Received 25 Febrer, 2025; revised ; Date of publication 21 Març, 2025; date of current version 25 Març, 2025.

Implementació del Model Vista Controlador en una Aplicació Pràctica

Josep Ferriol, Daniel García, Khaoula Ikkene, Biel Perelló

[1]Universitat de les Illes Balears, Departament d'Enginyeria Informàtica

Autor de contacte: Daniel García (email: daniel.garcia19@estudiant.uib.es)

ABSTRACT Aquest document presenta la implementació d'un algorisme de Backtracking per al dibuix d'estructures mitjançant recursivitat. Es detallen els conceptes teòrics del Backtracking, la divisió en subproblemes i la seva resolució mitjançant recursió. A més, es descriu l'arquitectura Model Vista Controlador (MVC) i l'ús del patró per esdeveniments per gestionar la interacció entre els components del sistema, considerant un enfocament centralitzat [11], [12].
L'aplicació desenvolupada permet generar patrons repetitius des dels simples trominos damunt una quadrícula de dimensions $2^n \times 2^n$ amb n fins a triangles de Sierpinski o quadrats recursius. En el cas dels trominos, l'usuari pot seleccionar la posició inicial del quadre lliure i modificar altres paràmetres a través de la interfície gràfica, implementada amb Java Swing [13].

A més cal mencionar que s'han implementat altres opcions optatives, com ara el càlcul del temps de previsió, la implementació de tres fractals adicionals; Triangles de Sierpinski, l'Estora de Sierpinski i el dibuix d'un arbre recursiu. A més, s'ha usat un RecursiveSolver per poder executar fils concurrents per a la resolució dels problemes. Així com dotar a l'usuari la possibilitat de triar la combinació de colors de les figueres. A nivell del dibuix, s'ha implementat, de forma opcional també, un buffer doble usant BufferStrategy [1].

Finalment, es presenta una anàlisi del comportament de l'algorisme segons diferents valors de profunditat de recursió, així com les conclusions sobre l'eficiència i l'escalabilitat del mètode utilitzat.

INDEX TERMS Arquitectura de Programari, Backtracking, Cost Asimptòtic, Fractals, Java Swing, Model-Vista-Controlador, Patró per Esdeveniments, Programació Recursiva

I. Introducció

Aquest projecte té com a objectiu implementar un algorisme de Backtracking que generi patrons de trominos i altres figures geomètriques com Triangles de Sierpinski, Estora de Sierpinski o arbres, aplicant una arquitectura MVC. A més, es pretén analitzar els beneficis d'aquest patró en termes de modularitat [9].

El Model Vista Controlador (MVC) és un patró d'arquitectura de programari àmpliament utilitzat per separar la presentació de dades, la gestió de dades i la lògica del programa. Aquesta separació permet millorar la modularitat i mantenibilitat dels sistemes, facilitant així el desenvolupament i manteniment de les aplicacions [8].

En aquesta memòria, es presenten els conceptes teòrics de l'arquitectura MVC, incloent la seva estructura i els avantatges que ofereix [10]. També s'explica el patró per esdeveniments, que permet una comunicació eficient entre els components del sistema mitjançant la gestió d'esdeveniments i notificacions.

A continuació, es descriu l'entorn de programació utilitzat per al desenvolupament de l'aplicació, incloent les llibreries de Swing per a la interfície gràfica i les eines emprades per a la gestió del codi i el control de versions [13].

La part central de la pràctica consisteix en desenvolupar una aplicació que implementi l'algorisme de Backtracking per dibuixar estructures com trominos sobre una quadrícula de dimensions $2^n \times 2^n$. L'usuari pot seleccionar quina figura vol que es dibuixi i, en el cas del tromino, la posició inicial del quadre lliure a través d'una interfície gràfica desenvolupada amb Java Swing.

Finalment, s'analitzen els resultats obtinguts i es presenten les conclusions del projecte i es proposen possibles millores futures per optimitzar el sistema.

II. Conceptes Teòrics

A. Model Vista Controlador (MVC)

El Model Vista Controlador (MVC) és un patró d'arquitectura de programari que separa l'aplicació en tres components principals: el Model, la Vista i el Controlador. Aquesta separació permet una millor modularitat i mantenibilitat del codi [8].

- **Model**: Gestiona les dades i la lògica de negoci de l'aplicació. És responsable de l'accés a la base de dades, càlculs i qualsevol altra lògica de negoci necessària. La lògica de negoci, també coneguda com a lògica de l'aplicació, és un conjunt de regles que es segueixen en el programari per reaccionar davant diferents situacions.
- **Vista**: S'encarrega de la presentació de les dades a l'usuari. És responsable de la interfície gràfica i de mostrar la informació de manera comprensible. La Vista actua com el frontend o interfície gràfica d'usuari (GUI).
- **Controlador**: Actua com a intermediari entre el Model i la Vista. Gestiona les interaccions de l'usuari, actualitza el Model i refresca la Vista. El Controlador és el cervell de l'aplicació que controla com es mostren les dades.

En aquest projecte, el Controlador rep les accions de l'usuari a través dels botons de la interfície i gestiona l'execució dels càlculs corresponents. Això inclou la creació i gestió dels fils de càlcul, la recollida dels resultats i la seva actualització en la Vista. Gràcies a aquest enfocament, la separació de responsabilitats facilita el manteniment i l'ampliació de l'aplicació [9].

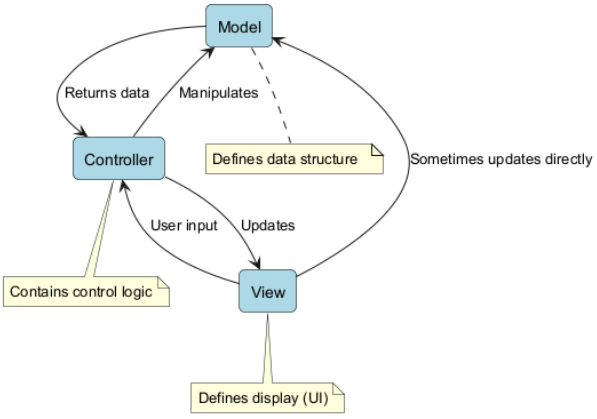

FIGURE 1. Esquema del Model-Vista-Controlador.

El concepte de MVC va ser introduït per primera vegada per Trygve Reenskaug, qui el va proposar com una forma de desenvolupar el GUI d'aplicacions d'escriptori. Avui en dia, el patró MVC s'utilitza per a aplicacions web modernes perquè permet que l'aplicació sigui escalable, mantenible i fàcil d'expandir.

