Introduction to Computer Graphics & Polygon Filling

Petridis Konstantinos

Computer Graphics @ ECE AUTh, April 2022

Abstract

This paper discusses a **Polygon Filling** algorithm implementation. The analysis entails detailed explanation of the strategies utilized, extensive description of the algorithms followed, clarification of every single step of the computational process and exhibition of the results. All implementations are done in Python and performance is boosted by applying vectorization techniques using the **Numpy** library.

1. Introduction

Computer Graphics is a sub-field of **Computer Science**, whose primary subject is the examination of how a human perception can be visualized, displayed and manipulated into a visualization medium. An object that is required to be visualized, goes through some stages that briefly include its mathematical description (coordinates, visual characteristics, properties etc), its placement into a coordinate system named scene (3D space), its projection to the two dimensional space and finally the actual display. To be able to display any object properly, we first have to divide it into elementary polygons which, which cover the entire surface of the object, and then color each polygon separately. Our project concerns the coloring of polygons, specifically triangles, which makes the implementation more simplistic and conceivable, since triangles are convex shapes. The reason for this is going to be clear after the illustration of the algorithms used.

2. Problem Definition

Given a $M \times N \times 3$ matrix, initialized with aces (white image), perform color filling according to some input matrices, which provide all the necessary information about the image. The input matrices are the following:

- **verts2d**: $L \times 2$ matrix containing the coordinates of every vertex. L vertices in total.
- **vcolors**: $L \times 3$ matrix containing the RGB colors of every vertex. L vertices in total.
- **faces**: $K \times 3$ matrix containing the indices from the vertices of every triangle. K triangles in total.
- **depth**:$L \times 1$ vector containing the depth of every vertex, as calculated before the projection from 3D space, to 2D. L vertices in total.

3. Problem Decomposition

3.1. Access Order

The first step, is obviously figuring out the correct order, in which the triangles shall be filled. The order can be determined by the reasonable and self-evident realization that, triangles that were the farthest inside the 3D scene and before the projection to the 2D space should be colored first. For this approach, we take advantage of the **depth** and **faces** matrices to obtain the order of the triangles, from that with the largest depth value, to the one with the smallest. In more detail, we create a new $K \times 1$ vector containing the depth of every triangle, which is computed as the mean of the depth values, from the vertices of the triangle. After this, we sort the new matrix in descending order, since we should access the triangles with the largest depth first. The method is summarized in the following sub-routine.

Algorithm 1 Triangle ordering based on depth

procedure TRIANGLESINORDER(faces, depth)
  $depth\_tr \leftarrow depth(faces)$
  $ordered\_tr \leftarrow sort(depth\_tr,' DESC')$
  return $ordered\_tr$

3.2. Triangle Limits

Since now we acquire the order, we start iterating over every triangle. The idea is to perform a **vertical scan** to each triangle, starting from the bottom. Then, for every horizontal line y of the vertical scan, we should perform a **horizontal scan**, to be able to paint the whole triangle. It is directly comprehensible that, the computation of the **limits** of each edge of the triangle, is a major prerequisite for this process to be fulfilled. So now, for the current triangle i, we compute $x_{ik,\min}$, $x_{ik,\max}$, $y_{ik,\min}$, $y_{ik,\max}$, $i \in [0, K-1]$ and $k = 1, 2, 3$. $x_{ik,\min}$ denotes the minimum abscissa for the edge k of triangle i. Now, we have gathered all the necessary information to continue our reasoning.

Algorithm 2 Triangle limits computation

procedure COMPUTELIMITS(verts2d)
  $edge\_verts \leftarrow$ vertex coordinates of each edge
  $x\_limits \leftarrow x_{\min}, x_{\max}$ of each edge
  $y\_limits \leftarrow y_{\min}, y_{\max}$ of each edge
  $edges\_sigma \leftarrow$ slope of each edge
  return $x\_limits, y\_limits, edges\_sigma$

3.3. Active Elements

At this point, we are examining the i^{th} triangle, and we have computed the limits for both the abscissa and the ordinate, of all three edges. We start the vertical scanning, starting from the bottom of the triangle, which is denoted as

$$y_{\min} = \min_k \{x_{ik,\min}\}$$

and stoping at

$$y_{\max} = \max_k \{x_{ik,\max}\}$$

At each step of the vertical scan, the edges that are crossed by the horizontal line y, are considered as **active edges**. The dis-

tinct points on which the line y intersects the **active edges**, are considered as **active vertices**.

