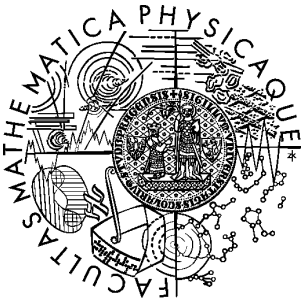
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Martin Kahoun

Procedural generation and realtime rendering of planetary bodies

Department of Software and Computer Science Education

Supervisor: Mgr. Jan Horáček,
Study programme: Computer Science, Programming

2010

I would like to thank to my supervisor, Mgr. Jan Horáček, for his time and advice.

I claim to have written my thesis by myself using only the cited sources. I agree with lending of this thesis and letting it publicly available.

In Prague on 5.8.2010 Martin Kahoun

Contents

List of Figures

Název práce: Procedurální generování planet a jejich zobrazování v reálném čase
Autor: Martin Kahoun
Katedra (ústav): Kabinet software a výuky informatiky
Vedoucí bakalářské práce: Mgr. Jan Horáček
e-mail vedoucího: Jan.Horacek@mff.cuni.cz

Abstrakt: Předložená práce se zabývá procedurálním generováním planetárních těles a jejich zobrazováním v reálném čase. Blíže zkoumáme jeden z možných přístupů a pokoušíme se vylepšit vizuální kvalitu produkovaných modelů. Při generování těles nebereme ohled na geofyzikální správnost planetárního povrchu ani neprovádíme fyzikální simulace procesů jeho vzniku. Představujeme, čeho je možné dosáhnout s použitím několika čísel, šumového generátoru a fraktálních funkcí. Důraz je kladen na vizuální dojem a poskytnutí podobného efektu jako v Google Earth, tedy možnosti prohlížet povrch nejen z velké výšky, ale i při přiblížení. Vedle toho umožňujeme export modelů do externího formátu vhodného ke zpracování ve 3D modelovacím software. Rovněž poskytujeme ukládání parametrů právě vygenerovaného tělesa a jejich opětovné načítání do aplikace.

Klíčová slova: počítačová grafika, procedurální terén, úroveň detailů

Title: Procedural generation and realtime rendering of planetary bodies
Author: Martin Kahoun
Department: Department of Software and Computer Science Education
Supervisor: Mgr. Jan Horáček
Supervisor's e-mail address: Jan.Horacek@mff.cuni.cz

Abstract: The presented work deals with procedural generation of planetary bodies and their rendering in realtime. We take a closer look on one of the available methods and we try to improve the visual quality of produced models. We don't consider geophysical correctness of the planetary surface during the generation, nor we do any physical simulations to create the planets. We present what is possible by using several numbers, noise generator and fractal functions. We emphasize aesthetic feeling, and we try to offer similar effect to the Google Earth, i.e., the opportunity to view a planet from high altitudes as well as a close-up zoom to the surface. Besides, we allow exporting generated models into external format suitable for processing in a 3D modelling software. We also support saving planetary parameters and their loading into the application.

Keywords: computer graphics, procedural terrain, level of detail

Chapter 1

Introduction

Procedural generation is a technique for an on-the-fly content creation. It is often related with multimedia, such as computer graphics applications, video games, and the film industry. In these particular fields using the procedural approach can rapidly cut down production costs. If we take video games as an example, the usual way their levels are designed requires a designer to spend hours of work to create the level geometry. Then several artists have to prepare textures and place them over the created geometry. This process consumes resources that could be spent elsewhere if the work on the levels and textures would be created algorithmically.

While this approach isn't applicable everywhere, there are numerous examples, in all sorts of areas, proving that the procedural content generation is effective. Ways of usage are different, ranging from procedurally generated unique details on in-game characters, through procedurally generated textures, up to the extreme of generating all content (examples include works related to the demoscene, films like *Tron*, and computer games like *Elite* or *Spore*).

Algorithms used for procedural generation must typically produce same results given the same input. We say, that they must be referentially transparent. Various fractals can serve as a good example.

We were interested in the procedural landscape generation, more specifically in the procedural generation of planetary bodies, as they can be used for quick content creation for independent (or hobby) game projects.

A large number of articles has been written in this area. However, most of them focus on generating "flat" landscapes. Nevertheless, there are two relevant works dealing with generating a spherical terrain. Hugo Elias posted an article *Spherical landscapes* [3] on his web-

site. Sean O'Neil wrote a series of articles called *A real-time procedural universe* [7, 8] which was featured on a prominent game industry server Gamasutra.com.

This was a starting point for our work. We took a path described by O'Neil and tried to elaborate on it. We focused mainly on the visual aesthetics of produced results. We utilized GLSL shaders to generate the terrain texture and to light the surface.

In chapter 2 we provide an overview of various techniques that can be used to generate a procedural landscape. We make a theoretical analysis of the chosen methods in chapter 3 and describe their implementation in chapter 4 as well as provide user's point of view to the program. In chapter 5 we present our results and key observations made during the development of the program. We make the summary of our work in chapter 6, and present topics for future work.

Chapter 2

Problem analysis

This chapter discusses several techniques for generating and realtime rendering of spherical landscapes. While the former task isn't that hard and a large variety of methods can be used, the latter task requires implementation of a suitable level of detail algorithm. Mainly due to the large scale of the landscape that needs to be rendered in real time.

In section 2.1 we list several methods applicable to procedural landscape generation and discuss their pros and cons. Section 2.2 discusses several techniques for applying level of detail algorithms to arbitrary landscapes. Section 2.3 then presents several ways of texturing such terrains.

2.1 Terrain generation techniques

We aimed, from the beginning, for an approach that would use a method which doesn't deal with any pre-generated data. In other words, we wanted to use some dynamic algorithm capable of yielding heights on demand, so we won't need to hold the whole detailed mesh of a planetary body in memory. Furthermore, implementing a level of detail algorithm above a fixed arbitrary geometry isn't an easy task.

We have studied several approaches that can be used to generate a random landscape with the above requirements in mind. We present the most notable of them.

2.1.1 The Fault lines algorithm

Algorithm proposed by Hugo Elias [3] is based on a flat algorithm that uses randomly placed fault lines to generate a heightmap. However, this

algorithm isn't dynamic – it generates a specified number of fault lines and raises or lowers a terrain along them.

The spherical version has a major shortcoming: the algorithm uses random planes cutting through the planet lowering, or raising, the land on either side of the sphere. If ran through enough iterations, it produces nice results; however, the back side of the generated planet looks like the front one turned upside down.

2.1.2 The Plasma algorithm

O'Neil talks about the Plasma algorithm [7] (also known as a diamond-square algorithm) that randomly subdivides a square to create mountainous terrain. It can be fitted for the on-the-fly generation and can be tailored for generating a spherical landscape. Unfortunately, its method is far too simple and produces an endless mountain range.

2.1.3 Other fractal based algorithms

Most of the fractal based methods are mostly static algorithms that pregenerate a heightmap. An example of these is a simple midpoint displacement algorithm. It recurrently subdivides a square, and at each step sets the height of the square's midpoint to a random number. Tailoring this algorithm for a spherical landscape would be tricky, not to mention, that the algorithm isn't fit for an on-demand execution.

Most of the other fractals are, on the other hand, computationally expensive and again not really fit for the on-the-fly generation we were aiming for.

2.1.4 Noise based approach

In his article [7], O'Neil proposes using a persistent noise generator (by persistent we mean that given the same seed and the same input value, the function will yield the same output value). One such method of creating random persistent noise is named after its inventor – the Perlin noise. More on the method can be found in a Ken Perlin's paper [9]. It's a popular noise generator with some nice properties:

- Supports arbitrary number of dimensions (the drawback is, that with each dimension the number of operations needed to calculate the result value grows exponentially).
- Same input coordinates produce the same result.

- Is easy to implement (see section 4.7).

Perlin noise gives good results, but they are either smooth in appearance, or look like a static from a television screen. By zooming to images produced by a Perlin noise generator one could see patterns observable in lower noise ranges – changing the range resembles zooming on a part of a noisy picture.

Lower ranges provide nice base for continent dislocation while higher ranges can be used to add details on the landmass. The idea for obtaining useful results is to combine the results from several ranges together.

For our application we have chosen the fractal Brownian motion method for summing the noise values of varying frequencies together. It is a simple fractal sum that can be written as follows:

$$noise(x) + \frac{1}{2} \cdot noise(2 \cdot x) + \frac{1}{4} \cdot noise(4 \cdot x) + ...$$

2.2 Spherical level of detail algorithm

Most of the level of detail algorithms are tailored for usage on "flat" terrains. Our goal was to find a suitable method that could be fitted into memory and would be easy to adapt for a spherical landscape. This requirement ruled out most of the available algorithms as they tend to focus on processing large amounts of data – they are typically used for displaying existing geographical data.

2.2.1 Geometry clipmaps

In a SIGGRAPH paper [4] Losasso and Hoppe present an approach based on the application of texture clipmaps to a regular grid elevation map. However, their algorithm decreases the level of detail purely by distance so it doesn't handle very well a high contrast terrain. Hoppe later improved the concept by heavy utilization of the GPU.

This concept was later applied to a spherical landscape by Malte Classen and Hans-Christian Hege [1]. However, this approach utilizes heavily the GPU as it requires complex calculations done in shaders. An approach that wasn't of a good use for us.

2.2.2 The ROAM algorithm

O'Neil described a simple adaptation [8] of the ROAM algorithm published by Duchaineau et al. [2]. ROAM stands for Real-time Optimally

Adapting Meshes, and it is an algorithm for displaying terrains in various levels of detail depending on the camera distance and a visible error metric, i.e., distant, or flat, areas are rendered with less details while near, or high contrast, areas are more detailed.

It basically treats the terrain as a single quad formed by two triangles. An error metric is then applied to determine when the visual error is large enough to justify an increase of the level of detail. This is achieved by splitting such triangles using a set of simple rules to prevent the occurrence of cracks in the mesh. When the high details aren't needed, we simply merge the triangles back to the larger ones.

It is this algorithm we have chosen for our application as it is easily adapted and implemented, produces good results, and doesn't require the use of expensive graphics hardware.

2.3 Texturing the planet

Texturing an arbitrary landscape is not a trivial task. Normally an artist would examine the landscape and adjust the texture coordinates in places where distortions occur. This is not an option with a procedural approach. Our task is even harder because we are trying to splice a texture to a spherical surface.

2.3.1 UV mapping

Standard methods for texturing a sphere include the use of polar coordinates. For this, a texture distorted along the equator is needed, so when applied to the surface it looks undistorted. One problem, with such approach, is the occurrence of visible and disturbing artefacts on the poles – the texture wraps around both poles and texels are often stretched from a single point towards the equator. Our assignment required prevention of such artefacts.

