Hatching Rendering Based on a Single Photo

David Delventhal Xiaowei Wang

Abstract

There exist many algorithms to create a non-photorealistic image from a photo. Many of these algorithms make the photo look like a painting or a cartoon. This project, however, gives the photo a hatching shading style. Existing techniques for hatching rendering either require the user to manually draw hatching or create a 3D model on which the algorithm rendering the hatching. This project focuses specifically on photos of architecture.

1. Introduction

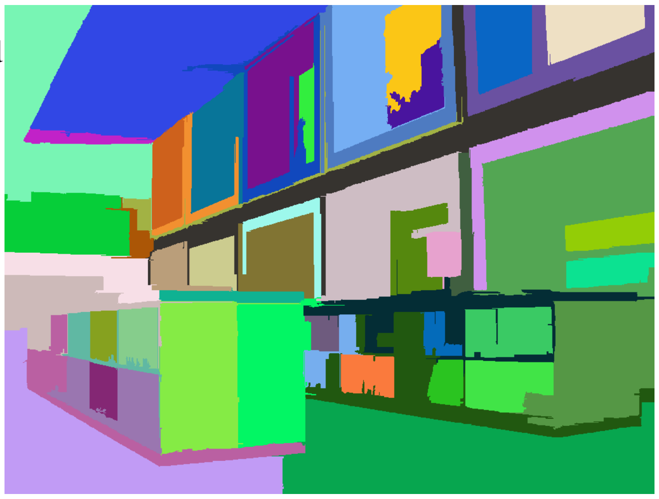
Over the past years, most of the research in computer graphics rendering has been devoted to photorealism. However, as computer graphics has matured, non-photorealistic rendering (NPR) has become a more and more important area of research in computer graphics. Much research have been devoted to investigating different techiniques to simulate a variety of artistic results, from pencil to watercolor. Among those, hatching is one of the most popular styles. Many NPR images created start from 3D models and combine many different algorithms to yield exactly the image the user desires. But this would be too expensive in some case, while only brief and simple results are needed. This paper introduce a convenient way of hatching-like rendering based on single input image.

2. Algorithm

First of all, we save the input color photo in order to keep the original information without any modification. While generating hatching effect, the most important and complicated part is to decide the direction of the hatching lines. By observation, we decided to make a region-based algorithm, in which we first divide the input image into smaller regions by a set of features include colors, light, and shapes. The direction of hatching within each single area is based on the geometric attributes of the region. The next step is to render the strokes with colors and light by making use of the information saved in the first place. Finally, outlines are added to the image to finish the procedure.

2.1 Segmentation

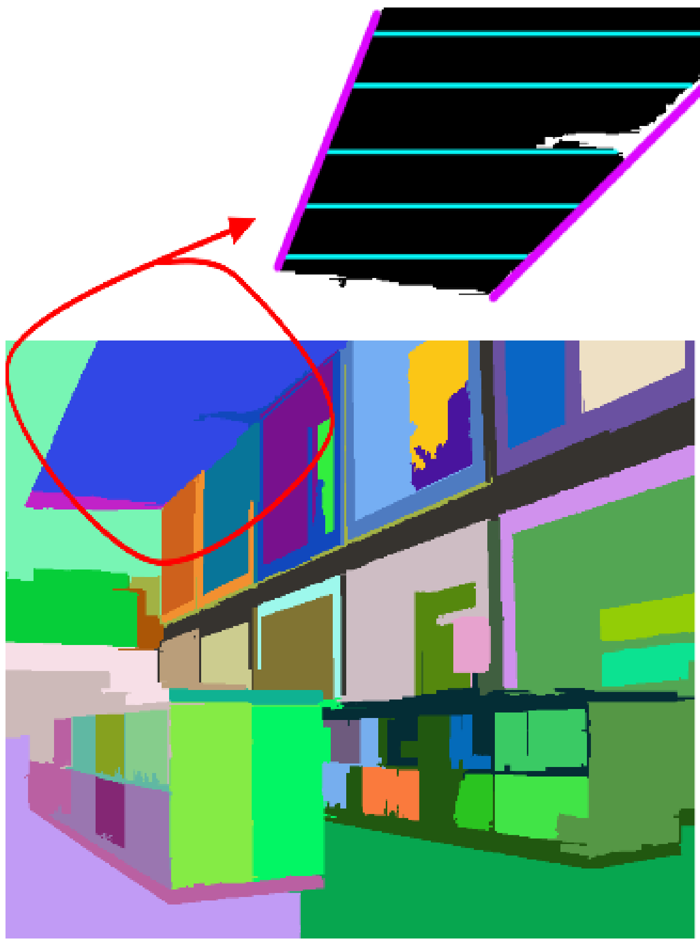
Our first step is to apply over-segmentation method of Felzenszwalb et al. [2004] to get a set of regions of the input picture. Each region is assigned an ID number as a label. The ID numbers are in increasing with the order of detection.



