A Project Report

On

Artistic Low Poly Rendering for images

Submitted in requirement for the course

LAB BASED PROJECT (CSN-300)

of Bachelor of Technology in Computer Science and Engineering

by

Haresh Khanna	15114031
Harjot Singh Oberai	15114032
Ketan Gupta	15114039
Nitish Bansal	15114048
Piyush Mehrotra	15114050

Project taken under

Dr. Ranita Biswas

Assistant Professor

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, ROORKEE

ROORKEE- 247667 (INDIA)

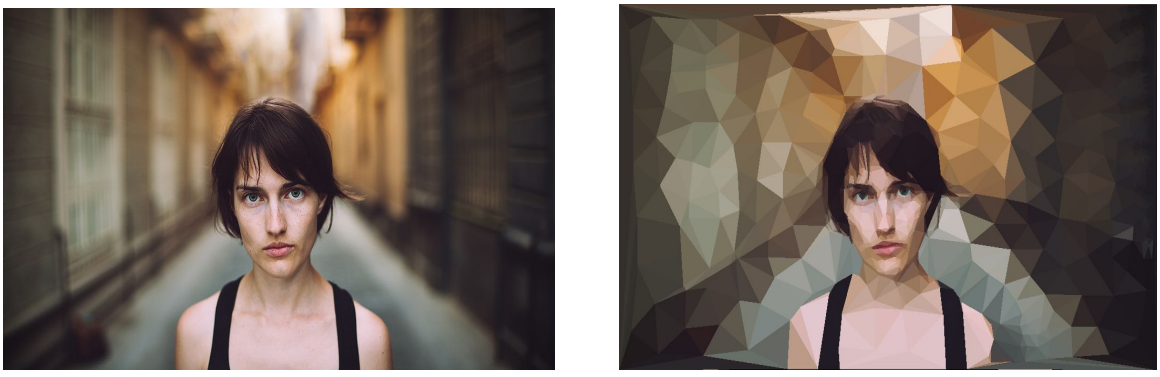
SPRING SEMESTER, 2018

1. Introduction

With the popularity of flat design, low poly style takes the fancy of more and more art designers. This design style origins from the early stage of the computer modeling, when the artists use a relatively small number of polygons to represent 3D meshes. But recently it has got new vitality in 2D illustration and graphics design. To get a abstract visual effect of an image, artists use polygons with flat coloring thus creating and artistic image. In this project we implement the paper Artistic Low Poly rendering [1] to achieve this triangle rendition for any given input image automatically.

(a) Original Image (b) Low-poly rendered image

Figure 1. Automatically generated low poly rendered image



(a) Original Image (b) Low-poly rendered image

Figure 2. Automatically generated low poly rendered image

2. Related Work

2.1 Image Compression

Image compression based on adaptive sampling and using triangle meshes has been a popular approach for reducing the size of very large images. This is the closest work related to out low poly rendering problem. Image compression is a NPR (Non Photorealistic Rendering) problem and thus does not take into account any artistic effects. It only focuses on the approximation of the reconstructed image to the original one, and does not care about the resulting mesh from the aesthetic aspects. Due to its application, such an image compression technique does not make use of error calculation techniques like PSNR (Peak Signal to Noise Ratio). Apart from this other image compression methods such as the error diffusion scheme (ED) [2] is although memory and time efficient but produces very low quality meshes.

2.2 Image Vectorisation

Vector graphics have been employed in a wide variety of applications due to their scalability and editability. Image vectorisation is a method of converting images into image vectors automatically. Mainly, triangle and quad meshes are used for this purpose. There is a subdivision based approach for triangle meshes which retains color continuity which is not required for our case. Another approach for image vectorisation which uses quad meshes needs the user to initialize the mesh manually which is a very time consuming and tedious task. Also, automation is what we seek in this implementation and manual initialisation goes against that.

2.3 Image tessellation

Image tessellation is the representation of an image in the form of tiles of varied shapes. One of the ways to achieve this is a Voronoi-based approach. That although, does retain a little detail of the image but it does not work as well as a triangle based

approach that is used in this project. The polygons in the Voronoi tessellation do not adapt to the shapes and details of the image very well.

3. Motivation for the project

These days, there is a lot of emphasis on material design and flat design hierarchies. Same trend is followed by artists to make images which are composed of triangles of the same color. That gives image an abstract representation and an artistic effect. Many of the previous vector editor softwares like Adobe Illustrator and CorelDraw can help the users to accomplish the task of creating triangle colored meshes. But it is obviously a tedious work that the artists must draw every single triangle by themselves. A later tool Image Triangulator App [3] provides a simpler way to generate triangle mesh. The artists can add vertices by clicking. But they must arrange every point carefully to generate the desired visual effects. So it still remains time consuming. Thus, having a method that takes into account the salient features of the image, edge features etc, such an algorithm that directly takes an image input and then converts that into a triangulated image is a very time efficient approach.

4. Objectives of work done

The input image is first processed by an edge detector. The tracked edges are then approximated by polygon curves. The edge of the polygons will be the constrained edges in the final Delaunay triangulation step. The feature edges are also used to compute a distance map to generate the feature flow field. This field will guide the arrangement of vertices in the mesh. At the same time, a saliency detection process is alternative to guide a non-uniform sampling between the front object and the background. We then optimize the sample points using a centroidal Voronoi diagram weighted by the feature flow field. After some iteration of the Lloyd relaxation algorithm, the sample points are add to the constrained Delaunay triangulation with the

constrained points in the previous step. Finally, we pick the color in the triangles and make some other color post-processing to generate the final result.

