Technical University of Munich

Department of Informatics

Bachelor's Thesis in Informatics: Games Engineering

Procedural generation of parameterizable earth-like planets with adaptive level of detail

Felix Brendel

Technical University of Munich

Department of Informatics

Bachelor's Thesis in Informatics: Games Engineering

Procedural generation of parameterizable earth-like planets with adaptive level of detail

Prozedurale Generierung von parametrisierbaren erdähnlichen Planeten in Verbindung mit adaptiven Detailgraden

Author:	Felix Brendel
Supervisor:	Prof. Dr. Gudrun Johanna Klinker
Advisor:	Sven Liedtke
Submission Date:	15 May 2018

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Garching, 15 May 2018 Felix Brendel

Abstract

Procedural content generation in games can be used to generate a more flexible and interesting environment for the player. This work explains the fundamentals of procedural generation while showing different kinds of generation methods. The **midpoint displacement** method and **noise functions** for generating terrain on a flat surface are shown and the problems of projection onto a sphere are explained. As a result methods of adapting the generation process to generate the terrain directly on the sphere are discussed and compared to each other. The hierarchical structure *index tree* is introduced which is used to allow for **adaptive level of detail**. The performance for different implementations are compared to each other and the visual results are shown. Additionally, for all relevant topics, sample implementations are provided and some programming techniques that proved helpful or more efficient are introduced.

Contents

1 Introduction

Procedural content generation is a technique of letting a computer program generate objects like virtual worlds, buildings, flora and fauna or even abstract concepts like writing systems. Often this program is using some kind of noise- or random functions to be able to generate a wide variety of content.

In this work, the fundamentals of procedural generation for games with the focus on generating planets are shown. The aim of this work is to give an overview of the fundamentals of procedural generation and showing the difference between generating terrain on a plane and on a sphere.

In contrast to the traditional approaches of mapping a two dimensional terrain onto the sphere, the main focus of this work lies on generating the geometry directly on the sphere to avoid projection imperfections and to produce a higher quality result. Additionally, the layout of the geometry in memory is an important subject of this work and the *index tree* will be introduced for this purpose and its operations and usages are discussed. Different approaches to generate spheres are shown and compared to each other. To generate terrain on them the two methods **midpoint displacement** and **3d noise** are used for generating the elevation information. The methods explored in the context of this work were all implemented to verify the functionality using C++ and the irrlicht 3d engine. Irrlicht was chosen because it is lightweight and it is easy to set low level data, for example switching the vertex buffer [1] between frames. Additionally to the explanations of the used algorithms many code examples and sample implementations are shown.

1.1 Definition of procedural generation

In general, procedural generation describes a program taking zero or more input parameters and generating an output. The input parameters and the output could be of any type, for example numbers, strings, images or 3d meshes. Procedural generation can happen **online** which means generating the assets while also using them at runtime or **offline** meaning the assets were procedurally generated beforehand and just used at runtime. [TYSB11]

1.2 Motivation of procedural generation

Procedural generation can be used to reduce the amount of work artists have to do while developing assets for video games. It can also enable artists to generalize assets

[1] more information regarding vertex buffers can be found in subsection 2.1.2

and let the program generate more assets following the artists description. For example an artist can design a part of a plant and describe the shape of plants with some rules so the program can generate plants that use the provided parts the artist designed and place and displace them in a way described by the rules. This kind of generation is called **rule based generation** and will be discussed together with other methods in section 1.4. The result of a procedural generation is defined by the parameters and therefore by randomizing the exact parameters a huge amount of assets can be generated, both reducing the work of artists while at the same time increasing the available assets in the game and thereby extend the quality of the game.

1.3 Random numbers

The motivation behind procedural generation is often times to be able to generate many different assets with the same program. Because of that many generation methods make use of random numbers to add more variation to the result. To generate seemingly random numbers the program makes use of a pseudo random number generator. This is an algorithm that produces seemingly random numbers. Since a computer is deterministic – it executes the given program – there is no way to produce completely random numbers without the use of an external source of noise, like the atmospheric noise or radiation. To still be able to generate different random numbers each time the algorithm is run the algorithm additionally takes a **seed** as an argument which is used for the initialization of the algorithm. With the same seed the algorithm will produce the same pseudo random values. A common technique is using the current time stamp as the seed so that the seed will never be the same.

Random number generators can have different distributions. With the **uniform distribution** the generator produces every possible number in the given range with equal possibility. This distribution should be used when there is no bias towards a specific number and can be used for example to simulate dice rolls or choosing an event based on a percentage as can be seen in Table 1.1.

Table 1.1: Lookup table to decide which random event occurred based on each event's probability and on the value of the random number chosen to determine the event. In this example the random number was chosen to be an integer in the interval [0, 99]

	probability	interval
event_1	10%	[0, 9]
event_2	20%	[10, 29]
event_3	25%	[30, 54]
event_4	45%	[55, 99]

In contrast to the uniform distribution, random numbers can also have a **normal distribution**. The user is able to specify the number μ which will be the number with the highest probability of being generated (expectation value) and the number σ which is the standard deviation. The probability density function for a normal distribution is

$$f(x)_{\mu,\sigma} = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

and can be seen in Figure 1.1 [Sil18].

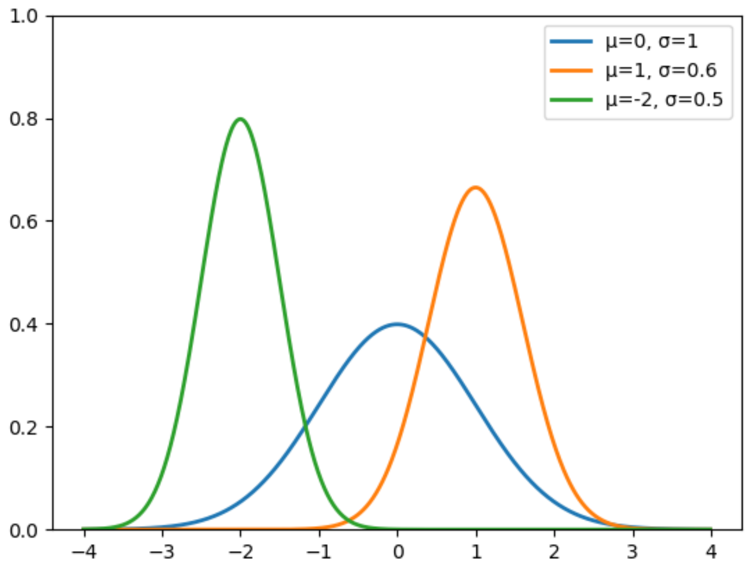


Figure 1.1: Exemplary probability density functions with different values for the expectation value μ and the standard deviation σ

A normal distribution should be chosen when a random variable should have the value around a certain value μ for example when the height information of a new vertex has to be calculated the resulting random hight should be around the height of the neighboring vertices.

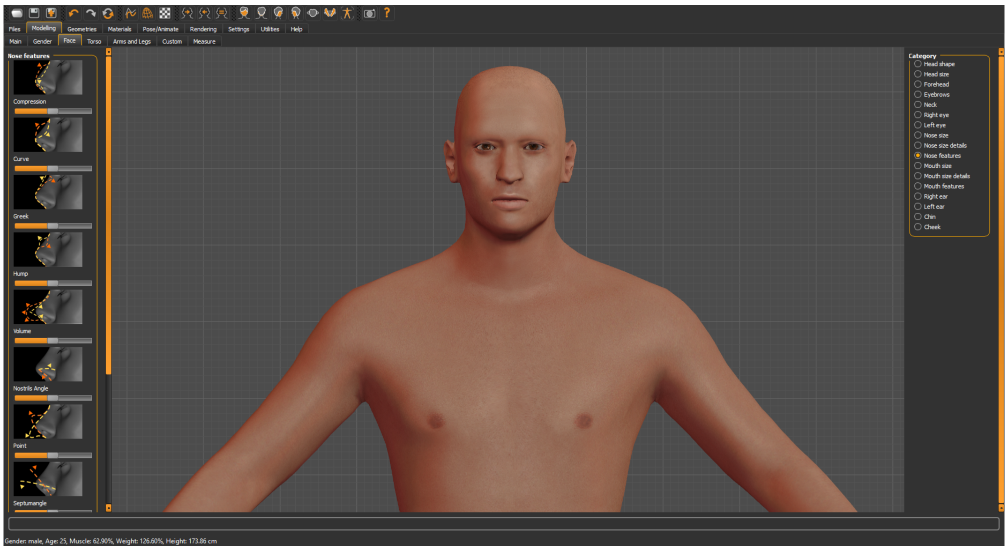
1.4 Types of generation

For the purpose of this work the field will be divided into four categories which fundamentally differ in the approach taken to generate. A procedural generation implementation is not limited to only one technique but can implement multiple types of generation for different tasks. For example the base terrain might be generated using noise, but hydraulic erosion or rivers are constructed using agents.

1.4.1 Feature based

The most intuitive way of generating assets procedurally is by generating feature based, meaning the parameters already give clear instructions on what the output should look like. The input parameters describe the output in a for the user predictable way. An example of this is "MakeHuman". It is a program designed to generate realistic looking 3d models of humans. The user has great control over the outcome and can set parameters like gender or age, height or even details like the nose tip position relative to the skull, as can be seen in Figure 1.2. [Mak]

In case of MakeHuman the parameters are limited to be in reasonable bounds so that the outcomes can look more realistic. The aspect that separates a feature based generation method from the other methods is that the parameters correspond to clearly expectable outcomes and there is no randomness involved.



Gender: male, Age: 25, Muscle: 62.90%, Weight: 126.60%, Height: 173.86 cm

Figure 1.2: In MakeHuman the user can provide a precise specification on what the resulting human mesh should look like

1.4.2 Grammar Based

Formal grammars have been introduced by Noam Chomsky and have a set of rules from which a given formal language can be created. Chomsky wanted to create a mechanism to formalize natural languages. As an exotic example, the rule set that generates the basic structure of (simplified) Korean sentences looks like this:

$$
\begin{aligned}
\langle \textit{korean sentence} \rangle &\rightarrow \langle \textit{base sentence} \rangle (\text{"."}|\text{"?"}|\text{"!"}) \\
\langle \textit{base sentence} \rangle &\rightarrow \langle \textit{subject} \rangle ?\ \langle \textit{object} \rangle ?\ \langle \textit{verb conjugation} \rangle \\
\langle \textit{base sentence} \rangle &\rightarrow \langle \textit{subject} \rangle ?\ \langle \textit{noun} \rangle \langle \text{이다 } \textit{conjugation} \rangle \\
\langle \textit{subject} \rangle &\rightarrow \langle \textit{noun} \rangle \text{"이"} ? \\
\langle \textit{object} \rangle &\rightarrow \langle \textit{noun} \rangle \text{"을"} ? \\
\langle \textit{verb conjugation} \rangle &\rightarrow \langle \textit{verb stem} \rangle \text{"아""요"} ? \\
\langle \text{이다 } \textit{conjugation} \rangle &\rightarrow \text{"이에요"}|\text{"예요"}|\text{"이야"}
\end{aligned}
$$

Every line corresponds to a rule in the grammar. When generating a "Korean sentence" a "base sentence" has to be created and then a symbol "." or "?" or "!" will be appended. The "|" symbol means selecting one from a set and the "?" expresses the possibility for the previous symbol to appear in the resulting string.

Even without any knowledge of Korean, given these rules and a set of words for the open variables "noun" and "verb stem" it is possible to generate sentences, taking a random applicable rule and apply it and substitute random words of the sets of open variables. This technique can be used to generate names phrases or sentences.

Grammars can also be used to generate other constructs, other than languages. Aristid Lindenmayer invented **Lindenmayer-systems** as a formal method to describe the fractal shapes of plants. An example of this can be seen in Figure 1.3 It also has an underlying rule set that gets applied to the axiom – the start state – and form there the rules will be applied from the last state. Lindenmayer-systems can also assign a probability to a rule, meaning if there are multiple rules that have equal left hand sides, it is decided stochastically which one to apply. [Smi84]

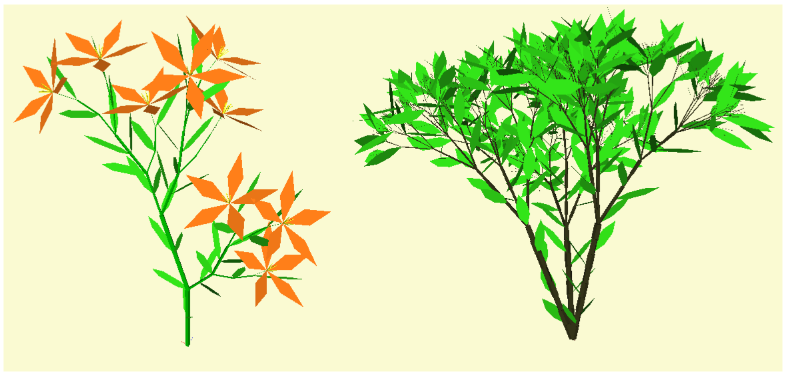


Figure 1.3: Plants generated with Lindenmayer systems. [Apa]

1.4.3 Agent based

A procedural generating agent is an entity that is influenced by and has influence on its environment and acts autonomous. When generating terrain with erosion by rivers, typically the sources of the rivers are spawned on mountains so that the river can find a path and flow down the mountains into the valleys. In that sense the rivers are generating agents since they are influenced by the environment – they flow in direction of the steepest slope down the hill – and they themselves influence the environment by applying erosion on the terrain.

