Ptolemy Linden
CSCI 411
Algorithm Report

CPU-Based Triangle Rasterization Using the Scanline Algorithm

Introduction

This report discusses the computer graphics technique of triangle rasterization. Triangle rasterization is not a single algorithm by itself, but rather a combination of many sub-algorithms that come together to allow us to render pixels on a screen in the shapes of triangles. The "core" algorithm is the Scanline Algorithm, which is the focus of this report. However, we'll also briefly cover the sub-algorithms required for triangle rasterization to happen, including Bresenham's Line algorithm.

What is Triangle Rasterization?

Triangle rasterization is the process of rendering triangles on a two-dimensional grid of pixels. The grid of pixels is called a "raster" and the process of "rasterization" is the drawing and rendering of images using those pixels. Why do we care so much about triangles? The reason is that triangles are the simplest polygon; we can render them extremely quickly and efficiently. Additionally, all possible shapes and polygons can be formed from triangles. This means that we can efficiently render any possible shape. In 3D graphics, every surface (from characters to terrain) is approximated as a mesh of triangles, making triangle rasterization the fundamental building block of rendering.

CPU vs GPU Rasterization

The original triangle rasterization algorithms were developed using CPUs, as GPUs did not yet exist. CPU Scanline algorithms have the benefit of being much simpler to write and implement, but they're extremely slow and inefficient. GPUs are far better at performing these rendering tasks and any modern triangle rasterization is performed on GPUs. For this report, we'll be focusing on the CPU-based approach.

The Overall Approach

The rasterization process happens in several stages:

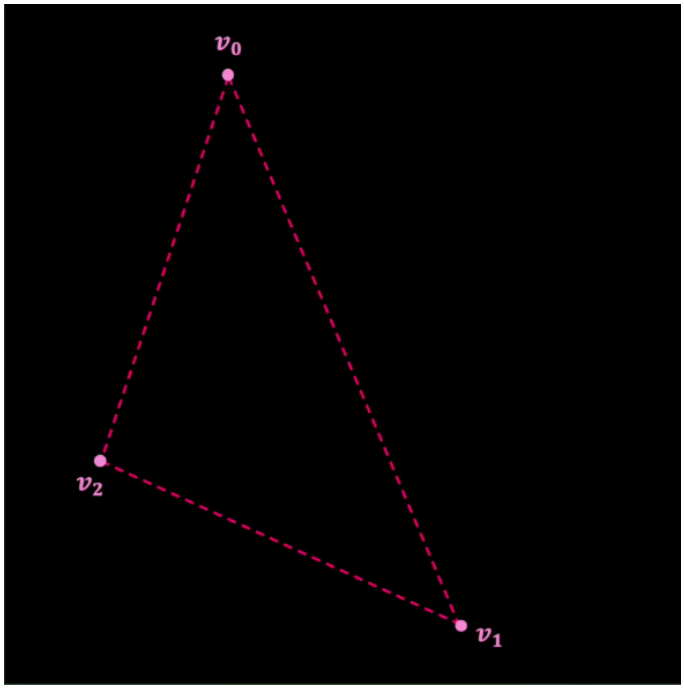
1. Sort the three vertices by their Y coordinate (top to bottom)
2. Calculate the edge pixels of the triangle using Bresenham's Line algorithm
3. Use the Scanline Algorithm to fill the interior
4. Interpolate colors smoothly across the triangle

By using the Scanline Algorithm, we can process the triangle one horizontal row (scanline) at a time, filling pixels from left to right.

Bresenham's Line Algorithm

Before we can fill the triangle, we need to draw its edges. Bresenham's Algorithm allows us to efficiently draw edges between the three vertices. Note the use of the word "efficiently." We could naively draw the edges by simply calculating the hypotenuse between two vertices and drawing that, but this requires floating-point arithmetic, which is slow at large scales. Bresenham's Line algorithm cleverly removes the need for any floating-point calculations by the CPU. Its implementation is complex and beyond the scope of this report, but suffice to say it's the necessary first step before we can use the Scanline Algorithm.

Figure 1)



