UNIVERSITY OF COPENHAGEN

Planet Map Generation by Tetrahedral Subdivision

Mogensen, Torben Ægidius

Published in:
Perspectives of Systems Informatics

Publication date:
2010

Document Version
Publisher's PDF, also known as Version of record

Citation for published version (APA):
Mogensen, T. Æ. (2010). Planet Map Generation by Tetrahedral Subdivision. In A. Pnueli, I. Virbitskaite, & A. Voronkov (Eds.), Perspectives of Systems Informatics: 7th International Andrei Ershov Memorial Conference, PSI 2009. Springer. (Lecture Notes in Computer Science, Vol. 5947).

Download date: 03. Feb. 2016

Planet Map Generation by Tetrahedral Subdivision

Torben Ægidius Mogensen

DIKU, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen O, Denmark
torbenm@diku.dk

Abstract. We present a method for generating pseudo-random, zoomable planet maps for games and art. The method is based on spatial subdivision using tetrahedrons. This ensures planet maps without discontinuities caused by mapping a flat map onto a sphere.
We compare the method to other map-generator algorithms.

1 Introduction

Computer games have featured (pseudo-)randomly generated maps since at least the 1980s. The best known uses of random maps are in strategy games, such as Sid Meier's Civilization series or Microsoft's Age of Empires series, but random maps have also been used in arcade games and role-playing games. The main advantage of randomly generated maps is replay-ability: The same game can be played on a near infinite number of different maps, providing different challenges each time. Random maps can also contain more detail than it would be realistic to expect from manually created maps.

Nearly all methods for creating pseudo-random maps create a 2D array of values, a so-called *height field*. This can then be rendered in various ways. In the simplest rendering, each altitude is mapped to a colour (like on topographic maps). More complex renderers add shadows or compute a 3D surface from the height field and render this using any 3D rendering algorithm (such as ray tracing). We will in this paper not consider rendering but focus on generation of heights.

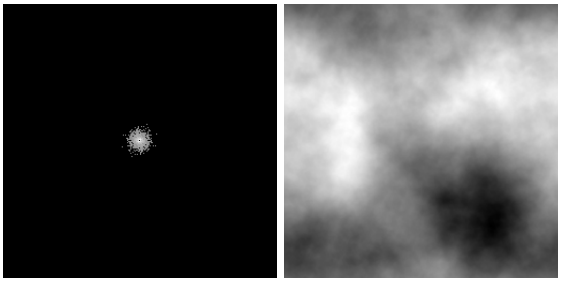
Landscapes are fractal of nature [7]. In practise, this means:

1. Maps are continuous, i.e., points close to each other differ little in altitude.
2. There is a similar degree of detail at all levels of magnification (up to a point), i.e., the map is self-similar.

Landscapes are also irregular, so no systematic construction of self-similar maps will give convincing natural maps. Hence, randomness is needed in the process, and the self-similarity is of a statistical rather than exact nature.

One way to make a statistically self-similar map is by generating noise where the power spectral density is inversely proportional to the frequency. This can be done by generating a large number of points in the frequency domain, distributed according to the power density, and then Fourier-transforming this into the amplitude domain to form a height field [1], as illustrated in figure 1. This, however, has a number of disadvantages:

1. It is computationally expensive.
2. You can not generate a map of a local region significantly faster than the full map.
3. The map is periodic, i.e., it "wraps around" both vertically and horizontally.



Before FFT After FFT

Fig. 1. Fourier-transforming $1/n$ noise

For these reasons, it is more common to use a recursive mid-point displacement method: An initial polygon (or polygon grid) is given height values at its vertices, and is subdivided into smaller polygons where the height values of the new vertices are obtained by adding pseudo-random offsets to the averages of the nearest vertices in the original polygon. By subdividing recursively down to the desired level of detail, a height field can be generated. A common subdivision algorithm is called the *diamond-square* algorithm [5]:

A square grid is given height values at each vertex. This is then subdivided in the following fashion:

Step 1 (diamond) The mid-point of each square is given a height that is an offset from the average of the heights of the four corners. The original square grid points and the new mid-points form a diamond grid.

