3D-Visualisierung einer Abbildung von neuronalen Netzwerkmodellen auf eine neuromorphe Hardware

Tobias Harion
eMail: Harion@stud.uni-heidelberg.de

Betreuer: Daniel Bruederle

September 7, 2008

Contents

Einführung

Die Electronic Vision(s) Gruppe des Kirchhoff-Instituts für Physik an der Universität Heidelberg hat eine neuromorphe Hardware entwickelt [3], welche in der Lage ist, neuronale Netze zu emulieren. Die zu simulierenden Netze werden in der pythonbasierten Scriptsprache pyNN [5] beschrieben. Um die so beschriebenen Netze auf die Hardwareeinheiten abzubilden wurde von der TU Dresden ein Graphenmodell entwickelt, welches anhand spezieller Algorithmen eine optimale Abbildung (Mapping) finden soll. Die verschiedenen Entwicklungsstadien der Hardware lassen sich auch in dem Graphenmodell wiederfinden. Die hier vorgestellte Visualisierung des Mappings ist dafür vorgesehen, die anfänglichen Modelle, welche mit Stage I Modellen bezeichnet werden, darzustellen. Unabhängig von dem Hardwaremodell lassen sich Simulationsdaten, welche von NEST [12] erstellt wurden, in dem beschriebenen Netz darstellen.

Das Graphenmodell

Bei der Simulation biologisch inspirierter Netze müssen zunächst die Neuronen und deren Verbindungen auf die Hardware abgebildet werden. Um dabei die zur Verfügung stehende Hardware optimal auszunutzen werden Algorithmen (der TU Dresden) verwendet um das vorliegende Netz auf die Hardware zu mappen. Für dieses Mapping wird ein Graphenmodell verwendet, welches in seinen Knoten die erforderlichen Daten speichert. Die Knoten sind untereinander durch entsprechende Kanten verbunden, welche angeben in welche Bedeutung der Zielknoten hat. Jeder Graph startet dabei mit einer Systemnode, welche auf weitere Teile des Graphenmodells verweist. Die Knoten des Graphenmodells sind alle von der Klasse GMNode und haben somit, unabhängig von ihrer Funktion, den gleichen Aufbau.

Aufbau der Visualisierung

Um die Visualisierung für andere Programme leicht zugänglich zu machen wurden zwei Klassen verwendet. Die Klasse *glControl* verwaltet die Grundlegenden Funktionen von OpenGL. So sind hier die Funktionen zu finden, um ein Fenster einer bestimmten Grösse zu erstellen und die Eingaben des Benutzers in diesem Fenster zu verarbeiten.
In der Klasse *GraphVisu* sind alle Funktionen zur Erzeugung der darzustellenden Szene zu finden, wie z.B. das Zeichnen des biologischen Systems und dessen Synapsen, oder die Mappingverbindungen zwischen dem Biomodell und dem Hardwaremodell. Ausgehend von der Funktion set_scene werden die gewünschten Teile gezeichnet. Zum erstellen der Visualisierung wird hier auf die Klasse glControl zurückgegriffen.

OpenGL

OpenGL [7] ist eine plattformunabängige Programmierschnittstelle für die Entwicklung von 3D Anwendungen. Der OpenGL-Standard wird vom OpenGL ARB (Architecture

Review Board) festgelegt. Mitglieder des ARB sind unter anderem führende Hersteller von Grafikkarten wie NVIDIA [6] und AMD/ATI [1]. Als Alternative zu OpenGL existiert DirectX [2], welches jedoch wegen seiner Plattformabhängigkeit für die Visualisierung nicht in Frage kam.

glControl

Die Klasse glControl bedient sich der freeglut [4] Library. Mit ihr lässt sich das Erstellen und Verwalten der Fenster programmieren, ohne die Anbindung des Fensters an den X-Server selbst kontrollieren zu müssen. Die weite Verbreitung der GL Utility Library gewährleistet die Kompatibilität der Klasse mit verschiedenen Betriebssystemen und Distributionen.

Erstellen eines Fensters

Um ein Fenster zu erzeugen in dessen Kontext die OpenGL Szene gerendert werden kann, müssen zunächst die Parameter glControl.w und glControl.h für die gewünschte Breite und Höhe des Fensters in Pixeln gesetzt werden. Anschliessend wird durch Aufruf der Funktion glControl.window(char* caption, int pos_x, int pos_y) das Fenster erzeugt. Der Funktion werden dabei die Position der linken oberen Ecke, sowie die Ueberschrift des Fensters übergeben.

