A Space Filling Algorithm for Generating Procedural Geometry and Texture

Paul Bourke
iVEC@UWA, The University of Western Australia, 35 Stirling Hwy, Crawley, Perth, West Australia 6009. Email: paul.bourke@uwa.edu.au

***Abstract* -- Here we present an algorithm for procedurally generating a range of digital assets including 2 dimensional textures and 2.5 dimensional texture roughness. The approach involves placing shapes randomly, without overlap and with a monotonically decreasing area, within a region on a plane (the 2 dimensional texture). If the process is continued to infinity then the result is space filling thus providing a variable and potentially infinite degree of visual detail. It will be proposed and illustrated that the process is independent of the actual shape being used and as such can find application to a range of texture effects. As a means of generating texture and form procedurally the result has the other desirable property of being fractal, that is, self similar across scales which is characteristic of many packings that occur in nature.**

Index terms* -- *procedural, texture, tiling, packing, space filling.

I. Introduction

The algorithmic (procedural) generation of assets, texture and geometry, within virtual worlds and for the digital movie industry is well established. Well known examples include various noise [1] and fractal based techniques for the generation of terrain [2], clouds [3], plants [4], and urban spaces [5]. These examples and others often involve fractal processes, not only because many natural phenomena can be represented by fractals, but fractal processes generally allow one to create geometric detail on demand and when or where required, for example, creating texture only in front of the player in a first person view. This ability to create variable levels of detail ensures distant objects or objects out of view can be represented at a lower geometric detail. The detail generated can increase as the viewer gets closer and can appreciate the higher resolution. There are other desirable properties of procedural methods including the ability to control performance by generating more or less detail and reducing the payload, detail is generated rather than contained within high resolution images or other data structures. The fact that it is generated also allows for random variation, that is, slightly different textures can be generated reducing the appearance of repetition in a scene.

Fractal processes by definition are those with self similarity across scales, that is, as one zooms into the object it looks similar to the zoomed out view. One would expect then that a process that is intended to mimic natural forms would need to possess this same property.

The above are some desirable properties such a procedural generator of textures should possess; the main ones are summarised here

- The ability to create the texture at variable levels of detail. For example, distant objects do not need as resolved textures as close objects.
- The texture detail can increase smoothly as it is inspected more closely. Absence of this results in so called "popping" in many level of detail (LOD) algorithms where the additional detail is introduced at discrete distances and thus appears as an abrupt visual artifact.
- Textures need to optionally be able to tile the plane seamlessly. This allows a small texture unit to be used to cover a large region.
- The texture can be generated with random variation, usually given by a random number generator seed.
- As an efficient way of delivering detail it must be able to be generated quickly and ideally with simple algorithms.

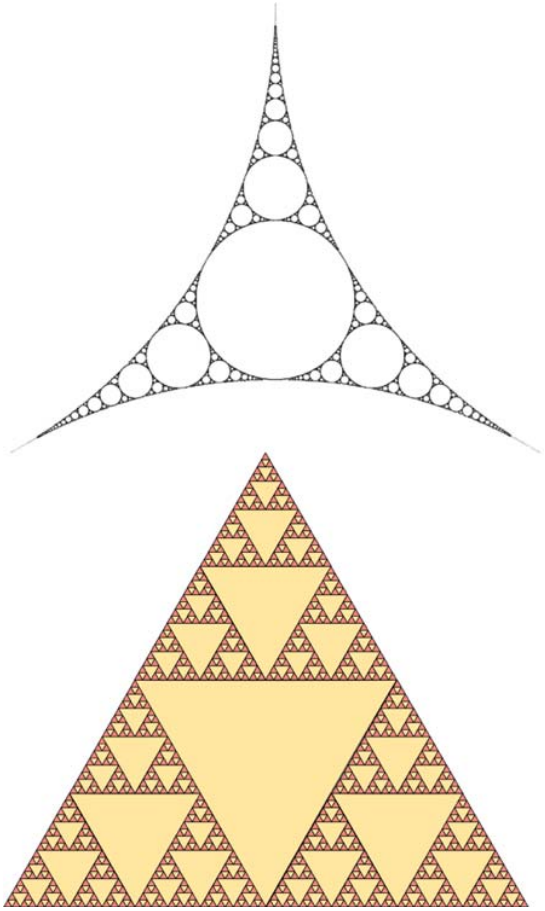


Fig 1. a) Appolonian packing. b) Sierpinski gasket.

