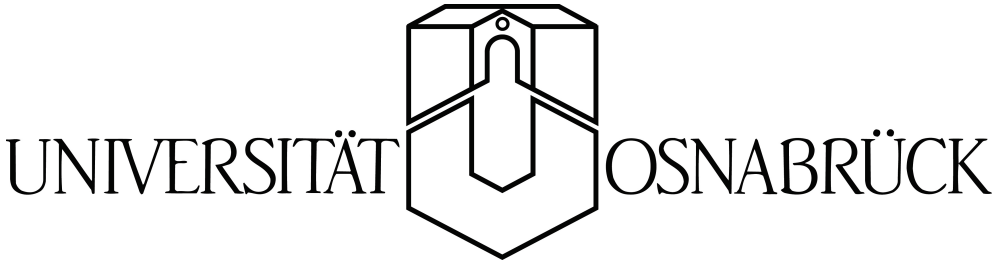

Flashanimationen von RoboCup Soccer 2D-Simulationen

Diplomarbeit
von
Tobias Schwegmann

betreut von
Prof. Dr. Oliver Vornberger
Prof. Dr. Martin Riedmiller

Fachbereich Mathematik/Informatik
Universität Osnabrück

16.03.2007

Vorwort

Diese Diplomarbeit entstand an der Universität Osnabrück im Institut für Informatik. Sie ist der schriftliche Teil der Diplomprüfungen für meinen Abschluss als Diplom-Mathematiker.

Danksagung

Ich möchte mich bei allen bedanken, die zum Gelingen dieser Diplomarbeit beigetragen haben. Besonderer Dank gilt den folgenden Personen:

- Herrn Prof. Dr. Oliver Vornberger für die gute Betreuung der Arbeit
- Herrn Prof. Dr. Martin Riedmiller für die gute Betreuung der Arbeit und die konstruktiven Vorschläge zur Applikation
- Patrick Fox für seine gute Betreuung und die konstruktiven Vorschläge zur Gestaltung der Arbeit
- Thomas Gabel für seine gute Betreuung, die konstruktiven Vorschläge und die Hilfe beim Erstellen der Applikation
- Dorothee Langfeld und Dr. Ralf Kunze für das Korrekturlesen dieser Arbeit
- Friedhelm Hofmeyer für die Bereitstellung eines Computers und der benötigten Software im Institut für Informatik

Ein ganz besonderer Dank gilt meinen Eltern, die mir das Mathematikstudium ermöglicht haben und mich während dieser Zeit unterstützt haben.

Warenzeichen

Inhaltsverzeichnis

Abbildungsverzeichnis

Teil I
Einleitung

„By the year 2050, develop a team of fully autonomous humanoid robots that can win against the human world soccer champion team.“

http://www.robocup.org

1 Einführung

Die vorliegende Diplomarbeit entstand auf Wunsch der AG Neuroinformatik des Instituts für Neuroinformatik an der Universität Osnabrück. Das Ziel der Arbeit ist eine gute und geeignete Visualisierung der RoboCup Soccer 2D Simulation zu erstellen. Diese Visualisierung soll es ermöglichen, einem breiten Publikum die Simulationen vorzustellen. Sie sollen zum einen als Download oder zum direkten Anschauen im Internet zur Verfügung gestellt werden. Außerdem soll es leicht möglich sein, ein Video für Vorführungen zu erstellen.

Der RoboCup Soccer Server [Web RCSS], bei dem sich die beiden spielenden Mannschaften anmelden, erstellt eine Logdatei, die alle benötigten Daten für eine Visualisierung enthält, wie zum Beispiel die Position der Spieler. Da die Daten nur in einem binären Format vorliegen ist es ohne Visualisierung schwierig den Verlauf des Spieles nachzuvollziehen. Daher wäre es eine große Hilfe, die Spiele zu visualisieren, auch um eine bessere Analyse der Spiele durchführen zu können.

Bisher gibt es lediglich die Möglichkeit, sich die Spiele mit einem zusätzlichen Programm, dem RoboCup Soccer Simulator Log Player[1] am eigenen PC anzuschauen. Um zu vermeiden, dass man ein zusätzliches Programm zur Betrachtung

benötigt, soll nun eine Form der Visualisierung gefunden werden, die drei Anforderungen berücksichtigt.

- Direkte Darstellung im Internet
- Download aus dem Internet
- Video zu Vorführzwecken in Vorträgen oder auf DVD

Das Flashformat scheint diese Anforderungen zu erfüllen, Flashanimation sind sehr gut für das Internet geeignet und es gibt Programme mit denen man Flashanimationen in ein Video umwandeln kann.

In der Arbeit wird nur die 2D-Simulation behandelt, es gibt zwar auch eine 3D-Simulation, diese ist aber noch nicht so weit entwickelt, dass es einen festen Standard für die resultierenden Logdateien gibt und somit ist es schwieriger eine gute und geeignete Visualisierung zu erreichen.

1.1 Aufgabenstellung

Wie oben bereits erwähnt ist das von Adobe (ehemals Macromedia) verbreitete Flashformat für die Animationen sehr gut geeignet. Um nicht jede Animation von Hand zu erstellen, soll eine Applikation entwickelt werden, die aus den vom RoboCup Soccer Server resultierenden Logdateien eine geeignete Visualisierung erstellt.

Mit Hilfe eines noch von Macromedia erstellten Software Development Kit für SWF[1]-Dateien soll das Tool erstellt werden. Es soll ein möglichst kleines und auch leicht zu bedienendes Tool erstellt werden, welches sich in eine Prozeßkette einfügen läßt, für eine einfache Verarbeitung mehrerer Logdateien oder eine Einbindung in den Arbeitsablauf des RoboCup Soccer Server.

1.2 Aufbau der Arbeit

Im folgenden Kapitel 2 soll die Robot Cup World Initiative (RoboCup) vorgestellt werden, insbesondere soll auf die Simulation von Agentenbasierenden Fußballspielen, die RoboCup Simulationsliga eingegangen werden. Ausserdem wird der RoboCup Soccer Server kurz beschrieben, und der Aufbau der Logdatei wird genauer betrachtet.

Der zweite Teil der Arbeit behandelt die verwendeten Techniken, in Kapitel 3 wird kurz beschrieben, warum für diese Arbeit C++ als Programmiersprache verwendet wurde. Im darauffolgenden Kapitel 4 wird dann das Vektorgrafikformat Flash

[1] Shockwave Flash

vorgestellt. In Abschnitt 4.1 wird ein kurzer Abriß zur Entwicklung des Flashformates gegeben. In Abschnitt 4.2 werden die Gründe für eine Verwendung von Flash diskutiert. Außerdem wird in Abschnitt 4.3 das SWF-Dateiformat genauer beschrieben, im folgenden Abschnitt 4.4 wird dann das Software Development Kit (SDK) zur Erstellung der Flashanimationen vorgestellt.

Der dritte Teil der Arbeit beschreibt dann das Tool zur Erstellung der Flashanimationen aus den Logdateien der RoboCup Simulationen. In Kapitel 5 wird ein kurzer Überblick über den Aufbau der Applikation gegeben, danach werden die einzelnen Klassen des Tools vorgestellt. In Kapitel 6 wird das aufrufende Programm RoboCup2Flash vorgestellt, Kapitel 7 stellt den LogFileReader vor, der die Methoden zum Auslesen der Logdatei bereitstellt. In Kapitel 8 wird eine Klasse zum Auslesen einer Konfigurationsdatei präsentiert. Das Kapitel 9 beinhaltet die einzelnen Objekte der Animation und in Kapitel 10 wird der FlashMaker vorgestellt, der die Animation zusammenstellt.

Im letzten Teil der Arbeit gibt es in Kapitel 11 ein Fazit zur erstellten Applikation, in dem die Ergebnisse der Arbeit zusammengefaßt werden. In Kapitel 12 wird dann noch ein kleiner Ausblick auf mögliche Erweiterungen und Weiterentwicklungen gegeben.

2 RoboCup Soccer 2D Simulation

Die RoboCup Soccer 2D Simulation ist ein Teil der gesamten Robot World Cup Initiative[2], kurz RoboCup, in dem es noch andere Ligen, wie die RoboCup Rescue und die bekannte RoboCup 4-legged League mit den Aibo-Robotern von Sony gibt. In diesem Kapitel soll die Simulationsliga vorgestellt werden und der zugehörige RoboCup Soccer Server (RCSS), der die Spiele hostet (veranstaltet).

Außerdem soll noch näher auf den Aufbau der Logdatei eingegangen werden, die vom Server erstellt wird. Diese Datei beinhaltet die Informationen die man zur Erstellung der Animationen braucht. Die Informationen zu diesen Themen lassen sich in der Anleitung zum RoboCup Soccer Server [RC2001] nachlesen.

2.1 Robot World Cup Initiative

Alan Mackworth hatte schon 1993 in [RC1993] die Idee fußballspielende Roboter für die Forschung auf dem Gebiet der künstlichen Intelligenz einzusetzen, er hatte aber zuerst damit keinen großen Anklang in der Forschergemeinde gefunden. Erst als Kitano, Asada und Kuniyoshi, drei japanische Wissenschaftler, seine Ideen in [RC1995] übernahmen und im Zuge eines Forschungsprojektes die Robot J-League[3] gegründet wurde, fand Mackworths Idee auch bei anderen Forschern Interesse. Nach einiger Zeit wurde die Robot J-League dann in die Robot World Cup Initiative umbenannt, oder abgekürzt RoboCup. 1997 gab es dann die ersten Robot World Cup Soccer Games. Das Ziel des RoboCup ist es neue Standardprobleme der KI vorzustellen und zu lösen.

Die RoboCup Federation wurde gegründet um die Arbeit der Forscher besser zu koordinieren. Der Zweck der RoboCup Federation ist es, den RoboCup bekannter zu machen und alljährlich die Weltmeisterschaften zu organisieren. Das eigentliche Ziel des RoboCup befindet sich im Zitat zu Anfang dieser Arbeit. Im Jahre 2050 sollen Roboter gegen den amtierenden Fußballweltmeister spielen und nach Möglichkeit gewinnen. Wir werden erst 2050 sehen, ob das Zitat nur zur Motivation der einzelnen Forscher dienen soll oder ob man es wirklich schaffen kann, den Menschen zu besiegen; beim Schach hat es ja auch geklappt[4].

[2] Siehe http://www.robocup.org
[3] Die J-League ist die japanische Profifußballliga
[4] Siehe http://de.wikipedia.org/wiki/Fritz_(Schach)#World_Chess_Challenge_2006

2.2 RoboCup Simulationsliga

Seit Beginn des RoboCup gibt es auch eine Simulationsliga, die Fußballspiele am Computer simuliert. Die Liga gibt es, um die programmierten Multi-Agenten-Systeme und die KI der Spieler zu testen, um sie dann später auf die Roboter zu übertragen. Außerdem sind die Roboter technisch bzw. mechanisch noch nicht in der Lage alle Feinheiten eines Fußballspiels zu vollbringen, wie zum Beispiel das Passspiel, welches es erst seit ein paar Monaten testweise bei einigen Mannschaften gibt. Im Gegensatz zu den richtigen Robotern wird auch schon nach offiziellen FIFA-Regeln gespielt.

Die Liga teilt sich ein in die 2D-Simulation und in die erst später dazugekommene 3D-Simulation. In dieser Arbeit wird aber wie bereits erwähnt nur die 2D-Simulation behandelt, da sich die Standards für die 3D-Simulation noch ständig ändern und es so schwierig ist eine robuste Visualisierung zu erstellen.

2.3 RoboCup Soccer Server

Der Soccer Server ist das System, welches den autonomen Agenten erlaubt ein Fußballspiel zu spielen. Das Spiel wird im Client/Server-Stil ausgetragen. Der Server liefert das virtuelle Spielfeld und simuliert alle Bewegungen des Balls und der Spieler. Jeder Client kontrolliert die Bewegungen eines Spielers. Die Kommunikation zwischen den einzelnen Spielern und dem Server findet über UDP/IP[5]-Sockets statt, man kann also ein Team auf unterschiedliche Weise programmieren (System und Sprache), man muss nur die Möglichkeit haben, die Daten über UDP/IP-Sockets zu senden und zu empfangen und sich dabei an die zur Kommunkation notwendigen Protokolle halten.

Der Soccer Server besteht aus zwei Programmen, dem `soccerserver` und dem `soccermonitor`. Der `soccerserver` ist wie schon beschrieben für die Bewegungen des Balls und der Spieler verantwortlich. Der `soccermonitor` hingegen ist ein Programm, welches das virtuelle Spielfeld über ein X-window System auf einem Monitor sichtbar machen kann. Außerdem erstellt der `soccermonitor` auch die Logdatei, in der sequentiell die aktuellen Spieldaten geschrieben werden.

Weitere Informationen zur Funktion des RoboCup Soccer Server findet man in [RC2001], Kapitel 4. Im Folgenden wird nun der Aufbau der Logdatei genauer beschrieben.

[5] User Datagram Protocol/Internet Protocol

2.4 Aufbau der Logdatei

Die Daten die vom Server ausgesandt werden, liegen in verschiedenen Formaten vor, welche bei der Initialisierung festgelegt werden können. Es gibt drei verschiedene Versionen, wie die Daten in die Logdatei geschrieben werden können.

Es handelt sich bei der Logdatei um eine Datei im Binärformat, bei den Daten handelt es sich um serialisierte C++-Strukturen, den `structs`. C++ ist auch die für die Arbeit verwendete Programmiersprache, dazu aber im nachfolgenden Kapitel mehr.

In der ältesten Version 1 der Logdateien steht nur eine Struktur, Die `dispinfo_t`-Struktur. Sie wird für jeden Zeitpunkt in die Datei geschrieben, so dass die Datei aus einer Aneinanderreihung von `dispinfo_t`-Strukturen besteht. Weitere Daten stehen nicht in der Logdatei.

```
typedef struct {
  short mode;
  union {
    showinfo_t show;
    msginfo_t  msg;
    drawinfo_t draw;
  } body;
} dispinfo_t
```

Die `dispinfo_t`-Struktur ist eine Vereinigung von drei anderen Strukturen, die aber nicht immer alle benötigt werden. Das führt dazu, dass viele Bytes geschrieben aber eigentlich nicht gebraucht werden. Die wichtigste der drei Strukturen ist die `showinfo_t`-Struktur, sie hat folgenden Inhalt:

```
typedef struct {
  char   pmode;
  team_t team[2];
  pos_t  pos[MAX_PLAYER * 2 + 1];
  short  time;
} showinfo_t
```

Die Struktur enthält zum Einen ein Zeichen, welches den aktuellen Spielzustand enthält, dann eine weitere Struktur `team_t`, die die Informationen zu den Mannschaften und dem aktuellen Spielstand enthält. Dann eine Reihe von Positionsinformationen für Ball und Spieler und zuletzt noch einen Zeitstempel, für den aktuellen Zeitindex. Die `msginfo_t`-Struktur enthält einen Index für die Art der Nachricht und die Nachricht selber als ein `char`-Array. Die `drawinfo_t`-Struktur enthält Informationen um die einzelnen Spieler und den Ball zu zeichnen.

