Comp Photography Final Project

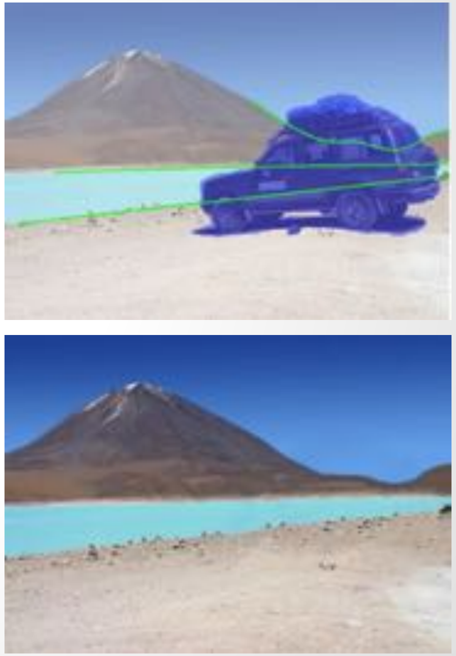
Graham Bryan & Nick Merlene

Spring 2019

gbryan7@gatech.edu, nmerlene3@gatech.edu

Image Completion with Structure Propagation

In this project, we “complete” an image by using structure propagation and texture synthesis. User-specified regions are removed first. Then, user-specified curves (structures) are preserved through structure propagation. Finally, texture synthesis is used to fill in the remainder of the unknown region(s).



Example from paper

The Goal of Your Project

Original project: Building Edge Detection and Camera Pose Estimation

Original project scope: Our original goal was to estimate the camera pose and detect the edges of buildings using perspectives photos from open source street platforms. Our final artifact would have been generating a figure similar to Figure 8 in this paper: https://arxiv.org/pdf/1601.07630v2.pdf, with the exception that we planned to use buildings/photos from our local city.

What motivated you to do this project? Our motivation to pursue this project came from two different sources. First and foremost, we both have a background in aerospace and have strong interests in satellite navigation, where building edge detection, and more specifically, camera pose estimation, have a lot of potential to be used to refine noisy GNSS locations in urban canyon environments. See the following paper for more information: http://openaccess.thecvf.com/content_cvpr_workshops_2014/W03/papers/Chu_GPS_Refinement_and_2014_CVPR_paper.pdf. Additionally, this project also had strong ties to Computational Photography, since we would have needed to use a variety of gradient/derivative filters and extract homography matrices in order to properly label building faces and detect building edges.

Scope Changes

- **Did you run into issues that required you to change project scope from your proposal?**

  Yes, unfortunately, there were several issues that prompted us to change the scope of our original project. The first issue we encountered is trying to find an urban landscape that was as "simple" as the one used in the paper we were trying to reproduce. For example, look at Figures A and C on the following side. The scene/landscape in Figure A is fairly clean, with mostly flat building faces and minimal trees, cars, street poles/signs, etc. However, in the example from our city (Austin, TX) in Figure C, the architecture of the buildings is a bit less box-shaped and there are also trees, traffic lights, awning, and many other obstacles. The second, and most prominent issue that we faced though, was in generating 3D building models. The original authors claimed that they were able to use building footprints with estimated heights in order to generate simple 3D building models (see Figure B on the next slide). However, in practice, this proved to be fairly complicated. We briefly experimented with using the 3D building models from Google Earth (see Figure D on the next slide), but this brought its own set of challenges. Alternatively, we could have mitigated some of these issues by using the same photos, models, etc. as the original authors, but we were unable to acquire the dataset used in the paper and what we were able to extract from the PDF was not sufficient.

  In the end, after a week of trying to make our original proposal work, we realized we were spending far too much time on trying to resolve issues that were not related to computational photography. For that reason, we decided to entirely change the scope of our project so that we could spend the remainder of our time focusing on computational photography, instead of sinking more time into resolving issues unrelated to the course material.

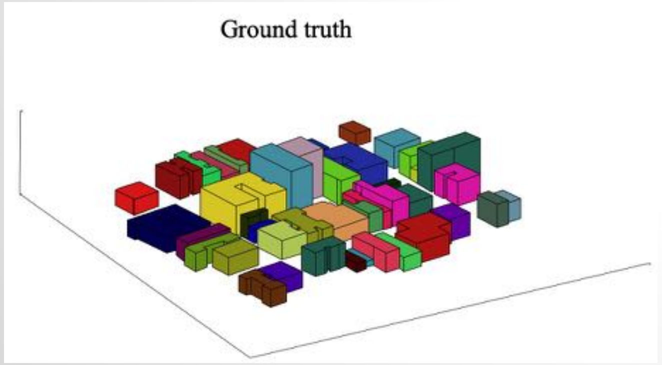
- **Give a detailed explanation of what changed:** See Slide 6

Scope Changes (Continued)

A

C

B



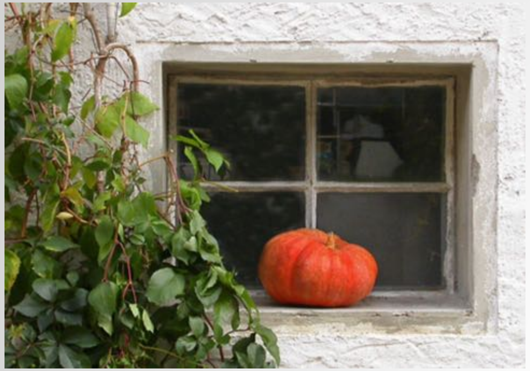
D

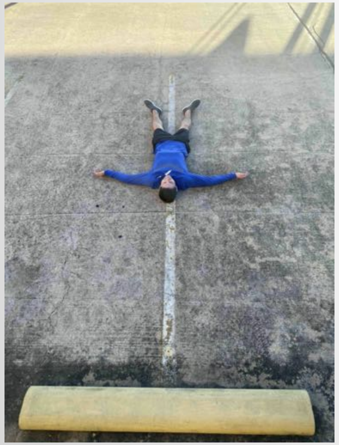
Scope Changes (Continued)

- **Give a detailed explanation of what changed:**

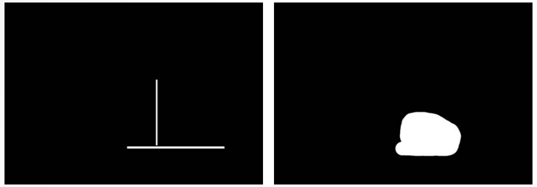
  Our scope changed entirely once we realized the project in our original proposal was not feasible. At that point, we scanned through the approved partner project proposals on Piazza and eventually settled on Image Completion with Structure Propagation. We chose this topic since it incorporates several topics that we have already touched on in one way or another, such as energy minimization, dynamic programming, texture blending/synthesis, etc. Additionally, it was approved as an adequate scope for two people and also seemed like it would have minimal complications that were unrelated to computational photography itself.

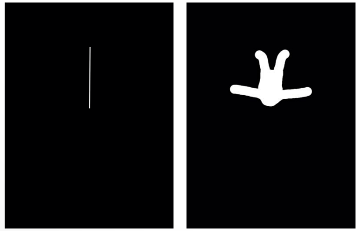
Showcase



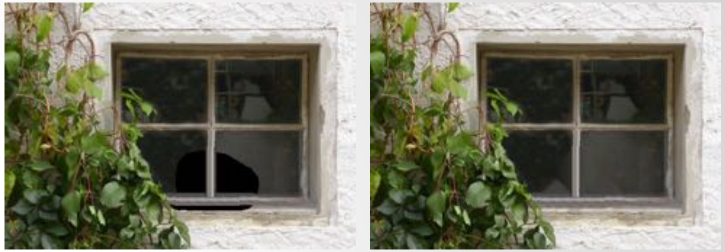


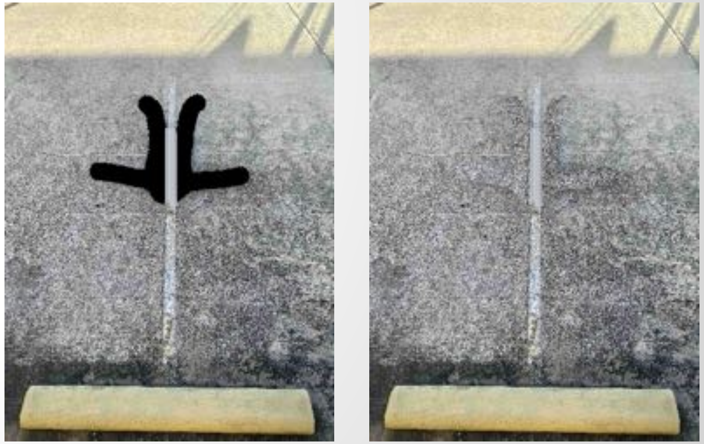
Input Image





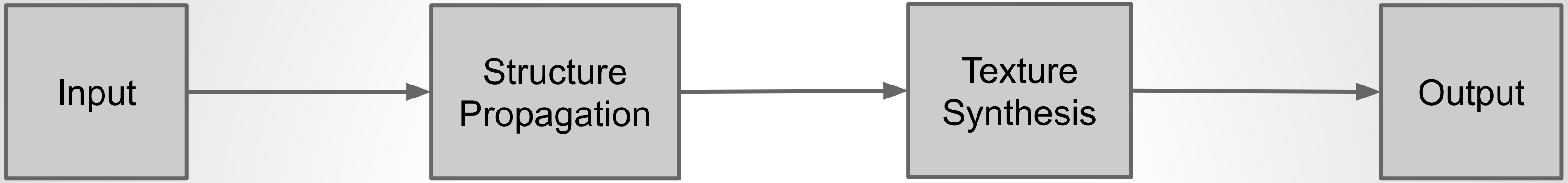
User Generated Masks





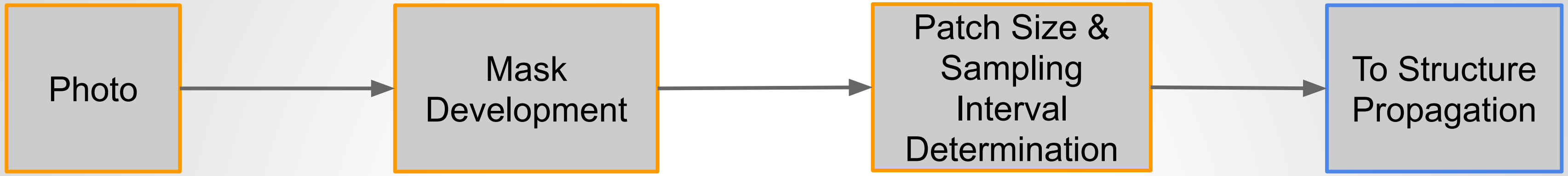
Output

Project Pipeline (High Level)



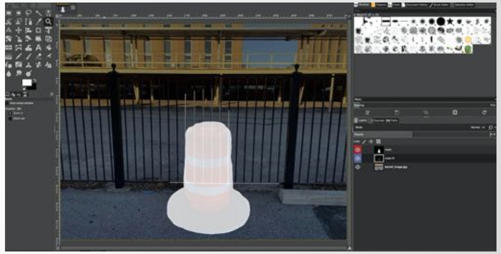
At a high level, our pipeline is composed of four steps: **Input**, **Structure Propagation**, **Texture Synthesis**, and **Output**. The slides that follow break down each step of the pipeline into further detail. Note that any specific code, functions, equations, etc. that are mentioned will be referred to at a high level and broken down into greater detail in the “Code Functional Description” slides.

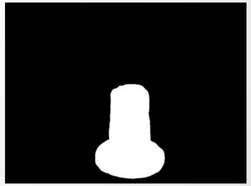
Project Pipeline (Input Pipeline)

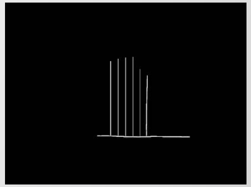


The pipeline starts with the **Input** step. Before any of actual processing takes place, several steps must be completed. First, a photo must be selected or captured. There will be more detail on this later, but when selecting or capturing a photo, we must keep in mind the "structure(s)" that we would like propagated and the "unknown" region that we would like to be removed (and subsequently filled in via texture synthesis). After a photo is chosen, two masks need to be developed. We used the GNU Image Manipulation Program (GIMP) for this part, as evidenced by Figure A. The first mask, known as the "unknown mask", specifies the unknown region. An example is shown to the right in Figure B. The other mask, the "structure mask", contains the user-specified curves (structures) that need to be propagated. An example of a structure mask is shown in Figure C. Next, the "patch size" and "sampling interval" need to be determined. Patch size, as the name implies, specifies the size of the patches that will be synthesized from the known (sample) region and applied to the unknown region along the applicable structure curve. Our original plan was to automate the determination of these parameters and remove the need for user input, but there were some difficulties with this. There will be more on patch size and sampling interval later in the "Code Functional Description" section. Finally, we need to specify whether or not we would like to use the "fast" option to decimate/downsample the input image to decrease processing time. While ideally, this would not be necessary, due to the nature of the dynamic programming algorithm, and the fact that the code is not greatly optimized, this is necessary for almost all input images to decrease the overall runtime from hours to minutes.

