mADE
my Academy for Digital Entertainment

NHTV Breda University of Applied Sciences

PROCEDURAL PLANETS INTO DETAIL

TWAN DE GRAAF

092002

VISUAL ARTIST

A dissertation submitted in partial fulfillment of the requirements for the Bachelor Degree in 'Engineering in Game Architecture and Design'.

The Academy of Digital Entertainment.

Breda University of Applied Sciences

Supervisor's name:

Kim Goossens

Block A - B / 2012-2013

&

Block C - D / 2012-2013

Submitted on:

31-05-2013

ABSTRACT

Figure 1: Resulting planet from afar and up close.

In this paper steps are laid out to create detailed procedurally generated worlds, intended to be used in videogames. The generator combines methods used in the past with pioneering methods. The product of this research project is a tool to generate data for the visualization of three-dimensional planets, from low to very high detail.

Contrary to most applications that are used to visualize planets, this generator separates the generation and visualization. This way the visualization system could be practically implemented into any game engine, but does not require the generation algorithm to be written for each separate engine, saving development time in the long run.

One of the aims is to give artistic control over the outcome. Procedural generation is often mistaken for random generation. While it uses a lot of randomly generated elements, the procedure makes sure the different elements work well together and form a governed, coherent structure. The power of procedural generation lies at and beyond the point where the need for ease outweighs the need for control. It creates the rest of the world after the artist has set his or her guidelines. This way the procedure could theoretically use these guidelines to create more and more detailed versions of the planet, without the need of intervention of the artist after the initial look is decided upon.

The procedure is controlled using a combination of vertex paint, bitmaps, sliders and gradients. This way it is easy to learn using the tool, even without prior knowledge of Houdini, the software that is used to create and run the generator. Houdini is being developed by Side-Effects since 1987.

The sliders can be used to create different kinds of generic planets. Vertex paint gives the artist more control over the different terrain types across the planet. Bitmaps give high control over small areas, enabling very specific terrain features to be projected onto the planet's surface. This helps to on one hand create realistic planets when using real elevation data. On the other hand bitmaps created by artists can make for interesting fantasy planets.

TABLE OF CONTENTS

INTRODUCTION

Each year people expect more from video game visuals. The industry tries to keep up with larger budgets and employing more people. Somewhere however is going to be a cap, where employing a larger budget is no longer going to be profitable. This is already starting to happen, even large returns do not always make up for the bloated budgets anymore. The industry is trying to come up with solutions for this such as using third party engines and outsourcing. Another solution is procedural generation. Procedural generation is a completely different approach to creating content. Instead of creating object after object manually, procedural content is content generated automatically by using rules and guidelines. Usually this approach takes more time up front, but saves a lot of time in the long run. This however does not exclude small projects at all.

Figure 2: SIP, a game I created with 5 other developers in roughly 48 hours, using a procedural level generator. (SIP team, 2012)

The strength of procedural generation lies in taking over tedious, repetitive tasks. An artist can make the important decisions and the procedure can fill in the details. While procedural generation is already widely used in film productions, the video game industry is still adapting to this mentality. Even though the concept itself not new for video games, Elite was created in 1984 by David Braben and Ian Bell. This game relied heavily on procedural generation to create a vast universe for the time.

With procedural generation a game can feature significantly more content, more cheaply. This is one of the main reasons for developers to consider using procedural generation alongside manual generation. Besides being personally interested in the topic, the way procedural generation may change the game industry is one of the main reasons for writing this paper.

PREVIOUS WORK

A game that heavily influenced this research is Spore, released by Maxis, Electronic Arts in 2008. It generates a practically infinite amount of planets to visit and edit. The planet generator is built into their game engine, which has benefits, such as giving real time control and being a lot faster, and taking less disk space. On the other hand it makes reusing the work on a different engine difficult and artists do not have direct control over the look. Instead they authored all the possible looks and let generator do the rest, which was of course fine for what was needed in Spore's case.

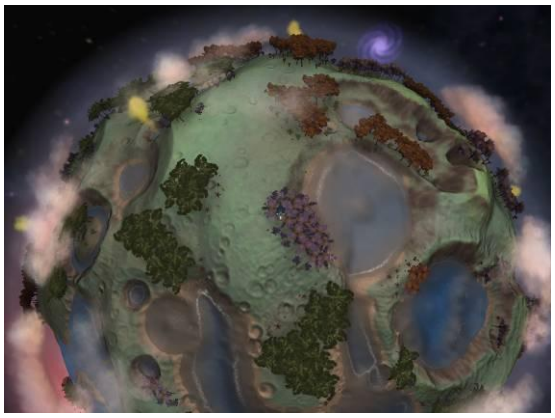


Figure 3: Spore, planet from space

Figure 4: Spore, planet close up (EA, 2008)

With my procedure I employed useful tricks I took from Spore, but in a way that is more suited for a more limited amount of planets, that require more control. Additionally my procedure is capable of creating more detailed terrain and artists can use 3-dimensional deformation, in comparison to the height-map-like deformation used in Spore.

Other games I took inspiration from on a more aesthetic level are:

Figure 5:Planetside 2 (Sony Online Entertainment, 2012)

Figure 6: Kerbal Space Program (Squad, 2012)

RESEARCH GOAL

The goal of this research is to find a way to create an industry-standard planet generator. It should be able to create a world suitable for a wide range of videogame genres or simulations. Focus points are: Control, Visual quality and Ease of use. The generator should be capable of more than just a height-map-terrain.

PLANET CREATION

This chapter will show the decisions that were made, why they were made and how they were executed.

SPHEROIDS

The starting point for creating the procedure was deciding what structure or more specifically, edge layout would serve best for this procedure. The two most apparent methods were tested: Icosahedron- spheres and spherified-cubes. Longitude-latitude-spheres proved to be unusable at the very start, due to the way polygons of those spheres are stretched too much at the poles in comparison to those at "the equator".

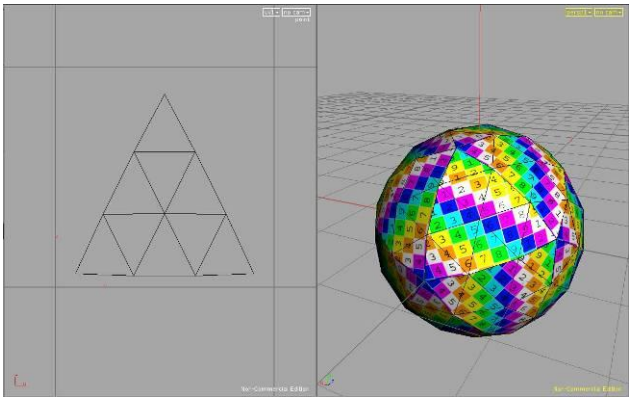


Figure 7: Icosahedron- sphere

The icosahedron-sphere has the least distortion of the three types of spheres. This comes at a cost: It exists of triangles instead of quads and it has 30 possible UV-seams. (See subchapter: Texture coordinates)

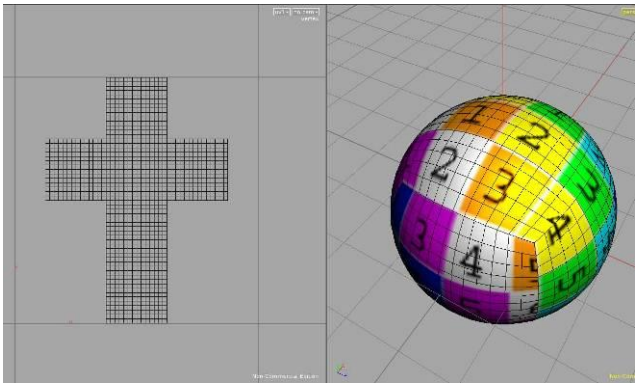


Figure 8: Spherified cube

The spherified cube has some minor distortion where the corners of the cube existed originally. The upside is that it can be entirely built up with quads. It also has only 12 possible UV-seams, compared to the Icosahedron, making it much easier to unfold it in a way that is easy to understand and without much additional UV-stretching.

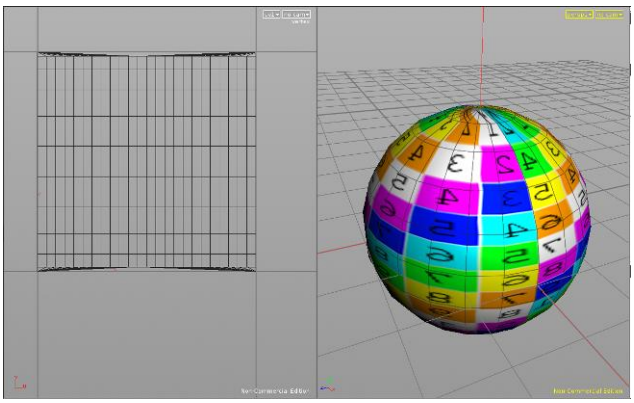


Figure 9: Longitude-latitude-spheres

Longitude-latitude-spheres are built up out of both triangles and quads, this results in varying polygon sizes across the latitude. It also creates a lot of UV-stretching, or illogical UV-layouts, whatever unwrap method is used. In the image, the sphere is unwrapped cylindrically.

From this quick analysis, the spherified-cube was of the most use and was used as a starting point for the rest of the project.

The procedure is not based on simple height map terrain. However height maps are utilized. Rather than loading in a single height map, multiple height maps can be used as brushes, projected onto the sphere. This method has a lot of benefits: it is more flexible, faster and combinations can be made. Using this method eliminates the need of creating or compositing height maps by hand, but still gives a lot more control compared to generators that exclusively use Perlin-like noise. (See subchapter: Terrain detailing, pure noise)

This is also one of the main features of Spore's (EA, 2008) planet generator. The generator that is built into that engine is able to quickly generate simple but good looking planets. Because Spore does this operation in the terrain shader, it is capable to edit and update this "height map" in real time, allowing for player interaction. Because of the calculation times, the rest of the terrain is relatively simple to keep the game running fluently.

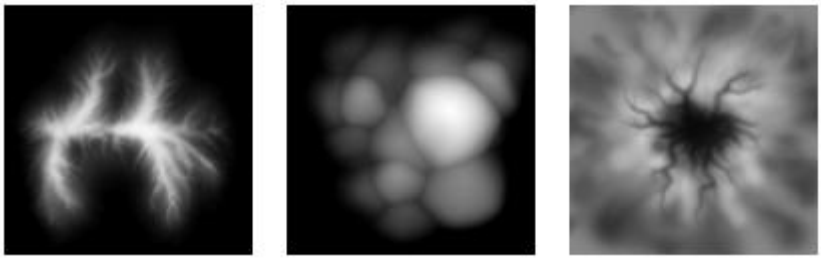


Figure 10: Three of the brushes used by Spore's System. (EA, 2008)

The generator described in this paper does not generate the terrain on the fly. This prevents adding direct player interaction on the scale of Spore. However, this approach does allow for more visual fidelity and more complex operations on the surface and sub surface, the latter being entirely impossible using the simplified system used in Spore.

Spore uses something similar to a particle system to place these brushes. It uses different versions of these systems to create single brushes, groups or streaks of brushes. This works fast and efficient for creating a lot of random planets. After that either the developers or players could paint over this with other brushes. This system also gives most of the control that is needed for this document's generator. Though more ways to control and edit the output of this system made it more versatile.

Figure 11: The first result after implementing the height-map-brush system and creating a simple terrain shader.

The brush placement system started with single brushes placed randomly on the surface. By using 2 different brushes, this gave the result on the last page. The brushes could be set to either lower the terrain or push it upwards. By simply choosing the right kind of brushes, an artist can influence the style of the planet.

Figure 12: A fantasy and a realistic height-map-brush were used to get the initial result. (Flower, 2011)

To improve on this, the placement of the brushes needed to be more coherent and structured, instead of placing each individual brush randomly or manually. To do this a climate simulation has been created.

CLIMATE GENERATION

The continents are random, yet easily adjustable. They are created by scattering points on the sphere and convert this to a polygonal solid using the "tetrahedralize" node in Houdini. It connects the points to form triangles. After that all the points are beveled and spherified to create polygons with 4 or more sides, which makes the connections between continents look better. After this, land is distinguished from sea randomly according to a ratio set by the user. The borders of each polygon or quasi continental plate are subdivided and noise is applied. Finally the entire surface is subdivided and spherified.

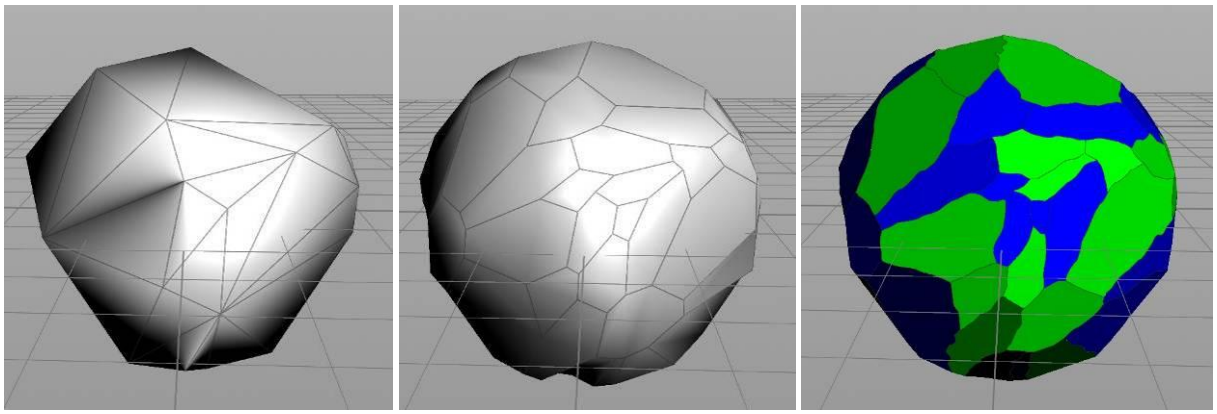
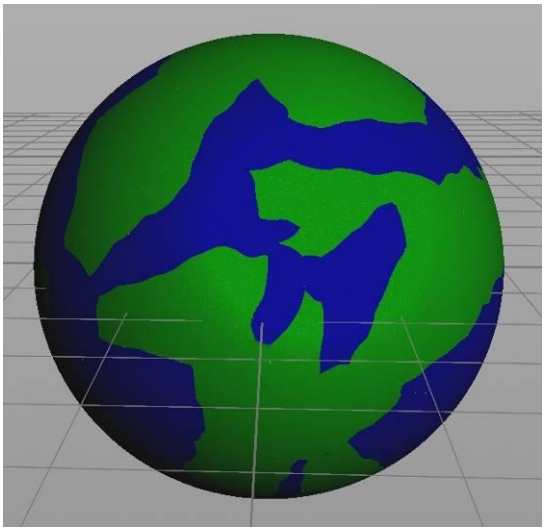


Figure 13: Polygonal solid, Beveled spherified solid & Solid with noisy border.



Using continental plates creates a realistic looking and visually appealing division between water and land. It can also be used to determine where mountain ranges and oceanic trenches could occur naturally. From this actual climates can be calculated using the latitude, distance to water and mountains. This gives artists a preset to work from.

Figure 14: Final Continental Base.

Most features can be controlled without having to manually edit them:

- The amount of continental plates.
- A seed for the overall topology of the plates,
- A ratio for land versus water,
- A seed for selecting the land and water plates

Besides these main settings, there are various settings to tweak the terrain, like the noise around the plate's borders.

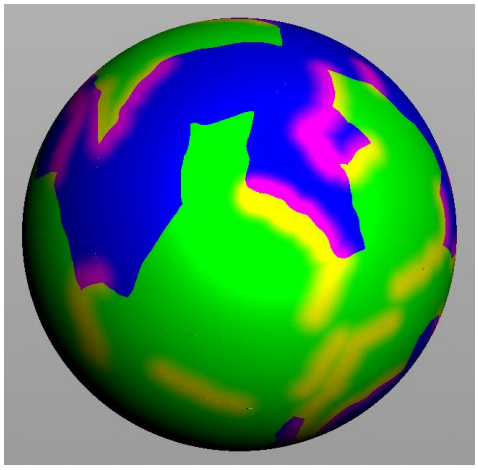


Figure 15: Mountains, trenches and ridges and.

From these continental base plates extra features can be calculated. The user can determine the likelihood that the continental plates create mountain ranges, oceanic trenches and oceanic ridges by "colliding", emulating the workings of Earth's tectonic plates according to the leading theories on this topic. (Wikipedia, 2013a)

With the land and water masses present as well as mountains, the rest of the climate can be generated. First a simple gradient is used to determine the base temperature across the latitude. After that a wind direction model is set, customizable over the latitude. (Wikipedia, 2013b)

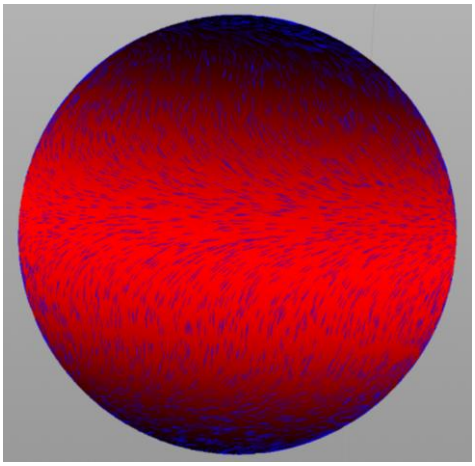
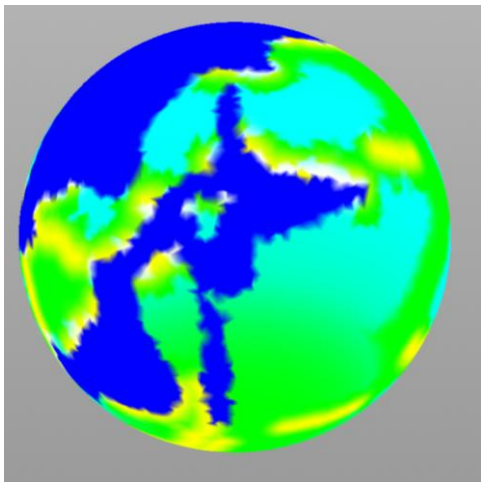


Figure 16: Base temperature and wind directions

For a more organic and correct climate a thermohaline circulation (Wikipedia, 2013c), or in other words, oceanic conveyor belt, is simulated. The upper currents of the conveyor belt are affected by the wind, the lower currents are disregarded as they do not affect the climate as much. The thermohaline circulation can make coastal regions at higher latitude relatively warm. For example Amsterdam has warmer winters on average than New York, which is at a lower latitude. The simulation is done by creating a path in the ocean following the wind direction. After that the temperature values are reversed and added to the result. The old values are subtracted to conform to the first rule of thermodynamics.

Figure 17: Thermohaline circulation

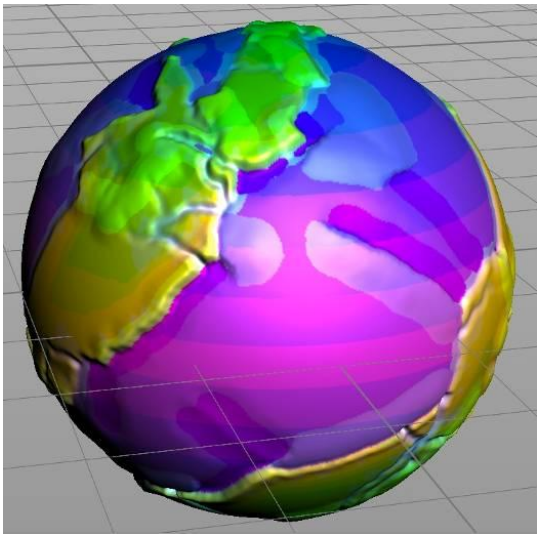


With the final temperature data and wind directions rainfall can be simulated. First the user has to set the maximum air moisture over the temperatures as air can hold a varying amount of water depending on the temperature. (Wikipedia, 2013d) After that, the ocean water is moved along the wind direction until the "clouds" reach a mountain or have traveled their maximum distance. This creates a rainfall map.

Figure 18: Rainfall map

From this rainfall map rivers can be generated. everywhere where rain falls "rain nodes" are created. Rivers start in areas with low rainfall. After that they travel from rain node to rain node in the general direction of the ocean. A river stops when it reaches either the ocean or another river. If a river cannot reach either, the river is discarded. The rivers get their width by adding up the rainfall of all rain nodes upstream. The rainfall and the rivers together determine the moisture levels of the climates on land.

Figure 19: Generated Rivers



When adding the placement of mountains, land, coast, oceanic ridges, the ocean floor and ocean trenches to the climate map, the final climate can be calculated. Instead of gradients the final climate is divided in steps. This makes the amount of climate types manageable. The user can define how many steps of temperature and moisture are needed. The height is always divided into 6 levels: oceanic trenches, ocean floor, oceanic ridges, coastal regions, main land and mountains.

Figure 20: Final, stepped climate with displaced geometry

This Climate map can be used to generate certain terrain features that can be intended for specific climate types. This includes height-map-brushes (See subchapter: Brush Placement System), terrain detailing (See subchapter: Terrain detailing, Texture based) and the placement of objects (See subchapter : Object placement)

In the stepped version of the climate on the last page there are 6 height levels, 7 temperature levels and 4 moisture levels. This equals 6*7*4 = 168 climate types. Dividing the climate like this makes it possible to give specific characteristics to each climate type. One of these possible settings is that certain height maps can be linked to each climate type. Each climate type can have one or more height maps, which can be used exclusively for that climate or a range of climate types. Adding no height map to a climate type simply means there are not going to be any height-map-deformations for that climate type.

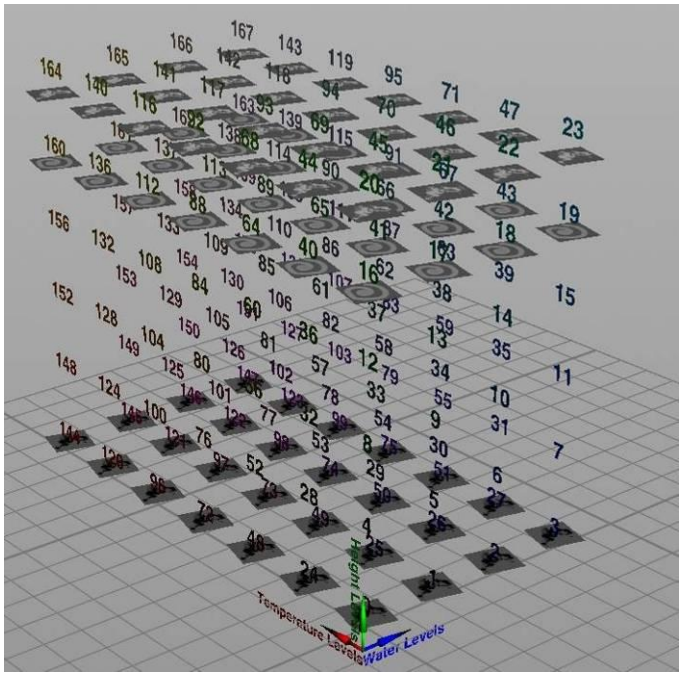
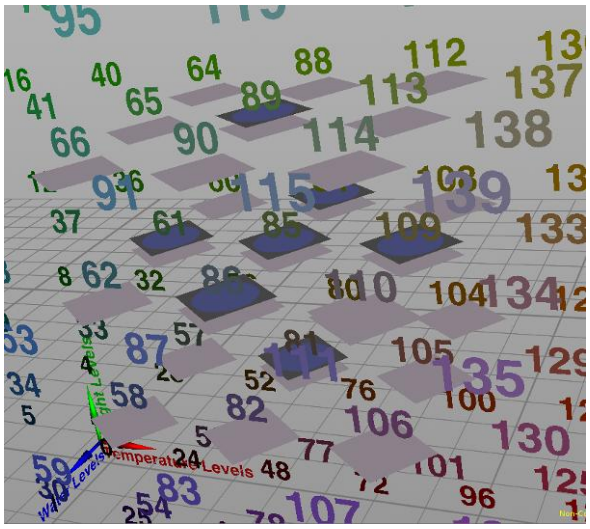


Figure 21: Brushes selected for terrain deformation

A system, as shown in the image, has been added to have an easy overview of what height-maps have been added to each climate type. The system uses a three-dimensional grid that conforms to the color values of the stepped climate. This means on the X-axis the different temperature levels are represented, the height levels are on the Y-axis, and the water levels are on the Z-axis.

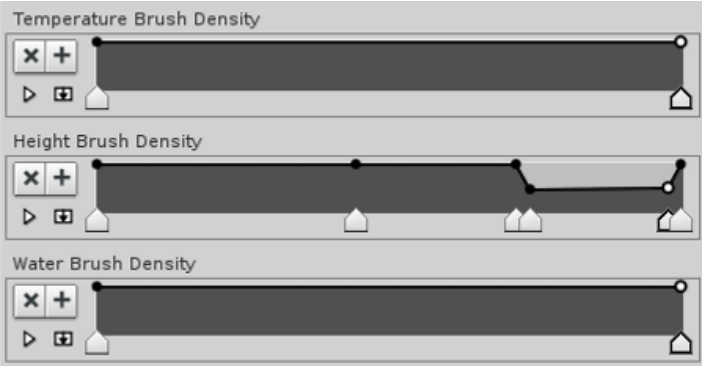
These height maps or terrain brushes can be added to this grid in three ways: A brush can be added to a single climate type, to a boxed range climates or a spherical range of climates.



These ranges adhere to the grid as well. First the middle point of a climate is defined, say 3,4,2 , from this middle point a range can be set to include other climate neighboring climate types as well. A range could be 1,1,0 as these ranges are three-dimensional as well. When set to a boxed range the climate types would go from 2,3,2 till 4,5,2; when set to a spherical range, the corners are simply left out. The system allows for five climate ranges to be set per brush, if that is insufficient, a brush can be simply added multiple times with an additional five ranges each time.

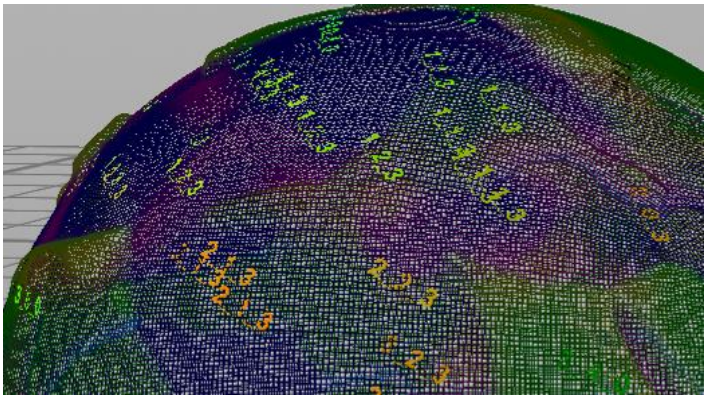
Figure 22: Boxed (gray) versus Spherical (blue) ranges, ranges here are 1 in each direction.

The height maps are scattered randomly on the planet's surface, on the appropriate climate type areas. The density of height maps can be determined by the user, as well as the seed. In addition to the total amount, the user is also able to determine the distribution of the amount of brushes among the climate types. This is done using three curves, one for each climate axis.



These three curves determine the density along the three climate axes. The height brush density curve has a dip near the end. This means there will be about half the amount of brushes with the height climate of 5 (out of 6), compared to the other height climates. Along the other axes the distribution is even.

Figure 23: Height map brush distribution along the three climate axis



To have a fast update each time the above curves are changed, the total density of the brushes or the seed is changed, a preview is added to get an idea where the brushes go. The numbers have been colored in the corresponding climate color.

Figure 24: Height Map Placement Preview

When the user is content with the settings, The height maps can be applied to the planet. This is done by placing the appropriate brushes on top of each intended brush spot. The height values are then transferred to the planet's mesh. The mesh can then be pushed up and down according to these values.

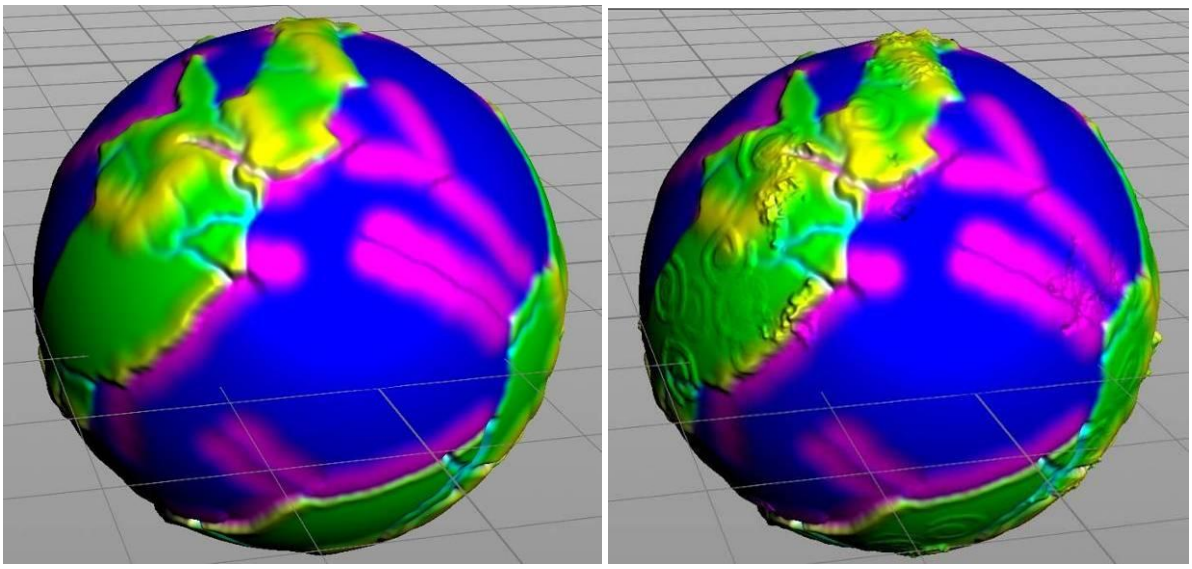


Figure 25: Before and after height map projection.

The generator needs to create worlds that look good, but they also need to run on a decent frame-rate. To make sure the GPU is capable of rendering the planets at a good frame-rate, the amount of triangles on screen must be within a certain range, along with texture sizes and shader complexity. One way to accomplish this, is making the amount of triangles per area dependent on the distance of the camera. This way areas nearby the camera get more triangles compared to those further away. These areas are called chunks. Chunks can be either 3d or quasi 3d chunks.

Spore (EA, 2008) most likely uses quasi 3d chunks. I say quasi 3d, as these chunks do exist in 3d space, but are basically 2d grids with height values, curved along the planet's surface.

