# canvas.h

```c
#ifndef CANVAS_H
#define CANVAS_H

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <float.h>
#include <assert.h>
#include <time.h>
#include <math.h>
#include <SDL.h>

#include "tools.h"
#include "linked_list.h"



// ----- STRUCTURE DEFINITIONS ----- //

struct cell_s {
    SDL_Rect fill_square; // rectangle for color filling - contains pixel coordinates
    double attractiveness;
    int service_presence_class; // -1 if unoccupied by a service
    int entertain_presence_type;

    double terrain_height;
};

struct vcanvas_s {
    int width_cells; // width in number of cells
    int height_cells; // height in number of cells
    int cell_size; // on-screen size of cells in pixels

    struct cell_s** cellsA;
    int* sorted_cell_indices_attractiveness; // by default, is equal to the identity
    // array - the sorting
    // is done according to attractivenesses and indices
    // correspond to the linearized positions of cells in
    // cellsA
    l_list service_border_pixelsL; // midpoints of service boundaries
    l_list service_interior_cellsL;
};
```

```
43
44    typedef struct cell_s cell;
45    typedef struct vcanvas_s* canvas;
46
47
48    // ----- GLOBAL CONSTANTS ----- //
49
50    extern const int cCellSizePixels;
51    extern const int cCellHomeLimit;
52
53    extern const int cNumServiceClasses;
54    extern const int cNumEntertainTypes;
55    extern const double cServiceWeights[7];
56
57    extern const double cDistanceScaleFactor; // factor by which
58
59
60    extern const int cWindowDims[2];
61
62
63    // ----- FUNCTIONS ----- //
64
65    // creating canvases
66    canvas CREATE_CANVAS (int CellSizePixels, const int WindowDims[2]);
67
68    // handy functions
69    int CANVAS_HEIGHT (canvas Canvas, const int WindowDims[2]);
70    int CANVAS_WIDTH (canvas Canvas, const int WindowDims[2]);
71    int LIN_COEFF (canvas Canvas, int row, int column);
72    void ASSIGN_DELIN_INDEX (int* prow, int* pcolumn, canvas Canvas, int index);
73
74    // cell-pixel correspondece
75    void ASSIGN_MIDPOINT_OF_CELL (int* pX, int* pY, canvas Canvas, int row, int column);
76    void ASSIGN_CELL_OF_PIXEL (int* prow, int* pcolumn, canvas Canvas, int X, int Y);
77
78    // cell tests
79    bool IS_CELL_IN_CANVAS (canvas Canvas, int i, int j);
80    bool ARE_NEIGHBOR_CELLS (canvas Canvas, int i1, int j1, int i2, int j2);
81
82    // ensuring correctness
83    void SORT_CELL_INDICES (canvas Canvas);
84
85    // calculation functions
86    double F_CLASS_PLACEMENT_SCORE (double SquareDist, int class);
```

```
87
88    // calculating canvas attributes
89    void CALCULATE_ATTRACTIVENESSES (canvas Canvas);
90
91    // quicksort
92    void SWAP (int* IntTab, double* DoubleTab, int pos1, int pos2);
93    int QS_PARTITION_CUT (int* CellIndices, double* CellScores, int start, int end, int pivot);
94    void QS_SORT_ELEMENTS_BY_SCORE (int* CellIndices, double* CellScores, int start, int end);
95
96
97    #endif
```

## canvas.c

```
1     #include "canvas.h"
2
3     // constants
4     const int cCellSizePixels = 10;
5     const int cNumServiceClasses = 7;
6     const int cNumEntertainTypes = 2;
7
8     const int cCellHomeLimit = 10;
9     const double cServiceWeights[7] = {1.2, 0.9, 0.6, 0.4, 0.3, -0.5, -1};
10
11    const double cDistanceScaleFactor = 2.0;
12
13
14
15    // returns an empty canvas (i.e. with all attributes initialized to their null
16    // or neutral value) covering the entire screen, and whose cells are sized
17    // CellSizePixels in size
18    canvas CREATE_CANVAS (int CellSizePixels, const int WindowDims[2]){
19
20        canvas Canvas = (canvas) malloc (sizeof(struct vcanvas_s));
21
22        Canvas->width_cells = ceil (WindowDims[0]/cCellSizePixels);
23        Canvas->height_cells = CANVAS_HEIGHT (Canvas, WindowDims);
24        Canvas->cell_size = CellSizePixels;
25
26        Canvas->service_border_pixelsL = NULL;
27        Canvas->service_interior_cellsL = NULL;
28
```

```
29        Canvas->cellsA = (cell**) malloc (Canvas->height_cells*sizeof(cell*));
30        Canvas->sorted_cell_indices_attractiveness = (int*) malloc
31        /**/ (Canvas->height_cells*Canvas->width_cells*sizeof(int));
32
33        for (int index = 0; index < Canvas->height_cells*Canvas->width_cells; index++)
34        Canvas->sorted_cell_indices_attractiveness[index] = index;
35
36        for (int i = 0; i < Canvas->height_cells; i++){
37
38            Canvas->cellsA[i] = (cell*) malloc (Canvas->width_cells*sizeof(cell));
39
40            for (int j = 0; j < Canvas->width_cells; j++){
41
42                Canvas->cellsA[i][j].attractiveness = 0;
43                Canvas->cellsA[i][j].service_presence_class = -1;
44
45                Canvas->cellsA[i][j].terrain_height = -DBL_MAX;
46
47                Canvas->cellsA[i][j].fill_square.x = j*cCellSizePixels;
48                Canvas->cellsA[i][j].fill_square.y = i*cCellSizePixels;
49                Canvas->cellsA[i][j].fill_square.w = cCellSizePixels;
50                Canvas->cellsA[i][j].fill_square.h = cCellSizePixels;
51
52            }
53        }
54
55        return Canvas;
56    }
57
58
59    /* hyp: Canvas->width_cells has been initialized */
60    // returns the appropriate value of Canvas->width_cells according
61    // to Canvas->height_cells and the window size as given by WindowDims
62    int CANVAS_HEIGHT (canvas Canvas, const int WindowDims[2]){
63
64        return (int) ((((double) WindowDims[1])/((double) WindowDims[0]))
65        /**/ * ((double) Canvas->width_cells));
66    }
67
68
69    /* precond: Canvas->height_cells has been initialized */
70    // returns the appropriate value of Canvas->width_cells according
71    // to Canvas->height_cells and the window size as given by WindowDims
72    int CANVAS_WIDTH (canvas Canvas, const int WindowDims[2]){
```

```
73
74          return (int) ((((double) WindowDims[1])/((double) WindowDims[0]))
75          /**/ * ((double) Canvas->height_cells));
76      }
77
78
79      int LIN_COEFF (canvas Canvas, int row, int column){
80
81          return row*Canvas->width_cells + column;
82      }
83
84
85      void ASSIGN_DELIN_INDEX (int* prow, int* pcolumn, canvas Canvas, int index){
86
87          *prow = index/Canvas->width_cells;
88          *pcolumn = index % Canvas->width_cells;
89      }
90
91
92      // assigns the on-screen pixel coordinates of cell Canvas->cellsA[row][column]'s
93      // midpoint to *pX and *pY
94      void ASSIGN_MIDPOINT_OF_CELL (int* pX, int* pY, canvas Canvas, int row, int column){
95
96          *pX = ((2*column)*Canvas->cell_size)/2;
97          *pY = ((2*row)*Canvas->cell_size)/2;
98      }
99
100
101     // modifies *prow and *pcolumn such that Canvas->cellsA[*prow][*pcolumn] contains
102     // the on-screen pixel of coordinates (X,Y)
103     void ASSIGN_CELL_OF_PIXEL (int* prow, int* pcolumn, canvas Canvas, int X, int Y){
104
105         *prow = Y/Canvas->cell_size;
106         *pcolumn = X/Canvas->cell_size;
107     }
108
109
110     // tests whether or not (i,j) forms a valid cell index for Canvas
111     bool IS_CELL_IN_CANVAS (canvas Canvas, int i, int j){
112
113         return (i >= 0 && i < Canvas->height_cells && j >= 0 && j < Canvas->width_cells);
114     }
115
116
```

```
117   /* hyp: IS_CELL_IN_CANVAS (Canvas, i1, j1) && IS_CELL_IN_CANVAS (Canvas, i2, j2) */
118   // tests whether or not cells
119   bool ARE_NEIGHBOR_CELLS (canvas Canvas, int i1, int j1, int i2, int j2){
120
121       assert (IS_CELL_IN_CANVAS (Canvas, i1, j1) && IS_CELL_IN_CANVAS (Canvas, i2, j2));
122       return (i1 >= i2-1 && i1 <= i2+1 && j1 >= j2-1 && j1 <= j2+1);
123   }
124
125
126   // updates Canvas->sorted_cell_indices such that the sequence
127   // (Canvas->cellsA[ik][jk].attractiveness), where LIN (Canvas, ik, jk) = k, increases
128   // with k
129   void SORT_CELL_INDICES (canvas Canvas){
130
131       int NumCells = Canvas->width_cells*Canvas->height_cells;
132       int* CellIndices = Canvas->sorted_cell_indices_attractiveness;
133       double* CellAttractivenesses = (double*) malloc (NumCells*sizeof(double));
134
135       for (int index = 0; index < NumCells; index++){
136
137           int ci, cj;
138           ASSIGN_DELIN_INDEX (&ci, &cj, Canvas, CellIndices[index]);
139           CellAttractivenesses[index] = Canvas->cellsA[ci][cj].attractiveness;
140       }
141
142       QS_SORT_ELEMENTS_BY_SCORE (CellIndices, CellAttractivenesses, 0, NumCells-1);
143
144       free (CellAttractivenesses);
145   }
146
147   // hyp: 0 <= class < cNumServiceClasses
148   // calculates the placement score contribution of service class class
149   // for a cell at distance sqrt(SquareDist) away from said class
150   double F_CLASS_PLACEMENT_SCORE (double SquareDist, int class){
151       double Dist = sqrt (SquareDist);
152       if (cServiceWeights[class] > 0)
153           return (0.02*cServiceWeights[class]*(10.+(Dist/2.))
154           /**/ *exp(-(cServiceWeights[class]*SquareDist)/12000.));
155       else
156           return (0.6*cServiceWeights[class]
157           /**/ *exp((cServiceWeights[class]*SquareDist)/20000.));
158   }
159
160
```

```
161   // assigns to each Canvas->cellsA[i][j] its normalized attractiveness value
162   // (calculated by summing the cell's class placement scores over all classes
163   // and then normalizing by the attractiveness score of a hypothetical cell whose
164   // class placement scores are maximal for each class)
165   void CALCULATE_ATTRACTIVENESSES (canvas Canvas){
166
167       // array that will contain, for each (i,j), Canvas->cellsA[i][j]'s distance to each class'
168       double* SqDistsToClasses = (double*) malloc (cNumServiceClasses*sizeof(double));
169
170       double Normalizer = 0;
171       for (int class = 0; class < cNumServiceClasses; class++){
172           if (cServiceWeights[class] > 0){
173               double ArgmaxScore = (10.*(sqrt(1+(60./cServiceWeights[class]))-1))
174               /**/ *(10.*(sqrt(1+(60./cServiceWeights[class]))-1));
175               Normalizer += F_CLASS_PLACEMENT_SCORE ((int) ArgmaxScore, class);
176           }
177       }
178
179       for (int i = 0; i < Canvas->height_cells; i++){
180           for (int j = 0; j < Canvas->width_cells; j++){
181
182               // only score unoccupied cells
183               if (Canvas->cellsA[i][j].service_presence_class == -1){
184
185                   int CellX = Canvas->cellsA[i][j].fill_square.x;
186                   int CellY = Canvas->cellsA[i][j].fill_square.y;
187
188                   // reset array for current cell's distance calculations
189                   for (int class = 0; class < cNumServiceClasses; class++){
190                       SqDistsToClasses[class] = DBL_MAX;
191                   }
192
193                   // calculate distance to each class by looking at all service borders
194                   // and updating array
195                   l_list ServiceBorders = Canvas->service_border_pixelsL;
196
197                   while (ServiceBorders != NULL){
198
199                       int class = ServiceBorders->spec;
200                       double SqDist = ((ServiceBorders->x - CellX)*(ServiceBorders->x - CellX)
201                       /**/ + (ServiceBorders->y - CellY)*(ServiceBorders->y - CellY))
202                       /**/ /cDistanceScaleFactor;
203                       SqDistsToClasses[class] = min_double (SqDist, SqDistsToClasses[class]);
204
```

```
205                     ServiceBorders = ServiceBorders->next;
206                 }
207 
208                 // calculate attractiveness
209                 Canvas->cellsA[i][j].attractiveness = 0;
210                 for (int class = 0; class < cNumServiceClasses; class++){
211                     if (SqDistsToClasses[class] != DBL_MAX){
212                         Canvas->cellsA[i][j].attractiveness +=
213                         /**/ F_CLASS_PLACEMENT_SCORE (SqDistsToClasses[class], class);
214                     }
215                 }
216                 Canvas->cellsA[i][j].attractiveness =
217                 /**/ Canvas->cellsA[i][j].attractiveness*(1000/Normalizer);
218             }
219         }
220     }
221     SORT_CELL_INDICES (Canvas);
222     free (SqDistsToClasses);
223 }
224 
225 
226 // hyp: if Array is of length l, 0 <= pos1 < l && 0 <= pos2 < l
227 // swap Tab[pos1] and Tab[pos2]
228 void SWAP (int* IntTab, double* DoubleTab, int pos1, int pos2){
229 
230     int TempInt = IntTab[pos1];
231     IntTab[pos1] = IntTab[pos2];
232     IntTab[pos2] = TempInt;
233 
234     double TempDouble = DoubleTab[pos1];
235     DoubleTab[pos1] = DoubleTab[pos2];
236     DoubleTab[pos2] = TempDouble;
237 }
238 
239 
240 // hyp: start, pivot and end are valid indices for ElemIndices and ElemScores
241 // swaps elements in ElemIndices[start..end] and ElemScores[start..end] identically then
242 // returns an index PivotPlace such that after the procedure the following are true:
243 // - ElemScores[PivotPlace] has ElemScores[pivot]'s initial value
244 // - for all i < PivotPlace, ElemScores[i] <= ElemScores[PivotPlace]
245 // - for all i > PivotPlace, ElemScores[i] > ElemScores[PivotPlace]
246 int QS_PARTITION_CUT (int* ElemIndices, double* ElemScores, int start, int end, int pivot){
247 
248     SWAP (ElemIndices, ElemScores, pivot, end);
```

```
    int PivotPlace = start;
    for (int pos = start; pos < end; pos++){
        if (ElemScores[pos] >= ElemScores[end]){
            SWAP (ElemIndices, ElemScores, pos, PivotPlace);
            PivotPlace++;
        }
    }
    SWAP (ElemIndices, ElemScores, PivotPlace, end);
    return PivotPlace;
}


// hyp: start and end are valid indices for ElemIndices and ElemScores
// sorts ElemScores[start..end] in decreasing order using quicksort,
// also sorting ElemScores following the same permutation
void QS_SORT_ELEMENTS_BY_SCORE (int* ElemIndices, double* ElemScores, int start, int end){

    if (start < end){
        int pivot = (rand () % (end-start+1)) + start;
        int PivotPlace = QS_PARTITION_CUT (ElemIndices, ElemScores, start, end, pivot);
        QS_SORT_ELEMENTS_BY_SCORE (ElemIndices, ElemScores, start, PivotPlace-1);
        QS_SORT_ELEMENTS_BY_SCORE (ElemIndices, ElemScores, PivotPlace+1, end);
    }
}
```