Registrieren der Callbackfunktionen

Erfolgt eine Eingabe über die Maus oder die Tastatur, so muss eine entsprechende Callbackfunktion ausgeführt werden. Die Callbackfunktionen selbst befinden sich in der Klasse glControl, müssen jedoch durch eine Wrapperfunktion registriert werden, welche sich bei der aufrufenden Klasse befinden. Will man den Tastatureingaben andere Funktionen zuordenen, so muss man diese in der Funktion glControl.keystroke oder glControl.specialkeys eintragen. Ebenso müssen die auszuführenden Funktionen bei Mauseingaben in den entsprechenden Funktionen definiert werden. Die Funktion set

Weitere Funktionen

Die Klasse beinhaltet weitere Funktionen um die Visualisierung einfacher zu steuern. Die Funktion *StartPicking()* selektiert das Objekt an der aktuellen Mausposition. Da das selektierte Objekt direkt in der Klasse GraphVisu eingetragen wird, lässt sich diese Funktion nicht ohne grössere Aenderungen auf andere Anwendungen übertragen.
Desweiteren sind Funktionen zur Darstellung von Text als 3D-Schrift an einer Position in dem Raum, sowie als 2D Text an einer Position in dem Fenster vorhanden.
Um Variablen zu kontrollieren existiert eine Konsole, welche über die Funktion *show_terminal()* ein- bzw. ausgeblendet wird. Zusammen mit der Funktion *parse_cmd* lassen sich so Werte von bestimmten Variablen einfach ändern. Um die bekannten Befehle zu erweitern muss die Funktion *parse_cmd()* erweitert werden.

GraphVisu

Aus dem Graphenmodell müssen die benötigten Informationen extrahiert werden, um die Neuronen und ihre Synapsen in einem dreidimensionalen Raum darzustellen. Welche Teile des Graphenmodells dargestellt werden wird von der Funktion set_ scene() gesteuert. Sie wird von der Funktion glControl→ display() aufgerufen und ist für das eigentliche Zeichnen der Objekte zuständig. Da die Geschwindigkeit, mit der der Graph durchlaufen wird zwar ausreichen für eine statische Darstellung des Systems ist, allerdings für Animationen zu langsame Zugriffszeiten aufweist, wird das Biomodell aus dem Graphen in STL Container übertragen. Dadurch verringert sich die Zugriffszeit merklich und ist ausreichend um auch Simulationen graphisch darzustellen.

Positionierung der Neuronen

Da das ursprünglische Graphenmodel in den Knoten keine Positionsangaben gespeichert hat, wurde die Klasse GMNode um ein Float-Array *pos* erweitert. Dort kann die Position des Neurons gespeichert werden.
Um dem Biosystem Positionen zuzuordnen stehen 3 Funktionen zur Verfügung:

Bio_SetPositions(): Ordnet die Neuronen auf einem kubischen Gitter an.

Bio_SetSpherePositions(): Ordnet die Neuronen zufällig innerhalb einer Kugel an.

Bio_SetPlanPositions(): Ordnet die Neuronen auf einem Gitter in einer Ebene an.

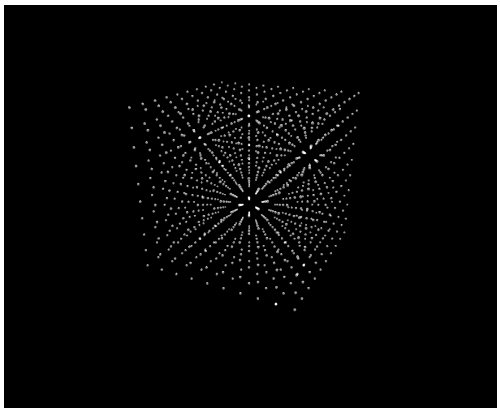

Figure 0.1: Darstellung der Neuronen auf einem kubischen Gitter

Die Verwendung dieser Funktionen ist nicht zwingend. Die Variablen für die Positionsangaben sind als Public definiert und somit für andere Funktionen zugänglich. Dadurch können beliebige Positionierungen implementiert werden, was vor allem bei der Darstellung von Netzen mit entfernungsabhängigen Synapsenverbindungen von Vorteil ist.

Da die Hardware im Gegensatz zu dem Biomodell immer aus einer bestimmten Anzahl an Spikeycores [9][10] mit jeweils 192 Neuronen besteht, existiert hierfür nur eine

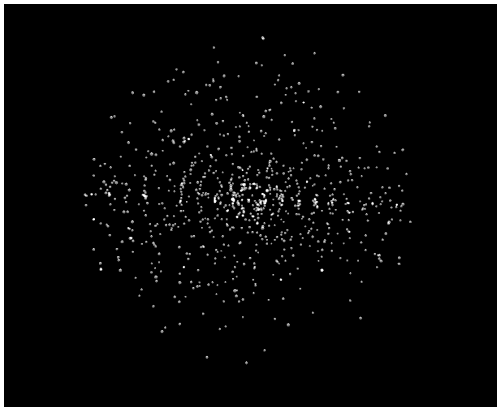

Figure 0.2: Darstellung der Neuronen in einer zufälligen kugelförmigen Anordnung

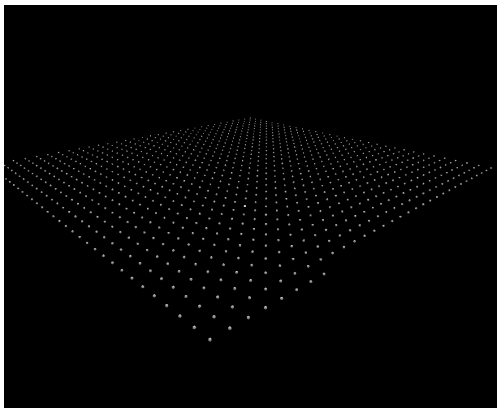

Figure 0.3: Darstellung der Neuronen in einer Ebenen

vordefinierte Positionierungsfunktion. Dabei werden die Hardwareneuronen eines Cores zusammen dargestellt und die Spikeycores etwas versetzt nebeneinander positioniert. Die Positionsvariablen sind auch hier von aussen zugänglich, sodass auch andere Darstellungen des Hardwaremodells möglich sind.

