Bachelor Thesis

Development of output Routines for a simulation Environment in virtual Embryology and Establishment of autoregulated Growth

Markus Dauschan
Supervisor: Mag. Dr. Thomas Schmickl
Course: Modul Modellierung Biologischer Systeme
Karl-Franzens-Universität Graz

June 20, 2010
Contents

1 Abstract 2
  1.1 Abstract ............................................. 2
  1.2 Kurzzusammenfassung ............................... 2

2 The Model 3
  2.1 Concept .............................................. 3
  2.2 Implementation in NetLogo ......................... 5
  2.3 Implementation in C++ ............................. 5

3 Changes to the C++ Implementation 7
  3.1 The epsplot Library ............................... 7
  3.2 Class Plot .......................................... 7
  3.3 A global Settings File ............................ 9
  3.4 Automation .......................................... 10
  3.5 A wider View ....................................... 13
  3.6 Modifications to Diffusion ....................... 14

4 Working with the Program 16
  4.1 Establishment of autoregulated Growth ............ 16
  4.2 Time Measurements ............................... 19

5 Summary 20

6 Appendix 21
  6.1 References ........................................... 21

List of Tables 21

List of Figures 22
1

Abstract

1.1 Abstract

For my Bachelor Thesis I worked on the C++ implementation of a bottom up multi agent model representing virtual embryos that are able to build up Artificial Neural Networks. The model is based on the findings of evolutionary developmental biology and consists of cells, patches and cell emitted diffusing substances that act on cells as morphogens. My work included inter alia the development and integration of a graphical output routine, the incorporation of a global settings file, several increases in usability of the program and an automated production of a movie of embryonal growth and ultimately the preparation of the program for large batch automated evolution processes.

1.2 Kurzzusammenfassung

2

The Model

2.1 Concept

The model is based on the findings of evolutionary developmental biology (evo devo) \cite{2} \cite{3} and is determined to emphasize the fundamental importance of the development process additionally to genetic material, forming virtual embryos and Artificial Neural Networks (ANN). It was originally developed by Thenius, R. in 2008 and extended by Bodi, M. 2009 \cite{1}. Several virtual genes are forming toolkits, turning on and off genes that are needed for certain adaptive reactions to the development surrounding, leading to several shapes and forms emerging from the same genetic base. The aim of the model is amongst others, to create certain patterns of ANN that match Neural Networks found in real embryos. This is achieved by universally provided, respectively cell emitted substances which act as morphogens and hence form the embryo and the resulting neural network by triggering certain cellular behaviors. The concept used in this model is based on multi-agent simulations wherein each cell acts and reacts individually to influences from its surroundings. These influences may consist of the existence of specific morphogens near the cell. Potential reactions of the cell on these influences are proliferation, development to a node within the ANN or death. Apart from the cells ability to emit morphogens itself, and therefore trigger these behaviors, or the emittance of further substances within other cells in its vicinity. This process is considered to be reversible. All possible actions of the cell are defined in the genome contained by each cell. Following the concepts of evo devo, an individual embryo is then exposed to evolution altering its genome. The resulting individuals are then measured up against certain fitness ideals and allowed to reproduce or not, depending on their score. Genomes are considered to be diploid, therefore the mating process takes one gamet from each individual which are established via a simplified crossover process. Those two gamets are then merged into a novel offspring genome. Application of this virtual evolution process results in self organized behaviour of the embryo cells, forming shapes and networks that can be compared to findings in real embryos.
Figure 2.1: Model overview indicating feedbacks between various effects in the model [4].

Figure 2.2: Schematic of the evolutionary optimization process and extraction of the ANN [4].
2.2 Implementation in NetLogo

