Computational Photography

Final Project

George Kudrayvtsev

CS6475 – Fall 2019

As outlined by the head TA on Piazza, I've approximated the report template using LaTeX instead of the provided slideshow. I tried to follow the same structure and ensured that the headings are consistent and easy to follow across pages.

A Tale of Two ~~Cities~~
Non-Photorealistic Rendering Techniques

Description: This report presents two independent non-photorealistic rendering (NPR) techniques: a "painterly" algorithm that simulates painting techniques like Claude Monet's "impressionist" style by emulating brush strokes on a canvas, and a "lowpoly" algorithm that emulates a modern digital art style of depicting a subject using minimal geometry.

Contents

I. Project Goals

Original Project Scope

The initial goal of the project was to take an image and stylize it to appear as though it was painted in an impressionist style, that is, as brush strokes on a canvas. There were some ideas for expanding scope to make the "brush strokes" look more realistic, such as clipping strokes at edges. I also planned to apply the stylization to videos and ultimately compose it all into a simple web-app that allowed a user to "draw" on the original image and get the impressionist effect where they drew.

What motivated you to do this project?

As someone who lacks any deep artistic talent aside from basic sketching, it was appealing to me to be able to emulate the unique and creative styles of capable and famous artists.

II. Scope Changes

Did you run into issues that required you to change your project scope?

I did not run into any issues, but rather a change of heart. The "painterly" papers I decided to implement ([4, 2] which are based on [8]) turned out to be rather simplistic, so I figured an increase in scope was necessary to meet the expectations of the final project.

Give a detailed explanation of what changed.

Most of the Litwinowicz paper [8] is implemented; contrary to my proposal, I did not implement brush stroke orientation coherence for video, nor did I implement gradient interpolation. Since I implemented optic flow algorithms both in *Computer Vision* and in *Educational Technology*, it felt disingenuous to claim it as novel work for this course.

After reading TA feedback to others' proposals, it was evident that the web-app portion would not be valuable from a grading perspective. I added a feature to the final script that will let you "draw" the result after it has been processed[1] just for fun, but there is no web version.

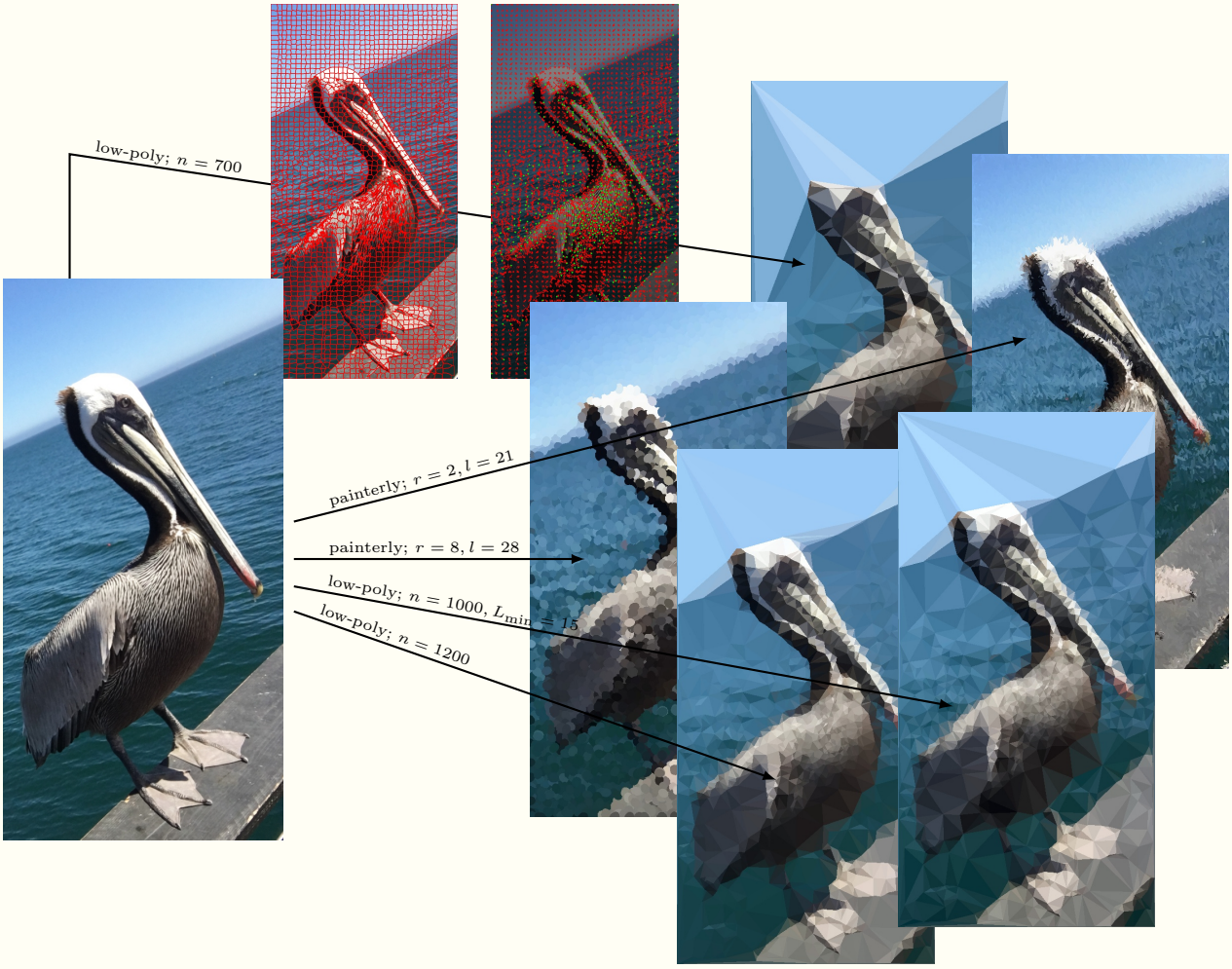
The painterly paper by Hertzmann [6] builds on [8], adding improvements like iterative refinement of brush strokes and spline-based stroke paths. It was a logical extension to the project, but I was instead inspired by the uniqueness of results and decided to explore another NPR technique rather than extend the current one.

> Specifically, I implemented a separate algorithm that emulates a low-polygon art style by triangulating the top N most important vertices in an image's superpixel corners [9, 5, 1]. These two, independent NPR techniques were then composed together to allow a user to choose between them from a single program and generate novel images easily:
>
> ```
> usage: main.py image {monet,lowpoly} ...
> ```

The work involved in this new paper ended up being *far* greater than that of the painterly paper, and I think it sufficiently covers my bases for scope expansion.

[1] If you don't pass `--save` or `--visualize`, you'll get a pop-up at the end that will let you "draw to reveal."

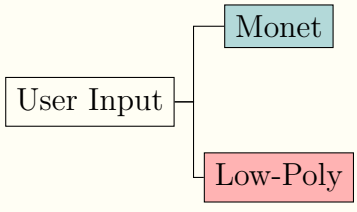
III. Showcase



The photo of this birb was taken on Santa Monica Pier in Los Angeles, CA; it's actually part of a selfie, if you can believe that I got that close...

IV. Project Pipeline

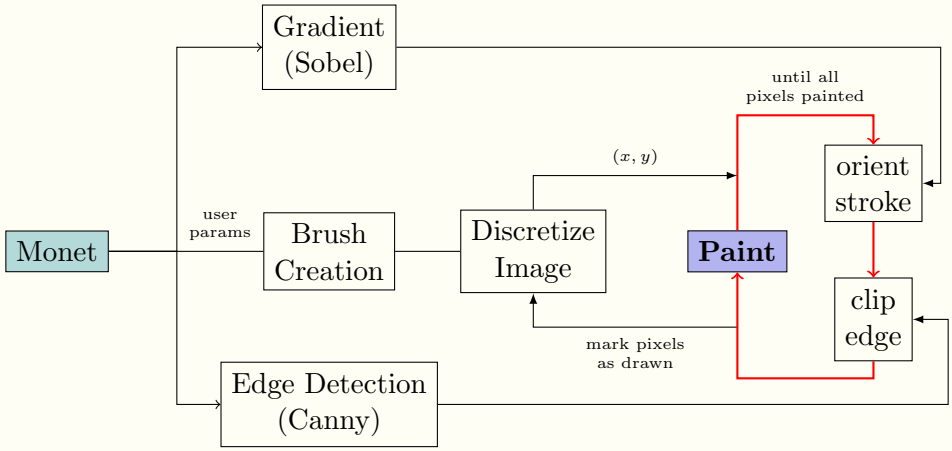
The two implementations are completely independent, joined by a script that parses user input and passes things along to the relevant pipeline. Note that both pipelines are completely automatic, though they each have unique configurable parameters.



IV. Project Pipeline (cont.)

Monet: Simulating Brush Strokes

At a very high level, the idea is simple: emulate brush strokes by blotting a region's average color in the shape of a brush at random locations.



The idea of specifically emulating impressionism is merely based on the type of brush parameters. A brush is composed of three metrics: its *radius*, r, its *length*, l, and its *orientation*, θ. These three combine to create a boolean mask. The suite also allows two stroke "shapes": rectangular and elliptical. This is the **brush creation** stage in the pipeline.

