RAPPORT DE LABORATOIRE I – CEG 4536

CEG 4536 - Architecture des ordinateurs III

Université d'Ottawa

Professeur : Mohamed Ali

Group : 9

Noms et numéros des étudiants :
Gbegbe Decaho Jacques 300094197
Jean Alexandre Elloh 300211921
Yann Kouadio 300155979
Aziz Tazrout 300266268
Lina Bel Bijou 300158103

Date de soumission: 1 Octobre 2024

LAB1:
Optimisation de la simulation de feux de forêt avec CUDA

Introduction

Ce laboratoire qui fait l'objet de notre étude se porte sur un algorithme de simulation de feux de fortes qui utilise une grille de dimension 1000 x 1000. On observe l'apparition de 100 différents feux dans la forêt. Le programme mis à notre disposition se comporte de manière séquentielle. Il simule la propagation du feu, la combustion des arbres et leur extinction. L'affichage de la grille est géré par la librairie OpenGL avec chaque cellule représente par un espace vide ou un arbre.

Objectif

L'objectif de ce laboratoire est de paralléliser l'algorithme donné en utilisant CUDA C afin d'exploiter la puissance des processeurs graphiques (GPU) et ainsi rendre la simulation plus rapide et plus efficace. Pour ce faire, nous devons identifier les parties du programme où l'optimisation par parallélisation est la plus pertinente, comme la mise à jour de l'état de chaque cellule de la forêt.

Objectif principal

L'objectif principal de ce laboratoire est de transformer le code séquentiel en une version optimisée utilisant CUDA C pour accélérer la simulation. Vous allez apprendre à :

- Identifier les sections du code qui peuvent être parallélisées.
- Utiliser CUDA C pour exécuter des calculs en parallèle sur un GPU.
- Mesurer les gains de performance obtenus grâce à la parallélisation.

3. Plateforme de développement

Le développement et l'optimisation du programme se feront sur des machines équipées de GPU compatibles CUDA. Les outils à utiliser incluent :

- CUDA Toolkit (12.6 ou supérieur) pour la compilation des programmes CUDA.
- Visual Studio 2022 pour l'édition et le débogage du code.
- CUDA Debugger pour tester et profiler vos kernels CUDA.

Vous utiliserez OpenGL pour l'affichage de la simulation, et le travail se fera sur des stations de travail avec des GPU NVIDIA prenant en charge CUDA.

4. Réalisation

a) Analyse du code fourni

Étape 1 : Comprendre le code de départ

- Analysez le code fourni. Il s'agit d'une simulation de feux de forêt où chaque cellule de la grille représente un arbre ou un espace vide. Un feu se déclenche à 100 endroits aléatoires, se propage à des cellules voisines, et les arbres en feu finissent par s'éteindre après un certain temps.
- Étape 2 : Identifier les opportunités de parallélisation
- La mise à jour de la grille est la section du code qui peut être parallélisée de manière significative. Chaque cellule de la grille peut être mise à jour indépendamment des autres.
- Analysez la fonction updateForest() qui est responsable de la mise à jour de l'état des arbres en feu et de la propagation du feu aux cellules voisines. C'est cette partie qui doit être optimisée en utilisant CUDA.

Etape 1: Analyse du code fourni

Ci-dessous, on peut trouver la portion de code chargé de dessiner la simulation attendue de feux de forêt. Elle assigne une couleur aux différents état de la forêt avant, pendant et après le feu tout en ajustant la taille des cellules par rapport à celle de la variable globale N (Taille de la grille).

```
void drawForest() {
    float cellSize = 2.0f / N;  // Taille de chaque cellule ajustée par la taille N // Adjusted cell size based on grid size N

    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            // Choisir la couleur en fonction de l'état de la cellule // Set color based on the state of the cell
            if (forest[i][j] == 0 && burnTime[i][j] == 0) {
                glColor3f(0.8f, 0.8f, 0.8f);  // Espace vide (gris) // Empty space (gray)
            }
            else if (forest[i][j] == 1) {
                glColor3f(0.0f, 1.0f, 0.0f);  // Arbre (vert) // Tree (green)
            }
            else if (forest[i][j] == 2) {
                glColor3f(1.0f, 0.0f, 0.0f);  // Arbre en feu (rouge) // Tree on fire (red)
            }
            else if (forest[i][j] == 3) {
                glColor3f(0.0f, 0.0f, 0.0f);  // Arbre brûlé (noir) // Burned tree (black)
            }

            // Dessiner la cellule // Draw the cell
            float x = -1.0f + j * cellSize;
            float y = -1.0f + i * cellSize;
            glBegin(GL_QUADS);
            glVertex2f(x, y);
            glVertex2f(x + cellSize, y);
            glVertex2f(x + cellSize, y + cellSize);
            glVertex2f(x, y + cellSize);
            glEnd();
        }
    }
}
```