2.3.2 Cube mapping and texture combination

One possible approach is to take a heightmap of the landscape and then combine several basic terrain textures together taking the terrain height into account. E.g., in low altitudes we would use grass texture while in high altitudes rock or snow textures. We can also use texture blending for nicer gradients between the terrain types.

In case of texturing a planet, we would generate six such textures, and then cube-mapped them to the surface. This method, however, has one major disadvantage – the landscape can be large while the texture will be relatively small.

2.3.3 Tri-planar texturing

We have decided to generate the terrain texture in the GLSL shaders. For this we have partially used previously discussed cube-mapping. The problem was a visible hard seam artefact that occurred where the textures touched. However, our decision to use shaders allowed us to take advantage of a technique called tri-planar texturing presented in [6, chapter 1]. The method uses three planar projections of the texture coordinates to obtain the best result for a given position.

Chapter 3

Implemented algorithms

This chapter explains the theory behind the chosen methods for generating planetary bodies used in our program. We will talk about the spherical version of the ROAM algorithm, the fractal Brownian motion technique, and a terrain texture generation.

3.1 The spherical ROAM algorithm

The ROAM algorithm uses a tree structure for representing a terrain composed of triangles. Each node in the tree is equal to a single right triangle with size proportional to its depth in the tree. Root node represents a top level triangle encompassing the whole terrain, on the contrary, all leaves represent triangles to be rendered. We begin with two triangles composing a square. By calculating an amount of visible error we decide which triangles must be split to increase, or decrease, the level of detail.

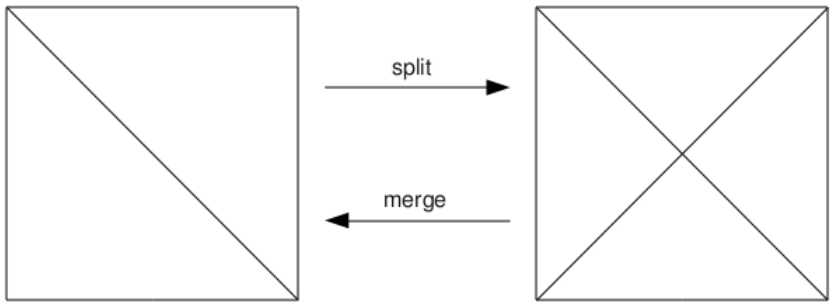


Figure 3.1: Split and merge operations

Triangles in the tree can compose two significant shapes – a square

and a diamond, as seen in figure 3.1 on the left and right, respectively. A square is defined as two triangles of the same size sharing the hypotenuse. A diamond is formed by four triangles sharing a vertex lying by their right angles.

We use a DFS algorithm to traverse the tree. For each leaf, we calculate the visible error and compare it with an error threshold. Based on the result of the comparison we either split the triangle, merge it, or leave it as it is. As a way of preventing rampant splitting or nearly infinite recursion, and the resulting buffer overruns, a maximum tree depth is defined. If the bottom floor is reached, no further split operation will be performed on the particular triangle.

In order to preserve water-tightness of the mesh, two simple rules must be followed when splitting or merging triangles. A split operation is allowed only when the triangle, we are about to split, is a part of a square; we will be actually splitting the triangle's neighbour along the longest edge as well. A situation in figure 3.2 shows how splitting of the triangle T requires an introduction of several new vertices, edges (shown in red), and triangles.

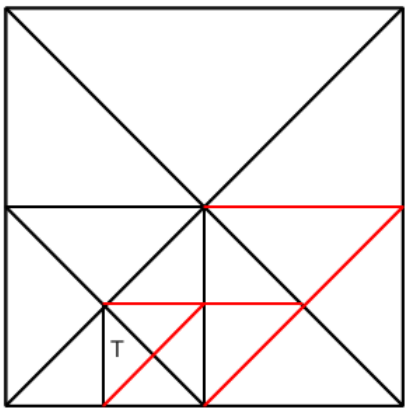


Figure 3.2: Recursive triangle splitting

Merging requires that the triangles to be merged form a diamond. Every split operation creates 1 diamond and destroys between 0, 1, or 2 diamonds. The merge operation does exactly the opposite – destroying 1 diamond and creating up to 2 diamonds. A care must be taken when operating on triangles lying at the edges of the initial quad to prevent memory leaks.

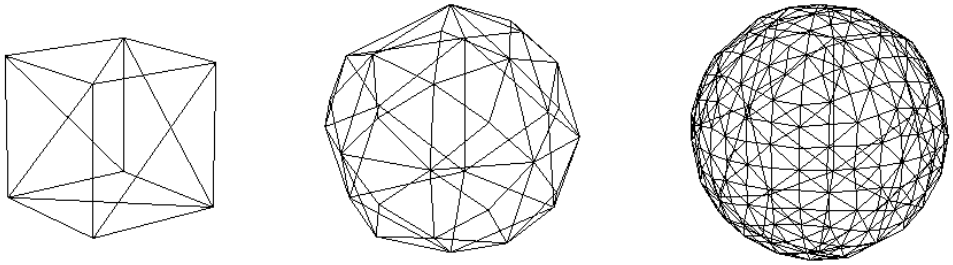


Figure 3.3: The ROAM Sphere at various levels of detail

We are using a spherical adaptation proposed by O'Neil [8] – it uses six ROAM squares to form a cube. This takes care of potential memory leaks on the edges of the quads since all triangles now have their neighbours. To create a sphere from the cube we move every new vertex to the surface of the sphere (by multiplying its direction vector by the sphere radius) and then we add the height obtained from the height generator. In figure 3.3 we can see the evolution of the initial cube.

3.2 Fractal Brownian motion

A fractal Brownian motion (often abbreviated as fBm) is a popular method for adjusting samples obtained from a Perlin noise generator to produce more interesting results. Instead of outputting the first value obtained from the noise generator, we take several noise samples, called octaves, with different frequencies and sum them together. This sum is then returned as a height value in a given direction. Following expressions show the first iteration of the fractal function:

$$w = 1.0$$

$$n = 1.0$$

$$h = 0.0$$

$$h = h + w \cdot noise(x \cdot n, y \cdot n, z \cdot n) \tag{3.1}$$

where h is the output value, w is the weight value, and n is the exponent (sometimes called lacunarity). We can see that each octave uses the same input vector, but with a different scale (n). We also influence how big impact these higher frequencies will have on the final value by weighing them (w). Expression 3.1 is then executed several times to combine the results of more than one octave (for implementation details

see section 3.2.1). Between the iterations we can modify both w and n values (or we can pre-calculate them) as follows:

$$w = w \cdot w_{factor}$$
$$n = n \cdot exponent$$

where w_{factor} and *exponent* are parameters to the function. The w_{factor} influences how much noise from the larger octaves will used. Values below 0.5 will result in a smooth terrain since the higher frequencies will not have big impact on the final value, while values over 0.5 will produce very turbulent noise which will result in large height differences on a small distance.

The *exponent* influences how high frequencies will be sampled. Values lower than 2.0 smooth out the terrain, but not as drastically as low values of the w_{factor}. Higher values, again, produce turbulent noise. Experimenting with these variables can bring both interesting, and uncanny results.

3.2.1 Simple fBm generator

A very simple fBm was implemented for testing purposes. A vector is passed as an input to the function and a single value representing the height in the given direction is returned. Prior return the result is truncated to $(-1.0, 1.0)$ and then scaled by the maximal height. The function is parametrized with w_{factor} and *exponent* values.

We can write the function as follows (the pseudo-code doesn't contain truncating and scaling for simplicity):

$w := 1.0$
$n := 1.0$
$result := 0.0$
for 1 to k **do**
  $h := w * noise(x \cdot n, y \cdot n, z \cdot n)$
  $w = w \cdot w_{factor}$
  $n = n \cdot exponent$
  $result := result + h$
end for
return result

where x, y, and z are terms of the input vector, and k is the number of octaves (predefined in our implementation – see further). Note, that

if k is large enough, the execution of this method can take considerable amount of time – the 3D noise function will be called k times. In the current implementation that constitutes 8 lattice lookups and 7 linear interpolations. Each lattice lookup constitutes of 3 multiplications and 3 modulo (division) operations. Together, we get the approximate time complexity of:

$$O((6 \cdot 8 + 7) \cdot k)$$

Theoretically, this isn't that bad, but 55 multiplications and divisions can cost some time. Especially, when we take into account the total number of fBm function calls; a request for height calculation happens every time a new triangle is introduced (each split operation creates four new triangles). Although we check whether the value hasn't been previously calculated by the triangle's neighbour along the longest edge, the number is calls is large.

3.2.2 Advanced fBm generator

The simple algorithm provides good, but uninteresting results with almost uniform height gradients. Therefore, we have implemented a more intricate fractal algorithm to produce more visually pleasing surface. We use a fractional number of octaves (k) and slightly different default settings of the fBm routine. The method is executed in a similar manner as the simple one.

First, we sum $\lfloor k \rfloor$ number of octaves and then do the following:

$$h = h + (k - \lfloor k \rfloor) \cdot w \cdot noise(x \cdot n, y \cdot n, z \cdot n)$$

Finally, we post-process the generated h value. If the resulting number is below zero, we assume that it creates the ocean floor. We leave the value as it is to some distance from the shore. We extract the root of values outside this threshold. This causes a sudden drop and flattening of the ocean floor. That results in darker hues of grey which in turn results in darker ocean colour in depths and brighter along the shores. As a side effect, if the user changes the ocean level, he can either expose the ocean floor or flood the landmass.

Positive values are raised with random exponents obtained from the noise generator. This action results in lower altitudes becoming mostly flatland, while higher altitudes obtain mountainous look. Values above 1.0 are then subtracted from 1.0 in order to create valleys or more interesting mountain regions. Another reason for doing so is more practical

– we have only a limited scale of grey hues and values over 1.0 would result in an all white region. The geometry would use the returned value, though. The heightmap would, on the other hand, have the same value everywhere and the normal map would result in a flat terrain. From a distance we would see a mesa, but in the close-up zoom we would see that the geometry runs higher with the terrain lit as if it was flat.

Time complexity of this method is similar to the simple one, but one has to take into consideration three conditions and power function calls that aren't trivial. This results in a more time costly method than the simple implementation. As in the case of the simple generator, the result is truncated to $(-1.0, 1.0)$ and then scaled by the maximal height.

3.3 Terrain texturing

Texturing in our project is achieved through the combination of a precalculated heightmap texture that further serves as a height lookup for shader programs. The heightmap is relatively small (the exact size can be set up by the user), nevertheless it is sufficient for providing good amount of visible details.

