Simulation de feu de forêt en C++

SINGARIN-SOLE

March 21, 2024

Contents

1 Introduction

Ce projet se concentre sur la création d'une simulation interactive de feu de forêt, développée en C++ avec l'utilisation de la bibliothèque Qt pour l'interface graphique. Les feux de forêt, en tant que phénomènes naturels complexes, présentent des défis importants en termes de compréhension et de gestion. En simulant leur comportement, nous cherchons à obtenir des connaissances sur la propagation des incendies.

1.1 Objectifs du Projet

1. Modélisation de l'Environnement de la Forêt : Développer une représentation de grille de l'environnement forestier où chaque cellule peut représenter un arbre, un rocher, un lac, ou un état de feu. Cette modélisation vise à imiter la diversité des terrains et des conditions susceptibles d'affecter la propagation d'un feu de forêt.

2. Simulation de la Propagation du Feu : Implémenter un algorithme pour simuler la manière dont le feu se propage d'une cellule à l'autre. Cela inclut la prise en compte des probabilités de propagation et l'effet des différents types de cellules sur la vitesse et la direction de propagation.

3. Interaction Utilisateur : Offrir aux utilisateurs la possibilité d'interagir avec la simulation. Cela peut inclure le déclenchement manuel de feux, la modification des conditions environnementales, et l'utilisation d'outils pour étudier la propagation du feu.

4. Visualisation Intuitive : Utiliser les capacités graphiques de Qt pour créer une interface utilisateur riche et intuitive, permettant une visualisation claire de la simulation en temps réel. Cela aidera à rendre les données complexes accessibles et compréhensibles.

5. Étude des Algorithmes d'Exploration : Expérimenter avec différents algorithmes d'exploration tels que le parcours en largeur et en profondeur, Dijkstra, et A* pour optimiser la recherche du chemin le plus court dans le contexte de la propagation du feu.

2 Implémentation de la grille

Nous avons conçu notre classe 'ForestGrid' de manière à encapsuler les fonctionnalités essentielles de notre simulation de feu de forêt, en adhérant aux principes de la programmation orientée objet (POO). Voici comment nous avons structuré notre code et justifié nos choix :

2.1 Encapsulation et Structure de la Grille

... Nous avons commencé par encapsuler les détails de notre grille forestière dans la classe 'ForestGrid'. Cette classe stocke les dimensions de la grille ('width' et 'height') et les cellules ('grid') elles-mêmes. En rendant ces membres privés, nous avons assuré que l'état interne de notre grille ne peut être modifié que de manière contrôlée à travers les méthodes publiques de la classe, respectant ainsi l'encapsulation.

2.2 Utilisation des Classes 'Config' et 'Cell'

Classe 'Config' : Nous avons choisi d'utiliser une classe distincte 'Config' pour définir les configurations liées aux cellules de notre grille. Elle s'occupe de définir les probabilités et les couleurs pour chaque état possible d'une cellule. Ce choix respecte le principe de responsabilité unique en POO, où chaque classe a une raison claire et unique de changer.

Classe 'Cell' : Chaque cellule de notre grille est représentée par une instance de 'Cell'. Cette classe stocke l'état actuel, la couleur et les dégâts de la cellule, ce qui simplifie la gestion de chaque cellule et sa représentation. Initialisation et Génération Aléatoire des Cellules

Dans notre constructeur 'ForestGrid', nous avons initialisé la grille avec des cellules dont les états sont déterminés aléatoirement. Cela permet de simuler une forêt avec une distribution variée d'arbres, de rochers, de lacs, etc. Nous avons utilisé un générateur de nombres aléatoires pour définir l'état de chaque cellule selon les probabilités spécifiées dans 'Config'. Cette méthode garantit que chaque exécution de notre simulation commence avec une configuration unique de la forêt.

2.3 Mise à Jour de la Grille

La méthode 'update' de 'ForestGrid' est au cœur de notre simulation. Ici, nous utilisons une copie de la grille actuelle pour déterminer l'état futur de chaque cellule sans interférer avec le processus de mise à jour en cours. Cela est particulièrement important pour gérer correctement la propagation du feu d'une cellule à ses voisines. En suivant ce processus, nous assurons une mise à jour cohérente et précise de l'état de la grille à chaque itération de notre simulation.

