Architektur zu Hexchess

Inhaltsverzeichnis

Grober Überblick Software-Architektur ---------- 1

Aufbau Client ---------- 2

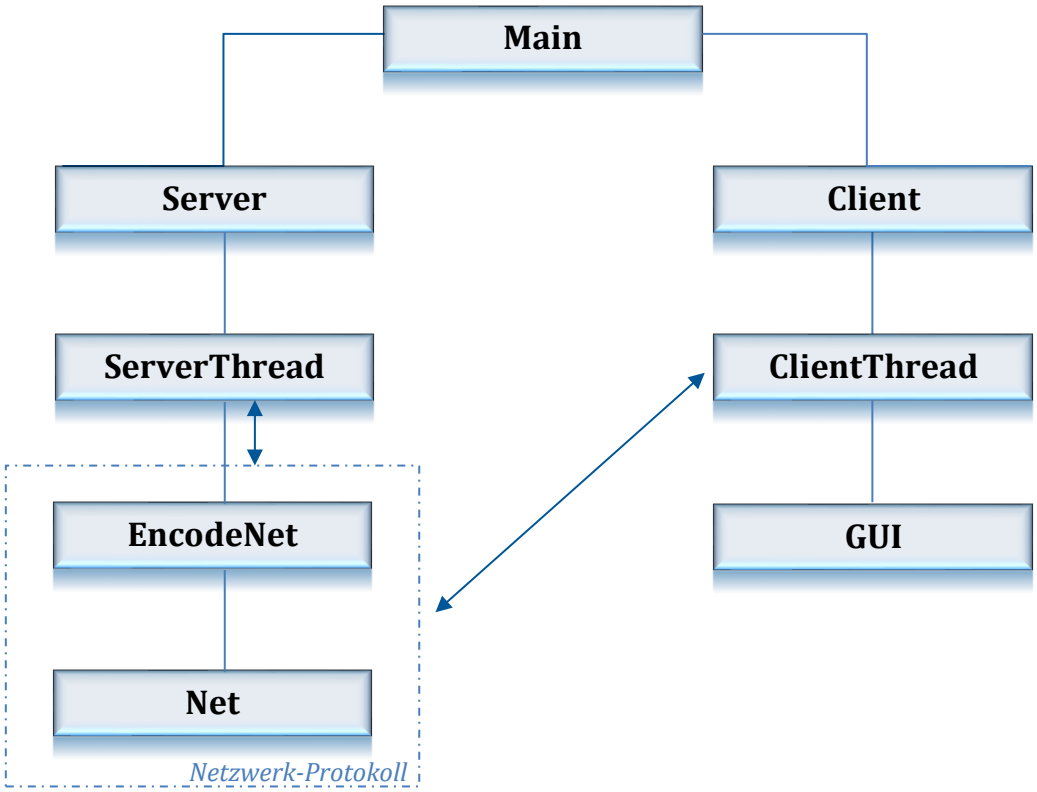
Aufbau Server ---------- 6

Netzwerk ---------- 8

Sonstiges ---------- 9

Details zu GFX ---------- 10

Grober Überblick Software-Architektur[1]



Dieses Diagramm ist eine sehr allgemeine und grobe Darstellung unseres Programmes.

Kurz und knapp erklärt, werden pro Spieler/in je ein ServerThread und ein ClientThread erzeugt.

Die Klassen EncodeNet und Net können als Netzwerk-Protokoll zusammengefasst werden und werden nur serverseitig direkt aufgerufen.

Wie bereits aus dem Diagramm ersichtlich, laufen alle GUI-Klassen über den Client.

