Proyecto Final. Poisson Blending.

Visión por Computador

Iván Garzón Segura
Víctor Alejandro Vargas Pérez

UNIVERSIDAD
DE GRANADA

Índice

1. Introducción

El objetivo de este proyecto es implementar una herramienta que permita recortar objetos de una imagen y pegarlos en otra imagen sin que aparentemente se note en la imagen final que los objetos han sido pegados. Para ello, existen diferentes métodos, como el uso de pirámides Laplacianas. Sin embargo, hemos optado por utilizar el método Poisson Blending, que obtiene mejores resultados. Dicho método se describe en detalle en el paper Poisson Image Editing (2003), e indicaremos a continuación lo más relevante del mismo.

Esta técnica tiene como elemento principal la ecuación diferencial de Poisson con la condición de frontera de Dirichlet, que especifica la Laplaciana de una función (imagen) desconocida en un dominio de interés, junto con sus valores para la frontera del dominio.

Resolver la ecuación de Poisson se puede ver como un problema de minimización en el que se calcula la función cuyo gradiente es el más cercano a un campo vectorial guía establecido bajo unas condiciones de contorno dadas. De esta forma, la función (imagen) reconstruida interpola las condiciones de contorno hacia dentro, mientras sigue las variaciones espaciales del campo vectorial guía de la forma más fiel posible.

2. Fundamentos Teóricos

Para presentar esta técnica, consideremos los siguientes elementos:

- S: dominio de la imagen (subconjunto cerrado de $\mathbb{R}^2$)
- Ω: subconjunto cerrado de S con frontera $\partial\Omega$ (zona de pegado en la imagen destino). Para nuestro caso concreto (imágenes) podemos definir $\partial\Omega$ como todos los píxeles de S fuera de Ω que tienen algún vecino (uno de los cuatro píxeles adyacentes) en Ω.
- f*: función escalar definida en S menos el interior de Ω, que representa los valores de la imagen destino fuera del área de pegado.
- f: función desconocida dentro Ω, es decir, los píxeles finales de dicha zona tras pegar el objeto.
- g: función escalar dentro de Ω cuyos valores corresponden a los píxeles del objeto en la imagen fuente.
- v: campo vectorial guía definido sobre Ω. La elección de este campo guía determinará el efecto de la técnica, y en el presente proyecto se contemplarán las dos opciones presentadas en el paper relacionadas con la inserción de imágenes(importing gradientes y mixing gradients).

La interpolación de f de f* sobre Ω, con la restricción añadida de un campo vectorial v, corresponde al siguiente problema:

$$\min_f \iint_\Omega |\nabla f - \mathbf{v}|^2 \text{ con } f|_{\partial\Omega} = f^*|_{\partial\Omega}$$

La solución de este problema corresponde a la siguiente ecuación diferencial de Poisson con condiciones de frontera de Dirichlet:

$$\Delta f = \text{ divv over } \Omega, \text{ con } f|_{\partial\Omega} = f^*|_{\partial\Omega}$$

Esta ecuación es fundamental, pues su solución para f contendrá los valores finales de los píxeles en la zona de pegado. Por consiguiente, la implementación del algoritmo consistirá en calcular la divergencia de v, y posteriormente obtener f como la solución de Af=b, donde:

- b es dicha divergencia (considerando las condiciones de frontera), en forma de un vector columna con tantos valores n como píxeles en Ω. Así, dado un píxel p en Ω, y cada uno de sus vecinos q, su valor en b corresponde a la suma de sus v_{pq} (depende del campo guía v elegido) y del valor en f* de sus vecinos que se encuentren en $\partial\Omega$.
- f es otro vector columna del mismo tamaño (píxeles de la solución).
- A es el operador laplaciano, esto es, una matriz dispersa nxn con 4's (en la diagonal) y -1's, tal que al multiplicarla por un vector columna, calcula otro vector columna con la laplaciana de cada valor del primero. Dado un píxel, su laplaciana es la suma de sus gradientes, o dicho de otra forma, la suma de las diferencias con sus cuatro vecinos (dado $x_{i,j}$, su laplaciana sería $4 * x_{i,j} - x_{i-1,j} - x_{i+1,j} - x_{i,j-1} - x_{i,j+1}$)

Esta implementación se verá en detalle en la sección 3.

