COMPUTATIONAL PHOTOGRAPHY

- Broadly defined as algorithms that enable high-capability generation of images and other visual measurements, through improved camera design and/or providing advanced image editing abilities.

Intersection of Three disciplines

- Optics - Vision - Graphics

Two kinds of Comp. Photography Work

- Intersection of Optics + Vision: Better cameras that make inference easier
- Intersection of Vision + Graphics: Generate better images using vision-based reasoning

We’ll first talk about the latter.

CG2REAL

- The problem: Photo-realistic scenes take many many hours to render.

Need to have complex shape and texture models for a scene, simulate complex light-transport (inter-reflections, scattering, etc.) by ray tracing.

A simpler rendering on the other hand doesn’t look real.
Use a simpler renderer, and "transfer" detail by looking at a dataset of real images.

1. Find Similar Images
   - Use SIFT and other scene features to recover a small set of similar images from a real image dataset, given a query 'rendered' image.
   - Similar idea to content-based image retrieval: except that the query is a rendered image.
   - Incorporate user input: automatically select 30 images, and have a user select the best 5.

2. Co-Segmentation: Find corresponding regions to transfer local detail and color properties.
   - For each potential real image, consider the pair of (rendered, real) image and do a co-segmentation.
   - Assign every pixel in each image a label to minimize an energy which promotes:
     - Same label for spatial neighbors in each image independently.
     - Same label for pixels with similar appearance across images.

3. Local Style Transfer
   - Now that you have matched real regions to rendered regions, transfer color and texture properties.
   - Color and tone transfer by matching histograms of intensities and gradients across regions.
   - Texture transfer by aligning different shifted copies of real region with rendered region, and solve a graph-cuts problem to figure out which shifted copy to transfer gradients from at each location.
Example of where computational image generation can be done by approximate image-analysis rather than ‘rendering’.

In general, there is a large class of methods that allow you to manipulate images in this way.

Instead of “inverse rendering” an image to get a full description of the scene (texture, material, depth, lighting, etc.), modifying that description, and re-rendering ...

approximate with operations that work directly in the image domain.

Sometimes called image-based rendering.

Consider the case where you are given an image, and want to change it’s aspect ratio: from landscape to portrait.

Option 1: Just do imresize.

But this warps the image and makes things look stretched out.
Option 2: Do a crop.

But this is sub-optimal. You might be forced to crop out interesting things in the image.

In this case, you might have wanted to show the shape of the rock and its boundary with the sea.

But the original image has a lot of redundant information. Don’t need to see as much of the sea or the beach.

Could we remove ‘un-interesting’ pixels, in a way that overall content structure is preserved?

Option 3: Seam Carving!

Basic idea, to reduce the height (or width) by one pixel, find an "optimal" line of pixels from left-to-right (top-to-bottom) to delete from the image.

This line is determined by solving an optimization problem: minimize the gradient magnitudes at all pixels you are deleting, while making sure they form a connected line.

To decrease height by \( N \) pixels, apply this procedure successively \( N \) times.

Let’s say we want to decrease height.

Define an energy function

\[
\epsilon(x, y) = | \nabla_x I(x, y) | + | \nabla_y I(x, y) |
\]

- Sum of magnitudes of \( x \)- and \( y \)- derivatives.
- The idea is that if we delete a pixel \((x, y)\) in the image with low \( \epsilon \), it won’t be "missed".

Now our goal is to find a horizontal line defined as: \((1, y_1), (2, y_2), (3, y_3), \ldots (W, y_W)\)

- Where \( |y_x - y_{x-1}| \leq 1 \).
- Such that \( \sum_{x=1}^{W} \epsilon(x, y_x) \) is minimized. This is the energy of the line or "seam".
- Once we have found the optimal seam, we delete it by shifting all pixels up:

```python
inew = np.zeros(np.shape(I[:,:-1]))
for x in range(W):
inew[:y[x],x] = I[:y[x],x]
inew[y[x]+1:,:x] = I[y[x]+1:,:x]
```

Same idea applies for finding a 'vertical' seam.
SEAM CARVING

\[ y_1, y_2, \ldots, y_W = \arg \min x \sum_{x=1}^{W} e(x, y_x), \quad |y_x - y_{x-1}| \leq 1, \forall x \]

- This is an optimization over discrete pixel locations \( \{y_x\} \), where each \( y_x \in \{1, \ldots, H\} \).
- You can solve this by dynamic programming. How?

Viterbi Decoder

- Initialization: \( M(1, y) = e(1, y) \quad \forall y \)
- Go from left-to-right:
  \[ M(x, y) = e(x, y) + \min \{M(x-1, y-1), M(x-1, y), M(x-1, y+1)\} \]
  \[ C(x, y) = \arg \min \{M(x-1, y-1), M(x-1, y), M(x-1, y+1)\} \]
- Once you’ve reached the end, \( \{W, \arg \min_y M(W, y)\} \) gives you the end of the optimal seam.
- Backtrack by looking at \( C(x, y) \) to get the rest of the line.

SEAM CARVING

Now let’s say you want to reduce both height and width: from \((H, W)\) to \((H’, W’)\)

- Option 1: Uniformly scale by \( s \) and then do seam-carving to fix aspect-ratio.
  - \( sH = H’, sW \geq W’ \). Carve vertical seams.
  - \( sW = W’, sH \geq H’ \). Carve horizontal seams.
- Option 2: Remove both \((H - H’)\) horizontal and \((W - W’)\) vertical seams.

