Automates cellulaires et feu de forêt

Nous proposons d'utiliser le modèle des *Automates Cellulaires*[1] afin simuler une propagation de feu de forêt sur une grille bidimensionnelle – soit une image.

1 Principe

Il s'agit de considérer l'image comme une représentation, en vue de haut, d'un terrain comprenant des zones plus ou moins denses de végétation. Afin de simplifier ce modèle, notre configuration d'automates cellulaires est décrite de la manière suivante :

- Le terrain est représenté par une image de dimensions $W \times H$;
- Chaque pixel (x, y) de l'image est une position sur ce terrain, appelée cellule, sa couleur indique (reflète) un état. Nous proposons les états suivants pour les cellules de notre grille 2D :
  - Etat **vide** : absence de végétation (par exemple pas d'**arbre**) dans cette cellule, nous affectons la couleur **marron** ;
  - Etat **arbre** : présence d'arbre(s) dans cette cellule, nous affectons la couleur **verte** ;
  - Etat **feu** : présence d'arbre(s) en feu dans cette cellule, nous affectons la couleur **rouge** ;
  - Etat **calciné** : présence d'arbre(s) calciné(s) dans cette cellule, nous affectons la couleur **noir** ;

Pour finir de définir le fonctionnement de notre automate cellulaire il nous reste à spécifier deux choses :

1. Nous nous intéressons plus particulièrement ici au modèle appelé *Le Jeu de la Vie*. Pour plus d'informations concernant les automates cellulaires, nous vous invitons à consulter la page WIKIPÉDIA à l'adresse :
`https://fr.wikipedia.org/wiki/Automate_cellulaire`

1. Quelles sont les règles de changement d'état d'une cellule ? Ces règles s'appliquent au sein d'un mécanisme *tour par tour*. Nous choisissons une situation simple – l'automate à états finis [2] donné en Figure 1 résume le comportement de chaque type de cellule, parfois en fonction de son voisinage – qui pourra être modifiée afin d'obtenir des résultats différents :
   (a) L'état **vide** reste toujours **vide** ;
   (b) Une cellule à l'état **arbre** passe à l'état **feu** si elle est en *contact* (voir les conditions de *contact* plus bas) avec une cellule dont l'état est **feu** ;
   (c) Une cellule à l'état **feu** passe à l'état **calciné** ;
   (d) L'état **calciné** reste toujours **calciné** ;
2. Définir les conditions de *contact*, c'est-à-dire quand considère-t-on que deux cellules sont en contact. Pour se faire nous proposons deux stratégies reprisent par les deux configurations présentées en Figure 2 :
   (a) La 4-connexité : une cellule est connectée avec celles se trouvant juste à sa droite (1), en haut (2), à sa gauche (3) et en bas (4). Notons que dans certaines configurations une ou plusieurs des voisines n'existent pas ; par exemple pour une cellule se trouvant sur le bord gauche de l'image, cette dernière n'aura pas de voisine à gauche (la 3), ou encore une cellule se trouvant dans le coin bas-droit de l'image n'aura ni voisine à doite (la 1) ni en bas (la 4) ;
   (b) La 8-connexité : une cellule est connectée avec celles se trouvant juste à sa droite (1) – **Est**, en haut à droite (2) – **Nord-Est**, en haut (3) – **Nord**, en haut à gauche (4) – **Nord-Ouest**, à sa gauche (5) – **Ouest**, en bas à gauche (6) – **Sud-Ouest**, en bas (7) – **Sud** – et en bas à droite (8) – **Sud-Est**. Notons que dans certaines configurations une ou plusieurs des voisines n'existent pas ; par exemple pour une cellule se trouvant sur le bord gauche de l'image, cette dernière n'aura pas de voisines **Nord-Ouest** (la 4), **Ouest** (la 5) et **Sud-Ouest** (la 6).

