Metamorfosis de Imágenes Vectorizada

Guido Tagliavini Ponce
Universidad de Buenos Aires
guido.tag@gmail.com

Índice

1. Abstract **1**

2. Introducción **2**

3. Background matemático **3**
3.1. Elementos de Álgebra Lineal 3
3.2. Interpolación Lineal 5

4. Algoritmo de Beier y Neely **6**
4.1. Metamorfosis dirigida por segmentos 6
4.2. Deformación de una imagen 6
4.3. Esquema general de la metamorfosis 6
4.4. Transformación con un único segmento 7
4.5. Transformación con múltiples segmentos 9
4.6. Complejidad temporal 11

5. Vectorización **11**

6. Experimentación **12**
6.1. Resultados 12
6.2. Análisis 13

1. Abstract

Presentamos una vectorización del algoritmo de metamorfosis de imágenes de Beier y Neely [1], utilizando las extensiones SIMD de la arquitectura x86-64 de Intel. Comparamos esta implementación contra una versión del mismo algoritmo en lenguaje C, compilada con el flag -O3 del compilador de Intel (Intel®Parallel Studio XE). Las mediciones indican que nuestra versión es casi 14 veces más rápida, para tamaños prácticos de la entrada.

2. Introducción

La metamorfosis de imágenes (en inglés, *image morphing*) es una técnica para transformar una imagen (la *imagen origen*) en otra (la *imagen destino*). Esta técnica ha sido ampliamente utilizada en el ámbito de la industria cinematográfica. Un ejemplo canónico de su aplicación puede verse en el video de la canción *Black or White* de Michael Jackson, en el cual hay una secuencia de transformaciones entre las caras de personas de muy distintas características.

Diversos algoritmos para realizar esta metamorfosis han sido desarrollados, siendo el más básico aquel conocido como *cross-dissolving*, que consiste en hacer desaparecer la imagen origen a medida que se hace aparecer la imagen destino, de forma gradual, creando un efecto de disolución de ambas imágenes. Concretamente, esta técnica consiste en interpolar de a pares los colores de los píxeles de la imagen origen con los de la imagen destino, para luego variar el valor de la interpolación a lo largo del tiempo.

Figura 1: Cross-dissolving

Esta técnica no resulta satisfactoria, puesto que la transformación no es natural, en el sentido de que no sugiere que lo que ocurre es una metamorfosis. Esto se debe a que las características de la imagen origen no se transforman naturalmente en ciertas otras características de la imagen destino. Por esta razón, las técnicas más elaboradas de image morphing emplean la deformación de imágenes (en inglés, *image warping*) acompañado de un cross-dissolving, para obtener esa naturalidad buscada en la transformación. La principal diferencia entre los distintos métodos de morphing está en la forma en que se mapean las características de la imagen origen en las de la imagen destino.

En este trabajo nos concentramos en la técnica desarrollada por Beier y Neely, presentada en [1], que realiza el mapeo de características a través de segmentos dirigidos (líneas rectas con principio, fin y sentido) en la imagen origen asociados a segmentos dirigidos en la imagen destino. Esto no sólo determina cómo deben ser transformados los puntos sobre esos segmentos, sino que también dan información sobre la transformación de los puntos que los rodean. Implementamos una versión de este método basada en instrucciones SIMD, y la comparamos contra una versión tradicional, que realiza el procesamiento en serie, ganando, la primera, en términos de tiempo de ejecución,

por un amplio margen. No tenemos registro de ninguna publicación sobre una implementación de este tipo, ni siquiera de los resultados de la vectorización de otros algoritmos de metamorfosis de imágenes, con lo cual nuestro aporte es, en pequeña dosis, novedoso.

Figura 2: Cross-dissolving con warping

Organizamos esta exposición en cuatro partes. En primer lugar, proveemos un sencillo marco teórico para comprender la matemática atrás de esta técnica. En segundo lugar, explicamos el algoritmo de Beier y Neely, con el máximo grado de detalle posible. En tercer lugar, desarrollaremos la vectorización realizada, contrastando esta implementación contra la tradicional. Finalmente, presentamos los experimentos y, en base a ello, concluimos sobre la efectividad de la vectorización realizada.

3. Background matemático

3.1. Elementos de Álgebra Lineal

En esta sección hacemos breve repaso de las herramientas matemáticas necesarias para comprender la razón de ciertos cómputos involucrados en la técnica de image morphing en la que nos basamos. Debemos notar, sin embargo, que la matemática es sólo un elemento auxiliar para esta técnica, y no aporta demasiado al entendimiento general de la misma. En esta sintética exposición de álgebra lineal, no usaremos plena generalidad y en algunos casos priorizaremos la intuición antes que la rigurosidad.

Dado que la técnica de Beier y Neely procesa a la imagen como un plano de dos dimensiones (y no maneja, por ejemplo, la profundidad de la imagen), hablaremos siempre en el contexto del $\mathbb{R}$-espacio vectorial $\mathbb{R}^2$. Durante todo este trabajo, las negritas minúsculas $\mathbf{x}$ serán vectores, y las itálicas minúsculas x serán escalares. Como el espacio vectores que estamos considerando es un conjunto de puntos del plano, usaremos los términos *punto* y *vector* indistintamente.

