Ray-tracing Procedural Displacement Shaders

Wolfgang Heidrich and Hans-Peter Seidel

Computer Graphics Group

University of Erlangen

{heidrich,seidel}@informatik.uni-erlangen.de

Abstract

Displacement maps and procedural displacement shaders are a widely used approach of specifying geometric detail and increasing the visual complexity of a scene.

While it is relatively straightforward to handle displacement shaders in pipeline based rendering systems such as the Reyes-architecture, it is much harder to efficiently integrate displacement-mapped surfaces in ray-tracers. Many commercial ray-tracers tessellate the surface into a multitude of small triangles. This introduces a series of problems such as excessive memory consumption and possibly undetected surface detail.

In this paper we describe a novel way of ray-tracing procedural displacement shaders directly, that is, without introducing intermediate geometry. Affine arithmetic is used to compute bounding boxes for the shader over any range in the parameter domain. The method is comparable to the direct ray-tracing of Bézier surfaces and implicit surfaces using Bézier clipping and interval methods, respectively.

Keywords: ray-tracing, displacement-mapping, procedural shaders, affine arithmetic

1 Motivation

Procedural displacement shaders [7] are an important means of specifying geometric detail and surface imperfections in synthetic scenes, which increases the realism of computer generated images. Small surface features, which would be tedious to model explicitly using a geometric modeling system, can often be described by a few lines of code in a dedicated shading language, such as the RenderMan Shading Language [20, 10, 24].

In contrast to simple surface shaders and bump maps [4], displacement shaders actually change the geometry of the underlying surface. This allows for effects like self-occlusion and self-shadowing, and results in more realistically looking images, in particular in silhouette regions (see Figure 1). Procedural displacement shaders are resolution independent and storage efficient.
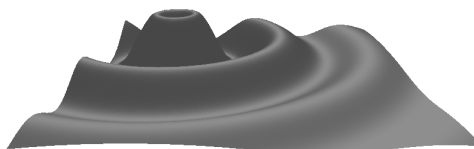

Figure 1: A simple displacement shader showing a wave function with exponential decay. The underlying geometry used for this image is a planar polygon.

Unfortunately, because displacement shaders modify the geometry, their use in many rendering systems is limited. Since ray-tracers cannot handle procedural displacements directly, many commercial ray-tracing systems, such as Alias [1], tessellate geometric objects with displacement shaders into a multitude of small triangles. Unlike in pipeline-based renderers, such as the Reyes-architecture [17], these triangles have to be kept around, and cannot be rendered independently.

This approach, however, defeats two of the reasons for using procedural displacements in the first place: resolution independence and storage efficiency. The storage requirements for representing a large number of polygons (often hundreds of thousands or more), as well as the spatial subdivision hierarchy required for speeding up the intersection tests (often octrees) can be excessive, even for relatively simple shaders.

But tessellation-based methods also bear other, more fundamental problems. Because the shader can only be evaluated at a discrete set of points, the vertices of the output polygons, it is possible to miss narrow features. This is particularly disturbing if the displacements of those features are large. Most existing systems either disallow long and skinny displacements, or require the user to explicitly specify bounding volumes for them.

As a final problem, numerical errors may occur in intersection tests with many small polygons. These can result in cracks between adjacent polygons, which are difficult to avoid in implementations.

Given the potential benefits of displacement shaders, and the shortcomings of tessellation-based implementations, the question is whether there is a way to *directly* ray-trace procedural displacements. Such an algorithm should work directly on the procedural description, without generating any intermediate geometry. This would prevent cracking and preserve resolution independence. The method should also *guarantee* intersections, in the sense that if an intersection exists, it will be found, no matter how fine the feature is. Since intersections with procedural displacements cannot be determined analytically, the algorithm should use a numerical method to iteratively compute the closest intersection along a ray.

2   Prior Work

Several related algorithms for finding intersections with a variety of different geometric objects have been published. Nishita et al. [16] developed Bézier clipping to compute ray intersections with (rational) Bézier patches. The algorithm first determines parameter regions that cannot contain intersection points. The surface is the re-parameterized over the remaining parameter range. This process is recursively repeated until the intersection point has been determined with a given precision.

Barr [2, 3] computed ray intersections with deformed objects by solving an initial value problem. This method works for both parametric and implicit surfaces, but only for relatively simple global deformations such as twists. It does not directly translate to complex procedural deformations.

Later, Kalra and Barr [13] developed a method for finding *guaranteed* intersections with implicit surfaces. Lipschitz bounds (global bounds on the derivative of the implicit function) are used to spatially subdivide 3-space into cells, each of which contains at most one intersection of the surface with the ray. The exact intersection point is then found using the Newton method or regula falsi. Unfortunately, the computation of Lipschitz bounds cannot be automated, and thus the use of this algorithm is limited to cases where it is acceptable for the user to specify them.