Cellular automata are related to procedural generating agents in that they act depending on their environment, they are can be defined as:

> "A cellular automaton is a discrete model often studied in mathematics in the context of computability theory. It consists of a regular grid of cells, each in one of a finite number of states. Time is also discrete, and the state of a cell at time $t + 1$ is a function of the states of the cells in its neighborhood at time t." [Coo09]

In Figure 1.4 a seashell can be seen who's shell has similarities to the result of the cellular automaton "rule 30". These similarities are not random but rather the cells in the automaton have a similar behavior as the color in the shell. Like this, cellular automata can be used to model real world phenomena.

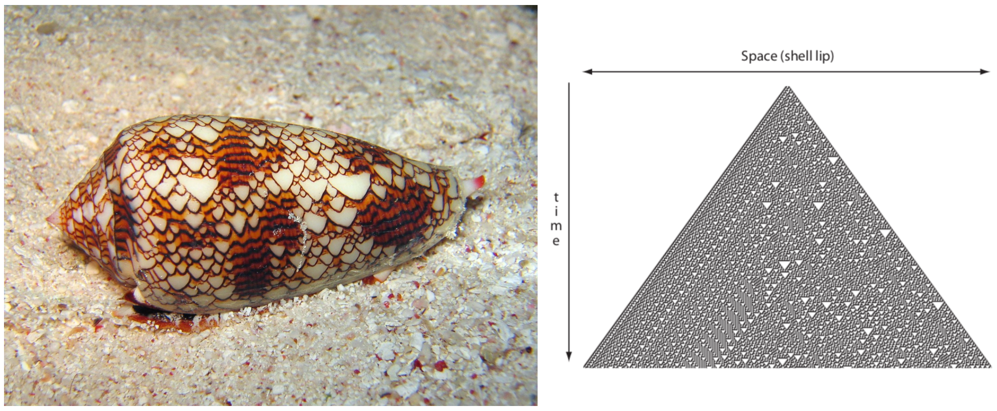


Figure 1.4: left: a seashell that exhibits a pattern that resembles a texture generated by a "rule 30" cellular automaton, right: the rule 30 automaton. [Coo09]

1.4.4 Noise based

Noise is like a n-dimensional map where at each position there is a random value. Two-dimensional noise is often used in terrain generation because the noise map only has to be calculated once and can be used for the whole landscape. If the noise values correspond to the elevation the noise map is called **height map**. An application of this can be seen in Figure 1.5. Different kinds of noise can be seen in Figure 1.7.

White noise When generating a noise map with white noise, every position in the map has a random number chosen independently from their surroundings, making it a discontinuous function. The result can be seen on the top left in Figure 1.7.

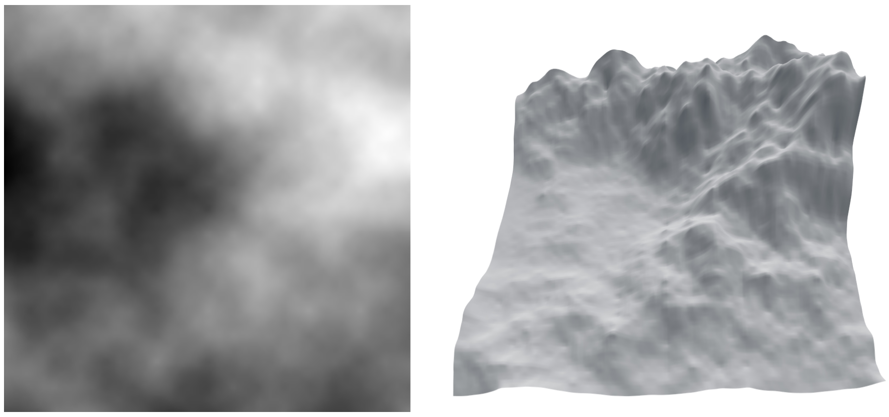


Figure 1.5: left: height map generated with the midpoint displacement algorithm, right: the resulting terrain

Cell noise To generate cell noise, first **seeds** have to be distributed over the map and the value of the cell noise map at a certain position equals the distance to the nearest seed. Also other metrics are used, for example the distance to the second nearest seed or de n-nearest seed in general. The resulting cells correspond to the voronoi diagram when using the seeds as vertices. Therefore the cells are guaranteed to be convex and can be used as caustics in water simulations or if each seed is colored differently the map can be used as influence for biomes or other characteristics in map generation.

Midpoint displacement Midpoint displacement is an iterative method of generating noise. It starts by setting random values at the corners of the line, square, or n-dimensional volume. Then the remaining points will be filled iteratively by finding the midpoints of regions generated by the last iteration and setting it relative to the value of its neighbors. In theory this can be done to a n-dimensional volume of any shape as long as there is a clear notion of determining midpoints. Since points close together should also have a noise value not far apart – to make the function continuous – the value for the new point is picked as the average of the neighboring points plus a normal distributed random value. To reduce the variance with each iteration the random value is taking to the n-th power in the n-th iteration. The **diamond square** algorithm is a two-dimensional variation of the midpoint displacement algorithm that produces a square noise field. Its steps are as follows:

1. Set the four corners to random values
2. **Diamond Step**: For every square generated in the last iterations, create a new vertex in the center of the square and set its noise value according to the rules, using the four corner vertices as reference points
3. **Square Step** For the same squares generate new vertices in the center of the edges of the square with the same rules, using the edges' vertices and the

nearest two vertices generated in the diamond step as reference points

4. goto step 2 as often as how many iterations are desired

Since the noise is pre computed at discrete points the noise map can not be directly used for values between two generated points, however adjacent points can be interpolated. A noise texture generated with the diamond square algorithm can be seen in Figure 1.7 on the top right.

Simplex noise Simplex noise is a more efficient version of **Perlin noise** and able to compute a n-dimensional continuous noise field. A property of simplex noise is that the noise details in the field have the same size. This can be used to generate fractal noise, meaning a noise field where details occur in deferent scales. Since real world landscapes often also have fractal properties, simplex noise is often used to generate height maps. To generate fractal noise, multiple scaled simplex noise fields are added together.

Warped noise When using the output of a function as the input to a noise function the resulting noise map is called "domain warped noise", depending on the function, different patterns can form. In subsection 2.2.3 a noise function will be used as input to another noise function so that some parts of the terrain can have a higher roughness than others. Examples of shapes formed by warped noise can be seen in Figure 1.6.

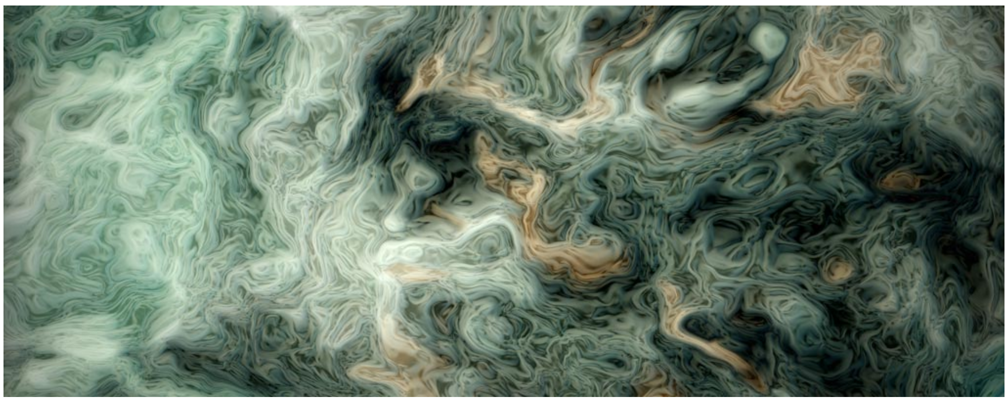


Figure 1.6: Domain warped noise sometimes creates patterns where nearby visible lines seem to "flow" in the same direction, like particles in the air with disturbances and thus looks like smoke. [Qui]

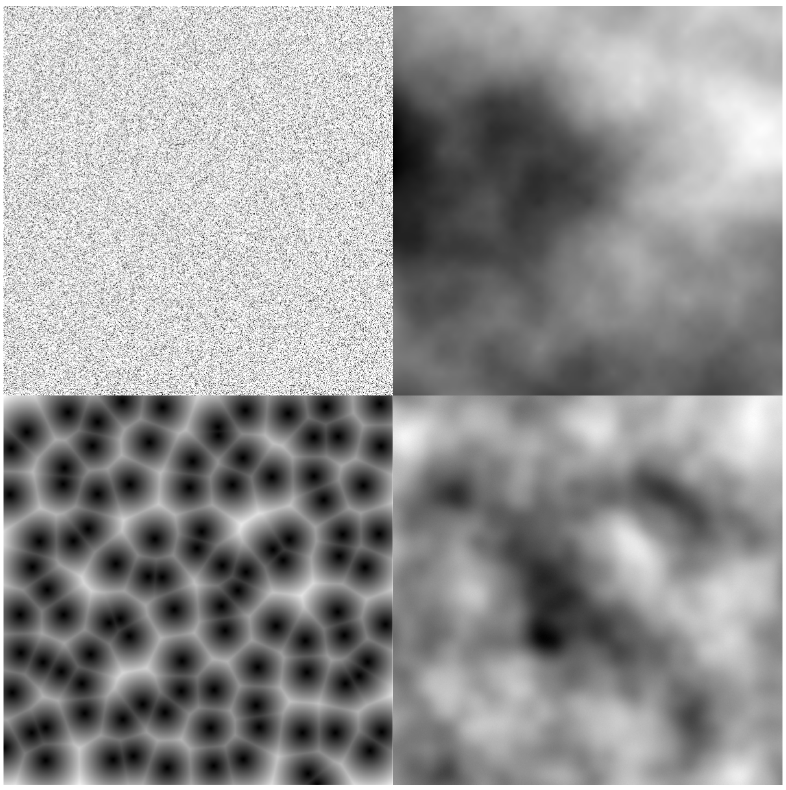


Figure 1.7: Comparison of different noise maps. Top left: white noise, top right: midpoint displacement, bottom left: cell noise, bottom right: simplex noise

1.5 Level of Detail

When dealing with highly detailed meshes there is a method to reduce the complexity for the computer to render it but at the same time keep the level of detail for the player. Because the more vertices the computer has to render, the more time is has to spend on each frame rendering the exact result. To keep the game interactive it should have a frame rate of at least around 30 frames per second since the human eye can clearly see the video stuttering if the frame rate drops below that.

The level of detail or **LOD** can be reduced for objects that are far away or in areas of the mesh that the camera does not see, for example because it is hidden behind another object, or because it is behind the camera. Geometry that is not seen usually does not have a big impact on the scene [2]. Traditionally games often have multiple versions of the high detailed mesh that differ in vertex count and depending on the distance to the camera, the appropriate mesh will be rendered so that a difference in detail can not be seen or appears minimal. An example of this can be seen in Figure 1.8 where there are different meshes representing different detail levels. The leftmost has the least amount of vertices and should be rendered when the object is far away and most detailed one should be rendered when the camera is really close to the object.

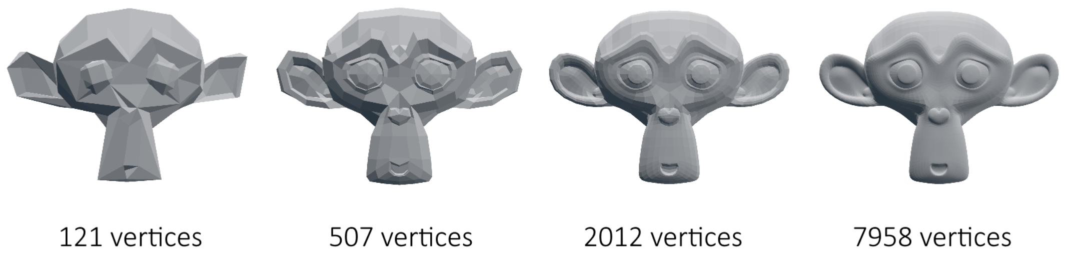


Figure 1.8: The same model in different level of details. The model can be exchanged depending on the camera's distance to the model, so that the total number of vertices that need to be rendered can be reduced without a visible loss of quality

1.5.1 Chunks

Another method to deal with massive amounts of geometry is to partition the geometry in parts called **chunks**. These chucks contain all the necessary information in that region and when again placed next to each other will look like one big continuous model. With chunks only the momentarily relevant parts are kept in memory and rendered. This method is used for example with the enormously large worlds in "MineCraft" where the world consists of chunks the size of 16x16x256 blocks (meters) [Per10]. Of course more

[2]Exceptions occur for example with reflections where an hidden object is reflected by a surface the camera can see, so the hidden object becomes visible

than one chunk has to be loaded at a time so that the horizon does not appear to be so near. An approach to keeping the necessary chunks in memory is to check which chunks are nearer to the player than the desired distance to the horizon. This is illustrated in Figure 1.9.

