MINISTRY OF SCIENCE AND HIGHER EDUCATION
OF THE RUSSIAN FEDERATION

FEDERAL STATE BUDGETARY EDUCATIONAL
INSTITUTION OF HIGHER EDUCATION
"RUSSIAN STATE PEDAGOGICAL UNIVERSITY
NAMED AFTER A. I. HERZEN"

Basic professional educational program
Field of study 01.03.02 Applied Mathematics and Informatics
Form of study – full-time

Final qualification work
(English version)

Perlin noise as a method for generating pseudo-random textures

4th year student
Dmitry Sergeevich Rayskiy

Supervisor:
Candidate of Pedagogical Science, Associate Professor
Elena Victorovna Shumara

Reviewer:
Doctor of Pedagogical Science, Professor
Evgenia Vasilyevna Baranova

Saint Petersburg, 2020

CONTENTS

INTRODUCTION

In the modern computer graphics industry, the issue of creating and using textures is becoming increasingly relevant. Textures are used in the design of raster two-dimensional images and surfaces of three-dimensional objects, in the creation of three-dimensional models, as well as in the creation and animation of fire, clouds, smoke, fog, and other effects. However, existing technologies do not always meet the requirements: most algorithms have high execution times, often produce unrealistic results, and are difficult (or impossible) to seamlessly extend the texture. There is an algorithm that compensates for these shortcomings—the Perlin noise algorithm—which is currently the ***most relevant*** algorithm for procedural texture generation.

The goal of this work was to create a computer system that generates pseudo-random textures using Perlin noise.

- ***The object*** of research is computer systems for procedural texture generation.
- ***The subject*** of the study is Perlin noise generation software tools.

To achieve the stated goal, the following tasks were formulated:

1. Analysis of literature sources devoted to the Perlin noise algorithm and its mathematical apparatus, as well as modern programming tools and methods.
2. Analysis of software products that generate pseudo-random textures, and in particular those that use the Perlin noise algorithm for this purpose.
3. Analysis of existing development environments, identification of their advantages and disadvantages in relation to the stated goal; selection of an environment for developing a computer system.
4. Design and software implementation of a computer system that generates pseudo-random textures using Perlin noise.

Research methods that were used to solve the tasks and achieve the goal of the work:

- theoretical: analysis of literary sources, systematization of the data obtained;
- practical: design of a computer system; development of a computer system that generates pseudo-random textures using Perlin noise; testing of the developed computer system.

The work has practical significance. Creating textures today is a relevant but difficult task. The developed computer system offers an effective and quick solution to this problem.

The work consists of an introduction, two chapters, a conclusion, a list of references containing 11 titles, and appendices.

CHAPTER 1. GENERATION OF PSEUDORANDOM TEXTURES

1.1. Perlin Noise as a Method for Generating Pseudo-random Textures

Let us introduce a number of definitions. Let $\mathbb{I} = [0, 1]$.

Definition 1. Any vector $\vec{u} \in \mathbb{I}^4$ is called a color.

Let us explain. In computer graphics in the RGBA system, the color $\vec{u} = (r, g, b, a)$ is interpreted as the brightness of red (for r), green (for g), and blue (for b) tones in the resulting color of the screen pixel. If more than one color needs to be mixed at a point on the screen, the number a is interpreted as the transparency of that pixel. For r, g and b, zero corresponds to minimum brightness, and one corresponds to maximum brightness. For a, zero corresponds to a completely transparent pixel, and one corresponds to a completely visible pixel.

Definition 2. Let $m, n \in \mathbb{N}$. An image is a two-dimensional array of $m \times n$ colors. Each cell of the array is called a pixel of the image, m and n are the width and height of the image, respectively.

Note. Any digital image is an image in the sense of definition 2.

Example of a 1600×600 pix image:

Definition 3. A texture is any image.

Note. This definition is quite formal. In practice, the concept of texture is used when it is clear from the context of the task at hand that an image satisfying certain criteria is required.

A program that generates a sequence of numbers or vectors, the elements of which are almost independent of each other and obey a distribution close to uniform, will be called a

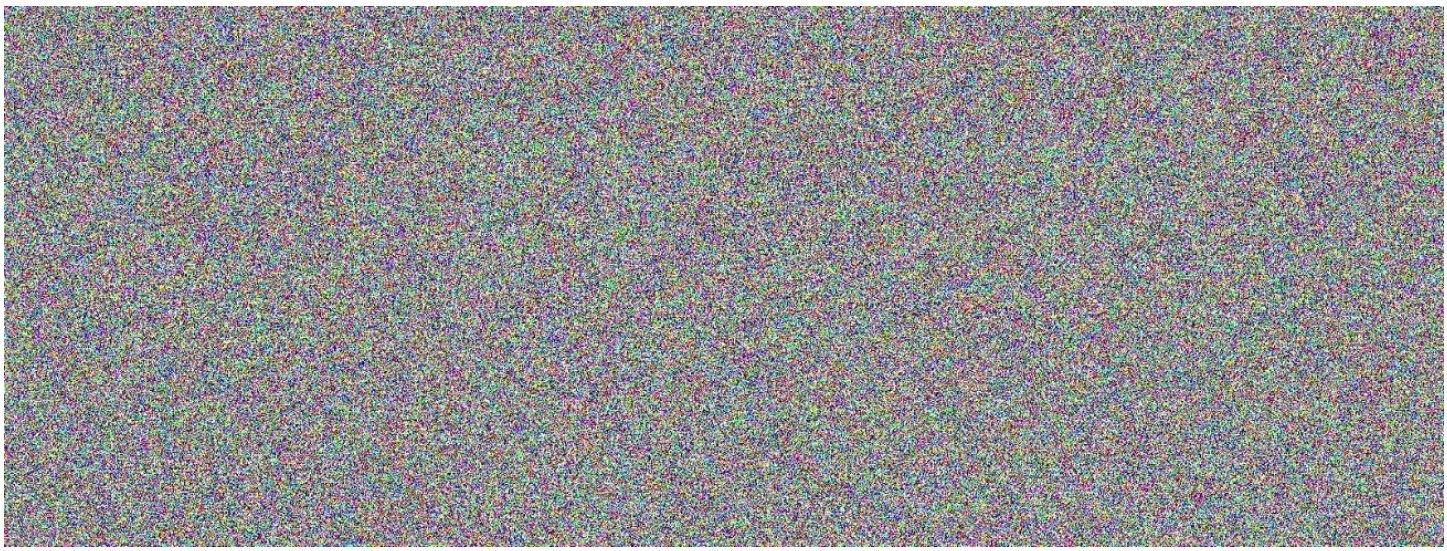
generator of pseudo-random numbers or vectors, respectively. This definition is not strict, but in practice it is sufficient in most cases.

Definition 4. A pseudorandom number is a number obtained as a result of the operation of a pseudorandom number generator.

Definition 5. A pseudo-random vector is a vector obtained as a result of the operation of a pseudo-random vector generator.

Definition 6. Noise is an image in which each pixel has a pseudo-random color.

Example of noise 1600×600 pix:



There is a concept related to the concept of noise. Gradient noise is noise obtained as a result of the operation of a certain algorithm based on the generation and subsequent use of pseudo-random gradients (unit vectors).

Example of gradient noise 2048×2048 pix:

There are a number of texturing tasks for which it is useful to obtain gradient noise. Here are some of them.

- Creating a height map — a black-and-white texture where lighter pixels are interpreted as higher points and darker pixels as lower points.
- Creating effects such as smoke, fire, fog, clouds, malachite, etc. based on gradient noise.

For these tasks, gradient noise is called a texture.

One of the most well-known algorithms for generating gradient noise is Perlin noise algorithm. This is described in article [1]. The author of this article does not provide definitions of the terms used and describes only the two-dimensional case. Let us give a rigorous mathematical description of the algorithm n-dimensional Perlin noise.

Consider a system of N points in space $\mathbb{R}^2$:

$$M = \{(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)\},$$

where $x_1 < x_2 < \cdots < x_N$.

Definition 7. A continuous numerical function f on $[x_1,\ x_N]$ is called an interpolation of the system M if:

$$\begin{cases} f(x_1) = y_1 \\ f(x_2) = y_2 \\ \quad \dots \\ f(x_N) = y_N \end{cases}$$

Definition 8. Let $a, b \in \mathbb{R}, t \in \mathbb{I}$. Then

$$iLerp(a, b, t) \stackrel{\text{def}}{=} (b - a) \cdot Q(t) + a, \text{where}$$

$$Q(t) = t^3 \cdot \big((6t - 15)t + 10\big).$$

Note. Let a and b be fixed. Then $iLerp$ is continuous numerical function on $[0, 1]$ such that $iLerp(a, b, 0) = a,\ iLerp(a, b, 1) = b$. Therefore, $iLerp$ is a nonlinear interpolation of the system $\{(0, a),\ (1, b)\}$.

Let $n \in \mathbb{N}, n \geq 2$. And let $G_r : \mathbb{Z}^n \to \mathbb{R}^{n1}$ be a mapping for $r \in \mathbb{R}$, where $\mathbb{R}^{n1}$ is the set of all n-dimensional vectors of unit length, and this mapping takes pseudo-random values.

Let us introduce the following notation:

- $\lfloor \vec{a} \rfloor$ — coordinate-wise rounding to the smallest integer,
- $\vec{a} \cdot \vec{b}$ — scalar product of vectors $\vec{a}$ and $\vec{b}$,
- $coord_i(\vec{a})$ — the i[th] coordinate of the vector $\vec{a}$,
- $\overrightarrow{e_i}$ — a vector whose i[th] coordinate is equal to 1, and the rest are equal to 0.

Definition 9. n-dimensional Perlin noise is defined as the mapping

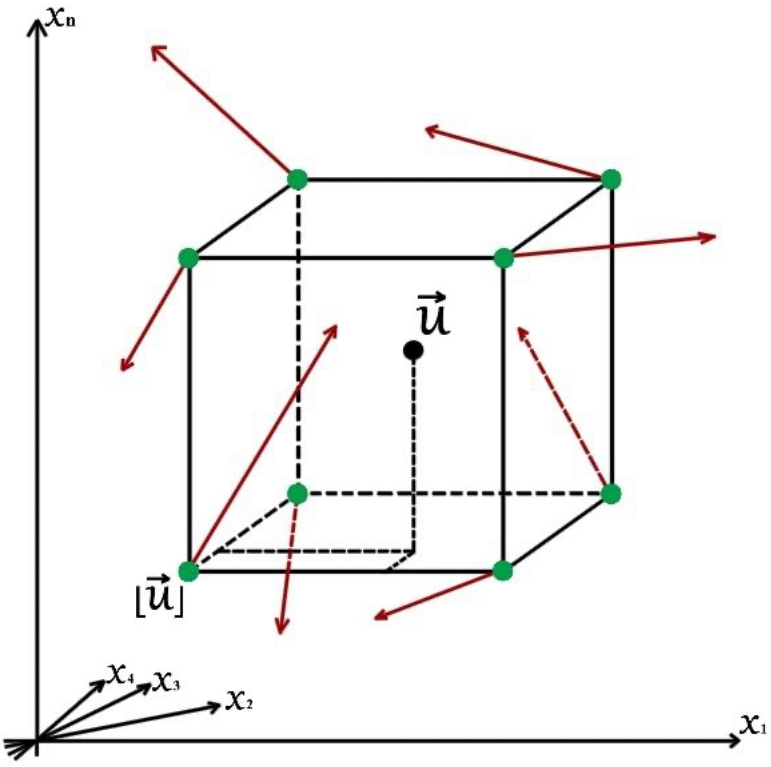
$$P_r^n : \mathbb{R}^n \to \mathbb{R},$$

whose value at an arbitrary point $\vec{u}$ of the n-dimensional space is calculated as follows.

Step 1. To determine the vectors $\overrightarrow{grad}(\vec{k}) = G_r(\lfloor \vec{u} \rfloor + \vec{k})$ for each n-dimensional vector $\vec{k}$, each of whose coordinates is either zero or one.

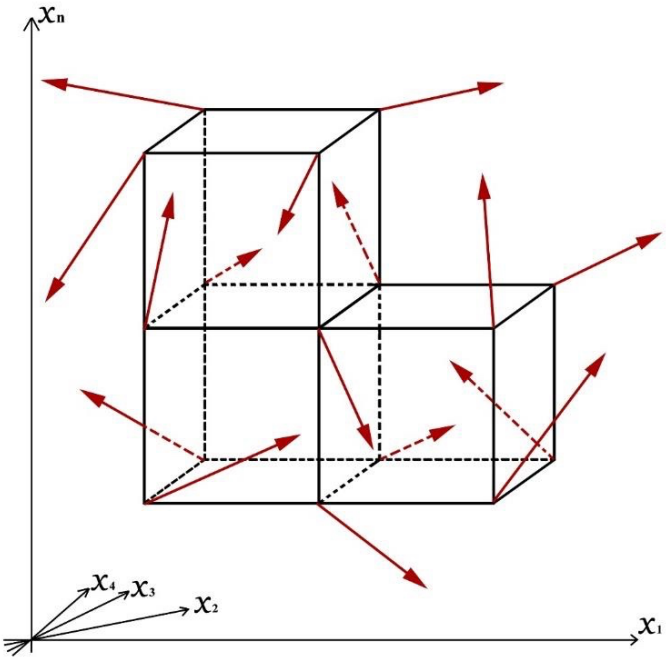
Let us explain. The point $\vec{u}$ belongs to a certain unit n-dimensional cube, at each vertex $\lfloor \vec{u} \rfloor + \vec{k}$ of which a pseudo-random gradient vector is defined. It is because of this definition that term "gradient noise" got its name.

Auxiliary figure:



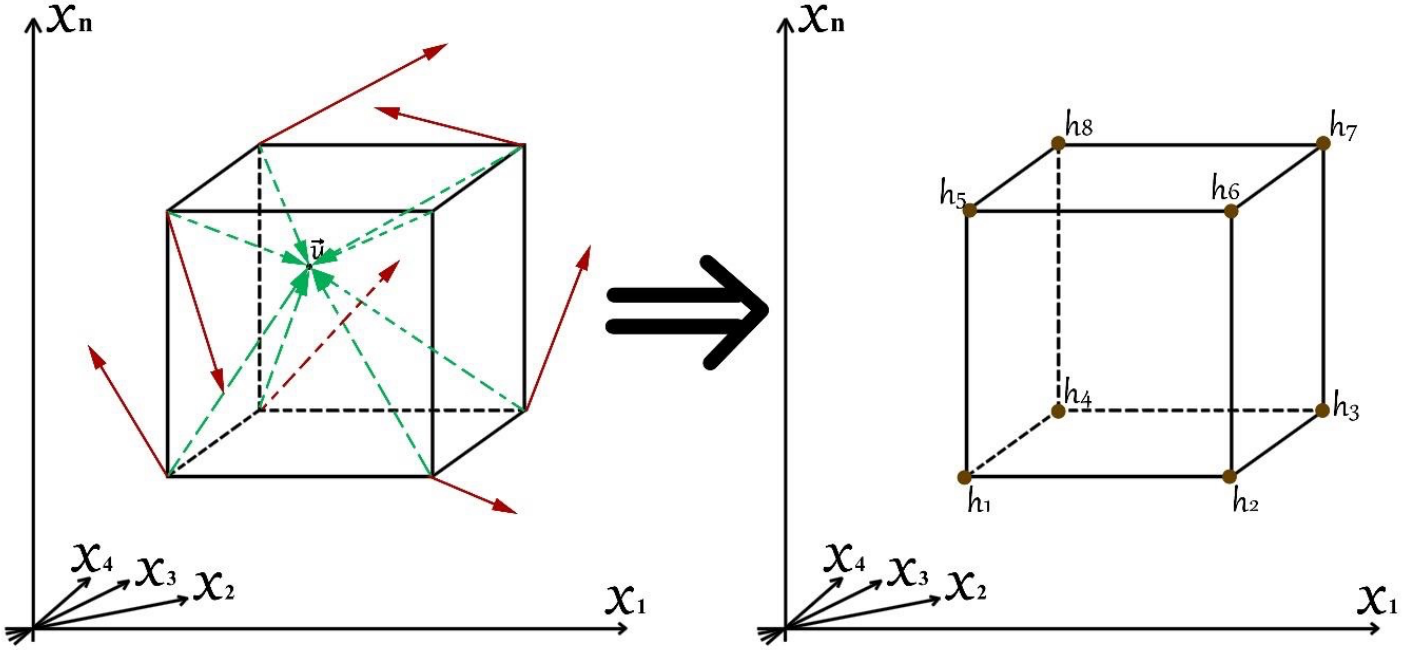
The vertices of the cube (green), the locations of the points $\vec{u}$ and $\lfloor \vec{u} \rfloor$, and the gradient vectors (red) are shown.

Important note. It is important to understand that the gradient vectors at all integer points in the entire space $\mathbb{R}^n$ are fixed. Regardless of $\vec{u}$, each cube "knows" its gradients:



Step 2. To determine the number $h(\vec{k}) = \overrightarrow{grad}(\vec{k}) \cdot (\vec{u} - [\vec{u}] - \vec{k})$ for each such $\vec{k}$.

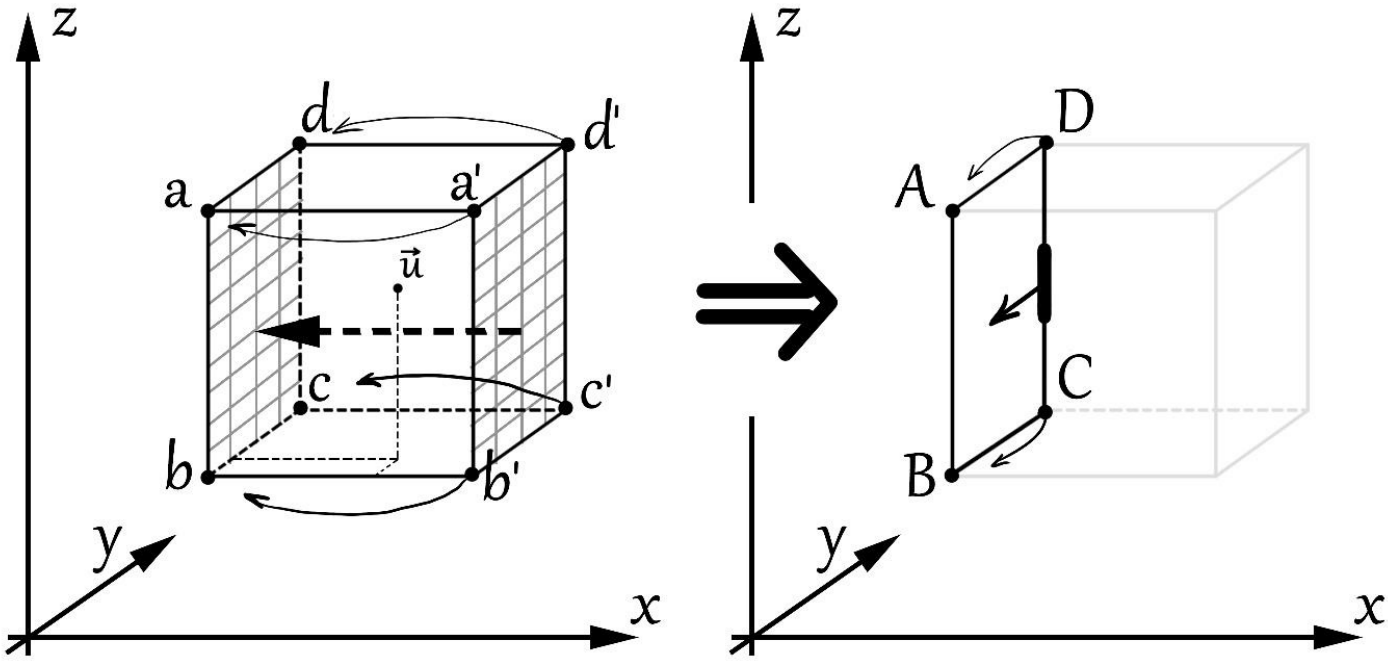
Let us explain. Each vertex of the cube is identified with a number equal to the scalar product of the gradient vector at that vertex and the vector with its origin at that vertex and ending at the point $\vec{u}$. Auxiliary figure:



At each vertex, two vectors (green and red) are multiplied scalarly. After that, each *h* vertex is identified with the result of the corresponding multiplication.

Step 3. Cyclically redefine numbers $h(\vec{k}) = iLerp\left(h(\overrightarrow{k_\iota}), h(\overrightarrow{k_\iota} + \overrightarrow{e_\iota}), coord_i(\vec{u} - \lfloor\vec{u}\rfloor)\right)$ for each i from 1 to n and each vector $\vec{k}$ whose first i coordinates are zero, and the rest are zero or one.

Let us explain. There is a cyclic reduction in the dimension of the n-dimensional cube down to zero. At each i^{th} iteration, two opposite faces of the cube, i perpendicular to the i^{th} coordinate axis, are "glued" together, where the values at the "glued" vertices are interpolated so that the final value h at the "glued" vertex will be closer to the value h at the vertex closest to the point $\vec{u}$ (which is achieved by taking the i^{th} coordinate of the vector $\vec{u} - \lfloor\vec{u}\rfloor$ for interpolation. Auxiliary figure:



This shows how, in the first iteration of the three-dimensional case, the (a, b, c, d) is "glued" to the edge (a', b', c', d'). The result is a two-dimensional cube in which

$$\begin{cases} A = iLerp(a, a', \vec{u}_x - \lfloor\vec{u}\rfloor_x) \\ B = iLerp(b, b', \vec{u}_x - \lfloor\vec{u}\rfloor_x) \\ \dots \end{cases}$$

In the next iteration, the edges (AB) and (CD) will be "glued" together.

Step 4. In summary, the value of Perlin noise at point $\vec{u}$ is taken to be the number $h(\overrightarrow{k_n})$ obtained at the last iteration.

End of definition.

Statement. Perlin noise is a continuous mapping.

We omit a rigorous proof of this statement. The continuity of Perlin noise follows from the continuity of the scalar product with a constant vector and the continuity of the nonlinear interpolation $iLerp$ defined above. It is precisely the properties of pseudo-randomness and continuity that make Perlin noise useful for solving texturing problems.

Let us clarify that the concepts of Perlin noise and the Perlin noise algorithm are, generally speaking, different. Perlin noise is a continuous pseudo-random function that returns a number, while the Perlin noise algorithm is a set of instructions for generating gradient noise. To describe the Perlin noise algorithm, we need to introduce a few more definitions.

Let us choose a point A and two orthonormal vectors $\vec{u}$ and $\vec{v}$ in the space $\mathbb{R}^n$. Let us denote by $L[A, \vec{u}, \vec{v}]$ a two-dimensional plane spanned by the point A and the vectors $\vec{u}$ and $\vec{v}$. Then the mapping

$$T = T_{A,\vec{u},\vec{v}} : \ \mathbb{R}^2 \to \mathbb{R}^n$$

of translating local coordinates (based on $\vec{u}$ and $\vec{v}$) of some point $p \in L[A, \vec{u}, \vec{v}]$ into global coordinates is uniquely defined.