Snyder [22, 23] and Duff [6] used interval arithmetic to recursively enumerate implicit surfaces with a hierarchical data structure. For each cell, the implicit function is evaluated using interval arithmetic. This yields bounds for the value of the function in this cell. If these bounds include the value zero, the

surface potentially passes through the cell, and the cell is subdivided in an octree fashion, unless it is already small enough. Unlike Lipschitz bounds, interval bounds can be automatically computed by specific implementations of the basic library functions.

Comba and Stolfi [5] and Figueiredo and Stolfi [9] used a similar algorithm, but replaced interval arithmetic with affine arithmetic, which they developed for this purpose. They showed that affine arithmetic produces significantly tighter bounds than interval arithmetic, although at a higher cost.

In [12], we used affine arithmetic to obtain conservative bounds for the value of procedural surface shaders over a given parameter interval. This can be used to drive a hierarchical sampling process, for example in hierarchical radiosity.

In this paper we extend the above ideas to ray-tracing displacement shaders. We use affine arithmetic to hierarchically enumerate the geometry generated by procedural shaders. In Section 3 we describe the basic algorithm. In Section 4 we then briefly review affine arithmetic, and discuss the issues involved when applying it to procedural shaders. Sections 5 and 6 contain refinements of the basic algorithm. Finally, in Section 7, we present experimental results obtained with our method.

3   Direct Ray-tracing of Displacement Shaders

Our iterative method for intersection tests with displacement shaders is based on the idea of hierarchically subdividing the parameter domain of the surface. Parameter regions, in which intersections might occur, are identified, and those regions in which intersections are not possible are discarded. Parameter ranges containing potential intersections are recursively refined, until the intersection has been determined within a pre-defined accuracy. The pseudo-code in Figure 2 summarizes this algorithm.

In order to detect potential intersections in a given parameter range, we compute a bounding box for the value of the displacement shader over this range. The recursion is terminated if the bounding box is small enough. For primary rays this means that the projection of the box onto the screen is smaller than some fraction of a pixel. As we will show in Section 3.1, the bounding boxes can be obtained with the help of affine arithmetic.

Figure 3 shows an example of this method. On the left side, you see the hierarchically subdivided parameter domain for the intersection of a ray with the simple shader from Figure 1. On the right side,

```
intersect(ray,shader,u1,u2,v1,v2)
{
  bBox= computeBBox(shader,u1,u2,v1,v2);
  if( intersects(ray,bBox) )
    if( isSmallEnough(bBox) )
      return bBox;
    else
    {
      ll= intersect(ray,shader,
          u1,(u1+u2)/2,v1,(v1+v2)/2);
      lr= intersect(ray,shader,
          (u1+u2)/2,u2,v1,(v1+v2)/2);
      ul= intersect(ray,shader,
          u1,(u1+u2)/2,(v1+v2)/2,v2);
      ur= intersect(ray,shader,
          (u1+u2)/2,u2,(v1+v2)/2,v2);
      return closest(ll,lr,ul,ur);
    }
  return NONE;
}
```

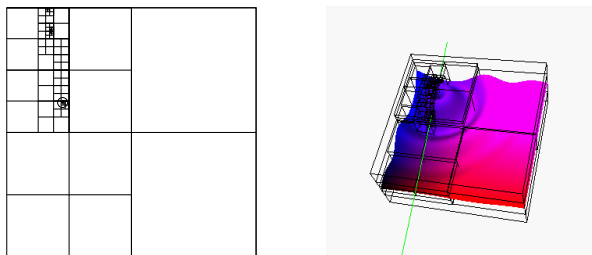Figure 2: The basic intersection algorithm.

Figure 3: The hierarchically subdivided parameter domain, and the resulting bounding boxes for the intersection of one ray with the wave shader from Figure 1.

you see the displaced surface with the resulting hierarchy of bounding boxes.

3.1 Bounding Box Computation

To understand how the bounding boxes can be computed, we first describe how displacement shaders can be conceptually integrated into the rendering system.

Displacement shaders are procedures that describe perturbations of points in 3-space. For the purposes of this paper, we assume that this process is strictly separated from other parts of the renderer, in particular geometry processing and surface shading. Displacement shaders use a well-defined interface to communicate with other parts of the system. An example for such an interface is defined by the RenderMan Shading Language [20, 10], on which we base our discussion.

The displacement shader communicates with two other parts of the rendering system, the geometry processing stage and a surface shading stage. The data flow between those parts is depicted in Figure 4.
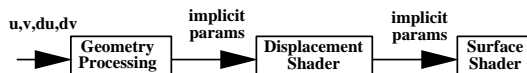

Figure 4: The data flow between selected parts of the rendering system.

