OAMK
OULUN AMMATTIKORKEAKOULU

Henry Vuontisjärvi

PROCEDURAL PLANET GENERATION IN GAME DEVELOPMENT

OAMK

OULUN AMMATTIKORKEAKOULU

PROCEDURAL PLANET GENERATION IN GAME DEVELOPMENT

Henry Vuontisjärvi
Opinnäytetyö
Kevät 2014
Tietotekniikan koulutusohjelma
Oulun seudun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietotekniikan koulutusohjelma, ohjelmistokehityksen suuntautumisvaihtoehto

---

Tekijä(t): Henry Vuontisjärvi
Opinnäytetyön nimi: Procedural planet generation in game development
Työn ohjaaja(t): Veikko Tapaninen, Marjo Heikkinen
Työn valmistumiskuukausi ja -vuosi: Kevät 2014
Sivumäärä: 33

Opinnäytetyön tavoitteena oli toteuttaa lisäosa Unity-pelimoottoriin, jolla voi luoda pelinkehitykseen sopivia kolmiuloitteisia planeettamalleja käyttäen ohjelmallisen sisällön luomisen tekniikoita (procedural generation). Lisäosa muodostuu kolmesta osasta: Unityn editorissa toimiva työkalu pinnan muotojen visuaalista suunnittelua varten, kolmiulotteisia malleja tehokkaasti laskeva ohjelmakoodi sekä ohjelman ajonaikana lisää yksityiskohtia tuottava ohjelmakoodi. Työssä käydään läpi käytetyt tekniikat ja esitellään projektin lopputulokset.

Työssä hyödynnettiin aiemmin Java-sovelluksena toteutetun planeettageneraattorin tekemisen aikana opittuja tietoja. Työ toteuttiin käyttäen Unity-pelimoottoria ja C#-kieltä Monodevelop-kehitysympäristössä.

Työn tuloksena syntyi lisäosa joka julkaistiin Unity Asset Storessa.

Asiasanat: ohjelmallinen sisällön luominen, planeetat, pelit, pelinkehitys, Unity

ABSTRACT

Oulu University of Applied Sciences
Information Technology, Software Development

Author(s): Henry Vuontisjärvi
Title of thesis: Procedural planet generation in game development
Supervisor(s): Veikko Tapaninen, Marjo Heikkinen
Term and year when the thesis was submitted: Spring 2014
Pages: 33

The subject of this thesis was to produce an editor extension for the Unity engine that generates three dimensional planetary terrain models suitable to be used in game development. The plugin has three main parts: a node editor tool within Unity editor that enables visual design of noise functions to be used on the surface, a robust code for generating 3D models during runtime and a dynamic level of detail system. This thesis details the techniques used and the results of the project.

The work was based on previous knowledge learned during creating a similar application using Java. The work was implemented using the Unity engine with C# programming language in MonoDevelop development environment.

The work resulted in a working plug-in that was released in the Unity Asset Store.

Keywords: procedural generation, planet, game development, Unity

TABLE OF CONTENTS

TIIVISTELMÄ 3

ABSTRACT 4

TABLE OF CONTENTS 5

ABBREVIATIONS 6

1.INTRODUCTION 7

2. PROCEDURAL CONTENT GENERATION 8

2.1. Usage in game development 8

2.2. Terrain generation 9

2.3. Planet generation 12

3.ALGORITHMS AND THEORY 13

3.1. Pseudo-random number generator 13

3.2. Coherent noise 13

3.2. Perlin noise 14

3.3. Fractional Brownian motion 15

3.4. Sphere tessellation 15

3.5. Height mapping 17

3.6. Applying height maps to a sphere 18

4. PLANETARY TERRAIN 20

4.1.Requirements 20

4.2.Tools 20

4.2.1. Unity 20

4.3. System architecture 21

4.4. Node-editor 22

4.5. Planet mesh generator 24

4.6. Runtime dynamic level of detail 25

4.7. Shading 26

4.8. Vegetation 28

5.CONCLUSION 30

REFERENCES 31

ABBREVIATIONS

FBM	Fractional Brownian Motion
PCG	Procedural Content Generation
PRNG	Pseudo-Random Number Generator
LOD	Level of detail

1.INTRODUCTION

Creating large amounts of interesting content is a problem that game developers often face when designing their games. The best results are usually achieved by manually designing and implementing the content, but this requires lots of work and the amount of content will always be finite; the player will eventually run out of it. One way to tackle this problem is to instruct the computer to create this content, a method referred to as procedural content generation (PCG).

This thesis describes the theory of some of the common procedural methods used in game development context focusing on terrain generation. Finally there is a case study of a planet generator solution created during this project and review of the methods used in it.

The project is based on a previous work which was started as a study of Java programming language, OpenGL graphics library and the procedural planet generation task itself. Because originally the reason was to learn as much as possible, everything was created from scratch. Eventually the project reached a point where the basic 3D graphics renderer and a simple planet model was done and the project was completed in its initial scope.

When the thesis project was started, the original project was recreated in the Unity engine and polished to make it more accessible for other developers to use. Unity’s easily extendable editor provided the possibility to create a visual node-based editor for the noise functions which made the project much more user-friendly. The finished version was released in December of 2013 in the Unity Asset Store (1).

2. PROCEDURAL CONTENT GENERATION

Procedural content generation can be defined as *the algorithmical creation of game content with limited or indirect user input* (2). It may utilise simulation and mathematical concepts such as fractal geometry and pseudo-random algorithms in an attempt to create what the user desires. Almost anything can be generated with sufficient logic and the quantity of content created this way can be seemingly infinite.

2.1. Usage in game development

There are two main uses for procedural content generation (PCG) in game development: generating playable content and generating assets. More possible uses exist, such as generation of game rules, but their applicability depends on how we define the keyword “content”. One such definition describes: *content is most of what is contained in a game: levels, maps, game rules, textures, stories, items, quests, music, weapons, vehicles, characters etc. (3).*

Generating assets refers to generation of content that is not directly related to gameplay, but it is there to support the game experience. This includes graphical and audible content such as 3D models and textures.

Generating playable content refers to generation of game levels, stories, characters or units, items and any other interactive content that the player consumes.

The methods of procedural generation are often based on randomised algorithms that are seeded with pseudo-random number generators (PRNG) that use predefined seed values to produce the same results if given the same seed and parameters. The methods include noise generators, fractal systems, cellular automata, specialised logic routines and simulation.

Simulation in this context rarely means simulating the actual processes that created the real world counterparts. The level of abstraction is usually very high.

For comparison we can think what would it take to simulate the creation of a patch of terrain with hundred percent precision: one would have to simulate billions of years of geological processes on atomic level. Clearly this is not feasible nor necessary. Fortunately computer games are all about illusion, creating an image to the mind of the viewer. In this context the simulation can be only superficial and in the end it is more important that the end result looks convincing than that it is accurate.

Historically one of the reasons for developing procedural generation was that *the earliest computer games were severely limited by memory constraints* (4). The programmers developed algorithms that would generate lots of data from small amount of seed data.

Another reason for using PCG is the lack of resources to manually design large amounts of content. PCG methods can provide that but the initial investment of resources to develop the generator is higher. With manual content creation the amount of invested resources will rise in correlation with the amount of finished content, as with the PCG methods the initial investment is higher, but after the generator is finished more further content can be generated with reduced cost.

PCG can also be used as a tool to enhance manual content creation. The procedural generator can save the designer from having to do repetitive tasks, for instance to create variations or trees for a forests.

One of the considerations with PCG is quality assurance. The algorithms may be time consuming to develop to a point where acceptable results can be guaranteed for all cases. One solution to this is to introduce validity testing procedures that ensure that the generated content fulfils all requirements.

