UKRAINIAN CATHOLIC UNIVERSITY

BACHELOR THESIS

Displaying weather forecast using generative art

Author:
Maryana MYSAK

Supervisor:
Yurii ARTYUKH

A thesis submitted in fulfillment of the requirements for the degree of Bachelor of Science

in the

Department of Computer Sciences
Faculty of Applied Sciences

Lviv 2019

Declaration of Authorship

I, Maryana MYSAK, declare that this thesis titled, "Displaying weather forecast using generative art" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

UKRAINIAN CATHOLIC UNIVERSITY

Faculty of Applied Sciences

Bachelor of Science

Displaying weather forecast using generative art

by Maryana MYSAK

Abstract

This work is about generative art and its possibilities. The best way to show the full beauty of generative art is to try to display something perfect. Nature or rather weather conditions is a great choice. In this work was made a system, that displays natural animated landscape, that should show in an accessible way the weather conditions. The weather conditions were analyzed and a special landscape view was prepared.

Every part of the full landscape composition was generated by the system without using any libraries. Every part of landscape was made with inspiration and love.

Acknowledgements

Firstly, I want to thank my supervisor Yurii Artukh for always good advices.

I am deeply grateful to all my teachers from university, who showed me the world of computer science.

I am thankful to my family for their support throughout all four years.

The special and the biggest thanks to Pavlo Berezin, the one, who made me do this work.

Contents

List of Figures

List of Abbreviations

PCG	**P**rocedural**C**ontent **G**eneration
ACM	**A**ssociation for **C**omputing **M**achinery's
SIGGRAPH	**S**pecial **I**nterest **G**roup on **GRAPH**ics and Interactive Techniques

Dedicated to the beauty of the nature

Chapter 1

Introduction

1.1 Relevance of the topic

The modern world is rapidly developing and causing changes in all parts of human life activities. And the art is not an exception of such changes. Coming through the ages, art had a lot of different forms depending on the development of the surrounding world. Today we have a great growth in technologies which causes new forms in the art to arise. These forms were unavailable before because of the absence of tools to realize them. Generative art is great and at the same time a beautiful way to represent a new form of modern art. This is the type of art that is performed and displayed with the help of the machines. With the help of modern technologies, people have an opportunity to take a look at art from a different perspective. Generative art is a special kind of art because the creation of some art pieces is done by human and machine together. In this case, the machine is represented by the autonomous system, that can make some decisions based on the randomness and the rules, that was set by human. That randomness can be controlled by human or can be organized in the special order, that will provide some limits by a human. As a result, we have a product of random limited by human restrictions, which is named as the art but looks like controlled chaos. This form of art is getting popular among artists in the modern art, pop-art, minimalistic art.

1.2 Goal

Generative art is a great example of the successful tandem of artist and machine. The best way to demonstrate the power and beauty of generative art is to try to reach perfection. And as we know, there is an assumption that nature is perfect. So it can be a great challenge to trying to show nature as close to real life as it possible with generative art. Nature is really multifaceted and wide, but one of the most beautiful and exciting facets of nature is the weather. The phenomenon of weather is independent of people, but people depend on the weather a lot. The weather has a lot of different conditions and ways how those conditions can be represented with the help of visualization. Also, there are a big amount of ways and nature rules that can cause a change in weather conditions. This point makes generative art a good way to demonstrate the beauty and wideness of weather. The demonstration of weather conditions can use a lot of different forms, however, all of them need to be located into the environment, which can reflect all conditions. The virtual environment, that can show all weather conditions should be close in its appearance to the natural look so it would be easy for a human to identify the real force, degree, or condition of the weather properties.

1.3 Objective

The virtual environment is a way of weather conditions representation. What does mean "virtual environment" term? The imaginary scene of nature, that will be able to visualize weather conditions. All possible weather conditions have to be defined to understand requirements for the virtual environment. The nature scene should display all four seasons of the year, the cloudiness of the sky, the precipitation (snow or rain), wind strength, and wind direction. The nature scene should have such components, which will display all of these factors. The main component of the landscape will be the tree. This component will be in the spotlight. The tree component will show the strength and direction of the wind. Also, it will be able to display the current season of the year. The tree component demands the ground component, as the tree needs to be grounded. The ground will be a large wide grass field. This component is able to display both the wind and the season. The only component, that is able to display the cloudiness is the sky. This component should be included in the landscape view. The last component is the precipitate. It will have a different form in accordance with the season of the year. The approximate scheme of future landscape view is shown in **fig.1.1**.

FIGURE 1.1: The scheme of landscape view.

Chapter 2

Background

2.1 Generative art

There is a common question: “what is generative art?”. Philip Galanter proposed a great explanation, who said that there are three blind men, who feel different parts of one elephant. The first man, who touches the elephant leg, exclaim that it is like a tree trunk. The second man, who touches the elephant trunk says that it is like a snake and the last man, who feels the elephant side believes that it is like a huge wall (Galanter, 2003). The same trend is with generative art. People explain generative art with those things, that they are doing on their own. However there is the most widely used definition of this kind of art supposed by Philip Galanter: “Generative art refers to any art practice in which the artist uses a system, such as a set of natural language rules, a computer program, a machine, or other procedural invention, that is set into motion with some degree of autonomy, thereby contributing to or resulting in a completed work of art.”(Galanter, 2003)

For a better understanding of the term “generative art” there are some historical facts about it. Both terms “computer art” and “generative art” are closely intertwined for a lot of time. The first generative art exhibition was held in 1965 in Stuttgart (Nake, 2005). It was named “Generative Computergraphik”. The works with generative art were represented by Georg Nees. Later he had made his Ph.D. work, related to the generative art (Nees, 1969). George attended the exhibition for several times when there was Frieder Nake with his works. Both artists called their work as generative art, which meant that works were made with a partly autonomous computer system(Boden, 2009). The first music piece composed with computer system was the "Illiac Suite for String Quartet". Both Lejaren Hiller and Leonard Isaacson were working on this project in 1957 (Hiller and Isaacson, 1958). However, the key person in computer generated music was Iannis Xenakis, who had made "Stochastic Music Program" and had articles on "Formalized Music"(Xenakis, 1971). Kenneth Martin was an abstract art artist, that is painted with simple geometric figures and with established rules. Later Martin added some randomness to his works (Martin, 1951/54). Nowadays there is an annual conference in Milan, which debuted in 1998. And Brian Eno is influencer and promoter of using generative art (Eno, 1996).Generative art in music and art fields is now widely used. The generative art term is used to determine the system, which has a part of an autonomous system and strict rules, that will limit the process of generation.

The generative art phenomenon is popular mostly for music and computer graphics with animations generation. The Markov chains were used to build the system, which helps to analyze musical scores. (Schwanauer and Levitt, 1993). When the gathered data was analyzed, the system was used to generate new music scores based on previous scores. Today there are a lot of new ways to generate music with a computer program, but the topic of this work is the visual generation. There is a

great number of different methods of visual generative art. One of them is Perlin Noise (Perlin, 1985). This system is used to imitate human hair, animal fur, smoke and etc. Also, the interesting approach is L-systems, that are used to make mostly plant patterns(Prusinkiewicz and Hanan, 1990). There are a lot of different interesting generative approaches described in the Association for Computing Machinery's (ACM) and Special Interest Group on Graphics and Interactive Techniques (SIGGRAPH) organizations (Galanter, 2016). There are a lot of sites with examples of modern generation art (*Intro to generative art*).

To sum up, generative art is an art form that has been created wholly or partially using autonomous systems. An autonomous system means that it doesn't need human and can make decisions about features in artwork without human help. Sometimes human creators of the generative system claim that the system represents their artistic idea, and in some cases, such claim is not provided. Generative art is mostly created by using algorithms, however, it can be made with systems of biology, mechanics, chemistry, mathematics, symmetry, data mapping, and others.