Given a parametric position (u, v) and sampling rate (du, dv), the geometry processing stage computes a set of purely geometric terms for a given parametric surface. This includes the point in 3-space, as well as the geometric and the shading normal in this point.

This set of geometric terms is then passed on to the displacement shader, which is allowed to change the position of the point, as well as the shading normal. The most important parameters of a displacement shader are listed in Table 1. The shader does not have direct access to the underlying geometry of the surface.

Variable	Description
P	Position in 3-space
N	Shading normal in point P
Ng	Surface normal in point P
E	Eye point
u, v	Surface parameters
du, dv	Change of surface parameters

Table 1: The most important implicit parameters of displacement shaders. P and N can be modified by the shader.

The perturbed point and normal, together with information from other parts of the system, such as incoming light directions and intensities, are then passed to the surface shader. There, they are used to determine the intensity of the point.

The data flow for the three modules shows the difficulty of ray-tracing procedural displacements: in order to evaluate the shader, we require a parametric position (u, v). This position, however, is not known before the intersection has been determined. This illustrates the necessity to iterate the first two stages of Figure 4 in order to find both the intersection, and the parametric position of it.

Both the parametric function of the surface and

the procedural displacement can usually be evaluated only at discrete points in parameter space. However, in order to compute a bounding box for a specific parameter range, we have to compute conservative bounds for the value of the shader over that range. This can be done using an automated range analysis method such as affine arithmetic.

Using affine arithmetic instead of normal floating point arithmetic for the evaluation of the respective parametric formulas for the geometry, we obtain conservative bounds for parameters of the displacement shader, such as the surface point and the normal. Conservative bounds for the displaced point and normal can then be computed by evaluating the shader, again using affine- instead of floating point arithmetic. The ranges of the x, y, and z components of the resulting point are at the same time bounding boxes for the geometry after the displacement. In the following we give a brief overview of affine arithmetic, and describe in more detail how it can be applied to procedural shaders.

4 Affine Arithmetic

Affine arithmetic, first introduced in [5], is an extension of interval arithmetic [14]. It has been successfully applied to several problems for which interval arithmetic had been used before [15, 22, 23]. This includes the adaptive enumeration of implicit surfaces [9].

Like interval arithmetic, affine arithmetic can be used for manipulating imprecise values, and for evaluating functions over intervals. It is also possible to keep track of truncation and round-off errors. In contrast to interval arithmetic, affine arithmetic also maintains dependencies between the sources of error, and thus manages to compute significantly tighter error bounds.

Affine arithmetic operates on quantities known as *affine forms*, given as polynomials of degree one in a set of *noise symbols* ϵ_i.

$$\hat{x} = x_0 + x_1\epsilon_1 + x_2\epsilon_2 + \cdots + x_n\epsilon_n$$

The coefficients x_i are known real values, while the values of the noise symbols are unknown, but limited to the interval $\mathbf{U} := [-1, 1]$. Thus, if all noise symbols can independently vary between -1 and 1, the range of possible values of an affine form $\hat{x}$ is

$$[\hat{x}] = [x_0 - \xi, x_0 + \xi], \quad \xi = \sum_{i=1}^{n} |x_i|.$$

Computing with affine forms is a matter of replacing each elementary operation and library function

$f(x)$ on real numbers with an adequate operation $f^*(\epsilon_1, \ldots, \epsilon_n) := f(\hat{x})$ on affine forms.

If f is itself an affine function of its arguments, normal polynomial arithmetic yields the corresponding operation f^*. For example, we get

$$\hat{x} + \hat{y} = x_0 + y_0 + (x_1 + y_1)\epsilon_1 + \cdots + (x_n + y_n)\epsilon_n,$$
$$\hat{x} + \alpha = (x_0 + \alpha) + x_1\epsilon_1 + \cdots + x_n\epsilon_n,$$
$$\alpha\hat{x} = \alpha x_0 + \alpha x_1\epsilon_1 + \cdots + \alpha x_n\epsilon_n$$

for affine forms $\hat{x}, \hat{y}$ and real values α.

If f is not an affine operation, the function $f^*(\epsilon_1, \ldots, \epsilon_n)$ cannot be exactly represented as a linear polynomial in the ϵ_i. In this case, it is necessary to find an affine function $f^a(\epsilon_1, \ldots, \epsilon_n) = z_0 + z_1\epsilon_1 + \cdots + z_n\epsilon_n$ that approximates $f^*(\epsilon_1, \ldots, \epsilon_n)$ as well as possible over $\mathbf{U}^n$. An additional *new* noise symbol ϵ_{n+1} has to be added to represent the error introduced by this approximation. This yields the following affine form for the operation $z = f(x)$:

$$\hat{z} = f^a(\epsilon_1, \ldots, \epsilon_n)$$
$$= z_0 + z_1\epsilon_1 + \cdots + z_n\epsilon_n + z_{n+1}\epsilon_{n+1}$$

The coefficient x_{n+1} of the newly introduced noise symbol ϵ_{n+1} has to be an upper bound for the error introduced by the approximation of f^* with f^a.

