Szegedi Tudományegyetem

Informatikai Intézet

Fraktál bemutató keretrendszer

Szakdolgozat

Készítette:
Péter Albert
programtervező informatikus
szakos hallgató

Témavezetõ:
Tóth Zotán Gábor
egyetemi tanársegéd

Szeged
2020

Tartalomjegyzék

Feladatkiírás

A szakdolgozatom központi témájául a fraktálokat választottam. Matematikában nem túl jártas egyéneknek lehetséges, hogy nem túl sokat mondó az a fogalom, hogy fraktál. Az én célom egy olyan webes applikáció elkészítése volt, amelyet használva mindenki mélyebb belátást nyerhet a fraktálok világába. A webes applikációban nyolc előre megírt algoritmus szerint generálhatunk fraktálokat. A felhasználóknak lehetőségük van az algoritmusok testreszabására, minden algoritmus egyéni konfigurálási lehetőségekkel rendelkezik. Ilyenek például az elforgatási szög módosítása, tetszőleges méretű, pozíciójú, valamint tetszőlegesen rotálható gyökér elem megadása, növekedés irányának módosítása, szín beállítása... Az applikáció arra is lehetőséget biztosít, hogy valamely időpontban az algoritmus futását szüneteljük, valamint ha szeretnénk, egy csúszka használatával visszatekerhetjük az algoritmus pillanatnyi állapotát valamely előző állapotra. Ezen kívül az alkalmazásban egy olyan funkció is található, amely segítségével a felhasználók a fraktálok generálása az adott fraktálról egy rövid leírást olvashatnak, ezzel is lehetőséget adva a felhasználóknak, hogy tudásukat bővítsék. Végül a generált fraktálokat kép formájában kiexportálhatjuk a saját gépünkre, ha szeretnénk.

Tartalmi összefoglaló

A dolgozat legfőbb célja a webalkalmazás munkamenetének leírása. A feladat egy fraktál bemutató keretrendszer elkészítése volt, amelyben többek között olyan hasznos funkciók találhatók, mint az algoritmusok konfigurálhatósága, az algoritmusok futásának szüneteltetése, az algoritmus állapotának visszaállítása egy korábbi állapotra, illetve a generált fraktál kiexportálása kép formájában. A dolgozat magába foglal egy bevezető szekciót, ahol a fraktálokról általános tudnivalókat olvashatunk. Itt lényegében a fraktálok eredetéről, fontosabb tulajdonságairól, illetve főbb felhasználási területeiről van szó. Ezen kívül, a dolgozat további részeiben a magáról webalkalmazásról van szó, ismertetem a felhasznált technológiákat, valamint részletezem a fejlesztés menetét.

Bevezetés

A fraktálokról átalánosságban

A fraktálok egy viszonylag újnak nevezhető terület a matematikában, és habár komolyabban vizsgálni őket csak a számítógépek megjelenése után kezdték el igazán, már a XX. század előtt is fogalkoztak a matematikusok ezekkel a különös geometriai alakzatokkal. Az elnevezést 1975-ben Benoît Mandelbrot adta, a latin fractus (vagyis törött, törés) szó alapján, ami az ilyen alakzatok tört számú dimenziójára utal, bár nem minden fraktál tört dimenziós (ilyenek például a síkkitöltő görbék). Tört dimenziósnak nevezik azokat az alakzatokat, amelyek nincs területük, többek mint egy egyenes, viszont kevesebbek mint egy síkidom. Nagyszerű példa erre a Sierpiński-szőnyeg. Mandelbrot *The Fractal Geometry of Nature (A Természet Fraktálgeometriája)* című munkája mutatta be, és magyarázta el először a fogalmakat, melyek alapjául szolgálnak ennek az új területnek.

A fraktálok önhasonló, végtelenül komplex matematikai alakzatok, melyek változatos formáiban legalább egy felismerhető ismétlődés tapasztalható. Az önhasonlóság azt jelenti, hogy egy kisebb rész felnagyítva ugyanolyan struktúrát mutat, mint egy nagyobb rész. Ilyen bizonyos léptékig például a természetben a villám mintázata, a levél erezete, a hópelyhek alakja, a fa ágai. A fraktál szóval rendszerint az önhasonló alakzatok közül azokra utalnak, amelyeket egy matematikai formulával le lehet írni, vagy meg lehet alkotni. A fraktálok egy másik tulajdonsága, hogy sehol sem differenciálhatók. Ennek oka az, hogy bár a fraktálok folytonosak, végtelenül gyűröttek. A fraktálokat úgy írhatjuk le, hogy megvizsgáljuk, hogyan változnak különbözô felbontásoknál.

A fraktálok felhasználási területei

A fraktálok gyakorlati felhasználásának köre folyamatosan bővül. Rengeteg helyen találkozhatunk velük a művészetektől az orvosláson át a számítástechnikáig. Ma már bizonyított tény, hogy fraktálok, illetve fraktálszerű alakzatok a természetben is gyakran előfordulnak. Megfigyelések alapján tudjuk, hogy ezek az alakzatok egy meghatározott növekedési struktúrát követnek. Ilyenek a fák, kristályok, vagy a felhők. Számtalan művész használta a fraktálok ábrázolásának technikáját, annak ellenére, hogy pontos definícióját, matematikai hátterét nem ismerték. Olyan neveket érdemes itt megemlíteni, mint például Vincent Van Gogh, vagy akár Maurits Cornelis Escher. A építészetben is sok önismétlődő részletet találhatunk gondoljunk csak a gótikus és barokk épületek ismétlődő támpilléreire, oszlopsoraira.

A fraktálok kiválóan alkalmazhatóak képi és hanganyag tömörítésére, feldolgozására. Ha egy hangot vagy képet fraktálokra bontunk, utána az adott darabot leíró algoritmussal és paramétereivel könnyen összehasonlíthatóvá válnak. Továbbá az algoritmusok és a hozzá tartozó paraméterek letárolása kevesebb területet vesz igénybe, mintha a nyers adatokat tárolnánk le.

Háttér

Az alkalmazás elkészítése előtt kiválasztottam azokat a fraktál generáló algoritmusokat, amelyeket implementálni szerettem volna. Nyolc algoritmust választottam, ezek a következők:

- Sierpiński- háromszög
- Sierpiński-szőnyeg
- Fraktál-fa
- Pitagorasz-fa
- Lévy C-görbe
- Koch-görbe
- Hilbert-görbe
- H-fa

Választáskor többek között figyelembe vettem az algoritmusok bonyolultságát, milyen, és mennyi konfigurációs lehetőséggel tudom felruházni az adott algoritmust, valamint a fraktálok fajtáját. A választott algoritmusok között vannak helykitöltő algoritmusok is, mint például a Hilbert-görbe és a H-fa. Ezek az algoritmusok nagyjából kivétel nélkül már mások által implementálva van, a forráskódjuk bárki számára elérhető az interneten, viszont ezek az algoritmusok egyike sem támogatja a konfigurálhatóságot, ami az én alkalmazásom egyik alapköve.

