SUNG-EUI YOON, KAIST

RENDERING

FREELY AVAILABLE ON THE INTERNET

First printing, July 2018

6 Clipping and Culling

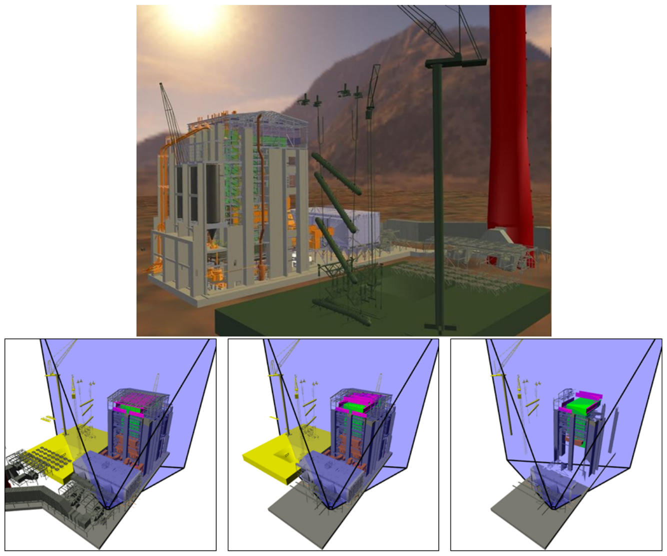

Figure 6.1: The top model shows a coal-fired power plant model consisting of 12 millions of triangles. The model has many pipes within the green, furnace room. It has drastically irregular distributions of triangles across the model ranging from a small bolt to large walls in the furnace. This model is courtesy of an anonymous donor. Bottom images show effects of performing various culling operations. The middle image is the result after performing view-frustum culling to the original power plant model shown in the left. We show these models in a 3rd person view, while the light blue shown in black lines represents the 1st person's view where we perform various culling. The right image shows the result after performing occlusion culling. Since the model has a depth complexity, occlusion culling shows a high culling ratio in this case.

The performance of rasterization linearly increases as we have more triangles. While GPU accelerates the performance of rasterization, it improves only a constant factor, not the time complexity, i.e., growth rate, of the rasterization method. Especially, when we have so many triangles in a scene, it may be prohibitively slow for such scenes. An example includes a power plant scene consisting of 12 millions of triangles (Fig. 6.1).

In this chapter, we discuss two acceleration techniques, clipping and culling, to improve the performance of rasterization. At a high level, their main concepts are:

1. **Culling.** Culling throws away entire primitives (e.g., triangles) and objects that cannot possibly be visible to the user. This is one of important rendering acceleration methods.

2. **Clipping.** Clipping clips off the visible portion of a triangle and throws away the invisible part. This simplifies various parts of the rasterization process.

6.1 *Culling*

Culling conservatively identifies a set of triangles and objects that are invisible to the viewer, and does not pass them to the rendering pipeline. Since the culling process itself can have its own overhead, it is important to design an efficient culling method, while identifying a large portion of invisible triangles among their maximum set.

Fig. 6.1 shows two culling methods, view-frustum culling and occlusion culling, applied to the power plant model. Since this model has a high depth complexity, i.e., many triangles map to a pixel in the screen image, and widely distributed triangles across its scene, such culling methods can be very effective, while they have their own computational overheads. Some of culling methods work as the following:

1. **Back-face culling.** We cannot see triangles heading away from us, unless such triangles are transparent. In opaque models, back-face triangles are blocked by front-face triangles. Back-face culling can be done quite easily and integrated in the rendering pipeline (Sec. 6.5).

2. **View-frustum culling.** The view-frustum (Fig. 6.1) shows an example of the view-frustum and its culling result. Typically, the view-frustum is defined as a canonical view volume within the rendering pipeline and performed by checking whether a triangle or an object is inside the volume or not.

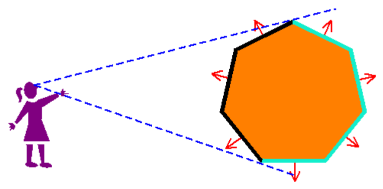
Figure 6.2: Back-face triangles of closed objects are invisible, and back-face culling aims to cull such triangles.

3. **Occlusion culling.** In the case of opaque models, we cannot see triangles located behind the closest triangle to the viewer. As we have more complex models, such models tend to have more numbers of triangles and thus more numbers of triangles map to a single pixel, resulting in a higher depth complexity. In this case, occlusion culling identifies such occluded triangles or objects. Typically, occlusion culling has been more difficult to be adopted, since knowing whether a triangle is occluded or not may require rasterizing the triangle, which we wanted to avoid initially through occlusion culling.

