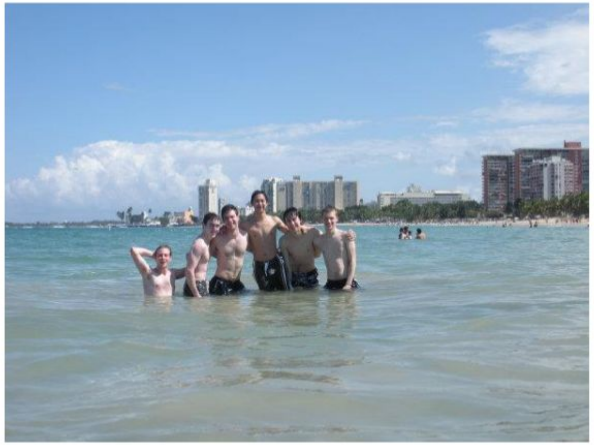
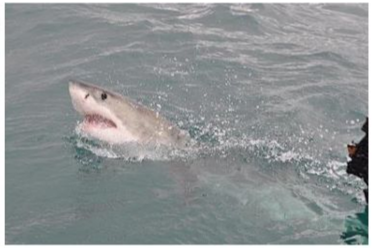
C2 - Optimisation in Computer Vision
Part 2 - Poisson Editing
Team no. 4

Members:
Miruna-Diana Jarda
Gunjan Paul
Diana Tat

Challenge



+



=

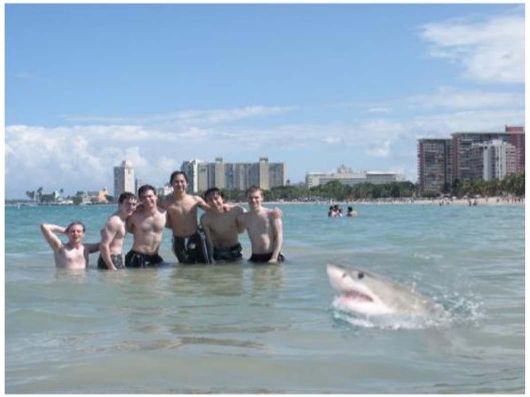


Fig 1. Image A

Fig 2. Image B

Fig 3. Image H(new)

How to achieve it?

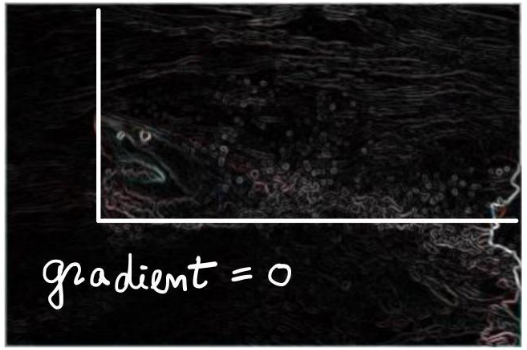


Figure 4. Gradient explained

In order to achieve that result we need to set three criterias.

1) Image A remains the same.
2) We want to smooth the image B, which means that the gradient of image B should be zero.
3) We want to keep the details of image B.

We calculated the gradient of B and we noticed that most of the time, the gradient is zero and when it is not zero, we can see the details of the image. So, to make it easier, we can combine the last two criterias into a single one that says that we should only use the gradient of image B.

When the gradient is zero, we keep it and obtain a smooth image.When the gradient is not zero, we use it and get the details which are the splashes and the edges of the shark.

Review the mathematical concepts

Laplacian: The Laplacian is a second-order differential operator that measures the difference between the value of a function at a point and the average of its neighbors. In the context of images, it gives a measure of how much a pixel stands out from its neighbors.

The Laplacian of an image I is defined as the divergence of its gradient:

$$\Delta I = \nabla . \nabla I$$

So, Laplacian can be written as

$$\Delta I = [\frac{\partial^2 I}{\partial x^2}, \frac{\partial^2 I}{\partial y^2}]$$

so, we can write the change of this first derivative as

$$\frac{\partial^2 I}{\partial x^2} \approx (I(x+1, y) - I(x, y)) - (I(x, y) - I(x-1, y))$$

$$\frac{\partial^2 I}{\partial y^2} \approx (I(x, y+1) - I(x, y)) - (I(x, y) - I(x, y-1))$$