During an arbitrary iteration of the vertical scan, let y be the ordinate of the horizontal scan-line. The idea is that, if there is an edge of this triangle, such that $y_{ik,\min} = y$, then this line is placed into the active edges. This condition is always going to be meet during the first iteration of the vertical scan, when $y = y_{\min}$. It should be declared at this point, that if an edge is **horizontal**, we immediately exclude it from the active edges, because it has infinite intersection points with the scan-line. This case is illustrated in *Figure 1*, as the vertical scan-line (red line) meets the lowest horizontal edge. The points marked with black color are the active vertices. Notice how we take into consideration only the vertices, incident to the non-horizontal edges.

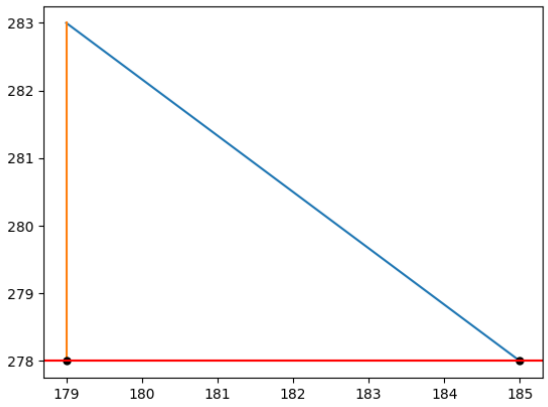

Figure 1: *Vertical scan meets horizontal edge*

During iteration $i + 1$ of the vertical scan, we update the **active vertices** using the following formula, for an active vertex v_i, $i = 1, 2, 3$.

$$\begin{pmatrix} x_{i+1} & y_{i+1} \end{pmatrix} = \begin{pmatrix} x_i + \dfrac{1}{m_k} & y_i + 1 \end{pmatrix} \quad (1)$$

where m_k denotes the slope of the edge k, which is the edge, to which the **active vertex** of interest belongs. Our implementation examines this part, taking into account four, distinct cases.

1. $\mathbf{m_k = 0}$: This is the case of a horizontal edge, which is either encountered at the very beginning of the vertical scan, or the end. The first case is already treated, by not considering it a **active edge**, thus, there is no **active vertex** to update and we don't run into any computational issues. In the latter, a sub-routine is called to paint the horizontal line, by performing a horizontal scan on it, coloring every point in the range $[x_{\min}, x_{\max}]$
2. $\mathbf{m_k = \infty}$: This is the case of vertical edge. By utilizing the (1) updating formula, we obtain the desired result, even in this particular case, since when the edge is vertical, we only update the ordinate.
3. $\mathbf{m_k = nan}$: In some triangles, two of the vertices happen to belong to the same pixel, which occurred after the projection of the object from the 3D scene, to the 2D space. The edge formed by such two vertices has undefined slope. In this case, another sub-routine is called to paint only the visible edge of the triangle.
4. **Other**: Every other case is treated by the formula (1)

Cases *1, 2, 4* are all depicted in *Figure 2-5*, while case 3 is displayed by *Figure 5*. The procedure to locate the initial active elements is presented in *Algorithm 3*, while *Algorithm 4* illustrates the updating of both **active edges** and **active vertices**

Algorithm 3 Initial active elements computation

```
1:  procedure INITACTIVEELEMENTS
2:      y_min ← minimum ordinate of the triangle
3:      y_max ← maximum ordinate of the triangle
4:      for i in indices of the triangle edges do
5:          if max(y_limit[i]) = y_max then
6:              if edge_sigma[i] = 0 then ▷ Upper horizontal
7:                  color the edge
8:          if min(y_limit[i]) = y_min then
9:              if edge_sigma[i] = 0 then ▷ Lower horizontal
10:                 continue
11:             if edges_sigma[i] = nan then ▷ Non visible
12:                 Color the visible edge only
13:             active_edges ← edges[i]
14:             active_verts ← edge vertex with y = y_min
15:     return active_edges, active_verts, img
```

