Politechnika Wrocławska

Faculty of Computer Science ang Management

Field of study: Computer science
Specialty: Information Systems Design

Master Thesis

Methods of procedural environment generation in games

Grzegorz Maciejewski

Keywords:
terrain, terrain generation, comparison, review, Unity, Godot, procedural generation

Short summary:
Thesis includes review of methods of procedural terrain generation and comparison of chosen algorithms in two most popular engines, research is focused around performance and conclusions are supported by statistical analysis and tests.

Supervisor	Dr inż. Marek Kopel	....	....
	Title/degree/name and surname	*grade*	*signature*
The final evaluation of the thesis			
Przewodniczący Komisji egzaminu dyplomowego	....	....	....
	Title/degree/name and surname	*grade*	*signature*

*For the purposes of archival thesis qualified to:**

a) Category A (perpetual files)

b) Category BE 50 (subjects to expertise after 50 years)

** Delete as appropriat*

Stamp of the faculty

Wrocław [2020]

Streszczenie

Praca skupia się na porównaniu różnych możliwych metod proceduralnego generowania terenu w silnikach, które służą do tworzenia gier komputerowych. Część teoretyczna pracy wprowadza w tematykę generowania terenu, a także przedstawia poszczególne metody jego generowania z podziałem na algorytmy teleologiczne oraz ontogenetyczne. Dalsza część pracy oraz badania skupia się na tych drugich czyli ontogenetycznych, wybrane zostały cztery algorytmy szumów: simpleksowy, Perlina, fraktalny oraz diagram Voronoia. Na danych zebranych z wykorzystaniem wybranych algorytmów została przeprowadzona analiza wstępna oraz badania statystyczne weryfikujące konkretne hipotezy badawcze. Końcowa część pracy zawiera podsumowanie przeprowadzonych badań oraz wnioski wyciągnięte na ich podstawie, wskazane zostały również możliwości przeprowadzenia dalszych badać, być może będących kontynuacją lub rozszerzeniem tej pracy.

Abstract

Thesis is focused around comparison multiple possible methods of procedural terrain generation with engines used to create video games (game engines). The theoretical part of the work is an introduction into procedural terrain generation topic, and also contains brief description of possible methods broken down into individual algorithms, both teleological and ontogenetical. Further part of the thesis including research are focused around ontogenetical algorithms, four of those were chosen to further investigation, these noise algorithms are: simplex noise, Perlin noise, fractal noise and Voronoi. The data collected using chosen algorithms, initial analysis was performed, further analysis included proper statistical tests to verify proper statistical hypothesis. Final part of the thesis contains summary of performed tests and research along with conclusions drawn on the basis of research, there are also indicated further possible ways of research possibly extending those performed in this thesis.

Table of contents

Introduction

In game development there is always a part when designers face the problem of creating the world. Game world must be interesting and real. There are many successful games where designers make the terrain from scratch such as Warcraft III, but sometimes the requirements are too big when faced by humans. What if the requirement is that the world must be infinite, or unusually enormous. That is when procedural generation can come with help.

Procedural generation is a way to generate massive areas at once, without human hand. It is desired that generated environment should be as much similar to the real world as possible and also it should be non-linear. Procedural generation is used to make forests, wide-empty mountainous areas, and areas covered with vegetation.

There are many various algorithms and methods capable to accomplish the goal, some of them will be referenced in this work.

1. Review of related work

In his work Adam Malek describes briefly the topic [1] of generating boards in online games, that topic is slightly related to this thesis topic, as I am particularly interested in auto-procedural generating algorithms that are using noises.

Most important things that are mentioned in this work [1] are generating the shape of the board as well as generating separate elements of the board. Thesis mentions about evaluating a solution – algorithm, which can be very helpful as it will be possible to reference to this work when evaluating other algorithms. In chapter 3.7 there are specified a few determinants to evaluate a solutions, the determinants are:

- The correctness of the content
- Uniformity of gameplay
- Time of generating and loading of the board
- The amount of the data sent do a player, needed for the board to be created.

In another article author describes his way of generating terrain [2] using Minecraft as an example, it is generated in the time when player is actually moving - “on the fly” therefore it is infinite. That term “infinite” is specifically explained by the author of article, so called infinite it nothing else than just generating the surrounding area around the player while he moves forward and it is not really infinite hence we are still limited by the hardware for example and also by the limit of integer number that can be stored in the register.

The cubes of which the whole world is created have coordinates, each cube has its position in the whole world and when map is too large, coordinates of single cube are getting closer to integer maximum number. Once the number is exceeded, errors may appear. The problem was partially solved in Minecraft, by segmenting the whole world for smaller areas and coordinates of cubes are referencing to the local area – not the whole world. However as mentioned in [1], there still are algorithms which use the world position of the cube instead of local area position.

Figure 1: Minecraft cubes

Local areas in Minecraft are called chunks which is an 3D object of size 16x16x128 blocks [2]. Author of terrain-generation article writes about 2D Perlin noise [3] and 3D Perlin noise:

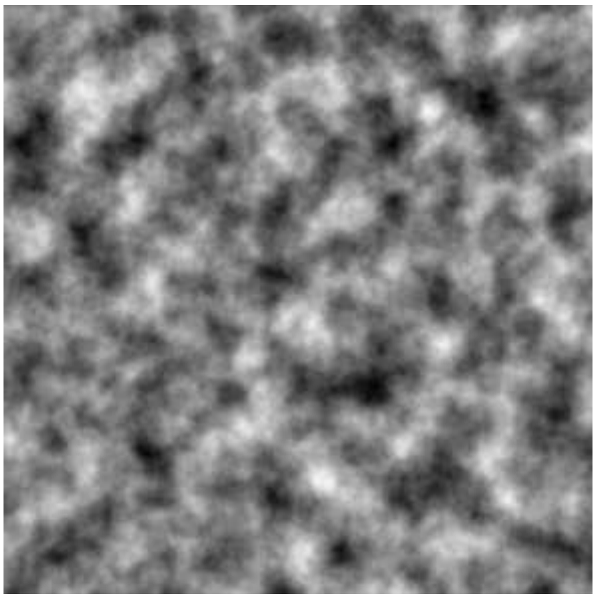


Figure 2: Perlin noise illustration [1]

Notch mentions about lack of performance when switched from 2D to 3D Perlin noise, due to massive sampling. It is worth to mention that this problem was solved by sampling at lower resolution. It resolved also problem regarding “playability issues” [2], because of unpleasant terrain.

Using Perlin noise to generate maps is well known approach and it will be referenced further.

In another work [4] Laura Oravakangas touches the topic of environment creation process. The whole thesis is very detailed and involves not only environment designing process itself but also how to create assets, how designing methods are related to game environment. Work also mentions a topic of optimization of particular elements and its cooperation such as game environment, optimal assets, level design.

Oravakangas distinguished two types of environments:

- Seamless
- Modular

The difference between these two are intuitive, the first one is more natural, there are no rapid changes in environment. For example in modular world it is possible to cross from the ice world to lava world directly. Designing modular world requires a lot more effort than designing seamless world.

Seamless world is kind of constant in its form, without rapid transitions between the areas like in modular world. There are illustrations [4] to see the difference more clearly:

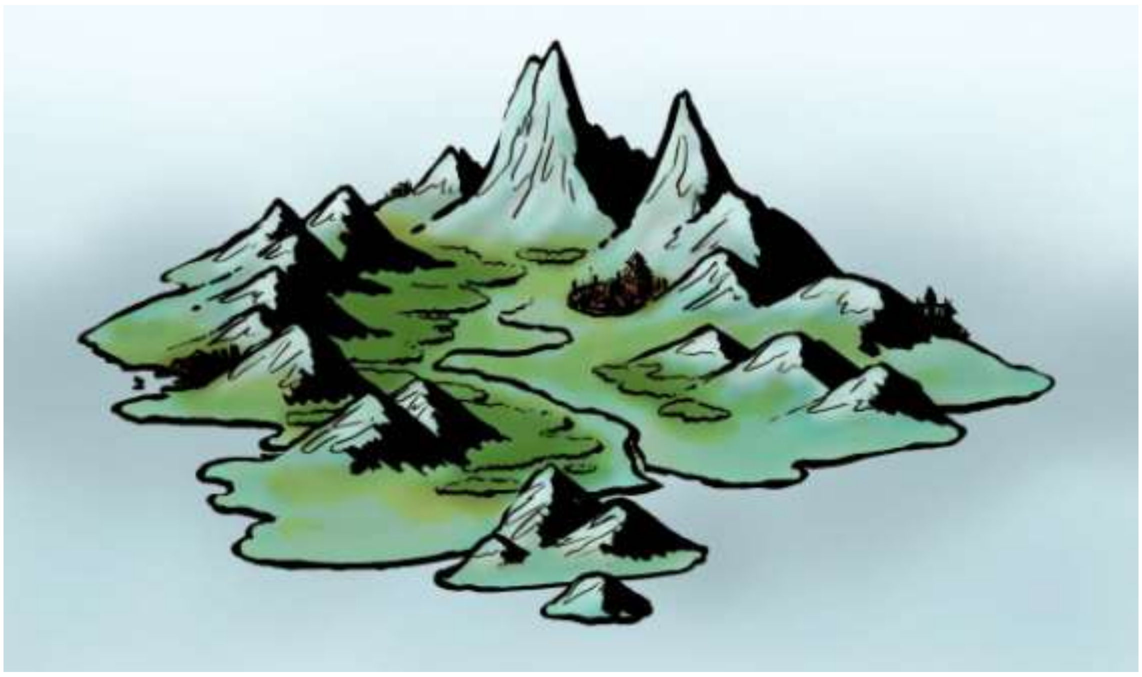


Figure 3: Seamless world [4]

Figure 4: Modular world [4]

It is up to the designers which world would they want to use, and which one will meet their requirements best. Normally environment is designed and created by hand, designers or artists make the world from scratch using graphic programs and game development environments.

Specifically in chapter 9.4 in work [4] procedural generation is mentioned, and again Minecraft is referenced. It is crucial to have possibility to modulate parameters of procedural generation algorithm as it should not be fully random.

Author mentions that there is possibility to generate procedural environment using previously crafted areas, good example of that could be Diablo III game.

A problem with auto generated terrain is accessibility, it is possible that there will be not-accessible point – that problem will be developed further in the thesis. As a disadvantage of procedural environment we can stand out also repetitive patterns [4], while playing the game we can be under the impression that we see the same part of the map multiple times. Oravakangas also refers to [5] claiming that it is highly not efficient to generate small areas using procedural-generation algorithms, and it is better to use hand-crafted areas.

1.1. Objective and scope of the thesis

As mentioned before, there are a lot of possible solutions to generate a map, board or anything we need in particular, there always must be a choice. Developers and designers face many problems, one of them is performance and the end result at the end of it. I did not found any proper comparison of procedural generation algorithms which could help solve that problem, in particular comparison of noise algorithms used in procedural terrain generation.

Therefore objective of this thesis is to compare a few procedural generation algorithms and implementation with each other in order to discover differences especially in performance in terms of the final result.

To evaluate results I will use determinants which will be described in following chapters

Scope of the work includes:

- Analysis of literature
- Analyzing procedural generation noise algorithms
- Prepare tests
- Gather results of the tests
- Analyze test results
- Conclusions

In this thesis I will focus on comparing noise algorithms, which are often used to generate height maps in order to generate part of the map.

2. Algorithms

In this chapter I will describe multiple types of algorithms that can be used to generate terrain, which will lead to choose specific implementations.

Procedural generation algorithms can be divided for

- **Teleological**
- **Ontogenetical**.

Teleological algorithms, are trying to simulate real world processes in order to accomplish desired goal – which is to create world as closest to reality [6], these algorithms are simulating natural processes they are a kind of nature-reproducing machines.
On the opposite to teleological algorithms there are ontogenetical algorithms, these one are not simulating physical or natural processes as teleological. Ontogenetical algorithms are in simple words "goal based" which means that it is trying to reach the end result without intermediate steps.

2.1. Teleological algorithms

There are several teleological algorithms known, among them [6]

- Genetic algorithm
- Rain drop
- Fire propagation
- Markov chain

2.1.1. Genetic algorithm

Genetic algorithm is an example of teleological algorithm [7], it reproduces the process of population evolution in can be described in a few steps:

- Generate initial population
  - Repeat:
    - Evaluate fitness value for entire population, separately for each member
    - Remove members that are not good enough

- Crossover the rest to refill population

Above steps illustrate the simplest version of genetic algorithm but the general idea is visible, the goal is to maximize the value of fitness function so the only best members of population reproduce their genotype. Genetic algorithm can be successfully used in procedural generation of multiple elements, for example it can be used to generate levels of platform game [8], however it can also be implemented in order to generate terrain in 3D games [9], authors of referenced work used genetic algorithm to create 3D data sets based on input database of given terrain data, generated terrain is the result of processing provided database by the algorithm.
Genetic algorithm was a constant target of terrain generation related research and study [10], Walsh and Prasad are introducing method called "*Auto Terrain Generation System*" based on genetic algorithm. Another example would be an incredible work made by multiple authors Li, Qicheng, Wang, Guoping [11] which introduces another generic algorithm based approach to produce terrain so called four-process approach.

2.1.2. Rain drop

Rain drop algorithm is kind of self-explaining, because it is literally algorithm based on reproducing the effect of rain dropping on the surface for a long time.
The idea is to show the effect of erosion. This algorithm is specifically used to add some shapes to the world that is too smooth.

The way it works is simple. Some pixels on the world are selected randomly [12] these will represent actual raindrops to "fall". Once the pixels are selected, first thing to check for each pixel is if the raindrop is not below sea level which in the game world simply means some certain vertical position on the map – if it is, then go to next raindrop. Otherwise simulate gradient descent process, which in the simple words means to identify local neighborhood of the raindrop and the material in raindrop itself then move the raindrop random amount of material down the hill, along the gradient. It starts with a small amount of material and grows up while its moving down. The process stops once raindrop reaches predefined sea level.
Erosion is a process that was an object of multiple scientific research especially about procedural terrain generation, it is natural process that changes environment in real life situations, it is reasonable that there are number of publications like Olsen [13], that raise the topic of erosion among others, this publication also mentions other methods of procedural generation like Voronoi or midpoint-displacement and fractal noise.

See below comparison of before and after the algorithm was run.

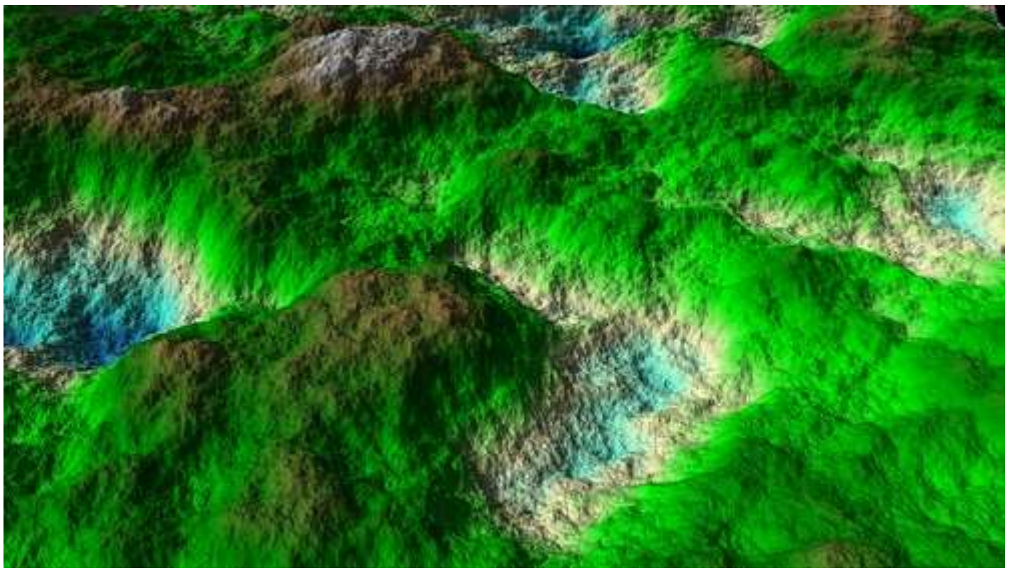


Figure 5: Terrain in its original state [14]

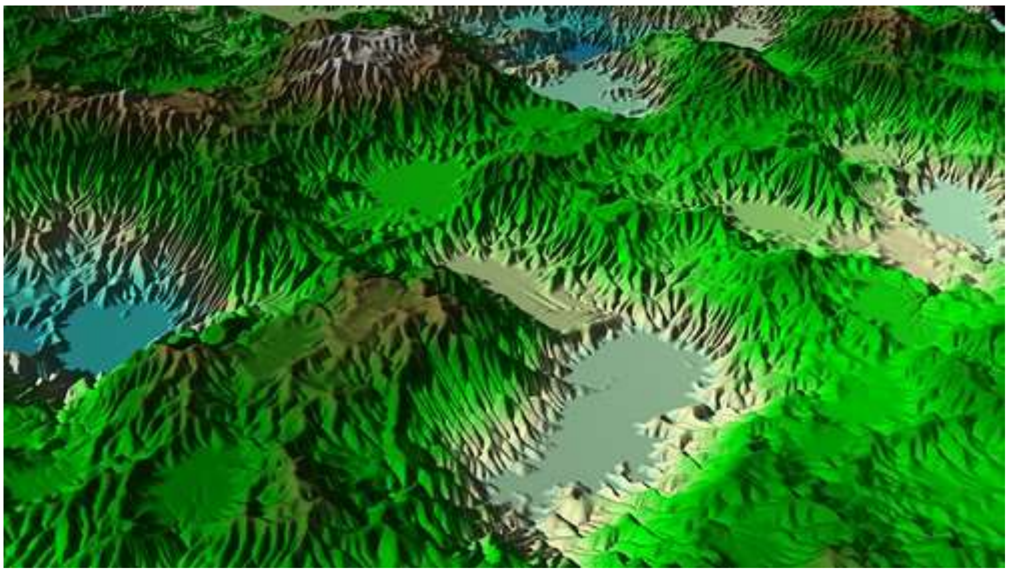


Figure 6: Terrain after a few iterations of raindrop algorithm [14]

Figure 7: The effect of raindrop algorithm running too long [14]

2.1.3. Fire propagation

Fire propagation algorithm simulates the process of spreading of fire [15]. It can be implemented in more that a few ways. It is mentioned that the algorithm reflects the way of spreading fire under dynamic-systems [15] such as weather or other actions.

Depending on the game type (3D or 2D) fire propagation algorithm may work differently, for example in 3D game it might use particle system to spread fire. In 2D game we might just use grid on the board and light up areas adjacent to the source if other conditions are met.

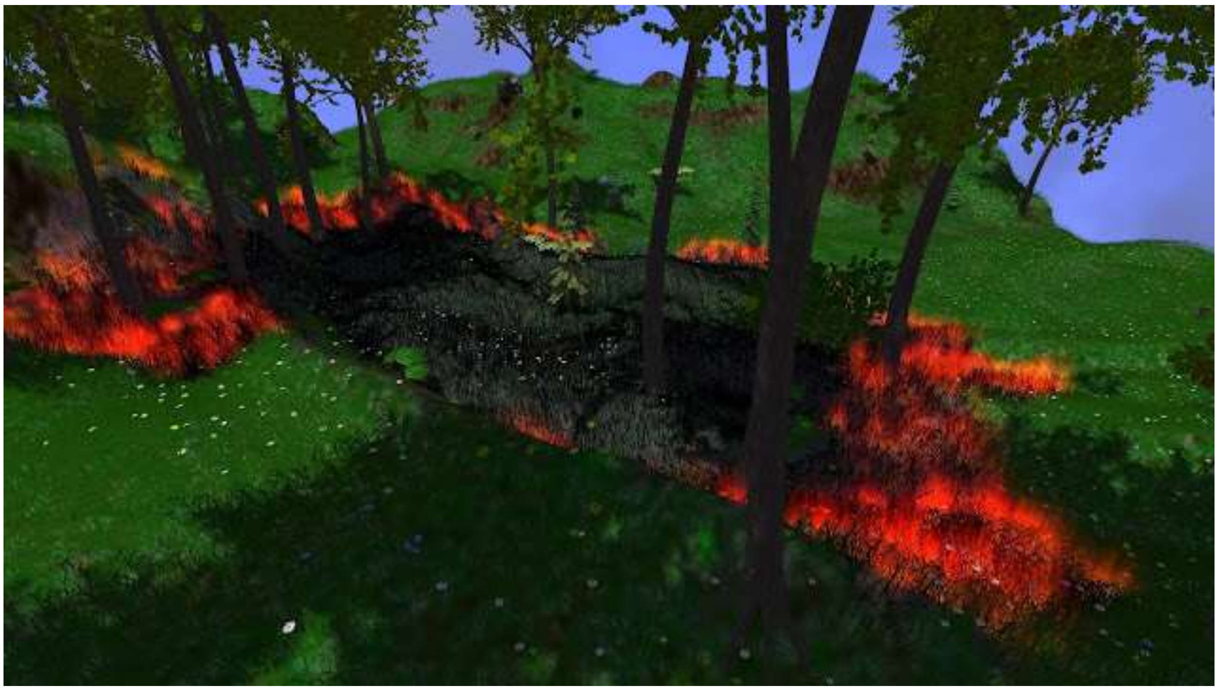


Figure 8: Fire propagation algorithm in 3D [16]

Fire propagation algorithm is demonstrates a way to actually modify already existing terrain. It changes the environment by simulating fire spread, therefore it can be said that this is the actual way to “generate”. Fire propagation algorithm was an object of research over the years as well, Mueller in his thesis [17] successfully implemented his own approaches to fire propagation algorithm using snowdrop engine, the project is available to further extensions and adjustments for own use.

Cellular Automata

One of worth mentioning ways to model fire propagation algorithm particularly on 2D grid is cellular automata [18]. It is a simple grid of cells with specific behavior implemented, each cell on a grid has a state. The sate is determined by the other cells around, in other words, the decision of which state the cell is in, is based on the states of adjacent cells.

2.1.4. Markov chain

Markov chain is an algorithm used to create entitled chain of values connected with each other – on the opposite to the chaotic non related noise generated values.
In general Markov chain describes a chain of events where probability of incoming event depends totally and only from the result of preceding event [19] however some higher order Markov chains allow the states to be dependent on more than one previous state.

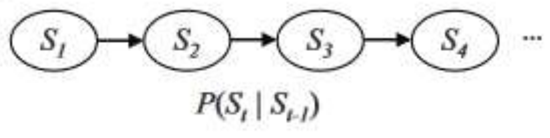

$$\begin{tikzcd} S_1 \arrow[r] & S_2 \arrow[r] & S_3 \arrow[r] & S_4 & \cdots \end{tikzcd}$$
$P(S_t \mid S_{t-1})$

Figure 9: Markov chain illustration [19]

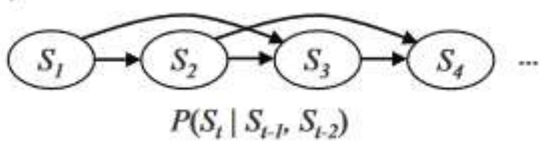

$$\begin{tikzcd} S_1 \arrow[r] \arrow[rr, bend left] & S_2 \arrow[r] \arrow[rr, bend left] & S_3 \arrow[r] & S_4 & \cdots \end{tikzcd}$$
$P(S_t \mid S_{t-1}, S_{t-2})$

Figure 10: Higher order Markov chain illustration [19]

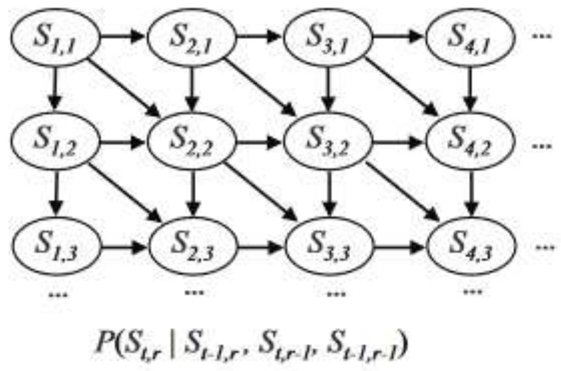


Figure 11: Two dimensional higher order Markov chain [19]

The described algorithm can be successfully implemented and used to procedurally generate a board or map for 2D game as in cited example. Authors used Markov chain to generate the map of Mario game. At first the algorithm needs to be learned, in order to do that there is an input matrix D which is a dependency matrix on the basis of which further probabilities of Markov chain are calculated. The learning method of course depends on the need and might be different for each map or game. Markov chain is widely used in number of publications, other example can be generating terrain for platform games as Gleidson and Borchartt did [20], it is mention that this methodology provides unique and non-repetitive terrain, authors based their approach on hidden Markov chain, below see the example of complete level generated using the method proposed by the authors:

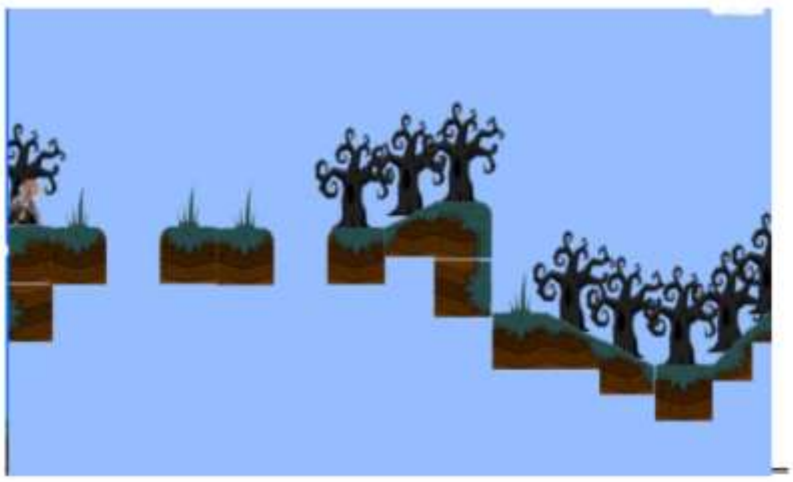


Figure 12: 2D game level generated by Markov chain method [20]

2.2. Ontogenetical algorithms

On the other side to the teleological algorithms, there are ontogenetical algorithms such as

- Noise-based algorithms
  - Simplex noise
  - Perlin noise
  - Fractal noise
- L-System
- Midpoint displacement algorithm
  - Diamond-Square algorithm – an improvement to the midpoint displacement algorithm
- Voronoi diagram

Noises

Noises are very important part of procedural terrain, maps and textures generation [21], noises itself are used to receive pseudo-random values. Most common usage of noises is to generate areas and textures such as wood, cloud, mountains – terrain in general.
Its worth mentioning that noises were not commonly used to generate terrain and textures in real time due to high computing power consumption [21], nowadays it is possible due to parallel computing on multiple threads.

