STAG: Smart Tools & Apps for Graphics (2014)
Andrea Giachetti (Editor)

Enhanced Sphere Tracing

Benjamin Keinert[1] Henry Schäfer[1] Johann Korndörfer Urs Ganse[2] Marc Stamminger[1]

[1]University of Erlangen-Nuremberg [2]University of Helsinki

Figure 1: *Example scenes showing the techniques presented in this paper. The jewel scene (left) and the city scene (center) both show real-time renderings (21.1 ms (47.4 Hz) and 26.7 ms (37.5 Hz) respectively) of scene geometry entirely represented by a single signed distance bound. The image on the right shows the result of integrating our techniques into a non-real-time GPU path tracer.*

Abstract

In this paper we present several performance and quality enhancements to classical sphere tracing: First, we propose a safe, over-relaxation-based method for accelerating sphere tracing. Second, a method for dynamically preventing self-intersections upon converting signed distance bounds enables controlling precision and rendering performance. In addition, we present a method for significantly accelerating the sphere tracing intersection test for convex objects that are enclosed in convex bounding volumes. We also propose a screen-space metric for the retrieval of a good intersection point candidate, in case sphere tracing does not converge thus increasing rendering quality without sacrificing performance. Finally, discontinuity artifacts common in sphere tracing are reduced using a fixed-point iteration algorithm. We demonstrate complex scenes rendered in real-time with our method. The methods presented in this paper have more universal applicability beyond rendering procedurally generated scenes in real-time and can also be combined with path-tracing-based global illumination solutions.

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Curve, surface, solid, and object representations—Geometric algorithms, languages, and systems I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture

1. Introduction

Sphere tracing has been known as a rendering technique for signed distance bounds for at least 20 years (Hart et al. [HSK89], [Har96]) and has recently seen increased interest due to advances in graphics hardware and the increased significance of procedural content generation. The algorithm has not only been applied to the direct real-time rendering of implicit scene descriptions [RMD11] but also to per-pixel displacement mapping [Don05]. Signed distance bounds – expressed as functions e.g. in a pixel shader – can be a more elegant, compact, and flexible representation of geometry and animation compared to traditional triangle-based or volumetric representations. Sphere tracing as a rendering technique – as opposed to rasterization – reflects a straightforward implementation that enables the practical application of signed distance bound geometry representations in the context of real-time rendering. We show how sphere trac-

© The Eurographics Association 2014.

DOI: 10.2312/stag.20141233

delivered by EUROGRAPHICS DIGITAL LIBRARY www.eg.org diglib.eg.org

ing can be made a viable tool even for rendering complex scenes in real-time, overcoming quality and performance restrictions that are present when using the classical sphere tracing algorithm.

2. Previous work

The basic principle behind sphere tracing was first applied to the rendering of deterministic fractal geometry by Hart et al. [HSK89] in 1989. A canonical overview of this standard method can be found in Hart et al. [Har96]. Based on this method, Evans [Eva06] presented methods for effectively approximating ambient occlusion lighting by exploiting the properties of signed distance bounds.

Singh et al. [SN10] proposed a method for real-time ray tracing of arbitrary implicit surfaces on the GPU. In contrast to sphere tracing based methods the accuracy and performance of their technique depends on a predefined surface dependent marching step size. Additionally their method requires the repeated evaluation of the surface gradient for their Taylor root-containment test. This is not feasible for more complex scenes consisting of multiple primitives.

Suitable methods for generating signed distance functions remain a field of active research. While converting other geometry representations into distance functions via a distance transform is feasible [Lik08], these often suffer from the resulting distance field being of low resolution. Reiner et al. [RMD11] present a system for interactive modeling of analytic descriptions of distance functions, suitable for use in a sphere tracing based rendering system.

An improved prevention of self-intersection in secondary rays by locally selecting an ε value was addressed in [DK06], whose algorithm specifically focusses on ray tracing Bézier patches and triangles.

