Non-Photorealistic Rendering: Paintings

Christina Ou
Carnegie Mellon University
christinaou@cmu.edu

Connor Lin
Carnegie Mellon University
connorlin@cmu.edu

Abstract

We present a pipeline for creating painterly renderings of images using a canvas of individual stroke objects. Our pipeline determines the properties of each stroke, which include location, orientation, length, and color. The final result is created by merging multiple layers of increasing detail, and various painting styles such as Impressionism can be emulated by varying brush parameters and rendering textures. Additionally, we explore and evaluate different methods for estimating stroke orientations based on gradient domain processing.

1. Introduction

The computer graphics community strives to create realistic and/or expressive renderings of content. One area of interest is imitating the styles of popular art. As brush paintings have much variation between art styles and different artists, there are many parameters that can define how a canvas is painted.

The pipeline is flexible enough to emulate different art styles, but we specifically sought to replicate the "Impressionist" art style in this project. This style seeks to capture a feeling or experience rather than to achieve accurate depiction. One paragon of impressionism is Claude Monet's "Impression: Sunrise" (Figure 1). We seek to recreate such images through our pipeline.

2. Related Work

Much of our work is predicated upon the paper "Image and and Video Based Painterly Animation" by James Hays and Irfan Essa [2]. As the paper remains very general about the pipeline, we sought to more explicitly explain the steps required in the pipeline so that any user could re-implement this pipeline.

The original pipeline proposed by Hertzmann [1] and Litwinowicz [3] uses local gradients to estimate the stroke orientations, which can be done using MATLAB's imgradient function. We evaluated an additional method, the long edge detector, to generate gradients. This method is detailed in the Gradient Shop paper and utilizes local gradient

Figure 1. "Impression: Sunrise" by Claude Monet. A key example from the Impressionism movement.

Figure 2. Image that the pipeline will render.

saliency. This method weights gradients based on the length of their edge, with higher weights placed on longer edges.

3. Process

Now we will enumerate the different steps in our process, which involves determining the properties of each stroke and rendering the final result. We tested our pipeline initially on Figure 2.

The foundation in our pipeline lies in our brush "stroke." These are the main building blocks of our rendered painting and their properties help determine our image style (Figure 3). Our strokes have the following properties:

- (r, c), rol, col of stroke anchor
- ang, angle of stroke with respect to anchor
- w, width of brush stroke
- l_1, l_2, lengths 1 and 2 of brush stroke. These represent the two length directions that are opposite of the anchor.
- $color$, color of entire stroke in R,G,B
- $opacity$, opacity used for alpha compositing
- $strong$, boolean of "strong" designation. Strong strokes have stronger gradients.
- $stroke_{pixels}$, all of the (y, x) pairs representing all of the pixels on the canvas that the stroke will cover.

In our pipeline, we will determine all of these properties for each stroke in order to render it onto our canvas. The stroke will be additionally manipulated by a user-provided texture and alpha image. These stroke images help determine a style for the rendered painting.

Each layer will add on more detail to our canvas. The base layer contains the largest brush strokes and helps set a foundation for the canvas. Each higher layer has more narrow brush strokes and helps fill in detail along the edges in the image. With this process, we emulate the technique of an artist adding on finer and finer detail.

3.1. Brush Stroke Initialization

We must first determine the placement of our brush strokes. The placement of brush strokes determines where the detailing on our canvas will lie. An important parameter to consider is w_b, the width of the brush stroke. To avoid having strokes that are too close together, all brush strokes are spaced w_r apart, where $w_r < w_b$.

Our base layer contains the most generalized rendering of our image. In this layer, we want our entire canvas to be filled with the painted rendering of the image. Thus, the brush strokes (r, c) on the base layer are uniformly random throughout the image while fulfilling the condition that $||r_2 - r_1, c_2 - c_1||_2 < w_r$.

We first used random sampling in the canvas space to pick a position (r, c). However, this becomes inefficient as we start to fill the canvas as we may not randomly sample a position in a small corner of the canvas.