## terrain.h

```
#ifndef TERRAIN_H
#define TERRAIN_H

#include "canvas.h"



// ----- FUNCTIONS ----- //

double SMOOTHSTEP (double lambda);
double F_HEIGHT_OF_PIXEL (double** VertexHeights, int NoiseFrequency, int PixelX, int PixelY);
void ADD_NOISE_LAYER (canvas Canvas, int NoiseFrequency, double RotationRadians,
/**/ double LayerWeight);
void GENERATE_TERRAIN_HEIGHTMAP (canvas Canvas, double Amplitude, int Frequency,
```

```
15  /**/ int NumOctaves, double Lacunarity, double Persistence, double Exponentiation);
16
17  #endif
```

## terrain.c

```
1   #include "terrain.h"
2
3
4   const int cNoiseFrequencyPixels = 200;
5
6
7   // returns smoothstep (lambda)
8   double SMOOTHSTEP (double lambda){
9
10      return 3*lambda*lambda - 2*lambda*lambda*lambda;
11  }
12
13
14  // given a noise tiling whose tiles are sized NoiseFrequency and the noise values at each
15  // tile's corners contained in VertexHeights, returns the height map's value at the point
16  // of coordinates (PixelX, PixelY) obtained by interpolating between the nearest vertices
17  // according to the smoothstep function
18  double F_HEIGHT_OF_PIXEL (double** VertexHeights, int NoiseFrequency, int PixelX, int PixelY){
19
20      // find the position of the tile containing PixelX and PixelY
21      int Tile_i = PixelY/NoiseFrequency;
22      int Tile_j = PixelX/NoiseFrequency;
23
24      // printf ("%d,%d\n", Tile_i, Tile_j);
25
26      // extract the height at each one of the tile's corners
27      double a = VertexHeights[Tile_i][Tile_j];     // top-left
28      double b = VertexHeights[Tile_i][Tile_j+1];   // top-right
29      double c = VertexHeights[Tile_i+1][Tile_j];   // bottom-left
30      double d = VertexHeights[Tile_i+1][Tile_j+1]; // bottom-right
31
32      double i_diff = (double)PixelY/(double)NoiseFrequency - (double)Tile_i;
33      double j_diff = (double)PixelX/(double)NoiseFrequency - (double)Tile_j;
34
35      return a + (b-a)*SMOOTHSTEP(j_diff) + (c-a)*SMOOTHSTEP(i_diff) +
```

```
36          /**/ (a-b-c+d)*SMOOTHSTEP(j_diff)*SMOOTHSTEP(i_diff);
37      }
38
39
40      //
41      void ADD_NOISE_LAYER (canvas Canvas, int NoiseFrequency, double RotationRadians,
42      /**/ double LayerWeight){
43
44          // calculate number of tiles in length, in width and diagonally to cover window rectangle
45          int WidthTiles = cWindowDims[0]/NoiseFrequency;
46          int HeightTiles = cWindowDims[1]/NoiseFrequency;
47          int DiagonalSizePixels = (int)ceil(sqrt((double)cWindowDims[0]*cWindowDims[0] +
48          /**/ (double)cWindowDims[1]*cWindowDims[1]));
49
50          // determine heights of tile vertices randomly
51          int InscribingSquareSizeTiles = 2*DiagonalSizePixels/NoiseFrequency;
52          double** VertexHeights = (double**) malloc ((InscribingSquareSizeTiles)*sizeof(double*));
53
54          for (int row = 0; row < InscribingSquareSizeTiles; row++){
55
56              VertexHeights[row] = (double*) malloc ((InscribingSquareSizeTiles)*sizeof(double));
57
58              for (int column = 0; column < InscribingSquareSizeTiles; column++){
59
60                  VertexHeights[row][column] = LayerWeight*(double)rand()/(double)RAND_MAX;
61              }
62          }
63
64          // calculate height of each canvas cell's midpoint pixel according to the above values
65          int CellX, CellY;
66
67          for (int i = 0; i < Canvas->height_cells; i++){
68              for (int j = 0; j < Canvas->width_cells; j++){
69
70                  ASSIGN_MIDPOINT_OF_CELL (&CellX, &CellY, Canvas, i, j);
71                  int HalfWidth = cWindowDims[0]/2;
72                  int HalfHeight = cWindowDims[1]/2;
73
74                  // rotate heightmap by RotationRadians
75                  int RotatedX = cos(RotationRadians)*(CellX-HalfWidth)
76                  /**/ - sin(RotationRadians)*(CellY-HalfHeight) + DiagonalSizePixels/2;
77                  int RotatedY = sin(RotationRadians)*(CellX-HalfWidth)
78                  /**/ + cos(RotationRadians)*(CellY-HalfHeight) + DiagonalSizePixels/2;
79
```

```
80              // printf ("%d, %d ---> (%f) %d, %d\n", CellX, CellY, RotationRadians, RotatedX, RotatedY);
81              Canvas->cellsA[i][j].terrain_height += F_HEIGHT_OF_PIXEL (VertexHeights,
82              /**/ NoiseFrequency, RotatedX, RotatedY);
83          }
84      }
85  }
86
87
88  //
89  void GENERATE_TERRAIN_HEIGHTMAP (canvas Canvas, double Amplitude, int Frequency, int NumOctaves,
90  /**/ double Lacunarity, double Persistence, double Exponentiation){
91
92      for (int i = 0; i < Canvas->height_cells; i++){
93          for (int j = 0; j < Canvas->width_cells; j++){
94
95              Canvas->cellsA[i][j].terrain_height = 0;
96          }
97      }
98
99      double NoiseFrequency = Frequency;
100     double LayerWeight = (1-Persistence);
101
102     for (int layer = 0; layer < NumOctaves; layer++){
103
104         double RandAngle = 2*3.141592*(double)rand()/(double)RAND_MAX;
105
106         ADD_NOISE_LAYER (Canvas, NoiseFrequency, RandAngle, LayerWeight);
107         LayerWeight *= Persistence;
108         NoiseFrequency = (int)ceil((NoiseFrequency/Lacunarity));
109     }
110     for (int i = 0; i < Canvas->height_cells; i++){
111         for (int j = 0; j < Canvas->width_cells; j++){
112
113             Canvas->cellsA[i][j].terrain_height = Amplitude
114             /**/ * pow(Canvas->cellsA[i][j].terrain_height, Exponentiation);
115         }
116     }
117 }
```

# dispo.h

```h
#ifndef DISPO_H
#define DISPO_H

#include "canvas.h"



// ----- STRUCTURE DEFINITIONS ----- //

struct unit_s {
    int num_homes;
    double local_density;
    double local_entropy;
};

struct vdisposition_s {
    int width_units;
    int height_units;

    struct unit_s** unitsA;
    int* sorted_unit_indices_density;

    int num_inhabitants;
    int max_possible_inhabitants;

    double max_possible_density;
    double entropy;
};

typedef struct unit_s unit;
typedef struct vdisposition_s* disposition;


// ----- GLOBAL CONSTANTS ----- //

extern const double cDensityToleranceRadius;
extern const double cEntropyToleranceRadius;
extern const double cHomeAttributionInflation;


// ----- FUNCTIONS ----- //

// creating and handling dispositions
disposition CREATE_CANVAS_DISPOSITION (canvas Canvas);
```

```
45  disposition COPY_DISPOSITION (canvas Canvas, disposition Dispo);
46
47  // ensuring correctness
48  bool IS_DISPOSITION_HOME_ASSIGNMENT_COHERENT (disposition Dispo);
49  void SORT_UNIT_INDICES (canvas Canvas, disposition Dispo);
50
51  // manipulating and placing homes in dispositions
52  bool TRANSFER_HOMES (canvas Canvas, disposition Dispo, int Amount, int i1, int j1, int i2, int j2);
53  void INITIALIZE_DISPOSITION_FROM_CELLS_1 (canvas Canvas, disposition Dispo, int NumHomes,
54  /**/ double HomeAttributionInflation);
55  void INITIALIZE_DISPOSITION_FROM_CELLS_2 (canvas Canvas, disposition Dispo, int NumHomes,
56  /**/ double HomeAttributionInflation);
57  void INITIALIZE_DISPOSITION_BLINDLY (canvas Canvas, disposition Dispo, int NumHomes);
58
59  // calculation and attribution functions
60  double F_DENSITY_CONTRIBUTION (double SqDist, int NumHomes);
61  double F_HOME_ATTRIBUTION_PROPORTION (double Attractiveness, double HighestAttractiveness,
62  /**/ disposition Dispo, double HomeAttributionInflation);
63  double F_ENTROPY_CONTRIBUTION (double Ratio);
64
65  // calculating disposition attributes
66  void CALCULATE_LOCAL_DENSITIES (canvas Canvas, disposition Dispo);
67  void CALCULATE_LOCAL_ENTROPIES (canvas Canvas, disposition Dispo);
68  void CALCULATE_ENTROPY (canvas Canvas, disposition Dispo);
69
70
71  void FREE_DISPOSITION (disposition Dispo);
72  #endif
```