Simplifying this:

$$\Delta I(x, y) = I(x+1, y) + I(x-1, y) + I(x, y+1) + I(x, y-1) - 4I(x, y)$$

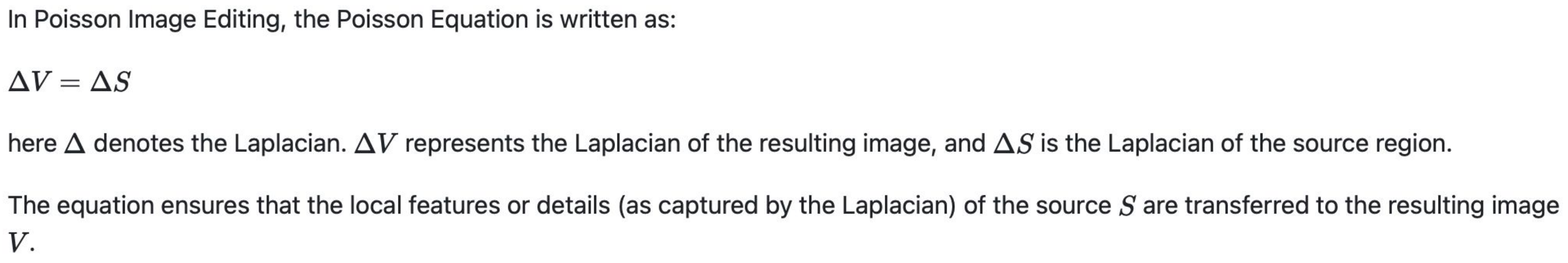
Role in image processing

The main idea behind Poisson Image Editing is to paste the details (or the internal structure) of a source image region onto a target image in a way that is seamless and consistent with the boundary of the target region.

In essence, instead of copying the pixel values directly from the source to the target, we aim to transfer the features or details represented by the Laplacian.

The Laplacian of an image gives a measure of the local variation at each pixel with respect to its neighbors. In other words, it captures the details or features of the image.

The Laplacian ensures that we do so while allowing the overall intensity to adjust and blend with the surrounding target region.



In Poisson Image Editing, the Poisson Equation is written as:

$$\Delta V = \Delta S$$

here Δ denotes the Laplacian. ΔV represents the Laplacian of the resulting image, and ΔS is the Laplacian of the source region.

The equation ensures that the local features or details (as captured by the Laplacian) of the source S are transferred to the resulting image V.

Approach

For solving this problem we set up two criterias:

1) **We want the transition between A and B to be smooth.**

   H(x,y)=A(x,y), at each (x,y) of the boundary of B

2) **We want to keep the details of image B, i.e. the gradients at each point.**

   $\nabla H(x,y) = \nabla B(x,y)$ *at each (x,y) inside B*

To have a more realistic result, we will use the background of image A, blend together with image B, according to the equation:

$$4H(x,y) - \sum H(x+dx, y+dy) = 4B(x,y) - \sum B(x+dx, y+dy)$$

Approach - details

The point of interest on the 2D grid is denoted by (x, y), while "N" represents the number of valid neighbors that the pixel actually has within the selection region, including the boundary, which is equal to 4. Then the possible neighbor locations, ranging over {(-1, 0), (1, 0), (0, -1), (0, 1)}, are denoted by (dx, dy). This accounts for the 4 possible neighbors of each point.

The left-hand side of the equation calculates the spatial gradient of the unknown point H(x, y) by summing the difference between H(x, y) and all of its N neighbours. Each difference that contributes to the gradient has the form H(x, y) - other(x', y'), where (x', y') represents the position of a neighbor.

The first sum on the left-hand side represents the difference of H(x, y) with other H(x', y') points that are located on the interior of the selection.

The second term represents the difference of H(x, y) with border points, which are fixed at the value of the image that we're pasting onto, A. This is why we have to treat them separately, as they do not vary and we do not solve for them.

