AutoCell

Generated by Doxygen 1.8.13

Contents

Chapter 1

Main Page

To generate the Documentation, go in Documentation directory and run `make`.

It will generate html doc (in `output/html/index.html`) and latex doc (pdf output directely in Documentation directory (`docPdf.pdf`).

Chapter 2

Presentation

What is AutoCell

The purpose of this project is to create a Cellular Automate Simulator.

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

Chapter 5

Class Documentation

5.1 Cell Class Reference

Contains the state, the next state and the neighbours.

`#include <cell.h>`

Public Member Functions

- Cell (unsigned int state=0)
  Constructs a cell with the state given. State 0 is dead state.
- void setState (unsigned int state)
  Set temporary state.
- void validState ()
  Validate temporary state.
- unsigned int getState () const
  Access current cell state.
- bool addNeighbour (const Cell *neighbour)
  Add a new neighbour to the Cell.
- QVector< const Cell * > getNeighbours () const
  Access neighbours list.

Private Attributes

- unsigned int m_state
  Current state.
- unsigned int m_nextState
  Temporary state, before validation.
- QVector< const Cell * > m_neighbours
  Cell's neighbours.

5.1.1 Detailed Description

Contains the state, the next state and the neighbours.

Definition at line 10 of file cell.h.

5.1.2 Constructor & Destructor Documentation

5.1.2.1 Cell()

```
Cell::Cell (
            unsigned int state = 0 )
```

Constructs a cell with the state given. State 0 is dead state.

Parameters

state	Cell state, dead state by default

Definition at line 8 of file cell.cpp.

5.1.3 Member Function Documentation

5.1.3.1 addNeighbour()

```
bool Cell::addNeighbour (
            const Cell * neighbour )
```

Add a new neighbour to the Cell.

Parameters

neighbour	New neighbour

Returns

False if the neighbour already exists

Definition at line 52 of file cell.cpp.

References m_neighbours.

5.1.3.2 getNeighbours()

```
QVector< const Cell * > Cell::getNeighbours ( ) const
```

Access neighbours list.

Definition at line 63 of file cell.cpp.

References m_neighbours.

5.1.3.3 getState()

```
unsigned int Cell::getState ( ) const
```

Access current cell state.

Definition at line 41 of file cell.cpp.

References m_state.

5.1.3.4 setState()

```
void Cell::setState (
            unsigned int state )
```

Set temporary state.

To change current cell state, use setState(unsigned int state) then validState().

Parameters

state	New state

Definition at line 22 of file cell.cpp.

References m_nextState.

5.1.3.5 validState()

```
void Cell::validState ( )
```

Validate temporary state.

To change current cell state, use setState(unsigned int state) then validState().

Definition at line 33 of file cell.cpp.

References m_nextState, and m_state.

5.1.4 Member Data Documentation

5.1.4.1 m_neighbours

```
QVector<const Cell*> Cell::m_neighbours  [private]
```

Cell's neighbours.

Definition at line 26 of file cell.h.

Referenced by addNeighbour(), and getNeighbours().

5.1.4.2 m_nextState

```
unsigned int Cell::m_nextState  [private]
```

Temporary state, before validation.

Definition at line 24 of file cell.h.

Referenced by setState(), and validState().

5.1.4.3 m_state

```
unsigned int Cell::m_state  [private]
```

Current state.

Definition at line 23 of file cell.h.

Referenced by getState(), and validState().

The documentation for this class was generated from the following files:

- cell.h
- cell.cpp

5.2 CellHandler Class Reference

Cell container and cell generator.

```
#include <cellhandler.h>
```

Classes

- class iterator

  Implementation of iterator design pattern.

Public Member Functions

- CellHandler (QString filename)

  Construct all the cells from the json file given.
- virtual ~CellHandler ()

  Destroys all cells in the CellHandler.
- Cell * getCell (const QVector< unsigned int > position) const

  Access the cell to the given position.
- void nextStates ()

  Valid the state of all cells.
- iterator begin ()

  Give the iterator which corresponds to the current CellHandler.
- bool end ()

  End condition of the iterator.

