Rendering and physical simulation of planets by icosahedral subdivision

Christian Cosgrove

September 6, 2015

1 Introduction

The problem of rendering terrain procedurally has been explored thoroughly in the field of computer graphics, and many continuous level-of-detail (CLOD) methods have been proposed. For rectangular domains, CLOD has been achieved through the rectangular subdivision of terrain, often by using partitioning structures like quadtrees[1]. For spherical domains, it is possible to adapt this method by parameterizing the sphere by six rectangular domains; however, this suffers significant distortion difficulties near the vertices of the cube.

Although other[2] methods of representing spherical terrain exist, this documentation presents a method that recursively subdivides an icosahedron to generate arbitrary terrain detail. Attention will be focused on reducing distortion, a problem that most mappings suffer to a greater or lesser extent, and handling floating-point limitations. As the algorithm produces only fictive planets, we do not generate planets on Earth's scale, but we do seek to maximize the level of detail of the engine.

In addition to CLOD, we describe other noteworthy features of the terrain rendering implementation, namely a sorting procedure for vertex indexing and an N-body simulator for simulation of fictional planetary interactions.

[1] Pajarola, R. (1998) Large scale terrain visualization using the restricted quadtree triangulation

[2] Clasen, C., and Hege, H. (2006) Terrain rendering using spherical clipmaps

2 Icosahedral subdivision

2.1 Mapping scheme

When considering the ways of representing spherical terrain data, a polyhedral one is attractive because of its simple implementation and extensibility. By selecting a simple base mesh with a finite number of symmetries (as opposed to the infinite rotational symmetry of a sphere), dynamic subdivision (and therefore CLOD) is simple. One of the simplest polyhedral base meshes is the cube, but mapping a cube to a sphere results in significant distortion. The next apparent base mesh is the icosahedron, a regular polyhedron with 20 triangular faces. Unlike the cube, there is no obvious way of subdividing an icosahedron with rectangular quadtrees, but other methods of subdivision are possible.

One promising yet simple method of subdividing an icosahedron is to recursively form triangles by connecting the midpoints of the sides of each face (tesselation by triangular tiling). In this system, each level of subdivision introduces four new congruent faces. This subdivision method, commonly known as the geodesic grid, produces less distortion than the cubic base mesh example (Figure **??**).

To justify using an icosahedron over a cube, we provide a quantitative measure of the distortion of a polyhedron-to-sphere mapping. One measure of a polyhedron's distortion is the ratio between its inradius and its circumradius. The closer this quantity is to 1, the more spherical the polyhedron and the less distortion that will be present. To derive a meaningful expression for this quantity, we begin with known formulas for the inradius and circumradius of Platonic solids. Coxeter[3] gives the following formulas for the circumradius and inradius of a Platonic solid $\{p, q\}$:

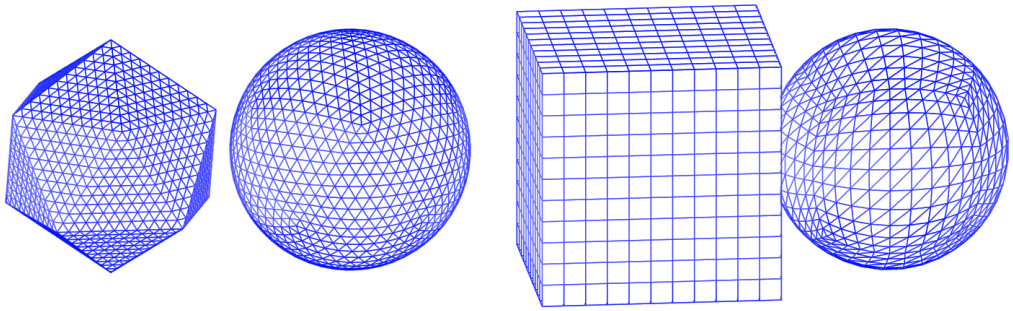
$$r_{\text{circum}} = l \sin\frac{\pi}{q} \csc\frac{\pi}{h}$$

