Procedural Generation of landmasses and terrain

Adrian Pop
Constantin Nandra
Cristinel Mihai Mocan
Dorian Gorgan
Technical University of Cluj-Napoca, Romania
adipop2106@gmail.com
{constantin.nandra, cristi.mocan, dorian.gorgan}@cs.utcluj.ro

ABSTRACT
This paper describes a possible implementation of procedural generation to obtain landmasses with realistic terrain featuring unique climates and vegetation. This implementation will be integrated within an application which allows the user to modify the different parameters and see in real time the applied modifications. In order to achieve the generation of an infinite amount of realistic and diverse terrain, there are several challenges that need to be overcome: the generation of data that reflects reality, the placement of trees and other props in a controlled way, assuring a continuous and seamless placement of generated land and a consistent performance on the user's device. Those challenges will be addressed through the implementation of different methods and algorithms addressing the use of noise and spacial transformations.

Author Keywords
Terrain Generation; Procedural Map Generation; Perlin Noise

DOI: 10.37789/rochi.2023.1.1.18

INTRODUCTION

Since the start of its mainstream use in computer games in 90's, 3D graphics have become a staple in the industry due to the increasing accessibility of consumer GPUs and the evolution of console hardware. While in the beginning performance was stable and the models and textures employed were characterized by low levels of detail, the progress made in the field of computer graphics led to an increase in available features, like lighting effects and models of higher vertex count and larger, more detailed textures. These, in turn, have led to an increase in the graphics workload and have consequently increased the strain on hardware requirements. For example while the sixth generation of consoles presented character models with a facial polygon count of just 1200, the eight generation of consoles were capable of a facial polygon count of 32000 [1]. This represents an increase in just facial detail of 2000% in a time period of 13 years.

In every project that extensively utilizes 3D graphics, whether it's a video game or a CGI-intensive film, the presence of a 3D artist or a team of them is crucial. Their responsibility is to create various 3D models for specific scenes, ensuring the accurate definition of shapes and textures, as well as placing and adjusting light sources within a scene. The task of creating 3D models also encompasses the modeling of the environment and terrain featured in a scene. The process of modeling, texturing, and aligning pieces of terrain represents one of the most demanding tasks for a 3D artist, primarily due to the large size of the models involved and the imperative to accurately place every detail. Attention to detail is crucial, particularly when the terrain will be viewed up close, as minor errors such as textures cutoff or sharp falloffs in the terrain can compromise the viewer's immersion. To alleviate the artist's workload and ensure generation within specific parameters, one solution is the utilization of procedural terrain generation.

Procedural terrain generation has been a part of video game media since its inception, being employed in various projects such as the rogue-like dungeon crawler Angband and its derivatives [2]. Its purpose is to facilitate the creation of unique and infinitely diverse content. Procedural terrain generation has been employed multiple times to provide users with a distinct experience each time they interact with an application, while also offering developers a means of automating content creation. However, the challenge lies in fine-tuning the generation process to ensure that the generated content adheres to the required parameters and that the application's performance is not significantly impacted, when compared to a non-procedurally generated version.

In this paper, we will present a method that enables the generation of realistic and diverse terrain by leveraging various parameters and methods. This approach allows us to exert influence over the overall shape of the terrain and the placement of objects within specific zones defined by unique climates.

RELATED WORKS

When it comes to procedural generation, the developer must first grasp the process of generating the data and determine if the chosen method can be seamlessly integrated into the application.

In the case of terrain generation there are several factors that need to be taken into consideration:

- How varied should the terrain be?
- What rules of nature are followed?
- Does the terrain follow a logic different from realistic terrain?
- How much data can be generated?
- Can the data be extended?

The implementation of a terrain generation technique is described in [3]. The implemented technique is utilized within a custom-built rendering engine, which can be expanded by users proficient in programming to develop desktop applications with 3D rendering capabilities. Over the years, various methods have been presented for procedurally generating content using different algorithms or component structures, each offering a unique perspective on programmatically creating data. [4] provides a comprehensive overview of several of these methods, categorizing procedural generation into distinct categories: Pseudo-Random Generators, Generative Grammar, Spatial Algorithms, Simulation of Complex Systems, and Artificial Intelligence.

