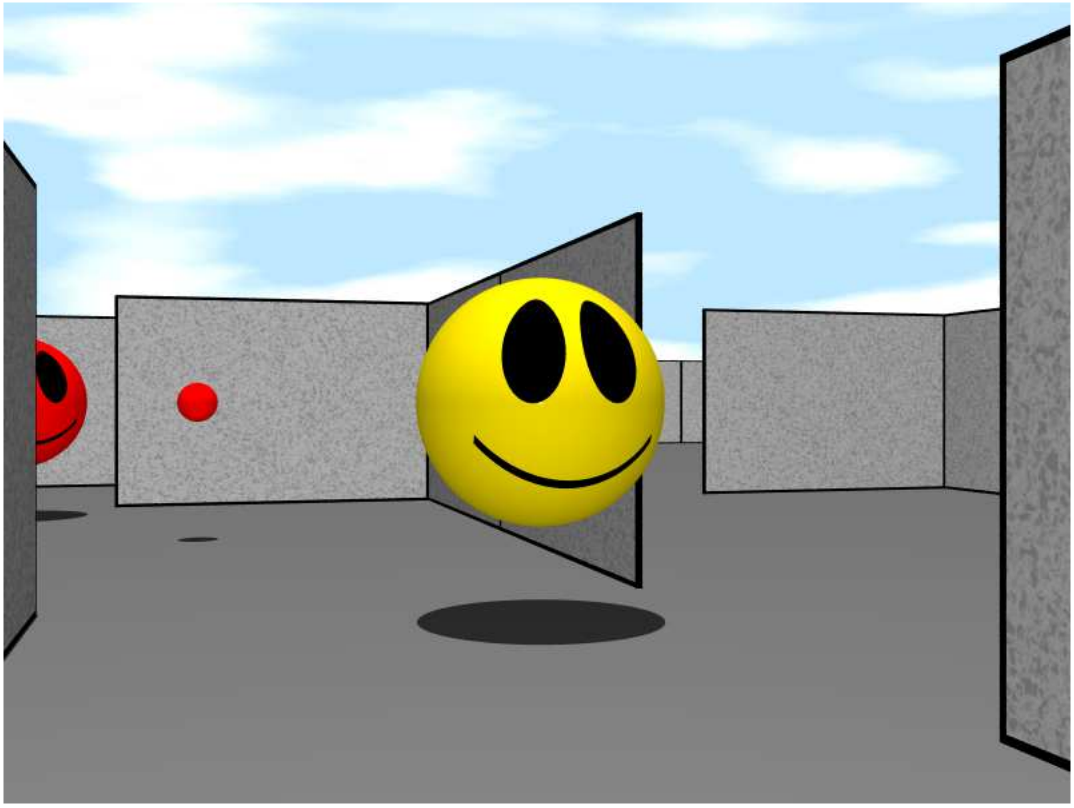
Softwarepraktikum
Netzwerkprogrammierung unter Unix

am Beispiel des Spiels

iMaze



Jörg Czeranski
Hans-Ulrich Kiel

1993/94

Inhaltsverzeichnis

Kapitel 1

Einleitung

Ziel des Softwarepraktikums „Netzwerkprogrammierung unter Unix“ war es, ein auf dem Atari ST verbreitetes Computerspiel unter Unix mit X11-Oberfläche neu zu entwickeln und dabei ein Netzwerkprotokoll auf Basis von TCP/IP zu realisieren, das den besonderen Anforderungen eines Spiels gerecht wird.

Das Atari ST - Spiel *MidiMaze* existiert bereits in drei verschiedenen Versionen. Allen Versionen gemeinsam ist eine Vernetzung auf Basis der in Atari ST's standardmäßig eingebauten Midi-Schnittstelle – daher der Name des Spiels. Diese serielle Schnittstelle wird vorwiegend zum Anschluß von Musikgeräten verwendet, dient in diesem Fall jedoch zum Aufbau eines ringförmigen Netzes aus maximal 16 Computern. Über diesen Ring, der mit 31.250 Bits/s vergleichsweise langsam ist, teilen alle Spieler ihre Eingaben und Positionen den anderen Spielern mit. Ein Computer im Netz steuert als „Master“ das Spiel, während alle anderen als „Slave“ die vorgenommenen Einstellungen übernehmen.

Die hier beschriebene Umsetzung der Spielidee auf Unix und das Internet wurde analog „*InternetMaze*“, kurz „*iMaze*“, genannt.

1.1 Die Spielidee

Alle Mitspieler befinden sich in einem labyrinthartigen Spielfeld, vom dem jeder in einer dreidimensionalen Sicht einen Ausschnitt sieht. Er kann vor- und rückwärts laufen und sich drehen. Die anderen Mitspieler sieht er in seinem Blickfeld als Gesicht. Auf einer Spiel-Karte kann man sich einen Überblick verschaffen. Ziel ist es, andere Mitspieler abzuschießen, ohne jedoch selbst getroffen zu werden. Abgeschossene Spieler entstehen nach einer kurzen Wartezeit an einer zufällig gewählten Stelle im Labyrinth neu und können weiter teilnehmen. Über ein Punktesystem, das noch nicht implementiert ist, wird ein Sieger ermittelt.

1.2 Ein Spiel als Softwarepraktikum?

Dies ist eine Frage, die uns häufig gestellt wurde und wir müssen zugeben, daß wir anfänglich selbst große Zweifel an dem Projekt hatten, sind jetzt jedoch von der Idee überzeugt.

Als Begründung könnte man anführen, daß ein solches Projekt nicht trivial ist, daß wir mit ca. 15.000 Zeilen C-Quelltexten und ca. 500 Stunden Arbeit deutlich über dem Durchschnitt für Softwarepraktika liegen oder daß viele Benutzer das Programm begeistert aufgenommen haben. Aber das alles rechtfertigt die Vergabe eines Softwarepraktikums nicht. Trotzdem halten wir die Idee für gut und wiederholbar. Warum?

- Das Projekt deckt von der Programmierung einer grafischen Oberfläche bis zur Realisierung eines Netzwerkprotokolls fast alle Bereiche der Unix-Programmierung ab. Von fundierten Kenntnissen der Programmiersprache C einmal abgesehen mußte eine Einarbeitung in X11- und TCP/IP-Programmierung erfolgen.

- Die dreidimensionale Projektion des Spielfeldes und die Berechnungen der Kollisionen von Spielern mit Spielern oder Wänden sind teilweise durchaus anspruchsvoll.

- Ein Spiel, bei dem ständig eine Bewegung erfolgt, stellt hohe Anforderungen an die Programmierung, da jede Verzögerung sofort negativ auffällt. Hier treten Echtzeit-Bedingungen auf, die auf Multitasking- und Multiuser-Plattformen schwer zu beherrschen sind.

- Im Gegensatz zum Vorbild des Midi-Rings ist die Übertragungsgeschwindigkeit und -qualität bei TCP/IP stark schwankend. Es können Pakete verloren gehen oder Unterbrechungen und Verzögerungen auftreten. Trotzdem soll die Spielbarkeit und eine Gleichberechtigung aller Mitspieler gewährleistet sein.

- Um den Rahmen nicht zu sprengen, wurden Spieloptionen, die zwar Programmieraufwand bedeuten, aber wenig anspruchsvoll sind, z. B. eine Punktezählung, in der hier beschriebenen Version nicht realisiert. Der Quelltext und das Protokoll mußten dadurch so angelegt werden, daß eine nachträgliche Erweiterung einfach möglich ist.

- Die Portierung auf andere grafische Oberflächen oder gar andere Betriebssysteme wurde weitgehend durch Schnittstellen im Programm vorgesehen.

- Dadurch, daß der Quelltext allen Spielern zugänglich ist, muß zusätzlich die Möglichkeit, durch Manipulation Vorteile zu erlangen, unterbunden werden.

Alle diese Punkte rechtfertigen in unseren Augen die Vergabe eines Softwarepraktikums. Viele hier „spielerisch“ erarbeitete Verfahren wie die dreidimensionale Grafikausgabe und das Protokoll unter Echtzeit-Bedingungen können in anderen Software-Projekten als Anregung dienen.

1.3 Aufgabenstellung

Aufgabe war es, die Spielidee von MidiMaze möglichst originalgetreu und vollständig auf Unix mit X11-Oberfläche zu verwirklichen. Als Vorbild wurde *MidiMaze 2* gewählt, da sich diese Version in langen Testreihen als besonders gelungen erwiesen hat.

Da sich abzeichnete, daß eine Umsetzung aller Spieloptionen von MidiMaze 2 den Rahmen des Softwarepraktikums sprengen würde und viele dieser zusätzlichen Funktionen erstens wenig anspruchsvoll zu programmieren und zweitens für das Aufkommen des nötigen Spielspaßes nicht zwingend erforderlich sind, wurde als Schwerpunkt der Aufgabe die Verwirklichung der Grundidee und der anspruchsvollen Teile festgelegt, wie Projektion der dreidimensionalen Sicht und Definition des Protokolls.

Da uns Sun-Workstations sehr gut verfügbar und langjährig bekannt waren, wurde als Hardwareplattform für die Programmerstellung Sun SPARCstation 10 mit SunOS 4.1.3 und OpenWindows 3.0 gewählt. Das Projekt sollte jedoch so angelegt werden, daß eine Portierung auf andere Unix-Systeme und Unix-Oberflächen vorgesehen ist. Unter diesen Bedingungen bietet sich die Programmiersprache C an. Um möglichst alle C-Compiler zu unterstützen, haben wir uns für C nach K&R entschieden.

Die Aufgabe gilt als erfüllt, wenn die Grundidee von MidiMaze 2 in einem voll funktionsfähigen Programmpaket, bestehend aus einem Server und Clients realisiert wird. Ferner sollten folgende Anforderungen berücksichtigt werden:

- Sowohl die Quelltexte als auch das Protokoll sollen so strukturiert werden, daß später Erweiterungen einfach eingefügt werden können, insbesondere solche, die zum vollen Funktionsumfang von MidiMaze 2 noch fehlen.
- Ältere Client- oder Serverversionen sollten möglichst mit weiterentwickelten Versionen kommunizieren können.
- Eine Verbindung sollte mit unterschiedlichen TCP/IP-Stationen, auch über die Grenze eines Subnetzes hinaus, aufgebaut werden können. Auch bei langsamen und qualitativ schlechten Leitungen soll eine Teilnahme am Spiel nicht ausgeschlossen sein.
- Kein Spieler darf Vor- oder Nachteile haben, sofern die Verbindung eine minimale Qualität nicht unterschreitet.
- Es ist eine Schnittstelle im Grafik-Client vorzusehen, die eine Portierung auf andere Unix-Oberflächen erlaubt.
- Diese Schnittstellen und das Protokoll müssen entsprechend geplant und dokumentiert werden.

- Obwohl das Spiel vorrangig für TCP/IP gedacht und dafür zu implementieren ist, sollte das Protokoll so definiert sein, daß es von einem spezifischen Netzwerk unabhängig ist.

- Das Dateiformat für die Spielfelder ist ebenfalls festzulegen und zu dokumentieren.

1.4 Grundlegende Konzepte

Die Ringtopologie, die bei MidiMaze als Basis für die Verbindung der beteiligten Computer diente, hat einige Eigenschaften, die die Entwicklung eines solchen Spieles vereinfacht:

- Das Netz über die Midi-Schnittstelle ist auch auf Atari-Computern nicht üblich, d.h. ein solches Netz wird in der Regel nur für das Spiel installiert und nicht dauerhaft betrieben oder gleichzeitig für andere Netzwerkdienste genutzt. Deshalb kann man davon ausgehen, daß alle angeschlossenen Computer am Spiel teilnehmen, solange das Spiel läuft.

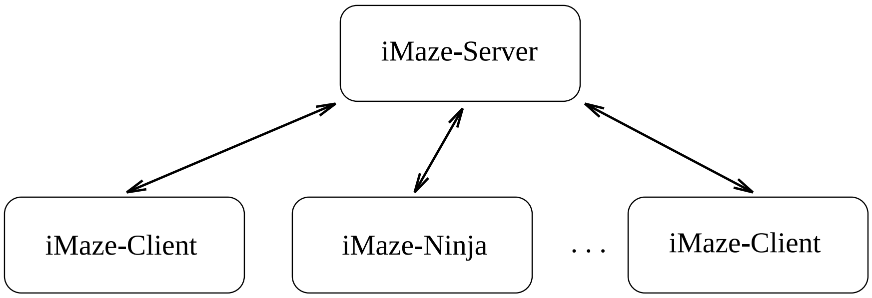
- Der eher langsame Ring dient gleichzeitig zur Synchronisation der angeschlossenen Computer. Jeder Spieler macht erst dann wieder einen neuen Schritt, wenn das Datenpaket alle anderen Computer durchlaufen hat und bei ihm wieder eintrifft. Es ist so sichergestellt, daß alle Mitspieler den eigenen Schritt empfangen haben.

- Ist der Ring einmal installiert, sollte ein unterbrechungsfreier Betrieb mit konstanter Signallaufzeit gewährleistet sein.

Alle diese Bedingungen sind im Internet nicht gegeben. Das Netz ist dauerhaft installiert und wird für die verschiedensten Dienste verwendet. Die Verbindungen können unterschiedlich schnell sein und in der Qualität laufend schwanken. Man muß damit rechnen, daß ein beliebiger Spieler das Spiel verläßt oder daß seine Verbindung zusammenbricht. Ein Ring wie bei MidiMaze ist unter diesen Bedingung nicht realisierbar. Stattdessen ist ein völlig neues Konzept erforderlich.

Erschwerend kommt hinzu, daß die Quelltexte frei verfügbar sind und von jedem Spieler manipuliert werden können. Aber davon abgesehen läuft das Spiel unter Umständen auf ganz unterschiedlichen Hardware-Plattformen, so daß allein dadurch eine Chancengleichheit nicht selbstverständlich gegeben ist.

Wir haben uns als grundlegendes Konzept für eine *Client-Server*-Lösung entschieden. Für ein Spiel wird einmal ein Server gestartet und jeder Spieler startet einen (oder mehrere) Client(s), der eine Verbindung zum Server herstellt. Ein solcher Client ist normalerweise ein Programm, das eine grafische Schnittstelle für einen Benutzer bietet, mit der er sich durch das Labyrinth bewegen kann. Denkbar ist aber auch ein Client, der nicht von einem Benutzer bedient wird, sondern nach einer mehr oder weniger guten Strategie vom Computer gesteuert wird. Diese Computerspieler werden im folgenden auch *Ninja* genannt. Für den Server sind diese beiden Client-Arten jedoch gleichwertig.



Der Server übernimmt alle entscheidenden Aufgaben. Er synchronisiert mit Hilfe des Protokolls alle Teilnehmer und wertet die Spielzüge aus. Die Clients erhalten alle notwendigen Informationen vom Server und haben selbst keine Funktionalität zur Durchführung eines Spiels. Ihre Aufgabe besteht nur darin, die aktuelle Spielsituation anzuzeigen bzw. bei Computer-Spielern die Berechnung eines neuen Zuges durchzuführen und die Reaktion an den Server zu übertragen.

Die Clients erhalten nur soviel Informationen, wie für diese Aufgabe zwingend erforderlich ist. Als einzige Komponente hat der Server alle Informationen und ist in der Lage, damit eine neue Spielsitutation zu berechnen. Dadurch ist eine Chancengleichheit auch bei weiterentwickelten Clients oder bei unterschiedlich schneller Hardware weitgehend gewährleistet.

Nicht ganz verhindern kann man, daß Teilnehmer mit schlechten Netzverbindungen unter Umständen benachteilig sind. Das Protokoll kann dieses Problem etwas mildern, aber nie ganz verhindern. Nachteilig kann auch sein, daß der Server für jeden Spielschritt Paktete von jedem Client erhält und zu jedem Client schicken muß. Die maximale Anzahl von Mitspielern ist also durch die Leistungsfähigkeit des Server-Computers und der dort verfügbaren Netzwerkbandbreite gegeben. (Z. B. ist bei 42 Spielern eine freie Übertragungskapazität von bis zu 200 kByte/s erforderlich.)

1.5 Zum Inhalt

Nach dieser allgemeinen Einleitung soll die Realisierung des Programms detailiert beschrieben werden. Neben einem Überblick über die Struktur des Programms (Kapitel 2) und einer Beschreibung der wichtigsten Datenstrukturen steht eine Dokumentation der geforderten Portabilitätsschnittstellen (Kapitel 3) im Vordergrund.

Weitere Kapitel werden das Netzwerkprotokoll (Kapitel 4) und das Labyrinth-Dateiformat (Kapitel 5) definieren. Hier wird auch auf denkbare Erweiterungen eingegangen.

Abschliessend wird auf das Compilieren, Installieren (Kapitel 6) und die Bedienung der mitgelieferten Programme (Kapitel 7) eingegangen.

Die Teile Programmstruktur und Schnittstellen, Protokoll, Labyrinth-Format sowie Compilierung und Bedienung bilden jeweils für sich eine Einheit und sollten auch unabhängig von einander verständlich sein. Dem Leser ist es freigestellt, eine Auswahl zu treffen.

