Computers & Graphics 34 (2010) 167–175

Contents lists available at ScienceDirect

Computers & Graphics

journal homepage: www.elsevier.com/locate/cag

Chaos and Graphics

Realistic rendering 3D IFS fractals in real-time with graphics accelerators

Tomasz Martyn *

Institute of Computer Science, Warsaw University of Technology, ul Nowowiejska 15/19, 00-665 Warsaw, Poland

ARTICLE INFO

Article history:
Received 16 June 2009
Received in revised form
3 August 2009
Accepted 3 August 2009

Keywords:
Fractal
Iterated function system
Real-time rendering
Geometry instancing
Game engine

ABSTRACT

In this paper we present a novel approach to realistic real-time rendering scenes consisting of many affine IFS fractals. In order to illuminate the fractals, we propose a new method for estimating normals at fractal surface points. The method is based on approximations of the convex hulls for fractal subsets. The geometry of a fractal is represented as a collection of circular splats. We also show how to take advantage of self-similarity and hardware geometry instancing so as to store fractal models with extremely small memory requirements. As a result, the geometry data related to hundreds of fractals can be kept entirely in the video RAM of a graphics adapter and rendered in real-time using even medium-power graphics accelerators.

1. Introduction

The classic books *The fractal geometry of nature* [1] by Mandelbrot and *Fractals everywhere* [2] by Barnsley as well as a great deal of later work show that the mathematical language of fractal geometry is well-suited for describing natural phenomena. The main reason for this is that fractals, like many creations encountered in nature, are usually irregular, intricate shapes which are organized according to some rules of self-similarity. Moreover, the mathematical beauty of fractals arises from the fact that despite their geometrical complexity, fractals are usually defined using relatively simple, compact mathematical expressions. This seeming paradox that links the complexity of nature with simple mathematical descriptions is just what makes fractal geometry to be regarded as the "geometry of nature".

However, despite the huge potential to describe and, thus, mimic shapes found in nature, fractals are often not utilized as models of real-time graphics applications, e.g. computer games. In the present games fractal algorithms are used only to generate textures [3] and the geometry that can be easily represented by meshes of triangles. The main reason for this is that the architecture of today's graphics adapters is mostly dedicated to processing batches of triangles. A well-known example of fractal objects used in games are terrains, which can be natively represented in the form of a grid of vertices perturbed by a fractal noise [4]. As another example one could point out branched structures of trees represented as unions of cylinder meshes generated algorithmically using L-systems (if we accept that the product of L-systems can be considered as being within the broad meaning of the term "fractal"). Nevertheless, due to the relative large number of polygons involved in a cylinder mesh, the branching patterns utilized nowadays in computer games are characterized by a low level of branching—probably too low so as to regard the patterns as intricate fractal shape. Anyway, foliage (if present) of trees is represented "non-geometrically" by means of (usually) a group of textures with an alpha channel, as is also the case for herbaceous plants and grass. Undoubtedly such approaches may give good results when perceived at appropriate distances and preferred viewpoints. However in applications with a free-moving FPP or TPP camera[1] it usually quickly becomes apparent that the visually attractive plants are only an illusion based on textures.

One of the main obstacles for real-time rendering of fractal objects is that fractals, in general, have normal vectors undefined at their points (some exceptions can be found, e.g. in [5]). The normals, however, are necessary to do shading and lighting.[2] Therefore, realistic rendering of fractals is not trivial.

Second, fractals are point sets and to render them at accuracy that goes with the nowadays screen resolutions usually hundreds of thousands of points per fractal are needed. In terms of memory requirements this can be read as about 30 MB per fractal, and the application of an adaptive level of detail (LOD)—an essential

* Tel.: +48 22 660 75 41.
E-mail address: martyn@ii.pw.edu.pl

[1] That is, a First-Person Perspective and, respectively, Third-Person Perspective camera.

[2] Ignoring some lighting models applied in volume visualization and so-called non-realistic computer graphics, where normals are not used to compute the color at a point of a surface.

0097-8493/$ - see front matter © 2009 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cag.2009.10.001

element of every game engine—additionally multiplies this number. Since at the time of this writing graphics adapters have at most 1024 MB of VRAM (and transferring data between CPU and GPU is relatively slow) taking advantage of fractals as models for computer game purposes is a challenge.[3]

There have been several approaches to realistic rendering of 3D fractals proposed. Moreover, taking into account the today's graphics accelerators, some of the approaches may be regarded as real-time visualization methods even if they were not recognized as such originally.

An early paper by Norton [6] presents an approach to rendering 3D projections of quaternion Julia sets. Using the boundary tracing algorithm, the Julia set boundary points were determined and visualized with Z-buffer algorithm. In order to apply lighting, the normal vector at a rendered point was determined based neighboring Z-buffer values. (The same method for visualizing quaternion Julia sets can be also noticed in [7].) The method can be implemented in real-time on a modern GPU.

A method for interactive rendering quaternion Julia sets was presented by Hart et al. [8]. To generate Julia set points they used an extension of the inverse iteration algorithm. Then, the points were passed to Z-buffer to determine visibility. The normals were estimated using Z-buffer values like in [6].

A paper [10] by Hart et al. shows how to visualize quaternion Julia sets with ray tracing. To determine the ray-fractal intersection a point-to-fractal distance estimator and unbounding volumes were used. The normal at the point of a fractal surface was estimated as the gradient of a scalar field generated by the distance estimate function.

In a paper by Hepting et al. [11] they present a method for ray-tracing affine IFS fractals. The approach approximates an IFS fractals with tiny spheres using the adaptive-cut algorithm. In order to light a fractal surface, the normals at the points of ray-sphere intersections were used.

Similar approaches to computing normals at points of an IFS fractal surface can be found in a few next papers devoted to ray-tracing IFS attractors. Like in the paper mentioned above, all the methods estimated the normals as the normals at points of the geometric primitives approximating the subsets of an IFS attractor. For example, in a paper by Traxler et al. [12], as approximating primitives, boxes were used, whereas Gröller [13] and then Wonka et al. [14] utilized for this goal images of a box under some nonlinear mappings.

Another paper by Hart et al. [15] devoted to ray-tracing IFS attractors, presented a different method to compute normals at an IFS fractal surface. There, the normal was determined hierarchically as the weighted sum of the surface normals at the intersections of a ray with the ancestry of bounding volumes that surround the ray-fractal intersection point. As the bounding volumes ellipsoids were used, and three methods of weighting normals were described. The application of this method for computing normals can also be noticed in other approaches to ray-tracing affine IFS attractors [16,17].

Chen et al. [18] proposed a method to rendering affine 3D fractals in real-time. An IFS attractor was rendered as a cloud of points colored using a lighting scheme that did not involve normals into computation. As a result, the approach produced images that from the standpoint of today's computer graphics standards cannot be referred to as realistic ones.

[3] This memory budget problem becomes even more apparent when considering game consoles because they have much less RAM than PCs. For instance, the current generation consoles Microsoft Xbox 360, Sony Playstation 3, and Nintendo Wii are equipped with only 512 MB of RAM.