Perhaps the most well-known pseudo-random generator used in procedural generation is the one developed by Ken Perlin in 1981. As described in the transcript [5], he wished to find a way of generating realistic textures through the creation of visual noise. This was made as a response to the "machine-like" textures used in the film Tron (1981). While noise is usually seen as a disturbance in the transmission of data signals, the algorithm elaborated by Perlin generates noise in a structured and controlled manner, while also keeping an aspect of randomness in the results. Thus, through the use of Perlin Noise, developers and artists are able to generate textures that give off the appearance of depth.

Generative Grammar refers to a theory introduced by linguist Noam Chomsky in 1968, proposing that a limitless number of grammatically correct phrases can be generated by employing a set of rules and a dictionary. This theory has paved the way for the development of various techniques for generating data, allowing content to be generated by defining a set of actions and their sequence. One notable method is the Lindenmayer Systems or L-Systems. Inspired by the concepts proposed by biologist Aristid Lindenmeyer in [6], L-Systems enable the specification of a command sequence that defines the generation process. As it is exemplified in [7] where, through a series of commands which denote actions like movement or rotations, L-Systems can be used in defining the generation of plants like ferns."

To generate models and textures directly, developers can employ spatial algorithms that manipulate the workspace through various operations. One notable method of generation is the utilization of fractals. These, as defined by Benoît Mandelbrot in [8], involve the concept of constructing intricate patterns by iteratively adding a shape to itself at varying scales. Through this method complex models and shapes can be obtained only by using the same simple object. Models that employ fractals present the property of self-similarity, which denotes that, theoretically, a shape can have an infinite amount of detail and observing it at any scale will present similar appearances.

Another implementation of spatial algorithms is through the use of grid subdivision. This represents the action of generating detail by continuously splitting a grid into smaller grids, thus increasing the potential detail only in certain areas. This is implemented in certain algorithms where the emphasis is put on saving performance, like the Patch-Lod algorithm from [9]. In this algorithm the mesh of the terrain has its surface subdivided in smaller portions based on the the viewer's distance. The closer the viewer is, the more detail is presented on the mesh while farther sections have a reduced amount of detail. An approach related to terrain synthesis from minimal-detail user-provided height maps is described in [10]. The article highlights the problems arising from insufficient detail in user input, particularly abrupt changes in altitude and over-simplified feature edges. It explain how the terrain synthesis algorithm is employed to address these issues and generate a level of detail that closely resembles realistic terrain models.

If the developer aims to closely replicate reality, utilizing algorithms based on the simulation of complex systems can be advantageous. The simulation of complex systems denotes algorithms and methods whose processes of creation are inspired by elements found in real life, be it natural phenomenon or human responsibilities. For instance, one potential approach to simulating complex systems involves the utilization of agents, which represent components that interact with each other and whose results influence the results of other agents. The program described in [11] presents a possible use of agents in creating a city with a logically elaborated infrastructure. Using a map that denotes a piece of land, following certain regulations, agents which resemble real life professions will create the road infrastructure, improve it and place different areas which will represent residential or business-oriented buildings. Thus, after a certain amount of time the resulting map will resemble that of an urban area.

While limited in the creation of graphics, AI had been used before in the creation of proceduraly generated content, like in the case of genetic algorithms. In [12], genetic algorithms have been used in the creation of plant meshes. This is achieved by observing the features of every plant and combining only the features needed in order to obtain a new generation of plant models whose appearance is steered in a certain direction.

IMPLEMENTATION

In this paper, several aspects discussed in the previous section will be utilized, combined, and modified to create a landmass that closely resembles those found in reality. Each individual terrain segment, referred to as a "chunk," will exhibit a diverse range of biomes and specific vegetation types. The application presented herein was created using the Unity Engine for the generation of the terrain and the bpy module of Blender to generate tree models using the Sapling Tree Gen add-on. As Unity provides a built-in real-time rendering engine and various options for asynchronous communication, the developer's efforts could be directed towards ensuring the proper

functioning of every aspect of the procedural generator. The Sapling Tree Gen add-on allows the 3D artist to generate in Blender different kinds of tree meshes while also offering the option to manipulate different aspects of the tree's appearance: number of branches, branch splitting distance, the shape of the leaves etc. By utilizing the Python bpy module, developers can interact with the Blender application through scripting without the need to directly access the application itself.

