May 17, 2017

# Path rendering by counting pixel coverage

Brian Salomon

Christopher Dalton

Allan MacKinnon

# Path rendering by counting pixel coverage

ABSTRACT

A frequent task in computer graphics is to render a closed path, e.g., a polygon or other shape.  Such shapes are found in typography, vector graphics, design applications, etc. Current path-rendering techniques have certain drawbacks, e.g., paths cannot scale too far during animation, control points within the path must remain static, etc. The ability to render paths efficiently and with fewer constraints allows interfaces and applications with richer and more dynamic content. This disclosure describes techniques for efficient path rendering using a GPU. In particular, it introduces the concept of fractional coverage counting, which ameliorates aliasing at the edges of shapes. These techniques can reduce or eliminate reliance on hardware multisampling to achieve anti-aliasing, and open up the possibility of sophisticated graphics rendering on mobile devices or other platforms with resource constraints.

KEYWORDS

- Computer graphics

- Path rendering

- Triangle fanning

- Anti-aliasing

BACKGROUND

In various computer applications, e.g., user interfaces, animations, typeface design, computer-aided design, etc., a frequent task is to render (fill) pixels within a closed shape such as a polygon or closed Bézier curve with a certain color, depth, texture or gradient. A popular

technique for filling polygons is the method of fanning triangles, which determines the set of pixels within a polygon ("path containment test").

In this method, the polygon is overlaid with triangles that have one vertex at a certain vertex of the polygon. Triangles fan out from that vertex. The polygon is thereby divided into regions that are covered with zero, one, or more triangles. Depending on the orientation (clockwise or anticlockwise) of the triangles that overlay a given region, the fill-status of the region is determined.
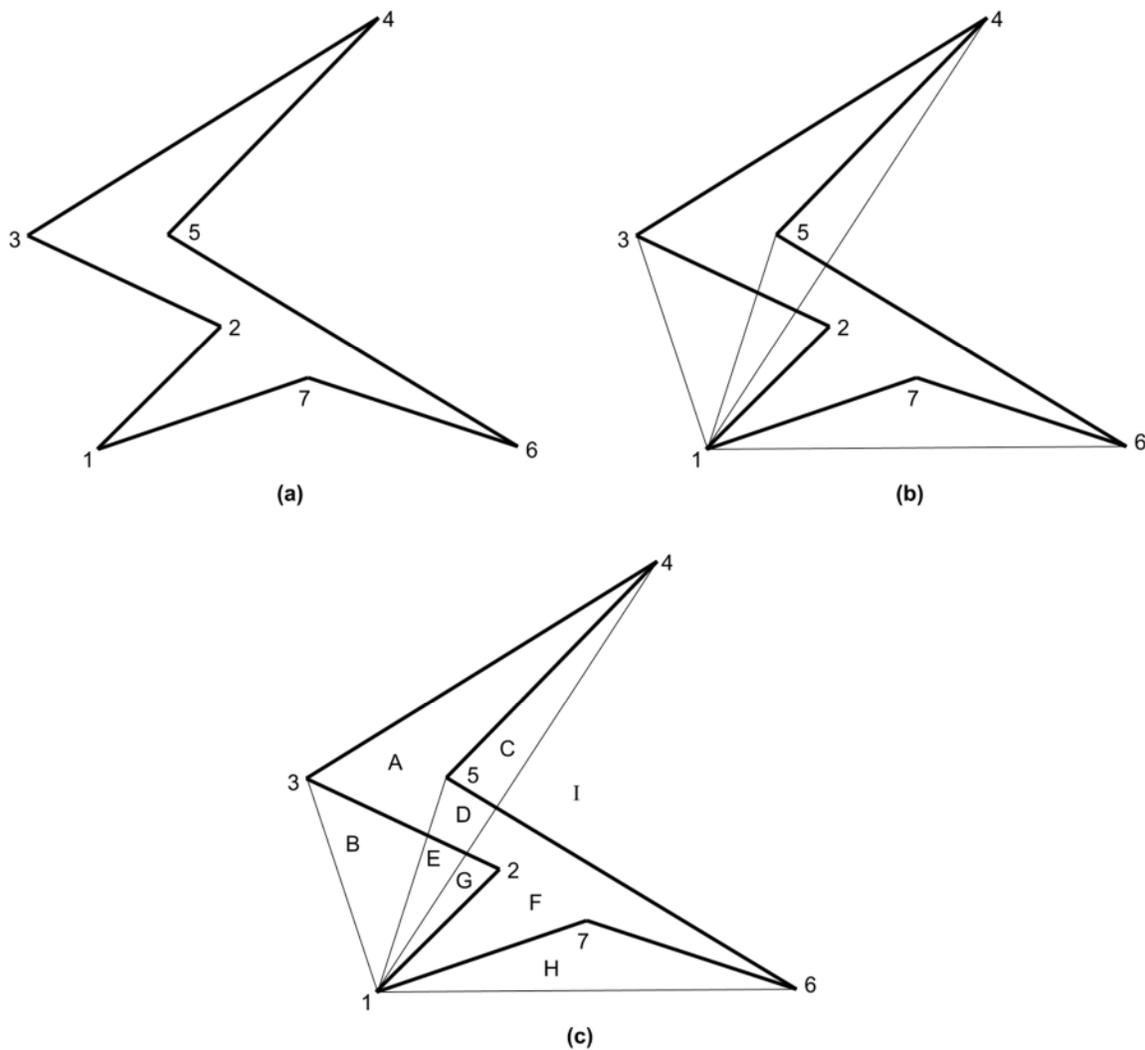