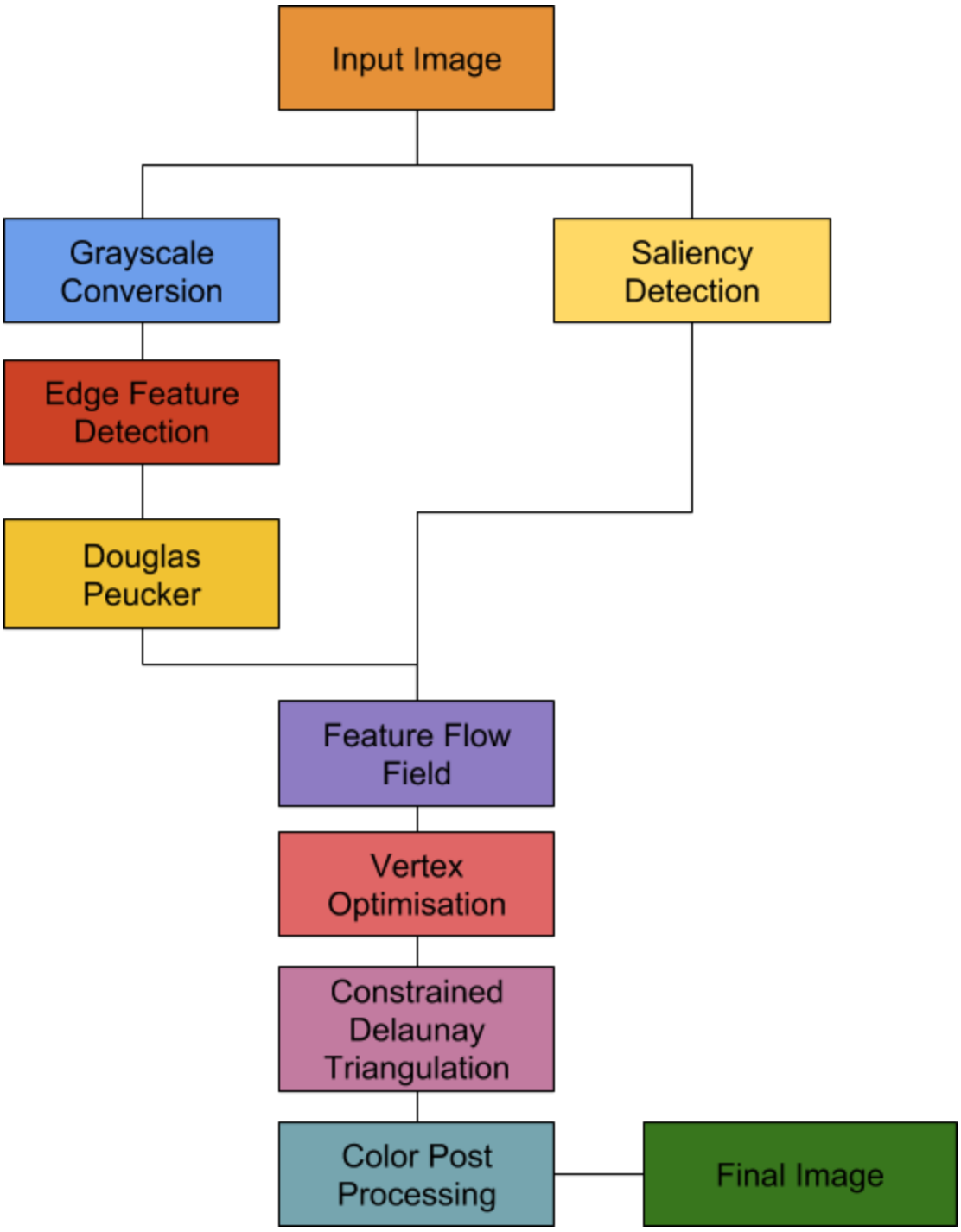


Figure 3. Project Workflow

5. Work plan to meet the objectives

We implemented all the steps, one at a time making sure that the output of that step was in the correct format for the next algorithm. This made sure that at the end of all the work we did not have to worry about the integration of code much, and this also gave us the flexibility to use both Python and C++ wherever the use of each of them made the job easier. After the completion of each step, the output was approved by the supervisor and any feedbacks in the regular meetings were taken seriously and worked upon.

6. Work description

6.1 Modules of the project

The project has the following components :

(a) Install and Run scripts

(b) Colored to grayscale image conversion

(c) Edge feature detection

(d) Douglas Peucker algorithm

(e) Salient feature detection

(f) Feature Flow Field marking

(g) Optimising seeds obtained from Feature Flow Field

(h) Constrained Delaunay Triangulation

(i) Color Post Processing for triangle coloring

6.2 Detailed description of the modules

All the output images are generated in **outputs/images** and all the output data is generated in **outputs/data**. Output images after each step are generated and shown below.

(a) Install and Run scripts

1. Install Script : A bash install script was written that installs all the required dependencies and copies the necessary library files from the **lib/** folder to the appropriate destinations
2. Run Script : A python run script **run.py** which takes an image as a command line input and converts it into a PNG image and saves in **outputs/images/** as **original.png**. Thereafter it runs code from within each of the folder in **src/** in the order required and saves the final image as **final.png** in the images folder.

Figure 4. Original Image (original.png)

(b) Colored to Grayscale image conversion

This takes the original colored image as input and converts it into a grayscale image in PGM format. This is written in Python and makes the use of **PIL (Python Image Library)** and its function **image.convert(<type>)** to do the job, first converting into grayscale and then to PGM format.

