Seventh Hungarian Conference on Computer Graphics and Geometry, Budapest, 2014

A Geometry Model for Logarithmic-time Rendering

László Szécsi

Budapest University of Technology and Economics, Budapest, Hungary

Abstract

Complex geometries, like those of plants, rocks, terrain or even clouds are challenging to model in a way that allows for real-time rendering but does not make concessions in terms of visible detail. In this paper we propose a modeling approach called KRS, or kernel-reflection sequences*, inspired by iterated function systems. KRS visualization avoids expanding the procedural definition into polygons all along the rendering pipelines. The model is composed of kernel geometries and reflection transformations that multiply them. We show that a distance function can be evaluated over this structure extremely effectively, allowing for the implementation of real-time sphere tracing in pixel shaders. We also show how the algorithm easily delivers continuous level-of-detail and minification filtering. We discuss how exploiting screen-space coherence can enhance ray-casting performance. We propose several techniques to hide symmetry that could be disturbingly obvious when viewing the models from certain angles. In order to prove that the seemingly limited model can be used to realize various natural phenomena in uncompromising detail without obvious clues of symmetry, we render trees, grass, terrain and rock in real-time.*

Categories and Subject Descriptors (according to ACM CCS): Computer Graphics [I.3.7]: Three-Dimensional Graphics and Realism Fractals

1. Introduction

Geometries occurring in nature typically feature intricate detail and great extents at the same time. They pose challenges throughout both the modeling and the rendering pipeline, from content creation to final visualization. The usual approach of modeling triangle meshes and rendering them through incremental rasterization fails at both stages. First, every leaf in a forest cannot be modeled manually. It has to be scripted, thus already requiring a procedural description of the geometry, and a way of expanding that description to a visualizable representation. Second, the resulting geometry can be too complex to be rendered in real-time even with the immense triangle throughput of the latest graphics hardware. That gives emergence to convoluted level of detail techniques, which all need to address the issue of transition between different triangle mesh representations.

The later in the pipeline the procedural description is expanded, the less stages can prove to be a bottleneck. CPU expansion would require large amounts of geometry to be communicated to the graphics hardware. With geometry shaders, this is no longer necessary, which has propelled the use of procedural geometries in real-time applications to considerable momentum[6]. However, a large number of triangles would still need to be rasterized, and level-of-detail schemes are still necessary.

In this paper, we propose a formally limited, but practically versatile procedural modeling and visualization approach that defers the expansion of the procedural geometry to the final, pixel shader stage. In other words, we show that ray tracing of the procedural geometry can be done in real-time, at a computational complexity superior to rasterization, and thus vastly outperforming incremental rasterization for sufficiently complex geometries.

2. Previous work

IFS, or *iterated function systems* were conceived by Hutchinson[5]. An IFS describes self-similar geometry by specifying a set of functions mapping the self-similar components to itself[10]. The geometry is the attractor of the iteration of the function system, meaning the geometry can be approximated with arbitrarily small error by iterating any initial bound point set. IFSs are capable of generating geometries of fractal dimensions, occasionally resembling natural phenomena, but usually with an easily discernable pattern.

CSG, or *constructive solid geometry* defines geometries as results of regular set operations on point sets, which are either specified in the same manner, or are primitives. These primitives are usually given as implicit surfaces. Self-similar, natural geometries can be generated by introducing circles in the construction graph, leading to cyclic object-instantiation graphs, or CSG-PL-Systems[2]. Efficient ray tracing can be performed by generating bounding objects for self-similar components[12].

F-rep[7] defines objects as sets of points for which a function is non-negative. It supports set operations and recursion.

Sphere tracing was proposed by Hart[3]. It is an iterative technique for ray intersection against a geometry for which a distance function is known. This distance function must return a tight underestimation for the distance of a point and the ray traced geometry, or, in other words, the radius of an *unbounding* sphere [4] centered at the point. The algorithm progresses by advancing a point along the ray with this distance, to the surface of the unbounding sphere.

Procedural models can be expanded to triangle meshes in geometry shaders. Algorithms for L-systems[8] and split grammars have been discussed by Sowers[9].

