Bachelor Semester Project Report at École Polytechnique Fédérale de Lausanne

LGG (Computer Graphics and Geometry Laboratory)

Real-Time Procedural Planet with OpenGL

Abstract

The goal of this project is to to demonstrate how to generate procedural planets in real-time on recent hardware with a reasonable compatibility among the current hardware using OpenGL 3.

We present the necessary knowledge and techniques to achieve our results. This project implements Perlin noise, Fractional Brownian Motion (fBm) and a variation of ROAM; and it uses GLSL shaders. We will show how we combine level of detail with procedural methods to achieve real-time rendering with a limited amount of memory. We present the method to use noise to build planets.

1 Introduction

The problem of generating procedural planets is challenging because they have enormous sizes and complex features. The set of data required to describe such objects accurately are quite high, thus we can use procedural algorithms to generate them. Nowadays procedural methods allows us to render entire planets at the level of details needed to be credible. In this project we develop a solution for this problem.

We begin in Section 3 with a small explanation of the base code we chose for developing our project. We briefly compare some widespread frameworks and argument why we decided to only use small libraries.

Then we speak about the algorithms that calculate the features of our planets (like the height and the choice of textures). They rely on noises and interpolations with some optimizations for real-time generation such as Real-Time Optimally Adapting Meshes (ROAM). Noises will be discussed in Section 5 in which we talk about building noise signals from pseudo-random functions. From there we build more interesting patterns with Perlin noise used as a basis function for fBm. We then explain how we use these methods for the contents.

In Section 6 we talk about Real-time optimizations. LOD, ROAM and the variation of ROAM, we call SRAM, will be explained deeply. We will see about their uses and how they work in our project. The real-time constraint of our project lead us to work on optimizations, they are detailed in this section.

Finally we will conclude with our results along some screenshots in Section 7 and then we give our final words in Section 8.

2 Related Work

For the procedural content generation we used the famous work of Ken Perlin on noises[Per99]: the Perlin noise. We used the implementation described in Musgrave's paper [Mus00]. It is based on a gradient-type noise which uses a noise basis (like Perlin Noise) and fractional Brownian motion (fBm). The paper develops how to generate and render realistic and real-time flat terrain. For spherical landscapes a recent paper [DGGK11] discussed how to generate rivers on a procedurally generated planet. They incorporated their computations inside the terrain level of detail procedures. We tried to use their work but some important details were missing especially in the way they compute heights for their different cases.

The level of detail (LOD) algorithm required for displaying the planet mesh was inspired by Real-time Optimally Adapting Meshes (ROAM). It is based on the original paper [DWS$^+$97]. This work was initially designed for flat terrain, it was adapted to the sphere shape by Sean O'Neil [O'N01]. Although his work was written as a few pages in blog articles, it had formalism and enough details to be implemented. That's the reason we decided to continue some of his work. His articles describe how he adapted the initial ROAM algorithm by bootstrapping the mesh from a cube and how the new algorithm starts from this point. At the time of the article, shaders were not available yet, thus he could not configure the graphical pipeline and make computations on the GPU, thus he generated vertices and textures on the CPU side using fBm with Perlin noise. Several years later when shaders became available he focused his attention on atmospheric scattering.

Another related work not published as a paper but as few small blog articles explains the enormous effort to create a planetary engine [Bre13]. It can generate Earth-like and other variety of planets. The published posts describe for example: how to manage textures, how to generate them, give some insights on how to texture a planet and discuss some performance problems. Unfortunately the sources are not available (because the project is commercial) and the articles enter superficially over the details but we found there some valuable materials. The results are quite impressive and may even be state-of-the-art of planet generation and rendering in real-time. The techniques they use for example are: fBm, multi fractal noise and Voronoi diagrams on the GPU, some noise like: the Diamond-Square algorithm for the heightmap and a tons of shader effects like: depth of field, atmosphere scattering, shadow mapping, parallax mapping and a lot of their homemade heuristics. Their work has begun more than five years ago.

3 Project Setup

The C++ language was chosen because it offers good generic data structures (list, hashmap, tree), is quite efficient (nearly as fast as C), offers a good abstraction level

(Object oriented paradigm) and finally it is widely used standard, which enables one to choose among a wide selection of libraries and frameworks.

We then decided whether to use a framework, some smaller libraries or to write all from scratch.

3.1 Framework

There are a few generic engines that offer a great level of abstraction. We looked at a few of them and decided to compare them to choose a candidate to use in the project:

Ogre: A framework oriented towards scene and real-time rendering. It is open-source, has an active community and is multi-platform. It offers great facilities for rendering and managing scenes with meshes and there are procedural libraries integrated. The documentation is huge and we considered that the learning curve to correctly use this framework during the semester was too important.

Unity3D: A cross-platform framework made for video games. We consider this engine not a valid solution for us due to its licence (proprietary) which would have forced us to blindly accept the results and we would have not been able to fix any error as the sources are not distributed.

Irrlicht: A cross-platform framework. It is a small and yet young framework but does not offer official integrated procedural libraries and we prefer not to rely on unofficial content for this project which may not be properly tested.

Microsoft XNA: A set of tools oriented towards video game development. We cannot consider this as a valid choice for us because there is no official support for OpenGL.

