EasyLogic

CT Projekt

Herr Lempp

Philipp Faller  
Oliver Suchan

Inhalt

[2 Informationen zum Projekt 3](#_Toc390975595)

[2.1 Um was geht es bei Wireworld? 3](#_Toc390975596)

[2.2 Um was geht es bei unserem Projekt? 5](#_Toc390975597)

[3 Informationen zum Quellcode 6](#_Toc390975598)

[3.1 Aufgabenteilung 6](#_Toc390975599)

[3.2 Schichten-Architektur 6](#_Toc390975600)

[3.3 Information zur Ungarischen Notation 7](#_Toc390975601)

[3.4 Informationen zur Globals-Klasse 8](#_Toc390975602)

[4 Information zur Datenbank 10](#_Toc390975603)

[5 Informationen zur Dokumentation 12](#_Toc390975604)

[6 Software von Drittanbietern 13](#_Toc390975605)

[6.1 GitHub 13](#_Toc390975606)

[6.2 Qt 5.2 15](#_Toc390975607)

[7 Creatio 16](#_Toc390975608)

[8 Entwicklung des Quellcodes 17](#_Toc390975609)

[9 Prestigio – der letzte Akt 22](#_Toc390975610)

[10 Konflikte 26](#_Toc390975611)

[11 Liste der Logischen Bausteine für WireWorld 27](#_Toc390975612)

[12 Quellen 28](#_Toc390975613)

[12.1 GitHub 28](#_Toc390975614)

[12.2 Qt 28](#_Toc390975615)

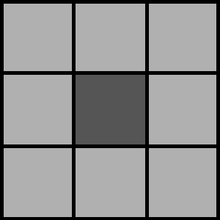
[12.3 Ungarische Notation 28](#_Toc390975616)

# Informationen zum Projekt

Der Titel dieses Projekts lautet „*EasyLogic*“ und impliziert somit, dass hiermit einfach („Easy“) logische Gatter („Logic“) erzeugt und simuliert werden können. Dieses Projekt baut auf das sogenannte *Wireworld*-Konzept auf. Dazu aber später mehr. Getestet wurde diese Software lediglich auf *Windows 7* und auf *Ubuntu 12.04*. Nichtsdestotrotz sollte jene Software auch auf allen anderen Plattformen lauffähig sein, auf welchen *Qt* (dazu später mehr) verfügbar ist.

## Um was geht es bei Wireworld?

*Wireworld* ist ein sogenannter zellulärer, turingmächtiger Automat. Zellulärer Automat bedeutet, dass dynamische Modelle aus Zellen modelliert werden können und die Verbreitung, lediglich von den Zuständen der vorgegebenen Nachbarschaft abhängt. In dem Fall *Wireworld* herrscht eine sogenannte *Moore-* oder auch *8er-Nachbarschaft* vor. Das heißt, dass alle 8 Zellen um eine Zelle für die „Verbreitung“ benutzt werden. Dies sieht wie folgt aus:



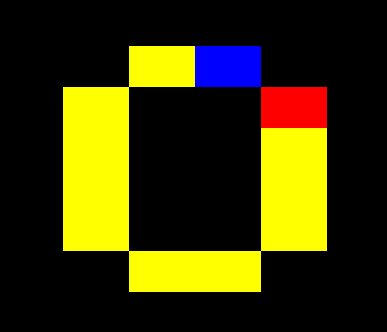
Wireworld besitzt 4 verschiedene Zellen Zustände:

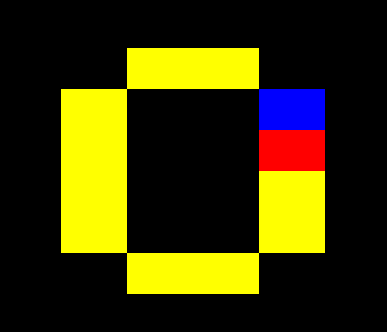
* Leer
* Leiter
* Elektronen-Kopf
* Elektronen-Ende

Die Verbreitungsregeln oben genannter Zellen sehen wie folgt aus:

* Leer bleibt Leer
* Aus einem Leiter wird ein Elektronen-Kopf, wenn genau eine oder zwei der sich in der Moore-Nachbarschaft befindlichen Zellen Elektronenköpfe sind
* Aus einem Elektronen-Kopf wird ein Elektronen-Ende
* Aus einem Elektronen-Ende wird ein Leiter

Das bedeutet, dass sich die Status der Zellen in jedem Schritt auf diese Weise ändern. Aus diesem Grunde gilt dieser *zelluläre Automat* auch als turingmächtig, das heißt, man kann mit jenem sogar vollständige Computer erzeugen. Hier ein Beispiel, wie so eine Verbreitung aussehen kann:

Simulationsschritt 1: Simulationsschritt 2:



Dieses Beispiel ist der verständnishalber sehr simpel gehalten. Hierbei bewegen sich die blaue und rote Zelle – Elektronen-Pärchen - permanent im Kreis.

## Um was geht es bei unserem Projekt?

Bei unserem Projekt geht es darum, das oben genannte *Wireworld*-Konzept so gut wie möglich in die Tat umzusetzen. Wir persönlich sind mit unserer erbrachten Leistung sehr zufrieden, da es einerseits performance-technisch eine hohe Leistung erreicht und des Weiteren noch – unserer Meinung nach – sehr leicht zu bedienen ist. Wir verheimlichen keineswegs, dass es Möglichkeiten gäbe noch etwaige Performance-Schübe durch Code-Änderungen herauszuholen, aber die gibt es doch immer. Nichtsdestotrotz sind wir – wie bereits gesagt – sehr zufrieden und das ist ja das was zählt.

Was unterscheidet unser *EasyLogic* von anderen *Wireworld*-Simulations-Softwares?