3. Kernel-reflection sequences

Let us start with self-similar geometry defined by an IFS. For the attractor point set A it is true that:

$$A = \bigcup_{j=0}^{m-1} \mathfrak{F}_j A,$$

where $\mathfrak{F}_j$ is a contractive, invertible linear transformation for any j, and the number of component transformations is m. This attractor can be obtained as a limit of the sequence:

$$A_{i+1} = \bigcup_{j=0}^{m-1} \mathfrak{F}_j A_i,$$

$$A_0 = K_0,$$

where K_0 is an arbitrary point set, which we will call a *kernel set*. At this point, because of the contractiveness of the operators, the limit is independent of the choice of K_0. Later, we will abandon contractiveness and this will not be true. In practice, A is approximated as A_n with a suitably large n. For all equations below, $i \in \{0,\dots,n-1\}$ must be true unless otherwise noted. As we wish to use sphere tracing, the kernel set will be defined by distance function $k_0(\mathbf{x})$, which gives the geometric distance between point $\mathbf{x}$ and K_0.

First, let us generalize this construction by allowing different linear transformations on different iterations. $\mathfrak{F}_{j,i}$ denotes the jth transformation operator for iteration i. Then, without the loss of generality, we can assume that $m = 2$, as the polyadic union can always be expressed as a composition of dyadic unions. Thus, the recursive formula for A_i becomes

$$A_{i+1} = \mathfrak{F}_{0,i} A_i \cup \mathfrak{F}_{1,i} A_i.$$

For sphere tracing, an underestimation $a_i(\mathbf{x})$ for the distance between point $\mathbf{x}$ and A_i is required. As described by Hart[3], this can be obtained if the Lipschitz constants of the transformations (denoted by $\mathrm{Lip}\mathfrak{F}$) are known.

$$a_{i+1}(\mathbf{x}) \leq \min\left(a_i(\mathfrak{F}_{0,i}^{-1}\mathbf{x})\mathrm{Lip}\mathfrak{F}_{0,i}^{-1}, a_i(\mathfrak{F}_{1,i}^{-1}\mathbf{x})\mathrm{Lip}\mathfrak{F}_{1,i}^{-1} \right),$$

$$a_0(\mathbf{x}) = k_0(\mathbf{x}).$$

The recursive computation of this formula for $a_n(\mathbf{x})$ requires 2^n evaluations of $k_0(\mathbf{x})$, which means that the performance would be exponential in n, and linear in the number of generated kernel instances. Thus, the algorithmic complexity of sphere tracing could be at best identical to incremental rendering of the same geometry, but with a much worse constant factor. It is also notable that ray tracing in general has logarithmic average complexity[11], but it is linear in worst case, and recursively traversing subdivision hierarchies makes it ill-fit for GPU processing.

If bounding hulls are known, then bounds for $a_i(\mathfrak{F}_{0,i}^{-1}\mathbf{x})$ and $a_i(\mathfrak{F}_{1,i}^{-1}\mathbf{x})$ can be obtained, and the performance can be significantly improved by prioritization and lazy evaluation of recursion branches. However, we aim for complete, unconditional elimination of the recursion in order to get an algorithm that has linear complexity in n, and thus logarithmic complexity in the number of kernel instances. Such an algorithm can be implemented as an iteration, effectively executable on graphics processors. To those ends, we drastically limit the transformations we can use. Let $\mathfrak{F}_{0,i}$ always be identity and $\mathfrak{F}_{1,i} \equiv \mathfrak{R}_i$, a reflection on the plane of equation $\mathbf{m}_i \cdot \mathbf{x} - c_i = 0$. We always choose $\mathbf{m}_i$ to have unit length, thus the value of $\mathbf{m}_i \cdot \mathbf{x} - c_i$ also gives the signed distance of point $\mathbf{x}$ to the plane. The operator $\mathfrak{R}_i$ means reflection on this plane.

$$\mathfrak{R}_i \mathbf{x} = \mathbf{x} - 2(\mathbf{m}_i \cdot \mathbf{x} - c_i)\mathbf{m}_i.$$