Algorithm 4 Active elements updating

```
1:  procedure UPDACTIVEELEMENTS
2:      updated = [ ]
3:      Y ← ordinate of the vertical-scan line
4:      for i in indices of the triangle edges do ▷ Edge update
5:          if min(y_limit[i]) = Y then ▷ Meet new edge
6:              active_edges ← edges[i]
7:              active_verts ← edge vertex with y = Y
8:              updated ← verts[i]
9:          if max(y_limit[i]) = Y − 1 then ▷ Leave edge
10:             exclude edges[i] from active_edges
11:     idx = 0
12:     for s in edge_sigma do
13:         if edge active and s ≠ 0 and idx ∉ updated then
14:             update active vertices, based on (1)
            idx = idx + 1
15:     return active_edges, active_verts
```

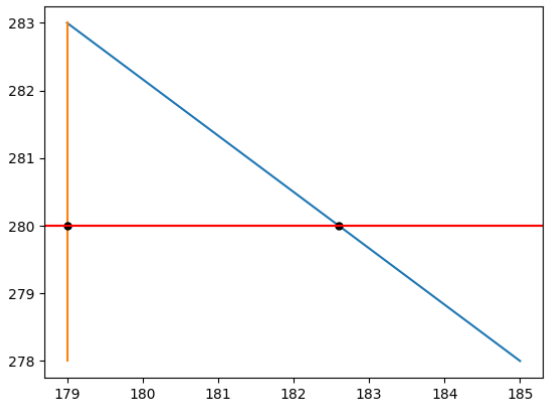

Figure 2: *Updating active vertices when* $m_1 = 0$, $m_2 = \infty$ *and* $m_3 < 0$

