A fundamental operation that is used extensively in computer graphics and visualization is the process of scan conversion or rasterization. Given a polygon in image space, this process determines the pixels that intersect the polygon. This process is utilized in visible-surface algorithms, incremental-shading techniques, polygon-fill algorithms, ray-tracing-acceleration algorithms, and a number of other tasks that are critical to the understanding of the computer graphics field.

**Scan Conversion of Trapezoids in Device Space**

Scan Conversion is the process of finding the screen pixels that intersect a polygon. To do this, we find it convenient to move to a copy of image space that is scaled to closely correspond to the pixels in our display window. This space, commonly called device space is parameterized so that the lower-left-hand corner is at $(0, 0)$, and so that the pixels can all be indexed by integers.
To scan convert a polygon in this space, we will split the polygon into a set of trapezoids as is shown below, and then scan convert each trapezoid. Each trapezoid will be of a special form where the top and bottom edges of the trapezoid are parallel to the scanlines (i.e., of a constant $y$ value). We will also consider “degenerate” trapezoids – triangles – which have either the top or bottom edge of zero length. In the following illustration, the polygon is split into five trapezoids. The top and bottom trapezoids are actually triangles.

The union of all pixels that intersect the set of trapezoids will be the set of pixels that intersect the
The idea here is easy. We will establish an “edge tracker” which follows the endpoints of the lines formed by intersecting each scanline with the trapezoid.
This edge tracker can be easily defined as a simple data structure which is initialized for the scanline at the top of each trapezoid and updated for each subsequent scanline.

### Initializing an Edge Tracker

Suppose we are given an edge defined by two points $P_1 = (x_1, y_1, z_1)$ and $P_2 = (x_2, y_2, z_2)$, defined in device space. If we define $\Delta x = x_2 - x_1$, $\Delta y = y_2 - y_1$ and $\Delta z = z_2 - z_1$, then the following illustration shows how to calculate the initial point (only the $x$ and $y$ values are shown for simplicity).

We see that the initial point is

$$(x_2 + \delta x_d, \lfloor y_2 \rfloor, z_2 + \delta z_d)$$
To calculate \( x_d \), we notice that there are two similar triangles in the illustration, and therefore we can write

\[
\frac{x_d}{y_2 - \lfloor y_2 \rfloor} = \frac{x_1 - x_2}{y_2 - y_1} = -\frac{dx}{dy}
\]

or

\[
x_d = -(y_2 - \lfloor y_2 \rfloor) \frac{\Delta x}{\Delta y}
\]

Similarly, we could calculate

\[
z_d = -(y_2 - \lfloor y_2 \rfloor) \frac{\Delta z}{\Delta y}
\]

**Updating an Edge Tracker for Subsequent Scanlines**

Suppose we are given an edge defined by two points \( P_1 = (x_1, y_1, z_1) \) and \( P_2 = (x_2, y_2, z_2) \), defined in device space. If we define \( \Delta x = x_2 - x_1 \), \( \Delta y = y_2 - y_1 \) and \( \Delta z = z_2 - z_1 \), then the following illustration shows how to update the edge tracker (only the \( x \) and \( y \) values are shown for simplicity).
Here we have again similar triangles, and it can be seen that to move the tracker from one scanline to another, we only need update the endpoint

\[(x, y, z)\]

to

\[(x - x_{inc}, y - 1, z - z_{inc})\]

and this same calculation works for each endpoint.

We can calculate \(x_{inc}\) directly from the similar triangles:

\[x_{inc} = \frac{\Delta x}{\Delta y}\]

and similarly

\[z_{inc} = \frac{\Delta z}{\Delta y}\]

Thus, updating the endpoint tracker is very easy, all we do is add simple increments onto the \(x\), \(y\) and \(z\) coordinates, and we move to the endpoint on the next line.

---

**The Endpoint Data Structure**

The easiest way to implement an trapezoid edge tracker is to create a simple data structure that holds both the \((x, y, z)\) information for the endpoints and the increments that are necessary to move the endpoint from scanline to scanline. The node typically contains (at least) the following information.

<table>
<thead>
<tr>
<th>Endpoint Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x)</td>
</tr>
<tr>
<td>(y)</td>
</tr>
<tr>
<td>(z)</td>
</tr>
<tr>
<td>(x_{inc})</td>
</tr>
<tr>
<td>(z_{inc})</td>
</tr>
<tr>
<td>(y_{min})</td>
</tr>
</tbody>
</table>
where \((x, y, z)\) is the endpoint, \(x_{inc}\) is the increment added to the \(x\) coordinate to move from scanline to scanline, \(z_{inc}\) is the increment added to the \(z\) coordinate to move from scanline to scanline, and \(y_{min}\) is the lower bound on the trapezoid that enables us to tell our algorithm when to stop tracking.