Definition 10. A two-dimensional cross-section (slice) of an n-dimensional Perlin noise by a plane $L[A, \vec{u}, \vec{v}]$ is called a mapping

$$_2P_r^n \ \stackrel{\text{def}}{=} \ P_r^n \circ T \ : \quad \mathbb{R}^2 \to \mathbb{R}.$$

The following definition is introduced to relate Perlin noise values to pixel colors.

Definition 11. Let $p \in \mathbb{I}$. Then

$$color(p) \stackrel{\text{def}}{=} (p, p, p, 1).$$

In this case, it is usually said that $color$ colors the interval $\mathbb{I} = [0, 1]$ in a black-and-white gradient. Note, for $p \in \mathbb{I}$, the color is of the form $(p, p, p, 1)$, when visualized in computer graphics, is a completely visible shade of gray. This is due to the physical structure of color. The $color$ mapping chart is presented:

In article [2] proves that n-dimensional Perlin noise is a mapping onto $\left[-\sqrt{\frac{n}{4}}, \sqrt{\frac{n}{4}}\right]$. Let us redefine Perlin noise as

$$P_r^n(\vec{u}) \stackrel{\text{def}}{=} \frac{2 \cdot \frac{P_r^n(\vec{u})}{\sqrt{n}} + 1}{2}.$$

Then we have surjection $P_r^n : \mathbb{R}^n \to \mathbb{I}$. Hereinafter, we will refer to this mapping as Perlin noise.

Let us return to Perlin noise algorithm. We will show how to obtain a texture of size $a \times b$ pix.

Step 1. Construct the mapping P_r^n.

Step 2. Take its two-dimensional slice.

Step 3. Select the rasterization frequency $t > 1$.

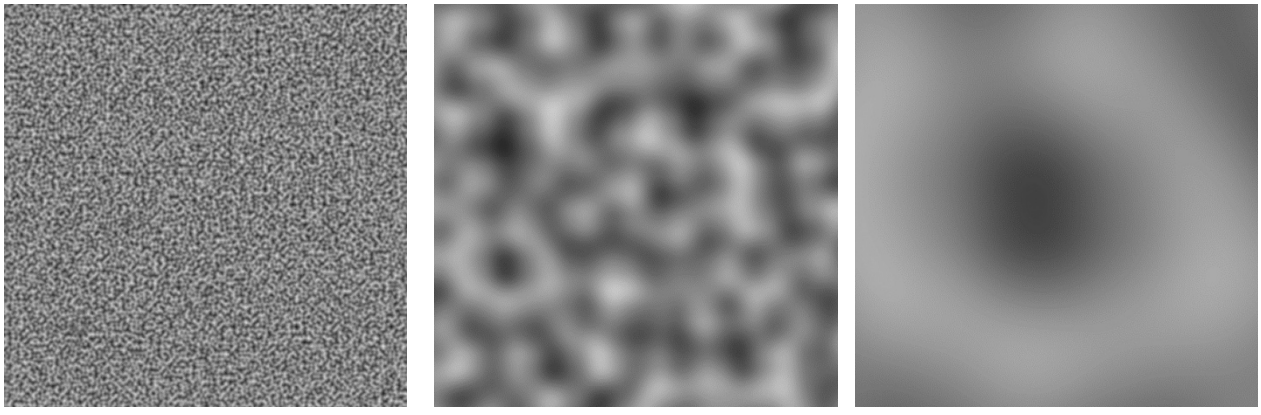
Step 4. Fill the two-dimensional array M of size $a \times b$ as follows

$$M[i][j] \coloneqq color\left({}_2P_r^n\left(\frac{i}{t}, \frac{j}{t}\right)\right).$$

This concludes the description of Perlin noise algorithm.

Here are some examples of gradient noise, obtained using the described algorithm.

$t = 20$ $t = 240$ $t = 1000$



Although the Perlin noise algorithm is described, exactly the P_r^n mapping is used in the development of a computer system for generating textures, as noted in the goal of the work: "creation of a computer system, generating pseudo-random textures using Perlin noise."

Some mappings derived from P_r^n are also suitable for obtaining a gradient noise textures: these derivatives are colored with a black-and-white gradient and then rasterized. These are the derivatives used in the developed computer system.

1.2. Analysis of Texture Generation Software

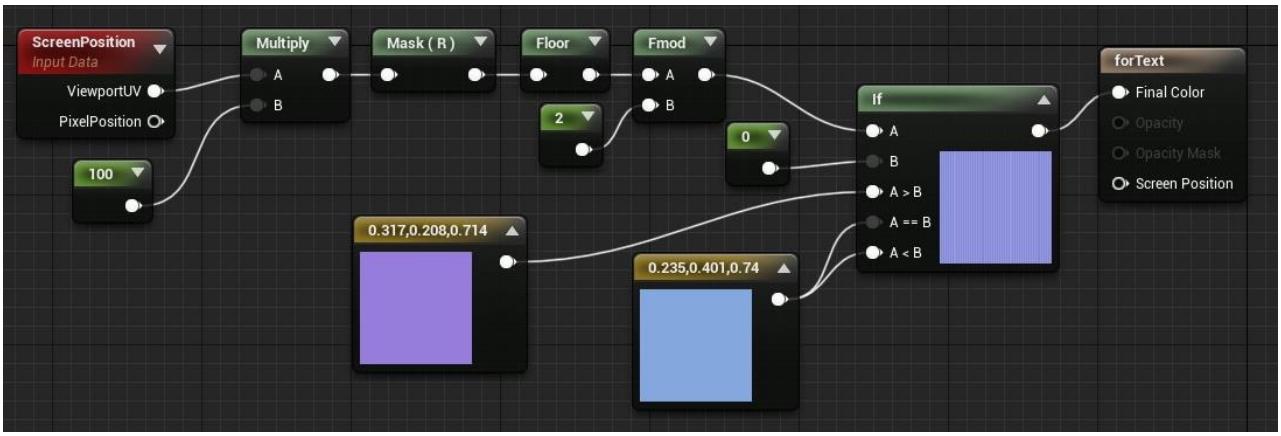
Before we start developing a computer system that generates pseudo-random textures using Perlin noise, let us analyze some existing procedural texture generation software products that are used for industrial-scale texturing tasks.

1. Houdini is a computer system for procedural generation of textures, 3D surface models, visual effects, etc. It is most often used to create complex effects that use mathematical calculations [3].
2. Gaea is a computer system for procedural generation of textures and 3D surface models. It is most often used to create height maps and other textures (including service textures) for surfaces. For example, maps of shades, occlusions/darkening, metallicity, or roughness [4].
3. World Machine is a computer system for procedural generation of textures for surface terrain models. It has very convenient tools in the form of special brushes, with which you can locally change procedural textures with ordinary strokes (computer mouse movements) [5].
4. Substance Designer is a large and powerful computer system that allows you to create "smart textures" for import into another computer system: Substance Painter. The "intelligence" of such textures lies in their "ability" to determine their position on the surface, thereby forming transitions on the edges of textured objects that are "understandable to the viewer." This computer system was used to develop models in cinema: this is how the visual effects of the Iron Man costume were created in Marvel films [6].

The study found that all of the listed computer systems for procedural texture generation have Perlin noise generation tools. To describe such tools, we will define the following concepts.

- A shader is a computer program designed to be executed by graphics card processors (also known as GPU) [7].
- Material. In reality, material is a convenient abstraction of the concept of a shader. Just as the material of a real physical object determines its mechanical, optical, and other properties, material in computer graphics describes the properties of a three-dimensional model's surface: color, luminosity, reflectivity, normal direction, displacement, etc. at each point on the surface.

- Node-based visual programming is the process of creating computer programs (in particular, shaders) in the style of a graphics editor, based on placing nodes on a certain "field" and creating connections between them. Each node is a function that accepts and returns certain data structures (most often vectors). The connection between nodes indicates the transfer of output data from one node to the input of another.
  An example of visual material code:



Each of the computer systems for procedural texture generation discussed is based on the creation of materials and provides the ability to program them visually. In each of these systems, a predefined node acts as a Perlin noise generator.

Houdini. In the Houdini computer system, several nodes that generate Perlin noise are predefined. These are the "Perlin noise", "Original Perlin noise" and "Zero centered Perlin" nodes, which are derivatives of classic Perlin noise. As described in the official documentation [8], each of these nodes has parameters (input data) for roughness and attenuation, but the full functionality of the nodes is not described in the documentation. For a detailed introduction to the tools provided by the Houdini system, you should purchase this system.

Gaea. The Gaea system also provides the user with a predefined "Perlin" node that generates Perlin noise. The official documentation [9] describes all the parameters of this node in detail. These include scale, number of octaves, frequency, ridge sharpness (noise peaks that visually resemble rock peaks when rasterized), and others.

World Machine. In the World Machine computer system, the "Advanced Perlin noise" node is predefined for generating Perlin noise. The official documentation [10] provides a very complete description of all its parameters and their purposes. The use of the word "Advanced" in the node's name is indeed justified: this node provides the ability to very flexibly configure Perlin noise generation — it is possible to set the number of octaves, scale, frequency,

persistence (how the octaves of noise are mixed), steepness (the visual steepness of slopes when rasterized), and much more.

Substance Designer. The Substance Designer computer system has two predefined Perlin noise generating nodes: "3D Perlin noise" and "3D Perlin noise fractal" [11]. The first one is classic Perlin noise and has virtually no parameters except for scale. Of greater interest is the "3D Perlin noise fractal" node. This node has more parameters, including scale, frequency, roughness, detail, and an interesting option to set contrast—an interpolation function different from $iLerp$.

Each of the described nodes that generate Perlin noise is only a mathematical mapping that returns a color at each point in a n-dimensional space (most often for $1 \leq n \leq 4$). To obtain the final texture, the user must build the corresponding material based on this node and apply this material to a plane (for rasterization) or a three-dimensional surface (for height extrusion) in order to "observe" the changes when the parameters of the predefined node used are changed. After that, the user must export the material as an image to obtain the final texture.

Thus, in the computer systems studied, it is not enough to use only predetermined nodes to obtain the final texture of gradient noise based on Perlin noise. To obtain such a texture, the user must understand the structure of materials, know at least the basics of programming, and have sufficient knowledge of complex tools of the selected computer system. Such "demanding" nature of the systems studied in terms of specialized knowledge is their main drawback.

However, it should be noted that a user with a deep understanding of materials, in-depth knowledge of programming and computational geometry as applied to computer graphics, and sufficient knowledge of the interface and tools of the system used, is capable of generating much more complex textures (and materials) than gradient noise. This is a significant advantage of the computer systems studied.

In summary, let us briefly list the advantages and disadvantages of the four procedural texture generation software products studied.

The advantages include:

- a wide range of capabilities;
- a convenient and multifunctional interface;

- a large number of built-in algorithms (most often in the form of nodes);
- versatile export options: available formats include textures, 3D models, animation effect files, materials, and C++ classes.

The disadvantages include:

- the need for large amount interdisciplinary knowledge for use;
- complex interface and organization (logic) that require learning;
- these computer systems are pair.

When setting a task for general texture generation, when it is necessary, for example, to "find" a certain texture that is not an object of a known texture class (for example, a noise class), the use of the software solutions studied may be justified. However, when setting the task of obtaining gradient noise based on Perlin noise, choosing these solutions may not be rational.

Based on the results of the analysis of existing computer systems for procedural texture generation, we can conclude that there is a need for a computer system that generates textures using Perlin noise, provides a simple and intuitive interface, offers a convenient viewing control system, and does not require the user to have in-depth specialized knowledge.

1.3. Choosing an Environment for Developing a Computer System

A computer system that generates and displays textures in real time requires the computer to perform a large number of operations per second. For example, the 2048×2048 pix texture generation program in real time with using an algorithm requiring 10 calculations per pixel, at a rendering speed of 100 frames per second, requires the computer to perform more than four billion operations per second ($2048^2 \cdot 10 \cdot 100$). When the program works with four-dimensional vectors, operations that are more complex than arithmetic calculations, and complex algorithms, the number of operations required per second can increase to several hundred billion. For a computer's central processing unit, such a number of operations per second (tens of times greater than what is critically possible) is unfeasible.

When writing a program with a heavy computational load or a program that performs graphics calculations, writing shaders is almost always the most rational solution. It is precisely for such tasks that the video card of a modern computer is equipped with thousands of processors. Moreover, the video card is the "graphical heart" of the computer: when the central processor receives a command to render, it still sends this command to the video card, which takes time. Shaders are executed directly by the video card, so when rendering commands are executed, there is no such "leakage" of time to transfer these commands from one place to another.

To develop a computer system that generates pseudo-random textures using Perlin noise, it was decided to use shaders. The main criteria for choosing a development environment were: first, the availability of built-in shader development tools, and second, the convenience of these tools when writing, compiling, and assembling shaders. Shader assembly refers to the inclusion in the main program of logic code for loading compiled shaders onto the video card and using them.

General-purpose development environments (such as Eclipse or Microsoft Visual Studio) do not typically include built-in tools for developing shaders. However, there are plugins (add-ons) that extend the functionality of development environments and enable you to write, view, and compile shaders within these environments. This solution may be reasonable for some projects because it allows you to control all stages of development. However, it requires the developer to have in-depth knowledge of shader programming and assembly. Since there are alternative solutions that offer visual shader programming tools and automatic compilation and assembly, we will not dwell on the option of using general-purpose development environments.

The next "level" of development environments includes game engines: Unity, Unreal Engine, CryEngine, etc. Such development environments use an exclusively object-oriented programming paradigm. A significant advantage of engines is that they contain a large number of predefined classes responsible for such fundamental aspects of computer modeling and 3D graphics as lighting, physics engines, 3D models, materials, etc. Most engines offer programmers a built-in 2D editor for creating graphical user interfaces and a 3D editor for working with three-dimensional objects. Moreover, some of these development environments offer a built-in material editor that allows you to design shaders using nodes.

Let's choose the Unreal Engine for developing a computer system that generates textures using Perlin noise. This development environment was chosen because of its following significant advantages:

- it has a built-in 3D scene editor;
- it has a built-in graphical user interface editor;
- it allows for visual programming of both the main part of the program and materials (shaders);
- many classes responsible for fundamental concepts in computer modeling and 3D graphics are predefined;
- there are automatic project assembly tools capable of assembling the main part of the program, compiled shaders, resource files (e.g., images of graphical user interface elements), etc.

Conclusion on Chapter 1.

The creation of a computer system that generates pseudo-random textures using Perlin noise, with an intuitive interface and a simple texture viewing system that does not require the user to have specialized knowledge (in programming, 3D graphics, computational geometry, etc.) is currently relevant and practically justified. It is advisable to use the Unreal Engine to develop such a system.

CHAPTER 2. CREATION OF A COMPUTER SYSTEM FOR TEXTURE GENERATION

2.1. Application of Perlin Noise

Let $n \in \mathbb{N}, n \geq 2$, and let the mapping be given:

$$f: \quad \mathbb{R}^n \to [0,1].$$

By rasterization of the mapping f we will henceforth mean the creation of a texture T of size 2048×2048 pixels such that for $i = \overline{0,2047}$ and $j = \overline{0,2047}$ the following is true:

$$T[i][j] = color\left(f\left(\frac{i-1024}{400}, \frac{j-1024}{400}, 0, 0, \dots, 0\right)\right).$$

Recall that classical Perlin noise is a mapping:

$$P_r^n: \quad \mathbb{R}^n \to [0,1],$$

which, when rasterized, produces a texture of the form:

When generating more realistic gradient noise textures for rasterization, we use not Perlin noise itself, but some of its derivatives. In order for the computer system, we are developing, to be used to solve a larger class of texturing problems, let us construct a mapping P, derived from classical Perlin noise.

For convenience, let us denote the three-dimensional classical Perlin noise by P_r:

$$P_r \stackrel{\text{def}}{=} P_r^3: \quad \mathbb{R}^3 \to [0,1].$$

Recall that the index r just denotes a number and determines the pseudo-random gradient used in the calculation of P_r^3.

Let $n \in \mathbb{N}$, and $\{k_n\}$ be a sequence of numbers. Let us define a mapping F which is a certain "layering" of several classical Perlin noises on top of each other:

$$F(\vec{u}) \stackrel{\text{def}}{=} \frac{\frac{1}{k_1}P_1(k_1 \cdot \vec{u}) + \frac{1}{k_2}P_2(k_2 \cdot \vec{u}) + \cdots + \frac{1}{k_n}P_n(k_n \cdot \vec{u})}{\frac{1}{k_1} + \frac{1}{k_2} + \cdots + \frac{1}{k_n}}.$$

Note. Mapping F is a mapping onto $[0, 1]$ and is therefore applicable for rasterization.

We will call the number n *the octave number* of the mapping F, and the numerical sequence $\{k_n\}$ *the sequence of octave mixing coefficients*. For use in the computer system, we plan to set $\{k_n\}$ in one of the following ways:

- $k_n = 1$,
- $k_n = n$,
- $k_n = n^2$,
- $k_n = 2^{n-1}$.

Depending on the selected number of octaves and the sequence of their mixing coefficients, different textures will be obtained during rasterization F.

Example 1.

$$k_n = n^2$$

$n = 1$

$n = 10$

Example 2.

$$n = 4$$

$k_n = n^2$ $k_n = 2^n$


Let $p, s \in \mathbb{R}$. We use another classical Perlin noise for uneven distortion of the coordinate space. More formally, let us introduce the mapping:

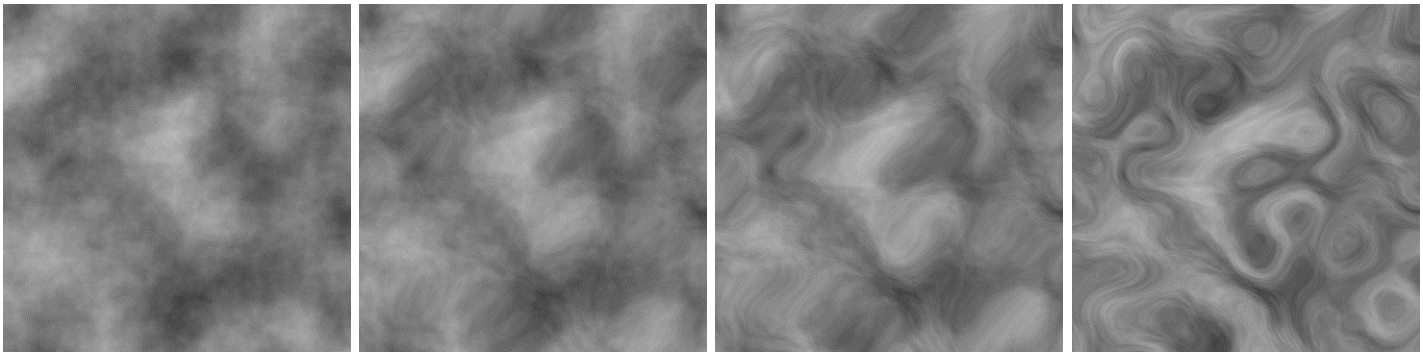
$$U\colon \mathbb{R}^3 \to \mathbb{R}^3,$$

$$U(\vec{u}) \stackrel{\text{def}}{=} \vec{u} + p\left(2 \cdot P_{-1}\left(\frac{\vec{u}}{s}\right) - 1\right)\frac{\vec{u}}{|\vec{u}|}.$$

This mapping lengthens or shortens the vector $\vec{u}$ by an amount not exceeding the value of the parameter p, depending on the value of the noise P_{-1} at the point $\frac{\vec{u}}{s}$.

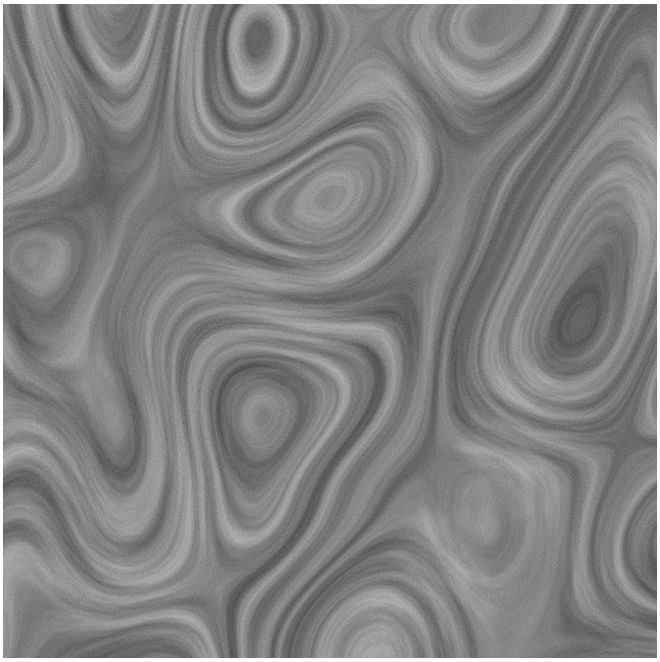
Here are some examples of rasterization of the mapping $F \circ U$ for fixed values $n = 10$, $k_n = n^2$, $s = 1$, and for

$p = 0$ $p = 1$ $p = 2$ $p = 4$



This distortion of coordinate space allows us to achieve interesting results. For example, we can obtain a malachite texture with

$$p = 20$$



Recall the mapping from chapter 1:

$$Q(t) = t^3 \cdot \big((6t - 15)t + 10\big).$$

To construct the final mapping P, derived from classical Perlin noise, let us introduce another mapping:

$$H_{\vec{u}} : \quad \mathbb{I} \to \mathbb{I},$$

$$H_{\vec{u}}(h) \stackrel{\text{def}}{=} \frac{1}{2} + Q^3\big(P_{-2}(\vec{u})\big) \cdot \left(h - \frac{1}{2}\right),$$

$$\text{where } Q^3 = Q \circ Q \circ Q.$$

Such a mapping takes values "close" to h at points where the noise P_{-2} takes values greater than $1/2$, and values "close" to $1/2$ at points where the noise P_{-2} takes values less than $1/2$.

Definition 12. Let $a, b \in \mathbb{R}$. The function

$$lerp(a, b, t) \stackrel{\text{def}}{=} a + (b - a)t$$

is called a linear interpolation function.

Note. Such a function is obviously an interpolation of the system of points $\{(0,1),(1,b)\}$, see Definition 7, Section 1.2.

To parameterize the mapping $H_{\vec{u}}$ we will redefine it:

$$H_{\vec{u}}(h) \stackrel{\text{def}}{=} lerp(h, H_{\vec{u}}(h), t), \text{ where } t \in \mathbb{I}.$$

Such a redefinition allows choose the "degree of impact" of the above mapping $H_{\vec{u}}(h)$ on the value of the number h.