A





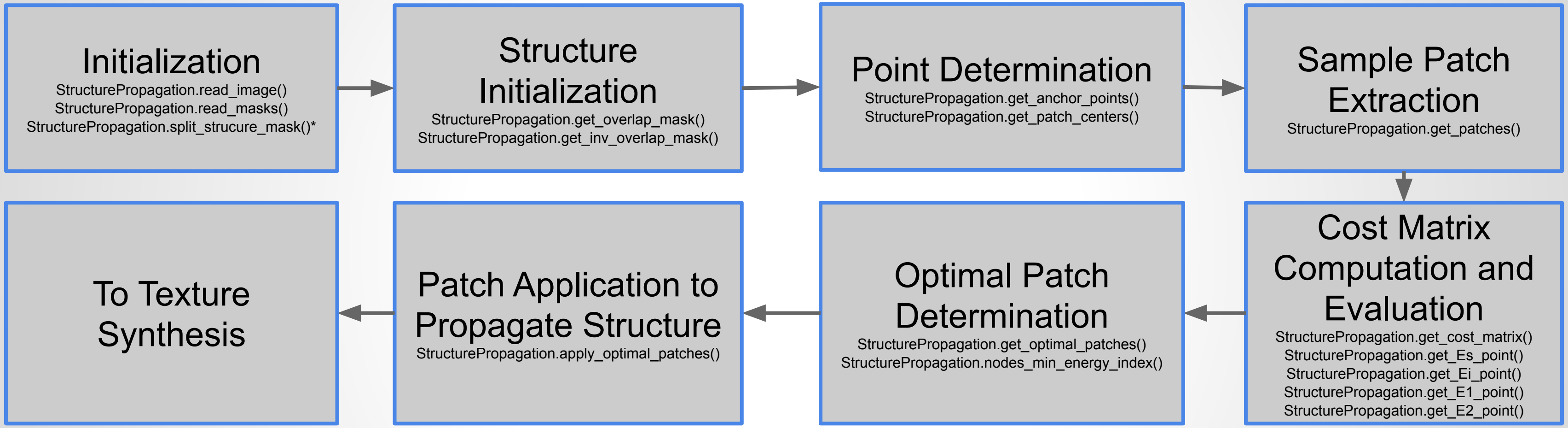


B

C

* = External Packages/Code Used
🟧 = Manual Step
🟦 = Automatic Step

Project Pipeline (Structure Propagation)


Initialization
StructurePropagation.read_image()
StructurePropagation.read_masks()
StructurePropagation.split_strucure_mask()*
Structure Initialization
StructurePropagation.get_overlap_mask()
StructurePropagation.get_inv_overlap_mask()
Point Determination
StructurePropagation.get_anchor_points()
StructurePropagation.get_patch_centers()
Sample Patch Extraction
StructurePropagation.get_patches()
Cost Matrix Computation and Evaluation
StructurePropagation.get_cost_matrix()
StructurePropagation.get_Es_point()
StructurePropagation.get_Ei_point()
StructurePropagation.get_E1_point()
StructurePropagation.get_E2_point()
Optimal Patch Determination
StructurePropagation.get_optimal_patches()
StructurePropagation.nodes_min_energy_index()
Patch Application to Propagate Structure
StructurePropagation.apply_optimal_patches()
To Texture Synthesis

The next major step in the pipeline is **Structure Propagation**. Note that major function names are specified below the name of the step in each of the boxes above.

First, is the "**Initialization**" step, which is composed of the following sub-steps:

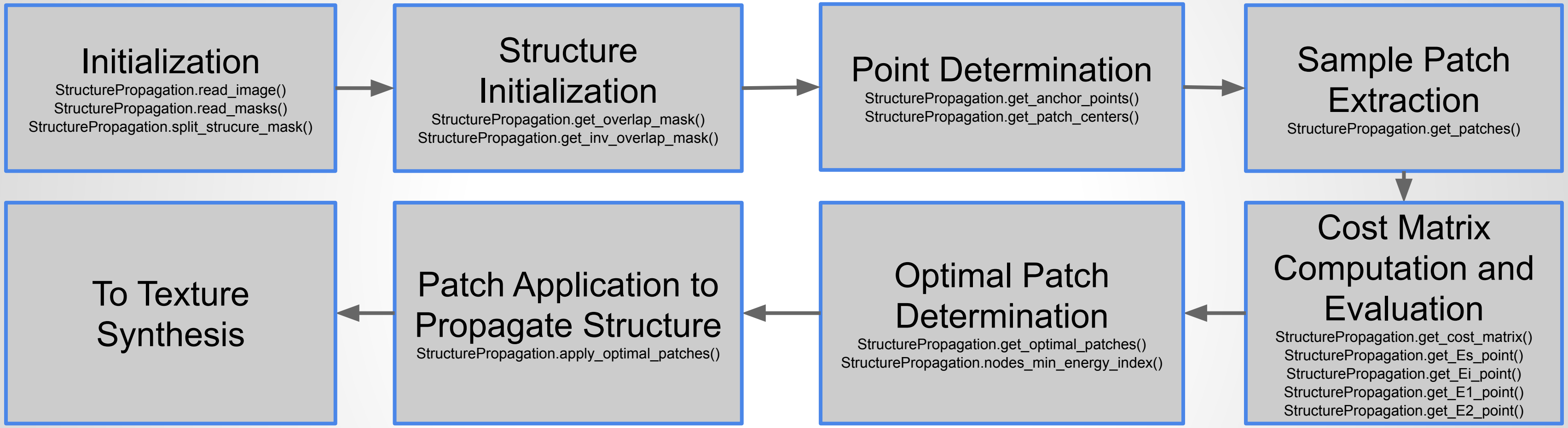
1. Image and Image Masks (Unknown and Structure) are read
2. Image is downsampled from the original size, if the "fast" option is specified
3. Structure Mask is split into separate masks, one for each structure, so that we can isolate one structure to propagate at a time

Next, is the "**Structure Initialization**" step, composed of the following sub-steps:

1. Generate the structure "overlap mask", which indicates the overlap between the structure mask and unknown mask
2. Generate the "inverted overlap mask", which indicates where the structure mask and unknown mask do NOT overlap

* = External Packages/Code Used
🟧 = Manual Step
🟦 = Automatic Step

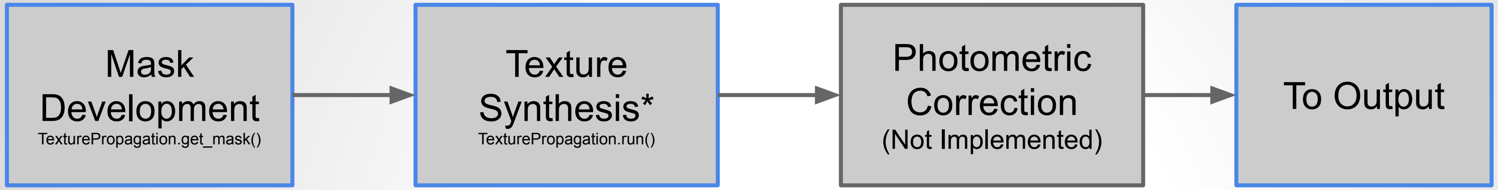
Project Pipeline (Structure Propagation, Continued)


Initialization
StructurePropagation.read_image()
StructurePropagation.read_masks()
StructurePropagation.split_strucure_mask()
Structure Initialization
StructurePropagation.get_overlap_mask()
StructurePropagation.get_inv_overlap_mask()
Point Determination
StructurePropagation.get_anchor_points()
StructurePropagation.get_patch_centers()
Sample Patch Extraction
StructurePropagation.get_patches()
Cost Matrix Computation and Evaluation
StructurePropagation.get_cost_matrix()
StructurePropagation.get_Es_point()
StructurePropagation.get_Ei_point()
StructurePropagation.get_E1_point()
StructurePropagation.get_E2_point()
Optimal Patch Determination
StructurePropagation.get_optimal_patches()
StructurePropagation.nodes_min_energy_index()
Patch Application to Propagate Structure
StructurePropagation.apply_optimal_patches()
To Texture Synthesis

Next, is the **Point Determination** step, in which both the anchor (target) points are determined for the unknown region (using the structure mask), and the sample (source) points are determined from the known region (using the inverted overlap mask). Then comes **Patch Extraction**, where we extract several patches from our known region that we want to use later for application to the unknown region along the user-specified structure curve. Then, the **Cost Matrix Computation and Evaluation** takes place. More detail will be provided in the "Code Functional Description" slides, but this step is where Dynamic Programming is used to find the minimum cost path in order to minimize the energy for structure propagation. The paper also goes into great detail about Belief Propagation. We briefly explored using this, instead of Dynamic Programming, but since our input image set did not have any intersecting curves, we reverted to using only Dynamic Programming since the paper specifies that Belief Propagation is only used if "multiple intersecting curves are specified." Next, we backtrace through the cost matrix to determine the optimal source patch to use to apply to each anchor point in the **Optimal Patch Determination** step. Then, we go through each anchor point in the unknown region (along the user-specified structure curve) and apply its optimal patch in the **Patch Application to Propagate Structure** step. Finally, we either move on to the next structure, if needed, or proceed to Texture Synthesis.

* = External Packages/Code Used
🟧 = Manual Step
🟦 = Automatic Step

Project Pipeline (Texture Synthesis)

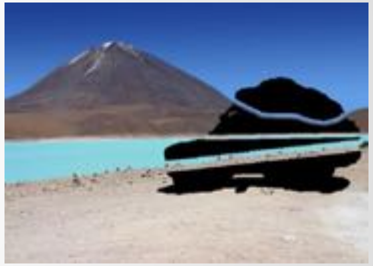


After structure propagation is complete, we are still left with one or more unknown regions that need to be properly filled in, such as in Figure A to the right, with **Texture Synthesis**. First, we need to read in an image that looks similar to the figure to the right, where unknown regions are black. Then, a mask is automatically generated, similar to Figure B, in which only the regions that need to be filled in are colored white, and the rest is colored black.

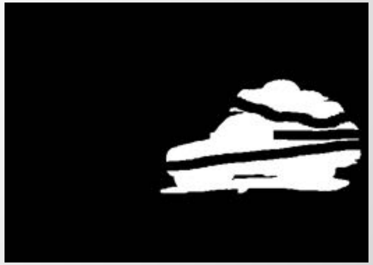
Next, for the actual texture synthesis step, the research paper that we are attempting to replicate did not include much information at all about how texture synthesis was achieved. They essentially deferred to texture synthesis algorithms that were defined in other research papers. They use the algorithm defined by *Ashikhmin 2001; Hertzmann et al. 2001; Jia and Tang 2003* to propagate texture information from each subregion, where a subregion is each of the unknown region partitions that exist after structure propagation. In order to define the propagation order, they use confidence map similar to those defined by *Drori et al. 2003* and *Criminisi et al. 2003*.

See the next slide for more detail on how we tackled texture synthesis given the above information.

A

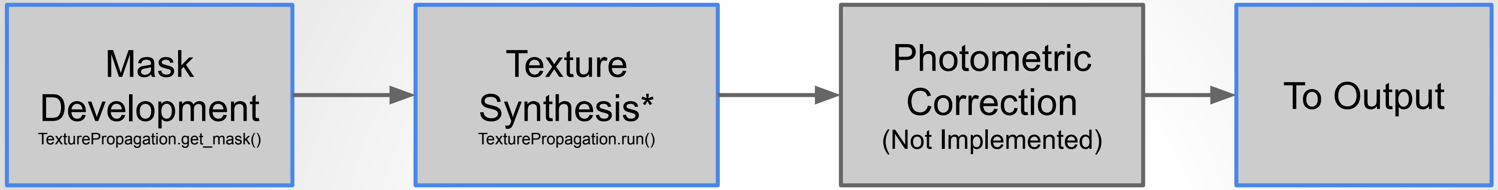


B



* = External Packages/Code Used
🟧 = Manual Step
🟦 = Automatic Step

Project Pipeline (Texture Synthesis, Continued)

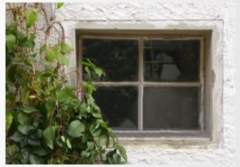


Since the texture synthesis step of the paper was accomplished using other, previously established algorithms, and was not explicitly defined, we ultimately made the decision that it would be out of the scope of this project to dive down into the papers the original authors reference to re-create the texture synthesis algorithms used. We wanted to avoid scope creep and to focus on only the algorithms explicitly defined in our paper. For that reason, we simplified our process to just include two simple texture synthesis algorithms, each with its own set of strengths and weaknesses.