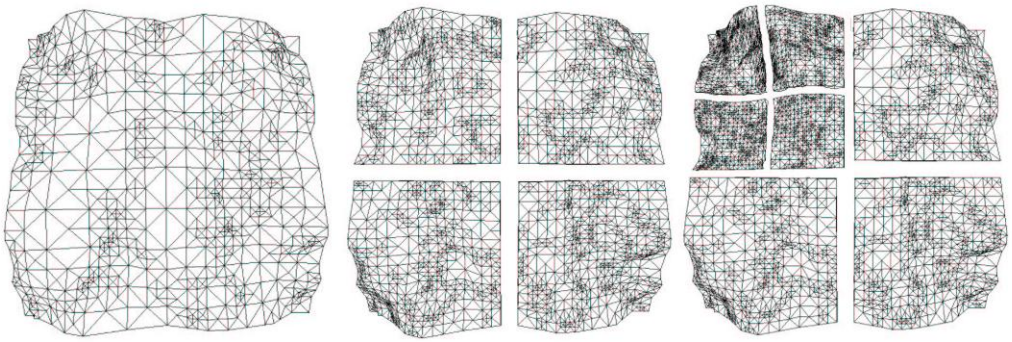


Figure 26: The polygons get more dense as the camera zooms in on the upper left hand corner. (Ulrich, 2002)

The generator uses chunks that are more true to actual 3d chunks, allowing for multiple heights on each original surface point. This can enrich the terrain with features like caverns and natural arches.

There were two possible ways to chunk a planet: A radial-polar approach or a Cartesian (or boxed) approach. A polar approach would be similar to the quasi 3d layout, picking arc segments of the planet's circumference. This would give chunks of each roughly equal size around the surface. However subsurface chunks would get smaller when closer to the center of the planet. They can also be either square or triangular, which makes things more complicated.

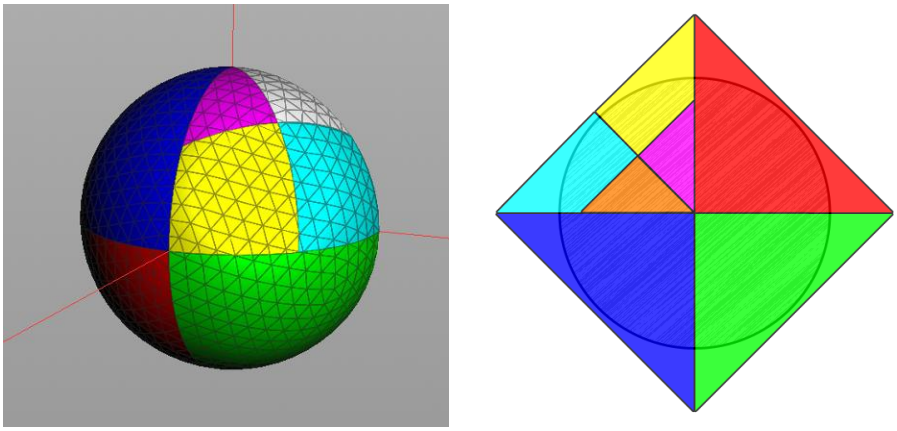
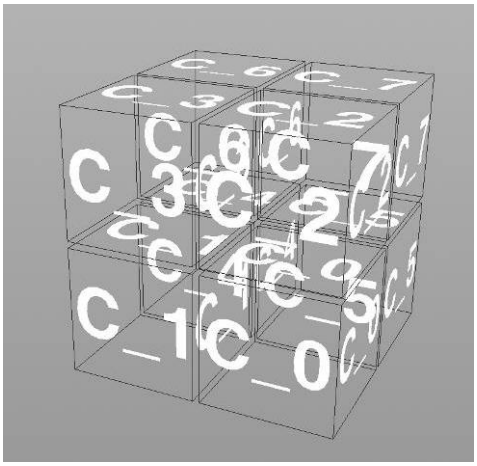


Figure 27: Polar segmentation and Radial segmentation. (cross-section)

Additionally, radial chunks are not very optimized for working with the voxel system used to create caverns and arches (See subchapter: Voxels). This is because the voxel system in Houdini works most efficient with box shaped objects. These were all reasons to instead choose a boxed, or Cartesian approach.

Using the Cartesian approach, instead of picking areas relative to the sphere, the chunks bounding boxes exist relative to the world axis. This basically gives the same chunks on the first subdivision level. On higher subdivision levels the amount of polygons per chunk can vary a lot and may even be empty altogether. The bounding boxes of the chunks however, are all the same within one subdivision level and the actual density of polygons is the same.

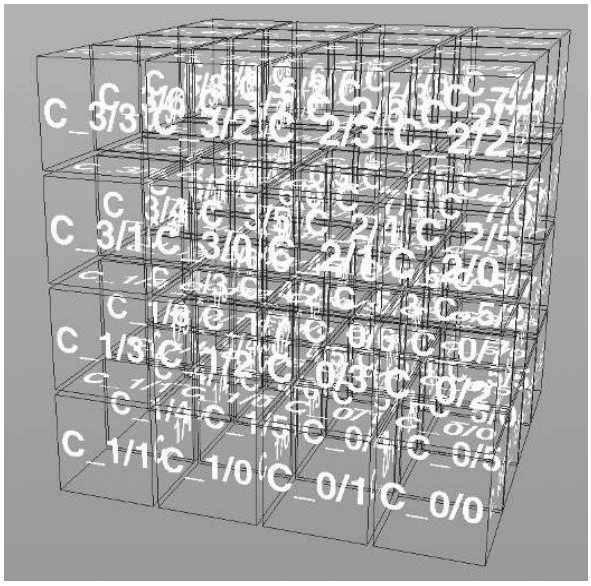


The Planet is divided in 8 equally sized cubes. Each time it is divided again, each chunk is divided in 8. This means the amount of chunks each time increases with the power of 8: 8^0 = 1 chunk, 8^1 = 8 chunks 8^2 = 64, 512, 4096, 32768 etc.

The amount of polygons that are available for all the chunks are the same, the volume of the chunks however gets smaller. This means the polygon density gets higher as the subdivision of the chunk gets higher.

Figure 28: Cartesian Chunks: Subdivision level 1.

The chunks in the procedure are hierarchal, this means they adhere to something like a folder structure. Each chunk has 8 "children" in the subdivision level above and 64 "grand children" in the subdivision above that. This information is useful to connect the chunks to each other in applications that make use of the output of the procedure. This data is given to the output files. So for instance the name of the files can be for instance: C_0, C_0_4 or C_7_0. This way an application or game engine can see that C_0_4 is a child of C_0, but C_7_0 is not. This information can be used to switch between the various subdivision levels of the chunks.



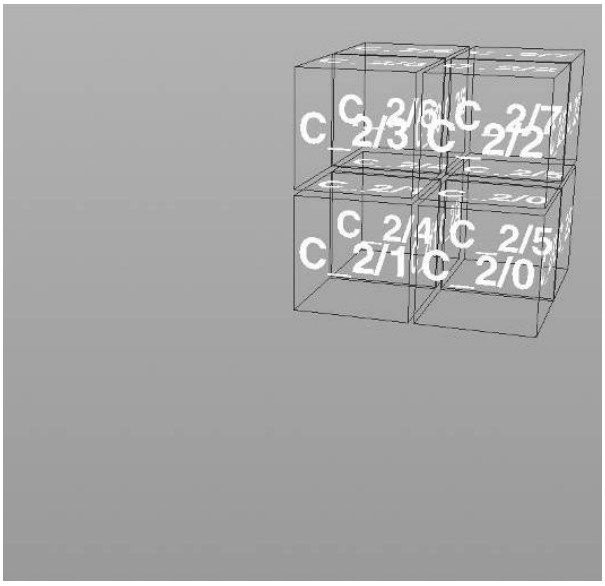


Figure 29: Cartesian Chunks: Subdivision lvl2 & Chunk 2 of Subdivision lvl1, divided in 8.

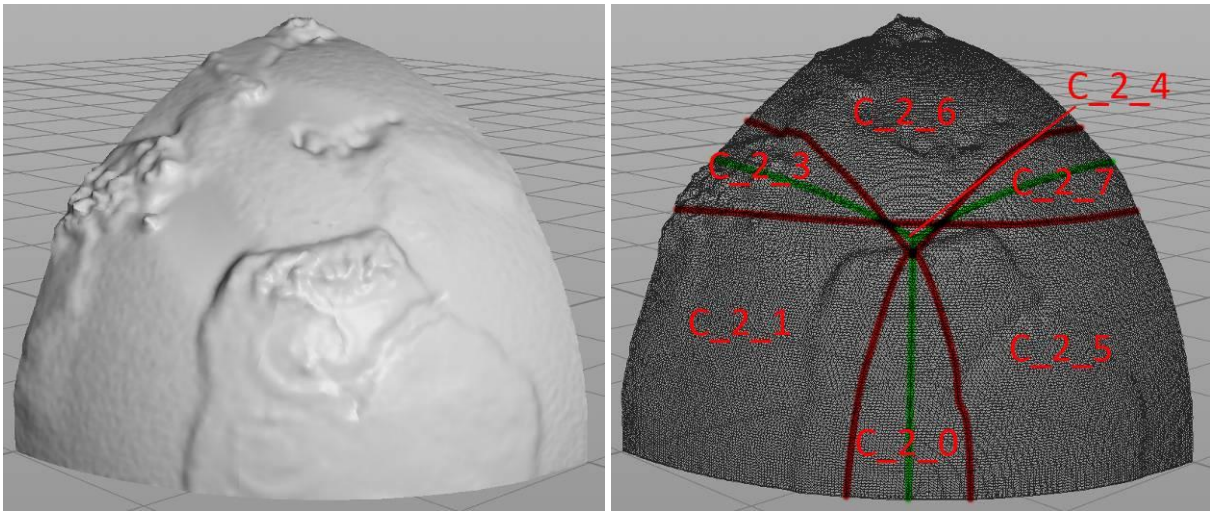


Figure 30: An example of one eighth of a planet, divided into 8 smaller chunks.
Note that chunk_2_2 is empty. The green border marks the UV border.

Because of the chunking and converting to and back from volumes (See subchapter: Voxels) the borders of each chunk may differ slightly, meaning they do not line up perfectly.

To solve this, the procedure has been set up in such a way that the calculation of each chunk is divided into multiple steps. First the basic geometry of each chunk is created with a volume conversion. After that, each chunk is loaded back in for the next step. In the second step each chunk is connected to all its neighbors and cut again. This fixes almost all of the gaps between the chunks. Additionally an extra polygon border is created along the UV border (See subchapter: Texture coordinates) This border is visible in figure 30b.

As a fallback the border of each chunk is also extruded down. This polygon strip is exported separately. This way it can be made invisible in a game or render engine. This is useful as at large scales and distances, the depth buffer of these engines may be too inaccurate to properly show these borders. As at these large distances small gaps will not be visible, the border strip can be made invisible to prevent visual artifacts.

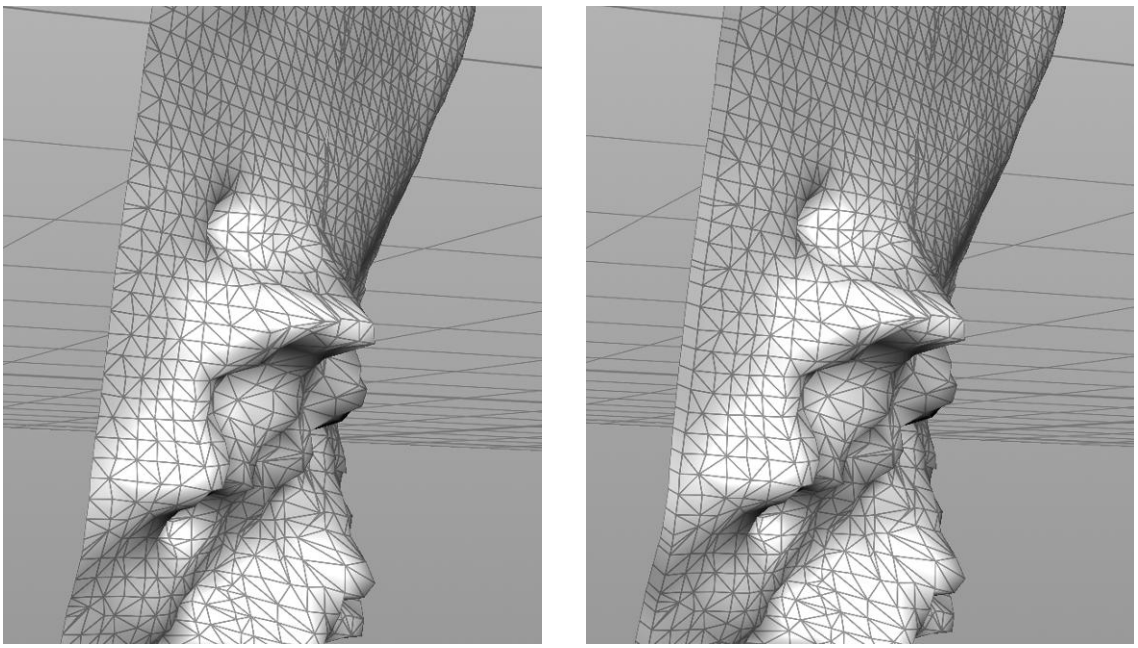


Figure 31: Polygon strip around the chunk borders

For the integration of three-dimensional deformations, like caves and natural arcs, Houdini volumes, or from Houdini 12.5, VDB volumes are used. These volumes use a form of voxels. Voxels can be described as points on a three-dimensional grid. These points can have all kinds of values. These values can be then used for all kind of representations.

Figure 32: An in-game view of Minecraft. (Speed, 2011)

Minecraft (Mojang, 2011) uses these values to represent a usable block in the world. The game is possibly the most well known application to use voxels. The way Minecraft stores data in different chunks also has similarities with the way this generator is "chunked" (See subchapter: Chunking, Cartesian)

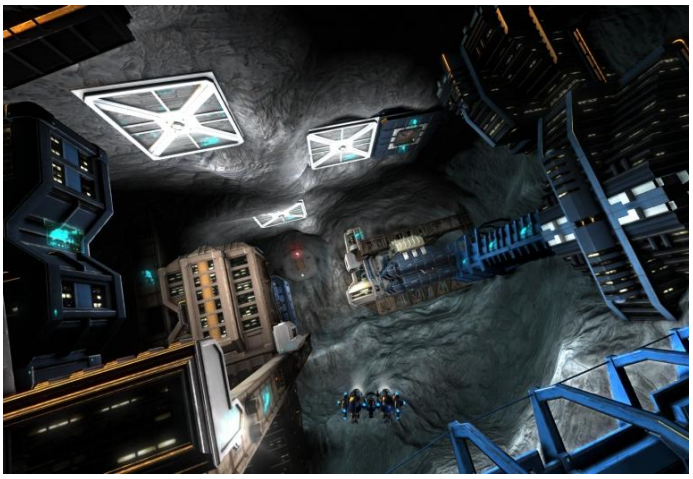


Figure 33: An in-game view of Miner wars 2018. (Keen Software House, 2012)

Miner wars 2081 (Keen Software House, 2012) Uses voxels to determine the borders of a polygonal mesh. This is similar to the approach taken in this document's world generator. Voxels with values above a certain number are inside the mesh, the rest are outside the mesh.

The procedure uses this voxel system for three purposes. First as mentioned, to include things like caves and overhangs or any other terrain features that need multiple elevations at a certain latitude and longitude. This in a sense is an improvement over a typical height map terrain, which allows only for one elevation at one point latitude-longitude. Instead of needing separate models to fill in for these multiple heights, this can now be done in one seamless mesh.

Secondly it is used to determine the final polygon density of a certain chunk (See subchapter: Chunking, Cartesian). The system uses a three-dimensional grid of data points or voxels. The amount of voxels is the same for each detail level, however the volume get smaller at higher detail levels, resulting in more voxels per cubic unit, or in other words a higher sampling rate.

Finally converting from polygons to voxel volumes and back creates a polygonal mesh that has a uniform distribution of polygons. This means especially steep slopes will get less polygonal stretching, which results in better vertex-lighting quality. This reduction of stretching also allows for more interesting noise patterns. (See subchapter: Noise)

The collection of voxels is sampled from an initial planet mesh that is basically a height map terrain. After that other volumes can be added or subtracted. Caves being one of the volumes that get subtracted. At a certain point a higher sampling will not increase the likeness to the initial mesh anymore, but relatively small details, like caverns need a high sampling rate to be actually sampled. From these volumes again a polygonal mesh is created. The higher the sampling rate, the more polygons per cubic unit. This again allows for more detailed noise patterns.

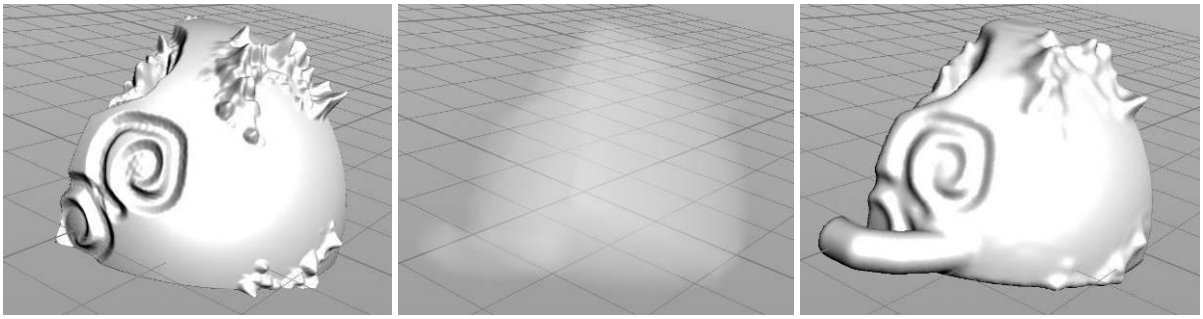


Figure 34: Initial polygon mesh, Houdini Fog volume (Voxels), Output polygon mesh.

In this conversion the original polygon mesh has more detail than the output polygon mesh. This is because at this sampling rate the polygons and vertices is less dense on the output than they are on the input. However the vertices are distributed more evenly across the mesh. Also note the absence of seams where the arch meets the terrain.

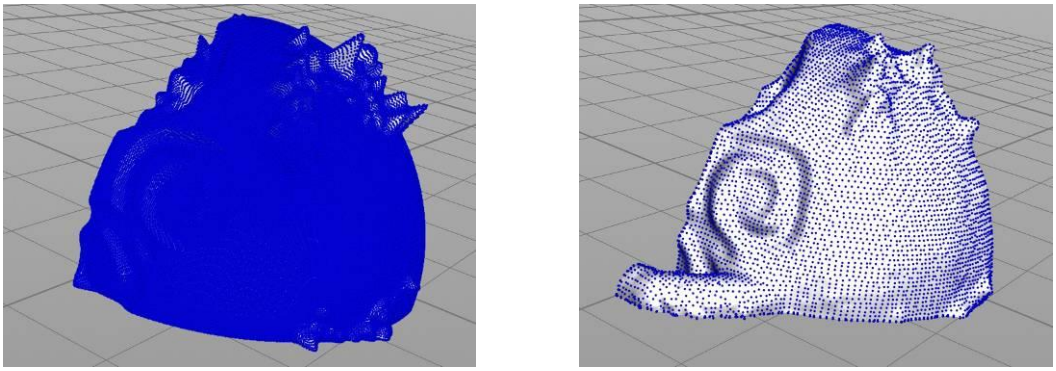


Figure 35: Note the difference in the distribution of vertices at the mountainous areas.

Due to the nature of volumes, they give a "stepped" result by default. When converting from a polygon mesh to a volume, a voxel is either inside or outside the mesh and when converted back this binary nature shows this as a stepped mesh. Luckily Houdini has the option of blurring the volume to counteract this. VDB volumes generally create a less stepped terrain.

Additionally volumes need a very high sampling to represent sharp corners. This would be very time consuming to calculate, so instead a chunk (See subchapter: Chunking, Cartesian) is first created slightly larger than needed. After converting back this part is simply cut off, this way chunks meet up.

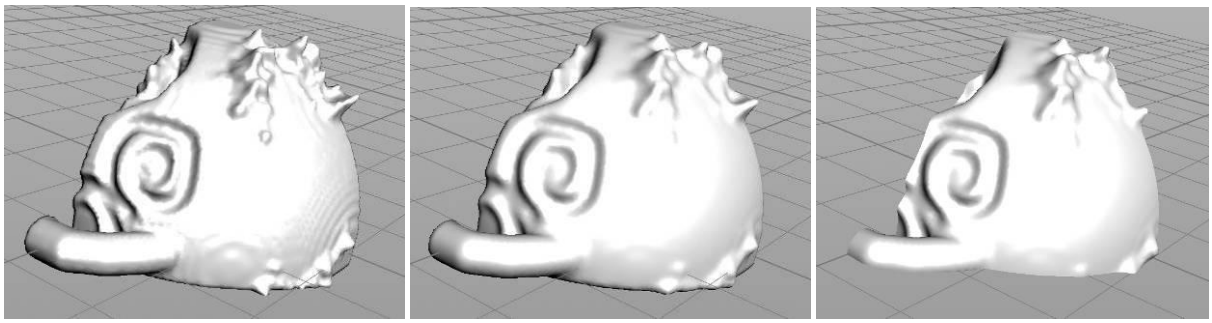


Figure 36: "Stepped" mesh , Mesh from blurred volume & Mesh cut to the correct size.

NOISE

In the generator a lot of parts use kinds of noise. Noise in this context can be best described a collection of pseudo-random values. These values can be 1-dimensional, 2-dimensional, 3d, etc. These collections are created by adding multiple layers of 1-D functions on top of each other, for every dimension.

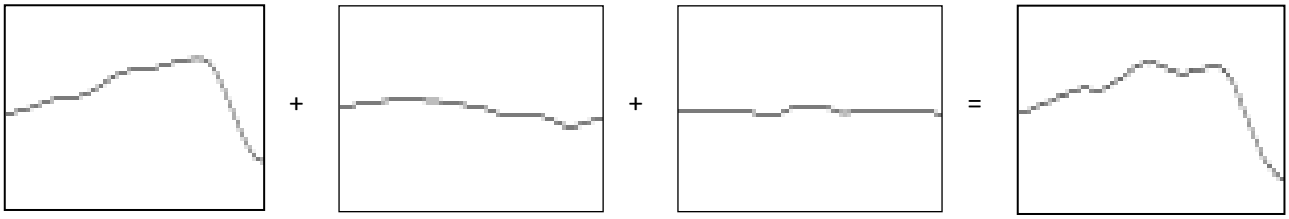


Figure 37: A one-dimensional noise pattern, visualized as a graph. (Elias, 2003)

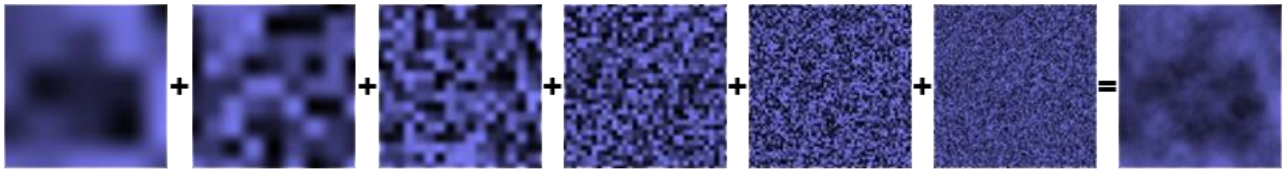


Figure 38: A two-dimensional noise pattern, visualized as a bitmap. (Elias, 2003)

The procedure makes use of 1D, 2D and 3D noise. Noise can be used to make something look more random very effectively as well as making things look more organic.

When creating continents (See subchapter: Climate generation) noise is used to pseudo-randomly offset the borders of the continental plates perpendicular to the sphere's normals.

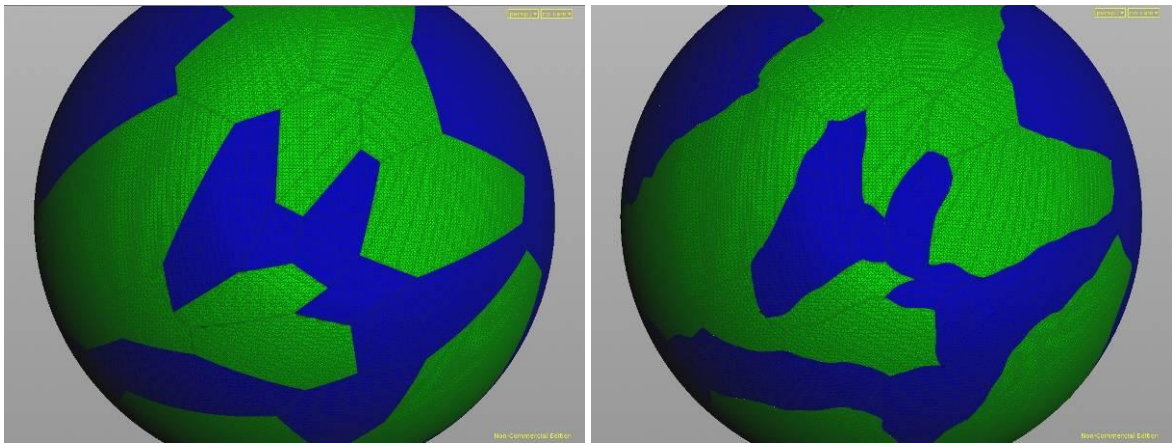


Figure 39: Borders without noise and borders with noise.

As noise is applied, the connections of the land's plates become much less obvious and get an organic look. A great advantage of pseudo-random is, that each time you use the exact same noise settings, you will get the exact same result, even though it looks random. This way detail could be 'saved' into formulas instead of baked into polygon positions. The generator does bake the result into polygon positions at the end of each generation step and when exporting. The reason to do this, instead of saving the noise data, is to make the procedure capable of generating for multiple applications. Otherwise each time the procedure is used inside a new engine, a lot of work has to be done to incorporate this noise functionality. Unfortunately, this baking prevents easy editing and storing these edits at runtime in those engines, so both systems have their merits.

Another part of the procedure that was heavily dependent on noise is the terrain detailing. While the approach taken became different in the end, which is outlined in the next sub chapter, the following method still has its relevance and could have been used instead.

It uses a set of parameters to add certain features to the terrain mesh. Some of the parameters are: latitude, distance to the center of the planet and the steepness of the terrain.

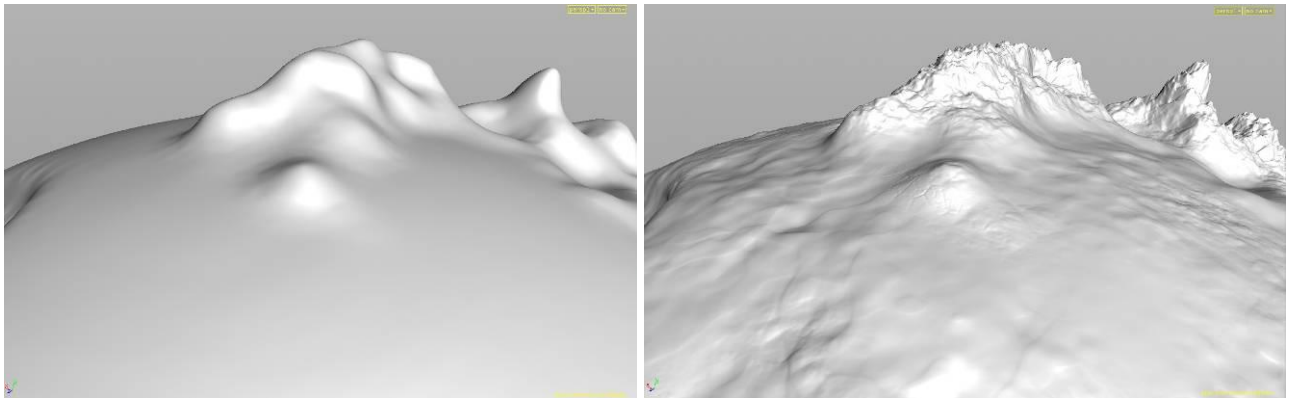


Figure 40: Planet surface Without noise and planet surface with noise.

Note that most of this data will be saved into a normal map (See subchapter: Normal maps). Using noise to create this kind of detail can speed up the artistic creation process. Instead of having to hand paint everything, the artist can determine what kind of terrain types belong where and how each terrain type should look, the rest is handled by the generator. It is even possible to let the procedure determine where each terrain types go by using another layer of noise. This is done in Figure 40.

This part of the procedure, as the rest of the noise, is done in VEX, another node based language within Houdini. VEX is used in SHOP(Shader Operator) and VOPSOPs(VEX Surface Operator). The latter uses mathematical operations on vertices, or points rather, as Houdini calls them. A point is basically collection of all connected vertices that occupy the same location. The network below is used to create the terrain noise shown above. It uses various nodes and sub-nodes, among them the nodes that create the noise patterns. Most of the noise patterns use the position, a three-dimensional value or vector to look up the corresponding value from the noise pattern. This value is then used to push the terrain up or down according to a vector which is equal to a line from the center to the position of that point, divided by the length of that vector, a spherical normal.

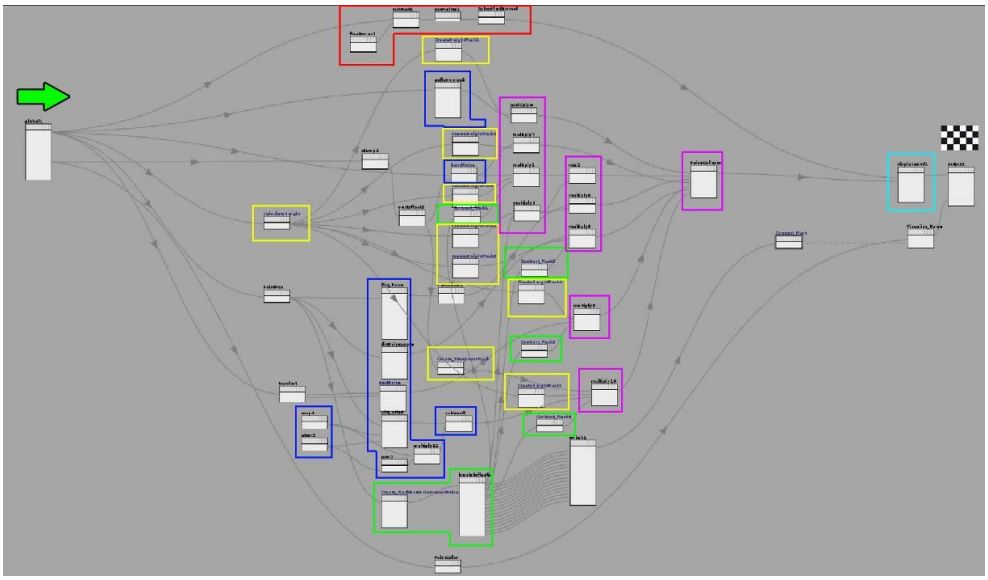
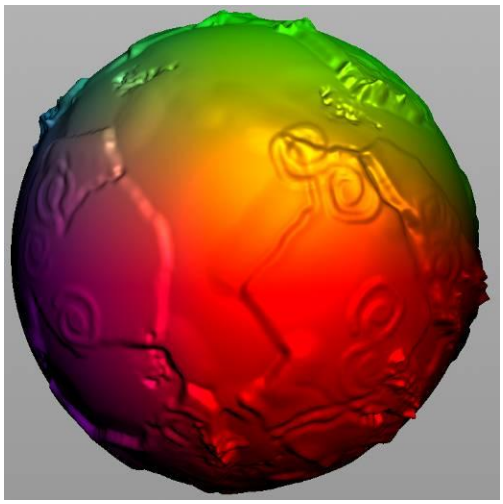


Figure 41: A VEX network inside a VopSop, responsible for detail noise generation.