Segmented regions.

2.2 Stroke Direction

Once the separate regions of the image have been indexed, horizontal rays are cast across the image. Whenever, one of these rays intersects the border between two regions, the intersection point is saved in a list associated with the region being entered. A ray is only allowed to intersect a region once, any other intersections will be discarded. This step is performed twice, once with the rays going left to right, and another time right to left. The first pass finds points along the left side of the region, and the second finds the ones along the right. Next, for each region's points, a vector is found from point i to point i + 1. Each of the vectors for a particular region is then normalized and compared to all the others. The vector whose direction is most similar to all the others is chosen as the direction for that region.

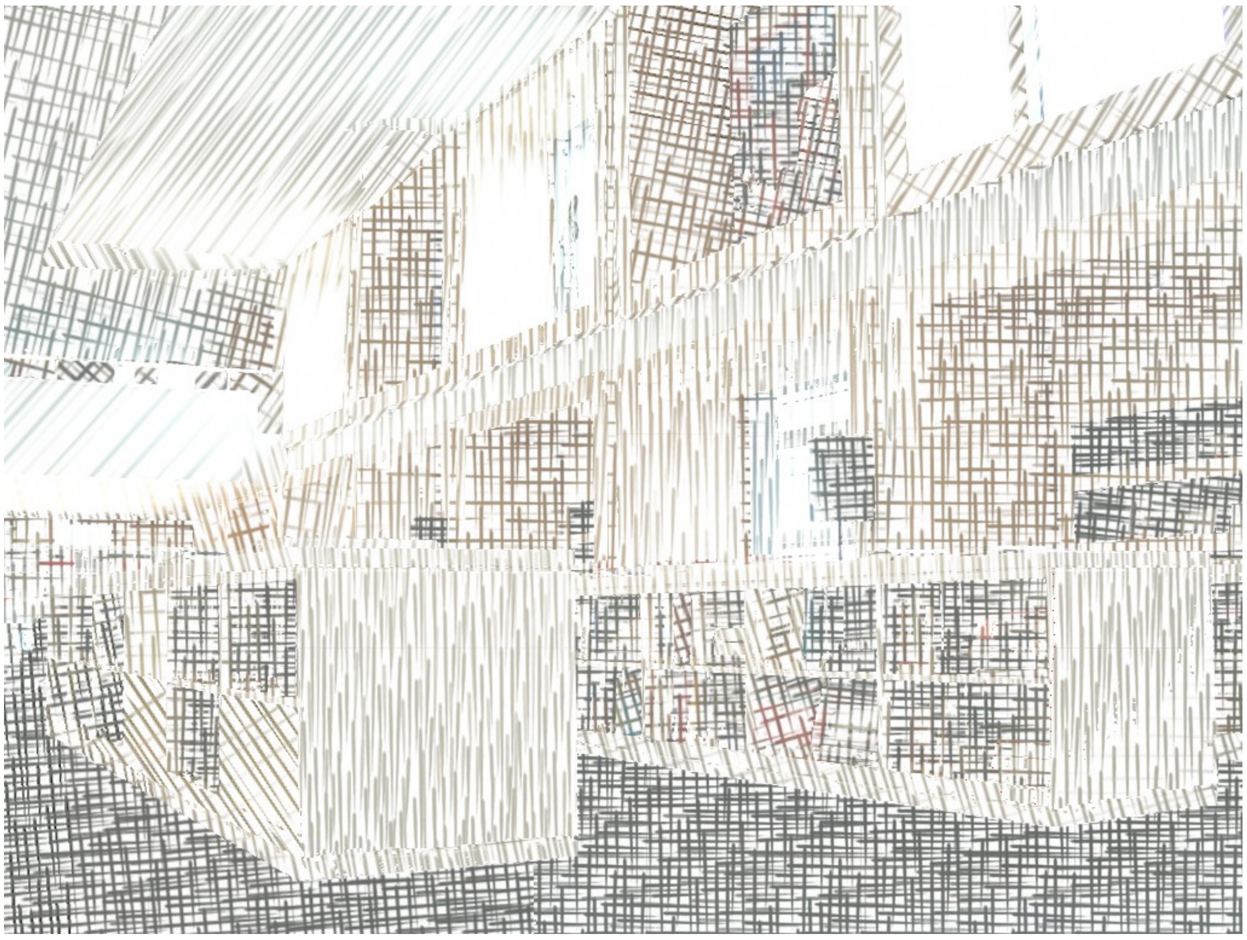


Horizontal rays intersect with a region. Intersection points are then used to plot lines on either side of the region. These line are then used to determine the direction of the region.

