CHAPTER 6

A Fast and Robust Method for Avoiding Self-Intersection

Carsten Wächter and Nikolaus Binder
NVIDIA

ABSTRACT

We present a solution to avoid self-intersections in ray tracing that is more robust than current common practices while introducing minimal overhead and requiring no parameter tweaking.

6.1 INTRODUCTION

Ray and path tracing simulations construct light paths by starting at the camera or the light sources and intersecting rays with the scene geometry. As objects are hit, new rays are generated on these surfaces to continue the paths. In theory, these new rays will not yield an intersection with the same surface again, as intersections at a distance of zero are excluded by the intersection algorithm. In practice, however, the finite floating-point precision used in the actual implementation often leads to false positive results, known as *self-intersections*, creating artifacts such as shadow acne, where the surface sometimes improperly shadows itself.

The most widespread solutions to work around the issue are not robust enough to handle a variety of common production content and may even require manual parameter tweaking on a per-scene basis. Alternatively, a thorough numerical analysis of the source of the numerical imprecision allows for robust handling. However, this comes with a considerable performance overhead and requires source access to the underlying implementation of the ray/surface intersection routine, which is not possible in some software APIs and especially not with hardware-accelerated technology, e.g., NVIDIA RTX.

In this chapter we present a method that is reasonably robust, does not require any parameter tweaking, and at the same time introduces minimal overhead, making it suitable for real-time applications as well as offline rendering.

E. Haines, T. Akenine-Möller (eds.), *Ray Tracing Gems*, https://doi.org/10.1007/978-1-4842-4427-2_6

6.2 METHOD

Computing a new ray origin in a more robust way consists of two steps. First, we compute the intersection point from the ray tracing result so that it is as close to the surface as possible, given the underlying floating-point mathematics. Second, as we generate the next ray to continue the path, we must take steps to avoid having it intersect the same surface again. Section 6.2.2 explains common pitfalls with existing methods, as well as presents our solution to the problem.

6.2.1 CALCULATING THE INTERSECTION POINT ON THE SURFACE

Calculating the origin of the next ray along the path usually suffers from finite precision. While the different ways of calculating the intersection point are mathematically identical, in practice, the choice of the most appropriate method is crucial, as it directly affects the magnitude of the resulting numerical error. Furthermore, each method comes with its own set of trade-offs.

Computing such a point is commonly done by inserting the hit distance into the ray equation. See Figure 6-1. We strongly advise against this procedure, as the resulting new origin may be far off the plane of the surface. This is, in particular, true for intersections that are far away from the ray origin: due to the exponential scale of floating-point numbers, the gaps between representable values grow exponentially with intersection distance.

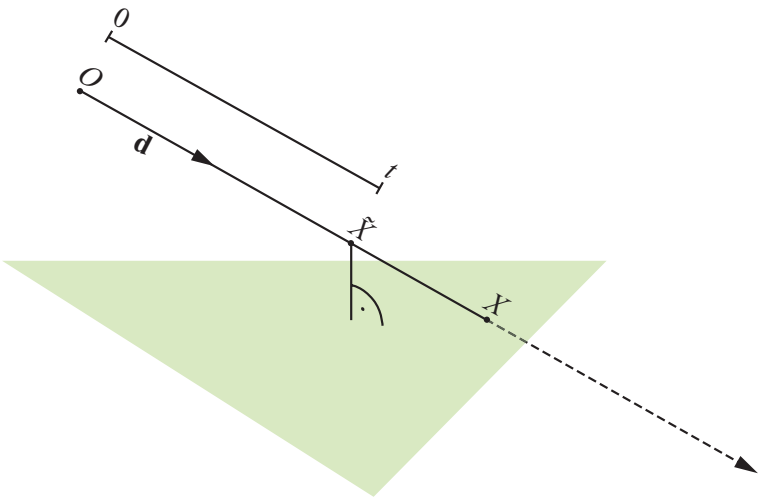

Figure 6-1. *Calculating the ray/surface intersection point X by inserting the intersection distance t into the ray equation. In this case, any error introduced through insufficient precision for t will mostly shift the computed intersection point $\tilde{X}$ along the ray direction* ***d****—and, typically, away from the plane of the triangle.*