La progression du feu forêt est ainsi conditionnée par les états et conditions de changement d'état décrites par l'automate à états finis implémenté (cf. Figure 1) mais aussi et plus simplement par la configuration de connexité

2. Si vous souhaitez en savoir plus sur les automates à états finis, nous vous invitons à consulter cette page WIKIPÉDIA :
`http://fr.wikipedia.org/wiki/Automate_fini`

choisie. La Figure 2 en illustre deux mais rien ne nous empêche d'en imaginer d'autres[3], voire d'utiliser des configurations différentes selon l'état de la cellule ou faire varier localement ou globalement cette configuration en fonction de paramètres plus globaux. Nous pouvons notamment penser à un paramètre simulant la direction et l'intensité du vent et qui de manière probabiliste modifierait localement et ponctuellement la connexité.

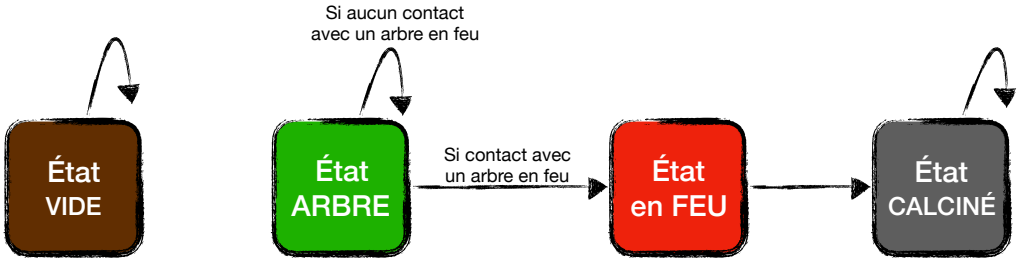

FIGURE 1 – Automate à états finis représentant le comportement basique de notre automate cellulaire simulant un feu de forêt.

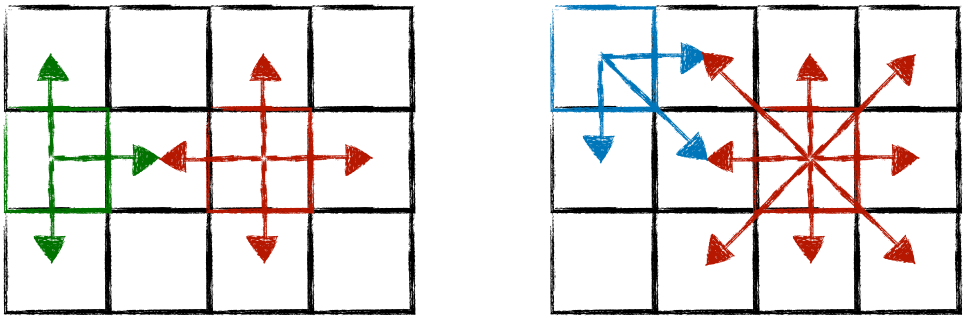

FIGURE 2 – Possibilités de configurations déterminant un *contact* entre une cellule et ses voisines.

Revenons à une situation plus basique illustrée par la Figure 3 où les règles citées plus haut sont utilisées sur une carte de 7×7 cellules ne contenant que des arbres et où nous mettons le feu à l'abre central. La série du haut montre la progression sur 3 étapes du feu de forêt appliqué à une configuration en

3. Un exemple simple à réaliser serait de décider d'avoir plusieurs graduations de l'état **feu** afin d'augmenter légèrement sa durée de vie, voire son intensité, donc sa capacité de propagation.

4-connexité tandis que la série du bas montre cette même progression quand une 8-connexité est appliquée.