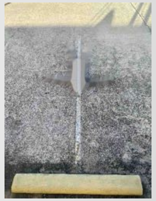
The first method we used is a simple wrapper around *cv2.inpaint*, which performs image inpainting to fill unknown regions of an image by blending in values from neighboring regions. See the images to the right for an example of where this method worked well (Figure A) versus where it did not work as well (Figure B).

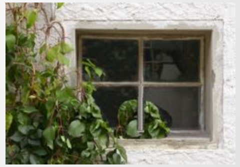
The second algorithm, which is a bit more rudimentary, is what we termed as "texture fill using adjacent regions". This algorithm works by taking a mask of the unknown region(s) and shifting it to the left of, right of, above, and below the unknown region(s), extracting the pixels from the shifted region, and applying them back to the unknown region. In most cases, this algorithm does not perform well, especially in cases where there is a lot of variation in the background. However, in scenes with a constant/consistent background which do not have a lot of variation, e.g. a blue sky, a solid white canvas, lots of natural foliage, etc, it generates results that look more natural than those produced by the other method, which relies on *cv2.inpaint*. See the images to the right for an example of where this method worked well (Figure D) versus where it did not work well (Figure C).

A

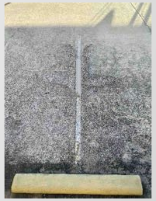


B





C



D

Finally, the last step, if we had implemented it, would be **Photometric Correction**. We chose not to implement this since it was only briefly touched on in the paper, referenced other papers rather than explicitly mentioning the algorithm, and also seems to only be necessary for images with "significant spatial variations in intensity or color" to reduce the visibility of the seams between overlapping patches.

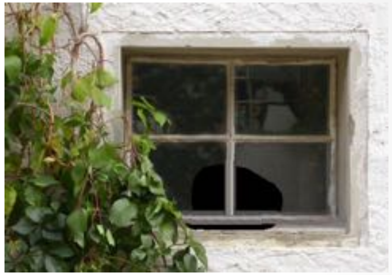
* = External Packages/Code Used
🟧 = Manual Step
🟦 = Automatic Step

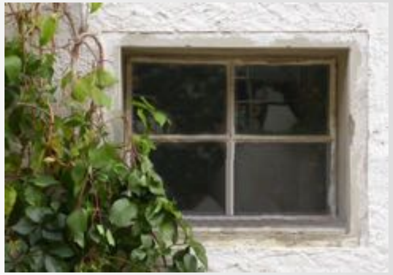
Project Pipeline (Output)

In the **Output** step, several figures are saved to disk. The first is the original image after structure propagation is complete, but before texture synthesis has occurred (Figure A). Next, the image with structure propagation applied and texture synthesis applied using the first method (*cv2.inpaint*) mentioned on the previous slide is saved (Figure B). Finally, a set of images with structure propagation applied and texture synthesis applied using the second method (texture fill using adjacent regions) are saved. There will be as many as 4 images in this set since we look for pixels to the left of, right of, above, and below the unknown region. However, in some cases we cannot generate the images for each shift since sometimes the shifted pixels might be outside of the image. For example, there would be no "shift down" image output from the image set to the right.

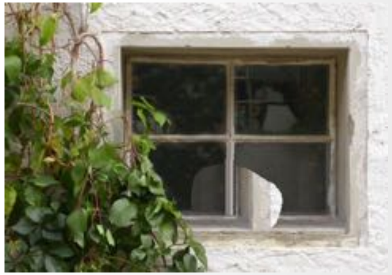
The final step is to manually select which of the (up to 5) output images best represent (to the human eye) how the original image *should* look if the object we are trying to remove never existed. In most cases, this was a trial and error process, and we ran each image set with several different combinations of parameters, and analyzed several of the different output images, before finally settling on an output image that looks the "best."

A





B



C

* = External Packages/Code Used
🟧 = Manual Step
🟦 = Automatic Step

Demonstration: Result Sets

Key for viewing results slides

Source Image:

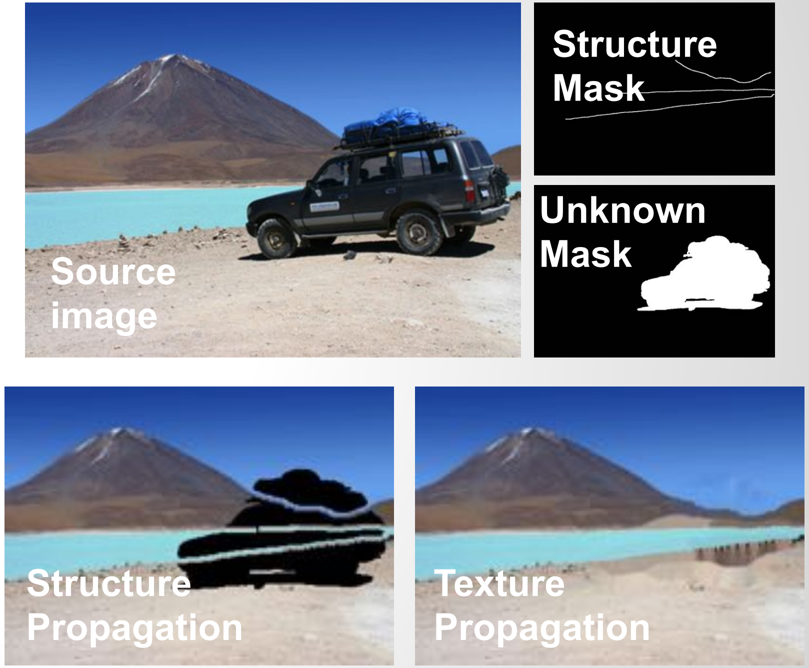
Details the source image

Masks:

Details the unknown and structure masks

Result:

Details the final structure and texture propagation results



Demonstration: Result Sets (Continued)

Car

Source Image:

The car on the beach image was taken from the technical paper for algorithm comparison purposes.

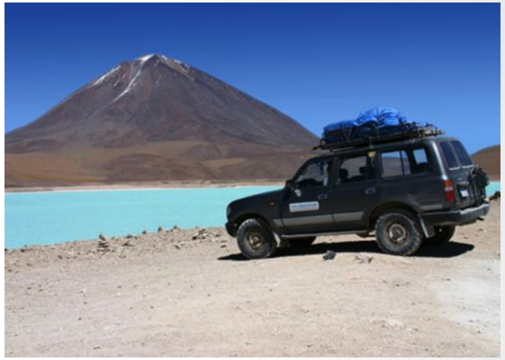
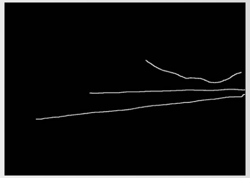
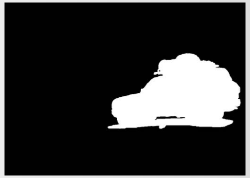
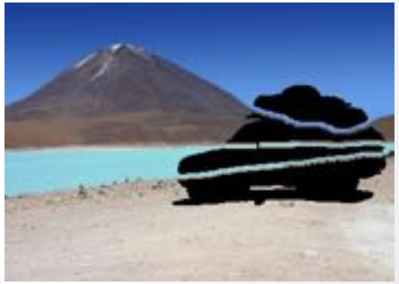
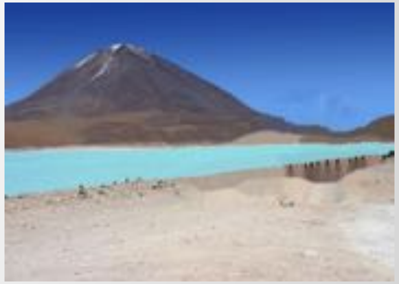
Masks:

Unknown Mask: This mask was one of the most difficult to make due to the cars shadows.

Structure Mask: This image had three curves (structures) to propagate: the mountain rim, base, and beach.

Result:

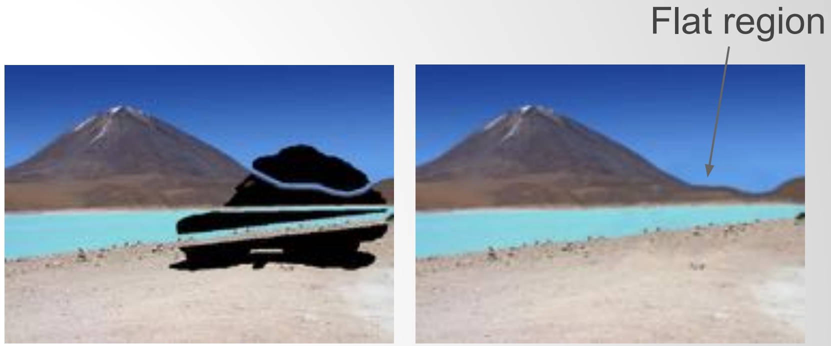
The mountain base structure faced difficulties running through our automated system. A black line can be seen in its structure due to there not being enough anchor points, meaning not enough patches. This could be resolved by tweaking the tuning parameters for only that curve, but our system takes parameters that are applied to all structures. Finally, the resulting inpaint image is not perfect, but all structures are completed and the unknown regions filled with reasonable patches.

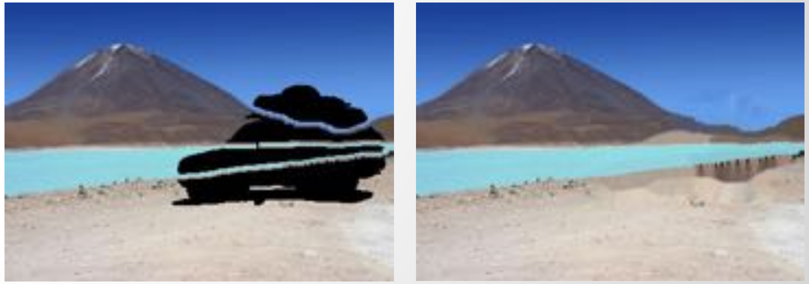
Demonstration: Result Sets (Continued)

Car (Continued)

Here we tried to reproduce the papers results. Clearly, there are differences between the two in both structure and texture propagation. As it will be mentioned in later slides, the papers' authors did a tremendous amount of patch tweaking to get each structure as flawless as they did. This can be seen clearly in the flat region of the mountains rim. To be able to fill in this structure, samples from outside the unknown region were taken along mountains rim to fill in the structure curve within the unknown region. Are there any flat samples to be seen? No, there are not. The authors took samples along the mountain slope and rotated each patch (hundreds to thousands) by some angle to fill in that region. Overall, our final results still produce a comparable image. With more time to tune each structure we believe we could achieve a high quality replica.



Result image from paper



Our Result

Demonstration: Result Sets (Continued)
Pumpkin

<u>Source Image:</u>

This pumpkin image was taken from the technical paper for algorithm comparison purposes.

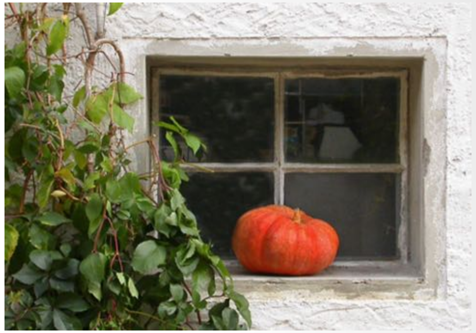
<u>Masks:</u>

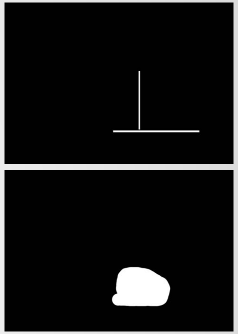
Unknown Mask: This is the region where the pumpkin and its shadow was.

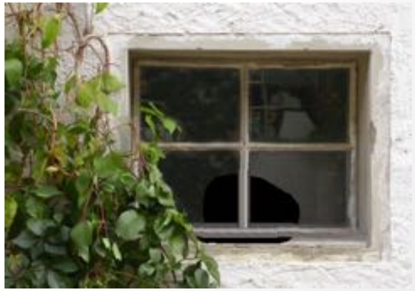
Structure Mask: The structures in this image completed the window frame (its vertical and horizontal beams) behind the pumpkin.

