; revised .

Patrons repetitius. Desenvolupament i anàlisi d'una aplicació per a la representació de fractals i geometria auto-similar.

Dylan Canning Garcia[1], Antonio Marí González[2], Juan Marí González[3], Antoni Jaume Lemesev [4]

[1]Estudiant Enginyeria Informàtica UIB, 49608104W
[2]Estudiant Enginyeria Informàtica UIB, 46390336A
[3]Estudiant Enginyeria Informàtica UIB, 46390338M
[4]Estudiant Enginyeria Informàtica UIB, 49608104W

Alumne d'entrega: Dylan Canning (email: dcg750@id.uib.eu).

ABSTRACT

Aquest document descriu el disseny i la implementació d'un programa de càlcul i visualització de geometria auto-similar i fractals per a la assignatura d'Algorismes Avançats. Concretament, es detallen les tècniques emprades com són la definició de classes, l'enfocament de Dividir i Vèncer utilitzat, la programació concurrent i la distribució de funcions al model arquitectònic resultat, basat en el Model-Vista-Controlador d'un sistema distribuït. Tot això seguit d'un enumerat d'optimitzacions, unes breus pinzellades dels algorismes implementats, decisions preses al llarg de la implementació i càlculs de cost computacionals dels algorismes implementats.

Finalitzarem el document amb un anàlisi detallat dels resultats obtinguts, juntament amb les gràfiques adients per a la visualització de la complexitat dels diferents fractals i figures geomètriques, les dificultats que suposen aquest tipus d'algorismes en funcions de visualització interactiva i, finalment, possibles millores del nostre codi i implementació.

I. Introducció

AQUESTA pràctica s'ha desenvolupat mitjançant una metodologia estructurada basada en l'arquitectura Model-Vista-Controlador (MVC) vista a classe. La modularitat que presenta aquesta estructura ens ha permès distribuir la feina en paquets d'esforç disjunts, la qual cosa ha agilitzat molt el desenvolupament de la pràctica. Fent ús de les incidències apreses a la primera pràctica, hem reestructurat el funcionament intern d'aquestes per a reduir la dependència entre els paquets de vista, controladora i model, definint de primera mà les interaccions/esdeveniments entre les diferents parts del programa (interfícies de notificació).

La distribució de tasques entre el nostre equip de quatre membres s'ha realitzat seguint els components propis del patró MVC [10]: un membre s'ha encarregat de la implementació de la interfície gràfica (Vista), dos membres han desenvolupat la part de dades i algorismes (Model), i un altre ha implementat la part controladora que coordina les interaccions entre els components anteriors. A diferència de la primera pràctica, la definició d'una interfície d'*algorisme* ens ha donat la facilitat de poder implementar algorismes de manera modular, la qual cosa ha suposat una llibertat i facilitat a l'hora d'afegir models nous de manera senzilla.

Per facilitar la col·laboració i el seguiment del projecte, hem fet ús de l'entorn de treball creat a la primera pràctica (repositori GitHub) per a dur un control de versions, permetent documentar i compartir els avenços de manera sistemàtica. Addicionalment, en retrospectiva amb la primera pràctica, hem fet ús d'una eina anomenada *Obsidian* que permet dibuixar diagrames i crear relacions entre idees i arxius. Igualment, amb la mateixa justificació que la primera pràctica, hem seguit fent feina amb l'ús de l'IDE IntelliJ per mantenir la compatibilitat dels formats de projecte.

Un cop establertes aquestes bases metodològiques, hem procedit a definir els aspectes tècnics específics de cada secció (Algorismes, Estructures de Dades, Processos, etc.) necessaris per a la implementació. L'objectiu central del nostre disseny ha estat aplicar i maximitzar els coneixements

adquirits a l'assignatura, especialment en relació amb les estratègies de Dividir i vèncer, Concurrència i Disseny Modular, que constitueixen els pilar per a la facilitat d'implementació de una gran quantitat d'algorismes de patrons repetitius /tessel·lats.

II. Metodologia de feina

Per aquest projecte hem seguint amb una metodologia estructurada i col·laborativa, que ens ha permès integrar diferents eines i tècniques per aconseguir un producte final coherent sense haver de fer intensius presencials amb tots els companys. En aquest apartat detallarem els aspectes relacionats amb l'entorn de programació i el sistema de control de versions que hem emprat al llarg del desenvolupament, com els canvis en respectiva a la primera pràctica.

A. Entorn de Programació

El desenvolupament de l'aplicació l'hem dut a terme utilitzant l'IDE *IntelliJ IDEA*, que ens ha proporcionat un entorn robust i modern per a programar en Java, bastant més amigable que *NetBeans*. S'ha treballat amb la versió *Java 23*, aprofitant les seves millores i estabilitat per garantir una execució eficient dels càlculs i processos implementats, com també la facilitat de maneig amb objectes gràfics.

Per a la creació de la interfície gràfica s'han utilitzat les llibreries **Swing** per als components de la interfície gràfica i la seva subclasse **AWT** per a la representació de punts dels patrons calculats. Com hem dit Swing ha estat fonamental per al disseny i la gestió dels components interactius de la GUI, mentre que AWT ha permès generar les representacions gràfiques de manera dinàmica i de bona resolució per a la visualització correcte per part de l'usuari.

B. Control de Versions

En comparació a la primera pràctica hem estès la estructura del repositori, per poder tenir tot centralitzat, pràctica 1 i pràctica 2. Tot i així hem seguit utilitzant el nostre sistema de control de versions basat en **Git**, amb el repositori centralitzat allotjat a **GitHub**. Aquesta estratègia ha facilitat:

- Mantenir un historial detallat de totes les modificacions realitzades al codi font.
- Coordinar el treball col·laboratiu entre els membres de l'equip, permetent la integració de diferents funcionalitats de manera ordenada.
- Assegurar una revisió contínua del codi i una resolució ràpida dels conflictes que sorgeixin durant el desenvolupament.