Nikiel [19] used a 3D grid of voxels to represent an IFS fractal approximation. The volume of voxels was then rendered in real-time using one of the techniques of volume visualization.

Another method by Nikiel [19] was to approximate an IFS attractor with 3D vectors, which could be replaced with CSG primitives (e.g. spheres and cones) to perform rendering by means of the one of the popular graphics APIs.

More recently, an interesting paper by Bourke [20] presents how to employ the online 3D virtual environment of Second Life by Linden Labs [21] to represent and explore approximations of 3D fractals in real-time. The fractal approximations were built using some classic fractal constructions based on the iterative/recursive replication of a geometric primitive. To encode the construction procedures a built-in Second Life scripting language was used. As geometric building primitives the volumes provided by the Second Life modeling environment were used, including boxes, spheres, cones, etc. The lighting of the fractal model was consigned to the Second Life engine, which (presumably) based lighting computation on the surface normals of the building primitives.

In this paper we show how to realistically render scenes consisting of many 3D IFS fractals in real-time. In Section 2 we recall some basic facts concerning iterated function systems. Then, in Section 3, we introduce our method of estimating normals at points of an IFS fractal surface, and discuss a point-based graphics approach to display fractal surfaces. Section 4 shows how hardware geometry instancing can be utilized to store fractal models with extremely small memory requirements. Also, we give some details concerning an implementation based on DirectX 9.0c. Results and some further improvements, including adaptive LOD and occlusion culling, are discussed in Section 5. Finally, Section 6 summarizes the paper.

2. Affine IFS fractals

In this section we recall the major definitions and properties of iterated function systems as well as establish the notation used this paper.

An affine *iterated function system* (IFS) on $\mathbb{R}^3$ is a finite set $\{w_1, \ldots, w_N\}$ of N contractive affine mappings $w_i : \mathbb{R}^3 \to \mathbb{R}^3$. An affine mapping w_i is contractive if there exists an $s \in [0,1)$, such that

$$\|w_i(x) - w_i(y)\| \le s\|x-y\|, \quad \forall x,y \in \mathbb{R}^3. \tag{1}$$

The minimum number for which the above inequality holds is called the *contractivity factor* of the mapping w_i. We will denote this number by $\lambda(w_i)$.

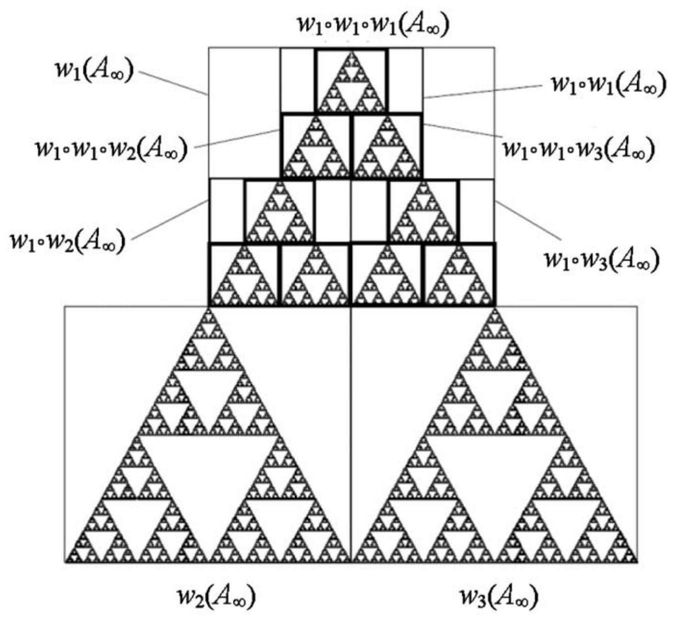
Let $w_i(E)$, $E \subset \mathbb{R}^3$, denote the image $\{w_i(x) : x \in E\}$ of a set E under the mapping $w_i : \mathbb{R}^3 \to \mathbb{R}^3$. An *attractor* of an IFS is the unique solution of the set equation

$$A_\infty = \bigcup_{i=1}^{N} w_i(A_\infty) \tag{2}$$

specified on the family of all the nonempty, bounded and closed subsets of $\mathbb{R}^3$.

The above formula reveals an important characteristic of IFS attractors, namely, they are self-similar sets in the sense that every IFS attractor consists of subsets that are its images under the mappings of the underlying IFS. Since the IFS mappings are contractive, the component subsets $w_i(A_\infty)$, $i = 1, \ldots, N$, are smaller than the attractor itself, for

$$\begin{aligned}\mathrm{diam}(w_i(A_\infty)) = \sup\{&\|w_i(x) - w_i(y)\| \\ &: w_i(x), w_i(y) \in w_i(A_\infty)\} \le \lambda(w_i)\sup\{\|x-y\| \\ &: x,y \in A_\infty\} = \lambda(w_i)\mathrm{diam}(A_\infty),\end{aligned}$$



$w_1 \circ w_1 \circ w_1(A_\infty)$
$w_1(A_\infty)$
$w_1 \circ w_1(A_\infty)$
$w_1 \circ w_1 \circ w_2(A_\infty)$
$w_1 \circ w_1 \circ w_3(A_\infty)$
$w_1 \circ w_2(A_\infty)$
$w_1 \circ w_3(A_\infty)$
$w_2(A_\infty)$
$w_3(A_\infty)$

Fig. 1. The IFS attractor anatomy.

where diam(.) denotes the diameter of a set defined by

$$\operatorname{diam}(E) = \sup\{\|x-y\| : x, y \in E\}. \quad (3)$$

Like the entire attractor, each subset $w_i(A_\infty)$ can be decomposed into N smaller affine copies $w_i \circ w_k(A_\infty)$, $k = 1, \dots, N$, of the attractor, because by Eqs. (1) and (2)

$$w_i(A_\infty) = w_i\left(\bigcup_{k=1}^{N} w_k(A_\infty)\right) = \bigcup_{k=1}^{N} w_i \circ w_k(A_\infty)$$

and

$$\operatorname{diam}(w_i \circ w_k(A_\infty)) \le \lambda(w_i \circ w_k)\operatorname{diam}(A_\infty) \le \lambda(w_i)\lambda(w_k)\operatorname{diam}(A_\infty).$$

Proceeding this way, one can decompose the attractor into smaller and smaller subsets that reach the attractor's points in the limit (see Fig. 1).

This self-similar geometry of the IFS attractors makes it possible to approximate them by means of geometry instancing. Starting with a set enclosing the attractor A_∞, one can construct recursively a set A_ε being union of volumes, by exchanging each volume with the union of N smaller ones. The replacement procedure is continued until the diameter of every volume is less than a given $\varepsilon > 0$. The resulting set A_ε includes and approximates the attractor A_∞ with an error less than ε:

$$h(A_\varepsilon, A_\infty) < \varepsilon,$$

where h denotes the Hausdorff distance between sets defined by

$$h(A,B) = \max(\sup_{x \in A}\inf_{y \in B}\|x-y\|, \sup_{x \in B}\inf_{y \in A}\|x-y\|), \quad A, B \subset \mathbb{R}^3. \quad (4)$$

The union of volumes substituted for a volume bounding the attractor's subset $w_{i_1} \circ \cdots \circ w_{i_k}(A_\infty)$, $i_j \in \{1, \dots, N\}$, is determined on the basis of the $(k+1)$–fold compositions of IFS mappings $w_{i_1} \circ \cdots \circ w_{i_k} \circ w_{i_{k+1}}$, $i_{k+1} = 1, \dots, N$ [11,15–17].