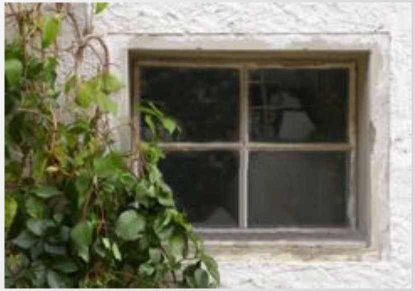
<u>Result:</u>

This propagation was one of our best results. The structure propagation was almost flawless in generating a completed window frame through the unknown region (i.e., the pumpkin). Texture propagation using image inpainting worked very well in this instance, since the textures in the windows were not incredibly detailed. We discuss in more detail in the next slide.





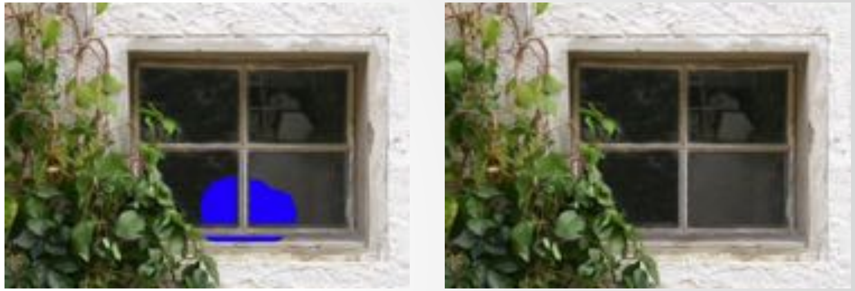


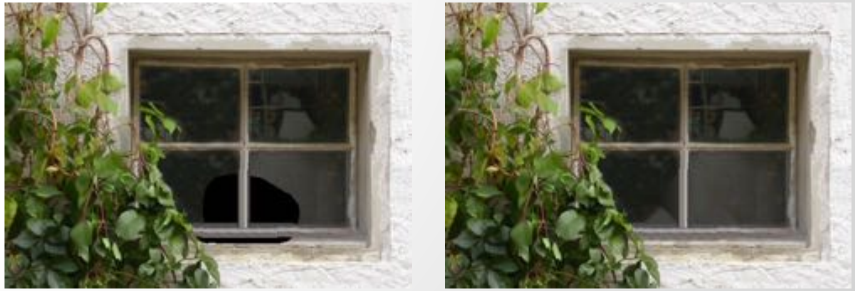


Demonstration: Result Sets (Continued)

Pumpkin (Continued)

Here is our best result compared to the papers. The resulting images after texture propagation are almost identical. However, it is clear that the image inpainting does not do as good as a job as the photometric correction used in the paper. If you look at the third window quadrant (bottom left) the image inpainting leaves the triangular murky bown shape caused by propagating the texture directly below it while the papers' does not have this. The vertical and horizontal structures of the window frame are close to identical between the comparisons. Ours however seems to be misaligned slightly in both structures. We believe this is due to how precisely the structure mask is drawn. It works best when the mask is evenly drawn down the center of the structure and the size of the patch is exactly the pixel by pixel size of the structure. Overall, we are happy with product of this image.


Result image from paper


Our Result

Demonstration: Result Sets (Continued)

Barrel

Source Image:

This image of a construction barrel was taken outside an office building in Austin, TX. We used an Iphone 11.

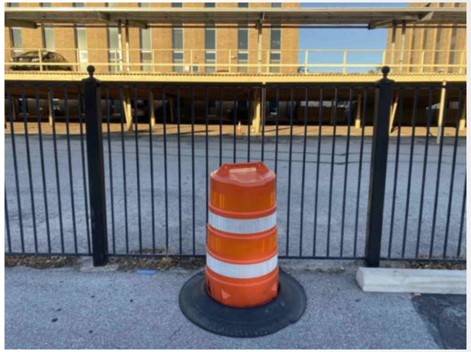
Masks:

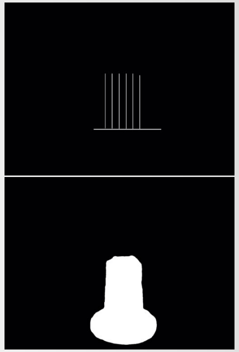
Unknown Mask: The unknown region is the construction barrel we wish to propagate the fence through.

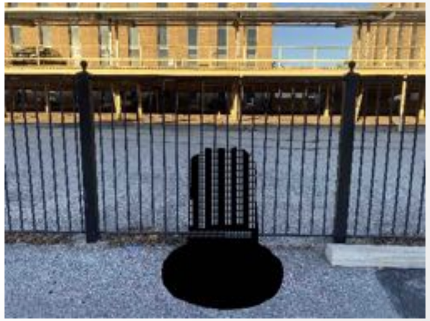
Structure Mask: One of our more complicated structures due to the amount of them. Each fence piece is masked to generate a structure mask with a total of 7 curves.

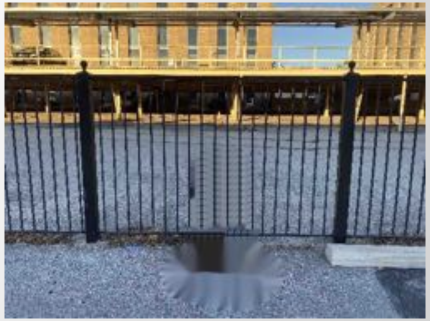
Result:

This propagation was one of the hardest to tune and propagate into a quality product. We were not happy with the end structure and texture propagation. The structures were not completely propagated due to not producing enough anchor points within the unknown region. This could be remediated by decreasing the sample interval but this increases the processing time significantly. In the end we left it how it is to show the difficulties in processing many structures. The final result used image inpainting to propagate the surrounding textures.







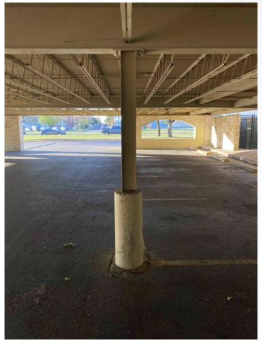


Demonstration: Result Sets (Continued)

Concrete

Source Image:

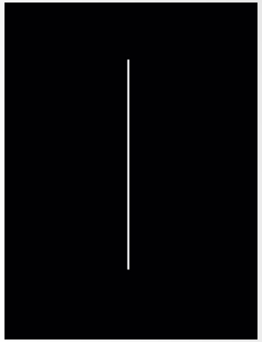
This image of a concrete and metal column was taken in a parking garage in Austin, TX. We used an Iphone 11.

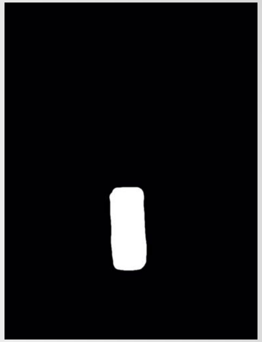


Masks:

Unknown Mask: This mask is of the cylindrical concrete column.

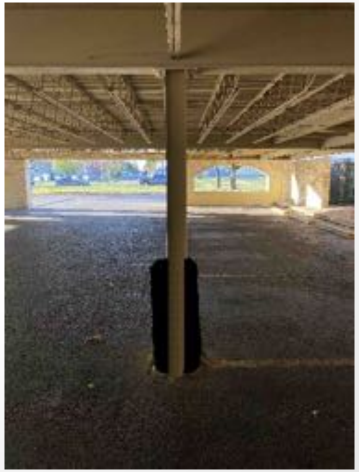
Structure Mask: The structure is the metal column we wish to propagate through the concrete cylinder.

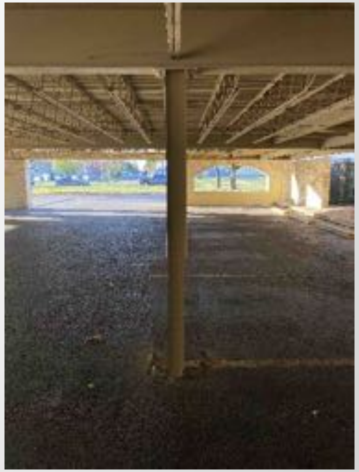




Result:

The structure was propagated with a patch size of 15 pixels, which is almost exactly the width of half the structure (so the structure is 30 pixels wide). If we went any wider than our patches would include non-structure regions (e.g, a piece of the car in the background). This is the first of our results where we used our texture synthesis technique, texture fill using adjacent regions, since it produced the best result. This technique mapped the unknown region after structure propagation to samples to the right of the column that were then superimposed back into the unknown region. Overall we were happy with the final result, especially since the texture fill allowed us to also propagate the yellow parking stripes.





Demonstration: Result Sets (Continued)

Stop Sign

Source Image:

This image of a stop sign was taken in Dallas, TX. We used an Iphone 11.

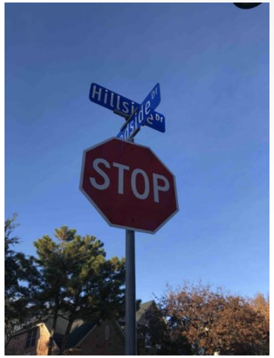
Masks:

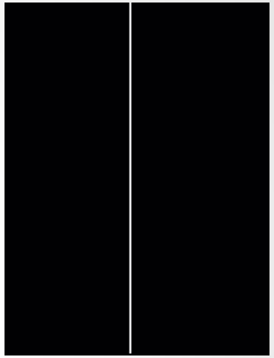
Unknown Mask: We took a different approach here by masking out the stop sign, the street names, and a region above it. The part to note is the region above, which was done in order for us to propagate the entire pole through the image.

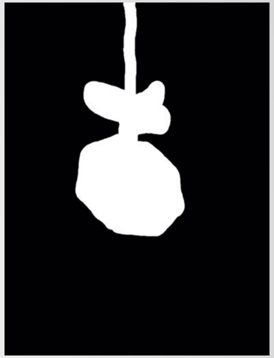
Structure Mask: This mask split the image in line with the pole.

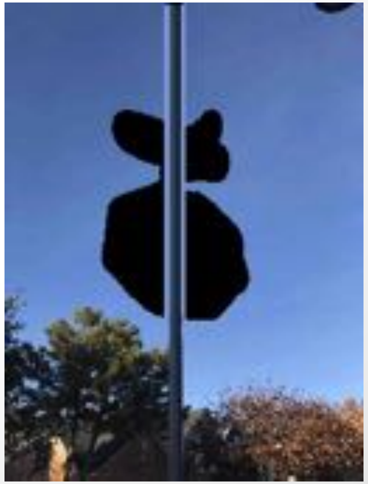
Result:

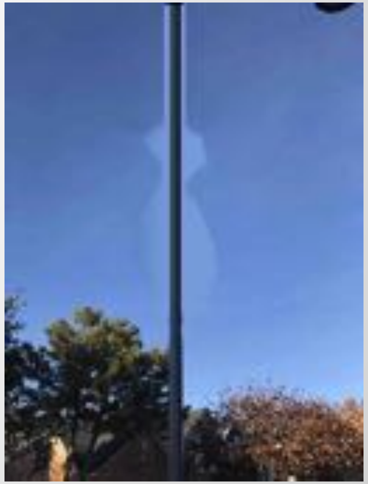
The length of structure mask curve does not have to expand the entire image, since its purpose is to let the algorithm know where to pick samples from. The reason we do it here is to demonstrate our algorithm choosing the best matching samples. In this case we were happy to see that samples above the unknown region were not chosen in the final structure propagation. The ending texture propagation using the inpaint technique shows our completed image with a fully structured pole through the image with the stop sign removed.