Noise itself is a multi-dimensional function, it can be defined in any number of dimensions – more dimensions consumes more computing power. Most common implementation includes 3 dimensions therefore three parameters (coordinates) must be provided to the function.

2.2.1. Perlin Noise

Perlin noise is the most basic and popular noise among others. It was initially developed by Ken Perlin in 1983, and it is illustrated on Figure 2: Perlin noise illustration . The purpose of It initially was to procedurally generate 3D textures looking more natural. Most common implementation includes 3 steps [22].

- Grid definition
- Dot product
- Interpolation

Grid definition

In this step a n-dimensional grid of unit vectors is generated. Pseudo-random gradient vector is assigned to each node of the grid.

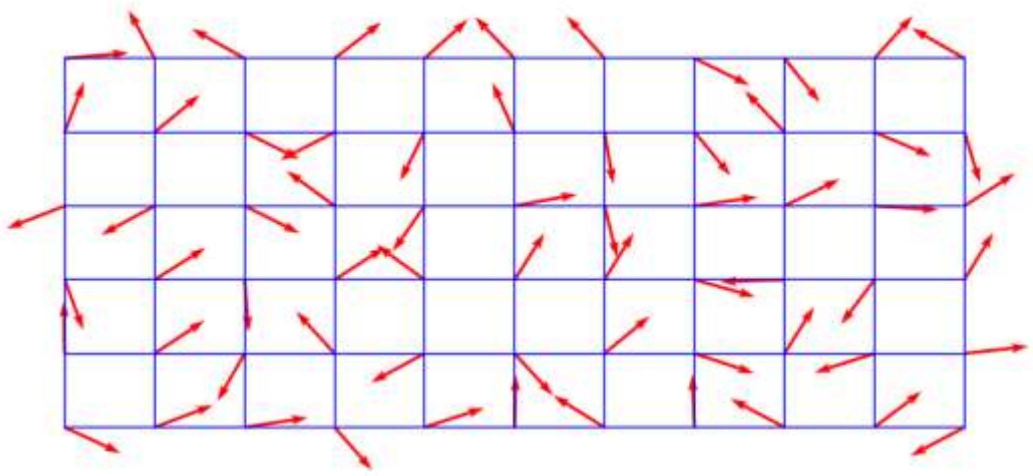


Figure 13: 2-dimensional grid of gradient unit vectors [22]

Dot product

In dot product phase, we have to calculate one vector, for each node – so 4 for 2D board. The vector should connect the point for which we want co calculate the value of Perlin noise function

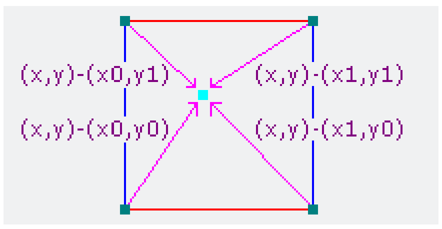


Figure 14: Dot product phase illustration for single cell [3]

The dot product is the result of scalar product between these two vectors calculated in previous steps (gradient vector and distance vector shown above).

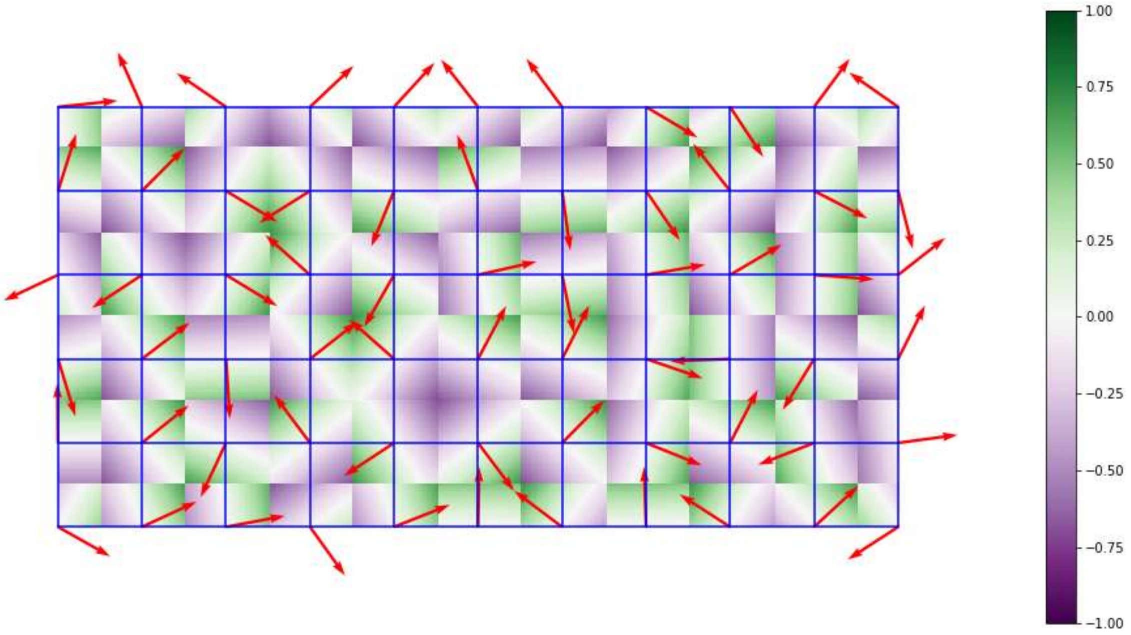


Figure 15: The grid after dot product phase [22]

Interpolation

As shown on Figure 15: The grid after dot product phase , the result of dot product are usually measured between 1 and -1. The sign depends on the direction of the gradient. The last phase after that is to interpolate all four values in order to receive outcome value.

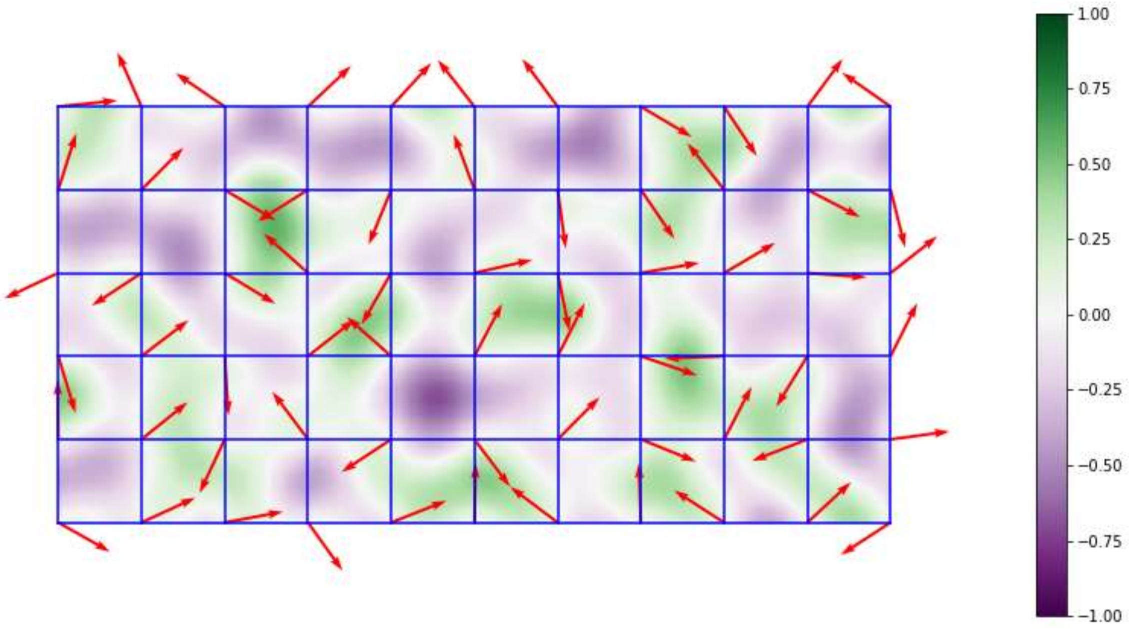


Figure 16: Result of interpolation [22]

Perlin noise, as the most basic noise is an object of multiple research and scientific works for example [23] or [24], and many others. Perlin noise is an starting point or point of reference in works due to its similarity to other noises such as simplex noise.

2.2.2. Simplex noise

Simplex noise is comparable to Perlin noise [25], it can be used for n-dimensions. The difference is in computational complexity – simplex noise is less complicated and requires less computing power to reach the same goal. It's complexity is:

$$O(N^2)$$

Where N is number of dimensions. Perlin noise complexity is:

$$O(2^N)$$

Other difference between simplex and Perlin noise is that simplex noise operates on the grid on n-dimensional triangles instead of hypercubes as Perlin noise do [26].

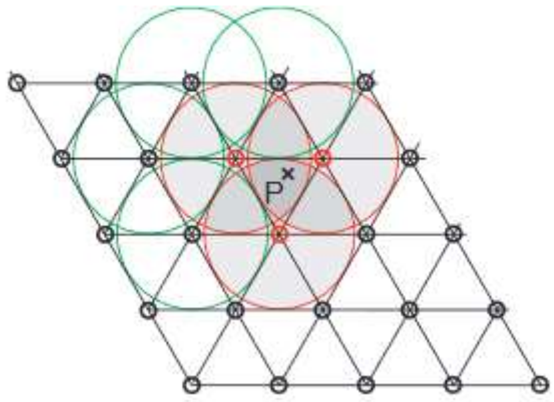


Figure 17: Simplex noise triangle-grid [26]

As noise very close to Perlin, simplex noise is also very often compared to others. It is expected that it will work faster than Perlin noise in large number of dimensions. References to simplex noise in comparison to others can be found in number of publications [27], or [28] where authors investigate different methods of procedural generation focused on Two-Dimensional terrain. They also point at the problem that there is no proper survey or comparison of procedural terrain generation methods.

2.2.3. Ridged – multifractal noise

Ridged multifractal noise very similar to others with one exception [29], all output values that are less than zero are taken into absolute value. As a result of that ridge formations are created therefore this particular noise of often used to generate mountainous terrain areas.

References to described multi-fractal noise, or fractal noise can be found in many publications often with Perlin and simplex noise for example [13] simple fractal noise has been used to generate terrain along with erosion algorithms, fractal noise was also a part of an research [30] mentioned as a way to generate heightmap for mountainous areas.

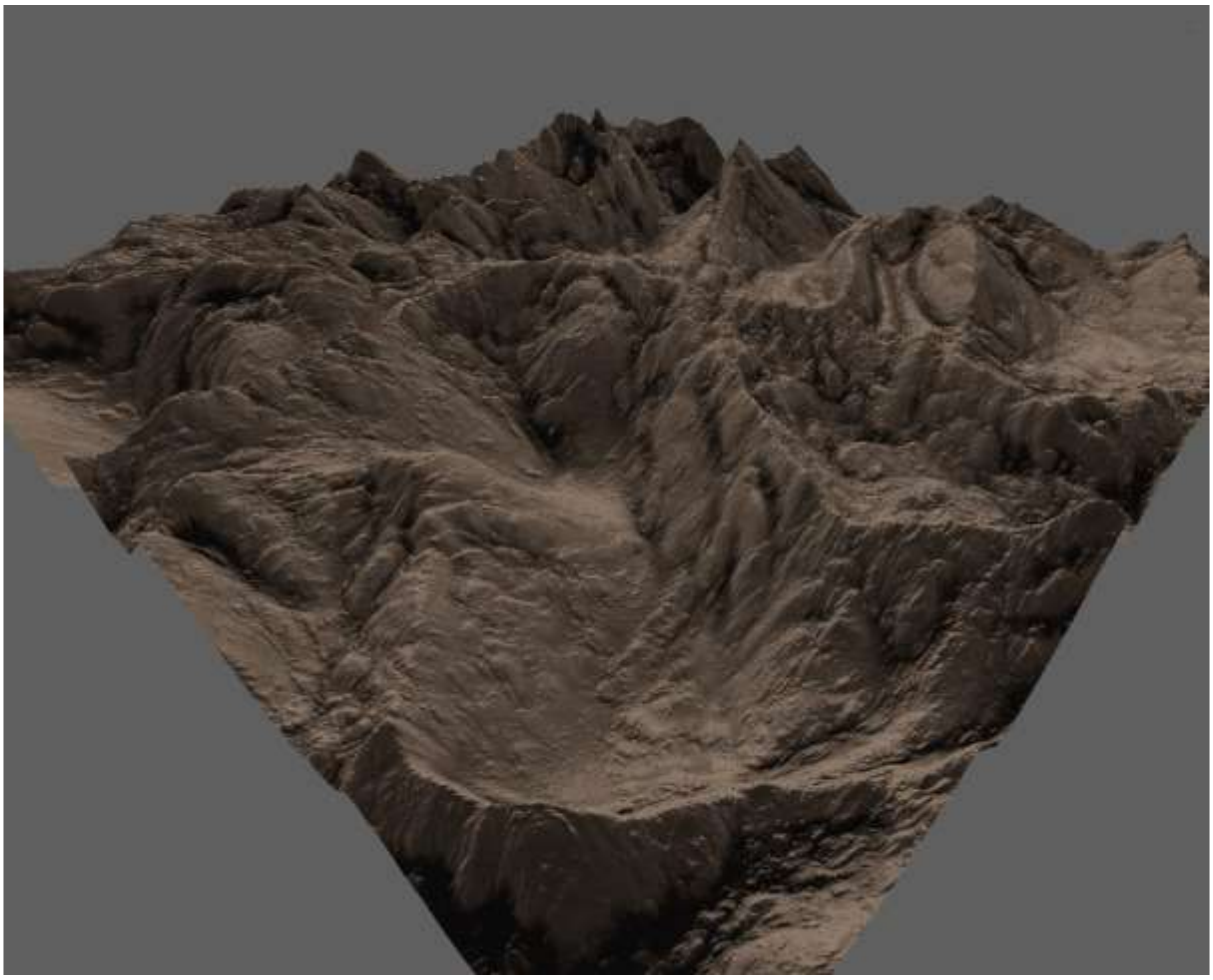
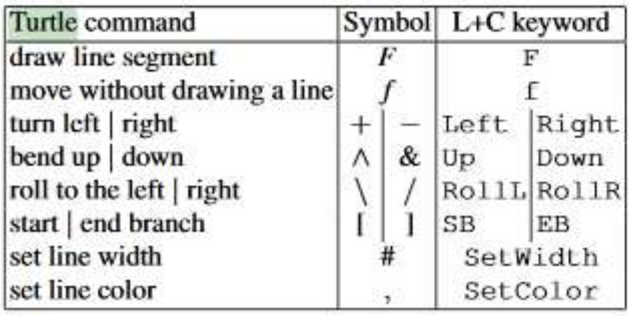


Figure 18: Terrain generated using multifractal noise [31]

2.2.4. L-System

L-System algorithm is an ontogenetical algorithm which is often used in procedural plants modeling [32]. In general L-system is an shortcut from Lindenmayer system and it is a set of rules allowing user to generate fractal-like creations, algorithm uses provided rules in order to generate set of instructions to reach the designated goal. The L-systems to generate plants are called Branching L-Systems. Authors of the article are writing about L-system with turtle interpretation, to visualize the concept:



Turtle command	Symbol	L+C keyword
draw line segment	F	F
move without drawing a line	f	f
turn left | right	+ | −	Left | Right
bend up | down	∧ | &	Up | Down
roll to the left | right	\ | /	RollL | RollR
start | end branch	[|]	SB | EB
set line width	#	SetWidth
set line color	,	SetColor

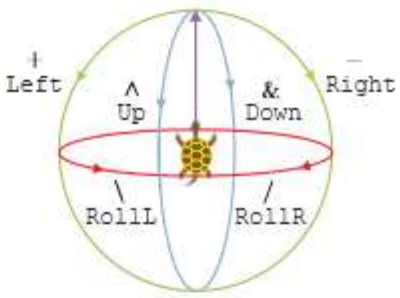


Figure 19: L-System with turtle interpretation [32]

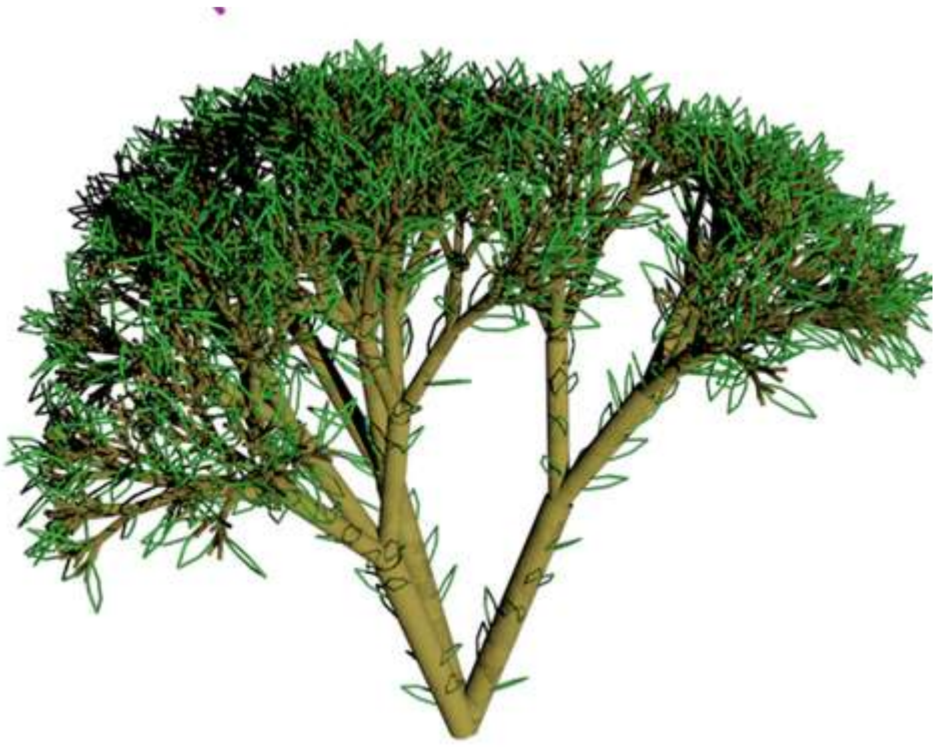


Figure 20: Plant generated with L-System branching algorithm [33]

In [30] authors used L-System algorithms to create entire network of roads for the purpose of their survey, it visualizes how many possibilities and ways to model generated terrain are available when described algorithms are used combined.

2.2.5. Midpoint-displacement algorithm

Midpoint displacement algorithm goal is to generate height map [34] based on a rectangle surface. It assigns certain height value to all 4 corners. In the next step, initial rectangle is divided to four smaller rectangles and do its corners (of the smaller rectangles) mean height value of the parent rectangle corners.

The algorithm is referenced in [13] in fact authors implemented diamond-square algorithm, however this is clearly marked that diamond-square algorithm is simply upgraded and therefore faster version od midpoint-displacement algorithm. In next chapter there will be visualized of how diamond-square algorithm works.
Archer in his publication [35] referenced to midpoint-displacement algorithm as one of many other algorithms tested in his work, among others there are already mentioned diamond-square algorithm, value noise or Perlin noise as well.
Midpoint displacement algorithm is with the Perlin noise and diamond-square algorithm one of most frequently cited algorithms.
Below there is an example of generated terrain:

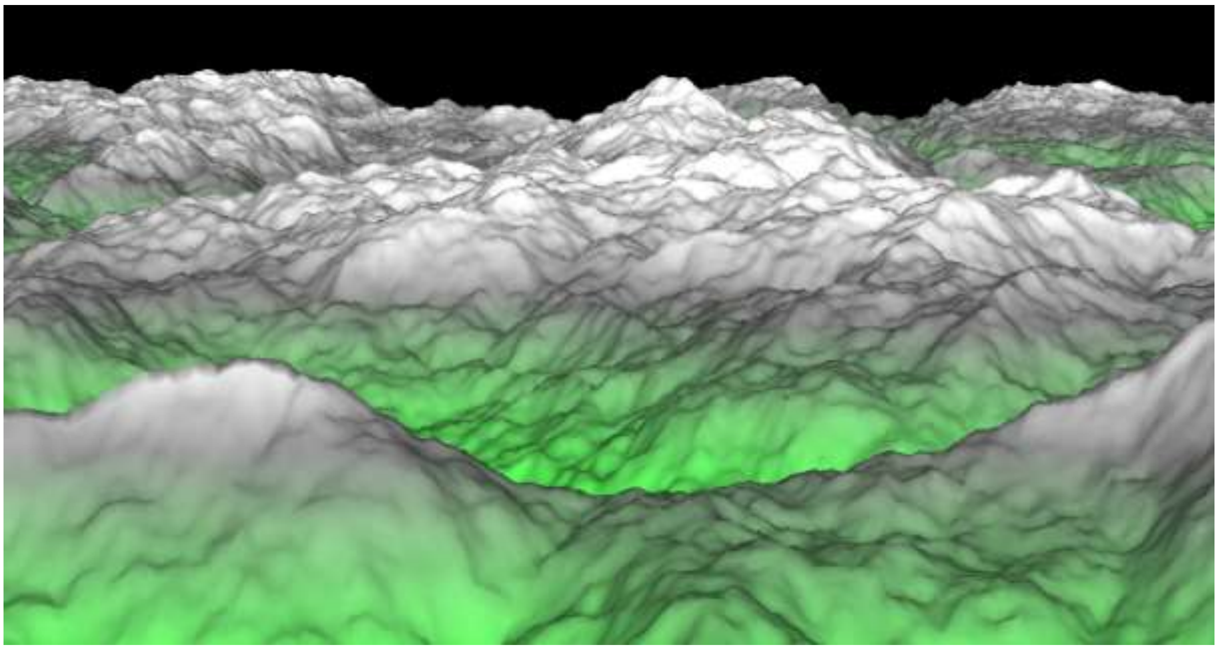


Figure 21: Terrain generated using midpoint displacement algorithm [36]

2.2.6. Diamond-Square algorithm

In reference to Midpoint-displacement algorithm there is Diamond-square algorithm, it is better version of previous algorithm, also known as random-midpoint displacement fractal, cloud fractal or plasma-fractal.

It works similar to MDS, initially starts in four corners of a rectangle [37], in the next step calculate an average value, based on the values in the corners, and it is called a square-step.

After that, it is time for the diamond step where for each square on the array we have to set the middle point of that diamond shape as the average of four cornets of the diamond.