3. Enhancing sphere tracing

The function $f : \mathbb{R}^3 \to \mathbb{R}$, representing a signed distance bound of an implicit surface $f^{-1}(0)$, can directly be ray-traced using the sphere tracing algorithm as introduced by Hart [Har96]. The function $f : \mathbb{R}^3 \to \mathbb{R}$ is a *signed distance bound* of the corresponding implicit surface $f^{-1}(0)$ if and only if Equation 1 is satisfied. We use $dist(x, f^{-1}(0))$ to denote the Euclidean distance of x from the implicit surface $f^{-1}(0)$.

$$f(x) \leq \begin{cases} -dist(x, f^{-1}(0)) & \text{if } x \text{ inside of } f^{-1}(0) \\ +dist(x, f^{-1}(0)) & \text{if } x \text{ outside of } f^{-1}(0) \end{cases} \quad (1)$$

If equality holds for Equation 1, f is called a *signed distance function* for the surface $f^{-1}(0)$.

Sphere tracing facilitates approximating the intersection of a ray $r(t) = d \cdot t + o$, where d is the normalized direction, o the origin of the ray, and f the signed distance bound, which represents the geometry. This intersection is computed by finding the smallest positive solution t of the root finding problem $f \circ r(t) = 0$.

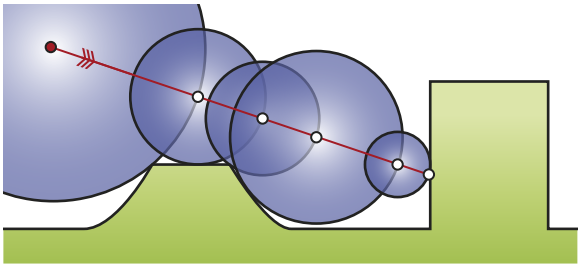

Figure 2: *Illustration of the classical sphere tracing algorithm: The intersection point with a surface is determined by traversing along the ray, using the distance to the closest surface at each iteration step until the distance is below a threshold*

```
// o, d : ray origin, direction (normalized)
// t_min, t_max: minimum, maximum t values
// tau: radius threshold
float t = t_min;
int   i = 0;
while (i < MAX_ITERATIONS && t < t_max) {
   float radius = f(d*t + o);
   if (radius < tau)
      break;
   t += radius;
   i++;
}
if (i == MAX_ITERATIONS || t > t_max)
   return INFINITY;
return t;
```

Listing 1: *A basic implementation of sphere tracing.*

Figure 2 shows the underlying principle of the algorithm: Starting at $p_0 = o$, a new position p_{i+1} is determined by advancing the previous position p_i along the ray direction d with the radius $f(p_i)$ of the unbounding sphere at this position (i.e. the sphere with radius $f(p_i)$ around p_i which is guaranteed not to intersect the surface $f^{-1}(0)$), yielding the iteration rule $p_{i+1} = p_i + d \cdot f(p_i)$. The iteration can be terminated using different criteria. An example implementation using a maximum number of iterations or a threshold τ as a termination criterion can be found in Listing 1.

In the following sections we present five techniques that expound upon the basic sphere tracing algorithm, two of which led to accelerated traversal of the ray (Sections 3.1 and 3.5), two of which increase visual quality by reducing common artifacts (Sections 3.2 and 3.4) and the last of which presents an optimized approach for preventing self-intersections when tracing through transparent objects (Section 3.3).

© The Eurographics Association 2014.

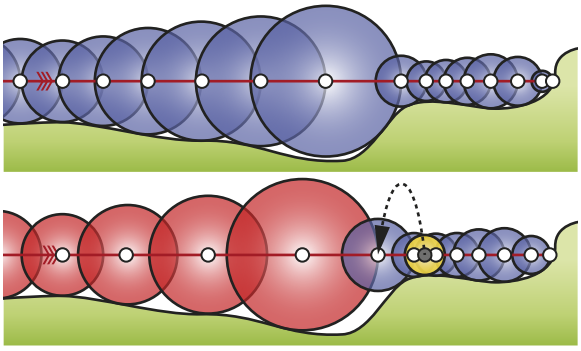

Figure 3: *Comparison of sphere tracing without (top) and with our over-relaxation method (bottom). Blue circles denote iteration steps conducted with standard sphere tracing, red circles denote steps with over-relaxation using* $\omega = 1.6$*. The rightmost red circle and the yellow circle show an iteration step where over-relaxation fails (i.e. the circles do not overlap). The arrow points towards* $p_{fallback}$ *which is used as a starting point for completing tracing of the ray after over-relaxation failure has been detected. Note how fewer iterations are needed to reach a similar position on the ray in the bottom image.*