Thus, we instead first generated the list of all possible stroke positions (r, c). For the base (layer Figure 4), the set of strokes is

$$S = (r, c)_{w_b \leq r \leq H - w_b, w_b \leq c \leq W - w_b} \quad (1)$$

where H is the canvas height and W is the canvas width. We randomly sample from our list of possible stroke positions to generate random ordering. As we pick points, we reduce our possible stroke positions that conflict with (r, c).

For the higher layers, we want brush strokes in areas with more detail. In most paintings, these are along the edges of the image content. With more detail, w_b also decreases for finer strokes. To find these edges, we used a Canny edge detector and varied σ proportional to w_b for narrower edge boundaries.

In our higher layers (Figure 5), the possible stroke positions are the positive values in the Canny edge detector. Then, we can sample from our positive positions, removing possible stroke centers that conflict with our sampled point.

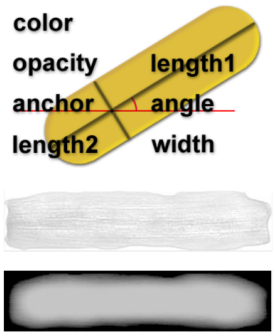

Figure 3. A stroke and its properties. Below it we show a brush stroke texture and alpha mask pair.

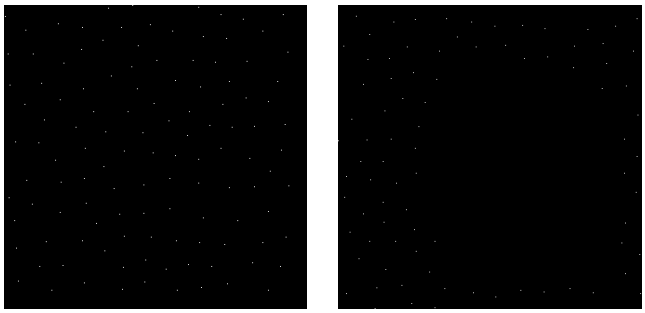

Figure 4. Stroke anchors in the area near the bottom right peach for base layer (left) and layer 1 (right). Anchors for the base layer are randomly uniform, while anchors on layer 1 must be located near the edge outline of the peach.

3.2. Stroke Orientation

Next, we determine the orientation of each stroke. Each layer is first blurred using a Gaussian kernel with width proportional to the brush width w_b. The gradients of each layer were then computed using MATLAB's imgradient. We use the notion of "strong strokes," in which a stroke is considered to be "strong" if its anchor lies on a gradient with magnitude greater than some threshold t, where $t = 0.15$ was chosen for our results.

These strong strokes are determined for each layer and contribute to the same radial basis function. Our first approach was to use MATLAB's radial basis neural network with the strong stroke gradients as the input data points. The gradients for each non-strong stroke were then determined by feeding the desired anchor point into the network. However, the network seemed to always output the same gradient values regardless of the mean squared error goal or spread used when training the network.

Therefore, we took a different and more intuitive approach to estimating gradients by only considering the n nearest strong strokes to each non-strong stroke s, where n was chosen to be 20 for our results. The gradients for s were then approximated by weighting the gradients of the strong strokes by their inverse distance to s. Both linear and Gaussian weighting were experimented with, and linear weighting appeared to produce better results.