DOI: 10.5176/2251-3043_3.2.260

As an introduction to the algorithm we will present how it may be used to create textures, both image based and 2.5 dimensional surface roughness. It should be noted that the general technique readily extends into other dimensions and has been studied extensively in both 1 and 3 dimensions [6]. The process outlined here also provides a new answer to the question of how one might fill a bounded region with an infinite number of identical shapes (this will be relaxed later) so as to eventually fill the region, that is, fill in all the gaps, also referred to as the gasket [7].

An existing packing solution and one that has been studied extensively is a family known as Apollonian [8][9] packings. In the most general form, shapes are iteratively introduced randomly at empty locations, they grow and perhaps move until they touch one or more existing shapes or when they cannot move or grow any further. Other solutions are recursive fractals such as the Sierpinski gasket [10] in which triangles fill the plane. These are both fractal and can be space filling, they both have the property that the objects touch, known as "kissing". They have been used as models of packing in nature, such as pore spaces [11] and rock packings. However casual inspection of many packings in nature show that this kissing property doesn't exist, some examples are shown in figure 2.

The packing presented here on the other hand sees shapes with a particular size added iteratively at random locations. If the shape does not overlap with any existing shape then it is placed permanently at that position otherwise a new random position is tried. The shape does not grow to fill the available gaps, and thus in general the shapes to not touch other shapes, rather the algorithm determines the size of each shape in advance. The question then is "*what is the monotonically decreasing function that determines the size of the added shape at the current iteration*". Clearly, if the size decreases too quickly then space filling will not be achieved. If the size decreases too slowly then the process will run out of space, there will be no gap large enough to add the next shape such that it doesn't overlap with the existing shapes.

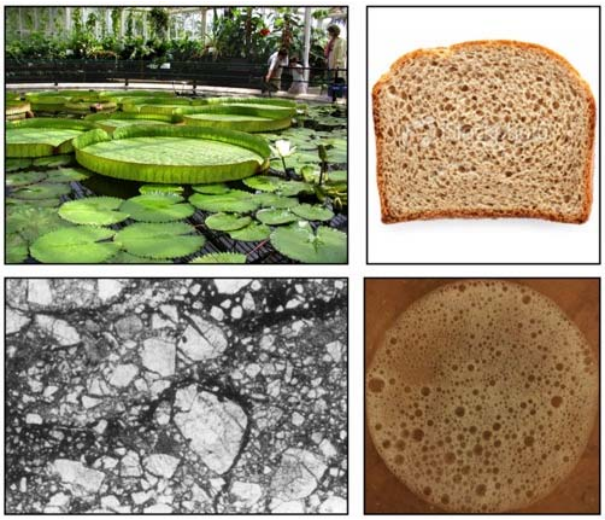


Fig 2. Selection of natural packings that do not exhibit the kissing property, lily pond fronds, pore spacing in breads, rock packings, and bubbles.

II. Algorithm

If A_0 is the area of the first shape and $g(i)$ is the area scaling of the shape on each iteration i then the series of areas is given by

$$A_0, A_0\, g(1), A_0\, g(2), \ldots, A_0\, g(n), \ldots \quad (1)$$

The total area *A* is given by the sum of the above terms, namely

$$A = A_0 \sum_{i=1}^{\infty} g(i) \quad (2)$$

Therefore one is seeking a series $g(i)$ that is monotonically decreasing and sums to a constant, namely the area to be filled. If $g(i)$ decreases too fast then space filling is not achieved, if it doesn't decrease fast enough the procedure fails due to insufficient space for the next shape, see figure 3. While the discussion here concentrates on space filling, low values of *g(i)* can be used as models for where space filling does not occur, or occurs over a limited range of scales.

A series that satisfies the requirement and the one used here to achieve space filling is as follows

$$g(i) = \frac{1}{i^c} \quad (3)$$

where *c* is some constant. This series is recognised as the Riemann Zeta function [12] which is known to converge for $c > 1$. For space filling one can choose a value of *c*, the sum of the series is used to determine the value of A_0 given the area *A* to be filled. Alternatively A_0 can be chosen which in turn determines the value of *c* for space filling.