Given the set of parameters, we can **discretize the image** and approximate how many "strokes" it will take to replace all of the pixels. This is achieved with a mask that is "blotted" the same way the image is: we choose random non-blotted pixels as the next position for a brush stroke. In the following image we see the next "block" in the mask (left) partially-filled with strokes and the corresponding painting (right):

We **repeat the painting process** until all pixels are filled. For each unfilled pixel, we calculate its gradient direction (using OpenCV's `cv.Sobel`), then place the **oriented brush** stroke at that position. The brush stroke is **clipped off at edges** (assisted with `cv.floodFill`): the areas in the brush mask that are separated from the center by edges are ignored.

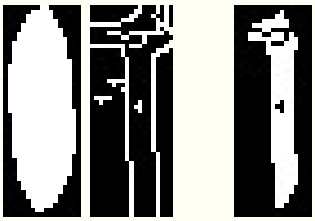
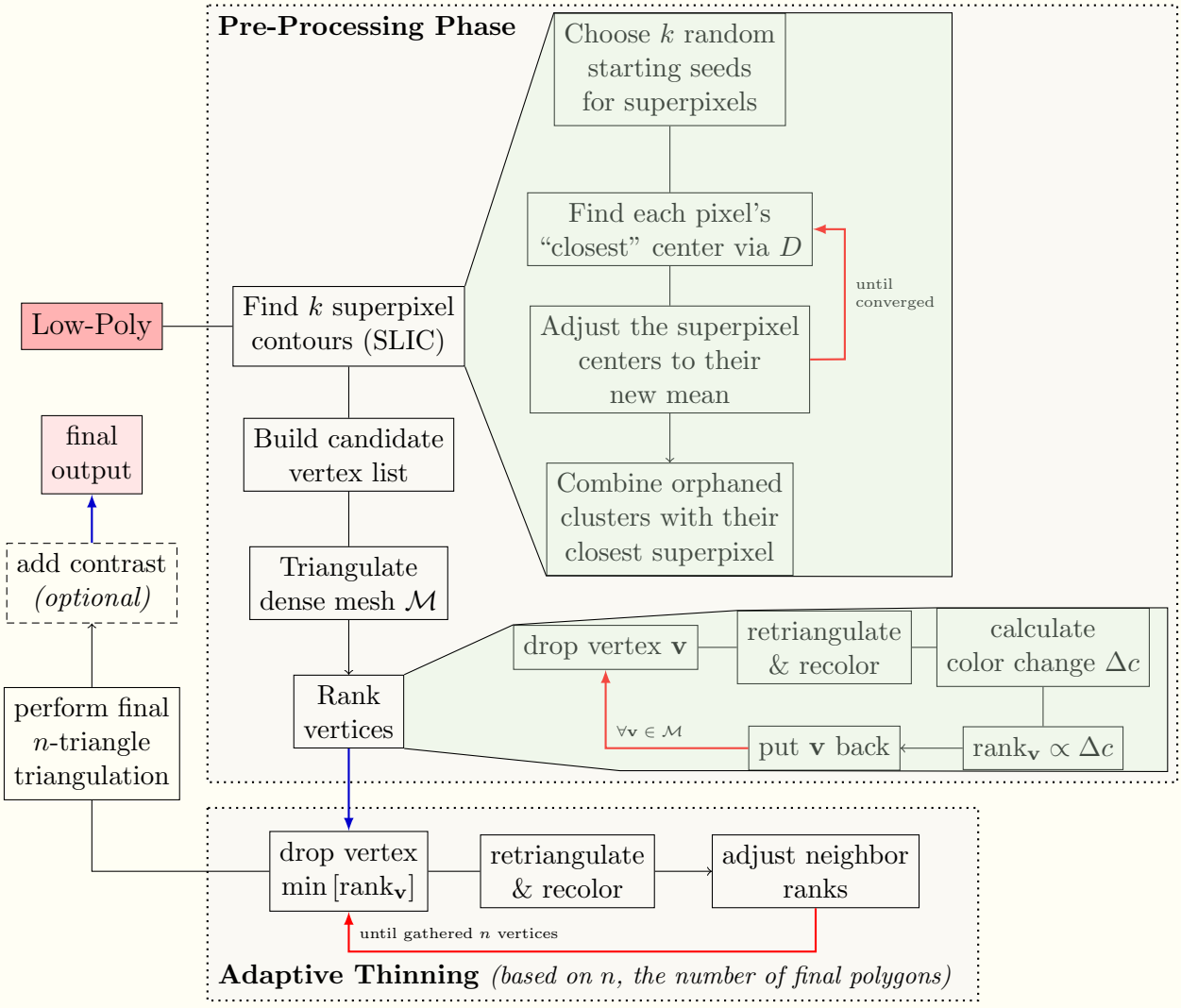


Figure 1: A visualization of how the brush mask (left) is clipped by edges (middle) after a flood fill.

IV. Project Pipeline (cont.)

Low-Poly: Minimal Triangles

This pipeline is far more complex, though the principles are simple: find the n most important pixels in the image and triangulate them.



We begin by discretizing the image into $k = 3000$ **superpixels using SLIC** [1] to determine the **initial vertex candidates** (which are the superpixel corners, determined via `cv.cornerHarris`). The image is then triangulated into a **high-resolution mesh** (assisted by `scipy.spatial.Delaunay`), $\mathcal{M}$, which we treat as our approximation of the original image. Then, we **rank the vertices** based on their importance to the image by determining how "wrong" the image would look if that vertex was dropped from the mesh, where "wrong" measures the total difference in color from the original image. Finally, we **choose the best** n **vertices** (where "best" means lowest-rank, and n is user-defined) and **triangulate** the final low-poly image. Note that the ranks of neighboring vertices change as we drop each vertex, so we need to adjust accordingly as we drop. This prevents us from building a list of viable vertices in advance. Optionally, we **balance the contrast** using Equation 1.

V. Demonstration: Result Sets

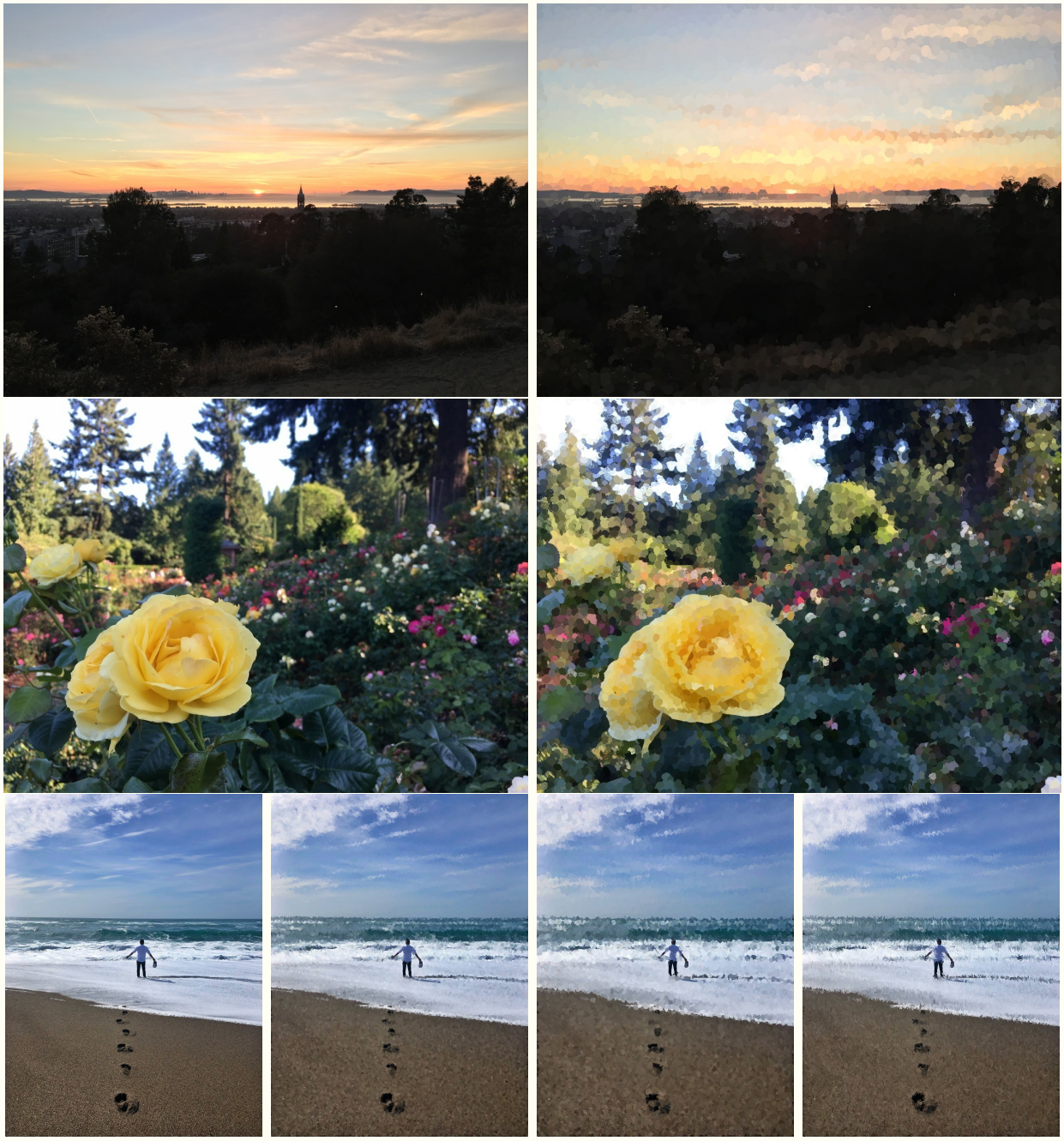


Figure 2: A series of impressionist renderings. **Top row**: an impressionist rendering of a photo I took of the sunset in the Berkeley hills; you can see a bit of San Francisco in the distance and the campanile on the U.C. Berkeley campus. Notice how its edges are well-preserved and not smudged by brush strokes. **Middle row**: a photo I took at the Rose Garden in Portland, Oregon earlier this year; the small flowers make ideal subjects for single brush strokes. **Bottom row**: this photo, taken on a random beach along the coast of northern California after getting lost on the way to Point Reyes (featured in Figure 6, later) is composed with a variety of brushes to demonstrate their artistic impact. **Parameters**: in the same order as before, the brush is $(r = 10, l = 23)$ and Canny thresholds are $(50, 150)$; then, we have $(r = 6, l = 17)$ and thresholds $(150, 400)$. Finally, from left to right on the last row, the we demonstrate brushes with fine circular strokes $(r = 4, l = 13)$, coarse circular strokes $(6, 21)$, and long skinny strokes $(2, 21)$ with edge thresholds $(350, 450)$.