2.2. Terrain generation

Generating terrains is perhaps one of the most common applications of PCG.

In game development context the requirements for the terrains greatly vary. Ultimately the environment is there to serve the gameplay and therefore realism is

secondary, but still the environment should be detailed enough to immerse the player in to the game setting.

The scope and the scale of the game also alter the requirements for the environment. A game where the player views the game world from a distant birds-eye view has very different requirements compared to one where the player walks across the landscape.

Side-scrolling games often use one dimensional terrain representation such as in IMAGE 1. This type of terrain has varying height over certain length (the length can be infinite).

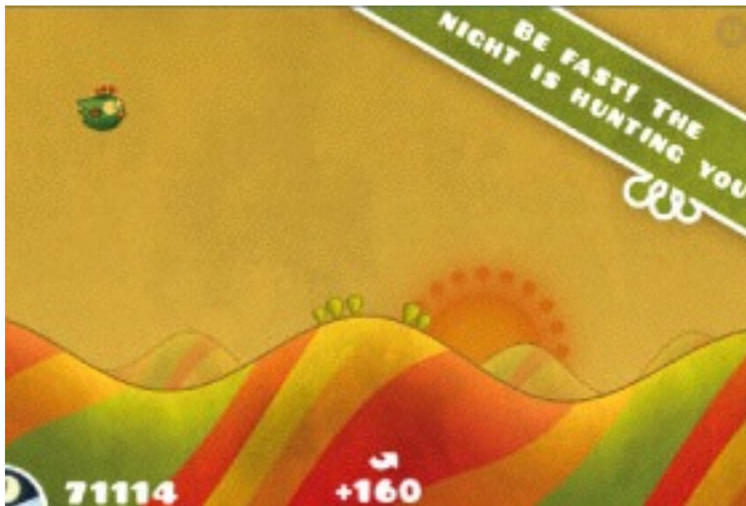


IMAGE 1. “Tiny Wings” on iOS (5)

Height maps are often used in when the player can move mainly two dimensionally over the terrain. The representation of the terrain may also be two or three dimensional using the same height maps. The drawback is that caves or overhangs cannot be represented using height maps since each cell only has one height value (IMAGE 2).

IMAGE 2. Procedural terrain in “Civilization V” (6)

Volumetric terrain representation has became more popular in recent years. It allows terrains to be truly three dimensional with extensive case systems (IMAGE 3).

IMAGE 3. “Minecraft” (7)

There are many ways to generate terrain data but the most prevalent are the midpoint-displacement algorithms (8) and noise functions such as Perlin noise (9).

2.3. Planet generation

There are number of examples of procedurally generated planets in game development including:

- Procedurally generated planets in “Spore” (10),
- Kerbal Space Program (11),
- I-Novae-engine (12),
- “My First Planet” by Alex C. Peterson (13),
- Work of Sean O’Neil (14),
- Making Worlds by Steven Wittens (15).

IMAGE 4. Planets in “Spore” (16)

3.ALGORITHMS AND THEORY

3.1. Pseudo-random number generator

One of the essential technologies to computationally generate new information is the ability to produce seemingly random sequences of numbers. A pseudo-random number generator (PRNG) produces numbers that approximate a truly random function. PRNG sequences are not considered truly random since they are completely determined by a small set of initial values (17). However this level randomness is sufficient for most purposes related to computer game content creation.

The initial values of a PRNG are called *seeds* and usually represented by an integer. Using the same seed will produce the same sequence of seemingly random numbers. This feature of a PRNG makes it possible to store a vast amount of generated information in to a single integer number.

3.2. Coherent noise

Coherent noise is a *type of smooth pseudorandom noise* (18) that has no sharp discontinuities in the output values (IMAGE 5). A coherent noise function and takes n-dimensional coordinate parameters and returns a single output value. The function is based on a PRNG and therefore returns the same output value each time with the same parameters.

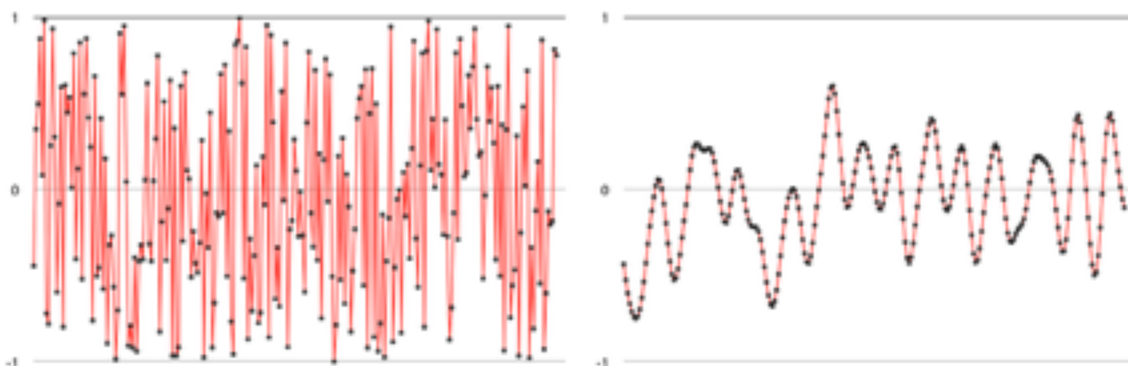


IMAGE 5. One dimensional non-coherent noise (left) and coherent noise (right). Notice how the adjacent data points on the non-coherent noise have large value differences and therefore very little continuity.

The basis for all coherent-noise functions is the integer-noise function (19). This kind of function takes an integer value as a parameter and returns a floating-point value. The important features are that the same values are always returned with the same input parameter and that the outputs of neighbouring input values have no correlation between each other (in other words are in a random sequence).

The next step is to make the previous discrete function continuous by adding interpolation. The simplest is linear interpolation (bilinear and trilinear for higher dimensions) but usually cubic or quintic interpolation is required to produce good quality results. The function simply interpolates between the original integer-noise values using the chosen interpolation function.

Next improvement of the noise function is to use random gradient vectors at the integer boundaries. This method displaces the noise in random directions and therefore gets rid of the grid-like appearance of the previous steps. The function needs to *calculate the influence of each pseudorandom gradient on the final output, and generate our output as a weighted average of those influences* (20).

The end result of this method is referred to as Perlin noise (IMAGE 6).

IMAGE 6. The stages of coherent noise generation in 2 dimensions. From left to right: integer-noise, linear interpolation, cubic interpolation and Perlin noise. Image from LibNoise (19).

3.2. Perlin noise

Perlin noise is named after Ken Perlin who originally created the algorithm while working on the movie “TRON” in 1983. He was awarded an Academy Award (Oscar) for technical achievement for it and the algorithm has since become a standard for the computer graphics field both in movies and games.

Perlin himself describes noise as *texturing primitive you can use to create a very wide variety of natural looking textures* (21).

The problem Perlin was facing was the fact that computer graphics can render shapes with perfect mathematical precision and such shapes do not appear natural. Real world surfaces have small imperfections that contribute to the overall look and character of the surfaces. Perlin created the coherent noise function to add more realism to the graphics.

The original paper describing the usage of noise in texture generation was released in 1985. Perlin has since published another paper titled “Improving noise” (22) that describes an improved noise algorithm that fixes minor discontinuities in the original algorithm.

3.3. Fractional Brownian motion

Fractional Brownian motion (also called fractal Brownian motion, abbreviated FBM) is often used in a method of layering multiple octaves of coherent noise to make the image have more varied appearance. Each subsequent layer has its frequency doubled and amplitude halved. Applying this method to the coherent noise can give it features that appear self-similar (23), a typical property of fractals.