## dispo.c

```
1   #include "dispo.h"
2
3
4   const double cDensityToleranceRadius = 100;
5   const double cEntropyToleranceRadius = 50;
6
7   const double cHomeAttributionInflation = 0.6; // the smaller, the more tolerant attribution
8   // is with regards to attractiveness
9
10
```

```
11  // returns an empty disposition (i.e. with all attributes initialized to their default
12  // value) that is valid for Canvas (i.e. whose dimensions in units are those of
13  // Canvas in cells)
14  disposition CREATE_CANVAS_DISPOSITION (canvas Canvas){
15
16      disposition Dispo = (disposition) malloc (sizeof(struct vdisposition_s));
17      Dispo->width_units = Canvas->width_cells;
18      Dispo->height_units = Canvas->height_cells;
19
20      Dispo->num_inhabitants = 0;
21      Dispo->max_possible_inhabitants = Dispo->width_units*Dispo->height_units*cCellHomeLimit;
22
23      Dispo->unitsA = (unit**) malloc (Dispo->height_units*sizeof(unit*));
24
25      Dispo->sorted_unit_indices_density = (int*) malloc
26      /**/ (Dispo->width_units*Dispo->height_units*sizeof(int));
27
28      // initialize units
29      for (int i = 0; i < Dispo->height_units; i++){
30
31          Dispo->unitsA[i] = (unit*) malloc (Dispo->width_units*sizeof(unit));
32
33          for (int j = 0; j < Dispo->width_units; j++){
34
35              Dispo->unitsA[i][j].num_homes = 0;
36              Dispo->unitsA[i][j].local_density = 0;
37              Dispo->sorted_unit_indices_density[LIN_COEFF (Canvas, i, j)] =
38              /**/ LIN_COEFF (Canvas, i, j);
39          }
40
41      }
42      Dispo->entropy = 0;
43
44      // calculate maximum possible density
45      Dispo->max_possible_density = 0;
46
47      int X_min = -(int)cDensityToleranceRadius; int X_max = (int)cDensityToleranceRadius;
48      int Y_min = -(int)cDensityToleranceRadius; int Y_max = (int)cDensityToleranceRadius;
49      int InscribingSquareSideCells = (int)ceil ((float)(X_max-X_min)/(float)cCellSizePixels);
50
51      for (int row = 0; row <= InscribingSquareSideCells; row++){
52          for (int column = 0; column <= InscribingSquareSideCells; column++){
53
54              int CurrentX = X_min + column*cCellSizePixels;
```

```
55              int CurrentY = Y_min + row*cCellSizePixels;
56              double SqDist = CurrentX*CurrentX + CurrentY*CurrentY;
57
58              if (SqDist <= cDensityToleranceRadius*cDensityToleranceRadius){
59                  Dispo->max_possible_density += F_DENSITY_CONTRIBUTION
60                  /**/ (SqDist, cCellHomeLimit);
61              }
62          }
63      }
64      return Dispo;
65  }
66
67
68
69  void FREE_DISPOSITION (disposition Dispo){
70
71      /* printf ("freeing dispo\n");
72      free (Dispo->unitsA);
73      free (Dispo->sorted_unit_indices_density); */
74  }
75
76
77  // hyp: Dispo is a disposition on Canvas
78  // returns a copy of Dispo
79  disposition COPY_DISPOSITION (canvas Canvas, disposition Dispo){
80
81      disposition DispoCopy = (disposition) malloc (sizeof(struct vdisposition_s));
82      DispoCopy->width_units = Dispo->width_units;
83      DispoCopy->height_units = Dispo->height_units;
84
85      DispoCopy->num_inhabitants = Dispo->num_inhabitants;
86      DispoCopy->max_possible_inhabitants = Dispo->max_possible_inhabitants;
87      DispoCopy->max_possible_density = Dispo->max_possible_density;
88
89      DispoCopy->unitsA = (unit**) malloc (DispoCopy->height_units*sizeof(unit*));
90
91      DispoCopy->sorted_unit_indices_density = (int*) malloc
92      /**/ (DispoCopy->height_units*DispoCopy->width_units*sizeof(int));
93
94      for (int i = 0; i < DispoCopy->height_units; i++){
95
96          DispoCopy->unitsA[i] = (unit*) malloc (DispoCopy->width_units*sizeof(unit));
97
98          for (int j = 0; j < DispoCopy->width_units; j++){
```

```
99
100             DispoCopy->unitsA[i][j].num_homes = Dispo->unitsA[i][j].num_homes;
101             DispoCopy->unitsA[i][j].local_density = Dispo->unitsA[i][j].local_density;
102             DispoCopy->unitsA[i][j].local_entropy = Dispo->unitsA[i][j].local_entropy;
103             DispoCopy->sorted_unit_indices_density[LIN_COEFF (Canvas, i, j)] =
104             /**/ Dispo->sorted_unit_indices_density[LIN_COEFF (Canvas, i, j)];
105         }
106     }
107     return DispoCopy;
108 }
109
110
111 // returns whether or not both statements are simultaneously true:
112 // - all of Dispo's cells contain between 0 and cCellHomeLimit houses
113 // - summing the amount of homes per cell across all cells yields Dispo->num_inhabitants
114 bool IS_DISPOSITION_HOME_ASSIGNMENT_COHERENT (disposition Dispo){
115
116     int NumHomes = 0;
117     for (int i = 0; i < Dispo->height_units; i++){
118         for (int j = 0; j < Dispo->width_units; j++){
119             if (Dispo->unitsA[i][j].num_homes < 0) return false;
120             NumHomes += Dispo->unitsA[i][j].num_homes;
121         }
122     }
123     return (NumHomes == Dispo->num_inhabitants);
124 }
125
126
127 // sorts Dispo->sorted_unit_indices_density such that the sequence
128 // (Dispo->unitsA[ik][jk].local_density), where (ik,jk) is the delinearized coefficient
129 // obtained from k, increases in k
130 void SORT_UNIT_INDICES (canvas Canvas, disposition Dispo){
131
132     int NumUnits = Dispo->height_units*Dispo->width_units;
133     int* UnitIndices = Dispo->sorted_unit_indices_density;
134     double* UnitDensities = (double*) malloc (NumUnits*sizeof(double));
135
136     for (int index = 0; index < NumUnits; index++){
137
138         int ci, cj;
139         ASSIGN_DELIN_INDEX (&ci, &cj, Canvas, UnitIndices[index]);
140         UnitDensities[index] = Dispo->unitsA[ci][cj].local_density;
141     }
142
```

```
143         QS_SORT_ELEMENTS_BY_SCORE (UnitIndices, UnitDensities, 0, NumUnits-1);
144 
145         /* printf ("freeing unitdensities\n");
146         free (UnitDensities); */
147     }
148 
149 
150     // attempts to move Amount homes between Dispo's units of indices (i1,j1) and (i2,j2)
151     // while ensuring that their numbers of homes remains between 0 and cCellHomeLimit
152     // returns true if and only if a non-zero number of homes were actually moved
153     bool TRANSFER_HOMES (canvas Canvas, disposition Dispo, int Amount, int i1, int j1, int i2, int j2){
154 
155         // printf (" into %d\n", Dispo->unitsA[i2][j2].num_homes);
156         Amount = min (min (Amount, Dispo->unitsA[i1][j1].num_homes), cCellHomeLimit -
157         /**/ Dispo->unitsA[i2][j2].num_homes);
158         if (Amount > 0 && Canvas->cellsA[i2][j2].service_presence_class == -1){
159 
160             Dispo->unitsA[i1][j1].num_homes -= Amount;
161             Dispo->unitsA[i2][j2].num_homes += Amount;
162             return true;
163         }
164         else return false;
165     }
166 
167 
168     // hyp: Dispo is a disposition on Canvas && CALCULATE_ATTRACTIVENESSES (Canvas) has
169     // been called
170     // distributes NumHomes homes to Dispo's units according to a random process in which
171     // each unit's expected final amount of homes increases with the corresponding cell's
172     // attractiveness value in Canvas
173     void INITIALIZE_DISPOSITION_FROM_CELLS_1 (canvas Canvas, disposition Dispo, int NumHomes,
174     double HomeAttributionInflation){ // METHOD 1
175 
176         assert (NumHomes <= Dispo->max_possible_inhabitants);
177         Dispo->num_inhabitants = NumHomes;
178         // printf ("%d/%d\n", Dispo->num_inhabitants, Dispo->max_possible_inhabitants);
179 
180         // reset disposition's housing
181         for (int i = 0; i < Canvas->height_cells; i++){
182             for (int j = 0; j < Canvas->width_cells; j++){
183 
184                 Dispo->unitsA[i][j].num_homes = 0;
185             }
186         }
```

```
187         // sort cell indices by descending attractivenesses and extract highest attractiveness
188         SORT_CELL_INDICES (Canvas);
189 
190         int BestCell_i, BestCell_j;
191         ASSIGN_DELIN_INDEX (&BestCell_i, &BestCell_j, Canvas,
192         Canvas->sorted_cell_indices_attractiveness[0]);
193         double HighestAttractiveness = Canvas->cellsA[BestCell_i][BestCell_j].attractiveness;
194 
195         int NumRemainingHomes = NumHomes;
196         int pos = 0;
197         int ci, cj;
198 
199         // distribute homes by prioritising high-attractiveness cells
200         while (NumRemainingHomes > 0){
201 
202             ASSIGN_DELIN_INDEX (&ci, &cj, Canvas, Canvas->sorted_cell_indices_attractiveness[pos]);
203             if (Canvas->cellsA[ci][cj].service_presence_class == -1){
204 
205                 int NumReceivedHomes = min (min (NumRemainingHomes, cCellHomeLimit), rand () %
206                 /**/ ((int)(ceil ((double)cCellHomeLimit)*F_HOME_ATTRIBUTION_PROPORTION
207                 /**/ (Canvas->cellsA[ci][cj].attractiveness, HighestAttractiveness, Dispo,
208                 /**/ HomeAttributionInflation))+ 1));
209                 Dispo->unitsA[ci][cj].num_homes += NumReceivedHomes;
210                 NumRemainingHomes -= NumReceivedHomes;
211                 // printf ("%d\n", NumReceivedHomes);
212             }
213             pos = (pos + 1) % (Canvas->width_cells*Canvas->height_cells);
214         }
215     }
216 
217 
218     // hyp: Dispo is a disposition on Canvas && CALCULATE_ATTRACTIVENESSES (Canvas) has
219     // been called
220     // distributes NumHomes homes to Dispo's units according to a random process in which
221     // each unit's expected final amount of homes increases with the corresponding cell's
222     // attractiveness value in Canvas
223     void INITIALIZE_DISPOSITION_FROM_CELLS_2 (canvas Canvas, disposition Dispo, int NumHomes,
224     /**/ double HomeAttributionInflation){ // METHOD 2
225 
226         assert (NumHomes <= Dispo->max_possible_inhabitants);
227         Dispo->num_inhabitants = NumHomes;
228         // printf ("%d/%d\n", Dispo->num_inhabitants, Dispo->max_possible_inhabitants);
229 
230         // reset disposition's housing
```

```
231         for (int i = 0; i < Canvas->height_cells; i++){
232             for (int j = 0; j < Canvas->width_cells; j++){
233 
234                 Dispo->unitsA[i][j].num_homes = 0;
235             }
236         }
237         // sort cell indices by descending attractivenesses and extract highest attractiveness
238         SORT_CELL_INDICES (Canvas);
239 
240         int BestCell_i, BestCell_j;
241         ASSIGN_DELIN_INDEX (&BestCell_i, &BestCell_j, Canvas,
242         /**/ Canvas->sorted_cell_indices_attractiveness[0]);
243         double HighestAttractiveness = Canvas->cellsA[BestCell_i][BestCell_j].attractiveness;
244 
245         int NumRemainingHomes = NumHomes;
246         int pos = 0;
247         int ci, cj;
248 
249         // distribute homes by prioritising high-attractiveness cells
250         while (NumRemainingHomes > 0){
251 
252             ASSIGN_DELIN_INDEX (&ci, &cj, Canvas,
253             /**/ Canvas->sorted_cell_indices_attractiveness[pos]);
254             if (Canvas->cellsA[ci][cj].service_presence_class == -1){
255 
256                 double AttributionPeak = F_HOME_ATTRIBUTION_PROPORTION
257                 /**/ (Canvas->cellsA[ci][cj].attractiveness, HighestAttractiveness, Dispo,
258                 /**/ HomeAttributionInflation);
259 
260                 int NumReceivedHomes = min (min (NumRemainingHomes,
261                 /**/ cCellHomeLimit-Dispo->unitsA[ci][cj].num_homes),
262                 /**/ max (1, (int)ceil((double)cCellHomeLimit* // previously, max (0, ...)
263                 /**/ RAND_VAR_HALF_NORMAL_DISTRIBUTION (AttributionPeak, 0.25*AttributionPeak))));
264 
265                 Dispo->unitsA[ci][cj].num_homes += NumReceivedHomes;
266                 NumRemainingHomes -= NumReceivedHomes;
267             }
268             pos = (pos + 1) % (Canvas->width_cells*Canvas->height_cells);
269         }
270     }
271 
272 
273     void INITIALIZE_DISPOSITION_BLINDLY (canvas Canvas, disposition Dispo, int NumHomes){
274 
```

```
275         assert (NumHomes <= Dispo->max_possible_inhabitants);
276         Dispo->num_inhabitants = NumHomes;
277 
278         // reset disposition's housing
279         for (int i = 0; i < Canvas->height_cells; i++){
280             for (int j = 0; j < Canvas->width_cells; j++){
281 
282                 Dispo->unitsA[i][j].num_homes = 0;
283             }
284         }
285 
286         int NumRemainingHomes = NumHomes;
287         while (NumRemainingHomes > 0){
288 
289             int i = rand () % Canvas->height_cells;
290             int j = rand () % Canvas->width_cells;
291 
292             if (Canvas->cellsA[i][j].service_presence_class == -1){
293                 int NumReceivedHomes = min (cCellHomeLimit - Dispo->unitsA[i][j].num_homes,
294                 rand () % (cCellHomeLimit+1));
295 
296                 Dispo->unitsA[i][j].num_homes += NumReceivedHomes;
297                 NumRemainingHomes -= NumReceivedHomes;
298             }
299         }
300 
301     }
302 
303     // returns the amount of homes for which a unit with NumHomes houses accounts for
304     // in the density calculation of a unit SqDist pixels away
305     double F_DENSITY_CONTRIBUTION (double SqDist, int NumHomes){
306 
307         return ((double)NumHomes)*exp(-SqDist/10000);
308     }
309 
310 
311     // hyp: Dispo->num_inhabitants has been assigned the intented number of homes to place
312     // returns the fraction of cCellHomeLimit's homes given to a cell of attractiveness
313     // Attractiveness, living in a canvas of highest attractiveness HighestAttractiveness
314     // and depending on Dispo's intended and maximum number of houses
315     // (see paper for more precise description of parameter influences)
316     double F_HOME_ATTRIBUTION_PROPORTION (double Attractiveness, double HighestAttractiveness,
317     /**/ disposition Dispo, double HomeAttributionInflation){
318 
```

```
319         double v = pow(((double)Dispo->max_possible_inhabitants)/(2*(double)Dispo->num_inhabitants),
320         /**/ HomeAttributionInflation)*(Attractiveness-HighestAttractiveness);
321         double u = 1 + v/(15*pow(HighestAttractiveness, 0.75));
322         double w = 1.03*u*exp(-((875-1000*u)/300)*((875-1000*u)/300)*((875-1000*u)/300)*((875-1000*u)/300));
323         return max_double (0, w);
324     }
325
326
327     double F_ENTROPY_CONTRIBUTION (double Ratio){
328         if (Ratio == 0) return 0;
329         else return -Ratio*log(Ratio);
330     }
331
332
333     // hyp: Dispo is a disposition on Canvas
334     // calculates and assigns to each Dispo->unit[i][j] its local house density, i.e.
335     // the distance-weighted sum of the number of homes contained in cells within distance
336     // cDensityToleranceRadius of said unit
337     void CALCULATE_LOCAL_DENSITIES (canvas Canvas, disposition Dispo){ // METHOD 1
338
339         for (int i = 0; i < Dispo->height_units; i++){
340             for (int j = 0; j < Dispo->width_units; j++){
341
342                 if (Canvas->cellsA[i][j].service_presence_class == -1){
343
344                     Dispo->unitsA[i][j].local_density = 0;
345
346                     int UnitX, UnitY;
347                     ASSIGN_MIDPOINT_OF_CELL (&UnitX, &UnitY, Canvas, i, j);
348
349                     // find range of pixels that are for sure within distance cDensityToleranceRadius
350                     // (forming a square of sidelength 2*cDensityToleranceRadius)
351                     int X_min = max (UnitX - (int)cDensityToleranceRadius, 0);
352                     int X_max = min (UnitX + (int)cDensityToleranceRadius, cWindowDims[0]-1);
353                     int Y_min = max (UnitY - (int)cDensityToleranceRadius, 0);
354                     int Y_max = min (UnitY + (int)cDensityToleranceRadius, cWindowDims[1]-1);
355                     int i_min, i_max, j_min, j_max;
356
357                     // convert to unit indices
358                     ASSIGN_CELL_OF_PIXEL (&i_min, &j_min, Canvas, X_min, Y_min);
359                     ASSIGN_CELL_OF_PIXEL (&i_max, &j_max, Canvas, X_max, Y_max);
360
361                     // add up unit contributions
362                     for (int row = i_min; row <= i_max; row++){
```

```
363                     for (int column = j_min; column <= j_max; column++){
364
365                         int CurrentX, CurrentY;
366                         ASSIGN_MIDPOINT_OF_CELL (&CurrentX, &CurrentY, Canvas, row, column);
367                         double SqDist = (CurrentX-UnitX)*(CurrentX-UnitX)
368                         /**/ + (CurrentY-UnitY)*(CurrentY-UnitY);
369
370                         // count contribution iff the distance is actually less than
371                         // cDensityToleranceRadius
372                         if (SqDist <= cDensityToleranceRadius*cDensityToleranceRadius)
373                         Dispo->unitsA[i][j].local_density += F_DENSITY_CONTRIBUTION
374                         /**/ (SqDist, Dispo->unitsA[row][column].num_homes);
375                     }
376                 }
377             }
378         }
379     }
380 }
381
382
383 void CALCULATE_LOCAL_ENTROPIES (canvas Canvas, disposition Dispo){
384
385     int* HomeCountOccurrences = (int*) malloc ((cCellHomeLimit+1)*sizeof(int));
386     Dispo->entropy = 0;
387
388     for (int i = 0; i < Dispo->height_units; i++){
389         for (int j = 0; j < Dispo->width_units; j++){
390
391             if (Canvas->cellsA[i][j].service_presence_class == -1){
392
393                 int NumCellsInRange = 0;
394                 for (int num_homes = 0; num_homes <= cCellHomeLimit; num_homes++){
395
396                     HomeCountOccurrences[num_homes] = 0;
397                 }
398
399                 Dispo->unitsA[i][j].local_entropy = 0;
400
401                 int UnitX, UnitY;
402                 ASSIGN_MIDPOINT_OF_CELL (&UnitX, &UnitY, Canvas, i, j);
403
404                 // find range of pixels that are for sure within distance cEntropyToleranceRadius
405                 // (forming a square of sidelength 2*cEntropyToleranceRadius)
406                 int X_min = max (UnitX - (int)cEntropyToleranceRadius, 0);
```

```
407                 int X_max = min (UnitX + (int)cEntropyToleranceRadius, cWindowDims[0]-1);
408                 int Y_min = max (UnitY - (int)cEntropyToleranceRadius, 0);
409                 int Y_max = min (UnitY + (int)cEntropyToleranceRadius, cWindowDims[1]-1);
410                 int i_min, i_max, j_min, j_max;
411
412                 // convert to unit indices
413                 ASSIGN_CELL_OF_PIXEL (&i_min, &j_min, Canvas, X_min, Y_min);
414                 ASSIGN_CELL_OF_PIXEL (&i_max, &j_max, Canvas, X_max, Y_max);
415
416                 // add up entropy contributions contributions
417                 for (int row = i_min; row <= i_max; row++){
418                     for (int column = j_min; column <= j_max; column++){
419
420                         int CurrentX, CurrentY;
421                         ASSIGN_MIDPOINT_OF_CELL (&CurrentX, &CurrentY, Canvas, row, column);
422                         double SqDist = (CurrentX-UnitX)*(CurrentX-UnitX)
423                         /**/ + (CurrentY-UnitY)*(CurrentY-UnitY);
424
425                         // count contribution iff the distance is actually less than
426                         // cEntropyToleranceRadius
427                         if (SqDist <= cEntropyToleranceRadius*cEntropyToleranceRadius &&
428                         /**/ Canvas->cellsA[row][column].service_presence_class == -1){
429
430                             HomeCountOccurrences[Dispo->unitsA[row][column].num_homes]++;
431                             NumCellsInRange++;
432                         }
433                     }
434                 }
435
436                 for (int num_homes = 0; num_homes <= cCellHomeLimit; num_homes++){
437
438                     double Proportion = (double)HomeCountOccurrences[num_homes]
439                     /**/ /(double)NumCellsInRange;
440                     Dispo->unitsA[i][j].local_entropy += F_ENTROPY_CONTRIBUTION (Proportion);
441                 }
442
443                 Dispo->entropy += Dispo->unitsA[i][j].local_entropy;
444             }
445         }
446     }
447     /* printf ("freeing homecountoccurrences\n");
448     free (HomeCountOccurrences); */
449 }
```

