Organización del Computador II

Primer Cuatrimestre de 2008

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

Trabajo práctico II

Rescatando a la Princesa Peach 2
(la venganza de Homero)

Abstract

Modificación de BMPs píxel a píxel, comparaciones en asembler y otras yerbas para la confección de un escenario de Mario Bross. Uso de MMX y FPU para realizar cálculos con números enteros y de punto flotante

Palabras clave

Mario, Princesa, Asembler, Caño, Moneda, asm, BMPs, byte, RGB, MMX, FPU, Cazafantasmas, Máscaras, Funciones trigonométricas.

Grupo 15

Integrante	LU	Correo electrónico
Cristian Alejandro Archilla	433/03	gnigel@gmail.com
Matías López y Rosenfeld	437/03	matiaslopez@gmail.com
Tomás Scally	886/03	horizontedesucesos@gmail.com

Índice

1. Introducción **3**

2. Desarrollo **3**

3. Conclusiones **6**

4. Manual del usuario **7**

4.1. Detalles del contenido del cd 7

4.2. Uso del contenido del cd 7

5. Agradecimientos **8**

1. Introducción

El objetivo de este tp, es optimizar las funciones que habíamos realizado durante el desarrollo del tp1 utilizando, esta vez, la tecnología **MMX** para poder, de esta forma, "agilizar" nuestro programa. Además, se agregó una función que nos permitiera utilizar la Unidad de Punto Flotante (**FPU**) para realizar cálculos con números no enteros.

Una breve descripción de las funciones que realizamos, a modo de introducción, sería la siguiente:

- **generarFondo**: Pinta el cielo y copia el piso.
- **recortar**: Recorta una parte de un sprite.
- **blit**: Saca el color de off para mimetizar la imagen con el fondo.
- **chequearColisiones**: Comprueba si hay colisión entre 2 objetos, en este caso, **Mario** y el caño.
- **generarRayo**: Es lo que produce que el rayo de **Mario** se vea.
- **apagar**: Va cambiando el color de las monedas.

Esperemos que distrute la lectura nuestro trabajo.

2. Desarrollo

1. generarFondo: Como primera función a reescribir en **MMX**, demoró unas cuantas pasadas hasta dejarnos conformes con el resultado, cumpliendo así con su rol de debut. Utilizar registros **MMX** como banco de memoria para no utilizar variables locales fue una de las habilidades adquiridas aquí, además de aprender a dividir el procesamiento paralelamente explotando las posiblidades que nos brinda **MMX**.

   Para este tp logramos una renovada y casi diría revolucionaria forma de calcular el resto de dividir por 4, cosa que necesitamos para calcular la basura. En lugar de dividir hacemos un AND de **0x3** con el número y eso nos da el resto de dividir por 4. Luego le restamos esto a 4 para obtener la basura a menos que el resto sea 0 donde no tenemos basura.

   El piso lo procesamos de forma lineal, en lugar de pintar cuadrado por cuadrado lo pintamos directo sobre el buffer de salida.

2. recortar: En esta función calculamos la basura de la imagen para saber cuánto tenemos que avanzar por fila y la basura que le corresponda al sprite para pasarla al resultado. No requirió mucha mas ciencia que pasar de a 8 bytes usando **MMX**.

3. blit: Para esta función nos generamos 2 máscaras con el fucsia. Para procesar 2 píxeles con **MMX** utilizamos una máscara con los 3 bytes alineados al centro de la forma **0x00FF00FFFF00FF00**. La máscara para procesar de a un píxel en los extremos es **0xFF00FF00**, este píxel está alineado sobre la parte alta del registro.

   La idea de alinear al centro los 2 píxeles al procesar en **MMX** surgió con la idea de poder realizar operaciones con 2 dword y que cada resultado le corresponda a un píxel distinto.

   Aquí también calculamos la basura de la imagen para sumarla cuando terminamos de procesar cada fila al puntero y así apuntar al inicio de la siguiente fila. En el ciclo de **MMX** centramos los 2 píxeles y los comparamos con el fucsia. Usando un **PANDN**[INT97] con el resultado de la comparación y los píxeles centrados colocamos 0 donde había fucsia y el color que había antes en el otro caso. Acomodamos los pixeles y los volvemos a guardar, volviendo a colocar los valores que no nos correspondía procesar.

4. chequearColisiones: En esta función no utilizamos ninguna máscara pre definida. En cada iteración procesamos por sprite. Teniendo en cuenta que los rectángulos con los que íbamos a trabajar están alineados al eje de coordenada, pudimos simplificarnos un poco los cálculos, tomar los valores que son iguales como uno solo y dividirnos al calcular los rectángulos con base alineada al eje X y los de base alineada al eje Y. Para simplificarnos los cálculos y realizar menos, renombramos los parámetros del sprite:

$$a = xSprite$$

$$b = ySprite$$

$$c = xSprite + wSprite$$

$$d = ySprite + hSprite$$

De acuerdo al orden de los parámetros nombramos a los triángulos como Tab al triángulo formado por la arista $(a, b)(c, b)$ del cuadrado. Analogamente, tenemos Tbc, Tcd, Tda. Vamos a chequear si hay colisión calculando las áreas como indica en el enunciado usando la siguiente fórmula:

$$Area = \frac{a_1(b_2 - c_2) + b_1(c_2 - a_2) + c_1(a_2 - b_2)}{2}$$

Como lo que nos interesa es saber el signo que nos da esta fórmula, entonces, podemos dejar de lado la parte de $\frac{1}{2}$ ya que esto no influye en el signo de la expresión entera.

