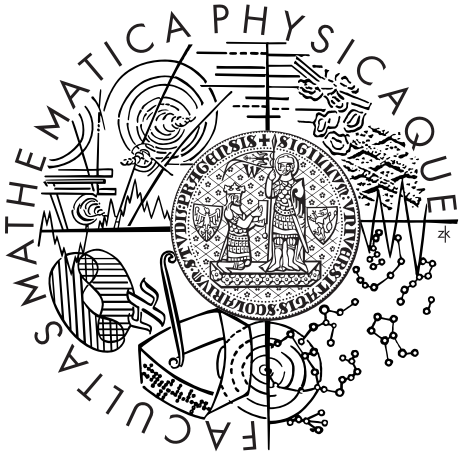
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Martin Kahoun

Realtime library for procedural generation and rendering of terrains

Department of Software and Computer Science Education

Supervisor of the master thesis: Mgr. Jan Horáček
Study programme: Computer science
Specialization: Software systems

Prague 2013

I would like to express my gratitude to my supervisor, Mgr. Jan Horáček, for his support, availability, help and guidance. Many thanks to my family and friends for their support and never ending patience. Very special thanks to my girlfriend who has had the patience with me and supported me to get this work done.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague, August 2, 2013 Martin Kahoun

Název práce: Realtime library for procedural generation and rendering of terrains

Autor: Bc. Martin Kahoun

Katedra: Kabinet software a výuky informatiky

Vedoucí diplomové práce: Mgr. Jan Horáček, KSVI

Abstrakt: Techniky procedurálního generování grafického obsahu zažily za posledních třicet let širokého užití. Jedná se o stále aktivní oblast výzkumu s aplikacemi v 3D modelovacím software, videohrách a filmech. Tato práce se zaměřuje na algoritmy pro generování a zobrazování virtuálních terénů v reálném čase. Prezentujeme přehled dostupných metod a některé z nich poskytujeme v rámci rozšiřitelné knihovny pro syntézu procedurálních krajin.

Klíčová slova: procedurální terén, fraktální krajina, úroveň detailů

Title: Realtime library for procedural generation and rendering of terrains

Author: Bc. Martin Kahoun

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Jan Horáček, KSVI

Abstract: Techniques for procedural generation of the graphics content have seen widespread use in multimedia over the past thirty years. It is still an active area of research with many applications in 3D modeling software, video games, and films. This thesis focuses on algorithmic generation of virtual terrains in real-time and their real-time visualization. We provide an overview of available approaches and present an extendable library for procedural terrain synthesis.

Keywords: procedural terrain, fractal landscape, level of detail

Contents

Introduction

Procedural graphics content generation has a very long history dating to 1980's. It is a technique for creating models, textures, and effects in an algorithmic way. As such it is widely used in multimedia: computer graphics software, video games, and the movie industry. One of the first appearances of the procedural techniques was in the film *Star Trek II: The Wrath of Khan* (1982) where a computer generated sequence of a barren planet undergoing a rapid terraformation was created.

Since then the procedural techniques have found a way into many applications, and interfaces for procedurals have been implemented in many production renderers such as Pixar's RenderMan. Books and papers have been written on the subject, among the most popular ones is *Texturing & modeling: A procedural approach* [13]. Examples of procedural content used in video games include: *Elite* (1984) which sported several procedurally generated galaxies with solar systems; *Spore* (2008) where creatures and whole planetary surfaces were generated algorithmically; *Borderlands 2* (2012) featuring unique system of weapon procedural generation from defined set of components; *The Elder Scrolls IV: Oblivion* (2006) with procedurally generated terrain based on general characteristics set by the developers; and *.kkrieger* (2004) a first person shooter where all of the content is generated procedurally at a game start, the game fits in 100KiB.

These techniques can be used to abstract the content creation, so rather than storing all the details of a complex model, for example, we can describe that model with an algorithm. Not only do we gain storage space savings, we can also cut down the costs of having artists to do the work on the model while having a great flexibility in respect to the model modification.

These techniques can be used to rapidly create lots of content that would be otherwise too expensive or nearly impossible to create (due to the cost or time constraints for example). They can be employed to provide replay value for games, add procedural details at run-time on in-game characters, create non-repeating realistic texture of a surface, or generate objects of nature such as vegetation or trees.

We have delved into the area of procedural terrain synthesis in our previous work [18] and we were interested in its generalization and improvement. This work has served as a basis for this thesis that aims to *1*) map out the area of procedural landscape generation in more detail; and *2*) provide a solid framework for further development, research, and experimentation. Our motivation is to provide a set of extendable functions that can be used to create virtual landscapes in game projects, or to serve as a basis for terrain modeling software.

The rest of this thesis is organized as follows: in chapter 1 we present an analysis of the studied techniques for procedural terrain synthesis; chapter 2 presents the selected methods in more depth; chapter 3 describes our implementation of the chosen techniques; and in chapter 4 we discuss the results and make an assessment over the implemented methods and the final product.

The library is provided on the accompanying CD (see appendix C for its contents). We provide a brief manual to the software in appendices A and B.

1. Problem analysis

In this chapter we analyze the problem of the procedural terrain generation and its rendering in real-time. First, we focus on the pros and cons of using procedurals in general. Then, we review different approaches that can be used for procedural terrain synthesis. Next, we discuss the level of detail (or LOD) algorithms that can be used for rendering large digital elevation maps (or DEM) in real-time. We conclude this chapter by discussing techniques for creating terrain textures.

Let us first state the goal of our work. We aim to to provide framework for creating and rendering DEM in real-time [1]. Rendering of a DEM in real-time is important for flight simulators, map programs, or strategy games, to name a few examples. It can get quite complicated depending on the size of the terrain. As the size grows we need to employ LOD algorithms as well as advanced data structures for storing and retrieving the DEM because the amount of data can — and most of the times will — grow very large.

In our work we focus only on the LOD part of the problem as we create the DEM on the fly using procedural algorithms. Thus, we avoid the storage and retrieval problems when working with measured datasets such as USGS (U.S. geological survey). Instead, we create the height-field just as we need more details which *a*) dramatically reduces the memory footprint to only displayed vertices; and *b*) allows for theoretically unlimited resolution[2]. This is also an objective of this work.

1.1 Advantages and disadvantages of procedural techniques

Use of the procedural content generation has lots of advantages but also some disadvantages. It is not suited for every purpose and one must know the limitations in order to decide whether to use it or not. The biggest advantage of the procedural approach is its randomness. Often, the methods employ some sort of pseudo-random noise to generate patterns. We can get quite naturally looking materials like wood or marble just by algorithmically defining their structure and we will get almost infinite number of variations and patterns just like in nature. This is something even a skilled 2D artist can't do very easily.

However, too much randomness is often criticized on procedurals along with the lack of artistic control over their output. With procedurals we have only the *parametric control*, that is we have a set of defined input parameters that affect the outcome and some of the techniques produce dramatically different results for even the slightest input parameter changes. On the other hand, we can assign sensible meaning to the parameters, e.g., terrain roughness. We can spare a lot of space by using procedural textures and we can avoid texture repetition problems. It will also allow for rapid content generation and cost reductions because we simply don't need to pay for the expensive artist's work. One such example could be *The Elder Scrolls IV: Oblivion* (2006) where the procedural techniques

[1] By real-time we mean rendering at least 30 frames per second.

[2] Depending on the chosen method, floating point accuracy and the *Nyquist frequency*

were employed to speed up the process of landscape creation. This, however, has led to some criticism stating that the landscape is blunt, vast, and mostly vacant. In the follow up game — *Skyrim* — Bethesda has returned to the handcrafted landscapes.

Another aspect of the procedural content generation is the element of surprise as we can obtain pleasing, yet unexpected, results from the procedures, especially stochastic ones. In the end, the suitability of the procedural approach depends on the use case scenario and design goals of the concrete project.

1.2 Terrain synthesis

Algorithms for terrain synthesis can be divided into two main categories, we may call them static and dynamic, based on their *modus operandi*; alternatively we can look at them as fractal based synthesis or physical erosion simulation — but, as we will see later on, these two approaches often overlap. By static we mean such algorithms that compute the whole desired terrain mesh and store it in the memory. The calculations may take considerable amount of time and once the algorithm finishes the resulting model has limited resolution in which it was computed. However, this approach lends itself well to erosion simulation for example. The main disadvantage of this approach is its disability to provide height information at arbitrary points. This has several consequences: *a*) we have to store the height-field in memory and on the disk because it is costly to recreate it; *b*) we are limited by the resolution of the calculated height-field; and *c*) we must tailor the LOD algorithm to use exactly the height-field grid points or implement some sort of interpolation. As stated above, we focus on the real-time terrain synthesis and algorithms requiring considerable amount of pre-computation time are of little use to us.

In our previous work [18] we have mentioned two examples of static algorithms. There is the Fault lines algorithm [15] based on the idea of depressing and elevating the terrain along randomly placed fault lines. By our previous definition this algorithm belongs to the static group, and when used to generate spherical terrain it produces a planetary body where one side is an inverted mirror image of the other side.

A better way to generate a terrain is the midpoint displacement algorithm, or the diamond-square algorithm which is a variant of the former one [16, 24]. Both begin with four vertices forming a square, height values are assigned to these vertices and then the square is subdivided into four sub-squares. Each new edge vertex is assigned a mean value of the original corner values composing that edge, while the center vertex is the mean of all corner values with random offset proportional to the square size. This method is sometimes called the Plasma fractal and with some work it can be modified to yield values on demand as mentioned by O'Neil [28]. Unfortunately, this algorithm is flawed (as discussed by Miller [24]) as it produces visually disturbing artefacts. Another drawback is its simplicity as it creates endless mountain range, again O'Neil [28].

This brings us to a category of dynamic algorithms which require little to no pre-computation and are able to evaluate arbitrary points without interpolation of the height-field. Many of these methods are based on evaluation of some persistent noise function, examples can be found in the *Texturing & modeling* [13, p. 67–83]

book, or in the article by Ken Perlin [30]. Often, the noise values are sampled at several scaled successive frequencies and summed up together to obtain a fractal geometry which is quite common in the nature, see the work of B. Mandelbrot [21] or F. K. Musgrave [13, ch. 14, 15, 16, 20]. Given this, we can only store the input parameters of the selected method and use them to recreate the terrain at any resolution any time we see fit. An obvious disadvantage of this approach is the disability to make complex calculations inside the heightmap query because we need the method to execute quickly, should we want to use it in real-time. As a consequence we are limited in the final terrain appearance which may lack the details obtained via thermal and hydrodynamic erosion. To a degree, these features can be created by carefully crafted methods of noise values manipulation, but it should be stated that we cannot expect realistic results. However, we can get results that look quite natural and may be satisfying for our intents, we can also get surreal landscapes using these methods.

A different approach for terrain synthesis is the physical erosion simulation. It is often combined with the fractal landscape synthesis — first, the fractal terrain is generated and the result is then used for physical simulation. One such example can be seen in the work of F. K. Musgrave [25], or in more recent works by P. Krištof et al. [19] or M. Hudak [17]. Other works focus on creating rivers: there is the RiverLand by S. T. Teoh [37] which allows users some degree of interaction (namely defining ridges the rivers can't cross), the approximate river path is then generated from the river mouth backwards to its spring. The algorithm then fits curvature between the control points to create meandering river, tributaries are also grown. When the river network is completed it is then used to constrain the height-field in a way that no mountains can occur where the river flows. While this approach produces good results, in our opinion, the river networks generated by this algorithm look too artificial due to the curvature.

More naturalistic results come from the work of Belhadj and Audibert [2]. They begin with flat height-field. First, they create a set of ridge lines and deform the elevation map to obtain smooth elevation profile. In the next step, rivers are created starting on the ridge lines and using method similar to the work of Chiba et al. [7]. Then, the DEM is reset and contains nodes with three possible states *a*) the grid point belongs to the ridge line and its height is thus known; *b*) the node is on the river path and its height is also known; or *c*) the grid coordinate has not been computed yet. Two midpoint displacement processes are then applied to calculate the height-field between the ridge lines and rivers. One of the original authors later revised the method [1] removing its drawbacks and has shown that it can be used for data reconstruction, i.e., taking a DEM acquired by satellite and filling the gaps using the discussed algorithm. User-guided terrain synthesis from a database of examples has been proposed by Zhou [39]. Finally, there is an interesting work of J. Schneider et al. [34] proposing user guided interaction with fractal basis functions and thus allowing better control over the resulting terrain.

Other natural phenomena

Apart from terrain synthesis, we may employ procedural techniques for simulation or generation of other natural phenomena that can be used to enrich the virtual worlds we want to create. However, these topics are outside the scope of this thesis, should the reader be interested in them we present very brief overview

of sources for further research. A good starting point is the already mentioned *Texturing & modeling* [13] book. We can also recommend taking a look at the 2004 SIGGRAPH course *The Elements of Nature* [11].

Most significant phenomena that greatly improve the aesthetic qualities of the synthetic terrain include: atmospheric light scattering [32, ch. 16] [3, 14]; water surface rendering and water light scattering [5]; water flow simulation [38]; and vegetation [33, 4].

Available software

To our knowledge, some of the aforementioned techniques have been adopted into a commercially available software. Probably the most popular choice amongst visual artists is the Terragen by the PlanetSide Software. It is still in active development and older versions are offered as freeware. It generates 2-dimensional heightmaps that can be exported into other modeling or image manipulation software to create virtual terrain meshes or skyboxes. Another well known software is the MojoWorld originally created by F. K. Musgrave. Both feature fractal based procedural terrain synthesis with global parametric control. Other examples include Grome, Bryce, or World machine. These also feature fractal based terrain synthesis but add erosion models and provide the user with terrain brushes.

During our research we have found an open source project[3] by Mark Ridgewell, however, it is focused for offline terrain generation and rendering. We have also reviewed two works concerning the user interaction with the process of the terrain creation. The first [31] of these presents an experimental application where the user is allowed to stack up multiple generators and modifier functions to create the terrain. The second [40] presents a scripting language that can be used to define a complex terrain generator that can be then run from the script interpreter. We should also mention the libnoise library[4]. It is an open-source library that can be used for generating coherent noise. It comes with some examples how to use the library's noise functions to create a rather complex terrain.