La utilització de GitHub segueix sent clau per a la gestió de les versions, degut a que garanteix que el projecte romangui coherent i actualitzat en tot moment, i afavorint la millora contínua i la transparència en el procés de desenvolupament. A més de l'esmentat, la IDE emprada ens ha permès dur a terme aquest control de versions de manera senzilla pels integrants que no han emprat abans la feina, ja que posseeix eines de gestió de versions sense haver d'utilitzar la terminal.

III. Fonaments Teòrics

En aquesta secció tractarem els diversos elements teòrics que hem emprat per a la nostra pràctica.

A. Model Vista Controlador

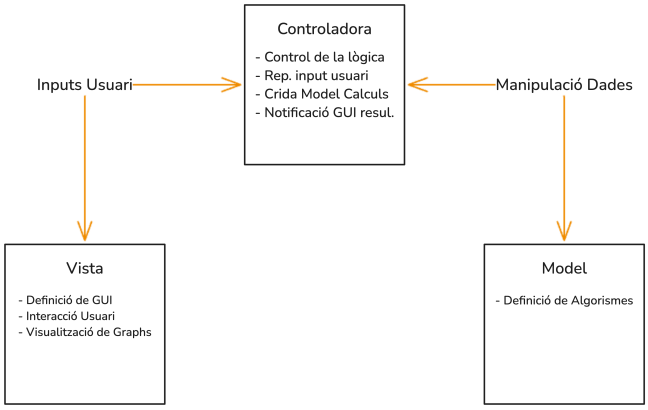

Figura 1. Model Vista Controladora

El Model-Vista-Controlador (MVC) segueix sent el mateix que l'implementat a la primera pràctica, és a dir, és un patró de disseny de software emprat normalment per interfícies d'usuari que té com a objectiu separar la lògica del programa en tres elements:

- Model: La representació interna de la informació.
- Vista: La interfície gràfica que fa de portal entre l'usuari i la lògica interna del programa, es a dir, mostra informació i recull informació de l'usuari.
- Controladora: Fa de connexió entre el Model i la Vista.

En el cas del nostre projecte, hem definit els tres nuclis tal com diu la definició, la part de la interfície gràfica és la encarregada de dur a terme tota la interacció amb l'usuari; és interessant fer una pausa i analitzar la importància de la interacció d'usuari i interfície gràfica, podem considerar el programa com una capsa negre amb la qual l'usuari a d'interactuar a través de la *Grafic User Interface*, per tant, recau en la GUI l'amigabilitat i facilitat d'ús. En aquesta pràctica hem intentat fer un poc més d'èmfasi en la interfície, augmentant l'enteniment de les funcions del programa, minimitzant els inputs erronis per part de l'usuari, això es pot apreciar en els components de *Swing* en la interfície acompanyats per etiquetes i en el cas de l'input de la dimensió de n amb un *slider*, que permet mantenir control de les dimensions: Múltiples de 2^n i màxim / mínim de dimensió.

Fixant l'atenció en l'estructura interna del programa, tenim els tres models molt diferenciats. La part de la vista que és l'encarregada de dur a terme tota la interacció amb l'usuari, mostra els patrons de tessel·lat repetitiu i els fractals

implementats al programa, permet a l'usuari introduir en el cas del Tromino la casella fixa de manera interactiva, fent ús del ratolí o introduint les coordenades en els camps acordes; també deixa elegir l'algorisme a calcular com també la dimensió del taulell per als X-mino o la profunditat en el cas de fractals. Aquestes variables les explicarem amb més detall més endavant. La controladora fa de funció de servidor, processa les peticions per part de la Vista/Usuari i acciona els "trigers"apropiats per dur a terme les accions especificades per l'usuari. Finalment, el Model engloba tot allò relacionat a estructures de dades i els mètodes de manipulació d'informació, en el cas d'aquesta segona pràctica són els *Algorismes de Fractals* (Sierpinski, Koch, Hilbert, etc.) i *Geometria Repetitiva* (Tromino i Dominó), aquestes classes encapsulen tots els procediments i funcions per dur a terme els càlculs/modificacions per a calcular els fractals.

Anam a especificar amb més detall cadascuna d'aquestes parts:

1) Interacció Vista $\Longleftrightarrow$ Controladora

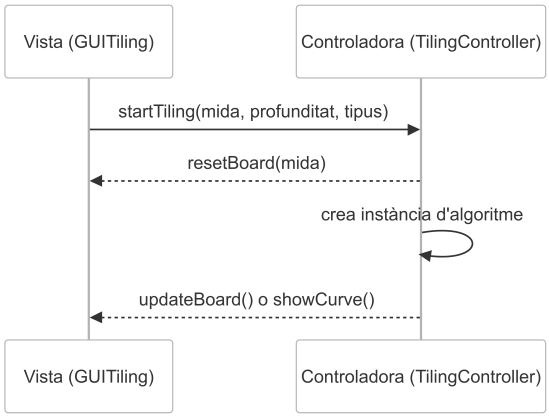

Figura 2. **Interacció Vista amb Controladora**

En aquesta figura es mostra la interacció seqüencial entre la Vista i la Controladora. La classe `GUITiling` (Vista) és la responsable de recopilar les entrades de l'usuari, com la selecció de l'algorisme, la dimensió del tauler, i els paràmetres específics (com les coordenades de la peça absent en el cas del tromino). Quan l'usuari prem el botó "Start", es crida el mètode `startTiling(...)` del `TilingController` (Controladora). Aquest mètode comprova primer si hi ha un càlcul en curs (controlat mitjançant un flag `isRunning`) i, si no, inicialitza els executors corresponents (un `ScheduledExecutorService` per a l'animació o un `ForkJoinPool` per a càlculs paral·lels). A més, la Controladora crida a la vista per a reinicialitzar el tauler mitjançant el mètode `resetBoard(...)` abans d'iniciar el càlcul. D'aquesta manera, la vista envia la petició de càlcul i, posteriorment, rep actualitzacions mitjançant els mètodes `updateBoard(...)` o `showCurve(...)` que actualitzen la representació gràfica.

