Simuler les feux de forêt

Comment utiliser l'informatique pour réduire l'impact des feux de forêts en transformant le moins possible ces dernières ?

N° SCEI 14423

Victor Sarrazin

Contexte

Les feux de forêt sont de plus en plus fréquents. L'informatique peut se révéler être un atout de taille pour prévoir et anticiper ces derniers.

Figure: Feu de forêt à Malibu[1]

[1] National Geographic Education

Sommaire

Choix d’implémentation

Langage : `C99`

Motivations :

- Bas niveau → Gain temps exécution (calculs importants)
- Écosystème de bibliothèques vaste et mature (vs. `OCaml`)

Bibliothèques externes :

- `SDL2` : Interface graphique (visualisation en temps réel)
- `cJSON` : Manipulation de `JSON` (configurations initiales)
- png : Création/manipulation de PNG (export des résultats)

Outil de configuration :

- Site web interactif (`React.JS`) pour la conception des forêts
- Export au format `JSON` (pour la simulation en C)

Un premier modèle de feux de forêt

Automate cellulaire (2D)

- Une grille
- Un état par case
- Un ensemble de règles de transitions entre les états

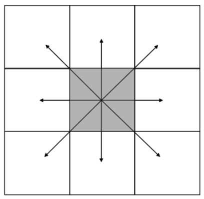


Figure: Voisinage de Moore [2]

[2] Science Direct

Un premier modèle de feux de forêt

Types de cases :

- Arbres
- Champs
- Feu
- Case brulée *
- Eau *

* Ne peuvent pas/plus bruler

p_b	Voisin direct	Voisin diagonal
Arbres	$\frac{1}{8}$	$\frac{1}{16}$
Champs	$\frac{1}{8}$	$\frac{1}{16}$

Figure: Probabilité de changement d'état

Un premier modèle de feux de forêt

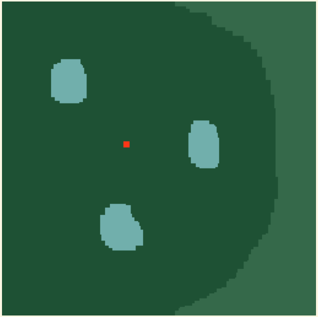


Figure: À $t = 0$

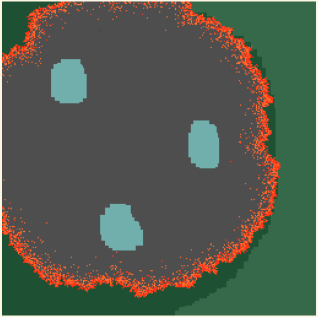


Figure: À $t = 300$

Vers le modèle d’Alexandridis

> ***Idée***
> Il serait intéressant de prendre en compte des données du milieu : vent, densité de végétation

Modèle d'Alexandridis pour les feux de forêt

Nouveau type de case :

- Arbres denses

Règles de transition

Pour tout $(i, j, t) \in \mathbb{N}^3$, on a :

- Si $m_{i,j}(t) = \texttt{feu}$ alors $m_{i,j}(t+1) = \texttt{brulé}$
- Si $m_{i,j}(t) = \texttt{feu}$ alors $m_{i\pm1,j\pm1}(t+1) = \texttt{feu}$ avec une probabilité p_b
- Si $m_{i,j}(t) = \texttt{brulé}$ alors $m_{i,j}(t+1) = \texttt{brulé}$

Probabilité d'inflammation p_b

On a $p_b = p_h(1 + p_{veg})(1 + p_{den})p_{vent}$ avec $p_h = 0.27$ une constante

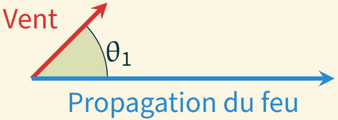
Plus la végétation est dense, plus p_{den} est élevée Plus la végétation a du combustible, plus p_{veg} est élevée

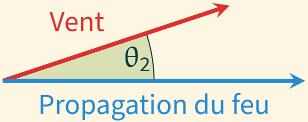
	p_{veg}	p_{den}
Arbres	0.3	0
Arbres denses	0.3	0.3
Champs	-0.1	0

Figure: Probabilités p_{veg} et p_{den} selon le type de végétation

Probabilité liée au vent p_{vent}

On a $p_{vent} = exp(0.045v) \times exp(0.131v \times (cos(\theta) - 1))$ avec θ l'angle entre la propagation du feu et la direction du vent et v la vitesse du vent (en m/s)





Comparaison du premier modèle ($t = 300$) et de celui d'Alexandridis ($t = 150$).

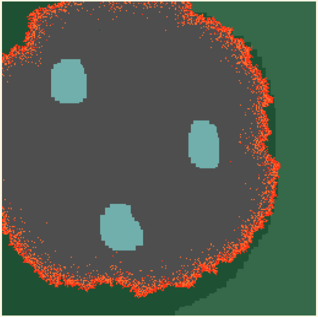


Figure: Modèle 1

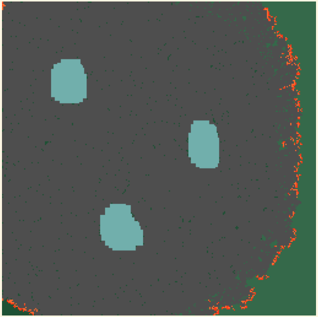


Figure: Modèle 2

Modèle d’Alexandridis pour les feux de forêt

Comparaison selon la densité de végétation avec 15 m/s de vent vers l’est.