2.2 Procedural generation

Procedural content generation (PCG) is an algorithmic generation, which is usually used for game development. PCG has limited or even no human part. What is generated with PCG? All game content can be generated with procedural content generation. The game content is maps, vegetations, structures, rules, characters, dynamics and a lot of other things, that depends on game possibilities(Togelius and Stanley, 2013). The great and relevant to the current topic example of PCG is the SpeedTree system. This system is used as a generator of nature for commercial games. This system can generate different vegetations: grass, tree, brushes, and others.

PCG is actually often used in the generation of vegetations. As vegetations usually have a similar structure, they can be formed with PCG approach(KELLY and MCCABE, 2006). The big percentage of plants can be procedurally generated with the help of fractals or L-systems(HENDRIKX and A, 2013). There is a system, that growths the tree and takes to accordance the weather conditions to make the tree more realistic(WEBER and PENN, 1995). Procedural content generation can also make complex elements of nature. For example, clouds can be generated with Perlin noise or other PRNG techniques(RODEN and PARBERRY, 2005).

To sum up, procedural content generation (PCG) is content generation, which is performed by the program with a random or pseudo-random mechanism that allows getting a large number of different possible forms of content. Random has a big role in this due to the need of converting a few parameters into a great number of possible forms of content that are being created. Word procedural means the type of process, which performs some function. A good example of procedural generation is fractals, that are geometric patterns, which can repeat or textures and meshes. Procedural generation can be also used in such an area as sound or music, especially in different genres of electronic music. The procedural generation also found application in modern demoscene. It uses procedural generation to manage a large number of audiovisual content in small computer programs.

2.3 Fractals

Fractal art is a representation of procedural generation, which is based on algorithms. It is made by calculating fractal objects and visualizing it in different forms:

images, animations or media. Fractal art is a genre of abstract art, that is actually composition with computer art and generative art. Usually, fractal art is made by special software without the help of a human. Sometimes the result of fractal-generating software can be modified with other graphics programs, the name of this process is post-processing. There is an assumption that fractal art could never get so popular and accessible for humans without computers and their ability to quickly compute difficult calculations.

Fractals are the product of generation with iterations to solve non-linear and polynomial equations. Fractals can take different irregular shapes and curves, that repeats itself when it is decreased or increased in size(*Fractal Packs*). Fractals have some special properties as infinitely self-similarity, iterations and detailed math constructs that have fractal dimensions(*What is Fractal Management*). A lot of such dimensions have been studied greatly and are already formulated. Fractals are used not only to visualize beautiful patterns. For example, it can be used to show time processes. Even more, fractals with different degrees of self-similarity were discovered in structures, images, sounds, technologies, architecture, found in nature. Fractals have a connection with chaos theory as the graphs of a big mount chaotic processes have a form of fractals. The example of a fractal is illustrated in fig 2.1.

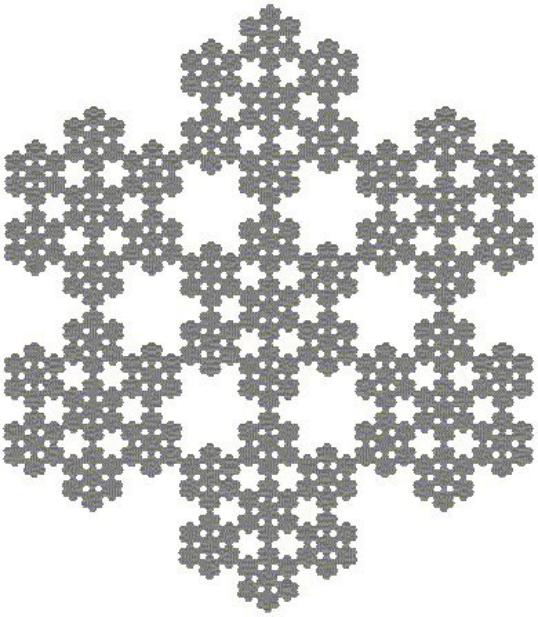

FIGURE 2.1: The snow particle made with fractals.

Chapter 3

Program implementation

3.1 Tree

3.1.1 Background

The developed systems main goal is to firmly represent the weather conditions in any given moment and one of the best visual tool to achieve this is a tree component. The ability of the tree to react to the wind in the most representative way can prove this point. Exactly the tree is the main indicator of the strength of the wind because of its location on the flat-rolled land. It can also be the best way to represent the current season of the year by the color or absence of leaves on it. The tree is the main and central component in the whole composition. Tree component also takes the biggest number of calculations per second.

3.1.2 Implementation

For a better understanding of the component, its main parameters should be declared and described here. As the most complicated and the biggest component tree demands a lot of properties be set by the artist for the right configuration. Variables w and h contain the current screen height and width for further implementation of adaptive tree representation. This approach demands a specific depth of the tree, that will define how many branches will be built from the trunk. The *maxDepth* property is responsible for the depth of the tree. It means that the number of branches from the trunk to any of ending branches will be equal to the *maxDepth* number. This variable was set with 13, which is the optimal way to represent a lush tree and avoid a lot of extra branches, that will not add to the visual part of the component but will decrease its performance.

Every branch of the tree has its own specific branch length to achieve a more realistic look. The *maxBranchLenght* and *minBranchLenght* variables define the maximum and minimum possible branch length. The *maxBranchLenght* variable is calculated with $Math.ceil(h/100)$formula. In such a way the tree general height will be greater on the bigger screens. Every branch (except the last branches) has two other branches and they should have their own angle of incline, which is limited by two variables: *minAngle* and *maxAngle*. The *bendability* property should always be greater than 1 for the correct work of the component. The greater the value of *bendability*, the more the thin branches will bend first. The tree structure is built with a fractal method. As the tree structure is formed with the fractal approach trunk should have its own x and *bendability* start coordinates. The x coordinate will be formed automatically according to the current screen width, but y coordinate should be set in the beginning according to the start line of the ground component. As the ground component y coordinate of the start line is set to the $h - 170$, the tree

should be set a little bit lower to avoid any unexpected blank space between the ground and tree. This value is set to *treeStart* variable.

The tree structure does not consist only from the branches, but also with leaves. The leaves also need some properties to be set. The *leafHeight* variable defines the height of leaves in the tree. The *leafDeviation* variable defines the angle, that will be used for further determination of leaves width. The more this value is, the wider the leaf is. But it influences not only the width of the leaf but also its shape. The more this value is, the more leaf will be more drop-like and vise versa. Leaves have its own *bendability* indicator, which is defined in the *leafBendability* variable. The bigger this value is, the more sensitive to the wind leaves become.

The main and important property of leaves is its colors. Colors of leaves will help to determine the current season of the year. To reach a more realistic look of the tree colors shouldn't be fully randomly generated, but with the special order. To implement it tree crown should be divided into few groups of branches and then these groups should be colored in their own manner. To form the groups in tree crown few branches should be chosen which will represent the start of the group. All branches, that will be generated from the specific branch will be in one group. To divide tree crown into approximately even groups it is needed to define the depth, on which branches will become the start of groups. For the tree with 13 maximum depth, the optimal depth for start division is 10, which is defined in the *branchGroupDepth* variable. The *leavesGroupSize* variable should be defined for the further proper leaves coloring. It shows how many leaves there are on each crown group. As each branch have two child branches(except the last branches) this value can be calculated by bringing 2 to the power of depth of the $group - 1$. The *groupCounter* variable will be used for further determination of the start of the new crown branch. Also, there is a defined array of colors for leaves in *leafColorArray* variable. This array should contain colors in a special order: from the darker colors to lighter colors. This order should be followed to reorganize this array and save it in the colors variable. The reorganization of the array consists of concatenating the same but reversed array with the current one. As a result, there is an array of colors, that starts with dark colors, in the middle with light colors and it goes to dark colors again. The *leafColorArrayLen* variable defines the length of the array.