Az implementáció elkezdése előtt utánanéztem olyan webes alkalmazásoknak, amik hasonlóak, mint az én tervezett alkalmazásom. Találtam hasonlót, ahol egy-egy fraktál

generáló algoritmus valamilyen szinten konfigurálható, viszont ezeken az alkalmazások jellemzően csak egy-egy fraktállal foglalkoznak különösebben, így csak az adott fraktál generálására van lehetőségünk, és kevesebb konfigurálási lehetőséggel rendekeznek, ha egyáltalán rendelkeznek, mint az én tervezett webalakalmazásom, valamint nem támogatnak kiexportálási és visszatekerési lehetőséget. Az algoritmusok konfigurálása ezekben az alkalmazásokban egyszerű, jelölönégyzettel, vagy szám típusú beviteli mezőkkel történik, hasonlóan, mint ahogyan én is terveztem.

Az interneten való keresgélést követően kiválasztottam azt a technológiát, amellyel magát a keretrendszert implementálni szerettem volna. Mivel ez egy webes alkalmazás, a választás az Angular keretrendszerre esett, pontosabban az Angular 8-as verziójára. Ez nem a legfrissebb verziója az Angular keretrendszernek, viszont korábban is használtam, és stabilitás szempontjából ez az egyik legjobb, ugyanis vannak olyan különböző API-k, amelyek még nem teljesen kompatibilisek az Angular legújabb verziójával. A fontosabb Angular API-k, amiket használtam a fejlesztés során a következők: Angular Material, Angular Flex Layout és Rxjs. Az Angular egy TypeScript keretrendszer, amellyel, a benne található csomagok segítségével, rendkívül kényelmesen és könnyen készíthetünk Single Page webalkalmazásokat. Ezen kívül a JavaScript-es könyvtárak nagyon nagy részével kompatibilis, ami nagyon nagy előny volt számomra, hiszen az algoritmusokat P5.js-ben implementáltam, ami egy nyílt forráskódú, vászon alapú rajzolásra alkalmas JavaScript könyvtár. Az alapok nagyon könnyen elsajátíthatóak, és a közismert alakzatok nagy részének a rajzolása előre definiált függvények segítségével történik, ami rendkívül kényelmessé teszi a használatát. Ezen kívül vektorok használatát is támogatja, a vektorműveletek már előre definiálva vannak benne, amelyeket az algoritmusok implementálásakor előszeretettel használtam. Egyik funkcionalitása, ami még nagyon sokat segített az algoritmusok implementálásakor, az a translate függvény, amely segítségével a vászon koordinátarendszerének origóját tudjuk eltolni. Ez egy rendkívül hasznos függvény, különböző rotációknál, és olyan alakzatok rajzolásánál, amelyek pozíciója az egérmutató pozicíójától függ. Ezek a műveletek a függvénynek köszönhetően sokkal kevesebb számolással kivitelezhetőek.

A fejlesztés Windows 10 operációs rendszer alatt történt. A programozáshoz, futtatáshoz, teszteléshez a Visual Studio Code fejlesztői környezetet használtam. A szoftver

verziókövetése, illetve a fájlok tárolása GitHub repository-n keresztül valósult meg.

Az alkalmazás alapfunkciói

A alábbi fejezetben az elkészült alkalmazás főbb funkcióit mutatom be, fejtem ki részletesen, külön kitérve a fontosabb, érdekesebb részekre.

Lejátszást vezérlő funkciók

A fraktál generáló algoritmusok futását három gombbal tudjuk vezérelni, ezek a Play, Pause és Start gombok. Mivel az alkalmazást megpróbáltam minél több komponensre bontani, így kellett egy külön szolgáltatás, amelynek a feladata az volt, hogy az egyes eseményeket, amelyeket a felhasználók a gombok megnyomásával váltanak ki, közvetítse az algoritmusnak, amely a fraktál generálására szolgál. Így az algoritmus tudni fogja mikor kell szünetelni, mikor kell rajzolni, és mikor kell újrakezdeni. A szolgáltatás kódja a következőképpen néz ki:

```
export class AnimationStateManagerService {
        private state: BehaviorSubject<boolean>;

        constructor() {
                this.state = new BehaviorSubject<boolean>(false)
                    ;
        }

        setState(state: boolean): void {
                this.state.next(state);
        }
```

```
        getState(): Observable<boolean> {
                return this.state;
        }
}
```

A szolgáltatás alapja tehát egy *BehaviorSubject* objektum, amely *boolean* típusú adatot tud közvetíteni a feliratkozóknak. Amikor a felhasználó a lejátszást vezérlő gombok egyikével interakcióba lép, a szolgálatás *setState()* metódusa hívódik meg, amelynek a paramétere egy igaz-hamis adat, attól függően, hogy az algoritmus futása szünetel-e vagy sem. Amikor ez a metódus meghívódik, akkor a szolgáltatás a paraméterként kapott változó értéket közvetíti a feliratkozóknak, így azok tudni fogják, hogy a felhasználó interakcióba lépett a vezérlő gombok valamelyikével. A feliratkozó ebben az esetben az algoritmus, amely az adott fraktált generálja.

Algoritmus lista és konfigurációs panelek

A webalkalmazás két oldalsó panellel rendelkezik. Bal oldalon a fraktál generáló algoritmusok egy oszlopba rendezett listáját tekinthetjük meg. Az algoritmusok mindegyike rendelkezik előnézettel, tehát aki nem ismerné valamelyik algoritmust, láthatja, hogy az adott algoritmus milyen fraktált generál. Ezek az előnézetek maguknak az algoritmusoknak egy lebutított változatai, konfigurációs lehetőségek nélkül, az adott algoritmusok paraméterei alapértékekre vannak állítva. Minden algoritmus előnézete egy bizonyos szintig iterál, majd ha ezt a szintet elérte, újraindul.

Jobb oldalon a konfigurációs panel található. Ennek a tartalma dinamikus, aszerint generálódik, hogy az adott algoritmusnak milyen konfigurációs lehetőségei vannak. A konfigurálás három féle módon történhet:

- csúszka segítségével állíthatunk be értéket az adott algoritmus valamely paraméterének, egy megadott értékkészleten belül
- jelölőnégyzet ki-be pipálásával tudjuk jelezni, hogy az adott konfigurációs lehetőséget használni szeretnénk-e
- színpaletta segítségével tudjuk a fraktál színét beállítani

Az algoritmusok listáját egy külön szolgáltatás tárolja. Minden komponens, amelynek szüksége van erre a listára, mint például a fent említett két panel, ettől a szolgáltatástól kéri el. Az algoritmusok listájának modellje a következőképpen néz ki:

```
export interface IAlgorithmList {
        name: string;
        previewId: string;
        preview: any;
        algorithm: any;
        configurations: IAlgorithmConfiguration[];
        about: string;
}
```

Az algoritmusok listájában tárolódik tehát minden algoritmus

- neve
- előnézetének azonosítója, amely az előnézet megjelenítéséhez kell
- a fraktál előnézetét kirajzoló algoritmus referenciája, amely a tényleges algoritmus egy lebutított változata
- a tényleges algoritmus referenciája, amely már konfigurációs lehetőségekkel rendelkezik
- az adott algoritmus konfigurációs lehetőségei, tömbben tárolva
- egy pár mondatos ismertető az adott algoritmusról, ez egy külön funkció lételeme, amelyről a későbbiekben lesz szó

A lista panelnek az algoritmusok előnézet azonosítójára, és az előnézet algoritmus referenciájára van szüksége, a konfigurációs panelnek pedig az algoritmusok konfigurációs lehetőségeire, amelyek tömbben tárolódnak. Ezen kívül mindkét panel eltüntethető és előhozható egy-egy gomb segítségével, ezzel is növelve az alkalmazás kompaktságát. Az eltüntetés és előhozás animációval történik, amely abból áll, hogy gomb megnyomásra, CSS szinten tolom a panelek pozicióját X tengelyen jobb vagy balra, attól függően, hogy

előhozni, vagy eltüntetni szeretnénk a paneleket. Az animáció megvalósítására az Angular Animations API-ját használtam. Ennek a menete, hogy a CSS animációkat egy tömbben definiáltam, majd hozzákötöttem az adott panelekhez egy boolean változó segítségével. A boolean változó értéke a gomb nyomására változik, így fogja tudni az alkalmazás, hogy eltüntetni, vagy megjeleníteni kell.