2.1. Importing Gradients

Para pegar un objeto de una imagen en otra, la elección básica para el campo v es el gradiente de la parte correspondiente al objeto en la imagen fuente (g). De esta forma, $\mathbf{v} = \nabla g$, y por lo tanto la ecuación de Poisson pasaría a ser la siguiente:

$$\Delta f = \Delta g \text{ over } \Omega, \text{ con } f|_{\partial\Omega} = f^*|_{\partial\Omega}$$

De cara a su implementación, en este caso $v_{pq} = g_p - g_q$, donde g indica el valor del píxel correspondiente en la imagen fuente.

2.2. Mixing Gradients

Si bien la elección anterior para v funciona correctamente cuando se tratan objetos opacos, los resultados no serían satisfactorios para objetos con transparencias o agujeros, pues v no contiene información del fondo de la imagen destino. Por consiguiente, se propone usar para v variaciones tanto de g como de f*, escogiendo en cada caso aquella de mayor valor:

$$\text{para todo } \mathbf{x} \in \Omega, \mathbf{v}(\mathbf{x}) = \begin{cases} \nabla f^*(\mathbf{x}) & \text{si } |\nabla f^*(\mathbf{x})| > |\nabla g(\mathbf{x})| \\ \nabla g(\mathbf{x}) & \text{en otro caso} \end{cases}$$

Por consiguiente, en este caso el cálculo de la divergencia de v se hará teniendo en cuenta que para cada vecino q de un píxel p en Ω:

$$v_{pq} = \begin{cases} f_p^* - f_q^* & \text{si } \left|f_p^* - f_q^*\right| > |g_p - g_q| \\ g_p - g_q & \text{en otro caso} \end{cases}$$

3. Implementación

3.1. Función Principal: poissonBlending

La implementación de esta herramienta se encuentra en el archivo **poisson_blending.py**. En ella, la función principal es **poissonBlending(objeto, fuente, destino, despl)**, que se encargar de realizar el pegado de un objeto de una imagen en otra mediante la técnica poisson blending. Concretamente, devolverá dos resultados: un usando importing gradients y otro con mixing gradients. Sus parámetros son:

- objeto: posiciones en la fuente de la objeto a pegar.
- fuente: imagen en color (3 canales) que contiene el objeto que se va a pegar en el destino.
- destino: imagen en color (3 canales) en la que se va a pegar el objeto.
- despl: pareja de enteros que indica el desplazamiento que hay que realizar de las posiciones del objeto, para transformar sus coordenadas en las deseadas para la imagen destino.

Aunque no es necesario que fuente y destino coincidan en dimensiones, sí que es preciso que el objeto quepa dentro de la imagen destino, y que el desplazamiento indicado no permita que el objeto se salga por algún extremo de la imagen destino.

El objeto de este algoritmo es resolver la ecuación Af=b (comentada en la sección anterior) para obtener f como los valores finales del objeto al pegarlo en la imagen destino. Para ello, en primer lugar se obtiene la matriz de Poisson A acorde a los píxeles del objeto. A continuación, de forma independiente en cada

canal de la imagen, se obtiene el vector columna b (divergencia de v) tanto para importing_gradients como mixing_gradientes, y se resuelve así la ecuación Af=b con la ayuda de la biblioteca scipy, que implementa el método del gradiente conjugado (método iterativo) en linalg.cg(A,b). El motivo por el que se usa este método es que Af=b (ecuación en derivadas parciales) es un sistema disperso demasiado grande para ser tratado por un método directo.

Finalmente, cada solución es igual a la imagen destino con los píxeles correspondientes a omega (posiciones del objeto + desplazamiento) modificados al valor obtenido en f.Como se tratan de imágenes, estos valores han de ser truncados entre 0 y 255 (los negativos pasan a 0, y los valores mayores que 255 a este número), y serán transformados en enteros cuando se realice la visualización.