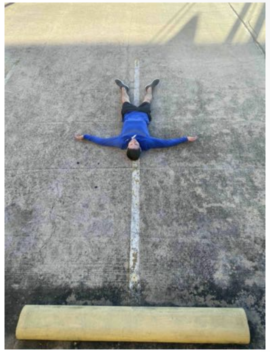
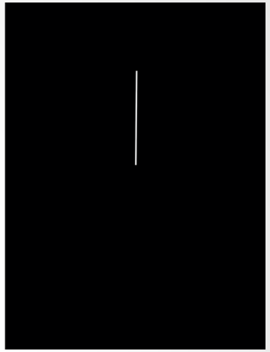
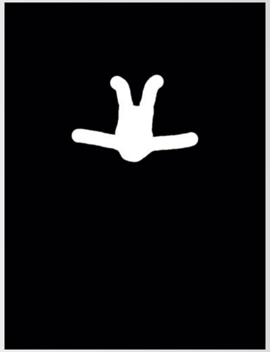
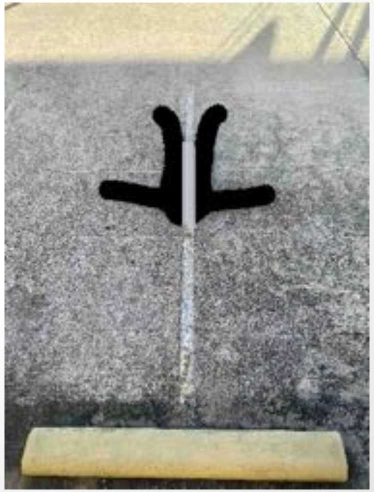
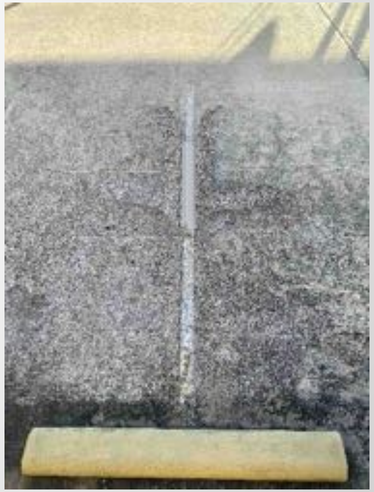






Demonstration: Result Sets (Continued)

Parking

<u>Source Image:</u>

This image of Nick was taken in a parking lot in Austin, TX. We used an Iphone 11.

<u>Masks:</u>

Unknown Mask: The unknown region that is masked is the full body of Nick.

Structure Mask: We wished to propagate the parking spot stripe through the body, so a curve going through the center of the stripe and body was masked.

<u>Result:</u>

The final result was one our best. We used texture adjacent propagation for this result to complete the parking space without the body. We tried multiple different patch sizes and intervals to produce this final result. Some of our failed results can be seen in the failures slide.

Project Development

Project Narrative and Issues Faced

We encountered several ups and downs as we progressed towards completion of our project. As mentioned in previous slides, we originally started with a project of an entirely different scope. However, after this didn’t work out, we encountered our first major setback when we had to quickly switch projects and finish our new project in just two weeks. While we were getting acclimated to the new project, we faced several difficulties ranging from misinterpreting some of the math behind the algorithms to dealing with uncertainties and ambiguities in details that were omitted from the paper. After we were able to resolve those issues, we faced several more difficulties in implementing the code itself, most particular with the energy calculations and dynamic programming. Finally, we faced another issue in dealing with Texture Synthesis, since the original paper doesn’t provide much detail on how this was done and merely referenced other papers. The next slide breaks down the most significant issues into more details and explains how we resolved and/or mitigated each of them.

Project Development

Problem Descriptions and Resolutions

- General Interpretation/Understanding of Algorithm
    - One of our most significant issues was simply general interpretation and understanding of the algorithm presented in the paper. While most equations were provided and although we both have fairly strong backgrounds in linear algebra, we faced a lot of issues and confusion just trying to understand the algorithm, particularly when interpreting some of the more obscure mathematical notations that we had not seen before. While we don't consider the algorithm or topics presented in the paper to be much more or less complicated than that of the seam carving paper from earlier in the semester, for that project, we at least had the benefit of bouncing ideas and questions off of ~500 other students and the TAs. On the other hand, for this project, we were never able to contact any other group working on our specific project to help resolve some of our misunderstandings. In the end, we were able to overcome our confusion by walking through each equation, step by step, and working out the equation in pseudocode on a markerboard until it made sense and we felt we had properly interpreted it.
- Weights
    - Equation 2 in the paper, the energy term for coherence constraints, is shown below. In the paper, it is mentioned that k_s and k_i are relative weights and that values of 50 and 2, respectively, were used for their experiments. However, no further detail is provided to give insight as to how those values were determined and whether they will be sufficient for any image set. We experimented with several different values for each, ranging from 0.001 to 100, but in the end settled on setting them to the values used in the paper.

$$E_1(x_i) = k_s \cdot E_S(x_i) + k_i \cdot E_I(x_i).$$

- Energy Minimization
    - We initially struggled with correct interpretation of the (4) energy equations, particularly in understand which point or patch should be used where and how to compute the sum of the normalized squared differences. As previously mentioned above, we resolved this by talking through each equation and converting it to pseudocode on a markerboard.

Project Development (Continued)

Problem Descriptions and Resolutions (Continued)

- Edge Cases
  - In this context, “edge” cases actually means edge cases. In some cases, the most ideal patch to fill a patch in the unknown region, was a source/sample patch that was on the edge of the image, meaning that the source patch was not the same size as the patch it was supposed to fill, resulting in exceptions being thrown in the code. To remedy this, we simply just did not use any source patches that were on the edge of the image, instead taking the “second-best” patch.
- Belief Propagation
  - According to the paper, Belief propagation “is a probability inference algorithm” that “has become popular lately in machine learning and computer vision.” In the context of the paper, Belief Propagation was used for any complex scenes or scenes with intersecting curves since its complexity is $O(2LN^2)$ versus $O(LN^{2+k})$ for Dynamic Programming, resulting in run times of a few seconds in cases where using Dynamic Programming might take hours. Although we initially welcomed the idea of exploring an increasingly popular and exciting algorithm, we had a lot of trouble translating the Belief Propagation equations outlined in the paper into code since it required several unique data structures and a paradigm that was difficult for us to understand. In the end, we abandoned it and reverted to Dynamic Programming for the following reasons:
    - We were already working inside an abbreviated timeline and were spending far too much time on Belief Propagation
    - Nearly all of our image sets contained non-intersecting curves, and as specified by the paper, only image sets with intersecting curves used Belief Propagation
    - Dynamic Programming is a concept we previously touched on in class and thus, we thought it might be more relevant to focus our efforts on it instead
- Dynamic Programming
  - While we previously implemented Dynamic Programming for the Seam Carving project, we could not simply use the same algorithm. This project necessitated its own flavor of Dynamic Programming, which came with its own unique set of challenges, especially in terms of figuring out what the loop structure should look like, where each energy equation should be called, and how to both fill and backtrace through the energy matrix to find the optimal patches.

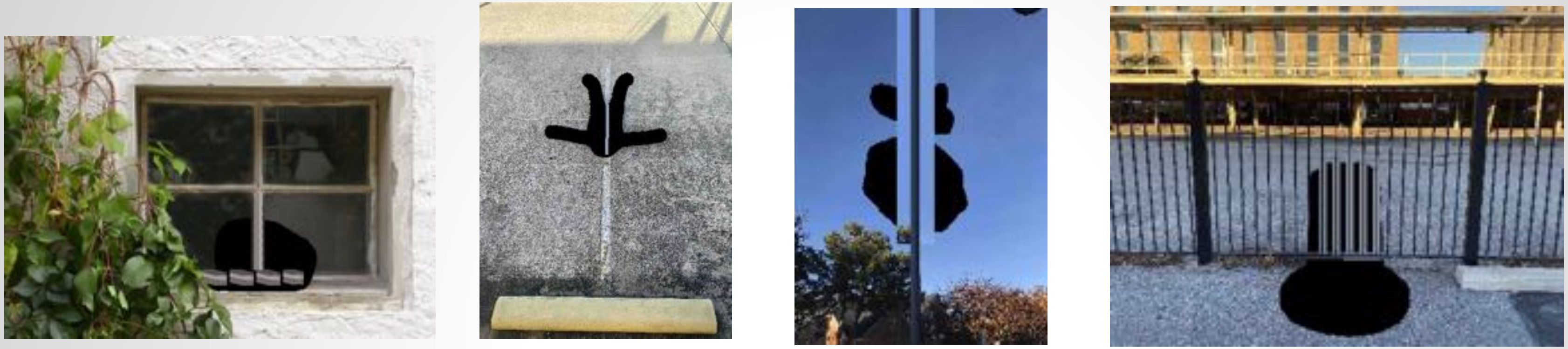
Project Development (Continued)

Problem Descriptions and Resolutions (Continued)

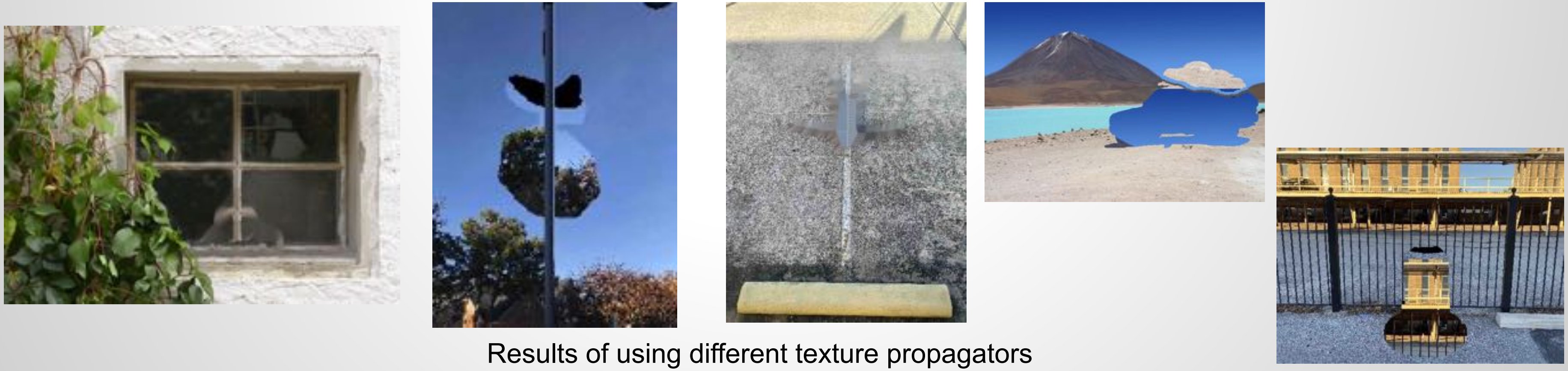
- Texture Synthesis
  - This is detailed above in the “Project Pipeline” section, but essentially the authors of the original paper focused the majority of the content in the paper on Structure Propagation, and then only briefly mention how they used various Texture Propagation algorithms from other papers to achieve Texture Synthesis after all structures have been propagated. We decided not to implement the algorithms in the (3) papers that are referenced in Section 4.1 of the original paper since we assumed it would be out of scope for the project. Alternatively, we developed our own set of Texture Synthesis algorithms, which will be further details in the “Code Functional Description” slides.
- Significant User Input/Manual Manipulation
  - The original authors of the paper do not attempt to hide the fact that a great deal of user input/specification is needed for each image set. In addition to user-specified curves (structures), which have been previously mentioned, the authors also mention in their Results section (Section 5) that they manually specified many of the input parameters, such as patch size, for each image set. Furthermore, the authors also utilized user-specified rotations for each source patch for many of their image sets using a process that they refer to as Sample Transformation. More detail on Sample Transformations and why we did not allow it will follow on a later slide, but essentially the process allows the user to manually rotate each source patch (of which there may be hundreds or thousands) in the known region so that it best aligns with the user-specified curve segments. While the authors are explicit in listing many of the ways in which manual input is required, we found it is likely that they are still omitting some detail about other ways in which manual intervention was needed. For example, in the mountain image set (Figure 9 of the original paper), the authors mention that a patch size of 9 is used. However, upon closer inspection, it appears that a different patch size was likely used for each curve (structure) that they are propagating. In a nutshell, the amount of manual input by the user is quite significant. We spent a great deal of time and attention for each and every image set in order to generate our final results, just as we suspect the authors of the original paper had to.
- Runtime
  - Despite our best efforts to use vectorized solutions, our structure propagation algorithm takes much longer than we had hoped to run. In the end, we were able to get the runtime down from hours to ~5 minutes or less for each image set by downsampling the images and using relatively sparse sampling intervals.

