Triangle Filling in Computer Graphics

Dimitrios Karatis 10775, *Electrical and Computer Engineering, AUTH*

***Abstract*—This project implements and compares different triangle filling algorithms in computer graphics, focusing on both flat shading and texture mapping approaches. The implementation includes a rendering function that handles depth sorting for proper object visualization. The project demonstrates fundamental computer graphics techniques using Python with NumPy for efficient array operations, providing insights into rasterization processes and color interpolation methods essential for 3D graphics rendering.**

***Index Terms*—computer graphics, triangle filling, rasterization, flat shading, texture mapping, linear interpolation, depth sorting**

A. `vector_interp` function

The `vector_interp` function performs linear interpolation between two vectors, V1 and V2, which are defined at two distinct spatial points p1 and p2. The goal is to estimate the interpolated vector at a given coordinate along either the x-axis (dim=1) or y-axis (dim=2). The function first checks which dimension to interpolate over, calculates the corresponding scalar parameter t based on the relative distance between the coordinate and the two points, and then uses t to linearly interpolate between V1 and V2. To ensure numerical stability, it also handles the edge case where the two points lie on the same axis value (e.g., same x or y), by returning one of the original vectors directly.

Below we can see the pseudocode for the `vector_interp` function:

Algorithm 1 Vector Interpolation Function

function VECTOR_INTERP(p_1, p_2, V_1, V_2, coord, dim)
  if dim $= 1$ **then**
    denominator $\leftarrow p_2.x - p_1.x$
  else if dim $= 2$ **then**
    denominator $\leftarrow p_2.y - p_1.y$
  else
    return Error
  end if
  if $|\text{denominator}| < \epsilon$ **then** $\{\epsilon$ is a small threshold$\}$
    return V_1
  end if
  if dim $= 1$ **then**
    $t \leftarrow \dfrac{\text{coord} - p_1.x}{\text{denominator}}$
  else
    $t \leftarrow \dfrac{\text{coord} - p_1.y}{\text{denominator}}$
  end if
  $V \leftarrow (1 - t) \cdot V_1 + t \cdot V_2$
  return V
end function=0

Detailed code for this function can be found in the vector_interp.py file.

B. `f_shading` function

The `f_shading` function applies flat shading to a triangle defined by three vertices in an image. It begins by calculating the average color of the triangle's three vertex colors to be used as the flat fill color. The function then determines the bounding box of the triangle to limit the region over which it will operate. A grid of pixel coordinates within this bounding box is created, and each pixel is tested to determine whether it lies inside the triangle using a vectorized implementation of the edge function (based on 2D cross products). Pixels determined to be inside the triangle are filled with the flat color.

Below we can see the pseudocode for the `f_shading` function:

This document was prepared by Dimitrios Karatis as part of the first assignment for the Computer Graphics course at Aristotle University of Thessaloniki.

Algorithm 2 Flat Shading Function

function F_SHADING(img, vertices, vcolors)
  updated_img ← copy of img
  triangle ← first two components (x, y) of each vertex in vertices
  flat_color ← average of vcolors along rows
  min_x ← max(floor(min(triangle[:, 0])), 0)
  max_x ← min(ceil(max(triangle[:, 0])), img width - 1)
  min_y ← max(floor(min(triangle[:, 1])), 0)
  max_y ← min(ceil(max(triangle[:, 1])), img height - 1)
  xs, ys ← meshgrid of pixel coordinates from (min_x, min_y) to (max_x, max_y)
  points ← stack(xs, ys) into 2D array of pixel positions
  A, B, C ← vertices of triangle
  function CROSS(a, b, c)
    return $(b_x - a_x)(c_y - a_y) - (b_y - a_y)(c_x - a_x)$
  end function
  d1 ← cross(A, B, points)
  d2 ← cross(B, C, points)
  d3 ← cross(C, A, points)
  has_neg ← (d1 $<$ 0) OR (d2 $<$ 0) OR (d3 $<$ 0)
  has_pos ← (d1 $>$ 0) OR (d2 $>$ 0) OR (d3 $>$ 0)
  inside ← NOT (has_neg AND has_pos)
  for each (x, y) in points where inside is true **do**
    updated_img[y, x] ← flat_color
  end for
  return updated_img
end function=0

In order to test this function the script ***f_shading_test.py*** was created. This script demonstrates the use of a flat shading technique on a triangle drawn within a 100x100 white image. It begins by creating a blank image and defining the triangle through its 2D vertices and corresponding RGB colors (red, green, and blue). The `f_shading` function is then applied to fill the triangle area with the average color of its vertices. The script visualizes the original triangle outline with its colored vertices on the left and the shaded triangle result on the right using `Matplotlib`. Finally, the script saves the comparison as an image file (`test_f.png`) and displays it. *(To ensure the image is saved correctly, the user may need to specify a custom path where they want the file to be stored.)*

Below we can see the `test_f.png` image, which seems to be correct: The triangle is filled with a flat color, which is the average of the vertex colors (red, green, blue). The average of $[1, 0, 0]$, $[0, 1, 0]$, and $[0, 0, 1]$ is $[0.33, 0.33, 0.33]$, resulting in a grayish color.

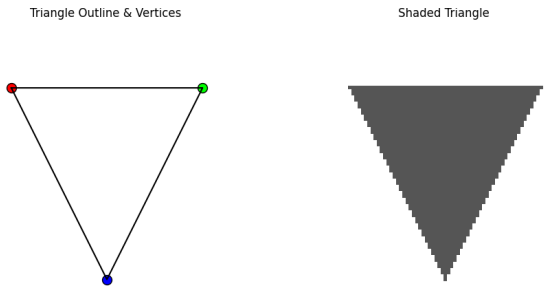

Fig. 1: f_shading test

Detailed code for this function can be found in the f_shading.py file.

C. `t_shading` function

The `t_shading` function implements texture mapping by rasterizing a triangle and interpolating UV coordinates across its surface. First, it sorts the triangle's vertices by their y-coordinates to facilitate scanline traversal. For each horizontal scanline intersecting the triangle, the function computes two points: one along the left edge and one along the right edge of the triangle, along with their corresponding UV coordinates. Then, for each pixel between these two points, it performs a linear interpolation of the UV values. These UV coordinates are mapped to the corresponding pixel in the texture image, which is sampled and used to color the pixel in the output image. The method effectively wraps a 2D texture onto a 3D triangle using perspective-correct interpolation, using the `vector_interp` function from before.

Below we can see the pseudocode for the `t_shading` function:

Algorithm 3 Texture Shading Function

function T_SHADING(img, vertices, uv, textImg)
  Convert `vertices` and `uv` to float
  Sort `vertices` by y-coordinate; reorder `uv` accordingly
  Assign sorted vertices to `C1, C2, C3` and corresponding UVs to `uv1, uv2, uv3`
  Determine vertical scanline range: `y_min` to `y_max`
  for each scanline `y` from `y_min` to `y_max` **do**
    if `y < C2.y` **then** {Top half of triangle}
      Interpolate point A and UV_A between `C1` and `C2`
    else{Bottom half}
      Interpolate point A and UV_A between `C2` and `C3`
    end if
    Interpolate point B and UV_B between `C1` and `C3`
    if `A.x > B.x` **then** {Ensure left-to-right direction}
      Swap `A` with `B`, and `UV_A` with `UV_B`
    end if
    Determine horizontal pixel range: `x_min` to `x_max`
    for each pixel `x` from `x_min` to `x_max` **do**
      Interpolate UV_P between `A` and `B`
      if `UV_P` is invalid (NaN or outside [0, 1]) **then**
        Continue to next pixel
      end if
      Map `UV_P` to texture coordinates `tex_x, tex_y`
      Sample color from `textImg[tex_y, tex_x]` and write to `img[y, x]`
    end for
  end for
  return modified `img`
end function=0

In order to test this function the script ***t_shading_test.py*** was created. This script begins by creating a blank 400×400 RGB image, which serves as the rendering canvas. It then defines a triangle using three 2D vertices and associates each vertex with a UV coordinate in the normalized texture space $[0, 1]^2$. A texture image is loaded from file, and the `t_shading` function is invoked to apply the texture onto the triangle area of the blank canvas based on the UV coordinates. For visualization purposes, the script also displays the texture image with the triangle's UV coordinates outlined, alongside the resulting image where the textured triangle has been rendered. The final comparison is shown side-by-side using `Matplotlib`, and the output is saved as an image file (`test_t.png`) for reference.
(To ensure the image is saved correctly, the user may need to specify a custom path where they want the file to be stored.)

