajes, se muestran por pantalla una a una las figuras de la tira, en la misma posición (o con algún desplazamiento si se quiere mostrar que el personaje se traslada). El usuario observa un movimiento por parte de los personajes pues el ojo humano no alcanza a percibir que en realidad se trata de varias figuras una detrás de la otra.

1.3. Los fondos

En nuestro juego, tenemos dos fondos; uno para cada escenario. Los fondos tienen un cierto ancho, en nuestro caso mayor al de la pantalla. Inicialmente se muestra en la pantalla la primera parte del fondo (observándolo de izquierda a derecha). A medida que el personaje avanza hacia un lado u otro de la pantalla nuevas porciones del fondo se hacen visibles, generando la sensación de que el personaje está recorriéndolo.

1.4. El *color-off*

Siendo bitmaps, lógicamente las imágenes de los personajes y las de los fondos tienen un color asignado a cada pixel que los conforma. Sin embargo, para que el movimiento del personaje a través del escenario sea más realista, preferiríamos que ciertas zonas de su bitmap no se mantengan estáticas, sino que vayan cambiando conforme al contexto que lo rodea en cada momento. Además, también puede sernos útil que el mismo fondo se comporte de manera similar; por ejemplo, podríamos querer que en cierta parte del fondo designada al cielo las nubes vayan moviéndose, en lugar de que se mantenga estática.

Para lograr esto se elige un color en el bitmap al cual se lo llama "color-off". Simplemente se trata de un color RGB que no será utilizado en el bitmap sino para designar pixeles que serán reemplazados por otros. En el caso del personaje, estos pixeles podrán reemplazarse por pixeles correspondientes al fondo.

1.5. El *plasma*

Para colaborar con la ambientación de las escenas, se utilizó un fondo de *plasma*. Básicamente, se trata de una imagen que se genera a partir de una función senoidal y ciertas designaciones de colores a los puntos. El efecto resultante podría describirse como una serie de círculos y circunferencias de distintos colores (en nuestro caso, mayormente verde azul y negro) agrandándose, achicándose y trasladándose a lo largo de la pantalla.

1.6. La lista y el iterador

Para disponer de personajes estáticos (que no se trasladan) en la escena que, a su vez, pueden desaparecer de ella (por ejemplo, al ser atacados por nuestro personaje principal), puede disponerse de una lista que los contenga, de manera que pueda eliminárselos de ella en caso de necesitarlo.

Para recorrerla con facilidad, se construyó también un iterador muy simple que permite atravaserla.

2. Desarrollo

2.1. Funciones desarrolladas

2.1.1. Función `recortar`

```
void recortar(Uint8* sprite, Uint32 instancia, Uint32 ancho_instancia,
Uint32 ancho_sprite, Uint32 alto_sprite, Uint8* res, bool orientacion);
```

Parámetros:

- `sprite` es un puntero a la imagen que contiene el sprite
- `instancia` es el número correspondiente a la figura que se quiere obtener recortando del sprite
- `ancho_instancia` es el ancho de cada una de las instancias (figuras) en el sprite (asumimos que todas las instancias tienen un mismo ancho)
- `ancho_sprite` es el ancho de la imagen del sprite
- `alto_sprite` es la altura de la imagen del sprite
- `res` es un puntero a un espacio en memoria reservado para depositar la imagen recortada

- `orientacion` indica la orientación que tendrá la imagen recortada (si el valor de orientación así lo indica, la figura recortada se mostrará orientada hacia el lado contrario al que aparece en el sprite)

Archivo en el que se halla la función: `src/asm/recortar.asm`

Pseudocódigo:

```
void recortar(Uint8* sprite, Uint32 instancia, Uint32 ancho_instancia, Uint32
    ancho_sprite, Uint32 alto_sprite, Uint8* res, bool orientacion) {

    Color *pos_sprite = ((Color*) sprite) + (instancia * ancho_instancia),
    *pos_res = (Color*) res;

    int basura_sprite = calcular_basura(ancho_sprite),
        basura_instancia = calcular_basura(ancho_instancia),
        sentido = 1, defasaje = 0;

    if (orientacion == false) {
        sentido = -1;
        defasaje = ancho_instancia -1;
    }

    for (Uint32 i = 0; i < alto_sprite; i++) {
        Color* comienzo = pos_sprite;
        pos_sprite += defasaje;

        for (Uint32 j=0; j < ancho_instancia; j++, pos_sprite+=sentido, pos_res++)
             {
            copiar_color(pos_res,pos_sprite);
        }

        pos_sprite = ajustar_arreglo(comienzo + ancho_sprite,basura_sprite);
        pos_res = ajustar_arreglo(pos_res,basura_instancia);
    }
}
```

