Projektpraktikum Informatik – Bericht

3D-Visualisierung einer wafer-scale neuromorphen Hardware und des darauf abgebildeten biologischen neuronalen Netzwerkes

Radoslav Rusanov
E-Mail: rrusanov@physi.uni-heidelberg.de

Betreuer: Daniel Bruederle

Inhalt

1 Einführung.....3

2 Das Graphenmodell.....3

3 Der Aufbau der Visualisierungssoftware.....4

4 OpenGL.....4

5 *glControl*.....5

6 *GraphVisu*.....5

7 Lokale Container.....6

8 *transferGraphs()*.....8

9 Drawing.....10

10 Fazit.....11

11 Referenzen.....12

1 Einführung

Die Arbeitsgruppe Electronic Vision(s) des Kirchhof Institut für Physik an der Universität Heidelberg hat eine neuromorphe analoge Hardware[3],[4] entwickelt, mit der sich neuronale Netzwerke emulieren lassen. Der Hauptvorteil einer solchen Hardware ist die Parallelisierung der Abläufe. Die Netze, die simuliert werden sollen, werden in der Python[6] basierten Scriptsprache pyNN[5] beschrieben, die eine Standardisierung darstellt und immer mehr an Bedeutung gewinnt. Nach der erfolgreichen Entwicklung und Realisierung der sogenannten Stage1[3] Hardware, bei der es sich um einzelne Chips mit jeweils 2 mal 192 Neuronen und etwa 100.000 synaptischen Verbindungen handelt, arbeitet die Electronic Vision(s) Gruppe jetzt an einer Stage2[4] Hardware, die auf ganzen Wafern implementiert werden soll. Diese soll aus etwa 400 sogenannten Hicanns pro Wafer bestehen, auf den jeweils 2 mal 256 Neuronen und etwas mehr als 100.000 synaptische Verbindungen emuliert werden können. Für die Abbildung der biologischen Netze auf die Hardware wurde von der TU Dresden ein Graphenmodell[1] entwickelt, welches mit Hilfe von Algorithmen es möglich macht, die optimale Abbildung (Mapping) zu finden. In dem Graphenmodell lassen sich die Struktur und die Fähigkeiten der jeweiligen Hardware wiederfinden. Meine Aufgabe bestand darin, die bereits vorhandene Visualisierungssoftware[2] umzuschreiben und anzupassen, sodass damit nun auch die Stage2 Hardware dargestellt werden kann.

2 Das Graphenmodell

Um die biologischen Netzwerk-Modelle auf die Hardware emulieren zu können muss man zunächst das sogenannte Bio-Graph erstellen, in dem die Neuronen durch Knoten dargestellt werden. Die Eigenschaften der Neuronen werden jedoch in eigene Knoten als Parameter gespeichert und können mittels Kanten den Neuronen zugewiesen werden. Das ermöglicht das Sparen von Speicherplatz bei einer guten Skalierbarkeit, denn der Satz möglicher Neuronen-Parameter ist hardware-technisch begrenzt. Die Kanten zwischen zwei Knoten können dabei auch die Rolle einer synaptische Verbindung spielen. Eine Hyper-Kante zeigt dann von der „Synapse“ auf einen entsprechenden Parameter-Knoten.

Die Hardware wird in dem sogenannten Hardware-Graph dargestellt, wobei dieser komplett hardware-spezifisch ist. Die Abbildung des neuronalen Netzwerks auf die Hardware geschieht

mittels eines Mapping-Algorithmus, der versucht die Neuronen auf dem Wafer so zu verteilen, dass später möglichst viele synaptische Verbindungen realisierbar sind. Danach versucht der Routing-Algorithmus die Schaltmatrizen so zu konfigurieren, dass das Bio-Netzwerk exakt auf der Hardware emuliert wird. Dabei muss der Routing-Algorithmus sicherstellen, dass eine möglichst kleine Signaldichte auf den Bussen vorliegt.
Man kann nun die so emulierten neuronalen Netzwerke sich entwickeln lassen, wobei man die Entwicklung mittels zusätzlicher digitaler Hardware mitverfolgen und speichern kann. Die so gewonnen Simulationsdaten sollen auch visualisiert werden.