Ci dessous, les variables globales définies au début du programme qui permette de :

-fixer la taille de la grille à 1000
-Donner le temps alloué au feu de forêt
-Donner le nombre d'incendie qui sera déclaré pendant ce temps

```
#define N 1000  // Taille de la grille // Grid size
#define BURN_DURATION 5000  // Durée de combustion d'un arbre en millisecondes (5 secondes) // Tree burning duration in milliseconds (5 seconds)
#define FIRE_START_COUNT 100  // Nombre initial d'incendies // Initial number of fire locations
```

Etape 2 : Identifications des opportunités de parallélisations

A partir du code fourni, nous avons pu repérer les différentes portion de code à paralléliser en utilisant le langage CUDA. On a :

- La propagation du feu.
- L'Initialisation de la forêt.
- Le Rendu ou mise à jour des états des cellules

1- La propagation du feu

En analysant la méthode **updateForest()** qui a pour rôle de mettre à jour l'état de chaque cellule de la forêt, nous avons conclu qu'elle pouvait être parallélisée en allouant une cellule de la grille a chaque thread GPU.
La boucle for présenté dans le code peut être parallélisée afin que chacun des threads puisse gérer une cellule de la grille et détermine si celle-ci est en feu, si elle propage le feu ou si elle est brûlée.

2- L'initialisation de la forêt

La méthode **initializeForest()** peut aussi être parallélisée. En effet, elle initialise la forêt et les arbres puis allume le feu a 100 emplacements différents. On peut utiliser plusieurs thread qui vont s'occuper chacun d'une cellule de la grille.

3- Le rendu ou mise à jour des états des cellules

drawForest() est la méthode qui dessine chaque cellule de la forêt et lui attribue une couleur en fonction de son état. Elle se sert de plusieurs boucles pour le faire. En utilisant un thread GPU pour dessiner une cellule et déterminer la couleur, on peut paralléliser cette méthode.

Étape 3 : Implémenter la parallélisation avec CUDA C

- Initialisation de CUDA : Allouez la mémoire pour la grille (forest) et le temps de combustion (burnTime) sur le GPU à l'aide de cudaMalloc().
- Kernel CUDA : Implémentez un kernel qui met à jour l'état de chaque cellule de la forêt en parallèle.
- Exécution parallèle : Assurez-vous que chaque cellule de la grille est mise à jour en parallèle à l'aide de plusieurs threads sur le GPU.
- Gestion des blocs et des threads : Découpez la grille en blocs de threads CUDA pour une exécution optimisée.

Étape 4 : Mesurer les performances

- Mesurez le temps d'exécution du programme séquentiel et comparez-le à la version optimisée avec CUDA. Utilisez les outils de profilage CUDA pour identifier les gains de performance et toute optimisation supplémentaire possible.

Les étapes suivantes ci-dessous sont issues d' un rapport de profilage généré par NVPROF, un outil de profilage pour les applications CUDA qui exécute des calculs sur GPU.

Une explication détaillée de ce que chaque étape signifie :

1. Démarrage du profilage

==3377== NVPROF is profiling process 3377, command: ./Project3D_init

Elapsed time: 60052.7 ms

NVPROF est en train de profiler le processus avec l'ID 3377, qui correspond à l'exécution du programme ./Project3D_init.

Le temps total écoulé pour exécuter le programme est de 60052.7 millisecondes (environ 60 secondes).

2. Activités GPU

==3377== Profiling result: Type Time(%) Time Calls Avg Min Max Name

Ce tableau indique les activités du GPU et combien de temps chacune a pris. Voici une description des colonnes :

Type : Catégorie de l'activité (dans ce cas, il s'agit d'activités sur le GPU).