3.1. Over-relaxation sphere tracing

We apply the principle of over-relaxation to sphere tracing: Instead of stepping along the ray using the radius of the unbounding sphere at each iteration, a step size $\delta_i = f(p_i) \cdot \omega$ can be used, where $f(p_i)$ denotes the value of the signed distance bound at the position of the i-th iteration and $\omega \in [1;2)$ the relaxation parameter. By itself, such over-relaxation can cause stepping into and over objects represented by the signed distance bound. It is therefore necessary to detect and handle these cases.

Whenever the unbounding spheres of two consecutive marching steps overlap, it is ensured that the segment of the ray in between the union of these unbounding spheres cannot intersect any geometry. Using this criterion we can easily detect and handle scenarios in which over-relaxation might be overshot. If $|f(p_{i-1})| + |f(p_i)| < \delta_{i-1}$ over-relaxation may miss an intersection with a surface. In this case our implementation no longer uses over-relaxation and defaults to conventional sphere tracing starting at position $p_{fallback} = p_i + d \cdot \delta_{i-1} \cdot (1 - \omega)$.

Figure 3 (bottom) illustrates defaulting to conventional sphere tracing in cases where a surface intersection may have possibly been missed. It must be noted, however, that if the unbounding sphere around $p_{fallback}$ overlaps with the unbounding sphere around p_i the previous step using the over-relaxation method is guaranteed to not pass through any surface and tracing can be continued from p_i. This is a trivial extension, but requires more branching and state handling in the innermost loop of our GPU implementation, yielding an overall diminished rendering performance due to diverging threads. An implementation of the relaxation technique combined with the method described in the next section can be found in Listing 2.

3.2. Screen-space aware intersection point selection

Whenever the maximum number of iterations has been exceeded, we face the problem of choosing an appropriate point along the ray that is to be regarded as the intersection. This situation occurs frequently when a ray grazes an object's edge without intersecting it, as shown in Figure 4. The large number of iterations needed to clear the grazed object may result in the ray terminating in mid-air behind the object. While increasing the maximum number of iterations can reduce the number of pixels that show this behavior, they can never be completely eliminated and – particularly with moving images and HDR rendering in high-contrast areas – still frequently produce visible artifacts. Conversely, we wish to keep the maximum number of iterations minimal for performance reasons. The traditional approach of picking the last point evaluated is thusly detrimental in this case. Instead, we employ a screen-space error based metric to select the best shading point: We choose the point along the ray with the smallest unbounding sphere radius, as measured in screen-space. Additionally, the same criterion is used to terminate the ray early whenever this radius is smaller than one half the size of a pixel. An implementation of this technique can be found in Listing 2. To compensate for the objects' inflation of half a pixel, a level set of the distance bound can be used, as demonstrated in [Har96]. Figure 8 shows the result of selecting the intersection point using our method versus using the last point along the ray.

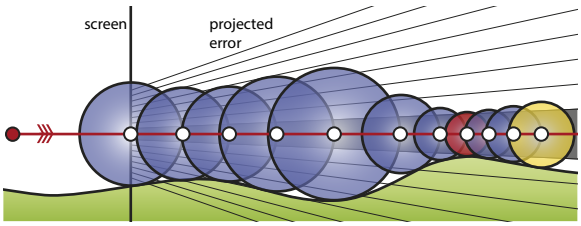

Figure 4: *Intersection point selection based on the screen-space size of the unbounding sphere. The yellow circle represents the unbounding sphere around the very last point evaluated when the maximum number of iterations has been reached. The red circle represents the smallest unbounding sphere measured in screen-space.*

© The Eurographics Association 2014.