The implementation in NetLogo considers all main concepts described above. To simplify the model for simulations, it is rendered to be discrete, and two-dimensional. The virtual world containing the embryo consists of a grid of ‘patches’ which may or may not hold a cell of the embryo. Each of these patches may also contain a certain amount of morphogens, even if it is not occupied by a cell. Diffusion of the morphogens is also rendered to be discrete resulting in the possibility to monitor the substance levels for every timestep in every cell or patch [1]. If a cell duplicates, the additional cell will move all cells in the direction that holds the least number of cells. This results from the consideration that pushing other cells always takes place in the direction of the least pressure. Genomes defining the emission of morphogens and rates that trigger cells to emit morphogens or other actions are the base of these simulations. These genomes are files that are read in by NetLogo, enabling an easy switch between a vast variety of embryo shapes. Every genome holds several genes which consist of a quadruple of unsigned integer values. The first number contained by the gene represents the index of a substance which triggers the gene followed by minimum and maximum values of the substance received. The last parameter stands for the reaction of the cell to the received morphogen. As shown above this can be proliferation, emittance of another morphogen, building of links and cell death. The NetLogo implementation also features an evolutionary process, enabling the automated creation of genomes that is followed by a selection of fitness. In this case fitness is related to the shape the cells form given a certain genome.

2.3 Implementation in C++

The implementation of the model core in C++ was done in 2009 by Schwarzer, C. Following an object oriented approach the C++ implementation of the model uses the following main classes separated into two main routines:

- NeuralGenesis
  - class NeuralGenesis
  - struct Patch
  - class Cell
  - class Genome
- RTNN (Recurrent Time Neural Network)
  - class rttn
  - class Neuron
  - struct Link
  - class ActivationFunctions
  - struct fixedtype
Neural Genesis is the main interface class for embryonal development. It generates a grid of patches which is occupied at first with only one cell. During the process of embryogenesis the cell proliferates and can be set to emit morphogens to influence the shape developed by the group of cells.

RTNN stands for Recurrent Time Neural Network which is the second main routine in the program. It is basically intended to create and extract the neural network resulting from the 'NeuralGenesis' process. In future development of the program this class can act as an interface to a neural net interpreter. All cells are transformed into neurons and links between them are established according to the data obtained in the 'NeuralGenesis' process.

One main difference between the NetLogo and the C++ implementation also lies within the genomes. The C++ genome holds an additional fifth parameter $c \in [0, ..., 255]$ which stands for the intensity the product is expressed. In case of releasing another morphogen, the parameter simply corresponds to the release rate.

During my work I focused on the extension and debugging of the NeuralGenesis part of the program and neglected the development of Neural Networks to favor the emergence of suitable embryonal shapes first.
3

Changes to the C++ Implementation

3.1 The epsplot Library

One major aim of my work was to implement an output routine designed for the visualization of the embryogenesis process using the .eps format. During my research I found a C++ plot library written by Niemitalo, O. which is freely available and usable [5]. This reduced my work to the creation of an interface class 'plot' collecting all relevant data from the main program and delivering it to the plot engine in the correct format. Nevertheless some simplifications had to be considered to reduce the complexity of the output and also make way for large scale patch operations resulting in multiple plots, one for every timestep in 'NeuralGenesis'.

Considering the process is taking place in only two dimensions, a cell is being represented by a circle, while patches are represented by a grid lying underneath the cells. Considering this representation of the simulation, only one morphogen gradient may be displayed in a single timestep plot. The gradient is represented by coloration of the grid from green to white, green corresponding to a high level of morphogen concentration.

3.2 Class Plot

'Plot' is the interface class between the main program and the epsplot library. The whole class was designed to be initialized as a pointer within an overhead routine and be renewed in every timestep. It consists of the following methods:

\[
\text{plot ( NeuralGenesis * genesis, char * title )}
\]

The parameter constructor of class 'plot' takes a pointer to an instance of 'NeuralGenesis' that contains the data used for the plot and an array of characters consisting of the title for the plot. The delivered pointer to 'NeuralGenesis' is stored as a private member variable to be easily accessed. The title is handed over directly to the epsplot library resulting in the filename for the single plot, which usually is the number of the timestep that is currently being plotted.
void plotgrid ( )

Plotts a basic grid of grey lines into the .eps file representing the array of patches which can later be filled with color (gradient) or circles (cells).

void plotcell ( float x1, float y1 )

This method plots single cells into the grid using x1 and y1 as coordinates within the grid. The procedure includes the creation of an array of 75 points which are aligned radial around a center point. This array is then drawn into the grid using the drawline routine from the epsplot library.

void plotgradient ( int xpos, int ypos, int level )

Coloring a single patch within the grid in green tones according to a morphogen gradient delivered by NeuralGenesis this method calculates the corresponding color using the following procedure:

```c
// calculating color according to level
float color;

if ( level == 0 )
{
    _plot -> setRGBColor ( 1., 1., 1. );
}
else
{
    color = 1 - ( level / 256. );

    _plot -> setRGBColor ( color, 1., color );
}
```

Calculation of the color level within the plotgradient method of class 'Plot'.