Eine Verbessrung gab es als man Version 2 der Logdateien einführte und versuchte redundante oder nicht gebrauchte Daten zu vermeiden. Das hat allerdings zur Folge, dass man keine einheitliche Datenstruktur mehr hat, da jetzt verschiedene Strukturen in der Logdatei stehen und nicht nur eine. Da diese eine unterschiedliche Größe haben, ist es nicht möglich in der Datei einfach hin und her zu springen um verschiedene Daten zu lesen, da man nicht voraussagen kann, wie groß die einzelnen Abschnitte sind, die übersprungen werden sollen. Man muss also die Datei entweder einmal komplett einlesen oder sie sequentiell, das heisst nach und nach einlesen.

Das Format der Version 2 der Logdateien sah nun wie folgt aus:

- Head:
  Drei Zeichen 'ULG', die angeben, dass es sich um ein Unix-Logfile handelt.
- Version:
  Zeichen, dass die Version der Logdatei angibt
- Body:
  Im Rest der Datei stehen die eigentlichen Informationen

Man hat nun verschiedene Modi, um zu sagen, welche Information jetzt in der Datei folgen.

```
NO_INFO      0
SHOW_MODE    1
MSG_MODE     2
DRAW_MODE    3
BLANK_MODE   4
```

Bei einem `SHOW_MODE` folgt eine `showinfo_t`-Struktur und bei einem `MSG_MODE` folgt eine `msginfo_t`-Struktur. Andere Blöcke wie `DRAW_MODE` und `BLANK_MODE` werden nicht in die Logdatei geschrieben, sie sind nur für die Darstellung im `soccermonitor` wichtig. Außerdem gibt es immer noch einige Informationen, die man optimierter in die Datei schreiben könnte. Zum Beispiel könnten die Teamnamen Teil des Kopfes sein und somit nur einmal in die Datei geschrieben werden.

Version 3 der Logdatei bringt dazu einige Verbesserungen, sie hat nur Änderungen im Body. Es werden weitere Modi eingeführt um Platz zu sparen und um Daten für heterogene Spieler abzuspeichern.

```
PM_MODE      5
TEAM_MODE    6
PT_MODE      7
PARAM_MODE   8
PPARAM_MODE  9
```

Der PM_MODE gibt den aktuellen Zustand des Spieles an, eine Liste aller möglichen Zustände befindet sich in Anhang A. Dieser Block wird nur geschrieben, falls sich der Spielzustand ändert, bei einer Abseitsstellung zum Beispiel.

Der Zahl für den TEAM_MODE folgen zweimal eine team_t-Struktur, die jeweils den Namen und die Anzahl der Tore der Mannschaft enthalten. Dieser Block wird einmal zu Beginn eines Spieles geschrieben und dann immer nur wenn ein Tor gefallen ist.

Der PT_MODE wird nur einmal an den Anfang der Logdatei geschrieben, für jeden der 22 Spieler folgt eine player_type-Struktur.

Auch der PARAM_MODE steht nur einmal zu Beginn der Logdatei und enthält eine server_params_t-Struktur, die die aktuellen Serverparameter enthält.

Auch der neue PPARAM MODE wird nur einmal zu Beginn in die Datei geschrieben, er enthält die zusätzlichen Parameter für die Spieler der 3D-Simulation um heterogene Spieler zu haben, diese Informationen sind in den player_params_t-Strukturen gespeichert.

Eine Änderung gab es auch für den SHOW_MODE anstelle der showinfo_t-Struktur folgt jetzt eine showinfo_t2-Struktur. Diese Struktur enhält zusätzlich die Informationen zum Ball und den Spielern und hat folgendem Inhalt:

```
typedef struct {
  char      pmode;
  team_t    team[2];
  ball_t    ball;
  player_t  pos[MAX_PLAYER * 2];
  short     time;
} showinfo_t2;
```

Für die Visualisierung sind aber nur drei Modi wichtig, der SHOW_MODE, der PM_MODE und der TEAM_MODE. Darauf wird im Kapitel 7 näher eingegangen. Alle anderen Modi können beim Einlesen der Daten einfach übergangen werden. Um Kompatibilität unter verschiedenen Plattformen zu erreichen, werden alle Werte in der Netzwerkanordnung gespeichert, das heisst, dass Zahlen, die über das Netz verschickt werden unabhängig von der internen Zahlendarstellung des jeweiligen Rechners sein müssen. In der Netzwerkanordnung kommt immer das höherwertige Bit zuerst.

Einige Werte wurden zudem mit einer Konstanten SHOWINFO_SCALE2 multipliziert, damit diese Werte ganze Zahlen sind, dazu gehören zum Beispiel die x- und y-Positionen der Spieler und des Balls oder der Winkel wie der Körper der Spieler zum Spielfeld steht.

Weitere Informationen zu den einzelnen Strukturen lassen sich in der Anleitung zum RoboCup Soccer Server [RC2001], Kapitel 5 nachlesen.

Teil II
Techniken

3 C++

An dieser Stelle soll kurz darauf eingegangen werden, warum für das Projekt C++ als Programmiersprache verwendet wurde. C++ ist eine bevorzugte Entwicklungssprache für die Mehrheit der Programmierer, da diese schnelle, kleine Programme liefert. Und das Ziel dieser Arbeit ist es ein „kleines“ Tool zur Erstellung der Animationen zu entwickeln und nicht ein eher komplexes Programm mit grafischer Benutzeroberfläche zu erstellen. Das Tool soll wie ein Unix Tool zu verwenden sein, ein kleines Programm für gezielte Anwendungen. Zum Beispiel soll es sich auch mit weiteren Tools zu einer Prozeßkette verknüpfen lassen können.

Die Zielgruppe ist hier auch nicht der Endnutzer, also derjenige, der sich die Spiel anschauen möchte. Ein Interessent an den Simulationsspielen schaut sich nicht die Logdateien an oder versucht damit zu arbeiten, er möchte direkt etwas sehen und dazu dienen dann die Flashanimationen. Die Applikation richtet sich an diejenigen, die die Animationen aus den Logdateien erstellen möchten um sie so zu verbreiten, den Veranstaltern von RoboCup Spielen und Turnieren und außerdem an die Entwickler der Mannschaften.

Alle bisherigen Werkzeuge, wie der RoboCup Soccer Server zum Beispiel sind

in C/C++ geschrieben sind. Man kann dann dort vorhandene Ressourcen, wie zum Beispiel die Strukturen der Logdatei einfach übernehmen, ohne sich Gedanken über mögliche Strukturen bzw. Objekte bei anderen Programmiersprachen zu machen. Jede Änderung an der Logdatei kann so auch leicht in das Tool übernommen werden. Und alle, die für die RoboCup Soccer Simulation programmieren sind mit der Sprache und dem Layout der Logdatei vertraut. Der Quellcode des Tools soll veröffentlicht werden, damit das Tool weiterentwickelt werden kann, um neue Features hinzufügen zu können oder um ein komplett anderes Layout der Animation zu erstellen.

Der letzte und wahrscheinlich wichtigste Punkt für C++ als Programmiersprache ist das API zur Erstellung der Flashanimationen, es liegt nur in C++ vor. Es wäre daher sehr umständlich eine andere Sprache zu verwenden, wo man entweder eine Schnittstelle zu dem API schreiben oder ein eigenes API entwickeln müsste.

3.1 Kompilierungsmodell von C/C++

In diesem Abschnitt soll kurz vorgestellt werden wie eine C/C++-Klasse kompiliert wird, der Vorgang ist in drei Schritte unterteilt.

- **Präprozessor**
  Auflösung von Direktiven wie `#include`, `#define`, `#ifdef` etc.
- **Kompiler**
  Konvertierung von Quellcode-Dateien in Objekt-Dateien
- **Linker**
  Erzeugung einer ausführbaren Datei

Es können wie bei anderen Programmiersprachen auch üblich mehrere Quellcode-Dateien verwendet werden. Jede Quellcode-Datei „.c“ wird aber einzeln kompiliert, aus jeder Datei wird eine Objektdatei „.o“ erstellt. Zwischen diesen Dateien können aber verschiedene Beziehungen wie zum Beispiel externe Variablen offen bleiben. Zum Schluß führt der Linker die einzelnen Objektdateien zu einem ausführbaren Binary zusammen, dabei werden unter Umständen auch externe Bibliotheken eingefügt. Als weiterführende Literatur eignet sich [Cpp2000], eine umfassende Einführung in die Programmiersprache C++. Für das Projekt wurde eine `Makefile` erstellt, eine gute Anleitung dazu schrieb Stuart I. Feldman in [Make79].

4 Adobe Flash

Adobe Flash wurde als Ausgangsformat für die im Rahmen dieser Diplomarbeit entstehenden Animationen der RoboCup Soccer 2D Simulationen gewählt. Das folgende Kapitel zeigt einige Vor- und Nachteile einer Flashanimation gegenüber anderen Vektorgrafikformaten. Desweiteren soll das ShockwaveFlash-Dateiformat (SWF) genau vorgestellt werden, und außerdem wird eine Programmierschnittstelle (API) zur Erzeugung von Flashanimationen vorgestellt, die bei der Erstellung der Animationen benutzt wird. Zu Beginn soll aber die Entstehung von Flash beschrieben werden (Quelle: [Web FHist]).

4.1 Die Geschichte von Flash

Das Softwareunternehmen Macromedia mit Sitz im kalifornischen San Francisco entstand im Jahr 1992 durch den Zusammenschluss der Firmen MacroMind und Authorware. Drei Jahre später konnte Macromedia seinen Kundenstamm durch die Übernahme von Altsys, dem Hersteller des Grafikprogramms Freehand erstmals bedeutend vergrößern.

Ebenfalls 1995 veröffentlichte die Firma FutureWave aus San Diego das vektororientierte Illustrationsprogramm SmartSketch und ein dazugehöriges Plugin namens FutureSplash-Player, um die mit SmartSketch erzeugten Illustrationen mit einem Webbrowser betrachten zu können. Aufbauend auf SmartSketch entwickelte FutureWave 1996 das Animationsprogramm FutureSplash-Animator - den Vorläufer der heutigen Entwicklungsoberfläche Flash Studio.

Ende 1996 übernahm Macromedia die Firma FutureWave. Die Produkte FutureSplash Animator und FutureSplash-Player wurden unter dem Namen Flash und Shockwave Flash-Player weiterentwickelt. Flash 1 und 2 erschienen daraufhin im Jahr 1997, mit Flash 1 wurde die Einbindung von Audio- und Rastergrafikformaten integriert. Mit Version 2 wurde die Einbindung von Rastergrafiken wesentlich optimiert und dem Entwickler stehen neue Aktionen zur Verfügung, mit diesen lassen sich einfache Interaktionen umsetzen.

1998 wurd Flash Version 3 veröffentlicht und enthält bereits einen erweiterten Befehlssatz. Die Interaktionsmöglichkeiten wurden gegenüber den Versionen 1 und 2 stark erweitert. Nun lassen sich Aktionen auch auf Schlüsselbilder und Bilder zuweisen. Zusätzlich wird das Testen von Flash-Projekten während der Entwicklungsphase durch den in der Entwicklungsumgebung integrierten Player wesentlich erleichtert.

1999 erscheint Flash Version 4, sie enthält weitgehende Verbesserungen der nun integrierten Programmiersprache ActionScript. Es stehen erste Kontrollstrukturen wie Bedingte Anweisungen und Schleifen zur Verfügung, mit deren Hilfe die Entwicklung von Computer Based Training, POI oder Spiele-Projekten möglich

wird. Die Eingabetextfelder versetzen den Entwickler in die Lage, komplexe Formulare zu erstellen und die eingegebenen Daten über das Common Gateway Interface (CGI) zu empfangen, auszuwerten und mit dynamisch generierten Webseiten darauf zu reagieren.

Im Sommer 2000 veröffentlicht Macromedia die Version 5. ActionScript wurde in dieser Version stark verändert und an den ECMAScript-Standard angepasst, auf dem auch JavaScript aufbaut. Dadurch soll Entwicklern, die mit anderen Programmiersprachen zu tun hatten, der Einstieg in ActionScript erleichtert werden. Neue Objekte erleichtern die Integration externer Formate, darunter XML. Zusätzliche Hilfsmittel wie der Debugger erleichtern die Fehlersuche in Flash-Projekten.

Macromedia hat im Dezember 2000 weltweit mehr als 1200 Mitarbeiter und entwickelt außer Flash weitere Softwareprodukte im Bereich Webauthoring und Grafikerstellung (z.B. Macromedia Dreamweaver, Macromedia Fireworks). Laut einer unabhängigen Studie arbeiten zu dem Zeitpunkt mehr als eine Million professionelle Entwickler mit den Produkten von Macromedia.

Im März 2002 erscheint Flash MX. Diese Version hat eine umfangreichere Funktionsbibliothek. Bemerkenswert ist insbesondere die neue Zeichnen-API, die die Erstellung dynamischer Formen erlaubt. Weiterhin enthält diese Version einen Videocodec und Unterstützung für Unicode. ActionScript entspricht noch mehr dem ECMAScript-Standard. Zusätzlich wurde das Objekt- und Ereignismodell erweitert.

Im Mai 2002 klagt die Softwarefirma Adobe Systems 2,8 Millionen US-Dollar von Macromedia ein. Macromedia hatte eine Benutzeroberflächenkonzept verwendet, das Adobe Systems seit 1995 durch ein umstrittenes Softwarepatent für sich beansprucht. Kurz darauf revanchierte sich Macromedia mit mehreren Klagen, die ebenfalls auf Softwarepatenten beruhten, und gewann. Der von Adobe Systems zu zahlende Schadensersatz betrug fast 5 Millionen US-Dollar. Im Juli 2002 legten die beiden Firmen ihren Patentstreit jedoch außergerichtlich bei. Über die Bedingungen, die zu der Einigung führten, wurde allerdings nichts bekannt.