After considering these frameworks we were not convinced to use one for this project. We searched libraries for vectors, matrices and other elementary operations used in computer graphics. As a result we were fully in control of the graphic pipeline ourself and the curving lines was greatly reduced due to the generality of these tools. We found out that the *OpenGL Samples Pack* by *g-truc*[1] contains a lots of interesting samples to learn from. We used another library called “LGL” for Light Graphic Library“, it contains various math operations and useful matrices.

3.2 OpenGL

We decided to use OpenGL for several reasons and will briefly explain why we chose OpenGL 3.0 for this project. The first point was its availability on nearly all platforms in the market today: Windows, Mac, iOS, Linux, this underlines the current API portability and popularity. The second point was the experience in OpenGL we accumulated during the studies at EPFL, we did not spend time to learn another API, which in this case is Microsoft Direct3D, it was made only for Windows. Moreover OpenGL offered the same if not more recent features than its counterpart Direct3D in the last versions (OpenGL 4.3 at the time of writing).

[1]Website: `http://www.g-truc.net/project-0026.html`.

We could have chosen the very last version of OpenGL for our work, but it wasn’t practical for various reasons: the lack of supported hardware it and weak documentation of the new features. Indeed the new cards supporting that version may be quite expensive instead we wanted to use our actual hardware (dating before 2011). Moreover we do not needed the very last features that was offered by the last OpenGL, except for the tessellation (that we did not used). The advantage of choosing OpenGL 3.x is good availability of documentations, such as books and articles on the subject for nearly everything. And finally, more low-end level cards like the integrated Intel cards which support OpenGL 3 on recent laptops. This allows us to run on a variety of platforms (at least Unix flavors, Windows and even mobile systems like Android and iOS with some adaptation).

4 Initial Work

Our last experience of OpenGL was with the fixed pipeline, however we needed shaders and VBO for this project. These ones are available since OpenGL 2, which involved some learning. The details that need to be done at some point were considered during this initial work, for example: perspective projection matrix, shaders instantiation and binding, vertex buffer object (VBO). We decided to draw a basic square-shaped terrain to test the first procedural algorithm: Perlin Noise with 2D-cosine interpolation. We used static texture mapping and simple lighting based on the height of pixels. The resulting Figure 1 is simple but it demonstrates the working of the basic functionalities.

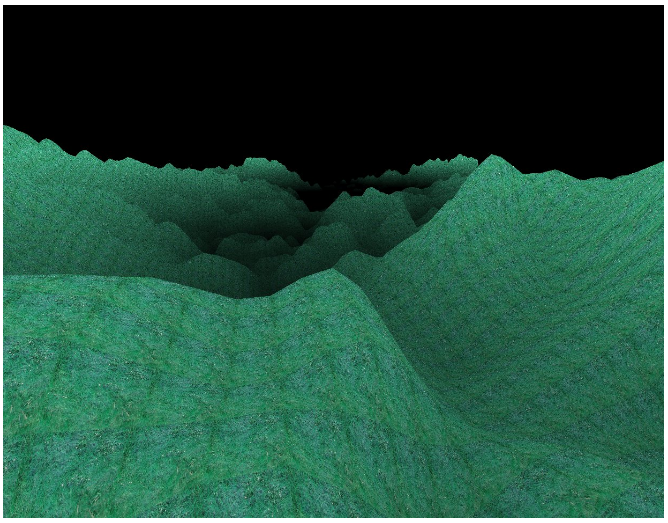

Figure 1: Basic 2D terrain test with one texture.

5 Procedural Generation

Procedural generation enables us to create a wide variety of interesting objects based on some deterministic cleverly chosen algorithms (functions) and some input values (seeds). The data given to the program is nearly null thanks to the procedural functions. Moreover procedural objects can be detailed as desired by increasing the sampling resolution of these algorithms. This allows near object to be drawn in a realistic manner in real-time while far objects are drawn with lower details. This process is made on the fly based on estimations of the required level of details (LOD). The LOD is calculated depending on the camera view and permits us to adjust polygons on the fly.

In the following sections we briefly introduce in what kind of projects noise were used and then we will explain in more details how we compute noise in practice.

5.1 Perlin noise

In a computer graphics talk made in 1999, Ken Perlin presented an algorithm [Per99] that produces a new type of noise: the Perlin noise. The major difference with the noises already available that time was its particular focus on rendering fire, smoke and clouds. When the Perlin noise is combined itself with different scales – called a fractal noise – one can obtain very fine-grained textures with adjustable level of detail.

Perlin noise coupled with interpolation has been successfully used in projects like video games (eg: to create walls, grounds, clouds with more random details), graphics demos (eg: procedural planets, smoke). As our project is a real-scale planet it is the right algorithm to choose because it can be tweaked easily and is efficient for real-time generation. Moreover the general idea behind the fractal Perlin noise is quite simple and easily done either on the CPU or the GPU.

To compute a point $\vec{x} \in \mathbb{R}^n$ of the Perlin noise we first sample the 2^n integral neighbors of $\vec{x}$ of a pseudo-random integral function, $f : \mathbb{N}^n \to \mathbb{N}$, then we interpolate the values. For the 3D case, we need to evaluate f at $2^3 = 8$ points then we can use a trilinear cosine interpolation to "combine" the 8 values to get the value of f at $\vec{x}$ as if f was a smooth function.

5.2 Fractional Brownian Motion

The purpose of the fBm noise or the Fractional Brownian Motion is to sample multiple times a basis function like the Perlin noise in order to increase the details of the output at different scales.

