# Organización del Computador II

28 de abril de 2009

*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** **3**
1.1. Los *sprites* . . . . . 3
1.2. Los fondos . . . . . 4

# 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 faltantes 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 `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 especifidada. 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.

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. 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. El usuario percibe un movimiento por parte

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

de los personajes pues el ojo humano no alcanza a percibir que en realidad se trata de varias figuras una detrás de la otra.

| |
|---|
| Llegué hasta acá retocando el informe. (Pablo A.) |

...

## 1.2. Los fondos

En nuestro juego, tenemos dos fondos; uno para cada escenario. Los fondos también pueden denominarse Sprites aunque estos no tengan movimiento, ya que el movimiento lo persibe el usuario. Por otra parte, para que los personajes no se vean como cuadras desencajados de otro color diferente al fondo, de declara un color-off que actuará como transparente a la hora de programar (eso sucederá porque se irá intercambiando el color-off con el fondo) y de este modo el personaje simulará estar inmerso en el escenario.

* Los BMP

Notamos que este tipo de archivos a veces su longitud no es múltiplo de cuatro, entonces encontramos bytes con "basura", este comportamiento lo encapsulamos en una macro para que el código assembrer sea más legible.

*2.1. Generar Fondo*

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

Recibe como parametros un puntero al lugar donde esta la pantalla y sus dimensiones en pixeles (w de ancho y h de alto). Estos pixeles RGB también hay que tener en cuenta que los bytes tienen "basura" (no siendo mod 4) por lo que hacemos el tratamiento para eliminarla.

*2.2. Recortar*

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

Una imagen BMP puede guardar varios Sprites, en este caso se tiene que optar por la imagen necesaria, debiendose recortar el Sprite indicado por la instancia. Recibe como parametros un puntero a la imagen, y sus dimensiones en pixeles. El número de instancia comenzando desde cero, y el ancho de la misma. También un booleano que indicará la orientación del personaje dentro del escenario (esto es para que tenga un buen efecto a la hora de hacerlo cambiar de dirección, que camina hacia un lado u otro de la pantalla) y estos bytes serán devueltos en el puntero res.

*Mirando un poco nuestro Código:* (vamos a destacar algunas partes, ya que el código está comentado, y adjuntado junto a este informe.)

En principio ponemos en la pila los parámetros de la función. Por ejemplo el primer parámetro lo tenemos en:

_ %define ptrSprite ‘[‘ebp+8‘]‘_

Y así con los demás sabiendo que son de 32 bits. El %define es para legibilidad en el código, así en vez de llamar a [ebp+8] usamos un nombre más amigable como ptrSprite. También definimos las variables locales, por convención en la pila las colocamos alrevez. Ejemplo:

%define ancho_sprite_bytes [ebp-4]

Luego comienza nuestra función: (la función entera se puede ver en el archivo recortar.asm, no parece interesante copiar el código entero, pero allí está la funcionalidad comentada. ) global recortar recortar:

entrada_funcion 12 <—- cantidad de parámetros

. . .

calcular_pixels ebx,ancho_instancia ;ebx: ancho de la instancia sobre el sprite en ;pixeles (sin la basura)

calcular_basura eax, ebx ;basura para la instancia

. . . sub ebx, 03h ;cantidad de bytes para avanzar del primer al ultimo pixel de una

fila Esto es para explicar la razon de hacer sub con 03h, recordar (rgb) En el ciclo, donde voy a ir copiado los bytes de cada fila de la instancia, muevo hacia un *registro de 8 bits*, en este caso *bl (parte baja)* y luego avanzo de a 1 para moverme hacia el otro pixel. Y termino el ciclo cuando el contador de hace cero, usando la instrucción *loopne*. Por último, la iteración se hace pasando por todas las filas (pensando el sprite como una matriz, aunque sabemos que realmente es una tira de bytes.) salida_funcion 12 <—- tiene la restauración de la pila, y también la convención C.