By instead calculating the previous ray's intersection point based on surface parameterization (e.g., using the barycentric coordinates computed during ray/primitive intersection), the next ray's origin can be placed as close as possible to the surface. See Figure 6-2. While again finite precision computations result in some amount of error, when using the surface parameterization this error is less problematic: when using the hit distance, any error introduced through finite precision shifts the computed intersection point mostly along the line of the original ray, which is often away from the surface (and consequently bad for avoiding self-intersections, as some points will end up in front of and some behind the surface). In contrast, when using the surface parameterization, any computational error shifts the computed intersection point mostly along the surface—meaning that the next ray's origin may start slightly off the line of the preceding ray, but it is always as close as possible to the original surface. Using the surface parameterization also guarantees consistency between the new origin and surface properties, such as interpolated shading normals and texture coordinates, which usually depend on the surface parameterization.

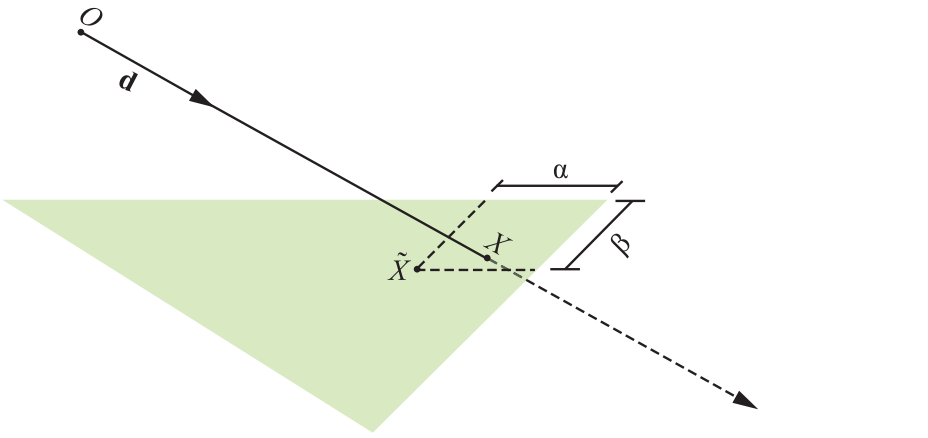

Figure 6-2. *Calculating the intersection X with barycentric coordinates (α, β). In this case, the finite precision of (α, β) means that the computed intersection point $\tilde{X}$ may no longer lie exactly on the ray—but it will always be very close to the surface.*

6.2.2 AVOIDING SELF-INTERSECTION

Placing the origin of the new ray "exactly" on the surface usually still results in self-intersection [4], as the computed distance to the surface is not necessarily equal to zero. Therefore, excluding intersections at zero distance is not sufficient, and self-intersection must be explicitly avoided. The following subsections present an overview of commonly used workarounds and demonstrate the failure cases for each scheme. Our suggested method is described in Section 6.2.2.4.

6.2.2.1 EXCLUSION USING THE PRIMITIVE IDENTIFIER

Self-intersection can often be avoided by explicitly excluding the same primitive from intersection using its identifier. While this method is parameter free, is scale invariant, and does not skip over nearby geometry, it suffers from two major problems. First, intersections on shared edges or coplanar geometry, as well as new rays at grazing angles, still cause self-intersection (Figures 6-3 and 6-4). Even if adjacency data is available, it would be necessary to distinguish between neighboring surfaces that form concave or convex shapes. Second, duplicate or overlapping geometry cannot be handled. Still, some production renderers use the identifier test as one part of their solution to handle self-intersections [2].

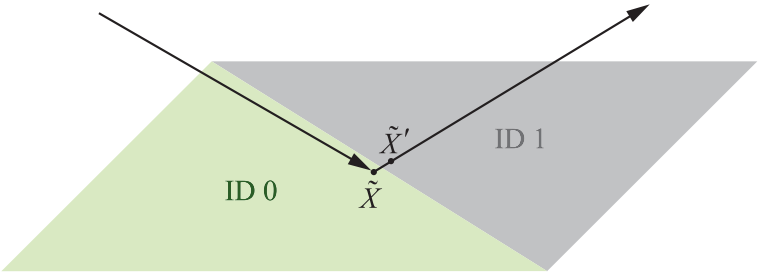

Figure 6-3. *Rejecting the surface whose primitive identifier matches the ID of the primitive on which the previous intersection $\tilde{X}$ was found can fail for the next intersection $\tilde{X}'$ if the previous intersection $\tilde{X}$ was on, or very close to, a shared edge. In this example $\tilde{X}$ was found on the primitive with ID 0. Due to finite precision a false next intersection $\tilde{X}'$ will be detected on the primitive with ID 1 and is considered valid since the IDs mismatch.*