The $\mathfrak{M}_i$ operator denotes the reflection of a direction vector:

$$\mathfrak{M}_i \omega = \omega - 2(\mathbf{m}_i \cdot \omega)\mathbf{m}_i.$$

Both the indentity and reflection transformations are isometries, thus their Lipschitz constants are unity. Identity and reflection are not contractions, and the sequence is divergent. The kernel geometries are preserved at their original size and detail. Increasing n will increase the extents of the set. Now, the sequence of attractor iterations is as simple as:

$$A_{i+1} = A_i \cup \mathfrak{R}_i A_i,$$

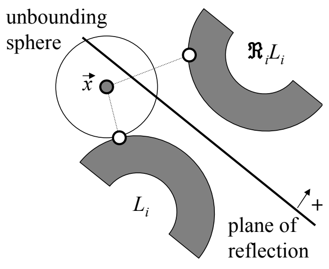

Figure 1: *The distance of the closest point and its reflected image.*

$$A_0 = K_0.$$

We further assume that

$$\mathbf{m}_i \cdot \mathbf{x} - c_i > 0 \longrightarrow \mathbf{x} \notin A_i,$$

$$\mathbf{m}_i \cdot \mathbf{x} - c_i \leq 0 \longrightarrow \mathbf{x} \notin \mathfrak{R}A_i.$$

This means that the geometry is composed of two disjoint parts on the two sides of the mirror plane, which are reflected images of each other (see Figure 1). We call this the assumption of *kernel separation*. A simple test can tell which part is at less distance to point $\mathbf{x}$. We just need to determine if the point is behind or in front of the mirror plane, that is, whether $\mathbf{m}_i \cdot \mathbf{x} - c_i \leq 0$. The closest point of the geometry must be on the same side. (If it were not, the reflected image of the closest point would be even closer, resulting in contradiction.)

This construction allows only for reflection-multiplied instances of the same, unscaled kernel geometry. While this might be enough to model some natural phenomena, hierarchies (like branches of a tree) and non-symmetric parts are not covered. In order to remedy this, let us add a new kernel set K_i at each iteration. These are defined by a sequence of possibly all different $k_i(\mathbf{x})$ distance functions, where $i \in \{0, \ldots, n\}$. To emphasize the difference in construction, we replace the notation A_i with L_i, and call L_i an *expansion level*. These also form a finite sequence, where L_{i+1} is recursively defined as:

$$L_{i+1} = K_{i+1} \cup L_i \cup \mathfrak{R}_i L_i,$$

$$L_0 = K_0.$$

Thus, an expansion level consists of two symmetric instances of the previous level, and an additional kernel set.

We call such a construct a KRS, or *kernel-reflection sequence*. Formally, it is an ordered pair of two finite sequences, one consisting of kernel sets, and another of reflection operators.

$$\text{KRS} = (K_0, \ldots, K_n; \mathfrak{R}_0, \ldots, \mathfrak{R}_{n-1})$$

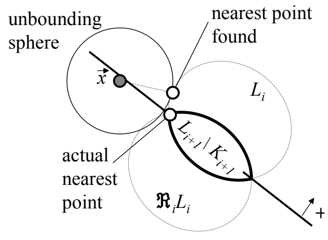

Figure 2: *Clipped kernel part might influence the distance estimate.*

The distance function for a KRS is:

$$l_{i+1}(\mathbf{x}) = \min(k_{i+1}(\mathbf{x}), l_i(\mathbf{x}), l_i(\mathfrak{R}_i\mathbf{x})),$$

$$l_0(\mathbf{x}) = k_0(\mathbf{x}).$$

In order to evaluate the formula, we do not have to compute all the terms. As L_i and $\mathfrak{R}_i L_i$ are known to be mirrored images, we can decide which distance is going to be smaller by finding on which side of the mirror plane $\mathbf{x}$ is.

$$l_{i+1}(\mathbf{x}) = \begin{cases} \min(k_{i+1}(\mathbf{x}), l_i(\mathbf{x})) & \text{if } \mathbf{m}_i \cdot \mathbf{x} - c_i \leq 0 \\ \min(k_{i+1}(\mathbf{x}), l_i(\mathfrak{R}_i\mathbf{x})) & \text{if } \mathbf{m}_i \cdot \mathbf{x} - c_i > 0 \end{cases}$$

Note that if the assumption of kernel separation does not hold, with this decision we implicitly enforce it by clipping the distance functions to the mirror plane. Therefore, we do not need to take care of this assumption when picking kernels or reflection planes. The returned value might be smaller than the actual distance. In Figure 2, $\mathbf{x}$ is on the negative side, so L_i is closer. The clipped part of the kernel is also considered, making the underestimation somewhat less tight, but still a conservative choice for sphere tracing. The iterative algorithm to evaluate the distance is given in Algorithm 1.