---

**Identifying the Intersecting Pixels on Each Scanline**

To find the pixels that intersect a trapezoid, we create two edge trackers and iteratively work down (scanline by scanline) through the trapezoid. For each scanline, this enables us to determine the endpoints of a line that forms the intersection of the scanline and the polygon.

To find the pixels that intersect the trapezoid on a scanline, we only need find the lower-left-hand corner of each pixel. If we have calculated endpoints \((x_{left}, y, z_{left})\) and \((x_{right}, y, z_{right})\) on the scanline with integer value \(y\), then the pixels intersecting the trapezoid are indexed by

\[
\begin{align*}
&\left(\left\lfloor x_{left} \right\rfloor, y \right) \\
&\left(\left\lfloor x_{left} \right\rfloor + 1, y \right) \\
&\vdots \\
&\left(\left\lceil x_{right} \right\rceil - 1, y \right) \\
&\left(\left\lfloor x_{right} \right\rfloor, y \right)
\end{align*}
\]

---

**Incrementing the Z Value for Consecutive Pixels**

Establishing an increment for the \(z\) value as we move from scanline to scanline is quite straightforward. The same is true for as we move from pixel to pixel – however the two increments are different. Both
increments can be calculated directly from the normal vector to the trapezoid (trapezoids are assumed to be planar, as are polygons).

To see this, let \( \mathbf{n} \) be the normal vector to the trapezoid that is defined in device space. If we have two points \( Q_1 \) and \( Q_2 \) in the plane of the trapezoid, then we know that

\[
\mathbf{n} \cdot (Q_2 - Q_1) = 0
\]

since \( Q_2 - Q_1 \) is a vector in the plane. Now, suppose we let

\[
Q_1 = (x, y, z)
\]

and

\[
Q_2 = (x + 1, y, z + z_{\text{hinc}})
\]

which represent the coordinates of two consecutive pixels in device space and \( z_{\text{hinc}} \) is unknown. If we let \( \mathbf{n} = (x_n, y_n, z_n) \), we can calculate the dot product as

\[
0 = (x_n, y_n, z_n) \cdot ((x + 1, y, z + z_{\text{hinc}}) - (x, y, z))
\]

\[
= (x_n, y_n, z_n) \cdot < 1, 0, z_{\text{hinc}} >
\]

\[
= x_n + z_n z_{\text{hinc}}
\]

which can be solved to give

\[
z_{\text{hinc}} = -\frac{x_n}{z_n}
\]

the horizontal increment for \( z \) from pixel to pixel. **We note that this is not the same as the vertical increment for \( z \) in general.**

---

**Establishing a Depth Value at Each Pixel**

Since a pixel is identified by its device-space coordinate at the lower-left-hand corner of the pixel, it is useful in our rendering algorithms to be able to assign a \( z \) value for a trapezoid at this point. We have already shown how to calculate the \( z \) values at the endpoints, we only need modify this value so that it is appropriate for the coordinate of the lower-left-hand corner of the pixel. So given an endpoint \( (x, y, z) \), consider the figure below.
From the figure, we can see that we should subtract from $z$, a portion of the horizontal $z$ increment that corresponds to the distance of $x$ from the left side of the pixel. That is,

$$z' = z - (x - \lfloor x \rfloor)z_{hinc}$$

with this $z$ value, and the horizontal $z$ increment $z_{hinc}$ we can specify a depth value for the trapezoid for each pixel along the scanline.

Using the above calculations, we can use the endpoint data structure to set up edge-tracking mechanisms, and can both enumerate the pixels within a trapezoid and assign a $z$ value to each pixel.

**Summary**

Scan Conversion is a procedure that is used repeatedly in computer graphics algorithms. It is a simple procedure, designed around an edge-tracking paradigm, which can be implemented by adding simply-calculated increments onto base values. One of the primary uses of the algorithm is to establish depth ($z$) values at each
pixel for polygons in the scene, this enables us to retain only the visible polygons in our final renderings. However, we can also utilize quantities relating to color, texture, and parameterization as information in our endpoint nodes and increment them as well. These quantities are included in the algorithm in a manner similar to the \( z \) coordinate above.