Visszatekerés funkció

A visszatekerés funkció egy sajátos funkciója az alkalmazásomnak, hiszen egy hasonló alkalmazás sem rendelkezik ilyen lehetőséggel. Alapja egy csúszka, ami, ha a felhasználó interakcióba lép vele, az algoritmust visszatekerő módba helyezi. A csúszka kódja az alábbi módon néz ki:

```
<div class="slider-container">
<input #slider (click)="rollBack(slider.value)" type="range" min
    ="0" [max]="sliderLength" [value]="sliderLength" class="
    slider" id="myRange">
</div>
```

Lényegében ez egy egyéni stílusú, range típusú input mező, amelynek a max és a value direktivákkal lehet értéket beállítani. A value paraméter mondja meg, hogy a csúszka éppen hol áll, ez alapértelmezetten 0-ra van állítva. A max paraméter azt adja meg, hogy a csúszka milyen hosszúságú, ennek az értéke az algoritmus futásával párhuzamosan változik, amit egy BehaviourSubject típusú változóval oldottam meg. Lényegében, amikor az algoritmus egy új iterációját rajzolja le a fraktálnak, ez a BehaviorSubject objektum egy új értéket közvetít a feliratkozóknak, aki jelen esetben a csúszka, így annak a hossza mindig a megfelelő értékre lesz beállítva. Ha a csúszkára kattint a felhasználó, az adott algoritmus visszatekerő módba kerül, és meghívódik a rollBack metódusa, amellyel minden algoritmus rendelkezik. A visszatekeró mód annyit jelent, hogy beállítódnak azok az értékek, amelyeket kiolvasva, az algoritmus tudni fogja meddig kell visszatekerni a fraktált. Ennek a kódja így néz ki:

```
rollBack(p: any) {
        if (this.rollBack) {
```

```
                if (this.play) {
                        for (let i = 0; i < this.list.length; i
                            ++) {
                                this.list[i].draw(p);
                        }
                        this.rollBack = false;
                }
                else {
                        p.background(this.canvasColor);
                        for (let i = 0; i < this.rollBackTo; i
                            ++) {
                                this.list[i].draw(p);
                        }
                }
        }
}
```

Ez a függvény minden képfrissítéskor újrarajzolja a hátteret, ennek köszönhetően eddig megrajzolt alakzat törlődik a vászonról. Ezután a függvény végigiterál azon a listán, amelyben az algoritmus tárolja a visszatekerés előtt még megrajzolt alakzatokat, és ez alapján rajzolja vissza azokat az elemeket, amelyek szükségesek. Minden algoritmus rendelkezik ilyen listával, hiszen ez a visszatekerés funkciójának egy fontos alapeleme.

Fraktál leírás funkció

Mindegyik fraktált generáló algoritmus rendelkezik egy leírással. Ebben a leírásban egy rövid ismeretőt olvashatunk az adott fraktálról, ki fedezte fel, hol használják, jellegzetes tulajdonságok megemlítése. A leírást egy előugró ablak tartalmazza. Ez az előugró ablak egy globális komponens, minden algoritmus esetén ugyanaz az ablak ugrik elő, csupán a tartalma változik dinamikusan. Az ablakot egy gomb megnyomásával hívhatjuk elő. Az ablak HTML kódja így néz ki:

```
<div>
        <h1>{{data.title}}</h1>
```

```
        <p>{{data.about}}</p>
        <button (click)="close()" mat-button>Bezár</button>
</div>
```

Ez egy eléggé rövid kódsor, azonban látszik, hogy a címet és magát a leírást a data változó tartalmazza. A data változó akkor definiálódik, amikor az ablakot előhozó gombra kattint a felhasználó, az értéke pedig az algoritmus listából olvasódik ki, attól függően, hogy éppen melyik fraktál generáló algoritmus aktív. A gomb megnyomására a következő függvény fut le:

```
about(): void {
        let dialogRef = this.dialog.open(AboutComponent, {
                width: '800px',
                data: {
                        title: this.title,
                        about: this.algorithmList[this.
                            activeAlgorithm].about
                }
        });
}
```

A width property értelemszerűen az előugró ablak szélességét adja meg, ebben az esetben az ablak 800 pixel széles lesz. Magasságot azért nem definiáltam, hogy különböző hosszúságú szövegek esetén se csússzon szét az ablak, a data változóról pedig pár sorral fentebb volt szó.

Algoritmusok

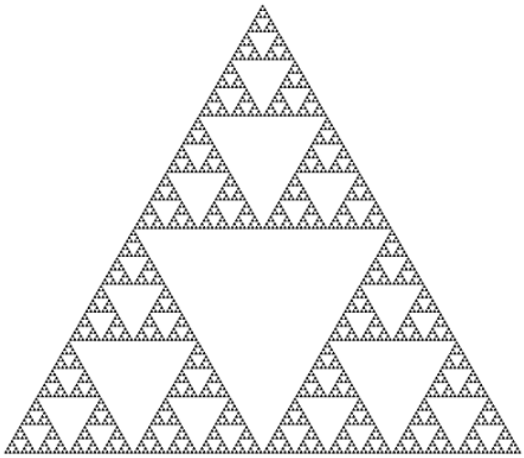
A alábbi fejezetben azokrók az fraktál generáló algoritmusokról lesz szó, amelyeket a webalkalmazás részeként leimplementáltam. Minden algoritmusról egy először rövid ismertetőt írok, majd bemutatom az implementáció menetét.

0.1. Sierpiński-háromszög

A Wacław Sierpinski lengyel matematikus által megtalált fraktál úgy áll elő, hogy egy szabályos háromszögből elhagyjuk az oldalfelező pontok összekötésével nyert belső háromszöget, majd az így maradt három háromszögre rekurzívan alkalmazzuk ugyanezt az eljárást. Hausdorff-dimenziója log(3)/log(2) 1,585. A Sierpiński-háromszög konstrukciójához többnyire egyenlő oldalú háromszöget választanak. Ez azonban nem kötelező, bármely háromszögből lehet Sierpiński-háromszöget készíteni. A Sierpiński-háromszög konstrukciójának lépései:

- Rajzolj három pontot, melyek a háromszöget határolják be, mindegyik pont a háromszög egy csúcsa
- Rajzolj egy pontot egy tetszőleges helyre, ez lesz a kezdőpont
- Válaszd ki a háromszög egyik csúcsát valamilyen véletlenszerű módon, majd a választott csúcsot és a kezdőpontot összekötő képzeletbeli vonal közepére rajzolj be egy újabb pontot, ez lesz az új kezdőpont
- Ismételd ezeket a lépéseket

Minden lépésben a keletkező kis háromszögek oldalhossza megfeleződik, és területük a negyedére csökken, miközben a középső háromszög eltűnik.



