Moonscape:
Realtime Procedural Planetary Terrain Rendering Using
Quadtrees and Discrete Levels of Detail

Nathanael Deraney
Yale University
CPSC 490b
Advisor: Holly Rushmeier

1 May 2013

Abstract
Rendering of the spherical terrain of planets offers significant challenges, including curvature, wrapping, and inability to use static heightmaps given extreme variation in levels of detail required. I develop and implement an algorithm that uses a quadtree with fixed seamless meshes for dynamic level of detail. Detail level changes at most one level between nodes and so only sixteen meshes need be used for seamless matching of vertices. The algorithm offloads as much as possible to the GPU including transform and scale to planet parameters, spherical mapping, calculation of fractal Brownian motion noise for displacement and normal generation, the latter being rendered to texture for caching on node creation. After detailing the implementation of this algorithm and issues encountered I offer ideas for improvement including using the GPU to precalculate vertex information and store that as well as normals in textures per node, batch rendering and combining node textures into texture atlases, and copy height information from parent nodes so one only needs add one new level of noise per tree level.
The Problem

In this project I have set out to render in realtime a procedurally-generated planet. Rendering procedural planetary terrain presents two main challenges. First, the terrain is spherical rather than flat and finite. This results not merely in the difficulty of adapting existing dynamic level of detail algorithms but also in the problem of consistency since terrain curves and wraps. Second, the environment is so large and so many levels of detail will be required when traversing the planet that pregenerating content is impractical, and many algorithms depend on being able to access a master heightmap. The algorithm I have developed overcomes these challenges.

The Algorithm

My algorithm must meet three requirements. First, it must offload as much as possible to the GPU. Second, it must be adaptable for use on a sphere. Third, it must not pregenerate geometry. I will present a high-level overview of the algorithm before covering implementation details. My algorithm uses a cube-to-sphere mapping [8]; a quadtree approach for level of detail [10], where each node uses only one of only six geometric cases (stored as sixteen so rotation is unnecessary); vertex displacement in the shader [2,3]; and normal maps created by rendering to texture, using the GPU to compute the noise function.

In order to create spherical terrain, the sphere can be processed as a cube of six flat faces, with each vertex “spherified” to move from its cube to its sphere position. This cube-to-sphere mapping is done as follows:

\[
\begin{bmatrix}
  x' \\
  y' \\
  z'
\end{bmatrix}
= \begin{bmatrix}
  x\sqrt{1 - \frac{y^2}{2} - \frac{z^2}{2} + \frac{y^2z^2}{3}} \\
  y\sqrt{1 - \frac{z^2}{2} - \frac{x^2}{2} + \frac{z^2x^2}{3}} \\
  z\sqrt{1 - \frac{x^2}{2} - \frac{y^2}{2} + \frac{x^2y^2}{3}}
\end{bmatrix}
\]  

[8]

This means that each face of the cube may be treated as flat terrain and can be recursively divided with a quadtree approach. The complication is that consistency must be maintained not only within the face but at the face edges, and so (a) noise displacement must depend on the final model-space location of the vertex, not its original cube-bound position or mapping coordinates and (b) special handling must be used when finding a node’s neighbor if that crosses a cube-face boundary. [10]

The quadtree approach works by subdividing the terrain into square patches. Each patch may be split into four equal square child patches. Each patch
therefore is at a discrete resolution. For spherical terrain, we begin with six root nodes, one per face of the cube. Here I follow [10], although I use a different cube-to-sphere mapping. During each update, the tree is traversed; nodes too far from the camera, or out of view, have their children merged recursively; unsplit nodes near the camera are split. However, to maintain consistency the vertices at the borders between nodes must match. I will turn to the geometry used next, but for now it is important to note that nodes can only have as their neighbors nodes of the same depth, one depth level more, or one depth level less. This means each node in the tree must store a table of its four neighbors, if they are at the same depth, or null, if the node along that edge exists only at the prior depth level. When splitting a node, if it lacks any neighbors, the parent node's neighbor in that direction is first split, and this continues recursively. Thus consistency throughout the tree is maintained.