2.4 Gestion des Voisins

Nous avons également implémenté une méthode pour obtenir les cellules voisines d'une cellule donnée. Cette fonctionnalité est cruciale pour déterminer comment le feu se propage dans notre simulation, en fonction de l'état des cellules voisines.

En respectant ces principes de POO, nous avons créé une base solide pour notre projet de simulation. Notre code est bien organisé, chaque classe a un objectif clair, et la logique de notre simulation est encapsulée de manière appropriée, ce qui rend notre projet à la fois maintenable et extensible.

Bien sûr, lors de la conception de notre projet de simulation de feu de forêt, nous avions plusieurs options pour structurer notre code et implémenter certaines fonctionnalités. Voici comment nous avons abordé nos choix :

1.Gestion de l'État des Cellules : On aurait pu choisir de gérer l'état des cellules directement dans la classe 'ForestGrid', en utilisant un tableau bidimensionnel ou une structure de données similaire. Nous avons opté pour une classe 'Cell' séparée. Cette approche nous a permis de mieux encapsuler les propriétés d'une cellule individuelle, comme son état, sa couleur et d'éventuels dommages. Cela a rendu notre code plus modulaire et a facilité les modifications et l'ajout de nouvelles propriétés à une cellule.

2.Mise à Jour de la Grille : Une autre méthode aurait été d'actualiser la grille en place, sans utiliser de grille clonée. Nous avons décidé de cloner la grille actuelle avant de la mettre à jour. Cette méthode nous permet de prendre des décisions basées sur l'état initial de chaque cellule à chaque itération, ce qui est crucial pour simuler correctement la propagation du feu sans être influencé par les changements d'état qui se produisent pendant le processus de mise à jour.

3.Représentation de la Grille : On aurait pu représenter la grille sous forme d'un tableau à deux dimensions pour une correspondance plus directe avec l'affichage graphique. Nous avons choisi d'utiliser

un vecteur linéaire pour la grille. Cette décision simplifie la gestion de la mémoire et l'accès aux cellules, en transformant simplement les coordonnées bidimensionnelles en un indice linéaire. Cela rend notre code plus concis et réduit la complexité de l'implémentation.

4.Algorithmes de Simulation de Feu: Nous aurions pu implémenter des algorithmes plus complexes ou basés sur des modèles physiques réalistes pour la propagation du feu. Nous avons opté pour un modèle plus simple basé sur des probabilités et des états de cellules. Cette approche garantit que notre simulation reste accessible et compréhensible tout en étant suffisamment flexible pour simuler différents scénarios de feu de forêt.

En résumé, nos choix étaient guidés par le désir de garder notre code modulaire, facile à comprendre et à maintenir. Nous avons cherché à équilibrer la complexité et la fonctionnalité tout en créant une base solide pour des extensions futures.

3 Implémentation de l'interface (window, render_area)

3.1 Initialisation de RenderArea

Dans notre projet, nous avons mis en place une interface utilisateur graphique pour notre simulation de feu de forêt en utilisant Qt, et nous avons implémenté avec succès la classe 'RenderArea'. Notre approche était de créer une fenêtre interactive où l'utilisateur peut visualiser la grille forestière et interagir avec elle. Voici comment nous avons procédé et les raisons derrière nos choix :