Step 2 (square) The mid-point of each diamond is given a height that is an offset from the average of the heights of the four corners. The original square grid points plus the points generated in step 1 and 2 now form a square grid with twice the resolution as the original.

Repeat from step 1 until you get the desired resolution.

Figure 2 illustrates this. The new grid points are shown as filled circles, and the dotted lines connect to the old grid points (shown as outlined circles) used to generate the new.

This is computationally cheap, you can generate a regional map to any detail without computing the full map, and you don't have any undesired wrap-around.[1]. Some artefacts may occur in the images due to the use of a regular axis-aligned grid and because the initial vertices are present in the final grid. There are various refinements to

[1] It is still possible to make the map wrap around if this is desired

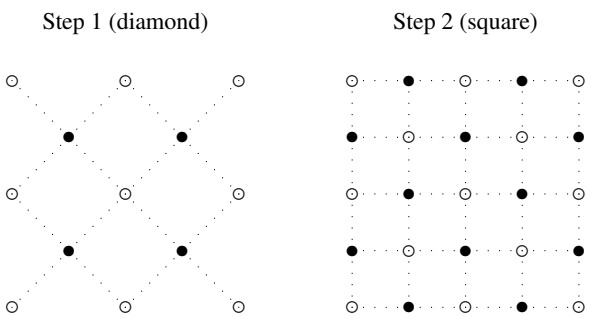


Fig. 2. The diamond-square algorithm

the basic algorithm that reduce some of these effects [8]. More algorithms are described briefly in [3].

Both the Fourier-transform method and the diamond-square algorithm generate height fields: An altitude value assigned to each point on a 2D grid. This means that the generated landscapes can't have overhangs or caves, and since the displacement gets smaller at smaller scales, you don't get vertical cliff faces either. While this limits the type of landscapes that can be created, it is rarely a problem for the typical applications (generating maps for games or landscapes for animation).

The focus of the present article is different: The desire is to make a zoomable map of a full planet. The above methods all generate flat maps, and though they can be made to wrap, this will either create cylindrical or toroidal maps, not spheres. If you wrap these maps on a sphere, you will get distortion and discontinuities (e.g., at the poles). The distortion can be minimised by using *conformal*, i.e., angle-preserving maps, such as stereographic or Mercator projections (in reverse), but these will still give discontinuities in at least one or two points on the sphere. Hence, we will abandon the idea of generating a flat map and wrap this around a sphere, but instead create an everywhere continuous inherently spherical map which (for display purposes) can be projected onto a flat surface.

2 Extending to 3D

To get a full planet map, we need to move to three dimensions. We can easily extend the Fourier-transform to 3D by using a three-dimensional FFT. The result is a cubical grid with a height value at each grid point. To create a spherical map, you embed a sphere in this grid and for each point on the surface of the sphere use a weighted average of values at the nearest grid points.

The altitude values represent displacements of the surface points relative to the centre of the sphere, similar to the way the 2D methods described earlier generate altitudes that represent vertical displacements of points on a 2D grid.

Note that you can embed any surface in the cubical grid, but since worlds tend to be (near) spherical, we will focus on spheres.

Due to the nature of FFT, we can not effectively generate values only for the grid points that are close to the spherical surface, so we generate a lot of values that we never use.

The diamond-square algorithm can also be modified to 3D using a cubical grid and three steps to get to a finer cubical grid:

Step 1 Find a value for the middle of each cube using the values at its eight corners.
Step 2 Find a value for the middle of each face of the cubes using the values at the four corners of the face and at the midpoints (from step 1) of the two adjoining cubes.
Step 3 Find a value for the middle of each edge of the cubes using the values at its two ends and at the midpoints (from step 2) of the four adjoining faces.
Repeat from step 1 until the desired resolution is obtained.

This, like the Fourier-based method, computes all points in a 3D grid, while we in the end use only small subset of these. It would be natural to subdivide only cubes that contain parts of the spherical surface, which would drastically reduce the required time. However, steps 2 and 3 use information from adjoining cubes, so we can't just omit these. It is possible to modify the algorithms so it in step 2 only uses the four corners of the face and not the adjoining cubes and step 3 so it uses only the two end-points of the edge. This will, however, reduce the quality of the maps.

