Czech Technical University in Prague

Faculty of Electrical Engineering

Bachelor thesis

Generation of planetary models by means of fractal algorithms

Ondřej Linda

Thesis supervisor: Ing. Jaroslav Sloup

Study program: Electrical Engineering and Informational Sciences

Specialization: Computer Science

August 2007

Acknowledgments:

Mainly I would like to thank to main supervisor Ing. Jaroslav Sloup who offered me this topic for a thesis and thus evoked my interest in fractal algorithms and terrain generation. He also helped me and gave me a lot of valuable advices during our consultations. Also I would like to thanks to Associate Professor William Henry Hsu who was supervising me on this project during my stay at Kansas State University and gave me a lot of good references to litereture.

Prohlášení

Prohlašuji, že jsem svou Bakalářskou práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.
Nemám závažný důvod proti užití tohoto školního díla ve smyslu § 60 Zákona c.121/2000Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 20. srpna 2007 .....

Abstract

The goal of this bachelor thesis was to investigate existing fractal algorithms for generation of a complex planetary model. That means algorithms for terrain generation and algorithms for terrain coloring.

In order to further investigate the capabilities of these algorithms we created a testing application with user interface. This application enables a user to investigate both the differences among different algorithms and the influence of some key parameters on the planetary model.

For overall optimization of the application and for the ability to explore the planetary model in a higher detail, we implemented a data structure and an algorithm for dynamic level of detail.

Abstrakt

Cílem této bakalářské práce je prostudovat existující fraktální algoritmy vhodné pro generovaní komplexního planetárního modelu. Jedná se tedy o algoritmy vytvářející povrch planety a algoritmy vytvařející obarvení daného terénu.

Za účelem bližšího prozkoumání možností daných algoritmů, jsme vytvořili testovací aplikaci s uživatelským rozhraním, která umožní prostudovat jak rozdíl mezi použitými algoritmy, tak i vliv klíčových parametrů na výsledný planetární model.

Pro optimalizaci celé aplikace a možnost detailnějšího prozkoumaní vygenerovaného terénu jsme vytvořili datovou strukturu a algoritmus umožnující dynamickou úroveň detailu.

Contents

LIST OF FIGURES..... - xii -

LIST OF TABLES..... - xiv -

1 INTRODUCTION 1
1.1 Generation of Planetary Models 1
1.2 Existing Applications 1
1.3 Goals of Our Work 3

2 SURVEY OF EXISTING ALGORITHMS 4
2.1 Inspiration by Real World 4
2.1.1 High Scale Features 4
2.1.2 Low Scale Features 5
2.2 Introduction to Fractal Algorithms 6
2.2.1 What Is a Fractal? 6
2.2.2 Perlin Noise Algorithm 7
2.2.3 What Is a Multifractal? 10
2.3 Algorithms for Terrain Generation 10
2.3.1 Random Faults 11
2.3.2 Mid-Point Displacement 11
2.3.3 Mid-Point Displacement Multifractal 13
2.3.4 Perlin Noise 14
2.3.5 Perlin Noise Multifractal 15
2.3.6 Perlin Noise Ridged 16
2.3.7 Perlin Noise Ridged Multifractal 18
2.4 Algorithms for Terrain Coloring 18
2.4.1 Linear Interpolation 18
2.4.2 Spline Function 20
2.4.3 Perlin Noise 21
2.4.4 Perlin Noise Variations 22
2.4.5 Perturbation 23

3 ANALYSES 24
3.1 Planet Structure 24
3.1.1 Dynamic Structure 24
3.1.2 Hierarchical Structure 24
3.1.3 Linked List Data Structure 24
3.2 Dynamic Level of Detail 25
3.2.1 Level of Detail of Perlin Noise Algorithm 25
3.2.2 Only What We Can See Matters 25
3.2.3 Efficient Dynamic Level of Detail 26
3.3 Analyses of the Application Architecture 26
3.3.1 Modules 26
3.3.2 Application Architecture 27

4 IMPLEMENTATION 28
4.1 Development Framework 28
4.2 Used External Libraries 28
4.3 System Architecture 28
4.4 Mesh Data Structure 29
4.4.1 Planet Representation 29

4.4.2 Data Structure 31
4.5 Dynamic Level of Detail 33
4.5.1 Testing Triangles 33
4.5.2 Updating Mesh 36
4.5.3 Problems and Known Issues 38
4.6 Terrain Generation 39
4.6.1 Water Layer 39
4.6.2 Craters Implementation 40
4.7 Terrain Coloring 42
4.7.1 Altitude Based Coloring 42
4.7.2 Altitude Based Coloring with Perturbation 42
4.7.3 Perlin Noise Coloring 42
4.7.4 Altitude Based + Perlin Noise Coloring 43
4.7.5 Turbulence Coloring 43
4.7.6 Earth-like Coloring 43
4.7.7 Gradient Based Coloring 43
4.7.8 Moon-like Coloring 43
4.8 Graphical User Interface 45

5 COMPARISON OF ALGORITHMS 46
5.1 Comparison of Terrain Colorings 46
5.1.1 Coloring of Low Scale Features 46
5.1.2 Parametric Control of Coloring Algorithms 48
5.2 Comparison of Generated Terrains 49
5.2.1 Differences between Planets 50
5.2.2 Differences in Low Scale Features 52
5.2.3 Parametric Control of Terrain Generation Algorithms 54
5.2.4 The Moon 57
5.3 Comparison of Static and Dynamic Level of Detail 58
5.3.1 Visual Comparison 58
5.3.2 Quantitative comparison 59

6 CONCLUSION AND FUTURE WORK 62
6.1 Conclusion 62
6.2 Extensions to the Current Application 62
6.3 Future Development 63

7 REFERENCES 64

A USER MANUAL 67
A.1 Hardware Requirements 67
A.2 Installation of the Application 67
A.3 Operating Manual 67
A.3.1 Opening Screen 67
A.3.2 Change Shape 68
A.3.3 Random Faults 68
A.3.4 Planet Coloring 68
A.3.5 Method of Iteration 69
A.3.6 Perlin Noise 69
A.3.7 Dynamic Level of Detail 69
A.3.8 Craters 70
A.3.9 Water Level 70
A.3.10 Others 70
A.3.11 Additional Control 70

A.3.12 Keyboard Control.....71

B CONTENT OF THE ENCLOSED CD..... 72

List of Figures

1.0 Terragen's workspace……..- 2 -
1.1 Images from Terragen 2
1.2 Images from TerraJ 3
2.0 Examples of planets……- 4 -
2.1 Earth's vegetation zones....5
2.2 Terrain formations on Earth....5
2.3 Examples of mountains 6
2.4 Examples of fractals....6
2.5 The basis function of the Perlin Noise function....7
2.6 Basis function at different scales 8
2.7 The Perlin Noise function....8
2.8 2D Perlin Noise function 9
2.9 Terrain generated with a multifractal function 10
2.10 Terrain generation with Random Faults algorithm 11
2.11 Increasing the resolution of a triangle. 12
2.12 Terrain generation with Mid-Point Displacement algorithm....13
2.13 Interpolating methods....15
2.14 Smoothing of the function 15
2.15 Perlin Noise Ridged function....17
2.16 Comparison of planets generated with normal and Ridged Perlin Noise 18
2.17 RGB color space 19
2.18 Coloring using linear interpolation 20
2.19 Spline function....20
2.20 Coloring using the Spline function 21
2.21 Coloring using Perlin Noise function. 22
2.22 Coloring using Turbulence. 23
2.23 Coloring using Spline function with Perturbation....24
3.0 The architecture of the system.……...- 27 -
4.0 Octahedron as a starting object for sphere generation.……...- 30 -
4.1 Iterative generation of sphere 30
4.2 Two neighboring triangles sharing an edge....30
4.3 The hierarchy of triangles and points....31
4.4 Implementation of linked lists holding the mesh structure....32
4.5 Four planes bounding our view....34
4.6 Triangles in our view....34
4.7 Examples of front and back side decision rule. 35
4.8 Thresholds for the level of detail. 36
4.9 Splitting triangles causing crack to occur....37
4.10 Comparison of terrain patches with and without cracks....38
4.11 Adding water layer into the model....40
4.12 Examples of moon craters 40
4.13 Schema of different zones inside the crater. 41
4.14 The same planet colored with different coloring algorithms 44
4.15 Planet colored with Moon-like coloring. 44
4.16 Example of the appliacation's work space 45
5.1 The same terrain patch colored with different coloring models 47
5.2 Altitude based coloring with perturbation 48
5.3 Different values of lambda for the Perlin Noise function....49
5.4 The dependency of the coloring on the amplitude of the perturbation function 49
5.5 Planets generated with different algorithms 50
5.6 Terrain patches generated with different algorithms....53
5.7 The size and distribution of continents of Mid-Point Displacement algorithm 54
5.8 Different values of lambda and the number octaves of Perlin Noise algorithm 56

5.9 Number of octaves of multifractal Perlin Noise algorithm.....57
5.10 Moon-like planet. 57
5.11 Comparison of static and dynamic level of detail. 58
5.12 Zooming in with the Dynamic Level of Detail Algorithm. 59
5.13 The dependency of the number of triangles in the mesh on the level of detail.....60
5.14 The dependency of the number of triangles on our distance from the surface 61

List of Tables

Table 1: Random Faults Algorithm. 11
Table 2: Mid-Point Displacement Algorithm....12
Table 3: Mid-Point Displacement Multifractal Algorithm....13
Table 4: Perlin Noise Algorithm....14
Table 5: Perlin Noise Multifractal Algorithm 16
Table 6: Perlin Noise Ridged Algorithm....17
Table 7: Linear Interpolation of RGB color 20
Table 8: Spline Function 21
Table 9: Linear Interpolation using Perlin Noise....22
Table 10: Linear Interpolation with Perturbation 23
Table 11: Dynamic Level of Detail Algorithm....33
Table 12: Split() and Reduce() functions 37
Table 13: The number of triangles in the mesh structure with a certain level of detail. 60

1 Introduction

This is an introduction chapter explaining the main principles and motivations of this work. It describes the reasons why we have chosen a planetary model generation as a field of study and why we are specializing in the fractal algorithms as the mean of their creation. It also presents a short study of the state of the art in this area and lists the main goals of this thesis.

1.1 Generation of Planetary Models

Terrain generation is a very important part of computer graphics. For instance we can see various terrain models in many computer games or computer animations. In the majority of these cases a realistically looking model of terrain is the desired outcome. Obviously we can use real world height maps or manually create the whole terrain model; however those approaches have many limitations. Using real world height maps limits us in the number of different terrain models that we can create and we must somehow obtain the real world data. Manual creation enables us to create unlimited number of terrain models, but creating a realistically looking complex model of terrain would be very time consuming if not impossible.

This is especially true if we are considering creating a complex terrain model for a whole planet. In order for the planet to look realistically, we must take care of the right distribution of continents and islands, the right distribution of jagged mountains and smooth lowlands, and emphasize these features by the right coloring scheme. Fortunately procedural approach to terrain modeling and using fractal algorithms in particular outcomes the mentioned problems and gives us complex terrain models with the desired properties.

Procedural approach

In the procedural approach we do not have to specify and code all the details of the model. All this knowledge is contained in a function or a procedure. Hence we can ask the procedure for some output only where and when we need it. This provides us with a savings in memory space and probably even more importantly, it requires the programmer only to write the procedure and then let the computer power to evaluate the procedure and create the terrain model.

In addition, the result of the procedure's evaluation is very easy to be influenced by adjusting the input parameters. For instance by adjusting one single parameter we can control the size of the continents or the roughness of the mountains. This helps us find the right settings for the algorithms and create a terrain with a desired look and it also enables us to create all kinds of various terrain models with a different features and properties.

Fractal algorithms

When we are looking at any photographs of some real world landscape, we can see that a real natural world is characterized by a high complexity and randomness. These are exactly the same properties that we request from our procedural algorithm for terrain generation. Up to this date fractal algorithms seam to be the best known way for creating a complex terrain model that does contain sufficient amount of randomness and a high complexity into some extent. Definitions and further explanation of fractal algorithms will follow in Chapter 2.

1.2 Existing Applications

There are a lot of commercial and non-commercial applications for terrain modeling. A good source mapping the development of all kinds of software projects and resources in this area could be found in [21]. Most of those software applications try to visualize the real world data, for example Google Earth [22] or Earth3D [23]. But we are interested in artificially generating a realistically looking terrain not based on the real world data. The majority of the other applications generate only a small terrain patch on a 2-dimensional grid. Although these applications use in many cases very

similar algorithms as the ones that we are interested in, we are more concerned about a creation of a complex planetary model. Probably the most typical member of this group of software applications and the most well known terrain modeling program is Terragen [24]. It enables the user to create a terrain patch with several terrain generation algorithms and to control the terrain generation by adjusting a big variety of parameters. We can freely explore the terrain and photo-realistically render the selected view. Right now there is an ongoing development of Terragen 2, which is supposed to be capable of creating a whole complex planetary model. A snapshot of the Terragen's workspace is shown in Figure 1.0.

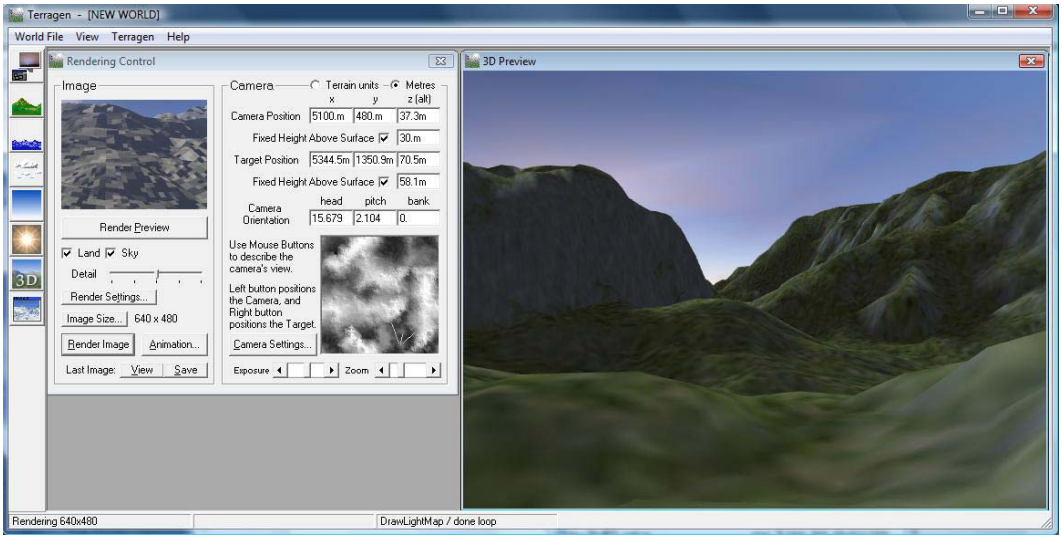


Figure 1.0: The workspace of Terragen.

In order to see what Terragen's rendering engine is really capable of we can see examples in Figure 1.1. But here we must emphasize that the goal of this thesis is not to study various rendering techniques and implementing them. We are more concerned about the terrain generation of the planetary model itself. The rendering engine would be a topic for a whole different thesis.

Figure 1.1: Examples of terrains generated with the Terragen software. (Images taken from [24])

There are several other applications that generate a whole planetary model. One representative of this group is the TerraJ application[25]. Examples of planetary models generated with TerraJ software are shown in Figure 1.2.

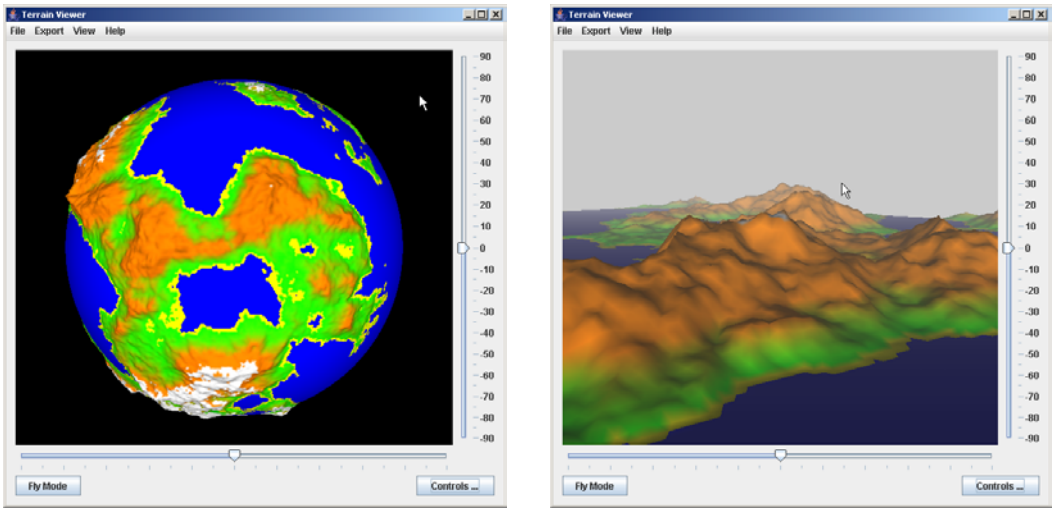


Figure 1.2: Terrain generated with TerraJ software. (Images taken from [25])

1.3 Goals of Our Work

Our goal is to do a survey of fractal algorithms that could be used for terrain generation and see how they could be applied to a generation of a complex planetary model. We want to create a testing application that will let us observe the differences between various algorithms and that will enable us to see the changes in the terrain caused by changing the values of some key parameters.

Rather than generating a complex planetary model in a single step, we want to have a free hand during the creation and to be able to combine different algorithms together. This way we want to create a framework where a planet can be created with one algorithm and then used as an input for another terrain generation algorithm.

In order to see the true power of fractal algorithms we want to add the Dynamic Level of Detail algorithm to allow the user not only to see the whole planet but also to zoom in and see a detail of the terrain anywhere on the planet.

2 Survey of Existing Algorithms

This chapter contains a description of the most important fractal algorithms for terrain generation and for terrain coloring. Before we look at the actual algorithm we will analyze what we are actually trying to create. We will look for typical terrain and coloring features on Earth and other planets. Afterwards we will go over some basic definitions, principles and terminology related to fractal algorithms, which is necessary for further understanding of the presented algorithms.

2.1 Inspiration by Real World

When we are looking at different planets we can see that there are literally millions of different terrain formations and different color combinations. We can just compare how different are Earth, Moon and Mars looking in Figure 2.0.

(a) *(b)* *(c)*

Figure 2.0: Planets. (a) Earth. (b) Moon. (c) Mars. (Image 2.0a taken from [14], Image 2.0b taken from [15], Image 2.0c taken from [16])

For our project we will mainly concentrate on generation of Earth-like looking planets. As Figure 2.0a shows a very important feature of Earth is the atmosphere. We will omit this feature of the planetary model in our implementation and leave it for a future work.

What is so stunning and amazing about Earth and other planets is that they are so complex and so complicated. We can look at Earth from a long distance and see its round shape with visible continents and oceans. If we zoom closer we start seeing mountain ranges, deserts and lowlands. Once we get close enough we can see that there are hills covered with forest and jagged mountains covered with snow. We would like our model to have the same properties. We should not be just interested in a high scale features like the distribution of continents and oceans. But we should also care about all kinds of low scale features that become visible as we approach closer to the surface. Creating a planetary model with both high and low scale features is the goal of our work. The complexity of the model should be our main concern.

2.1.1 High Scale Features

In Figure 2.1 we can clearly see the distribution of continents on Earth. We can notice what the ratio between continents and water approximately is. Also we should note how diverse the shoreline is. There are parts of the terrain where the shore is quite uniform like the coast of western Africa. But we can find parts of the world like south-eastern Asia or northern Canada where the shore is very wildly divided with many islands of different sizes along the coast. All of those features should be incorporated into our planetary model.

From the coloring point of view if we look at Figure 2.0a or at Figure 2.1, we can notice that there is a certain vegetation profile. Because of the differences in the climate caused by different intensity of the solar radiation and other climatic influences, there is a certain vegetation mixture presented in each part of Earth's surface. In addition we can see that this mixture is quite unique and symbolic for certain latitude zones.

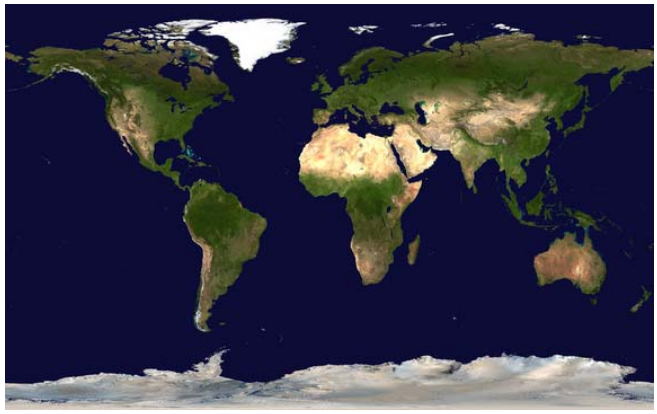


Figure 2.1: Earth's vegetation zones. (Image taken from [17])

We can see that there is a zone of green rain forests around the equator. It is followed by zones of deserts and plains with khaki–ochre color. Then we can see mild green forests that are slowly changing into grayish tundra. The last part of Earth near the poles is heavily covered with white snow and ice. This all shows us that a good coloring model will have to take into account the latitude of the point on the surface of the planet.

2.1.2 Low Scale Features

When we zoom closer to the surface, we can see all the different low scale features colored with all kinds of different colors. This is the moment where the surface of Earth really becomes diverse and we can see an unbelievable amount of different terrain formations. From all kinds of different sharp mountains, through round shaped hills to wide opened flat plains, we can see almost any shape somewhere on Earth. Just a little example of the diversity could be seen in Figure 2.2.

Figure 2.2: Terrain formations on Earth. (Images taken from [26], [27] and [28])

Realistically no artificially created planetary model can get anywhere near any real planet. Our model will always be just a very rough approximation. But our inspiration from the real world for the terrain generation should be its diversity, complexity and randomness. Only planetary model with those features could look at least a little bit naturally and realistically.

The diversity and randomness could be also seen in the coloring of the Earth's surface. We can find mountains and rocks of almost any color on Earth. Also there is a whole spectrum of green forests. A big portion of Earth is covered with deserts, which can also maintain almost any color at least a little bit similar to the color of sand or rock. Although this seems that the color of the Earth is more or less random on different places on Earth, there are some basic principles and patterns. As well as we described the latitude dependency of the vegetation zones, we can see an altitude dependency at the smaller scale. This could be seen on the examples in Figure 2.3.

(a) *(b)*

Figure 2.3: Examples of mountains. We can easily see how the color is changing with the altitude. (Image 2.3a taken from [18])

Thrse two examples clearly show the dependency of the color on the elevation. It changes from plains into mountain foothills covered with forests. As we go higher the forests are disappearing and the mountains surface is more and more covered with snow. This observation suggests that our coloring model should be somehow dependent on the elevation.

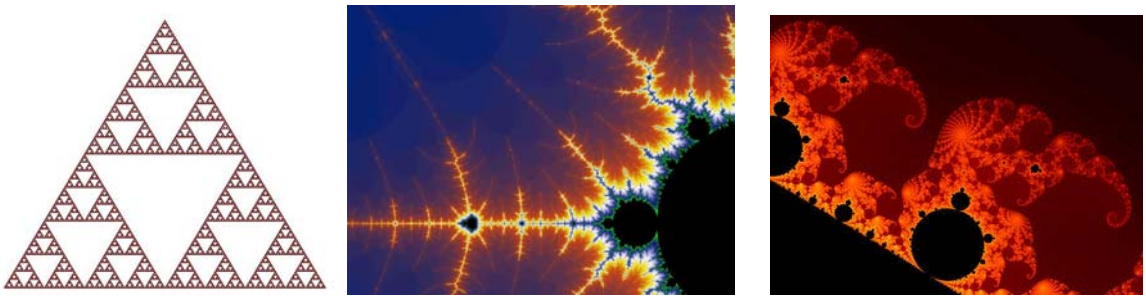
Another thing that is apparent at a closer look is that the trees do not grow when the terrain is too steep. The same works with the snow cover. The snow simply slides down from areas that are too steep. This observation implies that the terrain color should be also dependent on the gradient of the terrain.

2.2 Introduction to Fractal Algorithms

As it was mentioned in the introduction part, fractal algorithms are capable of creating a terrain model with the desired properties of randomness and high complexity. They play very important role in computer graphics and in computer animation. This section is a brief description of what a fractal is and what its most important properties are.

2.2.1 What Is a Fractal?

According to [1] a fractal is: "a geometrically complex object, the complexity of which arises through the repetition of form over some range of scale". The best way to explain a fractal is to give an example from a real world. If we look at a mountain from a long distance or if we hold a piece of its rock in our hand, they both have very similar appearance. We can see the same features and the same roughness of the surface, even though we look at them at a very different scale. This is exactly what a fractal is. The fractal keeps its features over different scales; it is self similar. Some fractals could be quite simple while others create very complex formations as shown in the Figure 2.4. For more information about fractals see [2].



(a) *(b)* *(c)*

Figure 2.4: Examples of fractals. (a) Sierpinski's Gasket (b) Mandelbrot's set (c) Detail of Mandelbrot's set (Images taken from [10], [12] and [11]

2.2.2 Perlin Noise Algorithm

One of the key fractal algorithms not only for terrain generation is the Perlin Noise algorithm. It was first introduced by Ken Perlin in his paper [4] in 1985. Because this algorithm is considered to be the most important one, we will look at it in this section about introduction to fractal algorithms and it will be used for further explanation of some important terms and features of fractals. For simplicity we will now describe only one dimensional version of this algorithm. More detailed description of Perlin Noise algorithm could be found in [2].

Basic function

The most important thing that influences the final appearance of any fractal is its basic function. It defines the basic element of fractal's self similarity. The Perlin Noise algorithm uses a seeded random number generator to create its basis function. Unlike most of random number generators, this random number generator has to output a random value as a function of some input. In other words for the same input the generator will always output the same value, unless different seed is used. If we use this random number generator to generate values in some discrete equidistant points we might get for example a response similar to the one shown in the Figure 2.5a.

If we use some kind of interpolation technique between those values we will get a continuous function that returns some real value for any input point from the chosen interval. This is the basic function of the Perlin Noise algorithm. It is shown in Figure 2.5b.

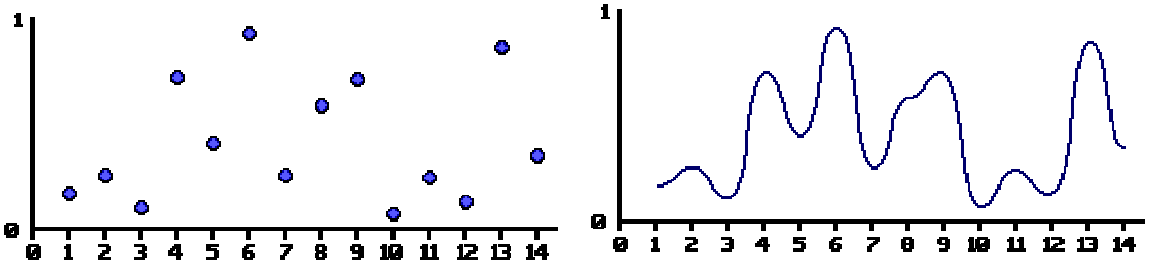


Figure 2.5: The basis function of the Perlin Noise function. (a) Random output value for each discrete point. (b) Smooth interpolation between the points. (Images taken from [3])

For the purpose of terrain generation it is useful to scale the output of the seeded random generator so that it produces values within the interval [-1 ; 1]; So from now on, we will assume that the values are from this interval, unless specified differently.

Repetition over different scales

In the definition of a fractal it is said that there is a repetition of a form, which means the basis function, over some range of scales. If we want to transfer the basis function into another scale we have to appropriately scale its frequency and its amplitude. Although we can use different ratios for scaling them, the most common way to obtain the basis function at a lower scale is to double the frequency and to reduce the amplitude into one half. This way we can obtain a sequence of the same basis function at different scales as shown in Figure 2.6.