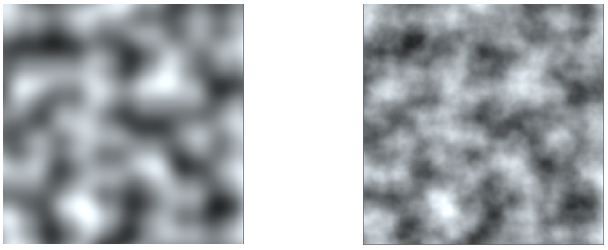
The application will be separated in two different components. the procedural generator created in Unity and a Python script which communicates with the Blender application through the API offered by the bpy module.

Terrain Data Generation

In order to generate the terrain, we need to obtain data based on which we can differentiate the various terrain formations and biomes. For that, we require a method in which we can generate a set of values which would be suitable to denote and influence the positions of the vertices constituting the terrain mesh. One solution for this is the use of noise maps. By utilising a controlled noise algorithm, like Perlin Noise, we will be able to obtain values where the variation between them will be smoother and the transition will allow for the definition of coherent shapes. For the generation of the terrain we need 3 noise maps: a height map for defining the different heights of the terrain and a temperature and humidity map for the definition of biomes.

For each noise map we generate a set of vectors whose values are pseudo-randomly generated based on a seed value. The origin of the vectors will define grids that are part of the map. For every point in the map, there will be vectors in the direction of the point with their origin in the corresponding grid corners. After a function between the grid vectors and the point vectors is applied, the values obtained will be interpolated in order to mitigate possible sharp edges at the value formations. The resulting noise map will be visually represented by different dense worm-like formations as seen in Figure 1.a. The resulting noise map only vaguely resembles real life terrain. In order to achieve a closer appearance to real landmass formations, we need to employ the use of fractals. By adding up multiple noise map layers of different scales, we obtain a noise map with a cloud-like appearance to it, representing softer transitions between different landmasses as observed in Figure 1.b. For the application of fractals, there are several parameters defined that influence the amount of detail added to the noise map: octaves, representing the number of fractals, lacunarity, which will represent the increase of frequency of every layer, and persistence, used to define the detail contribution of every layer.

While the height map uses the normal fractal noise generation, the temperature and humidity distributions should be influenced by the generated height. In the case of the temperature map, the values are directly influenced by the height map, where for example temperatures will be reduced in areas with high altitudes, correlated to the low temperatures normally found in mountainous regions. For the humidity map, there will be several factors determining the humidity for both high and low temperatures. With the noise maps obtained, the placement of biomes on the terrain can be dictated by the temperature and humidity maps.



(a) Simple Perlin Noise (b) Fractal Perlin Noise

Figure 1: Difference between basic and fractal Perlin Noise

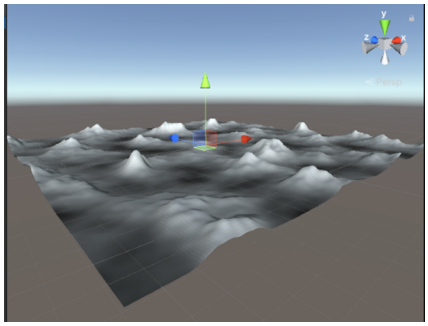


Figure 2: The generated chunk mesh with the height map applied as texture

Chunk Mesh Generation

In order to generate the necessary mesh for the terrain chunk we need to start with a flat 3D plane with the size of the desired chunk. To modify the height of the mesh vertices, we need to use the values generated in the height map. By translating the values of the height map into world space coordinates, the height of the vertices will be modified to resemble the formations found in the height map, as seen in Figure 2. In the visual representation of the noise maps, high values will be represented through shades of light gray to white, thus denoting mountain peaks, and lower values will be represented through shades of dark grey to black, representing areas of lower (negative) elevation. To allow the placement of textures, vertices need to be connected through triangles with the neighboring vertices in order to allow the definition of the UV map, which is used in mapping textures to the mesh.