This results in a lush green when the delivered level is very high (near to 255) and the lower the delivered level the more it tends to white (Figure 3.1).

void finishplot ( )

This Method is called when all cells and optionally the grid and gradient are already drawn. It is used to draw the borderlines of the plot.

void plotembryo ( int gradient, bool grid )

This is the main plotting procedure which is called by an overhead routine to draw the entire plot. Its input parameters are the index of the substance to plot the gradient for and a bool variable determining if a grid is to be drawn.
Figure 3.1: Plot of a single timestep of a simulation using the C++ implementation including the gradient of substance 0. Cells are represented by circles, patches are represented by a grid lying underneath the cells. The color gradient indicates the concentration of morphogen 0. A lush green corresponds to a high concentration and white to a very low concentration of the morphogen.

or not. If the index of the gradient is negative or above the total number of substances used in the simulation, no gradient is plotted. Method ‘plotembryo’ iterates through the grid of patches stored in ‘NeuralGenesis’ and determines whether a certain patch holds a cell and the level of the substance used for the gradient plot. These informations are then delivered to the methods ‘plotcell’ and ‘plotgradient’. If ‘bool grid’ is set to ‘true’ also a grid is drawn underneath the cells and the gradient. The resulting plots look similar to those produced by the NetLogo implementation (Figure 3.1).

3.3 A global Settings File

One of the main problems in the original C++ implementation of the model that I noticed right at the begin of my work is:
Encapsulation, a main concept of object oriented programming, was seriously neglected and abandoned in favor of speed. This resulted in quite a number of global constants of the type ’static const int’ or similar which may be accessed by
every part of the program very fast. One major disadvantage of such constants is that they need to be declared and initialized at the same point in the program and cannot be changed at runtime. Therefore it is not possible to have a settings file holding all these values which is parsed by the program. The only way to change these values via a global settings file without modifying the basic structure of the program is to change them right in the source code. Using a shellscript this can be achieved in an automated way but of course it comes at a price: The source code has to be recompiled every time those global settings change, but the original, fast structure of the program core is conserved this way. Also considering a large number of individual embryos calculated in one batch operation, which is a more distant aim of the implementation, the time it takes the program to compile can be neglected.

Reading the values from the settings file and writing them into the source files is achieved by the use of the unix stream editor ‘sed’.

```
#!/bin/bash

echo configuring NeuralGenesis - setting globals

# getting gridsize
TEMP=$(sed -n 3p settings)
# writing gridsize
sed -i "s/^.*static const int GRIDSIZE.*$/ static const int GRIDSIZE = $TEMP;/" NeuralGenesis/Genome.h

# getting Number of Substances
TEMP=$(sed -n 5p settings)
# writing Number of Substances
sed -i "s/^.*static const int NUMSUBSTANCES.*$/ static const int NUMSUBSTANCES = $TEMP;/" NeuralGenesis/Genome.h
.
.
.
```

This shell script reads the values for the global variables from the settings file and writes them into the source code using the unix stream editor (‘sed’).

It is assumed that the replaced string exists only in one position within the target file. This is indeed the case as the global variables are declared and initiated only once in the source code. The procedure of changing the values of the constants is done by the shellscript ‘configure.sh’.

### 3.4 Automation

A more extensive automation is achieved by another shellscript called ‘run.sh’, which executes first ‘configure.sh’ and then the makefile. Another aim of my work was to achieve the automated output of one individual embryogenesis process in .eps files and make a film out of the single timesteps. By running ‘run.sh’
with optional parameters `-m` or `-q` this can be done automatically:

- **run.sh**
  The values of the global constants in the source code are replaced by the ones in the file 'settings' using 'configure.sh'. After that the makefile is executed and the source code is compiled. The resulting file is executed and the generated plots are moved into the subdirectory '/Output/eps'.