2) Interacció Controladora $\Longleftrightarrow$ Model

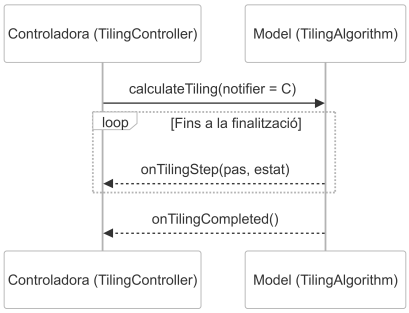

Figura 3. **Interacció Controladora amb Model**

En aquesta figura s'il·lustra la comunicació entre la Controladora i el Model. La interfície `TilingAlgorithm` defineix el contracte per als algorismes (com `TrominoTiling`, `HilbertCurve`, `KochCurve` i `SierpinskiTriangle`), i el seu mètode `calculateTiling(...)` és invocat per la Controladora. A cada pas, l'algorisme invoca mètodes de la interfície `TilingNotificar` (implementada per la Controladora) per comunicar el progrés: `onTilingStarted(...)` per iniciar, `onTilingStep(...)` per cada pas recursiu i `onTilingCompleted()` en finalitzar. Aquest mecanisme de notificació permet que el model estigui desacoblat de la vista; és la Controladora qui intercedeix per traduir aquests esdeveniments en actualitzacions de la GUI.

B. Esdeveniments

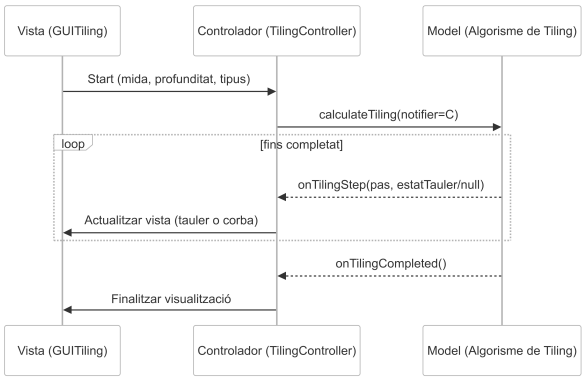

Figura 4. **Esdeveniments del programa**

Aquest diagrama representa el patró d'esdeveniments del nostre programa, on es destaquen els missatges de comunicació entre els diferents components. El model, durant l'execució d'un algorisme, emet notificacions (inici, pas, completió o error) a través de la interfície `TilingNotificar`. La Controladora rep aquests esdeveniments i, d'acord amb el seu estat intern, actualitza la Vista. Així, es garanteix que la comunicació sigui asíncrona i desacoblada, evitant bloquejos en el fil principal de la GUI i facilitant la gestió de la concurrència.

El patró d'esdeveniments simplifica la coordinació entre components:

- **Usuari:** Interactua amb la GUI per especificar paràmetres de càlcul i inicia l'operació (esdeveniment "run").
- **GUIOperacionsMatrius:** La interfície gràfica recull les accions de l'usuari i envia notificacions a la Controladora (p.ex., mitjançant el botó "Start").
- **TilingController:** La Controladora implementa la interfície `TilingNotificar` i actua com a receptor dels esdeveniments enviats pel Model. També tradueix aquestes notificacions en actualitzacions a la GUI.
- **Model:** Cada algorisme implementa `TilingAlgorithm` i, durant el seu procés recursiu, invoca els mètodes `onTilingStarted`, `onTilingStep` i `onTilingCompleted` per notificar el seu estat.

1) Per què implementació basada en esdeveniments?

L'aplicació ha estat dissenyada emprant un patró d'esdeveniments que optimitza la responsivitat de la interfície i millora la modularitat del codi. Aquest enfocament permet que cada component actui de forma independent i reaccionar a esdeveniments específics sense bloquejar el flux principal.

Per exemple, la interfície `TilingNotificar` actua com a contracte per a la comunicació d'esdeveniments clau:

```
1 public interface TilingNotificar {
2     void onTilingStarted(int boardSize, int
          maxDepth);
3     void onTilingStep(int step, int[][] boardState)
          ;
4     void onTilingCompleted();
5     void onTilingError(String errorMessage);
6 }
```

Aquest contracte permet al Controlador notificar la GUI sobre l'inici, la finalització o els errors en els càlculs. Així, quan s'inicia un càlcul, la Controladora crida `onTilingStarted`; durant l'execució, el Model crida `onTilingStep(...)` per actualitzar el progrés; i en finalitzar, s'informa mitjançant `onTilingCompleted()`. Aquesta implementació asíncrona assegura:

- **Responsivitat:** El càlcul es fa en fils separats, evitant bloquejar la GUI.
- **Modularitat:** La Vista, la Controladora i el Model estan desacoblats, cosa que facilita el manteniment i la futura expansió del codi.

IV. Gestió de la Concurrència i Comparació d'Implementacions Seqüencials i Paral·leles [15] [17]

Aquest apartat presenta una anàlisi meticulosa del funcionament intern del sistema implementat, posant especial èmfasi en el patró d'esdeveniments, la comunicació entre Vista, Controladora i Model, i la gestió del multi-threading i paral·lelisme emprat per tots els algorismes (TrominoTiling, HilbertCurve, KochCurve, SierpinskiTriangle i SquareModel).

