STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LVII**, Number 2, 2012

OPTIMIZATIONS IN PERLIN NOISE-GENERATED PROCEDURAL TERRAIN

ALEXANDRU MARINESCU

Abstract. The following article wishes to be the first of a series focused on different aspects involving procedural generation, with its ultimate goal being that of building an entire realistic 3D world from a single number (known in literature as the "seed"). It should allow for free exploration and render at interactive frame rates on mid to high-end graphics hardware. To begin with, we will discuss the manner in which we have generated procedural landscapes and some techniques we have borrowed from rendering algorithms in order to optimize the terrain generation and drawing process. Regarding the rendering framework, we have gone for Microsoft XNA Game Studio, the key factors of our choice being its simplicity plus the fact that we can focus more on the structure and realization of the algorithms and less on implementation and API details. As in our previous work [10], we have considered that this outweighs its limitations and that the concepts presented here should very well fit any programming language/drawing API.

1. About Procedural Generation

We feel it is our duty to first familiarize the reader with the field of procedural generation which, surprisingly, pre-dates the era of computers. The idea behind procedural generation is quite simple: what if we could take a complex object (and in our context, by "object" we usually refer to textures, vertices, polygons and meshes) and approximate it in a coarser or finer manner (depending on needs and hardware) by a function or procedure. In other words, when discussing procedural generation we must investigate to what extent it is possible to parametrize an object [16, 15].

Received by the editors: May 30, 2012.

2010 *Mathematics Subject Classification.* 68U05.

1998 *CR Categories and Descriptors.* A.1 General Literature [**INTRODUCTORY AND SURVEY**]; I.3.7 Computing Methodologies [**COMPUTER GRAPHICS**]: Three-Dimensional Graphics and Realism – *Virtual reality* .

Key words and phrases. procedural generation, seed, perlin noise, terrain, heightmap, LOD, chunk.

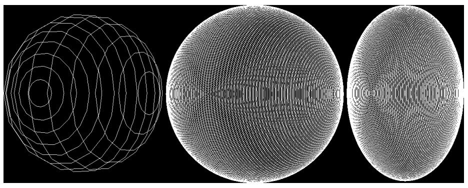

Figure 1. Different parametrizations for a procedural ellipsoid.

To better understand this process of parameter refinement, one should visualise a simple sphere. It is described by the coordinates of its centre in 3D world space and its radius, with all the points which create the "shell" of the sphere, (x, y, z) following the equation $(x - x_C)^2 + (y - y_C)^2 + (z - z_C)^2 = R^2$.

This is what we call the Sphere(Centre, Radius) parametrization. However simple it may seem, for the sphere to be drawn on the screen, the graphics pipeline must be fed primitives (triangles). This means that we must take discrete points (vertices) on the shell of the sphere, process which can return less or more vertices, depending on the division step by which we discretized the sphere. So now, apart from the centre and radius we have an extra parameter – a step increment which, for larger and smaller values yields, respectively, coarser and finer approximations of a sphere – the Sphere(Centre, Radius, StepDivision) parametrization. Additionally, we can substitute different increment values for the XY circle planes and for the Z axis, allowing for more control over the final shape. We can associate an elongation factor to a certain direction, effectively turning the sphere into an more general ellipsoid – Sphere(Centre, Radius, StepDivision, ElongationFactor, ElongationDirection). It is vital to notice that each new iteration is a generalization of the previous, so we can still generate everything we could before, while at the same time being able to provide new classes of objects (Figure 1).

At this point, the reader may already speculate that we can add an arbitrary number of parameters to our objects, each of them tweaking a different aspect of the final outcome, but still one should keep in mind some important aspects. The more parameters, the more control we have over the resulting object, but a more complex parametrization usually comes at the expense of speed and storage, since the resulting object needs to reside somewhere in memory.