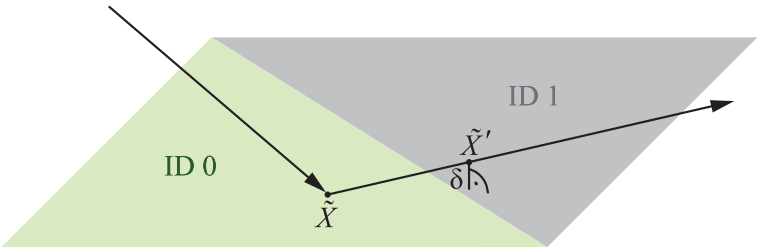

Figure 6-4. *Rejection with primitive IDs also fails on flat or slightly convex geometry for intersections anywhere on the primitive if the next ray exists at a grazing angle. Again, the distance δ of the false intersection $\tilde{X}'$ to the surface of the other primitive gets arbitrarily close to zero, the primitive IDs mismatch, and hence this false intersection is considered valid.*

Furthermore, note that exclusion using the primitive identifier is applicable to only planar surfaces, as nonplanar surfaces can exhibit valid self-intersection.

6.2.2.2 LIMITING THE RAY INTERVAL

Instead of only excluding intersections at zero distance, one can set the lower bound for the allowed interval of distances to a small value ε: $t_{\text{min}} = \varepsilon > 0$. While there is no resulting performance overhead, the method is extremely fragile as the value of ε itself is scene-dependent and will fail for grazing angles, resulting in self-intersection (Figure 6-5) or skipping of nearby surfaces (Figure 6-6).

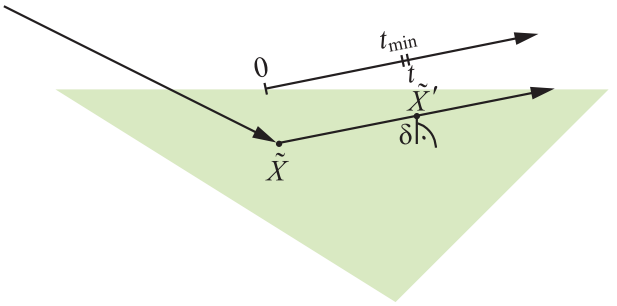

Figure 6-5. *Setting t_{min} to a small value $\varepsilon > 0$ does not robustly avoid self-intersection, especially for rays exiting at grazing angles. In the example the distance t along the ray is greater than t_{min}, but the distance δ of the (false) next intersection $\tilde{X}'$ to the surface is zero due to finite precision.*

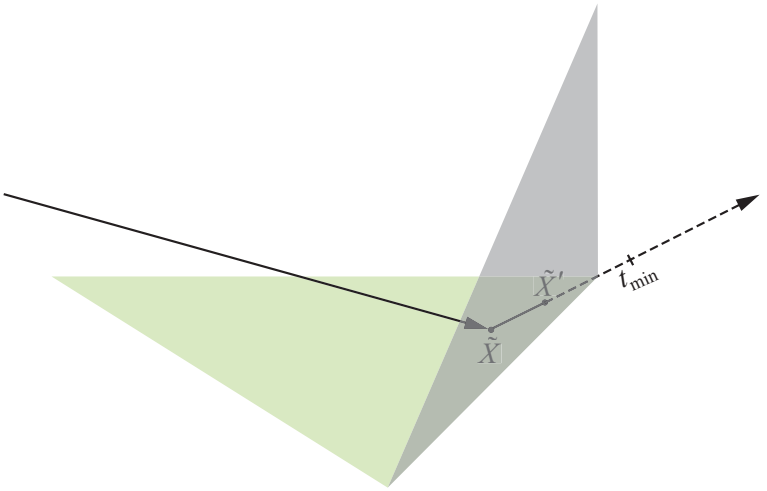

Figure 6-6. *Skipping over a valid intersection $\tilde{X}'$ due to setting $t_{min} = \varepsilon > 0$ is especially visible in corners due to paths being pushed into or out of closed objects.*

6.2.2.3 OFFSETTING ALONG THE SHADING NORMAL OR THE OLD RAY DIRECTION

Offsetting the ray origin along the shading normal is similar to setting the lower bound of a ray $t_{\text{min}} = \varepsilon > 0$ and features the same failure cases, as this vector is not necessarily perpendicular to the surface (due to interpolation or variation computed from bump or normal maps).

