Triangle Filling

Gitopoulos Giorgos, 9344

Computer Graphics @ ECE AUTh, March 2021

Abstract

Triangles are among the most common polygons used in **Computer Graphics**, due to their simple structure and some significant properties. Thus, triangles are used very often to represent graphic displays. This project, deals with **triangle filling** - determining the color of each pixel inside a certain triangle. Given an input of triangles (vertices and vertex colors) that represent an image, we apply our triangle filling algorithm to fill all of these triangles and display the image.

1. Introduction

The code of the project was written in *Matlab* and can be found here **here**. There are two versions of the triangle filling algorithm: **Flat** and **Gouraud**. Both of them can be tested by running the respective demo file (*demo_flat*, *demo_gouraud*). Each demo file, after loading the image workspace (all the triangles that constitute the image, including the colors of their vertices and the depth of each vertex) calls the *render* function with the appropriate parameter *renderer* (1 for *Flat* or 2 for *Gouraud*).

The *render* function, computes the depth of every triangle (as the mean of the depths of its vertices) and sorts the triangles by descending depth. Afterwards, the proper paint function is executed for each triangle (*paint_triange_flat* or *paint_triange_gouraud*), starting from the deepest triangle and implements the algorithm that will be described in [2.1] to paint it. These two functions return the updated image after filling the new triangle, while the *render* function returns the final image.

Also, two helping functions were implemented: *vector_interp*, which computes the linear interpolation of two given vectors and *paint_line_gouraud*, which fills a specific line of the triangle in the second version. Both are used in *Gouraud* version.

2. Algorithm implementation

2.1. Description

In this section, the presented **triangle filling algorithm** (version 1 and 2) will be described.

2.1.1. Flat version

In the first version, supposing that the image is a $M \times N$ array, while x determines the column and y the row number of a specific pixel, the implemented algorithm is the following.

Algorithm 1 Flat triangle filling

$color \leftarrow mean(vertex\_color)$
$edges \leftarrow LIST\_EDGES(vertices\_2d)$
$y\_min \leftarrow min(edges.y\_min)$
$y\_max \leftarrow max(edges.y\_max)$
for $y = y\_min : y\_max$ **do**
  $UPDATE\_ACTIVE\_X1(active\_x1, active\_edges)$
  $UPDATE\_ACTIVE\_X2(active\_x2, active\_edges)$
  $active\_x\_min \leftarrow round(min(active\_x1, active\_x2) + 0.5)$
  $active\_x\_max \leftarrow round(max(active\_x1, active\_x2) + 0.5)$
  for $x = active\_x\_min : active\_x\_max$ **do**
    $PAINT\_PIXEL(Img, x, y, color)$
  end for
  for $e\ in\ active\_edges$ **do**
    if $e.y\_max = y$ **then**
      $UPDATE\_ACTIVE\_EDGES(active\_edges, y)$
    end if
  end for
end for

Firstly, we compute the triangle color and we use a data structure to store the edges. We compute the lower and upper bounds of rows and then we pass each row of the triangle, starting from the lowest. We initialize the $active\_x1$ and $active\_x2$ values to the x value of the lowest vertex. The special case where y_min row contains an edge, is handled in a different way. Also, *active_edges* are stored and updated when an edge finishes. *Active_edges* determine the new $active\_x1$ and $active\_x2$ values, that are updated recursively, using the gradient of the current $active\_edges$. So, in each row, we colorize the pixel that are between $active\_x1$ and $active\_x2$ with the precomputed color. In the case of a lowest horizontal edge, we just colorize the whole edge, initialize properly the *active edges* and *points* and execute the same procedure.

Figure 1: *Triangle filling with Flat version (left) and Gouraud version (right). Vertex colors are red, green and blue.*

2.1.2. Gouraud version

The second version is completely the same with the first concerning the pixel passing procedure, but differs in the way that the color is computed. Now, each pixel has a unique color, that is defined by its position inside the triangle. The vertices of the triangle are painted in the beginning. Our purpose, is to soften the color differences between the given triangles of the image and make the result more realistic. So, we use a **vector interpolation** function to paint, at first, the *active points* of each row, with respect to their *vertical distances* of the two vertices of the current *active edge*. Subsequently, we pass the pixels between the *active points* and using again the **vector interpolation** to colorize them with respect to their *horizontal distances* from the *active points*. The algorithm is presented below.

Algorithm 2 Gouraud triangle filling

for $v \ in \ vertices\_2d$ **do**
  $PAINT\_PIXEL(Img, v, vertex\_colors)$
end for
$edges \leftarrow LIST\_EDGES(vertices\_2d)$
$y\_min \leftarrow min(edges.y\_min)$
$y\_max \leftarrow max(edges.y\_max)$
for $y = y\_min : y\_max$ **do**
  $UPDATE\_ACTIVE\_X1(active\_x1, active\_edges)$
  $UPDATE\_ACTIVE\_X2(active\_x2, active\_edges)$
  $active\_x1\_color \leftarrow vect\_interp(y, active\_edges(1))$
  $active\_x2\_color \leftarrow vect\_interp(y, active\_edges(2))$
  $active\_x\_min \leftarrow round(min(active\_x1, active\_x2) + 0.5)$
  $active\_x\_max \leftarrow round(max(active\_x1, active\_x2) + 0.5)$
  for $x = active\_x\_min : active\_x\_max$ **do**
    $color \leftarrow vect\_interp(x, active\_x1, active\_x2, active\_x1\_color, active\_x2\_color)$
    $PAINT\_PIXEL(Img, x, y, color)$
  end for
  for $e \ in \ active\_edges$ **do**
    if $e.y\_max = y$ **then**
      $UPDATE\_ACTIVE\_EDGES(active\_edges, y)$
    end if
  end for
end for

2.2. Assumptions

We assume that the given vertices are inside the image bounds. Before drawing a pixel $Img(x, y)$, we round x values in order to fit the image pixels (need to be *integers*) - y values are already *integers*. The case that a given triangle is not a real triangle (e.g. two same vertices) is handled and rejected. Also, executing the *vector_interp* function, if the given points have equal coordinates in the given dimension, we randomly return the color of the first point to avoid an infinite λ value. In addition, to avoid confusing indexing, we swap the horizontal with the vertical coordinates for pixel indexing, while in *Matlab* the first coordinate refers to the rows and the second to columns. Thus, we swap the two columns of the input *vertices_2d* array (swap x and y coordinates) to fix the angle for each triangle. The performance was tested and the time overhead was negligible.

3. Results

Running the two demo files of our project, using as input a *racoon* image, the output images are the following.

Figure 2: *Flat triangle filling result*

Figure 3: *Gouraud triangle filling result*

As it was expected, the *Flat* version, produces an image in which the initial triangles are visible, as each one has a standard color. On the other hand, the triangle limits are not visible in the *Gouraud* version, where the result has a smooth display.