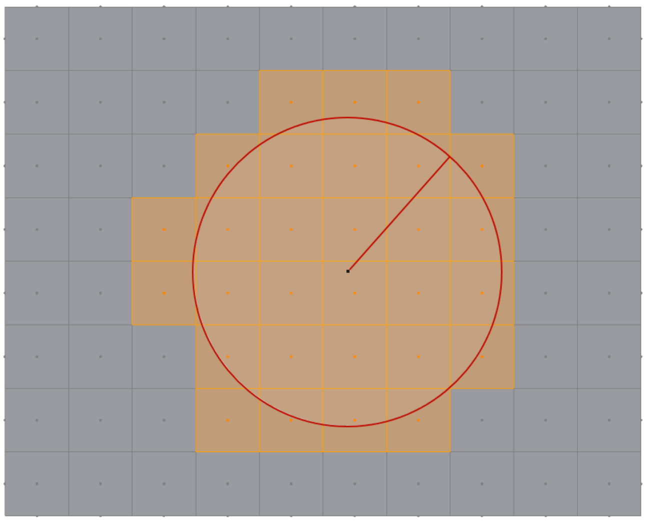


Figure 1.9: Simplified top down view on a world made of chunks with the player position marked black and the desired horizon marked red. The chunks that the game should hold in memory are the ones inside or touching the horizon circle

This technique makes it possible to have an nearly infinite game world while still having control over how many actual details to hold in memory.

1.5.2 Spatial hierarchies

A more advanced technique to deal with a huge amount of geometry is the usage of a spatial hierarchy. It is a hierarchical structure that divides space. In three dimensions, a well known example is the **octree**. An octree divides the space into 8 parts, for example the octants of the three dimensional Euclidean coordinate system. Spatial hierarchies can be used for reducing the search space for collision detection or also as a LODing method.

In subsection 2.1.2 the *index tree* is presented. The index tree is another kind of spatial hierarchy that solves the problem of an adaptive level of detail very well and is suitable for procedurally generating meshes.

1.6 Problems when generating on a sphere

For terrain that is projected onto a plane a planar height map can be generated and then used to displace the plane. This works since it is easy to project a flat texture onto a flat plane. Projecting a flat texture onto the surface of a sphere is more complicated and there are many different methods of projecting a planar surface onto a sphere. Two methods will be shown and their properties will be discussed.

1.6.1 Mercator projection

The mercator projection was first presented by Gerardus Mercator in 1596 and is one of the best known map projections and used in Google Maps [Ran06]. It is a cylindrical projection that preserves angles. The surface of the sphere is projected onto a cylinder and the resulting map, points with equal latitude or longitude lie on straight lines. But because the actual distance around the sphere decreases with higher latitude, the resulting map displays areas near the poles distorted as seen in Figure 1.10.

> "The classic comparison of areas is between Greenland and South America. Greenland appears larger, although it is only one-eighth the size of South America. Furthermore, the North and South Poles cannot be shown, since they are at infinite distance from other parallels on the projection, giving a student an impression they are inaccessible (which of course they seemed to explorers long after the time of Mercator)." [Sny78]

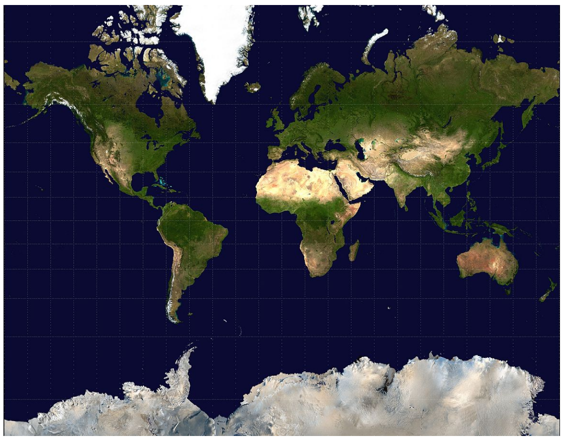


Figure 1.10: The poles can not be represented in a mercator projection and the areas near the poles are distorted

1.6.2 Peirce quincuncial projection

Peirce developed his quincuncial projection to "have a projection of the sphere which shall show the connection of all parts of the surface" [Pei79]. His work was published 1879. A benefit over the mercator projection is having a lower average exaggeration of scale, and therefore having a more equally distributed information density. The sphere is projected onto square that can be tiled. An image showing the peirce quincuncial projection of the earth can be seen in Figure 1.11. The equator is represented as a square and the other lines of constant latitude are complex curves.

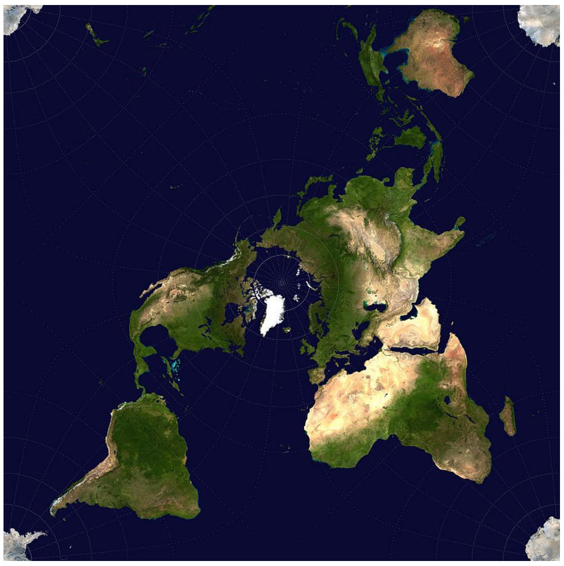


Figure 1.11: The peirce quincuncial projection maps the sphere onto a square in a way that the image can be tiled and so that the map shows the connection of all parts on the earth

1.6.3 Drawbacks of map projections

Every projection method onto a sphere has some kind of distortion and is therefore not optimal. The information density is not equally distributed along the surface and some areas will have more details while others have less. Mapping can be avoided by generating the height and color information directly on the sphere, without generating a flat map first. In section 2.2 two methods will be shown to generate elevation without generating a flat map and projecting it on the sphere.

1.7 Games and comparisons

As examples of procedural generation, in this section two games are presented that generate terrain procedurally and their approaches are analyzed. The source code of these games is not public and therefore it is difficult to get information on how the games implement the terrain generation internally. However, some developers talk about the development in blogs or interviews, which this section is based on.

1.7.1 MineCraft

In MineCraft players can explore and modify procedurally generated worlds, which are defined by the seed the player provides when the world is first set up, or chosen random if no seed is provided. Because of the nature of seeds in random number generation, two identical worlds will be generated given the same seed. The creator of MineCraft

Markus Persson described in his blog that MineCraft uses three dimensional perlin noise to generate the terrain. A screenshot of MineCraft can be seen in Figure 1.12. The terrain in MineCraft is made out ouf blocks and the noise function is checked at each 8th position on the planar axes and every 4th position on the vertical axes. The noise values are then linearly interpolated for every block position. If the noise function has a positive value after subtracting the block's height, a block is placed otherwise air is placed. [Per11] More elaborate techniques are still needed to generate different **Biomes** which are areas with certain vegetational features, like desert, taiga or tundra. As described in subsection 1.5.1, MineCraft uses chunks to store and load the relevant piece of terrain, since it is impossible to hold the whole terrain in memory.

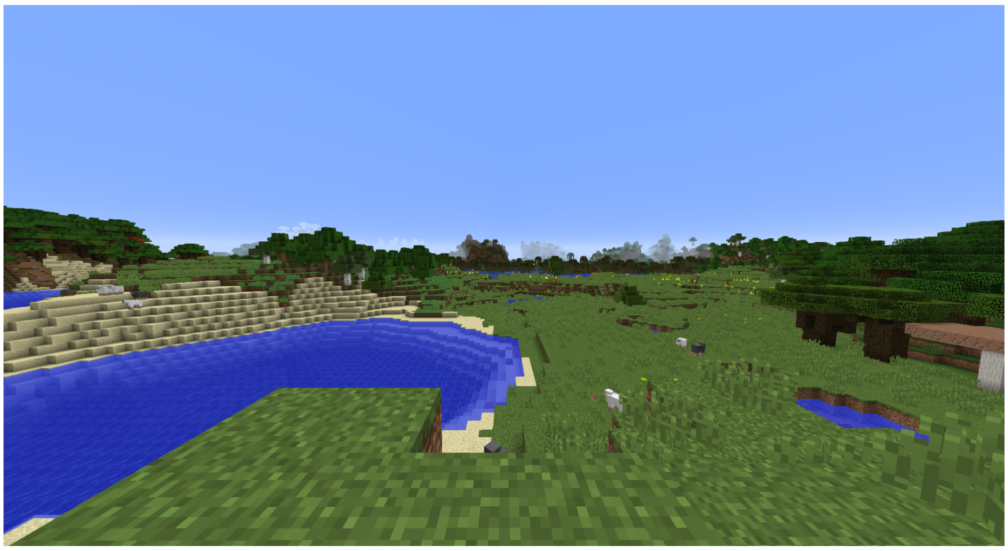


Figure 1.12: The world in MineCraft is made of blocks in a 3d grid. The noise function is evaluated at many grid cells and depending on the value a block or air is placed. The blocks between the evaluation points are linearly interpolated

1.7.2 Astroneer

In Astroneer the players explore spherical planets. The planets are generated using points that are generated by a noise function. Similar to MineCraft the noise function is evaluated at many points and based on the result a point is placed. These points then are combined to a mesh using the **marching cubes** algorithm. It takes a set of points as input and tries to connect them using edges and faces. Since the whole planet consists over a enormous amount of points, only the relevant ones are kept in memory. For that purpose an **octree** is used. Using an octree it is easier to determine which points are relevant for the players position and should be rendered. [Lie16]

Figure 1.13: The terrain in Astroneer is made out many points laying on a grid, connected using the marching cubes algorithm and spatially divided using an octree. [Ast]

2 Generating planets

The task of generating planets can be split up into two parts. The first part corresponds to generating a sphere. This is necessary as the terrain later will be generated on the sphere. So in the first step, methods of generating spherical meshes are shown and the hierarchical structure *index tree* is introduced that will be used to subdivide regions or delete unneeded geometry. In the second part, ways to generate the geometry are explored. The geometry will be generated directly on the sphere to avoid distortions caused by projections.

2.1 Generating a sphere

In this section, ways of programmatically generating sphere-like meshes are compared to each other. The methods should be able to create the geometry incrementally as only the relevant geometry should have to be generated. As the player moves around the planet, new geometry should be created and the parts that are no longer needed should be deleted. Building incrementally allows existing vertices to keep their positions and just add the new ones.

2.1.1 Base mesh

There are a number of different approaches of generating spheres. In the case of procedurally generating a planet it is required that the information density should be equal around the planet to avoid regions that are more detailed in comparison to others. All areas should have an equal level of detail. That means, that the standard deviation of the distances from the vertices to their neighbors should be minimal, or in other words, the vertices should be evenly distributed on the sphere. In this subsection different methods of generating polygonal meshes that approximate spheres are introduced and evaluated with regards to the information density. Additionally formulas to calculate the exact number of vertices and faces are presented, as it is sometimes necessary to know in advance how much memory should be allocated to store the geometry in. For that purpose also tables are shown which contain the number of vertices and faces up until the subdivision level 8.

Cube as base mesh

The probably simplest method of creating an approximation of a sphere is by taking a cube and subdividing it in a way to make it look spherical. To do this, all 6 faces of the cube are getting subdivided n times. Table 2.1 shows the number of vertices and faces for the subdivision levels and the resulting meshes can be seen in Figure 2.2. One Subdivision creates 4 new squares from every original face. After subdividing the sphere as often as desired, every point in the mesh will be scaled to have the desired sphere radius as distance from the origin of the mesh. The information density heavily depends on the position on the sphere, which can be seen by looking at the different edge lengths in Figure 2.1.

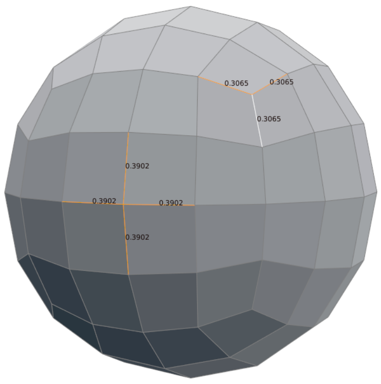


Figure 2.1: The information density of a subdivided and normalized cube varies across the surface. The edge length is displayed for the selected edges. Near the original corners the vertex densitiy is higher and in the areas of the original face centres, the vertex density is lower.

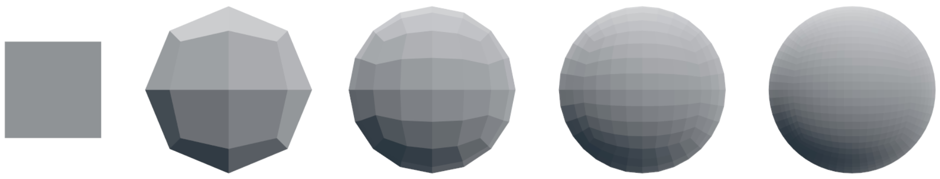


Figure 2.2: The first fice subdivision levels of a subdivided and normalized cube

The face count can be expressed as

$$\#Faces(s) = 6 \cdot 4^s$$

where s is the subdivision level. This is the case because the initial number of faces is 6 and for every subdivision every face will be divided into four new faces. The vertices follow the formular

$$\#Verts = 6 \cdot 4^s + 2$$

which is always two more than the face count.