The process of generating affine approximations f^a for arbitrary functions f is described in detail in [5], [8] and [9]. This process is relatively straightforward for most library functions. Once all library functions have been implemented using affine arithmetic, they can be used to compute conservative bounds for any expression that only uses functions from this library.

4.1 Affine Arithmetic and Procedural Shaders

In addition to the functions found in typical math libraries, shading languages often provide specific additional functionality. This includes a set of problem-specific functions, as well as derivatives of arbitrary expressions, and, of course, control structures like **if**-statements and **for**-loops. In [12], we have shown how the feature set of the RenderMan Shading Language can be implemented with affine arithmetic. In the following, we will briefly review some of these results.

Derivatives of arbitrary expressions are frequently used in procedural shaders. In displacement shaders they can be used to compute the normal vector in a displaced point: the derivatives of the point with respect to the parametric directions u and v

yields two tangent vectors of the surface. The desired normal is the crossproduct of the two. In the RenderMan Shading Language, the built-in function `calculatenormal()` uses this method to generate normals.

The RenderMan standard [20] defines derivatives with the help of divided differences: $\mathbf{Du}(f(u)) := (f(u + du) - f(u))/du$. In this context, du is the sampling rate in u-direction, which is provided to the shader as an implicit parameter (see Table 1). Derivatives with respect to arbitrary expressions are computed using the chain rule: $\mathbf{Deriv}(f, g) := \mathbf{Du}(f)/\mathbf{Du}(g) + \mathbf{Dv}(f)/\mathbf{Dv}(g)$. Using affine arithmetic, we can evaluate these formulas, and thereby obtain conservative bounds for the value of expressions as defined by the RenderMan standard. If bounds for the true mathematical value of derivatives are desired, we can use automatic differentiation [21]. In this case, we store the triple $(\hat{x}, \partial\hat{x}/\partial u, \partial\hat{x}/\partial v)$ for each expression x. The partial derivatives for each operation $f(x)$ is then computed using the chain rule:

$$f\left(\hat{x}, \frac{\partial\hat{x}}{\partial u}, \frac{\partial\hat{x}}{\partial v}\right) = \left(f(\hat{x}), f'(\hat{x})\frac{\partial\hat{x}}{\partial u}, f'(\hat{x})\frac{\partial\hat{x}}{\partial v}\right).$$

Like most programming languages, shading languages also have control statements like `for`-loops and `if-then-else`. These statements are challenging to deal with, because the correct execution path is selected at runtime, based on a number of comparisons (typically inequalities when dealing with floating point values). Unfortunately, two affine forms can have overlapping ranges, in which case comparisons between the two are neither true nor false. It is then necessary to execute both possible execution paths, and merge the resulting ranges into a single affine form.

A simple way of determining whether both paths or only one of them has to be executed, is to replace all inequalities with a step function, for which an affine approximation can be found easily. For example $x < y$ becomes $\mathbf{step}(y - x)$ where $\mathbf{step}(x) := 0$ for $x < 0$ and $\mathbf{step}(x) := 1$ otherwise. Boolean expressions can then be replaced by arithmetic expressions: $x_1 < y_1$ **and** $x_2 < y_2$ becomes $\mathbf{step}(x_1 - y_1) \cdot \mathbf{step}(x_2 - y_2)$.

4.2 Affine Arithmetic and Displacement Shaders

Applied to displacement shaders, affine arithmetic allows us to compute a region $\hat{P} = (\hat{x}, \hat{y}, \hat{z})$ that encloses the displaced surface for a given parameter range $(u \pm du/2, v \pm dv/2)$. The ranges of the components $\hat{x}$, $\hat{y}$, and $\hat{z}$ give us an axis aligned bounding box for the displacement shader.

Note that, in general, the region $\hat{P}$ is smaller than the complete bounding box, because usually the affine forms $\hat{x}$, $\hat{y}$, and $\hat{z}$ will depend on common noise symbols. This means that the values of the components are not independent. More precisely, $\hat{P}$ is a projection of the n-dimensional unit cube:

$$\hat{P} = \begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix} + \begin{pmatrix} x_1 & \cdots & x_n \\ y_1 & \cdots & y_n \\ z_1 & \cdots & z_n \end{pmatrix} \cdot \begin{pmatrix} \epsilon_1 \\ \vdots \\ \epsilon_n \end{pmatrix}$$

This dependency is ignored by evaluating the ranges of the components separately. However, directly dealing with the projection of n-dimensional unit cubes is expensive (there are 2^n vertices), and so the axis aligned bounding box is preferable for intersection tests in ray-tracing applications.