2.1.2. Función `blit`

```
void blit(Uint8 *image, Uint32 w, Uint32 h, Uint32 x, Uint32 y, Color
rgb);
```

Parámetros:

- `image` es un puntero a la imagen que se desea tratar
- `w` es el ancho de la imagen
- `h` es la altura de la imagen
- `x` e `y` son las coordenadas de la posición en la pantalla a partir de la cual se obtendrán los datos para reemplazar en la imagen
- `rgb` representa al color-off, es decir, el color en el que están pintados en la imagen los pixeles a ser reemplazados

Archivo en el que se halla la función: `src/asm/blit.asm`

Pseudocódigo:

```
void blit(Uint8 *image, Uint32 w, Uint32 h, Uint32 x, Uint32 y, Color rgb) {

    Color *comienzo = screen_pixeles + y * SCREEN_WIDTH + x,
    *pos_buff = (Color*) image;
    int basura = calcular_basura(w);

    for (Uint32 i = 0; i < h; i++) {
        Color *actual = comienzo;

        for (Uint32 j = 0; j < w; j++, actual++, pos_buff++) {

            if (color_igual(pos_buff,&rgb))
                copiar_color(pos_buff,actual);
            }

            pos_buff = ajustar_arreglo(pos_buff,basura);
            comienzo += SCREEN_WIDTH;
    }
}
```

2.1.3. Función `generarFondo`

```
void generarFondo (Uint8 *fondo, Uint32 fondo_w, Uint32 fondo_h, Uint32
screenAbsPos);
```

Parámetros:

- `fondo` es un puntero al lugar en el que se encuentra el fondo cargado en memoria
- `fondo_w` es el ancho de la imagen del fondo
- `fondo_h` es la altura de la imagen del fondo
- `screenAbsPos` es la posición en X del fondo a partir de la cual se quiere comenzar a dibujar

Archivo en el que se halla la función: `src/asm/generarFondo.asm`

Pseudocódigo:

```
void generarFondo (Uint8 *fondo, Uint32 fondo_w, Uint32 fondo_h, Uint32
    screenAbsPos) {

    if (screenAbsPos > fondo_w - SCREEN_WIDTH)
        screenAbsPos = fondo_w - SCREEN_WIDTH;

    int basura_fondo = calcular_basura(fondo_w);
    Color *pos_screen = screen_pixeles, *pos_fondo = ((Color*) fondo) +
        screenAbsPos;

    for (Uint32 i = 0; i < SCREEN_HEIGHT; i++) {
        Color* comienzo = pos_fondo;

        for (Uint32 j = 0; j < SCREEN_WIDTH; j++, pos_screen++, pos_fondo++)
            copiar_color(pos_screen, pos_fondo);
```

```
        pos_fondo = ajustar_arreglo(comienzo + fondo_w,basura_fondo);
    }
}
```

2.1.4. Función `generarPlasma`

```
void generarPlasma(Color rgb)
```

Parámetros:

- `rgb` es el color del que estarán pintados los pixeles que serán reemplazados por el efecto de plasma

Archivo en el que se halla la función: `src/asm/generarPlasma.asm`

Pseudocódigo:

```
void generarPlasma (Color rgb)
{
  int x;
  for (int i = 0; i < SCREEN_HEIGHT; i++)
  {
     for (int j = 0; j < SCREEN_WIDTH; j++)
     {
         x = colores[(g_ver0 + 5*j) %512] +
           colores[(g_ver1 + 3*j) %512] +
           colores[(g_hor0 + 3*i) %512] +
           colores[(g_hor1 + i) %512];

         Uint8 index = 128 + (x >> 4);

         if (color_igual(&screen_pixeles[i * SCREEN_WIDTH + j],&rgb))
         {
             if (index < 64) {
                 screen_pixeles[i * SCREEN_WIDTH + j].r = 255 - ((index << 2) +
                     1);
                 screen_pixeles[i * SCREEN_WIDTH + j].g = index << 2;
                 screen_pixeles[i * SCREEN_WIDTH + j].b = 0;
             } else if (index < 128) {
                 screen_pixeles[i * SCREEN_WIDTH + j].r = (index << 2) + 1;
                 screen_pixeles[i * SCREEN_WIDTH + j].g = 255;
                 screen_pixeles[i * SCREEN_WIDTH + j].b = 0;
             } else if (index < 192) {
                 screen_pixeles[i * SCREEN_WIDTH + j].r = 255 - ((index<<2) + 1);
                 screen_pixeles[i * SCREEN_WIDTH + j].g = 255 - ((index<<2) + 1);
                 screen_pixeles[i * SCREEN_WIDTH + j].b = 0;
             } else if (index < 256) {
                 screen_pixeles[i * SCREEN_WIDTH + j].r = (index << 2) + 1;
                 screen_pixeles[i * SCREEN_WIDTH + j].g = 0;
                 screen_pixeles[i * SCREEN_WIDTH + j].b = 0;
             } else if (index >= 256) {
                 screen_pixeles[i * SCREEN_WIDTH + j].r = 0;
                 screen_pixeles[i * SCREEN_WIDTH + j].g = 0;
                 screen_pixeles[i * SCREEN_WIDTH + j].b = 0;
             }
         }
     }
  }
  g_ver0 += 9;
```

```
  g_hor0 += 8;
}
```

