Organización del Computador II

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

Trabajo Práctico 1: "Oportuncrisis" (Segunda Entrega)

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 ahora por pantalla? . . . 1
1.2. El efecto *negative* . . . 2
1.3. El efecto *smooth* . . . 2

2. Desarrollo **2**
2.1. Funciones desarrolladas . . . 2
2.1.1. Función `negative` . . . 2
2.1.2. Función `smooth` . . . 2
2.2. Funciones optimizadas . . . 3
2.2.1. Función `recortar` . . . 3
2.2.2. Función `blit` . . . 3
2.2.3. Función `generarFondo` . . . 3
2.2.4. Función `generarPlasma` . . . 3
2.3. Optimización . . . 3
2.3.1. Realizando operaciones en paralelo (SIMD) . . . 3
2.3.2. Evitando saltos . . . 3
2.3.3. Acceder a memoria no es *tan* lento (memoria cache) . . . 3
2.3.4. Operaciones aritméticas con `lea` y *shifts* . . . 4

3. Resultados **4**

4. Conclusiones **4**

1. Introducción

La segunda entrega de este trabajo práctico consiste, básicamente, en:

- la *optimización* de las funciones que formaron parte de la primera entrega, y
- la inclusión de *dos nuevas funciones*: `smooth()` y `negative()`

La intención principal de este segundo trabajo es sacarle provecho a las distintas facilidades que proveen las extensiones SIMD[1] de la arquitectura x86 de Intel, así como aplicar distintas técnicas para la optimización del código que se ejecuta en dicha arquitectura.

1.1. ¿Qué se muestra ahora por pantalla?

Recordemos cuál era el ciclo que se repetía todo el tiempo en el juego:

```
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.
```

Ahora, en la etapa 4 ("mostrar resultados por pantalla"), tiene lugar la siguiente secuencia (extendida):

```
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)
6  si la tecla 1 o 3 se encuentra oprimida, comenzar la
       transición a un nuevo escenario
7  si la tecla 1 o 3 se encuentra oprimida, aplicar el efecto
       negative
8  si se esta realizando la transición a un nuevo escenario:
9      aplicar el efecto smooth
10     si hay más pixels negros que de otro color:
11         finalizar el cambio de escenario
```

[1] *Single Instruction, Multiple Data*

1.2. El efecto *negative*

1.3. El efecto *smooth*

2. Desarrollo

2.1. Funciones desarrolladas

2.1.1. Función `negative`

```
void negative();
```

Parámetros:

- Ninguno.

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

Pseudocódigo:

```
void negative():
    para cada componente RGB c de cada pixel de la pantalla:
        sumatoria = superior(c) + inferior(c) + anterior(c) + posterior(c) + 1
        c = f(sumatoria)
```

Donde la función f es:

$$f(x) = \frac{1}{\sqrt{sumatoria}} \times 255$$

2.1.2. Función `smooth`

```
bool smooth();
```

Parámetros:

- Ninguno.

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

Pseudocódigo:

```
bool smooth():
    para cada componente RGB c de cada pixel de la pantalla:
        sumatoria = superior(c) + inferior(c) + anterior(c) + posterior(c)
        c = sumatoria/4
        si sumatoria == 0:
            contador_negros += 1
        si no:
            contador_blancos +=1
    retornar evaluar(contador_negros > contador_blancos)
```

2.2. Funciones optimizadas

2.2.1. Función `recortar`

2.2.2. Función `blit`

2.2.3. Función `generarFondo`

2.2.4. Función `generarPlasma`

2.3. Optimización

2.3.1. Realizando operaciones en paralelo (SIMD)

Desde la aparición del procesador *Pentium II* de *Intel*, se agregaron a la arquitectura varias extensiones que proveen soporte para instrucciones SIMD. Entre ellas se encuentran: MMX, SSE, SSE2, SSE3 y SSE4.

Las instrucciones SIMD se utilizan para realizar operaciones en paralelo; se suele trabajar con vectores de datos en lugar de datos individuales. Las mejoras en *performance* son generalmente apreciables, y muchas veces determinan la utilidad de ciertas aplicaciones.

En general, las más beneficiadas por la utilización de instrucciones SIMD son las aplicaciones que realizan procesamientos similares de grandes cantidades de datos (por ejemplo, las aplicaciones multimedia).

En nuestro caso, las instrucciones SIMD nos permitieron procesar, en ocasiones, hasta 5 pixels en paralelo. Esto produjo la agilización de muchas de nuestras rutinas que trabajan sobre un gran número de pixels de la pantalla.

2.3.2. Evitando saltos

En la realización de este segundo trabajo, se trató especialmente de realizar la menor cantidad de saltos (incluidas las llamadas a subrutinas) posibles.

Sucede que la máquina realiza *fetches* de instrucciones que presume serán utilizadas en el corto plazo. Si bien existe un predictor de saltos, su heurística puede fallar. En esos casos, los saltos pueden llegar a interrumpir la secuencia de *prefetch* que realiza la máquina, haciendo que esta pierda *performance*.

2.3.3. Acceder a memoria no es *tan* lento (memoria cache)

Hoy en día, las *memorias cache* son cada vez más grandes y rápidas. Si bien los accesos a la memoria principal suelen tomar varios ciclos de reloj, la realidad es que (siguiendo los *principios de localidad espacial y temporal*) las probabilidades de que los datos que buscamos se encuentren en alguna de las memorias cache de nuestra computadora, son altas.

Esto hace que el uso de memoria para almacenar datos temporales, útiles durante la ejecución de nuestro programa, cuando ya no disponemos de registros, no sea algo demasiado prohibitivo.

En lo que refiere a este trabajo, en ocasiones optamos por alojar datos (cálculos, información auxiliar) en memoria, sabiendo que, dependiendo de la operación, acceder a ellos podía ser más rápido que recalcularlos.

2.3.4. Operaciones aritméticas con `lea` y *shifts*

En ocasiones, es preferible usar `lea` para realizar operaciones aritméticas (en lugar de otras instrucciones específicas), en especial cuando el objetivo es conseguir una dirección de memoria a partir de la cual obtener un dato, no sólo para ganar claridad, sino también porque puede ser más rápido al momento de la ejecución de nuestro programa.

También es importante tener en cuenta que, para multiplicaciones y divisiones por múltiplos de 2, es preferible utilizar *shifts* (aritméticos o lógicos, dependiendo el caso) en lugar de usar las instrucciones de multiplicación y división que son notablemente más costosas.

A lo largo del trabajo, tuvimos en cuentas ambas recomendaciones.

3. Resultados

4. Conclusiones

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/