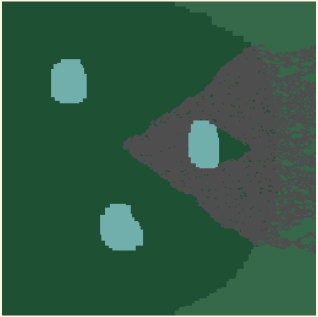


Figure: Végétation normale

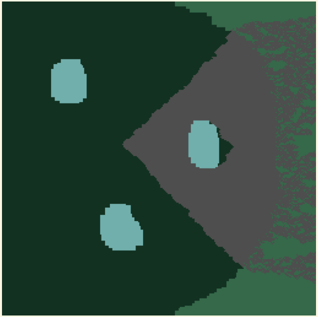


Figure: Végétation dense

Objectif

Mieux protéger la forêt contre les incendies en la modifiant le moins possible.

Figure: Chemin forestier - Forêt du Rouvray [3]

[3] ONF

Transformations envisageables

Nouveau type de case :

- Chemins/Tranchées avec $p_{veg} = -0.55$ et $p_{den} = 0$

Figure: Exemple de chemins

Transformations envisageables

Comparaison entre une forêt avec une tranchée et une sans

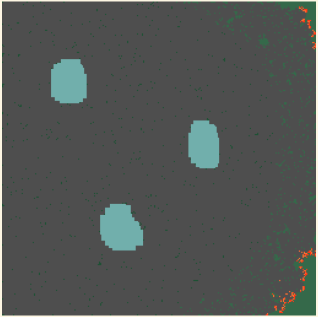


Figure: Sans tranchées

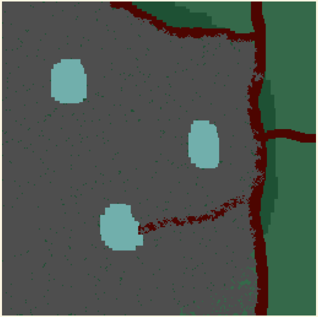


Figure: Avec tranchées

Comparaison entre une forêt avec une tranchée et 15 ou 30 m/s de vent

Figure: 15 m/s de vent

Figure: 30 m/s de vent

Transformations envisageables

Comparaison entre deux tranchées de largeurs différentes

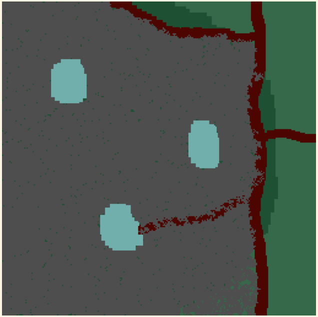


Figure: Tranchée de 8m

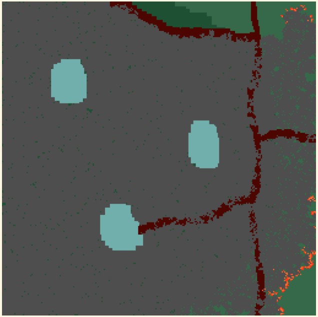


Figure: Tranchée de 4m

Conclusion

Le modèle d'Alexandridis permet de visualiser l'effet de tranchées contre la propagation du feu, donc de réduire l'impact des incendies

Possibilité de prendre en compte pour améliorer les simulations :

- Humidité
- Altitude/pentes
- Température

D'autres transformations sont envisageables :

- Lacs/Cours d'eau
- Réduction de la densité de végétation

Navigateur

Diapo 1
Diapo 2
Diapo 3
Diapo 4
Diapo 5
Diapo 6
Diapo 7
Diapo 8
Diapo 9
Diapo 10
Diapo 11
Diapo 12
Diapo 13
Diapo 14
Diapo 15
Diapo 16
Diapo 17
Diapo 18

Annexe : Génération de grilles

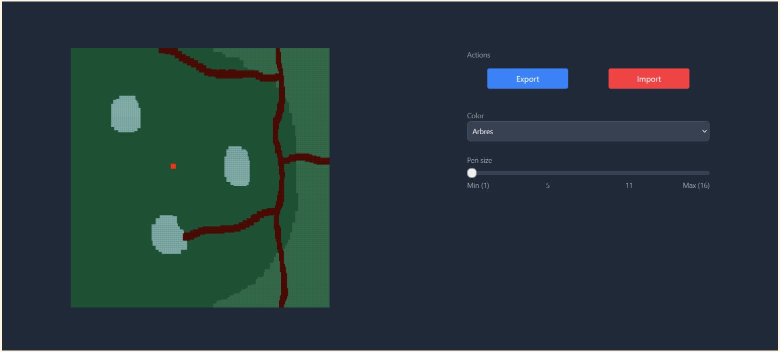


Figure: Site de génération de grilles

main.c (1/10)

```c
#include <time.h>
#include "grid.c"

/**
 * Main function of the program
 * <p>
 * The program can be launched with the following arguments:
 * <ul>
 * <li>--model [model]: The model of the grid (0-2)</li>
 * <li>--count [count]: The number of grids to simulate</li>
 * <li>--iterations [iterations]: The max number of iterations</li>
 * <li>--enable_graphics [0/1]: Whether graphics are disabled</li>
 * <li>--tick [ms]: The number of milliseconds between each tick</li>
 * <li>--export_csv: Export grids in csv format</li>
 * <li>--export_png: Export grids in png format</li>
 * <li>--wind_direction [direction]: The wind direction (0 to 360)</li>
 * <li>--wind_speed [speed]: The wind speed</li>
 * <li>--generate_mean: Generate the mean of the grids (useful only if you export the
 ↪  grids)</li>
 * <li>--help: Display the help message</li>
 * </ul>
 * </p>
 * @param argc The number of arguments
 * @param argv The arguments
 * @return The exit code
 */
int main(int argc, char * argv[]) {
  // Command arguments management
```