Additionally, just like the use of a regular square grid in the 2D diamond-square algorithm can produce artefacts that make features orient to the grid, the use of a regular cubical grid in the 3D extension can produce artefacts that make features orient to the 3D grid. This means that, even after rotating the spherical map, it is often possible to visually identify the orientation of the grid.

2.1 Spatial subdivision

We will now present a method that attempts to address these problems.

Let us say that we wanted to find the height value of just a single point on the surface of a sphere. We can do this using the following algorithm:

1. Embed the sphere inside a polyhedron, where each vertex is assigned a height value
2. Cut the polyhedron into two smaller polyhedra, generating height values for the new vertices from their neighbouring old vertices.
3. Select the polyhedron in which the desired point is located
4. Repeat from step 2 until the polyhedron is small enough
5. Use the average height values of the vertices of the final polyhedron as the height value of the desired point.

When you render a picture of a planet, you find the visible points on the planet surface and apply the above algorithm to each point to find its height value. For example, in ray tracing each ray that hits the surface will define a visible point.

A question is how small is "small enough" in the above algorithm? Basically, you would not want two neighbouring pixels in the image to get the same value because you stop subdivision at the same polyhedron for both pixels. Since the volume of the

polyhedron is halved in each step, three steps will double the resolution of the details. So a rough estimate is that a 1000×1000 image will require 30 subdivisions for each pixel (since $2^{10} = 1024$), but this assumes that all pixels represent equal areas on the sphere, which is not typically the case. A better solution is to project the corners of the pixel onto the sphere and stop only when at most one of the projected corner points is inside the polyhedron that holds the desired point (which is the projection of the middle of the pixel). In all but the most extreme cases, this will ensure that neighbouring pixels end in different polyhedra. A somewhat simpler and almost as good method is to estimate the pixel's projected size on the sphere and stop when the polyhedron diameter is smaller than this size. If a standard map projection is used, it is not difficult to make a good estimate of projected pixel diameter without actually projecting the pixel corners to the sphere. For example, in the Mercator projection, the area of a pixel projected to the surface at point (x,y,z) is $\sqrt{1-y^2}$ times the area if a pixel projected to the equator.[2]

The next question is which type of polyhedron to use and how to subdivide it. The most obvious choice is to use a rectangular box and cut it into two boxes. Indeed, if you choose the proportions correctly,[3] the two new boxes will have the same relative proportions as the original, much like A4 paper is cut into two A5 papers that have the same relative side-lengths. An earlier version of the algorithm presented in this paper did, indeed, use such rectangular boxes. However, using rectangular boxes has several disadvantages:

1. This is equivalent to using a regular axis-aligned grid, which can give artefacts.
2. Four new vertices are created at each subdivision. We would like fewer new vertices at each step to keep the number of calculations per step low.

Our solution is to use a tetrahedron. A tetrahedron can be divided into two tetrahedra by cutting along a plane defined by two vertices and a point on the opposing line, as illustrated in figure 3.

There are no possible proportions of a tetrahedron that will make such a cut produce two tetrahedra with the same relative proportions as the original (even if we allow the two smaller tetrahedra to be of different size). But this is actually an advantage: By keeping the tetrahedra irregular and different, we avoid the artefacts created by using axis-aligned boxes.

We will use a tetrahedron where all edges are of different lengths and we will always cut the longest edge at its midpoint. The algorithm requires that there is always a unique longest edge to cut. We have no mathematical proof that this is the case with the chosen initial side lengths, but we have tested this uniqueness property down to a large number of subdivisions (by making a full-world map of very high resolution and a large number of detail maps with large magnification) and found no cases where there is no unique longest edge.

With tetrahedron cuts, we now create only one new vertex at each subdivision, so the calculation of new vertices has been cut by four compared to using rectangular boxes. However, extra time is needed to identify the longest edge, and the calculations

[2] If the sphere has radius 1 and the polar axis is the y-axis.