Recordemos que el producto interno canónico en $\mathbb{R}^2$ es la función $\langle \cdot, \cdot \rangle_2 : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}$ tal que

$$\langle (x_1, y_1), (x_2, y_2) \rangle_2 = x_1 x_2 + y_1 y_2$$

Como todo producto interno, induce una norma, que es la función $||\cdot||_2 : \mathbb{R}^2 \to \mathbb{R}$ tal que

$$||\mathbf{p}||_2 = \sqrt{\langle \mathbf{p}, \mathbf{p} \rangle_2}$$

que reescribiendo $\mathbf{p} = (x, y)$ queda

$$||(x, y)||_2 = \sqrt{x^2 + y^2}$$

A su vez, toda norma induce una distancia entre vectores, que es la función $d_2 : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}$ tal que

$$d_2(\mathbf{p}, \mathbf{q}) = ||\mathbf{p} - \mathbf{q}||_2$$

y al reescribir $\mathbf{p} = (x_1, y_1)$ y $\mathbf{q} = (x_2, y_2)$ queda

$$d_2((x_1, y_1), (x_2, y_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

que ésta es la conocida distancia euclídea. En lo que sigue, y por simplicidad, notaremos $\langle \cdot, \cdot \rangle$, $||\cdot||$ y d a las tres funciones anteriores.

A partir de la distancia entre vectores surge el concepto de distancia entre un vector y un subespacio. Recordemos que un subespacio no es más que un subconjunto de vectores que es, en sí mismo, un espacio vectorial. Por ejemplo, en $\mathbb{R}^2$, las rectas que pasan por el origen son subespacios. La distancia entre un vector $\mathbf{p}$ y un subespacio L es

$$d(\mathbf{p}, L) = \min_{\mathbf{x} \in L} d(\mathbf{p}, \mathbf{x})$$

Los subespacios que vamos a considerar son rectas por el origen, que son los únicos subespacios propios no nulos de $\mathbb{R}^2$, con lo cual vamos a estar calculando la distancia entre un punto y una recta. Es posible calcular fácilmente esta distancia, y para esto vamos a caracterizar el único punto del subespacio que realiza el mínimo.

El concepto clave para dar con esta caracterización es el de *proyección ortogonal* de un vector sobre un subespacio. Supongamos que tenemos un punto $\mathbf{p}$ y una recta L, y tracemos la recta perpendicular $L^\perp$. El punto $\mathbf{p}$ se puede escribir como la suma de un punto $\mathbf{q} \in L$ y otro punto $\mathbf{r} \in L^\perp$. En la figura 3 podemos ver la situación.

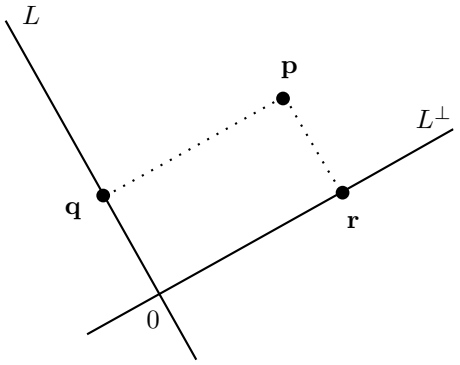

Figura 3: Proyección ortogonal

En este escenario, el punto $\mathbf{q}$ es la proyección de $\mathbf{p}$ sobre el subespacio L, en la dirección de la recta ortogonal $L^\perp$. Esto es lo que se denomina, más sintéticamente, proyección ortogonal de $\mathbf{p}$ sobre L. Como indica la intuición, éste es el punto de L más cercano a $\mathbf{p}$, es decir que

$$d(\mathbf{p}, L) = d(\mathbf{p}, \mathbf{q})$$

Además, esta distancia es exactamente la longitud del vector $\mathbf{r}$, con lo cual

$$d(\mathbf{p}, \mathbf{q}) = ||\mathbf{r}||$$

Como L es una recta, podemos escribir $L = \langle \mathbf{u} \rangle = \{\alpha \mathbf{u} : \alpha \in \mathbb{R}\}$ para cierto vector $\mathbf{u}$, un vector director de la recta. Como $\mathbf{q}$ es un punto de L, existe $u \in \mathbb{R}$ tal que $\mathbf{q} = u\mathbf{u}$. Este u es el coeficiente que determina la posición de $\mathbf{q}$ sobre la recta L. Se puede demostrar que

$$u = \frac{\langle \mathbf{p}, \mathbf{u} \rangle}{||\mathbf{u}||^2} \tag{1}$$

Análogamente, si $L^\perp = \langle \mathbf{w} \rangle$ y $\mathbf{r} = w\mathbf{w}$, entonces

$$w = \frac{\langle \mathbf{p}, \mathbf{w} \rangle}{||\mathbf{w}||^2}$$

Como $L^\perp$ es la recta perpendicular a L, hay una fuerte relación entre los vectores directores de ambas rectas. Si $\mathbf{u} = (x, y)$, llamamos $perp(\mathbf{u}) = (y, -x)$ al vector que se obtiene de rotar $\pi/2$ radianes en sentido antihorario a $\mathbf{u}$. Se puede ver que $perp(\mathbf{u})$ es perpendicular a $\mathbf{u}$ y tiene la misma norma. Este vector $perp(\mathbf{u})$ resulta ser un director de $L^\perp$, con lo cual podemos poner $\mathbf{w} = perp(\mathbf{u})$, y por lo tanto

$$w = \frac{\langle \mathbf{p}, perp(\mathbf{u}) \rangle}{||\mathbf{u}||^2} \tag{2}$$

En definitiva, las ecuaciones 1 y 2 caracterizan las proyecciones $\mathbf{q}$ y $\mathbf{r}$ en función del punto proyectado $\mathbf{p}$ y un vector director $\mathbf{u}$ de la recta sobre la que se proyecta.

3.2. Interpolación Lineal

Dados dos puntos (x_1, y_1) y (x_2, y_2), existe una única recta que los interpola. Si $x_1 \neq x_2$, esta recta es descripta por la fórmula $y(x) = ax + b$, donde

$$a = \frac{y_2 - y_1}{x_2 - x_1}$$
$$b = y_1 - ax_1$$

Los puntos de la recta serán de la forma $(x, y(x))$.

