Linköping University

REAL-TIME PROCEDURAL PLANETS

TNM084 – PROCEDURAL METHODS FOR IMAGES

Christian Alfons, chral647@student.liu.se

2011-03-03

Abstract

This report describes a project in which procedural planets with real-time level-of-detail management was implemented. The planetary terrain structure is based on quadtrees, where each node in the quadtree represents part of the terrain. Six quadtree faces are used to form a cube, which is then transformed into a sphere. Vertices on the sphere are translated along their normals to shape the terrain. Quadtree nodes can be split or merged, depending on relative camera position, to set the level of detail dynamically.

Height values are calculated using the ridged multifractal algorithm with three-dimensional Perlin noise. The height values are used to set vertex positions, and to create heightmap and normalmap textures. These textures allow the terrain to look very detailed, despite having simple geometry. For a performance boost, the textures are generated in shaders on the GPU.

Contents

Chapter 1

Introduction

Procedural content in computer games presents many possibilities, but it also creates several challenges. Procedural terrain, or more specifically procedural *planetary* terrain, has two primary challenges, the first of which is finding methods for generating interesting and convincing landscapes.

The second challenge is to sample the procedural terrain wisely. The sampled data must be sparse enough for modern personal computers to handle, but at the same time dense enough to look realistic. This is the technical task commonly known as *level-of-detail (LOD) management*.

This report documents a project for the course *TNM084 – Procedural Methods for Images*, Linköping University, with real-time procedural planet rendering as chosen subject.

1.1 Aim

The aim of this project is to create procedural planets suitable for games. Planet geometry is assumed to be static, but the user should be able to view planets from far away and up close, and to navigate seamlessly between different planets. All texture images, heightmaps, etc., are to be procedurally generated.

1.2 Method

There are many methods for creating spherical terrain [1], each with their own advantages and disadvantages, but this project uses the approach with six *quadtree* faces [2, 3]. Terrain generation is done using Ken Musgrave's *ridged multifractal* terrain model [4].

Geomorphing, the process of smoothly interpolating between different levels of detail, aims to reduce the *popping* effect when refining a mesh. This project, however, doesn't use geomorphing; popping is already mild when using dense triangle meshes and high-resolution textures.

The project is implemented using OpenGL in C++, but this report will be kept fairly implementation-independent. The OpenGL library GLFW is used for window and input handling, and GLee for loading OpenGL extensions. GLSL shaders are used for rendering, and utilizing the computational power of the GPU where necessary.

Chapter 2

Terrain Representation

This chapter explains the terrain representation, data structure, level-of-detail control, etc.

2.1 The Quadtree Structure

The quadtree is a popular data structure in terrain rendering. It is simple and efficient, but traditionally limited to 2D space partitioning. This section briefly describes the quadtree structure and how it can be adapted to work with planetary terrain.

2.1.1 The Quadtree Node

A quadtree consists of a number of *nodes*. In regular terrain rendering, each node can be thought of as a square that represents a certain region of a 2D map. A quadtree node may have exactly four children; one for each ordinal direction (or *quadrant*) – NW, NE, SE and SW (Figure 2.1). If a node has no children it is called a *leaf node*, and if it has no parent it is called the *root node*. When a node is split (when its children are created), each new child node represents one fourth of the original node. A node's *depth* tells us its position in the hierarchy (the root node has depth 0, its children depth 1, etc.).

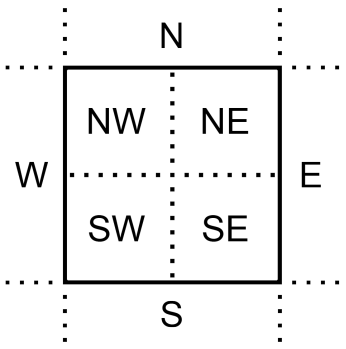

Figure 2.1: The four children (NW, NE, SE and SW) and the four neighbors (N, E, S and W) of a node.

The quadtree structure used here also allows the nodes to store neighbors. Any node that isn't the root node will always have at least two neighbors (which are also its siblings), and at most four neighbors; one for each cardinal direction – N, E, S and W (Figure 2.1). If no node of equal depth exists in a given direction, the parent's neighbor is used instead. If the node's parent doesn't have a neighbor in the given direction either, the parent's parent is checked, etc.

2.1.2 Terrain Patches

Each node has its own *patch*, which is the actual renderable object – a triangle mesh. The patches of all leaf nodes together form the current terrain. Adjacent leaf nodes may have different depths, which causes gaps to form between the patches if nothing is done to prevent it (since the vertices will not line up perfectly when the vertex density differs).

Thatcher Ulrich's *chunked LOD* technique uses a quadtree structure to render heightfields [5]. Ulrich uses *skirts* to hide the cracks between node meshes, which is simple and looks better than one might guess. Although doable, skirts are a bit impractical for planetary terrain (when building the cube, which will be covered later).

Another approach is to change the patch triangle structure based on the node's neighbors' quadtree depths. The patch geometry (vertex data) doesn't have to be updated, and the topologies (triangle structures) can be precalculated and shared by the patches. Figure 2.2 shows how a patch changes topology to match its neighbors' detail levels.

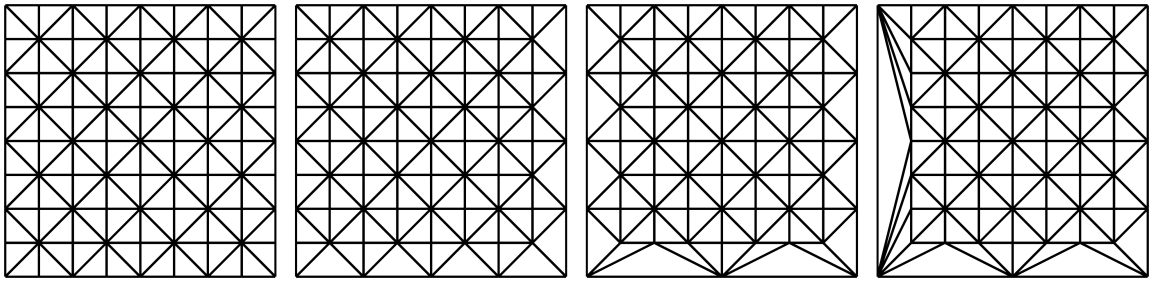

Figure 2.2: Examples of different patch topologies.

This method requires more programming work than skirts, but was chosen because it connects node patches of different depths seamlessly (Figure 2.3).