main.c (2/10)

```
int model = 0;
int count = 1;
int iterations = -1;
int tick_ms = 10;
bool enable_graphics = true;
bool export_csv = false;
bool export_png = false;
double wind_direction = 0;
double wind_speed = 0;
bool generate_mean = false;

if (argc > 1) {
  for (int i = 1; i < argc; i++) {
    if (strcmp(argv[i], "--model") == 0) {
      if (i + 1 < argc) {
        model = atoi(argv[i + 1]);
      }
    } else if (strcmp(argv[i], "--count") == 0) {
      if (i + 1 < argc) {
        count = atoi(argv[i + 1]);
      }
    } else if (strcmp(argv[i], "--iterations") == 0) {
      if (i + 1 < argc) {
        iterations = atoi(argv[i + 1]);
      }
    } else if (strcmp(argv[i], "--tick") == 0) {
      if (i + 1 < argc) {
        tick_ms = atoi(argv[i + 1]);
      }
```

```c
} else if (strcmp(argv[i], "--enable_graphics") == 0) {
  if (i + 1 < argc) {
    enable_graphics = atoi(argv[i + 1]);
  }
} else if (strcmp(argv[i], "--help") == 0) {
  printf("Usage: %s --model [model] --count [count] --iterations [iterations]
  ↪  --enable_graphics [0/1] --tick [ms] --export_png --export_csv
  ↪  --wind_direction [direction] --wind_speed [speed] --generate_mean
  ↪  --help\n\nArguments:\n--model [model]: The model of the grid
  ↪  (0-2)\n--count [count]: The number of grids to simulate\n--iterations
  ↪  [iterations]: The max number of iterations\n--enable_graphics [0/1]:
  ↪  Whether graphics are disabled\n--tick [ms]: The number of milliseconds
  ↪  between each tick\n--help: Display this help message\n--export_csv:
  ↪  Export grids in csv format\n--export_png: Export grids in png
  ↪  format\n--wind_direction [direction]: The wind direction (0 to
  ↪  360)\n--wind_speed [speed]: The wind speed\n--generate_mean: Generate the
  ↪  mean of the grids (useful only if you export the grids)\n--help: Display
  ↪  the help message\n",
       argv[0]);
  return 0;
} else if (strcmp(argv[i], "--export_csv") == 0) {
  export_csv = true;
} else if (strcmp(argv[i], "--export_png") == 0) {
  export_png = true;
} else if (strcmp(argv[i], "--wind_direction") == 0) {
  if (i + 1 < argc) {
    wind_direction = atof(argv[i + 1]);
  }
} else if (strcmp(argv[i], "--wind_speed") == 0) {
```

```c
      if (i + 1 < argc) {
        wind_speed = atof(argv[i + 1]);
      }
    } else if (strcmp(argv[i], "--generate_mean") == 0) {
      generate_mean = true;
    }
  }
}

printf("Launching simulation\nModel %d\nCount %d\nIterations %d\nGraphics %d\n",
↪  model, count, iterations,
     enable_graphics);

srandom(time(NULL));

if (tick_ms < 2) {
  printf("Invalid tick, setting to 2\n");
  tick_ms = 2;
}

if (count <= 0) {
  printf("Invalid count, setting to 1\n");
  count = 1;
}

int remaining = count;
Grid * grids = malloc(count * sizeof(*grids));

// Choose the number of grids to display per line and per column
```

main.c (5/10)

```c
int max_x;
int max_y;
if (count == 1) {
  max_x = 1;
  max_y = 1;
} else if (count <= 2) {
  max_x = 2;
  max_y = 1;
} else if (count <= 6) {
  max_x = 3;
  max_y = 2;
} else if (count <= 18) {
  max_x = 6;
  max_y = 3;
} else if (count <= 36) {
  max_x = 9;
  max_y = 4;
} else {
  max_x = 14;
  max_y = 7;
}

remove("grids.csv");
remove("grids_png");

// Create the window and the grids
Window window;
if (enable_graphics) {
  window = create_window(max_x, max_y);
```

main.c (6/10)

```c
} else {
  window = (Window) {
      .window = NULL,
      .surface = NULL
  };
}

for (int i = 0; i < count; i++) {
  grids[i] = create_grid(model, window, i % max_x, i / max_x, export_csv,
  ↪  export_png);
  grids[i].wind_direction = wind_direction;
  grids[i].wind_speed = wind_speed;
}

// Main loop to update the grids and tick until all grids have ended
do {
  SDL_Event event;
  while (SDL_PollEvent(&event)) {
    // Used to close the window if the user clicks on the close button
    if (event.type == SDL_QUIT) {
      for (int i = 0; i < count; i++) {
        destroy_grid(grids[i]);
      }

      free(grids);
      return 0;
    }
  }
```

main.c (7/10)

```c
    // Update the grids
    for (int i = 0; i < count; i++) {
      if (!grids[i].ended) {
        tick(&grids[i]);

        grids[i].ended = is_ended(grids[i]);
        // If the grid has ended, we decrease the number of remaining grids
        if (grids[i].ended) {
          remaining--;
        }
      }
    }

    wait(tick_ms);
  } while (remaining > 0 && --iterations != -1);

  wait(2500);

  // DO SOMETHING WITH GRIDS IF NEEDED
  if (generate_mean) {
    int *** data = malloc(GRID_SIZE * sizeof(*data));
    for (int i = 0; i < GRID_SIZE; i++) {
      data[i] = (int **) malloc(GRID_SIZE * sizeof(*data[i]));
      for (int j = 0; j < GRID_SIZE; j++) {
        data[i][j] = malloc(TILE_TYPE_SIZE * sizeof(*data[i][j]));
      }
    }

    for (int i = 0; i < count; i++) {
```