In the next section, we discuss inside/outside tests that are basis for many culling and clipping methods.

6.2 *Inside/Outside Tests*

Many culling and clipping methods check whether a point (or other primitives) is inside or outside against a line in 2D or a plane in 3D. We thus start with a definition of a line for the sake of simplicity; the discussion with the line naturally extends to 3D or other dimensions.

Among many alternatives on definitions on lines, we use the following implicit line representation:

$$\begin{aligned}(n_x, n_y) \cdot (x, y) - d = 0 &\rightarrow \\ n_x x + n_y y - d = 0 &\rightarrow \\ \begin{bmatrix} n_x & n_y & -d \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = 0 &\rightarrow \\ \bar{l}\dot{p} = 0, & \qquad (6.1)\end{aligned}$$

where $(n_x, n_y) \equiv \vec{n}$ is a unit normal vector of the line equation and $\dot{p}$ is a point in the homogeneous coordinate. We use $\bar{l}$ to denote coefficients of the line.

Given the line equation, we also define the positive half space, $\dot{p}^+$, where $\bar{l}(\dot{p}^+) \equiv \bar{l}\dot{p}^+ > 0$; we also define the negative half space in a similar way. We use the following lemma for explaining culling techniques.

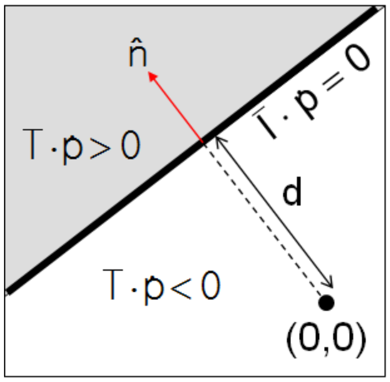

Figure 6.3: Notations of the implicit line equation.

Lemma 6.2.1. *When the normal of the line equation, Eq. 6.1, is a unit normal vector, d gives the L2 distance from the origin of the coordinate system to the line.*

Proof. Let us define (x, y) to be the point in the line realizing the minimum $L2$ distance from the origin to the line, and we then have the following equation:

$$\begin{aligned}(n_x, n_y) &= s(x, y), \\ n_x^2 + n_y^2 &= 1, \\ s^2(x^2 + y^2) &= 1. \qquad (6.2)\end{aligned}$$

where s is a non-zero constant. Since the point (x, y) is in the line, we have the following equation:

$$\begin{aligned}n_x x + n_y y &= d, \\ d &= \frac{1}{s}\left(n_x^2 + n_y^2\right) = \frac{1}{s}, \\ &= \sqrt{x^2 + y^2} \because \text{Eq. } 6.2. \qquad (6.3)\end{aligned}$$

$\square$

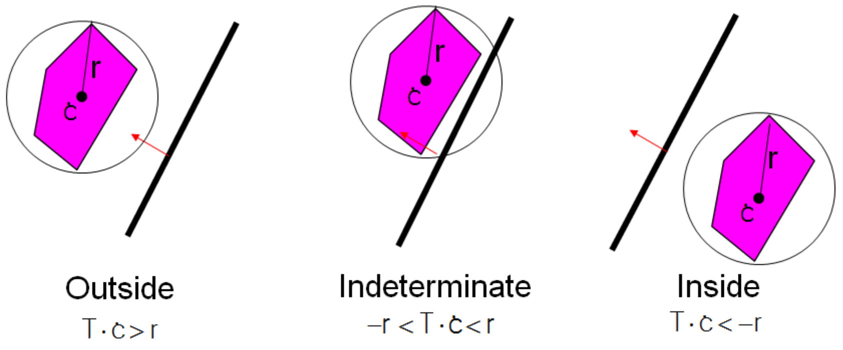

Figure 6.4: This shows three different cases of culling the polygon given its enclosing spherical bounding volume.

In a similar way of proving the lemma, we can see that given a point (x, y) that is or is not on a line whose normal is (n_x, n_y), $n_x x + n_y y$ gives a distance from the point to the line. We also utilize this property for designing culling techniques.

6.3 *View-Frustum and Back-Face Culling*

Let us discuss a simple culling scenario against a line before moving to view-frustum and back-face culling. Suppose that we have a polygon, and we can cull it when the polygon is located totally outside a culling line, as shown in Fig. 6.4. Since it takes a high culling overhead against each vertex of the polygon with many vertices against the line, we use a bounding volume that tightly encloses the polygon.