Heightmap is also used for generating a normal map used for per-pixel lighting which drastically improves the visual quality of the landscape. All terrain texturing is then done inside the GLSL fragment shader. We use four basic terrain textures to create the surface texture. These textures are loaded from external files, though[1].

3.3.1 Heightmap and normal map generation

We use a technique called cube mapping to place the heightmap texture to the surface. This technique uses 6 textures (each representing one face of a cube) that are then projected to the sphere. We generate all textures inside two loops going through all their texels. Essentially, we are making $n \cdot n$ height samples, where n is the size of the texture specified by the user (see appendix C).

First, we set six initial vectors pointing towards the cube vertex to which the top-left corner of the texture is mapped. Then we calculate the increment between the samples by dividing length of the cube edge by n. This way we get non-linear probing ray distribution in the given wedge (pyramid with the apex in the point of origin and base coincidental with the respective cube face). Samples from areas near the face edges will be

[1] It was beyond the scope of the project to generate them in code.

more dense than samples from the middle of the face. This can help to reduce some distortions along the texture seams.

We then loop $n \cdot n$ times and gradually increment the initial vectors along which we cast the probing rays. We use them to read the height (obtained from the planet's generator) in the given direction. Then, given the maximal height for the planet we calculate a hue of grey and write it to the texture to appropriate coordinates.

The normal map is generated in a similar fashion. But, instead of polling the generator for height information, we use the heightmap. This saves time that would otherwise be repeatedly spent by polling the generator (see section 3.2 – a paragraph describing the time complexity). We make total of five height samples (current position, up, down, left, and right), and then calculate a normal as a sum of two cross products of vectors pointing to all four directions.

Usage of more samples, or even other techniques for the normal-map generation, is quite problematic since we have to take samples from other textures. Even now we take the border values from the neighbouring cube face and have to smooth out borders of all six normal maps to reduce the visible seam artefact and remove the hard seam artefact.

The object space normal is then encoded into RGB value and stored in the texture.

Our weather zone map generation uses the Perlin noise with an inverted seed to obtain smooth pattern of wet and dry zones. This way doesn't provide realistic desert or tropical zone dislocations, but realism wasn't the focus of this project. Results are quite satisfactory and create surface diversity which in turn improves the visual quality.

3.3.2 Shader based texturing

We use GLSL 1.1 shader programs to replace the fixed OpenGL rendering pipeline with our own that generates the terrain texture The shaders are defined in the `vertex_shader.shd` and `fragment_shader.shd` files, and must be distributed along the executable.

The vertex shader mainly transforms vertices according to the projection matrix. All vertices that are under the ocean level are moved to the ocean level, thus creating the water surface. The fragment shader then uses information about the fragment obtained from the cube map texture and parameters supplied by the application to determine the final colour of the current fragment.

First, we determine the base terrain texture according to the height of the particular fragment (height-based texturing). Underwater regions

are coloured with a hue of blue that depends on the depth of the sea. On land we blend terrain textures to obtain smooth transitions between the various regions. This colour is then blended with the desert texture (created as a mixture of sand and dirt textures based on the fragment's latitude) if the region is a dry one (information obtained from the weather zone texture, see section 3.3.1), or adjusted to become "greener" if the region is more wet.

Texture coordinates for terrain textures are calculated with a technique called tri-planar texturing. It uses three planar projections of 3D texture coordinate used to sample the cube-map and blends between them in order to use the projection that is closest to the direction the current fragment is facing. This technique however requires three times more texture fetches than other techniques, on the other hand, it produces nice results and isn't complicated.

Fragment shader also calculates per-pixel lighting using the Phong lighting model. Normals are taken either from the normal map or calculated in the fragment shader for all water regions (it is more accurate than the normal map).

Several thoughts and methods implemented in the fragment shader has been inspired by approaches used in the demo application Kris Nicholson wrote for his report [5].

Chapter 4

Implementation

This chapter introduces the *Slartibartfast*[1] planet generator from the user's and programmer's perspective. It covers the installation process, possible problems that might occur in the course of building the program from sources, program setup, program controls, and an overview of the code structure accompanied with key implementation details.

Slartibartfast has been written with respect to the object oriented programming paradigm, therefore, nearly all program code is enclosed in an interconnected system of classes. This enables easy extensibility, as well as good code re-usability in further projects which was part of the goal.

We started programming from scratch; however, we have reused some previous work of mainly supportive character. The first step was to create an OpenGL application that could render a cube. With this basic setup, we have implemented the spherical version of the ROAM algorithm. The program, at this point, was able to render a sphere subdivided from the initial cube in wireframe. The next step was the implementation of the simple world generator.

Then, came the generation of the cube map texture and its application to the surface, we have also added per-vertex lighting. Up to this milestone, all variables that influence the final result were implemented, but there were no means for the user to change them in the program. The simple command line interface was added to provide the user interface. All other functionality, such as saving, loading, or exporting was coded as well.

Last steps included the development and incorporation of the GLSL

[1] The name is taken from *The Hitchhiker's Guide to Galaxy* by Douglas Adams. The book contains a character of the same name whose job is best described as a planet designer.

shaders. These replaced the OpenGL fixed rendering pipeline in order to create terrain textured surface, as well as improvement to the visual quality by using normal maps and per-pixel lighting. Outside the shaders we use normals that are calculated at the time a new vertex is created

While we use a method of geomorphing to slowly move the vertices into their positions, we don't recalculate their normals during the movement to prevent high CPU load. In other words, if we would use per-vertex lighting, all the effort to reduce visible vertex "popping" would have gone to waste. User is encouraged to compare the lighting quality of the GLSL and lit heightmap texture modes.

The chapter is supplemented with appendices A, B, and C containing an overview of all control keys, available console commands, and possible program settings along with their short descriptions.

4.1 Installation

Slartibartfast is a cross platform application written in C++ with utilization of the OpenGL graphics library and the Allegro library for providing an operating system independent window management and user input. It is distributed under the zlib license which is included with the source code. The program also makes use of GLSL1.1 shaders to render the terrain textured surface (the user should have a video card that implements at least the GLSL1.1 shaders).

The source code should be portable to virtually any platform that has a port of the C++ compiler, the Allegro library, and an OpenGL implementation. Portability has been tested on Windows and GNU/Linux environments (both 32 and 64 bit). Because of this, the application doesn't contain installation program of any sort. It is distributed as a compressed archive containing the source code, makefile (build-able with GNU Make) and pre-compiled binaries for 32 bit Windows.

All libraries needed to successfully build and run the application are included in a dedicated folder. However, the OpenGL implementation depends on the video card vendor – the open source implementation shipped with the source uses software only renderer and doesn't support GLSL shading language, thus the functionality may be somewhat limited.

Remainder of this section describes how to build the program from sources.

4.1.1 Prerequisites

In order to successfully compile this program, the following libraries must be present on the target system:

- OpenGL SDK - Should be provided by a video card vendor – version 2.0 or higher is required (mainly for GLSL1.1 support). Use of MESA3D library is strongly discouraged. Important OpenGL headers are, nevertheless, included in the `GL` folder that can be found in the root of the source package. Should the user be unable to obtain and install OpenGL headers on the target system, he should consult section 4.1.5, page 31, paragraph dealing with outdated OpenGL headers.
- Allegro 4.2.2 - It is supplied in the libs directory. A care must be taken when getting the library from the Internet to not use Allegro 5 branch because it has a different API.
- AllegroGL 0.4.3 - This library is also supplied with the program and provides binding between Allegro and OpenGL.

To build the Allegro libraries, the user should simply unzip both source packages and follow build instructions provided for the target platform. On Windows a pre-compiled distribution can be used. The Allegro library can be found in the packages manager on some GNU/Linux distributions. However, the AllegroGL will seldom be there.

If any problems occur during the prerequisites build process, user should consult section 4.1.5 covering possible problems that might occur. Platform specific instructions follow.

4.1.2 GNU/Linux instructions

Slartibartfast is supplied with a GNU Make makefile. User should check the file to make sure that the `PLATFORM` variable is set to `"linux"`, and then type `"make"` into the shell. After that, the compiled binary should be present in the `bin` folder ready for use.

If the Code::Blocks IDE is installed, the supplied project file can be used. Again the user should make sure that the linker options are set up correctly. They should read the following (the order is important):

```
-lagl
‘allegro-config --libs‘
-lGL
```

```
-lGLU
```

After a successful build the compiled binary should occur either in `bin/Debug` or `bin/Release` directories depending on the specified target.

4.1.3 Windows instructions

On Windows the process isn't that straightforward. However, there should be no pitfalls. The easiest way to build *Slartibartfast* on Windows is to have Code::Blocks installed, or, at least, the MinGW compiler suite[2]. On the other hand, any compiler or IDE should be sufficient.

Using just the MinGW suite the user can build the program by checking that the `PLATFORM` variable is set to `"windows"` and invoking `make` or `mingw32-make` (the exact command depends on the version of the MinGW package).

Should the user decide to use Code::Blocks for building the project, he should make sure that the linker settings are set up correctly – they should read the following (the order is important):

```
-lagl
-lalleg
-luser32
-lgdi32
-lopengl32
-lglu32
```

On 64 bit systems it is likely that the linker settings will look slightly different, depending whether the user wants to build a 32 bit binary or a 64 bit binary. After successful build the compiled binary should occur either in `bin/Debug` or `bin/Release` directories, depending on the specified target.

Building the project using other IDE's will require some work. First, a new project for console application must be created. Then, all source and header files located in `src` and `include` directories must be added to the project. The following libraries must be linked with the executable (the "d" suffix by the Allegro libraries designates the debug version of the library):

[2]The suite is a Windows port of the GNU C Compiler and is a part of the Code::Blocks installation package.

- AllegroGL (agl/agld)
- Allegro (alleg/allegd)
- User32 (or 64)
- GDI
- OpenGL
- GLU

If the Allegro library was linked dynamically, the `alleg42.dll` should be distributed alongside the executable. It is also a good idea to include Windows runtime libraries.

4.1.4 Other operating systems

Ports for other than Windows and GNU/Linux environments weren't tested, but as stated earlier, the code should be portable and the only limiting factor is the existence of the C++ compiler, OpenGL2.0, and Allegro implementation on the target platform. Therefore little to no effort should be required to build the program on such platforms.

4.1.5 Troubleshooting

Generally, following the build instructions provided by both Allegro libraries should not yield any problems. Though some circumstances may complicate the build process. Bellow are the known issues that might arise during the build of either Allegro, AllegroGL, or the project itself.