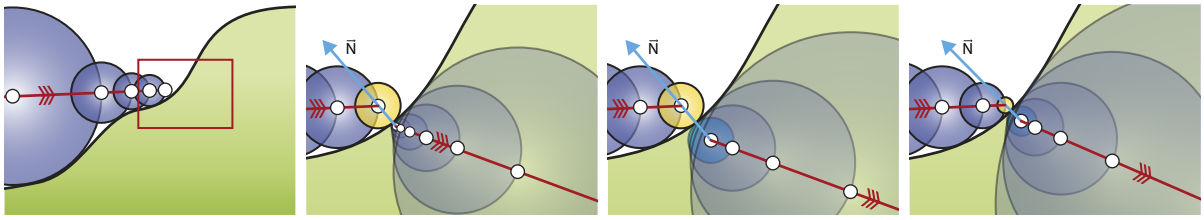

Figure 5: *Self-intersection prevention for refraction rays. The two rightmost images show the use of our $\varepsilon_{dynamic}$. Overview of the scenario (left). Use of an overly small global ε value (center left). Our method (center right): The new ray origin derived by offsetting p (surrounded by a yellow circle) along the normal using $\varepsilon_{dynamic} = 2 \cdot |f(p)|$ since $|f(p)| \geq \varepsilon_{min}$. Usage of ε_{min} (right): The point p is very close to the surface ($|f(p)| \leq \varepsilon_{min}$) and the new ray origin is computed by offsetting along the normal using $\varepsilon_{dynamic} = 2 \cdot \varepsilon_{min}$).*

```
// o, d : ray origin, direction (normalized)
// t_min, t_max: minimum, maximum t values
// pixelRadius: radius of a pixel at t = 1
// forceHit: boolean enforcing to use the
//           candidate_t value as result
float omega = 1.2;
float t = t_min;
float candidate_error = INFINITY;
float candidate_t = t_min;
float previousRadius = 0;
float stepLength = 0;
float functionSign = f(o) < 0 ? -1 : +1;
for (int i = 0; i < MAX_ITERATIONS; ++i) {
    float signedRadius = functionSign * f(d*t + o);
    float radius = abs(signedRadius);

    bool sorFail = omega > 1 &&
        (radius + previousRadius) < stepLength;
    if (sorFail) {
        stepLength -= omega * stepLength;
        omega = 1;
    } else {
        stepLength = signedRadius * omega;
    }

    previousRadius = radius;
    float error = radius / t;

    if (!sorFail && error < candidate_error) {
        candidate_t = t;
        candidate_error = error;
    }

    if (!sorFail && error < pixelRadius || t > t_max)
        break;
    t += stepLength;
}

if ((t > t_max || candidate_error > pixelRadius) &&
    !forceHit) return INFINITY;
return candidate_t;
```

Listing 2: *An implementation of over-relaxation sphere tracing with screen-space-based intersection point selection.*

3.3. Dynamic ε for self-intersection prevention

Preventing self-intersections of secondary rays in ray tracing methods is a well-known and greatly researched problem [DK06]. In many cases, it is sufficient to virtually offset the intersection point using a small global ε value along the ray direction or the normal to retrieve the new origin for a secondary ray.

However, with sphere tracing, the intersection points' distances from the surface can exhibit relatively large variations, particularly in case a screen-space metric is used for iteration termination. Thus, the use of a global ε value is rendered inappropriate for offsetting the ray origin for refraction rays. We resolve this problem by dynamically computing a local value $\varepsilon_{dynamic} = 2 \cdot \max(\varepsilon_{min}, |f(p)|)$ at each intersection point p. This yields $\varepsilon_{dynamic} \geq 2 \cdot \varepsilon_{min}$ which obviates numerical problems (i.e. $f(p) \approx 0$) and chooses an appropriate offset by incorporating the distance between the intersection point and the surface.

Additional care has to be taken when choosing ε_{min} since an excessively small value might not only cause numerical problems, but also diminish the performance of tracing a new refraction ray starting at the offset position as sphere tracing has to accelerate away from the surface initially. However, manually choosing ε_{min} proved to be much easier and more robust than choosing a global ε. The cost for computing $\varepsilon_{dynamic}$ is negligible given that $f(p)$ is already known after tracing to the surface.