The key to this algorithm is the geometry used. The approach is inspired by [4] who outline an algorithm which divides square nodes into four triangular patches and stitching triangle strips. My algorithm uses quads directly with the strip attached. Each node may have one of six different meshes: normal resolution on all four edges; low resolution on one edge; low resolution on two adjoining edges; low resolution on two opposite edges; low resolution on three edges; low resolution on all four edges. Low resolution means half the number of quads along that edge.

This means that nodes can safely border lower-resolution nodes by having low-resolution edges; the edges will agree on the resolution and the vertices will still match. Since nodes are guaranteed to differ only one depth level with their neighbors, consistency is maintained.

Transforms of geometry (both position/rotation/scale and spherifying) is offloaded to the GPU. This means that only sixteen meshes need be stored, one for each case (low resolution neighbor on top edge and bottom edge, or low resolution neighbor on top, right, and bottom edges, for example). Thus very little geometry storage is required and only the transformation matrix need be passed to the GPU per node rendered to place the node correctly on the sphere.

In order to generate the planetary terrain, procedural noise is used. When each node is created, a normal map is generated on the GPU and stored in texture memory. Vertices in a given node are displaced outward along the normal to the center of the planet according to the noise function call. In this way geometry remains generic in memory and is only displaced at render-time. Normal maps are required because the vertex displacement changes the actual surface normals. Thus normals must be recalculated based off the displaced geometry. Note that the pregeneration of these maps on node creation is only an optimization; all noise calls could be made per vertex per frame. However, using a normal map in
particular yields a useful balance: it allows more detailed surface normals than per-vertex normals while not requiring per-pixel noise calculation in the fragment shader.

Figure 1: The six base cases

Figure 2: Nodes of different resolutions bordering one another
Implementation Details

I have implemented this algorithm in OpenGL 3.3 on Linux. I have used as my code base the final project I created for CPSC 478, Moonflight (which itself is partly based on the OpenGL tutorial code base by [6]). It uses FreeGLUT to handle routine OpenGL tasks, GLLoad to load OpenGL 3.3 functions, GLM for matrix math, minIni to read ini files, GLImg to read textures, and Open Asset Import Library to read object files. To this I have added the planet rendering code, the quadtree part of which is based on code by [9].

I will focus here on the planet rendering itself rather than other aspects of the implementation. The planet is stored as a Planet object that has six QuadNode roots. The planet object stores a radius, maximum height (displacement above radius), the six root nodes, level-of-detail-related members, and the geometry and materials and their associated members required to render the planet. The geometry required for planet rendering is created and stored in sixteen Vertex Array Objects. A single material for the planet is created. Six rotational matrices are generated, one per face, to transform the geometry to align with each face of the cube (recall that geometry is mapped to sphere in the vertex shader). Besides standard methods for handling members it includes an Update method and a Render method. I will return to these methods after examining the quadnode object.

The quadnode object stores a pointer to its planet, a pointer to its parent (null just in case it is a root node), pointers to its four children (null just in case it is a leaf node), pointers to its neighbors, one per edge (null just in case the node across that edge is at a lower resolution and so does not correspond), its depth in the tree, various flags (which face, which quadrant of parent node, camera in node, beyond horizon, outside of frustum), the XY position of its corners on the face of the cube, and the textures used for normals. Its methods include getters and setters, constructors for root nodes and for child nodes, neighbor-handling methods, coordinate handling methods, split and merge methods, and update and render methods.