The algorithm can be summarised as follows. Decide upon the bounded region to be filled and calculate the area *A*. Choose a value of *c* and based upon that and the sum of $g(i)$ calculate the area of the first shape A_0. On each iteration choose a random position from a uniform distribution for the current shape of area $A_0\, g(i)$. If the shape positioned at this candidate position does not result in any overlap with current shapes or the boundary then place the shape at this position, otherwise keep trying other randomly selected positions until successful placement can be made. Given the series proposed here and for a range of values of *c*, a space will always be available for the next shape.

There are four requirements for an implementation that creates these space fillings:

1. A function that calculates the area of the shape given the parameters that define the shape.
2. A function that calculates the parameters of the shape given the area. This is essentially the inverse of the function in requirement (1).
3. A function that performs an intersection test between a shape and the boundary of the region being filled. This is used to determine whether the shape lies within the boundary and is also used to decide on issues of tiling.
4. A function that performs an intersection test between two shapes. This is performed on each proposed placement to ensure the shape to be added does intersect any existing shape, this is the computationally critical comparison since it is applied between the shape to be added at the current iteration and every existing shape.

III. Properties

The value of *c* controls the fractal dimension. For an embedding in dimension *D* (1, 2, 3 ...) the fractal dimension *d* is given by

$$d = D / c \tag{4}$$

Not any value of *c* may be chosen, in each dimension there is an upper limit [6] and the limit depends on the shape being packed. For example in two dimensions (*D*=2) and for spheres the maximum value of *c* is close to 1.5.

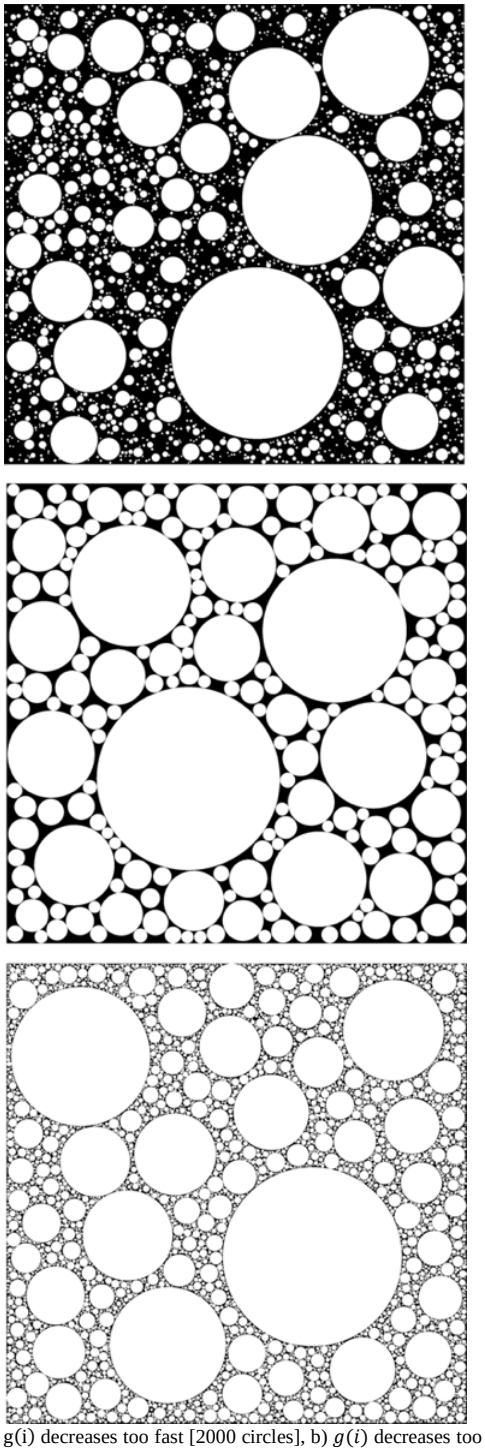


Fig 3. a) g(i) decreases too fast [2000 circles], b) $g(i)$ decreases too slowly [no space after 200 circles], c) $g(i)$ decreases as presented to achieve space filling [5000 circles].

If *c* and A_o are chosen as proposed then it is the authors claim that the process does not halt, that is, it does indeed allow an infinite space filling packing.