The actual computation of the Perlin noise with fBm can be summarized by the formula [Mus00]:

$$f(\vec{x}) = \sum_{i=0}^{N} H^{-i} b(f^i \vec{x})$$

where the noise settings are as follows:

N is the number of octaves (or dimensions). This is the number of times the basis function is sum-weighted[2].

H is the persistence (or the lacunarity). This is the amount by which we divide the amplitude for every octaves, thus giving less weight to greater octaves.

b is a basis function. This is the interpolated noise function we sample.

f is the frequency. This is the factor by which the sampling point is compressed, thus greater octaves are more "noisy" than smaller octaves which are more "smooth".

Thus to have the desired results we need to tweak the noise settings that give the best results. Note that the output of f should be normalized, this is more convenient for later uses. We give an example for 1D noise in Figure 2.

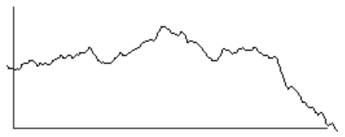

Figure 2: A weighted sum of a basis function.

Note that the number of dimensions of $\vec{x}$ can be arbitrary large as noted by Ken Perlin, it is completely generalizable to any dimension. In practice we often use it in 2 or 3 dimensions. In this project we began by the 2D version for the flat terrain, then we moved to the more complex 3D version for the planet generation.

5.3 Content

In this subsection we explain how the noise was actually used to give interesting features to the planet. The reasons we use noise for content generation is twofolds. First the amount of data needed by the program is very low, it only needs the algorithms and some small input like a seed. Secondly we can sample these algorithms at an arbitrary precision on the fly, that means we can generate every features of the planet in real-time. Although some computations can be very expensive leading to poor performances, optimizations can partially fix this problem. We will come back on optimizations and for now we discuss the theory behind the heightmap generation.

5.3.1 Planet Mesh

We begin by approximating a planet surface by the surface of a perfect sphere. That is a simple 3D object with a center $\vec{c}$ and a constant radius, R. We assume that the planet radius has the same value regardless of the direction (the position on the surface) which is not true for Earth for example (poles and equatorials have different heights) because Earth is an oblate spheroid.

The next step is to add irregularities to the surface such as mountains, hills, valleys, plains: difference of height, thus in slopes of the terrain but we need to keep some coherence in the shapes. To achieve this goal we use the aforementioned fBm noise which has both irregularities and coherence in the output as a perturbation of the radius. These perturbations should be small compared to the radius. To achieve this we weight these irregularities by a constant factor that we call the (maximum) height, $H << R$, of the planet. The height perturbation is described by: $\delta(\vec{x}) = Hf(\vec{x})$, where f is the fBm noise in our case. Therefore the total height is: $h(\vec{x}) = R + \delta(\vec{x})$.

In our project, this vector $\vec{x}$ is of dimensions 3 and represents a direction. This means the noise is always

[2]We decided to present the integer-version but [Mus00] has a more general version where $N \in \mathbb{R}$.

sampled on a unit-sphere. This is convenient because it won't depend on the chosen settings: like the planet radius or the maximum height. Moreover it must be said that a sample is only calculated when it is needed for a vertex, there is no precomputation and it is done on the CPU and the mesh is stored in RAM (and duplicated in the GPU memory).

5.3.2 Texturing

Once the triangles of the planet (made of vertices) is computed, important features come from the textures applied on the surface. There are multiple effects that are combined in our project: we sample two predefined textures, blend them and add a diffuse lighting shading. The set of textures we took is a base of freely-available photos of different materials[3]: fine gravel, flint rock, grass, mountains, rock, sand and ice. We transformed them so they can be tilable in all directions. We did not generate these procedurally because we would have spend an important time to generate such a variety of materials and moreover we think we would have run too short on performance. The major benefits would have been smooth transitions on the entire surface and a bigger diversity at the expense of a great loss of performance. To determine the textures applied at a given pixel we lookup in a 2D table on the GPU based on the height and the longitudinal coordinate of the point. We have defined 2-dimensional tables that tries to mimic the different zones of Earth (ice poles, deserts, tropical and forests) by mapping a certain range of height and polar coordinates to a particular textures. This process is completely done on the GPU.

We use cube mapping to compute which pixel of the texture to sample (imagine a cube surrounding the planet then we project the $\vec{x}$ into the first face it crosses, it gives us the 2D texture coordinate). Then with the two pixels we blend them additively: we sample α the first and $1-\alpha$ the second texture, where $\alpha = P(\omega\vec{x})$ and where P is a Perlin noise function and ω a constant factor that change the scale of the blending "patches".

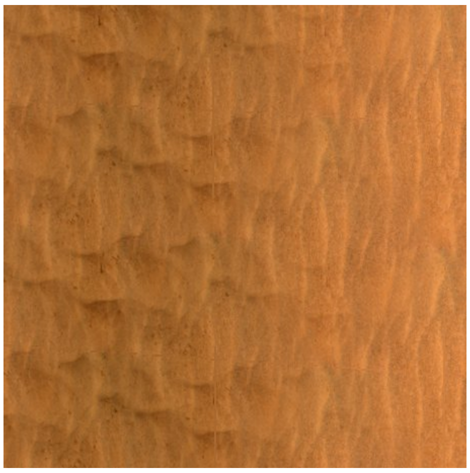
Figure 3: Example of our texture blending with two versions of the sand textures.