5 Handling Discontinuities

In the form presented in Section 3, the hierarchical subdivision is terminated when the projection of the corresponding bounding box is small enough. This works in continuous areas of the shader, because the region represented by $\hat{P}$ converges to a single point as the parameter range is subdivided. In discontinuous areas, however, the region converges to a higher-dimensional structure, typically a line segment. As a consequence, the axis-aligned bounding box does not, in general, converge to zero around discontinuities of the shader.

Thus, it is necessary to detect discontinuities of the shader during the subdivision process, and to terminate the recursion appropriately. Discontinuities can be caused either by discontinuous library functions, like the **step()** function, or by the use of control statements that cause different execution paths for adjacent points on the surface. Both cases can be dealt with by adding a flag to the implementation of affine forms. The flag is set whenever a library function causes a discontinuity, or the results of two different execution paths are merged into one affine form.

The recursive subdivision is then terminated when the bounding box is small enough OR the surface is discontinuous AND the size of the bounding box did not change significantly during the last subdivision.

Having detected discontinuities, the question is, whether or not intersections with their bounding boxes should be reported as surface intersections. Both alternatives are possible and yield different semantics. If the intersection is not reported, discon-

tinuities will result in holes or disconnected surface parts. If they are reported, this gives the shader author a convenient way of specifying vertical surfaces. The latter approach is consistent with RenderMan, and has been used for rendering Figure 5. The figure shows a simple displacement shader, which, applied to a disk, results in a nail-like surface. Besides the handling of discontinuities, this figure also illustrates that the algorithm is capable of naturally dealing with rather extreme displacements.
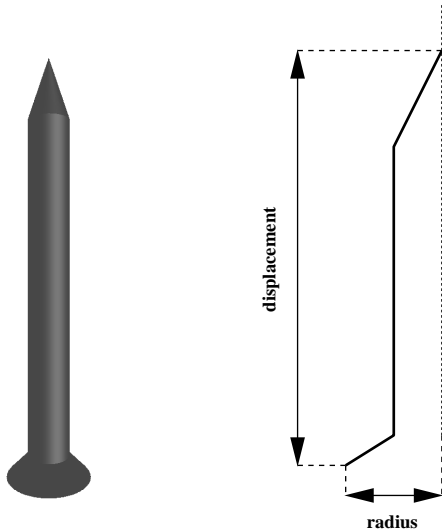

Figure 5: A simple nail shader applied to a small disk at the base of the nail. This illustrates the ability of the algorithm to handle discontinuous and rather extreme displacements.

Note that, if divided differences are used to compute derivatives, the normal vectors for the discontinuous parts automatically converge to the desired result.

6 Caching and Memory Considerations

With the above modifications to the basic algorithm, we are able to ray-trace arbitrary procedural shaders. Unfortunately, the algorithm as presented in Section 3 is relatively slow, since it computes a hierarchy of bounding boxes for each ray. Obviously, there is a lot of coherence between adjacent pixels, and the corresponding rays produce largely identical hierarchies of bounding boxes.

Depending on the complexity of the displacement shader, its evaluation is significantly more expensive than an intersection test of a ray with an axis aligned bounding box. It is therefore desirable to trade memory for computation time by caching some of the bounding boxes already computed for future use.

We do this by storing the bounding boxes in a quadtree. Each node of the tree corresponds to a certain range of the hierarchically subdivided parameter domain. In each node we only need to store the axis aligned bounding box itself (six floating point values), as well as one pointer to each of the four children of the node. The parameter range represented by a quadtree node is implicitly available through the path from the root to the node. All other parameters, which are only required for some of the leaf nodes anyway, can be obtained by evaluating the shader.

This results in a fairly storage efficient data structure. Each node requires 40 to 80 Bytes, depending on whether we use single or double precision floats and 32 or 64 bit pointers. The total size of the tree can still be extremely large, especially for large image resolutions, which require more levels of subdivision. Thus, it is necessary to remove parts of the tree that are not likely to be required again soon.

In our implementation of the ray-tracer, which uses a normal scanline traversal order for the pixels, we allocate a certain amount of memory for the nodes in the tree. If the memory requirements exceed this limit, we traverse the tree, deleting every node except the ones used by the previous ray.