Private Member Functions

- bool load (const QJsonObject &json)

  Load the config file in the CellHandler.
- void foundNeighbours ()

  Set the neighbours of each cells.
- void positionIncrement (QVector< unsigned int > &pos, unsigned int value=1) const

  Increment the QVector given by the value choosen.
- QVector< QVector< unsigned int > > * getListNeighboursPositionsRecursive (const QVector< unsigned int > position, unsigned int dimension, QVector< unsigned int > lastAdd) const

  Recursive function which browse the position possibilities tree.
- QVector< QVector< unsigned int > > & getListNeighboursPositions (const QVector< unsigned int > position) const

  Prepare the call of the recursive version of itself.

Private Attributes

- QVector< unsigned int > m_dimensions

  Vector of x dimensions.
- QMap< QVector< unsigned int >, Cell *> m_cells

  Map of cells, with a x dimensions vector as key.

5.2.1 Detailed Description

Cell container and cell generator.

Generate cells from a json file.

Definition at line 18 of file cellhandler.h.

5.2.2 Constructor & Destructor Documentation

5.2.2.1 CellHandler()

```
CellHandler::CellHandler (
            QString filename )
```

Construct all the cells from the json file given.

The size of "cells" array must be the product of all dimensions (60 in the following example). Typical Json file:

```
{
"dimensions":"3x4x5",
"cells":[0,1,4,4,2,5,3,4,2,4,
         4,2,5,0,0,0,0,0,0,0,
         2,4,1,1,1,1,1,2,1,1,
         0,0,0,0,0,0,2,2,2,2,
         3,4,5,1,2,0,9,0,0,0,
         1,2,0,0,0,0,1,2,3,2]
}
```

Parameters

filename	Json file which contains the description of all the cells

Definition at line 23 of file cellhandler.cpp.

References foundNeighbours(), and load().

5.2.2.2 ~CellHandler()

```
CellHandler::~CellHandler ( )  [virtual]
```

Destroys all cells in the CellHandler.

Definition at line 56 of file cellhandler.cpp.

References m_cells.

5.2.3 Member Function Documentation

5.2.3.1 begin()

```
CellHandler::iterator CellHandler::begin ( )
```

Give the iterator which corresponds to the current CellHandler.

Definition at line 87 of file cellhandler.cpp.

5.2.3.2 end()

```
bool CellHandler::end ( )
```

End condition of the iterator.

See iterator::operator!=(bool finished) for further information.

Definition at line 97 of file cellhandler.cpp.

5.2.3.3 foundNeighbours()

```
void CellHandler::foundNeighbours ( )  [private]
```

Set the neighbours of each cells.

Careful, this is in O(n∗3^d), with n the number of cells and d the number of dimensions

Definition at line 192 of file cellhandler.cpp.

References getListNeighboursPositions(), m_cells, m_dimensions, and positionIncrement().

Referenced by CellHandler().

5.2.3.4 getCell()

```
Cell * CellHandler::getCell (
            const QVector< unsigned int > position ) const
```

Access the cell to the given position.

Definition at line 67 of file cellhandler.cpp.

References m_cells.

5.2.3.5 getListNeighboursPositions()

```
QVector< QVector< unsigned int > > & CellHandler::getListNeighboursPositions (
            const QVector< unsigned int > position ) const  [private]
```

Prepare the call of the recursive version of itself.

Parameters

position	Position of the central cell (x1,x2,x3,..,xn)

Returns

List of positions

Definition at line 253 of file cellhandler.cpp.

References getListNeighboursPositionsRecursive().

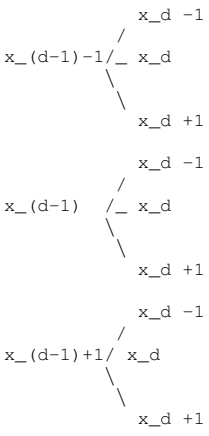
Referenced by foundNeighbours().