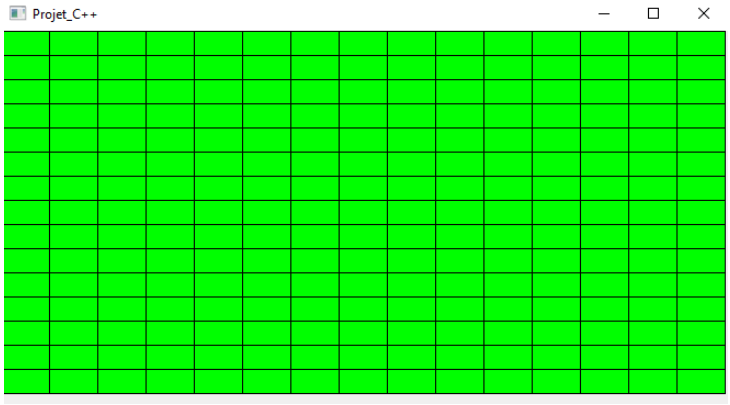

Figure 1: Illustration des arbres

3.2 Initialisation de 'RenderArea'

Dans le constructeur de 'RenderArea', nous avons initialisé notre grille 'ForestGrid' et configuré un 'QTimer'. Le timer est réglé pour déclencher un événement toutes les 5 secondes, ce qui appelle la méthode 'updateGrid'. Cette méthode met à jour l'état de la grille puis redessine la zone de rendu. En faisant cela, nous avons automatisé le processus de mise à jour, permettant à la simulation de se dérouler en temps réel.

3.3 Gestion des Événements de Peinture

Nous avons réimplémenté 'paintEvent' pour dessiner la grille forestière. Cette méthode est appelée chaque fois que Qt détermine qu'une partie du widget doit être redessinée. Ici, nous dessinons chaque cellule de la grille en fonction de son état, en utilisant les couleurs définies dans 'Config'. Cette visualisation rend la simulation intuitive et facile à comprendre.

3.4 Interaction Utilisateur avec la Grille

Nous avons réimplémenté 'mousePressEvent' pour gérer les clics de souris sur la grille. Cette fonctionnalité permet à l'utilisateur d'interagir directement avec la simulation en cliquant sur les cellules. Avec un clic droit, l'utilisateur peut allumer le feu sur une cellule arbre, et avec un clic gauche, l'utilisateur

peut déclencher l'algorithme de recherche de chemin pour trouver le chemin le plus court vers le feu le plus proche.

3.5 Implémentation de l'Algorithme de Recherche de Chemin

Nous avons choisi d'implémenter un algorithme de recherche en largeur pour la méthode 'trouverChemin'. Cet algorithme est relativement simple et efficace pour trouver le chemin le plus court dans un environnement de grille. Lorsque le chemin est trouvé, il est stocké dans 'chemin', et 'paintEvent' le dessine sur la grille. Cependant on a eu des difficultés à implémenter cet algorithme dans notre code.

Lors de la conception de notre interface pour la simulation de feu de forêt, nous avons dû faire des choix stratégiques concernant la structure du code et l'implémentation des fonctionnalités. Voici quelques alternatives que nous aurions pu envisager, et pourquoi nous avons finalement opté pour nos choix actuels :

1.Gestion du Temps de la Simulation :

Nous aurions pu choisir d'utiliser une boucle de mise à jour continue au lieu d'un 'QTimer'. Nous avons opté pour l'utilisation d'un 'QTimer' pour réguler les mises à jour de la grille toutes les 5 secondes. Cette approche offre une gestion du temps plus contrôlée et évite l'utilisation excessive des ressources du système, rendant ainsi notre application plus performante et réactive.

2.Interactivité avec l'Utilisateur :

Une autre option aurait été de créer des boutons ou des contrôles d'interface pour gérer l'interaction, plutôt que d'utiliser des clics de souris directement sur la grille. Nous avons décidé d'implémenter une interaction directe avec la grille via les clics de souris. Cela rend notre interface plus intuitive et immersive, car l'utilisateur peut interagir directement avec les éléments de la simulation.

3.Algorithme de Recherche de Chemin :

L'utilisation d'un algorithme plus complexe comme A* pour la recherche de chemin. Nous avons choisi d'utiliser l'algorithme de recherche en largeur (BFS) pour sa simplicité et son efficacité dans notre contexte de grille uniforme. Cela a rendu l'implémentation plus directe et a suffi pour les besoins de notre simulation.