There are many different types of bounding volumes (BVs) including spheres, boxes, oriented boxes, etc. Commonly, spheres and axis-aligned bounding boxes (AABBs) are frequently chosen bounding volumes, since they are easy to compute with a low computational overhead and a reasonably high culling ratio. Detailed discussions are available in the chapter of bounding volumes and bounding volume hierarchy for ray tracing (Sec. 10.3). In this section, we simply use the sphere for the sake of clear explanation.

Suppose that we use a sphere enclosing the polygon. As a simple culling method in this case, we use its center, c, and radius, r, irrespective of how many vertices the polygon has. Specifically, we test the center against a culling line, $l(\dot{p})$, by plugging its center position to its implicit line equation. There are three different cases (Fig. 6.4), depending on the value of $l(c)$. Since we assume to cull the polygon when it is located outside the line, we focus on this case only in this chapter.

The value of $l(c)$ indicates the $L2$ distance from the line to the center c. When $l(c) > r$ indicates that the sphere is conservatively

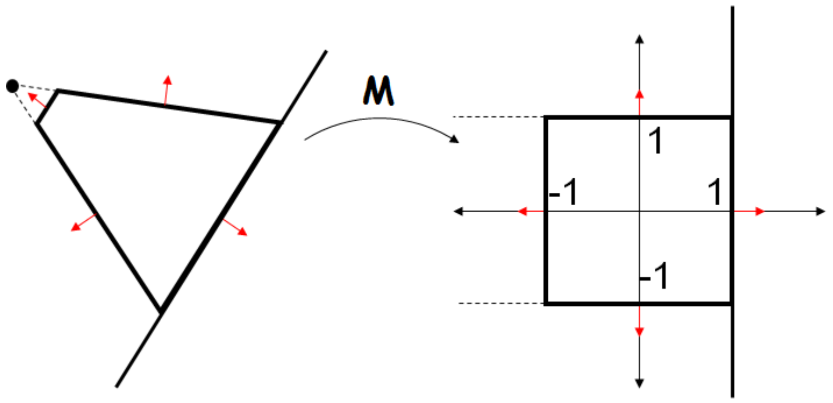

Figure 6.5: The left image shows a view-frustum in 2D, while the right image shows its canonical view volumes. These lines in 2D and planes in 3D of the canonical view are compactly represented and thus can result in fast runtime performance.

outside the line, we can cull it from the further rendering process. With the provided information, it is unclear whether this simple culling operations results in a higher rendering performance than naively rendering all those objects. Nonetheless, we have discussed a basic concept of culling against a line. The performance of this basic approach can be significantly improved by using hierarchically computed bounding volumes, known as bounding volume hierarchy (Sec. 10.3).

Let's see how we perform the view-frustum culling. In rasterization, we assume that we see objects only located within the view-frustum, while this is not the case in reality [1]. Based on this assumption, we can safely cull triangles located outside the view-frustum.

[1] We can see other objects that are reflected by objects (e.g., mirror) located within the view-frustum.

The view-frustum is defined as the left image of Fig. 6.5. We can define such planes with the implicit plane equations, but the view-frustum defined by the given camera setting is transformed to the canonical view volume, which are defined as $x = \pm 1, y = \pm 1, z = \pm 1$, as mentioned in Sec 4.2. The right image of Fig. 6.5 shows the canonical view-volume in 2D.

When a triangle is located outside either one of these six planes, we cull the triangle. This operation applies to each triangle, and is adopted in the rendering pipeline. For large-scale scenes where the view-frustum contains only a portion of them, we can apply the culling method in a hierarchical manner by using a hierarchical acceleration data structure such as bounding volume hierarchy. This approach is more involved and thus a rendering engine supports it.

Back-face culling can be done in a different way. In this section, we discuss a method utilizing the inside/outside tests. One can observe that we cannot see a triangle, when it faces backward (Fig. 6.2). More specifically, suppose that we compute a plane passing the triangle. Then, the triangle is classified as the back-face, when the eye is located in the negative half side of the plane.

To compute such a plane, we need a normal, the orthogonal vector heading outward to the plane. Given a vertex ordering from v_0, v_1, v_2 in the counter-clock wise, the normal of the triangle, $\vec{n}$, and the distance, d, of the plane is computed as the following:

$$\begin{aligned} \vec{n} &= (v_1 - v_0) \times (v_2 - v_0), \\ d &= \vec{n} \cdot v_0, \end{aligned} \tag{6.4}$$

where the dot product computes the projected distance of the vertex v_0 to the normal direction.

Later, in Ch. 7.3, we discuss a faster back-face culling method, which is more appropriate to be adopted in the rendering pipeline.