Approach - matrix

The next step is to set up the matrix equation of the form: **Ax = b.**

Then we need to:

- Initialize x to all zeros (this is an all black image)
- Compute the product Ax
- Compute the difference (b - Ax), which measures the error between what the current guess of x (our H image) is and what we need it to be.
- Add the difference (b - Ax) back to x. This is the "gradient descent" part where we try to get our guess of the solution (x) to move in the right direction
- In the end we will end up with a system of matrices like this:

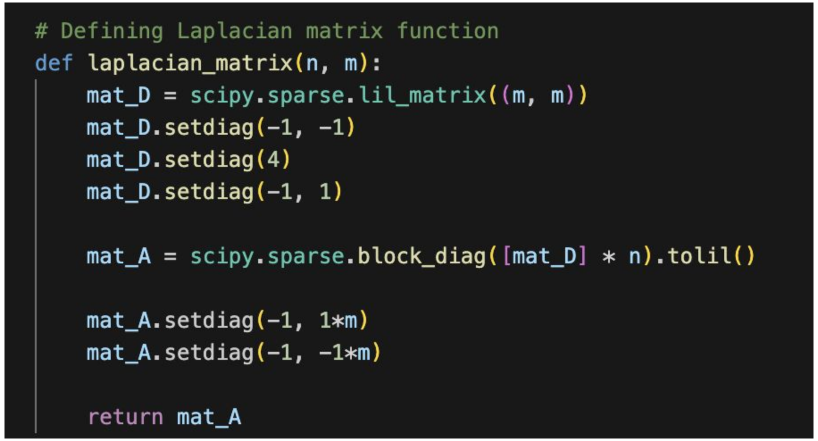
$$\underset{\substack{\uparrow \\ \text{Size = ((m+2)*(n+2) , (m+2)*(n+2))}}}{\begin{pmatrix} 2 & -1 & 0 & \dots & 0 & -1 & 0 & & & & \dots & & & & 0 \\ & & & & & & & & \dots & & & & & & \\ 0 & & & \dots & & & & 0 & 1 & 0 & & \dots & & & 0 \\ & & & & & & & & \dots & & & & & & \\ 0 & \dots & 0 & -1 & 0 & \dots & 0 & -1 & 4 & -1 & 0 & \dots & 0 & -1 & 0 \; \dots \; 0 \\ & & & & & & & & \dots & & & & & & \\ 0 & & & & \dots & & & & & 0 & -1 & 0 & \dots & 0 \; -1 & 2 \end{pmatrix}} * \underset{\substack{\uparrow \\ \text{Size = ((m+2)*(n+2) , 1)}}}{\begin{pmatrix} v_1 \\ \dots \\ v_{i,j} \\ \dots \\ \dots \\ \dots \\ v_{(m+2)*(n+2),(m+2)*(n+2)} \end{pmatrix}} = \underset{\substack{\uparrow \\ \text{Size = ((m+2)*(n+2) , 1)}}}{\begin{pmatrix} 0 \\ \dots \\ u_{i,j} \\ \dots \\ 0 \\ \dots \\ 0 \end{pmatrix}} \begin{array}{l} \leftarrow \text{North boundary} \\ \\ \leftarrow \text{Region A} \\ \\ \leftarrow \text{Region B} \\ \\ \leftarrow \text{South boundary} \end{array}$$

Implementation - part 1

The function laplacian_matrix(n, m) is utilized to construct a Laplacian matrix for a grid with n rows and m columns.

The main diagonal and the two diagonals adjacent to it in mat_D are set as follows:
  - The main diagonal is assigned a value of 4, representing the number of connections each node has with itself.
  - The two diagonals adjacent to the main diagonal are assigned a value of -1, indicating that each node is connected to its two immediate neighbors in the grid.

A block diagonal matrix mat_A is then created, consisting of n blocks, with each block being a copy of mat_D. This matrix mat_A represents a grid with n rows and m columns.
The two diagonals that connect the blocks of mat_A are set:
  - The diagonal positioned 1*m positions above the main diagonal of each block is assigned a value of -1, connecting each node to the one above it.
  - The diagonal positioned -1*m positions below the main diagonal of each block is assigned a value of -1, connecting each node to the one below it.



```python
# Defining Laplacian matrix function
def laplacian_matrix(n, m):
    mat_D = scipy.sparse.lil_matrix((m, m))
    mat_D.setdiag(-1, -1)
    mat_D.setdiag(4)
    mat_D.setdiag(-1, 1)

    mat_A = scipy.sparse.block_diag([mat_D] * n).tolil()

    mat_A.setdiag(-1, 1*m)
    mat_A.setdiag(-1, -1*m)

    return mat_A
```