On creation of a node, a normal map is generated. This is done through OpenGL’s framebuffer object support and offscreen rendering to texture. To create the normal map a texture with a higher resolution than the geometry per node is used, for smooth normal interpolation over faces. The high-detail-on-all-four-edges geometry and the node’s transforms are sent to the vertex and fragment shader. We also pass to the shader an offset in two directions to be used (calculated per node size and per face). For each pixel we read the interpolated world position, based on spherification. We then offset this world position positively and negatively by each of the two offset vectors. We then map these four points to the surface of the sphere, then call displacement on them. We then calculate the normal by taking the cross product of the vectors between each
opposite pair of points. This normal is output as a 3-channel float to the texture. The basic noise function used is 3D Simplex Noise [5]. This is used for fractal Brownian motion in four levels.

A helper class is used by both of these object types: PlanetCoord. This is used to translate between the planet coordinate system (0.0, 0.0 to 1.0, 1.0, per face) to world space and vice versa. It can also be used to determine whether a given set of planetcoords is inside or outside of a node, and what planetcoord is nearest to a given point in world space.

I will now consider the update and the render paths respectively. The planet's update method first determines the camera's altitude above the surface, and from this calculates the distance to the horizon (nodes beyond this distance can be merged together, see below). It also initializes the view frustum from the camera's view-perspective matrix. Next it sorts the six faces of the cube by distance from the camera and updates the root nodes in sorted order, closest first.

The quadnode update function first determines the distance from camera to the node's center. It then calculates the size of the node in world space and uses that, plus position and the precalculated frustum, to do frustum culling. If the node is outside the frustum, or it is over the horizon, its children are merged recursively and it is so marked. Next, the node size (as multiplied by level-of-detail constants) is compared to the distance to the camera; if it is too far away its children are again merged. If none of these conditions hold the node is split (if it is not already at max depth and if there are not already too many nodes). Finally the node's children are updated in order closest to the camera first (depth-first traversal), and this sort order is stored as we go.

I will now consider the split and merge logic before describing the render path. When split is called on a node, it is first checked to see if it is already split or if it is already at max depth. If so no work need be done. Next each neighbor is checked to see if that neighbor need be split, for if the neighbor is already at lesser depth it must be split to match the node's current depth before the node can be split. If so, split is called on the neighbor and this process continues recursively. Finally, four new child nodes are created for the nodes and the child nodes' neighbors are updated (which will in turn update the neighbors of surrounding nodes).

The merge logic operates in reverse. For a non-leaf node, if any of the nodes surrounding a child node have children, the child node cannot be merged into its parent and so the merge on the node fails. Otherwise merge is called on each of the node's children (and so recurses down the tree) and then finally the node's children are deleted and neighbor pointers are updated.
I now turn to the render path. The planet render method first runs `Update` on the planet. It then activates the planet material and sends radius and maxheight as uniforms to the shader. Then in closest-to-camera order each root quadnode is rendered. In the quadnode render method, we first check if the node is outside the frustum or beyond the horizon and if so skip. Next we do a sorted depth-first traversal using the sort order stored in the update pass. If the node is a leaf, however, we proceed to render it. We create a transform matrix based on the node's face and its extents on that face; we then bind the height and normal maps; we then activate the appropriate VAO based on which low resolution edges the node has and then render the triangles in the VAO. In the shader, each vertex is mapped to sphere by means of the transformation matrix (which translates, rotates, and scales the plane so that it is appropriately placed on the unit cube), by the spherify function (which translates each vertex to its position on the sphere). It is then displaced out to the radius of the planet and then displaced further by calling the `fBm` function on its world position. In the fragment shader the normal per pixel is found by texel lookup. We do one final noise call to add a bit of color and specular difference between pixels and add a fine layer of roughness to the planet's surface; without this if one zooms in closely the surface looks quite smooth since no texture is applied. This single per-pixel noise call does not add appreciable slowness to rendering.

A final note on geometry. You will note in figures 1 and 2 a node width of 4 quads is shown. Throughout this project for simplicity I had kept to a width of 8, but towards the end, to take advantage of GPU architecture, I upped the width to 16. This means that each quadtree node is approximately 512 faces (fewer for nodes with low-resolution edges). This yields less optimality in subdivision of the planet, but far more frames per second for an equivalent number of triangles. Further, I settled on a normal map multiplier of 2, so normal maps are 33x33. This yields a good balance between visuals and performance.