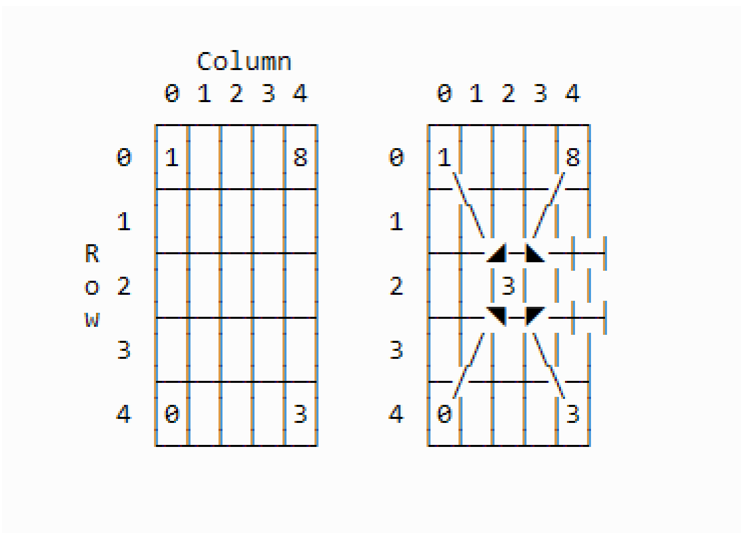


Figure 22: Square step in diamond-square algorithm [37]

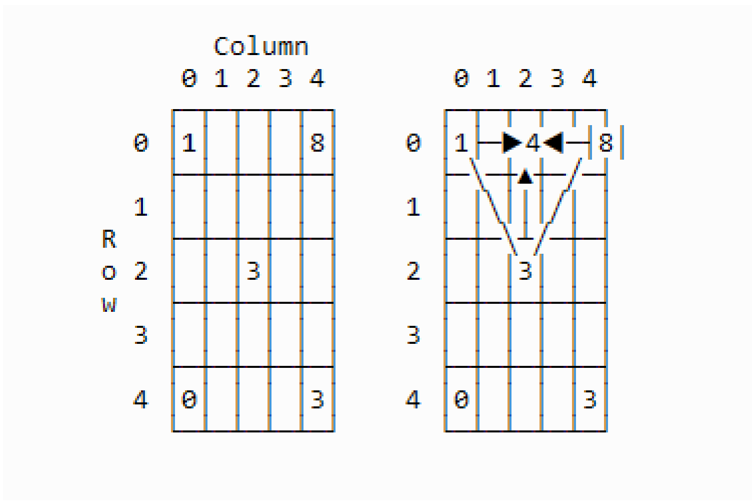


Figure 23: Diamond step in diamond-square algorithm [37]

As mentioned diamond-square is slightly improved version od midpoint displacement and it often appears in publication next to the previously described algorithm, for example in [13], [35] are referenced in comparison

2.2.7. Voronoi diagram

Voronoi diagram is an algorithm, initially randomly chooses n points localized on some surface, next the surface is divided for n areas so each point is in only one area [38] the important thing in this division is that no other point in any other area is closer to any point in other areas. In other words point a in area A can be faster anywhere in area A borders than any other point. Voronoi diagrams can be used to generate maps division, or to generate some sharp-edge shapes, sea or land. In games Voronoi diagrams can be used for example to properly displace resources for players, it is not limited only to terrain issues.

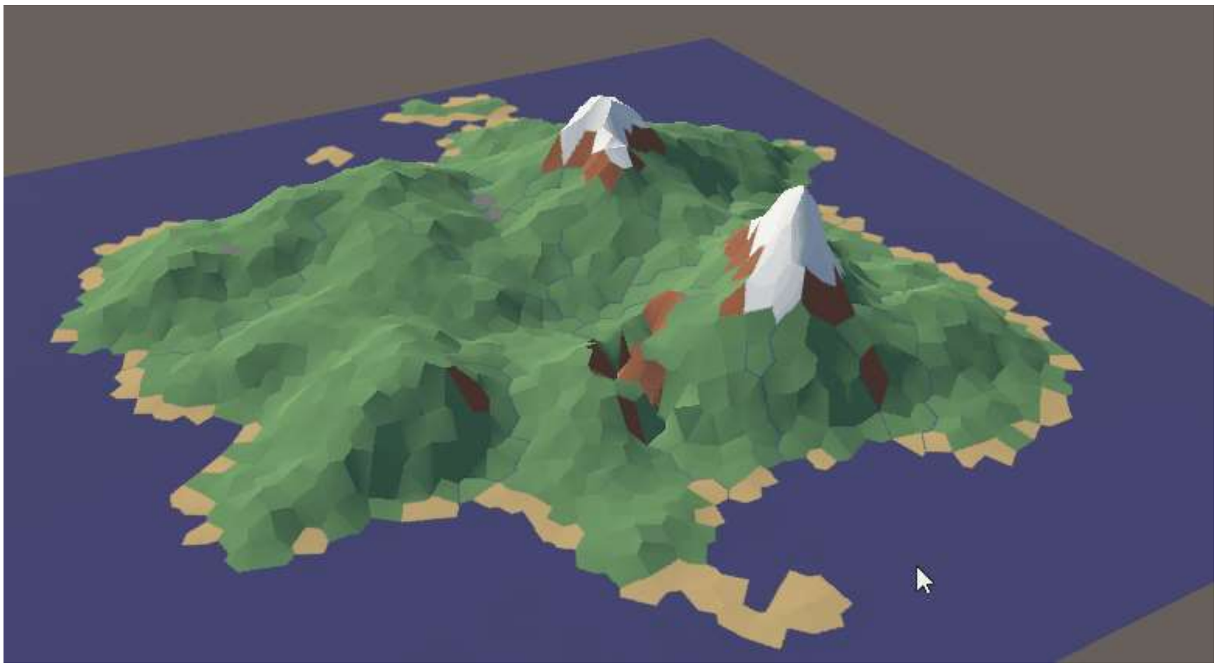


Figure 24: Sample map generated using Voronoi diagram [39]

The algorithm implementation itself does not have to generate Voronoi diagram in one step, and id usually does not [40] it is often implemented in a way where initially our points are the nodes of a graph and then the graph is converted to Voronoi diagram.

Voronoi diagrams are widely used in various publications [13], Olsen used Voronoi diagrams in cooperation with noise functions to investigate different erosion methods. In other work diagrams were used in order to procedurally generate cave systems [41].

3. Review of other literature

To perform a proper review of literature I have looked among major journals, IEEE (Institute of Electrical and Electronics Engineers) and Springer – International publishing, and others. To search for appropriate materials I have used search engine of electronical resources provided by my university – Wroclaw university of science and technology, and search tools directly on sites of IEEE and springer.

I have made a few assumptions about how to count some positions as relevant, to first filter all totally not related positions, and later to filter out all positions less related.

First the title must contain at least some of the words: noise algorithms, noise, procedural terrain, procedural generation or other similar combinations of these formulas.

Not all references refer to the material I have found through the site directly, some of them are placed on different sites, because of availability.

3.1. Review of similar research

There are number of publications about procedural terrain generation. Some of them involves noise functions as the main part of the research.

Hyttinen [42] work mentions multiple noise algorithms such as Perlin, Simplex, Gabor, and few more. In chapter 3.6 entitled “Comparison of the noise functions”, author does not mention any numeric results about actual performance of different noise algorithms while generating terrain. Mostly publication if focused around usability of particular noises, it is mentioned that noise functions are widely used in game industry to generate large areas, and realistic worlds when properly combined with other procedures and algorithms.

Hyttinen points that procedural noises are very important and base element of procedural techniques, and generating complex terrain, and it thanks to noise algorithm it is possible to generate infinite worlds. Good example of such usage is well known game entitled “No Man’s Sky”. The game was released in 2016, and it is a world which is build from almost 18 Q planets with procedurally generated terrain [42].

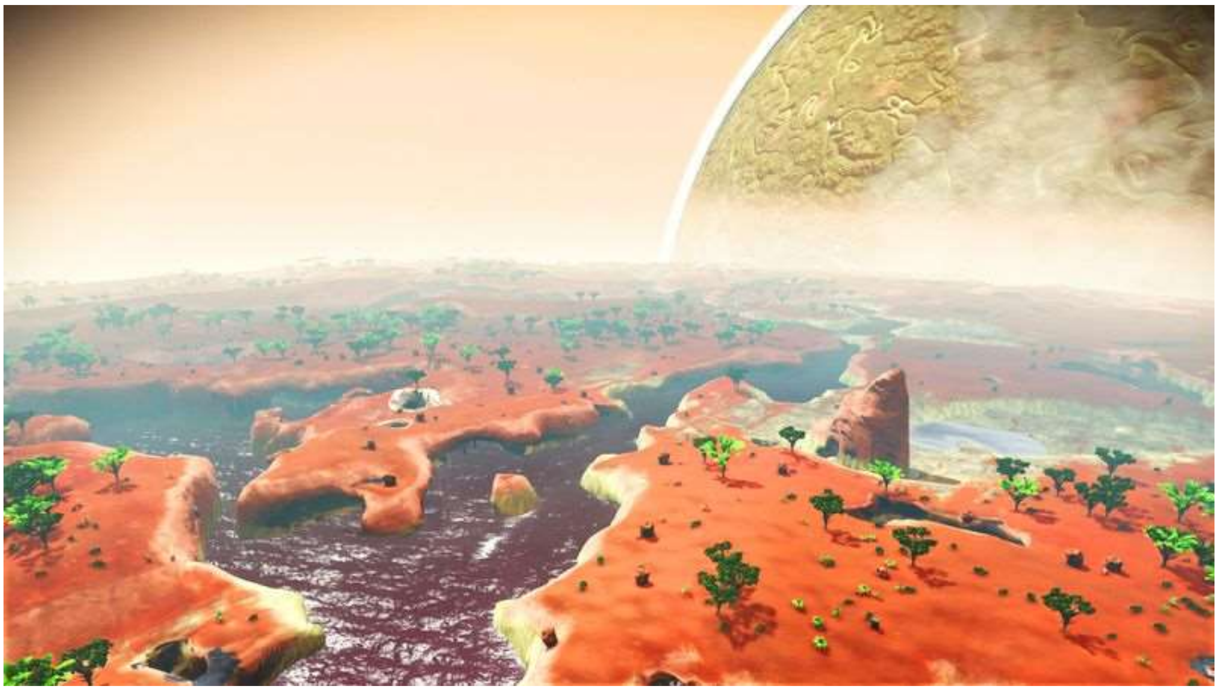


Figure 25: No man's sky terrain example [43]

Publication does not provide proper comparison of mentioned noise algorithms, therefore it is direction worth of exploring, as noises are the very base of procedural terrain generation in many approaches. Based on the criteria accepted in this chapter publication can deliver useful information about noises and different techniques of terrain generation, which can be helpful to extend the view at the noise algorithms and provides information about how important noise algorithms are. Based on this research it is possible to say that noises are the fundamental element for terrain generation in.

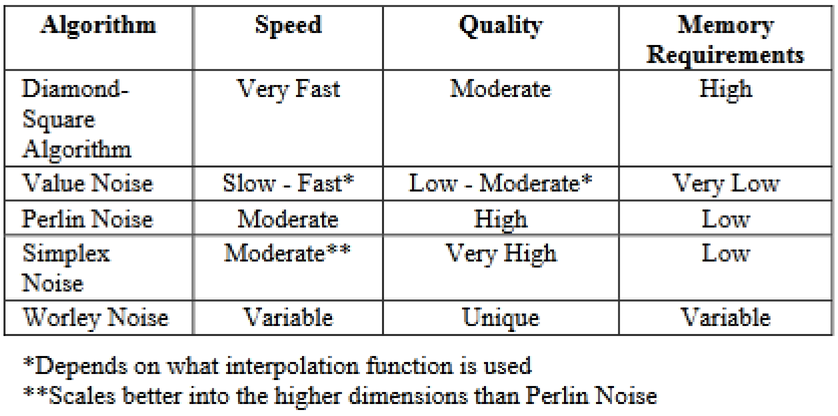
Thorimbert and Chopard in their work [23] are testing specific noise algorithms, Perlin noise is one of them. Study is focused around texture generation methods which allow to generate 3D terrain which is close to research that will be carried out in this study. Authors are stating that their new method is significantly faster than other methods based on fractal Brownian [23] approach. Mentioned work is relatively close to performance study, however it is based on other terms and it does not show the results in the way in which it will be easy to understand, this work is a valuable source of information about Perlin noise and because simplex noise and ridged – fractal noise are pretty close to Perlin noise it can deliver important information for analysis of the conclusions after the experiment.

By looking at the result of the tests that were run by Thorimbert and Chopard we can distinguish two compared methods one using Perlin and second approach called D2M1N3. This research however does not give us the results in a straight way, and work is mainly focused on showing differences between this new approach and others. Therefore there is an existing gap that could be filled with proper research comparing not only noises between each other but also looking at the performance and time indicators from perspective of two different engines.

Spiridis master thesis [44] treats about procedural generation with special care to heightmap generation which means indirectly that attention of this work is can be gathered around noise algorithms. Thesis contains a number of benchmarks, in chapter 10.1 entitled *heightmap* *generation benchmark* and in another chapter 10.2 *Real-time heightmap generation benchmark.* Both are more based on heightmap size and are using only one noise implementation as mentioned 4D Perlin noise, by Ian McEwan, therefore we do not have clear vision of noise algorithm performance itself as I would like to show it in current work.

Multiple authors in “*A Survey of Procedural Noise Functions”* [45] are introducing a research about multiple noise functions, among all listed in chapter 3 are gradient noises, Gabor noise, spot noise, Perlin noise, and many more. Authors are discussing variety of important noise-algorithm related topics such as studying noises functioning on surfaces, filtering. In chapter 6.2 [45] researchers presented analysis results. Comparison does not cover performance testing, meaning time needed to generate simple part of terrain, CPU usage as well as memory consumption, in terrain generation using chosen noises.

“Algorithms and Approaches for Procedural Terrain Generation”, is a good source and a brief comparison of noise algorithms [27]. This work can be good point of reference in interpretation of the experiment results. Number of noise algorithms were tested and compared, and evaluated for speed, quality, memory requirements – results were presented in the table:



Algorithm	Speed	Quality	Memory Requirements
Diamond-Square Algorithm	Very Fast	Moderate	High
Value Noise	Slow - Fast*	Low - Moderate*	Very Low
Perlin Noise	Moderate	High	Low
Simplex Noise	Moderate**	Very High	Low
Worley Noise	Variable	Unique	Variable

*Depends on what interpolation function is used
**Scales better into the higher dimensions than Perlin Noise

Figure 26: Results of noise algorithms comparison [27]

All referenced articles and scientific thesis are related to the topic in some direct or indirect way, all works are raising the topic of procedural terrain generation at least or in addition noise algorithms. Every mentioned work was accepted according to assumptions that was made on the beginning of this review, and is considered relevant to the research being carried out.

Noise algorithms were an object of study of many works not mentioned here, it would be impossible to reference to them all. I believe that this work is unique nonetheless thanks to different approach to the issue raised. It is well known that many different tools are used by developers and graphic designers all around the world, and two chosen engines are just a drop in the ocean of possible research in this area. I believe that performance in such matter as live terrain generation is something worth exploring considering wide use of this technology.

In following chapters noise algorithms will be described in detail – chosen noise algorithms which will be tested in experiment part.

3.2. Detailed description of chosen algorithms

In this chapter we will have a closer look at noise algorithms which will be tested later on. I have chosen following algorithms due to their popularity I have estimated during my review of similar work and research. Among many algorithms described in previous chapters following were selected:

- Perlin noise
- Simplex Noise
- Voronoi diagram
- Ridged-Multifractal noise

All of the noises distinguished above, were already introduced in previous chapters
As mentioned before this study will be run on two game engines, which means there will be eight implementations of these noises in total, used in two implementations, one for each engine. More details about implementation alone will be described in following chapters.

3.2.1. Perlin noise

Most important feature and also well known fact is that it gives very natural results in case of generated terrain. The reason why Perlin noise looks natural is that it generates smooth pseudo-random number sequences, there are no rapid changes in adjacent numbers therefore when we would look at the numbers generated used Perlin noise we would receive pretty good looking mountain-like graph [46]. Have a look at the comparison between numbers generated by Perlin noise and non-Perlin pseudo random numbers generator.

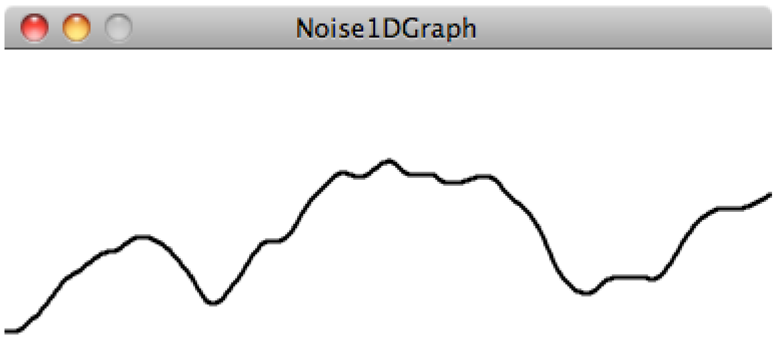


Figure 27: Perlin noise generated number sequence [46]

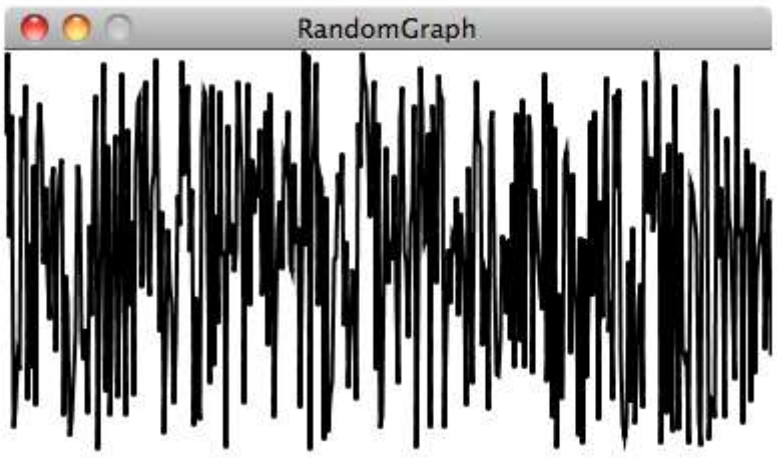


Figure 28: Pseudo-random generated number sequence [46]

Technical aspect of how Perlin noise works was well described already in previous chapters, therefore it would not be repeated in this part.

Complexity of this algorithm scales around:

$$O(2^n)$$

Where n is number of dimensions. It is similar complexity to simplex noise, which will be shown in further part. It is reliant on the needs, but most frequently Perlin noise is implemented for 3 or 4 dimensions. For the purposes of this work, a 3D implementation of Perlin noise will be used, there will be two implementations as mentioned, one for Unity engine and the second one for Godot, in chapter dedicated to implementation there will be more detailed description about the source code and functionality from code perspective.

3.2.2. Simplex noise

For this research simplex noise was chosen because of its similarity to Perlin noise. It is expected for simplex noise to be faster. Once again have let us have a look at simplex noise complexity:

$$O(N^2)$$

Differences between simplex and Perlin noise were introduced in chapter 2.2 – to be more specific the difference about working on triangles in case of simplex noise and grid in case of Perlin noise. It is worth mentioning that the difference is more visible in higher dimensions, it will be verified in test part to compare with all other noises, but in case of our tests – complexity mentioned above will not affect the final result in much visible way. If N is number of dimensions then looking just at the complexity it is in fact probable that Perlin noise will be a bit faster in 3D terrain generation. There is one more difference mentioned [25], [26], in comparison to Perlin noise "grid" simplex noise is based on simplices which are in simple worlds n-dimensional triangles, each dimension-specific triangle representation shall have its own name, for example 2D is a triangle.

Simplices creates a triangle based grid on the contrary to grid used in Perlin noise, hypercube in n dimensions will have 2^n corners while simplice will have only $n + 1$.
Due to mentioned differences there are obviously differences in implementation, which will be presented in following chapters along with implementation of other noises, both for Godot engine and Unity.

3.2.3. Voronoi diagram

Voronoi diagram is as mentioned before, method to divide certain area to small areas based on given points, one in each area, it is often call “Voronoi decomposition’ due to that fact [38]. Generating Voronoi diagram can be described in following way.

First a group of points (feature points) is chosen randomly, and placed on the surface we want to generate as final result. In next step each texture part (texel) value is set based on the distance to the closest feature point, there might be a number of ways to calculate value of a texel but this is the simplest way. Based on the texel values a final height map is generated and therefore the terrain as well.

Voronoi diagrams are not an algorithms directly, it is a final product that can be used in procedural terrain generation. There are a few algorithms dedicated to generate such diagrams. Most known one is fortune algorithm and it is dedicated only to generate Voronoi diagrams. [47].

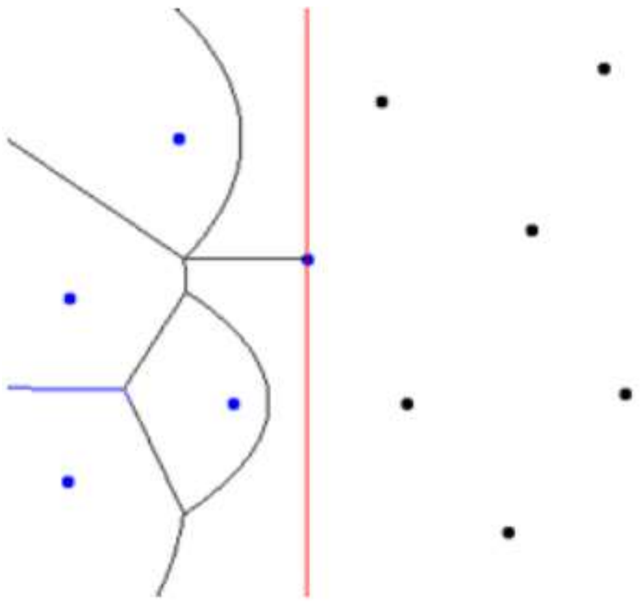


Figure 29: Fortune algorithm work progress in generating Voronoi diagram [47]

Fortune algorithm complexity is:

$$O(n \log n)$$

Where n is number of points. It is worth mentioning that terrain generated using Voronoi diagrams looks very specific and of course it is possible to edit it further, in this study we will only generate most basic terrain only for academic purposes, but it can be more efficient to use different approach to get more natural result without initial changes and still save some more time.

3.2.4. Ridged multifractal noise

Ridged-multifractal noise, works similar to other noise-based methods. As mentioned before there is one major difference. The ridged noise returns absolute value in case of implementations used, of Perlin noise. The procedure is as follows, values in range from -1 to 1 are generated by noise algorithm, then absolute value is extracted and the final step includes multiplication by -1 [48].

Due to common parts with Perlin noise complexity should be similar:

$$O(2^n)$$

3.3. Review summary and research problem

There are similar researches in the literature, some of them are closely related to the goal of this thesis and will be helpful comparison in the research and conclusions part. As mentioned however, there is a gap that can be filled up with current research. The goal is to have the data and research results indicating which noise will be better considering chosen indicators, therefore in this thesis chosen algorithms will be reviewed in two most popular engines (game engines), in order to determine if there is significant difference not only between noises itself within one implementation, meaning implementation in one engine, but also in multiple engines. Significant difference in statistical meaning, between indicators described in further chapters. This require similar implementations in two chosen engines, and corresponding noise algorithms implementations.

4. Introduction to methodology and research

4.1. Tools and implementation

4.1.1. Godot

Godot Engine, is an open source project, it can work as engine for 2D and 3D games [49]. Godot implementation gives us a lot of prepared tools, engine itself is made of modules which are visible when we have access to raw engine code accessible on GitHub – it is possible to implement some modules on our own and then adjust modification to Godot standard. This way we can customize the engine for our purposes.

Engine allows us to implement games for all major platforms, Linux, Windows, macOS, Android, iOS, HTML5, whole project is available on MIT license. Godot support four languages at the moment, and these languages are

- Godot script (GDScript)
- Visual Script
- C#
- C++

GDScript is recommended language to implement logic in Godot, it has the best support and documentation in engine itself, therefore it is easy to work with it.
Godot script is very well integrated with C++, most of the modules is written in C++ - that was one of the reasons why this is the engine chosen for the purpose of this research.

Visual scripting is not really a programming language, it is programming based on visual interaction with the workspace. User has properly designed interface at his disposal in order to reach the programming goals and desired functionality. Mentioned features make visual scripting pretty easy to understand especially for less advanced game developers, or not developers at all, but for example designers. It can be also good tool to prepare some demo-game to be later explained for the wider and diverse public.