3.4. Sphere tessellation

There is no single obvious way to divide a spherical surface into regular sections or grids. (24) One of the requirements for a procedural planet with a dynamic level of detail is the ability to tessellate sphere surface in such a way that further subdivision is possible. Another requirement is that the tessellation should be as uniform as possible in a way that the detail is same in all areas from the equator to the poles.

Different tessellations that were considered include: (IMAGE 7)

1. Geodesic sphere which is formed by subdividing the edges of an icosahedron,

2. UV sphere which is based on latitude and longitude coordinates,
3. Quad sphere which is formed by mapping each point in a cube surface on to a sphere.

The most common one in 3D-modelling is the UV sphere, but this approach has the downside of increasing detail around the poles and having less detail in the equator making it unsuitable for this purpose.

The geodesic sphere has uniform tiles and therefore no distortion at the poles, but the nature of the shape makes the calculations involved in generating the mesh and further detail levels more complex.

A quad sphere also has the problem that the tile size and shape is not uniform across the surfaces and has slight distortions at the corners of the cube, but the distortion is small enough to be tolerable in this application.

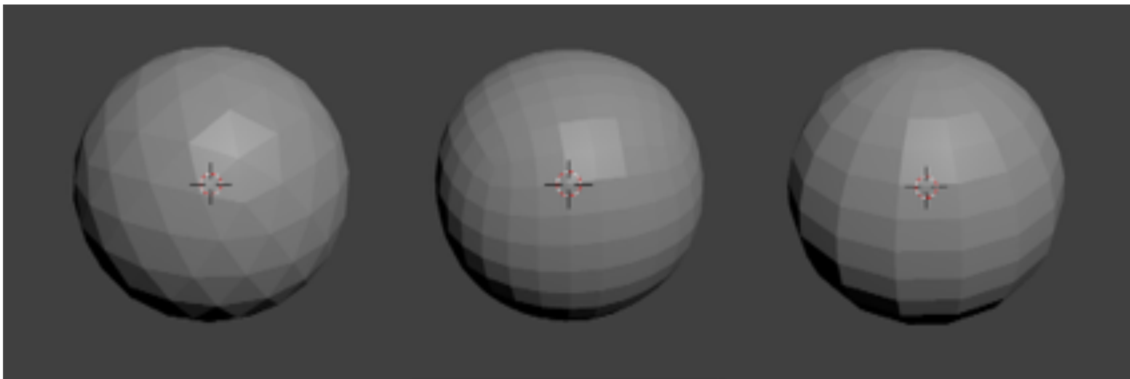


IMAGE 7. From left to right: subdivided icosahedron, quad sphere and UV sphere. Notice the tile sizes of each sphere type: icosahedron has triangular tiles of uniform size; quad sphere has rectangular tiles of approximately same size but distortions near the corners; and UV sphere has rectangular tiles that are larger at the equator and approach singularity at the poles.

The main reason for selecting the quad sphere is the easy further subdivision of the surfaces. The shape is essentially a cube, making each side a rectangle and therefore all subdivided surfaces are smaller rectangular shapes (IMAGE 8).

The 3D vector coordinates of the points that form the cube are stored as six arrays of points, one for each side of the cube. The number of points determine the detail level of the resulting model.

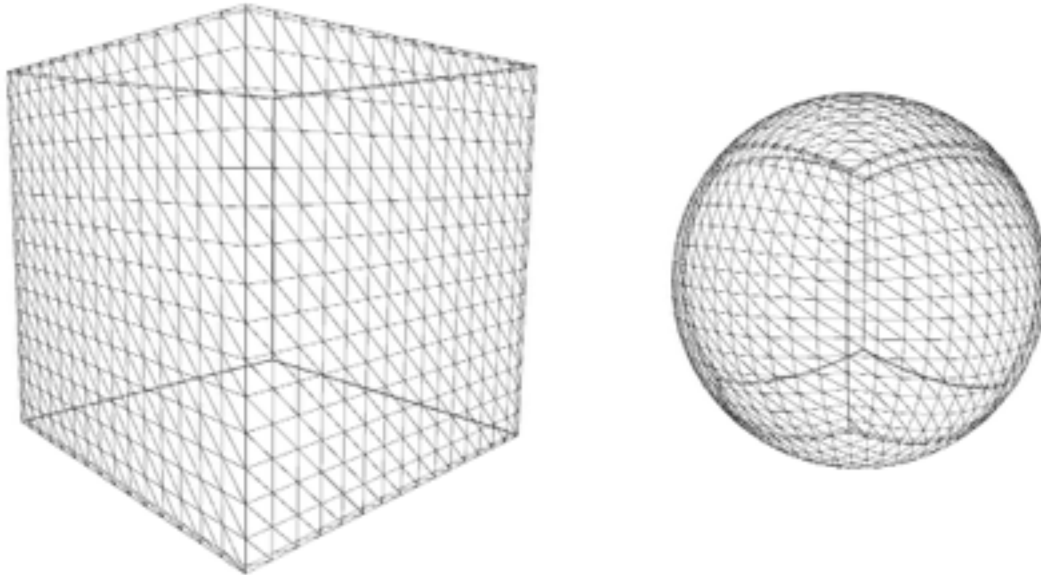


IMAGE 8. Wireframe image of a cube before and after transformation.

A mathematical formula is then applied to each of the points, mapping them in to points on a unit sphere surface (FORMULA 1). The formula used here was originally developed by Philip Nowell (25).

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

FORMULA 1. The formula for transforming a point on a cube to a point in sphere surface (25).

3.5. Height mapping

A common technique of representing the terrain in a modern game is a 3D model based on a height map.

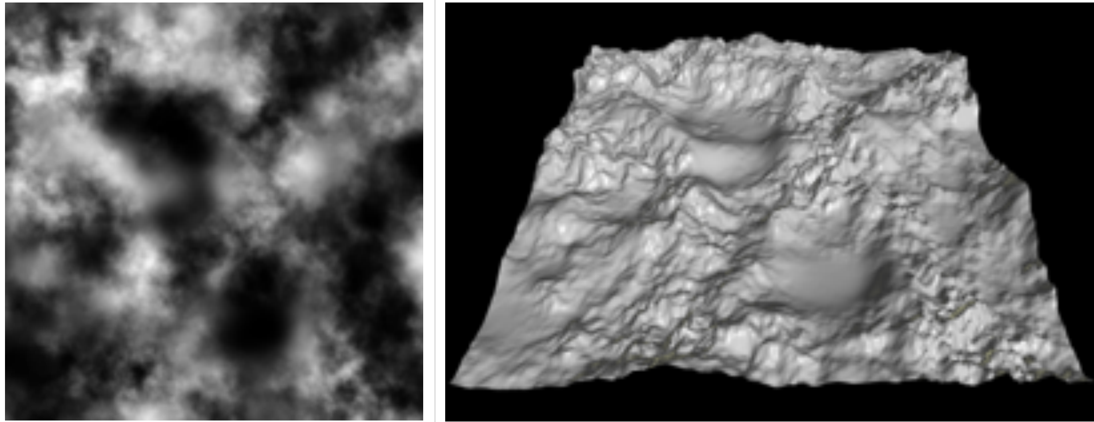


IMAGE 9. A height map displayed as a greyscale image (left) and a 3D surface formed by the same height map (right). (26)

Height maps or height fields are 2-dimensional tables where the *value of each cell corresponds to height of the terrain (over some baseline) at that point* (27). The advantages of this method are efficiency of storage and easy access to the data. The drawback is that height map can only represent one height for each position and therefore it is impossible for a terrain represented this way to have caves, cliffs or overhangs.

The table structure of a height map closely resembles that of a raster image or bitmap and therefore they are often visualised as a greyscale images (IMAGE 9). The height values are mapped to a gradient with 0 being black and 1 white.