[3] $1 \times 2^{\frac{1}{3}} \times 2^{\frac{2}{3}}$

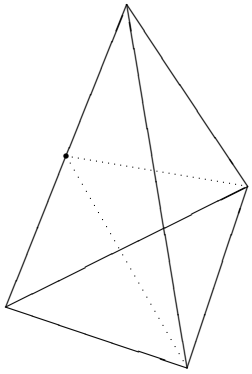


Fig. 3. Cutting a tetrahedron in two

required to identify in which tetrahedron the goal point is located are more complex, so there is no overall reduction in computational cost. Hence, the main advantage of using tetrahedra is reduction of artefacts caused by using a regular grid.

A more detailed algorithm description can be found in figure 4. Notes:

- The function $random(s)$ produces a pseudo-random value from a seed s. The same seed will always give the same result. Note that $random()$ is seeded by the average of the seeds of the end points of the edge, so the result is independent of the other vertices and of the order of v_1 and v_2. Since the same edge is shared between several tetrahedrons, all of which may cut this edge, we need to cut the edge in the same way regardless of which tetrahedron it is considered a part of.
- The function $offset(s, l, a_1, a_2)$ provides an offset that depends on the seed s, the length l of the edge and the altitude values a_1 and a_2 at the end-points of the edge. In the simplest case, the offset is simply proportional to l, but more realistic landscapes can be made by making the offset proportional to l^q, where $q < 1$ (since altitudes tend to differ relatively more over short distances than over longer distances), and by adding in a contribution that is proportional to the altitude difference $|a_1 - a_2|$ (since steep areas tend to be less even than flatter areas). Other variations are possible, such as taking the distance from sea-level into account (since areas at higher altitude tend to be more rugged than low-altitude areas).
- The altitude values at the vertices of the initial tetrahedron will affect the altitudes on the surface of the sphere, but since the vertices are outside the sphere, they can not determine exact altitudes of any points on the surface. Hence, while setting these four altitudes to sea level will typically give planet maps with roughly equal parts land and sea, the amount of land and sea can vary greatly if different seeds are used.
- A normal vector of the landscape at a point on the sphere surface can be estimated from the altitude values and coordinates of the four vertices in the final tetrahedron. This can be used for shadow effects and bump maps.

Input: A target point p and four vertices $v_1, \ldots, v_4$ of a tetrahedron, each with the following information:

- Coordinates (x_i, y_i, z_i)
- Seed for pseudo-random number generator s_i
- Altitude value a_i

Repeat:

1. Re-order vertices $v_1, \ldots, v_4$ so the longest edge of the tetrahedron is between v_1 and v_2, i.e., such that $(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2$ is maximised.
2. Define new vertex v_m by

$$\begin{aligned}(x_m, y_m, z_m) &= ((x_1 + x_2)/2, (y_1 + y_2)/2, (z_1 + z_2)/2) \\ l &= \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2} \\ s_m &= \mathit{random}((s_1 + s_2)/2) \\ a_m &= (a_1 + a_2)/2 + \mathit{offset}(s_m, l, a_1, a_2)\end{aligned}$$

3. If p is contained in the tetrahedron defined by the four vertices v_m, v_1, v_3 and v_4, set $v_2 = v_m$. Otherwise, set $v_1 = v_m$.

Until: l is small enough
Return: $(a_1 + a_2 + a_3 + a_4)/4$

Fig. 4. Tetrahedron subdivision

Step 1 (reordering vertices) uses about 50% of the time, 30% is spent in step 2 (calculation of new vertex) and 20% in step 3 (determining in which sub-tetrahedron the point is located).

An optimisation is possible based on the observation that adjacent pixels in a rendered image are likely to correspond to close points on the sphere, so the first many subdivisions will be the same. The idea is that, after a number of subdivisions, we store the vertices of the current tetrahedron before we continue. When we repeat the algorithm for the next pixel, we check if the projected point is inside the stored tetrahedron. If it is, we start subdivision from the stored tetrahedron instead of from the top level. For detailed maps of smaller regions of a planet, this can dramatically reduce the overall calculation time.