5.2.3.6 getListNeighboursPositionsRecursive()

```
QVector< QVector< unsigned int > > * CellHandler::getListNeighboursPositionsRecursive (
            const QVector< unsigned int > position,
            unsigned int dimension,
            QVector< unsigned int > lastAdd ) const  [private]
```

Recursive function which browse the position possibilities tree.

Careful, the complexity is in O(3^dimension)
Piece of the tree:



The path in the tree to reach the leaf give the position

Parameters

position	Position of the cell
dimension	Current working dimension (number of the digit). Dimension = 2 <=> working on x2 coordinates on (x1, x2, x3, ..., xn) vector
lastAdd	Last position added. Like the father node of the new tree

Returns

List of position

Definition at line 295 of file cellhandler.cpp.

References m_dimensions.

Referenced by getListNeighboursPositions().

5.2.3.7 load()

```
bool CellHandler::load (
              const QJsonObject & json )  [private]
```

Load the config file in the CellHandler.

Exemple of a way to print cell states :

```
position.clear();
for (unsigned short i = 0; i < m_dimensions.size(); i++)
{
     position.push_back(0);
}
for (unsigned int j = 0; j < m_cells.size(); j++)
{
     std::cout << m_cells.value(position)->getState() << " ";
     position.replace(0, position.at(0)+1);
     for (unsigned short i = 0; i < m_dimensions.size(); i++)
     {
         if (position.at(i) >= m_dimensions.at(i))
         {
             position.replace(i, 0);
             std::cout << std::endl;
             if (i + 1 != m_dimensions.size())
                 position.replace(i+1, position.at(i+1)+1);
         }

     }
}
```

Parameters

json	Json Object which contains the grid configuration

Returns

False if the Json Object is not correct

Definition at line 133 of file cellhandler.cpp.

References m_cells, m_dimensions, and positionIncrement().

Referenced by CellHandler().

5.2.3.8 nextStates()

```
void CellHandler::nextStates ( )
```

Valid the state of all cells.

Definition at line 76 of file cellhandler.cpp.

References m_cells.

5.2.3.9 positionIncrement()

```
void CellHandler::positionIncrement (
            QVector< unsigned int > & pos,
            unsigned int value = 1 ) const  [private]
```

Increment the QVector given by the value choosen.

Careful, when the position reach the maximum, it goes to zero without leaving the function

Parameters

pos	Position to increment
value	Value to add, 1 by default

Definition at line 223 of file cellhandler.cpp.

References m_dimensions.

Referenced by foundNeighbours(), and load().

5.2.4 Member Data Documentation

5.2.4.1 m_cells

```
QMap<QVector<unsigned int>, Cell* > CellHandler::m_cells  [private]
```

Map of cells, with a x dimensions vector as key.

Definition at line 75 of file cellhandler.h.

Referenced by foundNeighbours(), getCell(), load(), nextStates(), CellHandler::iterator::operator*(), CellHandler::iterator::operator->(), and ~CellHandler().

5.2.4.2 m_dimensions

```
QVector<unsigned int> CellHandler::m_dimensions  [private]
```

Vector of x dimensions.

Definition at line 74 of file cellhandler.h.

Referenced by foundNeighbours(), getListNeighboursPositionsRecursive(), CellHandler::iterator::iterator(), load(), CellHandler::iterator::operator++(), and positionIncrement().

The documentation for this class was generated from the following files:

- cellhandler.h
- cellhandler.cpp

5.3 CellHandler::iterator Class Reference

Implementation of iterator design pattern.

`#include <cellhandler.h>`

Public Member Functions

- iterator (const CellHandler *handler)

  Construct an initial iterator to browse the CellHandler.
- iterator & operator++ ()

  Increment the current position and handle dimension changes.
- Cell * operator-> () const

  Get the current cell.
- Cell * operator* () const
- bool operator!= (bool finished) const
- unsigned int changedDimension () const

  Return the number of dimensions we change.