Nun, die Simulations-Softwares für *Wireworld* die wir bis jetzt entdeckt haben, waren hauptsächlich in Java geschrieben und nur als Applets verfügbar. Des Weiteren war es bei diesen Applikationen nur möglich eine vorgegebene Map zu bearbeiten und diese zu simulieren. Bei uns hingegen ist es möglich, fast alle Einstellungen zu personalisieren. Darunter folgende Optionen, die uns fehlten:

* Beliebige Map-Größe ( (*INT\_MAX \* INT\_MAX*) )
* Benutzerdefinierte Farben für die Zell-Status
* Beliebig viele eigene Bausteine/ Elemente speichern und laden
* Generationen zurückrechnen/ -simulieren
* Rein- bzw. rauszoomen

Außerdem wurde unser Projekt mit C++11 und Qt 5.2 programmiert und ist somit als native Anwendung auf einem Großteil der heutzutage verwendeten Betriebssystemen verwendbar.

# Informationen zum Quellcode

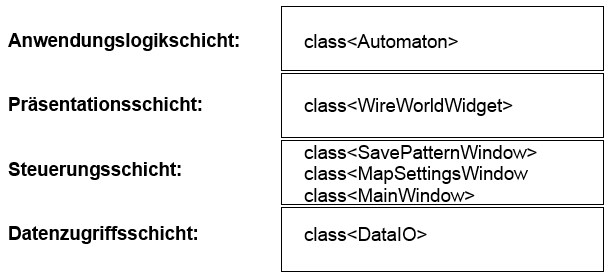
Unser Projekt wurde vollständig in C++11 programmiert. Dies taten wir einerseits der Einfachheit – bezüglich Threading – halber und da es uns die Möglichkeit gibt, schöneren, lesbareren Quellcode zu gestalten. Wie zum Beispiel for-each-Schleifen die aus Java bekannt sein sollten.  
Probleme beim Verständnis der Programmiersprache hatte weder Oliver noch ich, da wir beide bereits uns mal entschlossen C++ zu lernen. Ich lernte es jedoch erst letztes Schuljahr.

## Aufgabenteilung

Den logischen Part – sprich den zellulären Automaten an sich, das Herz – haben Philipp und ich komplett zusammen programmiert. Doch dann ging es auf die Dateiverwaltung und grafischen Oberflächen unseres Projektes zu. Da Philipp gerne die Dateiverwaltung zum Leben erwecken wollte, da er schon beim letzten Projekt viel mit Grafik zu arbeiten hatte und ich bereits beim letzten Projekt viel mit Dateiverwaltung gearbeitet hatte, einigten wir uns darauf, dass Philipp die komplette Dateiverwaltung übernimmt und ich mich um das Aussehen kümmere. Nichtsdestotrotz trennten sich unsere Wege nicht vollständig und wir halfen einander, falls wir selbst nicht weiterwussten (Siehe dazu *Konflikte* in den folgenden Kapitel).

## Schichten-Architektur

Eine weitere zwingendnotwendige Eigenschaft unseres Projekts ist die Einteilung der Software in verschiedene Schichten, um der Software eine stilvolle Gestalt zu verleihen. In unserer Software sind folgende Schichten aufzufinden:



Ausführliche Beschreibungen zu den einzelnen Klassen folgen später.

## Information zur Ungarischen Notation

Da Oliver mich anfänglich überzeugen konnte, dass die Ungarische Notation ein positiver Aspekt sei, entschlossen wir uns, sie auch in diesem Projekt zu applizieren. Doch im Laufe des Projektes konnte ich Oliver überzeugen, dass die Ungarische Notation sich sehr stark – negativ – auf die Lesbarkeit des Quellcodes auswirkt, da man teilweise Kilometerlange Bezeichner entziffern muss, um eine Aufgabe der Variable mit jenem Bezeichner erschließen zu können. Trotz dieses Defizits entschlossen wir uns die Ungarische Notation beizubehalten und auch ihr einen Platz in der Dokumentation zu widmen.

Abgesehen von den normalgültigen Bezeichner-Strukturen gilt eine weitere, sehr simple Struktur:

Präfixe und Datentypen werden aus den ersten 2 Zeichen des Namens verwendet. Wenn 3 Zeichen verwendet werden, wird einfach ein beliebiger Konsonant, der sich in der Zeichenkette des Namens befindet, angehängt, sofern dieser eine eindeutige Identifikation ermöglicht.

## Informationen zur Globals-Klasse

Die Globals-Datei(en) ist eine sich in der Singleton-Designstruktur befindlichen Klasse. Ihre Aufgabe ist, für alle Klassen „globale“ Informationen bereitzustellen. Diese Klasse bekommt eine extra Widmung in dieser Dokumentation, da es ein sehr kontrovers diskutiertes Thema ist, ob man in C++ sogenannte Globals-Klassen/-Dateien verwenden sollte. Sie gilt als sehr riskant, da, wenn man sich mal ein Projekt mit mehreren Tausend oder sogar Millionen Zeilen Code vorstellt, die alle auf diese globalen Variablen basieren und diese sogar bearbeiten, kann es schwer werden, rauszufinden, wer wann auf diese Variablen zugreift. Des Weiteren erhöht die Verwendung von globalen Variablen die Komplexität des Quellcodes. Da das erste Problem bei uns jedoch nicht auftreten kann, entschlossen wir uns, unser Globals-Konzept beizubehalten. Üblicherweise werden solche globalen Variablen ganz anders angelegt:

globals.h:

#ifndef GLOBALS\_H

#define GLOBALS\_H

extern int g\_iGlobalInteger;

#endif // GLOBALS\_H

globals.cpp:

#include "globals.h"

int g\_iGlobalInteger = 0;

main.cpp:

#include <iostream>

#include "globals.h"

int main(int argc, char \*\*argv)

{

::g\_iGlobalInteger += 1;

std::cout << ::g\_iGlobalInteger << std::endl;

return 0;

}

Das resultierende Ergebnis wird ohne Zweifel „1“ lauten.