No root privileges

The user can have limited access to the machine and thus cannot install any third party libraries (this typically happens on UNIX or GNU/Linux workstations). Both Allegro libraries offer local install option that will install them into the home directory instead of the standard location. When doing so, the user needs to adjust his `PATH` variable to include his home directory – specific instructions are contained within build documentation supplied with the libraries.

On Windows Vista and higher, the user should run the program within a directory he has a write permission to, otherwise some functions may not be working properly.

AllegroGL related errors

On some compilers the user may observe the following (or similar) cryptic message while building the AllegroGL:

```
allegrogl/GLext/gl_ext_api.h:1827: error: ‘<anonymous>’ has incom...
allegrogl/GLext/gl_ext_api.h:1827: error: invalid use of ‘GLvoid’
```

It is a known issue on some versions of the *g++* compiler, but there is a simple workaround. User needs to open up the mentioned file (`gl_ext_api.h`) and change the line 1827 from:

```
AGL_API(void, EndTransformFeedbackNV,     (GLvoid))
```

to this:

```
GL_API(void, EndTransformFeedbackNV,     (void))
```

Outdated OpenGL headers

One can observe a compiler error message regarding the `glUseProgram()` or any other shader related function saying that it wasn't declared. This indicates that the OpenGL headers present on the system are of version bellow 2.0. Optimally the user should obtain newer version of the OpenGL distribution (usually from the video card vendor). But that may not always be an option.

Slartibartfast comes with the OpenGL2.0 headers included inside the `GL` directory. On Windows, some headers from the Mesa library might need to be included in the compiler's `libs` directory. The user has just to make sure that the compiler looks for the included OpenGL headers first and includes them at compile time. When using the makefile the following line should be uncommented to achieve the described effect:

```
#override CCFLAGS += -IGL
```

4.2 Usage

For a quick reference, all control keys, console commands, and program settings are listed and described in tables at the end of this text (see

appendices A, B, and C). Rest of this section takes a deeper look on using the planet generator.

Upon the first run *Slartibartfast* will use default settings – running in 800x600x32 windowed mode – and will generate the configuration file. User can edit this file in order to change the settings. All options should be self-explanatory, or should be explained in the comments written above them inside the generated file. User can also consult appendix C.

One of the important settings is the texture size. The bigger the size, the longer it will take to generate any planet. On the other hand, the planet will possibly look much better and will appear to have more details. Also various level of detail settings can be tuned to improve the performance.

If something isn't working as expected, or the program terminates unexpectedly, the user can inspect the `log.txt` kept next to the executable. All sorts of information are written to it on every run, so the user can get a clue of what has happened prior the occurrence of a problem, and possibly pinpoint its cause.

Slartibartfast is provided with a sample set of basic terrain textures contained within the `Textures` directory. Deleting or renaming them will result in error messages and premature termination of the program. The user is, however, encouraged to swap the default textures with his own as he wishes. Replacement textures must be named exactly as the default ones and have to be in the bmp format.

4.2.1 Basic controls and camera movement

After the initial planet is generated, the user is presented with a GLSL shaded view of the planet with various information printed in the top-left and the bottom-left corner (as seen in figure 4.1). The default camera (called "homeworld") will always be focused to the centre of the planet. The camera can be zoomed in and out using the mouse wheel. It can be rotated by holding the right mouse button and moving the mouse in the desired direction. In-program help can be viewed in the top-right corner after pressing `F1`. Program can be terminated by either pressing `Esc` key or clicking the cross button (in windowed mode only).

Pressing the `'C'` key switches the camera to the "spectator" mode which allows free looking and moving around using the mouse (again while holding right mouse button) and keys `'W'`, `'S'`, `'A'`, `'D'`. Pressing the `'C'` again will switch back to the "homeworld" camera.

Key `'L'` toggles on/off the dynamic level of detail. Key `'R'` toggles on/off the rotation of the light (visible only in some render modes – see

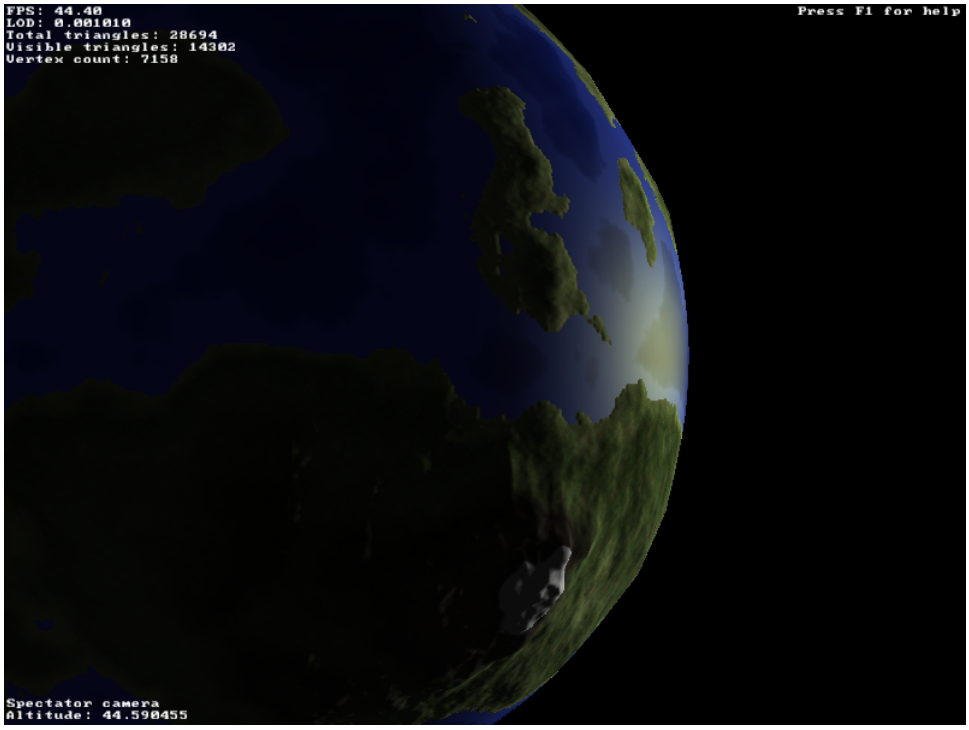


Figure 4.1: Planet rendered with the GLSL shader

further). Keys '1', '2', '3', '4', and'5' (the numbered keys bellow the function keys) switch between wireframe, culled wireframe, grayscale texture, lit grayscale texture, and terrain texture rendering modes.

The first two are good for viewing how the mesh is constructed. The third mode offers view of the heightmap texture and how it is placed to the surface. The fourth mode switches on the lighting; it is good for comparing the per-vertex lighting with the per-pixel lighting used in the fifth mode that uses the GLSL shaders to replace the OpenGL rendering pipeline in order to produce terrain textured surface.

4.2.2 Console

For all non-trivial tasks the *Slartibartfast* implements simple command line interface that can be activated by pressing the '~' (tilde) key. All commands are composed of a single word – a combination of alphanumeric characters and other characters except the '~' (tilde). All arguments are space separated – this disallows filenames or paths (in commands that use them) with spaces. The console doesn't interface with the filesystem or the operating system, nor it supports command auto-

completion.

Planets are generated, saved, and loaded from here. Overview of all available commands can be invoked by executing the `cmdlist` command (by typing in the command and pressing `Enter`). Another useful command is the `help` command. When invoked without parameters, it prints some potentially helpful information. It can also be executed with another command as an argument. That way, the user will get usage info about a particular command he is interested in. For detailed information about all currently implemented commands see appendix B.

4.2.3 Generating a new planet

In order to generate a new planet the user must open the console and execute the `generate` command. At that moment, the old planet is destroyed, and a new planet is generated. During this time the program may become unresponsive for a moment (the length of the lag depends mostly on the specified texture size). The `nparams` command can be used to review parameters that will be used to generate the new planet, while the `params` command lists the current planet parameters.

To change any of the parameters one simply types in the parameter name followed by the new value separated by a space. All parameters accept floating point numbers, except the random number generator seed which is treated as an integer. All parameters are listed and explained in appendix B (second half of the table) or the next section.

One must also choose the preferred generator. Currently, there are two of them implemented. List of all implemented generators can be obtained by executing the `generator` command without any arguments. Executing the same command with a number from the list, as an argument, will set that particular generator to be used for the next planet.

Most of the parameters influence dimensions of the planet and other various aspects. There are however two parameters that may not be clear at first sight. These are the `exponent` and the `w_factor`. They are set by default to 0.5 and 2.0, respectively, and both are tied with the fractal Brownian motion function used in both generators to obtain heights (see section 3.2).

4.2.4 Saving and loading planets

The currently displayed planet can be saved for later usage by typing `save filename` into the console where `filename` is a name of the file (including the path without any spaces) in which the data should be

stored. *Slartibartfast* will try to create a file with the exact filename; however, it won't give any warning when such file already exists, so care must be taken. Planets are saved in a plain text, but hand editing of the files isn't recommended.

To load a previously saved planet, the user should type `load` followed by a path to the file he wants to load. The only inconvenience lies in the fact that the program doesn't support file listing, so the user must know the filename. Both commands will notify the user whether their execution was successful or not. In the latter case, more info why the attempt has failed can be obtained by inspecting the `log.txt`. The program may become unresponsive for a moment during the load operation.

Format of the save files is simple (it is essentially the same format as configuration files – see section 4.3.1). Even the parser is the same. All parameters needed for successful saving and loading are these:

- Generator – defines which generator should be used to generate the planet.
- Radius – the mean radius of the planet.
- Max height – defines maximal/minimal height of the terrain measured from the radius. Care should be taken when specifying it, high values can produce non-pleasing outputs or can get occluded by OpenGL.
- Ocean level – defined as a radius below which we deem the terrain as an ocean floor. Visible in textured rendering via shaders.
- Angular velocity – defines rotation of the planet (in the current implementation rotation of the light source around the planet).
- Seed – random number seed for the noise object inside the planetary generator.
- Weight factor – explained in section 3.2.
- Exponent – explained in section 3.2.

4.2.5 Exporting a planet

For usage outside of this program, every planet can be exported into the Waveform's obj format[3]. This can be achieved by typing `export` into

[3]It is a plain text file with a list of all vertices, normals, edges, faces and other attributes belonging to the mesh – http://en.wikipedia.org/wiki/Obj

the console followed by an integer (N) and a filename. N is the level of subdivision. Level 0 will dump only the basic cube, further levels will dump recursive subdivisions of this cube. Input values of N larger than 12 aren't recommended. This is due the limitation imposed on our implementation of the vertex buffer – it can hold only 65535 vertices. Larger values will thus result in a non-watertight mesh.