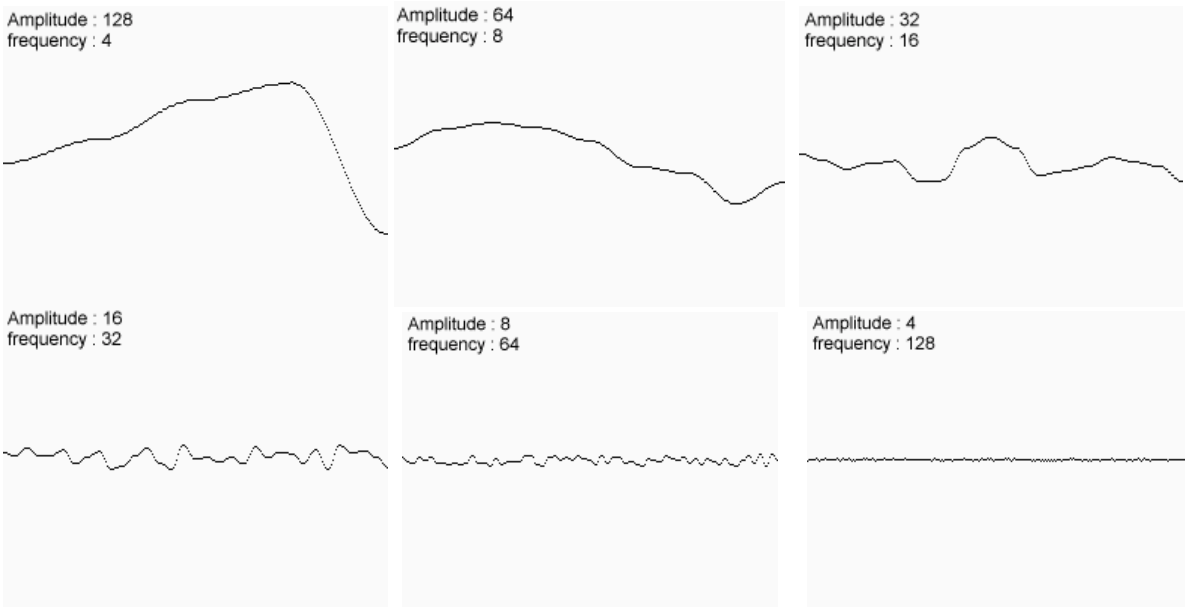


Figure 2.6: The same basis function at different scales. Every scaling the amplitude is reduced in half and the frequency is doubled. (Images taken from [3])

In order to create one single function that will repeat the same form over different scales, we simply add all these appropriately scaled basis functions together. The result of this superposition is the Perlin Noise function. It is shown in Figure 2.7. We can clearly see how the function is self similar and it repeats the same form over different scales.

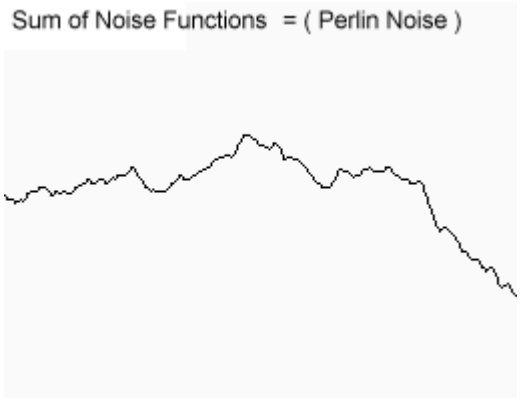


Figure 2.7: The Perlin Noise function. (Image taken from [3])

Even from this simple example of one dimensional Perlin Noise function, we can see that it satisfies our requirements that we have stated at the beginning. It contains a big portion of randomness and also a high complexity at different scales. Hence it is perfect function for artificial creation of naturally looking terrain models. In Figure 2.8 is the same principle demonstrated on 2-dimensional Perlin Noise function. The output value of the Perlin Noise algorithm is interpreted as the mixing coefficient between black and white colors. We can see that the higher scales significantly influence the global appearance of the function, whereas the lower scales add detail and randomness to the result.

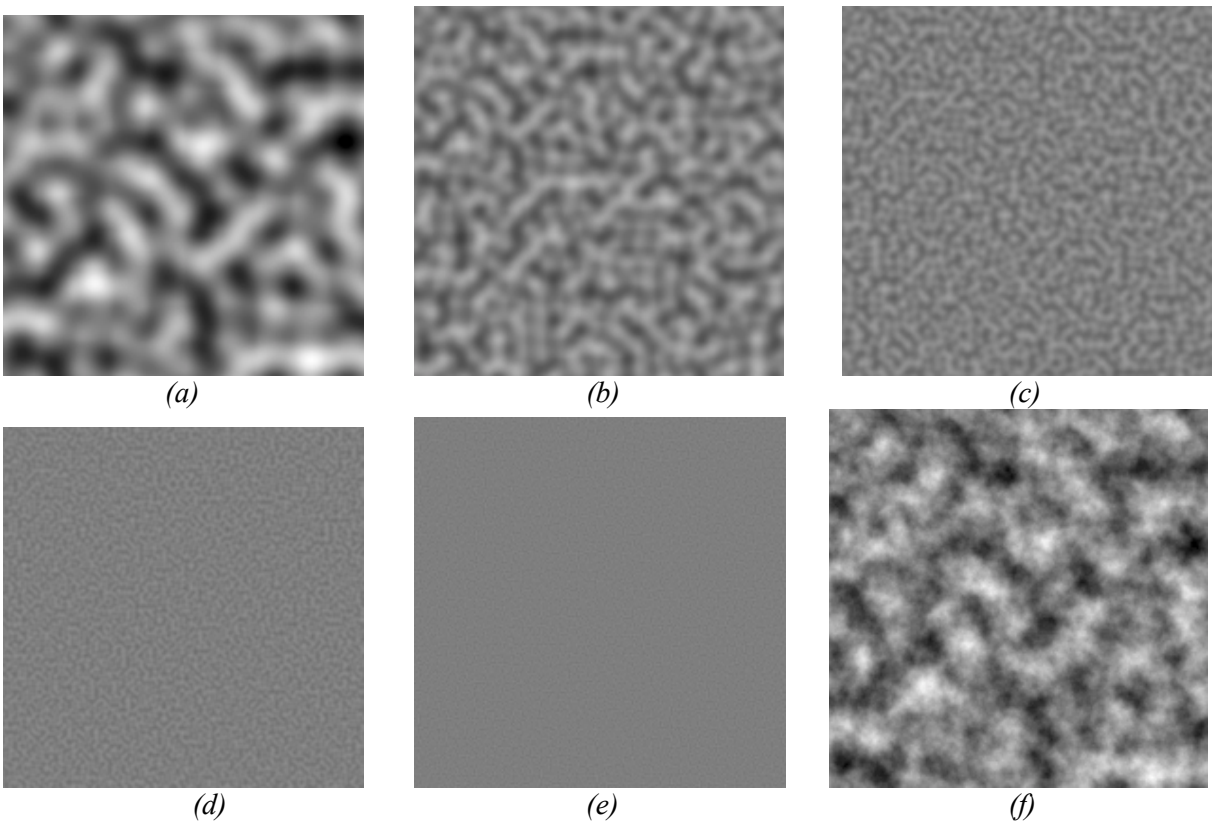


Figure 2.8: 2D Perlin Noise function. (a) – (e) octaves number 1 – 5. (f) The result of summing up all the octaves. (Images taken from [13])

Important parameters

Defining some basic terms related to fractals is necessary for further explanation of fractal algorithms. *Frequency* of the basic function refers to the distance between two neighboring points that we use for the interpolation of the continuous function. The frequency could be sometimes substituted by *lambda*, which is a wavelength of the function and it is 1/frequency. The *amplitude* is the height of the function. We can see it as the range of possible values that we can assign to each point. Every scaled basic function that is superposed in the final function is referred to as an *octave.* We can very easily express the complexity of the fractal by specifying the number of octaves that we have used for its generation.

There are two basic terms that refer to the relationship of two consecutive octaves. *Lacunarity* comes from a Latin word for "gap" and it specifies how the frequency of the current octave is related to the previous one octave. In most cases the lacunarity is equal to 2, although for example in Chapter 3 of [1] we can find several examples of functions with different values of lacunarity. The scaling of the amplitude controls an attribute called *persistence.* It determines how much will be the amplitude of a consecutive octave reduced compared to the previous one. Smaller values result in a rough function, while higher values produce smoother functions.

In addition to those basic terms we should also define the *fractal dimension*. In contrast to the Euclidian dimension the fractal dimension can be a non-integer value. The integer part specifies the underlying Euclidian dimension and the decimal fraction is referred to as the *fractal increment*. The 0.0 value of the fractal increment means that the fractal only occupies its underlying Euclidian dimension. When the value approaches 0.99 the fractal is more and more filling the higher dimension. We can understand this as that the higher values of fractal increment will cause higher roughness of the fractal, while lower values will results in a smoother fractal. For our purpose this is a sufficient understanding of a fractal dimension. Further mathematical description of the fractal dimension could be found in [5].

To be more precise using the terms that we have just defined, we can mathematically formulate the Perlin Noise function as the following equation, where the vector x can stand for a point in 1D, 2D or 3D space:

$$PerlinNoise(\vec{x}) = amplitude * \sum_{i=0}^{octaves-1} \frac{Noise(lacunarity^i * \vec{x})}{persistence^i}$$

2.2.3 What Is a Multifractal?

As we can see in the Figure 2.7 or 2.8f a classical Perlin Noise function is a homogenous and isotropic function. The fractal has the same roughness and the same appearance everywhere and in all directions. Although we said that this fractal function is very useful for generation of naturally created objects, we can usually see that real natural objects are far more complex. Their complexity comes from their not being homogenous and isotropic. If we think about mountain range, we can see that it is quite smooth at the foothills and as we proceed closer to the peaks the mountain becomes rougher and jagged. It is therefore heterogeneous. A fractal that is heterogeneous is called a *multifractal*. In particular multifractals are fractals with a different fractal dimension at different locations. They are usually generated by a modification of a classical fractal algorithm, where we make the fractal dimension of the terrain a function of some other attribute. For example the fractal dimension could be a function of the elevation. Further explanation of the generation and features of multifractal terrain could be found in section 2.3.3 for instance. An example of a terrain generated with multifractal function could be seen in the Figure 2.9.

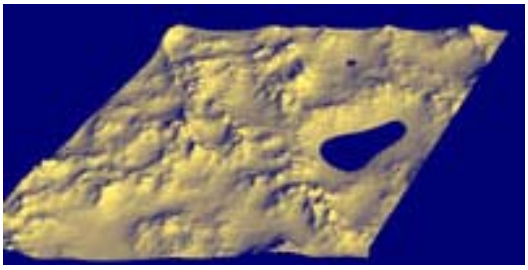


Figure 2.9: Terrain generated with a multifractal function. We can see that the roughness of the terrain is a function of the altitude. (Image taken from [6])

2.3 Algorithms for Terrain Generation

This section contains and overview and a description of fractal algorithms that are commonly used for terrain generation. The survey starts with a Random Faults algorithm, then we will describe the principles of Mid-Point Displacement algorithm and we will finish with a description of several modifications of the already mentioned Perlin Noise algorithm.

In most cases when those algorithms are described in a literature, they are meant for generation of a limited patch of a terrain like the one shown on Figure 2.9. This means that the mesh grid usually is a 2-dimensional plane and the algorithms displace the vertexes in the 3rd dimension. Because we are going to use these algorithms for generating a terrain on a planet, we are more interested in the application of those algorithms on a sphere or some other kind of solid. However this does not usually require major changes to the algorithms and we will point out any significant differences.

In this section our primary goal is to describe the principles of the algorithms. Therefore we will abstract from any kind of implementation issues and problems. We will talk about these in the following Chapter 3 about the analyses and in Chapter 4 about the implementation. For this reason we will assume that the algorithms are working over some kind of a data structure. For now we will call this data structure a mesh, without any further details about how it is actually implemented. The only thing that is important now is that the mesh gives us access to all the information that we need.

2.3.1 Random Faults

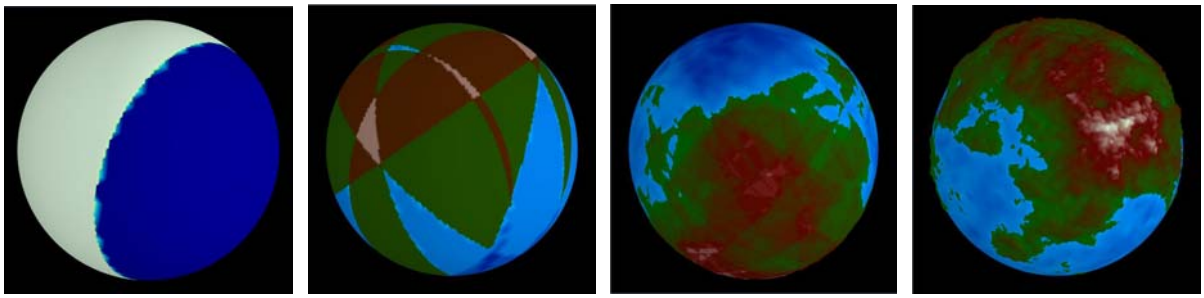
Although the Random Faults algorithm is not usually considered to be a fractal algorithm it could be found in the literature as one of the basic algorithm for terrain generation. On the other hand for example in [2] it is classified as an algorithm capable of producing a terrain model with certain fractal features. Table 1 shows the pseudo-code of the algorithm.

```
1 Random Faults
2 Input: N – number of iterations, Mesh – mesh structure
      Amp - amplitude
3 Output: Mesh – modified mesh structure

4 For (1 : N){
5     Find random plane cutting the Mesh;
6     R = random value from range 0 … Amp;
7     Split the Mesh by the cutting plane into two hemispheres;
8     Enlarge one hemisphere by R;
9     Reduce the other hemisphere by R;
10 }
```

Table 1: Random Faults Algorithm.

We can see that this algorithm does not create the terrain model in one single step. In every iteration it adds more and more features into the model and the terrain becomes more and more random looking. For that reason the number of iterations is very important parameter influencing the final appearance of the terrain. Although we are using a flat plane to cut the sphere and the cut could be seen as a straight line on the surface, after adding sufficient number of iterations together the straight lines start disappearing as a result of random orientations of these cutting planes. This could be seen in Figure 2.10.



(a) *(b)* *(c)* *(d)*

Figure 2.10: Terrain generation with Random Faults algorithm. (a) Planet after 1 iteration. (b) Planet after 10 iterations. We can clearly see straight lines. (c) After 100 iterations. Straight lines are disappearing.. (d) After 1000 iterations. Regularities disappeared. The terrain has a random look.

2.3.2 Mid-Point Displacement

Unlike the Random Faults algorithm, Mid-Point Displacement creates the terrain model during the generation of the mesh structure. In every iteration it refines the resolution of the Mesh and displaces the new points in the desired way. Table 2 summarizes the pseudo-code of the algorithm.

```
1 Mid-Point Displacement
2 Input: Mesh – input mesh structure, Amp – amplitude,
      L – level of detail
3 Output: Mesh_Out – new Mesh with increased resolution

4 For (each polygon in Mesh){
5     Increase the resolution of the polygon by adding new
      points and creating new polygons;
6     For (each new point){
7          Offset = random value from range -Amp … Amp;
8          Offset = Offset / 2^L;
9          point->height = average(parents) + Offset
10     }
11 }
```

Table 2: Mid-Point Displacement Algorithm

Increasing the resolution of the polygonal mesh

In Table 2 at line 5 we are increasing the resolution of every polygon in the mesh. There are several ways of doing that. Primarily they depend on what we actually mean by polygon. Typically the terrain model is represented as a mesh consisting of triangles or quadrangle. Because the most usual and probably the easiest way how to represent a sphere as a mesh is using triangles, we are interested in increasing the resolution of a triangular mesh. The way we refine the resolution of a triangle is shown in Figure 2.11.

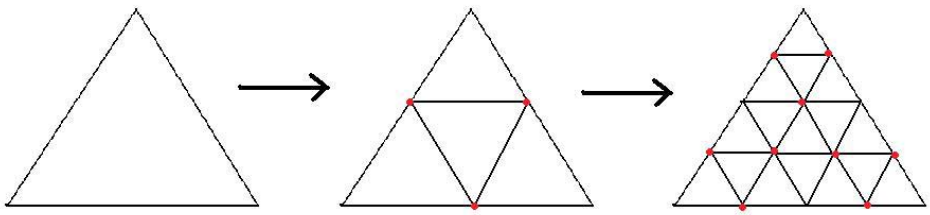


Figure 2.11: Increasing the resolution of a triangle. New points are marked with red dots.

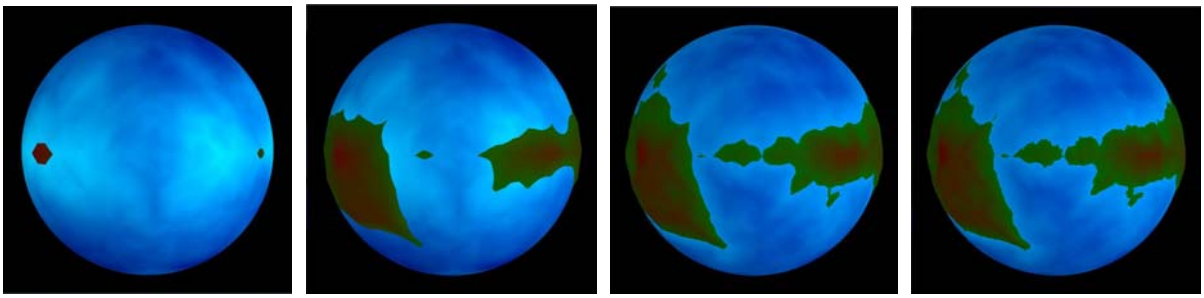
Level of detail

One of the input parameters of the algorithm is the level of detail of the current mesh. The reason why we have to know it is that it is necessary for setting the right offset for displacing the new points. If we think of the amplitude as the maximal elevation that we would like to have in the generated terrain, then in every iteration of the Mid-Point displacement we have to appropriately scale the amplitude of the displacement of the newly added points. If we did not do so then the terrain would be just a set of points with completely random elevations and with no relation to one other. Because every new point reduces the length of the edge of its parent in half, we also reduce the amplitude in every iteration in half. Therefore in the Table 2 at line 8 we reduce the offset by 2 to the power of L, where L is the level of detail.

Displacement of points

The base for setting up the elevation of the new point is the average elevation of its parents. By parents we mean the end points of the edge on which the new point was added. In addition this point is displaced from this basic position by some random offset, which is a value from a range appropriately scaled relatively to the level of detail as we described in the previous section. Because

we are reducing the amplitude in each iteration, we can say that points created during the first few iterations are responsible for a big features in the terrain like where will the mountains, islands or sea be. On the other hand points generated in later iterations add more detail into the terrain but do not really influence the global appearance of the terrain. This is illustrated in Figure 2.12 where we display the planet after 1, 3, 5 and 7 iterations of the Mid-Point Displacement algorithm. We can see that already the first iteration determines where the mountains will be, while the last iterations add only detail to the terrain.



(a) *(b)* *(c)* *(d)*

Figure 2.12: Terrain generation with Mid-Point Displacement algorithm. (a) After 1 iteration. (b) After 3 iterations. (c) After 5 iterations. (d) After 7 iterations.

2.3.3 Mid-Point Displacement Multifractal

The principle of this algorithm is identical to the classical Mid-Point Displacement described in the section above. The classical Mid-Point Displacement creates a homogenous terrain model that has the same properties everywhere. As we mentioned in section 2.2.3 a real terrain is usually heterogeneous, in a way that the terrain properties like roughness or smoothness are different at different places. Multifractal modification of the Mid-Point Displacement creates terrain with such features.

In order to make the terrain multifractal we have to vary the fractal dimension. One basic way to do this is to make the fractal dimension a function of the elevation of the terrain. This comes from a simple assumption that jagged mountains are situated usually in higher elevations, while smooth lowlands will be near the sea level. The following Table 3 shows the pseudo-code of the algorithm.

```
1 Mid-Point Displacement Multifractal
2 Input: Mesh – input mesh structure, Amp – amplitude,
     L – level of detail
3 Output: Mesh_Out – new Mesh with increased resolution

4 For (each polygon in Mesh){
5    Increase the resolution of the polygon by adding new
     points and creating new polygons;
6    For (each new point){
7         Offset = random value from range -Amp … Amp;
8         Offset = Offset / 2^L;
9         Offset = Offset * average(parents) * k;
10        point->height = average(parents) + Offset
11    }
12 }
```

Table 3: Mid-Point Displacement Multifractal Algorithm

The only difference compared to the classical Mid-Point Displacement is at line 9. Here we scale the offset proportionally to the average elevation of the new point's parents. If the point is

located in lower elevation and its parent's elevation are close to 0, then the offset will be significantly reduced and the terrain will be smooth without any sharp edges or sudden changes. But if the point is located in higher elevations then the offset will be multiplied by a bigger number, which will result in the terrain's being jagged and rough with a big elevation changes between neighboring points. The constant k is just needed in order to appropriately scale the elevation. We can see the difference between normal and multifractal version of the Mid-Point Displacement algorithm in Figure 5.6a and 5.6b.

2.3.4 Perlin Noise

Perlin Noise is probably the most known fractal algorithm for terrain generation and for texturing. Its use is not restricted only to those two areas. It is also commonly used in computer animation for simulating a naturally looking movement, see [7], and pretty much anywhere where we want anything to look like being created by nature.

We have already described the basic principle of Perlin Noise algorithm in section 2.2.2. In order to be able to use this algorithm for terrain generation we have to extend the described 1D version into 2D and in our case of generating a planetary model, into 3D version. Although it would be sufficient to create only 2D terrain mesh and then map it on a 3D sphere, creating a 3D Perlin Noise function and then using it for determining the elevation of every point on the sphere is a better solution. This way we do not have to take care of any issues like smoothly connecting the blocks of mapped 2D terrain together.

As we have explained earlier Perlin Noise function is a continuous function and therefore the 3D version will map the terrain smoothly and continuously all over the planet. This brings another advantage compared to Mid-Point Displacement. Perlin Noise takes as the input an already existing sphere or other solid and only appropriately displaces the points. This means that the terrain model is created in a single step, unlike Random Faults or Mid-Point Displacement, and we can use object of any shape as the input for Perlin Noise. The pseudo-code of the algorithm is shown in Table 4. Examples of a planet terrain generated with this algorithm could be seen in Figure 5.5d.

```
1 Perlin Noise
2 Input: Mesh – input mesh structure, oct – number of octaves,
      lambda – wavelength of the function, Amp – amplitude of
      the function
3 Output: Mesh – modified mesh structure

4 For (each point in Mesh){
5     Offset = PerlinNoise3D(point->x, point->y, point-z, oct,
            lambda, Amp);
6     Point->height += Offset;
7 }

8 PerlinNoise3D(x,y,z, octaves, lambda, Amplitude){
9     For (oct = 1 : octaves){
10          Amp = Amplitude / (2^oct);
11          Lam = lambda / (2^oct);
12          Sum += Interpolate(x/lam, y/lam, z/lam) * Amp;
13    }
14    Return Sum;
15 }
```

Table 4: Perlin Noise Algorithm.

Displacement of points

At line 4 starts the main cycle of the algorithm that goes through the whole mesh structure. It assigns each point new increment of its elevation and displaces this point accordingly. Because we are working with a 3D Perlin Noise function we evaluate the function in each point based on its x, y and z coordinates. The final appearance of the function can be influenced by setting the number of octaves, the wavelength of the function and its amplitude. The meaning of these parameters was explained in section 2.2.2.

Evaluation of the Perlin Noise function

At line 8 starts the *PerlinNoise3D()* function, which is responsible for the actual evaluation of the function at the given point. Those few lines of code summarize what was explained and shown in Figures in section 2.2.2. We evaluate each octave, which is the for-cycle starting at line 9, and then add their contribution together in the variable Sum. Each consecutive octave has amplitude reduced in half at line 10. Each consecutive octave has a doubled frequency, which is equivalent to reducing the wavelength in half at line 11. At line 12 we evaluate the current octave for the given point's coordinates that are appropriately scaled according to the new wavelength. The result is scaled according to the amplitude of the current octave.

When we explained the principle of 1D Perlin Noise function in section 2.2.2, we were interpolating only between two neighboring points of the current point. The only main difference between 1D and 3D version of Perlin Noise function is, that we have to interpolate between neighboring points in all three dimensions. We can imagine this as that the point in which we are trying to evaluate the function is closed in a cube determined by 8 corner points with given values and we have to interpolate among them.

There are several ways how to interpolate between two points. The easiest way is a linear interpolation, which does not produce quite smooth function. A little bit better result could be achieved with cosine interpolation. But probably the best looking results are obtained when using the cubic interpolation. Examples of these three interpolation technique are shown in Figure 2.13. Further explanation of interpolation methods could be found in [3].

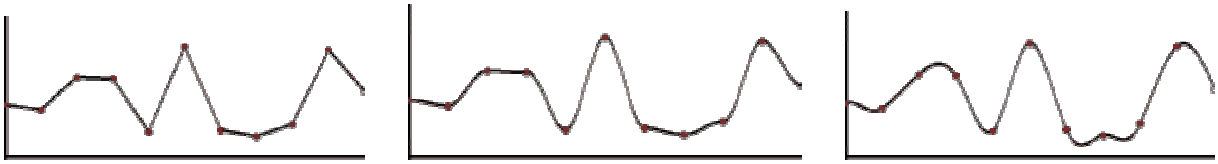


Figure 2.13: Interpolating methods. (a) Linear Interpolation. (b) Cosine Interpolation. (c) Cubic Interpolation. (Images taken from [3])

In order to produce smoother, less random and less square output we should also apply some smoothing function to the result of the interpolation. An example is shown in Figure 2.14.

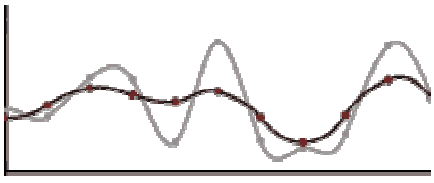


Figure 2.14: Smoothing of the function. Light grey color marks the original function. Dark grey color marks the result of the smoothing. (Image taken from [3])

2.3.5 Perlin Noise Multifractal

This is a multifractal variation of the classic Perlin Noise algorithm. The motivation for this modification is the same as described in section 2.2.3 about Multifractals and in section 2.3.3 about multifractal variation of the Mid-Point Displacement algorithm. In order to produce more realistically looking terrain we have to vary the fractal dimension. Again an obvious way to do this is to make the fractal dimension a function of the elevation. Because the principle of this algorithm is very similar to the classical Perlin Noise and the main cycle stays the same as in Table 4, I will show here only the modified function *PerlinNoise3DMultifractal()* that evaluates the Perlin Noise function at the given point. Table 5 shows the pseudo-code. Example of a planet generated with this algorithm could be seen in Figure 5.5e.

The principle is very similar to the multifractal version of the Mid-Point Displacement algorithm. The main difference is at line 7. We used the elevation of the point, which is determined by the sum of contributions of all octaves calculated so far, to scale the addition of the current octave. If the elevation is close to 0, the terrain will look smoother, because the contribution of higher octaves is reduced. On the other hand if the elevation of the point is higher, then the contribution of higher octaves is emphasized and the terrain will look jagged and rough. The constant k is needed in order to appropriately scale the range of the elevation.

```
1 PerlinNoise3DMultifractal(x,y,z, octaves, lambda, Amplitude){

2     For (oct = 1 : octaves){
3          Amp = Amplitude / (2^oct);
4          Lam = lambda / (2^oct);
5          Add = Interpolate(x/lam, y/lam, z/lam) * Amp;
6          If (oct > 1){
7               Add *= k * Sum;
8          }
9          Sum += Add;
10    }
11    Return Sum;
12 }
```

Table 5: Perlin Noise Multifractal Algorithm

The reason why we don't vary the fractal dimension of the first octave, which is caused by the If-statement at line 6, is that the first octave determines the basic elevation. Consequently based on the elevation added by this first octave, the fractal dimension of the consecutive octave could be influenced.