1. ábra. Sierpinszki háromszög

0.1.1. Implementáció

A Sierpinszki-háromszög implementációjához, először egy segédosztályt írtam, mint ahogy a többi algoritmus nagy részének implementációja is segédosztályok segítségével valósult meg. Ez a segédosztály a Point osztály, mivel a Sierpinszki háromszöget az algoritmus pontok segítségével rajzolja ki. Itt definiálva vannak a pontokhoz tartozó adattagok, ez ebben az esetben egy vektor, ahol tárolódik a pont x és y pozíciója a vászon koordináta rendszerében, valamint egy draw függvény, ami kirajzolja az adott pontot. A Point osztály kódja így néz ki:

```
export class Point {
        public point: p5.Vector;

        constructor(point: p5.Vector) {
                this.point = point;
        }

        draw(p: any): void {
                p.point(this.point.x, this.point.y);
        }
}
```

Az implementáció többi része a fő osztályban történt meg, ami a SierpinskiTriangleCon-

figurable osztály. Itt többek között definiálva vannak az algoritmus konfigurációs lehetőségei is, amelyek a következők:

- Gyorsaság
- Pontvastagság
- Véletlenszerű pontvastagság
- Pontok közötti távolság
- Fixált kezdőpontok
- Háromszög részeinek kijelölése
- Szín
- Szivárvány mód

Minden algoritmus rendelkezik egy setup és egy draw függvénnyel, amelyek a p5 könyvtárhoz tartoznak, és megvalósításuk elengedhetetlen a vászonra való rajzoláshoz. A draw függvény minden képpontfrissítéskor lefut, a setup viszont csak egyszer, mielőtt a rajzolás elkezdődne. Itt állítódnak be azok a paraméterek amelyekkel a rajzolást elkezdjük, például pontvastagság, vonalvastagság, szín, képfrissítési ráta, valamint itt jön létre a vászon objektum. Maga a rajzolás a draw függvényben történik. A SierpinszkiTriangleConfigurable osztály setup és draw függvényei így néznek ki:

```
p.draw = () => {
        this.setConfigurables(p);

        if (this.play) {
                let rand = p.floor(p.random(3));
                if(this.randomStrokeWeight) {
                        p.strokeWeight(p.random(0, 10));
                }

                if(this.rainbowMode) {
                        let h = p.map(p.floor(p.random(this.list
                            .length)), 0, this.list.length, 0,
                            360);
                        p.stroke(h, 255, 255);
                }
                else if (this.customColors && rand == 0) {
                        p.stroke(this.color1);
                }
                else if (this.customColors && rand == 1) {
                        p.stroke(this.color2);
                }
                else if (this.customColors && rand == 2) {
                        p.stroke(this.color3);
                }

                let newPoint = p5.Vector.lerp(this.points[rand],
                        this.refPoint,
                        this.lerpValue);
                p.point(newPoint.x, newPoint.y);
                this.refPoint = newPoint;

                this.list.push(new Point(newPoint));
                this.rollBackList$.next(this.list);
        }
}
```

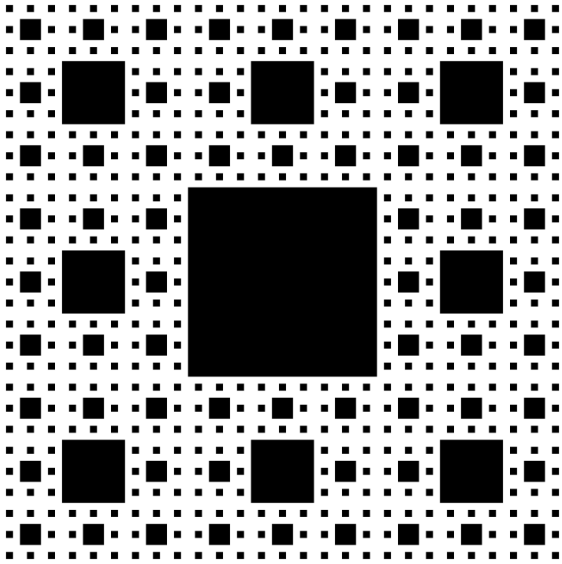
Ez egy lerövidített kód, mivel az eredeti túl hosszú, így megpróbáltam csak a lényeget szemléltetni. Látható hogy a rajzolás a különböző feltételekhez van kötve, amelyek a konfigurációktól függnek, mint például a véletlenszerű pontvastagság és a szivárvány mód. Továbbá látható, hogy a függvény csak akkor fog rajzolni, ha az algoritmus futása éppen nem szünetel. A points tömb tárolja a három pontot, amelyek a háromszöget határolják be, a refPoint változó pedig azt a pontot, amelytől a háromszög egy véletlenszerűen választott csúcsáig számítódik az a félút, ahová az új pont kerül. A véletlenszerű csúcsot a függvény a p5 random metódusa segítségével választja ki. A félút a p5 könvytár beépített lerp függvényével számolódik ki. Az refPoint változó minden új pont rajzolása esetén értékül kapja ezt az új pontot. A rollBackList$ változó a draw függvény minden lefutása után, pontosabban minden új pont kirajzolása után, egy új értéket emittál, ezzel beállítva a visszatekerő csúszka hosszát.

0.2. Sierpinski-szőnyeg

A Sierpiński-szőnyeg szintén Wacław Sierpiński lengyel matematikus által megtalált fraktál, amely úgy áll elő, hogy egy négyzetet oldalai harmadolásával kilenc kisebb négyzetre bontunk, a középsőt elhagyjuk, és a maradék nyolcon elvégezzük ugyanezt az eljárást (vagyis azoknak is elhagyjuk a közepét), majd az így maradt 8×8 kisebb négyzeten is, stb. Az eredményül kapott alakzat területe nulla, kerülete végtelen nagy. Hausdorff-dimenziója log 8/log 3 1,8928. A Sierpinski-szőnyeg konstrukciójának lépései:

- Vegyünk egy négyzetet
- Osszuk fel minden oldalát három részre
- A kijelölt pontokat összekötve osszuk fel a négyzetet kilenc kis négyzetre
- Töröljük el a középső négyzetet
- Ismételjük az előző lépéseket minden kis négyzetre.

Ezzel az eljárással a négyzet egyre inkább kiürül. Végtelenszer megismételve a Sierpiński-szőnyeg marad.