Prop Generation

When first considering the placement of props meant to represent vegetation, we need to first define a method through which we can ensure a pseudo-random placement. In that sense, we require a placement algorithm that doesn't lead to situations where props are placed close enough to clip through each other or where there are large patches with or without too much vegetation. Using a sampling algorithm, like Poisson Disk Sampling, we can achieve a placement of props that can

suitably occupy the needed space. The placement is deterministic, since it is following a certain seed which is used to generate the chunk, and offers a decent complexity that doesn't impact the process of terrain generation.

The Poisson Disk Sampling Algorithm follows the main rule of not allowing the placement of samples whose distance to the nearest neighbors is lower than a value R. By following the version of the algorithm presented in [13], every correct new sample will be placed in a list of active samples. After randomly selecting an active sample from the list a candidate will be placed in the area denoted by the interval [R,2R] around the selected sample. For every candidate sample, the algorithm checks if the candidate's distance to every neighboring sample is at least R. If not, then it is eliminated. If, after a certain number of tries, no new sample can be placed around the selected one, then the selected one is eliminated from the active sample list. This algorithm will continue as long as the active samples list is not empty. After finishing, the resulting map of samples will have it's space maximized and every sample will be situated at consistent distances from each other, as seen in Figure 3.

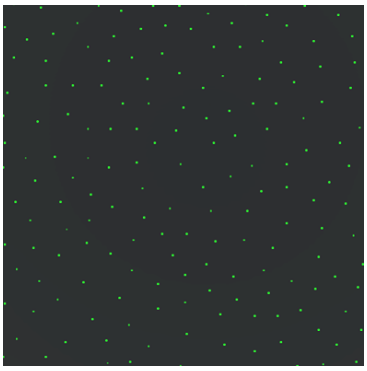


Figure 3: Sample map generated with R=15

The generation of models for the different vegetation props will be achieved by using a Python script which uses the bpy module to communicate with the 3D modelling application Blender. After initially cleaning the scene of the default components, based on the type of model requested, the script will search for data related to said model and, if it finds it, the script will send it to the Sapling Tree add-on in order to generate the mesh. The data in question represents the parameters used in the add-on function and the textures that will be applied to the trunk and, optionally, leaves. After applying the corresponding textures to the model, the script will save the prop model as a fbx file, ready to instantiated in the scene. To ensure that every biome that allows props offers diversity, we shall generate multiple versions of every prop that will be randomly chosen to be placed in the location of a generated chunk sample. Through the use of multi-threading, where the generation script will be called in separated threads, the time necessary to generate a large number of props will be reduced. For every valid prop placement sample, as in a sample situated in a biome that allows props, we shall use the Unity coroutine system to asynchronously check if the necessary model has been generated. If it has, then it will be placed in the corresponding world position coordinates, coordinates which are calculated during the terrain mesh generation. Through this combination, we shall obtain a large series of 3D models of plants and trees that will be placed on the terrain chunk, their appearance resembling that of unique plants fitting for every biome, as seen in Figure 4.

Figure 4: Tree model generated by Sapling Tree Gen

By combining the data obtained in the previous paragraphs, the resulting terrain chunk will be represented by a deformed plane mesh, every deformation representing a different terrain formation, like mountains, as seen in Figure 5. Every biome will be represented by a specific colour, which is applied based on the values of the temperature and humidity map, and every prop instantiated in the scene will have it's model unique to the placed biome.