2.3.6 Perlin Noise Ridged

This is a slight modification to the classical Perlin Noise algorithm. The main principle of the algorithm stays the same. The only difference is in the way we use the output value generated by the Perlin Noise function. Table 6 shows the modification.

The If-statement starting at line 6 changes the Perlin Noise into a Perlin Noise Ridged. For more explicitness Figure 2.15 shows the Perlin Noise Ridged function's response as a function of the response of the classical Perlin Noise function.

```
1 PerlinNoise3DRidged(x,y,z, octaves, lambda, Amplitude){

2     For (oct = 1 : octaves){
3          Amp = Amplitude / (2^oct);
4          Lam = lambda / (2^oct);
5          Add = Interpolate(x/lam, y/lam, z/lam);
6          If (Add > 0){
7               Add = (-Add) + 1;
8          }
9          Else {
10              Add = Add + 1;
11         }
12         Sum += Add * Amp;
13    }
14    Return Sum;
15 }
```

Table 6: Perlin Noise Ridged Algorithm

The main difference in the terrain produced by classical Perlin Noise and by Perlin Noise Ridged is that Perlin Noise Ridged tends to create long and narrow islands, peninsulas and mountain ranges. This is quite reasonable considering what Perlin Noise Ridged is actually doing. If we interpret the classical Perlin Noise from the terrain point of view, then the value -1 usually means bottom of a sea and the value 1 means high mountain. Opposed to this, Perlin Noise Ridged interprets both sides of the interval [-1 ; 1] as a bottom of a sea and only lifts the line between them as mountains.

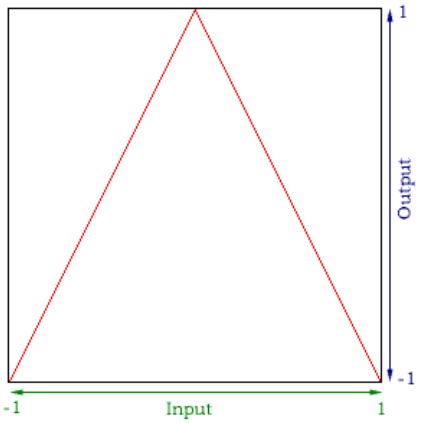
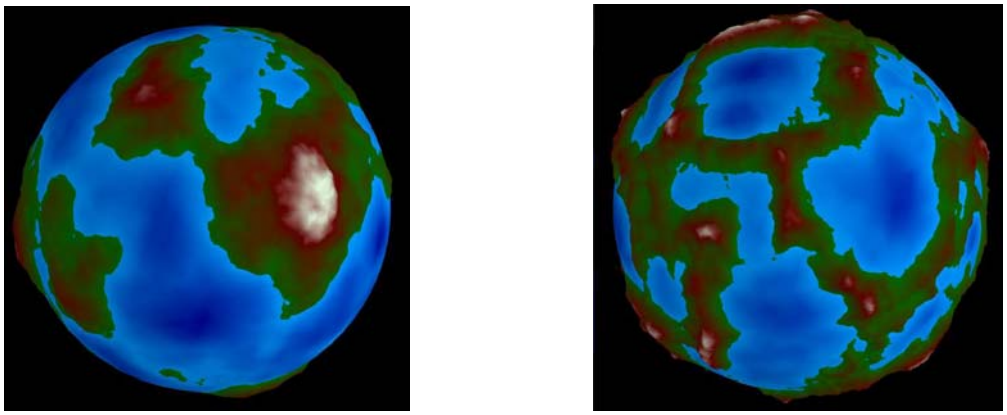


Figure 2.15: Perlin Noise Ridged. The Perlin Noise function is the Input and the Perlin Noise Ridged is the Output. (Image taken from [8])

This could be seen in Figure 2.16, where we have a planet generated with the same Perlin Noise function. In 2.16a was used normal Perlin Noise algorithm and in 2.16b was used the Perlin Noise Ridged algorithm. We can see that where there was a deep see or high mountains on the first planet there is a deep see in the second planet. At the same time where there was a shoreline on the first planet there are mountains in the second one. In order for the Ridged Perlin Noise algorithm to produce more realistically looking terrains, we have to use bigger value of lambda for the Perlin Noise function. We can see that the same value of lambda as the one used for the normal Perlin Noise produces too many small islands and long ranges.



(a) *(b)*

Figure 2.16: Comparison of planets generated with normal Perlin Noise (a) and with Ridged Perlin Noise (b).

2.3.7 Perlin Noise Ridged Multifractal

This modification of Perlin Noise Ridged algorithm creates more realistic terrain by incorporating the multifractal principle of varying the fractal dimension. We are not going to explain this version of Perlin Noise in much detail, because it is using principles that have already been explained here. We calculate and modify the response of the Perlin Noise function as explained in section 2.3.6 about Perlin Noise Ridged algorithm. Then we use the same principle to add the multifractality into the terrain that has been already explained in section 2.3.5. That means that we make the fractal dimension a function of the elevation. The algorithm will produce smooth lowlands and coasts and rough and jagged mountains. An example of a planet created with this algorithm could be seen in Figure 5.5g.

2.4 Algorithms for Terrain Coloring

One part of generating realistically looking planetary models is the generation of the terrain. But there is no doubt that any terrain model will never look realistically without a good coloring of the surface. In the matter of fact in our case of generating planets, the coloring is very often even more important than the terrain itself. This is especially true when we are looking at the planet from a bigger distance. In this situation terrain features like mountains or valleys are not visible at all. The only way how we can see them is if they are emphasized by a good coloring model.

In this section we will describe some basic techniques and principles for coloring of the planet's surface. Some are only dependent on the terrain model that they are used on; others incorporate fractal principles to add more randomness and more natural look. Rather then a description of specific coloring algorithms and coloring schemes, this section will explain some of the most commonly used techniques for coloring. In further chapters we will present our own coloring schemes that we have created by using these techniques and combining them together.

2.4.1 Linear Interpolation

The color of a terrain usually changes slowly. For example if you imagine a border between forests and rocky mountains we can see that usually it is not a sharp line. As we approach into higher altitudes there is still less and less trees and the color slowly and continuously changes from tones of green to some kind of a grey rocky color. We would like to achieve the same in our coloring model. Instead of sharp transitions between two colors, we need to continuously change the tone of the color. Linear interpolation is a basic technique for doing that.

Color representation

Before we can start interpolating we have to know what kind of values we are actually using. In our case we would like to interpolate between two colors. Obviously color is not an exact value that could be used for interpolation, so we should know how the color is represented. There are several different color representations used in computer graphics. Without any further arguing about advantages or disadvantages of one or another, we will assume in further explanations that our color is represented using the RGB system. Each color could be described as a weighted composition of three basic colors: Red, Green and Blue. In Figure 2.17 is shown the color space. This is a usual representation of color space that is used in many programming languages.

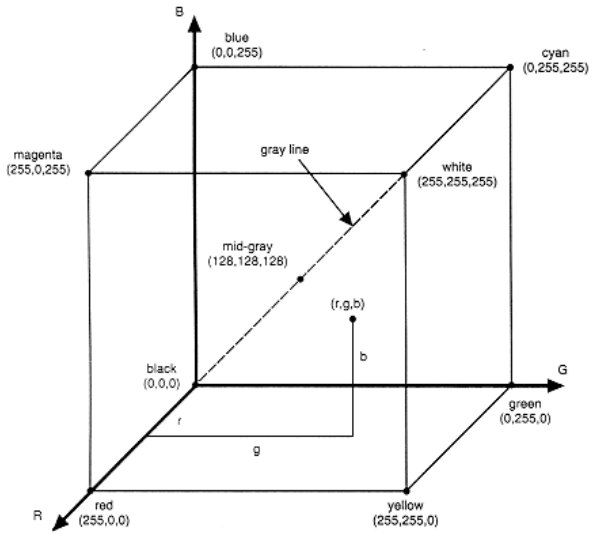


Figure 2.17: RGB color space. (Image taken from [9])

When we represent the color as a composition of three RGB parts, we finally have our exact values that we can use for interpolation. So if we want to interpolate between two colors, we have to first interpolate between their appropriate RGB parts and then put those interpolated parts together to receive the final color.

Interpolation function

There is nothing tricky about the linear interpolation. The reason why we are explaining this method here is that it shows some basic principles and issues with coloring algorithms and it is a good introduction to slightly more complicated coloring techniques.

The principle of the interpolation is that the color is a function of some other variable and it is changing according to this variable. At the beginning of this section we have described an example of how color is changing when we go from hills covered with forests into rocky mountains tops covered with nothing else but white snow. In this case our variable would be the altitude. For simplicity we will assume now that we interpolate only between two colors in one direction. The x coordinate of a point in the mesh is a variable that controls the interpolation. The pseudo-code is in Table 7.

We can see at lines 5, 7 and 9 how each of RGB parts is interpolated separately based on the x coordinate of the point. At the end at line 8, all those parts are combined together to determine the new color of the point. An example of a linear interpolation could be seen on Figure 2.18.

```
1 Linear Interpolation
2 Input: Color1, Color2, Mesh – structure containing points,
          Interval – the length of the interval of interpolation
3 Output: Mesh – structure with colored points

4 For (each Point in Mesh){
5     offsetRed = Point->x * ((Color2->R – Color1->R) / Interval);
6     Red = Color1->R + offsetRed;
7     offsetGreen = Point->x * ((Color2->G – Color1->G) / Interval);
8     Green = Color1->G + offsetG;
9     offsetBlue = Point->x * ((Color2->B – Color1->B) / Interval);
10    Blue = Color->B + offsetB;
11    Point->Color = Color(Red, Green, Blue)
12 }
```

Table 7: Linear Interpolation of RGB color

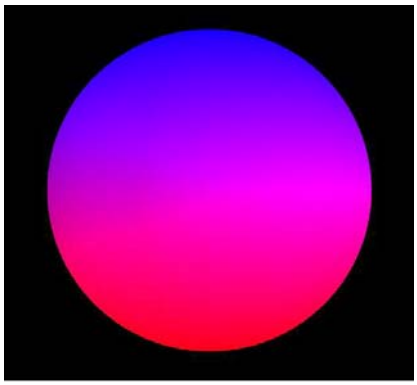


Figure 2.18: Coloring using the linear interpolation applied to a sphere.

2.4.2 Spline Function

The linear interpolation is truly just a simple and basic coloring technique. In many situations we need something more sophisticated. First of all in many cases we have more than just two colors to interpolate between. Therefore it would be very useful to construct a function that will smoothly spread all those colors along the interval and continuously interpolate between them. Second of all the linear function does not always give us the best results. The transition would be nicer if we use some more complicated function that will interpolate between the chosen colors. The Spline function solves both of these problems.

Spline function uses a cubic polynomial function to interpolate between all defined points. As a result we can construct a nice looking and smooth interpolation function that continuously changes the color based on some controlling variable going through all the predefined color points. The best way to understand this is to see an example of a Spline function in Figure 2.19.

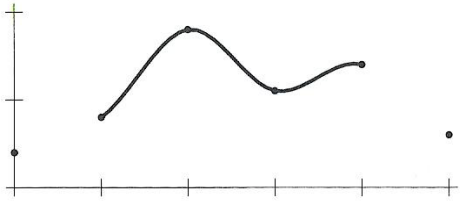


Figure 2.19: Spline function. The first and the last point only determine the derivates of the spline at the end points. (Image taken from [1])

Each point that we would like our spline function to go through is called a knot. The knot could represent almost any value. Because we are using a cubic polynomial function for the interpolation there must be at least 4 knots. The first and the last knot only determine the derivates of the function at its endpoints.

```
1 Spline Function
2 Input: x – Point->x coordinate of the point, knot[N] – an array
      containing all the knots, N – number of knots,
      Interval – the length of the interval of interpolation
3 Output: value – the interpolated output value of the function

4 Define cubic polynomial coefficients;
5 Based on x find the surroundings knots;
6 c0,c1,c2,c3 = Calculate the cubic coefficients using the
      surroundings knots;
7 value = ((c3*x + c2)*x + c1)*x + c0;
8 return value;
```

Table 8: Spline Function

Table 8 shows only a very rough pseudo-code of the Spline function. For more detailed description and a complete source code see pages 30-31 in [1].

First at line 4 we have to define the cubic polynomial coefficients that control the actual appearance of the function. Then at line 5 based on the x coordinate, the length of the interval and the number of knots we find the neighboring knots of the point. In the next step at line 6 we calculate the cubic coefficients of the particular segment of the Spline function. The last thing to do at line 7 is to use a Horner's rule to evaluate the function using the calculated coefficients and the actual coordinate.

As we already mentioned in section 2.4.1 in order to interpolate between two colors represented in the RGB system, we have to interpolate each color part separately and then put them together. The same principle has to be used for the Spline function. We have to use three separated Spline function to interpolate each RGB part separately. This means that the knots for the Spline function represent the values of the appropriate RGB part of the color. At the end we can mix those three components together to obtain the final color as the result of the interpolation. An example of sphere colored with a Spline function is shown in Figure 2.20.

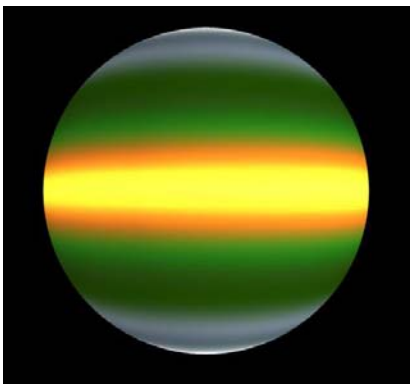


Figure 2.20: Coloring using the Spline function applied to a sphere.

2.4.3 Perlin Noise

In the examples from the previous two sections we used the x coordinate of a point as a variable that was controlling the interpolation. In the terrain coloring this variable might be for example the elevation or the latitude. But sometimes this technique might give us only too synthetically looking results. Sometimes we need to add some randomness to the coloring. What would be a better way of doing that than incorporating Perlin Noise algorithm into the interpolation?

We are still going to use the same interpolation techniques as before. The only difference is that instead of using the x coordinate straight for the interpolation; we will first use the x coordinate as an input value for the Perlin Noise function and then use the output of the Perlin Noise function as the variable controlling the interpolation. An example of how the Perlin Noise function might be used together with the linear interpolation is shown in Table 9. Because the majority of the algorithm is similar to the Linear Interpolation shown in Table 7, I will only show the modified lines for one RGB part as an example.

```
1 Linear Interpolation with Perlin Noise

2     Noise = PerlinNoise(point->coordinates, oct, lambda, Amp);
3     offsetRed = Point->Noise * (Color2->R – Color1->R);
4     Red = Color1->R + offsetRed;
```

Table 9: Linear Interpolation using Perlin Noise

We can see that at line 2 we use the coordinates of the point to calculate the response of the Perlin Noise function and then this value is used to interpolate the color. Opposed to how we described the output of the Perlin Noise function in section 2.2.2, here we need the response of the function to be from the interval [0 ; 1]. So in this example the value 0 corresponds to Color1 and value 1 corresponds to the Color2. Example of coloring based on this technique is shown in Figure 2.21.

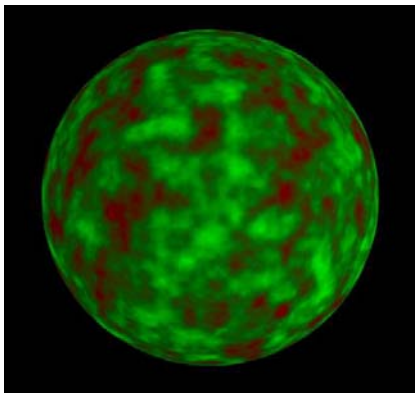


Figure 2.21: Coloring using the Perlin Noise function applied to a sphere.

2.4.4 Perlin Noise Variations

In addition to just using the response of the Perlin Noise function as it is, we can use it as an input for some other function and then use this value as the control variable for the interpolation. For example we can use the Perlin Noise function response as an input argument for cosine or sine function. Or we can simply take the absolute value from the response of the function.

The second example is called a Turbulence function. As we have mentioned the only difference is that we take the absolute value from the response of the Perlin Noise function. The coloring done by this technique could be seen in Figure 2.22.

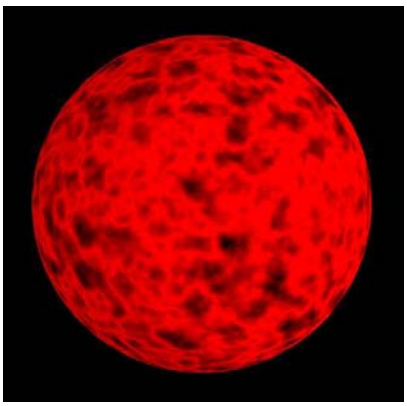


Figure 2.22: Coloring using the Turbulence function applied to a sphere.

2.4.5 Perturbation

Another very useful technique for coloring is called Perturbation. This technique adds a certain amount of noise and randomness into some other function. At the beginning of section 2.1.2 we have described an example of altitude dependent coloring when we the trees stop growing at some certain altitude and further there are just rocks. In a real world this line is not always exactly at the same elevation. It changes a little bit. Sometimes it is a little bit higher, sometimes a little bit lower. Perturbation helps us create a similar effect of randomness.

Again this technique could be used with any kind of interpolation. So for simplicity we will only describe here the most important changes to the code that was already shown in previous sections. The pseudo-code could be found in Table 10.

```
1 Linear Interpolation with Perturbation
2 For (each Point in Mesh){
3     Pertrub = point->x + PerlinNoise(point->x, oct, lambda, Amp);
4     offsetRed = Pertrub * ((Color2->R – Color1->R) / Interval);
5     Red = Color1->R + offsetRed;
```

Table 10: Linear Interpolation with Perturbation

The most important step is at line 3. We still use the coordinate for the interpolation, but we add some noise to its value. If we imagine the coordinate x as the elevation of the point, then the noise will add randomness into the pattern. Another obvious advantage of this approach is that the attributes of the Perlin Noise function enable us very easily control the appearance of the final function. For example by appropriately changing the amplitude we can control the amount of noise in the function. An example of the Perturbation used together with the Spline function is shown in Figure 2.23.

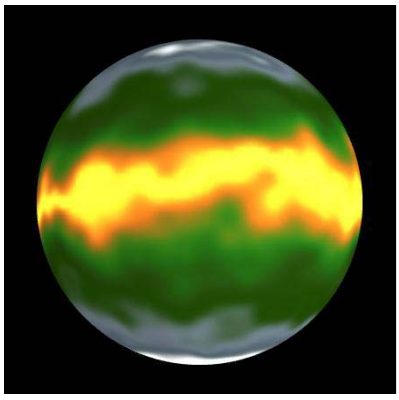


Figure 2.23: Coloring using the Spline function with Perturbation applied to a sphere.

3 Analyses

In order to examine the true capabilities of the terrain generation algorithms and algorithms for terrain coloring there is no better way then writing our own testing application. We have mentioned several commercial and non-commercial applications that are available and that let you experiment with algorithms for terrain generation. But only writing our own program and implementing the algorithms ourselves give us the opportunity to design the application in our way and really concentrate on important aspects of the generation of planetary models that we are interested in.

This application is not meant to be a final version of a fully functioning application for an end user. Rather than that we wanted to create a testing application that will enable us to apply the algorithms for terrain generation in praxis. Finishing this application and making it user friendly with all the necessary features is a subject of future development.

This section provides an analysis of the design of the application. Considering what algorithms we want to implement and what all features we want to incorporate into the application we will analyze what should be the right data structure and the framework for the application. We will look at what should be the right architecture for the program and what performance issues and problems might be.

3.1 Planet Structure

Every algorithm that we have described in Chapter 2 was working over some data structure that we were calling a mesh. Now it is time to look closer at this structure and think about what it should actually consist of and how it should be designed. It is not only important so that the algorithms work the right way, but we also have to think about many performance issues and optimizations that could be done in order for the application to work effectively.

3.1.1 Dynamic Structure

When we create a planetary model we usually start from a sphere, because it is a shape that usually quite well approximates the planet's shape. Afterwards we apply some algorithm to displace the points on the surface and create the desired terrain model. But some terrain generation algorithms work in a different way.

Random Faults algorithm and all variations of Perlin Noise algorithm work with an already created sphere. They keep the resolution of the sphere and they only displace the points. In contrast to that the Mid-Point Displacement algorithm generates the terrain during the creation of the solid itself. This means that we have to be able to dynamically change the complexity of the structure. We must be able to increase its resolution and add more points, more polygons and more detail into it. For that reason the data structure will certainly have to be dynamic.

3.1.2 Hierarchical Structure

All the algorithms that we have described needed a direct access to every point in the mesh, so that they could displace it or change its color. On the other hand computer graphics libraries are usually optimized to render the mesh from polygons. Polygons could have almost any shape but as we have already mentioned in section 2.3.2, for the representation of a sphere or a planet the triangles will be the most suitable ones. So it is clear that in our data structure we will need to have access to both. Therefore we will need some kind of a hierarchical structure that consists of objects pointing to one another. Each triangle should know where its points are.

3.1.3 Linked List Data Structure

In order to render the planet, to generate its terrain or to color it, we will need to effectively iterate through all the points and polygons in the mesh. An array structure would be very suitable for

this. Not only it would let us iterate through all its members but we could also access any point or any polygon at a constant time. However this is in conflict with our need for a dynamic structure that we have explained in section 3.1.1. We need to be able to dynamically change the number of points and polygons in the mesh. For this reason the linked list data structure will be the most suitable one for us. Although this data structure is not really optimal if we want to find one particular point or polygon, it is very effective for adding more members into it. With an array structure we would have to copy the whole array into a bigger one and then add the new points. When we use the linked list we can simply add the new members to the end of the list in a constant time.

3.2 Dynamic Level of Detail

The fractal algorithms give us the great opportunity to create a planetary model with a high level of detail and high complexity. But one of the most common issues in computer graphics and especially when trying to represent some structure with a high resolution, is the size of the structure. The structure is either too big and we are using too much memory space or it contains too many members and it takes too much time to iterate through it and the rendering of the scene is not smooth. So to be able to use the capabilities of fractal algorithms and to overcome those performance issues we need to implement the Dynamic Level of Detail algorithm. Unlike the static level of detail, which maintains the same resolution everywhere on the planet, the dynamic level of detail let us explore the detail of the planet by increasing the resolution only where it is necessary and thus keeping the data structure small.

3.2.1 The Level of Detail of Perlin Noise Algorithm

With fractal algorithms like Perlin Noise the level of detail is very easy to adjust. In the matter of fact the highest possible level of detail of the terrain generated by Perlin Noise is only determined by the highest octave. If we use 10 octaves instead of 8 we will generate terrain with a higher detail. But this assumption only holds if the resolution of the mesh is capable of displaying such a high detail. In other words if the side of the polygon in the mesh is greater than the lambda of the highest octave, then adding this octave into the model will not give us any benefit of increasing the resolution, because we are not able to see it. So in order to create the terrain with all the detail that Perlin Noise algorithm can give us, we need to be able to effectively increase the resolution of the mesh.

3.2.2 Only What We Can See Matters

Adding one more octave to the generation of Perlin Noise should not require too much extra computational time. But increasing the resolution of the planet will certainly require a lot of extra memory space. In Figure 2.11 we have shown what the best way of increasing the resolution of a triangle is. From that picture it is clear that each triangle will be substituted by 4 new ones. So the number of triangles as a function of the level of detail could be calculated as follows.

$$N_{tris} = 4^{l}$$

N_{tris} means the number of triangles in the mesh and l stands for the level of detail of the whole structure. So every time we increase the detail and we reduce the length of the side of one triangle in half the data structure will four times increase its size. It is important to think about whether we really need all the triangles to increase their resolution. It is useful to think about what is actually rendered on the screen. We are certainly not able to see what is too small to be displayed on the screen. And we are even more certainly not able to see what is outside of our view. That means what is figuratively beyond the edges of the screen.

What is too small

The resolution of the screen is given in pixels. Pixel is the smallest thing that could be displayed on the screen. Therefore anything that is smaller the one pixel simply could never be visible,

because the display is not capable of visualizing it. So if we are looking at a planet from a bigger distance we cannot see each mountain top in a mountain range or every single hill or lake. Those things are too small and un-displayable considering how far are we from the surface. But if we start moving towards the planet and we are getting closer and closer to its surface, terrain features like mountain tops and hills will become visible.

So it is obvious that the level of detail of the planet should take into account our distance from the planet. There is no need to have a huge data structure containing all the details of the whole planet when we cannot even see it. Rather than that we should keep the data structure small and only add new polygons when we are able to see the detail provided by them.

What is not in our view

If we are looking at a planet from a bigger distance we can see the whole planet because it is all in our view. But once we get close enough to the surface only a certain part of the planet stays in our view. If we are getting closer smaller and smaller part of the planet is in our view. The static level of detail maintains the same resolution everywhere on the planet. So if we want to see the part that we have zoomed into with a higher detail and we increase the resolution, then the resolution of the whole planet will be increased. But because we cannot see most of the surface, this is certainly a big waste of memory space. It is much more reasonable to increase the resolution only in the area that we are looking at and keep a low resolution in areas of the planet that are not visible. This would certainly save us a lot of memory space and also computational time, because we are not wasting so much time trying to render something that is not going to be on the screen anyways.

3.2.3 Efficient Dynamic Level of Detail

It is for sure that dynamic level of detail will save us a lot of memory space. However we will need to implement several efficient algorithms in order for it to work smoothly and save us computational time as well. Unlike the static level of detail, the dynamic level of detail will need to do a lot computation before rendering each frame. Based on our analyses in the previous section we will need to check how far is each triangle from our position in order to not to have too small triangles in the mesh. Before each rendering of the scene we will need to check what triangles are actually going to be rendered on the screen so that we do not increase the detail of those which will not be visible. And probably most importantly we will have to take care of effective adding and removing of the new triangles into and out of the mesh.

3.3 Analyses of the Application Architecture

We have analyzed so far what kind of a data structure should we use and how the dynamic level of detail should work. In this section we will summarize are findings from previous sections and reason about the architecture of the application. It is important to split the whole program into several modules responsible for different actions and correctly connect those modules together.

3.3.1 Modules

The module in this meaning does not have an exact definition. It could be an object holding the mesh or it could be a group of algorithms responsible for example for the terrain coloring. We can roughly split the whole system into several modules as follows.

Probably the most important module will be an object holding the whole mesh structure. As we explained in section 3.1 this module should encapsulate a hierarchical structure of linked lists connected with each other by pointers. We have to be able to easily iterate through this structure and at any time obtain all the information about the terrain that we need.

Other two modules will group algorithms responsible for terrain generation and for terrain coloring. They will consist of mostly stand alone procedures that will take the mesh structure as input and based on some controlling parameters they will generate the terrain or change its color accordingly. These modules perform their actions only when they are specifically called by the user.

Unlike the previous two modules the module responsible for the dynamic level of detail will be running in the background all the time. Anytime we move the planet, rotate it or zoom into it, we will have to calculate how the mesh changes. This module has to calculate what triangles will be added to the mesh and which ones are to be removed. The computational time required for computations done by this module will be critical for the overall performance of the whole system.

Another module should take care of the rendering of the scene itself. It takes information from the current mesh and renders the planet model onto the screen. This module will be realized by some graphics library and we will only use its interface to pass it the necessary information.

In order for us to be able to somehow interactively control the generation of the planetary model, we will have to build a module taking care of some kind of a user interface. This way we are able to adjust the values of important parameters for the terrain generation. This module will also be realized by some library closely connected to the graphics library.

3.3.2 Application Architecture

When we defined the modules from which should our application consist, we should also design how will be those modules connected together. The following diagram shows the architecture of the whole system.