Si $x_1 = x_2$, podemos expresar la recta sencillamente como $x(y) = x_1$, y sus puntos son de la forma $(x(y), y)$.

Podemos expresar a ambas rectas, como una parametrización $\alpha : [0, 1] \to \mathbb{R}^2$, haciendo que la variable independiente varíe entre los extremos de la interpolación, a medida que el argumento t de la parametrización $\alpha(t)$ varía de entre 0 y 1.

Si $x_1 \neq x_2$, esta parametrización tiene la forma $\alpha(t) = (x(t), y(x(t)))$, donde

$$x(t) = (x_2 - x_1)t + x_1$$
$$y(x(t)) = a(x_2 - x_1)t + ax_1 + b$$

Si $x_1 = x_2$, la parametrización tiene la forma $\alpha(t) = (x(y(t)), y(t))$, donde

$$x(y(t)) = x_1$$
$$y(t) = (y_2 - y_1)t + y_1$$

La expresión paramétrica tiene una gran ventaja respecto de la forma explícita (dejando una variable libre, mediante la cual se expresa la restante), que es que permite manejar los dos casos

$x_1 = x_2$ y $x_1 \neq x_2$ de la misma manera, expresando, en ambos casos, las dos componentes de la parametrización como funciones lineales de t. Por el contrario, la dependencia entre las variables en la forma explícita cambia según el caso, siendo y dependiente de x si $x_1 = x_2$, y x dependiente de y si $x_1 \neq x_2$.

4. Algoritmo de Beier y Neely

4.1. Metamorfosis dirigida por segmentos

Para mapear las características de la imagen origen en las características de la imagen destino, esta técnica utiliza segmentos dirigidos. A través de ellos podemos indicar que todos los puntos sobre un sector de la imagen origen deben ser mapeados a cierto otro sector de la imagen destino. Estos segmentos establecen un campo de influencia en su entorno, de modo tal que todos los pixeles que se encuentren en estas inmediaciones serán mapeados al entorno del segmento destino asociado.

4.2. Deformación de una imagen

Si bien estos pares de segmentos nos dan una idea de cómo debe ser la transformación, no es evidente el lugar exacto de la imagen destino en el que debe ser mapeado cada pixel de la imagen origen, ni tampoco cómo deben moverse estos pixeles sobre las imágenes intermedias, a lo largo de la deformación.

¿Cómo construimos una imagen intermedia? Cada una de éstas estará compuesta por pixeles que provienen de la imagen origen y de la imagen destino. La pregunta es cuáles son esos pixeles fuente.

Hay dos formas de mapear pixeles entre las imágenes origen y destino, y la imagen intermedia. La más intuitiva, aunque poco efectiva, es el mapeo hacia adelante (en inglés, *forward mapping*), que consiste en mapear pixeles desde las imágenes fuente (las imágenes origen y destino) hacia la imagen intermedia. Lo malo de esta técnica es que podrían quedar pixeles de la imagen intermedia sin pintar.

La alternativa es el mapeo hacia atrás (en inglés, *reverse mapping*) y consiste en, a contrapelo de lo anterior, mapear pixeles desde la imagen intermedia, hacia los extremos. En otras palabras, para cada pixel de la imagen intermedia, determina qué pixel de cada imagen fuente le corresponde. Esto asegura que cada pixel de la imagen intermedia obtendrá un color.

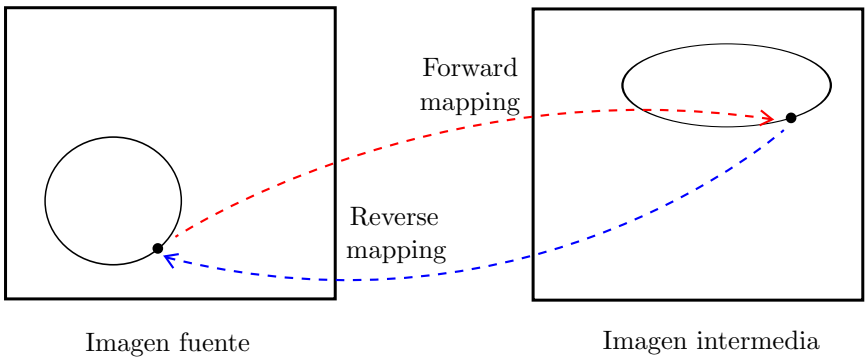

Figura 4: Forward mapping y reverse mapping

4.3. Esquema general de la metamorfosis

Fijados los pares de segmentos en la imagen origen y destino, interpolamos los extremos de cada uno de estos pares, de modo tal de determinar la ubicación de estos segmentos en cada una de

las imágenes intermedias en la metamorfosis. En este trabajo optamos por interpolar linealmente, aunque bien podría utilizarse otro método.

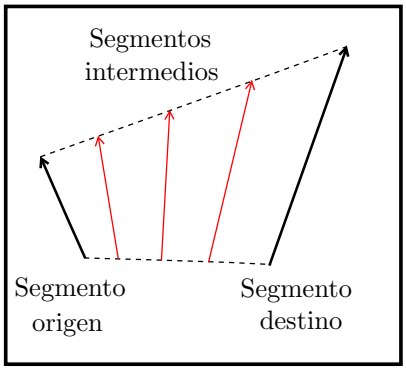

Figura 5: Interpolación de segmentos

Para cada imagen intermedia a generar, determinamos las posiciones de los segmentos en dicha imagen. Luego, para cada pixel de esta imagen determinamos de qué pixel de la imagen original proviene, basándonos en la posición actual de los segmentos. Repetimos el proceso, ahora para determinar de qué pixel de la imagen destino proviene. Finalmente, hacemos una mezcla (en inglés, *blending*) de los pixeles origen y destino calculados, como parte del cross-dissolving. Esta mezcla consiste en la suma ponderada, componente a componente RGB, de los pixeles involucrados, en la que el peso de cada componente depende del momento de la metamorfosis que estamos construyendo. Hacia el principio de la metamorfosis, el color del pixel original dominará, mientras que hacia el final, el color del pixel destino lo hará.