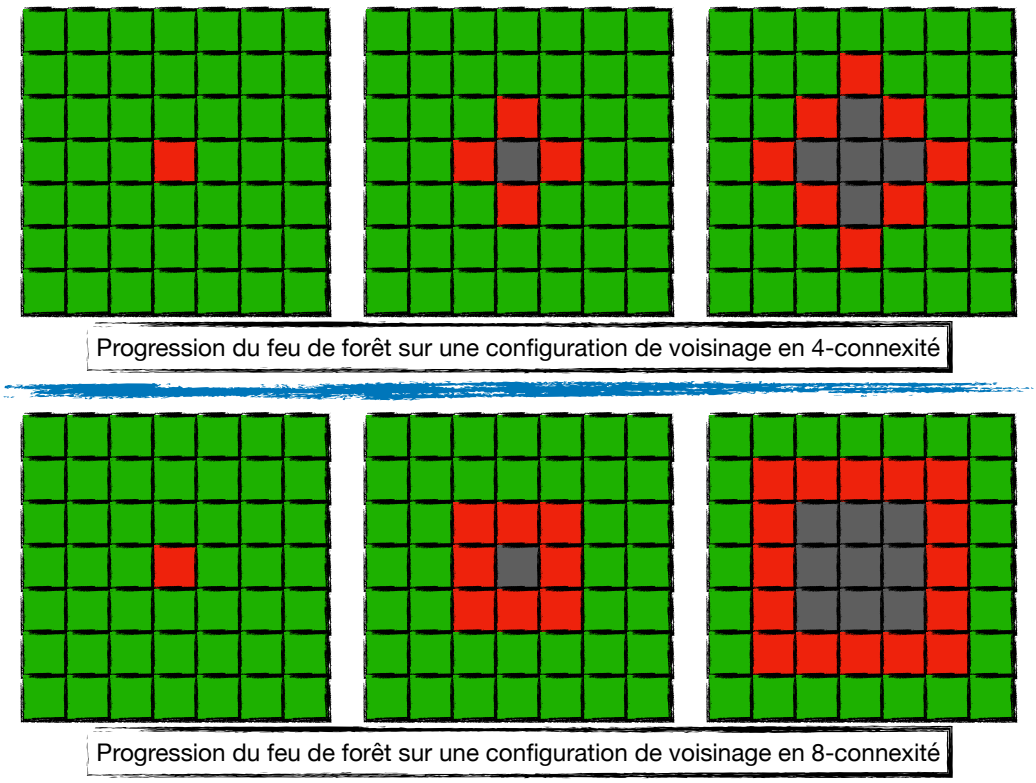

FIGURE 3 – Évolution en 3 étapes d'un feu de forêt partant du centre de la carte : en haut, une configuration en 4-connexité est utilisée, en bas, la 8-connexité est appliquée.

2 Implémentation

Pour l'implémentation, nous proposons de partir de l'exemple de code `sc_00_03_feu2foret-0.1.tgz` et de l'étoffer progressivement avec les éléments de code donnés ci-après jusqu'à arriver au résultat souhaité. L'archive du code de départ est accessible à cette adresse :

`https://code.up8.edu/amsi/capg/-/tree/main/02_DM_feu2foret/sc_00_03_feu2foret-0.1.tgz`

Ainsi, en tête du fichier `window.c`, nous ajoutons les inclusions suivantes qui seront utiles aux nouvelles fonctionnalités :

```
#include <GL4D/gl4dm.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
```

`GL4D/gl4dm.h` est ajouté pour pouvoir utiliser la fonction `gl4dmURand`[4] qui permet de retourner un nombre pseudo-aléatoire dans la fourchette de "réels"[5] $[0\,;1[$ (*i.e.* incluant tous les nombres de 0 à presque 1, donc 1 strictement est exlu), ici les tirages sont uniformes[6] et permettrons de disposer au hasard et uniformément des arbres sur la carte.

Le fichier `stdlib.h` – pour bibliothèque standard du langage `C` – est ajouté pour l'utilisation des deux fonctions `void * malloc(size_t size)` et `void free(void *ptr)` permettant respectivement d'allouer dynamiquement de la mémoire de taille `size` octets et de la libérer en passant en paramètre l'adresse de cette mémoire allouée.

Le fichier `string.h` – accès à des fonctions de gestion de chaînes de caractères en langage `C` – est ajouté car nous ferons appel à la fonction `void * memcpy(void *destination, const void *source, size_t n_octets)` pour copier une zone mémoire de `n_octets` depuis l'adresse – « en lecture seulement » – `source` vers l'adresse `destination`.

Enfin, le fichier `assert.h` – une assertion est une proposition censée être vraie – est ajouté pour l'usage de la *macro* `assert(expression)` déclarant que l'expression passée en paramètre est forcément vraie. La macro `assert` court-circuite le programme s'il s'avère que l'expression est fausse, le fichier et la ligne où se trouvent cette assertion sont imprimés sur la sortie standard d'erreurs. Les assertions sont utilisées en *débuggage*, elles sont censées être désactivées dans la version *release* de l'application concernée.