2. ábra. A Sierpinski-szőnyeg 4. iterációja

0.2.1. Implementáció

Az implementációt, a Sierpinszki-háromszöghöz hasonlóan, itt is egy segédosztály megírásával kezdtem. Ez a segédosztály a Rectangle osztály. A segédosztály adattagként tárolja az adott négyzet középpontját, ami egy vektor objektum, és méretét. Mivel ennek a forráskódja hosszú, így csak azt a függvényt szemléltetem, amely a kilencedelést végzi. Ez az alábbi módon néz ki:

```
divide(): Rectangle[] {
        let rectangles = [];

        rectangles.push(new Rectangle(
                new p5.Vector(this.center.x - this.size, this.
                    center.y),
                this.size / 3));
        rectangles.push(new Rectangle(
                new p5.Vector(this.center.x - this.size, this.
                    center.y + this.size),
                this.size / 3));
        rectangles.push(new Rectangle(
```

```
                new p5.Vector(this.center.x - this.size, this.
                    center.y - this.size),
                this.size / 3));
        rectangles.push(new Rectangle(
                new p5.Vector(this.center.x + this.size, this.
                    center.y),
                this.size / 3));
        rectangles.push(new Rectangle(
                new p5.Vector(this.center.x + this.size, this.
                    center.y + this.size),
                this.size / 3));
        rectangles.push(new Rectangle(
                new p5.Vector(this.center.x + this.size, this.
                    center.y - this.size),
                this.size / 3));
        rectangles.push(new Rectangle(
                new p5.Vector(this.center.x, this.center.y +
                    this.size), this.size / 3));
        rectangles.push(new Rectangle(
                new p5.Vector(this.center.x, this.center.y -
                    this.size),
                this.size / 3));

        return rectangles;
}
```

Ez a függvény mindig egy négyzetre van meghívva, így az ő kilenced részét osztja fel további kilenc részre. Az algoritmus egy kezdő négyzettel indul, ami a vászon közepén helyezkedik el, így az algoritmus első iterációjában erre a négyzetre van meghívva a divide függvény. Az algoritmus konfigurációs lehetőségei a következők:

- Gyorsaság
- Kezdő négyzet mérete

- Szín
- Szivárvány mód

Ez az algoritmus konfigurációs lehetőségek terén nem olyan színes mint például a Sierpinszki-háromszög. Ez annak köszönhető, hogy egyszerűen nincs annyi állítható paraméter, mint az említett fraktál algoritmus esetén. A kevés konfigurációs lehetőség miatt a draw függvény is jóval egyszerűbb, mint a Sierpinszki-háromszög esetén.

```
p.draw = () => {
        this.setConfigurables(p);

        if (this.play) {
                let newRectangles: Rectangle[] = [];

                for (let i = this.iter; i < this.list.length; i
                   ++) {
                        if(this.rainbowMode) {
                                let h = p.map(i, this.iter, this
                                   .list.length, 0, 360);
                                p.fill(h, 255, 255);
                        }
                        this.list[i].draw(p);
                        newRectangles = newRectangles.concat(
                           this.list[i].divide());
                }

                this.rollBackList$.next(this.list);
                this.iter = this.list.length;
                this.list = this.list.concat(newRectangles);
        }
}
```

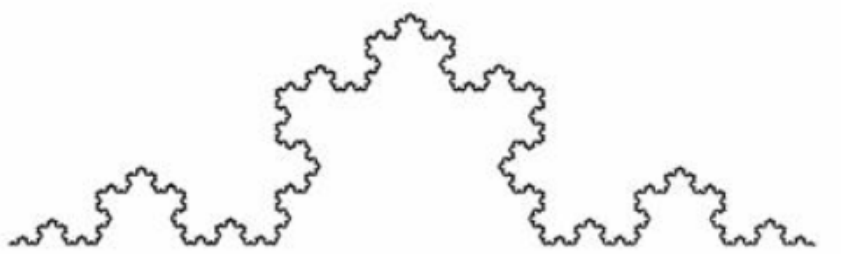
Ez a függvény a négyzetek tömbjén iterál végig, viszont segédváltozók segítségével mindig csak azokon a négyzeteken végez műveleteket, amelyek még nincsenek kirajzolva,

így növelve az algoritmus teljesítményét. Az adott négyzetet kirajzolja, majd kilencedeli a már ismert divide függvény segítségével. A kilencedeléssel kapott négyzeteket ebbe a tömbbe teszi, így a következő iterációkor a draw függvény ezeken a négyzeteken fogja ezeket a műveleteket elvégezni.

0.3. Koch-görbe

A Koch-görbe vagy Koch-hópehely Helge von Koch svéd matematikus által 1904-ben leírt fraktál, mely ilyen minőségében az egyik legelső. A görbét úgy állítjuk elő, hogy veszünk egy szabályos (egyenlő oldalú) háromszöget, minden oldalát megharmadoljuk, és a középső harmadszakaszra újabb szabályos háromszögeket rajzolunk. Majd az így keletkezett háromszögoldalakra újra feltesszük ezt a "kinövést", és ezt a műveletet a végtelenségig folytatjuk. A görbe (bármennyire is egyenes vonalakból áll) egyre jobban egy hópehelyhez fog hasonlítani. (Ezért hópehely-görbének is szokás nevezni.) Természetesen az igazi, teljes hópehely lerajzolása lehetetlen, csupán a hozzá vezető állapotok egymásutánját tudjuk ábrázolni. Amint újabb és újabb „kinövéseket” szerkesztünk a háromszögek oldalaira, a hópehely kerülete egyre nő, azaz a hópehely kerülete valójában végtelen. Mivel maga az alakzat megmarad az első háromszög köré írt körének belsejében, így azt mondhatjuk, hogy a területe viszont véges.

A másik meglepő dolog a Koch-görbe dimenziójának megadása. Ehhez Felix Hausdorff német matematikus dimenzióelméletét vesszük alapul. A szokásos alakzatok körében a Hausdorff-dimenzió megegyezik az ismert értékekkel: az egyenesé 1, a négyzeté 2, a kockáé 3. Ez abból fakad, hogy a Haussdorff-dimenzió a hosszúság és a terület mérésén alapul. Azaz például ha egy négyzet oldalát a háromszorosára növeljük, akkor a terület a kilencszeresére változik, és mivel 9 = 32. így a kétdimenziós négyzet Hausdorff-dimenziója is 2. A hópehelygörbe alapeleme az egyenes szakasz. Ha ezt a háromszorosára nagyítjuk, majd rátesszük a „kinövést”, a szakaszok hosszúsága az eredeti négyszerese lesz. Így a dimenzióra az alábbi összefüggés teljesül: 3d=4. Innen pedig d=log34=lg4lg31,2619. Ez pedig azt jelenti, hogy egy olyan alakzathoz jutottunk, amelynek a dimenziója nem is egész.



3. ábra. Koch-görbe

0.3.1. Implementáció

A Koch-görbét generáló algoritmus alapja a Line segédosztály. A Line segédosztályban tárolódik minden egyenes két végpontja és hossza. Az osztálynak két fontos metódusa van, az expandLeft és expandRight metódusok. Ezek a metódusok az adott egyenesre rajzolnak háromszöget úgy, hogy a háromszög alapja az egyenes, vagy annak egy bizonyos százaléka, akkor az expandRight metódus adja a háromszög bal oldalát, az expandRight pedig a jobb oldalát.

```
expandLeft(p: any, direction: number, lerp: number, angle:
    number): Line[] {
        let len = this.length * lerp;
        let alpha = 180 - 2 * p.degrees(angle);
        let sideLength = len * p.sin(angle) / p.sin(p.radians(
            alpha));

        let lerpAmount = (1 - lerp) / 2;

        let a = p5.Vector.lerp(this.A, this.B, lerpAmount);
        let dir = p5.Vector.sub(this.B, this.A);
        dir.rotate(-direction * angle);
        let offset = p5.Vector.add(a, dir);

        let x = p5.Vector.lerp(a, offset, sideLength / p5.Vector
            .dist(a, offset));
```

```
        let newLines = [];
        newLines.push(new Line(this.A, a));
        newLines.push(new Line(a, x));

        return newLines;
}
```