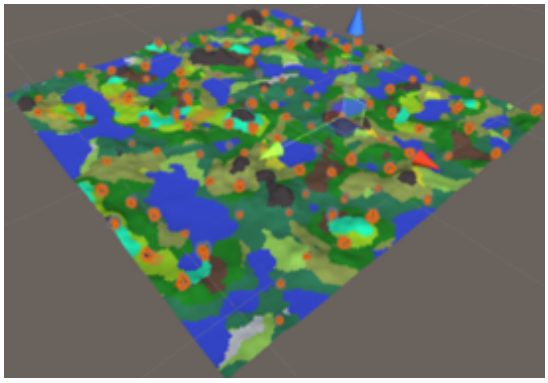


Figure 5: Terrain chunk. Every prop is highlighted by an orange outline

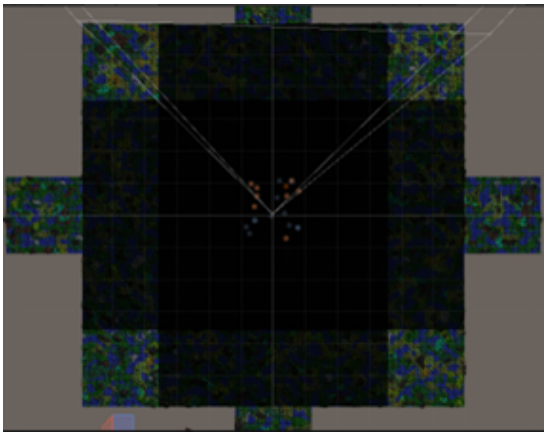


Figure 6: Different LOD meshes applied in the Endless Terrain

(a) Single chunk interpolation effect

(b) Multiple chunk interpolation effect

Figure 7: Difference between different interpolation methods

Endless Terrain Generation

The next step in the generation process is the placement and generation of different chunks in order give off the illusion of an infinite, continuous world. Initially, a set of chunks situated in a certain range from the center of the coordinate system will be generated. By keeping track of the position of those chunks, the coordinates can be applied to the grid vectors used in the noise generation phase. This ensures that the noise generated is continuous between adjacent chunks. Because it is technologically impossible to generate an infinite amount of chunks at once, there needs to be condition which will guarantee that chunks would be generated only on certain events. This is achieved by keeping track of a reference object, designated viewer, which will act as a representative for the user interacting with the application. Said object is capable of being moved using the WASD keys. At every frame, the location of the viewer will be checked. If the viewer travelled a certain distance in a direction then new chunks will be generated.

To ensure that performance isn't severely affected by the large number of meshes that are generated in the scene, the hiding of far away objects and the use of LODs or Levels Of Detail is applied. For every prop and chunk that is considered too far away for the viewer to properly see, their meshes will be hidden and made visible again once the user gets within a certain minimum distance. For the terrain chunks that are in certain proximity tiers to the viewer, there will be different levels of detail applied to the meshes. Every chunk will have several meshes based on a varying level of detail. Those meshes will be switched around based on the viewer's position to the chunks. During the generation of a LOD mesh, based on the level value, consecutive vertices will be ignored, thus ensuring that the meshes will still keep the same overall shape but with less vertices present for the render engine to compute. The different levels of detail applied to the meshes can be observed in Figure 6.

While the generated noise values are continuous between chunks, the vertices that make up the chunks are not guaranteed to align properly. This will cause the apparition of seams in the terrain. This is due to the fact that the interpolation used for the height map values in the single chunk generation is based on the maximum and minimum height values. These values won't be consistent between multiple chunks. For the sake of chunk connectivity, during the endless generation phase, a different interpolation method will be used, based on the maximum height between all chunks. This will ensure that the height of the vertices will follow the same range of values. The comparison between the two interpolations can be observed in Figure 7.

Another problem related to continuity that needs to be corrected is the presence of seams caused by the texture colours. Due to the fact that every chunk is generated separately, the values of the normal vectors are also treated as separate, which will lead to light influences that will create subtle but noticeable seams at the edges of the terrain chunks. This is corrected by recalculating the normal vectors for every vertex during the chunk generation phase. Every vertex will be placed into 2 categories: marginal vertices and inner vertices. The triangles that contain only inner vertices will compute the normal value for the triangle, which will be added to the normal value of every vertex. In the case of triangles which contain marginal vertices, the calculated normal value will be added only to the inner vertices. Through this method, every marginal vertex will have the same normal value, which would lead to the disappearance of the seams caused by different light interpretations.