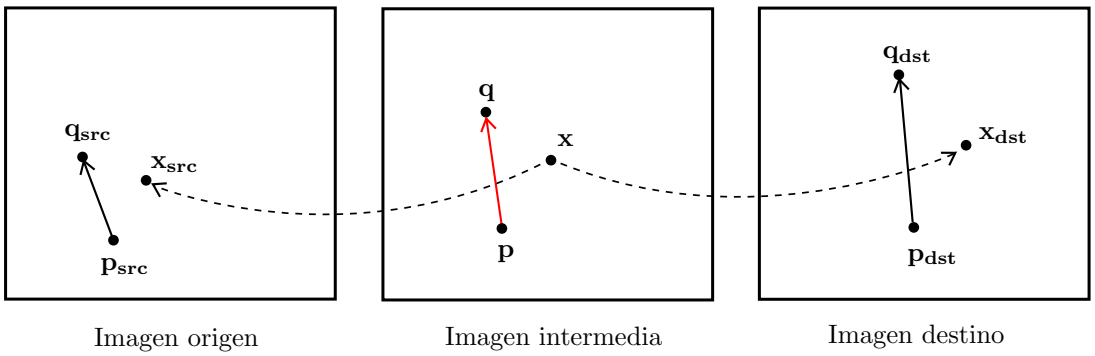

Figura 6: Construcción de una imagen intermedia

4.4. Transformación con un único segmento

Supongamos por el momento que hay un único segmento definido. En el contexto de la figura 6, ¿cómo hacemos para encontrar la coordenada $\mathbf{x_{src}}$? Lo que haremos es encontrar la posición del punto $\mathbf{x}$ relativa al segmento dirigido $\mathbf{pq}$ y encontrar el pixel correspondiente a esta misma posición relativa pero con respecto al segmento dirigido $\mathbf{p_{src}q_{src}}$. Para esto, vamos a utilizar la proyección de $\mathbf{x}$ sobre la recta que determina $\mathbf{pq}$, y sobre su ortogonal.

Lo primero que haremos es trasladar todo al origen, por lo que todos los cómputos los haremos en términos del punto $\mathbf{x} - \mathbf{p}$ y las rectas de directores $\mathbf{q} - \mathbf{p}$ y $\mathbf{q_{src}} - \mathbf{p_{src}}$. Por la ecuación 1, el coeficiente de la proyección de $\mathbf{x} - \mathbf{p}$ sobre la recta $\langle \mathbf{q} - \mathbf{p} \rangle$ es

$$u = \frac{\langle \mathbf{x} - \mathbf{p}, \mathbf{q} - \mathbf{p} \rangle}{||\mathbf{q} - \mathbf{p}||^2} \tag{3}$$

de modo tal que la proyección es exactamente $u(\mathbf{q} - \mathbf{p})$, es decir, la proyección se encuentra en el múltiplo u del vector $\mathbf{q} - \mathbf{p}$. Luego,

$$u(\mathbf{q_{src}} - \mathbf{p_{src}}) \tag{4}$$

es la proyección asociada a $\mathbf{x_{src}}$, proporcional al segmento $\mathbf{p_{src}q_{src}}$. Notar que las proporciones se preservan porque la magnitud u es una proporción. Intuitivamente, hemos determinado a qué *altura* del segmento $\mathbf{p_{src}q_{src}}$ se encuentra $\mathbf{x_{src}}$.

Para determinar cuán lejos se encuentra, vamos a usar la proyección sobre el ortogonal $\langle \mathbf{q}-\mathbf{p} \rangle^{\perp}$. El coeficiente de la proyección es, según la ecuación 2,

$$w = \frac{\langle \mathbf{x} - \mathbf{p}, perp(\mathbf{q} - \mathbf{p}) \rangle}{||\mathbf{q} - \mathbf{p}||^2}$$

Podríamos usar este coeficiente para ubicar $\mathbf{x_{src}}$, aunque esto significaría que las proporciones también se preservan en la dirección ortogonal a $\mathbf{pq}$. Según indican los creadores de la técnica en su artículo, es más util *no* mantener la escala en la dirección ortogonal al segmento. En otras palabras, la distancia del punto $\mathbf{x_{src}}$ a la recta $\langle \mathbf{q_{src}} - \mathbf{p_{src}} \rangle$ debe ser la misma que del punto $\mathbf{x}$ a la recta $\langle \mathbf{q} - \mathbf{p} \rangle$, sumado a que los puntos deben preservar su ubicación con respecto a los segmentos, en el sentido de que o bien ambos puntos se encuentran a izquierda de los segmentos, o bien ambos se encuentran a derecha. Nosotros hemos seguido su consejo.

Notemos que podemos escribir la proyección $w\ perp(\mathbf{q} - \mathbf{p})$ como $\frac{\langle \mathbf{x}-\mathbf{p}, perp(\mathbf{q}-\mathbf{p}) \rangle}{||\mathbf{q}-\mathbf{p}||} \frac{perp(\mathbf{q}-\mathbf{p})}{||\mathbf{q}-\mathbf{p}||}$. Llamemos

$$v = \frac{\langle \mathbf{x} - \mathbf{p}, perp(\mathbf{q} - \mathbf{p}) \rangle}{||\mathbf{q} - \mathbf{p}||} \tag{5}$$

La clave es que, como $\frac{perp(\mathbf{q}-\mathbf{p})}{||\mathbf{q}-\mathbf{p}||}$ es unitario y tiene la dirección y el sentido de $perp(\mathbf{q}-\mathbf{p})$, entonces

$$v\ \frac{perp(\mathbf{q_{src}} - \mathbf{p_{src}})}{||\mathbf{q_{src}} - \mathbf{p_{src}}||} \tag{6}$$

es una componente en dirección ortogonal a $\mathbf{p_{src}q_{src}}$ que preserva distancia y sentido, en la forma en que se indicó en el párrafo anterior.