Shifting the new ray origin along the old ray direction will again suffer from similar issues.

6.2.2.4 ADAPTIVE OFFSETTING ALONG THE GEOMETRIC NORMAL

As could be seen in the previous subsections, only the geometric normal, being orthogonal to the surface by design, can feature the smallest offset, dependent on the distance to the intersection point, to escape self-intersection while not introducing any of the mentioned shortcomings. The next step will focus on how to compute the offset to place the ray origin along it.

Using any offset of fixed length ε is not scale invariant, and thus not parameter free, and will also not work for intersections at varying magnitudes of distance. Therefore, analyzing the error of the floating-point calculations to compute the intersection point using barycentric coordinates reveals that the distance of the intersection to the plane of the surface is proportional to the distance from the origin (0,0,0). At the same time the size of the surface also influences the error and even becomes dominant for triangles very close to the origin (0,0,0). Using only normalized ray directions removes the additional impact of the length of the ray on the numerical error. The experimental results in Figure 6-7 for random triangles illustrate this behavior: We calculate the average and maximum distance of the computed intersection point to 10 million triangles with edge lengths between 2^{-16} and 2^{22}. As the resulting point can be located on either side of the actual plane, a robust offset needs to be at least as large as the maximum distance.

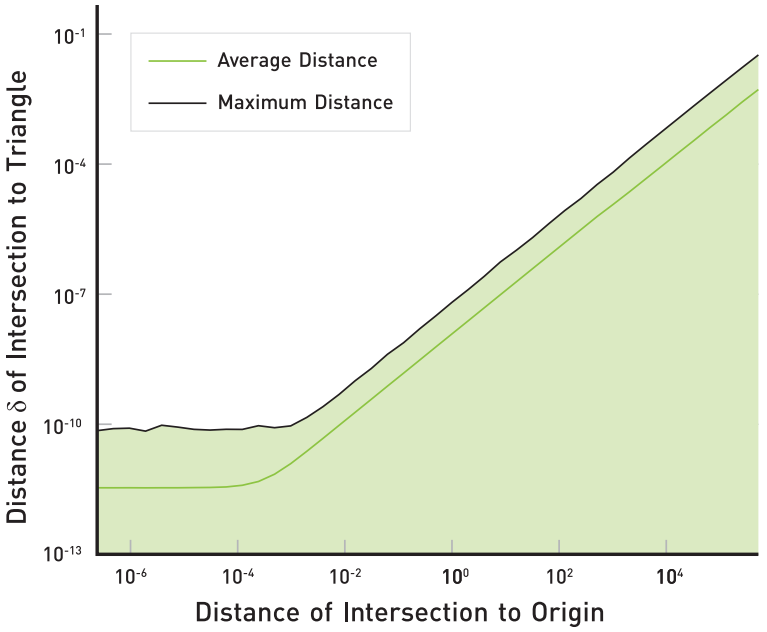

Figure 6-7. *The experimental analysis of the average and maximum distance of a point placed on a triangle using barycentric coordinates to its plane for 10 million random triangles at different distances to the origin provides the scale for the constants used in Listing 6-1.*

To handle the varying distance of the intersection point implicitly, we use integer mathematics on the floating-point number integer representation when offsetting the ray origin along the direction of the geometric normal. This results in the offset becoming scale-invariant and thus prevents self-intersections at distances of different magnitudes.

To handle surfaces/components of the intersection point that are nearly at the origin/zero, we must approach each one separately. The floating-point exponent of the ray direction components will differ greatly from the exponents of the components of the intersection point; therefore, offsetting using the fixed integer ε is not a viable option for dealing with the numerical error that can arise during the ray/plane intersection calculations. Thus, a tiny constant floating-point value ε is used to handle this special case to avoid introducing an additional costly fallback. The resulting source code is shown in Listing 6-1. The provided constants were chosen according to Figure 6-7 and include a small margin of safety to handle more extreme cases that were not included in the experiment.