Table 2.1: The vertex and face count for each subdivision level of the subdivided cube until level 8

subdivision level	vertex count	face count (quadrilaterals)
0	8	6
1	26	24
2	98	96
3	386	384
4	1538	1536
5	6146	6144
6	24578	24576
7	98306	98304
8	393218	393216

This follows Euler's spherical polyhedra formula

$$\#Faces + \#Verts - \#Edges = 2$$

where $\#Faces$ denotes the number of faces, $\#Verts$ the number of vertices and $\#Edges$ the number of edges in the mesh [Law97]. The mesh of the subdivided cube exclusively consists of quadrilaterals and every face has four edges around, but since every edge will be counted by the adjacent two faces the result has to be divided by two. So the formula for the number of edges is

$$\#Edges(s) = \frac{4 \cdot \#Faces(s)}{2} = 2 \cdot \#Faces(s)$$

Entering this in Euler's formula the result is:

$$\begin{aligned}\#Verts(s) &= 2 + \#Edges(s) - \#Faces(s)\\ &= 2 + 2(6 \cdot 4^s) - 6 \cdot 4^s\\ &= 2 + 6 \cdot 4^s\\ &= 2 + \#Faces(s)\end{aligned}$$

UV-sphere

The vertices of a UV-sphere lay on planes with equal latitude or longitude. The resulting sphere looks similar to models of the globe where the lines with equal latitude and longitude are marked. The mesh consists of quadrilaterals as every face except those connected to the pole vertices which form triangles. An image of a UV-sphere can be seen in Figure 2.3. A UV-Sphere is defined by its number of **rings** (horizontal face loops) and **segments** (vertical face loops).

Around the poles the vertex density is higher than at the equator because every horizontal layer has the same number of vertices but near the poles the distance between them is much smaller. To build the mesh incrementally the number of rings and segments have to be doubled or halved from one LOD-level to the next. This can be seen in Figure 2.4.

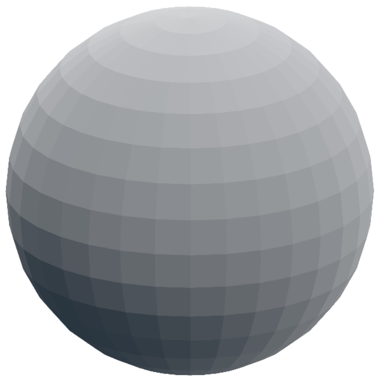


Figure 2.3: The vertices of a UV-sphere lie on circles with equal latitude or longitude

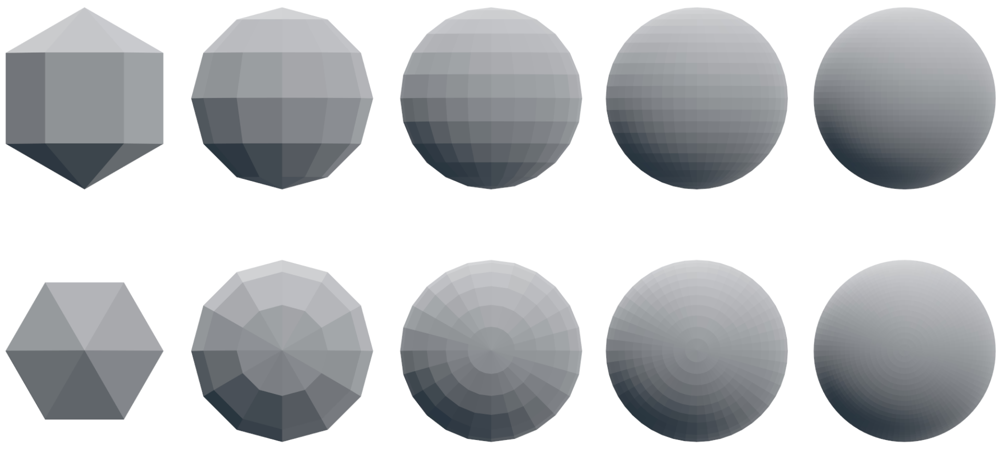


Figure 2.4: The first five subdivision levels of a UV-sphere starting with 3 rings and 6 segments and doubling both at each following subdivision

When a UV-sphere is getting subdivided the new rows are created by taking the average of all adjacent vertices that share the same longitude (subdivide the vertical edges) and scaling the points to the desired sphere radius. The new segments are generated by subdividing all edges between vertices that have the same latitude (subdivide the horizontal edges) but instead of scaling them to have the given radius they have to have the same latitude as the vertices they were created from. So they have to be moved out on the plane with equal latitude until their distance from the sphere's origin is equal to the desired radius, making the computation a little more elaborate.

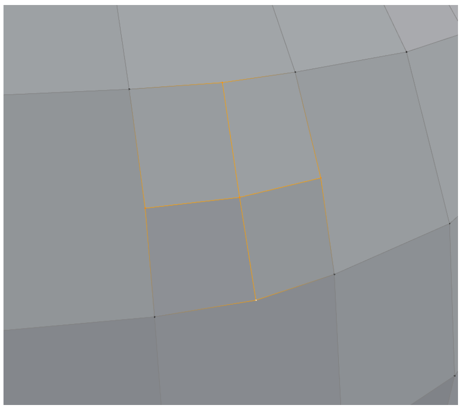


Figure 2.5: To subdivide individual faces on the UV-sphere, the faces are split vertically and horizontally and the new vertices have to be moved to sphere's surface according to the subdivision rules

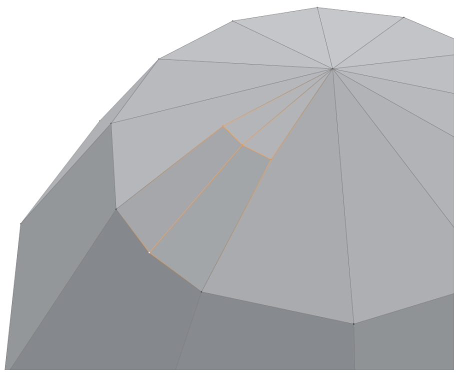


Figure 2.6: When the triangles at the poles are subdivided, two new quadrilaterals and two new triangles are created

The subdivision can also be applied locally, subdividing individual quadrilaterals (or triangles near the poles). When a quadrilaterals is subdivided, between one and five new vertices have to be created, depending on the subdivision level of the adjacent quadrilaterals, as shown in Figure 2.5. The rules for their displacement are the same as the global subdivision. If a triangle at the poles is being subdivided, two new quadrilaterals and two new triangles are created. This can be see in Figure 2.6.

The actual numbers of the faces and vertices when subdividing the whole sphere can be seen in Table 2.2. To calculate the exact numbers depending on the subdivision level the rows and segments are used. The number of rows and segments doubles every subdivision. Starting with 6 rows and 3 segments the formulas for the rows and segments is

$$\begin{aligned} \#Rows(s) &= 3 \cdot 2^s \\ \#Segments(s) &= 6 \cdot 2^s \end{aligned}$$

Based on these formulas the vertex count and the number od triangles and quadrilaterals can be calculated. The triangles occur only in the topmost and bottommost face loop. So for each row on the UV-sphere there are two triangles, one in the top face loop and one in the bottom face loop.

Table 2.2: The vertex and face count for each subdivision level of the UV-sphere until level 8.

subdivision level	vertex count	quadrilaterals	triangles
0	14	6	12
1	62	48	24
2	266	240	48
3	1106	1056	96
4	4514	4416	192
5	18242	18048	384
6	73346	72960	768
7	294146	293376	1536
8	1178114	1176576	3072

$$\begin{aligned}\#Tris(s) &= 2 \cdot \#Rows(s)\\ &= 6 \cdot 2^s\end{aligned}$$

The quadrilaterals make up the rest of the faces and can be calculated by multiplying two less than the number of rows with the number of segments. It is two less than the total number of rows because the top and bottom row are just triangles and not quadrilaterals.

$$\begin{aligned}\#Quads(s) &= (\#Rows(s) - 2) \cdot \#Segments(s)\\ &= (3 \cdot 2^s - 2) \cdot 6 \cdot 2^s\\ &= 18 \cdot 2^{2s} - 12 \cdot 2^s\end{aligned}$$

Finally, the number of vertices is equal to the number of vertical edge loops, which is one less than the number of face loops, multiplied by the number of segments plus the two vertices at the poles.

$$\begin{aligned}\#Verts(s) &= (\#Rows(s) - 1) \cdot \#Segments(s) + 2\\ &= (3 \cdot 2^s - 1) \cdot 6 \cdot 2^s \ + 2\end{aligned}$$

Icosahedron as base mesh

An Icosahedron a spherical mesh that consists solely of triangles. An icosahedron together with its subdivisions can be found in Figure 2.7. To subdivide the icosahedron, every face will be split into four new faces like shown in Figure 2.8.

The spherical meshes in the former sections do not have an even distribution of the vertices along the surface of the sphere. The vertex distribution of an icosahedron is much better but not perfectly even still. Subdividing an icosaeder is straight forward, create a new vertex at the center of every existing edge and scaling it until it has the desired radius as distance from the origin. Like this every former face will be divided in four new faces. Another benefit of icosahedra is that they exclusively consist of triangles, making it easier to render them. Triangles are preferred since they always lie on a plane making it easier to for example calculate face normals.

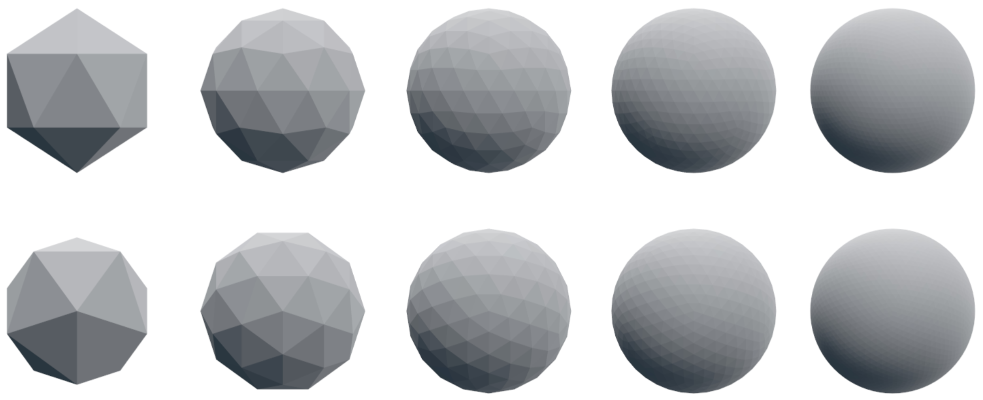


Figure 2.7: The first five subdivision levels of an icosahedron

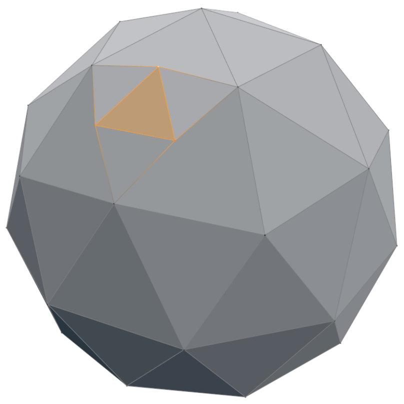


Figure 2.8: For each subdivision, every face will be split into four new faces

The unsubdivided icosahedron has 20 faces and for each subdivision level, each face will be divided into four new faces so the number of faces will quadruple every subdivision.

$$\#F(s) = 20 \cdot 4^s$$

The vertex count seems to follow the formula

$$\#V(s) = 2^{2s} \cdot 10 + 2$$

Table 2.3: The vertex and face count for the icosahedron for the subdivision levels until 8.

subdivision level	vertex count	face count (triangles)
0	12	20
1	42	80
2	162	320
3	642	1280
4	2562	5120
5	10242	20480
6	40962	81920
7	163842	327680
8	655362	1310720

The 12 vertices of an icosahedron with edge length 2 can be described as

$$\{\begin{pmatrix} 0 \\ \pm\varphi \\ \pm 1 \end{pmatrix}, \begin{pmatrix} \pm 1 \\ 0 \\ \pm\varphi \end{pmatrix}, \begin{pmatrix} \pm\varphi \\ \pm 1 \\ 0 \end{pmatrix}\}$$

where φ is the golden ratio. [Bae11]

$$\varphi = \frac{1}{2}1 + \sqrt{5}$$

To generate an icosahedron with radius 1, every coordinate will be normalized.

$$\{\begin{pmatrix} 0 \\ \pm Z \\ \pm X \end{pmatrix}, \begin{pmatrix} \pm X \\ 0 \\ \pm Z \end{pmatrix}, \begin{pmatrix} \pm Z \\ \pm X \\ 0 \end{pmatrix}\}$$

where

$$X \approx 0.525731112119133606$$
$$Z \approx 0.850650808352039932$$

Since the icosahedron has the most equally distributed information density, it will be used to implement the planets. When referring to the planet mesh it will always be based off an icosahedron. This allows further optimizations.

Avoiding generating duplicate geometry

When subdividing the faces of any base mesh, an edge in the face might already have been subdivided by the adjacent face. To avoid generating the same vertex again, the vertices can be stored in a map that maps the edge to the vertex that got created when subdividing the edge. When a face is being subdivided, for all edges it is checked if the edge is in the map and thus already has a center vertex. If it is not found in the map, a new vertex is created and pushed onto the vertex buffer, and its position is written in the map for the subdividing edge. Then the center vertex is used to build the inner structure of the new faces. An implementation of this can be seen in Listing 2.1. The two vertex indices of the edge are packed into a unsigned 64 bit variable and used as a key in the map.

Listing 2.1: To avoid generating duplicate vertices and store redundant information in the vertex buffer, a map is used which maps the edge to its center vertex when it is subdivided. When an adjacent face is getting subdivided, the center vertex of the edge does not have to be generated again if it is found in the map.

```
struct edge {
    u32 from, to; // vertex indices
};

struct edgeHash {
    u64 operator() (const edge e) const {
        u64 ret = e.from;
        ret <<= 32;
        ret |= e.to;
        return ret;
    }
};

struct mesh {
    std::unordered_map<edge, u32, edgeHash> edgeToCenterVertex;
    /* other members:
    - index buffer
    - vertex buffer
    ...
    */
};
```

To be able to use the edge – consisting of the indices of the two connected vertices – as the key in the map, the two unsigned 32 bit integers are packed into a single 64 bit value. Since the edge from vertex $v1$ to vertex $v2$ is semantically equivalent to the edge from $v2$ to $v1$ there is an ambiguity in representing the edge. This can be solved by always guaranteeing that $v1$ is always smaller than $v2$, or flipping them if needed when packing them into an 64 bit integer. That's why, when looking up an edge in the map, there is only one key possible that has to be checked.