Um das Spiel starten zu können, wird Main durch die zwei möglichen Befehle gestartet:

java -jar Hexchess.jar server port

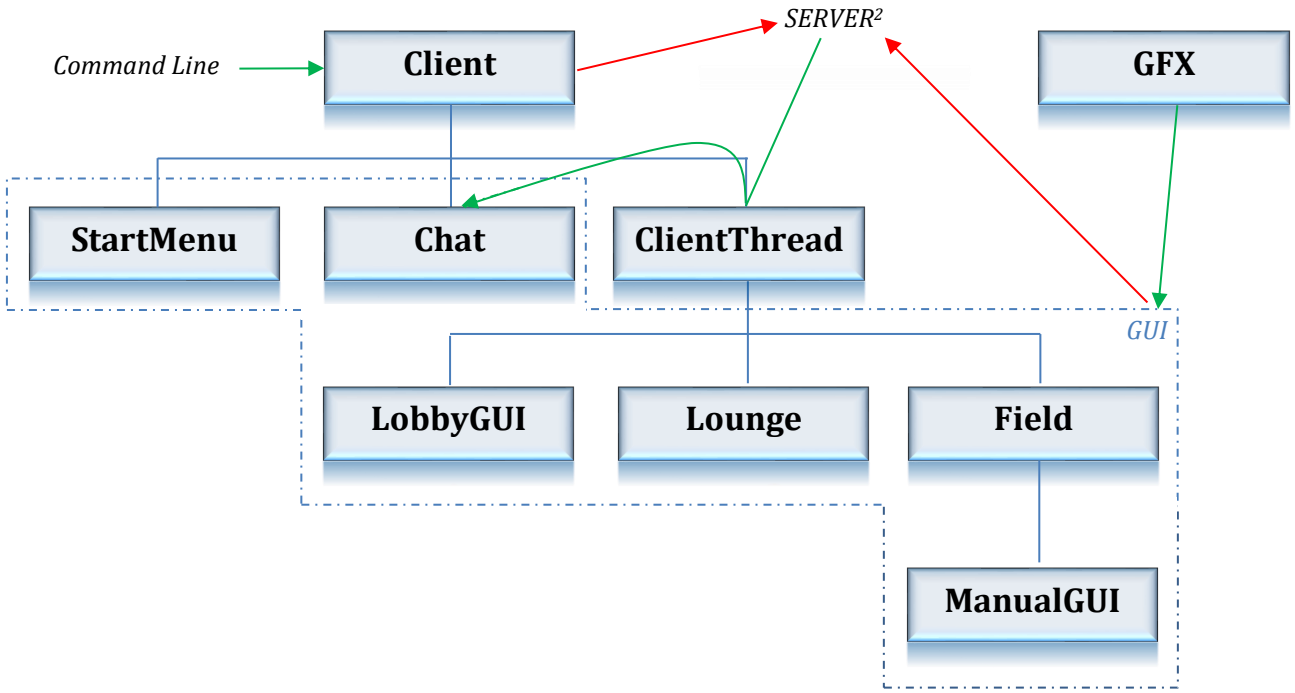
java -jar Hexchess.jar client ip:port [username optional]

Diese Befehle rufen dementsprechend Server oder Client auf und die Main-Klasse übermittelt die erhaltenen Daten.

Im Folgenden wird detaillierter auf die bereits erwähnten Klassen und der Architektur unseres Programmes eingegangen.

[1] Im Diagramm wird auf ".java" verzichtet. Mit Ausnahme von „GUI", die eine Ansammlung von verschiedenen GUI-Klassen bezeichnet, sind alle Begriffe Klassennamen.

Aufbau Client



Allgemein werden alle neu erstellten Objekte, die über Threads gehandhabt werden, mit der Methode «javax.swing.SwingUtilities.invokeLater» aufgerufen, um unerwünschte Probleme mit Threads zu vermeiden. Auch erfolgt das Versenden von Nachrichten an den Server direkt über die entsprechenden Klassen, wenn nichts anders steht.

Client: Beim Aufruf von Client wird ein InputStream- und OutputStream-Objekt erstellt. Diese sind dazu da, Nachrichten vom Server zu empfangen und Nachrichten an den Server zu senden. Diese beiden Stream-Objekte werden an alle nachfolgenden Klassen in diesem Diagramm übergeben.

