SUNG-EUI YOON, KAIST

RENDERING

FREELY AVAILABLE ON THE INTERNET

http://sglab.kaist.ac.kr/~sungeui/render

First printing, July 2018

7
Rasterization

The main idea of rasterization is to project a triangle into the view space and rasterize it into fragments in the color and depth buffers. In this chapter, we assume that vertices of the triangle are projected into the view space, after they undergo various transformations, followed by clipping and NDC transformation.

7.1 Primitive Rasterization

Rasterization is optimized for processing triangles thanks to their simplicity.

For the rasterization process, we commonly use triangles as input primitives, mainly because it is the simplest polygon and simplifies the rasterization process. Nonetheless, these other representations are also decomposed into a set of triangles and fed into the rasterization process.

Rasterization process has two main goals: 1) pixel coverage determination (Fig. 7.1) and 2) parameter interpolation (Fig. 7.5). Given a pixel of the color buffer (or other buffers), we determine whether the pixel belongs to a given triangle or not. Once the pixel is covered by the triangle, we also need to compute its color or other parameters such as its depth value for the depth buffer.

For the coverage problems, many directions are possible. One is to check whether the center of a pixel is inside of a triangle. Another is to measure an area coverage ratio of a pixel against the triangle. The first one is based on a point sample, while the latter one is based on area computation. While the area based computation is more correct, the sample based approach is more efficient, and thus is commonly adopted for rasterization process. They share common pros and cons between point sample based and area based approaches, as we discussed for image-space and object-space methods (Ch. 5.2).

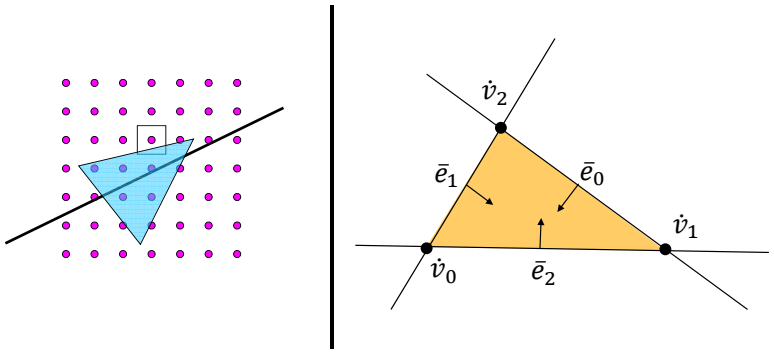

Figure 7.1: The left shows one, pixel coverage determination, of two main goals of the rasterization process. The right shows configurations of vertices and edges of a triangle used for our discussion.

Scanline based triangle rasterization. Some of early techniques for rasterization are based on a concept of scanline, a row of pixels that span a triangle (Fig. 7.2). At those days, the memory was very expensive, and thus having the full resolution of color and depth buffers is not preferred. Instead, these scaneline based approaches maintain a scanline and incrementally update the scanline to raster the whole triangle. Specially, we rasterize an input triangle from top to bottom. Once we meet a vertex of the triangle, we setup the scanline information (e.g., starting and end coordinates shown as red pixels in the figure). For the next scanline, we incrementally update those starting and end coordinates by utilizing slope information of two edges starting from the vertex.

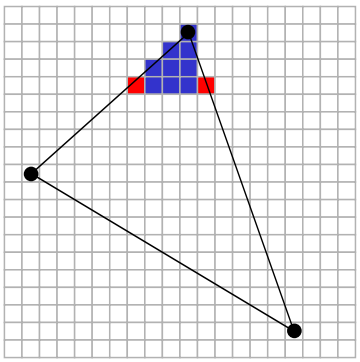

Figure 7.2: This shows a scanline based rasterization. The scanline can be incrementally computed between two neighboring rows.

While this technique was adopted early on, it was identified to show poor scalability to handle scenes with many triangles, since this technique relies on expensive sorting operations and is not friendly for parallelization. Instead, ray tracing and Z-buffer techniques as visible surface determination, i.e., visibility techniques, are prevail techniques in these days (Sec. 10.4).

In the next section, we discuss another rasterization technique combined with the Z-buffer technique.

7.2 *Rasterization with Edge Equations*

In this section, we discuss a rasterization technique for triangles based on edge equations, as shown in the right side of Fig. 7.1. We will see that this approach is simply and friendly for parallelization, to achieve a high performance and thus handle a scene with many triangles.