INPUT - original.png

OUTPUT - grayscale.pgm

Figure 5. Grayscale Image (grayscale.pgm)

(c) Edge Feature Detection

Canny edge detection is the most widely used edge detection algorithm these days. Most traditional algorithms use a series of low and high pass filters to obtain a gradient map which has edge areas highlighted. First, the low pass filter removes any noise from the image and then the resulting image is passed through a high pass filter to obtain edge areas. The final task is then to go over the remaining pixels in the edge areas and obtain thin, contiguous, non-jittered and well-localized edges. This is the most important and difficult step in edge detection, and is achieved by such techniques as skeletonization, non-maximal suppression, edge thinning, or the application of morphological operators.

But, the algorithm used in this paper i.e. Edge Drawing [4] works in a different manner. It chooses a few anchor points after obtaining edge areas and then uses a routing algorithm to connect those anchor points to form edges. Anchor points are the points of maximum gradient (computed using algorithms such as Prewitt). Also direction is stored along with gradient value. If $|G_x| \geq |G_y|$ then there is a vertical edge and vice-versa. The connecting algorithm takes into account direction as well as gradient values for the anchor points. Starting from an anchor, if the direction is horizontal, left/right movement is permitted and if the direction is vertical top/down motion is permitted. Only three immediate neighbours in the direction of movement (left/right/top/down) are checked and the one with the maximum gradient value is selected until convergence i.e.

(a) Edge point is reached

(b) Point with $G = 0$ is reached.

Repeating the above steps to include all anchor points gives the final Edge Map for a given input image.

INPUT - grayscale.pgm

OUTPUT - EdgeFeature.pgm + edgeSegments.data

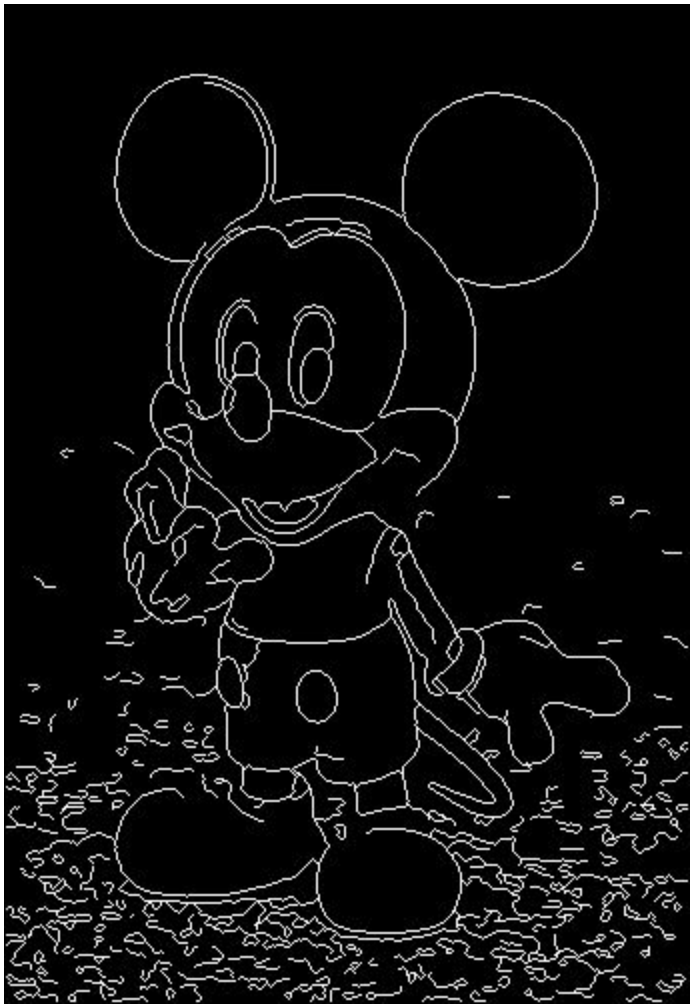


Figure 6. Edge Feature Detection (EdgeFeature.pgm)

(d) Douglas Pecker [5]

This algorithm takes a curve composed of line segments as input and outputs a similar curve with fewer number of points than in the original curve. For each line segment in the original curve it initially marks only the first and last point in the segment. It then finds the point which is farthest from the line segment made by using the first and last point as end points. If the farthest point is closer than a certain distance *epsilon* then only the first and last point are returned for the original curve, else if the distance is greater than *epsilon* the farthest point is marked and then the algorithm recursively calls itself with the first point and the farthest point, and the farthest point and the last point.

Epsilon is a pre-defined constant. Time complexity of this algorithm **O(nlogn)**(Average) and **O(n^2)**(Worst case). This algorithm is run on each of the segments output by Edge Feature Detection to simplify every segment.