3 Der Aufbau der Visualisierungssoftware

Da die Visualisierungssoftware, welche mit OpenGL[7] realisiert wird, in einem pyNN-Script integriert werden soll, wurde diese aus zwei Klassen aufgebaut, um sie für andere Programme zugänglich zu machen und die Verwaltung zu vereinfachen.
Die Klasse ***glControl*** verwaltet die allgemeinen OpenGL Funktionen, wie zum Beispiel das Fenster-Management oder die Menüs. Sie enthält aber auch Funktionen, mit dessen Hilfe man die Darstellung der 3D-Scene auf dem Bildschirm steuern kann oder auch die verschiedenen Graphen ein- und ausblenden kann.
Die Klasse ***GraphVisu*** ist für die Erzeugung des darzustellenden Bio-Netzwerkes und der Hardware zuständig. Sie enthält Funktionen, mit denen man den Bio-Graphen und den Hardware-Graphen ausließt und in lokale Container speichert, was für die flüssigen Darstellung mit OpenGL nötig ist. Die Klasse enthält außerdem Informationen darüber, was und wie gemalt werden soll.

4 OpenGL

OpenGL ist eine Programmierschnittstelle, die zur Erstellung von 3D-Anwendungen benutzt wird. Führende Graphik-Chip-Hersteller (NVIDIA, ATI) unterstützen die OpenGL-Standards. Da OpenGL plattformunabhängig ist, kann es zur Realisierung der Visualisierung der Graphen benutzt werden, was bei DirectX zum Beispiel nicht geht.

5 *glControl*

Die Klasse ***glControl*** dient, wie der Name schon nahe legt, der Kontrolle der Scene. Ein großes Stück dieses Teils der Visualisierungssoftware wurde übernommen, so wie es bereits für die Stage1 Software geschrieben wurde[2]. ***glControl*** übernimmt Aufgaben, wie z.B. das Erzeugen eines Fensters mit vorgegeben Parametern (Breite, Höhe, Position und Überschrift), oder das Management von Eingabe durch Tastatur und Maus mit entsprechenden Callbackfunktionen[2]. Eine weitere wichtige Funktion ist die Erstellung des Menüs, das dem Benutzter vielfältige Möglichkeiten bietet. Man kann damit verschiedene Teile der Visualisierung ausblenden oder anzeigen lassen, außerdem bietet es verschiedene Einstellungsmöglichkeiten bzgl. der Farben- und Belichtungsschemas oder der Fenstergröße bzw. Full-Screen-Modes. Das Menü erlaubt es auch, die Positionen der biologischen Neurone zu verändern und diese zum Beispiel zufällig auf eine Kugeloberfläche oder in einem Würfel zu platzieren. ***glControl*** verwaltet auch die OpenGL Matrizen, welche für die Darstellung der 3D Objekte auf dem Bildschirm wichtig sind. In den Funktionen ***draw3Dview*** und ***draw2Dview*** wird dafür gesorgt, dass die Hardware einmal von der Seite, also in 3D sozusagen, oder von oben (2Dview) dargestellt wird. Mit Hilfe von ***glControl*** wird auch das Auswählen von Objekten mit der Maus auf dem Bildschirm realisiert. Diese werden dann mit einer anderen Farbe dargestellt und ihre Parameter, die mit Hilfe der ***GraphVisu*** Klasse aus dem Bio- und dem Hardware-Graphen eingelesen wurden, werden eingeblendet. Die dafür nötigen Funktionen, die Texte auf dem Display erzeugen sind auch ein Teil der ***glControl*** Klasse.