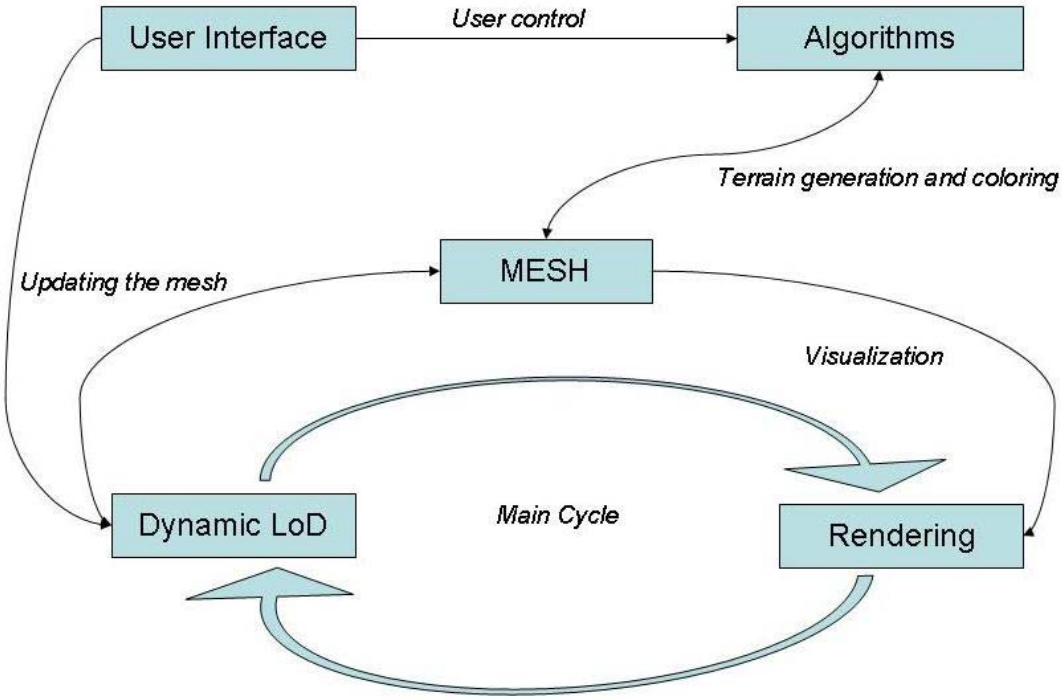


Figure 3.0: The architecture of the system.

The diagram in Figure 3.0 is just a summary of this whole chapter. As we said the fundamental part of the system is the data structure holding the mesh. The main cycle in the bottom part of the diagram is running in a loop. The rendering module takes the mesh and renders it on the screen. Then the mesh is updated by the dynamic level of detail based on how the user changed our position in the scene. If the user wants to use some algorithm either for terrain generation or for terrain coloring, the algorithms module takes the values of control parameters from the user interface module and updates the mesh accordingly. Afterwards we can see the changes in the terrain when the scene is rendered next time.

4 Implementation

In this chapter we will describe the implementation of the application. First we will specify the development framework and what external libraries did we use. We will continue by writing about specific implementation of the architecture that we have analyzed and designed in the previous chapter. Rather than a complete description of all details of the whole system, I we will concentrate on the most interesting parts of the implementation. I we will describe the data structure that was used for the representation of the mesh and how it is optimized for a better performance of the whole system. Another interesting part is the actual implementation of the Dynamic Level of Detail algorithm and several problems that we have encountered while implementing the algorithm. Then we will mention several interesting things about the actual algorithms for terrain generation and terrain coloring. And finally the last part of this chapter will be addressed to the graphical user interface and the way that user can control the generation of the planetary model.

4.1 Development Framework

Most of the application was developed under Microsoft Visual Studio 2003 running on Microsoft Windows XP. The development was finished under Microsoft Visual Studio 2005 running on Microsoft Windows Vista Business. The program is written in C++ programming language.

4.2 Used External Libraries

In some aspects of this application we had to use some external libraries. This section specifies what libraries we chose.

OpenGL

OpenGL [30] stands for the "Open Graphics Library". It is a software interface providing an access to the graphical hardware. This interface consists of several hundred functions and procedures that enable us to specify objects and operations for the graphical output. From the programmer's point of view, the OpenGL is a set of commands controlling the creation of 2D and 3D objects and the way that they will be rendered on the screen.

Glut

Glut [31] is a simple and platform independent program interface primarily designed for creating a simple user interface for applications using the OpenGL. In addition to this, Glut library has also a lot of other functions that extend the OpenGL library and that we are going to use in this application.

Glui

Glui [32] is a Glut-based user interface library. It provides some basic user control mechanisms for OpenGL applications. It provides controls such as buttons, checkboxes, radio buttons, spinners or list boxes. It fully relays on Glut to handle all system-dependent issues like window and mouse management.

4.3 System Architecture

The architecture of the program was implemented according to the diagram in Figure 3.0. This section will briefly describe the organization of the whole source code before we will start explaining the details of particular parts of the application.

The main function is located in file *Fractal.cpp*. Besides that this file also embodies all the important methods controlling the initialization and the life of an OpenGL application. All the important global variables and also the GUI are defined here.

All methods that together implement the dynamic level of detail could be found in file *Lod.cpp*. Most importantly it contains the constructor and destructor of the whole data structure that holds the planet. Further more there are methods dealing with the initialization of the planet, with increasing and decreasing its resolution and also the actual drawing of the structure on the screen. All other objects that together create the whole hierarchical data structure could be found in the following files: *point.cpp*, *pointNode.cpp*, *pointArray.cpp*, *triangle.cpp*, *meshNode.cpp* and *meshTrianlge.cpp*.

The algorithms for terrain generation are implemented in several files. Random faults could be found in *randomFaults.cpp*, Mid-Point Displacement is implemented in *midPointDisplacement.cpp* and all methods dealing with the generation and use of the Perlin Noise are in the file called *perlinNoise.cpp*. The file *planetShader.cpp* holds all the methods implementing various coloring schemes and additional methods for interpolation of the color.

The last two files contain the implementation of some additional features of the application. The *craters.cpp* implements the generation and adding of the craters on the surface of the planet and the *waterLevel.cpp* takes care of coloring and initializing of the water layer.

4.4 Mesh Data Structure

This section is divided into two parts. The first part will explain how the geometrical model of the planet represented is and the second part will be dealing with the actual data structure that is holding this representation.

4.4.1 Planet Representation

As we have already mentioned in previous chapters most of the algorithms has to be applied to an already existing object of some 3-Dimensional shape. Because we are generating a planetary model, the most suitable shape for us is a sphere. It is a very good approximation to the shape of most planets. In order to apply terrain generation and terrain coloring algorithms we need to have access to every point in the sphere structure. This means that we have to find a way how to generate the sphere-like shaped object and store its complete structure in a memory.

It is obvious that our final object will only be an approximation to the real sphere. This is mainly due to the fact that the mesh will consist of flat polygons. The smaller the polygons are the closer to the exact shape of a sphere we are. But this is limited by several performance issues like memory space usage as we discussed in section 3.2. So the final structure will be a compromise between good approximation of sphere and good performance of the application.

We have also already decided on the shape of the polygon in section 2.3.2. Our mesh will consist of triangles, because it is the most suitable representation for many graphics libraries including OpenGL, which is optimized for using triangles. Further in the section 2.3.2 we have shown a way how to increase the resolution of a mesh consisting of triangles. If we apply this iterative algorithm for increasing the resolution to a simple object consisting of triangles and roughly approximating the sphere shape, we can generate our sphere structure.

Iterative sphere generation

It is important to decide what our simple starting object will be. It has to consist of identical triangles that will roughly approximate the shape of a sphere. The octahedron, which is shown in Figure 4.0, does fulfill our needs.

The octahedron has only 6 points and 8 triangles. Therefore it is quite easy to calculate the coordinates of those 6 points manually based on the chosen diameter of the sphere. Placing the center of the sphere into the [0; 0; 0] point of the coordinating system will make it even easier. After specifying the points' coordinates we have to add the 8 triangles by saying which points are supposed to be connected together.

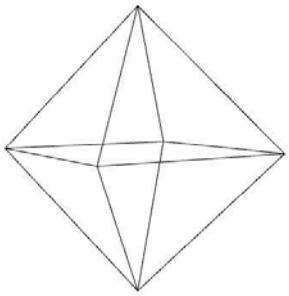


Figure 4.0: Octahedron as a starting object for sphere generation

Now the iterative algorithm for generation of a sphere has to increase the resolution of each triangle in the structure according to the rule shown in Figure 2.11. Each iteration will in sequence remove each triangle and substitute it with 4 new ones. In order for the algorithm to work well, we have to displace the new points so that they are located on the surface of the sphere. Figure 4.1 shows the progress of sphere generation over the first 4 iteration.

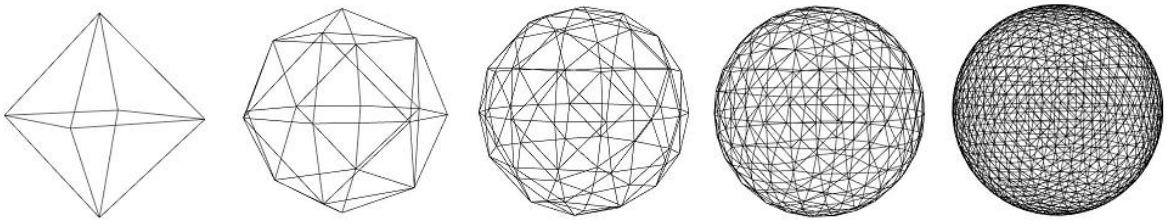
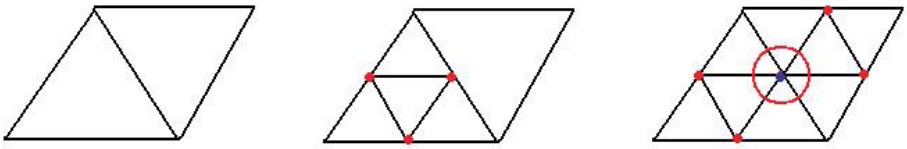


Figure 4.1: Iterative generation of sphere. From left to right the sphere consists of 8, 32, 128, 512, 2048 triangles.

Sharing points

In order to keep the mesh structure as small as possible we should take advantage of the fact that neighboring triangles are sharing an edge. If we increase the resolution of both triangles that are neighbors separately we will have two points located at the same edge and at the same position but each belonging to different triangles. Figure 4.2 illustrates this situation.



(a) *(b)* *(c)*

Figure 4.2: Two neighboring triangles sharing an edge during the iteration. The conflicting point is marked with red circle on image (c).

Image 4.2a shows the two neighboring triangles sharing an edge. In the next image 4.2b we can see that the left triangle has split into four new ones and it has added new point to the conjunctive edge. On the last image 4.2c we can see the problem that can happen if we split the second triangle the same way as we did the first one. The second one will also add a new point to the conjunctive edge and place it at the exact same position. In the red circle we see the conflict between the two triangles.

First of all this will add nearly twice as many points into the mesh than it is necessary. Another problem is that in the matter of fact the mesh is not interconnected together. The triangles are

not sharing points together. The whole mesh would be just a set of separate triangles, which might cause unnecessary artifacts during the terrain generation.

The solution to this problem is that each triangle must be aware of its neighbors. Before it is supposed to split it has to check if its neighbors have been split already and eventually share the point instead of adding a new one. The exact algorithm will be described in section 4.5 about the Dynamic Level of Detail.

4.4.2 Data Structure

The data structure holding the mesh is implemented as a set of linked lists containing hierarchical structure of objects. The linked lists ensure that we can easily iterate through the whole structure. The hierarchy of objects is needed for us to be able to work with the structure the right way and to be able to find out how the objects are connected together. Figure 4.3 shows the whole hierarchy of triangles and points.

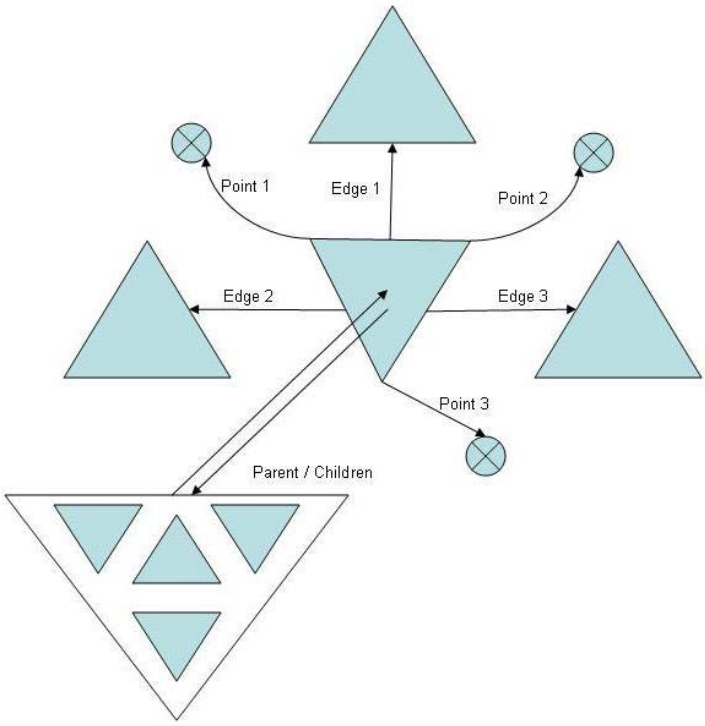


Figure 4.3: The hierarchy of triangles and points. The arrows symbolize pointers to other objects.

Triangle – Points hierarchy

The OpenGL needs to know the information about all triangles in the scene. But in order to specify the triangle exactly we have to know the coordinates of its corner points. For that reason each triangle in the structure has pointers to its three corner points.

Parent – Children hierarchy

In order for the dynamic level of detail to work well, we have to be able to dynamically increase and eventually decrease the resolution of the mesh. This means that later we might need to put back the original triangle that we have removed from the structure when we increased its resolution. Therefore each triangle has four pointers to its eventual children and each triangle also has a pointer to its parent triangle from which it originated.

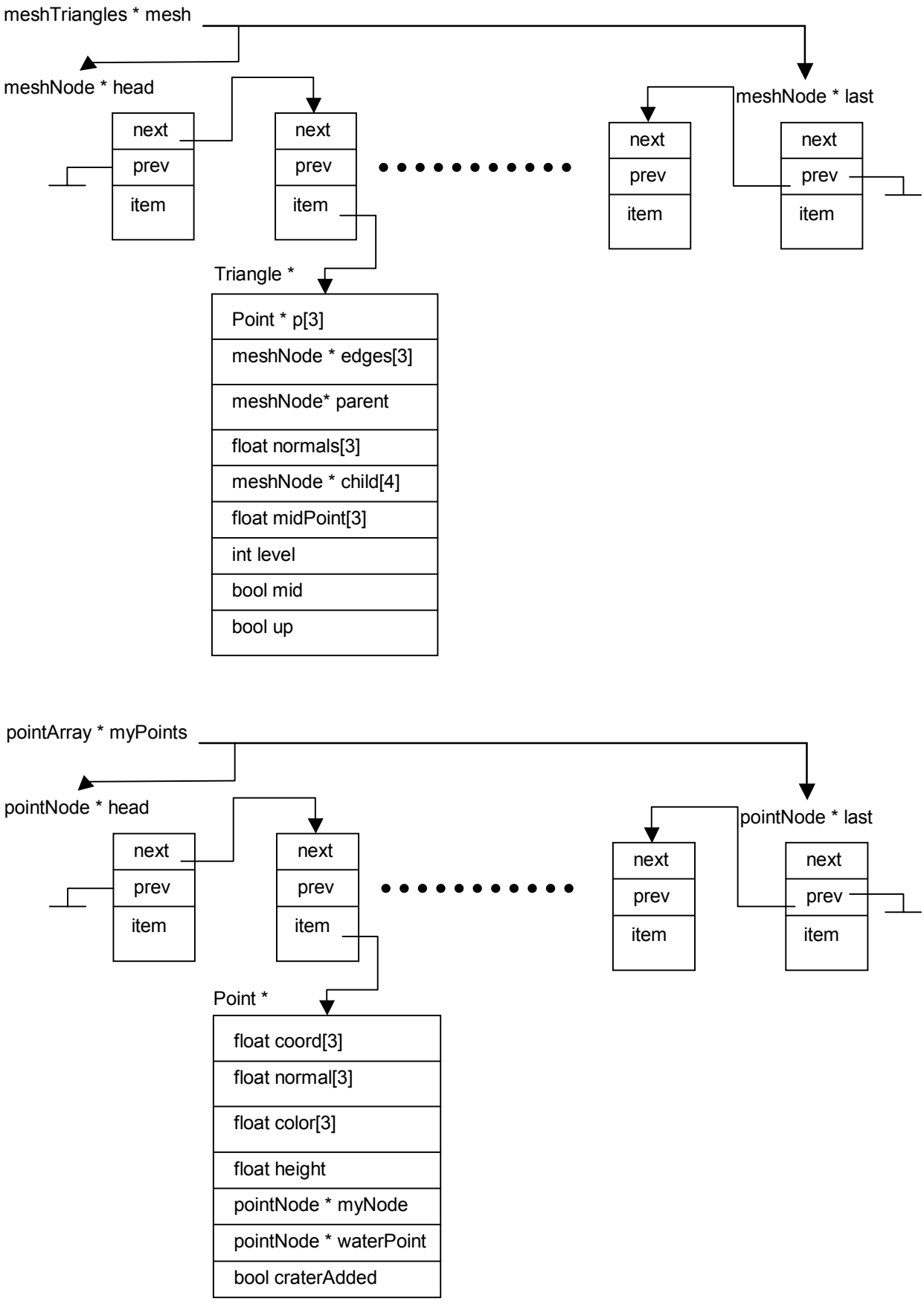


Figure 4.4: Implementation of linked lists holding the mesh structure and the implementation of objects holding triangles and points.

Neighbors structure

In section 4.4.1 we said that each triangle must know about its neighbors in the mesh. This is necessary in order for each triangle to know whether it is supposed to create a new point or share point that was already created by neighboring triangle when it split.

Linked lists hierarchy

The whole mesh is encapsulated in one single object. This object contains two main linked lists. One consists of all triangles in the mesh and the other one consists of all points in the mesh. The whole linked list is also implemented as an object consisting of pointers to its head and to its end. Particular members are created as nodes holding the actual triangles or points and then accompanied by pointers to the previous and to the next item in the linked list. The structure of both linked lists and the exact representation of objects holding a triangle and a point are shown in Figure 4.4.

4.5 Dynamic Level of Detail

From section 3.2 in the analysis of the application we know that the Dynamic Level of Detail algorithm will have to take care of several things. It has to find all triangles that are located in our view. Then it has to calculate which of these are supposed to change their level of detail based on the distance from our viewpoint. And finally it has to take care of successful adding and removing items into and out of the mesh while keeping it consistent. Table 11 summarizes the pseudo code of the whole algorithm. In the following sections we will discuss implementation of particular steps and sub functions of the algorithm. Our own implementation of this algorithm was inspired by [19].

```
1 Dynamic Level of Detail Algorithm
2 Input: Mesh – structure holding the mesh
3 Output: Mesh – updated mesh structure

4 While (main cycle){
5    For (each triangle in the Mesh){
6         If (triangle in our view){
7              Dist = distance(triangle, viewpoint);
8              Level = levelOfDetail(triangle);
9              Threshold = lookUpThresh(Level);
10             If (Dist < Thresh) && (fineToSplit(triangle)){
11                  Split(triangle);
12             }
13             Else If (Dist > Thresh) && (fineToReduce(triangle)){
14                  Reduce(triangle);
15             }
16        }
17   }
18 }
```

Table 11: Dynamic Level of Detail Algorithm

4.5.1 Testing Triangles

If we want to find out whether the current triangle is supposed to change its level of detail, we have to perform several tests. First thing to do is to determine what part of the whole scene, where the planetary model is placed, is located in our view. This is dependent on the current position of the camera, on the direction and the way we are looking at the scene and finally at the position of the

triangle itself. Once we know that the triangle will be in our view, we have to find out whether it is eligible for updating its resolution. This is dependent again on the current position of the camera, on the distance between us and the triangle, the level of detail of the triangle and some arbitrary threshold for the distance.

Determining the camera view

Our view is bounded by the edges of the screen or the window into which we render the scene. So we can imagine each side of our view as a plane that cuts through the whole space. These four planes together determine what will be visible and what will not. What is in between the planes will be rendered what is outside will be hidden to our eyes.

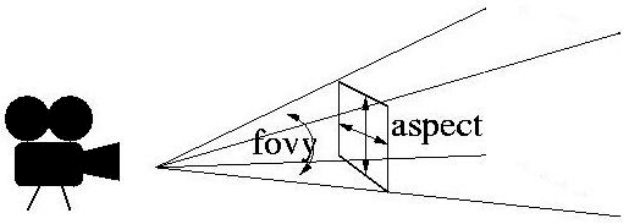


Figure 4.5: Four planes bounding our view. (Image taken from [20])

In OpenGL when we define the perspective projection we specify the aspect ratio of the screen and an angle under which we see the scene. Illustration of this is in Figure 4.5. Therefore because we can calculate the coordinates of the camera in the scene and we can infer the angle of the planes bounding our view from the way we specified the perspective projection, we can calculate an analytic equation in space for each of the four planes. The rest is a just a simple analytical geometry. We have the coordinates of the triangle and we test it with all four equations for each plane. This way we find out whether the triangle is located in our view or not. Other more sophisticated and methods with better performance could be found in [35] or [36].

Back face culling

Actually to be absolutely correct in the previous section we did not find everything that is going to be rendered on the screen but everything that is located in the area that is determined by our four clipping planes. Thus we also found triangles that are not going be rendered on the screen either because they are covered by other objects in the scene or because they might even be located on the other side of the planet. This is scenario is shown in Figure 4.6.

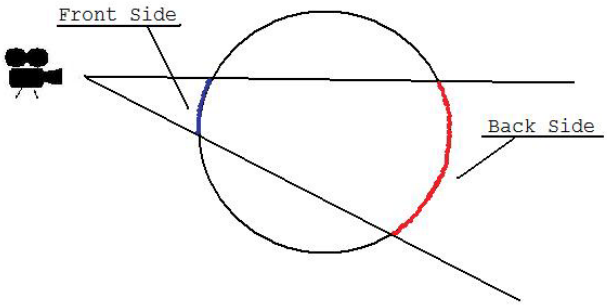


Figure 4.6: Triangles in our view located on the front and back side of the planet. Triangles on the back side are not going to be visible.

Especially for this second reason I tried to implement the Back Face Culling algorithm. It is mainly because I wanted to determine whether the triangle is located on the side of the planet towards the camera or on the other side. The Back Face Culling algorithm calculates the vector product between the vector connecting the camera position with the given triangle and the normal vector of the triangle itself. So it is an easy way to determine whether the triangle is facing us or not. Consequently

we are going to exclude all triangles that are not facing us from the dynamic level of detail, because they are not going to be visible in the scene. However this had a negative impact on the final look of the displayed terrain. If we are looking at a mountain then the Back Face Culling algorithm will exclude its back side from increasing its resolution. But this will also be apparent from the front side, because the lower resolution on the back side will affect the edges of the mountain visible from the front.

For this reason we gave up using the Back Face Culling algorithm and we only implemented a simple rule that decides whether the triangle is on the front or on the back side of the planet. The vector connecting our viewpoint with the center of the planet is a normal vector to a plane that cuts the planet in half. Everything in the front of this plane will be eligible for the dynamic level of detail and everything behind it will not, because it is on the other side of the planet. Figure 4.7 shows two examples of what part of the planet will be affected by the dynamic level of detail.

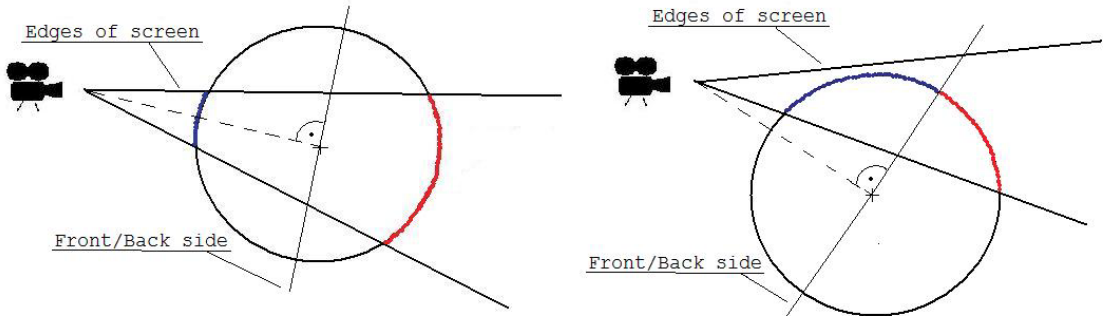


Figure 4.7: Two examples of how the implementation of front/back side decision works. The part of the surface where dynamic level of detail is applied is marked with blue color. Red color means triangles that are inside the four planes clipping our view but are excluded from the dynamic level of detail because they are on the back side. Blue color marks triangles that are eligible for the dynamic level of detail.

Distance threshold look-up table

Once we have decided about the triangle's being in our view, we need to know if it is time for it to change its resolution or not. This depends mainly on our distance from the triangle. That is calculated easily as the Euclidian distance between the camera position and the position of the triangle. Now we need to test whether the distance of the triangle is too big or too small. This is done by comparison to some arbitrarily set threshold. Logically for triangles with different level of detail this threshold will be also different. If we are far away from the planet, the triangles are relatively big. Therefore the distance threshold for those triangles also has to be relatively far from the planet's surface. As we proceed closer to the surface the triangles are getting smaller and smaller and again so is the distance threshold for splitting. We implemented this as a Look-Up Table. This table is represented as an array. The index to this array is the level of detail of the triangle. The value in the array is the actual threshold for the distance. We can see that in Table 11 at line 9 where we have to look up the right threshold for the triangle based on its level of detail. This threshold is then at line 10 and 13 compared to our distance from the point.

If we are zooming into the scene then whenever the threshold is bigger than the distance we split the triangle and increase its resolution. On the other hand if we are zooming out from the scene we want to reduce the resolution. So when we test that the distance is bigger than the threshold, we have to reduce its resolution. Because during each splitting the triangle reduces the length of its edge by half, we also set each threshold as the half distance of the previous threshold. This way we manage to still keep more or less the same resolution of the planet model in the scene rendered on screen. An illustration of the thresholds is shown in Figure 4.8.

This implementation allows a triangle to change its level of detail by 1 in one iteration. If we eventually have a triangle that has to change its resolution by more than 1, then this is possible, but it will happen in the next few cycles of the dynamic level of detail. In other words, during the first iteration a triangle with insufficient level of detail is detected and its resolution is increased. After this

the scene is rendered on the screen. In the next iteration this triangle is detected again and its resolution is again increased. This goes untill the resolution of the triangle is sufficient.

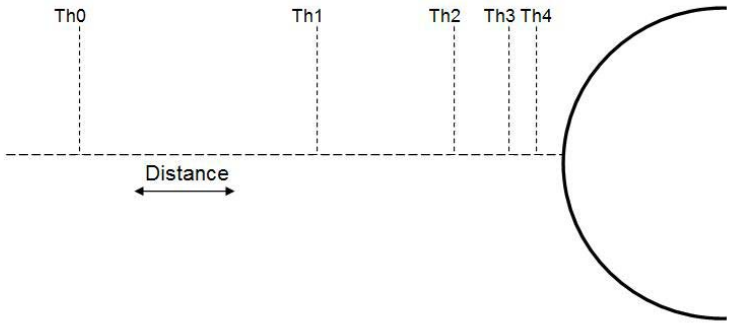


Figure 4.8: Thresholds for the level of detail of the planet's surface. Each threshold is placed in the middle of the distance of the previous one from the surface.

The look-up table presents a quite easy way how to adjust the performance of the whole application. We have to realize that these values of distance thresholds influence the number of triangles and points in the planetary model that we can see. As we said before if the model is too big it might require too much memory space or be too complicated to be rendered smoothly in real time. But the term "too big" mainly depends on the hardware specification of the computer that we are working with. So in order to adjust the application to work well on the given computer, we can increase or decrease the thresholds in the table. If we increase them then the resolution of the planet will increase when we are further away and we can see a bigger part of the surface. Therefore the structure will contain more triangles. If we reduce the thresholds, the resolution will increase later and there will be fewer triangles in the structure, because just a smaller part of the planet is visible.