Upon export, one file with the `.obj` extension is created. The program then dumps vertex positions, normals and cube-map coordinates along with polygons into the file. Two sets of six bitmap files containing appropriate faces of the cube-map are also created. The first set contains the heightmap and the other contains the normal map.

4.3 Program workflow

This section provides a general overview of the program's code. Figure 4.2 shows a basic workflow of the program. The program's main function obtains an instance of the `Core` class and calls its `Init()` method.

If everything is set up correctly, the control reaches the `Run()` method that implements our main loop. This method also makes sure that all frames are calculated in fixed time steps, thus the program should appear to execute (in the sense of visible movement etc.) at the same speed on all machines that can run it (see section 4.3.2). When the control returns to the `main()` function, we hand it over to the `ShutDown()` method which will clean all dynamically created objects, then the program terminates.

4.3.1 Initialization

The first thing we do during the initialization phase, is an attempt to open and parse the configuration file. Whether this method fails or not doesn't bother us because either way we'll have a valid configuration parameters – in the case of non-existent/corrupted configuration file, or errors in the file, all of the wrong options are set to defaults and then written back before the application exits.

Format of the configuration file is simple: any line beginning with the `'#'` character will be treated as a comment. All other lines are either blank or in the format: `option = value;`. The parser omits all blank lines and comments. Other lines are split by `' '`, `'='`, `';'` characters, and then the parser matches the first word on the line against all known options (see appendix C). When a match is found, it tries to parse the argument and save the value into the appropriate variable. If an invalid

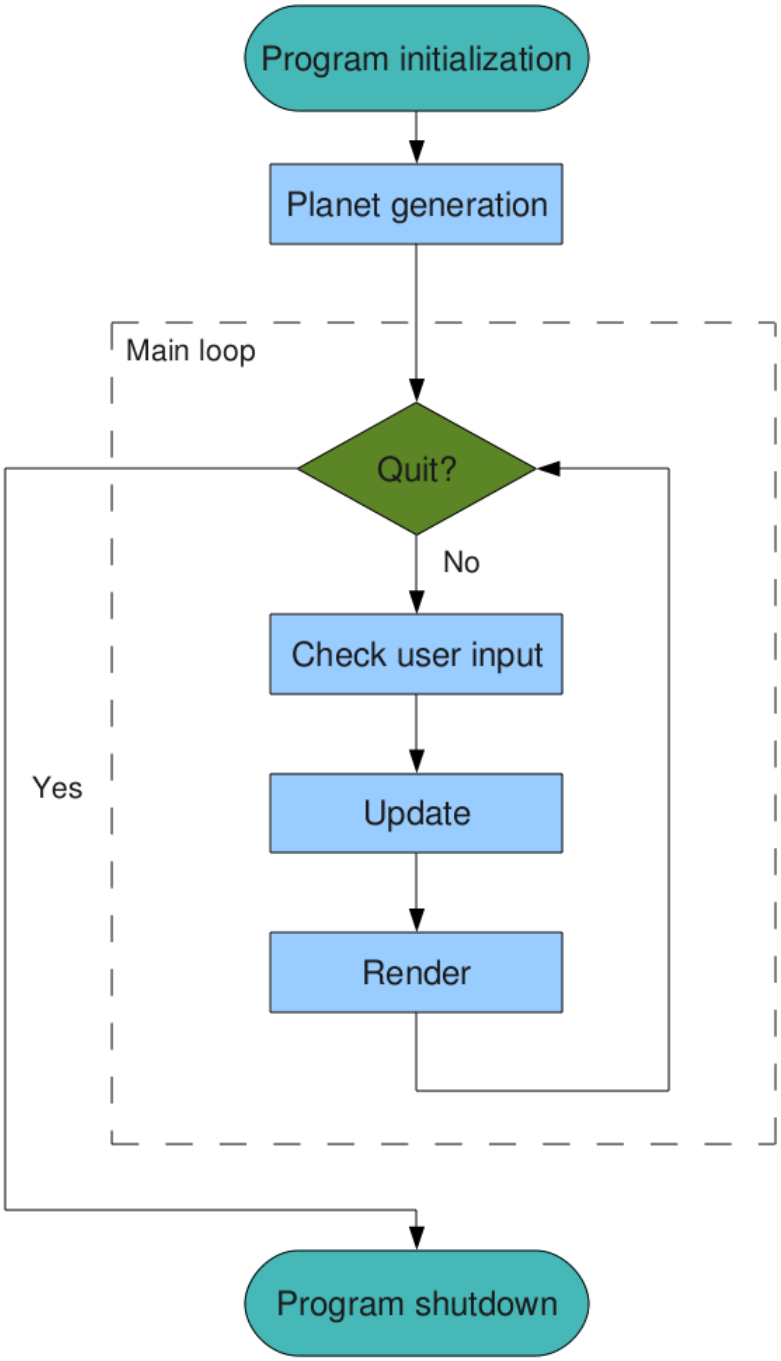


Figure 4.2: Program flowchart

value or syntax error is encountered, it is logged and the parsing resumes. All adjustable variables are initially set to defaults, so if the error occurs, such variable is left in the default state.

When the program obtained its configuration, we need to initialize the Allegro library. This is done through the standard API functions; we prepare keyboard and mouse input. At this point, we try to set up the graphics window. If this fails, the initialization is terminated immediately and the program exits. Notion of what exactly went wrong should be written in the log file.

From this point onward we should be looking into a black window and have the OpenGL rendering context. The vertex buffer should be ready for use; we load textures and compile shaders. Any errors during their loading and compilation are reported to the log file; the user is notified via a pop-up window that the shaders have failed to load and will not be available.

Next on the initialization list is the planet generation. We create the initial planet with predefined attributes and randomly selected seed for the random number generator. This is the longest process of the whole initialization phase, and its length depends on the size of the cube-map texture (see section 3.3.1). User can notice some lag, but he shouldn't be worried. Finally we prepare cameras and the console. If everything went smooth, we are about to enter the main loop.

4.3.2 Main loop

Upon entry into the main loop, we call an Allegro function to initialize the timer. It sets another thread that periodically calls small, user defined function 60 times per second. Inside this function we increment two helper variables – one for frame synchronization and one for updating the FPS counter (implemented as a simple circular buffer used to calculate the average framerate).

Further, we check whether the inner state hasn't changed to the exit state upon which we terminate the main loop. Follows a nested loop that is ideally executed at 60 times per second (unless we are forced to drop frames) thus ensuring a constant logical framerate of 60 FPS. In this loop we calculate all the program logic.

After the logical loop, we do the rendering. If it's faster than $1/60s$, we render the frame again. When the rendering is done, we give back some time to the CPU, unless the user hasn't specified he wants the maximal performance. This loop update method (however, more sophisticated) is, for example, the default method used in the XNA Game

Framework for .NET 2.0[4].

Logic inside the nested loop takes care of the following:

- Checking for the general user input, i.e. control keys. We basically check the state of the keys we are interested in, and execute the logic behind them – like swapping the camera, adjusting the OpenGL parameters, switching to console, or setting the state to exit.
- Updating the planet object, camera, lighting, and the vertex buffer.
- Updating the console if it's open.

Should the user issued planet loading or generating completely new planet, we stop the execution, and return back to the planet generation phase. We then resume the execution of the main loop when the newly created planetary body is ready.

4.3.3 Rendering

We use the visitor pattern to separate rendering from the ROAM algorithm implementation. We call the planet's render method with an appropriate visitor that will fill our vertex buffer (for details see section 4.5) with triangles to be rendered. We then push the vertex data to the GPU – we can't use the VBO [11, p. 93] because of the nature of our data – they change from frame to frame. Therefore, we use only vertex arrays, but we call it the vertex buffer for short (see section 4.5).

When the buffer is filled with polygons to be rendered, we update the projection matrix and call OpenGL API functions to render data stored on the GPU using an index buffer. What functions will be used depends on the selected render mode:

- Wireframe – all triangles are rendered only as edge lines.
- Culled wireframe – same as above, except some primitive calculations are done to occlude triangles on the back of the sphere.
- Grayscale – the planet is rendered as a textured solid. Heightmap texture is cube-mapped to the surface.

[4] As posted by one of its developers on his blog – http://blogs.msdn.com/b/shawnhar/archive/2007/11/23/game-timing-in-xna-game-studio-2-0.aspx

- Lit grayscale – ditto, but the solid is lit using the pre-calculated vertex normals. Good for comparing visual quality with the per vertex lighting that uses a normal map.

- Textured – custom vertex and fragment shader (see section 3.3.2) programs are used instead of the fixed rendering pipeline to produce per pixel lit mesh with terrain texture and oceans.

4.4 Module overview

In figure 4.3 we can see an overview of all important classes and their logical structure. Lines between the classes show interactions, inheritance, and instantiation.

All the blue classes are singletons and serve mainly as utility classes, i.e., they implement setting up the graphics window, user control, and low level rendering (physical vertex pushing through the pipeline). All the light-green classes are abstract interfaces from which we derive actual implementation of these components.

Finally, all the green classes, except the `Camera` class, compose the main kernel of the demo – the actual algorithms for generating a planetary body. Should anybody decide to use these in their work, all he needs to do, is taking the green classes and importing them into their project. Small changes will be necessary as implied by the diagram. These include:

- Method for creating and removing vertices that would fit existing codebase in the target project.

- Physical rendering of the planet – this is actually very easy, since virtually everything needed to achieve this, is deriving a new class from the `RenderVisitor` interface and writing new code into its `Render()` method. Render visitors are instantiated in the `Planet` class. They are passed to the `ROAM_Sphere.Render()` method (not shown in the diagram) and propagated all the way down to the `ROAM_Triangle.Render()` method that serves as an acceptor.

- Passing through the camera position – right now the `Core` class passes down active camera object, so the algorithm knows where is the eye and where is it looking. It uses these information to decide where it should increase visible mesh details and where it isn't necessary.