# display.h

```
1   #ifndef DISPLAY_H
2   #define DISPLAY_H
3
4   #include "canvas.h"
5   #include "dispo.h"
6   #include "terrain.h"
7
8
9
10  // ----- GLOBAL CONSTANTS ----- //
11
12  extern const int cGridlineColor[4];
13
14  extern const int cServiceDisplayColors[7][3]; // should have size cNumServiceClass x 3
15  extern const int cTerrainDisplayColors[9][3];
16
17  extern const double cTerrainCosts[9];
18
19  extern const double cDeepWaterHeight;
20  extern const double cShallowWaterHeight;
21  extern const double cLowGrassHeight;
22  extern const double cNormalGrassHeight;
23  extern const double cHighGrassHeight;
24  extern const double cHillsHeight;
25  extern const double cLowMountainsHeight;
26  extern const double cElevatedMountainsHeight;
27  extern const double cMountainPeaksHeight;
28
29
30  // ----- FUNCTIONS ----- //
31
32  // utilitary
33  void ASSIGN_COLOR_OF_SERVICE_CLASS (int class, int* pRed, int* pGreen, int* pBlue);
34  void ASSIGN_COLOR_OF_TERRAIN_TYPE (int type, int* pRed, int* pGreen, int* pBlue);
35  int TERRAIN_TYPE_OF_HEIGHT (double Height);
36
37  // displaying attributes
```

```
38  void SDL_DisplayCanvasGrid (SDL_Renderer* Renderer, canvas Canvas);
39  void SDL_DisplayTerrainHeights (SDL_Renderer* Renderer, canvas Canvas);
40  void SDL_DisplayCellAttractivenesses (SDL_Renderer* Renderer, canvas Canvas);
41  void SDL_DisplayUnitDensities (SDL_Renderer* Renderer, canvas Canvas, disposition Dispo);
42  void SDL_DisplayUnitEntropies (SDL_Renderer* Renderer, canvas Canvas, disposition Dispo);
43
44  // rendering objects
45  void SDL_RenderExistingServices (SDL_Renderer* Renderer, canvas Canvas);
46  void SDL_RenderExistingHomes (SDL_Renderer* Renderer, canvas Canvas, disposition Dispo);
47  void SDL_RenderTerrain (SDL_Renderer* Renderer, canvas Canvas);
48
49  // filling
50  void SDL_FloodFillService (SDL_Renderer* Renderer, canvas Canvas, int class, int i, int j);
51
52
53
54  // unused
55  void SDL_DoNothing (void);
56
57
58  #endif
```

## display.c

```
1   #include "display.h"
2
3   const int cWindowDims[2] = {1920,1080};
4   const int cGridlineColor[4] = {255, 255, 255, 30};
5   const int cServiceDisplayColors[7][3] = {
6       {239,149,91},
7       {157,197,105},
8       {52,167,80},
9       {122,185,190},
10      {136,113,158},
11      {133,30,42},
12      {100,23,17}
13  };
14
15
16  const int cTerrainDisplayColors[9][3] = {
17      {0,36,172},
```

```
18          {0,177,249},
19          {51,236,56},
20          {139,218,45},
21          {165,180,55},
22          {162,128,49},
23          {149,117,71},
24          {190,173,136},
25          {229,240,245}
26      };
27
28      const double cTerrainCosts[9] = {2000,1900,80,50,60,90,120,160,230};
29
30      const double cDeepWaterHeight = 0.25;
31      const double cShallowWaterHeight = 0.28;
32      const double cLowGrassHeight = 0.38;
33      const double cNormalGrassHeight = 0.52;
34      const double cHighGrassHeight = 0.65;
35      const double cHillsHeight = 0.75;
36      const double cLowMountainsHeight = 0.85;
37      const double cElevatedMountainsHeight = 0.92;
38      const double cMountainPeaksHeight = 1;
39
40      // hyp: 0 <= class < cNumServiceClasses
41      // assigns to pRed, pGreen and pBlue the color coordinates of service class class's
42      // display color as determined by cServiceDisplayColors
43      void ASSIGN_COLOR_OF_SERVICE_CLASS (int class, int* pRed, int* pGreen, int* pBlue){
44
45          *pRed = cServiceDisplayColors[class][0];
46          *pGreen = cServiceDisplayColors[class][1];
47          *pBlue = cServiceDisplayColors[class][2];
48      }
49
50
51      // hyp : 0 <= type < 9
52      // assigns to pRed, pGreen and pBlue the color values of terrain type type's display
53      // color as determined by cTerrainDisplayColors
54      void ASSIGN_COLOR_OF_TERRAIN_TYPE (int type, int* pRed, int* pGreen, int* pBlue){
55
56          *pRed = cTerrainDisplayColors[type][0];
57          *pGreen = cTerrainDisplayColors[type][1];
58          *pBlue = cTerrainDisplayColors[type][2];
59      }
60
61
```

```
62  int TERRAIN_TYPE_OF_HEIGHT (double Height){
63      if (Height <= cDeepWaterHeight) return 0;
64      else if (Height <= cShallowWaterHeight) return 1;
65      else if (Height <= cLowGrassHeight) return 2;
66      else if (Height <= cNormalGrassHeight) return 3;
67      else if (Height <= cHighGrassHeight) return 4;
68      else if (Height <= cHillsHeight) return 5;
69      else if (Height <= cLowMountainsHeight) return 6;
70      else if (Height <= cElevatedMountainsHeight) return 7;
71      else if (Height <= cMountainPeaksHeight) return 8;
72  }
73
74
75  // displays Canvas's grid with lines of color coordinates cGridlineColor
76  void SDL_DisplayCanvasGrid (SDL_Renderer* Renderer, canvas Canvas){
77
78      SDL_SetRenderDrawColor (Renderer, cGridlineColor[0], cGridlineColor[1],
79      /**/ cGridlineColor[2], cGridlineColor[3]);
80      // draw horizontal lines
81      for (int i = 0; i < Canvas->height_cells; i++)
82      SDL_RenderDrawLine (Renderer, 0, i*Canvas->cell_size, cWindowDims[0], i*Canvas->cell_size);
83      // draw vertical lines
84      for (int j = 0; j < Canvas->width_cells; j++)
85      SDL_RenderDrawLine (Renderer, j*Canvas->cell_size, 0, j*Canvas->cell_size, cWindowDims[1]);
86      SDL_RenderPresent (Renderer);
87  }
88
89
90  // fills each Canvas->cellsA[i][j].fill_square in cyan with an opacity proprtional to the
91  // terrain height at said cell
92  void SDL_DisplayTerrainHeights (SDL_Renderer* Renderer, canvas Canvas){
93
94      for (int i = 0; i < Canvas->height_cells ; i++){
95          for (int j = 0; j < Canvas->width_cells; j++){
96
97              int CyanValue = (int) (255.*Canvas->cellsA[i][j].terrain_height);
98              SDL_SetRenderDrawColor (Renderer, 0, CyanValue, CyanValue, 255);
99              SDL_RenderFillRect (Renderer, &Canvas->cellsA[i][j].fill_square);
100         }
101     }
102     SDL_RenderPresent (Renderer);
103 }
104
105
```

```
106   // displays the individual values of Canvas->cellsA[i][j].attractiveness in red,
107   // with brighter shades corresponding to higher attractiveness levels
108   void SDL_DisplayCellAttractivenesses (SDL_Renderer* Renderer, canvas Canvas){
109
110       for (int i = 0; i < Canvas->height_cells ; i++){
111           for (int j = 0; j < Canvas->width_cells; j++){
112
113               if (Canvas->cellsA[i][j].service_presence_class == -1){
114                   int RedBlueValue = (int) (Canvas->cellsA[i][j].attractiveness*(255./1000.));
115                   if (RedBlueValue < 0){
116                       SDL_SetRenderDrawColor (Renderer, 0, 0, -RedBlueValue, 255);
117                   }
118                   else
119                   SDL_SetRenderDrawColor (Renderer, RedBlueValue, 0, 0, 255);
120                   SDL_RenderFillRect (Renderer, &Canvas->cellsA[i][j].fill_square);
121               }
122           }
123       }
124       SDL_RenderPresent (Renderer);
125   }
126
127
128   // fills each Canvas->cellsA[i][j].fill_square in green with an opacity proportional
129   // to the local densities, i.e. Dispo->unitsA[i][j].local_density
130   void SDL_DisplayUnitDensities (SDL_Renderer* Renderer, canvas Canvas, disposition Dispo){
131
132       for (int i = 0; i < Canvas->height_cells ; i++){
133           for (int j = 0; j < Canvas->width_cells; j++){
134
135               int GreenValue = (int) (255.0*(Dispo->unitsA[i][j].local_density)
136               /**/ /Dispo->max_possible_density);
137               SDL_SetRenderDrawColor (Renderer, 0, GreenValue, 0, 255);
138               SDL_RenderFillRect (Renderer, &Canvas->cellsA[i][j].fill_square);
139           }
140       }
141       SDL_RenderPresent (Renderer);
142   }
143
144
145   // fills each Canvas->cellsA[i][j].fill_square in magenta with an opacity proportional
146   // to the local entropy value stored in Dispo->unitsA[i][j]
147   void SDL_DisplayUnitEntropies (SDL_Renderer* Renderer, canvas Canvas, disposition Dispo){
148
149       double MaxEntropy = log(cCellHomeLimit);
```

```
150
151         for (int i = 0; i < Canvas->height_cells ; i++){
152             for (int j = 0; j < Canvas->width_cells; j++){
153
154                 int MagentaValue = (int) (255.0*(Dispo->unitsA[i][j].local_entropy)/MaxEntropy);
155                 SDL_SetRenderDrawColor (Renderer, MagentaValue, 0, MagentaValue, 255);
156                 SDL_RenderFillRect (Renderer, &Canvas->cellsA[i][j].fill_square);
157             }
158         }
159         SDL_RenderPresent (Renderer);
160     }
161
162
163     // renders the services which have been drawn and filled on Canvas
164     void SDL_RenderExistingServices (SDL_Renderer* Renderer, canvas Canvas){
165
166         l_list BorderPixels = Canvas->service_border_pixelsL;
167         l_list InteriorCells = Canvas->service_interior_cellsL;
168         int i, j;
169         int Red, Green, Blue;
170
171         // render borders
172         while (BorderPixels != NULL){
173
174             ASSIGN_COLOR_OF_SERVICE_CLASS (BorderPixels->spec, &Red, &Green, &Blue);
175             SDL_SetRenderDrawColor (Renderer, Red, Green, Blue, 255);
176             ASSIGN_CELL_OF_PIXEL (&i, &j, Canvas, BorderPixels->x, BorderPixels->y);
177             SDL_RenderFillRect (Renderer, &Canvas->cellsA[i][j].fill_square);
178
179             BorderPixels = BorderPixels->next;
180         }
181         // render interiors
182         while (InteriorCells != NULL){
183
184             ASSIGN_COLOR_OF_SERVICE_CLASS (InteriorCells->spec, &Red, &Green, &Blue);
185             SDL_SetRenderDrawColor (Renderer, Red, Green, Blue, 255);
186             SDL_RenderFillRect (Renderer, &Canvas->cellsA[InteriorCells->x][InteriorCells->y].fill_square);
187
188             InteriorCells = InteriorCells->next;
189         }
190         SDL_RenderPresent (Renderer);
191     }
192
193
```

```
194  // colors each cell Canvas->cellsA[i][j] in white with an opacity proportional to
195  // the number of homes contained in Dispo->unitsA[i][j]
196  void SDL_RenderExistingHomes (SDL_Renderer* Renderer, canvas Canvas, disposition Dispo){
197
198      for (int i = 0; i < Canvas->height_cells ; i++){
199          for (int j = 0; j < Canvas->width_cells; j++){
200
201              int WhiteValue = (int) (255.0*((double) Dispo->unitsA[i][j].num_homes)
202              /**/ /(double)cCellHomeLimit);
203              SDL_SetRenderDrawColor (Renderer, WhiteValue, WhiteValue, WhiteValue, 255);
204              SDL_RenderFillRect (Renderer, &Canvas->cellsA[i][j].fill_square);
205          }
206      }
207      SDL_RenderPresent (Renderer);
208  }
209
210
211  void SDL_RenderTerrain (SDL_Renderer* Renderer, canvas Canvas){
212
213      int Red, Green, Blue;
214
215      for (int i = 0; i < Canvas->height_cells ; i++){
216          for (int j = 0; j < Canvas->width_cells; j++){
217
218              int TerrainType = TERRAIN_TYPE_OF_HEIGHT (Canvas->cellsA[i][j].terrain_height);
219              ASSIGN_COLOR_OF_TERRAIN_TYPE (TerrainType, &Red, &Green, &Blue);
220
221              SDL_SetRenderDrawColor (Renderer, Red, Green, Blue, 50);
222              SDL_RenderFillRect (Renderer, &Canvas->cellsA[i][j].fill_square);
223          }
224      }
225      SDL_RenderPresent (Renderer);
226  }
227
228
229  // hyp: the current SDL render draw color is the display color of service class class
230  // && cell (i,j) of Canvas is surrounded by a closed boundary of the same class
231  // if Canvas->cellsA[i][j] is a valid cell, fills it, marks it as occupied by service class
232  // class and adds it to updates Canvas->service_interior_cellsL
233  // then, applies itself recursively on all neighboring cells which have not yet been
234  // marked as occupied by class, i.e. which have yet to be filled
235  void SDL_FloodFillService (SDL_Renderer* Renderer, canvas Canvas, int class, int i, int j){
236
237      if (IS_CELL_IN_CANVAS (Canvas, i, j)){
```

```
238
239             // fill starting cell and update canvas cell attributes and service interiors list
240             SDL_RenderFillRect (Renderer, &Canvas->cellsA[i][j].fill_square);
241             Canvas->cellsA[i][j].service_presence_class = class;
242             l_insert (&Canvas->service_interior_cellsL, i, j, class);
243
244             // recursive calls to all four neighboring cells
245             if (Canvas->cellsA[i+1][j].service_presence_class != class){
246                 Canvas->cellsA[i+1][j].service_presence_class = class;
247                 SDL_FloodFillService (Renderer, Canvas, class, i+1, j);
248             }
249             if (Canvas->cellsA[i][j+1].service_presence_class != class){
250                 Canvas->cellsA[i+1][j].service_presence_class = class;
251                 SDL_FloodFillService (Renderer, Canvas, class, i, j+1);
252             }
253             if (Canvas->cellsA[i][j-1].service_presence_class != class){
254                 Canvas->cellsA[i+1][j].service_presence_class = class;
255                 SDL_FloodFillService (Renderer, Canvas, class, i, j-1);
256             }
257             if (Canvas->cellsA[i-1][j].service_presence_class != class){
258                 Canvas->cellsA[i+1][j].service_presence_class = class;
259                 SDL_FloodFillService (Renderer, Canvas, class, i-1, j);
260             }
261         }
262     }
263
264
265     // literally does nothing (made for convenience and ease of operation of SDL keypress
266     // registers)
267     void SDL_DoNothing (void){}
```