The network shown on the last page can be divided into groups that fulfill certain tasks. The red group is responsible for creating the vector described on the last page. The output of each group can be visualized with colors for more clarity.



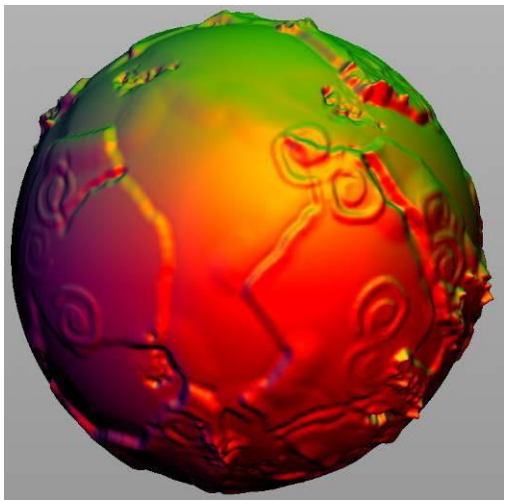
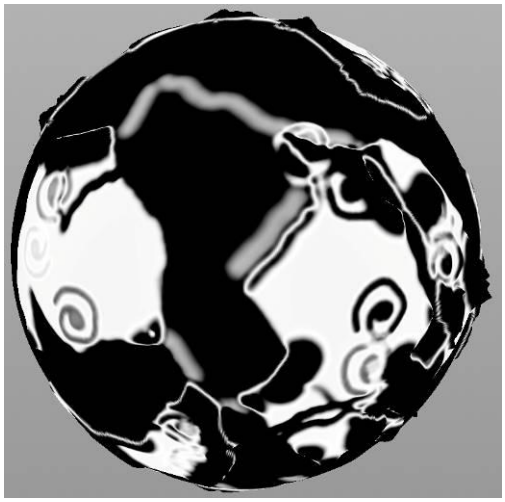
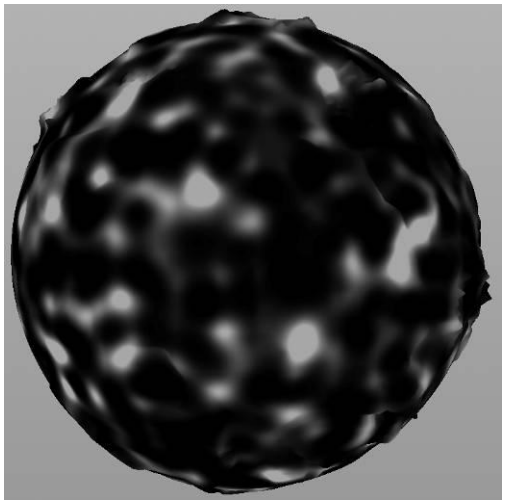


Figure 42: VopSop, Red Group, applies spherical normals. & original surface normals.



The yellow group creates various height masks. These determine at which height range a certain type of noise may occur. The mask fades out gradually at the edges. Each terrain type has a different height mask.

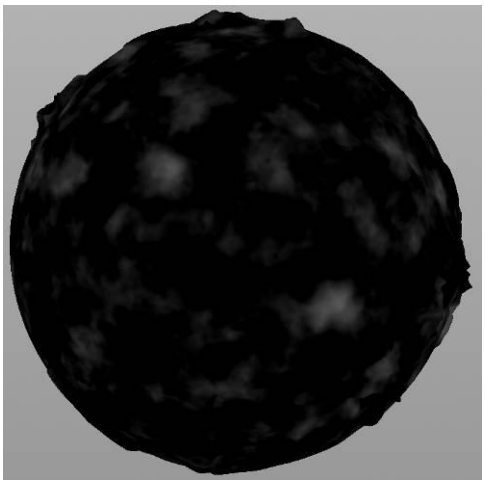
Figure 43: VopSop, yellow group, height masks.



The green group creates a set soft noise patterns. These are used to create and fade-in patches of certain terrain types. This is a fallback for when the locations of the various terrain types have not been calculated and/or set beforehand.

At this stage the climate types (See subchapter: Climate generation) was not yet used to control the various noise types. This is however a feature of the final texture based terrain detailing. (See subchapter: Terrain detailing, texture based)

Figure 44: VopSop, green group, fallback climate types masks.



The blue group creates the actual offset values for the terrain. These are noise patterns as well. There are several types of terrain, which all have a different noise pattern, sometimes with similar settings.

Figure 45: VopSop, blue group, offset noise patterns.

The purple group combines all the masks from the yellow and green groups and multiplies this combination with the noise patterns from the blue groups. This limits the noise patterns to certain parts of the terrain that were either picked by an artist, set pseudo-randomly or a combination of the two.

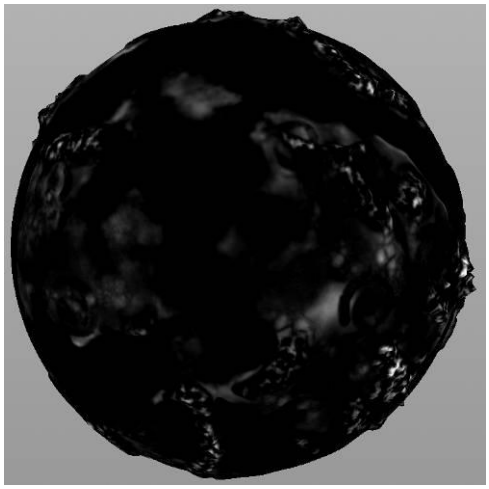


Figure 46: VopSop, purple group, combined and masked offset values.

The teal node moves the points up or down. This uses the direction calculated by the red group. The distance the points move is determined by the result of the rest of the nodes, combined by the purple group.

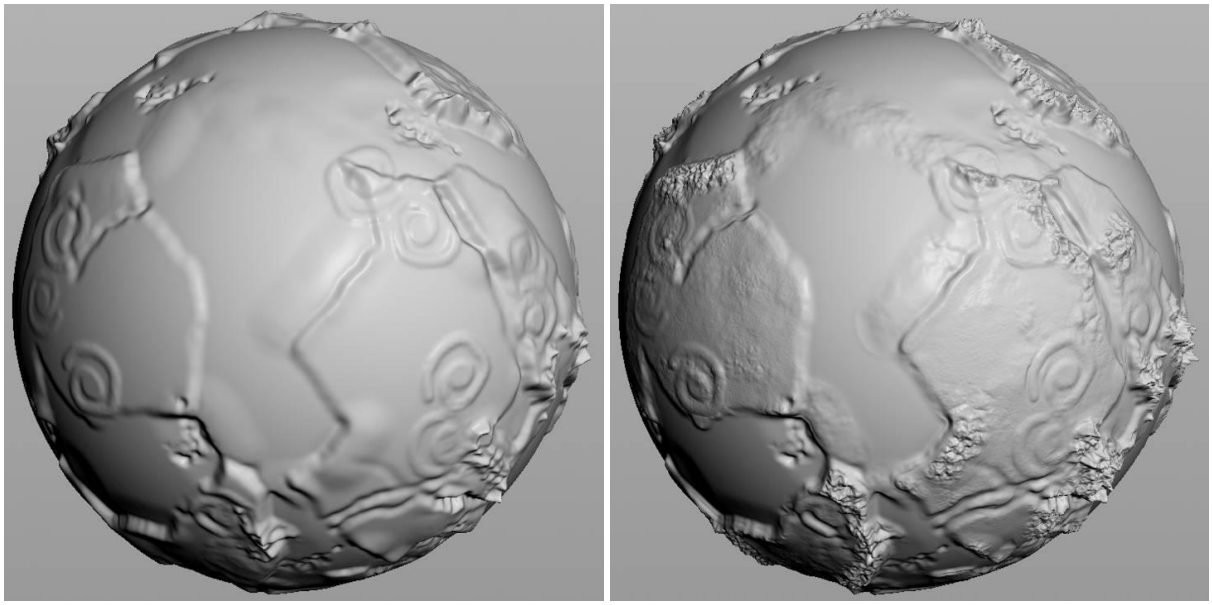


Figure 47: VopSop, the input and final output of the teal node.

VopSops can be used for a wide range of tasks. As well as being very powerful, they are also quite fast. In many cases one VopSop can do the things a group of other nodes do combined, and usually faster. For example when a Houdini ray Sop is used to morph an object into a sphere. A VopSop can do this in a different way, generally four times as fast. Simple VopSop operations can alternatively be done in a single point Sop. This can lead to complex expressions, and longer calculation times but can be a help when unfamiliar with the VEX language.

The pure noise approach provided what was needed for the generator for a while, but it was lacking on multiple fronts. First of all, the amount of noise types that could be assigned was locked, it was possible to simply not use certain types of noise, but it was not possible to add additional types without hand editing the mathematics within the VopSop.

Another problem with the pure noise approach was the interface. The entire interface consisted of sliders and some toggles. While this worked technically, it was hard to keep track of all the values and time consuming to tweak.

A final problem was the limit of artistic freedom the system offered. The noise procedure was capable of creating more realistic noise types, but there was no way to supply this system with specific shapes to use. This severely limited the noise types that were available and would make stylistic or cartoony planets generated by the system less interesting.

The method for detailing the terrain was rebuilt to make use of textures instead. The system was set up in such a way it would eliminate all three problems with the pure noise approach. It also improves on other areas that would be possible to incorporate in the pure noise approach as well. To tackle the first problem, the system separated the noise types into different operators. To clarify this, in Figure 41 the purple node group combines all noise types inside the VopSop, in the new approach the noise types are combined at a higher level, so you would not need to go inside a VopSop anymore to edit the amount of noise types. With the ability to combine the different noise types on a higher level, it is very easy to add additional noise types on top of the ones already existing. After combining all the noise types, a final node displaces the points, like the teal node in Figure 41.

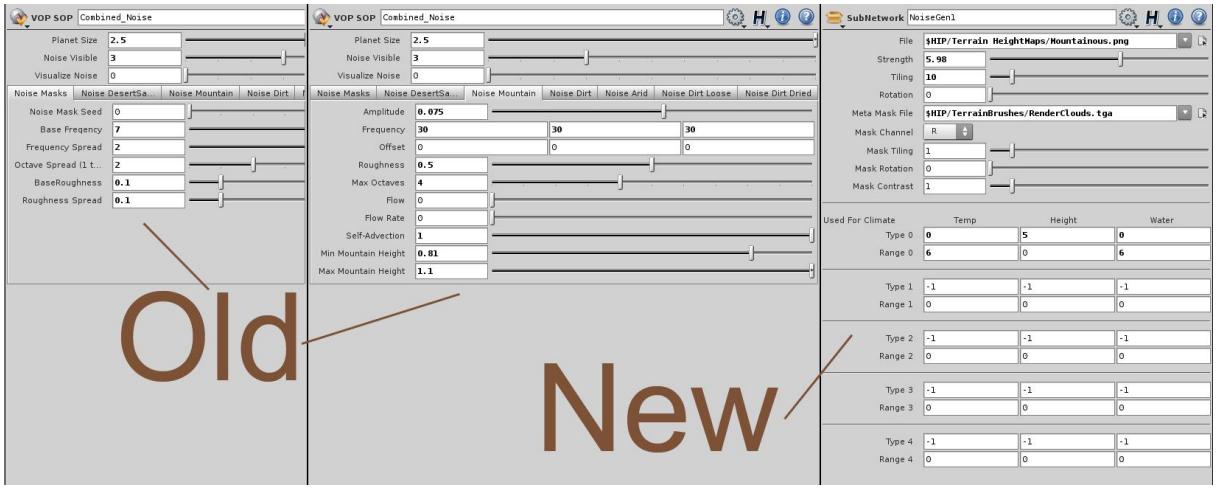


Figure 48: Old versus new noise Interface

By using textures, the amount of variables you may want to change are more limited, than the pure noise approach. With the old system artists had to set values that may have seemed arcane to them, like: "Turbulence", "Attenuation", "Flow" or "Self-Advection". With the new system the values that can be changed are more straightforward, like: "Tiling"(Frequency), "Strength"(Amplitude) and "Rotation". Another problem with the old system was, that with each noise type, there were different values to set, making it even less accessible.

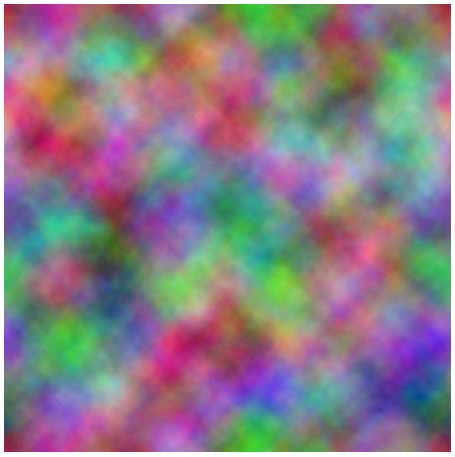


Figure 49: "Render cloud-meta-mask"

The system also supports the usage of a "Meta mask" texture to reduce visible tiling. This mask can also be tiled, rotated and the contrast can be adjusted. This "Meta mask" can be something like a "Render clouds" texture.

The new method creates more direct control over the generated noise and is more easily adjusted per noise type than the old system. Note that in the old system, everything was controlled within one node, whereas the several noise types are now separate, with each their own interface.

The last problem mentioned with the old system, the limited artistic freedom, is inherently solved by the texture approach. For nice looking noise, the artist will need to supply the system with tiling textures, but apart from that requirement, the possibilities are endless, allowing for the freedom that the original system was lacking. To still allow for the pure noise approach, a compromise has been made in the form of a small extra tool to convert Perlin noise to a tiling texture.

Of course the new system had its own difficulties in the creation process. The old system used point position for the seed of the noise. For textures to work properly you need to supply 2D coordinates or UVs (See subchapter: Texture coordinates). So the first step was deciding on how to convert these 3d point coordinates to 2d coordinates. Instead of using the actual UV coordinates, which are generated in a later stage of the procedure, polar coordinates are used. To more specific one set of polar coordinates for each axis (X,Y and Z). This is done by the orange node group in Figure 50. This group also sets the tiling and rotation for the noise type. The other node groups have been color coded in the same way as figure 41: The blue group samples the noise from the texture, The yellow node loads the climate mask from another VopSop, the green node loads the meta mask from another VopSop and the purple nodes combine the result. The way the meta mask is generated is very similar to figure 50.

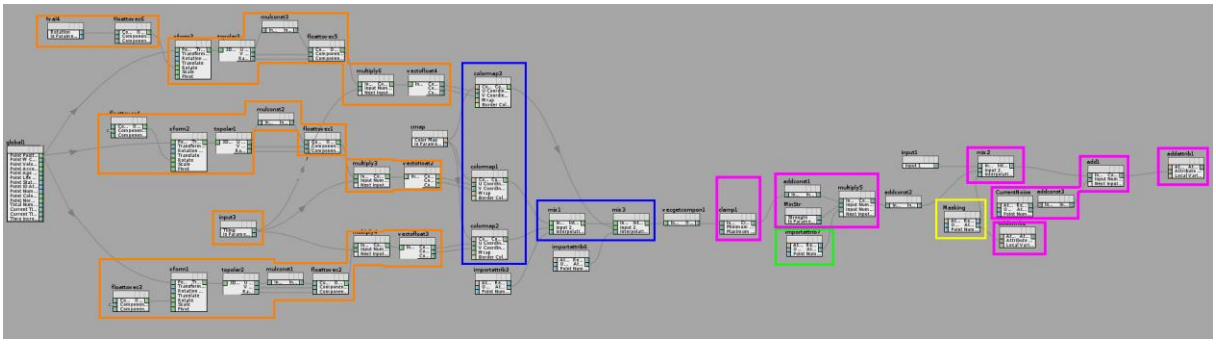


Figure 50: A texture noise generator - texture sampler.

The calculation of climate masks is now done in a separate VopSop and improved altogether. In the old system the climate of each noise type was locked, it was possible to tweak the height level at which the noise occurred, but the moisture level for example was locked per noise type. This was one of the things that really needed to be changed and was therefore updated in the new system. In the new system it is possible to select 4 applicable climate type ranges, in the same way as the height map brush system (See subchapter: Brush Placement System). This way it is possible to use a noise type for several climate types as well as having several noise types per climate type.

Another update applied to this system, as well as the height brush system is the way textures are used. At first black was neutral and white was either up or down. This prohibited using a texture for both up and down displacement. A solution to this was making gray neutral, white up and black down. The whiter the texture is above the central gray value, the stronger the displacement. The same applies inversely for black. This is one extra thing an artist will need to know to properly use the system, but artists get more freedom in return.

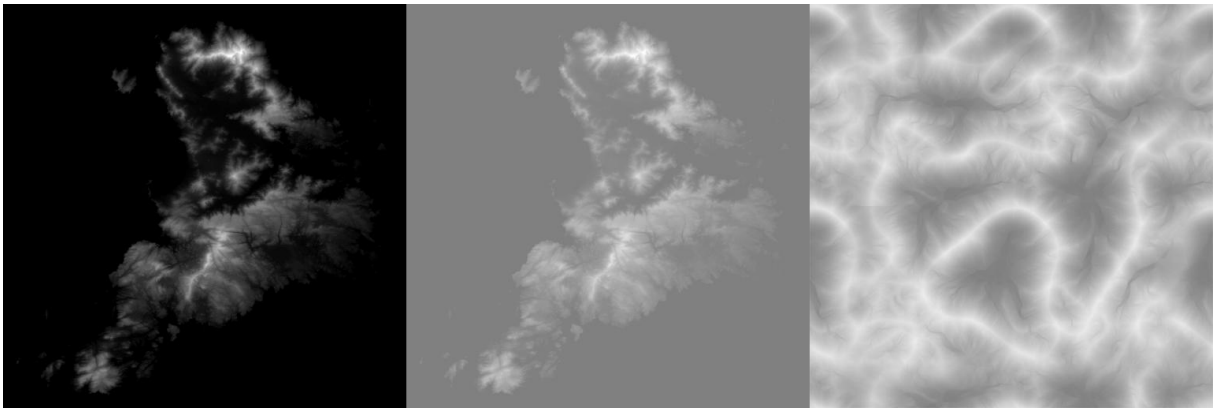


Figure 51: Old Brush, new brush and tiling detail noise texture

A final addition is a high resolution preview option for the new noise. The computer used to create this procedure has 8 Gigabytes of RAM, this proved to be a bottle neck when trying to have more than 3 million points on screen. In short this means it is impossible to run the entire planet on very high detail. However to show the maximum fidelity of the noise you may need to subdivide the planet mesh more than three times. creating about 25.6 million points for four subdivisions. The trick is of course to only show a small area at this high resolution.

By subdividing the mesh in a small area in the middle of the screen it is possible to create a much faster preview. This allows for quick editing of the various noise types and viewing the results.

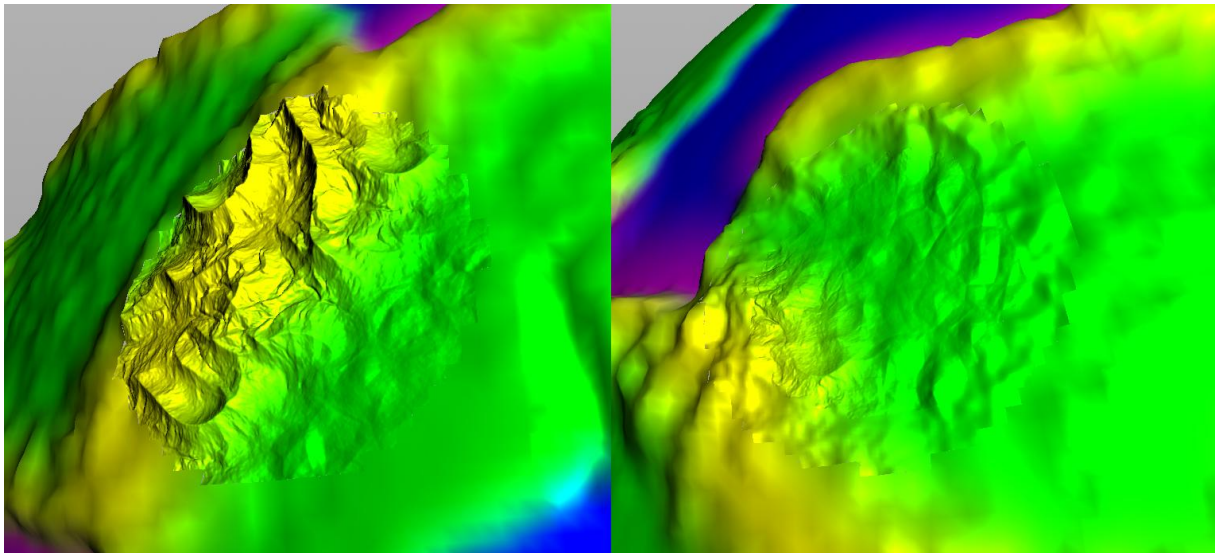


Figure 52: noise preview

One of the main reasons to use voxels (See subchapter: Voxels) was to make the integration of caves easier and more seamless. This way the creation of the planetary base mesh and caves could be partially separated.

The base mesh is taken after continents are formed and height maps are applied. (See subchapter: Brush placement system) From this the cave systems are generated. The user defines at what terrain steepness cave entrances may occur and how many and how close entrances may be to each other. These entrance points are connected using a "tetrahedralize" node in Houdini. This creates paths that form triangular shapes. Connections that are either too long or too short are removed.

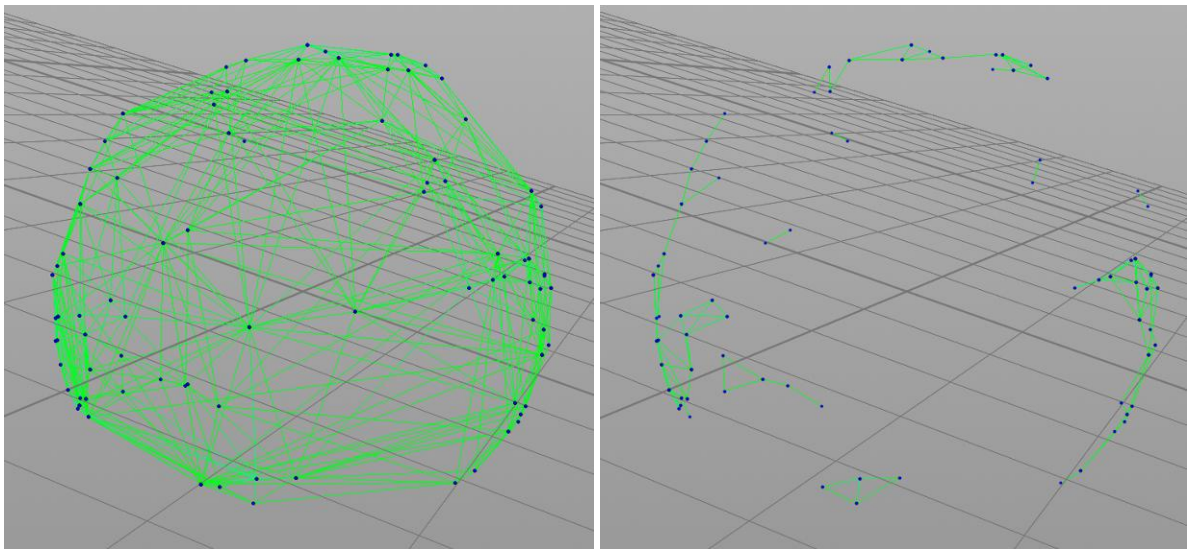


Figure 53: Connecting cave entrances and removing unwanted caverns

All the connections that are left are transformed into more detailed caves. This is done by replacing the lines connecting the cave entrance points by L-Systems. L-Systems were invented by Aristid Lindenmayer in 1968 (Wikipedia, 2013e) L-systems are a type of language where through multiple generations, each time the complexity of an object is increased by adding and replacing parts. L-systems are most commonly used for creating and simulating flora, but can used in many more ways, even architecture. In this case L-systems are used to create various different possible cave structures. Each line is replaced by a sequence of cave structures depending on the length of the line. Note that the caves are still straight and have white and black areas.

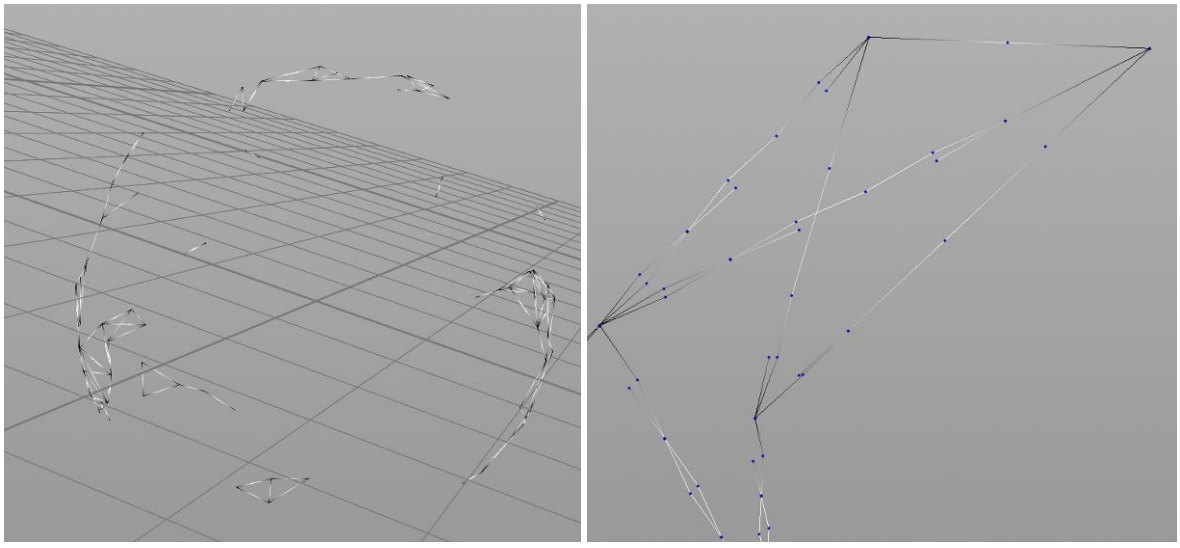


Figure 54: Replacing connections with L-systems and close-up of L-systems

After implementing the L-systems the caves are bent towards the center of the planet. This is done to prevent intersection with other parts of the terrain. The White areas of the caves are allowed to bend, whereas the black areas are kept in place. The same goes for the noise that is applied after. (See subchapter: Noise) Noise is applied to make the caves appear more organic and make sure each cave is unique.

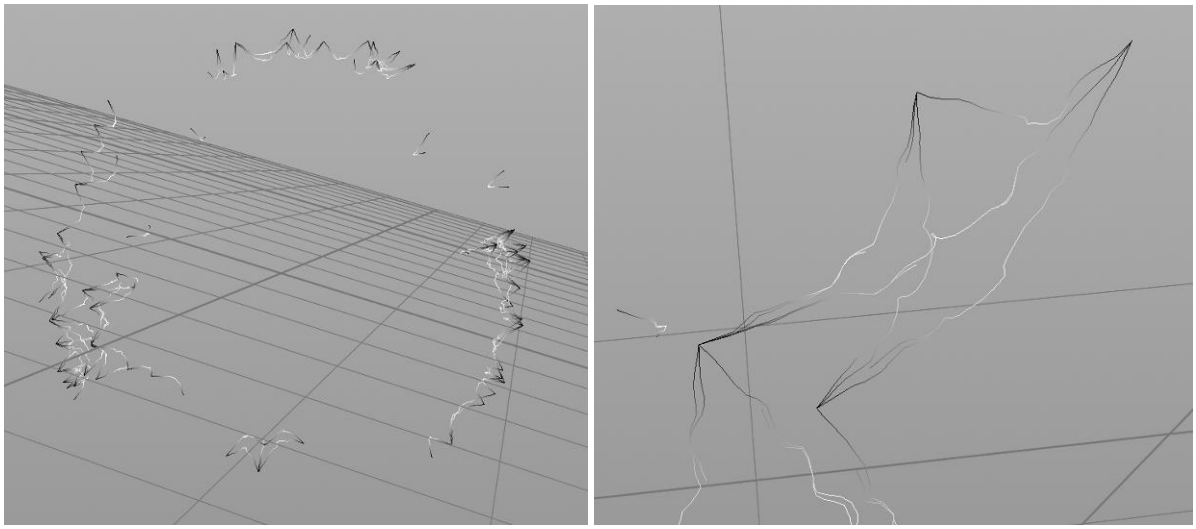
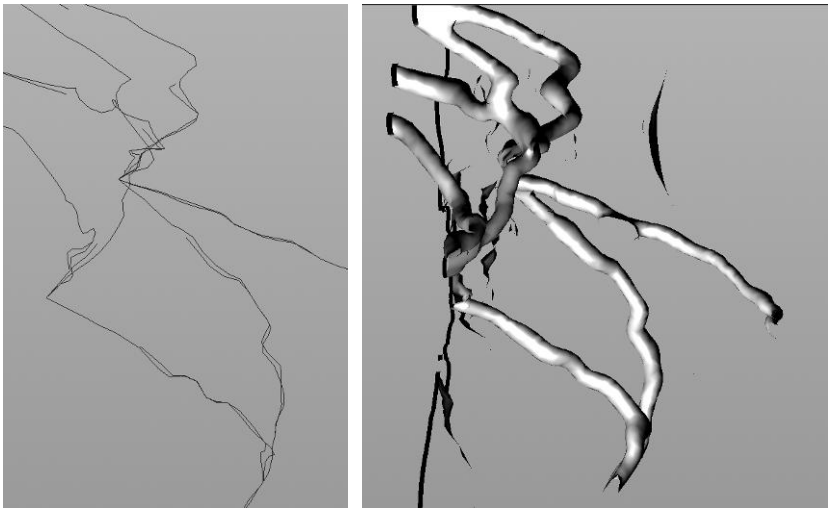
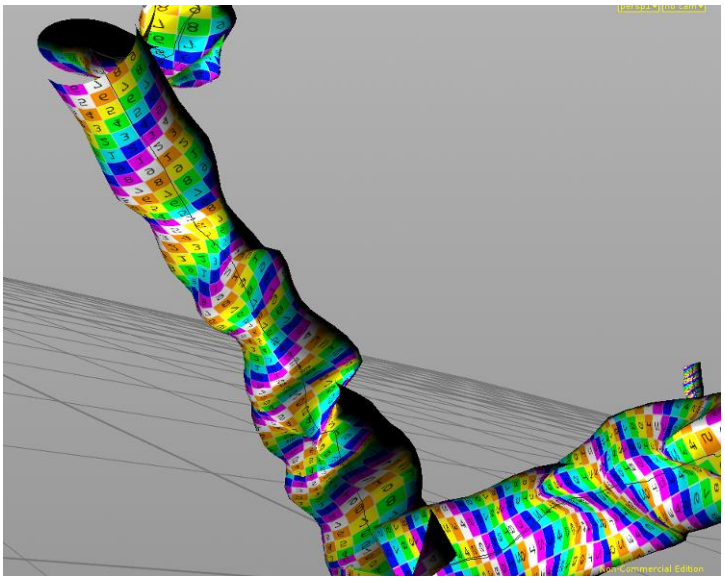


Figure 55:Caves bent towards the center with noise applied and a close-up of the noise



Note that the caves do not have actual geometry at this point. The geometry is created by removing geometry around the cave lines in a certain radius. This is done in Houdini volumes. (See subchapter: Voxels) Remember that this subtraction is done per chunk (See subchapter: Chunking, Cartesian)

Figure 56: Subtracting to geometry



This creates another problem: Lines cannot effectively carry enough information for the generation of UVs. To solve this, a "sleeve" is created around each cave with UVs applied. These UVs can later be used, when transferred back to the geometry. This is optional as the overall UV technique creates less seams against more distortion, whereas the sleeve method creates less distortion, but obvious seams.