The *treeCanvases* and *leavesCanvases* are used to save already painted canvas to optimize the process of drawing new canvases. The *generatedBranches* variable has a function to store primarily generated properties of branches. Variables branches and leaves are used as stores for the branches and leaves. The key features of those storages are that both leaves and branches are stored under the key, which is equal to the depth in the case of branches and color in the case with leaves. It stores canvases in such a way for improving the performance of drawing the figures on canvas. The *branchCounter* variable is used to iterate through the primary generated branches.

```
const w = window.innerWidth;
const h = window.innerHeight
const maxDepth = 13;
const maxBranchLenght = 8;
const minBranchLenght = 1;
const maxAngle = 20;
const minAngle = 15;
const bendability = 2;
const treeStart = 655;
const leafHeight = 20;
```

```
const leafDeviation = 120;
const leafBendability = 17;
const branchGroupDepth = 10;
const leavesGroupSize = 2 ** (branchGroupDepth-1);
let groupCounter = 0;
let leafColorArray = ['#b8d23d', ...];
const leafColorArrayLen = leafColorArray.length;
const colors = leafColorArray.concat(leafColorArray.slice(0).reverse());
let treeCanvases = {};
let leavesCanvases = {};
let generatedBranches = [];
let branches = {};
let leaves = {};
let branchCounter = 0;
```

After all needed properties are set it is time to implement tree structure. To begin tree generation it is important to understand how this tree is going to be built. The tree is going to be a fractal tree. A fractal tree is a structure, where the patterns are repeating and every part of a tree is going to be repeated, but with smaller or bigger scale. The tree is going to have a form of a binary tree. Each branch in the tree is going to have two child branches, and the child branches are going to have their own child branches. The exception is the head branch because it is going to be a tree trunk, so it does not have any parent branches. And also exception is the last two layers of branches in the whole tree. That is because the tree generates in such a way, that the last branch is actually not a branch, but the leaf. The branches, that are going to have as children leaves have only one child. In such a way every last branch has only one leaf in the end. The goal of tree structure generation is to form the array filled with branches parameters, that is rather properties of branches. Each branch should have its own angle of incline, its own length, and color for leaves coloring.

The fractal tree structure is going to be generated recursively. The $generate()$ function is recursively calling and requires angle for the current branch, the current depth of the tree and array as a store. The store array is gradually filling with three properties for each branch. When the depth of the tree is not equal to 0 (the end of the tree) and at the same time is more than 1 (not the last branch of the tree) function $generate()$ is calling twice to generate two child branches. In the other case it means that it was the last branch and now is time to generate leaf, so function $generate()$ is calling only once. When the depth is equal to 0 it means that it is the end of tree and recursion stops. To begin the recursion in the beginning the function is called with such arguments: $generate(-90, maxDepth, arr)$. The first angle is -90, as it is the angle for the tree trunk and so it will always appear as straight vertical.

The angle for each next branch is formed by adding or subtracting the current branch angle and random angle for the next branch. The angle for the new branch is chosen randomly between defined limits in $minAngle$ and $maxAngle$. When it is not the last branch and function is called twice, the one call of the function should have the sum of current and next angle as an argument and the other call of a function should have the angles difference. In such a way two new child of the branch will be aimed in the different ways relative to the current branch. In the case when it was the last branch, the angle for the next branch, (which is actually a leaf) is the same. Next property of each branch is its length, which is randomly chosen between $minBranchLenght$ and $maxBranchLenght$. This random length for tree branches is used to achieve a more realistic look of the tree. The tree looks too symmetric and

unnatural when the branches have the same length. The difference between a tree with branches of the same length and tree with random branches length is illustrated in **fig.3.1**.

FIGURE 3.1: The fractal tree with even branch length and the fractal tree with random branch length.

The third property, that should be generated, is color, which is used only when leaves are painted. The tree coloring is quite a complicated task because of the need to consider that the leaves colors are not random. If the leaves color will be random, the tree would look like on **fig.3.2**. The color of tree leaf depends on many factors, but one of the most important is the light. The leaves, that are in the deep of tree crown have darker colors and leaves that are located on top of the tree have lighter colors. Because of that, the crown of the tree is divided into groups. Each group of tree crown is colored in the same way: in the middle, it is colored with the darker colors and on the sides, it is colored with lighter colors. The fractal tree becomes more natural when there are several groups inside of the tree crown. The *groupCounter* variable is needed to implement the division of tree crown. This counter is increasing by 1 every time when one leaf is generated. The value of counter resets to 0 when the generation of the new crown group starts. So it means that this counter shows the number of current leaf in its crown group. Each crown group is vertically divided into several parts, where each part will be colored into its own color. Function divide(min, max, units, value) is used to determine in which part of the crown current leaf is located. This function divides the interval between min and max into smaller even intervals. The units is a number of intervals in which larger interval should be divided. This function determines which of that intervals the value belongs and returns the index of that interval. The formula inside the function is $min + Math.floor(Math.random() * (max + 1 - min))$. The 0 and *leavesGroupSize* will be as minimum and maximum for divide function, as the *leavesGroupSize* is a number of leaves in one group. The number of parts in one group is equal to the number of possible colors. The *groupCounter* will be used as value argument. The result of the function call will be the index of the needed color in colors array. The fractal tree will look like in **fig.3.2** after dividing its crown into groups.

FIGURE 3.2: The fractal tree with random colors of leaves and the fractal tree with grouped leaves.

```
function generate(angle, depth, arr) {
 let randomLeafColor = colors[divide(0, leavesGroupSize,
   (leafColorArrayLen - 1) * 2, groupCounter)];

  arr.push({
   angle,
   branchArmLength: random(minBranchLenght, maxBranchLenght),
   color: randomLeafColor
 });
 if (depth === branchGroupDepth) { groupCounter = 0; }
 if (depth === 0) { groupCounter++; }
 if (depth != 0) {
   if (depth > 1) {
     generate(angle - random(minAngle, maxAngle), depth - 1, arr);
     generate(angle + random(minAngle, maxAngle), depth - 1, arr);
   } else {
     generate(angle, depth - 1, arr);
   }
 }
}
```

The tree is not a static component as it can be changed by the wind. The branch function role calculates new coordinates for branches according to new wind value. It updates with every new wind value. This function is recursively implemented in the same way as it is inside of generate() function. The function calculates only coordinates for the end of a branch, as the coordinates for start of the branch is the coordinates of the end of the parent branch. Each branch move is actually a circular motion, which depends on its length, depth, angle, parents angle, wind strength, and wind direction. To calculate new coordinates for the branch $calcX(angle, r)$ and $calcY(angle, r)$ functions is used. These functions are made to determine x and y coordinates for point (the orange point at **fig.3.3**), that is moving in the circular trajectory. Both functions need to have two arguments: angle of incline to the x-axis (

in **fig.3.3**) and radius of the circle. To calculate coordinates of point it is needed to multiplicate radius with the cosine of the angle or sine of the angle. So the formula for $calcX()isr * Math.cos(angle)$ and formula for $calcY()$ is $r * Math.sin(angle)$. In the case of the tree branch, the radius of the circle will be the length of the current branch multiplicated by current depth. Usually, a natural tree has its top branches thinner and shorter than branches, that are near the trunk. The length of the branch is multiplicated by the depth to make top branches smaller than branches below. If all branches will have almost the same length, the tree would look too unnatural.

The angle for each branch is built by this formula $angle + wind * windSideWayForce * bendabiityOfCurrentBranch$. The angle is each branch own angle, that was randomly generated in generate() function. The wind is the strength of the wind. The windSideWayForce variable is formed by $(windX * yy - windY * xx)$formula, where windX and windY can be -1, 0 or 1. These variables are taken from wind config and define in which side wind is blowing. The xx and yy values are formed by$calcX(dir, depth)$ and $calcY(dir, depth)$ functions. In such a way $windSideWayForce$ shows the direction of the wind. The last factor for the branch angle is $bendabiityOfCurrentBranch$. This value is extracted by this formula: $(1 - (depth * 0.7) / (maxDepth * 0.7)) * *bendability$. This calculation is needed to make top branches bend stronger than lower branches (the branches with less depth). As the thickness of the branch is formed according to its depth, it makes thin branches be more sensible to the wind gusts.