4.Représentation des Cellules dans la Grille :

Stocker plus d'informations dans chaque cellule, comme son niveau d'humidité ou de combustibilité. Nous avons maintenu une structure de cellule simple, avec l'état, la couleur, et les dégâts. Cela a permis de garder notre modèle de données et notre logique de mise à jour gérables et compréhensibles.

5.Mise en Place de l'Interface Graphique :

Utiliser une bibliothèque graphique 3D pour une représentation plus réaliste. Nous avons opté pour une interface 2D simple en utilisant Qt. Cela a permis de faciliter le développement tout en offrant une représentation claire et suffisante pour les objectifs pédagogiques de notre simulation.

En résumé, nos choix étaient guidés par le désir de créer une simulation à la fois simple, efficace et intuitive. Bien que nous ayons eu de nombreuses options, nous avons cherché à trouver un équilibre entre complexité et facilité d'utilisation, tout en fournissant une expérience éducative et engageante pour l'utilisateur.

4 Génération aléatoire de la grille

4.1 Génération Aléatoire de la Grille

Allons-y pour une explication du code sans le répéter :

4.2 Construction de 'ForestGrid'

Dans le constructeur, chaque cellule de la grille est initialisée avec un état déterminé par des probabilités. Un générateur de nombres aléatoires est utilisé pour sélectionner l'état de chaque cellule en fonction de ces probabilités, qui sont définies dans la classe 'Config'.

4.3 Classe 'Config'

Cette classe contient les configurations de la grille, y compris les probabilités pour chaque état de cellule (arbre, feu, cendre, roche, lac) et leurs couleurs correspondantes. Les méthodes 'proba' et 'ColorState' fournissent ces informations.

Classe 'Cell'

Chaque cellule de la grille a son propre état et couleur, et peut potentiellement contenir d'autres attributs comme les dégâts.

4.4 Mise à Jour et Voisins de la Grille

La méthode 'update' de 'ForestGrid' est conçue pour simuler la propagation du feu. Elle crée d'abord un clone de la grille actuelle pour référence, puis met à jour l'état de chaque cellule en fonction des états de ses cellules voisines. La méthode 'Voisins' aide à déterminer les cellules adjacentes nécessaires pour cette mise à jour.

4.5 Interaction Utilisateur

La classe 'RenderArea', qui hérite de 'QWidget', gère l'affichage de la grille et l'interaction utilisateur. La méthode 'mousePressEvent' permet à l'utilisateur de modifier l'état des cellules en cliquant dessus, déclenchant ainsi de nouvelles dynamiques dans la simulation.

4.6 Dessin de la Grille

'paintEvent' dans 'RenderArea' est responsable de dessiner la grille à l'écran, en représentant visuellement l'état actuel de chaque cellule.

Dans notre projet de simulation de feu de forêt, nous avons pris des décisions importantes concernant la génération aléatoire de notre grille forestière. Notre objectif était de créer un environnement varié et réaliste pour simuler la propagation du feu. Voici les choix que nous avons faits et pourquoi :

4.7 Génération Aléatoire de la Grille

Nous aurions pu initialiser la grille avec un état prédéfini ou un motif fixe, ce qui aurait donné des résultats de simulation uniformes et prévisibles. Nous avons opté pour une génération aléatoire de la grille. En utilisant 'std::random$_device$'et'std :: mt19937'$pourgénérerdesnombresaléatoires, nousavonsattribuéàchaquecelluleunétatba$

4.8 Utilisation des Probabilités pour l'État des Cellules

Une approche alternative aurait été d'utiliser des règles déterministes ou des motifs fixes pour l'attribution des états des cellules. Nous avons décidé d'utiliser un système basé sur des probabilités pour déterminer l'état des cellules. Cela donne à notre simulation une grande variété et un caractère imprévisible, similaire à ce qui peut se produire dans de vrais feux de forêt.

4.9 tructure de la Grille et Accès aux Cellules

Nous aurions pu structurer notre grille d'une manière qui favorise des calculs plus simples, comme un tableau à deux dimensions. Nous avons choisi d'utiliser un vecteur linéaire pour représenter notre grille. En convertissant les coordonnées (x, y) en un seul indice, nous avons simplifié l'accès aux cellules et rendu notre code plus lisible et facile à maintenir.