The philosophy behind procedural generation states that, for the same combination of parameters we must have the exact same end result, making it possibly one of the most efficient methods of data compression to date. This turns it into an invaluable tool for game developers building extreme large scale environments, where the use of art assets sculpted by 3D modellers is virtually impossible. Then again, this is the main reason why the authors have chosen it for the task of building a large scale realistic environment. We list here some sources of inspiration for our project: [1] – a free-roam racing game set in a post-apocalyptic environment with the total land surface estimated to an equivalent $14400km^2$, [3] – a galactic-scale god game with procedurally generated animations for creatures the player can model at his own will, [4] – a 96 kb FPS with all assets procedurally generated, from sound effects to weapon models, and last but not least, the many demo groups that push the limits of what can be done with a tiny amount of code – [2, 24]. The following sections will present an overview of the perlin noise generation algorithm and our attempt at implementing a synchronized CPU/GPU perlin noise generator (or PNG) focusing afterwards on extracting terrain heightmap data from the PNG.

2. Perlin Noise

Perlin noise is a type of gradient noise first developed by Ken Perlin in 1985 [17]. It relies on interpolating between the values of a lattice of random gradients to produce hyper-textures of pseudo-random appearance. The interpolant is usually a function having first and preferably second derivatives at endpoints 0. In our case, we used the improved interpolant $6t^5 - 15t^4 + 10t^3$ suggested by Ken Perlin in 2002 [18]. For further reading regarding the algorithm itself, please refer to [19, 5, 22, 14, 6].

The essential fact for us is that perlin noise is an application from $\mathbb{R}^n$ to $\mathbb{R}$, that is, for each point in an n-dimensional hyperspace, $PNG(x_1, x_2, ..., x_n)$ returns the evaluation of the perlin noise primitive in that point. Moreover, the PNG is consistent, meaning that, if $(x_1, x_2, ..., x_n) = (y_1, y_2, ..., y_n)$ then $PNG(x_1, x_2, ..., x_n) = PNG(y_1, y_2, ..., y_n)$, or the value returned by the perlin noise generator for the same point in space is always the same. This makes it very well suited to the context of procedural generation.

Because the PNG will be used extensively both by the CPU and the graphics pipeline, we have attempted to create a hybrid PNG that produces "similar" results, for up to 4-dimensional coordinate space. We say similar, because from a numerical point of view, the values evaluated in the same points by the CPU and GPU will never coincide, owing to different architectural implementations [6]. Moreover, the results are not even the same between different GPUs, but as we've stated before, we are interested in the visual appearance of the generated hyper-texture. We have not gone beyond 4D space because there was no interest for it in our application. Below one can find an outline of the hybrid CPU/GPU PNG:

- Initialize a random number generator with a given seed;

- Construct a 1D 256 permutation table containing a random permutation of the numbers 0 through 255;
- Construct a 2D 256*256 permutation table having 4 fields per element containing all the possible combinations of 2 indexes taken 0 through 255 and then hashed using the previous permutation table (all operations being performed modulo 256):

$$\begin{aligned}
&\forall i = \overline{0,255}, \forall j = \overline{0,255}\\
&A = Permutation[i] + j, B = Permutation[i+1] + j\\
&Permutation2D[i,j].Field_1 = Permutation[A]\\
&Permutation2D[i,j].Field_2 = Permutation[A+1]\\
&Permutation2D[i,j].Field_3 = Permutation[B]\\
&Permutation2D[i,j].Field_4 = Permutation[B+1]
\end{aligned} \tag{1}$$

- Construct hashed permutation tables for the static 1D, 2D, 3D and 4D gradients. At this moment we have all we need for the CPU side.
- Generate (only once per application since these do not change) 1D textures coding the gradients using XNA's NormalizedByte4 surface format;
- Generate (every time the PNG is re-seeded) the 1D and 2D textures containing the permutation tables using XNA's Color surface format;
- Generate (again, with each re-seed) the 4 textures corresponding to the hashed permutation tables using XNA's NormalizedByte4 surface format;
- Finally, feed these textures to the graphics pipeline each time a shader utilizes procedural noise primitives.