Auch werden neue Objekte der Klassen StartMenu und Chat erzeugt. Diese werden dem ClientThread übergeben, welche mit einem Thread gehandhabt wird.

Nebst der Erstellung von Objekten ist der Client dafür zuständig, Nachrichten von der Command Line zu empfangen und diese an den Server weiterzuleiten.

Wenn kein Name übergeben wurde (siehe Aufruf Main.java II), ermittelt die Klasse Client den Desktopnamen der jeweiligen Spieler/innen.

ClientThread:

Im ClientThread werden Nachrichten vom Server gemäss Netzwerk-Protokoll empfangen. Je nach Nachricht wird diese dem Chat weitergeleitet resp. die Nachricht wird in der GUI des Chats sichtbar sein. (Diese Klasse selbst versendet keine Nachrichten an den Server.)

Spezifische Nachrichten des Servers sorgen dafür, dass ein neues Objekt für LobbyGUI, Lounge oder Field erstellt wird oder gewisse Eigenschaften

[2] Hier ist nicht die Klasse Server gemeint, sondern allg. Server.

aktualisiert werden. Auch kann wieder zurück in die Klasse StartMenu gewechselt werden.

Chat: In der Klasse Chat werden eingetippte Nachrichten an den Server weitergeleitet. Der Empfang von Chat-Nachrichten findet, wie bereits erwähnt, im ClientThread statt.

Das Chat-Fenster ist unabhängig von der restlichen GUI und kann freiwählbar manövriert werden.

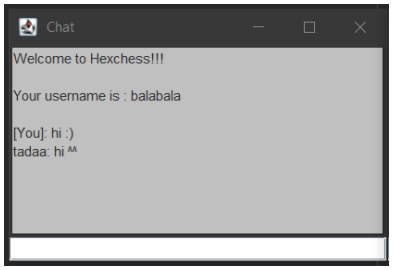


Abb. 1: Chat GUI

StartMenu:

Im StartMenu kann der Spielname eingegeben werden. «Enter» kann über die Tastatur oder den Button erfolgen.

Abb. 2: StartMenu

Im Textfeld ist bereits ein Name initialisiert. Dieser wurde vom Client übermittelt. Je nach dem handelt es sich um den Desktopnamen oder den zu Beginn eingegebenen Namen (siehe Aufruf Main.java II).

Der eingegebene Name wird dem Server zur Validierung übermittelt. Ist der Name einzigartig, so wird dieser im Server und StartMenu gespeichert. Die Person betritt die Lobby.

LobbyGUI:

Beim Aufruf der LobbyGUI verschwindet die StartMenu-Fenster aus dem sichtbaren Bereich («setVisible(false)») Dadurch, dass das StartMenu-Objekt noch existiert, kann das StartMenu-Fenster jederzeit wieder sichtbar gemacht werden.

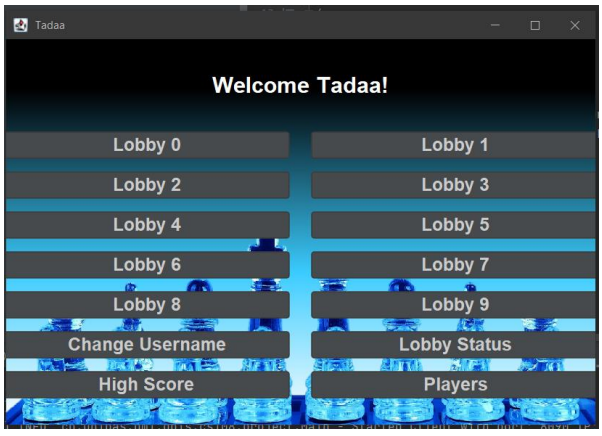


Abb. 3: Lobby GUI

Das LobbyGUI-Fenster enthält verschiedenste JButtons. Mit Ausnahme von einem Button wird nach jedem Klick eine Nachricht an den Server gesendet. Je nach Nachricht wird auch gehandelt. So können im Server gespeicherte Listen abgerufen werden oder die Person kann die Lounge betreten.

