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 $\{w_1,\dots,w_N\}$ 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 $w_i(A_\infty)$, $i=1,\dots,N$, are smaller than the attractor itself, for

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