4.10 Méthode de Mise à Jour

Une mise à jour en temps réel ou en continu de la grille aurait pu être envisagée. Nous avons mis en place un système de mise à jour périodique grâce à 'QTimer'. Cela permet à la grille de se mettre à jour de manière cohérente, avec une propagation du feu qui peut être observée et étudiée au fil du temps.

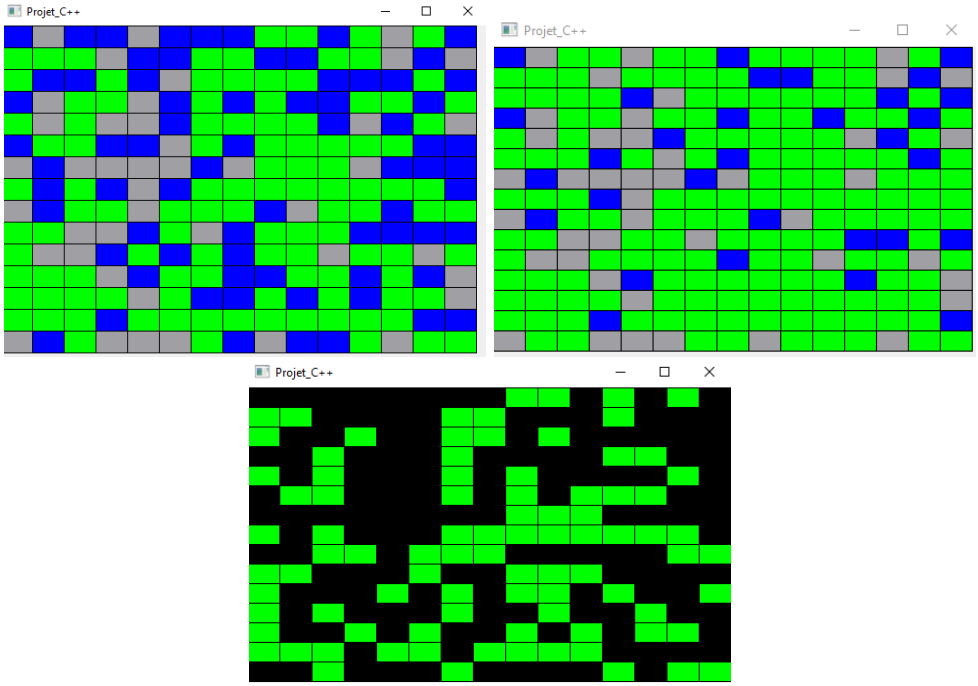

Figure 2: Générations aléatoires de terrain

5 Comportement des cellules

5.1 Configuration et Cellules

Configuration et Cellules

- Implémentation actuelle : Nous avons conçu une classe ‘Config‘ qui contient les configurations pour les différents états des cellules, comme les probabilités et les couleurs. Chaque ‘Cell‘ a un état, une couleur et des dégâts. Cette approche segmente bien les responsabilités : ‘Config‘ gère les configurations globales, et ‘Cell‘ gère les attributs individuels.

- Respect des Principes de la POO : Ici, nous appliquons l’encapsulation, car chaque classe détient et gère ses propres données. De plus, cela suit le principe de responsabilité unique. - Alternatives : On aurait pu intégrer directement les configurations dans ‘ForestGrid‘, mais cela aurait rendu cette classe plus encombrée et moins modulaire.

5.2 Initialisation de la Grille

- Implémentation actuelle : Dans le constructeur de ‘ForestGrid‘, nous initialisons la grille en attribuant à chaque cellule un état basé sur des probabilités. Ceci est réalisé grâce à un générateur de nombres aléatoires. - Respect des Principes de la POO : Cette approche assure que la création et l’initialisation des cellules sont bien encapsulées dans ‘ForestGrid‘. - Alternatives : Une autre méthode aurait été d’avoir une grille pré-définie ou des motifs constants. Toutefois, nous avons choisi l’approche aléatoire pour plus de variété et de réalisme dans la simulation.