1.3 Terrain rendering

As mentioned in the beginning of this chapter, terrain data sets can get very large. So in order to render them efficiently in real-time we need to employ LOD algorithms to distinguish important areas that deserve more polygons, from areas that are far enough to be approximated just by a few polygons. We will briefly discuss several algorithms that can be used for real-time terrain rendering.

1.3.1 Real-time Optimally Adapting Meshes

Let us begin with a very popular algorithm created by M. Duchaineau et al. [12], it is called Real-time Optimally Adapting Meshes (or simply ROAM). It is a CPU based algorithm that constructs hierarchical structure of right isosceles triangles above the terrain heightmap. The tessellation changes at runtime to provide accurate representation of the terrain.

[3] Available at `http://www.markridgewell.co.uk/planet/index.htm`

[4] Available at `http://libnoise.sourceforge.net/`

There are several rules defined to change the triangulation to ensure that the resulting mesh is water-tight. A triangle split is permitted only one level deeper than its neighbours, otherwise, recursive neighbour splitting is enforced. No longer needed triangles are merged back — but only one level higher in the hierarchy. The algorithm is driven by an error metric that can be based on the camera position and orientation relative to the surface and the difference between the triangle hypotenuse midpoint and the corresponding heightmap sample.

The algorithm can be relatively easily adapted for rendering spherical landscapes as shown by O'Neil [29] and in our previous work [18]. Also, the error metric makes the algorithm very flexible in terms of decision about the split-merge operations, thus, we can implement an error metric that suits our needs.

1.3.2 Geometry clipmaps

A different approach to terrain level of detail has been presented by Lossaso and Hoppe [20]. Their algorithm — geometry clipmaps — is based on the idea of using texture clipmaps [36] on an elevation grid to obtain a hierarchy of regular grids of varying density. A fine grained resolution of the heightmap is presented in an area immediately around the observer with bands of progressively coarser grids appearing as the distance from the camera increases. There are two main drawbacks with this technique: first, the occurrence of T-cracks in the mesh between regions with different LOD, which can be to a degree masked by blending; second, the LOD is based purely on the distance from the viewer and doesn't account for terrain features. Later, the algorithm was updated to be able to run on the GPU and then Classen and Hege [8] have adapted the algorithm for spherical landscapes.

1.3.3 Other techniques

Today, with the advent and general widespread of programmable GPU's, the trend in the terrain LOD field is to move most, if not all, calculations onto the GPU. There are several other methods we feel to mention apart from the two above (the list is far from definitive), we refer the reader to look for details into the respective sources. There is a white paper [6] from NVIDIA Corporation that presents method used for terrain LOD in *Tom Clancy's H.A.W.X. 2* — it strongly resembles the geometry clipmaps but takes into account the terrain profile.

Schneider et al.[34] have used technique called projected grid. This method runs on the GPU and works in a way of projecting screen space regular grid onto the fractal surface. As noted by the authors there are notable limitations of this approach and for their future work they wanted to investigate the suitability of geometry clipmaps. J. Schneider [35] has also worked on a technique suited for large textured terrain data sets that is GPU friendly.

1.4 Terrain texturing

There is a plethora of texturing techniques that can be applied to virtual landscapes, however, many of these assume that we have the whole terrain mesh at

our disposal. In his work [27], K. Nicholson presents a run-down of techniques for texturing of 3D terrains.

A simple approach applicable on terrains of known size would be to sample the procedural heightmap and use the information to construct one large terrain texture from several terrain type textures, e.g., taking several terrain type textures and stitching them together based on the height at the texel coordinates. However, this approach has a serious drawback — the terrain geometry can be quite large and the texture size is limited by the GPU memory and texture unit constraints. We would end up with blurred texture that would lack the details in close up view.

Far better solution would be to use the aforementioned texture clipmaps [36]. This way, we can have very large terrain texture (for example 32k on 32k resolution) stored on the disk (we could either synthesize it or use satellite imagery) and stream only the relevant data in the appropriate resolution to a much smaller texture that fits into the GPU memory. These methods can be found under several different names: there is the original method called texture clipmaps; some of the recent derived works are called mega textures (as used in the *IdTech4* engine) and virtual textures [27, 31].

Unfortunately, these methods are far from useful when dealing with arbitrary procedural terrains, on the other hand they provide artistic control over the final result and this is probably one of the reasons for their adoption in the current game engines. In the previous work [18] we have resorted to procedural texturing done in shaders. When a new vertex is created texture coordinates are calculated. In the fragment shader the height of the current fragment is retrieved using the pre-sampled heightmap texture and based on that value the respective terrain texture is used or blended.

This approach creates the terrain texture on the fly and doesn't require one large terrain texture as the technique discussed in the beginning of this section. The downside of this method is the increased number of texture fetches, for we need to fetch texels from all possible terrain textures, even-though, they might not be used at all. On the other hand, we have a fine control over the apparent resolution of the final terrain texture. For example, in close up views we can adjust the texture repetition factor to obtain detailed ground texture.

1.4.1 Removing texturing artefacts

As mentioned, usually, the terrain texture is tiled across the terrain. Even-though we use seamless textures, that is a textures that can be tiled without visible seams, the tiling will be evident by its noticeable repeating pattern. An example of this artefact can be seen in figure 4.1 on the left — notice the repeating pattern of slanted stripes. A simple, yet powerful, workaround is possible — adjusting the repetition factor based on the distance from the terrain, thus creating an effective texture LOD. For hiding the LOD transitions we will blend between the two consecutive LOD levels as shown by Nicholson [27].

A more elaborate way of removing the periodicity is to synthesize a texture using samples from the original ground texture we have intended to use. Widely popular technique for doing so is the use of Wang tiles [9, 27]. This way, we can use samples of a single texture to create much larger one by aperiodic tiling. It

can be done either manually or procedurally. Nevertheless, we must be aware of the potential problems with this technique. Often the input texture contains shapes that we humans know well but the algorithm splits them, thus creating very noticeable artefact.

Another troublesome artefact concerning the texturing of arbitrary terrains is the texture stretching. This usually occurs when the concerned polygon is not oriented with the texture projection plane. For example, when we map the x and z terrain geometry coordinates to the s and t texture coordinates, the texture stretching would affect steep slopes. The idea how to remove this problem is to change the parameters of the planar projection so they would match the polygon orientation more closely, and that is exactly what the tri-planar texturing algorithm [26, ch. 1] [27] tries to do. In short, the method calculates three planar projections along x, y and z axes and then chooses the best one based on the surface normal, blending in the other two to avoid visual artefacts. The only drawback is the need to make 3 texture fetches per terrain type [18]. One possible inconvenience is a visible texture blur in some areas introduced by the blending.

1.4.2 Spherical terrains

There is one more thing that needs to be addressed, and that is the texturing of spherical terrains. A classical way of applying texture to a sphere is unwrapping its surface into a rectangle with 2:1 aspect ratio and using the spherical projection to distort the texture space. There are several drawbacks with this method, however, the texture will be stretched close to the poles and visible artefacts on the poles themselves will appear, especially the latter is quite disturbing. It is caused by the singularity on the poles where the single texel is stretched along the texture base.

In our previous work [18] the requirement was prevention of such artefacts. We needed to pass the height information into the GLSL shaders via a heightmap texture. In this case we have sent 6 heightmap textures that formed a cube and then we have used the cube-mapping technique to project them onto the surface. While this removes the aforementioned artefacts at the poles it introduces slight texture stretching at the cubemap corners (this can be very visible on the normal map). A care must also be taken on the cubemap face boundaries due to the possible value repetition and the fact that older OpenGL implementations didn't perform linear filtering between the cubemap faces. The latter is usually not a concern these days [22, p. 241].

2. Solution analysis

In the previous chapter we have made a brief introduction about possible ways of implementing the virtual terrain rendering and synthesis. In this chapter we will take a closer look on the techniques we have chosen for our library. We will delve into the necessary details and make some assessments about their suitability, advantages, and shortcomings. The overall summary and application examples are given in chapter 4. The implementation details are provided in chapter 3.

2.1 The ROAM algorithm

Again, we have chosen the ROAM algorithm for the terrain LOD. We have done so because of its suitability for flat and spherical terrains, relative ease of implementation, and its flexibility (as mentioned in section 1.3.1).

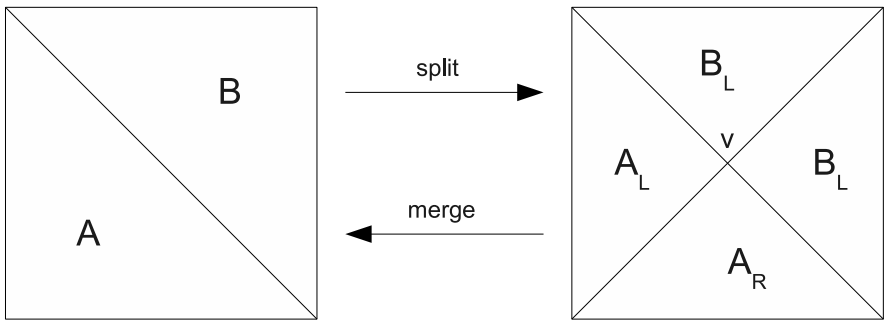


Figure 2.1: ROAM split and merge operations. Diagram was reused from our previous work [18] and updated.

In essence, the ROAM algorithm is a binary tree representing a triangulation of a mesh with a set of node manipulation rules. Each node in this tree represents a single right triangle with the root node being the whole terrain, and the leaves equal to all triangles to be rendered. In practice, we will need two ROAM trees to represent a single flat terrain patch — the root nodes will share the hypotenuse edge as seen in figure 2.1 on the left — we will call this shape a square. Each square is a potential candidate for splitting which will introduce a new shape called a diamond (four triangles sharing a vertex by their right angle) seen in figure 2.1 on the right. Each diamond is a potential candidate for merging. An error metric is calculated for each leaf triangle in order to determine whether the triangle needs to be split or merged — when the calculated value crosses some predefined threshold the respective operation is triggered.

An example of a triangle split-merge operation can be seen in figure 2.1. When the algorithm determines that a square of triangles A and B needs to be split, a new vertex v introduced as well as four new triangles one subdivision level deeper — triangles A_L, A_R as children of the triangle A, and triangles B_L, B_R as children of the triangle B. When the algorithm later determines that the diamond A_L, A_R, B_L, B_R is no longer needed, these leaves are deleted and we move one level higher in the hierarchy.

As mentioned in section 1.3.1, to prevent cracks, or discontinuities, in the mesh we must ensure that the subdivision (or tree) depth difference between the

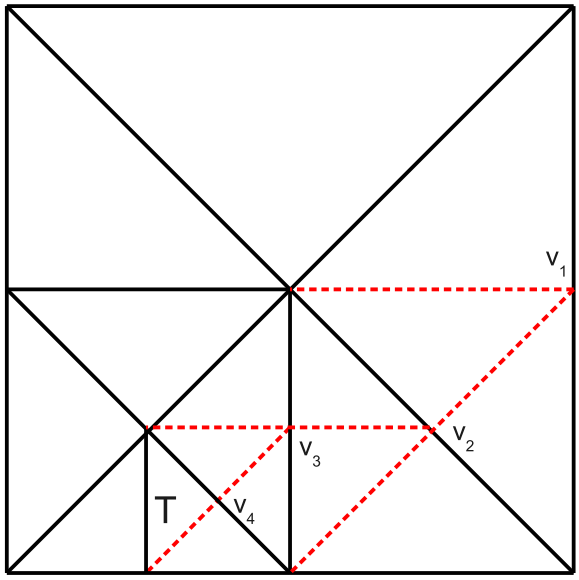


Figure 2.2: ROAM recursive splitting. Diagram was reused from our previous work [18] and updated.

neighbouring triangles in the mesh (the ROAM tree leaves) is at most 1. This is achieved by allowing only triangles in a square configuration to be split. Thus, when a triangle we are about to split does not form a square we must first split its neighbour along the hypotenuse edge. This will often require several more recursive splits as seen in figure 2.2. There, in order to split a triangle T and introduce the vertex v_4, we need to perform several other splits indicated in a dashed line. This will introduce vertices v_1, v_2 and v_3 ensuring that a triangle T is part of a square and thus can be split.

In the case of a reverse operation, the requirement that triangles form a diamond also ensures the depth constraint. Each split operation will introduce exactly one new diamond and destroy between 0, 1, or 2 existing ones. Analogously, the merge operation will destroy exactly one diamond while creating up to 2 new ones. A special care must be taken when operating on triangles that lie on the world boundaries in order to prevent memory leaks and to handle special cases where only a half of the diamond is created as the other half lies outside the terrain boundaries and is not created (triangle T in figure 2.2 is a part of such diamond).

2.1.1 The spherical variant

Adapting the ROAM algorithm for spherical terrains is a matter of several slight changes [29, 18]. Instead of a single ROAM square representing a patch of terrain, we will take six such squares (12 ROAM triangles) and use them to build a cube configuration. As an added bonus we won't need to be concerned with the special cases on the world boundaries mentioned above, as all triangles will now have proper neighbours. One other change concerns the calculation of a new vertex position upon a triangle split: we take the appropriate height of that vertex obtained from the heightmap, add a base sphere radius to it, and use the resulting value to multiply the direction vector towards the new vertex to obtain

its position.

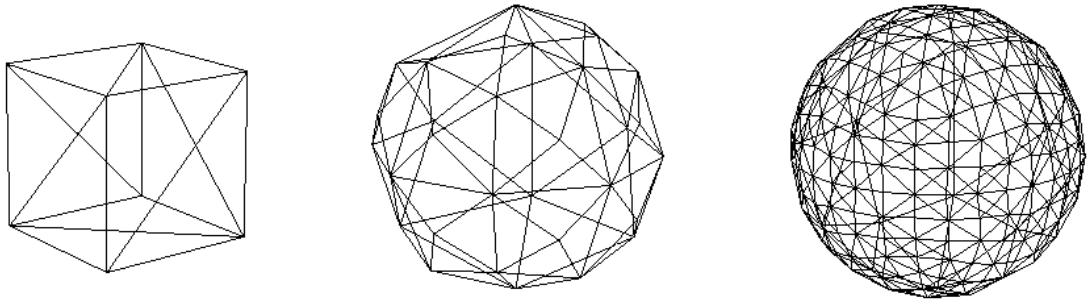


Figure 2.3: Spherical ROAM at various levels of detail. Image reused from our previous work [18].