**Fig. 1: Fanning triangles to fill a polygon**

Fig. 1 shows an example illustrating filling a polygon with the method of fanning triangles. The interior of a polygon with the vertices 1, 2, 3, 4, 5, 6, and 7 (polygon 1234567) as shown in Fig. 1(a) is to be filled. Using the method of fanning triangles, a number of triangles that fan out of vertex 1, e.g., 123, 134, 145, 156, and 167 are drawn as shown in Fig. 1(b). Each triangle has an orientation that is decided by the ordering of its vertices. For example, the ordering 1-2-3 of the vertices of triangle 123 indicates anti-clockwise orientation. As a further example, the ordering 1-3-4 of the vertices of triangle 134 indicates clockwise orientation. The triangles 123, 134, 145, 156, and 167 divide space into disjoint, non-overlapping regions A, B, C, D, E, F, G and I, as shown in Fig. 1(c). The region "I" indicates space outside of the original polygon or any triangle.

Each pixel has attributes e.g., color, depth etc., that are typically stored in buffers associated with that pixel. One such buffer ("stencil buffer") is used to store a winding number for the pixel. The winding number is the number of triangles that overlay the pixel, with triangles with clockwise orientation counted as positive and triangles with anti-clockwise orientation counted as negative. Thus, a pixel in region A, which is overlaid by just one triangle (the clockwise-oriented triangle 134) has a winding number of +1. A pixel in region B has a winding number that is the sum of +1 (triangle 134) and −1 (triangle 123), which amounts to 0. Computation of the winding numbers of other regions are in Fig. 1(c) is shown below:

C: +1(triangle 134) −1(triangle 145) = 0

D: +1(triangle 134) −1(triangle 145) +1(triangle 156) = 1

E: −1(triangle 123) +1(triangle 134) −1(triangle 145) +1(triangle 156) = 0

F: +1(triangle 156) = 1

G: −1(triangle 123) +1(triangle 156) = 0

H: +1(triangle 156) −1(triangle 167) = 0

I:  0 (no triangle)

Regions with non-zero winding numbers are filled in, while regions with zero winding numbers are not. Thus, in the example illustrated in Fig. 1(c), regions A, D, and F are filled in, and together, these regions cover the interior of the polygon. The method of fanning triangles is extendable to include curves, e.g., by constructing a fragment shader that discards fragments outside the curve before updating the stencil buffer.