2.1.5. Lista

La lista consta de dos estructuras muy simples:

```
struct Nodo {
        Uint64  ID;
        SDL_Surface *surfaceGen;
        SDL_Surface *surfacePers;
        Uint32 coord_x;
        Uint32 coord_y;
        Nodo* prox;
        Nodo* prev;
};
struct Lista {
        Nodo* primero;
};
```

La idea es que las variables del tipo `Lista` sólo contengan un puntero al primer elemento de la lista. Los nodos se encuentran doblemente enlazados (cada nodo tiene una referencia al nodo previo y al nodo próximo). El puntero al nodo previo del primer elemento debe apuntar a `NULL`, lo mismo que el puntero al próximo del último.

Se pedían las siguientes funciones:

```
Lista* constructor_lista();
void inicializar_nodo(Nodo* nuevo, SDL_Surface *surfacePers,
    SDL_Surface *surfaceGen, Uint32 x, Uint32 y, Uint32 ID);
bool verificar_id (Lista* la_lista, Uint32 id);
void agregar_item_ordenado(Lista* la_lista, SDL_Surface*
    surfacePers, SDL_Surface* surfaceGen, Uint32 x, Uint32 y,
    Uint32 ID);
void borrar(Lista* la_lista, Uint32 x, Uint32 y);
void liberar_lista(Lista* l);
```

Las construimos a todas, con la excepción de `inicializar_nodo` que era opcional. Pudimos haberla usado en la función `agregar_item_ordenado`, pero decidimos no hacerlo; nos pareció que agregaría un *overhead* innecesario.

Todas las funciones que atañen a la lista se encuentran en el mismo archivo, `src/asm/funciones_lista.a`

Pseudocódigo:

```
Lista* constructor_lista() {
    Lista *res = (Lista*) malloc(sizeof(Lista));
    res->primero = NULL;
    return res;
}

bool verificar_id (Lista* la_lista, Uint32 id) {
```

```
    Nodo* sgte = la_lista->primero;
    while (sgte) {
        Nodo* prox = sgte->prox;
        if (sgte->ID == id) return false;
        sgte = prox;
    }
    return true;
}

void conectar (Nodo* a, Nodo* b) {
    a->prox = b;
    b->prev = a;
}

void agregar_item_ordenado(Lista* la_lista, SDL_Surface* surfacePers, SDL_Surface*
     surfaceGen, Uint32 x, Uint32 y, Uint32 ID) {

    Nodo *sgte = la_lista->primero, *anterior = NULL;

    while (sgte && sgte->ID < ID) {
        anterior = sgte;
        sgte = sgte->prox;
    }

    if (sgte == NULL || (sgte->ID =! ID)) {
        Nodo* nuevo = (Nodo*) malloc(sizeof(Nodo));
        nuevo->ID = ID;
        nuevo->surfaceGen = surfaceGen;
        nuevo->surfacePers = surfacePers;
        nuevo->coord_x = x;
        nuevo->coord_y = y;
        nuevo->prox = NULL;
        nuevo->prev = NULL;

        if (anterior) conectar(anterior,nuevo);
        else la_lista->primero = nuevo;

        if (sgte) conectar(nuevo,sgte);
    }
}


void borrar(Lista* la_lista, Uint32 x, Uint32 y) {

    Nodo *sgte = la_lista->primero, *proximo = NULL;

    while (sgte) {
        proximo = sgte->prox;

        if ( abs(sgte->coord_x - x) < 50 && abs(sgte->coord_y - y) < 50 ) {

            if (sgte->prev == NULL) {
                la_lista->primero = sgte->prox;
                if (sgte->prox != NULL) sgte->prox->prev = NULL;
            } else if (sgte->prox == NULL) sgte->prev->prox = NULL;
            else conectar(sgte->prev,sgte->prox);

            free(sgte);
        }
        sgte = proximo;
    }
}
```

```
void liberar_lista(Lista* l) {

    Nodo* sgte = l->primero;

    while (sgte) {
        Nodo* prox = sgte->prox;
        free(sgte);
        sgte = prox;
    }

    free(l);
}
```