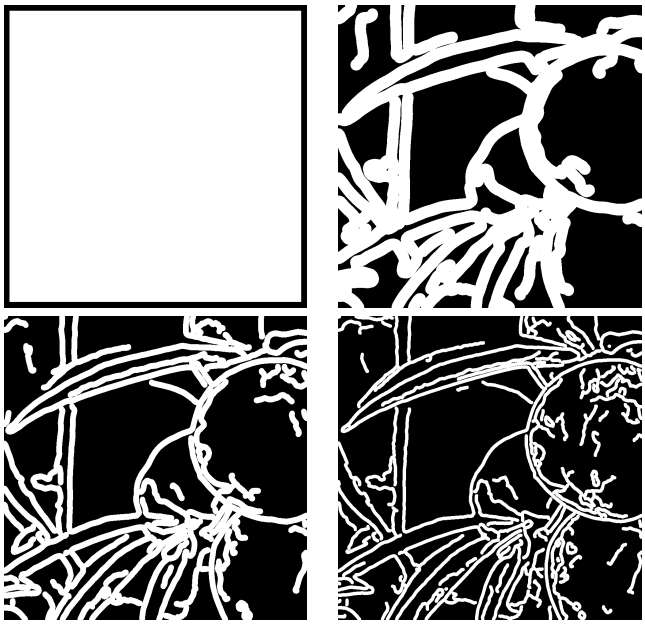

Figure 5. Top left (base layer), top right (layer 1), bottom left (layer 2), bottom right (layer 3). White areas indicate possible positions for stroke anchors at each layer.

$$\text{grad}_{\text{stroke}} = \frac{1}{\sum_{i \in n} \frac{1}{\text{dist}_i}} \sum_{i \in n} \text{grad}_i \frac{1}{\text{dist}_i} \tag{2}$$

After computing the weighted gradients $[g_x, g_y]$ for a stroke s, its orientation is set as $s_\theta = \text{atan2}(g_y, g_x) + 90$, where we add 90 degrees to the angle so that the stroke is oriented alongside nearby edges. See Figure 6 for a plot of the computed stroke gradients.

3.3. Edge Clipping

With our stroke positions and angles, we now need to determine the stroke's lengths. We need to compute both length1 and length2, which represent both lengths from the stroke's anchor that the stroke can grow.

We grow the stroke in accordance to the stroke anchor's angle (s_θ for length1 and $s_\theta + 180$ for length2). Our update rules for a stroke pixel (x, y) are

$$(dX, dY) = \begin{cases} (1, \tan(\theta_{stroke}), & \text{if } \tan(\theta_{stroke}) < 100 \\ (0, 1), & \text{otherwise} \end{cases}$$

We add pixels to our strokes according to the following equations. S_{pixel_line} defines the coordinates of points

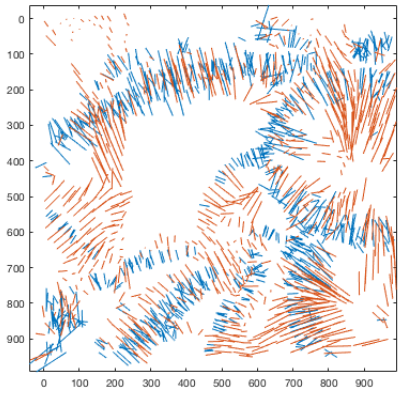

Figure 6. Quiver plot of gradients from Layer 3. Blue lines represent gradients from strong strokes and orange lines are our interpolated gradients.

Figure 7. layerMask for base layer.

starting from our anchor (y, x) in either direction. The line continues until the brush strokes hit a boundary as defined in our "layerMask". Each "layerMask" is a boolean image representing the boundaries for strokes in that layer. The baseLayerMask is shown in Figure 7 and the other layer's masks are the same as those in Figure 5. These (y_i, x_i) points thus define the coordinates of points in our brush line.

$$\begin{aligned} S_{pixel\_line} = (y_i = y + i * dY, x_i = x + i * dX) \quad (3) \\ \text{where } 1 \leq y_i \leq \text{numRows}, \\ 1 \leq x_i \leq \text{numCols}, \\ \text{layerMask}(y_i, x_i) = \text{true}, \\ ||y_i - y, x_i - x||_2 \leq 2w_b \text{ (*See Appendix A)} \end{aligned}$$

Our last condition was added on after testing our entire pipeline. Please see Appendix A for a discussion on why we limit half of the brush stroke length to be less than $2w_b$.