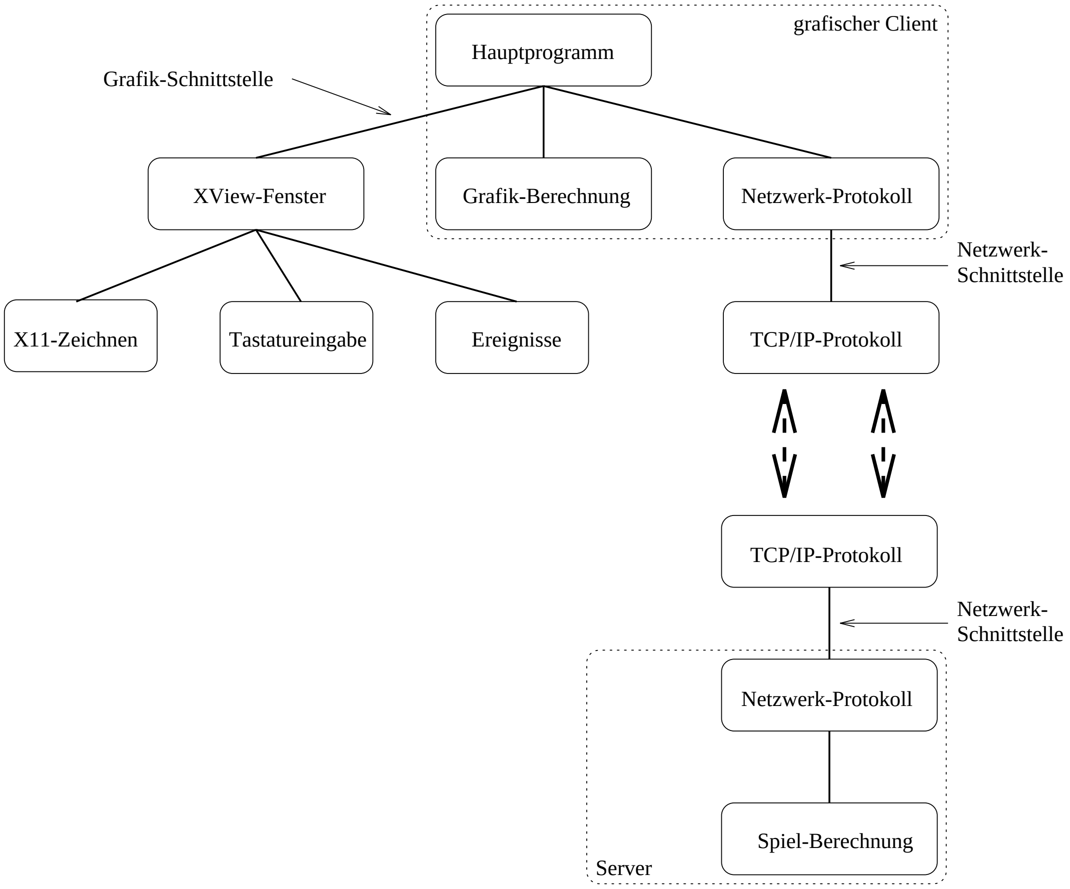
Alle Begriffe und Produktbezeichnungen werden ohne Rücksicht auf eventuell urheberrechtlichen Schutz verwendet.

Kapitel 2

Programm- und Datenstrukturen

2.1 Gliederung

Die wichtigsten Bestandteile des *iMaze*-Pakets sind der Server (`imazesrv`), der die Spiel-Steuerung übernimmt, und der Client (`imaze`), mit dem ein Benutzer am Spiel teilnehmen kann. Die grobe Gliederung beider Komponenten soll folgende Abbildung veranschaulichen:



Aus der Gliederung wird ersichtlich, daß sowohl der Client als auch der Server einen festen Bestandteil haben (dargestellt durch den gestrichelten Rahmen), als auch *Module*, die bei einer Portierung ausgetauscht werden können. Die Funktionsaufrufe zwischen den Modulen sind komplexer als in dieser Zeichnung darstellbar. Kapitel 3 wird sich detailiert mit den Schnittstellen zwischen den austauschbaren Modulen beschäftigen.

Der *Basisteil* des Clients enthält das Hauptprogramm und globale Funktionen, sowie Routinen, die Datenstrukturen anlegen und verwalten. Von hier wird die Neuberechnung und das Neuzeichnen des Fensterinhalts gesteuert. Ebenfalls fester Bestandteil des grafischen Clients sind die Funktionen, die aus den Positionen der Spieler eine dreidimensionale Sicht und dann auf dem Bildschirm zu zeichnende Grafikobjekte wie Vierecke und Kreise berechnen. Die Berechnung der Grafikobjekte für das Zeichnen der Karte verläuft analog. Alle Grafikdaten werden für einen virtuellen Bildschirm mit festen Ausmaßen erstellt und müssen vom Grafikmodul nur noch passend skaliert und in Zeichenbefehle umgesetzt werden.

Der *Grafikteil* des mitgelieferten Clients ist in zwei Module unterteilt. Die Verwaltung der Fenster, der Menüs und der X11-Events ist mit *XView* programmiert, da XView für diese Aufgaben eine sehr komfortable Schnittstelle zur Verfügung stellt. Hingegen ist das Zeichnen des Fensterinhalts mit Standard-X11-Befehlen gelöst. Bei einer Portierung auf Motif oder andere X11-Fenstermanager muß nur der erste Teil ersetzt werden. Auf XView basiert auch die Tastatureingabe. An ihre Stelle kann aber eine Maus- oder Joystick-Steuerung treten; genauso wie die Ausgabe der Ereignisse, die bisher nur einen einfachen Signalton kennt, durch eine Ausgaberoutine für Sounds ersetzt werden kann.

Zum Basismodul des Clients gehört ferner ein Programmteil, der das Netzwerkprotokoll verarbeitet. Dieser Teil behandelt nur die allgemeine, systemunabhängige Spezifikation des Protokolls (s. Kapitel 4). Die eigentliche Übertragung der Daten mit TCP/IP übernimmt ein getrenntes Modul.

Das Gegenstück bildet auf der *Server*-Seite ein Modul auf TCP/IP-Basis, das im Fall einer Portierung ausgetauscht werden kann. Der Kommunikationsteil des Servers ist aber erheblich aufwendiger, da sich neben dem laufenden Spiel jederzeit Clients an- oder abmelden können. Der Server teilt sich daher nach dem Start in zwei Prozesse. Der eine Teil wartet auf neue Clients, während der andere die laufende Spielrunde steuert. Für die Phase, in der ein neuer Client grundlegende Informationen, wie den Aufbau des Labyrinths erhält, wird vom Netzwerkteil kurzfristig ein weiterer Prozeß abgespalten. Die Berechnung eines neuen Spielstandes im laufenden Spiel ist in ein weiteres Quelltext-Modul ausgelagert.

Der Computergegner (`ninja`) ist ein vereinfachter Client. An die Stelle der Grafikberechnung und -ausgabe tritt hier eine Computer-Strategie. Die Module des Clients für die Kommunikation über das Protokoll und die Datenübertragung mit TCP/IP kommen unverändert zum Einsatz.

Das Programm `genlab` ist eigenständig und wird zur Erstellung neuer Labyrinth-Dateien verwendet (s. Kapitel 7.4).

Für den Compilationsvorgang wird ein weiteres Programm benötigt: `gentrigtab`. Es erstellt Tabellen der trigonometrischen Funktionen, die in den Programmen aus Zeitgründen nicht direkt berechnet werden können.

2.2 Inhalt der Dateien

In diesem Kapitel sollen alle C-Quelltext-Dateien aufgelistet und deren Inhalt bzw. Bedeutung kurz beschrieben werden. Sofern eine eindeutige Zuordnung zu den oben beschriebenen Komponenten möglich war, wurde eine entsprechende Gliederung vorgenommen.

2.2.1 Allgemeine Quelltexte

Einige Grundfunktionen werden in allen oder mehreren Komponenten verwendet:

`global.c` enthält ergänzende Funktionen, wie einen systemunabhängigen Zufallsgenerator und eine Integer-Wurzelfunktion.

`global.h` beinhaltet neben den Prototypen zu `global.c` Makros (`min`, `max`) und globale Definitionen (u. a. für die automatisch generierten trigonometrischen Tabellen in `trigtab.c`).

`speicher.c` stellt Funktionen zum Anfordern und Freigeben von Speicher sowie zur Verwaltung und zum Kopieren von Listen bereit.

`speicher.h` enthält nur die Prototypen zu `speicher.c`.

`system.c` enthält systemnahe Funktionen, die Timer starten, stoppen oder auswerten. Ferner sind hier die Funktionen zur Kompatiblität mit verschiedenen Unix-Varianten implementiert.

`system.h` enthält nur die Prototypen zu `system.c`.

`spieler.h` definiert die Datenstruktur, in der die Informationen über Position und Aussehen der Spieler bzw. Schüsse abgelegt wird (s. Kapitel 2.3.2). Dazu existieren Makros, mit denen Positionsangaben in die Datenstruktur geschrieben, daraus gelesen oder verändert werden können. In dieser Datei ist auch der Durchmesser eines Spielers und eines Schusses sowie die maximale Anzahl von Spielern definiert.

`labyrinth.h` enthält die Definition der Datenstrukturen, in denen der Grundriß des Spielfeldes gespeichert wird (s. Kapitel 2.3.1). Hier sind auch die maximalen Ausmaße des Labyrinths und die Unterteilung in ein Feinraster festgelegt.

`protokoll.h` ordnet allen Kennungen, die im Netzwerkprotokoll verwendet werden, einen Zahlenwert zu (s. Kapitel 4).

`ereignisse.h` spezifiziert alle vom Server übermittelten Spielereignisse (wie „Abschuß“ oder „Abgeschossen“), die von einem Client audio-visuell dargestellt werden können.

`signale.h` enthält die Definition der möglichen Steuersignale, die ein Client dem Server übermitteln kann (wie Spieler vor, zurück, drehen etc.).

`fehler.h` enthält den Prototyp für eine in allen Programmteilen verwendete Fehlerausgaberoutine.

2.2.2 Der grafische Client

Der grafische Client besteht aus einem *Basisteil*, der für alle Portierungen unverändert bleiben sollte, einem *Grafikteil*, der hier für X11 mit XView vorliegt und dem vom Übertragungsprotokoll abhängigen *Netzwerkteil* für TCP/IP.

Basisteil des Clients:

`client.c` enthält das Hauptprogramm für den grafischen Client und die Deklaration globaler Variablen. Die Routinen im Netzwerk-, Grafik- und Berechnungsteil werden von hier aufgerufen. Diese Programmteile rufen ihrerseits wieder Funktionen in `client.c` auf. Von zentraler Bedeutung ist eine Funktion, die alle Schritte für das Neuberechnen und Anzeigen der Grafikdarstellung aufruft.

`client.h` definiert die Prototypen zu `client.c`.

`spiel.h` beinhaltet Prototypen aus dem Hauptprogramm und dem Netzwerkteil und definiert somit die Schnittstelle zwischen beiden Komponenten.

`rechne3d.c` Dieses Modul übernimmt die komplette Berechnung der dreidimensionalen Ansicht. Als Eingabe dienen der Grundriß des Labyrinths und die Koordinaten der Spieler und Schüsse. Zuerst werden alle Wände projeziert und deren Verdeckungen errechnet und zum Schluß werden die sichtbaren Spieler und Schüsse in die Szene plaziert. Die Ausgabe beschreibt zu zeichnende Bildschirmobjekte – wie Vierecke und Kreise – skaliert für einen virtuellen Bildschirm fester Größe.

`rechne_karte.c` berechnet aus dem Grundriß und den Spielerpositionen die Grafikobjekte einer Karte für einen virtuellen Bildschirm. Dabei werden zwei Funktionen angeboten: die eine berechnet die unverändert bleibenden Linien für die Wände, die andere die Kreise für Position der Spieler.

`rechne.h` enthält die Prototypen zu `rechne3d.c` und `rechne_karte.c`.

`netzwerk.c` Dieser Quelltext beinhaltet die Verarbeitung des Kommunikationsprotokolls auf der Client-Seite, das ausführlich in Kapitel 4 beschrieben ist. Das Protokoll ist von der Art des verwendeten Netzwerks weitgehend unabhängig. Während das Übermitteln der Pakete von einer tiefer liegenden Programmschicht übernommen wird, erfolgt hier die Auswertung empfangener Pakete einschließlich der Konvertierung in das eigene Datenformat und das Zusammenstellen der Antwortpakete.

`netzwerk.h` definiert die Schnittstelle vom netzwerkunabhängigen `netzwerk.c` zu dem netzwerkabhängigen Programmteil, der die Verbindung zum Server aufbaut und die Pakete übermittelt.

`grafik.h` bildet die Schnittstelle zwischen dem Basis-Teil des Clients und dem Grafik-Teil. Dazu zählt auch die Deklaration der Datentypen, in denen die zu zeichnenden Grafikobjekte übergeben werden (siehe Kapitel 2.3.3). Die Ausmaße des virtuellen Bildschirms, für den die Grafikdaten berechnet werden, und weitere Parameter, die Einfluß auf die Darstellung haben, sind hier definiert.

`einaus.h` enthält die Prototypen für die Funktion zur audio-visuellen Darstellung von Ereignissen und zur Auswertung der Steuersignale des Benutzers. Diese Header-Datei ist ebenfalls der Grafik-Schnittstelle zuzuordnen.

X11/XView-Teil des Clients:

`xv_fenster.c` In diesem Programmteil ist das Fenstermanagement mit XView programmiert. Dazu zählen das Öffnen und Schließen des Hauptfensters und der Unterfenster, Behandlung von Veränderungen der Fenstergröße und andere Maus-Aktionen, Aufbau und Verwaltung der Menüs, Routinen zur Ausgabe von Fehlern und nicht zuletzt die Hauptschleife. Diese Hauptschleife wird nach der Initialiserungsphase aufgerufen und erst bei Spielende wieder verlassen. Das Programm ist Ereignis-gesteuert; bei Benutzereingaben oder neu ankommenden Netzwerkpaketen werden entsprechende Routinen angesprungen. Die Zeichenoperationen innerhalb der Fenster sind in diesem Teil nicht realisiert, sondern werden an den X11-Teil `X_grafik.c` weitergereicht.

`X_grafik.c` Das Zeichnen innerhalb der Fenster ist in Plain-X11 programmiert. Die Zeichenfunktionen erhalten die berechneten Grafikobjekte aus dem Basisteil, skalieren sie auf die aktuelle Fenstergröße und stellen sie mit X-Zeichenoperationen dar. Zeichenfunktionen gibt es für die 3D-Sicht, für den Kartengrundriß, für die Spieler auf der Karte und den Kompaß.

`X_grafik.h` definiert die Schnittstelle zwischen dem Fenstermanagement mit XView in `xv_fenster.c` und den Zeichenfunktionen mit Plain-X11 in `X_grafik.c`. Ferner sind die Datenstrukturen deklariert, in denen die zu zeichnenden Grafikobjekte für den Fall eines Neuaufbaus von Fenstern (z. B. infolge einer Veränderung der Fenstergröße durch den Benutzer) zwischengespeichert werden.

`X_farben.c` beinhaltet die Festlegung der Farben für Farbdarstellung und der Muster für Schwarz-Weiß-Darstellung. Neben Grundfarben für den Boden, den Himmel, Umrandungen usw. sind 15 Farben für Wände bzw. Türen und 16 Farben und 3 Augenfarben für Spieler definiert.

`X_farben.h` benennt Indizes für den Zugriff auf die Farbtabelle in `X_farben.c`.

`farben.h` spezifiziert Informationen über Farben, die von dem Grafiksystem unabhängig sind, wie Anzahl der Farben und den Wert, der als transparent zu interpretieren ist.

`xv_tasten.c` Die Benutzersteuerung ist in der vorliegenden Version nur über die Tastatur möglich. Dafür ist hier eine Routine vorhanden, die Tastatur-Events auswertet und die Eingaben bis zur nächsten Antwort an den Server sammelt.

`xv_ereignisse.c` Der Benutzer wird auf besondere Ereignisse im Spiel durch Aufblitzen des Bildschirms oder einen Piep aufmerksam gemacht. Entsprechende Routinen für XView sind hier implementiert.

`xv_einaus.h` Neben der Tastatureingabe ist auch eine Joystick-Eingabe o. ä. denkbar. Genauso kann die Untermalung von Ereignissen durch Sounds verbessert werden. Deshalb ist hier eine Schnittstelle für diese Ein-/Ausgaben zu `xv_fenster.c` definiert.

`xv_icon.h` enthält das Icon für *iMaze* im XView-Icon-Format.

`xv_icon_maske.h` enthält eine Pixel-Grafik zum Ausmaskieren des Icons.

TCP/IP-Teil des Clients:

`ip_netz.c` beinhaltet die Kommunikation mit dem Server auf Basis von TCP/IP. Die Kommunikation erfolgt wie in Kapitel 4.3 beschrieben sowohl über TCP als auch über UDP. Für beide Protokollarten sind Routinen für das Senden und Empfangen von Daten implementiert, ferner für den Aufbau und die Beendigung der Verbindung und den Wechsel von TCP auf UDP.

2.2.3 Der Ninja

Die Computergegner verhalten sich für den Server wie ein grafischer Client, und enthalten daher dieselben C-Quelltexte für die Kommunikation (`netzwerk.c` und `ip_netz.c`, s. oben). Die Grafikberechnung und Ein-/Ausgabe entfällt jedoch und wird durch eine Computerstrategie ersetzt. Der in Version 1.0 mitgelieferte Ninja ist mehr ein Abfallprodukt aus der Testphase als ein Beispiel für vorbildliche Programmierung von solchen Computergegnern.

Spezieller Quelltext des Ninjas:

`ninja.c` Dieser Quelltext ersetzt `client.c` des grafischen Clients und enthält das Hauptprogramm für den Computergegner. Anstelle der Aufrufe für die Berechnung und das Zeichen auf dem Bildschirm tritt hier eine primitive Computerstrategie.