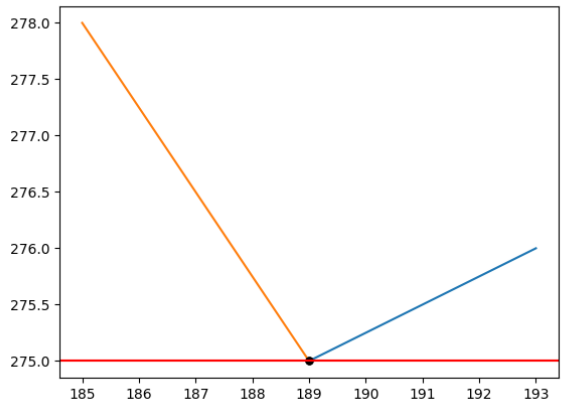

Figure 3: *Updating active vertices when* $m_i \in [0, 1]$, *iter=0*

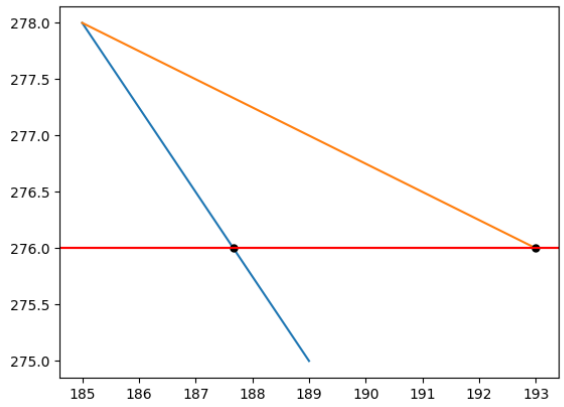

Figure 4: *Updating active vertices when* $m_i \in [0, 1]$, *iter=1*

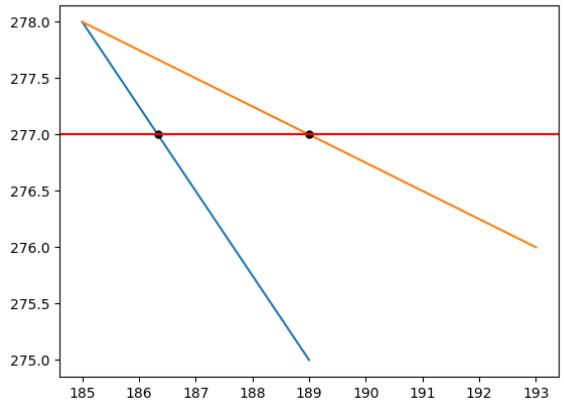

Figure 5: *Updating active vertices when* $m_i \in [0, 1]$, *iter=2*

3.4. Horizontal scan

Currently, we are in the middle of a vertical scan for a specific triangle, so for the horizontal line y, we want to perform a **horizontal scan** and determine which points belong inside the triangle. Those who satisfy this condition are going to be filled with color.

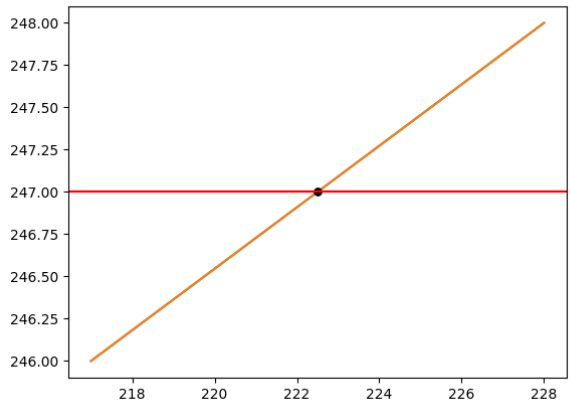

Figure 6: *Updating active vertices when* $m_i = nan$ *for some i*

The method is the following. We scan inside the range $[x_{\min} - x_{\max}]$ and we keep track of the occurrences with **active vertices**. We name this variable $cross\_count$. For some $x \in [x_{\min} - x_{\max}]$, of the horizontal line y, (x, y) belongs to the triangle only if $cross\_count$ is an odd number, which is consequent of the convexity of the triangular shape. For every point (x, y) that satisfies the preconditions mentioned, the coloring sub-routine is called and assigns the new color, using the method of choice ('flat' or 'gouraud'). The exact method is depicted below, in *Algorithm 5*

Algorithm 5 Horizontally scanning a line y

```
1: procedure HSCAN
2:     cross_counter = 0
3:     for x = x_min : x_max do
4:         if x equals the abscissa of an active vertex then
5:             cross_counter = cross_counter + 1
6:         if cross_counter is odd then
7:             draw pixel (x, y)
8:     return img
```

3.5. The Complete Algorithm

Now that we have decomposed the initial problem into smaller and more simple sub-problems, which can be easily resolved, we are able to appropriately combine the routines and construct the final algorithm. The complete process is described in *Algorithm 6*, which is the implementation for **flat rendering**. The **gouraud rendering** is almost identical, with the exception that, there is the extra need to compute the colors of the active vertices, via the color interpolation of the color vectors from the triangle vertices. An additional method is inserted to *Algorithm 6*, in order to resolve the extra requirement, producing *Algorithm 7*.

Algorithm 6 Render using a single color for each triangle

procedure RENDERFLAT
  $verts2d, vcolors, faces, depth \leftarrow load\_data()$
  $ordered\_tr \leftarrow$ TRIANGLESINORDER()
  for t in $ordered\_tr$ **do**
    $verts2d\_tr \leftarrow verts2d[faces[t]]$
    $vcolors\_tr \leftarrow vcolors[faces[t]]$
    $color = mean(vcolors\_tr)$
    $x\_limits, y\_limits, s \leftarrow$ COMPUTELIMITS()
    $e\_act, v\_act, img \leftarrow$ INITACTIVEELEMENTS()
    for $y = y_{\min} : y_{max}$ **do**
      $e\_act, v\_act \leftarrow$ UPDACTIVEELEMENTS()
      $img \leftarrow$ HSCAN()
  return

