Final Project
ME112: Introduction to Matlab
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY
May 1 – May 26, 2022, Shenzhen, Guangdong, China

ME112 - INTRODUCTION TO MATLAB

LOW-POLY STYLE PICTURE REALIZATION

Liu Leqi[1], Li Peiru[1]

[1]Southern University of Science and Technology, Shenzhen, Guangdong, China

ABSTRACT

Low-poly design was once widely used in early computer modeling and animation due to the technical limits. Although the technology has a considerable development, after a period of high-poly prevalence, some designers wanted to eliminate the interfering factors and represent the picture more abstract but concise so that to deliver the valid information to users. Besides, some technologies used in low-poly generation are also influence the development of other field.

Keywords: Low-poly, Image preprocessing

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. Artists use adaptive polygons (mostly triangles) to represent the objects in a image to get a specific abstract visual effect. Now It has been a new design trend in the artist community. In this paper, we focus on generating low-poly rendering for images automatically by MATLAB.

2. FRAMEWORK

2.1 GRAY PICTURE

To make it convenience to conduct after and speed up the calculation, we first need to transform the original image from color to gray. However, if the original image is already gray, it will no need to transform. Here we used a built-in function of MATLAB named $rgb2gray()$.

2.2 EDGE DETECTION

Edge detection is a practical method to partition the image based on the sudden change areas of image grey value. According to the gray section, the edge detection model can be categorized to be three kinds as Figure 2, 3 and 4.

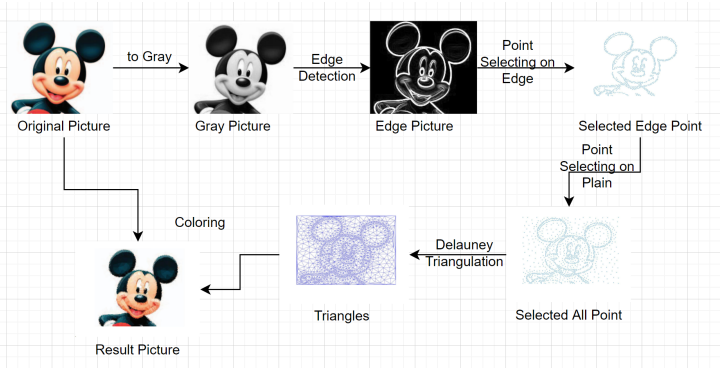

FIGURE 1: MAIN FRAMEWORK

FIGURE 2: STAGE MODEL

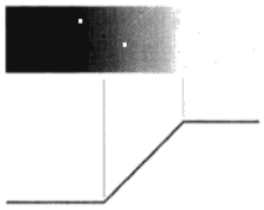

FIGURE 3: RAMP MODEL

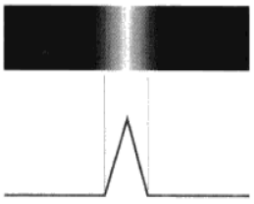

FIGURE 4: ROOF MODEL

The pixels on the edges are expected to contain more detail information of the original image. In edge detection, the density of information in the original image will be represented.
During edge detection, we will not use the built-in function of MATLAB such as $edge()$ or $imfilter()$ since these functions

will return 0-1 matrix (black-white matrix), which is inconvenient to handle the detail, instead of gray matrix. (See Figure 7 and 8) Thus, we implemented an edge detection function by using a 3×3 Sobel kernel.

$$S_x = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \tag{1}$$

$$S_y = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \tag{2}$$

$$G_x = S_x * A \tag{3}$$

$$G_y = S_y * A \tag{4}$$

$$G = \sqrt{G_x^2 + G_y^2} \tag{5}$$

where A is the original image (after transformed to gray).

FIGURE 5: ORIGINAL IMAGE

FIGURE 6: GRAY IMAGE

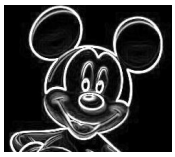

FIGURE 7: SOBEL EDGE

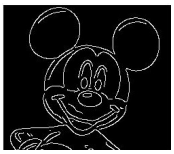

FIGURE 8: EDGE DETECTION USING BUILT-IN FUNCTION

2.3 POISSON DISK SAMPLING

During sampling, we need to select the pixel points randomly so that the triangles in result image can distribute randomly. However, if we only use random sampling, the generating triangles will be shape and hardly satisfy the smooth edge so that it will break the picture. Therefore, we chose Poisson Disk Sampling, a method to sample random points evenly in a degree, making the generating triangles more uniform.