Figure 57: UV "sleeve"

Apart from the base terrain mesh the procedure also supports automatic placement of objects. The procedure generates all the necessary data, which is then exported and imported separately.

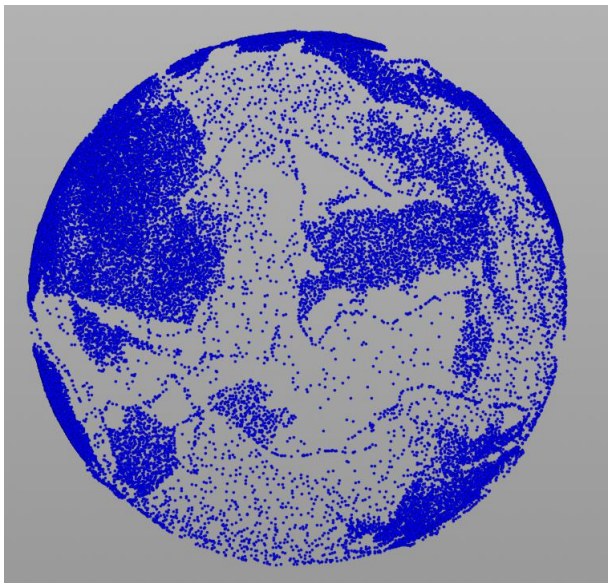


Figure 58: Object data point cloud

The procedure does this in the form of a point cloud. Each point contains the following data: position, rotation, scale, type of object, rendering distance and the amount of objects of the specific type to keep in memory. This data is then read inside Unity and processed there. Note that the actual objects are not manipulated in the procedure, only their associated data. The objects and their data are linked in Unity. (See chapter: Implementation/Object Placement)

The controls of the object placement also uses the procedure's climate simulation, just like the height brush and terrain detailing controls (See subchapter: Climate generation). In figure 59 first you may select either a set amount of objects or a relative amount per world unit. Note that the radius of the planet during generation is 1 world unit without deformation, so the surface area is $4\pi r^2$ or about 12.6 world units for a perfect sphere. With a density of 10000, this makes for about 126000 objects to be placed.

Normally objects are distributed uniformly, however to create more appealing placement, clustering of objects may be preferable. The range determines the cluster size, the actual clustering attracts the object points to the center of the cluster. With a value of 0 all object retain their uniform position. With a value of 1 all objects are dragged on top of each other.

The minimum distance between points can be set to prevent overlapping, the larger the object, the larger this value should be. Note that overlapping points are subtracted from the total amount of objects. The same goes for the climate limitations. With a value of 0.005 about 93700 of a hypothetical 126000 objects would remain. Note that this value must never be higher than the clustering range, or all clusters would be merged to one point.

For all climate axes (Temperature, Height and Water), river and steepness a range may be set where objects may be placed. A range from 0 to 1 means that no points are removed. A temperature range of 0 to 0.2 for example would limit points to only the polar regions and mountain tops.

Setting a falloff gradually removes more points towards the range borders. separate falloff values may be set for both ends of the range. At the start of a falloff value all points are removed, at the end no points are removed. To illustrate: With a range of 0.3 to 0.6 and a range of 0.1-0, at 0.35 about half of the points are removed and high border there is a sharp border, keeping all points right up until 0.6.

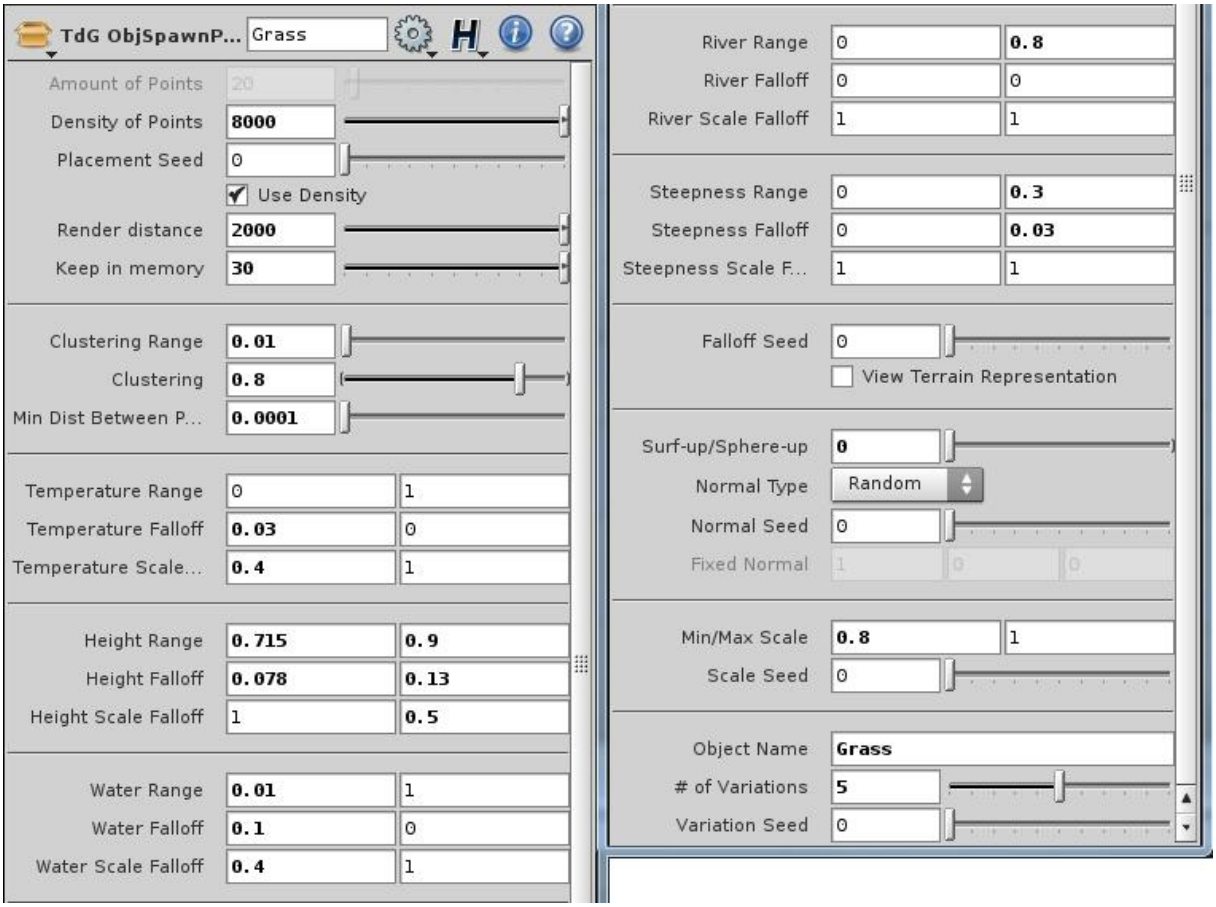


Figure 59: Object placement interface

The scale falloff values are specifically useful for flora objects. For example trees grow highest in their ideal climate, but grow smaller in harsh environments. This value emulates just that and works in conjunction with the falloff value. The scale falloff is a multiplier: when the scale falloff is set to 0.4-0.9 the scale is multiplied by 0.4 at the start of the falloff and by 1 at the end of the falloff at the end of the high falloff the scale would be multiplied by 0.9. The scale multipliers stack multiplicative across all climate limitations. From figure 59 for example a cold or a dry environment would both result in a scale multiplication of 0.4. In an environment that is both dry and cold, the scale multiplication would be 0.4*0.4 = 0.16.

On the right side of the interface, normally below the rest, a river limiter has been added. Mainly to be able to prevent the placement of trees underwater, but can also be used to place different objects at river bottoms/banks than ocean floors/coasts. Likewise a steepness range has been added. This way the user can remove all points from mountainsides, or limit the placement to mountainsides. Steepness 0 equals flat and 1 equals 90 degrees of the surface.

Below that is a falloff seed, which gives different variations on which points are removed over the separate falloffs. "View Terrain Representation" gives a preview of the surface areas that remain after applying all the climate limitations and falloffs.

The next block handles the rotation, which is split into an up vector and a normal vector. The first slider sets how "upright" an object is compared to the steepness of the terrain. With 0 the object adheres the steepness of the terrain totally, with 1 the object is totally upright against gravity. The normal can be set to either: Random, Aimed or Fixed. The normal is always 90 degrees to the up-vector. Random is self explanatory, aimed uses a second point to aim all points towards while adhering to the 90 degrees rule. With a fixed normal it uses the normal closest to the chosen value that is still 90 degrees to the up-vector.

The Min/Max scale is also relatively straightforward. The resulting scale is multiplied by the final scale falloff.

Finally a name must be set for each point group. This way file parser (See chapter: Implementation/Object Placement) can link the correct data to the correct object. with the "# of variations" it is possible to have multiple names within the same group. When the name is "Grass" and the #-value is set to 0 the output will be just "Grass", when set to 3, the output would be "Grass0", "Grass1" and "Grass2".

When one object type is set it is relatively easy to add more types. These can simply be merged together. The point cloud in Figure 58 has 5 different object types merged together. Note that because the points are generated separately, it is possible that they overlap between object types. To fix this, after merging all overlapping points, those with lower "minimum distance between points" are removed. When two objects with the same minimum distance overlap, one is picked at random.

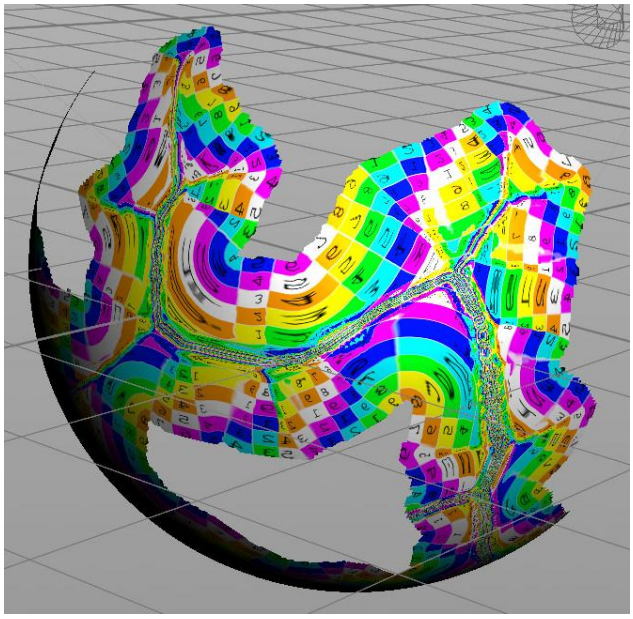
The cloud in Figure 58 is generated on the template mesh, see Figure 25B, to preview the placement as well as to be able to "View the Terrain Representation". For the final export the points need to be generated per chunk, on the highest level of detail (See subchapter: Chunking, Cartesian). This way the points do not float above or are sunken into the surface. Note that very large objects, that would be visible from space, are best placed manually.

WATER

While the placement of water is mostly handled by the climate generation, (See subchapter: Climate generation) there is a need for actual water meshes. The climate generation handles the placement of the ocean floor as well as the river bottom. These can easily stand in for ocean and river surface at very high distances.

At these distances having two meshes relatively close to each other is even problematic. At this distance point floating point accuracy is insufficient to determine correctly which mesh should be rendered in front of which.

However, from up close there is a need for separate water surface meshes. Separate water meshes allows for having more elaborate water shaders (See chapter: Shader creation/Water shaders) as well as well as creating actual underwater areas. (See chapter: Implementation/Under water)



The rivers and main ocean are created separately. The ocean surface is more or less a copy of the ocean floor, extended land inwards and displaced to sea-level. Apart from the standard UVs, a second UV channel is created which roughly follows the coastline. This way coastal waves can be added in the ocean shader.

Figure 60: Ocean surface mesh - 2nd UV channel

Rivers are recreated from the river curves, generated during the climate generation (See subchapter: Climate generation). Per curve the width of the resulting river is calculated and used to create two river border curves. These curves need to be cleaned to avoid overlaps in the end result.

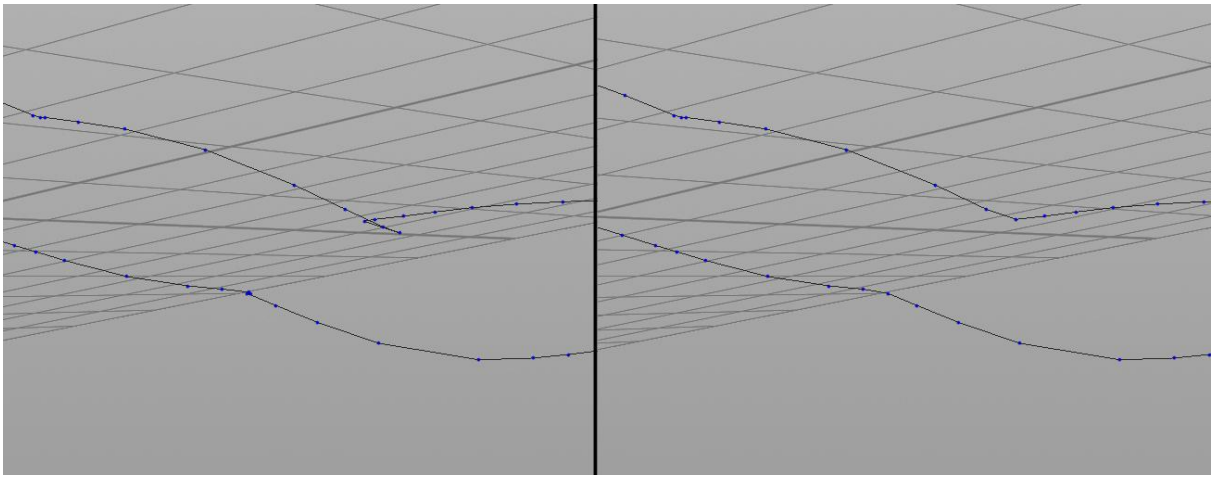


Figure 61: Original river border curves and cleaned curves

Due to the way of constructing, by using NURBS, usable UVs are automatically created. These UVs follow the flow of the river. This allows the shader to animate the textures downstream. With these UVs however, as they exceed the 0-1 range, alpha blending using a texture map is not possible. To solve this, alpha values are instead saved inside the vertex attributes to be used by shaders later on. This is done to create softer edges at the riverbanks.

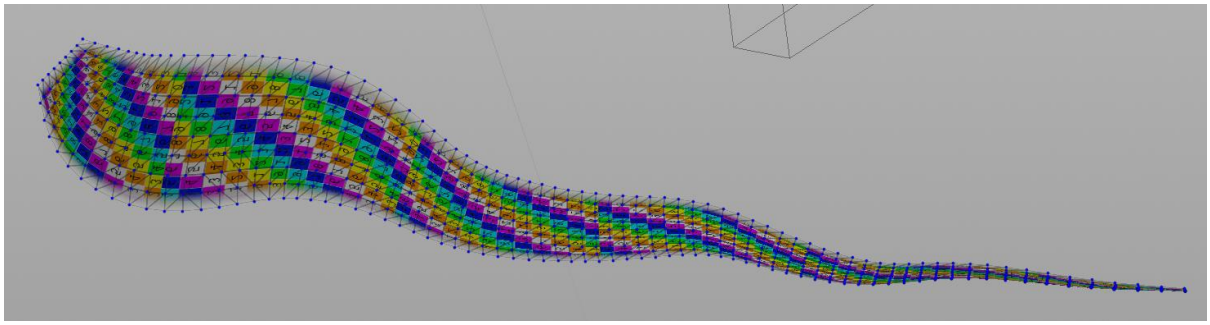


figure 62: Final river mesh

Applying texture maps is a common method of increasing detail of a 3d model. However to project these texture maps correctly on a model, texture coordinates or "UVs" are required. The difficulty with UVs is that they exist in 2d space, whereas the model exists in 3d space. This problem is worked around by "unwrapping" a 3d model. With UV coordinates the model has information on how to apply a 2d-texture back on a 3d-model. If you would peel an orange and cut this peel in such a way it can lie flat on a surface, this peel would technically be an UV-Unwrap. At certain points the peel would try to move back to its original shape, creating bulges or stretching. At these points in a texture, there would be a slight distortion. Good UVs have the least stretching possible as well as the least possible cuts. Stretching versus cuts or seams is a balance, depending on the subject this balance may vary.

Since the subject of this generator produces organic shapes, reducing seams has the priority, while keeping stretching at an acceptable level. At the beginning of the project the general layout of the UVs was decided to be a cube-layout. (See subchapter: Spheroids) The planets are spherical, but a cube can work just like a sphere, or in other words, can have the same topology. Using a cube layout means there are 6 UV-patches, inherited from the cube's faces. Since a cube has 12 edges, there are 12 UV seams by default, however the UV-patches can be stitched together, leaving 7 UV-seams.

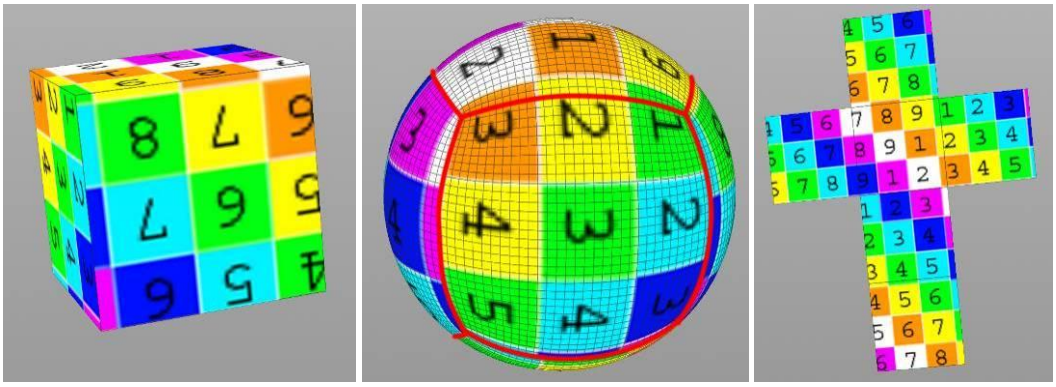


Figure 63: Cube to sphere & unwrapping the spherified cube

Unfortunately the UV-coordinates are lost during volume conversions. (See subchapter: Voxels) This means the UV-coordinates must be transferred back from the UV-template (figure 63B) to the result. This works for the most part, but creates issues at the original UV-seams, because it picks UV-coordinates from either side of the border, but these borders does not necessarily meet, as only 5 out of 12 borders can be lined up. To fix this problem, extra polygon edges are added where the UV-borders should occur. This way the UV-coordinates do not continue beyond the borders.

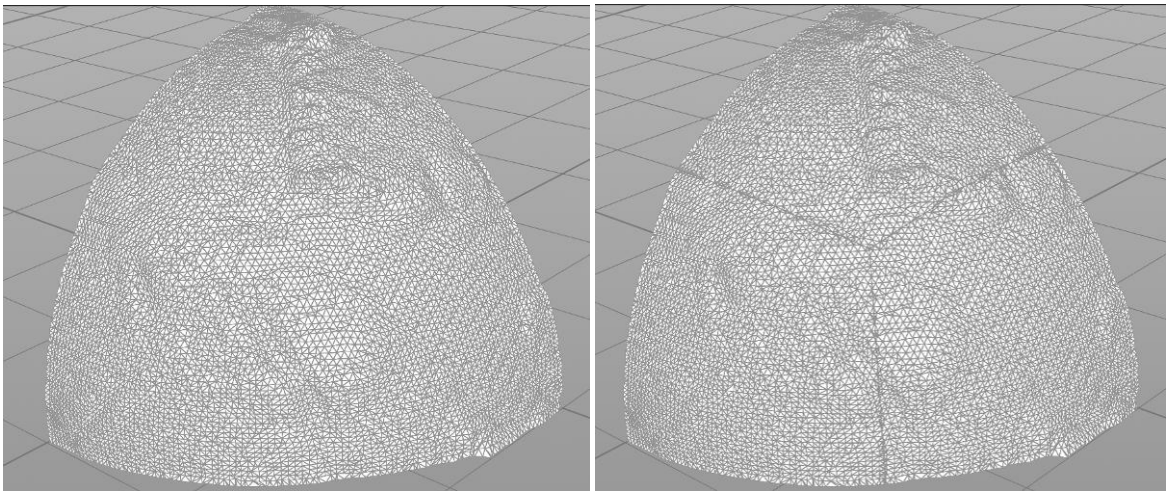


Figure 64: Polygonal edges are added where the UV borders will occur.

This leaves one final problem, which is inherent to the "Attrib Transfer" node in Houdini. At higher resolutions UV-coordinates will "snap" to the UV-template's coordinates, creating distortion in a regular grid, which was very noticeable. To counteract this, the UVs are "relaxed". This means the flat triangles in UV-space are given a more evenly distributed space compared to the size of the original triangles in 3d-space. Since Houdini does not have a feature to relax UVs by default, I would have to create my own. Luckily Charles Trippe (2011) already created a tool to do this inside Houdini. This tool has been slightly adjusted and used to relax the UVs.

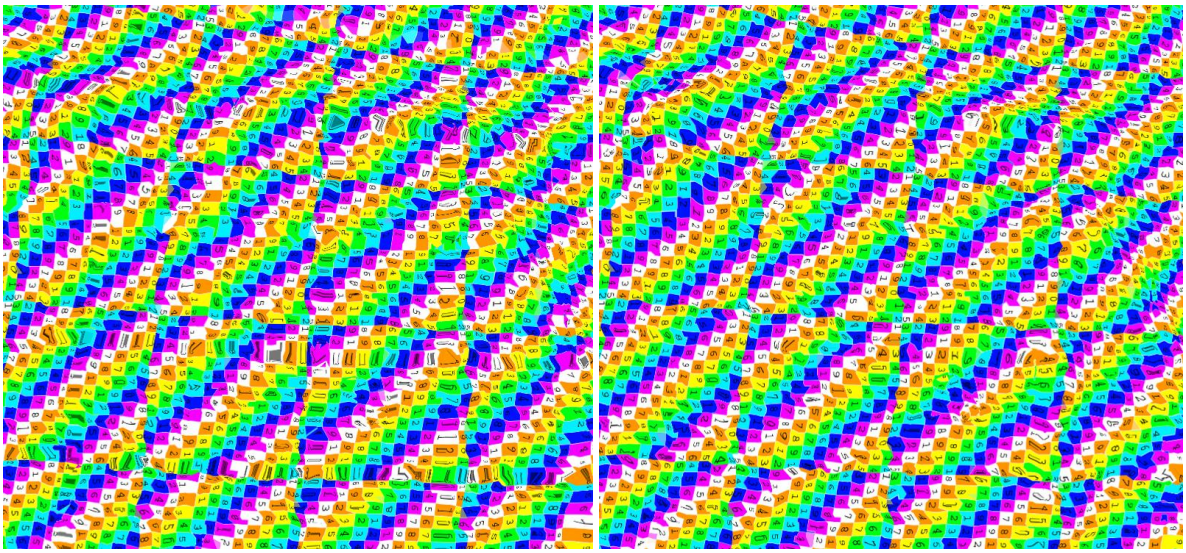
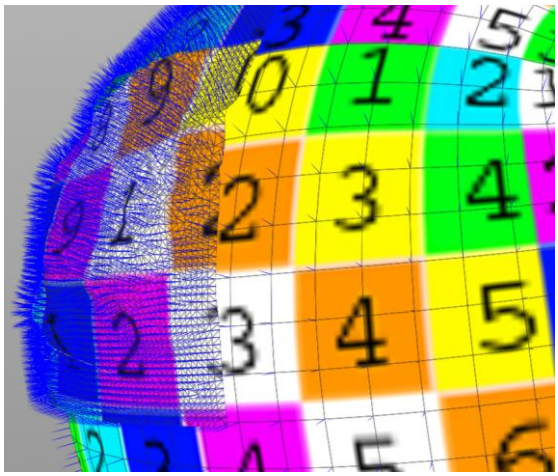


Figure 65: Before and after relaxing UVs

To further improve the UV coordinates, later in the project, the transmitting of UVs was switched to another method using ray casting to transfer the UVs.



Instead of the weighted distance approach used in earlier stages of the project. The ray casting approach did not suffer from the "UV-snap" problem, making the relaxing no longer necessary. The ray casting method works as follows: instead of looking for an weighted average per point, a ray is cast per point opposed to the spherical normal. Where the ray hits the UV template mesh the UV coordinate is extracted and set as the UV coordinate of the mesh. As an added bonus, this new approach is also slightly faster.

Figure 66: Ray Cast UVs

Note that both the old and new system do not handle very steep terrain well. The steeper the terrain, the more distortion occurs, just like with a old fashioned height-map terrain. When the terrain gets steeper than 60 degrees up, the distortion is going to be visible, this does almost not occur with more realistic types of terrain. Solving this problem altogether would be possible by using additional UV-sets, however is at this point beyond the scope of the project and moreover would about triple the shader complexity when trying to render in real time.

The generator is capable of exporting certain data to texture maps. These texture maps can be used in materials in render and game engines. They can be used as maps, such as diffuse and normal maps (See subchapter: Normal maps), but also as masks inside materials. The textures are created by transforming the planet's points to its UV-coordinates as world X and Z. After that vertex colors are applied with the same values as the map that needs to be rendered. For example normals have 3 axis: left, up and forward. These can be converted to a color's axis: red, green and blue. To prevent texture tiling problems, the 6 parts of the texture map are copied and moved to extend the texture map. The result is simply rendered with an orthographic camera from the top.

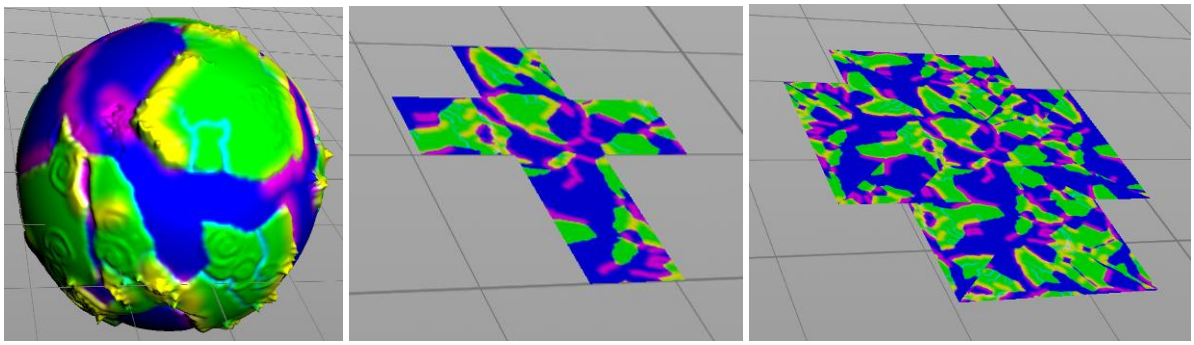


Figure 67: Transforming the planet to UV coordinates and extending the texture map

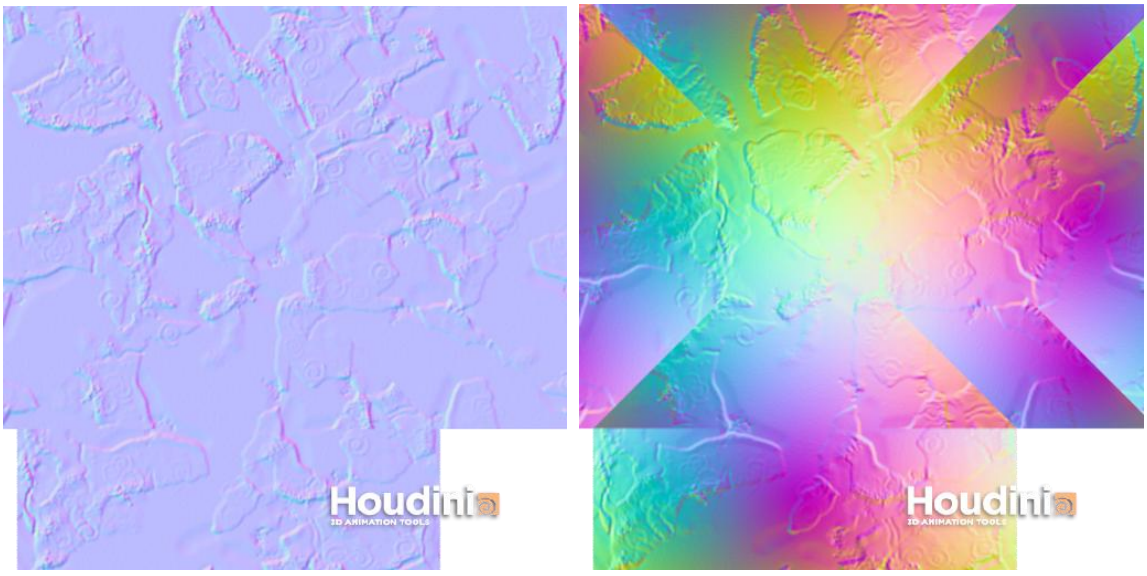


Figure 68: Renders of tangent space and object space normal maps.

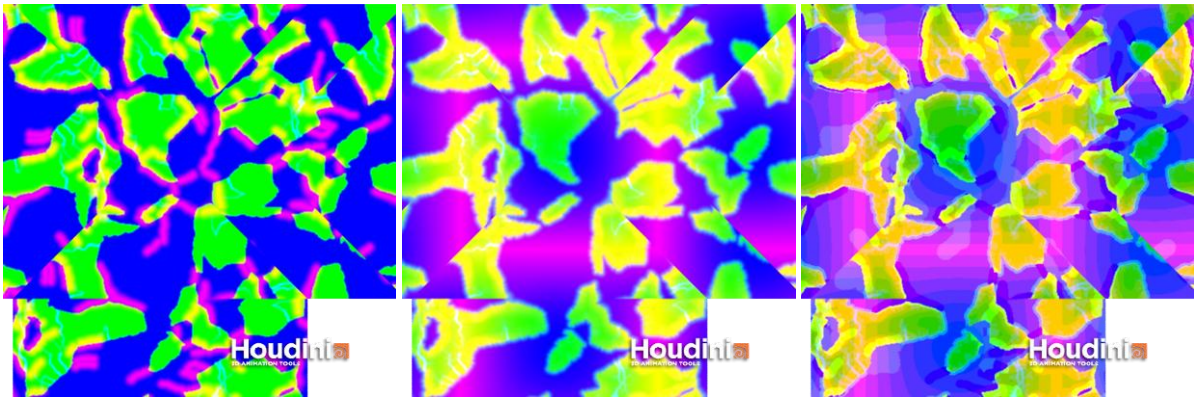


Figure 69: Renders of various masking textures.