The full documentation for the above mentioned surface formats and their use cases can be found at the following resources: [12, 13, 11, 9].

The higher dimensionality of gradient noise is interpretable, for example, 2D noise yields a texture which can be applied on a model with UV texture coordinates or used as a heightmap to generate 3D terrain (no folding of terrain surface such as caves or archways). 3D noise can be used to texture a 3D model without UV texture coordinates, based solely on the coordinates of its vertices, it can generate fully fledged 3D terrain, or it can be used to create animated textures, such as fire (with time being the extra 3-rd dimension). One can already realize why we have considered 4D noise sufficient for our purposes. We have also brought our personal contribution to this area, by fixing a minor bug in the HLSL implementation of 4D perlin noise from Nvidia GPU Gems 3 [14] and finding an optimization for 4D noise which uses fewer texture addresses and is consequently faster.

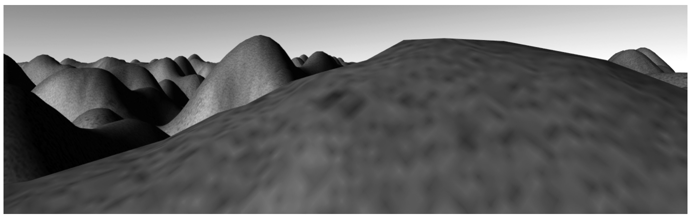

FIGURE 2. The lack of precision of a classic heightmap is responsible for the rough aspect of the terrain.

3. 3D Terrain Generation

The first thing that usually comes to mind when building a 3D world is the terrain. It is the basic support for other systems such as vegetation, water, buildings and so on. There are a number of ways in which we can import the terrain into our virtual reality application, heightmap-based generation being by far the simplest [7, 8]. Terrain can also be modelled as a 3D mesh with the use of 3rd party software, but this is out of the question for large scale environments.

A heightmap is normally a 2D grayscale image for which each pixel stores the height of the corresponding vertex (usually in the R channel). For example, a 64*64 image will yield 4096 vertices, with the width and height dimensions becoming X and Z coordinates respectively, possibly multiplied with some scale factors afterwards. The greatest drawback of this approach is that the color channels of the heightmap are very limited with respect to the range of values they can store (256 discrete values per channel). Visually, this lack of precision manifests itself as the "jagged" aspect of the terrain, which is noticeable even for small differences in height for adjacent vertices (Figure 2).

In order to fix this issue, we have used the PNG which we have discussed in the previous chapter to generate very smooth terrain. From an abstract point of view, we have considered our world, albeit it being in 3D, as "sitting" on a 2D lattice that extends infinitely on the X an Z axes in the XZ plane. All lattice points have integer coordinates, and the square delimited by 4 points will be denoted in our terminology as a "chunk". As a consequence, we have divided our "world" into chunks, each chunk being identified solely by the integer coordinates of its upper left corner (Figure 3). This idea was inspired by the manner in which [21] handles its large scale environment.

Afterwards we generate the actual terrain by splitting the intervals $(UpperLeft_X, UpperLeft_X + 1)$ and $(UpperLeft_Y, UpperLeft_Y + 1)$ into a desired number of subintervals. In our actual implementation we divided both intervals to obtain a smaller lattice of 64 by 64 vertices whose height can now

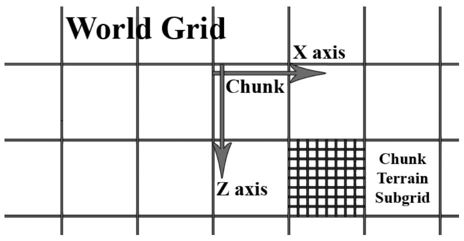

FIGURE 3. The lattice over which the 3D world resides.