main.c (8/10)

```
      for (int x = 0; x < GRID_SIZE; x++) {
        for (int y = 0; y < GRID_SIZE; y++) {
          data[x][y][grids[i].data[x][y].current_type]++;
        }
      }
    }

    Tile ** tiles = malloc(GRID_SIZE * sizeof(*tiles));

    for (int x = 0; x < GRID_SIZE; x++) {
      tiles[x] = malloc(GRID_SIZE * sizeof(*tiles[x]));
      for (int y = 0; y < GRID_SIZE; y++) {
        tiles[x][y] = (Tile) {
            .default_type = TILE_TYPE_SIZE,
            .current_type = TILE_TYPE_SIZE,
            .state = 0
        };

        int max = 0;
        int max_type = 0;
        for (int i = 0; i < TILE_TYPE_SIZE; i++) {
          if (data[x][y][i] > max) {
            max = data[x][y][i];
            max_type = i;
          }
        }

        tiles[x][y].current_type = max_type;
      }
```

```
    }

    Grid grid = (Grid) {
        .data = tiles,
        .window = window,
        .model = model,
        .ended = true,
        .coord_x = -1,
        .coord_y = -1,
        .export_png = export_png,
        .export_csv = export_csv,
        .wind_direction = wind_direction,
        .wind_speed = wind_speed
    };

    destroy_grid(grid);

    for (int i = 0; i < GRID_SIZE; i++) {
      for (int j = 0; j < GRID_SIZE; j++) {
        free(data[i][j]);
      }
      free(data[i]);
    }
    free(data);
  }

  // Free the memory and close the window
  for (int i = 0; i < count; i++) {
    destroy_grid(grids[i]);
```

main.c (10/10)

```c
  }

  if (enable_graphics) {
    destroy_window(window);
  }
  free(grids);

  return 0;
}
```

misc.c (1/3)

```c
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>

/**
 * Get the maximum of two integers
 *
 * @param a The first integer
 * @param b The second integer
 * @return The maximum of the two integers
 */
int max(int a, int b) {
  return a > b ? a : b;
}

/**
 * Get the minimum of two integers
 *
 * @param a The first integer
 * @param b The second integer
 * @return The minimum of the two integers
 */
int min(int a, int b) {
  return a < b ? a : b;
}

/**
 * Read a file and return its content as a string
```

misc.c (2/3)

```
 *
 * @param file The file to read
 * @return The content of the file
 */
char * readfile(FILE * file) {
  // Check if the file is null or if the seek failed
  if (file == NULL || fseek(file, 0, SEEK_END)) {
    return NULL;
  }

  long length = ftell(file);
  rewind(file);
  // Check if the length is invalid
  if (length == -1 || (unsigned long) length >= SIZE_MAX) {
    return NULL;
  }

  // Convert from long to size_t
  size_t ulength = (size_t) length;
  char * buffer = malloc(ulength + 1);
  // Check if the buffer is null or if the read failed
  if (buffer == NULL || fread(buffer, 1, ulength, file) != ulength) {
    free(buffer);
    return NULL;
  }
  // Finish the string
  buffer[ulength] = '\0';

  return buffer;
```

misc.c (3/3)

```c
}

/**
 * Get a random number between 0 and max (excluded)
 *
 * @param max The maximum value
 * @return The random number
 */
int get_random(int max) {
  return (int) (random() % max);
}
```

grid.c (1/21)

```c
#include "draw.c"
#include <cjson/cJSON.h>
#include <unistd.h>
#include <png.h>
#include <sys/stat.h>

/**
 * Model 0 constants
 *
 * The following probabilities are 1 / [number]
 */
/**
 * The probability for a tree tile to burn
 */
const int M0_PROBA_TREE_BURN = 8;
/**
 * The probability for a grass tile to burn
 */
const int M0_PROBA_GRASS_BURN = 8;
/**
 * The probability for a tile to change state between fire and burnt
 */
const int M0_PROBA_STATE_CHANGE = 16;

/**
 * Model 1 constants
 *
 * The following probabilities are 1 / [number]
```

```
 */
/**
 * The probability for a tree tile to burn (in direct neighbors)
 */
const int M1_C_PROBA_TREE_BURN = 8;
/**
 * The probability for a grass tile to burn (in direct neighbors)
 */
const int M1_C_PROBA_GRASS_BURN = 8;
/**
 * The probability for a tree tile to burn (in diagonal neighbors)
 */
const int M1_D_PROBA_TREE_BURN = 16;
/**
 * The probability for a grass tile to burn (in diagonal neighbors)
 */
const int M1_D_PROBA_GRASS_BURN = 16;
/**
 * The probability for a tile to change state between fire and burnt
 */
const int M1_PROBA_STATE_CHANGE = 16;

void write_to_file(Grid grid);

/**
 * Create a grid
 *
 * @param model The model of the grid
 * @param window The window to draw the grid
```

```c
 * @param coord_x The x coordinate of the grid
 * @param coord_y The y coordinate of the grid
 * @return The created grid
 */
Grid create_grid(int model, Window window, int coord_x, int coord_y, bool
↪  export_csv, bool export_png) {
  // Create the grid
  Grid grid = {
      .data = (Tile **) malloc(GRID_SIZE * sizeof(*grid.data)),
      .window = window,
      .model = model,
      .ended = false,
      .coord_x = coord_x,
      .coord_y = coord_y,
      .export_csv = export_csv,
      .export_png = export_png
  };

  // Initialize the grid
  for (int i = 0; i < GRID_SIZE; i++) {
    grid.data[i] = (Tile *) malloc(GRID_SIZE * sizeof(*grid.data[i]));
  }

  // Load the grid from a json file if it exists, otherwise create a random grid
  if (access("grid.json", F_OK) == 0) {
    // The json file exists, we load the grid from it
    cJSON * grid_json = cJSON_Parse(readfile(fopen("grid.json", "r")));
    cJSON * grid_json_object = cJSON_GetObjectItem(grid_json, "grid");
```

```
    for (int i = 0; i < GRID_SIZE; i++) {
      cJSON * row = cJSON_GetArrayItem(grid_json_object, i);
      for (int j = 0; j < GRID_SIZE; j++) {
        // Get the value of the tile and set it to the grid
        int value = cJSON_GetArrayItem(row, j)->valueint;

        grid.data[i][j].current_type = value;
        grid.data[i][j].default_type = value;
        grid.data[i][j].state = 0;
      }
    }
  } else {
    // The json file does not exist, we create a random grid
    for (int i = 0; i < GRID_SIZE; i++) {
      for (int j = 0; j < GRID_SIZE; j++) {
        // Get a random value between 0 and 3 and set it to the grid
        int value = get_random(4);

        grid.data[i][j].current_type = value;
        grid.data[i][j].default_type = value;
        grid.data[i][j].state = 0;
      }
    }
  }

  return grid;
}

/**
```

```c
 * Copy a grid
 *
 * @param data The grid to copy
 * @return The copied grid
 */
Tile ** copy_grid(Tile ** data) {
  // Malloc the copy of the grid
  Tile ** copy = (Tile **) malloc(GRID_SIZE * sizeof(*copy));

  // Fill the copy with the data of the grid
  for (int i = 0; i < GRID_SIZE; i++) {
    copy[i] = (Tile *) malloc(GRID_SIZE * sizeof(*copy[i]));
    memcpy(copy[i], data[i], GRID_SIZE * sizeof(*copy[i]));
  }

  return copy;
}

/**
 * Get the tile at a point
 *
 * @param grid The grid
 * @param point The point
 * @return The tile at the point
 */
Tile get_tile(Grid grid, Point point) {
  return grid.data[point.x][point.y];
}
```