V. Demonstration: Result Sets (cont.)

Figure 3: This series of the Space Needle in Seattle, WA (feat. glass art from the Chihuli Garden and Glass Museum) demonstrates how changing the triangle count ($n = 550, 1100, 2200$) affects the final render. Notice how the triangles cluster in areas of increasing importance even from a human perspective: we see the glass curls get more defined and start seeing the individual ridges in the needle rim, while the sky only gets a few dozen triangles at best. The bottom-right image is the "high-resolution" dense mesh that we use to approximate the full image and rank vertex importance (here, $n \approx 44000$, which is still a far cry from the $1024 \times 1365 \approx 1.4$ million pixels in the photo).

Note: we set $L_{\min} = 10$ for these results, which allows more distinction of the colors in the subject at the cost of the sky looking a little worse.

Figure 4: A low-poly ($n \in \{50, 150, 250, 450\}$) rendering of my avatar (a pixel-art drawing I made years ago of a bird in the *Halcyon* genus), demonstrating that the algorithm is still pretty effective on small (300×300) images.

V. Demonstration: Result Sets (cont.)

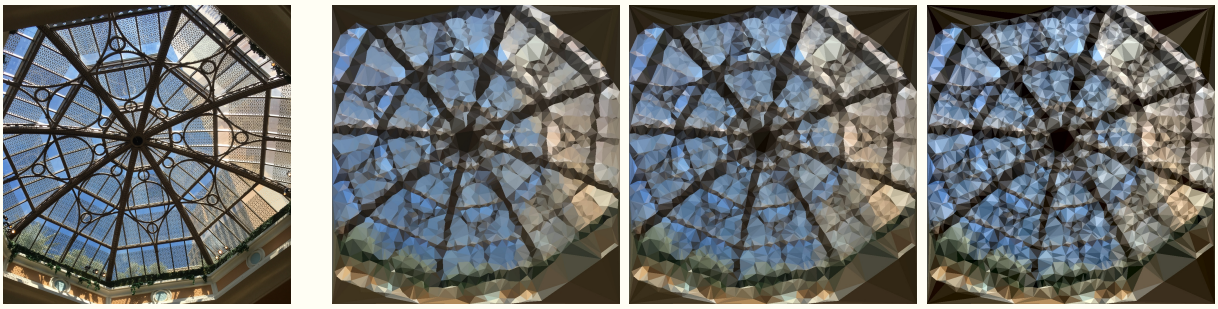


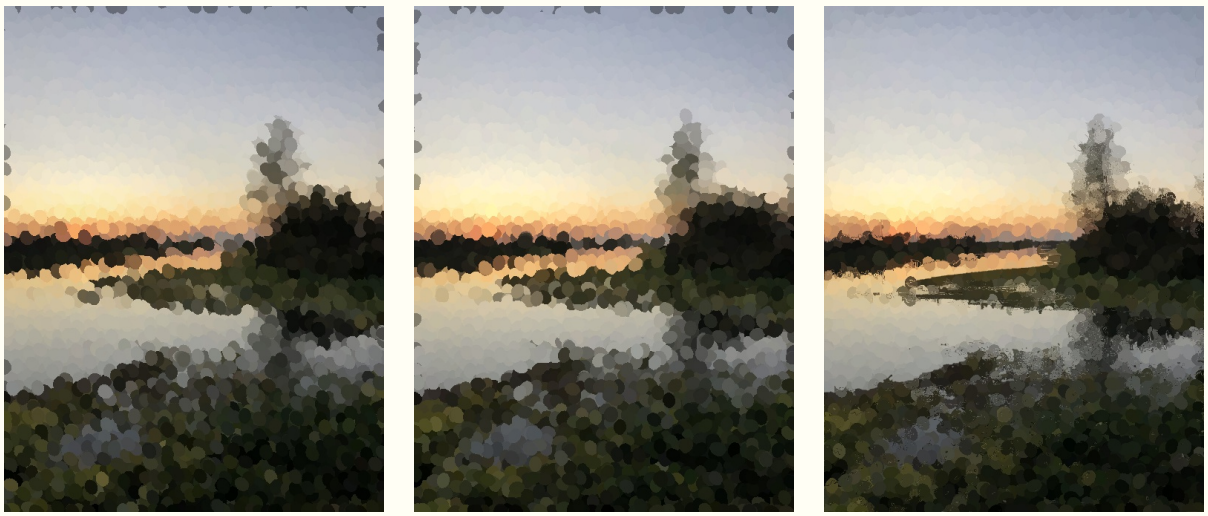
Figure 5: Here, we shatter the glass ceiling of the Sacramento Capitol building to demonstrate the power of Equation 1 with L_{min} set to 0 (the normal result), then 20 and 50 respectively with $n = 2000$. Note that triangulation is deterministic, so all that changes here is the contrast between neighboring triangles.

Figure 6: A series of photos that look kinda neat in either style. **Top row**: a photo of some flowers near the beach in Santa Barbara, CA I took last summer. We render with the brush $(r = 2, l = 27; 400, 550)$ and we use $n = 2500$ triangles. **Bottom row**: in this snapshot from the series *Photos Taken Seconds Before Disaster*, the waves crash onto the rocks at Point Reyes, CA. The brush parameters are $(8, 25; 200, 300)$ and $n = 3000$ triangles.

VI. Project Development

Painterly: Narrative, Progress, & Process

The narrative of progress on painterly-style NPR follows the literature [8, 4, 2] closely. Aside from edge clipping, all of the algorithms were fairly straightforward to implement.



(a) A single brush stroke shape and orientation.

(b) The stroke orientation is determined by the gradient angle.

(c) Brush strokes are clipped off at detected edges.

Figure 7: Progression of the painterly algorithm over time. In all of the images, the seed was fixed to ensure consistent brush stroke placement. A large brush stroke was chosen intentionally to enhance differences rather than aesthetic.

At a chosen position, take the average color within a 5×5 window and "draw" a brush stroke there with the chosen parameters (radius and length, (r, l)) and orientation θ (rotated to match the gradient angle, or hard-coded). If specified, the brush stroke is clipped away at edges. OpenCV's functionality is used for calculating the gradient (`cv.Sobel`) as well as for finding the edges (`cv.Canny`). The gradient angle is then simply $\theta = \arctan\left(\frac{\nabla I_y}{\nabla I_x}\right)$ and determines the stroke orientation.

Finally, in all cases, brush strokes were perturbed by "human error," with the orientation being perturbed by $\mathcal{N}(\mu = 0, \sigma = 20°)$ (Gaussian), the BGR color values by $[-8, +8]$ (uniform, out of 255), and the overall color intensity by $[-0.025, +0.025]$ (uniform, out of 1.0).

Since the brush stroke is just a mask, it was relatively trivial to add support here for both rectangular and elliptical brushes at the end (right).

Figure 8: Demonstrating the visual difference between rectangular (left) and elliptical (right) brush strokes.

VI. Project Development (cont.)

Difficulty: Discretization — One important development that took some time was implementing an efficient discretization of the image. Initially, it was essentially just N random samples of image locations; for a sufficiently-large N, the probability that all pixels were painted was really high. This is obviously not ideal from a performance perspective, nor is it a guarantee that every pixel *will* be painted.

Instead, every brush stroke happened twice: once on the actual "canvas," and once on a boolean mask that hence tracks which pixels have been drawn on. During discretization, then, a random undrawn pixel was chosen from the mask. This proved to *still* be too slow, since both the `np.where` and `np.random.choice` calls had a lot of choices to pick through. A further improvement was chunking the image into equal-sized rectangles and painting within them one at a time. The smaller pool of options meant a faster stroke position choice, while still emulating a semblance of randomness.

Difficulty: Edge Clipping — Clipping strokes at edges was quite a challenge. In hindsight, flood filling from the center of the brush stroke is the ideal solution, but it was a journey and a half getting there. Neither a weird approach that found the minimum index of an edge in each column nor connected components did the job.

Note that tweaking the Canny edge parameters has significant effect on the output: the more aggressive the thresholding, the more "blobby" the outputs.