Transfer des Biomodells

Um die Neuronen des Biomodells zu übertragen, wird von dem Knoten *Neuronsnode* aus alle Unterknoten durchlaufen, welche gerade den Neuronen des Biomodells entsprechen. Dabei wird von jedem Knoten die Adresse des Knotens im Speicher, die ihm zugewiesene Position, sowie die von ihm ausgehenden Synapsenverbindungen in STL Containern [11] gespeichert. Die ausgehenden Synapsenverbindungen werden dabei ebenfalls als STL Map gespeichert, da hier ausser einem Pointer auf das Target selbst noch das Gewicht der Verbindung und die Position des Targets abgelegt sind.

Da das Hardwaremodell noch in einem Entwicklungsstadium war (fehlende Verbindungen zwischen den Hardwarekomponenten u.a.) wurde das Hardwaremodell noch nicht für schnelleren Zugriff in einen analogen Container übertragen. Da die Simulationen an dem Biomodell dargestellt werden bringt eine Konvertierung keinen signifikanten Geschwindigkeitsvorteil. Sobald innerhalb des Hardwaremodells ebenfalls die Verbindungen angezeigt werden, sollte das Modell ebenfall in geeignetere Container übertragen werden.

Darstellen der Modelle

Um nun das Biomodell darzustellen wird mit einem Iterator der Container *bio_map* durchlaufen und je nach gesetzten Flags zusätzlich die synaptischen Verbindungen oder die Daten der Simulationsdatei dargestellt. Dazu werden die zuvor übertragenen Positionsdaten an die entsprechenden OpenGL Befehle gegeben und an den Positionen Kugeln gezeichnet. Eine kurze Einführung in die dazu benötigten OpenGL Befehle wird im Anhang gegeben.

Das Hardwaremodell wird ähnlich gezeichnet, jedoch werden hier die Daten direkt aus dem Graphenmodell ausgelesen. Dabei wird zunächst anhand der Bezeichnungen der Knoten eines Spikeycores ausfindig gemacht und anschliessend alle Unterknoten durchlaufen. Dadurch werden die Knoten der Spikeycores zu den einzelnen Cores gruppiert. Bei dem Zeichnen werden die einzelnen Knoten geprüft, ob das aktuell selektierte Neuron des Biomodells von dem Algorithmus auf dieses Hardwareelement abgebildet wurde. Das Markieren der Knoten wird von der Funktion *draw_MappingOfSelNeuron* übernommen, welche auf jeden Fall vor dem Zeichnen des Hardwaremodells aufgerufen werden sollte. Sie vermerkt neben der Abbildung des selektierten Neurons auch die Abbildungen der mit diesem Neuron verbundenen Neuronen. Dadurch lässt sich überprüfen, ob der Mappingprozess die verbundenen Neuronen auf einen Spikeycore abbildet, oder ob diese über mehrere Cores aufgeteilt werden. Da das Hardwaremodell noch unvollständig ist, wäre es sinnvoll, die nötigen Informationen wie auch bei dem Biomodell in einer geeigneten Datenstruktur zu speichern. Dadurch muss bei Veränderungen des Hardwaremodells nur die entsprechende Funktion zur Uebertragung angepasst werden, und nicht mehr die gesamte Zeichenfunktion.

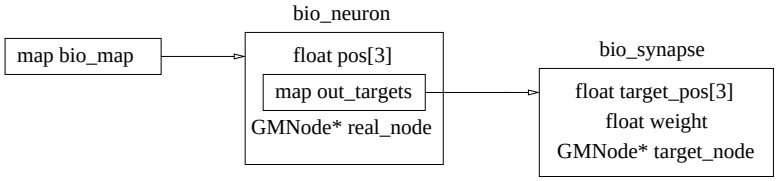

Figure 0.4: Aufbau des Containers für das Biomodell

Darstellung von Simulationsergebnissen

Die Ergebnisse von Simulationen lassen sich an dem Biomodell darstellen, sofern die Simulationsdatei in einem NEST-konformen Format vorliegt:

```
<Zeit>        <NeuronID>
```