Normal maps are an efficient way to fake polygonal detail. Normals allow a game or render engine to determine in which direction a light source reflects compared to the surface. The advantage of normal maps above surface normals is that surface normals can only exist at vertices. Normal maps, as they are texture based, can have reflection data for each pixel on screen. To recreate this level of fidelity with polygons, each pixel on screen would need to have at least one vertex. Few GPUs are capable of rendering this amount of polygons, yet normal maps are supported by all but the oldest GPUs around.

There are two general approaches to creating normal maps textures. The first one is creating related bump-maps by hand and later converting these to normal maps. This can be an efficient method when dealing with angular shapes, like buildings or vehicles. The second approach is creating a higher resolution mesh in for example Z-Brush (Pixologic, 2012). The surface normals from this hi-res mesh can then be baked using the UVs of a lower resolution model. When dealing with organic shapes the latter is most often used.

My Procedure uses a hybrid between the two. As the UVs are pre-determined, the surface normals of a high resolution planet can be directly transposed onto the UVs. Rather than using ray-casts that many applications, like Maya (Autodesk, 2012), use to calculate normal maps.

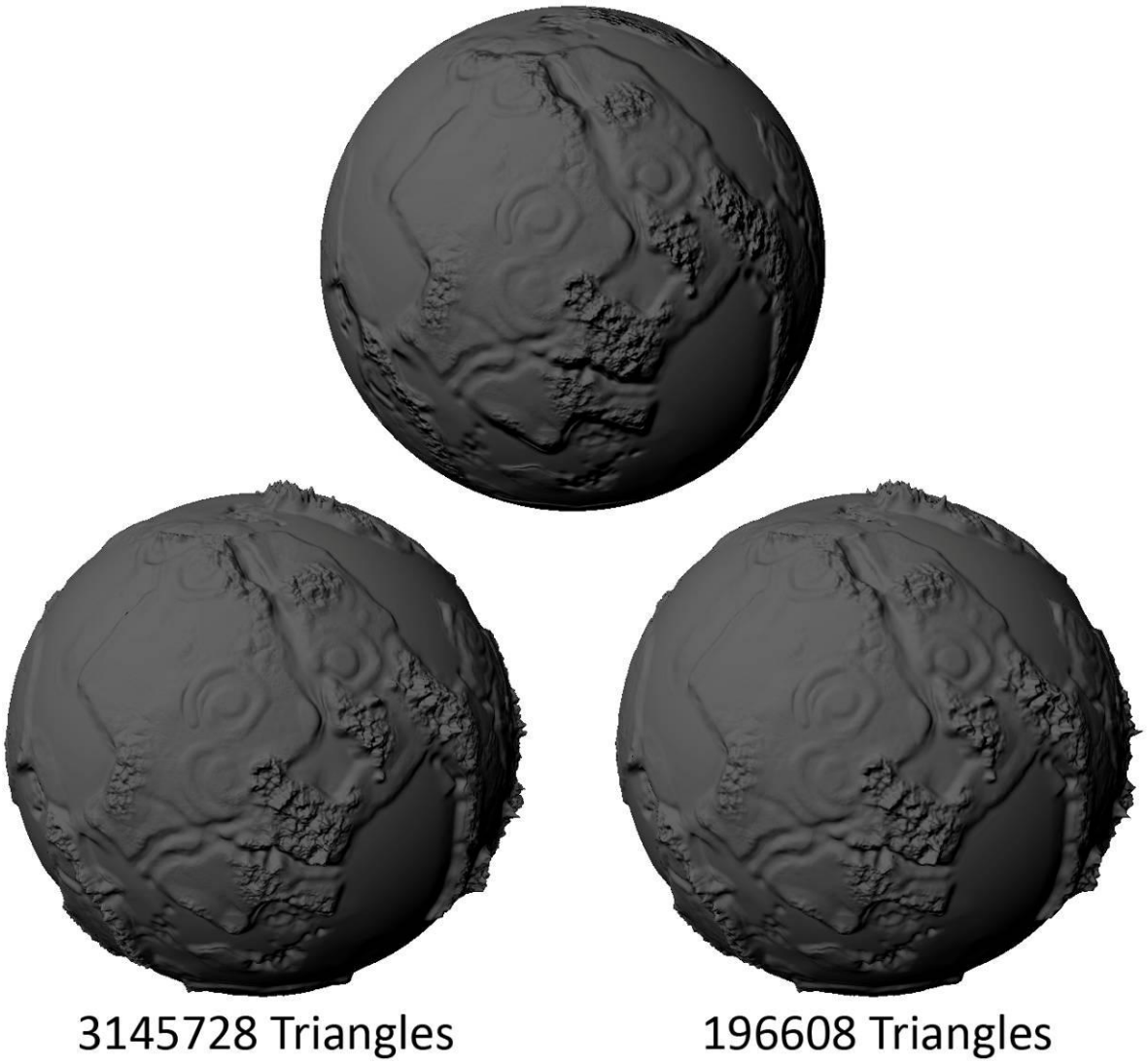


Figure 70: Surface normals of a hi-res mesh saved in a texture map and applied to both a low-res mesh and a sphere.

As visible in the image, normal maps are not a cure-all. They are merely a tool to better balance visual quality and performance. As stated before normal maps fake polygonal details. This best visible on the sphere. Relatively small indents and bumps can be convincing when compared to the high resolution mesh, however when looking at the mountains in the center, the normal map looks flat. When looking at the contour of the planet it is immediately obvious that the effect is faked. There are more advanced shading technologies, such as tessellation and displacement mapping, being developed. These partially solve this problem, but are not supported by game engines of last generation. This project is showcased on the Unity engine, which did also not support new features like tessellation. However, halfway through the project a new version of Unity came out which did support this new feature.

There are two types of normal maps: tangent space normal maps and object space normal maps. These are different in the sense, to what the normal is relative to. A tangent space normal map is relative to the surface normal. These kind of maps will most times appear blue-ish. Here the RGB (Red, Green, Blue) value is relative to the XYZ value of the surface normal, Z being the forward pointing direction. Because the last Axis gets the last color (xyZ becomes rgB) it will most likely be blue. The flatter the surface is, the normal map represents, the less red and green are present, and the more blue the normal map is.

Object space normal maps have the basic same setup but all the values are relative to the objects direction, rather than the surface direction. This means that with spherical objects, most of the entire color spectrum is likely used, resulting in a more rainbow-like appearance.

There are also bump maps. These are grayscale images that define height variations within the surface. These can be converted to tangent space normal maps by various tools, but are not widely supported as-is.

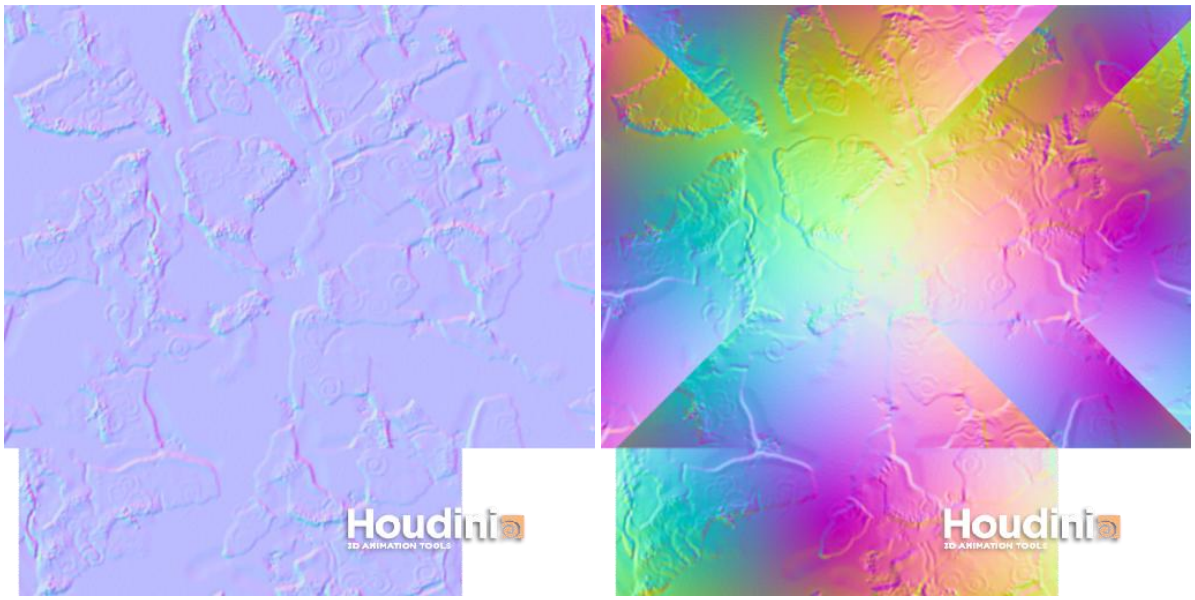


Figure 71: Tangent space normal map & Object space normal map

There are several reasons for using one normal map type over the other. Tangent space normal maps are most commonly used, as they allow for deformation, character animation for example. Problems may arise when detail is needed across sharper edges. The details may stretch too much or give other visual artifacts. Object space normal maps do not suffer from this last problem as these edges can also be baked into this texture. Since planets usually do not deform a great deal, both normal map types are usable.

However when building up the planet, each time the detail level gets higher, the geometry incorporates more and more features faked by the normal map. This causes problems when using a tangent space normal map. Because a feature that is already present in the geometry, but also present in a tangent space normal map will duplicate this effect. This results in visual artifacts, except at the geometry level the map has been made for. This can be solved in two ways: by modifying a tangent space normal map for each detail level, or instead, by using one high-resolution object space normal map.

The reason why object space normal maps do not suffer from this problem, is the same reason why object space normal maps can have details that go across sharper edges: object space normal maps have information of where the detail is, as well as where the surface changes direction. Tangent space normals only have information for the detail itself.

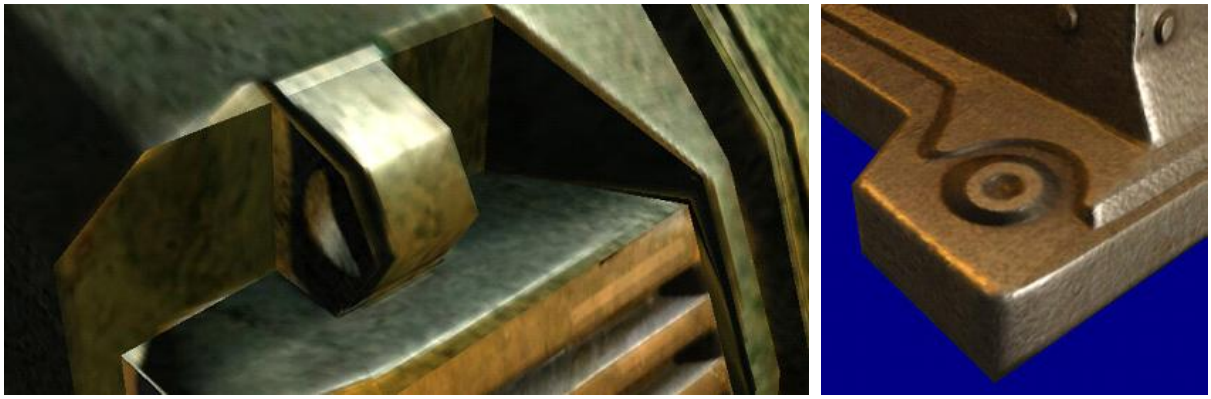


Figure 72: Tangent space normal map versus Object space normal map (Crytek, 2012)

Since not all render engines and game engines, such as Unity 3.x, support object space normal maps, both types of normal maps and bump maps can be exported using the generator. For half of the project, tangent space normal maps were used. However when Unity 4 came out, with Direct X 11 support, it was possible to have more complex shaders. Unity 4 still did not support object space normal maps by default, but it was now possible to create a custom shader to do this, because of the extra memory available.

The problem with this was, that there were none of these shaders available at this point, so I did some research on matrices, most notably the "Tangent space matrix" (García, 2012) and eventually managed to create such shader. The move to object space normals proved very beneficial, making the transitions between levels of detail less visible and easier to accomplish.

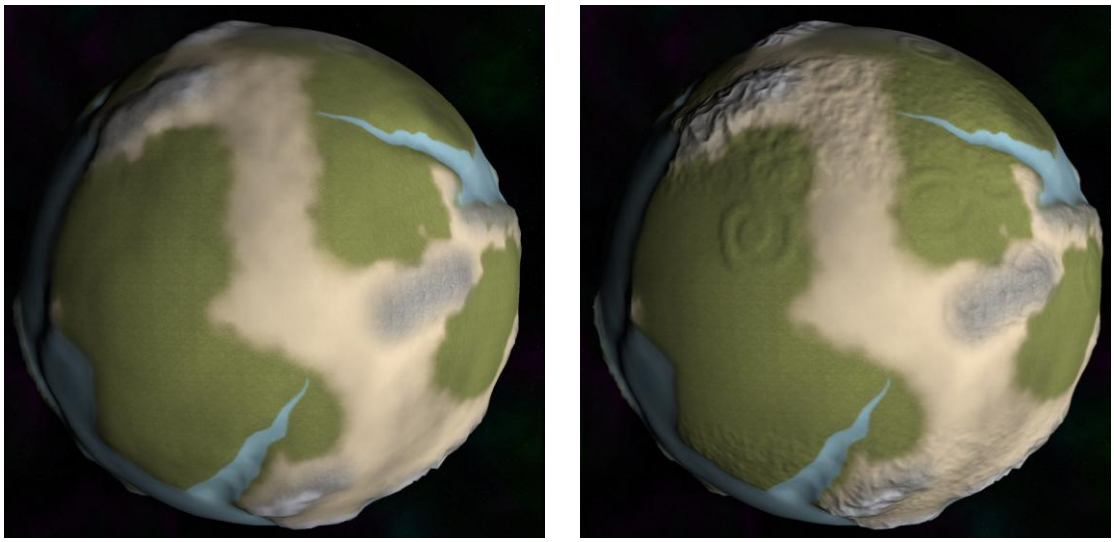


Figure 73: Global normal map in Unity, using object space normals

This still leaves another problem. One texture map responsible for an entire globe is not ideal. From further away everything may look fine, but as soon as one pixel in the texture must represent too many pixels on screen, the visual quality drops. This could be counteracted in a couple of ways: by making the texture very large, by using multiple textures or by blending multiple normal maps.

Making the texture very large, larger than 4096x4096, is unacceptable as it takes up too much visual memory. Cutting up the texture into multiple parts and only using the needed ones could have worked. However since the generator uses 3d chunks, instead of quasi 3d chunks (See subchapter: Chunking, Radial/Polar) the textures would not in all cases line up nicely with the texture. This means in certain cases a chunk could do with one texture and sometimes would need as much as 4 textures, which could create unpredictable frame rates and makes texture seams more likely.

Instead more detail could again be faked by adding an entire layer of normal maps on top. The larger normal map loses significance, the closer the camera gets towards the planet. The normal map does not have a high enough pixel density any more, to add visual detail, especially when extra geometry makes the normal map obsolete. To add more detail at this level, additional normal maps can be blended on top. These normal maps would be associated with the various terrain types. These normal maps would however present a limit of how much closer the camera can get. There is going to be no more geometry to represent these normal maps unless real-time tessellation is added. Additionally to work as tiling textures, these normal maps will have to be in tangent space.

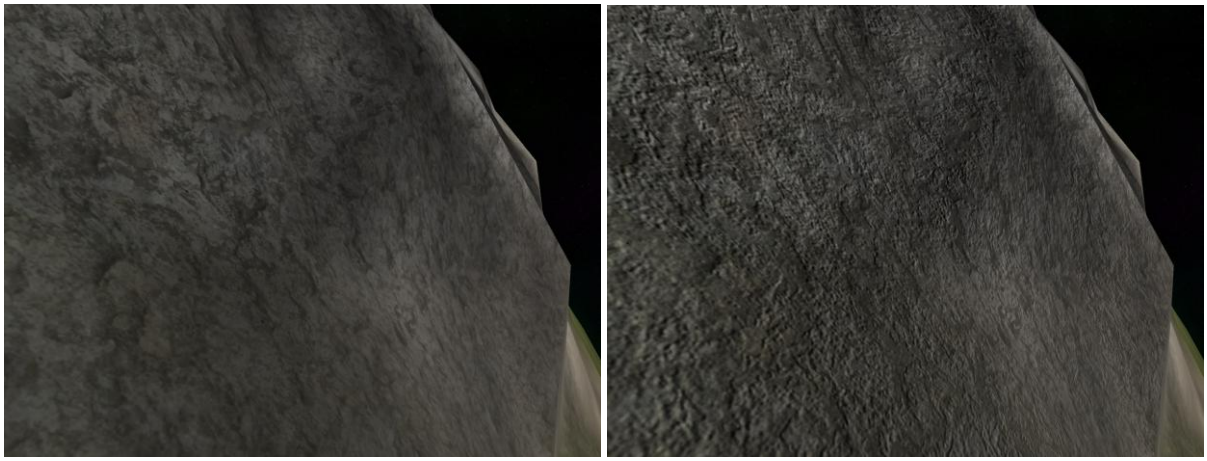


Figure 74: Detail normal map in Unity, using tangent space normals

When switching detail levels, (See subchapter: Chunking, Cartesian) the difference is small, but noticeable, because of both the silhouette and the lighting. To reduce the difference even further, the information of the lower detail level is saved for the higher detail level. This way the terrain shader can create a gradual transition between the two based on the distance to the camera. With a more traditional planet dynamic level of detailing, this data would have been saved inside a texture, because of the quasi-3d nature of their chunks (See subchapter: Chunking, Radial/Polar). However, because this generator allows for multiple height values for each spherical coordinate, tunnels for example, textures saving this data will not suffice. Instead the data is saved in the actual vertex attributes. This data consists of the vertex blend position and the vertex blend normal.

This immediately proved to be a problem since the amount of vertex attributes that are usable in Unity shaders are limited, they are: vertex position, vertex normal, 2 UV channels, vertex tangent and vertex color. Of these attributes at this point, only one UV channel and the vertex color were left unused. On top of that, the UV channel does not take in account the W coordinate, so there are only 2 floats to use. The color attribute has 4 channels: RGBA. This makes a total of 6 channels which is exactly the amount that is needed. Unfortunately the color attributes only have an accuracy of steps of 1/255. When a planet's radius is 10000 meters, which is the value used in the demo, one step equals 39 meters. This is way too inaccurate to define a vertex position, fortunately It is accurate enough so define a vertex normal.

To make this work, instead of using cartesian (X,Y,Z) coordinates, the vertex blend positions are saved as polar coordinates, this way the more accurate UV channels are used for the two polar rotations and the alpha channel from the vertex color is used to save the radius. the RGB values are used to save the (1+vertex blend normals)/2, this is done as normals range from -1 to 1 in each axis, whereas a vertex color only support values from 0 to 1. Using these workarounds mean the actual values for the blend position and normal need to be calculated back, this is done in real-time in Unity using a shader. It took a lot of iterations to find the best workaround, but eventually this is the one that is used.

All this data is transferred by using ray casts; The higher detail level takes the blend attributes from the larger, but lower detailed, level. To ensure that no rays are going past the lower detail level, the borders are extended slightly. After that the attributes are "encrypted" as described in the above paragraph. To test the data, a system was created to decrypt the data in the same way the real-time shader in Unity would do. With a simple slider the user is able to preview how the attributes blend.

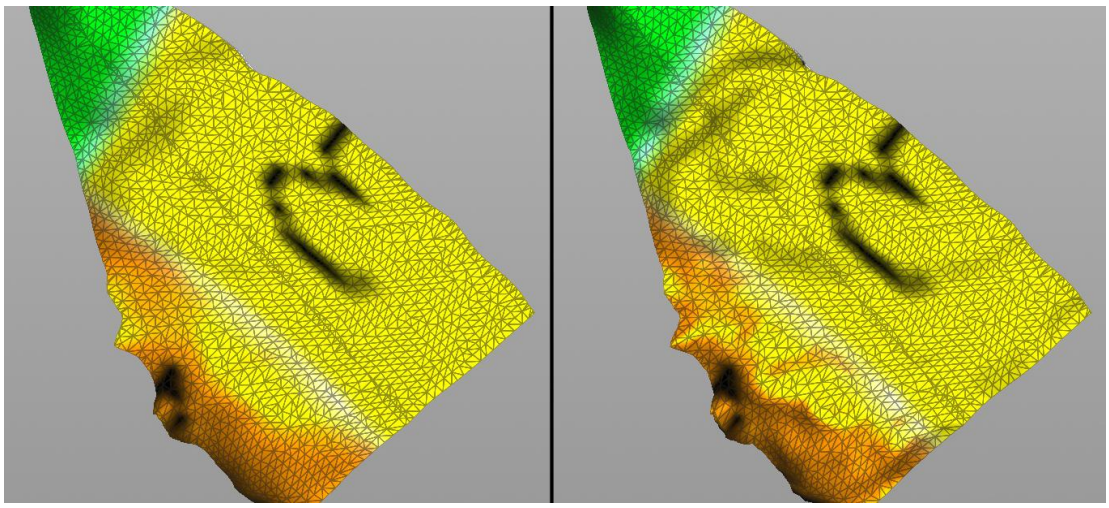


Figure 75: vertex blending from low to high detail

Optimization is a very important aspect of creating more complex procedures. There are two basic things that can be optimized: calculation time and memory. Calculation speed is mostly limited by the CPU, but the memory needed should not exceed the available RAM or Houdini will wait until new memory comes available. The latter happens when trying to store millions of polygons. While not likely to happen, running out of memory can cause Houdini to freeze or to never finish its calculations.

Houdini 12 features a performance monitor. When active, it records the amount of time and memory each node takes. This makes finding the bottlenecks in a procedure a lot easier. The performance monitor also makes it possible to test if optimizations are effective, do not make any difference or are actually counterproductive.

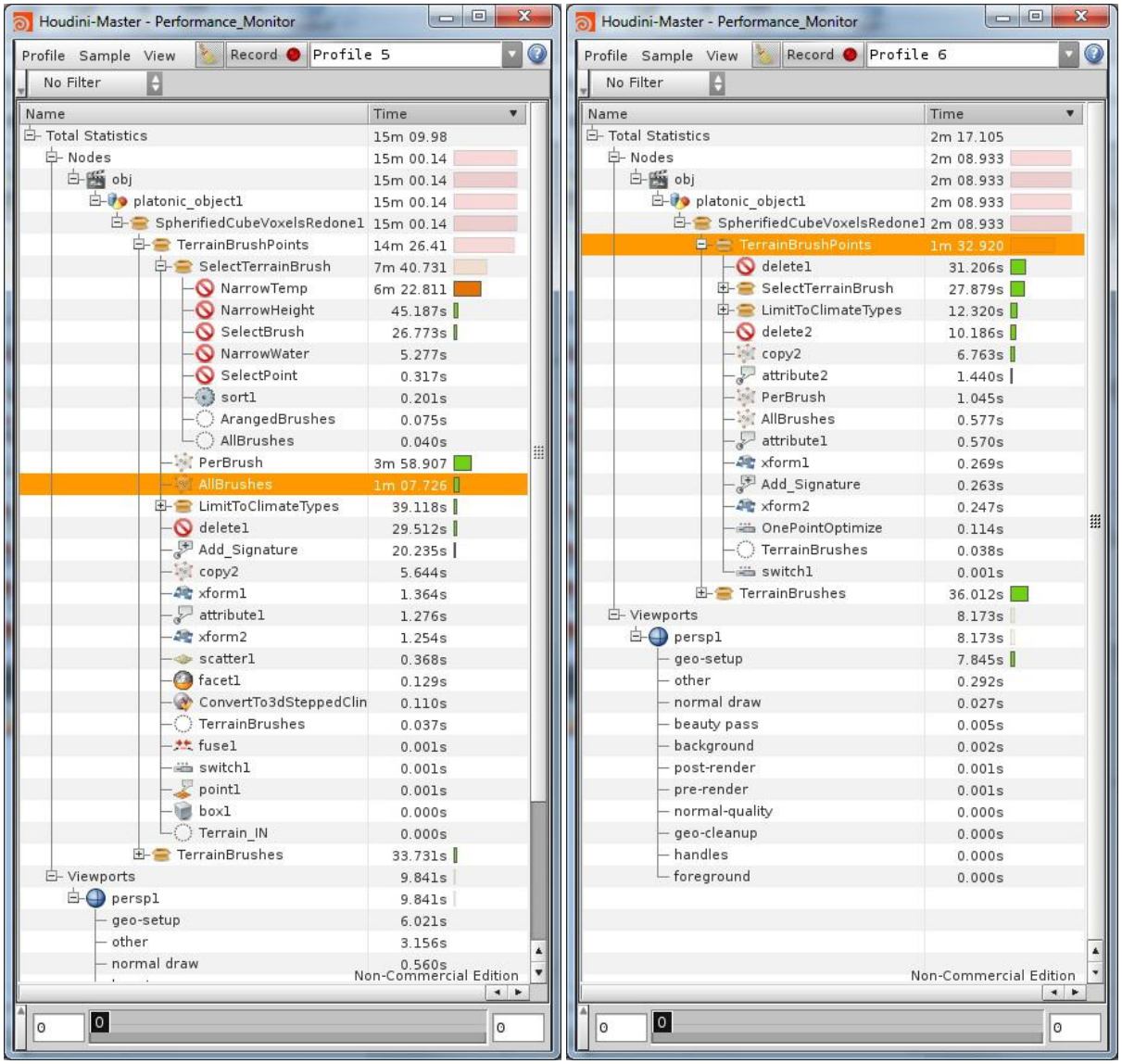


Figure 76: Houdini 12 Performance monitor, a linear part of the procedure from 7:41 minutes to 1:33 minutes

Memory usage can most easily be brought down by removing data that is no longer needed, such as group data and certain attributes. It can be brought down further by optimizing the variables used. For instance: if a 16 bit float can be used instead of a vector. If you only need one specific value, sometimes it is possible to use a detail attribute instead of point or primitive attributes. Reducing the needed memory may also slightly reduce the calculation times.

Calculation time can be divided into three categories: Real-time, short and long calculation times. Real-time can be defined as where a change is made, the result is visible in less than a second. Remember that anything that can be updated in real-time on a certain machine, can take a couple of seconds on a different machine. Short can be anywhere from a few seconds to a minute or two. Long calculation times can be defined as anything that takes more than two minutes. These three types require different kinds of optimizations.

Real-time calculations are nice to have and do not really require optimizations on their own. However when adding features to these calculations that make the calculation slower, it can be worthwhile to make it possible to temporarily turn these features off, to make editing faster as long as the end result is accurate enough.

Short calculation times can be optimized by checking the most "expensive" calculations. Expensive calculations are mostly calculations done per primitive, point or especially vertex. In Houdini the most expensive nodes used in the procedure are, in order: Attrib transfer, Poly cap, Hole, Volume operations, Clip, fuse and Delete in certain cases. Avoiding unnecessary use of the first three and disabling functionality within these nodes that you do not need is a good way to start. Another way to shorten the calculation is by removing primitives or points that are not necessary before the more expensive calculations. Figure 76 is an example of this kind of optimization.

Long calculation times are almost exclusively caused by combining a multitude of nodes, especially when they are in loops that are executed many times. When working inside loops, the most effective way of optimizing a procedure is tackling the most expensive nodes. For example: a loop is executed 100 times and takes 4 hours in total. When you manage to take off half a second per loop, that would save half an hour in total.

The planet creation procedure has linear as well as looping parts. It is best to focus optimizations on the looping parts as 100*10 seconds is of course more than 1*60 seconds. The amount of chunks that need to be rendered makes it worthwhile to save each possible second, especially on the higher detail levels. (See subchapter: Chunking, Cartesian) on the second detail level a maximum of 512 chunks can be calculated and on the third level a maximum of 4096 chunks. At this point saving one second for each chunk would save more than an hour. This view is unfortunately not entirely correct, as certain chunks will take more times than others. This is due to the varying number of polygons inside each chunk. Luckily there are more ways to reduce the time these kinds of calculations take.

For instance when a certain chunk appears to be empty, the 8 chunk's children are most likely empty as well and can be skipped altogether. Other steps can be skipped for certain chunks when the chunk for instance does not contain cave systems. These kinds of optimizations can be disabled for the final production render for slightly more accuracy.

By optimizing and fixing some bugs in the system, the calculation of 4096 chunks was at some point brought down to 5 hours, from an initial 20 hours. Later the calculation time had to be increased again, to increase the accuracy of the UVs (See subchapter: Texture Coordinates) and Chunk borders (See subchapter: Chunking, Cartesian). At the end, due to various optimizations and the introduction of VDB volumes, which are much faster than the original Houdini volumes, the entire export process was reduced to just over 6 hours. This includes textures, meshes and object points, along with all the required intermediary steps.

With all the different facets created, they still need to be exported, to both model files, textures and data files. The system required at least some form of automation as no one wants to export 20000 files manually. The first step to automation was to find formulas to make the progression through chunks linear. The first level goes from 0 to 7 and each level above that has levels 0-7 for each level above. In other words: 8^1 = 8 chunks 8^2 = 64, 512, 4096, 32768 etc. (See subchapter: Chunking, Cartesian) This means each detail level requires a slightly different formula to work as a linear sequence: 0-7 on level 0, (0_0 - 0_7), (1_0 - 1_7), (2_0 - 2_7) etc. on level 1.

For this I have found the following formulae:

$F stands for animation frame or iteration, ch(string) is an expression function in Houdini to query a channel.

Level0: floor($F/8^ch("LevelOfDetail"))%8
Level1: ((floor($F/8^(ch("LevelOfDetail")-1))%8)*(ch("LevelOfDetail")>=1))
Level2: ((floor($F/8^(ch("LevelOfDetail ")-2))%8)*(ch("LevelOfDetail ")>=2))
Level3: ((floor($F/8^(ch("LevelOfDetail ")-3))%8)*(ch("LevelOfDetail ")>=3))
etc.

"*(ch("LevelOfDetail")>=1)" multiplies the chunk value by 0 if the detail level is not at least 1.

Following this formulae, for $F = 51:

for LevelOfDetail = 0, The chunk is 3.
Note that the chunk is 3 for $F(int) = (8X+3)
for LevelOfDetail = 1, The chunk is 6_3.
for LevelOfDetail = 2, The chunk is 0_6_3.
for LevelOfDetail = 3, The chunk is 0_0_6_3.