grid.c (6/21)

```c
/**
 * Get the direct neighbors of a point
 *
 * @param grid The grid
 * @param point The point
 * @return The neighbors of the point
 */
Point * get_direct_neighbors(Grid * grid, Point point) {
  Point * neighbors = (Point *) malloc(4 * sizeof(*neighbors));

  neighbors[0] = (Point) {point.x - 1, point.y};
  neighbors[1] = (Point) {point.x + 1, point.y};
  neighbors[2] = (Point) {point.x, point.y - 1};
  neighbors[3] = (Point) {point.x, point.y + 1};

  return neighbors;
}

/**
 * Get the diagonal neighbors of a point
 *
 * @param grid The grid
 * @param point The point
 * @return The neighbors of the point
 */
Point * get_diagonal_neighbors(Grid * grid, Point point) {
  Point * neighbors = (Point *) malloc(4 * sizeof(*neighbors));

  neighbors[0] = (Point) {point.x - 1, point.y - 1};
```

```c
  neighbors[1] = (Point) {point.x + 1, point.y - 1};
  neighbors[2] = (Point) {point.x - 1, point.y + 1};
  neighbors[3] = (Point) {point.x + 1, point.y + 1};

  return neighbors;
}

/**
 * Check if a point is valid (ie inside the grid)
 *
 * @param point The point to check
 * @return True if the point is valid, false otherwise
 */
bool is_valid(Point point) {
  return point.x >= 0 && point.x < GRID_SIZE && point.y >= 0 && point.y < GRID_SIZE;
}

/**
 * Check if the grid is ended
 *
 * @param grid The grid to check
 * @return True if the grid is ended, false otherwise
 */
bool is_ended(Grid grid) {
  if (grid.model == 0 || grid.model == 1 || grid.model == 2) {
    bool is_fire = false;

    // Check if there is no more fire, if there is no more fire, the grid is ended
    for (int i = 0; i < GRID_SIZE; i++) {
```

```
      for (int j = 0; j < GRID_SIZE; j++) {
        if (grid.data[i][j].current_type == FIRE) {
          is_fire = true;
          break;
        }
      }
    }

    return !is_fire;
  } else {
    // Unknown model
    return true;
  }
}

/**
 * Check a probability : if the tile is of the given type and the probability is
 ↪  valid
 *
 * @param grid The grid
 * @param point The point to check
 * @param type The type to check
 * @param proba The probability
 * @return True if the probability is valid, false otherwise
 */
bool check_probability(Grid * grid, Point point, TileType type, int proba) {
  return get_tile(*grid, point).current_type == type && get_random(proba) == 0;
}
```

```c
/**
 * Apply the rules to a cell (model 0 and 1)
 *
 * @param grid The grid
 * @param copy The copy of the grid
 * @param point The point to apply the rules to
 * @param neighbors The neighbors of the point
 * @param tree_burn The probability for a tree tile to burn
 * @param grass_burn The probability for a grass tile to burn
 * @param state_change The probability for a tile to change state between fire and
 ↪  burnt
 */
void apply_to_cell(Grid * grid, Tile ** copy, Point point, Point * neighbors, int
↪  tree_burn, int grass_burn,
           int state_change) {
  // First step, change the state of the neighbors based on the probability
  for (int k = 0; k < 4; k++) {
    if (is_valid(neighbors[k])) {
      if (check_probability(grid, neighbors[k], TREE, tree_burn) ||
        check_probability(grid, neighbors[k], GRASS, grass_burn)) {
        Tile * tile_copy = &copy[neighbors[k].x][neighbors[k].y];

        tile_copy->current_type = FIRE;
        tile_copy->state = 0;
      }
    }
  }

  // Second step, change the state of the point based on the probability to a new
  ↪  state or to burnt
```

```c
  Tile point_tile = get_tile(*grid, point);
  if (check_probability(grid, point, FIRE, state_change)) {
    Tile * tile_copy = &copy[point.x][point.y];

    // If the tile is newly on fire, we increment the state of the tile, otherwise we
    ↪  set it to burnt
    if (point_tile.state == 0) {
      tile_copy->state++;
    } else {
      tile_copy->current_type = BURNT;
      tile_copy->state = 0;
    }
  }

  free(neighbors);
}

/**
 * Get the burn probability (used for Alexandridis model)
 *
 * @return The burn probability
 */
double get_burn_probability(Tile tile, Point point, Point parent, Grid * grid) {
  double p_v;
  switch (tile.current_type) {
    case TREE:
    case DENSE_TREE:
      p_v = 0.3;
      break;
```