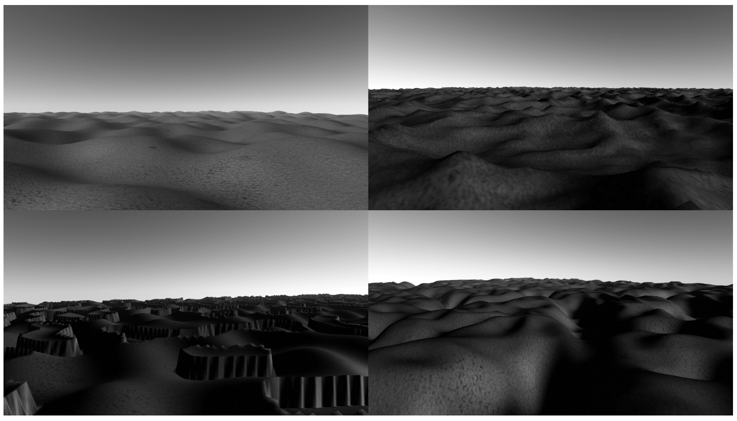

FIGURE 4. Examples of procedural terrain obtained using various combinations of perlin noise primitives. For each vertex, the normal component was computed based on the approach from [7, 8] and the tangent/binormal frame was calculated using the algorithm from [25] in order to apply more advanced lighting.

be determined by simply querying the value of the PNG at each given vertex (we already know the X and Z coordinates). Remark that, because of the properties of perlin noise and because we have split the 3D world in such a manner, the resulting terrain will be seamless. One should experiment with different combinations of noise primitives, visualize the outcome and try to understand the behaviour of various compositions of noise functions (Figure 4).

4. Optimizations

At this point there is still a lot of room for improvement and this section will present some of the steps we took in order to optimize the application, some of them targeting the algorithmic side whilst others address efficiency of the rendering pipeline.

4.1. **Architectural Robustness.** We have already mentioned the chunking mechanism for generating our procedural world: each chunk is an abstract container, not only for terrain, but for everything that lies in that specific sector of our virtual world. In order to keep this high level of decoupling between chunks, all that should be publicly available to them at any moment must be only the PNG which is shared by the entire virtual world, and the coordinates of the upper-left corner which uniquely identify it. Since there are no cross-references between chunks, it is very easy to transition from single to multi-threaded chunk generation. Chunks inherit from XNA's DrawableGameComponent base class, which allows them to behave as standalone game components [23, 13, 11, 9]; when overriding the Update(GameTime) and Draw(GameTime) methods we have added a boolean flag which specifies whether the chunk is fully loaded and ready for rendering. If the world seed does not change, the chunk data does not change, so it is not necessary to generate the chunks with each run of the application. We have employed a serialization/deserialization type mechanism for saving generated chunks and loading those that have already been processed. Interesting enough, at this early phase of the application it is cheaper to always re-generate everything than to deserialize from disk, but it is expected that, as chunk data becomes larger and procedural content generation becomes more computationally expensive, saving and loading data will benefit the application.

4.2. **Tweaking the Rendering Process.** Considering these aspects, the virtual world becomes a collection of chunks. Notice that we do not need to draw all chunks with the same level of detail. Of course, chunks that are closer to the player camera must be drawn in higher detail than those further away. If this were the case of terrain sculpted by an artist, it would be quite difficult to obtain these different levels of detail, but with procedurally generated terrain, we simply apply the algorithm of splitting the lattice into smaller subintervals, but we will substitute a SkipFactor proportional to the required level of detail. To further exemplify, remember we split at 64 discrete points on each side; this would correspond to a SkipFactor of 1 and a level of detail (LOD) of 0 – the highest detail. Now, imagine we need an LOD of 1, this means a SkipFactor of $2^1 = 2$ which would correspond to the same lattice now split into 32*32 vertices. As the LOD increases, the number of vertices continues to halve, until it reaches 4 – a single quad (we cannot draw a single vertex). As a general rule, $SkipFactor = 2^{LOD}$ and thus $MaxLOD = \log_2 TerrainSize$. Since we have access to the terrain vertices, we can easily compute some auxiliary shapes that are extremely useful for hidden surface removal. First of all we can compute the centroid of all the vertices for the terrain chunk data, which is done by taking their component-wise arithmetic mean: $G = \frac{1}{VertexCount}\sum_{i=1}^{VertexCount} Vertex_i$. Moreover, we can compute the BoundingBox and BoundingSphere surrounding the terrain as being the smallest box and respectively sphere that surround the entire chunk. This enables us to perform a simple camera frustum containment check and reject chunks outside the player's view, which also allows for a simple form of hardware occlusion query where the much lighter bounding boxes are drawn instead of the entire geometry and fully occluded chunks are also rejected.