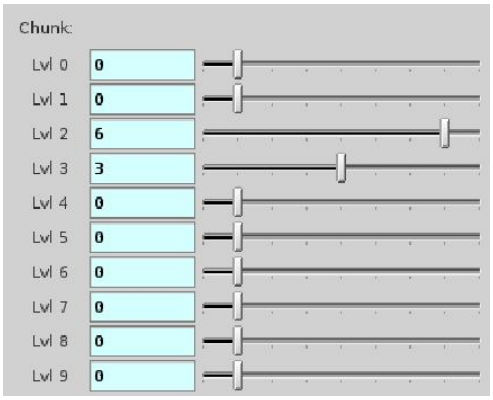


Figure 77: Chunk numbering Interface

This was a nice step in the right direction. With this in place it was possible to export everything as if it was an animation. If the "animation" was played and the correct node was visible, it would export a file each frame. The problem was that for each separate level of detail, you needed to manually edit the animation length (0 to 7, 63, 511 or 4095) and change the level of detail and start the animation again. This was needed for each chunked facet of the generation. This was fine for testing the procedure, but unacceptable for a production level generator.

To improve on this process, a system was setup in the /out context in Houdini, where renderers and exporters reside.

When the user has created the planet template (See subchapter: Climate generation). The procedure is able to cut the mesh into chunks in "GEO_PreProcessSET" and create the water masses in "MESH_WaterMasses", but these are not yet needed at this point. The procedure is also able to generate normal maps and other texture maps at this point, using "PNG_PostProcessSET".

When all base chunks have been created, UVs and noise can be applied to them in "GEO_MainProcessSET", note that the output of "GEO_PreProcessSET" is saved separately to disk, so "GEO_MainProcessSET" can be re-exported without having to redo the previous step as well. This applies to all facets of the generation process.

After the noise and UVs are applied, the procedure can create the vertex blend data (See subchapter: Vertex Blend Data Generation) export the actual mesh files in "MESH_PostProcessSET". It can also create collision meshes in "MESH_CollisionSET". Finally it can create the object point data in "TXT_PropPlacementPoints".

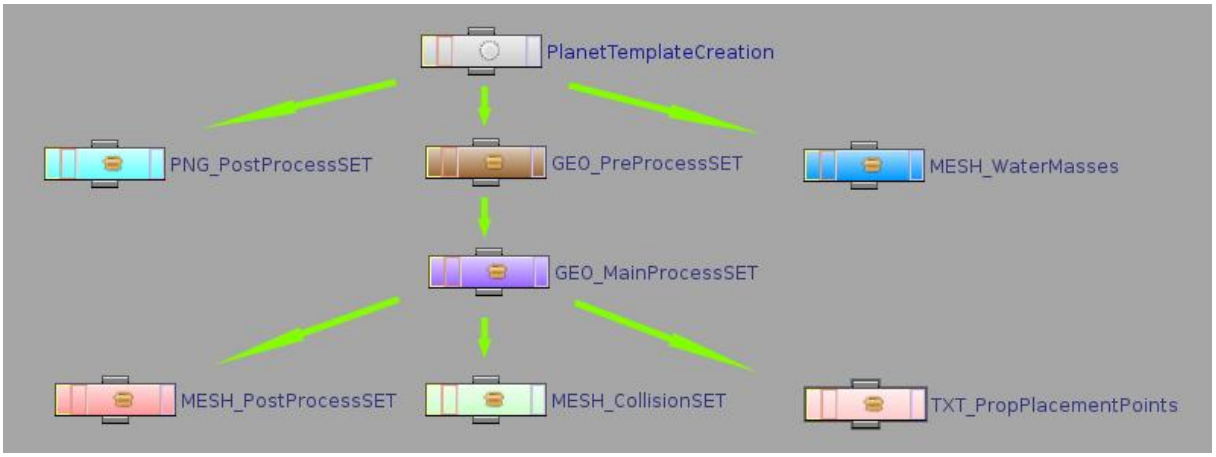
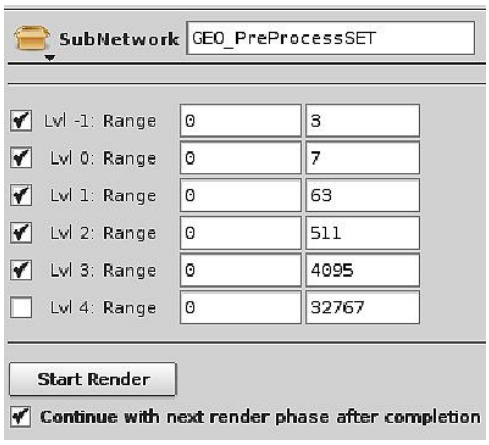


Figure 78: Export Dependencies

It was difficult to get the complex dependency setup using normal node connections. This is because the animation length differs for each detail level as well as some other factors. The dependency is also not entirely linear.



Instead the dependencies were enforced by some Python code. Additionally the python code automatically sets the level of detail and updates the animation lengths. It also allows to do a partial re-export, in the case of a crash for instance.

Finally the code allows for sequencing exporting the various facets. This allows the user start the "render farm" and view the result after a couple of hours, depending on the machine, but without having to intervene after the export is started.

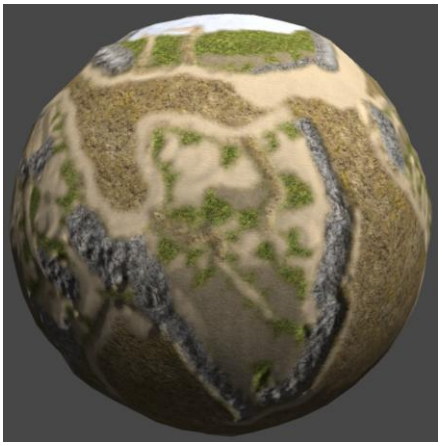
Figure 79: Export selection interface - GEO_PreProcessSET

SHADER CREATION

To color and manipulate vertices (See chapter: Planet Creation/Vertex Blend Data Generation) inside render engines additional code has to be written. More specifically shader code, shaders are handled by the GPU instead of CPU, which makes the code slightly different, but allows for real-time rendering.

Unity's built-in shaders were not sufficient, as they are quite generic. Instead very specific shaders were needed to display everything as intended.

TERRAIN SHADER



The basic terrain shader rendered multiple color textures, blended together according to two blending textures. These textures are climate masks rendered out in Houdini (See chapter: Planet Creation/Texture maps) as well as one global tangent space normal map. This shader was used before Unity4, meaning there was no access to DX11 shaders yet. Note that it was not possible to even render water within the same shader at this point. There were also texture seams visible at this point.

Figure 80: Basic Terrain Shader (DX9)

When Unity4 came out, the shader could be significantly upgraded, as with DX11 more texture samplers could be used. This way there was room to use per-terrain-type normal maps. To allow for even more textures, the blend masks were saved inside the appropriate color maps. As a bonus the order of textures was no longer fixed and no longer hard to keep track of. By changing the blending order, the user can also determine what is blended on top of what, for instance snow should be blended on top of mountains, or the snow would not be visible on mountain tops.

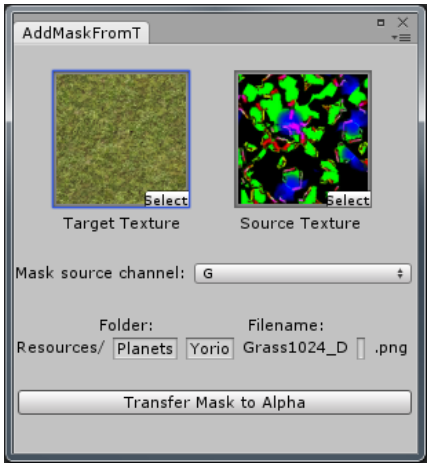


figure 81: Texture mask linker

To make assigning these masks easier and not require the use of Photoshop, a small tool was created in Unity. The allows to pick a color map and mask texture. After that one of the RGBA channels can be picked and this is transferred to the A channel of the target texture. Then the texture with the mask is saved as a copy with the specified name.

Some additional tricks have been employed to further improve the visual quality. It uses a secondary, real time calculated UV map, as the vertex attributes were already maxed out (See chapter: Planet Creation/Vertex Blend Data Generation). With this UV channel the texture is blended along the original UV seams, eliminating those original seams altogether. The below code snippet generates polar UVs with the seam at the top (0.5π).

```
float Pi = 3.1415926535;
float4x4 rotationMatrix = float4x4(cos(0.5*Pi), -sin(0.5*Pi), 0.0, 0.0,
                           sin(0.5*Pi), cos(0.5*Pi), 0.0, 0.0,
                           0.0, 0.0, 1.0, 0.0,
                           0.0, 0.0, 0.0, 1.0);


float3 vertexR = mul(rotationMatrix,v.vertex.xyz) ;
float radius = distance(vertexR.xyz, mul(_Object2World, float3(0,0,0)));
float2 polarUV = float2(atan2(vertexR.y,vertexR.x),acos(vertexR.z/radius)*2);
polarUV = polarUV*float2(0.1,0.1);

o.meshUV.xy = (v.texcoord.xy);
o.meshUV.zw = (polarUV);
```

Figure 82: calculated secondary UV channel code.

It also blends the textures with itself according to a noise map (render cloud texture). This way the tiling factor of the textures can be very high before displaying visible tiling. This noise map, also add darker spots in the texture, further reducing the visible tiling.

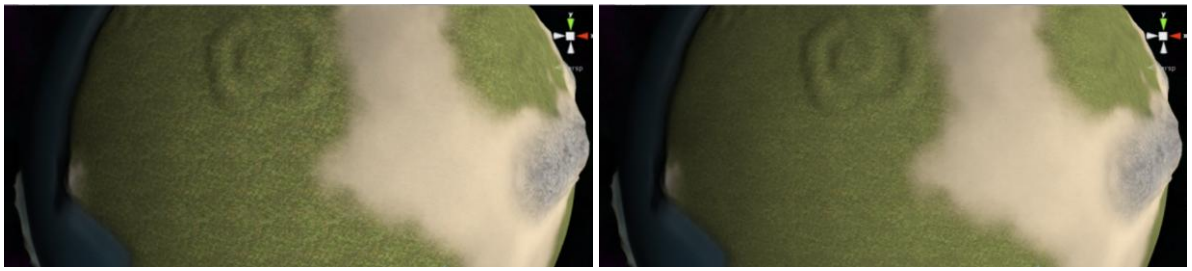
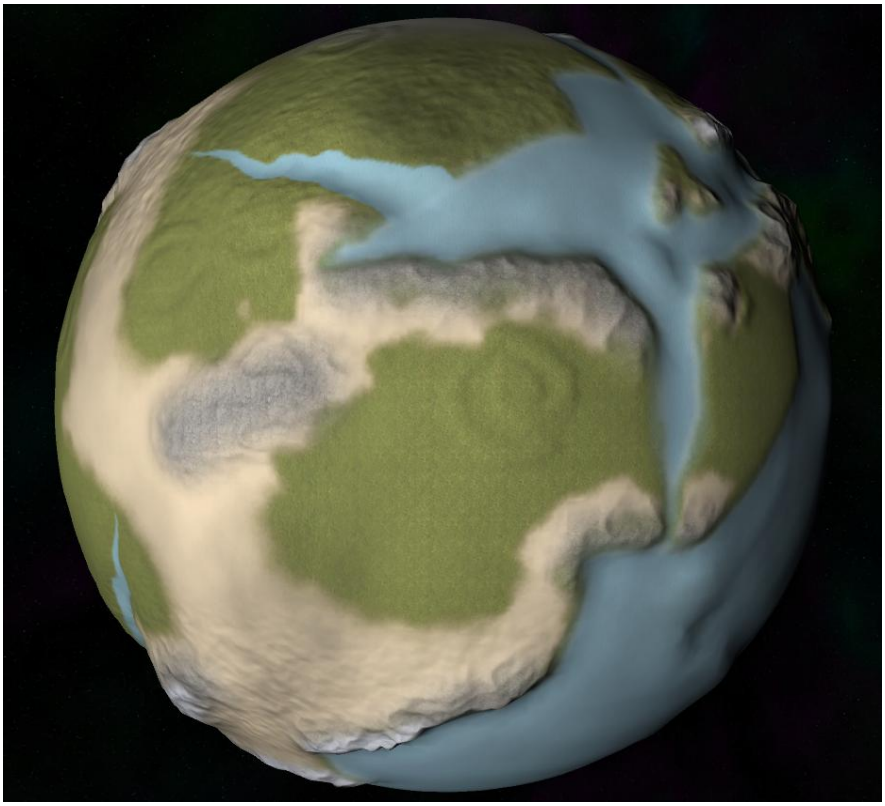


Figure 83: Without and with texture self blending



Finally gloss was added to the shader, adjustable per terrain type. The ocean parts can be blended between water and ocean floor. This way the actual water mesh can be hidden from far away, avoiding Z-Buffer issues. Note that in the final shader, for the global normal map, an object space normal map has been used instead of a tangent space normal map. (See chapter: Planet Creation/Normal maps)

Figure 84: Final terrain shader (DX 11)

To reduce the visible difference between levels of detail (See chapter: Planet Creation/Chunking, Cartesian) when switching between them, vertex blending has been added. Vertices are moved inside the shader between two positions. One set of positions are the standard positions and the other set emulates the surface shape of the mesh, one detail level lower. These positions are saved inside the vertex attributes (See chapter: Planet Creation/Vertex blend data generation)

```
float radius0 = distance(v.vertex.xyz,mul(_Object2World, float3(0,0,0)));

float4 polar = float4((v.texcoord1.x*Pi*2),(v.texcoord1.y*Pi),((v.color.a-0.5)*0.05)+radius0,0);
float4 fromPolar = float4(sin(polar.y)*cos(polar.x)*polar.z,(sin(polar.y)*sin(polar.x)*polar.z)*-1,cos(polar.y)*polar.z,0);
fromPolar = mul(_World2Object,fromPolar);

float dist = distance(mul (_Object2World, v.vertex).xyz, _WorldSpaceCameraPos.xyz);
float distRange = clamp((_StartB-dist)*(1/(_StartB-_EndB)),0,1);
float distWRange = clamp((15000-dist)*0.0001,0,1);

float3 lowNormals = float3(v.color.r,v.color.g,v.color.b);
lowNormals *= float3(-2,2,2);
lowNormals -= float3(-1,1,1);
lowNormals = normalize(lowNormals);

if(_EnableVert)
{
    v.vertex.xyz = fromPolar + ((v.vertex.xyz-fromPolar)* distRange);
    v.normal.xyz = lowNormals + ((v.normal.xyz-lowNormals)* distRange);
}
```

Figure 85: Vertex Blending code

These two position sets are blended into each other according to the distance to the camera. The distance margins are calculated per detail level (See chapter: Implementation/Chunk loading system) . First however, first the blend position must be converted back from polar to cartesian coordinates. This is done in the first 4 lines in figure 85. After that the distance is measured and remapped to a 0-1 range, based on the "_StartB" to "_EndB" distances. Below that a separate blending range is calculated for the blending between ocean surface and ocean floor textures. In the next block the blend normals are remapped from 0 to 1 to -1 to1. Finally in the last block the blended positions and normals are applied.

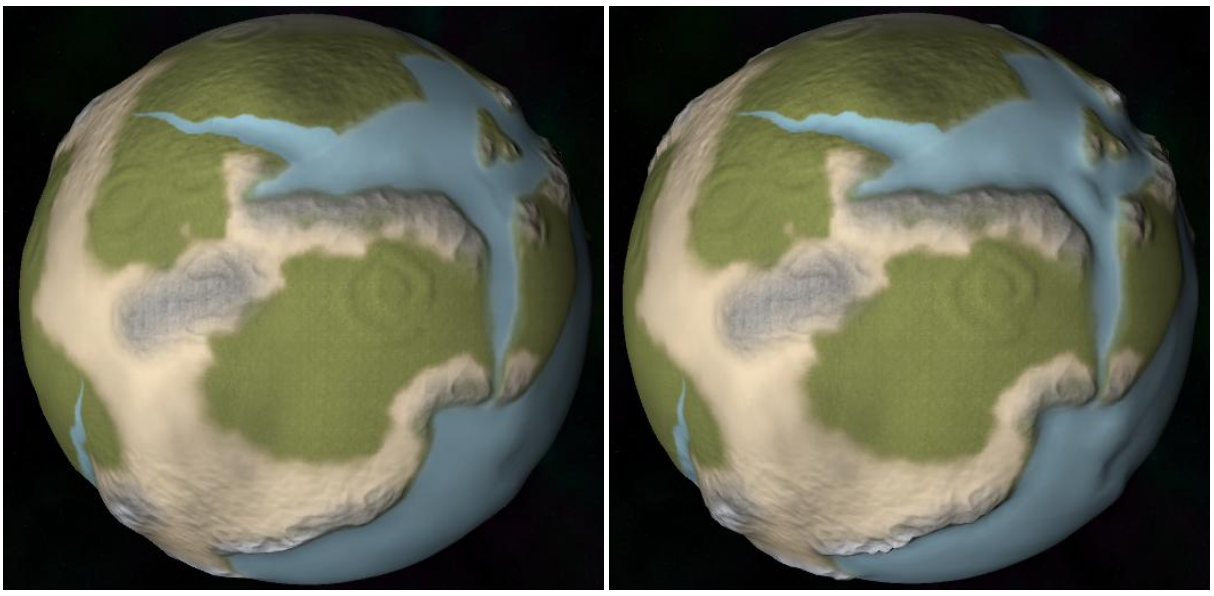


Figure 86: Vertex blend low & high

The difference may not be immediately apparent, but if you closely look at the mountainous areas and the silhouette, you can see the difference. Basically the difference between the planets in figure 86 would be the amount of "popping" you would see when switching to a higher level of detail without this feature. With the blending feature, the shader gradually blends the left image towards the right.

Water is present in the terrain shader, however this water is only to be used for far away. For closer up a separate water mesh is used along with its own shader. This separate water mesh is not used from afar as Unity is not accurate enough. The relatively small difference in distances between the ocean floor and ocean surface from far away causes sorting issues. Instead when getting closer to the planet, the terrain shader blends the ocean surface texture towards the ocean floor texture. Meanwhile the separate water meshes are getting more opaque.

There are four water shaders: Ocean above surface, Ocean below surface, River above surface and River below surface. The shaders for below the surface differ slightly in render order and do not fade out near the coast or river banks, but get more transparent when the camera gets closer. Otherwise they are identical to the other shaders. The river and ocean shaders differ in their animation.

The ocean shader has animated UVs. In figure 87 you can see 5 UV modifiers: FlowUV1A, FlowUV1B, FlowUV2A, FlowUV2B and FlowCoast.

```
float2 FlowUV1A = IN.meshUV.xy;
float2 FlowUV1B = FlowUV1A;
FlowUV1A.x += (_Time.x*0.1+(0.0015*sin(_Time.x+5)))*_FlowSpeed;
FlowUV1A.y += (_Time.x*0.1+(0.0015*sin(_Time.x+10)))*_FlowSpeed;

FlowUV1B.x -= (_Time.x*0.11+(0.0007*sin(_Time.x+8)))*_FlowSpeed;
FlowUV1B.y -= (_Time.x*0.11+(0.0007*sin(_Time.x+7)))*_FlowSpeed;


float2 FlowUV2A = IN.meshUV.zw;
float2 FlowUV2B = FlowUV2A;
FlowUV2A.x += (_Time.x*0.11+(0.0015*sin(_Time.x+5)))*_FlowSpeed;
FlowUV2A.y += (_Time.x*0.11+(0.0015*sin(_Time.x+10)))*_FlowSpeed;

FlowUV2B.x -= (_Time.x*0.09+(0.0007*sin(_Time.x+8)))*_FlowSpeed;
FlowUV2B.y -= (_Time.x*0.09+(0.0007*sin(_Time.x+7)))*_FlowSpeed;

float2 FlowCoast = IN.coastUV;
float CoastBlend = pow(clamp(FlowCoast.y+0.6,0,1),2);
FlowCoast.y += _Time.x*_FlowSpeed*2;
```

Figure 87: Ocean UV animation code

FlowUV2A and B are for the areas where the standard UV set has seams. Like the terrain shader this is solved by using a real-time calculated secondary UV set. Per UV set the texture is animated in two ways (A and B) and blended on top of each other to get waves in both directions. Flow Coast is a tertiary UV set (See chapter: Planet Creation/Water) to get coastal waves. These are all blended on top of each other.

The river shader also features animated UVs, only they go downstream instead of both ways. It also has an UV animation for the riverbanks, Like the Ocean shader has for the coasts. However due to the nature of the UV mapping of the rivers only one UV set is needed to accomplish this. Note that the river bank UV animation is masked by the UVs itself in the last two lines.

```
float2 flowUV = IN.meshUV.xy;
flowUV += float2(0,_Time.x*_FlowSpeed);

float2 flowUVB = flowUV + float2(0,_Time.x*0.5*_FlowSpeed);
float2 flowUVMacro = flowUV - float2(0,_Time.x*0.5*_FlowSpeed);

flowUV.x += ((0.05*sin(_Time.x*10))+(0.0015*sin(_Time.x*64)))*_FlowSpeed;

float2 flowUVRB1 = IN.meshUV.xy;
float2 flowUVRB2 = IN.meshUV.xy;

flowUVRB1.x += _FlowSpeed*_Time.x*5;
flowUVRB2.x -= _FlowSpeed*_Time.x*5;

float Pi = 3.1415926535;
float RiverBankMask = 1-sin(IN.meshUV.x*Pi);
```

Figure 88: River UV animation code

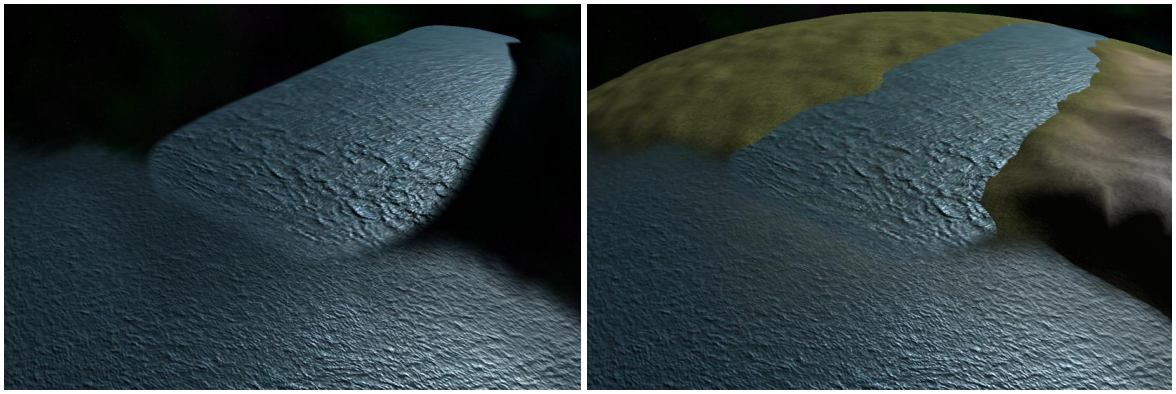


Figure 89: Water shader in Unity

The Earth's sky is mostly blue. This is because of particles in the atmosphere distorting the sunlight slightly. This is called Atmospheric scattering. The longer the distance the sunlight has to travel through the atmosphere, the more red it gets. This is why the sky is more red during sunrise and sunset. (Wikipedia, 2013f)

To emulate this effect accurately in a render or game engine a lot of shortcuts need to be taken to keep the frame rate high. This is because the actual formula to calculate this for earth's atmosphere is quite complicated and requires mathematical operations that are too taxing on GPUs to be used for real-time rendering. To make this scattering faster there are multiple things that can be done. To start the color can be calculated per vertex instead of per pixel. Additionally there are multiple ways to simplify the equation that calculates the atmospheric color (O'Neal, 2005). Explaining the exact workings atmospheric scattering are beyond the scope of this project. In short it compares the distance of vertices to the camera with the distance a light ray travels to reach this vertices.

```
float3 v3Pos = v.vertex.xyz;
float3 v3RayB = v3Pos - _v4CameraPos.xyz;
float fFar = length(v3RayB);
float3 v3Ray = v3RayB / fFar;

float fNear = getNearIntersection(_v4CameraPos.xyz,v3Ray,_fCameraHeight2,_fOuterRadius2);
float3 v3Start = _v4CameraPos.xyz + v3Ray * fNear;
fFar -= fNear;

float v3StartAngle = dot(v3Ray,v3Start) / _fOuterRadius;
float v3StartDepth = exp(-1.0 / _fScaleDepth);
float v3StartOffset = v3StartDepth * expScale(v3StartAngle);
float fSampleLength = fFar / _Samples;
float fScaledLength = fSampleLength * _fScale;
float3 sampleRay = v3Ray * fSampleLength;
float3 samplePoint = v3Start + sampleRay * 0.5f;

float3 frontColor = float3(0,0,0);
float3 attenuate;

for (int i = 0; i < 1; i++) {
  float height = length (samplePoint);
  float depth = exp(_fScaleOverScaleDepth * (_fInnerRadius - height));
  float lightAngle = dot(_v4LightDir.xyz, samplePoint) / height;
  float cameraAngle = dot(-v3Ray, samplePoint) / height;
  float scatter = (v3StartOffset + depth * (expScale (lightAngle) - expScale (cameraAngle)));

  attenuate = exp(-scatter * (_cInvWaveLength.xyz * _fKr4PI + _fKm4PI));
  frontColor += attenuate * (depth * fScaledLength);
  samplePoint += sampleRay;
}
```

Luckily there were others that previously attempted a real-time implementation of a atmospheric scattering shader in Unity and shared this collaboration on the Unity forum (PRD et al., 2013). The final shader used in the project is a modified version of the shader taken from this collaboration.

Figure 90: Atmospheric scattering code snippet

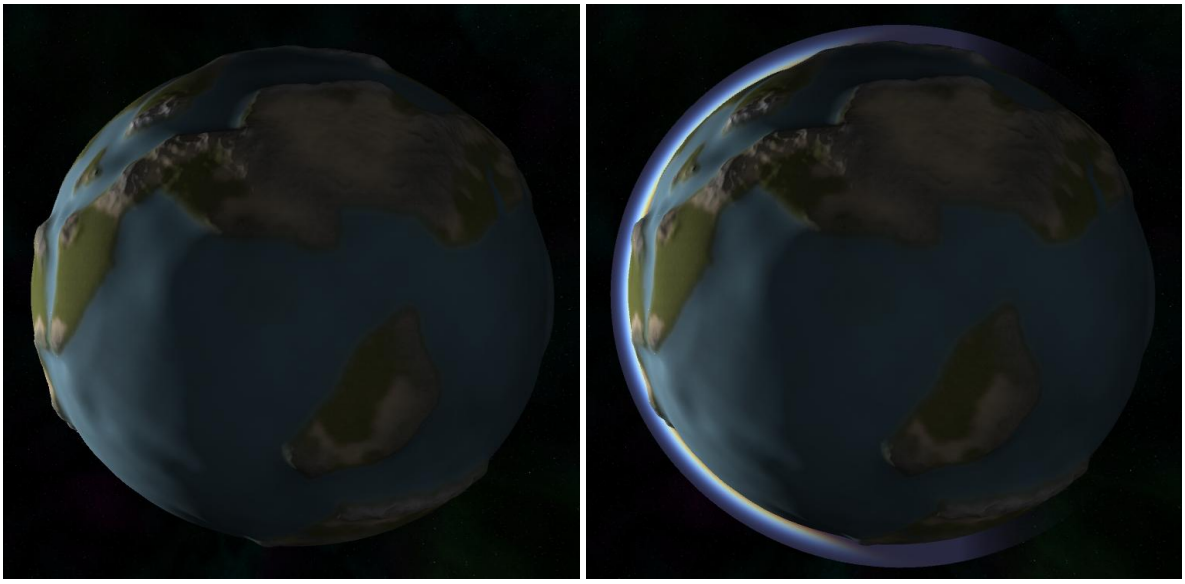


Figure 91: Without and with atmosphere

Note that away from the sun the atmosphere is more transparent simulating night. This is one of the additions made to the shader.

Along with the atmospheric scattering clouds were added as a separate layer. Just like the water shaders, the clouds consist of two shaders, one for from outer space and another for from within the atmosphere. Using one 2-sided shader could have been an option, but this resulted in sorting issues.

The clouds follow the climate generated (See chapter: Planet Creation/Climate generation). This is done by rendering the final planetary moisture and wind directions to a texture.

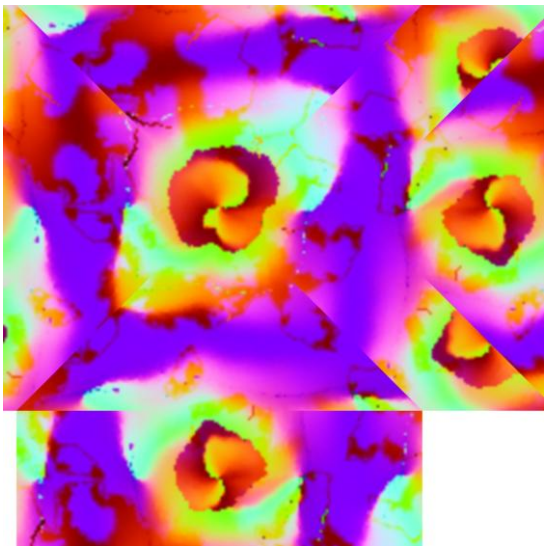


Figure 92: Cloud mask

In the end the actual wind directions were not used in the shader as it did not produce visually pleasing results. The moisture, which is the blue channel of the texture is used however. It increases the chances of clouds appearing over water and tropical climates, where it reduces the amount of clouds over deserts and polar regions.