Nuestras 4 fórmulas a calcular quedaron:

$$Tab = a(y_1 - b) + x_1(b - b) + c(b - y_1) = a(y_1 - b) + c(b - y_1)$$

$$Tbc = c(y_1 - d) + x_1(d - b) + c(b - y_1)$$

$$Tcd = c(y_1 - d) + x_1(d - d) + a(d - y_1) = c(y_1 - d) + a(d - y_1)$$

$$Tad = a(y_1 - b) + x_1(b - d) + a(d - y_1)$$

Primero calculamos Tab y Tcd y luego Tbc y Tad, ya que estos últimos necesitan de más términos. Utilizamos operaciones en words, el **PMADDWD**[INT97] y operaciones en dwords.

Una vez que terminamos con el algoritmo y lo quisimos probar tuvimos problemas como que no chequeaba correctamente. Después de revisar intensamente el algoritmo varias veces decidimos verificar la fórmula del enunciado. Aquí nos encontramos que ésta servía para el eje cartesiano común, y no para el de la pantalla, donde el 0 de las Y está arriba. Probamos varias veces y llegamos a la conclusión de que debíamos marcar como colisión cuando todas las áreas nos daban 0 o negativas.

Un detalle implementativo que nos surgió durante el desarrollo de este algoritmo, fue que el ciclo resultó ser demasiado largo, y el jump para volver no se declaraba como jump corto, por lo que tuvimos que reemplazarlo por otros saltos intermedios. Curioso.

5. generarRayo: Al realizar esta función nos encontramos con el desafío de utilizar la FPU en un programa hecho por nosotros mismos por vez primera. Esto parecía prometedor aunque complicado. Tuvimos que ser muy prolijos al momento de usarla, cosa que nos costó más de un dolor de cabeza.

Para empezar, tuvimos que calcular el ángulo. En un apunte que hizo Emi decía que para calcularlo debíamos hacer $angulo = seno^{-1}(\frac{opuesto}{tangente})$. Para nosotros donde dice "tangente" debería decir "hipotenusa". Por otro lado, resulta que $seno^{-1}$ no es $\frac{1}{sin}$ sino el $arcsin$.

Investigando cómo hacer esto nos encontramos con: "As you can see, we are forced to use the ArcSin and ArcCos functions. Unfortunately, the FPU doesn't have these functions. However, it has the FATAN function, which computes the ArcTangent. In order to obtain the arcsin and

arccos we can use the following formulas: ArcSin = ArcTan(x/sqrt(1-sqr(x)))"[VLJ]. A pesar de haber encontrado una forma de resolver el problema, esta misma no nos convencía. Por lo cual seguimos buscando e investigando. Buscando en la web encontramos una página de la [WIKI] que hablaba de **Inverse trigonometric functions** en la cual había una sección titulada **Recommended method of calculation** [WIKITRI]. Allí se puede observar la siguiente expresión:

$$\arcsin x = \arctan \frac{x}{\sqrt{1 - x^2}}$$

que dada la existencia de la función de la FPU que calcula el *arctan*, más conocida como **FPATAN**, decidimos usar esta cuenta, que además verificamos con otras fuentes su correctitud.

Hasta este punto teníamos un rayo de longitud **tangente**, pero que si pensábamos al punto (x_i, y_i) como el origen de coordenadas, entonces sólo dibujaba en el 1^{er} cuadrante, sin importar dónde estuviera apuntando. Para solucionar esto hicimos el siguiente razonamiento: supongamos que α es el ángulo en cuestión si está en el 1^{er} cuadrante, entonces debíamos usar directamente ese. Si está en el 2^{do} era $angulo = \pi - \alpha$, que era el ángulo llano menos el ángulo que representa a la trayectoria del rayo. Si está en el 3^{er} cuadrante $angulo = \pi + \alpha$ y en el 4^{to} simplemente: $angulo = 2 \times \pi - \alpha$.

Además de esto, vimos que existían muchas expresiones que íbamos a necesitar calcular muchas veces como ser: $\frac{4 \times \pi}{128}$, 40, 8. Entonces estos datos los calculamos inicialmente, o los cargamos de memoria al comenzar, y los guardamos en la FPU sin modificarlos para poder usarlos tantas veces como fuera necesario.