3. IFS fractal normals for real-time rendering

3.1. Estimating normals

As already mentioned in Section 1, one of the main problems when rendering 3D fractals and hence 3D IFS attractors by means of the graphics pipeline of today's hardware is that normals at the points of a fractal surface are undefined. This well-known characteristic of fractals is the direct implication of the fact that, ignoring some special cases (see, e.g. [15]), fractal surfaces are nondifferentiable. However, the knowledge of normals is needed to compute diffuse and specular reflections of light emitted by light sources that have been placed along with the fractal within the scene to be rendered.

A general idea that is commonly used to handle this problem is to approximate the attractor subsets at a given level of detail with some differentiable primitives, and then just render the fractal as a collection of these primitives. Since each building primitive has normals defined at its points, so does the resulting approximation of the fractal.

Very often the general methods of rendering affine IFS fractals engage as the building primitives spheres [11,16], ellipsoids [15], boxes [12], and axis-aligned boxes [17] (see also the more recent paper by Bourke [20]). The main drawback of such approaches is that the geometry of a sphere or an ellipsoid when chosen *a priori* to approximate the fractal subsets has in general little in common with the shapes of these subsets themselves. As a result, the application of a lighting model with respect to the normals determined this way usually makes it apparent that the rendered approximation of a fractal is made up of, for example, tiny spheres (see, e.g. results in [13]). Sometimes such a "bead" effect can be considered interesting. However, it is usually regarded as an undesired side effect because it effectively destroys the impression of infinite level of detail that characterizes fractal sets.

Moreover, there are some fractals for which intuition tells us the most relevant shape of primitives to approximate a given fractal. Consider the Sierpiński tetrahedron, for instance. Undoubtedly, the most appropriate primitive to approximate the subsets of this fractal is the one reflected in its name, namely, just a tetrahedron. On the Internet one can easily find images of the Sierpiński tetrahedron approximated by means of small tetrahedra. These approximations of the classic fractal look usually much more "correct" than of the ones built with other primitives. Besides, there is also a mathematical reason for this. Namely, for any subset $w_{i_1} \dots w_{i_k}(A_\infty)$ of the Sierpiński tetrahedron A_∞, a tetrahedron approximates this subset much better than a sphere and any other convex sets, because it is the convex hull of this set. As such, the tetrahedron is the smallest convex set to approximate the subset $w_{i_1} \dots w_{i_k}(A_\infty)$. Hence, the Hausdorff distance between the tetrahedron and the subset is minimum of the Hausdorff distances between $w_{i_1} \dots w_{i_k}(A_\infty)$ and any set from the family of all convex sets. The conclusion is that the tetrahedron "pretends" to be the subset $w_{i_1} \dots w_{i_k}(A_\infty)$ much better than any other convex set does.

On the other hand, there are potentially infinite number of fractals for which the limitation of the approximating shape geometry to a convex set is at least discussible (see Fig. 2). Let us consider the 3D version of the Barnsley fern as an instance. In this example there is no reason why we should limit our choice of the approximating sets to convex sets. Nevertheless, given a fern subset $w_{i_1} \dots w_{i_k}(A_\infty)$, let us choose only those facets $\{F_j\}$ of the convex hull $\operatorname{conv}(w_{i_1} \dots w_{i_k}(A_\infty))$ for which there is a point in $w_{i_1} \dots w_{i_k}(A_\infty)$ such that F_j is the closest convex hull facet to the point.[4] Then, both with respect to the Hausdorff distance and our intuition the collection of the facets F_j is still a good approximation of $w_{i_1} \dots w_{i_k}(A_\infty)$.

Summarizing the above considerations, we propose the following, simple method of computing normals at IFS fractal points for the purpose of lighting the fractal A_∞ in real-time. Let $\varepsilon > 0$ be given.

[4] More precisely we should speak about a polyhedral approximation of the convex hull rather than the convex hull itself. This is because, in general, the convex hull of a fractal does not need to be bounded by planar facets (see [22] for more detailed discussion).

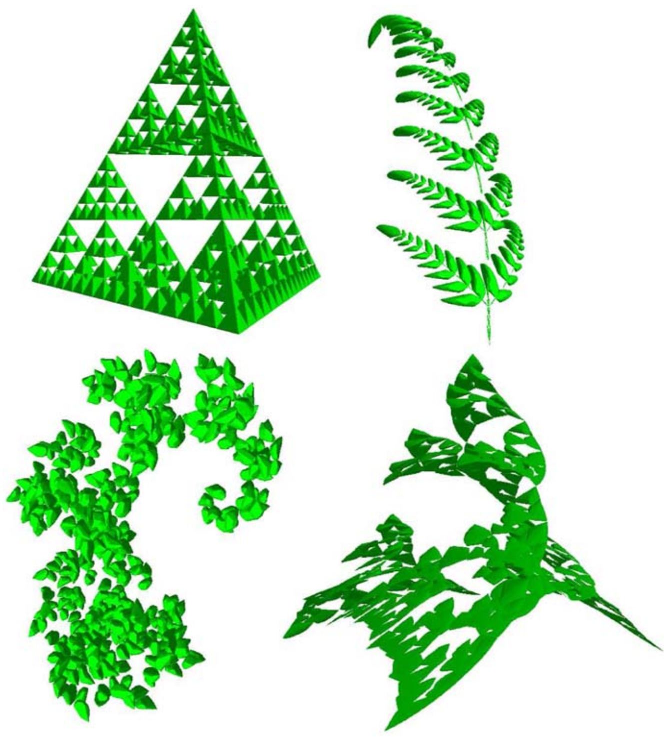


Fig. 2. Examples of approximating IFS fractals with the convex hulls of their subsets. From left to right, top to bottom: the Sierpiński tetrahedron, the 3D Barnsley fern, a 3D dragon, and a perturbed Sierpiński tetrahedron (see [11,2,23] for the relevant IFS codes).

Then for any point p in the attractor subset $w_{i_1}\dots w_{i_k}(A_\infty)$, such that $\operatorname{diam}(w_{i_1}\dots w_{i_k}(A_\infty)) < \varepsilon \le \operatorname{diam}\ (w_{i_1}\dots w_{i_{k-1}}(A_\infty))$, we set the normal at p to be the normal of the closest facet of the convex hull $\operatorname{conv}(w_{i_1}\dots w_{i_k}(A_\infty))$. If there are more than one such facets, we choose the normal of any of the facets.