The similar calculations are needed to determine the angle for leaves angle, but there are some different points related to leaves depth. As the depth of leaves is always equal to 0, the value of $windSideWayForce$ will be equal to 0 too. In such a way during the calculation of xx and yy values, there should be used 1 instead of depth in $calcX()$ and $calcY()$functions. Also, leaves own $bendability$ force $leafBendability$ should be used instead of $bendabiityOfCurrentBranch$ value. Both leaves and branches are saved into new objects with specific keys. The depth is a key for branches and the color is key for leaves.

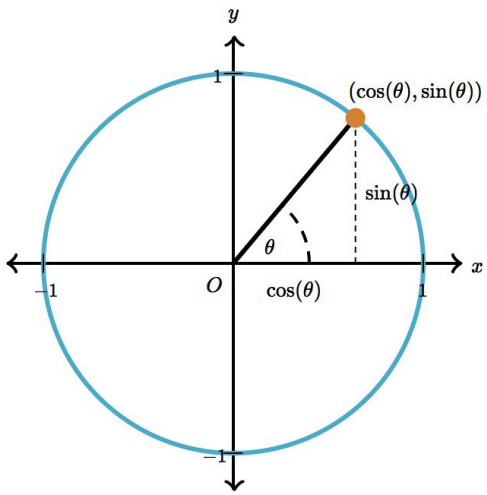

FIGURE 3.3: Illustration of the circle unit.

```
function branch(x1, y1, arr, depth, windConfig) {
 if (depth != 0) {
   dir = angle + wind * bendabiityOfCurrentBranch * windSideWayForce;
   let x2 = x1 + calcX(dir, depth * branchArmLength);
   let y2 = y1 + calcY(dir, depth * branchArmLength);
   lines[depth].push([x1, y1, x2, y2]);

   if (depth > 1) {
     branch(x2, y2, arr, depth - 1, windConfig);
     branch(x2, y2, arr, depth - 1, windConfig);
   } else {
     branch(x2, y2, arr, depth - 1, windConfig);
   }
 } else {
   const leafAngle = angle + wind * windSideWayForce * leafBendability;
   leaves[color].push([x1, y1, leafAngle]);
 }
}
```

When all branches and leaves are generated its time to visualize it. The drawing on the canvas is a quite time-consuming thing. There is an assumption, that for better performance each canvas should have a specific limit of painted objects. That is why tree branches and leaves are divided into two different canvases. Both branches and leaves are saved into the object with special keys for another performance optimization. The function *beginPath*() should be called to begin drawing something on canvas and *closePath*()should be called after all drawing manipulations are over. This two action takes a lot of time in the program process, so the use of these two functions should be minimized. However, there is one problem with avoiding using these two functions: the stroke color, fill color and stroke width cannot be changed in the scope of one path. For example, two different paths are needed to draw two similar rectangles, but with the different fill color. For that reason branches were grouped and saved in the object by the depth (as the stroke width depends on depth). Now it is easy to go through the object and draw all branches with the same width in the scope of one canvas path. The code snippet below shows the process of exactly drawing each branch. When there is an array of branches with the same depth, the *lineWidth* variable can be defined for all of them. The program starts to iterate through the array after the path was started. Each branch is painted with a simple line, that has start coordinates and the end coordinates. After all branches were painted path closes and all lines get stroked at the same time.

```
context.lineWidth = depth * 0.7;
context.beginPath();
   while (lines.length) {
     const [x1, y1, x2, y2] = lines.pop();
     context.moveTo(x1, y1);
     context.lineTo(x2, y2);
   }
   context.closePath();
   context.stroke();
 });
```

The leaves are implemented with a similar method of optimization. All leaves are grouped by color and saved by color as a key. In addition to optimization, the store of leaves should be sorted by the lightness of colors to draw light leaves in the end. As light leaves have such light color because of the sun, they should be not in the depth of a tree. Also, the process of leaves drawing is more complicated because of its convex form. This shape of a leaf can be implemented with bezier curves. Bezier curves can be formed with three, four or more control points. For leaf is enough two quadratic bezier curves (Armstrong, 2005). This curve is formed with three control points as it is in **fig.3.4**. Two points are responsible for the end and start of the curve. The other one point is responsible for convex of the curve. Each leaf is painted with four control points as for its implementation is needed two quadratic bezier curves. There is an illustration on fig.4.5 of tree leaf, formed with two quadratic bezier curves. Also, four control points that form tree leaf are shown. The blue points show the start and end of the curves. The yellow points are used to form convex of the curve. The canvas has already implemented function $quadraticCurveTo(x1, y1, x2, y2)$, which is drawing quadratic bezier curve. This function takes four arguments: the first two are coordinates of the control point, that forms deviation of the curve and the next two arguments are coordinates of the control point, which indicates the end of the curve. The coordinates for the start of the curve will be set at that point, where the previous curve has the end. In the case when there is no previous curve the start of the current curve should be set with canvas function $moveTo(x, y)$.

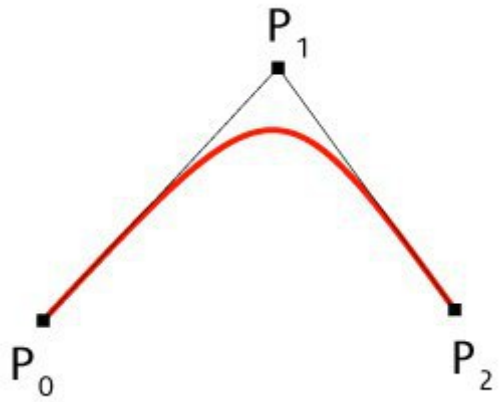

FIGURE 3.4: Quadratic bezier curve.

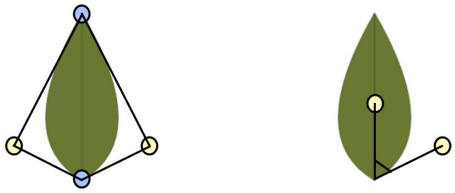

FIGURE 3.5: The tree leaf visualization with quadratic bezier curve.

The control points for tree leaf depend on the leaf start coordinates, leaf angle, leaf height, and shape. The start coordinates of the leaf are x and y. The next steps of generation are to determine another three points, where $x2$ and $y2$ will be the coordinates of the leaf end. The $x3$, $y3$, $x4$, and $y4$ are the coordinates of points, that forms the curve convex. As the leaves are moving on the circular trajectory, the $calcX()$ and $calcY()$ functions will help to define all needed control points. To calculate coordinates for the leaf top, leaf height will be used as the radius and the leafs angle will be used as an angle. The control points for curve convex is calculated with half leaf height as a radius. These control points angle depends on the angle of the leaf. The angle for one of that control points will be the $angle + leafDeviation$. The other control point will be calculated in the same way, but the angle will be caluclated as $angle - leafDeviation$. The $leafDeviation$ is the angle, that illustrated at **fig.3.5.** To make the leaf more realistic line is painted, this line goes through whole leaf height. In the end leaf is filled with its color. It also has a stroke, to color the lines and leaf borders. The color of stroke is the next color in the palette. In such a way borders of the leaf is always a little bit darker than the leaf itself.

```
context.fillStyle = color;
context.strokeStyle = strokeColor;
context.beginPath();
     while (leaves.length) {
         const [x, y, angle] = leaves.pop();
         let x2 = x + calcX(angle, leafHeight),
         y2 = y + calcY(angle, leafHeight),
         x3 = x + calcX(angle + leafDeviation, leafHeight / 2),
         y3 = y + calcY(angle + leafDeviation, leafHeight / 2),
         x4 = x + calcX(angle - leafDeviation, leafHeight / 2),
         y4 = y + calcY(angle - leafDeviation, leafHeight / 2);

         context.moveTo(x, y);
         context.quadraticCurveTo(x3, y3, x2, y2);
         context.quadraticCurveTo(x4, y4, x, y);
         context.moveTo(x, y);
         context.lineTo(x2, y2);
     }
context.closePath();
context.fill();
context.stroke();
```

As it was said earlier, the process of drawing on the canvas is quite complicated and takes a lot of time. In spite of all previous optimization, there are still too many

calculations and drawings. To improve performance, there was a decision to save already painted canvases. Canvas can be saved in a variable and can be used without redrawing all content. As the redrawing of tree canvas depends on the wind, the canvases will be saved in the object with wind values as a key. In such a way, when the new wind is generated, the program will first look if there is already saved canvas for such a wind value. If there is painted canvas, it will be used, but if there is not such canvas it will be painted and saved. However, this approach has one disadvantage. Some period of time should pass before all possible canvases will be generated and saved. That is why at the beginning of program all canvases are generating with all possible wind values, while the user sees loader on the screen. It takes some time, but when the program starts, the user will see really smooth drawing with a great performance. This optimization approach makes performance works three times better than without it.