There is no need for a more complicated basic shape (such as icosahedron) because the error metric combined with vertex position calculation will provide us with a nice looking sphere as evidenced by figure 2.3.

2.1.2 Implementation remarks

The easy and straightforward implementation of the ROAM algorithm uses the explicit binary tree. The upside of this approach is a relative simplicity of the triangle split-merge operations as we just add or remove 4 children at once. The downside is the fact that each update requires tree traversal which can be expensive in terms of CPU cycles, not to mention the memory required to hold the whole tree structure when only the leaves are of any concern to us (only the triangles at the bottom of the tree are updated or rendered).

Previously, we have used this approach in conjunction with a list of leaves used to speed up the rendering by not traversing the tree. As discussed by O'Neil [29] we can implement the whole ROAM algorithm with just only the list of leaves. We only need to cleverly adjust the split-merge operations to be able to track down which triangle spawned which and be able to track back the changes.

Let us analyze the naive implementation a bit closely. Assume, for a moment, that we want full details everywhere — we will obtain a balanced binary tree of height h. The number of triangles in the mesh will be equal to the number of the ROAM tree leaves, i.e., $n_0 = 2^{h-1}$, whereas the number of all ROAM tree nodes will be $n = 2^h - 1$. We see that n_0 renderable triangles require almost as many tree nodes above them. Theoretically, it is not a problem, but in practice code profiling has revealed that for a flat terrain patch of a maximal subdivision depth $h = 12$ the application spent 25% of its running time in the ROAM triangle update method responsible for the DFS. For a spherical variant this was only 7%, but only because of the larger execution times of procedural heightmap queries. Nevertheless, the ROAM triangle update method scored third.

Therefore we have changed the implementation to store only the list of leaves. This has decreased the total time spent on ROAM triangle update execution to about 1.8% for both versions. It is also advisable to keep track of all diamonds currently present in the structure to speed up the merging process, we have done so previously as we have done now. In addition, we have imposed limits on

number of split-merge operations per frame as these were identified as the main reason for sudden framerate drops (sometimes the application even appeared to freeze). This adjustment, however, required also an implementation of priority queuing of triangles and diamonds by their error metric in order to ensure that details will be added in the most important areas first.

This, and improvements in the error metric calculations, have also dramatically reduced the flickering triangle problem observed in our previous work [18, p. 50]. In the former implementation some triangles were split only to be merged again in the same or the upcoming frame resulting in visible vertex jitter. This time, the default error metric takes into account whether the triangle is on the horizon in addition to: distance from the observer; difference between the potential vertex position and the hypotenuse midpoint; orientation of the triangle relative to the camera viewing direction; and whether the triangle is even in the camera field of view.

The ROAM algorithm can also be extended to use geomorphing — animating either the new vertex to its position from the triangle hypotenuse or back if the said vertex is being removed. The idea is to create new vertex at the hypotenuse midpoint and then adjust its position over several frames. While the use of geomorphing reduces the sudden appearance of vertices (also know as popping), it also slows down the mesh updates as we need to morph the vertex to the new position and back before we remove it. This effect is very noticeable especially on the silhouette triangles. But we have designed our error metric to prioritize triangles lying on the horizon. These are therefore almost fully subdivided and the vertex popping is not noticeable. We have also found that the geomorphing may be even more disturbing than a sudden vertex appearance. Hence our current implementation lacks this technique.

2.2 Terrain synthesis

To keep up with our requirement of real-time DEM creation the implemented algorithms for terrain generation are based around fractal and multifractal synthesis. There are two main reasons for this decision. First, real-world terrain features exhibit fractal nature: rocks and mountains are self-similar for example, hence geologists put an item of a known scale next to the samples when taking pictures of them; river networks when viewed as a whole (e.g. from high altitudes) are self-similar too. This makes fractal based methods especially suitable for terrain synthesis. For more details we refer to B. Mandelbrot [21] Second, many of these methods are able to produce the synthetic height-field on-demand which is exactly the goal we wanted to achieve.

For the purpose of this work let us borrow a definition of a fractal by F. K. Musgrave [13, p. 431]: "a geometrically complex object, the complexity of which arises through the repetition of a given form over a range of scales". This definition is sufficient for us as the potential users of our library are merely the end users of the methods involved and a complex study of the theory behind is grossly out of the scope of this work. A simple fractal can thus be obtained by repeating the same pattern at different scales. We will get an object that retains its overall shape no matter how much we zoom in or out, only the details may vary.

The kind of fractal we are using, and that has proven usable for terrain synthesis, is called fractional Brownian motion (or fBm). First, we will need a source "shape" for the repetition, let us call it a basis function. A good basis function should have a stochastic nature in order to produce irregular, seemingly random patterns. The basis function should also be persistent, that is for the same input it should yield the same output, otherwise we would get different result each time the function is executed.

Good choice for such basis function is the Perlin noise function [30], although, it is not the only one that can be used, other methods work as well. The idea behind the fBm is simple, in essence, it is a weighted sum of basis function values at different frequencies. Weight usually changes between the successive frequencies and defines their amplitude. The difference between the successive frequencies is usually called lacunarity which is a Latin word for a gap. Because the lacunarity is usually equal to 2.0 we speak about the number of octaves[1], i.e., the number of successive doubled up frequencies.

A very simple fractal that can serve as a procedural heightmap can be formulated as follows

$$h_{\vec{p}} = \sum_{i=1}^{k} w_i \cdot f(n_i \cdot \vec{p}),$$

$$w_1 = 1,\ w_i = \frac{1}{2} \cdot w_{i-1},$$

$$n_1 = 1,\ n_i = 2 \cdot n_{i-1}$$

where f is our basis function and $\vec{p}$ is the evaluated point. Then $h_{\vec{p}}$ is the appropriate height for that point. Many of the implemented algorithms in the library are based on this simple formula, although, they are more complex and allow to change the input parameters. Namely, the amplitude, lacunarity and the number of octaves are user controlled.

This basic approach has one major flaw and that is its homogeneity. The produced height-field will result in an endless mountain range of the same roughness which is not a very common occurrence in nature. Therefore the methods used for demonstration purposes are more complex, they are sometimes called multifractals as the nature of the fractal constructed by them changes with the frequency. For details we refer to the *Texturing and modeling* book [13, ch. 16] as we have implemented some of the terrain functions presented there. For an overview of the implemented functions see section 3.2.

2.2.1 Generating rivers

As a part of the assignment we have tried to implement more advanced terrain generator based on the already mentioned that would add river networks to the final terrain. The idea behind our algorithm is loosely based on the work of Belhadj and Audibert [2]. In our case we do not know the terrain in advance so our approach is inevitably top-down, unlike in the cited work where it is bottom-up. Nonetheless, we first have to do an initialization step for the generator, the generator itself takes another world generator as one of the input parameters,

[1] This term comes from music where doubling the frequency raises the given pitch by one octave.

e.g., we can use the riverbed generator to create rivers on a terrain created by the fBm function.

First thing we do in the preprocessing stage is the sampling of the procedural heightmap into a a texture of the same size as the final sampled heightmap and normal map textures (see section 2.3 for details) to obtain the rough terrain profile. Next, we apply low pass filtering to obtain the average elevation map, for that purpose we use the Gaussian blur with 20 pixel radius which smooths out even the roughest areas of the underlying terrain.

In the next step we initialize this average elevation map with randomly distributed river springs. From every such spring we track the river flow in the manner of Belhadj and Audibert [2] using a pseudo-physical simulation, i.e., we define river mass (used for determining the river depth and breadth, see below) and the initial velocity. The direction of the velocity vector is determined by the local gradient of the average elevation map — specifically we are interested in the direction of the steepest descent.

In each step of the algorithm we approximate the mass increase of the river particle by the height difference in the steepest descent direction. We use the updated mass value to update the velocity vector using the steepest descent direction. Then, we update the position of the river particle by a vector obtained by normalizing the projection of the velocity onto the surface. This ensures that in each step we move at most by one grid coordinate in the sampled average elevation map.

For every step we also insert the position and mass of the river particle into a look-up table of the same size as the sampled heightmap. Should a collision between the particles appear, we end the tracking of the current river flow and update the mass of the river particle we have collided with by the current particle mass. We also end tracking of the river flow when the particle drifts from the terrain boundaries or if the particle reaches a local extreme of the heightmap.

The last step of the algorithm consists of applying a Gaussian blur onto the river skeleton lookup table. The radius of this blur is user controllable parameter and determines the width of the rivers. This concludes the preprocessing stage. When the generator is finally used for height queries we first query the underlying terrain generator and then subtract a value obtained from the river skeleton look-up table using the bilinear interpolation.

There's an inherent drawback in this method, though. That is its tendency to find local extremes of the heightmap function. See section 4.3 for a full assesment.

2.3 Texture synthesis

With the advent of OpenGL 3.0 the trend is to use GLSL shaders for all rendering. Previously, we have used a procedural terrain texture synthesis in the shaders with a great success. The example GLSL shaders provided with the demonstration application use this technique. Shaders are also responsible for lighting and as we have learned in our previous work [18] it must be calculated per-pixel using a normal map. For if we had used the calculated vertex normals, uncanny lighting effects would ensue.

Namely, the triangle nature of the mesh would stand out and the terrain features would diminish with decreasing level of detail. The latter effect is especially

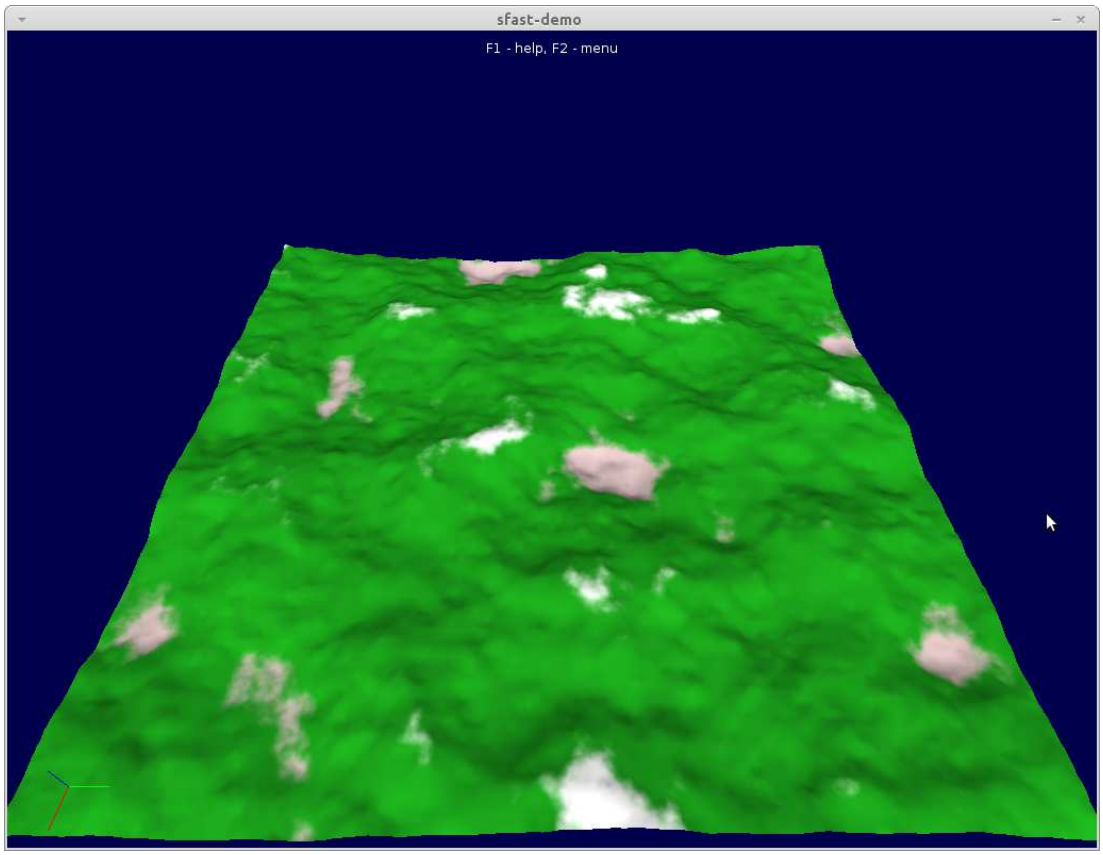


Figure 2.4: A terrain rendered using the per-vertex lighting with vertex normals.

disturbing for a human eye because even-though we are seeing the mountain range from high altitude we can clearly see that terrain isn't flat due to the lighting.

An example of the mentioned problem is illustrated by figures 2.4 and 2.5 respectively. The first picture shows the effect of using the vertex normals for per-vertex lighting. Notice especially the lack of details compared to the second image which is rendered using the per-pixel lighting method with normals stored in the normal map.

The shaders also need to know about the approximate height to apply the terrain texturing accordingly. Hence we need to provide a sampled heightmap texture to the shaders. For both — flat and spherical terrains — the sampling is done by iterating through all texels and encoding the height obtained from the procedural heightmap for the corresponding UV coordinate.

The sampled heightmap texture is then used for creating the normal map. This was done in order to speed up the process as the procedural heightmap queries are time expensive so we want to use already sampled data given the fact that for each texel in the normal mapwe will need 3 x 3 neighbourhood of the sampled heighmapt texel to calculate the normal.