With the coordinates (y_i, x_i) along the line, we now calculate all of the brush points factoring in the width of the brush stroke in order to define the elliptical brush stroke shape.

$$\begin{aligned} S_{pixels} = (y_j = y_i \pm j, x_j = x_i \pm j) \\ \text{if } 1 \leq y_j \leq \text{numRows}, \\ 1 \leq x_j \leq \text{numCols}, \\ ||y_j - y_i, x_j - x_i||_2 \leq 2w_b \end{aligned} \quad (4)$$

3.4. Color Extraction

Stroke color is relatively straightforward to compute once we have determined the stroke anchor, orientation, and lengths. Each stroke S takes on the average color of the pixels that S covers.

$$S_{\text{color}} = \frac{1}{\sum_p \delta(p)} \sum_p p_{\text{color}} \delta(p) \quad (5)$$

where $\delta(p) = 1$ if stroke S covers p, and 0 otherwise.

3.5. Rendering

With all of the stroke information, we now can render the strokes onto each layer of our canvas and combine our layers. In order to determine our stroke's color values, we utilize the texture and alpha mask images as shown in Figure 3.

Starting from the base texture and alpha mask images, we first add color to the texture. Then we scale texture and alpha by w_b, the brush width, and $(\text{length}_1 + \text{length}_2)$, the brush total length.

To rotate the stroke, we must first note that we want to rotate the stroke by its center, which may not necessarily be $w_b/2$ or $(\text{length}_1 + \text{length}_2)/2$. MATLAB's imrotate function only rotates an image by its center point, so padding is necessary for the stroke to be accurately centered and rotated. We determine our stroke's column value to be weighted by either length_1 or length_2 depending on the stroke's angle.

```
function center_img(img,stroke_col)
if img_h < img_w
    radius = max(stroke_col, img_w-col)
    top_bot_pad = radius - img_h/2
    if col > img_w-col
        right_pad = radius-(img_w-col)
    else
        left_pad = radius-col
    end
else
    radius = img_h/2
    left_pad = radius-(img_w-col)
    right_pad = radius-(img_w-col)
end
centered_img = pad img by 4 pad lengths
```

We center and rotate both the texture and alpha images. As we have the pixel positions and color for the entire brush stroke, we can now place these onto our canvas. We use alpha compositing to handle opacity from multiple brush strokes overlapping with one another.

3.6. Alpha Compositing

We experimented with different methods of adding each stroke to the canvas. MATLAB does not directly support alpha values for an RGBA image, so we used the following methods to test alpha compositing.

For a pixel p on the canvas, let $C = [C_r, C_g, C_b]$ and C_α represent the color and alpha of p. Likewise, let $T = [T_r, T_g, T_b]$ and T_α represent the color and alpha of the corresponding pixel on the texture.

The first method is to ignore the opacity and place strokes on top of each other with no compositing:

$$C' = T \tag{6}$$

This produced somewhat rough and grainy results, unlike the typically smooth strokes used in Impressionist paintings.

Therefore, we tried using premultiplied alpha compositing. The RGB channels are premultiplied by alpha, which is composited the same way as the RGB channels themselves. $C = [C_\alpha C_r, C_\alpha C_g, C_\alpha C_b, C_\alpha]$ and $T = [T_\alpha T_r, T_\alpha T_g, T_\alpha T_b, T_\alpha]$.

$$C' = T + (1 - T_\alpha)C \tag{7}$$

Figure 8 shows the base layer with and without alpha compositing. The brush strokes are very distinct and rectangular without alpha compositing and have a much smoother, blended appearance with alpha compositing.

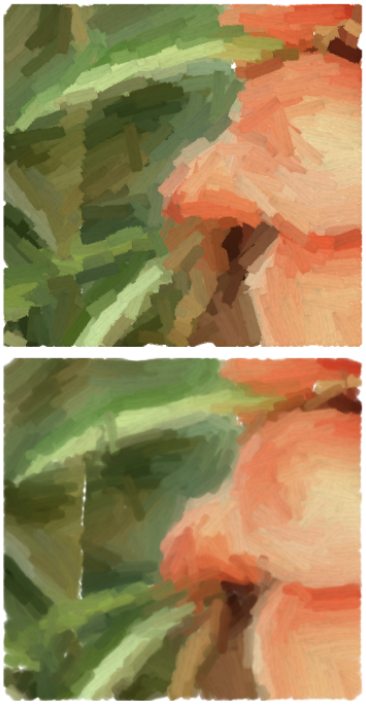

Figure 8. Top: base layer without alpha compositing. Bottom: base layer with alpha compositing