3.2 Ground

3.2.1 Background

Because the main weather representative is a tree component it needed someplace it would look appropriate. The rolling field with grass cover is a good way to complement the tree component and visualize weather conditions in its own manner. The grass on the field can show the same weather conditions as a tree but in the other way. The grass field can perfectly represent the wind gusts by bending grass piles on the field. Grass can also illustrate the current season of the year by changing the color of grass piles and its structure. In the early spring, the grass is small and thin with a pale color of grass piles. But till the end of the spring grass is going to be tall and thick with saturated pile colors. Through the summer grass is not going to change, but with the beginning of autumn in the grass will appear some yellow and orange colors. Till the end of the autumn, all grass will become dry, thin and yellow. The grass will also bend to the ground as it is natural for the tall grass in the field. When winter will come and bring snow, the whole ground will become white and some dry grass piles will stay above the snow.

3.2.2 Implementation

Ground implementation needs some properties to be set in the beginning. The variable colors is 2d array, which contains groups of colors, that will be used to color the grass. To achieve the feeling of a big rolled field with grass and avoid feeling that there is one grass row right in front of the user display colors should be divided into groups. The farthest layers of grass have to be painted in more smooth and light colors and the closest layers of grass to be painted in more contrast colors. The smooth transition should be provided between the most contrast and the most smoothed groups of colors. In this particular case, the field is divided into six layers, which is the optimal number for the field with such a height. This value is defined in the $fieldAreas$ variable.

The h and w variables contain the values of the height and width of the current user screen. All needed constant parameters for grass implementation is set below. Two variables $grassWidth$ and $grassHeight$ will provide the width and height of each pile of grass. The $fieldTopStart$ variable has a value, which indicates a point that will be the start point of the field from the top. This value depends on the current user screen height because of the grass field should have the same height independently

to the user's screen. That is why this point of field start is calculated by$h - 170$. The 170 number, in this case, is actually the field height. The $fieldBottomDeviation$ will indicate how much space should be added to the bottom of the screen for the formation of the grass fields bottom line. The number variable has a number of piles of grass that will be generated on the grass field. This variable depends on the width of the current screen. When the screen has 1440px, there will be approximately 4000 blades of grass on the grass field. In the nature piles of grass is rarely vertically straight, so for the more realistic look of grass, each blade will have its own angle of incline to the ground. So the $maxAngleDeviation$ variable indicates the max angle of pile incline. Usually, in a real grass field, every pile is moving in the wind with different speeds. To achieve this feeling there are defined two variables $minSpeed$ and $maxSpeed$ that will show max and min speed value for piles. Variable canvases will be used to store already painted canvases. Variable $grassDots$ will be used to store x and y coordinates on the field of each pile of grass.

```
const colors = [["#7f9032", ...], ["#6b7f26", ...]];
const fieldAreas = 6;
const grassWidth = 3;
const grassHeight = 70;
const fieldTopStart = 650;
const fieldBottomDeviation = 40;
const number = 4000;
const maxAngleDeviation = 15;
const minSpeed = 2;
const maxSpeed = 6;
const canvases = {};
const grassDots = [];
```

When all the needed properties of grass are provided, the process of grass implementation starts. This process starts with generating every grass pile with its own parameters. The $generate()$function will provide at its output the array of generated properties for each grass pile. Each pile should have such parameters as x, y, $color$, $angle$, and $speed$. To achieve a more realistic look of the grass field each grass pile will be located on the field randomly. That's why the x coordinate is randomly generated between the start and end of the screen and y coordinate is randomly generated in the space, which is limited by $fieldTopStart$ above and height of screen + $fieldBottomDeviation$ value. Extra space for the field in the bottom of the screen is provided to generate some grass piles below the user view. With such an approach, the user will see in the tops of grass piles at the bottom of the screen, which will achieve the feeling that the user is located right in the middle of the field. In addition, it will help to avoid having blank space on the ground.

As the common color of grass piles is changing vertically from smoothed to more contrast there should be provided proper color for each grass pile according to its location on the y-axis. If all grass piles would be chosen randomly without color layers the field will look unnatural as on **fig. 3.6** The grass field should be conditionally divided into some horizontal layers, where each layer will have its own group of colors, that will color grass piles on that area. For the implementation of grass layers, every pile of grass has to belong to one of the six layers. The function divide(min, max, units, value) will help with the determination of the proper group of colors for each pile. The work and implementation of this function are explained in the tree component. This function will return the index of the needed color in the color array.

In the case of the grass field, the min and max arguments have to be the bottom and top limits of the field. The units argument takes the number of parts in the field and the argument value takes the y coordinate of the grass pile. When there is an index of needed color group, color for a pile is chosen randomly from that group.

The next property of the grass pile is the angle of incline to the ground and it's generated randomly between $-maxAngleDeviation$ and $maxAngleDeviation$ because grass pile can be inclined into different sides. The last property for grass pile is the speed of movement due to the wind and it is randomly chosen from the interval between minimal and maximal speed value. When the array of piles properties is generated it should be sorted by y coordinate from the smallest to the largest one. The sorting of the array is needed for further grass piles drawing. When the array is not sorted all piles are drawing in the random locations on the field as coordinates of piles are generated randomly. In such a case, piles each other in the wrong order and the field of grass is getting messy, which is illustrated in**fig. 3.7(b)**. However, if the array will be sorted, the farthest piles will be painted first and the closest piles will be painted last, which will make the field of grass looks closer to real life.

```
function generate(number) {
 for (var i = 0; i < number; i++) {
   var y = random(fieldTopStart, h + fieldBottomDeviation);
   var x = random(0, w);
  var colorGroup = divide(fieldTopStart, h + fieldBottomDeviation  + 1,  fieldAreas, y);
   var color = colors[colorGroup][random(0, colors[colorGroup].length)];
   var angle = random(-maxAngleDeviation, maxAngleDeviation);
   var speed = random(minSpeed, maxSpeed);
   dots.push([x, y, color, angle, speed]);
 }
 dots.sort();
}
```

(A) The grass without color layers.

(B) The grass without sorting.

(C) The grass field.

FIGURE 3.6: Different ways to draw grass.

After the process of generating all needed parameters for the grass piles, it's time to draw it on the canvas. Every grass pile consists of two quadratic bezier curves (Armstrong, 2005). It is the same curves, that are explained in tree component while drawing tree leaves. In the case of a grass blade, three control points for each curve is enough for its visualization. Each grass blade is painted with five control points as for its implementation is needed two quadratic bezier curves. On the **fig. 3.7** there are illustrated three different examples of drawing grass pile with two quadratic bezier curves. On the picture is also shown that each grass pile has five control points. The blue control points mark start and the end of the curves and yellow control points form the deviation of curves. The canvas has already implemented function $quadraticCurveTo(x1, y1, x2, y2)$, which is drawing the quadratic bezier curve. The work of this canvas function is already explained while the tree leaves drawing. First, two arguments of the function are coordinates of the curved end, and the second two arguments are coordinates of a point, that shows curve convex.