The method of fanning triangles does not always handle edges well. In particular, since each pixel can only either be "inside" or "outside", edges may suffer from "aliasing." Aliasing is an artifact by which smooth curves or lines become pixelated (jagged) due to insufficient resolution. An example of an aliased figure is shown in Fig. 2(a) - e.g., the edges of the triangle in Fig. 2(a) are pixelated. Aliasing is ameliorated as shown in Fig. 2(b), e.g., by smoothing the edges.
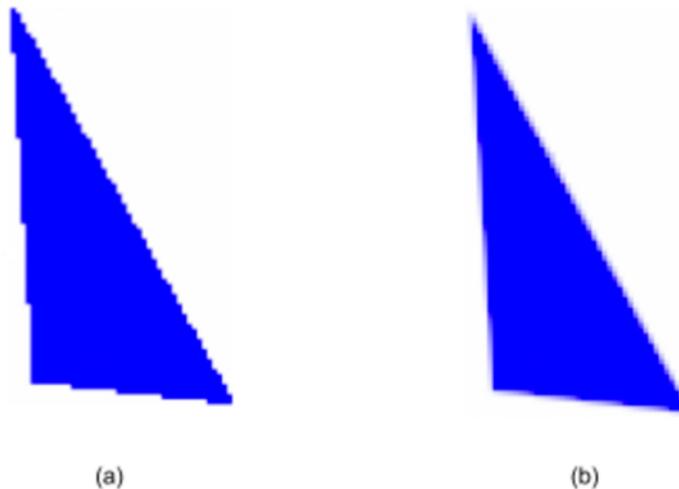
(a)

(b)

**Fig. 2: (a) Aliased image (b) Image after application of anti-aliasing filter**

To date, the only known method for anti-aliasing triangle-fan path rendering has been through hardware multisampling ("multisampling anti-aliasing," or MSAA). In this approach, each pixel computes multiple winding counts at subpixel locations, and its total "coverage" is proportional to the number of samples that are found inside the shape. Hardware multisampling is typically a compute-intensive operation. Indeed, multisampling has been described as a sledgehammer that performs on power-hungry graphics processing units (GPUs) but is prohibitively expensive on certain devices, e.g., mobile devices. Multisampling requires multisampled render buffers that consume significant memory and are expensive to resolve into usable form. Enabling multisampling to anti-alias path rendering can slow down the entire render pipeline.

Other methods of path rendering, such as tessellating and distance field methods do well within certain constraints. However, these methods rely on cached information that is expensive to compute. These approaches also fail with dynamic content, including simple scaling.

DESCRIPTION

This disclosure describes techniques that efficiently use a GPU to efficiently render paths, e.g., closed, oriented contours comprising piecewise Bezier segments. Anti-aliasing is efficiently handled by allowing the number of oriented triangles that overlay a pixel, known as "coverage count" to be fractional.
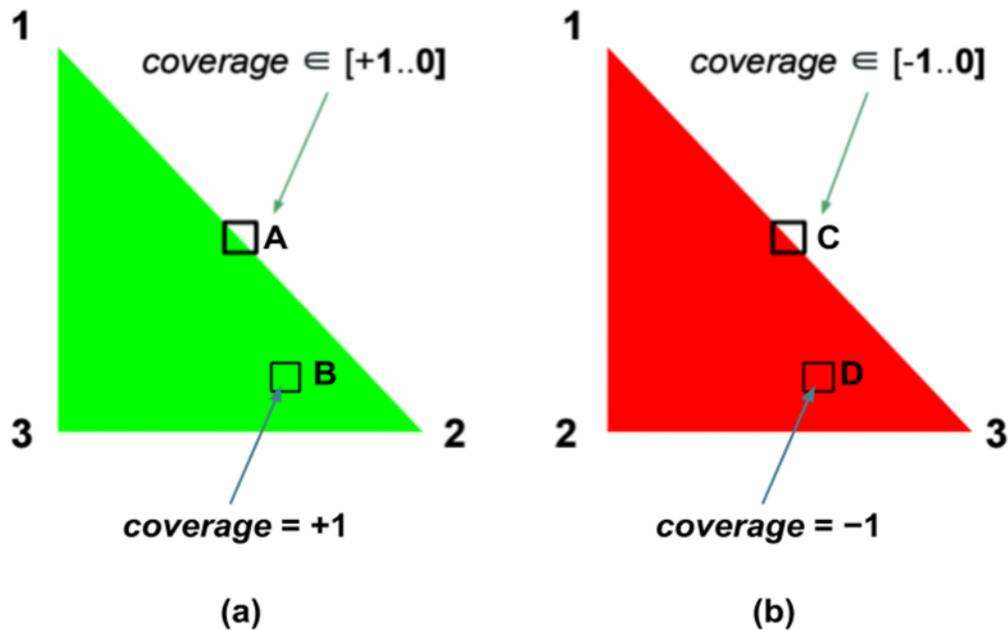
**Fig. 3: Fractional coverage**

Fig. 3 illustrates the concept of fractional coverage, per techniques of this disclosure. Fig. 3(a) illustrates a positively oriented (green-colored) triangle 123. Pixel B which is located fully in the interior of the green triangle emits a coverage count of +1. Pixel A which is located on the edge of the green triangle emits a fractional positive coverage, e.g., equal to the fraction of the pixel that lies within the green triangle.