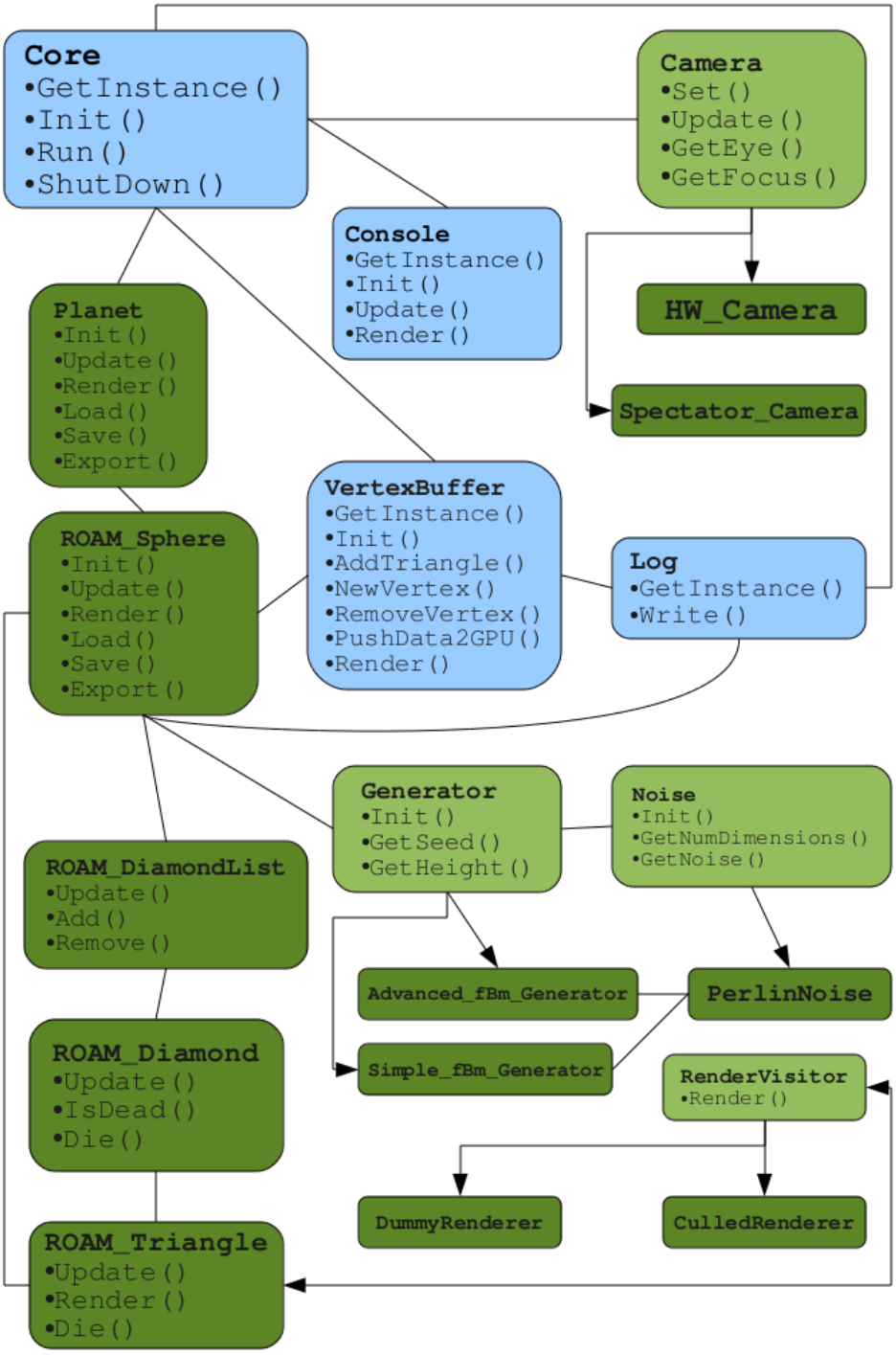


Figure 4.3: Class diagram

- Omitting or changing the logging system – currently it uses very simple method of writing unformatted messages into a file.

Follows a closer look upon some implementation details of the key modules.

4.4.1 Utility classes

The `Core` class encompasses most of the boring, yet necessary, code. It is a singleton that provides interface for initializing, running, and shutting down the graphics engine. If a strict OO design would be followed, it would also provide an interface for controlling the planetary objects. However, a decision to omit such proxy methods has been taken as this program is mostly a demo. So the class is a friend with the `Console` class. I.e., the console can access all data members of the `Core` class (because it serves as the user interface).

The `Core` implements a small finite state machine that decides what to do once inside the logical loop (see section 4.3.2). Possible states are:

- Init – a default state during which we initialize everything. Should this state occur inside the main loop, it is perceived as an error that is reported to the program log and then the application terminates.
- Run – this is the state in which we update planet's level of detail, camera position, etc.
- Console – in this state all keyboard input is used to type in commands in order to control the generator.
- Exit – this state is issued upon pressing the key `'Esc'` or typing `"exit"` into the console. When this state is encountered we terminate the main loop and clear all dynamically created data from memory.

The `Console` class contains basic command line interface implementation. Only the basics are implemented. While limited in functionality, it provides an easy interface that allows the user to generate new planets, load them, and/or save them. The command parser works in the same manner as the configuration file parser (see section 4.3.1).

The `Log` class provides an interface for writing text messages into a log file.

4.5 Vertex buffer

The `VertexBuffer` class hides the implementation of the OpenGL vertex arrays from the user. It contains an array of 65535 vertices (a vertex is an aggregate structure defined in the `vertex.h` header), and provides an interface for their creation or removal. Structures described later require shared vertices; having them stored in one place and in a way that allows usage of the vertex arrays proved to be more elegant, and efficient, than having them scattered all around the memory inside objects with reference counters. This approach is much less error prone and improves the performance a bit because it doesn't use the OpenGL immediate mode – first, it can be slow on some occasions [11, p. 80] and new versions of OpenGL even mark them as deprecated [10, p. 404].

4.5.1 Keeping track of vertices

The vertex structure holds information about its position, colour, texture coordinates, and normal. The `VertexBuffer` class can use all four attributes for rendering through specifying vertex, colour, texture coordinates, and normal arrays – usage of these can be switched on and of through the `Enable*Array` and `Disable*Array` methods. The structure also implements a method for time dependent vertex displacement to reduce artefact known as vertex "popping" which can be very disturbing. It is a form of geomorphing proposed by Duchaineau [2] for his ROAM algorithm.

Each vertex can be in one of several states, e.g., morphing into the position, morphing out, idle, or marked for removal. Upon insertion, the triangle position is set to the midpoint of the split square (see section 4.6.2). For several oncoming frames it is moved to its final position (the coordinates calculated during the split operation). When we are about to remove the vertex, we gradually move it into the midpoint again, and then remove it. Other states do nothing or signalise the `VertexBuffer` class that the vertex has morphed out and can be deleted.

Adding a new vertex doesn't involve any permanent memory consumption because we store them in a static array. All we do is updating the vertex counter, free space information, and the element in the array. Upon successful creation of a vertex (which can be created either blank or initialized) an index into the buffer is provided. Access to the stored vertices is made possible through a method that returns reference to the vertex structure stored under the supplied index.

Because the speed is crucial, no boundary checks are done, so it is

a programmer's responsibility to supply valid index. Obtaining a vertex from any position marked as free space results in an undefined behaviour – a valid reference is always returned, but data can contain anything.

To keep track of the free space (vertex array elements that do not store any active vertex) inside the buffer we are holding a pointer to a position after the last active vertex. Besides that, we keep track of the non-continuous free space (between the beginning and the highest peak) in a priority queue. When we add a new vertex, we first look into the priority queue if there's a space bellow the highest peak. If there is, we pop it from the queue and store the vertex there. This operation is $O(log\ n)$ – we use STL's container `priority_queue` that constructs heap above the supplied vector of numbers. If the queue is empty, or the highest peak is bellow the first element in the queue, we store the vertex on the peak position and move the peak one index to the right.

When we run out of space, we simply return the `V_BUF_SIZE` constant (it denotes the size of the buffer – 65535 vertices in the current implementation).

4.5.2 Rendering

In order to render anything, polygons must be constructed using indices to the vertex array. This is done via the `AddTriangle()` method. The method takes three indices into the vertex array and stores them in the given order into the index array (which is 4 times bigger than the vertex array). A care must be taken to pass vertices in the correct order, otherwise it can happen that the OpenGL pipeline will treat the polygon as back faces and won't render them.

The `PushData2GPU()` method must be called before any actual rendering. It takes all enabled arrays and copies them into the appropriate arrays on the GPU unit. The `Render()` method then takes indices from the index buffer and calls the appropriate OpenGL functions (see section 4.3.3 for an overview of render modes).

The index buffer is then cleared to make space for new frame data. When a new planet is generated, the `Purge()` method is used to get rid of all vertices in the buffer – it resets the free space pointer to the beginning and clears the priority queue.

The `DumpObjData()` method can be used to dump contents of the vertex buffer into an external file in the Wavefront's obj format.

4.6 The ROAM implementation

Our implementation of the spherical ROAM algorithm is spread across several classes designated with the "ROAM" prefix (see figure 4.3). The top level class is then harboured inside the proxy `Planet` class. By swapping this private object for anything else, one can implement his own algorithm instead of using the current implementation while maintaining the same interface.

The sphere object contains 12 ROAM trees – we hold pointers to their roots. We use a DFS algorithm to update all of them every frame. Each frame, we check whether any of the leaves needs splitting. We also need to check, if we can merge some of the triangles. We could do this by checking all nodes inside the tree, but that would also mean we would need to check whether such triangles form a diamond in every frame.

As an optimization we cache all present diamonds (we know that they can only be created during the split and merge operations), This cache is provided via a linked list. We then simply go through all diamonds in the list to check for un-needed triangles.

Triangles and diamonds to be removed are at first marked as dead in the function that finds them unnecessary (either the split or merge operation). They are collected in the next frame by the update method. If the user wants to export the planet, we first reset the sphere back to the initial cube, and then call a special version of the update method that recursively subdivides all triangles to form same level of detail over the whole surface.

4.6.1 Initialization

First, the initialization process calls the `Init()` method on the world generator object that has been supplied via constructor. Then, we create 12 triangles (instances of the `ROAM_Triangle`) – each two composing one face of a cube. These 12 triangles represent roots of a tree structure generated by split operations (see below). This approach literally creates a sphere from a cube.

We prepare 8 primary vertices and assign them to the previously created triangles. Figure 4.4 shows the layout of the vertices and triangles (black numbers denote vertices, red numbers denote triangles, and red lines indicate neighbour relationships).