When used to create textures that tile seamlessly within a rectangular bounded region of width *w* and height *h*, the algorithm can be modified such that when a placement is made it is also performed at +-*w* and +-*h*. Figure 4 illustrates examples of tileable textures, bounded by a square but with toroidal boundary conditions. By comparison the examples in figure 3 were bounded within a square.

The derivation says nothing about the shape itself, only the area. As such the algorithm will work for any shape and indeed even for mixtures of shapes so long as the decreasing function of area on each iteration, *g(i)*, is honoured. While this property is difficult to prove, it feels intuitively correct and the authors have not identified any shapes that cause the algorithm to fail. This includes shapes with holes, which also get filled, see figure 4 and highly convoluted shapes, see figure 5.

The algorithm is not limited to filling rectangular regions, indeed any shaped region can be filled, with any shape. This capability only requires an intersection test between the shape being tiled and the boundary shape. An example is presented in figure 6, circles filling a toroidal shape. Note that this is a relatively rare property for texturing which is normally based upon rectangular patches and thus often does not look natural at the boundaries of non-rectangular regions.

Figure 5 is an example of a non-simple shape being used for the filling. Extensive experimental tests have been performed with a range of fllling and boundary shapes, no combinations have resulted in the algorithm failing, at least for a range of values of *c*.

The objects are placed randomly within the region being filled. The same random number sequence will produce the same result, similarly variation can be achieved with different random number seeds. The macroscopic appearance of the texture is largely determined by the placement of the first few objects, this is to be expected and indeed is the case for packings found in nature, they are dominated by the largest objects.

A minor modification to control the size of the first shape (and all subsequent shapes) is to modify the function of *g(i)* to

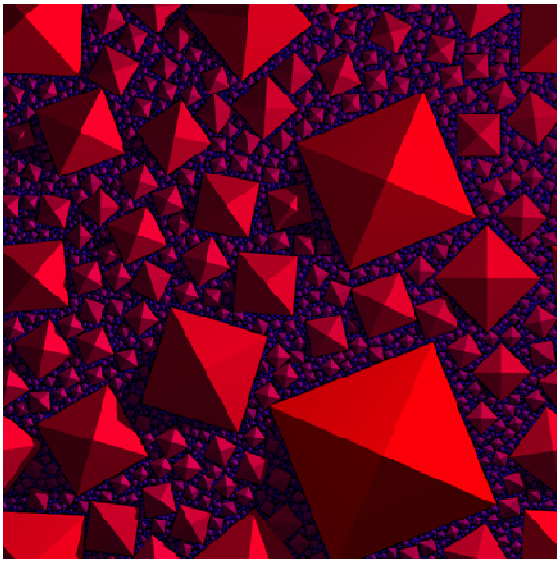
$$g(i) = \frac{1}{(i+N)^c} \tag{5}$$

Where *N* can be any positive real number but is usually an integer. In general this lowers the size of the early shapes while at the same time reducing the rate at which subsequent shapes reduce in area.

The algorithm is readily extended into other dimensions. In one dimension it transforms into packing line segments into a length *L*. The function *g(i)* now defines how the length of each line segment is reduced on each iteration, so similar to equation 2.

$$L = L_0 \sum_{i=1}^{\infty} g(i) \tag{6}$$

In three dimensions $g(i)$ governs how the volume of each shape reduces on each iteration.



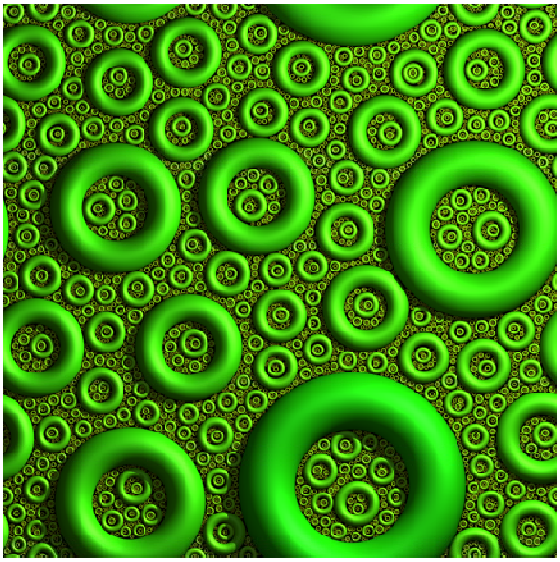


Fig 4. Examples of different shapes and toroidal boundary conditions Randomly orientated pyramids (Upper). Shapes with holes (Lower).

