Image processing

Assignment 4

Pouria malek khayat

Exercise 1

In this part we implement edge detection and hole filling operations on grayscale and binary images. This code utilizes various image processing techniques and libraries, such as OpenCV and Matplotlib, to process and display the results.

Functions:

The code defines the following image processing functions:

a) **edge_detection**(image): This function applies the Sobel operator to detect edges in the input image. The Sobel operator calculates the gradient of the image intensity at each pixel, highlighting regions of rapid intensity change. The resulting edges are converted to a grayscale image using appropriate data type conversions.

b) **hole_filling**(image): This function performs hole filling on the input image using a morphological operation called dilation. Dilation expands the boundaries of objects in the image, effectively filling small holes or gaps in the object regions.

Image Loading:

Then we load four images from the specified file paths: grayscale1.jpeg, grayscale2.png, binary1.png, and binary2.png. These images serve as inputs for the image processing operations.

Edge Detection:

We apply the edge_detection function to the grayscale images grayscale_image1 and grayscale_image2, as well as the binary images binary_image1 and binary_image2. The results are stored in the variables grayscale_edges1, grayscale_edges2, binary_edges1, and binary_edges2, respectively.

Hole Filling:

Then we perform hole filling on the grayscale images using the hole_filling function. The filled images are stored in the variables grayscale_filled1 and grayscale_filled2. Similarly, the binary images are processed for hole filling, and the filled images are stored in the variables binary_filled1 and binary_filled2.

Results Visualization:

The code utilizes Matplotlib to display the original images, their corresponding edges, and the filled images in separate subplots. The images and their respective titles are displayed using the imshow and title functions.

Exercise 2

This code that I wrote allows the synthesis of textures by combining smaller patches from input textures. The process involves selecting and arranging patches to create a seamless and visually appealing texture synthesis result. The code consists of several functions, each serving a specific purpose in the texture synthesis process. Functions in detail:

1. **calculate_patch_error**(patch, length, overlap, result, y, x):

  - Description: Calculates the error between a patch and the overlapping region of the result texture.

  - Parameters:

    - patch: The current patch being considered.

    - length: The length (side length) of the patch.

    - overlap: The overlap between patches.

    - result: The synthesized texture result.

    - y: The y-coordinate of the current patch's position.

    - x: The x-coordinate of the current patch's position.

- Returns: The calculated error between the patch and the overlapping region.

2. **find_best_patch**(texture, length, overlap, result, y, x):

  - Description: Finds the best patch from the given texture based on the error metric.

  - Parameters:

    - texture: The input texture from which patches are selected.

    - length: The length (side length) of the patch.

    - overlap: The overlap between patches.

    - result: The synthesized texture result.

    - y: The y-coordinate of the current patch's position.

    - x: The x-coordinate of the current patch's position.

  - Returns: The best patch selected from the texture.

3. **find_min_cut_path**(errors):

  - Description: Finds the minimum error cut path in a 2D error map.

  - Parameters:

    - errors: The 2D error map representing the errors between patches and overlapping regions.

  - Returns: The minimum error cut path as a list of indices.

4. **apply_min_cut_patch**(patch, overlap, result, y, x):

  - Description: Applies the minimum error cut technique to blend a patch with the existing result texture.

  - Parameters:

    - patch: The current patch being considered.

- overlap: The overlap between patches.
    - result: The synthesized texture result.
    - y: The y-coordinate of the current patch's position.
    - x: The x-coordinate of the current patch's position.
  - Returns: The patch after applying the minimum error cut.

5. **synthesize_texture**(texture, patch_length, num_patches, mode="cut"):
  - Description: Performs the texture synthesis process by iteratively selecting and blending patches.
  - Parameters:
    - texture: The input texture used for synthesis.
    - patch_length: The length (side length) of the patches.
    - num_patches: A tuple specifying the number of patches in the synthesized texture (height, width).
    - mode: The blending mode to use ("cut" by default).
  - Returns: The synthesized texture as a NumPy array.

6. **plot_synthesized_texture**(texture, patch_length):
  - Description: Plots the original texture and the synthesized texture side by side.
  - Parameters:
    - texture: The original texture.
    - patch_length: The length (side length) of the patches used for synthesis.
  - Output: Displays a plot with the original texture and the synthesized texture.

We start by loading input textures, and then the `plot_synthesized_texture` function is called for each input texture, generating the synthesized texture using the synthesize_texture function with the "cut" blending mode.

Overall, this code provides a basic implementation of texture synthesis using patch-based methods. It demonstrates how smaller patches can be combined to create visually coherent textures.

Notebook link

https://colab.research.google.com/drive/1UOfZ4gyYMDVMlcivA0Z2f-X-UVtLohqZ#scrollTo=E6hvzx-gBnup