4. Voir la documentation de référence de *GL4Dummies* :
`http://gl4d.api8.fr/doxygen/html/index.html`
5. En pratique c'est des nombres flottants de précision *double*.
6. Un tirage uniforme est une sélection aléatoire d'un élément parmi N où chaque élément a une probabilité $\frac{1}{N}$ d'être tiré.

Étant donné que la carte (*i.e.* l'image ou le *screen*) que nous utiliserons contiendra uniquement des cellules (*i.e.* des pixels) qui correspondent à l'automate cellulaire du *Feu de Forêt*, nous proposons de créer une énumération [7] afin de restreindre et nommer les états possibles pour nos cellules. Ajoutons cet `enum` juste après l'inclusion des fichiers en-tête (*i.e.* les *headers*) :

```
enum ca_states_t {
  EMPTY = RGB(96, 46, 8),
  TREE = RGB(41, 175, 29),
  FIRE = RGB(235, 38, 31),
  CALCINED = RGB(94, 94, 94)
};
```

Notez que les valeurs des constantes de l'énumération sont explicités à des valeurs correspondant à des couleurs codées sur un entier (voir la documentation de référence de *GL4Dummies*, fichier `gl4dp.h`, macro `RGB`).

Ainsi, nous commençons par ajouter une fonction `initForest` qui permet d'initialiser le *screen* courant, qui joue le rôle de la carte, afin d'y déposer

7. Une énumeration est un type de données contenant un nombre fini de valeurs ; ces valeurs prennent la forme de noms et éventuellement d'une valeur précise liée à ces noms. Ce type de données est souvent utilisé pour stocker un nombre d'états fixes pour une variable de ce type. Par exemple, nous pouvons créer une énumération `matiere` qui va stocker les états de la matière et nous écrirons :

```
enum matiere {
   solide,
   liquide,
   gazeux
};
```

Ici les trois constantes `solide`, `liquide` et `gazeux` prennent respectivement les valeurs 0, 1 et 2. Si nous modifions l'énumération de la manière suivante :

```
enum matiere {
   solide,
   liquide = 5,
   gazeux
};
```

ces mêmes trois constantes prendront respectivement les valeurs 0, 5 et 6 ; donc pour les valeurs suivantes non-définies, un incrément automatique est utilisé à partir de la dernière valeur explicitement définie pour déterminer la leur (toujours 0 pour la première si elle n'est pas explicitement définie). Enfin, il est possible d'expliciter toutes les valeurs des constantes de l'énumération :

```
enum matiere {
   solide = 25,
   liquide = 5,
   gazeux = 10
};
```

nous aurions donc respectivement 25, 5 et 10. Notez que si `liquide` n'était pas explicité à 5, il y aurait la valeur 26 pour cette constante.

des arbres (écrire la constante `TREE` dans le pixel) ou non (écrire la constante `EMPTY` dans le pixel). L'idéal est que cette fonction prenne un paramètre indiquant la densité en arbre de la forêt. Voici donc une implémentation possible :

```
static void init_forest(double density) {
  GLuint i;
  GLuint wh = gl4dpGetWidth() * gl4dpGetHeight();
  /* le pointeur vers le début de mes pixels est "la forêt" */
  GLuint * forest = gl4dpGetPixels();
  /* parcours direct, pas besoin d'abscisse-ordonnée */
  for(i = 0; i < wh; ++i)
    forest[i] = (gl4dmURand() < density) ? TREE : EMPTY;
}
```

Ici `density` représente la densité en arbres de la forêt, cette valeur doit être prise dans l'intervalle $[0\,;1]$. Par exemple si `density` reçoit `0.8` l'idée est que la forêt soit couverte à 80% d'arbres. La variable `wh` (pour $width \times height$) stocke le nombre de pixels dans la carte en récupérant le produit de la largeur du *screen*, renvoyée par `gl4dpGetWidth()`, et de sa hauteur, renvoyée par `gl4dpGetHeight()`. Aussi, `forest` stocke le pointeur (*i.e.* un `GLuint *`) vers la mémoire de pixels, soit chaque pixel `forest[i]` est de type `GLuint` qui est un entier non-signé sur 32 bits.