**Difficulties Encountered: General**

The major difficulties encountered are those mentioned at the beginning of this paper. Traditional dynamic level of detail algorithms require considerable reworking for use with planetary terrain. First, many of the savings these algorithms offer depend on the inherent flatness of the original terrain; displacement is always and only along the Y axis so only a vertex's X and Z coordinates need be stored, and even if procedural displacement is used one only needs 2D functions. Either the spherical geometry must be precalculated and stored (negating the advantage of displacement on the GPU) or it must be stored flat and warped into spherical shape on the GPU, complicating displacement. Next, consistency becomes difficult because the terrain wraps, and it does not wrap consistently. On flat terrain, the left edge of one node always meets the right edge of another, but on a sphere using cube to sphere mapping to enable quadratic
division that does not hold. Edge-to-edge mappings must be checked from a table whenever two different cube faces meet.

The other main problem mentioned at the beginning of this paper also deserves expansion. Most terrain rendering algorithms rely on stored height information. That is impractical in this case for two reasons. First, the planetary terrain is so massive that it is impossible to store all the planet's height and normal information at the finest level of detail, so it must be calculated on the fly. Second, because this information is unknown the algorithm cannot take advantage of precalculating geometry or using different mesh resolutions for "compressible" areas of terrain (where height variation is low). This has necessitated an approach where height information is entirely procedurally generated, but this can be quite expensive, even offloaded to the GPU, and where dynamic level of detail operates regularly, the quadtree approach.

**Difficulties Encountered: Specific**

I encountered four areas of specific difficulty in implementing this algorithm. First, it was difficult to guarantee that consistency is maintained when splitting and merging nodes. I had to develop a method for traversing up, down, and sideways across the tree to ensure that all a node's neighbors were at such level of detail as to allow a split and to split them if necessary, and then to make sure that all neighbors were updating during, not just after, these splits and merges.

Second, it was difficult to create a working formula for when to split nodes. First, the splitting cannot lead to too many nodes to render; second, one ideally wishes to split nodes in the correct order. An ideal approach would calculate the onscreen size of all nodes, split the largest, and loop, until the node limit is hit. However, this would be very difficult to implement because nodes are not flat when transformed to fit a sphere's surface and so it is hard to simulate screen size with bounding boxes. Instead I have roughly paralleled Sean ONeil's approach, wherein the edge length of a node is approximated, and it is then multiplied (and raised to a power) by split factors, and then checked against camera distance. If greater, it is split.

Third, normals were very difficult. Ordinary one could create vertex normals by averaging face normals, but since vertices are displaced in the shader at rendertime (not merely by height, but to a sphere) the normal must be calculated based off that. Further, since this is done in the shader, no other vertices or faces may be accessed. The method I use is not terribly efficient, because it does five fbm noise calculations per pixel in the normal map, but it does lead to good normals. At first I did these calculations (and height calculations) entirely on the fly, but the performance was too slow. Thus I
switched to precalculating the maps, but making OpenGL’s render-to-texture functionality work proved difficult in itself.

Fourth, I had originally intended to do the same thing with vertex displacement as with normals: precalculate on node creation the displacement values per vertex and store that in a 1-channel float texture. However, due to issues with the noise function, and texture coordinates and interpolating vertex data in the pixel shader this proved not worth the effort since there was negligible speedup from switching to a texture. There remains, however, one final reminder of this problem: a slight tear in one spot on the planet. It appears to be an issue of interaction between the noise function and the specific world position passed to it; given that this function cuts many corners for performance it is unsurprising.