grid.c (11/21)

```
    case GRASS:
      p_v = -0.1;
      break;
    case TRENCH:
      p_v = -0.50;
      break;
    default:
      p_v = -1;
      break;
  }

  int dx = point.x - parent.x;
  int dy = point.y - parent.y;
  double theta = 0;

  switch (dx) {
    case 0: {
      if (dy == 0) {
        theta = 0; // Should not happen
      } else if (dy == 1) {
        theta = 90;
      } else {
        theta = 270;
      }
      break;
    }
    case 1: {
      if (dy == 0) {
        theta = 0;
```

```c
      } else if (dy == 1) {
        theta = 45;
      } else {
        theta = 315;
      }
      break;
    }
    case -1: {
      if (dy == 0) {
        theta = 180;
      } else if (dy == 1) {
        theta = 135;
      } else {
        theta = 225;
      }
      break;
    }
  }

  theta = (grid->wind_direction - theta) * M_PI / 180;

  double p_d = 0; // TODO : Implement density
  if (tile.current_type == DENSE_TREE) {
    p_d = 0.3;
  }

  double p_h = 0.34; // TODO : Compute value, best value is 0.58 according to the
↪   paper
  double p_w =
```

```c
    exp(0.045 * grid->wind_speed) * exp(grid->wind_speed * 0.131 * (cos(theta) -
    ↪  1)); // TODO : Implement wind
  double p_s = exp(0.078 * 0 /* TODO : Add angle for slope */);

  return p_h * (1 + p_v) * (1 + p_d) * p_w * p_s;
}

/**
 * Update the grid
 *
 * @param grid The grid to update
 */
void tick(Grid * grid) {
  Tile ** copy = copy_grid(grid->data);

  // Update the grid based on the model
  if (grid->model == 0) {
    // MODEL 0 -> 4 neighbors
    for (int i = 0; i < GRID_SIZE; i++) {
      for (int j = 0; j < GRID_SIZE; j++) {
        Point point = (Point) {i, j};

        // If the tile is not on fire, we continue
        if (get_tile(*grid, point).current_type != FIRE) {
          continue;
        }

        // Apply the rules to the cell
        apply_to_cell(grid, copy, point, get_direct_neighbors(grid, point),
        ↪  M0_PROBA_TREE_BURN,
```

grid.c (14/21)

```
                M0_PROBA_GRASS_BURN,
                M0_PROBA_STATE_CHANGE);
      }
    }

    free(grid->data);

    grid->data = copy;

    draw_grid(grid->window, *grid);
  } else if (grid->model == 1) {
    // MODEL 1 -> 8 neighbors (same as model 0 but with diagonal neighbors)
    for (int i = 0; i < GRID_SIZE; i++) {
      for (int j = 0; j < GRID_SIZE; j++) {
        Point point = (Point) {i, j};

        // If the tile is not on fire, we continue
        if (get_tile(*grid, point).current_type != FIRE) {
          continue;
        }

        // Apply the rules to the cell
        apply_to_cell(grid, copy, point, get_direct_neighbors(grid, point),
        ↪  M1_C_PROBA_TREE_BURN,
                M1_C_PROBA_GRASS_BURN,
                M1_PROBA_STATE_CHANGE);
        apply_to_cell(grid, copy, point, get_diagonal_neighbors(grid, point),
        ↪  M1_D_PROBA_TREE_BURN,
                M1_D_PROBA_GRASS_BURN,
```

```
                M1_PROBA_STATE_CHANGE);
      }
    }

    free(grid->data);

    grid->data = copy;

    draw_grid(grid->window, *grid);
  } else if (grid->model == 2) { // Alexandridis
    for (int i = 0; i < GRID_SIZE; i++) {
      for (int j = 0; j < GRID_SIZE; j++) {
        Point point = (Point) {i, j};

        // If the tile is not on fire, we continue to the next tile
        Tile tile = get_tile(*grid, point);
        if (tile.current_type != FIRE) {
          continue;
        }

        Tile * copy_tile = &copy[point.x][point.y];

        if (tile.state == 0) {
          Point * direct_neighbors = get_direct_neighbors(grid, point);
          Point * diagonal_neighbors = get_diagonal_neighbors(grid, point);

          for (int k = 0; k < 4; k++) {
            Point direct_point = direct_neighbors[k];
            if (is_valid(direct_point)) {
```

```
      Tile direct_tile = get_tile(*grid, direct_point);
      double p_burn = get_burn_probability(direct_tile, direct_point, point,
      ↪  grid);

      if (get_random(1000000) < p_burn * 1000000) {
        Tile * copy_direct_tile = &copy[direct_point.x][direct_point.y];

        copy_direct_tile->current_type = FIRE;
        copy_direct_tile->state = 0;
      }
    }

    Point diagonal_point = diagonal_neighbors[k];
    if (is_valid(diagonal_point)) {
      Tile diagonal_tile = get_tile(*grid, diagonal_point);
      double p_burn = get_burn_probability(diagonal_tile, diagonal_point,
      ↪  point, grid);

      if (get_random(1000000) < p_burn * 1000000) {
        Tile * copy_diagonal_tile =
        ↪  &copy[diagonal_point.x][diagonal_point.y];

        copy_diagonal_tile->current_type = FIRE;
        copy_diagonal_tile->state = 0;
      }
    }
  }

  free(direct_neighbors);
```