A metódus paraméterei közé tartozik a direction, ami azt az irányt adja meg, hogy felfelé, vagy lefelé történjen a háromszög oldalának rajzolása, a lerp, ami azt mondja meg, hogy a háromszög alapja hány százaléka az egyenesnek, az angle pedig a háromszög alapja és az oldalai között bezárt szöget adja meg. Ezek a paraméterek a konfigurációs panelben állíthatóak. A függvény kiszámolja a háromszög bal oldalának pontjait, ezekből létrehoz egy új Line objektumot, majd visszaadja egy tömb elemeként.
A Koch-görbe konfigurációs lehetőségei a következők:

- Gyorsaság
- Szín
- Vonalvastagság
- Vonalhosszúság
- Szög
- Háromszög alapjának mérete (%)
- Fixált kezdővonal
- Irány
- Szivárvány mód

A fixált kezdővonal opcióval testreszabhatjuk a kezdő egyenes helyét, és méretét, valamint az egér görgőjével rotálhatjuk, majd kattintással elhelyezhetjük a vászonon. A kezdő egyenes megadása esetén az egér mozgására a változások valós időben láthatóak. Kattintáskor az alábbi függvény fut le:

```
p.handleMousePressed = () => {
        if (this.play && !this.useFixedRoot && this.customRoot
           == null) {
                let center = p.createVector(p.mouseX, p.mouseY);
                let x = p.createVector(p.mouseX - this.length /
                   2, p.mouseY);
                let y = p.createVector(p.mouseX + this.length /
                   2, p.mouseY);

                let xDir = p5.Vector.sub(x, center);
                xDir.rotate(this.rotation);

                let yDir = p5.Vector.sub(y, center);
                yDir.rotate(this.rotation);

                let xOffset = p5.Vector.add(center, xDir);
                let yOffset = p5.Vector.add(center, yDir);

                this.customRoot = new Line(xOffset, yOffset);
                this.root = this.customRoot;
                this.lines = [this.root];
        }
}
```

Egérrel való kattintáskor az egér pozíciója az egyenes közepén van. A függvény ennek a pontnak az x pozíciójához hozzáadja és kivonja belőle az egyenes hosszának felét, az y pozíció mindkét pont esetén változatlan marad, így megkapjuk az egyenes két végpontját. Az így kapott egyenes vízszintes, így ha a felhasználó az egér görgője segítségével rotálta, további számolásokra van szükség. Mindkét végpontból kivonjuk az egyenes középpontját, így két megfelelő irányba mutató vektort kapunk amit a vektorok beépített rotate függvényével könnyen megfelelő pozícióba rotálhatunk. A this.rotation változó tárolja azt az értéket, amennyivel a felhasználó a kezdőegyenest rotálta, ezt az értéket kapja paraméterül ez a függvény. A Koch-görbe draw függvénye hasonló, mint a Sierpinszki-

szőnyegé, így ezt nem részletezném.

0.4. Levy-C Görbe

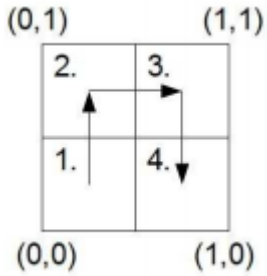
A Lévy C görbét először Enresto Cesàro és Georg Faber írta le és tanulmányozta a differenciálhatóságát, azonban mégis Paul Lévy francia matematikus nevét viseli, aki a fraktál önhasonlóságát definiálta. Ez a fraktál alapjáraton nagyon hasonlít a Koch görbére, egyedüli eltérés, hogy az egyenesekre való háromszögek rajzolásakor nem az egyenes egy bizonyos százalékából lesz a háromszög alapja, hanem a teljes egyenesből. Mivel az algoritmus implementációja és a konfigurációs lehetőségek is nagyon hasonlítanak a Koch görbe alguritmuséhoz, így ezt a fraktált nem részletezem tovább.

0.5. Hilbert görbe

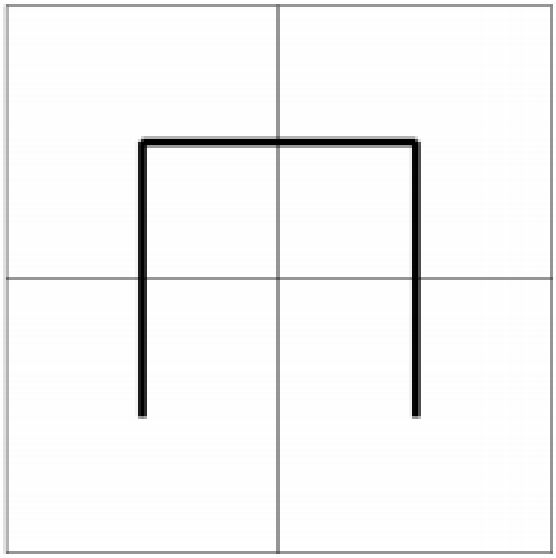
A Hilbert görbe a térkitöltő görbék csoportjába tartozik. Az első térkitöltő görbét ugyan Peano alkotta meg, de Hilbert volt az, aki először érthető geometriai képet tudott mutatni egy általa definiált térkitöltő görbéről. Megadott egy geometriai generáló eljárást, amivel létrehozta ezen görbék egy osztályát. Egy térkitöltő görbét rekurzívan adhatunk meg, bizonyos lépések végtelen sokszori alkalmazásával. A Hilbert görbe első iterációjának megrajzolása úgy történik, hogy veszünk egy egységnégyzetet a következő pontokkal:

- (0, 0) pont a négyzet bal alsó sarka
- (0, 1) pont a négyzet bal felső sarka
- (1, 1) pont a négyzet jobb felső sarka
- (1, 0) pont a négyzet jobb alsó sarka

Ezt az egységnégyzetet képzeletben további 4 szabályos négyzetre bonjuk, és egy törött-vonalat húzunk, úgy, hogy az átmenjen a mind a négy négyzet középpontján az alábbi sorrendet követve: A bal alsó sarokban levő négyzet középpontjából indul, majd felfelé megy a bal felső négyzet középpontjáig, innen jobbra a jobb felső négyzet középpontjáig, majd a jobb alsó négyzet középpontjában áll meg. Ezzel egy fordított U alakzatot kapunk.



4. ábra. Hilbert görbe 1. iterációja

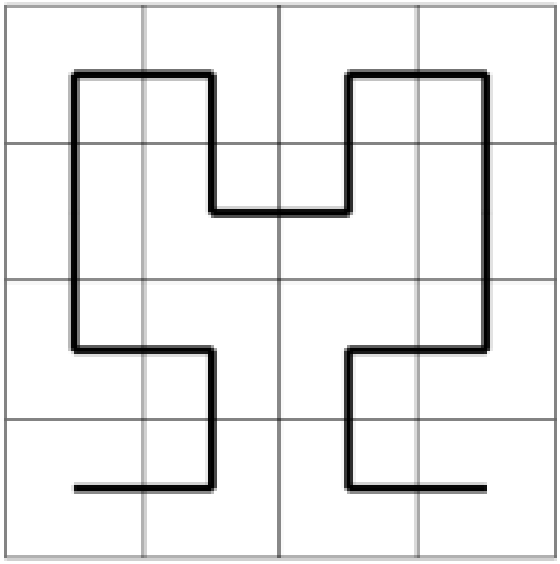


5. ábra. Hilbert görbe 1. iterációja

A Hilbert görbe második iterációja 4 egységnégyzetből áll, mindegyikben megrajzolva az első iteráció fordított U alakzatát. A bal alsó négyzetet 90 fokkal jobbra, a jobb alsó négyzetet 90 fokkal balra forgatjuk, így a két alsó U alakzat egymástól "elfelé" néz. Végül az U alakzat megrajzolásával megegyező sorrendben (bal alsó, bal felső, jobb felső, jobb alsó) összekötjük mind a 4 egységnégyzetben levő alakzatok egymáshoz legközelebbre eső végpontjait.