Since we deal with affine IFS fractals we do not have to compute convex hulls individually for each attractor subset. Such an approach would be computationally expensive, because the number of attractor subsets to consider grows exponentially as ε decreases. Instead, we can just compute the convex hull for the entire fractal and then transform it with the IFS mapping compositions $w_{i_1}\dots w_{i_k}$, $i_j \in \{1,\dots,N\}$, to obtain the hulls for the attractor subsets $w_{i_1}\dots w_{i_k}(A_\infty)$.

In this work to approximate the convex hull for an IFS attractor we apply our vertex cut-off algorithm [24] but any convex hull algorithm for a finite set of points in $\mathbb{R}^3$ can be used [25].

3.2. Rendering

Having normals computed for every point of an approximation of a fractal we can load a vertex buffer with the points and associated normals (and maybe other vertex attributes), and then call the graphics API to draw the buffer as a list of isolated points.

However, rendering of fractals in real time just as a cloud of points usually is not a good idea. First of all, the relevant pixels tend to flicker as the viewer moves about the scene. Second, fractals are often very complex topologically in the sense that they includes infinitely many holes. This is reflected in that fractal approximations with points are usually very sparse,[5] ghost-like objects that appear to be transparent and tend to dissolve in the air as the viewer is getting closer.

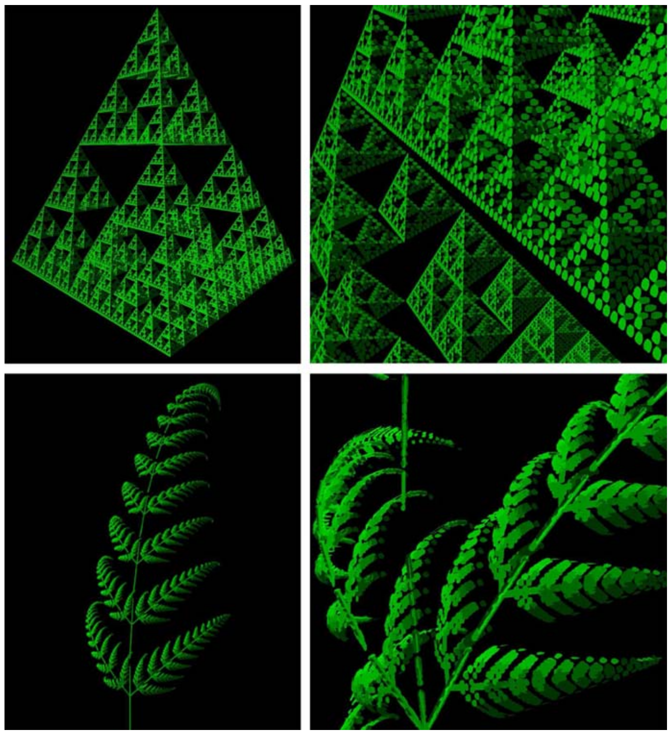


Fig. 3. Effects of rendering the Sierpiński tetrahedron and, respectively, the 3D Barnsley fern with circular splats.

To remedy this we utilize a simplified form of a method from point-based graphics [28], and represent points of a fractal approximation by circular object-aligned splats. The splats are oriented according to the normals estimated at related points using the method from Section 3.1. In order to correctly rasterize and project the splats on the screen in real-time we use the technique proposed by Pajarola et al. [29]. The approach relies on representing each splat by an equilateral triangle onto which a special circular alpha texture is mapped. In the texture each texel within the splat circle is assigned a positive alpha value and all the texels outside circle are assigned a zero alpha value. Now, taking advantage of the alpha test allows one to efficiently discard all the pixels that do not belong to the splat.

As a result, rather than drawing the list of isolated points, we load the vertex buffer with the triangle vertices and normals, and then call the graphics API to draw the buffer as a list of triangles. Effects of the application of this approach to fractals can be seen in Fig. 3.

4. Taking advantage of geometry instancing API

4.1. General idea

As long as the rendered scene consists of a single fractal (and maybe some other relatively simple piece of geometry), there is no problem with storing and rendering the fractal using a single vertex buffer. However, a problem will appear if we want to render a scene composed of hundreds of different fractals, e.g. a field of unique fractal plants in a computer game. The reason is that a single 3D fractal approximation so as to meet the nowadays screen resolutions should consist of hundreds of thousands of points. This in terms of memory requirements reflects in the number of about 30 MB per fractal surface. Moreover, applying an

[5] In fact, one can show that fractals in general do not have well-defined densities, at least in the classical, Lebesgue sense [26,27].

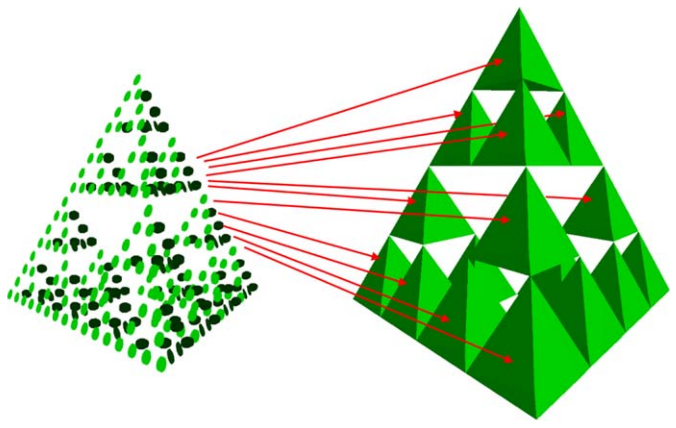


Fig. 4. The idea of representing an IFS fractal in terms of a geometry $\mathcal{G}$ arranged in space according to an instancing pattern $\mathcal{I}$.

adaptive LOD algorithm based on storing the fractal approximation at different levels of detail additionally multiplies this number. As the today's graphics cards have at most 1024 MB of VRAM, a rather small number of fractal models could fit in the memory of a graphics card. This makes such an approach practically unusable from the point of view of real-time graphics applications (computer games for instance).

Fortunately, fractals are self-similar sets and as such they are well suited to the Geometry Instancing facility that has been implemented in hardware for quite a few years (in any hardware that fully supports DirectX 9.0c and Shader Model 3.0, e.g. regarding NVidia as of GeForce 6 Series). Originally the Geometry Instancing API was developed as a mechanism of rendering the same mesh multiple times in a single call to Direct3D so as to minimize the number of relatively slow calls to the driver. As such, the mechanism was intended to accelerate rendering, and so the emphasis was on reducing rendering time. Nevertheless, if we look at all the instances of a given geometry as a one big object, say a forest for example, we will see that the Geometry Instancing is also a way to reduce space complexity. Rather than representing the forest as a set of individual trees, the use of the Geometry Instancing allows us to store and render the forest based on a model of a single tree equipped with a set of mappings to transform the model to obtain the forest. Since to encode a mapping itself a few floating-point numbers are usually enough,[6] whereas a representation of a tree model often consumes much more space, the benefits of the Geometry Instancing as a way of decreasing memory requirements are obvious.