El código de esta función es el siguiente:

```
def poissonBlending(objeto, fuente, destino, despl):

    # Cada solución (import_gradients y mixing_gradients)
    # se inicializa con los valores de la imagen destino
    solucion_import = np.copy(destino)
    solucion_mix = np.copy(destino)
    # 1. Calcular matriz de coeficientes para omega
    A = matrizPoisson(objeto)
    # Para cada canal
    for canal in range(destino.shape[2]):
        # 2.1 Calcular vector columna b para importing y mixing
        b_import, b_mix = calcularDivGuia(
            objeto, fuente[:, :, canal], destino[:, :, canal], despl)

        # 2.2 Resolver A*f_omega=b
        f_omega_import, _ = linalg.cg(A, b_import)
        f_omega_mix, _ = linalg.cg(A, b_mix)

        # 2.3 Incorporar f_omega a f
        for i, posicion in enumerate(objeto):
            solucion_import[posicion[0]+despl[0], posicion[1]+despl[1],
                            canal] = f_omega_import[i]
            solucion_mix[posicion[0]+despl[0], posicion[1] +
                         despl[1], canal] = f_omega_mix[i]

    # Devolver todas las soluciones, con valores entre 0 y 255
    return np.clip(solucion_import, 0, 255), np.clip(solucion_mix, 0, 255)
```

Como se puede ver, se hace uso de dos funciones auxiliares: **matrizPoisson** para obtener la matriz A, y **calcularDivGuia** para obtener el vector b.

3.2. Cálculo de la matriz de Poisson A

Tal y como se introdujo en la sección 2, la matriz de A es una matriz cuadrada dispersa con tantas filas como píxeles en omega. De esta forma, su función es calcular la Laplaciana de cada uno de los elementos de un vector columna con el mismo número de elementos que filas tiene A.

Su cáculo se hace a partir de las posiciones del objeto a pegar, asignando para elemento (fila) un 4 en la diagonal y un -1 en la columna correspondiente a cada uno de sus vecinos dentro de omega (como máximo serán 4). La implementación sigue lo descrito:

```
def matrizPoisson(omega):
    # Calculamos el número de píxeles de omega
    N = len(omega)
    # Matriz dispersa de NxN
    A = sparse.lil_matrix((N, N))

    # Para cada píxel/posición
    for i, posicion in enumerate(omega):
        progress_bar(i, N-1)  # OJO
        A[i, i] = 4 # 4 en la diagonal (píxel actual)

        # -1 en las columnas de los vecinos en omega
        for vecino in getVecindario(posicion):
            if vecino in omega:
                j = omega.index(vecino)
                A[i, j] = -1

    return A
```

La función getVecindario(posicion) es sencilla: devuelve los índices correspondientes a los 4 vecinos de una posición dada: (i+1,j), (i-1,j), (i,j+1), (i,j-1)

3.3. Cálculo del vector b: divergencia del campo guía v con criterios de contorno

El vector b consta de tantos elementos como píxeles hay en omega, calculando para píxel la suma de los valores en la imagen destino de aquellos vecinos que estén fuera de omega (frontera), junto a la divergencia del campo guía v en dicha posición. Esta divergencia depende del método usado, devolviéndose en esta función el resultado tanto para Importing Gradients como para Mixing Gradients, para así poder comparar ambos fácilmente. La función se define como sigue:

```
def calcularDivGuia(omega, fuente, destino, despl):
    N = len(omega)
    b_import = np.zeros(N)
    b_mix = np.zeros(N)

    # Para cada píxel en omega
    for i in range(N):
        # Condición contorno (común a ambos métodos)
        valorBorde = calcularValorBorde(omega, omega[i], destino, despl)
        # Diverguencia de v + valorBorde
        b_import[i] = getLaplaciana(fuente, omega[i]) + valorBorde
        b_mix[i] = getLaplacianaMix(
            fuente, destino, omega[i], despl) + valorBorde

    return b_import, b_mix
```