Lounge:

Abb. 4: Lounge

In der Lounge gibt es nur die Möglichkeit wieder zurück zur LobbyGUI zukehren. Auch hier wird dann eine Nachricht an den Server gesendet und vom ClientThread empfangen.

In der Lounge gibt es zudem eine Zählvariabel, die vermerkt, wie viele Spieler/innen bereits in der Lounge sind. (Die Anzahl Spieler/innen in einer Lobby werden auch im Server individuell gehandhabt. Genaueres siehe «Aufbau Server»)

Field: Sobald die dritte Person in derselben Lobby ist, wird das Field-Objekt durch den ClientThread initialisiert und gleichzeitig alle Lounges der drei Spieler/inne unsichtbar gesetzt.

Diese Klasse stellt das Spiel mit der GUI dar. Alle Züge werden dem Server gesendet, überprüft und je nach Feedback des Servers wird ein Zug gemacht. Mit dieser GUI wird gespielt.

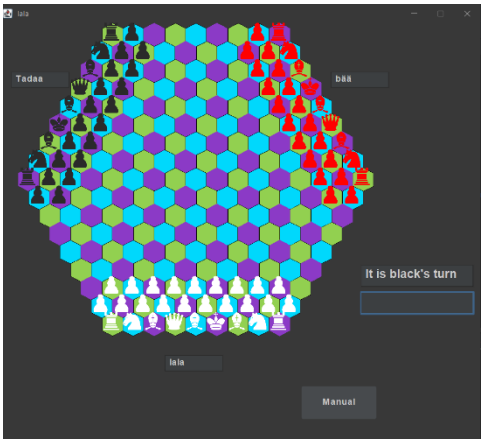


Abb. 5: Field

ManualGUI:

Das ManualGUI-Objekt kann durch anklicken des Buttons im Field erstellt werden. Dieses GUI ist eine visuelle Anleitung, wie das Spiel gespielt wird.

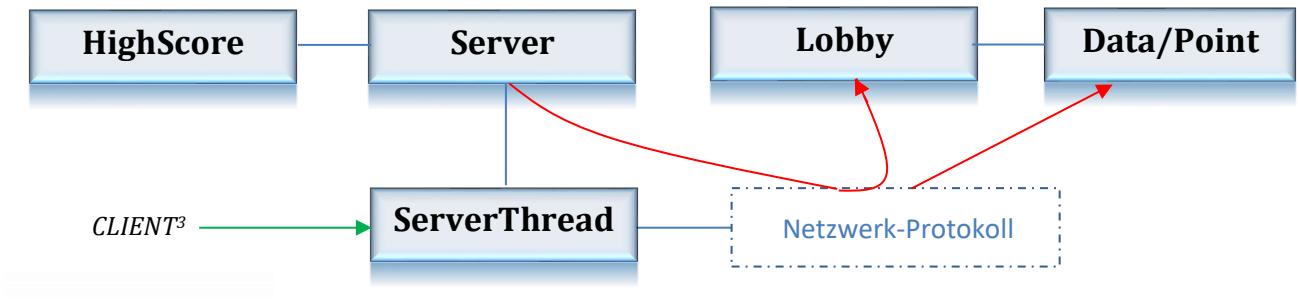
GFX:

Diese Klasse stellt die Engine für GUI dar. (Mangels Zeit konnte sie nicht mehr effektiv genutzt werden.)

Die Klasse GFX kann von allen Klassen auf der „Client-Seite“ zugegriffen werden.

Für Interessent/innen ist im letzten Kapitel Details zu GFX detaillierter beschrieben, wie genau diese Engine funktioniert.