## gen-algo.h

```
1   #ifndef GENALGO_H
2   #define GENALGO_H
3
4   #include "canvas.h"
5   #include "dispo.h"
6   #include "terrain.h"
7   #include "display.h"
8
```

```
 9
10  struct individual_s {
11      disposition dispo;
12
13      double potential_exploitation_ratio;
14
15      double density_score;
16      double attractiveness_score;
17      double entropy_score;
18      double cost_penalty;
19
20      double total_score;
21  };
22
23  struct vpopulation_s {
24      int num_individuals;
25      struct individual_s* individualsA;
26      int* sorted_individual_indices_score;
27  };
28
29
30  typedef struct individual_s individual;
31  typedef struct vpopulation_s* population;
32
33
34  extern const double cOptimalDensityToMaxDensityRatio;
35
36  extern const double cScoreFactorWeights[2];
37
38  extern const double cMutationHomeDisplacementRadius; //add: pixels
39
40  extern SDL_Renderer* RendererGENALGO;
41
42  // double F_CELL_ATTRACTIVENESS_EXPLOITATION (canvas Canvas, disposition Dispo);
43
44  // elementary score functions
45  double F_DENSITY_SCORE_FROM_UNIT (unit Unit, double MaxDensity);
46  double F_ATTRACTIVENESS_SCORE_FROM_UNIT_CELL (unit Unit, cell Cell);
47  double F_ENTROPY_SCORE (unit Unit);
48
49  // generating and evaluating individuals
50  void INITIALIZE_INDIVIDUAL (individual* pIndiv, canvas Canvas, int NumHomes);
51  void CALCULATE_SCORES_OF_INDIVIDUAL (individual* pIndiv, canvas Canvas); // invert
52  /**/ // argument order
```

```
53
54  // creating populations and ensuring correctness
55  population CREATE_POPULATION (canvas Canvas, int PopSize);
56  void SORT_INDIVIDUAL_INDICES (population Popl);
57
58  // genetic processes
59  individual* MUTATE_INDIVIDUAL (canvas Canvas, individual Indiv, double ProportionMutatedUnits);
60  individual* CROSS_INDIVIDUALS (canvas Canvas, individual Parent1, individual Parent2,
61  /**/ double MaxProportionOfFirst, double StdDeviation);
62  void RENEW_POPULATION (canvas Canvas, population Popl, double KeepProportion,
63  /**/ double MutateProportion, double CrossProportion);
64
65  /* population GENERATE_RANDOM_POPULATION (canvas Canvas, int PopSize, int NumHomes);
66  population SELECT_FITTEST_INDIVIDUALS (population Popl);
67  individual* MUTATE_INDIVIDUAL (canvas Canvas, individual Indiv, double ProportionMutatedUnits); */
68
69  // void ITERATE_POPULATION_RENEWAL (canvas Canvas, population Popl, double ConservationProportion);
70
71
72
73  void FREE_INDIVIDUAL (individual* pIndiv);
74
75  #endif
```

# gen-algo.c

```
1   #include "gen-algo.h"
2
3   const double cOptimalDensityToMaxDensityRatio = 0.15;
4
5   const double cMutationHomeDisplacementRadius = 800;
6
7   SDL_Renderer* RendererGENALGO;
8
9
10  // returns the signed density score contribution of Unit, which accounts for its local
11  // density in comparison to the theoretical maximum possible density in a manner dictated
12  // by cOptimalDensityToMaxDensityRatio, and is proportional to Unit.num_homes
13  double F_DENSITY_SCORE_FROM_UNIT (unit Unit, double MaxDensity){
14
15      double y = (0.96/cOptimalDensityToMaxDensityRatio)*(Unit.local_density/MaxDensity);
```

```
16          return 0.25*Unit.num_homes*((1.8*exp(-(y-1)*(y-1))) - 1./(1.+exp(-y-0.5)));
17      }
18
19      // hyp : Unit's corresponding canvas cell is Cell
20      // returns the attractiveness score contribution of Unit based on Cell's attractiveness
21      double F_ATTRACTIVENESS_SCORE_FROM_UNIT_CELL (unit Unit, cell Cell){
22
23          return 1.5*Unit.num_homes*(Cell.attractiveness/1000);
24      }
25
26
27      // returns the entropy score ontribution of a Unit
28      double F_ENTROPY_SCORE (unit Unit){
29
30          double u = 6*Unit.local_entropy/log(cCellHomeLimit);
31          return 0.5*(1.1*exp(-u*u*u/10)-0.12*(u+exp(-1))*log(u+exp(-1))-1);
32      }
33
34
35      double F_COST_PENALTY (unit Unit, cell Cell, int NumHomes){
36
37          double CostCoeff = cTerrainCosts[TERRAIN_TYPE_OF_HEIGHT (Cell.terrain_height)];
38          return -(0.001*CostCoeff)*(double)Unit.num_homes*(1-Cell.terrain_height)
39          /**/ *(1-Cell.terrain_height)*0;
40      }
41
42
43      // intiializes an individual whose disposition is obtained by calling CREATE_CANVAS_
44      // DISPOSITION with a random HomeAttributionInflation parameter
45      // then, calculates said individuals's scores (therefore calculating it's disposition's
46      // densities and entropies) before returning it
47      void INITIALIZE_INDIVIDUAL (individual* pIndiv, canvas Canvas, int NumHomes){
48
49          pIndiv->potential_exploitation_ratio = 0;
50          pIndiv->density_score = 0;
51          pIndiv->attractiveness_score = 0;
52          pIndiv->total_score = -DBL_MAX;
53
54          pIndiv->dispo = CREATE_CANVAS_DISPOSITION (Canvas);
55
56          double RandHomeAttributionInflation = 2.*((double)rand()/(double)RAND_MAX);
57
58          INITIALIZE_DISPOSITION_BLINDLY (Canvas, pIndiv->dispo,
59          /**/ NumHomes); //, RandHomeAttributionInflation);
```

```
60
61          CALCULATE_LOCAL_DENSITIES (Canvas, pIndiv->dispo);
62          CALCULATE_LOCAL_ENTROPIES (Canvas, pIndiv->dispo);
63          CALCULATE_SCORES_OF_INDIVIDUAL (pIndiv, Canvas);
64      }
65
66
67      // calculates and assigns to *pIndiv its attractiveness, density and total score if the
68      // latter has not yet been calculated, i.e. has its initializing value -DBL_MAX
69      void CALCULATE_SCORES_OF_INDIVIDUAL (individual* pIndiv, canvas Canvas){
70
71          if (pIndiv->total_score == -DBL_MAX){
72
73              pIndiv->density_score = 0;
74              pIndiv->attractiveness_score = 0;
75              pIndiv->entropy_score = 0;
76              pIndiv->cost_penalty = 0;
77
78              for (int i = 0; i < pIndiv->dispo->height_units; i++){
79                  for (int j = 0; j < pIndiv->dispo->width_units; j++){
80
81                      if (pIndiv->dispo->unitsA[i][j].num_homes > 0){
82
83                          pIndiv->density_score += F_DENSITY_SCORE_FROM_UNIT (pIndiv->dispo->unitsA[i][j],
84                          /**/ pIndiv->dispo->max_possible_density);
85                          pIndiv->attractiveness_score += F_ATTRACTIVENESS_SCORE_FROM_UNIT_CELL
86                          /**/ (pIndiv->dispo->unitsA[i][j], Canvas->cellsA[i][j]);
87                          // printf ("%f | ", pIndiv->density_score);
88                          pIndiv->entropy_score += F_ENTROPY_SCORE (pIndiv->dispo->unitsA[i][j]);
89                          pIndiv->cost_penalty += F_COST_PENALTY (pIndiv->dispo->unitsA[i][j],
90                          /**/ Canvas->cellsA[i][j], pIndiv->dispo->num_inhabitants);
91                      }
92                  }
93              }
94              pIndiv->total_score = pIndiv->density_score + pIndiv->attractiveness_score
95              /**/ + pIndiv->entropy_score + pIndiv->cost_penalty;
96          }
97      }
98
99
100     // allocates memory for a population of PopSize individuals and returns the corresponding
101     // pointer
102     population CREATE_POPULATION (canvas Canvas, int PopSize){
103
```

```
104         population Popl = (population) malloc (sizeof(struct vpopulation_s));
105 
106         Popl->num_individuals = PopSize;
107         Popl->individualsA = (individual*) malloc (PopSize*sizeof(individual));
108         Popl->sorted_individual_indices_score = (int*) malloc (PopSize*sizeof(int));
109 
110         for (int index = 0; index < PopSize; index++){
111 
112             Popl->sorted_individual_indices_score[index] = index;
113         }
114 
115         return Popl;
116     }
117 
118 
119     // sorts Popl->sorted_individual_indices_score such that the sequence
120     // (Popl->individualsA[k]) increases in k
121     void SORT_INDIVIDUAL_INDICES (population Popl){
122 
123         int* IndividualIndices = Popl->sorted_individual_indices_score;
124         double* IndividualScores = (double*) malloc (Popl->num_individuals*sizeof(double));
125 
126         for (int index = 0; index < Popl->num_individuals; index++){
127 
128             IndividualScores[index] = Popl->individualsA[IndividualIndices[index]].total_score;
129         }
130 
131         QS_SORT_ELEMENTS_BY_SCORE (IndividualIndices, IndividualScores, 0,
132         /**/Popl->num_individuals-1);
133 
134         /* printf ("freeing individualscores\n");
135         free (IndividualScores); */
136     }
137 
138     individual* MUTATE_INDIVIDUAL (canvas Canvas, individual Indiv, double ProportionMutatedUnits){
139 
140         SORT_UNIT_INDICES (Canvas, Indiv.dispo);
141 
142         individual* pMutatedIndiv = (individual*) malloc (sizeof(individual));
143         pMutatedIndiv->dispo = COPY_DISPOSITION (Canvas, Indiv.dispo);
144 
145         // number of units to transfer homes from
146         int NumRemainingUnits = (int) ceil (ProportionMutatedUnits*Indiv.dispo->height_units
147         /**/ *Indiv.dispo->width_units);
```

```
148         // printf ("%d to mutate\n (proportion: %f)", NumRemainingUnits, ProportionMutatedUnits);
149 
150         int pos = 0;
151         int ci, cj;
152 
153         while (NumRemainingUnits > 0){
154 
155             ASSIGN_DELIN_INDEX (&ci, &cj, Canvas, Indiv.dispo->sorted_unit_indices_density[pos]);
156 
157             int UnitX, UnitY;
158             ASSIGN_MIDPOINT_OF_CELL (&UnitX, &UnitY, Canvas, ci, cj);
159 
160             double RandDouble = ((double)rand()/(double)RAND_MAX);
161             if (RandDouble <= Indiv.dispo->unitsA[ci][cj].local_density){
162 
163                 int X_min = max (UnitX - cMutationHomeDisplacementRadius, 0);
164                 int X_max = min (UnitX + cMutationHomeDisplacementRadius, cWindowDims[0]-1);
165                 int Y_min = max (UnitY - cMutationHomeDisplacementRadius, 0);
166                 int Y_max = min (UnitY + cMutationHomeDisplacementRadius, cWindowDims[1]-1);
167 
168                 int DestinationX, DestinationY;
169                 double SqDist = DBL_MAX;
170 
171                 while (SqDist > cMutationHomeDisplacementRadius){
172 
173                     DestinationX = rand () % (X_max - X_min + 1) + X_min;
174                     DestinationY = rand () % (Y_max - Y_min + 1) + Y_min;
175                     SqDist = (DestinationX-UnitX)*(DestinationX-UnitX) +
176                     /**/ (DestinationY-UnitY)*(DestinationY-UnitY);
177                 }
178                 int Destination_i, Destination_j;
179                 ASSIGN_CELL_OF_PIXEL (&Destination_i, &Destination_j, Canvas,
180                 /**/ DestinationX, DestinationY);
181 
182                 int NumTransferredHomes = rand () % (cCellHomeLimit+1);
183                 if (TRANSFER_HOMES (Canvas, pMutatedIndiv->dispo, NumTransferredHomes,
184                 /**/ ci, cj, Destination_i, Destination_j)){
185                     NumRemainingUnits--;
186                 }
187             }
188             pos++;
189         }
190         pMutatedIndiv->total_score = -DBL_MAX;
191         CALCULATE_LOCAL_DENSITIES (Canvas, pMutatedIndiv->dispo);
```

```
192         CALCULATE_LOCAL_ENTROPIES (Canvas, pMutatedIndiv->dispo);
193         CALCULATE_SCORES_OF_INDIVIDUAL (pMutatedIndiv, Canvas);
194         return pMutatedIndiv;
195     }
196
197
198     individual* CROSS_INDIVIDUALS (canvas Canvas, individual Parent1, individual Parent2,
199     /**/ double MaxProportionOfFirst, double StdDeviation){
200
201         /* int NumUnits = Canvas->height_cells*Canvas->width_cells;
202         int* ShuffledUnits = (int*) malloc (NumUnits*sizeof(int));
203         for (int pos = 0; pos < NumUnits; pos++){
204             ShuffledUnits[pos] = pos;
205         }
206         for (int num_swapped = 0; num_swapped < NumSwaps; num_swapped++){
207
208             int pos1 = rand () % (NumUnits);
209             int pos2 = pos1;
210             while (pos2 == pos1){
211                 pos2 = rand () % (NumUnits);
212             }
213
214             int TempIndex = ShuffledUnits[pos1];
215             ShuffledUnits[pos1] = ShuffledUnits[pos2];
216             ShuffledUnits[pos2] = ShuffledUnits[pos1];
217         }
218
219         int end_index = (int)ceil(ProportionUnitsFromFirst*NumUnits); */
220
221         individual* pChildIndiv = (individual*) malloc (sizeof(individual));
222         pChildIndiv->total_score = -DBL_MAX;
223
224         pChildIndiv->dispo = CREATE_CANVAS_DISPOSITION (Canvas);
225         pChildIndiv->dispo->num_inhabitants = Parent1.dispo->num_inhabitants;
226
227         int NumRemainingHomes = Parent1.dispo->num_inhabitants;
228
229         for (int index = 0; index < Canvas->height_cells*Canvas->width_cells &&
230         /**/ NumRemainingHomes > 0; index++){
231
232             int ci, cj;
233             ASSIGN_DELIN_INDEX (&ci, &cj, Canvas, Canvas->sorted_cell_indices_attractiveness
234             /**/ [index]);
235
```

```
236             int NumFromParent1 = Parent1.dispo->unitsA[ci][cj].num_homes;
237             int NumFromParent2 = Parent2.dispo->unitsA[ci][cj].num_homes;
238             double ProportionOfFirst = max_double (0, RAND_VAR_HALF_NORMAL_DISTRIBUTION
239             /**/ (MaxProportionOfFirst, StdDeviation));
240 
241             int NumReceivedHomes = min (cCellHomeLimit, (int)ceil(ProportionOfFirst*NumFromParent1 +
242             /**/ (1-ProportionOfFirst)*NumFromParent2));
243 
244             // printf("Proportion: %lf ; attempted to merge %d from first and %d from second ; yields %d\n",
245             /**/ //ProportionOfFirst, NumFromParent1, NumFromParent2, NumReceivedHomes);
246 
247             pChildIndiv->dispo->unitsA[ci][cj].num_homes = NumReceivedHomes;
248             NumRemainingHomes -= NumReceivedHomes;
249         }
250         pChildIndiv->total_score = -DBL_MAX;
251         CALCULATE_LOCAL_DENSITIES (Canvas, pChildIndiv->dispo);
252         CALCULATE_LOCAL_ENTROPIES (Canvas, pChildIndiv->dispo);
253         CALCULATE_SCORES_OF_INDIVIDUAL (pChildIndiv, Canvas);
254         return pChildIndiv;
255     }
256 
257     void RENEW_POPULATION (canvas Canvas, population Popl, double KeepProportion,
258     /**/ double MutateProportion, double CrossProportion){
259 
260         int NumKeptFromPreviousPop = (int)ceil(KeepProportion*Popl->num_individuals);
261         int NumMutatedFromPreviousPop = (int)ceil(MutateProportion*Popl->num_individuals);
262         int NumCrossedFromPreviousPop = (int)ceil(CrossProportion*Popl->num_individuals);
263         int NumNewlyGenerated = Popl->num_individuals - NumKeptFromPreviousPop
264         /**/ - NumMutatedFromPreviousPop - NumCrossedFromPreviousPop;
265 
266         int NumHomes = Popl->individualsA[0].dispo->num_inhabitants;
267 
268         // keep NumKeptFromPreviousPop and mutate NumMutatedFromPreviousPop of them into
269         // new individuals
270         for (int index = NumKeptFromPreviousPop; index < NumKeptFromPreviousPop
271         /**/ + NumMutatedFromPreviousPop; index++){
272 
273             individual ReferenceIndiv = Popl->individualsA
274             /**/ [Popl->sorted_individual_indices_score[index]];
275             individual* pMutatingIndiv = &Popl->individualsA
276             /**/ [Popl->sorted_individual_indices_score[index]];
277 
278             double RandMutationProportion = 0.15*((double)rand()/(double)RAND_MAX);
279 
```

```
280             *pMutatingIndiv = *(MUTATE_INDIVIDUAL (Canvas, ReferenceIndiv,
281             /**/ RandMutationProportion));
282             // FREE_INDIVIDUAL (pMutatingIndiv);
283         }
284 
285         // cross between the best until
286         int parent_1 = 0;
287         int parent_2 = 1;
288         for (int index = NumKeptFromPreviousPop + NumMutatedFromPreviousPop;
289         /**/ index < NumKeptFromPreviousPop + NumMutatedFromPreviousPop + NumCrossedFromPreviousPop;
290         /**/ index++){
291 
292             individual ParentIndiv1 = Popl->individualsA[Popl->sorted_individual_indices_score
293             /**/ [parent_1]];
294             individual ParentIndiv2 = Popl->individualsA[Popl->sorted_individual_indices_score
295             /**/ [parent_2]];
296 
297             individual* pChildIndiv = &Popl->individualsA[Popl->sorted_individual_indices_score
298             /**/ [index]];
299 
300             double MaxProportionOfFirst = ParentIndiv1.total_score/(ParentIndiv1.total_score +
301             /**/ ParentIndiv2.total_score);
302             double StdDeviation = 0.5*MaxProportionOfFirst*(double)rand()/(double)RAND_MAX;
303 
304             *pChildIndiv = *(CROSS_INDIVIDUALS (Canvas, ParentIndiv1, ParentIndiv2,
305             /**/ MaxProportionOfFirst, StdDeviation));
306 
307             parent_1++;
308             parent_2++;
309         }
310 
311         // complete the rest with randomly generated individuals
312         for (int index = NumKeptFromPreviousPop + NumMutatedFromPreviousPop + NumCrossedFromPreviousPop;
313         /**/ index < Popl->num_individuals; index++){
314 
315             individual* pCurrentIndiv = &Popl->individualsA
316             /**/ [Popl->sorted_individual_indices_score[index]];
317 
318             // FREE_INDIVIDUAL (pCurrentIndiv);
319             INITIALIZE_INDIVIDUAL (pCurrentIndiv, Canvas, NumHomes);
320             // printf ("Individual %d/%d generated randomly.\n", index, Popl->num_individuals);
321             // printf ("Attractiveness: %f | Density: %f | Entropy: %f | Cost penalty: %f || Total: %f\n",
322             // /**/ index,
323             // /**/ pCurrentIndiv->attractiveness_score,
```

```
324             // /**/ pCurrentIndiv->density_score,
325             // /**/ pCurrentIndiv->entropy_score,
326             // /**/ pCurrentIndiv->cost_penalty,
327             // /**/ pCurrentIndiv->total_score);
328             // SDL_RenderClear (RendererGENALGO);
329             // SDL_RenderExistingServices (RendererGENALGO, Canvas);
330             // SDL_RenderExistingHomes (RendererGENALGO, Canvas, pCurrentIndiv->dispo);
331             // SDL_DisplayUnitEntropies (RendererGENALGO, Canvas, pCurrentIndiv->dispo);
332
333             double RandMutationProportion = 0.15*((double)rand()/(double)RAND_MAX);
334             *pCurrentIndiv = *(MUTATE_INDIVIDUAL (Canvas, *pCurrentIndiv, RandMutationProportion));
335             int parent_1 = rand () % (Popl->num_individuals);
336             int parent_2 = parent_1;
337             while (parent_2 == parent_1)
338             parent_2 = rand () % (Popl->num_individuals);
339
340             individual ParentIndiv1 = Popl->individualsA[Popl->sorted_individual_indices_score
341             /**/ [parent_1]];
342             individual ParentIndiv2 = Popl->individualsA[Popl->sorted_individual_indices_score
343             /**/ [parent_2]];
344
345             *pCurrentIndiv = *(CROSS_INDIVIDUALS (Canvas, ParentIndiv1, ParentIndiv2,
346             /**/ 0.5, 0.25));
347             // printf ("Individual %d/%d mutated.\n", index, Popl->num_individuals);
348             // printf ("Attractiveness: %f | Density: %f | Entropy: %f | Cost penalty: %f || Total: %f\n",
349             // /**/ index,
350             // /**/ pCurrentIndiv->attractiveness_score,
351             // /**/ pCurrentIndiv->density_score,
352             // /**/ pCurrentIndiv->entropy_score,
353             // /**/ pCurrentIndiv->cost_penalty,
354             // /**/ pCurrentIndiv->total_score);
355             // printf("\n");
356             // SDL_RenderClear (RendererGENALGO);
357             // SDL_RenderExistingServices (RendererGENALGO, Canvas);
358             // SDL_RenderExistingHomes (RendererGENALGO, Canvas, pCurrentIndiv->dispo);
359             // SDL_DisplayUnitEntropies (RendererGENALGO, Canvas, pCurrentIndiv->dispo);
360         }
361
362         // resort individuals
363         SORT_INDIVIDUAL_INDICES (Popl);
364     }
365
366
367     void FREE_INDIVIDUAL (individual* pIndiv){
```

```
368
369         printf ("freeing individual\n");
370         FREE_DISPOSITION (pIndiv->dispo);
371     }
```