For each texel in the normal map we calculate the normal using the surface gradient formula presented by Mikkelsen [23]. For a given surface S, its normal $\vec{n}$ and a heightmap β the perturbed normal can be expressed as

$$\vec{n}' = \vec{n} - \nabla_S \beta$$

where $\nabla_S \beta$ is the surface gradient. For any point p on S the surface gradient is

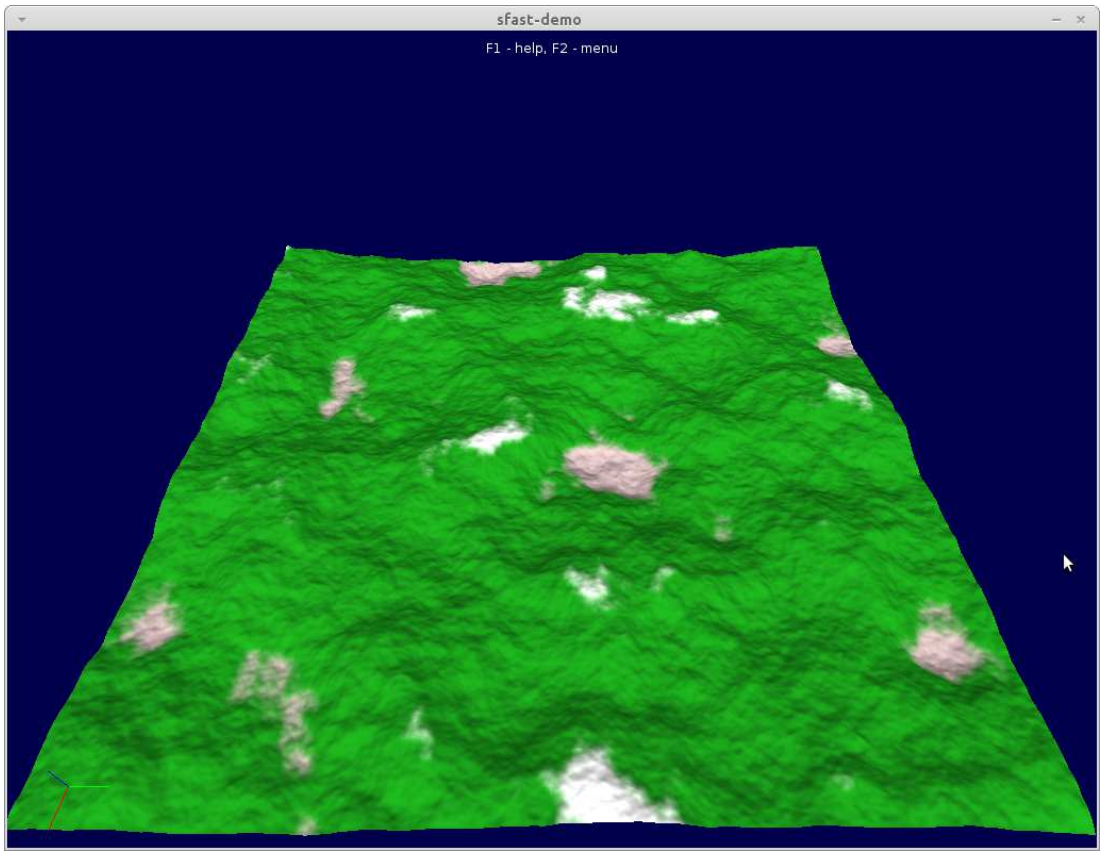


Figure 2.5: A terrain rendered using the per-pixel lighting with normals obtained from the normal map.

equal to

$$\nabla_S \beta = \nabla\beta - \vec{n} \cdot (\vec{n} \bullet \nabla\beta)$$

given that there exists some local extension of β which is a smooth function containing p. Which applies for any reasonable heightmap function we may use. In our case β is the heightmap texture obtained earlier, S is either a flat plane or a sphere depending on the terrain type, and $\vec{n}$ is the normal of the respective surface at sample point p.

To approximate the gradient of the heightmap function $\nabla\beta$ we use the Sobel operator. For every sample point p we obtain the corresponding texel from the heightmap and its 3 x 3 neighbourhood H_p. Border cases are handled depending on the terrain type: for flat terrains we do a mirrored repeat, for spherical terrains we fetch the corresponding texel from a respective neighbouring face in the same manner OpenGL implements seamless cubemap filtering [22, p. 242]. To prevent visible cubemap seams on the normal map — due to the different s and t texture coordinate mapping on the faces that would result in different orientation of the gradients among the faces — we search for the neighbouring texels in the polar coordinates. However, this introduces some visual artefacts around the poles, the equator, and some of the meridians on the spherical surfaces.

By convolving this neighbourhood with the the Sobel operator kernels we get

approximations of its derivatives in x and y axes

$$g_x = \frac{1}{8}\begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * \mathrm{H}_p$$

$$g_y = \frac{1}{8}\begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathrm{H}_p$$

thus we can express tangents at point p as $\vec{t_x} = (\Delta x, 0, g_x)$ and $\vec{t_y} = (0, \Delta y, g_y)$. Where Δx and Δy is the sample difference in respective directions. The gradient $\nabla\beta$ is then equal to

$$\nabla\beta = \vec{t_x} \times \vec{t_y}$$

The heightmap and normal map texture size is a user controllable parameter and even relatively low resolutions (128 x 128 pixels on a terrain of UV size 5.0 x 5.0) are capable of providing sufficient details on the surface for medium to long distances of the camera from the surface. However, the resolution cannot be increased endlessly to obtain more details. This is due to the Nyquist frequency when sampling the procedural heightmap. Yet, even sooner we will notice that the spatial resolution is higher than the height resolution of the sampled heightmap used for normal map sampling. Therefore the height difference between two neighbouring samples will so small that after quantization both will result in the same value and we will get seemingly flat area in the normal map.

2.3.1 Shader based texturing

There is a lot of things one could do for creating the terrain texture. The demonstration shaders bundled with the application do two very simple things that, however, provide quite good results.

At first we apply the height-based texturing. For each fragment we look up its height from the sampled heightmap texture and based on this height we choose the appropriate terrain. For areas near and below the sea level we use a texture of sand; for lowland areas we use a grass texture; we use the same texture even for highland areas, but we tone it down little bit; finally for areas high enough we apply the texture of snow. For smoother look areas between the regions are blended together using the smoothstep function. Next, we use the normal obtained from the supplied normal map to apply slope-based texture. That is for steep enough slopes we blend in the texture of rock.

Because the terrain textures would appear skewed and stretched, especially when applied on spherical terrains or steep slopes, we use the tri-planar texturing technique [26, 27] mentioned in chapter 1. So, for all terrain textures (sand, grass, snow, and rock) we make three texture fetches — one for each planar projection along a basic axis — and blend them together using the weights calculated from the surface normal using the formula described by Ryan Geiss [26].

The water surface is rendered separately in a simple water shader — each fragment under the sea level is discarded during the terrain shader stage — that renders only the fragments under the sea level in blue with the appropriate normal and discards all surface fragments. For that matter we use a specialized vertex

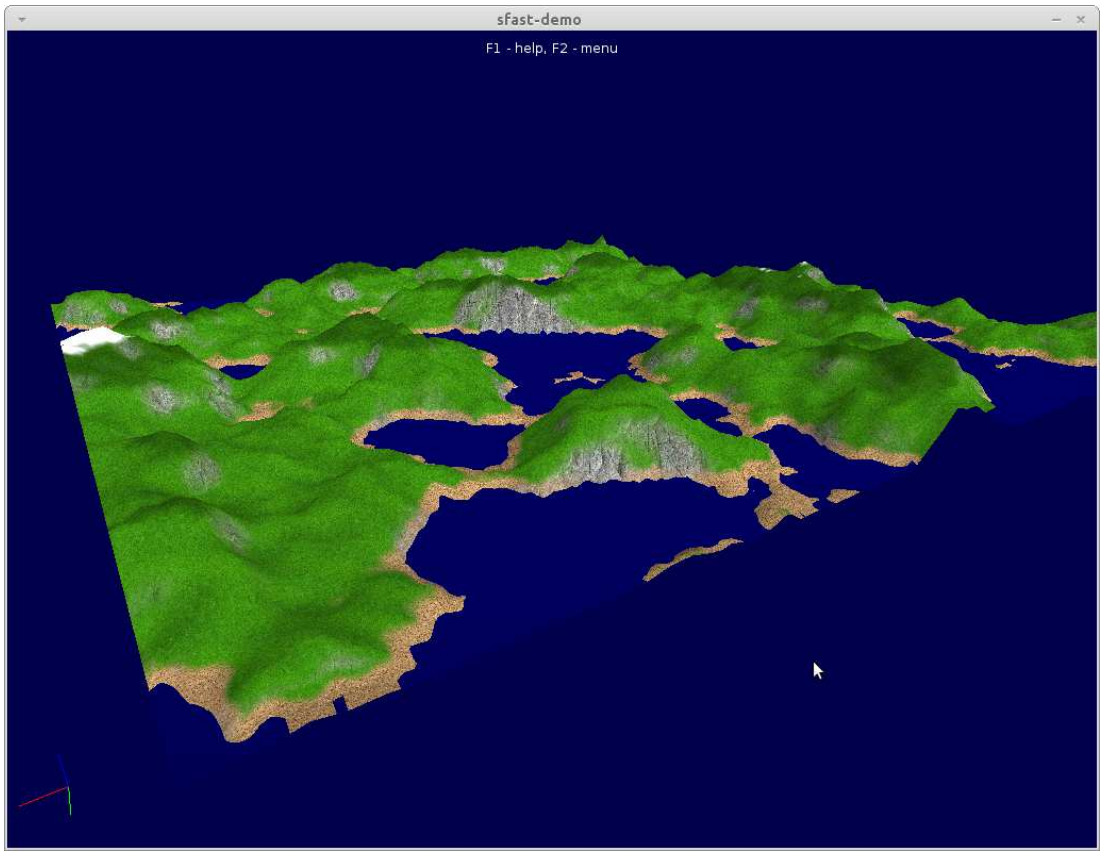


Figure 2.6: GLSL rendered scene of a fractional Brownian motion terrain (10 octaves, lacunarity = 2.0, weight factor = 0.5, random seed = 0).

shader that pushes all the underwater vertices to the sea level. A typical scene rendered using this method can be seen in figure 2.6.

3. Library design & implementation

This chapter is dedicated to a discussion about the design and implementation of our terrain library that uses the theoretical background from the previous chapter. Build instructions as well as the overview of the demonstration application are available in appendix B. We have recycled the name of our planetary generator from the previous work [18], thus, the library bears the name *Slartibartfast*[1].

When designing the library our primary goal was its accessibility for the users and its extensibility. We have used only a minimal amount of other libraries to keep dependencies as low as possible. The main reason was not to deter potential users by a rapid influx of dependent software. A secondary goal was to provide a cross-platform solution. Therefore we are currently dependent on the OpenGL graphics library and on the Allegro library, see section B.1. The Allegro library is used prominently in the demonstration application. Inside the terrain library itself it is used for image and texture handling. Optionally, the OpenMP framework can be used to parallelize the texture sampling. The *Slartibartfast* library was written in C++ using the latest standard C++11.

A typical use case for the library as we see it involves the use of a one terrain type with a single terrain generator. Hence we have designed the library with this use case in mind. In keeping up with modern trends of C++ development we have chosen a compile time polymorphism (class templates) as we are not expecting mixing of the terrain types and terrain generators to justify the use of run-time polymorphism (virtual methods). As a result, the main class that represents the world is a template class with two parameters: *1*) the terrain generator to be used as a procedural heightmap; and *2*) the terrain LOD structure used for rendering.

This makes the library very flexible as the potential user may either use the provided solutions for rendering and generation, or he can supply his own and plug them in given they use the interface expected by the library. The same principle is applied to the classes implementing the actual terrain generators. Each generator takes a basis function as a template parameter, however, we can pass in another generator instead of a basis function. This is actually how the riverbed generator has been implemented.

The library is composed of 5 larger modules, each having its own separate name space. Each module is responsible for a different aspect of the process of terrain synthesis and rendering. There is one module housing the common mathematical functions and other utilities; one module containing rendering support classes; a module dedicated for the ROAM algorithm implementation; one module with all the already implemented terrain generators; and finally one module that binds it all together in one class. Below, we describe said modules more closely.

[1] Acknowledgement to Douglas Adams and his *The Hitchhiker's Guide to Galaxy*.

3.1 World body module

The most important of all modules in the *Slartibartfast* library is the `WorldBody` module. This module contains two classes used for representing the virtual landscape, these are the `World` class that can represent any generic terrain and is normally used for flat terrains; and the `Planet` class that is a specialization of the `World` class and, as the name suggests, is used for spherical terrains.

The `World` class itself is a container class that provides interface for the terrain LOD algorithm and the procedural heightmap. As mentioned, this is realized through the C++ template mechanism as both — the LOD algorithm and procedural heightmap — are passed in as template parameters. In order to obtain an instance of the class a static factory method is provided. This is the preferred method of instancing, however, it can be overridden in derived classes by the user.

The factory method provided in the `World` class creates a flat world patch. Among others it takes two important parameters: an initialized `VertexBuffer` instance (see section 3.4), and an initialized procedural heightmap object (see section 3.2), the terrain LOD structure is created inside the class constructor.

Aside from the LOD related methods, the interface of the class contains a method for updating the LOD structure based on the current camera position and orientation, and a method for rendering the terrain. The rendering is handled using the state pattern, i.e., the `World` class holds an instance of an active `RenderMode` which implements the actual code for rendering the contents of the vertex buffer.

The `RenderMode` class keeps a reference to the `VertexBuffer` instance used by the `World` class. Its render method takes the index of the first triangle to be rendered and the number of triangles to render. This lets the user to use the `VertexBuffer` instance for several terrains or other models (see section 3.4).

3.2 Terrain generators module

The `TerrainGenerators` module contains two mutually connected groups of objects. One of them, as the name suggests, is composed of the actual terrain generators, and the other of the basis functions used by the generators. Each terrain generator is basically a template class that takes the basis function as the template parameter. Each of the implemented generators assumes that the function object is already created and initialized, the valid instance of the basis function is then passed into the generator's constructor.

Generators themselves may or may not implement initialization functions. The only compulsory interface is the `getValue` method with one, two, or three `double` type parameters returning a single `double` value. These values are used for yielding a height value from the generator for 1D, 2D, or 3D coordinate respectively. The one dimensional variant is truly optional, implementation of the other two is up to the user and the type of the terrain he expects — flat terrain patches are usually generated by the two dimensional variant of the query method while spherical terrains use the three dimensional version.

The basis function interface has also only one mandatory method and that is the `getValue` method with the same signature as in the case of terrain generator.

This makes the *Slartibartfast* library very flexible as the basis function itself can be, if needed, used as the procedural heightmap, and in turn a terrain generator can be used as a basis function for another generator. The flexibility also comes from the fact that the world container classes assume that the passed procedural heightmaps are already initialized, and the rest of the library only requires the presence of the `getValue` method.

This effectively means that the user is free to implement and provide any type of the generator, even such that requires lengthy calculations in the initialization stage. The user has also free hands in implementing his own basis functions such as different persistent noise implementations, e.g., a basis function interface wrapper around the libnoise library. The following subsections summarize the available basis functions and terrain generators provided with the library.