A. Gestió de la Concurrència i Sincronització

S'ha evitat introduir locks o monitors que poguessin causar interbloqueigs. Cap dels mètodes sincronitzats de baix nivell (com `synchronized` o `wait()/notify()`) s'ha considerat necessari gràcies a l'ús d'eines de mes alt nivell (atòmics, executors i futures). L'absència de seccions critiques amb bloqueig mutu es tradueix en la inexistència de deadlocks clàssics. Per exemple, el model no espera cap confirmació de la GUI (envia notificacions i continua), i la GUI nomes accedeix a l'estat del model a traves de dades immutables passades en cada esdeveniment (com una copia de la matriu o de la llista de punts). Això elimina qualsevol dependència cíclica entre fils.

A mes, les subtasques del `ForkJoinPool` segueixen un esquema de *divideix-i-vence* sense compartir memòria mes enllà de dades de nomes lectura (com els punts inicials en l'algoritme de Sierpinski), o estructures sincronitzades ja esmentades. El framework Fork/Join incorpora estratègies perquè els fils inactius "robin" feina d'altres fils ocupats (work stealing) [1], la qual cosa minimitza el risc d'inanició de cap fil i contribueix a equilibrar la carega sense intervenció manual.

Finalment, cada nou calcul reinicia l'estat (es crea un nou objecte de model i es reinicia el panell de la GUI) i, gracies al disseny basat en esdeveniments, no hi ha fuites de memòria relacionades amb listeners: el `TilingController` és essencialment l'únic observador i el model finalitza sense mantenir referencies a observadors innecessaris (després de `onTilingCompleted` no es guarden referencies). En resum, la gestió de múltiples fils en aquest sistema es du a terme de manera segura i eficient: s'usa executors especialitzats per a animació i càlcul paral·lel, amb sincronització lock-free mitjançant variables atòmiques, separació estricta de responsabilitats entre els fils de càlcul i la GUI, i una devolució ordenada dels recursos un cop la feina ha conclòs, evitant col·lisions de concurrència i consum innecessari de recursos.

B. Comparació entre Implementacions Seqüencials i Paral·leles

El codi proporciona, en essència, dues maneres d'executar cada algoritme: seqüencial (amb animació pas a pas en un sol fil) i paral·lela (càlcul en múltiples fils). Aquesta comparació qualitativa ens permet entendre els beneficis i els costos de cada enfoc:

- **Mode seqüencial (animat):** L'algoritme s'executa pas a pas en un únic fil, intercalant càlcul i pauses per actualitzar la GUI. Això te l'avantatge de simplicitat i predictibilitat, ja que nomes un fil modifica les estructures de dades del model, eliminant la necessitat de sincronit-

zació complexa. Per exemple, en `TrominoTiling` en mode seqüencial es crida recursivament `.compute()` en lloc de `invokeAll()` [2]. Així, l'ordre temporal dels passos es manté, fet important per a la visualització. No obstant, aquest mode utilitza un sol nucli de la CPU, cosa que pot resultar en temps d'execució mes llargs per a problemes de gran mida (per exemple, dibuixar una corba de Hilbert d'ordre alt).

- **Mode paral·lela (no animat):** Es fa ús de concurrència per dividir la carrega de càlcul entre múltiples fils. Per exemple, `KochCurve` subdivideix el segment inicial en quatre parts de manera paral·lela; amb quatre nuclis, cada segment es pot calcular simultàniament, aconseguint un speed-up proper a $4\times$. De manera similar, `TrominoTiling` pot processar quatre quadrants del tauler simultàniament un cop col·locada la peça central. No obstant, el paral·lelisme comporta una sobrecàrrega en crear i gestionar fils, dividir tasques i combinar resultats. Segons la Llei d'Amdahl [16], el guany real en temps està limitat per la fracció de codi que roman seqüencial i pel cost addicional de coordinació [2]. Aquesta sobrecàrrega es fa evident en tasques amb gran granularitat, on per exemple, si la profunditat de `KochCurve` es baixa, el cost de crear subtasques pot superar el benefici de paral·lelitzar.
- **Complexitat del codi:** Les implementacions paral·leles requereixen estructures addicionals (com classes `RecursiveTask`, gestio de `join()`, variables atòmiques) i per tant son mes difícils de depurar i mantenir en comparació amb les versions seqüencials. No obstant, l'ús de Fork/Join facilita la gestió de fils i la reutilització dels mateixos.

Per a clarificar aquesta comparació, el Llistat 2 mostra un pseudocodi resumit de la funció `TrominoTiling.calculateTiling()` que il·lustra el contrast entre els dos modes:

```
void calculateTiling(TilingNotificar notifier, int maxDepth) {
    notifier.onTilingStarted(boardSize, maxDepth);
    boolean animate = (notifier instanceof TilingController)
                      && ((TilingController) notifier).isAnimationEnabled();
    if (animate) {
        // Mode animacio: usar executor planificat (un fil)
        ScheduledExecutorService scheduler = ((TilingController) notifier).getAnimationExecutor();
        AtomicInteger activeTasks = new AtomicInteger(0);
        activeTasks.incrementAndGet();
        // Inicia tasca recursiva inicial
        scheduler.execute(() ->
            scheduleTiling(0, 0, boardSize, missingRow, missingCol, notifier, activeTasks)
        );
        // La funcio scheduleTiling planificara sub-tasques amb retard
    } else {
        // Mode no animat: usar ForkJoinPool per parallelaitzar
        ForkJoinPool pool = ((TilingController) notifier).getComputeExecutor() instanceof ForkJoinPool
            ? (ForkJoinPool)((TilingController) notifier).getComputeExecutor()
            : new ForkJoinPool();
        pool.invoke(new TrominoTask(0, 0, boardSize, missingRow, missingCol, notifier));
        notifier.onTilingCompleted();
    }
}
```

En resum, el mode seqüencial és més senzill i garanteix l'ordre de les actualitzacions (important per a la visualització), mentre que el mode paral·lel aconsegueix un millor rendiment en màquines multi-nucli, a costa d'una major complexitat en la gestió dels fils i la sincronització

V. Fractals i Geometria Repetitiva [19] [20]

Abans de continuar amb el càlcul del cost computacional, creim adient una breu explicació de que són els algorismes implementats, i que els fa únics.