The actual clouds are produced by multiplying the multiple channels of a render cloud texture with each other. All channels have animated UVs with different speeds. This makes clouds appear, move slightly and disappear. By multiplying the separate channels of the texture the clouds get more varied and get more contrast. After this animation, the clouds are multiplied with the blue channel of the mask in figure 92. Finally some inversed Fresnel is applied to fade out the clouds on shallow angles, this way the transition through the atmosphere looks more fluent. The clouds also fade out when the camera gets very close, to the same effect.

```
float2 t1 = IN.meshUV.xy;
float2 t0 = IN.meshUV.zw;
half4 Manager = tex2D (_CloudManager,IN.meshUV.xy);

t1.x += _Time.x*_Speed;
t1.y += _Time.x*_Speed;
t0.x += _Time.x*_Speed;
t0.y += _Time.x*_Speed;
half4 Tex2DA0 = tex2D (_Diffuse, t0*_Scale);
half4 Tex2DB0 = tex2D (_Diffuse, t1*_Scale);

t1.x -= _Time.x*_Speed*2;
t1.y -= _Time.x*_Speed*2;
t0.x -= _Time.x*_Speed*2;
t0.y -= _Time.x*_Speed*2;
half4 Tex2DA1 = tex2D (_Diffuse, t0*_Scale*2);
half4 Tex2DB1 = tex2D (_Diffuse, t1*_Scale*2);

t1.y += _Time.x*_Speed*2;
t0.y +=  Time.x* Speed*2;
half4 Tex2DA2 = tex2D (_Diffuse, t0*_Scale*4);
half4 Tex2DB2 = tex2D (_Diffuse, t1*_Scale*4);

t1.x += _Time.x*_Speed*2;
t1.y -= _Time.x*_Speed*2;
t0.x += _Time.x*_Speed*2;
t0.y -= _Time.x*_Speed*2;
half4 Tex2DA3 = tex2D (_Diffuse, t0*_Scale*8);
half4 Tex2DB3 = tex2D (_Diffuse, t1*_Scale*8);

half cloudsA = Tex2DA0.x*Tex2DA1.y*Tex2DA2.z*Tex2DA3.w;
half cloudsB = Tex2DB0.x*Tex2DB1.y*Tex2DB2.z*Tex2DB3.w;
```

Figure 93: Cloud animation code

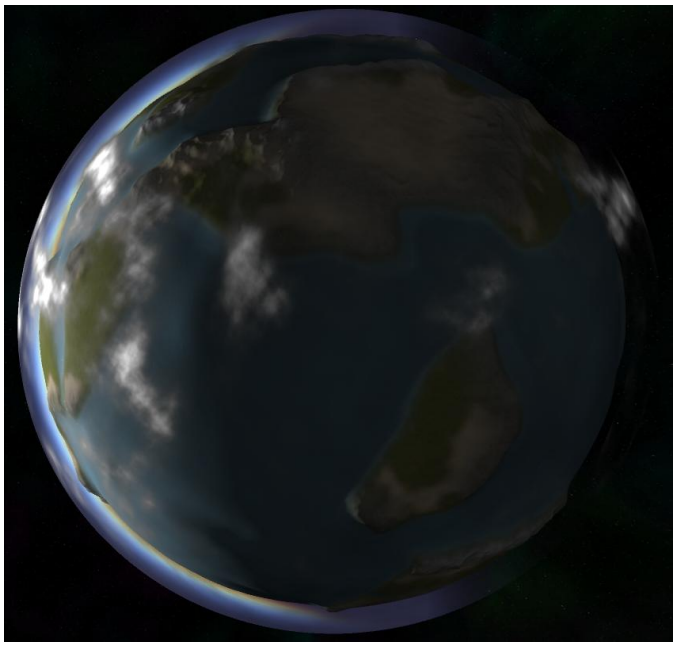
```
float BlendP0 = tex2D(_Mask,IN.meshUV.xy).a;
half LerpP0 = lerp(cloudsB,cloudsA,BlendP0);

float Fresnel0= acos(dot( normalize(float3(IN.viewDir.x, IN.viewDir.y,IN.viewDir.z)), float3(0,0,1)))/3.1415926535;
Fresnel0 = 1-clamp(pow(Fresnel0,_RimPower)*_RimBonus,0,1);

float Alpha = LerpP0*LerpP0*_Strength*(0.5+0.5*Manager.z)*IN.fade*Fresnel0;
o.Alpha  = Alpha;
```

Figure 94: Cloud inversed Fresnel and alpha code

Note that "LerpP0" blends the two UV sets together, just like the terrain shader does. "IN.fade" contains the blending value for the opacity from the camera distance, again just like the terrain shader does as described in vertex blending.



Note the absence of clouds above the desert and polar areas in figure 95. The clouds move along the atmosphere, combine to form denser clouds and fade away when becoming to sparse.

Figure 95: Clouds from space

Figure 96: Without and with clouds from within the atmosphere at noon

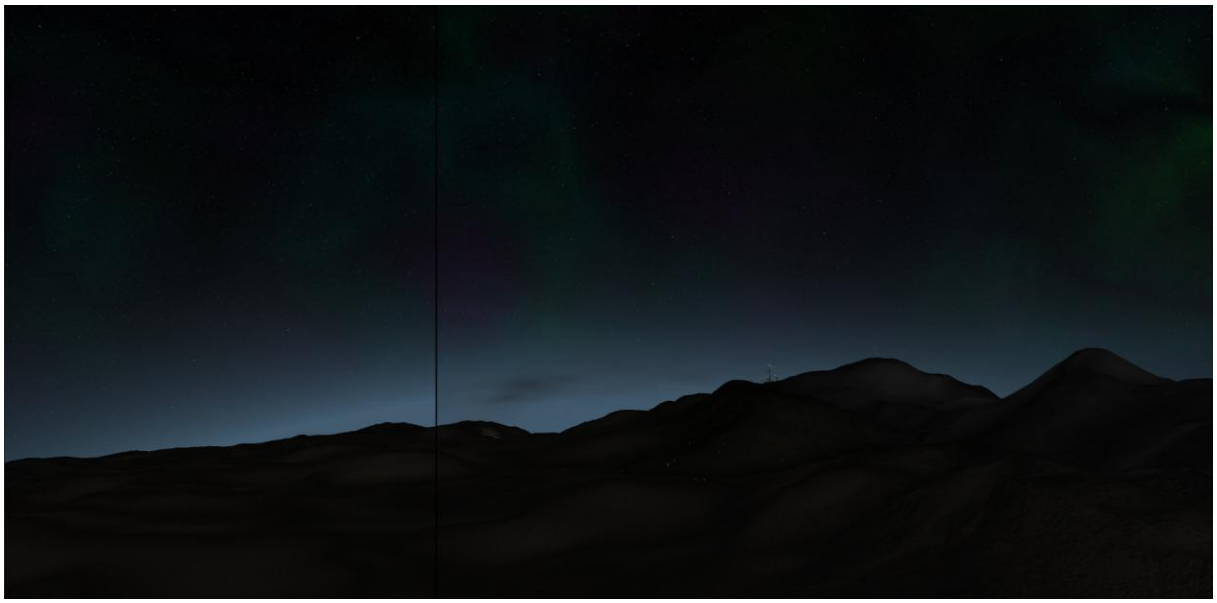


Figure 97: Without and with clouds from within the atmosphere at dawn

The trees and plants used in the demo were taken from two sources: The palm trees and deciduous trees are created by "killst4r" and were taken from turbosquid.com. The other plants and trees have been generated by using "tree[d]" by Frecle ApS.

Figure 98: Trees by "killst4r"

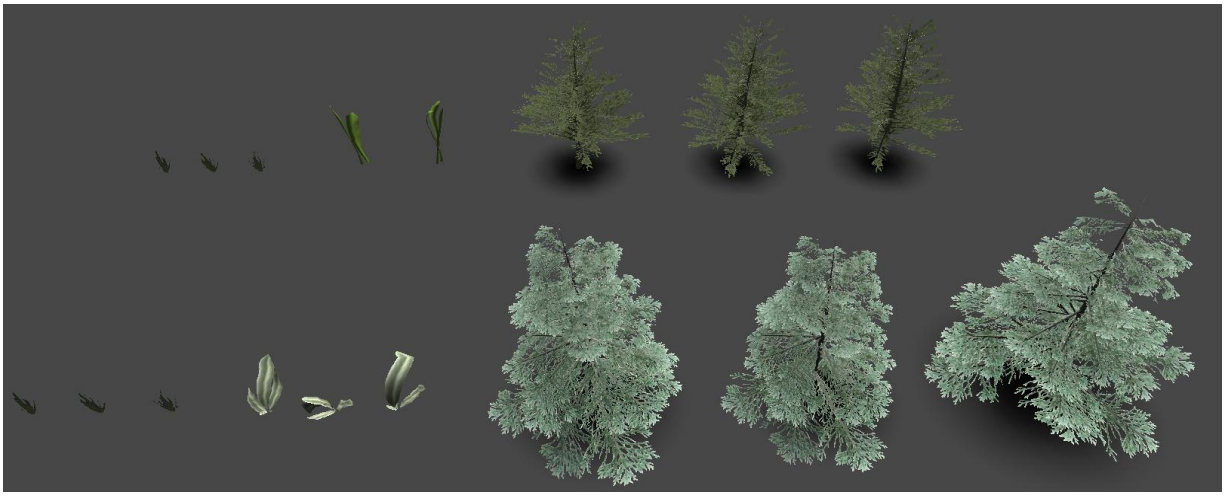


Figure 99: Trees and plants generated with "tree[d]"

For these props five different shaders have been used: A tree trunk shader, a leaf shader, a seaweed shader, a grass shader and a shadow shader. All these shaders have one thing in common: they are scaled depending on their distance to the camera. Objects that are spawned start out very small and grow larger when the camera gets closer. This way the objects do not "pop" into existence but seem just further away than they actually are until the camera gets closer.

```
float distRangeV = clamp((_StartBV-(distance(mul (_Object2World, v.vertex).xyz, _WorldSpaceCameraPos.xyz)))*(1/(_StartBV-_EndBV)),0,1);
float3 minOffset = float3(0,0,0);

v.vertex.xyz = minOffset + ((v.vertex.xyz-minOffset)* distRangeV);
```

Figure 100: Terrain prop scaling code

Apart from this, the leaf and seaweed shaders have additional vertex animation, to emulate wind and water movement. The seaweed is animated slightly different, the x and z offset is multiplied by the height value of the vertex. This way the seaweed moves more further away from the bottom.

```
float x = v.vertex.x+(_Time.x*_AnimationSpeed);
float z = v.vertex.z+(_Time.x*_AnimationSpeed);

float3 animOffset = v.vertex.xyz+(v.vertex.xyz*float3(cos(x),0,cos(z))*_AnimationStrength);
```

Figure 101: Leaf shader animation code

The procedure's visuals can for the most time be checked within Houdini itself, however the visuals may vary between how different render and game engines handle 3D assets. For instance certain imperfections can be either more or less apparent in different engines.

A personal game prototype, created in Unity3D, has been used to test and showcase the procedure. The testing of the procedure outside of Houdini was limited to the Unity3D engine.

Apart from it being very gratifying to see the procedure working as intended, it also gave direction on what to add or improve next. For instance the chunk borders and UVs underwent multiple iterations, each time checking the result in Unity.

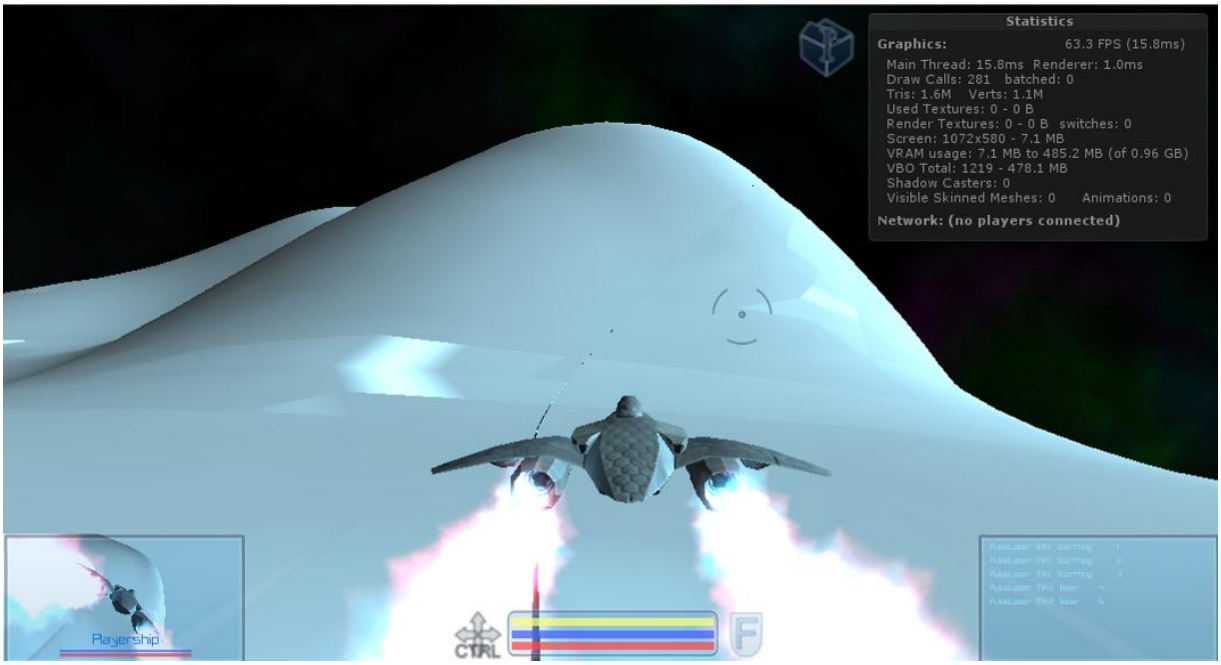


Figure 102: Initial imperfections in the export, visible gaps and lighting artifacts due to UV problems.

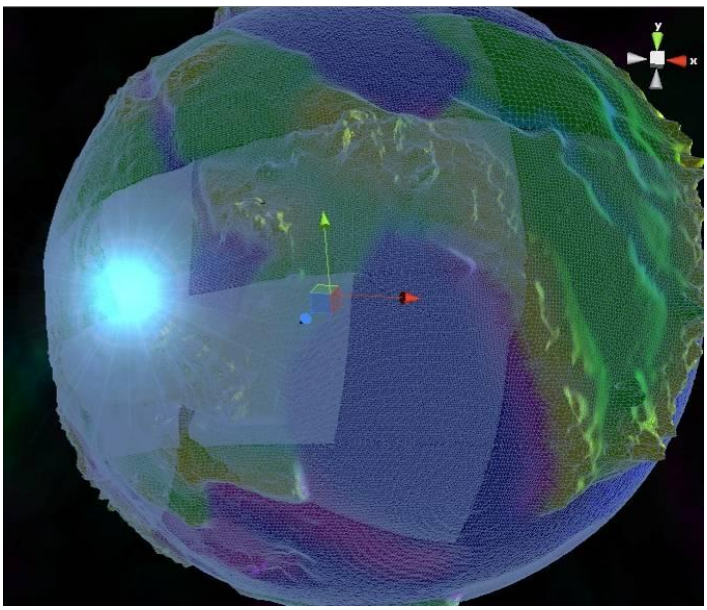


Figure 103: Chunking working in Unity3D.

Unity loads and unloads the various levels of detail. Some scripting was needed to accomplish this. First all the files are loaded into the scene, get the right data and all but the lowest resolution is made invisible. When the game is running, the visible chunks check the distance to, and the direction of the camera. Using this information the chunks are switched to the appropriate detail each time. The camera is located at the blue light to the left of the image. Note that the blue colored areas are marking the ocean floor, not the water surface.

After creating all the chunks they need to be loaded into the game/render engine, in this case Unity. In this chapter's introduction is explained in short how this works, this chapter goes into more detail. The loading is divided into two steps: first everything is loaded into a level file and saved. Secondly, during playtime all this data must be loaded in and out to make sure the amount of data is not overflowing and the frame rate is decent. For the first part a tool was written for Unity.

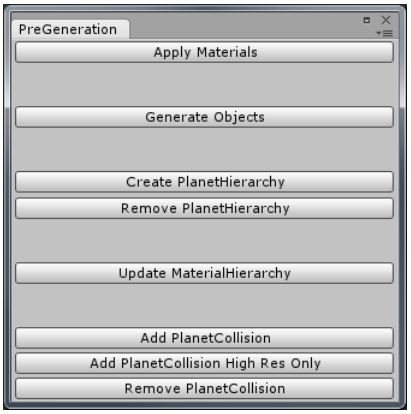


Figure 104: Pre Generation interface

In the middle is the option "Create PlanetHierarchy". After placing the root of the planet (the lowest detail level), pressing this button will the editor look up all the chunks and assign them as children, grandchildren, etc. according to their detail levels. With the resulting hierarchy, it is easier to build a system to keep track of which chunks should be visible during playtime. During the generation several values, such as final chunk size and polar position are measured, these are later used in the materials. Finally, the hierarchy is also cleaned and unused components, such as animators are removed.

When creating the hierarchy, the basic materials are also created, using the root material as a template. These materials contain the values that determine the amount of vertex blending (See chapter: Planet Creation/Vertex Blend Data Generation) for a smoother transition when switching chunks. To properly set this data, press "Update MaterialHierarchy", these values are calculated from the final measured values.

After that collision can be added. For this there are two options: a dynamic collision by using "Add PlanetCollision" which allows for switching collision along with the chunks. or a static collision by using "Add PlanetCollision High Res Only" With this option the highest resolution collision is loaded at all times, and lower resolutions are never loaded. Because of the nature of Unity, the latter option was surprisingly faster, where the dynamic collision caused frame hangs each time the overall collision was updated during playtime.

During playtime all enabled chunks are evaluated each frame: When the camera gets closer than a certain value a chunk is disabled and the child chunks are enabled. Likewise when the camera goes further than a certain value for all chunks that share the same parent, these chunks are disabled and the parent is enabled. Beside the distance, the polar direction and position relative to the camera's rotation are evaluated. If the polar direction is too different from the camera's polar position, in other words, when a chunk is on the other side of the planet compared to the camera, this chunk is hidden, but kept enabled. When a chunk is behind the camera in its entirety the same is done.

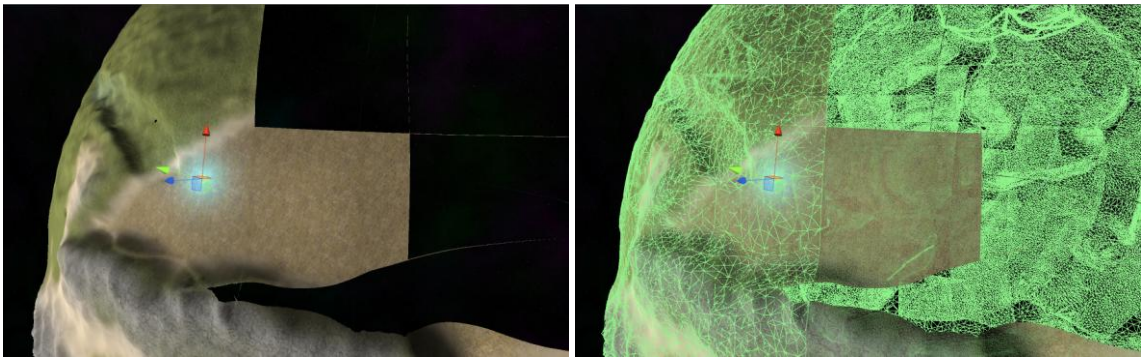


Figure 105: Visible and hidden chunks and collision (green)

To make the terrain more appealing terrain props were added. To determine the placement of these props a file is exported from Houdini (See chapter: Planet Creation/Object placement) . This file is then loaded into Unity which reads out all the data.

The difficult part for this was to make "teach" Unity to read this data in a sensible way. For this a file parser was written. In the past I created a parser for Houdini 11, together with R. Marx. Since Houdini 12 however, the .geo format has been reworked to include the handling of edges and to be more in line with the .FBX format. This also meant the file parser had to be redone. I took this chance to make the parser more versatile and stable as well.

```
"tuples",[[0.931895852089,0.354416638613,0.0771941393614],[-0.235697686672,-0.70396053791,-0.669989705086],
```

Figure 106: A line from the H12 .geo format

The file parser reads in all the lines as strings, at this point the parser regards everything as text, not as numbers. The parser splits these sentences, converts everything to usable numbers and stores them away in lists. There are Lists for each variable that is stored in the file: N, P, EXTRADATA, up, name and scale. Each entry has the same place in each list, this makes it possible to assign all variables to the correct object. For example: N[134], P[134] and up[134] all belong to the same, 135th, object. Note that the first entry is 0, not 1.

Once all the variables have been stored away in lists Unity can use the data to actually place the objects. On smaller scales it would be possible to create all the objects and be done. On a planetary scale however the total amount objects would be either to vast or the objects would appear very sparse.

Instead the system used in the demo creates, or relocate objects and removes them on the fly, but only near and in front of the camera. In the "EXTRADATA" variable a render distance is defined. If an object is in front of the camera and within render distance the object is created or an unused object of this type is moved into position. The latter is done to avoid using the UnityEngine.GameObject.Instantiate() function where possible, as this is a slow function to use frequently. To save the CPU more work, the objects are also not created when they are below the horizon. This means the lower the camera, the less objects are visible as they would be occluded by the horizon anyway. To avoid objects "popping" into existence, objects spawned by this system all have specific shaders to scale them gradually based on camera distance.

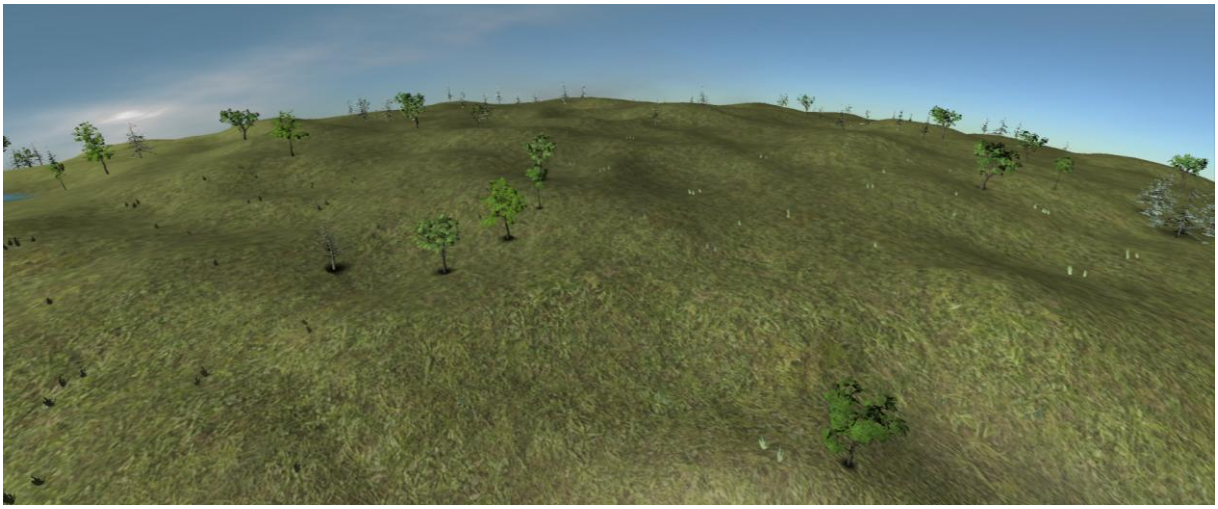


Figure 107: Object placement in Unity

With actual water meshes (See chapter: Planet generation/Water) and terrain underneath it, Unity is capable of showing underwater areas. Changing from above to below the water surface requires a couple of adjustments, different fog settings and some additional effects, disregarding game play mechanics at this point.

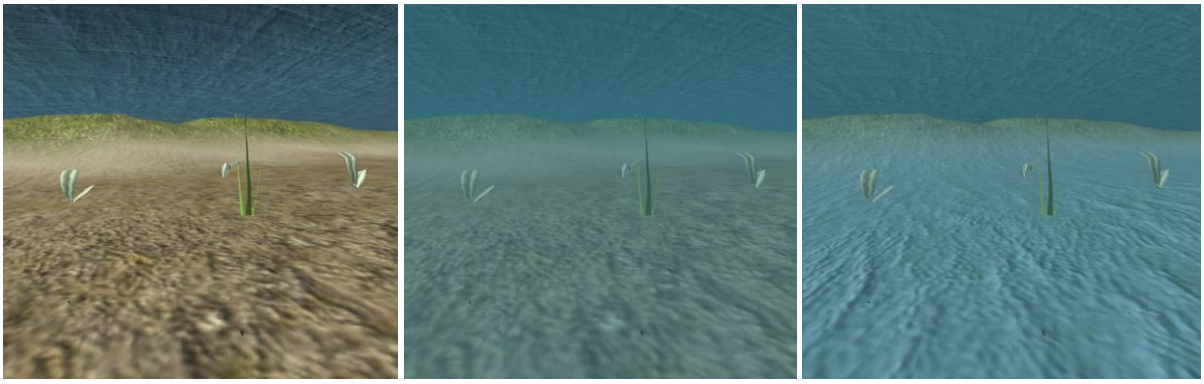


Figure 108: Layering of underwater changes.

The first picture of figure 108 shows how it would have looked without changes. In the second picture a plane is added in front of the camera with an animated water shader. In the last picture the ocean floor is colored blue to reduce contrast. Finally fog is added underwater.

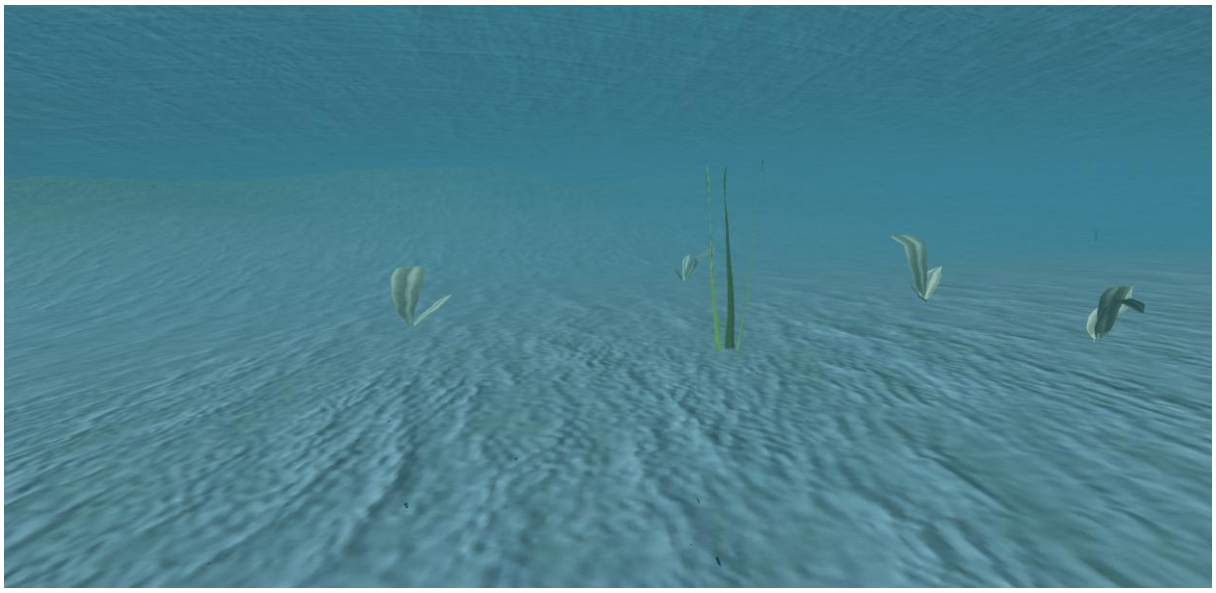


Figure 109: Final under water with added fog effect

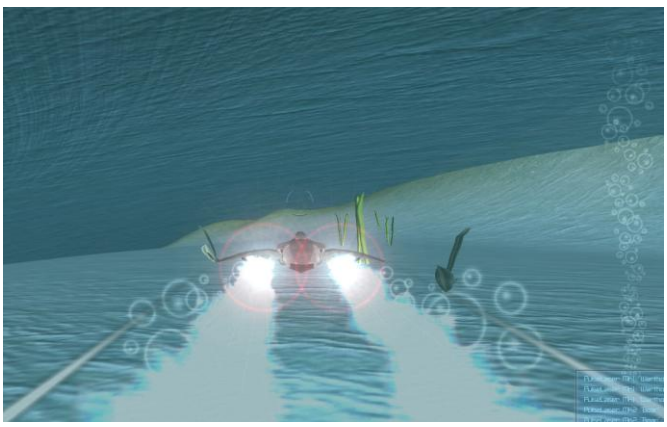


Figure 110: Underwater particles

Apart from this extra particle effects have been added for the underwater areas. Ambient bubbles are spawned just like seaweeds. There are also added bubbles for engine effects and weapon trails in the demo.

Apart from the underwater effects, there are also water surface effects. These surface effects consist out of water splashes and water wakes. The latter needed a custom particle manager, because the needed orientation of the wakes was not a supported option in the built-in shuriken particle system of Unity.

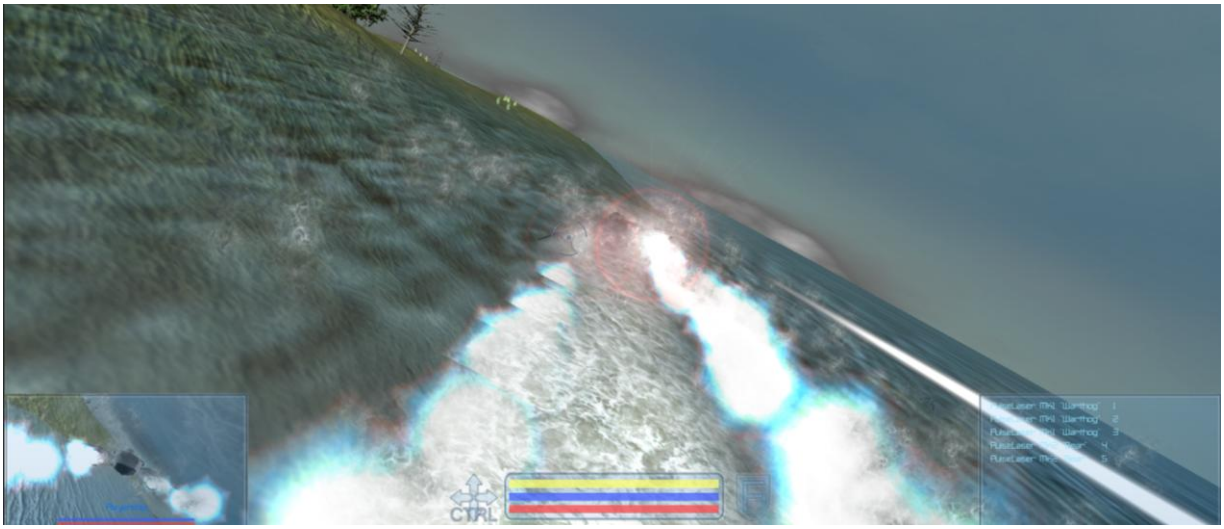


Figure 111: Surface water effects

Determining whether the camera and player are either above, under or at water surface level required building separate system as well. Basically when the player touches a specific sphere collider it is at ocean level. However the system also distinguishes whether the player is not above land at the same elevation. To do this the system looks at the polar position and looks up the corresponding climate via a texture mask. If the climate confirms that the player/camera are actually above water the particles are shown and the underwater visuals can be toggled.

There is also an particle effect added for re-entry. The strength of the effect is determined by the velocity and density of the atmosphere. In figure 112 the effect is obscured mostly by the lens flares from the engines, on the secondary camera however, the effects is more visible. Apart from the particle effect, the actual craft's material is colored according to the strength of the particle effect. Note that the curve at which this effect occurs is set arbitrarily in the form of a sine along the distance to the planet.