3.2.1 Available basis functions

Constant basis

The constant basis function returns a constant value over the whole support. This can be useful for testing and debugging purposes, other than that this basis function has little to no use. The constant returned can be set in the constructor, otherwise it is set to 0.

Image basis

The image basis function was implemented with the debugging purpose in mind[2]. For that matter it doesn't perform any advanced form of interpolation but a nearest neighbour method. It is supplied with an image filename and dimensions of the world to be created. The `getValue` method then finds the corresponding pixel for the input coordinates and returns its value as a number in range of $(0, 1)$ assuming that the input image is a grayscale heightmap. This basis function is intended for flat terrain patches only.

Perlin noise

The Perlin noise basis function is an implementation of a gradient noise method developed by Ken Perlin [30]. We have used the implementation from the *Texturing and modeling* [13, ch. 2] book and an implementation from our previous work [18] as a reference for our own. This basis function is able to yield up to three-dimensional noise values. Upon creation the number of required dimensions is passed in as well as the random number generator seed.

The random number generator used in the initialization method is our own implementation of linear congruential generator implemented in the same manner as the glibc implementation. This is to ensure that the same RNG seed passed into the `PerlinNoise` class will result in the same output over all platforms — an experience gained during our previous work where we have initially depended on the `rand` function provided by the C compiler.

[2] It is not interfaced in the demonstration application as it can fail if the image file is not found or could not be opened.

3.2.2 Available terrain generators

Terrain generator

A simple terrain generator, called just `TerrainGenerator`, only returns the value of the basis function. It has been implemented for reference purposes and for the purposes of not using the basis function as the generator itself, nevertheless, such practice is possible, though not encouraged.

Fractional Brownian motion generator

The fractional Brownian motion generator is a slightly advanced implementation of the fBm method mentioned in section 2.2. This version has several configurable parameters:

Number of octaves — specifies the number of noise octaves to process. Real value can be passed in with the fractional remainder accounted for after the inner loop finishes. Good results are obtained for values between 6 and 10.

Lacunarity — specifies the gap between the successive frequencies, usually set to 2.0. Setting this parameter close to 0 may result in the positive feedback loop and potential divergence of the generator.

Weight factor — specifies the attenuation of the amplitude weight between the successive frequencies, usually set to 0.5. Setting this parameter close to 1 or even larger value will result in a very noisy result unusable for terrain generation.

Other than that the implementation is almost identical to the basic example shown in section 2.2. The only difference is in addition of the fractional remainder of the number of octaves. That is, after summing up the $\lfloor k \rfloor$ (where $k \in \mathbb{R}$ is the number of octaves passed into the generator) octaves in the inner loop we add

$$(k - \lfloor k \rfloor) \cdot w_{\lfloor k \rfloor + 1} \cdot f(n_{\lfloor k \rfloor + 1} \cdot \vec{p})$$

to the result. An example of such terrain can be seen in figure 2.6

Heterogeneous terrain generator

The `HeteroTerrainGenerator` class implements the heterogeneous terrain function by F. K. Musgrave [13, p. 500]. It has been designed to emulate the effects of erosion in lower areas while retaining jaggy mountain ranges in higher areas. It has similar parameters as the fBm generator:

Number of octaves — same as the fBm generator.

Lacunarity — same as the fBm generator.

H — determines the fractal dimension of the roughest areas, in other words the roughness of the mountain ranges. Values around 0 result in very rough and jaggy mountains, while values around 1 provide smoother results.

Offset — the offset raises the whole terrain from the "sea level". If the value is around -1 the resulting terrain is quite flat, values approaching 1, on the other hand, cause very high elevation in the mountain regions. It may happen that the values will exceed the expected limits set in the texture sampling methods inside the demonstration application resulting in seemingly flat areas in the normal map.

Good starting values are: 10 octaves with lacunarity 2, H set to 0.5 and offset at 0. With very rough terrains the normal map may not be able to capture all the needed details. Also, with high offsets it is advisable to scale down the terrain as the generator output values may be so high that the terrain with scale factor of 1.0 would end up outside the viewing frustum.

Hybrid multifractal terrain generator

Another terrain function by F. K. Musgrave we have implemented is the hybrid multifractal [13, p. 502]. The parameters are the same as in the heterogeneous terrain function above. Good starting values are: 10 octaves with lacunarity 2, H = 0.3 and offset = 0.2. Negative offsets will invert the terrain with smooth hills and rough areas near or below the "sea level".

Ridged multifractal terrain generator

Last of the terrain functions by F. K. Musgrave we have chosen to implement is the ridged multifractal [13, p. 504]. On top of the previous parameters, it introduces a gain parameter used to weigh the contribution of the previous octaves in the inner loop. Higher gain results in more accented mountain ridges. We recommend using this generator for spherical terrains as the 2-dimensional variant produces quite surrealistic results. Recommended starting values are: 10 octaves with lacunarity = 2, H set to 0.75, offset set to 0.7 with gain equal to 2.0.

Riverbed generator

The `RiverbedGenerator2D` class contains the implementation of the algorithm pitched out in section 2.2.1. As the name suggests it is currently available for flat world patches, reasons for this are given in section 4.3. This generator requires another terrain function to serve as its basis, hence it is available inside the demonstration application as a layer above the fBm generator and the three terrain functions by F. K. Musgrave.

In addition to the underlying generator parameters it takes the following ones:

Random number seed — used for seeding the RNG (see the Perlin noise basis above).

River count — determines the number of river springs seeded randomly around the terrain.

River depth — defines the initial mass of river particles and the impact the river has on the terrain. In other words the depth of the river.

The generator also needs to know about the terrain dimensions as well as the size of the sampled heightmap texture. The initialization function then uses estimates of the underlying generator output values. A parameter called "river width" is then passed in, it determines the approximate width of generated rivers relative to the texture resolution — this parameter is used as a size of the Gaussian kernel that blurs the river skeleton network, see section 2.2.1 for details. Good starting values are 5 rivers, with width of 5.0 and depth set to 0.1.

3.3 ROAM algorithm module

Figure 3.1 shows the class diagram of the ROAM algorithm module. The diagram contains all of the important classes, though, not all of their methods and attributes are shown. The most important class in this module is the `ROAMworld` class that is essentially a container binding the structures mentioned in section 2.1. The algorithm is realized through the interface of this class, among others it contains methods for setting up split/merge thresholds[3], current LOD (as the number in range $[0, 1]$, maximal tree depth etc.

The `ROAMworld` class keeps an instance of the error metric functor, list of triangles representing the leaves of the ROAM tree, and a list of currently active diamonds — so we do not need to check for them on the fly since we know exactly when they are created or destroyed (see section 2.1). The update method, when invoked, updates both lists using the appropriate update method based on the compile-time settings (see section B.1.2). Either we perform split/merge operations as they appear and stop after the set amount of them, or the priority queues are used. In the latter case, we first calculate split/merge priorities for triangles and diamonds respectively, and then perform the split/merge operations on them starting with the entity with highest respective priority, i.e., error metric.

The behaviour of the `ROAMworld` class as well as the `ROAMtriangle` class can be modified using appropriate policy classes. These define different behaviour for the flat version of the algorithm and for the spherical version. The appropriate configuration is kept inside the `WorldTraits` class that is passed into the classes in this module (or the concrete types within the traits class). For the `ROAMworld` class this is achieved by public inheritance from the respective world policy class — not indicated in the diagram as the inheritance graph could be misleading and indicate multiple inheritance. For the `ROAMtriangle` class the configuration is done via private inheritance from the respective triangle policy class — indicated in figure 3.1 by the meta class `ROAMtrianglePolicy`.

The `ROAMerrorMetricPolicy` functor provides customizable calculations of the error metric for either the split or merge operation. The default metric can be provided by the camera field of view to cull triangles outside the viewing frustum, i.e., force them to be merged. The error metric takes the essential information about the triangle — in case we want to calculate the split priority we pass in the triangle information themselves, otherwise we pass in the information how would the situation looked like if the merge of the diamond was performed — such

[3] The threshold range in the testing application was set empirically as the terrain scaling factor times $5 \cdot 10^{-5}$. This value works well with the default error metric and ideally represents the maximal difference between the hypotenuse midpoint and new vertex position where we can consider the two identical.

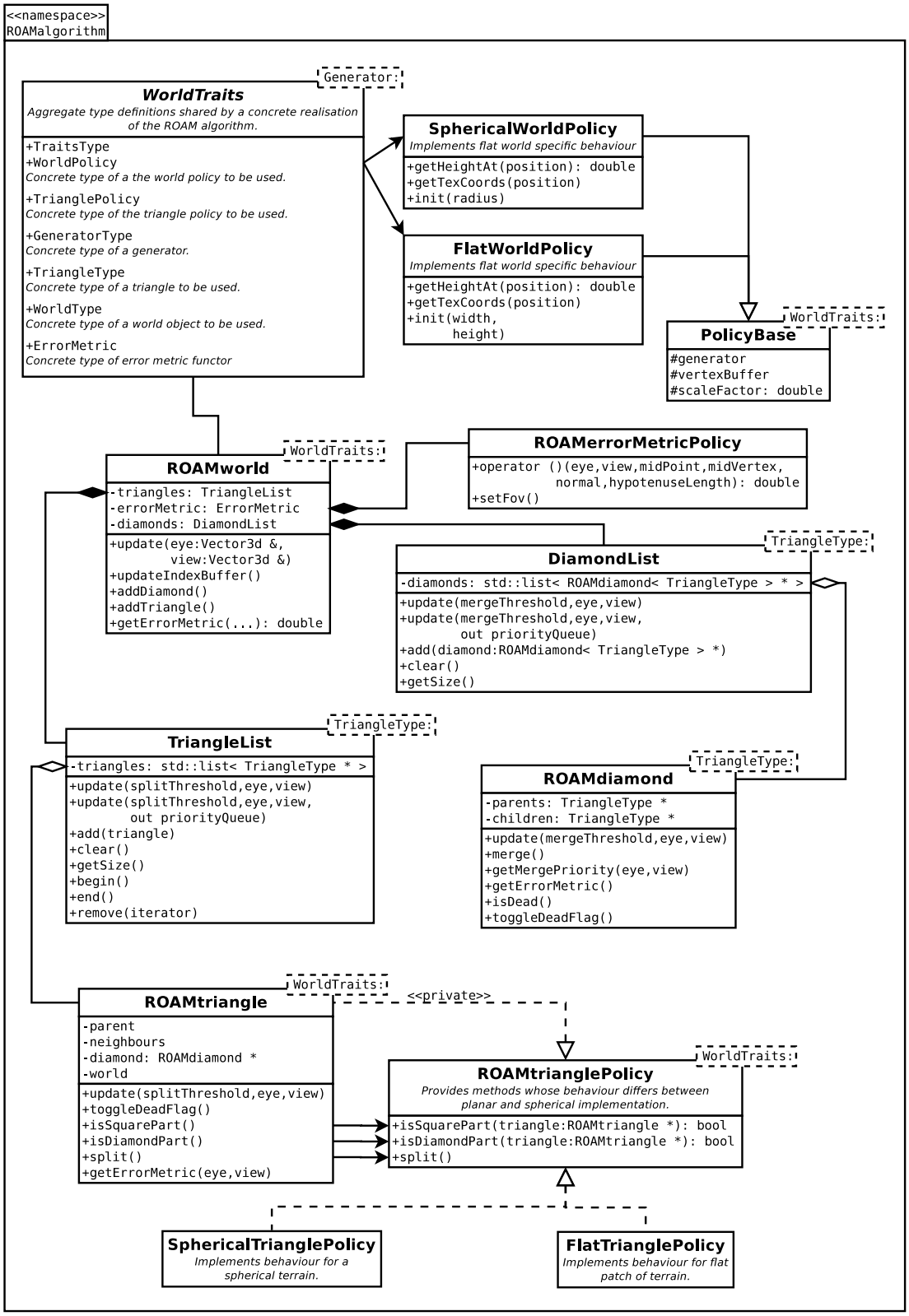


Figure 3.1: Class diagram of the ROAM algorithm module

as: hypotenuse midpoint, position of the vertex after splitting the hypotenuse, length of the hypotenuse and the triangle normal. Apart from these information we have to pass in the current camera position and viewing direction. Our ROAM implementation is configurable by providing user implemented error metric policy

should the provided one doesn't suit the user's needs.

3.4 Common and graphics module

The last two library modules are the `Common` and `GfxUtils` modules respectively. The common module provides us with the implementation of the 3-dimensional vector class, a class used for error logging, and a `Utility` class that contains useful mathematical functions and the implementation of a random number generator.

In figure 3.2 we can see a class diagram of the graphics module. This module contains the implementations of various texture utilities as well as texture samplers. The texture samplers are effectively small functors passed as a template parameter into the texture's sample method. The sample method iterates through all of the texels of the texture invoking the sampler. The implemented samplers are: the heightmap sampler used for sampling the procedural heightmap into a 2D texture or a cubemap; and the normal map sampler that uses the sampled heightmap texture to synthesize a normal map (see section 2.3).

Interface for the camera and helper classes for GLSL shader handling are also provided. Of all the graphics' module classes, the `VertexBuffer` is the most important one. This class serves as the interface between the application and the graphics card. It internally holds all of the vertices created by the ROAM algorithm (the ROAM algorithm itself uses just their indices), and all of the triangles they compose — this list is updated via the `updateIndexBuffer` method in the `ROAMworld` class.

The buffer itself can hold as many vertices as defined upon its creation, though the size is limited by the maximal value of the unsigned integer type. In addition the size of the index buffer is computed as 8 times the number of vertices, thus, the maximum number of vertices the buffer can hold is the maximal value of the unsigned integer type divided by 8 (the maximum number of triangles in the ROAM structure that can share a single vertex). Theoretically, the buffer can hold vertices and triangles of multiple ROAM instances. In order to render correctly a following trick can be used.

As the triangle structure changes from frame to frame, we need to reset and fill the index buffer again each frame. We can store the initial index buffer position and the number of triangles to be rendered for each ROAM structure we want to render. Then, when the actual rendering takes place, i.e., we call the render method of the appropriate render mode (see section 3.1), we pass the starting index buffer position and the number of triangles to the render method.

The inner implementation of the `VertexBuffer` class is heavily modified implementation of the same class from our previous work [18]. Though, we use the name "vertex buffer" the OpenGL vertex arrays [10, p. 80] are used in fact.