Figure 5.1. Code explained

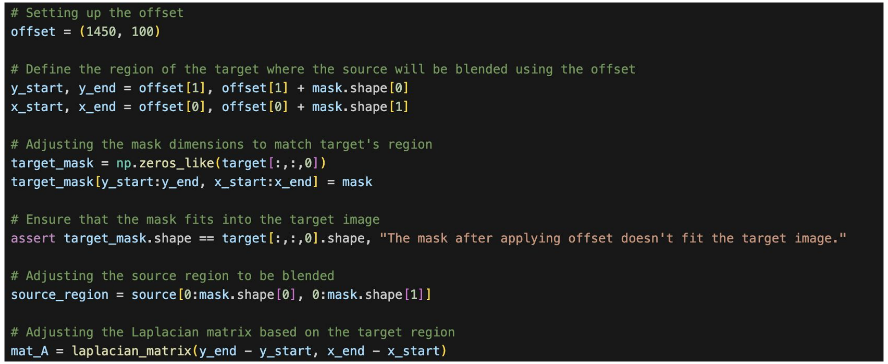
Implementation - part 2

We set an offset that specifies the location for blending a source image into a target image.

The lines "y_start, y_end = offset[1], offset[1] + mask.shape[0]" and "x_start, x_end = offset[0], offset[0] + mask.shape[1]" calculate the start and end coordinates for the region in the target image where the source image will be blended. The dimensions of the mask are used to determine the size of the region.

Then we use "source_region = source[0:mask.shape[0], 0:mask.shape[1]]" extracts a region from the source image that corresponds to the shape of the mask. This selects the portion of the source image that will be blended into the target image.

Finally, the line "mat_A = laplacian_matrix(y_end - y_start, x_end - x_start)" creates a Laplacian matrix with dimensions that correspond to the size of the region where blending will occur.



```python
# Setting up the offset
offset = (1450, 100)

# Define the region of the target where the source will be blended using the offset
y_start, y_end = offset[1], offset[1] + mask.shape[0]
x_start, x_end = offset[0], offset[0] + mask.shape[1]

# Adjusting the mask dimensions to match target's region
target_mask = np.zeros_like(target[:,:,0])
target_mask[y_start:y_end, x_start:x_end] = mask

# Ensure that the mask fits into the target image
assert target_mask.shape == target[:,:,0].shape, "The mask after applying offset doesn't fit the target image."

# Adjusting the source region to be blended
source_region = source[0:mask.shape[0], 0:mask.shape[1]]

# Adjusting the Laplacian matrix based on the target region
mat_A = laplacian_matrix(y_end - y_start, x_end - x_start)
```

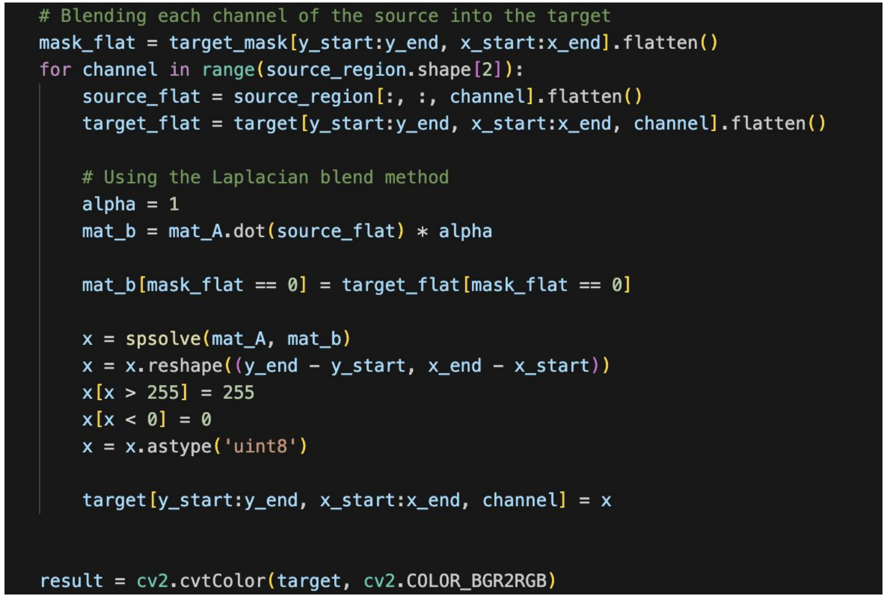
Figure 5.2. Code explained

Implementation - part 3

In this part we want to achieve the blending of each color channel of a source image into a target image by utilizing the Laplacian blend method.
Initially, we flatten a binary mask that represents the specific region in the target image where the blending will occur. Subsequently, we set the code to enter a loop to individually process each color channel of the source image.