INPUT - edgeSegments.data

OUTPUT - douglasSegments.data + douglasPoints.data + Douglas.pgm

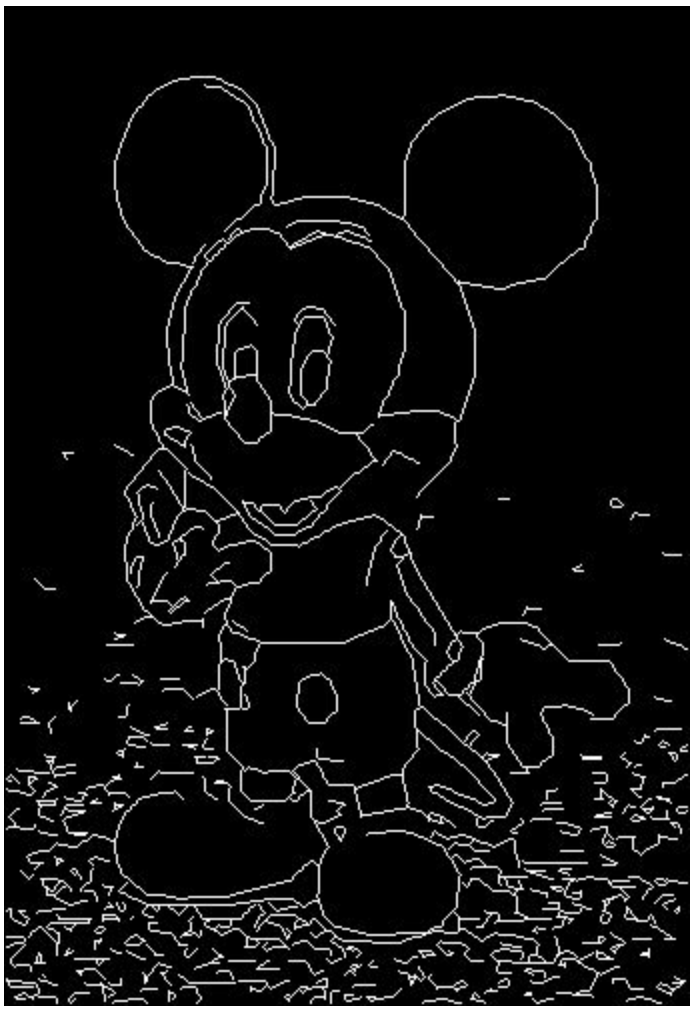


Figure 7. Douglas Peucker (Douglas.pgm)

(e) Salient Feature Detection

Salient feature detection separates the foreground from the background. To achieve the same, initially, image soft segmentation is done using mean shift operation. It helps to

improve the saliency results for some images. This groups together close pixels that are similar and generates a very smooth version of the image. Then a saturation histogram of the image is generated and back projected on the image to generate a initial saliency map in which the most salient features are dark. The initial back projection is then processed and inverted to obtain a smooth saliency map with the most salient features having white color. Now, as a heuristic we use the salient object with the largest area and ignore the other salient areas. This salient region with greatest area is used as the bounding box for foreground estimation using Grabcut algorithm. This makes the salient region segmentation much more accurate.

INPUT - original.png

OUTPUT - salientPoints.data + SaliencyMap.png

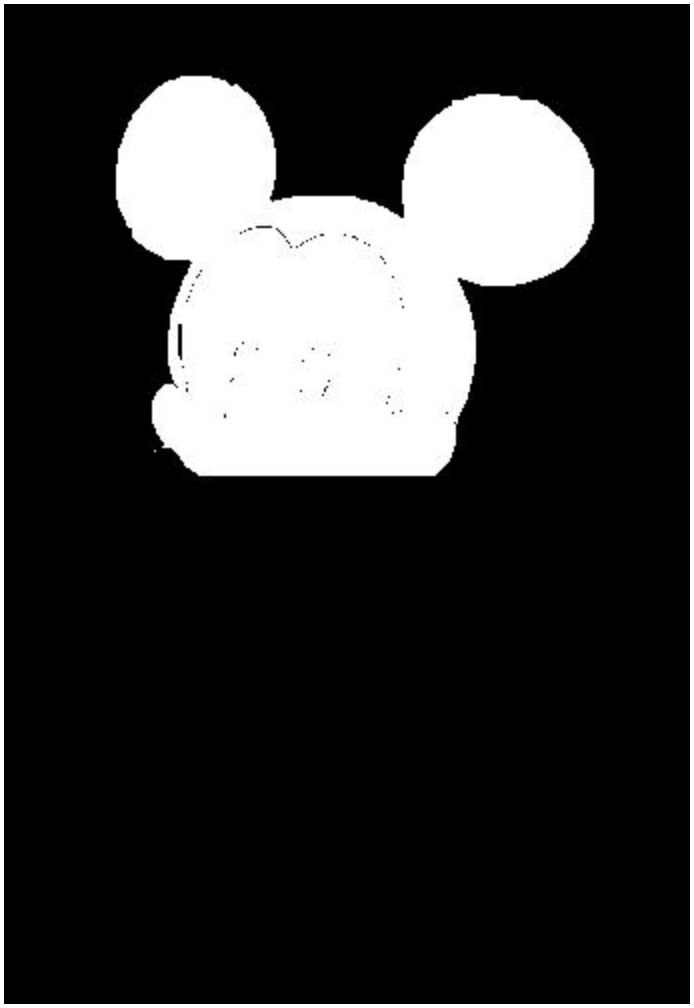


Figure 8. Saliency Map (SaliencyMap.png)

(f) Feature Flow Field Marking

Feature flow field is used to calculate **F** value for every pixel and then select anchor points for triangulation, using the **F** value calculated. The value **F** is calculated using the Distance values(**D)** of all points obtained for Voronoi-Tessellation. **D** is the euclidean distance between the given point and its seed (from Voronoi Tessellation). This distance is calculated using Jump Flooding Algorithm(JFA) [6]. This algorithm takes **O(nlogn)** time where n is the number of points in the image. All the points, which acts as seed for JFA are the edge points from the Edge Feature Detection method. It propagates the distance information in a jumping manner so that it can finish computing the distance map in log(n) rounds for an image of size nxn, regardless of the number of starting points. Then F is calculated as follows:

$$F(x) = \begin{cases} \frac{255}{m} D(x) \mathrm{mod}(m), & \text{if } \frac{D(x)}{m} \mathrm{mod}(2) = 1 \\ \frac{255}{m}(m - D(x) \mathrm{mod}(m)), & \text{if } \frac{D(x)}{m} \mathrm{mod}(2) = 0, \end{cases}$$