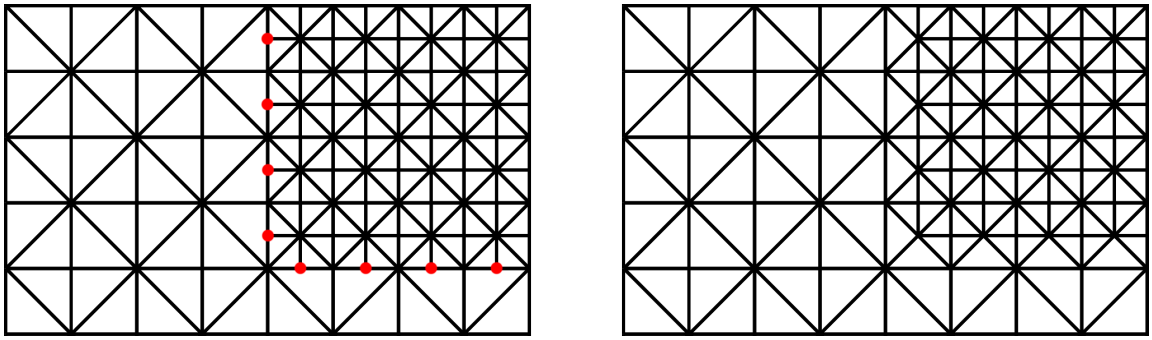

Figure 2.3: Seamless patching.

By setting the maximum allowed detail (depth) difference (D_{max}) between two adjacent nodes, the total number of possible topologies (T_{tot}) can be calculated.

$$T_{tot} = (D_{max} + 1)^4 \tag{2.1}$$

The *patch size* (P) is the number of rows (or columns) of triangle pairs in the patch mesh. D_{max} can be used to calculate the minimum patch size required (P_{min}).

$$P_{min} = 2^{D_{max}} \tag{2.2}$$

Knowing the patch size (P), the number of vertices per patch (V) is easily calculated.

$$V = (P + 1)^2 \tag{2.3}$$

All V vertices are only used at once for one topology (default); when there's at least one neighbor node of a different depth, a proper subset of the vertices is used, which is illustrated in Figure 2.2.

For this project, $D_{max} = 4$ was chosen, which results in $T_{tot} = 625$ and $P_{min} = 16$. In an attempt to find a good balance between detail and speed, $P = P_{min}$ was selected.

2.2 Level of Detail

Based on the camera distance to the patch bounding volume, a node can be either split or merged (have its children removed). First, the patch bounding sphere radius is multiplied by a number (S). A value of S greater than 1 should be chosen to expand the sphere; the more it is expanded the earlier nodes are split, which offers a trade-off between visual quality and computational intensity.

If the camera is inside the expanded sphere, the node is split (unless it has children already). Similarly, if the camera is outside the expanded sphere, the node is merged (if it has children). By repeatedly updating the quadtree terrain, its mesh is continuously refined or coarsened as the camera moves Figure 2.4.

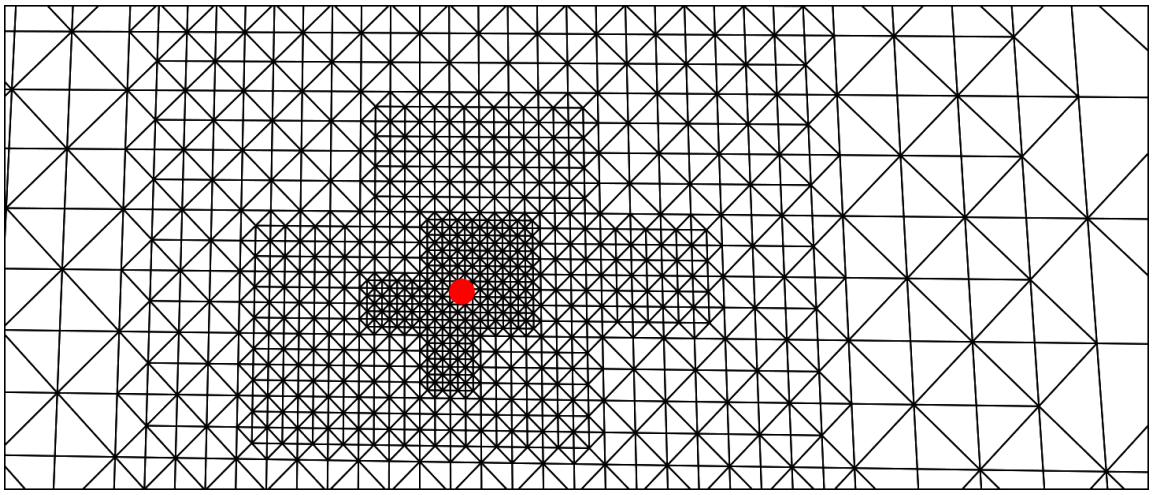

Figure 2.4: Position-based level of detail.

The quadtree is updated using depth-first traversal, which is easy to perform on the quadtree structure. The value chosen for D_{max} above may seem a bit excessive – if the node splitting rule is chosen wisely, and the terrain isn't *extremely* spiky, adjacent nodes shouldn't have to differ by more than one in depth – but by increasing D_{max} we can update the terrain in parts without having to worry about cracks frequently appearing between updates.

Cracks may still appear briefly in extreme situations (when approaching the terrain at *very* high speeds), so partial updates using breadth-first quadtree traversal may be a valid alternative.

2.3 Building a Quadtree Cube

One quadtree structure is obviously not enough to represent a whole planet, but how about six of them? With six root nodes, we can construct a cube where each node represents a cube face, which can be seen in Figure 2.5. The cube is of edge length 2, and is centered at the origin.

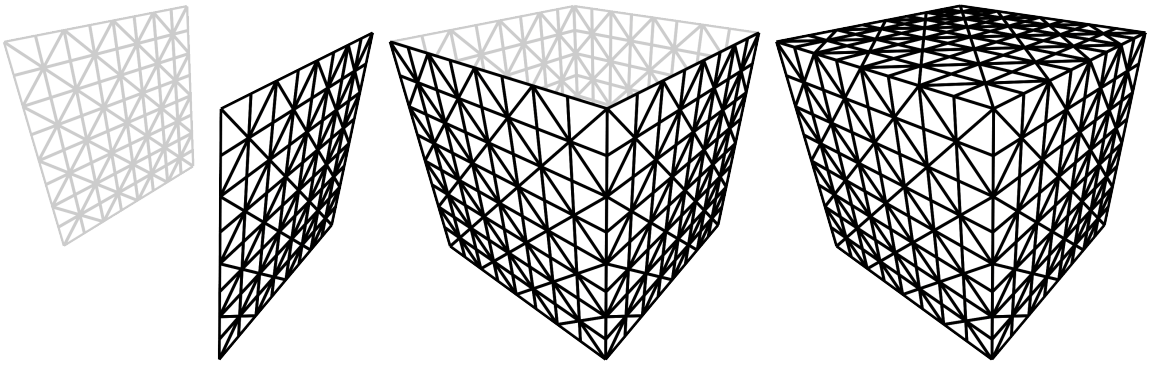

Figure 2.5: Making a cube out of six quadtree faces.