Fig. 3(b) illustrates a negatively oriented (red-colored) triangle 123. A pixel D which is located in the interior of the triangle emits a coverage −1. A pixel C which is located on the edge of the triangle emits a fractional negative coverage, e.g., equal to −1 multiplied by the fraction of the pixel C that lies within the red triangle. For example, fractional coverage may be stored in a single-channel 16-bit floating point buffer.
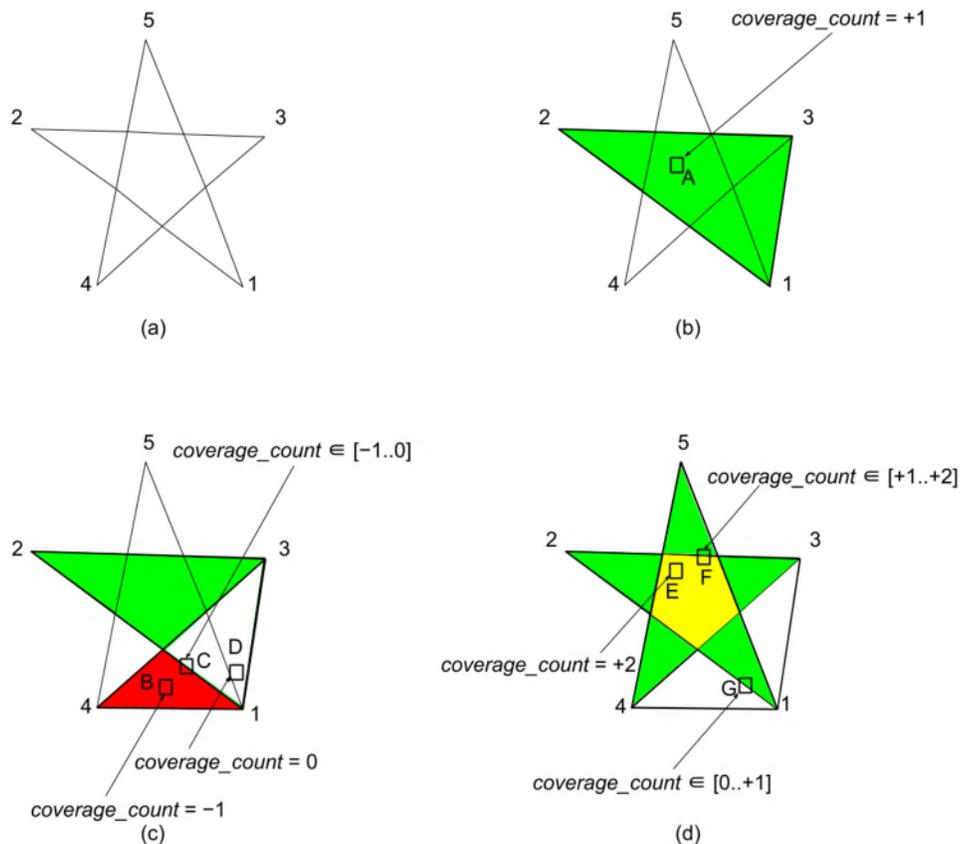
Example 1: *Rendering a star-shaped polygon*



**Fig. 4: Rendering a star-shaped polygon**

Fig. 4 illustrates the rendering of a star-shaped polygon, using the coverage counting techniques disclosed herein. The task is to fill a polygon 12345 of Fig 4(a). Triangles are drawn with one vertex centered at vertex 1 of the polygon. A first triangle is triangle 123 of Fig. 4(b), a clockwise-oriented triangle. The region interior to triangle 123, for example pixel A, has a coverage count of +1, corresponding to the positive orientation of triangle 123.