2.3 Stroke Rendering

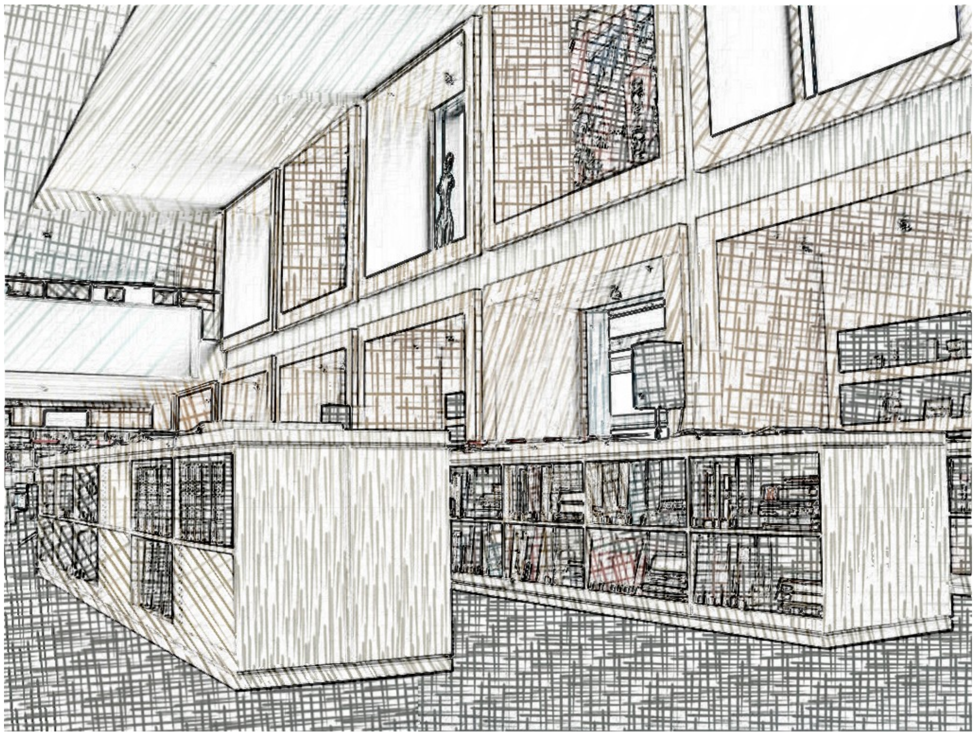
After a direction has been associated with each region, the strokes are rendered. First, for each pixel in the image, the pixel's direction is determined by using its region index to look up its region's direction. Next, this direction is used to fetch a pixel from each of the stroke images. Then, the intensity value from the input photo is used to determine the amount of strokes to render: the darker the pixel, the more strokes. This intensity value to used in a function which blends between stroke images. Thus, the smaller the value, the more weight the stroke images with more strokes gets. This stroke amount value is then used to linearly interpolate between the input photo's color and white in order to obtain colored strokes. This technique is very similar to the way that [Card and Mitchell] used for hatching rendering.

Hatching images which are blended to produce out strokes


The result after the strokes have been rendered.