$$r_{\text{in}} = l \cot\frac{\pi}{p} \cos\frac{\pi}{q} \csc\frac{\pi}{h}$$

The constants l and h appear symmetrically in these formulas, so they cancel out in the final expression. The ratio $r_{\text{in}}/r_{\text{circum}}$ is:

$$\cot\left(\frac{\pi}{p}\right) \cot\left(\frac{\pi}{q}\right),$$

[3]Coxeter, H.S.M. (1948) *Regular Polytopes*, p. 21



(a) Icosahedral base mesh (b) Cubic base mesh

Figure 1: Comparison of mapping an icosahedron to a sphere (geodesic grid) and a cube to a sphere.

where q is the number of edges connected to a vertex, and p is the number of edges per face. This applies to Platonic solids, where the number of edges meeting at a vertex and the number of faces meeting at a vertex are uniform throughout the polyhedron. For an icosahedron (Schäfli symbol $\{p, q\} = \{3, 5\}$), the distortion is $\frac{1}{\sqrt{15-6\sqrt{5}}} \approx 0.795$, while for a cube ($\{p, q\} = \{3, 4\}$), it is $\frac{1}{\sqrt{3}} \approx 0.577$. With its reduced distortion, the icosahedron base mesh proves to be a compromise between complexity and accuracy for representing a procedural planet. This choice of base mesh is known as the *geodesic grid*. This relatively low amount of distortion makes the geodesic grid an attractive medium for numerical simulation, as well[4].

Table 1: Values of the inradius-circumradius ratio for various Platonic solids

polyhedron	inradius-circumradius ratio	
tetrahedron	$1/3$	≈ 0.33
cube	$1/\sqrt{3}$	≈ 0.56
octahedron	$1/\sqrt{3}$	≈ 0.56
icosahedron	$1/\sqrt{15-6\sqrt{5}}$	≈ 0.80
dodecahedron	$1/\sqrt{15-6\sqrt{5}}$	≈ 0.80

[4]Williamson, D. (1968) Integration of the barotropic vorticity equation on a spherical geodesic grid

2.2 Terrain subdivision

This proposed method of planet rendering uses recursive subdivision of the icosahedral base mesh to achieve CLOD. The number of levels of subdivision is a function of the camera's distance from the surface of the planet. Every face of the generated mesh undergoes a recursive procedure: it is checked to be 1. close enough to the camera to be subdivided, and 2. far enough from the camera for it (and its neighbors) to be combined into a larger face. If either of these is satisfied, a similar procedure is applied to the children (in the case of subdivision) or parent (in the case of combination) of the given face.

Algorithm 1 Subdivision and combination procedures

procedure TRYSUBDIVIDE(face, l, $\vec{p}$) ▷ face is a member of a tree; l is the current level of subdivision; $\vec{p}$ is the camera position
  if $|\text{face}.\vec{c} - \vec{p}| \leq 2^{-l-s}$ **then** ▷ Continue subdividing if distance is less than exponential factor
    SUBDIVIDE(face) ▷ Generates child faces
    for all child ∈ face.children **do**
      TRYSUBDIVIDE(child,$l + 1$, $\vec{p}$)
    end for
  end if
end procedure

procedure TRYCOMBINE(face, l, $\vec{p}$)
  if $|\text{face}.\vec{c} - \vec{p}| \geq 2^{-l-s+1}$ **then** ▷ Continue combining if distance is greater than exponential factor
    COMBINE(face) ▷ Recursively removes child faces
    if $l > 0$ **then**
      TRYCOMBINE(face.parent, $l - 1$, $\vec{p}$)
    end if
  end if
end procedure

The distance threshold for either subdividing or combining a mesh is an exponential function of the current level of detail l: when the camera gets twice as close to the planet's surface, a new level of detail is generated. This choice of threshold function results in smooth, uniform geometry once projected onto the viewport.

The additional factor of 2^1 in the distance check in TRYCOMBINE(face, $l, \vec{p}$) ensures that geometry is not subdivided and combined simultaneously.