Como se dijo previamente, el valor del borde se obtiene sumando el valor en la imagen destino de los vecinos fuera de omega. Por lo tanto, este valor será 0 para todos los píxeles en el interior de la máscara (que no se encuentra en colindan con la frontera).

```
def calcularValorBorde(omega, posicion, destino, despl):
    valorBorde = 0
    if (esBorde(omega, posicion)):
        for vecino in getVecindario(posicion):
            if not vecino in omega:
                if dentroImagen(destino, vecino + despl):
                    valorBorde += destino[tuple(vecino + despl)]
                else:
                    valorBorde += destino[tuple(posicion + despl)]

    return valorBorde
```

La función esBorde simplemente comprueba si el píxel tiene algún vecino fuera de omega (pues, de ser así, valoBorde será 0). Por su parte, dentroImagen comprueba si una posición está dentro de la imagen, esto es: sus valores son mayores o igual que 0 y menores que sus dimensiones.

Un caso peculiar a tener en cuenta es el que se da cuando alguna posición de omega se encuentra en el borde de la imagen destino, y por lo tanto tiene un vecino (o dos) que no está ni en omega ni en los valores definidos fuera de este. Este caso no es baladí, pues de no tratarse, se estaría asumiendo que la condición de frontera en ese píxel es 0, lo que ensombrecería a partir de esa zona. Por este motivo, hemos decidido seguir una política de bordes de copia del último valor: el valor de un píxel exterior a la imagen es igual al valor en el píxel límite de la misma. De esta forma, si un vecino de un píxel p en omega se sale de la imagen, tomará el valor de la imagen destino en p.

Respecto al cálculo de la diverguencia de v, en el caso de Importing Gradients se realiza sumando las diferencias de los valores (en la imagen fuente) con sus vecinos que se encuentra en omega (es decir, calculando la laplaciana).

```
def getLaplaciana(fuente, posicion):
    laplaciana = 0
    for vecino in getVecindario(posicion):
        if dentroImagen(fuente, vecino):
            laplaciana += fuente[posicion] - fuente[vecino]

    return laplaciana
```

En el caso de Mixing Gradients, se consideran también los gradientes en la imagen destino, comparando ambos y eligiendo en cada caso el de mayor intensidad (valor absoluto). Casos especiales a tener en cuenta se dan cuando el vecino de un elemento de omega se sale de la imagen fuente o destino. En esos casos, como usamos política de bordes de copia, el gradiente correspondiente es 0 (la resta es entre valores iguales), por lo que se utiliza directamente el gradiente en la otra imagen (si esta sí que contiene al vecino). La implementación se encuentra en la siguiente página.

```
def getLaplacianaMix(fuente, destino, posicion, despl):
    laplaciana = 0
    for vecino in getVecindario(posicion):
        # Caso general: calcular y comparar gradientes en las dos imágenes
        if dentroImagen(destino, vecino+despl) and
                dentroImagen(fuente, vecino):
            grad_fuente = fuente[posicion] - fuente[vecino]
            grad_destino = destino[tuple(
                posicion+despl)] - destino[tuple(vecino+despl)]
            if abs(grad_fuente) > abs(grad_destino):
                laplaciana += grad_fuente
            else:
                laplaciana += grad_destino
        # Si una de las imágenes no contiene al correspondiente vecino:
        # -> política de bordes de copia: gradiente intensidad 0.
        #   -> Se usa directamente el gradiente de la otra
        elif dentroImagen(destino, vecino+despl):
            grad_destino = destino[tuple(
                posicion+despl)] - destino[tuple(vecino+despl)]
            laplaciana += grad_destino
        elif dentroImagen(fuente, vecino):
            grad_fuente = fuente[posicion] - fuente[vecino]
            laplaciana += grad_fuente

    return laplaciana
```

3.4. Obtención de las Posiciones del Objeto y pegar y el Desplazamiento

Hasta ahora, se ha descrito el proceso por el que, a partir de una imagen fuente, posiciones de un objeto en la fuente, una imagen destino y un desplazamiento, se obtiene una imagen final con el objeto pegado en el destino mediante Poisson Blending.

Sin embargo, es preciso detallar el procesamiento previo a esta función, necesario para obtener las posiciones del objeto y el desplazamiento en el destino.