Project Development (Continued)

Failed and Good Interim Results



Results of using different patch sizes



Results of using different texture propagators

Project Development (Continued)

Finished and Unfinished Work

While there are obviously some parts of the original paper that we did not implement, in general, we developed most of the pieces in the pipeline that is outlined. For Structure Propagation, our pipeline includes input parameter specification, anchor point determination, sample patch extraction, energy minimization via dynamic programming, and cost matrix backtracing to find the most optimal sample patch for each anchor point. Furthermore, for Texture Synthesis, we developed a few rudimentary texture propagation algorithms to fill in the remaining areas of the unknown region(s). Having said that, there are a few steps in the overall pipeline that we either omitted for one reason or another, including:

- Belief Propagation (and Loopy Belief Propagation)
  - Omitted due to algorithmic complexity and since it was not necessary for almost all of our input image sets
- Texture Propagation
  - Substituted for a rudimentary set of texture propagation algorithms since implementing the algorithms in the referenced papers seemed out-of-scope
- Photometric Correction
  - Omitted since it seemed unnecessary for many of image sets and also would have required implementing algorithms in other papers
- Sample transformation
  - Omitted since it would have allowed manual intervention at a level far greater than we wanted an “automated” solution to have

Project Development (Continued)

Future Considerations

If we were to repeat this project again in the future, there are several ways in which we would improve upon our results. First and foremost, we ideally would have chosen this project from the start so that we could have had more time to devote towards it. Beyond that, we would fully implement each of the unfinished steps that were mentioned in the previous slide: Belief Propagation, Texture Propagation, Photometric Correction, and Sample Transformation. We would also choose a more diverse set of images to further identify the types of images that work well with the algorithm versus the types of images that do not work well. Additionally, we would like to work on a more automated solution. The structure and unknown masks, for example, should be automatically generated, rather than requiring the user to manually generate masks. This could be accomplished via some sort of graphical user interface that allows the user to specify both the unknown region(s) and the structure curves. Furthermore, while the original authors were able to achieve incredible results, they did so with a great deal of manual manipulation that was refined for each particular image set. Ideally, many of the parameters, such as patch size, sampling interval, patch orientation, etc, could be automatically determined to minimize user involvement and save the user the time that it takes to generate an acceptable looking output image.

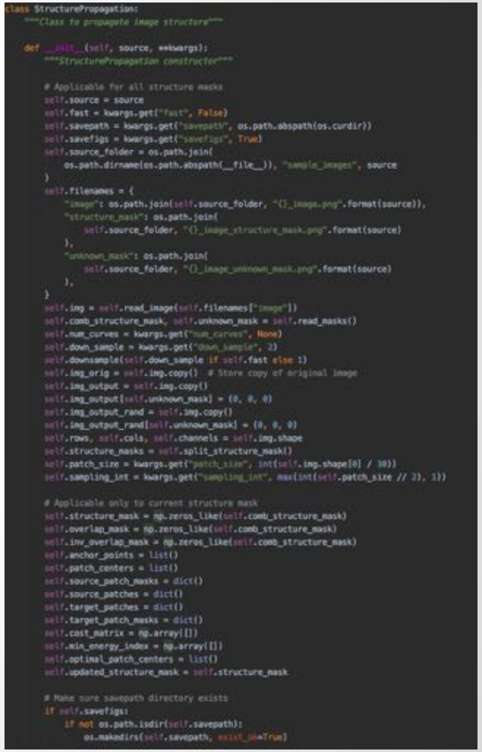
Computation: Code Functional Description

structprop.py

Our implementation of Structure Propagation is contained in the StructurePropagation class in **structprop.py**. StructurePropagation has several methods, but the most important ones are:

- __init__()
- split_structure_mask()
- get_overlap_mask()
- get_inv_overlap_mask()
- get_anchor_points()
- get_patch_centers()
- get_patches()
- get_cost_matrix()
- get_Es_point()
- get_Ei_point()
- get_E1_point()
- get_E2_point()
- get_norm_SSD()
- get_optimal_patches()
- apply_optimal_patches()
- run()

We will walk through each of these, starting with **__init__()**. In **__init__()**, the constructor for StructurePropagation, the image and masks are read in, the structure mask is split into multiple structure masks, and all data structures are initialized.

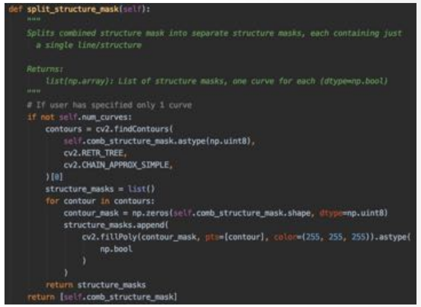


```python
class StructurePropagation:
    """Class to propagate image structure"""

    def __init__(self, source, **kwargs):
        """StructurePropagation constructor"""

        # Applicable for all structure masks
        self.source = source
        self.fast = kwargs.get("fast", False)
        self.savepath = kwargs.get("savepath", os.path.abspath(os.curdir))
        self.savefigs = kwargs.get("savefigs", True)
        self.source_folder = os.path.join(
            os.path.dirname(os.path.abspath(__file__)), "sample_images", source
        )
        self.filenames = {
            "image": os.path.join(self.source_folder, "{}_image.png".format(source)),
            "structure_mask": os.path.join(
                self.source_folder, "{}_image_structure_mask.png".format(source)
            ),
            "unknown_mask": os.path.join(
                self.source_folder, "{}_image_unknown_mask.png".format(source)
            ),
        }
        self.img = self.read_image(self.filenames["image"])
        self.comb_structure_mask, self.unknown_mask = self.read_masks()
        self.num_curves = kwargs.get("num_curves", None)
        self.down_sample = kwargs.get("down_sample", 2)
        self.downsample(self.down_sample if self.fast else 1)
        self.img_orig = self.img.copy()  # Store copy of original image
        self.img_output = self.img.copy()
        self.img_output[self.unknown_mask] = (0, 0, 0)
        self.img_output_rand = self.img.copy()
        self.img_output_rand[self.unknown_mask] = (0, 0, 0)
        self.rows, self.cols, self.channels = self.img.shape
        self.structure_masks = self.split_structure_mask()
        self.patch_size = kwargs.get("patch_size", int(self.img.shape[0] / 30))
        self.sampling_int = kwargs.get("sampling_int", max(int(self.patch_size // 2), 1))

        # Applicable only to current structure mask
        self.structure_mask = np.zeros_like(self.comb_structure_mask)
        self.overlap_mask = np.zeros_like(self.comb_structure_mask)
        self.inv_overlap_mask = np.zeros_like(self.comb_structure_mask)
        self.anchor_points = list()
        self.patch_centers = list()
        self.source_patch_masks = dict()
        self.source_patches = dict()
        self.target_patches = dict()
        self.target_patch_masks = dict()
        self.cost_matrix = np.array([[]])
        self.min_energy_index = np.array([[]])
        self.optimal_patch_centers = list()
        self.updated_structure_mask = self.structure_mask

        # Make sure savepath directory exists
        if self.savefigs:
            if not os.path.isdir(self.savepath):
                os.makedirs(self.savepath, exist_ok=True)
```

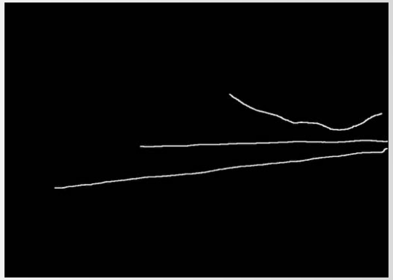
Computation: Code Functional Description (Continued)

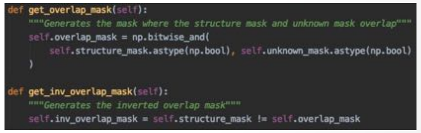
structprop.py (Continued)

The next major function is **split_structure_mask()**, which takes the “combined” structure mask, e.g. a mask that contains one or more curves, such as the one shown, and splits it up into separate masks, with each mask only having a single curve. This is done so that we can process a single structure/curve at a time and we implemented the splitting of the mask using *cv2.findContours* and *cv2.fillPoly*, which work together to translate each curve into separate contours, which can then be extracted to form separate masks. The next two major functions are **get_overlap_mask()** and **get_inv_overlap_mask()**. As the names suggest, these two functions are responsible for generating the mask where the structure mask (where a structure mask contains only a single structure) and unknown mask overlap and the mask where they do not overlap, respectively. Next, we have **get_anchor_points()** and **get_patch_centers()**.
get_anchor_points() sparsely samples the structure curve in the unknown region, to generate a set of anchor points, which will later be replaced with sample patches from the known region.
get_patch_centers() is responsible for generating the sample set, which contains the centers for all patches that are within a narrow band along the sample structure curve, outside the unknown region. Then, we have **get_patches()**, which generates the source/sample patch boolean masks for the patches in the known region, the target patch boolean masks for the patches in the unknown region, and the source patches themselves (so that we don’t have to recompute them

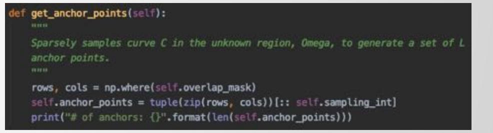


```python
def split_structure_mask(self):
    """
    Splits combined structure mask into separate structure masks, each containing just
      a single line/structure

    Returns:
        list(np.array): List of structure masks, one curve for each (dtype=np.bool)
    """
    # If user has specified only 1 curve
    if not self.num_curves:
        contours = cv2.findContours(
            self.comb_structure_mask.astype(np.uint8),
            cv2.RETR_TREE,
            cv2.CHAIN_APPROX_SIMPLE,
        )[0]
        structure_masks = list()
        for contour in contours:
            contour_mask = np.zeros(self.comb_structure_mask.shape, dtype=np.uint8)
            structure_masks.append(
                cv2.fillPoly(contour_mask, pts=[contour], color=(255, 255, 255)).astype(
                    np.bool
                )
            )
        return structure_masks
    return [self.comb_structure_mask]
```

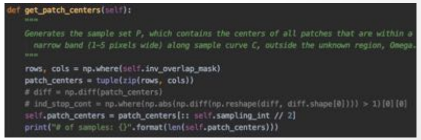




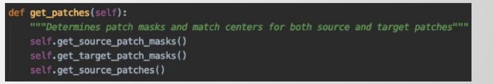
```python
def get_overlap_mask(self):
    """Generates the mask where the structure mask and unknown mask overlap"""
    self.overlap_mask = np.bitwise_and(
        self.structure_mask.astype(np.bool), self.unknown_mask.astype(np.bool)
    )

def get_inv_overlap_mask(self):
    """Generates the inverted overlap mask"""
    self.inv_overlap_mask = self.structure_mask != self.overlap_mask
```



```python
def get_anchor_points(self):
    """
    Sparsely samples curve C in the unknown region, Omega, to generate a set of L
    anchor points.
    """
    rows, cols = np.where(self.overlap_mask)
    self.anchor_points = tuple(zip(rows, cols))[:: self.sampling_int]
    print("# of anchors: {}".format(len(self.anchor_points)))
```



```python
def get_patch_centers(self):
    """
    Generates the sample set P, which contains the centers of all patches that are within a
      narrow band (1-5 pixels wide) along sample curve C, outside the unknown region, Omega.
    """
    rows, cols = np.where(self.inv_overlap_mask)
    patch_centers = tuple(zip(rows, cols))
    # diff = np.diff(patch_centers)
    # ind_stop_cont = np.where(np.abs(np.diff(np.reshape(diff, diff.shape[0]))) > 1)[0][0]
    self.patch_centers = patch_centers[:: self.sampling_int // 2]
    print("# of samples: {}".format(len(self.patch_centers)))
```

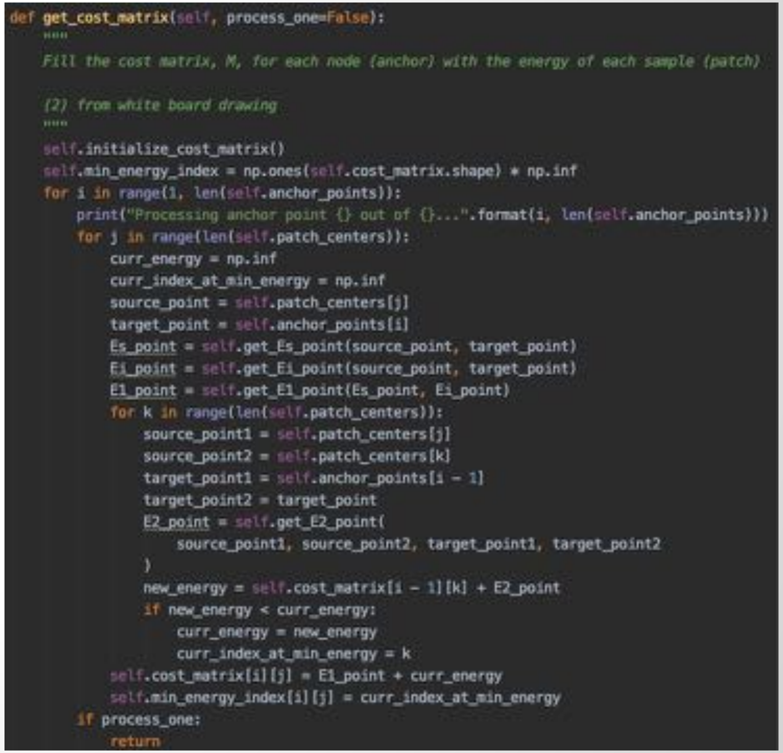


```python
def get_patches(self):
    """Determines patch masks and match centers for both source and target patches"""
    self.get_source_patch_masks()
    self.get_target_patch_masks()
    self.get_source_patches()
```