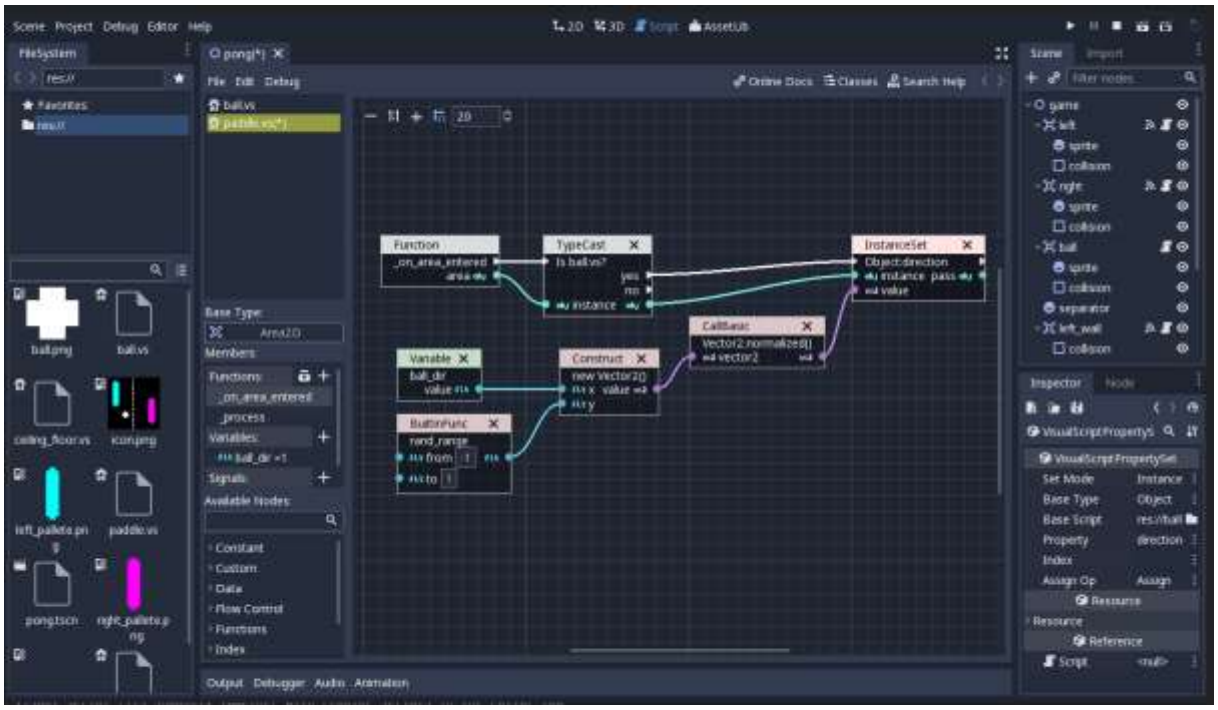


Figure 30: Visual script godot [49]

C# in godot is hardly comparable to GDScript, it is much less user-friendly, and harder to work with at its current level of advancement in the development environment (20.04.2020). It is much easier to use GDScript, however C# has all the functions and modules available as well.

4.1.2. Unity

Unity is a multiplatform game engine, it has very extensive offer of assets. Mostly paid assets however there are a few pre-prepared asset packages to use for beginning developers. This program is free to use as long as income does not exceed 100.000 USD per year. In Unity it is possible to develop software for platforms:

- Android
- Windows
- Linux
- Tizen
- WebGL
- Play station
- Others..

It is also possible to create games and other software for virtual reality. Currently primary development language in unity is C#, development environment itself is very intuitive and simple, many elements work on the basis of drag and drop.

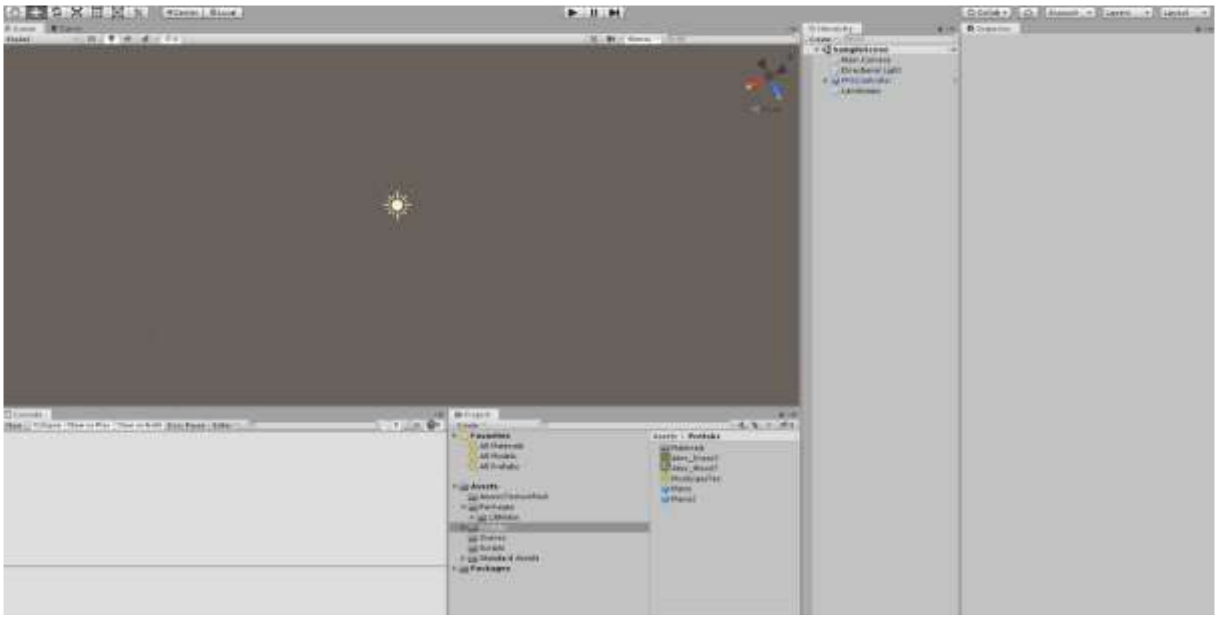


Figure 31: Unity environment main screen

Unity is also most frequently used game engine according to gamedesigning [50]. According to the same ranking Godot is no 4th place. There are two worth mentioning possibilities to write code, as I noticed before, C# is the main language for Unity, therefore Visual studio is obvious choice to implement scripts, VS is also more helpful for beginners despite that it is one of the biggest code environment, there is of course an alternative to Visual studio build in Unity engine itself, there is an environment called Mono-develop one of the most basic tools simply to write code. It is important to mention, that before we will be able to write code or debug with Visual studio it is necessary to install proper extension to our environment, it is simple as build in VS installer gives the chance to install it through user-friendly interface. Therefore it is safe to say then, that tests will be run on most frequently used game engines.

4.1.3. Monitoring resources and time

There will be two tools used for the purpose of measuring results. First there will be time in microseconds [μs] measured directly from the code. In implementation there will be a part of code responsible for generating one chunk, based on a plane mesh with certain amount of vertices, vertices amount will be the same for both Unity engine and Godot..
After each chunk is generated with noise, the time will be saved and stored for further usage.

Second there will be process monitor software, free Microsoft tool of windows sys-internals [51]. It is possible there to register process and thread activity to view it later, using process monitor it is possible to view CPU and memory usage in each second of monitoring time. Below there are two screenshots showing the list of processes available to choose and second detailed view of measures possible to preview.

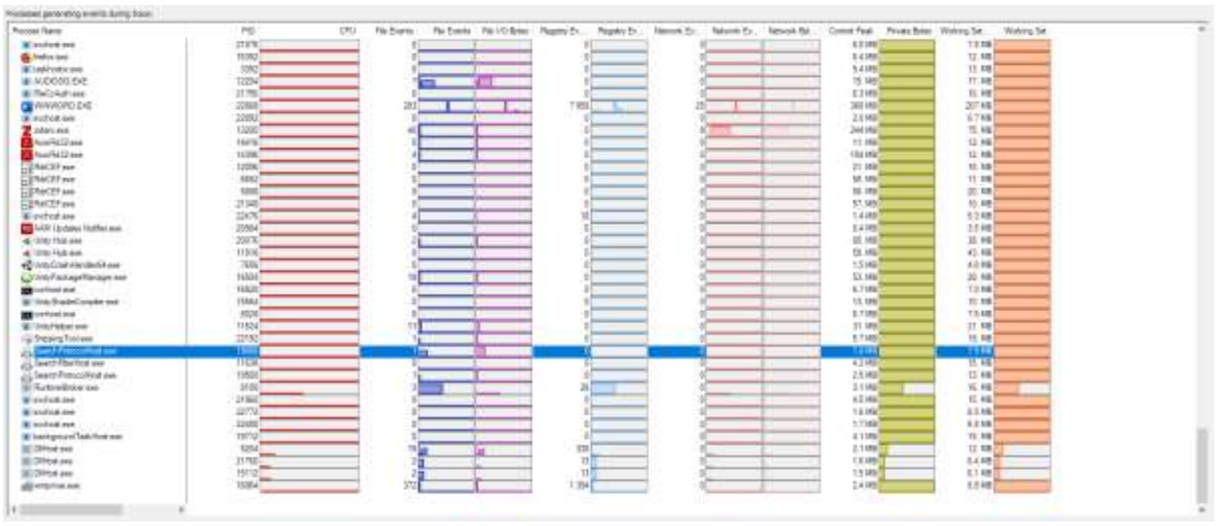


Figure 32: Process list in procss monitor

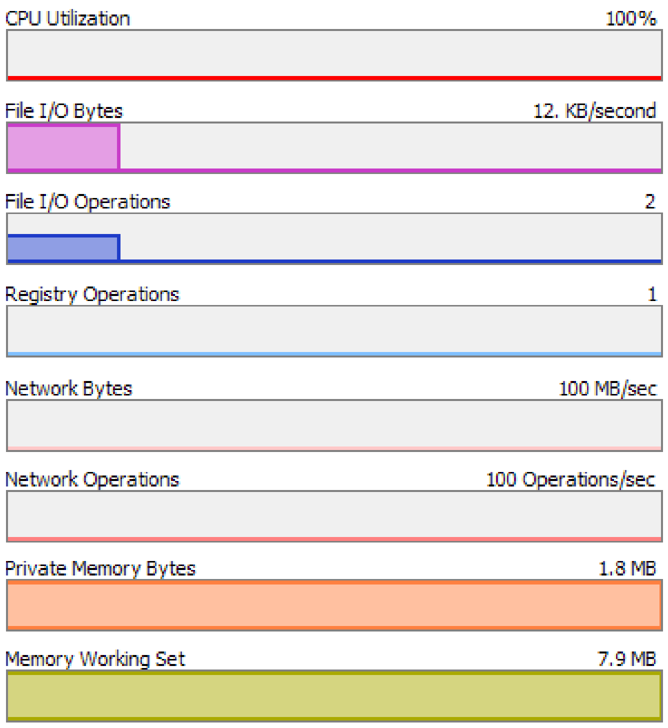


Figure 33: Process monitor window with process details

4.1.4. Unity noises library

Lib noise library is available on GitHub [52] originally implemented by Jason Bevins, it offers multiple noises free to use in unity implementations. Available noises are:

- Ridged multifractal
- Voronoi
- Perlin
- Cylindric
- Spherical
- Billow
- Checker

What is more to compare simplex noise between Unity and Godot engine, simplex noise implementation is based on implementation used in another work including procedural terrain generation [53].

Whole library is implemented using C# language since it is designed for Unity. It is not confirmed if that implementation is optimal, it is possible that some of the algorithms could be rewritten with positive impact on performance, however we accept that fact as a part of certain error threshold, and if the difference will be statistically significant but the numbers would be close for both engines, it would be a good direction for further research, to try improve performance of some chosen algorithms in order to run some more specific and detailed tests in more narrowed area.

4.1.5. Godot noises library

Fast noise library is also available on GitHub [54], whole library is implemented in C++ and it is prepared to integrate with Godot Engine, however it is necessary to compile Godot on your own. Noises offered by fast noise library are:

- Perlin
- Simplex
- Ridged multifractal
- Cubic
- Voronoi

4.1.6. Research implementation for Unity

Unity implementation is based on LibNoise library available on GitHub. Analogical to Godot implementation, terrain generated is in its simplest form in order to get rid of all unnecessary interference which may cause differences in research when monitoring chunk generation time and overall performance. Simplest implementation means that the terrain is just nothing more than a land shaped with the height map generated with tested noise. The terrain generation is a huge project with a lot of elements, noises are used all around them in order to generate randomness to create natural-looking parts of designed terrain, such as trees, rivers, further shaping of mountains, roads. Noise algorithms are inseparable part of that process, and the purpose of this thesis is to verify if time impact, resources consumed are huge, what is the difference between two game engines in similar use, and corresponding noises. Therefore that implementation should be good enough to meet the requirements.

4.1.7. Research Implementation for Godot

Godot implementation uses fast noises library, described in previous chapter. Environment generated by both implementations is the simplest possible result, in order to test actual influence of used noise on the time and performance, in procedural terrain generation height map generation is kind of constant part therefore I did not want to make it possible for other disturbances such as other environment elements – plants, stones, trees, water, water movement, grass, any kind of surface graphics to influence result of the experiment. Therefore terrain generated by my program is really "raw" and that is exactly what was expected to happen. On the screen below there is example of the terrain generated using Perlin noise. Implemented program works based on the player position, chunks are generated around the player while he moves, initially there is set number of chunks generated. With the movement of the player, chunks that are furthest from us are deleted and new chunks are generated in the direction of movement.

4.1.8. Generated terrain examples

In this chapter there will be examples of the surface, generated with each noise. In both implementations terrain looks similar.

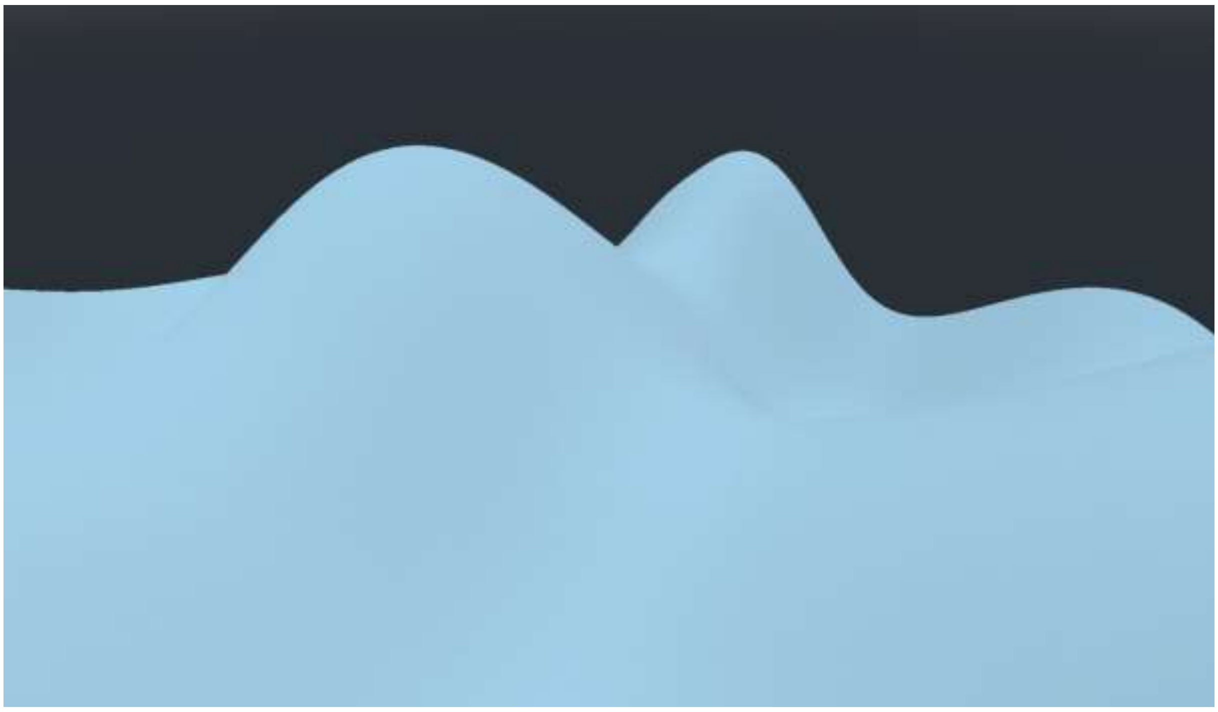


Figure 34: Terrain generated using Perlin noise

Terrain generated with Perlin noise seems to be most smooth, and there are no visible edges on the plane mesh.

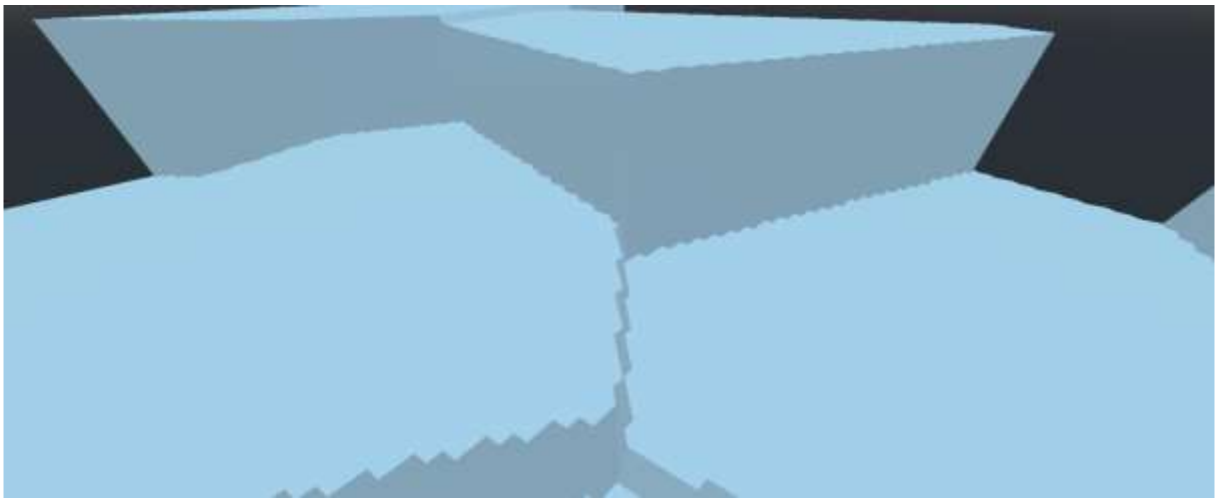


Figure 35: Terrain generated using Voronoi

Terrain generated with Voronoi is the most exotic one, the look of it is pretty similar to honeycomb structure.

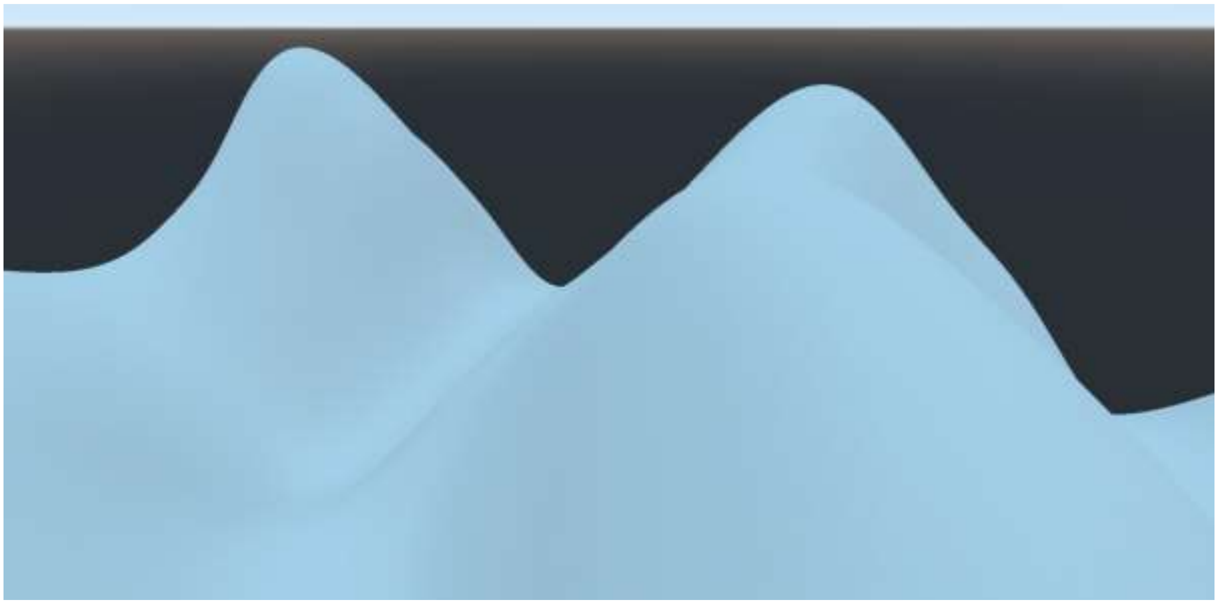


Figure 36: Terrain generated using ridged-multifractal (fractal) noise

Fractal noise, is very similar to Perlin noise presented on first figure in this chapter, edges are also pretty smooth and there are no visible edges.

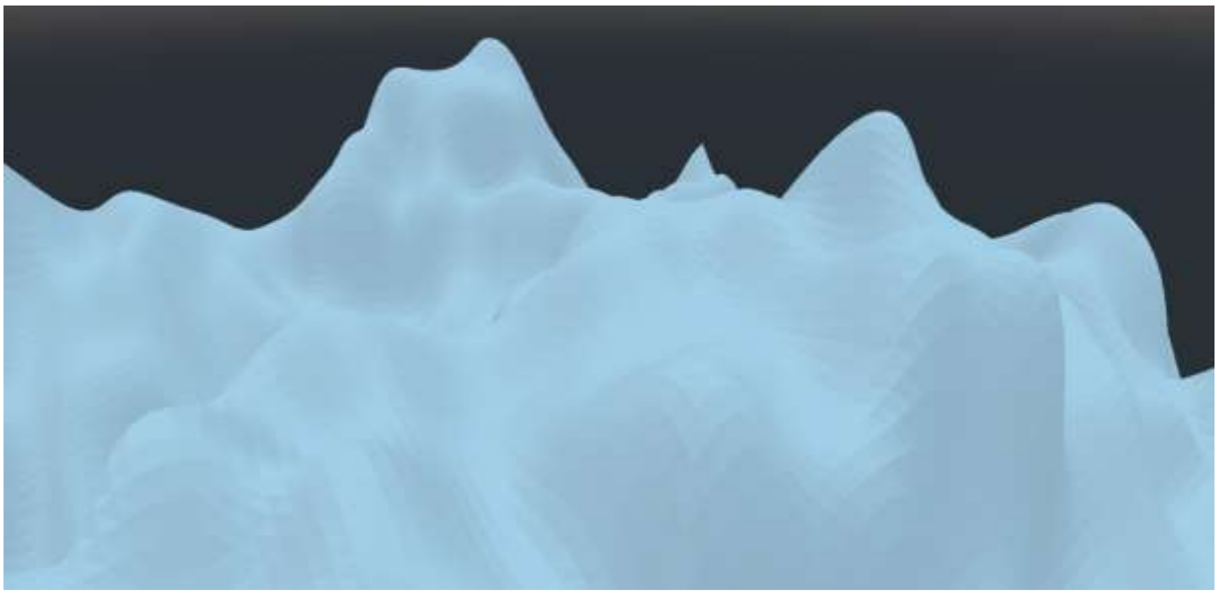


Figure 37: Terrain generated using simplex noise

The last noise example is simplex noise, as the example is from Godot engine it is Open-Simplex noise due to that Simplex noise is patented. In this case there is visible grid on the plane mesh, however overall structure is similar to Perlin and fractal noise.

4.2. **Research procedure and tests**

4.2.1. Research introduction

In this chapter research procedure will be described, at the beginning indicators will be described, how it would be determined if one algorithm is fast or more efficient than the other. Next subchapter will contain information about single research run, which mean single test of one algorithm, it is planned to run three tests for each algorithm therefore there will be three sets of data, after that results will be analyzed with statistic tools.

4.2.2. Research platform

For the purposes of this thesis I use test platform in available in me personal resources which include following components:

- CPU: Intel I5-7400
- GPU: NVIDIA GeForce GTX 1060
- 8 GB ram 2400 MHz

Research platform is one of the most significant elements of that test, it is essential element of each computing task. Many of disturbances revealed in the process might be coming and be caused by the test platform, it would be ideal to run similar tests on multiple platforms to verify what is the significance of platform, and how it would change the final results.

4.2.3. Studied indicators

As mentioned in the introduction of the thesis and later in previous chapters, there will be few indicators, determinants based on which algorithms performance will be measured:

- Chunk generation time in microseconds
- CPU usage in each second of the test
- Memory usage in each second of the test

Above measures will be saved and verified using programs described in chapter 4.1.3.
It is noticeable that no FPS (frames per seconds) are measured, it is planned action because noise algorithm works before rendering phase which uses GPU therefore frames per seconds would not show us the impact on the time in which algorithm works.
It is expected that most meaningful indicator here will be the time, after all eventually it is the final and most direct indicator of performance, and also the thing we care about most. If algorithm consumes more CPU power it is fine as long as we have power available, same thing applies to memory usage, if we have memory available it is free to use as long as we manage it properly in our implementation, for example do not store terrain that is too far away from the player in the memory. And so time is the thing that is most crucial, because we can afford CPU power and memory usage, but can not afford to lose time.

