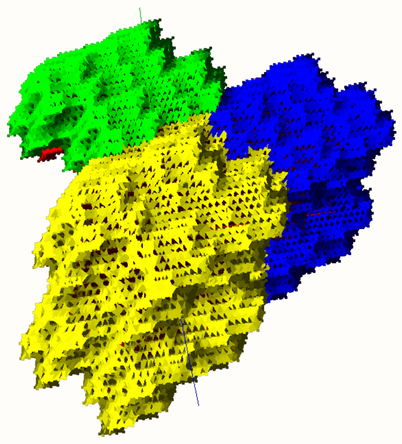
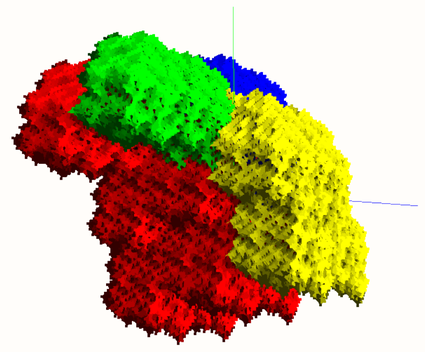


Rauzy fractal 3D rendering

Alexander Taran[1]

[1]Moscow Institute of Physics and Technology, Russia



Introduction

In this poster we are presenting the 3D generalization of Rauzy fractal introduced in [1]. The usual Rauzy fractal is made by building a sequence of points in three-dimensional space and projecting them on contracting plane. Here we build a sequence of points in $\mathbb{R}^4$ and making the orthogonal projection on the contracting subspace. The resulting fractal can be approximately rendered as 3D polygonal surface.

Standard Rauzy fractal

Firstly we describe how a standard Rauzy fractal can be built.

Tribonacci word

The infinite tribonacci word is the word constructed by scheme:

$$t_0 = 1$$
$$t_1 = 12$$
$$t_2 = 1213$$
$$t_n = t_{n-1}t_{n-2}t_{n-3}$$

Fractal construction

Firstly, interpret the sequence of numbers as sequence of unitary vectors with the following rules:

$$1 = \text{direction } x\ (1,0,0)$$
$$2 = \text{direction } y\ (0,1,0)$$
$$3 = \text{direction } z\ (0,0,1)$$

The sequence of numbers defines the sequence of such vectors. Then, build a "stair" tracing the points reached by the sequence of vectors. Every point can be coloured according to the corresponding letter.

Finally, project all the points on the contracting plane (the plane which is orthogonal to the main direction of propagation of the points). Note, none of these projected points escape to infinity.

The detailed description can be found in [1] and [2].

3D generalization of Rauzy fractal

We can build 3D generalization the same way as 2D Rauzy fractal. Firstly, we define "Fourbonacci" sequence:

$$t_0 = 1$$
$$t_1 = 12$$
$$t_2 = 1213$$
$$t_3 = 12131214$$
$$t_n = t_{n-1}t_{n-2}t_{n-3}t_{n-4}$$

(Or we can use substitution rule $\sigma(1) = 12$; $\sigma(2) = 13$; $\sigma(3) = 14$; $\sigma(4) = 1$).

Then we interpret the sequence of numbers as sequence of unitary vectors in $\mathbb{R}^4$ space.

The resulting sequence of points has main direction and orthogonal three-dimensional subspace. We then project all these points onto this subspace and get four sets of points which form the fractal.

Fractal rendering

The implementation is divided on several steps.

First step is just generating points and projecting them onto three-dimensional subspace. Due to insufficient precision of standard floating point numbers precision, we use a third-party library to make projecting computations using custom precision.

Detecting the surface

Second step is trying to build the surface of the fractal. Due to the surface isn't convex, commonly known convex hull algorithms are not applicable here.

Instead of this we firstly try to classify all points on outer and inner points. Simple heuristic can be used here: let's find k nearest neighbors A_i of a point A and find vector sum of all directions from this point to its neighbours: $\vec{v} = \sum_{i=1}^{k} \overrightarrow{AA_i}$. The length of $\vec{v}$ for inner points should be near zero and for outer points should be far from zero.

The experiment shows this method allows us definitely classify all the points. As bonus, it gives us the normal vector to the surface at the point: the normal direction equals $\vec{n} = -\vec{v}$.

Lighting

The surface of the fractal is not smooth but it's very "noisy". To see how it looks we've implemented the lighging using Phong Shading Model [3]. We just multiply the color of every point on shading coefficient:

$$k = A + D * \cos(\vec{v}, \vec{l})$$

Where A states for Ambient lighting component and equals 0.2, D states for Diffuse component and equals 0.8. D is multiplied on cosine between $\vec{n}$ (normal vector to the surface) and $\vec{l}$ (vector from the point to the light source).