3.4. Discontinuity reduction

Usually, a limited number of sphere tracing iterations and a constant threshold τ or a screen-space based criterion for terminating the sphere tracing is utilized. Regardless of the termination criterion, the intersection points' distance from the surface and the distance traveled along the ray are both discontinuous over the screen-space domain as can be seen in Figure 10 (top right).

This can lead to very characteristic and unpleasant artifacts in the resulting image, particularly when procedural texturing is used with the intersection point obtained as an input. To reduce these artifacts and to be able to generate satisfying images even with reduced precision and a low number of iterations, we employ a fixed-point iteration scheme to smooth out the discontinuities in $f(p)$. We aim at points satisfying Equation 2.

$$f(p) = err\left(\|p_i - o\|_2\right) \tag{2}$$

© The Eurographics Association 2014.

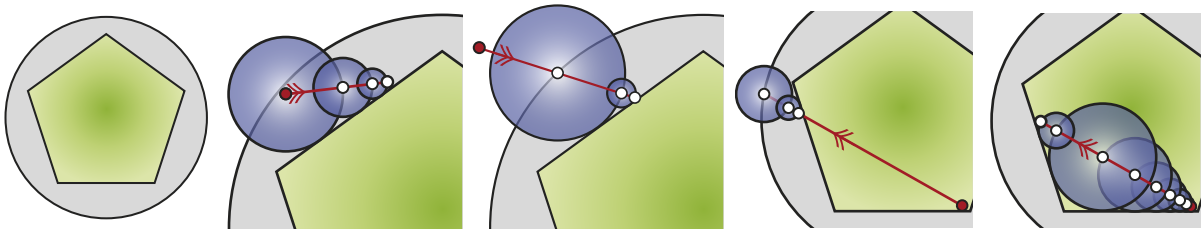

Figure 6: *Illustration of our optimization for sphere tracing through convex objects represented by a signed distance bound. From left to right: Convex object enclosed by a convex bounding volume; ray origin inside bounding volume but outside the object; ray origin outside bounding volume and outside the object; ray origin inside the object and bounding volume with sphere tracing from the outside; the same scenario as the previous with sphere tracing through the interior of the object.*

where o denotes the origin of the ray and the function $err : \mathbb{R} \to \mathbb{R}$ describes the permitted error parameterized by the distance from the origin using a screen-space metric as described by Hart and DeFanti [HD91]. The condition (Equation 2) will be achieved by the following fixed-point iteration scheme:

$$p_{i+1} = p_i + d \cdot \left(f(p_i) - err\left(\|p_i - o\|_2 \right) \right) \tag{3}$$

We employ Equation 3 to iteratively post process the positions p retrieved by sphere tracing beforehand.

3.5. Optimization for convex objects

Hart et al. [Har96] proposed various optimizations for sphere tracing convex objects. Most of these optimizations involve the often costly computation of the gradient per iteration step which, for performance reasons, is not feasible in real-time applications.

Our optimization for convex objects focuses on accelerating tracing through an object represented by a signed distance bound enclosed by a convex bounding object. To determine the intersection of a ray with the object we must handle three cases:

1. Ray origin outside bounding volume
2. Ray origin inside bounding volume and outside the object
3. Ray origin inside bounding volume and inside the object

Case 1 (Figure 6 - center) is handled by sphere tracing from the intersection with the bounding geometry. Case 2 (Figure 6 - center left) can be easily handled by applying sphere tracing from the ray origin to the surface. If the ray is inside the object (Case 3, Figure 6 - center right), instead of tracing through the object, we advance the ray to its intersection with the bounding geometry. Then, we perform sphere tracing in the inverse ray direction, starting at the intersection with the bounding object, which yields the same intersection point as if having traced through the object.

Consequently, costly marching through the inside of the object (Figure 6 - right) can be avoided completely. In comparison to tracing through the object, far fewer iterations are required to find a ray-surface intersection.

4. Results

As shown in the teaser (Figure 1), our techniques can be used in a wide variety of use cases and enable high-quality, real-time rendering of scene representations with a signed distance bound. Except for the rendering of the city scene all images are generated without further image space post-processing effects.