For each channel, the code flattens the corresponding regions in both the source and target images. The Laplacian blend equation is then solved to obtain the new values for the source region. Pixels in the source that should not be blended, as indicated by the mask, remain unchanged.

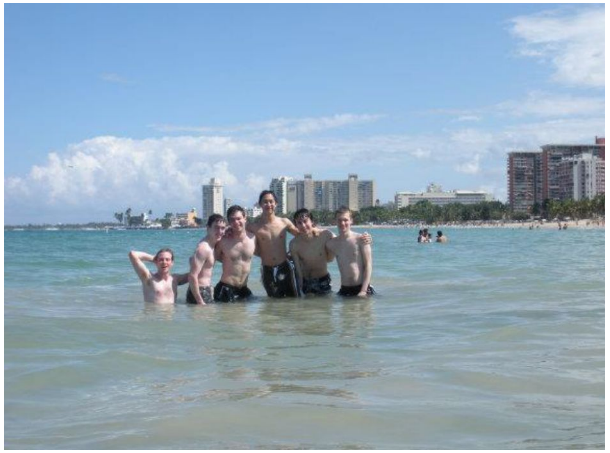
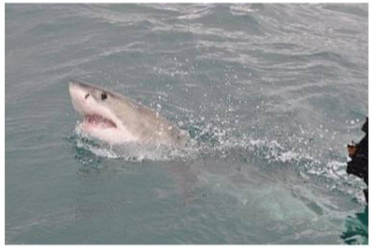
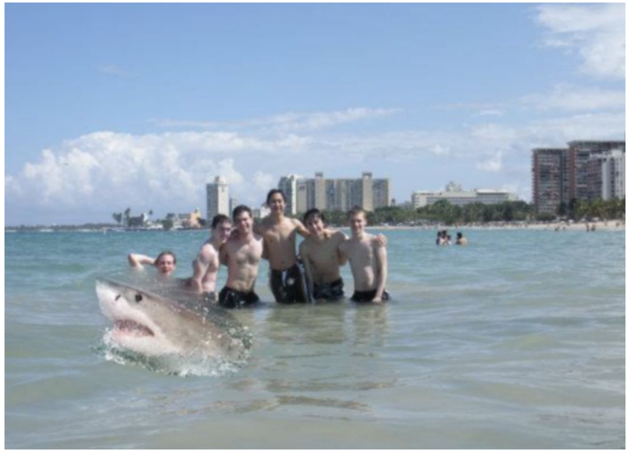
The resulting values are clipped to ensure they remain within the valid range for image data, typically ranging from 0 to 255, and are cast to 8-bit unsigned integers. Following the channel-wise blending, the target image is updated with the blended outcomes. Finally, the code converts the target image from the BGR color space to RGB, resulting in the production of the final blended image. This process enables the seamless integration of the source image into the specified region of the target.



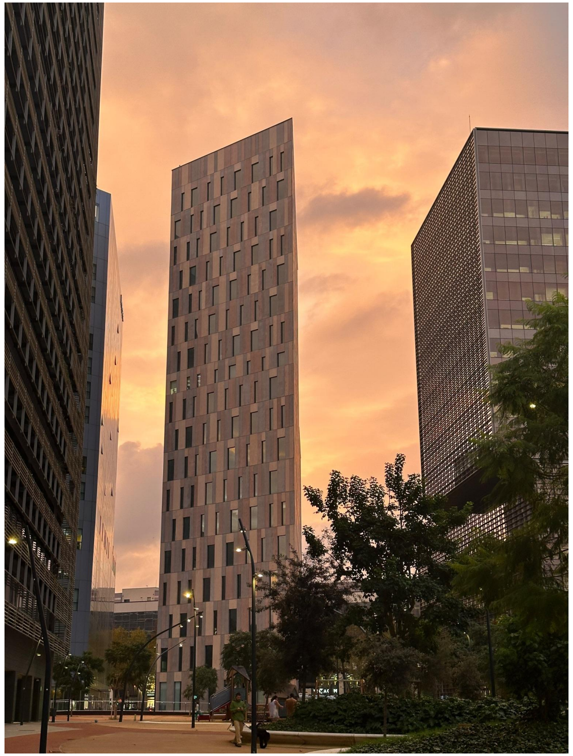
```python
# Blending each channel of the source into the target
mask_flat = target_mask[y_start:y_end, x_start:x_end].flatten()
for channel in range(source_region.shape[2]):
    source_flat = source_region[:, :, channel].flatten()
    target_flat = target[y_start:y_end, x_start:x_end, channel].flatten()

    # Using the Laplacian blend method
    alpha = 1
    mat_b = mat_A.dot(source_flat) * alpha

    mat_b[mask_flat == 0] = target_flat[mask_flat == 0]

    x = spsolve(mat_A, mat_b)
    x = x.reshape((y_end - y_start, x_end - x_start))
    x[x > 255] = 255
    x[x < 0] = 0
    x = x.astype('uint8')

    target[y_start:y_end, x_start:x_end, channel] = x


result = cv2.cvtColor(target, cv2.COLOR_BGR2RGB)
```