Algorithm 1 Returns distance between $\mathbf{x}$ and the KRS.

```
 1: function DISTANCE($\mathbf{x}$)
 2:     $\mathbf{p} \leftarrow \mathbf{x}$
 3:     $d \leftarrow k_n(\mathbf{p})$
 4:     for $i = n-1$ downto 0 do
 5:         if $\mathbf{p} \cdot \mathbf{m}_i - c_i > 0$ then
 6:             $\mathbf{p} \leftarrow \mathfrak{R}_i\mathbf{p}$
 7:         end if
 8:         $d \leftarrow \min(d, k_i(p))$
 9:     end for
10:     return $d$
11: end function
```

4. Ray-casting

There are multiple options for the GPU visualization of a KRS, practically any IFS visualization method could be generalized. Most prominently, kernel instance transformations can be computed in geometry shaders, and then kernel geometries rendered with geometry instancing. However, the main motivation behind the construction of KRSs is that they can be ray-traced with an iterative algorithm. Thus, in this paper, we focus on visualization with ray-casting. A full-viewport quadrilateral is rendered, and pixel shaders find the intersection points using sphere tracing. The search is terminated when the ray has passed through the scene or when the computed distance falls below an error treshold level. The value is inversely proportional to the camera depth, and is set to assure that the final unbounding sphere, projected onto the viewport, is smaller than a pixel. A higher treshold level will, in practice, make the kernel geometries appear thicker, as points in close proximity are considered to be members. Therefore, geometries at a large distance will merge into smoother formations, loosing sub-pixel details. This, combined with the smooth shading and texturing techniques we describe in Sections 6.1 and 6.2, eliminates aliasing and achieves automatic, continuous level-of-detail.

4.1. Acceleration with unbounding spheres

The sphere tracing process can be accelerated if we exploit the screen-space coherence of the ray-casting problem. Shaders processing neighboring pixels will execute very similar steps, at least in the beginning. These can be avoided, if, in a cheap preprocessing step, we can find tighter *free distances* to start sphere tracing from. Various algorithms could be based on the idea that when an unbounding sphere is found, the information might be useful for more than one pixel. Unbounding spheres may be rasterized with depth buffering suitably set up, or stored in a searchable data structure and queried from final ray-casting pixel shaders. We conjecture that any such method will produce a starting distance field of practically similar quality, when the cost compared to the full ray-casting itself has to be negligible. We base this claim on the intuitive recognition that a *free distance* map coarser than the viewport resolution can only be useful in front of the first layer of depth. For those expensive, and not uncommon rays that have passed by the geometry closely, the map can give no more clues.

We implemented a scheme that divides the viewport into tiles, and traces beams of primary rays that pass through a tile. Sphere tracing progresses along the ray through the center of the tile up to the last unbounding sphere that covers the complete solid angle of the beam (Figure 3). Then, this sphere and a few more are stored for the tile, and used when ray-casting in pixels of the tile. Care is taken to ensure that the union of all stored spheres, intersected by the beam is convex as seen from the eye. Figure 4 depicts this process. At the ray exit point on the last stored unbounding sphere E, a new, tentative unbounding sphere T is generated. The next stored sphere F must touch the intersection of the two unbounding sphere shells, and its tangents there must go through the eye. This acceleration scheme resulted in about 10-30% less rendering time.

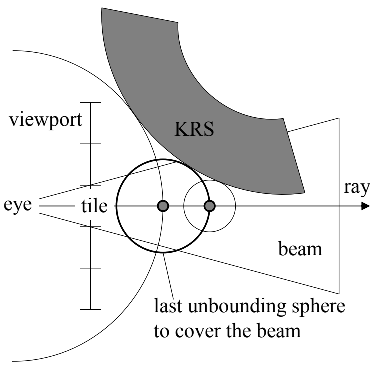

Figure 3: *Traversing a ray for a tile, up until the last sphere that covers the beam.*