Time(%) : Pourcentage de temps total passé dans chaque activité.

Time : Temps total en secondes passé dans chaque activité.

Calls : Le nombre d'appels effectués à cette fonction.

Avg : Temps moyen par appel.

Min et Max : Temps minimum et maximum pour chaque appel.

3. Analyse des activités du GPU

GPU activities: 99.92% 57.8236s 129038 448.11us 439.80us 718.71us updateForestKernel(int*, int*, curandStateXORWOW*)

0.07% 40.938ms 1 40.938ms 40.938ms 40.938ms Forest_init_kernel(int*, int*, __int64)

0.01% 3.5004ms 2 1.7502ms 1.7022ms 1.7982ms [CUDA memcpy DtoH]

0.00% 125.18us 1 125.18us 125.18us 125.18us igniteTrees(int*, int*, int, int, __int64)

updateForestKernel : Cette fonction a consommé 99.92% du temps total d'exécution sur le GPU, soit 57.82 secondes sur 129038 appels. Chaque appel prend en moyenne 448.11 microsecondes.

Forest_init_kernel : Cette fonction a pris 0.07% du temps total, soit 40.938 millisecondes, mais elle n'a été appelée qu'une seule fois.

[CUDA memcpy DtoH] : Cette ligne montre le temps pris pour la copie de données du GPU vers le CPU (Device to Host), qui a pris 3.5004 ms pour 2 appels.

igniteTrees : Cette fonction a pris 125.18 microsecondes pour allumer les arbres au début de la simulation et n'a été appelée qu'une fois.

4. Activités API

API calls: 98.00% 58.9030s 129040 456.47us 5.1790us 40.952ms cudaDeviceSynchronize

1.68% 1.00904s 129040 7.8190us 3.3350us 12.947ms
cudaLaunchKernel

0.31% 186.55ms 2 93.273ms 2.9870us 186.54ms cudaEventCreate

0.01% 6.2236ms 2 3.1118ms 3.0321ms 3.1915ms cudaMemcpy

0.00% 771.57us 3 257.19us 138.85us 428.96us cudaFree

Cette section concerne les appels API CUDA (fonctions qui contrôlent l'exécution CUDA depuis le CPU).

cudaDeviceSynchronize : Cette fonction a pris 98% du temps total des appels API, soit 58.90 secondes. Elle a été appelée 129040 fois. Cette fonction assure que toutes les opérations GPU sont terminées avant de passer à l'étape suivante.

cudaLaunchKernel : A pris 1.68% du temps pour lancer les kernels sur le GPU (envoyer le travail au GPU). Elle a été appelée 129040 fois.

cudaMemcpy : A été utilisée pour copier des données entre l'hôte (CPU) et l'appareil (GPU), prenant 6.22 ms pour 2 appels (principalement la copie des résultats après la simulation).

Résumé

updateForestKernel est la fonction la plus coûteuse en temps, consommant presque tout le temps d'exécution GPU (99.92%). C'est le cœur de la simulation de propagation du feu.

Les appels API montrent que cudaDeviceSynchronize a également pris beaucoup de temps. Cela est normal car cette fonction attend que le GPU termine son travail avant de passer à la suite.

Les copies de mémoire entre le GPU et le CPU (via cudaMemcpy) ont pris un temps relativement court par rapport à l'ensemble de la simulation.

En résumé, la majeure partie du temps est passée à exécuter les kernels sur le GPU pour simuler l'évolution de la forêt en feu, avec un peu de temps pris pour la gestion de la synchronisation et la communication entre le GPU et le CPU.