3.4.1. Posiciones del Objeto

Para obtener estas posiciones, se requiere el uso de una imagen en blanco y negro (máscara) con las mismas dimensiones que la imagen fuente, de forma que el color blanco representa al objeto seleccionado. Estas máscaras deben crearse de forma externa, y en nuestro caso las hemos creado con el programa de edición de imágenes GIMP, que es de software libre.

Por lo tanto, una vez se dispone de dicha imagen máscara, se puede pasar su ruta como parámetro a la función **getObjeto(nombre_mascara)**, que devuelve una lista con las posiciones correspondientes al objeto. Para ello, se carga la imagen en blanco y negro con valores enteros, y, para asegurar que no hay outliers en la representación, se utiliza la función de openCV cv2.threshold(mascara, 0, 255, cv2.THRESH_OTSU), que con estos parámetros cambia a 255 el valor de los píxeles próximos a 255, y a 0 el resto. Es decir, binariza la máscara.

Hecho esto, se utiliza la función getPosicionesObjeto(mascara) para crear la lista de posiciones, recorriendo la máscara y añadiendo las posiciones de los píxeles con valor no cero (blanco).

3.4.2. Desplazamiento en el Destino

La posición del objeto en la imagen fuente es independiente de la que va a tener en el destino. Por lo tanto, debe indicarse cuál es el lugar del destino en el ha de pegarse en el objeto. Para ello, se declara una lista con la posición central en la que debe ubicarse el objeto, que puede ser bien en forma de coordenadas concretas en el destino, o, si se utilizan valores reales, la posición relativa respecto a las dimensiones (un porcentaje). Por ejemplo, si tenemos una imagen de 200x200, el punto central 100x100 se podría indicar como [100, 100] o como [0.5, 0.5], e incluso podría indicarse cada eje de forma distinta (uno de forma absoluta y el otro de forma relativa).

Definidas las coordenadas destino, se hace uso de una función **calcularDesplazamiento(pos_dest, objeto, destino)** que obtiene el desplazamiento a aplicar a los píxeles del objeto (en la imagen fuente) para obtener las posiciones correspondientes en la imagen destino, de forma que el centro del objeto se ubique finalmente en el la posición dada por pos_dest.

Cabe destacar que, con el fin de evitar desplazamientos no deseados (objeto queda parcial o totalmente fuera de la imagen) esta función también devuelve un valor booleano que indica si el desplazamiento resultante de la posición dada es válido. Para ello, se comprueba que al aplicar dicho desplazamiento al objeto, este queda completamente contenido en la imagen destino, comprobando para ello el resultado de aplicar el desplazamiento a los extremos del objeto (max_x, min_x, max_y, min_y).

```
def calcularDesplazamiento(pos_dest, objeto, destino):

    # Si se pasan reales en pos_dest, se interpreta como un porcentaje
    # respecto al tamaño de la imagen destino. En caso contrario,
    # son coordenadas directas.
    if not type(pos_dest[0]) is int:
        pos_dest[0] = int(destino.shape[0]*pos_dest[0])
    if not type(pos_dest[1]) is int:
        pos_dest[1] = int(destino.shape[1]*pos_dest[1])
    pos_dest = np.array(pos_dest, dtype=np.uint32)

    # Obtener los extremos de la máscara y calcular su centro
    posiciones = np.array(objeto)
    max_x = np.max(posiciones[:, 1])
    min_x = np.min(posiciones[:, 1])
    max_y = np.max(posiciones[:, 0])
    min_y = np.min(posiciones[:, 0])

    x_centro = int((max_x + min_x) / 2)
    y_centro = int((max_y + min_y) / 2)

    # Obtener el desplazamiento como la diferencia del objetivo con el
    # centro de la máscara
    despl = pos_dest - (y_centro, x_centro)

    # Comprobar si es un desplazamiento válido
    despl_valido = False
    if dentroImagen(destino, [max_y, max_x] + despl) and
            dentroImagen(destino, [min_y, min_x] + despl):
        despl_valido = True

    return despl, despl_valido
```

Tal y como se puede observar, tras interpretar la posición destino dada apropiadamente, en esta función se calcula el centro de la máscara obteniendo la media de sus extremos (max_y-min_y/2, max_x+min_x/2). Posteriormente, se calcula el desplazamiento como la diferencia entre la posición destino y este centro, y se comprueba si es válida.