This **F** and **D** values are plotted by scaling them from **0** to **255** thus giving a spectrum from black to white with grays in-between. After the calculation of **F** for each point, anchor points need to be selected, which is the output for this module. It also needs to take into account the saliency region detected and choose more anchor points in the salient region than the non salient one in order to bring in more detail to the important part of the image. This point selection is achieved as follows:

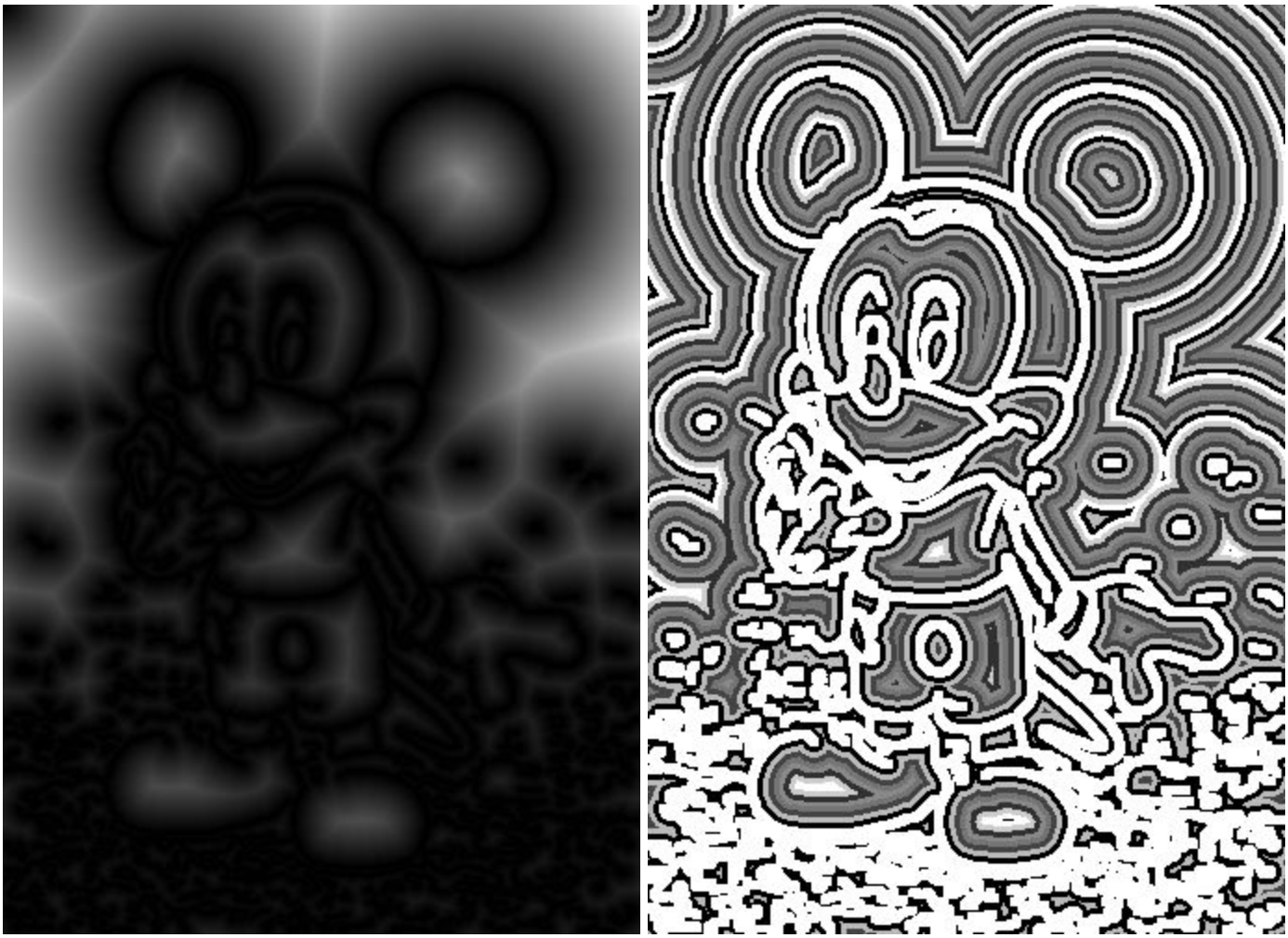
```
for each point p:
     if f(p) < 50: // choosing points with 0 < F < 50
          if salient(p) == true:
               add p to list_salient_selectable
          else:
               add p to list_non_salient_selectable

to_be_selected = 400
lambda = 0.25 // constant
in_salient = lambda * to_be_selected
in_non_salient = (1 - lambda) * to_be_selected
```

```
while(in_salient--):
      select random point from list_salient_selectable
while(in_non_salient--)
      select random point from list_non_salient_selectable
```

This **F** value is used for random vertex selection to choose points more uniformly across the whole grid. The selected points are then used in the next step of vertex optimisation to distribute these points in a manner that suits the image better.