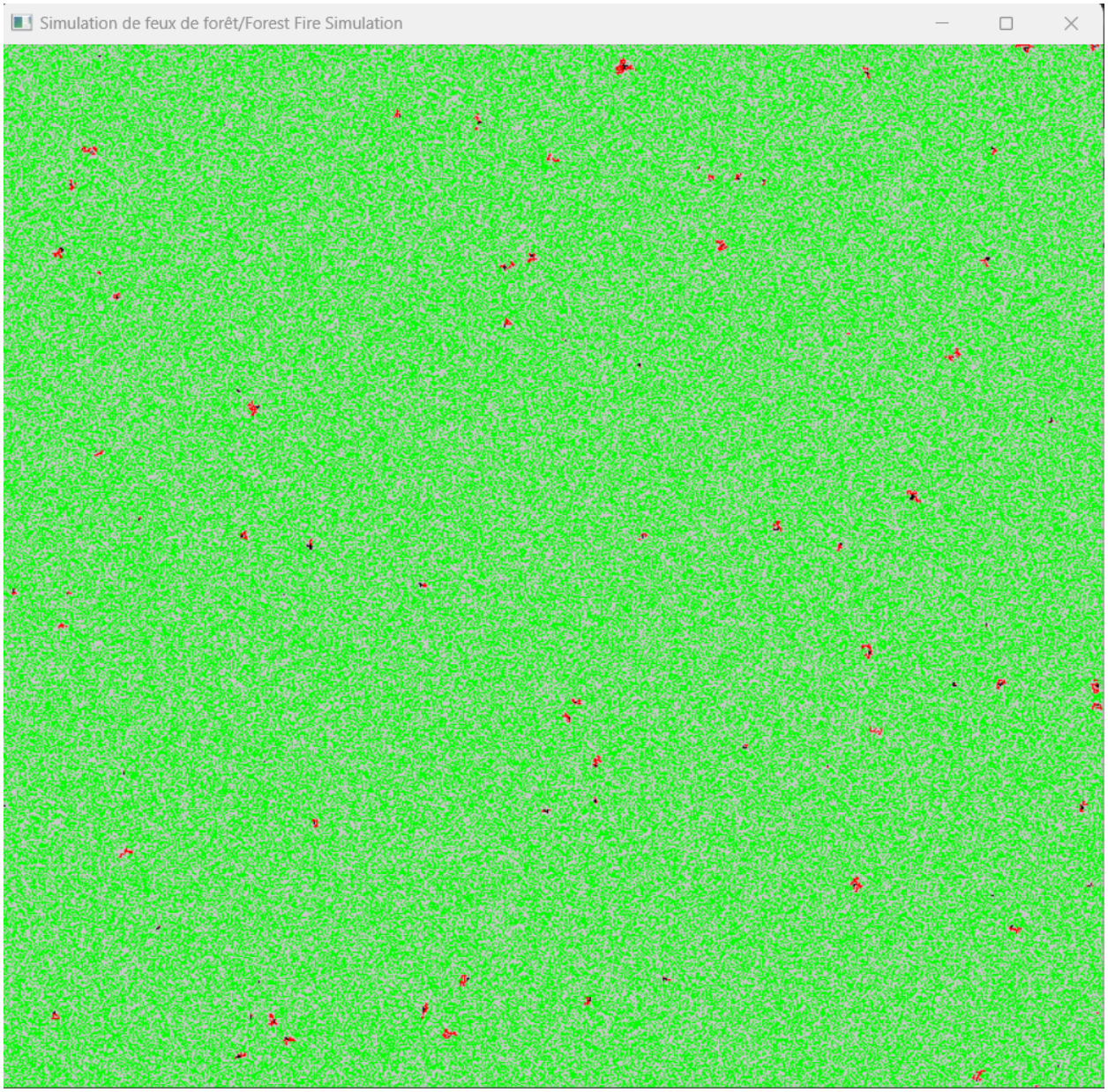
Discussions et problèmes

Nous présentons une simulation d'incendie forestier utilisant CUDA pour le calcul parallèle et OpenGL pour le rendu. La fonction “initializeForest()” crée une grille aléatoire d'arbres, mais nous constatons que l'utilisation de “rand()” pourrait être améliorée par “std::mt19937”. Dans le noyau “updateForestKernel”, nous pensons qu'il serait préférable de déplacer l'initialisation de l'état de “curand” hors du noyau pour éviter des problèmes de synchronisation. Bien que la parallélisation améliore nos performances en permettant des calculs rapides sur le GPU, nous rencontrons des difficultés pour déterminer quelles parties du code exécuté en parallèle, notamment à cause des limitations de Google Colab sans le module freeglut. De plus, nous proposons d'ajouter des vérifications d'erreurs après les appels CUDA, d'intégrer une interface graphique, d'ajouter des statistiques sur l'incendie et de gérer dynamiquement les ressources afin d'enrichir la simulation. Enfin, nous pensons que l'introduction de conditions météorologiques et de tests unitaires améliorerait la robustesse et le réalisme de notre simulation.