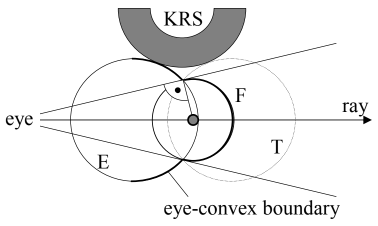

Figure 4: *Finding an unbounding sphere that makes a convex profile. (Note that the center of T has been placed further away than the free distance for a more readable figure.)*

4.2. Acceleration with procedural bounding geometry

Free distances for sphere tracing can also be found by rendering bounding objects. Here, we can easily avoid the exponential explosion, as a choice of level-of-detail is not critical. Abrupt changes in the bounding geometry will not influence the correctness of sphere tracing, and there are no popping artifacts.

5. Techniques to hide symmetry

There are two features of a KRS that strain its credibility as a natural occurrence. Symmetry might be visible and identical motifs are repeating. One countermeasure is that non-symmetric kernel elements are added on every iteration. More importantly, planes of reflection should be selected so that the eye can only be near to a few of them at the same time. If the eye is not near to the mirror plane, the symmetry of the 3D object will not be perceived on a 2D image. Thus, it cannot happen that our geometry appears like an obviously artificial fractal pattern from a viewpont. The undesirable symmetry effects on the global scale are eliminated. Figure 5 offers a comparison.

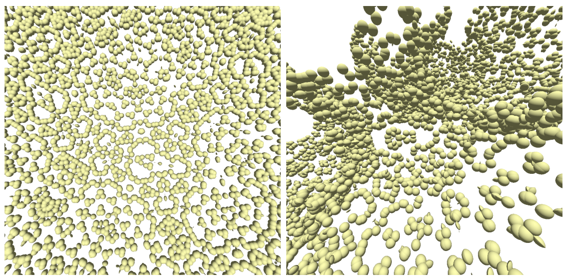

Figure 5: *Spheres reflected by aligned mirrors and unaligned mirrors.*

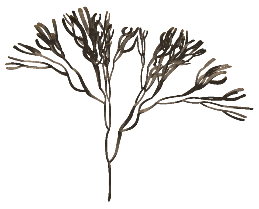

Figure 6: *A tree with isometric transformations and with Lipschitz distortion.*

Fighting symmetry on the local scale is more challenging. There will always be a viewpoint from where two subsets are visibly the reflected images of each other. This can only be handled if the symmetry is indeed broken.

5.1. Combination of multiple KRSs

Where a single KRS cannot produce the desired effect of natural disorder, the union of multiple KRSs can. When, in a forest, there are three completely different trees between the two that are mirror images of each other, symmetry is undetected. Sphere tracing multiple KRSs can be effectively implemented by maintaining the free distance along the ray for all components, and always advancing the ray to the minimum. Compared to the single KRS case, the performance is only decreased where different silhouettes overlap.

Procedural or projective texturing can also be considered to be a way of combining features that exhibit periodicity at different frequencies. We will detail techniques in Section 6.2.

5.2. Distance distortion

The Lipschitz constant of KRS transformation functions is unity, resulting in exact values for the distance (save for non-separated kernels). By sacrificing some performance, we can handle any Lipschitz transformation of the KRS geometry as described by Hart [3]. The resulting geometry does not have to be symmetric any more (Figure 6).

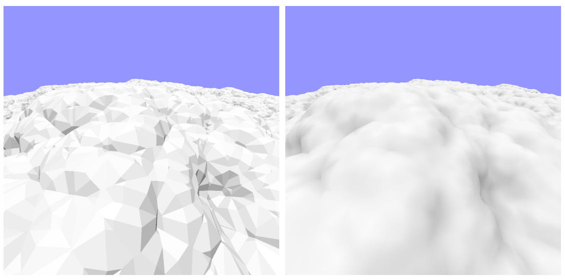

Figure 7: *Rock composed of planar kernels with flat and smooth shading.*

6. Combination with other real-time techniques

KRS ray-casting can only be a viable alternative to geometry-shader produced geometries if it supports all the incremental image synthesis techniques contributing to realism.