grid.c (17/21)

```
          free(diagonal_neighbors);

          copy_tile->state = 1;
        } else {
          copy_tile->current_type = BURNT;
          copy_tile->state = 0;
        }
      }
    }

    free(grid->data);

    grid->data = copy;

    draw_grid(grid->window, *grid);
  } else {
    // Unknown model :(
    free(copy);
  }
}

/**
 * Write to png file
 *
 * @param grid The grid to write
 */
void write_png(Grid grid) {
  struct stat st = {0};
  if (stat("grids_png", &st) == -1) {
```

```
  mkdir("grids_png", 0700);
}

char * file_name = malloc(100 * sizeof(*file_name));
sprintf(file_name, "grids_png/grid-%d-%d.png", grid.coord_x, grid.coord_y);

FILE * fp = fopen(file_name, "wb");
if (!fp) {
  fprintf(stderr, "Failed to open file %s for writing\n", file_name);
  return;
}

png_structp png = png_create_write_struct(PNG_LIBPNG_VER_STRING, NULL, NULL, NULL);
if (!png) {
  fprintf(stderr, "Failed to create png write struct\n");
  fclose(fp);
  return;
}

png_infop info = png_create_info_struct(png);
if (!info) {
  fprintf(stderr, "Failed to create png info struct\n");
  png_destroy_write_struct(&png, NULL);
  fclose(fp);
  return;
}

if (setjmp(png_jmpbuf(png))) { // To handle errors
  printf("Error during png creation\n");
```

```
    png_destroy_write_struct(&png, &info);
    fclose(fp);
    return;
  }

  png_init_io(png, fp);

  // Write the header (8-bit color depth, RGB format)
  png_set_IHDR(png, info, 512, 512, 8, PNG_COLOR_TYPE_RGB,
         PNG_INTERLACE_NONE, PNG_COMPRESSION_TYPE_DEFAULT, PNG_FILTER_TYPE_DEFAULT);
  png_write_info(png, info);

  png_bytep row = (png_bytep) malloc(3 * 512 * sizeof(png_byte));

  Tile tile;
  Color color;
  for (int y = 0; y < 512; y++) {
    for (int x = 0; x < 512; x++) {
      tile = grid.data[x / TILE_SIZE][y / TILE_SIZE];
      color = get_color(tile.current_type, tile.state);
      row[x * 3 + 0] = color.r; // Red
      row[x * 3 + 1] = color.g;   // Green
      row[x * 3 + 2] = color.b;   // Blue
    }
    png_write_row(png, row);
  }

  // Finish writing the file
  png_write_end(png, NULL);
```

```
  // Free resources
  fclose(fp);
  png_destroy_write_struct(&png, &info);
  free(row);

  free(file_name);
}

/**
 * Write to a csv file
 *
 * @param grid The grid to write
 */
void write_csv(Grid grid) {
  FILE * fp = fopen("grids.csv", "a");

  fprintf(fp, "NEW GRID\n"); // Grid Separator

  for (int x = 0; x < GRID_SIZE; x++) {
    for (int y = 0; y < GRID_SIZE; ++y) {
      fprintf(fp, "%d-%d-%d,", grid.data[x][y].current_type,
      ↪  grid.data[x][y].default_type, grid.data[x][y].state);
    }

    fprintf(fp, "\n");
  }

  fclose(fp);
```

grid.c (21/21)

```c
}

/**
 * Destroy a grid
 *
 * @param grid The grid to destroy
 */
void destroy_grid(Grid grid) {
  if (grid.export_png) {
    write_png(grid);
  }

  if (grid.export_csv) {
    write_csv(grid);
  }

  // Free the data of the grid
  for (int i = 0; i < GRID_SIZE; i++) {
    free(grid.data[i]);
  }

  free(grid.data);
}
```

typings.c (1/7)

```c
#include <SDL2/SDL.h>
#include <stdbool.h>

/**
 * Represents the size of the grid
 */
const int GRID_SIZE = 256;
/**
 * Represents the size of a tile
 */
int TILE_SIZE = 2;

/**
 * Represents a window
 */
typedef struct {
  /**
   * The SDL window
   */
  SDL_Window * window;
  /**
   * The SDL surface
   */
  SDL_Surface * surface;
} Window;

/**
 * Represents a color
```

typings.c (2/7)

```
 */
typedef struct {
  /**
   * The red component of the color (0-255)
   */
  int r;
  /**
   * The green component of the color (0-255)
   */
  int g;
  /**
   * The blue component of the color (0-255)
   */
  int b;
} Color;

/**
 * Represents a point
 */
typedef struct {
  /**
   * The x coordinate of the point
   */
  int x;
  /**
   * The y coordinate of the point
   */
  int y;
} Point;
```

typings.c (3/7)

```c
/**
 * Represents a tile type
 */
typedef enum {
  /**
   * A tree tile
   */
  TREE,
  /**
   * A dense tree tile
   */
  DENSE_TREE,
  /**
   * A water tile
   */
  WATER,
  /**
   * A grass tile
   */
  GRASS,
  /**
   * A fire tile
   */
  FIRE,
  /**
   * A burnt tile
   */
  BURNT,
```

```c
  /**
   * A trench tile
   */
  TRENCH,
  /**
   * Just to have a size for the enum
   */
  TILE_TYPE_SIZE
} TileType;

/**
 * Represents a tile
 */
typedef struct {
  /**
   * The default type of the tile
   */
  TileType default_type;
  /**
   * The current type of the tile
   */
  TileType current_type;
  /**
   * The state of the tile (for example, the state of a fire)
   */
  int state;
} Tile;

/**
```

```
 * Represents a grid
 */
typedef struct {
  /**
   * The data of the grid
   */
  Tile ** data;
  /**
   * The window of the grid
   */
  Window window;
  /**
   * The model of the grid
   */
  int model;
  /**
   * Whether the grid has ended
   */
  bool ended;
  /**
   * The x coordinate of the grid
   */
  int coord_x;
  /**
   * The y coordinate of the grid
   */
  int coord_y;
  /**
   * Whether to save the content into a png file
```

```c
   */
  bool export_png;
  /**
   * Whether to save the content into a csv file
   */
  bool export_csv;
  /**
   * Wind direction
   */
  double wind_direction;
  /**
   * Wind speed
   */
  double wind_speed;
} Grid;

/**
 * Get a color according to a tile type and a state
 *
 * @param type The type of the tile
 * @param state The state of the tile
 * @return The color of the tile
 */
Color get_color(TileType type, int state) {
  switch (type) {
    case TREE:
      return (Color) {30, 81, 52};
    case DENSE_TREE:
      return (Color) {18, 49, 33};
```

