Scissors Intelligent

Autores:

- Jesús Maillo Hidalgo
- María Reyes Gentil

INDICE

1. Descripción del problema.

Una de las funcionalidades necesarias en cualquier editor de imágenes actual es dar la posibilidad al usuario de recortar fácilmente un objeto o zona de la imagen a tratar. Esta herramienta suele aparecer bajo el nombre de tijeras de selección en GIMP® o como lazo imantado en Photoshop®. Han tenido una gran acogida gracias a su fácil uso, dando versatilidad tanto a profesionales del diseño gráfico como al usuario medio.

El software desarrollado realiza esta funcionalidad siendo una versión simplificada de un editor para tratamiento de imágenes. De esta forma, se da la posibilidad de continuar con el desarrollo del mismo mejorando las funcionalidades ya existentes y añadiendo nuevas herramientas para convertirlo en un editor especializado a disposición de cualquier usuario.

El software se apoya en una interfaz gráfica sencilla e intuitiva para el usuario basada en un frame para la imagen y tres botones que le permiten al usuario utilizar esta herramienta cómodamente y sin necesidad de tener un conocimiento previo sobre la misma. A través del conjunto de botones el usuario podrá seleccionar la imagen sobre la que quiere trabajar, completar el último tramo evitándole la seleccionar exacta en el punto inicial y cerrar la aplicación.

2. Algoritmo (Pseudocódigo)

El algoritmo empleado se basa en una función de coste relativa a los pixeles para obtener el camino óptimo desde cualquier pixel de la imagen al pixel indicado.

Este coste es local entre dos pixeles, viéndose afectado por los cruces por cero de la Laplaciana, la magnitud del gradiente y la dirección del mismo. A continuación, describiremos cada una de ellas:

- Cruces por cero de la Laplaciana (f_z Laplacian Zero-Crossing):
  En primer lugar suavizamos la imagen para eliminar el posible ruido.
  A continuación, aplicamos la Laplaciana a la imagen. Los puntos en los que la Laplaciana de cero, f_z será 0, y los puntos con valor de Laplaciana distinto de cero, f_z valdrá 1.
  Es decir:

$$f_Z(q) = \begin{cases} 0; & \text{if } I_L(q) = 0 \\ 1; & \text{if } I_L(q) \neq 0 \end{cases}$$

- Magnitud del gradiente (f_g Gradient Magnitude)
  En primer lugar calculamos el gradiente en la dirección X y el gradiente en la dirección Y. A partir de estos valores obtendremos un valor G para cada pixel dado por la raíz cuadrada de la suma de cuadrados del gradiente en X e Y.
  El valor de f_g será invertido y normalizado por su máximo. Su expresión matemática es:

$$f_G = \frac{\max(G) - G}{\max(G)} = 1 - \frac{G}{\max(G)}$$

  Finalmente, el valor calculado anteriormente será ponderado por teniendo en cuenta la distancia Euclídea: sí el vecino se encuentra en la diagonal f_g será multiplicado por 1, y si es un vecino de la horizontal o vertical se multiplicará por $1/\sqrt{2}$

- Dirección del gradiente (f_d Gradient Direction)
  Para obtener este valor se deben realizar varias operaciones sobre la pareja de pixeles (pixel central y vecino).
  Comenzamos calculando L(p,q), el enlace o "vector eje" bidireccional entre los pixeles p y q.

La expresión matemática que determina este valor es:

$$L(p,q) = \begin{cases} q-p; & \text{if } D'(p)\cdot(q-p) \geq 0 \\ p-q; & \text{if } D'(p)\cdot(q-p) < 0 \end{cases}$$

Donde:

-p: pixel central

-q: pixel vecino

-D'(p): transpuesta de la dirección del gradiente, siendo la dirección del gradiente D(p)=(Iy(p), -Ix(p)).