Els fractals i els patrons de geometria repetitiva es caracteritzen per la seva auto-similitud i per una definició recursiva que es repeteix a diferents escales. En el context d'aquest projecte, s'han implementat diversos algoritmes d'aquest tipus – incloent **Tromino Tiling**, la **Corba de Hilbert**, la **Corba de Koch** i el **Triangle de Sierpinski** – amb l'objectiu d'estudiar-ne el comportament computacional. Cadascun d'aquests patrons exemplifica propietats clau: un intens ús de la recursió, un creixement combinatori del nombre d'operacions amb el nivell de profunditat i una estructura altament modular gràcies a la seva auto-similitud [21].

A. Tromino Tiling

El *Tromino Tiling* es basa en un algoritme de divideix-i-vence que resol el problema de cobrir un tauler quadrat de dimensions $2^n \times 2^n$ (amb n com a profunditat recursiva) amb una única casella buida, fent servir peces tromino en forma de L. El procediment es resumeix en els passos següents:

- Dividir el tauler en quatre quadrants iguals.
- Identificar el quadrant que conté la casella ja ocupada.
- Col·locar una peça tromino al centre del tauler que cobreixi una casella de cadascun dels altres tres quadrants.
- Resoldre recursivament el problema en cadascun dels quadrants, que ara tenen exactament una casella buida.

Així, cada subtauler és una versió reduïda del problema original, i el cost total del algoritme creix en funció de l'àrea del tauler, resultant en una complexitat de $O(N^2)$ per a un tauler amb N cel·les. A més, aquest algoritme permet una implementació modular que facilita la paral·lelització en el moment de processar els quatre quadrants simultàniament.

B. Corba de Hilbert

La *Corba de Hilbert* es defineix com un camí continu que visita totes les cel·les d'una quadrícula $2^d \times 2^d$, on d representa el nivell de profunditat. L'algoritme recursiu construeix la corba de manera inductiva:

- Es parteix d'un cas base simple (una corba d'ordre $d = 1$ que forma una Ü").
- Cada iteració genera quatre subcorbes de nivell $d - 1$, orientades de manera que la corba final ompli el quadrat.

El nombre de segments dibuixats es multiplica per 4 en cada iteració, resultant en una complexitat de $O(4^d)$, la qual és equivalent a $O(N^2)$ si $N = 2^d$ és la dimensio lineal de la quadrícula. Aquest enfoc recursiu fa que el càlcul de la corba esdevingui intensiu en els nivells superiors, tot i que la implementació modular i la possibilitat de paral·lelitzar la computació de les subcorbes faciliten la gestió de la complexitat.

C. Corba de Koch

La *Corba de Koch*, també coneguda com a "floc de neu de Koch", parteix d'un segment de línia recta i, en cada iteració recursiva, substitueix aquest segment per quatre segments més petits que formen un patró dentat. Concretament:

- Cada segment es divideix en tres parts iguals.
- Es calcula un punt intermedi que forma un triangle equilàter amb el terç central.
- Es generen quatre segments nous: del principi al primer punt, del primer punt al punt intermedi, del punt intermedi al segon punt i del segon punt al final.

Així, amb n iteracions es generen 4^n segments, i la complexitat temporal del procés és de $O(4^n)$. L'algoritme aprofita la recursió per calcular cada subsegment i pot ser paral·lelitzat mitjançant el model Fork/Join per reduir el temps d'execució, tot i que el cost total en operacions es manté.

D. Triangle de Sierpinski

El *Triangle de Sierpinski* es construeix a partir d'un triangle equilàter inicial. L'algoritme segueix un esquema recursiu ternari:

- Dividir el triangle en quatre subtriangles congruents i eliminar el triangle central.
- Aplicar recursivament el mateix procediment als tres triangles resultants.

Cada iteracio genera 3 vegades més triangles, de manera que el nombre total de subtriangles creix com 3^n per n iteracions, resultant en una complexitat de $O(3^n)$ en termes de passos o segments generats. La implementacio modular permet la paral·lelitzacio dels tres subtriangles en cada pas, millorant el rendiment sense alterar la complexitat total.

E. Eficiència, Modularitat i Paral·lelisme Integrat

Tots aquests algorismes comparteixen una estructura recursiva i auto-similar que afavoreix una implementacio modular. Mitjançant la interfície unificada `TilingAlgorithm`, cada patró (Tromino, Corbes fractals i Triangle) s'ha implementat com a mòdul independent. Aquest disseny facilita la integracio de noves funcionalitats i la paral·lelitzacio, ja que cada subproblema pot ser processat de forma concurrent sense dependre directament dels altres.

A mes, la possibilitat de visualitzar el procediment pas a pas o de mostrar directament el resultat final (mitjançant un mode animat o no animat) reforça l'aspecte interactiu del projecte. El sistema, a mes, gestiona de manera eficient la comunicacio entre fils i la GUI, assegurant que l'usuari pugui apreciar la evolucio de la figura sense bloquejar la interfície.