FIGURE 9: COMPARISON OF THREE SAMPLING METHODS

Since the detail on the edge and on the plain has a huge difference, it needs to be sampled twice for one on the edge first and the other except edge based on the sampling edge points. (See Figure 10 and 11) Then the distribution of the sample points can be distinguished to be two categories.

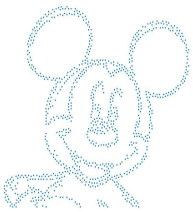

FIGURE 10: SAMPLING ON EDGE

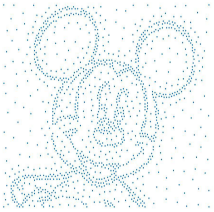

FIGURE 11: SAMPLING ON PLAIN

The procedure of Poisson Disk Sampling is as following. Here we used the Fast Poisson Disk Sampling algorithm proposed by Robert Bridson.

1. The first point is randomly chosen, and put in the output list, processing list.
2. Until the processing list is empty, do the following:
   (a) Choose a random point from the processing list.
   (b) For this point, generate up to k points (here we chose k=30), randomly selected from the annulus surrounding the point. For every generated point:
      i. Check for points that are too close to this point.
      ii. If there is none, add the point to the output list, processing list.
   (c) Remove the point from the processing list.
3. Return the output list as the sample points.

2.4 DELAUNEY TRIANGULATION

Triangulation involves creating from the sample points a set of non-overlapping triangularly bounded facets, the vertices of the triangles are the input sample points. One of the popular algorithm is Delauney triangulation.
The Delauney triangulation is closely related geometrically to the Direchlet tessellation, also known as the Voronoi or Theissen tesselations. It splits the plane into a number of polygonal regions called tiles. Each tile has one sample point in its interior called a generating point. All other points inside the polygonal tile are closer to the generating point than to any other. The Delauney triangulation is created by connecting all generating points which share a common tile edge.

Here we used a built-in function of MATLAB named $delaunay()$.

2.5 Coloring

We simply used the color in the original image at the mass center of each triangle to be the triangle's color.

$$Color(T_{abc}) = I(\frac{x_a + x_b + x_c}{3}, \frac{y_a + y_b + y_c}{3})$$

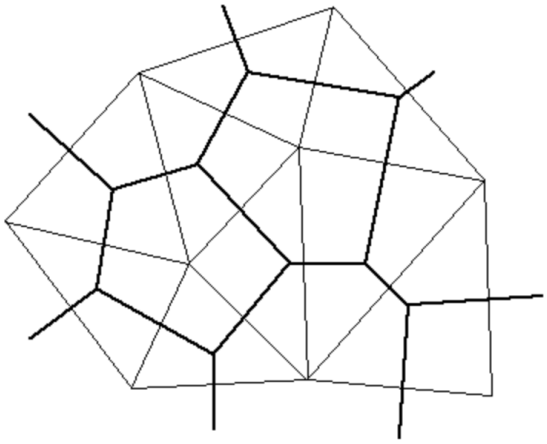

FIGURE 12: DELAUNEY TRIANGLES (THIN LINES) AND ASSOCIATED DIRECHLET TESSELATIONS (THICK LINES) FOR NINE GENERATING POINTS. TRIANGLE EDGES ARE PERPENDICULAR BISECTORS OF THE TILE EDGES. POINTS WITHIN A TILE ARE CLOSER TO THE TILE'S GENERATING POINT THAN TO ANY OTHER GENERATING POINT.