Hicimos una modificación siguiendo los consejos de **Emi** en la 2^{da} vez que calculamos y, que fue cambiar un $\times 8$ por un $\times 4$, para lograr la dispersión deseada del rayo.

Intentamos arreglar también un problema que tenía la función cuando entraba a la misma con $x_i = x_f$, pero desgraciadamente al cierre de esta edición no lo habíamos logrado.

Lo demás no tiene mucho más para ser resaltado. Salvo que tuvimos algunos problemas al momento de pintar que se solucionaron haciéndolos de otra forma y no sabemos por qué no andaban con el código original.

6. APAGAR: Al igual que anteriormente dejamos los últimos 3 píxeles para que se procesen de a uno. En esta función utilizamos 2 máscaras, una para el fucsia de **MMX** y la otra para el procesado simple.

   Aquí también calculamos la basura de la imagen para sumársela, cuando terminamos de procesar cada fila, al puntero. Así apuntamos al inicio de la siguiente fila.

   Lo primero que generamos es una máscara con el valor del contador para completar 0 en R y con el contador en G y B. Como todas las otras máscaras que utilizamos para procesar en paralelo 2 píxeles, ubicamos los 3 bytes en el centro.

   En el ciclo de **MMX** centramos los 2 píxeles y los comparamos con el fucsia. Usamos esta comparación para llenar con el valor anterior donde da igual (**PAND**) y llenar con el valor de la máscara con el puntero donde no da igual (**PANDN**). Luego, acomodamos los píxeles y los volvemos a guardar y volvemos a colocar los valores que no nos correspondía procesar. Finalmente restamos al contador 8 y si nos da menor o igual a 0 guardamos 0.

   La forma de comparar y guardar los resultados fue evolucionando a medida que iterábamos en la confección de esta función, ya que en un principio comparábamos 2 veces, luego hacíamos una comparación y la negábamos comparándola con 0 y luego se nos encendió la lamparita y con una sola comparación, el **PAND** y el **PANDN**, nos alcanzó.

   Lo más difícil fue generar la máscara con el contador de manera correcta. Además nos resultó complicado darnos cuenta del orden en que vienen los píxeles a causa del little endian

para luego acomodarlos en el centro de la máscara. Sin contar con que el **MOVD** de un registro a **MMX** nos jugó la mala pasada de pisarnos la parte alta.

3. Conclusiones

Los cálculos de punto flotante, aunque son medio incómodos por motivos de espacio, se pueden realizar perfectamente con la FPU si uno es lo suficientemente cuidadoso. Además de que nos parece que no se puede hacer código que use la FPU sin tener todo el tiempo dibujado al lado el estado de la misma.

El tp2 anda mucho más rápido que el tp1. Esto se nota al correr ambos tps en la misma máquina.

4. Manual del usuario

4.1. Detalles del contenido del cd

En el **CD** usted podrá ver 6 archivos .asm. Cada uno se llama como la función cuyo codigo guarda en su interior, un archivo Makefile para poder compilar el tp, un archivo .c y un directorio imágenes con su contenido, estos últimos dos provistos por la cátedra.

El listado de archivos sería:

```
apagar.asm
chequearColisiones.asm
generarRayo.asm
blit.asm
imagenes
generarFondo.asm
recortar.asm
Makefile
main.c
tp2.pdf

./imagenes:
bs.bmp
coins.bmp
hspt.bmp
marioJumpL.bmp
marioStandL.bmp
pipe.bmp
questB.bmp
cloud.bmp
estupe.bmp
hsptr.bmp
marioSpriteW.bmp
marioStandR.bmp
piso.bmp
quest.bmp
coin.bmp
go.bmp
marioJump.bmp
marioSpriteWL.bmp
mon.bmp
PrincessToadstool.bmp
victoria.bmp
```

4.2. Uso del contenido del cd

El cd viene con sus correpondiente Makefile para correr el tp como ya explicamos en la entrega del tp1.

Para ganar el juego usted debe agarrar un par de monedas con el rayo, agarrar una vez la caja y después parado lo más a la derecha que se pueda dispararle al caño.

Aunque cabe aclarar que resulta más divertido o pintoresco o raro o sorprendente o innovador o inesperado o jeje o asombroso o increible o intrigante o intrínseco o divagante o extravagante o extraño o amazing o ... o groso perder el juego cosa que se logra disparandole al caño o a muchas

monedas.

5. Agradecimientos

No queremos terminar este informe sin antes agradecer a las siguientes personas que gracias a su compañía, consejo y buena onda llegamos a terminar el tp que está usted leyendo en este momento:

- Emi
- Fran
- Facu
- y a los mismos de siempre

¡Gracias totales!

Referencias

[INT97] Intel 64 and IA-32 Architectures Software Developer's Manual

[WIKI] http://en.wikipedia.org/

[WIKITRI] http://en.wikipedia.org/wiki/Inverse_trigonometric_function#Recommended_method_of_calculation

[VLJ] http://vx.netlux.org/lib/vlj03.html