RESULTS

By applying all the previous methods and algorithms, the application is capable of generating an infinite amount of terrain chunks, each one with different landmass formations, different concentrations of biomes and a different placement of specific props. The chunks are capable to be placed next to each other, in order to create the illusion of one continuous mesh, that will be continually generated based on how much the viewer travels in a certain direction.

For the purpose of performance testing, the application had been tested on a Asus laptop with the following specifications:

- Intel Core 2.20 Ghz CPU
- 8GB RAM memory
- NVIDIA GeForce GTX 1050TI GPU.
- Windows 10 OS

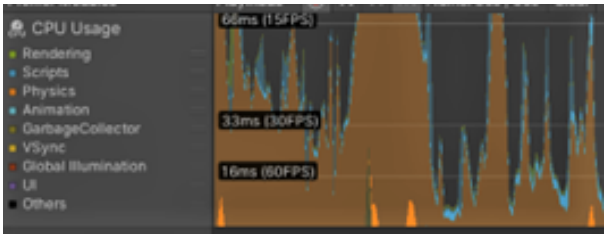


Figure 8: Profiler results when running both the application and the prop model generation

Since the generation of terrain chunks is limited to only a certain size and the entire data is generated on the go, the impact that comes from the chunk generation is minimal at best. On the other hand, due to the fact that the generation of props is tied to an outside application and that the communication is done through an API, the placement of props will cause the most amount of performance hits.

Several tests were conducted to evaluate the performance of the endless terrain generation phase. The focus was on measuring the loading time of data and the overall performance in terms of Frames Per Second (FPS). The tests considered variations in the sample radius and the generation of prop models. The Profiler tool provided by Unity was utilized to collect the necessary data for analysis. The Profiler will track the overall performance in FPS of the application and the sources that cause fluctuations: script activity, the physics engine, the rendering etc.The results can be observed in the following table:

Nr	Show Props (bool)	Generate new Props (bool)	Sampling Radius	Start time (sec)	Average FPS
1	False	False	-	<1	50-60
2	True	False	40	7	50
3	True	True	40	3	20-30
4	True	False	20	13	45
5	True	False	10	30	30-45

By observing the values found in the table, we can deduce that the action of generating all the necessary prop models has a significant impact on the overall performance of the application, causing a decrease of at least 20 FPS when compared to the case where no new models are generated. The impact of the generation activity is presented in the profiler screenshot found in Figure 8, where it can be observed multiple spikes in FPS loss, resulting in an overall reduced and slow performance. These issues are caused by the CPU resources being consumed by the threads used to execute the prop generation scripts.

When the model generation is disabled, the use of props still has a certain impact on the performance. By decreasing the sampling radius, the number of instantiated props will increase, which in turn will lead to a denser concentration of props that need to be rendered. In addition, the application will require a larger amount of time to start, due to the high amount

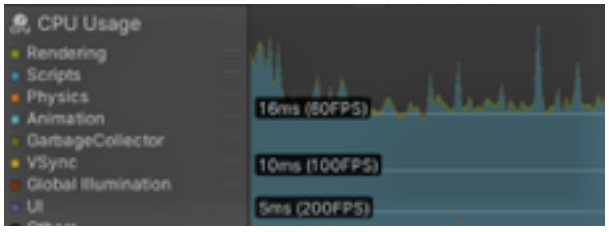


Figure 9: Profiler results when running the application with a minimum sample distance of 20

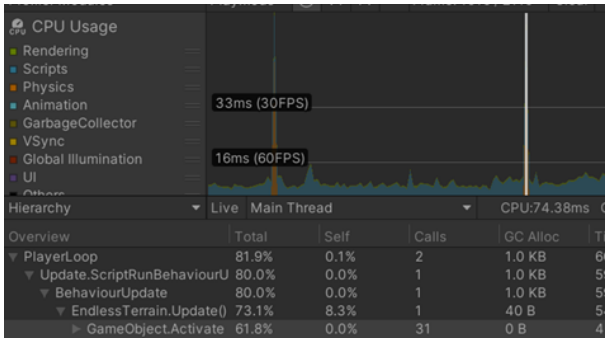


Figure 10: Profiler results when generating only the terrain chunks

of coroutines that the engine needs to switch between. The impact of this is shown in Figure 9. When stationary, the performance remains consistently at 60 FPS, but when the viewer is in motion, the visibility of the props is updated, causing slight fluctuations between 50 and 60 FPS.