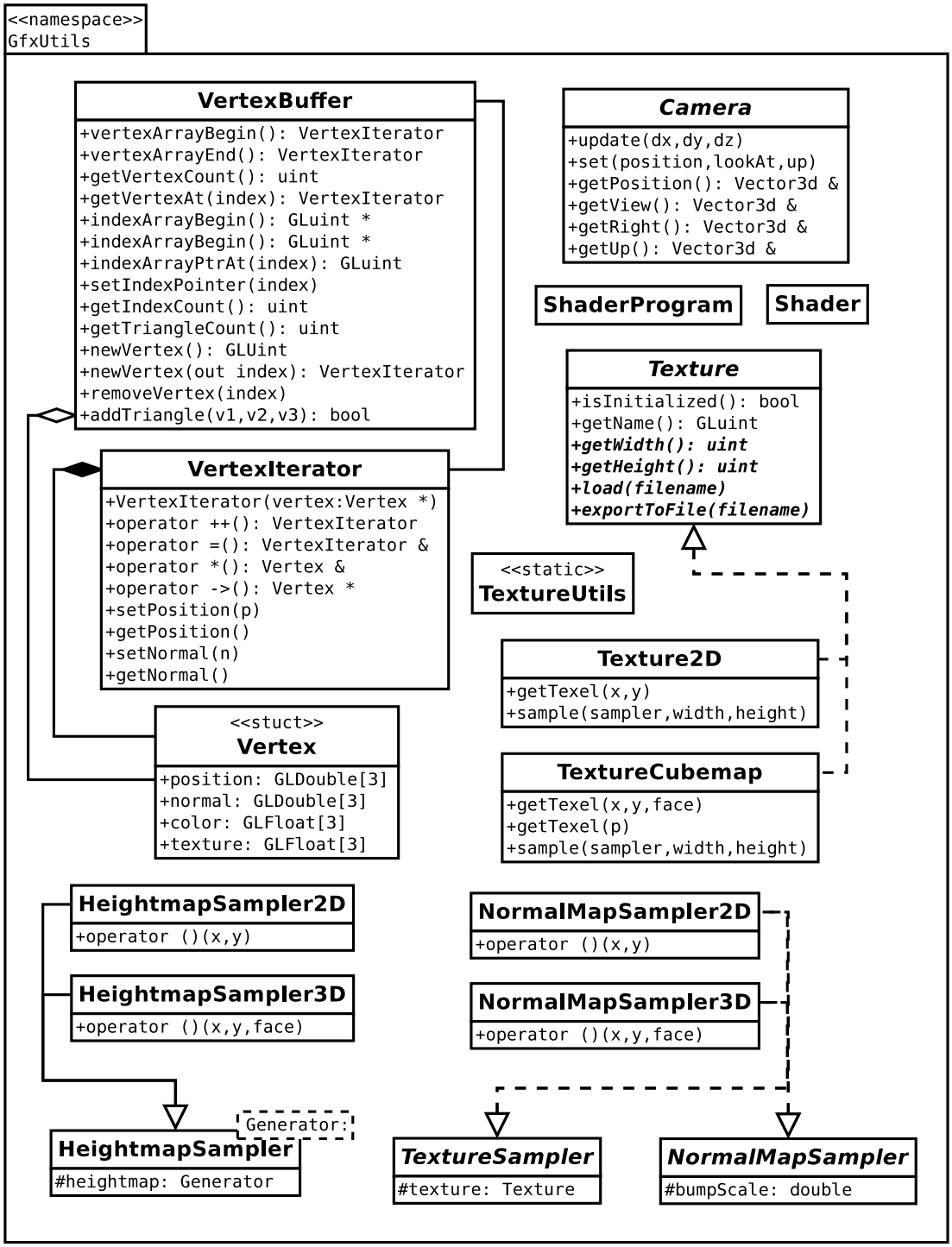


Figure 3.2: Class diagram of the graphics module

4. Application & assessment

This chapter summarizes our work, we discuss the observations made during the development and testing as well as the possible applications of the library and its future. Let us begin with the overall assessment.

We have created flexible and configurable library that can be used for generating and rendering of virtual terrains in real-time. The flexibility comes from the fact that we do not limit potential users into using real-time only techniques as noted in chapter 3. In comparison with our previous work [18] the code of the library is available for immediate use and doesn't require modifications to use it separately as in the case of the application from the cited work. On the other hand, the previous work has generated visually nicer results, however, this time we have focused on the library design. Thus, the provided demonstration application focuses on illustrating the capabilities of the library and provide only basic shaders sufficient for that purpose.

As far as the possible applications for our library are concerned, we can imagine that the library could serve as a basis for a virtual terrain editor similar to the software mentioned in chapter 1. Another possibility is its use in a game, e.g., sandbox space simulation game where we would need lots of randomly generated planets and moons such as *Elite* (1984). Or we could use the flat terrain generator for creating maps for strategy games either at run-time with the possibility of letting the user to change the face of the landscape (as seen in *Populous* (1989) for example), or to create the basic terrain shape and make some adjustments as in the case of *Darwinia* (2005) and *Multiwinia* (2008). A more recent example of the randomly generated terrain would *Planetary Annihilation* still in development.

4.1 Performance

For testing and development purposes we have implemented an experimental application called *sfast-demo* that serves as the library showcase, see appendix A for more information. The software has been tested on the following hardware configurations:

- Laptop: Intel Core 2 Duo T9400 @ 2.53GHz, 4GB RAM, ATI HD3430 512MB shared video RAM
- Desktop: Intel Core i7 Quad 3770 @ 3.40GHz, 16GB RAM, GTX 660Ti 2GB video RAM

In the tests we have used the terrain seen in figure 2.6 and its spherical equivalent, i.e., spherical terrain with fBm generator with the same input parameters. We have used the texture size set to 512 x 512 pixels for most of the tests. The ROAM algorithm was set to maximal tree depth of 15 and to provide maximal available details. For measuring the performance we have used the built-in FPS[1] counter, or the wall clock provided by the operating system used for time-stamping the application log messages.

[1] Frames per second

To obtain a good performance it is safe to assume that 1GB RAM will be more than sufficient as the only limiting factor is the graphics card and specifically its drivers. In case of the desktop hardware configuration the software ran smoothly at approximately 60 frames per second using the official NVIDIA drivers on GNU/Linux. We have tried all of the available rendering modes (see appendix A) and for all of them more than 30k triangles were visible in the scene.

The performance on the laptop was a bit troublesome as the ATI card has been obsoleted in the official driver package from AMD, the software has thus been tested using the open source driver on GNU/Linux. While using the fixed OpenGL rendering pipeline, the software ran at 60 frames per second rendering around 30k triangles. However, when ran with the GLSL shaders the performance dropped to rate of 10 frames per second with strange artefacts appearing all over the surface. At least half of the surface fragments was black for an known reason. We conclude that this may be an error in the OSS graphics driver as the software from our previous work [18] has shown similar results on this configuration while it has worked good previously on the same configuration with the official AMD driver.

Another test was targeted at testing the length of the pre-process stage (texture sampling). This test was performed on the desktop configuration for a flat terrain with 4096 x 4096 texture size [2]. It took the software around 10 seconds to sample the heightmap texture and the normal map texture when run single threaded. When the same code is run in parallel using the OpenMP framework the same operation takes approximately 3 seconds. Which is, in our opinion, within the requirement to run in real-time. The pre-process time for spherical terrains is approximately six times longer due to the fact that 6 cubemap faces are processed.

4.2 Terrain level of detail

This time, we have optimized the ROAM algorithm using the priority queuing and imposing limits on the number of split/merge operations. In comparison with our previous implementation [18] we have reduced, if not removed, the vertex jitter mentioned in section 2.1. By the introduction of the priority queuing and maximal

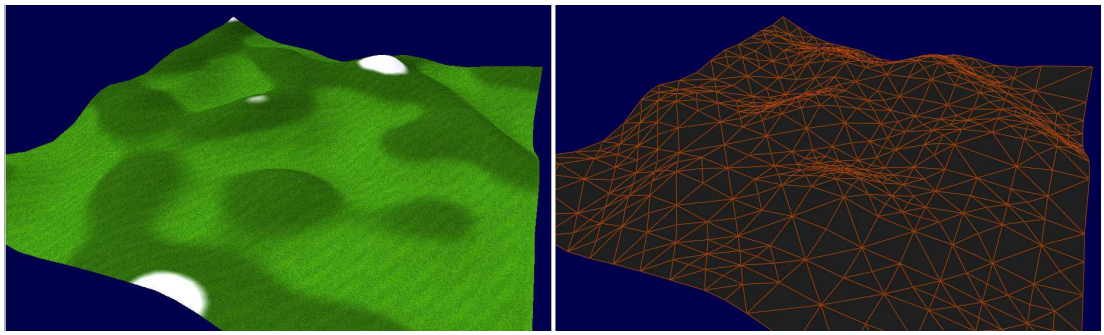


Figure 4.1: The same scene rendered with GLSL shaders (on the left) and with constant coloring and outlined triangle edges (on the right).

[2] Due to the internal representation of the cubemap texture we were unable to create a cubemap texture with a face of this size.

number of split/merge operations per frame we have been able to remove the sudden performance drops that plagued the previous implementation. We have also improved the error metric to take into account whether the triangle is in the viewing frustum and/or lies on the silhouette.

An effect of this can be seen in figure 4.1 on the right, where the hilltops and slopes seen at creasing angles are subject of greater subdivision than the triangles everywhere else. The effect is best illustrated on spherical terrains in the wireframe mode — the horizon is nicely round when compared to our previous work. Thanks to the stable normal map the lighting in 4.1 on the left is stable and doesn't give away the triangular structure of the mesh.

Nonetheless, the two major drawbacks of the ROAM algorithm remain: *a*) the lack of any comprehensive mesh structure that would allow some operations over the terrain mesh such as finding the nearest vertex for a specified position; and *b*) the resulting unpredictable mesh updates that force us to send all the data to the graphics card every frame. We could probably overcome the second problem by using the OpenGL vertex buffer objects, however, we would still need to update rather large portions of the data in a random manner other than some defined chunks. The first problem is much worse for situations where we would want to access the current LOD of the mesh. Yet, we think that the ROAM algorithm is a good overall choice, though, the spread and low price of the modern programmable GPU's make this algorithm outdated as it is not particularly GPU friendly and is not adaptable, to our knowledge, for the GPU-only use.

Depending on the application the user also has to be aware of the relatively low details in close-up zooms at the terrain. One reason for this is the limited resolution of the floating point arithmetic used which limits the spatial resolution of spherical terrains. Another reason is the limited texture resolution. For the actual procedural terrain texture this can be solved to a degree by using either the Wang tiles or the texture LOD as mentioned in section 1.4.1. The texture LOD, though, introduces texture blur [27, 18] which is already present due to the tri-planar texturing. However, the actual sampled heightmap texture and, especially, the normal map texture are limited in its resolution (either because of texture size limit or by the Nyquist frequency). What is more, with some terrain generators the normal map is prone to aliasing.

One way out of this could be the use of tessellation and geometry shaders for adding the actual geometrical details onto the terrain. Another way that could be employed is the use of detail texture maps (see the Cascades demo by NVIDIA [26, ch. 1] for example). In the end we still have to deal with the limited resolution of the sampled textures used by the shaders. To resolve this issue we would have to either move the procedural heightmap evaluation onto the GPU itself, or find a way to send it to the shaders without sampling it first. The work of J. Schneider et al. [34] shows a possible solution for this problem, and M. Mikkelsen [23] has shown that we can calculate the normals in the shader if we have the heightmap.

4.3 River network generation

Now, let us examine the algorithm we have designed for river network generation in section 2.2.1. An example of an exaggerated output of this algorithm is shown

in figure 4.2. We can immediately notice that this doesn't look a lot like river network. This is also the main reason why we have implemented the generator only for the flat terrains and didn't delve into spherical terrain adaptations.

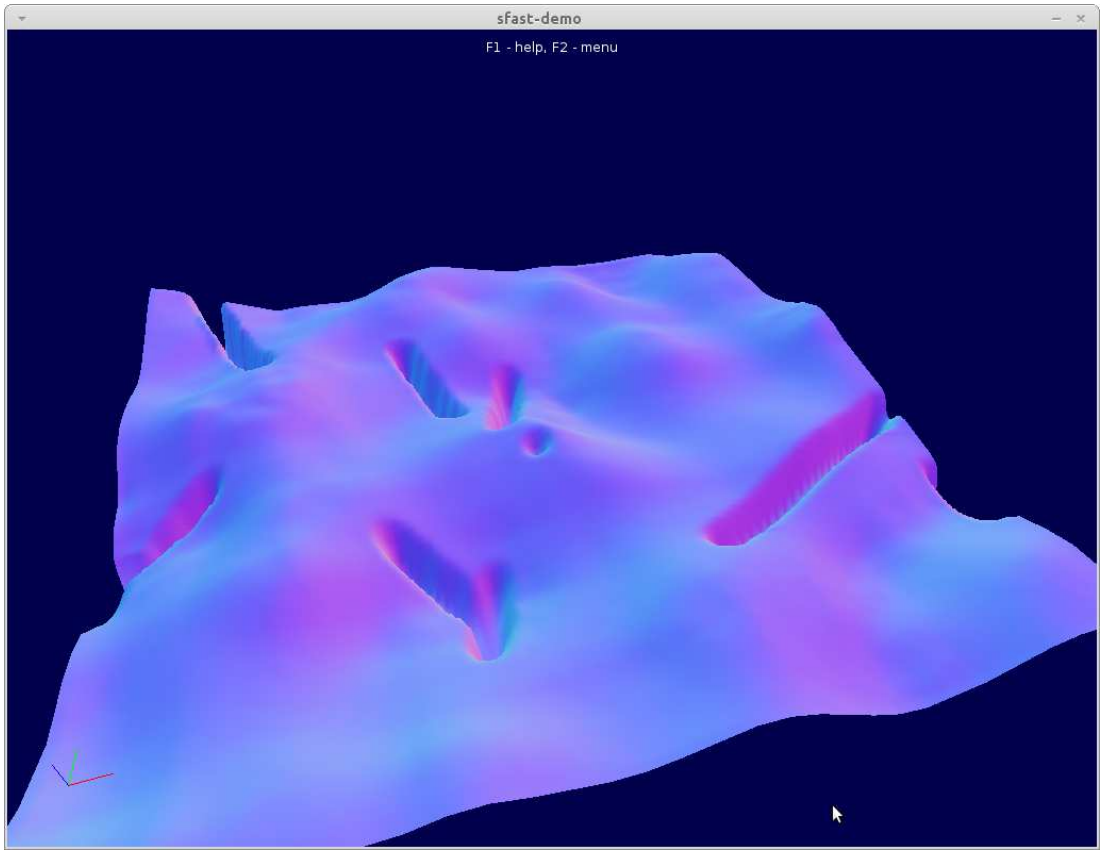


Figure 4.2: An exaggerated output of our riverbed generator with normal map applied.