Dabei steht hinter jeder Zeitmarke nur eine NeuronID, welche mit den ID-Nummern des Biomodells übereinstimmen müssen. Bei NEST Simulationsdateien ist dies gerade der Fall.
Feuern mehrere Neuronen gleichzeitig, so wird die Zeitmarke mehrmals untereinander aufgefuehrt mit der entsprechenden NeuronID. Zeilen mit # als erstes Zeichen gelten als Kommentar und werden nicht in der Simulation dargestellt.
Die Simulationsdatei wird in einen STL Container "spike_events" übertragen, dabei wird jeder Zeitmarke ein Set an NeuronIDs zugeordnet. Das Set enthält gerade die zu diesem Zeitpunkt feuernden Neuronen. Um eine Simulationsdatei zu laden, muss in der Konsole der Befehl "simfile Datei" aufgerufen werden.
Zur Darstellung der Simulation wird mit OpenGL eine Animation durchgefuehrt. Dies bedeutet, dass vor jedem Zeichnen der Szenen noch eine Funktion aufgerufen wird, welche die darzustellenden Objekte beeinflusst. Für die Simulation ist dies die Funktion *glControl →show_spikes()*. In ihr wird lediglich eine Zeitmarke, welche die aktuelle Simulationszeit angibt um ein einen festen Wert erhöht. Simulationszeitpunkt und Zeitintervalle können ebenfalls ueber die Konsole eingestellt werden.
Anschliessend wird die Szene erneut gezeichnet und dabei überprueft, ob die ID des gerade zu zeichnenden Neurons einen Eintrag bei der aktuellen Zeitmarke hat. Ist ein Eintrag vorhanden, so wird die Farbe vor dem Zeichnen verändert.
Die Simulation benötigt besonders bei grossen Netzen und einer langen Simulationsdauer viel Rechenleistung und sollte daher auf langsameren Rechnern nur mit angepassten Simulationsintervallen verwendet werden.

Anhang: Benötigte Befehle zum Zeichnen von einfachen Objekten

In diesem Abschnitt werden einige elementare Befehle für OpenGL beschrieben. Für eine vollständige Beschreibung bietet sich das Redbook an[8].

Matrixmanipulation

OpenGL nutzt Matrizen, um die zu zeichnenden Objekte in dem dreidimensionalen Raum entsprechend zu Positionieren und zu drehen. Die verwendete Matrix lässt sich dabei durch elementare Funktionen verändern um Drehungen und Translationen zu erreichen. Es wird zwischen der Modelviewmatrix und der Projectionmatrix unterschieden. Die Modelviewmatrix ist wie bereits erwähnt für die Positionierung und Ausrichtung eines zu Zeichnenden Objektes im Raum verantwortlich. Die Projectionmatrix bildet die dargestellte Szene auf den Bildschirm ab. Durch verändern der Projectionmatrix lassen sich Verzerrungen oder andere Projektionsmethoden der Szene auf den Bildschirm einstellen. Die Klasse glControl kontrolliert die Projectionmatrix und sorgt dafür, dass

auch bei Veränderung der Fenstergrösse keine Verzerrungen auftreten. Daher sollte sichergestellt werden, dass Veränderungen nur an der Modelviewmatrix vorgenommen werden.
Um den verwendeten Matrixstack auszuwählen dient die Funktion *glMatrixMode(GL_MODELVIEW)* bzw. *glMatrixMode(GL_PROJECTION)*

Verändern der Modelviewmatrix

Zunächst kann mit dem Befehl *glLoadIdentity()* eine Einheitsmatrix geladen werden. Wird nach diesem Befehl ein Objekt gezeichnet, so wird es immer ohne Rotation und Translation in dem Koordinatenursprung gezeichnet.
Weitere nützliche Funktionen sind *glPushMatrix()* und *glPopMatrix()*. OpenGL verwaltet 2 Matrixstacks. Der Stack für die Modelviewmatrizen kann bis zu 32 Matrizen fassen. Der Stack für Projektionsmatrizen kann nur 2 Matrizen aufnehmen. Der Befehl *glPushMatrix()* sichert die aktuelle Matrix auf dem Stack, sodass man nach dem Zeichnen eines Objektes an einem bestimmten Punkt im Raum nicht eine Translation in die entgegengesetzte Richtung ausführen muss, sondern durch einen Aufruf von *glPopMatrix()* wieder zu der Ausgangsposition zurückgelangt.
Um sich an einen Punkt in dem Raum zu bewegen wird die Funktion *glTranslatef(float x,float y,float z)* benutzt. Mit ihr bewegt man sich relativ zu der aktuellen Position an eine neue Position, welche durch den Vektor (x,y,z) gegeben ist.
Um ein Objekt rotiert darzustellen, dient die Funktion *glRotatef(float winkel, float x,float y, float z)*. Dieser Befehl führt eine Rotation um den gegebenen Winkel in Grad aus. Die Rotationsachse wird durch den Vektor (x,y,z) gegeben.
Um ein Objekt zu verformen oder zu skalieren dient die Funktion *glScalef(float x, float y, float z)*, welche die Koordinaten anschliessend mit den gegebenen Faktoren skaliert.