4.2.4. Research run description

Single research run steps will be as follows:

- Check if initial conditions are met
- Run the program with desired noise algorithm
- Move player with constant speed for 30 seconds in order to generate new chunks
- Stop player movement
- Save time results
- Save CPU usage results
- Save memory usage results
- Bring back test platform to initial conditions

Initial conditions include neutral CPU usage, and neutral memory usage in state of inactivity. Which exactly means that programming environment (Unity or Godot) do not use more CPU or memory that it is used in idle state.

In discussed cases chunk will be an object, plane mesh, with 120 vertexes, for each vertex there will be calculated noise value. Above procedure will be performed three times for each tested noise.

4.3. Research results and analysis

4.3.1. Research goals

First there is performance matter to verify, this appears between algorithms at first, but it is expected that the difference between algorithms within one implementation or one game engine will work in similar time or at least without any statistically significant difference in time, CPU usage or memory usage. It is expected however that different engines – in this case Godot and Unity will show some differences, it is expected that implementation used in Godot engine, fast noise library will be noticeably faster that noise library imported to unity due to C++ implementation in Godot. Most significant determinant here will be time obviously because that is eventually what we about when generating terrain, it better if algorithm use more CPU computing power if available and finish faster, than compute less and finish later the same rule applies to memory – of course the power should be consumed as long as there is available computing power and memory.
From that other assumptions come out, it is expected that more power-consuming noise algorithms will generate chunks faster.

4.3.2. Godot research

For Godot engine tests all gathered results are presented in **Attachment A**, under **Godot engine research charts**, in next chapter summarized results are presented on collective charts, with moving average as well.

4.3.3. Godot – summarized results

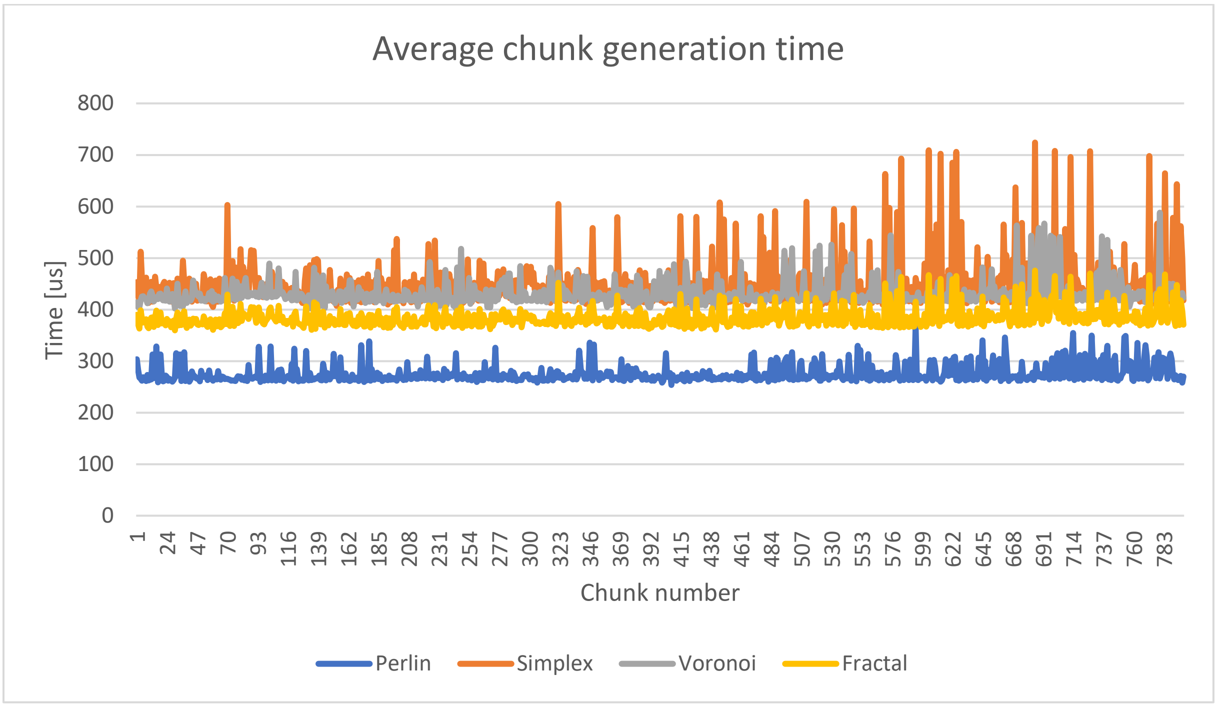


Figure 38: Average chunk generation time Godot – summarized

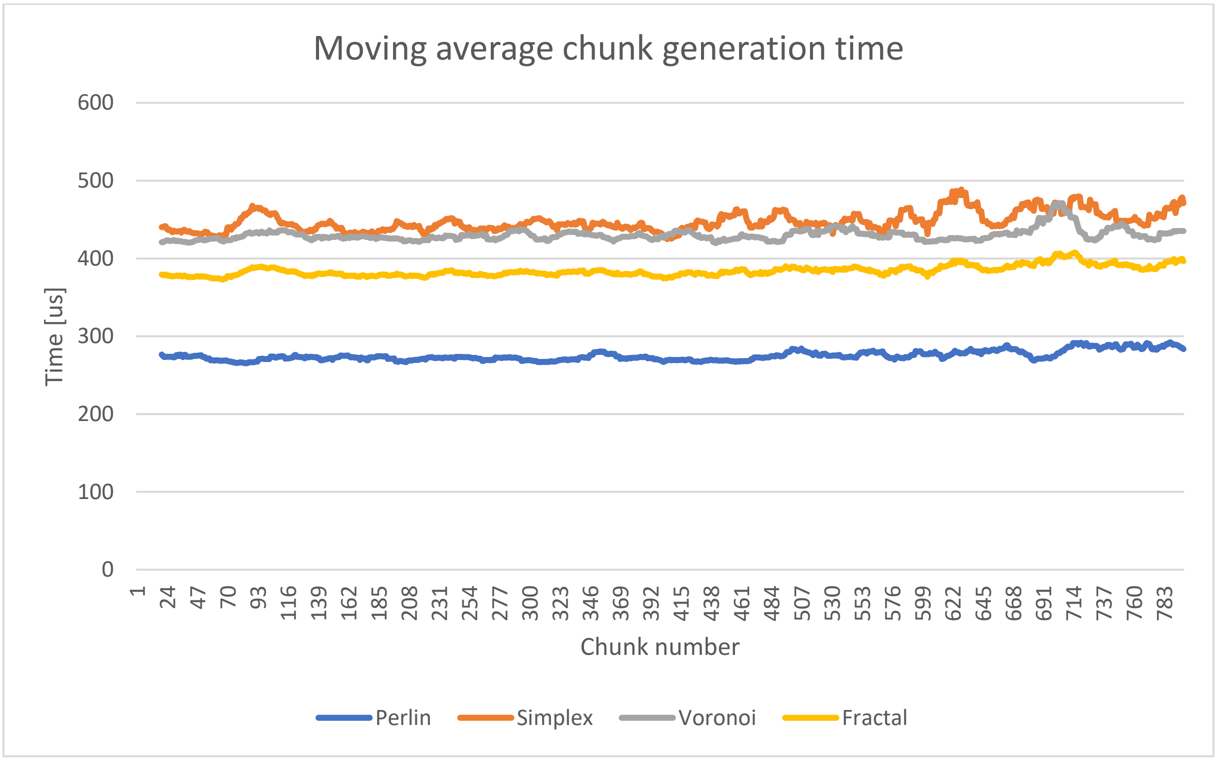


Figure 39: Moving average chunk generation time in Godot

Moving average for Godot unity confirms, and more clearly shows what is already visible on normal average chart. All tested noises showing common trend to work stable, simplex, Voronoi and fractal noises are working in similar time. Differences between noises will be further tested using statistical analysis.

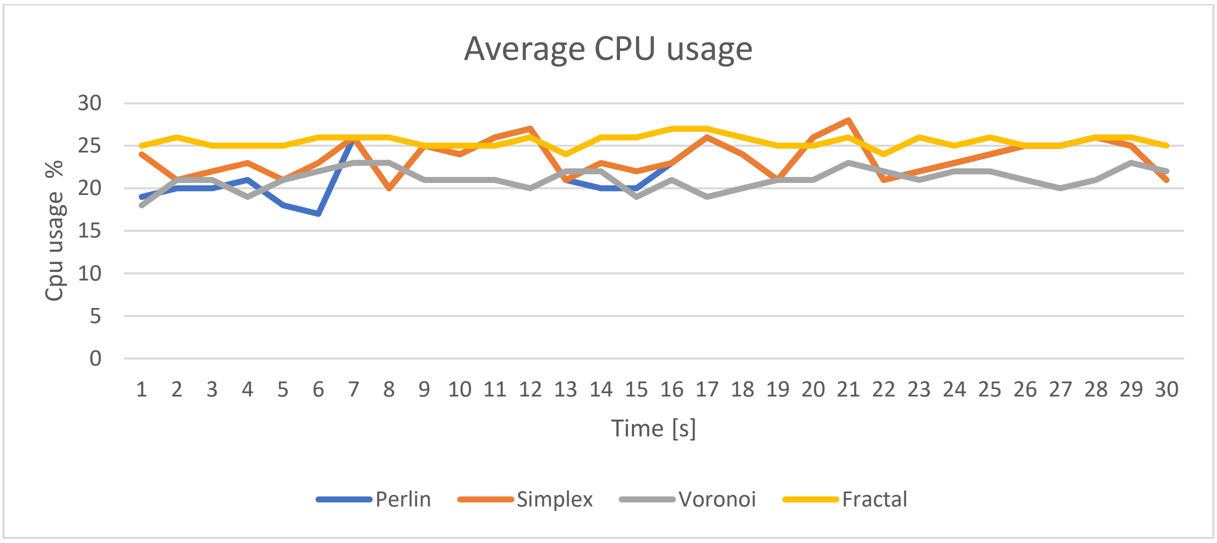


Figure 40: Average CPU usage Godot – summarized

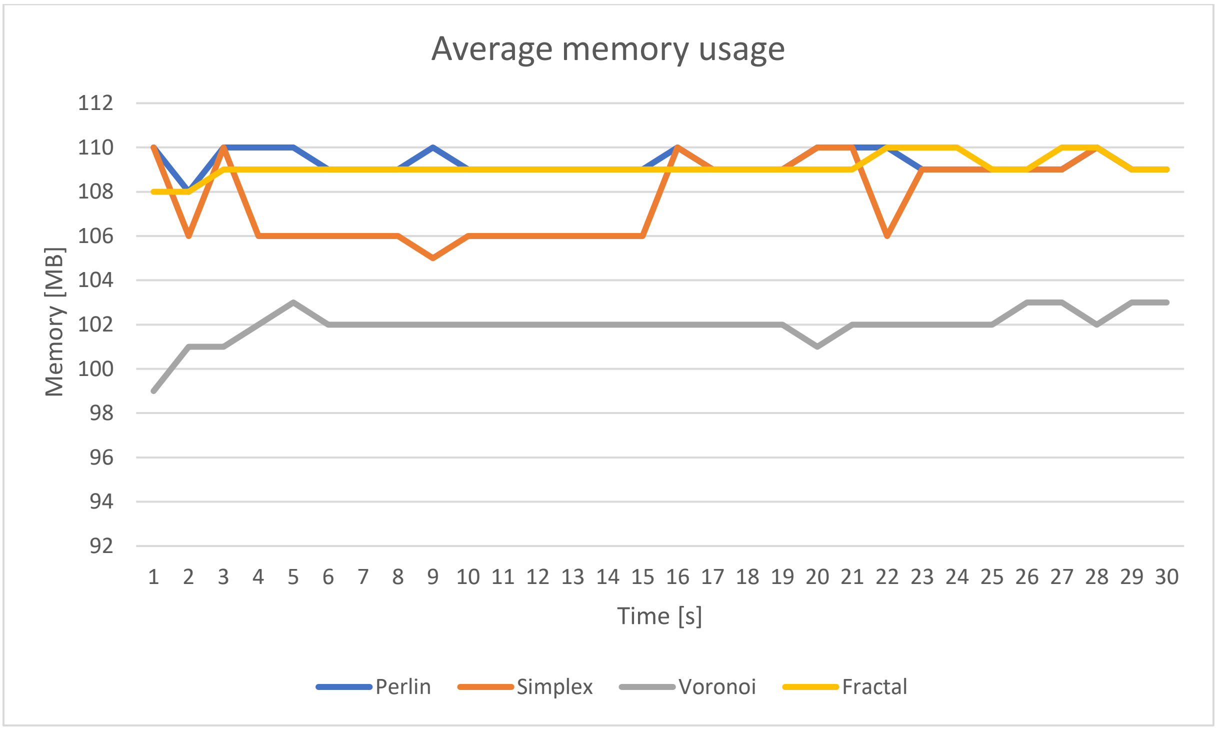


Figure 41: Average memory usage Godot - summarized

4.3.4. Unity research

For Unity engine tests all gathered results are presented in **Attachment A**, under **Unity engine research charts**, in next chapter summarized results are presented on collective charts. For this engine analogous to previous case, results are presented on multiple charts, to make it easier to read the results with different scale, moving average was introduced on Figure 65.

4.3.5. Unity – summarized results

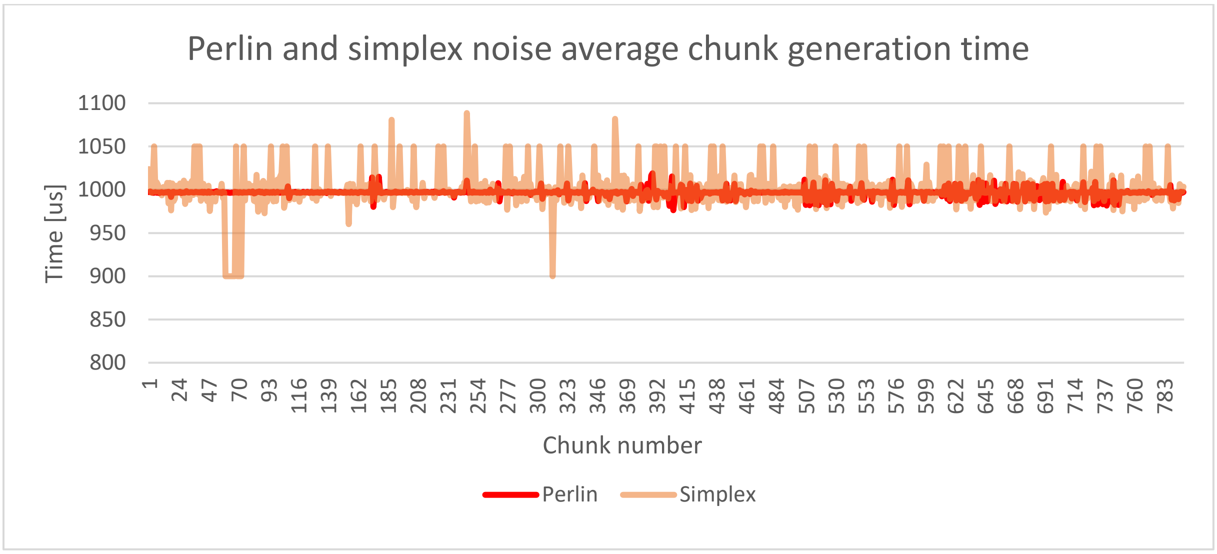


Figure 42: Perlin and Simplex noise summarized chunk generation time

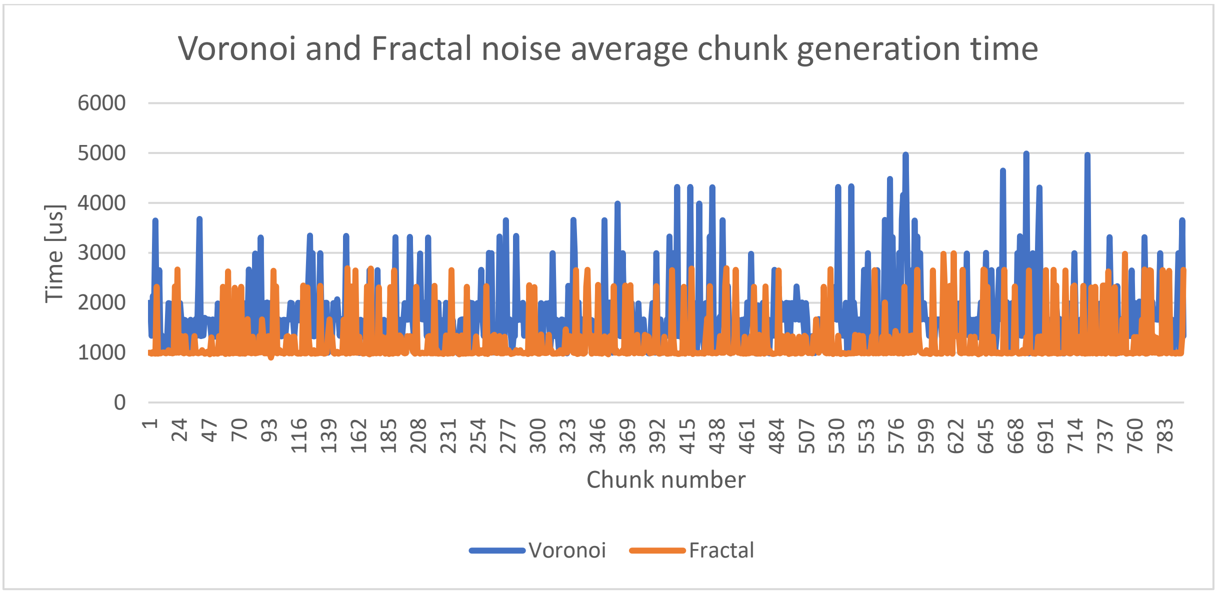


Figure 43: Voronoi and Fractal noise summarized chunk generation time

There are separate charts for Voronoi together with fractal and other for Perlin and simplex noises due to significant difference in scale, and this way charts are more readable.

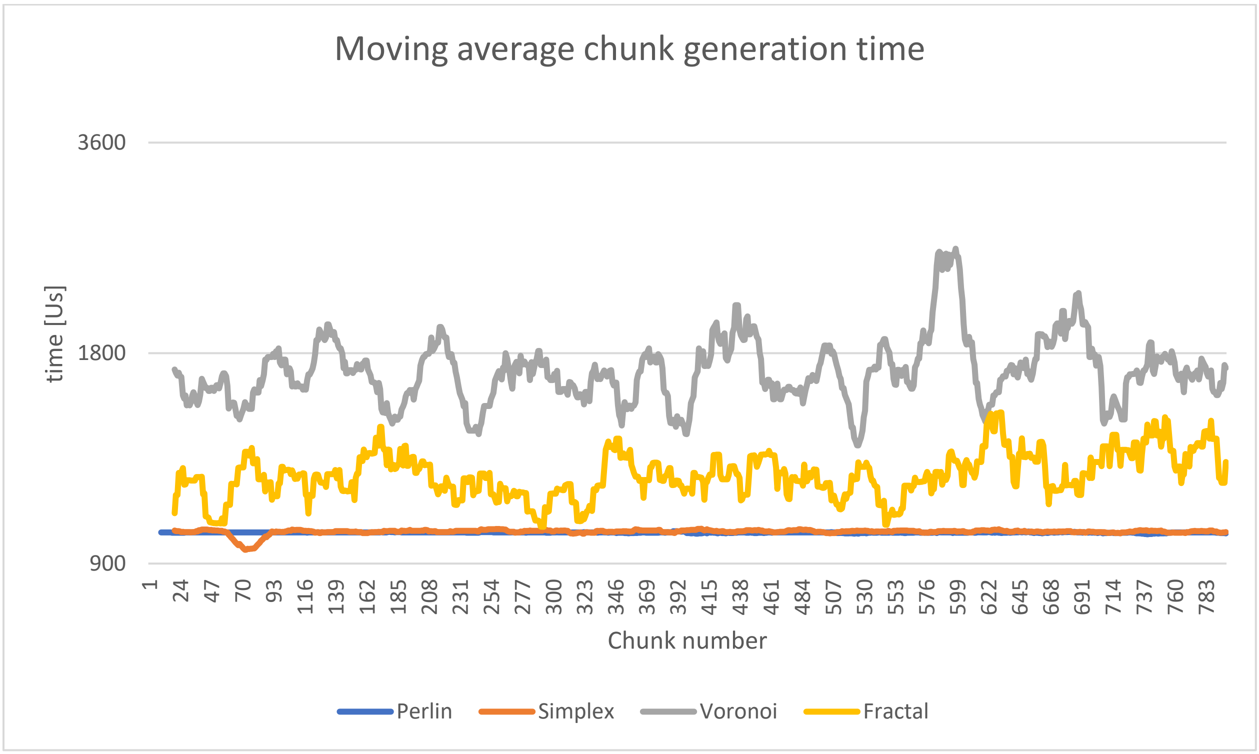


Figure 44: Moving average of chunk generation time

It is also good to show moving average, it helps more to see the results, and the gap between them. On figure 56 it is very clear that Perlin and simplex noise behavior is more constant, and Voronoi with fractal noises are more unstable, in unity engine, for this particular implementation. There are a number of possible causes for this abnormality of Fractal and Voronoi algorithms, some of them will be raised in further part of the thesis, when conclusions will be formulated.

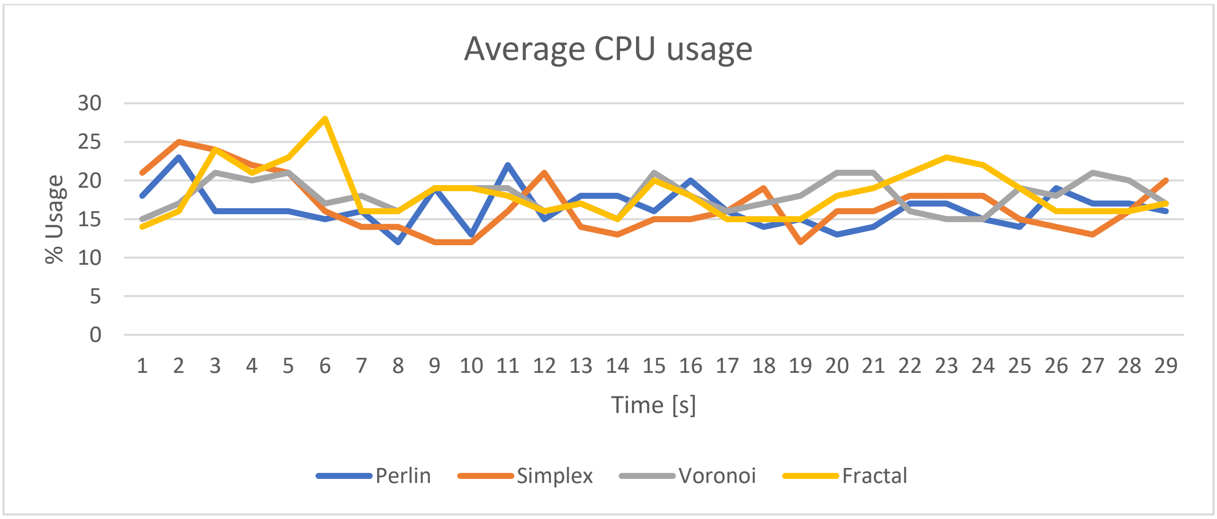


Figure 45: Summarized CPU usage data in Unity

CPU usage chart for Unity engine looks similar to the one for Godot engine, which would suggest that compute power usage for both engines will be similar – that will be verified in statistical tests part.