Back-face culling in OpenGL. To cull back facing triangles in OpenGL, we use $glCullFace(\cdot)$ after enabling the feature (e.g., GL_CULL_FACE). OpenGL identifies back-face or front-face based on its normal computed from its vertex ordering (Ch. 7.3). OpenGL also provides a way of defining back-face and front-face based on a winding order of vertices between clockwise or counter-clockwise. The counter-clockwise ordering indicates that when we wrap those vertices starting from v_0, passing v_1 to v_2 with the hand, the thumb direction is the front-face. By culling away such back facing triangles, we can avoid to generate fragments from those triangles, resulting in a higher performance.

6.4 *Clipping*

In this section, we discuss clipping that identifies only a visible portion of a primitive, i.e., triangle, and pass it to the following stage (e.g., rasterization stage) in the rendering pipeline.

Let's first discuss a simple case, clipping a line segment consisting of two points, $\dot{p}_0, \dot{p}_1$, against another line, whose coefficient is represented by $\vec{l}$. Our goal here is to identify the clipping point, $\dot{p}$, that intersects with another line $\vec{l}$. To compute the point, we present $\dot{p}$ with a line parameter, t, as the following:

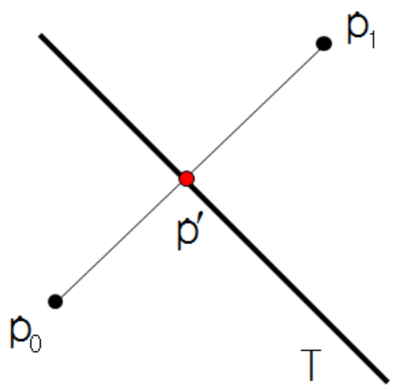

Figure 6.6: A configuration of culling an edge against a line.

$$\dot{p} = \dot{p}_0 + t(\dot{p}_1 - \dot{p}_0). \tag{6.5}$$

The point should be in another line and thus $\bar{l}\dot{p} = 0$. We then have the following equation:

$$\begin{aligned} &\bar{l} \cdot (\dot{p}_0 + t(\dot{p}_1 - \dot{p}_0)), \\ &t = \frac{-(\bar{l} \cdot \dot{p}_0)}{\bar{l} \cdot (\dot{p}_1 - \dot{p}_0)}. \end{aligned} \tag{6.6}$$

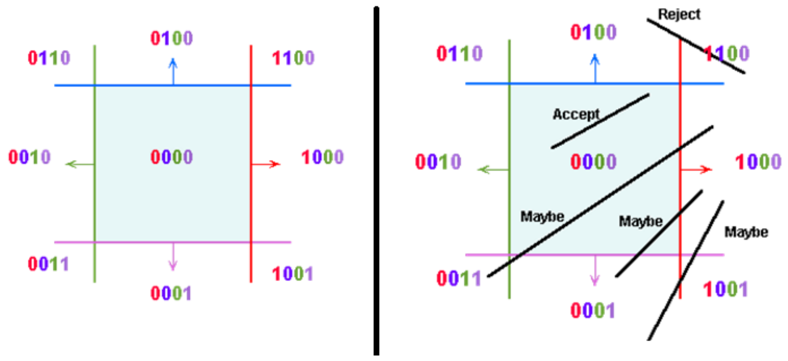

Figure 6.8: The left shows outcodes for each region defined by four lines of the view region. The right shows results of culling edges based on the Cohen-Sutherland method.

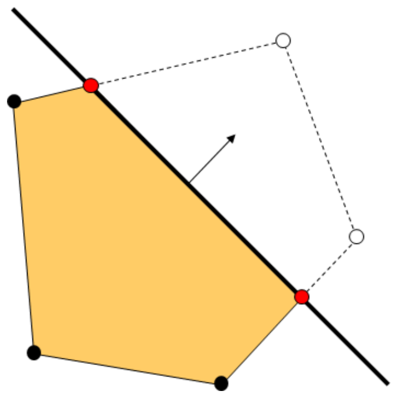

Figure 6.7: The Sutherland-Hodgman method computes a clipped polygon against a line.

Each vertex is also associated with other attributes like colors and texture coordinates. We can also compute those attributes for the clipping point based on the same interpolation method.

Based on this simple line-by-line clipping method, we explain a clipping method, Sutherland-Hodgman algorithm for a polygon including a triangle against a line (e.g., a line of the viewport rectangle) of a convex viewport.

In this method, we traverse each edge of the polygon and check whether the edge is totally inside against the line or not. When it is totally inside or outside, we keep it or throw away it, respectively. Otherwise, we compute two clipping points as shown in Fig. 6.7 and connect them with a new edge. We also apply this process repeatedly against each line of the viewport region.