The Scanline Algorithm

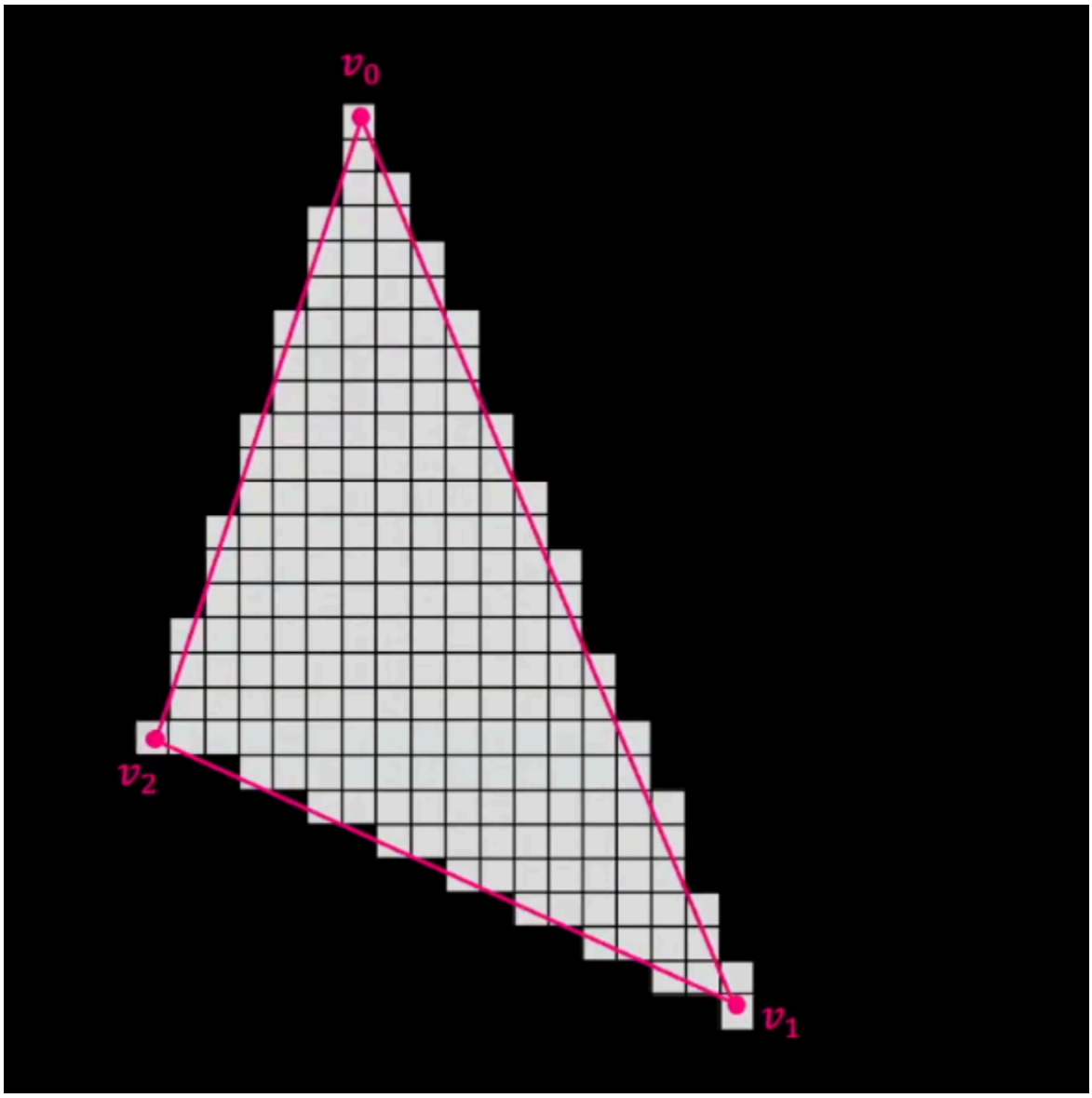
This is the core of triangle rasterization and the focus of this report. The algorithm works as follows:

1. Sort vertices by Y coordinate. We ensure v0 is the topmost vertex, v2 is the bottom and v1 is in the middle.
2. Divide the triangle into two halves. Every triangle can be split horizontally at the middle vertex into a top half (from v0 down to v1) and a bottom half (from v1 down to v2)
3. Process each scanline (horizontal row of pixels).
   For each Y coordinate from top to bottom:
   a) Calculate where the current scanline intersects the triangle's edges
   b) This gives us two X coordinates: the left and right boundaries
   c) Fill all pixels horizontally between these boundaries
4. Track two edges at once
   For each scanline, we track:
   a) The "long edge" from v0 to v2 (spans the entire triangle height)
   b) The "short edge" that switches at the middle vertex:
   * Top half: v0 to v1
   * Bottom half: v1 to v2

To find where each edge intersects the current scanline, we use linear interpolation based on how far down the edge we've traveled (parameter t, ranging from 0.0 to 1.0)

The time complexity of this algorithm is O(height x width), where height and width refer to the triangle's bounding box dimensions. Each pixel is visited exactly once.