2.2.4 Quelltexte des Servers

Bis auf die allgemeinen Funktionen aus Abschnitt 2.2.1 sind die Quelltexte des Servers von denen der Clients unabhängig. Wie eingangs erwähnt, teilt sich der Server in zwei – kurzfristig auch mehr – Prozesse. Die Quelltexte sind hauptsächlich nach den verschiedenen Phasen der Kommunikation mit den Clients und damit nach den Teilprozessen geordnet:

`server.c` enthält das Hauptprogramm des Servers. Nach dem Laden einer Labyrinth-Datei läuft die Hauptschleife, die auf neue Verbindungen von Clients wartet. Die Kommunikation mit den Clients läuft hier aber nicht ab.

`server.h` beinhaltet einen Prototyp aus `session.c` und einige Makro-Definitionen.

`session.c` Der Teil-Prozeß, der das laufende Spiel bearbeitet. In einer Hauptschleife wird der Zeittakt abgewartet, um dann alle von den Clients eingetroffenen Pakete auszuwerten. Dabei wird mit Paketseriennummern geprüft, ob die ankommenden Pakete zulässig sind oder ob der Client wegen Unterbrechungen in der Übertragung kurzfristig aus dem Spiel genommen werden muß. Nach der Berechnung aller neuen Positionen in `bewegung.c` werden die Pakete an die Clients zusammengestellt und abgeschickt. Dieses Modul behandelt die Kommunikation mit dem Client auf höherer Ebene, unabängig von der Art des Netzwerkprotokolls (TCP/IP).

`bewegung.c` berechnet die neuen Positionen der Spieler und Schüsse. Für jeden Client werden die ankommenden Steuersignale ausgewertet und geprüft, ob die gewünschte Bewegung zulässig ist. Falls nein, wird eine Ausweichposition errechnet (diese Aufgabe erfordert aufwendige geometrische Berechnungen). Neben den neuen Positionen der Spieler und Schüsse werden alle Ereignisse, wie Abschüsse etc. festgestellt und berücksichtigt. Die Länge der Schritte für Spieler und Schüsse sowie die Pause für tote Spieler sind am Anfang dieser Datei definiert.

`bewegung.h` enthält die Prototypen zu `bewegung.c`.

`init_spieler.c` Der Teilprozeß, der mit einem neuen Client in der *Prolog-Phase* des Protokolls (s. Kapitel 4) kommuniziert und z. B. das Labyrinth überträgt. Wie schon bei `server_spiel.c` ist auch hier die Kommunikation von der verwendeten Netzwerkart unabhängig.

`init_spieler.h` beinhaltet den Prototyp zu `init_spieler`.

ip_server.c übernimmt den Aufbau von Verbindungen und das Senden und Empfangen von Paketen auf Basis des TCP/IP-Protokolls. Der Quelltext enthält ferner Routinen zur Kommunikation der Teilprozesse untereinander. Die Interprozeß-Kommunikation läßt sich von der Kommunikation mit den Clients nicht vollständig trennen.

server_netz.h definiert die Schnittstelle zwischen dem Netzwerk-abhängigen Programmteil (`ip_server.c` für TCP/IP) und den übrigen Programmkomponenten sowie für die Interprozeß-Kommunikation.

2.2.5 Sonstige Quelltexte

Neben den Clients und dem Server gibt es noch folgende kleine Programme:

gentrigtab.c bildet ein eigenständiges Programm, das die trigonometrischen Tabellen für Sinus, Cosinus und Arcustangens berechnet und in der Datei `trigtab.c` ablegt, die sowohl im Client als auch im Server eingebunden wird.

baulab.c Ebenfalls ein eigenständiges Programm (`genlab`), das über einen Zufallsgenerator unter Berücksichtigung einiger Parameter ein neues Labyrinth generiert und in eine Datei abspeichert (s. Kapitel 5 und 7.4).

Der Aufbau des `Makefile`s sowie das Konfigurieren und Compilieren wird in Kapitel 6 beschrieben.

2.3 Übergreifende Datenstrukturen

Ein Programm mit einem Umfang von *iMaze* arbeitet mit einer Fülle von Datenstrukturen und -typen. Einige Datenstrukturen werden aber nur lokal verwendet und sind für das Verständnis des Programms weniger von Bedeutung. Daher wird die Beschreibung der Datenstrukturen in diesem Kapitel auf diejenigen beschränkt, die eine zentrale Rolle spielen und bei Portierungen und Erweiterungen berücksichtigt werden müssen.

2.3.1 Das Spielfeld

Die interne Datenstruktur, in der der Aufbau des Spielfelds abgelegt ist, ist in der Datei `labyrinth.h` definiert. Die Sichtweise entspricht der bei dem Labyrinth-Dateiformat (s. Kapitel 5.3).

Das Spielfeld ist in maximal 127*127 quadratische Felder unterteilt, die eine feste Außenlänge haben. Zu jedem Feld werden für die vier Seiten, betrachtet vom Feldinneren, die Farbnummer und weitere Informationen gespeichert.

Eine solche Seite eines Feldes (`wand` genannt) hat folgende Struktur:

```
struct wand
{
        u_int unbegehbar:1;
        u_int schusssicher:1;
        u_int tuer:1;
        u_int tueroffen:1;
        u_int farbe:4;
}
```

Alle Informationen über eine solche Wand sind also in 8 Bit festgehalten. Davon bestimmen vier Bits die Farbe (15 und `TRANSPARENT` stehen zur Wahl) und je ein Bit die Spezifikation einer Eigenschaft. Die Eigenschaften `unbegehbar` (die Wand kann von einem Spieler in dieser Richtung nicht durchschritten werden) und `schusssicher` (die Wand ist von einem Schuss nicht durchquerbar) werden im Server verwendet, während der Client die Farben erhält und in der internen Berechnung `tueroffen` und `tuer` setzt (zur Darstellung).

Ein Feld (hier `block` genannt) besteht aus vier Wänden:

```
typedef struct wand block[4];
```

Sie sind in der folgenden Reihenfolge abgelegt:

```
#define NORD 0
#define WEST 1
#define SUED 2
#define OST 3
```

Das Spielfeld ist ein zweidimensionales Feld von diesen Blöcken:

```
extern block **spielfeld;
```

Der Block mit den Koordinaten $(0,0)$ liegt im Nordwesten. Die Speicherung erfolgt spaltenweise.

Zusätzlich zu dem normalen Spielfeld verwendet der grafische Client noch vier Spielfelder, die sich durch Drehung des Original-Spielfeldes in 90 Grad-Schritten ergeben.

```
extern block **feld_himricht[4];
```

2.3.2 Positionen der Spieler

Die Position von Spielern und Schüssen im Spielfeld sowie Informationen über ihr Aussehen wird in einer Struktur abgelegt, die in `spieler.h` definiert ist. Die Position erfolgt durch Angabe des Blockes – im Sinne der Definition von oben – (grobe Position) und innerhalb eines Blockes noch einmal durch ein 256*256 Punkte großes Gitterraster (feine Position):

```
struct position
{
        u_char xgrob;
        u_char xfein;
        u_char ygrob;
        u_char yfein;
};
```

Zu dieser Position kommen bei Spielern noch die Blickrichtung `blick` (gemessen in Schritten von $\frac{2\pi}{\texttt{WINKANZ}}$ gegen den Uhrzeigersinn) und der Farbe (eine aus 42 oder `TRANSPARENT`).

```
struct spieler
{
        struct position pos;
        s_char blick;
        u_char farbe;
};
```

Bei Schüssen entfällt die Blickrichtung:

```
struct schuss
{
        struct position pos;
        u_char farbe;
};
```

Von diesen Datentypen gibt es Variablen und Felder für die Speicherung von einzelnen bzw. allen Spielern und Schüssen. Die Reihenfolge der Spieler bzw. Schüsse in den Feldern ist nicht definiert.

2.3.3 Grafikdaten

Die Grafikberechnung im Client erzeugt aus den Positionen der Spieler und dem gespeicherten Spielfeld Listen mit den auf dem Bildschirm zu zeichnenden Grafikobjekten. Die Datenstrukturen, mit denen diese Grafikobjekte beschrieben werden, sind in `grafik.c` definiert und sollen im folgenden erläutert werden.

Alle Grafikobjekte sind für einen virtuellen Bildschirm der Größe `FENSTER_BREITE` und `FENSTER_HOEHE` berechnet und müssen passend skaliert werden. Der Nullpunkt wird oben links angenommen. Alle Positionsangaben für den virtuellen Bildschirm basieren auf dem `punkt`:

```
struct punkt
{
        int x, y;
};
```

Für das Zeichnen der Wände in der dreidimensionalen Sicht gibt es den Datentyp `flaeche`. Dabei wird ein Wandsegment als ein Viereck beschrieben:

```
struct flaeche
{
        struct punkt ecke[4];
        u_char farbe;
        u_char tuer;
};
```

`ecke[0]` ist die Ecke links oben, `ecke[1]` links unten, `ecke[2]` rechts unten und `ecke[3]` rechts oben. `farbe` ist eine der 15 Wand- und Türfarben bzw. `TRANSPARENT`. Das Flag `tuer` ermöglich Schwarz-Weiß-Clients die Hervorhebung von Türen.

Die Grafikberechnung ermittelt alle Überdeckungen von Grafikobjekten und reduziert damit den Zeichenaufwand. Die Wandsegmente werden zuerst gezeichnet. Da Spieler teilweise von Wänden verdeckt werden können, enthält die Datenstruktur `kugel` ein Feld mit den Sichtbarkeitsbereichen. Ein solcher `ausschnitt` beginnt bei einer x-Position und hat eine Breite. In y-Richtung überspannt er die volle Fensterhöhe.

```
struct ausschnitt
{
        int x, breite;
};
```

Damit kann jetzt die Beschreibung für das Aussehen eines Spielers oder Schusses definiert werden:

```
struct kugel
{
        struct punkt mittelpunkt;
        struct ausschnitt *sichtbar;
        int sichtanz;
        int radiusx, radiusy;
        int schatteny, schattenry;
        int blick;
        u_char farbe;
};
```

Ein Spieler oder Schuß ist eine Ellipse mit einem `mittelpunkt` und den Halbachsen `radiusx` und `radiusy`. Der Schatten ist eine Ellipse mit dem Mittelpunkt (`mittlepunkt.x`,`schatteny`) und den Halbachsen `radiusx` und `schattenry`. Falls `schattenry`=0 ist, wird kein Schatten gezeichnet. Das Zeichnen des Gesichts wird dem Grafikteil überlassen. Die Struktur enthält nur die Blickrichtung des Spielers (`blick`). Der Winkel wird als Index auf die trigonometrischen Tabellen in `trigtab.c` (0 bis `TRIGANZ`) dargestellt. 0 entspricht der Blickrichtung des eigenen Spielers. Bei -1 wird kein Gesicht gezeichnet, was für Schüsse die Regel ist. Hinzu kommt noch die Farbe (eine aus 42 oder `TRANSPARENT`). Die Sichtbarkeitsbereiche stehen in einem Feld von `ausschnitten`, wobei die Anzahl der Ausschnitte in `sichtanz` zu finden ist. Ein unverdeckter Spieler hat einen Sichtbarkeitsbereich, der so breit wie das Fenster ist.

In einer übergeordneten Struktur werden alle Grafikobjekte (Wände und Kugeln) für die dreidimensionale Darstellung zusammengefaßt:

```
struct objektdaten
{
        int hintergrund_zeichnen;
        int anzwaende;
        struct flaeche *waende;
        int anzkugeln;
        struct kugel *kugeln;
};
```

`objektdaten` enthält jeweils ein Feld von `flaechen` und `kugeln` und zwei Integervariablen `anzwaende` und `anzkugeln` mit der Anzahl der Elemente in den Felder. Das Flag `hintergrund_zeichnen` legt fest, ob Boden, Himmel und Horizont zu zeichnen sind.

Für das Zeichnen der Karte existieren eigene Strukturen mit den dabei verwendeten Grafikobjekten. Für den Grundriß der Karte benötigt man Linien, die zur Hervorhebung von Türen mit verschiedenen Zeichenstilen gezogen werden sollten. Genauer gesagt sind es Doppellinien, da eine Wand auf der einen Seite andere Eigenschaften haben kann als auf der anderen.

```
struct kartenlinie
{
        struct punkt ecke;
        int laenge;
```

```
        u_char senkrecht;
        u_char typ1;
        u_char typ2;
};
```

Ein Linienelement `kartenlinie` besteht aus einem Punkt auf dem Bildschirm und einer Länge in Bildpunkten (`laenge`). Durch Setzen des Flags `senkrecht` wird entschieden, daß die Linie vom angegebenen Punkt nach unten zu ziehen ist, anderenfalls nach rechts. In `typ1` wird die Art der Wand aus der Sicht von links bzw. oben definiert und in `typ2` aus der Sicht von rechts bzw. unten. Dabei gibt es folgende Möglichkeiten:

```
#define KEINELINIE 0
#define WANDLINIE  1
#define TUERLINIE  2
```

Auf der Karte sieht man nur die Position der Spieler, nicht ihre Blickrichtung oder ihre Schüsse. Die Datenstruktur für die zu zeichnenden Ellipsen vereinfacht sich daher gegenüber der oben beschrieben Objekte für die dreidimensionale Sicht erheblich:

```
struct kartenkreis
{
        struct punkt mittelpunkt;
        int radiusx, radiusy;
        u_char farbe;
};
```

Die Ellipsen haben einen `mittelpunkt`, die Halbachsen `radiusx` und `radiusy` sowie eine der 42 Spielerfarben (bzw. `TRANSPARENT`).

2.3.4 Signale und Ereignisse

Die Steuersignale des Spielers werden intern in einem Feld mit `SIGNALANZ` Elementen abgelegt. Für alle unterstützten Steuersignale ist ein Index in `signale.h` definiert. Eine Steuerung wurde eingegeben, wenn das entsprechende Element =1 ist.

Analog werden die Ereignisse, die im Spiel auftreten können, in einem Feld der Länge `ERG_ANZ` gespeichert. Die Indizes dazu sind in `ereignisse.h` definiert. Auch hier gibt es nur die Einträge 1 und 0.

Diese Art der Speicherung unterscheidet sich von der Konvention für die Übertragung der Signale bzw. Ereignisse, die in Kapitel 4 beschrieben wird.

2.3.5 Trigonometrische Tabellen

Da trigonometrische Funktionen mit mäßiger Genauigkeit in den Berechnungen sehr häufig vorkommen und die System-Routinen dafür ungeeignet und zu langsam waren, wird eine Datei mit trigonometrischen Tabellen von dem Programm `gentrigtab` während des Compilier-Vorganges generiert und eingebunden. Die Tabellen haben `TRIGANZ` Einträge (definiert in `global.h`).

```
int costab[TRIGANZ];
int sintab[TRIGANZ];
int atantab[TRIGANZ +1];
```

Die Cosinus- und Sinus-Tabellen enthalten Werte mit einer Schrittweite von $\frac{2\pi}{\texttt{TRIGANZ}}$, die mit `TRIGFAKTOR` multipliziert wurden. Die Arcustangens-Tabelle für den Bereich -1 bis 1 enthält Winkel skaliert mit $\frac{\texttt{TRIGANZ}}{2\pi}$ mit einer Schrittweite von $\frac{2}{\texttt{TRIGANZ}}$.

Kapitel 3

Schnittstellen

Wie bereits in der Einleitung beschrieben, bestand die Aufgabe des *iMaze*-Projekts darin, eine Implementation der MidiMaze-Idee für eine Unix-Plattform durchzuführen. Die Entscheidung fiel auf SunOS 4.1.3 mit OpenWindows 3. Neben dieser Implementation sollte die Portierung auf andere – vorwiegend Unix-ähnliche – Systeme vorgesehen werden. Für dieses Ziel wurden Schnittstellen definiert, an denen andere Quelltexte eingefügt werden können.

Die Abbildung auf Seite 9 zeigt die grobe Strukturierung der Programme in Komponenten und die wichtigsten Schnittstellen. Diese Schnittstellen sollen nun genauer erläutert werden. Dabei wollen wir uns auf die Schnittstellen im grafischen Client beschränken, da hier die meisten Portierungen auf andere Oberflächen, Steuerungen oder Sound-Ausgaben zu erwarten sind. Im Server existiert ohnehin nur eine Schnittstelle im Netzwerkteil, die den TCP/IP-Teil vom übrigen Server trennt. Die Schnittstellen zum TCP/IP-Teil im Client und Server sind aber eher ein Produkt einer sinnvollen Programmstrukturierung als eine Programmierschnittstelle für die Portierung auf andere Netzwerke. Eine solche Portierung ist zwar denkbar, aber nicht primäres Ziel des Softwarepraktikums.

Der grafische Client enthält neben dieser Netzwerkschnittstelle im wesentlichen die *Grafik-Schnittstelle*. Sie trennt die Berechnung der Grafik, die für alle Systeme gleich sein sollte, von der Darstellung der Grafik, die von der vorhandenen Oberfläche abhängt. Ebenfalls systemabhängig ist die Art der Steuerung des Spielers und die Art der Untermalung von Ereignissen.

Neben dieser Schnittstelle gibt es eine weitere, die die übergreifenden Hilfsfunktionen den Programmodulen des Clients, aber auch anderer Programme, verfügbar macht.

In der hier vorliegenden Implementation gibt es noch drei weitere Schnittstellen, die aber bei anderen Implementationen nicht vorhanden sein müssen:

1. Die *X11-Schnittstelle*. Dadurch daß das Fenstermanagement mit XView arbeitet und das eigentliche Zeichnen aber mit Plain-X11 erfolgt, ergibt sich hier eine Schnittstelle innerhalb des Grafikteils, die es erlaubt, das Fenstermanagement auf ein anderes X11-basierendes System zu übertragen, ohne die Zeichenroutinen neu zu schreiben.

2. Die *Schnittstelle zur Steuerung* gestattet, die Eingabe des Spielers, die nur für Tastatur-Steuerung implementiert wurde, gegen eine Joystick- oder Maus-Steuerung auszutauschen. Der Eingabe-Teil ist eine Unterkomponente des XView-Fenstermanagements.

3. Die *Schnittstelle für Soundausgabe* ist ebenfalls im XView-Teil angeordnet und erlaubt eine spätere Ergänzung zur Sound-Unterstützung.

3.1 Grafik-Schnittstelle

Die im folgenden beschriebene Schnittstelle trennt den systemspezifischen bzw. von der grafischen Oberfläche abhängigen Teil des Clients von dem für alle Systeme unveränderten Basisteil.

3.1.1 Initialisierung

Nach der Initialisierung des Netzwerkteils wird aus der `main`-Funktion, die im Basisteil des Clients liegt, die Funktion

```
int grafik_init(int *argc, char **argv);
```

aufgerufen. Sie bekommt die Aufrufparameter des Clients übergeben, woraus die Grafikspezifischen Angaben (wie Fenstergröße, etc.) auszuschneiden sind. Die restlichen Parameter werden wieder zurückgegeben. Der Rückgabewert gibt an, ob die Initialisierung erfolgreich war (0=ok, 1=Fehler).

Die Grafik-Initialisierung beinhaltet das Erzeugen und Vorbesetzen von lokalen Variablen und Feldern des Grafik-Teils und die Vorbereitung der Grafikausgabe, der Untermalung von Ereignissen und der Steuerung. Vorbereiten heißt hier, daß noch keine Anzeige erfolgt und keine Steuerung aktiv ist, aber alle notwendigen Strukturen dafür angelegt und beim Betriebssystem angemeldet werden, so daß bei Beginn des Spiel die Ausgabe sofort erfolgen kann. Bei grafischen Oberflächen sollten z. B. die Fenster und Menüs sowie die Event-Handler, die auf Mausaktionen reagieren, angelegt werden.

Sollte der Start der Grafik fehlschlagen (z. B. weil eine passende Hardware fehlt), so kann in besonders gravierenden Fällen das Programm noch direkt abgebrochen werden. Andernfalls erfolgt eine Rückkehr in die `main`-Funktion (ggf. mit „Fehler“ als Rückgabewert).

3.1.2 Hauptschleife

Nach dem Aufbau der Verbindung zum Server und dem Empfang der Spiel-Parameter, wie der Aufbau des Labyrinths, und der Umschaltung des Netzwerkprotokolls auf die Spielphase wird die Hauptschleife des Grafikteils aufgerufen.

```
void grafik_schleife(void);
```

„Hauptschleife“ soll heißen, daß diese Routine während der Teilnahme des Clients an einem Spiel nicht wieder verlassen wird. Die Schleife startet die Ausgabe und aktiviert die Steuerung.

Es wird eine Möglichkeit angeboten, das Programm regulär zu verlassen (z. B. durch einen Menüpunkt). Aber auch für alle anderen erwarteten und unerwarteten Abbrüche des Programms (z. B. durch `kill`) ist eine Vorkehrung zu treffen, die die Funktion

```
void client_spiel_beenden(void);
```

aus dem Basisteil aufruft. Nach dem Aufruf dieser Funktion fällt das Programm aus der Hauptscheife heraus und kehrt in die `main`-Funktion zurück.

Die Grafikschleife prüft durch regelmäßige Abfrage, ob neue Pakete vom Server eingetroffen sind und leitet damit ggf. das Neuberechnen und Neuzeichnen ein. Dazu wird die Funktion

```
void bewegung(int ms, int zeichnen);
```

aus dem Basisteil aufgerufen. Dabei ist mit `ms` anzugeben, wieviele Millisekunden auf Daten vom Server gewartet werden darf (0=nicht warten; −1=unbegrenzt warten). Mit dem Flag `zeichnen` wird bestimmt, ob das Neurechnen und -zeichnen eingeleitet werden darf, falls Pakete angekommen sind (0=nein, 1=ja). Eine Ereignis-gesteuerte Hauptschleife kann durch Auswerten des *Deskriptor* [1] , der von der Funktion

```
void *bewegung_deskriptor(void);
```

geliefert wird, aufgebaut werden, bei der der Aufruf der Funktion `bewegung` nur dann erfolgt, wenn neue Daten vom Server eintreffen.

3.1.3 Zeichnen

Wird beim Aufruf der Funktion `bewegung` das Zeichnen zugelassen, so wird nach der Auswertung der neuen Daten die Berechnung der Grafik ausgeführt. Darauf werden die Zeichenroutinen im Grafikteil aufgerufen. Die Zeichenaufrufe beginnen mit der Funktion

```
void zeichne_sync_anfang(void);
```

die der Synchronisation dient. Der Zeichenvorgang wird mit Aufruf der Funktion

```
void zeichne_sync_ende(void);
```

abgeschlossen. Alle Zeichenaufrufe finden in direkter Folge zwischen diesen beiden Aufrufen statt. Alle Zeichenroutinen dürfen keine anderen Funktionen der Grafikschnittstelle aufrufen. Während des Zeichenvorgangs darf `bewegung` nicht mit der Zeichenoption aufgerufen werden, um eine Überlagerung zu vermeiden. Nach dem Abschluß des Zeichenvorgangs kehrt `bewegung` in die Hauptscheife zurück.

Das Zeichnen der dreidimensionalen Sicht wird mit der Funktion

```
void zeichne_blickfeld(struct objektdaten *objekte_neu);
```

aufgerufen, wobei `*objekt_neu` von dem in Abschnitt 2.3.3 Typ ist. Falls diese Daten intern erneut benötig werden, ist eine Kopie anzulegen (z. B. falls durch äußeres Einwirken ein Neuzeichnen des Fensterinhalts erzwungen werden kann). Der Aufruf der Funktion

```
void zeichne_rueckspiegel(struct objektdaten *objekte_rueck_neu);
```

für den optionalen Rückspiegel erfolgt analog.

Der Aufruf für die Karte ist zweigeteilt. Die Grafikdaten für das Zeichnen des Grundrisses werden für jedes Labyrinth nur einmal zu Beginn übertragen, da das Labyrinth im Laufe des Spiels unverändert bleibt:

```
void zeichne_grundriss(int anzlinien_neu, struct kartenlinie *linien_neu);
```

`linien_neu` ist ein Feld von dem in Abschnitt 2.3.3 beschriebenen Datenstruktur für Kartenlinien und `anzlinien_neu` die Anzahl der Elemente in dem Feld. Da diese Daten nicht laufend geschickt werden, ist hier das Anlegen einer lokalen Kopie besonders geboten. Der Aufruf für das Zeichnen der Spieler auf der Karte erfolgt dagegen bei jedem Neuzeichnen:

[1] Der Datentyp ist systemabhängig und bei der Portierung des Netzwerkteils festzulegen. Bei Unix bietet sich ein Zeiger auf den *File-Deskriptor* an, über den die Daten vom Server empfangen werden

```
void zeichne_karte(int spieler_index, int anzkreise_neu,
        struct kartenkreis *kreise_neu);
```

`kreise_neu` ist ein Feld von dem bereits beschriebenen Datentyp für Kreise auf der Karte und `anzkreise_neu` beinhaltet die Anzahl der Elemente. Um den eigenen Spieler hervorheben zu können, ist dessen Index in dem Feld in `spieler_index` abgelegt. (Er ist =-1, falls der Spieler gerade tot ist.) Dem Programmier bleibt überlassen, ob die Karte bei jedem Zeichenvorgang komplett neugezeichnet wird, oder ob der Grundriß erhalten bleibt und die alten Spieler auf der Karte zuvor gelöscht werden. Zu überlegen ist auch die Frage, wie die Karte bei sehr großen Spielfeldern hantiert wird.

Optional kann ein Kompaß angezeigt werden. Der Aufruf lautet

```
void zeichne_kompass(int blickrichtung_neu);
```

wobei `blickrichtung_neu` ein Winkel in $\frac{2\pi}{\texttt{TRIGANZ}}$ ist (0=Norden, zunehmend gegen den Uhrzeigersinn). Der Kompaß kann genauso wie der Rückspiegel bei der Portierung entfallen; dann existieren im Grafikteil dazu nur leere Funktionen.

3.1.4 Signale und Ereignisse

Die Abfrage von Steuersignalen des Benutzers und die Untermalung von besonderen Spielereignissen ist ebenfalls eine Aufgabe, die dem Grafikteil zuzuordnen ist; wobei eine interne Schnittstelle im Grafikteil zur Trennung von der Grafikausgabe sinnvoll erscheint, da es für eine Oberfläche verschiedene Ein- und Ausgabegeräte geben könnte. Während eine solche Schnittstelle dem Programmierer überlassen bleibt, beschränkt sich der hier zu definierende Teil auf je eine Funktion.

Die Steuerung ist von dem Grafikteil zu initialisieren (etwa in `grafik_init`) und zu beenden (in `grafik_ende`). Sie sollte während der Hauptschleife ständig aktiv sein. Die Steuersignale werden auf nicht näher spezifizierte Weise im Grafikteil erfaßt und zwischengespeichert. In bestimmten Abständen werden alle seit der letzten Abfrage erfolgten Steuerungen von der Funktion

```
void eingabe_abfragen(char signale[SIGNALANZ]);
```

ausgelesen. Das übergebene Feld entspricht der Datenstruktur nach Abschnitt 2.3.4. Diese Funktion wird vom Basisteil aufgerufen, wenn ein neues Server-Paket angekommen ist und die Antwort darauf zusammengestellt wird. Alle z. Z. unterstützten Signale sind in der Datei `signale.h` beschrieben. Es gibt zwei Arten von Steuersignalen. Am Beispiel einer Tastatursteuerung erläutert besteht folgender Unterschied:

1. Die Cursor-Tasten zum Bewegen des eigenen Spielers (vor, zurück, drehen) schicken bei jeder Abfrage zwischen Drücken und Loslassen der Taste das Signal.

2. Alle übrigen Tasten schicken nur einmal beim Drücken der Taste das passende Signal und erst beim nächsten Niederdrücken erneut.

Die Routinen für die Untermalung von Ereignissen durch Sounds oder andere Effekte wird analog im Grafikteil initialisiert und beendet. Der Aufruf zur Darstellung solcher Ereignisse ähnelt einem Zeichenaufruf. Die Funktion

```
void ereignisse_darstellen(char ereignisse[ERG_ANZ]);
```

wird nach allen Zeichenaufrufen vor `zeichne_sync_ende` aufgerufen. Das Feld `ereignisse` entspricht der Beschreibung in Abschnitt 2.3.4. Die verfügbaren Ereignisse sind in dem Headerfile `ereignisse.h` definiert.

3.1.5 Fehler und Abfragen

Bestandteil des Grafikteils ist auch eine minimale Dialogschnittstelle. Vorgesehen ist eine Funktion, die einen Text anzeigt und eine Auswahl zwischen zwei Antworten läßt, und eine Funktion, die eine Fehlermeldung ausgibt und das Ende des Programms einleitet.

Eine `rueckfrage` kann verwendet werden, um auf leichtere Fehler hinzuweisen oder eine Auswahl für das weitere Vorgehen bereitzustellen. Die Funktion

```
int rueckfrage(char **meldung, char *weiter_knopf, char *abbruch_knopf);
```

erhält eine mehrzeilige Meldung als Feld von Strings (`*meldung`) und die beiden zur Wahl stehenden Antworttexte in `*weiter_knopf` und `*abbruch_knopf` und gibt als Booleschen Wert zurück, welche Alternative gewählt wurde. Bei der Definition der Funktion wurde eine Ja/Nein-Auswahl angenommen, wobei `*weiter_knopf` die positive Antwort (Rückgabe 1) und `*abbruch_knopf` die negative Antwort (Rückgabe 0) darstellt. Die Übergabe eines Textes für die beiden Alternativen ist optional. Wird `NULL` angegeben, so werden die Standardwerte „Continue“ und „Quit“ verwendet.

Für Fehler und Vorkommnisse, die einen Weiterbetrieb des Programms ausschließen, gibt es im Basisteil die Funktion `ueber_fehler`, die an jeder Stelle des Programms aufgerufen werden darf. Diese Funktion ruft im Grafikteil

```
void ueblen_fehler_anzeigen(char **meldung, char *knopf);
```

auf. Zu den Parametern von `ueblen_fehler_anzeigen` gilt dasselbe wie zu `rueckfrage`. Hier ist nur eine Antwort vorgesehen, die mit „Bad Luck“ vorbesetzt ist, falls in `*knopf` nichts übergeben wird. Die Meldung sollte unbedingt angezeigt werden; ggf. auf `stderr`, falls eine Grafikausgabe nicht mehr möglich ist. Nach der Bestätigung durch den Benutzer leitet der Basisteil das Programmende ein.

3.1.6 Ende

Der Grafikteil leitet das Programmende im Normalfall durch Verlassen der Hauptschleife ein und kehrt in die `main`-Funktion zurück. Aus dem Basisteil wird dann die Funktion

```
void grafik_ende(void);
```

im Grafikteil aufgerufen, die alle laufenden Routinen (auch der Steuerung) beendet und lokal verwendete Daten aufräumt. Die Funktion kann im Fehlerfall auch bei noch laufender Hauptschleife aufgerufen werden. Diesem Umstand ist Rechnung zu tragen.

3.2 Interne Schnittstellen im Grafikteil

Das letzte Unterkapitel enthielt die Beschreibung der allgemeinen Grafik-Schnittstelle. Die hier vorliegende Implementation für X11/XView beinhaltet innerhalb des Grafikteils noch weitere Schnittstellen, die die Erweiterung der Ein- und Ausgabe und Portierungen auf andere

X11-Systeme erleichtern sollen. Für die Portierung auf andere Oberflächen ist daher dieses Unterkapitel irrelevant.

Wie bereits mehrfach erwähnt wurde, ist das Fenstermanagement mit XView programmiert worden, während das Zeichnen innerhalb der Fenster mit Plain-X11 Aufrufen auskommt. Da diese Zeichenaufrufe bei Portierungen auf andere X11-basierende Oberflächen wie z. B. Motif ähnlich sein sollten, macht es Sinn, eine Schnittstelle zwischen dem X11-Teil und dem XView-Teil zu definieren.

3.2.1 Datenstrukturen für X11-Teil

Neben den in Kapitel 2.3.3 definierten Datenstrukturen für die zu zeichnenden Grafikobjekte werden für die Schnittstelle zum X11-Teil einige Erweiterungen davon benötigt, die in der Datei `X_grafik.h` definiert sind.

Die Funktion für das Zeichnen der dreidimensionalen Sicht und des Rückspiegels wurde im X11-Teil zu einer zusammengefaßt. Der einzige Unterschied aus der Sicht der Zeichenfunktionen besteht darin, daß der Rückspiegel kein Fadenkreuz zum Zielen enthält. Die Datenstruktur mit den Grafikdaten für den Aufruf der gemeinsamen Funktion wurde um diese Information erweitert:

```
struct blickfelddaten
{
        struct objektdaten objekte;
        int fadenkreuz;
};
```

`objektdaten` entspricht dem Datentyp der für die allgemeinen Grafikdaten bereits definiert wurde. Ist das Flag `fadenkreuz` =1, so wird ein Zielkreuz gezeichnet.

Alle Informationen für das Zeichnen der Karte (Grundriß und Spieler darin) sind in eine übergreifende Datenstruktur zusammengefaßt:

```
struct grundrissdaten
{
        int anzlinien;
        struct kartenlinie *linien;
        int anzkreise;
        struct kartenkreis *kreise;
        int anzkreise_alt;
        struct kartenkreis *kreise_alt;
};
```

`linien`, `kreise` und `kreise_alt` sind Felder von den in Kapitel 2.3.3 beschriebenen Strukturen. `anzlinien`, `anzkreise` und `anzkreise_alt` enthalten die Anzahl der Elemente in diesen Feldern. Da der X11-Teil die Karte nicht laufend komplett neu aufbaut, sondern die Kreise für die Spieler löscht und an anderer Stelle neu zeichnet, werden neben den aktuellen Zeichendaten auch die des letzten Zeichenvorgangs übergeben.

Technische Informationen über die Fenster für die X11-Zeichenroutinen werden in einer weiteren Datenstruktur übergeben:

```
struct fenster
{
        Display *display;
        Window window;
        int breite, hoehe;
        unsigned farbtiefe;
};
```

Dabei dienen `*display` und `window` zur Identifikation des zu verwendenen Fensters. Ferner ist die aktuelle Größe des Fensters in Pixeln und die Anzahl der verfügbaren Farbebenen (Bitplanes) enthalten.

3.2.2 X11-Schnittstelle

Im Gegensatz zu der Beschreibung der allgemeinen Grafikschnittstelle liegt der Schwerpunkt der folgenden Erläuterung weniger auf der Arbeitsweise der Funktionen als auf Art und Zeitpunkt des Aufrufs, da bei dieser Schnittstelle das aufrufende Quelltextmodul getauscht werden kann.

```
int X_farben_init(Display *display, int screen);
```

wird aus `grafik_init` vor allen anderen X11-Routinen aufgerufen. `display` ist der Deskriptor zur Identifikation des X11-Servers und `screen` spezifiziert den Bildschirm dort. Die Funktion stellt fest, ob Farbdarstellung oder Schwarz-Weiß-Darstellung möglich ist, und definiert die Farben bzw. Muster. Falls dabei Fehler auftreten oder der Benutzer im Falle einer nicht ausreichenden Farbenanzahl Abbruch wählt, wird 1 als Rückgabewert übergeben. Dann ist das Programm unverzüglich zu beenden.

Die X-Zeichenfunktionen arbeiten mit privaten Datenstrukturen, die bei jedem Aufruf einer Zeichenroutine mit übergeben werden. Die folgenden Funktionen legen diese privaten Daten (`**X_daten`) an und geben sie wieder frei:

```
void X_fenster_init(void **X_daten, struct fenster *fenster);
void X_fenster_freigeben(void **X_daten, struct fenster *fenster);
void X_karte_init(void **X_daten, struct fenster *fenster);
void X_karte_freigeben(void **X_daten, struct fenster *fenster);
```

Die Initialisierungsfunktionen müssen in `grafik_init` für alle Fenster aufgerufen werden und die Freigabe-Funktionen beim Beenden der Grafik. Für das Karten-Fenster sind spezielle Funktionen erforderlich (`X_karte_init` und `X_karte_freigeben`), während für alle anderen `X_fenster_init` und `X_fenster_freigeben` aufgerufen werden muß. Bei jeder Änderung der Fenstergröße muß die alte Struktur freigegeben und danach neu initialisiert werden; dabei müssen die im folgenden Absatz beschriebenen Routinen zur Synchronisation der Grafikausgabe aufgerufen werden. Das Fenster und die Größe desselben werden anhand der übergebenen Strukturen vom Typ `fenster` ermittelt.

Wie auch bei der allgemeinen Grafik-Schnittstelle gibt es Funktionen zur Synchronisation des Zeichenvorgangs.

```
void X_zeichne_sync_anfang(Display *display);
void X_zeichne_sync_ende(Display *display);
```

display dient der Identifikation des X11-Servers. Die Funktionen werden bei Aufruf von zeichne_sync_anfang und zeichne_sync_ende (Bestandteil der allgemeinen Grafik-Schnittstelle) aufgerufen und außerdem überall dort wo intern X-Zeichenroutinen gestartet werden (z. B. bei einem außerordenlichen Neuzeichnen der Fenster).

Die Zeichenaufrufe aus der allgemeinen Grafik-Schnittstelle werden im Fensterverwaltungsteil zu den folgenden Aufrufe der X-Zeichenroutinen durchgereicht:

```
void X_zeichne_blickfeld(void *X_daten, struct fenster *fenster,
        void *daten);
void X_zeichne_karte(void *X_daten, struct fenster *fenster, void *daten);
void X_zeichne_grundriss(void *X_daten, struct fenster *fenster,
        void *daten);
void X_zeichne_kompass(void *X_daten, struct fenster *fenster, void *daten);
```

X_zeichne_blickfeld wird dabei sowohl für die Vorwärts- als auch für die Rückwärtsansicht (Rückspiegel) verwendet. Alle Zeichenfunktionen haben dieselbe Syntax: X_daten sind die privaten X-Daten, die mit den oben beschriebenen Funktionen angelegt werden. fenster ist die im letzten Abschnitt beschriebene Struktur für die technischen Informationen zu den X-Fenstern. Die zu zeichnenden Grafikdaten werden in daten übergeben. Diese Struktur enthält alle für das betreffende Fenster notwendigen Zeichendaten. Die Strukturen wurden z. T. in den allgemeinen Datenstrukturen (s. Abschnitt 2.3.3) und z. T. im letzten Abschnitt beschrieben.

Die Zuordnung der erwarteten Datenstrukuren für daten und die passende Funktion, mit der die X_daten angelegt werden müssen, soll folgende Tabelle erläutern:

Fenster	Zeichenfunktion	daten	anlegen von X_daten
3D-Ansicht	X_zeichne_blickfeld	blickfelddaten	X_fenster_init
Rückspiegel	X_zeichne_blickfeld	blickfelddaten	X_fenster_init
Kartengrundriß	X_zeichne_grundriss	grundrissdaten	X_karte_init
Spieler auf Karte	X_zeichne_karte	grundrissdaten	—
Kompaß	X_zeichne_kompass	blickrichtung	X_fenster_init

Alle X-Zeichenroutinen können mehrfach für verschiedene Fenster verwendet werden – bei X_zeichne_blickfeld wird bereits davon Gebrauch gemacht. Aus diesem Umstand ergibt sich die Notwendigkeit, die Syntax der Aufrufe zu vereinheitlichen und die privaten X-Daten mit zu übergeben.

Der X11-Teil erkennt selbständig nicht, wann ein außerordentliches Neuzeichen des Bildschirminhalts erforderlich ist, z. B. beim Öffnen eines Fensters oder Verändern der Fenstergröße. Neben der Verwaltung der Fenster, Menüs, Steuerung etc. ist der Fensterverwaltungsteil auch dafür verantwortlich.

3.2.3 Schnittstelle für Steuerung

Da es neben der Tastatursteuerung eine ganze Reihe anderer Geräte zur Eingabe gibt, wie Joysticks, Maus usw., die z. T. sehr systemabhängig sind, macht es Sinn, eine kleine Schnittstelle vorzusehen, die den Tausch der Quelltextkomponente xv_tasten.c im XView-Teil einfach ermöglicht. Wie der Dateiname des Headerfiles xv_einaus.h, der die Schnittstelle zum XView-Teil definiert, schon andeutet, ist diese Schnittstelle eine Besonderheit der XView-Implementation.

Neben dem Aufruf von `eingabe_abfragen` aus der allgemeinen Grafik-Schnittstelle (s. Abschnitt 3.1.4) kommen zwei weiter hinzu:

```
void xv_eingabe_init(Frame hauptframe, Panel panel);
void xv_eingabe_ende(void);
```

Die Initialisierung wird aus `grafik_init` nach dem Anmelden der Fenster aufgerufen. Hier können Event-Routinen eingefügt oder Speicher belegt werden. Die Ende-Funktion wird aus `grafik_ende` aufgerufen.

3.2.4 Schnittstelle für Soundausgaben

Für die Unterstützung von Sounds gilt ebenfalls, daß sie unabhängig von XView sehr systemspezifisch sein kann. Auch hier ist eine kleine Schnittstelle innerhalb des XView-Teils vorgesehen, die es erlauben sollte, `xv_ereignisse.c` gegen eine andere Komponente auszutauschen.

Die Funktion `ereignisse_darstellen` genügt der allgemeinen Grafik-Schnittstelle (s. Abschnitt 3.1.4). Die zusätzlichen Funktion der XView-Schnittstelle werden analog den oben beschriebenen Funktionen für die Steuerung aufgerufen.

```
void xv_ereignisse_init(Frame hauptframe, Panel panel);
void xv_ereignisse_ende(void);
```

3.3 Globale Funktionen

Die verschiedenen Komponenten des Clients greifen auf eine Reihe von globalen Funktionen zu, die der Basisteil zur Verfügung stellt. Diese Funktionen werden z. T. auch in den anderen Programmen mitverwendet. Daher darf eine Beschreibung an dieser Stelle nicht fehlen.

3.3.1 Speicherfunktionen

Da die Programmteile intensiv mit variablen Speicherfeldern operieren, gibt es einige kleine Funktionen, die die Handhabung etwas vereinfachen sollen.

Zum Anmelden und Freigeben von Speicher gibt es:

```
void speicher_belegen(void **zeiger, int groesse);
void speicher_vergroessern(void **zeiger, int groesse);
void speicher_freigeben(void **zeiger);
```

Dabei wird nur `malloc`, `realloc` und `free` aufgerufen. Falls die Operation fehlschlägt, wird eine Fehlermeldung ausgegeben und das Programm beendet.

Zur Behandlung von Listen bzw. variablen Feldern werden folgende Funktionen angeboten:

```
void liste_initialisieren(int *anzahl, void **liste);
void liste_verlaengern(int *anzahl, void **liste, int element_groesse);
void liste_kopieren(int *anzahl_neu, void **liste_neu, int *anzahl,
        void **liste, int element_groesse);
void liste_freigeben(int *anzahl, void **liste);
```

Man kann eine Liste neu anlegen, sie um ein Element verlängern, eine Liste in eine nicht initialisierte neu kopieren und sie wieder löschen. In `anzahl` wird die Anzahl der Elemente in der Liste übergeben und zurückgegeben. `**liste` ist ein Zeiger auf die Liste und `element_groesse` die mit `sizeof` ermittelte Größe eines Elements.

3.3.2 Fehler

Alle Programmteile rufen bei schweren Fehlern, die ein Weiterlaufen des Programms nicht gestatten, die Funktion

```
void uebler_fehler(char **meldung, char *knopf);
```

auf. Die Syntax des Aufrufs entspricht dem der Funktion `uebler_fehler_anzeigen` aus der Grafikschnittstelle (vgl. Abschnitt 3.1.5). `*meldung` ist ein Feld von Strings, das die auszugebende Fehlermeldung beinhaltet. Der String `knopf` für einen Text auf dem Bestätigungsknopf ist optional. Falls `NULL` übergeben wird, findet ein Default-Text Verwendung.

3.3.3 Systemabhängige Funktionen

Das Quelltextmodul `system.c` enthält Funktionen, die sehr systemnah sind bzw. die zur Anpassung an andere Systemvarianten dienen. Daher ist bei einer Portierung auf diesen Quelltextteil besonders zu achten.

```
void (*signal(int signum, void (*handler)()))();
```

stellt auf System V Unix Umgebungen eine `signal`-Routine mit einer Semantik wie bei BSD-Systemen üblich zur Verfügung.

```
int max_deskriptor_anzahl(void);
```

gibt die maximale Anzahl von Deskriptoren zurück (wird für die Verwendung von `select` benötigt).

```
int puffer_lesen(int deskriptor, int puffer_laenge, void *puffer);
int puffer_schreiben(int deskriptor, int puffer_laenge, void *puffer);
```

lesen und schreiben einen Puffer durch einen Deskriptor. Es werden soviele Daten gelesen bzw. geschrieben, bis entweder ein Fehler auftritt oder der Puffer komplett übertragen wurde. Der Rückgabewert enthält die Anzahl der übertragenen Bytes (also bei Erfolg `puffer_laenge`) oder -1, falls ein Fehler auftritt.

Der Umgang mit *Timern* wird durch weitere Funktionen erleichtert:

```
void timer_starten(int ms);
void timer_stoppen(void);
int timer_restzeit(void);
void timer_abwarten(void);
void *timer_retten(void);
void timer_restaurieren(void *zustand);
```

Der Timer wird mit einer Zeit in Millisekunden gestartet (`timer_starten`) und löst bei Ablauf ein `SIGALRM` aus. Man kann ihn jedoch vorzeitig stoppen und das Auslösen des Alarms damit verhindern (`timer_stoppen`). `timer_restzeit` liefert die verbleibende Zeit und `timer_abwarten` kehrt erst wieder zurück, wenn der Timer abgelaufen ist. Bei Aufruf von `timer_retten` wird der Timer angehalten und zusätzlich der aktuelle Zustand zurückgegeben. Mit `timer_restaurieren` kann man den Timer dann an der abgebrochenen Stelle wieder aufsetzen.

3.3.4 Sonstige Funktionen

In `global.c` bzw. `global.h` verbergen sich weitere kleine Funktionen. Makros werden für `min`, `max` und `sqr` (quadrieren) angeboten. Ferner gibt es eine Integer-Wurzelfunktion

```
long wurzel(unsigned long x, int runden);
```

Das Flag `runden` mit den Werten `W_ABRUNDEN`, `W_RUNDEN` und `W_AUFRUNDEN` gibt an, wie die Rundung vorgenommen werden soll.

Um auf allen Systemen einen einheitlichen Zufallsgenerator verwenden zu können, gibt es

```
long zufall(void);
void zufall_init(unsigned long startwert);
```

Der Zufallsgenerator gibt Zahlen von 0 bis `LONG_MAX` zurück. Aus demselben Wertebereich kann ein Startwert vorgegeben werden.

Kapitel 4

Netzwerk-Protokoll

Ein zentraler Punkt in der Aufgabenstellung ist die Entwicklung eines Netzwerkprotokolls, das alle Anforderungen, die ein Spiel wie *iMaze* stellt, in einer Unix-Umgebung unter TCP/IP erfüllt. In der Einleitung (Kapitel 1.4) wurde schon angedeutet, wie diese Anforderungen und das darauf basierende Konzept aussehen.

4.1 Aufgaben

Wir hatten festgestellt, daß eine Client-Server-Lösung am besten geeignet ist. Der Server erfüllt alle Aufgaben der Spielverwaltung und Spielsteuerung, während die Clients jeweils einem Spieler seine aktuelle Situtation anzeigen und seinen nächsten Zug zum Server schicken. Die Kommunikation zwischen diesen Komponenten wird über eine Netzwerkverbindung abgewickelt. Das darauf ablaufende Netzwerk-Protokoll muß folgende Aufgaben erfüllen:

1. Aufbau einer Verbindung zwischen Client und Server; gegenseitige Identifikation.

2. Der Server muß dem Client den Aufbau des Spielfeldes und weitere Parameter mitteilen. Da das Labyrinth statisch ist, muß dies nur einmal erfolgen.

3. Im Spiel teilt der Server den Clients die aktuelle Spielsituation mit und dem einzelnen Spieler weitere Informationen, falls ihn besondere Ereignisse betreffen.

4. Die Clients schicken dem Server ihre nächsten Züge.

5. Beenden des Spiels und weitere Ausnahmesitutationen müssen ausgetauscht werden.

6. Die Synchronisation der Clients und damit des ganzen Spiels erfolgt über das Protokoll.

7. Das Protokoll sollte unterschiedliche Übertragungsqualitäten im Internet abfedern und grundsätzlich Chancengleichheit für alle Mitspieler gewährleisten.

Hinzu kommt, daß das Protokoll für spätere Versionen erweiterbar sein muß. Solche Erweiterungen könnten z. B. Steuerung des Servers, Auswahl einer von mehreren am Server laufenden Spielrunden, Übertragung von weiteren Information wie Punkten, Übertragungen von zusätzlich wählbaren Spieloptionen, etc. sein. Das Protokoll wurde also entsprechend großzügig erweiterbar angelegt. Zwar ist auch denkbar, bei Veränderungen am Spiel das Protokoll ebenfalls zu ändern. Ein solches Vorgehen sollte jedoch möglichst vermieden werden, damit das Zusammenspiel geringfügig unterschiedlicher Client- und Server-Versionen nicht ausgeschlossen wird.

Zwar wurde das hier beschriebene Protokoll für Unix und TCP/IP entwickelt, ein gutes Anwendungsprotokoll ist aber von den darunter liegenden Übertragungsschichten unabhängig und kann auch auf Basis anderer Netzwerkprotokolle als TCP/IP, wie z. B. Novell-Netz, eingesetzt werden. Das *iMaze*-Netzwerk-Protokoll wurde entsprechend ausgelegt.

Die weitere Beschreibung des Protokolls ist deshalb in zwei Abschnitte unterteilt. Der erste wird den Ablauf, die übertragenen Daten und bei der Realisierung zu beachtende Anforderungen beschreiben. Im zweiten Teil wird dann die konkrete Realisierung auf Basis von TCP/IP beschrieben.

4.2 Definition des Protokolls

Der Aufbau einer Verbindung vom Client zum Server und die korrekte Beendung derselben ist systemabhängig und Aufgabe tiefer liegender Netzwerkschichten. Hier muß auch die tatsächliche Übertragung der Daten realisiert sein. Die Daten sind in Pakete verpackt; daß heißt aber nicht, daß die Übertragung auf unteren Netzwerkschichten zwingend paketorientiert ablaufen muß. In diesem Teil wird nur der Aufbau der Daten-Pakete behandelt und auf Besonderheiten hingewiesen, die bei der Übertragung der Pakete beachtet werden müssen.

Alle Daten in den Paketen sind byteweise gespeichert. Jedes Daten-Paket beginnt mit zwei Bytes, in denen die Länge des Pakets in Bytes steht, und enthält eine *Kennung*, die den Inhalt eindeutig kennzeichnet. Pakete können auch nur aus einer Kennung bestehen, wenn sie die eigentliche Information schon zum Ausdruck bringt. Die Kennungen sind in der Datei `protokoll.h` definiert. Allen Zahlenwerten ist dort ein Name zugeordnet, der hier verwendet wird.

Das Protokoll ist in zwei Phasen gegliedert. In der ersten Phase werden alle einmalig zu übertragenen Informationen und Daten ausgetauscht, wie der Aufbau des Spielfeldes. Die erste Phase heißt auch *Prolog-Phase*. Sie endet mit dem Beginn des eigentlichen Spiels. Wenn das Spiel läuft, schickt der Server in der zweiten Phase – oder auch *Spiel-Phase* – den Clients die aktuellen Spielsituationen und erhält als Antwort die einzelnen Spielzüge. Eine Rückkehr in die erste Phase findet nicht statt.

4.2.1 Prolog-Phase

In der ersten Phase ist wichtig, daß alle Daten die Gegenseite erreichen und von ihr bearbeitet werden. Daraus ergeben sich folgende Anforderungen:

1. Alle Pakete müssen bei der Gegenseite in der richtigen Reihenfolge ankommen.

2. Fehler in den Paketen müssen ausgeschlossen werden.

3. Die Geschwindigkeit spielt eine untergeordnetet Rolle.

4. Die Übertragung erfolgt im ständigen Wechsel. Alle Pakete werden von der Gegenseite beantwortet.

Alle Pakete in dieser Phase beginnen mit zwei Byte, die die Paketlänge enthalten, wobei das erste Byte $laenge/256$ (*highbyte*) und das zweite $laenge\%256$ (*lowbyte*) ist. $laenge$ ist die

Anzahl der Bytes im Datenteil, also ohne die Längenbytes selbst. Dadurch ergibt sich eine maximale Länge des Datenbereichs von $2^{16} - 1$. Sind keine weiteren Daten angehängt, ist $laenge = 1$. Dann bestehen die Daten nur aus der Kennung.

Beide Teilnehmer der Verbindung senden abwechselnd. Hat ein Teilnehmer ein Paket gesendet, wartet er auf eine Antwort von der Gegenseite und sendet erst danach ein neues Paket. Bei jedem Paket außer dem ersten wird in der Kennung codiert, ob das zuletzt von der Gegenseite erhaltene Paket verstanden und gebraucht wurde oder nicht. Dafür ist das oberste Bit in der Kennung reserviert.

`ignoriert`-Bit	Bedeutung
0	letztes Paket wurde verstanden und ausgewertet
1	letztes Paket wurde nicht verstanden oder nicht benötigt

Es können so nur Kennungen von 0 bis 127 vergeben werden.

Die Reihenfolge, in der die Daten gegenseitig übertragen werden, ist nicht festgelegt. Der Client beginnt die Übertragung, üblicherweise mit einem Paket `PT_PROGRAMM_NAME`, das eine eindeutigen Identifikation enthält (z. B. `iMaze xview client JC/HUK 1.0`). Der Server antwortet mit einem Paket von ihm usw. Während Informationen zur Identifikation der Teilnehmer u. ä. unaufgefordert übertragen werden, gibt es für das Spielfeld ein Paket `PT_WILL_LABYRINTH`, mit dem der Client die Übermittlung von `PT_LABYRINTH` explizit anfordert. Solche Pakete, zu denen Anfragen existieren, dürfen nur auf Anforderung verschickt werden und zwar in dem direkt darauf folgenden Paket. Hat ein Teilnehmer keine Daten zu übertragen und keine Anfragen mehr zu stellen, so antwortet er mit `PT_LEER` bzw. der Client mit `PT_BEREIT_FUER_SPIEL`. `PT_LEER` mit gesetztem `ignoriert`-Bit darf auch gesendet werden, wenn eine Anfrage nicht beantwortet werden kann. Der Client darf auf `PT_LEER` vom Server nicht wieder mit `PT_LEER` antworten (Endlosschleife). Der Server darf auf `PT_BEREIT_FUER_SPIEL` nicht mehr mit `PT_LEER` antworten, sondern mit `PT_SPIEL_BEGINNT`, wenn auch er alle Daten ausgetauscht hat. Dieses Paket wird als einziges nicht beantwortet, sondern beendet unwiderruflich die erste Phase des Protokolls. Jeder Teilnehmer kann im Fehlerfall oder aus beliebigen anderen Gründen ein `PT_ABBRUCH` senden und die Verbindung damit abbrechen.

In einer Tabelle noch einmal alle in Version 1.0 verwendeten Pakete:

Sender	Kennung	mit Daten?	Bedeutung
beide	`PT_LEER`	nein	keine Daten
	`PT_ABBRUCH`	nein	Abbruch der Verbindung
	`PT_PROGRAMM_NAME`	ja	String zur Identifikation des Programms
Client	`PT_BEREIT_FUER_SPIEL`	nein	alle Daten bekommen
	`PT_WILL_LABYRINTH`	nein	Client fordert Spielfeld an
Server	`PT_SPIEL_BEGINNT`	nein	Server schaltet auf Spiel
	`PT_LABYRINTH`	ja	Server schickt Labyrinth-Daten

Die Labyrinth-Daten sind vergleichbar aufgebaut, wie im Kapitel 5.3 für das Dateiformat beschrieben. Hier nur eine kurze Zusammenfassung:

Das Spielfeld ist in maximal $127 * 127$ quadratische Felder unterteilt, die eine feste Außenlänge haben, nämlich die minimale Gangbreite. Zu jedem Feld ist für die vier Seiten, betrachtet vom Feldinneren, die Farbnummer in einem Byte gespeichert. Im Gegensatz zum Server erhält der Client nur so viel Informationen, wie zur Darstellung notwendig sind, nämlich die Farbe der Wände und Türen, bzw. 0 falls keine Wand oder eine unsichtbare Wand vorliegt. Die Daten beginnen mit einem Byte mit der Anzahl der Felder in x-Richtung und einem für die y-Richtung. Darauf folgen zeilenweise beginnend mit der nordwestlichen Ecke die vier Bytes pro Feld in der Reihenfolge: Nord, West, Süd, Ost.

In späteren Versionen können weitere Kennungen hinzukommen. Jeder Verbindungsteilnehmer muß alle Pakete beantworten, ggf. mit `ignoriert`, falls ihm die Kennung unbekannt ist. Das bedeutet auch, daß jeder Teilnehmer darauf vorbereitet sein muß, auf ein vom ihm verschicktes Paket ein `ignoriert` zu erhalten, notfalls kann er dann abbrechen.

4.2.2 Spiel-Phase

Die Spiel-Phase ist ganz auf Geschwindigkeit ausgelegt. Schließlich wird mit dem Protokoll der Zeittakt übertragen. Deshalb sind die Anforderungen bei der Implementierung ganz andere:

1. Die Daten müssen so schnell wie möglich übertragen werden. Verzögerungen sind zu vermeiden.
2. Dabei kann in Kauf genommen werden, daß einige Pakete nicht ankommen. Sie sind mit dem nächsten Zeittakt sowieso überholt.
3. Pakete deren Inhalt überholt ist, werden ignoriert.
4. Die Verbindung darf keinesfalls blockiert werden.
5. Daten, die ankommen, müssen fehlerfrei sein.
6. Es muß feststellbar sein, wie lange die Pakete unterwegs sind.
7. Mehrere Pakete können gleichzeitig oder in falscher Reihenfolge ankommen.
8. Die Programme müssen jederzeit mit dem Ankommen neuer Pakete rechnen und diese schnellstmöglich abarbeiten.

Ist die Verbindung von Natur aus gleich schnell und sicher, wie beim seriellen Ring von MidiMaze, so kann die Phase 2 mit der Phase 1 zusammenfallen. Die folgenden Überlegungen sind dann ebenfalls unwesentlich.

Der Server schickt den Clients in festgelegten Abständen (etwa 10 mal pro Sekunde) den aktuellen Spielstand. Trifft die Information bei einem Client ein, so antwortet er mit seiner nächsten Bewegung (bei einem interaktiven Client mit den seit dem letzten Paket gemachten Eingaben) und startet das Neuzeichnen oder Neuberechnen. War die letzte Auswertung noch nicht fertig, so kann sie verworfen werden, da nur der aktuelle Stand zählt. Dasselbe gilt, wenn beim Client mehrere oder veraltete Pakete eintreffen. Für den Server sind auch ältere Pakete interessant, da sonst Eingaben verloren gehen könnten. Andererseits dürfen nicht beliebig viele Pakete zusammengefaßt werden, da daraus Verfälschungen entstehen können. Pro Zeittakt können maximal drei Pakete vom gleichen Typ zusammengefaßt werden. Benötigt die Antwort auf ein Paket länger als 10 Zeittakte, so wird der Spieler auf Pause gesetzt, damit er für seine Mitspieler nicht eine wehrlosen „Zielscheibe“ darstellt. Kommen länger als 1000 Zeittakte keine Pakete, so muß davon ausgegangen werden, daß die Verbindung zusammengebrochen ist. Der Client darf nur so viele Pakete mit neuen Bewegungen schicken, wie er Pakete vom Server erhält. Der Server prüft die Einhaltung dieser Bedingung mit Seriennummern, da sonst der Zeittakt unterlaufen werden könnte. Auch wenn der Spieler sich nicht bewegt, muß der Client ein Paket schicken, damit der Server sicher gehen kann, daß die Verbindung noch besteht.

Die Pakete der Spielphase, also die Einheiten, die pro Zeittakt verschickt werden, können mehrere Informationsblöcke beinhalten, die den Datenpaketen im letzten Abschnitt entsprechen. Das erste Byte eines Blocks enthält eine Kennung. Ein Block kann nur aus der Kennung bestehen oder noch weitere Daten beinhalten. Welcher Fall zutrifft, wird anhand der Kennung mit dem Makro `DATENLAENGE_VARIABEL` entschieden. Besteht ein Block nicht nur aus der Kennung, so steht die Länge des folgenden Datenteils im zweiten Byte.

Der Client überträgt die Eingaben seit dem letzten Paket. Wir bezeichnen sie auch als *Signale*. Die Signale sind in `signale.h` codiert. Pro Übertragung kann jedes Signal nur einmal gesendet werden. Nach `BK_SIGNALE` folgen die Nummern der betätigten Eingabesignale. Diese Liste kann auch leer sein.

Der Server schickt normalerweise die Position des angesprochenen Clients im Labyrinth (`BK_SPIELER_POS`) und seines Schusses (`BK_SPIELER_SCHUSS_POS`), die Positionen der anderen im Spiel befindlichen Mitspieler (`BK_GEGNER_POS`) und deren Schüsse (`BK_GEGNER_SCHUSS_POS`) sowie aufgetretene *Ereignisse* (`BK_EREIGNISSE`), die in `ereignisse.h` definiert sind. Da die Ereignisse als einzige Information nicht verlorengehen sollen, werden sie so oft in folgenden Paketen wiederholt, bis sichergestellt ist, daß der Client die Information erhalten hat. Diese nachgeschickten Ereignisse haben dann die Kennung `BK_ALTE_EREIGNISSE`. Im Fall eines Abschusses wird die Farbe des „Verursachers“ übertragen (`BK_ABGESCHOSSEN_DURCH`) solange der Spieler tot ist.

Sind keine Informationen vorhanden, so können einzelne Blöcke entfallen. Die Reihenfolge ist nicht festgelegt. In einem Paket ist aber jeder Blocktyp nur einmal enthalten.

Will ein Client das Spiel verlassen, so sendet er `BK_ENDE_SIGNAL`. Der Server antwortet mit `BK_ENDE_DURCH_CLIENT` oder mit `BK_ENDE_DURCH_SERVER`, falls er selbst das Ende bewirkt hat.

In dieser Tabelle sind noch einmal alle Kennungen aufgeführt:

Sender	Kennung	mit Daten?	Bedeutung
Client	`BK_ENDE_SIGNAL`	nein	Bitte um Abmeldung
	`BK_SIGNALE`	ja	Eingabesignale
Server	`BK_ENDE_DURCH_SERVER`	nein	Spielende vom Server veranlaßt
	`BK_ENDE_DURCH_CLIENT`	nein	Abmeldung bestätigt
	`BK_SPIELER_POS`	ja	Position des eigenen Spielers
	`BK_GEGNER_POS`	ja	Position der Mitspieler
	`BK_SPIELER_SCHUSS_POS`	ja	Schuß-Position des eigenen Spielers
	`BK_GEGNER_SCHUSS_POS`	ja	Schüsse der Mitspieler
	`BK_ABGESCHOSSEN_DURCH`	ja	Farbe des Schützen
	`BK_EREIGNISSE`	ja	aufgetretene Ereignisse
	`BK_ALTE_EREIGNISSE`	ja	wiederholte Ereignisse

Das Spielfeld ist bekanntlich in Felder unterteilt. Diese Unterteilung bildet bei der Positionsangabe die grobe Rasterung. Die Felder sind noch einmal in ein Feinraster unterteilt, das $256*256$ Punkte groß ist. Hinzu kommt die Farbe des Spielers und der Blickwinkel. Blickwinkel werden in *iMaze* in Schritten von 0 (Norden) bis 63 gegen den Uhrzeigersinn angegeben. Die Position eines Spielers ist sechs Byte lang: x-Position grob, x-Position fein, y-Position grob, y-Position fein, Farbe, Blickwinkel. Bei Schüssen entfällt der Blickwinkel. Die Positionen der Mitspieler und deren Schüße werden alle aneinandergehängt und im Block übertragen. Die Anzahl ergibt sich aus der Paketlänge und der Anzahl von Bytes pro Position (z. B. bei den Schüssen: $anz = (laenge - 1)/5$). Die Reihenfolge, in der die Positionen im Block stehen, ist nicht festgelegt.

In der Version 1.0 überträgt der Server an jeden Client die Informationen aller Mitspieler. In späteren Versionen ist damit zu rechnen, daß nur noch die nötigsten Daten übertragen werden, also die Spielerpositionen der gerade sichtbaren Spieler.

Informationsblöcke, deren Inhalt unbrauchbar oder deren Kennung unbekannt ist, werden ignoriert.

4.3 Realisierung mit TCP/IP

TCP/IP stellt im wesentlichen zwei unterschiedliche Protokolle zur Auswahl:

TCP (*Transmission Control Protocol*) ist ein verbindungsorientiertes Kommunikationsprotokoll. D. h. es wird für die Zeit der Kommunikation eine feste Verbindung aufgebaut. TCP garantiert, daß alle Daten in unveränderter Reihenfolge und fehlerfrei die Gegenseite erreichen. Bei Unterbrechungen können die beteiligten Programme aber blockiert werden. Typische Protokolle basierend auf TCP sind *FTP* und *telnet*.

UDP (*User Datagram Protokoll*) ist dagegen ein verbindungsloses Kommunikationsprotokoll. Eine feste Verbindung besteht nicht, stattdessen werden Pakete verschickt. Bei UDP ist lediglich garantiert, daß ankommende Pakete fehlerfrei sind. Pakete können aber verloren gehen oder in vertauschter Reihenfolge ankommen. Der Vorteil besteht in höherer Geschwindigkeit und Vermeidung von Blockierungen der Programme. Der bekannteste Vertreter von UDP-Diensten ist wohl das *Network File System* (NFS).

iMaze verwendet beide Protokolle. In der ersten Phase, in der die Verbindung hergestellt und wichtige Spielparameter übertragen werden, liegt der Schwerpunkt auf der Übertragungssicherheit, während in der Spielphase die Geschwindigkeit an erster Stelle steht.

4.3.1 Prolog-Phase

Da jederzeit neue Clients zu einem laufenden Spiel hinzukommen können, muß der Server gleichzeitig beide Protokollarten behandeln können. Deshalb teilt er sich nach dem Start in zwei Prozesse, von denen der eine neue Verbindungen über TCP annimmt und die erste Protokollphase abwickelt und der anderer die laufende Spielrunde über UDP steuert. Beide Unterprozesse des Servers kommunizieren ebenfalls untereinander. Dazu wird ein sog. „*socketpair*“ verwendet.

Der Server lauscht auf einem bestimmten TCP-Port (default: Port Nummer 5323) und wartet auf Clients, die sich neu anmelden wollen. Baut ein Client eine Verbindung zu diesem Port auf, so spaltet der Server einen weiteren Unterprozeß ab, der darüber die erste Protokollphase abwickelt. Nach Beendigung der ersten Phase mit dem Serverpaket `PK_SPIEL_BEGINNT` sendet der Server die Adresse und den Port des UDP-Teils, auf dem die Spielrunde läuft. Diese Information ist 6 Byte lang und wird in Netzwerk-Byteorder übertragen. Die ersten vier Bytes enthalten die numerische Internet-Adresse, das fünfte Byte $portnr/256$ und das letzte *portnr*%256. Die erneute Übertragung der Netzadresse kann sinnvoll sein, wenn der Spielrunden-Prozeß über eine andere Adresse erreichbar ist.

Beispiel: Das Spiel laufe auf `139.174.2. 11:4567`. Dann wird folgendes übertragen:

Byte	Inhalt
1	139
2	174
3	2
4	11
5	17 (4567/256)
6	215 (4567%256)

Der Client schickt seine Adresse und Portnummer in derselben Codierung zurück. Der Server gibt die Antwort über die interne Verbindung an den Spielteil weiter, der den neuen Teilnehmer in seinen Verteiler einträgt und ab sofort mit dem Spielstand versorgt. Die TCP-Verbindung und der Unterprozeß des Servers werden jetzt beendet.

4.3.2 Spiel-Phase

Als Anforderung für die zweite Phase ist eine Auswertung der Laufzeit aller Pakete gefordert. Ferner soll unterbunden werden, daß ein Client den Zeittakt durch ununterbrochenes Senden von Steuersignalen unterläuft. Deshalb beginnen alle Pakete in dieser Phase mit einer zwei Byte langen Seriennummer (zuerst $nr/256$, dann $nr\%256$). Der Server zählt die Seriennummer mit dem Zeittakt hoch. Der Client muß mit seiner Antwort die Seriennummer zurückschicken, die er vom Server erhalten hat. Der Server kann dann feststellen, wie lange ein Paket von ihm und die Antwort darauf unterwegs waren und bei Unterbrechungen geeignete Maßnahmen einleiten. Der Server ignoriert alle Antworten deren Seriennummer über die aktuelle hinausgeht. Damit ist sichergestellt, daß kein Client mehr Antworten senden kann, als Spielstände übertragen wurden.

Die Verbindungen im Internet können qualitativ sehr schwanken. Während eines Zeittakts kommen manchmal von einem Client keine Antwortpakete an, dafür im nächsten Takt gleich mehrere. Um Spieler mit schlechten Verbindungen nicht zu sehr zu benachteiligen, werden bis zu drei Steuersignale eines Clients zusammengefaßt. Er macht dann einen kleinen Sprung. Kommen noch mehr Pakete an, so gehen die ältesten von ihnen verloren. Würde man sie auch mit einbeziehen, bestände die Gefahr einer Verfälschung des Spielablaufs.

Ist beim Server die Wartepause des Zeittaks abgelaufen, so wertet er alle bis dahin von den Clients eingetroffenen Steuersignale aus und berechnet eine neue Spielsituation, die er dann an die Clients schickt. Danach wartet er wieder bis zum Ablauf der Taktzeit. Der Client wartet ständig auf neue Pakete, speichert deren Inhalt zwischen und schickt sofort eine Antwort mit den seit dem letzten Paket erfolgten Eingaben oder ggf. auch ein leeres Paket. Erst danach wertet er das Paket aus oder zeichnet den Bildschirm neu.

Antwortet ein Client sehr lange nicht mehr (länger als 1000 Zeittakte) oder hat er den Wunsch geäußert aufzuhören, so wird er nach der Übermittlung einer Meldung aus der Liste der zu versorgenden Clients gestrichen. Die Verbindung ist damit beendet.

Kapitel 5

Labyrinth-Dateiformat

Die Dateien, die das Aussehen der Spielfelder enthalten, werden mit einem Editor oder einem anderen Programm erstellt und dann vom Server geladen. Da hier unterschiedliche Programme von möglicherweise verschiedenen Autoren beteiligt sind, muß das Format dieser Dateien genau definiert werden, um Schwierigkeiten beim Austausch der Dateien zu verhindern.

5.1 Binär oder Text?

Von den Versionen 1 und 2 von MidiMaze sind zwei Varianten eines solchen Formats vorgegeben. MidiMaze 1 verwendet eine Text-Datei, in der alle Komponenten in Buchstaben codiert sind, während MidiMaze 2 ein nicht dokumentiertes Binärformat verwendet. Die erste Variante hat den Vorteil, daß die Spielfelder mit jedem Texteditor erstellt werden können. Dies ist aber auch ein Nachteil, weil ein Texteditor den korrekten Aufbau eines Labyrinths nicht kontrollieren kann. Fehler in der Datei sind aufgrund der wenig überschaubaren Texte-Dateien unvermeidlich. (Man erinnere sich an die typische Meldung: *„MidiMaze: Level booboo. Who cares?“*)

Weil der Aufbau der Labyrinthe in *iMaze* noch deutlich komplexer ist, haben wir uns ebenfalls für eine Binär-Datei entschieden. Sie ist nicht von verschiedenen Text-Zeichencodierungen abhängig und muß bei Datenübertragung nicht konvertiert werden. Da das Format von MidiMaze 2 nicht bekannt war und die Fähigkeiten von *iMaze* auf diesem Gebiet über die von allen MidiMaze-Varianten hinausgehen, wurde ein neues Format definiert, das an die interne Darstellung der Labyrinthe angelehnt ist. (s. Kapitel 2.3.1) Ein Konverter für die alten Dateiformate sollte aber leicht realisierbar sein.

Um auch in der Unix-Shell Informationen über eine Spielfeld-Datei gewinnen zu können, ist an die eigentlichen Daten ein Text als Kommentar angehängt. Der Inhalt dieses Kommentars wird aber nicht ausgewertet.

5.2 Grundlegende Vereinbarungen

Bezüglich der Formate der Labyrinth-Dateien gelten folgende Festlegungen:

1. Das Spielfeld wird vom Server gelesen und an alle angeschlossenen Clients verschickt.
2. Die Labyrinthe sind statisch. Sie können in einem laufenden Spiel nicht verändert werden.

3. Jedes Spielfeld ist in einer eigenen Datei gespeichert. Der Aufbau dieser Dateien ist genauer zu spezifizieren.
4. Die Dateien erhalten eine Versionsnummer. Nach Veränderungen der Spezifikation wird eine neue Nummer vergeben.
5. Text, der in den Dateien enthalten ist, wird im 7-Bit ASCII-Code gespeichert. Er dient nur als Kommentar und hat keine funktionelle Bedeutung.
6. Die binären Daten werden Byte-weise gespeichert, so daß sie von der Byte-Order verschiedener Systeme unabhängig sind.
7. Dateien, die sich nicht exakt an eine Spezifikation halten, können als fehlerhaft abgewiesen werden.
8. Programme, die mit diesen Dateien arbeiten, müssen neben einer aktuellen auch alle alten Versionen korrekt lesen können.

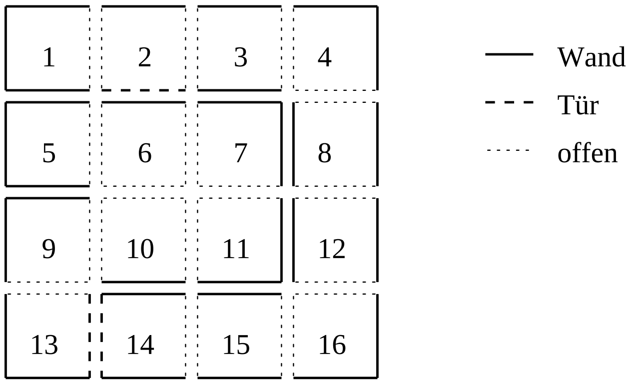
Nach diesen grundlegenden Überlegungen soll jetzt die Version 1 des Dateiformates definiert werden.

5.3 Format der Version 1

Ein Labyrinth ist in Felder unterteilt. Ein Feld ist quadratisch und hat festgelegte Ausmaße. Sie entsprechen der minimalen Breite eines Ganges. Die dieses Feld umgebenen vier Seiten werden aus der Sicht des Feldmittelpunktes beschrieben. Wände sind unendlich dünn.

Eine zwei Felder trennende Wand kann aufgrund der Tatsache, daß sie einmal aus der Sicht des einen Feldes und einmal aus der Sicht des anderen Feldes beschrieben wird, unterschiedliches Aussehen oder Eigenschaften abhängig vom Standpunkt haben. Auch lange, das ganze Labyrinth durchtrennde Wände sind aus Einzelsegmenten zusammengesetzt, die sich jeweils aus der Seite eines Feldes ergeben.

Folgende Abbildung zeigt schematisch ein $4 * 4$ Felder großes Labyrinth. Die einzelnen Felder wurden zur Veranschaulichung auseinandergerückt.



Zu jeder Seite eines Feldes werden Flags über die Eigenschaften und die Farbe gespeichert. Eigenschaften sind:

unbegehbar: Diese Seite kann von einem Spieler in Blickrichtung nicht durchlaufen werden.

schusssicher: Diese Seite kann in Blickrichtung nicht durchschossen werden.

Es gibt 16 Farben, die wie folgt zugeordnet sind:

Farbe	Zuordnung
0	durchsichtig von innen nach außen
1 ... 7	Wände mit unterschiedlichen Farben
8 ... 15	Türen mit unterschiedlichen Farben

Dadurch ergeben sich eine Reihe von Kombinationen:

unbegehbar	schusssicher	Farbe	Bedeutung
0	0	0	offen
1	1	1 ... 7	Wand
1	1	0	unsichtbare Wand
0	1	8 ... 15	Tür

Alle anderen Kombinationen sind in dieser Spezifikation nicht vorgesehen. Dadurch, daß eine Trennung zwischen zwei Feldern von beiden Seiten unterschiedlich betrachtet wird, gibt es viele Kombinationen, mit denen man einseitig durchsichtige Wände, einseitige Türen und „Einbahnstraßen“ realisieren kann. Die Farben der Wände und Türen bleiben im Spiel, außer bei der Darstellung, unberücksichtigt.

Die Datei beginnt mit einem Identifikationsstring, der wie folgt aussieht:

`iMazeLab1\n`

wobei die „1“ für Formatspezifikation 1 steht und „`\n`“ ein Zeilenumbruch (Wert des Bytes =10) ist. Darauf folgen die Binärdaten:

Im ersten Byte steht die Breite des Spielfeldes (x-Richtung) in Feldern. Der Wert kann zwischen 1 und 127 liegen. Darauf folgt ein Byte mit der Länge des Spielfeldes (y-Richtung), ebenfalls in Feldern, von 1 bis 127.

Im folgenden Teil werden zeilenweise beginnend mit der nordwestlichen Ecke die Informationen für jedes Feld abgespeichert. Das sind 4 Byte pro Feld, je 1 Byte pro Himmelsrichtung gesehen von der Feldmitte. Die Himmelsrichtungen sind im mathematisch positiven Sinn (gegen den Uhrzeigersinn) gespeichert, also Nord (oben), West (links), Süd (unten) und Ost (rechts). Ein solches Byte mit der Information für eine Seite eines Feldes ist folgendermaßen codiert:

Bit	Bedeutung
0 ... 3	Farbe (0 ... 15)
4	ungenutzt
5	ungenutzt
6	schusssicher
7	unbegehbar

Die zugehörigenden Datenstrukturen für die interne Verarbeitung sind in `labyrinth.h` definiert.

Es ist zusätzlich gefordert, daß die Außenseiten des Spielfeldes von einer Wand im oben beschriebener Codierung umgeben sind. Die mit `ungenutzt` gekennzeichneten Bits in der obigen Codierung sind mit 0 vorzubesetzen.

An diese Daten schließt sich ein Kommentar in Form eines Strings an. Er erstreckt sich bis zum Ende der Datei und wird mit einem Zeilenumbruch abgeschlossen. Wie „Kommentar“ schon sagt, wird der Text nicht ausgewertet, sondern dient lediglich zur Information. Der Inhalt ist nicht näher spezifiziert, sinnvoll erscheint aber folgendes:

- Programm, mit dem die Datei erstellt wurde
- Größe des Spielfeldes und weitere Parameter
- Ersteller der Datei
- Hinweise und Besonderheiten

Kapitel 6

Compilieren und Installieren

6.1 Unterstützte Plattformen

Die hier beschriebene Version 1.0 von *iMaze* wurde entwickelt auf Sun Workstations mit dem Betriebssystem SunOS 4.1.3 und der OpenWindows-Oberfläche. Es wurde erfolgreich compiliert mit:

- cc unter SunOS 4.1.3
- gcc 2.4.5 unter SunOS 4.1.3
- cc (SparcWorks 2.0.1) unter Solaris 2.3
- gcc unter Linux 1.0 (Slackware 1.2.0)

Wahrscheinlich ist, daß sich *iMaze* auch auf anderen Systemen compilieren läßt. Erforderlich ist aber die XView-Library, genauer die Libraries `X11`, `xview` und `olgx`. Alle drei Libraries sind als Public Domain von verschiedenen FTP-Servern erhältlich. Aufgrund des Umfangs dieser Libraries ist jedoch zu bezweifeln, daß ein fehlerfreies Compilieren auf allen Systemen einfach machbar ist.

Wie bereits in der Einleitung und im Kapitel 3 beschrieben, ist eine Schnittstelle für die Portierung auf anderen Oberflächen und X-Libraries vorgesehen, so daß alle Anwender der oben nicht erwähnten Systeme auf eine baldige Verfügbarkeit für ihre Plattform hoffen können.

6.2 Verzeichnisstruktur

Nach dem Auspacken des Pakets `imaze1.0.tar.Z` finden Sie ein Verzeichnis `imaze` mit allen Daten. Die darin enthaltene Datei `README` sollte unbedingt zuerst gelesen werde, da dort aktuelle Veränderungen beschrieben sind. Das Unterverzeichnis `labs` enthält einige Dateien mit Beispiel-Labyrinthen und `source` alle Quelltexte zu *iMaze*.

6.3 Anpassen des Makefiles

Im Unterverzeichnis `source` finden Sie eine Datei `Makefile`, die wie unter Unix üblich das Compilieren steuert. Sie muß je nach System angepaßt werden muß. Die mitgelieferten Einstellungen sind für SunOS 4.1.3 und `cc` vorgegeben. Das `Makefile` ist in zwei Teile unterteilt: im ersten Teil können Anpassungen an die Systemumgebung vorgenommen werden, während der zweite Teil unverändert bleiben sollte. Der erste, interessante Teil ist hier abgedruckt.

```
...

# zuerst die systemabhaengigen Anpassungen (unpassendes auskommentieren):

# SunOS 4.1.3, OpenWindows:

GRAF-OBJ=$(XV-OBJ)
LIBS=

# SunOS 5.3, OpenWindows:

#GRAF-OBJ=$(XV-OBJ)
#LIBS=-lsocket -lnsl -lucb

# Linux Slackware 1.20, OpenWindows:

#GRAF-OBJ=$(XV-OBJ)
#LIBS=
#SYS-DEFS=-DSYSV

############################################################################

CC= gcc
COPT= -O
LD= $(CC)
LINT= lint
INSTALL= install -s -m 755

BINDIR= ..

LIBS=
XLIBS= -L$(OPENWINHOME)/lib -lxview -lolgx -lX11
INCLUDES= -I$(OPENWINHOME)/include

# moegliche Defines:
# aus baulab.c:    -DNORMALFARBE=5
# aus bewegung.c:  -DSCHRITT='(RASPT/8)' -DSCHUSSSCHRITT='(RASPT/4)'
#                  -DSPIELER_TOT_ZEIT=30 -DMIN_GEGNER_NAEHE=5
#                  -DMIN_SCHUSS_NAEHE=3
# aus global.h:    -DTRIGANZ=1024
# aus grafik.h:    -DSICHTW=16
# aus ip_netz.c:   -DDEFAULT_SERVER='"imaze.rz.tu-clausthal.de"'
```

```
#                   -DDEFAULT_PORT=5323
# aus ip_server.c: -DDEFAULT_PORT=5323
# aus netzwerk.c:  -DSPIEL_VERLASSEN_TIMEOUT=3
# aus session.c:   -DZEITTAKT=90
# aus speicher.c:  -DDEBUG_SPEICHER
# aus system.c:    -DSYSV
# aus X_grafik.h:  -DAUGENECKENANZ=32 -DMUNDECKENANZ=32 -DKREUZ_INRAD=4
#                   -DKREUZ_AUSRAD=10

DEFINES=
LDFLAGS= $(COPT)
CFLAGS= $(INCLUDES) $(COPT) $(DEFINES) $(SYS-DEFS)
LINTFLAGS=

# bis hier Optionen eintragen
############################################################################

...
```

Am Anfang sind die Definitionen für das gewünschte System zu aktivieren, während alle anderen mit einem Kommentarzeichen (`#`) desaktiviert werden müssen. Im folgenden Teil kann man den zu verwendenen C-Compiler (`CC`) und dessen Optionen für Optimierung oder Debug-Informationen (`COPT`) eintragen sowie den Aufruf für den Linker (`LD`) und das Programm zum Installieren (`INSTALL`). Die Voreinstellung sieht eine Installation mit den Dateizugriffsrechten `rwxr-xr-x` vor. Mit `BINDIR` wird das Verzeichnis angegeben, in das die fertigen Programme installiert werden sollen. Da die wenigsten Benutzer von *iMaze* die Rechte haben dürften, Programme in `/usr/local/bin` installieren zu können, ist hier das übergeordnete Verzeichnis voreingestellt.

Wichtig ist die Einstellung, wo der C-Compiler die Include-Dateien und die Libraries zu X11 und XView suchen soll. Bei `XLIBS` und `XINCLUDES` ist nach `-L` bzw. `-I` der passende Pfad einzutragen. Bei Systemen mit OpenWindows sollte die Environment-Variable `$OPENWINHOME` diesen Pfad enthalten.

Zum Schluß können noch eine Reihe von Werten definiert werden (`DEFINES`). Alle möglichen Punkte und deren Voreinstellung sind darüber aufgelistet. Es gibt keine Notwendigkeit für eine Veränderung der meisten Werte. Deshalb wird hier auf eine vollständige Auflistung verzichtet. Empfehlenswert ist nur das Setzen des Eintrags `DEFAULT_SERVER` auf einen lokalen Host. Sollte der Port `5323` schon für andere Programme belegt sein (s. `/etc/services`), so kann mit `DEFAULT_PORT` ein anderer gewählt werden.

6.4 Compilieren

Ist das `Makefile` korrekt angepaßt worden, reicht das Eingeben von `make` oder `make all`, um alle Programme zu erzeugen. Die Programme können auch einzeln generiert werden, dazu ist `make imaze`, `make imazesrv` usw. einzugeben. Mit `make install` und `make clean` erfolgt die Installation und das Löschen aller nicht notwendigen Dateien. Bei den oben aufgeführten Systemen sollte es keine Fehler beim Compilieren geben. Für andere Systeme ist ggf. eine

Anpassung am Quelltext unumgänglich. Die Option `-Wall` vom GNU-C-Compiler gibt eine Reihe von Warnmeldungen aus, die jedoch bedeutungslos sind.

6.5 Installieren

Sind die compilierten Programme und die Labyrinthe in einem geeigneten Verzeichnis verfügbar gemacht worden, empfiehlt sich abschließend auf Unix-Systemen eine Erweiterung der Datei `/etc/magic`. In dieser Datei sind charakteristische Informationen von Dateien abgelegt, anhand derer sie von dem Kommando `file` identifiziert werden. Für die *iMaze*-Labyrinthdateien ist folgendes einzutragen:

```
0       string          iMazeLab        iMaze Labyrinth
>8      string          1               Version 1
```

Kapitel 7

Bedienung

Für ein neues Spiel muß als erste Komponente der Server gestartet werden. Benutzer an der TU Clausthal können den ständig laufenden Server auf dem Host `imaze.rz.tu-clausthal.de` verwenden. Benutzer ausserhalb Clausthals sollten einen eigenen Server starten. Danach können bis zu 42 Spieler und Ninjas (Computergegner) an einer Spielrunde teilnehmen.