2.1.2 Index tree

Traditionally when rendering a mesh, the user has to provide a **vertex buffer** and an **index buffer** holding the necessary information for the positions of the individual vertices in the mesh and their connections. The vertex buffer is a list of vertices, so a list of coordinates. The index buffer declares which vertices are forming triangles. It is a list of triples of the indices of the vertices in the vertex buffer. An example of a vertex buffer can be seen in Table 2.4, an index buffer is shown in Table 2.5 and the resulting mesh is show in Figure 2.9.

Table 2.4: Sample vertex buffer with two dimensional coordinates that defines the position of the vertices. The buffer can be extended to three dimensional coordinates by adding a third column.

index	x	y
0	0	1
1	-1	0
2	1	0
3	0	-1
4	3	0

Table 2.5: Sample index buffer describing the connections between the vertices from Table 2.4. Each row corresponds to a triangle.

index	v1	v2	v3
0	0	1	2
1	1	3	2
2	0	2	4
3	3	4	2

To be able to have more control over the mesh – to subdivide specific regions while leaving others the same while generating new vertices – is hard with only the vertex and index buffer. Suppose the planets mesh is an icosahedron at subdivision level 3. According to Table 2.3 it has 642 vertices and 1280642 faces. Suppose now further that the geometry on the backside of the planet is no longer needed and should be removed. The backside cannot be completely deleted since then it requires much work to restore the geometry later. The geometry has to be reduced in a controlled manner to restore the original geometry of the icosahedron. This is not possible with just the index and vertex buffer. Instead an **index tree** is used. An index tree consists of index tree nodes. Each node corresponds to a triangle and contains the three indices of the vertices in the vertex buffer. If the triangle is subdivided it also has pointers to its four children nodes. The implementation can be seen in Listing 2.2. An image showing an exposed view of a partially subdivided sphere using the index tree can be seen in Figure 2.10.

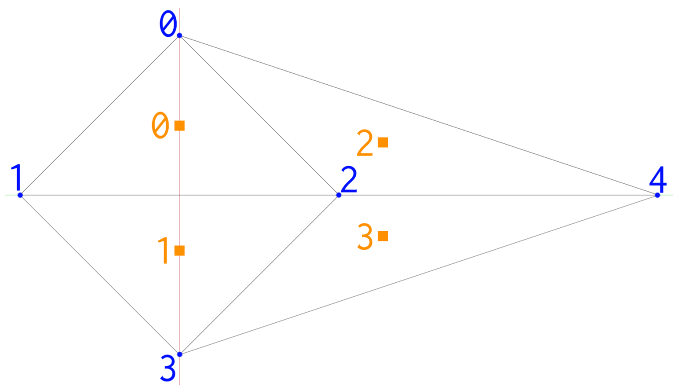


Figure 2.9: An example mesh created with the vertex buffer in Table 2.4 and the index buffer in Table 2.4. The gray lines correspond to the edges of the mesh, which are surrounding the faces (marked orange). The blue dots are the position of the vertices.

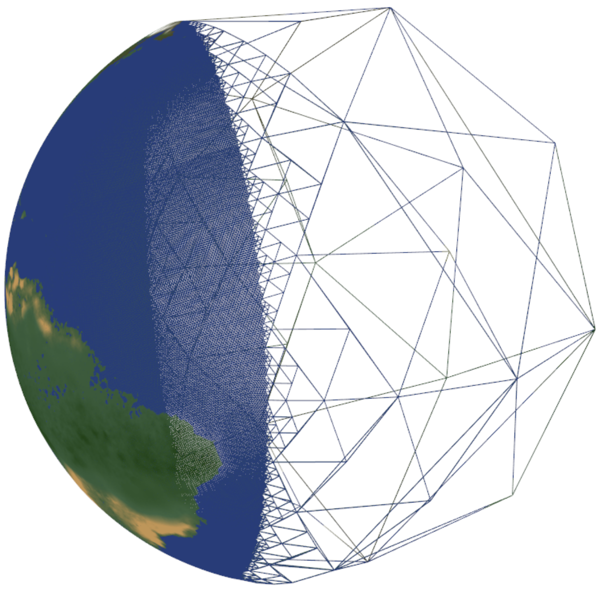


Figure 2.10: Exposed view from the side over the generated geometry. The geometry will be generated in a way so that the player will always face the side of the planet that has a high level of detail. On the backside of the planet, the geometry will not be generated.

Listing 2.2: Implementation of the index tree. Because an icosahedron has 20 faces, the root has 20 children. Every other node has either 4 or 0 children. However it is not known how many faces are currently represented in the tree. Because of that, the exact number is stored in the attribute `numberFaces` in the `indexTree` struct

```
struct indexTree {
    u32 numberFaces;
    indexTreeNode* children;

    ~indexTree() {
        if (children) {
            delete[] children;
        }
    }
};

struct indexTreeNode {
    u32 v1, v2, v3; // vertex indices
    indexTreeNode* children;

    ~indexTreeNode() {
        if (children) {
            delete[] children;
        }
    }
};
```

Subdividing a node

A fundamental benefit of using the index tree over the index buffer is being able to generate more geometry in a specific region in a controlled manner. This can be done recursively by subdividing the tree nodes until the tree's leafs – the most subdivided triangles – are reached and then generating new geometry on them. A sample implementation for subdividing the tree nodes can be found in Listing 2.3.

Listing 2.3: Recursive implementation to subdivide the whole sphere. When the leaf nodes are reached the new geometry will be generated.

```
void subdivTreeNode(mesh* m, indexTreeNode* n) {
    if (n->children) {
        subdivTreeNode(m, n->children+0);
        subdivTreeNode(m, n->children+1);
        subdivTreeNode(m, n->children+2);
        subdivTreeNode(m, n->children+3);
    } else {
        subdivTriangle(m, n);
    }
}
```

The function that generates the new geometry then checks if the edges already have been subdivided by the neighboring faces like described in section 2.1.1, by looking up the edges in the `edgeToCenterVertex` map . If edge was not found in the map, a new vertex will be created. Then the tree node creates the four new children of itself corresponding to the four new faces created from the triangle represented by the tree node.

Reducing geometry of a node

When a part of the geometry is not needed anymore it can be deleted from the index tree by simply deleting the children of a index tree node and freeing their memory. Since this only affects the index tree, the vertices will still be present in the vertex buffer and also be referenced in the `edgeToCenterVertex` map but not rendered any more. If new geometry is generated and never deleted this will turn into a problem and will be addressed in section 2.1.2. Sometimes it is better, if the vertices that are not used right now are not deleted immediately after they are removed from the index tree. When the player visits the area again after the faces were removed from the index tree – and the vertices have to be rendered again – they do not have to be generated again if they still are in the vertex buffer and in the `edgeToCenterVertex`. Like this they are cached and not in use right now, but it is trivial to render them again by rebuilding tree nodes that reference them.

Updating the geometry when the player moves around the planet

As the player moves or flies around the planet the geometry has to be updated all the time because only the relevant geometry is shown to the player to improve the performance. When the player has moved enough to trigger a geometry update, first the geometry that is too far away has to be deleted from the tree. Like this for every of the 20 nodes in the index tree it is checked if they contain faces that are too far away from the region of interest. It makes sense not to check **all** of the faces in the index tree, because there are possibly really many of them, where most of them also are not too far away because if the player just moves enough to trigger the geometry update, most of the faces will still be near enough. So what can be done instead is only checking the tree until a certain depth on each branch. The value 2 seems to be a good balance between deleting enough faces and not having to check many faces for their distance to the player. A value of 2 means that the children of the children of the triangles of the icosahedron are checked for deletion but not their children. A triangle is considered in range, if it has at least one vertex that is in range. A lower number means less checks and thus potentially more vertices that are not deleted even if they are too far away. A higher number deletes more faces that are too far away but the process could take exponentially longer since the branching factor in the tree is 4.

Deleting the faces in the tree does not delete the vertices themselves. In general it is not safe to delete the vertices of faces that are being deleted in the tree because these vertices might be used by faces that are still near enough to not be deleted. The easiest way to deal with the *dead* vertices is having a threshold of how many vertices are allowed in the vertex buffer, regardless if they are rendered or not, and once the vertex count is bigger than the threshold, do a full rebuild of the mesh and delete the old one. This is the easiest solution and at the same time the slowest, because rebuilding all the geometry is unnecessary overhead.

There is a way to find out which vertices in the vertex buffer are actually dead by keeping track of by how many faces the vertices are used. However this information is not really helpful because then the vertex buffer has to be defragmented, and all the indices will be wrong. So additionally every index in the index tree, index buffer and the `edgeToCenterVertex` has to be updated, which seems to be an expensive operation and was not explored further.

Another way of cleaning up the vertex buffer is by rebuilding the index tree, enumerating every vertex index and in parallel rebuild the vertex buffer. After that only the `edgeToCenterVertex` map has to be updated. The algorithm that cleans up the whole planet can be seen in Listing 2.4.

The key to this algorithm is the `oldToNewIndices` map. When, while traversing the tree, the indices of a tree node are found in the map, it will be assigned to the updated one. If indices are encountered that are not in the map yet, they will be assigned the next free numbers and added to the map. At the same time the vertices at these indices will be pushed into the new vertex buffer. After following this algorithm and visiting every node in the index tree, the vertex buffer will only contain the vertices referenced by the nodes in the tree. In section 4.3 the different approaches are compared.

Listing 2.4: The algorithm that is used to clean the planet from not needed vertex information.

```
void cleanup(planet* p) {
    auto n_vb_water = new CVertexBuffer(EVT_STANDARD);
    auto n_vb_land  = new CVertexBuffer(EVT_STANDARD);
    auto oldToNewIndices = new std::unordered_map<u32,u32>;

    for (u32 i = 0; i < 12; ++i) {
        n_vb_water->push_back((*p->water->vertexBuffer)[i]);
        n_vb_land ->push_back((*p->land ->vertexBuffer)[i]);
        (*oldToNewIndices)[i] = i;
    }

    auto n_it = cleanTree(p->indexTree, p->water->vertexBuffer,
                          p->land->vertexBuffer, n_vb_water, n_vb_land, oldToNewIndices);

    p->water->vertexBuffer = n_vb_water;
    p->land ->vertexBuffer = n_vb_land;
    p->indexTree = n_it;

    auto n_etc = new std::unordered_map<edge, u32, edgeHash>;

    for (const auto e : *p->edgeToCenterVertex) {
        // skip if vertex is not used any more
        if ((*oldToNewIndices)[e.second] == 0) continue;

        // the indices in a edge have to be sorted
        u32 from = (*oldToNewIndices)[e.first.from];
        u32 to   = (*oldToNewIndices)[e.first.to];

        if (from < to) {
            (*n_etc)[{from, to}] =
                (*oldToNewIndices)[e.second];
        } else {
            (*n_etc)[{to, from}] =
                (*oldToNewIndices)[e.second];
        }
    }

    p->edgeToCenterVertex = n_etc;
}
```

Generating an index buffer from the tree

The index tree can then be used to extract an index buffer, containing the most subdivided faces, which correspond to the leaf nodes of the tree. Since an icosahedron has 20 faces, its index tree has 20 children. For each of the tree nodes it is checked if it has further subdivision information, meaning if it has any children. If it has, the indices are not written to the buffer but the children have to be checked recursively. This can be seen in Listing 2.5. The function `getIndexBufferFromTree` gets the pointer to a `indexTree` as a parameter and returns a new index buffer.

Listing 2.5: Simple recursive implementation of the algorithm to generate an index buffer from the index tree

```
void  writeIndicesToBuffer(indexTreeNode* n, CIndexBuffer* buff) {
    if (n->children) {
        for (u32 i = 0; i < 4; ++i) {
            writeIndicesToBuffer(n->children+i, buff);
        }
    } else {
        buff->push_back(n->v1);
        buff->push_back(n->v2);
        buff->push_back(n->v3);
    }
}

CIndexBuffer* getIndexBufferFromTree(indexTree* t) {
    // three vertices per face
    auto buff = new CIndexBuffer(EIT_32BIT);

    // the indexTree of an icosahedron has always exactly 20 children
    for (u32 i = 0; i < 20; ++i) {
        writeIndicesToBuffer(t->children+i, buff);
    }
    return buff;
}
```

2.2 Generating the terrain

In the previous section different methods were discussed for generating a sphere and only generate and render the relevant parts using a tree structure to hold the geometry in a hierarchical structure. This section continues with generating elevation information and displacing the sphere to generate a planet base. Additionally a second not modified spherical mesh will be used as the water on the planet. Like with the different methods of generating a spherical mesh, two different methods of generating elevation data will be presented. And a technique for a variable roughness around the planet and assigning colors to the vertices are shown.

2.2.1 Noise based approach

As discussed in section 1.6 mapping a flat texture onto a sphere should be avoided if possible to have equal information density around the sphere. To avoid having to map a flat texture onto a spherical surface, a spherical noise map will be generated which can directly be mapped onto the sphere. Like explained in subsection 1.4.4 simplex noise can be n-dimensional. For this approach a tree dimensional simplex noise will be used since the coordinates of the surface of the sphere are also three dimensional.