Now, instead of the forest we can just take any IFS fractal and consider it as an intricate object built of instances of a geometry $\mathcal{G}$ that is arranged in space according to an instancing pattern $\mathcal{I}$ (see Fig. 4). The geometry $\mathcal{G}$ is a coarse approximation of the entire fractal (a number of points of the order of hundred is usually enough), and computed by means of the one of the known algorithms such as the adaptive-cut algorithm [11] or the chaos game [2]. The instancing pattern is defined by transformations that maps $\mathcal{G}$—geometry space to the adequate position in local space of the fractal. We specify the transformations as IFS mapping compositions $w_{i_1}\dots w_{i_k}$, $i_j \in \{1,\dots,N\}$, of the contractivity factors $\lambda(w_{i_1}\dots w_{i_k}) < \varepsilon \le \lambda(w_{i_1}\dots w_{i_{k-1}})$ for some $\varepsilon > 0$ given.

In this setting the total number of points to render is equal to the number of points of the geometry $\mathcal{G}$ multiplied by the number of the instances forming the instancing pattern. On the other hand, applying Geometry Instancing involves $sizeof(\mathcal{I})+sizeof(\mathcal{G})$ memory to store a fractal. Hence, taking advantage of the Geometry Instancing in rendering of 3D affine IFS fractals results in

$$\frac{size\ of(\mathcal{G}) \times NumInstances}{size\ of(\mathcal{I}) + size\ of(\mathcal{G})} \tag{5}$$

times less memory to store a fractal when compared to the method that uses a single vertex buffer to store all the points of a fractal approximation. Moreover, in the correct application of our method $size\ of(\mathcal{I}) \gg size\ of(\mathcal{G})$, so the value of (5) is approximately equal to

$$\frac{size\ of(\mathcal{G})}{size\ of(\mathcal{I})/NumInstances}. \tag{6}$$

As a result, in the case of an instancing pattern defined with 3D affine transformations, the application of our method involves approximately $size\ of(\mathcal{G})/(12 \times size\ of(float))$—times less memory for storage with respect to storing the points of the fractal approximation explicitly. In turn, in terms of the Hausdorff distance the resulting set of points approximates the IFS fractal A_∞ with an error not exceeding the value of $\varepsilon\,\mathrm{diam}(A_\infty)(\mathcal{G}, A_\infty)$ (Eqs. (3) and (4)).

4.2. Implementation details

Since the most popular operating system for PC computer games is still Microsoft Windows XP equipped with the DirectX 9.0c API we decided to implement the presented ideas in a DirectX 9.0c application.

Like in the "ordinary" utilization of the Geometry Instancing API, in our fractal case two Direct3D vertex buffers have to be employed. The primary stream contains the single copy of the points of the geometry $\mathcal{G}$ (each point represented by three vertices to define a splat with a triangle) accompanied by normals. The secondary stream contains the instancing pattern data, that is, the matrices representing affine IFS mapping compositions which transform the geometry $\mathcal{G}$ into adequate parts of an IFS fractal. We encode each matrix by storing its first three columns into three 4D texture coordinates[7] and, thus, ignoring the matrix's last, forth column (which is always $[0,0,0,1]^T$) to minimize bandwidth usage.

In addition, in order to render an IFS fractal with the Geometry Instancing API we have to set the proper stream frequency divider values for the geometry $\mathcal{G}$ and instancing pattern streams. We do this in a standard way with the use of the `SetStreamSourceFreq` method of the `IDirect3DDevice` interface [30].

Moreover, to prevent computing the inverse-transpose of 3×3 matrices (that are responsible for transforming normals of the geometry $\mathcal{G}$) in a vertex shader, we compute them off-line and then supply to the vertex shader as additional data of the instancing pattern. We encode each matrix with three texture coordinates in which we save the matrix's rows.

Since the Geometry Instancing API requires the vertices of the instanced geometry to be indexed, we additionally have to assign successive natural numbers to the $\mathcal{G}$—geometry entries. The numbers are loaded into a Direct3D index buffer.

As a result, we use the following vertex structure:

```
struct FRACTAL_VERTEX {
          struct GEOMETRY_DATA {
            D3DXVECTOR4 pos;
            D3DXVECTOR3 normal;
          };
```

[6] E.g. a general 3D affine mapping needs only 12 floating-point numbers to store.

[7] Recall that DirectX uses the column matrix arrangement.

```
            GEOMETRY_DATA GData;
            struct INSTANCE_DATA {
              // columns of the instance matrix
              D3DXVECTOR4 G2ICol1;
              D3DXVECTOR4 G2ICol2;
              D3DXVECTOR4 G2ICol3;
              // rows of the inverse transpose of the
              // linear part of the instance matrix
              D3DXVECTOR3 ITG2IRow1;
              D3DXVECTOR3 ITG2IRow2;
              D3DXVECTOR3 ITG2IRow3;
            };
            INSTANCE_DATA InstanceData;
};
```

which we map into the following Direct3D9 vertex declaration (we omitted the `D3DDECL` prefixes):

```
D3DVERTEXELEMENT9 FractalVertDecl[] =
{
            // the 0th stream
              {0, 0, FLOAT4, DEFAULT, POSITION, 0},
              {0, 16, FLOAT3, DEFAULT, NORMAL, 0},
            // the 1st stream
              // columns of the instance matrix
              {1, 0, FLOAT4, DEFAULT, TEXCOORD, 0},
              {1, 16, FLOAT4, DEFAULT, TEXCOORD, 1},
              {1, 32, FLOAT4, DEFAULT, TEXCOORD, 2},
              //rows of the inverse transpose matrix
              {1, 48, FLOAT3, DEFAULT, TEXCOORD, 3},
              {1, 64, FLOAT3, DEFAULT, TEXCOORD, 4},
              {1, 80, FLOAT3, DEFAULT, TEXCOORD, 5},
};
```

We retrieve the matrices in the vertex shader with the following HLSL code:

```
float4 row1 = float4(G2ICol1.x, G2ICol2.x,
  G2ICol3.x, 0.0f);
float4 row2 = float4(G2ICol1.y, G2ICol2.y,
  G2ICol3.y, 0.0f);
float4 row3 = float4(G2ICol1.z, G2ICol2.z,
  G2ICol3.z, 0.0f);
float4 row4 = float4(G2ICol1.w, G2ICol2.w,
  G2ICol3.w, 1.0f);
float4x4 G2IMat = float4x4(row1, row2, row3, row4);
float3x3 ITG2IMat = float3x3(ITG2IRow1, ITG2IRow2,
  ITG2IRow3);
```

Now, using the `G2IMat` and `ITG2IMat` matrices we transform vertices and, respectively, associated normals from $\mathcal{G}$–geometry space to local space of the fractal. Then, the vertices are processed in the standard way, amongst other the vertex shader transforms each vertex to its proper projection space position, computes a vertex color, etc.