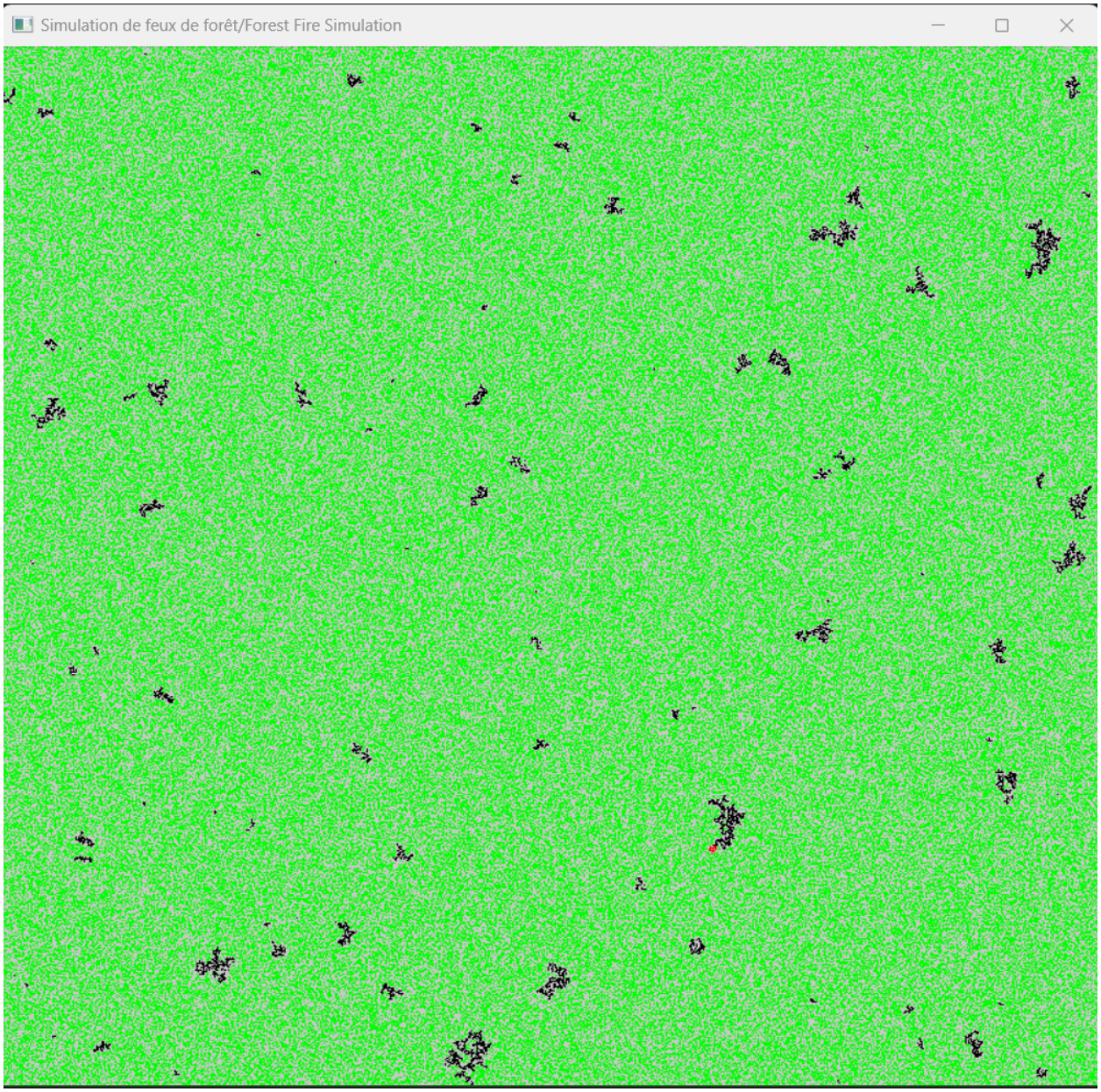
Conclusion

En conclusion, ce laboratoire explore une simulation de feux de forêt sur une grille 1000×1000, avec 100 points de départ pour les incendies. La version initiale du programme est implémentée de manière séquentielle et utilise OpenGL pour représenter graphiquement l'état de chaque cellule de la grille, simulant la propagation du feu, la combustion des arbres et leur extinction. Pour pouvoir exécuter l’ensemble du code en parallèle on a transformé les fonctions tels que initializeForest et updateForest et on a aussi ajouté des fonctions pour gérer les kernels pour gérer différentes opérations à exécuter sur la forêt.