4.5.2 Updating Mesh

Once we have a triangle, resolution of which we want to change, we have to make sure that we will do it the right way. The first issue is keeping track of parent triangles that are being removed from the mesh because they are substituted by their children when the resolution is increased. Later on if we want to reduce the resolution of the mesh again, we must recall this parent triangle from memory and place it back into the scene instead of its children. The second problem is that increasing the resolution of one triangle affects also its neighborhood. Without taking into account the level of detail of neighboring triangles, unwanted artifacts like cracks and various terrain distortions may occur.

Parents list

Our need to remember all parent triangles that were removed from the mesh, results in our having additional linked list in the mesh data structure. This list is called the Parents List and its implementation is identical to the list of all triangles in the scene shown in Figure 4.4. Triangles from this list are not used for the rendering of the scene. They are only stored in the memory for the case that we need to recall them and add them back into the scene. In Table 12 is a detailed description of methods *Split()* and *Reduce()* from lines 11 and 14 in Table 11.

The pseudo code in Table 12 should be more or less self explanatory. When we split the triangle, we have to create 4 children triangles and place them into the mesh. Afterwards we remove the parent triangle and we store it in the Parents List. When we reduce the resolution we are doing inverse procedure to splitting. We take all 4 children and recall their parent triangle from the Parents List. We place it into the mesh and we delete all 4 children.

```
1 Split (triangle T){
2     Children = Create 4 children triangles of T;
3     Remove T from Mesh;
4     Add 4 Children into the Mesh;
5     Add T into the Parents List;
6 }

7 Reduce (triangle T){
8     Parent = Recall parent of T from Parents List;
9     Children = All children of Parent; /* including T */
10    Delete Children;
11    Add Parent into the Mesh;
12    Remove Parent from the Parents List;
13 }
```

Table 12: Split() and Reduce() functions

Perhaps one quite interesting part is at lines 8 and 9 when we have to find all 4 children triangles. The input for the *Reduce()* function is one single triangle that was chosen for a reduction of resolution. But in order for us to do that we must know where all 4 children are and not just the one triangle that we know about. So we recall the parent triangle and we find the other 3 children based on its children pointers. This way we can find all 4 children triangles in a constant time if only one them is given.

No cracks

When we use dynamic level of detail, we have a model of a planet detail of which varies in different location. We can see that there are areas with higher detail, which means smaller triangles and areas with lower detail, which means triangles of a bigger size. The border between those areas where triangles of different sizes meet represents a problem for smooth connection of the triangles together without any artifacts.

The problem is with the point that two neighboring triangles share. This was already discussed in section 4.4.1, where we wanted the neighboring triangles to share this point. Now we have a problem with the point's displacement. If one of neighboring triangles increases its resolution it adds a new point to the conjunctive edge. In order for us to increase the resolution of the terrain we have to displace this new point to the elevation that it is supposed to be at, based on the terrain generating algorithm. However if we do this before the neighboring triangle increases its level of detail to the same level, a crack will occur. This is shown in Figure 4.9.

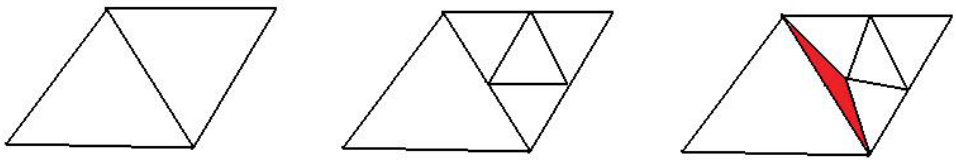
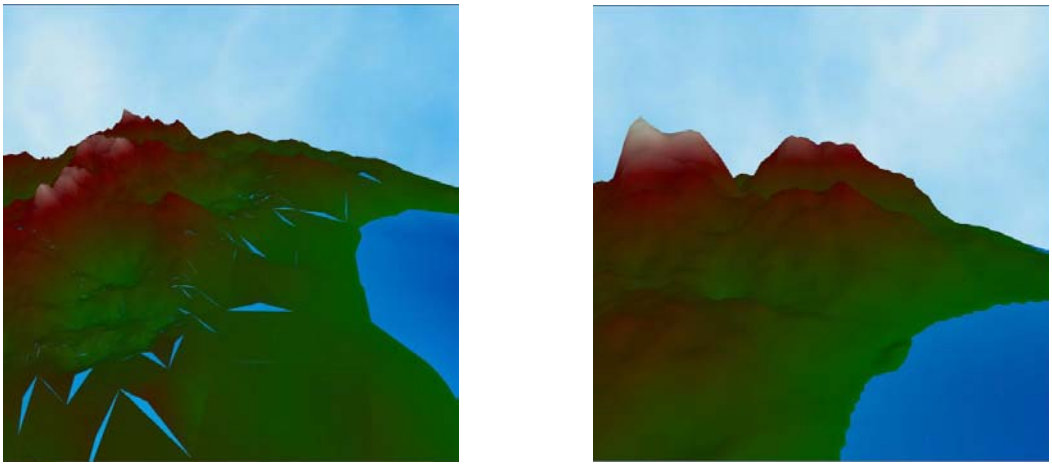


Figure 4.9: Splitting triangles causing crack to occur. (a) Two adjacent triangles. (b) The right triangle increases the resolution. (c) Crack occurs after displacing the new point.

To avoid the cracks from occurring, we have to first follow one simple rule. The difference in the level of detail of two adjacent triangles must be never bigger than 1. This is hidden at lines 10 and 13 in Table 11 in functions *fineToSplit()* and *fineToReduce()*. Before we increase the resolution of the triangle we have to look at the adjacent triangles and check if their resolution is not smaller then ours.

If it is then we cannot split, because splitting would result in having two neighboring triangles with the difference of the level of detail of 2. We have to watch this as well when we reduce the resolution.

Once we make sure that this condition is fulfilled, it is quite easy not to generate cracks. If the neighbor has the same level of detail as the splitting triangle, than the new point that is created is not displaced but left on the conjunctive edge. But if our neighbor has a higher level of detail, we share the point on the conjunctive edge and displace it accordingly. In other words the point on the conjunctive edge is only displaced if triangles on both sides of this edge have the same level of detail. If either of them has a lower level of detail then we cannot displace this point, because doing so would create a crack in the mesh. Figure 4.10 shows a comparison of two terrain patches, where one was created without taking care of the cracks and the other does take care of them. More detailed work on other methods of crack avoiding could be found [37] or in [38].



(a) (b)

Figure 4.10: Comparison of terrain patches with and without cracks. (a) Terrain patch generated WITHOUT taking care of the cracks. (b) Terrain patch generated WITH taking care of the cracks.

4.5.3 Problems and Known Issues

Although the Dynamic Level of Detail algorithm, in the way that we have implemented it, is working considerably well and certainly improves the performance of the application, there are still several problems that need to be solved. Despite the fact that we have not come up with a solution yet, we will mention here these problems. Modifying the algorithm and taking care of those issues is a subject of future development.

Update of colors and normals

When the level of detail generates new triangles it adds new points to the mesh. In order for OpenGL to render the terrain the right way we have to specify what the normal of the surface in the point is and also what its color is. To do this we would have to calculate this for all new points during every iteration of the dynamic level of detail. However this would require significant amount of computational time and really slows down the whole rendering process. Especially when using some kind of a Perlin Noise algorithm for the coloring the computation might be quite costly. For the normals, we have to realize that adding new triangle into the mesh might change the position of several neighboring triangles. Therefore normals of all of those triangles would have to be recalculated. With the data structure that we have designed it is impossible to keep track of all triangles that might be affected by adding a new one. So the only option would be to calculate the normals of all points in each iteration, which requires too much computational time.

Our temporal solution to this issue is that the user can manually say when the normals and colors are supposed to be calculated. During the generation of the new points we assign them only the average color of their parent points and we assign them some default normal. This way user can quite

smoothly explore the planet surface and when he desires he can manually choose to calculate the proper colors and normals for the given view.

Improved back face culling

In section 4.5.1 we have explained why a classical Back Face Culling algorithm is not suitable for our purpose. Leaving out all triangles that were facing away from us from the dynamic level of detail had a negative impact on the look of the terrain. In the same section we proposed a simple technique for determining which triangles are on the front and what are on the back side of the planet. However this method still involves too many triangles into the dynamic level of detail and thus the resulting mesh structure is too big. This means that unnecessary too much computational time is needed for the rendering and other operations with the mesh, although it does not provide us with any improvement of the detail of the generated terrain. Hence a better algorithm for deciding which triangles will be actually rendered on the screen is needed in order for a better performance of the whole application.

4.6 Terrain Generation

The implementation of the terrain generation algorithms is quite straight forward. It follows the pseudo codes presented in section 2.3. The most important part of the implementation was to make sure that the algorithms will have a direct access to all triangles and points as they required. This was ensured by the data structure that we have designed and described in section 4.4. The displacement rules for particular points follow the principles described in section 2.3 without any major modifications. So instead of repeating here what was already explained once in the survey part of this work, we will mention here couple of things that we consider to be quite interesting and that have not been discussed yet. In this section we will describe the implementation of the water layer and the implementation of the crater generation algorithm.

4.6.1 Water Layer

An important part of the planetary model is the water layer. Especially on earth-like planets the water covers up to 2/3 of the whole surface. One way to implement this is to override the terrain generation algorithm everywhere where the altitude is lower than some arbitrarily chosen water level and leave these points at the water level. This approach brings several difficulties like problems with coloring of the sea shore or problems with changing the water level. In this application we have implemented a different approach that requires more memory space but is certainly easier to implement and also overcomes both above mentioned problems.

In my implementation the water layer is represented as another sphere. This sphere is completely separated from the mesh structure holding the planet model. So we could actually see it as another planet. This sphere is placed at the same center point as the planet itself. The desired water level at the planet is equal to the diameter of this sphere. When OpenGL renders the scene, all surface that is under the water layer will be covered by the water sphere, and thus it creates an effect of water on the planet. We can quite easily change the height of the water level by enlarging or reducing the water sphere's diameter.

One issue connected with creating the water layer this way is its coloring. We have implemented a simple coloring of the water based on the linear interpolation of tones of blue color based on the depth of the see. But in order to do that we have to know what point on the see bottom matches with appropriate point on the water layer. This is not quite easy task because there is no way how to find the appropriate points in the linked list that was created with the iterative algorithm that was described in section 4.4.1.

My solution to this problem was inspired by a simple assumption that if we generate two identical sphere structures with the iterative algorithm then the ordering of the points and triangles will be the same in both linked lists. Also another thing to realize is that the water layer is not going to be represented by a mesh with such a big resolution as the resolution of the planet itself. This is because it does not need to display so much detail as the planet's surface. So during the initialization of the

application when we are creating the planetary model, we create both the planet and the water layer to a certain level of detail. Then we iterate simultaneously through both linked list containing all points in the planet and water mesh and we set a pointer from each point in the water layer to the appropriate point in the planet mesh. Once this is done, we can further increase the resolution of the planet, but the points in the water layer will keep track of where the appropriate point on the sea bottom underneath them is located and they can set their color accordingly. For this purpose there is a pointer called *waterPoint* in the Point object shown in Figure 4.4. It is used only for the water layer. For the terrain mesh this pointer is set to null and not used.

In Figure 4.11 we can see an illustration of how the water layer works. We can see on the left picture of a terrain generated with Perlin Noise algorithm. The right picture displays the same terrain after adding the water layer. We can see how all points on the planet under the water layer are covered by the water sphere. Also it is apparent how the water layer is colored based on the depth of the sea.

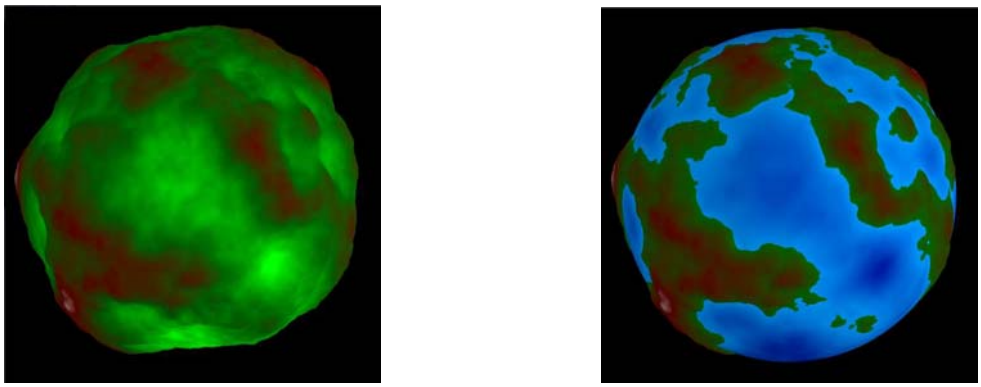


Figure 4.11: Adding water layer into the model.

4.6.2 Craters Implementation

In addition to the fractal generation algorithms that generate the whole surface of the planet we have also implemented an algorithm for adding craters onto the surface of the planet. This is mainly for the purpose of creating a Moon-like looking planet. The generation of craters has two main parts. One part is creating the actual crater. We have to displace the terrain around some previously chosen mid-point and create the right shape of the crater. The second and also very important part of the algorithm is a function that will distribute the craters randomly over the planet's surface.

Modeling of a crater

If we want to model the shape of a real crater we have to keep in mind its regular round shape but also its randomness, which is caused by different angle of impact or by erosion that took place after the crater was created. Figure 4.12 shows examples of real Moon craters. We can compare those to the craters generated by this application in Figure 5.10.

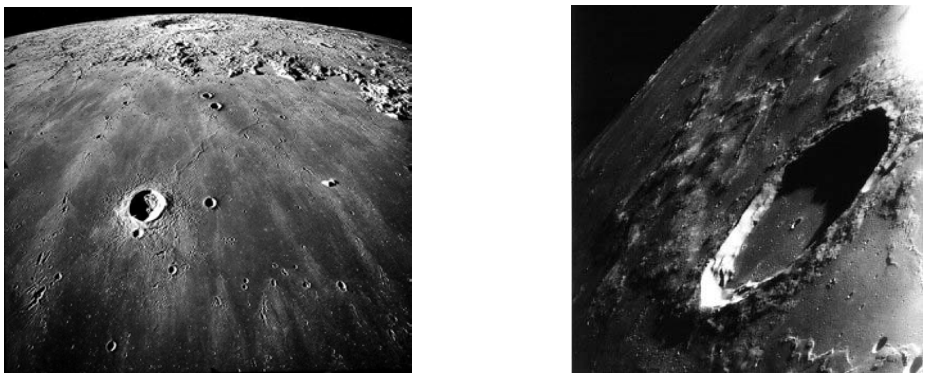


Figure 4.12: Moon craters. (Images taken from [33] and [34])

The regular shape could be implemented as a several concentric circles that separate different zones in the crater. The middle quite flat area of the crater is ended by the inner rim. From this point the crater is raising till it reaches the maximum at a certain distance from the mid-point that we can call the outer rim. Now the crater is steeply declining down until it reaches its end, which is at a distance that could be called the size of the crater. Figure 4.13 shows a schema of the zones inside the crater.

The actual algorithm is implemented as a set of different displacement functions. First we find in which zone the current point lies based on its distance from the mid-point. Consequently we displace the point based on its position within the zone. The smooth rising and declining of the edge rim of the crater is approximated by a smooth step function that uses a cubic interpolation between two defined elevations.

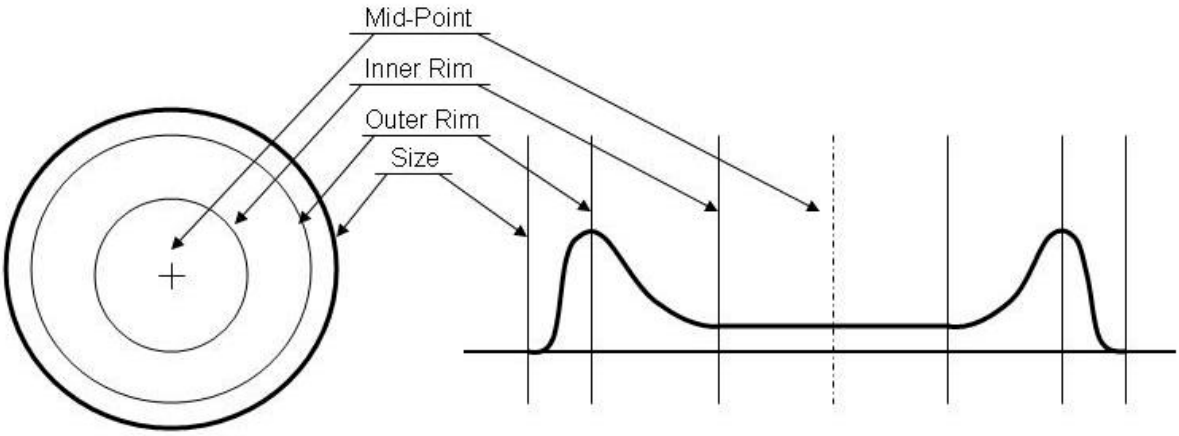


Figure 4.13: Schema of different zones inside the crater.

The randomness is implemented by incorporating the perturbation function that was described in section 2.4.5. We used one perturbation function to perturb the elevation displacement of the crater. This causes that the rim is not lifted into the same elevation everywhere. Further more we used another noise function to perturb the distance of the different zones from the mid-point. This resulted in the craters not regular round shape everywhere but instead in a certain amount of randomness in the model.

There are several ways how to control the final appearance of the crater. The most important ones are the size of the crater and the distances of different zones in the crater from the mid-point. In addition to this we can also control how big will be the vertical displacement relatively to the size of the crater and also different parameters of the perturbation functions will affect the final look of the crater. An example of the crater is shown in Figure 5.10.

Distributing function

It is completely random where the meteorite hit the planet's surface and creates the crater. We would like to have the same distribution of craters in our model. In order to that we had to come up with a function that will return a random point on the planet's surface that will be used as the center point of the crater.

We have tried several ways of distributing the craters randomly over the planet including constructing random vectors starting at the center point of the planet and then finding the point in the mesh that was the closest one to this vector. But all of those methods were either too computationally demanding or the distribution of the craters was not uniform and the algorithm tended to place more craters into the same areas. Finally we used a very simple algorithm that takes advantage of the linked list data structure that holds all the points in the mesh.

Because we know the number of points in the mesh, we can simply generate random integer number from 1 to the number of points. Then we iterate through the linked list holding all the points and we output the point that is at a position with ordinal number equal to the random number that we

have generated. This simple algorithm works fast and it randomly and uniformly distributes the craters all over the planet.

4.7 Terrain Coloring

In this section we will describe my actual implementation of the coloring algorithms for the planet. In section 2.4 we have talked about basic techniques for color interpolation and for incorporating fractal algorithms into the coloring. Here we will explain how were these techniques used and combined together in order to create various coloring schemes. In this section we are only going to describe our motivations for creating that particular coloring scheme. Comparison and some evaluation of presented techniques will be a subject of the following chapter.

One important thing to understand here is that the final look of any coloring model will highly depend on the actual color tones that we use for the interpolation. Not even the best coloring algorithms cannot create anywhere near realistically looking planet if we are not using the right color palette. During our work on this project we have found out that choosing the right color palette is a very difficult task and it takes a lot of time to pick the really good and realistically looking colors. So far we have not included an option into the GUI for creating user's own color palette. However this would enable the user to adjust the colors so that they fit the desired look and it is certainly a feature that we are planning on including into the application in the future.

4.7.1 Altitude Based Coloring

This coloring model was inspired by a simple assumption explained in section 2.1.2 that the color of terrain is in many cases dependent on the elevation. We have used linear interpolation between three predefined colors. First green color symbolizes lower altitudes, mainly flat parts of the planet. Hills stretching to little bit higher altitudes are marked with brown color. The last section of mountain tops is colored with white color to symbolize snow cover. The control variable for the interpolation is the elevation of the point on the surface. Therefore all the points on the planet with the same altitude have the same color. For example the water layer is using this kind of a coloring scheme as well. In that case the water is colored based on the depth of the sea bottom underneath. Example of planet colored with this coloring model is in Figure 4.14a.

4.7.2 Altitude Based Coloring with Perturbation

This coloring model is very similar to the previous one. The only difference is that we perturb the altitude value that controls the interpolation according to the principle described in section 2.4.5. The motivation for this is that with a simple altitude based coloring all points with the same altitude have the same color. This might create unnaturally looking coloring in some cases. Therefore we perturb the altitude by adding some noise to it. This way the altitude zones are not constant but they vary based on the perturbation.

Because altitude zones are a low scale feature and adding the perturbation does not change the appearance of the whole planet from a bigger distance, we did not include an image of a planet with this coloring into Figure 4.14. At this scale the planet would be almost identical to the planet in Figure 4.14a.

4.7.3 Perlin Noise Coloring

In this case we wanted to color the terrain with a completely random pattern using the fractal algorithms. This coloring uses the 3-dimensional Perlin Noise function for coloring of the planet's surface. We use the coordinate of each point in the 3-dimensional space as the input for the Perlin Noise function. The response of the Perlin Noise function in each point controls the linear interpolation between two colors. In this particular case we have chosen green and brown color. Because we are using 3-dimensional function the coloring is smoothly distributed everywhere on the planet and there are no artifacts caused by connecting different 2-dimensional textures together. An example of planet colored with this coloring model is in Figure 4.14b.

4.7.4 Altitude Based + Perlin Noise Coloring

This coloring is a mixture of the previous methods. The main motivation for this one is to incorporate some randomness into the quite regular coloring pattern created by simple altitude based coloring. We have achieved this by using the linear interpolation controlled by the altitude in two sets of four colors. In each set there is a color for lowlands, one for highlands, one for mountains and one for snowy mountain tops. The coefficient of blending those two sets together is determined by the Perlin Noise function. For this purpose we are using 3-dimensional Perlin Noise function, which response is calculated based on the point's coordinates. This way we can see an altitude dependency of the color but the coloring also appears to be quite random and has more natural look. Example of planet colored with this coloring model is in Figure 4.14c.

4.7.5 Turbulence Coloring

In section 2.4.4 we have described how we can modify the output of the Perlin Noise algorithm to create more interesting coloring patterns. The turbulence function interpolates between two colors based on the absolute value of the response of the Perlin Noise function. Because this modification is often used for creation of fire-like textures we have used black and red color for the interpolation. Example of a planet colored with this coloring model is in Figure 4.14d.

4.7.6 Earth-like Coloring

This coloring is a mixture of several principles described in section 2.4. The motivation for it is to create an Earth-like looking planet. We tried to create several vegetations zones that depend on the latitude of the point as shown in Figure 2.1. We also wanted to add certain amount of randomness by incorporating the Perlin Noise function into the model. And finally also the altitude dependency of the color was our goal.

We used the spline function to create several vegetation zones based on the latitude of the point on the planet. In order for these zones not to be just straight stripes we perturb the latitude by adding certain amount of noise to the latitude. Another step was to add randomness and detail by incorporating Perlin Noise algorithm. We created another spline function determining another set of vegetation zones but with a little bit different colors for each zone than the first spline. A 3-dimensional Perlin Noise function was used to specify how the colors from these two splines are blended together. Last thing to do was to add the altitude dependency. In order to do that we have created another pair of spline functions that specify the color of the mountain tops. These two were also blended together based on the 3-dimensional Perlin Noise function. The final color of the point was obtained by linear a interpolation based on the altitude of the point between the results of the blending between the two pairs of spline functions. Example of planet colored with this coloring model is in Figure 4.14e.

4.7.7 Gradient Based Coloring

This coloring model is inspired by the dependency of the terrain color on the gradient of the surface. On steep hill-sides there are no trees but only rocks. We can see the same thing on the mountains where the snow slides down from areas that are too steep. In this coloring scheme the color is linearly interpolated between two colors based on the normal to the surface in the given point. Example of a planet colored with this coloring model is in Figure 4.14f.

4.7.8 Moon-like Coloring

In order to create an impression of a Moon-like looking planet we have designed a special coloring model that was inspired by terrain features and color tones that could be seen in Figure 2.0b. This coloring model does not only color the terrain but also changes the color of the water layer to.

In the real photographs of the Moon we can see that it has big areas of darker grey color that are called seas (for example Mara Tranquillitatis, Mare Imbrium or Oceanus Procellarum). According to [29] these are not seas filled with water but they used to be big pools of lava. So in order to create those lunar seas we simply used the water layer. We colored it with a dark grey tone and for a little bit more realistic impression we used a Perlin Noise function to blend two dark grey tones together.

The rest of the planet is a quite random and rough terrain with a lot of craters and other terrain formation. This area is represented by the surface of the planet that is above the sea level. It is colored in brighter tones of grey and we used a simple altitude based coloring to emphasize the terrain differences. The craters are added later into the model by applying the craters generation algorithm that was described earlier in section 4.6.2. The complete Moon planet model is shown in the comparison chapter in Figure 5.10. An example of the Moon-like coloring is shown in Figure 4.15.

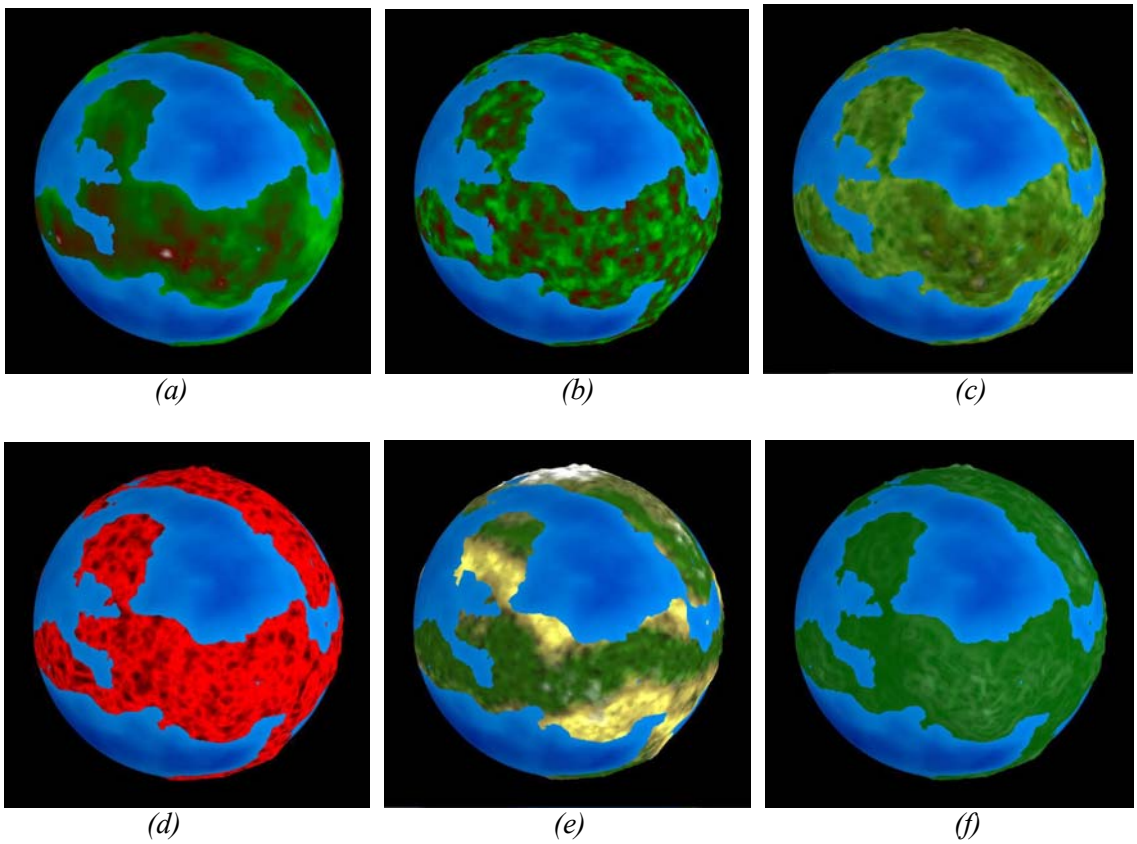


Figure 4.14: The same planet model colored with different coloring algorithms. (a) Altitude dependent. (b) Perlin Noise. (c) Altitude dependent + Perlin Noise. (d) Turbulence. (e) Earth-like. (f) Gradient dependent.