So when generating the planet surface, the noise map is queried with the yet undisplaced vertex on surface of the sphere. It is important that the point used to query the map lies on e surface of the sphere and is not just the average of its neighbors to have a continuous noise map for the surface.

Typically when using simplex noise to generate a terrain multiple layers of noise at different wavelengths to generate a fractal landscape. An example of this can be seen in Listing 2.6. The function `get3dNoiseHeightAt` is used to find out the definite elevation of a vertex on the sphere. In it multiple results of calling the function `heightNoise` are being accumulated, which directly looks up a specific coordinate in the map. The `waveLen` parameter specifies the wavelength of the noise, so the amount the point is scaled by before it is looked up in the map. The different layers of noise can be seen as having different purposes. The first call to `heightNoise` is scaled down by less than the others because it will be used to generate the basic shape of the continents, therefore its wavelength is high (so the point is scaled less in the noise field). A smaller wavelength would result in more smaller continents. To add finer details to the planet more layers will be added on top. Every noise layer tries to keep the overall shape of the planet but add more details to it. That is why the noises are scaled down more, the finer details the add to the planet.

2.2.2 Midpoint displacement based approach

As explained in subsection 1.4.4 midpoint displacement works by taking the midpoint of an existing structure and displacing it. So when a triangle will be subdivided the three new vertices will lie in the center of the edges and their height depends on the vertices that the edges connects. A sample implementation can be seen in Listing 2.7.

Listing 2.6: Example implementation of a function to get the elevation of a point on the sphere using fractal noise

```
// generates values between -1 and 1 then scales down
f32 heightNoise(f32 x,f32 y,f32 z, f32 waveLen, f32 scaleDown) {
    f32 r;
    r = noiseHeight->GetSimplex(x * waveLen, y * waveLen, z * waveLen);
    return r * scaledown;
}

f32  get3dNoiseHeightAt(f32 x,f32 y,f32 z) {
    f32 n = 0;

    n += heightNoise(x, y, z, roughness * 200,   0.0009f);
    n += heightNoise(x, y, z, roughness * 400,   0.00045f);
    n += heightNoise(x, y, z, roughness * 2000,  0.0002f);
    n += heightNoise(x, y, z, roughness * 4000,  0.0001f);
    n += heightNoise(x, y, z, roughness * 8000,  0.00005f);
    n += heightNoise(x, y, z, roughness * 16000, 0.000025f);
    n += heightNoise(x, y, z, roughness * 32000, 0.0000125f);

    return n;
}
```

Listing 2.7: Basic algorithm showing how to generate new points on the unit sphere and displacing them with the midpoint displacement method.

```
vec3 oldVertex1 = edgeVertex1;
vec3 oldVertex2 = edgeVertex2;
normalize(&oldVertex1);
normalize(&oldVertex2);

vec3 newVertex = oldVertex1 + oldVertex2;
normalize(&newVertex);

f32 vertex1_height = vector_lengh(&oldVec1);
f32 vertex2_height = vector_lengh(&oldVec2);

f32 scaledown = vector_lengh(&(oldVec1-oldVec2));
f32 newVertex_height = (vertex1_height + vertex2_height) / 2;
newVertex_height += (noiseValue*scaledown);
newVertex = newVec * newVertex_height;
```

To generate a new vertex the average of its two normalized neighbors is taken and normalized so the new vertex lies exactly between its neighbors in terms of latitude and longitude. Then to generate the elevation information, the average height of its neighbors is taken and a normally distributed random value is added on top. The random value however is scaled down by the distance of the two vertices, since the distance always halves, it has the same effect as multiplying it with $c\frac{1}{2^i}$ where c is the distance of two adjacent vertices in the icosaeder at subdivision level 0 and depends on the planets radius and i is the subdivision level. As a last step, the new height is applied to the vertex by multiplying it with its new height.

But since the same portion of the planet should look the same every time the player visits it – and the LODing system might have deleted the geometry in between the visits – the same random value has to be the same for every vertex every time. A way of doing this is using the vector on the sphere before applying the height information as seed for the normally distributed random number generator. To get a single value from a vector to be used as a seed, a hashing function can be used.

The hashing function in Listing 2.8 works by reinterpreting the floating point values as signed integers and adding them together. With this method the same random values will be generated for the same vertices. Floating numbers have a limited accuracy but the same vertex should always have the exact same position because the vertices in the iteration before have to be at the exact same position since they all came from the first set of 12 vertices, which will always be the same. The height information at some vertices has no influence on the position of the new vertex because it is normalized before being used as a reference point for the new vertex. Even if the calculation of the normalized vector is not fully accurate, it will be wrong by the same amount every time, making the process deterministic.

Listing 2.8: Hashing algorithm for a three dimensional vector. Idea taken from: `https://cs.stackexchange.com/questions/37952/hash-function-floating-point-inputs-for-genetic-algorith`

```
s32 getSeedFromCoordinate(f32 x, f32 y, f32 z) {
    s32 floatAsInt;
    s32 h = 1;

    floatAsInt = *(s32*)(&x);
    h = 31 * h + floatAsInt;

    floatAsInt = *(s32*)(&y);
    h = 31 * h + floatAsInt;

    floatAsInt = *(s32*)(&z);
    h = 31 * h + floatAsInt;

    return h;
}
```

A big benefit of using the midpoint displacement method over the simplex noise method is being able to use preexisting data. The algorithm works iteratively, meaning using the data from the previous iteration. This makes it possible to feed a base mesh of the planet into the program and let it calculate as many more steps as are necessary. This gives artists a great flexibility to only define the basic structure of the planet, for example until subdivision level 2 and let the program generate the rest. To illustrate this possibility, elevation information was queried from the Google Maps API at the locations the subdivided icosaeder has its vertices. A visualization of the vertices of the icosaeder mapped to the Google Maps map can be seen in Figure 2.11

The data from Google Maps is stored in a file where each line corresponds to a single vertex in the icosahedron mesh and contains the queried latitude, longitude and the elevation data. When read into the program the latitude and longitude will be rounded to a few decimal places as a 32 bit wide floating point value and then packed together into a single 64 bit unsigned integer value that can be used as a key in a map to query the elevation or stored sorted in an array. With this method the elevation data from earth could be stored and queried up to subdivision level 8 without missing a height information.

The extended midpoint displacement algorithm also needs the elevation information as parameter and when generating a new vertex, first checks if elevation information for its position is available, and if it is, applying it directly or if there is none, generate it in

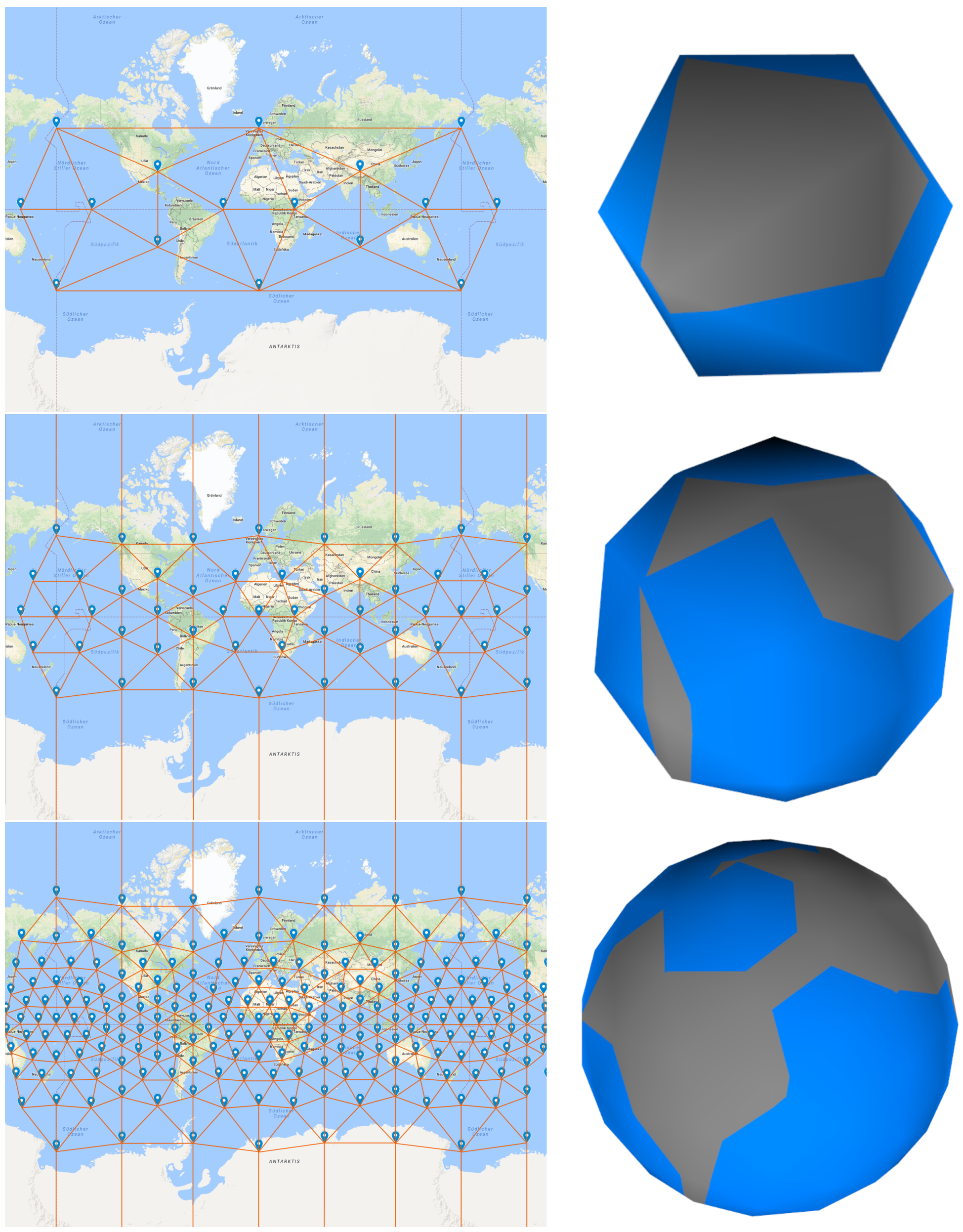


Figure 2.11: The position of the vertices of an icosahedron at the first three subdivision levels converted to latitude and longitude and plotted on the world map

the same way as described above.

Since the midpoint displacement method is able to work on a given base mesh, it is possible to use it in combination with the 3d noise method. The base mesh can be created by using the 3d noise approach for the first few LOD levels and after that the midpoint displacement algorithm can be used to generate the new geometry starting from the existing elevation information.

2.2.3 Variable roughness

In terrain generation the **roughness** describes how fast the height can change over the distance. When the overall roughness is set low, the planet tends to have less but bigger continents, while setting it higher results in many smaller islands. To add more variation to the generation and avoid having the same island or continent size on the whole planet, the roughness can be generated depending on the position on the planet. The updated function to generate 3d noise values can be seen in Listing 2.9.

Listing 2.9: The roughness has impact on the wavelength when looking up the height at a specific position while the roughness itself is depending on the position. Since the roughness is used as an input to the height noise map, the output is a warped noise

```
f32 get3dNoiseHeightAt(f32 x,f32 y,f32 z) {
    f32 n = 0;// = -0.0001f;

    // normally distributed between 0 and 1
    f32 roughness = getRoughnessAt(x, y, z);

    n += heightNoise(x, y, z, roughness * 200,   0.0009f);
    n += heightNoise(x, y, z, roughness * 400,   0.00045f);
    n += heightNoise(x, y, z, roughness * 2000,  0.0002f);
    n += heightNoise(x, y, z, roughness * 4000,  0.0001f);
    n += heightNoise(x, y, z, roughness * 8000,  0.00005f);
    n += heightNoise(x, y, z, roughness * 16000, 0.000025f);
    n += heightNoise(x, y, z, roughness * 32000, 0.0000125f);

    return n;
}
```

For that a tree dimensional noise map can be used, for example simplex noise. The wavelength should be fairly low so that the roughness does not vary too much since this looks unnatural and later will produce results that look like **warped noise**. An example of a landscape with strong warped noise features can be seen in Figure 2.12.

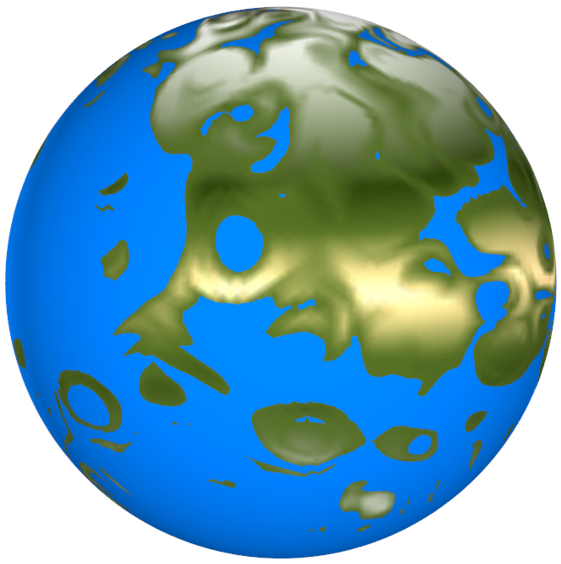


Figure 2.12: If the wavelength in the roughness map is set too high, the terrain will show noticeable features of warped noise.