Im Oktober 2003 erscheint Flash MX 2004 und damit auch ActionScript 2.0. Die integrierte Programmiersprache wurde in Version 1.0 (objektbasiert) und 2.0 (objektorientiert) geteilt. Darüber hinaus lässt sich nun auch die Flash-API mit Hilfe von Flash-JavaScript komfortabler erweitern und auf die eigenen Bedürfnisse anpassen. Eine weitere Neuerung stellt die Integration zweier neuer Arbeitsweisen dar, Bildschirm- und Formularanwendungen sind nun auch in Flash realisierbar.

Im Jahre 2005 übernimmt Adobe Macromedia für 3,4 Milliarden US-Dollar. Mitte Juni stellt Macromedia die „Flash Platform“ vor, die vor allem Unternehmenskunden adressiert. Am 8. August wurde zusammen mit „Studio 8“ auch Flash Professional 8 vorgestellt. Die deutsche Version wurde im September veröffentlicht, ebenso wie der neue Flash Player 8. Zu den Neuerungen in Flash 8 gehören:

die Möglichkeit Rastergrafiken (Bitmaps) zu erzeugen oder zu verändern, Filter wie Gaußscher Weichzeichner, Schlagschatten oder Verzerrung, Blending-Modes wie in Photoshop, Datei-Upload, eine neue Text-Engine namens FlashType, Bitmap-Caching, einstellbares Easing, ein neuer Videocodec mit Alphakanal-Unterstützung, ein stand-alone Video-Encoder mit Stapelverarbeitung, sowie eine verbesserte Programmoberfläche.

Im Dezember 2005 schloss Adobe Systems die Akquisition von Macromedia ab. Zunächst führt Adobe die Bezeichnung Macromedia Flash für die Produkte weiter. Mit den nächsten Produktzyklen wurden dann alle Macromedia-Produkte in das Adobe-Namensschema überführt, weshalb die Technologie dann Adobe Flash heißt.

Im Sommer 2006 erschien der Adobe Flash Player 9 für Windows und Mac OS X. Er enthält Anpassungen für eine bessere Integration in Adobe Flex 2. Die neue ActionScript-Version 3.0 enthält E4X[6] und damit eine weitergehende Unterstützung für XML. Die Ausführung der Skripte wurde durch die Einführung eines Just-in-Time-Compilers beschleunigt. Im Januar 2007 wurde die Version 9 des Adobe Flash Players für Linux veröffentlicht.

4.2 Adobe Flash und seine Vorteile

Das Adobe Flash-Dateiformat bzw. das Shockwave Flash-Dateiformat (SWF) liefert Vektorgrafiken und Animationen für das Internet zum Adobe Flash Player. Es ist laut [Wol2002] der Standard für Vektorgrafiken und Animationen im Internet. Laut Adobe[7] haben circa 98% aller internetfähigen Desktops weltweit sowie zahlreiche andere Geräte den Adobe Flash Player installiert. Diese Aussage wird später in diesem Kapitel genauer untersucht.

Shockwave Flash ist ein zweidimensionales Vektorgrafikformat, in das man aber ebenfalls pixelbasierte Grafiken (Bitmaps) einbinden kann. Pixelbasierte Grafiken bestehen aus einer Matrix von Bildpunkten. die die Informationen zur Farbe der einzelnen Pixel enthalten. Vektorgrafiken hingegen werden mathematisch durch Verktoren und deren Eigenschaften beschrieben. Für eine Linie beispielsweise werden ein Anfangs- und ein Endpunkt definiert, sowie die Farbe und Dicke der Linie. Eine Kurve hat eine weitere Eigenschaft, die Krümmung muss zusätzlich angegeben werden (siehe Abbilung 4.1). Zusätzlich kann man durch Vektoren eingeschlossene Flächen mit einer Farbe bzw. einem Farbverlauf füllen.

Außerdem haben pixelbasierte Grafiken den entscheidenden Nachteil, dass sie nur für eine einzige Auflösung erzeugt wurden, sie sind also auflösungsabhängig. Wenn

[6] ECMAScript for XML (E4X) bringt native XML-Unterstützung für ECMAScript-konforme Programmiersprachen (z.B. JavaScript). Das Ziel ist es, eine alternative, einfachere Syntax für das Bearbeiten von XML-Dokumenten zu bieten als die bekannte DOM-Schnittstelle

[7] http://www.adobe.com/products/flashplayer/productinfo/faq/#item-1-3

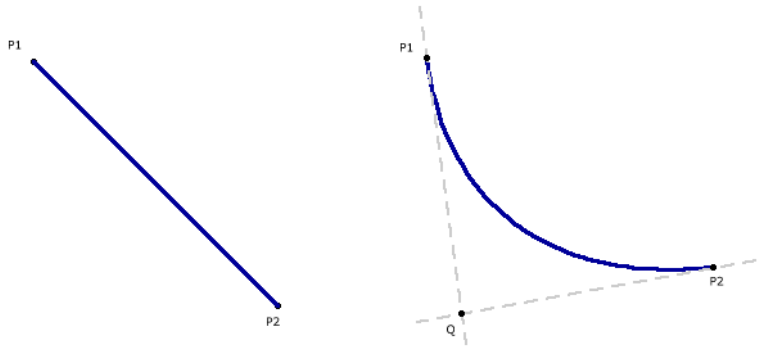

Abbildung 4.1: Links eine Linie, rechts eine Kurve mit Stützpunkt fur die Krümmung

man einen Ausschnitt einer Pixelgrafik vergrössert, werden auch die einzelnen Pixel größer, das Bild stellt sich dann wie ein Mosaik dar. Auch intelligente Anzeigetechniken, die in der Vergrößerungsstufe versuchen, durch Interpolation, d.h. das Einfügen von Pixeln mit Übergangsfarben, die Auflösung der Pixelgrafik zu erhöhen, bringen keine nennenswerten Verbesserungen. Das führt zu einer Art Schlierenbildung und die Grafik verliert an Schärfe.

Eine entscheidene Eigenschaft für die Verwendung von Vektorgrafiken ist die Möglichkeit, die Grafiken leicht zu animieren. Man kann Objekte leicht morphen (verändern) oder einfach versetzen, was für die im Rahmen dieser Arbeit zu erstellenden Animationen sehr wichtig ist.

Der eigentliche Vorteil von vektorbasierten Grafiken liegt darin, dass man sie in jeder Vergrößerungsstufe verlustfrei darstellen kann, sie sind auflösungsunabhängig. Die Darstellung der Vektorgrafik wird erst auf Clientseite, also für die entsprechende Auflösung und den ausgewählten Bereich, berechnet. So können auch vergrößerte Ausschnitte von Grafiken die maximale Auflösung des Ausgabegerätes nutzen.

Abbildung 4.2 zeigt den Buchstaben 'a' zwölffach vergrößert, einmal als Pixelgrafik, wo man die einzelnen Pixel deutlich erkennen kann, und rechts die Vektorgrafik. Da bei Vektorgrafiken nicht die Informationen für jeden einzelnen Bildpunkt in der Datei gespeichert werden müssen, sondern nur die Informationen der Linien und Kurven, sowie deren Farbe, Strichstärke und Füllung, sind Vektorgrafiken außerordentlich klein. Das ist ein weiterer Vorteil der in Flash verwendeten Vektortechnologie, im Normalfall sind Dateien mit Vektorgrafiken kleiner als pixelbasierte Grafiken sofern es nicht große Farbunterschiede gibt, wie auf Fotografien zum Beispiel. Das gilt sowohl für einzelne statische Bilder als auch für die erstellten Animationen. Damit kann man auch das Problem einer begrenzten Bandbreite des Internets umgangen werden, um eine Flashanimation herunterzu-

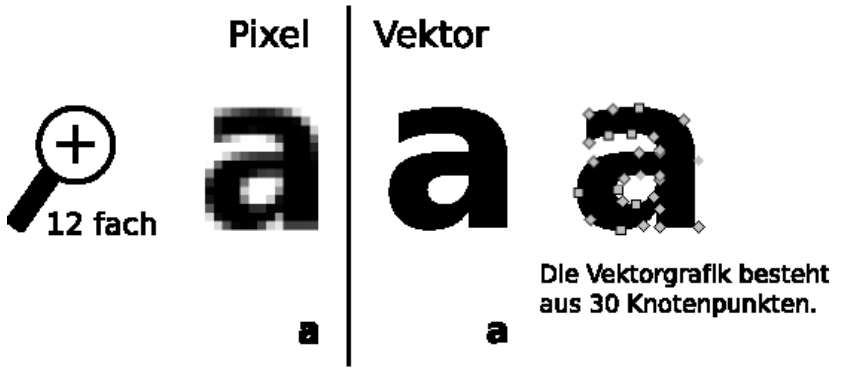

Abbildung 4.2: Unterschied zwischen Pixel- und Vektorgrafik

laden bzw. im Webbrowser anzuschauen. Das ist um ein Vielfaches schneller als wenn man sich ein Video der Animationen herunterladen bzw. direkt anschauen möchte.

Für die Darstellung von Flashgrafiken in einem Webbrowser benötigt man auf jedem System den Flash Player, den es kostenlos als Plugin für jeden (bekannten) Webbrowser oder aber auch als kostenpflichtige Standalone-Version gibt. Somit werden Flashgrafiken bei beliebiger Auflösung und Plattform identisch dargestellt, da der Flash Player die Darstellung steuert. Selbst Schriftarten stellen kein Problem dar, da sie fest in die Grafik (als Konturen) eingebaut werden können. Somit muss nicht jede Schriftart auf jeder Plattform installiert sein und es kann dadurch nicht zu Problemen kommen.

Das Grafikformat Shockwave Flash ist also auflösungsunabhängig, plattformunabhängig und bandbreitenschonend. Macromedia (jetzt Adobe) hat Flash aber nicht nur als Vektorgrafikformat entwickelt, sondern vor allem für interaktive Vektorgrafiken und Animationen.

Anfangs wurden aufgrund der langsamen Internetverbindungen mit Flash 1 und 2 nur interaktive Buttons und ähnliche Navigationselemente (Menüs) erstellt, erst später als ISDN und DSL zur Verfügung standen ging man dazu über, vollflächige Animationen mit Flash zu gestalten, da mit diesen Techniken die Datenübertragung kein großes Problem mehr ist. Seitdem erstellen zahlreiche Internetagenturen komplette Webauftritte, Produktpräsentationen und Werbespots in Flash.

Vektor- und Bitmaptransparenz, komplexe Navigationen, problemlose Einbindung von Sound im komprimierten MP3-Format und der Einsatz beliebiger Schriftarten werden für Effekte aller Art genutzt, Überblendeffekte, Bewegung und sogar Morphing sind möglich geworden. Man spricht deshalb auch von „Flash-Filmen“.

Für Interaktionen mit dem Benutzer stehen Texteingabefelder und eine Skriptsprache namens ActionScript zur Verfügung. Aktionen können an bestimmte Bilder einer Filmsequenz oder an Buttons gekoppelt werden und den Ablauf der

Animation steuern, neue Filme beziehungsweise Webadressen laden oder Eigenschaften der momentanen Grafikobjekte verändern.

Alle Objekte einer Flashanimation liegen meist in einer einzigen Datei, zusätzlich können aber auch externe Dateien genutzt werden. Eine Animation ist außerdem streamingfähig, das heißt, dass der Benutzer die ersten Frames (Einzelbilder) der Animation bereits sehen kann, während die restlichen Frames im Hintergrund noch geladen werden.

Es gibt bei Computeranimationen zwei Arten, wie Bewegungen entstehen. Zum einen durch Morphing, das ist das Überblenden eines Grafikobjekts in ein anderes oder als zweite Art durch zeitbasierte Veränderungen. Dabei berechnet der abspielende Computer die einzelnen Veränderungen, dadurch wird die Dateigröße weiter verringert. Die Steuerung der einzelnen Animationsphasen erfolgt dabei über die sogenanten Keyframes, die in es in vielen Multimedia-Anwendungen gibt, wie beim Videoschnitt oder ähnlichem - dabei beschreiben Schlüsselbilder (Keyframes) die Animationsphasen auf der Zeitleiste. Die Bewegungen entstehen durch Interpolation zwischen den einzelnen Keyframes.

In den letzten Absätzen wurden die Vorteile von Vektorgrafiken und -animationen, insbesondere von Flashanimtionen beschrieben, aber es gibt natürlich auch ein paar Einschränkungen.

Natürlich kann man mit Flash ansprechende und animierte Webseiten erstellen, doch die Gestaltung eines Webauftrittes mit einfachen HTML-Dokumenten und darin eingebundenen Pixelgrafiken ist meist weniger aufwändig und auch kostengünstiger. Außerdem ist die zu transferierende Datenmenge bei Flash-Dokumenten immer noch um ein Vielfaches höher als die für ein reines HTML-Dokument, wenn nur textbasierte Informationen übertragen werden sollen, denn ein HTML-Dokument wird für die Übertragung auch komprimiert.

Bei der ausschließlichen Verwendung von Flash für Web-Seiten kann außerdem die Benutzerfreundlichkeit stark eingeschränkt sein. Sie behindern den Zugang unter anderem für Menschen mit eingeschränkten Fähigkeiten, gleichzeitig aber auch für Benutzer rein textbasierter Browser, welche Flashdateien nicht darstellen können.

Im Vergleich zu eines mit Flash gestalteten Webauftrittes stehen kostenfreie Editoren für HTML zur Verfügung. Mit PHP und dem Apache Webserver lassen sich kostenfrei dynamisch erzeugt Webseiten erstellen. Im Gegenzug dazu kommen zu den knapp 200 Euro für die Flash Entwicklungsumgebung noch über 1000 Euro pro Jahr für Adobe Flex, dem Webserver-Produkt von Adobe, falls man mit Flash dynamische Seiten erzeugen möchte.