The terrain still doesn't look anything like a planet, but now it at least encloses a volume.

2.3.1 Connecting Cube Neighbor Nodes

Now it's time to connect neighbor nodes, since nodes need neighbor information to select the correct patch topology. This part is quite tricky; the idea is fairly simple, but there's a lot of room for mistakes. Connecting neighbor nodes can be skipped when using skirts, but – since we're not using skirts – let's connect some neighbors.

When two neighbor nodes are within the same quadtree (have the same root node), going between them is a simple task of *mirroring* the neighbor side (or direction); if node A is the north (N) neighbor of node B, then B must be the south (S) neighbor of A, etc. This relationship is used to create a two-way communication between neighbors, so that two neighboring nodes can easily be connected, and a node can tell its neighbors to update their neighbor data when it's being deleted.

To actually connect the neighbors, node quadrants have to be mirrored (or *reflected*) as well. A quadrant is always reflected in a given direction – for instance, the quadrant NW can be reflected to both SW (for directions N and S) and NE (for directions E and W). Hanan Samet thoroughly explains how this is used in quadtree neighbor searches [6].

However, when the two nodes belong to different quadtrees – when the root nodes of the quadtree cube are connected as neighbors – we have a problem; it's impossible to build a cube with six faces such that all faces are connected north-to-south and east-to-west [2]. This causes inconsistent neighbor mapping.

The solution in this project was to create two functions (which could probably be replaced by two lookup tables for a minimal performance boost) – one that mirrors the side (N, E, S or W), and one that reflects the quadrant (NW, NE, SE or SW) in a given direction, based on source and destination cube faces.

2.4 Cube-to-Sphere Mapping

The next step in creating a planet is to turn our cube into a sphere. The obvious way to map the cube described above to the unit sphere would be to simply normalize the cube points (Figure 2.6).

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \frac{1}{\sqrt{x^2 + y^2 + z^2}} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{2.4}$$

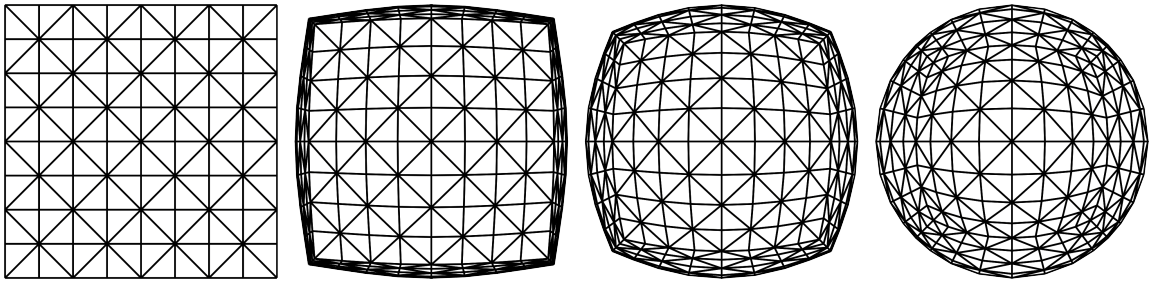

Figure 2.6: Cube-to-sphere mapping using normalization.

Philip Nowell describes another method for mapping a cube to a sphere [7].

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} x\sqrt{1 - \frac{y^2}{2} - \frac{z^2}{2} + \frac{y^2 z^2}{3}} \\ y\sqrt{1 - \frac{z^2}{2} - \frac{x^2}{2} + \frac{z^2 x^2}{3}} \\ z\sqrt{1 - \frac{x^2}{2} - \frac{y^2}{2} + \frac{x^2 y^2}{3}} \end{bmatrix} \tag{2.5}$$

This method is slower, but was chosen because it distributes points in a grid on the cube more evenly on the sphere (Figure 2.7).

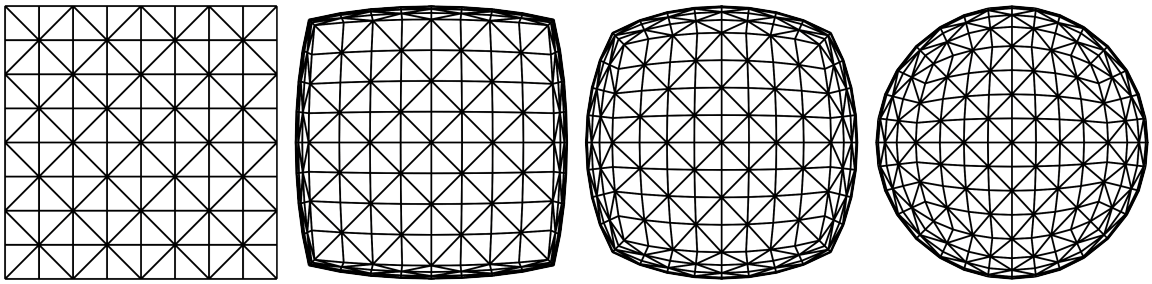

Figure 2.7: Cube-to-sphere mapping using Nowell's method.

2.5 View Frustum Culling

Before rendering, the six frustum planes for the current view are extracted [8]. By first testing if the bounding volumes of quadtree terrain node patches are located behind or in front of these planes, it might be possible to exclude nodes when rendering.

If the patch bounding sphere of a node is located completely behind either frustum plane, that node can be ignored when rendering (because its geometry is not in view). Since we're using a quadtree structure we assume that all children of such a node can also be ignored, making view frustum tests very efficient.

Chapter 3

Terrain Generation

Similarly to vertices in a heightfield being translated along their normals pointing up, vertices in a spherical terrain are translated along their normals – their normalized positions – pointing out of the sphere. How much the vertices are translated is determined by the *terrain height*, which varies over the sphere. This chapter explains how the terrain height values are calculated.

3.1 CPU Versus GPU

Vertex positions are calculated on the CPU, because the CPU offers double precision (64-bit) floating-point calculations. The double precision is very valuable when working on the different scales planetary terrain requires. The CPU implementation is also used for collision detection, using height values to test if objects are located beneath the terrain surface.

Generating geometry on the CPU could become a bottleneck, but the quadtree structure offers a very simple but important optimization – when a node is split, its newly created children can reuse its vertex positions (Figure 3.1).

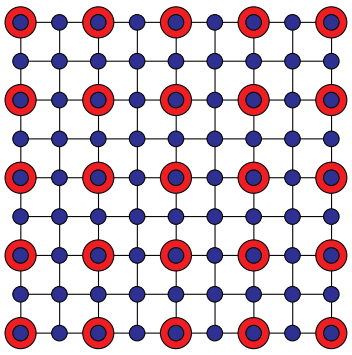

Figure 3.1: Vertices of a quadtree terrain node patch (blue) and its parent (red).