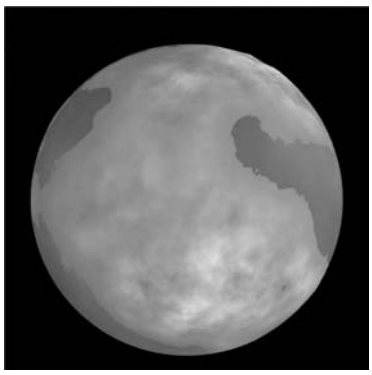


Figure 4.15: Planet colored with Moon-like coloring.

4.8 Graphical User Interface

We have used the glui library to create a simple graphical user interface for this application. The main purpose of this GUI is to give the user an access to some key parameters that are used for controlling of the terrain generation. We can also choose the terrain generation algorithm that will be used for the creation of the planetary model. Consequently one of several coloring schemes could be chosen and applied to the surface. Furthermore the model could be enhanced by several additional effects like adding the water layer, displaying the sky or adding craters to the surface. Another important option is to choose the terrain generation algorithm that will be used for the dynamic level of detail and whether to turn it on or not. The GUI also displays some important characteristics of the geometrical model of the planet. For example it shows the maximal dynamic level of detail in the scene or the overall number of triangles in the mesh. An example of the GUI is shown in Figure 4.16.

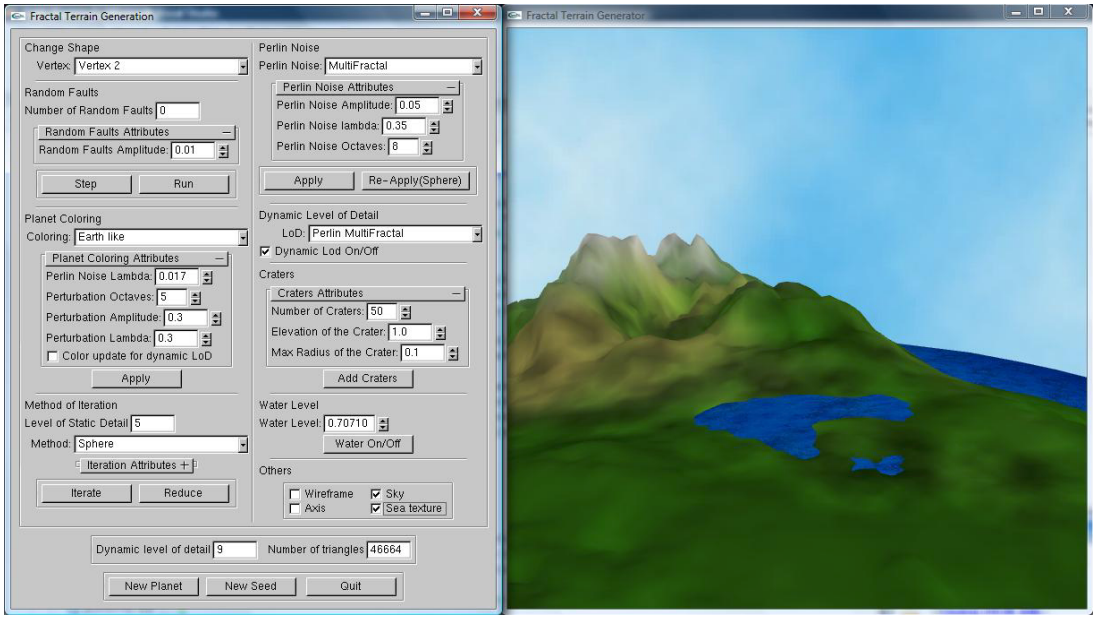


Figure 4.16: Example of the work space. On the left is the graphical user interface. The right window displays the rendered terrain.

In addition to the control features of the GUI, user can use the mouse and several key buttons to freely explore the planet's surface. Complete description of all control blocks and available keyboard control could be found in the User Manual in Appendix A.

5 Comparison of Algorithms

This chapter is supposed to be sort of an experimental results chapter. Although it is sometimes very hard to compare different terrain models and evaluate them based on how realistic they look, I will try to do that here and draw some conclusion about what algorithms for terrain generation and for terrain coloring are the most suitable ones for creating a planetary model. Because everyone has a subjective opinion about which planet or mountain range looks better or more realistic we will also look at the technical aspects of the algorithms. We will compare them according to how the results could be influenced by different parameters. This is a very desired property that enables us to quite easily influence the final look of the terrain.

First we will look at what coloring models produce the best looking coloring of the surface and how some of the models could be modified by adjusting some parameters. Then we will follow by probably the most important part of this chapter, the comparison of terrain generation algorithms. We will present several examples and explain how the generated terrain satisfies our demands that we have stated in section 2.1. Also we will show how the look of the generated terrain is dependent on different values of the input parameters. The last part of this chapter will deal with a qualitative and quantitative comparison of the static and dynamic level of detail. We will show examples of what the limits of static level of detail are and how dynamic level of detail overcomes these problems and significantly improves the performance of the whole application and gives us a better opportunity to study the generated terrain.

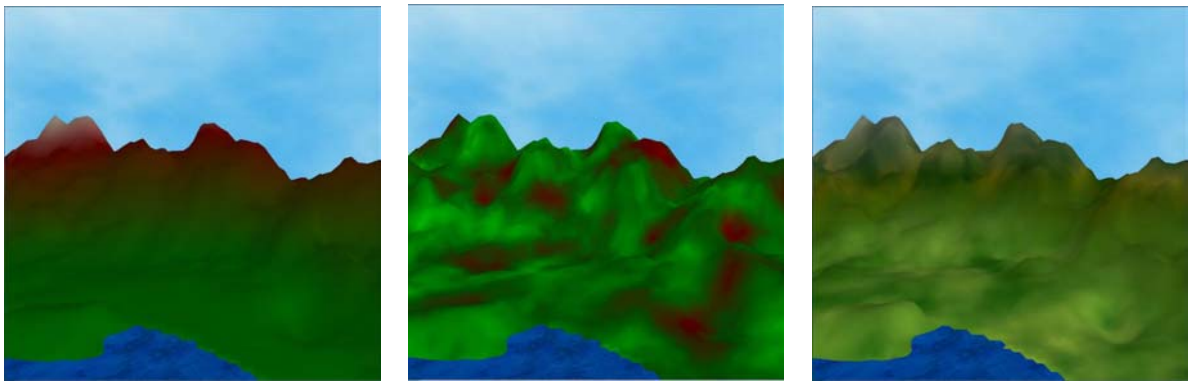
5.1 Comparison of Terrain Colorings

In Figure 4.14 we could see views of the whole planet colored with different coloring models. These images give us an idea of how the coloring is suitable for high scale planetary features. In this section we will add similar figure showing the same coloring models at low scale planetary features. Furthermore we will look at the most important parameters of the terrain coloring algorithms and how they control the final look of the coloring.

5.1.1 Coloring of Low Scale Features

In this section we will look at how particular coloring models perform on low scale terrain features. This means that we are interested in whether the coloring distinguishes between mountains and lowlands, whether it adds sufficient amount of detail to the terrain model or whether it emphasizes the plasticity of the terrain or not.

For this comparison we will use the same terrain patch for all coloring models. We have chosen a specific part of the terrain where we can find quite flat areas but also high mountains. This way we can observe all aspects of the coloring in one view. Figure 5.1 shows the rendered views.



(a) *(b)* *(c)*

(d) (e) (f)

Figure 5.1: The same terrain patch colored with different coloring models. (a) Altitude dependent. (b) Perlin Noise. (c) Altitude dependent + Perlin Noise. (d) Turbulence. (e) Earth-like. (f) Gradient dependent.

Emphasizing plasticity of the terrain

From Figure 5.1 we can see that some colorings are better in emphasizing the 3-dimensionality of the model and thus creating a better impression of the terrain. Perlin Noise and Turbulence coloring in Figure 5.1b and 5.1d do not emphasize the plasticity at all. This is because they are not altitude dependent models. We can see that other 4 models are quite good in this regard, because we can easily distinguish between different altitude zones. However simple Altitude dependent coloring in Figure 5.1a produces quite unnaturally looking coloring because the altitude zones are too visible. The best sense of plasticity probably gives us the Gradient based model in Figure 5.1f, because it emphasizes every change in elevation by changes in the color tone. The Perlin Noise + Altitude dependent and the Earth-like model in Figure 5.1c and 5.1e give quite good impression of the plasticity by interpolating the color based on the altitude.

Adding additional detail

The coloring can improve the final impression by adding additional detail into the model. We can see that Altitude dependent model does not add much additional detail, because it does not incorporate Perlin Noise at all. Gradient based coloring does not use Perlin Noise as well but it adds additional detail because it emphasizes every little curve in the terrain that would not be visible without the coloring. The remaining four models incorporate Perlin Noise algorithm and thus add quite big amount of additional detail into the model. However we can see that plain Perlin Noise and Turbulence coloring add the detail in a quite unnatural way. This might be due to the too contrasting color tones used for the interpolation. The best and the most realistically looking addition of additional detail is provided by the Perlin Noise + Altitude dependent and the Earth-like models. They add sufficient amount of detail that does not distract the eye from other terrain features. The Earth-like model adds additional diversity into the model, by having different vegetation zones based on the latitude. This is certainly a better property compared to the same color tone of Perlin Noise + Altitude dependent coloring all over the planet.

Conclusion about coloring models

From Figure 5.1 it is clear that the right planet coloring must add additional detail and must emphasize the plasticity of the terrain. Perlin Noise, Turbulence and Altitude dependent colorings have only one of those properties and thus do not produce such a good results. The Gradient based coloring has quite good results, however it should be combined together with some other techniques to produce more complex coloring model. The best results were achieved with Perlin Noise + Altitude dependent coloring and with Earth-like coloring. We can see in Figure 4.14 that these two models

created quite realistically looking planets even at a high scale. Producing good coloring at different scales was our main goal.

5.1.2 Parametric Control of Coloring Algorithms

One of the most important properties of fractal algorithms is their ability to be easily controlled by adjusting some key parameters. In this section we will look at how the coloring could be adjusted by this parametric control. Because there are many parameters and even more combinations of their values we have chosen only the most important cases or parameters that we have already mentioned in the previous text. We will look at how the perturbation of the altitude zones changes the coloring, how the lambda of the Perlin Noise changes the amount of additional detail added to the model and how the amplitude of perturbation function effectively influences the lay-out of vegetation zones in the Earth-like coloring.

Perturbation of altitude zones

In section 2.4.5 where we were describing the principles of perturbation technique, we mentioned that this technique might be used for modifying the altitude zones, so that they are not located at exactly the same elevations. This way we wanted to create more realistic look of the colored terrain. In Figure 5.1a we can see that the altitude zones seem quite narrow. For this reason we have implemented Altitude based coloring with Perturbation that adds some noise to the altitude of the point. Figure 5.2 shows examples of the same terrain with different perturbation functions.

Figure 5.2: Altitude based coloring with perturbation. The same terrain patch with different perturbation functions.

We can see from the examples in Figure 5.2 how the perturbation influences the snow line on the mountain. However we can not really say that this much contributes to the coloring being more realistic compared to the classical Altitude based coloring in Figure 5.1a. On the other hand various altitude of the snow line on different mountains on the planet contributes at least a little bit to the complexity and randomness of the overall model.

Perlin Noise lambda

Although it might not seam quite obvious at the first glance, the lambda parameter of the Perlin Noise function is the key parameter controlling the amount of additional color detail added to the model. It determines the size of the features created by blending the colors together according to the Perlin Noise function. The smaller lambda we are using the smaller is the size of those color features and consequently the higher is the amount of the optical detail added. If we use too big lambda then the features will be too big and the terrain coloring will not look natural. Figure 5.3 shows that there is a certain optimal value for the lambda of the Perlin Noise function.

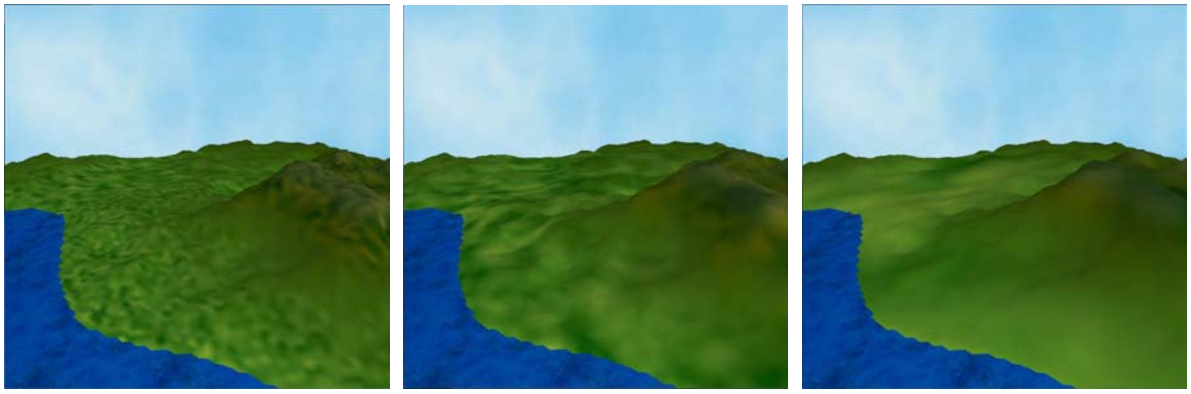


Figure 5.3: Different values of lambda for the Perlin Noise function used in Perlin Noise + Altitude dependent coloring. From left to right lambda = 0.001, 0.01, 0.05.

From Figure 5.3 we can see that the value 0.01 for lambda produces the color features of the optimal size. Value 0.001 adds too much additional detail while value 0.05 produces too big features. But we should keep in mind that those features look too big or too small relatively to the other terrain features. This means that if we are looking at the planet from a long distance the lambda value might be just right. But when we zoom in and we get close enough to the surface the lambda becomes too big for the size of the terrain features. Hence it is sometimes a problem to set the right value if we want to produce not too small color features at high scale but not too big color features at low scale.

Perturbation of vegetation zones

In order to create an Earth-like coloring of the planet, we have implemented several vegetation zones that are dependent on the latitude of the point as described in section 4.7.6. So as to add some randomness into the overall lay-out of those zones we have perturbed the latitude of the point by adding some noise to it. In the following Figure 5.4 we can see that very important parameter of the perturbation is the amplitude.

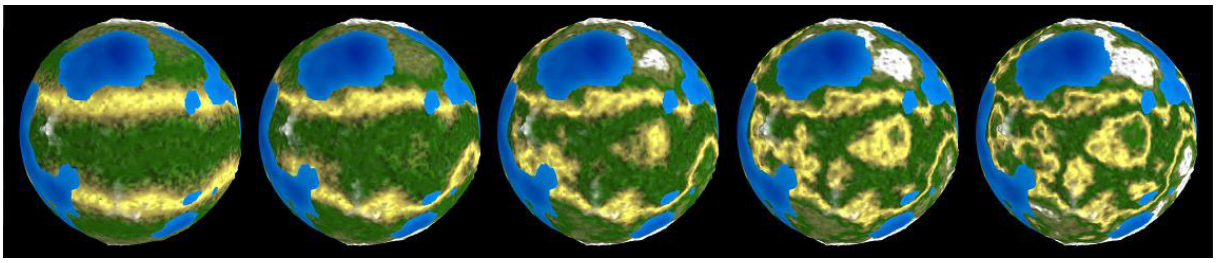


Figure 5.4: The dependency of the perturbation of the lay-out of vegetation zones on the amplitude of the perturbation function. From left to right Amplitude = 0.1, 0.3, 0.5, 0.7, 0.9.

The higher the value of the amplitude is the more scattered the lay-out is. Also here we can see that in order for the planet to look realistic there is a certain optimal value for the amplitude. Value 0.1 does not perturb the altitude enough, while values 0.7 and 0.9 add too much noise to it. The optimal are values between 0.3 and 0.5, which produce quite realistically looking lay-out of vegetation zones.

5.2 Comparison of Generated Terrains

In this section we will look at the differences between terrains generated by different terrain generation algorithms. We will describe the differences between high scale features, for example the distribution of continents, and low scale features, for example the look of the mountains. We will also look closer at the differences between homogenous fractal terrain and terrain with multifractal properties. Similarly to the previous section we will also look at the most important parameters and how they control the look of the generated terrain.

5.2.1 Differences between Planets

If we are looking at a planet from a bigger distance we are concerned about the high scale features like the size and distribution of whole continents, peninsulas and islands. Each algorithm that we have implemented produces planets with a little bit different look. Sometimes the differences are just subtle, sometimes they are more apparent. Figure 5.5 shows typical planets produced by different algorithms.

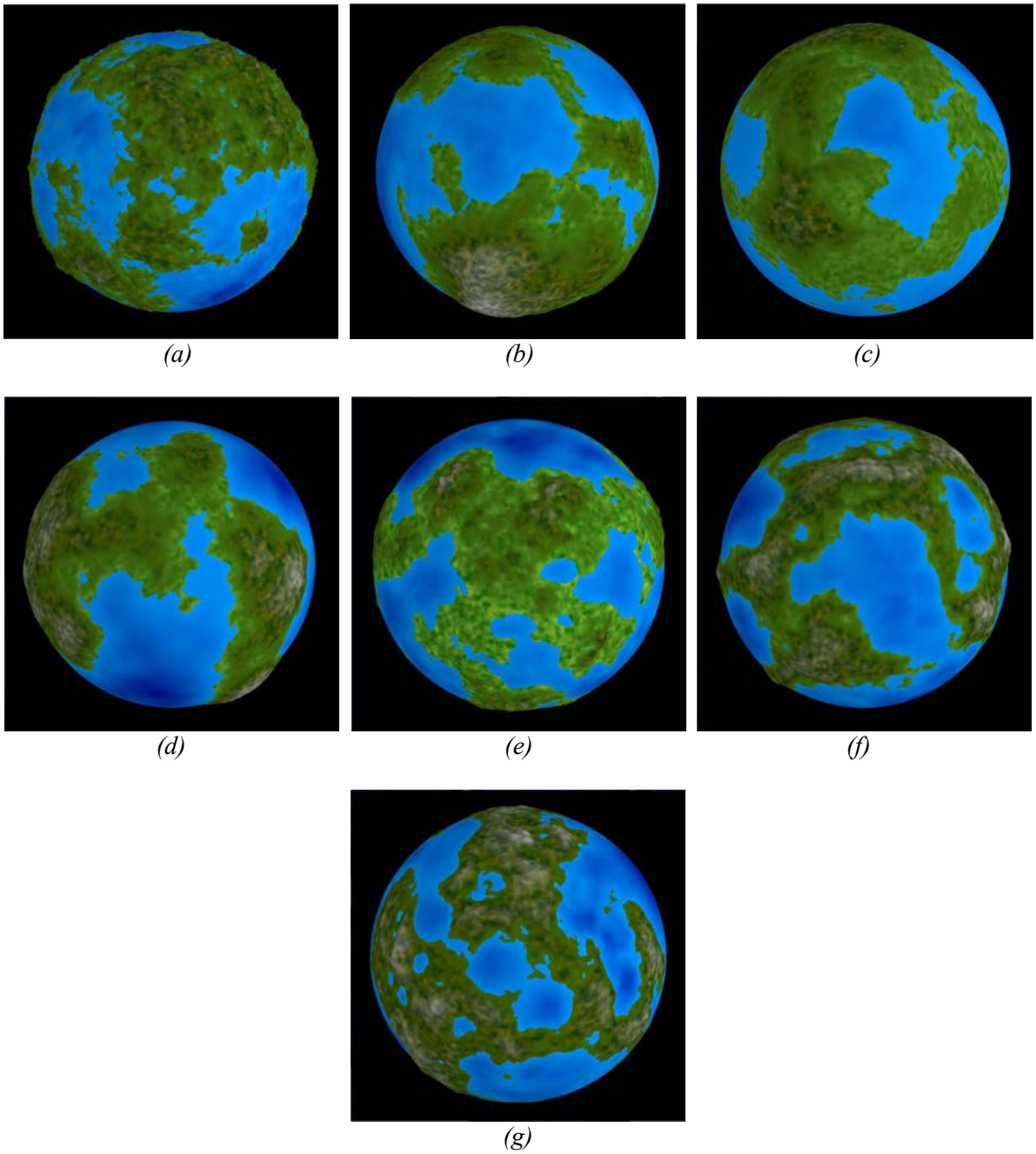


Figure 5.5: Planets generated with different algorithms. (a) Random Faults. (b) Mid-Point Displacement. (c) Mid-Point Displacement Multifractal. (d) Perlin Noise. (e) Perlin Noise Multifractal. (f) Perlin Noise Ridged. (g) Perlin Noise Ridged Multifractal.

Because the coloring has a quite big influence on the overall impression of the planet, we will use the Perlin Noise + Altitude dependent coloring for all terrains compared in this section. The reason for this is that this coloring produces quite nice looking terrains and unlike the Earh-like coloring it uses the same color tones all over the planet. Therefore no matter from what view we are looking at

the planet, the difference in the impression should be only caused by the differences in the terrain and not for example by different distribution of the vegetation zones, as it would be if we used the Earth-like coloring.

Another thing to note is that as we will explain further in this section, the look of the generated terrain is very dependent on the actual values of some parameters. However we cannot just use one identical setting of parameters for all algorithms. The same value of lambda will produce terrains with very different size of continents for the classical Perlin Noise and for the Ridged Perlin Noise algorithm. Or the same value of amplitude will result in terrains with quite different maximal altitude for non-multifractal and for multifractal algorithm. This is mostly due to the way the algorithms are implemented and the way each algorithm interprets and uses the parameters. We have tried to adjust these values for each algorithm to produce terrain with approximately the same features like the size of continents or the elevation differences on the planet.

Random Faults algorithm

In Figure 5.5a we can see a planet generated by Random Faults algorithm. As we have shown in Figure 2.10 the look of the terrain depends on the number of iterations of the algorithm. This particular planet was generated by applying 1500 iterations to a regular sphere. This is a sufficient number of iteration for any regularity to disappear. Compared to the other planets the terrain looks very broken and there are many small islands and peninsulas. In the matter of fact it might sometimes look too random. Although it is not the case of this particular planet, the algorithm often tends to create one big continent and one big ocean.

The Random Faults algorithm cannot be controlled by many parameters. The only attribute of the terrain that we can set is the amplitude of the displacement of the hemisphere. But even this does not ensure the final maximal altitude on the planet, because the displacements are randomly added together and the higher the number of iterations is the bigger is the maximal altitude on the planet.

Mid-Point Displacement algorithm

In Figure 5.5b and 5.5c is a planet generated with Mid-Point Displacement algorithm and Multifractal Mid-Point Displacement respectively. We can see that compared to the other planets this algorithm generates rather bigger compact continents with not many smaller islands or jagged coastline. If we look carefully at the coastline or some other terrain features emphasized by the coloring we can see some regularities and too synthetically looking formations. Sometimes the coastline is running straight forward without any random curving and so is revealing the underlying mesh structure.

The classical Mid-Point Displacement also produces unnaturally looking mountains, which look more like a one big plateau than jagged mountains. The Multifractal Mid-Point displacement has better results in this regard. Because there are much bigger differences in altitudes due to the higher elevations being rougher, the mountains colored with the altitude based coloring look more realistically. However this is still far away from the ideal planet model and from the mountains produced by some variation of Perlin Noise algorithm.

Also the Mid-Point Displacement algorithm is quite hard to control with parameters. We can quite easily control the maximal elevation of the terrain by setting the appropriate value of the amplitude. The size of the continents could not be adjusted by any parameter of the algorithm, however in section 5.2.3 we will show an easy way how to do this by combining the Mid-Point Displacement algorithm with the iterative sphere generating algorithm.

Perlin Noise algorithm

Figures 5.5d and 5.5e show planets generated with the classical Perlin Noise algorithm and with Multifractal Perlin Noise algorithm. We can see on these planets that Perlin Noise produces quite nice looking terrain with the right distribution of continents that have adequately forked coastline. The terrain is not too wildly jagged like in the case of Random Faults nor does it show any regular artifacts like in the case of Mid-Point Displacement.

Again we can clearly see the difference between non-multifractal and multifractal terrain. The classical Perlin Noise produces not so naturally looking mountains, which are nothing more than one big elevated area. The multifractal modification of the algorithm produces just smaller and randomly curved mountain ranges. Also on these two planets we can see one artifact of the altitude based coloring. The planet created with multifractal Perlin Noise has a little bit different tone of the color than the other planet. This is because there is a big altitude difference between the rather smooth lowlands covering most of the planet and the mountains reaching higher altitudes. In the case of the non-mutlifractal planet, the points are evenly distributed in all elevations and therefore the color is interpolated in a different way.

The reason why the planets generated with Perlin Noise show good characteristics is that the Perlin Noise algorithm can be controlled by parameters in many ways. We can easily adjust the size and distribution of continents by choosing different value of lambda for the Perlin Noise function or adjust the smoothness of the terrain by choosing the number of octaves to be included in the model. More detailed comparison of the influence of different parameters on the generated terrain is in section 5.3.2.

Perlin Noise Ridged algorithm

In Figures 5.5f and 5.5g are examples of planets generated with non-multifractal and multifractal Ridged Perlin Noise. We can see that the typical terrain features produced by those algorithms are long and quite narrow continents and peninsulas and round shaped and sometimes closed bays or seas.

Also as in the two previous cases the multifractality of the planet model affects the shape and the appearance of the mountain ranges. Due to the higher differences in elevation and the terrain being rougher the mountains have more realistic look.

The way of controlling the Ridged Perlin Noise is identical to the way we can control the classical Perlin Noise algorithm. We can adjust the distribution and size continents by appropriately setting the lambda parameter or set amplitude of the Perlin Noise function to determine the maximal altitude of the terrain on the planet.

5.2.2 Differences in Low Scale Features

As we have already said several times before it is important for the planetary model to be complex and look realistic at different scales. Section 5.2.1 compared the characteristics of whole planets. This section will look at the differences between terrain patches on the planet in a higher detail. We will be mostly interested in the properties of the terrain like the right shape of mountains or the coastline, the difference in the roughness between lowlands and mountains and also any artifacts that should not be in the model. Figure 5.6 shows typical examples of terrain patches created by different algorithms at the same level of detail.

In order to see a high-detailed view of the terrain the Dynamic Level of Detail algorithm will be used to generated additional points in the mesh. We will use the same algorithm for generating the new points as we used for the generating the whole planet. Although this might seem obvious, we have also some other possibilities. The architecture of this application enables us to for example create the planetary model with certain detail with Perlin Noise and then use Mid-Point Displacement algorithm to displace the new points added by the Dynamic Level of Detail. In this section we will not consider any such a combination of terrain generating algorithm, even though the user of the application has a chance to experiment with this and observe the difference in the generated terrain.

Because of the reasons described in the previous paragraph we will exclude Random Faults algorithm from the comparison of the low scale features. This algorithm is only applied to a solid with a certain resolution. Once we finish iterating the algorithm and we want to explore the terrain in a higher detail, there is no way how the Random Faults can generate additional points into the model. We don't have any track of how would be that particular point displaced by the faults. So in order to apply the Dynamic Level of Detail algorithm, we will have to use some Mid-Point Displacement algorithm to generate the new points. But the Mid-Point Displacement and its parameters like the amplitude of the displacements influence the final look of the terrain at the lower scale so much that is a question whether it should be any longer considered as a terrain created by Random Faults.

Therefore we will leave the Random Faults algorithm as an algorithm for creating a planetary model at higher scale and not as a suitable one for high detailed terrain model.

