Визуализация фрактала Rauzy

Александр Таран

28 декабря 2012 г.

1 Описание фрактала

Фрактал генерируется следующим способом.

Берем 4 строчки, состоящие из цифр 1,2,3,4. Например, «12», «13», «14», «1». Это будут первые 4 члена нашей последовательности. Далее производим некоторое количество итераций. Одна итерация состоит в следующем: берем 4 последних члена последовательности, склеиваем их, как строчки и дописываем в конец. То есть пятый элемент будет равен «1213141», и так далее. Получаем что-то вроде последовательности Фибоначчи. Проделаем какое-то количество итераций и возьмем самую последнюю получившуюся строчку. Теперь сопоставим каждому элементу последовательности координаты точки в стандартном Евклидовом четырехмерном пространстве и один из четырех цветов. Координата по первой оси будет равна числу единиц в последовательности, находящихся "нестрого левее"этого элемента. То же самое для всех остальных осей. Цвет поставим равным текущему элементу.

Теперь имеем набор точек. Известно, что есть вектор, к которому эта последовательность приближается на бесконечности. Спроецируем все точки на ортогональное дополнение к этому вектору и при стремлении числа точек к бесконечности, получающееся множество будет искомым фракталом.

2 Реализация

Для быстрой работы состоит из двух частей: генератора и визуализатора.

2.1 Генератор (generator)

Для сборки есть Makefile в папке generator (должны быть установлены $make, g++, boost$ и $mpfr$) Генератор запускается с единственным параметром: число итераций в генерации фрактала. Оптимальные значения: до 17 (при 17 работает уже ощутимое время и занимает несколько сотен мегабайт на HDD). На выходе в папке gen получаются файлы v17.gen, c17.gen, n17.gen и f17.gen (вершины, их цвета, нормали и грани - тройки индексов

вершин, обозначающие одну грань). Вместо 17 подставляется число итераций, таким образом можно нагенерировать и одновременго хранить данные для разного числа итераций.

Запуск:

./process.sh 17

Генератор состоит из нескольких подпрограмм, последовательно обрабатывающих данные и обменивающихся ими с помощью записи на HDD, таким образом, теоретически объём оперативной памяти не должен быть ограничением. Фактически одна из программ, использующая октодерево для поиска ближайших соседей, загружает сразу все точки в память (но этого можно в будущем избежать, сделав октодерево во внешней памяти и кэшем/частичной загрузвой в память). Остальные используют $O(1)$ памяти и работают в потоковом режиме, поэтому могут потенциально обработать любой объем данных.

Далее следует описание подпрограмм

2.1.1 genSeq

Принимает один числовой аргумент командной строки и выводит на *stdout* последовательность в текстовом виде (состоящую из чисел 1,2,3,4). Инициализационный вектор: «12», «13», «14», «1» (можно поменять в исходнике и пересобрать с другими строчками).

2.1.2 genPoints

Принимает на *stdin* последовательность в текстовом виде и выдает на *stdout* сооответствующую последовательность точек в текстовом виде. Каждая точка описывается четырьмя координатами. Координаты разделены пробелом, точки - прееводом строки.

2.1.3 genColors

Принимает также на *stdin* последовательность в текстовом виде и возвращает для каждого символа соответствующий цвет в виде трех float-значений как (r,g,b). Используются float от 0 до 1, т.к. это формат данных для OpenGL.

2.1.4 grepMainDir

Принимает последовательность точек из реультата предыдущей команды, возвращает направление, вдоль которого будет вычисляться проекция. На данный момент просто возвращает последнюю точку, но можно пытаться усреднять. В любом случае, на конечную картинку это никак не влияет (т.к. главное направление проецирования ярко выражено).