Résultats de la Simulation



État initiale de la forêt avec un debut d’incendie



État finale du feu de forêt avec tentative de propagation de la part sur les voisins

Annexes du code en langage CUDA

kernel.cu

Fichiers divers (Portée globale)

```cpp
#include "device_launch_parameters.h"
#include <algorithm>
#include <stdio.h>
#include <time.h>
#include <vector>
#include <random>
#include <cuda_runtime.h>
#include <curand_kernel.h>
#include <fstream>
#include <iostream>
#include <chrono>
#include <stdlib.h>
#include <GL/glut.h>
#include <curand_kernel.h>

using namespace std;

#define N 1000  // Taille de la grille
#define BURN_DURATION 5000  // Durée de combustion d'un arbre en millisecondes (5 secondes)
#define FIRE_START_COUNT 100  // Nombre initial d'incendies

// Utilisation de vecteurs pour gérer la mémoire
std::vector<std::vector<int>> forest(N, std::vector<int>(N, 0));
std::vector<std::vector<int>> burnTime(N, std::vector<int>(N, 0));

int simulationDuration = 60000;  // Durée de la simulation (60 secondes)
int startTime = 0;  // Temps de départ en millisecondes
int elapsedTime = 0;  // Temps écoulé
float spreadProbability = 0.3f;  // Probabilité que le feu se propage à un arbre voisin

bool isPaused = false;  // Indicateur de pause
int pauseStartTime = 0;  // Temps de début de la pause

float zoomLevel = 1.0f;  // Niveau de zoom
float offsetX = 0.0f, offsetY = 0.0f;  // Décalage horizontal et vertical pour le déplacement
float moveSpeed = 0.05f;  // Vitesse de déplacement de la vue

bool dragging = false;  // Indicateur de glisser-déposer avec la souris
int lastMouseX, lastMouseY;  // Dernière position de la souris lors du clic

// Fonction pour initialiser la forêt et démarrer les incendies
void initializeForest() {
    // Génère aléatoirement des arbres (1) ou des cellules vides (0) pour chaque case de la forêt
    for (auto &row : forest) {
        std::generate(row.begin(), row.end(), []() { return rand() % 2; });
    }

    // Initialise le temps de combustion à 0 pour toutes les cellules
    std::fill(burnTime.begin(), burnTime.end(), std::vector<int>(N, 0));

```

kernel.cu

Fichiers divers (Portée globale)

```
    // Récupère toutes les positions des arbres disponibles (valeur 1)
    std::vector<std::pair<int, int>> availablePositions;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if (forest[i][j] == 1) availablePositions.emplace_back(i, j);
        }
    }

    // Mélange les positions disponibles pour sélectionner des arbres aléatoires pour démarrer le feu
    std::shuffle(availablePositions.begin(), availablePositions.end(), std::mt19937(std::random_device{}()));

    // Sélectionne un certain nombre d'arbres pour allumer le feu
    for (int fire = 0; fire < FIRE_START_COUNT && fire < availablePositions.size(); fire++) {
        int fireX = availablePositions[fire].first;
        int fireY = availablePositions[fire].second;
        forest[fireX][fireY] = 2; // 2 représente un arbre en feu
        burnTime[fireX][fireY] = BURN_DURATION; // Temps de combustion initial
    }

    // Initialise le temps du début de la simulation
    startTime = glutGet(GLUT_ELAPSED_TIME);
    elapsedTime = 0;
    isPaused = false;
}


// Fonction pour initialiser les paramètres OpenGL
void initGL() {
    glClearColor(1.0, 1.0, 1.0, 1.0);  // Définit le fond blanc
    glEnable(GL_DEPTH_TEST);  // Active le test de profondeur pour un rendu 3D correct
}

// Fonction pour dessiner la forêt
void drawForest() {
    float cellSize = 2.0f / N;  // Taille de chaque cellule dans la fenêtre OpenGL

    // Fonction lambda pour définir la couleur selon l'état de la cellule
    auto setColor = [](int state) {
        switch (state) {
            case 0: glColor3f(0.8f, 0.8f, 0.8f); break; // Cellule vide (gris)
            case 1: glColor3f(0.0f, 1.0f, 0.0f); break; // Arbre (vert)
            case 2: glColor3f(1.0f, 0.0f, 0.0f); break; // Feu (rouge)
            case 3: glColor3f(0.0f, 0.0f, 0.0f); break; // Cendres (noir)
        }
    };
```