Although heights are guaranteed to be continuous along the planet surface, the polar coordinates are not. Thus this gives multiple noticeable discontinuities around the globe as shown in the leftmost figure 13. We introduced a factor to displace the polar coordinates on the GPU depending on the latitudinal coordinates using our favorite noise: Perlin noise. On the same figure we show how the displacement factor results in more natural boundaries.

Two more visual features were added to give a more interesting look to the planet: animated clouds and waters. Although they look quite different the underlying algorithm to generate them is the same: we use Fractional Brownian Motion. The main difference is in the way the color is derived from the noise value which is computed from the pixel position. Both are mainly computed in the fragment shader, they use a trivial fragment shader with a little addition for the water vertex shader: it varies the height with the time to give the impression of tides.

For clouds we use shades of grey directly based on the noise value between 0 and 1 but elevated to some power, in our case 8 was found by experimenting. This transformation reduces the number of low values, thus creating a few brighter clouds.

For waters the derivation process is more complicated. First it chooses between two cases: using a blue tone and adding the noise value as a way to shade this color; or it uses a predetermined color among a small list in order to saturate some parts giving the impressions of shadows and specular lighting. These choices are based on the noise value itself whether they fit in a particular range of values.

6 Real-Time

The term real-time is used to describe an interactivity constrain such that the application should react to user actions in a fluid manner. In our project that means we should target a reasonable frame rate (FPS) so that the user finds the experience smooth. One of our goal was to target 60 frames per second on a modern graphic card, we think the integrated graphic card are too limited to achieve this goal. We chose that number because it is the most common speed of vertical refresh of screens[4]. To effectively attain this goal we had to find some optimizations. In this section we describe parts of the architecture of the spherical ROAM algorithm and the optimizations we used in general. The interested readers can dig into the code for more details.

6.1 Level of Detail

The level of detail (or LOD) is a technique that decreases the geometry complexity (aka: triangles) of a mesh as the viewer moves far and increases it back as he moves nearer. The goal is to draw and to send only a limited amount of triangles to the GPU at once in order to increase the FPS. This is possible because the viewer has a limited visibility, thus he can barely see far triangles. By merging these triangles into bigger ones the viewer will probably not notice the the difference because the LOD keeps the number of triangles near the camera high enough. There are some

[3]source: `http://www.texturewarehouse.com/gallery/`.
[4]source: `https://en.wikipedia.org/wiki/Refresh_rate`.

notions that need to be introduced like an error metric on triangles to decide when to merge or split triangles and an algorithm that perform these two operations.

6.1.1 Real-Time Optimally Adapting Meshes

The ROAM algorithm introduces these two notions of error metrics and split/merge and we will briefly describe the way used in the reference paper [DWS+97] as our goal is not to replace this paper, we will only highlight the important points we used in this project. We have used ROAM because it is a popular choice for terrain rendering (one reason is its simplicity and another one is its efficiency), its design works well with procedural content generation and it is quite well documented.

The original ROAM algorithm works with triangles and diamonds, the two operations are: split on triangles and merge on diamonds as shown on Figure 4. A diamond is a particular configuration of triangles such that four triangles form a square where the shared middle point is opposite to the hypotenuse side of the triangles, thus when a diamond is merged (from four to two triangles) the continuity of the terrain is guaranteed. When we split a couple of triangles, a new point (midpoint) appears which offset is computed with the noise value at this position. These two operations were designed to be simple and efficient.

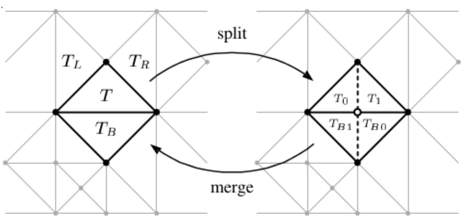

Figure 4: Split and merge operations (source: [DWS+97]).

The other important point of the paper is the error metrics used to decide when these two operations should take place. Although there are more complex metrics we chose a simple one for efficiency: our error is $e(\cdot) = \epsilon/\delta^c$ where ϵ is the height error, δ is the distance from the object to the camera and c is a constant found experimentally. The δ is calculated as follows: for a triangle we take the offset by which the split would elevate the middle longest edge, for a diamond we take the offset by which the merge would lower the middle shared point. Then given a triangle or a diamond we decide to split or merge respectively if the $e(\cdot)$ of this object is greater or lower respectively than a threshold C, which represents the visual error we accept. It was chosen experimentally by considering the number of triangles in the scene with respect to the camera distance.

The ROAM algorithm is optimal because it uses two queues: one for splitting and one for merging ordered by priority. Indeed every frames, the ROAM algorithm decided to perform the most important operations first, therefore if it has to abort earlier (to guarantee a constant FPS) it will output an optimal mesh for the time it was given. Our approach is not optimal in this sense because we don't do these operations in the order of priority, instead we iterate over the primitives and if we need to exit earlier we keep track of where the algorithm stopped and then in the next frame it will continue where it left. We think the overhead needed to keep up to date the priority of the triangles was too important, thus the paper proposes some optimizations but we decided to completely discard these ideas. Indeed we would have to recalculate the priority of all primitives every time. We used our own optimizations which are discussed below.

6.1.2 Spherical Real-time Adapting Meshes

There are some details that are specific to the spherical nature of our algorithm (SRAM) which are discussed now. As described by Sean O'Neil [O'N01] we start by approximating a planet (very badly at first) by a cube with 6 faces, each formed by 2 triangles, thus for a total of 12 triangles, see Figure 12.