In the process of visualizing blade of grass, some important factors came up, that have an influence on its appearance: angle, wind, and speed. The angle and speed are properties of each pile, generated in the beginning and wind contains the value of actual wind in the current moment. This value is taken from the wind config and is changing several times per second. When the value of wind changes all canvas is re-drawn with new shapes of blades in such a way there is a feeling of the moving of grass piles. To perform the proper movement for the top of the grass pile (coordinates of the first curve end) the speed on the x-axis should be twice as large as the speed of points, that form curve deviation. Also to improve the movement of grass pile the changing of the y coordinate of the grass pile top is calculated. The module is needed as the wind can have negative values, which is important for the x-axis, but not for the y-axis. With such an improvement in the case of stronger wind, the pile will not only move with the wind, but it also will incline to the ground, which will make grass field movement more realistic.

```
context.fillStyle = color;
context.beginPath();
context.moveTo(x, y);
context.quadraticCurveTo(
     x - wind * speed + angle,
     y - grassHeight / 2,
     x - grassWidth / 2 + wind * speed * 2 + angle,
     y - grassHeight + Math.abs(wind * speed + angle)
);
context.quadraticCurveTo(
     x - grassWidth - wind * speed + angle,
     y - grassHeight / 2,
     x - grassWidth,
     y
);
context.fill();
context.closePath();
```

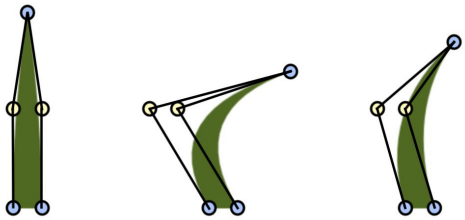

FIGURE 3.7: The grass pile implementation using quadratic bezier curve.

The process of the grass field drawing is quite consuming as there are about 4000 grass blades on the field and every blade demands some calculations. All 4000 grass blades are re-drawn with the new value of the wind to show the movement. In addition, the process of drawing on the canvas takes some extra time. To improve the performance of the grass field moving in the wind, the canvas store was implemented as it was implemented with three components. It is a simple object, where the keys are the value of the wind and the values are already painted canvases. In such a way every time when the wind gets a new value the program will firstly look for already painted and stored canvas and show it if it's already present, which is more efficient and much quicker then calculating and drawing new canvas every frame of time.

3.3 Sky

3.3.1 Background

The landscape with ground and tree is incomplete without the sky component. The sky is an important part of any landscape view. It is also important as a possible way to represent current weather conditions. The sky can not show the current season of the year, but it is the only component that can show how cloudy the weather is. Clouds are a very important part of the weather conditions. The clouds can also show wind strength and its direction. The sky look depends only on the index of cloudiness. The clouds are very interesting natural phenomenon, as they have strange shapes and specific way of movement.

3.3.2 Implementation

For a better understanding of how clouds are going to be implemented, all definitions should be explained. The w and h variables define the current screen width and height. It is needed to make the view of component responsive. Every cloud has its own canvas, that can overlay on each other and form bigger clouds. The sky is actually one big canvas with a background gradient color of the current sky. The sky canvas also contains the clouds canvases and draw them with $drawImage()$ canvas function. The $cloudMinWidth$ and $cloudMaxWidth$ variables define the minimum and maximum possible cloud width. The same function has the $cloudMinHeight$ and $cloudMaxHeight$ variables, but they define the minimum and maximum possible height of the cloud. In such a way each cloud has its own width and height. The $cloudCanvasWidth$ and $cloudCanvasHeight$ variables contain the values of the width and height of clouds canvases. All canvases of clouds have the same width

and height, but clouds have different sizes, as they do not need to fill all canvas and have their own parameters. The width and height of cloud canvases are bigger than the maximum possible width and height of the cloud because of cloud will be not static and will be changing its shape. The bigger size of the canvas will help to avoid clouds go outside its canvas. The *cloudWidth* and *cloudHeight* variables are calculated by a random choice between their limits and define the width and height for the current cloud.

The *cloudsAmount* variable defines the number of clouds on the sky. This variable in app depends on the value from API, which shows how cloudy the weather is. Both *speedOnSky* and *speed* variables show the speed for clouds. The clouds in the sky usually move in some direction on the sky with some speed. This speed is set to the *speedOnSky* variable. However, if to look closer to the clouds, there is more movement than just across the sky. Each cloud is also always moving inside itself, causing a change of its form. The speed variable defines the speed of movement the cloud inside itself. Each cloud consists of particles, that will be moving in the circular trajectory. Both these variables in the app depend on the wind strength. The radius variables the radius of the circle movement for cloud particles.

As it was in previous components, the clouds have to be optimized too. That is why we need the canvases variable, that will be store for already painted canvases. However, the difference is that in previous components the process of saving the keys for saved canvases was the wind value. In the case with clouds, the frame variable will be the key for saved canvases. The cloud particles will be filled with the image, which will improve the performance of the program. There are two types of cloud particles, that will be colored into different images. The particle variable stores the image for the main particle of the cloud and *innerParticle* variable stores the image for inner particles.

```
const w = window.innerWidth;
const h = window.innerHeight;
const cloudMaxWidth = 200;
const cloudMinWidth = 80;
const cloudMaxHeight = 50;
const cloudMinHeight = 20;
const cloudCanvasWidth = 350;
const cloudCanvasHeight = 200;
const cloudWidth = random(cloudMinWidth, cloudMaxWidth);
const cloudHeight = random(cloudMinHeight, cloudMaxHeight);
const cloudsAmount = 15;
const speedOnSky = 0.1;
const speed = 1;
const radius = 10;
let frame = 0;
const canvases = {};
const particle = new Image();
particle.src = icons.cloud;
const innerParticle = new Image();
innerParticle.src = icons.innerCloud;
```

After all needed constants are defined becomes the time to generate the clouds. The task of *generateClouds*() function is to produce the array filled with the properties for each cloud. This function needs only one argument: the number of clouds,

that should be generated. Each cloud should have three properties: the function, that draws the cloud and coordinates of its location on the sky canvas. The first argument is the call of cloud function, which generate the random cloud. The x coordinate can be any point on the screen width and even outside the screen. This causes the generation of clouds, that are not fully visible for user and it makes the sky more natural. The y coordinate can be at any point between the top of screen and cloud own height. Thus the clouds are located on a different height on the sky.

As the clouds are not static components, they need to have recalculation function for their coordinates. The$updateCloud()$function updates the coordinates for each cloud canvas according to its new location on the sky. The $updateCloud()$function makes the clouds to move across the sky. This function is quite simple. It just adds the *speedOnSky* to the current x coordinate of the cloud canvas or subtracts *speedOnSky* from the x coordinate of the cloud canvas according to the wind direction. This makes the clouds move. When the cloud is going outside the right side of screen, its x coordinate becomes $-cloudWidth$. When the cloud is going outside the left side of the screen its y coordinate is. So, when the cloud goes outside the one side of the screen it appears on the other side of the screen. It appears outside the other side of the screen and with wind goes inside the screen. In such a way all clouds are repeating, what is good for program performance.

```
function generateClouds(amount) {
 const clouds = [];
 for (let i = 0; i <= amount; i++) {
   const getCloudCanvasFn = cloud();
   const startinPosX = random(-cloudWidth, w);
   const startinPosY = random(0, cloudHeight);
   clouds.push([getCloudCanvasFn, startinPosX, startinPosY]);
 }
 return clouds;
```

The process of generation each cloud is not very complicated. As it was said, each cloud consists of particles. There are two types of particles: basic particles and inner particles. The difference is in the look of those particles. The basic particles have a light color and are located in the whole cloud area. The inner particles are darker and are located at the bottom part of the cloud area to make an illusion of the cloud shadow. The amount of particles depends on the area of the current cloud. The particles and *innerParticles* variables define the number of basic particles and inner particles. The number of inner particles should be smaller, than the number of basic particles. The particles of the cloud should be properly located on the cloud canvas. The inner movement of the cloud is represented by moving these particles. While the cloud canvas is moving the whole cloud on the x axes, the particles have their own circular trajectory of movement. Thus each particle should have its *circleX* and *circleY* coordinates, that are not actually coordinates of the particle. The particle will be moving around this point. The properties, needed for particles drawing are angle, x and y. The angle of the particle is generated randomly. Because of the random angle for each particle, they will move like a random. In spite of the same radius and speed, it will look chaotic because of different angles. Some particles will be moving to the bottom and some will be moving to the top at the same time. These coordinates are calculated in such a way, that the particle can be located in any place inside the cloud area and not go away outside this area.