Tabelle 1 zeigt deutlich die hohe Verfügbarkeit des Flash Player, die von Adobe angegebenen 98% werden zwar nur für ältere Versionen des Flash Players erreicht, aber da für die Realisierung der Applikation nur ein API für Version 4 vorliegt

	Version 6	Version 7	Version 8	Version 9
Gesamt	98.3%	97.3%	94.2%	55.8%
USA	98.4%	97.6%	94.2%	56.4%
Europa	98.0%	96.4%	93.0%	59.0%
Japan	98.5%	98.0%	95.6%	50.3%
Andere	95.9%	91.0%	86.4%	65.4%

Tabelle 1: Verfügbarkeit des Macromedia Flash-Players (Dezember 2006, [Web FPoll])

kann man von einem sehr hohen Verbreitungsgrad ausgehen. Nach Version 4 gab es auch kaum richtig großen Änderungen für das SWF-Dateiformat selber, es wurde nur um einige Objekte erweitert (Quelle: [Adb2005]). Die dritte Version führt zum Beispiel Vektortransparenz, Morphing und Actions ein. Flash 4 erweitert die Skriptsprache ActionScript, bringt Eingabetextfelder zur Erstellung von Formularen mit und kann das Audioformat MP3 verarbeiten. Die Neuerungen von Flash 5 betreffen schließlich vor allem die Bedienfreundlichkeit von Flash. Die grafische Entwicklungsumgebung ähnelt nun dem Erscheinungsbild anderer Macromedia-Produkte und die ActionScript-Syntax wurde an das bekannte JavaScript angepasst und um einen Debugger erweitert. Mit Flash 6 gab es einen neue Signatur für die Dateien, die Datei werden ab dort zusätzlich komprimiert, dazu später mehr. Zusätzlich gab es noch einige Erweiterungen von ActionScript und das FlashVideo-Dateiformat (FLV) wurde eingeführt. Mit Version 7 und 8 des Players gab es nur weitere Verbesserungen bei ActionScript und FlashVideo. Die ältere API, die zur Verfügung steht lässt noch kein ActionScript zu, das ist für die Animationen, die erstellt werden sollen aber auch nicht nötig.

Die Verbreitung des Flash Player-Plugins basiert nicht nur darauf, dass das derzeit ca. 1,3 MB große Plugin (oder dessen Vorgänger) von fast jedem (98% aller) Internetnutzer heruntergeladen wurde. Adobe und vorher Macromedia haben eine geschicktere Strategie gewählt: Durch Kooperationen mit Firmen wie Microsoft und Apple wird es inzwischen mit jedem neuen Betriebssystem ebenso wie mit jedem neuen Webbrowser für diese Systeme (Internet Explorer und Safari) standardmäßig mitgeliefert. Einzige Ausnahme ist der momentan populäre Mozilla Firefox, bei dem das Plugin zusätzlich installiert werden muss. Seit 1999 existieren außerdem Versionen des Plugins für Linux und Solaris. Das Flash-Plugin steht damit heute auf neuen Rechnern normalerweise automatisch zur Verfügung und selbst Unix-Systeme werden unterstützt.

Um positiv auf die Verbreitung und die Akzeptanz des Flash-Dateiformates einzuwirken und sich damit gegen ehemalige Konkurrenten wie Adobes PGML zu schützen, veröffentlichte Macromedia bereits im Juni 1998 im Rahmen des „Flash Open File Format“-Programms das Dateiformat der Flashanimationen. Macromedia verkündete offiziell, Flash damit als offenen Internet-Standard etablieren

zu wollen.

Ein erstes Ergebnis der sogenannten “SWF file format specification“ war Oliver Debons Flash-Plugin für Netscape unter Linux, das vor dem offiziellen Plugin von Macromedia zum Download bereit stand.

Die noch unvollständige und teilweise fehlerhafte Dokumentation des Dateiformates löste Macromedia Anfang 2000 durch das SWF Software Development Kit (kurz: SWF-SDK) ab, das kostenlos aus dem Internet zu beziehen war. Darin beschreibt die vollständig überarbeitete Spezifikation sehr ausführlich das Dateiformat und eigene SWF-Dateien lassen sich mit einer Sammlung von C++-Klassen erzeugen. Die Idee von Macromedia ist offensichtlich: Mit Hilfe des SWF-SDK lassen sich Flashanimationen über eine Programmierschnittstelle erzeugen, was Dritthersteller ermutigen soll, Unterstützung für das SWF-Dateiformat in ihre Programme zu integrieren. Adobe hat dies bereits für seine Produkte Illustrator und LiveMotion getan, Corel integrierte eine Exportschnittstelle für SWF-Dateien in CorelDRAW.

Aber nicht nur die Verbreitung des Flash-Dateiformates wird von Macromedia forciert. Zeitgleich mit dem SWF-SDK zur Erzeugung von Flashanimationen veröffentlichte Macromedia die Quellen des Flash Players. Softwarehersteller können sich von Macromedia lizensieren lassen und so ihre Produkte mit Hilfe der Flash Player-Quellen um die Fähigkeit der Wiedergabe von Flashanimationen erweitern.

Für die vorliegende Diplomarbeit ist das genannte SWF-SDK sehr interessant, mit ihm lassen sich ohne Einsatz des kostenpflichtigen Flash Studios Flashanimationen erzeugen. Aus den vorhandenen Logdateien können die Daten so mit Hilfe eines Programmes in eine Flashanimation umgewandelt bzw. gespeichert werden.

Der folgende Abschnitt geht genauer auf das SWF-Dateiformat ein, der darauffolgende Abschnitt widmet sich dann schließlich dem SWF Software Development Kit. Bei der Benutzung des SDK ist es sehr hilfreich sich genauer mit dem SWF-Format auseinanderzusetzen. Zwar wird durch die Benutzung des API der Umgang mit dem Format sehr erleichtert, weil anschauliche Objekte zur Kapselung von Informationen verwendet werden, aber um die Idee und Struktur eine Flashanimation zu verstehen wird nun das SWF-Dateiformat beschrieben.

4.3 Das ShockwaveFlash-Dateiformat (SWF)

Das Dateiformat der Flashanimationen wurde von Macromedia mit dem Ziel entworfen, Vektorgrafiken und Animationen über das Internet zu übertragen. Deshalb wurde es als effizientes Übertragungsformat und nicht als Austauschformat zwischen verschiedenen Grafikeditoren entwickelt. Macromedia gibt in der Spezifiaktion [Adb2005] an, dass das SWF-Dateiformat die folgenden Gesichtspunkte erfüllt:

- **Bildschirmdarstellung**
  Das Format wurde hauptsächlich zur Darstellung auf einem Bildschirm entworfen und unterstützt deshalb Anti-Aliasing, schnelles Rendering in ein Bitmapformat mit beliebiger Farbtiefe, Animation und interaktive Schaltknöpfe.

- **Erweiterbarkeit**
  Das Format ist tagbasiert, d.h. die internen Daten werden durch Tags (Marken) in Blöcke bestimmter Typen eingeteilt. Indem der Flash Player unbekannte Tags und damit den entsprechenden Datenblock ignoriert, kann das Format um neue Möglichkeiten erweitert werden, während die Kompatibiliät zu älteren Flash Playern erhalten bleibt.

- **Netzwerkübertragung**
  Die Dateien können über ein Netzwerk mit geringer Bandbreite übertragen werden, da sie komprimiert und streamingfähig sind. SWF ist ein binäres Dateiformat und damit nicht mit einem Texteditor lesbar wie z.B. HTML. Techniken wie Bit-Packing und Strukturen mit optionalen Feldern werden benutzt, um die Dateigröße zu minimieren.

- **Einfachheit**
  Das Format ist einfach gehalten, damit der Flash Player von geringer Größe und leicht auf andere Systeme zu portieren ist. Der Flash Player greift außerdem auf nur wenige Betriebssystemfunktionen zu.

- **Datenunabhängigkeit**
  Die Dateien können unabhängig von externen Datenquellen wie z.B. Schriftendefinitionen dargestellt werden.

- **Skalierbarkeit**
  Unterschiedliche Computer besitzen unterschiedliche Bildschirmauflösungen und Farbtiefen. Für die Darstellung der Flashanimationen reicht eine einfache Hardware-Ausstattung aus, der Vorteil leistungsstärkerer Hardware wird jedoch genutzt, sofern diese vorhanden ist.

- **Geschwindigkeit**
  Das Format unterstützt sehr schnelles Rendering bei hoher Qualität.

Bevor wir nun zum Aufbau der SWF-Datei kommen sollen noch kurz die Datentypen von Flash vorgestellt werden. Damit die Animationen möglichst platzsparend sind, wird für Koordinaten-, Transformations- oder andere Werte immer der kleinstmögliche Datentyp genutzt. Die wichtigsten Datentypen im SWF-Format sind in der Tabelle aufgeführt.

Aus diesen Werten werden weitere Datentypen abgeleitet, z.B. der Datentyp `RECT`, der exemplarisch näher aufgeschlüsselt werden soll. Der Datentyp `RECT`

Datentyp	Erläuterung
SI8	Signed 8-bit Integer Wert
SI16	Signed 16-bit Integer Wert
SI8[n]	Array mit n Signed 8-bit Integer Werten
SI16[n]	Array mit n Signed 16-bit Integer Werten
UI8	Unsigned 8-bit Integer Wert
UI16	Unsigned 16-bit Integer Wert
UI32	Unsigned 32-bit Integer Wert
UI8[n]	Array mit n Unsigned 8-bit Integer Werten
UI16[n]	Array mit n Unsigned 16-bit Integer Werten
SB[nBits]	Signed Bit Wert (n Bits werden genutzt)
UB[nBits]	Unsigned Bit Wert (n Bits werden genutzt)

Tabelle 2: Die einfachen Datentypen des SWF-Dateiformat (Quelle: [Adb2005])

enthält die Koordinaten für ein Rechteck und wird unterschiedlich genutzt, z.B. als Definition für die Größe einer Flashanimation oder zur Definition eines Rechteckobjektes in der Animation.

Datenfeld	Datentyp	Erklärung
NBits	UB[5] = nBits	Anzahl Bits in den folgenden Feldern
Xmin	SB[nBits]	X-Minimum Position des Rechtecks
Ymin	SB[nBits]	Y-Minimum Position des Rechtecks
Xmax	SB[nBits]	X-Maximum Position des Rechtecks
Ymax	SB[nBits]	Y-Maximum Position des Rechtecks

Tabelle 3: Der Datentyp RECT

In der Datenstruktur RECT ist zunächst ein Feld von fünf Unsigned Bits enthalten. In diesem Feld wird festgelegt, wieviele Bits für die weiteren Datenfelder benötigt werden. So ist es möglich, die Bitlänge für unterschiedlich große Koordinaten variabel zu halten ohne spezielle Markierungen einzuführen, wann der nächste Koordinatenwert beginnt. Dies wird einfach über die vorher definierte Länge ermittelt. So wie die RECT-Struktur werden auch weitere Strukturen definiert, auf die hier nicht näher eingegangen werden soll, weitere Informationen sind in der Dokumentation [Adb2005] nachzulesen.

Alle SWF-Dateien beginnen mit einem Header (Tabelle 4), der mit der Signatur `FWS` und ab Version 6 `CWS` beginnt, das “C“ ab Version 6 sagt aus, dass es sich um eine komprimierte Animation handelt, da zu später mehr. Nach diesen initialen Zeichen folgt die Versionsnummer des Dateiformates. Danach folgen die Dateigröße in Bytes und die Breite und Höhe der Flashanimation. Dann wird die Framerate angegeben, wieviele Einzelbilder pro Sekunde angezeigt werden sollen und danach wird noch die Anzahl der Frames angegeben.

Inhalt	Datentyp	Erläuterung
Signature	UI8	Seit Version 6 'C', vorher 'F'
Signature	UI8	Immer 'W'
Signature	UI8	Immer 'S'
Version	UI8	Flashversion (z.B. 0x04 für SWF4)
FileLength	UI32	Länge der gesamten Datei
FrameSize	RECT	Framegröße in TWIPS
FrameRate	UI16	Bildwiederholrate: Frames pro Sekunde
FrameCount	UI16	Anzahl der Frames im Flashfilm

Tabelle 4: Dateikopf einer Flash-Animation

Nach dem Header folgen dann die einzelnen Blöcke (Tags). die die einzelnen Daten zur Darstellung und Steuerung enthalten, zum Schluss gibt es dann einen abschließenden Endblock.

Abbildung 4.3: Struktur einer SWF-Datei

Alle Tags haben ein einheitliches Format, welches im Folgenden näher erläutert werden soll. Tags die vom Interpreter (dem Flash Player) nicht verstanden werden, können so einfach übersprungen werden, außerdem kann ein passendes Werkzeug (Programm) so leicht einzelne Tags modifizieren, löschen oder hinzufügen.

Jeder Block beginnt mit einer Zahl, die angibt von welchem Typ der Block ist, bei längeren Blöcken (größer 63 Bytes) folgt noch die Gesamtlänge des Blockes und dann der eigentliche Inhalt.

Inhalt	Datentyp	Kommentar
Taglänge	UI16	10 Bit für ID des Tags 6 Bit für die Länge des Tags

Tabelle 5: Block mit einer Gesamtlänge kleiner 63 Bytes

Inhalt	Datentyp	Kommentar
Taglänge	UI16	10 Bit für ID des Tags 6 Bit: immer `0x3F`
Länge des Blocks	UI32	

Tabelle 6: Block mit einer Gesamtlänge größer 63 Bytes

Man unterscheidet desweiteren zwei Arten von Blöcken, den Definitionsblock (Definition Tag) und den Kontrollblock (Control Tag), im Definitionsblock werden alle Objekte definiert, Shapes, Texte, Bitmaps etc. Jeder dieser Tags bekommt eine eindeutige ID zugewiesen, die der Player in einem Verzeichnis (*Dictionary*) speichert, so dass er auf die Tags zugreifen kann. Ein Definitionsblock hat keinerlei Funktionalität um sich zu rendern, dafür gibt es die sogenannten Control Tags, diese Blöcke instanziieren einen Tag aus dem *Dictionary* und können so die Animation steuern, sie manipulieren einzelne Objekte, löschen sie oder fügen sie der Animation hinzu. Alle aktuellen Objekte werden vom Player in der sogenannten *DisplayList* gespeichert, die aktuellen Objekte sind die, die angezeigt werden sollen. Und jedes Mal wenn ein neuer Frame kommt wird die aktuelle *DisplayList* gezeichnet.