Computation: Code Functional Description (Continued)

structprop.py (Continued)

The next major function is **get_cost_matrix()**, and is perhaps one of the most import methods on the StructurePropagation class. **get_cost_matrix()** is responsible for using dynamic programming to populate the cost matrix and thus, essentially determine how good of a match each sample patch is for each anchor point. We start by looping through each anchor point. Then, we loop through each sample patch points (from the known region) and compute the energy terms Es, Ei, and E1. Next, there's another loop through the sample patch points to determine the value for E2. Finally, the energy information, along with information about the corresponding index, are stored in separate matrices.



```python
def get_cost_matrix(self, process_one=False):
    """
    Fill the cost matrix, M, for each node (anchor) with the energy of each sample (patch)

    (2) from white board drawing
    """
    self.initialize_cost_matrix()
    self.min_energy_index = np.ones(self.cost_matrix.shape) * np.inf
    for i in range(1, len(self.anchor_points)):
        print("Processing anchor point {} out of {}...".format(i, len(self.anchor_points)))
        for j in range(len(self.patch_centers)):
            curr_energy = np.inf
            curr_index_at_min_energy = np.inf
            source_point = self.patch_centers[j]
            target_point = self.anchor_points[i]
            Es_point = self.get_Es_point(source_point, target_point)
            Ei_point = self.get_Ei_point(source_point, target_point)
            E1_point = self.get_E1_point(Es_point, Ei_point)
            for k in range(len(self.patch_centers)):
                source_point1 = self.patch_centers[j]
                source_point2 = self.patch_centers[k]
                target_point1 = self.anchor_points[i - 1]
                target_point2 = target_point
                E2_point = self.get_E2_point(
                    source_point1, source_point2, target_point1, target_point2
                )
                new_energy = self.cost_matrix[i - 1][k] + E2_point
                if new_energy < curr_energy:
                    curr_energy = new_energy
                    curr_index_at_min_energy = k
            self.cost_matrix[i][j] = E1_point + curr_energy
            self.min_energy_index[i][j] = curr_index_at_min_energy
        if process_one:
            return
```

Computation: Code Functional Description (Continued)

structprop.py (Continued)

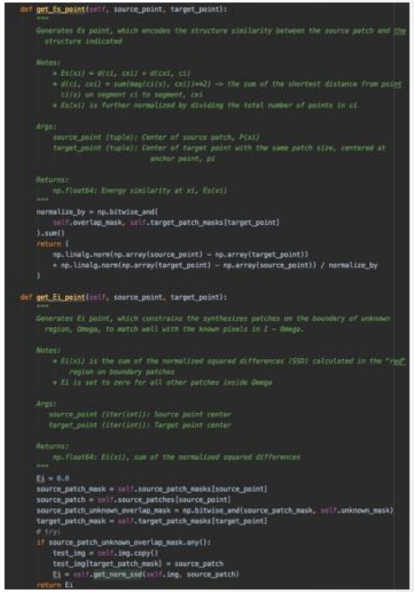
The next major set of functions to explain are those for determining the energy terms, Es, Ei, E1, and E2. Es, computed in **get_Es_point()**, encodes the structure similarity between the source patch from the known region and the structure point for a given anchor, and can be calculated as shown below, where *d* is the distance between two points, ci is the anchor point (unknown region), and cxi is the source point (known region):

$$E_S(x_i) = d(c_i, c_{x_i}) + d(c_{x_i}, c_i),$$

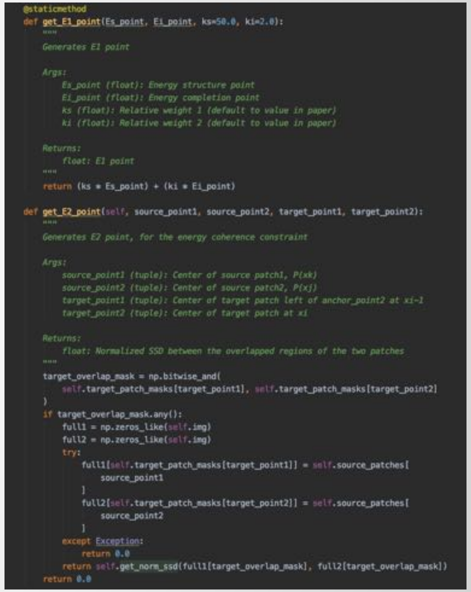
Next, we have Ei, computed in **get_Ei_point()**, which constrains the synthesized patches on the boundary of the unknown region and is equal to the sum of the normalized squared differences (SSD) calculated in the overlapping region between the image and the source/sample patch from the known region. Then, we have **get_E1_point()**, which uses the previously mentioned relative weights and can be calculated as:

$$E_1(x_i) = k_s \cdot E_S(x_i) + k_i \cdot E_I(x_i).$$

Finally, we also have E2, computed in **get_E2_point()**, which encodes the coherence constraint between two adjacent synthesized patches. It is computed as the normalized SSD of the overlapped region between two adjacent synthesized patches.



```python
def get_Es_point(self, source_point, target_point):
    """
    Generates Es point, which encodes the structure similarity between the source patch and the
      structure indicated

    Notes:
        * Es(xi) = d(ci, cxi) + d(cxi, ci)
        * d(ci, cxi) = sum(mag(ci(s), cxi))**2) -> the sum of the shortest distance from point
          ci(s) on segment ci to segment, cxi
        * Es(xi) is further normalized by dividing the total number of points in ci

    Args:
        source_point (tuple): Center of source patch, P(xi)
        target_point (tuple): Center of target point with the same patch size, centered at
                              anchor point, pi

    Returns:
        np.float64: Energy similarity at xi, Es(xi)
    """
    normalize_by = np.bitwise_and(
        self.overlap_mask, self.target_patch_masks[target_point]
    ).sum()
    return (
        np.linalg.norm(np.array(source_point) - np.array(target_point))
        + np.linalg.norm(np.array(target_point) - np.array(source_point)) / normalize_by
    )

def get_Ei_point(self, source_point, target_point):
    """
    Generates Ei point, which constrains the synthesizes patches on the boundary of unknown
      region, Omega, to match well with the known pixels in I - Omega.

    Notes:
        * Ei(xi) is the sum of the normalized squared differences (SSD) calculated in the "ref"
            region on boundary patches
        * Ei is set to zero for all other patches inside Omega

    Args:
       source_point (iter(int)): Source point center
       target_point (iter(int)): Target point center

    Returns:
        np.float64: Ei(xi), sum of the normalized squared differences
    """
    Ei = 0.0
    source_patch_mask = self.source_patch_masks[source_point]
    source_patch = self.source_patches[source_point]
    source_patch_unknown_overlap_mask = np.bitwise_and(source_patch_mask, self.unknown_mask)
    target_patch_mask = self.target_patch_masks[target_point]
    # try:
    if source_patch_unknown_overlap_mask.any():
        test_img = self.img.copy()
        test_img[target_patch_mask] = source_patch
        Ei = self.get_norm_ssd(self.img, source_patch)
    return Ei
```

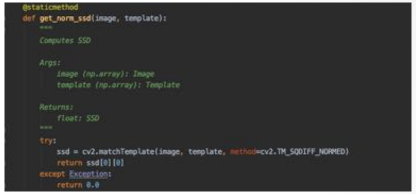


```python
@staticmethod
def get_E1_point(Es_point, Ei_point, ks=50.0, ki=2.0):
    """
    Generates E1 point

    Args:
        Es_point (float): Energy structure point
        Ei_point (float): Energy completion point
        ks (float): Relative weight 1 (default to value in paper)
        ki (float): Relative weight 2 (default to value in paper)

    Returns:
        float: E1 point
    """
    return (ks * Es_point) + (ki * Ei_point)

def get_E2_point(self, source_point1, source_point2, target_point1, target_point2):
    """
    Generates E2 point, for the energy coherence constraint

    Args:
        source_point1 (tuple): Center of source patch1, P(xk)
        source_point2 (tuple): Center of source patch2, P(xj)
        target_point1 (tuple): Center of target patch left of anchor_point2 at xi-1
        target_point2 (tuple): Center of target patch at xi

    Returns:
        float: Normalized SSD between the overlapped regions of the two patches
    """
    target_overlap_mask = np.bitwise_and(
        self.target_patch_masks[target_point1], self.target_patch_masks[target_point2]
    )
    if target_overlap_mask.any():
        full1 = np.zeros_like(self.img)
        full2 = np.zeros_like(self.img)
        try:
            full1[self.target_patch_masks[target_point1]] = self.source_patches[
                source_point1
            ]
            full2[self.target_patch_masks[target_point2]] = self.source_patches[
                source_point2
            ]
        except Exception:
            return 0.0
        return self.get_norm_ssd(full1[target_overlap_mask], full2[target_overlap_mask])
    return 0.0
```

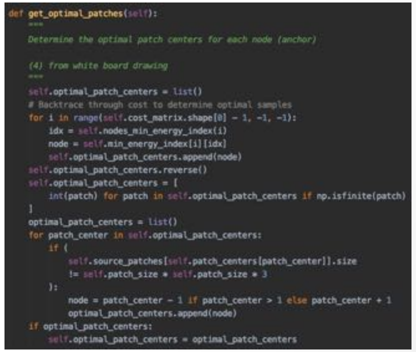
Computation: Code Functional Description (Continued)

structprop.py (Continued)

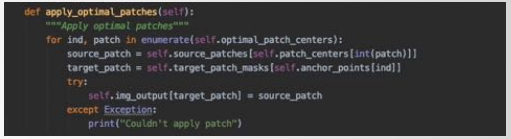
As mentioned on the previous slide, we are computing a normalized SSD in the calculations for both Ei and E2. We compute the normalized SSD using *cv2.matchTemplate* in **get_norm_ssd()**. Next, we have **get_optimal_patches()**, which contains the logic to backtrace through the cost matrix and identify the optimal patch for each anchor point in the unknown region. Then, **apply_optimal_patches()** is then used to actually apply each synthesized patch to the unknown region. Last, but not least, **run()**, is the top-level method on the StructurePropagation class that ties everything together in order to run the whole Structure Propagation pipeline.



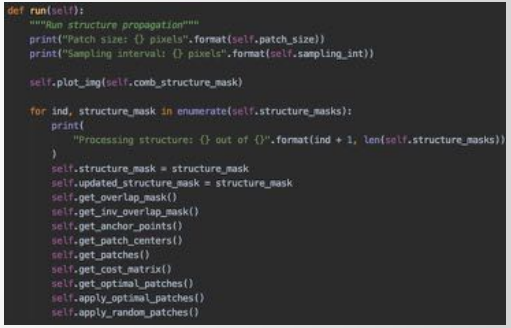
```
    @staticmethod
    def get_norm_ssd(image, template):
        """
        Computes SSD

        Args:
            image (np.array): Image
            template (np.array): Template

        Returns:
            float: SSD
        """
        try:
            ssd = cv2.matchTemplate(image, template, method=cv2.TM_SQDIFF_NORMED)
            return ssd[0][0]
        except Exception:
            return 0.0
```



```
def get_optimal_patches(self):
    """
    Determine the optimal patch centers for each node (anchor)

    (4) from white board drawing
    """
    self.optimal_patch_centers = list()
    # Backtrace through cost to determine optimal samples
    for i in range(self.cost_matrix.shape[0] - 1, -1, -1):
        idx = self.nodes_min_energy_index(i)
        node = self.min_energy_index[i][idx]
        self.optimal_patch_centers.append(node)
    self.optimal_patch_centers.reverse()
    self.optimal_patch_centers = [
        int(patch) for patch in self.optimal_patch_centers if np.isfinite(patch)
    ]
    optimal_patch_centers = list()
    for patch_center in self.optimal_patch_centers:
        if (
            self.source_patches[self.patch_centers[patch_center]].size
            != self.patch_size * self.patch_size * 3
        ):
            node = patch_center - 1 if patch_center > 1 else patch_center + 1
            optimal_patch_centers.append(node)
    if optimal_patch_centers:
        self.optimal_patch_centers = optimal_patch_centers
```



```
    def apply_optimal_patches(self):
        """Apply optimal patches"""
        for ind, patch in enumerate(self.optimal_patch_centers):
            source_patch = self.source_patches[self.patch_centers[int(patch)]]
            target_patch = self.target_patch_masks[self.anchor_points[ind]]
            try:
                self.img_output[target_patch] = source_patch
            except Exception:
                print("Couldn't apply patch")
```