By using the centroid of the terrain as a coarse identifier for the chunk, we can determine, based on the distance to the camera, which LOD to draw for the terrain. In the actual implementation we have found that

$$\max(\min(\lfloor \frac{DistanceToCamera}{e * TerrainDiagonal} \rfloor, MaxLOD), 0) \tag{2}$$

suits our purposes (where TerrainDiagonal is the diagonal of the terrain lattice scaled to world units). Most GPUs have what is known as an early depth test, meaning they can discard fragments from the pixel shader stage if another object has already been drawn in front of them. This allows us to speed up the rendering process by simply drawing closer (and presumably larger) chunks first, which is also done by taking into account the distance to the player camera. Furthermore, the GPU prefers drawing larger numbers of polygons in a small number of Draw calls to drawing fewer polygons with a high count of Draw invocations; it is best to gather the polygons from the furthest chunks into a single batch to be sent to the graphics pipeline [6, 7, 8].

One could implement simplex noise instead of perlin noise, which was also discovered by Ken Perlin later in 2001 [20] and has been proven to have a much lower complexity (n^2 compared to 2^n where n is the dimensionality of the noise function). Another useful aspect regarding generating chunks on the fly is that we can attach a camera predictor to attempt to guess in advance where the player might move next based on his previous actions and invoke the procedural generation mechanism even before the player reaches his destination.

5. Conclusions

In the closing chapter of this article we wish to point out our achievements so far, these being the implementation of a hybrid CPU/GPU perlin noise primitive generator and the realization of a large scale virtual world chunking mechanism at an abstract level, with optimized procedural terrain generation, having a pleasant visual appearance (Figure 5). Regarding the improvements

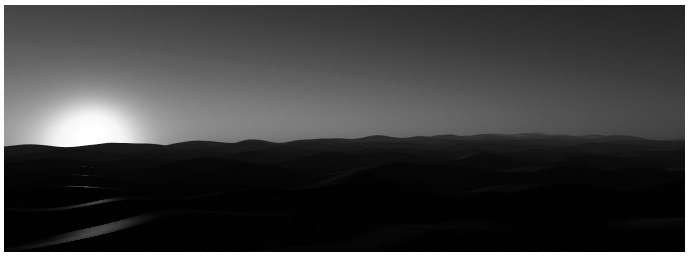

FIGURE 5. Our application running at little over 500 FPS 1366*768 fullscreen, on an Intel Core i5-460M 2.53 GHz CPU and Nvidia GeForce GTX 460M GPU system after all these optimizations have been applied. The world consists of 100 chunks.

over a similar architecture from [21] we should mention that their respective chunking system has voxels as the smallest unitary entity, effectively limiting the engine to drawing volumes made up of cubes, whilst our implementation allows for any type of surfaces/meshes. There have not been many attempts in the gaming industry at creating huge outdoor environments mainly because it is difficult to give the player a goal in such a world; ours has purely research value. Also, a transition to the state-of-the-art DirectX 11 is probable in the near future, and making use of its enhanced API and improved shader capabilities would surely benefit our application, but we feel that there are optimizations which can still be carried out in the older DirectX 9 version supported by XNA.