Les principals utilitats del patró MVC inclouen:

- **Modularitat**: Facilita la separació de responsabilitats, permetent que cada component es desenvolupi i es mantingui de manera independent. Això respecta el principi de la responsabilitat única, on una part del codi no ha de saber què fa tota l'aplicació, només ha de tenir una responsabilitat específica.
- **Reutilització de codi**: Els components poden ser reutilitzats en diferents parts de l'aplicació o en altres projectes.
- **Facilitat de manteniment**: La separació de responsabilitats facilita la detecció i correcció d'errors, així com la implementació de noves funcionalitats.
- **Separació de preocupacions (SoC)**: El patró MVC ajuda a dividir el codi frontend i backend en components separats, fent que sigui molt més fàcil gestionar i fer canvis a qualsevol dels costats sense que interfereixin entre si.

El patró MVC és especialment útil quan diversos desenvolupadors necessiten actualitzar, modificar o depurar una aplicació completada simultàniament. Aquesta separació de preocupacions permet que els desenvolupadors treballin en diferents parts de l'aplicació sense afectar altres parts del codi. En el cas d'aquesta pràctica, podries canviar de vista o de solver i continuaria funcionant amb els nous components.

En resum, el patró MVC és una eina poderosa per a la construcció d'aplicacions escalables i mantenibles, permetent una clara separació de responsabilitats i facilitant la gestió del codi.

B. Patró per Esdeveniments

El patró per esdeveniments és un mecanisme de comunicació entre components que permet gestionar esdeveniments i notificacions de manera eficient [12]. En aquest patró, els components poden generar esdeveniments i subscriure's a esdeveniments generats per altres components.

- **Generació d'esdeveniments**: Els components poden generar esdeveniments quan es produeixen canvis en el seu estat o quan es completen determinades accions. En aquest projecte, per exemple, l'acció de prémer un botó per iniciar el càlcul de les distintes formes genera un esdeveniment que inicia l'execució dels càlculs.
- **Subscripció a esdeveniments**: Els components poden subscriure's a esdeveniments generats per altres components per rebre notificacions i actuar en conseqüència. En l'aplicació, el Controlador s'encarrega de subscriure's als esdeveniments generats per la Vista (com un clic en un botó) per gestionar l'execució dels càlculs i actualitzar la Vista amb els resultats.
- **Desacoblament**: El patró per esdeveniments permet desacoblar els components, ja que no necessiten conèixer els detalls dels altres components amb els quals interactuen. Això facilita l'escalabilitat i el manteniment de l'aplicació, ja que es poden afegir nous components o modificar els existents sense afectar la resta de

l'aplicació. Per exemple, la Vista i el Model estan desacoblats, i el Controlador s'encarrega de gestionar la interacció entre ambdós a través dels esdeveniments.

C. Càlcul del Cost Asimptòtic

El càlcul del cost asimptòtic s'utilitza per analitzar l'eficiència dels algorismes en termes de temps d'execució i ús de memòria. Es basa en la notació Big-O, que descriu el comportament d'un algorisme a mesura que la mida de les dades d'entrada creix fins a l'infinit. Aquesta notació permet mesurar com s'incrementa el temps d'execució a mesura que augmenta la mida de l'entrada. Matemàticament, defineix el límit d'una funció quan els seus arguments tendeixen a l'infinit o a un valor determinat, representant el pitjor cas del temps d'execució en funció de la mida de les dades.

La notació Big-O és important perquè permet als desenvolupadors i científics de dades avaluar i comparar l'eficiència de diferents algorismes. Això ajuda a triar l'algorisme més adequat per a una tasca específica, optimitzant el rendiment d'aplicacions i sistemes [3].

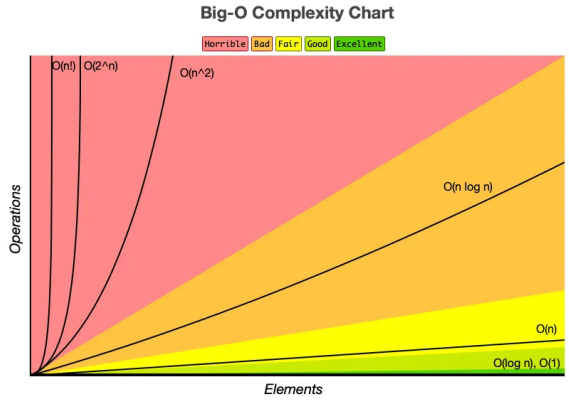

FIGURE 2. Exemple de diferents complexitats de la notació Big-O.

Per calcular la notació Big-O d'un algorisme, s'analitzen els bucles, bucles anidats, recursions i operacions dominants del codi, que són les que més afecten el temps d'execució. A més, un factor important que afecta el rendiment i l'eficiència del programa és el maquinari, el sistema operatiu i la CPU que s'utilitza. No obstant això, això no es té en compte quan s'analitza el rendiment d'un algorisme. En canvi, el que importa és la complexitat temporal i espacial en funció de la mida de l'entrada.

En resum, la notació Big-O és una eina essencial per avaluar l'eficiència dels algorismes i triar els més adequats per a cada situació, optimitzant així el rendiment de les aplicacions.

A la part d'analisi s'explicará amb detall el cost exponencial $O(a^n)$, que és el propi dels algorismes desenvolupats en aquesta pràctica.

D. Backtracking

El backtracking és una tècnica algorísmica per resoldre problemes mitjançant l'exploració de diferents opcions de manera incremental. Si una elecció no condueix a una solució, es retrocedeix fins al punt de decisió anterior per provar una alternativa. Aquest enfocament és especialment útil en problemes que requereixen examinar múltiples possibilitats per trobar una solució vàlida o òptima. Quan s'arriba a un carreró sense sortida, l'algoritme torna enrere al punt de decisió anterior i explora un camí diferent fins que es troba una solució o s'han esgotat totes les possibilitats.

```
boolean findSolutions(n, other params):
if (found a solution):
displaySolution();
return true;
for (val = first to last):
    if (isValid(val, n)) :
        apply Value(val, n);
        if (findSolutions(n+1, other params))
            return true;
        removeValue(val, n);
    return false;
```

FIGURE 3. Esquema general d'una algorisme de backtracking

E. Fractals

Un fractal és una forma geomètrica que exhibeix autosimilitud, és a dir, la seva estructura es repeteix a diferents escales, de manera que una part ampliada conserva patrons similars a la forma global. A diferència de les figures geomètriques clàssiques, els fractals poden tenir una dimensió fractal no entera, situant-se entre les dimensions tradicionals, com entre una línia i una superfície. Es generen mitjançant processos iteratius o recursius, on una funció s'aplica repetidament, produint patrons d'alta complexitat. En programació, els fractals es construeixen amb funcions recursives, aplicació de fórmules matemàtiques iteratives (com en els conjunts de Mandelbrot i Julia [5]) o a través sistemes de funcions iterades (IFS) [6]) basats en transformacions geomètriques. Els fractals tenen múltiples aplicacions en informàtica, com en gràfics per ordinador per modelar paisatges i textures, en compressió d'imatges per reduir-ne la mida sense pèrdua de qualitat, en la simulació de fenòmens naturals com el creixement de plantes o la turbulència en fluids, i fins i tot en criptografia i anàlisi de dades, per a la generació de claus segures i la detecció de patrons complexos.