The materials used in our scenes are procedurally generated on the GPU and connected to the signed distance bound, which defines the scene. To evaluate the material at an intersection point, we use modified, constructive solid geometry operators, which also propagate material information. The translucent materials in our real-time scenes are computed in a fashion similar to Whitted [Whi80]. Unless otherwise noted, the timings in this section do not include material evaluation, shading or post processing. We measure the performance of our techniques on an NVIDIA GTX Titan at

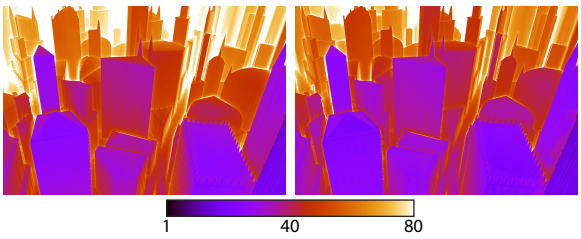

Figure 7: *Comparison of the number of iterations required without our over-relaxation (left) and with ($\omega = 1.2$) (right). The image shows the number of iterations at which a ray terminates. Note, how more buildings become visible in the distance in the right image since the maximum number of iterations (80) is reached less quickly.*

© The Eurographics Association 2014.

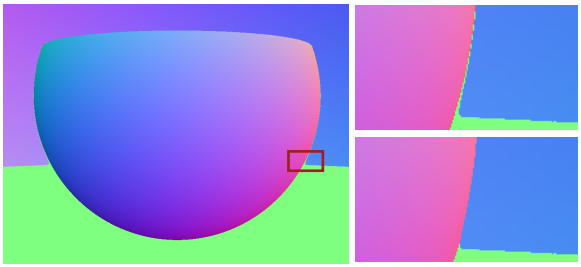

Figure 8: *Rendering object-space normals with (bottom right) and without (top right) screen-space intersection point selection using a maximum of 64 sphere tracing iterations each. Note, how the thin layer of green pixels – depicting the normals of the floor – following the silhouette of the object disappears with our method.*

a resolution of 1280 x 720. The sample scenes achieving real-time rendering performance are entirely implemented in DirectX/DirectCompute and rendered using a single dispatch call. The proposed techniques are not implementation-dependent. Hence, we also integrated these into a GPU path tracer implemented in CUDA to render signed distance bounds with global illumination [Kaj86] as shown in Figure 1 (right). The path tracer uses analytic intersection tests

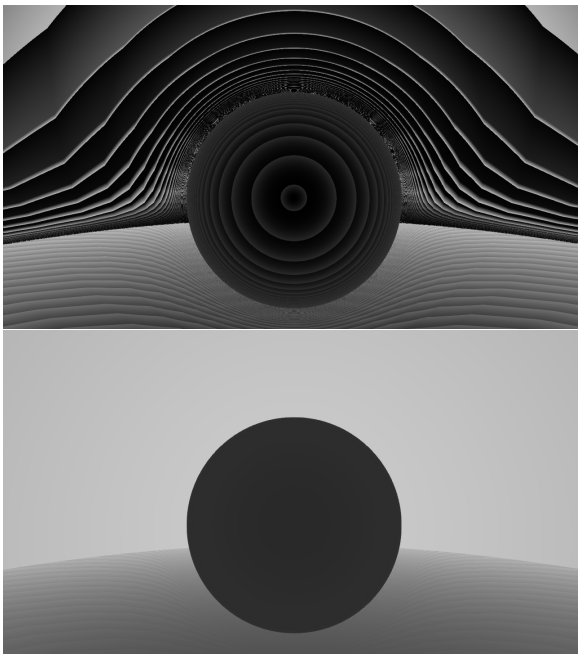

Figure 9: *Visualization of $f(p) \cdot 100$ at the intersection points p without (top) and with (bottom) our discontinuity reduction technique (5 iterations).*

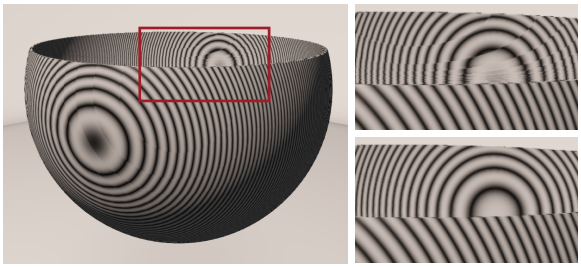

Figure 10: *Our discontinuity reduction applied to a procedurally textured object. The close-up on the right shows a significant quality improvement after applying 3 iterations of the method (bottom).*

for the parts of the scene that are not represented as a signed distance bound, e.g. the walls and the mirror sphere.