Aufbau Server



Server: Im Server werden ein HighScore- und ServerSocket-Objekt erstellt. Wann immer ein/e Spieler/in sich einlogged, wird eine Socket erstellt und mit ServerSocket.accept() initialisiert. Diese wird dem neu erstellten ServerThread-Objekt übergeben, welches wiederum durch einen Thread gehandhabt wird.

Die Clients werden im Server in einem Vector gespeichert. Auch werden die ersten zehn Lobbies über das Netzwerk-Protokoll initialisiert und dort gespeichert.

ServerThread:

Die Klasse ServerThread empfängt die Nachrichten, die vom Client gesendet wurden und überprüft, ob diese Nachricht eine „Ping-Pong-Nachricht" ist. Falls dies nicht der Fall ist, wird die gesamte Nachricht an das Netzwerk-Protokoll weitergeleitet (Details siehe dazu später).

HighScore:

Die Klasse HighScore, die vom Server initialisiert wird, liest zuerst aus der Datei „HighScore.txt" und speichert dessen Inhalt in eine LinkedList.

Über das Netzwerk-Protokoll wird dieselbe Klasse HighScore wieder aufgerufen, um einen neuen Sieg zu speichern und diesen in die HighScore-Datei zu speichern.

Lobby: Die Lobby wird in der Klasse Server initialisiert und über das Netzwerk-Protokoll gesteuert.

In der Lobby gibt es ein Array der Länge drei, in der die Spieler/innen, die in dieser Lobby sind, gespeichert werden. Nach dem Eintritt der dritten Spieler/in wird ein Data-Objekt erzeugt und das Spiel kann beginnen.

Data/Point:

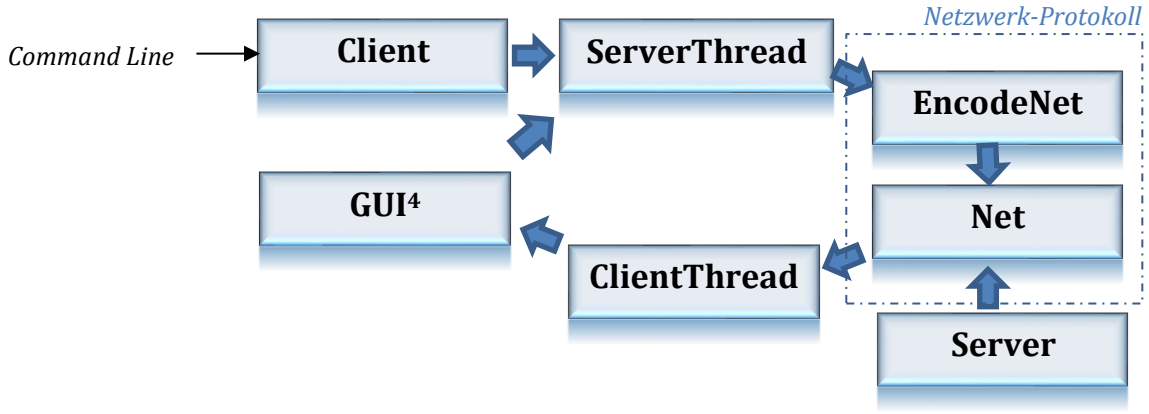
Die Klasse Data ist die sogennante Spielzug-Kontrollinstanz. Alle Nachrichten des Clients betreffend den Spielzügen werden der Klasse Data übermittelt.

[3] Wie bei Fussnote 2 gilt auch hier, dass mit «CLIENT» das gesamte Konstrukt gemeint ist und nicht nur eine einzelne Klasse.

Diese überprüft die Schachregeln und sendet eine Feedback-Nachricht zurück an den Client. Je nach Nachricht war der Zug erlaubt oder eben nicht.

Um Spielzüge überprüfen und ausführen zu können, werden die Koordinaten der Schachfiguren benötig. Diese werden über die Klasse Point gehandhabt.