kernel.cu

Fichiers divers (Portée globale)

```
    // Parcourt chaque cellule pour dessiner la forêt
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            setColor(forest[i][j]);  // Détermine la couleur en fonction de l'état
            float x = -1.0f + j * cellSize, y = -1.0f + i * cellSize;
            glBegin(GL_QUADS); // Dessine chaque cellule comme un carré
            glVertex2f(x, y); glVertex2f(x + cellSize, y);
            glVertex2f(x + cellSize, y + cellSize); glVertex2f(x, y + cellSize);
            glEnd();
        }
    }
}

// Kernel CUDA pour mettre à jour l'état de la forêt
__global__ void updateForestKernel(int* forest, int* burnTime, float spreadProbability, int A, int seed) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x + (blockIdx.y * blockDim.y + threadIdx.y) * A;
    curandState state; curand_init(seed, idx, 0, &state);

    // Si l'arbre est en feu et que le temps de combustion est écoulé, il devient des cendres
    if (forest[idx] == 2 && --burnTime[idx] <= 0) forest[idx] = 3;
    // Sinon, on propage le feu aux arbres voisins avec une probabilité donnée
    else if (forest[idx] == 2) {
        auto spreadFire = [&](int i, int j) {
            int nIdx = i * A + j;
            if (forest[nIdx] == 1 && curand_uniform(&state) < spreadProbability) {
                forest[nIdx] = 2; burnTime[nIdx] = BURN_DURATION;
            }
        };
        int i = idx / A, j = idx % A;
        if (i > 0) spreadFire(i - 1, j);     // Propage le feu vers le haut
        if (i < A - 1) spreadFire(i + 1, j); // Propage le feu vers le bas
        if (j > 0) spreadFire(i, j - 1);     // Propage le feu à gauche
        if (j < A - 1) spreadFire(i, j + 1); // Propage le feu à droite
    }
}

// Fonction principale pour lancer la mise à jour de la forêt avec CUDA
void updateForestCUDA() {
    int* d_forest, * d_burnTime;

    // Allocation de mémoire sur le GPU pour la forêt et le temps de combustion
    cudaMalloc((void**)&d_forest, N * N * sizeof(int));
    cudaMalloc((void**)&d_burnTime, N * N * sizeof(int));

    // Copie les données de la forêt et du temps de combustion depuis l'hôte (CPU) vers le GPU
    cudaMemcpy(d_forest, forest[0].data(), N * N * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_burnTime, burnTime[0].data(), N * N * sizeof(int), cudaMemcpyHostToDevice);

    // Définition de la grille et des blocs pour le lancement du kernel CUDA
    dim3 threadsPerBlock(16, 16), blocksPerGrid((N + 15) / 16, (N + 15) / 16);
    updateForestKernel <<<blocksPerGrid, threadsPerBlock>>>(d_forest, d_burnTime, spreadProbability, N, time(NULL));
```