5.3 Mise à Jour de la Grille

- Implémentation actuelle : La méthode ‘update‘ crée une copie de la grille actuelle pour déterminer l’état futur des cellules. Cela permet de simuler la propagation du feu sans que les modifications en cours influencent le processus.. - Alternatives : On aurait pu mettre à jour la grille directement sans la cloner, mais cela aurait pu entraîner des problèmes de synchronisation dans la simulation de la propagation du feu.

6 Gestion des Voisins

- Implémentation actuelle : La méthode ‘Voisins‘ détermine les cellules adjacentes nécessaires pour les calculs de propagation. - Alternatives : Plutôt que de calculer les voisins à chaque mise à jour, on aurait pu stocker les voisins de chaque cellule dès l’initialisation, mais cela aurait augmenté la complexité et la consommation de mémoire.

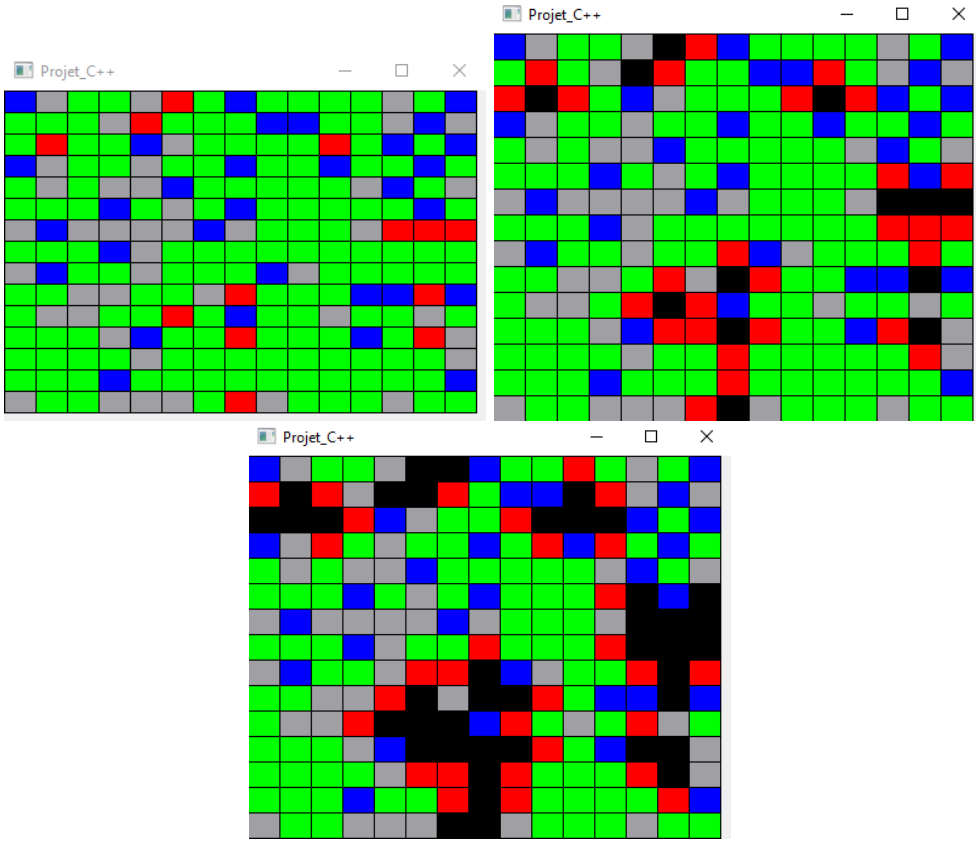

Figure 3: Propagation du feu

7 Implémentation d'une méthode pour allumer le feu

Dans notre projet de simulation de feu de forêt, nous avons conçu la classe 'ForestGrid' pour gérer la logique centrale de la simulation, tout en respectant les principes de la programmation orientée objet (POO). Voici une explication détaillée de notre code et de nos choix de conception :