3.6. Applying height maps to a sphere

In order to apply noise on a spherical surface and make sure the mapping wraps in all directions is to use a three dimensional noise function and use the normalised vector between the sphere centre and the surface position coordinates as the input parameters (IMAGE 10).

This approach also makes it possible to have a dynamic level of detail: additional points added between the previous points can call the same noise function and get the correct height values.

It is not necessary to have the height maps as actual images at any point as the mesh generation function can directly call the noise function.

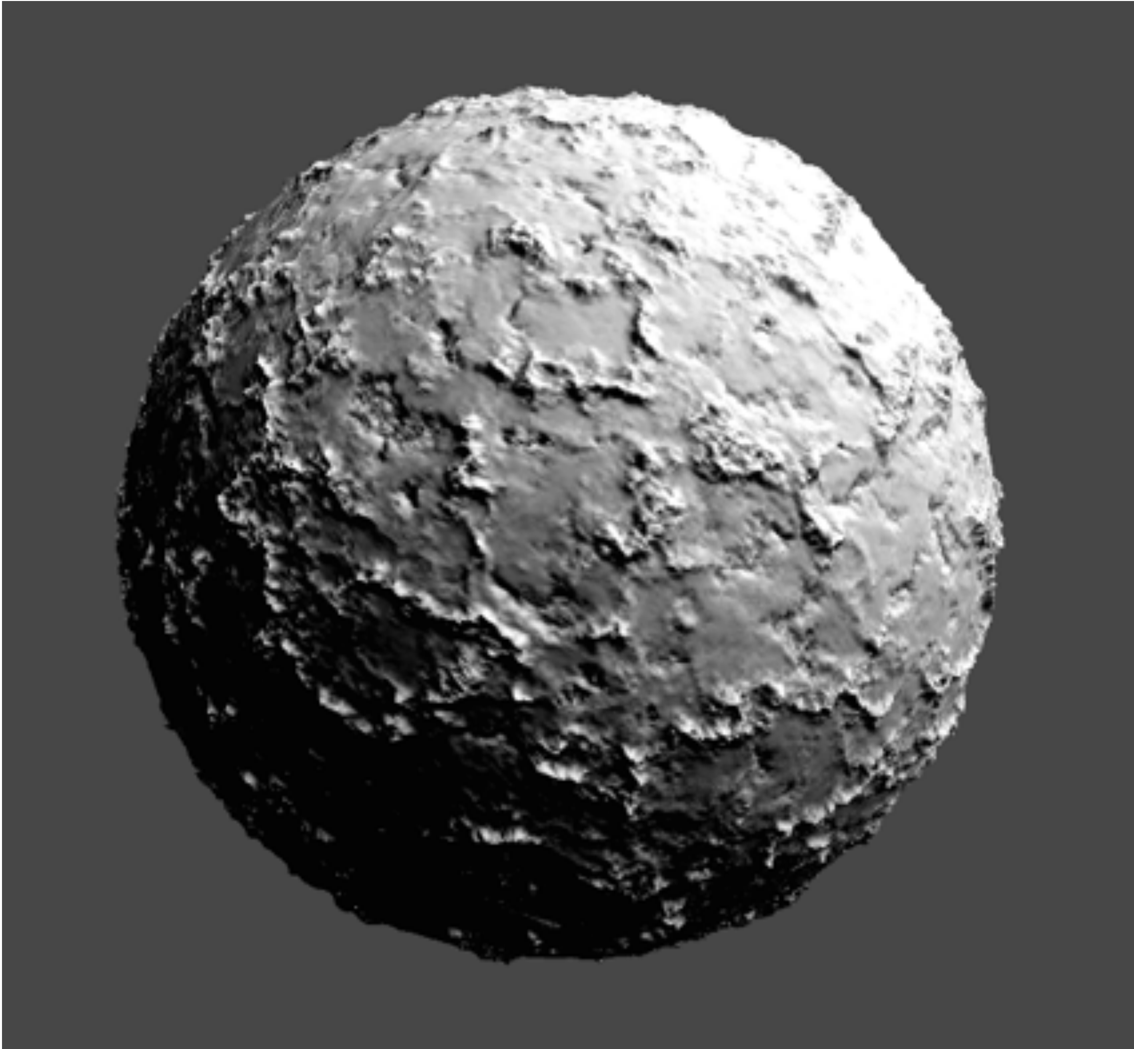


IMAGE 10. Advanced coherent noise function applied on the surface of a quad sphere.

4. PLANETARY TERRAIN

4.1.Requirements

Generating and rendering terrain on a planetary scale poses a new set of technical challenges. The scale can be massive and the spherical shape requires use of more complex vector mathematics than a regular square terrain.

The planet must also wrap on all directions and not have any abrupt discontinuities. The tessellation must be uniform to provide equal amounts of detail from the equator to the poles.

Additional requirement is to have a dynamic level of detail system which generates more detail as the camera approaches the surface and to create an visual node-editor which enables intuitive creation of planet surface shapes.

4.2.Tools

This project required a visual editor interface to allow intuitive designing of the noise functions which make up the planet surface form, multithreaded runtime calculation capabilities to generate the surface meshes and a robust realtime 3D graphics engine to render the final models.

Therefore the technical requirements to even start with the actual planet specific tasks are quite high. The options are to program a 3D graphics engine from the scratch or use an existing engine. The best choice considering productivity is to use an existing engine which allows the development resources to be targeted towards the aim of the project instead of the supporting technologies.

4.2.1. Unity

The project was developed using the Unity-engine (28) which provided a solid foundation to build upon.

Unity is a fully featured game engine. It consist of an unified editor which works as the interface for the developer to use the engine to develop their games. The main features provided by Unity include C# .NET functionality in the form of the

open source Mono-platform, 3D rendering capabilities with advanced shader models, asset pipeline, multi-platform support with input handling and an extendable visual editor.

Programming was done using the MonoDevelop development environment bundled with Unity.

4.3. System architecture

The planet generator is formed of three main parts: the node-editor, the planet mesh generator and the runtime dynamic level of detail (LOD) system (DIAGRAM 1). The system itself is built on top of the Unity engine and the Mono-runtime built-in Unity.

The node-editors main function is to provide the ability to create node-graphs and save/load them from the hard-drive.

The planet mesh generator creates surface meshes based on the node-graphs and can be run both from the editor and the runtime application.

The runtime dynamic LOD-system tracks the distance of each surface to the viewer object and triggers further subdivision of surfaces and mesh generation to provide more detail where it is needed.

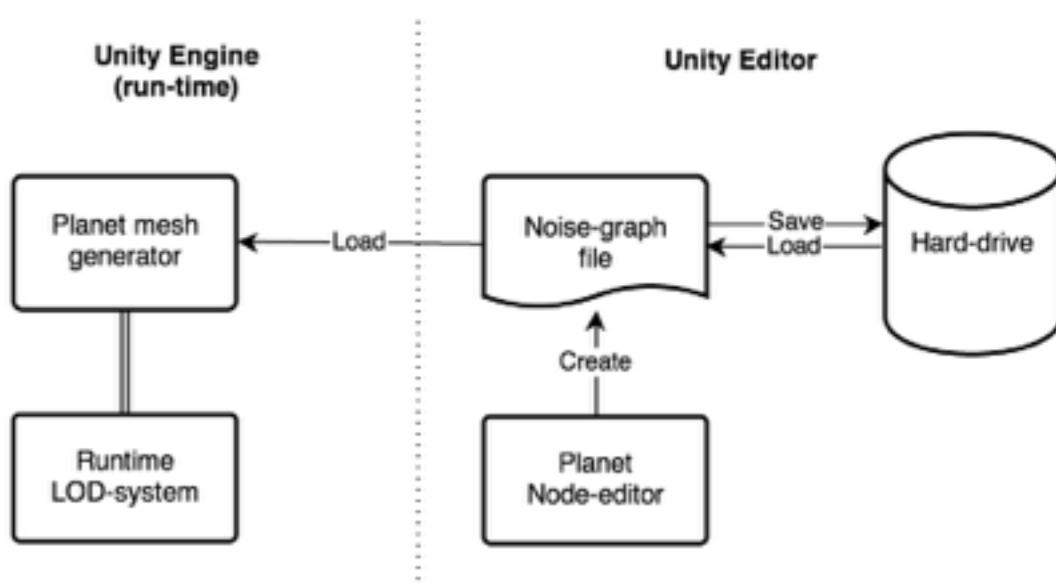


DIAGRAM 1. Planetary Terrain architecture.