Areas of Improvement

My algorithm offers a number of areas of improvement, besides optimization of the code. First, as described above, the splitting and merging logic based on camera distance could be improved, perhaps using bounding box approximation and transforming to screen space to calculate actual size on screen. Second, it remains to be seen whether, since normals are already being precalculated, it might be faster to precalculate vertex position information as well. (This is the method used in [9], creating a VBO per node). There is no reason this could not be rendered to a RGB32F 3-channel float texture like the normals are. Storing only displacement rather than calculating fBm displacement on the fly did not seem to make a difference, but perhaps this might, if the three square root calls in the spherify function are also slow.

Third, conversely if this method is not done, it is worth examining whether it would be faster to render all nodes using a specific VAO in one pass rather than binding and unbinding per node. This would require rendering in node-type order rather than forward-Z order and so would lose the optimization of depth culling, but might be worth cutting the number of state changes from over a thousand to sixteen. Fourth, a similar optimization might be done for the height and normal maps: if they are added to a texture atlas and each node stores texture coordinates rather than texture pointers, then we could reduce the number of texture binds each rendering pass from two thousand plus to two. However, that would require either sending the whole texture back into video memory whenever it changes, or storing the texture as read-write and doing texel operations.

Fifth, using an idea from [1] on geometry clipmaps, when creating a node we could pull the height map from the parent node and only add new levels of noise detail. This would mean that instead of computing the entire fBm noise function every time, we could only calculate node-appropriate levels. If the node is one of the roots, for example, then perhaps only the first or second level of
simplex noise need be calculated to get an accurate-enough sample of a vertex's displacement. For deeper nodes, we could pull the parent node's texture and then add level-appropriately-sized noise. This would also have the advantage of allowing infinitely increasing (to the limit of memory for node storage) noise detail; since my fBm noise function has only four levels, terrain becomes smoother and smoother as the level of detail increases. This limiting of fBm calls in the shader would make a great difference in framerate, since the noise calls use most of the GPU time. However, as GPU compute performance continues to improve it will become more and more optimal to do noise calculations on the GPU.

Finally, sixth, the creation of new nodes is expensive. It is worth considering queuing up splits and doing them in down time rather than trying to do all splits all at once. As it stands there is noticeable slowdown when the camera moves such that it causes nodes to split. This is still, amortized, far faster than redoing all noise calculations per frame, however; once the node generation is done the framerate quadruples at least.

**Usage**
Moonscape is invoked by ./Moonscape
   Any arguments passed trigger fullscreen mode.
   Every second, Moonscape will print the FPS and the number of nodes in the tree to the console.
Mouse moves camera around the target point. Scroll wheel changes camera altitude. Note that zoom rate is dependent on altitude: the closer the camera is to the planet, the smaller the zoom step on scroll.
I,J,K,L move the camera's target point across the planet's surface. I moves it toward the north pole, K the south: J moves it west, L east. Note that movement speed is dependent on camera altitude.
P dumps the planet's quadtree to the console.
ESC exits

**Performance**
30fps, 14 nodes (7,168 faces)       14fps, 174 nodes (89,088 faces)
20fps, 42 nodes (21,504 faces)      10fps, 302 nodes (154,624 faces)
15fps, 106 nodes (54,272 faces)     8.5fps, 370 nodes (189,440 faces)

Note: this engine is a proof of concept for seamless terrain rendering, built on top of an existing engine, and so I do not expect the excellent performance other algorithms in the field have gotten. Nonetheless this is an order of magnitude faster at high node counts than the engine demo provided by [9], as well as being far more scalable to GPU architecture (tested on a lowend Quadro NVS 295 of 2008.)
Screen Captures

Figure 3: The planet using normal fragment shader.

Figure 4: The planet showing subdivision. Note that each square on the checkerboard represents 2x2 quads
Figure 5: near the surface of the planet, showing normal and noise maps.

Figure 6: A different, closer angle.
Figure 7: The darker side, showing normals in higher relief

Figure 8: Showing the subdivision near the surface. See Figure 4 for explanation.
References