Durch den ::-Operator wird gesagt, dass die globale Variable verwendet werden soll, falls eine gleichnamige lokale Variable existieren sollte.  
Hier ein Beispiel dafür:

main.cpp:

#include <iostream>

#include "globals.h"

int main(int argc, char \*\*argv)

{

int g\_iGlobalInteger = 4;

g\_iGlobalInteger += 2;

::g\_iGlobalInteger += 1;

std::cout << ::g\_iGlobalInteger << std::endl;

std::cout << g\_iGlobalInteger << std::endl;

return 0;

}

Die Ausgaben lauten:

* 1 (globale *g\_iGlobalInteger* Variable)
* 6 (lokale *g\_iGlobalInteger* Variable)

Natürlich widerspricht das obige Beispiel der Ungarischen Notation, jedoch geht es nur um die Verdeutlichung der Unterscheidung zwischen lokaler und globaler Variable.

Natürlich hätten wir diese Struktur auch in unserem Projekt einbauen können, aber da wir die Unübersichtlichkeit dadurch förmlich riechen konnten, entschlossen wir uns, das Singleton-Designpattern zu verwenden.

Im Gegensatz dazu, werden globale Variablen bei uns folgendermaßen benutzt:

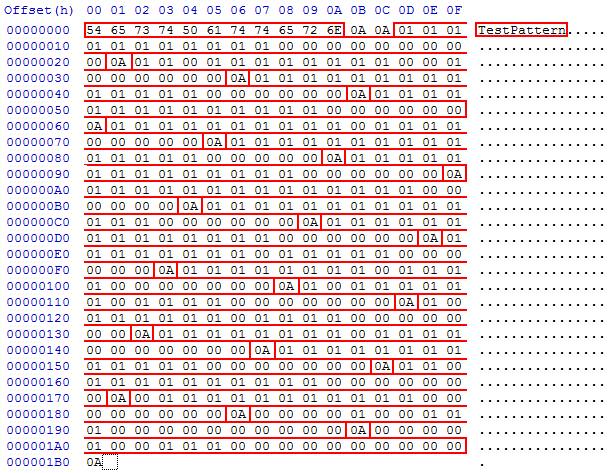
(rein exemplarisch)

Globals::getInstance().m\_rqcColors[p\_iIndex] = m\_pqcdColorPicker->currentColor();

Es ist viel mehr schreibaufwand, keine Frage. Jedoch ist der Sinn der Variable so viel einfacher nachzuvollziehen.

# Information zur Datenbank

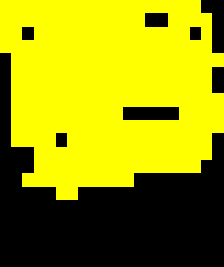
TestPattern.pattern – Hex-Editor-Ansicht:



0x0A bezeichnet einen Zeilenumbruch, das heißt, die *TestPattern.pattern*-Datei hat 23 Zeilen. Dabei stellt die 1. Zeile den Namen des Elements dar und wird von 2 Zeilenumbrüchen begleitet. Danach werden die Zustände durchnummeriert und je nach Element als Hex-Zahl gespeichert. Eine Zeile stellt auch tatsächlich eine Reihe in der Map dar. Das heißt die Zellen-Zustände sind jeweils ein Byte einer Zeile/Reihe.

Sprich in diesem Beispiel ist der Baustein 20 x 20 groß, da eine Zeile aus 20 Byte besteht und es 20 ausgefüllte Zeilen gibt.

Der Baustein sieht geladen wie folgt aus:



Dieser Baustein wurde nur zu Testzwecken erzeugt und besitzt keine bestimmte Aufgabe.

Auf diese Art und Weise sind problemlos komplexe und große Bausteine speicherbar und besitzen dennoch eine relativ kleine Dateigröße.  
Lässt man den Baustein-Namen Außen vor ist die maximale Dateigröße die erreicht werden kann:

(mit den Beiden Zeilenumbrüchen die nach dem Baustein-Namen folgen; sollte vorerst reichen)

# Informationen zur Dokumentation

C++ bietet den Vorteil, dass Deklaration und Definition komplett voneinander getrennt sind. Sprich: in verschiedene Dateien ausgelagert sind. Die Header-Datei, in der sich die Deklarationen befinden, und die Quell-Datei, in der sich die Definitionen befinden. Nichtsdestotrotz arbeiten diese beiden doch so unterschiedlichen Dateien sehr eng zusammen. Darum ist es meistens nicht notwendig beide Dateien zu dokumentieren sondern nur eine dieser Dateien. Die Header-Datei. Jedoch muss hier beachtet werden, dass die Dokumentation jener sich in der Header-Datei befindlichen Deklarationen so gut dokumentiert sein muss, dass es tatsächlich nicht notwendig ist, die Quell-Datei zu dokumentieren. In C++ gibt es eine bestimmte Struktur um Quellcode zu dokumentieren, welche der aus Java bekannten Methode sehr ähnelt.

Hier ein Beispiel für solch eine Dokumentation:

/\*\*

\* @brief Überladener Left-Shift-Operator,<BR>

\* um benutzerdefinierte Elemente zu speichern

\* @param p\_cpPattern Name des Elements<BR>

\* (ohne Dateiendung)

\* @return DataIO-Referenz

\* @throws FileAlreadyExistsException

\*/

DataIO &operator<<(const CellPattern &p\_cpPattern);

In diesem kommentar-ähnlichen, blau geschriebenen Text, sind Wörter gefolgt durch @-Zeichen erkennbar, auch Tags genannt, die die Dokumentation immens verbessern, da so viele Informationen hinzugefügt werden können. Außerdem ist es möglich HTML-Code in dieser Art von Dokumentation zu verwenden.

Wenn man nun seinen gesamten Quellcode so dokumentiert hat, gibt es Möglichkeiten – in Form von Softwares – diese Dokumentation nochmals übersichtlicher zu machen. Für Java gibt es das sogenannte Javadoc, welches aus dieser Art von „Kommentaren“ eine Dokumentation im Webseiten-Format generiert. Exakt dasselbe gibt es auch für C++, nur, dass es sich Doxygen nennt und mehr Möglichkeiten hat, die Generierung zu eigenen Zwecken anzupassen.

Genau diese Software verwendeten wir zum Erzeugen unserer Webseiten-Dokumentation.