III. Entorn de Programació

Per al desenvolupament d'aquest projecte s'ha emprat un conjunt d'eines i tecnologies que han facilitat la implementació del patró MVC i l'anàlisi dels algorismes de suma i producte de matrius. A continuació, es descriuen les principals eines utilitzades:

- **Llenguatge de programació: Java**
  Java és un llenguatge de programació orientat a objectes que ofereix una gran portabilitat i una àmplia gamma de llibreries i eines per al desenvolupament d'aplicacions. La seva sintaxi clara i la seva robustesa el fan ideal per a projectes que requereixen una alta mantenibilitat i escalabilitat. A més, Java és independent de la plataforma, la qual cosa permet executar l'aplicació en diferents sistemes operatius sense necessitat de modificar el codi.
- **Llibreries utilitzades: Swing**
  Swing és una llibreria de Java per a la creació d'interfícies gràfiques d'usuari (GUI). Proporciona un conjunt complet de components gràfics, com ara botons, camps de text, taules i panells, que permeten construir interfícies d'usuari riques i interactives. Swing és altament personalitzable i permet crear interfícies gràfiques que s'adapten a les necessitats específiques de l'aplicació.
- **IDE: IntelliJ IDEA**
  IntelliJ IDEA és un entorn de desenvolupament integrat (IDE) per a Java que ofereix una àmplia gamma de funcionalitats per a facilitar el desenvolupament d'aplicacions. Algunes de les seves característiques més destacades inclouen la completació de codi intel·ligent, la navegació ràpida pel codi, les eines de refactorització i la integració amb sistemes de control de versions. Aquestes funcionalitats ajuden a millorar la productivitat i a mantenir un codi net i organitzat.
- **Control de versions: GitHub**
  GitHub és una plataforma de desenvolupament col·laboratiu que utilitza el sistema de control de versions Git. Permet gestionar el codi font del projecte, fer seguiment dels canvis, col·laborar amb altres membres de l'equip i mantenir un historial complet de les modificacions realitzades. GitHub també ofereix eines per a la gestió de projectes, com ara issues i pull requests, que faciliten la coordinació i la revisió del codi.

L'ús d'aquestes eines i tecnologies ha permès desenvolupar una aplicació robusta i mantenible, seguint els principis del patró MVC i assegurant una bona gestió del codi i la col·laboració entre els membres de l'equip.

IV. Desenvolupament

En aquesta secció es descriu el desenvolupament de la pràctica, que consisteix en la implementació d'una aplicació de dibuix de patrons geometrics basada en el patró Model Vista Controlador (MVC) i el patró per esdeveniments.

A. Estructura

- **Model**: Per a la gestió de les dades del programa, s'ha creat la classe **Dades**, que registra el tauler, element usat per guardar de forma intermedia, els resultats de dibuix dels algorismes implementats, un enum Tipus, que indica el tipus de peça que construeix un algorisme donat, la profunditat que tria l'usuari i una variable per guardar el valor de la constant multiplicativa. A part de la classe Dades, Model implementa també els algorismes per solucionar els següents problemes: Problema dels Tromins, problema dels triangles de Sierpinski, problema de l'estora de Sierpinski, i el dibuix recursiu d'arbres. A més, implementa una classe abstracta (`RecursiveSolver`) que implementen els **solvers** i que'ls permet utilitzar múltiples fils concurrents.
  - **RecursiveSolver** Aquesta classe abstracta realitza l'encapsulament d'un `solver` per què pugui llançar fils de manera transparent. Per cada cridada recursiva es crea un fil, fet que permet augmentar la eficiencia del proces, però es necessària una gestio dels fils per a no saturar la CPU. És per a la gestió de fils s'utilitza un `ExecutorService` de tamany linealment variable amb el numero de nuclis de la CPU i una cua de `Runnable` de fils pendents d'executar.
- **Vista**: Les classes `Finestra` i els distints `visualitzadors` s'encarreguen de la presentació de les dades a l'usuari. Utilitzen la llibreria Swing per crear la interfície gràfica i mostrar els resultats dels `solvers`. A més, els diferents `visualitzadors` implementen una classe abstracta que els permet executar-se en un propi fil amb un doble buffer.
  - **Finestra** Aquesta classe representa la finestra principal de l'aplicació. Conté un `JComboBox` per seleccionar el tipus de visualització (tromino, triangles, quadrat i arbre), botons per executar, aturar, esborrar i activar/desactivar colors en la visualització, així com un camp de text per introduir valors numèrics i quatre botons per seleccionar els colors per a la visualització. Gestiona el canvi dinàmic entre diferents visualitzadors mitjançant la funció `replace(Comunicar nouDibuix)`.
  - **CanvasDobleBuffer** Aquesta classe abstracta permet encapsular la lògica del doble buffer per als altres `visualitzadors`. La implementació utilitza un `BufferStrategy` i un fil que repinta el Canvas de forma periòdica, per exemple 60FPS. D'aquesta manera no es necessari que els `solvers` comuniquin un repintat a cada canvi, alliberant de carrega el procés de redibuixat.
  - **DibuixSierpinski** Classe encarregada de dibuixar el Triangle de Sierpinski en un panell gràfic (`CanvasDobleBuffer`). Recorre una matriu de dades per determinar quins triangles ha de pintar i utilitza colors per diferenciar les profunditats de la figura. Inclou un "easter egg" per canviar el color a daurat en determinades condicions. [7]
  - **DibuixTromino** S'encarrega de representar la solució del problema dels Trominos. Dibuixa una quadrícula i pinta les peces dels trominos amb colors diferenciats quan l'opció de color està acti-

vada. També detecta clics de l'usuari per identificar caselles i enviar informació a la classe principal (`Main`).

– **Altres:** Per pintar els resultats de l'estora i els arbres hi ha les dues classes encarregades, (`DibuixCarpet`) i (`DibuixImage`) respectivament.

- **Controlador** La classe `Main` actua com a controlador principal de l'aplicació, gestionant les interaccions de l'usuari i coordinant les operacions entre el model i la vista. També és responsable de la gestió de processos i de l'execució de les diferents funcionalitats.
  Elements principals

  – **Inicialització**:

  * L'atribut `dades` representa el model de dades de l'aplicació.
  * L'atribut `finestra` és la interfície gràfica, la qual permet a l'usuari interactuar amb el sistema.
  * La llista `processos` conté els processos en execució.
  * L'objecte `executor` és un `ExecutorService` amb un *thread pool* de mida 16 per executar tasques en paral·lel.