Zur genauen Erläuterung folgt ein kleines Beispiel:

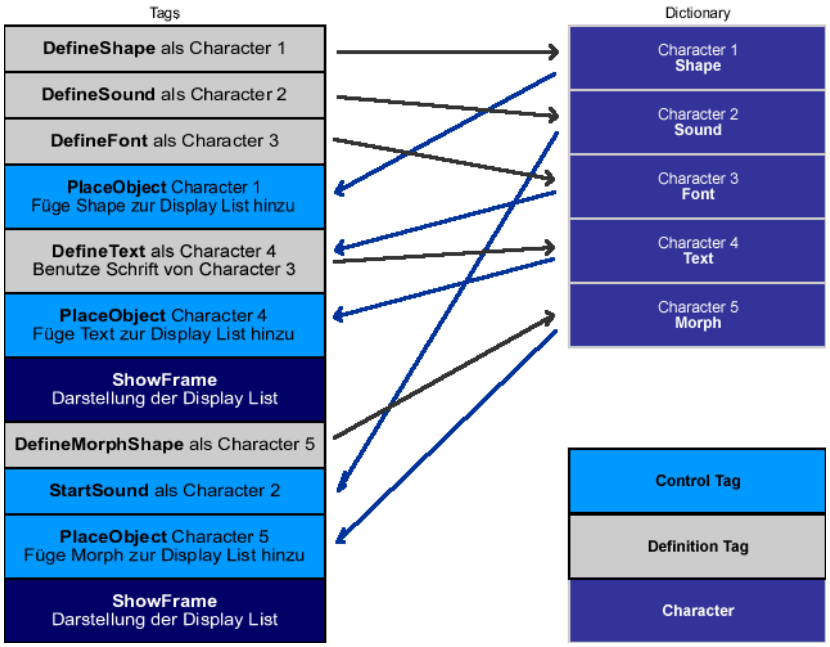

Abbildung 4.4: Ablauf einer Animation

Abbildung 4.4 zeigt einen kleinen Ausschnitt aus einer Flashanimation, links sieht man den Inhalt der Datei, rechts oben das virtuelle *Dictionary* des Flash Players. Als erstes wird ein *Shape*, ein *Sound* und ein *Font* definiert und dem *Dictionary* hinzugefügt. Dann wird das *Shape* (Character 1) durch *PlaceObject* der *DisplayList* hinzugefügt. Anschliessend wird mit Hilfe des *Font* (Character3) ein Text definiert und als Character 4 in das *Dictionary* abgespeichert, dieses wir dann wiederum durch *PlaceObject* der *DisplayList* hinzugefügt. Durch *ShowFrame* werden alle Objekte der *DisplayList* gezeichnet und der nächste Frame beginnt. Hier wird ein *MorphShape* definiert (Character 5). Danach wird

der *Sound* (Character 2) gestartet und das *MorphShape* wird dann anschliessend der *DisplayList* hinzugefügt. Und dann wird durch *ShowFrame* dieser Frame dargestellt. Alle vorher vorhandenen Objekte sind und bleiben natürlich in der *DisplayList*. Nur durch ein *RemoveObject* kann man einzelne Objekte aus der *DisplayList* entfernen.

Bevor wir nun im nächsten Abschnitt genauer auf die einzelnen verschiedenen Objekte einer Animation zu sprechen kommen, sollen noch kurz einige Strategien zur Komprimierung von Flashdateien vorgestellt werden. Es gibt mehrere Strategien, die Dateien zu komprimieren um sie möglichst klein zu halten, damit sie leicht über Netzwerke, wie das WorldWideWeb (WWW) verschickt bzw. bereitgestellt werden können.

Zum einen gibt es die Wiederverwendbarkeit der Objekte die im Dictionary stehen. Ein Shape kann einmal erstellt werden, es kann aber mehrfach darauf referenziert werden, d.h. das Objekt kann mehrfach platziert werden.

Die Kompression der einzelnen Shapes ist ein weiterer Aspekt der Kompression. Bei einem *Shape* ist es oft so, dass die Endkoordinaten einer Linie die Anfangskoordinaten der nächsten Linie sind. Distanzen sind also oft relativ zu ihrer letzten Position angegeben.

Viele Objekte wie Matrizen und Farbtransformationen haben einige Felder die öfter genutzt werden als andere. zum Beispiel bei Matrizen ist die Translation das meistgenutzte Feld, Skalierung und Rotation werden hingegen weniger genutzt. Falls die Skalierung also nicht angegeben wird, wird ein Skalierungsfaktor von 1 in beide Richtungen angenommen. Bei der Rotation wird der Wert 0 angenommen.

Außerdem werden immer nur die Änderungen eines *Shapes* bzw. Objektes in der SWF-Datei abspeichert. Das entspricht dem sogenannten *Place/move/remove* - Modell der *DisplayList*. Objekte werden entweder gesetzt, transformiert oder gelscht.

Eine vereinheitlichte Struktur der *Shapes* hilft die Objekte möglichst klein zu halten und das Rendern von anti-aliased Objekten auf dem Monitor effizienter zu machen.

Neben den aufgeführten Ideen des Dateiformates ist vor allem die tagbasierte Strukturierung der Daten interessant für die spätere Erzeugung von Flashanimationen mit dem SWF-SDK. Im folgenden Abschnitt soll nun auf einige C++-Klassen des SWF-SDK und dessen Anwendung eingegangen werden.

4.4 Das SWF-SDK

Das „Macromedia Flash File Format (SWF) Software Development Kit (SDK)“, ursprünglich von der Firma MiddleSoft entwickelt, stellt Klassen für die Erstellung von Flashanimationen zu Verfügung. Allerdings wurde seine Entwicklung nach Version 4 eingestellt, nachdem MiddleSoft von Macromedia übernommen wurde. Seit dem stehen Entwicklern die Flash Dateiformat Spezifikationen nur in Papierform zur Verfügung. Das SDK ist in C++ verfasst, das ist für das Projekt dieser Arbeit von Vorteil, da die Applikation in C++ verfasst werden soll, da alle Programme, wie der Robocup Soccer Server in C++ verfasst sind und das Tool somit von der Programmiersprache gut in das Schema des RoboCup passt. Auch das nur Animationen bis Version 4 und mit einigen leichten Änderungen auch bis Version 5 zu programmieren sind, ist ausreichend, da nur selbstlaufende Animationen mit rudimentären Steuerelementen erstellt werden sollen. Die Verwendung von ActionScript ist hier nicht nicht nötig, da alle verwendeten Elemente schon in Flash 4 zur Verfügung stehen

Es gibt zwar auch aktuelle Versionen von Drittanbietern, bei denen es aber leider einige Fehler gibt. Erwähnenswert is hier nur die Firma Flagstone Software[8], die eine API für Java für die Flashversion 8 entwickelt hat. Bei einigen Tests kam es dort aber zu Fehlern bei der Rotation, so dass doch wieder auf die alte funktionierende Version des SDK zurückgegriffen wurde. Neben den Klassen enthält das SDK natürlich noch eine kurze Dokumentation und einige Beispiele.

Die mitgelieferten Klassen sind mit einer Einschränkung frei zu verwenden. Man darf sie nur zur Entwicklung von SWF-exportierenden Programmen nutzen. Macromedia verlangt, dass die mit Hilfe des SDK entwickelten Produkte Flashanimationen im korrekten Format speichern. Die so erzeugten Flash-Dateien müssen erstens fehlerfrei mit einem aktuellen Flash Player abgespielt werden können und zweitens fehlerfrei in die grafische Entwicklungsumgebung Flash importiert werden können. Macromedia übernimmt keine Garantie für die Korrektheit der bereitgestellten Software und behält sich vor, das SDK nach Bedarf zu aktualisieren. Nun wird das SDK von Adobe und vorher auch schon von Macromedia aber nicht mehr unterstützt. Das SDK gibt es leider nicht mehr zum freien Download, dadurch wird die Entwicklung eines Flash-Tools nahezu unmöglich.

Die Klassen des SDK lassen sich in zwei Bereiche unterteilen, zum einen den Low Level Manager und zum anderen in den High Level Manager. Der High Level Manager zum Beispiel stellt eine Klasse `HFPolygon` zur Verfügung, die Methoden zum Erzeugen eines Polygons hat. Aber um einen Linienzug zu erzeugen müssen Linie für Linie bzw. Kurve für Kurve einzeln erstellt werden und dem Polygon hinzugefügt werden. Das Polygon wird dann über weitere Methoden der Animation hinzugefügt. Man kann aber auch im Low Level Manager direkt die Linien

[8] http://www.flagstonesoftware.com

erstellen und sie zu einem Shape zusammenfassen, im Endeffekt kommt dabei dasselbe heraus. Der Umgang mit den einzelnen Bereichen bleibt Geschmackssache, Vorteile oder Nachteile lassen sich nur schwer herausarbeiten. Der High Level Manager bietet Klassen an, die ein Objekt auf einfache Art erstellen lassen, ohne die Struktur einer Flashdatei genau aufzuzeigen. Auf den High Level Manager soll aber nicht genauer eingegangen werden, da ausschliesslich die Klassen des Low Level Manager für das Projekt benutzt wurden. Der Low Level Manager bildet die Struktur einer Flashdatei sehr genau ab, so kann man besser Objekte zeichnen, diese besser kontrollieren und ausserdem mehrere Optionen für Objekte setzen.

Für jeden Defintions- und Kontroll-Block existiert eine Klasse, die von der Basisklasse `FObj` abstammt. Die Animation selber wird von der Klasse `FObjCollecton` erstellt. Sie sammelt die `FObj`s und schreibt sie schließlich in eine Datei. Zusätzlich gibt es noch Klassen für Linien, Kurven, Farben etc. Es gibt außerdem noch Hilfsklassen, die das Einbinden von Bildern oder Sounddateien ermöglichen.

FObjCollection	sammelt Tags (FObjs) und schreibt die Datei
FObj	Basisklasse aller Tagklassen
Control Tag Classes	Kontrolltags (beginnen mit FCT)
Definition Tag Classes	Definitonstags (beginnen mit FDT)
Primitive Data Classes	weitere Records wie Füllstil, Farbe, Linie, ...
Helper Classes	Hilfsklassen für Sound und Pixelgrafiken

Tabelle 7: Übersicht des Low Level Managers

Die Arbeitsweise des Low Level Managers lässt sich am Besten anhand eines Beispieles erklären. Das folgende Codebeispiel zeichnet ein rotes Quadrat mit schwarzem Rand auf einen grauen Hintergrund.

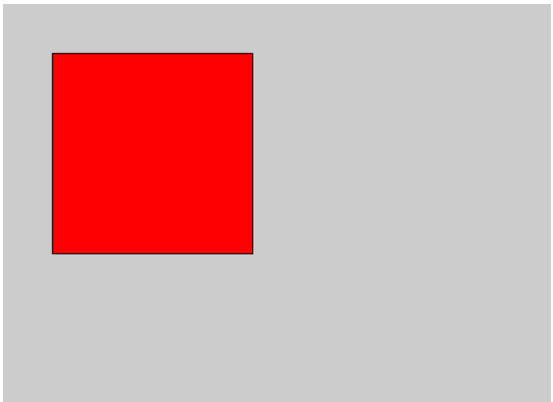

Abbildung 4.5: Beispiel einer Flashgrafik

```
#include "F3SDK.h"  // einbinden des SDK

void CreateRectangleMovie(){

  FObjCollection animation;

  const FColor grey(0xaa, 0xaa, 0xaa);
  FCTSetBackgroundColor *background =
                          new FCTSetBackgroundColor(new FColor(grey));
  allTags.AddFObj(background);

  FRect *rectBounds = new FRect(1000, 1000, 5000, 5000);

  FDTDefineShape *rectangle = new FDTDefineShape(rectBounds);
```

Am Anfang wird die `FObjCollection` erstellt, die alle Objekte der Animation speichert. Danach wird ein Rechteck erstellt in dem das eigentliche *Shape* gezeichnet wird, Größenangaben werden in Flash stets in Twips (TWentieth of an Inch Point) abgespeichert, 1 Twip entspricht dabei 1/20 Pixel. Das heisst, im Beispiel ist das Rechteck nicht 4000x4000 Pixel gross, sondern nur 200x200 Pixel. Das Rechteck wird dem Shape zu gewiesen.

```
  U16 rectangleID = rectangle->ID();

  FColor red = FColor(0xff, 0, 0);
  U32 redfillID = rectangle->AddSolidFillStyle(red);

  FColor black = FColor(0, 0, 0);
  U32 blackLineStyleID = rectangle->AddLineStyle(20, black);

  rectangle->FinishStyleArrays();
```

In diesem Abschnitt wurden dem *Shape* eine rote Füllung und ein schwarzer Rand zugewiesen. Mit `FinishStyleArrays()` wird angezeigt, dass die Daten für das Aussehen alle übergeben wurden. Ab hier beginnt das eigentliche Zeichnen des *Shapes*.

```
  FShapeRec *rectangleShapeRecords[6];
  rectangleShapeRecords[0] =
           new FShapeRecChange(false, true, false, true,
                               5000, // xPos
                               1000, // yPos
                               0,
```

```
                                redfillID,
                                blackLineStyleID,
                                0, 0);

  rectangleShapeRecords[1] = new FShapeRecEdgeStraight(0, 4000);
  rectangleShapeRecords[2] = new FShapeRecEdgeStraight(-4000, 0);
  rectangleShapeRecords[3] = new FShapeRecEdgeStraight(0, -4000);
  rectangleShapeRecords[4] = new FShapeRecEdgeStraight(4000, 0);
  rectangleShapeRecords[5] = new FShapeRecEnd();

  for (int i = 0;  i < 6 ;  i++)
    rectangle->AddShapeRec(rectangleShapeRecords[i]);

  allTags.AddFObj(rectangle);
```

Es wird ein *ShapeRecord* erstellt in dem man die Linien (aber auch Kurven) speichert. Am Anfang wird ein Punkt, die Linien- und Füllfarbe übergeben, bei den weiteren Objekten werden immer nur relative Angaben gemacht, wie weit man von aktuellen Punkt in x- und y-Richtung geht. Dieses *ShapeRecord* wird dann dem *Shape* hinzugefügt, welches wiederum der *Collection* hinzugefügt wird.

```
  FCTPlaceObject2 *placeRectangle =
                  new FCTPlaceObject2(false, false,
                                      true, false,
                                      1, // Tiefe
                                      rectangleID,
                                      0, 0, 0, 0, 0);

  animation.AddFObj(placeRectangle);

  FCTShowFrame *showFrame = new FCTShowFrame();
  animation.AddFObj(showFrame);
  animation.CreateMovie("ExampleRectangle.swf", 11000, 8000, 12);
}
```

Über *PlaceObject* wird dann das *Shape* in der Szene platziert. Mit *ShowFrame* wird dann noch die aktuelle *DisplayList* gerendert, hier also das rote Rechteck. Am Schluß wird die Animation mit der Methode `CreateMovie` gespeichert, ihr wird der Name der Animation, die Höhe und die Breite sowie Bildwiederholrate übergeben. Kompiliert und startet man den Quellcode des Beispiels, wird die Flashgrafik aus Abbildung 4.5 erstellt.