Private Attributes

- const CellHandler * m_handler

  CellHandler to go through.
- QVector< unsigned int > m_position

  Current position of the iterator.
- bool m_finished = false

  If we reach the last position.
- QVector< unsigned int > m_zero

  Nul vector of the good dimension (depend of m_handler)
- unsigned int m_changedDimension

  Save the number of dimension change.

Friends

- class CellHandler

5.3.1 Detailed Description

Implementation of iterator design pattern.

Example of use:

```
CellHandler handler("file.atc");
for (CellHandler::iterator it = handler.begin(); it != handler.end(); ++it)
{
     for (unsigned int i = 0; i < it.changedDimension(); i++)
         std::cout << std::endl;
     std::cout << it->getState() << " ";
}
```

This code will print each cell states and go to a new line when there is a change of dimension. So if there is 3 dimensions, there will be a empty line between 2D groups.

Definition at line 37 of file cellhandler.h.

5.3.2 Constructor & Destructor Documentation

5.3.2.1 iterator()

```
CellHandler::iterator::iterator (
            const CellHandler * handler )
```

Construct an initial iterator to browse the CellHandler.

Parameters

handler	CellHandler to browse

Definition at line 335 of file cellhandler.cpp.

References CellHandler::m_dimensions, m_position, and m_zero.

5.3.3 Member Function Documentation

5.3.3.1 changedDimension()

```
unsigned int CellHandler::iterator::changedDimension ( ) const
```

Return the number of dimensions we change.

For example, if we were at the (3,4,4) cell, and we incremented the position, we are now at (4,0,0), and changed↩ Dimension return 2 (because of the 2 zeros).

Definition at line 396 of file cellhandler.cpp.

References m_changedDimension.

Referenced by operator!=().

5.3.3.2 operator"!=()

```
bool CellHandler::iterator::operator!= (
            bool finished ) const  [inline]
```

Definition at line 47 of file cellhandler.h.

References changedDimension(), and m_finished.

5.3.3.3 operator∗()

```
Cell * CellHandler::iterator::operator* ( ) const
```

Definition at line 385 of file cellhandler.cpp.

References CellHandler::m_cells, m_handler, and m_position.

5.3.3.4 operator++()

```
CellHandler::iterator & CellHandler::iterator::operator++ ( )
```

Increment the current position and handle dimension changes.

Definition at line 349 of file cellhandler.cpp.

References m_changedDimension, CellHandler::m_dimensions, m_finished, m_handler, m_position, and m_zero.

5.3.3.5 operator->()

```
Cell * CellHandler::iterator::operator-> ( ) const
```

Get the current cell.

Definition at line 377 of file cellhandler.cpp.

References CellHandler::m_cells, m_handler, and m_position.

5.3.4 Friends And Related Function Documentation

5.3.4.1 CellHandler

```
friend class CellHandler  [friend]
```

Definition at line 39 of file cellhandler.h.

5.3.5 Member Data Documentation

5.3.5.1 m_changedDimension

```
unsigned int CellHandler::iterator::m_changedDimension  [private]
```

Save the number of dimension change.

Definition at line 55 of file cellhandler.h.

Referenced by changedDimension(), and operator++().

5.3.5.2 m_finished

```
bool CellHandler::iterator::m_finished = false  [private]
```

If we reach the last position.

Definition at line 53 of file cellhandler.h.

Referenced by operator!=(), and operator++().

5.3.5.3 m_handler

```
const CellHandler* CellHandler::iterator::m_handler  [private]
```

CellHandler to go through.

Definition at line 51 of file cellhandler.h.

Referenced by operator*(), operator++(), and operator->().

5.3.5.4 m_position

```
QVector<unsigned int> CellHandler::iterator::m_position  [private]
```

Current position of the iterator.