kernel.cu

Fichiers divers (Portée globale)

```
149         // Copie des données du GPU vers l'hôte après l'exécution du kernel
150         cudaMemcpy(forest[0].data(), d_forest, N * N * sizeof(int), cudaMemcpyDeviceToHost);
151         cudaMemcpy(burnTime[0].data(), d_burnTime, N * N * sizeof(int), cudaMemcpyDeviceToHost);
152
153         // Libération de la mémoire sur le GPU
154         cudaFree(d_forest);
155         cudaFree(d_burnTime);
156     }
157
158     // Fonction d'affichage principale
159     void display() {
160         glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Efface la fenêtre
161         glPushMatrix(); // Sauvegarde la matrice actuelle
162         glScalef(zoomLevel, zoomLevel, 1.0f); // Applique le zoom
163         glTranslatef(offsetX, offsetY, 0.0f); // Applique le déplacement
164         drawForest(); // Dessine la forêt
165         glPopMatrix(); // Restaure la matrice
166         glutSwapBuffers(); // Échange les buffers pour l'affichage
167     }
168
169     // Fonction de mise à jour en continu (idle)
170     void idle() {
171         if (!isPaused && elapsedTime < simulationDuration) {
172             elapsedTime = glutGet(GLUT_ELAPSED_TIME) - startTime; // Met à jour le temps écoulé
173             updateForestCUDA(); // Met à jour l'état de la forêt avec CUDA
174             glutPostRedisplay(); // Redessine la fenêtre après la mise à jour
175         }
176     }
177
178     // Fonction de gestion des entrées clavier
179     void keyboard(unsigned char key, int, int) {
180         switch (key) {
181             case 'r': initializeForest(); break; // Réinitialise la simulation
182             case 'p': isPaused = !isPaused;       // Met en pause ou relance la simulation
183                       if (isPaused) pauseStartTime = glutGet(GLUT_ELAPSED_TIME);
184                       else startTime += glutGet(GLUT_ELAPSED_TIME) - pauseStartTime;
185                       break;
186             case '+': zoomLevel *= 1.1f; break;  // Zoom avant
187             case '-': zoomLevel *= 0.9f; break;  // Zoom arrière
188             case 'w': offsetY -= moveSpeed; break; // Déplacement vers le haut
189             case 's': offsetY += moveSpeed; break; // Déplacement vers le bas
190             case 'a': offsetX += moveSpeed; break; // Déplacement vers la gauche
191             case 'd': offsetX -= moveSpeed; break; // Déplacement vers la droite
192         }
193         glutPostRedisplay(); // Redessine la fenêtre après une action
194     }
195
196     // Fonction de gestion des clics de souris
197     void mouse(int button, int state, int x, int y) {
198         if (button == GLUT_LEFT_BUTTON) dragging = (state == GLUT_DOWN); // Active le mode "drag" avec le bouton gauche
199         if (dragging) lastMouseX = x, lastMouseY = y; // Enregistre la position initiale de la souris
200     }
```

kernel.cu

Fichiers divers (Portée globale)

```
203     void motion(int x, int y) {
204         if (!dragging) return;
205         offsetX += (x - lastMouseX) * 0.01f; // Déplace la vue en fonction du mouvement horizontal
206         offsetY -= (y - lastMouseY) * 0.01f; // Déplace la vue en fonction du mouvement vertical
207         lastMouseX = x, lastMouseY = y;      // Met à jour la dernière position de la souris
208         glutPostRedisplay();                 // Redessine la fenêtre après le mouvement
209     }
210
211     // Fonction principale du programme
212     int main(int argc, char** argv) {
213         // Initialisation de GLUT (bibliothèque pour la gestion de fenêtres et OpenGL)
214         glutInit(&argc, argv);
215         glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH); // Mode d'affichage en double buffer, RGB et test de profondeur
216         glutInitWindowSize(800, 800); // Définit la taille initiale de la fenêtre
217         glutCreateWindow("Simulation d'Incendie Forestier"); // Crée la fenêtre avec un titre
218
219
220         initGL();// Initialisation des paramètres OpenGL
221         initializeForest();// Initialise la forêt et démarre les feux
222
223         glutDisplayFunc(display);// Fonction d'affichage appelée à chaque redessin de la fenêtre
224
225         glutIdleFunc(idle);// Fonction appelée lorsque l'ordinateur est en mode inactif (utilisée pour la mise à jour continue)
226         glutKeyboardFunc(keyboard);// Fonction appelée lors des événements clavier (comme les touches 'r', 'p', etc.)
227         glutMouseFunc(mouse);    // Fonction appelée lors des événements de la souris (comme les clics)
228         glutMotionFunc(motion);    // Fonction appelée lors du mouvement de la souris lorsque l'on fait glisser
229         glutMainLoop();    // Lance la boucle principale de GLUT (infinie) qui gère les événements et redessine la fenêtre
230
231         return 0; // Retourne 0 lorsque le programme se termine (ce qui n'arrive pas avec glutMainLoop)
232     }
233
234
```