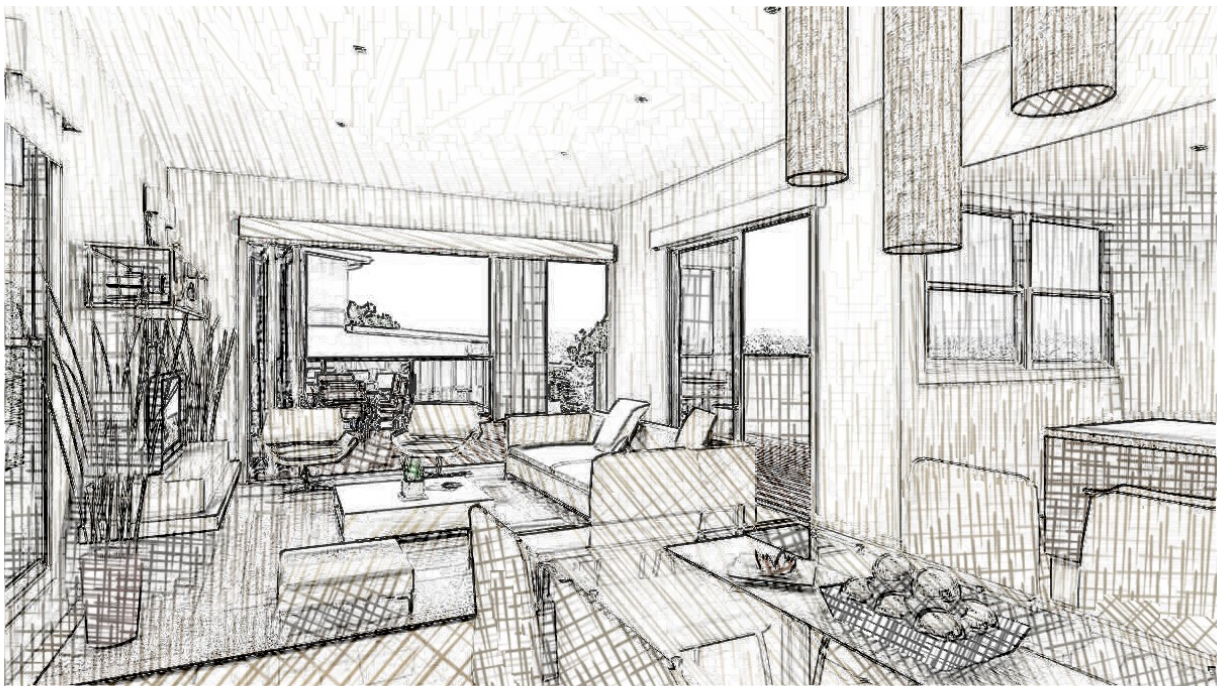
2.4 Outlines

The final step of the algorithm is to render dark outlines around the objects in the photo. This is accomplished by first applying a sobel filter to the input photo. Then, the edge intensities of each pixel in the image are found by computing the vector length of each output pixel from the sobel filter. Next, this edge intensity image is clamped to a maximum value to make the lines only dark and not entirely black. Finally, subtract this edge image from the image on which the strokes were rendered to produce the final output image.



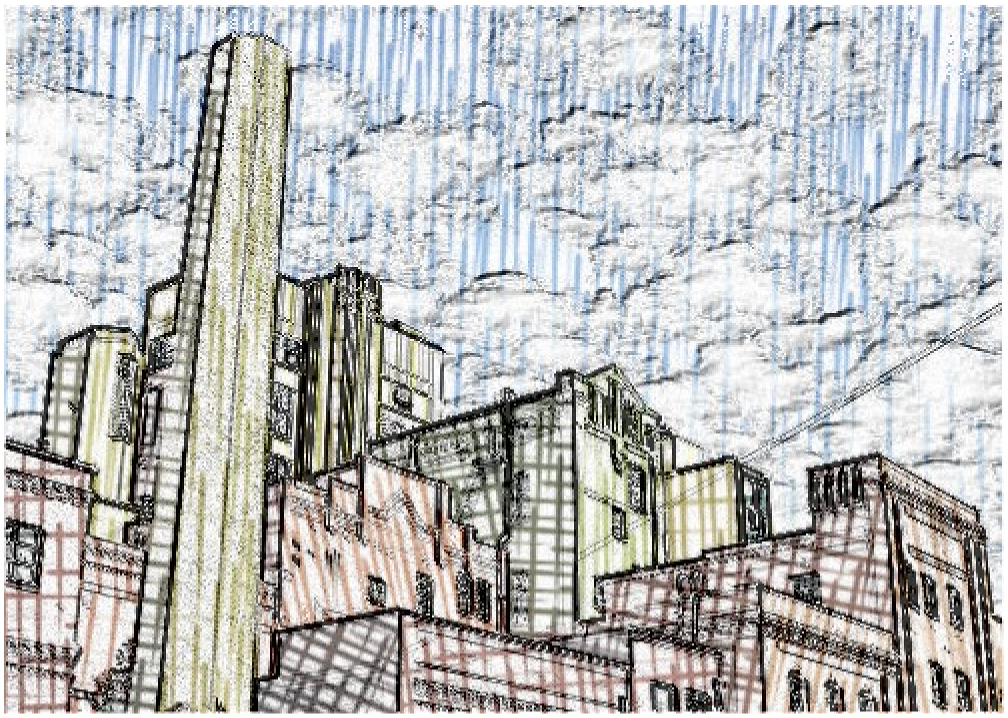
Outlines added to the previous image to provide a final result.

3. Results



4. Discussion

This algorithm could be extended in many ways. First of all, the edge detection algorithm could be improved to work better with highly detailed photos. The sobel filter detects all lines in the image, but the result be superior if the high frequency lines were removed. Secondly, support for curved surfaces in the input image could be added to the stroke rendering process. This would require finding a curve along a region instead of simply finding general direction. Finally, the color values to be modified in some artistic way. Currently, the stroke color is taken directly from the source photo. Maybe instead of this, we could make the individual strokes some solid color, perhaps the average of the colors from the input photo along the stroke.



Example failure case: there are strange edges around the clouds!

References

P. Felzenszwalb and D. Huttenlocher, “Efficient graph-based image segmentation,” IJCV, vol. 59, no. 2, 2004.

Drew Card and Jason Mitchell, “Non-Photorealistic Rendering with Pixel and Vertex Shaders”