There are several sources of problems that lead to the observed behaviour. Let us examine them more closely. First and foremost, in in the center of figure 4.2 we see the inherent problem of the chosen method already mentioned in section 2.2.1 — the tendency of the steepest descent method to find local extremes of the function. This is the reason why we see a crater in the middle of the terrain — the gradient at that particular spot is pointing towards the z axis, therefore the river particle will not move from the initial position. This is also the reason why the other rivers amidst the terrain patch doesn't reach the border — this behaviour would not bother us that much as we would see an occurrence of a lake in the nature, though, in reality the water would then flow out of the lake via another stream.

Secondly, we spawn the initial river particles randomly over the terrain. Should we use some processing to prioritize mountain ranges during the selection for the river springs, we could probably obtain somewhat better results.

Another problem is the narrowness of the river stream, we don't see meandering taking place. We assume that this is partly due to the nature of the underlying terrain and due to our implementation where we have wanted to ensure only discrete steps to neighbouring cells in the look-up table. A throughout study of the implementation [2] that served as the major source of inspiration

could give a more insight into this particular problem.

Another problem that occurred to us during the development was the fact that we will not be able to render the water surface in the river in any sane way. For the sea level water surface we use the algorithm mentioned in section 2.3, an uncanny result of this approach is the elevated water near the shoreline, yet this effect is noticeable only in the close-up zoom. Moreover, the water surface rendering depends on the application: for flyovers at high altitudes and velocities this method can be sufficient; for walking around the shore line this will look terrible.

A good solution is to render the water as a separate geometric surface and clip it against the landscape. This, however, requires at least the access to the terrain mesh in a comprehensible way to find the clipping, which cannot be done with the ROAM algorithm and the procedural terrain as mentioned above. Nevertheless, the shoreline would change with decreasing details and a z-buffering could occur for distant views. Rendering of the water surface inside the river would require separate geometry clipped against the riverbed geometry which, however, changes with the viewing position.

All in all, we have to conclude that this algorithm is a dead end. A possible solution, which would however require access to the mesh for clipping, would lie in finding out the approximate river flow path along the terrain. We would need to begin our search in mountainous regions and find out spots in the landscape where the river would most likely flow in nature. For that matter we could adapt the part of the current algorithm responsible for finding out the river path, naturally, after removing its deficiencies as the core idea seems to work [2].

In the next step, we would connect these path nodes to an actual river path. We would represent the river as a separate polygonal structure with its own level of detail based on the subdivision of the underlying landscape. For this step, the access to the terrain mesh would be needed in order to determine the right shape of the river segments. Essentially we would end up with rendering 2D polygonal lines over the surface similarly to the work of Yu et al. [38]. This would be sufficient for a flight simulator for example but most probably not for an open-world RPG game like *Skyrim* (2011).

Conclusion

In this thesis we have presented a good overview of available methods that can be used for procedural creation of virtual landscapes. Moreover, we have created an extendable library using some of the mentioned techniques that can be immediately used to create a fractal terrain at run-time. Due to its flexibility it can be used for implementation of other methods, even methods that require lengthy pre-processing. As such, our software can be used as a solid basis for future work in the procedural terrain field.

We think that, during the development, we have reached the limits of the pure procedural approach for terrain synthesis. We have tried to implement an algorithm that would create river networks at run-time. The algorithm not only requires some degree of pre-processing but also doesn't work as expected. We have discussed its flaws in section 4.3 and we have made some conclusions from the mistakes.

We believe that the fractal based methods used in our library have their use. However, we think that their future lies in their use as a basis for more elaborate methods that would allow some degree of user control over their results such as the works of J. Schneider et al. [34], Zhou et al. [39], and Belhadj [1].

Nonetheless, this thesis can provide a good starting point for anybody who wants to venture into the fascinating world of procedural synthesis of virtual terrains. Either by offering an insight into the studied approaches, giving directions where to look next, or by providing a framework for further experimenting.

Future work

As implied by chapters 1 and 4 and the previous paragraphs, the field of procedural terrain synthesis, and procedural synthesis in general, is an area of active research. We see a lot of potential in it and we would like to continue in further research in this area in a direction indicated above.

We feel that the topic of better user control over the procedural methods is an unexplored space that deserves to be addressed as it is often the most criticized aspect of procedural techniques. We can imagine combining some of the cited approaches [34, 39, 1, 31, 40] into a user-friendly software suite for terrain synthesis and modeling.

We have identified several areas of further improvement of our library: finding out better terrain LOD solution that would remove the main issues of the ROAM algorithm as discussed in section 4.2; improving the heightmap and normal map texture sampling functions for cubemaps as the current implementation is plagued by artefacts caused by different sampling techniques employed in the case of heightmap and in the case of a normal map; and researching a way of implementing the procedural terrain generators on the GPU with access to the resulting mesh, or at least being able to pass the procedural heightmap directly to the shaders thus omitting the texture sampling stage.

Bibliography

[1] Farès Belhadj. Terrain modeling: a constrained fractal model. In *Proceedings of the 5th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*, AFRIGRAPH '07, pages 197–204, New York, NY, USA, 2007. ACM.

[2] Farès Belhadj and Pierre Audibert. Modeling landscapes with ridges and rivers: bottom up approach. In *Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, GRAPHITE '05, pages 447–450, New York, NY, USA, 2005. ACM.

[3] Éric Bruneton and Fabrice Neyret. Precomputed atmospheric scattering. *Comput. Graph. Forum*, 27(4):1079–1086, June 2008. Special Issue: Proceedings of the 19th Eurographics Symposium on Rendering 2008.

[4] Eric Bruneton and Fabrice Neyret. Real-time Realistic Rendering and Lighting of Forests. *Computer Graphics Forum*, 31(2pt1):373–382, May 2012. Special issue: Proceedings of Eurographics 2011.

[5] Eric Bruneton, Fabrice Neyret, and Nicolas Holzschuch. Real-time realistic ocean lighting using seamless transitions from geometry to brdf, 2010.

[6] Iain Cantlay. DirectX 11 Terrain Tessellation. Technical report, NVIDIA Corporation, 2071 San Tomas Expressway, Santa Clara CA 95050, U.S., 2011.

[7] Norishige Chiba, Kazunobu Muraoka, and Kunihiko Fujita. An erosion model based on velocity fields for the visual simulation of mountain scenery. *Journal of Visualization and Computer Animation*, 9(4):185–194, 1998.

[8] Malte Clasen and Hans-Christian Hege. Terrain rendering using spherical clipmaps. In *Proceedings of the Eighth Joint Eurographics / IEEE VGTC conference on Visualization*, EUROVIS'06, pages 91–98, Aire-la-Ville, Switzerland, Switzerland, 2006. Eurographics Association.

[9] Michael F. Cohen, Jonathan Shade, Stefan Hiller, and Oliver Deussen. Wang tiles for image and texture generation. *ACM Trans. Graph.*, 22(3):287–294, July 2003.

[10] Shreiner D., Woo M., Neider J., and Davis T. *OpenGL: Průvodce programátora (authorised transl. of OpenGL programming guide: The official guide to learning OpenGL, version 2, 5th edition.* Computer Press, a.s., Brno, 1st edition edition, 2006.

[11] Oliver Deusen, David S. Ebert, Ron Fedkiw, F. Kenton Musgrave, Przemyslaw Prusinkiewicz, Doug Roble, Jos Stam, and Jerry Tessendorf. The elements of nature: interactive and realistic techniques. In *ACM SIGGRAPH 2004 Course Notes*, SIGGRAPH '04, New York, NY, USA, 2004. ACM.

[12] Mark Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein. Roaming terrain: real-time optimally adapting meshes. In *Proceedings of the 8th conference on Visualization '97*, VIS '97, pages 81–88, Los Alamitos, CA, USA, 1997. IEEE Computer Society Press.

[13] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. *Texturing and Modeling: A Procedural Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2002.

[14] Oskár Elek. Rendering planetary atmospheres in real-time. Bachelor thesis, Charles University in Prague, Faculty of Mathematics and Physics, 2008.

[15] Hugo Elias. Spherical landscapes. `http://freespace.virgin.net/hugo.elias/models/m_landsp.htm`.

[16] Alain Fournier, Don Fussell, and Loren Carpenter. Computer rendering of stochastic models. *Commun. ACM*, 25(6):371–384, June 1982.

[17] Matej Hudak. Physical animation of wetting terrain and erosion. In *Proceedings of CESCG 2011: The 15th Central European Seminar on Computer Graphics (non-peer-reviewed)*, 2011.

[18] Martin Kahoun. Procedural generation and realtime rendering of planetary bodies. Bachelor thesis, Charles University in Prague, Faculty of Mathematics and Physics, 2010.

[19] Peter Krištof, Bedřich Beneš, Jaroslav Křivánek, and Ondřej Šťava. Hydraulic erosion using smoothed particle hydrodynamics. *Computer Graphics Forum (Proceedings of Eurographics 2009)*, 28(2), mar 2009.

[20] Frank Losasso and Hugues Hoppe. Geometry clipmaps: terrain rendering using nested regular grids. *ACM Trans. Graph.*, 23(3):769–776, August 2004.

[21] Benoit B. Mandelbrot. *The Fractal Geometry of Nature*. W. H. Freedman and Co., New York, 1983.

[22] Kurt Akeley Mark Segal. The OpenGL Graphics System: A Specification (Version 4.2 (Core Profile) - August 8, 2011. Technical report, The Khronos Group Inc., 2011.

[23] Morten S. Mikkelsen. Bump mapping unparametrized surfaces on the gpu. *J. Graphics, GPU, & Game Tools*, 15(1):49–61, 2010.

[24] Gavin S P Miller. The definition and rendering of terrain maps. *SIGGRAPH Comput. Graph.*, 20(4):39–48, August 1986.

[25] F. K. Musgrave, C. E. Kolb, and R. S. Mace. The synthesis and rendering of eroded fractal terrains. *SIGGRAPH Comput. Graph.*, 23(3):41–50, July 1989.

[26] Hubert Nguyen. *Gpu gems 3*. Addison-Wesley Professional, first edition, 2007.

[27] Kris Nicholson. GPU based algorithms for terrain texturing, 2008.

[28] Sean O'Neil. A real-time procedural universe, part one - generating planetary bodies. `http://www.gamasutra.com`, 2001.

[29] Sean O'Neil. A real-time procedural universe, part two - rendering planetary bodies. `http://www.gamasutra.com`, 2001.

[30] Ken Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296, July 1985.

[31] Oto Petřík. GPU Supported Terrain Editing. Bachelor thesis, Charles University in Prague, Faculty of Mathematics and Physics, 2011.

[32] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005.

[33] William T. Reeves and Ricki Blau. Approximate and probabilistic algorithms for shading and rendering structured particle systems. *SIGGRAPH Comput. Graph.*, 19(3):313–322, July 1985.

[34] Jens Schneider, Tobias Boldte, and Ruediger Westermann. Real-time editing, synthesis, and rendering of infinite landscapes on GPUs. In *Vision, Modeling and Visualization 2006*, 2006.

[35] Jens Schneider and Rüdiger Westermann. Gpu-friendly high-quality terrain rendering. *Journal of WSCG*, 14(1-3):49–56, 2006.

[36] Christopher C. Tanner, Christopher J. Migdal, and Michael T. Jones. The clipmap: a virtual mipmap. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '98, pages 151–158, New York, NY, USA, 1998. ACM.

[37] Soon Tee Teoh. Riverland: An efficient procedural modeling system for creating realistic-looking terrains. In *Proceedings of the 5th International Symposium on Advances in Visual Computing: Part I*, ISVC '09, pages 468–479, Berlin, Heidelberg, 2009. Springer-Verlag.

[38] Qizhi Yu, Fabrice Neyret, Eric Bruneton, and Nicolas Holzschuch. Scalable real-time animation of rivers, mar 2009. to appear.

[39] Howard Zhou, Jie Sun, Greg Turk, and James M. Rehg. Terrain synthesis from digital elevation models. *IEEE Transactions on Visualization and Computer Graphics*, 13(4):834–848, July/August 2007.

[40] Matěj Zábský. Geogen - scriptable generator of terrain height maps. Bachelor thesis, Charles University in Prague, Faculty of Mathematics and Physics, 2011.

Attachments

A. User documentation

This appendix provides guidelines for working with the *sfast-demo* application. The user can either make use of the precompiled binaries or follow the build instructions in section B.1.

A.1 System requirements

The software can run on wide variety of platforms and hardware, see the testing configurations in 4.1 and operating system support in B.1. The minimal requirements to run the software can be summarized as follows:

- Intel Core 2 Duo 2.53GHz or better
- 2GB RAM
- OpenGL 2.0 (GLSL 1.20) compatible video card (e.g. ATI HD3430) or better
- GNU/Linux 2.6 or newer, Windows 7 or newer (both 32 bit and 64 bit architectures)

For a smooth experience a modern graphics card is recommended (e.g. GTX 660Ti). In any case we recommend using the official drivers provided by the card vendor as they are guaranteed to provide hardware accelerated OpenGL implementation with the GLSL support. The application needs write access to its working directory otherwise the configuration file cannot be used to save user preferences.

A.2 Using the application

The application must be run within the `demo/bin` directory as its working directory. By default the application starts in the windowed mode with the 1024x768 resolution. Upon the first run a configuration file *Slartibartfast.ini* is created which can be used to modify some of the application settings, see section A.3 for overview of the available options. The user is presented with clean viewport and visible GUI that can be used for creating a terrain as seen in figure A.1.

A.2.1 List of controls

Short version of this list is available inside the application by pressing the F1 key. The GUI can be shown or hidden by pressing the F2 key. It is advised to do so while inspecting the world (see section A.4). Camera movement is controlled by a mouse and a keyboard:

- Right mouse button — by pressing it and moving the mouse the viewport is tilted around.