Eine Liste und ausführliche Dokumentation zu allen in Doxygen verfügbaren Tags und deren Syntax sind auf folgender Webseite zu finden:

<http://www.stack.nl/~dimitri/doxygen/manual/commands.html>

# Software von Drittanbietern

## GitHub

GitHub ist eine Versionsverwaltungs-Software, das heißt, speziell zur Verwaltung von Software-Projekten gestaltet. Man stelle sich einmal folgende Situation vor. 2 Schüler – exemplarisch nennen wir sie einfach einmal Oliver und Philipp – besuchen ein Technisches Gymnasium mit dem Profilfach Informationstechnik. Aktuell befinden sie sich in Kursstufe 2 und haben ein freiwilliges Fach namens Computertechnik-Labor gewählt. In diesem Fach – abgekürzt CT – haben diese beiden sehr tüchtigen Schüler sich in Gruppen zu finden und in diesen Gruppen, Projekte zu programmieren. Doch halt! Projekte in Gruppe programmieren? Das kann doch oft zu Problemen führen? Einer ändert aus Versehen etwas im Quellcode und es gibt kein Backup… Und wie stellt man seinen Quellcode einfach seinem Partner zur Verfügung?

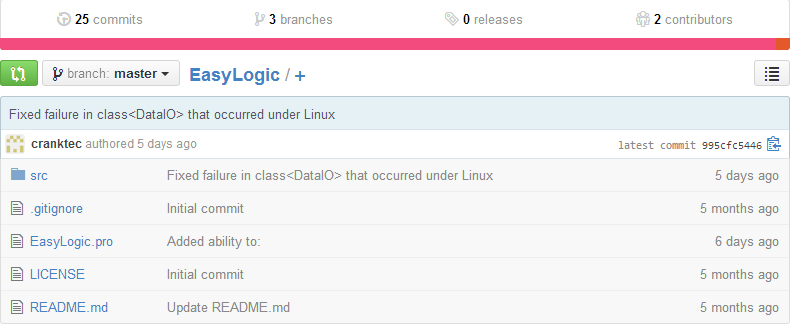
Ach, das ist doch kein Problem. Die Antwort lautet Dropbox. Genau falsch. Dropbox hat nicht Möglichkeit von Versionsverwaltung, weswegen es von vornherein ausgeschlossen wird (werden sollte).

GitHub ist ein Präventivschlag für genau so etwas. Man arbeitet zusammen an einem Projekt und möchte nun seinen Quellcode seinem Gruppen-Partner zur Verfügung stellen. GitHub hostet sogenannte Repositorien. Diese Repositorien sind im Prinzip einfach das Projekt. Bei diesen Repositorien kann man seinen Quellcode commiten zu Deutsch: „übergeben“ oder „einliefern“. Diese Commits sagen, welche Dateien an welcher Stelle geändert wurden. Man kann jederzeit zu einem vorherigen Commit zurückkehren und verliert somit niemals seinen Quellcode.

Unser GitHub-Respository ist unter folgender URL kostenlos und Quelloffen verfügbar:

<https://github.com/cranktec/EasyLogic>

Und sieht in etwa so aus:



Hier sieht man sofort, dass unser Repository 25 Commits besitzt. Man kann jede einzelne Änderung, die am Quellcode vorgenommen wurde, durch klicken der jeweiligen Commits einsehen.

Des Weiteren ist erkennbar, dass es 3 Zweige („Branches“) gibt. Diese Zweige sind:

* master
* Philipp
* Oliver

Bei dem Zweig *master* haben Philipp und ich anfangs noch zusammen gearbeitet, doch später, als es zur Trennung kam, war die klügste Entscheidung neue Zweige zu erstellen, damit wir uns nicht behindern. Am Schluss wurden dann die beiden Zweige *Philipp* und *Oliver* zu *master* hinzugefügt.

Um unseren Quellcode zu „pushen“, verwendeten wir das sich in der Qt IDE intern befindliche Plugin für GitHub.

Das ich bereits beim letztens CT-Projekt viel Erfahrung im Bezug auf GitHub mir aneignen konnte, kam es bei diesem Projekt zu keinen ernstlichen Problemen, wie es beim letzten Mal leider der Fall war.

Klar können wir das volle Potential von GitHub vielleicht noch nicht ausschöpfen, aber was wir bis jetzt erreicht haben, half uns jedenfalls sehr weit auf unseren Wegen.

## Qt 5.2

Zur Programmierung unseres Projekts verwendeten wir die IDE QtCreator und benutzten des Weiteren das Framework Qt ([[kjuːt](http://de.wikipedia.org/wiki/Liste_der_IPA-Zeichen" \o "Liste der IPA-Zeichen)]), welches eine sogenannte cross-platform-ability besitzt. Das bedeutet so viel wie, dass Projekte die mit reinem Qt programmiert wurden, ebenfalls auf vielen weiteren Betriebssystemen verfügbar sind, ohne das spezielle Code-Veränderungen vorgenommen werden müssen. Wir entschlossen uns für das „Standard-Qt“, welches mit C++ und C arbeiten kann.

Kleine Anmerkung:

Qt ist für viele weitere Programmiersprachen verfügbar, darunter auch Java. Ebenso ist Qt für mobile Betriebssysteme im Betrieb:

* Android
* iOS
* UbuntuPhone (Apps werden zur Zeit ausschließlich mit Qt programmiert)

Qt bietet noch weitere Vorteile, wie zum Beispiel, dass es für Jedermann komplett kostenlos und Quelloffen zur Verfügung steht. Wenn man Qt jedoch für proprietäre Anwendungen nutzen möchte, muss man eine Lizenz erwerben oder bestimmte Lizenzbestimmungen einhalten. Dies ist bei uns jedoch nicht der Fall, da unser Projekt komplett opensource und frei zur Verfügung steht, weshalb wir uns auch nicht mit so viel Lizenzkram rumschlagen mussten.