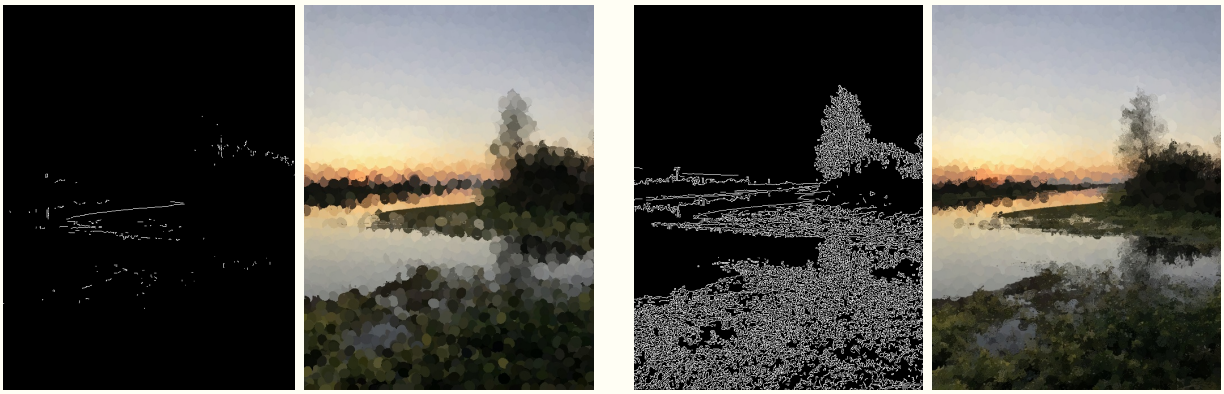


Figure 9: With lenient edge thresholds (right), we get heavy brush stroke clipping, to the point at which the greenery at the bottom is nearly untouched.

Unfinished Business

Edge clipping is still imperfect. The edges are ignored completely, meaning they preserve their color from the original image rather than the average color of the brush stroke. Given more time, a custom flood-filling algorithm to override this behavior would improve the resulting image quality, especially in photos with many fine edges.

As outlined earlier in the discussion of Scope Changes, a big improvement that is missing is gradient interpolation in low-magnitude areas. Both Hertzmann and Litwinowicz discuss interpolation the gradient in these areas, but the simpler approach employed here just chooses a random brush angle where the gradient magnitude is $\leq \pm 5$.

VI. Project Development (cont.)

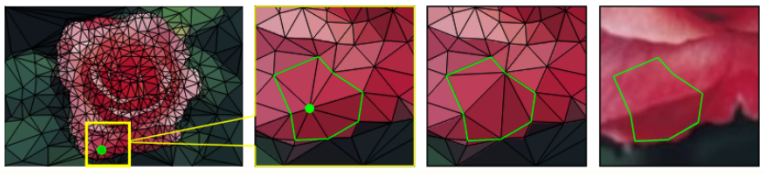
Low-Poly: Narrative, Progress, & Process

The story of emulating the minimalistic geometric style of low-poly artists is a little more interesting. Note that this paper was implemented in its entirety; the only thing technically missing is the real-time performance and interactive adjustment of the parameters.

The parts of the algorithm are pretty discrete, with each piece building on the next but staying independent. Implementing SLIC was fairly straightforward, as the paper is easy to read and it truly is a simple yet effective superpixelation algorithm [1]. The idea of a weight to balance between spatial and color clustering of superpixels is quite elegant. The only difficulty that arose was merging orphaned pixels with their neighboring superpixel. Since SLIC wasn't the main focus of the low-polygon algorithm, a hacky algorithm (ab)used `cv.connectedComponents` to merge each superpixel's smaller components with their closest superpixel center.

Each of the images in the bottom row of Figure 10 represents one phase of the pipeline: **superpixelation**, **vertex identification**, **dense mesh approximation**, and **ranking** (the green vertices). In terms of the actual triangulation of the vertices, `scipy.spatial.Delaunay` was relied upon heavily, but there is far more to the algorithms than the triangulation itself. Ranking vertices; identifying, drawing, and colorizing the individual polygons; and adjusting neighboring triangles and ranks all merely used the triangles as a means to an end. This doesn't even include the effort to do all of the aforementioned tasks optimally, without blowing up memory requirements or processing time. Finally, adding the contrast adjustment algorithm (demonstrated in Figure 5 earlier and described in further detail in Computation: Code Description) was the cherry on top.

Difficulty: Local Triangulation — The heart of the low-poly algorithm is the vertex ranking process, which involves dropping a vertex and retriangulating the neighborhood to see the effect on the color approximation. The following image from the paper [9, pg. 4] explains it well:



Notice that only the local vertices need to be retriangulated. The `Delaunay` object gives us both neighboring vertices (that is, which vertices touch a particular *vertex*) and triangles (that is, which triangles touch the edges a particular *triangle*), but does not give us the triangles of a vertex (that is, which triangles use a particular vertex). This has to be queried manually. Furthermore, the class does not allow for points to be removed from the mesh, so we must instead manipulate the list of triangles and vertices as we go. Note that during the thinning process (where we *actually* remove vertices), this also adjusts the *ranks* of the neighbors. All of this required careful consideration and was prefaced by pages upon pages of triangle drawings in a notebook.

VI. Project Development (cont.)

Figure 10: A low-poly rendering of Marilyn Monroe's portrait ($n = 750$). The bottom row gives a visual insight into the pipeline; from left-to-right we see: 3000 superpixels, the chosen candidate vertices in red and the top 750 vertices in green (notice the higher density at edges and other high-detail areas), and the high-resolution mesh from the initial red vertices. It is this mesh that provides the close approximation between a triangulation and the original image, and we determine the best vertices using this mesh as a baseline for pixel-wise color comparisons.

VI. Project Development (cont.)

Though there were a lot of odd results during developing the triangulation portion of adaptive thinning, two stand out: average color calculation, and rank sorting. In both of the below results, the images are ordered as (original, final algorithm, failure case).

Failure: Average Color — Both the source paper and its inspiration [5, 9] recommend using the median color in *Lab*-space (sorted by L) rather than the mean color to avoid the zig-zag effect. This had an... unexpected effect:

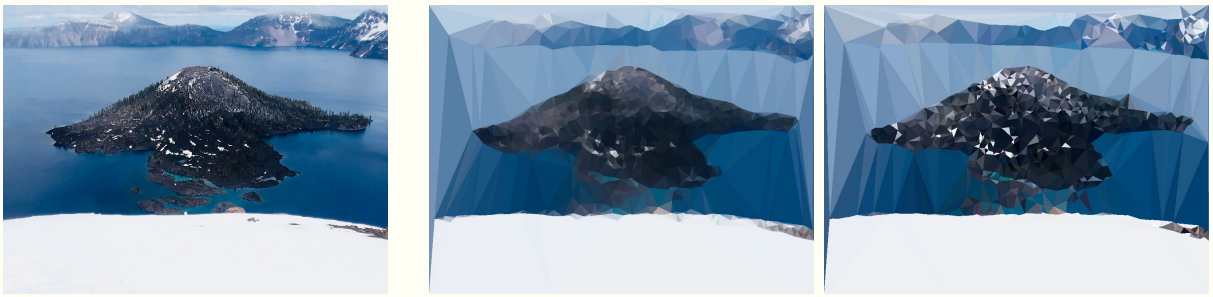


Figure 11: A photograph of Crater Lake, Oregon taken two summers ago.

Say what you will about the aesthetics: it definitely doesn't reduce the zig-zag effect.

Failure: Vertex Ranking — The paper states to choose the top-n vertices with the *minimum* rank. This seemed strange, since wouldn't you want to pick the vertices with the *highest* cost, since it suggests their importance? Well...

Things to Do Differently

As far as the paintings go, it would have been neat to add "height" to the brush strokes, much like described in [7]. Since many of the brush strokes overlap, it would have been possible to add lighting and depth to the strokes to create a further illusion of paint layers. Implementing the Phong illumination model, though, would've required considerably more effort and begun to bleed much more into the world of computer graphics. Adding brush textures (that is, something a little more complicated than ellipses or rectangles) would also be a neat, artistic improvement.

VI. Project Development (cont.)

Things to Do Differently (cont.)

For the low-poly implementation, the intent of the source work [9] is for the amount of triangles n to be adjustable in real time. The pre-processing phase takes some time, but the adaptive thinning is much faster. Since the implementation here does everything in sequence and is executed from the command line, there's not really any opportunity to allow interactivity. Furthermore, the lack of full control over the Delaunay triangulation means there is some overhead in the way the rank adjustment is implemented. Having the time to study and implement the triangulation, as well as sprinkling `np.load`/`save` around to preserve various deterministic results and avoid recalculations would have gone a long way towards reaching true real-time performance for changing n.

From an artistic standpoint, I wish I had figured out what types of photos make low-poly art look good... the results from the subjects in my photo sets is a bit of a mixed bag and it's honestly pretty hard to tell which of them actually look good (likely because of how long I've stared at the results at this point).

Finally, the system of equations for adjusting contrast could be improve to use the sparse matrix classes provided by Scipy (like `scipy.sparse.lil_matrix`) to have both better memory usage (most rows are full of zeroes) and faster (potentially even real-time!) contrast adjustment. However, wrapping my head around the way the indices are supposed to work was not worth it given the fact that things, well, worked as-is.

VII. Computation: Code Description

Painterly Computation

Creating the brush mask ended up being straightforward, both for ellipses (which have a native way to "rotate" them) and rectangles (for which `scipy.ndimage.rotate` fits the bill). With this and an optimized way to find the average color at each pixel:

```python
@staticmethod
def create_average_image(image):
    """ Calculates the average pixel color of every pixel.

    Set Brush.NEIGHBORS accordingly to change how big of a window is
    considered.
    """
    k = Brush.NEIGHBORS // 2
    kernel = np.ones((k + 1, k + 1), dtype=np.float32)
    kernel /= kernel.size

    average = cv.filter2D(image, -1, kernel)
    return Brush.simulate_human_error(average)
```

```python
class RectangularBrush(Brush):
    """ Represents a standard rectangular brush.

    Note that the `radius` still specifies *half* of the width to ensure
    compatibility with round brushes.
    """

    def make_mask(self):
        return self.rotate_mask(
            np.ones((self.length, self.radius * 2 + 1)),
            self.orientation
        ).astype(np.bool)

    @staticmethod
    def rotate_mask(mask, angle):
        # Rotate the ellipse to match our orientation.
        mask = ndimage.rotate(mask, angle, order=1)
        return Brush.force_odd_shape(mask)


class EllipticalBrush(Brush):
    """ Create an elliptical brush mask as specified by its parameters.
    """

    def make_mask(self):
        # Just "draw" an ellipse (thank u OpenCV):
        major_axis = int(max(self.length, self.radius * 2 + 1) * 1.5)
        img = np.zeros((major_axis, major_axis, 1))
        cx = cy = major_axis // 2
        cv.ellipse(img,
                   (cx, cy),                    # center
                   (self.length // 2, self.radius),    # half-axes
                   self.orientation,            # angle
                   0, 360,                      # arc length
                   255,                         # "color"
                   thickness=-1)

        return Brush.force_odd_shape(img.reshape(major_axis, major_axis))
```

We can actually get the baseline functionality working from this handful of snippets: pick random positions in the image and set the local region, masked by the brush, to the average color. Of course, there's far more to the algorithm than that.

```python
def stroke(self, target, center, averages, clip=None, src=None):
    """ Performs a "brush stroke" on the target image.

    The input should include an averaged image created via
    `Brush.create_average_image()` for performance.

    :param   target    image on which to draw the brush stroke
    :param   center    location at which to make the stroke
    :param   averages  average color w/in a neighborhood of each pixel
    :param   clip[=None] a mask (same size as target) that specifies
        clipping parameters (like edges) for brush strokes. Any brush stroke
        that has a clipping area will only fill the area that is allowed by
        the clip mask.
    :param   src[=None]  the "true" source image that we're processing.
        Including it will allow better color sampling for edge-clipped
        brush strokes.

    :returns   (dy, dx), mask
        the deltas from the center that were considered, as well as the
        final mask that was applied at that location
    """
    if src is None: src = target
    cx, cy = map(int, center)

    # Add a perturbation to the brush stroke angle.
    original_orientation = self.orientation
    self.orientation += np.random.normal(0, ANGLE_DELTA)
    mask = self.make_mask()
    self.orientation = original_orientation

    h, w = mask.shape
    dy, dx = h // 2, w // 2
    subimage = target[
        cy-dy : cy+dy+1,
        cx-dx : cx+dx+1
    ]

    color = averages[cy, cx]
```

```python
# We want edges to "stop" brush strokes, so we flood fill from the
# center to find stopping points, then AND that with the brush stroke
# itself to find the overlap.
if clip is not None:
    # +2 w/h because `floodFill` requires it
    subclip = clip[cy-dy-1 : cy+dy+2, cx-dx-1 : cx+dx+2]
    subclip_mask = subclip.copy()

    ccy, ccx = subimage.shape[0] // 2, subimage.shape[1] // 2
    mask[ccy, ccx] = True
    cv.floodFill(subclip[1:-1, 1:-1], subclip_mask, (ccx, ccy),
                 1, 0, 0, flags=4 | cv.FLOODFILL_MASK_ONLY | (42 << 8))

    subclip = (subclip_mask[1:-1, 1:-1] == 42)

    # If the clip mask changed anything, we should resample the color to
    # only include the pixels that we're "brushing" rather than the
    # entire region.
    if np.any((subclip & mask) != mask):
        n = self.NEIGHBORS // 2
        window = src[
            cy - n : cy + n + 1,
            cx - n : cx + n + 1,
        ]

        mask &= subclip
        color_chunk = mask[ccy-n:ccy+n+1, ccx-n:ccx+n+1]
        color = window[color_chunk].mean(axis=0)
        color = self.simulate_human_error(color)

subimage[mask] = color
return dy, dx, mask
```

Here is the core `Brush.stroke()` method in its entirety. In the left snippet, we add some noise to the brush stroke orientation, extract the neighborhood window of the target image at (c_x, c_y) and find its average color. In the right snippet, we optionally clip the mask using `cv.floodFill` with whatever is given to us in the `clip` variable (usually the Canny edges, as we'll see later). If any of the mask was clipped, we may need to re-adjust the color since the "average" from before no longer necessarily applied. The `Brush.simulate_human_error()` function just adds some noise as specified in the pipeline discussion.

VII. Computation: Code Description (cont.)

To elaborate on the flood filling, we do a fill from the *relative* center (cc_x, cc_y) (that is, relative to the sub-image extracted from the target at (c_x, c_y)), specifying a 4-connected fill, only on the boolean mask, with a value of 42 (note the `flags=`). Then, wherever the mask has the value 42 is where the fill happened.

```
marked_mask = np.logical_not(edges.copy())
def mask_generator(shape):
    submask = marked_mask[pad_y:-pad_y, pad_x:-pad_x]
    chunk_x, chunk_y = 0, 0
    chunk_w, chunk_h = 64, paintbrush.length * 2

    while chunk_y < h:
        chunk = submask[chunk_y : chunk_y+chunk_h, chunk_x : chunk_x+chunk_w]
        yy, xx = np.where(chunk)
        yy_len = len(yy)

        if yy_len <= 10:    # faster than sampling, and "random enough" by the end
            for i in range(yy_len):
                yield xx[i] + chunk_x, yy[i] + chunk_y
            yy_len = 0

        if not yy_len:
            chunk_x = min(w, chunk_x + chunk_w)
            if chunk_x >= w:
                chunk_y += chunk_h
                chunk_x = 0

            continue

        i = np.random.randint(0, yy_len)
        y, x = yy[i], xx[i]
        yield x + chunk_x, y + chunk_y

    raise StopIteration
```

Finally, we set the masked pixels in the original image to the computed average color and return the details. The returned `mask` variable is what helps us know what parts of the image have been "painted on," and which haven't (allowing a faster discretization).

In fact, that discretization can be seen on the right. The `marked_mask` tracks all areas that have been painted on, and we pull a random position from it as the next (c_x, c_y). We operate in $64 \times 2l$ chunks so that the calls are faster. Notice that the mask starts with the edges already marked, since we know that edges should be preserved.

```
# We keep an untouched version of the image around since we want colors to
# be sampled from the "true" image (if need be, since we typically sample
# from the averages) rather than the partially-painted version.
padded = result.copy()

for pix, (x, y) in enumerate(mask_generator(result.shape)):
    px, py = (x + pad_x, y + pad_y)

    angled_brush = paintbrush.copy(new_orient=orientations[py, px])
    src = padded if clip_edges else None
    clip = edges if clip_edges else None

    dy, dx, mask = angled_brush.stroke(result, (px, py), averages,
                                       src=src, clip=clip)

    marked_mask[py-dy:py+dy+1, px-dx:px+dx+1][mask] = False
```

Each of these pieces combines into the final loop, on the right. Obviously, plenty of code has been left out because of its simplicity: finding the gradient direction, parsing command-line parameters, adding padding to avoid out-of-bounds errors, etc.

This is the high-level composition of our functions: for every chosen pixel, orient a new brush based on the gradient orientation, perform the brush stroke, then mark all of those pixels as painted.

VII. Computation: Code Description (cont.)

Low-Poly Computation

This section will be far more dense and is broken down in the same way as before: we compose the individual pieces into the final process.

Superpixel Creation—SLIC

The SLIC algorithm [1] is implemented in `superpixelate()`. We first distributes the k superpixel "seeds" at equally-spaced locations in the $l \times w$ image, where the spacing depends on $S = \sqrt{wh/k}$. To avoid seeding the superpixels with an edge, we then "slide" the seeds to the lowest-gradient area in their 3×3 neighborhood:

```
height, width = image.shape[:2]
resolution = height * width
labimg = utils.cielab(image)

_, _, gradient = utils.gradient(utils.grayscale(image))
gradient = cv.copyMakeBorder(gradient, 1, 1, 1, 1, cv.BORDER_CONSTANT, value=np.inf)

# First, sample the image every S = sqrt(N / k) pixels to initialize the
# cluster centers. We choose the seed pixel where the gradient is lowest in
# a 3x3 window.
centers = []
S = np.sqrt(resolution / float(count))
x, y = S, S

for ix in np.arange(S - 1, width, S).astype(int):
    for iy in np.arange(S - 1, height, S).astype(int):
        grad_window = gradient[iy:iy+3, ix:ix+3]    # recall 1x1 padding

        # Get index of 2D matrix: https://stackoverflow.com/a/30180322
        row, col = np.divmod(grad_window.argmin(), grad_window.shape[1])
        iy += row - 1       # we want the *offset* relative to the center
        ix += col - 1

        l, a, b = labimg[iy, ix]
        centers.append([l, a, b, iy, ix])

centers = np.array(centers, dtype=np.float32)
```

Each superpixel seed is a tuple of a location (c_x, c_y) and its *Lab* color. We can now associate pixels with their centers: each pixel gets the label (a.k.a. row index) of the superpixel "closest" to it (defined soon).

In each iteration, we look in the $2S \times 2S$ window of each superpixel center. Since we expect each superpixel to be *roughly* $S \times S$, this doubled search area is sufficient. This reduced search area is the key to a fast convergence; other algorithms like k-means try to balance the entire image at once. Our complexity is $O(k)$ instead of $O(wh)$.

We find how "close" all of the pixels in the window in window are to the center, then update their labels and min-distances accordingly (note that `distances` starts as a matrix $\mathbf{D}_{h \times w} = \infty$).

```
distances = np.inf * np.ones((height, width), dtype=np.float64)
labels = -1 * np.ones_like(distances, dtype=int)

# Perform the segmentation by iteratively adjusting the centroids.
#
# The paper suggests that 10 iterations are sufficient, but we also track
# error (how much centers move each iteration) to break out early if we can.
error, threshold = np.inf, np.sqrt(count) / 2.
print("Clustering %d superpixels... (threshold=%0.1f)" % (count, threshold))

for i in range(10):
    print("  iteration %d: " % (i + 1), end="")
    for k, center in enumerate(centers):
        (y, x), s = center[-2:].astype(int), int(S)

        # Create a 2S x 2S region around the center to do the search.
        min_y, min_x = max(0, y - s), max(0, x - s)
        max_y, max_x = min(height, min_y + 2*s + 1), min(width, min_x + 2*s + 1)
        search = np.s_[min_y:max_y, min_x:max_x]

        dists = get_distance(labyx[search], center, S, spatial_weight=WEIGHT)

        # Wherever the distances are lower, set the label to this cluster center
        # and the distance accordingly.
        updates = dists < distances[search]
        distances[search][updates] = dists[updates]
        labels[search][updates] = k + 1

    # Update the cluster centers based on our adjustment.
    new_centers = np.empty(centers.shape)
    for k in range(centers.shape[0]):
        belonging = (labels == k + 1)
        if not np.any(belonging):
            new_centers[k] = centers[k]
            continue

        new_centers[k] = labyx[belonging].mean(axis=0)

    # Calculate the residual error
    error = np.linalg.norm(centers[:, 3:] - new_centers[:, 3:], axis=1).sum()
    centers = new_centers
    print("error = %0.1f" % error)
    if error < threshold: break
```

The paper recommends 10 iterations to converge the centers, but also suggests that an error threshold can work, where the total error for an iteration is how much the superpixel centers moved. We use the error threshold of $\sqrt{k}$; this often results in convergence in 6-8 iterations.

VII. Computation: Code Description (cont.)

The notion of "closeness" between pixels is key to the algorithm. Specifically, the authors introduce a weight, m, that balances between superpixels clustering based on spatial vs. color proximity. The formula is:

```python
def get_distance(area, center, window_scale, spatial_weight=10):
    """ Calculates the distance in CIELAB color space and positional space.
    """
    assert spatial_weight in range(1, 41), spatial_weight

    lab_a, yx_a = area[..., :3], area[..., 3:]
    lab_b, yx_b = center[:3], center[3:]

    dc = np.linalg.norm(lab_b - lab_a, axis=-1)
    ds = np.linalg.norm(yx_b - yx_a, axis=-1) / window_scale

    return np.sqrt((dc ** 2) + (ds ** 2) * (spatial_weight ** 2))
```

$$d = \sqrt{d_c^2 + \left(\frac{d_s}{S}\right)^2 m^2}$$

where d_c and d_s are the standard Euclidean distances in Lab-color and xy-space, respectively.

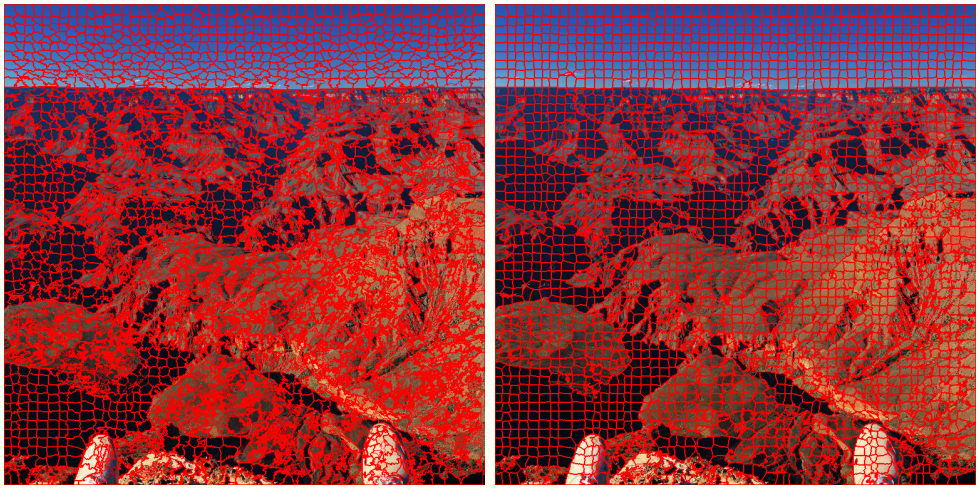


Figure 12: Here we see superpixelation with $m = 5$ (left) and $m = 35$ (right). The source photograph was taken at the Grand Canyon last summer (feat. my white shoes which are now tinted red).

The superpixels on the left have far more spatial flexibility, so they morph to maximize the color similarity. We used $m = 35$ for all results in this report.

Finally, we need to make sure that our superpixels are all contiguous chunks. Since we didn't enforce connectivity above, some pixels may be associated with a superpixel label but not actually within that superpixel's contour. To alleviate this, we merge each superpixel's smaller components with their closest superpixel center, using a bounding box to avoid a distance calculation with all k centroids.

```python
# We didn't enforce connectivity, so some pixels may have been orphaned away
# from their superpixel. The paper recommends using connected components to
# assign these pixels the label of their nearest cluster center.
#
# We do this more simply by merging orphaned pixels in a label with their
# neighbors.
for k, (y, x) in enumerate(centers[:, 3:]):
    label = k + 1
    truth = (labels == label).astype(np.uint8)

    # Find the largest component which will be the "main" one.
    cc_count, cc = cv.connectedComponents(truth, connectivity=8, ltype=cv.CV_16U)
    if cc_count == 1: continue      # empty superpixel...
    cc_main = np.argmax([np.count_nonzero(cc == ccn) for ccn in range(1, cc_count)]) + 1

    # Merge the others into their best neighbor.
    for to_merge in range(1, cc_count):
        if to_merge == cc_main: continue

        # Craft the bounding box around the component so we can see all of
        # its neighbors and decide on the "best" one.
        rect = cv.boundingRect((cc == to_merge).astype(np.uint8))
        tl_x, tl_y, wd, ht = rect

        subset = np.s_[tl_y:tl_y+ht+1, tl_x:tl_x+wd+1]
        candidates = np.unique(labels[subset][labels[subset] != label])

        cands = centers[candidates - 1]
        dists = get_distance(cands, centers[k], S, spatial_weight=WEIGHT)
        labels[subset][cc[subset] == to_merge] = candidates[dists.argmin()]

    # Recalculate the center since we've removed a bunch of orphan pixels.
    mean = labyx[labels == label].mean(axis=0)
    centers[k, :] = mean
```

At the end of this process, we have a list of centroids and a map of labels that associate each pixel with one of the k superpixels. Now we can determine our initial set of candidate vertices.

VII. Computation: Code Description (cont.)

Identifying Vertices

The paper [9] states that "superpixel corners" compose the initial set of vertices. This naturally leads to `cv.harrisCorners`. First, we draw the superpixel contours on a mask, then identify the corners. We then filter the corners to exclude any that are too close together, to not cluster *too* much at edges.

```
centers, labels = superpixelate(original,
    count=3000, spatial_proximity_weight=35)

# For our initial set of vertices, we "choose the intersections of several
# adjacent superpixels as the initial set of vertex candidates."
#
# We do this by drawing each superpixel's contour and then run Harris corner
# detection on that outline.
binmap = np.zeros((height, width), dtype=np.uint8)
superpixel_outline = 255 * np.ones((height, width, 1), dtype=np.uint8)

for x, y in centers:
    mask = (labels == labels[y, x])
    binmap[mask] = 1
    contours, heir = cv.findContours(binmap, cv.RETR_LIST, cv.CHAIN_APPROX_NONE)
    cv.drawContours(superpixel_outline, contours, -1, 0)
    binmap[mask] = 0

corners = cv.cornerHarris(superpixel_outline, 2, 3, 0.04)
too_small = corners < 0.1 * corners.max()
corners[too_small] = 0

# Only allow one corner within an 8-connected region.
considered = np.zeros_like(corners, dtype=np.bool)
for y, x in np.array(np.where(corners)).astype(int).T:
    if corners[y, x] and not np.any(considered[y-1:y+2, x-1:x+2]):
        considered[y, x] = True
corners[~considered] = 0
```

Note that this was the *final* solution to make it in, but definitely not the first. A number of other things were tried, including: sampling from the contours directly, weighing the samples based on how many contours each pixel takes part in, and a rudimentary corner identification via a set of simple 3×3 kernels. All of these resulted in a lackluster distribution of vertices.