6 *GraphVisu*

Diese Klasse erfüllt einige wichtige Aufgaben der Visualisierung und beschäftigt sich viel mit der eigentlichen Aufgabe: der Darstellung des biologischen Netzwerkes und der Hardware. Da es viel zu langsam wäre jedes mal die Informationen aus den Graphen auszulesen, falls man bei der Visualisierung die Scene neu malen möchte, was jedoch bei jeder Bewegung des Beobachters geschieht, werden die Graphen ein Mal mit der Funktion ***transferGraphs()*** gelesen und alles, was für die Visualisierung wichtig ist, wird in lokale Container gespeichert, was für eine mühelose und flüssige Darstellung sorgt. Diese Funktion besteht aus zwei Teilen, der erste Teil ließt den Hardware-Graphen aus und der zweite kümmert sich nicht nur um den

Bio-Graphen, sondern auch um das Mapping. Da der Hardware-Graph sehr kompliziert ist, bestand ein großer Teil meiner Arbeit darin, diesen richtig auszulesen. Die Klasse ***GraphVisu*** enthält auch Funktionen, die für das Malen (Drawing) von den Neuronen, Synapsen und der Hardware-Komponenten zuständig sind, oder die Berechnung der OpenGL-Positionen der darzustellenden Objekte im Raum übernehmen. Die Informationen über die Komponenten der Graphen werden in ***structs*** gespeichert, welche wiederum in Listen und Vektoren verwaltet werden. Die Klasse GraphVisu wurde komplett überarbeitet, da sie hardware-spezifisch ist.

7 Lokale Container

In den lokalen Containern werden alle Daten gespeichert, die für die Visualisierung nötig sind. Für das biologische neuronale Netzwerk werden zwei Arten von ***structs*** gebraucht: bio_synapse und bio_neuron. Beide sind im folgenden Ausschnitt aus dem Code dargestellt:

```
///struct for the synapses
struct bio_synapse
{
    GraphModel::GMConnection* graphConnection;      ///< Connection in the BioModel this synapse represents
    bio_neuron* pre;                                ///< Pre-synaptic neuron of the synapse
    bio_neuron* post;                               ///< Post-synaptic neuron of the synapse
    float weight;                                   ///< Weight of the synapse
};

///struct that contains all the basic information needed for fast rendering
struct bio_neuron
{
    GraphModel::GMNode* graphNode;          ///< Corresponding node in the BioModel for the neuron
    GLfloat pos[3];                         ///< Position of the neuron in space
    // the Ins-and-Outs
    std::vector<bio_synapse*> ins;          ///< Outgoing synapses of the neuron
    std::vector<bio_synapse*> outs;         ///< Incoming synapses of the neuron
    std::vector<hw_dendrite*> mappings;       ///< Hardware-dendtires mapped to this bio-neuron
};
```

Listing1: Der Code für die Container von biologischen Neuronen und Synapsen

Für das Darstellen des Hardware-Graphen braucht man mehr ***structs***, da hier eine Hierarchie vorhanden ist. Ein System besteht aus mehreren Wafern, jeder Wafer ist aus Hicanns aufgebaut, die wiederum Syndrivers, Dendrites und andere Hardware-Elemente enthalten. Genau für diese vier, oben genannte Elemente bietet auch *GraphVisu* ***structs: „***hw_wafer“, „hw_hicann“, „hw_dendrite“ und „hw_syndriver“. Die Hierarchie, die die Beziehungen dieser Elemente untereinander regelt, ist auf in dem Aufbau der entsprechenden ***structs*** zu finden. Der ***struct*** „hw_wafer“ zum Beispiel enthält einen Vektor „hicanns“, wo man Pointer auf alle zu diesem Wafer gehörenden Hicanns findet. Der ***struct*** „hw_hicann“ besitzt einen Pointer namens „parentWafer“, der auf dem Wafer zeigt, der diesen Hicann enthält. Diese doppelt vorhandene

Informationen sind trotzdem für die Visualisierung des Hardware-Graphen notwendig.