En definitiva, las expresiones 4 y 6 determinan las componentes de $\mathbf{x_{src}} - \mathbf{p}$, de manera que

$$\mathbf{x_{src}} = u(\mathbf{q_{src}} - \mathbf{p_{src}}) + v\ \frac{perp(\mathbf{q_{src}} - \mathbf{p_{src}})}{||\mathbf{q_{src}} - \mathbf{p_{src}}||} + \mathbf{p} \tag{7}$$

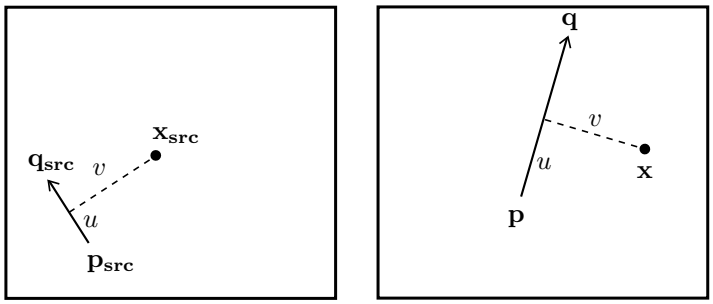

Imagen origen Imagen intermedia

Figura 7: Transformación con un único segmento

Para calcular $\mathbf{x_{dst}}$, el procedimiento es el mismo, ahora utilizando el segmento $\mathbf{p_{dst}q_{dst}}$ en lugar de $\mathbf{p_{src}q_{src}}$.

Teniendo las coordenadas $\mathbf{x_{src}}$ y $\mathbf{x_{dst}}$, sólo resta hacer el blending entre tales puntos para obtener los colores de $\mathbf{x}$. Para cada componente RGB c, ponemos

$$c(\mathbf{x}) = (1-t)\ c(\mathbf{x_{src}}) + t\ c(\mathbf{x_{dst}}) \tag{8}$$

donde $t \in [0,1]$ es el tiempo transcurrido en la metamorfosis.

4.5. Transformación con múltiples segmentos

En general, las metamorfosis en las que estemos interesados serán complejas y requerirán más de un segmento. Cuando hay más de un segmento, cada punto no se ve afectado por un único segmento cercano, sino por todos, que ejercen influencia en mayor o menor medida. Para determinar el pixel origen correspondiente a un punto $\mathbf{x}$, vamos a tomar cada uno de los segmentos de la imagen intermedia, y su segmento asociado en la imagen origen, y hacer el reverse mapping como si el segmento considerado fuera el único. El procesamiento para el i-ésimo segmento, que llamamos $\mathbf{p}[i]\mathbf{q}[i]$, resultará en un punto $\mathbf{x_{src}}[i]$, lo que significa que, según $\mathbf{p}[i]\mathbf{q}[i]$, $\mathbf{x}$ se desplaza en $\mathbf{d}[i] = \mathbf{x_{src}}[i] - \mathbf{x}$. Cada uno de estos desplazamientos tendrá un peso, que será inversamente proporcional a la distancia de $\mathbf{x}$ a $\mathbf{p}[i]\mathbf{q}[i]$, y directamente proporcional a la longitud de $\mathbf{p}[i]\mathbf{q}[i]$. El desplazamiento definitivo, con el que determinaremos $\mathbf{x_{src}}$, será el promedio ponderado de estos desplazamientos parciales.

El peso que utilizaremos es

$$weight = \left(\frac{||\mathbf{q}[i] - \mathbf{p}[i]||^c}{a + d(\mathbf{x}, \mathbf{p}[i]\mathbf{q}[i])} \right)^b \tag{9}$$

donde a, b y c son constantes prefijadas. La constante a cumple la función de que la expresión no se indefina para los puntos que se encuentran sobre el segmento $\mathbf{p}[i]\mathbf{q}[i]$, por lo que se debe elegir apenas mayor que cero. La constante c determina cuán influyente es la longitud de un segmento, de modo tal que a valores grandes de c, más influencia tienen los segmentos largos. Finalmente, la constante b indica cuánto cae la fuerza de un segmento en función de la distancia a la que se encuentra. Si b es grande, cada punto será afectado sólo por los segmentos que están cerca. Notar que si b es cero, todos los segmentos afectan a todos los pixeles en igual medida.

Para calcular la distancia $d(\mathbf{x}, \mathbf{p}[i]\mathbf{q}[i])$, podemos usar el mismo coeficiente u calculado usando la fórmula 3. Si $u \in [0,1]$ entonces la proyección de $\mathbf{x}$ cae sobre el segmento $\mathbf{p}[i]\mathbf{q}[i]$, y el punto de la proyección realiza la distancia, y se puede ver que esta distancia es $|v|$, para el valor de v calculado en la ecuación 5. Si $u < 0$ entonces la proyección cae en un punto que tiene sentido contrario al segmento, por lo que la distancia se realiza sobre el extremo $\mathbf{p}[i]$, y vale $||\mathbf{x} - \mathbf{p}[i]||$. Finalmente, si $u > 1$, la distancia se realiza sobre el otro extremo $\mathbf{q}[i]$, y vale $||\mathbf{x} - \mathbf{q}[i]||$.