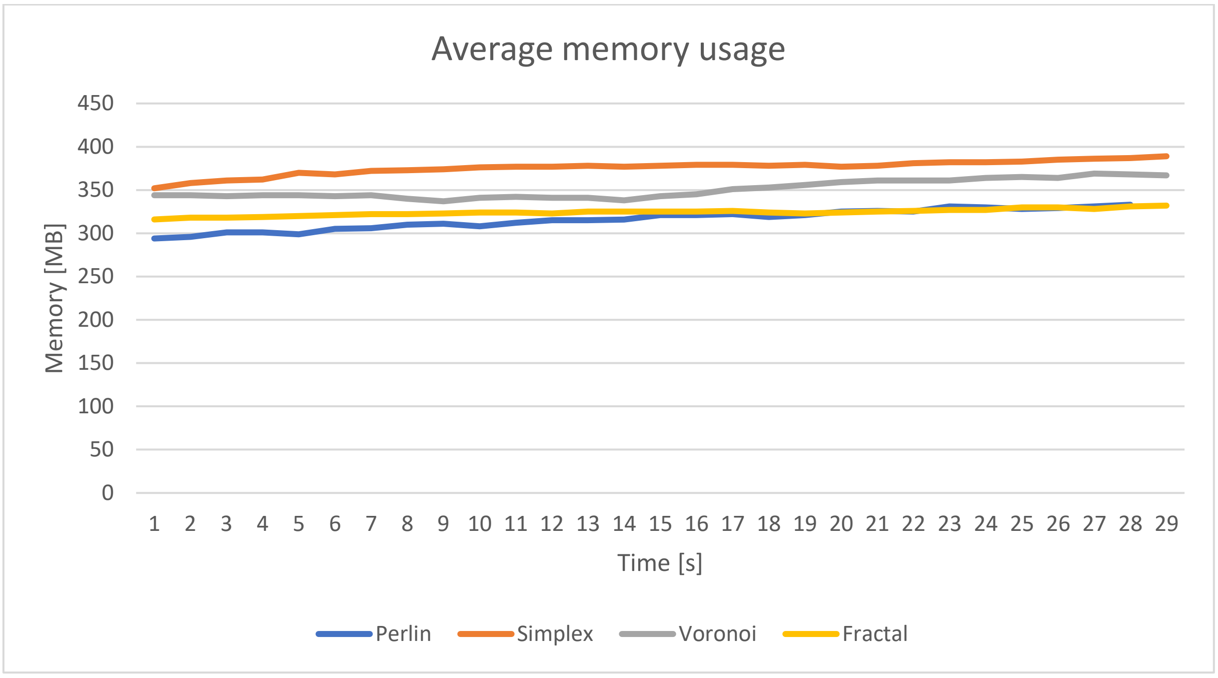


Figure 46: Summarized memory usage data in Unity

Memory usage for Unity is quite constant and as presented on Figure 67, memory usage chart is almost linear.

4.4. Statistical analysis

From the data gathered in previous part of the work, there is visible difference mostly between time, not only among different engines, but also within one engine – Unity at most. Based on that data it logical and expected to see if difference is significant for given significance level using proper statistics methods. However before that will be done, in general each noise worked as expected according to descriptions found in literature mentioned in chapters before, and differences between them are acceptable or – as it will be confirmed, statistically insignificant. By the observation of the charts and the data, it is safe to say that noises implemented in Godot engine works faster, and more efficient considering CPU usage as well as memory, there is also big difference in time of generating single chunk. Based on that, I assume that noises implemented in Godot engine and therefore in C++ language will be statistically faster and have better performance than noises implemented in C# for unity. Based on CPU usage and memory hence there is hardly noticeable difference, I would not assume anything on that level, and let us just have a look at statistical tests.

For the purpose of statistical tests there will be statistical hypothesis formulated for all noises in both engines, tests will be carried out for time data, CPU usage and memory usage.

4.4.1. Statistical tests

Data collected in the test phase is independent from each other, all tests were run independently and in the same initial conditions.

Using shapiro-wilk test or ‘sw-test’, I have tested time data for all algorithms to verify if gathered time has normal distribution. The zero-hypothesis stated that the data has normal distribution against the alternative hypothesis that the data does not have normal distribution. According to results of the tests, zero-hypothesis has been rejected and therefore we assume that data does not have normal distribution.

To test statistical differences between the datasets and listed algorithms I will perform a Mann-Whitney U-test, which is based on medians, therefore written hypothesis will apply to median. All tests are run with significance level alpha equal to 0.05. What is more in Mann Whitney U-test an approximate method of calculating p-value will be used, due to the amount of data.

Tests results will be presented in a table, as hypothesis for all tests will be identical. Results shall be read as follows, 0 means that there is no grounds to reject zero-hypothesis and 1 means that H1 hypothesis should be accepted and H0 rejected.

4.4.2. Chunk generation time tests

H0: Medians for both Unity and Godot chunk generation time are equal, there is no statistically significant difference

H1: Median for noise in Godot engine is smaller than the other noise in Unity

Table 1: Chunk generation time statistical tests results

	Godot				
	Noise	Perlin	Simplex	Voronoi	Ridged multifractal
Unity	Perlin	1	1	1	1
	Simplex	1	1	1	1
	Voronoi	1	1	1	1
	Ridged multifractal	1	1	1	1

Tests indicated, that in all cases when comparing godot and unity engines, chunk generation time is more favorable in for godot, and difference is statistically significant.

4.4.3. CPU usage tests

H0: Medians for both Unity and Godot CPU usage are equal, there is no statistically significant difference

H1: Median for noise in Godot engine is smaller than the other noise in Unity

Table 2: CPU usage statistical tests results

	Godot				
	Noise	Perlin	Simplex	Voronoi	Ridged multifractal
Unity	Perlin	0	0	0	0
	Simplex	0	0	0	0
	Voronoi	0	0	0	0
	Ridged multifractal	0	0	0	0

Because there are no grounds to reject H0, more tests will be run with different H1 hypothesis – right sided, to verify if median for Unity engine is smaller.

H0: Medians for both Unity and Godot CPU usage are equal, there is no statistically significant difference

H1: Median for noise in Unity engine is smaller than the other noise in Godot

Table 3: CPU usage statistical test results with different hypothesis

	Godot				
	Noise	Perlin	Simplex	Voronoi	Ridged multifractal
Unity	Perlin	1	1	1	1
	Simplex	1	1	1	1
	Voronoi	1	1	1	1
	Ridged multifractal	1	1	1	1

Right sided test result indicates that CPU usage median is smaller in Unity in comparison to Godot engine, and the difference is statistically significant.

4.4.4. Memory usage tests

Based on the data gathered in previous chapters, alternative hypothesis will be left sided, stating that Godot memory usage median is smaller.

H0: Medians for both Unity and Godot memory usage are equal, there is no statistically significant difference
H1: Median for noise in Godot engine is smaller than the other noise in

Table 4: Memory usage statistical tests results

	Godot				
	Noise	Perlin	Simplex	Voronoi	Ridged multifractal
Unity	Perlin	1	1	1	1
	Simplex	1	1	1	1
	Voronoi	1	1	1	1
	Ridged multifractal	1	1	1	1

In all cases test result indicates that zero hypothesis should be rejected and, alternative H1 hypothesis accepted – which means that for all noises, memory usage is statistically significant smaller for Godot engine.

4.4.5. Statistical tests summary

Statistical tests were performed for all the data gathered in the research chapter. It should be noted, that there is no doubt there is a difference big enough to be statistically significant. According to the data presented on charts, time difference gives the impression of being the greatest and therefore can be most significant – after all it is time that eventually matters.

Statistical tests shows that difference between medians for all noises, is significant and furthermore, alternative hypothesis gives the assurance that in Godot engine time median is smaller for not only corresponding noise, but for all other noises in comparison, in Unity.
Different result than one indicating superiority of Godot over Unity, were given by statistical test comparing CPU percentage usage, when last statistical test will be taken under consideration, the conclusions are that terrain generation in Godot, more specifically – noise algorithms in terrain generation, works faster, statistically significant faster, as a logical consequence noises in Godot use more percent of computing power, and use smaller amounts of memory. In Unity, as algorithms are working for a longer period of time, and use lesser percent of computing power. There are more differences to discuss, deviations in the data visible on charts. There are a few factors that may cause those aberrations, and abnormalities between Perlin noise and simplex noise for example – in Godot engine. Those matters will be raised in last chapter of the thesis, along with indication of further possible research in that area.

5. Conclusion

In this last part of the thesis, I will refer to the scope of the thesis, research questions that were raised, as well as the research part itself together with the results. The goal of the thesis has been achieved, the comparison of corresponding algorithms in chosen implementations showed differences.

5.1. Research summary

The main purpose of this work was to review different methods of procedural terrain generation, and in next step compare them in narrow area – the noises, and verify if noise used to generate terrain will show significant difference in corresponding implementations, which means not only different noises between each other, but also noises between engines – as it was stated in previous chapters.

Performed research, and in next step – statistical research based on medians with Mann Whitney U-test revealed that in case of time, Godot holds the advantage in all tested noises. Statistical tests revealed that alternative hypothesis that median for Godot engine is smaller than for Unity is correct, the same conclusions could be learned from the raw analysis of the data presented on charts in chapters previous to statistical tests. I have gathered analogical results about memory usage, noises implemented in Godot showed lower tendency to consume memory – observation visible on charts. That premise was confirmed later in statistical tests, Godot memory consumption is statistically significant smaller than Unity. Only one difference was in computing power, Godot consumed more CPU percent than Unity, and again the difference in CPU usage was statistically significant, in favor of the Unity. However it is not quite proper to say that the difference is in favor of Unity. When looking at the result as a one, and connecting all the dots with each other the results indicate that this is logical for Godot to consume more CPU power, as it works faster. Advantage of Godot engine is presumably in the noise library implementation, and also in whole Godot implementation. As mentioned before Unity implementation is written in C#, and so is the noises library, Godot on the other hand has been written in C++ language, which when used properly has much more potential in optimization than C#. Language difference and optimization gap would also explain memory usage, as C++ language does not use garbage collector mechanism, therefore it is possible to manage memory more efficiently. Eventually all those things and factors mentioned, result in better time efficiency of Godot engine. However it is hard to say if it would be some kind of constant behavior, it would be good to run some additional tests on completely different test environment, different machine.

There are visible differences on some data visualizations presented in research chapter, data show visible to the naked eye differences in time for simplex noise and Perlin noise for Godot engine, simplex noise Is represented by the highest values on Figure 38. This situation is understandable and might be caused by non-optimal implementation of simplex noise in Godot engine (and in Unity as well), the reason behind this state of affairs is that simplex noise is patented and therefore not available in sources such as git hub or Godot implementation itself, which is also an open source project. Godot offers implementation of open simplex noise and that algorithm was tested. For Unity engine the case looks a bit different as thanks to [53] I was able to obtain implementation of simplex noise implemented in C#. There difference visible on charts, between fractal and Voronoi noises in Unity and similarly to previous cases, it might be caused by additional operations in both cases, as Voronoi is not noise itself, the diagram is generated first by different algorithm (fortune), and then processed further. It might also be a matter of better parametrization of each algorithm, tests was run with standard parameters to preserve uniformity of the tests, but of course it is possible to define a set of parameters and run analogical tests just to emerge the best ones. Finally the issue may lay in test platform, as processes monitored by the tools covered in previous chapters are related to single execution, in other words CPU usage for example does not cover all processes running on the computer – therefore tests run on better test machine could give better times, CPU and memory usage results – then it is possible that difference will not be that much noticeable.

Statistical tests along with the data presented on charts certainly indicates that there is a difference between algorithms and engines, what is more – the difference is statistically significant in all testes cases:

- Chunk generation time
- CPU usage
- Memory usage

5.2. Further work

Obviously it is important not to finish the research on the current level. There is still much possibilities for experiments as well as more possible indicators to verify. This research is only an introduction to possible research. The thesis covers only few ontogenetical algorithms used in broadly understood procedural terrain generation. Much more algorithms might be, and should be tested in terms of performance but also in terms of attractiveness, because after all generated terrain should also – at the very end, be nice to the final user.

As mentioned, there are a lot of possible way to research in further steps.

- Research additional (new) algorithms – perhaps including teleological algorithms

- Examine other indicators – other than time, CPU usage and memory usage, maybe not completely different, but additional.

- Run tests on more platforms – Research will be more complete if identical tests, will be performed on multiple platforms, preferably with

    - Low configuration
    - Medium configuration
    - High configuration

  By configuration I understand computer components of central unit.

Next research step would involve more combined algorithms, as described tests included only generating of kind of 'raw' terrain, without any user-friendly elements, or any elements at all. The purpose of generating terrain for example in games is to use it in real game, without additional work of designers, therefore it would be good to eventually test how all combined algorithms work together – and using previous research regarding noises, such as this thesis, try to get the best final result.

Bibliography

[1] A. Małek, "Metody Generowania Plansz w Grach Sieciowych," 2016. [Online]. Available: https://www.ii.pwr.edu.pl/~kopel/mgr/2016.06%20mgr%20Malek.pdf. [Accessed 16 12 2019].

[2] Notch, „The Word of Notch, Terrain Generation, part 1," [Online]. Available: https://notch.tumblr.com/post/3746989361/terrain-generation-part-1. [Data uzyskania dostępu: 10 12 2019].

[3] Flafla2, „Understanding Perlin Noise," 09 August 2014. [Online]. Available: https://flafla2.github.io/2014/08/09/perlinnoise.html. [Data uzyskania dostępu: 20 12 2019].

[4] L. Oravakangas, "Game Envirnoment Creation: Efficient and Optimized Working Methods," 2015. [Online]. Available: https://www.theseus.fi/bitstream/handle/10024/101918/Thesis_LauraOravakangas.pdf?sequence=1&isAllowed=y. [Accessed 17 11 2019].

[5] E. Credits, „Procedural Generation - How games create infinite words - Extra credits," 22 07 2015. [Online]. Available: https://www.youtube.com/watch?v=TgbuWfGeG2o&index=24&list=PLhyKYa0YJ_5CciOENqQoB_QuZJ41r9sML. [Data uzyskania dostępu: 26 12 2019].

[6] „Teleological algorithms," [Online]. Available: http://pcg.wikidot.com/pcg-algorithm:teleological. [Data uzyskania dostępu: 20 01 2020].

[7] p. wikidot, „Genetic algorithm," [Online]. Available: http://pcg.wikidot.com/pcg-algorithm:genetic-algorithm. [Data uzyskania dostępu: 23 01 2020].

[8] L. P. C. T. Lucas Ferreira, „A multi-population genetic algorithm for procedural generation of levelf for platform games," [Online]. Available: https://www.researchgate.net/publication/266658225_A_multi-population_genetic_algorithm_for_procedural_generation_of_levels_for_platform_games. [Data uzyskania dostępu: 22 02 2020].

[9] R. S. J. K. J. J. L. Teong Joo Ong, „Terrain generation using genetic algorithms," [Online]. Available: http://research.cs.tamu.edu/keyser/Papers/GECCO2005.pdf. [Data uzyskania dostępu: 15 01 2020].

[10] P. G. Paul Walsh, „Terrain Generation using an interactive genetic algorithm," *IEEE Congress on evolutionary computation,* pp. 1-7, 2010.

[11] Q. W. G. Z. F. T. X. Y. K. Li, „Example-based Realistic Terrain Generation," *Springer,* pp. 811-818, 2006.

[12] M. Klingensmith, „How we generate Terrain in DwarfCorp," [Online]. Available: https://www.gamasutra.com/blogs/MattKlingensmith/20130811/198049/How_we_Generate_Terrain_in_DwarfCorp.php. [Data uzyskania dostępu: 31 1 2020].

[13] J. Olsen, „Realtime Procedural terrain generation," 2004.

[14] Jason, „Softologyblog,” [Online]. Available: https://softologyblog.wordpress.com/2016/12/09/eroding-fractal-terrains-with-virtual-raindrops/. [Data uzyskania dostępu: 31 01 2020].

[15] p. wiki, „Fire propagation,” [Online]. Available: http://pcg.wikidot.com/pcg-algorithm:fire-propagation. [Data uzyskania dostępu: 30 01 2020].

[16] F. Gennari, „3DWorld, fire propagation,” [Online]. Available: http://3dworldgen.blogspot.com/2017/. [Data uzyskania dostępu: 31 01 2020].

[17] M. Mueller, „Developing a Fire Propagation System for a Massively Multiplayer Online Game,” 2018.

[18] P. wiki, „Cellular Automata,” [Online]. Available: http://pcg.wikidot.com/pcg-algorithm:cellular-automata. [Data uzyskania dostępu: 29 01 2020].

[19] S. O. Sam Sondgrass, „Generating Maps Using Markov Chains,” [Online]. Available: https://pdfs.semanticscholar.org/379f/a1bb03257804bef9a0b186c6d128509f9696.pdf. [Data uzyskania dostępu: 28 01 2020].

[20] T. B. B. Gleidson Mendes Costa, „Procedural terrain generator for platform games using Markov chain,” *Proceedings of SBGames - Computing Track,* 2018.

[21] A. Tatarinov, „Perlin noise in Real-time Computer Graphics,” [Online]. Available: https://pdfs.semanticscholar.org/b49d/a45b19f6ad6c28b3748223b715810711d15f.pdf. [Data uzyskania dostępu: 25 01 2020].

[22] „Perlin Noise - wiki free encyclopedia,” [Online]. Available: https://en.wikipedia.org/wiki/Perlin_noise. [Data uzyskania dostępu: 12 12 2019].

[23] B. C. Yann Thorimbert, „Polynomial methods for fast procedural terrain,” 4 December 2018. [Online]. Available: https://www.researchgate.net/publication/309037528_Polynomial_method_for_Procedural_Terrain_Generation. [Data uzyskania dostępu: 23 03 2020].

[24] I. Parberry, „Designer Worlds: Procedural Generation of Infinite Terrain from Real-World Elevation Data,” *Journal of Computer Graphics Techniques,* tom 3, 2014.

[25] „Simplex noise - wikipedia,” [Online]. Available: https://en.wikipedia.org/wiki/Simplex_noise. [Data uzyskania dostępu: 31 01 2020].

[26] S. Gustavson, „Simplex noise demystified,” *Linkoping University, Research Report,* 2005.

[27] T. J. B. A. G. Rose, „Algorithms and Approaches for procedural terrain generation - A Brief Review of Current Techniques,” w *International Conference on Games and Virtual Worlds for Serious Applications*, 2016.

[28] L. J. Y. K. C. M. Q. L. H. Y. J. M. Dong Junyu, „Survey of procedural Methods for Two-Dimensional Texture Generation,” *Sensors,* tom 20, 2020.

[29] „Ridged MultiClass Reference,” [Online]. Available: http://libnoise.sourceforge.net/docs/classnoise_1_1module_1_1RidgedMulti.html.

[30] K. J. d. k. S. A. G. T. T. R. B. Ruben M Smelik, „A Survey of Procedural methods for Terrain Modeling,” *Proceedings of CASA workshop on 3D Advanced Media In Gaming And Simulation,* 2009.

[31] „Stackoverflow, Procedural terrain with ridged fractal noise,” [Online]. Available: https://stackoverflow.com/questions/36796829/procedural-terrain-with-ridged-fractal-noise. [Data uzyskania dostępu: 31 1 2020].

[32] M. C. P. F. J. H. Przemyslaw Prusinkiewicz, „Modeling plant development with L-Systems”.

[33] A. Pike, „Modeling plants with L-Systems,” [Online]. Available: https://allenpike.com/modeling-plants-with-l-systems/.

[34] „Midpoint displacement algorithm,” [Online]. Available: http://pcg.wikidot.com/pcg-algorithm:midpoint-displacement-algorithm.

[35] T. Archer, „Procedurally Generating Terrain,” *Morningside College.*

[36] T. Wheeler, „Fractal Terrain : Midpoint displacement algorithm,” [Online]. Available: http://tim.hibal.org/blog/fractal-terrain-midpoint-displacement-algorithm/.

[37] S. losh, „Terrain generation with diamond square,” [Online]. Available: https://stevelosh.com/blog/2016/06/diamond-square/.

[38] „Voronoi diagram,” [Online]. Available: http://pcg.wikidot.com/pcg-algorithm:voronoi-diagram. [Data uzyskania dostępu: 21 01 2020].

[39] S. Johnstone, „github,” [Online]. Available: https://github.com/SteveJohnstone/VoronoiMapGen.

[40] Destrolas, „Terrain generation 3: Voronoi Diagrams,” [Online]. Available: https://leatherbee.org/index.php/2018/10/06/terrain-generation-3-voronoi-diagrams/.

[41] X. C. S. H. I. S. P. G. B. Aitor Santamaria-Ibirika, „Procedural Playable Cave Systems based on Voronoi Diagram and Delaunay Triangulation,” *International Conference of Cyberworlds,* 2014.

[42] Y. Hyttinen, „Terrain synthetis using noise,” University of Tampere, Faculty of Natural Sciences, 2017.

[43] „No man's Sky, gamepedia,” [Online]. Available: https://nomanssky.gamepedia.com/Terrain_(Atlas). [Data uzyskania dostępu: 23 03 2020].

[44] O. Spiridis, „Procedural heightmap generation for level of detail purposes,” Politechnika Wrocławska, Wrocław, 2017.

[45] S. L. R. C. T. D. G. D. S. E. J. L. K. P. M. Z. Ares Lagae, „A survey of Procedural Noise Functions,” *Computer Graphics Forum,* 2010.

[46] K. academy, „Khan academy - perlin noise,” [Online]. Available: https://pl.khanacademy.org/computing/computer-programming/programming-natural-simulations/programming-noise/a/perlin-noise. [Data uzyskania dostępu: 20 04 2020].

[47] „Wikiwand - Fortune's algorithm,” [Online]. Available: https://www.wikiwand.com/en/Fortune%27s_algorithm. [Data uzyskania dostępu: 19 04 2020].

[48] J. Kriz, „Multi-Fractal Terrain Generation,” *Masaryk University,* 2019.

[49] godotengine, „Godot engine docs,” Open source, [Online]. Available: https://docs.godotengine.org/en/3.1/about/introduction.html. [Data uzyskania dostępu: 21 04 2020].

[50] gamedesigning team, „Gamedesigning,” 2020. [Online]. Available: https://www.gamedesigning.org/career/video-game-engines/. [Data uzyskania dostępu: 24 04 2020].

[51] M. Russinovich, „Microsoft docs,” Microsoft, 18 12 2018. [Online]. Available: https://docs.microsoft.com/en-us/sysinternals/downloads/procmon. [Data uzyskania dostępu: 03 05 2020].

[52] Ricardojmendez, „Github - LibNoise,” [Online]. Available: https://github.com/ricardojmendez/LibNoise.Unity. [Data uzyskania dostępu: 12 02 2020].

[53] M. Rutkowska, Proceduralne generowanie map w Unity, Wrocław, 2019.

[54] Zylann, „Godot - FastNoise,” [Online]. Available: https://github.com/Zylann/godot_fastnoise. [Data uzyskania dostępu: 12 2 2020].

[55] D. S.Ebert, F. K. Musgrave, D. Peachey, K. Perlin i S. Worley, Texturing and Modeling, Third Edition: A procedural approach, Morgan Kaufmann, 2002.

[56] J. Hell, M. Clay i H. Elarag, „Hierarchical Dungeon Procedural Generation And Oprimal Path Finding Based On User input,” w *CCSC:Northestern Conference*, 2017.

[57] I. Parberry, „Modeling Real-World Terrain with Exponentially Distributed Noise,” *Journal of Computer Graphics Techniques*, tom 4, 2015.

[58] W. edit, „Fortune's algorithm,” [Online]. Available: https://en.wikipedia.org/wiki/Fortune%27s_algorithm. [Data uzyskania dostępu: 19 04 2020].