Netzwerk



Wie schon im ersten Kapitel ersichtlich, verläuft die direkte Kommunikation zwischen Client und Server über das Netzwerk-Protokoll ausgehend über den beiden Klassen ServerThread und ClientThread.

In den vorherigen Kapiteln wurde bereits erklärt, welche Klassen Nachrichten empfangen und senden. Das oben abgebildete Diagramm dient der Zusammenfassung des Sendens und Empfangens von Nachrichten.

In diesem Kapitel wird lediglich auf die Klassen des Netzwerk-Protokolls eingegangen.

EncodeNet:

Diese Klasse wird vom ServerThread aufgerufen und filtert die Art der Nachricht. Sprich: Sie decodiert die Nachricht und führt die entsprechende Aktion durch. Grundsätzlich bedeutet dies, dass eine Methode in der Klasse Net aufgerufen wird.

Net: Die Klasse Net ist eine Ansammlung der Funktionen, die vom Server ausgeführt werden soll und/oder die Nachrichten an den Client sendet. (z.B. Funktionen für Chat-Nachrichten, Namensänderung, Schachzüge, usw.)

In dieser Klasse sind zwei LinkedLists vorhanden, die den Usernamen der Spieler/innen und die Lobby-Objekte speichern.

Wichtig anzumerken ist, dass nur serverseitige Klassen auf die Net-Methoden direkt zugreifen dürfen.

[4] Alle Begriffe sind Klassennamen, mit Ausnahme von GUI; Dies ist eine Ansammlung von verschiedenen Klassen, die das Spiel graphisch darstellen (siehe Aufbau Client)

Sonstiges

- Nebst der Klasse Net gibt es noch zwei weitere Klassen, die eine Ansammlung von verschiedensten Methoden beinhalten:

Tools: Alle Funktionen, die sehr oft benötigt werden, sind in dieser Klasse vorhanden. Sowohl serverseitige wie clientseitige Klassen können und dürfen auf diese Klasse zugreifen.

ToolsGUI: Wie der Klassename schon sagt, handelt es sich hier um eine Klasse, dessen Inhalt spezifische Funktionen für die GUI sind. Somit wird sie hauptsächlich clientseitig aufgerufen, da alle GUI über den Client verlaufen.

- Zusätzlich sind zwei weitere Klassen vorhanden, die die Main-Klasse aufrufen. Diese wurden v.a. zu Beginn gebraucht, als noch keine .jar-File existierte:
CallClient und **CallServer.**

- Packages:

Alle Klassen sind in einem Package: «ch.unibas.dmi.dbis.cs108.project»

Details zu GFX

Durch Aufruf der Funktion "pipeline" können Frames generiert werden, die dann direkt als Bild auf den Bildschirm projiziert werden können. Diese sind ein 3-dimensionales Array, wobei der erste Index y, der zweite x, und der dritte den Farbkanal (RGB) bezeichnet.

Die Funktion ist so konzipiert, dass nur das, was benötigt wird, speziell definiert werden muss:

- Möchte man eine einzige Farbe darstellen, so definiert man "bgcol" mit der gewünschten Farbe.
- Den Hintergrund (z.B. das Spielfeld) kann man direkt als 3D-Array übergeben (jeder Eintrag entspricht einem Farbkanal (RGBA) eines Pixels). Hierfür muss man jedoch die Parameter für die Affinabbildung angeben. Diese ermöglichen das Rotieren, Strecken in x-Richtung und/oder y-Richtung, Verschieben, Spiegeln und Scheren. Ergibt die Affinabbildung keinen genauen Pixel, sondern eine Position zwischen Pixeln, so wird der Farbwert bilinear interpoliert.
- Die Figuren werden per 2D-Array übergeben (jeder Eintrag bezeichnet einen Eintrag in der Farbpalette). Zusätzlich muss die Position und die benötigte Farbpalette definiert werde. Dies ermöglicht es, für alle Spieler/innen die gleichen Figuren zu verwenden, jedoch mit anderen Farbpaletten. Die Farben sind im RGBA-Format, was es uns ermöglicht, "gläserne" Figuren zu zeichnen. Es ist zusätzlich möglich, den "priorityBit" zu setzen, was dazu führt, dass eine Figur über allen gezeichnet wird, was z.B. für Figuren wie der Springer hilfreich sind, wenn diese gezogen werden.