typings.c (7/7)

```c
    case WATER:
      return (Color) {113, 175, 172};
    case GRASS:
      return (Color) {53, 105, 74};
    case FIRE:
      switch (state) {
        case 0:
          return (Color) {253, 54, 23};
        case 1:
          return (Color) {255, 108, 46};
        default:
          return (Color) {253, 54, 23};
      }
    case BURNT:
      return (Color) {78, 78, 78};
    case TRENCH:
      return (Color) {77, 5, 0};
  }

  return (Color) {0, 0, 0};
}
```

draw.c (1/6)

```c
#include <stdbool.h>
#include <unistd.h>
#include "typings.c"
#include "misc.c"

/**
 * Draw a pixel on the window
 *
 * @param window The window to draw on
 * @param point The point to draw
 * @param color The color of the pixel
 * @param update Whether to update the window (ie to display the pixel)
 */
void draw_pixel(Window window, Point point, Color color, bool update) {
  // If the point is outside the window, do nothing
  if (point.x < 0 || point.x >= window.surface->w || point.y < 0 || point.y >=
  ↪  window.surface->h) {
    return;
  }

  // Get the pixel at the point and set its color
  Uint32 * pixel = (Uint32 *) window.surface->pixels + point.y *
  ↪  window.surface->pitch / 4 + point.x;
  *pixel = SDL_MapRGB(window.surface->format, color.r, color.g, color.b);

  // If we want to update the window, we update it
  if (update) {
    SDL_UpdateWindowSurface(window.window);
```

```
  }
}

/**
 * Draw a square on the window
 *
 * @param window The window to draw on
 * @param point The top-left corner of the square
 * @param size The size of the square
 * @param color The color of the square
 * @param update Whether to update the window (ie to display the square)
 */
void draw_square(Window window, Point point, int size, Color color, bool update) {
  // Draw a square of pixels
  for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
      // Draw the pixel
      draw_pixel(window, (Point) {point.x + i, point.y + j}, color, false);
    }
  }

  // If we want to update the window, we update it
  if (update) {
    SDL_UpdateWindowSurface(window.window);
  }
}

/**
 * Draw the grid on the window
```

```c
 *
 * @param window The window to draw on
 * @param grid The grid to draw
 */
void draw_grid(Window window, Grid grid) {
  // Graphics not enabled
  if (!window.window) {
    return;
  }

  // Draw the grid using the constants defined in typings.c, and translate the grid
  ↪  to the right position
  for (int i = 0; i < GRID_SIZE; i++) {
    for (int j = 0; j < GRID_SIZE; j++) {
      Tile tile = grid.data[i][j];

      // Draw the tile as a square
      draw_square(window, (Point) {TILE_SIZE * (i + (GRID_SIZE + 1) * grid.coord_x),
                     TILE_SIZE * (j + (GRID_SIZE + 1) * grid.coord_y)}, TILE_SIZE,
            get_color(tile.current_type, tile.state), false);
    }
  }

  // Update the window to display the grid
  SDL_UpdateWindowSurface(window.window);
}

/**
 * Create a window
```

```c
  *
  * @param max_x The maximum number of grids on the x axis
  * @param max_y The maximum number of grids on the y axis
  * @return The window
  */
Window create_window(int max_x, int max_y) {
  // The window and the surface of the window
  Window window = {
      .window = NULL,
      .surface = NULL
  };

  // Define the size of the tiles based on the number of grids (to ensure that the
  ↪  window is not too big)
  TILE_SIZE = TILE_SIZE - (1.5) * min(max_y - 1, 4);

  if (SDL_Init(SDL_INIT_VIDEO) < 0) {
    // SDL initialization failed
    printf("SDL could not initialize! SDL_Error: %s\n", SDL_GetError());
    exit(1);
  } else {
    // Create the window
    window.window = SDL_CreateWindow(
        "TIPE",
        SDL_WINDOWPOS_UNDEFINED,
        SDL_WINDOWPOS_UNDEFINED,
        (max_x * (GRID_SIZE + 1) - 1) * TILE_SIZE,
        (max_y * (GRID_SIZE + 1) - 1) * TILE_SIZE,
        SDL_WINDOW_SHOWN
```

draw.c (5/6)

```
    );

    if (window.window == NULL) {
      // Window creation failed
      printf("Window could not be created! SDL_Error: %s\n", SDL_GetError());
      exit(1);
    } else {
      window.surface = SDL_GetWindowSurface(window.window);
    }
  }

  return window;
}

/**
 * Wait for a certain amount of time
 *
 * @param ms The number of milliseconds to wait
 */
void wait(int ms) {
  usleep(ms * 1000);
}

/**
 * Destroy a window
 *
 * @param window The window to destroy
 */
void destroy_window(Window window) {
```

draw.c (6/6)

```
  // Destroy the window and quit SDL
  SDL_DestroyWindow(window.window);
  SDL_Quit();
}
```