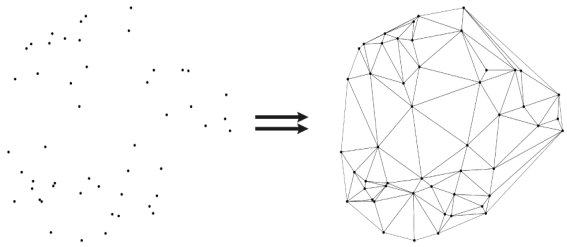

FIGURE 13: DELAUNEY TRIANGULATION

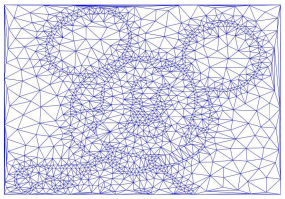

FIGURE 14: DELAUNEY RESULT

FIGURE 15: FINAL RESULT

3. CONCLUSION

In this project, we implemented the low-poly style image generation from the original image. During the implementation, we have learnt some knowledge about image processing and computer graphics.

In the future, we can try to implement it by using other languages such as *C++*, *C#* and so on. Besides, we can try to use hardware to accelerate the program.

ACKNOWLEDGMENTS

Thanks for the instruction of Dr. Wei, teaching us how to use MATLAB. Thanks for the authors of the papers *Artistic Low Poly rendering for images* and *Low-poly style image and video processing*. Our project is based on their papers.

REFERENCES

[1] Gai, Meng and Wang, Guoping. "Artistic Low Poly rendering for images." *The Visual Computer* Vol. 32 No. 4 (2016): pp. 491–500. DOI 10.1007/s00371-015-1082-2. URL https://doi.org/10.1007/s00371-015-1082-2.

[2] Zhang, Wenli, Xiao, Shuangjiu and Shi, Xin. "Low-poly style image and video processing." *2015 International Conference on Systems, Signals and Image Processing (IWSSIP)*: pp. 97–100. 2015. DOI 10.1109/IWSSIP.2015.7314186.

[3] Tulleken, Herman. "Poisson Disk Sampling." http://devmag.org.za/2009/05/03/poisson-disk-sampling/ Accessed May 20, 2022.

[4] Bourke, Paul. "Triangulate Efficient Triangulation Algorithm Suitable for Terrain Modelling or An Algorithm for Interpolating Irregularly-Spaced Data with Applications in Terrain Modelling." URL http://paulbourke.net/papers/triangulate/.

[5] zhiyishou. "Triangulation Algorithm (Delauney)." https://www.cnblogs.com/zhiyishou/p/4430017.html Accessed May 20, 2022.

APPENDIX A. MATLAB SCRIPT OF MAIN PART

```
clear all; clc;

Origin = imread('t4.png');
if size(Origin,3)==3
    Gray = rgb2gray(Origin);
else
    Gray = Origin;
end

Boundary = sobel_detect(Gray);

edge = Boundary;
max_val = max(max(edge))*0.4;
[edgeX, edgeY] = find(edge >= max_val);

Boundary_size = size(Boundary);

r_edge = min(Boundary_size)/80;

```

```
r_max = min(Boundary_size)/20;
r_min = min(Boundary_size)/40;


edge_point = poisson_edge([edgeY, edgeX],
    r_edge);
internal_point = poisson_disk(Boundary,
    edge_point, r_min, r_max);

Delaunay_result = delaunay(internal_point
    (:,1), internal_point(:,2));

vector1 = internal_point(Delaunay_result
    (:,1), :);
vector2 = internal_point(Delaunay_result
    (:,2), :);
vector3 = internal_point(Delaunay_result
    (:,3), :);
center = round((vector1+vector2+vector3)
    ./3);
tmp_list = center(:,2)+(center(:,1)-1)*
    size(Origin,1);
if size(Origin,3) == 3
    Rchannel = Origin(:,:,1);
    Gchannel = Origin(:,:,2);
    Bchannel = Origin(:,:,3);
    colorList(:,:,1) = Rchannel(tmp_list);
    colorList(:,:,2) = Gchannel(tmp_list);
    colorList(:,:,3) = Bchannel(tmp_list);
else
    colorList(:,:,1) = Origin(tmp_list);
    colorList(:,:,2) = Origin(tmp_list);
    colorList(:,:,3) = Origin(tmp_list);
end


z=zeros([size(internal_point,1),1]);
trisurf(Delaunay_result,internal_point
    (:,1),internal_point(:,2),z,'CData',
    colorList,'EdgeColor','none')
hold on;
set(gca,'XTick',[],'YTick',[],'XColor','
    none','YColor','none')
axis equal;
set(gca,'YDir','reverse','View',[0,90])
```