6.1. Local shading

KRS kernels are solids with well defined surface normals. The only difficulty is their transformation from kernel space to world space. If b_i is one if $\mathbf{p}\cdot\mathbf{m}_i - c_i > 0$ and zero otherwise, then the complete transformation of the kernel is:

$$\mathbf{p}_{\text{world}} = \mathfrak{R}_{n-1}^{b_{n-1}} \circ \mathfrak{R}_{n-2}^{b_{n-2}} \circ \cdots \circ \mathfrak{R}_0^{b_0} \cdot \mathbf{p}_{\text{kernel}}.$$

The normal ν must be transformed with the inverse transpose of the transformation matrix. The inverse of a reflection is itself, transposition and inversion both turn the order of matrix multiplication.

$$\nu_{\text{world}} = \mathfrak{M}_{n-1}^{b_{n-1}} \circ \mathfrak{M}_{n-2}^{b_{n-2}} \circ \cdots \circ \mathfrak{M}_0^{b_0} \cdot \nu_{\text{kernel}}.$$

Unfortunately, the evaluation of this formula requires us either to record decision variables b_i during Algorithm 1, or to maintain a product transformation as the iteration proceeds in decreasing order of i. The latter solution is desirable, as it scales better with increasing n. However, it is even more efficient to transform the world space light vector (and view vector, if necessary) into kernel space, and evaluate the shading there.

When kernels and mirrors are not selected so that kernel separation is upheld, there is likely to be a visible discontinuity of surface normals where the plane of reflection intersects the mirrored geometry. These can be smoothed by interpolating between normals or light directions near these planes. This technique allows the generation of smooth surfaces from a kernel as simple as an infinite plane. Figure 7 compares flat and smooth shading of a surface composed of planar kernels. The shading algorithm complete with light direction interpolation is listed as Algorithm 2. The *lerp* function performs linear interpolation between its first two arguments weighted by the third argument clamped to unit range.

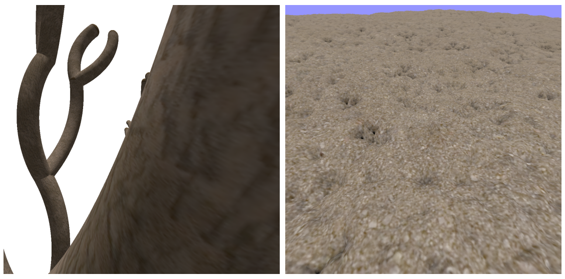

Figure 8: *Texturing with kernel parametrization on tree branches and triplanar projection on terrain.*

The e_{treshold} distance influences the amount of smoothing. To eliminate aliasing artifacts, it should be inversely proportional to the camera depth. The shading algorithm runs only once, after sphere tracing has found the intersection point.

Algorithm 2 Returns the diffuse shaded color of KRS at point $\mathbf{x}$ with surface normal ν illuminated by light from direction τ. $\mathbf{r}$ is the incoming radiance and $\mathbf{d}$ is the diffuse BRDF coefficient.

```
1: function SHADE($\mathbf{x},\nu,\tau,\mathbf{r},\mathbf{d}$)
2:     $\mathbf{p} \leftarrow \mathbf{x}$
3:     $\omega \leftarrow \tau$
4:     $d \leftarrow k_n(\mathbf{p})$
5:     for $i = n-1$ downto 0 do
6:         $e \leftarrow \mathbf{p}\cdot\mathbf{m}_i - c_i$        ▷ signed distance to plane
7:         if $e > 0$ then
8:             $\mathbf{p} \leftarrow \mathfrak{R}_i\mathbf{p}$
9:         end if
10:        $\rho \leftarrow \mathfrak{M}_i\omega$
11:        $\omega \leftarrow \text{lerp}(\omega,\rho,e/e_{\text{treshold}}+0.5)$
12:        $d \leftarrow \min(d,k_i(p))$
13:    end for
14:    return $d$
15: end function
```

6.2. Texturing

The kernel solids are usually simple objects (e.g. torus segments) that lend themselves to easy u,v parametrization. There are two cases when this solution is not feasible. First, if we use kernels where such a parametrization is not trivial. Second, if the kernels are simplistic, like infinite planes, that even the smallest details of geometry are determined by the reflection transformations. In this second case, texturing all kernel instances with the same coordinates would produce an extremely repetitive pattern, emphasizing symmetries undesirably. Procedural 3D or triplanar[1] texturing is applicable with convincing results (Figure 8). Procedural geometry and procedural or wrapped textures combine to eliminate the observable repetitiveness of each other.