Using only per-vertex normals and lighting, the terrain will not look very detailed, and popping becomes apparent when adding or removing vertices. Instead, a *heightmap* texture and a *normalmap* texture are created for each terrain patch. In order to be useful, these textures require relatively high resolutions, and are therefore calculated on the GPU using shader programs – using the CPU simply isn't fast enough in real-time applications (yet), and the task suits the parallel structure of the GPU very well.

3.2 Perlin Noise

Three-dimensional *improved Perlin noise* serves as the core function in the terrain generation. It was implemented on the CPU based on Ken Perlin's reference implementation [9, 10]. A corresponding GPU version was implemented in a fragment shader, accessing the same permutation and gradient lookup tables via a texture.

3.3 Ridged Multifractal

Perlin noise in its basic form doesn't create very convincing terrains. The ridged multifractal algorithm, on the other hand, can use Perlin noise to create more interesting terrains. Simply put, the ridged multifractal sums Perlin noise of different frequencies, using the (squared) absolute value to create *ridges* [4]. The input is a point (here, a 3D point on the unit sphere), and the output is the height value.

Algorithm 1 The ridged multifractal

procedure RIDGEDMULTIFRACTAL($point$,$octaves$,$gain$,$lacunarity$,$offset$,H)
  $frequency \leftarrow 1.0$
  for $i \leftarrow 0, octaves$ **do**
    $exponentArray[i] \leftarrow frequency^{-H}$
    $frequency \leftarrow frequency \cdot lacunarity$
  end for
  $signal \leftarrow (offset - |perlin(point)|)^2$
  $result \leftarrow signal$
  for $i \leftarrow 1, octaves - 1$ **do**
    $point \leftarrow point \cdot lacunarity$
    $weight \leftarrow clamp(signal \cdot gain, 0.0, 1.0)$
    $signal \leftarrow (offset - |perlin(point)|)^2 \cdot weight$
    $result \leftarrow result + signal \cdot exponentArray[i]$
  end for
end procedure

The exponent array is actually precalculated (and recalculated when parameters change). The ridged multifractal result is divided by the sum of the exponent array – for $offset = 1$ this scales the output to fit within the range $[0, 1]$. It is then scaled and biased to the range $[-1, 1]$. Negative values can intuitively be seen as below ground level, which for instance could be useful when adding water.

Shaping the terrain by setting the parameters is not very intuitive, but the following values were found to give good results: $octaves \in [12, 20]$, $gain \in [0.0, 5.0]$, $lacunarity \in [1.5, 3.0]$, $offset \in [0.7, 1.3]$ and $H \in [0.7, 1.3]$. A high amount of octaves is necessary so that the terrain doesn't lose detail when getting closer. An additional variable, $heightscale \in]0.0, 0.2]$, sets the terrain magnitude in relation to the planet radius. Figure 3.2 shows examples of spherical ridged multifractal terrain.

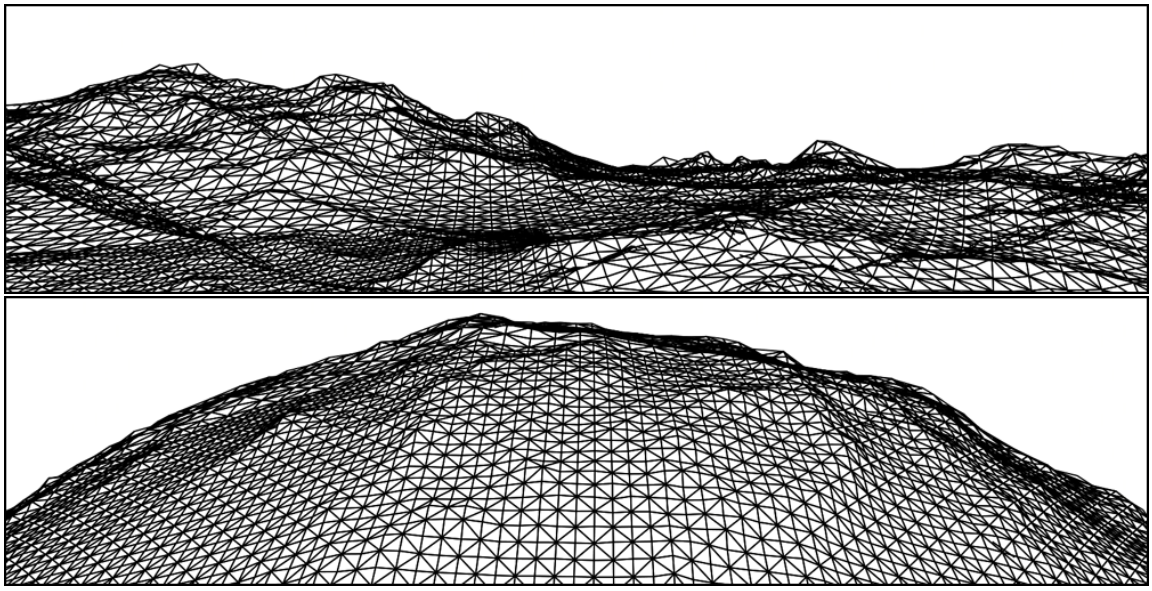

Figure 3.2: Spherical ridged multifractal terrain rendered in wireframe mode.

3.3.1 Positionmap

When a patch is created, a corresponding *positionmap* (in lack of a better name) is generated. The positionmap is an RGBA, 32 bits per channel float texture – which requires a lot of texture memory – but the positionmap only exists until it has been used to create the heightmap and the normalmap.

Using the cube-to-sphere mapping and the ridged multifractal algorithm previously described, a normalized height value ($[0, 1]$) is calculated. Using this height value, the terrain position is calculated and stored in the first three channels of the positionmap (RGB) – this will be used to calculate the normalmap. The fourth channel (A) is used to store the height value itself, to be used in the heightmap.

The heightmap and normalmap textures are both N by N texels (this project uses $N = 192$). The positionmap, however, has an extra border texel which makes it slightly bigger; $N + 2$ by $N + 2$ texels (Figure 3.3). This border texel is sampled *outside* the actual patch, and is used for normalmap calculations, to avoid discontinuities between neighboring normalmaps.

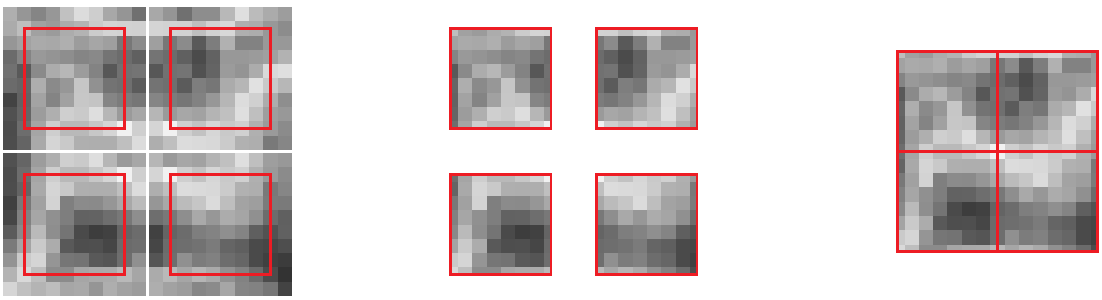

Figure 3.3: Four positionmaps, with the areas used when rendering marked in red.