List of figures

List of tables

Attachment

1. Attachment A

a. Godot engine research charts

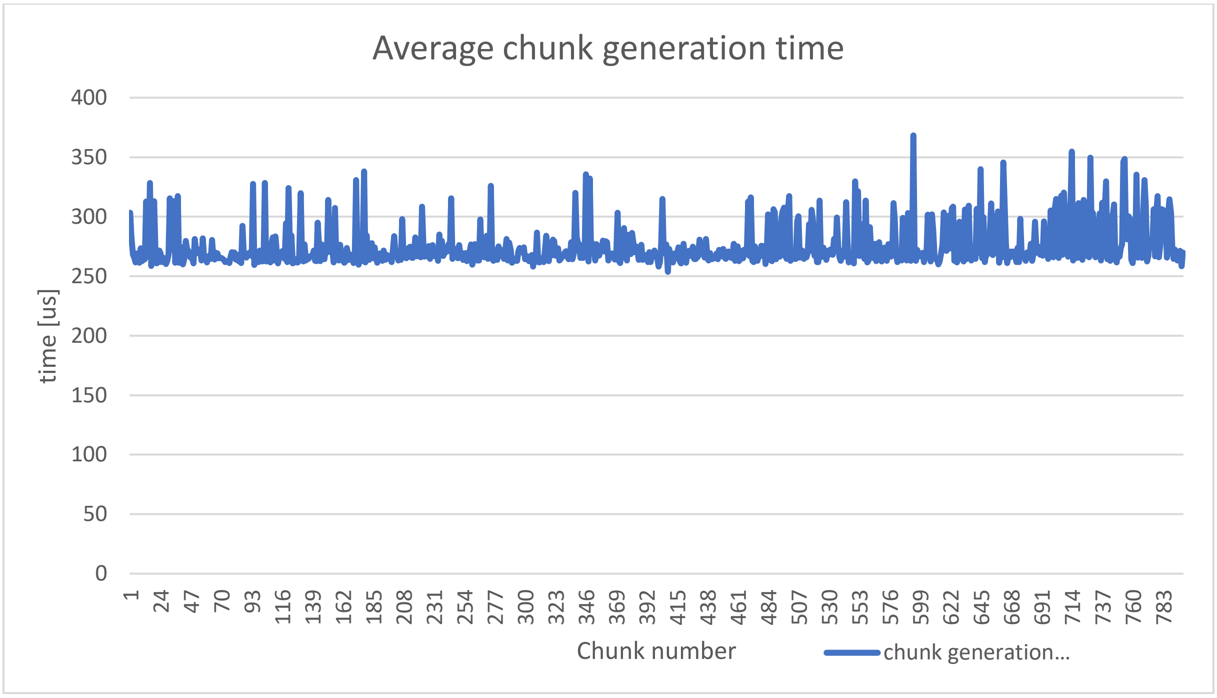


Figure 47: Chunk generation time for Perlin noise in Godot

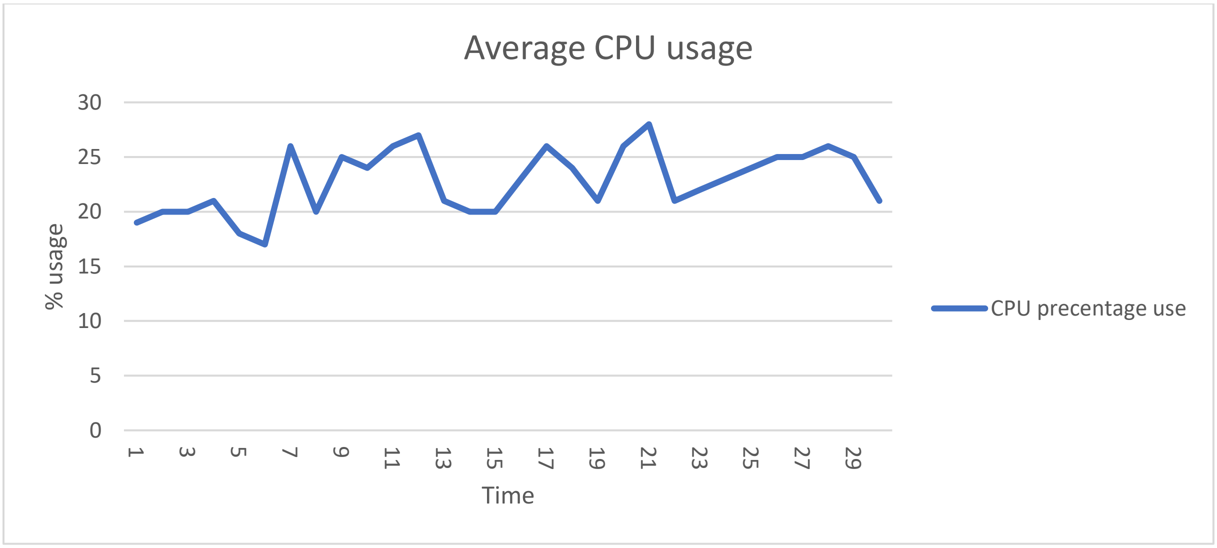


Figure 48: CPU usage for Perlin noise Godot

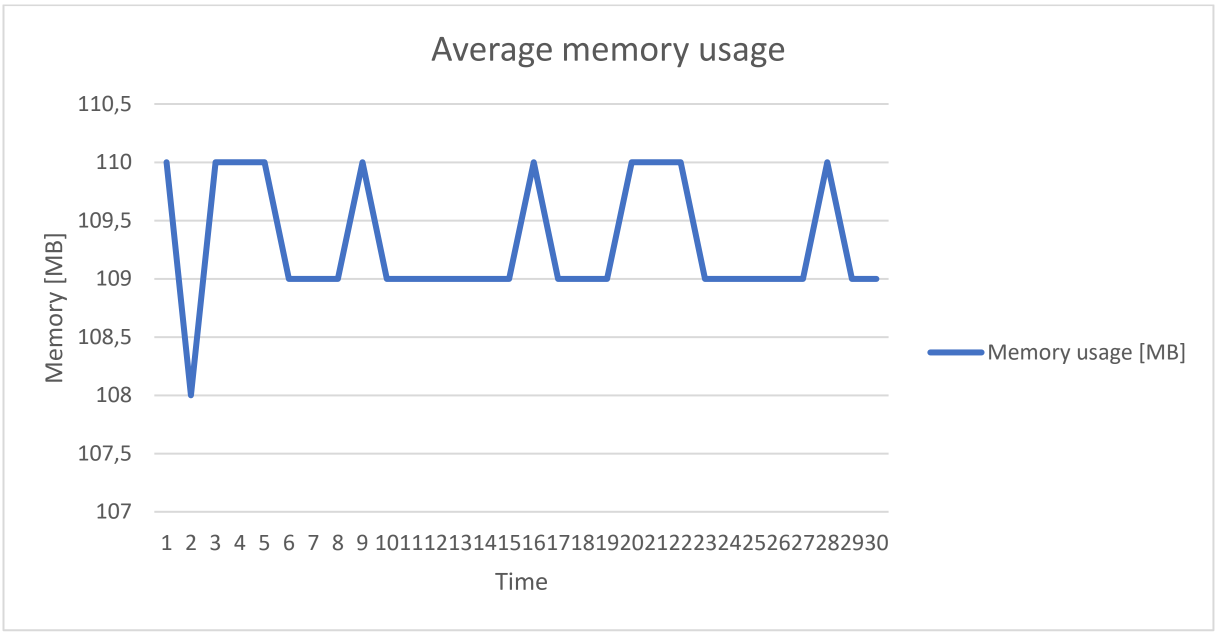


Figure 49: Memory usage in time for Perlin noise godot

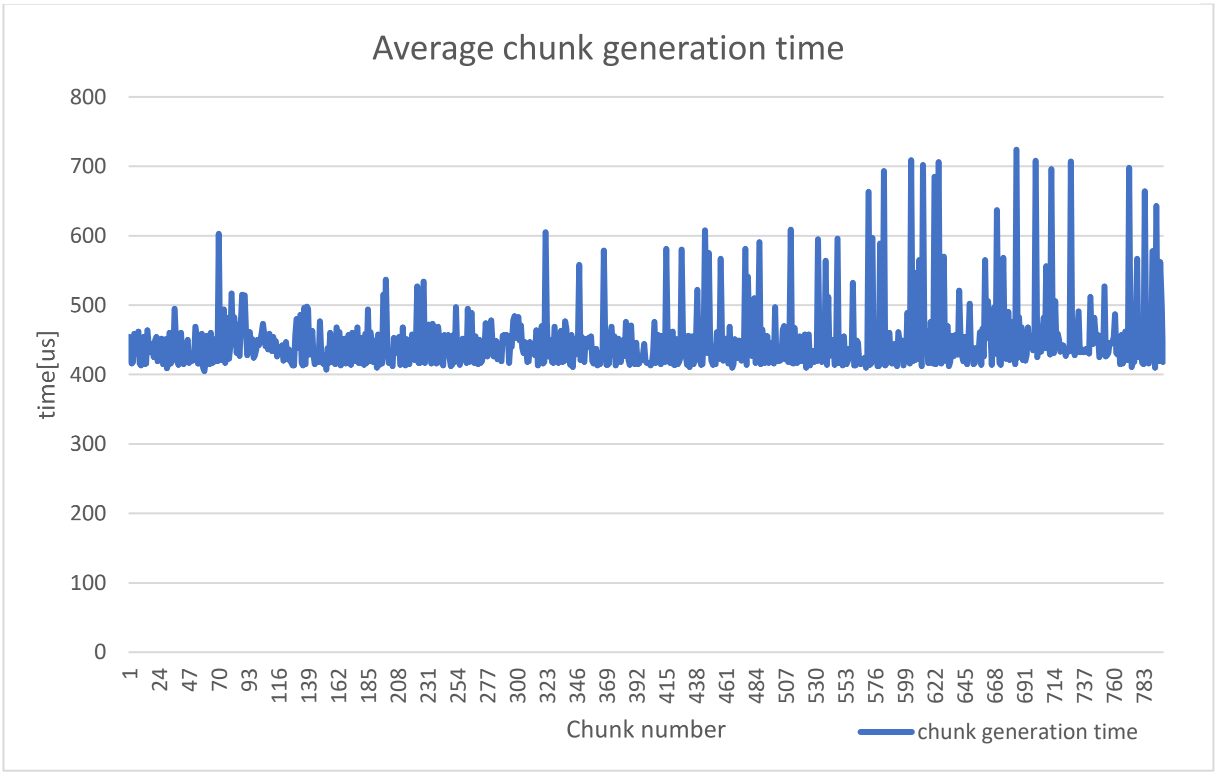


Figure 50: Chunk generation time for simplex noise in Godot

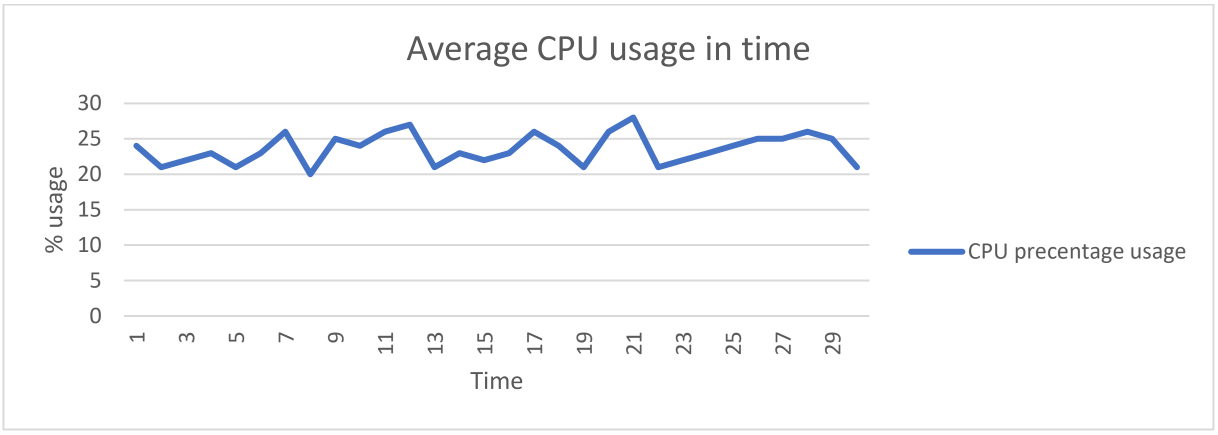


Figure 51: CPU usage for simplex noise in Godot

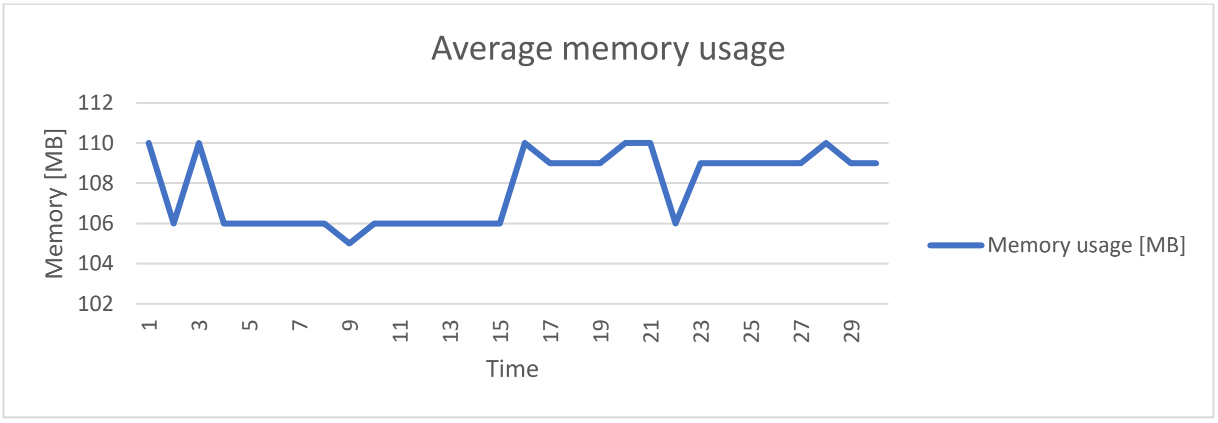


Figure 52: Memory usage for simplex noise in Godot

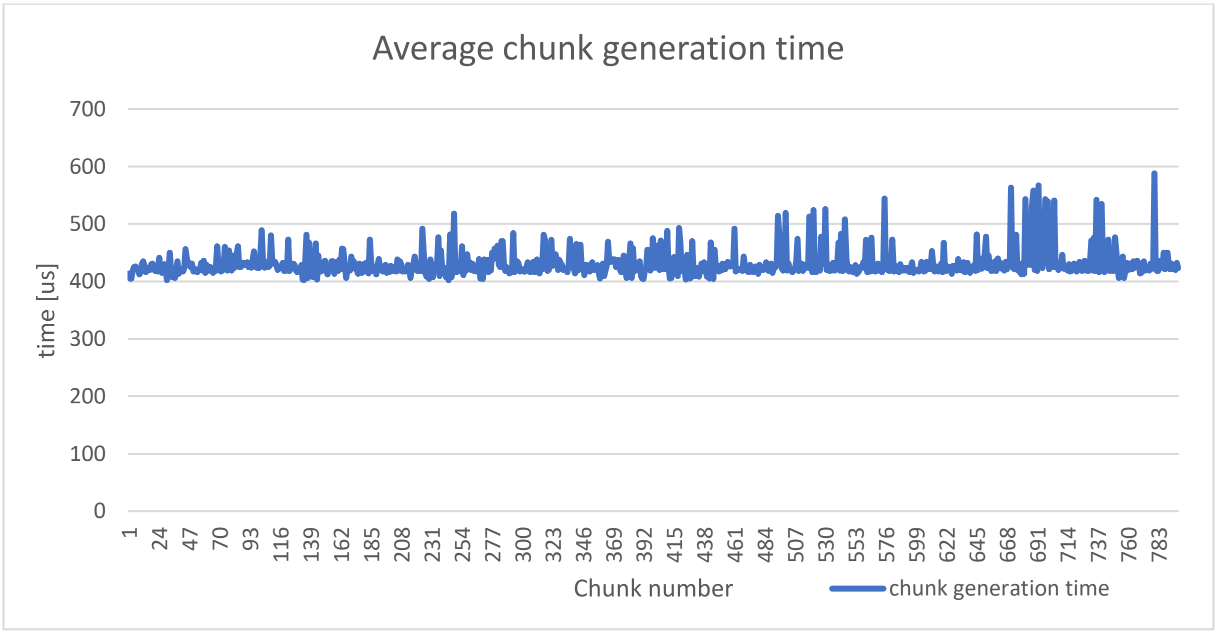


Figure 53: Chunk generation time for Voronoi in Godot

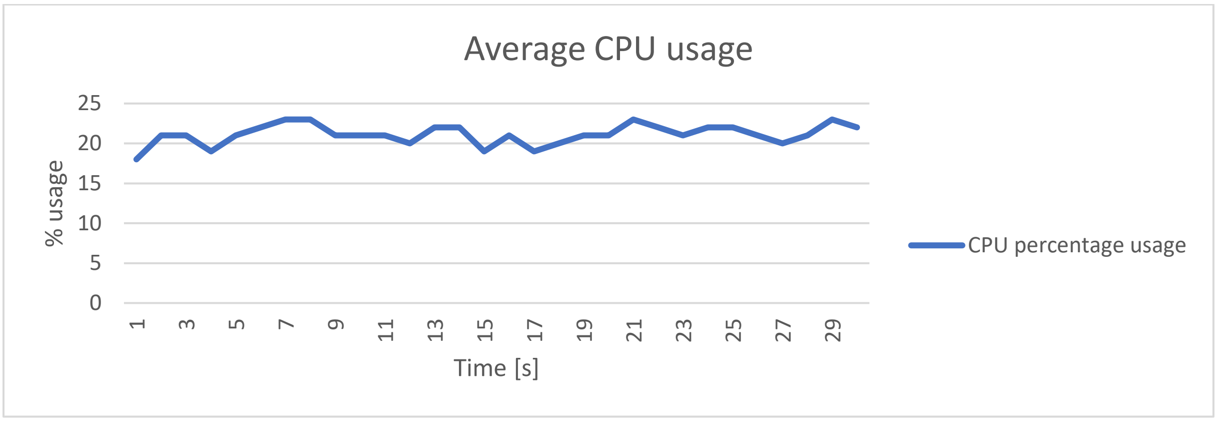


Figure 54: CPU usage for Voronoi in Godot

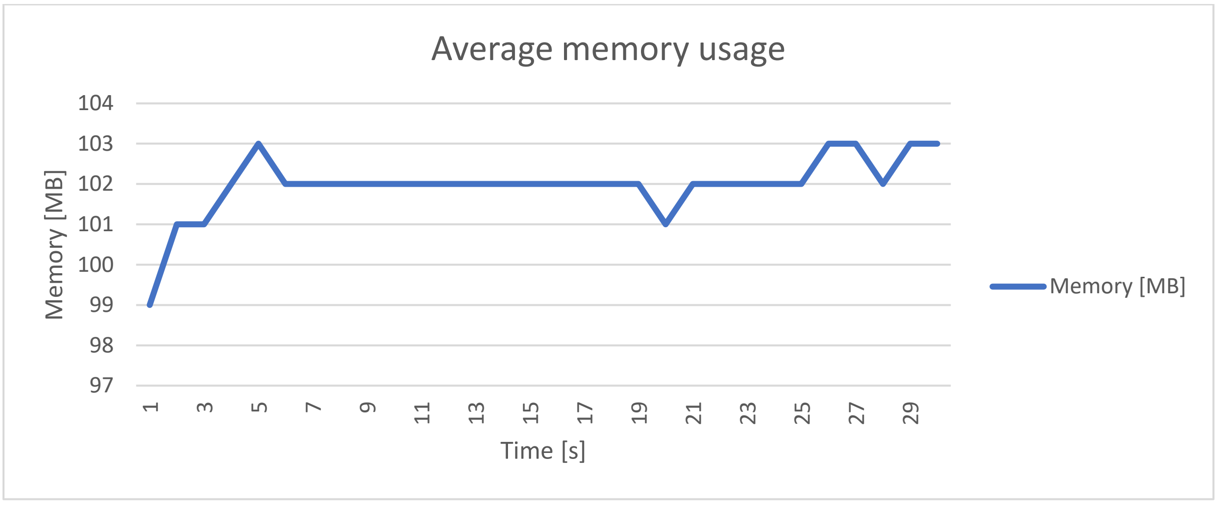


Figure 55: Memory usage for Voronoi in Godot

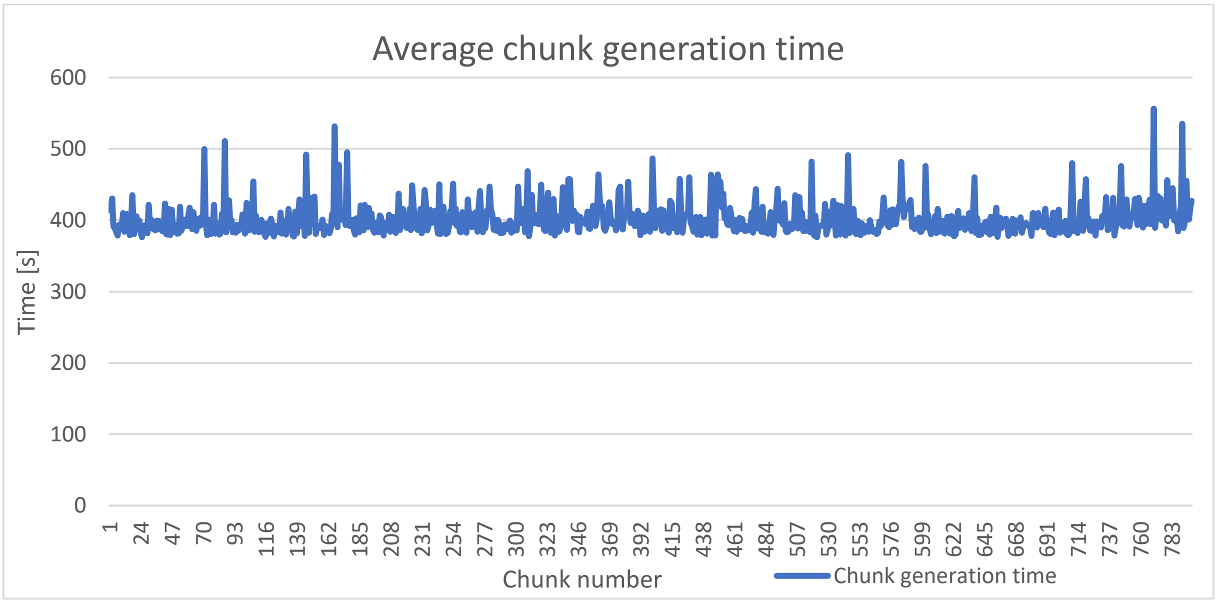


Figure 56: Average chunk generation time for fractal noise Godot

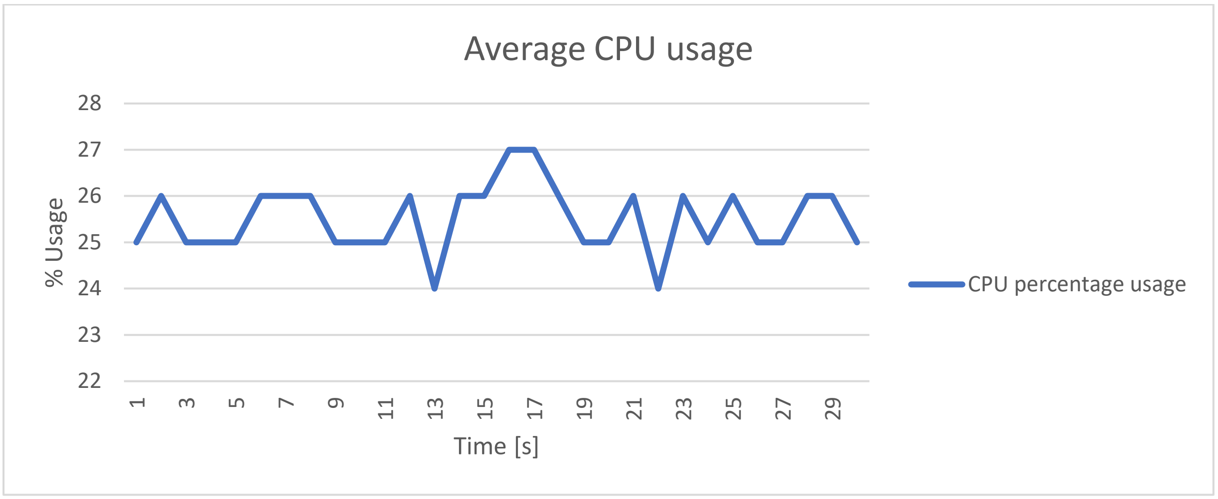


Figure 57: CPU usage for fractal noise Godot

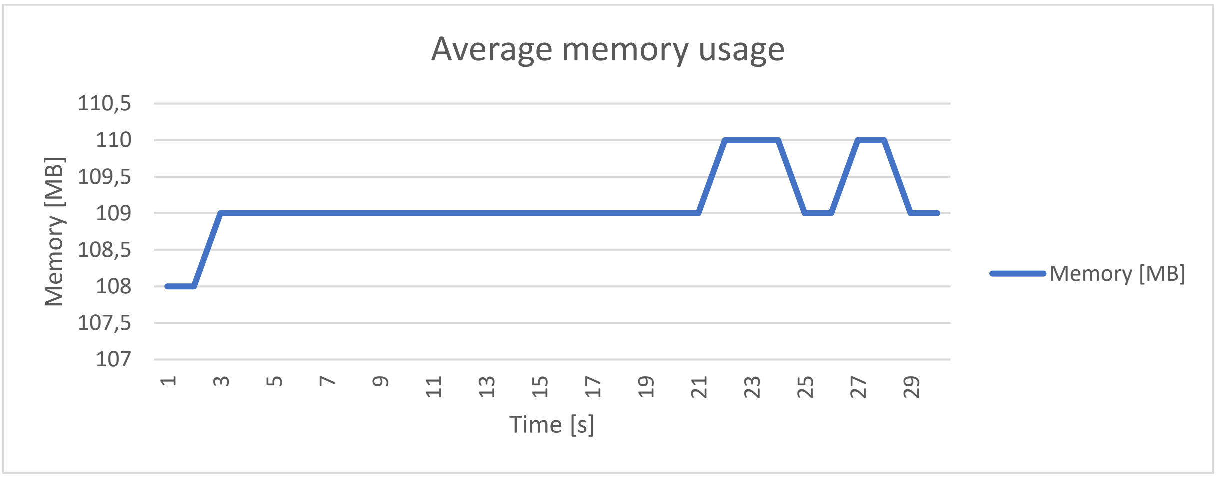


Figure 58: Memory usage for fractal noise Godot