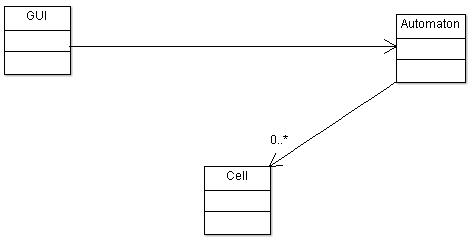
Qt war die einzige Fehleranfälligkeit unseres Projektes, dazu jedoch später mehr.

Dieses Mal verzichteten wir Voll und Ganz auf den integrierten WindowBuilder von Qt und erzeugten die gesamte GUI durch mühsam von Hand eingetippten Quellcode, mit kilometerlangen Bezeichnern.

# Creatio

Im Anschluss unser erstes Klassendiagramm zur Planung des EasyLogic-Projekts.

Vorwarnung: ja, wir dachten tatsächlich, dass das vollkommen ausreichen würde. Theoretisch tut es das auch, führt jedoch zu Unübersichtlichkeit und würde das implementieren unserer fortfolgenden Funktionen enorm erschweren.



An das Planen der Datenbanken hatten wir zu diesem Zeitpunkt noch nicht einmal annähernd gedacht. Wir hatten uns nur überlegt, wie wir am besten die Map beziehungsweise die Karte im RAM abspeichern sollten.

Dabei gab es viele Variationen, von Hashmaps bis hin zu Vektoren der Standard-Bibliothek, wofür wir uns letztendlich auch entschlossen.

Wir entschlossen uns einfach 2 ineinander geschachtelte Vektoren zur Erreichung der 2 Dimensionalität unserer Karten zu verwenden, was außerdem performancetechnisch einen Schub bewirkt haben sollte, da so die Koordinaten der Zellen implizit gespeichert werden, und nicht nochmal explizit gespeichert werden müssen.

Um den Quellcode zu vereinfachen, erzeugten wir schlichtweg einen *Typedef*:

typedef std::vector<std::vector<Cell> > CellArray;

Das bedeutet, das man statt std::vector<std::vector<Cell> > einfach nur CellArray schreiben kann, vorausgesetzt der *Typedef* ist an jener Stelle verfügbar.

# Entwicklung des Quellcodes

Legende:

* Klasse
* Singleton-Klasse
* Graphical-Userinterface

Als es dann an die tatsächliche Entwicklung des Projektes kam, merkten wir, dass wir für eine schöne Struktur, weitere Klassen benötigten, die verhindern sollten, dass wir Spaghetti-Code schreiben.

Die Klasse GUI, die oben zu sehen war, wurde in 4 Klassen aufgespalten:

* MapSettingsWindow
* MainWindow
* WireWorldWidget
* SavePatternWindow

(Detaillierte Beschreibungen der einzelnen Klassen folgen sofort)

Die beiden weiteren Klassen, *Cell* und *Automaton* blieben bestehen, genau wie ihre Assoziation, wurden jedoch durch die Erzeugung weiterer Klassen unterstützt.

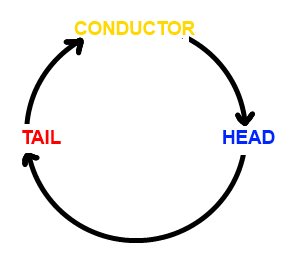
Zu allererst setzten wir uns an die Entwicklung der Klasse *Cell*, wobei diese lediglich den Zustand der aktuellen Zelle speichern und den folgenden Zustand bestimmen sollte. Dafür nutzten wir einen Enum, der aus dieser Reihenfolge bestand:

enum State {

EMPTY, CONDUCTOR, HEAD, TAIL

};

Auf diese Art und Weise der Anordnung werden die Zustände chronologisch durchnummeriert. Das bedeutet EMPTY hat intern den Wert 0 und TAIL besitzt intern den Wert 3. Auf diese Art und weise haben wir die Verbreitungsregel sehr genau darstellen können. Wir sagten einfach, dass sich ab dem Zustand CONDUCTOR die Zustände jeweils im Kreise bewegen:



Dadurch war es möglich den Zustand der Zelle sehr leicht zu ändern. Die Zustandsänderung von CONDUCTOR zu HEAD muss vorher jedoch überprüft werden, da sonst die Verbreitungsregeln zerstört werden.

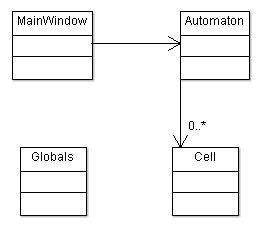
Da die Zustände auch für andere Klassen zu Verfügung gestellt werden musste, wurde die *Globals*-Klasse erzeugt, die genau dies übernahm (siehe vorherige Kapitel, für ausführliche Globals-Dokumentation). Zu diesem Zeitpunkt war die *Globals*-Klasse noch eine normale Klasse ohne Singleton-Designpattern.

Danach setzten wir uns an die Erzeugung des logischen Herzstücks. Den Zellulären Automaten. Dieser bewegt sich bei jedem neuen Erzeugen einer Generation iterativ durch die Karten-Vektoren und überprüft, wie der nächste Zustand der Zelle aussehen soll. Auch hier gab es keinen erkennbaren Performance-Verlust. Dieser Performance-Verlust entstand erst, nachdem wir das errechnete Zeichnen wollten.

Die Klasse GUI wurde durch die Klasse *MainWindow* ersetzt. Später diente eine Zwischenklasse namens *WireWorldWidget* zum Zeichnen der Map, welches wie ein Button auf Fenster platziert werden kann. Dazu jedoch später mehr.

Funktionen der Klassen:

* Automaton  
  Berechnet darauffolgende Generationen anhand der aktuellen Zellzustände der Map beziehungsweise Karte.
* Cell  
  Klasse zum Speichern der einzelnen Zellzustände und festlegen des kommenden Zustandes.
* Globals  
  Speichert Variablen die für mehre Klassen benötigt werden.
* MainWindow  
  Hatte die Aufgabe die gesamte Map zu visualisieren, ebenso die Simulation.