When rendering, one half texel around the heightmap and normalmap textures is excluded for seamless patching without repeated texels (Figure 3.3).

3.3.2 Heightmap

The heightmap is a single-channel (luminance) 32-bit float texture. 32-bit height values aren't always necessary, but the $2^8 = 256$ possible height values the standard 8 bits offer are rarely enough for planetary terrain.

Since the positionmap already holds the height values (a), they are simply copied to the heightmap (h). Because the positionmap has the extra border texel, the indices are shifted by 1.

$$h_{x,y} = a_{x+1,y+1} \tag{3.1}$$

The height values are used to vary the color of the terrain, using a *colormap* in the form of a simple 1D texture lookup.

3.3.3 Normalmap

The normalmap is an RGB, 8 bits per channel unsigned byte texture. The normal values are scaled and biased from the range $[-1, 1]$ to $[0, 1]$ prior to storing (to fit the unsigned format), and are transformed back to the original range when sampling the normalmap.

The normalmap is in object space (the local planet coordinate system). This approach is efficient because normals don't have to be transformed when rendering, but it also requires light source positions and directions in the same space (object coordinates, as opposed to view coordinates). Similarly to vertex normal calculations, *edges* ($\mathbf{e}_i$) to adjacent positions are calculated using the position that is the RGB part of the positionmap ($\mathbf{p}$). As when creating the heightmap, the indices are shifted by 1 due to the extra border texel in the positionmap.

$$\begin{bmatrix} \mathbf{e}_{0,x,y} \\ \mathbf{e}_{1,x,y} \\ \mathbf{e}_{2,x,y} \\ \mathbf{e}_{3,x,y} \end{bmatrix} = \begin{bmatrix} \mathbf{p}_{x+0,y+1} \\ \mathbf{p}_{x+2,y+1} \\ \mathbf{p}_{x+1,y+0} \\ \mathbf{p}_{x+1,y+2} \end{bmatrix} - \begin{bmatrix} \mathbf{p}_{x+1,y+1} \\ \mathbf{p}_{x+1,y+1} \\ \mathbf{p}_{x+1,y+1} \\ \mathbf{p}_{x+1,y+1} \end{bmatrix} \tag{3.2}$$

Using these edges, we can calculate two *normal vector candidates* (not necessarily of unit length); $\mathbf{n}_0$ and $\mathbf{n}_1$.

$$\begin{bmatrix} \mathbf{n}_{0,x,y} \\ \mathbf{n}_{1,x,y} \end{bmatrix} = \begin{bmatrix} \mathbf{e}_{0,x,y} \times \mathbf{e}_{2,x,y} \\ \mathbf{e}_{1,x,y} \times \mathbf{e}_{3,x,y} \end{bmatrix} \tag{3.3}$$

Either one of $\mathbf{n}_0$ and $\mathbf{n}_1$ could simply be normalized and used directly as the normalmap value, but that makes the result very direction-dependent, and for low normalmap resolutions this could cause visible discontinuities where cube faces meet.

Instead, the normalmap value ($\hat{\mathbf{n}}$) is calculated using the normalized sum of $\mathbf{n}_0$ and $\mathbf{n}_1$ (implicitly using their mean value).

$$\hat{\mathbf{n}}_{x,y} = \frac{\mathbf{n}_{0,x,y} + \mathbf{n}_{1,x,y}}{||\mathbf{n}_{0,x,y} + \mathbf{n}_{1,x,y}||} \tag{3.4}$$

Figure 3.4 shows terrain with normalmaps.