2.3 Terrain noise function

This planet rendering method uses a fractal noise algorithm, chosen to fit the icosahedral tree structure used to represent vertex data. After every recursive subdivision, a displacement is applied according to the below-displayed algorithm.

Algorithm 2 Recursive terrain noise function

function NOISEMIDPOINT(l, $\vec{v}_1$, $\vec{v}_2$, $\vec{p}_1$, $\vec{p}_2$) ▷ v_1 and v_2 are 3D Cartesian vectors; p_1 and p_2 are 2D vectors containing polar and azimuthal components of spherical coordinates

$\vec{m}_s \leftarrow \frac{\vec{p}_1+\vec{p}_2}{2}$ ▷ spherical coordinates of midpoint are average

$\vec{n}_1 \leftarrow \frac{\vec{v}_1}{|\vec{v}_1|}$ ▷ normalize $\vec{v}_1$

$\vec{n}_2 \leftarrow \frac{\vec{v}_2}{|\vec{v}_2|}$ ▷ normalize $\vec{v}_1$

$\vec{m}_c \leftarrow \frac{\vec{n}_1+\vec{n}_2}{|\vec{n}_1+\vec{n}_2|}$ ▷ $\vec{m}_c$: normalized vector sum of $\vec{n}$s

$\vec{m}_c \leftarrow \vec{m}_c$*TERRAINNOISE($\vec{m}_s$) $*2^{-l} * l^r$ ▷ scale normalized $\vec{m}_c$

$\vec{m}_c \leftarrow \vec{m}_c * \frac{|\vec{v}_1|+|\vec{v}_2|}{2}$ ▷ scale $\vec{m}_c$ by the noise values for neighboring vertices

return $\vec{m}_c, \vec{m}_s$ ▷ return Cartesian and spherical midpoints

end function

Each fractal noise layer is decreased in scale with further subdivision. As shown, the subdivided midpoint is scaled by an exponential factor of 2^{-l}, which scales down higher-detail noise. A power-law factor (l^r) is added for additional large-scale detail; this factor contains terrain regularity constant r, which determines how quickly the noise size falls off (increasing it enlarges large-scale terrain).

The noise function used in the algorithm TERRAINNOISE($\vec{m}$) is a deterministic random number generator (RNG) depending on the polar and azimuthal components of the vertex's spherical coordinates. A high-frequency sinusoid with period comparable to the machine epsilon was used as an RNG. By storing vertices' polar coordinates in double precision, truncation error at higher levels of detail can be partially avoided. The polar coordinates of a midpoint vertex are simply the average of the polar coordinates of the two parent vertices. Thus, we avoid repeated vector normalization (which would be necessary in the Cartesian representation). This reduces truncation error, allowing exploration of smaller features

on the planet surface.

3 Interfacing with the GPU

For the most part, procedural generation in this method is performed on the CPU. In order to render anything on the display device, it is vital that vertex information be sent to the GPU in the proper format.

3.1 Vertex indexing

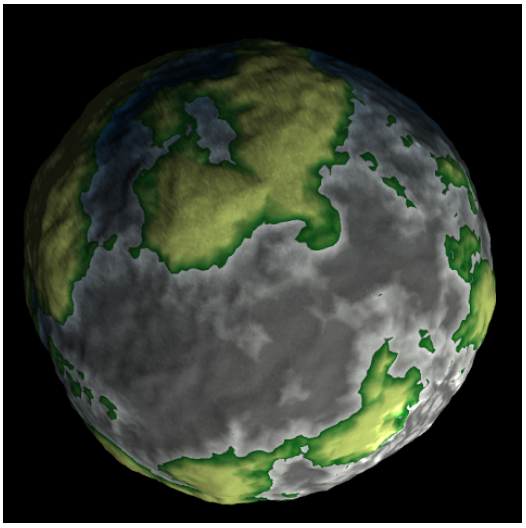
A common practice in the field of computer graphics is vertex indexing. This allows one to reduce the amount of data sent to the GPU to render a given number of triangles, and it is usually achieved through an index buffer. In a vertex indexing scheme, a triangle is represented by three integers that refer to vertices in a vertex buffer. Because shared vertices are not duplicated in the vertex buffer and integers are typically smaller than vertex structures in memory, this reduces the GPU bandwidth necessary.