Im folgenden Ausschnitt aus dem Code findet man die vollständige Definition der ***structs:***

```
struct hw_dendrite
{
    GraphModel::GMNode* graphNode;      ///< Corresponding node in the HWModell for this hardware-neuron
    unsigned int nr;                    /// number within the Hicann
    bio_neuron* mapping;                ///< Bio-neuron this hardware-dendrite is mapped to
    hw_hicann* parentHicann;              ///< hicann this hardware-dendrite is part of
    hicann::Block Block;                        ///< Block this dendrite belongs to
    // ??? GLfloat pos[3];
};


struct hw_hicann
{
    GraphModel::GMNode* graphNode;              ///< Corresponding node in the HWModell for this hicann
    unsigned int nrX;
    unsigned int nrY;
    std::vector< std::vector< int > > SynapseMatrix; ///< Matrix mit den Synapsegewichten
    std::vector<hw_dendrite*> hwDendrites;      ///< Collection of hardware-dendrites that form this hicann
    std::vector<hw_syndriver*> syndrivers;      ///< The syndrivers belonging to the hicann
    hw_wafer* parentWafer;                      ///< Wafer this core is part of
    // ??? GLfloat pos[3];
};

struct hw_wafer
{
    GraphModel::GMNode* graphNode;              ///< Corresponding node in the HWModell for this wafer
    unsigned int nr;
    std::vector<hw_hicann*> hicanns;            ///< The hicanns belonging to the wafer
};


struct hw_syndriver
{
    GraphModel::GMNode* graphNode;
    unsigned int nr;                            /// within the Hicann
    hw_hicann* parentHicann;
    Syndriver::InputType input;
    Syndriver::Side side;
    hicann::Block Block;
    std::vector<unsigned char> weights;
};
```

Listing2: Der Code für die Container für die Hardwareelemente

All diese Container werden in der *GraphVisu* Klasse in Listen verwaltet. Diese Listen enthalten nicht Pointer auf Objekte, sondern die Objekte selbst. Außerdem gibt noch einige weitere Vektoren, die Pointer auf die Objekte enthalten. Diese Vektoren dienen als Tabellen mit denen man zum Beispiel jedem bio_neuron eine *glid* zuweist (*std::vector<bio_neuron*> glidToNeuronTable*), was man später für die Identifizierung beim Auswählen von Objekten auf dem Bildschirm braucht.

8 *transferGraphs()*

Diese Funktion dient des Auslesens der Graphen und der Übertragung der nötigen Informationen auf die lokalen Container. Dabei geht man folgendermaßen vor:

Man geht zu dem Systemknoten des Hardware-Graphen und durchsucht seine Kinderknoten, bis man den Knoten „Wafers“ findet. Danach iteriert man durch die Kinderknoten von „Wafers“, die ja einzelne Wafer darstellen. Für jeden so gefundenen Wafer erzeugt man ein neues Objekt vom Typ hw_wafer (das ist ein lokaler ***struct***, in dem die Informationen über den Wafer gelagert werden sollen) in der Liste hwWafers und füllt diesen mit den Daten des Wafers aus dem Hardware-Graphen. Ist dies geschehen, so sucht man weiter in den Kinderknoten von diesem Wafer-Knoten nach „Hicanns“, dessen Kinderknoten wiederum einzelne Hicanns sind. Für jedes dieser Hicanns erzeugt man dann ein Eintrag in die Liste hwHicanns und füllt diese mit entsprechenden Informationen. Der folgende Ausschnitt aus dem Code des ***transferGraphs()*** erfüllt genau die Aufgaben, die ich oben beschrieben haben:

```
       std::vector<GraphModel::GMNode*>* graphSystemNodeComponents = hwGraph->SystemNode->Components;
       for (std::vector<GraphModel::GMNode*>::iterator graphSystemNodeComponentsIterator=graphSystemNodeComponents->begin();
graphSystemNodeComponentsIterator!=graphSystemNodeComponents->end(); graphSystemNodeComponentsIterator++) // iterator über Kinderknoten des Systemknotens
       {
           std::cout << "suche nach waferContainer: " << (*graphSystemNodeComponentsIterator)->Name << std::endl;
           if(strcmp((*graphSystemNodeComponentsIterator)->Name, "Wafers") == 0)
           {
               std::cout << "habe waferContainer" << std::endl;
               waferContainer = (*graphSystemNodeComponentsIterator); // elternknoten von den eizelnen wafers
               std::vector<GraphModel::GMNode*>* wafers = waferContainer->Components; // vektor aus wafers
               for (std::vector<GraphModel::GMNode*>::iterator wafersIterator=wafers->begin(); wafersIterator!=wafers->end(); wafersIterator++) //
iterator ist wafer
               {
                   std::cout << "hab' wafer" << std::endl;
                   wafer = (*wafersIterator);
                   hwWafers.push_back(hw_wafer());
                   hw_wafer* hwWafer = &hwWafers.back();
                   hwWafer->graphNode = wafer;
                   hwWafer->nr = atoi (wafer->Name);
                   std::vector<GraphModel::GMNode*>* waferComponents = wafer->Components; // vektor aus den Kinderknoten eines waferknotens
                   for (std::vector<GraphModel::GMNode*>::iterator waferComponentsIterator = waferComponents->begin(); waferComponentsIterator !=
waferComponents->end(); waferComponentsIterator++) // iterator über diese Kinderknoten
                   {
                       std::cout << "suche nach hicannContainer: " << (*waferComponentsIterator)->Name << std::endl;
                       if (strcmp((*waferComponentsIterator)->Name, "Hicanns") == 0)
                       {
                           hicannContainer = (*waferComponentsIterator); // enthält alle Hicanns
                           std::vector<GraphModel::GMNode*>* graphHicanns = hicannContainer->Components;
                           std::cout << "hab' hicannContainer: " << (*graphHicanns).size() << std::endl;
                           for (std::vector<GraphModel::GMNode*>::iterator hicannsIterator = graphHicanns->begin(); hicannsIterator != graphHicanns->end
(); hicannsIterator++) // iterator über alle Hicanns
                           {
```

Listing3: Ausschnitt aus dem Code der Funktion ***transferGraphs()***

danach arbeitet man sich nach dem gleichen Schema wie oben durch den gesamten Hardware-Graph und sammelt alle nötigen Informationen. Dabei wird auch noch eine wichtige ***map***, die die Dendriten aus dem Hardware-Graphen mit den entsprechenden ***structs*** vom Typ **hw_dendrite** verknüpft. Dies ist später für die Übertragung der Mapping-Daten aus den

Graphen sehr wichtig.

Dann geht man zu dem Bio-Graphen, wo man durch die Kinderknoten vom Knoten „Neurons“ iteriert und mit den sich dort befinden Neuronen die Liste „**Neurons**“ mit den lokalen Containern vom Typ ***bio_neuron*** füllt. Diese ***structs*** von Typ ***bio_neuron*** enthalten dann Informationen, wie zum Beispiel die Position des Neurons im Raum.

Für jeden in Bio-Graphen gefundenen Neuron schaut man sich auch noch alle Kanten, die von diesem Neuron-Knoten ausgehen. Handelt es sich bei einer bestimmtem Kante um eine „***NEURO_NEURO***“ Kante, so ist es eine Synapse, diese wird mit Hilfe der Funktion „***updateConnectionSynapseMap()***“ lokal angelegt und in die Liste der Synapsen eingetragen. Ist die Kante vom Typ „***MAPPING***“, so werden die entsprechenden Mapping-Informationen mit Hilfe der ober erwähnten ***map*** in die Lokalen ***structs*** übertragen.