The algorithm keeps track of the three neighbors for each triangle and the four triangles contained in each diamond. The split and merge operations stay the same as the original ROAM but we work exclusively on neighbor pointers. To split T: consider triangle T and its neighbor whose longest edge is common to form a new diamond by adding two more triangles; to merge: consider the four triangles in the diamond and replace them by two bigger triangles. The information contained in each triangle are: vertices, the neighbors information and its parents (triangle and diamond if any).

We introduced multiple optimizations to ensure a constant reasonable frame rate. The first one is to disable the SRAM update when we are far of the planet. We specified an initial level of details which looks reasonable from space. The second optimization is to stop the SRAM update when a certain amount of time has passed since the frame computation started, in our case 15 ms. The third is to discard primitives based on their spatial location with respect to the camera. More precisely we defined eight quadrants and assigned each triangle to a single quadrant. We consider triangles for update only if they are in the same quadrant as the camera, thus we skip easily more than one eighth of primitives. One complementary optimization is memoization of the noise function, that is: caching the values of a function into a table. We will discuss the impacts on performance in Section 7.

6.2 OpenGL Real-Time Rendering

We now discuss the low-level details required to draw our planet along some optimizations that were necessary due to the inefficiency of our first implementation in order to achieve our target FPS.

6.2.1 Buffers

To render any object with OpenGL 3 we need[5] to use a Vertex Buffer Object (VBO) which is a memory buffer stored on the GPU for storing anything like primitive positions, texture coordinates or even any custom value per vertex. We way to fill this GPU memory is by streaming the geometry through OpenGL which will manage the low-level parts for us. Our planet is no exception therefore we store every vertices of the triangles into a unique

[5]In fact we could have used Vertex Array (VA) but they offer lower performance.

VBO that we can draw with a single call. More precisely we use a position (3 floats), a normal (3 floats) and a noise value (1 float) per vertex.

The problem that arises with the use of a single VBO is the way we manage changes due to the SRAM algorithm on the triangles. In the first implementation we were resending the whole VBO, that is all the primitives, every frames. This naive way sets a performance bottleneck, thus we had to find a better way to incrementally send the geometry which leads us to a brief overview of the methods we tested.

The first improvement in performance was that we misused `glBufferData`. This function initializes a VBO memory each time it is called. We used this function in the drawing loop, thus filling the buffer and especially creating a new buffer each time. We moved this function into the initializing part (called once) and used `glSubData` instead to update the data. This had an important impact but it was not sufficient.

The second optimization was to incrementally update the data by streaming the changes as they are done in the SRAM algorithm. For that we first tried `glSubData`, that was inefficient but gave us the advantage of being able to choose more precisely when to stop our process. The second try was to use a feature introduced in OpenGL 4 and backported as an extension in OpenGL 3: `glMapBuffer`. It maps a VBO (GPU-side) into the client memory (CPU-side). This way we can map the buffer at the beginning of a frame, make the changes into memory and then commit at the end of the frame. Moreover we can give some usage hints addressed to OpenGL concerning this mapping: we set a write only flag thus the driver can optimize even more. One particular aspect of our implementation is that we allocate the VBO at the beginning with the maximum number of triangles and we draw the complete buffer every frames and we make sure the unused triangles are null or “empty”. By empty we mean that these triangles are drawn at the origin with a size zero, thus they are invisible. We found experimentally the cost of drawing null triangles to be negligible. This little trick enables us to have holes in our array otherwise we would have reimplemented a whole memory manager in our application (segmentation and defragmentation of memory).

7 Results

In this section we present how our program performs in practice on modern hardware and we discuss the impact of the different optimizations we tried on the performance. Then we presents screenshots that demonstrates the visual results we obtained with the presented methods. We present the results on two machines where one is adapted for demanding graphics and the other one uses an integrated graphic card, see Table 5. We did the most tests on the first machine and used the second one for our last test on low-end/high-end hardware comparison. We summarize the configuration the following tabular. We measured the frame rate with respect to the number of triangles in the scene on GNU/Linux. To ensure we can compare the results we used a predefined path which is followed by the camera then the program exits and prints the total number of frames which we use as a performance measurement. This path begins in space, flies longly through the planet surface and ends in space again. We use a window size of 1024 × 768 and we disable the frame-cap in the SRAM if not specified otherwise. To measure the FPS we used gDEBugger[6] which is a free specialized software to debug and capture the performance into graphs in real-time.

	High-end	Low-end
Processor:	Intel Q9450	Intel 2520M i5
Graphic card:	NVIDIA 285 GTX	Intel HD 3000
RAM:	6 Go	4 Go
Linux:	3.6.11	3.9.1

Figure 5: The machine configurations we used in our tests

7.1 Performance

We now talk about the performance of some important optimizations we made in more details:

Partial update: The first implementation used a loop that uploaded all geometry every frames but the final version uses an incremental update: when triangles are changed they are uploaded to the GPU too. The total number of frame before was: **270** frames and now: **480** frames, gain of +77%.

Quadrant zones: The planet is split into height quadrants where one only one is updated based on the camera position. We disabled clouds/water shaders. Before: **438**, after: **469**, +7.1%.

Activation distance: When the camera is too far in the final version we don’t even update the mesh of the planet. We disabled clouds/water shaders. Before: 473, after 469, -0.85%. Note that in the predefined test we don’t spend time in space thus this is the impact of the distance test.