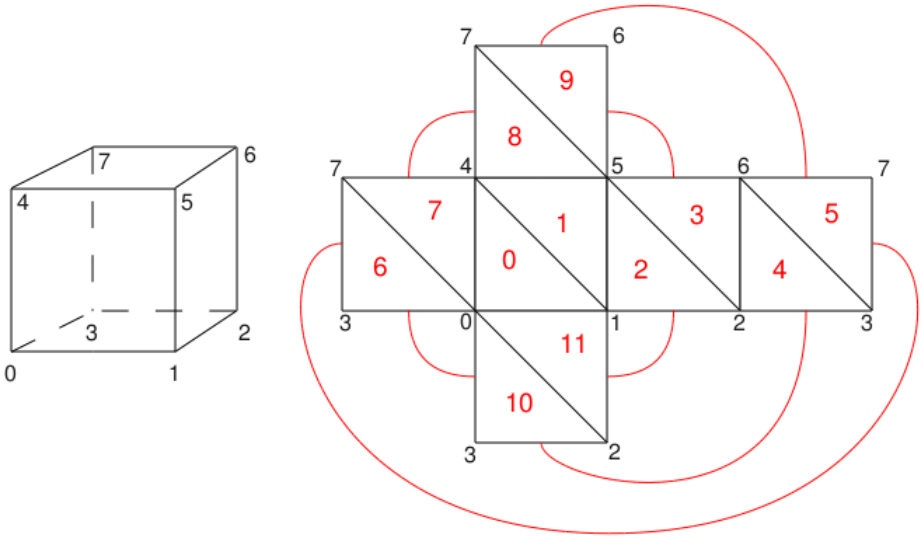


Figure 4.4: Triangle layout

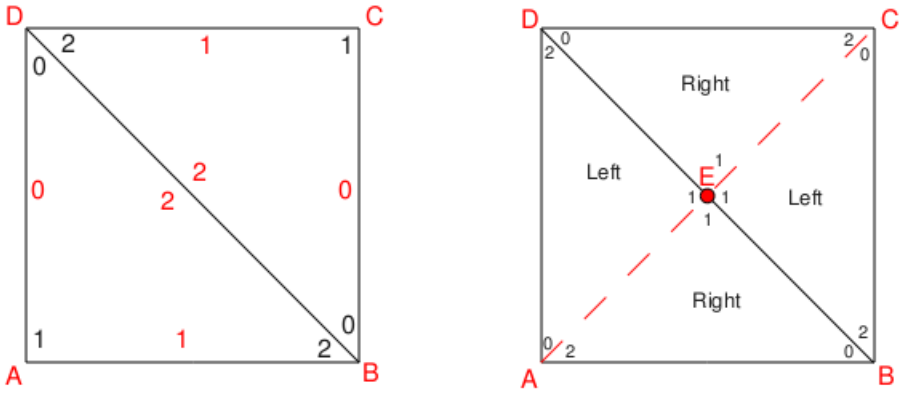


Figure 4.5: Triangle schema

4.6.2 Triangle splitting

Figure 4.5 shows how the situation looks before the split and after the split operation. Black numbers denote local vertex aliases, red numbers denote edge indices. We can see that both triangles on the left side of the figure share 2 vertices (B and D), but have them listed under different local indices. This is because we need to keep the same vertex-edge orientation for all triangles.

Each triangle has a three element array containing indices to the vertex buffer (red letters in the figure would correspond to these). Each

triangle also contains a three element array of pointers towards its neighbours along all edges. We refer to these with numbers shown in the figure (they correspond with vertex designations).

Now, when we split the triangle, the first thing we have to check is whether the triangle forms a square. For this purpose we simply do the following (in order to be as fast as possible, we don't check for null or invalid pointers):

if $neighbour[2]->neighbour[2] = this$ **then**
  return *true*
else
  return *false*
end if

If this test fails, a split method on the neighbour along the longest edge must be called before proceeding furter. Splitting of a triangle introduces new vertex E, so we ask the vertex buffer to give us its index. If there is no space left (the vertex buffer will notify us by returning the maximum capacity as an index), we quit immediately (so far we haven't changed the state of the triangle).

We assign the position (pre-calculated when the current triangle was created – it is obtained as a unit vector pointing towards the new vertex multiplied by radius plus the value returned by the height generator), texture coordinates, and normal (actually the normal is calculated at the end when we have all four new triangles ready, so we can calculate a normal as an average of the triangle normals around the new vertex) to the newly created vertex. We also set its state to morph-in.

Then, we allocate space for both children of the current triangle and the opposite one (neighbour along the hypotenuse). We set parents to these children and assign vertices to them (how it's done can be seen on the right side of figure 4.5). Then, we update the neighbourhood of all four new triangles – so each triangle has valid pointers towards its neighbours along all edges. To do so, we use information contained within their respective parents.

An important step is to check, whether or not the current triangle, or the the opposite one, wasn't a part of a diamond (see section 4.6.3). If so, we need to mark it dead because the split operation destroyed it. Finally, we add a new diamond to the list (the one we have just created). To check whether a triangle is a part of a diamond, we run in circle along one of the short edges. If we get back to the original triangle, we form a

diamond.

Last step depends on the use of recursive rendering (see section 4.6.4). If we do not use it, we add four new triangles to the list of leaves and remove both parent triangles from the same list. An optional way of implementing the tree would be storing just the list of leaves. However, that introduces overhead during merging as we need to re-calculate some values for the parent triangles. On the other hand, it can save memory, as we don't need to hold all nodes between the root and the leaves, and the time spent on the DFS execution (see 5.1).

4.6.3 Triangle merging

When the calculated visible error of a diamond is less or equal than the threshold, we merge it. First, we change the state of the middle vertex to morph out back to its original position. Then, several frames later, the actual merging takes place. We check the state of the vertex, and when it has changed to a state indicating we can remove the diamond from the list, we do so.

The merge operation is the inversion of splitting. We can see the post-merge situation (left) and the pre-merge situation (right) in figure 4.5. When a merge is invoked, we take parents of all four triangles (pairs of them share one parent), and update their neighbourhood using information from the diamond. Then, we take an optional step of removing two leaves from the list of leaves and adding a new for each parent (see section 4.6.4).

We need to check whether either of the parent triangles forms a diamond. If so, we add it to the list. Last three steps of merging consist of recalculation of the vertex normals, removing the vertex E from the buffer (by changing its state to dead), and marking four merged triangles as dead (they will be removed in the next triangle update cycle). We also mark the current diamond as dead.

4.6.4 Rendering the tree

As far as rendering is concerned, there are two possible ways to render the sphere in our implementation. Since we hold the whole tree structure, the first way is via standard recursion. We traverse the tree until a leaf is found, there we call the `Render()` method of the supplied visitor with the leaf as its argument.

The other way is non-recursive. It is achieved by creating a list of leaves updated during the update phase. Then we call the `Render()`

method of the supplied visitor on all elements in this list. Used render method can be switched at the compile time via defining or un-defining preprocessing constant `CFG_USE_RECURSIVE_RENDERING`. It can be used to compare which method is faster.

4.7 Perlin noise implementation

The `Noise` class provides an abstract interface for implementing any n-dimensional persistent noise, in practice we use only 3 dimensions; however, the number can be increased. The class declares virtual method to initialize the noise generator with a seed value, and provides three pure virtual methods for obtaining one, two, and three dimensional noise samples.

The `PerlinNoise` class uses the previously defined interface to implement the Perlin noise. We use 3-dimensional lattice of random numbers to generate the output. Up to three numbers are passed as the input (if less than three numbers are passed, the rest is padded with zeroes). We then take integral and fractional parts of the input values. The integral parts are used as indices to the lattice.

A single index is constructed from all three indices, and a three dimensional vector is taken from the pre-calculated array. All elements are then multiplied by the fractional parts respectively, and a single value obtained as a sum of all vector elements is returned.

For one dimensional noise we take two random numbers from the lattice, and interpolate between them (current implementation uses linear interpolation). To obtain the first number, we use the first element of the input vector as an index to the lattice. To obtain the second, we add 1 to the first element of the vector, and use it as a lattice index.

Two dimensional noise uses four random numbers obtained through the original vector and all possible combinations of adding 1 to its elements. Then, we interpolate the first two and the last two numbers, and then the results. For three dimensional noise, we do the same, but with eight numbers. The interpolation of the pairs yields four values. These are again interpolated, and the results are again interpolated.

Adding another dimension would result in an exponential increase of the values that compose the result. All return values are truncated to $(-1.0, 1.0)$.

Chapter 5

Observations

In this chapter we will discuss some key observations made during the development of this demo. Our key points are the suitability of the chosen LOD algorithm, the effectivity of the used texturing method, and the impact of normal mapping.

5.1 The effectivity of the LOD algorithm

We have decided to use the spherical version of the ROAM algorithm despite the fact that even O'Neil later rewrote his application using another LOD algorithm (a variation on the quad-tree algorithm). The ROAM algorithm isn't a bad choice, but it has one drawback briefly mentioned in section 4.3.3 – changes that occur in our vertex data can be predicted and detected with great difficulty, i.e., the data are too un-organized. Therefore, we are forced to push all vertices and polygons repeatedly to the GPU. Thus we cannot take advantage of the OpenGL vertex buffer objects which would greatly improve the rendering performance.

Another possible improvement would be the storage of the ROAM tree leaves only (as mentioned at the end of section 4.6.2). Some testing could be done to tell whether the DFS algortihm used for tree update causes such overhead that opting for the leaves only implmentation would prove much more effective. In spite of the overhead introduced to the merge operation.

Some time could also be spent on tuning the split and merge priorities. Our current implementation embodies a weird behaviour that can be occasionally visible on the horizon – some triangles get split only to be merged either in the same frame, or in the next one. This results in jittering vertices. Most of the time a small camera movement is enough

to remove this effect, though. The cause is unknown despite the effort dedicated to remove this problem.

5.2 The effectivity of our texturing method

From distance all generated planets look nice. However, on closer look we'll observe that there is something wrong with the scales. Because we use the tri-planar texturing that uses three texture fetches we cannot use dynamic changing of the texture repetition factor. We had to make a compromise between visible texture patterns and a texture scale on a close-up zoom.

We currently use four terrain textures and that makes 12 texture fetches to get the desired result. Should we use dynamic texture scaling that would require blending to hide observable texture movement, we would need 24 texture fetches, and that really impacts the rendering performance.

Even now, the framerate drops bellow 60 FPS when we activate the shaders, and we only try to render around 10 000 polygons. There are several realtime landscape renderers that claim to run at around 60 FPS with much larger numbers of rendered polygons (O'Neil's implementation included). A discussion can be made how big the impact is, and whether the bottleneck doesn't lie somewhere else (see section 5.1).

We also cannot forget about noticeable artefacts (blurred texture) that can occur as a result of tri-planar texturing (they were mentioned in [6, chapter 1]). Noticeable texture stretching is also visible in areas where three cube-map faces get together (this is mainly due to the distortion of the normal map).

Therefore, in order to improve the rendering performance and image quality, another terrain texture method should be sought and implemented.

5.3 The impact of normal mapping

Before switching to the shader approach we have used normals calculated in the program for per-vertex lighting. This hasn't produced nice results because all the terrain features diminished when the planet was viewed from high altitude. Also, as the surface got subdivided during zooming, new vertices were introduced and this resulted in visible vertex "popping".