  – **Creació i gestió dels components**:

  * El mètode `init()` inicialitza les dades i els processos i crea la interfície gràfica en un fil separat mitjançant l'executor.

  – **Gestió de la matriu de dades**:

  * El mètode `getMatriu()` retorna la matriu de dades.
  * Quan es rep un missatge de tipus `"N"`, es crea una nova matriu de mida $n \times n$ i es notifica la finestra perquè es redibuixi.

  – **Comunicació**:

  * La classe `Main` implementa la interfície `Comunicar`, que defineix el mètode `comunicar`.
  * Aquest mètode rep instruccions en forma de cadena de text i executa les accions corresponents mitjançant un sistema de control de missatges.
  * Exemples de missatges:
    · `"tempsReal"`, `"tempsEsperat"` → S'envien directament a la interfície gràfica.
    · `"N:n"` → Es crea una nova matriu de mida $n \times n$.
    · `"inici"` → Es defineix el forat per a la resolució del problema dels trominos i s'inicia el càlcul.

  – **Gestió de processos d'execució**:

  * Quan es rep un missatge de tipus `"executar"`, la classe `Main` inicia el càlcul corresponent segons el tipus de visualització:
    · `"tromino"`: Executa el solver dels Trominos.
    · `"triangles"`: Executa el solver del Triangle de Sierpinski.
    · `"quadrat"`: Executa el solver de l'estora de Sierpinski.
    · `"arbre"`: Executa el solver dels arbres
  * La creació i execució d'aquests processos es realitza mitjançant reflexió, instanciant dinàmicament les classes i executant-les en fils separats.

  – **Gestió de la finalització i neteja de processos**:

  * Quan es rep el missatge `"aturar"`, s'atura l'execució de tots els processos en marxa.
  * Amb el missatge `"borrar"`, s'atura tot, es netegen les dades i es redibuixa la interfície gràfica.

També s'ha de comentar que la interfície `Comunicar` defineix el contracte bàsic per a la comunicació entre els diferents components de l'aplicació, i conté un únic mètode:

```
void comunicar(String s);
```

Aquest mètode és utilitzat per transmetre missatges entre les diferents parts del sistema, especialment entre la vista (finestra gràfica), el controlador (`Main`) i el model (en aquest cas, la classe `Dades`). La interfície `Comunicar` és una eina fonamental per mantenir una separació clara entre la lògica de l'aplicació i la interfície d'usuari, permetent una comunicació eficient i flexible.
En resum, la classe `Main` és el nucli de l'aplicació, ja que actua com a intermediari entre la interfície gràfica i el model de dades. La seva estructura modular permet gestionar diferents tipus de visualitzacions i optimitzar el rendiment mitjançant execució concurrent.

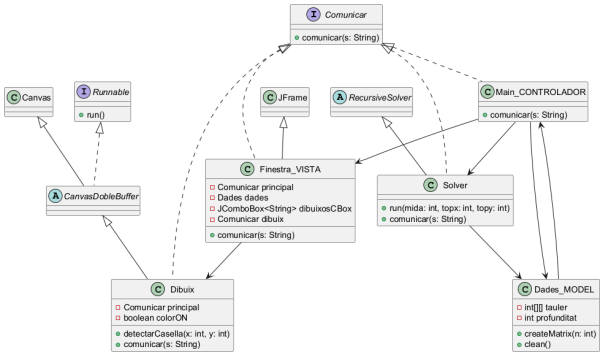

FIGURE 4. **Diagrama molt senzill de classes.**

B. Implementació del Patró per Esdeveniments

L'aplicació utilitza el patró per esdeveniments per gestionar la comunicació entre els components de manera eficient. Aquest patró permet que els components es comuniquin entre ells mitjançant l'enviament i la recepció de missatges, sense necessitat de conèixer els detalls interns dels altres components.

1) Interfície `Comunicar`

La interfície `Comunicar` defineix un mètode `comunicar` que permet enviar missatges entre els components. Aquesta interfície és implementada per diverses classes de l'aplicació, permetent una comunicació uniforme i consistent. La definició de la interfície és la següent:

```
public interface Comunicar {
    public void comunicar(String s);
}
```

2) Classe `Main`

La classe `Main` actua com a controlador central de l'aplicació i és responsable de gestionar la comunicació entre els components, com s'ha explicat anteriorment. Quan l'usuari interactua amb la interfície gràfica, la classe `Main` rep els missatges corresponents i els processa per iniciar, aturar o actualitzar les operacions de generació de figures. El mètode `comunicar` de la classe `Main` processa els missatges i crida als mètodes adequats per gestionar les operacions.

3) Classes de `solvers`

Tots els solvers implementen la interfície Comunicar per rebre missatges del controlador i aturar els fils d'execució quan sigui necessari. Quan es rep un missatge per aturar l'operació, el mètode comunicar d'aquestes classes estableix una variable booleana stop a true, la qual cosa fa que el mètode principal s'aturi. A més, estenen de la classe **RecursiveSolver** per poder executar concurrentment diversos fils.

4) Classe `Finestra`

La classe `Finestra` també implementa la interfície `Comunicar` per rebre missatges del controlador i actualitzar la interfície gràfica en conseqüència. Aquesta classe rep el missatge per comunicar i també l'envia a les classes de Dibuix per comunicar.

5) Flux de Comunicació

El flux de comunicació entre els components es pot resumir en els següents passos:

1) L'usuari interactua amb la interfície gràfica mitjançant la classe `Finestra`, per exemple, seleccionant una opció del menú o introduint un valor per dibuixar una figura.
2) La classe `Finestra` envia un missatge al controlador (`Main`) mitjançant el mètode `comunicar`.
3) El controlador (`Main`) processa el missatge i, si és necessari, executa un procés concret (per exemple, `TrominoSolver`, `SierpinskiSolver`, `RecursiveTree` o `CarpetSierpinski`) en un fil separat utilitzant un `ExecutorService`.
4) Els solucionadors (`TrominoSolver`, `SierpinskiSolver`, etc.) realitzen els càlculs corresponents en paral·lel i, durant l'execució, envien missatges al controlador per actualitzar l'estat del procés.
5) El controlador informa la interfície gràfica (`Finestra`) sobre l'estat de l'execució mitjançant missatges com `tempsReal` o `tempsEsperat`.
6) Quan un procés finalitza o es rep un missatge per aturar-lo, aquest comunica al controlador la seva terminació (`aturar`), i aquest ho notifica a la interfície perquè actualitzi els resultats.
7) La interfície gràfica (`Finestra`) utilitza la classe `DibuixTromino` , per exemple, per actualitzar la visualització de la figura calculada.

Aquesta arquitectura basada en missatges garanteix la separació de responsabilitats entre la interfície, el controlador i els processos de càlcul, permetent una execució eficient i flexible de les operacions.