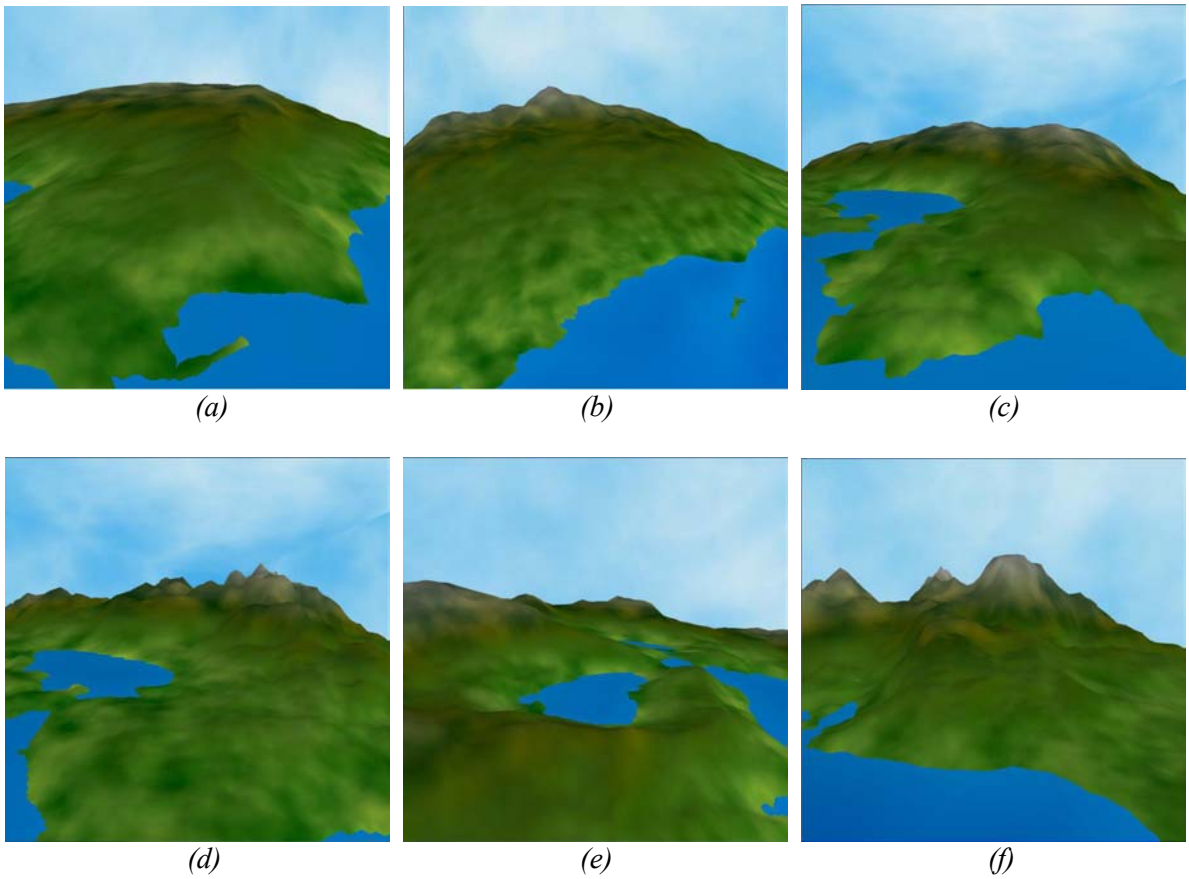


Figure 5.6: Terrain patches generated with different algorithms. (a) Mid-Point Displacement. (b) Multifractal Mid-Point Displacement. (c) Perlin Noise. (d) Multifractal Perlin Noise. (e) Ridged Perlin Noise. (f) Multifractal Ridged Perlin Noise.

Mid-Point Displacement algorithm

We can see in Figure 5.6a and 5.6b that non-multifractal and multifractal Mid-Point Displacement algorithms do not generate terrain with much detail at the higher level of detail. We could achieve rougher terrain by increasing the amplitude but then the differences in the altitude would look unrealistically too big relatively to the size of the planet.

Also the terrain looks a little bit too synthetically, especially in Figure 5.6a are still visible some straight lines revealing the underlying mesh. Similarly there is not much detail at the coastline and it looks rather straight than randomly curved.

One obvious thing is how the shape of the mountains is influenced by the multifractality of the algorithm. The skyline in Figure 5.6b much more resembles the skyline of real mountains. However there is still not much distinction between the lowlands and the mountains.

Perlin Noise algorithm

In Figure 5.6c and 5.6d are terrain patches created with classical and multifractal Perlin Noise algorithm. Compared to Mid-Point Displacement there is much more detail and randomness in the terrain. Also all terrain formations are nicely curved without any regularities or artifacts.

One obvious drawback of the model created with plain Perlin Noise algorithm is its homogeneity and the lack of any distinction between supposedly smooth lowlands and rough mountains. This is added to the model by incorporating the multifractal principles. In Figure 5.6d we can finally see a complex planetary model that contains smooth lowlands along the coastline and as we

proceed into higher elevation the terrain becomes rougher and jagged. Although a real terrain has much more complicated and detailed features, we can see that the mutlifractal Perlin Noise algorithm produces quite nice and realistic looking approximation of it.

Ridged Perlin Noise algorithm

The differences between classic and Ridged Perlin Noise mainly affects the high scale features like the distribution and the shape of the continents. Therefore in Figure 5.6e and 5.6f are terrain patches that have similar if not identical properties with the terrains shown in Figure 5.6c and 5.6d. For these terrain patches is more or less everything valid that we have said about terrain produced by classical Perlin Noise algorithm including the differences between non-multifractal and multifractal variation of the algorithm.

5.2.3 Parametric Control of Terrain Generation Algorithms

In this section we will look at how the output of the terrain generation algorithms can be influenced by setting different values for various parameters. There are not many options for the Random Faults and for the Mid-Point Displacement algorithms, but there is a huge variety of different settings of parameters for some variation of the Perlin Noise algorithm. This section is not supposed to be a complete list of all possibilities how to control the terrain generation. Rather than that we are only going to write about three main ways how to control the generated terrain. We will look at a little bit tricky way how to control the size and distribution of continents generated by the Mid-Point Displacement algorithm, how the look of the whole planet is influenced by different values of lambda and the number of octaves of the Perlin Noise algorithm and also how the number of octaves influences the amount of detail when we generate a terrain with multifractal Perlin Noise.

The distribution of continents of Mid-Point Displacement algorithm

In section 5.2.1 where we were comparing the planet generated with Mid-Point Displacement algorithm with planets generated by other algorithms, we have said that there is no parameter of the Mid-Point Displacement that would enable us to control the size and the distribution of the continents on the planet. However there is a way how we can do that if we combine the generating of the planet's terrain with the algorithm for iterative creation of a sphere.

Figure 4.1 shows the stages of the creation of a sphere. When we create the whole planet with the Mid-Point Displacement algorithm we substitute these regular iterations by displacing the new points according to the algorithm. Because in every iteration of the algorithm the amplitude is reduced in half, we can say that mainly the first couple iterations are responsible for the size and the distribution of the continents. Roughly speaking the size of the triangles in the mesh at the beginning of the algorithm approximately determines the size of the continents created by the algorithm. Therefore if we first increase the resolution of the starting octahedron with the iterative sphere generation algorithm and then start applying the Mid-Point Displacement, we will create terrain with smaller continents and they will be more randomly distributed over the planet's surface. Figure 5.7 shows the dependency of the planet's terrain on the level of detail of the regular sphere generated prior to applying the Mid-Point Displacement algorithm.

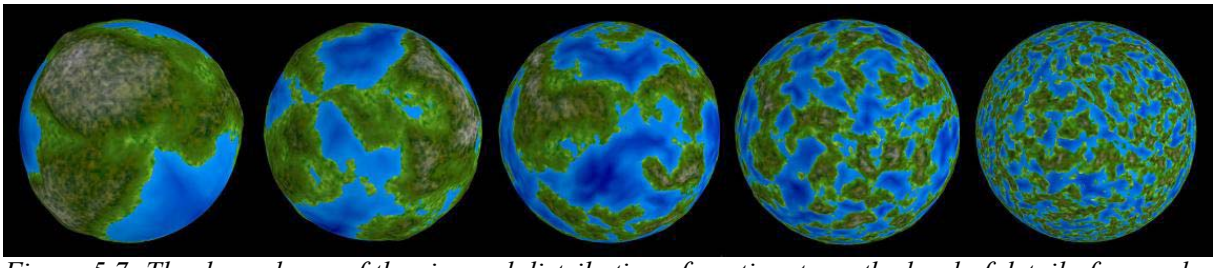


Figure 5.7: The dependency of the size and distribution of continents on the level of detail of a regular sphere created prior to applying the Mid-Point Displacement algorithm. From left to right the level of detail of the sphere was 0, 1, 2, 3, and 4.

Lambda and number of octaves of Perlin Noise algorithm

There are many ways how to control the appearance of the terrain produced by the Perlin Noise algorithm. If we are looking at the whole planet from a bigger distance, we are mainly interested in the size and distribution of continents. The two most important parameters affecting those properties of the model are the lambda and number of octaves of the Perlin Noise function.

The lambda determines the size of the continents. Because we reduce the amplitude in half for every consecutive octave, only the first couple octaves determines the size of continents and islands, while the other octaves adds only additional detail into the model. So by changing the lambda of the first octave we can control the size of the continents.

The number of octaves influences the detail of the whole model. If we use only small number of octaves the produced terrain will consist only of smooth rather flat terrain formation. Adding more octaves will result in adding more detail and randomness into the terrain.

Figure 5.8 shows the dependency of the planet look on those two parameters. We increase the number of octaves in the horizontal direction and the lambda in the vertical direction. From these images we can see that there is a certain optimal value for the lambda parameter in order for the terrain to look realistic. We can see that if we want to create an Earth-like looking planet the lambda should be somewhere between 0.2 and 0.3. Also we can notice that there is almost no difference between planet generated with Perlin Noise with 4 and 5 octaves. This is because the resolution of the mesh at this level of detail is not capable of displaying the highest octaves. Those would be revealed if we zoom closer to the surface.

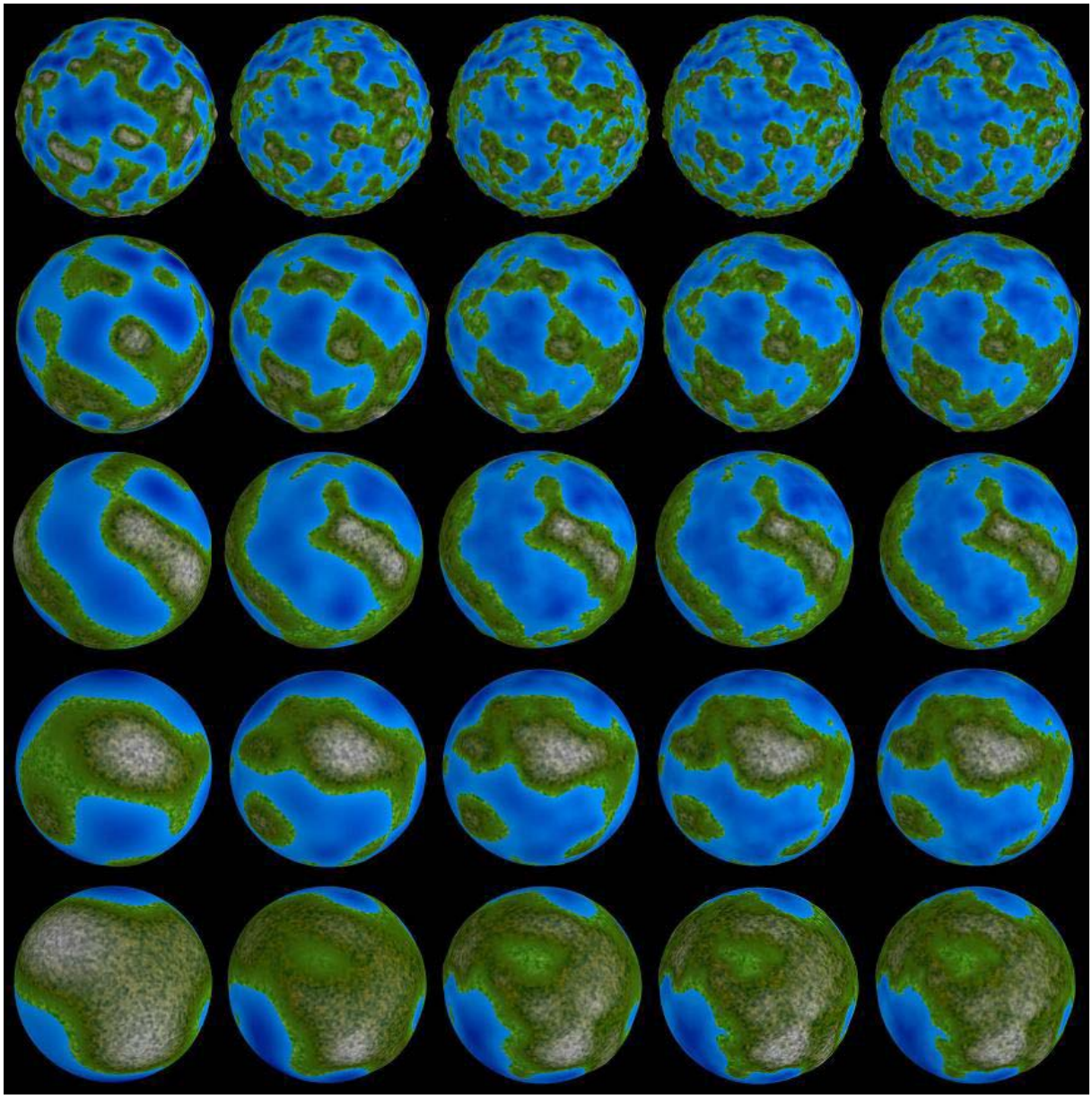


Figure 5.8: The effect of different values of lambda and the number of octaves on the terrain generated with Perlin Noise algorithm. From left to right octaves = 1, 2, 3, 4, 5. From top to bottom lambda = 0.1, 0.2, 0.3, 0.4, 0.5.

Number of octaves of multifractal Perlin Noise algorithm

In addition to looking at how the number of octaves influences the appearance of the whole planet, it is also interesting to look at how it influences the look of a high detailed terrain patch. The observation is even more interesting if we generate the model with multifractal Perlin Noise. In this case we can clearly see how the influence of higher order octaves is emphasized at higher elevation and reduced near the sea level. We can see these effects in Figure 5.9, where are 8 images of the same terrain generated with multifractal Perlin Noise. Each consecutive terrain was generated with function using one more octave then the previous one.

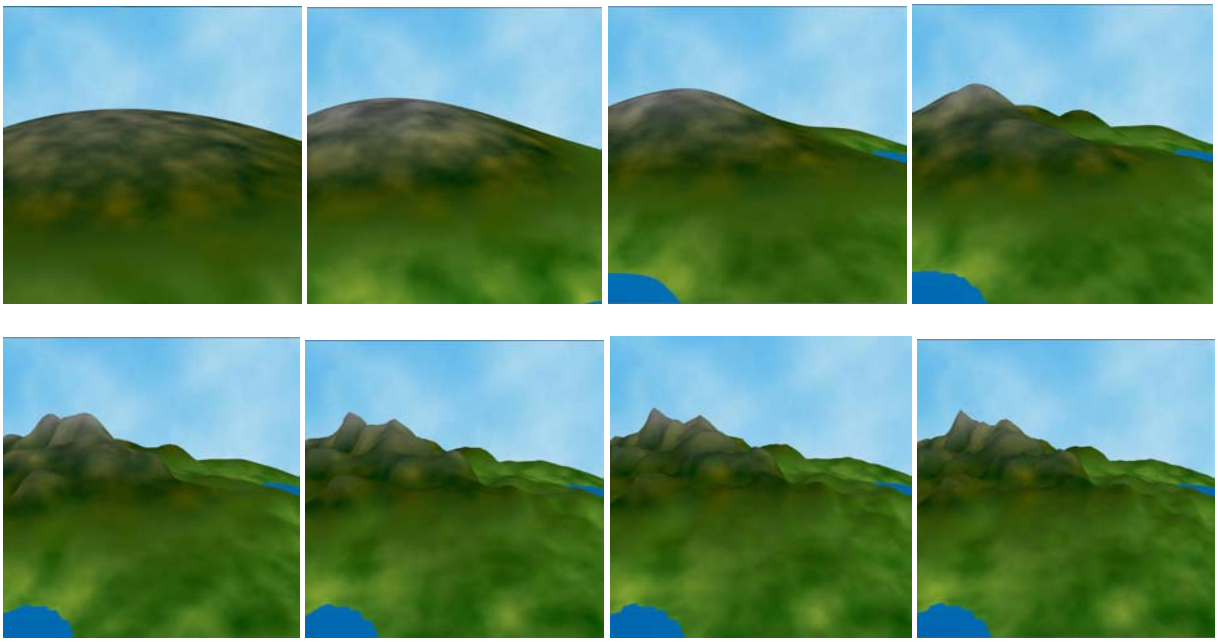


Figure 5.9: The influence of the number of octaves on the terrain generated with multifractal Perlin Noise. From left to right octaves = 1, 2, 3, 4, 5, 6, 7, 8.

5.2.4 The Moon

In this section we will look at the results of our experimenting with generating a Moon-like looking planet. One part of this model is the right coloring that was already described and shown in section 4.7.8. The other part is adding craters to the model. As was already explained in section 4.6.2 we can control the shape and the final look of the craters on the planet in several ways. We have tried to find the optimal size, elevation and also the number of craters so that the planet looks as much as Moon as possible. However in a detailed look at the Moon's surface we can see an incredible number of very small craters that we are not able to create because of the low resolution of the mesh. Because of this reason the crater model still a little bit suffers from the lack of high resolution.

Figure 5.10 shows an example of a Moon-like looking planet with craters and a detailed look at the surface.

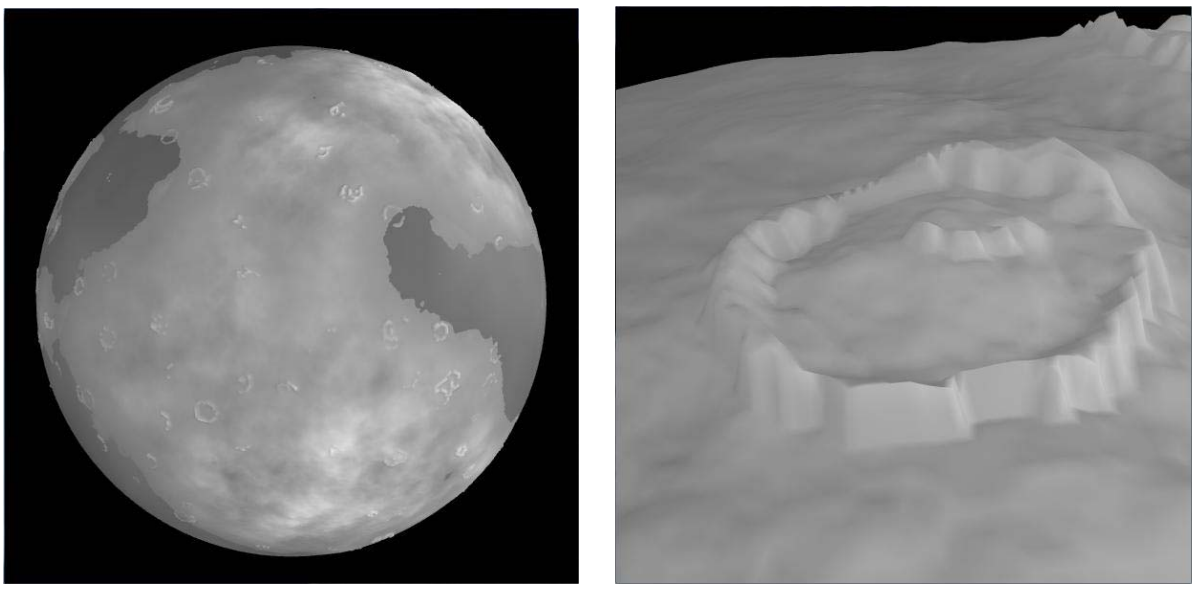


Figure 5.10: Moon-like planet. Overall view and a detail view of the surface.

This planet was created with a multifractal Perlin Noise and we added 1000 craters to its surface. Many of those craters had too small diameter and therefore they are not actually visible on the surface or just a small part of their rim is visible. In the detailed view of the Moon's surface we can see the crater in a higher detail. It is clearly visible how the size and also the vertical displacement of the rim are perturbed with Perlin Noise function to create more natural and random look.

5.3 Comparison of Static and Dynamic Level of Detail

In the last section of this chapter we will present a comparison of the static and dynamic level of detail. Although we can see from many images in this whole chapter that the dynamic level of detail enables us to zoom in and explore the surface of the planet in a higher detail, only thorough comparison of both methods will reveal the true performance boost that is provided by the Dynamic Level of Detail algorithm.

We will do the comparison from two points of view. First we will look at the visual differences in the terrain that we are able to display on the same computer with static and dynamic level of detail. The other point of view is a quantitative comparison. In this part of this section we will look at the size of the data structure and the number of triangles in the planetary model maintained by the static and dynamic level of detail.

5.3.1 Visual Comparison

In this section we will compare the two levels of detail according to what they are able to display. First we will compare terrain patches with the same number of triangles in the mesh structure but maintained by different level of detail algorithm. Then we will show more detailed example of how the Dynamic Level of Detail algorithm actually generates the additional detail and increases the resolution of the mesh.

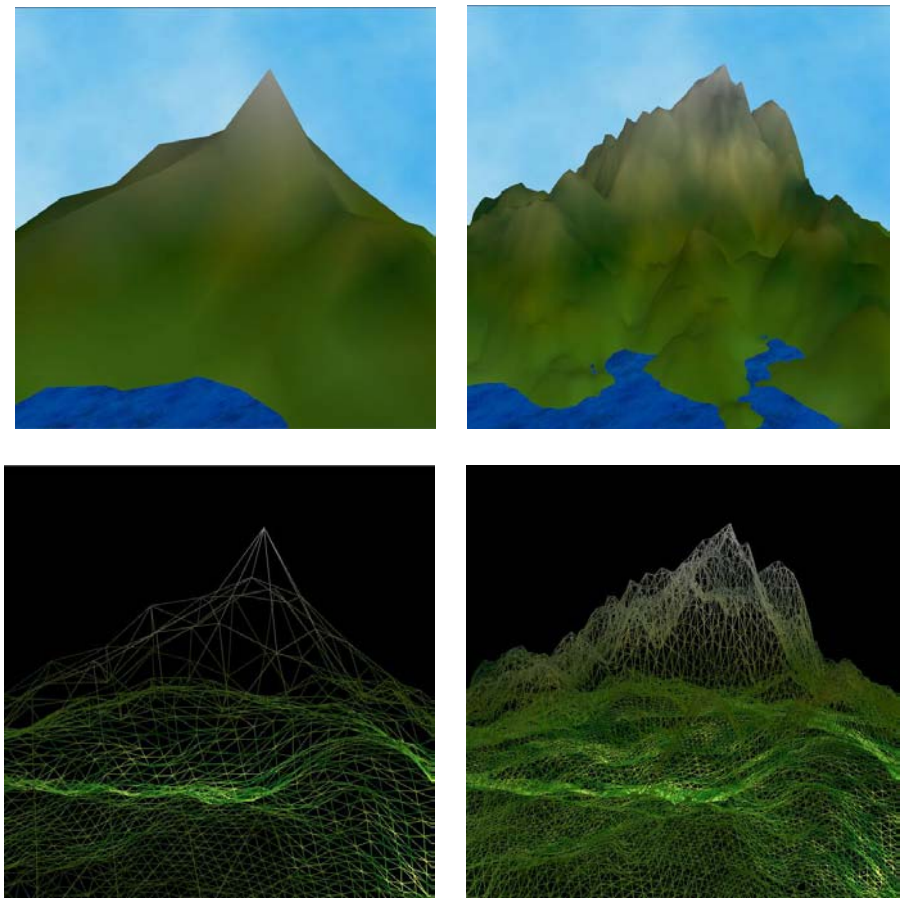


Figure 5.11: Comparison of static (left) and dynamic (right) level of detail.

Static versus Dynamic Level of Detail

Obviously the detail of the terrain that we are able to display on the screen depends on the performance of our computer. For that reason we will compare terrain models with approximately the same number of triangles in the mesh for both level of detail algorithms. In Figure 5.11 are identical terrain patches produced with multifractal Perlin Noise. Both planet models contain about 35000 triangles in the mesh. For a better visualization of the difference of the resolution of the mesh the figure also shows the wire frame model of the terrain.

Zooming in with Dynamic Level of Detail

Most of the images in this whole thesis are either showing an overall view of the whole planet or a high detailed view of some small part of the planet's surface. This cannot provide the right idea of how the Dynamic Level of Detail algorithm continuously increase the resolution of the planet mesh based on our distance from it. In this section we will show an illustration of this. In Figure 5.12 is a sequence of images generated during zooming in closer to the planet's surface. In each image the planet is generated with the same Perlin Noise function, only additional point and triangles are added into the model by the Dynamic Level of Detail algorithm.

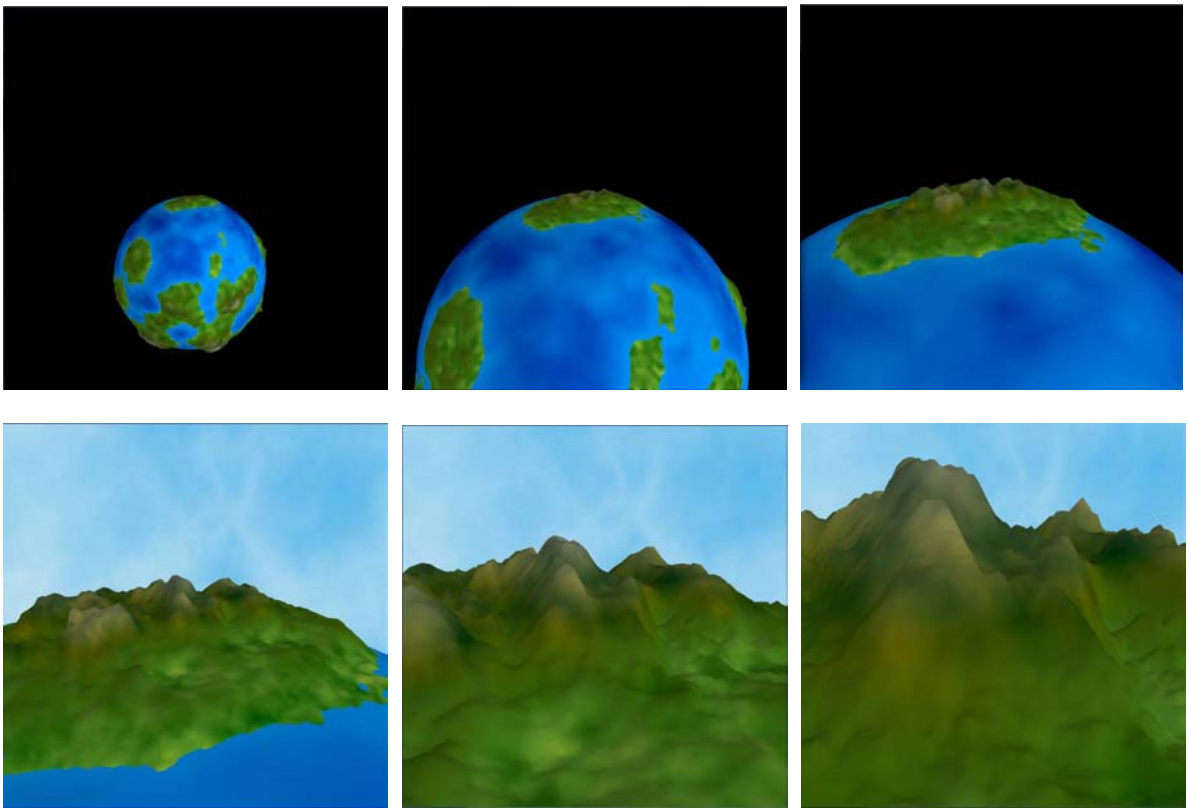


Figure 5.12: Illustration of increasing of the level of detail of the mesh structure by the Dynamic Level of Detail as we zoom in closer to the surface.