0.5.1. Implementáció

Az implementációt úgy kezdtem, hogy kiszámoltam hány négyzetből és hány pontból fog állni a Hilbert görbe. Ehhez a következő egyenlőségeket használtam:
$N = 2^I$, $P = N^2$, ahol N a négyzetek száma, I az iteráció, P pedig a pontok száma.



6. ábra. Hilbert görbe 2. iterációja

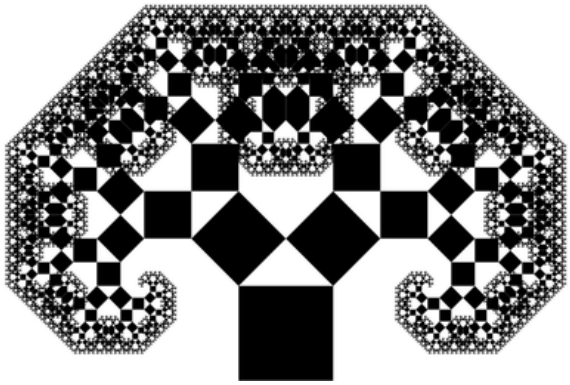
Az iteráció adott, ugyanis a felhasználó konfigurációs lehetőségként meg kell adja, hogy hanyadik iterációig szeretné, hogy az algoritmus fusson. Az algoritmus draw függvénye úgy működik, hogy minden képrfissítéskor eggyel mélyebbre megy a pontok tömbjén, és összeköti az adott pontot a tömbben levő előző ponttal, ezzel egy animációs hatást keltve. Az algoritmus viszonylag kevés konfigurációs lehetőséggel bír, ezek a következők:

- Gyorsaság
- Vonalvastagság
- Iteráció
- Szín
- Szivárvány mód

0.6. Pitagorasz-fa

A Pitagorasz-fa egy négyzetekből álló fraktál, amelyet Albert Bosman fedezett fel. A nevét onnan kapta, hogy minden egymást érintő négyzet hármas egy szabályos háromszöget zár be. Ha a legnagyobb négyzet, L x L méretű (törzs), akkor a teljes Pitagorasz-fa

elfér egy 6L x 4L méretű négyzetben. A hagyományos, egyenlő szárú Pitagorasz-fa a következőképpen konstruálható: az első iterációban létrejön a törzse, amely egy négyzet. A második iterációban a törzsnek a felső élére egy egyenlő szárú derékszögű háromszög rajzolunk úgy, hogy átfogója a négyzet felső éle, valamint a háromszög két befogójából kiágazik az első két ág, amelyek szintén négyzetek. Ezután minden iterációban ez ismétlődik, azaz minden korábbi négyzet felső élére egy egyenlő szárú derékszögű háromszög nő, és azok befogói új négyzetágakat növesztenek. Minden négyzet mérete sqrt(2)/2 értékkel skálázódik le a szülő négyzet méretéhez képest.



7. ábra. Pitagorasz-fa

0.6.1. Implementáció

Az kód átláthatósága érdekében itt is egy segédosztályt írtam először. Ebben az osztályban tárolódnak a négyzetek pontjai illetve mérete. Továbbá, két fontos függvényt tartalmaz, az expandLeft és expandRight függvényeket, amelyek az adott négyzet bal és jobb oldali négyzetágait adják meg.

```
expandRight(p: any, angle: number): Rectangle {
        let A: p5.Vector;
        let B: p5.Vector;
        let C: p5.Vector;
        let D: p5.Vector = this.B;

        let center = p5.Vector.lerp(this.A, this.B, .5);
```

```
        let dir = p5.Vector.sub(this.A, center);
        dir.rotate(angle);
        let offset = p5.Vector.add(center, dir);

        C = p5.Vector.lerp(center, offset, this.size * 0.5 / p5.
            Vector.dist(offset, center));

        A = p5.Vector.sub(D, C);
        A.rotate(-p.PI/2);
        A = p5.Vector.add(C, A);
        A = p5.Vector.lerp(C, A, 1);

        B = p5.Vector.sub(C, D);
        B.rotate(p.PI/2);
        B = p5.Vector.add(D, B);
        B = p5.Vector.lerp(D, B, 1);

        let left = new Rectangle(A, B, C, D);

        return left;
}
```

Az új négyzetek pontjai vektorműveletek sorozatával számolódnak ki. A pontok elnevezése a következő rendszert követi:

- A - bal felső
- B - jobb felső
- C - bal alsó
- D - jobb alsó

A jobb oldali négyzetág D pontja megegyezik a szülő négyzet B pontjával. A C pontot úgy kapom meg, hogy a szülő négyzet B pontjából egy vektort hozok létre, amelyből

kivonom a szülő négyzet felső élének középpontjából létrehozott vektort, így egy megfelelő irányba mutató vektort kapok, amelyet rotálok egy bizonyos értékkel. A rotálási szöget a függvény paraméterként kapja, ugyanis ez az érték testreszabható a konfigurációs panelben. A maradék A és B pontok már könnyen kiszámolhatóak, ezeket úgy kapom meg, hogy két új vektort hozok létre, egyik a C-ből D-be, másik a D-ből C-be mutat, majd ezeket 90 fokkal rotálom a megfelelő irányba. A függvény visszatérési értékként egy új négyzet objektumot hoz létre a kiszámított pontokból. Az expandLeft függvény hasonlóan működik, csupán a rotálási irányok változnak.

Az algoritmus konfigurációs lehetőségei a következők:

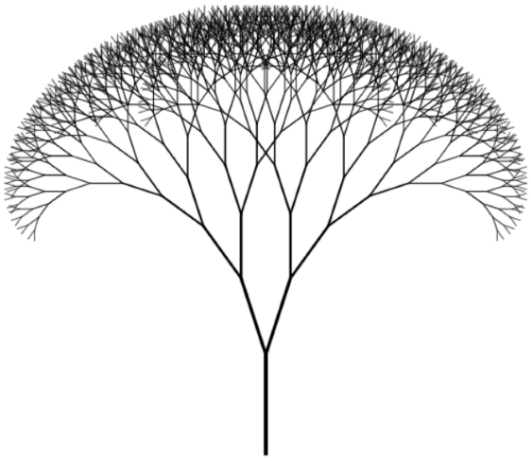
- Gyorsaság
- Fixált szög
- Fixált gyökér
- Oldalhosszúság
- Szín
- Szivárvány mód

Ha a fixált gyökér lehetőséget kikapcsoljuk, akkor tetszőleges helyen helyezhetünk el gyökér négyzetet a vászonon, ezt az egér görgője segítségével rotálhatjuk, illetve az oldalhosszúság opció érékének változtatásával megadhatjuk a méretét. Hasonlóan, a fixált szög lehetőség kikapcsolásával tetszőleges szöggel elhajlíthatjuk a négyzetágakat jobb vagy bal irányba. A szög megadása az egér segítségével történik. Ahhoz, hogy a felhasználóknak érthető legyen a szög megadásának módja, a törzs négyzet felső élének középpontjából egy egyenes indul az egérmutató pozíciója felé. Ennek egyenesnek a hossza az oldalhosszúság felével egyezik meg. Kattintáskor a program kiszámolja az egyenes és a négyzet felső éle között bezárt szöget, és ezt felhasználja a négyzetágak generálásakor.