b. Unity engine research charts

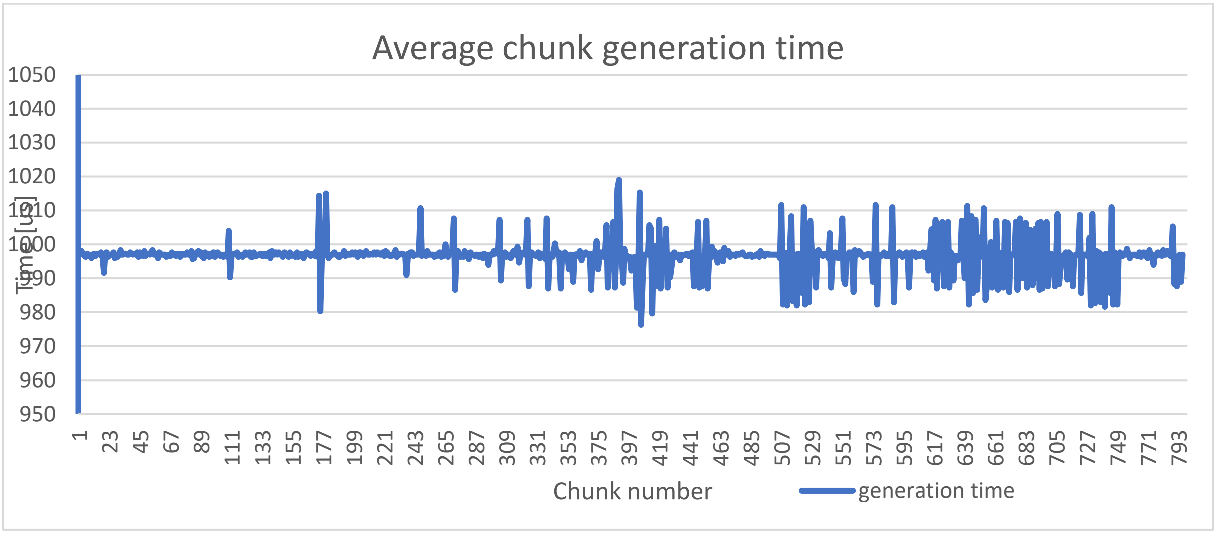


Figure 59: Average chunk generation time Perlin noise Unity

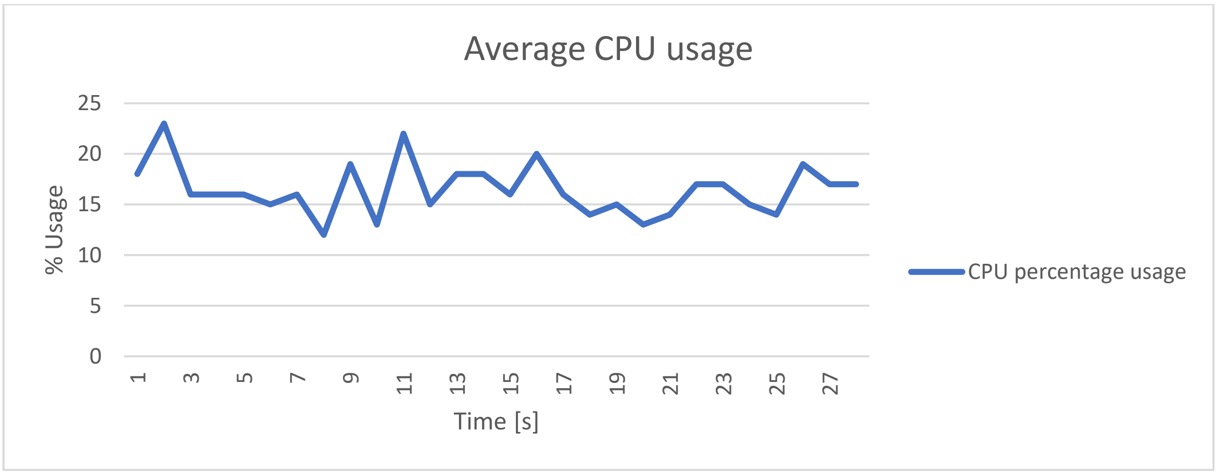


Figure 60: CPU usage for Perlin noise Unity

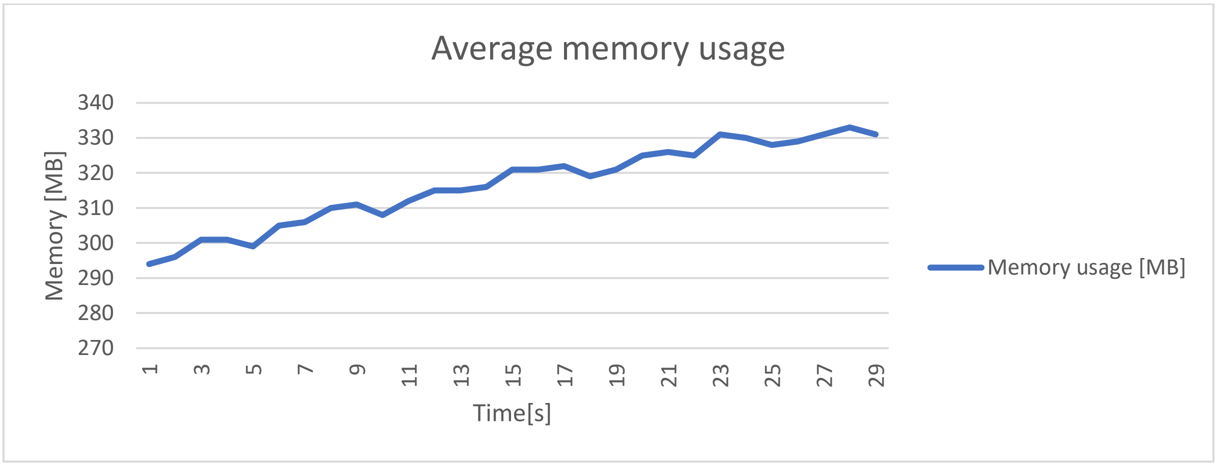


Figure 61: Memory usage for Perlin noise Unity

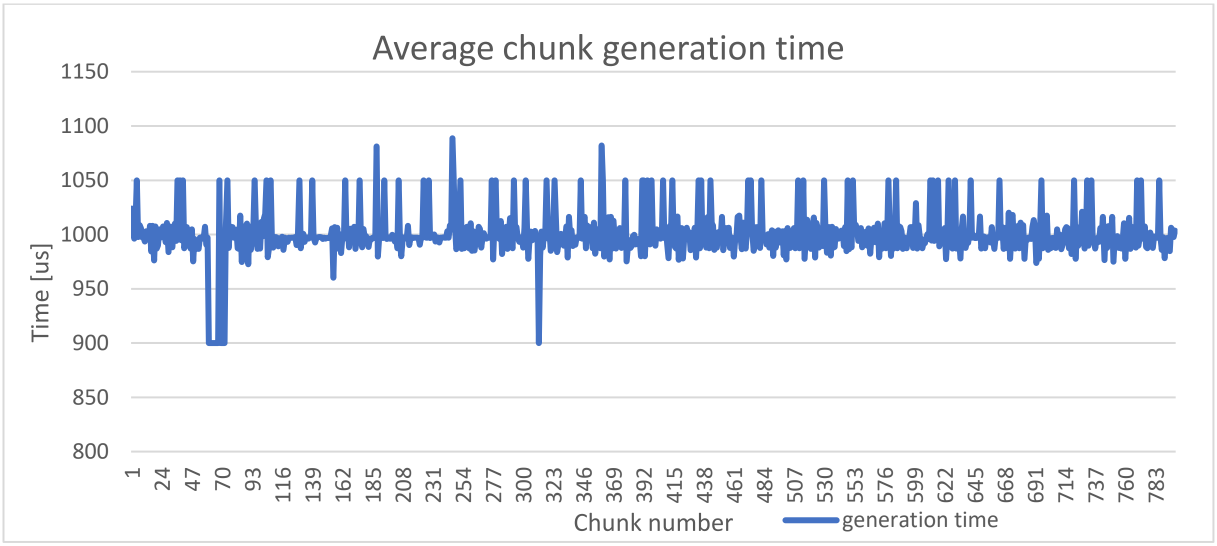


Figure 62: Average chunk generation time simplex noise Unity

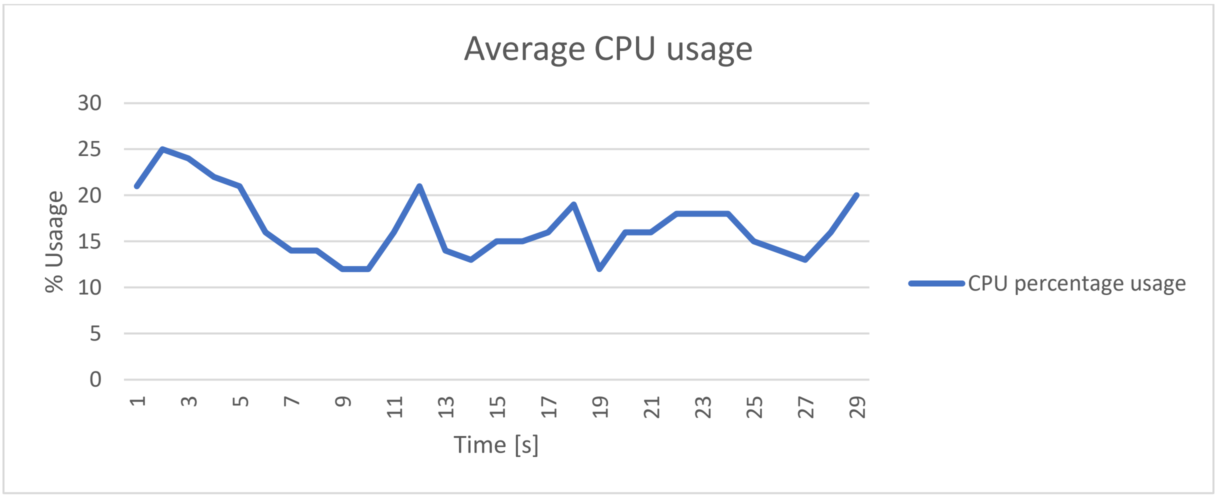


Figure 63: Average CPU usage for simplex noise Unity

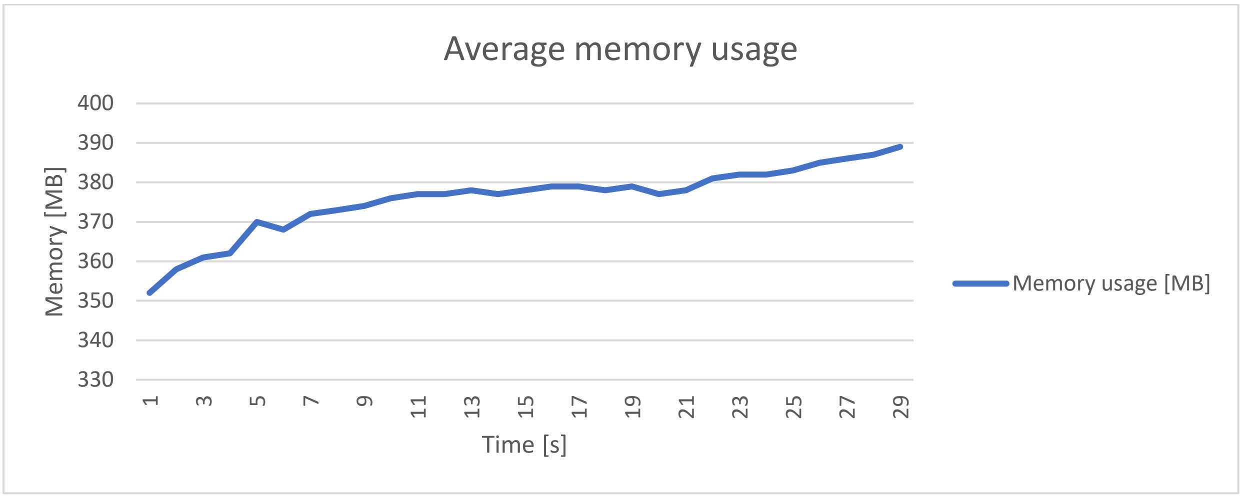


Figure 64: Memory usage for simplex noise Unity

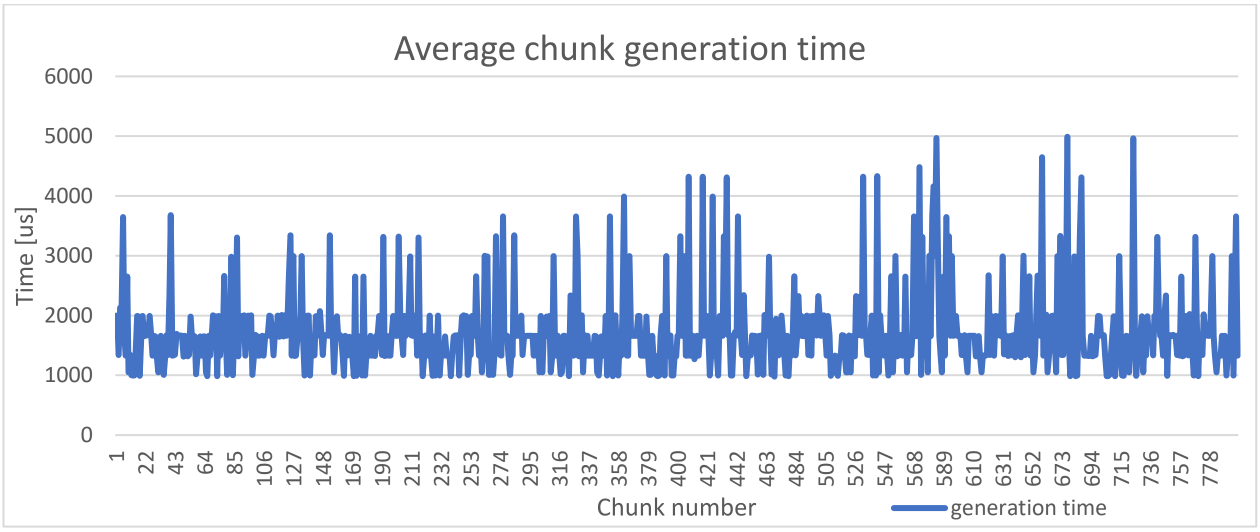


Figure 65: Average chunk generation time voronoi Unity

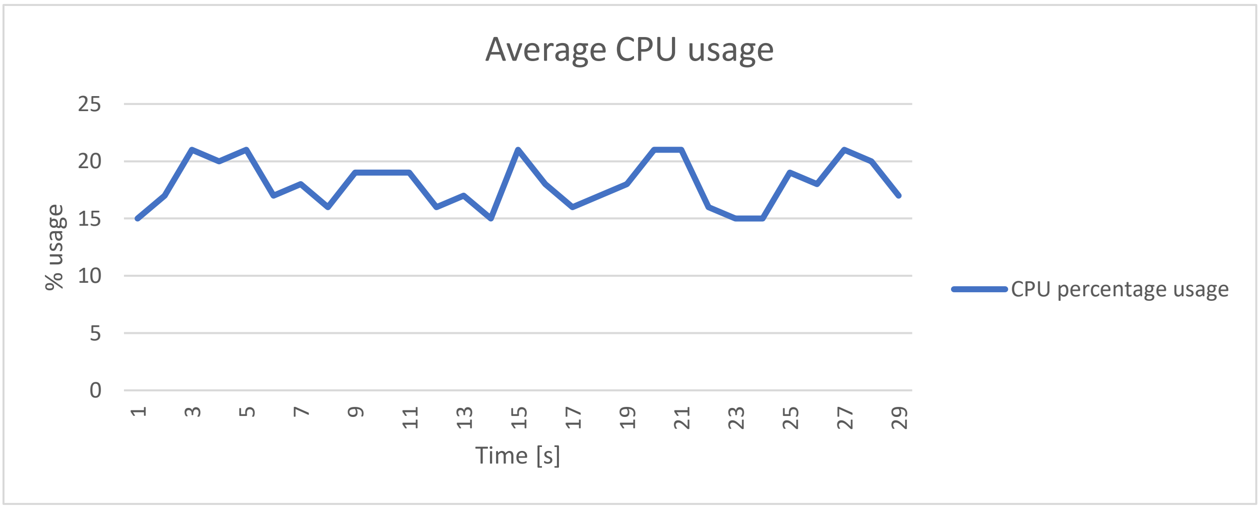


Figure 66: Average CPU usage for voronoi in Unity

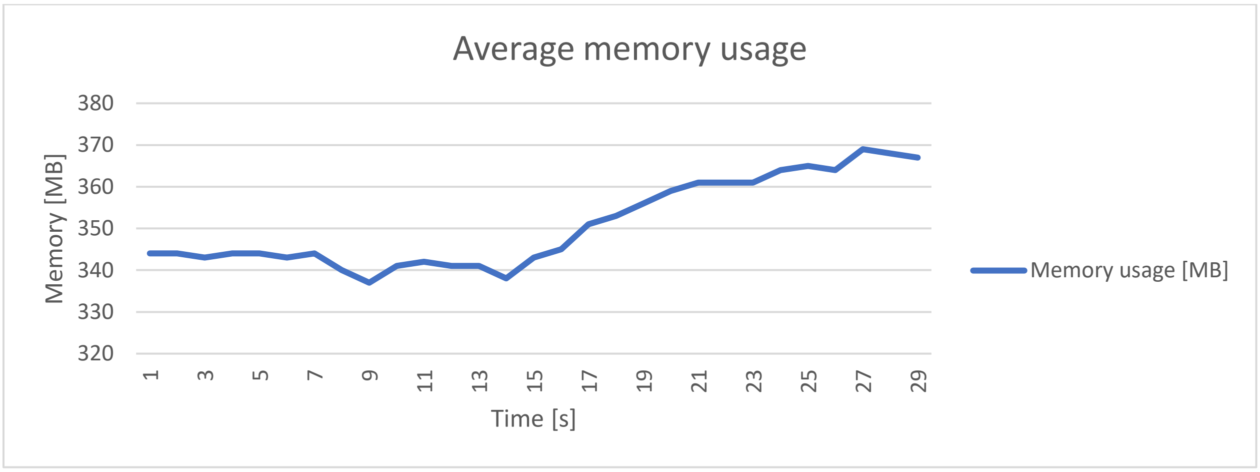


Figure 67: Memory usage for Voronoi in Unity

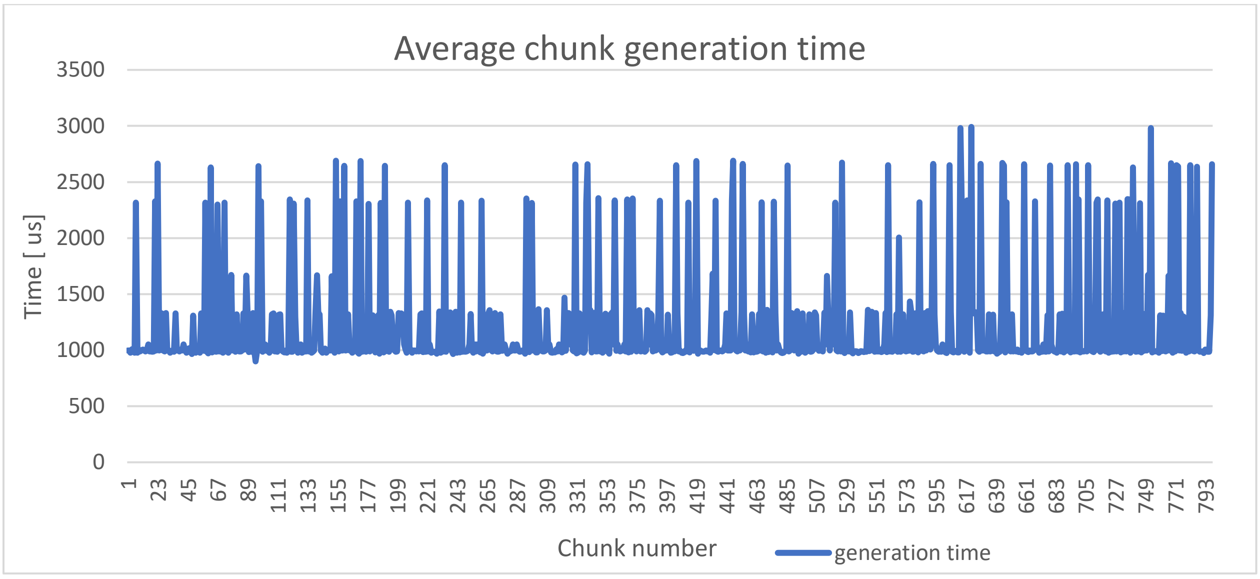


Figure 68: Average chunk generation time fractal noise Unity

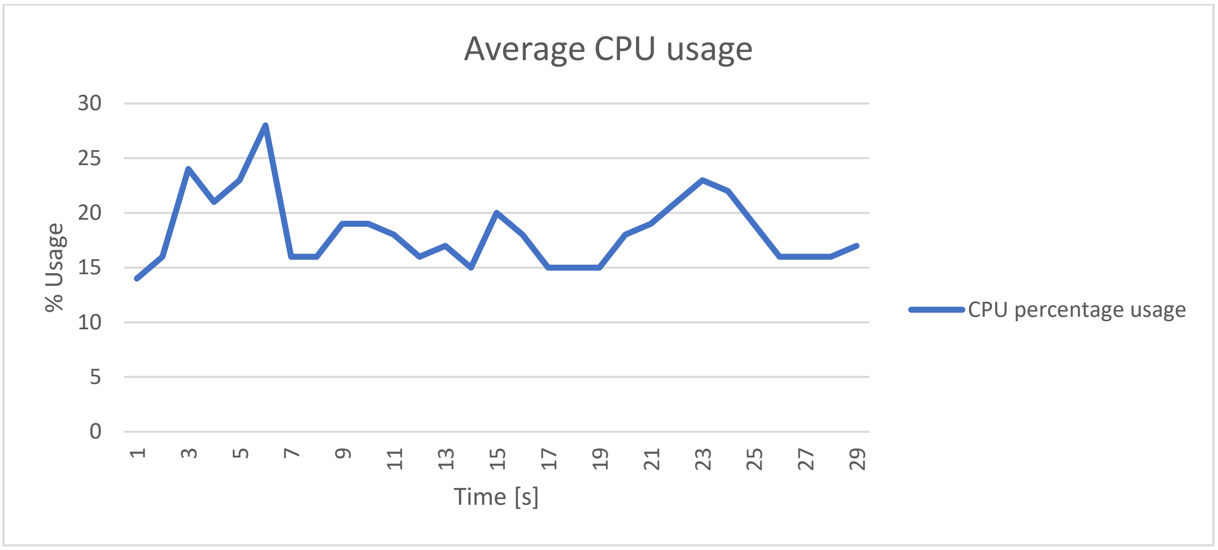


Figure 69: Average CPU usage for fractal noise Unity

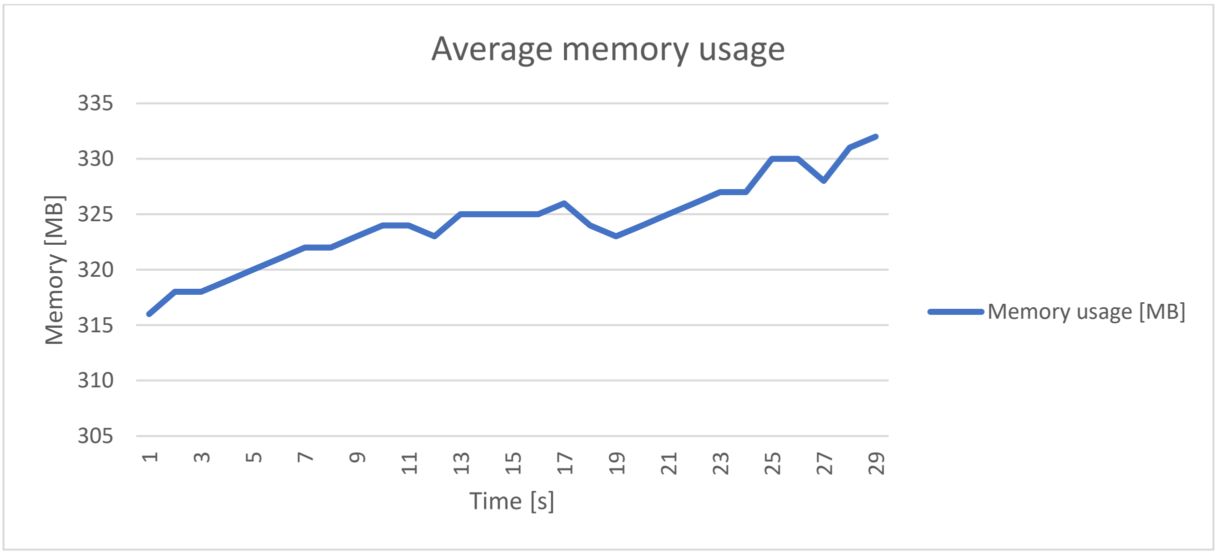


Figure 70: Average memory usage for fractal noise Unity