



DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Filtros de Imagen

Organización del Computador II
Primer Cuatrimestre - 2015

Integrante	LU	Correo electrónico
Christian Cuneo	755/13	chriscuneo93@gmail.com
Ignacio Lebrero	751/13	ignaciolebrero@gmail.com
Jorge Porto	376/11	cuanto.p.p@gmail.com



Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

<http://www.fcen.uba.ar>

Resumen

Los filtros de imagen son una herramienta poderosa a la hora de retocar una imagen, usados ampliamente en fotografía, publicidad, videojuegos, etc. Su uso brinda una gamma de opciones para modificar las imagenes de manera que sea mas flexible su edicion o analisis.

En este trabajo practico presentamos los metodos blur, merge y hsl ya existentes y los implementamos en lenguaje de ensamblador. Damos dos implementaciones de cada filtro siendo la segunda una optimizacion de la primera en merge y blur, y una variacion de implementacion C/Assembler a Assembler en hsl.

Nuestros experimentos demuestran.....

Índice

1. Intoducción	5
2. Desarrollo	5
2.1. Merge	5
2.1.1. Implementación en Assembler 1	5
2.1.2. Implementación 2	6
2.1.3. Implementación en C	6
3. Conclusiones y trabajo futuro	6

Índice

Resumen

En el presente trabajo se describe la problemática de ...

1. Introducción

El lenguaje C es uno de los más eficientes en cuestión de performance, pero esto no quiere decir que.. En este trabajo practico se realizan implementaciones en assembler..

2. Desarrollo

2.1. Merge

El merge nos permite a partir de dos imágenes y un valor entre cero y uno, obtener una combinación de estas ultimas según la proporción indicada por el valor.

2.1.1. Implementación en Assembler 1

Se recibe por parámetro dos punteros a dos imágenes almacenadas en memoria como una matriz de pixeles. Como los pixeles de estas imágenes ocupan 4 byte, y la cantidad de pixeles de las mismas es multiplo de 4, utilizando los registros xmm podemos traer de a 4 pixeles. Luego incrementamos los punteros a la imagen en 16 bytes y volvemos a traer los pixeles de memoria. De esta manera tenemos un ciclo principal cuya cantidad de iteraciones es la cantidad de pixeles dividido cuatro. Cada una de estas consiste en levantar de memoria cuatro pixeles y almacenarlos en un registro xmm, cuyo contenido puede verse en la figura 1

B	G	R	A	B	G	R	A	B	G	R	A	B	G	R	A
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Figura 1: Contenido del registro xmm al levantar de memoria cuatro pixeles

Luego utilizando la instrucción pshufb y una mascara apropiada ordenamos su contenido para que quede como se muestra en la figura 2

B	B	B	B	G	G	G	G	R	R	R	R	A	A	A	A
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Figura 2: Contenido del registro xmm luego de utilizar la instruccion pshufb

Utilizamos las instrucciones de desempaqueado de SIMD y un registro xmm lleno de ceros, para desempaquear la parte alta y baja, obteniendo registros xmm con el contenido como se muestra en la figura 3.

0	B	0	B	0	B	0	B	0	G	0	G	0	G	0	G
0	R	0	R	0	R	0	R	0	A	0	A	0	A	0	A

Figura 3: Contenido resultante de desempaquear parte alta y baja

Luego volvemos a desempaquear parte alta y baja de los dos registros obtenidos, y obtenemos cuatro registros xmm como se indica en la figura 4.

0	0	0	B	0	0	0	B	0	0	0	B	0	0	0	B
0	0	0	G	0	0	0	G	0	0	0	G	0	0	0	G
0	0	0	R	0	0	0	R	0	0	0	R	0	0	0	R
0	0	0	A	0	0	0	A	0	0	0	A	0	0	0	A

Figura 4: Contenido resultante de desempaquear parte alta y baja nuevamente

Utilizando la instrucción `cvtdq2ps` convertimos los cuatro valores de los registros 4, excepto el que contiene los bytes de transparencia(A), a tipo flotante. El objetivo es multiplicar cada color por `value`. Para hacer esto, previo al ciclo, utilizando la instrucción `shufps`, conseguimos en un registro xmm cuatro valores de tipo flotante con el valor que indica el índice de combinación de imágenes pasado por parametro, y en otro cuatro valores con `1-value` tal como se indica en la figura 5.

value	value	value	value
1 - value	1 - value	1 - value	1 - value

Figura 5: Contenido de los registros utilizados para multiplicar por `value` los colores

Utilizando la instrucción de `simd mulps`, multiplicamos los colores transformados a tipo flotante por `value`.

Se repite el procedimiento para la segunda imagen, excepto que no se desempaqueta el byte de transparencia(A), ya que solo interesa el byte de transparencia de la primera imagen. En este caso multiplicamos por `1-value`. Sumamos con la instrucción `addps` los valores obtenidos en la multiplicación, para los colores azul, verde y rojo. Luego convertimos a enteros de 32 bit, y con la instrucciones `packusdw`, y `packuswb` empaquetamos de forma que queden en un xmm los bytes en orden azul, verde, rojo y transparencia. Este ultimo se consideran los bytes de la primera imagen almacenados en un registro xmm como se muestra la figura 4. Finalmente con la instrucción `pshufb` ordenamos los colores para que queden en el mismo orden en que ingresaron, y escribimos en memoria el resultado.

2.1.2. Implementación 2

Es analoga a la implementación anterior, pero en este caso se hacen la suma y multiplicación en numeros enteros. Se tiene en cuenta que la multiplicación de dos enteros, da como resultado un entero que puede ocupar el doble de tamaño.

Antes de empezar el ciclo, multiplicamos en punto flotante `256` y `value`, y al resultado lo convertimos a enteros. En este proceso se pierden decimales. Como `value` es un numero entre cero y uno, al multiplicar por `256` y pasarlo a enteros, tenemos un valor entre `0` y `256`, con lo que ocupa menos de un byte de tamaño. Utilizando la instrucción `pshufb` y una mascara, almacenamos en un registro xmm 8 replicas de este valor. Luego hacemos la resta en enteros entre `256` y el valor obtenido en la multiplicación. Nuevamente tenemos un valor entero entre `0` y `256` que entra en un byte, almacenamos 8 replicas del mismo en un registro xmm tal como se ve en la figura 6. Estos registros seran utilizados para multiplicar los colores.

$256-256*value$	$256-256*value$	$256-256*value$	$256-256*value$	$256-256*value$	$256-256*value$	$256-256*value$	$256-256*value$
$256*value$	$256*value$	$256*value$	$256*value$	$256*value$	$256*value$	$256*value$	$256*value$

Figura 6: Contenido de los registros utilizados para multiplicar

El ciclo comienza igual que la implementación anterior, trayendo a un registro xmm 4 pixeles de memoria, ordenandolos con `pshufb` con una mascara

2.1.3. Implementación en C

3. Conclusiones y trabajo futuro