Die Elemente einer Flashanimation können mit Hilfe eines Übergabeparameters in verschiedene Ebenen positioniert werden. Höhere Ebenen liegen über niedrigeren und verdecken die darunterliegenden Objekte bzw. lassen diese je nach

Transparenz ihrer Füll- und Linienfarbe durchscheinen. Die Ebene, in der ein Grafikobjekt platziert ist, wird benötigt, um dieses Grafikobjekt während der Animation wieder vom Bildschirm verschwinden zu lassen. Flash referenziert die Elemente der Animation über ihre eindeutige ID und eine Ebenenangabe. Mehrere Instanzen eines Grafikobjekts dürfen deshalb nicht auf derselben Ebene existieren.

4.5 Ausblick

Zusammenfassend lässt sich feststellen, dass SWF-Dateien sowohl mit der grafischen Entwicklungsumgebung Flash Professional 8 von Adobe als auch mit dem Flash File Format SDK erzeugt werden können. Die grafische Entwicklungsumgebung kostet einige hundert Euro, aber die Erstellung von Animation ist wesentlich einfacher als mit der kostenfreien SWF-SDK, welches nicht mehr verfügbar ist. Ausßerdem setzt der Umgang mit dem Flash Professional 8 keinerlei Programmierkenntnisse voraus, da der Benutzer dort wie in einem Zeichenprogramm arbeiten kann, er kann seine Ergebnisse mitverfolgen und sofort testen. Das Software Development Kit ermöglicht das Erzeugen von Flashanimationen über eine Programmierschnittstelle, die abgesehen von Programmierfehlern auch Fehler im Layout nicht sofort kenntlich macht. Das Testen einer Animation erfordert stets das Kompilieren der Quelle und das Öffnen der erzeugten Animation mit einem Flash Player.

Dennoch eröffnet das SWF-SDK neue interessante Möglichkeiten: Die automatische Generierung von Flashanimationen mit ähnlichem Inhalt, die z.B. Daten desselben Formats visualisieren. Und genau das ist die Grundlage für die vorliegende Diplomarbeit. Aus den Logdateien des Robocup Soccer Server sollen Animationen erstellt werden, die jeder leicht anschauen kann ohne ein zusätzliches Programm zu installieren, welches nur diese Logdateien abspielt.

Ein Nachteil bei der Entwicklung des Tools bleibt: Für jede minimale Layoutänderung der erzeugten Flashanimation müssen die Programmquellen des Werkzeuges angepasst werden und neu übersetzt werden. Eine zusätzliche Möglichkeit zur Konfiguration ist wünschenswert, die einige Daten für die Animation enthält und diese beim Aufruf des Programmes übergibt. Eine mögliche Idee dazu ist eine Konfigurationsdatei, die dann auch bei diesem Projekt Verwendung findet. Das folgende Kapitel beschreibt nun das Programm und geht dabei auch kurz auf die Konfiguration ein.

Teil III
Tool

In diesem Teil der Arbeit soll das programmierte Tool vorgestellt werden. Als erstes wird ein kurzer Überblick über den Aufbau des Tools gegeben, danach werden die einzelnen Programmteile genauer betrachtet. Dabei werden auch weitere Klassen des SWF-SDK vorgestellt.

5 Überblick

Die Basis für das Tool sind die Logdateien (siehe Kapitel 2.4) des RoboCup Soccer Server, sie haben immer die Endung 'rcg'. Diese Dateien werden mit Hilfe des `LogFileReader` ausgelesen und danach im `FlashMaker` mit Hilfe des Flash API in eine Animation umgewandelt.

Die Klasse `Robocup2Flash` ist die Hauptklasse des Tools, dort findet die Verarbeitung der Optionen beim Programmaufruf statt, sie ruft dann zur Erstellung der Animation den `FlashMaker` auf.

Die einzelnen Objektklassen dienen nur als Hilfsklassen, die die Methoden für die Erstellung der einzelnen Animationsbestandteile zur Verfügung stellen. Die Objektklasssen lassen sich in zwei Typen aufteilen, einige sind objektorientiert modelliert, weil diese Objekte immer wieder neu gesetzt werden. Der andere Typ ist prozedural programmiert, da diese Methoden nur einmal aufgerufen werden bzw. die Objekte in der Animation nur einmal gezeichnet werden.

Eine besondere Rolle kommt dem `ConfigReader` zu, er liest bei Bedarf verschiedene Vorgaben aus einer Konfigurationsdatei aus.

Zum besseren Verständnis soll Abbildung 5.1 den Aufbau des Tools deutlich machen. Das Werkzeug wurde modular konzipiert, so dass es leicht zu ändern und

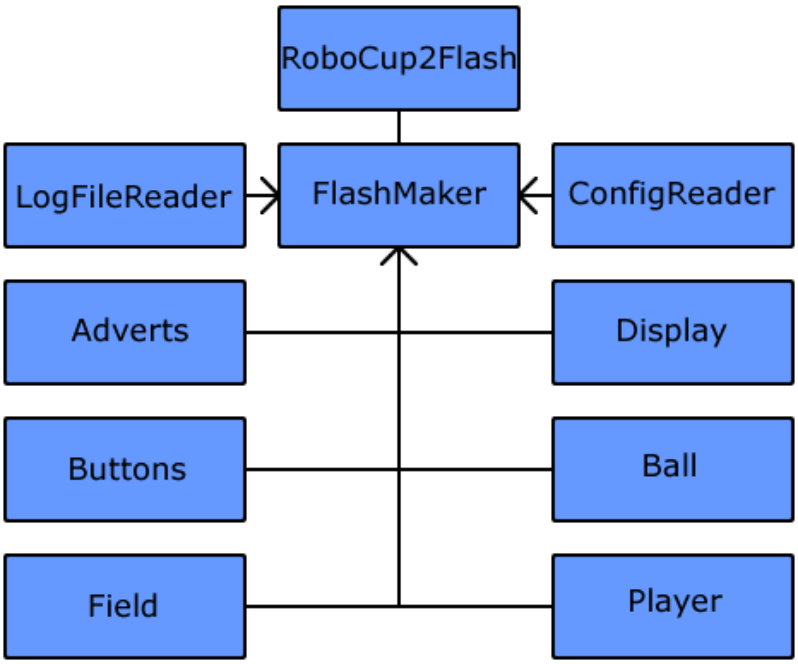

Abbildung 5.1: Übersicht der Klassen des Tools

erweiterbar ist, es können zum Beispiel die einzelnen Objekte leicht ausgetauscht oder verändert werden.

In den folgenden Kapiteln sollen nun die einzelnen Klassen und ihre Aufgaben genau vorgestellt werden.

6 RoboCup2Flash

Die Klasse `RoboCup2Flash` dient ausschließlich der Verarbeitung der Optionen beim Programmaufruf, sie beinhaltet die `main`-Methode, die beim Programmstart aufgerufen wird. Bei diesem Aufruf muss auf jeden Fall der Name der zu verarbeitenden Logdatei übergeben werden, optional hingegen ist der Name der Animation, die erstellt werden soll. Wird für die Animation kein Name übergeben, wird an den Namen der Logdatei einfach die Endung `.swf` angehängt.

Als einzige weitere Option steht `-c <configFile>` zur Verfügung, der man eine beliebige Konfigurationsdatei (siehe Kapitel 8) übergeben kann. Wird keine Konfigurationsdatei übergeben, wird die Datei `config.default` aus dem Verzeichnis des Tools benutzt.

In der Klasse `RoboCup2Flash` wird zum Schluß dann die Methode `makeAnimation` des `FlashMaker` aufgerufen, die die Namen der Logdatei, der Flashdatei und der Konfigurationsdatei übergeben bekommt.

7 LogFileReader

Der `LogFileReader` stellt die Methoden zum Auslesen der Logdatei (Kapitel 2.4) zur Verfügung. Die Daten werden in einzelnen Variablen zwischengespeichert, die dann über `get`-Methoden abgerufen werden können. Für das Auslesen und Steuern der Logdatei stehen folgende Methoden zur Verfügung:

- `OpenLogFile(char* filename)` öffnet die übergebene Datei
- `CloseLogFile()` schliesst die akltuelle Logdatei
- `Reset()` springt an den Anfang der Datei
- `ReadLogFileChunk()` liest einen Block aus der Logdatei aus

Die wichtigste Methode ist `ReadLogFileChunk()`, sie liest einen Block der Daten aus der Logdatei. Dieser Block wird in verschiedenen Datenfelder gespeichert, die in den folgenden C++-`structs` zusammengefasst werden:

```
typedef struct {
  float x;
  float y;
  float angle;
  bool visible;
  float stamina;
  float headangle;
```

```
  bool goalie;
} gameobject

typedef struct {
  gameobject player[MAX_PLAYER*2];
  gameobject ball;
} gameobjects
```

Diese Strukturen können dann über die folgenden `get`-Methoden ausgelesen werden.

- `isGoalie(int i)` überprüft, ob der durch `i` übergebene Spieler ein Torwart ist.
- `getX(int i)` gibt die x-Koordinate des Objektes `i` (Spieler oder Ball) wieder.
- `getY(int i)` gibt die y-Koordinate des Objektes `i` (Spieler oder Ball) wieder.
- `getAngle(int i)` gibt die Richtung des Körpers des Spielers `i` in Bezug auf das Spielfeld an (Winkelmaß).
- `getStamina(int i)` gibt die aktuelle Ausdauer des Spielers `i` zurück.
- `getHeadangle(int i)` gibt an, in welche Richtung der Spieler `i` schaut (ist abhängig vom Winkel des Körpers).
- `isVisible(int i)` gibt an, ob das aktuelle Objekt `i` sichtbar ist.

In der Variablen `i` ist die jeweilige ID des aktuellen Spielers oder des Balls abgespeichert. Die Daten für die folgenden Methoden werden in weiteren einfachen Variablen abgespeichert.

- `getPlaymode()` gibt den derzeitigen Spielmodus zurück.
- `getTime()` gibt den aktuellen Zeitstempel des Spieles wieder (1 bis 12000), entspricht einer Zehntelsekunde.
- `getTeamName(int i)` gibt den Namen des Teams `i` zurück.
- `getTeamScore(int i)` gibt den aktuellen Spielstand des Teams `i` zurück.

Bei den Teaminformationen ist `i` die ID eines der beiden Teams.

Beim Lesen eines Blockes wird zuerst ein Parameter ausgelesen, der angibt von welchem Typ der nächste Block ist. Folgende Möglichkeiten stehen zur Verfügung:

```
NO_INFO      0
SHOW_MODE    1
MSG_MODE     2
DRAW_MODE    3
BLANK_MODE   4
PM_MODE      5
TEAM_MODE    6
PT_MODE      7
PARAM_MODE   8
PPARAM_MODE  9
```

Für die Visualisierung sind nur drei der Typen wichtig. Zum einen der `PM_MODE`, der den aktuellen Spielstatus enthält, der aber nur in die Logdatei geschrieben wird, wenn sich der Modus ändert. Eine Liste der Spielmodi befindet sich im Anhang A.

Der zweite wichtige Modus ist der `TEAM_MODE`, dort werden alle wichtigen Teamdaten gespeichert, die Namen und der aktuelle Spielstand. Dieser Modus wird nur einmal am Anfang der Logdatei geschrieben und dann immer nur, wenn ein Tor fällt.

Der `SHOW_MODE` ist der wichtigste Modus. Er steht für jeden Zeitstempel des Spieles zur Verfügung und beinhaltet die Daten der einzelnen Spielobjekte, des Balls und der Spieler. Es werden die x- und y-Positionen der einzelnen Spieler und des Balls gespeichert. Für den Spieler werden zusätzlich noch die Richtungen (Winkel), in die die Spieler blicken gespeichert, außerdem wird die Ausdauer des Spielers gespeichert und ob der Spieler der Torwart ist. Es könnten natürlich noch viele weitere Daten vom `LogFileReader` ausgelesen werden, diese werden aber für eine Visualisierung nicht benötigt. Diese Daten müssen aber übersprungen werden, da es sonst beim Auslesen der Datei zu Problemen kommt. Aus diesem Grund müssen auch alle anderen Typen trotzdem vom `LogFileReader` ausgelesen werden.

Die Informationen des `SHOW_MODE` sind in einer `showinfo_t2`-Struktur gespeichert.

```
typedef struct {
  char pmode;
  team_t team[2];
  ball_t ball;
  player_t pos[MAX_PLAYER * 2];
  short time;
} showinfo_t2
```

Er enthält die Daten zum aktuellen Spielstand, zu den Teams, die des Balles und der Spieler sowie den aktuellen Zeitstempel. Für die Animationen wird aber nur

ein Teil der Daten benötigt. Aus diesem Grund wurde eine verkürzte Struktur entwickelt.

```
typedef struct {
  ball_t ball;
  player_t pos[MAX_PLAYER * 2];
  short time;
} short_showinfo_t2
```

Die Datenstruktur `short_showinfo_t2` stellt alle benötigten Informationen für die Animation zur Verfügung, die Daten der Spieler und des Balles, sowie den aktuellen Zeitstempel. Diese Struktur wird eingelesen und dann in der oben angegebene `gameobjects`-Struktur abgespeichert.

Die Methode `ReadLogFileChunk()` liest die Daten aus und speichert sie in die angegebenen Strukturen, sie selber gibt aber nur den Modus des eingelesenen Blockes zurück. Die eigentliche Verarbeitung der Informationen, also was bei welchem Chunktypen passiert, geschieht in dem aufrufenden Programm, hier der `FlashMaker`.

Der `LogFileReader` speichert also nur die Daten in den zugehörigen Datenfeldern, der `FlashMaker` ruft dann die passenden `get`-Methoden auf.

8 ConfigReader

Der `ConfigReader` ist eine Klasse zum Einlesen einer Konfigurationsdatei. Sie wird von der AG Neuroinformatik der Universität Osnabrück [Web NI UOS] bereitgestellt, die sie auch bei ihren eigenen Anwendungen benutzt. Das Format der Konfigurationsdatei ist somit zwar vorgegeben, dafür aber allgemein bekannt und auch leichter für Menschen lesbar. Da die Daten nur zeilenweise in der Datei stehen ist sie besser und schneller zu verändern und zu verarbeiten als das bei einer XML-Konfigurationsdatei der Fall ist.

Die Daten stehen zeilenweise in der Konfigurationsdatei, zur besseren Übersicht kann man verschiedene Blöcke angeben. Der Block `[SWF]` beinhaltet alle Informationen zur Animation selbst, zur Größe, Hintergrundfarbe etc., man kann außerdem angeben, ob die Buttons sichtbar sein sollen, ob man nach Toren eine Zeitlupe sehen möchte oder ob man eine einfache Ansicht der Spieler haben möchte, den sogenannten `classic_view` (siehe Abbildung 8.1).