Algorithm 7 Render using interpolate coloring

procedure RENDERGOURAUD
  $verts2d, vcolors, faces, depth \leftarrow load\_data()$
  $ordered\_tr \leftarrow$ TRIANGLESINORDER()
  for t in $ordered\_tr$ **do**
    $verts2d\_tr \leftarrow verts2d[faces[t]]$
    $vcolors\_tr \leftarrow vcolors[faces[t]]$
    $color = mean(vcolors\_tr)$
    $x\_limits, y\_limits, s \leftarrow$ COMPUTELIMITS()
    $e\_act, v\_act, img \leftarrow$ INITACTIVEELEMENTS()
    for $y = y_{\min} : y_{max}$ **do**
      $e\_act, v\_act \leftarrow$ UPDACTIVEELEMENTS()
      $img, v\_act\_color \leftarrow$ ACTIVEVERTSCOLOR()
      $img \leftarrow$ HSCAN()
  return

4. Conventions

In order to effectively render the image, conventions about the approach to treat some distinct but significant cases should be made. A brief reference to such strategies is present in the previous sections, however it's a topic deserving of attention, and definitely its own separate section. Let's break down the method to face each extremist case.

- Given a triangle, if there is an **upper horizontal edge** with ordinate $y = y_{\max}$, then the edge is colored.
- If a triangle has one vertex headed towards axis z, which is **not visible** in 2D space, we color only the visible edge, thus a single line
- If a triangle seems to be **contracted into a single distinct point** (all vertices fall to the same pixel), we just color the pixel of interest.
- During a vertical scan of a triangle, if a **scan line** Y **intersects a vertex which is incident to two edges**, we have three cases:
  1. $\mathbf{y} = \mathbf{y}_{\min}$ where we consider two points of intersection, one per edge.
  2. $\mathbf{y} = \mathbf{y}_{\max}$ where there is no active edge, by default so there is no active vertex either.
  3. $\mathbf{y} \in (\mathbf{y}_{\min}, \mathbf{y}_{\max})$ where it is considered as a single point of intersection. This happens because, as explained before, for a given scan-line Y, if $y_{i,\max} = Y$ for an edge i of the triangle, then the edge is excluded from the active edges, so even if it's in contact with the scan-line, there is no **active vertex** belonging to this specific edge.

Case 1 and *2* of the last bullet point, are illustrated in *Figure 7* and *8* respectively. Notice how, in *Figure 8*, one triangle edge is not appearing even when it is contacting the **active vertex**. That's because the exclusion condition $y_{i,\max} = Y$ is satisfied for that edge. *Figure 9* depicts *Case 2*. None of the edges shown are active, as explained above. They are only shown for visualization purposes.

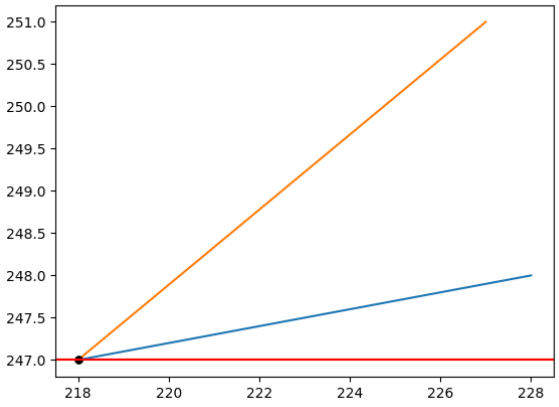

Figure 7: *Scan line intersects vertex, incident to two edges and* $y = y_{\min}$

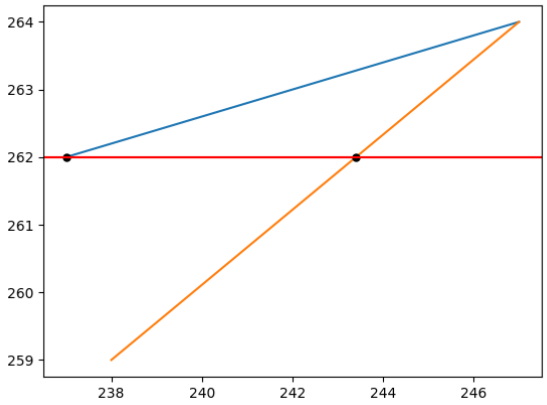

Figure 8: *Scan line intersects vertex, incident to two edges and* $y \in (y_{\min}, y_{\max})$

5. Results

Now that the analysis is complete and the algorithmic procedure has been formed, It's time to exhibit the results from the testing. The expected outcome should be fairly satisfying, as all the possible cases have been treated accordingly.