*2.3. Blit*

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

Esta función es la encargada de solapar un Sprite sobre el fondo de manera de que solo se vea el dibujo y no el color-off. Lo primero que hacemos al trabajar con este tipo de imágenes es calcularle la "basura", y multiplicamos por 3 recordando que estamos con RGB. Como parámetros tenemos el puntero a la imágen, el color-off de la misma, las dimensiones y las coordenadas donde aparece. Por otro lado estamos necesitando acceder a la pantalla, y esto lo hacemos accediendo.

extern screen_pixeles <— esta variable se encuentra en el main, al ser

una variable externa a este archivo la llamo de este modo.

*Blit:*

. . .

mov edi, ptrSprite

edi apunta todo el tiempo a la posicion dentro de sprite, en esta función se necesitan dos punteros, uno que apunte al sprite y otro que apunte al fondo (al escenario)

mov edx, SCREEN_W*3 cargamos el ancho de la pantalla en edx y lo multiplicamos por 3 como debemos hacer en todos los casos.

calcular_basura ebx,edx (calculo de la basura en ebx)

add edx, ebx (sumamos el valor de la basura al registro edx)

mov ancho_screen_bytes,edx

. . . mov esi, '['screen_pixeles ']' Acá es donde cargo el puntero a pantalla. Y luego cargamos las coordenadas...

. . . Una nueva_fila (la misma idea que en recortar, pensando el sprite como una matriz) mov ecx, anchoSprite

Y el ciclo no es más que intercambiar los bytes que tienen el color-off de nuestra imagen por los del fondo, también utilizando *registros de 8 bits* para RGB.

. . .

finBlit:

salida_funcion 16 <— idem recortar, restaura la pila como estaba al principio y los registros que se pushean por convención del lenguaje C.

2.4. Generar Plasma

.

.

.

2.5. Lista

La lista se utilizará para el personaje de "Gambito" que también aparecerá dentro de los escenarios en alguna posición en particular, y este personaje no va a tener el efecto de caminar, solo se aparecerá parado con un leve movimiento. Para ello se utilizará la lista, esta se consulta constantemente en cada Frame del juego para saber que se debe dibujar en pantalla.

Inicializar_nodo es opcional. En nuestro caso, podriamos haberla usado en agregar_item_ordenado pero no lo hicimos.

Las funciones para la lista las hicimos en el mismo archivo funciones_lista.asm

global inicializar_nodo

global verificar_id

global agregar_item_ordenado

global borrar

global liberar_lista

*Lista* constructor_lista()*

Para el constructor de lista pedimos memoria mediante la instrucción de C call malloc

bool verificar_id (Lista* la_lista, Uint32 id)

mov eax, verif_lista (aca tenemos el nodo* primero, porque vamos a tener que recorrer)

(y verif_lista está definido arriba del código con un %define)

. . .

mov ebx, '['eax']' ;cargo la parte menos significativa del Id del nodo

mov ecx, '['eax+4']' *;porq ID es de 64 bits*

Luego me muevo al prox, y comparo para ver si ya encontré o no.

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

También requiere *llamar a malloc*. Inicializo la estructura del nodo. En eax esta todo el tiempo el puntero al nodo nuevo y en ebx esta el puntero al nodo actual. Tomamos los tres casos: (el ítem que agrego debe ser el primero, o la lista está vacía o bien la coord_x es menor al prox.) Si no hay ningun nodo, agregar el nuevo (eax) al principio.

(cmp ag_x, ecx) si la coord x del primer nodo es menor a la que me pasaron por parametro.

Si el ítem debe ir en el medio.

connect_nodos edx,eax ; pongo el elemento nuevo (eax) despues del nodo actual (edx)

connect_nodos eax,ebx ; pongo el elemento nuevo (eax) antes del proximo (ebx)

Y si el ítem debe ser el último (en este caso el puntero a prox del nuevo ítem debe apuntar a null)

Donde ebx tiene un puntero al ultimo connect_nodos ebx,eax y pongo el elemento nuevo (eax) despues del nodo actual (ebx)

_void borrar(Lista* la_lista, Uint32 x, Uint32 y)_

Esta otra función es muy sililar a las anteriores, salvo porque hay que recordar liberar la memoria que se utilizó para ese elem. Del mismo modo que agregar_item, también actualizo los punteros de los items prev y prox, si es que existen (esos casos borde también son examinados aparte)