Instead the wavelength should be so low that around the planet some larger areas have a noticeable higher roughness and thus smaller islands but some continents should still be large. This helps creating the illusion of plate tectonics. Or if realism is not a goal of the generation but it is rather desired to create a fantasy world, a smaller wavelength can be used so that the warping produces a more fantasy like terrain like in Figure 2.12.

2.2.4 Coloring the terrain

So far only the generation of the elevation of the terrain was discussed. In this chapter a method to color in the geometry is presented. The method is based on coloring in the vertices, the resulting color in between the vertices will be interpolated. An alternate solution will require future research and will be shown in chapter 5. Primarily the colors visible from the orbit of the earth are dependent on its latitude. Satellite images of the earth show, that near the poles the prominent color is white, because of snow and ice. On the way down to the equator, brown to greenish colors are prominent in areas of tundra. In desert areas the terrain has a color resembling the color of sand. A simple algorithm just checks the latitude of the vertex that should be colored in and solely depending from that assigns a color. For this a simple linear gradient can be used. The implementation and other usages of gradients will be discussed in section 3.1. For a simple approximation, these colors can be assigned to their respective latitude in the gradient and mirrored on the southern hemisphere. To improve on this coloring scheme, areas next to water are colored in more green than otherwise, since the water is a source for plants to grow and make an area appear more green. It is however hard to determine how far away water exists, especially because the vertex colors are assigned during the generation, so it is possible that not all relevant data is available how close or even if there is near water at all during generation. Instead it has proven to be a good approximation to use the height over the water level to determine how close water is found. Near seas or oceans this works really well, since the water level is the same around the planet. If however the terrain has really low elevation near the water level but is not close to the water it will be colored in green regardless.

3 Implementation details

Throughout the implementation, small design choices or implementations have proven to work really well and allowed for a better runtime or easier and more maintainable code base. This section will introduced these concepts. Not all patterns and methods are exclusive to procedural planet generation and can also be used in other contexts.

3.1 Gradients

When setting up the equations for the view range in respect to the players distance from the planet or to determine the color of a piece of terrain, the values should be continuous in that they do not have noticeable jumps. For example the colors should not have visible borders but flow into each other. Instead of setting up many linear equations which linearly interpolate a value table like Table 3.1 gradients can be used to easily look up arbitrary values without the need of setting up the linear functions. Gradients are based on *lerp*, an easy mechanism for linear interpolation. The fundamental equation for *lerp* is

$$lerp(A, B, t) = t \cdot B + (1 - t) \cdot A$$

Table 3.1: Table showing the view range on the planet given the players distance from the planet

distance	view range
0	3
5	6
10	10
15	15
> 25	20

where A and B are values or vectorial values and t is the interpolation step in the range $[0, 1]$. A value of $t = 0$ just returns A and a value of $t = 1$ returns B, for every number between that the weighted average between A and B is calculated. The algorithm to look up a value in a gradient first has to check in which segment on fe gradient the desired position is, in other words which line in the gradient's table similar to Table 3.1 is relevant. Then it calculates the t value based on the distance to the A and B value specified by the line in the table.

$$t = \frac{point - A}{B - A}$$

Then the gradient value can be determined by evaluating the `lerp` function with these parameters.

3.2 Mesh optimization

Until now, for the planets two meshes are used. One for the land mass and the other one for the water. It is however enough to store only one index tree and index butter as well as the `edgeToCenterVertex` map and use it for both meshes, since both meshes are always updated simultaneously and the patch on the surface of the mesh that has a high definition is based only on the players position and will always be the same for both meshes. This gives a huge performance improvement, because the mentioned data structures only have to be updated once for both land and water.

3.3 Multi threaded generation

To avoid performance issues and low frame rates, the meshes can be updates in parallel to the rendering using a second thread. This thread waits in the background until the mesh should be updated and then generates the new mesh and once it is finished, swaps the new mesh with the old one. The implementation of the thread logic can be seen in Listing 3.1.

3.4 Hook based memory management

When setting up the terrain generation multi threaded like described in section 3.3, the meshes are generated and then swapped out with the old ones. However the old ones can't be deleted in the terrain generation thread because the render thread still expects the geometry to be there. Instead a hook based memory management system has proven to work really well. Hooks are a concept that allow execution of user defined code at specific events. A hook is a collection of functions. So for the purpose of the planet generation two hooks where created the `systemShutdownHook` and the `afterMeshReloadHook` that are executed before the program exits and after the mesh has been reloaded. The hooks are implemented as `std::vector` of `std::function`. Since c++ lambdas can be implicitly casted to a `std::function` is really easy to insert code into a hook that will be executed later. The implementation of the hooks can be found in Listing 3.2 and a usage can be seen in Listing 3.3. Using these hooks it was easy to prevent memory leaks, and in general specify the lifetime of an object at creation time.

Listing 3.1: The thread logic checks repeatedly if the meshes need to be updates and calls a procedure to subdivide the planet at the given position in that case so that the players position has a high definition in the mesh.

```
while (true) {
    if (info->shouldStart) {
        info->isBusy = true;

        subdivPlanetAt(info->p, &info->playerPos, info->playerDist);

        info->isBusy       = false;
        info->shouldStart  = false;
        info->shouldUpdate = true;

    } else {
        // check 50 times a second if the terrain needs to be updated
        Sleep(20);
    }
}
```

Listing 3.2: The implementation of hooks as collection of `std::function<void()>`

```
#include <vector>
#include <functional>

#define runHook(hook)    \
    for(auto f : hook)   \
        f();             \
    hook.clear()

#define addHook(hookName, lambda) \
    hookName.push_back(lambda)

extern std::vector<std::function<void()>> systemShutdownHook;
extern std::vector<std::function<void()>> afterMeshReloadHook;
```

Listing 3.3: The hooks take a lambda as parameter and can be used to delete the old geometry when the planet is getting reloaded

```
addHook(afterMeshReloadHook, [=] {
    delete oldLand;
    delete oldWater;
    delete oldTree;
    delete oldIBuffer;
    delete oldMap;
});
```

4 Results and discussion

The different methods discussed in the previous sections er all implemented and compared to each other. This section is dedicated to showing the results of the implementation and analyzing the performance. The visual results featuring the coastline and the color generation are presented and the runtime of the generation and cleanup algorithms is compared. In the end a critical look at the available parameterization with the given generation algorithms is taken and the general use cases of the index tree are analyzed.

4.1 Coastline

When comparing the coastline resulting from a midpoint displacement generation in Figure 4.1 to the coastline when generating using 3d noise in Figure 4.2, it seems the coastline in the first image looks cleaner and more realistic.

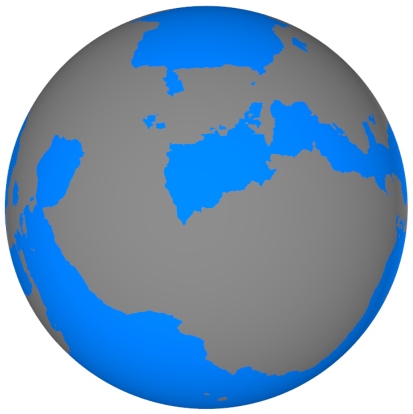


Figure 4.1: The coastline resulting from the midpoint displacement approach looks sharp and shows fractal features

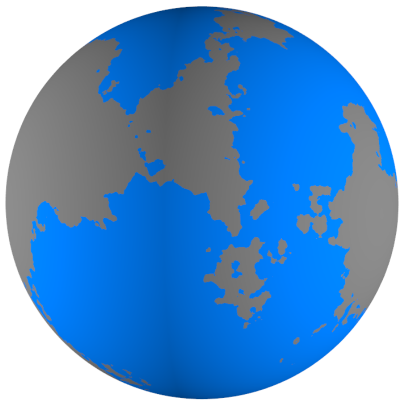


Figure 4.2: When generating with a 3d noise funciton hte coastline does not appear as sharp as the coastline generated with midpoint displacement in Figure 4.1

This is probably due to the layered noise values. The noise layers with the low wavelength which are responsible for the fine details that account for the fractal looking landscape will create little islands next to the main land.

4.2 Coloring

The colors play an essential role in as how realistically the result is perceived. The colors of the terrain in Figure 4.3 and Figure 4.4 were chosen to mimic the colors of earth that can be found on satellite images. To prevent the colors to just appear as parallel strips of colors, the y values of the coordinates that are used for the color determination are changed by another noise field. This adds variation and is an easy way to make the colors look more realistic.

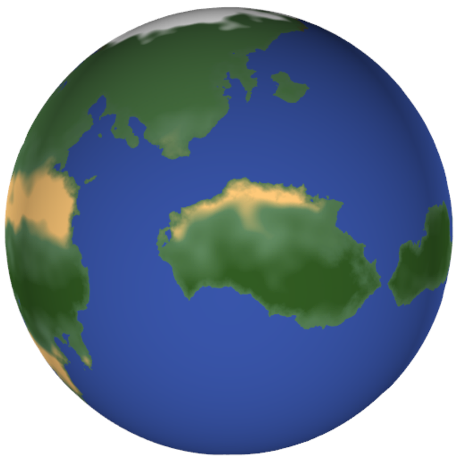


Figure 4.3: Vertex color and a distorted gradients were used to color in this planet

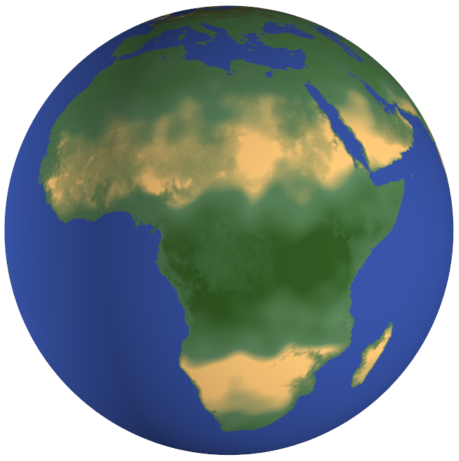


Figure 4.4: The gradient colors were chosen to resemble satallite images of the earth

4.3 Performance

During the implementation of the planet generator, the performance was measured and this section will illustrate the performance achieved by different settings or algorithms. Although the absolute numbers will not be the same as on other machines, they give an impression on how much influence some settings or algorithms have on the runtime. All tests were run on a Windows 10 machine with a Intel(R) Core(TM) i7-4790k cpu running at 4.00 GHz and a NVIDIA GeForce GTX 1070 and 8 GB of RAM.

To make the working conditions similar for all tests, the planet will be generated at the same subdivision levels with a constant view range large enough to fully subdivide half of the planet. The results can be seen in Table 4.1. The cleanup algorithm works like described in section 2.1.2. The `treeCheckDepth` variable controls the maximum depth in the tree that the faces will get checked for generation. This variable was introduced because it seems easier and more efficient to check only a few nodes deep if they are in range and then when reached the `treeCheckDepth` depth, just subdivide everything because otherwise possibly really many faces would have to be checked if they are in range. However looking at Table 4.1 it seems it is more efficient to make more checks rather than blindly generating the geometry. For the measurements in the last row, `treeCheckDepth` was set to 10, meaning it never skipped a check for all subdivision levels and it performs better for all tested subdivisions than only checking until depth 5 in the row above.

Table 4.1: Comparing the average runtime of a full rebuild, a single cleanup run and updating the geometry with the subdivision levels specified. The number after "update" is the depth in the tree that is checked for generation.

subdivisions / algorithm	4	5	6	7	8
full geometry rebuild	10 ms	38 ms	105 ms	616 ms	2453 ms
cleanup	5 ms	20 ms	84 ms	345 ms	1624 ms
update (5)	1 ms	4 ms	19 ms	84 ms	389 ms
update (10)	1 ms	4 ms	16 ms	77 ms	336 ms

When taking a look at the cleanup run it turned out that a big portion of the run time was used on updating the indices in the `edgeToCenterVertex` map. For test purposes the code to create the updated map was commented out and instead an empty map was used. This however creates many duplicate vertices as the algorithm for generating new vertices does not find the old existing vertices in the map and generates them again. This leads to fewer generation runs until the vertex buffer has to be cleaned up again. The run times for single cleanups and the average generation time as well as the average runs before a cleanup can be seen in Table 4.2. As the vertex cleanup threshold 75% of the vertices of a subdivided icosahedron at that subdivision level were used.

Table 4.2: Comparing the two approaches to update the `edgeToCenterVertex` map ("update map") and to leave it empty ("clear map")

subdivisions / max vertex count / test case	4 1922	5 7682	6 30722	7 122882	8 491522
clear map: cleanup time	5 ms	15 ms	55 ms	253 ms	1016 ms
update map: cleanup time	8 ms	25 ms	99 ms	417 ms	1758 ms
clear map: runs before cleanup	163	206	201	79	21
update map: runs before cleanup	288	305	282	129	26
clear map: overall aver. runtime	1 ms	4 ms	16 ms	83 ms	438 ms
update map: overall aver. runtime	1 ms	4 ms	16 ms	70 ms	440 ms

The overall run times for both methods, clearing the map and updating the map are not far apart for every subdivision level. Which implies that in general it does not matter if the indices in `edgeToCenterVertex` are updated during the cleanup. Which raises the question if the map is necessary at all. A benchmark testing the runtime without using the map at all compared to the previous run with updating the map from Table 4.2 can be seen in Table 4.3.