Ainsi, en parcourant l'ensemble des pixels de cette *carte*, chaque pixel `forest[i]` a une probabilité `density` de recevoir la couleur d'un arbre (*i.e.* la valeur constante `TREE`), soit d'être dans l'état arbre. La dernière ligne [8] de cette fonction se lit donc :

> **Si** un tirage aléatoire dans $[0\,;1[$ est strictement inférieur à `density`
> **alors** la valeur renvoyée dans `forest[i]` est `TREE`,
> **sinon** c'est `EMPTY`.

À ce stade si nous utilisons cette fonction (+ les *includes* et l'énumération donnés plus haut) après ajout dans `sc_00_03_feu2foret-0.1.tgz`, il sera nécessaire d'ajouter un appel à `init_forest` dans le `main`. En voici un exemple (les «...» indiquent que la partie à cet emplacement reste inchangée) qui créé une forêt à 65% de densité en arbres et dont le résultat est illustré dans la Figure 4 :

8. Cette ligne de code contient un test en ligne ; une sorte de *if/else* en une seule ligne et qui renvoie deux valeurs possibles selon un test. Le test en ligne se fait à l'aide des deux opérateurs «`?`» et «`:`». Il se résume à :
`v = (expression_à_tester) ? valeur_renvoyée_si_vrai : valeur_renvoyée_sinon ;`


```c
int main(int argc, char ** argv) {
  const int w = 256, h = 192;
  ...

  gl4dpInitScreenWithDimensions(w, h); /* donc juste après cette ligne */
  init_forest(0.65);
  /* ! et surtout supprimez la ligne qui clear le screen ! */

  ...
}
```

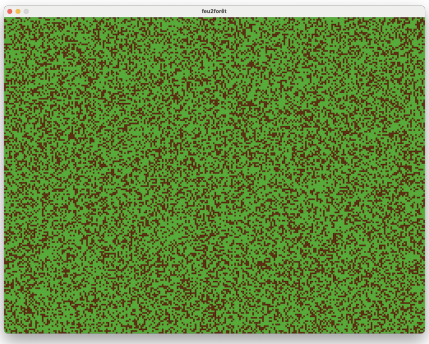

FIGURE 4 – Initialisation de la forêt à 65% de densité en arbres.

À ce stade, il est préférable de tester les premiers éléments donnés avant d'aller plus loin, faites tous les ajouts [9] et modifications cités plus haut, compilez puis exécutez le code afin vérifier que vous obtenez bien un résultat équivalent.

Avant d'aborder la partie simulation de l'automate cellulaire *Feu de Forêt*, il reste un petit détail à régler : *allumer le feu* à quelques arbres ! Pour se faire, nous proposons de séparer l'allumage de l'initialisation dans une fonction `light_the_fire` que nous appellerons juste après `init_forest` dans la fonction `main` :

```c
static void light_the_fire(void) {
  GLuint i;
  GLuint w = gl4dpGetWidth();
  GLuint wh = w * gl4dpGetHeight();
  GLuint * forest = gl4dpGetPixels();
  for(i = w - 1; i < wh; i += w)
    forest[i] = (forest[i] == TREE) ? FIRE : forest[i];
}
```

9. Attention au copier/coller, il ne produit pas toujours un résultat fidèle, soyez donc attentif · ves et utilisez un éditeur qui *sait* indenter. Aussi, pensez à prototyper, en haut du fichier `C` les nouvelles fonctions ajoutées.

Ici les variables utilisées sont quasiment les mêmes que dans `init_forest`, nous aurons juste besoin de la largeur seule stockée dans `w`. Il s'agit d'un «allumage» sur le bord droit de la carte, `w` nous sert à passer d'une ligne à la suivante. Ainsi, la variable d'itération `i` démarre au pixel `w - 1` (soit le pixel se trouvant en $(w-1, 0)$, en bas à droite) et passe, d'une itération à l'autre, au pixel juste en haut à l'aide de l'instruction `i += w`. Pour tous les pixels de la première colonne, parcourus de bas en haut, si la valeur en ce pixel est un arbre (`forest[i] == TREE`) alors y mettre le feu, sinon lui renvoyer sa propre valeur (pour le laisser inchangé).