INPUT - edgeSegments.data + douglasPoints.data + salientPoints.data

OUTPUT - randomSeeds.data + RandomSeeds.png + DistanceMap.png + FeatureMap.png



(a) Distance Map (b)Feature Flow Field

Figure 9. $F(x)$ and $D(x)$ plots

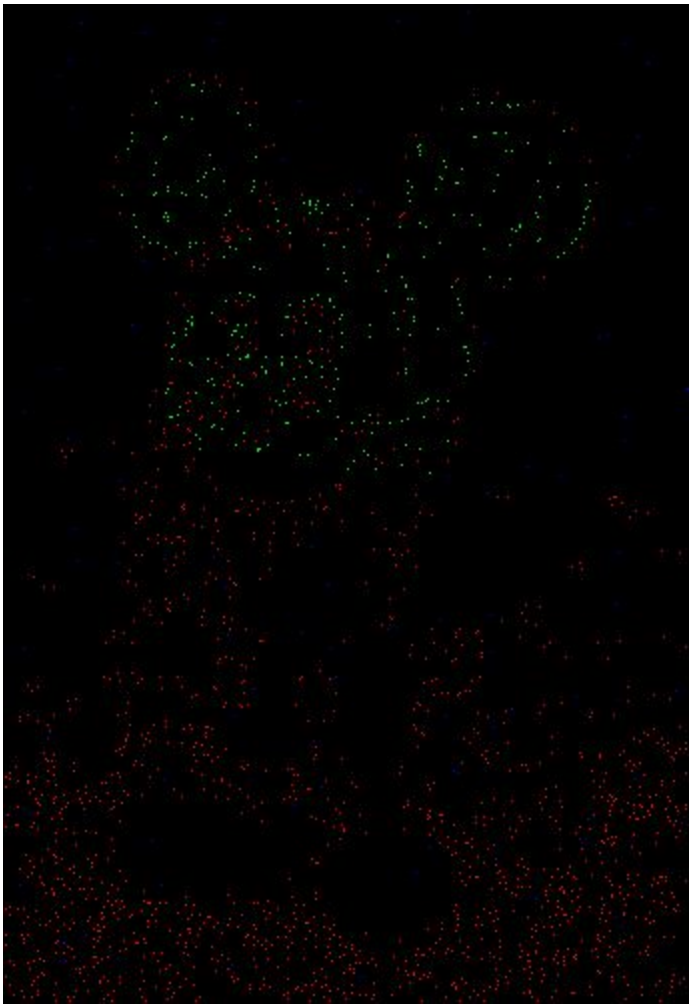


Figure 10. Random Seeds (RandomSeeds.png)

Douglas points are plotted as red, background points as blue and foreground points as green in the above random seeds plot.

(g) Vertex Optimisation

The CVT and Lloyd relaxation are often used to optimize the mesh quality. During the iteration, each seed is moved to its regions centroid. The centroid **c** of a Voronoi Cell can be calculated as follows :

$$c = \frac{\sum_{\mathrm{i}} w_{\mathrm{i}} x_{\mathrm{i}}}{\sum_{\mathrm{i}} w_{\mathrm{i}}},$$

For calculating the centroid the

$$w_i = 1$$

but for our case we take

$$w_i = \frac{F(x_i)}{255}$$

and as $F(x_i)$ is very less near to the edge points and thus more weight is assigned to farther points thus moving the centroid point far away from the edges. This ensures that the sampling vertices will not occlude the constrained points from Douglas Peucker. Vertex optimisation works by doing the following for every seed :

$$v_i^{(k+1)} \leftarrow \frac{\sum_i w_i x_i}{\sum_i w_i}$$

where $v_i^{(k+1)}$ is the position of i^{th} vertex after $(k+1)^{th}$ iteration. This is repeated for 5 iterations in our case with the w_i value as specified above. This gives the input points for the triangulation.

After optimising the seed points and writing the output to optimisedSeeds.data **IOFormat.cpp** is run to generate a PSLG(Planar Straight Line Graph) which will serve as in input for the triangulation algorithm in the next step. It generates triangle.poly file which has points as well as edges which serve as a constraint for Constrained Delaunay Triangulation.

INPUT - randomSeeds.data + douglasPoints.data + douglasSegments.data

OUTPUT - optimisedSeeds.data + OptimisedSeeds.png + triangle.poly

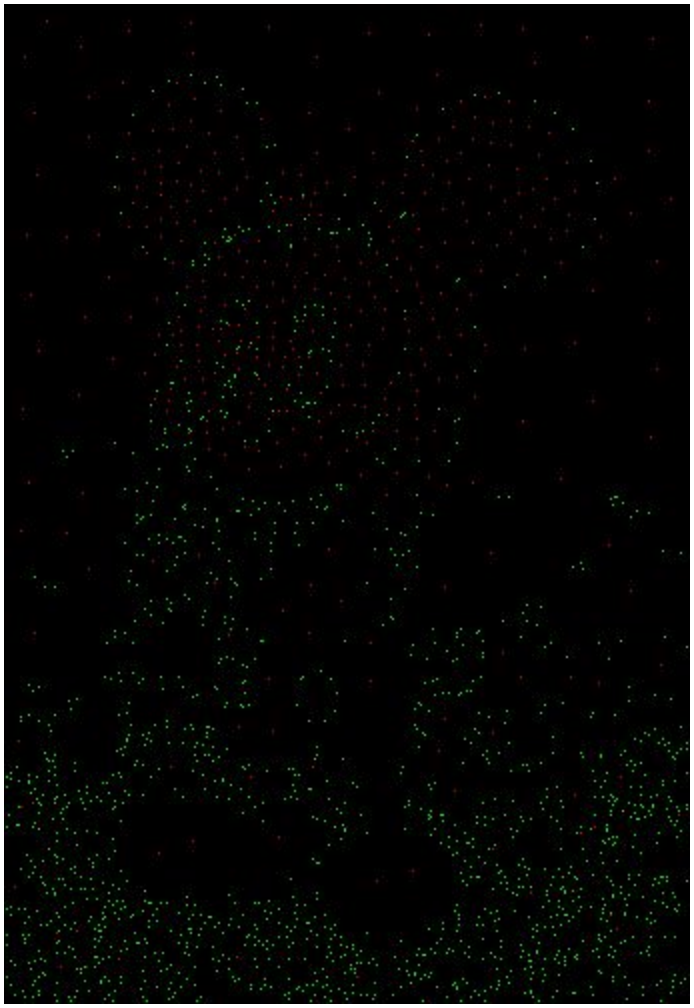


Figure 11. Optimised Seeds (OptimisedSeeds.png)

Green marks are the output points of douglas peucker algorithm and all the other foreground and background points are colored red after optimisation using Llyod relaxation.