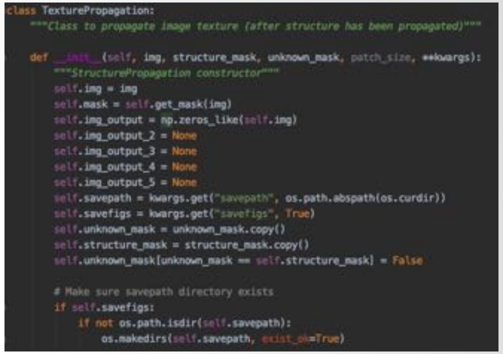
```
def run(self):
    """Run structure propagation"""
    print("Patch size: {} pixels".format(self.patch_size))
    print("Sampling interval: {} pixels".format(self.sampling_int))

    self.plot_img(self.comb_structure_mask)

    for ind, structure_mask in enumerate(self.structure_masks):
        print(
            "Processing structure: {} out of {}".format(ind + 1, len(self.structure_masks))
        )
        self.structure_mask = structure_mask
        self.updated_structure_mask = structure_mask
        self.get_overlap_mask()
        self.get_inv_overlap_mask()
        self.get_anchor_points()
        self.get_patch_centers()
        self.get_patches()
        self.get_cost_matrix()
        self.get_optimal_patches()
        self.apply_optimal_patches()
        self.apply_random_patches()
```

Computation: Code Functional Description (Continued)

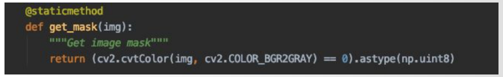
textprop.py

Our implementation of Texture Propagation is contained in the TexturePropagation class in **textprop.py**. TexturePropagation has several methods, but the most important ones are **__init__()**, **fill_with_source_texture()**, and **run()**.

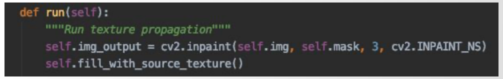
We will walk through each of these, starting with **__init__()**. In **__init__()**, the constructor for TexturePropagation, the image and masks are read in, and all data structures are initialized. **get_mask()** converts an image that has the structure propagated, but large areas of unknown region(s) remaining, and outputs a boolean mask to denote what needs to be filled in. **run()** is the top-level method which calls *cv2.inpaint* and **fill_with_source_texture()** to do the actual structure propagation. *cv2.inpaint* is an OpenCV function that lets you restore a specified region in an image by using the region neighborhood, which made it a good candidate to use to fill in unknown region(s) in our images for texture propagation purposes. Lastly, **fill_with_source_texture()**, is a method that we implemented at the last minute to achieve an alternative, albeit rudimentary, means for texture propagation. In **fill_with_source_texture()**, rather than trying to do any complex image inpainting, it simply fills in the unknown region(s) of the image with pixels to the left of, right of, above, or below the unknown region(s). In most cases, it generates results that are not terrific. However, in cases where the background of the scene is fairly consistent/constant and does not have a lot of variation, such as in our parking lot example in our results slides, it generates results that look much better than we could achieve by relying on *cv2.inpaint*.



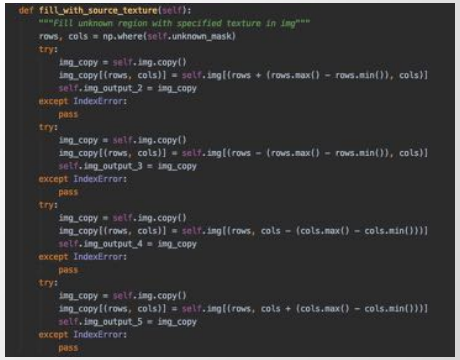
```python
class TexturePropagation:
    """Class to propagate image texture (after structure has been propagated)"""

    def __init__(self, img, structure_mask, unknown_mask, patch_size, **kwargs):
        """StructurePropagation constructor"""
        self.img = img
        self.mask = self.get_mask(img)
        self.img_output = np.zeros_like(self.img)
        self.img_output_2 = None
        self.img_output_3 = None
        self.img_output_4 = None
        self.img_output_5 = None
        self.savepath = kwargs.get("savepath", os.path.abspath(os.curdir))
        self.savefigs = kwargs.get("savefigs", True)
        self.unknown_mask = unknown_mask.copy()
        self.structure_mask = structure_mask.copy()
        self.unknown_mask[unknown_mask == self.structure_mask] = False

        # Make sure savepath directory exists
        if self.savefigs:
            if not os.path.isdir(self.savepath):
                os.makedirs(self.savepath, exist_ok=True)
```



```python
    @staticmethod
    def get_mask(img):
        """Get image mask"""
        return (cv2.cvtColor(img, cv2.COLOR_BGR2GRAY) == 0).astype(np.uint8)
```



```python
    def run(self):
        """Run texture propagation"""
        self.img_output = cv2.inpaint(self.img, self.mask, 3, cv2.INPAINT_NS)
        self.fill_with_source_texture()
```



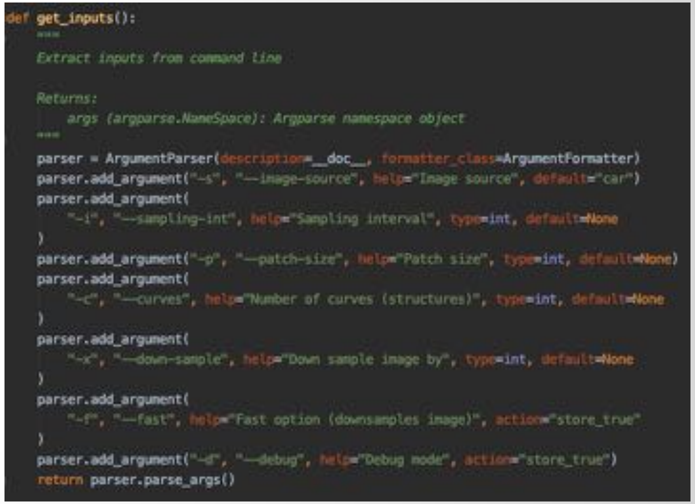
```python
    def fill_with_source_texture(self):
        """Fill unknown region with specified texture in img"""
        rows, cols = np.where(self.unknown_mask)
        try:
            img_copy = self.img.copy()
            img_copy[(rows, cols)] = self.img[(rows + (rows.max() - rows.min()), cols)]
            self.img_output_2 = img_copy
        except IndexError:
            pass
        try:
            img_copy = self.img.copy()
            img_copy[(rows, cols)] = self.img[(rows - (rows.max() - rows.min()), cols)]
            self.img_output_3 = img_copy
        except IndexError:
            pass
        try:
            img_copy = self.img.copy()
            img_copy[(rows, cols)] = self.img[(rows, cols - (cols.max() - cols.min()))]
            self.img_output_4 = img_copy
        except IndexError:
            pass
        try:
            img_copy = self.img.copy()
            img_copy[(rows, cols)] = self.img[(rows, cols + (cols.max() - cols.min()))]
            self.img_output_5 = img_copy
        except IndexError:
            pass
```

Computation: Code Functional Description (Continued)

main.py

Last, but not least, we have **main.py**. As the name suggests, **main.py** is the top-level script to launch the whole process, including both Structure Propagation and Texture Synthesis. The main input arguments are as follows:

- *--image-source* - Name of the image source, e.g. “car” or “pumpkin”
- *--sampling-int* - Interval at which to sample patches in the known region
- *--patch-size* - Size of the square patches to be extracted/synthesized (in pixels)
- *--curves* - Number of curves (structures) in the associated structure mask (this ended up not being necessary)
- *--down-sample* - Down sample the image by this input factor in both dimensions, e.g. 2 will downsample a 64x64 pixel image to 32x32
- *--fast* - Flag to turn on image downsampling
- *--debug* - Debug flag (ignore)



```python
def get_inputs():
    """
    Extract inputs from command line

    Returns:
        args (argparse.NameSpace): Argparse namespace object
    """
    parser = ArgumentParser(description=__doc__, formatter_class=ArgumentFormatter)
    parser.add_argument("-s", "--image-source", help="Image source", default="car")
    parser.add_argument(
        "-i", "--sampling-int", help="Sampling interval", type=int, default=None
    )
    parser.add_argument("-p", "--patch-size", help="Patch size", type=int, default=None)
    parser.add_argument(
        "-c", "--curves", help="Number of curves (structures)", type=int, default=None
    )
    parser.add_argument(
        "-x", "--down-sample", help="Down sample image by", type=int, default=None
    )
    parser.add_argument(
        "-f", "--fast", help="Fast option (downsamples image)", action="store_true"
    )
    parser.add_argument("-d", "--debug", help="Debug mode", action="store_true")
    return parser.parse_args()
```

Teamwork

Our original division of labor was to have one person tackle the Structure Propagation tasks, while the other person tackled the Texture Synthesis tasks. In reality, our actual division of labor was much different.

Since merely understanding the the Structure Propagation algorithm was a difficult task for just one person, we jointly worked everything out on a marker board and wrote code together via pair programming. Since that strategy worked pretty well and helped us each understand the content better than we could have individually, we tackled the Texture Synthesis pipeline in the same manner. After that, we alternated tasks quite a bit to make sure that no one person got stuck on a particular issue, but for the most part, Graham worked on mask development, code tweaking, and parameter tuning, while I focused on developing the report.

Overall, I'd say it was the division of labor was very equitable and I'd be happy to work with Graham again.

Lastly, I feel obligated to also note that we live in the same city, thus it was easy for us to physically meet up and work on the project together. This was a huge benefit for us and I imagine it would be much more difficult to work on a partner project remotely.

Resources

- **Building Edge Detection and Camera Pose Estimation Technical Papers**
  - https://arxiv.org/pdf/1601.07630v2.pdf
  - http://openaccess.thecvf.com/content_cvpr_workshops_2014/W03/papers/Chu_GPS_Refinement_and_2014_CVPR_paper.pdf
- **Google Maps and Google Street View**
  - Used for street level image and 3D building model image for original project (before it was abandoned)
- **Image Completion Technical Paper**
  - http://webee.technion.ac.il/people/cgm/Computer-Graphics-Multimedia/Undergraduate-Projects/2009/ImageCompletion/ImageCompletion_SIGGRAPH05.pdf
- **Piazza Posts**
  - In particular, credit to fellow student, Amrutha Govind, in Piazza post #871, for discovering the project and obtaining instructor approval
  - Also credit to Saeran Vasanthakumar, in Piazza post #871, for discovering and obtaining instructor approval for the project we originally attempted
- **Slack Posts**
- **Lectures**
  - Particularly ones that touched on energy minimization or dynamic programming
- **Past Assignment/Project Code**
  - While no code was explicitly copied from previous work, some ideas and design patterns were used from the Seam Carving code submitted by each of us for the Midterm Project
- **Stack Overflow**
  - Computing SSD in OpenCV
    - https://stackoverflow.com/questions/48799711/explain-difference-between-opencvs-template-matching-methods-in-non-mathematica
    - https://stackoverflow.com/questions/58158129/understanding-and-evaluating-template-matching-methods
  - Contour Filling
    - https://stackoverflow.com/questions/19222343/filling-contours-with-opencv-python
- **NumPy Documentation**
- **OpenCV Documentation**
  - https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_photo/py_inpainting/py_inpainting.html
  - https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_template_matching/py_template_matching.html
  - https://stackoverflow.com/questions/19222343/filling-contours-with-opencv-python
  - https://docs.opencv.org/2.4/modules/core/doc/drawing_functions.html

Appendix: Your Code

Code Language: Python, Bash (only for top-level main.sh script for instructors to use to make execution of all our sample cases easier)

List of code files:

- main.py
- structprop.py
- textprop.py
- Main.sh

Complete Set of Images Used in Report:
https://drive.google.com/drive/folders/1WFZ5DeEaenr81X_x448mbhs3h0FU-8aL?usp=sharing

Credits or Thanks

- Amrutha Govind, fellow student who discovered the Image Completion paper and got instructor approval for the project as a partner project
- Saeran Vasanthakumar, fellow student who discovered and got instructor approval for the project we originally attempted (Building Edge Detection and Camera Pose Estimation)