VI. Càlcul de complexitat asimptòtica dels algorismes

A continuació s'analitza la complexitat temporal de cada algoritme implementat, acompanyant-ho de pseudocodi simplificat per descriure'n el funcionament. Es considera la complexitat en el pitjor cas i en termes de la mida de l'entrada (per exemple, la dimensió del tauler o el nivell de profunditat fractal especificat).

A. TrominoTiling – Tiling de trominos (algorisme de divide and conquer)

Aquest algorisme resol el problema de cobrir un tauler $2^k \times 2^k$ amb una sola casella buida, utilitzant peces tromino en forma de L. Es pot descriure breument així:

Listing 2. Pseudocodi per a `trominoTiling`

```
Algorisme trominoTiling(tauler[0..n-1][0..n-1],
    r_missing, c_missing):
    si n == 1:
        retornar  // cas base: quadr cula 1x1 ja "
            coberta"
    mida_sub = n/2
    // Determinem en quin quadrant cau la pe a
        absent
    quad_missing = quadrant(r_missing, c_missing,
        mida_sub)
    // Coloca un tromino al centre cobrint les
        quatre subquadrants,
    // exceptuant el quadrant amb la pe a absent
```

```
    posaTrominoCentral(centre_quadrants,
        quad_missing)
    // Calcula coordenades de les peces absents
        induides en els altres 3 subquadrants
    per a cada quadrant i en (NW, NE, SW, SE)$:
        si i es el quadrant amb la pe a absent:
            (r_i, c_i) get (r\_missing, c\_missing)
                $  // peca absent original
        sin :
            (r_i, c_i) get coordenades del centre
                del quadrant i
        // Crida recursiva per cobrir el subtauler
            i-esim
        trominoTiling(subTauler_i, r_i, c_i)
```

En cada crida, l'algoritme divideix el tauler actual en 4 subtaulers de mida $n/2$, col·loca un únic tromino (cost constant) i després resol recursivament els quatre subproblemes. El temps d'execució $T(n)$ satisfà aproximadament la recurrència

$$T(n) = 4\,T\left(\frac{n}{2}\right) + O(1),$$

on $O(1)$ representa el temps per col·locar el tromino central. Resolent aquesta recurrència (per iteració o aplicant el Master Theorem), s'obté que $T(n) = O(n^2)$. Intuïtivament, el nombre de peces tromino col·locades és $(n^2-1)/3$, que creix proporcionalment a l'àrea del tauler (és a dir, $\Theta(n^2)$). En termes de potències de dos, si $n = 2^k$, el nombre de passos és $\Theta(4^k)$, però sempre equival a una complexitat quadràtica respecte a la longitud n. Cal notar que aquesta complexitat correspon al càlcul algorísmic pur; en mode animació, l'algoritme introduirà pauses de retard constants entre passos (afectant només la constant de temps per pas). Finalment, pel que fa a l'ús de memòria, el tromino tiling empra una pila recursiva de profunditat $k = \log_2 n$ i un tauler de dimensions $n \times n$, per tant l'espai és $O(n^2)$ (dominat pel tauler de sortida).

B. HilbertCurve – Corba de Hilbert (corba fractal space-filling)

La corba de Hilbert de nivell d és un camí continu que recorre totes les cel·les d'una quadrícula $2^d \times 2^d$, visitant-les una sola vegada. L'algoritme recursiu per generar els punts de la corba de Hilbert es pot expressar de la manera següent:

Listing 3. Pseudocodi per a `hilbertCurve`

```
Algorisme hilbertCurve(nivell, angle):
    si nivell == 0:
        retornar
    girar(angle)
    hilbertCurve(nivell-1, -angle)
    avanca_i_dibuixa()   // traca un segment
    girar(-angle)
    hilbertCurve(nivell-1, angle)
    avanca_i_dibuixa()
    hilbertCurve(nivell-1, angle)
    girar(-angle)
    avanca_i_dibuixa()
    hilbertCurve(nivell-1, -angle)
    girar(angle)
```

En aquest pseudocodi, l'angle inicial és de $90°$ (gir a l'esquerra) i l'algoritme efectua quatre crides recursives per a cada nivell, intercalant tres moviments de dibuix (avanços) amb canvis de direcció. La idea és que la corba de Hilbert de nivell d es construeix combinant quatre corbes de nivell $d-1$, orientades de manera adequada. Això dóna lloc a una complexitat de temps:

$$T(d) = 4\,T(d-1) + O(1),$$

on d representa el nivell de profunditat. Resolent aquesta recurrència, s'obté $T(d) = O(4^d)$. Notem que el nombre de segments dibuixats per una corba de nivell d és 4^d, la qual cosa també representa el nombre de passos de l'algoritme. En relació a la mida de la sortida, aquesta corba visita $N = 2^d \cdot 2^d = 4^d$ punts, de manera que la complexitat és lineal en el nombre de punts generats (és a dir, $O(N)$), però exponencial respecte al paràmetre de profunditat d. El cost espacial és $O(N)$ per emmagatzemar la llista de punts. Generalment, aquest algorisme es calcula seqüencialment, tot i que en mode no animat es pot executar dins d'un ForkJoinPool per mantenir la interfície responsiva.