5.3.2 Quantitative comparison

In this section we will compare the number of triangles in the mesh maintained by static and dynamic level of detail. The static level of detail algorithm maintains the same resolution all over the planet and this resolution stays the same no matter how far from the surface we are. So in order to be able to explore the terrain with a high detail after we zoom in, we have to generate a mesh with high resolution all over the surface and thus very high number of triangles in the structure. Sooner or later the computational power of the computer will not manage to render such a big model fast enough and the rendering will not be smooth any more. Fortunately this is not true with the Dynamic Level of

Detail. This algorithm increases the resolution of the mesh only when necessary and therefore keeping the mesh structure small. The measurements and data in the following sections show exactly the differences between the two level of detail algorithms.

Dependency of the number of triangles on the level of detail

In our implementation of the Dynamic Level of Detail algorithm we first create the planet with a certain static level of detail and then from this level of detail we start applying the dynamic level of detail. The following Table 13 shows the number of triangles in the mesh with a certain level of detail. By certain level of detail we mean level of detail of the scene that is rendered on the screen. We can see that we start applying the dynamic level of detail from level of detail 5. The measurements were done when zooming towards the center of the planet. That means that we were moving in a perpendicular direction towards the surface of the planet.

Level of Detail	0	1	2	3	4	5	6	7	8	9	10	11
Static LoD	8	32	128	512	2048	8192	32768	131072	524288	2097152	8388608	33554432
Dynamic LoD	8	32	128	512	2048	8192	15722	36092	34688	30068	29594	28646

Table 13: The number of triangles in the mesh structure with a certain level of detail.

The table clearly shows how the number of triangles incredibly increases for the static level of detail algorithm. It is also quite interesting that the Dynamic Level of Detail algorithm manages to maintain more or less the same number of triangles in the structure even if we further increase the level of detail. To visualize these results in a better way, we plot them in Figure 5.13. We will plot the logarithm of the number of triangles in the mesh as a function of the level of detail. This is necessary in order to be able to have the values for both level of detail algorithms in the same graph.

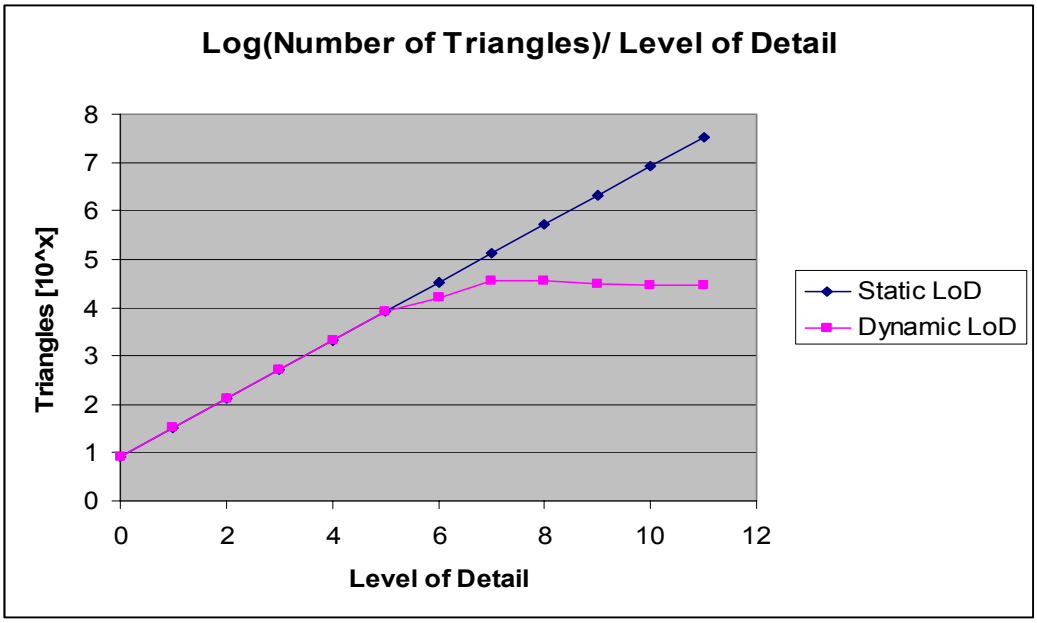


Figure 5.13: The dependency of the logarithm of number of triangles in the mesh on the level of detail.

Dependency of the number of triangles on the distance

We can obtain more interesting results if we closely measure the dependency of the number of triangles in the mesh structure maintained by the Dynamic Level of Detail algorithm when we are approaching the planet. The results are shown in Figure 5.14.

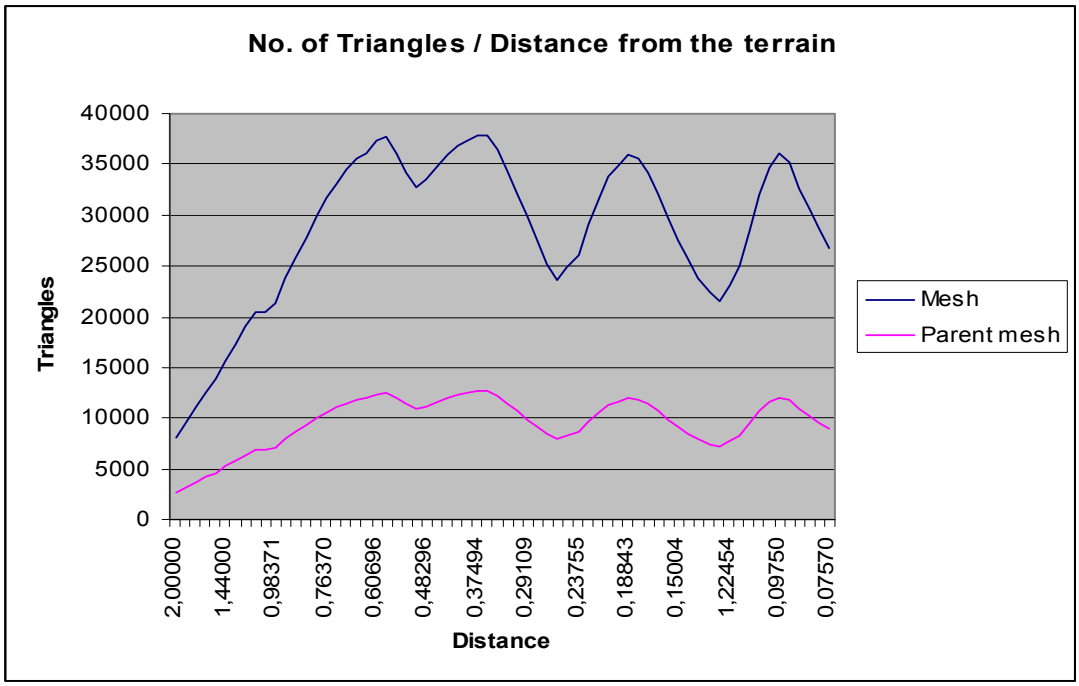


Figure 5.14: The dependency of the number of triangles on our distance from the surface for the Dynamic Level of Detail algorithm.

In this graph we can see how the number of triangles in the mesh changes as we are getting closer to the surface. For a complex idea about the size of the whole data structure the graph displays the number of triangles in both the mesh that is rendered on the screen and in the parent mesh that holds the parent triangles. There is couple of interesting things visible in this graph.

First of all we can see that the trend of the dependency becomes from a certain distance quite periodic. This periodic rising and regression of the curve is caused by the two main principles of the Dynamic Level of Detail algorithm. When the curve is at its lowest point, it means that the closest triangles in the mesh just become closer than the actual threshold for their refinement. When we keep zooming closer to the surface more and more triangles reaches this threshold and more and more triangles increase the resolution and add more new triangles into the structure. This appears in the graph as the rising part of the curve. At a certain distance from the surface all triangles have increased their resolution but no new ones yet reached the next threshold for the refinement of the resolution. If we keep zooming closer to the surface still smaller and smaller part of the surface is rendered in our view. In order to keep the mesh structure as small as possible the dynamic level of detail reduces the resolution of the triangles that move out from our view back to the low resolution. This is applied to all triangles at the edges of our view that are no longer visible on the screen. No new ones are added because we are still too far from the surface. This is happening till the closest triangles reach the next threshold for increasing their resolution and it appears in the graph as the regressing part of the curve.

Another interesting observation is that the size of parents list is not growing as we keep zooming closer. An obvious assumption would be that based on the hierarchical architecture of the data structure as we are increasing the detail of the mesh more levels of parents are added to the list, because we have to store all the ancestors of the currently displayed triangles all the way to the starting level of detail. But again the size of the parents list is also periodic because of the same principles that we have explained in the previous paragraph. As the triangles are removed from our view, they are reduced to the starting low resolution. This also removes all their parents from the parents list and thus keeping it small.

6 Conclusion and Future Work

In this conclusion chapter we will conclude and summarize what we have accomplished in this bachelor work and whether it fulfilled our goals that we have stated at the beginning of this thesis. Further we will mention several possible extensions and improvements to the application. In the last part we will describe several options for future development.

6.1 Conclusion

We can say that we have successfully accomplished all goals that we have stated at the beginning of this work. Our main contributions could be divided into the following points:

1. We have done a survey and found several algorithms for terrain generation and terrain coloring that are suitable for creating a planetary model.

2. We have analyzed, designed and implemented a hierarchical and dynamic data structure that is optimal for holding the geometry of the planet and that could be easily used as an input for terrain generation and terrain coloring algorithms.

3. An algorithm for dynamic level of detail was designed and implemented. The data structure holding the mesh was enhanced and optimized so that the dynamic level of detail works efficiently.

4. A testing application with graphical user interface was developed to enable us to generate a complex planetary model, to control the algorithms by adjusting the key parameters and to explore the planet model in a high detail.

5. We have carried out a comparison of terrain generation algorithms and algorithms for terrain coloring at different scales. Based on the results of this comparison we have described some problems and artifacts of some algorithms and we have reasoned about what algorithms create the best looking complex planetary model.

6. The last part of our work was a visual and quantitative comparison of static and dynamic level of detail algorithms. This comparison has clearly shown how big improvement is provided by the dynamic level of detail.

6.2 Extensions to the Current Application

Even though we said that we have accomplished all of our goals, there are still several problems that need to be solved and a lot of possible extensions to the current system. Here we will mention some of the most important ones

Most of the issues that still persist in our system are related to the more efficient implementation of the dynamic level of detail. An improved version of back face culling should be developed so that we don't waste computational time and memory space on triangles that are not visible on the screen. Also the whole data structure should be organized in a quad tree so that a faster and more efficient searching for points in our view could be implemented. The last set of problems is related to our current inability to calculate the normals and the coloring of the terrain in a real time while still maintaining smooth dynamic level of detail.

As we said several times before, this application was developed as a testing application mostly for the use by its programmers. In order for an external user to use it, the GUI should be improved and several important features must be added.

A quite big area of improvement and further research is the coloring of the planet. We could see on the figures in previous chapters that the level of detail of the geometry and the coloring applied to it still cannot provide sufficient amount of detail for the terrain to look realistic. In order to improve

this we have to texture the terrain. This would require implementation of texture synthesis module. This module would generate a unique texture for each terrain part based on its local and global properties. Applying this texture on the generated terrain would significantly improve the realistic impression of the rendered terrain.

6.3 Future Development

This application was developed for the purpose of studying fractal algorithms and it fulfils this requirement. However besides this the only output of the application is a nice looking image of a planet or a part of its terrain. In the future work I would like to re-implement the program as a plug-in for some 3D modeling software. The most probable candidate is modeling software called Maya. Having this application in the form of a plug-in would have several advantages.

First of all we could take advantage of the robust and fully equipped rendering engine of Maya. This would provide a huge improvement to the appearance of the produced planetary model. Secondly the output of the application would not be just a nice image but the geometry of the planetary model in the form of a Maya's object. This object could be stored, converted into another one for use by another software application or most importantly it could be added into a modeling scene and used for computer animation or other purposes.

In addition to this, this work could be used as a basic framework for implementing more complicated algorithms that are related to creating a complex planetary model. There are many ways how to further enhance the model. We can implement an algorithm simulating the tectonics of the continents. Also the erosion plays an important role in the forming of a real terrain and implementing an algorithm simulating this effect would significantly improve the model. We should add algorithms simulating the atmosphere of the planet as well. Another possible way of enhancement would be placing the planet in a real planetary system with other planets like moons around it and simulating the movement of the planets.

7 References

[1] D. S. Ebert, F. K. Musgrave, D. Peachey, K. Perlin, S. Worley. *Texturing and Modeling: A Procedural Approach*, Academic Press Inc, London, 1st edition, 1994.

[2] Heinz-Otto Peitgen, Dietmar Saupe. *The Science of Fractal Images*, Springer, 1st edition, 1988.

[3] Web page about the Perlin Noise algorithm, URL: http://freespace.virgin.net/hugo.elias/models/m_perlin.htm.

[4] Ken Perlin. An image synthesizer. In B. A. Barsky, editor, *Computer Graphics* (SIGGRAPH '85 *Proceedings*), volume 19, pages 287-296, July 1985.

[5] Harmut Jurgens, Heinz-Otto Peitgen, Dietmar Saupe *Chaos and Fractals*, Springer, 1st edition, 1993.

[6] Web page with Musgrave's images from the Texturing and Modeling book, URL: http://www.texturingandmodeling.com/ART/MUSGRAVE/CH14/Chapter14Art.html.

[7] Ken Perlin. Ken Perlin's home page. URL: http://mrl.nyu.edu/~perlin/.

[8] Web page about Perlin Noise function and its variations, URL: http://calyxa.pandromeda.com/tutorial/bigT/basis.html.

[9] Web page about visualization techniques, URL: http://www-viz.tamu.edu/.

[10] Andy Novocin. Web page with some images of fractals, URL: http://www.math.fsu.edu/~anovocin/pictures.html.

[11] Wikipedia. Web page about fractals, URL: http://en.wikipedia.org/wiki/Fractals.

[12] Anton Feenstra. Web page about Mandelbrot's set, URL: http://www.chem.vu.nl/~feenstra/mandel_gallery.html.

[13] Paul Bourke. Perlin Noise and Turbolence, URL: http://local.wasp.uwa.edu.au/~pbourke/texture_colour/perlin/.

[14] Russell Beattie. Web page with images of the Earth, URL: http://www.russellbeattie.com/notebook/1008356.html.

[15] The Open University. Web page about the Solar system, URL: http://rocksfromspace.open.ac.uk/The_Moon.htm.

[16] University Corporation for Atmospheric Research. Web Page about the Solar System, URL: http://www.windows.ucar.edu/tour/link=/cool_stuff/tour_mars_climate_1.html.

[17] J.C. Francois. Tutorial of the creation of a model of the Earth in 3D Studio Max containing images of the Earth, URL: http://www.noirextreme.com/earth.

[18] Rolf Hicker. Web page with images of the La Sal Mountains, URL:
http://www.hickerphoto.com/la-sal-mountains-8906-pictures.htm.

[19] Andrew True. Tutorial to the Dynamic Level of Detail Algorithm, URL:
http://www.gamedev.net/reference/articles/article2074.asp.

[20] Web page of the Computer Graphics course, URL:
http://service.felk.cvut.cz/courses/X36ZPG/.

[21] Virtual Terrain Project home page, URL:
http://www.vterrain.org/.

[22] Google Earth project home page, URL:
http://earth.google.com/.

[23] Earth3D project home page, URL:
http://www.earth3d.org/.

[24] Terragen project home page, URL:
http://www.planetside.co.uk/terragen/.

[25] TerraJ project home page, URL:
http://terraj.sourceforge.net/.

[26] Paddy Ryan. Web page with images of the Ayers Rock, URL:
http://www.ryanphotographic.com/Essay.htm.

[27] Web page with images of Canadian mountains, URL:
http://www.field.ca/activities/canoeing/.

[28] Frantisek Staud. Image gallery with photographs of Namibia desert, URL:
http://www.phototravels.net/namibia/namib-desert-aerial-vast.html.

[29] Wikipedia. Web page about the Moon, URL:
http://en.wikipedia.org/wiki/Moon.

[30] OpenGL home page, URL:
http://www.opengl.org/.

[31] Glut library source page, URL:
http://www.opengl.org/resources/libraries/glut/.

[32] Glui library source page, URL:
http://glui.sourceforge.net/.

[33] History of the Moon, URL:
http://starryskies.com/solar_system/Earth/lunar_history.html.

[34] Earth's Moon, URL:
http://nssdc.gsfc.nasa.gov/imgcat/html/object_page/a17_m_2444.html.

[35] Ulf Assarson, Tomas Möller. Optimized View Frustum Culling Algorithms for Bounding Boxes. In *Journal of Graphics Tools*, 5(1), pages 9-22, 2000.

[36] Mel Slater and Yiorgos Chrysanthou. View Volume Culling Using a Probabilistic Caching Scheme. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*, pages 71-78, 1997.

[37] Hanna Holst. *Avoiding cracks between terrain segments in a visual terrain database.* Department of Science and Technology, Linkopings University, 2004.

[38] David Hill. An efficient, hardware-accelerated, level-of-detail rendering technique for large terrains. University of Toronto, Department of Computer Science, 2002.

A User Manual

In this user manual we will go over the installation of the application and its technical requirements. Afterwards the description of all control blocks and their function will follow. As an extension to this User Manual the enclosed CD contains few tutorials that will show step by step how planets of various types could be created.

A.1 Hardware Requirements

The main bottleneck of the performance of this application is the size of the mesh structure. The bigger the mesh is and the bigger the resolution of the planet is the more memory space and computational time is required. However it is hard to specifically say what the minimum hardware requirements are because simply the better hardware configuration of the computer is, the better resolution and better quality of the final planetary model is. The hardware configuration determines the maximal level of detail of the planet, when the scene will can still be smoothly rendered. On Pentium M 1.7 GHz with 512 MB RAM the maximal smooth level of detail was 5. On Core 2 Duo 1.7 GHz with 2 GB RAM the maximal smooth level of detail is 7. Those values also strongly depend on the used graphical card.

A.2 Installation of the Application

The application was tested under the operational system Microsoft Windows XP and Microsoft Window Vista. On the enclosed CD is located the executable file. Any installation procedure is therefore unnecessary. However we have to make sure that all required libraries and textures are presented in the folder with the executable file as they are on the CD.

A.3 Operating Manual

In this section follow the detail descriptions of all control blocks in the GUI and available keyboard control of the application.

A.3.1 Opening Screen

When the application is started, the window will look as in Figure A.1. In the display window is a regular octahedron, which is the base for the future planet. All the parameters in the GUI are set to the default parameters.

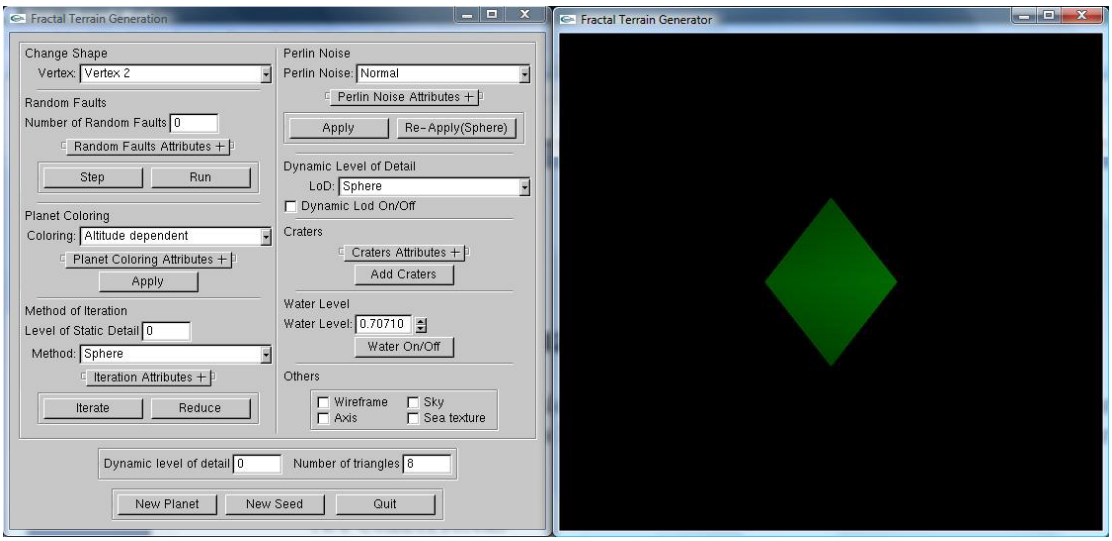


Figure A.1: The start of the application.

A.3.2 Change Shape

If we don't want to use the regular octahedron as the base shape for the planet, we can change its shape. First we have to choose which one of the octahedron's 6 vertex is supposed to be displaced. This is done chosen in the list box shown in Figure A.2. Afterwards we can use the mouse movement plus one of the keys "c", "b" or "v" to change the point's X, Y, Z coordinate.

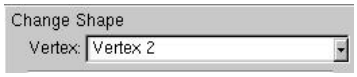


Figure A.2: Change Shape Listbox

A.3.3 Random Faults

The Random Faults control block can be used for setting parameters and starting and stopping the Random Faults algorithm. The control block could be seen in Figure A.3. The "Number of Random Faults" text field displays the number of iterations done so far. We can also set the amplitude of the Random Faults. By clicking on the button "Step" we will perform one iteration of the algorithm. If we click on the button "Run" the algorithm will start running in a loop performing a sequence of iterations. If we wish to stop it, then we click on the button "Run" again.

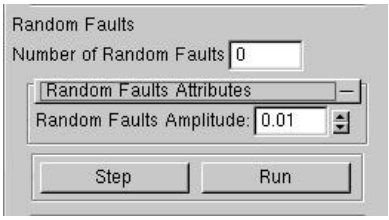


Figure A.3: Random Faults control block.

A.3.4 Planet Coloring

In this control block we can choose particular coloring scheme, set parameters of some of them and then apply our choice to the planet. The control block for planet coloring is shown in Figure A.4. In the "Coloring" list box we can choose one of the possible coloring schemes. We can set the lambda attribute for coloring schemes that use the Perlin Noise function. For schemes using the perturbation we can adjusts the number of octaves, the amplitude and the lambda of the used Perlin Noise function. This control block also includes checkbox that turns on and off the automatic color update while using the dynamic level of detail. Finally we can apply the chosen coloring scheme by clicking at the button "Apply".

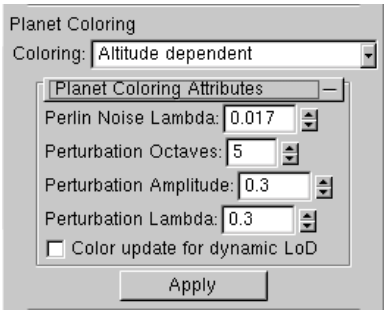


Figure A.4: The Planet Coloring control block.

A.3.5 Method of Iteration

This block deals with the selection of the algorithm that is used for displacing the new points when we increase the static detail of the planet. We can select on of possible algorithms in the "Method" listbox. We can also set the amplitude of the displacement done by the Mid-Point Displacement algorithm. This control block also displays the actual static level of detail. We can apply the chosen algorithm to the planet by clicking at the button "Iterate". Clicking at the button "Reduce" will decrease the resolution of the planet by 1 level.

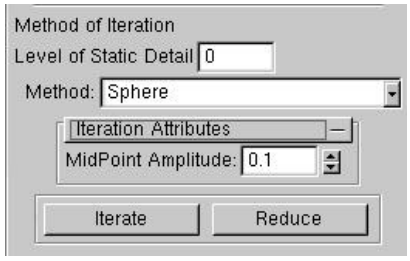


Figure A.5: The Method of Iteration control block.

A.3.6 Perlin Noise

This control block is in charge of applying various Perlin Noise based algorithms to the planet. We can choose particular Perlin Noise algorithm that we wish to apply in the "Perlin Noise" listbox. Consequently we can set the amplitude, the lambda and the number of octaves of the Perlin Noise function. We can apply the selected algorithm to the planet that we have previously created by clicking at the button "Apply". Another possibility is to apply the algorithm to the regular sphere. This could be done by clicking at the button "Re-Apply(Sphere)" and it enables us to apply the algorithms always to the same object and thus compare them with one another.

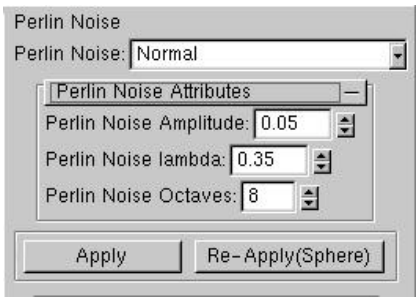


Figure A.6: The Perlin Noise control block.

A.3.7 Dynamic Level of Detail

This control block enables us to select the terrain generation algorithm that will be applied to the planet when we change the view and the Dynamic Level of Detail algorithm generates new points. It also includes the check box that turns the dynamic level of detail on and off.

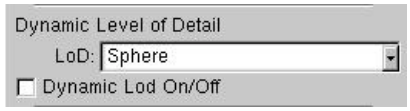


Figure A.7: The Dynamic Level of Detail control block.

A.3.8 Craters

In this control block we can control the generation of craters on the surface of the planet. We can set how many craters are supposed to be added during one step. Furthermore we can control the shape of the craters by adjusting the ration between the diameter and the elevation of the rim of the crater and by setting the maximal radius of the craters. The craters can be added to the planet's surface by clicking at the button "Add Craters".

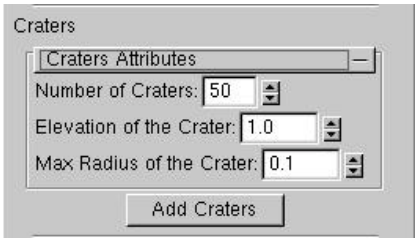


Figure A.8: The Craters control block.

A.3.9 Water Level

This control block controls the level of water on the planet. We can set the level of the water by adjusting the diameter of the water sphere. The water layer can be turned on and off by clicking at the button "Water On/Off".

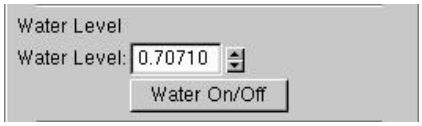


Figure A.9: The Water Level control block.

A.3.10 Others

This control block contains four checkboxes that can turn on and off some additional effects in the scene. "Wireframe" checkbox will display the planet model in the wireframe. By checking the "Axis" checkbox we can display the Earth axis going through the poles. "Sky" checkbox will turn on the sky texture. Similarly the "Sea texture" will control the displaying of the sea texture.

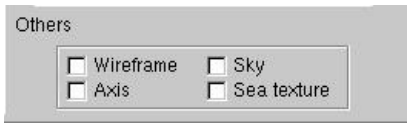


Figure A.10: Control block of additional effects.

A.3.11 Additional Control

This last control block located at the bottom of the GUI displays the actual dynamic level of detail of the planet and the number of triangles in the structure. By clicking at the button "New Planet" we will reset the GUI and start from the original regular octahedron. The "New Seed" button will reinitialize the random number generator. Lastly the "Quit" button will end the application.

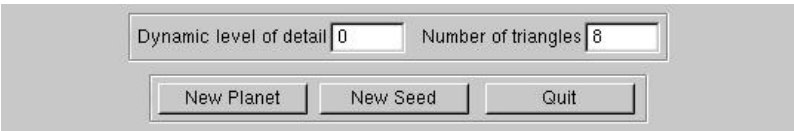


Figure A.11: Additional control.

A.3.12 Keyboard Control

In addition to the GUI control objects, several keyboard buttons also have some important function. The following table summarizes all buttons and their functions.

Keyboard	Action
Q + Mouse	Rotation of the planet
W + Mouse	Zoom In/Out
E	Shift the scene up
D	Shift the scene down
S	Shift the scene left
F	Shift the scene right
C + Mouse	Change the X-coordinate of the selected point
V + Mouse	Change the Y-coordinate of the selected point
B + Mouse	Change the Z-coordinate of the selected point
X	Apply the selected coloring to the planet
N	Calculate the normals of all triangles

Table A.1: List of keyboard controls and actions that they perform.

B Content of the Enclosed CD

Directory doc	Contains this thesis and the application tutorial in .doc and .pdf format.
Directory src	Contains the source codes and a project for Visual Studio 2003.
Directory Textures	Contains textures in jpg format.
glut32.dll	Glut library necessary for the application.
Frakt.exe	Executable version of the application.