The application is at it's peak when not showing any props, the endless generation being capable of holding an almost consistent 60 FPS. What needs to be noted though, is the fact that there are slight spikes in performance caused by the generation of new chunks, as shown in Figure 10. In the overview tab, one can observe that the majority of the performance impact occurs during the update of chunk values. This can be attributed to the generation of new chunks and the alteration in detail levels of the existing chunks.

CONCLUSIONS

The application described in this paper allows the generation of an infinite amount of unique terrain chunks. By utilising a combination of noise and sampling algorithms, multithreading techniques and asynchronous communication, the resulting performance of the terrain generation tool is quite decent, being able to keep an almost consistent value of 60 FPS. What does bring down the application is the generation of vegetation props which would require a rework. Due to the dependence to an outside application API and the use of threads, the number of FPS drops significantly since the CPU is pushed to it's limits.

Overall, the solution presented in this paper puts into perspective the idea of efficient terrain modelling while also allowing for the posibility of creating complex ecosystems with unique temperature and humidity distributions.

Acknowledgment

The work was supported by the project "Entrepreneurial competencies and excellence research in doctoral and postdoctoral programs—ANTREDOC", project co-funded by the European Social Fund, financing agreement no. 56437/24.07.2019.

REFERENCES

[1] S. Schuman, "The polygonal evolution of 5 iconic playstation characters," 12 2019, https://blog.playstation.com/2019/12/16/the-polygonal-evolution-of-5-iconic-playstation-characters/.

[2] N. Shaker, J. Togelius, and M. J. Nelson, *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2016.

[3] L.-V. Deaconu, C. Nandra, and D. Gorgan, "Terrain generation with continuous level of detail," 01 2021, pp. 44–51.

[4] J. V. A. I. M. Hendrikx, S. Meijer, "Procedural content generation for games: A survey," *ACM Transactions on Multimedia Computing, Communications and Applications*, vol. 9, no. 1, pp. 1–22, 2012. [Online]. Available: https://doi.org/10.1145/2422956.2422957

[5] K. Perlin, "Making noise," 10 2007, https://web.archive.org/web/20071011035810/http://noisemachine.com/talk1/.

[6] A. Lindenmayer, "Mathematical models for cellular interactions in development ii. simple and branching filaments with two-sided inputs," *Journal of Theoretical Biology*, vol. 18, no. 3, pp. 300–315, 1968.

[7] H. M. G. Kelly, "A survey of procedural techniques for city generation," *The ITB Journal*, vol. 7, no. 2, 2006.

[8] M. J. Kirkby, "The fractal geometry of nature. benoit b. mandelbrot. w. h. freeman and co., san francisco, 1982. no. of pages: 460. price: Â£22.75 (hardback)," *Earth Surface Processes and Landforms*, vol. 8, no. 4, pp. 406–406, 1983. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/esp.3290080415

[9] X. Pi, J. Song, L. Zeng, and S. Li, "Procedural terrain detail based on patch-lod algorithm," in *Technologies for E-Learning and Digital Entertainment*, Z. Pan, R. Aylett, H. Diener, X. Jin, S. Göbel, and L. Li, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 913–920.

[10] A. P. Mangra, A. Sabou, and D. Gorgan, "Tsch algorithm-terrain synthesis from crude heightmaps." *Romanian Journal of Human-Computer Interaction*, vol. 9, no. 2, 2016.

[11] T. Lechner and U. Wilensky, "Procedural city modeling," 01 2003.

[12] O. D. B. Lintermann, "Interactive modeling of plants," *IEEE Computer Graphics and Applications*, vol. 19, no. 1, pp. 56–65, 1999.

[13] R. Bridson, "Fast poisson disk sampling in arbitrary dimensions," 2007.