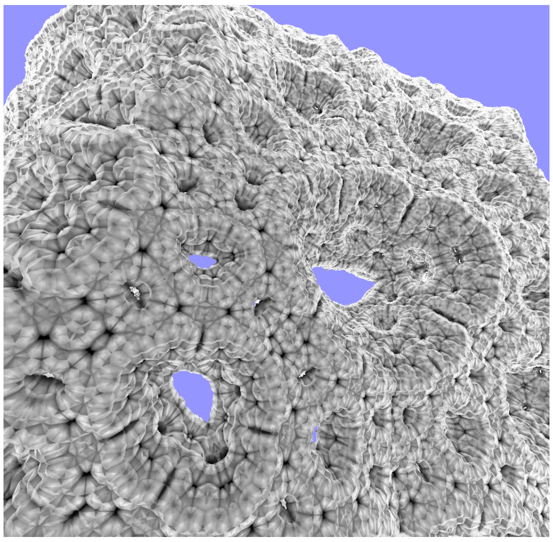

Figure 9: *Sponge with ambient occlusion.*

6.3. Depth composition

A ray-cast KRS can easily be integrated into scenes rendered incrementally. The depth buffer can be used for early termination of rays. The ray casting shader can also output depth if it is necessary, e.g. if transparent geometry is to be rendered afterwards, or if shading is deferred.

6.4. Collision and destructibility

Based on the distance function and surface normal computation, a KRS can be integrated into any collision detection and response scheme.

A KRS is not locally controllable. However, it is possible to create *empty zones* by specifying unbounding spheres over subsets. The regions of these spheres should be skipped during sphere tracing. The subset of the KRS within these empty zones can be substituted with instances of kernel geometry rendered incrementally. These solid instances can then be subjects of any kind of physical simulation. E.g. when a branch of a tree in a forest should break, the complete tree is covered with an empty sphere, and an identical tree of rigid body branches and leaves is built. This can then be manipulated independently.

6.5. Global shading

Beyond sheer triangle throughput, KRS geometry has another key advantage over incrementally rendered geometry. Sphere tracing can not only be performed for eye rays, but also for secondary rays. Shadows, reflections, ambient occlusion (Figure 9), or any global illumination techniques based on ray tracing can be implemented.

7. Modeling

KRSs appear to be limited at what can be modeled with them. At an iteration count and scale large enough for the individual kernels not to be distinguishable, the geometry tends to resemble the 3D equivalent of a Lévy C-curve. However, a forest from the air, a cloud in the sky, a hilly landscape, or a battered rock look exactly like that, from far enough. In this section we are going to present a few examples of application.

7.1. Coral, terrain and rock

Those natural features that are traditionally well modeled by IFS are good candidates for KRS. The variation in scale that is lost because we only use isometries is compensated by additional kernels and any statistical self-similarity induced by the choice of our transformations. When modeling these geometries, the choice of kernel sets is also less important: spheres or infinite planes are sufficient. The separation of kernels is neither desirable nor always possible. Even the smallest details will be defined by the transformations. Smooth shading and procedural or triplanar texturing greatly enhance the visual quality.

7.2. Tree and forest

A tree is a classic hierarchical structure, where the kernel sets added at each iteration become crucial. The first few kernels and reflections define the leaf geometry and the thinnest twig. Then, every new expansion level doubles this geometry, with a reflection placed so that the two main branches start from the same point. A new, thicker branch that ends at the junction is added as the next kernel. For kernels acting as branches, we used *toroidal capsules*, composed of a torus segment and two capping spheres. The coefficients of the distance functions are computed from intuitive modeling parameters. First, forking positions along one route from the trunk to a twig end must be given. Branches along this route will be the kernels. Then, a control point on every such branch can be moved to set curvature. Two forking positions and a control point define the generating circle of the torus. Branch width gives the section radius. Planes of reflection must be placed at forking positions, with editable normals. A forest can be generated by adding more reflections (identical to those of the terrain, if it is also present) with empty extra kernel sets.

8. Results

There were two distinct types of test scenes: ones with complex kernels (trees, grass) and ones with simples kernels (terrain, rock). In the following table we give how much time it took (in microseconds) to trace a single ray on average, versus the triangle count equivalent of the scene complexity (how many triangles would be necessary to achieve the same result.)