Removing duplicate vertices yields aesthetic benefits, as well. Many lighting models involve taking the dot product of a light vector with a face's normal vector. When duplicate vertices contain different normals, sharp creases in lighting become apparent.

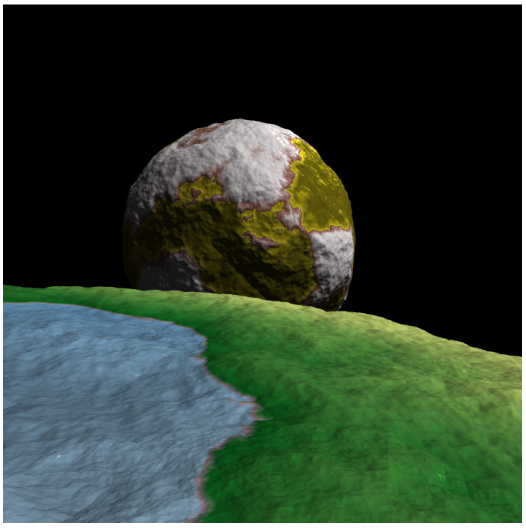
Even though for other problems it can be simple, vertex indexing is a nontrivial problem in this method. The quadtree structure representation of faces in the planet mesh makes determining and excluding duplicated vertices in better than $O(1)$ difficult. Thus, a sorting procedure is used to organize vertices based on spatial proximity (after sorting, common vertices are adjacent in the sorted vertex buffer).

Sorting was performed by feeding the vertex's spherical polar coordinates into a simple hash chosen for the problem. This hash $H(\theta, \phi)$ takes the sum of the polar angle θ and a relatively large multiple of the azimuthal angle ϕ. Spherical coordinates are stored in the vertex structure as double precision floating-point numbers. Since $0 \geq \theta \leq \pi$, $H(\theta, \phi) = \theta + k\phi$ for $k \gg \pi$. Making k too large, however, should be avoided because it can lead to floating point error.

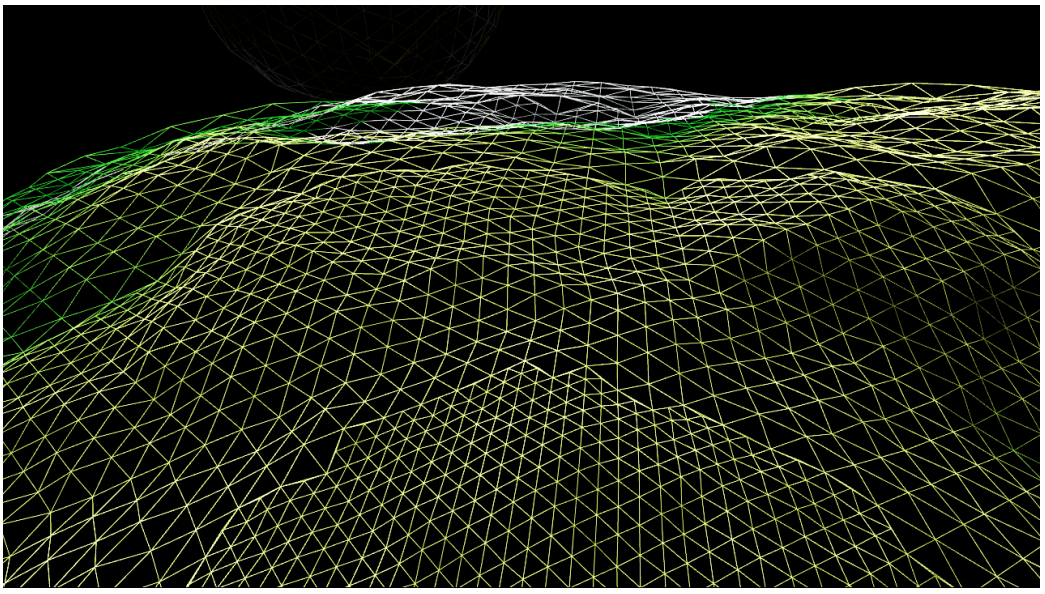
Though the global behavior of the sorted vertex buffer does not matter, this procedure places duplicate vertices (equal by floating-point comparison or simply very close to each other) next to each other. The final sorted array can be iterated