Let us first compute an edge equation given two vertices, $\dot{v}_0$ and $\dot{v}_1$, of a triangle (Fig. 7.3). Our goal is to construct an edge

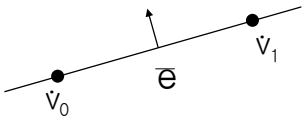

Figure 7.3: An edge representation from two vertices of a triangle.

equation, $\bar{e}$, whose normal vector heads towards the inside of the triangle. Overall, coefficients of the edge equation is given by the cross product between those two vertices:

$$\begin{aligned}\bar{e} &= \dot{v_0} \times \dot{v_1} \\ &= \begin{bmatrix} x_0 & y_0 & 1 \end{bmatrix}^t \times \begin{bmatrix} x_1 & y_1 & 1 \end{bmatrix}^t \\ &= \begin{bmatrix} (y_0 - y_1) & (x_1 - x_0) & (x_0 y_1 - x_1 y_0) \end{bmatrix} \\ &= \begin{bmatrix} A & B & C \end{bmatrix}. \end{aligned} \tag{7.1}$$

It is not intuitive to compute the edge equation in this way. Here is the rationale. Think of a line passing $\dot{v_0}$ in the homogeneous space, i.e, $(x_0 w, y_0 w, w)$ with an arbitrary value w. We also think another line passing $\dot{v_1}$. The edge in the 2D space maps to a plane in the 3D homogeneous space. Since these two lines and the plane passes the origin, $(0, 0, 0)$, of the 3D homogeneous space, the normal of the plane, i.e., the edge equation, is computed by the cross product between $\dot{v_0} - (0, 0, 0)$ and $\dot{v_1} - (0, 0, 0)$.

Once we set the edge equation $\bar{e}$ in this way, points, $\dot{p}$, inside the triangle have $\bar{e}\dot{p} > 0$. We then see that pixels of the triangle reside in the positive half-spaces against three edge equations from the triangle (Fig. 7.1).

While the aforementioned approach is simple enough to identify which pixels are inside a triangle, there are a few special cases requiring certain treatments. They are two cases of sharing edges and vertices.

Sharing an edge. The left image of Fig. 7.4 shows that a shared edge of two triangles passes the center of a pixel. This case arises rarely, but can happen, since there are many pixels, say 1 M pixels when we use a 1 K by 1 K image resolution. When we assign the pixel to both of those two triangles, the pixel color varies depending on an order of rendering those two triangles, which is not a desirable effect. We thus need a tie-breaker assigning only a single triangle to the pixel.

A simple method is to consider the normal of each edge of a triangle and to assign the pixel to either one of them. For example, we can use the following simple tie-breaker:

$$\text{bool} \quad t = \begin{cases} A > 0 & \text{if } A \neq 0, \\ B > 0 & \text{otherwise,} \end{cases}$$

where (A, B) are the normal vector of an edge computed by Eq 7.1. We then assign a triangle to the pixel, when $(\bar{e}(\dot{p}) > 0) \vee (\bar{e}(\dot{p}) = 0 \wedge t)$.

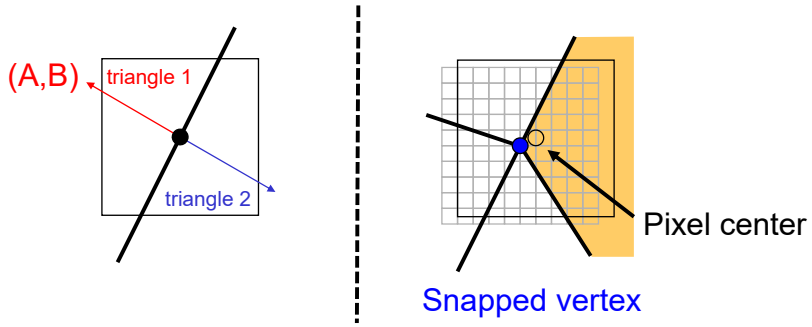

Figure 7.4: This figure shows two cases requiring special treatments for the pixel coverage.

Sharing a vertex. The right image of Fig. 7.4 shows another degenerated case, where a shared vertex of triangles is located at the center of the pixel. For handling this case, one can use a similar tie-breaker that we designed for the shared edge case. Another approach is to snap or quantize vertices of triangles in a way that those snapped or quantized vertices are not aligned with center coordinates of pixels.

7.3 *Interpolation Parameters*

In the last section, we discussed which pixels are covered by a triangle based its edge equations. In this section, we study how to compute colors and other parameters for the pixel, given associated information of the triangle.