Figure 2)



Pseudocode

```
function fillTriangle(v0, v1, v2):
    // Step 1: Sort vertices by Y coordinate (top to bottom)
    sort v0, v1, v2 such that v0.y <= v1.y <= v2.y

    // Step 2: Handle degenerate case
    if v0.y == v2.y:
        return  // Flat line, not a triangle

    // Step 3: Process each horizontal scanline
    for y from v0.y to v2.y:
        // Determine which half of triangle we're in
        if y < v1.y:
            // Top half: short edge is v0 → v1
            v_start = v0
            v_end = v1
        else:
            // Bottom half: short edge is v1 → v2
            v_start = v1
            v_end = v2

        // Calculate interpolation parameter for long edge (v0 → v2)
        t_long = (y - v0.y) / (v2.y - v0.y)
        x_long = v0.x + (v2.x - v0.x) * t_long
        color_long = interpolate(v0.color, v2.color, t_long)

        // Calculate interpolation parameter for short edge
        t_short = (y - v_start.y) / (v_end.y - v_start.y)
        x_short = v_start.x + (v_end.x - v_start.x) * t_short
        color_short = interpolate(v_start.color, v_end.color, t_short)

        // Determine left and right boundaries
        if x_long < x_short:
            x_left = x_long
            x_right = x_short
            color_left = color_long
            color_right = color_short
        else:
            x_left = x_short
            x_right = x_long
            color_left = color_short
            color_right = color_long

        // Fill horizontal span
        for x from x_left to x_right:
            t_span = (x - x_left) / (x_right - x_left)
            color = interpolate(color_left, color_right, t_span)
            setPixel(x, y, color)
```

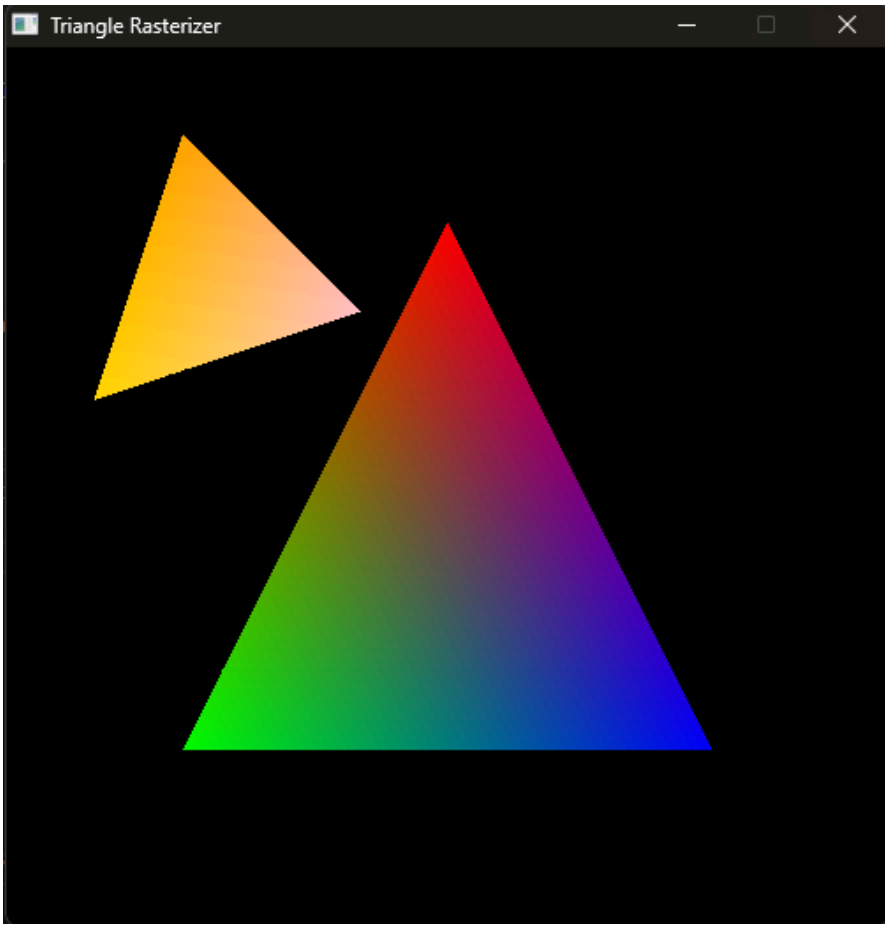
Notes

This pseudocode (as well as the code in my implementation) make use of color interpolation. This isn't strictly necessary for the algorithm to work; we could remove color interpolation and just have solid-colored triangles. But that's boring and we like pretty colors, so we've implemented it here.

Figures 1 and 2 are screenshots from "Triangle Rasterization" on Youtube, uploaded by Pikuma. Figure 3 is a screenshot of the output from Default Mode in my Triangle Rasterizer program found here: https://github.com/Adolin42/CPU-based-Triangle-Rasterizer

The Final Result

Figure 3)



References

Linden, Ptolemy "CPU-based Triangle Rasterizer" *Github*, https://github.com/Adolin42/CPU-based-Triangle-Rasterizer

“Triangle Rasterization” *YouTube,* uploaded by pikuma, 23 March 2023, https://www.youtube.com/watch?v=k5wtuKWmV48&t=931s.

Wikipedia contributors. "Bresenham's line algorithm." *Wikipedia*, *The Free Encyclopedia.* 9 November 2025. https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm