Programmierdokumentation von VNNlab

Norman Feske

29. Januar 2002

Inhaltsverzeichnis

1 Einführung

Diese Dokumentation soll einen Einblick in die wesentlichen programmiertechnischen Aspekte des Programmes 'VNNlab' geben. Es wird auf grundsätzliche Mechanismen und Datenstrukturen eingegangen. Allerdings wird auf die Darstellung von Algorithmen verzichtet. Diese Dokumentation gibt Programmierern den nötigen Einblick, um selbst Module für 'VNNlab' zu programmieren oder 'VNNlab' um neue Funktionen zu erweitern.

2 Konzept

Der Entwicklung von 'VNNlab' liegen einige Grundgedanken zu Grunde:

- **Nutzung von Standards:** Das Programmsystem wurde auf etablierten Standards der Unix-Welt aufgebaut. Dazu gehören 'Shared Objects', 'Posix-Threads', 'OpenGL' sowie 'Tcl/Tk'.

- **Kapselung von Funktionalität:** Es wurde bei der Strukturierung des Programmsystems darauf Wert gelegt, daß seine Komponenten möglichst allgemein verwendbar und in sich geschlossen sind. Dieser Gedanke wird vor allem durch die Verwendung von Modulen unterstrichen. Durch den übersichtlichen und modularen Aufbau des Programmpaketes soll eine einfache Wiederverwendbarkeit von Programmteilen ermöglicht werden.

- **Ergonomie:** Da das Programmsystem als Lernbeispiel Verwendung finden soll, wurde auf die Benutzeroberfläche großen Wert gelegt. Sie sollte eine weitestgehend intuitive Bedienung des Programmes für Leute gestatten, die sich mit dem behandelten Problembereich beschäftigen. Dabei sollte die Benutzeroberfläche stets bedienbar bleiben und nicht etwa durch lang laufende Prozesse blockiert werden.

- **Effizienz:** Große Teile von 'VNNlab' wurden mit der Zielsetzung programmiert, vorhandene Hardware möglichst effizient zu nutzen. Dazu gehört die Nutzung eventuell vorhandener 3D-Hardware durch OpenGL, aber auch der ressourcenschonende Umgang mit Rechenzeit und Speicher in Berechnungsprozessen.

2.1 Tcl/Tk

Für die grafische Oberfläche von 'VNNlab' wurde Tcl/Tk verwendet. Hierfür wurde 'wish' (die Tcl/Tk-Shell) in das Hauptprogramm gelinkt und um einige Befehle erweitert. Zur Initialisierung der Tcl/Tk-Shell dient die Funktion `Tcl_AppInit`. In ihr wird die Shell um zusätzliche Befehle erweitert und das Tcl/Tk-Script festgelegt, welches die Shell ausführen soll. Eine solche Initialisierungsfunktion hat folgende Gestalt:

```
int Tcl_AppInit(Tcl_Interp *interpreter) {

        /* check argument and init tk */
        if (Tcl_Init(interp) == TCL_ERROR) return TCL_ERROR;
        Tk_Init(interp);

        /* create new tcl commands */
       /* the name of the new command is 'NewCmd' */
       /* the corresponding C function is 'TCL_NewCmd' */
        Tcl_CreateCommand(interp,"NewCmd",TCL_NewCmd,NULL,NULL);

       /* tell the interpreter the name of the script to execute */
        Tcl_SetVar(interp, "tcl_rcFileName", "gui.tcl",TCL_GLOBAL_ONLY);
        return TCL_OK;
}
```

Die nun mit dem neuen Tcl-Befehl assoziierte C-Funktion sieht wie folgt aus:

```
int TCL_NewCmd(ClientData cd, Tcl_Interp *interp, int argc, char *argv[]) {
       /* do something... */
        return TCL_OK;
}
```

Dabei enthält das Array `'argv'` die Argumente des Funktionsaufrufes. Das Element `argv[0]` beinhaltet den Namen des Kommandos. Alle weiteren Elemente enthalten Zeiger auf die einzelnen Parameter. (ähnlich wie die Parameterübergabe an die `main()`-Funktion eines ANSI-C-Programmes)

Die Tcl/Tk-Shell wird mit der Tcl/Tk-Library-Funktion `Tk_Main(1, tclargv, Tcl_AppInit)` gestartet. Ihr wird die oben beschriebene Initialisierungsfunktion übergeben. `Tk_Main` sollte am Ende der `main()`-Funktion des Hauptprogrammes ausgeführt werden. Nach dem Aufruf startet die Tcl/Tk-Shell seine Hauptschleife (eventloop) und erhält die Kontrolle über das Programm.

2.2 Shared Objects

Die Verwendung von 'Shared Objects' erlaubt die Modularisierung von Programmen. Ein 'Shared Object' ist ein kleiner Programmteil, der erst bei Bedarf geladen wird. Der Zugriff aus 'Shared Objects' kann entweder über dynamisches Linken oder über die in der Header-Datei `<dlfcn.h>` definierten Systemaufrufe geschehen. In 'VNNlab' findet die zweite Möglichkeit Verwendung. Sie hat den Vorteil, daß das Hauptprogramm gezielt festlegen kann, wann mit welchen 'Shared Objects' gearbeitet wird.

2.2.1 Erzeugen von Shared Objects

Um aus einer C-Datei ein 'Shared Object' zu erzeugen, muß dem Compiler/Linker die Option `-shared` angegeben und der Name des 'Shared Objects' als Option `-soname` angegeben werden. Ein Makefile zum Erzeugen eines 'Shared Object' aus der Datei `'beispiel.c'` könnte so aussehen:

```
SOURCES = beispiel.c
INCDIR  = -I /usr/share/src/OpenGL/toolkits/include -I ../../

beispiel: $(SOURCES)
        cc  -fPIC -shared $(INCDIR) $(SOURCES) -Wl,-soname,$@.so -o $@.so

clean:
        rm -f *~ *.o *.so
```

2.2.2 Zugriff auf Shared Objects

Zunächst muß das 'Shared Object' mit dem Systemaufruf:

```
void *dlopen (const char *filename, int flag);
```

geöffnet werden. Dabei muß der entsprechende Dateiname und eine Bit-Maske von 'flags' angegeben werden. Im allgemeinen genügt die Angabe von `RTLD_LAZY` als Parameter für `'flag'`. Es wird ein 'handle' für das geöffnete 'Shared Object' zurückgeliefert (so_handle), mit dem auf das 'Shared Object' zugegriffen werden kann. Dieser Zugriff erfolgt stets über den Systemaufruf:

```
void *dlsym(void *so_handle, char *symbol);
```

Dieser liefert die (Speicher-)Adresse eines Symbols mit dem Namen `symbol` innerhalb des 'Shared Objects' `so_handle` zurück. Es werden keine Informationen über die Art des Symbols geliefert (Variable oder Funktion? Typ einer Variable?). Diese Informationen müssen für die Arbeit mit 'Shared Objects' klar definiert sein. Die in 'VNNlab' verwendeten Modulschnittstellen sind in Kapitel 5 beschrieben. Die Adresse des Symbols muß dann in den Typ des Symbols, auf das zugegriffen werden soll, 'gecastet' werden. Dies könnte so aussehen:

```
unsigned long *lvar;
lvar = (unsigned long *)dlsym(so_handle,”Iteration”);
printf(“Inhalt der Variable 'Iteration': %lu”,*lvar);
```

Nach der Benutzung sollte ein mit `'dl_open'` geöffnetes 'Shared Object' mit `'dl_close'` geschlossen werden:

```
int dlclose (void *handle);
```

Bei der Arbeit mit 'Shared Objects' können Fehler auftreten. Beispielsweise kann mit 'dlsym' nach einem Symbol gefragt werden, was im 'Shared Object' nicht definiert ist. Diese Fehler lassen sich mit dem Systemaufruf

```
const char *dlerror(void);
```

abfragen. Diese Funktion gibt einen String mit einer entsprechenden Fehlermeldung zurück. Weitere Informationen zu 'Shared Objects' findet man in den 'man-pages' von `'dlopen'`.

Zur Nutzung der hier beschriebenen Systemaufrufe muß das Programm mit der Bibliothek `'libdl'` (Linkeroption `'-ldl'`) gelinkt werden.

2.3 OpenGL

Für sämtliche grafische Ausgaben von 'VNNlab' findet 'OpenGL' in Verbindung mit 'glut' Verwendung. Dabei stellt 'glut' die Verbindung zwischen 'OpenGL' und dem System her. Ein typisches 'glut' Programm besteht aus einer Reihe von 'callback'-Funktionen und einer kurzen `main()`-Funktion, die lediglich Initialisierungen durchführt und danach die `'glutMainLoop'` startet. Die Initialisierungen beinhalten das Setzen von 'OpenGL'-Attributen und das Definieren von Ausgabefenstern (`'glutInitWindowSize()'`, `'glutCreateWindow()'`). Diese Initialisierungen sind für 'VNNlab' in der Datei `'handle_gl.c'` zu finden. 'VNNlab' benötigt zwei Ausgabefenster:

- Fenster zur Anzeige der Verteilung und des Netzwerkes,
- Fenster zur Anzeige des Visualisierungsergebnisses.

Für jedes Fenster können entsprechende 'callback'-Funktionen definiert werden. Dazu gehören `'redraw'` und `'reshape'`. Diese Funktionen werden immer dann ausgeführt, wenn das entsprechende Fenster neu gezeichnet werden muß oder seine Größe ändert. Desweiteren wird die 'callback'-Funktion `'idle'` definiert, die immer dann aufgerufen wird, wenn das Programm nichts zu tun hat. Diese Funktion findet vor allem in der Synchronisation von Threads ihre Bedeutung.

2.4 Multithreading - mehrpfädiger Programmablauf

Sowohl 'Tcl/Tk' als auch 'glut' benötigen jeweils eine eigene Hauptschleife (eventloop). Zusätzlich soll 'VNNlab' länger andauernde Berechnungen ausführen. Dies ist nur unter der Benutzung von 'Threads' möglich. In 'VNNlab' finden 'Posix-Threads' Verwendung. Diese stellen in der Unix-Welt ein Standard dar. Die Erzeugung eines Threads erfolgt über den Systemaufruf `'pthread_create'` , z.B.:

```
pthread_create(&threadinfo,NULL,&thread_function,NULL);
```

Dabei ist `'threadinfo'` eine Struktur vom Typ `'pthread_t'` und enthält nach dem Erzeugen des Threads seine Statusinformationen. Nach dem Erzeugen des Threads startet dieser mit der angegebenen Funktion `'thread_function'`. Eine solche Funktion kann folgendes Aussehen haben:

```
void *thread_function(void *arg) {
    /* so something... */
   return NULL;
}
```

Ihr wird als Argument der Pointer übergeben, der beim Aufruf von `'pthread_create'` als vierter Parameter angegeben wird. So können z.B. mehrerer Threads zur parallelen Verarbeitung verschiedener Bilder gestartet werden.

Zur Nutzung von 'Posix-Threads' muß die Bibliothek `'libpthread'` mit der Linker-Option `'-lpthread'` zum Hauptprogramm gelinkt werden. Die nötigen Definitionen und Prototypen für die Verwendung von 'Posix-Threads' sind in der Header-Datei `pthread.h` zu finden.

2.5 Dateien

2.5.1 Sourcecodes

Datei	Funktion
`main.c`	Das Hauptprogramm
`gui.tcl`	Tcl/Tk-Script für die grafische Benutzeroberfläche des Hauptprogrammes
`diagram.tcl`	Tcl/Tk-Script zur Erzeugung des Diagrammes im Hauptfenster
`console.tcl`	Tcl/Tk-Script für die Console (für Statusausgaben)
`handle_gl.c`	OpenGL-Initialisierungscode
`mesh.c`	allgemeine Funktionen zum Umgang mit 'mesh'-Daten
`image.c`	allgemeine Funktionen zum Umgang mit Bildern
`Makefile`	Makefile zur Erzeugung des Hauptprogrammes

Die Dateien `diagram.tcl` und `console.tcl` sind aus Gründen der Übersichtlichkeit nicht in `gui.tcl` enthalten. Sie werden mit dem Tcl-Kommando `'source'` an den entsprechenden Stellen in gui.tcl nachgeladen.

2.5.2 Header-Files

Datei	Funktion
`distrib.h`	Definition der Datenstrukturen von Verteilungen
`image.h`	Definitionen der Datenstruturen von Bildern und Prototypen der Funktionen von `image.c`
`mesh.h`	Definitionen der Datenstrukturen von Bildern und Prototypen der Funktionen von `mesh.c`

2.5.3 Unterverzeichnisse

Verzeichnis	Funktion
networks/	enthält die Module der Netzwerke
distributions/	enthält die Module zum Erzeugen von diskreten Verteilungen
depthevals/	enthält die Module zur Korrespondenzanalyse
visualisations/	enthält die Visualisierungsmodule

Die Modulverzeichnisse enthalten wiederum pro Modul ein Unterverzeichnis. Der Name des Unterverzeichnisses ist gleichzeitig der Modulname. In diesem Unterverzeichnis müssen sich mindestens die Dateien `<Modulname>.so` und `<Modulname>.tcl` befinden. Dabei beinhaltet das `so`-File (Shared Object) die eigentliche Funktionalität des Moduls und das `tcl`-File die Definition des Einstellungsdialogs des Modules.

3 Datenstrukturen

Innerhalb von 'VNNlab' und seinen Modulen werden einige Datenstrukturen häufig verwendet. Im Folgenden erfolgt eine kurze Beschreibung dieser Datenstrukturen und eventuell vorhandener Funktionen, mit denen auf diese Datenstrukturen zugegriffen wird.

3.1 Mesh

Für den Austausch von Daten zwischen den Verarbeitungsschritten 'Netzwerk' und 'Visualisierung' wird die Datenstruktur 'mesh' verwendet. Sie ist wie folgt definiert:

```
struct mesh_point {
        float x,y,z;            /* point coordinates */
        float nx,ny,nz;         /* normal vector of point */
};
typedef struct mesh_point mesh_point;

/* we are only using triangles... */
#define MAX_POLYPOINTS 3
struct mesh_poly {
        long num_points;        /* number of polygon points */
        long point_idx[MAX_POLYPOINTS];
                                /* array with polygon point indices */
        float nx,ny,nz;         /* normal vector of polygon */
};
typedef struct mesh_poly mesh_poly;

struct mesh {
        long num_points;        /* number of mesh points */
        mesh_point *points;     /* pointer to point array */
        long num_polys;         /* number of polygons */
        mesh_poly *polys;       /* pointer to polygon array */
};
typedef struct mesh mesh;
```

Das Gitter 'mesh' besteht aus einem Array von Punkten und einem Array von Polygonen. Ein Punkt besteht aus seinen Koordinaten und seinem Normalenvektor. Dieser Normalenvektor kann zur Beleuchtung der Darstellung des Polygonnetzes genutzt werden. Die Berechnung dieses Vektors geschieht durch die Aufsummierung der Normalenvektoren der angrenzenden Polygone.

Ein Polygon besteht aus einem Array von Punkten. Die Größe dieses Arrays ist auf 3 festgelegt, da nur mit Dreiecken gearbeitet wird. Desweiteren wird pro Polygon sein Normalenvektor gespeichert.

3.2 Image

Die Datenstruktur `'image'` wird von einzelnen Modulen von 'VNNlab' verwendet um Bilddaten zu verwalten. Die Datenstruktur eines Bildes ist wie folgt definiert:

```
#define IMAGE_RGBA_CHAR 1
#define IMAGE_GRAY_FLOAT 2

struct image_struct {
        long w;
        long h;
        long format;
        void *pixels;
};

typedef struct image_struct image;
```

Es werden bislang zwei Formen der Organisation der Pixel (Formate) unterschieden:

- IMAGE_RGBA_CHAR (Echtfarbbilder im RGBA-Format): Jedes Pixel besteht aus den vier Komponenten rot, grün, blau und alpha. Jede Komponente umfaßt ein Byte (unsigned char). Die alpha (Transparenz) Komponente wird zur Zeit von keinem Modul verwendet.
- IMAGE_GRAY_FLOAT (Grauwertbilder): Jedes Pixel besteht aus einem Grauwert, der als 32bit Fließkommazahl abgelegt wird (float).

Zum Einlesen und Verwalten von Bildern steht den Modulen die Datei IMAGE.C zur Verfügung. In ihr sind einige hilfreiche Funktionen zur Behandlung von Bildern implementiert:

```
extern  image   *Create_Image(long w,long h,int format);
```

Es wird ein Bild des angegebenen Formates im Speicher erzeugt und ein Zeiger auf dessen Struktur zurückgeliefert.

```
extern  image   *Load_RGB_Image(char *filename,int format);
```

Hiermit kann ein 'RGB'-Bild (SGI Rasterformat) eingelesen werden. Bislang werden nur unkomprimierte RGB-Bilder unterstützt. Achtung: xv speichert 'RGB'-Bilder immer 'rle'-komprimiert (run length encoding) ab. Unkomprimierte Bilder können beispielsweise mit 'Gimp' erzeugt werden. Als Parameter wird der Funktion der Dateiname des Bildes und das gewünschte Pixelformat angegeben. Im Fehlerfall wird ein Nullzeiger zurückgeliefert.

```
extern  void    Destroy_Image(image *img);
```

Diese Funktion entfernt (deallocate) ein Bild aus dem Speicher.

```
extern  image   *Image_float2rgba (image *src,image *dst);
```

Ein Bild im `IMAGE_GRAY_FLOAT` Format kann mit dieser Funktion in ein Bild des `IMAGE_RGBA_CHAR` Formates umgewandelt werden.

3.3 Distribution

Die von einem 'Verteilungsmodul' erzeugte Verteilung hat eine definierte Struktur. Diese Struktur wird verwendet um eine Verteilung vom 'Verteilungsmodul' an ein 'Netzwerkmodul' zu übergeben. Sie ist wie folgt definiert:

```
struct dist_point {
        float x,y;
};

typedef struct dist_point dist_point;

struct distribution {
        long num_points;
        dist_point *points;
};

typedef struct distribution distribution;
```

Dabei besteht eine Verteilung hauptsächlich aus einem Array von zweidimensionalen Punkten. Das Array hat die Größe `'num_points'`.

4 Erweiterungen der Tcl/Tk-Shell

Um die Kommunikation zwischen dem C-Programm und dem Tcl/Tk-Script zu ermöglichen, wurde die Tcl/Tk-Shell um einige Funktionen erweitert. Es wird zwischen allgemeinen Funktionen und Funktionen, die speziell für den hier behandelten Problembereich implementiert wurden, unterschieden.

4.1 Allgemeine Funktionen

Die allgemein verwendbaren Funktionen beziehen sich vor allem auf den Umgang mit 'Shared Objects'. Sie ermöglichen dem Tcl/Tk-Script, 'Shared Objects' zu öffnen, Adressen von Symbolen abzufragen und auf Zeichenketten und Variablen innerhalb der 'Shared Objects' zuzugreifen.

- `dlOpen <filename>`: Es wird ein 'Shared Object' geöffnet und ein 'handle' zum späteren Zugriff auf dieses 'Shared Object' zurückgeliefert.

- `dlClose <so_handle>`: Das 'Shared Object' mit dem angegebenen 'handle' wird geschlossen.

- `dlGetString <so_handle> <identifier>`: Es wird der Inhalt eines Strings mit dem Variablennamen `<identifier>` des 'Shared Objects' mit dem 'handle' `<so_handle>` zurückgeliefert. Diese Funktion wird beispielsweise verwendet, um den Titel eines Modules zu erfragen. Dieser wird innerhalb eines Modules immer in einem String mit dem Namen `'title'` abgelegt. Wenn das Symbol `<identifier>` nicht innerhalb des 'Shared Objects' definiert ist, wird der String `'not defined'` zurückgeliefert.

- `dlGetAdress <so_handle> <identifier>`: Diese Funktion liefert die Adresse eines Symbols `<identifier>` zurück. Dies kann z.B. die Adresse einer Variable oder Funktion sein.

- `dlPeek <pointer> <type>`: Mit dieser Funktion ist es möglich, den Inhalt einer Speicherzelle an Adresse `<pointer>` zu erfragen. Dabei muß der Variablentyp, auf den zugegriffen wird, angegeben werden. Es werden die Typen `'long'`, `'float'` und `'string'` unterstützt.

- `dlPoke <pointer> <type> <value>`: Hiermit ist es möglich, eine Variable des Typs `<type>` an der Adresse `<pointer>` auf einen neuen Wert `<value>` zu setzen. Diese Funktion wird zur Zeit nicht von 'VNNlab' verwendet.

4.2 Spezielle Funktionen

Die folgenden Funktionen beziehen sich auf das konkrete Programm 'VNN-lab' und dessen Funktionalität. Sie sollten nicht von Modulen (innerhalb ihrer tcl/tk-Scripte) verwendet werden, außer wenn darauf ausdrücklich hingewiesen wird. Die meisten Funktionen sind lediglich als 'Events' zu verstehen, die das Tcl/Tk-Script an das C-Programm sendet.

- `InitDistribution:` Diese Funktion veranlaßt die Initialisierung einer Verteilung mit dem aktuellen Verteilungs-Modul.

- `SetModule <modtype> <so_handle>:` Mit dieser Funktion wird ein neues Modul des Modultyps `<modtype>` gesetzt. Gültige Modultypen sind: `'net'` (Netzwerk), `'dist'` (Verteilung), `'depth'` (Tiefenschätzung) und `'vis'` (Visualisierung). Der Funktion muß das 'handle' eines entsprechenden 'Shared Objects' übergeben werden. Ein solches 'handle' wird von der Funktion `dlOpen` zurückgeliefert.

- `InitNetwork:` Hiermit wird die Initialisierung eines Netzes mit den eingestellten Parametern veranlaßt. Insbesondere wird die Netzwerkmodul-Funktion `'Create_Network'` aufgerufen und das alte Netzwerk mit der Netzwerkmodul-Funktion `'Destroy_Network'` gelöscht.

- `BeginLearning:` Diese Funktion setzt den Lernvorgang des Netzwerkes in Gang. Dies umfaßt auch die regelmäßige Aktualisierung des Netzwerk-Ausgabefensters.

- `StopLearning:` Hiermit wird der Lernvorgang des Netzwerkes angehalten. Die Funktion kann unter Umständen nicht sofort reagieren, da der aktuelle Netzwerk-Berechnungsschritt noch zu Ende ausgeführt wird.

- `RedrawNetwork:` Das Darstellungsfenster für das Netzwerk und die Verteilung wird hiermit neu gezeichnet. Es werden keine Berechnungen, sondern lediglich die entsprechenden 'redraw'-Funktionen ausgeführt. Vor dem Redraw findet eine Synchronisation der Variablen des Netzwerk-Modules statt.

- `RedrawDistribution:` Diese Funktion veranlaßt wie `'RedrawNetwork'` das Neuzeichnen des Netzwerk- und Verteilungsfensters. Der Unterschied zu `'RedrawNetwork'` ist, daß hier die Variablen des Verteilungsmodules synchronisiert werden.

- `InitDepthEval:` Hiermit wird 'VNNlab' veranlaßt, die Funktion `'Init_-DepthEval'` im aktuellen Korrespondenzanalyse-Modul aufzurufen.

- `RedrawVisualisation:` Der Aufruf dieser Funktion bewirkt ein Neuzeichnen des Visualisierungs-Fensters. Insbesondere wird hierbei die Funktion `'Redraw_Visualisation'` des aktuellen Visualisierungsmodules aufgerufen. Diese Funktion sollte vom Tcl/Tk-Script eines Visualisierungsmodules aufgerufen werden, wenn sich die Parameter der Visualisierung verändern. Dies macht beispielsweise ein interaktives Verändern der Betrachtungswinkel im Modul 'mesh3d' möglich.

- `UpdateVisualisation:` Diese Funktion bewirkt eine Neuberechnung des Polygonnetzes, der Triangulation und Tiefenschätzung. Hiermit wird eventuell ein lang laufender Prozeß in Gang gesetzt (abhängig von den verwendeten Algorithmen). Diese Funktion wird immer dann aufgerufen, wenn im 'Visualisation' Dialog der Button 'Update' aktiviert wird.

5 Modulschnittstellen

Ein 'VNNlab'-Modul befindet sich stets in einem eigenen Ordner. Der Name des Ordners (Modulordner) ist gleichzeitig der Name des Moduls. Im Modulordner müssen sich mindestens eine tcl-Datei und eine so-Datei befinden. Die tcl-Datei enthält eine Beschreibung des Einstellungsdialoges des Moduls. Die so-Datei (Shared Object) enthält die eigentliche Modulfunktionalität. Das Erstellen eines 'Shared Objects' ist in Kapitel 2.2 beschrieben. Alle 'VNNlab'-Module besitzen zwei Strings mit Informationen über das Modul:

- title: Dies ist der ausführlichere Name des Moduls. Der Modultitel erscheint z.B. immer über dem Einstellungsdialog des Moduls.

- tcl_symbols: Dies ist eine Liste von Variablen, die das C-Programm zusammen mit dem Tcl/Tk-Script verwendet. Vor dem Aufruf einer C-Funktion aus dem Modul werden diese C-Variablen aktualisiert, d.h. auf die Werte der entsprechenden Tcl-Variablen gesetzt. Beim Initialisieren eines Modules werden die Variablen-Initialisierungen des C-Programmes als Standardeinstellungen der Variablen verwendet. Die Elemente der Liste werden durch ein Komma getrennt. Ein Element besteht aus seinem Typ und seinem Variablennamen. Gültige Typen sind `'long'`, `'float'` und `'string'`. Zwischen den Elementen dürfen Leerzeichen, aber keine Tabulatoren enthalten sein. Ein Beispiel für eine solche Liste ist:

  ```
  char *tcl_symbols = "string filename,long iterations,float adaption"
  ```

Die im String `'tcl_symbols'` angegebenen Variablen müssen im C-Programm global deklariert werden. Dabei werden die Initialisierungswerte der Variablen gleichzeitig als Defaultwerte (auch im Tcl/Tk-Script) der Modulparameter verwendet. Die Deklaration eines 'Strings' muß als char-Array erfolgen, z.B.:

```
char *filename[256];
```

Die verschiedenen Modultypen unterscheiden sich nur in deren Funktionen, die das entsprechende Modul beinhalten muß. Im folgenden werden für die in 'VNNlab' verwendeten Modultypen diese Funktionen beschrieben.

5.1 Verteilungsmodule

Die Tcl/Tk-Funktion zum Erzeugen eines Dialoges für ein Verteilungs-Modul ist definiert als:

```
proc Distribution_Dialog {root}
```

Sie bekommt als Parameter ein Frame-Widget übergeben in welchem der Dialog erzeugt werden soll. Ein Beispiel für eine solche Definition eines Dialoges könnte wie folgt aussehen:

```
proc Distribution_Dialog {root} {
        global src_img_fname
        label $root.sil -text "Source Image:"
        grid  $root.sil -row 0 -column 0 -sticky e
        entry $root.si -textvariable filename -state disabled
        grid  $root.si -row 0 -column 1 -sticky ew
        bind  $root.si <ButtonRelease-1> "
                set fname \[tk_getOpenFile\]
                if {\$fname!=\"\"} {
                        set filename \$fname
                }
                $root.si xview end
                InitDistribution
        "
        label $root.trl -text "Iterations:"
        grid  $root.trl -row 1 -column 0 -sticky e
        scale $root.tr -variable iterations -from 1 -to 100 -orient horizontal
        grid  $root.tr -row 1 -column 1 -sticky ew
}
```

Dieser Dialog erzeugt ein Texteingabefeld. Beim Mausklick auf dieses Texteingebefeld öffnet sich eine Fileselektor-Box und erfragt einen Dateinamen (z.B. für ein Bild). Der Dateiname wird der Variablen `'filename'` zugewiesen. Desweiteren wird ein Schieberegler erzeugt und mit der Variable `'iterations'` verbunden. Zur Programmierung von Tcl/Tk ist das Buch “Practical Programming in Tcl/Tk” von B.Welch empfehlenswert.

Das C-Programm eines Verteilungs-Modules muß folgende Funktionen enthalten:

- `struct dist_data *Create_Distribution(void);`
  Diese Funktion erzeugt eine diskrete Verteilung. Sie gibt einen Zeiger

auf eine (nur innerhalb des Moduls verwendete) Struktur zurück. Dieser Zeiger wird außerhalb des Modul lediglich als 'handle' für eine Verteilung behandelt. Was sich hinter diesem Zeiger tatsächlich verbirgt, weiß nur das Modul.

- `void Destroy_Distribution(struct dist_data *d);`
  Diese Funktion zerstört eine Verteilung. Ihr wird der Zeiger auf eine (zuvor mit `'Create_Distribution'` erzeugte) Verteilung übergeben.

- `distribution *Get_Distribution(struct dist_data *d);`
  Hiermit werden aus der angegebenen Verteilung Daten exportiert, die der 'offiziellen' Datenstruktur `'distribution'` genügen. Die Datenstruktur `'distribution'` ist in der Datei `distrib.h` definiert. Diese Datenstruktur stellt somit die Schnittstelle zwischen einem Verteilungsmodul und einem Netzwerkmodul dar.

- `void Draw_Distribution(struct dist_data *d);`
  Diese Funktion zeichnet die angegebene Struktur. Sie wird normalerweise innerhalb einer 'glut'-redraw Funktion ausgeführt. Zum Zeichnen können sämtliche OpenGL Primitive verwendet werden.

- `long Get_Distribution_Width(struct dist_data *d);`
  Hiermit wird die 'Breite' der Verteilung (entspricht normalerweise der Breite des zugrundeliegenden Bildes) zurückgeliefert.

- `long Get_Distribution_Height(struct dist_data *d);`
  Hiermit wird die 'Höhe' der Verteilung (entspricht normalerweise der Höhe des zugrundeliegenden Bildes) zurückgeliefert.

5.2 Netzwerkmodule

Die Definition des Einstellungsdialoges eines Netzwerkmodules erfolgt analog zu der Definition eines Einstellungsdialoges von Verteilungsmodulen. Hier heißt die Tcl-Funktion:

```
proc Network_Dialog {root}
```

Das C-Programm des Netzwerk-Modules muß folgende Funktionen enthalten:

- `network *Create_Network(long init_w,long init_h);`
  Diese Funktion erzeugt ein Netzwerk. Die Datenstruktur `'network'` wird nur innerhalb des Modules verwendet. Außerhalb des Modules wird der Zeiger auf diese Struktur nur als 'handle' auf ein Netzwerk verwendet. Als Parameter erhält die Funktion die gewünschte Ausdehnung des Netzes bei dessen Initialisierung.

- `void Destroy_Network(network *net);`
  Hiermit wird das angegebene Netzwerk gelöscht (deallokiert).

- `void Draw_Network(network *net);`
  Diese Funktion zeichnet das angegebene Netzwerk. Sie wird normalerweise innerhalb einer 'glut'-redraw Funktion aufgerufen. Es können sämtliche OpenGL Primitive zum Zeichnen verwendet werden.

- `network *Compute_Network(network *net,distribution *dist);`
  Diese Funktion stellt die eigentliche Berechnungsfunktion des Netzwerkes dar. Sie führt genau einen Berechnungsschritt durch. Dieser Berechnungsschritt kann aber durchaus aus mehreren Iterationen bestehen. Die Funktion gibt wiederum einen Zeiger auf ein Netzwerk zurück. Dieser muß nicht mit dem Parameter 'net' übereinstimmen. Es kann durch diese Funktion beispielsweise ein komplett neues Netz erzeugt und zurückgeliefert werden. Dann jedoch sollte das alte Netz auch von dieser Funktion deallokiert werden (z.B. mit dem Aufruf von `'Destroy_Network'` innerhalb der Funktion). Als weiteren Parameter bekommt die Funktion die Verteilung übergeben, die als Grundlage für den Lernvorgang verwendet werden soll. Die Struktur der Verteilung ist in der Datei `distrib.h` definiert.