Suppose that each vertex has associated information such as color, normal, etc. For the sake of simplicity, we explain various concepts based on the color, especially, red channel information, $r(x, y)$, given a pixel (x, y). Given three red values associated with three vertices of a triangle, we need a way of interpolating these values for a pixel within the triangle. The simplest method is to pick a red value among those three values. While this is simple, it does not produce reasonably high-quality rendering results.

Among many options, we use the linear interpolation from those available values associated with three vertices (Fig. 7.5). The linear red plane is then defined as the following:

$$r(x, y) = A_r x + B_r + C_r, \tag{7.2}$$

where A_r, B_r, C_r are three coefficients of the 2D plane. There are three unknowns and we thus need three equations to compute the plane. Fortunately, these three equations are defined by available

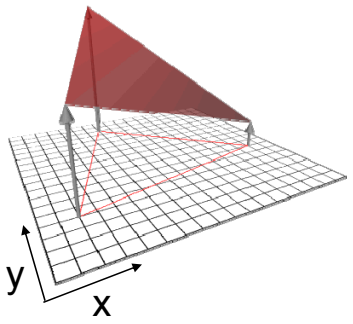

Figure 7.5: This shows the linear interpolation of color values associated with three vertices.

information of three vertices as the following equation:

$$\begin{bmatrix} r_0 & r_1 & r_2 \end{bmatrix} = \begin{bmatrix} A_r & B_r & C_r \end{bmatrix} \begin{bmatrix} x_0 & x_1 & x_2 \\ y_0 & y_1 & y_2 \\ 1 & 1 & 1 \end{bmatrix} \rightarrow$$

$$\begin{bmatrix} A_r & B_r & C_r \end{bmatrix} = \begin{bmatrix} r_0 & r_1 & r_2 \end{bmatrix} \frac{\begin{bmatrix} (y_1 - y_2) & (x_2 - x_1) & (x_1 y_2 - x_2 y_1) \\ (y_2 - y_0) & (x_0 - x_2) & (x_2 y_0 - x_0 y_2) \\ (y_0 - y_1) & (x_1 - x_0) & (x_0 y_1 - x_1 y_0) \end{bmatrix}}{\det \begin{bmatrix} x_0 & x_1 & x_2 \\ y_0 & y_1 & y_2 \\ 1 & 1 & 1 \end{bmatrix}}, \tag{7.3}$$

where r_0, r_1, r_2 are three red values associated to corresponding three vertices.

An interesting fact is that the area of the triangle, $A_{\dot{v}_0\dot{v}_1\dot{v}_2}$, is computed as the following by utilizing the determinant of a matrix:

$$\begin{aligned} A_{\dot{v}_0\dot{v}_1\dot{v}_2} =& \frac{1}{2} \det \begin{bmatrix} x_0 & x_1 & x_2 \\ y_0 & y_1 & y_2 \\ 1 & 1 & 1 \end{bmatrix} && (7.4) \\ =& \frac{1}{2} \left((x_1 y_2 - x_2 y_1) + (x_2 y_0 - x_0 y_2) + (x_0 y_1 - x_1 y_0) \right) \\ =& \frac{1}{2} (C_{20} + C_{12} + C_{01}), && (7.5) \end{aligned}$$

where C_{20}, C_{12}, C_{01} are coefficients of three edge equations, $\bar{e}_{20}, \bar{e}_{12}, \bar{e}_{01}$, respectively; $\bar{e}_{20}$ indicates the edge equation constructed from $\dot{v}_2$ to $\dot{v}_0$.

Note that when the area is zero, the triangle is invisible. Furthermore, when the area is negative, the triangle is back-facing. If the back-face culling is enabled (Ch. 6.3), we cull the triangle for later rasterization. Otherwise, we flip normals of edge equations and perform later rasterization.

Let's consider the interpolation equation (Eq. 7.3). Actually, other components of the top matrix are coefficients of three edge equations! We then have the following interpolation equation:

$$\begin{bmatrix} A_r & B_r & C_r \end{bmatrix} = \frac{1}{2A_{\dot{v}_0\dot{v}_1\dot{v}_2}} \begin{bmatrix} r_0 & r_1 & r_2 \end{bmatrix} \begin{bmatrix} \bar{e}_{20} \\ \bar{e}_{12} \\ \bar{e}_{01} \end{bmatrix}, \tag{7.6}$$

where $\bar{e}_{20}$ represents an 1 by 3 vector containing its three coefficients, A_{20}, B_{20}, C_{20}.

Once we compute coefficients of the red plane (Eq. 7.2), we can compute a color on any pixel within the triangle.