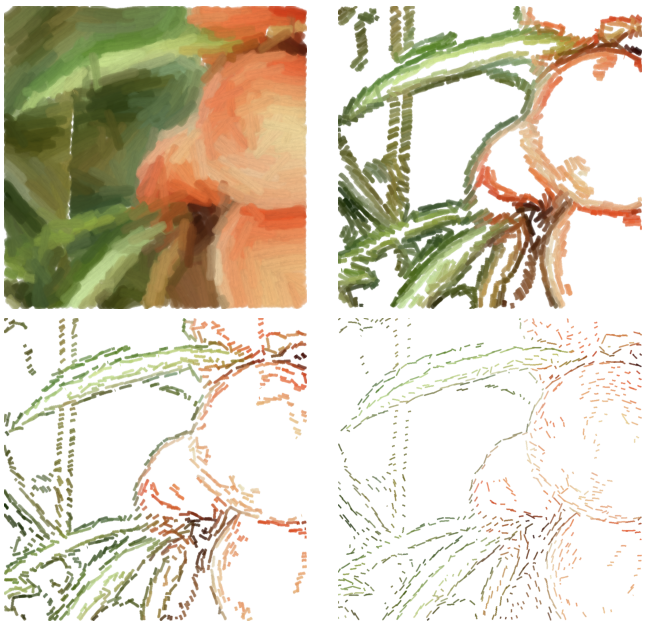

Figure 9. Brush strokes for each respective layer: top left (base layer), top right (layer 1), bottom left (layer 2), bottom right (layer 3).

4. Results

Our individual layers for the Peach image are shown in Figure 9. The original peach image and the final accumulation of layers are shown in Figure 10. Our rendering looks aethestically pleasing and does indeed capture the impression of the peaches. There is white space in some areas such as around the image boundaries, and this is due to a lack of brush strokes in those areas.

5. Applying GradientShop

One of the most flexible parts of our pipeline is the computation of stroke orientations. Therefore, we looked into the GradientShop paper by Bhat, Zitnick, Cohen, and Curless (2009) [4], which proposes a long-edge detector that can be used to measure local gradient saliency. This long-edge detector is capable of applications such as sharpening images or creating non-photorealistic cartoon renderings of images. We implemented and integrated the detector into

Figure 10. Original peach image and our final canvas rendering.

our pipeline for computing stroke orientations.

5.1. Long-Edge Detector Implementation

We began by finding the local gradient magnitude p_m and direction p_o at each pixel using MATLAB's imgradient function. In order to detect faint edges, the gradient magnitudes $\hat{p}_m$ for each pixel is normalized with respect to the magnitudes in its local 5x5 neighborhood.

Next, we followed the message-passing scheme described in the paper to compute m_0 and m_1, which together estimate the length of the edge that p sits on.

$$m_0^t(p) = \sum_q w_\alpha w_\theta(\hat{q}_m + m_0^{t-1}(q)) \tag{8}$$

$$m_1^t(p) = \sum_q w_\alpha w_\theta(\hat{q}_m + m_1^{t-1}(q)) \tag{9}$$

The neighborhood of pixels q are the 4 nearest pixels when projecting a distance of $\sqrt{2}$ along p_o for m_0 and $p_o + 180$ for m_1. w_α are the bilinear interpolation weights, and w_θ is a similarity metric between the orientations at p and q, where $\sigma_\theta^2 = \frac{\pi}{5}$.

$$w_\theta(q) = \frac{exp(-(p_\theta - q_\theta)^2)}{2\sigma_\theta^2} \tag{10}$$