(a) Image 1



(b) Image 2



(c) A low resolution wireframe render demonstrating CLOD.

through, removing duplicate vertices and creating common vertex references in the index buffer.

For this method, two possible sorting procedures were attempted. Both introsort and heapsort possess the same average time complexity, $O(n \log n)$. In this implementation, the two sorting algorithms differed in performance by a multiplicative constant. In general, the implemented heapsort algorithm performed worse than introsort. This is to be expected, given that introsort is a hybrid sorting algorithm that uses quicksort initially, and switches to heapsort when the array size exceeds a certain value.

Algorithm 3 Sorting of vertices

function VERTEXCOMPARISON(face1, face2)
$k_1 \leftarrow \begin{bmatrix} 1 \\ 4\pi \end{bmatrix} \cdot p_1$ ▷ a simple hash function that multiplies azimuthal angle by a large factor.
$k_2 \leftarrow \begin{bmatrix} 1 \\ 4\pi \end{bmatrix} \cdot p_2$
return $k_1 > k_2$
end function
procedure UPDATEMESH(vertices) ▷ vertices is a list of vertex structures generated recursively from the face tree
INTROSORT(vertices, VERTEXCOMPARISON)
maxneighbors $\leftarrow 6$
$\vec{n} \leftarrow \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$
for $i \leftarrow 1$ to LENGTH(vertices) **do**
for $j \leftarrow$ MAX(0, $i -$ maxneighbors) to MIN(LENGTH(vertices), $i +$ maxneighbors) **do**
if $|\text{vertices}[j] - \text{vertices}[i]| \leq \epsilon$ **then**
$\vec{n} \leftarrow \vec{n} + \text{GETNORMAL}(\text{vertices}[j])$
end if
end for
end for
$\vec{n} \leftarrow \vec{n}/|\vec{n}|$ ▷ ensure $\vec{n}$ is unit length
end procedure

Notice that in the grouping procedure the maxneighbors variable is set to six.

This is because in the mesh, a vertex can be neighbored by no more than six other vertices, so one must only check the six vertices before and after the given vertex in the sorted array. This part of the algorithm has only linear complexity.

4 Multiple planets

In this implementation, it is very easy to extend this rendering method to systems of multiple celestial objects. Since vertices remain static relative to the planet origins, it makes sense not to manipulate the vertex positions on the CPU, but rather on the GPU in a shader. More specifically, each planet can be drawn in a single API call, each time changing the model-view matrix sent to the GPU. In this way, planets can be animated in real-time so long as the relative positions of all of the vertices within a planet does not change.

4.1 Multithreading

In order to optimize real-time generation performance, multithreading should be employed. Fortunately, since all of the planets are decoupled, geometry construction of each planet can be performed on separate threads. In systems of multiple planets, this can improve performance significantly.

Using multithreading to accelerate the generation of a single planet is not as trivial as it is for multiple distinct planets. Because planet vertices are stored in a quadtree partitioned into groups of dramatically different size, there is no obvious way to parallelize the generation algorithm. One could try assigning a thread to generate each face of the icosahedron base mesh, but the work would be unequally distributed. Typically, the camera is close to only 1-3 icosahedron faces.

4.2 N-body simulation

A desired extension to the existing engine is to simulate gravitational between planets. This has been achieved through Velocity-Verlet integration of Newton's Law of Gravitation. An important feature of the N-body algorithm is that it use a *symplectic* integrator, i.e. one that preserves the energy of the system.