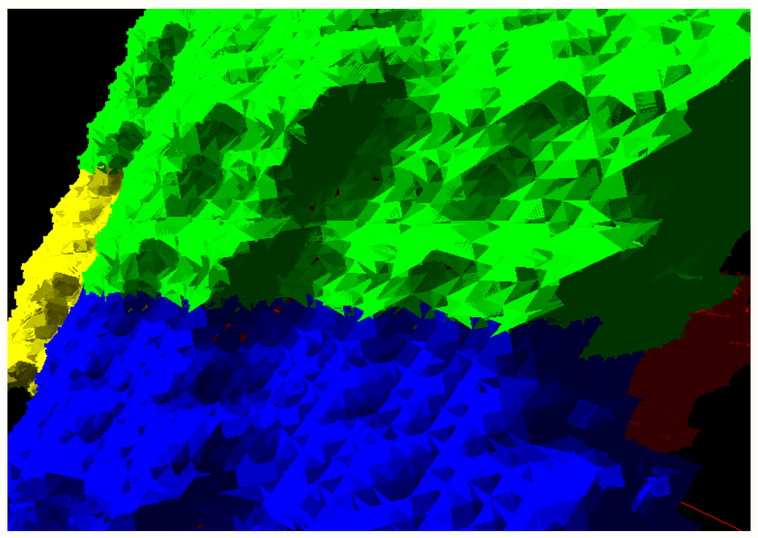


Figure 1: Shaded surface approximation of 3D Rauzy fractal.

Building the surface

The other thing we are trying to do is to approximate the surface of the fractal. As it was mentioned above, surface is not convex and commonly known convex hull algorithms are not applicable. Instead of this we're trying to reconstruct parts of the surface locally.

Consider the point and its k nearest neighbours. We already know the normal vector $\vec{n}$ at this point. After that we can sort these neigubours according their polar angle if we looking along the normal direction. Then we build loacl surface as "fan" structure: any two subsequent neighbors with central point form a triangle. Those triangles forming a lot of "fans" can be build for all points and rendered. As the result we get an approximation of the fractal surface.

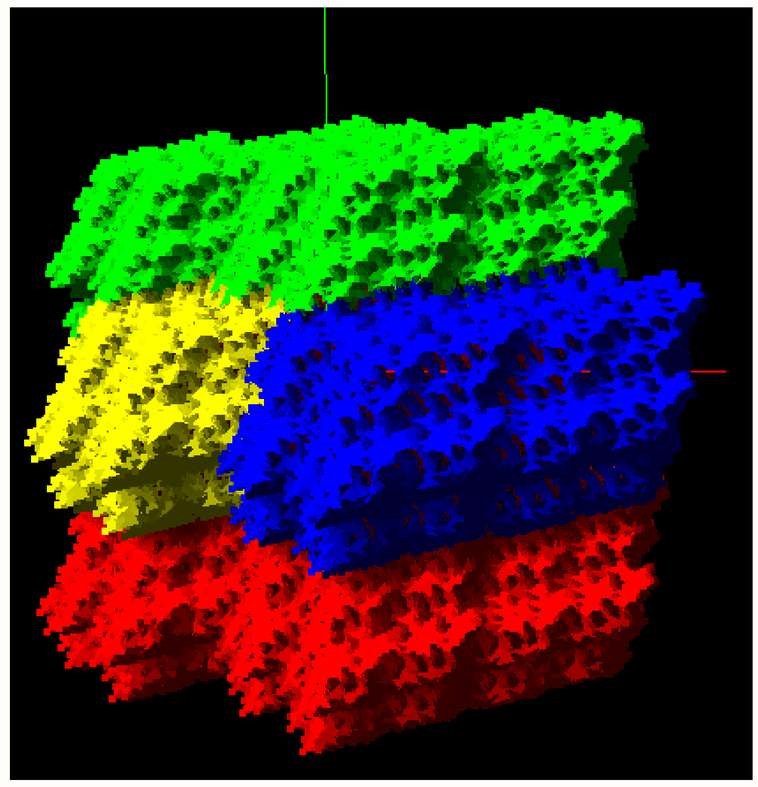


Figure 2: All four substructures of the fractal.

Future work

Calculating the surface and normals requires loading all the data in memory and consumes time. If we choose source sequence of length ~70K points (sequence item t_{14}), it consumes several minutes to calculate all necessary data, after that it allows realtime rendering and animated lighting. The most time consuming procedure is k-nearest neighbours search which loads all of them in memory and builds an octree.

However, surface reconstruction is not ideal: the surface in our construction of an approximating surface is covered by holes, also some of triangles in our construction cover each other. Moreover, the number of necessary neighbours for each point depends on entire number of points and should be determined automatically using fractal properties.

Also, k-nearest neighbours search can be potentially parallelized (with some overhead, but it seems to be faster than single-thread processing).

Actually, the amount of points can be potentially rendered is limited only by video memory size — time resources now are more important. The several millions of points can be processed less than a day and still all be rendered in real-time.

References

[1] G. Rauzy, Nombres algébriques et substitutions, *Bulletin de la Société Mathématique de France, 110:147-178, 1982*

[2] Anne Siegel, Substitution, Rauzy fractals and tilings, *School on Combinatorics, Automata and Number Theory, 2009*

[3] B. T. Phong, Illumination for computer generated pictures, *Communications of ACM 18 (1975), no. 6, 311–317*