7.1 Classe 'ForestGrid

- Initialisation de la Grille : Le constructeur de 'ForestGrid' initialise la grille avec des cellules dont les états sont déterminés aléatoirement, en fonction des probabilités définies dans 'Config'. C'est ce qui va permettre de générer le feu automatiquement - Mise à Jour de la Grille : La méthode 'update' gère la logique de mise à jour de notre grille. Elle crée une copie de la grille actuelle (pour maintenir l'immuabilité pendant les calculs) et met à jour les états des cellules en fonction de leurs voisins. - Voisins : La méthode 'Voisins' calcule les cellules voisines pour une cellule donnée, ce qui est crucial pour la logique de propagation du feu.

On a pas réussi à l'implémenter mais l'idée aurait été de pouvoir activer le feu non pas automatiquement mais en faisant un clic gauche sur une cellule contenant un arbre pour allumer le feu.

8 Recherche du chemin le plus court entre un arbre et un feu par parcours en largeur

L'idée était d'utiliser le parcours en largeur pour déterminer le chemin le plus rapide et le plus direct entre une cellule arbre spécifique et la cellule feu la plus proche dans la grille. Voici comment nous avions envisagé de procéder :

8.1 L'Objectif de l'Algorithme

Nous souhaitons que, lorsqu'un utilisateur clique sur une cellule arbre, notre programme puisse calculer et afficher le chemin le plus court menant à la cellule feu la plus proche. Cette fonctionnalité aurait permis aux utilisateurs de visualiser et de comprendre comment le feu pourrait se propager dans des situations réelles.

8.2 Implémentation Envisagée

Notre plan était de traiter chaque cellule comme un nœud dans un graphe, avec des connexions aux cellules adjacentes. Lors de l'exécution de l'algo :

- Nous aurions commencé à partir de la cellule arbre sélectionnée. - Ensuite, nous aurions exploré toutes les cellules voisines, en les ajoutant à une file d'attente pour les traiter. - Nous aurions marqué chaque cellule visitée pour éviter de la revisiter. - Dès que nous aurions trouvé une cellule feu, nous aurions retracé le chemin jusqu'à la cellule arbre de départ, en utilisant une structure de données pour stocker les prédécesseurs de chaque cellule.

8.3 Pourquoi Nous N'avons Pas Réussi à l'Implémenter

Après avoir implémenter l'algo la page se fermait dès qu'on lançait notre programme

Bien que nous n'ayons pas réussi à implémenter cette fonctionnalité, l'intérêt de son ajout aurait été de fournir une visualisation plus profonde de la dynamique du feu dans notre simulation. Cela aurait non seulement augmenté la valeur pédagogique de notre projet, mais aurait également offert une expérience utilisateur plus interactive et informative.

9 Questions

9.1 Question 4 : Utilisation de 'std::map' vs 'std::vector' pour un Tableau 2D

- std::map est préférable lorsqu'on travaille avec un tableau 2D qui est majoritairement vide ou lorsque les éléments sont dispersés irrégulièrement. 'std::map' permet de stocker des éléments de manière plus efficace dans de tels cas, car il ne stocke que les éléments présents, économisant ainsi de l'espace mémoire. - std::vector est optimal pour des tableaux 2D denses où la majorité des éléments sont utilisés. Il offre un accès rapide et séquentiel aux éléments et est plus efficace en termes de performances pour itérer sur des tableaux denses.

9.2 Question 5 : Complexité Algorithmique

La complexité algorithmique est une mesure de la quantité de ressources (temps d'exécution, espace mémoire) qu'un algorithme consomme en fonction de la taille des données d'entrée. Elle est souvent exprimée en termes de : - Complexité Temporelle : Temps nécessaire pour exécuter un algorithme. - Complexité Spatiale : Quantité de mémoire nécessaire pour exécuter l'algorithme.