SRAM deadline: We implemented a system that exits more early in the mesh update to guarantee a greater frame rate. We stressed the deadline by enabling all shaders. Before: **458**, after: **637**, +39%. Our SRAM deadline target was set to 15 ms and 15.28 ms was attained in average with this approach. The cost of having this test when shaders are disable is 4.7%. This can be tweaked with an internal constant.

Change internal DS: The planet is stored as three big arrays of vertices, triangles and diamonds in a data structure (DS). This DS is iterated over every frames thus it is a sensible part for performance. The initial design of this DS was a sparse array (with holes) and we implemented a new version with linked list, that allowed us to iterate only on used entries. The impact was not predictable thus we tested this change. In the end it was favorable. Before: **307**, after: **403**, a gain of +31.3%.

Some optimizations were clearly worth the effort according to these results. In fact it depends on the utilization of the planet. If it is used as a background object then

[6]official website: `http://www.gremedy.com/`.

the quadrant zones and especially the activation distance are crucial although they are not for this test.

We present the performance of the different part of the program. The number of triangles stays close to 207'000 triangles whereas the VBO was allocated for 400'000 triangles. We plotted in Figure 6 the original performance and multiple cases where we disabled one feature to see its impact. We can clearly see the impact of the cloud and water shaders whereas the SRAM has nearly null impact (with the deadline enforcer). Moreover we see that the SRAM deadline is essential to target a higher frame rate.

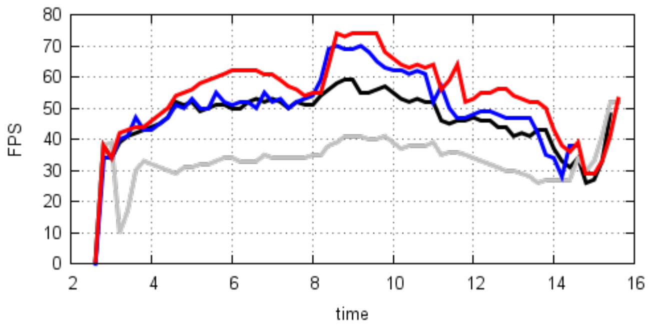

Figure 6: Performance in frame rate for multiple cases. Black: all; grey: no SRAM deadline, blue: no SRAM, red: no clouds/water.

We measured the different shaders alone and we came to the following results. The clouds alone have an average FPS of: ∼ **760**, water alone: ∼ **1500** and the planet surface shader alone **54.06** FPS. Thus we conclude power exponentiation (in the cloud shader) is more costly than multiple if/else (in the water shader). Finally the planet surface shader is the most costly of all our shaders without surprise.

We plotted the CPU utilization on Figure 7 during the same test capture. We can see a strange result: the processor is more busy when SRAM is disabled. This result may be explained by the fact that some synchronisation mechanism in OpenGL happens at the end of the planet mesh update (when we unmap the VBO buffer), thus lowering the CPU utilization.

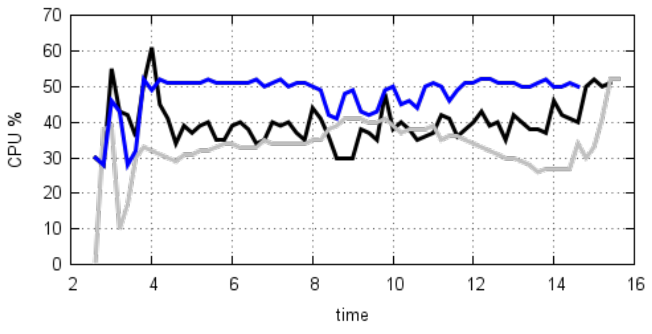

Figure 7: Use of the CPU. Black: all, grey: no SRAM deadline, blue: no SRAM.

Finally we compare the frame rate between the two machines in Figure 8. We enabled all features for this test. As expected the integrated card performs worse than the dedicated one. However the average frame rate, which is ∼ 30 FPS, is more than acceptable for interactivity.

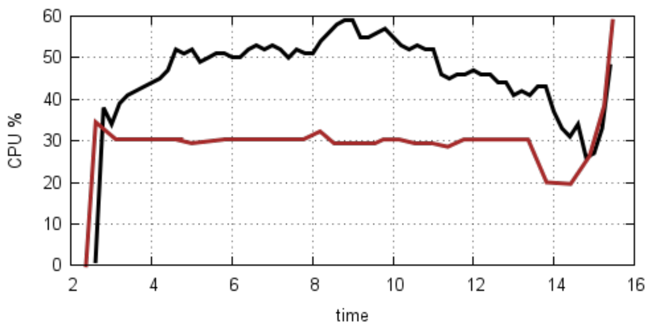

Figure 8: Frame rate comparison. Black: first machine, brown: second machine.

7.2 Screenshots

We present some screenshots of one planet from space to the surface in Figures 9, 10, 11.