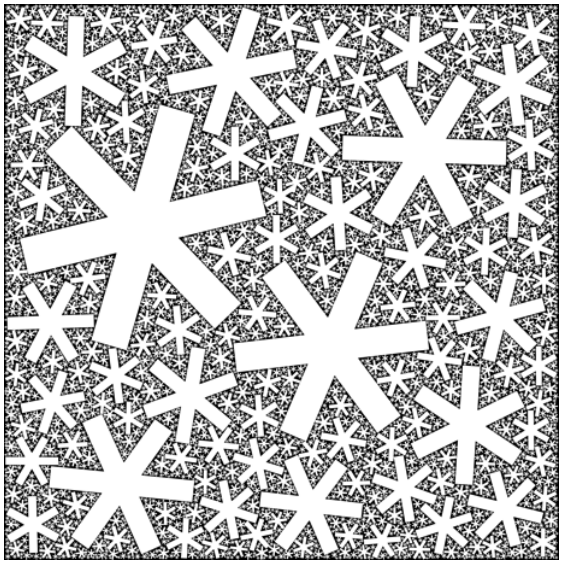


Fig 5. Filling with highly convoluted shapes.

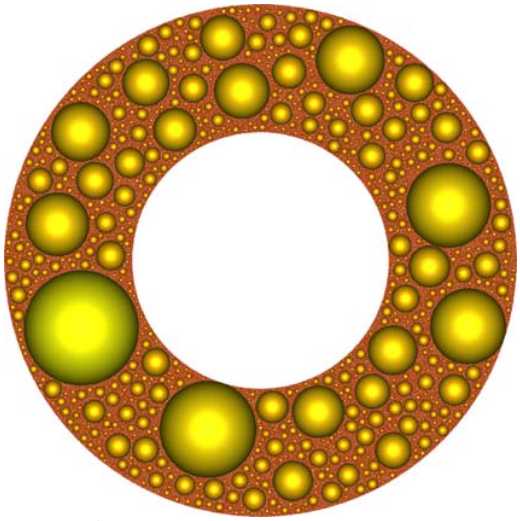


Fig 6. Example of non-rectangular bounded regions.

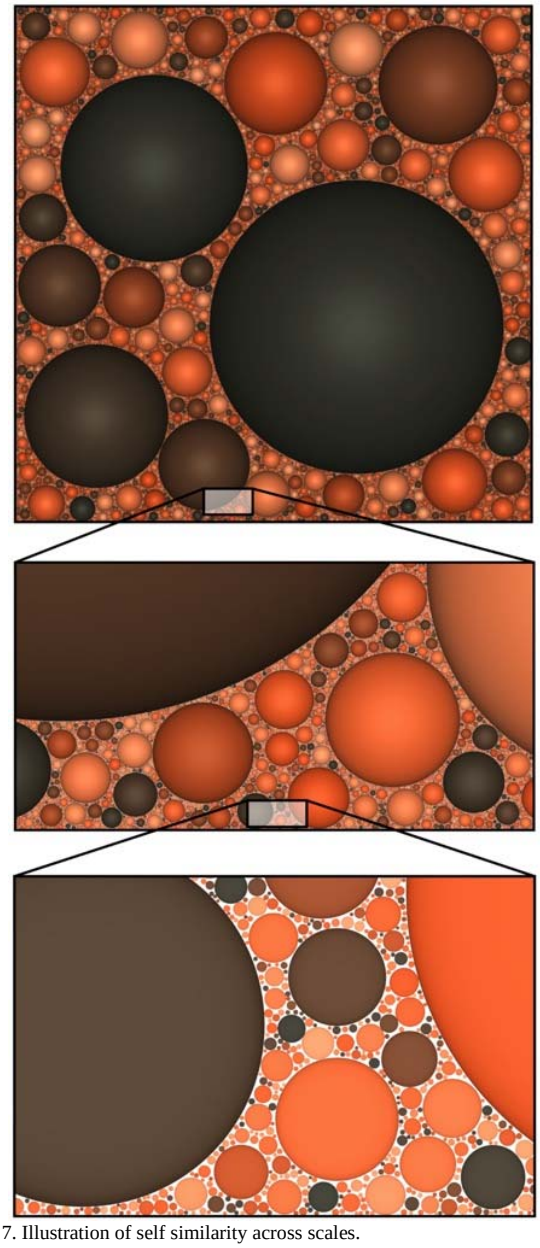


Fig 7. Illustration of self similarity across scales.