Tout est fin prêt pour mettre en place la simulation. Cette dernière peut prendre place dans la fonction `dessine` mais il est toujours préférable de séparer les simulations des *draw* ou *display*. Comme vu précédemment, *GL4Dummies* propose une fonction *callBack* permettant de *setter* une fonction à appeler pour réaliser du calcul avant d'appeler la *callBack* de *draw* (ici `dessine` qui reste strictement inchangée) ; il s'agit du *setter* `gl4duwIdleFunc`. Ce dernier prend en paramètre une fonction renvoyant `void` et prenant `void` paramètre (soit aucun paramètre). Nous proposons d'appeler notre fonction simulant l'automate `cellular_automaton_step`, il sera donc nécessaire de la *setter* comme *idle function* dans le `main` avant la boucle principale *event-simu-draw*. Donc finalement, les principaux ajouts dans la fonction `main` se résument à :

```
int main(int argc, char ** argv) {
  const int w = 256, h = 192;
  ...

  gl4dpInitScreenWithDimensions(w, h); /* donc juste après cette ligne */
  initForest(0.65);
  light_the_fire();
  gl4duwIdleFunc(cellular_automaton_step);
  /* ! et comme plus haut, surtout supprimez la ligne qui clear le screen ! */

  ...
}
```

La fonction `cellular_automaton_step` se charge donc de simuler un seul passage d'étape de l'ensemble des cellules, car elle est appelée à chaque cycle et que nous souhaitons voir l'évolution de la simulation à chaque *frame*. Nous en proposons l'implémentation ci-après :

```
static void cellular_automaton_step(void) {
  GLuint i;
  GLuint w = gl4dpGetWidth();
  GLuint h = gl4dpGetHeight();
  GLuint wh = w * h;
  GLuint * forest = gl4dpGetPixels();
  GLuint * forest_copy = malloc( wh * sizeof *forest_copy );
  assert(forest_copy);
  memcpy(forest_copy, forest, wh * sizeof *forest_copy);
  /* tout est prêt pour lancer l'automate sur l'ensemble des cellules */
  for(i = 0; i < wh; ++i) {
    switch (forest[i]) {
    case EMPTY: /* rien à faire, ou */
    case CALCINED: /* rien à faire aussi */
      break;
    case FIRE: /* il a brulé => devient calciné */
      forest[i] = CALCINED;
      break;
    case TREE: /* regardons les voisins */
      {
        /* récupérons l'abscisse et l'ordonnée de la cellule à l'indice i */
        int x = i % w, y = i / w; /* à retrouver dans le premier support de
            cours (intro) */
        /* l'abscisse et l'ordonnée du voisin (neighbor), un j pour les
            parcourir */
        int nx, ny, j;
        /* les 4 directions pour le voisinage en 4-connexité */
        const int d[][2] = { /* est   */ { 1,  0},
                             /* nord  */ { 0,  1},
                             /* ouest */ {-1,  0},
                             /* sud   */ { 0, -1} };
        for(j = 0; j < 4; ++j) {
          nx = x + d[j][0];
          ny = y + d[j][1];
          if( _in_screen(nx, ny, w, h) /* pour ne pas sortir de la forest */ &&
              /* ET ... */
              forest_copy[ny * w + nx] == FIRE /* ... ce voisin <était> en feu
                  */ ) {
            forest[i] = FIRE; /* la cellule i prend feu */
            break; /* plus besoin de chercher plus loin, il sera en feu qq soit
                le reste */
          }
        }
      }
      break;
    }
  }
  free(forest_copy);
  gl4dpScreenHasChanged();
}
```

L'élément clé de ce type de traitement est de comprendre la nécessité d'avoir l'ensemble des états à l'instant t afin de pouvoir calculer ceux de l'instant $t+1$. Il faut donc, avant toute modification, procéder en premier lieu à une sauvegarde des états de la carte, produits lors de l'étape précédente. En pratique, ceci se résume à faire une copie de la carte (*i.e.* du *screen*) afin de l'utiliser pour effectuer les tests provoquant éventuellement un changement d'état. Ainsi, une mémoire de la même taille que le *screen*

est dynamiquement allouée à l'aide de la fonction `malloc` et son adresse est stockée dans la variable `forest_copy`. La macro `assert` permet de garantir que l'allocation s'est bien déroulée (*i.e.* en vérifiant que le pointeur n'est pas `NULL`, donc différent de zéro) avant de poursuivre en copiant, à l'aide de la fonction `memcpy`, le contenu du *screen* pointé par `forest` vers la nouvelle zone mémoire pointée par `forest_copy`.