The $generateParticles()$ function below is the function for generating an array with properties for basic particles. There is the similar function $generateInnerParticles()$

, but it is for inner particles. The difference between these functions is that *circleX* and *circleY* are calculated in different ways. That is because basic particles should be located on the whole cloud area and inner particles should be located on the bottom cloud part. In that reason, coordinates for basic particles are calculated with $random(radius, cloudWidth/cloudHeight)$. The radius was used as a minimum to avoid particle falling outside the canvas. The same coordinates for inner particles are calculated with $random(40, 40 + cloudWidth/cloudHeight)$. The 40 is a top and the left margin of the cloud. In such a way only basic particles will be on the top and the left side of the cloud. These manipulations with cloud particles are needed to achieve the effect of cloud shadow and cloud inner movement, which is illustrated in **fig.3.8**. As the particles have to move on the circle trajectory, the only one thing, that should be changed is the angle. There is a $recalculateParticlePosition()$function, that adds to the angle the speed variable. In such a way the particles are moving with the same speed.

```
const particles = (cloudHeight * cloudWidth) / 250
const innerParticles = (cloudHeight * cloudWidth) / 330

function generateParticles(cloudWidth, cloudHeight) {
   const cloud = [];
   for (var i = 0; i < particles; i++) {
     var circleX = random(radius, cloudWidth)
     var circleY = random(radius, cloudHeight)
     var angle = random(0, 360)
     cloud.push([angle, circleX,  circleY]);
   }
   return cloud;
}
```

FIGURE 3.8: The implementation of the cloud with particles

```
const particles = (cloudHeight * cloudWidth) / 250
const innerParticles = (cloudHeight * cloudWidth) / 330

function generateParticles(cloudWidth, cloudHeight) {
   const cloud = [];
   for (var i = 0; i < particles; i++) {
     var circleX = random(radius, cloudWidth)
     var circleY = random(radius, cloudHeight)
     var angle = random(0, 360)
     cloud.push([angle, circleX,  circleY]);
   }
   return cloud;
}
```

The process of drawing cloud particles is very simple. As the particles are moving on the circular trajectory, there is a need to calculate the x and y coordinates of the particle position. These coordinates are calculated with the use of $calcX()$ and $calcY()$ functions. These functions were explained in the tree implementation subchapter. To get the proper coordinates of the point is needed to pass the angle and radius properties. The circle center coordinates should be added to the results of functions in accordance. When the coordinates of the particle are calculated it should be painted as an image with $drawImage(img, x, y)$ function. The two different particles were generated and saved into images. Both basic particle and inner particles were generated with $createRadialGradient()$ context function. The differences between these two particles are the color and circle center. The inner particle has a darker color and shifted the circle center to the right bottom corner. The examples of both particles are shown in the **fig.3.9**.

```
function drawCloud([angle, circleX, circleY], img, context) {
  var newX = calcX(angle, radius);
  var newY = calcY(angle, radius);
  const x = newX + circleX;
  const y = newY + circleY;
  context.drawImage(img, x, y);
}
```

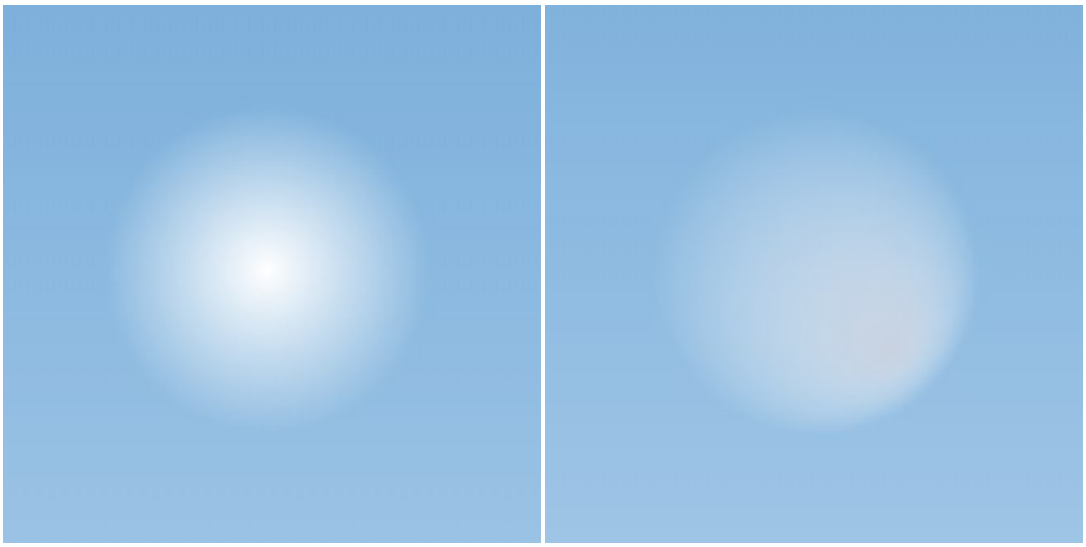

FIGURE 3.9: The visualization of basic and inner particles

The cloud implementation has the same optimization with canvas saving as the previous grass and tree components. However, there is one difference. As each cloud has its own canvas, all those canvases should be saved inside each cloud function. As the inside cloud movement is independent of the wind, there is a frame variable, that will be the key for canvases. This variable is calculated with every canvas update with $(frame + speed)\%360 formula$. In such a way the frame value will go from 1 to 360 and then again becomes the 1. So, when all particles have made the full circle all canvases will be already saved. Also, a lot of time in clouds generation takes gradient generation. That is why it was made the decision to generate the particles gradient, make images from it and use only gradient image. This improvement made the program performance faster in several times.

3.4 Precipitation

3.4.1 Background

The last, but very important visual component is precipitation. The visualization of weather conditions will be incomplete without displaying current precipitation. It is the only component that can show the precipitation index on the screen. However, it can also show the current season of the year, as in the winter precipitation change its form. So, there are two types of precipitation: rain in the warm weather and snow in the cold weather. The precipitation also can show the direction of the wind, as its particles are falling down in the wind direction.

3.4.2 Implementation

The snow is the first type of precipitation, that will be explained. There are some constants, which should be defined before the generation process. There are w and h variables, that define the current width and height of the screen. The particles array has a role of the snow particles store. The *particlesAmount* variable defines how much snow particles are painted on the screen. This value depends on the snow index from API in app implementation. As the snow particle is actually a circle, it should have a radius. The *minRadius* and *maxRadius* values define the minimum

and maximum possible radius for snow particle. The *minXSpeed* and *maxXSpeed* variables have values of minimum and maximum possible speed for snow particle. When the particle falls outside the screen it will appear again in the top of the screen. The snow particle will be generated a little bit higher than the screen top to avoid sudden particle appearance in the middle of the sky. The *particleStart* variable defines the value, which indicates how much higher than the screen particle will appear. The particles should be generated not only along the screen width but a little bit wider. Because of wind particles may be falling in one direction, the screen sides may be empty. The *sidesDeviation* variable is used to avoid such situations.

```
w = window.innerWidth;
h = window.innerHeight;
var particles = [];
var particlesAmount = 500;
const minRadius = 0.5;
const maxRadius = 5;
const minXSpeed = -3;
const maxXSpeed = 3;
const particleStart = -50;
const sidesDeviation = 0.5;
```

The function below shows the process of snow particles generation. Each particle should have its coordinate on x and y axes. The x coordinate is calculated according to the current screen width. The possibility of snow particles to appear wider than a screen with is also taken into account. The y coordinate is calculated in the same way according to the current screen height. Snow particles also have different color opacity, which is chosen in the generation process. Each particle should have its own radius, which is randomly chosen between $minRadius$ and $maxRadius$. Snow particle is always falling down with a specific speed, that defined in $YSpeed$ variable. This speed depends on the radius of the particle. In such a way bigger particles will be falling down faster, than small particles. It will help to achieve a feeling, that bigger and faster particles are located closer to the screen like in the 3-d scene. Particles are not falling straight vertically, so they need to have $XSpeed$ too. This one is calculated by a random choice between its maximum and minimum possible values.