2.1.6. Iterador

La estructura del iterador es la siguiente:

```
struct Iterador {
        Nodo *actual;
};
```

La idea es que el puntero del iterador se encuentre siempre apuntando a un nodo (el nodo actual) de la lista que recorre. A continuación, las funciones correspondientes al iterador pedidas:

```
Iterador *iterador_lista;
Iterador* constructor_iterador(Lista *lista);
void proximo(Iterador *iter);
Nodo* item(Iterador *iter);
bool hay_proximo(Iterador *iter);
void liberar_iterador(Iterador *iter);
```

Todas estas funciones se hallan en el archivo `src/asm/funciones_iterador.asm`.

Pseudocódigo:

```
extern "C" Iterador* constructor_iterador(Lista *lista) {
    Iterador* res = (Iterador*) malloc(sizeof(Iterador));
    res->actual = lista->primero;
    return res;
}

extern "C" void proximo(Iterador *iter) {
    iter->actual = iter->actual->prox;
}

extern "C" Nodo* item(Iterador *iter) {
    return iter->actual;
}

extern "C" bool hay_proximo(Iterador *iter){
    return iter->actual != NULL;
}

extern "C" void liberar_iterador(Iterador *iter) {
    free(iter);
}
```

2.2. Basura en los bitmaps

El formato de los bitmaps que utilizamos alineaba cada fila de pixeles a un múltiplo de 32 bits. Esto quiere decir que, cuando la fila de pixeles no ocupaba un múltiplo de 32 bits, el bitmap contenía, luego de los datos correspondientes a los colores de los pixeles, datos "basura" para llevar la fila a un múltiplo de 32 bits.

En cada una de las funciones que se ocupaban de leer o bien de escribir en los bitmaps cargados en memoria, tuvimos que tener en cuenta esto. Como el direccionamiento en la arquitectura que utilizamos es a byte, tuvimos que prestar atención al hecho de que cada fila de pixeles tendría que empezar en una dirección múltiplo de 4.

2.3. Manejo de la pila (`entrada_funcion` y `salida_funcion`)

Al arribar a la función, en la pila (*stack*) se encuentran los parámetros pasados y, a continuación y en el tope, la dirección de retorno a la cual deberá volverse luego de la ejecución de la función. Ese es el momento en el cual debe armarse el `stack frame` de la función. Para hacerlo, precisamos lo siguiente:

1. mover el puntero a la base de la pila (`ebp`) al tope de esta
2. de ser necesario, puede reservarse espacio en la pila para almacenar variables/datos locales (es decir, que sólo interesan a la función); para esto, sólo basta decrementar el puntero al tope de la pila tantas veces como palabras de 32 bits se quieran reservar

Además, como la convención C asegura que los registros `edi`, `esi` y `ebx` deben mantenerse en sus valores iniciales al momento del retorno, es una práctica común (que también adoptamos) enviarlos al tope de la pila.

Como debíamos realizar esta tarea en la gran mayoría de las funciones, nos fue útil construirnos una macro `entrada_funcion` y su contrapartida `salida_funcion`, encargadas de realizar las tareas mencionadas (la primera) y de dejar el stack en el estado inicial (la segunda).

A continuación el código correspondiente a las macros mencionadas:

Dado que, como mencionamos, la mayoría de las funciones recibían parámetros, nos fue útil la creación de una macro llamada `entrada_funcion` y su contrapartida `salida_funcion`. A continuación, las macros mencionadas:

```
%macro entrada_funcion 1
        push ebp
        mov ebp, esp
        %if %1 <> 0
        sub esp, %1
        %endif
        push edi
        push esi
        push ebx
```

```
%endmacro
%macro salida_funcion 1
        pop ebx
        pop esi
        pop edi
        %if %1 <> 0
        add esp, %1
        %endif
        pop ebp
    ret
%endmacro
```

2.4. Recepción de parámetros

La mayoría de las funciones que tuvimos que construir recibían parámetros. Estos parámetros, en su mayoría, eran pasados a través de la pila. Como las funciones eran utilizadas desde C/C++, debió seguirse la convención de C para el pasaje (y la recepción) de parámetros. Dicha convención dice que los parámetros que recibirá la función se introducen en la pila uno a uno, desde el último al primero, es decir, de derecha a izquierda.