Below we can see the `test_t.png` image, which seems to be correct: The textured triangle in the rendered image accurately matches the region defined by the UV coordinates on the texture image. The shape, orientation, and content within the triangle are consistent. There are no visible artifacts, and the texture is sampled within bounds.

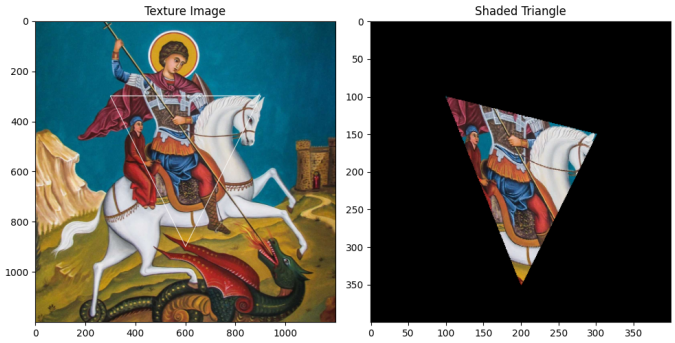

Fig. 2: t_shading test

Detailed code for this function can be found in the t_shading.py file.

D. `render_img` function

The `render_img` function is responsible for rendering a 3D scene on a 2D canvas. The function takes as input the triangle faces, vertex coordinates, vertex colors, UV coordinates, depth values, shading type, and texture image. It first initializes a blank canvas (a white image).

The function calculates the average depth for each triangle by averaging the depths of its vertices. Then, it sorts the triangles based on their depth, in descending order. This sorting ensures that the farther triangles are rendered first, which is a common approach for correct depth handling in rendering.

For each triangle, the function extracts its 2D coordinates, colors, and UV mapping (for texture shading). Depending on the specified shading type ('f' for flat shading or 't' for texture shading), it applies either the flat shading (`f_shading`) or texture shading (`t_shading`) function to the image.

The function prints debug information to track the triangles being processed and ensures that shading is correctly applied. Once all triangles are rendered, it returns the final image with the applied shading.

Below we can see the pseudocode for the `render_img` function:

Algorithm 4 Render a 3D Scene with Shading

function RENDER_IMG(faces, vertices, vcolors, uvs, depth, shading, texImg)
Initialize a white image canvas of size 512×512
Compute average depth for each triangle using the depth of its vertices
Sort triangles in descending order based on average depth
Reorder faces using the sorted depth indices
for each triangle in the sorted face list **do**
  Get vertex indices for the triangle
  Extract 2D positions, vertex colors, and UV coordinates
  if shading == 'f' **then**
    Apply flat shading using F_SHADING(img, verts_2d, colors)
  else if shading == 't' **then**
    Apply texture shading using T_SHADING(img, verts_2d, uv_coords, texImg)
  else
    raise error "Shading must be either 'f' or 't"
  end if
end for
return img
end function=0

Detailed code for this function can be found in the render_img.py file.

E. `demo_f.py` *and* `demo_g.py`

The `demo_f` and `demo_g` functions load the data required for texture mapping and shading of a 3D object. The functions load a texture image, `texImg.jpg`, and normalizes it to the range [0, 1] if its pixel values are in the range [0, 255].

Next, the functions load 3D object data from the file `hw1.npy`, which includes vertex positions, vertex colors, texture coordinates (UVs), and depth values. The texture is mapped onto the 3D object using a face index array. The `render_img` function is then called with the appropriate arguments to apply *F shading* or *T shading* respectfully. The resulting image is then displayed with `matplotlib` and saved as `render_f.png` or `render_g.png` respectfully. *(To ensure the image is saved correctly, the user may need to specify a custom path where they want the file to be stored.)*

Below we can see the `render_f.png` and `render_g.png` images:

Fig. 3: demo_f output

Fig. 4: demo_g output

Detailed code for these functions can be found in the demo_f.py and demo_g.py files.