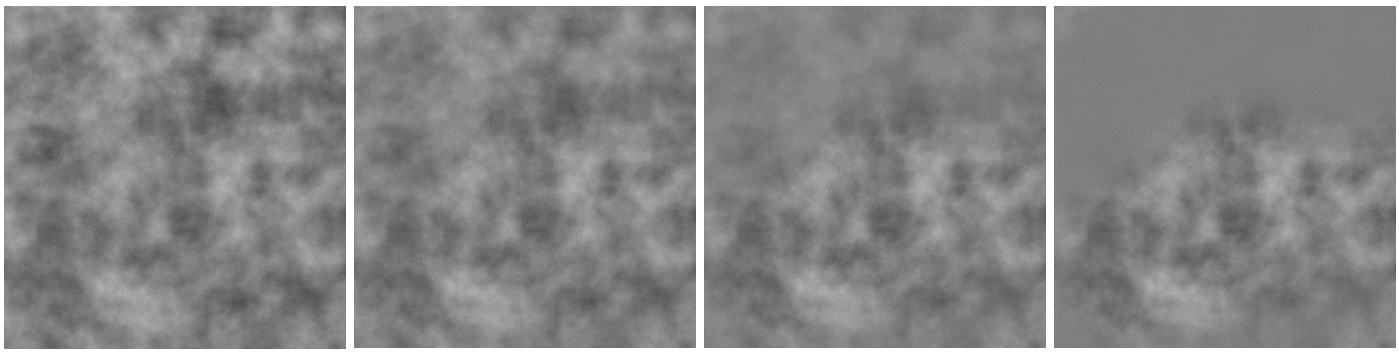
Here are some examples of rasterization of the mapping $H_{\vec{u}}\big(F(\vec{u})\big)$ for parameter values $n = 10,\ k_n = 2^n$.

Example 1.

$t = 0$ $t = 0.33$ $t = 0.66$ $t = 1$

Example 2.

$t = 0$ $t = 0.33$ $t = 0.66$ $t = 1$



In summary, we define our derivative of the classical Perlin noise mapping P as follows:

$$P(\vec{u}) \stackrel{\text{def}}{=} H_{\vec{u}}\Big(F\big(U(\vec{u})\big)\Big).$$

In reality, the following rasterization occurs in the developed computer system:

$$T[i][j] = color\left(P\left(\frac{i-1024}{scale} + xOffset,\quad \frac{j-1024}{scale} + yOffset,\quad z\right)\right),$$

$$\text{for } i = \overline{0,2047}, j = \overline{0,2047}.$$

$z, scale, xOffset$ and $yOffset$ are entered as parameters.

Let us briefly list all the parameters whose values are involved in constructing the final mapping P, derived from several classical Perlin noises, and the parameters used in the final rasterization of this mapping.

List of parameters affecting the final texture:

- n — number of noise octaves,
- $\{k_n\}$ — sequence of mixing coefficients,
- p — distortion strength,
- s — distortion scale,
- t — degree of "attenuation",
- z — height of the two-dimensional slice,
- $xOffset, yOffset$ — global coordinate offsets,
- $scale$ — main scale.

2.2. Key Features of the System

An analysis of existing software solutions for generating gradient noise based on Perlin noise has shown that it is advisable to create a solution that would compensate for certain shortcomings of the computer systems discussed above.

First, we need to determine the functions that the developed system should perform. So, first and foremost it must:

- operate in real time,
- perform customizable texture generation,
- dynamically display 2D and 3D models of the generated texture,
- export the final texture.

The functions listed above represent the basic capabilities of a computer system for generating pseudo-random textures.

Next, we need to determine the requirements for the system under development. To do this, we should take into account the above-mentioned shortcomings of existing computer systems so that the developed system can compensate for them.

So, our computer system that generates pseudo-random textures using Perlin noise should be distinguished by the following features:

- the absence no need for specialized knowledge (about the Perlin noise generator, programming, shaders, etc.),
- instant texture generation based on the set parameters,
- intuitive and easy-to-use graphical interface,
- informative texture viewing system,
- simple instant export system.

The compiled list of functions and system requirements can help in developing the structure, designing the user interface, and organizing the internal logic of the program.

2.3. Interface Overview and Recommendations for Users

This paragraph is devoted to describing the created system.

System name: Generica

Logo: 

Link: https://github.com/rayskiy7/generica/releases/download/v1.0.0/system.zip

Executable file: "Генератика.exe"

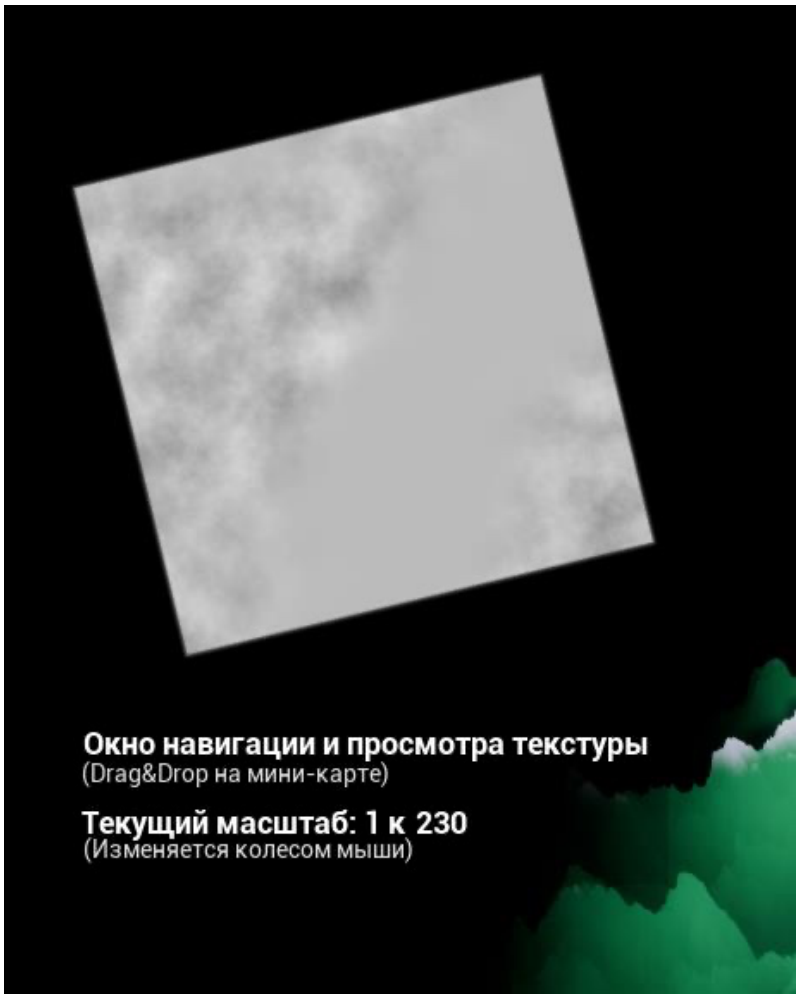
The developed computer system is designed in a "single window" style. The user works with a fixed graphical interface. Let us highlight and number all the key areas of this interface:

We will describe each key area of the interface, its elements, and their purpose.

The navigation and texture viewing area is located at the top left (1). There are:

- the generated texture (hereinafter: mini-map),
- user tips,
- the current scale.

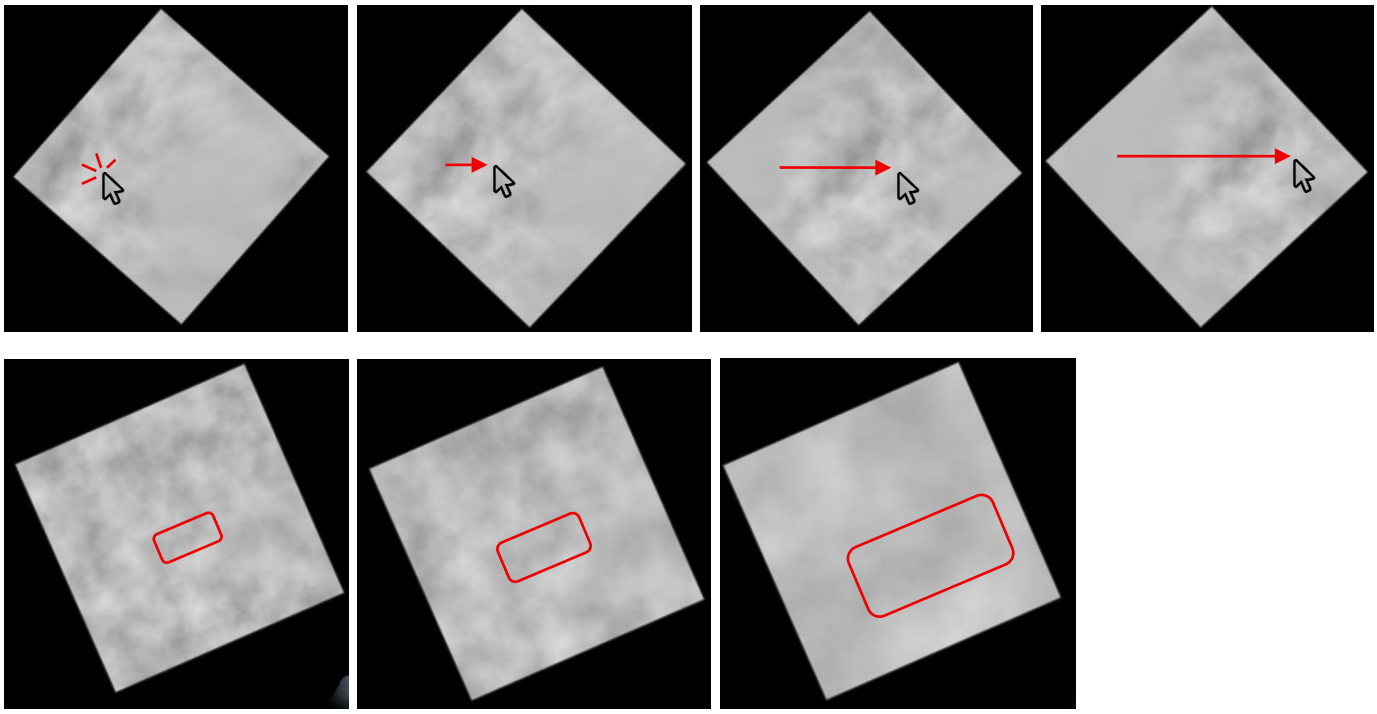
The mini-map is designed to control navigation. Drag-and-drop of the mini-map performs a shift the global coordinate system in accordance with the direction of mouse movement.



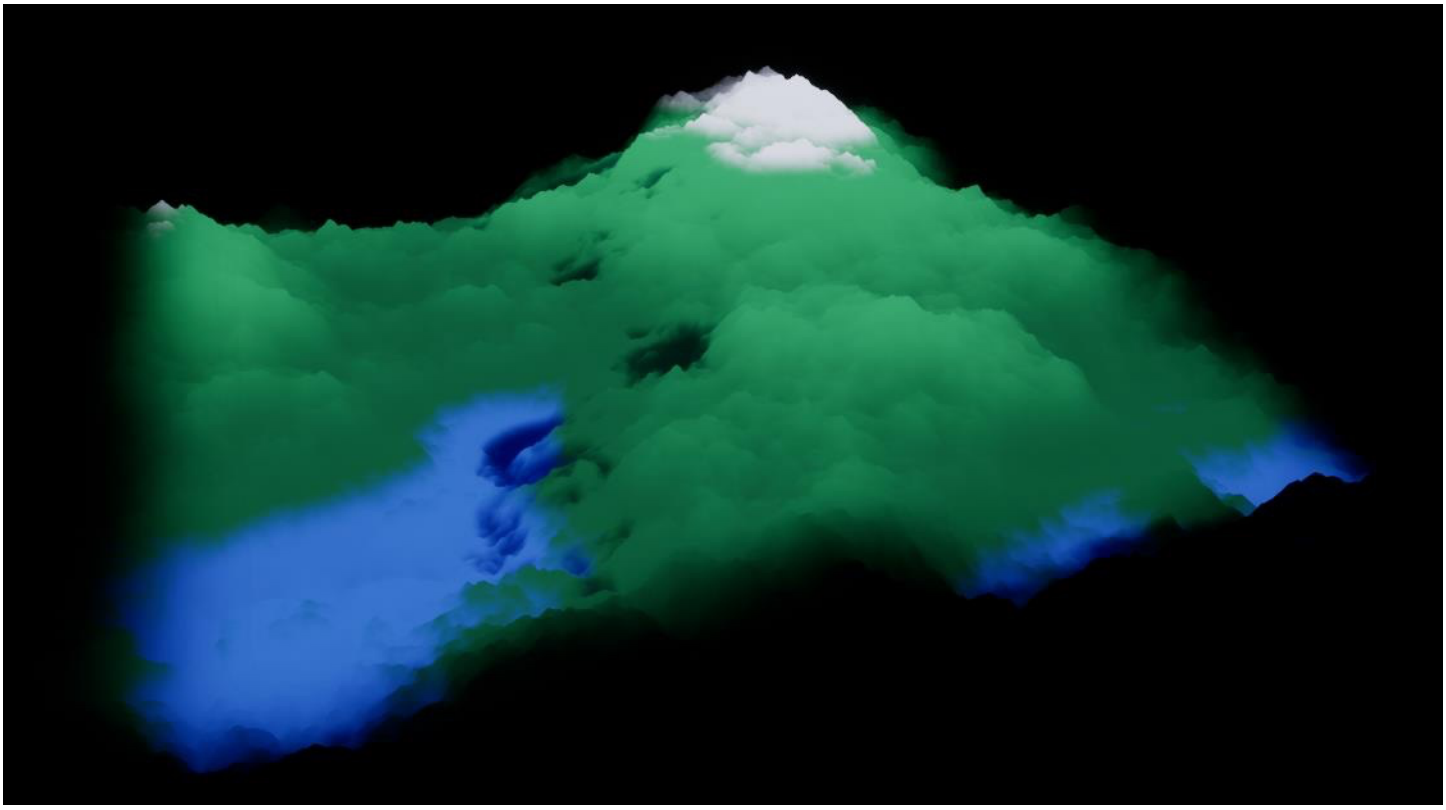
Rotating the mouse wheel changes the global scale parameter value. This increases or decreases the rasterization frequency, resulting in a visual "reduction" or "enlargement" of the texture.

Note. The system was initially developed in Russian, so the text elements of the interface are presented in it.

Below are examples of dragging to the right and rotating the mouse wheel (the red elements indicate what is happening):



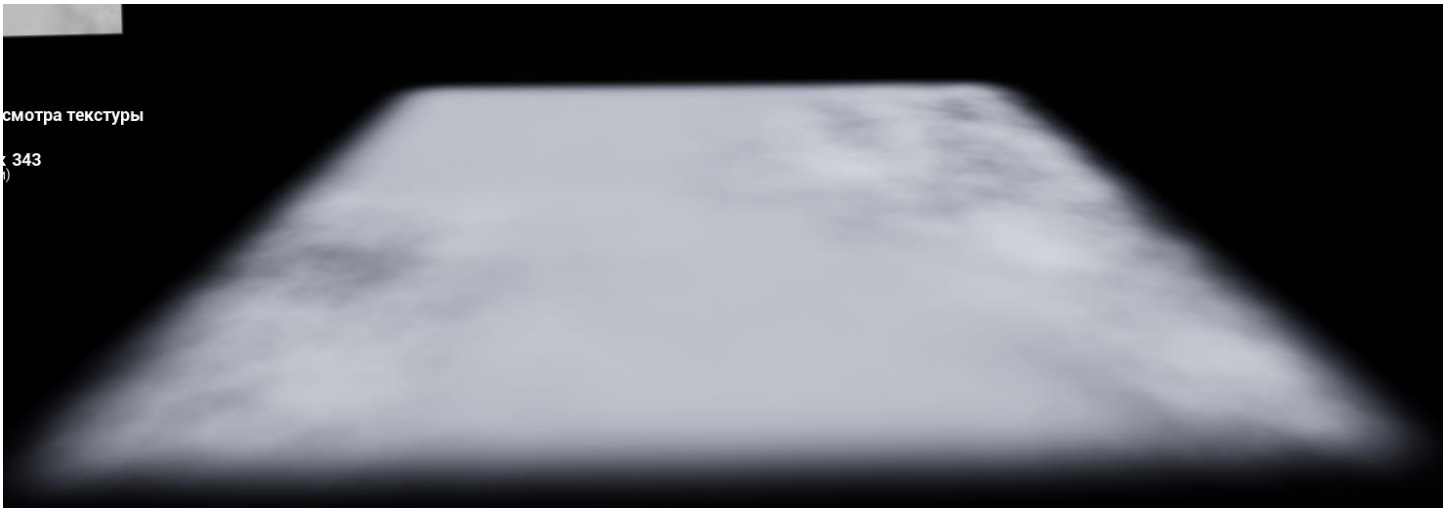
The main viewing area (2) is located in the center of the graphical interface:



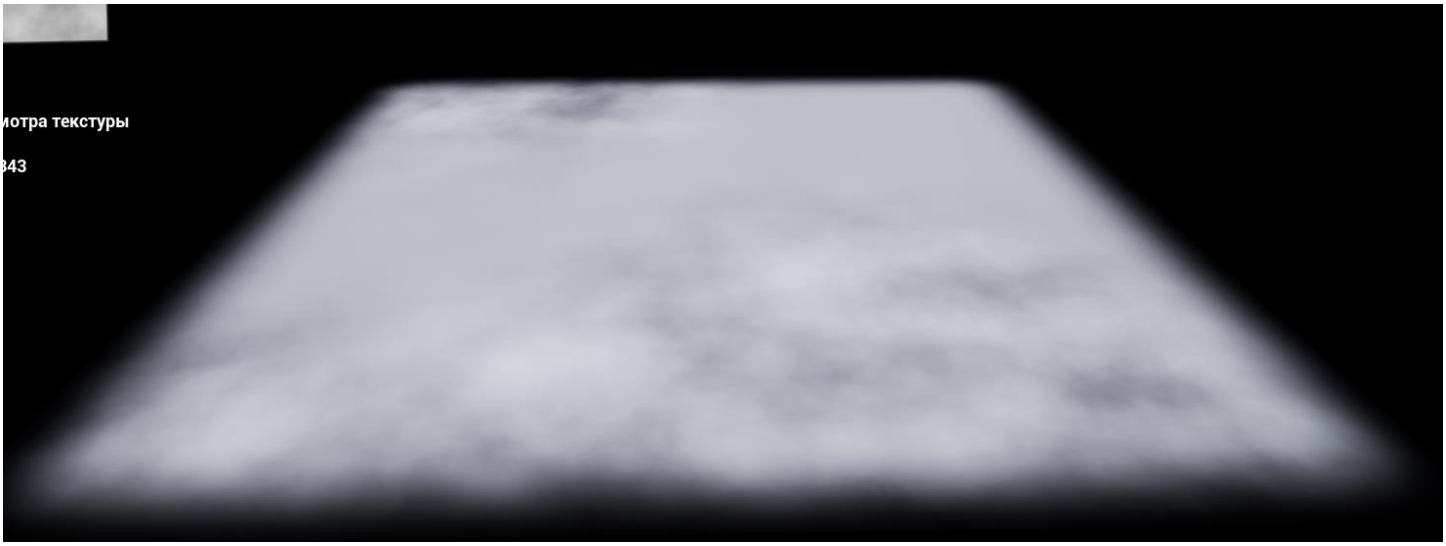
This area displays a three-dimensional model of the surface, which in a sense "repeats" the texture. It is important to understand that such a surface uses the generated texture as a height map, i.e., an image in which lighter pixels are interpreted as "higher" points and darker pixels as "lower" points.

This is what the surface would look like if it were simply "painted" with the colors of the texture.

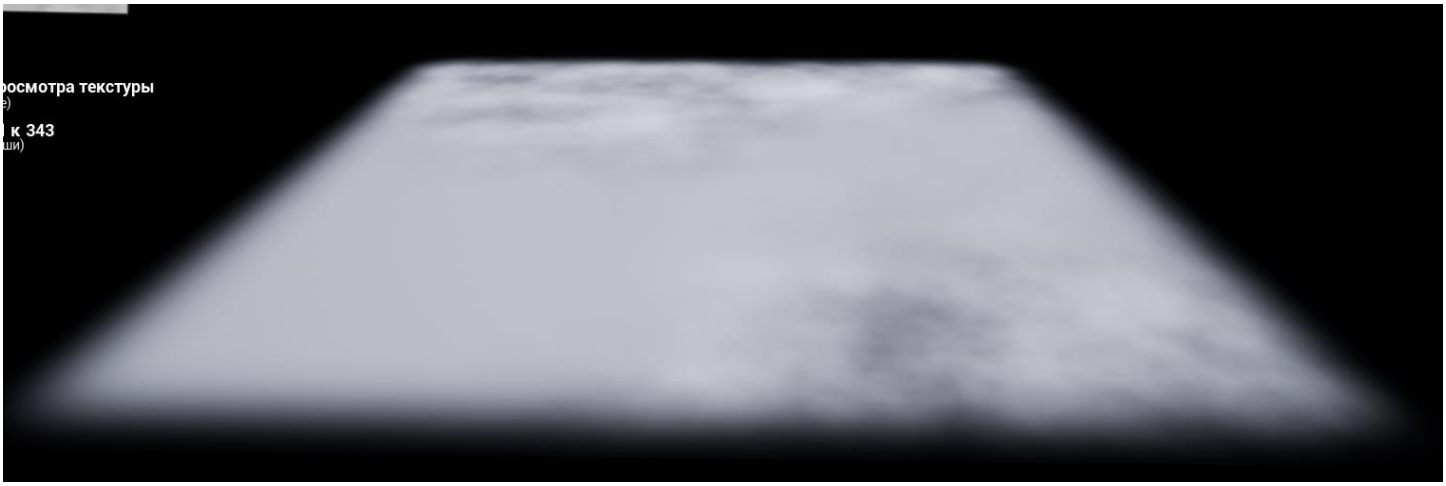
Front view:



View from the right:

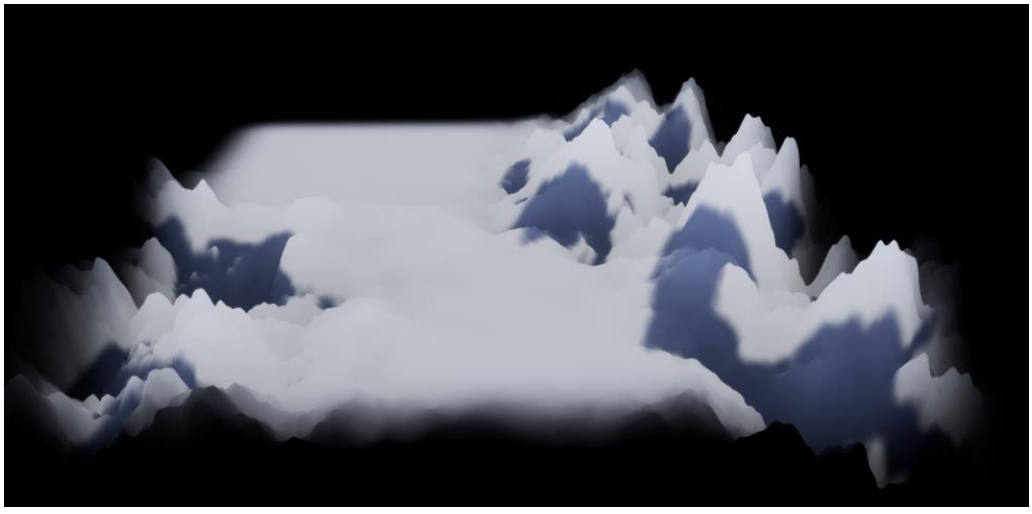


View from the left:

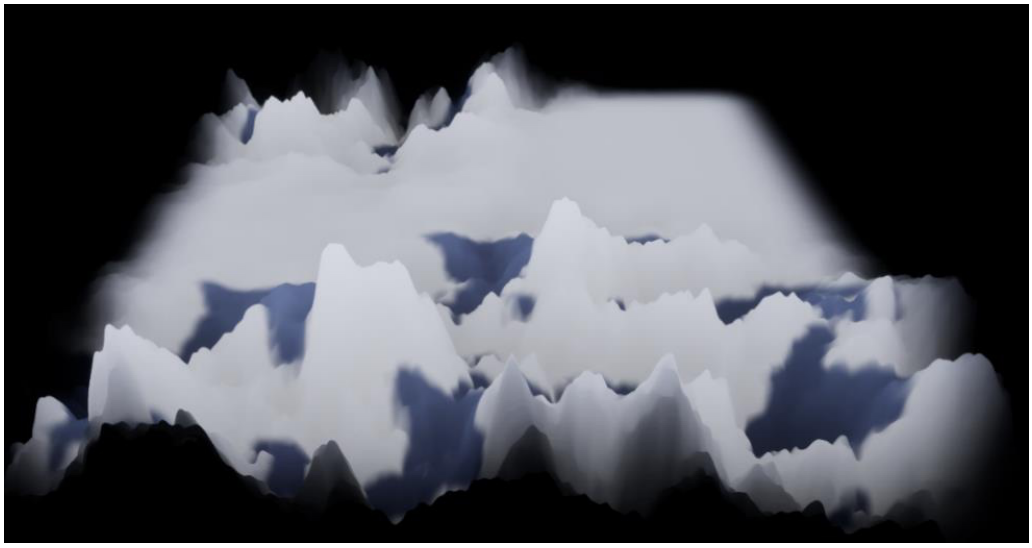


And this is what the same surface looks like, but "understanding" the texture as a height map, i.e., which is "higher" where the texture is lighter and "lower" where the texture is darker.