Darstellung von Objekten

Zur Darstellung von einfachen Objekten wie Würfeln, Kugeln oder Kegeln stehen Funktionen der GLUT zur Verfügung. Komplexere Objekte müssen als Polygone beschrieben werden.
Um eine einfache Kugel zu zeichnen genügt der Befehl *glutSolidSphere(float ,int, int)*. Das erste Argument bestimmt die Grösse der Kugel, die letzten beiden Argumente geben an, wieviele Segmente für die Darstellung der Kugel verwendet werden sollen. Je grösser diese Werte, desto runder wird die Kugel, jedoch werden dadurch auch Lichtreflektionen rechenintensiver.
Werden viele gleichartige Objekte gezeichnet lohnt sich der Einsatz von Displaylists. Dabei wird das zu Zeichnende Objekt vorkompiliert in dem Speicher der Grafikkarte gespeichert und kann mit einem einfachen Befehl deutlich schneller gezeichnet werden. Um eine Displaylist zu erstellen werden die Befehle *glNewList(int Nummer,GL_COMPILE)* und *glEndList()* benutzt. Ein Objekt, welches zwischen dem Aufruf von glNewList und glEndList gezeichnet wird, wird vorkompiliert in den Grafikkartenspeicher geschrieben und mit einer Integerzahl identifiziert, welche in der Variablen *nummer* gespeichert wird.

Das Objekt kann dabei auch aus mehreren einfacheren Objekten zusammengesetzt sein. Die Option GL_COMPILE bedeutet, dass das Objekt nur vorkompiliert werden soll, jedoch noch nicht gezeichnet wird. Um das Objekt zu zeichnen genügt dann ein Aufruf von *glCallList(int nummer)* mit der entsprechenden Nummer des Objekts.

Beispiel

Um die Verwendung der Befehle deutlicher zu machen wird hier noch ein kleines Beispiel gegeben, in dem Zwei Kugeln mit Hilfe von Displaylists gezeichnet werden.

```
void draw_spheres()
{
int nummer;

glMatrixMode(GL_MODELVIEW);  //Verwenden des Modelviewstacks
        glNewList(nummer,GL_COMPILE);  //Erstellen der Displaylist
        glutSolidSphere(0.07,10,10);   //Zeichnen einer Kugel mit Radius 0.07
        glEndList();
glLoadIdentity();  //In den Koordinatenursprung zur\"uckgehen
glTranslatef(5,0,0);  //An den Punkt (5,5,0) gehen
glPushMatrix();  //Matrix sichern
glRotatef(90,0,1,0);  //90 Grad um die y-Achse Rotieren
glTranslatef(1,0,0);  //das Gleiche wie eine Transformation direkt an den Ort (5,0,1)
glCallList(nummer);  //schnelles Zeichnen der Kugel
glPopMatrix();  //Zur\"uck an den Ort (5,0,0) ohne Rotation
glTranslatef(1,1,0);  //Jetzt sind wir an dem Ort (6,1,1)
glutSolidSphere(1,100,100);  //Das ist eine deutlich gr\"ossere und rundere Kugel
}
```

Tastenkürzel und Befehle der Kommandozeile

Tastenkürzel

Bewegungen innerhalb des Raumes

w nach vorne bewegen

s nach hinten bewegen

a nach links bewegen

d nach rechts bewegen

r nach oben bewegen

f nach unten bewegen

Kontrolle der Darstellung der Modelle

i Mappingkanten anzeigen

I Mappingkanten ausblenden

k Synapsen des Biomodells anzeigen

K Synapsen des Biomodells ausblenden

h Hardwaremodell anzeigen

H Hardwaremodell ausblenden

n Spikeycorenamen anzeigen

N Spikeycorenamen ausblenden

b Transparenz an (manche Objekte besser sichtbar)

B Transparenz aus

p Bioneuronen in einer Ebenen anordnen

u Bioneuronen sphärisch anordnen

j Bioneuronen in einem kubischen Gitter anordnen

Steuerung der Simulation

c Simulation starten

C Simulation stoppen

e Synapsen während der Simulation anzeigen

E Synapsen während der Simulation ausblenden

1 Zeitschritte 0.1 ms

2 Zeitschritte 1 ms

3 Zeitschritte 10 ms

4 Zeitschritte 100 ms

→ Einen Zeitschritt in der Simulation vorgehen

← Einen Zeitschritt in der Simulation zurückgehen

Tab Verwendete Simulationsdatei ausgeben

Weitere Tastenkürzel

˜ Konsole öffnen

q Visualisierung beenden

F Fenster maximieren

g Fenster zurück auf ursprüngliche Grösse skalieren

Befehle der Konsole

set simfile ⟨**Datei**⟩ Lädt die angegebene Datei für die Simulation

set simtime $\langle$**Zeit**$\frac{1}{10}$**ms**$\rangle$ Springt in der Simulation zu der angegebenen Zeitmarke

set neuronid ⟨**ID**⟩ Wählt das Bioneuron mit der angegebenen ID aus

set timeinterval ⟨**Intervalllänge**⟩ Setzt die Länge des betrachteten Zeitintervalls in der Simulation (in Zeitschritten)

set anaglyph Stereodarstellung einschalten

unset anaglyph Stereodarstellung auschalten

set param Parametersätze darstellen

unset param Parametersätze ausblenden

set hwmodel Hardwaremodell darstellen

unset hwmodel Hardwaremodell ausblenden

Bibliography

[1] *ATI Technologies ULC Website:*. `http://www.ati.com`.