Hier ein mit ArgoUML erzeugtes Klassendiagramm, das den aktuellen Stand visualisieren soll:

Die Globals-Datei besitzt hier keine Assoziationen, da lediglich die Member verwendet werden und nicht die Klasse an sich.

Danach folgte die Implementierung der *WireWorldWidget*-Klasse. Dieses sollte die Hauptaufgabe des *MainWindow*s übernehmen. Das heißt, das Anklicken der einzelnen Zellen und damit verbundene Ändern der Zellzustände und natürlich das zeichnen. Vorerst wurden die Zustandsänderungen und jegliche anderen Aktionen durch Tastendrücke bewerkstelligt. Das heißt die Tasten 0-3 waren für die Zellzustände belegt. Leertaste startete die automatische Zell-Generierung und beendete sie auch wieder. Enter führte lediglich eine Generierung aus. Diese Tastenkombinationen sind auch noch in der Release-Version verfügbar, umso eventuell schneller arbeiten zu können. Als neues Feature wurde ein Zoom eingebaut. Die *WireWorldWidget*-Klasse, war jene Klasse die uns am meisten Probleme erzeugte. Dazu jedoch später mehr.

(Die Entwicklung der *WireWorldWidget*-Klasse beziehungsweise generell der grafischen Benutzeroberfläche, erfolgte synchron zur Implementation der Datenzugriffsschicht, da, wie bereits gesagt, Philipp und ich eigene Zweige aufzogen.)

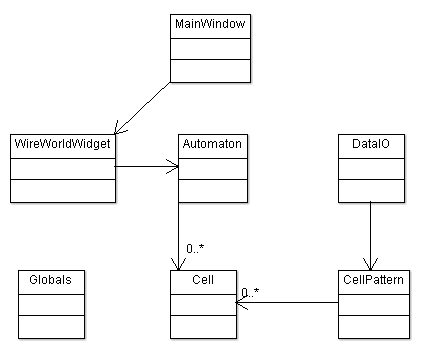
Danach bereicherten wir die *MainWindow*-Klasse um Buttons, damit man die Simulation nicht nur durch Tastenkombinationen steuern kann. Ebenfalls wurde eine Visualisierung des Zooms in der *MainWindow*-Klasse eingebaut.

Philipp implementierte erstmals Funktionen zum Schreiben von Patterns, von ihm benannt, und kurze Zeit darauf auch Funktionen zum Lesen dieser Patterns beziehungsweise Elementen. Dieses Patterns werden durch eine extra Klasse namens *CellPattern* identifiziert, in welcher sich ein *CellArray* und der Name des Patterns befinden.

Erweiterung zu *Funktionen der Klassen*:

* MainWindow  
  Beinhaltet eine *WireWorldWidget*-Komponente und Steuerungsmöglichkeiten zur Visualisierung wurden implementiert.
* WireWorldWidget  
  Stellt Simulationsprozess dar und ermöglicht durch Tastenkombinationen spezifische Änderungen vorzunehmen.
* DataIO  
  Ermöglicht es, Patterns zu speichern und zu laden.
* CellPattern  
  Speichert eine Map unter einem spezifischen Namen und wird von DataIO verwendet, um Patterns zu lesen und zu schreiben.

Hier ein daraus resultierendes Klassendiagramm:



Nun fehlten nur noch 2 Klassen, um den Entwicklungsprozess abzuschließen. Diese Waren 2 GUIs, welche die Aufgabe besaßen, Einstellungsmöglichkeiten der Map und Speichermöglichkeit von Patterns zu liefern:

Erweiterung zu *Funktionen der Klassen*:

* MapSettingsWindow  
  Gibt die Möglichkeit, spezielle Einstellungen an dem Aussehen der Applikation vorzunehmen. Darunter: sowohl benutzerdefinierte Größenfestlegung als auch Zustandsfarbenfestlegung.
* SavePatternWindow  
  Hiermit ist es möglich, Patterns unter einem beliebigen Namen zu speichern.
* MainWindow  
  Implementierung einer visuellen Schnittstelle für die beiden neu hinzugekommenen grafischen Benutzeroberflächen.

# Prestigio – der letzte Akt

Das Start-Fenster unserer finalen Version, ist das *MapSettingsWindow* und wird bei jedem Start ausgeführt.  
Jenes sieht folgendermaßen aus:

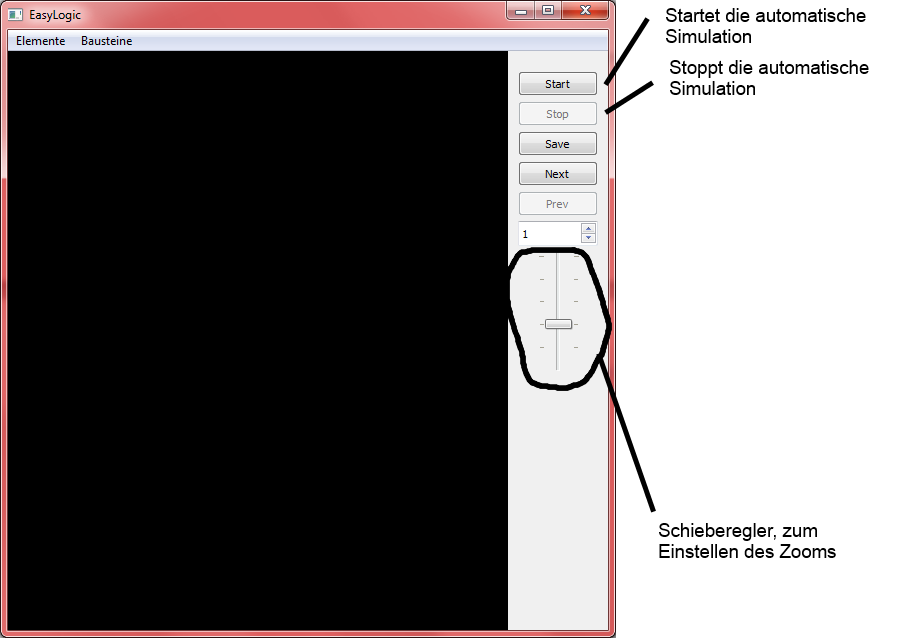
Man beachte hierbei, dass sich der Hintergrund dynamisch alle paar halben Sekunden ändert.