3 Implementation and uses

An implementation in the programming language C of the tetrahedron subdivision algorithm can be found at `http://www.diku.dk/~torbenm/`. For rendering, a number of standard map projections such as Mercator, Mollweide and Gnomonic are provided. The rendering is otherwise fairly primitive: The colour of a point on the surface depends on the altitude at and the latitude of this point, and shadow effects based on

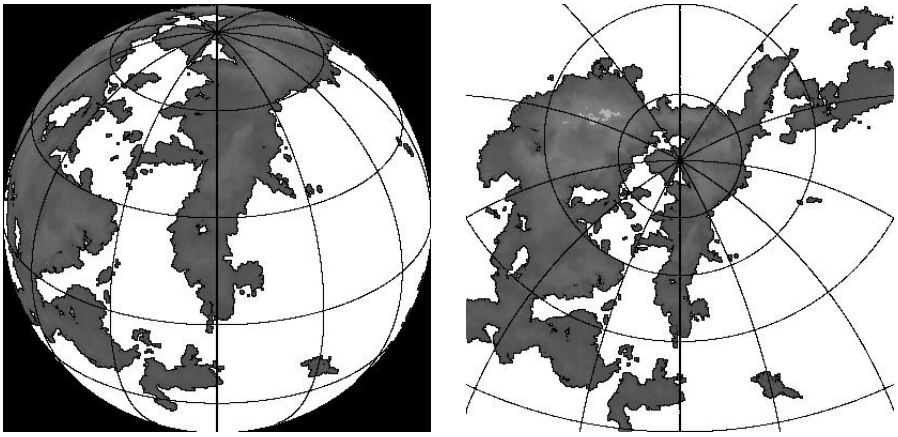


Fig. 5. Maps using orthographic and stereographic projections

the estimated surface normal can be added. The colouring scheme can be changed by modifying a palette file, and longitude/latitude lines can be added.

Example outputs (rendered in black and white) are shown in figure 5.

Figure 6 shows zooming in on a detail of the map. Each step uses four times the resolution of the previous step, so the last picture is 16384 times as detailed as the first.

While the above program can be used on its own to generate maps for use in role-playing games and such, the algorithm has also been used in open-source computer games, including an upcoming version of a Russian space game [10], where it is one of several planet texture generators. Sample planets rendered by the game engine is shown in figure 7. The clouds and craters are added by additional texture generators.

4 Comparison to other methods

We have already mentioned the diamond-square algorithm and Fourier-transform based methods, and noted that these are designed for flat maps and will yield discontinuities if these maps are projected onto spheres. Some tricks are often used to make this less noticeable:

- Force the edges of the map to have certain pre-specified values (such as deep water), so they can meet without discontinuities.
- Make the map wrap around horizontally to form a cylinder, map this to a sphere and add artificial "ice caps" to hide discontinuities at the poles. Fourier-based maps automatically wrap around, and diamond-square algorithm can be made to do so fairly easily.

Note that wrapping around both vertically and horizontally makes a torus, so this can't solve the discontinuity issue on a sphere.