5.1. Flat Rendering

In this section, the results of **flat rendering** are displayed. The algorithm used is *Algorithm 6*, which assigns to every triangle, the mean color of its three vertices. The image produced by **Flat Rendering**, is depicted in *Figure 10*. It seems that the outcome confirms the validity of our theoretical analysis. No visible imperfections seem to exist (empty spaces, white dots, outreaching lines etc), so it is safe to assume that we treated every possible case.

One thing to notice is that, observing *Figure 10* closely, the existence of **triangular shapes** on the surface of the object becomes apparent. This is completely expected, since we assigned one single color to each of the triangles, so it's consequent that the contour of some triangles is considerably prominent.

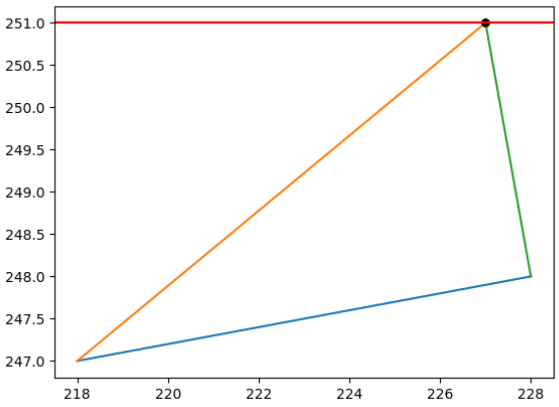

Figure 9: *Scan line intersects vertex, incident to two edges and* $y = y_{\min}$

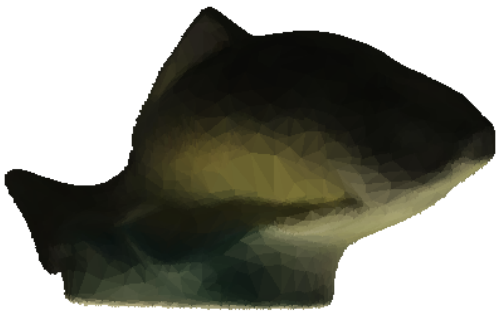

Figure 10: *Image after flat rendering*

5.2. Gouraud Rendering

In order to extinguish the appearance of triangles on the surface of the object, we use *Algorithm 7*, choosing a different color for each pixel, which results from interpolating the colors of two extremist vertices. This approach allows for a more smooth and **gradual transition of the color** throughout the entire object. The major drawback of this method, is the heavy computational work that's being added to the original version of the algorithm. This cannot be completely avoided, but can be minimized after applying method-related optimizations. *Figure 11* presents the final result. Indeed the color is more smoothly transitioned, and almost no triangles are visible.

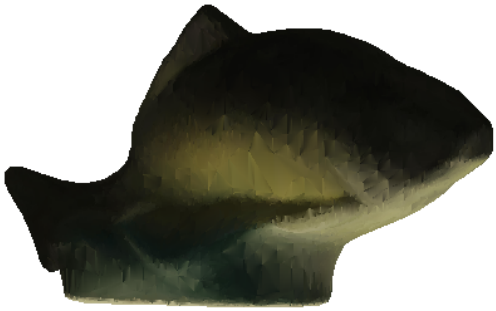

Figure 11: *Image after gouraud rendering*

6. Code Instructions

There are two **demo** files called, **demo_flat.py** and **demo_gouraud.py** under the 'src' directory. Run the rendering method of choice, by executing one of the two files and the result is going to be saved under the 'results' directory. All functions used for reading, coloring and general computational assist, are located inside 'inc' directory. Finally, the file containing all the necessary data is placed under the 'data' directory. Everything is set to run automatically and it's not a necessity for the user to interfere. Be sure to check out the **README** for direct instructions as well. The main library used is **Numpy** for the computations and data structures that it provides. There is a plan about enhancing the performance of the implementation, by using **CuPy**, in order to take advantage of the **GPU** speed.