C. Solvers

En aquesta secció, abordarem de manera detallada els algorismes *solvers*. En general, tenen la següent estructura:

1) Implementació d'un mètode recursiu que resol el problema donat.
2) Implementació del mètode per al cas base, que inclou l'emmagatzematge de les dades als taulers.
3) Implementació del mètode `run()`, ja que els *solvers* es basen en fils d'execució.
4) Implementació del mètode `stop()`, per aturar l'execució dels fils.
5) Finalment, implementació del mètode `communicar()`, per la comunicació entre fils o components.

1) TrominoSolver

Aquesta classe soluciona el següent problema; donat un tauler de mida exponencial 2^n, i amb un forat en qualsevol posició (x, y) del tauler, omple el tauler amb tromins de forma recursiva. Aquest algoritme segueix l'estratègia de divideix i venceràs, fragmentant el tauler en quadrants més petits fins a aconseguir una mida mínima de $2x2$. Per a cada

subdivisió, localitza la posició del forat existent i col·loca un tromino central per a garantir una cobertura completa del tauler. A mesura que avança la recursió, l'algoritme executa múltiples fils per resoldre simultàniament les diferents regions del tauler, millorant així el rendiment en entorns de processament paral·lel. Al final de l'execució, es mesura el temps emprat i es calcula una constant multiplicativa basada en la profunditat del tauler, la qual permet estimar el temps d'execució per a futures execucions.

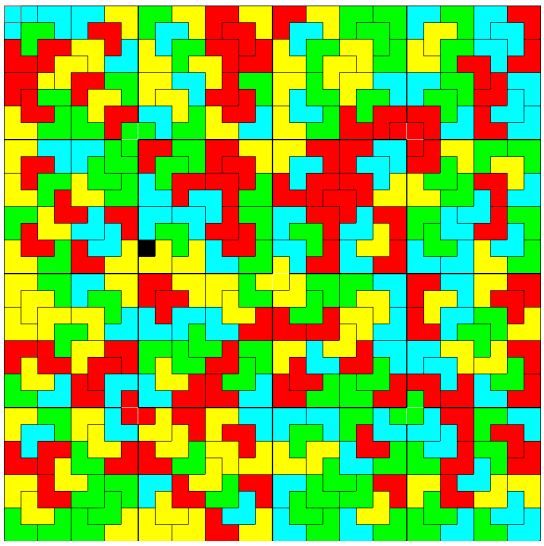

FIGURE 5. **Exemple d'un tromino de profunditat 5.**

2) SierpinskiTriangleSolver

La fractal dels triangles de Sierpinski tracta de dividir un triangle en tres triangles més petits de manera recursiva, eliminant el triangle central en cada iteració. Aquest procés continua fins que s'arriba a un nivell de profunditat determinat.

La classe SierpinskiTriangleSolver resol aquest problema aplicant recursivitat per generar la figura. Comença amb un triangle gran i divideix recursivament l'espai en tres sub-triangles fins que la mida mínima s'assoleix. La classe utilitza un tauler bidimensional int[][] tauler per representar la figura i marca les posicions corresponents amb data.setValor(x, y, 1). L'execució es fa de manera asíncrona mitjançant fils (runThread), permetent una representació dinàmica de la construcció del fractal. A més, mesura el temps d'execució i estima el temps esperat basant-se en la profunditat de recursió, comunicant la informació a través de p.comunicar, sent p la instància del Programa principal Main.

3) SierpinskiCarpetSolver

L'estora de Sierpinski és una fractal que segueix un patró consistent en dividir un quadrat en nou quadrats de mida igual, realitzant tres divisions horitzontals i tres verticals. Tots els quadrats es pinten, excepte el central.

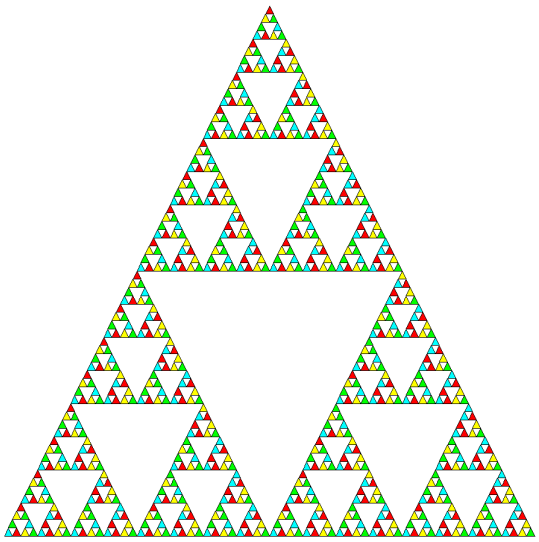

FIGURE 6. **Exemple d'un triangle de profunditat 7.**

L'algorisme recursiu de la classe implementa exactament aquest procés. Pel que fa al tauler, es va omplint amb valors enters incrementals que representen els quadrats, establint l'ordre en què han de ser pintats.

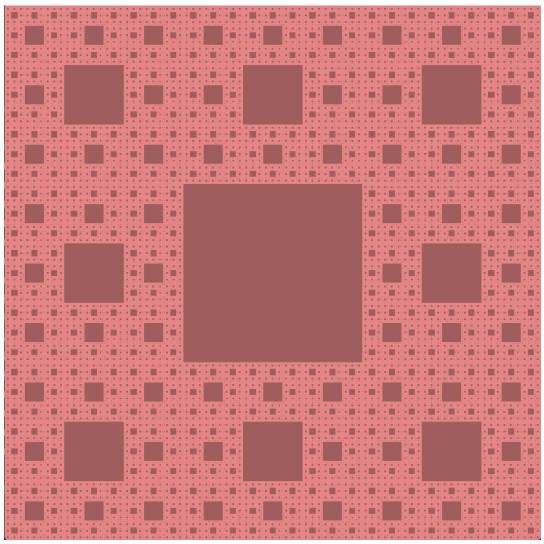

FIGURE 7. **Exemple d'un estora de profunditat 6.**

4) TreeSolver

Es tracta de dibuixar un arbre de forma recursiva. L'element base de l'arbre és la branca, i a mesura que s'incrementa la profunditat, les arrels es van afegint de manera exponencial.

La implementació de l'algorisme utilitza dues propietats geomètriques trivials per calcular les posicions de les branques, tenint en compte el seu angle d'inclinació i la seva longitud.

```
int x2 = x1 + (int) (Math.cos(Math.toRadians(angle
    )) * treeLogSize );
int y2 = y1 + (int) (Math.sin(Math.toRadians(angle
    )) * treeLogSize);
```

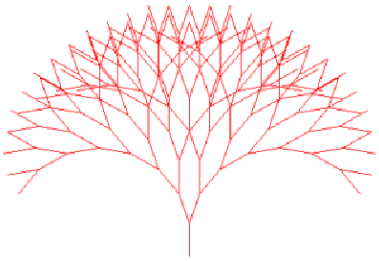

FIGURE 8. Exemple d'un arbre recursiu de profunditat 8.