Definition at line 52 of file cellhandler.h.

Referenced by iterator(), operator*(), operator++(), and operator->().

5.3.5.5 m_zero

```
QVector<unsigned int> CellHandler::iterator::m_zero  [private]
```

Nul vector of the good dimension (depend of m_handler)

Definition at line 54 of file cellhandler.h.

Referenced by iterator(), and operator++().

The documentation for this class was generated from the following files:

- cellhandler.h
- cellhandler.cpp

Chapter 6

File Documentation

6.1 cell.cpp File Reference

```
#include "cell.h"
```

6.2 cell.cpp

```
00001 #include "cell.h"
00002
00008 Cell::Cell(unsigned int state):
00009     m_state(state), m_nextState(state)
00010 {
00011
00012 }
00013
00022 void Cell::setState(unsigned int state)
00023 {
00024     m_nextState = state;
00025 }
00026
00033 void Cell::validState()
00034 {
00035     m_state = m_nextState;
00036 }
00037
00041 unsigned int Cell::getState() const
00042 {
00043     return m_state;
00044 }
00045
00052 bool Cell::addNeighbour(const Cell* neighbour)
00053 {
00054     if (m_neighbours.count(neighbour))
00055         return false;
00056     m_neighbours.push_back(neighbour);
00057     return true;
00058 }
00059
00063 QVector<const Cell*> Cell::getNeighbours() const
00064 {
00065     return m_neighbours;
00066 }
```

6.3 cell.h File Reference

```
#include <QVector>
#include <QDebug>
```

Classes

- class Cell

  Contains the state, the next state and the neighbours.

6.4 cell.h

```
00001 #ifndef CELL_H
00002 #define CELL_H
00003
00004 #include <QVector>
00005 #include <QDebug>
00006
00010 class Cell
00011 {
00012 public:
00013     Cell(unsigned int state = 0);
00014
00015     void setState(unsigned int state);
00016     void validState();
00017     unsigned int getState() const;
00018
00019     bool addNeighbour(const Cell* neighbour);
00020     QVector<const Cell*> getNeighbours() const;
00021
00022 private:
00023     unsigned int m_state;
00024     unsigned int m_nextState;
00025
00026     QVector<const Cell*> m_neighbours;
00027 };
00028
00029 #endif // CELL_H
```

6.5 cellhandler.cpp File Reference

```
#include <iostream>
#include "cellhandler.h"
```