Figure 5.3. Code explained

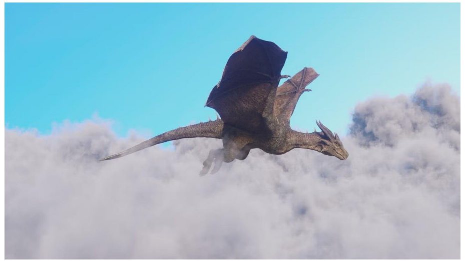
Implementation - results


+

=


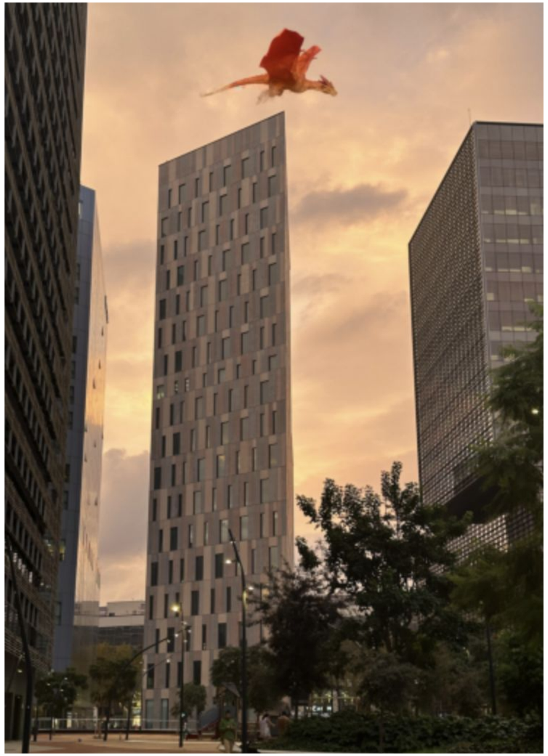
Implementation - results



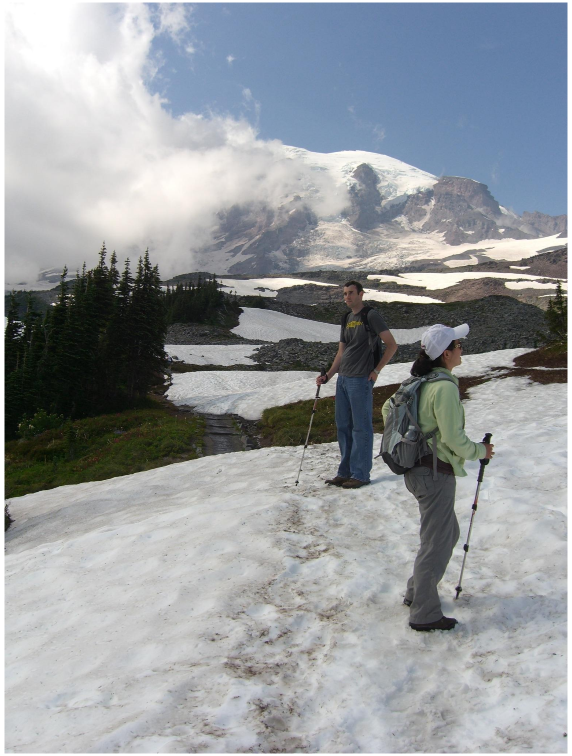
+



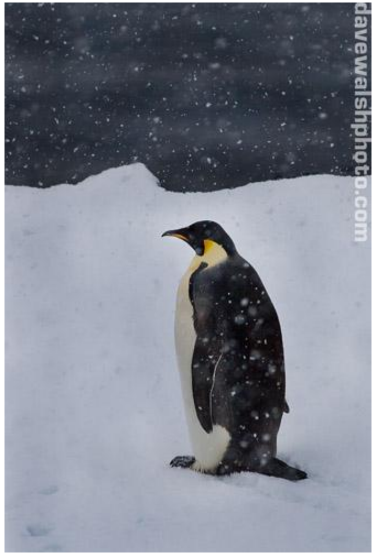
=



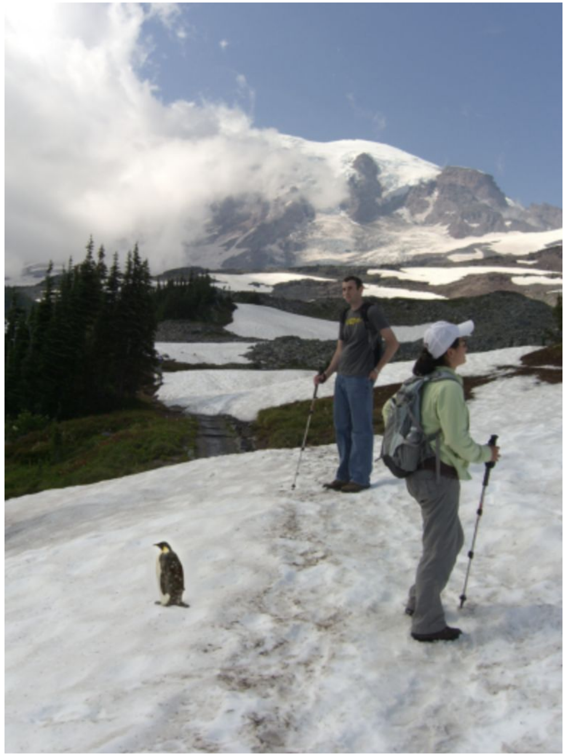
Implementation - results



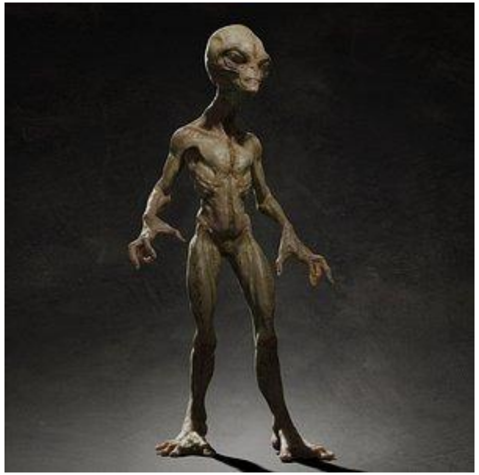
+



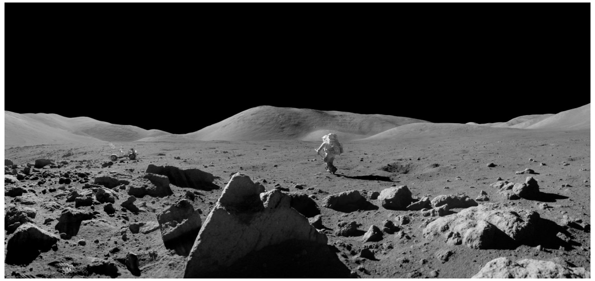
=



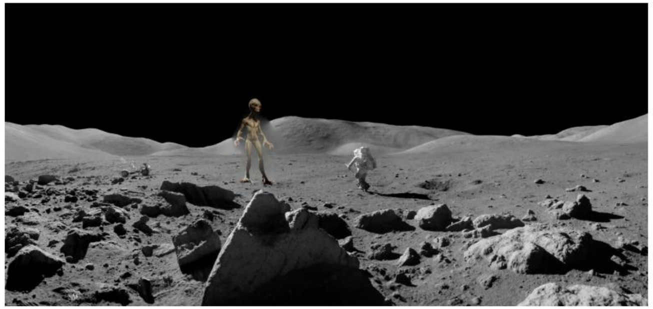
Implementation - results



+



=



Article - Poisson Image Editing
Overview

The paper starts by considering image A and image B as continuous functions in a two-dimensional space. A partial differential equation is then formulated and solved to describe the implantation of the gradient of image B onto image A in a least squares sense. This approach yields the "Poisson Equation" in continuous space, which is the basis for the technique known as "Poisson Image Editing".