4.4. Node-editor

The node-editor enables visual and more intuitive way to handle the design of the noise functions (IMAGE 11). The idea of the node-based solution was influenced by an open-source noise library called LibNoise.

There are three main types of nodes:

1. generators which provide an output value,
2. operators which manipulate an incoming value from another node
3. and an output node where the final output is displayed.

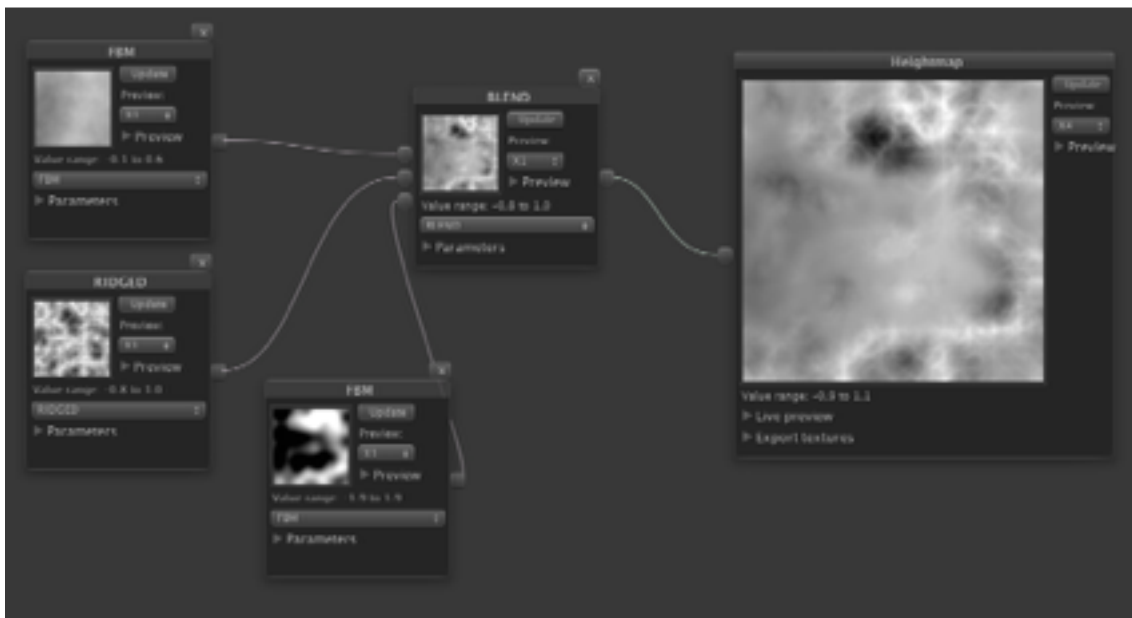


IMAGE 11. A node graph with multiple generators being blended together to form the final height map on the right.

The most important part of each of the nodes is a function which takes a 3-dimensional coordinate as a parameter and returns a floating point value which represents the height of the terrain at that position. The values are within -1 to 1 range.

float GetValue(Vector3 position); *FORMULA 2*

Generators (TABLE 1) provide a value based on a mathematical formula using the 3D position as the source, in this case Perlin noise.

Operators (TABLE 2) take output values from generators or another operators and transform those values based on a formula or logical operation. Examples of these operations include basic arithmetic operations such as addition, subtraction and multiplication, and more advanced operations such as blending multiple input values.

TABLE 1. Types of generator nodes

Node	**Description**
FBM	Fractal brownian motion based on 3-dimensional Perlin-noise.
Ridged	Variation of FBM which provide ridge-like terrain features.
Billow	Variation of FBM which provides negative ridges.
Const	Constant value defined by the user.

TABLE 2. Types of operator nodes

Node	**Description**
ABS	Returns absolute value of the input
ADD	Performs arithmetic addition of two input values
BLEND	Combines two input values based on the third which acts as a weight between the two.
CLAMP	Forces input value to be between given minimum and maximum
EXPONENT	Raises input value to power of value between -10 and 10
INVERT	Switches the sign of the value
MAX	Selects the larger value of two inputs
MIN	Selects the smaller value of two inputs
MULTIPLY	Performs arithmetic multiplication of two input values
POWER	Raises first input value to the power of the second value
SUBSTRACT	Performs arithmetic subtraction of two input values
TERRACE	Weights values between given range to come together at target value.

TRANSLATE	Moves the position parameter therefore changing the coordinate where the position is sampled.
DIVIDE	Performs arithmetic division of two input values
CURVE	Modifies the input value based on a user defined curve function.
WEIGHT	Moves input value towards target value at adjustable strength.
WARP	Performs 3-dimensional linear interpolation between the position vector and its normal vector, using the second input as a weight.
SELECT	Returns how close the input value is to a target value within given range.

4.5. Planet mesh generator

The planet mesh generator is provided with a node-graph file and parameters that define the resulting 3D model. These parameters include the radius of the planet, resolution of each mesh patch, height variation which defines how much the noise will displace the surface and a random seed value which is used to initialise the pseudo-random noise generators. Configuring these parameters allows for large variety of planets (IMAGE 12).

The mesh generation procedure creates an array of points in 3D space for each surface in the form of a cube, transforms the points to a sphere surface, applies the noise displacement and finally scales the points to desired planet size.

The generator will instantiate the initial six surfaces which can then be further subdivided by the dynamic LOD-system.