Nevertheless, this is not yet a full list of operations we have to perform in the vertex shader. Recall that an IFS may consist of general affine mappings, and this means that the mappings can bear a non-uniform scaling. Therefore, if we defined our circular splats in $\mathcal{G}$–geometry space and then just applied the `G2IMat` affine mapping to transform a splat to local space of the fractal, the result in general would be wrong. In such a case we would end up with a non-circular (elliptical or even degenerated to a line or a point) and incorrectly oriented splat image, whose normal did not match the vertex normal transformed with the `ITG2IMat` mapping. This implies that we cannot determine splats off-line on the CPU side. Instead, we have to perform the relevant computation in the vertex shader just after the transformation from $\mathcal{G}$–geometry space to fractal local space is done.

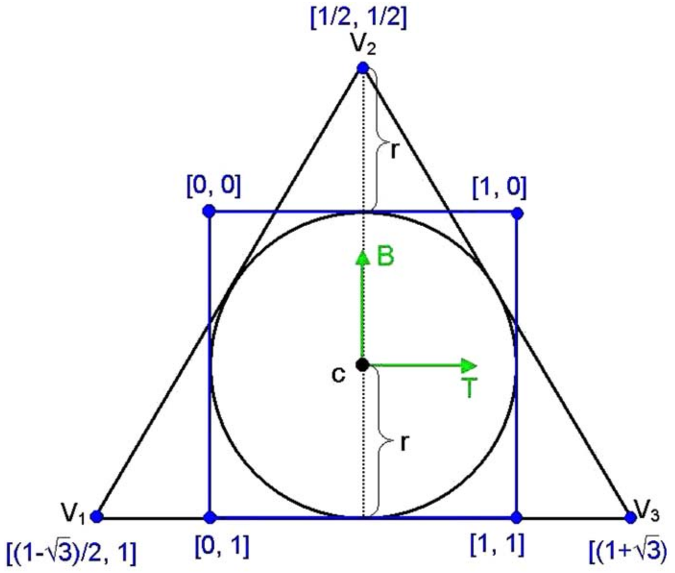


Fig. 5. Generating the splat-texture coordinates for the vertices of an equilateral triangle. The texture coordinates are marked in blue (see the ComputeSplat function in the main text).

As mentioned in Section 3.2, we define a splat using an equilateral triangle and a circular alpha texture mapped onto it. To specify the splat in the vertex shader we have to determine the coordinates of the triangle vertices in fractal local space and the associated texture coordinates on the basis of the coordinates of the normal and center in fractal local space. To do this we first compute the tangent `T` and binormal `B` vectors relative to the given normal `N` using the following, "double cross product" function (in HLSL):

```
void ComputeTB(in float3 N, out float3 T, out float3 B)
{
            float mincoord;
            float3 tmpV;
            if (abs(N.x) < abs(N.y)) {
              mincoord = abs(N.x);
              tmpV = float3(1.0f, 0, 0);
            }
            else {
              mincoord = abs(N.y);
              tmpV = float3(0, 1.0f, 0);
            }
            if (abs(N.z) < mincoord)
              tmpV = float3(0, 0, 1.0f);
            T = normalize(cross(N, tmpV));
            B = normalize(cross(T, N));
}
```

Having `T` and `B` vectors determined, we compute the coordinates of the triangle vertices and associated texture coordinates by means of the following simple HLSL function (see also Fig. 5):

```
void ComputeSplat(int NoVert, float3 T, float3 B,
  float3 c,
            out float3 Vert, out float2 Tex)
{
    switch(NoVert) {
    case(1) :
      Tex = float2((1.0f - sqrt(3.0f))/2.0f, 1.0f);
      Vert = c - r*(3.0f*T/sqrt(3.0f) + B);
      case(2) :
        Tex = float2(0.5f, −0.5f);
        Vert = c + 2.0f*r*B;
      case(3) :
        Tex = float2((1.0f + sqrt(3.0f))/2.0f, 1.0f);
        Vert = c + r*(3.0f*T/sqrt(3.0f) −B);
      }
  }
```

Here, NoVert is the number in $\{1,2,3\}$. It is used to distinguish between vertices of a given triangle. We set the value for each vertex while loading the $\mathcal{G}-$geometry vertex buffer and encode it in the w-coordinate of the vertex position.

5. Results and future work

In order to test the method presented in this paper in action we developed a FPP walking-on-terrain demo running in Microsoft Windows XP equipped with DirectX 9.0c API. A few screenshots are presented in Figs. 6–8.

The demo was based on a simple game engine which had been designed and coded in C++ previously by the author for educational purposes. It should be pointed out that the engine itself was not dedicated to visualization of fractals in any special sense and allows a game world to be built with any geometries that can be represented with Direct3D vertex and index buffers. This means that the approach presented in this paper does not require any special endeavors and modifications in an existing code and could be used along with any graphics engine that supports vertex buffers and geometry instancing. This potential compatibility with other engines is very important feature of our

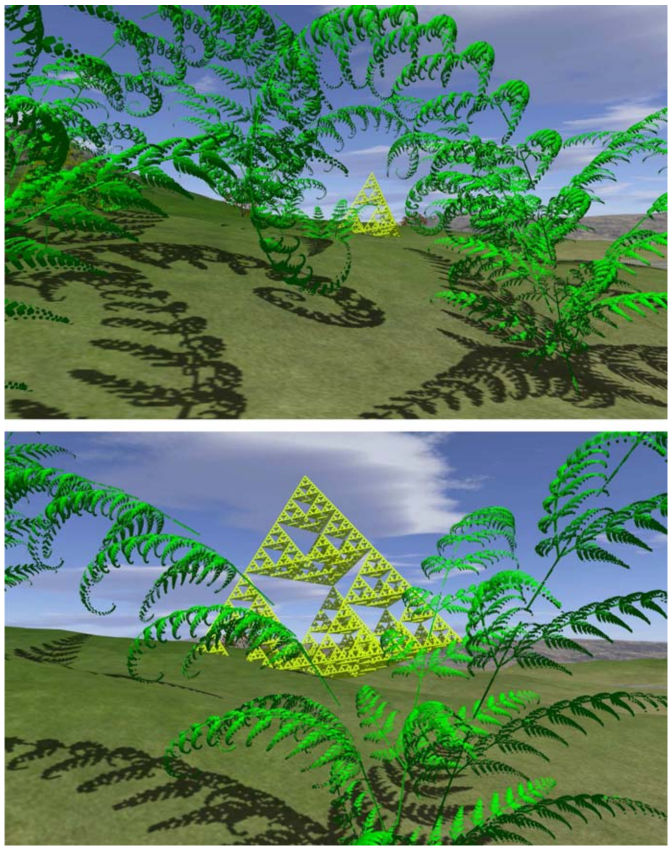


Fig. 7. The Sierpiński tetrahedron and ferns (screenshots from a real-time demo application).