Para mejorar la legibilidad de nuestro código, en todas las funciones que recibían parámetros, definimos algunas constantes con nombres declarativos para cada uno de los parámetros, los cuales referenciamos usando la base de la pila (`ebp`).

Por ejemplo, para el caso de la función `generarFondo`, cuya firma era

`void generarFondo (Uint8 *fondo, Uint32 fondo_w, Uint32 fondo_h, Uint32 screenAbsPos);`

definimos las siguientes constantes:

```
%define fondo [ebp + 8]
%define fondo_w [ebp + 12]
%define fondo_h [ebp + 16]
%define coord [ebp + 20]
```

2.5. Reserva y liberación de memoria (`malloc` y `free`)

En algunas funciones, como es el caso de los constructores (y destructores) del iterador y de la lista, teníamos que reservar (y liberar) memoria en el *heap*. Para esto, utilizamos las funciones de C `malloc` y `free`.

Siguiendo la convención de C, pasamos los parámetros a estas funciones a través de la pila. En el caso de `malloc` tuvimos que pasar el número de bytes a reservar, y en el caso de `free` la dirección de memoria del espacio a liberar.

Ambas funciones, también siguiendo la convención de C, retornaban valores a través del registro `eax`. `malloc` retornaba la dirección de memoria en la cual se ubicaba el espacio

reservado (o cero, en el caso de no haber podido reservar memoria) y `free` retornaba si la liberación había sido exitosa.

2.6. Acceso a variables globales

Algunas de nuestras funciones requerían acceder a variables globales (como, por ejemplo, la matriz de pixeles correspondiente a la pantalla), declaradas en el programa en C++. Para acceder a dichas variables, no hizo falta más que utilizar `extern variable` (siendo `variable` una variable global) en alguna parte de nuestros archivos de assembly.

El símbolo `variable` debió utilizarse luego como una dirección de memoria: la dirección que apunta a la variable. En otras palabras, utilizando `extern` contábamos con un puntero a la variable en memoria.

3. Resultados

3.1. *Debugging*

Los programas enfocados al *debugging* son herramientas muy importantes, especialmente para el programador de lenguajes de bajo nivel. En nuestro caso, nos sirvieron para encontrar errores en nuestro código y entender por qué al principio muchas de nuestras funciones no arrojaban los resultados esperados.

Los programas que más nos ayudaron en esta tarea fueron:

- gdb (y sus frontends: ddd, Nemiver)
- valgrind

3.2. El problema de la basura en los bitmaps

En un principio, no sabíamos con certidumbre cómo era el formato de bitmaps con el que estábamos trabajando. No todos los formatos de bitmaps son alineados a 32 bits, como este.

Al realizar las funciones sin pensar siquiera en la basura, encontrábamos resultados extraños: en `generarFondo`, por ejemplo, obteníamos imágenes torcidas/inclinadas.

Este fue un problema en el cual el *debugger* poco podía hacer: la cantidad de datos, en este caso los pixeles de las imágenes, era muy extensa y variada, y además analizar un mapa de la memoria no es tan fácil como observar el cambio de los valores de los registros.

4. Conclusiones

A partir de la realización de este trabajo práctico sacamos varias conclusiones:

- Escribir funciones que parecen muy simples y fáciles de construir en un lenguaje de alto nivel, puede tornarse una *tarea muy compleja* en un lenguaje de bajo nivel como el que manejamos.

- Por fortuna, existen herramientas para facilitarnos las cosas, como los *debuggers*.
- Escribir en *assembly*, si bien es una tarea ardua, resultó ser provechoso, pues nos posibilitó entender qué es lo que se esconde detrás de los lenguajes de alto nivel, de sus abstracciones para el manejo fácil de memoria, de sus sistemas de tipos.

Referencias

- Intel R . 64 and IA-32 Architectures Software 1: Basic Architecture
- Intel R. 64 and IA-32 Architectures Software 2A: Instruction Set Reference, A-M
- Intel R. 64 and IA-32 Architectures Software 2B: Instruction Set Reference, N-Z
- Información sobre bitmaps: http://en.wikipedia.org/wiki/BMP_file_format
- Documentación del NASM: http://www.nasm.us/doc/