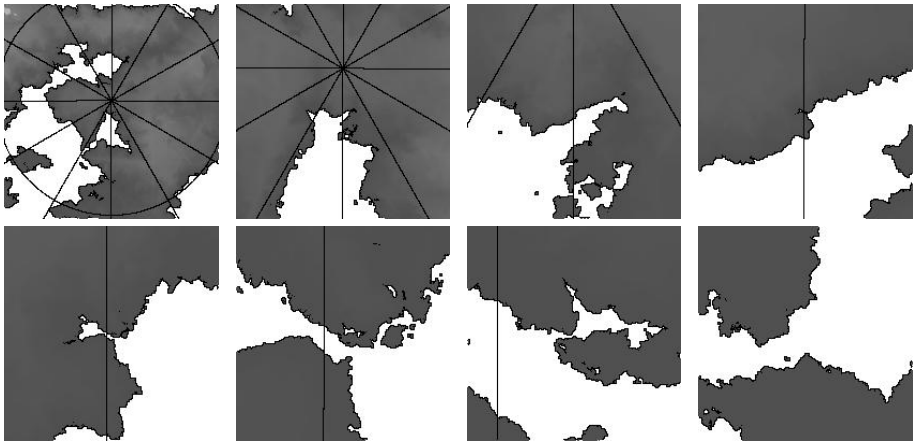


Fig. 6. Zooming in

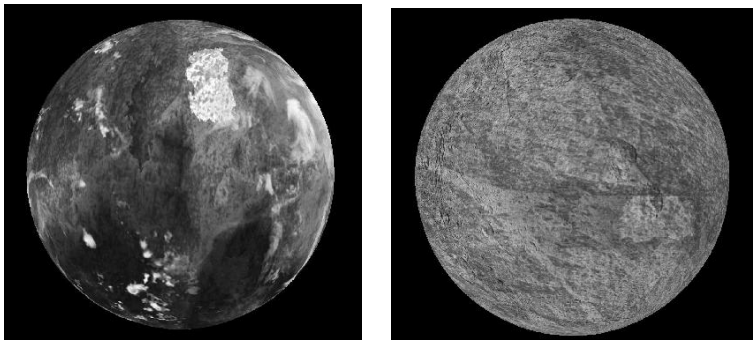


Fig. 7. Planet views from game prototype

Other map generators try to simulate plate tectonics [4]. These, also, are inherently full-map methods with no way of generating a region map significantly cheaper than the global map. The programs I have seen all use a rectangular grid, so they also have the problems with mapping to a sphere mentioned above.

The author is aware of one other map-generation method that inherently works on a spherical surface [11, 2, 9]:

1. Select a random great circle on the sphere
2. Raise the altitude for the hemisphere on one side of this great circle by a small amount and lower the altitude for the hemisphere on the other side by the same amount.
3. Repeat this a large number of times.

The method is illustrated in figure 8.

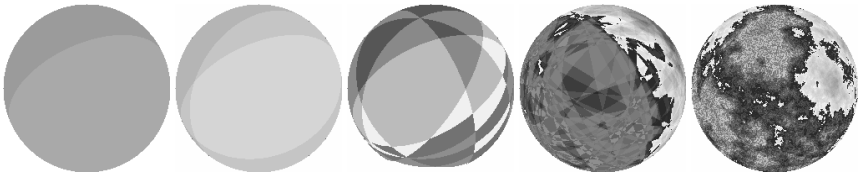


Fig. 8. Great-circle algorithm

The great-circle algorithm is normally used in a context where the full map is generated in a rectangle that is a projection of the sphere, mapping the great circles to curves in the projection and raising the altitudes on one side of the curve and lowering it on the other side. It is, however, fairly easy to modify this to find the altitude of a single surface point:

1. Start with altitude 0
2. Select a random great circle on the sphere
3. If the selected point is on one side of this great circle, raise the altitude, otherwise lower it.
4. Repeat from step 2 N times.
5. Return altitude

If the same initial seed for the pseudo-random number generator is used at every point, you get a consistent result.

In the standard version of the algorithm, the cuts define discontinuities, since all points on one side of the hemisphere are raised by the same amount and all all points on the other side lowered by the same. If this amount is small, this matters little, but at high magnifications, it will be visible. It is easy to modify the algorithm, so the change is more gradual, but that makes the terrain smooth at high magnifications unless you have many cuts, so this doesn't lower the required number of cuts significantly. More seriously, whenever a point on the surface is raised, the point opposite of it on the sphere is lowered, so the maps will have a kind of mirror symmetry: Islands on one side of the planet will have mirror-image lakes on the opposite side, as seen in figure 9, where the right map is centred at the opposite point as the left map and then mirrored. If you raise the waterline above 0 altitude, this is not immediately noticeable, but you can't have land on two opposing points on the sphere. Another way to reduce these artefacts is to make the cuts at smaller circles rather than great circles. There will still be a greater probability for water opposite land than one would expect, but there is no strict mirror-image effect.