But in what order? Select order to minimize energy of all removed pixels (computed on the image they were removed from).

Define \( T(r, c) \) as the cost of reducing width and height by \( r \) and \( c \) pixels respectively. \( I_{r,c} \) is corresponding reduced image.

\( T(0, 0) = 0, I_{r,c} \) is \( I \).

- Given \( T(r-1, c) \) and \( T(r, c-1) \) (unless \( r \) or \( c \) are 0, in which case consider only one)
  - Consider reducing width from \( I_{r-1,c} \) or height from \( I_{r,c-1} \).
  - The corresponding energies will be \( T(r-1, c) \) or \( T(r, c-1) \) plus the energy of their respective seams.
  - Pick whichever is lower, and set \( T(r, c) \) and \( I_{r,c} \) respectively.
- Do this in order till you get to \( T(r, c) \) for your desired reduction size.

Content Amplification

- Let’s say you don’t want to reduce the size of your image. Just make the content in it larger.
- Scale up by uniform scaling, and then use seam carving to get back to original size.
SEAM CARVING

Seam Insertion

- You can also use seam carving to “increase” width or height, by expanding instead of deleting seams.
- Find all the seams in your image. And now, instead of deleting them, insert additional pixels equal to the average of the pixels on either side of the seam.

(Left) Non-uniform Scaling. (Right) Seam Insertion.

SEAM CARVING

Object Removal

- User selects an object—set of pixels—to remove from a photograph.
- Apply seam-carving, where each seam passes through at least one of the pixels to be removed.
- Keep applying carving till all object pixels are removed.
SEAM CARVING

Object Removal

- After object removal, you can also do seam insertion to change image back to the original size.

SEAM CARVING

- Provides a basic framework for image re-targeting.
- Simple gradient energy works surprisingly well in many cases.
- There's a lot of work though to augment this with “saliency” information, to make sure you don't delete “important” parts of the image.
- For example, apply a face detector, and make the energy of regions with detected faces very high.

TEXTURE SYNTHESIS AND TRANSFER


Basic Task: I give you an example of texture of some size, fill in a larger region with the same texture.

- Top Left: Original. Everything else has a different shoe removed.
Version 1: Divide target region into non-overlapping blocks. For each block, select a random block from the source.

But this causes obvious boundaries between blocks.

Version 2: Place overlapping blocks, and when selecting a new block, make sure that it matches the existing block in area of overlap (error is less than some threshold).

- Do this by going in a raster scan order over blocks. When searching for a new block at a position, consider overlap with block to its left and top.
- Gives better results, but still some boundary artifacts.

Version 3: When placing overlapping blocks, find an optimal cut: which pixels to take from existing left/top and which from new right/bottom block.

- The cut is defined as the line which minimizes deviation between the two overlapping blocks.

Simple idea, but works surprisingly well in practice! Only parameter choice is block size.
TEXTURE SYNTHESIS AND TRANSFER

- Simple idea, but works surprisingly well in practice! Only parameter choice is block size.

- You can apply this idea also for texture transfer.

  - Now given a source texture and a target image.

  - The idea remains the same, fill in blocks selected from the source image on the target image plane. But in this case, you have an additional cost for selecting blocks: want to promote blocks that are ‘similar’ to the target image in the location you’re filling in.

NEURAL STYLE TRANSFER

- Similar idea. Make one image have the “style” or texture quality of another.

- But define content and texture based on activations of a deep CNN trained for some task.

  - Gatys et al., Image Style Transfer Using Convolutional Neural Networks, CVPR 2016:
    - Use a pre-trained network for classification on ImageNet.
    - Values of higher layers represent “content”: Try to preserve them
    - Covariances of other layers represent style: Try to match them with other image
NEURAL STYLE TRANSFER

Set this up as an optimization problem, and minimize with SGD+Backprop from a random init.

POISSON IMAGE EDITING


- Basic Idea: If I have horizontal and vertical gradients of an image at each pixel, as well as some absolute intensities at boundaries, this is enough to reconstruct the image.
- It is easier to edit, replace gradients than pixels, and reconstructing images from edited gradients looks more natural.

Poisson Solver

- Given estimates of \( V_{pq} \) of differences in intensities between pixels \( p \) and \( q \), i.e., \( V_{pq} = I_p - I_q \):
  - For example, can be horizontal and vertical gradients at all locations: \( I(x + 1, y) - I(x - 1, y) \), \( V(x, y) \)
  - And values of \( \{ I_p^* \} \) for a small set of ‘boundary’ locations \( p \in \partial \Omega \)

Reconstruct \( I \) by minimization:

\[
I = \arg \min (I_p - I_q - V_{pq})^2, \quad \text{such that} \quad -I_p = I_p^* \text{, } \forall p \in \partial \Omega
\]

- This is a simple least-squares minimization. Can solve by conjugate gradient (or Frankot-Chellappa, if you have the same gradient at all pixels).
- Remember, you did this for Normals -> Depth in CSE 559A.

- Applications: Conceal objects/textures. Remove \( V_{pq} \) for values in regions where you don’t want to retain objects.
POISSON IMAGE EDITING

- Applications: Texture Flattening: Set all gradients below a threshold to 0.

![Image of texture flattening example]

POISSON IMAGE EDITING

- Applications: Insert Objects by copying over their gradient field.

![Image of object insertion example]

POISSON IMAGE EDITING

- Applications: Insert translucent objects by mixing gradient fields. (Often, selecting the max by magnitude of the source and destination gradient).

![Image of translucent object insertion example]