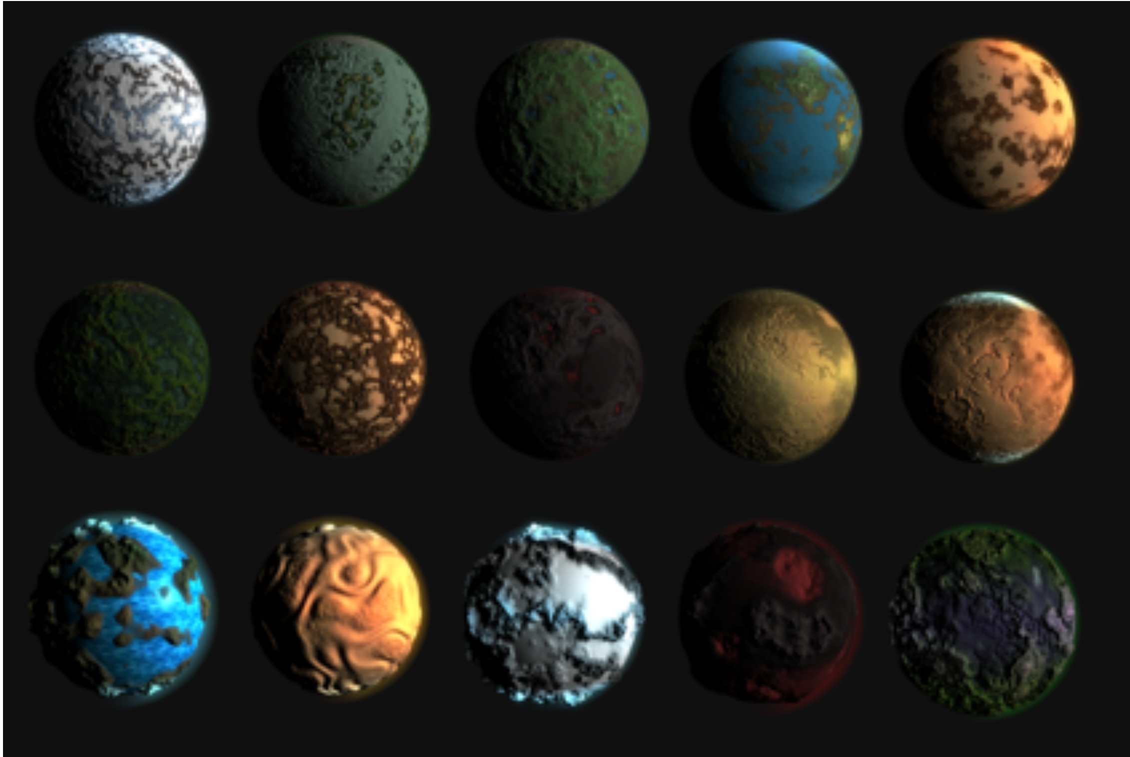


IMAGE 12. Planet meshes of varying scales generated using Planetary Terrain.

4.6. Runtime dynamic level of detail

The type of level of detail used is known as chunked LOD (29).

The LOD-system tracks an object in the 3D-world which it interprets as the viewer. On each frame when the world is updated the LOD-system calculates the distances between the viewer and the closest corner of each of the existing surfaces. These distances are then used to determine whether the surfaces need to be further subdivided to provide more detail or to be destroyed if they are unnecessarily detailed (IMAGE 12).

The LOD-system has a subdivision queue which keeps track of the surfaces that need to be subdivided. In order to avoid massive calculation spikes the queue is selectively emptied one at a time from the top of the queue. When a surface is selected it will be assigned a separate processing thread where the noise calculations and mesh generation happen asynchronously. When the operation is complete the thread is freed and another surface may be selected for subdivision.

The number of LOD-levels are defined by the user in the editor along with appropriate loading distances.

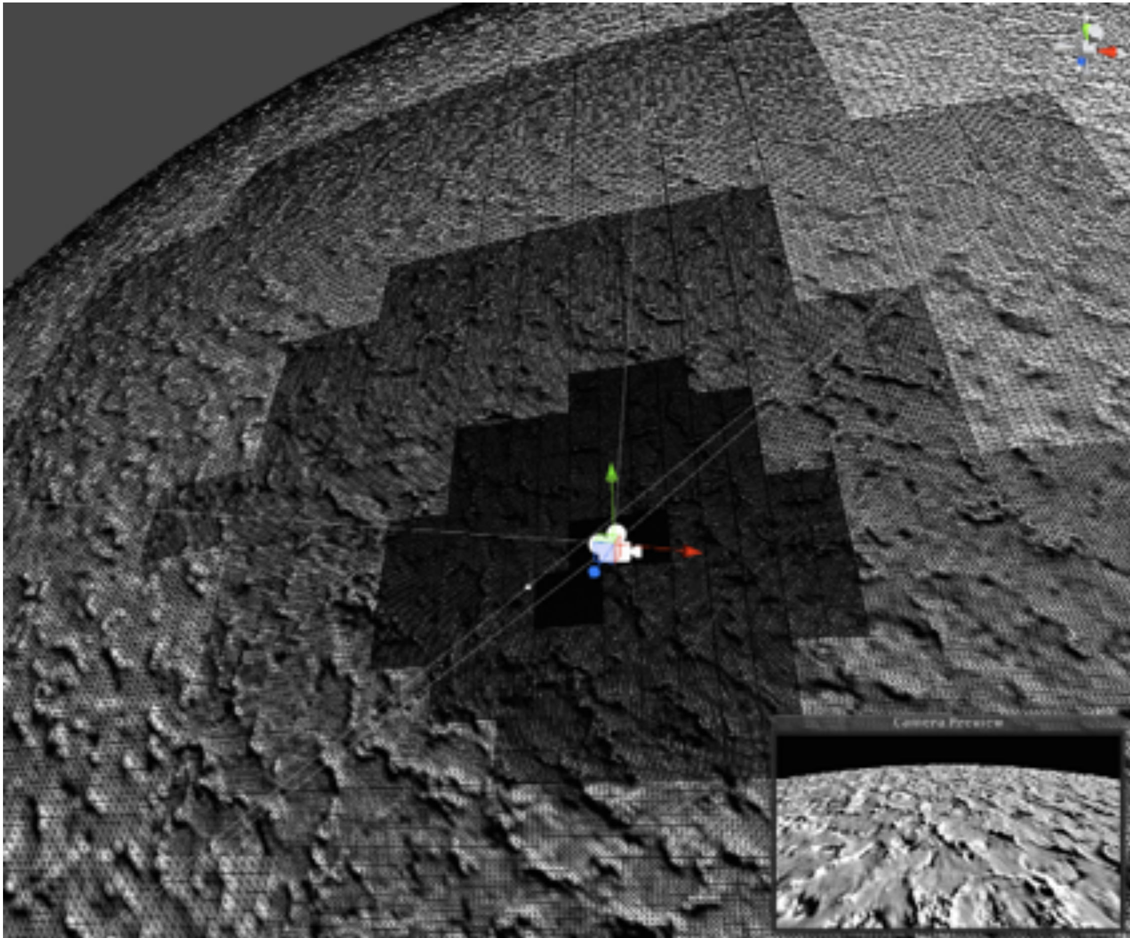


IMAGE 12. The surface generates more detail as the camera gets closer.

4.7. Shading

Shading the newly generated meshes properly is important part of getting a visually pleasing end result.

Simple approach is to use the noise value or height at each location and select a colour using a gradient (IMAGE 13). To get more variation the distance from the poles can be used to create specific effects for arctic and equator regions. Also the slope steepness can be used to control colours for cliffs and flat areas.

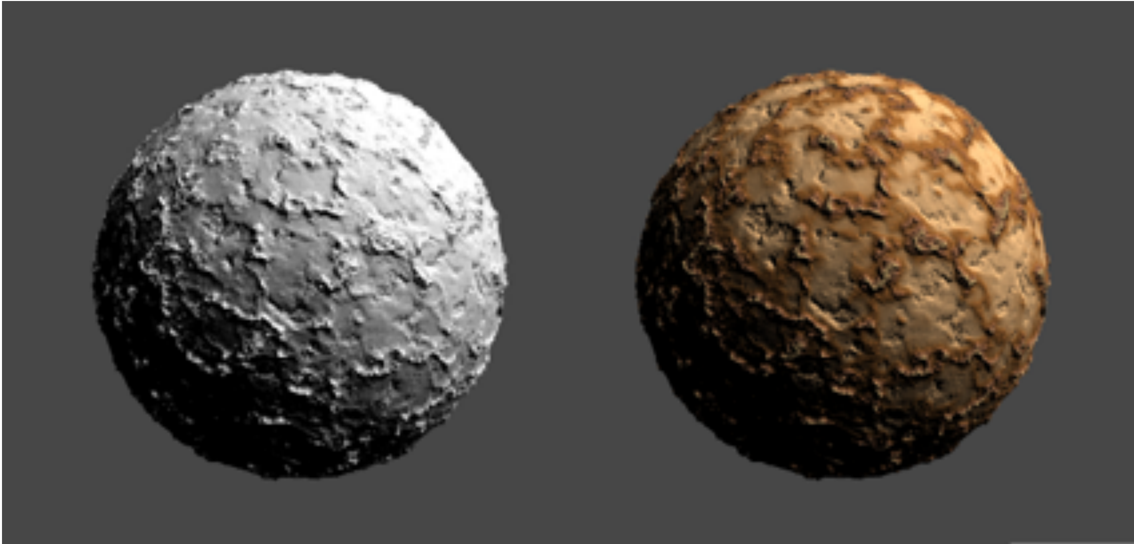


IMAGE 13. Colouring based on a colour selected from a gradient based on the height.