Terminamos esta explicación presentando el algoritmo completo. El algoritmo 1 es el encargado de, dado un pixel $\mathbf{x}$, encontrar el pixel $\mathbf{x'}$ correspondiente en una imagen fuente. En otras palabras, calcula $\mathbf{x_{src}}$ o $\mathbf{x_{dst}}$ según se use los segmentos de la imagen origen o los de la imagen destino, respectivamente. El algoritmo 2 es la transformación completa.

Algorithm 1: Cómputo de pixel fuente

Input : $\mathbf{x}$ pixel de la imagen intermedia
$\mathbf{p}[]\mathbf{q}[]$ segmentos de la imagen intermedia
$\mathbf{p'}[]\mathbf{q'}[]$ segmentos de la imagen fuente
Output : $\mathbf{x'}$ pixel de la imagen fuente

1 **d_sum** $\leftarrow (0, 0)$
2 $weight\_sum \leftarrow 0$
3 Sea s la cantidad de segmentos
4 **for** $i \leftarrow 1$ **to** s **do**
5 Calcular u y v en base a $\mathbf{p}[i]\mathbf{q}[i]$ y $\mathbf{x}$ (ecuaciones 3 y 5)
6 Calcular $\mathbf{x'}[i]$ en base a u, v, $\mathbf{p'}[i]\mathbf{q'}[i]$ y $\mathbf{p}[i]$ (ecuación 7)
7 $\mathbf{d}[i] \leftarrow \mathbf{x'}[i] - \mathbf{x}$
8 Calcular $weight$ en base a $\mathbf{p}[i]\mathbf{q}[i]$ y $\mathbf{x}$ (ecuación 9)
9 **d_sum** $\leftarrow$ **d_sum** $+ weight * \mathbf{d}[i]$
10 $weight\_sum \leftarrow weight\_sum + weight$
11 **end**
12 $\mathbf{x'} \leftarrow \mathbf{x} +$ **d_sum**$/weight\_sum$
13 **return** $\mathbf{x'}$

Algorithm 2: Metamorfosis

Input : Imagen origen
Imagen destino
$\mathbf{p_{src}}[]\mathbf{q_{src}}[]$ segmentos de la imagen origen
$\mathbf{p_{dst}}[]\mathbf{q_{dst}}[]$ segmentos de la imagen destino

1 Sea s la cantidad de segmentos
2 **for** $i \leftarrow 1$ **to** s **do**
3 Calcular la interpolación lineal de $\mathbf{p_{src}}[i]$ y $\mathbf{p_{dst}}[i]$ de forma paramétrica
4 Calcular la interpolación lineal de $\mathbf{q_{src}}[i]$ y $\mathbf{q_{dst}}[i]$ de forma paramétrica
5 **end**
6 **for** *t corriendo entre 0 y 1* **do**
7 **foreach** *píxel* $\mathbf{x}$ **do**
8 Calcular $\mathbf{x_{src}}$ en base a $\mathbf{x}$, los segmentos en tiempo t y los segmentos de la imagen origen (algoritmo 1)
9 Calcular $\mathbf{x_{dst}}$ en base a $\mathbf{x}$, los segmentos en tiempo t y los segmentos de la imagen destino (algoritmo 1)
10 Colorear $\mathbf{x}$ mezclando $\mathbf{x_{src}}$ y $\mathbf{x_{dst}}$ (ecuación 8)
11 **end**
12 Escribir la imagen intermedia
13 **end**

En el algoritmo 2 no está especificada la forma en que t toma valores entre 0 y 1. En la práctica, esto está determinado por la cantidad de imágenes que queremos que compongan la metamorfosis (es decir, la imagen origen, más las imágenes intermedias, más la imagen destino). Si llamamos f a esta cantidad , entonces t tomará los valores $i/(f-1)$, para $i = 0, \ldots, f-1$, en ese orden.

4.6. Complejidad temporal

El algoritmo 1 tiene un costo $O(s)$, puesto que el ciclo 4-11 se ejecuta s veces, y cada una de sus operaciones es $O(1)$.

Las líneas 2 a 5 del algoritmo 2, en las cuales se computan las interpolaciones, cuestan $O(s)$ en total, puesto que son $2s$ interpolaciones en total y cada una cuesta $O(1)$, como se puede ver en la sección 3.2. Los ciclos anidados 6-13 se ejecutan $f \times w \times h$ veces, donde w es el ancho de las imágenes y h es el alto. Las líneas 8 y 9 son $O(s)$ cada una, mientras que la línea 10 es $O(1)$. Por lo tanto, el ciclo 6-13 es $O(f \times w \times h \times s)$ y, en definitiva, éste es el costo del algoritmo.

5. Vectorización

En los algoritmos 1 y 2 se puede ver que técnica se compone de varias partes, claramente divididas. Primero aparece el cómputo de las interpolaciones. Luego tenemos la metamorfosis propiamente dicha, etapa en la que se construye cada una de las imágenes intermedias. Para cada una de estas imágenes, se itera sobre todos los pixeles, y se realizan dos tipos de operaciones: reverse mapping y blending.

Dado que la cantidad de segmentos s suele ser pequeña, se decidió no vectorizar el cómputo de las interpolaciones. El blending tampoco fue vectorizado, puesto que, suponiendo que tenemos varios píxeles de la imagen origen y destino que mezclar, es muy probable que no ocupen posiciones consecutivas, forzando a la realización de múltiples lecturas de memoria para obtener las componentes RGB de tales pixeles. Por esta razón, el único cómputo en paralelo posible aquí sería la cuenta de la ecuación 8. Si bien esto podría significar una ganancia de tiempo, es despreciable respecto del costo en que incurrimos al realizar los accesos a memoria, por lo que decidimos no vectorizarlo.