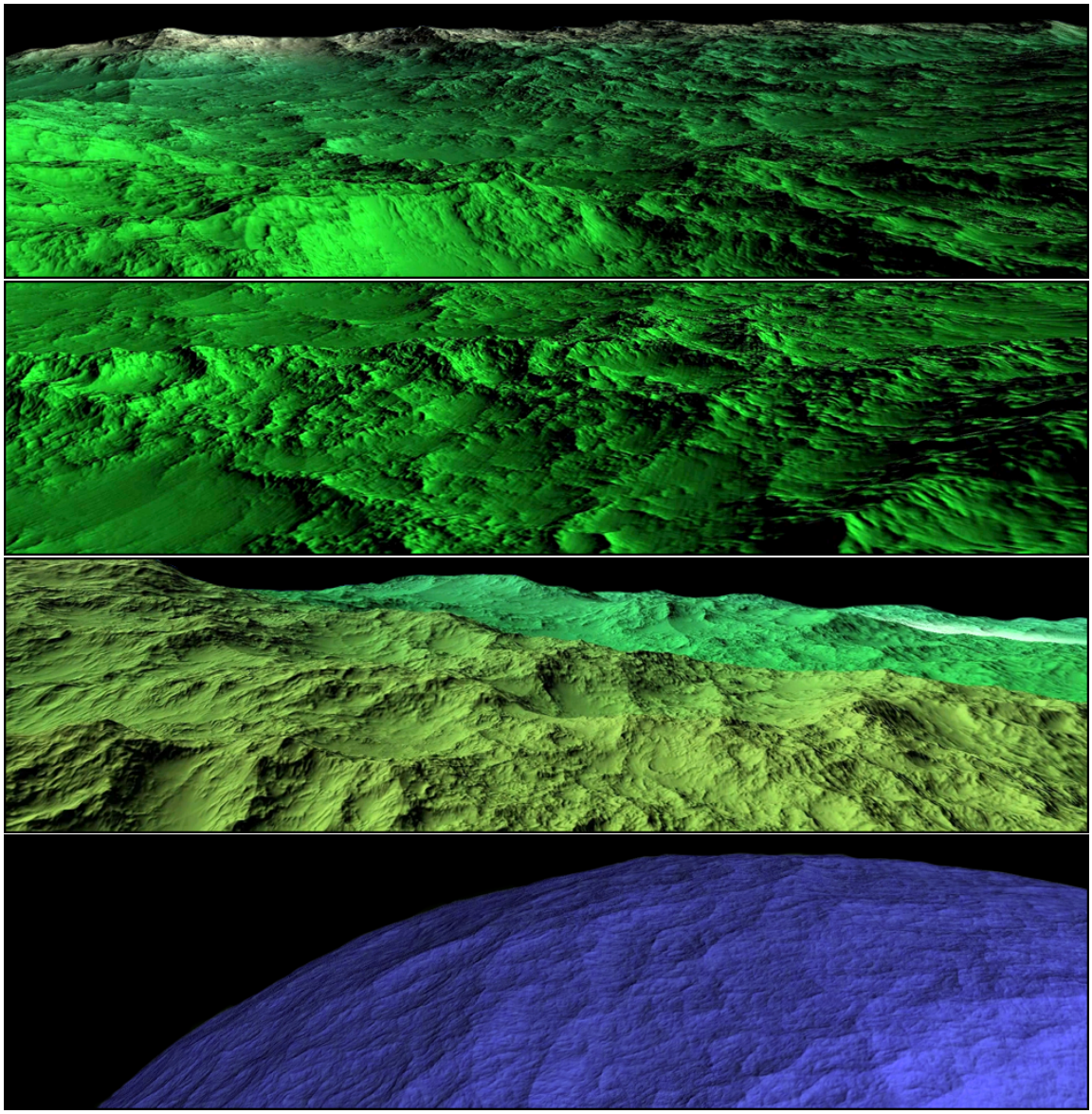

Figure 3.4: Spherical ridged multifractal terrain rendered with normalmaps, colormaps and lighting.

Chapter 4

Other Fancy Stuff

This chapter describes used techniques that – while interesting – aren't central to the project.

4.1 Skybox

Since a plain, black background quickly becomes pretty boring, a skybox is used to render more interesting patterns. The skybox is a small, inverted cube (i.e., only the inside is rendered), always rendered centered at the camera's current position. Since the skybox doesn't move relative to the camera, it creates the illusion of being very, very far away.

A shader program is used to render a skybox texture with cloud-like features (Figure 4.1) – perhaps best described as kind of funny-looking nebulae – let's just call it *space dirt* and hope there are no astronomers around.

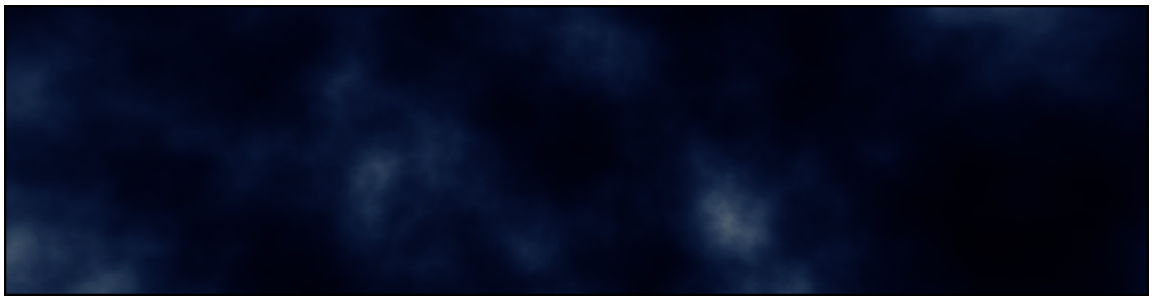

Figure 4.1: Space dirt.

The space dirt is created using an eight-octave multifractal with some color-mapping.

Algorithm 2 The space dirt multifractal

procedure SPACEDIRTMULTIFRACTAL($cubePos$,$octaves$)
  $spherePos \leftarrow cubePos/||cubePos||$
  $sum \leftarrow 0.0$
  $freq \leftarrow 4.0$
  $weight \leftarrow 1.0$
  for $i \leftarrow 1, octaves$ **do**
    $sum \leftarrow sum + weight \cdot perlin(freq \cdot spherePos)$
    $freq \leftarrow 2.0 \cdot freq$
    $weight \leftarrow 0.5 \cdot weight$
  end for
  $sum \leftarrow sum/octaves$
  $val \leftarrow 0.5 + 0.5 \cdot sum$
  $val \leftarrow (1.8 \cdot val)^{12.0}$
  $color.r \leftarrow 0.5 \cdot val \cdot val \cdot val$
  $color.g \leftarrow 0.5 \cdot val \cdot val$
  $color.b \leftarrow 0.5 \cdot val$
end procedure

4.2 Starfield

Similarly to the skybox, distant stars are static and rendered in the same manner. In fact, stars *could* be added to the skybox texture, but having such sharp features would require a much higher texture resolution than the low-frequency space dirt does. Instead, distant stars are treated as individual points of different sizes at a fixed distance from the camera, and rendered directly after the skybox (Figure 4.2).

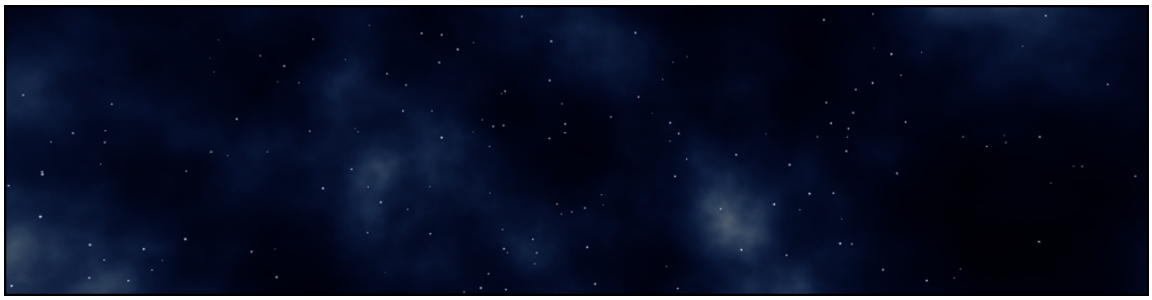

Figure 4.2: Space dirt with stars.

Since their positions relative to the camera are fixed, stars in the starfield use precomputed billboarding, and are rendered as squares with a simple Gaussian texture.

4.2.1 Custom Randomizer Class

A linear congruential generator (LCG) was implemented to ensure a cross-compiler safe randomizing function [11]. The LCG algorithm is simple, but it's sophisticated enough for randomizing star positions.

To distribute stars evenly on a sphere, it's not enough to just randomize positions in a cube and then normalizing them; this approach causes stars to be distributed more or less densely depending on the angle to the cube corners. Instead, 3D directions to be used as star positions are randomized using *sphere point picking* [12].

4.3 Atmosphere

An atmosphere is rendered as an inverted sphere, slightly larger than the planet it surrounds (Figure 4.3).