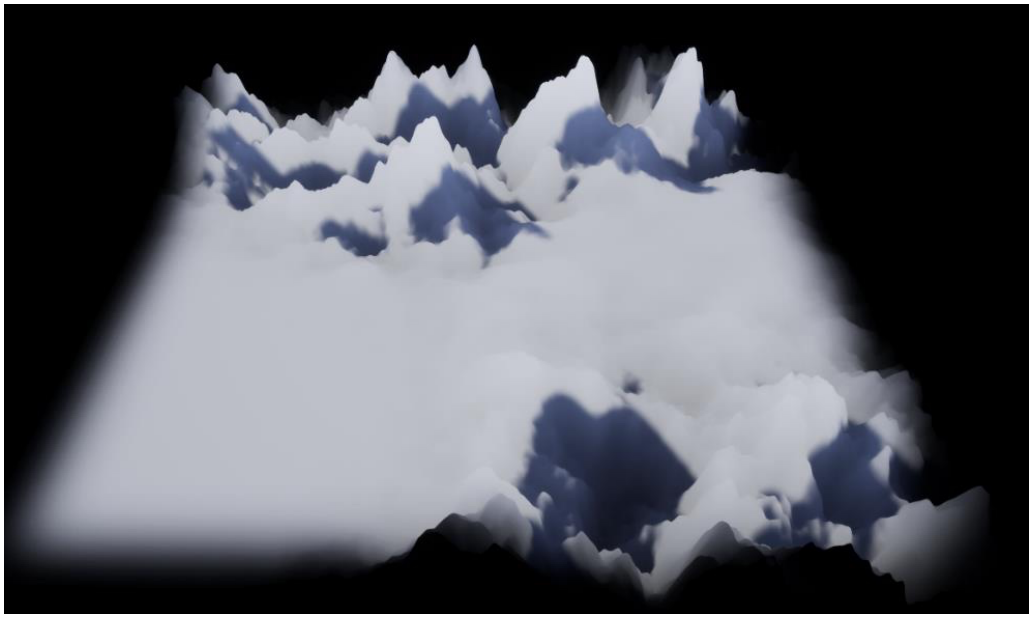
Front view:



View from the right:

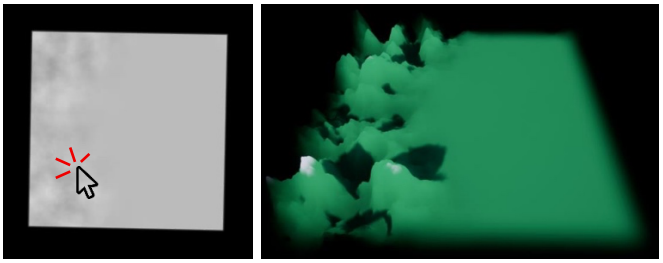


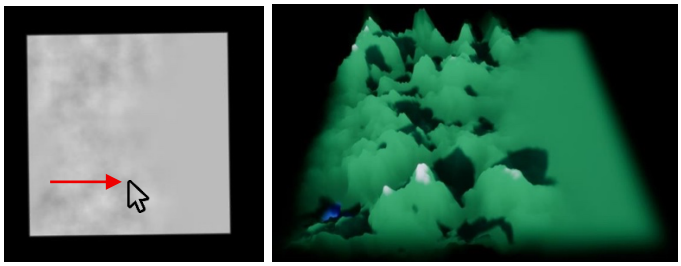
View from the left:

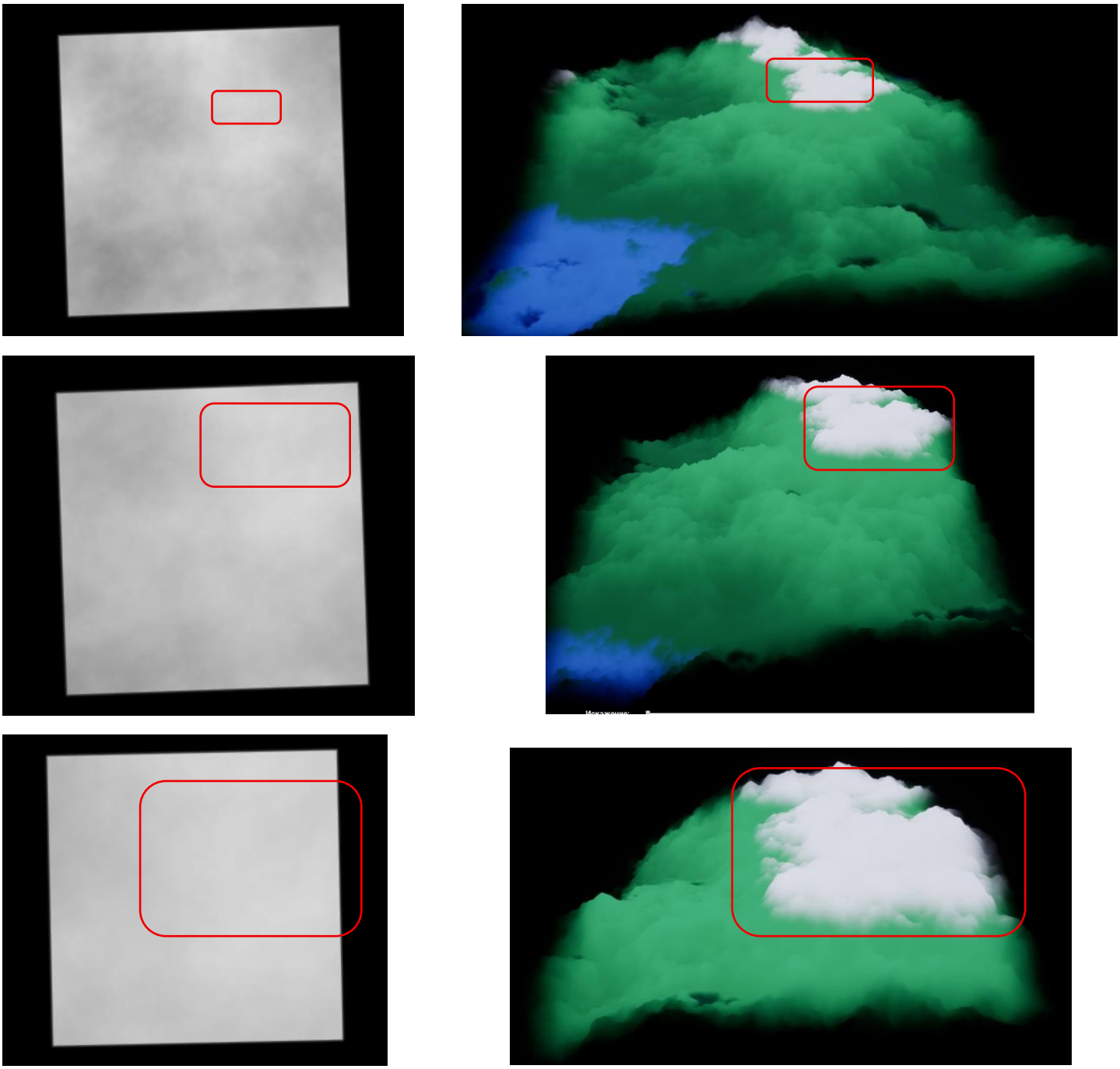


The "three-dimensionality" of the model becomes "tangible" thanks to a light source that exists in the software three-dimensional space of the developed system.

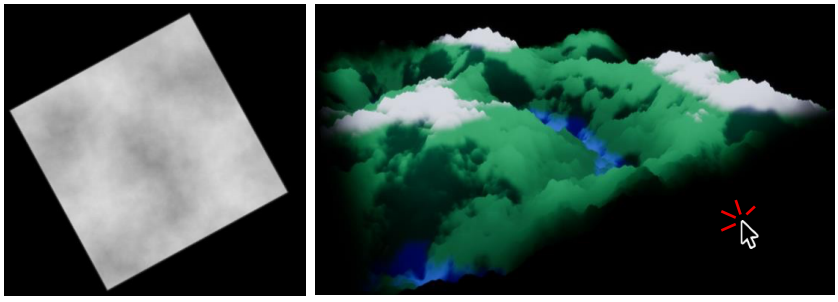
The main purpose of the three-dimensional model in area (2) is to provide the user with a better "understanding" of the texture. For example, let's show how the mini-map operations described above are reflected in the three-dimensional surface model.

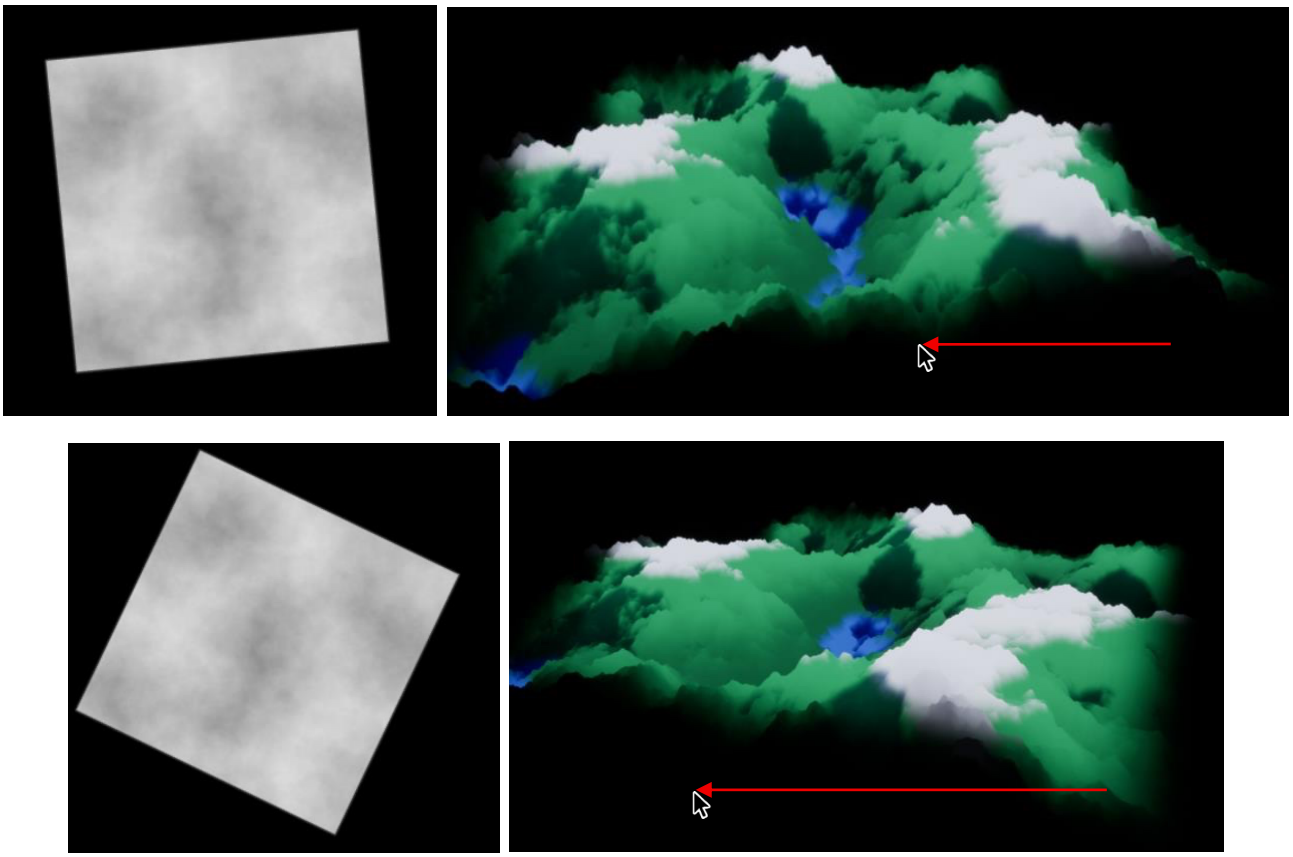




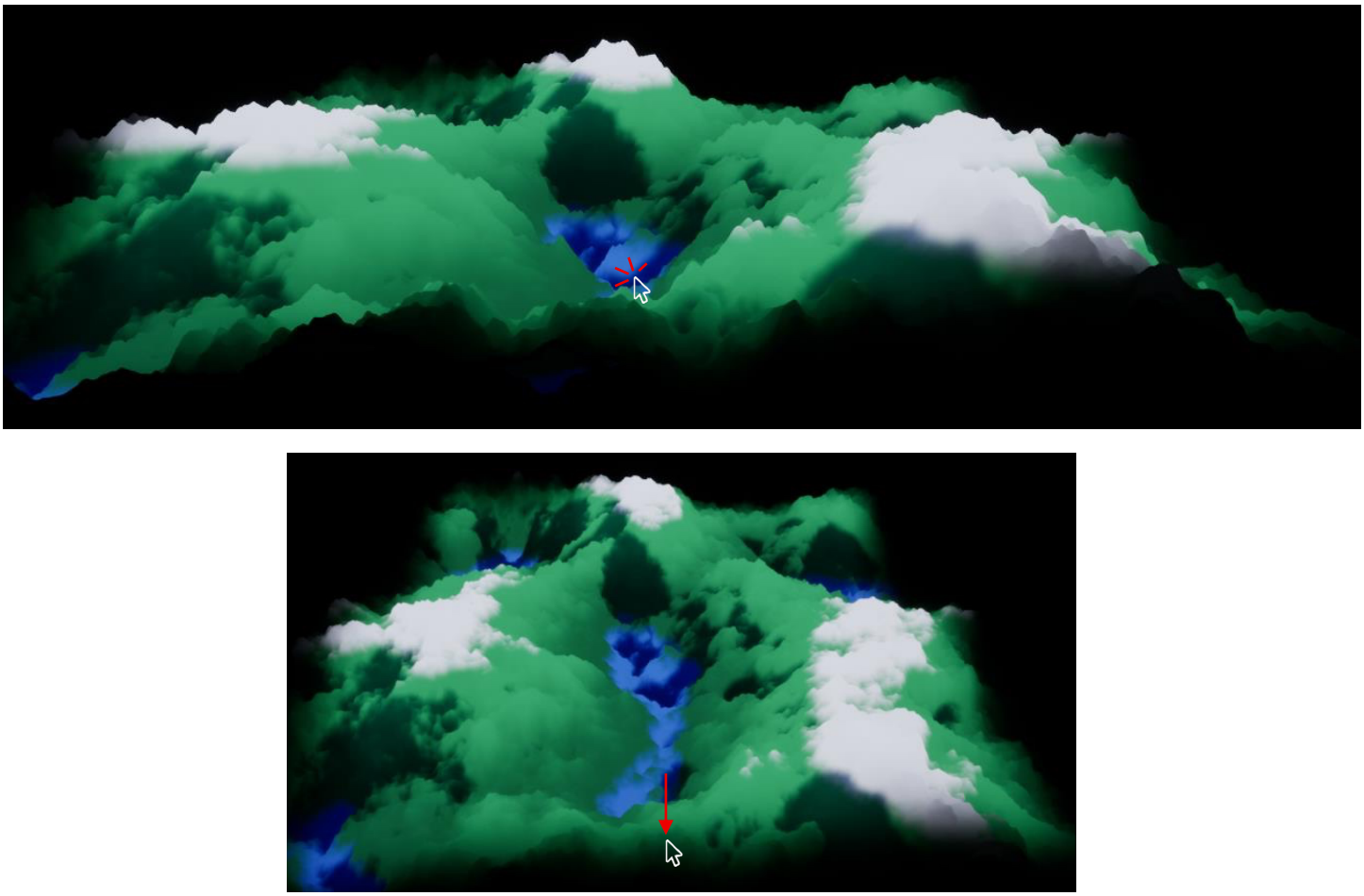


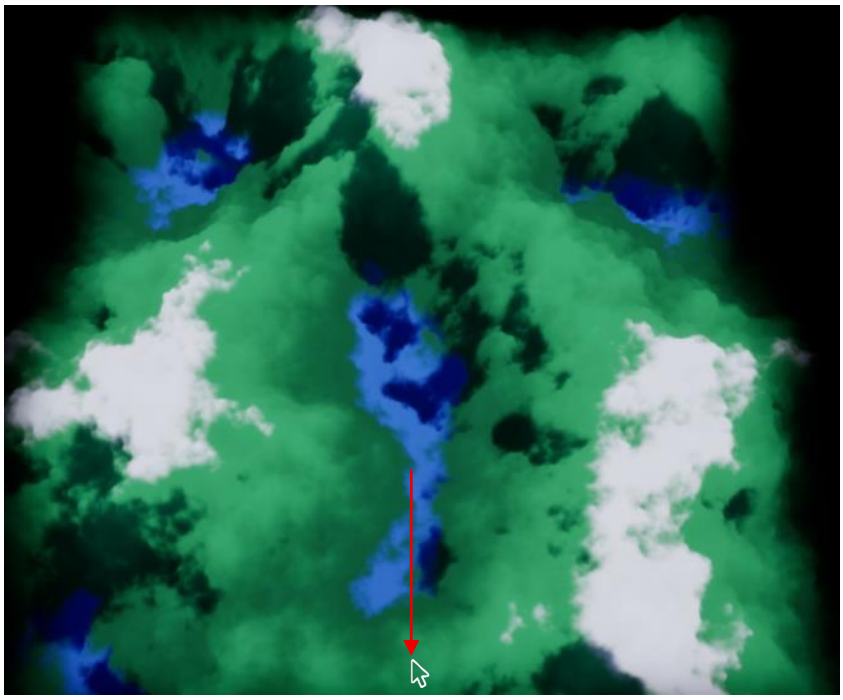
In addition, the central area (2) is used to control the rotation of the surface. The drag operation used in area (2) performs an intuitive rotation of the surface "following" the direction of the mouse movement.



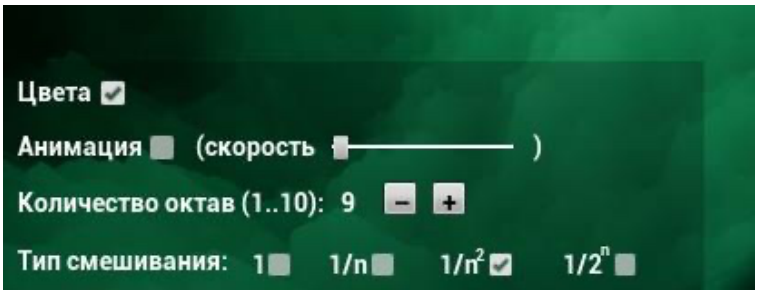


Another example:





In left lower corner of the developed computer system, there is a "switch panel" (3):

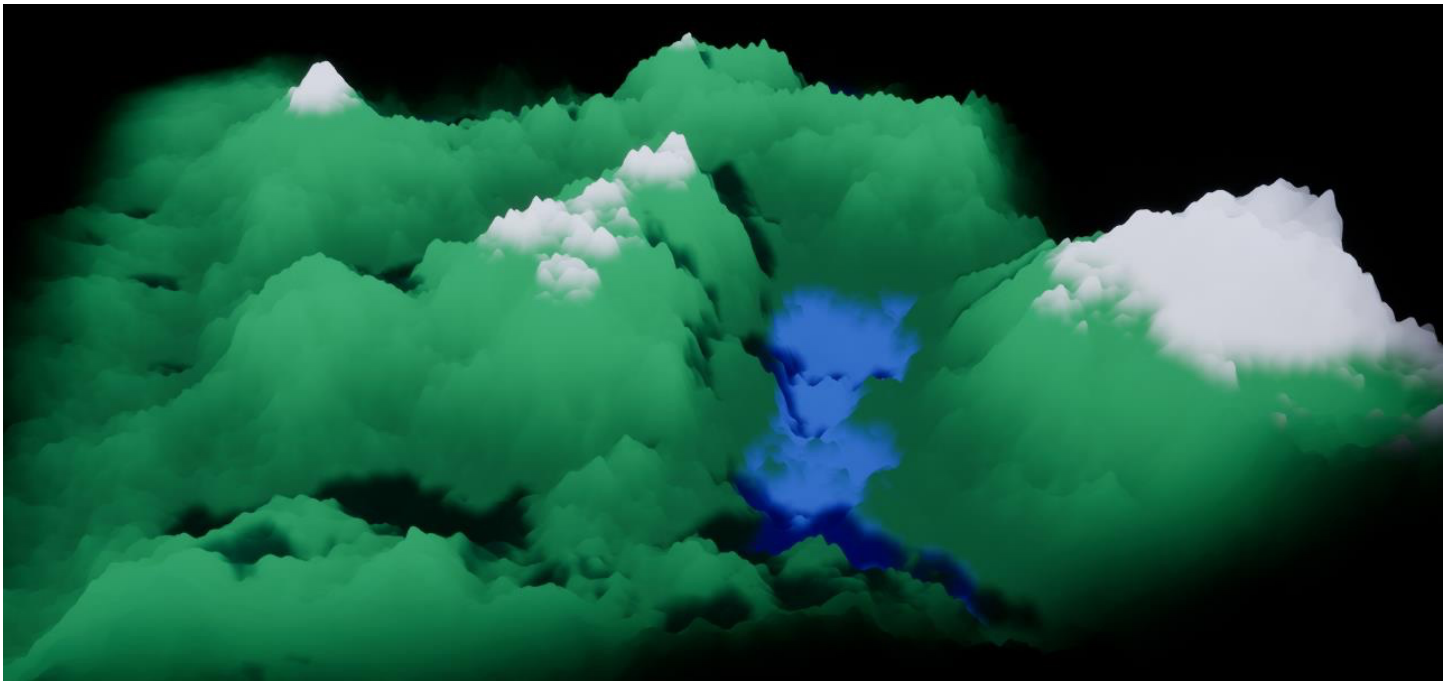


In the first row of this panel, you can turn the "Colors" checkbox. It is responsible for coloring the three-dimensional model located in area (2). It should be noted that the value of this switch does not affect the final texture, only the three-dimensional model of the surface is colored.