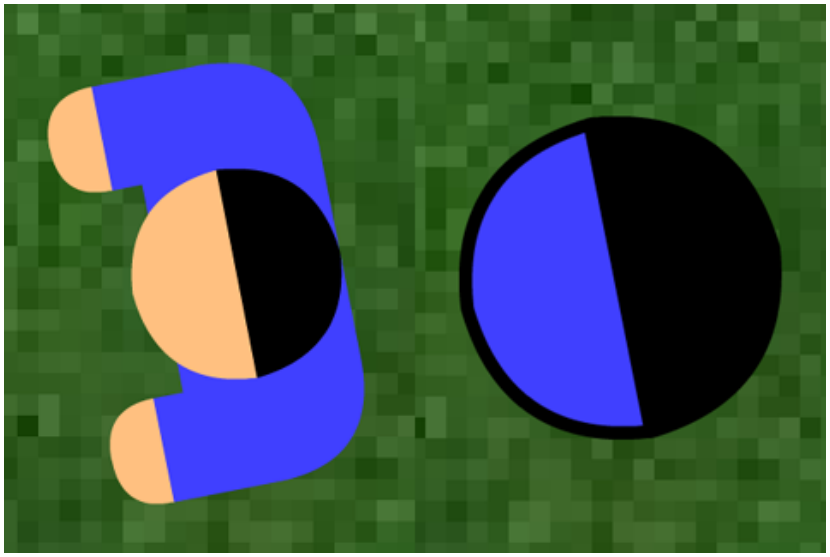

Abbildung 8.1: Links der normale Modus, rechts die klassische Ansicht

Der Inhalt der Datei `config.default` sieht wie folgt aus:

```
[SWF]
flashWidth = 800
flashHeight = 600
fieldMeasureX = 52.5
fieldMeasureY = 34
slowmotion = true
```

```
# time in frames (10 fps)
slowmotion_time = 50
bgcolor = 0x00 0x00 0x00
grass = grass.jpg
buttons = true
classic_view = false
```

Für jedes Team lassen sich die Farben der Trikots angeben, einmal für die Feldspieler (`teamColor`) und einmal für den Torwart (`goalieColor`).

```
[TEAM0]
teamColor = 0xFF 0x00 0x00
goalieColor = 0xFF 0xFF 0x40

[TEAM1]
teamColor = 0x40 0x40 0xFF
goalieColor = 0x80 0xC0 0xFF
```

Im Block `[SCENES]` wird angegeben, welche Szenen eines Spieles man sehen möchte. Zu Beginn einer solchen Szene kommt eine Einblendung mit Informationen zur Szene (Abbildung 8.2). Werden keine Szenen angegeben, wird in der Animation das komplette Spiel abgespeichert.

Abbildung 8.2: Beispiel einer Einblendung

```
[SCENES]
num = 0
#example for scenes
scene0_start = 2950
scene0_end = 3050
scene0_text = Half time test
scene1_start = 5950
scene1_end = 6000
scene1_text = End
```

Der Block `[ADVERTS]` kann Angaben über die Bilder für die Bandenwerbung beinhalten. Ist die Anzahl der Bilder größer als vier, bekommt man den Effekt, dass die Banden zwischendurch wechseln. Sind keine Bilder angegeben, werden in der Animation auch keine Banden gezeichnet.

```
[ADVERTS]
num = 2
texture0 = advert00.jpg
texture1 = advert01.jpg
```

Der `ConfigReader` stellt Methoden zur Verfügung, die das Auslesen von Strings, Zahlen in verschiedenen Formaten (auch Vektoren) und Boole'schen Variablen erlaubt. Die benötigten Daten werden erst bei Bedarf ausgelesen, sollte ein Feld nicht in der Konfigurationsdatei gesetzt sein, wird in der Applikation automatisch ein Defaultwert gesetzt.

Über die Methode `append_from_file(_configFile)` wird die Konfigurationsdatei eingelesen. Um die Werte der einzelnen Zeilen auszulesen, stehen eine Reihe von `get`-Methoden zur Verfügung, deren Aufbau immer gleich ist. Die `get`-Methoden bekommen immer zwei Argumente übergeben, als erstes den Schlüssel mit dem Namen der auszulesenden Variablen und als zweites die Variable in die der Wert des eingelesenen Schlüssels abgespeichert wird. Zum besseren Verständnis ein kurzer Auszug aus dem Programmtext des `FlashMaker`, `cr` ist die Instanz des `ConfigReader`.

```
cr.get("SWF::flashWidth", flashWidth);
cr.get("SWF::flashHeight", flashHeight);
cr.get("SWF::fieldMeasureX", fieldMeasureX);
cr.get("SWF::fieldMeasureY", fieldMeasureY);
```

Es werden in diesem Fall vier Zahlen eingelesen, die die Größe der Animation und des Spielfeldes angeben. Beim Schlüssel wird zuerst immer der Block angegeben, dann der Name der Variablen die eingelesen wird, getrennt durch zwei Doppelpunkte.

Bevor der Aufbau des `FlashMaker` und die Erstellung der Animation vorgestellt werden, soll nun zuerst auf die einzelnen Objekte der Animation eingegangen werden.

9 Objektklassen

In den Objektklassen werden die einzelnen Teile der Animation gezeichnet. Man kann diese in zwei Arten unterteilen, eine die nur einmal in der Animation gezeichnet wird, und die Zweite, die die sich fortlaufend ändernden Objekte zeichnet.

In der ersten Gruppe befindet sich beispielsweise das Spielfeld, welches zu Beginn der Animation nur einmal gezeichnet werden muss. Es ist nicht als Objekt modelliert um einen Vorteil der prozeduralen Programmierung zu nutzen, nämlich die Geschwindigkeit. Die Klassen der zweiten Gruppe sind für eine bessere Übersicht und leichter zu wartenden Code weitestgehend objektorientiert modelliert.

Im weiteren sollen nun die Objektklassen genauer vorgestellt werden.

9.1 Field

Diese Klasse stellt die Methode `DrawField()` zur Verfügung. Wie der Name schon sagt, wird dort das Spielfeld gezeichnet und an die Animation angehängt. Das Spielfeld ist statisch in der Animation, das heißt es verändert sich nicht während der Animation.

Über die Konfigurationsdatei kann man eine Textur für den Rasen (Abbildung 9.1) angeben. Falls für den Rasen keine Textur angegeben wurde oder die angegebene Datei nicht gefunden wurde, wird eine einfarbige grüne Füllung (Abbildung 9.2) als Rasen verwendet. Die Textur oder Farbe wird als Hintergrund des Flashobjektes gewählt.

Abbildung 9.1: Textur für den Rasen

Abbildung 9.2: Einfarbiger Hintergrund

Mit Hilfe der Klasse `FDTDefineBitsJPEG2` läßt sich eine Grafik in die Animation einhängen, über die Klasse `FFillStyleBitmap` ist sie dann als Füllung bzw. Hintergrund für das Spielfeld festgelegt. Als Linie des Spielfeldes, und gleichzeitig als Umrandung des Shapes wird eine weiße dicke Linie angegeben.

Mit den übergebenen Koordinaten für die Größe des Spielfeldes lassen sich mit Hilfe der offiziellen Maße eines Spielfeldes alle weiteren Linien bzw. Kreise zeichnen.

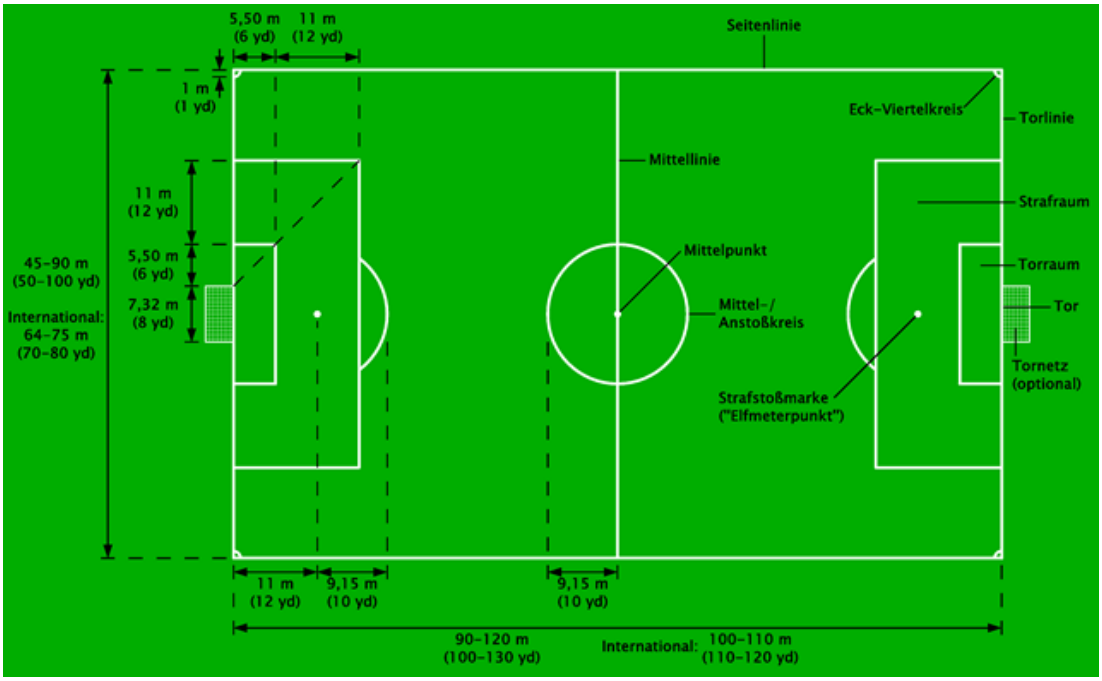

Abbildung 9.3: Fußballfeld mit Maßen und Beschreibung

Alle Maße sind in der Klasse als Konstanten vorgegeben, können also auch leicht verändert werden. Länge und Breite des Spielfeldes richten sich nach der Konfigurationsdatei, also der Größe der Animation und die Seitenverhältnisse des Spielfeldes.

9.2 Buttons

Diese Klasse zeichnet die vier Steuerelemente, die die Animation steuern können. Man kann sie über die Konfigurationsdatei auch ausblenden um eine selbstlaufende Animation zu erhalten, das ist insbesondere für eine Umwandlung in eine Videodatei von Vorteil. Da die Buttons nur zu Beginn der Animation einmal in diese eingefügt werden, lohnt sich auch hier die objektorientierte Modellierung nicht. Anhand des Play-Button soll nun die Vorgehensweise erklärt werden, wie man einfache *Actions* in eine Animation einfügt:

```
FDTDefineShape *shape = new FDTDefineShape(new FRect(-10,-10,
                                                     510,510));
const FColor buttonFill(255,255,0);
const FColor buttonLine(255,255,128);

fillID = shape->AddSolidFillStyle(new FColor(buttonFill));
lineID = shape->AddLineStyle(10,new FColor(buttonLine));
shape->FinishStyleArrays();

shape->AddShapeRec(new FShapeRecChange(false, true, true, false,
                                       true, 0, 0, 0, fillID,
                                       lineID, 0, 0));
shape->AddShapeRec(new FShapeRecEdgeStraight(500,250));
shape->AddShapeRec(new FShapeRecEdgeStraight(-500,250));
shape->AddShapeRec(new FShapeRecEdgeStraight(0,-500));
shape->AddShapeRec(new FShapeRecEnd());
flashObjects.AddFObj(shape);
```

Im ersten Teil des Beispiels wird das *Shape* erstellt, also das Aussehen des Play-Button.

Ab hier beginnt die eigentliche Erstellung des Steuerelementes. In diesem Teil wird dann der eigentliche Button, ein `FDTDefineButton2` erstellt. Der zugefügte `FButtonRecord2` speichert den aktuellen Status des Button und auf welche Events er reagieren soll und wie.

```
FDTDefineButton2 *playButton = new FDTDefineButton2(0);

playButton->AddButtonRecord(new FButtonRecord2(true, true, true,
                                               true, shape->ID(),
                                               1, 0,0));

FActionCondition *ac = new FActionCondition();
ac->AddCondition(OverDownToOverUp);
```

```
ac->AddActionRecord(new FActionPlay());
playButton->AddActionCondition(ac);
```

Mit der Klasse `FActionCondition` wird dann die eigentliche *Action* erstellt, `OverDownToOverUp` bedeutet, dass beim Loslassen der Maus innerhalb des zuvor definierten *Shapes* die *Action* ausgelöst wird. Mit `FActionPlay` wird dann der Typ der Action übergeben. Danach muss der Button noch der Animation hinzugefügt und mit Hilfe von `FCTPlaceObject2` in der Szene platziert werden.

```
flashObjects.AddFObj(playButton);
FMatrix *matrix = new FMatrix(0, 0, 0, 0, 0, 0, 800, 40);
flashObjects.AddFObj(new FCTPlaceObject2(false, true, true, false,
                                         1, playButton->ID(),
                                         matrix, 0, 0, 0, 0));
```

Für die weiteren Buttons wurde genau dasselbe gemacht. Natürlich wurde für jeden Knopf ein eigenes *Shape* erstellt und auch eine eigene *Action* zugewiesen. Für den Stop-Button gibt es die Klasse `FActionStop`, um an den Anfang der Animation zu springen gibt es die Klasse `FActionGotoFrame`, die eine Zahl übergeben bekommt, zu welchem Frame gesprungen werden soll. Mit `FActionNextFrame` läßt sich die Animation um ein Bild weiter vorspulen. Das Pendant `FActionPreviousFrame`, um ein Bild zurückzugehen sieht das SDK zwar vor, aber der Adobe Flash Player setzt es nicht richtig um, so dass das aktuelle Bild nicht richtig dargestellt wird. Aus diesem Grund wird das passende Steuerelement nicht in die Animation eingefügt.

Alle weiteren Actions des vorliegenden SDK sind für die Animation nicht nötig.

Aktion	**Beschreibung**
ActionGoToLabel	springt zu einem Frame mit dem vorher vergebenen Label
ActionWaitForFrame	warten auf einen bestimmten Frame
ActionGetURL	gibt URL zurück (bei Link-Buttons)
ActionStopSounds	stoppt alle wiedergegebenen Sounds
ActionToggleQuality	Wechsel zwischen hoher und niedriger Qualität auf dem Display
ActionSetTarget	ändert den Kontext für nachfolgende Actions auf das angegebene Objekt

Tabelle 8: Weitere Actions von Flash 3

In den folgenden Versionen von Flash wurden immer weitere *Actions* hinzugefügt, die vor allem der Interaktion mit dem Benutzer dienen, um Formulare auszuwerten oder um mathematische Operationen auszuführen. Bei den erstellten Animationen ist es aber nicht nötig, weitere Interaktionsmöglichkeiten hinzuzufügen.