This message-passing scheme was run for 60 iterations. The final edge length e^l and orientation e^o estimations are therefore:

$$e^l(p) = m_0^{60} + m_1^{60} + \hat{p}_m \tag{11}$$

$$e^o(p) = p_o \tag{12}$$

Finally, the local x and y gradients, u_x and u_y, were computed using MATLAB's imgradientxy function. These were scaled using e^l and e^o following the proposed saliency measure:

$$s_x(p) = \cos^2(e^o(p))e^l(p)u_x(p) \tag{13}$$

$$s_y(p) = \sin^2(e^o(p))e^l(p)u_y(p) \tag{14}$$

We can see that the scaled gradients s_x and s_y will emphasize gradients that lay on longer edges.

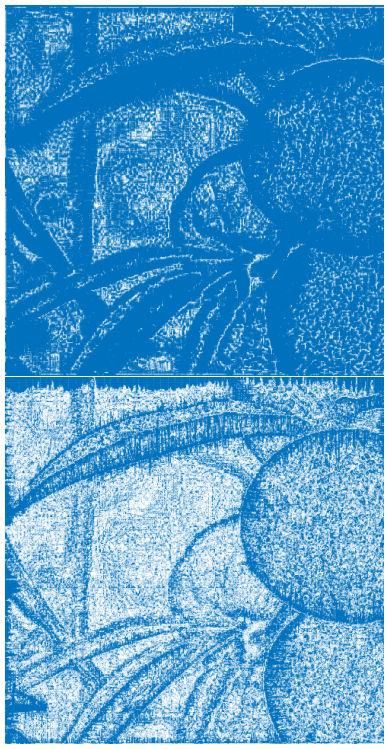

Figure 11. Top: MATLAB's imgradient result. Bottom: scaled gradients using the long-edge detector.

5.2. Comparison

Instead of using the local gradients to determine the strong strokes described in section 3.2, we used the scaled gradients s_x and s_y with a threshold of $t = 0.1$. See Figure 11 for a comparison between MATLAB's imgradient result and the scaled gradients using the long-edge detector. We can observe that the longer edges are emphasized more relative to the rest of the image after applying the proposed local gradient saliency.

The rest of the pipeline is then run using these new stroke orientations. The results are shown in Figure 12. We can see that longer edges are sharper, as the vertical stem on the left side of the image looks more accurate using the scaled gradients. However, shorter edges, such as those near the left-most peach, do not appear to be as sharp. This is somewhat expected, as the local gradient saliency measure only emphasizes longer edges.

6. Conclusion

In our recreation of the non-photorealistic rendering pipeline, we were able to transform images into an Impressionist style of art. Given general guidelines, we formulated the equations and algorithms necessary to produce this pipeline. We compared the IVBPA paper's local gradient

Figure 12. Top: Original result. Bottom: Result using scaled gradients using the long-edge detector.

methods with GradientShop's long edge detector and discovered the positives and limitations of each method. There were many learnings during the project process, and we were able to achieve accurate and aesthetically pleasing renderings.

7. Future Work

The pipeline for transforming images to paintings is quite flexible, and it would be interesting to see how different styles could be emulated by varying parameters such as brush width and stroke texture.

Extending the pipeline for videos would also be interesting, and the original paper mentions a method for doing so by computing optical flow and allowing the opacity of each stroke to change over time. However, this will also involve improving the speed of the pipeline, as there are several parts that depend on each other.

Additionally, the pipeline is flexible because it has a large number of parameters that can be varied. These include

- w_b, width of brush stroke
- w_r, regeneration width (space between stroke anchors)
- T_b, set of brush texture and alpha mask images
- $C_{s,max}$, number of previous frames to average color over
- P_r, palette reduction, size of color palette to render image
- v, amount of noise to add to each layer

Currently, if a parameter such as brush width is varied, the entire pipeline needs to be run from the beginning. It would be great if a tool could be developed or the pipeline rewritten for dynamically adjusting parameters, so that produced paintings can be more easily tuned to a user's satisfaction.