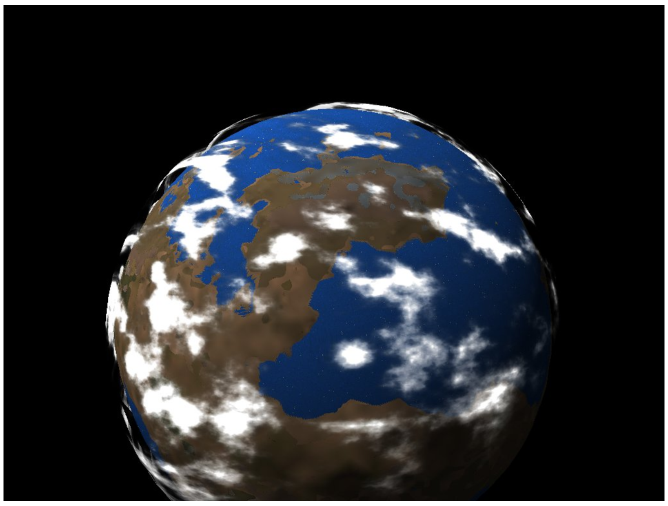

Figure 9: View of the planet from space

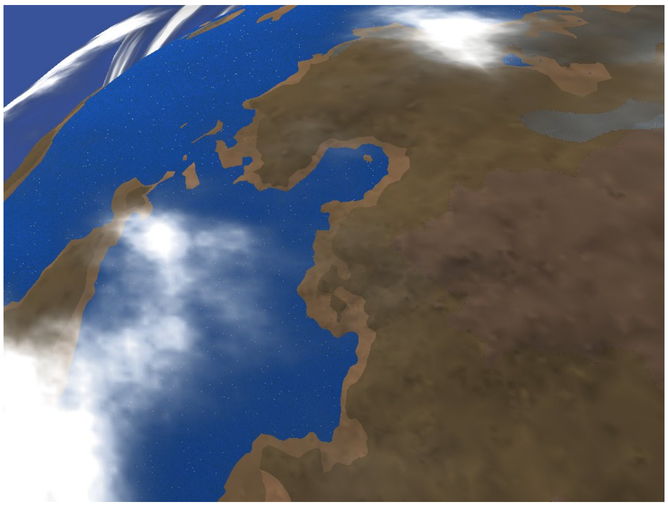

Figure 10: Aerial view of the planet

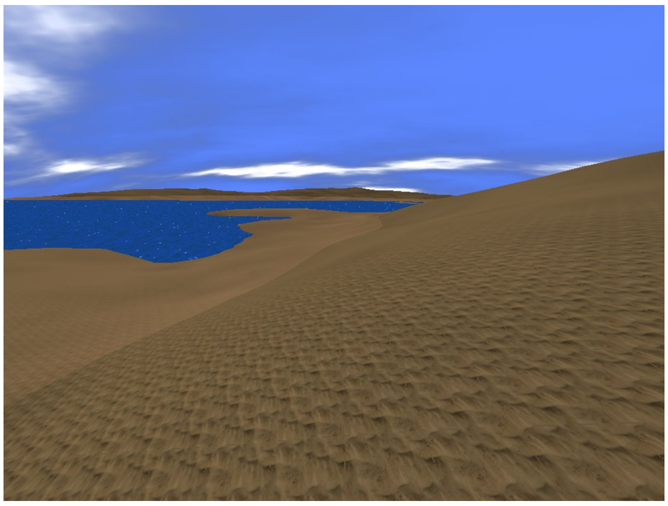

Figure 11: View of the planet surface

8 Conclusion

To conclude we developed a program to generate and draw procedural planets in real-time nearly from scratch. We learned valuable experience on shaders with OpenGL, on the performance problems that arises in practice with the compromise of computation on the CPU and the GPU and we tested different approaches to build noise signals and the way to use them. The resulting graphics are not state-of-the-art but it has the advantage of running fairly well even in more modest machines but running with a good frame rate on more modern machines. Our target frame rate of 60 FPS was not always attained on our high-end machine but the interactivity is good enough in our demo: above 40 FPS in average and stays over the cinema standard (below 30 FPS) for both machines.

Unfortunately we lacked some time to add more interesting features, besides clouds, water and planet texturing with blending, such that: surface vegetations, atmosphere scattering or a better shader for the planet surface with for example rivers [DGGK11] or fully procedural textures. Considering what can be done today [Bre13], our performance could be improved especially in the SRAM algorithm.

Finally we achieved some good results considering the initial code and the goals. This project really gave us valuable insights on procedural content generation and ideas of all it can offer in practice.

Acknowledgements

As the last words we would like to thank: Prof. Mark Pauly for accepting this project, Dr. Boris Neubert who followed and helped us a lot and finally Luc Girod and Regis Blanc who reviewed this report.

References

[Bre13] Flavien Brebion, *Infinity development blog*, `http://www.infinity-universe.com/Infinity/index.php?option=com_content&task=blogcategory&id=0&Itemid=47`, 2013, accessed 30-May-2013.

[DGGK11] Evgenij Derzapf, Björn Ganster, Michael Guthe, and Reinhard Klein, *River networks for instant procedural planets*, Computer Graphics Forum **30** (2011), no. 7, 2031–2040.

[DWS$^+$97] Mark Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein, *Roaming terrain: Real-time optimally adapting meshes*, Proceedings of the 8Th Conference on Visualization '97 (Los Alamitos, CA, USA), VIS '97, IEEE Computer Society Press, 1997, pp. 81–88.

[KCODL06] Johannes Kopf, Daniel Cohen-Or, Oliver Deussen, and Dani Lischinski, *Recursive wang tiles for real-time blue noise*, ACM SIGGRAPH 2006 Papers (New York, NY, USA), SIGGRAPH '06, ACM, 2006, pp. 509–518.