Der letzte Teil von ***transferGraphs()*** benutzt eine Klasse namens ***GMAccess***, welche von der Electronic Vision(s) Gruppe und an der TU Dresden entwickelt wurde. Diese Klasse stellt eine einfache Möglichkeit dar, auf den Graphen der Stage2 Hardware zuzugreifen und nötige Informationen ohne das schreiben von viel redundantem Code zu extrahieren. Hier wurde die Funktion ***getSynapses()*** von ***GMAccess*** benutzt mit dessen Hilfe, man die Matrix der synaptischen Gewichte für die einzelnen Hicanns bekommt. Dabei iteriert man über die lokale Liste der Hicanns, ruft für jeden ***getSynapses()*** auf und speichert die Gewichte-Matrix in den jeweiligen lokalen ***struct***. Hier ist der entsprechende Code:

```
GMAccess gmaccess;
gmaccess.init(hwGraph, bioGraph);
for (std::list<hw_hicann>::iterator hicannsIterator = hwHicanns.begin(); hicannsIterator != hwHicanns.end(); hicannsIterator++)
{
    hw_hicann* SynHicann = &*hicannsIterator;
    unsigned int X = SynHicann->nrX;
    unsigned int Y = SynHicann->nrY;
    std::vector<std::vector<GMAccess::syn_t> > syn;
    gmaccess.getSynapses(X, Y, syn);
    SynHicann->SynapseMatrix.resize(syn.size());
    for (unsigned int i = 0; i < syn.size(); i++)
    {
       SynHicann->SynapseMatrix[i].resize(syn[i].size());
       for (unsigned int j = 0; j < syn[i].size(); j++)
       {
            SynHicann->SynapseMatrix[i][j] = (int)( (syn[i][j]).weight ) ;
       }
    }
    if ( (SynHicann->nrX == 0) and (SynHicann->nrY == 0) ) {
    for (unsigned int i = 0; i < SynHicann->SynapseMatrix.size()/10; i++)
    {
       for (unsigned int j = 0; j < SynHicann->SynapseMatrix[i].size(); j++)
       {
            std::cout << SynHicann->SynapseMatrix[i][j] << " ";
       }
       std::cout << std::endl;
    } }
}
```

Listing4: Weiteres Beispiel aus dem Code der Funktion ***transferGraphs()***

9 Drawing

Das zeichnen des biologischen neuronalen Netzwerkes und der Hardware geschieht mit Hilfe von einer Gruppe von hierarchisch organisierten Funktionen. Dabei wird die Funktionen ***drawBioGraph()*** und ***drawHWGraph()*** vom ***glControl*** aufgerufen. d***rawBioGraph()*** stellt den biologischen Teil der Visualisierung dar und zeichnet neben den Neuronen auch die ein- und/oder ausgehenden Synapsen von angeklickten/ausgewählten Neuronen dar. Dieser Teil des Codes wurde weitgehend von der Software für Stage1 übernommen, das sich am Bio-Graph nichts geändert hat. ***drawHWGraph()*** geht die lokale Liste der Wafers durch und ruft für jedes Wafer die Funktion ***drawHWWafer()*** auf. Diese zeichnet eine Scheibe, die den Wafer repräsentieren soll und geht die Liste mit den zu diesem Wafer gehörenden Hicanns durch. Für jeden Hicann wird die Funktion aufgerufen, welchen einen Grundriss zeichnet. Wegen der großen Anzahl von Elementen (256 Synapsentreiber, 512 Dendrites, über 100.000 synaptische Gewichte) ist es nicht möglich alle gleichzeitig darzustellen, weil das die Ressourcen des Rechners zu stark in Anspruch nimmt. Stattdessen prüft ***drawHWGraph()***, ob ein Neuron ausgewählt ist und ruft dann die Funktion ***drawHWHicannDetailed***, wo die Dendrites, die Synapsentreiber und die synaptischen Gewichte auf den Grundriss des Hicann gemalt werden, welcher die auf den ausgewählten Neuron gemappten Dendrites enthält. Die entsprechenden Elemente werden in einer anderen Farbe dargestellt und es wird eine Mapping-Kante eingezeichnet. Damit wird es möglich die Realisierung des biologischen neuronalen Netzwerkes durch die Hardware zu beobachten. Hier sind noch zwei Screenshots von der Visualisierung zu sehen:

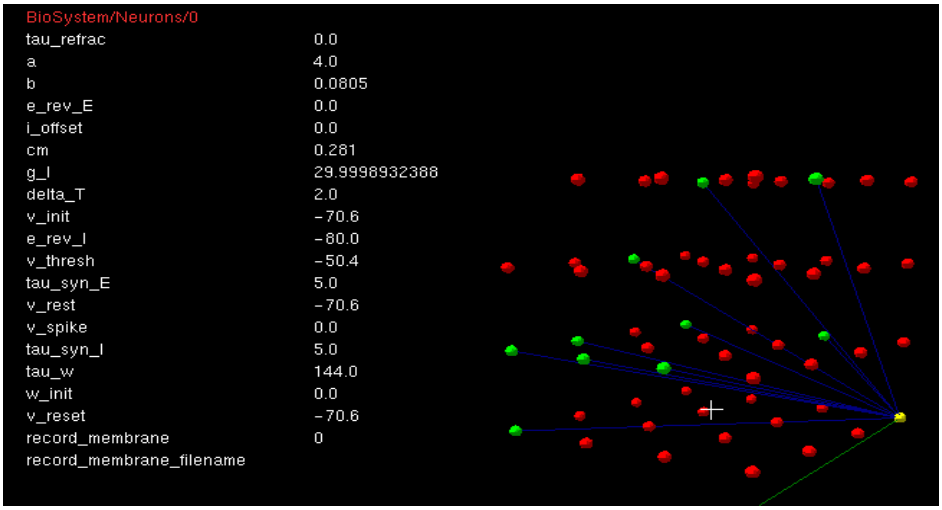

Bild1: Screenshot der Visualisierung des biologischen neuronalen Netzwerks

Im Bild1 auf der vorherigen Seite sieht man den Bio-Graphen mit allgemein roten Neuronen, mit einem gelben ausgewählten Neuron, von dem blauen Synapsen ausgehen und in grün markierte sekundären Neuronen eingehen. Die grüne Linie in der rechten unteren Ecke ist eine Mapping-Kante, die zum Hardware-Graphen führt:

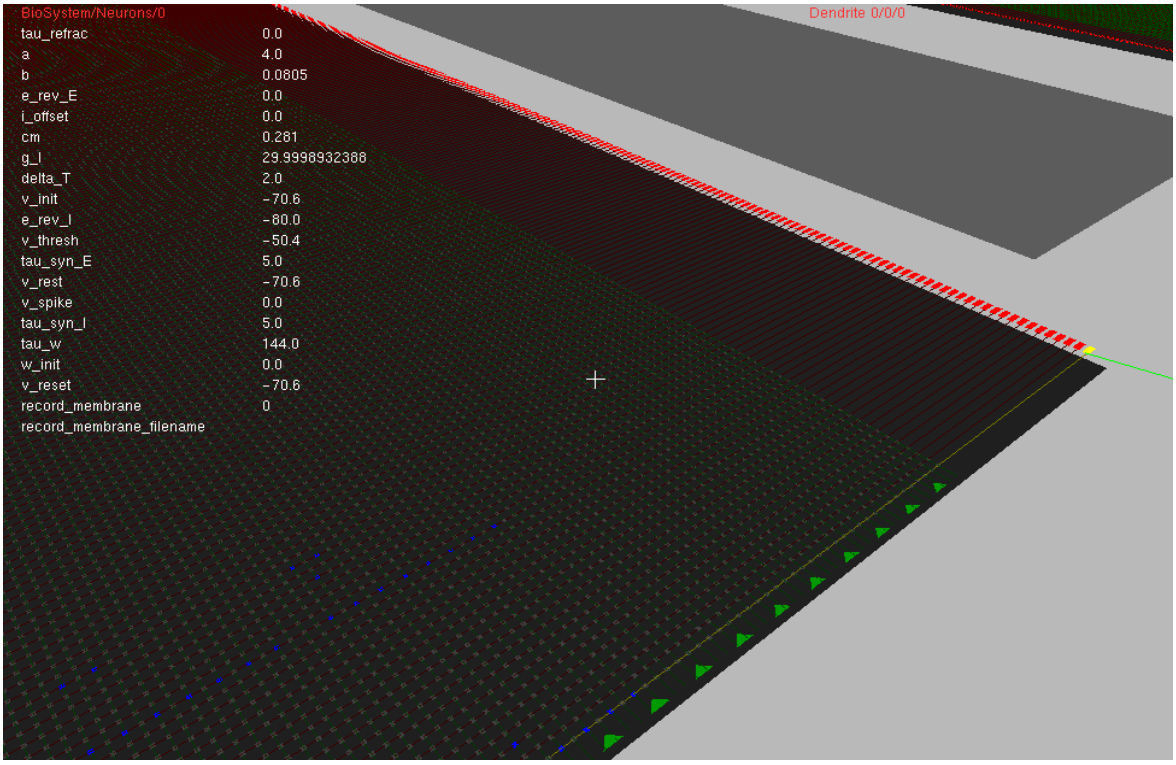

Bild2: Screenshot der Visualisierung der neuromorphen Hardware

Hier sieht man einen Ausschnitt aus dem Hardware-Graphen. Zu sehen sind Synapsentreiber (grüne Dreiecke), Dendrites (rote Rechtecke). Eine dünne grüne Linie zeigt auf einen gelb markierten Dendrite. Das ist die Mapping-Kante von oben. Zu sehen sind auch graue und blaue Quadrate; diese repräsentieren die synaptischen Gewichte (grau: inaktiv, blau: exzitatorisch, rot: inhibitorisch (hier nicht zu sehen)).

10 Fazit

Bei der Visualisierung der Stage2 Hardware hatte ich ein breites Spektrum an Aufgaben: das Erlernen und Anwenden von OpenGL, die Arbeit mit allgemeinem C++ Funktionen und die Benutzung von den Graphenmodellen. Dabei habe ich viel gelernt und konnte letztendlich die mir gestellten Aufgaben bewältigen. Hierbei wurde ich oft von dem Software-Team der Electronic Vision(s) Gruppe unterstützt. Die Arbeit an der Visualisierungssoftware für die

Stage2 Hardware ist jedoch noch nicht abgeschlossen. Ich sehe einige Möglichkeiten für eine Weiterentwicklung. Man sollte das Routing auf der Hardware visualisieren, so dass erkennbar ist, wie die synaptischen Verbindungen in dem Bio-Modell durch der Hardware realisiert werden. Eine andere Verbesserung würde durch die Skalierbarkeit der dargestellten Hardware entstehen. Denn momentan sind die Größen der Hardwareelemente in dem Code fest programmiert, es wäre jedoch sinnvoll die Hardware gegenüber des Bio-Netzwerkes skalieren zu können, was der besseren Veranschaulichung dienen kann.

11 Referenzen

[1] K. Wendt, M. Ehrlich, and R. Schueffny. A graph theoretical approach for a multistep mapping software for the facets project. In CEA'08: Proceedings of the 2nd WSEAS International Conference on Computer Engineering and Applications, pages 189–194, Stevens Point, Wisconsin, USA, 2008. World Scientific and Engineering Academy and Society (WSEAS). ISBN 978-960-6766-33-6.

[2] Tobias Harion, 3D-Visualisierung einer Abbildung von neuronalen Netzwerkmodellen auf eine neuromorphe Hardware, http://www.kip.uni-heidelberg.de/cms/fileadmin/groups/vision/Downloads/Internship_Reports/report_harion.pdf

[3] J. Schemmel, D. Bruederle, K. Meier, and B. Ostendorf. Modeling synaptic plasticity within networks of highly accelerated I&F neurons. In Proceedings of the 2007 IEEE International Symposium on Circuits and Systems (ISCAS'07). IEEE Press, 2007.

[4] J. Schemmel, J. Fieres, and K. Meier. Wafer-scale integration of analog neural networks. In Proceedings of the 2008 International Joint Conference on Neural Networks (IJCNN), 2008.

[5] PyNN. A Python package for simulator-independent specification of neuronal network models – website. http://www.neuralensemble.org/PyNN, 2008.

[6] Python. The Python Programming Language – website. http://www.python.org, 2009.

[7] OpenGL. Website. http://www.opengl.org.