The main purpose of this coloring is to enhance the "clarity" of the three-dimensional model. The user is offered to use fixed coloring of the surface in 3 colors: blue, green, and white.

- The lowest areas of the surface are colored **blue**, which symbolizes sea level.
- **Green** is used to color areas of medium height, symbolizing land covered with grass.
- The highest areas of the surface are colored white, symbolizing snow lying on mountain peaks.

An example of a three-dimensional model of the same texture with the "Colors" switch turned off and on, respectively:

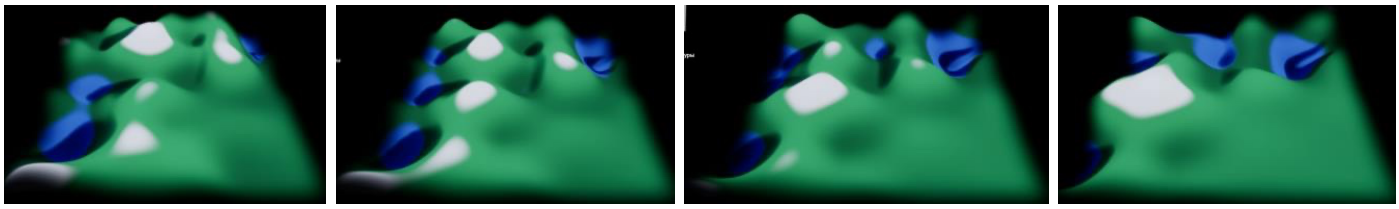


Functionality that improves the perception of textures in texture generation systems is important. In some procedural texture generation systems, performing such coloring requires programming skills and an understanding of the basics of shaders.

The second row of the "switch panel" (3) contains the "Animation" switch, which, when active, causes arbitrary (i.e., pseudo-random) continuous movement of the global coordinate system at a speed determined by the slider in the same row. This forces the texture to "change smoothly." This functionality of the developed computer system is designed to automate texture changes—the user does not need to perform any actions.

An example of a three-dimensional model changing over time, with the "Animation" switch active:

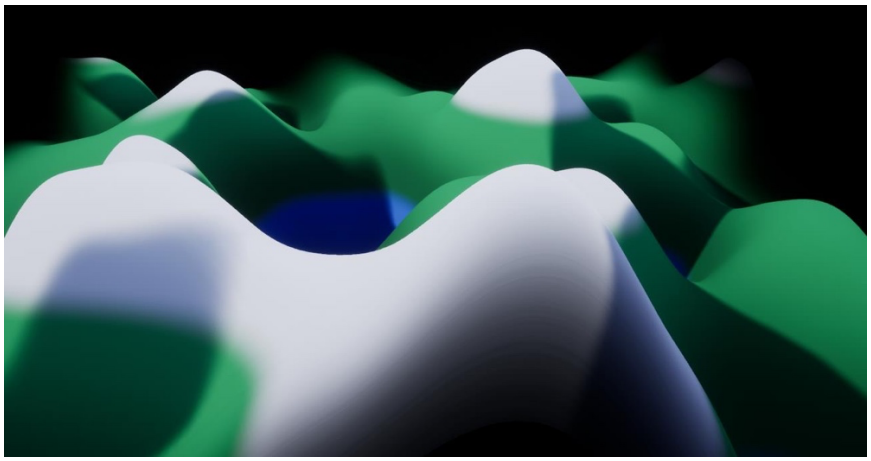
0 sec 1 sec 2 sec 3 sec



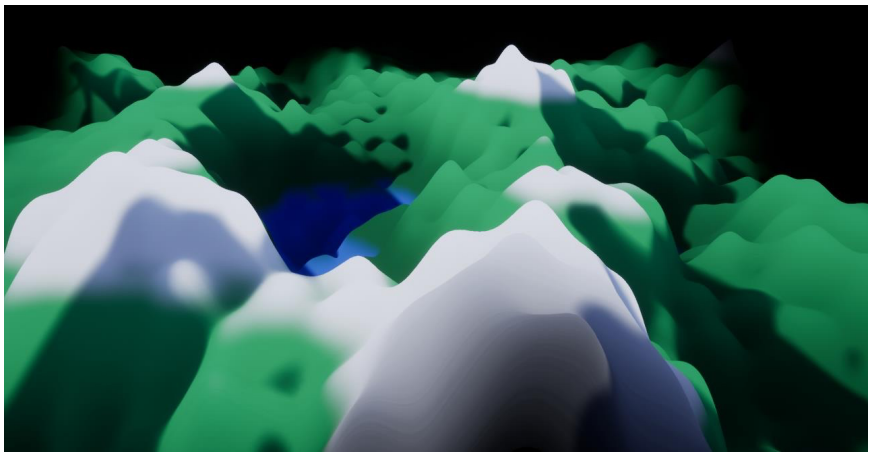
In the third row of the panel (3) using the buttons "+" and "–" buttons, you can change the number of octaves of noise used in texture generation (described in section 2.1). The current number of octaves is shown to the left of these buttons.

The value of this parameter plays an important role in forming the final texture. The user should perceive the number of octaves as the degree of texture detail. Let's show what texture detail is using the example of its three-dimensional model for different values of the number of octaves parameter:

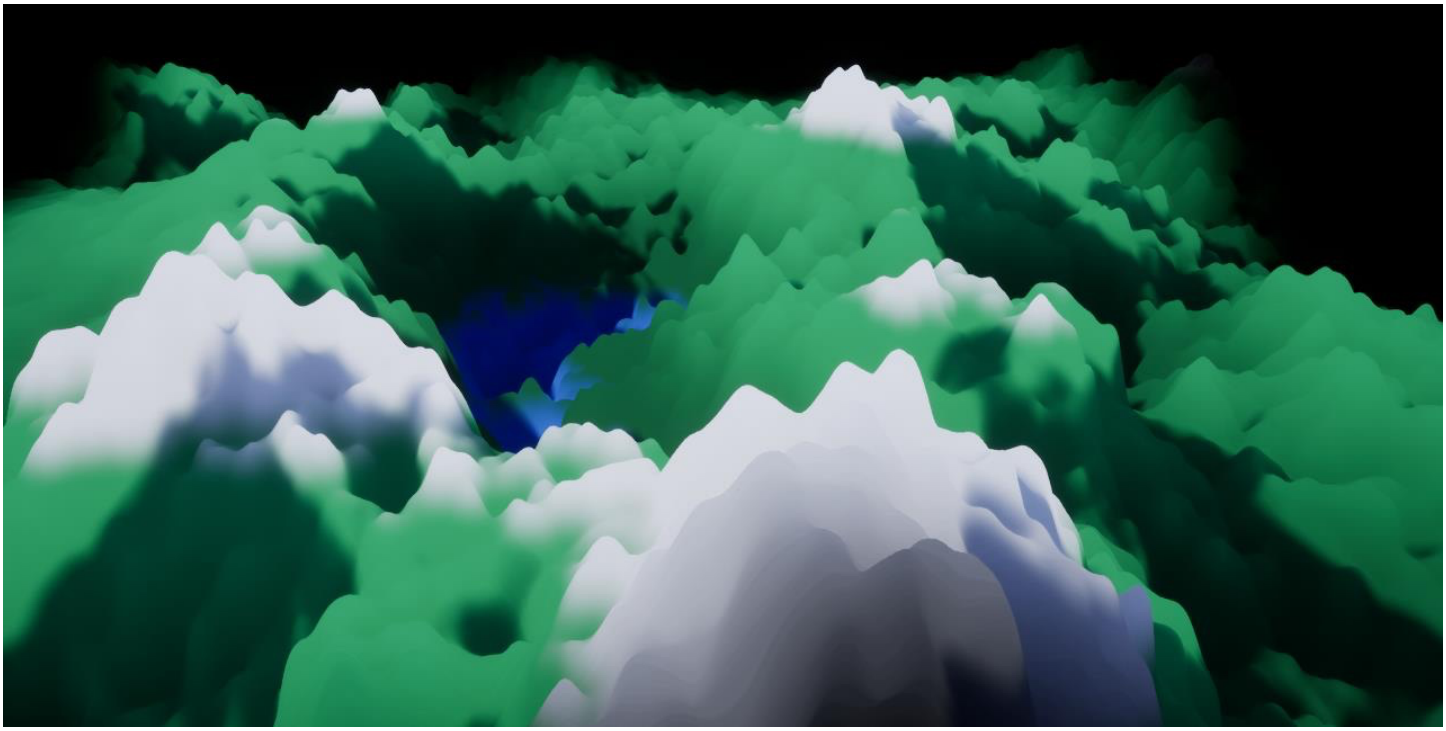
1 octave.



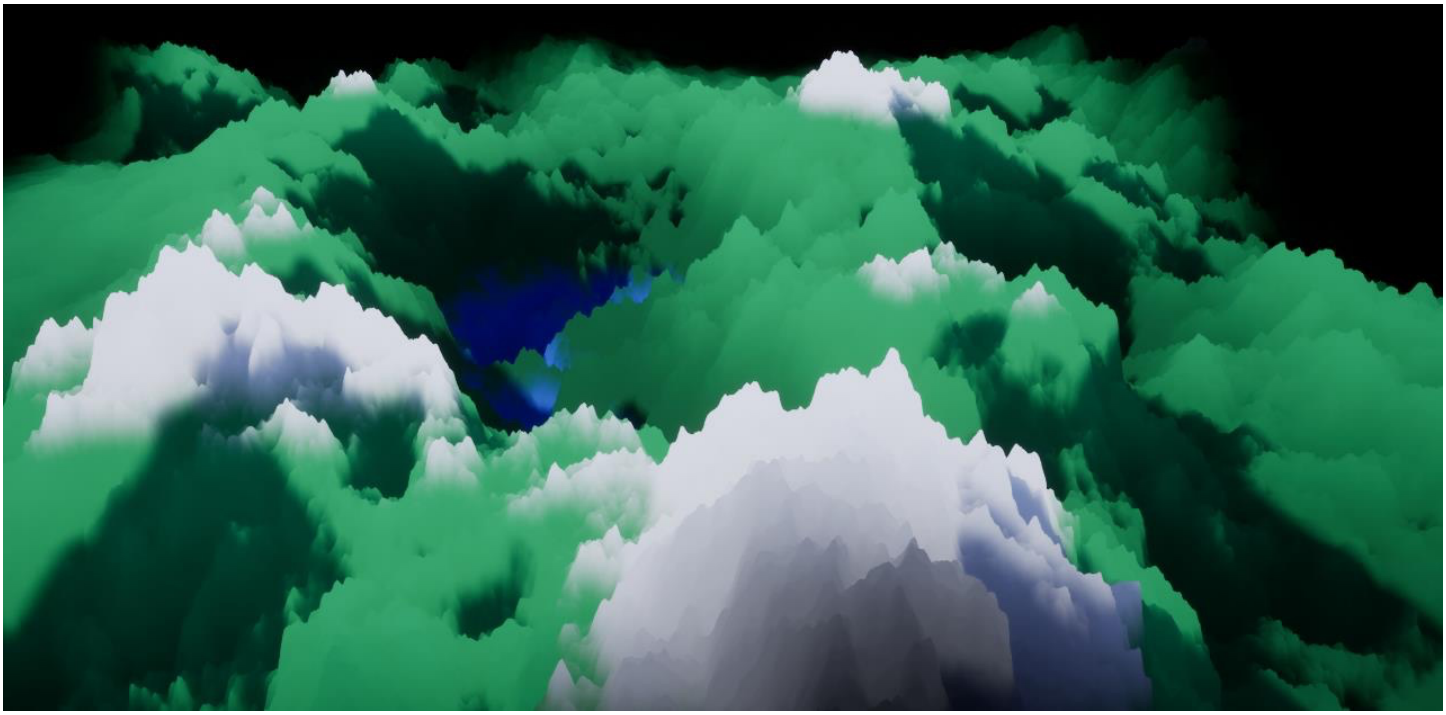
2 octaves.



4 octaves.



10 octaves.

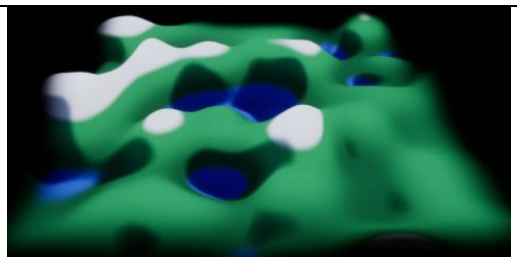
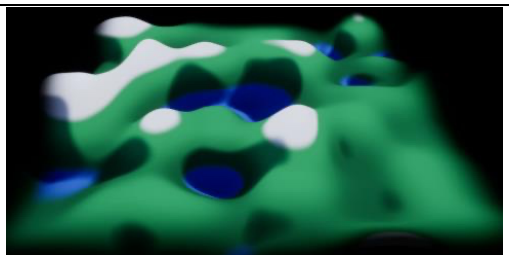
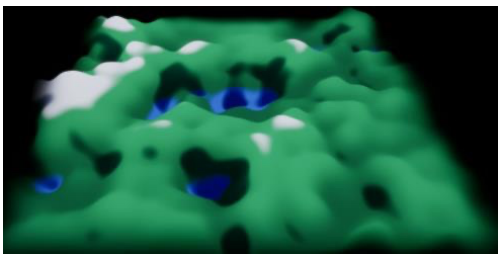
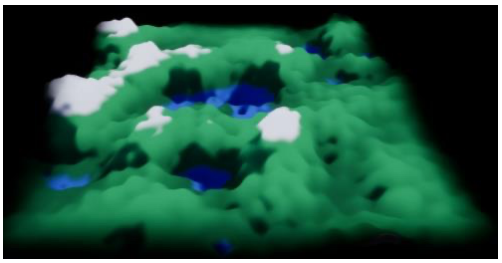
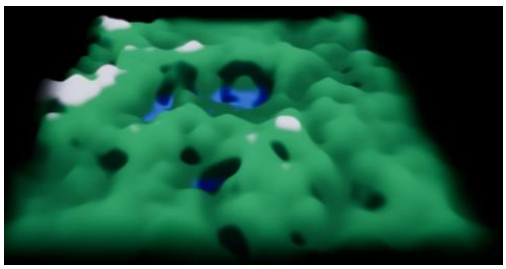
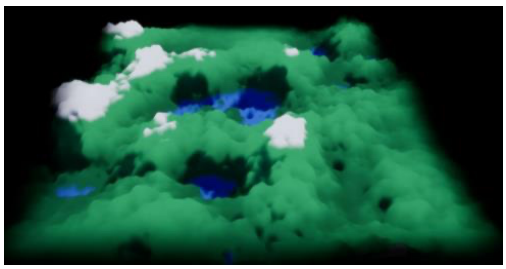


The last row of the "switch panel" (3) contains a selection row consisting of four switches, only one of which can be active. Each switch activates the corresponding octave mixing method (the process of which is described in section 2.1).

Users of the developed computer system are advised to choose the type of mixing they prefer from among:

$$1, \quad \frac{1}{n}, \quad \frac{1}{n^2}, \quad \frac{1}{2^n},$$

as the "strength of influence" of each n noise (each octave) on the final texture. That is, for example, when selecting "$1/n$", the first octave "acts" with a strength of 1 (100%), the second with a strength of $1/2$ (50%), and so on.

Example of octave blending for different blending types		Mixing type	
		$\frac{1}{n}$	$\frac{1}{n^2}$
Number of octaves	1		
	2		
	3		

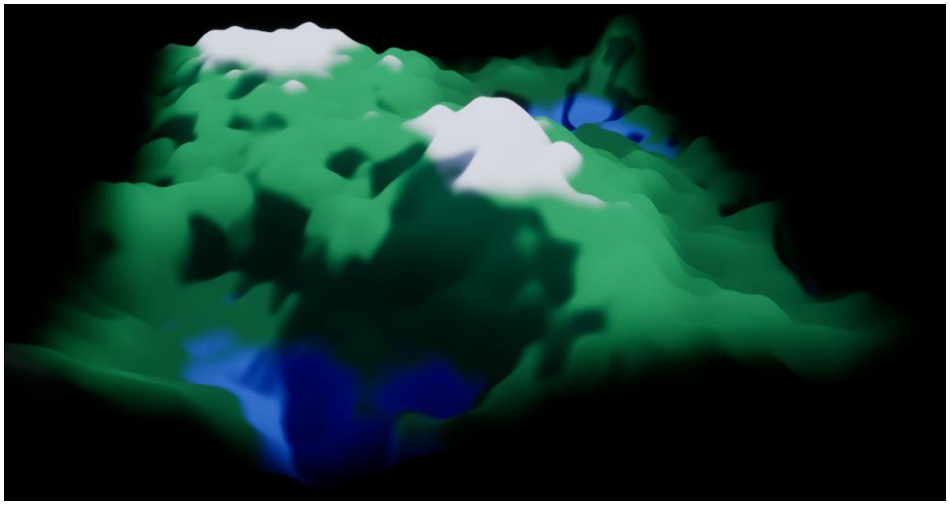
From the example given, it's easy to conclude that in the case of a mixture of the linear type ($1/n$) the octaves are mixed too "equivalently", in result the "picture" becomes less

detailed than in the case of quadratic type $(1/n^2)$ and more "wavy". The main purpose of being able to choose the octave blending type is to expand the class of solvable texturing problems. For some tasks, it makes sense to obtain a less detailed and more "blurred" texture.

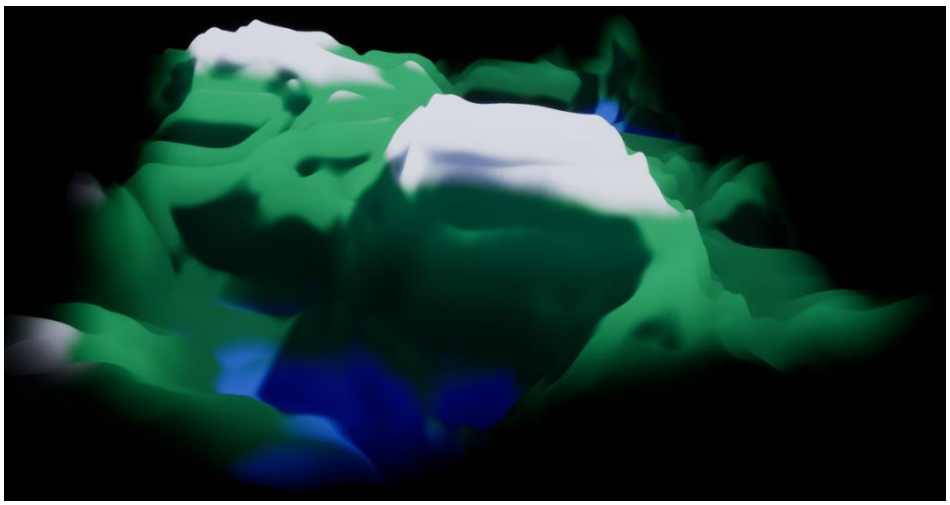
The next area of the graphical interface (4) is responsible for adjusting the distortion of the coordinate space.

The "Distortion" slider sets how much the coordinate space will be distorted: from value 0% — not at all, to value 100% — very strong.

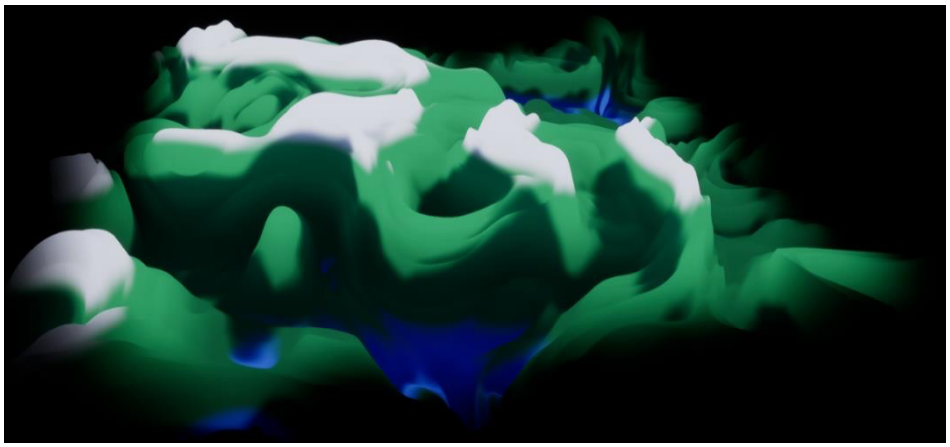
Example of a three-dimensional model of a certain texture for different values of the "Distortion" slider:



— 0%



— 5%

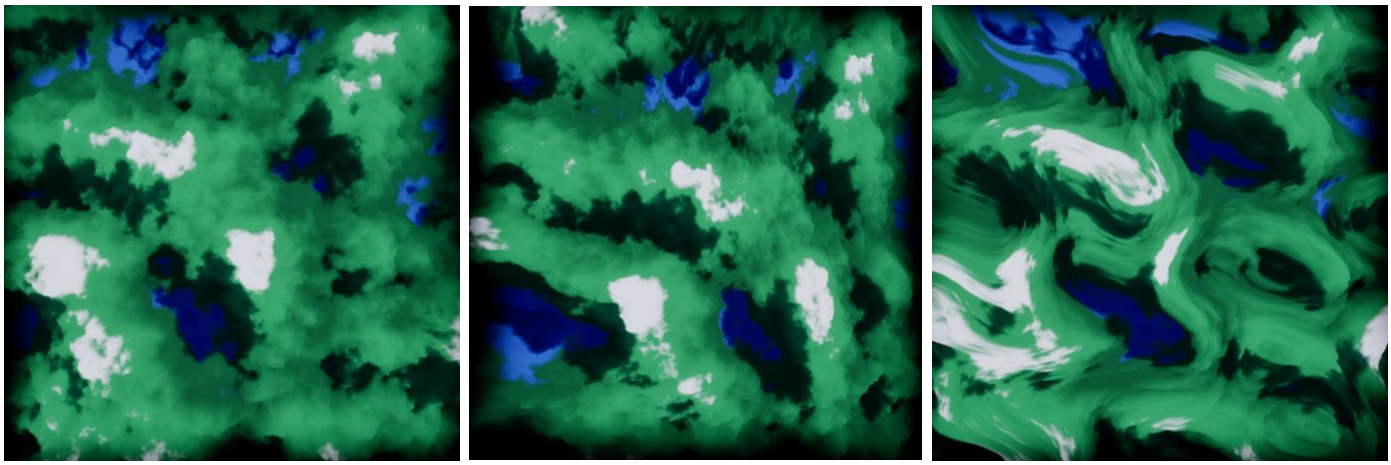


— 13%

Slider "Scale of distortion" sets how large Space will be distorted in "waves": from a value of 0%, corresponding to very small "waves," to 100%, corresponding to very large "waves".