[LLC$^+$10] Ares Lagae, Sylvain Lefebvre, Rob Cook, Tony DeRose, George Drettakis, D.S. Ebert, J.P. Lewis, Ken Perlin, and Matthias Zwicker, *State of the art in procedural noise functions*, EG 2010 - State of the Art Reports (Helwig Hauser and Erik Reinhard, eds.), Eurographics, Eurographics Association, May 2010.

[Mus00] Kenton Musgrave, *Procedural fractal terrains*.

[O'N01] Sein O'Neil, *A real-time procedural universe*, `http://www.gamasutra.com/view/feature/3042/a_realtime_procedural_universe_.php`, 2001, accessed 30-May-2013.

[Per99] Ken Perlin, *Making noise*, 1999, Talk presented at GDCHardCore.

[sof13] VTP software, *Artificial terrain generation*, `http://vterrain.org/Elevation/Artificial/`, 2013, accessed 30-May-2013.

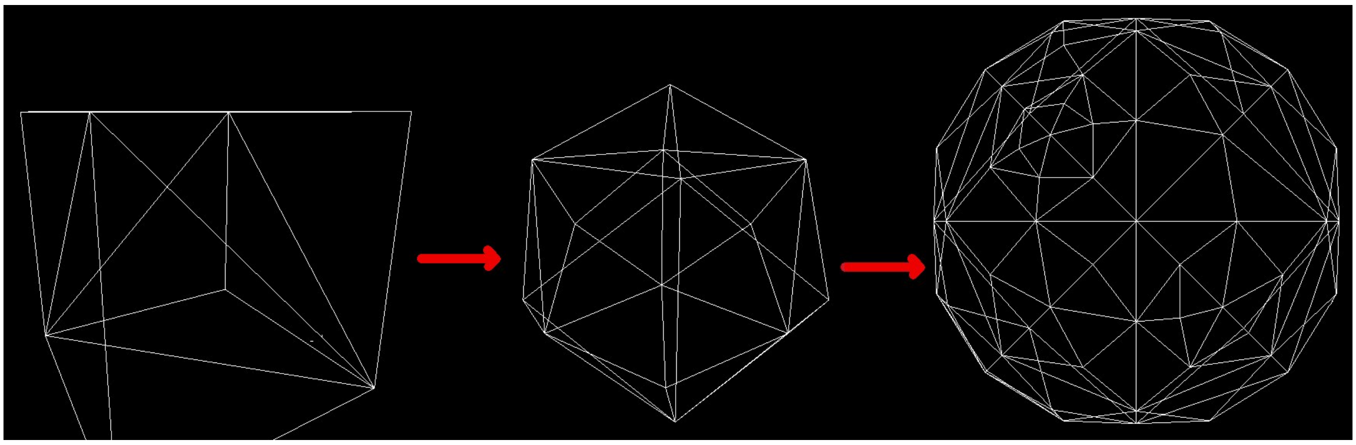

Figure 12: Our spherical ROAM from cube to hexagons.

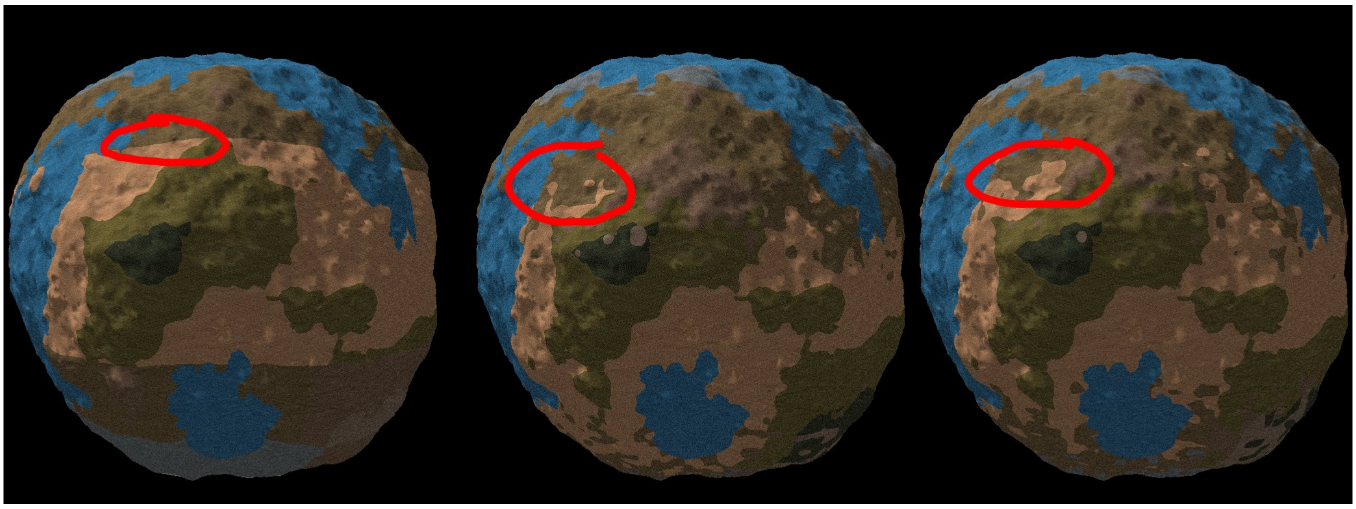

Figure 13: Comparisons between polar coordinates disrupted with an increasing factor of Perlin noise.