(h) Constrained Delaunay Triangulation

Delaunay triangulation for a given set **P** of discrete points in a plane is a triangulation **DT(P)** such that no point in **P** is inside the circumcircle of any triangle in **DT(P)**. Delaunay triangulations maximize the minimum angle of all the angles of the triangles in the triangulation. Constrained Delaunay triangulation is the one, in which certain

edges are also passed as an input argument to the algorithm along with the points, and the triangulation is constrained to have those edges as a part of triangulation edges. **DT(P)** is also simply a dual graph of Voronoi Tessellation of circumcenters of all triangles in **DT(P)**. Given 4 points, we can make 2 non-overlapping triangles from the points, say ABC and DBC. Now, if

$$\angle A + \angle D \leq 180^\circ$$

the triangulation is delaunay otherwise flip the common edge from BC to AD and by simple geometry we can deduce that the triangulation becomes delaunay. This is the same condition as to checking whether the 4th point lies in the circumcircle of the first three points. So, **flip** operation can help us attain delaunay triangulation for any given set of points.

The basic algorithm to approach this problem the incremental approach where you keep on adding one point to the triangulation at every interval and making the resulting mesh triangulated by flipping to satisfy the delaunay criteria. After understanding the algorithm, the triangle [7] library by CMU, was used to generate constrained delaunay triangulation. It takes a PSLG (Planar Straight Line Graph) as input in the form of .poly file which has a list of all the points and line segments and then return a .node file containing all the vertices of the triangulation, .poly file containing all the edges in triangulation, and a .ele file containing all the triangles, thus triangulating the given set of points.

INPUT - triangle.poly

OUTPUT - triangle.1.node + triangle.1.poly + triangle.1.ele

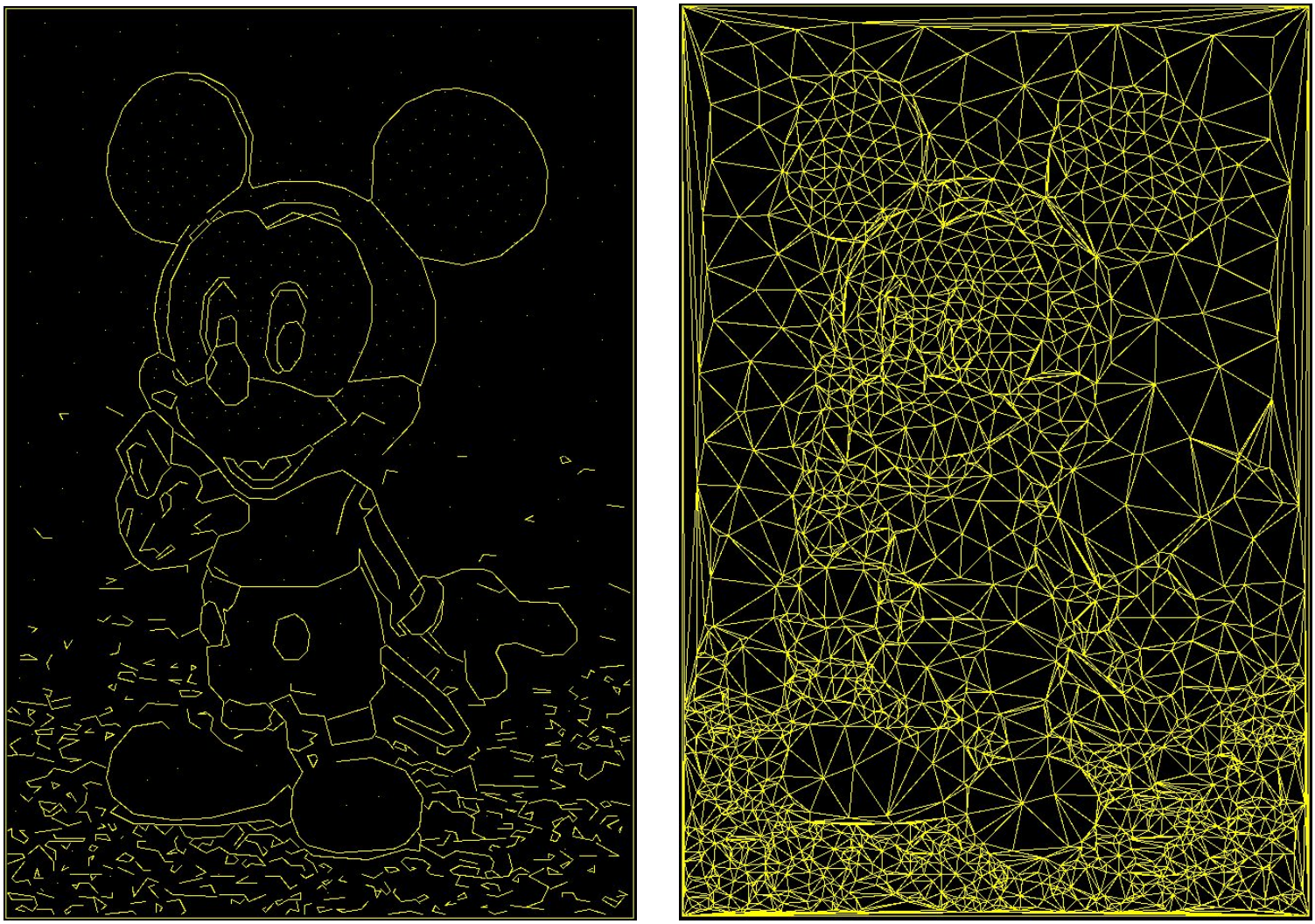


Figure 12.Input and output of triangulation (triangle.poly and triangle.1.ele)