If we allow arbitrary pixel traversal orders it is easy to think of other methods for limiting memory usage. For example, the screen could be subdivided into rectangular tiles, each of which is rendered separately, or a space filling curve could be used as in [18]. An even more promising approach would be the use of a ray cache as in [19]. However, these alternatives have not been explored yet.

Another improvement of the algorithm, both in performance and in memory requirements, can be achieved by adapting the traversal order of the subtrees. In ray-tracing, often only the first intersection of a ray with an object is required. The algorithm from Section 3 finds all intersections, and therefore
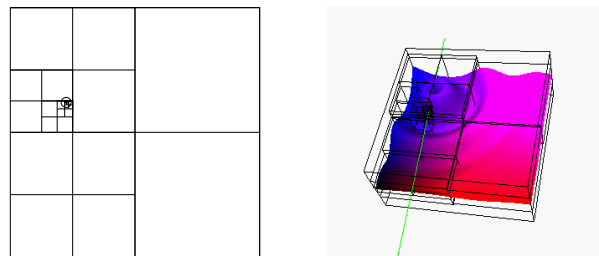

Figure 6: With the modified traversal order, the algorithm only finds the closest intersection. This results in a reduced number of shader evaluations.

has to refine the bounding box hierarchy in areas which are not relevant to finding the closest intersection point.

In an improved version of the algorithm, the branches whose bounding boxes are closest to the origin of the ray are refined first. Branches for which the bounding boxes are further away than already known intersections do not need to be traversed. Figure 6 shows how this method reduces the number of bounding boxes compared to Figure 3.

7 Results

We have timed our method with several shaders and cache sizes. As shaders we used the wave and nail shaders from above, as well as the "UFO"-shader depicted in Figure 7. The cache sizes were 40, 400 and 4000 KBytes, corresponding to 1000, 10000, and 100000 quadtree nodes with a size of 40 Bytes. The resulting timings are listed in Table 2. The resolution for the wave shader was 512×256, for the nail shader 256×512, and for the UFO shader 256×256. For the UFO shader, we always computed all intersections, whereas for the other two, we only computed the closest one.
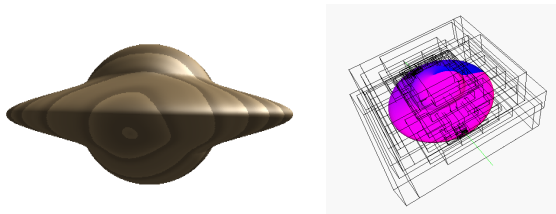

Figure 7: The "UFO" displacement shader applied to a sphere. The surface shader is the RenderMan wood shader.

Shader	40KB	400KB	4MB	unlimited
Wave	122.3	71.1	66.7	66.2
Nail	380.7	33.2	32.6	32.6
Ufo	121.4	89.5	82.5	82.2

Table 2: Timings (in seconds) for several shaders and cache sizes on a SGI O2 with 175MHz R10000 processor.

The numbers show that, even for our simple caching scheme and the moderate cache size of 400 KB, the performance penalty is below 10 percent compared to unlimited cache space. As with any caching scheme, the performance drops dramatically for small cache sizes, that cannot hold the working set of the application. In our case this corresponds to the number of nodes required for a complete scanline.

The loss of performance is particularly dramatic for the nail shader, in which the bounding boxes for the discontinuous portion can be reused over a large area of the image. If the cache is too small, these boxes have to be recomputed over and over again. A more sophisticated caching scheme would certainly help in this case.

7.1 Affine Arithmetic and Interval Arithmetic

In this paper we use affine arithmetic to obtain conservative bounds for shader values over a parameter range. In principle, we could also use any other range analysis method for this purpose. It is, however, important that the method generates tight, conservative bounds, because this reduces the number of bounding boxes to be generated, and therefore improves both performance and memory requirements.

In this sense, interval arithmetic performs worse than affine arithmetic for this application. It produces much wider bounds than affine arithmetic when applied to procedural shaders. Figure 8 shows a comparison of the subdivisions generated for the UFO shader by a single ray. For interval arithmetic, 1450 bounding boxes had to be generated, for affine arithmetic this number was only 174. The more sophisticated shaders are used, the worse interval arithmetic performs. These measurements are consistent with the results presented in [5], [9], and [8], as well as our own results for surface shaders [12].
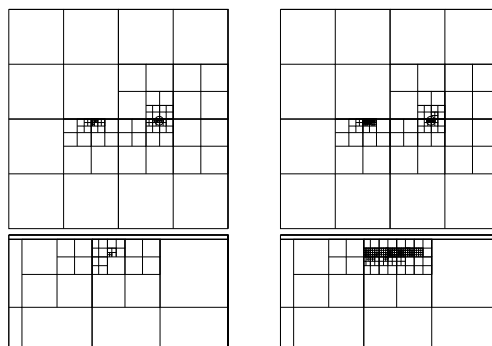

Figure 8: A comparison of affine arithmetic (left) and interval arithmetic (right). Affine arithmetic produced 174 bounding boxes, interval arithmetic 1450. A magnified portion is showed on the bottom.