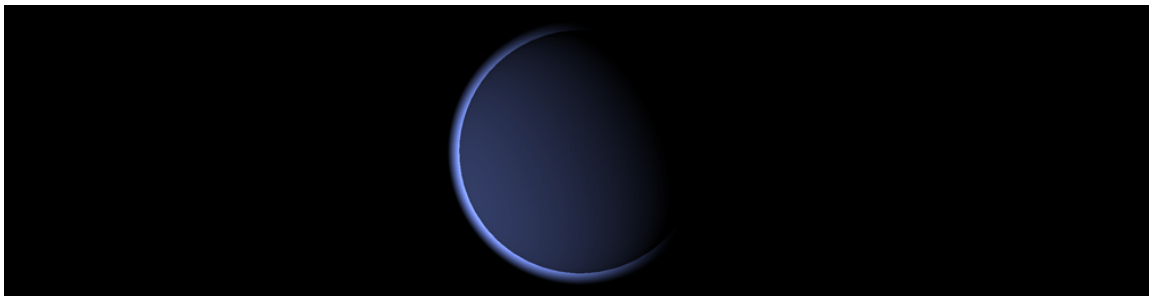

Figure 4.3: A very basic atmosphere.

The atmosphere in its current state is, to say the least, basic. There are no scattering effects; the atmosphere has only one color, and the atmosphere shader merely calculates a transparency value.

The shader samples along a ray, going from the camera position (or the atmosphere entry point, if outside the atmosphere) to the atmosphere exit point (the vertex position). For each sample, a light value (scalar product of sample direction and light direction) and a density value (fading linearly from 1 to 0 when going from surface to atmosphere height) are multiplied together, and the final transparency value is determined by the mean value of these products.

4.4 Clouds

Clouds are rendered as a textured two-sided sphere surrounding a planet. The cloud textures are based on Hugo Elias's work, using two simple settings – *cloud cover* ($C \in [0, 1]$) and *cloud sharpness* ($S \in [0, 1]$) – to convert a noise value ($v \in [0, 1]$) into a *cloud density value* ($d \in [0, 1]$) [13].

$$d = 1 - S^{max(0,v-C)} \tag{4.1}$$

This quickly produces decent-looking cloud textures (Figure 4.4).

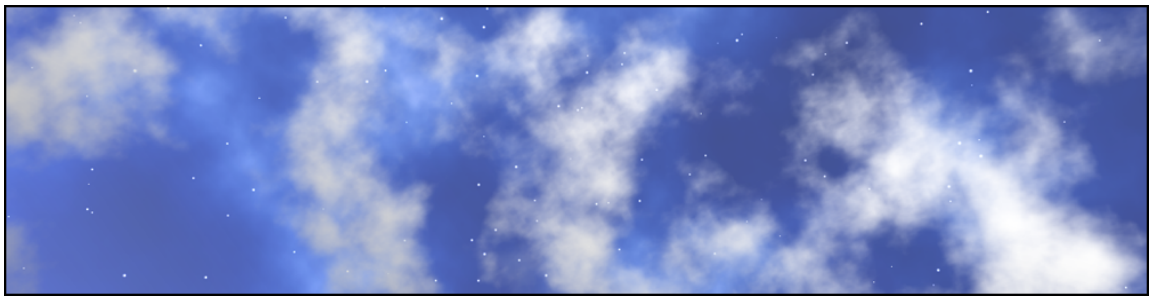

Figure 4.4: Clouds seen from a planet's surface.

Three similar cloud density images are generated and stored as separate channels in a single RGB texture. How much of each image is used is decided by three texture weights (w_r, w_g and w_b); the weighted sum is

used as the opacity value (α) of the cloud cover. The weights change with time (t), to animate the clouds.

$$\begin{bmatrix} w_r \\ w_g \\ w_b \end{bmatrix} = \frac{1}{3} \begin{bmatrix} 1 + cos(t) \\ 1 + cos(t - \frac{2}{3}\pi) \\ 1 + cos(t - \frac{4}{3}\pi) \end{bmatrix} \tag{4.2}$$

The time (t) is scaled to animate the clouds at different speeds. The sum of the three weights is 1 (Figure 4.5).

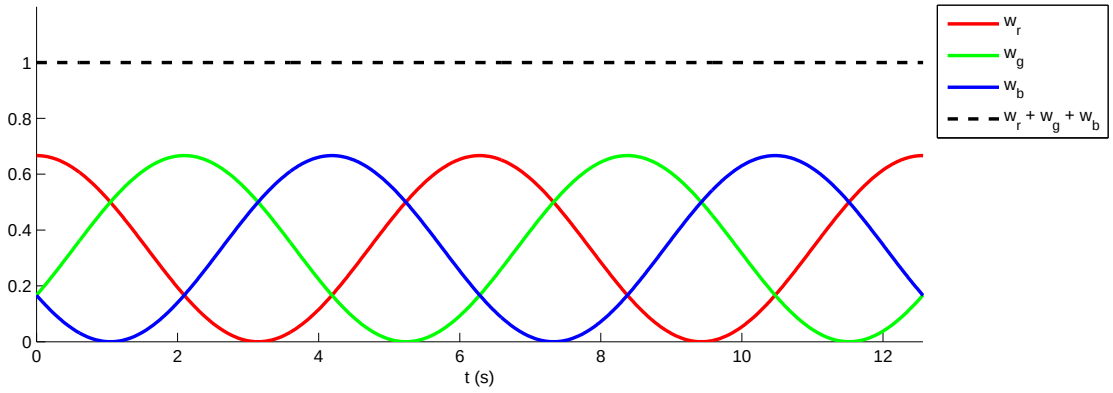

Figure 4.5: Cloud texture weights.

These weights create a smooth cloud animation, which looks okay when animated slowly, but at higher speeds the pattern becomes obvious.

4.5 Planetary Rings

A planetary ring is a disk with an inner and an outer radius. The disk uses a 1D texture where the texture coordinate is given by the vertex's distance from the planet's center; the texture coordinate is 0 where the ring begins (inner radius), and 1 where the ring ends (outer radius). Figure 4.6 shows an example of a planetary ring.

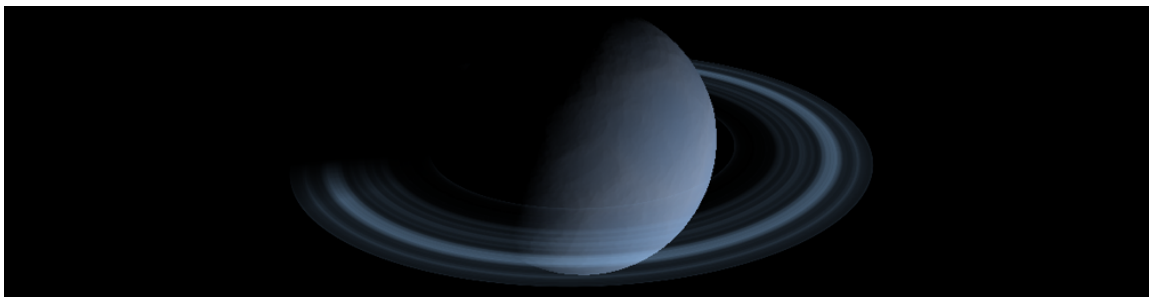

Figure 4.6: A ringed planet.