Colour in itself can be sufficient for viewing from large distances, but is not enough if the camera is positioned near the surface. The next step is to add textures using the same parameters of height, distance from poles and steepness.

Multiple shaders were created: height blending diffuse shader with 4 textures, bump-mapped shader with 4 diffuse and 4 normal map textures, versions of previous shaders with slope texturing and also versions with additional global colour map.

Additional nodes were also added to the node-editor in order to allow for more complex colourings of the surface (IMAGE 14).

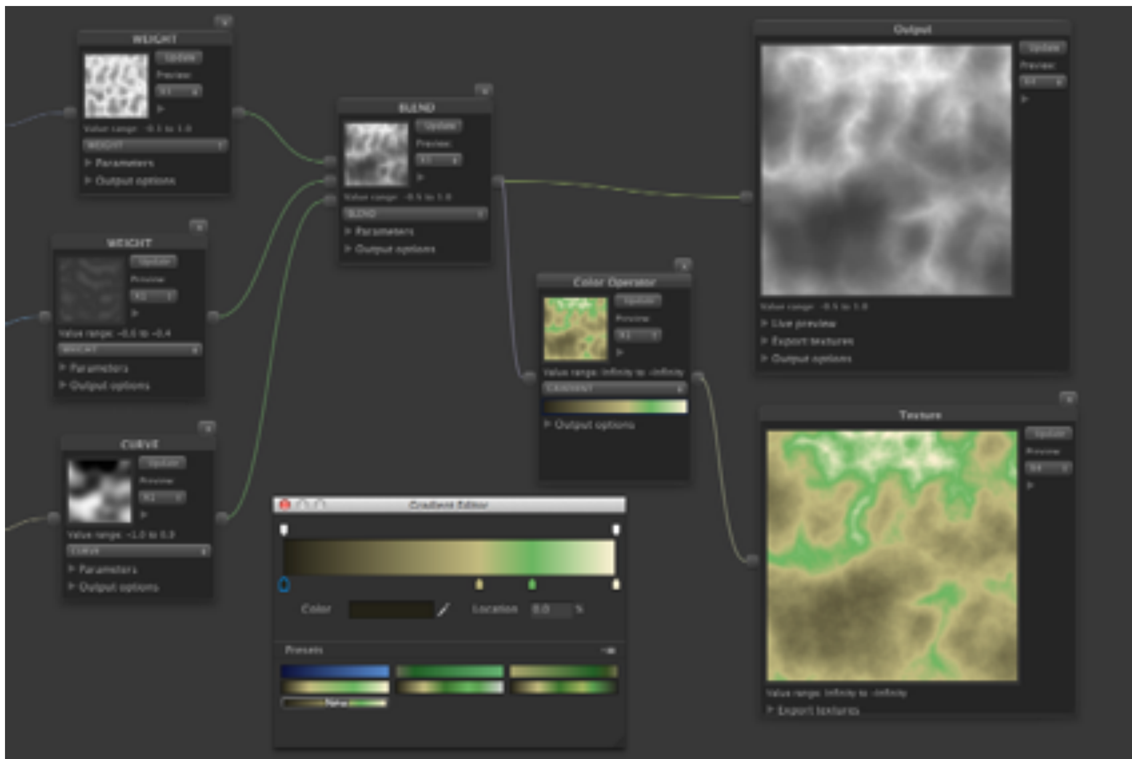


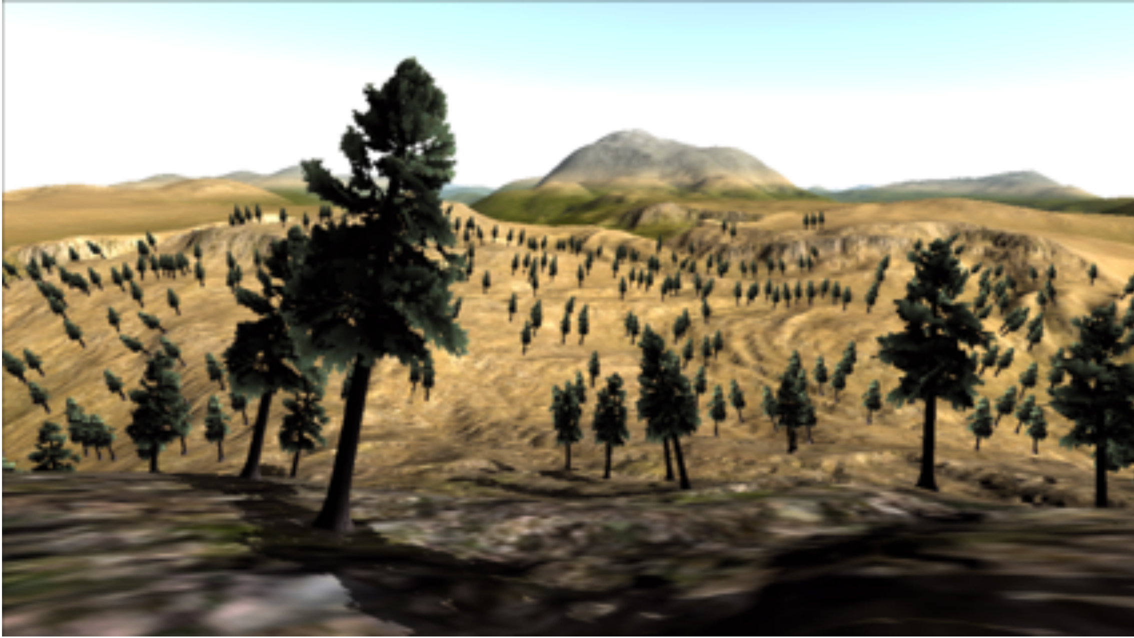
IMAGE 14. Colouring in the node-editor with a gradient-node.

4.8. Vegetation

Earth-like terrains require vegetation in order to appear natural. The problem with vegetation rendering often is performance, since the amount of separate trees and grass patches can get very high especially if the camera is viewing a large area. Optimisations included circumventing the Unity game object system and directly calling the DrawMesh-function to draw vegetation instances to the screen, and creating an impostor-system which replaces distant trees with a separate low polygon mesh (IMAGE 15).

The vegetation placement is based on the same noise module system as the terrain generation and also uses the terrains height, slope polarity values to se-

lect positions where the objects are placed.