0.7. Fraktál fa

A fraktál fa egy viszonylag egyszerűen megkonstruálható fraktál, ami a legismertebb fraktálok közé tartozik. Szerkezete nagyon hasonlít a Pitagorasz-fához, annyi különbséggel,

hogy négyzetek helyett egyszerű vonalakból tevődik össze. A fraktál fa első iterációjában csak a törzs van, a másodikban a törzsből két ág nő ki egy bizonyos szöget bezárva a törzzsel. Ez a szög tetszőlegesen állítható a konfigurációs panelben. A további iterációkban ez ismétlődik, tehát a már meglévő ágakból új ágak nőnek ki, ez a végtelenségig ismételhető.



8. ábra. Fraktál fa

0.7.1. Implementáció

A szerkezeti hasonlóságokból eredően az algoritmus implementációja sok helyen hasonlít a Pitagorasz-fáéhoz. Itt is egy segédosztály megírásával kezdtem az implementációt. Ez az osztály tárolja a vonalak A és B végpontjait, valamint hosszát, ezen kívül még tartalmaz egy branch nevezetű függvényt, amely egy adott vonalra meghívva kiszámolja annak az ágait.

```
branch(p: any, angleLeft: number, angleRight: number,
    lerpPercentage: number): Line[] {
        let lines: Line[] = [];
        let lerpAmount = this.length * lerpPercentage / this.
            length;

        let dir = p5.Vector.sub(this.A, this.B);
        let xRotated = dir.rotate(angleLeft);
```

```
        let xOffset = p5.Vector.add(this.A, xRotated);
        let yRotated = dir.rotate(-angleLeft - angleRight);
        let yOffset = p5.Vector.add(this.A, yRotated);
        let x = p5.Vector.lerp(this.A, xOffset, lerpAmount);
        let y = p5.Vector.lerp(this.A, yOffset, lerpAmount);

        lines.push(new Line(x, this.A));
        lines.push(new Line(y, this.A));

        return lines;
}
```

Ennek a függvénynek 3 fontos paramétere van, az angleLeft, ami megadja a bal oldali ág és a szülő ág által bezárt szöget, az angleRight, ami a jobb oldali ág és a szülő ág által bezárt szöget adja meg, valamint a lerpPercentage, ami azt adja meg, hogy az ágak hossza hány százaléka a szülő ág hosszának. Mivel az egyes vonalak A és B végpontjait vektor típusokként tárolja a segédosztály, ezért az új ágak végpontjai vektorműveletek segítségével könnyen kiszámolhatóak. Az A végpontból kivonva a B végpontot, egy, a vonal irányával megegyező irányú vektort kapok. Ezt, a paraméterben kapott értékekkel jobbra és balra forgatva, megkapom az új ágak A végpontjait. Az új ágak B végpontjai a szülő ág A végpontjával egyeznek meg értelemszerűen. A visszatérési értéke egy tömb, amely az új ágakat tartalmazza.

Az algoritmus konfigurációs lehetőségei a következők:

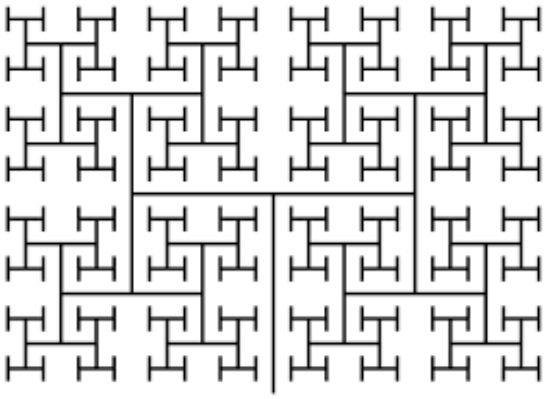
- Gyorsaság
- Vonalvastagság
- Vonalhosszúság
- Ágak száma
- Elforgatási szög
- Ág mérete (%)
- Véletlenszerű szög

- Fixált kezdővonal
- Szín
- Szivárvány mód

Az ágak száma opcióval megadható, hogy az egyes ágakból 2 vagy 3 új ág keletkezzen, 3 ág esetén a középső ág iránya megegyezik a szülő ág irányával. A fixált kezdővonal opció kikapcsolásával a fa törzsét adhatjuk meg tetszőlegesen, hasonlóan, mint az előző algoritmusoknál.

0.8. H-fa

A H-fa struktúrája hasonló a Fraktál-fáéhoz. Ez a fraktál merőleges vonalak közvetlen egymás mellé helyezésével konstruálható meg. Minden vonal mindkét végpontján egy rá merőleges vonal halad át, amelynek hossza mindig sqrt(2)-vel kisebb az előző vonal hosszától. Ez egy alapértelmezett érték, ami a konfigurációs panelben tetszőlegesen állítható. A fraktál a nevét onnan kapta, hogy a benne ismétlődő minta a H betűre emlékeztet. A H-fa Hausdorff dimenziója 2.



9. ábra. H-fa

0.8.1. Implementáció

A H-fa egy viszonlyag egyszerűnek mondható fraktál, ezért az implementációja is egyszerű. A függvények, amelyek az ágak pontjainak kiszámolását végzik itt is egy segéd-

osztályban vannak megírva, ezek az expandLeft és expandRight függvények.

```
expandLeft(p: any, lerp: number) {
        let dir = p5.Vector.sub(this.A, this.B);
        dir.rotate(p.PI / 2);
        let xOffset = p5.Vector.add(this.A, dir)
        let x = p5.Vector.lerp(this.A, xOffset, lerp / 2);

        dir.rotate(-p.PI);
        let yOffset = p5.Vector.add(this.A, dir);
        let y = p5.Vector.lerp(this.A, yOffset, lerp / 2);

        return new Line(x, y);
}
```

A segédosztályban a vonalak végpontjai vektorokként vannak tárolva. A végpontokat egymásból kivonva egy vektort kapunk, amelynek iránya megegyezik a vonal irányával, majd ezt jobbra és balra forgatva megkapjuk az bal oldali ág két végpontját. A két új végpontot nem teljesen kötjük össze a szülő vonal bal oldali végpontjával, az új végpontok irányába csak az új ág hosszának a felével megegyező mértékig megyünk, ezzel megkapva az ág megfelelő hosszát. A jobb oldali ág kiszámítása hasonlóan történik.

Az algoritmushoz tartozó konfigurációs lehetőségek a következők:

- Gyorsaság
- Vonalvastagság
- Vonalhosszúság
- Ág hosszúság (%)
- Fixált kezdővonal
- Szín
- Szivárvány mód

Nyilatkozat

Alulírott ..... szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Intézet ..... Tanszékén készítettem, ... diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem, hogy szakdolgozatomat / diplomamunkámat a Szegedi Tudományegyetem Informatikai Intézet könyvtárában, a helyben olvasható könyvek között helyezik el.

Szeged, 2020. május 6. .....

aláírás

Köszönetnyilvánítás