8. Appendix A

In our first work through of the pipeline, we used the IVBPA method to perform edge clipping. We created

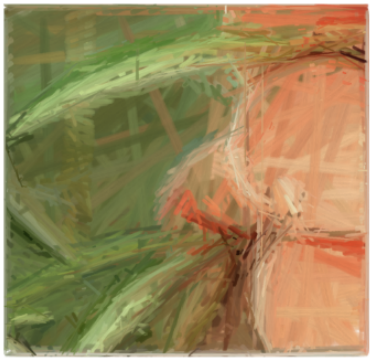

Figure 13. Original peach rendering before introducing additional condition.

S_{pixel_line}, which defines the coordinates of points starting from our anchor (y, x) in either direction. The line continues until the brush strokes hit a boundary as defined in our layerMask. These (y_i, x_i) points thus define the coordinates of points in our brush line. In the below equation for S_{pixel_line}, we only stop growing the brush stroke when we hit a boundary defined in layerMask.

$$S_{pixel\_line} = (y_i = y + i * dY, x_i = x + i * dX) \quad (15)$$
$$\text{where } 1 \leq y_i \leq \text{numRows},$$
$$1 \leq x_i \leq \text{numCols},$$
$$\text{layerMask}(y_i, x_i) = \text{true},$$

However, this led to the initial results of the peach image as shown in Figure 13. We realized that some of our brush strokes were growing too long, making some orientations look out-of-place and averaging too many color values throughout the stroke. We decided to include one more condition in order to shorten the brush strokes while still maintaining all of the other brush stroke properties. This added condition led us to the results we generated in Section 4.

$$\text{Added condition: } ||y_i - y, x_i - x||_2 \leq 2w_b \quad (16)$$

References

[1] A. Hertzmann. Painterly rendering with curved brush strokes of multiple sizes, 1998.

[2] I. E. James Hays. Image and video based painterly animation, 2004.

[3] P. Litwinowicz. Processing images and video for an impressionist effect, 1997.

[4] M. C. B. C. Pravin Bhat, C. Lawrence Zitnick. Gradientshop: A gradient-domain optimization framework for image and video filtering, 2010.

Figure 14. Mountain sunrise. Top left (base layer), top right (layer 1), middle left (layer 2), middle right (layer 3), bottom left (original image), bottom right (final result).

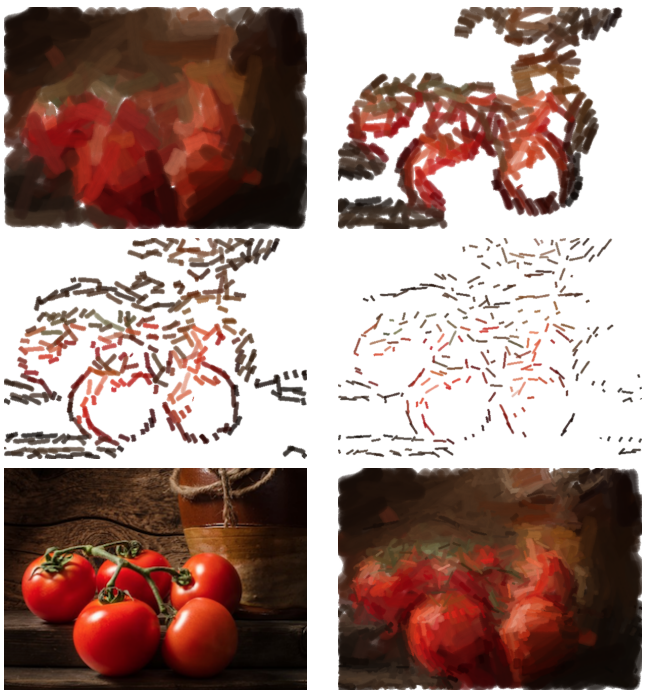

Figure 15. Still life tomatoes. Top left (base layer), top right (layer 1), middle left (layer 2), middle right (layer 3), bottom left (original image), bottom right (final result).