[2] *Microsoft DirectX Developer Center Website:*. `http://msdn.microsoft.com/directx`.

[3] Fast Analog Computing with Emergent Transient States (FACETS): *Homepage*. http://www.facets-project.org.

[4] *The freeglut Project:*. `http://freeglut.sourceforge.net/`.

[5] Initiative, The Neuralensemble: *PyNN Website*. http://neuralensemble.org/PyNN.

[6] *NVidia Corp. Website:*. `http://www.nvidia.com`.

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

[8] *The OpenGL Programming Guide - The Redbook:*. `http://www.glprogramming.com/red/`.

[9] Schemmel, J., D. Brüderle, K. Meier B. Ostendorf: *Modeling Synaptic Plasticity within Networks of Highly Accelerated I&F Neurons. Proceedings of the 2007 IEEE International Symposium on Circuits and Systems (ISCAS'07)*. IEEE Press, 2007.

[10] Schemmel, J., A. Grübl, K. Meier E. Mueller: *Implementing Synaptic Plasticity in a VLSI Spiking Neural Network Model. Proceedings of the 2006 International Joint Conference on Neural Networks (IJCNN'06)*. IEEE Press, 2006.

[11] *Silicon Graphics, Inc. Standard Template Library Website:*. `http://www.sgi.com/tech/stl/`.

[12] The Neural Simulation Technology (NEST) Initiative: *Website*. http://www.nest-initiative.org, 2008.

Reference Manual

Generated by Doxygen 1.5.4

Wed Jul 23 21:30:56 2008

Contents

Chapter 1

Class Index

1.1 Class List

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

Chapter 2

Class Documentation

2.1 bio_neuron Struct Reference

struct that contains all the basic information needed for fast rendering

`#include <GraphVisu.h>`

Public Attributes

- float **pos** [3]
  position of the neuron in space
- std::map< unsigned int, **bio_synapse** > **out_targets**
  targets of the outgoing synapses
- GraphModel::GMNode * **real_node**
  corresponding node in the biomodel

2.1.1 Detailed Description

struct that contains all the basic information needed for fast rendering

The documentation for this struct was generated from the following file:

- GraphVisu.h

2.2 bio_synapse Struct Reference

struct for the synapses

`#include <GraphVisu.h>`

Public Attributes

- float **target_pos** [3]
  position of the target
- float **weight**
  weight of the synapse
- GraphModel::GMNode * **target_node**
  pointer to the real node

2.2.1 Detailed Description

struct for the synapses

The documentation for this struct was generated from the following file:

- GraphVisu.h

2.3 glControl Class Reference

`#include <glControl.h>`

Public Member Functions

- **glControl** (BioModel *_BioModel, **GraphVisu** *graph_ptr)
  constructor
- void **window** (char *caption, int pos_x, int pos_y)
  Window generation.
- void **init** (char mode)
  Init the glParameters.
- void **reshape** (int width, int height)
- void **create_menu** ()
- void **set_ortho** ()
  set a orthogonal projection for rendering
- void **set_perspective** ()
  set a perspective projection for rendering
- void **display** ()
  display functions
- void **CreateDisplayLists** ()
  create display lists for better performance (not needed yet)
- void **keystroke** (unsigned char key, int x, int y)
  Callback functions provided by glut.
- void **specialkeys** (int key, int x, int y)
  keyboard callbacks
- void **active_mousemove** (int x, int y)
- void **passive_mousemove** (int x, int y)
- void **mouse** (int button, int state, int x, int y)
- void **menu** (int entry)
  callbackfunction for menuentries
- void **StartPicking** ()
  initiate the picking process
- void **StopPicking** ()
  stop the picking process and determine the selected object
- void **show_terminal** ()
  show or hide the terminal

- void **parse_cmd** ()

  commandline parser

- void **write_text** (char *text, float x, float y, float z)

  write text to the given position

- void **write_3dtext** (char *text)

  write 3d text in space in order to label the cores etc

- void **show_spikes** ()

  animation function that increases the timestep by 0.1 and redraws the display showing spiking neurons

Public Attributes

- BioModel * **bio**

  pointer to the the biomodel

- int **displaylists** [32]

  contains displaylist numbers

- float **sim_time**

  simulation time

- GLuint **w**

  Initial Functions.