To compare running times and results, the great-circle algorithm and the tetrahedron-subdivision algorithm were both run to produce 400×400 pixel full-world and zoomed ($10\times$) maps using the orthogonal projection. The two programs are identical except for the algorithm used to generate altitudes for points on the sphere. The great-circle algorithms pregenerates and stores the cuts, so they don't have to be regenerated for each point. The tetrahedron-subdivision algorithm uses an optimisation where the tetrahe-

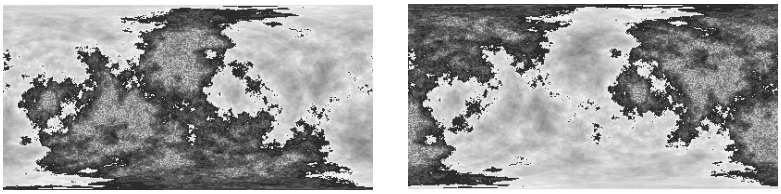


Fig. 9. Mirror effect

dron after k subdivisions is stored and reused for the next point if it is within the same tetrahedron.

The number of cuts used for the great-circle algorithm depends on the level of detail required. For the full-world map 2000 great circle cuts suffice, but artefacts are obvious when zooming in. At 10× zoom, ten times as many cuts are needed. This changes the appearance, as all cuts have equal contribution, so even maps at low resolution have to be generated with a high number of cuts if *any* map of the planet is needed at high resolution.

The results can be seen in figure 10.

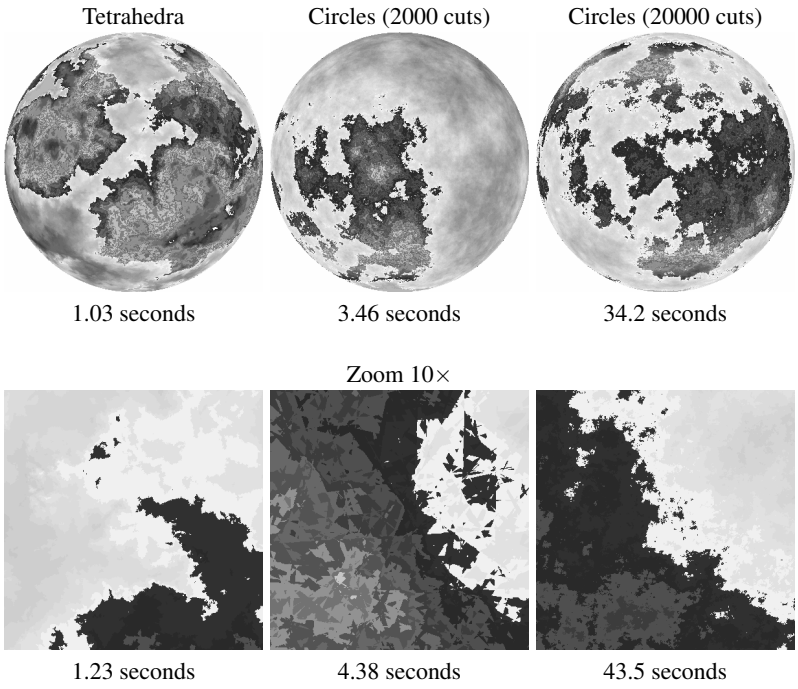
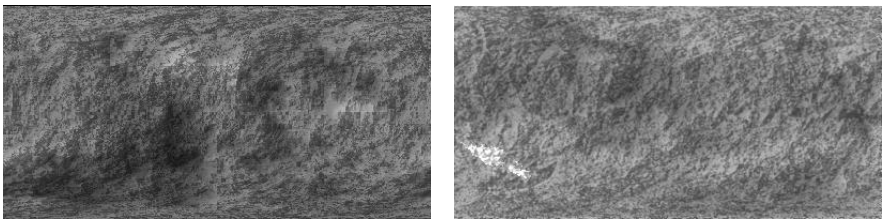


Fig. 10. Comparing great-circle and tetrahedron-subdivision algorithms

For a full planet map of $n \times n$ pixels, the great-circle algorithm requires time proportional to n^3, as the required number of cuts is proportional to the resolution. In contrast, the tetrahedron subdivision algorithm requires time proportional to $n^2 \log(n)$, as the required number of subdivisions is proportional to the logarithm of the resolution. To generate a map of a detail of the planet at zoom rate z, the great-circle algorithm needs z time more cuts and, hence, z times more time. The subdivision algorithm needs $\log(z)$ more subdivisions, so it needs only $\log(z)$ times more time.