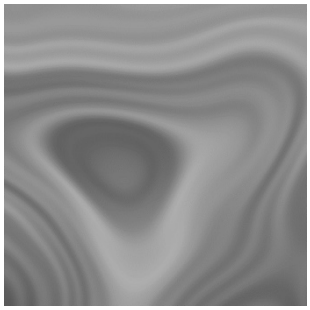
Here is an example of a three-dimensional model of a certain texture for different values of the "Distortion Scale" slider and fixed "Distortion" of 10%:

$d.scale = 0\%$ $d.scale = 50\%$ $d.scale = 5\%$



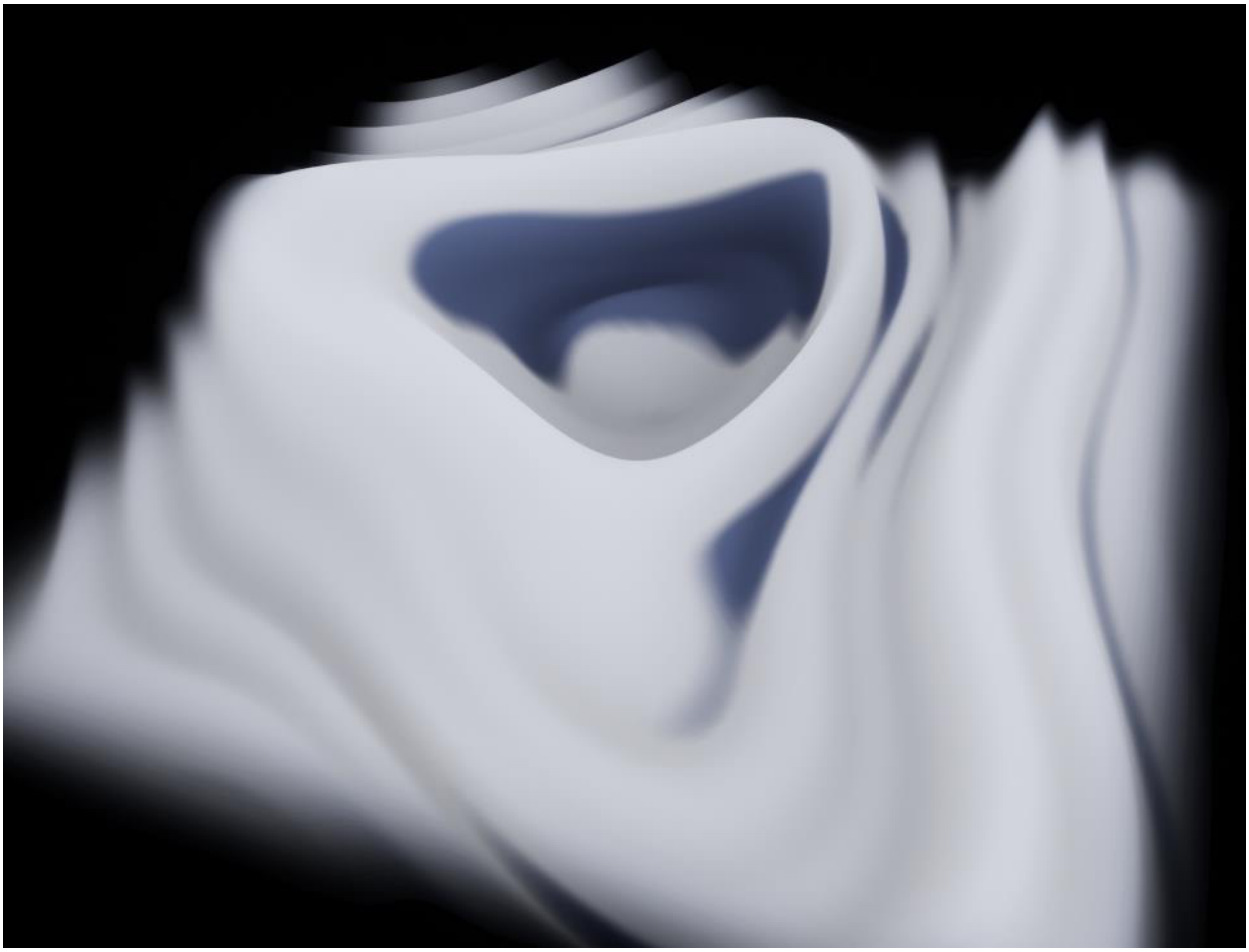
Using sliders in the distortion settings area (4) is a somewhat creative process. The user is offered a tool for changing the texture, primarily designed to expand the class of texturing tasks that can be solved using the developed computer system.

With some use of these sliders, you can obtain a texture like this:



Such a texture can be used, for example, as a malachite stone texture.

The three-dimensional model of that obtained texture looks as follows:



This can be used as a sample for modeling blanket or thick, flowing liquid.

The next area of the graphical interface (5) is located at the top center of the window. There are two sliders here:

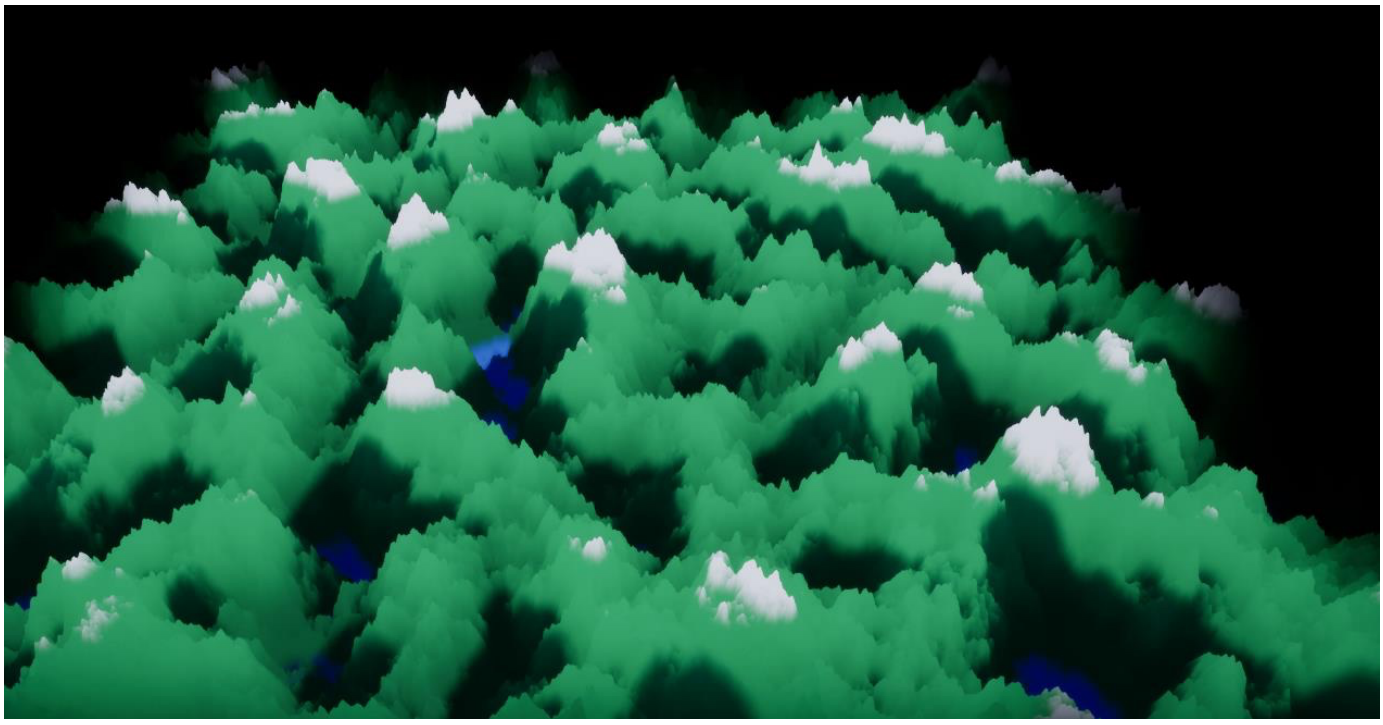
Changing the value of the "Variation" slider performs a slight shift of the global coordinate system, smoothly changing the generated texture (in fact, it changes the value of the parameter *z*, described in paragraph 2.1).

Here is an example of a three-dimensional model of a texture with different values of the "Variation" slider:

45% 50% 55% 60% 65%

The main purpose of the Variation slider is to provide the user with a tool that allows them to smoothly change the existing texture. For some texturing tasks, it is useful to obtain several similar but different textures. An example of such a task is creating a GIF animation of clouds.

The second slider in area (5) is the "Peak Density" slider. Let's consider a three-dimensional model of a certain texture:

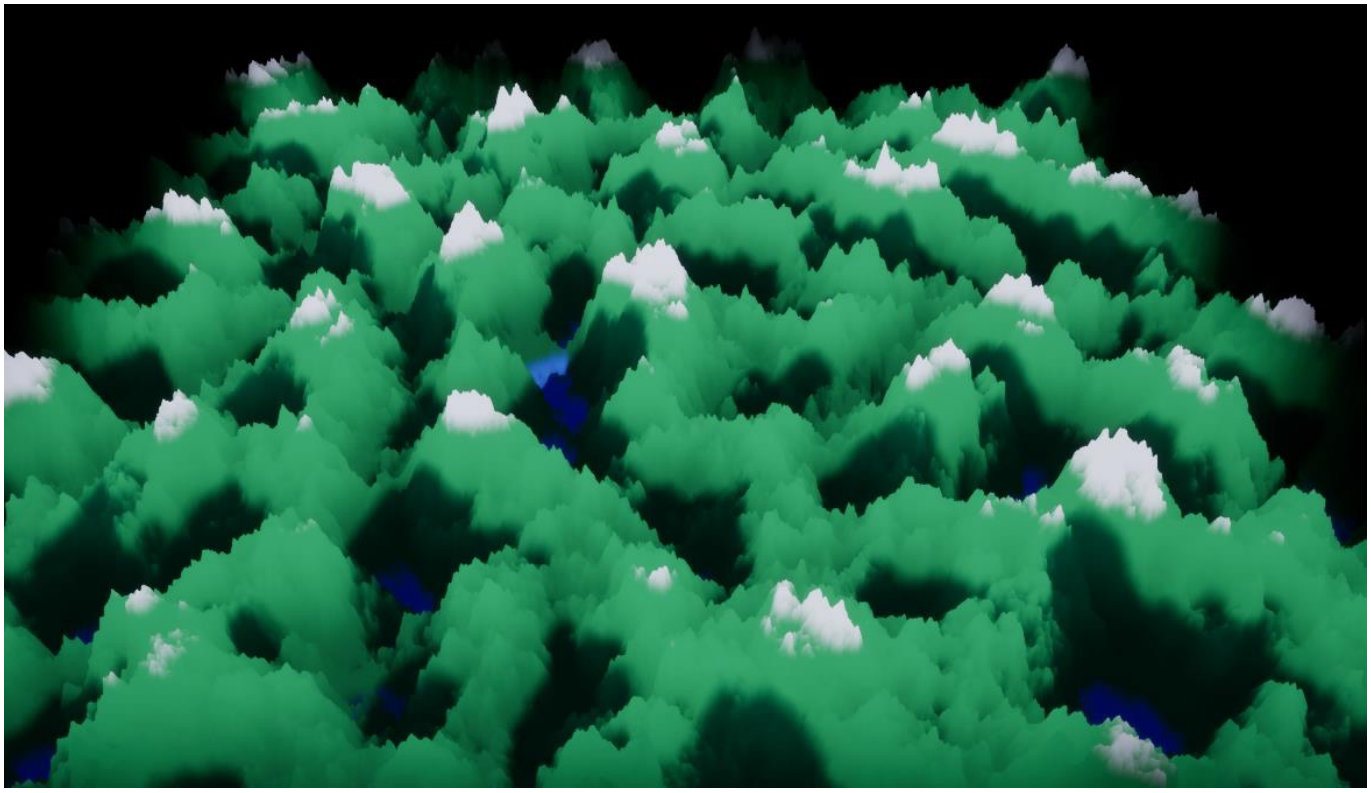


Note that the peaks (white areas) are distributed with constant density, i.e., on average, there is the same number of peaks on each square area (of a certain scale). For some texturing tasks, such as generating a game location, it is necessary to obtain a height map that is "mountainous terrain" in some areas and "flat terrain" in others. A texture with peaks distributed at a constant density is not such a height map.

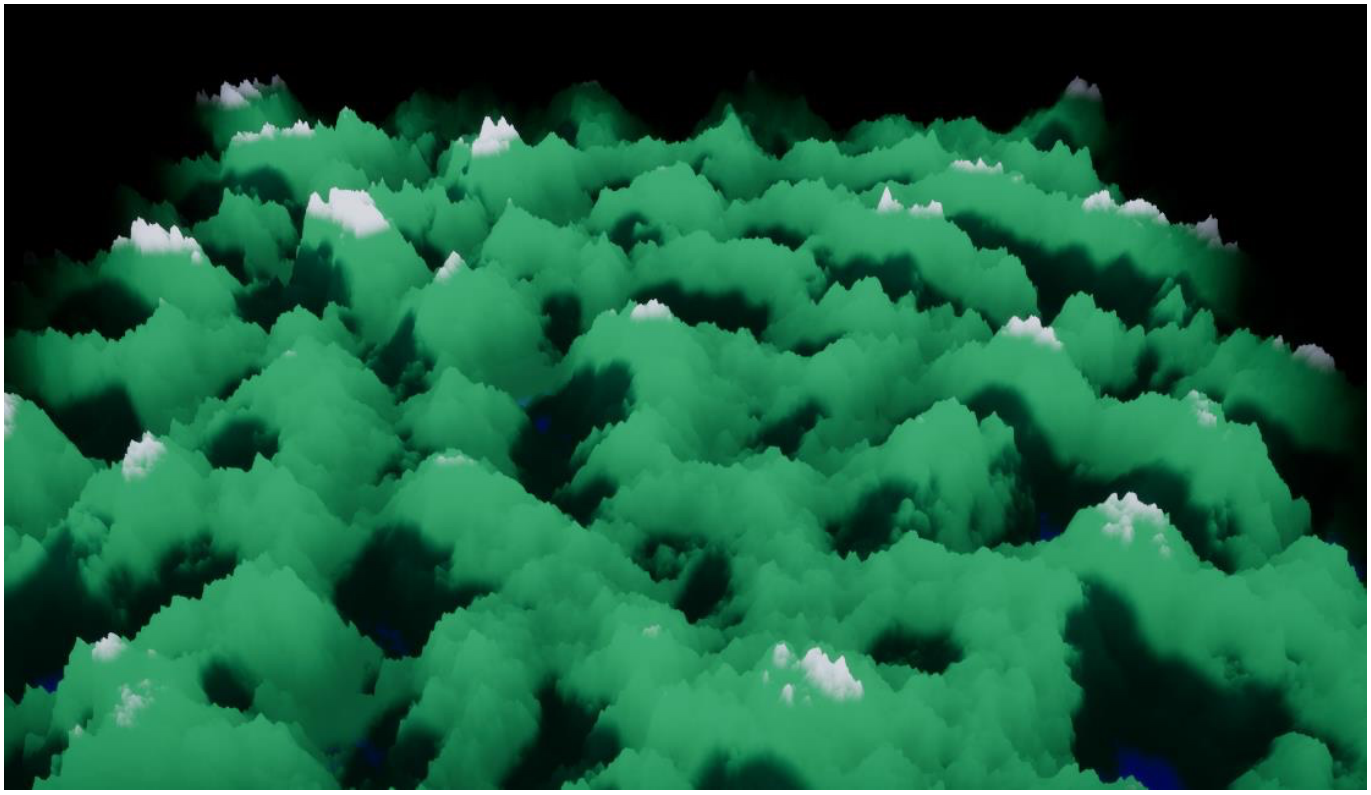
The "Peak Density" slider determines how random the density of peaks is in a fixed area (which the user sees).

Here is an example of a three-dimensional model of a texture for different values of the "Peak Density" slider:

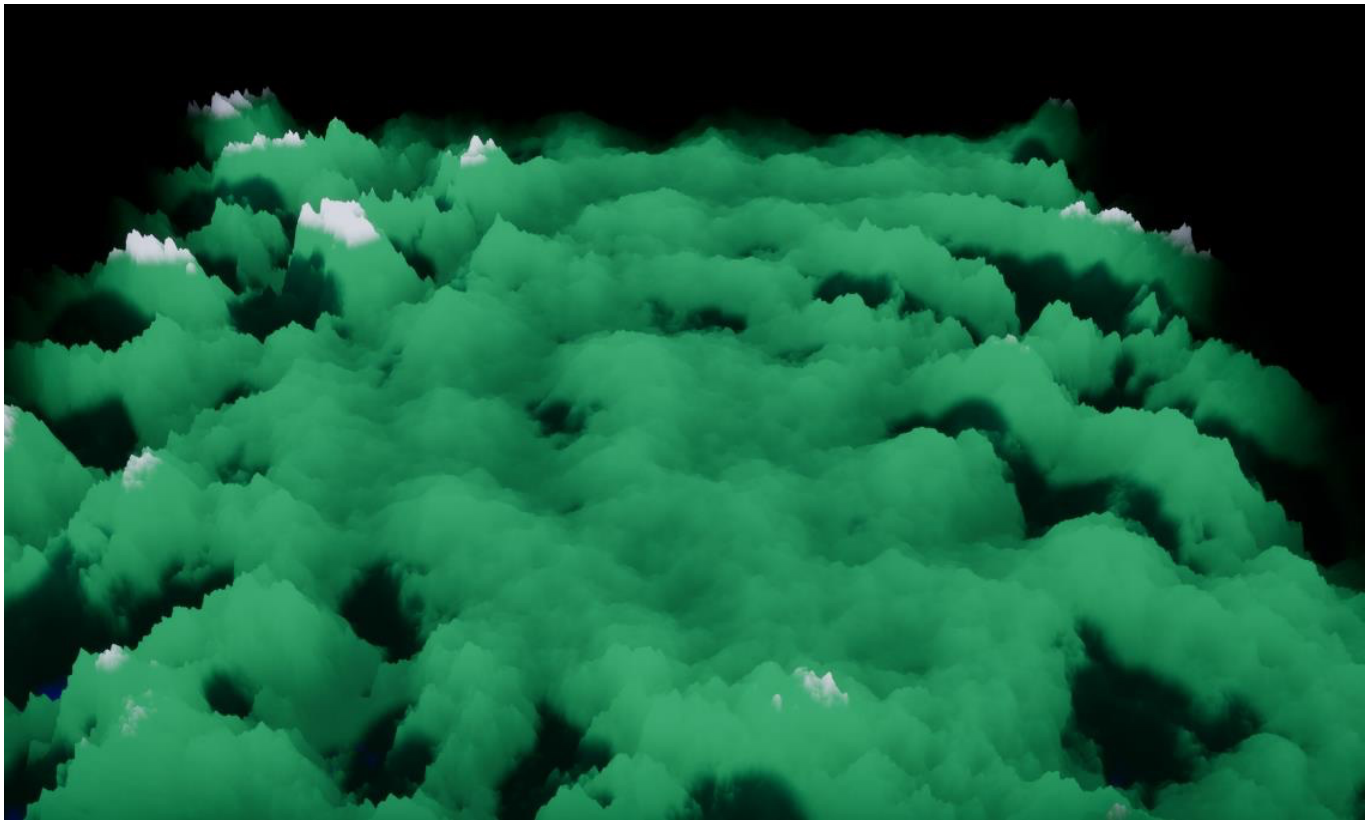
0% — the density of peaks is constant.



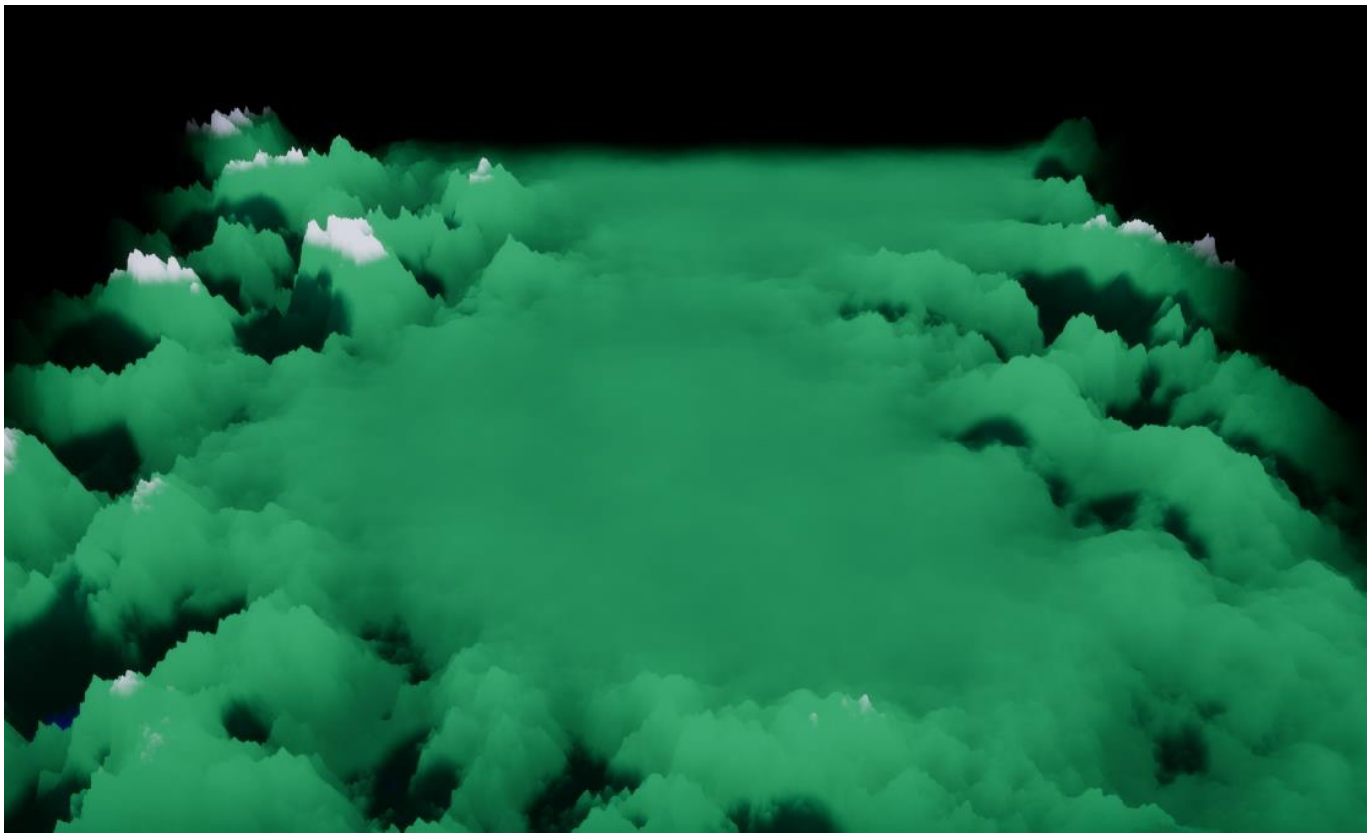
33% — peak density is "slightly" random.