Finally, we hope that our work has raised the reader's interest on the exciting subject that is procedural generation and we wish to provide some directions for further development which we will also explore: the generation of a procedural sky – star fields, real-time atmospheric scattering, cloud cover formation, movement and dissipation, volumetric clouds; procedural vegetation (including aging of plants) and plant movement according to wind fields; procedural texture generation for different types of materials. As our application's expenses grow larger we will investigate the possibility of offloading all or some of the procedural computations to a dedicated server/web service. With the ever-increasing communication bandwidth we consider this might be feasible.

6. Acknowledgements

This research is supported by Grant No. 30068/19.01.2012, Procedural Generation: Building a 3D world from a single seed, funded by Babeş-Bolyai

University, Cluj-Napoca, Romania, and has been carried out under the tutorship of Assoc. Prof. Ph.D. Simona Motogna[1].

References

[1] Codemasters, Fuel, http://www.codemasters.com/uk/#/uk/fuel-uk/360/
[2] Conspiracy, Binary Flow, http://conspiracy.hu/
[3] Electronic Arts, Spore, http://eu.spore.com/home.cfm
[4] Farbrausch, .kkrieger, http://www.farb-rausch.de/
[5] R. Fernando, GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics, Pearson Higher Education, 2004.
[6] GameDev, GameDev.net Developer Community, http://www.gamedev.net/
[7] R. Grootjans, Riemer's 2D & 3D XNA Tutorials, http://www.riemers.net/
[8] R. Grootjans, XNA 3.0 Game Programming Recipes: A Problem-Solution Approach, Apress, 2009.
[9] S. Hargreaves, Shawn Hargreaves Blog - Game programming with the XNA Framework, http://blogs.msdn.com/b/shawnhar/
[10] A. Marinescu, Achieving Real-Time Soft Shadows Using Layered Variance Shadow Maps (LVSM) in a Real-Time Strategy (RTS) Game, Studia Universitatis Babeş-Bolyai Informatica, 2011, p. 85-94.
[11] Microsoft, App Hub - XNA main developer portal for Windows Phone & Xbox 360, http://create.msdn.com/en-US/
[12] Microsoft, DirectX SDK June 2010.
[13] Microsoft, MSDN Library - Documentation for developers using Microsoft technologies and tools, http://msdn.microsoft.com/en-us/library
[14] H. Nguyen, GPU Gems 3, Addison-Wesley Professional, 2007.
[15] D. Oliver, FractalVision: put fractals to work for you, Sams, 1992.
[16] PCG, Procedural Content Generation Wiki, http://pcg.wikidot.com/
[17] K. Perlin, An image synthesizer, SIGGRAPH '85 Proceedings of the 12th annual conference on Computer graphics and interactive techniques, 1985, p. 287-296.
[18] K. Perlin, Improving noise, SIGGRAPH '02 Proceedings of the 29th annual conference on Computer graphics and interactive techniques, 2002, p. 681-682.
[19] K. Perlin, Making Noise, GDCHardCore, 1999, http://www.noisemachine.com/talk1/
[20] K. Perlin, Noise hardware, Real-Time Shading SIGGRAPH Course Notes, 2001.
[21] M. Persson, Minecraft, http://www.minecraft.net/
[22] M. Pharr, R. Fernando, GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation, Addison-Wesley Professional, 2005.
[23] A. Reed, Learning XNA 4.0: Game Development for the PC, Xbox 360, and Windows Phone 7, O'Reilly Media, 2010.
[24] RGBA, Elevated, http://pouet.net/prod.php?which=52938
[25] Terathon Software, C4 Engine, http://www.terathon.com/

Department of Computer Science, Babeş-Bolyai University, Kogălniceanu 1, 400084 Cluj-Napoca, Romania
E-mail address: `mams0507@scs.ubbcluj.ro`

[1]motogna@cs.ubbcluj.ro