6.6 cellhandler.cpp

```
00001 #include <iostream>
00002 #include "cellhandler.h"
00003
00023 CellHandler::CellHandler(QString filename)
00024 {
00025     QFile loadFile(filename);
00026     if (!loadFile.open(QIODevice::ReadOnly | QIODevice::Text)) {
00027         qWarning("Couldn't open given file.");
00028         throw QString(QObject::tr("Couldn't open given file"));
00029     }
00030
00031     QJsonParseError parseErr;
00032     QJsonDocument loadDoc(QJsonDocument::fromJson(loadFile.readAll(), &parseErr));
00033
00034
00035
00036     if (loadDoc.isNull() || loadDoc.isEmpty()) {
00037         qWarning() << "Could not read data : ";
00038         qWarning() << parseErr.errorString();
00039     }
00040
00041     // Loadding of the json file
00042     if (!load(loadDoc.object()))
00043     {
00044         qWarning("File not valid");
```

```
00045         throw QString(QObject::tr("File not valid"));
00046     }
00047
00048     foundNeighbours();
00049
00050
00051 }
00052
00056 CellHandler::~CellHandler()
00057 {
00058     for (QMap<QVector<unsigned int>, Cell* >::iterator it = m_cells.begin(); it !=
      m_cells.end(); ++it)
00059     {
00060         delete it.value();
00061     }
00062 }
00063
00067 Cell *CellHandler::getCell(const QVector<unsigned int> position) const
00068 {
00069     return m_cells.value(position);
00070 }
00071
00076 void CellHandler::nextStates()
00077 {
00078     for (QMap<QVector<unsigned int>, Cell* >::iterator it = m_cells.begin(); it !=
      m_cells.end(); ++it)
00079     {
00080         it.value()->validState();
00081     }
00082 }
00083
00087 CellHandler::iterator CellHandler::begin()
00088 {
00089     return iterator(this);
00090 }
00091
00097 bool CellHandler::end()
00098 {
00099     return true;
00100 }
00101
00133 bool CellHandler::load(const QJsonObject &json)
00134 {
00135     if (!json.contains("dimensions") || !json["dimensions"].isString())
00136         return false;
00137
00138     // RegExp to validate dimensions field format : "10x10"
00139     QRegExpValidator dimensionValidator(QRegExp("([0-9]*x?)*"));
00140     QString stringDimensions = json["dimensions"].toString();
00141     int pos= 0;
00142     if (dimensionValidator.validate(stringDimensions, pos) != QRegExpValidator::Acceptable)
00143         return false;
00144
00145     // Split of dimensions field : "10x10" => "10", "10"
00146     QRegExp rx("x");
00147     QStringList list = json["dimensions"].toString().split(rx, QString::SkipEmptyParts);
00148
00149     unsigned int product = 1;
00150     // Dimensions construction
00151     for (unsigned int i = 0; i < list.size(); i++)
00152     {
00153         product = product * list.at(i).toInt();
00154         m_dimensions.push_back(list.at(i).toInt());
00155     }
00156     if (!json.contains("cells") || !json["cells"].isArray())
00157         return false;
00158
00159     QJsonArray cells = json["cells"].toArray();
00160     if (cells.size() != product)
00161         return false;
00162
00163     QVector<unsigned int> position;
00164     // Set position vector to 0
00165     for (unsigned short i = 0; i < m_dimensions.size(); i++)
00166     {
00167         position.push_back(0);
00168     }
00169
00170     // Creation of cells
00171     for (unsigned int j = 0; j < cells.size(); j++)
00172     {
00173         if (!cells.at(j).isDouble())
00174             return false;
00175         if (cells.at(j).toDouble() < 0)
00176             return false;
00177         m_cells.insert(position, new Cell(cells.at(j).toDouble()));
00178
```

```
00179         positionIncrement(position);
00180     }
00181
00182     return true;
00183
00184 }
00185
00192 void CellHandler::foundNeighbours()
00193 {
00194     QVector<unsigned int> currentPosition;
00195     // Set position vector to 0
00196     for (unsigned short i = 0; i < m_dimensions.size(); i++)
00197     {
00198         currentPosition.push_back(0);
00199     }
00200     // Modification of all the cells
00201     for (unsigned int j = 0; j < m_cells.size(); j++)
00202     {
00203         // Get the list of the neighbours positions
00204         // This function is recursive
00205         QVector<QVector<unsigned int> > listPosition(getListNeighboursPositions(
      currentPosition));
00206
00207         // Adding neighbours
00208         for (unsigned int i = 0; i < listPosition.size(); i++)
00209             m_cells.value(currentPosition)->addNeighbour(m_cells.value(listPosition.at(i)));
00210
00211         positionIncrement(currentPosition);
00212     }
00213 }
00214
00223 void CellHandler::positionIncrement(QVector<unsigned int> &pos, unsigned int
      value) const