- GLuint **h**

  window size parameters

Private Attributes

- float **trans** [3]

  translation of the scene

- float **rot** [3]

  rotation of the scene

- float **mouse_pos** [2]

  mouse position

- float **light0_position** [3]

  position of lightsource in space

- float **light0_diffuse** [4]

  diffuse light component of LIGHT0

- float **light0_ambient** [4]
  ambient light component of LIGHT0
- unsigned int **SelNeuron**
  offset for synapse selection
- unsigned int **SelSynPattern**
  value of how many neurons are skipped
- char **status** [255]
  for writing status values
- std::string **cmd**
- int **window_id**
  id of the created window
- bool **terminal**
  is the terminal open
- **GraphVisu** * **_Graph**
  class for setting the scene with all the desired neurons

2.3.1 Detailed Description

Class for setting up an OpenGL window, controlling the callbacks i.e inputs from the keyboard, mouse, resizing events and preparing everything needed for drawing a scene

2.3.2 Constructor & Destructor Documentation

2.3.2.1 glControl::glControl (BioModel * *_BioModel*, GraphVisu * *graph_ptr*)

constructor

Parameters:

_BioModel pointer to the biomodel
graph_ptr pointer to calling **GraphVisu** (p. 10) class

2.3.3 Member Function Documentation

2.3.3.1 void glControl::window (char * *caption*, int *pos_x*, int *pos_y*)

Window generation.

Parameters:

caption set caption of the window
pos_x x position of window
pos_y y position of window

2.3.3.2 void glControl::init (char *mode*)

Init the glParameters.

Parameters:

mode glParameters (with depthtest, smooth shading etc.)

2.3.3.3 void glControl::reshape (int *width*, int *height*)

reshape function for controlling resizing of the window without Viewport transformation objects might appear distorted

Parameters:

width callback value of resize for new width

height callback value of resize for new height

2.3.3.4 void glControl::display ()

display functions

arrange the content of the window and call the routine displaying the neurons

handle the display

2.3.3.5 void glControl::keystroke (unsigned char *key*, int *x*, int *y*)

Callback functions provided by glut.

Parameters:

key pressed key

x mouseposition x

y mouseposition y

2.3.3.6 void glControl::specialkeys (int *key*, int *x*, int *y*)

keyboard callbacks

Parameters:

key keycode

x mouseposition x

y mouseposition y

2.3.3.7 void glControl::menu (int *entry*)

callbackfunction for menuentries

Parameters:

entry callback value of menu entry

2.3.3.8 void glControl::write_text (char * *text*, float *x*, float *y*, float *z*)

write text to the given position

Parameters:

text text to write

z where to write the text

2.3.3.9 void glControl::write_3dtext (char * *text*)

write 3d text in space in order to label the cores etc

Parameters:

text text to write

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

- glControl.h
- glControl.cpp

2.4 GraphVisu Class Reference

`#include <GraphVisu.h>`

Public Member Functions

- **GraphVisu** (BioModel *_BioModel, HWModel *_HWModel)
  constructor
- **~GraphVisu** ()
  destructor
- void **reset_callbacks** ()
  reassign the callback functions when the mode has changed
- void **Bio_PrintPositions** ()
  prints out the positioning of the neurons in space for debugging purposes
- void **Bio_SetPositions** ()
  sets arbitrary positions in space for neurons
- void **Bio_SetSpherePositions** ()
  arrange the neurons randomly within a sphere
- void **Bio_SetPlanePositions** ()
  arrange the neurons on a plane
- void **draw_BioModel** ()
  draw the Biomodel
- void **SetHWNodePositions** ()
  arrange the positions of the neurons in the HWModel
- void **draw_HWModel** ()
  draw the HWModel
- void **SetTargetToMap** (GraphModel::GMNode *target)
  add the target of the selected neuron in the biosystem to the maplist
- void **draw_MappingOfSelNeuron** (basics::string<> n_name)
  show the mapping of the selected neuron
- void **draw_SpikeyCoreConnections** (int core)
  draw the connections between the nodes on one spikeycore
- void **set_scene** ()
  set whatever we want to be displayed
- int **getNeuronID** (basics::string<> NeuronName)

returns the id of the node

- bool **NeuronIsTarget** (int id, short int *targets, int count)
  check if the neuron is marked as a target of an outgoing connection
- void **ReadParameterNode** (GraphModel::GMNode *Node)
- void **target_hwneuron** (unsigned int name)
  Decode and mark the selected neuron in the hwmodel for further processing.
- void **print_status** ()
  print the current status information
- void **display_parameters** ()
  display the parameterset of the selected neuron in the HWModel
- int **read_spikes** (const char *filename)
  read the spiking information from the given file. returns 0 if successfull
- void **print_simulation** ()
- void **transfer_biograph** ()
  copy all the needed information from the biograph into a map for better performance
- unsigned int **bio_map_size** ()
  returns the size of the bio_map
- void **gen_targetlist** (unsigned int neuronid)
  generates the targetlist when the selection has changed reducing overhead when drawing