El productor escalar de L(p,q) con D' nos dará el escalar que indica la proyección de un vector sobre el otro:

$$d_p(p,q) = D'(p)\cdot L(p,q)$$
$$d_q(p,q) = L(p,q)\cdot D'(q)$$

Finalmente, obtenemos el valor de la dirección del gradiente aplicando la suma de cosenos de dp y dq, diviendola por π. La expresión matemática correspondiente a f_D es:

$$f_D(p,q) = \frac{1}{\pi}\{\cos[d_p(p,q)]^{-1} + \cos[d_q(p,q)]^{-1}\}$$

Estos valores (f_d, f_z y f_g) no tienen la misma importancia, viéndose ponderadas de la siguiente forma:

$$l(p,q) = \omega_Z \cdot f_Z(q) + \omega_D \cdot f_D(p,q) + \omega_G \cdot f_G(q)$$

Donde:

- $W_z = 0.43$
- $W_D = 0.43$
- $W_g = 0.14$

Descripción del algoritmo principal (Pseudocódigo).

Entradas

s: punto inicial (semilla)

l(p,r): coste local explicado en el apartado anterior

Estructuras de datos

L: lista ordenada de pixeles activos junto con su coste (inicialmente vacío).

N(q): los 8 pixeles vecinos de q.

e(q): determinar si el pixel q ha sido expandido.

g(q): coste desde el pixel inicial hasta el pixel q.

Salida

p: estructura que indica el camino óptimo (mínimo coste) desde cada pixel hasta el pixel inicial.

Algoritmo (Pseudocódigo)

```
//Inicialización de la lista L con el pixel semilla de coste 0
L.pixel = s
L.coste = 0

//Mientras que la lista no esté vacía
mientras not L.empty
	//Obtenemos el pixel de menor coste y lo eliminamos de la lista
	q = min(L.pixel)
	g_q = min(L.coste)
	L.eliminar_q

	//Marcamos el pixel q como expandido
	e(q) = verdadero;

	//Para cada vecino de q que no haya sido expandido
	para cada vecino r de N(q) y not e(r)
		//Calculo del coste total para el vecino r
		g_tmp = g_q + l(q,r)

		//Si r está en la lista y g temporal es menor que g de r
		si r está en L y g_tmp < g_r
			//Eliminamos r de la lista
			Eliminar_Punto(L, r);

		//Si r no está en la lista
		si r no está en L
			// Actualizamos el coste de r
			g_r = g_tmp;
			// Añadimos q a p
			p(r) = q;
			//Añadimos r en la lista L de pixeles activos
			L.introducir(r)
```

3. Implementación

Interfaz (interfaz.h)

Campo imagen (`void campoImagen_MouseDown )`

1. Si es el primer click
   1.1. Tomamos las coordenadas del punto y calculamos la matriz de camino óptimo (mínimo) para ese pixel.
   1.2. Pintamos el punto.
   1.3. Refrescamos la imagen (con el punto dibujado).
   1.4. Marcamos con un booleano que el primer click se ha realizado.
2. Si no es el primer click
   2.1. Tomamos las coordenadas del punto.
   2.2. Pintamos el camino desde este punto hasta el punto anteriormente seleccionado.
   2.3. Pintamos el nuevo punto seleccionado.
   2.4. Volvemos a calcular el camino óptimo con este último punto seleccionado.
   2.5. Refrescamos la imagen.

Botón de selección de imagen (`Void selecImagen_Click )`

1. Llamamos al browser para seleccionar la imagen y la mostramos.
2. Pasamos el path de la imagen a tipo std::string.
3. Hacemos una copia de la imagen en blanco y negro para trabajar con ella y otra en color donde aplicaremos los cambios gráficos.
4. Inicializamos la estructura de datos y calculamos la matriz de costes.

Botón terminar (`Void terminar_Click`)

1. Pinta el camino óptimo (mínimo) desde el primer punto hasta el último punto seleccionado.

Botón de salir (`Void cerrar_Click`)

1. Se cierra la aplicación.

Algoritmo (tijeras_inteligentes.cpp / tijeras_inteligentes.h)

Funciones útiles

- void MarshalString(System::String ^ s, string & os)

  Función para pasar de System::String a std::string necesaria para realizar copias de la imagen en disco.

- bool ordenaDecreciente (const pixel_coste & first, const pixel_coste & second);

  Función utilizada para ordenar el vector de pixeles activos.