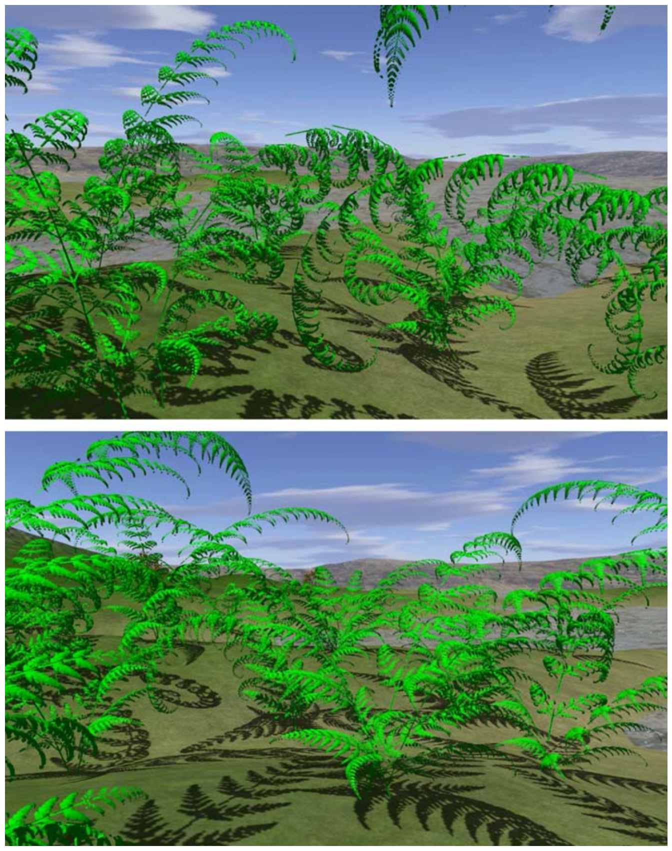


Fig. 6. Variations on the 3D Barnsley fern (screenshots from a real-time demo application).

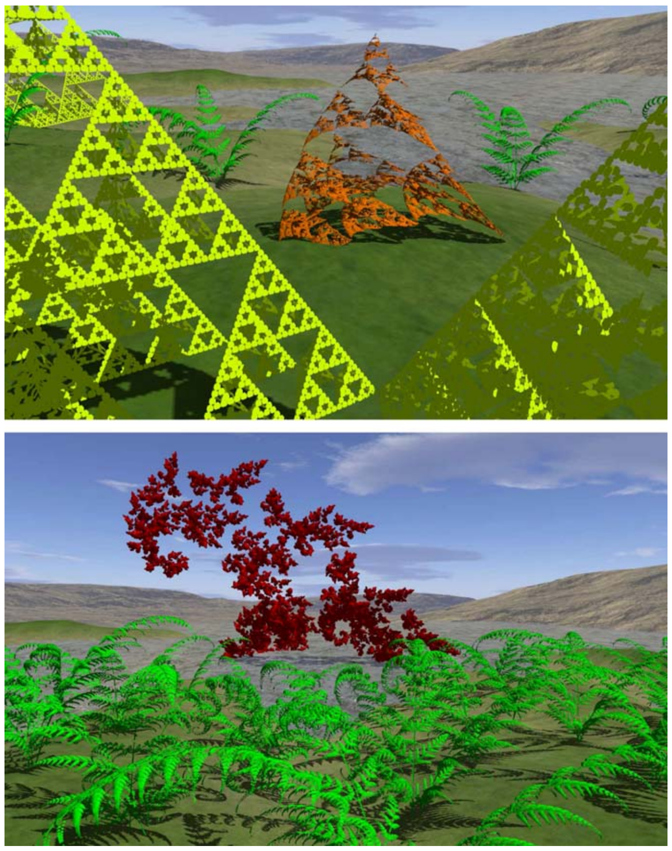


Fig. 8. A perturbed Sierpiński tetrahedron and a 3D dragon (screenshots from a real-time demo application).

approach, especially if we take into account that the development of a graphics engine is usually a complex and expensive process.

The world of the mentioned demo consists of a number of objects represented in a "standard" manner as triangle meshes (models of trees, water, and terrain itself) as well as more than a hundred of IFS fractals. The latter include a number of variations on the 3D Barnsley fern, a few classic Sierpiński tetrahedrons, their perturbed versions, and a 3D dragon. Each fractal was represented individually by its own couple of vertex buffers (an $\mathcal{G}$–geometry buffer and an instancing pattern buffer—see Section 4.2), independently of the other fractals. In terms of the number of splats this implies about 7×10^7 splats constituting the highest LOD resolution of all the fractals, and more than 1.3×10^8 splats for all six implemented LOD resolutions. However, thanks to taking advantage of hardware geometry instancing, each fractal model at all LOD resolutions took only about 100 KB of memory. As a result, despite the huge number of splats, all the fractal geometry required only about 10 MB of memory (that is less then a few medium-sized textures!) and fitted easily in VRAM of a graphics adapter. Let us note that if we try to store all the fractal geometry directly without the application of our techniques based on geometry instancing, it would require about 300 times more memory, that is about 3GB of memory!

To cast shadows and to obtain the self-shadowing effect of the fractal models the shadow mapping technique [31] was used.

When executed on a PC equipped with an NVIDIA GeForce 8800 GT graphics card at the 1680×1050 screen resolution and multisampling of eight samples per pixel, the application was running at the frame rate that did not drop below 50 fps. To speed up rendering the only acceleration techniques applied on the CPU side were the mentioned adaptive LOD and view frustum culling based on axis-aligned bounding boxes.

Although the results of this work are promising, the topic has not been exploited entirely and there are some open problems that need to be solved in the future. First of all, we would like to improve the fractal LOD algorithm. At the current stage, the LOD for fractal models is realized as discrete level of detail (see, e.g. [32]) through switching between models of varying degrees of resolution at predefined distances from the camera. In the demo, a LOD sequence for each fractal consisted of six resolutions which had been generated by changing the number of points of the relevant $\mathcal{G}$–geometry and keeping the instancing pattern constant. The benefits of such an approach are that to represent the entire LOD sequence one only needs to store the geometry $\mathcal{G}$ at different resolutions and a single copy of the instancing pattern. Hence, the switching between resolutions relies on switching between relevant $\mathcal{G}$–geometry buffers. The disadvantage is the popping effect appearing during the switch. In order to redeem the effect we additionally changed the splat size as the switching takes place—the result was good but not ideal. We believe that the simplest way to remedy this is through morphing between consecutive levels of the geometry $\mathcal{G}$.

Another subject of future work is to improve performance by eliminating computational redundancy in the splat computation at the vertex shader stage. Recall that in the current implementation, each splat is represented by three triangle vertices within the stream of a geometry $\mathcal{G}$. This means that all the data associated with each splat in $\mathcal{G}$–geometry space (basically, position and normal) have to be attached to each of the three vertices. As a result, not only the size of the $\mathcal{G}$–geometry buffer is tripled but also, what is more important, almost all computation performed for one of the vertices in the vertex shader (including lighting and computing the TBN basis) is repeated for the two others. In the next version of our engine, which will be based on DirectX 10 or, announced at the time of this writing, DirectX 11, we are going to eliminate this redundancy using the geometry shader capability of vertex emitting. We estimate that this will increase performance about, at least, two times.