_void liberar_lista(Lista* l)_

Esta función no es muy larga, me gustaría poner la mayor parte del código ya que es bastante claro: mov edx, l_lista ; cargo en edx el puntero a la lista mov ebx, '['edx']' ; cargo en ebx el puntero al primer nodo de la lista l_seguir: ; asumo q en ebx esta siempre el puntero al nodo actual y en edx el

puntero a la lista cmp ebx, 0 jne l_eliminar_elemento reviso si la lista esta vacia

mov edx, l_lista ; cargo en edx el puntero a la lista (por si lo perdi) push edx call free <— importante liberar en este caso add esp, 4

salida_funcion 0 <— recuerda que acá tenemos la convención C y pongo la pila como al principio.

Si no esta vacia

; elimino ebx l_eliminar_elemento: mov esi, ‘[‘ebx + prox‘]‘ push ebx call free <— importante liberar en este caso add esp, 4 mov ebx, esi

jmp l_seguir

Nos pareció que era más claro mostrar el código acá, ya que no es largo y fue bastante simple...

*2.6 Iterador*

Igual que con la lista las funciones del iterador las pusimos juntas en un mismo archivo “funciones_iterador.asm”

global constructor_iterador

global hay_proximo

global proximo

global item

global liberar_iterador

Iterador* constructor_iterador(Lista *lista) ; los punteros son de 32 bits

mov eax, 4 <— 4bytes xq es un registro de 32 bits

push eax

call malloc ; parecido a la lista pido memoria al construír (pensánsolo en C realmente es la misma idea)

add esp, 4

cmp eax, 0

; si malloc no me pudo dar memoria (raro que ocurra pero por las dudas si eso ocurriera salgo de la función)

je retornar ;salto a salir

mov ebx, const_it_lista ; ebx = direccion que apunta a la Lista

mov ebx, ´[´ebx´]´ ; ebx = direccion que apunta al Nodo

mov ´[´eax´]´, ebx ; En el espacio creado en memoria guardo ; la direccion que apunta al nodo.

bool hay_proximo(Iterador *iter)

En realidad un mejor nombre creo que sería hay_actual(). No parece que valga la pena copiar el código de esta función aunq es muy simple. La idea es simplemente ver si el nodo actual tiene un siguiente elemento, que no apunte a null.

void proximo(Iterador *iter)

entrada_funcion 0

mov eax, prox_pit ; eax = direccion que apunta al Iterador

mov ebx, ´[´eax´]´ ; ebx = direccion que apunta al Nodo actual

mov ebx, ´[´ebx + prox´]´ ; ebx = direccion que apunta al Nodo proximo

cmp ebx, 0

mov ´[´eax´]´, ebx

salida_funcion 0

Nodo* item(Iterador *iter)

Devuelvo la direccion que apunta al Nodo actual, en nuestro caso en el registro eax, claramente porq devuelvo un puntero.

void liberar_iterador(Iterador *iter)

%define lib_pit ´[´ebp + 8´]´

liberar_iterador: entrada_funcion 0

mov eax, lib_pit push eax call free add esp, 4

salida_funcion 0

Notar que lo más importante es llamar a free para liberar la memoria pedida.

*Conclusiones*

Además de practicar los conceptos teoricos y la interaccion de assembrer con C. trabajamos con algunas herramientas de debugger. Vimos como se manejan los bytes en memoria bien a bajo nivel. Realmente no resultó del todo cómodo escribir en este lenguaje pero es un aprendizaje realmente útil.

*Compilar y Ejecutar*

El trabajo práctico fue desarrollado sobre Linux y no es multiplataforma. Para compilarlo se pueden usar los comandos *bash compilar_asm.sh* elimina todos los .o en ./asm/, y ensambla todos los archivos presentes en ese directorio y *bash compilar_cpp.sh* compila el main.cpp y crea el ejecutable linkeando con los .o en ./asm. El ejecutable que genera se llama "prg" y se encuentra dentro de la carpeta src.

*Referencias*

1- Intel R . 64 and IA-32 Architectures Software 1: Basic Architecture

2- Intel R. 64 and IA-32 Architectures Software 2A: Instruction Set Reference, A-M

3- Intel R. 64 and IA-32 Architectures Software 2B: Instruction Set Reference, N-Z