Inicialización de las estructuras necesarias

- void ini_expandidos(Mat &im, vector< vector <bool> > & expandidos)

  Función para inicializar a false la estructura que indica si un pixel ha sido expandido o no (necesaria en la función calculo_camino_optimo)

- void ini_costes(Mat &im, vector< vector <costes_vecinos> > & m_costes)

  Función para inicializar la matriz de costes, estructura que almacena el coste de un pixel a sus vecinos.

- void ini_costes_final(Mat &im, vector< vector <float> > & m_costes_final)

  Función para inicializar la matriz de mínimo coste. Será una estructura de apoyo en el algoritmo principal para optimizar la búsqueda del coste de un pixel y ahorrar la búsqueda del pixel en la lista.

- void ini_p(Mat &im, vector< vector <Point> > & p)

  Función para inicializar la matriz de puntos. Esta estructura indica el siguiente pixel a seguir para obtener el camino óptimo.

- void actualiza_estructuras(Mat &im, vector< vector <float> > & m_costes_final, vector< vector <Point> > & p, vector< vector <bool> > & expandidos)

  Función para reinicializar los valores de m_costes_final, p y expandidos. Esta función será necesaria a partir del primer click,

Funciones parciales del algoritmo

- bool e(vector< vector <bool> > &expandidos , Point q)

  Función que indica si pixel ha sido expandido.

- int eliminarPunto(vector <pixel_coste> & L, Point p)

  Función que elimina un punto de la lista de pixels activos.

- void pintaPunto(Mat & im, Point p)

  Función para pintar un punto en la imagen.

Funciones principales del algoritmo.

- void cal_m_costes(const Mat &im, vector< vector <costes_vecinos> > & m_costes)

  Función que calcula la matriz de costes en base a la explicación realizada en el apartado 2. Algoritmo.

  1. Definición de las constantes W_z, W_D y W_g .
  2. Inicialización de los márgenes de la estructura. El coste asignado al borde de la imagen es un valor alto para que no evitar que estos pixeles entren en el camino óptimo.
  3. Calculamos f_z utilizando la Laplaciana.
     3.1. Suavizamos la imagen.
     3.2. Aplicamos la Laplaciana.
  4. Calculamos G.
     4.1. Calculamos el gradiente en X y en Y (Sobel).
     4.2. Determinamos la magnitud del gradiente G que será la raíz cuadrada de la suma de cuadrados de los gradientes en X e Y.
  5. Para cada pixel p:
     5.1. Para cada vecino q:
        5.1.1. Calculamos f_z
           - Si la Laplaciana de q vale 0, f_z vale 0.
           - Si la Laplaciana de q vale distinto de 0, f_z vale 1.
        5.1.2. Calculamos f_D
           - A partir de las expresiones matemáticas descritas en el apartado anterior calculamos el valor f_D teniendo en cuenta que valores pequeños de dp y dq (del orden de 10^{-9}) serán tomados como 0.
        5.1.3. Calculamos f_g.
           - Determinamos el valor de f_g
           - Si el vecino se encuentra en la diagonal, f_g será multiplicado por 1. En caso contrario, se multiplicará por $1/\sqrt{2}$
        5.1.4. El coste del vecino será la suma de estos tres componentes multiplicados por sus constantes.

- void calculo_camino_optimo(Mat & im, vector< vector <costes_vecinos> > & m_costes, Point s, vector< vector <bool> > & expandidos, vector< vector <Point> > & p, vector< vector <float> > & m_costes_final)

  Función para calcular la matriz de camino óptimo dado un punto inicial.

  1. Marcamos el pixel semilla con coste 0.
  2. Mientras que haya pixeles activos:
     2.1. Seleccionamos el pixel de menor coste.
     2.2. Marcamos el pixel con coste -1 para indicar que no está activo.
     2.3. Marcamos el pixel como expandido.
     2.4. Para cada vecino no expandido:
        2.4.1. Calculamos su coste
        2.4.2. Si r está activo y g temporal es menor que g_r.
           - Marcamos el punto con coste -1 para indicar que no está activo.
        2.4.3. Si r no está activo:
           - g_r toma el valor de g temporal.
           - Lo marcamos como camino óptimo en la estructura p.
           - Lo marcamos como pixel activo (actualizamos su coste)

- void pintar_camino(Mat & im, vector< vector <Point> > & p, Point t)

  Función que dibuja el camino pixel a pixel utilizando la estructura p. Modifica la imagen que se mostrará al usuario.