In den oberen beiden numerischen Feldern, ist es möglich die Map-Größe anzupassen, wobei zuerst die Breite und dann die Höhe folgt.  
(Minimal: 1, Maixmal: )

Wenn man nun die Farbe eines Zustandes ändern möchte, klickt man lediglich auf einen der zu sehenden Farben und es öffnet sich ein neues Fenster zum Auswählen einer Farbe.

Wenn man alle Einstellungen vorgenommen hat, kann man einfach auf „Anwenden“ klicken, um dieses logischerweise zu applizieren.

Sobald der „Anwenden“-Button im vorherigen Fenster gedrückt wurde, wird das Hauptfenster (*MainWindow*) geöffnet.   
Dies sieht wie folgt aus:  
(bei Standard-Einstellungen und einer 20 x 20 Map)



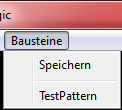
Das hier zu sehende schwarze Feld, ist eine leere Map. Durch Wählen eines Zustandes – im oberen Menü „Elemente“ – und Klicken einer bestimmten Stelle auf der Map, wird der Zelle dieser ausgewählte Zustand zugeschrieben. Der Zustand wird bis zum Wechseln des Zustandes beibehalten.

Durch den Button „Save“ an der rechten Seite, wird die aktuelle Map temporär zwischengespeichert, womit man später in der Lage ist, Generationen sozusagen wieder zu löschen und zu einer vorherigen Generation zurückzuspringen. Erst nach Drücken jenes Buttons, wird der „Prev“-Button aktiviert.

Der „Next“-Button geht eine bestimmte Anzahl an Generationen nach vorne.

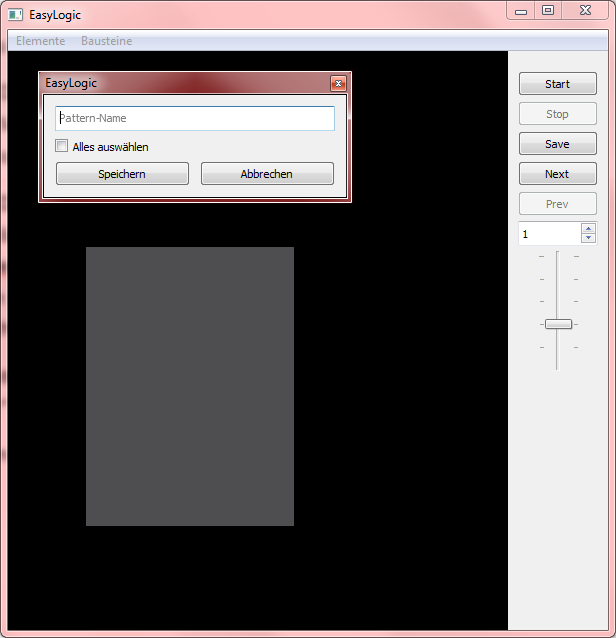
Für den „Next“- und „Prev“-Button gilt, dass der Wert des numerischen Feldes als Anzahl der Generationen, die vor- bzw. zurück-gerechnet werden sollen, verwendet wird.

Klickt man auf das Menü „Bausteine“ öffnet sich folgendes Kontextmenü:

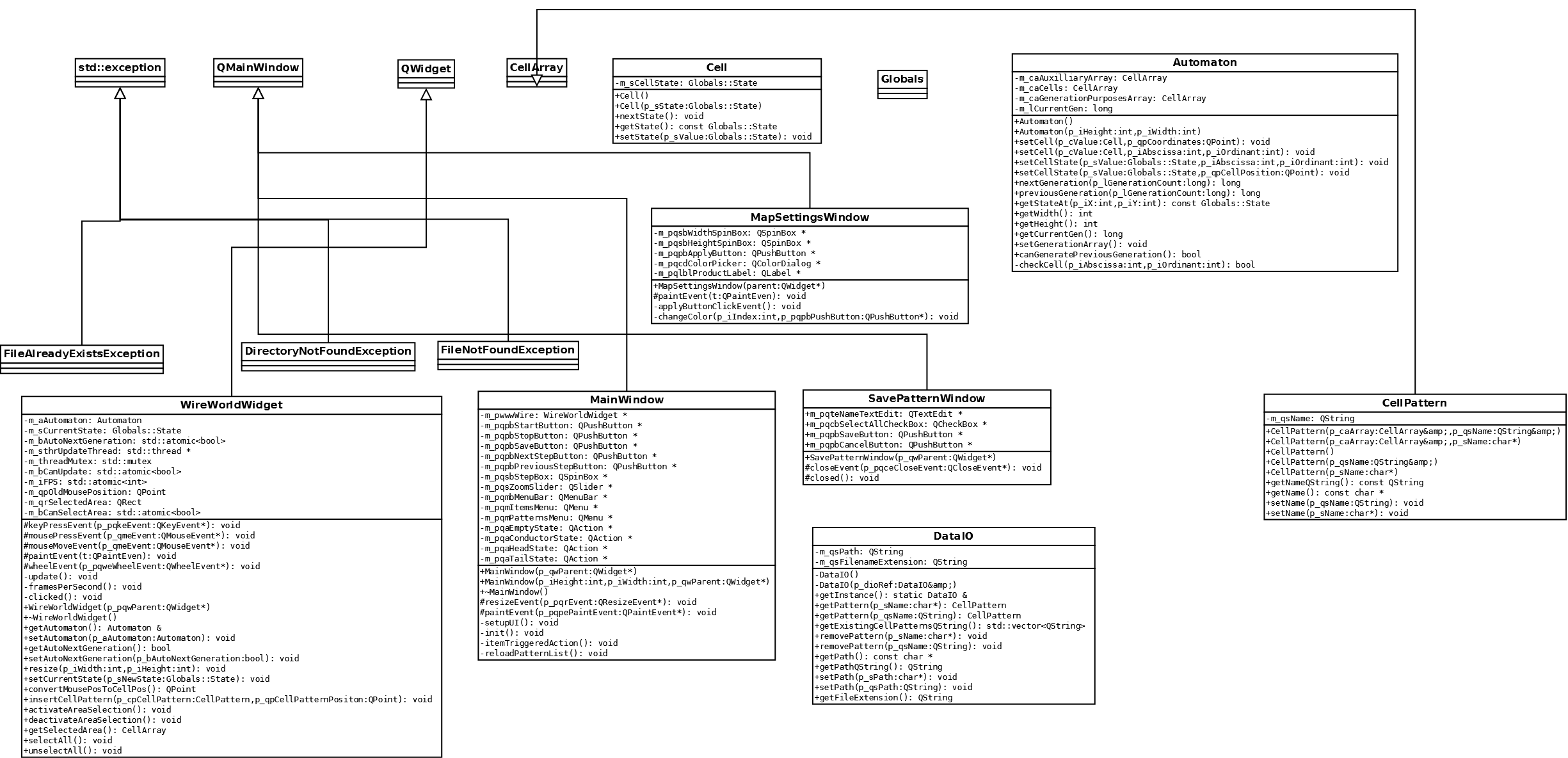
Durch das Klicken von Speichern, wird logischerweise die Möglichkeit eröffnet, Elemente auszuwählen und zu speichern.