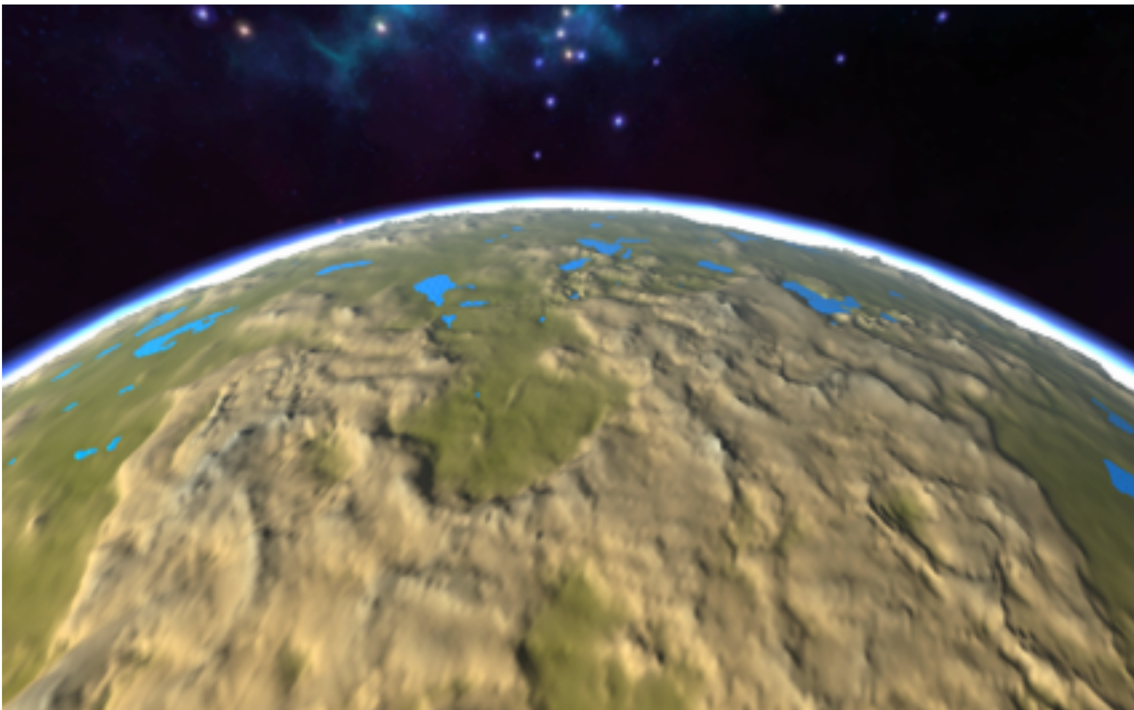


IMAGE 15. Trees on the planet surface in “Massive”-demo (30) included in Planetary Terrain (upper image) and view of the same planet from the orbit (lower).

5.CONCLUSION

This project had two main goals: to produce a procedural planet generator suitable to be used in game development and to create an intuitive editor for designing the planet surfaces. The procedural mesh generation was a familiar subject already, but in smaller scope and without dynamic level of detail, and the project managed to provide a suitable generic solution. The editor side was new territory but the Unity editor proved to be fairly easily extendable and facilitated the node-editor very well.

One of the bigger problems was the scope of the project as the project aim was to provide procedural planets suitable for game development, but did not further specify the type of the games. Games have massively varying needs for the planets: some games require small planets viewed only from the orbit and some require enough detail on the surface to facilitate realistic first person view. A project scope this broad runs the risk of providing overly generic solution that is not perfect for any specific need.

Overall the project was successful but during the development many new direction and improvement ideas arose.

Possible improvements for the project:

1. Move to noise generation task to graphics processing unit (GPU) by using modern shading languages. Generating noise on the central processing unit (CPU) forms a bottleneck to the data transfer speed between CPU and GPU. Generating noise directly on the GPU would eliminate the need to transfer the data in the first place.

2. Consider switching to icosahedron based sphere tessellation for more uniform tile size.

3. Further abstract the noise generation with terrain type or biome system.

REFERENCES

1. Unity Asset Store: Planetary Terrain. 2014. Available at: https://www.asset-store.unity3d.com/en/#!/content/13418

2. What is Procedural Content Generation? Mario on the borderline. Togelius, J., Kastbjerg, E., Schedl, D. & Yannakakis, G.N. 2011. http://julian.togelius.-com/Togelius2011What.pdf

3. Procedural Content Generation in Games: A Textbook and an Overview of Current Research: Chapter 1. Togelius, J., Shaker, N., Nelson, M.J. Springer 2014. Available at: http://pcgbook.com/wp-content/uploads/chap-ter01.pdf

4. Procedural generation. Wikipedia. 2014. Available at: http://en.wikipedia.org/wiki/Procedural_generation

5. Tiny Wings. Illiger, A. 2014. Available at: http://www.andreasilliger.com/

6. Procedural terrain generation in Sid Meier's Civilization V. Kloetzli, J. Firaxis 2013. Available at: http://www.firaxis.com/?/blog/single/procedural-terrain-generation-in-sid-meiers-civilization-v

7. Minecraft. Mojang 2014. Available at: https://minecraft.net/game

8. Generating random fractal terrain. Matrz, P. 1996. Available at: http://www.gameprogrammer.com/fractal.html

9. Noise and turbulence. Perlin, K. 2014. Available at: http://mrl.nyu.edu/~per-lin/doc/oscar.html

10. Creating spherical planets. Compton, K., Grieve, J., Goldman, E., Quigley, O., Stratton, C.,Todd, E., Willmott, A. Maxis, Electronic Arts. Presented at SIGGRAPH 2007. Available at: http://www.andrewwillmott.com/s2007

11. Kerbal Space Program. Squad 2011-2014. Available at: https://kerbalspace-program.com/

12. I-Novae Engine. I-Novae Studios 2014. Available at: https://www.inovaestu-dios.com/Technology

13. My First Planet. Peterson, A.C. 2008. Available at: http://alexcpeterson.com/

14. A Real-Time Procedural Universe, Part One: Generating Planetary Bodies. O'Neil, S. Gamasutra 2001. Available at: http://www.gamasutra.com/view/feature/131507/a_realtime_procedural_universe_.php

15. Making Worlds. Wittens, S. 2009. Available at: http://acko.net/blog/making-worlds-introduction/

16. Spore Planets Wallpaper. MC2009. Deviant art 2007-2014. Available at: http://mc2009.deviantart.com/art/SPORE-Planets-Wallpaper-52004157

17. Pseudorandom number generator. Wikipedia. 2014. Available at: http://en.wikipedia.org/wiki/Pseudorandom_number_generator

18. Coherent Noise. Bevins, J. 2005. LibNoise. Available at: http://libnoise.-sourceforge.net/glossary/index.html#coherentnoise

19. Generating coherent noise. Bevins, J. 2005. LibNoise. Available at: http://libnoise.sourceforge.net/noisegen/index.html

20. The Perlin noise math FAQ. Zucker, M. 2001. Available at: http://webstaff.it-n.liu.se/~stegu/TNM022-2005/perlinnoiselinks/perlin-noise-math-faq.html

21. Making Noise. Perlin, K. 1999. Presentation at GDCHardCore. Available at: http://www.noisemachine.com/talk1/3.html

22. Improving noise. Perlin, K. New York University. SIGGRAPH 2002. Available at: http://mrl.nyu.edu/~perlin/paper445.pdf,

23. Self-similarity. Wikipedia 2014. Available at: http://en.wikipedia.org/wiki/Self-similarity

24. Making Worlds 1: Of Spheres and Cubes. Wittens, S. 2009. Available at: http://acko.net/blog/making-worlds-1-of-spheres-and-cubes/

25. Mapping a cube to a sphere. Nowell, P. Math Proofs 2005. Available at: http://mathproofs.blogspot.fi/2005/07/mapping-cube-to-sphere.html

26. Heightmap. Wikipedia 2014. Available at: http://en.wikipedia.org/wiki/Heightmap

27. Procedural Content Generation in Games: A Textbook and an Overview of Current Research: Chapter 4. Togelius, J., Shaker, N., Nelson, M.J. Springer 2014. Available at: http://pcgbook.com/wp-content/uploads/chapter04.pdf

28. Unity. Unity Technologies 2014. Available at: http://unity3d.com/

29. Rendering massive terrains using chunked level of detail control. Ulrich, T. 2002. Available at: http://tulrich.com/geekstuff/sig-notes.pdf

30. Massive demo. Planetary Terrain 2014. Available at: http://planetaryterrain.com/unity/WebMassive.html