APPENDIX B. MATLAB SCRIPT OF FUNCTIONS

```
function output_list = poisson_edge(
    point_set, r)
    random_index_initial = randi([1,size(
        point_set,1)],1);
    random_point_initial = point_set(
        random_index_initial,:);
    output_list = random_point_initial;
    processing_list = point_set;

    K = 30;

    while ~isempty(processing_list)
        random_index = randi([1,size(
            processing_list,1)],1);
        random_point = processing_list(
            random_index,:);

        distance = sqrt(sum((point_set -
            random_point).^2,2));
        neighbour = find(r<=distance &
            distance<2*r);

        if length(neighbour) > K
            detect_point = point_set(
                neighbour(1:K),:);
        else
            detect_point = point_set(
                neighbour,:);
        end

        for index = 1:size(detect_point,1)
            tmp_point = detect_point(index
                ,:);
            tmp_distance = sqrt(sum((
                output_list-tmp_point)
                .^2,2));
            if all(tmp_distance >= r)
                output_list = [output_list
                    ; tmp_point];
                processing_list = [
                    processing_list;
                    tmp_point];
            end
        end
        processing_list(random_index,:) =
            [];
    end
end

function output_list = poisson_disk(
    Boundary, edge_point, r_min, r_max)
    [rows, cols] = size(Boundary);

    output_list = [edge_point; 1 1; 1 rows
        ; cols 1; cols rows];
    processing_list = edge_point;

    change = double(255 -Boundary);
    cmin = min(min(change));
    cmax = max(max(change));
    rpoint = change - cmin;
    rpoint = rpoint./(cmax-cmin).*(r_max-
        r_min)+r_min;

```

```
    K = 30;

    while ~isempty(processing_list)
        random_index = randi([1,size(
            processing_list,1)],1);
        random_point = processing_list(
            random_index,:);
        ra = rpoint(round(random_point(2))
            , round(random_point(1)));
        theta = rand(K,1)*2*pi;
        radius = (rand(K,1)+1)*ra;
        x = radius.*cos(theta)+
            random_point(1);
        y = radius.*sin(theta)+
            random_point(2);

        for i = 1:K
            point = [x(i), y(i)];
            if point(1)>=1 && point(2)>=1
               && point(1)<=cols && point
               (2)<=rows
                distance = sqrt(sum((
                    output_list-point)
                    .^2,2));
                if all(distance >= ra)
                    output_list = [
                        output_list;point];
                    processing_list = [
                        processing_list;
                        point];
                end
            end
        end
        processing_list(random_index,:) =
            [];
    end
end
```

```
function boundary = sobel_detect(Gray)
    sobelx = [-1 -2 -1; 0 0 0; 1 2 1];
    sobely = [-1 0 1; -2 0 2; -1 0 1];

    [rows, cols] = size(Gray);
    tmp = zeros(rows+2, cols+2);


    tmp(2:rows+1,2:cols+1) = Gray;
    tmp(1,2:cols+1) = Gray(1,:);
    tmp(2:rows+1,1) = Gray(:,1);
    tmp(rows+2,2:cols+1) = Gray(rows,:);
    tmp(2:rows+1,cols+2) = Gray(:,cols);
    tmp(1,1) = Gray(1,1);
    tmp(rows+2,cols+2) = Gray(rows,cols);
    tmp(rows+2,1) = Gray(rows,1);
    tmp(1,cols+2) = Gray(1,cols);


    Gx = zeros(rows, cols);
    Gy = zeros(rows, cols);

    for i = 1:3
        for j = 1:3
            t = tmp(i:rows+i-1,j:cols+j-1)
                ;
            Gx = Gx + t.*sobelx(i,j);
            Gy = Gy + t.*sobely(i,j);
        end
    end

    boundary = uint8(sqrt(Gx.^2 + Gy.^2));
end
```