Ara bé, aquest model no representa la figura de l'arbre en forma de matriu, sinó com una imatge de tipus `BufferedImage` de mida personalitzada.

D. Interfície Gràfica

L'aplicació utilitza la llibreria Swing per crear una interfície gràfica que permet a l'usuari seleccionar i visualitzar diferents figures geomètriques recursives. L'usuari pot triar entre trominos, triangles de Sierpinski, quadrats i arbres, així com modificar l'aparença dels dibuixos canviant els colors o actualitzant la representació.

1) Classes de Dibuix

Les classes de `Dibuix` s'encarregen de representar gràficament la solució del problema. Aquestes classe implementen la interfície `Comunicar` per comunicar-se amb la classe principal i actualitzar la visualització i estenen `CanvasDobleBuffer` per al doble buffer.

Per exemple, la classe `DibuixTromino` s'encarrega de dibuixar la matriu solució del solver. Els elements principals d'aquesta classe són:

- **Pintura i dibuix**: Utilitza el mètode `pintar()` per dibuixar la matriu de trominos sobre el panell gràfic.
- **Detecció d'interacció**: Implementa un `MouseListener` per detectar clics a la graella i comunicar la posició seleccionada a la classe principal.
- **Colorejat dinàmic**: Assigna colors diferents als trominos depenent de la selecció previa.
- **Dibuix de vores**: Dibuixa només les vores exteriors dels trominos per millorar la claredat visual.

2) Classe Finestra

La classe `Finestra` és responsable de crear la interfície gràfica principal de l'aplicació. Aquesta classe utilitza diversos components de Swing per permetre a l'usuari seleccionar diferents figures i interactuar amb la visualització.

Els elements principals de la interfície inclouen:

- **ComboBox de selecció**: Un `JComboBox` que permet seleccionar entre trominos, triangles de Sierpinski, quadrats i arbre.
- **Botons d'acció**: Inclou botons per pintar, aturar l'execució, esborrar la visualització i canviar els colors dels dibuixos.
- **Camp de text per paràmetres**: Un `JTextField` on l'usuari pot introduir valors per definir la mida de la figura generada.
- **Gestió dinàmica de components**: Quan es selecciona una nova figura, es reemplaça el panell de dibuix anterior per un de nou sense reiniciar tota la interfície.

3) Comunicació entre classes

Les classes `DibuixTromino` i `Finestra` es comuniquen mitjançant la interfície `Comunicar`, que permet enviar missatges per actualitzar l'estat de la interfície i la representació gràfica.

E. Gestió de Fils d'Execució

La classe `Main` utilitza un `ExecutorService` per gestionar els fils d'execució dels distints solvers. Això permet executar les operacions de manera concurrent i millorar l'eficiència de l'aplicació. Els fils es poden iniciar i aturar mitjançant missatges enviats a través de la interfície `Comunicar`.

1) ExecutorService

L'`ExecutorService` és una interfície que proporciona mecanismes per gestionar un conjunt de fils d'execució. En aquest projecte, s'utilitza per executar els distints solvers en fils separats [14]. La classe `Main` crea un `ExecutorService` amb un grup de fils fixos mitjançant el mètode `Executors.newFixedThreadPool(16)`:

```
executor = Executors.newFixedThreadPool(16);
```

Això permet tenir fins a 16 fils d'execució simultanis, assegurant que les operacions de càlcul i resolució de problemes es puguin executar de manera concurrent.

2) Inici i Aturada de Fils

Quan l'usuari interactua amb la interfície gràfica per iniciar una operació, la classe `Main` rep un missatge

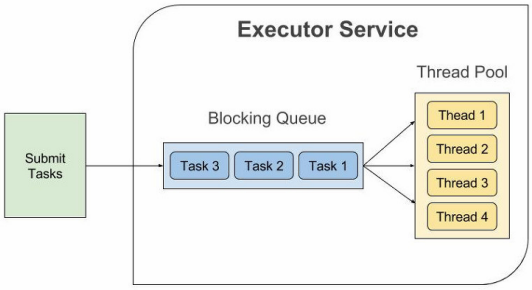

FIGURE 9. Esquema de funcionament de l'Executor Service a Java.

mitjançant el mètode `comunicar`. Aquest mètode processa el missatge i, si és necessari, crea instàncies de les classes `TrominoSolver`, `SierpinskiSolver`, `RecursiveTree` o `CarpetaSierpinski` per executar les operacions en fils separats. Per exemple, quan es rep un missatge per iniciar l'execució del Tromino Solver, es crea i s'executa una instància de la classe corresponent en un fil separat:

```
TrominoSolver trominoTask = new TrominoSolver(this,
     dades);
processos.add(trominoTask);
executor.execute(trominoTask);
```

D'aquesta manera, les operacions es realitzen de forma concurrent, aprofitant els recursos disponibles per millorar el rendiment de l'aplicació.

3) Casos de Concurrència

L'aplicació pot gestionar diferents casos de concurrència, depenent de les operacions que l'usuari decideixi executar:

- **Execució de Solvers Específics**: Quan l'usuari decideix executar un solver, com per exemple el Tromino Solver, es crea i executa una instància d'aquest solver en un fil separat, aturant l'anterior:

```
Comunicar proces = (Comunicar) classe
    .getConstructor(Main.class, Dades.
        class)
    .newInstance(this, dades);
processos.add(proces);
executor.execute((Runnable) proces);
```

4) Aturada de Fils

Quan es rep un missatge per aturar una operació, el mètode `comunicar` de la classe `Main` envia un missatge a les instàncies dels solvers per aturar els fils d'execució. Això es fa establint la variable booleana `stop` a `true`, la qual cosa fa que el bucle principal de cada solver s'aturi de forma controlada. Per exemple:

```
for (Comunicar enmarxa : processos) {
    enmarxa.comunicar("aturar");
}
```

Això assegura que les operacions es puguin aturar de manera controlada i que els recursos es puguin alliberar correctament.