8 Conclusions and Future Research

In this paper we have presented a new iterative method for computing intersections of rays with procedural displacement shaders. The method directly operates on the procedural description of the shader,

without introducing any approximate geometry. As a consequence, the method is resolution independent and storage efficient, and is guaranteed to find every intersection of the ray with the surface. Computation time can be traded for memory in order to improve performance.

We already mentioned that more sophisticated caching schemes like the one in [19] are likely to further improve performance. This is something to be explored in the future.

Another interesting area of research is the parallelization of the this method for loosely coupled distributed systems and rendering farms. The procedural shaders that have to be distributed across the network are extremely small. From this data, every host is able to compute its own cache for bounding boxes, instead of having to share large amounts of tessellated geometry with other hosts.

Last but not least, the design of a renderer that is completely based on range analysis methods such as affine arithmetic is a promising area of future research. The combination of work mentioned in Section 2, together with the work presented in this paper, allows for guaranteed intersection tests with almost any kind of commonly used geometric description.

References

[1] Alias/Wavefront. *OpenAlias Manual*, 1996.

[2] Alan H. Barr. Global and local deformations of solid primitives. In *Computer Graphics (SIGGRAPH '84 Proceedings)*, pages 21–30, August 1984.

[3] Alan H. Barr. Ray tracing deformed surfaces. In *Computer Graphics (SIGGRAPH '86 Proceedings)*, pages 287–296, August 1986.

[4] James F. Blinn. Simulation of wrinkled surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, pages 286–292, August 1978.

[5] João L. D. Comba and Jorge Stolfi. Affine arithmetic and its applications to computer graphics. In *Anais do VII Sibgrapi*, pages 9–18, 1993. Available from http://www.dcc.unicamp.br/~stolfi.

[6] Tom Duff. Interval arithmetic and recursive subdivision for implicit functions and constructive solid geometry. In *Computer Graphics (SIGGRAPH '92 Proceedings)*, pages 131–138, July 1992.

[7] David Ebert, Kent Musgrave, Darwyn Peachey, Ken Perlin, and Steve Worley. *Texturing and Modeling: A Procedural Approach*. Academic Press, October 1994.

[8] Luiz Henrique Figueiredo. Surface intersection using affine arithmetic. In *Graphics Interface '96*, pages 168–175, 1996.

[9] Luiz Henrique Figueiredo and Jorge Stolfi. Adaptive enumeration of implicit surfaces with affine arithmetic. *Computer Graphics Forum*, 15(5):287–296, 1996.

[10] Pat Hanrahan and Jim Lawson. A language for shading and lighting calculations. In *Computer Graphics (SIGGRAPH '90 Proceedings)*, pages 289–298, August 1990.

[11] Wolfgang Heidrich. A compilation of affine approximations for math library functions. Technical Report TR-1997-3, University of Erlangen Computer Graphics Group, 1997.

[12] Wolfgang Heidrich, Philipp Slusallek, and Hans-Peter Seidel. Sampling procedural shaders using affine arithmetic. *ACM Transactions on Graphics*, 1998.

[13] Devendra Kalra and Alan H. Barr. Guaranteed ray intersections with implicit surfaces. In *Computer Graphics (SIGGRAPH '89 Proceedings)*, pages 297–306, July 1989.

[14] Ramon E. Moore. *Interval Analysis*. Prentice Hall, Englewood Cliffs, New Jersey, 1966.

[15] F. Kenton Musgrave, Craig E. Kolb, and Robert S. Mace. The synthesis and rendering of eroded fractal terrains. In *Computer Graphics (SIGGRAPH '89 Proceedings)*, pages 41–50, July 1989.

[16] Tomoyuki Nishita, Thomas W. Sederberg, and Masanori Kakimoto. Ray tracing trimmed rational surface patches. In *Computer Graphics (SIGGRAPH '90 Proceedings)*, pages 337–345, August 1990.

[17] Ken Perlin. An image synthesizer. In *Computer Graphics (SIGGRAPH '85 Proceedings)*, pages 287–296, July 1985.

[18] Matt Pharr and Pat Hanrahan. Geometry caching for ray-tracing displacement maps. In *Eurographics Rendering Workshop 1996*, pages 31–40, June 1996.

[19] Matt Pharr, Craig Kolb, Reid Gerhbein, and Pat Hanrahan. Rendering complex scenes with memory-coherent ray tracing. In *Computer Graphics (SIGGRAPH '97 Proceedings)*, pages 101–108, aug 1997.