- **run.sh -m**
  Additionally to the production of .eps files they are converted into .jpg files using ImageMagick 'convert' and finally compressed into a .mpg movie using 'ffmpeg' showing the development of the embryo. For this procedure 'ghostscript' and 'ffmpeg' (open source) have to be installed on the executing system.

- **run.sh -q**
  This option uses a different program to convert the .eps files to .jpg files resulting in a much higher image quality and therefore a high quality movie. Additionally to 'ffmpeg' the perl program 'eps2png' by Vromans, J. [6] has to be installed on the executing system.

After all tasks have been completed, 'run.sh' uses the 'make clean' command to remove the object files.

```bash
#!/bin/bash

chmod +x configure.sh
./configure.sh
echo "compiling.."
make >log.txt
./main.exe
# cleaning up
rm -r Output
mkdir -p Output
mkdir -p Output/eps
mv *.eps Output/eps
case $1 in
  -m)
echo "creating jpg's.."
  mkdir -p Output/jpg
  TEMP=$(sed -n 17p settings)
  for (( i = 0 ; i <= ($TEMP - 1); i++ ))
    do
      convert -compress none -quality 100 -density 100 -resize 400% Output/eps/$i.eps Output/jpg/$i.jpg
done
  ffmpeg -f image2 -i Output/jpg/%d.jpg Output/gradient.mpg
  ;;
```
The settings file contains the following values:

- Gridsize
- Number of substances
- Diffusion type
- Decrease per time
- Decrease per space
- Probability of linkage
- Maximum number of linkage
- Number of timesteps
- Plot grid (yes/no)
- Index of substance used for gradient plot

These settings can be extended easily by just adding more values to the settings file and changing the `configure.sh` script accordingly.
3.5 A wider View

After my bachelor work on the simulation environment, the program is intended to be embedded into an evolutionary engine, which calculates several generations, themselves consisting of several (hundred) individual embryos. Complying to this need and also in an attempt to simplify the usage of the program I developed an overhead class called 'Generation' which is intended to take care of the automated calculation of several individuals. Another need was to automate the 'read in' and 'write out' processes of genomes, which form the base of every individual embryo. This need was met in the implementation of class 'Generation' as well. It consists of the following methods:

- **Generation ( )**
  With every construction of a 'Generation' object, a static integer called 'generation number' is increased by one, giving each generation an identification number and providing the total number of generations in use by the program.

- **void readGenome ( int Generation, int Individual )**
  This method reads the genome of the individual with the number 'Individual' from a .csv file located in the subfolder '/Genomes' which is named according to the current generation number. Although .csv files are not very easy to read, this method was chosen with the bigger view of automation via the evolutionary process in mind.
  Every line of the genome contained by the .csv file has the following format:
  
  4,0,0,0,240,255,112,255,0,20,30,4,120,4,90,120,80,180

  The first number stands for the individual followed by the individual numbers of its ancestors. Ancestry and evolution itself are not yet implemented, so every individual has the ancestors 0 and 0. After that, a variable number of genes may form the rest of the line, each one holding five parameters - the first four equivalent to the ones in NetLogo genomes, the fifth representing the morphogen release rate.
  Then the method reads each line into a string variable and checks if the individual number matches the one delivered as a parameter to the method. Therefore one individual number can only exist once within a generation genome .csv file. After finding the right line, the string variable is processed by cutting out the ancestor-numbers first and then each gene, until there is nothing left. This method of processing also requires exact quintuples of genes and exactly three numbers in the beginning of the line, each number comma separated from its consequential. The read out genes are added to 'currentGenome' which is a private member variable of type 'Genome'. Every time 'readGenome' is called, it renews this variable, so the old individual is overwritten.

- **void writeGenome ( )**
  This method works vice versa to 'readGenome' and stores the single numbers contained by 'currentGenome' in character arrays to process them into the generation .csv file.
Figure 3.2: C++ class hierarchy after extension of the implementation. Class 'Generation' acts as an overhead routine and is able to calculate a large number of individuals using 'NeuralGenesis'.