(i) Color Post Processing for triangle coloring

This module initially uses the original.png generated by the run script to generate a data file consisting of RGB values for each of the pixel. Then this generated data file along with the triangulation output is used to take out the median value of each of RGB for all points in the triangle and this median is the new color value for all the points in the triangle.

The key part in this module is to compute the points inside of the triangle for taking the median. To achieve this, an iteration is made from minimum y-coordinate value to

maximum y-coordinate value each time ascertaining the two x values for the given y such that (x1, y) and (x2, y) lie on the boundary. Thus, all the points on the line segment joining (x1, y) and (x2, y) are in the triangle.

Completing the entire iteration over the range of y we go over each point in the grid exactly once and thus the time complexity is **O(n)** where n is the number of points on the grid. After that, coloring of image according to the median colours, is done by OpenCV in python to obtain the final PNG image.

INPUT - original.png + triangle.1.ele + triangle.1.node

OUTPUT - originalColors.data + finalColors.data + final.png

Figure 13. Final output image (final.png)

7. Results

The algorithm was run on a variety of input images with different characteristics and objects like flowers, people, cars and the results are shown below.

Figure 14.

Figure 15.

Figure 16.

Figure 17.

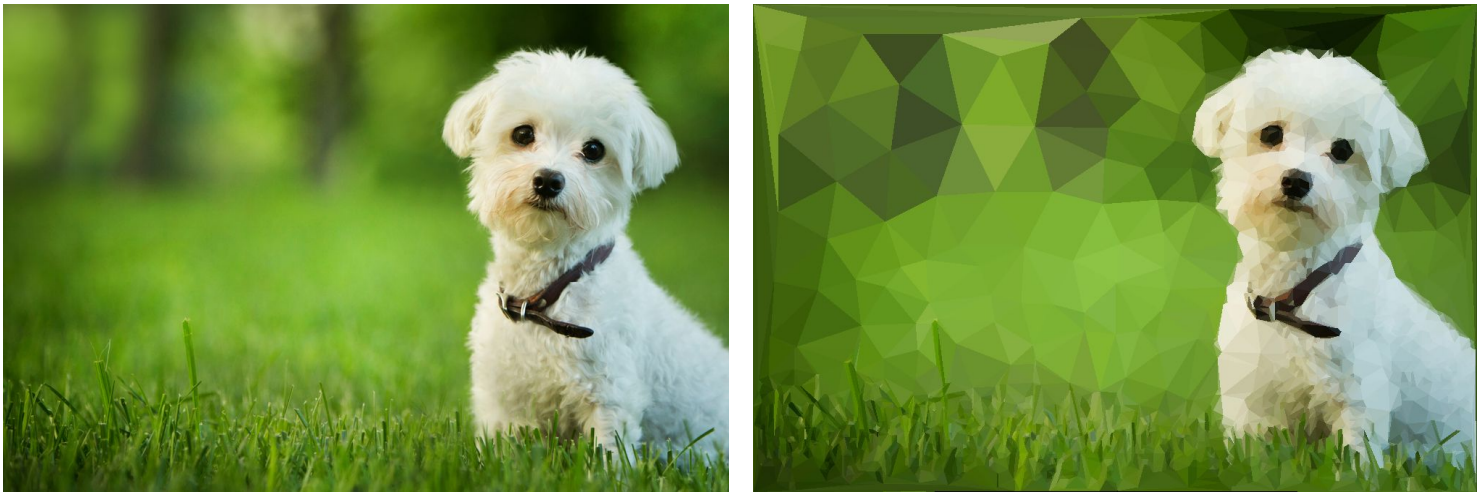


Figure 18.

8. References

[1] : Gai, M. & Wang, G. Vis Comput, 2016: Artistic Low Poly Rendering for images in *Springer Berlin Heidelberg*

*[*2]: Y.Yang, M.N. Wernick, J.G. Brankov, 2003: A fast approach for accurate content-adaptive mesh generation in *IEEE Transactions on Image Processing*

[3]: Image triangulator app that produces a non content aware triangulation result: http://www.conceptfarm.ca/2013/portfolio/image-triangulator/

[4]: Cihan Topal, Cuneyt Akinlar, 2012 : Edge Drawing: A combined real-time edge and segment detector in *Journal of Visual Communication and Image Representation*.

[5] : John Hershberger and Jack Snoeyink, 1994: An O(nlogn) implementation of the Douglas-Peucker algorithm for line simplification in *SCG '94 ACM, New York*.

[6]: Guodong Rong, Tiow-Seng Tan, 2006: Jump flooding in GPU with applications to Voronoi diagram and distance transform in *I3D '06*

[7]: A Two-Dimensional Quality Mesh Generator and Delaunay Triangulator library: https://www.cs.cmu.edu/~quake/triangle.html