C. KochCurve – Corba de Koch (fractal clàssic de dimensió fraccionària)

La corba de Koch parteix d'un segment de línia i, de manera recursiva, substitueix la part central per dues línies que formen un "pic" equilàter. El procediment recursiu es pot plantejar com:

Listing 4. Pseudocodi per a `kochCurve`

```
Algorisme kochCurve(puntA, puntB, nivell):
    si nivell == 0:
       dibuixaSegment(puntA, puntB)
       retornar
    // Calcula punts de divisio del segment AB en
        tres parts iguals
    punt1 = puntA + (1/3) * (puntB - puntA)
    punt3 = puntA + (2/3) * (puntB - puntA)
    // Calcula punt2 formant un pic equilater sobre
        el terc central
    punt2 = punt de pic equilater sobre [punt1,
        punt3]
    // Crides recursives pels quatre segments
    kochCurve(puntA, punt1, nivell-1)
    kochCurve(punt1, punt2, nivell-1)
    kochCurve(punt2, punt3, nivell-1)
    kochCurve(punt3, puntB, nivell-1)
```

Cada iteració substitueix un segment per quatre segments més petits (creant l'efecte "dent de serra"). La complexitat temporal satisfà la recurrència

$$T(n) = 4\,T(n-1) + O(1),$$

on n representa el nivell de recursió. Així, la solució és $T(n) = O(4^n)$, és a dir, el nombre de segments creix exponencialment amb el nivell. Per exemple, a nivell 1 hi ha 4 segments, a nivell 2 hi ha $4^2 = 16$, a nivell 3 hi ha 64, etc. En funció de la sortida, si N és el nombre total de segments produïts, la complexitat és $O(N)$, on $N = 4^n$. A més, la implementació aprofita la paral·lelització mitjançant un ForkJoinPool, dividint el problema en quatre subtasques, cosa que redueix el temps de mur sense modificar l'ordre total de treball. El cost espacial és $O(N)$ per emmagatzemar els punts calculats.

D. SierpinskiTriangle – Triangle de Sierpinski (fractal auto-similar de subdivisió)

El triangle de Sierpinski es construeix recursivament dividint un triangle equilàter en tres triangles més (el triangle central es descarta). L'algoritme de dibuix traça els segments de la frontera de cada triangle resultant. El pseudocodi és el següent:

Listing 5. Pseudocodi per a `sierpinskiTriangle`

```
Algorisme sierpinskiTriangle(vertexA, vertexB,
    vertexC, nivell):
    si nivell == 0:
        dibuixaTriangle(vertexA, vertexB, vertexC)
            // tra a els 3 segments
        retornar
    // Calcula el punt mig de cada aresta
    AB = midpoint(vertexA, vertexB)
    BC = midpoint(vertexB, vertexC)
    CA = midpoint(vertexC, vertexA)
    // Crida recursiva als 3 subtriangles dels
        v rtexs
    sierpinskiTriangle(vertexA, AB, CA, nivell-1)
    sierpinskiTriangle(AB, vertexB, BC, nivell-1)
    sierpinskiTriangle(CA, BC, vertexC, nivell-1)
```

Cada nivell de recursió genera 3 crides recursives, per tant la complexitat temporal satisfà:

$$T(n) = 3\,T(n-1) + O(1),$$

la qual cosa dóna $T(n) = O(3^n)$. Així, per a un nivell n, el nombre de petits triangles creix com 3^n. El cost espacial es manté proporcional al nombre de segments dibuixats, que en el pitjor cas és $O(3^n)$.

E. Resum de complexitats

En resum, els algorismes implementats presenten les següents complexitats asimptòtiques:

- **TrominoTiling:** $O(n^2)$, ja que l'algoritme recorre totes les n^2 cel·les del tauler.
- **HilbertCurve:** $O(4^d)$ per ordre d, que és equivalent a $O(n^2)$ si $n = 2^d$ és la resolució del grid.
- **KochCurve:** $O(4^n)$ per n iteracions, ja que el nombre de segments creix com 4^n.

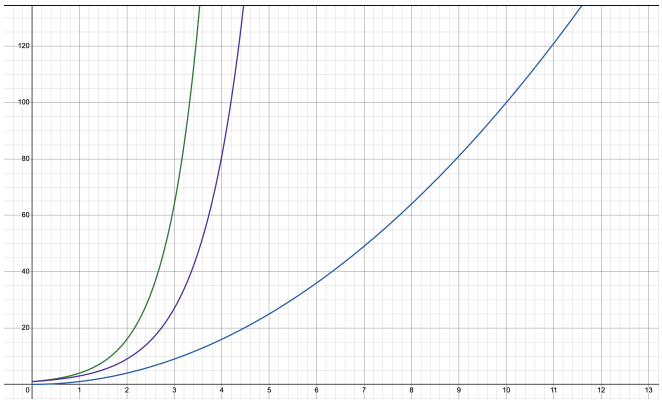

Figura 5. Cost computacion: Verd = Tromino i Hilbert, Purpura = Koch i

- **SierpinskiTriangle:** $O(3^n)$ per n iteracions, degut a la naturalesa ternària de la recursió.

Tots els algorismes fractals (Hilbert, Koch, Sierpinski) presenten complexitats exponencials en el nivell d'auto-similaritat, fet típic en fractals degut a la seva estructura repetitiva infinita. Cal destacar que la implementació concurrent no altera aquestes complexitats en termes de treball total, però pot reduir el temps d'execució real aprofitant diversos processadors en paral·lel. En canvi, el problema de tiling de trominos té una complexitat polinòmica (quadràtica) més favorable, fet que l'ha permès animar fàcilment fins a taulers de mida gran sense un cost prohibitiu.

VII. Conclusió

En aquesta pràctica s'ha dissenyat i implementat un sistema per a la representació de fractals i geometria auto-similar, incloent *Tromino Tiling*, la *Corba de Hilbert*, la *Corba de Koch* i el *Triangle de Sierpinski*. A nivell d'anàlisi computacional, s'ha constatat que l'algorisme de Tromino Tiling presenta un cost polinòmic $O(n^2)$, ja que ha de recórrer totes les n^2 celles d'un tauler, mentre que els fractals (Hilbert, Koch, Sierpinski) mostren costos exponencials en funció de la profunditat recursiva: $O(4^d)$ en Hilbert, $O(4^n)$ en Koch i $O(3^n)$ en Sierpinski. Això fa que, a mesura que augmenta la mida o el nivell de recursió, els temps d'execució dels fractals creixin ràpidament, mentre que Tromino escala de forma més manejable i és abordable per a taulers de dimensió moderada.