One more thing that we have not implemented yet but that could improve performance is an occlusion culling done at the level of subsets of a single fractal. The approach described in this work was developed keeping in mind rendering scenes consisting of not only a single fractal but mainly hundreds of fractals. The latter assumption when made in the context of solid objects usually involves high depth complexity. On the other hand, fractals usually include many holes, and this, in turn, causes the depth complexity to decrease. Therefore, detecting occlusions at the subset level of objects rather than the entire objects themselves appears to be a better approach in the case of fractals. This could be done relatively easily with occlusion queries with respect to the approximations of convex hulls $\text{conv}(w_{i_1} \dots w_{i_k}(A_\infty))$ from Section 3.1 which would be used this time as occluders. Moreover, discarding the entire, occluded subsets before the vertex shader stage would significantly improve performance in the self-occlusion cases of relatively dense fractals. One should note that due to the substantial number of vertices in proportion to the generated fragments, the technique presented in this paper is vertex shader bound rather than—a more common situation—pixel shader bound. As a result, acceleration methods based on the early Z-test (e.g. the popular implementation of deferred shading) are useless here.

Finally, it is worth noticing that the application of our approach is not limited to the geometry of "ordinary" IFS fractals. First of all, it can be extended with no effort to more general classes of iterated function systems, namely recurrent IFSs and hierarchical IFSs. Furthermore, to break the strict self-similarity of affine IFS fractals, which make them to look somewhat artificial as models to mimic shapes found in nature, one can add some randomization to the original geometry. Again, this can be done easily by applying perturbations to the instancing pattern.

6. Summary

We have presented a novel approach to realistic rendering IFS fractals in real-time using the today's graphics accelerators. In order to perform lighting calculations, we have introduced a new method for estimating normals at points of a fractal surface. The method is based on approximations of the convex hulls of IFS fractal subsets. To represent the fractal geometry itself we have used a simplified version of an approach from point-based graphics and render the fractals as collections of circular splats. We also have shown how to take advantage of self-similarity of IFS fractals along with geometry instancing supported by hardware so as to store fractal models with extremely small memory requirements. Thanks to such a solution, the geometry data related to hundreds of IFS fractals can be kept without any problems in VRAM of a graphics adapter and processed efficiently during rendering. An FPP demo application, which we developed to test our algorithms in action, shows that the presented approach is capable of rendering scenes consisting of hundreds of IFS fractals in real-time using even medium-power graphics accelerators. We have also pointed out some further improvements and solutions we would like to apply in future work. To conclude, we believe that IFS fractals and other variants of self-similar geometry are very good candidates to serve as models used in tomorrow's video and computer games.

Acknowledgments

The author thanks Prof. Jan Zabrodzki at CG Lab of Warsaw University of Technology for reading the first version of this paper.

Special thanks go to four anonymous reviewers for many valuable comments that helped to improve the presentation of this paper.

References

[1] Mandelbrot BB. Fractal geometry of nature. New York: W.H. Freeman; 1983.
[2] Barnsley MF. Fractals everywhere, 2nd ed. Boston Academic Press; 1993.
[3] Perlin K. An image synthesizer. Computer Graphics 1985;19(3):287–96.
[4] Ebert S, Musgrave FK, Peachey D, Perlin K, Steve W. Texturing and modeling. A procedural approach, 3rd ed. Los Altos, CA: Morgan Kaufmann; 2002.
[5] Cochran WO, Lewis RR, Hart JC. The normal of a fractal surface. Visual Computer 2001;17(4):209–18.
[6] Norton A. Generation and display of geometric fractals in 3-D. Computer Graphics 1982;16(3):61–7.
[7] Norton A. Julia sets in the quaternions. Computers & Graphics 1989;13(2): 267–78.
[8] Hart JC, Kauffman LH, Sandin DJ. Interactive visualization of quaternion Julia sets. In: Proceedings of Visualization. IEEE Computer Society Press; 1990. p. 209–18.
[10] Hart JC, Sandin DJ, Kauffman LH. Ray tracing deterministic 3-D fractals. Computer Graphics 1989;23(3):289–96.
[11] Hepting D, Prusinkiewicz P, Saupe D. Rendering methods for iterated function systems. In: Peitgen H-O, Henriques JM, Penedo LF, editors. Fractals in the fundamental and applied sciences. Amsterdam: North-Holland; 1991. p. 183–224.
[12] Traxler C, Gervautz M. Efficient ray tracing of complex natural scenes. In: Novak MM, Dewey TG, editors. Fractal frontiers. Singapore: World Scientific; 1997. p. 431–42.
[13] Gröller E. Modeling and rendering of nonlinear iterated function systems. Computers & Graphics 1994;18(5):739–48.
[14] Wonka P, Gervautz M. Raytracing of nonlinear fractals. In: WSCG Plzen 1998 Proceedings, 1998. p. 424–31.
[15] Hart JC, DeFanti TA. Efficient antialiased rendering of 3-D linear fractals. Computer & Graphics 1991;25(4):91–100.
[16] Martyn T. Efficient ray tracing affine IFS attractors. Computers & Graphics 2001;25(4):665–70.
[17] Martyn T. An approach to ray tracing affine IFS fractals. In: Novak MM, editor. Emergent nature. Singapore: World Scientific; 2001. p. 283–92.
[18] Chen YQ, Bi G. 3-D ifs fractals as real-time graphics model. Computers & Graphics 1997;21(3):367–70.
[19] Nikiel S. Iterated function systems for real-time image synthesis. London: Springer; 2007.
[20] Bourke P. Evaluating Second Life for the collaborative exploration of 3D fractals. Computers & Graphics 2009;33(1):113–7.
[21] Linden Lab, web reference ⟨http://lindenlab.com⟩.
[22] Strichartz RS, Wang Y. Geometry of self-affine tiles I. Indiana University Mathematics Journal 1999;48:1–23.
[23] Martyn T. Tight bounding ball for affine IFS attractor. Computers & Graphics 2003;27(4):535–52.
[24] Martyn T. The vertex cut-off algorithm for convex hulls of IFS attractors. Unpublished manuscript, 2008.
[25] Preparata FP, Shamos MI. Computational geometry: an introduction. New York: Springer; 1988.
[26] Falconer K. Fractal geometry, 2nd ed. Chichester: Wiley; 2003.
[27] Edgar GA. Integral, probability and fractal measures. New York: Springer; 1998.
[28] Gross M, Pfister H, editors. Point-based graphics. Amsterdam: Morgan Kaufmann Publishers; 2007.
[29] Pajarola R, Sainz M, Guidotti P. Confetti: object-space point blending and splatting. IEEE Transactions on Visualization and Computer Graphics 2004;10(5):598–608.
[30] Microsoft DirectX SDK, November 2008.
[31] Luna FD. Introduction to 3D game programming with DirectX9.0c: a shader approach. Worldware Publishing, Inc; 2006.
[32] Eberly DH. 3D game engine design, 2nd ed. Amsterdam: Morgan Kaufmann Publishers; 2007.