[20] Pixar. *The RenderMan Interface*. Pixar, San Rafael, CA, Sep 1989.

[21] Louis B. Rall. *Automatic Differentiation, Techniques and Applications*. Number 120 in Lecture notes in computer science. Springer, 1981.

[22] John M. Snyder. *Generative Modeling for Computer graphics and CAD: Symbolic Shape Design Using Interval Analysis*. Academic Press, 1992.

[23] John M. Snyder. Interval analysis for computer graphics. In *Computer Graphics (SIGGRAPH '92 Proceedings)*, pages 121–130, July 1992.

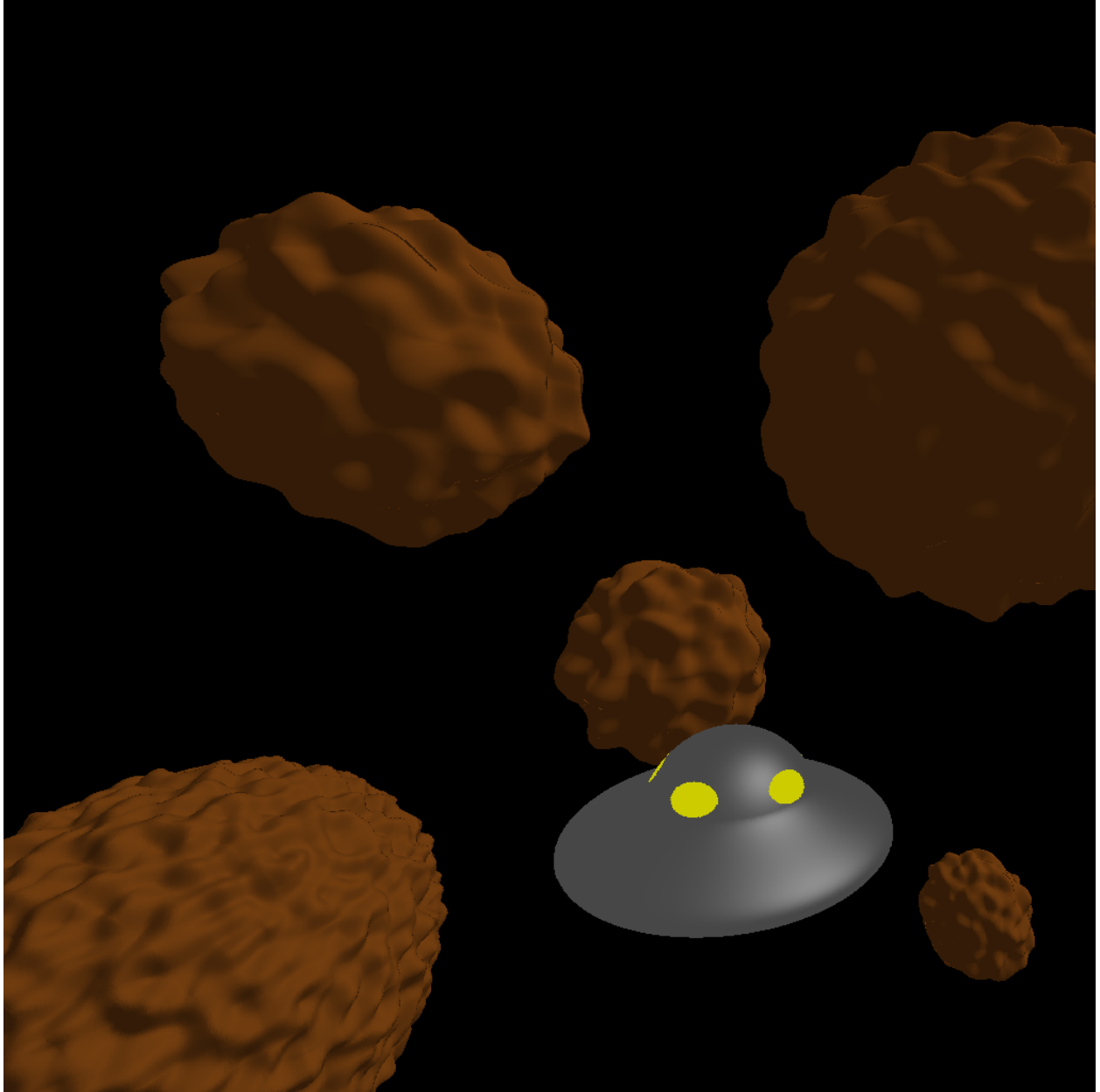[24] Steve Upstill. *The RenderMan Companion*. Addison Wesley, 1990.

Figure 9: A more complex scene with multiple objects. The underlying geometry of each object is a sphere.