Figure 7 shows a comparison of sphere tracing with and without our over-relaxation (Section 3.1) method visualizing an effective decrease in the number of required function evaluations. Our measurements with a maximum of 80 iterations show significant performance improvements for our over-relaxation method:

The function of the city scene shown in the teaser can be ray cast in 16.829 ms (59.421 Hz) compared to 20.409 ms (48.998 Hz) with the original sphere tracing algorithm. Ray casting the jewel scene yields similar results (with over-relaxation 2.978 ms (335.744 Hz) and without 3.472 ms (288.018 Hz)). Applying more complex lighting evaluation to the jewel scene with two reflective and eight refractive bounces using three discontinuity reduction iterations results in 21.11 ms (47.366 Hz) with our method and 22.470 ms (44.503 Hz) without. Improved performance can also be observed for the path tracing integration at 101.934 ms (9.8 Hz) compared to 118.302 ms (8.453 Hz) for one sample for each pixel of the image ($\omega = 1.4$, path tracing depth: 10 bounces).

It should be noted that finding an adequate value for the $\omega \in [1..2]$ parameter can be challenging. Choosing an appropriate ω allows for tracing deeper into the scene before the maximum iteration count is reached and the ray is terminated whereas using a larger ω causes an earlier premature defaulting from over-relaxation to conventional sphere tracing. In our test scenes, $\omega \approx 1.2$ yielded the best performance and allowed for improving visual quality.

The screen-space aware intersection point selection aids in choosing a better intersection point candidate where the maximum number of iterations has been reached but the screen-space error is still above sub-pixel accuracy. This effect is evident in the close-up in Figure 8 where our approach causes less aliasing and a more consistent rendering.

Using a dynamic $\varepsilon_{dynamic}$ allows for robustly preventing self-intersection. We merely need to define a global minimal

© The Eurographics Association 2014.

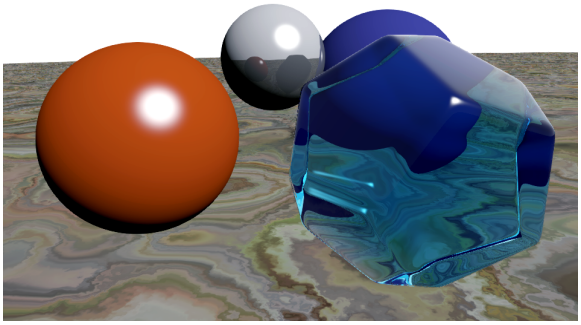

Figure 11: *The test scene for our convex optimization technique. Consisting of 3 spheres and a ground plane (directly ray traced by analytic intersection tests), the dodecahedron is represented by a signed distance bound as proposed by Akleman et al. [AC99] and enclosed by a bounding sphere.*

offset ε_{min} value to obviate numerical problems in case an intersection point is very close to the surface. The value for ε_{min} has to be chosen with care, since too small of an offset can not only cause numerical problems but also decrease the sphere tracing performance by spending a large number of iterations for tracing spheres away from the surface.

For our final quality renderings, we use three iterations of the proposed discontinuity reduction method. Visual artifacts occurring in combination with procedural texturing (Figure 10) can be significantly reduced. As shown in Figure 9 discontinuities in $f(p)$ can drastically be reduced with only a few smoothing iterations at a low, fixed cost. Note that the average of $f(p)$ may be increased after applying our technique, resulting in an overall greater error but in significantly reduced discontinuities, thus yielding better visual quality.

The test scene for the convex optimization technique is shown in Figure 11 and uses analytic intersection tests for the parts of the scene not represented as a signed distance bound. Using two bounces for reflections and eight bounces for refractions, we observed a speed up of around 10% (with convex optimization 2.73 ms, without 3.03 ms).