Durch klicken eines Items, das sich unter „Speichern“ befindet, ist es möglich, die Patterns zu laden. Klickt man nach dem Klicken auf beispielsweise „TestPattern“ auf das schwarze Feld, wird an genau dieser Stelle, das Pattern platziert. Sollte das Pattern über die Map herausragen, wird es einfach abgeschnitten.

Es werden alle Dateien, die sich im aktuellen Verzeichnis + /Patterns befinden und auf „.pattern“ enden, angezeigt.

Das Speichern von Patterns sieht wie folgt aus:

Der grau markierte Bereich stellt hierbei den ausgewählten Bereich dar.   
Diese Fläche erreicht man, durch Klicken auf die Map, den Klick halten und das sich erzeugende Rechteckt auf beliebige Größe ändern.  
Durch „Alles auswählen“, kann man ganz einfach die gesamte Map auswählen.  
Jetzt fehlt nur noch der Name des Bausteins und schon können wir auf speichern klicken.



# Konflikte

Die Konflikte die sich bei uns ereigneten wurden durch eine einzige Klasse hervorgerufen. Die *WireWorldWidget*-Klasse.

Problem war hierbei, dass wir feststellten, dass durch das permanente Zeichnen der Map, viel Leistung verloren ging und die Frames per Second immer Kleiner wurden. Daraufhin beschlossen wir ganz einfach OpenGL zu verwenden, da wir dachten, dass wir dadurch in der Lage wären, die Zeit die für das Zeichnen benötigt wird, zu verkürzen. Tatsächlich schien es auch so zu funktionieren, nach dem wir es ausprobierten. Auf Ubuntu. Das war ein Fehler, denn sobald wir anfingen, dass Programm auf Windows zu testen, mussten wir feststellen, dass es sich immer beendet.

Grund dafür war, dass die OpenGL Implementierung von Qt in der Qt-Version die wir verwendeten sozusagen über Board geworfen wurde, und erst wieder in Qt 5.4 eingebaut wurde. Jedoch wurde diese OpenGL-Schnittstelle nicht aus dem Qt für Linux rausgeschmissen, weswegen wir fälschlicherweise die Fehler erst irgendwo anders gesucht haben. Bis wir dann schließlich feststellten, dass das Zeichnen das Problem darstellte, entschlossen wir uns, zum nativen Zeichnen von Qt überzugehen und einfach unseren Algorithmus zu optimieren, was uns dann auch die gewünschte Leistung erbrachte.

Ansonsten ereigneten sich keinerlei weiteren Konflikte.

# Liste der Logischen Bausteine für WireWorld

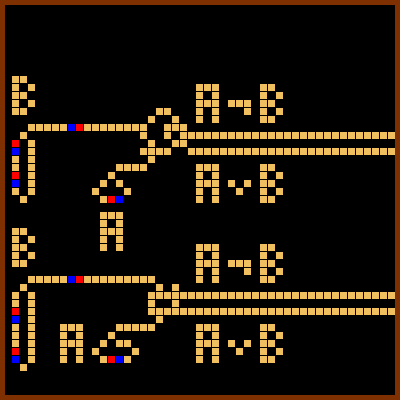
Es gibt eine Webseite die alle Logischen Bausteine für Wireworld nachgebaut hat. Ursprünglich war unser Plan, diese Webseite zu parsen und dort aufgelisteten Bausteine – im Form eines Bildes – zu parsen und eine Ausgabe zu liefern, deren Struktur unseren Datenbanken entspricht.

Da wir uns jedoch erstmal auf das Projekt an sich konzentrieren wollten, geriert diese Idee in Vergessenheit und wurde darum leider nicht in die Tat umgesetzt.

Nichtsdestotrotz nennen wir hier diese Webseite für alle Interessenten dieser Logischen Bausteine in Wireworld.

Jene Webseite ist unter folgendem Link erreichbar – zwar auf Englisch, aber dennoch eine gute Hilfe:

<http://karlscherer.com/Wireworld.html>

Außerdem entschlossen wir uns auch im Nachhinein diese Idee ruhen zu laden, da wie hier erkenntlich ist, werden nicht nur die logischen Schaltkreise erbaut, sondern auch noch „Kommentare“ oder mehrere Versionen hinzugefügt und da es für ein Programm schwer ist, das zu differenzieren, ließen wir es sicherheitshalber bleiben, um auch keine falschen Versprechen zu machen.

# Quellen

## GitHub

<https://github.com>

## Qt

<http://qt-project.org>

## Ungarische Notation

<http://de.wikipedia.org/wiki/Ungarische_Notation>