Ranking Vertices

Note: From this point on, we are working exclusively in *Lab* color space.

The first step to determining a vertex's importance is to create baseline for error. This is the hyper-dense mesh we create from the *full* set of initial vertices (see the bottom-right image of Marilyn, Figure 10). For each triangle in this mesh, we calculate the total difference in color between the triangle's color (the mean color) and the same area in the original image.

```
# First, we find the local error of each triangle based on the average
# color.
dense_errors = np.empty(triangles.shape[0], dtype=np.uint32)
approx = np.zeros_like(labimg)

print("  rendering mesh")
for j, triangle in enumerate(triangles):
    color, search, mask = render_triangle(labimg, triangle)
    approx[search][mask] = color

# The triangles will overlap one another *WHEN RENDERED* because of
# per-pixel imprecisions, so it doesn't make sense to calculate the
# error until after they've ALL been rendered.
print("  calculating mesh error")
for j, triangle in enumerate(triangles):
    search, mask = get_polygon_mask(triangle)
    mesh_area = approx[search][mask]
    img_area = labimg[search][mask]

    dense_errors[j] = get_mesh_error(mesh_area, img_area)
```

The next part of the processing involves simulating the retriangulation that would occur if a given vertex was dropped. This would change the triangle colors, and thus have an effect on the overall error of the image. This error is a metric of a vertex's "rank," which will determine our choices for the final n-polygon render later.

VII. Computation: Code Description (cont.)

The process relies heavily on some abstractions to helper functions:

- `get_triangles()` finds *all* of the triangles that use a particular vertex
- `get_neighboring_vertices()` finds all of the *vertices* that share an edge with a particular vertex
- `reform_mesh()` identifies the image area and respective recolored triangles that are encompassed by a set of vertices; it composes two other helpers:
  - `get_polygon_mask()` determines a boolean mask that is covered by some vertices, essentially turning our continuous edges into discrete pixel locations, and
  - `render_triangle()` uses that information to find the median color of a triangle within an image
- `get_mesh_error()` simply calculates the sum of absolute differences in color between two image areas

```python
def get_neighboring_vertices(delaunay, vert_idx):
    """ Gets indices of vertices that touch the given vertex.

    Adapted from:
        https://docs.scipy.org/doc/scipy-0.14.0/reference/gener
        https://stackoverflow.com/a/23700182
    """
    indices, pointers = delaunay.vertex_neighbor_vertices
    return pointers[indices[vert_idx] : indices[vert_idx + 1]]
```

```python
def get_triangles(delaunay, vert_idx, mask=None):
    """ Finds indices of all of the triangles that contain a vertex (by index).
    """
    simplices = delaunay
    if isinstance(delaunay, scipy.spatial.Delaunay):
        simplices = delaunay.simplices
    rows = np.where(simplices == vert_idx)[0]
    if mask is None: return rows
    return rows[mask[rows]]
```

```python
def reform_mesh(image, vertices):
    """ Given a set of vertices, find the triangulated mesh.
    """
    search, mask = get_polygon_mask(vertices, modify=True)
    subimg = image[search]
    submesh = np.zeros_like(subimg)

    fake_triangulation = scipy.spatial.Delaunay(vertices)
    fake_triangles = vertices[fake_triangulation.simplices]
    for tri in fake_triangles:
        # Render each new triangle to a fake image area.
        color, s, m = render_triangle(subimg, tri)
        submesh[s][m] = color

    return submesh[mask], subimg[mask], fake_triangulation
```

```python
def get_polygon_mask(verts, modify=False):
    """ Creates a rectangular mask that has the `verts` polygon filled in.
    """
    tx, ty, wd, ht = cv.boundingRect(verts)
    mask = np.zeros((ht, wd), dtype=np.uint8)
    cv.fillPoly(mask, [verts - [tx, ty]], 1)
    if modify: verts -= [tx, ty]
    return np.s_[ty:ty+ht, tx:tx+wd], mask.astype(np.bool)

def render_triangle(image, triangle):
    """ Returns the average color of `triangle` in `image`.
    """
    search, mask = get_polygon_mask(triangle)
    img_area = image[search][mask]
    return pick_median_color(img_area), search, mask
```

Ultimately, these are all used as part of the ranking process:

```python
# Now we rank each vertex according to its "importance" to the image by
# "pretending to" remove it from the triangulation.
rank = np.empty(vcount, dtype=np.float32)
for idx, vert in enumerate(vertices):
    # Find all triangles that use this vertex and all of the verts that
    # make up those triangles.
    tri_indices = get_triangles(triangulation, idx)

    # If there is only one triangle for this vertex, it's on the edge of
    # the image and should be "infinitely" important (no dropping!).
    if tri_indices.shape[0] <= 1:
        rank[idx] = np.inf
        continue

    # This is the error before dropping the vertex.
    local_error = dense_errors[tri_indices].sum()

    #
    # Now drop the vertex and re-triangulate locally.
    #
    vert_indices = get_neighboring_vertices(triangulation, idx)
    verts = vertices[vert_indices]

    # Find the area within the original image that is encompassed by
    # these neighboring vertices.
    submesh, subimg, _ = reform_mesh(labimg, verts)

    # Note that we store a SIGNED result, since this retriangulation
    # might actually be better...
    new_error = get_mesh_error(submesh, subimg)
    rank[idx] = new_error - local_error
```

VII. Computation: Code Description (cont.)

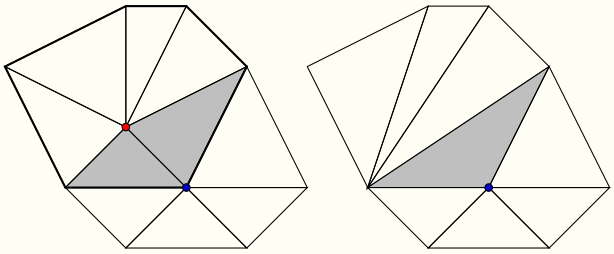
This concludes the preprocessing phase of the low-polygon pipeline. Beyond this point, the algorithm finally starts to depend on n, the user-specified number of triangles to render. It can be adjusted to get a different result relatively quickly.

Thinning Vertices

Every iteration, we pull the vertex with the lowest rank until we've extracted our n vertices. The problem is, though, that since dropping a vertex causes a local retriangulation, the ranks of the neighboring vertices change! Since this might affect the global "best" vertex rank, we need to make this adjustment as we go.

```python
def thin(self, count):
    """Renders an image with only the `count` best vertices.
    """
    vcount = self.vertices.shape[0]
    reformed = self.dense_mesh.simplices
    valid_tris = np.ones(reformed.shape[0], dtype=np.bool)
    queue = []

    print("Thinning out dense mesh:")
    for n in range(count):
        best_idx = self.vertices[:, 2].argmin()
        queue.append(best_idx)

        # This time, we ACTUALLY drop the vertex and retriangulate our mesh.
        tri_indices = get_triangles(reformed, best_idx, mask=valid_tris)
        neigh_indices = np.unique(reformed[tri_indices])
        neigh_indices = neigh_indices[neigh_indices != best_idx]
        neigh_verts = self.vertices[neigh_indices, :2].astype(np.int32).copy()

        submesh, subimg, retry = reform_mesh(self.image, neigh_verts)
        self.vertices[best_idx, 2] = np.inf

        retry.simplices = neigh_indices[retry.simplices]    # local -> global

        # Inject these new triangles back into our original set.
        tri_count = retry.simplices.shape[0]

        # Sometimes there might be more triangles, not less... in that case,
        # we need to extend the list of simplices rather than just replace.
        limit = tri_indices.shape[0]
        if tri_count > limit:
            extra = tri_count - limit
            reformed[tri_indices] = retry.simplices[:limit]
            reformed = np.vstack((reformed, np.zeros((extra, 3), dtype=np.int32)))
            reformed[-extra:] = retry.simplices[limit:]
            valid_tris = np.hstack((valid_tris, np.ones(extra, dtype=np.bool)))

        else:
            stay, gone = tri_indices[:tri_count], tri_indices[tri_count:]
            valid_tris[gone] = False
            reformed[stay] = retry.simplices

        # Now we need to adjust the local errors of the neighbors.
        for idx in neigh_indices:
            tri_indices = get_triangles(reformed, idx, mask=valid_tris)
            neigh_indices = np.unique(reformed[tri_indices])
            neigh_verts = self.vertices[neigh_indices, :2].astype(np.int32).copy()

            submesh, subimg, retry = reform_mesh(self.image, neigh_verts)
            self.vertices[idx, 2] = get_mesh_error(submesh, subimg)
```

This introduces quite a bit of complexity because we can't easily drop vertices from the Delaunay triangulation object. Hence, we modify a copy of the triangles instead: we keep a mask of valid triangles, replacing the ones that get dropped with a vertex with the new triangulation. This lets us minimize both memory and processing requirements. For adjusting neighbor ranks, we simply do the same error calculation from before. It's theoretically possible to do an adjustment directly, with no recalculation:



Notice how when we drop the red vertex, we can easily adjust the rank of the blue vertex, because the total change in error is the difference between the two old triangles and the new one. This optimization was deemed more effort than it was worth, though, processing-wise.

Once we've decided on our n vertices, we can go on and triangulate them for the final low-poly render. However, the paper offers an opportunity here to change the overall contrast of the image (we demonstrated this effect in Figure 5). This involves the introduction of a new parameter to represent the minimum contrast between neighboring triangles, L_{min}, and building and solving a system of equations with this parameter.