Table 4.3: The `edgeToCenterVertex` map is necessary to keep the overall runtime low, even if clearing or updating it does not seem to make a difference in Table 4.2

test case \ subdivisions / max vertex count	4 1922	5 7682	6 30722	7 122882	8 491522
no map: cleanup time	6 ms	21 ms	82 ms	326 ms	1492 ms
update map: cleanup time	8 ms	25 ms	99 ms	417 ms	1758 ms
no map: runs before cleanup	3	4	6	7	4
update map: runs before cleanup	288	305	282	129	26
no map: overall runtime	3 ms	12 ms	41 ms	161 ms	999 ms
update map: overall runtime	1 ms	4 ms	16 ms	70 ms	440 ms

When not using the map, the runs before cleanup shrink drastically forcing a cleanup much faster and the overall run time is slower as a result.

When comparing the two generation methods midpoint displacement and 3d noise, it seems that midpoint displacement is consistently around 20% slower than the 3d noise method, even with seven layers of noise. The midpoint algorithm has to calculate the elevation for the two neighboring vertices and additionally hash the position of the new vertex to get an seed for the normally distributed random number generator. Further studies are needed to improve on the way that midpoint displacement can be carried out on the surface of the sphere.

4.4 Parameterization

There are a few different ways how the user can influence the outcome of the generated planets. This section is dedicated to the parameters the user can set to generate a planet that is closer to the users expectation as a completely random planet.

4.4.1 Water level

By default the planet is made up from the water and the land mass mesh. The water mesh is a subdivided icosahedron with no additional hight information, thus all vertices lie on the surface of a sphere with the planets radius. The terrain is then generated using a normally distributed noise function. Thus about half of the terrain will be above water level and the rest will be below. On earth, 71% of the surface is covered by the oceans [Ins].

For the generated planet to look more earth-like and realistic, the percentage of surface covered with water should resemble the ratio in earth. As can be seen Figure 1.1 in section 1.3, the spread of the normal distribution is depending on the used σ value. Depending on σ a small negative value should be picked that is used as a basis of the accumulation of the noise values to lower the average height of the terrain. If generating with the midpoint displacement method this value will just be added onto the calculated new height, to also lower the terrain. Like this the user has more control over the water levels and can chose to generate a more earth-like planets or ocean planets where there are only small continents and islands, depending on the desired setting.

4.4.2 Coloring

The colors of the planet are given based on a color gradient that is provided by the user. In the simple implementation, the absolute value of the y coordinate is used as to look up the color value from the gradient. As an effect the colors are mirrored on the equator plane. If the user wants to have more control it is trivial to change the implementation so that the planet can have a different gradients on both hemispheres. The colors in the screenshots in section 4.2 were chosen to resemble the colors on satellite images of the earth but depending on the desired setting any colors can be chosen to be also able to generate alien looking planets.

4.4.3 Terrain generation

When using the 3d noise approach to generate the terrain, the user also has control over the different layers of noise that will be added on top of each other to produce the height data. To produce a realistic looking terrain with a fractal-like coast line, the wavelength should be doubled form one layer to the next, but is should be scaled down by double the amount as the layer before. Like this it is possible to introduce self similarities. However the lowest layer of the noise functions can be chosen to model the desired sized of the continents since it is the layer that is scaled down the least amount. By choosing a higher value for the wavelength, more and smaller continents and islands will be generated. If the wavelength is chosen bigger, less but bigger continents are created.

Also the coordinates on the sphere are not directly used as input to the height noise but rather they are scaled by the roughness at this point. If the roughness is low, the resulting wavelength will be low and the terrain looks less rough. However if the roughness map itself uses a wavelength that is too small, the resulting terrain will show features of warped noise, like seen in Figure 2.12. in subsection 2.2.3 This can be used deliberately as an effect, if creating fantasy like worlds is the goal.

4.5 Usages of the index tree

The index tree is a major result of this work and can also be used for other purposes other than holding geometrical data for planets. The benefit of using the index tree over just the index buffer or an octree is that is natively corresponds to the topology of the mesh and makes LODing very easy. In general the index tree can be used when different parts of the model should have different levels of detail. To generate the next LOD level, there must exist concrete rules to place the vertices so that te geometry will always look the same, or alternatively the vertex position has to be stored in a way that can be looked up upon generation like done with the earths elevation information in subsection 2.2.2. However the use cases are not limited to spherical meshes or even triangular meshes as long as there exists a clear rule set how the new vertices are generated. Because of the nature of a tree structure, most operations like generating or deleting geometry can be done in logarithmic time in respect to the number of nodes since only certain branches have to be explored. A cleanup or the generation of an index buffer requires linear time, since every node has to be visited.

To use the index tree for an arbitrary mesh, the faces of the lowest LOD level will be the children of the root tree node. In the example of generating planets there are the 20 faces of the icosahedron but this depends on the structure of the mesh that should be represented. Then, depending on the camera's position, the area of interest can be determined and the tree nodes that lie in this area should be subdivided until the desired level of detail is reached. If an area is not needed anymore it can be deleted by deleting the children of the tree nodes that encapsulates the unneeded geometry. Furthermore the vertex buffers can be cleaned from dead vertices like described in section 2.1.2.

5 Future work

This work had its focus on the more technical side of generation and representing the data in the memory. Even these aspects may not be optimal, but serve as a solid foundation for future work in any case and enable for more research to make the resulting planets appear more realistic. This section introduces possible next fields that can be explored that are based on this work.

5.1 Sharp cliffs with overhangs

A limitation of the current implementation is that the resulting terrain can never have sharp cliffs and overhangs, because the terrain only consists of a displaced spherical base mesh. To allow for cliff and overhangs, a position given by its latitude and longitude could have multiple vertices stacked on top of each other, for example one on the surface of the overhanging cliff and one on top of the cliff.

A simple solution is, when generating new vertices to also allow the displacement on the sphere and not only in the height. This displacement should however be small and only noticeable on the last subdivision steps, since the overall vertex density should stay roughly equal around the sphere. With this technique little overhangs will be possible. When it is however wanted to create extreme overhangs like caves, other methods have to be researched.

5.2 Texturing

Right now the mesh uses vertex colors to color in the terrain. The colors in between the vertices are interpolated between the colors of the surrounding vertices. Like this it is impossible to use high definition textures, since at every pixel there would have to be a vertex and so the number of vertices would be too large to run performantly. Instead a image texture can be mapped onto the surface on the planet to create a high resolution texture of the ground. However as described in section 1.6, when mapping a plane onto a sphere the mapping is never perfect in that the information density is inconsistent. What can be done instead is generate a texture for each face, as an additional attribute for every index tree node. The visible size of the triangles on the screen should always be roughly the same since according to section 1.5 geometry nearer to the player has to be more subdivided and geometry farer away from the player has to be less detailed.

5.3 Water level

Until now the water level is constant around the planet. This means that there will never be land that is under sea level, even if it is technically physically possible. For example parts of the Netherlands have a negative elevation compared to the sea level. Also it is impossible to have lakes higher than sea level. This can be fixed by giving parts of the water mesh a higher or lower elevation. However to prevent visibly uneven water surfaces, care has to be taken to have an even water surface at all times – the surface of the water is only allowed to be uneven under the ground.

5.4 Biomes

To create a richer diversity of sceneries different biomes could be generated. Each position on the world would be assigned to a biome or each position in the world has a weighted average as its biome influenced by the biome center points. Biomes should probably also be dependent on the latitude to provide realistic looking biomes. A good basis for the biomes can be a 3d cell noise where each cell corresponds to a biome. The seeds of the cells should be placed so that the resulting biomes seem natural. For example near the equator desert or rain forest biomes should be placed. To avoid the sharp corners between the cells, the cell noise can be warped using a simplex noise.

5.5 Clouds

When looking at the earth from the same distance as in the screenshots in chapter 4, the observer would not have such a clear sight on the terrain because there are clouds. Typical cloud formations can be seen in Figure 5.1.

Because of the nature of air, particles and even whole streams of air near each other move roughly in the same directions and some cloud formation resemble the looks of a warped noise field and might be able to be generated using warped noise. Other formations like the vortex shaped storms require another technique to be generated. To avoid placing storms using the vortex generation method over existing clouds generated by warped noise, the vortexes should be generated first. After that, the warped noise clouds will be placed on top of the whole world but leaving out the areas that are affected by storms. Since it seems that there are no regular clouds near the storms.

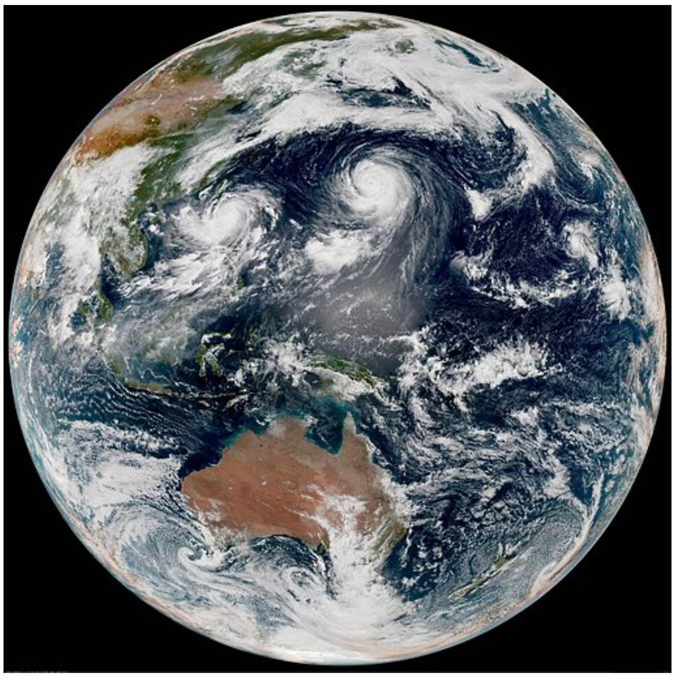


Figure 5.1: Typical cloud formations featuring typhoons and tropical storms [Age]

6 Summary

In this work a broad overview of the fundamentals and different methods of procedural generation was given. The different types of procedural generation were discussed and its usages shown. The problems arising when mapping flat textures onto a sphere where explained using the two projection methods mercator projection and peirce quincuncial projection. To avoid projection issues the terrain was generated directly on the sphere. For that, different kinds of spherical meshes have been introduced and compared to each other. Since the icosahedron has a relatively consistent vertex density it was chosen as a base mesh for the planet generation. To be able to have a adaptive level of detail, the index tree was introduced and it was explained how it can be used to increase or decrease the LOD in specific areas. To generate the actual elevation information, two methods were presented. The first one is based on 3d noise functions. By adding multiple layers of noise with different wavelengths and scaling them accordingly, a fractal noise can be created to mimic the fractal looking coastline shapes on earth. The second approach, the midpoint displacement method, generates a fractal noise incrementally. An advantage of using the midpoint displacement method is, that it can work with preexisting elevation information. To explore this, a mechanism to store and load elevation data was shown and the elevation data from the earth was used to demonstrate the ability to load the earth's elevation up until a certain level of detail and from that point on generate the remaining geometry using midpoint displacement. Both methods work directly on the sphere and no projection is required. A method involving another noise function was shown to achieve a varying level of detail across the surface of the planet so that some areas appear rougher and some are rather flat. Using gradients, the vertices could be colored in with colors that resemble satellite images of the earth to look more realistic. To avoid strips with the same color an additional mechanism to add variance to the colors was shown. The different methods and implementations were then compared to each other and the results were discussed. This work serves as a foundation for future research allowing the generation of more realistic earth-like planets.

7 References

[Age] Japan Meteorological Agency. Japan meteorological agency's website. http://www.jma.go.jp/jma/indexe.html.

[Apa] Apavlov. Apavlov's weblog. https://apavlov.wordpress.com/2011/08/24/bush/.

[Ast] Astroneer. Astroneer's website. https://astroneer.space/.

[Bae11] John Baez. Fool's gold. http://math.ucr.edu/home/baez/golden.html, 2011.

[Coo09] Stephen Coombes. The geometry and pigmentation of seashells. *Techn. Ber. Department of Mathematical Sciences*, 2009.

[Ins] Hawaii Pacific University Oceanic Institute. Aqua facts. https://www.oceanicinstitute.org/aboutoceans/aquafacts.html.

[Law97] Jim Lawrence. A short proof of euler's relation for convex polytopes. *Canadian Mathematical Bulletin*, 40(4):471–474, 1997.

[Lie16] Jacob Liechty. https://www.reddit.com/r/Astroneer/comments/56z055/have_the_developers_gone_into_detail_on_the/, 2016.

[Mak] MakeHuman. Makehuman's website. http://www.makehumancommunity.org/.

[Pei79] C. S. Peirce. A quincuncial projection of the sphere. *American Journal of Mathematics*, 2(4):394–396, 1879.

[Per10] Markus Persson. Hrmpth. trade-offs. https://notch.tumblr.com/post/729098863/hrmpth-trade-offs, 2010.

[Per11] Markus Persson. Terrain generation, part 1. https://notch.tumblr.com/post/3746989361/terrain-generation-part-1, 2011.

[Qui] Inigo Quilez. Inigo quilez' website. http://www.iquilezles.org/www/articles/warp/warp.htm.

[Ran06] Bill Rankin. Wall maps of the world. http://www.radicalcartography.net/?projectionref, 2006.

[Sil18] Bernard W Silverman. *Density estimation for statistics and data analysis.* Routledge, 2018.

[Smi84] Alvy Ray Smith. Plants, fractals, and formal languages. *SIGGRAPH Comput. Graph.*, 18(3):1–10, January 1984.

[Sny78] John P Snyder. The space oblique mercator projection. *Photogramm. Eng. Remote Sensing*, 44:585–596, 1978.

[TYSB11] Julian Togelius, Georgios N Yannakakis, Kenneth O Stanley, and Cameron Browne. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):172–186, 2011.