V. Conclusion

We have introduced a new algorithm with many of the features that are desirable for creating procedural multi-resolution 2 dimensional textures. The procedural and iterative properties of this algorithm make it ideal for creating textures and geometry at sufficient detail to meet frame rate dictated performance constraints within a gaming engine dependent on the players distance and viewing direction. The algorithm allows for texture generation over a range of fractal dimensions and readily supports the ability to create tileable textures within rectangular regions. The algorithm is simple to implement and can be computed quickly. It results in self similarity across scales and as thus is likely to provide textures that appear similar to naturally occurring texture.

Future work will focus on practical matters, there are two aspects that will be explored. The first is to identify real world approximations to the idealized mathematics and derive the parameters *c* and *N*. The second is to develop plugins for one or more mainstream modeling-rendering-animation software packages, this can ensure that non-experts can make use of the procedural algorithm presented.

Acknowledgement

The algorithm presented has been inspired by John Shier [7][13] who conducted the pioneering research under the title of “statistical geometry”. The work was supported by iVEC through the use of advanced computing resources located at the University of Western Australia.

References

[1] S. Green. nVidia Corporation. "Implementing Improved Perlin Noise". GPU Gems 2, Chapter 26,
[2] J.P. Lewis. "Generalized Stochastic Subdivision", Vol 6, 3 (1987)
[3] G.Y. Gardner. "Visual Simulation of Clouds". SIGGRAPH '85 Proceedings of the 12th annual conference on Computer graphics and interactive techniques..
[4] P. Prusinkiewicz, A. Lindenmayer. "The Algorithmic Beauty of Plants". Springer-Verlag. (1990). ISBN 978-0-387-97297-8.
[5] S. Greuter, J. Parker, N. Stewart, G. Leach. "Real-time procedural generation of `pseudo infinite' cities". GRAPHITE '03 Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia.
[6] J. Shier, P.D. Bourke. An Algorithm for Random Fractal Filling of Space. Computer Graphics Forum. 2013 [In Press].
[7] J Shier. "Filling Space with Random Fractal Non-Overlapping Simple Shapes". Hyperseeing summer 2011 issue, pp. 131-140, published by ISAMA (International Society of the Arts, Mathematics, and Architecture).
[8] P.D. Bourke. "Appolony fractal". Computers and Graphics, Vol 30, Issue 1, January 2006.
[9] C.A. Pickover. "Cleopatra's Necklace and the Aesthetics of Oscillatory Growth". The Visual Computer, Vol 9, No 3.
[10] I. Stewart. "Four Encounters with Sierpinski's Gasket". The Mathematical Intelligencer, 17, No. 1 (1995.
[11] Y.T. Wu, C.X Yang, X.G Yuan. "Drop distributions and numerical simulation of drop wise condensation heat transfer". International Journal of Heat and Mass Transfer, 44 (2001).
[12] T.M. Apostol. "Zeta and Related Functions". NIST Handbook of Mathematical Functions, Cambridge University Press, ISBN 978-0521192255
[13] P.D. Bourke, J. Shier. The authors' web sites http://paulbourke.net/randomtile http://john-art.com

Paul Bourke is a visualization researcher at the University of Western Australia providing scientific visualization services to researchers within the university and to the other iVEC partners. During his career he has worked in organizations where he concentrated on architectural, brain/medical, and astronomy visualization. Of particular interest are novel data capture and display technologies and how they may be used to facilitate insight in scientific research, increase engagement for public outreach and education, create immersive environments, and enhance digital entertainment.

Paul Bourke is the director of the iVEC facility located at the University of Western Australia. The facility hosts supercomputing resources on the campus and acts as an interface to the other supercomputing capabilities provided by iVEC.