3.6 Modifications to Diffusion

In the basic C++ implementation of the model, diffusion was considered to take place linearly, for every substance with the same constant rate. In an attempt to make the simulation more complex, two different types of diffusion were added:

- linear diffusion with different decrease rates for each substance
• linear diffusion with different decrease rates for each substance and a cut
to zero when the level reaches or is below a certain value

// value where diffusion 2 cuts the level to 0
int cut = 50;

for ( int i = 0; i < Config::NUMSUBSTANCES; ++i )
{
    // Applying the decrease over time or setting it to production level
    int oldLevel = substances[i].level;
    int newLevel = 0;
    int decrease_over_time = 0;
    int decrease_over_space = 0;

    if ( Config::DIFFUSION == 0 ) // original diffusion
    {
        decrease_over_time = 1;
        decrease_over_space = 1;
    }
    else if ( Config::DIFFUSION == 1 ) // substance dependent slope
    {
        decrease_over_time = 10 + (5 * i);
        decrease_over_space = 10 + (5 * i);
    }
    else if ( Config::DIFFUSION == 2 ) // steeper diffusion with cut to 0
    {
        if ( oldLevel >= cut )
        {
            decrease_over_time = 10 + (5 * i);
            decrease_over_space = 10 + (5 * i);
        }
        else
        {
            decrease_over_time = 10 + (5 * i);
            decrease_over_space = oldLevel;
        }
    }
}

*Implementation of different diffusion processes in class 'Patch'.*
Working with the Program

4.1 Establishment of autoregulated Growth

With a working graphical output and the possibility to parse whole genomes from files, the task that was left to accomplish was to establish autoregulated growth of an embryo. It showed, that the genomes of the NetLogo implementation of the model and the ones from the C++ implementation were not compatible because of the additional parameter and slight differences in implementation. Apart from that genomes of the C++ version are haploid and not yet diploid. Therefore new regulatory genes had to be estimated from scratch.

Table 4.1 shows the adapted list of cell behaviour which is used to encode the genes. Internal parameter modifications are not implemented yet, as they are mainly thought of being placeholders for future extensions of the implementation.

The concept of autoregulation implies that during the stage of proliferation cells emit a morphogen which causes the proliferation process to stop. The result is a stable, approximately round embryonal shape which is used as a base to develop further autoregulated forms and shapes. Finding the correct gene parameters for the embryo to produce autoregulated growth was a time taking trial and error process which finally resulted in the genome represented in table 4.2 which consists of four genes.

<table>
<thead>
<tr>
<th>List of Products</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-15: emit substance 0-15</td>
</tr>
<tr>
<td>16-47: sensitivity modifications</td>
</tr>
<tr>
<td>48-79: internal parameter modifications</td>
</tr>
<tr>
<td>80-95: kill cells</td>
</tr>
<tr>
<td>96-111: duplicate vertical</td>
</tr>
<tr>
<td>112-127: duplicate nondirectional</td>
</tr>
<tr>
<td>128-143: duplicate horizontal</td>
</tr>
<tr>
<td>144-159: link cell</td>
</tr>
</tbody>
</table>

Table 4.1: Encoding table for the product parameter in genes
Table 4.2: The first Genome producing autoregulated growth in the C++ implementation

<table>
<thead>
<tr>
<th>Gene</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gene 1</td>
<td>0,220,255,112,255</td>
</tr>
<tr>
<td>Gene 2</td>
<td>0,200,210,3,255</td>
</tr>
<tr>
<td>Gene 3</td>
<td>3,1,255,3,255</td>
</tr>
<tr>
<td>Gene 4</td>
<td>3,1,255,16,0</td>
</tr>
</tbody>
</table>

Figure 4.1: Autoregulated growth including the gradient for the stop-morphogen. Proliferation process has stopped and the embryo has come to a stable form. Circles indicate cells, squares indicate patches, coloration from green to white corresponds to the morphogen level.
According to gene 1 the cells proliferate nondirectional if the level of morphogen 0 - which is also called the primary gradient because it is emitted automatically in the middle of the grid - lies between 220 and 255. The second gene then triggers the release of morphogen 3 if the primary gradient lies between 0 and 200, which should be a bit off the center point. So when this gene comes into effect the cells should already have formed a small, approximately round shape. Once released, gene 3 amplifies the emittance of morphogen 3 which - according to gene 4 then shifts the cells sensitivities for substance 0 thus containing the growth of the embryo until it comes finally to a hold, when the sensitivity in all cells for morphogen 0 has reached 0. The resulting shape including the gradient for the stop-morphogen 3 can be seen in figure 4.1.