The next triangle is drawn, e.g. triangle 134 of Fig. 4(c), an anti-clockwise oriented triangle that emits a coverage equal to −1. The cumulative effect of the green triangle 123 and the red triangle 134 at different pixels is as follows. At pixel B, fully and solely in the interior of triangle 134, the coverage count is −1; at pixel D, where triangles 123 and 134 fully overlap

and hence fully cancel each other out, the coverage count is 0; at pixel C, where green and red triangles both assert varying degrees of influence, the coverage is a fraction between −1 and 0.

The next triangle drawn is triangle 145 of Fig. 4(d), a clockwise triangle whose pixels emit a coverage equal to +1. The cumulative effect of all three triangles at different pixels is as follows. At pixel E, where all three triangles assert equal influence, two positively (+1) and one negatively (−1), the coverage count equals +1+1−1=2. At pixel F, which is a transition region between regions of coverage +1 and +2, the coverage count is a fraction between +1 and +2. At pixel G, a transition region between regions with coverage 0 and +1, the coverage count is a fraction between 0 and +1. It is thus seen that the technique of coverage counting, as disclosed herein, ascribes high values of coverage to the interior of a polygon while ascribing fractional values to edges. In this manner, it achieves the objective of filling the polygon while simultaneously reducing or eliminating aliasing at the edges.

The coverage count of a pixel is mapped to its transparency (intensity) using a transform function, e.g., the map shown in Fig. 5 (known as "even/odd fill rule") or Fig. 6 ("winding fill rule").
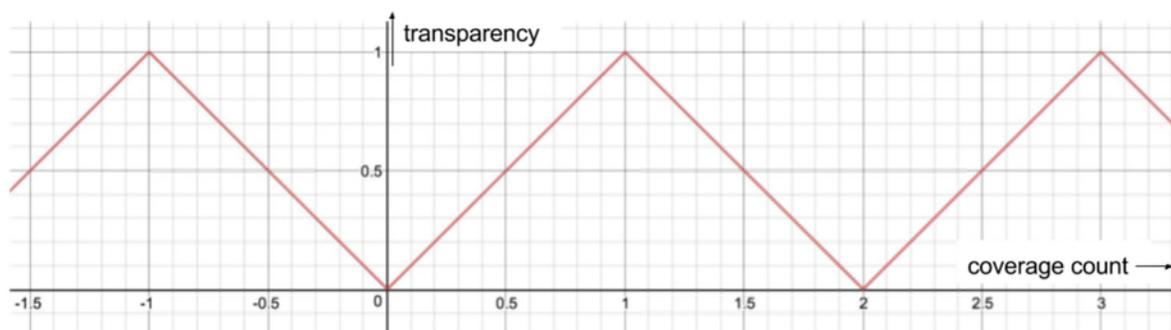


**Fig. 5: Map between coverage count and transparency of a pixel (even/odd fill rule)**

The "even/odd fill rule" of Fig. 5 is expressed mathematically as

```
t=mod(abs(coverage_count),2.0)

alpha =1.0 - abs(t-1.0),
```

where t is an intermediate variable and alpha represents transparency. An alternate map

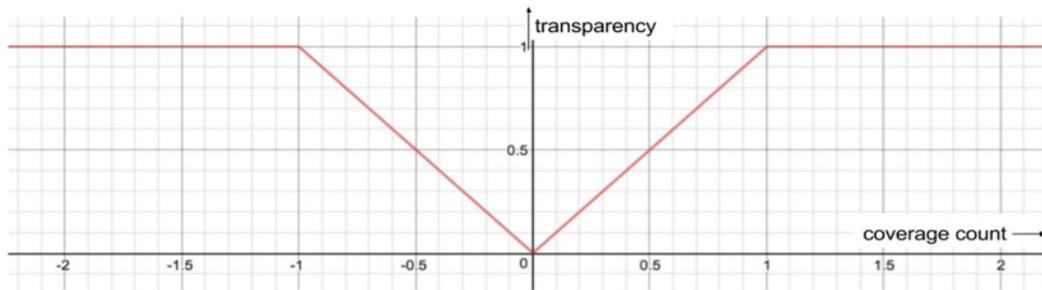between coverage count and transparency is shown in Fig. 6.