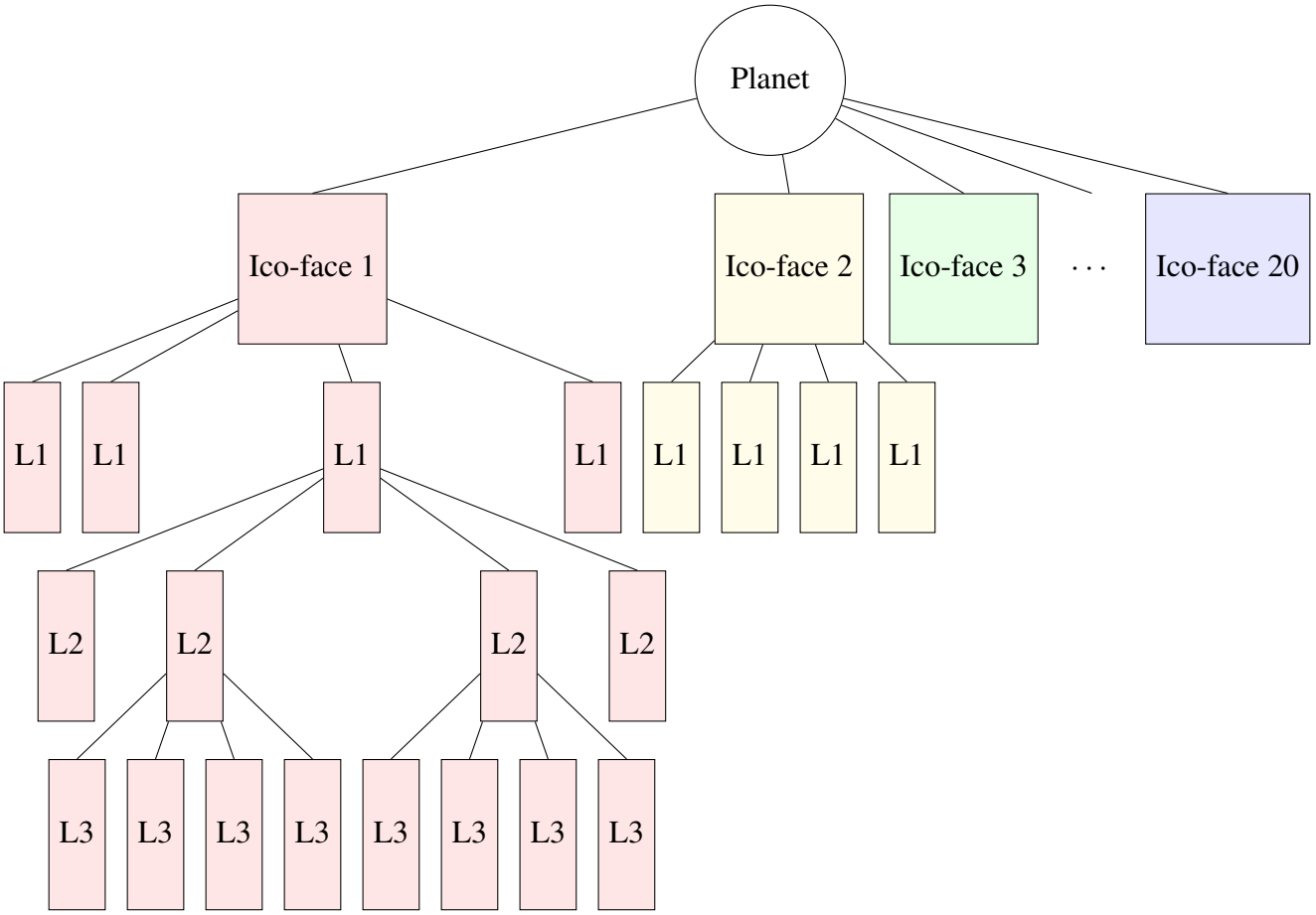


Figure 3: A typical example of the planet geometry quadtree. The maximum depth of the tree can be large, but the average depth of each icosahedron face is small. This illustrates the intra-planet multithreading problem: it is possible to divide work between threads at the root (shown by colors), but only a few threads will do work.