On the basis of this first autoregulatory genome another more complex lateral growth genome was designed, increasing the number of genes to a total of 10. The resulting shape can be seen in figure 4.2. During its development the embryo first induces lateral growth, then another morphogen is emitted which enables horizontal growth at the lateral ends of the embryo. These growth processes are detained in the same autoregulatory way as the first genome.

Figure 4.2: Autoregulated lateral and horizontal growth at the end points including the gradient for the stop-morphogen for the horizontal growth. The embryo has developed a lateral shape first then a morphogen is emitted that induces horizontal growth at its lateral ends. Circles indicate cells, squares indicate patches, coloration from green to white corresponds to the morphogen level.
4.2 Time Measurements

Using the C++ standard-library’s method ‘gettimeofday’, which returns the current time in both seconds and microseconds, I measured the time taken by one ’NeuralGenesis’ step and by one cell. It was not surprising that the time taken by a single NeuralGenesis step depends mainly on the number of cells currently being in existence within the embryo. So the time of one NeuralGenesis step differs between $1 \times 10^{-2}$ seconds and $2 \times 10^{-2}$ seconds while an update step for one single cell takes about $1.5 \times 10^{-6}$ seconds per cell on a Intel® Core™2 Duo CPU E8500 clocked at 3.16GHz.
Summary

Initiating my work with only the core of the program, I have extended this piece of software with very usable features such as a graphical output, a global settings file, automated in- and output of genomes and automated run of the program including optional movie production. One disadvantage of the current implementation is that it needs to be recompiled every time the global setting in the settings file are changed. The resulting time loss however can be neglected as the future application of the program will consist mostly in large scale batch operations. As genomes between the NetLogo and C++ implementation turned out to be incompatible, also new genomes have been created, leading to embryos that develop autoregulated growth in the C++ version of the program.
Appendix

6.1 References

1. THENIUS, R. Novel concept of modelling embryology for structuring an artificial neural network, MATHMOD Vienna (2009)


4. Diagrams curtesy of Ronald Thenius

5. The epslplot library used was programed by Olli Niemitalo in 2001 and is freely available under http://www.student.oulu.fi/ oniemita/DSP/INDEX.HTM

6. The perl program eps2png written by Johan Vromans is freely available under http://www.vromans.org/johan/software/sw_eps2png.html
List of Tables

4.1 Encoding table for the product parameter in genes . . . . . . . . 16
4.2 The first Genome producing autoregulated growth in the C++
implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
List of Figures

2.1 Model overview indicating feedbacks between various effects in the model [4]. .................................................. 4
2.2 Schematic of the evolutionary optimization process and extraction of the ANN [4]. ............................................... 4
2.3 C++ class hierarchy before changes to the implementation. 'NeuralGenesis' acts as an overhead routine for 'Patch' and 'Cell' using the information delivered by 'Genome'. ............................... 6

3.1 Plot of a single timestep of a simulation using the C++ implementation including the gradient of substance 0. Cells are represented by circles, patches are represented by a grid lying underneath the cells. The color gradient indicates the concentration of morphogen 0. A lush green corresponds to a high concentration and white to a very low concentration of the morphogen. .... 9
3.2 C++ class hierarchy after extension of the implementation. Class 'Generation' acts as an overhead routine for a large number of individuals calculated using 'NeuralGenesis' .......................... 14

4.1 Autoregulated growth including the gradient for the stop-morphogen. Proliferation process has stopped and the embryo has come to a stable form. Circles indicate cells, squares indicate patches, coloration from green to white corresponds to the morphogen level. 17
4.2 Autoregulated lateral and horizontal growth at the end points including the gradient for the stop-morphogen for the horizontal growth. The embryo has developed a lateral shape first then a morphogen is emitted that induces horizontal growth at its lateral ends. Circles indicate cells, squares indicate patches, coloration from green to white corresponds to the morphogen level. .... 18