Möchte man speziellere Effekte generieren, so kann man eine "displayList" definieren. Dies ermöglicht es einem beispielsweise, die Farbpalette nach jeder Zeile zu modifizieren. Oder man kann die Parameter zur Affinabbildung verändern, z.B. um perspektivische Abbildungen mit 1 Fluchtpunkt zu zeichnen. Der erste Eintrag beinhaltet stets, wie lange gewartet werden sollte, bis der nächste Befehl ausgeführt wird. Ist "normalised" wahr, so bezeichnet dies einen Bruchteil des Bildes (0.2= 20% des gesamten Bildes), sonst bezeichnet dies die Anzahl Bildzeilen.

Ist "fullDL" wahr, so besteht jeder Befehl aus: 1.der Anzahl zu wartenden Zeilen, 2. den 6 Parametern der Affinabbildung.

Ist "fullDL" falsch, so besteht jeder Befehl aus: 1. der Anzahl zu wartenden Zeilen, 2. einer Liste, was verändert werden soll, jede Zeile hat an erster Stelle einen Eintrag, ob die Liste fertig ist, gefolgt von "Adresse" und neuem Wert.Die "Adresse" bezeichnet, welcher Parameter zu verändern ist (Affine Parameter, Ursprung der Abbildung oder Farbpalette).

Im Anschluss sind noch zwei Bilder zur Veranschaulichung der Möglichkeiten von "displayList" zu sehen. (Hinweis: Mangels Zeit sind diese nicht mit GFX gerendert worden, es ist jedoch sicher möglich, dies zu tun. Daher ist es denkbar, dass die dazugehörenden displayLists dem gezeigten nicht vollständig entsprechen, da einige Werte mittels Lineals gemessen oder geschätzt wurden.)

Abb. 6: Dies ist ein Bild aus dem Startbildschirm von Project X" (Amiga, 1992).

BEISPIEL 1

Um dieses Bild zu generieren, wird das Bild ab einem bestimmten Punkt stets um eine Zeile nach unten verschoben, sodass diese Zeile des Hintergrundes erneut gezeichnet wird. (In diesem Beispiel ist "fullDL" wahr und "winY" entspricht hier der Bildschirmhöhe.)

Die entsprechende Initialisierung der Liste displayList sieht wie folgt aus:

```
GFX.displayList = {0.15, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, -1/winY, 0, 0, 0, 0, 0, 0, -2/winY...};
```

Daraus kann gelesen werden, dass 15 % des Bildes gewartet wird und die Verschiebung der y-Achse für 0 % des Bildes gewartet wird.

Abb. 7: Dieses Bild wurde aus "Desert Dream" (Amiga, 1993) entnommen.

BEISPIEL 2

Der Hintergrund wird hierbei in jeder Zeile gestreckt (Annahme: 181 zu streckenden Zeilen, sin(x) in Grad.) Hier ist "fullDL" falsch.

Die entsprechende Initialisierung der Liste displayList sieht wie folgt aus. (Die Beschreibung ist in der Java-Kommentar-Notation geschrieben; sprich nach «//»):

```
GFX.displayList = {0.24, 1, 6, 0.5, 1, 7, 0, -1,        //Abb.-Ursprung: (0.5,0), für 24% des Bildes warten
                   0, 1, 0, 1+sin(0), -1,               //Streckung der x-Achse um 1 + sin(0)
                   0, 1, 0, 1+sin(1), -1,
                   ...
                   0, 1, 0, 1+sin(180), -1,
                  -1};                                  //displayList fertig für dieses Frame
```