As the particles are always changing their coordinates, they need to have some update function, which will recalculate particles coordinates. This function adds to the current x particle coordinate $XSpeed + wind$. The x movement of the particle also depends on the wind to make the snow go to the wind direction. The y coordinate is calculated by adding $YSpeed$ to it. When the y coordinate of the particle becomes more than screen height, the location of this particle should be re-generated. The x coordinate generates at the same way, as it was in$generate()$function, and y coordinate has a *particleStart* value. In such a way particle appears on the top of the screen again.

```
function generate() {
 for (var i = 0; i < particlesAmount; i++) {
   var r = random(minRadius, maxRadius);
   particles.push({
     x: random(0 - sidesDeviation, 1 + sidesDeviation) * w,
     y: Math.random() * h,
     opacity: Math.random(),
```

```
    radius: r
    YSpeed: r,
    XSpeed: random(minXSpeed, maxXSpeed),
  });
}}
```

The snow particle is actually a simple circle, but with a radial gradient fill and specific opacity. The gradient for snow particle is made with$createRadialGradient()$ canvas function. It consists of three different colors, that are created with the opacity variable. The circle for snow particle is implemented with $arc(x, y, radius, 0, Math.PI * 2, false)$canvas function, which requires x and y coordinates of the circle, the circle radius, start angle, end angle, and clockwise argument. When the circle is painted, it will be filled with an already generated gradient. The result is shown in **fig.3.10**.

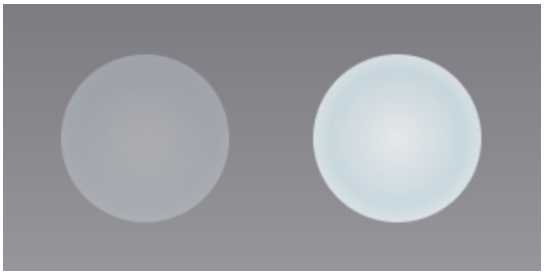

FIGURE 3.10: Generated snow particles with different opacities.

Another type of precipitation is rain. The rain implementation has a similar structure as the snow implementation. However, there are some differences. The rain consists of particles too, but these particles are lines. As the drops are falling from the sky, the user can see something similar to the line instead of drop. Each particle of the rain should have the same parameters as the snow particle, but instead of opacity and radius, there is length property. This property defines the length for each line. The particle's movement on the sky implemented in the same way as it was with snow particles. Each particles is painted with $moveTo(x, y)$ and $lineTo(x2, y2)$ canvas functions. The coordinates for $moveTo()$ function are x and y particle properties. The x2 and y2 coordinates for $lineTo()$ function are calculated with $x + length * XSpeed + wind$ and $y + length * YSpeed$ formulas in accordance. In such a way The line will be directed into the same way as the wind. The implementations of both precipitations are quite similar.

Chapter 4

Experiments

There are some landscape views shown below to display all the components together. Examples demonstrate different seasons and weather conditions.

FIGURE 4.1: Sunny weather in the summer.

FIGURE 4.2: Windy weather in the summer.

FIGURE 4.3: Windy and rainy weather in the summer.

FIGURE 4.4: Cloudy weather in the autumn.

FIGURE 4.5: Snowy weather in the winter.

Chapter 5

Conclusion

The goal of this work was the implementation of a landscape view, that will represent weather conditions with generative art. There were determines separate landscape components, that had to be implemented: tree, precipitations, ground, and sky. All of them had to be not only generated but also animated in accordance with the current weather conditions. As a result, all of those components were implemented successfully. As a result, there is a completed system, that is able to display any of the possible weather conditions in a quite accessible way. The visual landscape components have enough natural look to inform the human about the current weather outside.

Generative art is a really powerful method of visualization animated things. It can make the live tree only with math formulas, which is actually magic. The animated landscape visualization represents all weather conditions, that were required at the beginning. However, there are a lot of things, that can be improved as nature has no limits in its perfection.

This work is about generative art and its possibilities. The best way to show the full beauty of generative art is to try to display something perfect. Nature or rather weather conditions is a great choice. In this work was anal made a system, that displays natural animated landscape, that should show in an accessible way the weather conditions. The weather conditions were analyzed and a special landscape view was prepared. Every part of the full landscape composition was generated by the system without using any libraries. Every part of landscape was made with inspiration and love.

Bibliography

Armstrong, Jim (2005). "Quadratic Bezier Curves". In: *TechNotes in Macromedia Flash*.

Boden M.and Emonds, Ernest (2009). *What is generative art?'* Centre for Cognitive Science, University of Sussex, Creativity Cognition Studios, University of Technology, Sydney.

Eno, B (1996). "Generative Music: Evolving metaphors, in my opinion, is what artists do." In:

Fractal Packs. https://fractalfoundation.org/fractivities/FractalPacks-EducatorsGuide.pdf.

Galanter, Philip (2003). "What is Generative Art?" In: *Interactive Telecommunications Program*.

— (2016). *Generative Art Theory*.

HENDRIKX M., MEIJER S. VAN DER VELDEN J. and IOSUP A (2013). "Procedural Content Generation for Games: A Survey". In:

Hiller, L. and L. (1958) Isaacson (1958). "Musical Composition with a High-Speed Digital Computer". In: *Journal of the Audio-Engineering Society*.

Intro to generative art. https://dev.to/aspittel/intro-to-generative-art-2hi7.

KELLY, G. and MCCABE (2006). "A survey of procedural techniques for city generation". In: *ITB Journal*.

Martin, K (1951/54). "Abstract Art". In: *AIA exhibition catalogue*.

Nake, F. (2005). "Computer Art: A Personal Recollection". In: *Proceedings of the Fifth Conference on Creativity and Cognition*.

Nees, G (1969). "Generative Computergraphik". In: *Siemens AG*.

Perlin, Ken (1985). "An Image Synthesizer". In: *SIGGRAPH Computer Graphics*.

Prusinkiewicz Przemyslaw, Aristid Lindenmayer and James Hanan (1990). "The Algorithmic Beauty of Plants". In: *New York: Springer Verlag*.

RODEN, T. and PARBERRY (2005). "Clouds and stars: efficient real-time procedural sky rendering using 3d hardware". In: *ACE '05*.

Schwanauer, Stephan M. and David A. Levitt (1993). "Machine Models of Music". In: *Cambridge, MA: The MIT Press*.

Togelius J., Champandard A. Lanzi P.L. Mateas M. Paiva A. Preuss M. and Kenneth O. Stanley (2013). "Procedural Content Generation: Goals, Challenges and Actionable Steps". In:

WEBER, J. and J PENN (1995). "Creation and rendering of realistic trees". In: *SIGGRAPH Annual Conference on Computer graphics and Interactive Techniques*.

What is Fractal Management. https://davidboje.com/fractal/Chapter%201%20What%20is%20Fractal%20Management.htm.

Xenakis, I (1971). *Formalized Music: Thought and Mathematics in Composition*. Bloomington, Indiana University Press.