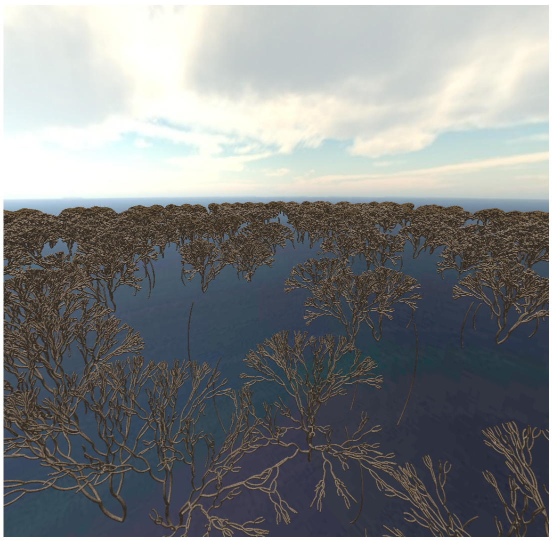

Figure 10: *A groove of trees.*

Triangle count	Complex kernel (μs)	Simple case (μs)
2^6	21	
2^{12}	37	
2^{18}	50	47
2^{20}	54	93
2^{22}	58	146
2^{24}	61	191
2^{26}	64	250
2^{28}	65	300
2^{30}	68	355
2^{32}	71	397

Increasing the expansion level by one resulted in an average increase of rendering time of 25 microseconds for complex kernels and 3 microseconds for simple ones. We can conclude that it is possible to achieve interactive speeds on scenes equivalent to billions of triangles.

9. Conclusion

KRS rendering can be used together with incremental image synthesis in real-time applications. It is capable of modeling a wide range of natural geometries, without the possibility of local control, but with fine details and large extent at the same time. Features that could only be represented by billions of triangles, like grasslands or forests, can be rendered in real time. Thus, KRS can add the desired natural richness of detail to virtual worlds without the need for customized level-of-detail techniques.

Animation of KRS geometries is left for future work. While animation of the model parameters is unlikely to produce credible motion, subtle changes to low-index transformations might be acceptable. Time-dependent Lipschitz transformations appear more promising.

10. Acknowledgements

This work has been supported by OTKA PD-104710 and the János Bolyai Research Scholarship of the Hungarian Academy of Sciences.

References

1. Ryan Geiss. *Generating complex terrains using the GPU*, chapter 1, pages 7–37. Addison-Wesley Professional, 2007.
2. M. Gervautz and C. Traxler. Representation and realistic rendering of natural phenomena with cyclic CSG graphs. *The Visual Computer*, 12(2):62–74, 1996.
3. J.C. Hart. Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 12(10):527–545, 1996.
4. J.C. Hart and D.J. Sandin. Louis H Kauffman t. Ray Tracing Deterministic 3D Fractals. *Computer Graphics*, 23(3), 1989.
5. J.E. Hutchinson, Dept. of Mathematics, and University of Melbourne. *Fractals and Self Similarity*. University of Melbourne.[Dept. of Mathematics], 1979.
6. H. Nguyen. Gpu gems 3. Part I - Geometry. 2007.
7. A. Pasko, V. Adzhiev, A. Sourin, and V. Savchenko. Function representation in geometric modeling: concepts, implementation and applications. *The Visual Computer*, 11(8):429–446, 1995.
8. P. Prusinkiewicz and A. Lindenmayer. *The algorithmic beauty of plants*. Springer-Verlag New York, Inc. New York, NY, USA, 1990.
9. B. Sowers, T. Menzies, T. McGraw, A. Ross, and W.V. Morgantown. Increasing the Performance and Realism of Procedurally Generated Buildings. 2008.
10. L. Szirmay-Kalos. *Számítógépes grafika*. Computer-Books, Budapest, 1999.
11. L. Szirmay-Kalos and G. Márton. Worst-case versus average-case complexity of ray-shooting. *Journal of Computing*, 61(2):103–131, 1998.
12. C. Traxler and M. Gervautz. Efficient ray tracing of complex natural scenes. *Proceedings Fractals*, 97, 1979.