The main idea of the article is to introduce a new and powerful approach to image editing that uses the Poisson formula as a basic tool. The central concept is to blend and edit images smoothly, while also keeping colors consistent at the borders of overlapping regions. The paper presents a framework for Poisson image editing, emphasizing the importance of defining appropriate boundaries and operating in the gradient domain to achieve great results. It also describes applications of this technique, including seamless cloning, image composition, and texture synthesis.

Article - Poisson Image Editing

Mixing gradients - overview and comparison

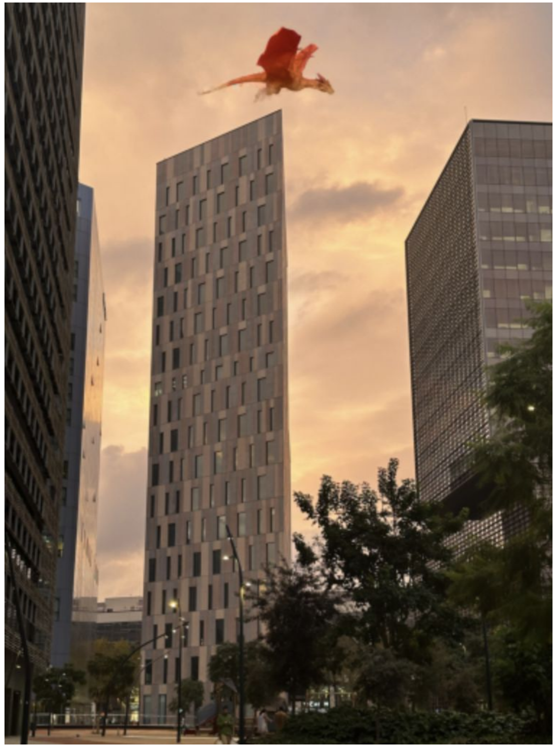
The concept underlying mixed gradients involves the computation of gradients for individual pixels within the region where the object is placed, which is represented by the omega parameter. Subsequently, a comparison is made between the gradient of the pixel from the cloned image and the gradient of the image to which it is being cloned. The selection is then made based on the stronger gradient among the two.

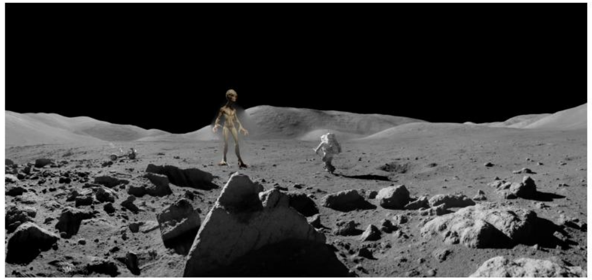
Although Poisson blending and blending gradients are both image processing techniques for combining images, they are not equivalent and may not always produce the same visual output.

Poisson blending excels at creating seamless transitions between source and destination images by solving the Poisson equation to ensure continuity of boundaries. In contrast, gradient blending involves manipulating the gradient of pixel values to produce a new image, the results of which may depend on the specific gradient blending algorithm used. Although the visual results of these techniques may appear similar in some cases, the choice between them should depend on the specific requirements of the task and the desired visual effects. Furthermore, the specific implementation and parameter choices can significantly affect the results, so experimentation and fine-tuning are critical to achieve the desired results for a specific image blending scenario.

Article - Poisson Image Editing

Mixing gradients - Results





Article - Poisson Image Editing

Seamless Cloning and Poisson Solution to Guided Interpolation

Seamless cloning is an application of the Poisson image editing framework, focusing on seamlessly integrating one part of an image into another. The goal is to achieve a visually seamless integration without any visible seams or discontinuities, ensuring that the integrated region appears as if it were an inherent part of the original image. It excels in scenarios where maintaining visual integrity is a top priority, but it may be more complex and resource-intensive than traditional Poisson blending.

This method use the power of Poisson's equation as a mathematical tool to achieve simple and guided interpolation in image editing in. When applied to the appropriate boundary conditions it serves as a key to compute a function whose gradient is close to the prescribed vector field, called the direction vector field. Thus the method gives the recursive function capable of projecting in boundary conditions, preserving the prescribed direction field. This method offers greater control and flexibility in image editing, allowing for the preservation of guided features and the integration of specific user-defined guidance. However, it comes with added complexity and requires more user involvement compared to traditional Poisson blending. The choice between the two methods depends on the specific requirements and goals of the image editing task at hand.

Github Repository

https://github.com/gunjanmimo/C2-IMAGE-INPAITING/tree/main/WEEK%202