Public Attributes

- unsigned int **SelNeuron**
  offset for synapse selection
- pthread_t **glThread**
  pointer to the current visualisation thread for pthread_join();
- **glControl** * **control**
  pointer to the glControler
- BioModel * **bio**
  pointer to the biomodel
- HWModel * **hw**
  pointer to the hardwaremodel
- GraphModel::GMNode * **MappingTarget**
  pointer to the mapping target of the currently selected bioneuron
- basics::string **MapTargetName**

name of a mapping target

- basics::string **MapTargetFrameName**
  name of the frame of the mapping target
- std::map< int, set< int > > **spike_events**
  contains the recorded spike times and the neurons where the events took place
- std::map< int, set< int > >::iterator **map_iterator1**
  iterators for the maps
- GraphModel::GMNode * **hwneuron_selection**
  pointer to the selected HW Neuron
- GraphModel::VectorGMNode **MapTTName**
  list of mapping targets of neurons connected to the selected Neuron in the BioSystem
- bool **with_mapping_selected**
  is the mapping of the selected neuron to be shown?
- bool **with_biosystem_conn**
  are the connections in the biosystem to be displayed?
- bool **with_hw_parameters**
  are the hw_parameters to be displayed?
- bool **with_core_names**
  show or hide the names of the spikeycores
- bool **with_spiking_animation**
  enable, disable animation
- bool **with_hwmodel**
  show, hide the hw model
- bool **with_animation_synapse**
  show the outgoing synapses of a spiking neuron
- bool **with_bio_parameters**
  show, hide the parameters of the biomodel neurons
- bool **anaglyph**
  use color filter for 3D
- std::string **sim_filename**
  filename of the simulation
- int **timestart**
  simulation timepoint

- int **timeinterval**

  spike-events will be displayed in the interval [timestart;timestart+timeinterval]

- int **time_resolution**

  size of the timesteps from 0.1 ms to seconds

- unsigned int **s_buffer** [1024]

Private Attributes

- char **status** [255]

  contains status informations

- std::map< int, set< int > >::iterator **map_iterator2**
- std::map< unsigned int, **bio_neuron** > **bio_map**

  map containing the need information extracted from the bio graph <neuronid, parameterset>. improves performance

- std::set< unsigned int > **targetlist**
- float **biomodelOrigin** [3]

  origin of the bio graph

- float **hwmodelOrigin** [3]

  origin of the hardware graph

- float **max_weight**
- float **min_weight**

  maximum and minimum value of the synaptic weights within the biosystem

2.4.1 Detailed Description

Management of the 3D scene that will be rendered with OpenGL contains pointer to the bio and hardware model as well as control flags for showing or hiding different parts of visualisation provides functions for drawing the biomodel and hardwaremodel, the connections between the neurons within the biomodel as well as the mapping of the biomodel neurons onto the hardwaremodel. another function reads in a simulation file and starts an animation showing the spiking events.

2.4.2 Constructor & Destructor Documentation

2.4.2.1 GraphVisu::GraphVisu (BioModel * *_BioModel*, HWModel * *_HWModel*)

constructor

Parameters:

_BioModel pointer to the biomodel

_HWModel pointer to the hwmodel

2.4.3 Member Function Documentation

2.4.3.1 void GraphVisu::Bio_SetPositions ()

sets arbitrary positions in space for neurons

arrange the neurons in a cube

2.4.3.2 void GraphVisu::SetTargetToMap (GraphModel::GMNode * *target*)

add the target of the selected neuron in the biosystem to the maplist

Parameters:

target target of source node in biosystem

2.4.3.3 void GraphVisu::draw_MappingOfSelNeuron (basics::string<> *n_name*)

show the mapping of the selected neuron

Parameters:

n_name Name of a Neuron in the BioSystem

2.4.3.4 void GraphVisu::draw_SpikeyCoreConnections (int *core*)

draw the connections between the nodes on one spikeycore

Parameters:

core number of core to select

2.4.3.5 int GraphVisu::getNeuronID (basics::string<> *NeuronName*)

returns the id of the node

Parameters:

NeuronName name identifier of the node

2.4.3.6 bool GraphVisu::NeuronIsTarget (int *id*, short int * *targets*, int *count*)

check if the neuron is marked as a target of an outgoing connection

Parameters:

id id of Neuron

targets list of target ids

count number of targets in the list

2.4.3.7 void GraphVisu::ReadParameterNode (GraphModel::GMNode * *Node*)

Read the names of the parameter node and the corresponding value node will write the names of the parameter node and the value node of GMNode* Node to char status[255]

Parameters:

Node source node

2.4.3.8 void GraphVisu::target_hwneuron (unsigned int *name*)

Decode and mark the selected neuron in the hwmodel for further processing.

Parameters:

name encoded name in the selection buffer Coding: (SpikeyCoreNumber + 1) * 1000 + (NeuronNumber + 1)

2.4.3.9 int GraphVisu::read_spikes (const char * *filename*)

read the spiking information from the given file. returns 0 if successfull

Parameters:

filename file containing the spiking informations

2.4.3.10 void GraphVisu::gen_targetlist (unsigned int *neuronid*)

generates the targetlist when the selection has changed reducing overhead when drawing

Parameters:

neuronid id of the selected neuron

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

- GraphVisu.h
- GraphVisu.cpp

Index