In order to reduce this uncanny effect we have incorporated geomorphing to our LOD algorithm. However, to prevent high CPU loads, we weren't updating vertex normals as the vertices were slowly moved into their positions, thus the outcome of this effort was nil. The mountains still looked horribly because the per-vertex lighting stressed out the triangles. Every mountain range therefore resembled a shape constructed from pyramids.

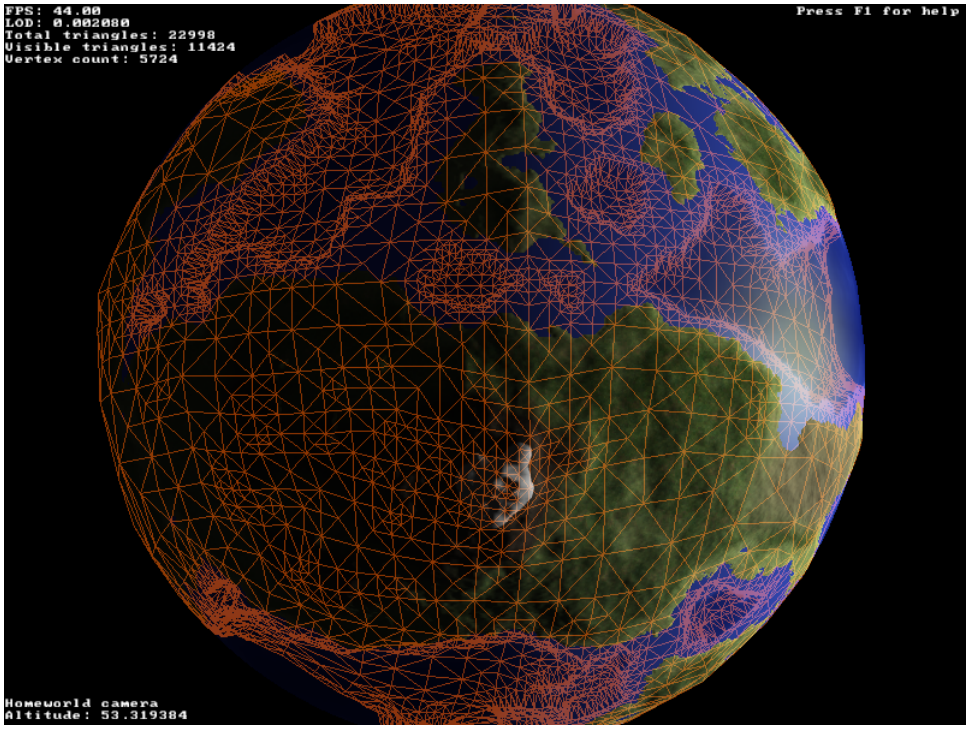


Figure 5.1: Normal mapping with wireframe overlay

The incorporation of normal maps and per-pixel lighting together with water reflection had really positive effect to all previously mentioned visual issues. First, the terrain features are clearly visible, although, the surface is composed only from several triangles. An example can be seen in figure 5.1.

Second, the vertex "popping" is now virtually invisible from high altitudes and nearly un-noticeable in close-up zooms during camera movement.

Chapter 6

Conclusion

In this work we have tested a method of procedural generation of a spherical landscape. To do so, we have implemented an adaptation of the ROAM algorithm and supplemented it with a shader based solution for generating terrain texture. We have experimented with some fractal based functions to transform generated noise values. This area provides a wide open space for further research as we have only touched the surface of the world of multifractals.

We have also confirmed that lighting has the biggest impact on the overall quality of the generated imagery. Scenes without lighting lack details that should be visible. We have also observed that the per-vertex lighting isn't enough for providing visually pleasing results. Normal mapping proved to be simple, easy-to-implement, yet powerful solution for many uncanny lighting effects we have observed. It improved the overall quality of produced images by several orders of magnitude.

If we were to write, for example, a free space simulation that would allow the player to travel from one star to another, where the player would watch planetary bodies from the orbit only (a clone of *Elite*), we could easily use our current implementation as it provides sufficient results.

Should we, on the other hand, write an application where the player could freely zoom to the surface and observe a wildlife, for example, further work would be needed. Nevertheless, we would have a solid base to begin with.

6.1 Further work

Our program serves as a demonstration of what is possible with several numbers and mathematical functions. There is, however, an open space

for further improvements. Topics for future work include:

- Optimization of the current ROAM algorithm implementation that would use only the list of leaves, so we would not be storing and traversing the whole tree structure.
- Implementation of a better LOD algorithm that could take advantage of the OpenGL vertex buffer objects.
- Improvements to the scale of the planet – so the user would really feel like he is descending onto a planet surface.
- Use of other texturing techniques instead of the tri-planar texturing.
- Implementation of a water shader for more visually appealing water surfaces.
- Normal map generation code improvements.
- Implementation of basic terrain texture generation in the code – for the time being we use sample textures obtained from the Internet.
- Introduction of the axial tilt of the planet and improvements to the weather zoning.
- More experimentation with the world generator.
- Rendering a star field and the sun.
- Implementation of a realtime atmospheric light scattering and a cloud layer.
- Introduction of scene anti-aliasing for better visual quality.
- Implementation of a graphical user interface.

Appendix A

List of control keys

General:	
`'Esc'`	Exits the program.
`'F1'`	Shows/hides help.
`'~'`	Opens/closes the command console.
`'R'`	Toggles light rotation.
`'L'`	Toggles dynamic level of detail.
`'='`	Increases level of detail.
`'-'`	Decreases level of detail.
Viewport control:	
`'C'`	Switches camera type.
`'W'`	Moves camera forward (spectator camera).
`'S'`	Moves camera backward (spectator camera).
`'A'`	Moves camera left (spectator camera).
`'D'`	Moves camera right (spectator camera).
Render mode switching:	
`'1'`	Switches to the wireframe mode.
`'2'`	Switches to the activates culled wireframe mode – faces on the back of the sphere are culled.
`'3'`	Switches to the grayscale mode – solid rendering using heightmap texture.
`'4'`	Switches to the lit grayscale mode – solid rendering using heightmap texture with per-vertex lighting.
`'5'`	Switches to the texture mode – solid rendering using custom rendering pipeline (shaders) and per-pixel lighting.

Appendix B

List of console commands

Command	Parameters	Description
`cmdlist`	–	Prints out list of available commands.
`exit`	–	Immediately quits the program.
`export`	N *filename*	Exports the planet in the `.obj` format as the N-th subdivision of the basic cube into the specified file.
`generate`	–	Generates new planet according to the specified parameters (see below).
`generator`	[G]	When executed without parameter, it will print out numbered list of all available generators. Supplying a parameter sets generator G to be used for the next planet generation.
`help`	[*command*]	Prints the basic help. If supplied with command name, it prints help for that particular command.
`load`	*filename*	Tries to load the specified planet.
`save`	*filename*	Saves current planet into the specified file.
`nparams`	–	Prints out parameters for the new planet.
`params`	–	Prints out parameters of the current planet.
`radius`	*(float)*r	Sets radius of the new planet to r.
`ocean_level`	*(float)*l	Sets ocean level of the new planet to l.

Command	Parameters	Description
`max_height`	*(float)*h	Sets maximal height of the new planet to h.
`seed`	*(int)*s	Sets random seed for the new planet to s.
`omega`	*(float)*o	Sets the angular velocity of the new planet to o.
`exponent`	*(float)*e	Sets the exponent used in the fBm algorithm of the new planet to e.
`w_factor`	*(float)*w	Sets the weight factor used in the fBm algorithm of the new planet to w.

Appendix C

Configuration file options

Option	Value	Description
`width`	*(int)w*	Sets graphics window width to w.
`height`	*(int)h*	Sets graphics window width to h.
`color_depth`	*(int)d*	Sets the colour depth to d bits. d should be either 8, 15, 16, 24 or 32.
`field_of_view`	*(int)fov*	Sets graphics window field of view to fov. Should be between 0 and 180 degrees.
`enable_fullscreen`	*(int)*	Toggles full screen mode on/off (1/0).
`enable_max_performance`	*(int)*	Toggles the laptop unfriendly mode on/off (1/0). If set to 1 the program eats as many system resources as it can get.
`texture_size`	*(int)sz*	Sets cube-map texture size to $sz * sz$. Note that there are totally 18 textures generated of this size. The bigger the size the longer it will take to generate the planet.

Option	Value	Description
`max_tree_depth`	emph(int)	Maximal depth of the ROAM tree. High depth means more detailed geometry, however, extremely high depths may result in vertex buffer overrun which will result in non-watertight mesh and sudden framerate drop.
`enable_dynamic_LOD`	*(int)*	Toggles the dynamic level of detail on/off (1/0).
`default_threshold`	*(float)t*	Sets the default triangle split threshold (level of detail) to t. Can be anything between 0.0 and 0.1.
`min_threshold`	*(float)min*	Sets the lower bound for the split threshold to *min*. Can be anything between 0.0 and 0.1
`max_threshold`	*(float)max*	Sets the upper bound for the split threshold to *min*. Can be anything between 0.0 and 0.1

Bibliography

[1] Clasen M., Hege H.-C., *Terrain Rendering using Spherical Clipmaps.* Eurographics, 2006.

[2] Duchaineau M. et al., *ROAMing Terrain: Real-time Optimally Adapting Meshes.* Proc. Visualization '97, 81-88, 1997.

[3] Elias H., *Spherical landscapes*, [Online], http://freespace.virgin.net/hugo.elias/models/m_landsp.htm

[4] Losasso F., Hoppe H. *Geometry clipmaps: Terrain rendering using nested regular grids.* Proc. SIGGRAPH 2004, 769-776, 2004.

[5] Nicholson K., *GPU based algorithms for terrain texturing* University of Canterbury, Christchurch, New Zealand, 2008.

[6] Nguyen H., et al., *GPU Gems 3.* NVIDIA Corporation, USA, 2007.

[7] O'Neil S., *A Real-time procedural universe, part one - generating planetary bodies.* [Online], www.gamasutra.com, 2001.

[8] O'Neil S., *A Real-time procedural universe, part two - rendering planetary bodies.* [Online], www.gamasutra.com, 2001.

[9] Perlin K., *An image synthesizer.* Proc. SIGGRAPH 85, 287-296, 1985.

[10] Segal, M., Akeley K., *The OpenGL Graphics System: A Specification*, [Online], www.opengl.org, Version 3.0, 2008.

[11] Shreiner D., Woo M., Neider J., Davis T., *OpenGL: Průvodce programátora (authorised transl. of* OpenGL programming guide: The official guide to learning OpenGL, version 2, 5th edition*).* Computer Press, a.s., Brno, 1st edition, 2006.