9.3 Question 6 : Variable et Méthode de Classe Statique

- Variable Statique : Une variable statique est associée à une classe plutôt qu'à des instances de cette classe. Elle conserve sa valeur entre les appels de fonction et est partagée par toutes les instances de la classe. - Méthode de Classe Statique : Une méthode statique est une méthode qui appartient à une classe et non à une instance spécifique de cette classe. Elle peut être appelée sans créer un objet de la classe et ne peut accéder qu'aux membres statiques de la classe.

9.4 Question 7 : Utilisation de Fonctions Récursives

Les fonctions récursives sont des fonctions qui s'appellent elles-mêmes pour résoudre un problème en le décomposant en sous-problèmes plus petits. Dans notre projet, nous avons choisi d'utiliser ou de ne pas utiliser des fonctions récursives en fonction de la situation : - Utilisation : Lorsque le problème peut être naturellement divisé en sous-problèmes similaires, la récursivité peut simplifier le code et le rendre plus

lisible. - Éviter la Récursion : Nous avons évité la récursion pour les tâches nécessitant une efficacité élevée ou pour éviter le risque de dépassement de pile dans les cas de très grandes données d'entrée.

9.5 Question 8 : Garantie de l'Encapsulation dans une Structure

Pour garantir l'encapsulation dans notre structure, voici les mesures que nous avons prises :

- Accès Restreint aux Membres de la Classe : Nous avons utilisé les spécificateurs d'accès 'private' et 'protected' pour cacher les détails internes des classes et empêcher l'accès direct aux données internes depuis l'extérieur de la classe. - Utilisation de Méthodes d'Accès: Des getters et setters ont été implémentés pour contrôler l'accès aux données membres, permettant une validation ou une transformation des données avant qu'elles ne soient affectées ou renvoyées. - Cohérence des Données : En ne permettant l'accès aux données membres que par des méthodes bien définies, nous avons garanti la cohérence de l'état de nos objets.

9.6 Question 9 : Stratégies et Implémentations pour un Concours de Programmation

Pour organiser un concours de programmation à partir de notre plateforme, nous pourrions adopter différentes stratégies ou implémentations :

- Challenges de Programmation : Proposer des défis spécifiques, comme optimiser l'algorithme de propagation du feu ou trouver des chemins efficaces dans des scénarios complexes. Les participants pourraient soumettre leurs solutions, qui seraient ensuite évaluées en fonction de leur efficacité et créativité. - Différentes Implémentations : Encourager les participants à proposer des implémentations alternatives pour des fonctionnalités spécifiques de la simulation, en mettant l'accent sur l'efficacité ou l'originalité de l'approche. - Tournois de Code : Organiser des compétitions où les participants doivent résoudre des problèmes de programmation en temps réel, en utilisant notre plateforme comme base pour les défis.

9.7 Question 10 : Séparation du Modèle, de la Vue, et du Contrôleur

Dans notre projet, nous avons adopté l'architecture Modèle-Vue-Contrôleur (MVC) pour structurer notre application :

- Modèle : La classe 'ForestGrid' et les structures associées représentent le "Modèle". Elles gèrent les données de la simulation, la logique métier et les règles. - Vue : L'interface utilisateur graphique, gérée par la classe 'RenderArea' et d'autres composants de Qt, constitue la "Vue". Elle est responsable de l'affichage des données de la simulation et de l'interaction utilisateur. - Contrôleur : Le lien entre le modèle et la vue est géré par des contrôleurs, qui traitent les entrées de l'utilisateur, manipulent le modèle et mettent à jour la vue.

Cette séparation nous a permis de maintenir notre code organisé, facilitant ainsi la maintenance et l'évolution de notre application.

10 Conclusion

En résumé, notre projet de simulation de feu de forêt, bien que confronté à certains défis, a été une aventure enrichissante dans le monde de la programmation orientée objet avec C++ et Qt. Nous avons développé une simulation dynamique, nous permettant d'explorer des concepts complexes comme la propagation du feu et les interactions entre différents éléments d'une forêt. Même si nous n'avons pas pu implémenter certaines fonctionnalités avancées comme l'algorithme de recherche en largeur, notre travail jusqu'à présent a jeté les bases d'une plateforme évolutive qui pourrait être améliorée et étendue à l'avenir.