5) Implementació de la Classe `Main`

La classe `Main` és la responsable de gestionar els processos i filtrar els missatges. A continuació es mostra una part de la classe `Main`, que inclou la gestió d'operacions en fils d'execució:

```
public class Main implements Comunicar {

    private Comunicar finestra;
    private Dades dades;
    private ArrayList<Comunicar> processos = null;
    private final ExecutorService executor =
        Executors.newFixedThreadPool(16);

    public void comunicar(String s) {
        String[] params = s.split(":");
        switch (params[0]) {
            case "executar":
                switch (params[1]) {
                    case "tromino":
                        executar(TrominoSolver.
                            class, (int) Math.pow
                            (2, Integer.parseInt(
                            params[2])));
                        break;
                    case "triangles":
                        executar(SierpinskiSolver.
                            class, (int) Math.pow
                            (2, Integer.parseInt(
                            params[2]) - 1));
                        break;
                    [...]
                    // altres operacions
                }
                break;
            case "aturar":
                for (Comunicar proces : processos)
                    {
                    proces.comunicar("aturar");
                }
                break;
            [...]
            // altres casos
        }
    }
```

```
    private void executar(Class<? extends Comunicar
        > clase, int profunditat) throws
        NoSuchMethodException,
        InvocationTargetException,
        InstantiationException,
        IllegalAccessException {
        for (Comunicar enmarxa : processos) {
            enmarxa.comunicar("borrar");
        }
        processos.clear();
        dades.setProfunditat(profunditat);
        Comunicar proces = (Comunicar) clase.
            getConstructor(Main.class, Dades.class)
            .newInstance(this, dades);
        processos.add(proces);
        executor.execute((Runnable) proces);
    }
}
```

6) Conclusió

L'ús de l'`ExecutorService` en la classe `Main` i `RecursiveSolver` per gestionar els fils d'execució és fonamental per a la implementació concurrent de les operacions. Aquesta gestió permet a l'aplicació realitzar càlculs intensius de manera eficient, millorant el rendiment i mantenint l'aplicació responsiva. Gràcies a la separació de les operacions en fils independents, es garanteix una bona escalabilitat i flexibilitat del sistema.

També gràcies al **MVC** ha estat possible implementar de forma independent els **solvers** com els **visualitzadors**, permetent una millora continua d'un sense perill de rompre l'altre.

V. Resultats i Anàlisi

En aquesta secció, analitzarem el cost asimptòtic dels algorismes implementats i compararem el temps real d'execució amb el temps estimat a partir de la constant multiplicativa.

Per a aquest anàlisi, executarem cada algorisme amb diferents valors de profunditat n i recopilarem els resultats.

Les gràfiques següents il·lustren els resultats obtinguts.

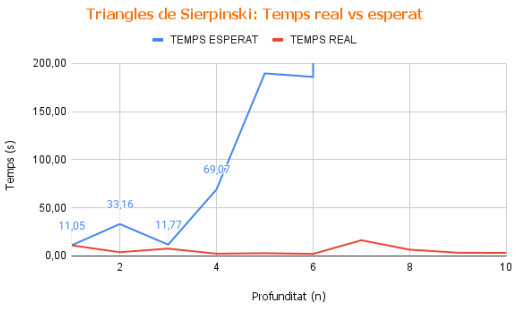

FIGURE 10. Triangles de Siperpinski: Temps real vs esperat

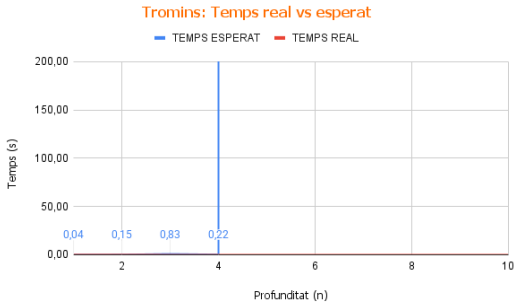

FIGURE 11. Tromins: Temps real vs esperat

FIGURE 12. Estora de Sierpinski: Temps real vs esperat

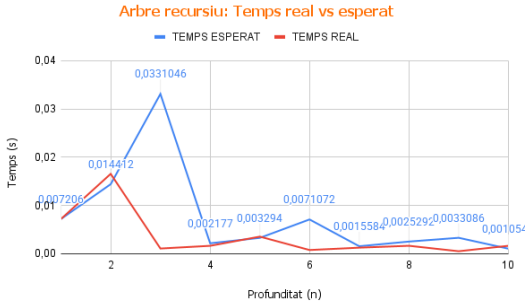

FIGURE 13. Arbre recursiu: Temps real vs esperat

Començant pels triangles de Sierpinski, el cost computacional del seu algorisme és de $O(3^{n/2})$, fent tres crides recursives, i en cada cridada el conjunt n es redueix a la meitat. Això implica que, a mesura que augmenta n, el nombre de subdivisions creix exponencialment, fent que el cost computacional es dispari i limitant la profunditat màxima viable per a la seva representació gràfica.

La constant multiplicativa usada per als càlculs estimats s'actualitza en cada execució fins que arriba a estabilitzar-se. Tot i això, s'observa que a partir de $n = 6$, el temps esperat es torna tan gran que ja no es pot apreciar en la gràfica, i per als darrers valors de profunditat, $n = 9$ i $n = 10$, el temps calculat tendeix a l'infinit, fent impossible la seva generació en temps raonable.

No obstant això, a la pràctica, el temps d'execució real de l'algorisme és gairebé lineal, amb una complexitat efectiva de $O(n)$. Ja que el procés d'omplir la matriu representativa és lineal, tenguint en compte que cada punt es marca només una vegada sense recomputacions innecessàries.

Per altra banda, el dibuix dels trominos i de l'estora de Sierpinski esdevé intractable fins i tot per a valors baixos de profunditat, a causa del seu cost computacional elevat: $O(4^{n/2})$ i $O(8^{n/3})$, respectivament. L'algorimse de solucionar els tromins, per una banda, redueix el conjunt n a la meitat en cada cridada recursiva, i llança quatre crides a ell mateix per omlipr els quatre quadrants nous. Per altra banda, l'algorisme de l'estora de Sierpinski, fa tres divisions verticals i altres 3 horizontals, de manera que obtenguim 9 quadrants de la mateixa mida, i el procés es repeteix fins que la profunditat sigui 1.

Finalment, l'**arbre recursiu** té el menor cost computacional, ja que es basa en una representació diferent-dibuixa directament les lines sobre un bufferedImage-, amb una complexitat de $O(2^n)$. Això explica per què els temps esperats són molt inferiors als altres casos, fent que aquest algorisme sigui més escalable per a valors més grans de n.

Per poder ordenar els algorismes, segons el seu cost computacional, calcularem per deu conjunts diferents els valors del costs computacionals respectius dels algorismes.

N	Arbre	Estora	Tromino	Triangle
1	2,00	2,00	2,00	1,73
2	4,00	4,00	4,00	3,00
3	8,00	8,00	8,00	5,20
4	16,00	16,00	16,00	9,00
5	32,00	32,02	32,00	15,59
6	64,00	64,00	64,00	27,00
7	128,00	128,00	128,00	46,77
8	256,00	256,00	256,00	81,00
9	512,00	512,00	512,00	140,30
10	1.024,00	1.024,00	1024,00	243,00

TABLE 1. Comparació dels temps exponencials dels quatre algoritmes

Segons la taula obtinguda, s'observa que l'algorisme per resoldre els arbres recursius, l'estora de Sierpinski i els tromins tenen el mateix cost. En canvi, el solucionador dels triangles de Sierpinski varia una mica i resulta ser més eficient.