Die folgende Beschreibung bezieht sich auf die Benutzung der in Version 1.0 enthaltenen Programme. Bitte beachten Sie kurzfristig erfolgte Veränderungen, die in der Datei `README` beschrieben werden.

7.1 Bedienung des Servers

Der Server wird aus einer Shell folgendermaßen gestartet:

```
imazesrv [-f] [-p port] labyrinth-datei
```

Für `labyrinth-datei` ist der Name und ggf. Pfad einer Datei anzugeben, die ein Labyrinth gemäß der Spezifikation in Kapitel 5 enthält. In der Distribution sind einige Beispiel-Labyrinthe enthalten. Neue können mit dem Programm `genlab` (s. Kapitel 7.4) erzeugt werden. Mit `-p port` kann die Nummer oder der Name des TCP/IP-Ports angegeben werden, über den der Server erreichbar sein soll. Standardmäßig wird der Port `5323` verwendet. Der Parameter `-f` sollte normalerweise nicht angegeben werden. Die Option bewirkt einen Start des Servers, auch wenn der Port nicht freigegeben ist.

Es empfiehlt sich, den Server als Hintergrundprozeß zu starten. Man sollte aber beachten, daß er dann nach einem „logoff" des Benutzers weiterläuft und auch dann, wenn momentan keine Spieler angemeldet sind. Der Server muß später mit `kill -TERM pid` beendet werden, wobei `pid` die Prozeßnummer ist, die man mit dem Kommando `ps` erhält.

Pro Host kann nur ein Server gestartet werden – es sei denn, es werden beim Start unterschiedliche Portnummern angegeben. Wenn ein Server läuft, kann das Aussehen des Labyrinths oder andere Einstellungen nicht mehr verändert werden. Um ein neues Labyrinth zu laden, muß der alte Server beendet und ein neuer gestartet werden.

In späteren Versionen ist eine deutliche Verbesserung der Handhabung des Servers denkbar, etwa mit einem *Control-Tool*.

7.2 Bedienen des XView-Clients

Während alle anderen hier beschriebenen Programme von jedem Terminal aus gestartet werden können, erfordert der Client, mit dem Benutzer am Spiel teilnehmen, die X11-Oberfläche. Wie jedes X11-Programm kann man *iMaze* auch mit Ausgabeumlenkung auf ein anderes X-Terminal starten.

7.2.1 Starten

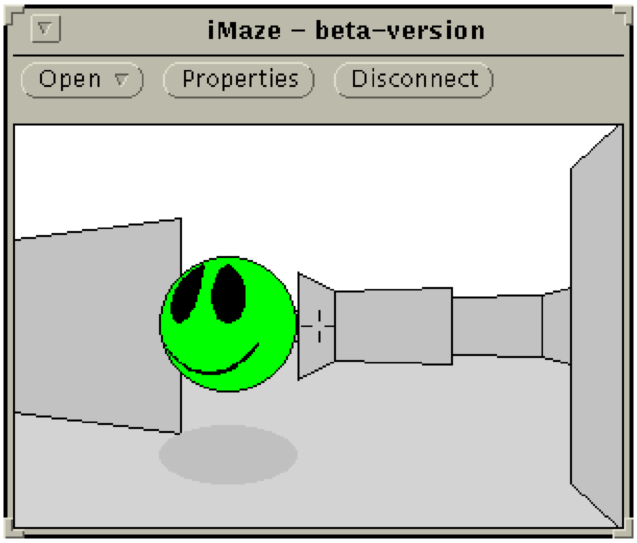
Bevor ein Client gestartet werden kann, muß zuerst ein Server laufen. Vergewissern Sie sich, daß dies der Fall ist! Der grafische Client wird aufgerufen mit

```
imaze [XView-optionen] [server-host]
```

`XView-optionen` sind die Parameter, die man an alle XView-Programme übergeben kann, wie z. B. `-Ws fenstergröße`. Alle möglichen Parameter erhält man auf den meisten Unix-Systemen mit `man 7 xview`.

Als weitere Option kann man den Rechner angegeben, auf dem der Server läuft. Wird kein Server angegeben, so wird der Default-Server genommen, der im `Makefile` eingetragen ist (s. Kapitel 6). Als `server-host` kann sowohl der Rechnername, wie `imaze.rz.tu-clausthal.de`, als auch die numerische Internet-Adresse, wie `139.174.2.11`, angegeben werden. An die Rechneradresse kann man mit `:5323` eine Portadresse des Servers anhängen. Wird dieser Port nicht explizit angegeben, so findet die Default-Portadresse Verwendung. Die Angabe von nur `:` als `server-host` veranlaßt das Programm, den Server auf dem eigenen Rechner (`localhost`) zu suchen.

Danach sollte sich das Hauptfenster zeigen. Nach einer kurzen Wartezeit erscheint darin die 3D-Ansicht vom Labyrinth. Das Anmelden beim Server ist jetzt vollständig abgeschlossen.



Das Hauptfenster und auch alle anderen Unterfenster können beliebig in der Größe verändert werden. Die Größe des Fensters ist entscheidend für die Geschwindigkeit des Spiels. Die optimale Fenstergröße hängt sehr von dem System ab, auf dem die Ausgabe erfolgt.

Kann eine Verbindung zum Server nicht hergestellt werden, ist nicht ausreichend Speicher vorhanden oder tritt ein anderer Fehler auf, so öffnet sich ein Fenster mit einer Fehlermeldung bzw. die Meldung wird auf `stderr` ausgegeben und das Programm beendet sich. Bei einfacheren Problemen, z. B. falls nicht genügend Farben verfügbar sind, kann der Benutzer auswählen, ob er das Programm fortsetzen oder lieber abbrechen möchte.

7.2.2 Menüs

Das Hauptfenster hat drei Menü-Punkte:

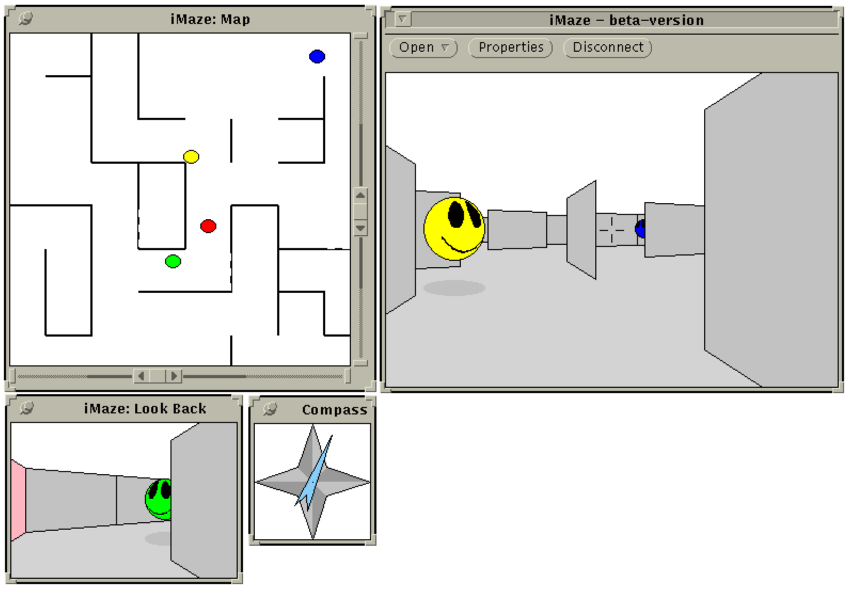
Open Hier können weitere Fenster mit Anzeigen geöffnet werden.

Map zeigt die Karte des Labyrinths mit den Spielern an. Auf der Karte ist der Grundriß des Labyrinths zu sehen. Dabei sind Wände als durchgezogene Linien, Türen als gestrichelte Linien, einseitige Türen als Kombination daraus und einseitig durchsichtige Wände gepunktet gezeichnet. Ferner werden alle im Spiel befindlichen Gegner dargestellt.

Look Back ist ein Rückspiegel. Er zeigt eine 3D-Sicht nach hinten an.

Compass zeigt die eigene Blickrichtung an.

Alle Unterfenster können jederzeit geschlossen und über das Menü wieder neu geöffnet werden. Jedes Fenster kann nur einmal gleichzeitig geöffnet sein.

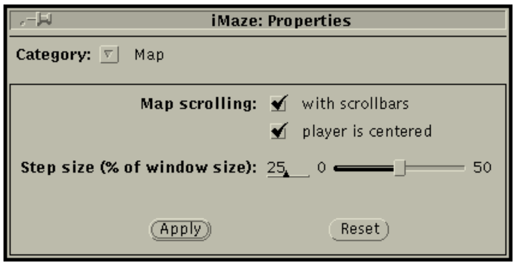


Properties öffnet ein kleines Fenster in dem Einstellungen vorgenommen werden können. Es gibt zwei Unterpunkte:

Game

Hier kann man festlegen, ob die Verbindung zum Server automatisch erhalten werden soll, wenn die Oberfläche blockiert ist, z. B. durch Verschieben von Fenstern oder durch andere Programme. (Bekanntlich wird ein Client, der sich eine bestimmte Zeit nicht mehr meldet, vom Server auf Pause gesetzt.) Für diesen Fall kann man eine Zeit vorgeben, nach der ggf. ein leeres Paket zum Server geschickt wird.

Map



Hier kann man wählen, ob das Kartenfenster mit Rollbalken ausgestattet werden soll. Im Normalfall ist die Karte immer so groß wie das Fenster. Ist die Option angewählt, kann die Karte nur noch vergrößert werden. Beim Verkleinern des Fensters erhält man einen Ausschnitt, den man mit den Rollbalken verschieben kann. Diese Option ist sehr sinnvoll bei größeren Labyrinthen. Ferner kann man angeben, ob der Kartenausschnitt immer so verschoben werden soll, daß man selbst in oder nahe der Mitte des Ausschnitts ist. Wie weit man sich von der Mitte entfernen kann, wird mit einem Schieberegler ausgewählt.

Disconnect Nach einer Sicherheitsabfrage wird hiermit die Verbindung zum Server beendet. In der aktuellen Version wird daraufhin auch das Programm beendet. Ein Wechseln des Servers ohne Beenden des Programms ist derzeit nicht möglich. Das Spiel wird ebenfalls durch Schließen des Hauptfensters oder mit `Control-C` oder `kill` beendet.

7.2.3 Spielsteuerung und -ablauf

Der vorliegende Client wird über Tastatur gesteuert. Damit die Eingaben das Programm erreichen, muß eines der *iMaze*-Fenster aktiv sein. Mit `Cursor-hoch` und `Cursor-runter` kann man vor und zurück laufen, mit `Cursor-links` und `Cursor-rechts` kann man sich drehen, und mit der `Space`-Taste kann man einen Schuß nach vorn abfeuern. (Einige Tastaturen erlauben leider nicht mehrere Aktionen gleichzeitig) Für spätere Version geplante kaufbare Spieloptionen, wie Sonderwaffen etc., sind dann über Ziffern-Tasten wählbar. Wer während einer kurzen Abwesenheit vom Rechner nicht zur Zielscheibe für alle anderen Mitspieler werden möchte, kann sich mit `Control-s` in einen Pause-Modus setzen lassen und mit `Control-q` wieder ins Spiel zurückkehren. Diesen Pause-Modus aktiviert der Server auch automatisch, wenn die Verbindung unterbrochen ist oder ein Client aus einem anderen Grund nicht mehr antwortet.

Zusammengefaßt die Tastaturbelegung in einer Tabelle:

Taste	Bedeutung
`Cursor hoch`	vorwärts laufen
`Cursor runter`	rückwärts laufen
`Cursor links`	drehen nach links
`Cursor rechts`	drehen nach rechts
`Leertaste`	Schuß abfeuern
`Control-S`	Pause beginnen
`Control-Q`	Pause beenden
`Control-C`	Progammende
`1 ... 0`	für spätere Erweiterungen

Jeder Spieler kann nur einen Schuß haben. Wird ein neuer abgeschossen, so wird der alte beendet. Ein Schuß verschwindet auch, wenn er eine Wand, eine Tür oder einen Gegner trifft. Ein getroffener Spieler verschwindet und entsteht nach einer kurzen Pause an einer anderen Stelle des Labyrinths wieder neu. Das Bilden von Teams ist nicht unterstützt; jeder spielt gegen jeden. Eine Bewertung nach Punkten ist ebenfalls nicht Bestandteil des Softwarepraktikums.

Wände sind im Labyrinth grau oder unsichtbar. Türen sind farbig (bei schwarz-weiß dunkel). Es gibt Türen, die in beide Richtungen oder nur in eine Richtung durchlaufen werden können. Die Spieler haben alle unterschiedliche Farben, bzw. zumindest unterschiedliche Augenfarben. Jeder Spieler hat dieselbe Farbe wie seine Schüsse.

Wurde man selbst abgeschossen, so bekommt man während der Pause das Gesicht des Schützen zu sehen.

7.3 Starten von Computergegnern

Die Computergegner werden ähnlich gestartet wie die normalen Clients. Es empfiehlt sich, die Ninjas als Hintergrundprozesse zu starten. Man sollte aber bedenken, daß sie dann weiterlaufen, bis sie mit `kill` wieder beendet werden oder der Server beendet wird. Der `ninja` benötigt keine X11-Oberfläche, er erzeugt im Normalfall keinerlei Ausgaben.

```
ninja [server-host [shot-percentage]]
```

`server-host` entspricht dem Parameter beim Client (s. 7.2.1). `shot-percentage` kann zusätzlich angegeben werden und bestimmt wie häufig (in Prozent) die Computergegner schießen sollen; default ist 100.

Der mitgelieferte Ninja ist nur ein Beispiel. Er hat keine aufwendige Strategie, sondern sucht sich den nächsten Mitspieler, verfolgt ihn und schießt dabei zufällig. Die Praxis hat jedoch gezeigt, daß schon zwei dieser Gegner einen ungeübten Spieler in Bedrängnis bringen können.

7.4 Erzeugen neuer Labyrinthe

In der hier beschriebenen Version ist noch kein interaktiver oder grafischer Editor enthalten, mit dem neue Labyrinthe erstellt werden können. Da die Labyrinth-Dateien im Binär-Format gespeichert werden, ist ein Editieren von Hand (z. B. mit einem Text-Editor) nicht möglich.

Die Labyrinthe, auch die mitgelieferten, werden mit dem Programm `genlab` über einen Zufallsgenerator erzeugt, wobei einige Parameter vorgegeben werden können.

```
genlab [-sv] [-w width] [-h height] [-d wall-density]
       [-D door-density] [-r random-seed] lab-file
```

`lab-file` ist der Dateiname des zu erzeugenden Labyrinths. Er muß immer angegeben werden, während alle anderen Angaben optional sind.

`-s` Keine Fallen (Räume ohne Ausweg) zugelassen

`-v` *verbose-mode* (Informationen ausgeben)

`-w width` Angabe der Labyrinthbreite (x-Richtung) in Feldern.
(`width` = 1 ... 127, default = 16)

`-h height` Angabe der Labyrinthlänge (y-Richtung) in Feldern.
(`height` = 1 ... 127, default = 16)

`-d wall-density` Anteil der Wände in Prozent
(`wall-density` = 0 ... 100, default = 38).
Zu große Werte können in Verbindung mit der Option `-s` lange Laufzeiten verursachen.

`-D door-density` Wieviel Prozent der Wände sollen Türen sein?
(`door-density` = 0 ... 100, default = 5)

`-r random-seed` Startwert des Zufallszahlengenerators.
(`random-seed` = 0 ... $2^{31} - 1$, default durch Systemzeit)

`genlab` trägt in den Kommentar der Labyrinth-Datei die verwendeten Parameter ein. Man kann sich die Werte zu einem Labyrinth mit `strings lab-datei` anzeigen lassen. Der Zufallsgenerator ist systemunabhängig. `genlab` liefert auf allen Rechnern das gleiche Ergebnis. Der Aufbau einer Labyrinth-Datei ist im Kapitel 5 ausführlich beschrieben.

Labyrinthe von `genlab` sind unterschiedlich gut. Unter Umständen benötigt man einige Versuche, bis man ein brauchbares Labyrinth erhält.

Literaturverzeichnis

[1] A. Nye: The Definitive Guides to the X Window System, Volume One, Xlib Programming Manual,
O'Reilly & Associates, Inc.

[2] A. Nye: The Definitive Guides to the X Window System, Volume Two, Xlib Reference Manual,
O'Reilly & Associates, Inc.

[3] D. Heller: The Definitive Guides to the X Window System, Volume Seven, XView Programming Manual,
O'Reilly & Associates, Inc.

[4] D. Heller, D. Dougherty, A. Nye: The Definitive Guides to the X Window System, Draft, Overview of XView Programming,
O'Reilly & Associates, Inc.

[5] O. Jones: Introduction to the X Window System,
Prentice Hall, Englewood Cliffs, N.J. 07632

[6] XView 1.0 Reference Manual: Summary of the XView API,
Sun microsystems

[7] Open Look Graphical User Interface Functional Specification Release 1.0.1,
Sun microsystems

[8] AnswerBook, Sun online Dokumentation,
Sun microsystems

[9] SunOS 4.1.3 man-pages, Online Dokumentation,
Sun microsystems

[10] M. Fritze: Anleitung MidiMaze 2,
Σ–Soft, 1990