**Fig. 6: Map between coverage count and transparency of a pixel (winding fill rule)**

The "winding fill rule" of Fig. 6 is mathematically expressed as

```
alpha = min(abs(coverage_count), 1.0)),
```

where alpha represents transparency of the pixel.
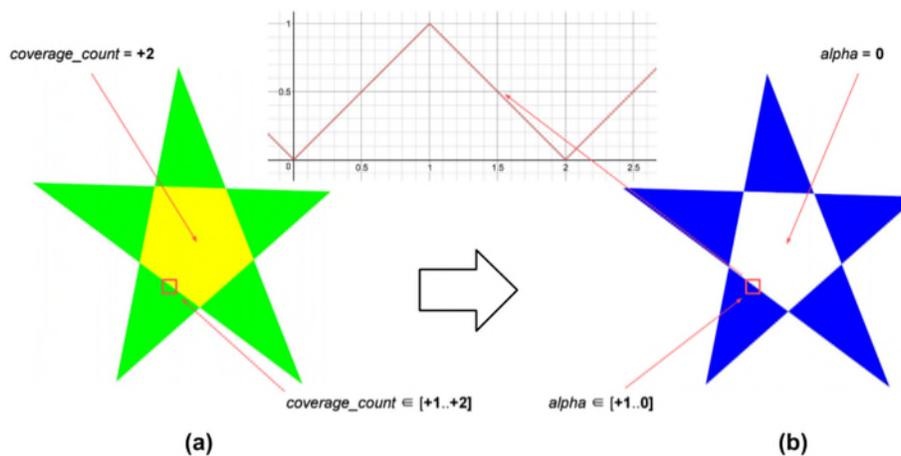
Example 2: *Even/odd fill rule*



**Fig. 7: Even/odd fill rule transforms (a) coverage count to (b) pixel transparency**

Under the even/odd fill rule of Fig. 5, a star-shaped polygon with coverage count of Fig. 7(a) maps to the transparency of Fig. 7(b). Thus, the even/odd fill rule is useful for polygons with holes within them.
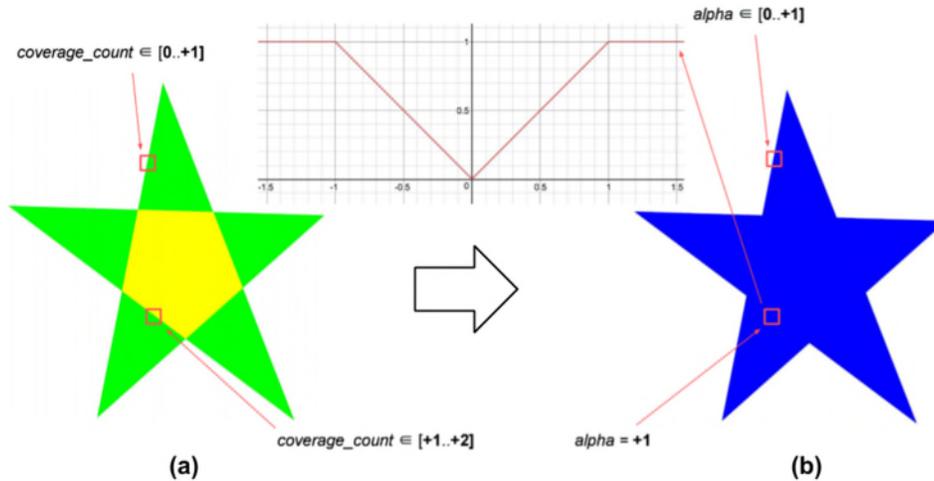
Example 3: *Winding fill rule*



**Fig. 8: Winding fill rule transforms (a) coverage count to (b) pixel transparency**

Under the winding fill rule of Fig. 6, a star-shaped polygon with coverage count of Fig. 8(a) maps to the transparency of Fig. 8(b). Thus, the winding fill rule is useful for polygons that have no holes within them ("simply connected polygons").

CONCLUSION

This disclosure provides techniques for efficient GPU rendering of paths constructed from closed, oriented contours comprising piecewise Bezier segments. Techniques provided herein reduce or eliminate aliasing at the edges of shapes. The techniques can be implemented without expensive, power-hungry operations such as hardware multisampling. The disclosed path-rendering techniques can handle dynamic scenarios such as scaling, animation, etc.

Efficient, low-power path-rendering, per techniques disclosed herein, opens up the possibility of real-time vector graphics content on consumer handheld devices, e.g., devices with limited computing capacity and/or battery power. Techniques that utilize surveyor's algorithm can also benefit from coverage counting.