5. Valoración de resultados

En este apartado vamos a valorar los resultados obtenidos en distintas imágenes. Analizaremos de manera independiente cada una de ellas:

Cuadrado

La figura más simple para analizar sería el cuadrado negro con el fondo blanco. Gracias a la simpleza del cuadrado se consigue el contorno exacto.

Rombo

Con una imagen algo más compleja como el rombo negro con el fondo blanco llega a reconocer realmente bien los contornos y se ajusta a ellos de una forma bastante razonable.

Aún con una selección de puntos menos amigable para el algoritmo, el resultado es muy bastante aceptable y similar al anterior. Lo cual hace que nuestro algoritmo sea robusto a los cambios de puntos.

Muñeca



En la imagen de la izquierda podemos ver que a pesar de tener un contorno entre el azul y el verde, se continúa ajustando al contorno negro.

Por otro lado, en la imagen central y en la imagen de la derecha vemos que el comportamiento no siempre es el deseado, en la misma imagen pero en otro sector del contorno, los resultados son mucho peores.

En la imagen central vemos que no lo hace mal, pero no trata de ajustar con fuerza al contorno, no se pega a la esquina del brazo de la muñeca.

La imagen de la derecha nos muestra que si puede equivocarse y tomar como contorno el existente entre el azul y el verde.

Lena


En la imagen de la izquierda vemos que se ajusta al contorno de la piel con bastante precisión. Si deseásemos un mejor ajuste en la parte de la cara tan solo necesitaríamos una cantidad de puntos mayor.

En la imagen de la derecha vemos que el cambio de contorno de la piel al sombrero no se realiza bien.

6. Conclusiones

Mediante distintas ejecuciones nos hemos dado cuenta de comportamientos erróneos que comentamos a continuación:

- Con imágenes artificiales como figuras geométricas negras con fondo blanco, se espera que de soluciones perfectas o casi perfectas. Sin embargo, las esquinas de las figuras geométricas son redondeadas en su selección por dentro de la figura o por fuera.
- Con imágenes reales como imágenes fotográficas, el comportamiento no es aceptable debido a la cantidad de ruido (contornos no deseables a seleccionar).
- Un aspecto importante a considerar tanto para imágenes artificiales como reales es la eficiencia (tiempo) de nuestro software. No obstante, este valor también depende del tamaño de la imagen que estemos tratando.

Aspectos a mejorar del algoritmo

- Con el fin de mejorar la selección de contornos se podría realizar un estudio más exhaustivo sobre el comportamiento del algoritmo dependiendo de los valores determinados para las constantes de coste. (W_g, W_D, W_z)
- El mayor tiempo de cómputo se debe a la gestión de pixeles activos y la obtención del pixel activo de menor coste. Por tanto, esto se podría mejorar utilizando estructuras propias en lugar de estructuras facilitadas por la STL.

Aspecto a mejorar de la interfaz

- Permitir la realimentación al usuario en tiempo real de la selección (camino óptimo).
- Cuando el usuario seleccione un punto desplazar este punto al de menor coste de un entorno limitado (por ejemplo, 10 pixeles alrededor).
- Añadir la funcionalidad de recortar la zona seleccionada de la imagen.

7. Bibliografía

Idea principal (Scissors Intelligent)

Intelligent Scissors for Image Composition.
Autores: Eric N. Mortensen, William A. Barrett y Brigham Young University.

Tutorial para hacer un visor de imágenes en c++:

https://www.youtube.com/watch?v=E_CBw-9FzH8

Cálculo de zero-crossing (Laplaciana):

http://docs.opencv.org/doc/tutorials/imgproc/imgtrans/laplace_operator/laplace_operator.html

Captura de eventos del ratón:

http://msdn.microsoft.com/es-es/library/ms171542(v=vs.110).aspx