Listing 6-1. *Implementation of our method as described in Section 6.2.2.4.*

```
constexpr float origin()      { return 1.0f / 32.0f; }
constexpr float float_scale() { return 1.0f / 65536.0f; }
constexpr float int_scale()   { return 256.0f; }

// Normal points outward for rays exiting the surface, else is flipped.
float3 offset_ray(const float3 p, const float3 n)
{
  int3 of_i(int_scale() * n.x, int_scale() * n.y, int_scale() * n.z);

  float3 p_i(
      int_as_float(float_as_int(p.x)+((p.x < 0) ? -of_i.x : of_i.x)),
      int_as_float(float_as_int(p.y)+((p.y < 0) ? -of_i.y : of_i.y)),
      int_as_float(float_as_int(p.z)+((p.z < 0) ? -of_i.z : of_i.z)));

  return float3(fabsf(p.x) < origin() ? p.x+ float_scale()*n.x : p_i.x,
                fabsf(p.y) < origin() ? p.y+ float_scale()*n.y : p_i.y,
                fabsf(p.z) < origin() ? p.z+ float_scale()*n.z : p_i.z);
}
```

Even with our method, there still exist situations in which shifting along the geometric normal skips over a surface. An example of such a situation is the crevice shown in Figure 6-8. Similar failure cases can certainly be constructed and do sometimes happen in practice. However, they are significantly less likely to occur than the failure cases for the simpler approaches discussed previously.

Figure 6-8. *Very fine geometric detail such as a deep, thin crevice cannot be robustly handled by any of the listed methods. In this example the initial intersection $\tilde{X}$ is slightly below the actual surface. Left: limiting the ray interval can help to avoid self-intersection for some rays (upper ray), but may also fail for others (lower ray). Right: offsetting along the surface normal may move the origin of the next ray $\tilde{X}'$ into the same or neighboring object.*

6.3 CONCLUSION

The suggested two-step procedure for calculating a robust origin for the next ray along a path first sets an initial position as close as possible to the plane of the surface using the surface parameterization. It then shifts the intersection away from the surface by applying a scale-invariant offset to the position, along the geometric normal. Our extensive evaluation shows that this method is sufficiently robust in practice and is simple to include in any existing renderer. It has been part of the Iray rendering system for more than a decade [1] to avoid self-intersection for triangles.

The remaining failure cases are rare special cases, but note that huge translation or scaling values in instancing transformations will result in larger offset values as well (for an analysis, see *Physically Based Rendering* (third edition) [3]). This phenomenon leads to a general quality issue because all lighting, direct and indirect, will be noticeably "offset" as well, which becomes apparent especially in nearby reflections, even leading to artifacts. To tackle this problem, we recommend storing all meshes in world units centered around the origin (0,0,0). Further, one should extract translation and scaling from the camera transformation and instead include them in the object instancing matrices. Doing so effectively moves all calculations closer to the origin (0,0,0). This procedure allows our method to work with the presented implementation and, in addition, avoids rendering artifacts due to large offsets.

As excluding flat primitives using the primitive identifier from the previously found intersection does not result in false negatives, this can in addition be included as a fast and trivial test, often preventing an unnecessary surface intersection in the first place.

REFERENCES

[1] Keller, A., Wächter, C., Raab, M., Seibert, D., van Antwerpen, D., Korndörfer, J., and Kettner, L. The Iray Light Transport Simulation and Rendering System. arXiv, https://arxiv.org/abs/1705.01263, 2017.

[2] Pharr, M. Special Issue On Production Rendering and Regular Papers. *ACM Transactions on Graphics 37*, 3 (2018).

[3] Pharr, M., Jakob, W., and Humphreys, G. *Physically Based Rendering: From Theory to Implementation*, third ed. Morgan Kaufmann, 2016.

[4] Woo, A., Pearce, A., and Ouellette, M. It's Really Not a Rendering Bug, You See... *IEEE Computer Graphics & Applications 16*, 5 (Sept. 1996), 21–25.