À ce stade, tous les éléments dont nous avons besoin pour éventuellement procéder aux changements d'états des cellules sont disponibles. Nous parcourons donc l'ensemble des cellules de la carte et testons pour chacune son état « avant modification » à l'aide d'un `switch` ; l'utilisation de `forest[i]` ou de `forest_copy[i]` dans ce contexte ne change rien au résultat car nous ne modifions potentiellement `forest[i]` que dans le cadre de ce test sur `forest[i]` lui même (*i.e.* il n'y a pas de problème de mélange de temporalités, elle serait bien modifiée à la suite d'un test sur elle même, ce qui n'est pas le cas si on faisait un test au niveau d'un voisin). Ce `switch` possède trois cas simples : vide (*i.e.* `EMPTY`) ou calcinée (*i.e.* `CALCINED`) qui restent toujours inchangés ; en feu (*i.e.* `FIRE`) qui passe automatiquement à l'état calciné et enfin arbre (*i.e.* `TREE`) dont le changement d'état dépend de son voisinage. Dans ce dernier cas, nous commençons par convertir l'indice `i` en un couple de coordonnées `(x, y)` qui nous permettra, à l'aide d'un tableau de quatre directions, de calculer les coordonnées `(nx, ny)` des quatre voisins en 4-connexité. Ces coordonnées sont testées afin de vérifier s'il y a un dépassement du cadre de la carte (*i.e.* cas des bords et des coins). Il suffira de trouver un seul voisin pour lequel, à l'étape précédente, `forest_copy` indique un état en feu (*i.e.* `FIRE`) pour que la cellule `forest[i]` passe aussi à cet état ; le `break` dans cette boucle sur le voisinage indique que nous n'avons plus besoin de chercher plus.

En sortie de la boucle parcourant l'ensemble des cellules, nous n'avons plus besoin de la sauvegade des états précédents des cellules. Il est donc nécessaire, afin d'éviter toutes *fuites* mémoire, de libérer la mémoire pointée par `forest_copy` à l'aide de la fonction `free`.

Enfin, étant donné que nous aurons potentiellement modifié la carte, qui pour rappel est le *screen* stocké en RAM-CPU, en utilisant directement le pointeur `forest`, il est nécessaire d'indiquer à *GL4Dummies* que le *screen* a changé pour que, plus tard, lors de l'appel à la *callBack* `dessine`, `gl4dpUpdateScreen` déclenche une mise à jour de l'affichage, soit un transfert de RAM-CPU vers RAM-GPU. En l'absence d'une déclaration de changements, `gl4dpUpdateScreen` ne ferait rien car elle estimerait qu'aucune mise à jour n'est nécessaire, d'où la présence de l'appel à `gl4dpScreenHasChanged()`

qui joue le rôle d'indicateur de changement. Il est à noter que cet appel n'est pas nécessaire quand nous utilisons les primtives de dessin telles que gl4dpPutPixel, gl4dpLine, ... qui déclarent un changement en leur sein.

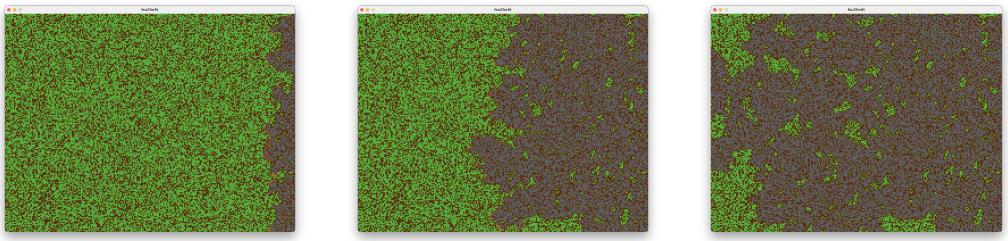

FIGURE 5 – Progression d'un feu de forêt 4-connexité sur une carte à 61% de densité en arbres.