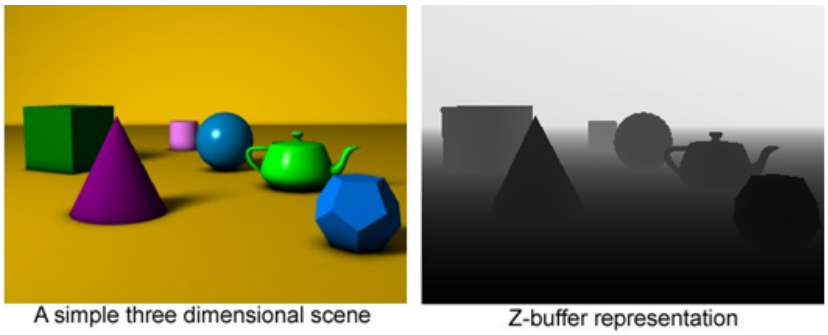

Figure 7.6: The left shows an input scene, while the right image shows its Z-buffer. The white color represents the farthest depth value, 1, while the black one indicates the closest value, 0.

7.4 *Z-Buffering*

The Z-buffer technique is a visibility determination technique, which encodes the depth value of a visible triangle per each pixel. Overall, it is an image-space technique to determine the visible triangle by using a 2D buffer, i.e., depth-buffer.

Z-buffer is one of the most important concepts for rasterization. Simply speaking, we address a complex problem of visibility determination using a 2D map.

Fig. 7.6 visualizes the depth buffer, i.e., Z-buffer, given a scene. The depth buffer simply contains depth values of visible triangles. Note that each vertex of a triangle has its position information (x, y, z). Once we project it to the image space, we also have its depth value in the canonical view volume (Ch. 4.2). The depth value in the canonical view volume spans in the range of $[0, 1]$, where 0 indicates the closest one, while 1 indicates the farthest one.

Given the depth value of each pixel, more correctly, fragment, rasterized from a triangle, we can easily know that whether the fragment has a depth value smaller than the one stored in the depth buffer and thus visible. Once the fragment has a smaller depth value, we update the depth buffer with that depth value at the pixel. We continue this process until we process all the fragments generated from the rasterization process.

As you can see, this Z-buffer is very simple, and thus can be well adopted to a hardware implementation. While there have been many advanced techniques, this Z-buffer technique is the most common technique adopted in rasterization. Nonetheless, recent ray tracing techniques are getting wider attentions thanks to its conceptual simplicity and better functionality supporting realistic rendering effects (Ch. 10).

Processing order. Note that the rasterization method based on the edge equation can be parallelized among different pixels. For example, a rasterization result of a pixel does not depends on anything of another pixel. This opens up various approaches to parallelize the

process for achieving higher performance.

Fig. 7.7 shows two examples of the processing ordering of pixels for rasterizing the triangle. In practice, we identify a bounding box covering the triangle and process the region based on tiles. A tile is a sub-region, say 4 by 4 pixels, of the image space. A GPU core is assigned to process each tile. Different GPU cores process those tiles in a parallel manner, to achieve a high performance. A GPU core assigned to a tile needs to setup three edge equations for a pixel, (x, y), in the tile. For the neighboring pixel, say $(x, y+1)$, we incrementally compute those edge equations, as the following:

$$\begin{aligned} E(x,y) &= Ax + By + C, \\ E(x+1,y) &= A(x+1) + By + c \\ &= E(x,y) + A. \end{aligned} \tag{7.7}$$

So far, we have discussed the rasterization process converting a triangle into a set of fragments. This is one of main concepts of rasterization, setting apart it from ray tracing.

While the rasterization process adopted back-face culling, it can be very slow, especially, when the given scene has so many triangles. There have been many scalable techniques (e.g., mesh simplification) to handle such cases.

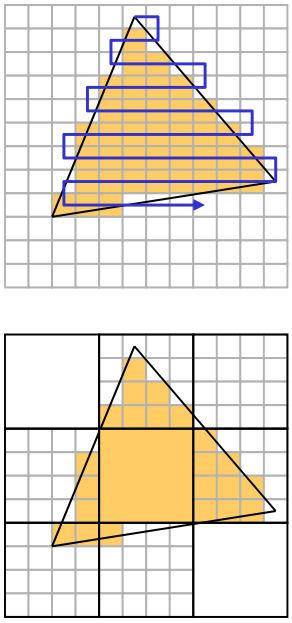

Figure 7.7: Rasterization process can be parallelized, and any ordering of processing pixels or tiles can be possible.