## interactions.h

```
1   #ifndef INTERACTIONS_H
2   #define INTERACTIONS_H
3
4   #include "canvas.h"
5   #include "terrain.h"
6   #include "display.h"
7   #include "dispo.h"
8   #include "gen-algo.h"
9
10  void USER_CLEAR_SCREEN (SDL_Renderer* Renderer);
11
12  void USER_CYCLE_SERVICE_CLASS_PREV (SDL_Renderer* Renderer, int* pServiceClass,
13  /**/ int* pRed, int* pBlue, int* pGreen);
14  void USER_CYCLE_SERVICE_CLASS_NEXT (SDL_Renderer* Renderer, int* pServiceClass,
15  /**/ int* pRed, int* pBlue, int* pGreen);
16
17  void USER_DRAW_SERVICE (SDL_Renderer* Renderer, int* pRed, int* pGreen, int* pBlue,
18  /**/ int* pMouseX, int* pMouseY, canvas Canvas, int* pFill_i, int* pFill_j, int ServiceClass);
19  void USER_FILL_SERVICE (SDL_Renderer* Renderer, int* pMouseX, int* pMouseY,
20  /**/ canvas Canvas, int* pFill_i, int* pFill_j, int ServiceClass);
21  void USER_RERENDER_SERVICES (SDL_Renderer* Renderer, canvas Canvas);
22
23
24  void USER_CALCULATE_AND_DISPLAY_ATTRACTIVENESSES (SDL_Renderer* Renderer, canvas Canvas);
25  void USER_DISPLAY_ATTRACTIVENESSES (SDL_Renderer* Renderer, canvas Canvas);
26
27  void USER_INITIALIZE_DISPOSITION_AND_DISPLAY_HOMES (SDL_Renderer* Renderer, canvas Canvas,
28  /**/ disposition Dispo);
29  void USER_CALCULATE_AND_DISPLAY_DENSITIES (SDL_Renderer* Renderer, canvas Canvas, disposition Dispo);
30
31  void USER_INITIALIZE_AND_SORT_POPULATION (canvas Canvas, population* pPopl);
32  void USER_RENEW_POPULATION (canvas Canvas, population Popl);
33
34  void USER_DISPLAY_POPULATION_INDIVIDUAL_DESC_SCORE (SDL_Renderer* Renderer, canvas Canvas,
```

```
35  /**/ population Popl);
36
37  void USER_INITIALIZE_AND_DISPLAY_INIDIVIDUAL (canvas Canvas, disposition Dispo,
38  /**/ individual* pIndiv);
39  void USER_MUTATE_AND_DISPLAY_INDIVIDUAL (SDL_Renderer* Renderer, canvas Canvas,
40  /**/ individual* pIndiv);
41
42  // void USER_LAUNCH_GENETIC_ALGORITHM (SDL_Renderer* Renderer, canvas Canvas);
43  void USER_CALCULATE_AND_DISPLAY_ENTROPIES (SDL_Renderer* Renderer, canvas Canvas, disposition Dispo);
44
45
46
47
48  void USER_COMPUTE_AND_DISPLAY_HEIGHTS (SDL_Renderer* Renderer, canvas Canvas);
49  void USER_DISPLAY_TERRAIN (SDL_Renderer* Renderer, canvas Canvas);
50
51
52  void INITIALIZE_AND_EVOLVE_POPULATION (SDL_Renderer* Renderer, canvas Canvas);
53  void USER_GENETIC_ALGORITHM (SDL_Renderer* Renderer, canvas    Canvas);
54
55
56
57  void USER_INITIALIZE_POPULATION (canvas Canvas, population* pPopl, double* pKeepProportion,
58  /**/ double* pMutateProportion);
59  void USER_GENETIC_ALGORITHM_STEP (canvas Canvas, population Popl,
60  /**/ double KeepProportion, double MutateProportion);
61
62  void USER_CROSS_AND_DISPLAY_INDIVIDUALS (SDL_Renderer* Renderer, canvas Canvas,
63  /**/ individual* pIndiv1, individual* pIndiv2);
64  void USER_INITIALIZE_AND_DISPLAY_INDIVIDUAL_AND_DISPO (SDL_Renderer* Renderer,
65  canvas Canvas, individual* pIndiv);
66
67  #endif
```