- `float Compute_Error(long iter,network *net,distribution *dist);`
  Diese Funktion errechnet den momentanen durchschnittlichen Quantisierungsfehler des angegebenen Netzes. Sie wird lediglich zum Ausgeben von Diagrammen verwendet.

- `mesh *Export_Mesh(network *net);`
  Hiermit wird aus dem angegebenen Netz ein 'Gitter' (mesh) erzeugt. Die Datenstruktur `'mesh'` ist in `mesh.h` definiert. Diese Datenstruktur dient als Schnittstelle vom Netzwerkmodul zur den weiteren Berechnungen. Ein komplettes `'mesh'` besteht aus Punkten und Polygonen. Es

müssen aber lediglich die Punkte exportiert werden. Wenn keine Polygone exportiert werden, ist das Element `'mesh->num_polys'` mit `0` und das Element `'mesh->polys'` mit `NULL` zu initialisieren. Dann nämlich werden die Polygone anschließend durch einen Triangulationsalgorithmus erzeugt.

5.3 Korrespondenzanalysemodule

Die Definition des Einstellungsdialoges eines Korrespondenzanalysemodules (Depth Evaluation) erfolgt analog zu der Definition eines Einstellungsdialoges von Verteilungsmodulen. Hier heißt die Tcl-Funktion:

```
proc DepthEval_Dialog {root}
```

Das C-Programm des Korrespondenzanalysemodules muß folgende Funktionen enthalten:

- `void Init_DepthEval();`
  Dies ist die Initialisierung des Modules. Diese Funktion sollte z.B. genutzt werden, um das Stereobildpaar einzulesen.

- `void Deinit_DepthEval();`
  Dies ist die Deinitialisierung des Modules. In dieser Funktion sollte beispielsweise das Stereobildpaar deallokiert werden.

- `float Get_Similarity(long *left, long *right, long w);`
  Diese Funktion stellt das Ähnlichkeitsmaß des Verfahrens dar. Sie wurde im Wesentlichen eingeführt, um Untersuchungen von 'Unähnlichkeitskurven' zu ermöglichen. Die Funktion wird von 'VNNlab' nicht genutzt! Dennoch sollte sie zuminstest als 'dummy'-Funktion implementiert werden. Die Parameter sind Pointer auf die Bildpositionen, die zu untersuchen sind. Der Parameter `'w'` gibt die Breite einer Bildzeile in Pixeln an.

- `float Get_Disparity(long x,long y);`
  Dies ist die eigentliche Funktion zur Berechnung von Disparitäten, die auch von 'VNNlab' genutzt wird. Es werden lediglich die Koordinaten eines Bildpunktes übergeben, von dem die Tiefe bestimmt werden soll. Zurückgegeben wird lediglich ein Disparitätswert (ein Maß für die Tiefe eines Bildpunktes).

5.4 Visualisierungsmodule

Die Definition des Einstellungsdialoges eines Visualisierungsmodules (Visualisation) erfolgt analog zu der Definition eines Einstellungsdialoges von Verteilungsmodulen. Hier heißt die Tcl-Funktion:

```
proc Visualisation_Dialog {root}
```

Das C-Programm des Visualisierungsmodules muß folgende Funktionen enthalten:

- `visdata *Create_Visualisation(mesh *m);`
  Mit dieser Funktion wird eine 'Visualisierung' erzeugt. Eine 'Visualisierung' ist eine Datenstruktur, die sämtliche notwendigen Daten zur Darstellung eines 'mesh' enthält. Diese Daten sind zunächst einmal die Punkte und Polygone, aber auch Texturen, Positionen von Lichtquellen o.ä.. Die 'Visualisierungs'-Struktur wird von der Funktion zurückgeliefert. Der Zeiger auf diese Struktur wird außerhalb des Modules jedoch lediglich als 'handle' verwendet um die 'Visualisierung' den anderen Funktionen des Modules zu übergeben.

- `void Draw_Visualisation(visdata *v);`
  Diese Funktion zeichnet die angegebene Visualisierung. Sie wird in der Regel innerhalb der 'glut'-redraw Funktion aufgerufen. Es können sämtliche OpenGL Primitive zum Zeichnen verwendet werden.

- `void Destroy_Visualisation(visdata *v);`
  Hiermit wird die angegebene 'Visualisierung' aus dem Speicher entfernt.