66% — peak density is almost random.



95% — the density of peaks is "highly" random.



Note that the last image (for a "Peak Density" value of 95%) visually resembles a plain surrounded by mountains. The texture corresponding to this three-dimensional model is suitable for solving the problem of generating a game location.

The Peak Density slider is designed to expand the class of texturing tasks.

The next area of the graphical interface (6) contains two vertical sliders:

These sliders are designed to control the three-dimensional surface model located in area (2).

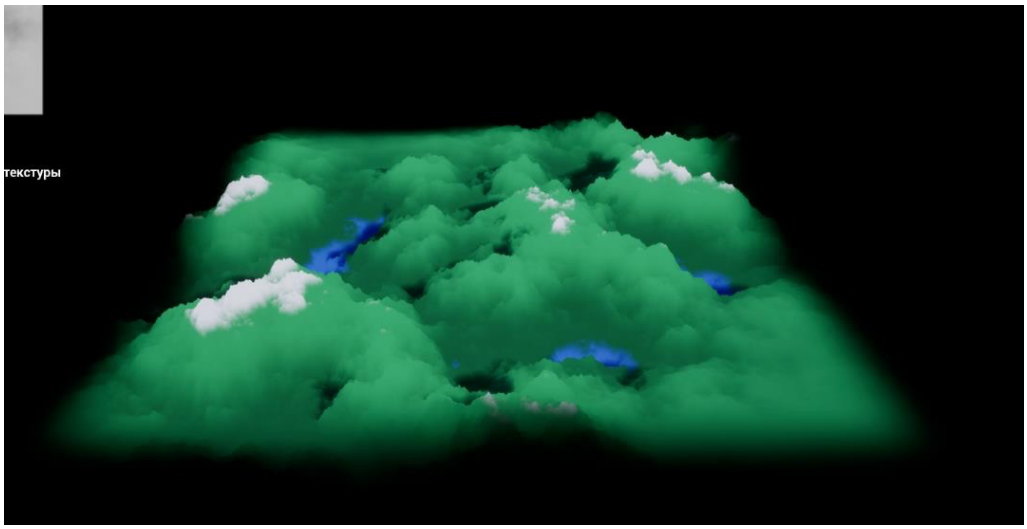
The values of these sliders do not affect the final texture.

The first (left) slider, "x, y," is responsible for the length and width of the three-dimensional surface model. When this slider is set to 0%, the surface model is as "small" as possible in length and width. When set to 100%, it is as "large."

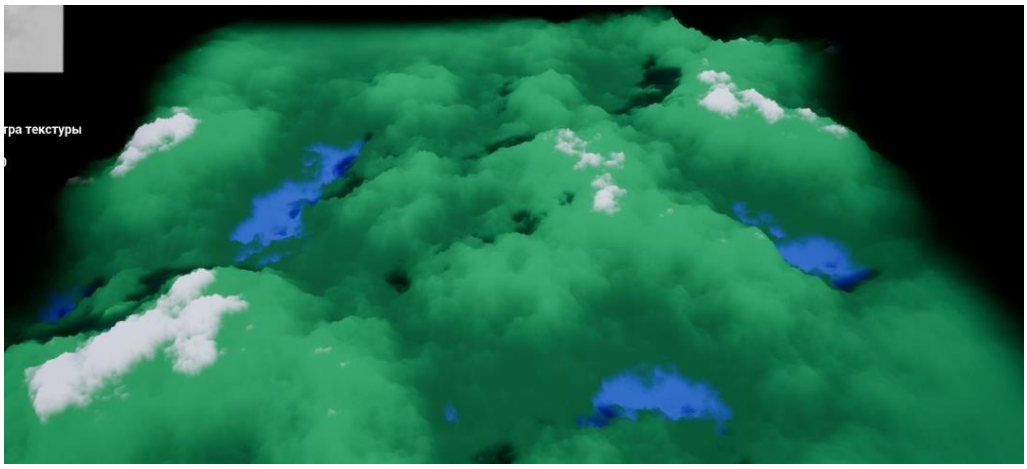
The second (right) slider, "z," is responsible for the height of the three-dimensional surface model. At a value of 0% for this slider, the surface model has zero height (extrusion according to the generated texture as per the height map). At a value of 100%, it has maximum height.

Here is an example of a three-dimensional model of a certain texture for different values of the "x, y" and "z" sliders.

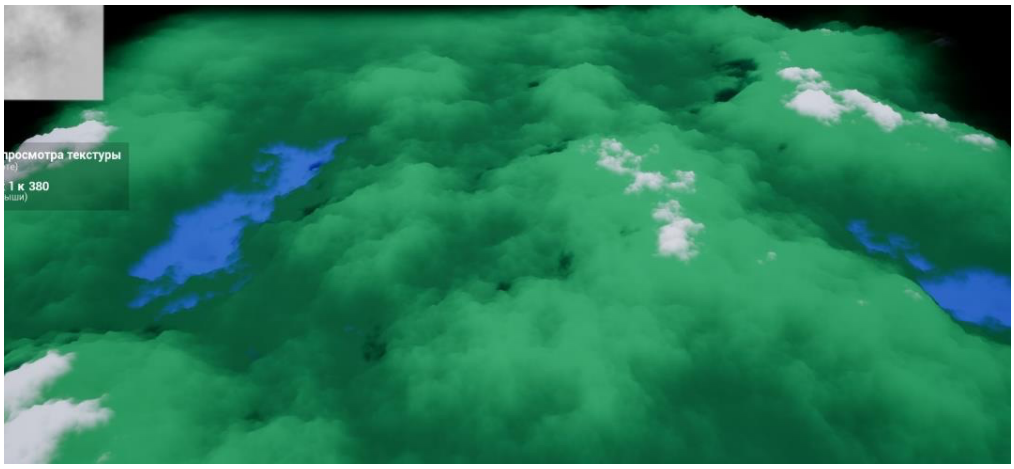
For the "x, y" slider:



— 0%



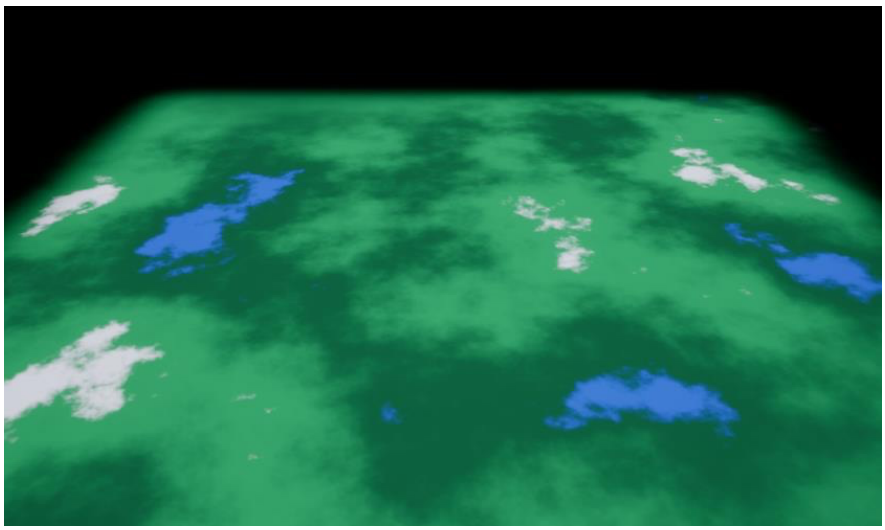
— 25%



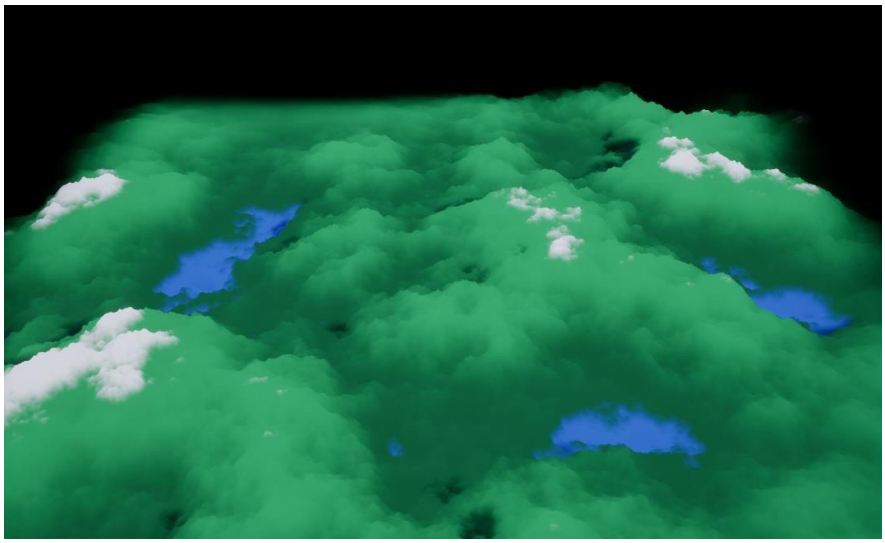
— 75%

The example shows that the higher the value of the "x, y" slider, the larger the three-dimensional model of the generated texture becomes.

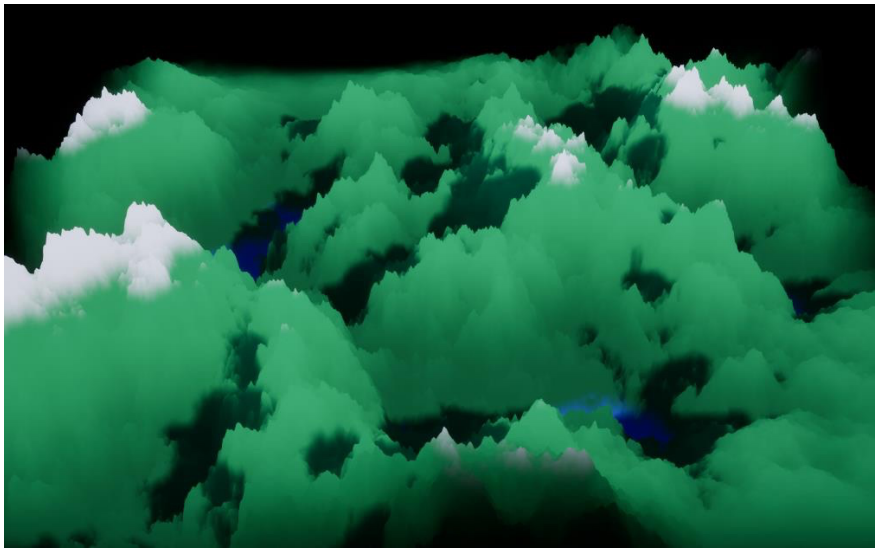
For the "z" slider:



— 0%



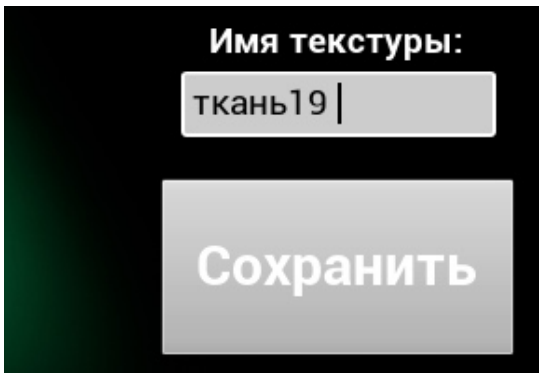
— 20%



— 55%

The example shows that the higher the value of the "z" slider, the more "stretched" upward the three-dimensional model of the generated texture becomes.

In the lower right corner of the graphical interface window is the "export area" (7):



This area is intended for exporting the generated texture.

Below the "Texture Name" label is a text field for entering the name of the exported texture. When you click the "Save" button, the texture displayed on the mini-map in area (1) is exported: the final image 2048×2048 pix in PNG format with the specified name will immediately appear in the "Сохраненные" folder, which is located in the folder with the developed computer system.

There are two more service elements in the system's graphical interface window:

In area (8) there is a "?" icon; when you hover the mouse cursor over it, you can see information about the author of the developed computer system.

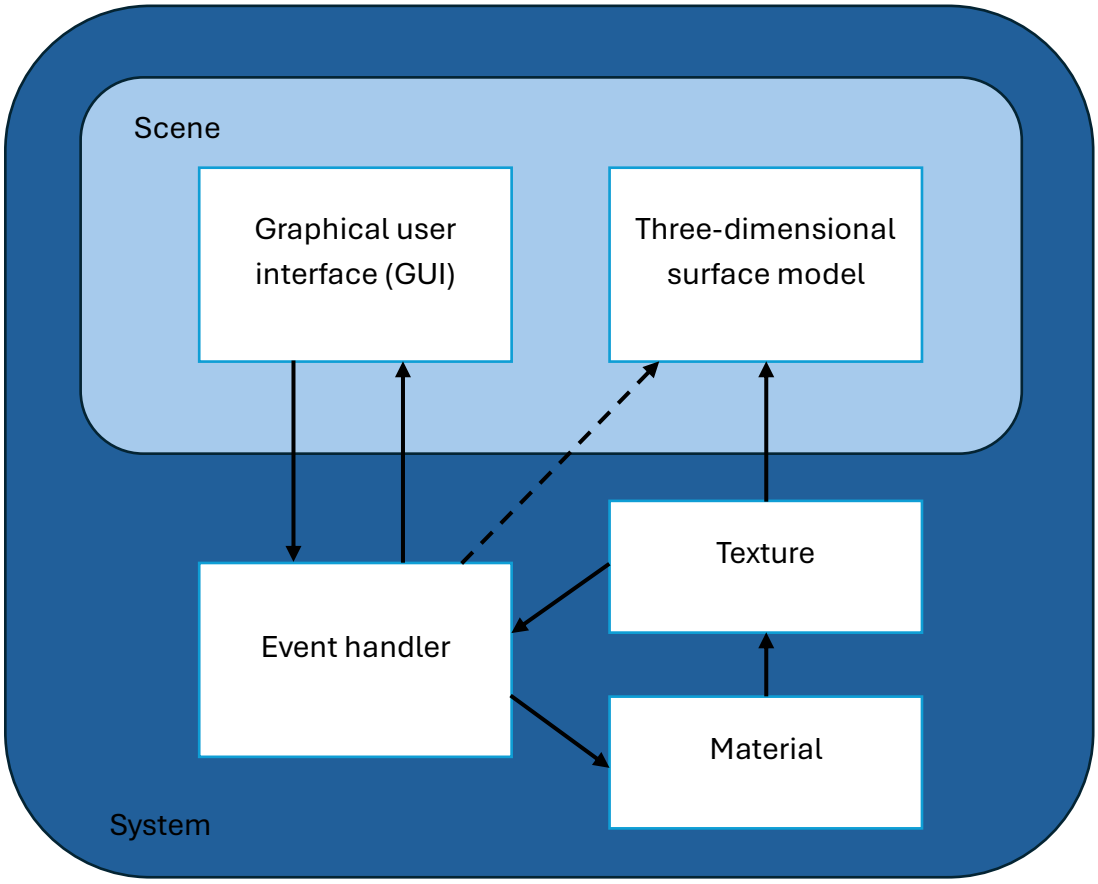
In area (9) there is a button which, when pressed, terminates the operation of the developed computer system.

Users of the developed computer system, which generates pseudo-random textures using Perlin noise, are advised to:

- to use a slider value of zero for textures that are not intended to be used as elevation maps, with the "Colors" switch turned off and the surface model viewed from above,
- to generate textures that are planned to be used as terrain height maps, use the "Peak Density" slider,
- to generate malachite and fabric textures, use the sliders in the distortion settings area numbered (4),
- to generate textures for clouds, fog, smoke, and fire, use the "Animation" switch,
- to give the exported texture files meaningful names,
- to exit the program, use the button in area (9).

2.4. Structure and Algorithms of the Developed System

The structure of the developed computer system that generates pseudorandom textures using Perlin noise can be described by the following block diagram:



Let us describe all the elements of the structure.

- Material — a set of shaders, together implementing the construction and rasterization of the display *P* described in paragraph 2.1
- Texture — a software class of an image (in the OOP paradigm)
- Scene — an abstract generalization of structural elements with which the user interacts in one way or another
- GUI — software implementation of the interface described in detail in section 2.3
- Three-dimensional surface model — software implementation of the three-dimensional surface model described in section 2.3
- Event handler — the core element of the developed computer system, coordinating user interaction with the system.

The work loop of the created program is as follows.

- Based on the set values of the texture generation parameters and viewing parameters, gradient noise and its three-dimensional model are created and rendered.
- When certain user-initiated events occur, the texture generation parameters and/or viewing parameters change.
- When the user clicks the "Save" button, the function of which is described in paragraph 2.3, the gradient noise is "baked" into a PNG file and then saved to the computer's hard drive.
- When the user clicks the exit button, the function of which is described in paragraph 2.3, the program objects are safely destroyed and the program is terminated.

The algorithms underlying the operation of most structural elements of the developed computer system are quite standard (e.g., event processing algorithms, algorithms for controlling the rotation of a three-dimensional model, etc.). We will omit a detailed description of these algorithms. Their program codes are presented in Appendix 1.

The main algorithm of the developed system is the algorithm for generating gradient noise based on Perlin noise. This algorithm is implemented as a set of shaders written in the HLSL shaders programming language. The algorithm consists of constructing the mapping P described in paragraph 2.1, and its subsequent rasterization. The program codes of the auxiliary shaders are given in Appendix 2. The program code of the shader that generates classic Perlin noise is given in Appendix 3. The program code of the main shader that generates the final gradient noise based on the values of all generation parameters is given in Appendix 4.

Conclusion on Chapter 2.

A computer system has been created that generates pseudo-random textures using Perlin noise, has an intuitive interface and a simple texture viewing system, and does not require the user to have specialized knowledge (in programming, 3D graphics, computational geometry, etc.)

The developed computer system is capable of solving a whole class of problems related to obtaining gradient noise. Such a system can be used:

- in companies engaged in the creation of computer graphics (e.g., raster images, animations, three-dimensional models, etc.),
- in companies involved in the creation of computer games,
- in the film industry.

Examples of the application of the developed system are presented in Appendix 5.

CONCLUSION

In the course of the work, a computer system for procedural texture generation based on Perlin noise was created. This system can be useful for any user who needs to automatically generate gradient noise.

The following tasks were solved in the course of the work.

1. An analysis of literary sources devoted to Perlin's noise algorithm and its mathematical apparatus, as well as modern programming tools and methods, was conducted.
2. An analysis was conducted of software products that generate pseudorandom textures, particularly those that use the Perlin noise algorithm for this purpose.
3. An analysis of existing development environments was conducted, their advantages and disadvantages in relation to the set goal were identified, and an environment for developing a computer system was selected.
4. A system that generates pseudo-random textures using Perlin noise was designed and implemented as a computer program.

The developed computer system has many options for further expansions:

- the introduction of new parameters for constructing a display derived from classical Perlin noise,
- adding new algorithms to the system that produce textures of other classes, different from gradient noise,
- expansion the export system: introduction new file formats available for selection when saving,
- creation of texture animation recording tools,
- adding the ability to generate three-dimensional textures.

Future plans include the design and software implementation of the listed extensions.