## interactions.c

```
1   #include "interactions.h"
2
3   // renders an empty black screen
4   void USER_CLEAR_SCREEN (SDL_Renderer* Renderer){
5
```

```
6         SDL_SetRenderDrawColor (Renderer, 0, 0, 0, 0);
7         SDL_RenderClear (Renderer);
8         SDL_RenderPresent (Renderer);
9     }
10
11
12    // assigns to *pServiceClass the previous service class ID and readies *pRed, *pGreen and
13    // *pBlue for drawing in the new class's display color
14    void USER_CYCLE_SERVICE_CLASS_PREV (SDL_Renderer* Renderer, int* pServiceClass,
15    /**/ int* pRed, int* pBlue, int* pGreen){
16
17        *pServiceClass = (*pServiceClass - 1) % cNumServiceClasses;
18        if (*pServiceClass < 0)
19        *pServiceClass += cNumServiceClasses;
20        ASSIGN_COLOR_OF_SERVICE_CLASS (*pServiceClass, pRed, pGreen, pBlue);
21        SDL_SetRenderDrawColor (Renderer, *pRed, *pGreen, *pBlue, 255);
22    }
23
24
25    // assigns to *pServiceClass the next service class ID and readies *pRed, *pGreen and
26    // *pBlue for drawing in the new class's display color
27    void USER_CYCLE_SERVICE_CLASS_NEXT (SDL_Renderer* Renderer, int* pServiceClass,
28    /**/ int* pRed, int* pBlue, int* pGreen){
29
30        *pServiceClass = (*pServiceClass + 1) % cNumServiceClasses;
31        ASSIGN_COLOR_OF_SERVICE_CLASS (*pServiceClass, pRed, pGreen, pBlue);
32        SDL_SetRenderDrawColor (Renderer, *pRed, *pGreen, *pBlue, 255);
33    }
34
35
36    // fills each blank cell of Canvas hovered over by the user's cursor in ServiceClass's
37    // dispay color, also updating Canvas->service_border_pixels by adding midpoints of all
38    // newly filled cells
39    void USER_DRAW_SERVICE (SDL_Renderer* Renderer, int* pRed, int* pGreen, int* pBlue,
40    /**/ int* pMouseX, int* pMouseY, canvas Canvas, int* pFill_i, int* pFill_j, int ServiceClass){
41
42        SDL_GetMouseState (pMouseX, pMouseY);
43        ASSIGN_CELL_OF_PIXEL (pFill_i, pFill_j, Canvas, *pMouseX, *pMouseY);
44
45        ASSIGN_COLOR_OF_SERVICE_CLASS (ServiceClass, pRed, pGreen, pBlue);
46        SDL_SetRenderDrawColor (Renderer, *pRed, *pGreen, *pBlue, 255);
47
48        if (Canvas->cellsA[*pFill_i][*pFill_j].service_presence_class == -1){
49
```

```
50              Canvas->cellsA[*pFill_i][*pFill_j].service_presence_class = ServiceClass;
51              SDL_RenderFillRect (Renderer, &Canvas->cellsA[*pFill_i][*pFill_j].fill_square);
52              SDL_RenderPresent (Renderer);
53
54              int X, Y;
55              ASSIGN_MIDPOINT_OF_CELL (&X, &Y, Canvas, *pFill_i, *pFill_j);
56              l_insert (&Canvas->service_border_pixelsL, X, Y, ServiceClass);
57          }
58      }
59
60
61      // hyp: Canvas' service whose borders surround the user's current cursor position has
62      // class ServiceClass
63      // fills the cells surrounded by the aforementioned service
64      void USER_FILL_SERVICE (SDL_Renderer* Renderer, int* pMouseX, int* pMouseY,
65      /**/ canvas Canvas, int* pFill_i, int* pFill_j, int ServiceClass){
66
67          SDL_GetMouseState (pMouseX, pMouseY);
68          ASSIGN_CELL_OF_PIXEL (pFill_i, pFill_j, Canvas, *pMouseX, *pMouseY);
69          SDL_FloodFillService (Renderer, Canvas, ServiceClass, *pFill_i, *pFill_j);
70          SDL_RenderPresent (Renderer);
71      }
72
73
74      // rerenders services which have been drawn on Canvas
75      void USER_RERENDER_SERVICES (SDL_Renderer* Renderer, canvas Canvas){
76
77          SDL_RenderExistingServices (Renderer, Canvas);
78      }
79
80      // calculates then displays the attractivenesses of Canvas' cells
81      void USER_CALCULATE_AND_DISPLAY_ATTRACTIVENESSES (SDL_Renderer* Renderer, canvas Canvas){
82
83          CALCULATE_ATTRACTIVENESSES (Canvas);
84          SDL_DisplayCellAttractivenesses (Renderer, Canvas);
85      }
86
87
88      // hyp: Canvas' cell attractivenesses have been previously calculated
89      // displays the attractivenesses of Canvas' cells
90      void USER_DISPLAY_ATTRACTIVENESSES (SDL_Renderer* Renderer, canvas Canvas){
91
92          SDL_DisplayCellAttractivenesses (Renderer, Canvas);
93      }
```

```
94
95
96    // creates and initializes a disposition on Canvas set to contain NumHomes homes using
97    // INITIALIZE_DISPOSITION_FROM_CELLS, then displays it
98    void USER_INITIALIZE_DISPOSITION_AND_DISPLAY_HOMES (SDL_Renderer* Renderer, canvas Canvas,
99    /**/ disposition Dispo){
100
101       int NumHomes;
102       printf ("Enter number of homes to place for test disposition :\n");
103       scanf ("%d", &NumHomes);
104
105       double HomeAttributionInflation;
106       printf ("Enter inflation parameter for home attribution :\n");
107       scanf ("%lf", &HomeAttributionInflation);
108
109       INITIALIZE_DISPOSITION_FROM_CELLS_2 (Canvas, Dispo, NumHomes, HomeAttributionInflation);
110       SDL_RenderExistingHomes (Renderer, Canvas, Dispo);
111    }
112
113
114    // hyp: TestDispo has been initialized
115    //
116    void USER_CALCULATE_AND_DISPLAY_DENSITIES (SDL_Renderer* Renderer, canvas Canvas, disposition Dispo){
117
118       CALCULATE_LOCAL_DENSITIES (Canvas, Dispo);
119       SDL_DisplayUnitDensities (Renderer, Canvas, Dispo);
120       SORT_UNIT_INDICES (Canvas, Dispo);
121    }
122
123
124    void USER_INITIALIZE_AND_SORT_POPULATION (canvas Canvas, population* pPopl){
125
126       int NumHomes;
127       printf ("Enter number of homes to place for population's individuals :\n");
128       scanf ("%d", &NumHomes);
129
130       int PopSize;
131       printf ("Enter number of individuals to generate :\n");
132       scanf ("%d", &PopSize);
133
134       *pPopl = CREATE_POPULATION (Canvas, PopSize);
135       for (int index = 0; index < PopSize; index++){
136          INITIALIZE_INDIVIDUAL (((*pPopl)->individualsA)+index, Canvas, NumHomes);
137          printf ("%f\n", (*pPopl)->individualsA[index].total_score);
```

```
138         }
139         SORT_INDIVIDUAL_INDICES (*pPopl);
140     }
141
142
143     // void USER_RENEW_POPULATION (canvas Canvas, population Popl){
144         //     double ConservationProportion;
145         //     printf ("Enter proportion of fittest individuals to keep and mutate :\n");
146         //     scanf ("%lf", &ConservationProportion);
147
148         //     ITERATE_POPULATION_RENEWAL (Canvas, Popl, ConservationProportion);
149         // }
150
151
152     void USER_DISPLAY_POPULATION_INDIVIDUAL_DESC_SCORE (SDL_Renderer* Renderer, canvas Canvas,
153     /**/ population Popl){
154
155         int IndivIndex;
156         printf ("Enter index of individual to display (between 0 and %d) :\n", Popl->num_individuals-1);
157         scanf ("%d", &IndivIndex);
158
159         USER_CLEAR_SCREEN (Renderer);
160         USER_RERENDER_SERVICES (Renderer, Canvas);
161         SDL_RenderExistingHomes (Renderer, Canvas,
162         /**/ Popl->individualsA[Popl->sorted_individual_indices_score[IndivIndex]].dispo);
163
164         printf ("\n---------\nAttractiveness : %f\nDensity : %f\nEntropy : %f (entropy is %f)\nCost : %f\nTota
165         /**/ Popl->individualsA[Popl->sorted_individual_indices_score[IndivIndex]].attractiveness_score,
166         /**/ Popl->individualsA[Popl->sorted_individual_indices_score[IndivIndex]].density_score,
167         Popl->individualsA[Popl->sorted_individual_indices_score[IndivIndex]].entropy_score,
168         Popl->individualsA[Popl->sorted_individual_indices_score[IndivIndex]].dispo->entropy,
169         Popl->individualsA[Popl->sorted_individual_indices_score[IndivIndex]].cost_penalty,
170         /**/ Popl->individualsA[Popl->sorted_individual_indices_score[IndivIndex]].total_score);
171     }
172
173
174     void USER_INITIALIZE_AND_DISPLAY_INIDIVIDUAL (canvas Canvas, disposition Dispo, individual* pIndiv){
175
176         pIndiv->dispo = Dispo;
177         pIndiv->total_score = -DBL_MAX;
178         CALCULATE_SCORES_OF_INDIVIDUAL (pIndiv, Canvas);
179         printf ("Attractiveness : %f || Density : %f || Total : %f \n",
180         /**/ pIndiv->attractiveness_score, pIndiv->density_score, pIndiv->total_score);
181     }
```

```
182
183 void USER_INITIALIZE_AND_DISPLAY_INDIVIDUAL_AND_DISPO (SDL_Renderer* Renderer,
184 canvas Canvas, individual* pIndiv){
185
186     pIndiv->dispo = CREATE_CANVAS_DISPOSITION (Canvas);
187
188     int NumHomes;
189     printf ("Enter number of homes to place for test disposition :\n");
190     scanf ("%d", &NumHomes);
191
192     double HomeAttributionInflation;
193     printf ("Enter inflation parameter for home attribution :\n");
194     scanf ("%lf", &HomeAttributionInflation);
195
196     INITIALIZE_DISPOSITION_FROM_CELLS_2 (Canvas, pIndiv->dispo, NumHomes, HomeAttributionInflation);
197     SDL_RenderExistingHomes (Renderer, Canvas, pIndiv->dispo);
198
199     pIndiv->total_score = -DBL_MAX;
200     CALCULATE_SCORES_OF_INDIVIDUAL (pIndiv, Canvas);
201 }
202
203
204 void USER_MUTATE_AND_DISPLAY_INDIVIDUAL (SDL_Renderer* Renderer, canvas Canvas,
205 /**/ individual* pIndiv){
206
207     double MutationProportion;
208     printf ("Enter proportion of units to be mutated for individual :\n");
209     scanf ("%lf", &MutationProportion);
210
211     individual* pNewIndiv = MUTATE_INDIVIDUAL (Canvas, *pIndiv, MutationProportion);
212     *pIndiv = *pNewIndiv;
213     USER_CLEAR_SCREEN (Renderer);
214     USER_RERENDER_SERVICES (Renderer, Canvas);
215     SDL_RenderExistingHomes (Renderer, Canvas, pIndiv->dispo);
216     CALCULATE_LOCAL_DENSITIES (Canvas, pIndiv->dispo);
217     SORT_UNIT_INDICES (Canvas, pIndiv->dispo);
218 }
219
220
221 void USER_CROSS_AND_DISPLAY_INDIVIDUALS (SDL_Renderer* Renderer, canvas Canvas,
222 /**/ individual* pIndiv1, individual* pIndiv2){
223
224     double MaxProportionOfFirst;
225     double StdDeviation;
```

```
226         printf ("Enter maximum proportion of first :\n");
227         scanf ("%lf", &MaxProportionOfFirst);
228         printf ("Enter standard deviation (ideally > (1/2)*MaxProportionOfFirst) :\n");
229         scanf ("%lf", &StdDeviation);
230         printf ("ok, %lf and %lf\n", MaxProportionOfFirst, StdDeviation);
231 
232         individual* pNewIndiv = CROSS_INDIVIDUALS (Canvas, *pIndiv1, *pIndiv2,
233         /**/ MaxProportionOfFirst, StdDeviation);
234 
235         *pIndiv1 = *pNewIndiv;
236         USER_CLEAR_SCREEN (Renderer);
237         USER_RERENDER_SERVICES (Renderer, Canvas);
238         SDL_RenderExistingHomes (Renderer, Canvas, pIndiv1->dispo);
239         CALCULATE_LOCAL_DENSITIES (Canvas, pIndiv1->dispo);
240         SORT_UNIT_INDICES (Canvas, pIndiv1->dispo);
241     }
242 
243     // void USER_LAUNCH_GENETIC_ALGORITHM (SDL_Renderer* Renderer, canvas Canvas){
244 
245         //     int NumHomes, GenerationSize, NumRenewals;
246         //     printf ("Enter number of homes to place on canvas :\n");
247         //     scanf ("%d", &NumHomes);
248         //     printf ("Enter number of individuals in population :\n");
249         //     scanf ("%d", &GenerationSize);
250         //     printf ("Enter the number of generations :\n");
251         //     scanf ("%d", &NumRenewals);
252 
253         //     population Popl = GENERATE_RANDOM_POPULATION (Canvas, GenerationSize, NumHomes);
254 
255         //     printf("a\n");
256         //     USER_CLEAR_SCREEN (Renderer);
257         //     USER_RERENDER_SERVICES (Renderer, Canvas);
258         //     SDL_RenderExistingHomes (Renderer, Canvas,
259         //     /**/ Popl->individualsA[Popl->sorted_individual_indices_score[0]].dispo);
260 
261         //     printf ("Best individual stats\n--------------------\nAttractiveness : %f || Density : %f || T
262         //     /**/ Popl->individualsA[Popl->sorted_individual_indices_score[0]].attractiveness_score,
263         //     /**/ Popl->individualsA[Popl->sorted_individual_indices_score[0]].density_score,
264         //     /**/ Popl->individualsA[Popl->sorted_individual_indices_score[0]].total_score);
265 
266         //     printf ("b\n");
267         //     for (int generation = 0; generation < NumRenewals; generation++){
268 
269             //         for (int individual = 0; individual < GenerationSize; individual++){
```

```
270             //              assert (IS_DISPOSITION_HOME_ASSIGNMENT_COHERENT
271             //              /**/ (Popl->individualsA[individual].dispo));
272             //          }
273         //          ITERATE_POPULATION_RENEWAL (Canvas, Popl, 0.6);
274 
275         //          USER_CLEAR_SCREEN (Renderer);
276         //          USER_RERENDER_SERVICES (Renderer, Canvas);
277         //          SDL_RenderExistingHomes (Renderer, Canvas,
278         //          /**/ Popl->individualsA[Popl->sorted_individual_indices_score[0]].dispo);
279 
280         //          printf ("Best individual stats\n---------------------\nAttractiveness : %f || Density
281         //          /**/ Popl->individualsA[Popl->sorted_individual_indices_score[0]].attractiveness_score
282         //          /**/ Popl->individualsA[Popl->sorted_individual_indices_score[0]].density_score,
283         //          /**/ Popl->individualsA[Popl->sorted_individual_indices_score[0]].total_score);
284 
285         //      }
286     //      printf ("\n\nDone.\n");
287     // }
288 
289 
290 void USER_CALCULATE_AND_DISPLAY_ENTROPIES (SDL_Renderer* Renderer, canvas Canvas, disposition Dispo){
291 
292     CALCULATE_LOCAL_ENTROPIES (Canvas, Dispo);
293     SDL_DisplayUnitEntropies (Renderer, Canvas, Dispo);
294 }
295 
296 void USER_COMPUTE_AND_DISPLAY_HEIGHTS (SDL_Renderer* Renderer, canvas Canvas){
297 
298     double Amplitude;
299     int Frequency;
300     int NumOctaves;
301     double Lacunarity;
302     double Persistence;
303     double Exponentiation;
304     printf ("Enter following parameters in order, seperated by semicolons : amplitude, frequency, number 
305     scanf ("%lf ; %d ; %d ; %lf ; %lf ; %lf", &Amplitude, &Frequency, &NumOctaves, &Lacunarity,
306     /**/ &Persistence, &Exponentiation);
307     printf ("%f ; %d ; %d ; %f ; %f ; %f\n", Amplitude, Frequency, NumOctaves, Lacunarity,
308     /**/ Persistence, Exponentiation);
309 
310     GENERATE_TERRAIN_HEIGHTMAP (Canvas, Amplitude, Frequency, NumOctaves, Lacunarity,
311     /**/ Persistence, Exponentiation);
312 
313     SDL_DisplayTerrainHeights (Renderer, Canvas);
```

```
314    }
315
316    void USER_DISPLAY_TERRAIN (SDL_Renderer* Renderer, canvas Canvas){
317
318        SDL_RenderTerrain (Renderer, Canvas);
319    }
320
321
322
323
324
325    void INITIALIZE_AND_EVOLVE_POPULATION (SDL_Renderer* Renderer, canvas Canvas){
326
327        int NumHomes;
328        int NumIndividuals;
329        int NumGenerations;
330
331        double KeepProportion;
332        double MutateProportion;
333        double CrossProportion;
334
335        printf ("Enter : NumHomes, NumIndividuals, NumGenerations, KeepProportion, MutateProportion, CrossProp
336        scanf ("%d ; %d ; %d ; %lf ; %lf ; %lf", &NumHomes, &NumIndividuals, &NumGenerations,
337        /**/ &KeepProportion, &MutateProportion, &CrossProportion);
338
339        // printf ("a\n");
340
341        // initialize first generation
342        population Popl = CREATE_POPULATION (Canvas, NumIndividuals);
343        for (int index = 0; index < Popl->num_individuals; index++){
344
345            INITIALIZE_INDIVIDUAL (&Popl->individualsA[index], Canvas, NumHomes);
346        }
347        SORT_INDIVIDUAL_INDICES (Popl);
348
349        // printf ("b\n");
350
351        USER_CLEAR_SCREEN (Renderer);
352        USER_RERENDER_SERVICES (Renderer, Canvas);
353        // USER_DISPLAY_TERRAIN (Renderer, Canvas);
354        SDL_RenderExistingHomes (Renderer, Canvas,
355        /**/ Popl->individualsA[Popl->sorted_individual_indices_score[0]].dispo);
356
357        // printf ("c\n");
```

```
358
359         printf ("INIT_POP:\n----------\n");
360         for (int index = 0; index < NumIndividuals; index++){
361             printf ("IND#%d : Attractiveness: %f | Density: %f | Entropy: %f | Cost penalty: %f || Total: %f\n
362             /**/ index,
363             /**/ Popl->individualsA[Popl->sorted_individual_indices_score[index]].attractiveness_score,
364             /**/ Popl->individualsA[Popl->sorted_individual_indices_score[index]].density_score,
365             /**/ Popl->individualsA[Popl->sorted_individual_indices_score[index]].entropy_score,
366             /**/ Popl->individualsA[Popl->sorted_individual_indices_score[index]].cost_penalty,
367             /**/ Popl->individualsA[Popl->sorted_individual_indices_score[index]].total_score);
368         }
369         printf ("\n");
370
371         // iterate renewal for NumGenerations generations
372         for (int generation = 0; generation < NumGenerations; generation++){
373
374             RENEW_POPULATION (Canvas, Popl, KeepProportion, MutateProportion, CrossProportion);
375
376             USER_CLEAR_SCREEN (Renderer);
377             USER_RERENDER_SERVICES (Renderer, Canvas);
378             // USER_DISPLAY_TERRAIN (Renderer, Canvas);
379             SDL_RenderExistingHomes (Renderer, Canvas,
380             /**/ Popl->individualsA[Popl->sorted_individual_indices_score[0]].dispo);
381
382             printf ("POP %d:\n----------\n", generation+1);
383             for (int index = 0; index < NumIndividuals; index++){
384                 printf ("IND#%d : Attractiveness: %f | Density: %f | Entropy: %f | Cost penalty: %f || Total:
385                 /**/ index,
386                 /**/ Popl->individualsA[Popl->sorted_individual_indices_score[index]].attractiveness_score,
387                 /**/ Popl->individualsA[Popl->sorted_individual_indices_score[index]].density_score,
388                 /**/ Popl->individualsA[Popl->sorted_individual_indices_score[index]].entropy_score,
389                 /**/ Popl->individualsA[Popl->sorted_individual_indices_score[index]].cost_penalty,
390                 /**/ Popl->individualsA[Popl->sorted_individual_indices_score[index]].total_score);
391             }
392             printf ("\n");
393         }
394     }
395
396
397
398     void USER_INITIALIZE_POPULATION (canvas Canvas, population* pPopl,
399     /**/ double* pKeepProportion,
400     /**/ double* pMutateProportion){
401
```

```
402         int NumHomes;
403         int NumIndividuals;
404 
405         printf ("Enter : NumHomes, NumIndividuals, KeepProportion, MutateProportion.\n");
406         scanf ("%d ; %d ; %lf ; %lf", &NumHomes, &NumIndividuals,
407         /**/ pKeepProportion, pMutateProportion);
408 
409         *pPopl = CREATE_POPULATION (Canvas, NumIndividuals);
410         for (int index = 0; index < (*pPopl)->num_individuals; index++){
411 
412             INITIALIZE_INDIVIDUAL (&(*pPopl)->individualsA[index], Canvas, NumHomes);
413         }
414         SORT_INDIVIDUAL_INDICES (*pPopl);
415 
416         printf ("\n----------\n");
417         for (int index = 0; index < NumIndividuals; index++){
418             printf ("IND#%d : Attractiveness: %f | Density: %f | Entropy: %f | Cost penalty: %f || Total: %f\n
419             /**/ index,
420             /**/ (*pPopl)->individualsA[(*pPopl)->sorted_individual_indices_score[index]].attractiveness_score
421             /**/ (*pPopl)->individualsA[(*pPopl)->sorted_individual_indices_score[index]].density_score,
422             /**/ (*pPopl)->individualsA[(*pPopl)->sorted_individual_indices_score[index]].entropy_score,
423             /**/ (*pPopl)->individualsA[(*pPopl)->sorted_individual_indices_score[index]].cost_penalty,
424             /**/ (*pPopl)->individualsA[(*pPopl)->sorted_individual_indices_score[index]].total_score);
425         }
426         printf ("\n");
427     }
428 
429     void USER_GENETIC_ALGORITHM_STEP (canvas Canvas, population Popl,
430     /**/ double KeepProportion, double MutateProportion){
431 
432         RENEW_POPULATION (Canvas, Popl, KeepProportion, MutateProportion, 0);
433         int NumIndividuals = Popl->num_individuals;
434 
435         printf ("\n----------\n");
436         for (int index = 0; index < NumIndividuals; index++){
437             printf ("IND#%d : Attractiveness: %f | Density: %f | Entropy: %f | Cost penalty: %f || Total: %f\n
438             /**/ index,
439             /**/ Popl->individualsA[Popl->sorted_individual_indices_score[index]].attractiveness_score,
440             /**/ Popl->individualsA[Popl->sorted_individual_indices_score[index]].density_score,
441             /**/ (Popl)->individualsA[Popl->sorted_individual_indices_score[index]].entropy_score,
442             /**/ (Popl)->individualsA[Popl->sorted_individual_indices_score[index]].cost_penalty,
443             /**/ (Popl)->individualsA[Popl->sorted_individual_indices_score[index]].total_score);
444         }
445         printf ("\n");
```

```
446 }
447
448
449 void USER_GENETIC_ALGORITHM (SDL_Renderer* Renderer, canvas Canvas){
450
451     INITIALIZE_AND_EVOLVE_POPULATION (Renderer, Canvas);
452 }
```