5. Conclusion

We have presented a number of technical improvements upon the classical sphere tracing algorithm that counteract characteristic artifacts and corner cases normally encountered with this method while at the same time improving ray traversal performance.

First, we presented an over-relaxation method, offering a performance improvement over the classical approach, particularly in applications where secondary rays are cast through objects. We improved the selection of the intersection point candidate by choosing the smallest unbounding sphere, as measured in screen space in case the maximum number of iterations has been reached without reaching convergence. Further, we have shown that a dynamic offset ε for tracing refractive objects allows for using screen-space error metrics and does not necessitate adhering a hard global error threshold, which would be detrimental to performance.

The proposed fixed-point iteration procedure along the ray after the final step of marching significantly reduces artifacts due to depth discontinuities, particularly when using procedural texturing (low cost, high gain). Three iterations of the fixed-point method are usually sufficient to eliminate this class of artifacts.

For convex objects with simplified bounding geometry, significant performance optimizations are possible if multiple intersection tests have to be performed (such as in glass rendering).

For future avenues of research based on these findings, the over-relaxation method could be improved upon by using more sophisticated logic to determine when to re-enable over-relaxation after defaulting to the conservative sphere tracing algorithm. An alternative could be automatic and/or adaptive choice of the over-relaxation parameter ω.

For implementations on GPUs, further research is also required for reducing computational penalties incurred in image areas where the number of steps required for convergence is highly non-uniform, thus leading to diverging threads when execution is performed on GPUs. A good heuristic for estimating the number of iterations required for each pixel would allow for reordering the threads into groups of similar workload and lead to a better utilization of the massive parallel computation power available.

Acknowledgements

This work was partly supported by the Research Training Group 1773 "Heterogeneous Image Systems", funded by the German Research Foundation (DFG).

References

[AC99] AKLEMAN E., CHEN J.: Generalized Distance Functions. In *Proceedings of Shape Modeling and Applications (SMI'99)* (1999), IEEE, pp. 72–79. 7

[DK06] DAMMERTZ H., KELLER A.: Improving ray tracing precision by object space intersection computation. In *Interactive Ray Tracing 2006, IEEE Symposium on* (2006), IEEE, pp. 25–31. 2, 4

[Don05] DONNELLY W.: Per-pixel displacement mapping with distance functions. In *GPU Gems 2* (2005), Addison-Wesley, pp. 123–136. 1

[Eva06] EVANS A.: Fast Approximations for Global Illumination on dynamic Scenes. In *ACM SIGGRAPH 2006 Courses* (2006), ACM, pp. 153–171. 2

[Har96] HART J. C.: Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer 12*, 10 (1996), 527–545. 1, 2, 3, 5

© The Eurographics Association 2014.

[HD91] HART J. C., DEFANTI T. A.: Efficient Antialiased Rendering of 3-D Linear Fractals. In *ACM SIGGRAPH Computer Graphics* (1991), vol. 25, ACM, pp. 91–100. 5

[HSK89] HART J. C., SANDIN D. J., KAUFFMAN L. H.: Ray tracing deterministic 3-d fractals. In *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1989), SIGGRAPH '89, ACM, pp. 289–296. 1, 2

[Kaj86] KAJIYA J. T.: The rendering equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1986), SIGGRAPH '86, ACM, pp. 143–150. 6

[Lik08] LIKTOR G.: Ray tracing implicit surfaces on the gpu. *Computer Graphics & Geometry 10*, 3 (2008), 36–53. 2

[RMD11] REINER T., MÜCKL G., DACHSBACHER C.: Interactive Modeling of Implicit Surfaces using a Direct Visualization Approach with Signed Distance Functions. *Computers & Graphics 35*, 3 (2011), 596–603. 1, 2

[SN10] SINGH J. M., NARAYANAN P. J.: Real-time ray tracing of implicit surfaces on the gpu. *IEEE Transactions on Visualization and Computer Graphics 16*, 2 (2010), 261–272. 2

[Whi80] WHITTED T.: An improved illumination model for shaded display. *Commun. ACM 23*, 6 (June 1980), 343–349. 5

© The Eurographics Association 2014.