The ring texture is created by using the ring color with an opacity value, determined by the texture coordinate and ten octaves of (1D) Perlin noise.

Algorithm 3 Planetary ring texture generation

procedure PLANETARYRINGOPACITY($texCoord$,$octaves$,no)
  $alpha \leftarrow 0.0$
  $frequency \leftarrow 2.0$
  $amplitude \leftarrow 20.0$
  for $i \leftarrow 1, octaves$ **do**
    $alpha \leftarrow alpha + amplitude \cdot perlin(no + frequency \cdot texCoord)$
    $amplitude \leftarrow amplitude \cdot 0.5$
    $frequency \leftarrow frequency \cdot 2.1$
  end for
  $alpha \leftarrow alpha/octaves$
  $alpha \leftarrow 0.5 + 0.5 * alpha$
  $alpha \leftarrow alpha^3$
end procedure

4.6 Planetary Movement

A planet can be animated to rotate about its own axis and orbit another astronomical object. The application uses only circular orbits, which makes the calculations easy. The planet orbits on a circle, which is transformed by an orbit orientation matrix. The planet also rotates about its own axis, which is transformed by a second orientation matrix.

Chapter 5

What's Next?

When building planets, there's apparently a lot that can be done. This chapter gives examples on how the work presented in this report could be extended and tweaked. Some ideas would most likely work, while others probably aren't very well thought-out.

Rivers and Craters

Terrain features such as rivers and craters could be created using *Worley noise* [14].

Simplex Noise

Simplex noise could be implemented to be used instead of (or together with) the current Perlin noise implementation, to improve the quality of the noise function.

Extended Colormaps

Instead of just coloring based on terrain height, another dimension could be added to the colormap. For instance, adding terrain slope as a second dimension could be used to make snow stick only where the terrain is more or less flat. Or, the angle from the equatorial plane (latitude) could be used to create geographical zones.

Textured Terrain

So far, the terrain only uses a colormap value and basic normalmap lighting. Texture images could be created (perhaps even *generated*, to stick to the procedural theme), and blended together based on terrain properties in order to create more convincing-looking terrain. Perhaps using tri-planar texturing, as seen in NVIDIA's *Cascades* demo [15], could be used to avoid visible texture seams where quadtree cube faces meet.

Proper Atmosphere with High-Dynamic-Range Rendering

A much better looking atmosphere could probably be achieved using Sean O'Neil's real-time atmospheric scattering method [16]. High-dynamic-range rendering would allow some nice effects, such as stars (but not the sun) becoming invisible during the day.

Water

The spherical quadtree terrain could be adapted to render a water sphere, intersecting the terrain at some height. Similarly to terrain texturing, it may also be possible to utilize tri-planar texturing for spherical water rendering.

Volumetric Clouds

The spherical cloud layer may look okay from the planet surface, but the illusion breaks when seen from higher altitudes.

Planetary Ring Particles

Planetary rings could consist of actual objects that become visible when nearby.

A Procedural Universe

In this project, because a controlled environment was necessary, all planets were created manually. However, the planets could also be created procedurally by having their properties – size, orbit, fractal parameters, colormap, etc. – randomized based on planet position used as input to some (Perlin) noise-based function.

Similarly, a star could randomize orbiting planet count and positions based on its own position. The stars themselves would probably require something way more sophisticated (to form galaxies and whatnot). When traveling great distances, such as from one star to another, the skybox texture should probably also be made dynamic.

Bibliography

[1] 'Terrain LOD on Spherical Grids', *Virtual Terrain Project*, viewed 2010-06-30 <http://www.vterrain.org/LOD/spherical.html>

[2] Sean O'Neil (2006-01-12), *A Real-Time Procedural Universe, Part Four: Dynamic Ground Textures and Objects*, viewed 2010-07-08 <http://www.gamasutra.com/view/feature/2511/a_realtime_procedural_universe_.php>

[3] Steven Wittens (2009-08-23), *Making Worlds: 1 - Of Spheres and Cubes*, viewed 2010-07-08 <http://acko.net/blog/making-worlds-part-1-of-spheres-and-cubes>

[4] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin & Steven Worley (2002), *Texturing and Modeling: A Procedural Approach, Third Edition*, 3rd edn, Morgan Kaufmann

[5] Thatcher Ulrich (2002-04-14), *Rendering Massive Terrains using Chunked Level of Detail Control*, viewed 2010-06-30 <http://tulrich.com/geekstuff/sig-notes.pdf>

[6] Hanan Samet (1981), *Neighbor Finding in Quadtrees*, in Proceedings of the IEEE Conference on Pattern Recognition and Image Processing'81 (August 1981), pp. 68-74, viewed 2010-09-18 <http://www.cs.umd.edu/~hjs/pubs/SametPRIP81.pdf>

[7] Philip Nowell (2005-07-06), *Math Proofs: Mapping a Cube to a Sphere*, viewed 2010-07-08 <http://mathproofs.blogspot.com/2005/07/mapping-cube-to-sphere.html>

[8] Mark Morley (2000-12), *Frustum Culling in OpenGL*, viewed 2010-11-05 <http://www.crownandcutlass.com/features/technicaldetails/frustum.html>

[9] Ken Perlin (2002), *Improving Noise*, in ACM Transactions on Graphics (Proceedings of SIGGRAPH 2002) 21(3), pp. 681-682, viewed 2010-10-06 <http://mrl.nyu.edu/~perlin/paper445.pdf>

[10] Ken Perlin (2002), *Improved Noise reference implementation*, viewed 2010-10-06 <http://mrl.nyu.edu/~perlin/noise/>

[11] Jeffrey Walton (2008-04-10), *Applied Crypto++: Pseudo Random Number Generators*, viewed 2011-01-05 <http://www.codeproject.com/KB/cpp/PRNG.aspx>

[12] 'Sphere Point Picking', *Wolfram MathWorld*, viewed 2011-01-05 <http://mathworld.wolfram.com/SpherePointPicking.html>

[13] Hugo Elias, *Cloud Cover*, viewed 2010-09-30 <http://freespace.virgin.net/hugo.elias/models/m_clouds.htm>

[14] 'GPU Terrain Generation, Cell Noise, Rivers, Crater' (2008-10-25), *Journal of Ysaneya*, viewed 2011-01-05 <http://www.gamedev.net/blog/73/entry-1832259-gpu-terrain-generation-cell-noise-rivers-crater/>

[15] Ryan Geiss & Michael Thompson (2007), *NVIDIA Demo Team Secrets – Cascades*, viewed 2011-01-05 <http://developer.download.nvidia.com/presentations/2007/gdc/CascadesDemoSecrets.zip>

[16] Sean O'Neil (2005), *GPU Gems 2: Chapter 16. Accurate Atmospheric Scattering*, viewed 2011-01-05 <http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter16.html>