## main.c

```
1  #include "canvas.h"
2  #include "dispo.h"
3  #include "gen-algo.h"
4
5  #include "display.h"
6  #include "interactions.h"
7
8  int main (int argc, char argv[]){
9
10     // initialize srand
11     srand(time(NULL));
12
13     // initialize and configure SDL
14     SDL_Window* Window = NULL;
15     SDL_Renderer* Renderer = NULL;
16
17     if (SDL_Init (SDL_INIT_VIDEO) != 0){
18         SDL_Log ("ERROR : SDL initialization failed > %s\n", SDL_GetError ());
19         exit (EXIT_FAILURE);
20     }
21
22     Window = SDL_CreateWindow ("", SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED, 1920, 1080,
23     /**/ SDL_WINDOW_MAXIMIZED);
24     if (Window == NULL){
25         SDL_Log ("ERROR : SDL window creation failed > %s\n", SDL_GetError ());
26         exit (EXIT_FAILURE);
27     }
28
29     Renderer = SDL_CreateRenderer(Window,-1,SDL_RENDERER_SOFTWARE);
30     if (Renderer == NULL) {
31         SDL_Log ("ERROR : SDL renderer creation failed > %s\n", SDL_GetError ());
```

```
32          exit (EXIT_FAILURE);
33      }
34      SDL_bool PRG_RUN = SDL_TRUE;
35
36      SDL_SetRenderDrawBlendMode(Renderer,SDL_BLENDMODE_ADD);
37      SDL_RenderPresent (Renderer);
38
39
40
41      RendererGENALGO = Renderer;
42      double KeepProportion;
43      double MutateProportion;
44
45
46      // initialize canvas and attributes
47      canvas MainCanvas = CREATE_CANVAS (cCellSizePixels, cWindowDims);
48      int CurrentServiceClass = 0;
49      int currentEntertainType = 0;
50
51      // declare/initialize test disposition and population
52      disposition TestDispo = CREATE_CANVAS_DISPOSITION (MainCanvas);
53      individual TestIndiv;
54      individual TestIndiv_bis;
55      population TestPopl;
56
57      // declare mouse interaction variables
58      bool IsMouseButtonClicked = false;
59      int MouseX, MouseY;
60
61      // declare drawing and filling varibales
62      int DrawRed;
63      int DrawGreen;
64      int DrawBlue;
65
66      int Fill_row;
67      int Fill_column;
68
69
70
71      SDL_Event Event;
72
73      // program execution
74      while(PRG_RUN){
75
```

```
76          while (SDL_PollEvent(&Event)){
77
78              switch (Event.type){
79
80                  case (SDL_KEYDOWN) :
81
82                  switch (Event.key.keysym.sym){
83
84                      case SDLK_q :
85                      USER_CLEAR_SCREEN (Renderer);
86                      continue;
87                      case SDLK_w :
88                      USER_CYCLE_SERVICE_CLASS_PREV (Renderer, &CurrentServiceClass,
89                      &DrawRed, &DrawBlue, &DrawGreen);
90                      continue;
91                      case SDLK_e :
92                      USER_CYCLE_SERVICE_CLASS_NEXT (Renderer, &CurrentServiceClass,
93                      &DrawRed, &DrawBlue, &DrawGreen);
94                      continue;
95                      case SDLK_r :
96                      USER_RERENDER_SERVICES (Renderer, MainCanvas);
97                      continue;
98                      case SDLK_t :
99                      USER_CALCULATE_AND_DISPLAY_ATTRACTIVENESSES (Renderer,
100                     MainCanvas);
101                     continue;
102                     case SDLK_y :
103                     USER_DISPLAY_ATTRACTIVENESSES (Renderer, MainCanvas);
104                     continue;
105                     case SDLK_f :
106                     USER_FILL_SERVICE (Renderer, &MouseX, &MouseY, MainCanvas,
107                     &Fill_row, &Fill_column, CurrentServiceClass);
108                     continue;
109                     case SDLK_u :
110                     USER_INITIALIZE_DISPOSITION_AND_DISPLAY_HOMES (Renderer,
111                     MainCanvas, TestDispo);
112                     continue;
113                     case SDLK_i :
114                     USER_CALCULATE_AND_DISPLAY_DENSITIES (Renderer,
115                     MainCanvas, TestDispo);
116                     continue;
117                     case SDLK_o :
118                     USER_INITIALIZE_AND_SORT_POPULATION (MainCanvas, &TestPopl);
119                     continue;
```

```
120                     case SDLK_p :
121                     USER_DISPLAY_POPULATION_INDIVIDUAL_DESC_SCORE (Renderer, MainCanvas,
122                     TestPopl);
123                     continue;
124                     case SDLK_a :
125                     USER_INITIALIZE_AND_DISPLAY_INIDIVIDUAL (MainCanvas,
126                     TestDispo, &TestIndiv);
127                     continue;
128                     case SDLK_s :
129                     USER_MUTATE_AND_DISPLAY_INDIVIDUAL (Renderer, MainCanvas, &TestIndiv);
130                     continue;
131                     /* case SDLK_d :
132                     USER_RENEW_POPULATION (MainCanvas, TestPopl);
133                     continue; */
134                     /* case SDLK_g :
135                     USER_LAUNCH_GENETIC_ALGORITHM (Renderer, MainCanvas);
136                     continue; */
137                     case SDLK_h :
138                     USER_CALCULATE_AND_DISPLAY_ENTROPIES (Renderer, MainCanvas,
139                     TestDispo);
140                     continue;
141                     case SDLK_j :
142                     USER_COMPUTE_AND_DISPLAY_HEIGHTS (Renderer, MainCanvas);
143                     continue;
144                     case SDLK_k :
145                     USER_DISPLAY_TERRAIN (Renderer, MainCanvas);
146                     continue;
147                     case SDLK_l :
148                     USER_GENETIC_ALGORITHM (Renderer, MainCanvas);
149                     continue;
150                     case SDLK_z :
151                     USER_INITIALIZE_POPULATION (MainCanvas, &TestPopl, &KeepProportion,
152                     &MutateProportion);
153                     continue;
154                     case SDLK_x :
155                     USER_GENETIC_ALGORITHM_STEP (MainCanvas, TestPopl, KeepProportion,
156                     MutateProportion);
157                     continue;
158                     case SDLK_c :
159                     USER_INITIALIZE_AND_DISPLAY_INDIVIDUAL_AND_DISPO (Renderer, MainCanvas,
160                     &TestIndiv);
161                     continue;
162                     case SDLK_v :
163                     USER_INITIALIZE_AND_DISPLAY_INDIVIDUAL_AND_DISPO (Renderer, MainCanvas,
```

```
164                     &TestIndiv_bis);
165                     continue;
166                     case SDLK_b :
167                     USER_CROSS_AND_DISPLAY_INDIVIDUALS (Renderer, MainCanvas,
168                     &TestIndiv, &TestIndiv_bis);
169                     continue;
170 
171                     default:
172                     continue;
173                 }
174 
175                 case (SDL_MOUSEBUTTONDOWN) :
176                 IsMouseButtonClicked = true;
177                 continue;
178 
179                 case (SDL_MOUSEBUTTONUP) :
180                 IsMouseButtonClicked = false;
181                 continue;
182 
183                 case (SDL_MOUSEMOTION) :
184                 if (IsMouseButtonClicked) USER_DRAW_SERVICE (Renderer, &DrawRed,
185                 &DrawBlue, &DrawGreen, &MouseX, &MouseY, MainCanvas, &Fill_row,
186                 &Fill_column, CurrentServiceClass);
187                 continue;
188 
189                 case SDL_QUIT:
190                 PRG_RUN = SDL_FALSE;
191                 SDL_DestroyRenderer (Renderer);
192                 SDL_DestroyWindow (Window);
193                 break;
194                 default:
195                 continue;
196             }
197         }
198     }
199 
200     return EXIT_SUCCESS;
201 }
```

# linked_list.h

```c
#ifndef LINKED_LIST_H
#define LINKED_LIST_H

#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>

struct l_node_s {
    int x;
    int y;
    int spec;
    struct l_node_s* next;
};
typedef struct l_node_s* l_list;

l_list create_l_node (int x, int y, int spec);
bool is_empty_l_list (l_list l);
void free_l_list (l_list l);
void l_insert (l_list* pl, int x, int y, int spec);

#endif
```

## linked_list.c

```c
#include "linked_list.h"

l_list create_l_node (int x, int y, int spec){
    // retourne une liste chaînée réduite à une cellule contenant les champs x, y et spec
    l_list l = (l_list) malloc (sizeof(struct l_node_s));
    l->x = x;
    l->y = y;
    l->spec = spec;
    return l;
}

bool is_empty_l_list (l_list l){
    return l == NULL;
}

void free_l_list (l_list l){
    // libère l'espace mémoire alloué à la liste l
```

```
18      while (l != NULL){
19          l_list temp = l;
20          l = l->next;
21          free (temp);
22      }
23  }
24
25  void l_insert (l_list* pl, int x, int y, int spec){
26      l_list new_nd = create_l_node (x, y, spec);
27      new_nd->next = *pl;
28      *pl = new_nd;
29  }
```

## tools.h

```
1   #ifndef TOOLS_H
2   #define TOOLS_H
3
4   #include <stdio.h>
5   #include <stdbool.h>
6   #include <stdlib.h>
7   #include <math.h>
8
9   extern const double pi;
10
11  int min (int a, int b);
12  int max (int a, int b);
13  double min_double (double a, double b);
14  double max_double (double a, double b);
15  double RAND_VAR_HALF_NORMAL_DISTRIBUTION (double Peak, double StdDeviation);
16
17  #endif
```

## tools.c

```
1   #include "tools.h"
2
3
4   const double pi = 3.141592654;
```

```
5
6
7   int min (int a, int b){
8       if (a < b) return a;
9       else return b;
10  }
11
12  int max (int a, int b){
13      if (a > b) return a;
14      else return b;
15  }
16
17  double min_double (double a, double b){
18      if (a < b) return a;
19      else return b;
20  }
21
22  double max_double (double a, double b){
23      if (a > b) return a;
24      else return b;
25  }
26
27  double abs_double (double x){
28      if (x >= 0) return x;
29      else return -x;
30  }
31
32  // generates a random float in ]-inf,Peak] according to a left half-normal distribution
33  // with mean Peak and standard deviation StdDeviation
34  double RAND_VAR_HALF_NORMAL_DISTRIBUTION (double Peak, double StdDeviation){
35      double Unif1, Unif2;
36      double SumSq = 1;
37      // generate Unif1 and Unif2 values following uniform distribution on [-1,1[
38      while (SumSq >= 1){
39          Unif1 = 2*((double)rand() / (double)RAND_MAX)-1;
40          Unif2 = 2*((double)rand() / (double)RAND_MAX)-1;
41          SumSq = Unif1*Unif1 + Unif2*Unif2;
42      }
43      // NormalStandardized follows a normal distribution of mean 0 and variance 1
44      double NormalStandardized = Unif1*sqrt((-2*log(SumSq))/SumSq);
45      return -StdDeviation*(abs_double(NormalStandardized)) + Peak;
46  }
```

# Makefile

```
tools.o : tools.c tools.h
gcc -c -g tools.c -o tools.o $$(sdl2-config --cflags --libs) -lm

canvas.o : canvas.c canvas.h
gcc -c -g canvas.c -o canvas.o $$(sdl2-config --cflags --libs) -lm

terrain.o : terrain.c terrain.h
gcc -c -g terrain.c -o terrain.o $$(sdl2-config --cflags --libs) -lm

display.o : display.c display.h
gcc -c -g display.c -o display.o $$(sdl2-config --cflags --libs) -lm

dispo.o : dispo.c dispo.h
gcc -c -g dispo.c -o dispo.o $$(sdl2-config --cflags --libs) -lm

gen-algo.o : gen-algo.c gen-algo.h
gcc -c -g gen-algo.c -o gen-algo.o $$(sdl2-config --cflags --libs) -lm

interactions.o : interactions.c interactions.h
gcc -c -g interactions.c -o interactions.o $$(sdl2-config --cflags --libs) -lm

linked_list.o : linked_list.c linked_list.h
gcc -c -g linked_list.c -o linked_list.o




clean :
rm -f *.o

clean_all :
rm -f *.o
rm main

main : main.c tools.c canvas.c terrain.c dispo.c display.c gen-algo.c interactions.c linked_list.c
    tools.o canvas.o terrain.o dispo.o display.o gen-algo.o interactions.o linked_list.o
gcc -g tools.o canvas.o terrain.o dispo.o display.o gen-algo.o interactions.o linked_list.o main.c
```

```
    -o main $$(sdl2-config --cflags --libs) -lm
rm -f *.o
```