00224 {
00225     pos.replace(0, pos.at(0) + value); // adding the value to the first digit
00226
00227     // Carry management
00228     for (unsigned short i = 0; i < m_dimensions.size(); i++)
00229     {
00230         if (pos.at(i) >= m_dimensions.at(i) && pos.at(i) <
      m_dimensions.at(i)*2)
00231         {
00232             pos.replace(i, 0);
00233             if (i + 1 != m_dimensions.size())
00234                 pos.replace(i+1, pos.at(i+1)+1);
00235         }
00236         else if (pos.at(i) >= m_dimensions.at(i))
00237         {
00238             pos.replace(i, pos.at(i) - m_dimensions.at(i));
00239             if (i + 1 != m_dimensions.size())
00240                 pos.replace(i+1, pos.at(i+1)+1);
00241             i--;
00242         }
00243
00244     }
00245 }
00246
00253 QVector<QVector<unsigned int> >& CellHandler::getListNeighboursPositions
      (const QVector<unsigned int> position) const
00254 {
00255     QVector<QVector<unsigned int> > *list = getListNeighboursPositionsRecursive
      (position, position.size(), position);
00256     // We remove the position of the cell
00257     list->removeAll(position);
00258     return *list;
00259 }
00260
00295 QVector<QVector<unsigned int> >*
      CellHandler::getListNeighboursPositionsRecursive(const
      QVector<unsigned int> position, unsigned int dimension, QVector<unsigned int> lastAdd) const
00296 {
00297     if (dimension == 0) // Stop condition
00298     {
00299         QVector<QVector<unsigned int> > *list = new QVector<QVector<unsigned int> >;
00300         return list;
00301     }
00302     QVector<QVector<unsigned int> > *listPositions = new QVector<QVector<unsigned int> >;
00303
00304     QVector<unsigned int> modifiedPosition(lastAdd);
00305
00306     // "x_d - 1" tree
00307     if (modifiedPosition.at(dimension-1) != 0) // Avoid "negative" position
00308         modifiedPosition.replace(dimension-1, position.at(dimension-1) - 1);
00309     listPositions->append(*getListNeighboursPositionsRecursive(position,
       dimension -1, modifiedPosition));
00310     if (!listPositions->count(modifiedPosition))
00311         listPositions->push_back(modifiedPosition);
```

```
00312
00313     // "x_d" tree
00314     modifiedPosition.replace(dimension-1, position.at(dimension-1));
00315     listPositions->append(*getListNeighboursPositionsRecursive(position,
      dimension -1, modifiedPosition));
00316     if (!listPositions->count(modifiedPosition))
00317         listPositions->push_back(modifiedPosition);
00318
00319     // "x_d + 1" tree
00320     if (modifiedPosition.at(dimension -1) + 1 < m_dimensions.at(dimension-1)) // Avoid position
      out of the cell space
00321         modifiedPosition.replace(dimension-1, position.at(dimension-1) +1);
00322     listPositions->append(*getListNeighboursPositionsRecursive(position,
      dimension -1, modifiedPosition));
00323     if (!listPositions->count(modifiedPosition))
00324         listPositions->push_back(modifiedPosition);
00325
00326     return listPositions;
00327
00328 }
00329
00335 CellHandler::iterator::iterator(const CellHandler *handler):
00336         m_handler(handler), m_changedDimension(0)
00337 {
00338     // Initialisation of m_position
00339     for (unsigned short i = 0; i < handler->m_dimensions.size(); i++)
00340     {
00341         m_position.push_back(0);
00342     }
00343     m_zero = m_position;
00344 }
00345
00349 CellHandler::iterator &CellHandler::iterator::operator++
      ()
00350 {
00351     m_position.replace(0, m_position.at(0) + 1); // adding the value to the first digit
00352
00353     m_changedDimension = 0;
00354     // Carry management
00355     for (unsigned short i = 0; i < m_handler->m_dimensions.size(); i++)
00356     {
00357         if (m_position.at(i) >= m_handler->m_dimensions.at(i))
00358         {
00359             m_position.replace(i, 0);
00360             m_changedDimension++;
00361             if (i + 1 != m_handler->m_dimensions.size())
00362                 m_position.replace(i+1, m_position.at(i+1)+1);
00363         }
00364
00365     }
00366     // If we return to zero, we have finished
00367     if (m_position == m_zero)
00368         m_finished = true;
00369
00370     return *this;
00371
00372 }
00373
00377 Cell *CellHandler::iterator::operator->() const
00378 {
00379     return m_handler->m_cells.value(m_position);
00380 }
00381
00385 Cell *CellHandler::iterator::operator*() const
00386 {
00387     return m_handler->m_cells.value(m_position);
00388 }
00389
00396 unsigned int CellHandler::iterator::changedDimension() const
00397 {
00398     return m_changedDimension;
00399 }
```