Per tant, l'ordenació dels algorismes queda de la següent manera:

$$O(3^{n/2}) < O(4^{n/2}) = O(2^n) = O(8^{n/3})$$

Així té molt de sentit, perquè si desenvolupem les expressions obtenim el següent:

$$8^{(n/3)} = (2^3)^{(n/3)} = 2^n$$

$$4^{(n/2)} = (2^2)^{(n/2)} = 2^n$$

Per últim, donada la rapidesa amb què creix la complexitat d'aquests algorismes, es fa evident que només es poden visualitzar per a valors petits de n abans que el temps d'execució sigui prohibitiu. Tampoc té molt de sentit executar aquests algorismes amb n grans ja que no es podria visualitzar el resultat per falta de resolució.

VI. Conclusions

Aquesta pràctica s'ha dut a terme en aproximadament una setmana. Al començament, es van repartir els rols, però no hi havia una indicació clara sobre com implementar els elements perquè es poguessin comunicar entre ells. A més, la manca de comunicació entre alguns membres de l'equip va provocar que es dupliqués feina en els models, fent que dues persones desenvolupessin diferents implementacions dels models de Tromino i dels triangles de Sierpinski.

Per resoldre aquesta situació, es va convocar una segona reunió on, aquesta vegada, es va definir de manera més clara com s'havien de plantejar els models solvers i com es comunicarien amb el controlador i la vista.

Una de les dificultats principals ha estat la representació dels fractals en una estructura de matriu. Això ha requerit un enfocament específic per garantir que la divisió i la propagació de les subestructures es fes correctament. En el cas del model de l'arbre recursiu, s'ha optat per un altre plantejament: en lloc de treballar directament sobre una matriu, la figura es dibuixa sobre un BufferedImage, i posteriorment s'escala aquesta imatge per adaptar-la a la mida desitjada, com s'ha explicat previament a la memòria.

A més, una de les complexitats més destacades ha estat la implementació del controlador i la gestió de la comunicació entre totes les classes del programa. Ha estat necessari definir un mecanisme eficient perquè els diferents models enviessin la informació correcta a la vista sense generar inconsistències. En particular, la representació de les solucions tant dels triangles de Sierpinski com dels trominos dins una matriu ha suposat un desafiament addicional, ja que s'ha hagut de garantir que cada element quedés correctament posicionat i identificat per ser posteriorment representat gràficament.

Altres característiques que varen presentar un desafiu va ser l'execució concurrent de les cridades recursives. És cert que utilitzar un `ExecutorService` va simplificar molt el problema, però va ser necessària la implementació d'una cua per no sobrecarregar el sistema, tenint molta cura de no provocar cap bloqueig.

Respecte a l'organització del treball, l'equip va repartir les tasques de manera estructura, com indica la següent figura il·lustrativa.

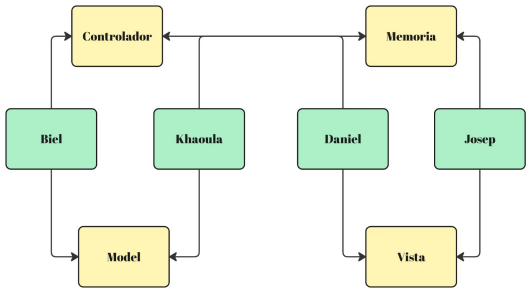

FIGURE 14. Repartiment de feina entre l'equip

Cada parella ha treballat en una part específica de la pràctica. Na Khaoula i en Biel s'han encarregat del Model, en Daniel i en Josep de la Vista, en Daniel i en Biel del Controlador, i na Khaoula i en Josep de la Memòria. A la vida real, així va ser la distribució, tot i que també és cert que els membres de l'equip han realitzat petites modificacions en les parts de la feina dels seus companys.

Pel que fa al feedback sobre aquesta pràctica, la majoria de l'equip ha trobat il·lusionant poder visualitzar el resultat final dels models implementats, fet que també ha motivat a seguir-ne desenvolupant més. A més, per a molts membres, aquesta pràctica ha estat la seva primera experiència programant el dibuix de fractals, cosa que ha fet que el treball tingui una gran utilitat tant a nivell d'aprenentatge com de motivació.

REFERENCES

[1] Oracle, *Full-Screen Exclusive Mode and BufferStrategy*, https://docs.oracle.com/javase/tutorial/extra/fullscreen/bufferstrategy.html, Accedido: 2025-03-25.
[2] Wikipedia. (s.f.). *Fractal*. Recuperat de: https://en.wikipedia.org/wiki/Fractal
[3] GeeksforGeeks. (2021). *Analysis of Algorithms — Big-O Analysis*. Recuperat de: https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/
[4] GeeksforGeeks. (s.f.). *Fractals in C++*. Recuperat de: https://www.geeksforgeeks.org/fractals-in-cc/
[5] Sims, K. (1999). *Julia Sets*. Recuperat de https://www.karlsims.com/julia.html
[6] ScienceDirect. (s.f.). *Iterated Function System*. Recuperat de https://www.sciencedirect.com/topics/mathematics/iterated-function-system#:~:text=Iterated%20function%20systems%20(IFS)%20are,substantial%20industry%20of%20image%20compression.
[7] Wikipedia (s.f). *El easter egg fa referencia a la Triforça de la reconeguda saga de videojocs de Nintendo "The Legend of Zelda"*. Recuperat de https://en.wikipedia.org/wiki/Triforce
[8] Trygve Reenskaug, "The Model-View-Controller (MVC) Pattern," *Software Engineering Notes*, vol. 10, no. 6, pp. 1-3, Dec. 1985. [En línia]. Disponible: https://en.wikipedia.org/wiki/Trygve_Reenskaug.
[9] GeeksforGeeks. (2021). *MVC Design Pattern*. Recuperat de: https://www.geeksforgeeks.org/mvc-design-pattern/
[10] GeeksforGeeks. (2021). *Benefit of Using MVC*. Recuperat de: https://www.geeksforgeeks.org/benefit-of-using-mvc/
[11] D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 3rd ed., Reading, MA: Addison-Wesley, 1997.
[12] Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
[13] Oracle, "Swing: A GUI Widget Toolkit," *Oracle Documentation*, Oracle Corporation, 2009. [En línia]. Disponible: https://docs.oracle.com/javase/8/docs/api/javax/swing/package-summary.html.
[14] Oracle. (2014). *ExecutorService Interface (Java Platform SE 8)*. Recuperat de: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html