La Figure 5 montre au travers de trois «moments» consécutifs de la simulation la progression du feu de forêt sur une carte où la densité en arbres a été fixée à 61% ; une grande partie de la «forêt» passe à l'état calciné.

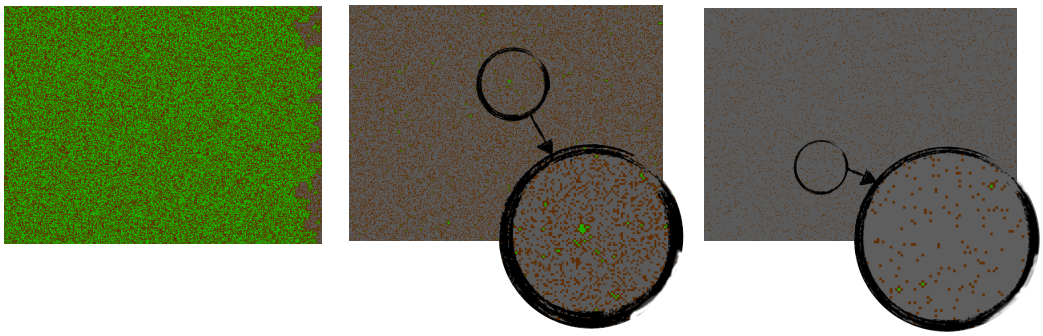

FIGURE 6 – Diverses situations terminales d'un feu de forêt 4-connexité. De gauche à droite, les densités en arbres sont respectivement : 55%, 75% et 95%.

La Figure 6 illustre plusieurs configurations de situations finales obtenues avec des densités différentes en arbre. Notez qu'à 55% le feu a du mal à se propager, qu'à 61% (cf. Figure 5) il arrive relativement fréquemment à calciner une grande partie de la «forêt», et qu'à partir de 75% il ne reste que très peu d'arbres non calcinés. À 95% de densité, il intéressant d'observer la configuration isolée de quelques arbres, en pratique souvent en 0 et 5, qui ont «survécu».

3 Exercice

Il serait intéressant d'utiliser une image[10], par exemple en niveaux de gris[11] où pour laquelle nous aurons extrait la luminance en chaque pixel, pour pondérer la probabilité liée à la densité en arbre. Ainsi, pour chaque pixel, plus un niveau de gris est clair – ou une luminance de RGB est élevée – plus la probabilié que la cellule correspondante en (x, y) soit un arbre. En utilisant une couleur pour l'état `EMPY` proche de celle de `TREE`, le «feu de forêt» fonctionnerait comme un révélateur d'une certaine trame de l'image initialement utilisée. Enfin, dans certains cas, il serait utile de modifier la connexité pour passer à 8 et ainsi améliorer la probabilité de propagation.

La Figure 7 montre ce que peut donner une telle implémentation : l'image de gauche est utilisée pour calculer la densité initiale de la forêt et l'image de droite le résultat final une fois la propagation terminée.

Inspirez-vous du code `https://code.up8.edu/amsi/capg/-/tree/main/01_GL4D_beginners/sc_00_01_load_image_and_binarize-1.0.tgz` qui binarise une image. En modifiant légèrement la fonction qui copie l'image BMP chargée en la binarisant dans le screen, vous pouvez faire en sorte d'écrire une nouvelle version de l'initialisation (exemple `init_forest_from_image` qui utilise la luminance et du pseudo-aléatoire pour déterminer si le pixel est `TREE` ou `EMPTY`.

FIGURE 7 – Utilisation d'une image comme condition pour une densité en arbres : plusieurs étapes de la simulation. Ici `TREE` et `EMPTY` ont quasiement la même couleur, jaune.

10. Dans le sens dessin ou photographie.
11. La transformation peut aussi se faire en calculant une luminance à partir des trois composantes d'une image RGB.