To compare tetrahedron subdivision with recursive subdivision of a rectangular box, figure 11 shows two maps using a "bump map" shading that exaggerates feature details. On the map generated with rectangular box subdivision, there are clear horizontal artefacts at the mid-right and top-middle parts of the map and a noticeable vertical artefact near the middle, while the map on the right doesn't show any clear alignment of features. Subdividing a rectangular box is, in fact, faster than tetrahedron subdivision. This is because it doesn't need to identify the longest edge at each step and because it is simpler to determine in which half-box a point is located.



Rectangular box subdivision: 0.25 seconds

Tetrahedron subdivision: 0.56 seconds

Fig. 11. Comparing rectangular-box and tetrahedron subdivision

5 Conclusion

The tetrahedron subdivision algorithm provides fast and zoomable generation of spherical planet maps. Unlike most other planet map generators, there are no discontinuities nor distortion caused by projecting a flat map onto a sphere. Also, the choice of an inherently irregular tetrahedron subdivision avoids grid-aligned artefacts.

Since single points can be sampled, the method is well suited in contexts where varying levels of detail are needed in the same picture, such as landscape views where the foreground needs more detail than the background: You simply sample points at the required density. Single-point sampling is also well-suited to ray tracing, since you can sample exactly the points that are intersected by rays, thus wasting no time generating details that are not visible.

A limitation with this (and most other fractal map generators) is that there is no erosion, no rivers, no silting and so on – the landscape is everywhere rough, as if newly created by tectonic faulting. Some map generators add erosion, rivers and sedimentation in a post-processing phase, but since these are not local phenomena, you need to

perform the process on the full map, which negates the zoomability and variable level of detail. As an example, the map generator Wilbur [6] allows non-local post-processing like flow incision and fill basins to emulate erosion and sedimentation.

There is no obvious solution to the locality problem, though various tricks can be employed to make the map appear more natural, such as making the roughness of terrain depend on the altitude, so areas under or near sea level are smoother than areas of high altitude. Wilbur [6] can remap the generated altitudes according to a user-specified mapping. A standard mapping compresses lower altitudes while higher altitudes are stretched, and a sudden drop in sub-sea altitudes is made to emulate continental shelves.

References

1. Poul Bourke. Frequency synthesis of landscapes (and clouds). http://local.wasp.uwa.edu.au/~pbourke/fractals/noise, 1997.
2. Poul Bourke. Modelling fake planets. http://local.wasp.uwa.edu.au/~pbourke/fractals/noise, 2000.
3. C. Burke. Generating terrain. http://www.geocities.com/Area51/6902/terrain.html, 1996.
4. C. Burke. Plate tectonics. http://www.geocities.com/Area51/6902/t_plate.html, 1996.
5. A. Fournier, D. Fussel, and L. Carpenter. Computer rendering of stochastic models. *Communications of the ACM*, 25(6):371–384, June 1982.
6. J.S. Layton. Wilbur. http://www.ridgecrest.ca.us/~jslayton/wilbur.html, 2009.
7. Benoît Mandelbrot. *The Fractal Geometry of Nature*. W.H. Freeman & Co, 1982.
8. Gavin S. P. Miller. The definition and rendering of terrain maps. *Computer Graphics*, 20(4):39–48, 1986.
9. John Olsson. Fractal worldmap generator. http://www.lysator.liu.se/~johol/fwmg/fwmg.html, 2004.
10. Oleg Petrov. Babylon 5; i've found her. http://ifh.babylonfive.ru/, 2008.
11. R. P. Voss. Random fractal forgeries. *Fundamental Algorithms for Computer Graphics*, 17:805–835, 1985.