Figure 112: Re-entry effects

figure 113: Planet from afar, at 72.6 Frames per second

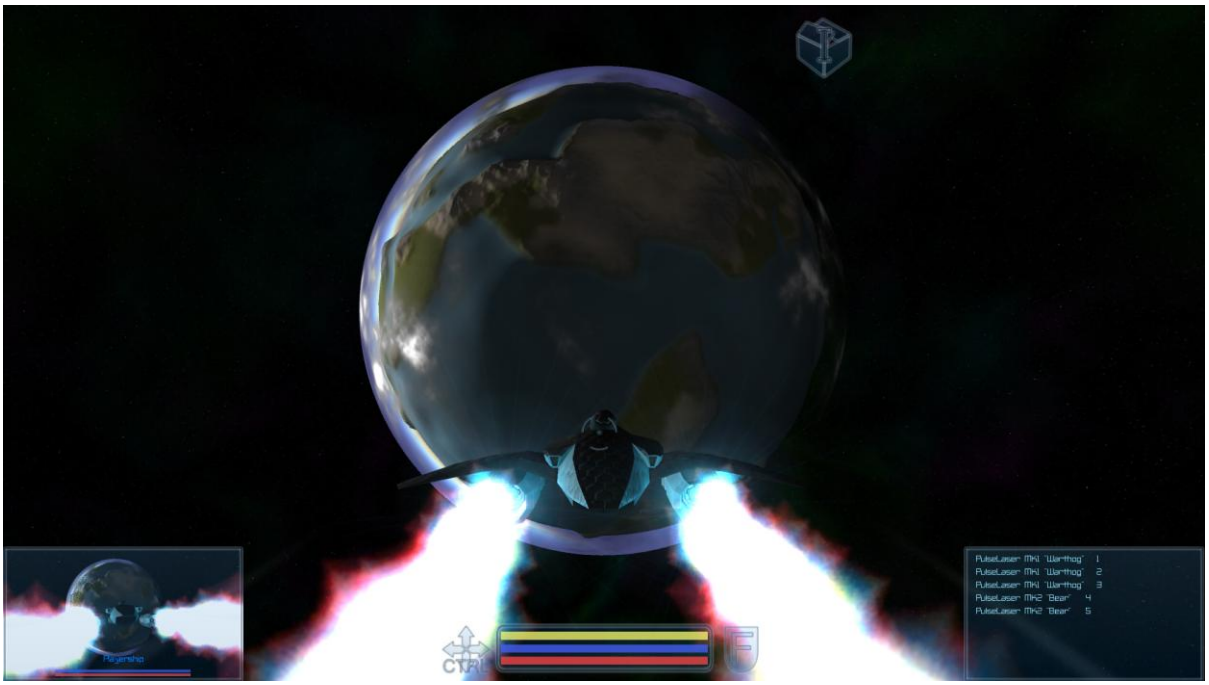


Figure 114: Planet from medium range.

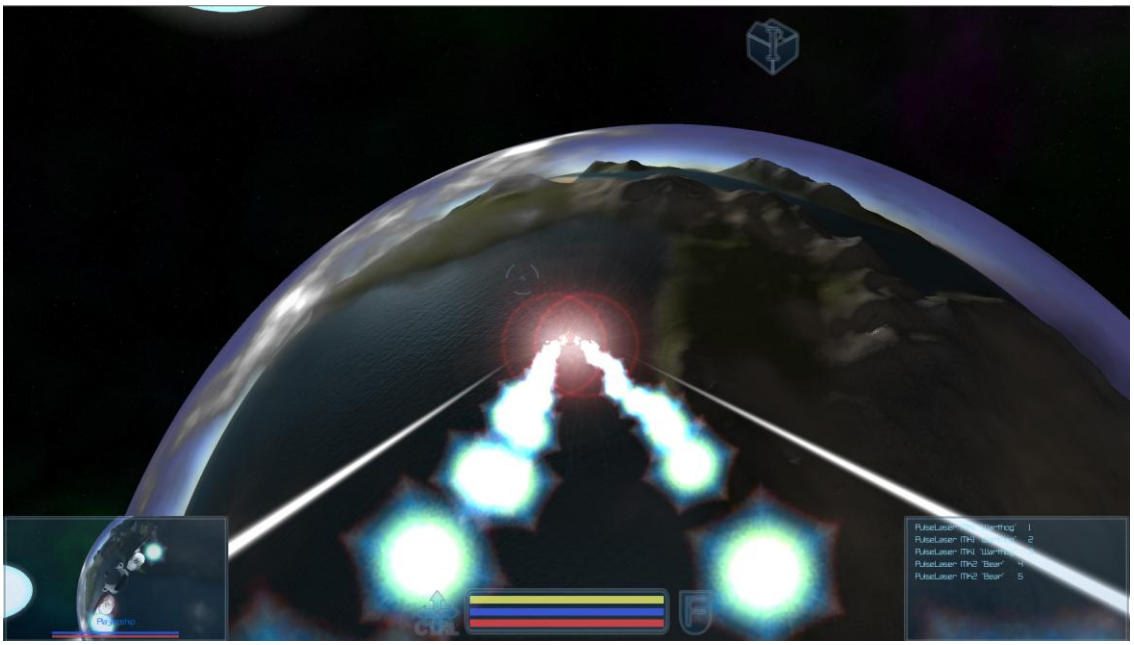


Figure 115: Planet from low orbit

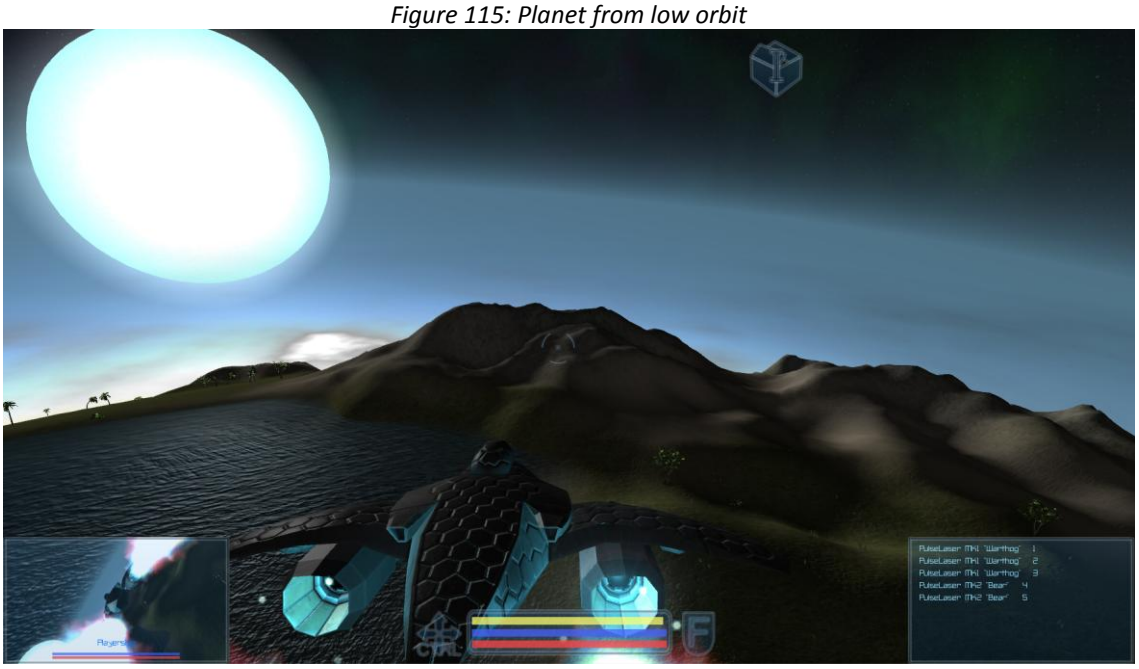


figure 116: Entering the atmosphere

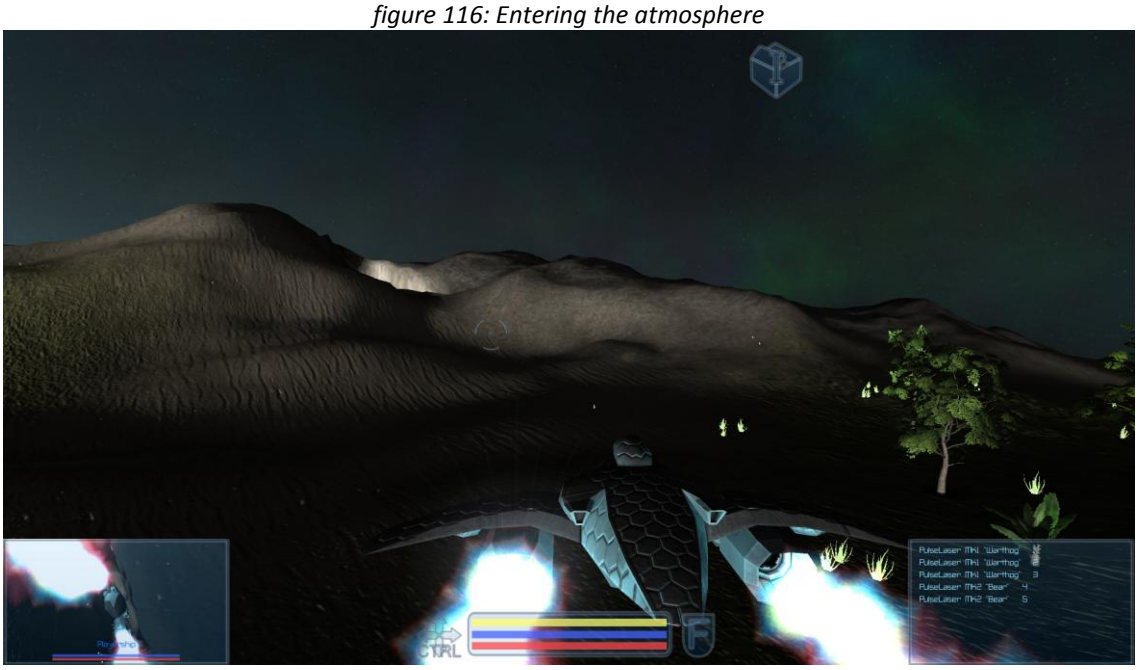


figure 117: Flying towards a cave entrance.

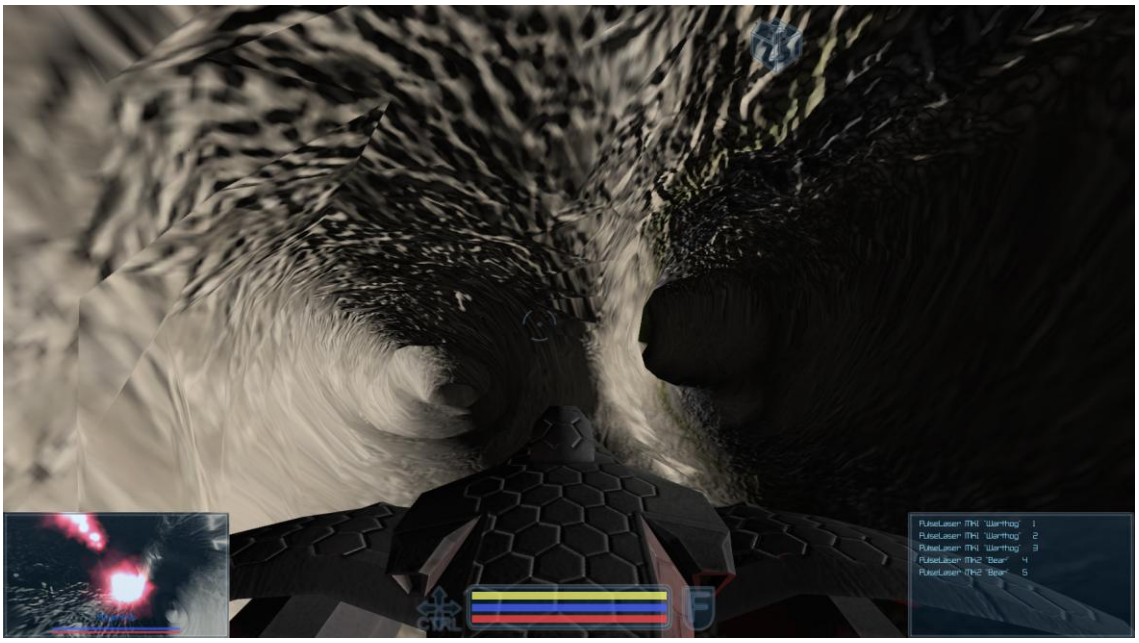


Figure 118: Inside a cavern

figure 119: Flying near the coast, at 63.9 frames per second.

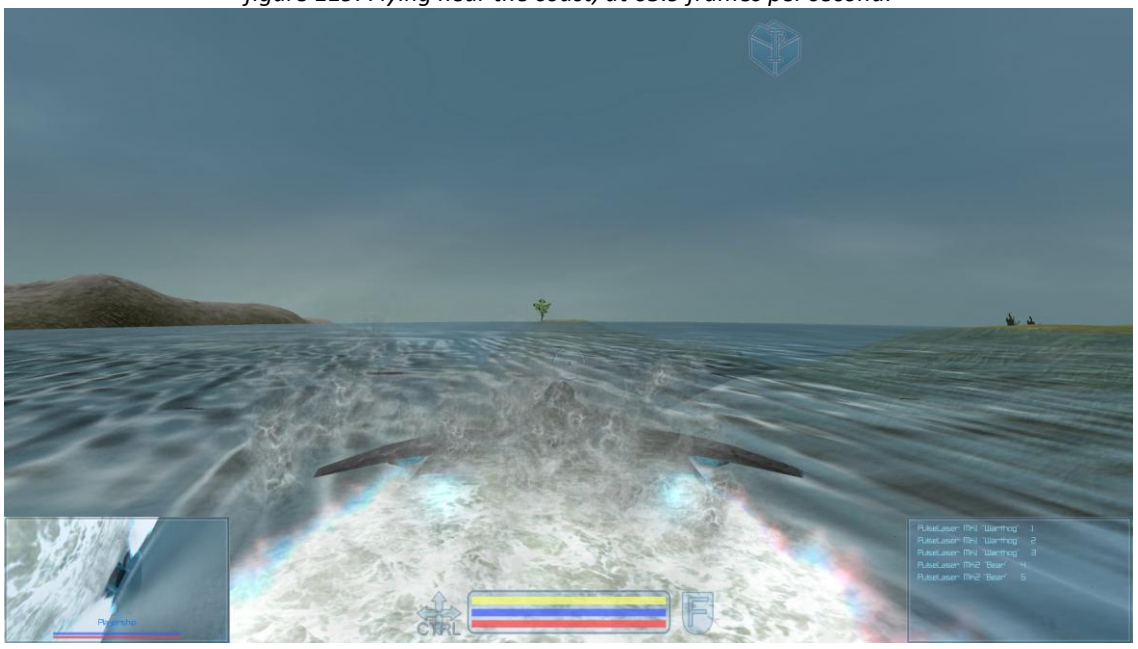


Figure 120: Skimming across the water surface.

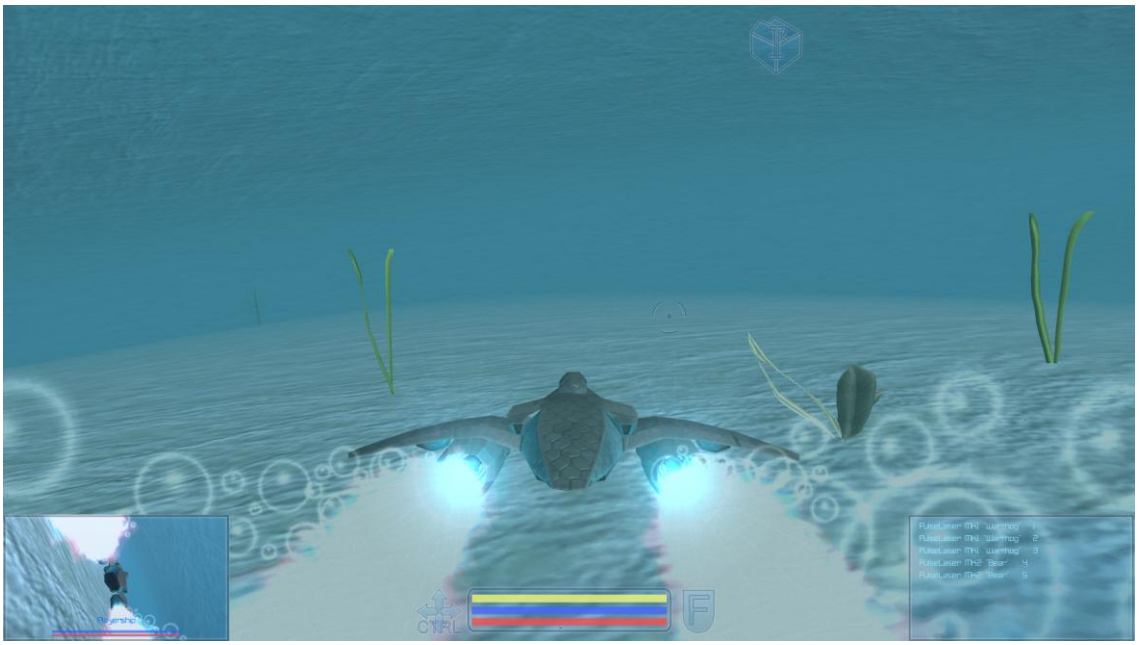


Figure 121: Underwater

Figure 122: Flying along a river

Figure 123: Flying over land, at 52.7 frames per second

Figure 124: Flying over the subtropics

Figure 125: Flying over the border of a desert

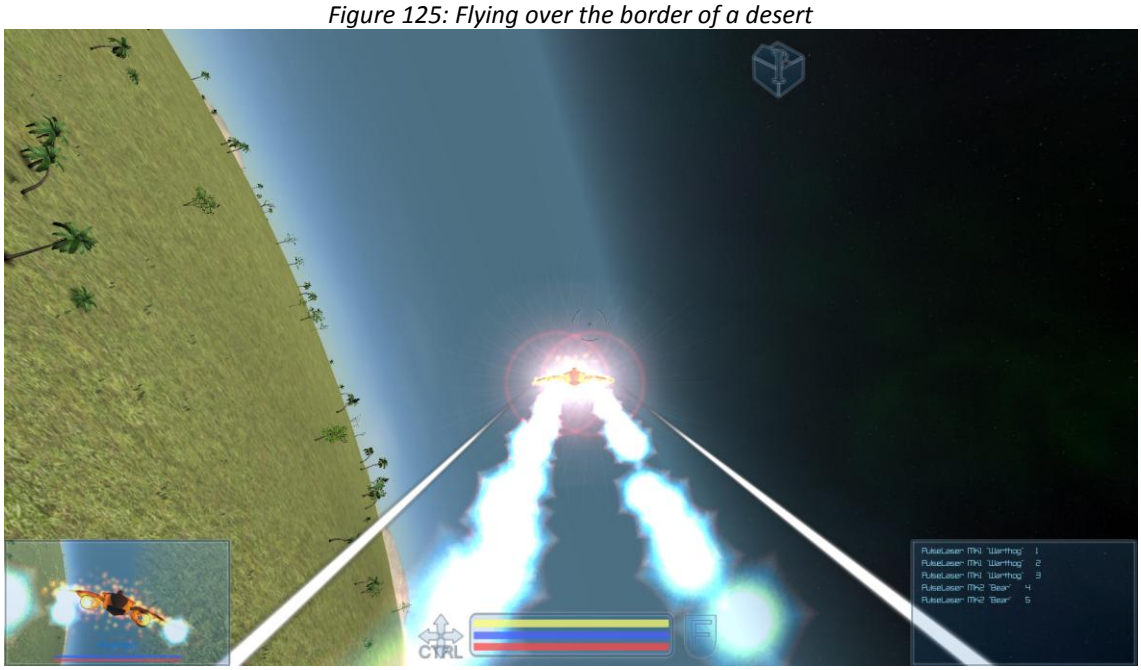


Figure 126: Grazing through the atmosphere

Figure 127: Flying at the night side of the planet

Figure 128: Flying towards dawn

SUMMARY

In this document the workflow of creating a planet procedure has been laid out, as well as the thought progress behind some of the decisions. It also incorporates some background on the various subtopics.

The procedure aims to greatly increase the efficiency at which planets can be created. The creation of one planet takes about two hours to set up along with an export time of about a day. In contrast, manually creating a planet at this level of detail would take one person at least two months. This project took around 28 weeks to complete, combining both my specialization and graduation phase. A simple calculation tells that the need for just 4 planets would justify this time cost already. On top of saving this artist the repetitiveness of doing this work for 32 weeks straight.

The project started by picking a basic layout. From there it grew organically, Features were added, updated and combined throughout the entire project. In a semi-variable order, with the associated subchapter of Planet creation:

- Support for height map projections. (Height map projections)
- Tectonic and Climate emulations (Climate Generation)
- A chunking system, using voxels. (Chunking, radial/polar; Chunking Cartesian & Voxels)
- Detail noise (Noise; Terrain Detailing, Pure Noise & Terrain Detailing, Texture based)
- Cave systems (Caves)
- Terrain props (Object placement)
- Texture generation (Texture maps)
- Terrain Blending data (Vertex blend data generation)
- Exporting everything (Export Pipeline)

While adding these features, optimizations have been done and accuracy was increased for the chunk borders and UVs. From halfway during the project the focus started to shift towards graphical quality.

Apart from creating the procedure, also an implementation has been done for the procedure in Unity3D, a game engine. This implementation has been used to test the exports of the procedure and determine how to improve the procedure further at first as well as showcasing the procedure. There is explained how the various shaders work that are used for the demo, as well as how the all the data that is generated for the planet is read back into the demo.

TUTORIALS AND ARTICLES

Side Effects Software (2012)
Houdini. *Company: About side effects*
Source: http://www.sidefx.com/index.php?option=com_content&task=view&id=22&Itemid=51

Wittens S. (2009)
Blog. *Acko: Making Worlds*
Source: http://acko.net/blog/making-worlds-introduction/

Willmott, A., Quigley, O., Grieve, J., Stratton, C., Compton, K., Todd, E., Goldman, E. (2007).
Maxis Sketches. *SIGGRAPH 2007: Creating Spherical Worlds.*
Source: http://www.cs.cmu.edu/~ajw/s2007/0251-SphericalWorlds.pdf

Wikipedia (2013a).
Wikipedia. *The free encyclopedia: Plate tectonics*
Source: http://en.wikipedia.org/wiki/Plate_tectonics

Wikipedia (2013b).
Wikipedia. *The free encyclopedia: Atmospheric circulation*
Source: http://en.wikipedia.org/wiki/Atmospheric_circulation

Wikipedia (2013c).
Wikipedia. *The free encyclopedia: Thermohaline circulation*
Source: http://en.wikipedia.org/wiki/Thermohaline_circulation

Wikipedia (2013d).
Wikipedia. *The free encyclopedia: Rain*
Source: http://en.wikipedia.org/wiki/Rain

Hoekstra, F. (2011).
Graduation Project. *IGAD: Procedural environmental design.*
Source:
http://freek3d.com/downloads/Freek_Hoekstra_Graduation_final_report_Final_PRINTED.pdf

Lueders, S., Duchesneau, F. (2008).
Mailing list. *Sidefx-houdini-list: Access nurbs parametric uvs.*
Source:
http://www.sidefx.com/index.php?option=com_mailarchive&Itemid=212&view=WEB&msgid=1933700598.189211200495142968.JavaMail.root@dahlback.prod.local&perpage=20&revdate=off

Elias, H. (2003).
Models. *http://freespace.virgin.net: Perlin Noise.*
Source: http://freespace.virgin.net/hugo.elias/models/m_perlin.htm

Wikipedia (2013e).
Wikipedia. *The free encyclopedia: L-system*
Source: http://en.wikipedia.org/wiki/L-system

Crytek (2012).
Cry Engine 3. *Doc 4. Technical Documentation: Tangent Space Normal Mapping*.
Source: http://freesdk.crydev.net/display/SDKDOC4/Tangent+Space+Normal+Mapping

García, J. (2012).
Computer graphics. Txutxi.com: Tangent space matrix (TBN).
Source: http://www.txutxi.com/?p=316

Wikipedia (2013f).
Wikipedia. *The free encyclopedia: Rayleigh scattering*
Source: https://en.wikipedia.org/wiki/Rayleigh_scattering

O'Neil (2005).
Nvidia. *GPU Gems 2: Accurate Atmospheric Scattering*
Source: http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter16.html

PRD et al. (2013).
Unity Forum. *Unity Community Support, ShaderLab: Atmospheric Scattering help*
Source: http://forum.unity3d.com/threads/12296-Atmospheric-Scattering-help

SIP Team. (2012).
Suicidal immortal phoenix. *Global Game Jam.*
Homepage: http://archive.globalgamejam.org/2012/suicidal-immortal-phoenix

David Braben & Ian Bell. (1984).
Elite. *Acornsoft.*
Homepage: http://elite.frontier.co.uk/

Maxis. (2008).
Spore. *Electronic Arts.*
Homepage: http://www.spore.com/

Sony Online Entertainment. (2012).
Planetside 2. *Sony Online Entertainment*.
Homepage: http://www.planetside2.com/

Squad. (2012).
Kerbal Space Program. *Squad.*
Homepage: http://kerbalspaceprogram.com/

Mojang. (2011).
Minecraft. *Mojang.*
Homepage: https://minecraft.net/

Keen Software House. (2012).
Miner Wars 2081. *Keen Software House.*
Homepage: http://www.minerwars.com

IMAGES

Page 4, Figure 2: http://www.twandegraaf.nl/Games.html

Page 5, Figure 3: http://wastetimepost.com/10-Fascinating-Impact-Craters-on-Earth.html

Page 5, Figure 4: http://spore.4fansites.de/galerie_5_1003.html

Page 5, Figure 5: http://www.thirteen1.com/2012/10/19/soe-announces-planetside-2-release-date/

Page 7, Figure 10: http://www.cs.cmu.edu/~ajw/s2007/0251-SphericalWorlds-slides.pdf

Page 8, Figure 12B: http://johnflower.org/tutorial/finding-height-maps-web

Page 13, Figure 26: http://acko.net/blog/making-worlds-1-of-spheres-and-cubes/

Page 16, Figure 32: http://www.divergeuk.co.uk/?p=490

Page 16, Figure 33: http://www.minerwars.com/Pics.aspx

Page 18, Figure 37&38: http://freespace.virgin.net/hugo.elias/models/m_perlin.htm

Page 36, Figure 72: http://freesdk.crydev.net/display/SDKDOC4/Tangent+Space+Normal+Mapping

SPECIAL THANKS TO

Frecle ApS, Software developer,
For creating a free to use flora generator.

Martijn Gerkes, IGAD student,
For giving me hints for using, and introducing me to, the Unity3D engine.

Ben Golus, Uber Entertainment,
for making a post on a forum that got me started on creating planets in Houdini.

Kim Goossens, IGAD teacher,
For introducing me to Houdini and making me realize procedural is the path I want to take into the game industry, and of course for supervising me during this project.

Freek Hoekstra, Ex-IGAD student,
For inspiring me to do world generation with Houdini, and for creating a high benchmark to reach as a project. As well as creating some of my literature material.

Killst4r, 3d-modeler,
For creating free to use tree packs, used in the demo.

Robin Marx, LuGus Studios,
For giving me the chance to increase my programming skills during my internship. And for helping with an earlier version of my Houdini to Unity3D file parser.

Laurens Mathot, Creative Programmer.
For helping me better understand real-time tessellation, even though it did not end up in the project in the end.

Charles Trippe, Effects artist,
For creating a UV-relax tool, that was used in the procedure.

The Unity Community
For helping me out with shader language as well as atmospheric scattering effects.

Unity Technologies,
For creating an open and user friendly game engine to showcase my project.

Various other people,
For giving suggestions, reviewing my work and being generally supportive.

To put rendering times and demo performance into context here is an overview of the machine used for this project generated by the DirectX Diagnostic Tool. The machine is now one and a half years old.

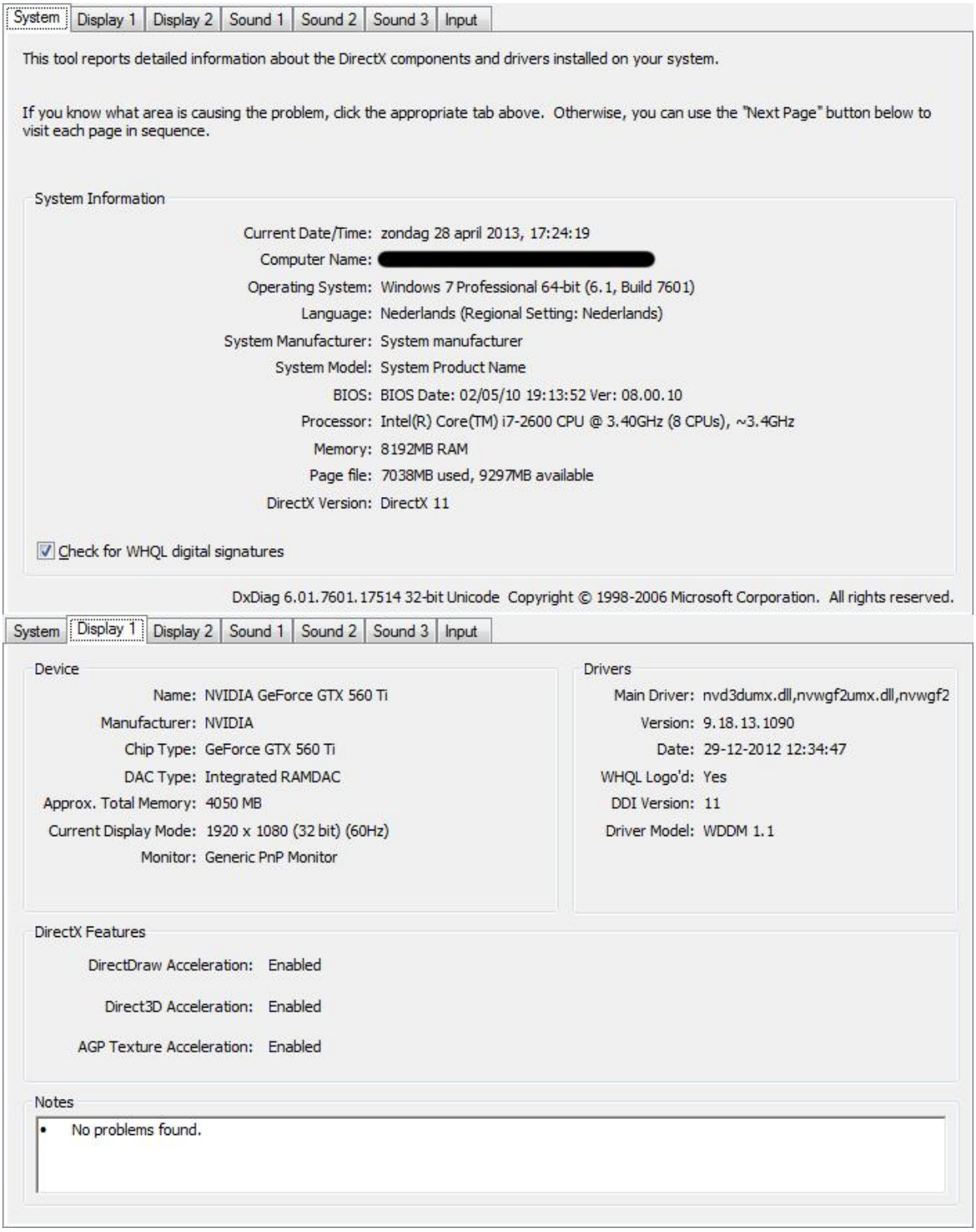


Figure 129: System information by "DxDiag"