En donde sí existía una gran posibilidad de vectorización, es en el reverse mapping, es decir, el algoritmo 1, que tiene dos características propicias. Por un lado, la independencia entre el procesamiento de distintos pixeles, puesto que cómputo del algoritmo para un pixel dado es independiente del cómputo para cualquier otro pixel. Por otro lado, la casi inexistencia de branching. El único momento en que debemos tomar decisiones es al calcular la distancia $d(\mathbf{x}, \mathbf{p}[i]\mathbf{q}[i])$ en la línea 8, aunque es posible evitar el branching computando el resultado de todas las ramas de ejecución posibles, quedándonos con el resultado de la que corresponda, a través de máscaras de bits. Esta técnica es clásica en el contexto de la vectorización de bloques de decisión, de modo que no profundizaremos en su implementación.

La vectorización de este algoritmo logra computar, para una entrada de cuatro pixeles, el pixel fuente que le corresponde cada uno. La idea atrás de esta implementación es tomar todas las magnitudes computadas en el algoritmo estándar (u, v, las coordenadas de todos los vectores involucrados, etc.), y disponer cada una en un registro XMM distinto. De esta forma, cada registro XMM contiene cuatro valores correspondientes a una misma magnitud, permitiendo el procesamiento simultáneo de cuatro instancias del algoritmo.

Es cuatro es la cantidad de pixeles que se pueden procesar al mismo tiempo, debido a que todas las magnitudes involucradas son números de punto flotante de precisión simple, que tienen 4B de longitud, mientras que los registros XMM tienen 16B de longitud. Todos los cómputos fueron realizados con representación de punto flotante de dicha precisión.

6. Experimentación

6.1. Resultados

Comparamos una implementación en C, con la vectorizada[1]. Para esto, tomamos dos imagenes de tamaño 1024×768, y trazamos 44 segmentos sobre cada una de ellas. En la figura 2 se puede ver la metamorfosis utilizando los 44 segmentos.

Para la comparación, decidimos dejar fijas las imágenes origen y destino, es decir que $w = 1024$ y $h = 768$, fijar la cantidad de frames de la metamorfosis en $f = 100$, y sólo variar la cantidad de segmentos s utilizados. Esta elección se basa en que el parámetro s es el que determina la intensidad del cómputo de cada una de las llamadas individuales a la implementación del algoritmo 1, que es la pieza que hemos vectorizado. Los parámetros f, w y h solamente marcan cuántas veces se llama a ese procedimiento.

Para cada una de las implementaciones, variando $s \in \{0, 5, 10, 15, 20, 25, 30, 35, 40\}$, medimos el tiempo de ejecución total del algoritmo 2. Esto incluye el overhead que impone el framework utilizado para la escritura de los frames del video a lo largo de todo el proceso, al finalizar cada iteración. Como veremos a continuación, este overhead es despreciable respecto del resto del procesamiento.

En la figura 8 presentamos los resultados de la comparación. Ambas versiones fueron compiladas mediante la herramienta de Intel, cuyo grado de optimización es mucho mayor que otros compiladores, como GCC. Además se utilizó el flag -O3, la opción de optimización más agresiva. Medimos el tiempo de ejecución en cantidad de ciclos de reloj, utilizando el registro TSC (Time Stamp Counter) de un procesador Intel®Core™i5-3470 compuesto de cuatro cores de 3.20GHz cada uno.

Recordemos que el costo del algoritmo 2 es proporcional a $f \times w \times h \times s$, de modo que el gráfico de tiempo de ejecución en función de s será una recta con pendiente proporcional a $f \times w \times h$.

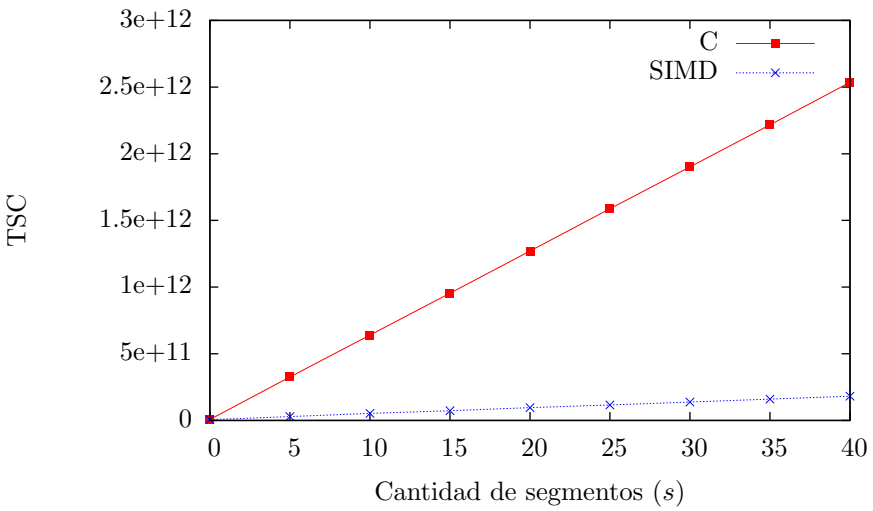

Figura 8: Comparación C vs. SIMD

La siguiente tabla muestra los tiempos de ejecución exactos, permitiendo calcular la diferencia de tiempo entre las implementaciones.

[1]El código de ambas se puede encontrar en `https://github.com/guidotag/Image-Morphing`.

s	C	SIMD	C / SIMD
0	7309812582	5941940863	1.23
5	326478638311	29467020103	11.08
10	639832096193	52546672036	12.18
15	954306269556	73077639985	13.06
20	1269998392581	96104580737	13.21
25	1587171271314	116337177498	13.64
30	1901684635986	138126686002	13.77
35	2216291340338	160337691233	13.82
40	2535347665489	182194876866	13.91

En el caso $s = 0$ se escribe el video casi sin realizar cómputos durante el algoritmo 1, por lo que es una buena medida del overhead que impone el procesamiento de video. Observemos que el tiempo en ese caso es del orden de 10^{10} ciclos de clock, mientras que para $s > 0$ todos los valores van de los 10^{12} a los 10^{13} ciclos, es decir, de 100 a 1000 veces más. Esto indica que el overhead en cuestión va del 1 % al 0,1 %, una cantidad despreciable de trabajo del procesador.