Un detalle a señalar a la hora de mostrar los resultados del pegado de imágenes es que, aunque los va-

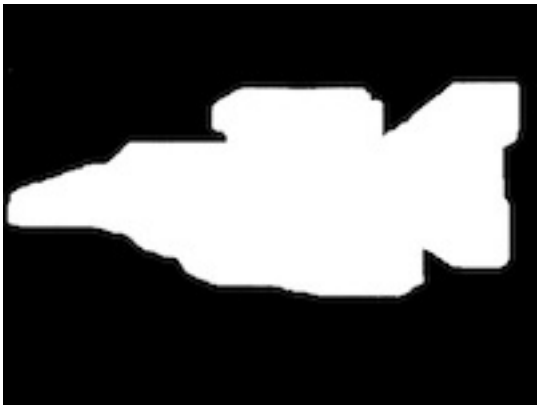
lores deben convertirse a entero, no se deben normalizar, pues eso podría alterar el valor de los píxeles fuera del área de pegado (así como los de dentro), lo cual no es deseable, pues alteraría el resultado. Por este motivo, las funciones auxiliares mostrarImagen y mostrarVariasImagenes (implementadas en las prácticas iniciales) tiene el parámetro de normalización desactivado por defecto.

4. Ejecución y Resultados

Con el fin de comprobar el funcionamiento de esta técnica, es el momento de utilizarla con diferentes imágenes ejemplo. Para ello, hemos buscado diferentes imágenes (unas como fuente y otras como destino), y hemos creado las máscara de las imágenes fuente. El objetivo es aplicar la técnica de Poisson Blending (con Importin Gradients y Mixing Gradients) y el solapado directo, para poder ver una comparativa. La posición concreta en la que se va pegar cada objeto la hemos obtenido experimentalmente, tratando de colocarlos en lugares que tengan sentido (como una taza de café encima de una mesa).

En primer lugar veamos un ejemplo que está también presente en el paper: un avión como imagen fuente y un paisaje en una montaña como destino.

(a) Imagen fuente



(b) Máscara

(a) Imagen destino

(b) Pegado directo

(c) Poisson Blending Importing Gradients

(d) Poisson Blending Mixing Gradients

Como se puede ver, pese a que la máscara no delimita al avión de forma precisa (hay cierto borde alrededor), la técnica Poisson Blending logra realizar un pegado de calidad. Si nos fijamos bien en el resultado con Importint Gradients, se puede ver un poco de ese borde (bruma blanca), mientras que en el método Mixing Gradients ha sido completamente removido este borde. Por contra, en la parte trasera inferior del avión podemos ver que ha aparecido la textura de la montaña de fondo, algo que no ocurre en Importing Gradients, pues en su interior solo se computan los gradientes del propio avión. En cualquier caso, los resultados son favorables, y de hecho coinciden con los mostrados en el paper de referencia, lo que es buen indicativo de que la implementación es correcta.

Veamos ahora un caso más difícil que también esta en el paper citado:

(a) Imagen fuente

(b) Máscara

(c) Imagen destino

(d) Pegado directo

(e) Poisson Blending Importing Gradients

(f) Poisson Blending Mixing Gradients

En esta ocasión, el método Mixing Gradients sí ha podido demostrar su ventaja de forma clara: mientras que con importing gradients se obtiene un fondo liso (el del papel) pero anaranjado (muro), con mixing, al considerar los gradientes de la imagen destino en el interior de omega, consigue pegar objetos con agujeros conservando el fondo de la imagen destino en estos. De esta forma, imágenes con escrituras pueden ser pegadas en diversas superficies con resultados realistas.

A continuación mostramos un ejemplo similar, esta vez pegando un grafiti en una pared.

(a) Imagen fuente

(b) Máscara

(c) Imagen destino

(d) Pegado directo

(e) Poisson Blending Importing Gradients

(f) Poisson Blending Mixing Gradients

5. Referencias