2.1.5 projector

Принимает grepMainDir-а через параметры командной строки, и результат genPoints через *stdin* Проецирует точки вдоль заданного направления. Использует библиотеку $mpfr$ для чисел с плавающей точкой высокой точности (1024 бита). Плоскость проекции ортогональна вектору, вдоль которого делается проекция. Формально, базис в ней не важен, поскольку это влияет только на то, как будет повернута в пространстве итоговая фигура, но для удобства выбирается так:

Пусть входной вектор - (x,y,z,w). Тогда дополняем его до очевидной ортогональной тройки векторами (-y, x, 0, 0) и (0, 0, -w, z). Далее надо найти четвертый с помощью решения линейной системы. правая часть системы имеет первые три нуля (условия ортогональности базиса) и четвертая строка задает линейное условие на искомый вектор, т.к. он может быть каким угодно по длине. Можно взять, например (1,1,1,1) и 1, то есть сумма координат равна единице. После всего этого нормируем вектора и получаем базис. Последние три вектора - базис в плоскости проекции. Программа возвращает координаты спроецированных точек именно в этом базисе (в этот момент у точек становится по три координаты).

2.1.6 genFastNearestNeighbours

Принимает список точек через *stdin* и число ближайших соседей как аргумент командной строки, возвращает для каждой точки в отдельной строке: координаты самой точки и затем последовательно координаты ее K ближайших соседей. Загружает сразу все точки в память и строит октодерево, поэтому работает на несколько порядков быстрее обычной сортировки.

2.1.7 genNormals

Принимает на вход результат работы предыдущей программы, вычисляет нормаль как минус сумму векторов от точки до каждого из ее соседей (без нормализации).

2.1.8 filterNormals.py

Принимает список нормалей через *stdin* и пороговое значение как аргумент командной строки, возвращает список из нулей и единиц (в отдельных строках). Если длина нормали больше порогового значения, то пишет 1, иначе 0. Таким образом точки разделаются на точки границы и внутренние.

2.1.9 applyFilterMask.py

Принимает имя файла с результатом работы предыдущей программы как аргумент командной строки и данные для фильтрации на *stdin*. Оставляет только те данные, для которых значение равно 1. Таким образом фильтруются все спроецированные точки, цвета и нормали. После фильтрации еще

раз запускается genFastNearestNeighbours уже с маленьким входным параметром (например, 6) для вычисления треугольников, аппроксимирующих поверхность.

2.1.10 genFaces.py

Принимает как входные аргументы имена трех файлов. В первом файле - точки и их соседи, во втором - вершины, в третьем - нормали. Программа пишет на выход комбинированные данные в определенном формате. Данные в файле - это список вершин (Vertices), записанный таким образом, что каждые три последовательные вершины представляют собой треугольник для рисования.

$Vertex := position_x, position_y, position_z, color_r, color_g, color_b, normal_x, normal_y, normal_z$

Делает он это так: берет точку и всех ее соседей, сортирует соседей по углу вокруг нормали в точке и генерирует треугольники так, что одна из вершин треугольника - это сама точка, а две другие - последовательные соседи (т.е. получается что-то вроде «веера»).

2.1.11 genVbo

Принимает на *stdin* последовательность float-значений в текстовом виде и записывает их на *stdout* в бинарном формате: 4 байта на число - это сделано по причине увеличения производительности: на C++ это делается быстро, а питон несколько миллионов точек будет переваривать медленно.

2.1.12 process.sh

Главный скрипт, вызывающий все предыдущие программы, принимающий на вход один параметр - число итераций и генерирующий на выходе готовые данные для визуализатора.

2.2 Визуализатор (renderer)

Написан на *python*, использует пакеты $pygame, pyopengl, numpy$ Запуск: # python rauzy.py 17 В таком случае он будет читать данные из файла «../generator/gen/f17.gen».

Оттестировано на Ubuntu Linux и на Mac OS X. На Windows не должно быть особых проблем, если стоят все необходимые утилиты.

Нормальная производительность обнаружена при нескольких миллионов треугольников в кадре (FPS около 60) при числе итераций 17. Тут можно достигать разных значений, варьируя число ближайших соседей (в двух местах) и число итераций при генерации последовательности.

Проект является Open Source и выложен в открытый доступ на github: https://github.com/AlexTaran/rauzy