Como se ve, la implementación SIMD es, a medida que s crece, casi 14 veces más rápida. Si bien la velocidad de crecimiento de la razón de los tiempos disminuye drásticamente a partir de $s = 15$, no se ve un estancamiento total de dicho valor, con lo cual podría seguir incrementándose para valores de s más grandes que 40. En este trabajo no exploraremos qué sucede más allá de $s = 40$, pues no son valores de interés práctico.

6.2. Análisis

En general, al vectorizar procedimientos obtenemos ganancias múltiplo de la cantidad de datos procesados al mismo tiempo. Sin embargo, en este caso, la vectorización supera ampliamente esa expectativa, al ser 14 veces más rápida, pese a que cada ejecución procesa sólo 4 pixeles en paralelo. En busca de una explicación para este fenómeno, comparamos con lupa ambas implementaciones.

La implementación con extensiones SIMD tiene dos características remarcables, directamente relacionadas con su performance:

- **Casi todas las magnitudes involucradas se mantienen, todo el tiempo, en registros.** Esto permite evitar accesos a memoria, que son más de 100 veces más costosos que los accesos a registros, en el caso de un cache miss.
- **Todas las funciones auxiliares están implementadas en forma de macros.** Si bien esto permite reducir el overhead de una implementación que usa etiquetas e instrucciones `call`, la motivación verdadera fue hacer la programación más sencilla, puesto que, en el intento de mantener la mayoría de los datos en registros, éstos debían ser manipulados cuidadosamente. Utilizando macros podemos decidir exactamente qué registros queremos que sean utilizados por una rutina.

Ambas características tienen desventajas. Por un lado, al intentar explotar al máximo la utilización de los registros, se hace mucho más complicada la programación, al ser limitada la cantidad de registros que tenemos a nuestra disposición. Sortear este problema es simplemente una cuestión de planificar bien el programa. Por otro lado, la utilización de macros tiene dos desventajas. Primero, no es posible una utilización dinámica del código de la macro, en el sentido de que cada invocación a ella replicará su código, haciendo que el código objeto de todo el programa sea mucho más largo. En nuestro caso, el código objeto de la vectorización consta de algunas cientos de líneas, de modo que no resulta una verdadera contra. En segundo lugar, no es posible depurar las líneas que componen a la macro, haciendo más difícil la detección y corrección de errores.

Para realizar un análisis detallado de la versión C, hicimos un dump de su código objeto, siendo dos las observaciones que se desprenden del mismo:

- **La cantidad neta de líneas del reverse mapping y todas las funciones auxiliares utilizadas, es de 750 líneas.** Esto es más del doble del tamaño de la versión SIMD, que

tiene alrededor de 350. Si bien un programa más compacto no es necesariamente mejor, en este caso tenemos 350 líneas que fueron cuidadosamente pensadas, contrariamente a las 750 elaboradas por un compilador, lo cual podría indicar que la mano humana está haciendo una diferencia.

- **Casi todas las variables locales involucradas en el ciclo principal del algoritmo fueron almacenadas en stack.** Esto obliga a hacer lecturas de memoria en cada llamada a una función auxiliar, y escrituras de memoria para almacenar su resultado. Resulta claro el contraste con la implementación en SIMD, en la que se buscó mantener todos los datos que fueran posibles en los registros.

Como se puede ver, la diferencia que parece estar inclinando la balanza a favor de la versión SIMD (aún más de lo que ya lo hace la vectorización) es el predominante uso de la memoria principal de la versión C. Para corroborar este hecho y comparar otras métricas de performance, realizamos un análisis con la herramienta *perf* de Linux. Tomamos el caso $s = 20$ y corrimos el programa completo, una vez utilizando la implementación en C, y otra con el procedimiento SIMD. Los resultados fueron los siguientes:

	C	**SIMD**
Ciclos de clock	$1{,}43 \times 10^{12}$	$1{,}07 \times 10^{11}$
Instrucciones	$9{,}69 \times 10^{11}$	$2{,}03 \times 10^{11}$
Instrucciones por ciclo	0,68	1,90
Referencias a cache	$1{,}96 \times 10^{7}$	$1{,}87 \times 10^{7}$
Cache misses	61,55 %	68,20 %
Branches	$1{,}20 \times 10^{11}$	$1{,}22 \times 10^{9}$
Branch misses	0,60 %	0,73 %
Lecturas de memoria	$2{,}70 \times 10^{9}$	$4{,}17 \times 10^{8}$
Escrituras de memoria	$1{,}99 \times 10^{11}$	$5{,}50 \times 10^{9}$

Los números confirman que la cantidad de accesos a memoria de la implementación SIMD es órdenes de magnitud menor. A esto se suma que la vectorización aprovecha mucho más al procesador, al ejecutar casi el triple de instrucciones por ciclo que la implementación en C. Por último, es remarcable la diferencia en la cantidad de branches (instrucciones de salto ejecutadas) que resulta 100 veces menor en la vectorización, siendo el porcentaje de errores de predicción aproximadamente el mismo, con lo cual, en total, la cantidad de errores de predicción también es 100 veces menor en la vectorización.

Referencias

[1] Thaddeus Beier and Shawn Neely. Feature-based image metamorphosis. *SIGGRAPH Comput. Graph.*, 26(2):35–42, July 1992.