Pel que fa a l'arquitectura, el projecte segueix el patró *Model-Vista-Controlador* (MVC) per separar responsabilitats i facilitar la modularitat. La *Vista* (GUI amb Swing/AWT) recull paràmetres i mostra resultats; la *Controladora* gestiona els esdeveniments de l'usuari i orquestra el procés de càlcul, i el *Model* encapsula els algorismes de fractals i de tessel·lació. Aquest disseny permet afegir fàcilment nous patrons aprofitant una interfície comuna (`TilingAlgorithm`), alhora que minimitza l'acoblament entre components.

Per tal d'aprofitar els processadors multi-nucli, s'han integrat mecanismes de *concurrència* i asincronia a través del *framework Fork/Join* [1] de Java. Cada algoritme divideix la seva tasca en subtasques recursives (p. ex. `RecursiveTask`), processant-les en paral·lel sense bloqueigs explícits ni seccions crítiques; això redueix la probabilitat de *deadlocks* i manté la GUI responsiva. S'ha implementat també un sistema de notificacions que desacobla el càlcul del dibuix, de manera que la Controladora rep missatges del Model (inici, passos intermedis, final) i actualitza la interfície sense interrompre el fil principal. A més, l'usuari pot triar entre execució *seqüencial amb animació* —més simple però menys eficient— i *paral·lela sense animació*, la qual divideix de manera immediata el treball entre múltiples fils, assolint *speed-ups* notables a costa d'una major sobrecàrrega i d'estar subjecta als límits de la Llei d'Amdahl [16].

En relació amb l'*entorn de desenvolupament*, s'ha emprat *Java 23* i l'IDE *IntelliJ IDEA*, tot mantenint un control de versions rigorós amb *GitHub*. Aquest enfoc ha facilitat l'organització de les tasques, la integració del codi i la col·laboració entre els membres de l'equip. Finalment, com a reflexió de millores futures, es podrien explorar optimitzacions addicionals (memoització o ús de GPU per a fractals molt profunds), afegir nous patrons (Mandelbrot, Peano, etc.), millorar la GUI (zoom, pan, barra de progrés o estil més modern) i refinar la gestió d'esdeveniments (introduir pausa/reprèn o tractaments d'error més avançats). En conclusió, la pràctica ha complert els objectius d'integrar algoritmes recursius avançats, arquitectura modular i programació concurrent en un entorn estable, i constitueix una base sòlida per a futures ampliacions en el camp de la geometria fractal i la computació d'altes prestacions.

Referències

[1] Oracle, *Fork/Join Framework*, Java Tutorials, 2014.

[2] S. Klymenko, *Java ForkJoinPool: A Comprehensive Guide*, Medium, 2021.

[3] A. Dix et al., *Model-View-Controller (MVC) and Observer*, Univ. of North Carolina, 2016.

[4] Sommerville, I., *Software Engineering*, 10th ed., Addison-Wesley, 2015.

[5] Chacon, S. and Straub, B., *Pro Git*, Apress, 2014.

[6] Cockburn, A., *Agile Software Development: The Cooperative Game*, Addison-Wesley, 2006.

[7] Loeliger, J. and McCullough, M., *Version Control with Git*, O'Reilly Media, 2012.

[8] Sutherland, J., *Scrum: The Art of Doing Twice the Work in Half the Time*, Crown Business, 2014.

[9] Gamma, E., Helm, R., Johnson, R. and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[10] Reenskaug, T., *Models, Views and Controllers*, 1979.

[11] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M., *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, Wiley, 1996.

[12] Dix, A., Finlay, J., Abowd, G. and Beale, R., *Human-Computer Interaction*, Prentice Hall, 2004.

[13] Fowler, M., *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003.

[14] Goetz, B. et al., *Java Concurrency in Practice*, Addison-Wesley, 2006.

[15] Lea, D., *Concurrent Programming in Java: Design Principles and Patterns*, Addison-Wesley, 2000.

[16] Amdahl, G. M., *Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities*, AFIPS Conference Proceedings, 1967.

[17] Herlihy, M. and Shavit, N., *The Art of Multiprocessor Programming*, Morgan Kaufmann, 2008.

[18] Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C., *Introduction to Algorithms*, 3rd ed., MIT Press, 2009.

[19] Falconer, K., *Fractal Geometry: Mathematical Foundations and Applications*, 2nd ed., Wiley, 2003.

[20] Barnsley, M. F., *Fractals Everywhere*, Academic Press, 1988.

[21] Mandelbrot, B. B., *The Fractal Geometry of Nature*, W. H. Freeman, 1982.

[22] Peitgen, H.-O., Jürgens, H. and Saupe, D., *Chaos and Fractals: New Frontiers of Science*, Springer, 1992.

[23] Edgar, G. A., *Measure, Topology, and Fractal Geometry*, Springer, 1990.

[24] Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C., *Introduction to Algorithms*, 3rd ed., MIT Press, 2009.

[25] Kleinberg, J. and Tardos, É., *Algorithm Design*, Pearson, 2006.

[26] Aho, A. V., Hopcroft, J. E. and Ullman, J. D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[27] Sipser, M., *Introduction to the Theory of Computation*, 3rd ed., Cengage Learning, 2012.

[28] Tarjan, R. E., *Data Structures and Network Algorithms*, SIAM, 1983.