After solving for the new set of $L_{1..n}$ values for the triangles, we can finally render them and convert $Lab \longrightarrow BGR$ color space for display.

VII. Computation: Code Description (cont.)

Increasing Image Contrast

We measure contrast as the signed difference in the "light" component of each triangle in *Lab*-color space, making it *at least* some value $L_{\min}$:

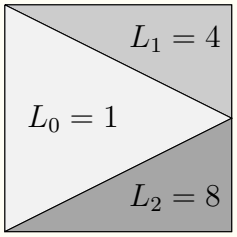
$$d'_{ij} = L_i - L_j$$
$$d_{ij} = \begin{cases} \max\left[d'_{ij}, L_{\min}\right] & \text{if } d_{ij} \geq 0 \\ \min\left[d'_{ij}, -L_{\min}\right] & \text{otherwise} \end{cases}$$

By default, we say $L_{\min} = 0$ (expressed with `--contrast=0` as a script parameter, though this is the default) because extra contrast is rarely a desired effect. Then, we can formulate a system of equations in which we balance the contrast as such:

$$\begin{cases} L'_i - L'_j = d_{ij} & \forall L_j \in \mathcal{N}(T_i) \\ L'_i = L_i & \forall L_i \in \mathcal{M}_n \end{cases} \tag{1}$$

where $\mathcal{N}(T_i)$ is short-hand for the (up to 3) neighbors of the i^{th} triangle, and $\mathcal{M}_n$ is the n-triangle low-poly mesh. Solving this overdetermined system (more equations than unknowns) results in a set of lighting values that increase the contrast of the image. We demonstrated this in Figure 5 earlier.

For clarity's sake, we work through an example. Suppose we have the extremely simple triangulation of an image on the right, where we've denoted the "lightness" value of each triangle's average color in *Lab*-space. With no adjustment (that is, $L_{\min} = 0$), our system can be formed as follows:



$$\begin{cases} L'_0 - L'_1 = d_{01} = -3 \\ L'_1 - L'_0 = d_{10} = 3 \\ L'_1 - L'_2 = d_{12} = -4 \\ L'_2 - L'_1 = d_{21} = 4 \\ L'_0 - L'_2 = d_{02} = -7 \\ L'_2 - L'_0 = d_{20} = 7 \end{cases} \quad \text{and} \quad \begin{cases} L'_0 = L_0 = 1 \\ L'_1 = L_1 = 4 \\ L'_2 = L_2 = 8 \end{cases}$$

This results in the $\mathbf{Ax} = \mathbf{b}$ system:

$$\begin{bmatrix} 1 & -1 & 0 \\ -1 & 1 & 0 \\ 0 & 1 & -1 \\ 0 & -1 & 1 \\ 1 & 0 & -1 \\ -1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} L'_0 \\ L'_1 \\ L'_2 \end{bmatrix} = \begin{bmatrix} -3 \\ 3 \\ -4 \\ 4 \\ -7 \\ 7 \\ 1 \\ 4 \\ 8 \end{bmatrix}$$

Unsurprisingly, it gives us the original L_0, L_1, L_2 values. Changing $L_{\min}$ will mean that there will not be an exact solution, so the least-squares approximation will change the upper values in $\mathbf{b}$ to cause a larger variance in lighting values. For example, setting $L_{\min} = 4$ only affects d_{10}, d_{01} and so:

$$\{L_1 \approx 0.71, L_2 \approx 4.3, L_3 = 8\}$$

as evaluated by Wolfram|Alpha here.

VIII. Additional Details

Reproducing Results

All of the input and output pairs that were generated for this report are available on OneDrive, here: https://1drv.ms/u/s!AvfgQ3Qt8rgWj6t88PqC-Mv2FmG0BQ?e=cySObK

For images with more than one variant on a style (like the pelican from the Showcase), the file suffix hints at the parameter(s). For example, `glass-lowpoly-2000-20.jpg` is generated with $n = 2000$ and $L_{\text{min}} = 20$:

```
python main.py images/glass.jpg lowpoly --triangles 2000 --contrast 20
```

To resolve any confusion, the `README.txt` defines all of the parameters and matches input and output filenames.

Note that this does *not* include images that demonstrate debugging output or failure cases, like Marilyn's vertices in Figure 10, the effect of m on superpixels in Figure 12, or the shattered girl in the rain (from Failure: Vertex Ranking) because the scripts can't really generate these automatically.

Program Usage

The NPR suite gives almost complete control over the rendering pipeline. Both the `lowpoly` and `monet` styles have plenty of things to configure.

The painterly-style "subprogram":

```
usage: main.py image monet [-h] [-b n n] [--brush-type {rectangle,ellipse}]
                           [--edges c c] [--angle theta] [--no-clip]

optional arguments:
  -h, --help            show this help message and exit
  -b n n, --brush n n   specifies the (length, radius) of a brush stroke
  --brush-type {rectangle,ellipse}
                        specifies the shape of a brush stroke
  --edges c c           specifies the (low, high) thresholds for Canny edge
                        detection
  --angle theta         forces a brush stroke angle (in degrees) instead of a
                        gradient-oriented stroke
  --no-clip             does not clip edges of brush strokes
```

And the low-poly subprogram:

```
usage: main.py image lowpoly [-h] [--triangles N] [--contrast N] [--simplify]

optional arguments:
  -h, --help     show this help message and exit
  --triangles N  specifies the number of triangles that will form the final
                 image
  --contrast N   forces additional contrast between triangles in the image
  --simplify     blurs the image w/ a bilateral filter to simplify small,
                 extraneous details and focus triangulation more on the
                 general subject(s) in the image
```

References

[1] Achanta, R., Shaji, A., Smith, K., Lucchi, A., Fua, P., and Süsstrunk, S. SLIC superpixels compared to state-of-the-art superpixel methods. *IEEE Transactions on Pattern Analysis and Machine Intelligence 34*, 11 (2012), 2274–2281. http://www.kev-smith.com/papers/SLIC_Superpixels.pdf.

[2] Akella, A. Modern art rendering. https://inst.eecs.berkeley.edu/~cs194-26/fa17/upload/files/projFinalProposed/cs194-26-aeh/paper.pdf, 2017.

[3] Arthipo. Marilyn Monroe portrait 34. online. https://www.arthipo.com/marilyn-monroe-portrait-34.html.

[4] Callahan, P. G. Fake impressionist paintings for images and video. https://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15463-s10/www/final_proj/www/pgcallah/PGCpaper.pdf, 2010.

[5] Gai, M., and Wang, G. Artistic low poly rendering for images. *The Visual Computer 32*, 4 (2015), 491–500. https://doi.org/10.1007/s00371-015-1082-2.

[6] Hertzmann, A. Painterly rendering with curved brush strokes of multiple sizes. In *Proceedings of the 25th annual conference on computer graphics and interactive techniques (SIGGRAPH)* (New York, NY, USA, 1998), ACM Press, pp. 453–460. https://www.mrl.nyu.edu/publications/painterly98/hertzmann-siggraph98.pdf.

[7] Hertzmann, A. Fast paint texture. In *ACM Symposium on Non-Photorealistic Animation and Rendering* (2002), ACM Press. http://www.dgp.toronto.edu/papers/ahertzmann_NPAR2002.pdf.

[8] Litwinowicz, P. Processing images and video for an impressionist effect. In *the 24th annual conference on computer graphics and interactive techniques (SIGGRAPH)* (New York, NY, USA, 1997), ACM Press, pp. 407–414.

[9] Ma, Y., and Chen, X. An interactive system for low-poly illustration generation from images using adaptive thinning. In *IEEE International Conference on Multimedia and Expo* (2017). http://staff.ustc.edu.cn/~xjchen99/lowpoly/lowpoly.htm.

[10] Style Galleries. Rainy day chic. online. http://www.style-galleries.com/2012/11/rainy-day-chic/.

IX. Appendix: Code

Code Language: Python 3.5.6, using the course environment

List of code files:

- `main.py` — the main executable that parses parameters and saves the resulting images (see the `README.txt` and `--help` for more details).
- `impressionist.py` — implements the high level painterly algorithm, including discretization of the image and integration of the edge and gradient images into the brush strokes.
- `brush.py` — implements the brush mechanic, allowing for flexibility in its shape (ellipse or rectangle), size, and orientation. It also implements the clipping mechanic and introduces "human error" to strokes (adding noise to color and orientation).
- `lowpoly.py` — implements the SLIC superpixel algorithm, and puts the pieces together between superpixelation, finding the initial vertices, and executing adaptive thinning.
- `thinning.py` — implements the adaptive thinning algorithm that ranks a set of vertices based on their importance in the image, and iteratively chooses the important vertices to compose the n-triangle low-poly image.

Acknowledgements

First and foremost, thank YOU, kind reader, for taking the time and putting the energy into looking over my work. If you're a TA, a double thank you for being one of the unsung heroes that make this course possible.

In general, thank you to the TAs for their responsiveness, proactivity, and general-helpfulness in promoting discussion, answering questions, and keeping this class organized and on-track. An additional shout-out goes to the Slack community, to my peers who in the collective struggle of this course and shared screenshots and discussion of both successes and failures.

Finally, thank you to all of the offline folk who put up with me while I took photos for this class or forced them to look at the results of the work we did in this class.