6.7 cellhandler.h File Reference

```
#include <QString>
#include <QFile>
#include <QJsonDocument>
#include <QtWidgets>
```

```
#include <QMap>
#include <QRegExpValidator>
#include "cell.h"
```

Classes

- class CellHandler

  Cell container and cell generator.
- class CellHandler::iterator

  Implementation of iterator design pattern.

6.8 cellhandler.h

```
00001 #ifndef CELLHANDLER_H
00002 #define CELLHANDLER_H
00003
00004 #include <QString>
00005 #include <QFile>
00006 #include <QJsonDocument>
00007 #include <QtWidgets>
00008 #include <QMap>
00009 #include <QRegExpValidator>
00010
00011 #include "cell.h"
00012
00018 class CellHandler
00019 {
00020 public:
00037     class iterator
00038     {
00039         friend class CellHandler;
00040     public:
00041         iterator(const CellHandler* handler);
00042
00043         iterator& operator++();
00044         Cell* operator->() const;
00045         Cell* operator*() const;
00046
00047         bool operator!=(bool finished) const { return (m_finished != finished); }
00048         unsigned int changedDimension() const;
00049
00050     private:
00051         const CellHandler *m_handler;
00052         QVector<unsigned int> m_position;
00053         bool m_finished = false;
00054         QVector<unsigned int> m_zero;
00055         unsigned int m_changedDimension;
00056     };
00057
00058     CellHandler(QString filename);
00059     virtual ~CellHandler();
00060
00061     Cell* getCell(const QVector<unsigned int> position) const;
00062     void nextStates();
00063
00064     iterator begin();
00065     bool end();
00066
00067 private:
00068     bool load(const QJsonObject &json);
00069     void foundNeighbours();
00070     void positionIncrement(QVector<unsigned int> &pos, unsigned int value = 1) const;
00071     QVector<QVector<unsigned int> > *getListNeighboursPositionsRecursive
      (const QVector<unsigned int> position, unsigned int dimension, QVector<unsigned int> lastAdd) const;
00072     QVector<QVector<unsigned int> > &getListNeighboursPositions(const
      QVector<unsigned int> position) const;
00073
00074     QVector<unsigned int> m_dimensions;
00075     QMap<QVector<unsigned int>, Cell* > m_cells;
00076 };
00077
00078 #endif // CELLHANDLER_H
```

6.9 homepage.md File Reference

6.10 homepage.md

```
00001 \page Presentation
00002 # What is AutoCell
00003 The purpose of this project is to create a Cellular Automate Simulator.
```

6.11 main.cpp File Reference

```
#include <QApplication>
#include <QDebug>
#include "cell.h"
```

Functions

- int main (int argc, char ∗argv[ ])

6.11.1 Function Documentation

6.11.1.1 main()

```
int main (
              int argc,
              char * argv[] )
```

Definition at line 5 of file main.cpp.

6.12 main.cpp

```
00001 #include <QApplication>
00002 #include <QDebug>
00003 #include "cell.h"
00004
00005 int main(int argc, char * argv[])
00006 {
00007     QApplication app(argc, argv);
00008
00009     return app.exec();
00010 }
```

6.13 README.md File Reference

6.14 README.md

```
00001 \mainpage
00002
00003 To generate the Documentation, go in Documentation directory and run `make`.
00004
00005 It will generate html doc (in `output/html/index.html`) and latex doc (pdf output directely in
       Documentation directory (`docPdf.pdf`).
```

Index