LIST OF REFERENCES

1) "Writing real Perlin noise" [Electronic resource] (accessed: 15.03.20)
   URL: https://habr.com/ru/post/265775

2) "THE RANGE OF PERLIN NOISE" [Electronic resource] (accessed: 17.03.20)
   URL: https://digitalfreepen.com/2017/06/20/range-perlin-noise.html

3) Josh Petty, "What is Houdini & What Does It Do?" [Electronic resource] (accessed: 21.03.20)
   URL: https://conceptartempire.com/what-is-houdini-software

4) GAEA DOCUMENTATION [Electronic resource] (accessed: 26.03.20)
   URL: https://docs.quadspinner.com

5) "An Introduction to World Machine" [Electronic resource] (accessed: 26.03.20)
   URL: https://help.world-machine.com/topics/manual

6) Substance Designer [Electronic resource] (accessed: 27.03.20)
   URL: https://docs.substance3d.com/sddoc/substance-designer-102400008.html

7) Shader [Electronic resource] (accessed: 28.03.20)
   URL: https://en.wikipedia.org/wiki/Shader

8) Turbulent Noise [Electronic resource] (accessed: 28.03.20)
   URL: https://www.sidefx.com/docs/houdini/nodes/vop/turbnoise.html

9) "Perlin" [Electronic resource] (accessed: 10.04.20)
   URL: https://docs.quadspinner.com/Reference/Primitives/Perlin.html

10) Generator device - advanced Perlin noise [Electronic resource] (accessed: 10.04.20)
   URL: http://www.world-machine.com/learn.php?page=devref#APRL

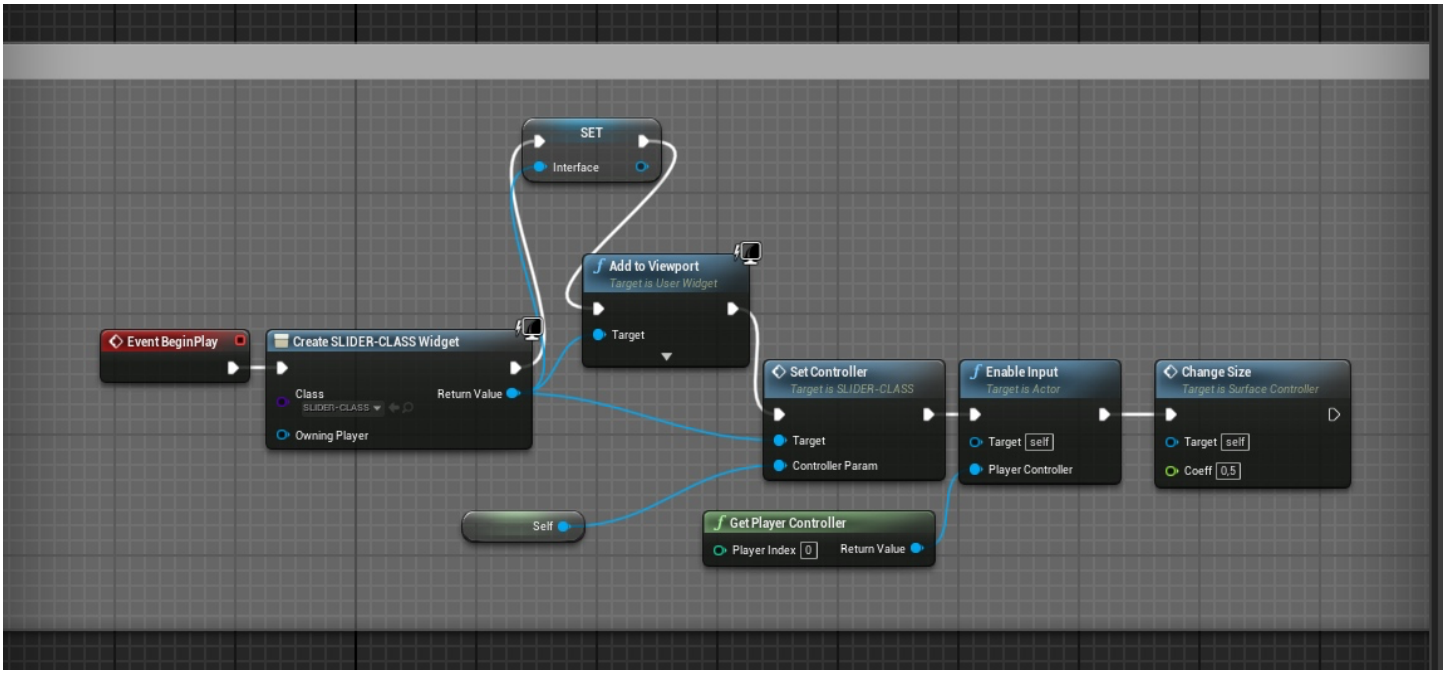
11) 3D Perlin Noise Fractal [Electronic resource] (accessed: 10.04.20)
   URL: https://docs.substance3d.com/sddoc/3d-perlin-noise-fractal-168199162.html

APPENDICES

Appendix 1. Listings of Standard System Element Codes

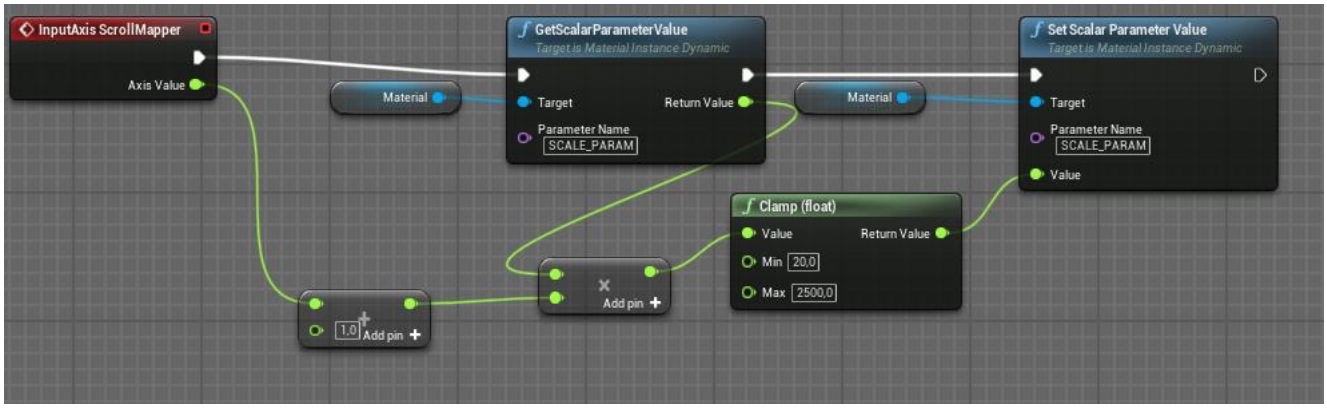
System elements are written in the node language described in section 1.2. Unreal Engine uses the Blueprint visual programming language. Here are some code blocks from the developed computer system.

Initialization of the "Event Handler":

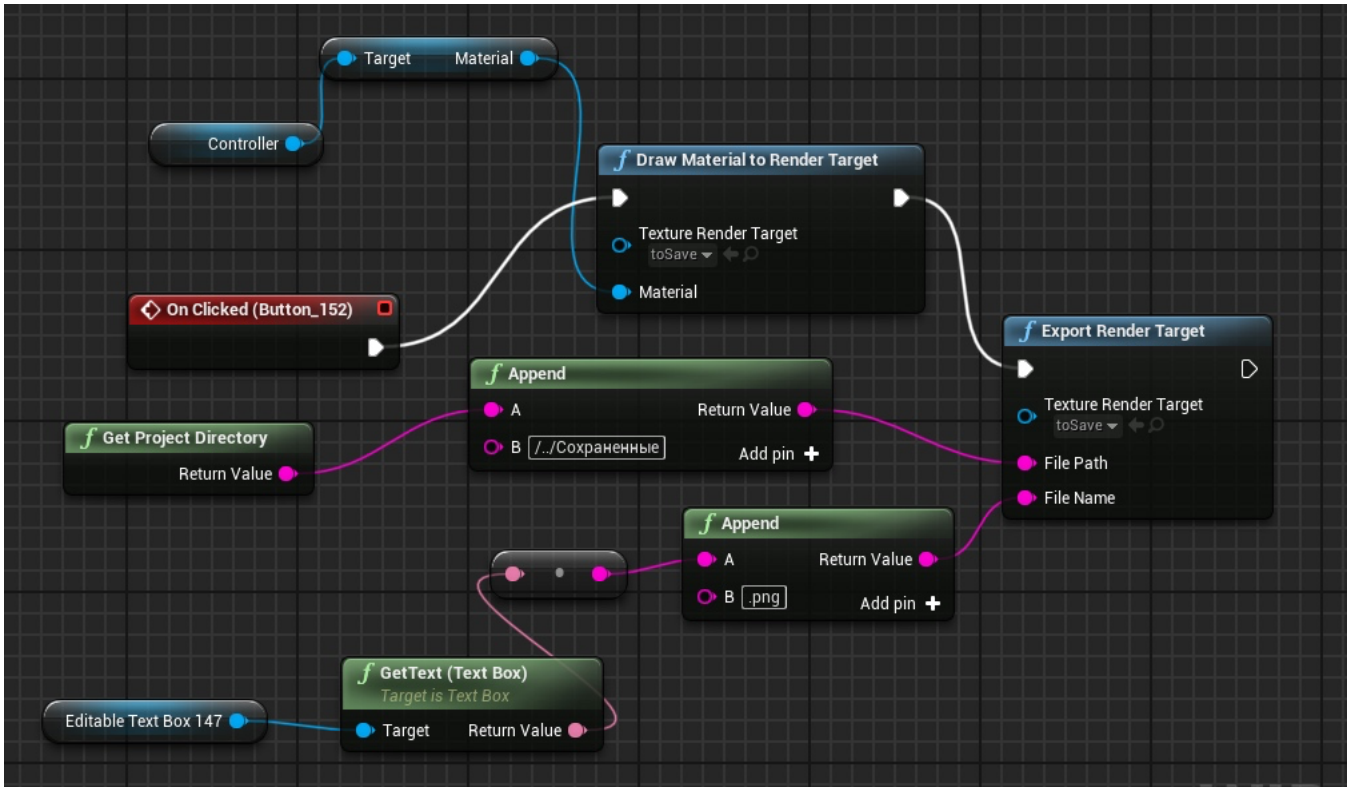


This section of code is responsible for initializing the "Event Handler," the "Graphical User Interface," and establishing a connection between them. For service purposes, the "Event Handler" also includes an additional ability to read mouse events.

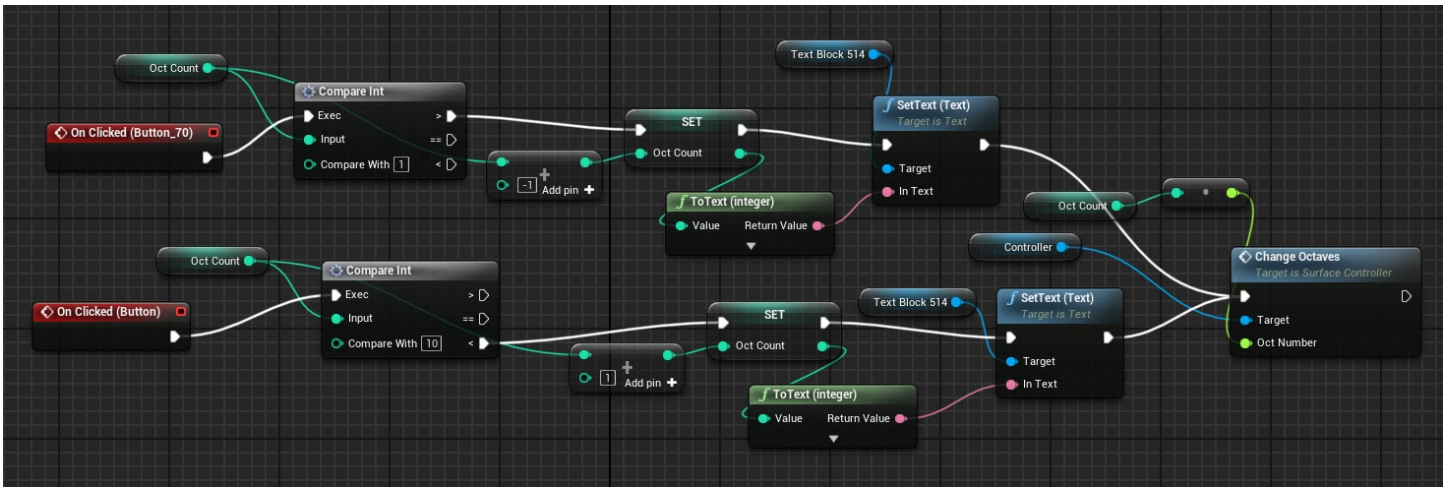
The following code block describes the logic of changing the global scale by rotating the mouse wheel:



The block of code responsible for exporting the texture:



An example of a "standard" algorithm, many of which were mentioned in section 2.4:



This block of code is responsible for responding to the user pressing the "+" and "–" buttons described in section 2.3. It can be seen that a very cumbersome code performs a simple check for maximum/minimum. This is one of the disadvantages of visual programming languages.

There are many other simple algorithms in the developed computer system that are not of particular interest. Here is a general listing of the "Event Handler" to show the approximate

amount of code in one of the eight main classes of the system that is not directly related to texture generation:

Appendix 2. Auxiliary Shaders

Here is a listing of the shader implementing the nonlinear interpolation $iLerp$ described in section 1.1.

Interpolations.ush:

```
#pragma once

float iQuad(float a, float b, float t)
{
    float quadCurve = t * t * t * (t * (t * 6 - 15) + 10);
    return lerp(a, b, quadCurve);
}
```

Here is the listing of the shader that defines the pseudo-random unit vector generator used in the system.

Random.ush:

```
#pragma once

float3 random3(float3 c) {
    float j = 4096.0 * sin(dot(c, float3(17.0f, 59.4f, 15.0f)));
    float3 r = float3(0.0f, 0.0f, 0.0f);

    r.z = frac(512.0f * j);

    j *= 0.125f;
    r.x = frac(512.0f * j);

    j *= 0.125f;
    r.y = frac(512.0f * j);

    return normalize(r - 0.5f);
}
```

Appendix 3. Classic Perlin Noise Shader

Let's take a look at a shader listing that implements Perlin's classic three-dimensional noise P_r. Perlin.ush:

```
#pragma once
#include "/Project/Interpolations.ush"
#include "/Project/Random.ush"

float perlinGradient(float3 position, float seed) {
    float3 p = floor(position);
    float3 k = abs(frac(position));

    float3 v000 = random3(p + seed);
    float3 v001 = random3(p + float3(0.0f, 0.0f, 1.0f) + seed);
    float3 v010 = random3(p + float3(0.0f, 1.0f, 0.0f) + seed);
    float3 v011 = random3(p + float3(0.0f, 1.0f, 1.0f) + seed);
    float3 v100 = random3(p + float3(1.0f, 0.0f, 0.0f) + seed);
    float3 v101 = random3(p + float3(1.0f, 0.0f, 1.0f) + seed);
    float3 v110 = random3(p + float3(1.0f, 1.0f, 0.0f) + seed);
    float3 v111 = random3(p + float3(1.0f, 1.0f, 1.0f) + seed);

    float h000 = dot(k, v000);
    float h001 = dot(k - float3(0.0f, 0.0f, 1.0f), v001);
    float h010 = dot(k - float3(0.0f, 1.0f, 0.0f), v010);
    float h011 = dot(k - float3(0.0f, 1.0f, 1.0f), v011);
    float h100 = dot(k - float3(1.0f, 0.0f, 0.0f), v100);
    float h101 = dot(k - float3(1.0f, 0.0f, 1.0f), v101);
    float h110 = dot(k - float3(1.0f, 1.0f, 0.0f), v110);
    float h111 = dot(k - float3(1.0f, 1.0f, 1.0f), v111);

    float a = 0.0f;
    float b = 0.0f;
    float c = 0.0f;
    float d = 0.0f;
    float A = 0.0f;
    float B = 0.0f;

    a = iQuad(h000, h100, k.x);
    b = iQuad(h010, h110, k.x);
    c = iQuad(h001, h101, k.x);
    d = iQuad(h011, h111, k.x);

    A = iQuad(a, b, k.y);
    B = iQuad(c, d, k.y);

    return (sqrt(3)/2 + iQuad(A, B, k.z))/sqrt(3);
}
```

Appendix 4. Main System's Shader

Before listing the main shader, let us list the auxiliary shader, which implements the mappings $Q, F, U, H_{\vec{u}}$ defined in section 2.1.

PerlinUse.ush:

```
#pragma once
#include "/Project/Perlin.ush"

float q (float t){
    return t * t * t * (t * (t * 6 - 15) + 10);
}

float3 U (float3 u, float p, float s){
    return u + p * (2 * perlinGradient(u/s,-1) - 1) * normalize(u);
}

float H (float h, float3 u, float t){
    float p = perlinGradient(u/5,-2);
    p = q(q(q(p)));
    return lerp(h, 0.5f + p * (h - 0.5f), t);
}

float F (float3 u, int n, int t){
    float max = 0.0f;
    float h = 0.0f;
    float k = 0.0f;
    for (int i = 1; i <= n; ++i) {
            if (t == 1) k = 1.0f;
            if (t == 2) k = i;
            if (t == 3) k = pow(i, 2);
            if (t == 4) k = pow(2, i - 1);

            max += 1.0f / k;
            h += perlinGradient(k*u, i) / k;
    }

    return h/max;
}
```

Listing of the main shader of the developed system:

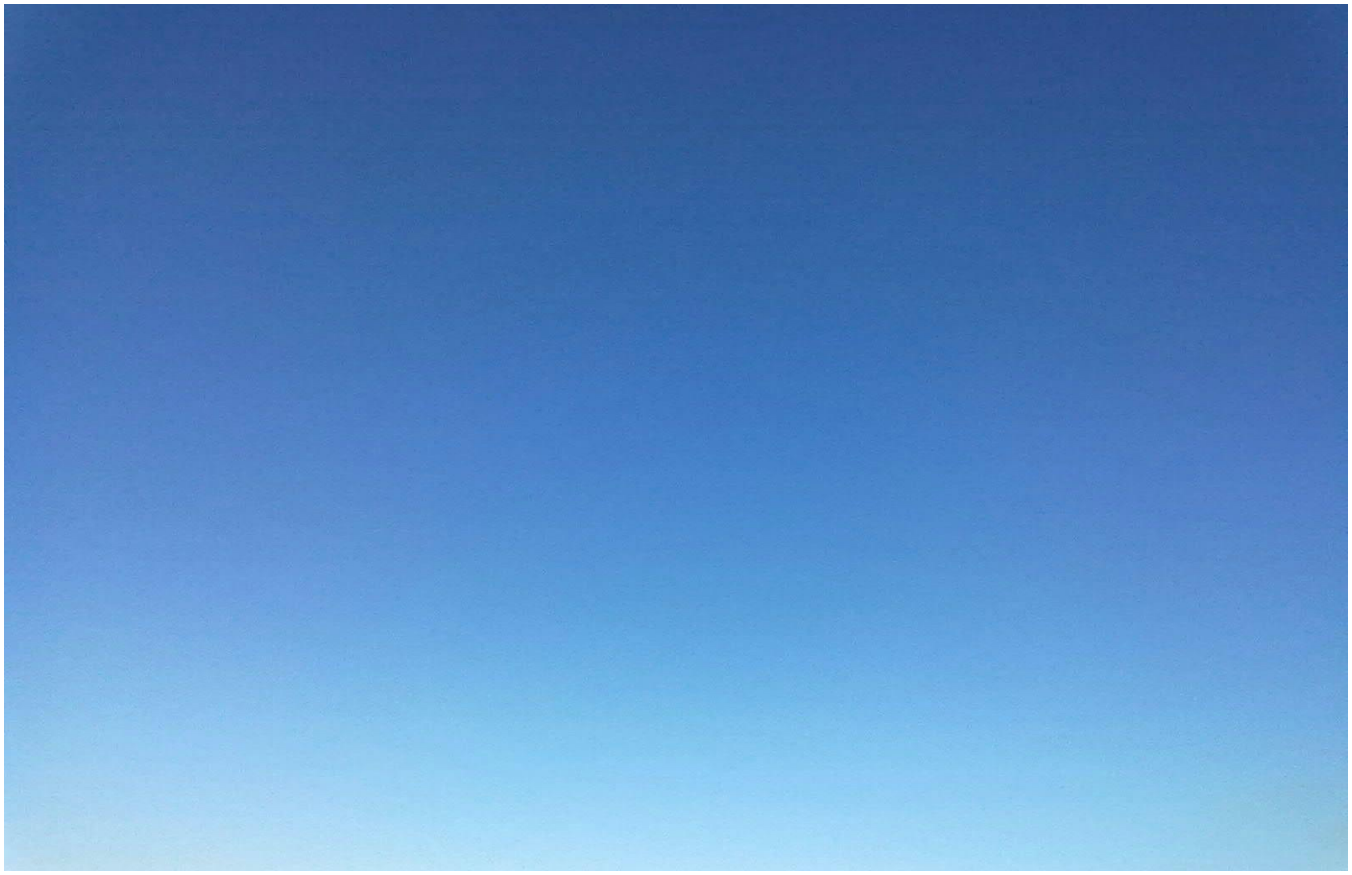
```
#pragma once
#include "/Project/Perlin.ush"

float main (
```

```
    int octaveCount,
    float scale,
    float3 position,
    float3 offset,
    int cffType,
    float flex,
    float density,
    float flexScale
){
    position = position / scale + offset;
    return H(F(U(position, flex, flexScale), octaveCount, cffType), position, density);
}
```

Appendix 5. Example of System Application

Example 1. Creating clouds

Let's take an image:



This image shows a clear sky. Task: make the image of cloudy sky.

Let's generate the following texture in the developed system:

Then applying the resulting texture to the original image of a clear sky, we get:

Let's compare:

BEFORE

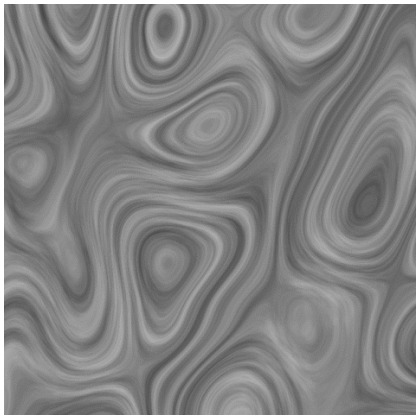
AFTER

Example 2. "Malachite texturing"

Let's take an image of a porcelain plate:

Task: make an image of the same malachite plate.

Using the developed computer system, we obtain the following texture:



Applying the resulting texture to the original image of the porcelain plate, we get the result. Let's compare:

BEFORE

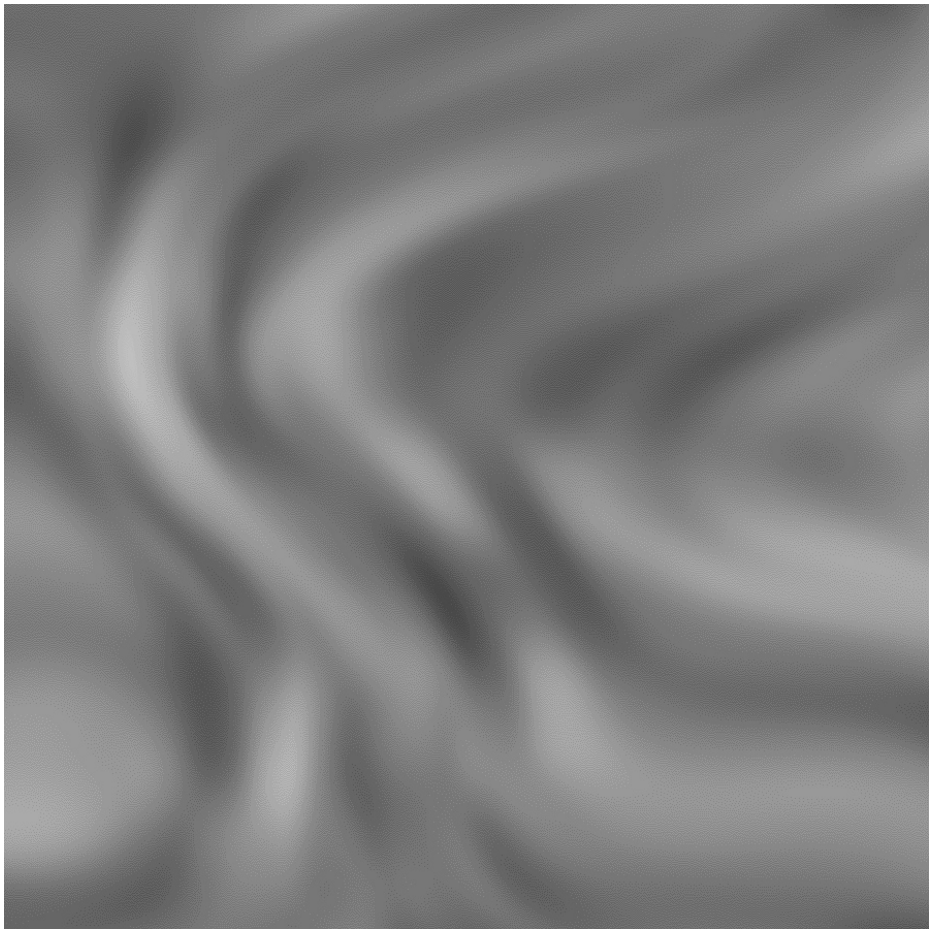
AFTER

Example 3. "Crumple the fabric"

Let's take an image of denim fabric:

Task: create an image of the same crumpled fabric.

Let's generate the following texture in the developed system:



We will shift the pixels of the denim fabric image according to the height map, using the generated texture (this can be done in many raster editors). We get:

This image has an unnatural "light and shadow pattern." To give the image the correct light and shadow, we will overlay this image on the texture generated in the developed system.

The result is as follows:

Let's compare:

BEFORE

AFTER