- W, S, A, D keys — move the camera forward-backward, left-right, respectively. Depending on the camera mode either the eye position or the focal point is moved.
- Spacebar and left shift — move the camera upwards or downwards. Again, either the eye or the focal point are moved depending on the mode.
- C — cycles between the camera modes.
- F — refocuses the camera to its original position with the focal point lying in the point of origin.

The two mentioned camera modes are:

Free roam — this is the default camera mode where the user affects directly the eye position by keyboard and looks around via the mouse movement. This type of camera is prevalent in many first person games.

Rotating — this is the alternative camera mode where the eye position revolves around the focal point in all directions with retaining the same distance. The focal point can be moved around using the keyboard and the viewing position on the virtual sphere can be changed by the mouse movement. This camera mode is especially useful with the spherical terrains.

Once the terrain is created there are several rendering modes available. They can be switched using the following keys:

- Key 1 — switches on the vertex rendering mode where only the active mesh vertices are displayed.
- Key 2 — switches on the wireframe rendering mode.
- Key 3 — switches on the opaque outline rendering mode. The terrain is rendered with constant color and wireframe overlay.
- Key 4 — switches on the heightmap textured mode.
- Key 5 — switches on the normal map texture mode.
- Key 6 — switches on the GLSL shader mode.

The application can be closed immediately by pressing the Escape key or by clicking on the close button in the window title (depends on the operating system and window manager).

A.2.2 GUI

Figure A.1 shows a typical session with the *sfast-demo* application. In the middle of the viewport the current terrain is shown with a bounding box. In the bottom-left corner of the viewport we can see visual aid showing us the orientation of primary axes with respect to the camera — the x axis in the red, the y axis in the green and the z axis in the blue.

On the right the main application menu is shown. From here the user can adjust some parameters of the current terrain and prepare parameters for the new terrain he wishes to create. The offered controls are from the top to the bottom:

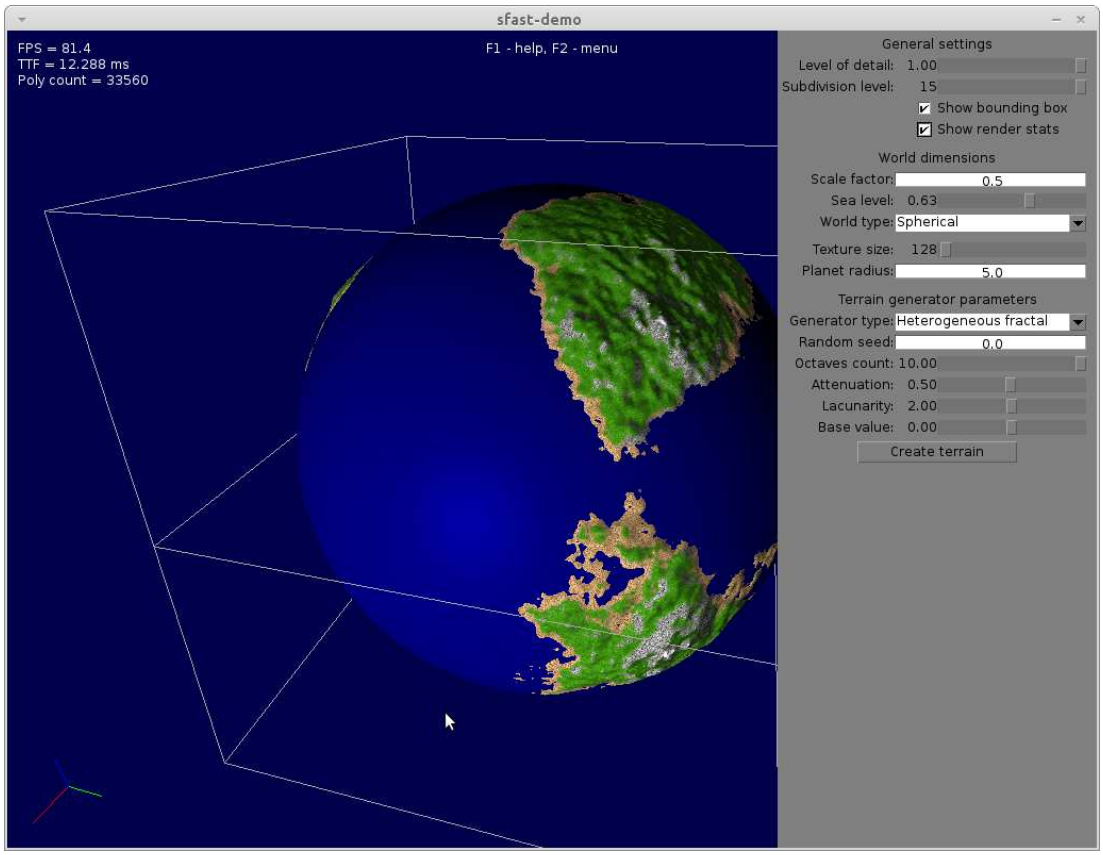


Figure A.1: Graphical user interface

Level of detail — this slider controls the amount of displayed details at the current level of mesh subdivision. The value of 0 means aggressive LOD with less details displayed while the value of 1 means the most available details. This option is interactive.

Subdivision level — this slider controls the subdivision level of the mesh, i.e., maximum depth of the ROAM tree. When the slider is set to 0 the mesh subdivision is off and we see only the basic shape. This option is interactive.

Show bounding box — this check-box switches on/off the bounding box rendering (the white box seen around the terrain). It usually shows us the maximal terrain dimensions as estimated for the respective terrain generation method. For flat terrains the current sea level is also shown as the middle plane. It shows the planet equator for spherical terrains.

Show render stats — this check-box shows/hides the rendering information displayed in the top-left corner. Displayed statistics are: the average frames per second rendered; the average time to frame in milliseconds; and the number of currently rendered triangles.

Scale factor — this field allows the user to enter a scaling factor applied on the output of the procedural heightmap for the next terrain.

Sea level — this slider lets the user to change the sea level for the current and next terrain in a interactive manner. This is the only world and generator

option that is interactive. The results can be seen in shader mode only.

World type — this drop-down box lets the user to select the terrain type, either flat or spherical terrains are currently supported.

Texture size — this slider allows the user to set up the texture size. For flat terrains the texture width and height can be adjusted separately.

Planet radius — this field allows the user to specify the radius of the spherical world in OpenGL coordinate units. For flat terrains the patch width and height can be set instead.

Generator type — this drop-down box lets the user specify the terrain generator function to be used. The lists differ for the world type selected as well as the controls shown below the drop-down box. For the full list of controls see the list below or refer to the respective generator in section 3.2.

The slider values can be changed either by dragging the mark with a mouse or by selecting the slider (either by a mouse click or Tab key) and using the mouse wheel for precision set-up. The text field values can be changed by clicking on them, deleting their contents and inputing a new value. The value must be confirmed by pressing the Enter key, a validation is then run against the entered value to sanitize the input, e.g., to ensure that a new scale factor value entered is a positive one.

Terrain generator settings

Base value — for the constant basis generator this is its output values, for all others (heterogeneous, hybrid, and ridged) it is the offset value. See section 3.2 for details.

Random seed — the number used to seed the random number generator.

Octaves count — the number of fBm octaves.

Attenuation — for the fBm generator it is the weight factor used for weighing the amplitudes of successive frequencies. For all others (heterogeneous, hybrid, and ridged) it is mapped to the H value. See section 3.2 for more details.

Lacunarity — defines the gap between the successive fBm frequencies.

Gain — defines the gain for the ridged multifractal 3.2, i.e., the contribution of the previous frequency in weighing the current one.

River count — defines the number of seeded rivers for the riverbed generator.

River depth — defines the depth (or impact on the terrain) of the riverbeds. The larger the value the deeper the river.

River width — defines the width of the river, also affects the depth. The larger the value the wider and shallower the river.

A.3 Configuration file options

As mentioned above, the *sfast-demo* application uses the *Slartibartfast.ini* configuration file for storing the user preferences and last used values in the GUI. We won't go over all of the options in this section as most of them are not intended to be edited by hand. The default configuration is provided with explanatory comments which should be sufficient for understanding. Thus, we will cover the most important options:

Screen width — defines the width of the viewport in pixels.

Screen height — defines the height of the viewport in pixels.

Field of view — defines the field of view of the viewing frustum. Values are in degrees in range $[0, 180]$, see the OpenGL documentation [10] for more information.

Near clip plane — defines the near clipping plane of the viewing frustum.

Far clip plane — defines the far clipping plane of the viewing frustum.

Vertex buffer size — defines the size of the vertex buffer. Values greater than the largest unsigned integer divided by 8 will most likely result in an undefined behaviour.

Mouse sensitivity — value in range $[0.1, 10.0]$ that defines the sensitivity of the mouse when used for camera movement.

Camera speed — value in range $[0.1, 10.0]$ that defines the speed of the camera when moved around by keyboard.

The rest of the entries can be changed within the application GUI and holds the last used values.

A.4 Known issues

This is an experimental application that is unfortunately far from being polished. Thus, there are several issues in the UI known to us but due to the time constraints and their relative low priority weren't addressed. First and foremost the application was written as single threaded, i.e., all calculations and rendering are done in the same thread. In consequence the application may appear as frozen when creating a new terrain, especially when large terrain texture size is selected and/or the application is run in the debug mode.

When in the shader mode the geometry may appear white or black, this is caused by the shaders not being compiled or linked properly. The reason for this will be most probably stated in the error log file *Slartibartfast.log*. Possible reasons include: *a*) syntax error in the shader code if the user has tried to edit it; *b*) lack of GLSL 1.20 support on the graphics card (see section A.1); *c*) or an error in the graphics card driver.

Finally, there are two known issues with the GUI:

- When the fullscreen mode is switched on (by changing the code or by pressing Alt + Enter) the GUI won't appear. This is most probably a bug in the alguichan library.
- When the GUI is visible the camera controls collide with the GUI, e.g., changing the sliders with the mouse wheel may change the zoom of the camera, pressing the spacebar can trigger currently selected GUI widget. Therefore we advise to hide the GUI when viewing the terrain.

Please, keep in mind that a very large terrain may not be fully visible due to the near and far clipping planes of the camera. The camera may also appear inside the terrain (affects mainly spherical terrains) either upon the terrain creation or after the camera repositioning using the F key.

B. Programmer's guide

This attachment provides build instructions for the library and the demonstration application. The application itself is composed of several classes and namespaces plus the collection of demonstration GLSL shaders (`.fs` for fragment shaders and `.vs` for vertex shaders). The `Core` namespace contains code for setting up the graphical window and handling the user interface. The `Terrain` namespace then provides wrapper classes around the library terrain class[1] The source code of both the library and the demo application is well documented using doxygen.

B.1 Build instructions

The *Slartibartfast* library and the *sfast-demo* application (further referred as software) were written in C++. The software depends on the OpenGL 2.0 library and makes use of the Allegro 5.0.5 library. The alguichan library — providing the GUI functionality — is distributed as part of the *sfast-demo* project. The software is distributed under the zlib licence bundled with the source code.

The source code should be portable among different platforms that provide the C++11 compliant compiler, OpenGL 2.0 implementation and a port of the Allegro 5.0.5 library. These include Windows, GNU/Linux or MacOS. The software was written and tested on the GNU/Linux operating system. It is distributed as compressed archive with all the source files, *Code::Blocks* project files and GNU/Make makefiles.

B.1.1 Dependencies

As noted above, the software has several dependencies, namely:

- OpenGL 2.0
- Allegro 5.0.5
- alguichan

The OpenGL implementation is video card vendor specific, we provide the MESA library which is the open source OpenGL implementation. However, it is limited and provides software-only renderer and doesn't support GLSL shading language. Building the software against this library is not advised and will result in limited functionality of the *sfast-demo* application.

Sources for the Allegro 5.0.5 library are provided[2]. Should the user decide to build the Allegro library from the sources there may be further dependencies, but these are well covered by the library documentation. The basic library must be built with at least the image add-on, font add-on, TTF add-on and primitives add-on.

[1] The library design prefers the use of a single terrain type with a single terrain generator as noted in chapter 2.

[2] For the help, or possible binary distribution, we refer to `https://www.allegro.cc/`

The alguichan library is built as part of the *sfast-demo* application so there's no need to build it separately. Further, the user might want to enable the OpenMP support but it is not required for a successful build.

B.1.2 Build options

These options are available for the user at compile time as macro defines:

- `NDEBUG` — disables debugging functionality (assertions, etc.).
- `ROAM_USE_PRIORITY_QUEUE` — tells the ROAM algorithm to use priority queues for split and merge operations.

Please note that the ROAM algorithm is built as a part of the final application not the actual library.

B.1.3 Building on GNU/Linux

The best way to build the software on Linux is to use the provided GNU/Make makefiles. These were created from the *Code::Blocks* project files and use the directory structure found in the supplied zip archive. Another way is to open the supplied project files in *Code::Blocks* and build the software there.

First the *Slartibartfast* library needs to be built and then the *sfast-demo* inside the `demo` directory. The respective makefiles and project files are located there.

B.1.4 Building on Windows

The best way to build the software on Windows is to obtain the *Code::Blocks* IDE with the mingw tool-chain and build the software from here. However, the linker settings must be adjusted according to the dependency naming conventions, e.g., `-lopengl32`. Otherwise, all of the sources found in the `src` directory must be compiled and linked into a static library. In order to build the demo application all of the source files the `demo/src` and `demo/lib` must be compiled and linked into the executable. The application will need to link the *Slartibartfast* library and will need to know the location of the library header file — it is located within the `src` directory.

B.1.5 Building on other platforms

We haven't tested the software nor the build process on other platforms. However, it should not be hard to adapt the provided GNU/Make makefile for other platforms.

C. Contents of the attached CD

The accompanying CD contains the following files:

- `GL` — a folder with 3 OpenGL header files.
- `allegro-5.0.5.tar.gz` — an archive containing the Allegro library needed to build the software.
- `slartibartfast.zip` — an archive containing the source files for the software, i.e., the *Slartibartfast* library and the *sfast-demo* application and all the files needed to build and run the software.
- `kahoun_martin_master_thesis.pdf` — an electronic version of this thesis.