6.4.1 *Cohen-Sutherland Clipping Method*

The Cohen-Sutherland method is used to quickly check whether an edge is totally inside or outside given the view region, by using the concept of outcodes. An outcode is assigned to each vertex of primitives, whose each bit encodes whether the vertex is inside or outside against its corresponding line (Fig. 6.8). For example, the first bit in the figure corresponds inside (1) or outside (0) regions against the red line.

When we consider two binary codes, c_1 and c_2, assigned to two vertices of an edge, we have the following conditions and actions:

- If $(c_1 \vee c_2) = 0$, the edge is inside.
- If $(c_1 \wedge c_2) \neq 0$, the edge is totally outside.
- If $(c_1 \wedge c_2) = 0$, the edge potentially crosses the clip region at planes indicated by true bits in $(c_1 \oplus c_2)$. Nonetheless, this could be false positive, meaning that they are identified to be potentially crossing the clip region, but are not actually.

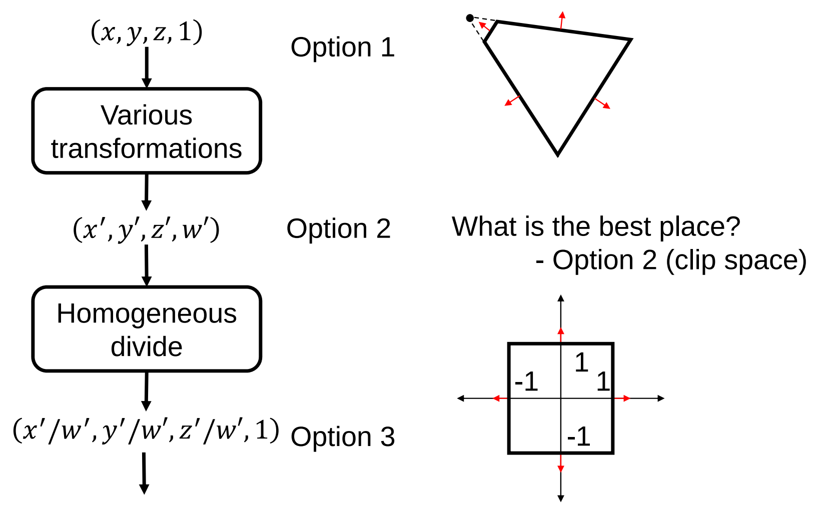

Figure 6.9: This shows different stages of the rendering pipeline with vertex coordinates and view-frustum in each space.

This also applies to a triangle case by utilizing three outcodes computed from three vertices of the triangle.

6.5 *Clipping in the Pipeline*

We discussed how to clip an edge against a plane of the view-frustum before. We would like to now discuss in which stage of the rendering pipeline we perform the clipping operation.

Fig. 6.9 shows how vertex coordinates change as we perform different steps in the rendering pipeline. Overall, there are three different places where we can perform the clipping operation. The first option is the world space where the view-frustum is defined. The second and third options are before and after performing the homogeneous divide.

Each option has its pros and cons. The most intuitive option would be the first one. Also, the third option seems to be good, since the plane equations of the view-frustum in that space are canonical like $x = 1$, and the clipping operation can be done quite quickly. Nonetheless, if we do not clip an edge that spans outside the view-frustum before this stage, the edge flips around due to the projection carried by the homogeneous divide, and generates an unexpected behavior. As a result, the third option is not possible.

Interestingly, the second option has been identified empirically to show the best place to perform the operation, since it does not have the problem of the option three and their plane equations are also defined quite easily. The space of the option two is known as the clip space. Let us discuss how the view-frustum is defined in this clip space. Specifically, $x'/w' = 1$ in the third space corresponds to $x'/w'w' = w' \rightarrow x' = w'$ in the clip space, which does not depend on the camera setting, and thus can be done efficiently.

[2] As you may realize through this discussion, the rendering pipeline has been heavily tested and optimized to deliver the highest rendering performance. Nonetheless, these choices can change depending on different workloads (e.g., some games use geometry or texture heavily) and hardware performance (e.g., faster memory read or computation).

[2] Structures of the rendering pipeline are not fixed and can be changed for better performance and usability.

6.6 Common Questions

Even though some objects are outside the view frustum, they can be seen though transparent objects or reflected from mirrors. Exactly. The rasterization algorithm is a drastically simplified rendering algorithm over the real interactions between lights and materials. The direct illumination, seen thought primary rays, are well captured by rasterization, while other indirect illuminations are not captured well in the rasterization. To address this problem, many techniques have been proposed in the field of rasterization. However, the most natural way of handling them is to use ray tracing based rendering algorithms.