9.3 Display

Das Display zeigt wie bei einem echten Fußballspiel einige wichtige Informationen, die Teamnamen, die Tore und natürlich die aktuelle Spielzeit. Außerdem wird auf dem Display immer der aktuelle *Playmode* angezeigt.

Da das Display ständig verändert wird, ist es objektorientiert modelliert. Außer dem Konstruktor, der einmalig den Hintergrund zeichnet, gibt es noch folgende Methoden:

- `setTime(int _t)` zeichnet die neue Zeitangabe auf das Display. Das übergebene `_t` ist dabei der aktuelle Zeitstempel aus der Logdatei, der noch in ein vernünftig lesbares Format umgewandelt werden muss.
- `setScore(int _score, int _side)` setzt das neue Ergebnis eines bestimmten Teams auf eine Seite des Display, die Seitenangabe `_side` wird gemacht um einen Seitenwechsel zu realisieren, sie kommt beim folgenden `setTeam` auch vor.
- `setTeam(char* _name, int _side)` setzt den Teamnamen an die bestimmte Seite, auch hier wie erwähnt um einen Seitenwechsel zur Halbzeit deutlich zu machen.
- `setPlaymode(int _id)` zeichnet den aktuellen Spielzustand auf das Display, eine Liste mit allen möglichen Zuständen befindet sich in Anhang A.

Alle Methoden arbeiten mit der Klasse `FDTDefineEditText`, die einen einfachen Text realisiert. Der Konstruktor bekommt als wichtigste Parameter, neben einigen anderen, die Größe, Farbe und Ausrichtung der Schrift, sowie den eigentlichen Text übergeben. über das `PlaceObject`-Objekt wird das jeweilige Textobjekt dann in der Szene platziert. Hier werden allerdings keine Positionen beim platzieren angegeben, da die Textfelder schon vorher über das `FRect`-Objekt an die richtige Stelle gesetzt wurden. Bevor eines der Textobjekte gezeichnet wird, wird natürlich ein vorher schon gezeichnetes Objekt des selben Typs durch `RemoveObject` gelöscht.

Eine Besonderheit existiert in der Klasse `Display`. Um die Flashanimation nicht unnötig zu vergrössern, werden für die Zeitangabe alle Zeichen einmal erstellt, das heisst die Ziffern '0' bis '9', '.' und ':' werden als eigene Objekte gespeichert und dann jeweils bei der Methode `setTime` passend ausgesucht und dann an der richtigen Stelle in der Szene platziert. Würde man für jede Zehntelsekunde bzw. jeden Frame ein eigenes Objekt über `FDTDefineEditText` erstellen, wäre der Speicherbedarf der Animation sehr viel grösser.

Bei den anderen Methoden ist diese Art der Platzersparnis nicht nötig, die Teamnamen ändern sich ja zum Beispiel nur einmal zur Halbzeit und die Anzahl der Tore auch nur, nachdem eines gefallen ist. Auch der Spielzustand ändert sich relativ selten und es sind auch nicht immer alle Zustände in einer Partie zu sehen, so daß es nicht nötig ist, für jeden Playmode ein eigenes Textobjekt anzulegen.

9.4 Adverts

Die Bandenwerbung ist ein optionaler Teil der Animation, er dient in erster Linie dazu, den Sponsoren des RoboCup auch in der Simulation einen Platz zu geben, sich zu präsentieren.

Wie Bilder in eine Animation eingebunden werden, haben wir schon im Abschnitt 9.1 kennengelernt. Das besondere an der Bandenwerbung ist, dass sie sich wie in richtigen Fußballstadien ändern kann. Da nur beschränkt Platz für Werbung in der Animation im Spiel ist, stehen nur vier Banden zur Verfügung. Sollte es also mehr als vier Banden geben, werden die Banden wie in einem richtigen Fußballstadion alle zehn Sekunden ausgetauscht.

Im Konstruktor der Klasse wird eine Bande erstellt, dieses erstellte Objekt wird dann einer Liste zugefügt, die die Banden speichert. In der Konfigurationsdatei kann man die Anzahl und die Bilddateien der einzelnen Banden angeben.

In der untenstehenden Methode `draw` wird dann eine Bande gezeichnet. Als erstes wird aber eine alte Bande an der bestimmten Stelle `_place` aus der *DisplayList* gelöscht.

```
void Adverts::draw(int _place, int _x, int _y) {
  FCTRemoveObject2 *remove = new FCTRemoveObject2(2300 + _place);
  flashObjects.AddFObj(remove);
```

Dann wird die Bande an die über eine Matrix angegebene Position(`_x`,`_y`) gesetzt. Mit `2300 + _place` bleibt der Layer eindeutig besetzt und wird von keiner anderen Bande beziehungsweise irgendeinem anderen Objekt der Animation besetzt. So kann es zu keinen Konflikten kommen.

```
  FMatrix *matrix = new FMatrix(false, // U32    _hasScale
                                0,     // SFIXED _scaleX
                                0,     // SFIXED _scaleY
                                false, // U32    _hasRotate
                                0,     // SFIXED _rotateSkew0
                                0,     // SFIXED _rotateSkew1
                                _x,    // SCOORD _translateX
                                _y);   // SCOORD _translateY

  FCTPlaceObject2 *placeObj =
     new FCTPlaceObject2(false,      // U16      _hasClipDepth
                         false,      // U16      _hasRatio
                         true,       // U16      _hasCharID
                         false,      // U16      _hasMove
                         2300 + _place, // U16   _depth
```

```
                        id,         // U16      _characterID
                        matrix,     // FMatrix* _matrix
                        0,          // FCXForm* _colorTransform
                        0,          // U16      _ratio
                        0,          // FString* _name
                        0);         // U16      _clipDepth

    flashObjects.AddFObj(placeObj); // setze neue Bande
  }
```

Um den Aufruf der Methode `draw` deutlicher zu machen folgt ein kleiner Auszug aus dem `FlashMaker`:

```
  if (timer % 100 == 0) { // all 100 frames
    // draw adverts
    for (int i = 0; i < 4; i++) {
      int whichAdd = (adCounter + i ) % adNumber;
      adverts[whichAdd].draw(i, adCoordX[i], adCoordY);
    }
    adCounter++; // increment for changimg adverts
  }
```

Alle 100 Frames (das entspricht zehn Sekunden) werden die `adverts`-Objekte gezeichnet. Wie in Abbildung 9.4 zu sehen, ist die y-Koordinate immer dieselbe, die x-Koordinaten wurden in einem Array gespeichert. Um die wechselnden Banden zu realisieren wird ein `adCounter` immer um eins erhöht und damit man nicht aus der `adverts`-Liste herausspringt wird noch einmal Modulo der Anzahl der Banden `adNumber` gerechnet.

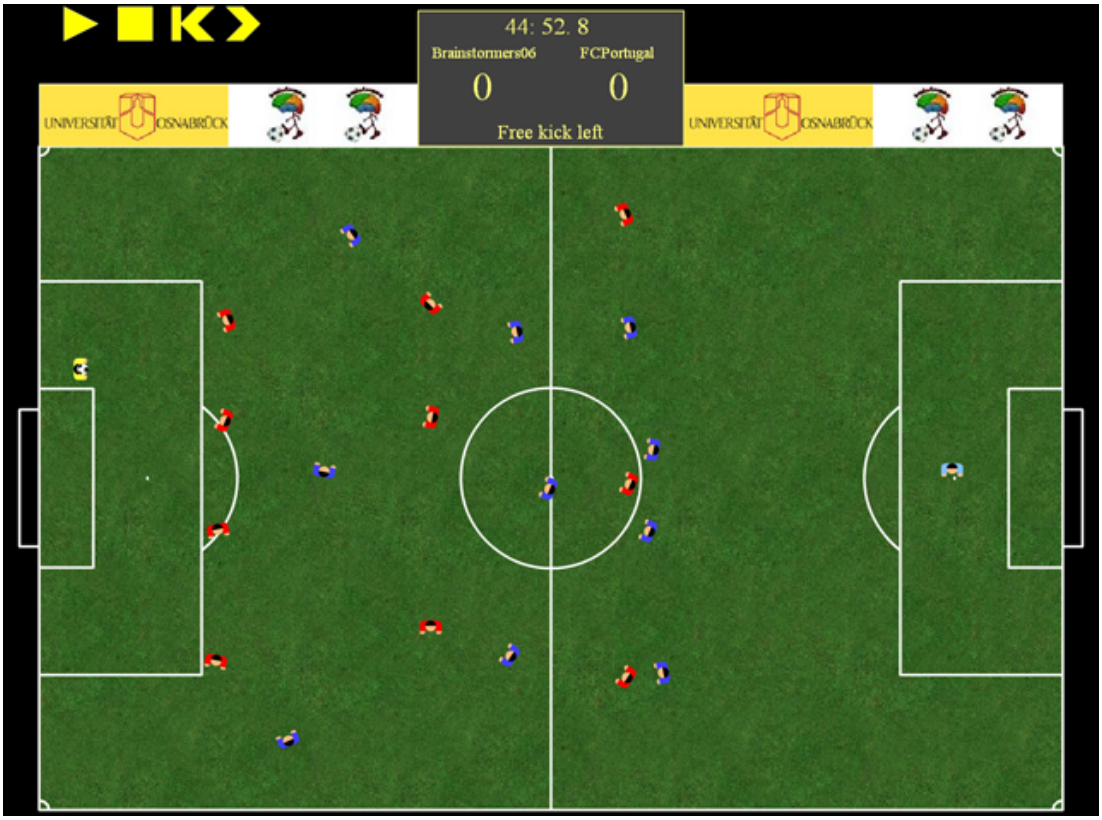

Abbildung 9.4: Ein Frame aus der Animation

9.5 Ball

Der Ball, das wichtigste Objekt eines Fußballspiels, wird hier über eine Textur realisiert oder besser gesagt über mehrere Texturen.

Abbildung 9.5: Die Texturen des Balles

Bei der Erstellung der Objekte muss darauf geachtet werden, dass nicht einfach ein Rechteck genommen wird in das die Textur als Hintergrund eingefügt wird. Man muss einen Kreis erstellen der als Füllung dann die Textur bekommt. Kurven werden über das `FShapeRecEdgeCurved`-Objekt erstellt. Es handelt sich dabei um quadratische Bézierkurven mit einem Stützpunkt. Als Startpunkt dient dabei der aktuelle Punkt, dem `FShapeRecEdgeCurved`-Objekt werden dann noch zwei Punktpaare übergeben, zum einen der Stützpunkt der Kurve und der Endpunkt der Kurve.

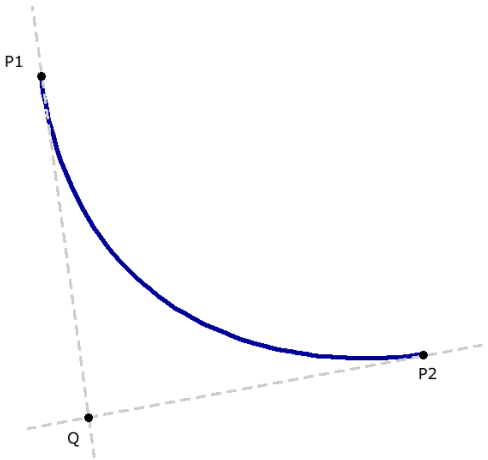

Abbildung 9.6: Darstellung einer Bézierkurve

Eine quadratische Bézierkurve ist der Pfad, der durch die Funktion C(t) für die Punkte P_1, Q und P_2 verfolgt wird. Der Berechnung der Kurve liegt folgende Gleichung zugrunde.

$$C(t) = (1-t)^2 P_1 + 2t(1-t)Q + t^2 P_2 \qquad t \in [0;1]$$

Mit dem SDK kann man leider nicht direkt Kreise zeichnen, aber durch die Aneinanderreihung von vier Bézierkurven läßt sich auch leicht ein Kreis erstellen, die Stützpunkte müssen dabei auch nicht erst berechnet werden, da die jeweiligen Koordinaten der Stützpunkte in den Anfangs- bzw. Endpunkten der Kurve vorkommen.

In der Animation werden mehrere Bälle verwendet, um eine Rotation des Balles zu simulieren. Bei jedem Frame, oder genauer gesagt, nach dem der Ball seine Position verändert hat, wird eine neue Textur für den Ball verwendet. Die Idee ist dieselbe wie bei der Bandenwerbung:

```
if(!((oldX == newX) && (oldY == newY))) {
  ball[ballCounter%4].set(x, y, angle);
  ballCounter++;
}
```

Immer wenn der Ball eine neue Position einnimmt, wird der zuvor erstellte Ball an die jeweilige Position gesetzt, danach wird die Variable `ballCounter` um eins erhöht, damit beim weiteren Setzen des Balles der nächste Ball verwendet wird. Die `set`-Methode des Balles sieht wie alle vorhergehenden aus. Zuerst wird der alte Ball oder genauer gesagt, der Layer auf dem der Ball gezeichnet wurde, gelöscht. Danach wird über `FMatrix` eine Transformationsmatrix angegeben und der neue Ball in der Szene platziert.

```
FMatrix* matrix =
   new FMatrix(true,                          // U32    _hasScale
               (int)(cos(_angle)*Fixed1),  // SFIXED _scaleX
               (int)(cos(_angle)*Fixed1),  // SFIXED _scaleY
               true,                          // U32    _hasRotate
               (int)(sin(_angle)*Fixed1),  // SFIXED _rotateSkew0
               (int)(-sin(_angle)*Fixed1), // SFIXED _rotateSkew1
               _x,                            // SCOORD _translateX
               _y);                           // SCOORD _translateY
```

Das ist die verkürzte Schreibweise einer 3x3-Transformationsmatrix mit folgendem Inhalt:

$$\begin{pmatrix} \_scaleX & \_rotateSkew1 & \_translateX \\ \_rotateSkew0 & \_scaleY & \_translateY \\ 0 & 0 & 1 \end{pmatrix}$$

Die Konstante `Fixed1` ist eine von der Flashspezifikation vorgebene Größe um Winkel in ganzzahlige Werte umzuwandeln. Falls der Wert für die Skalierung `_hasScale` und der Wert für die Rotation `_hasRotate` auf `false` gesetzt ist,

wird automatisch ein Wert 1 für die Skalierung und der Wert 0 für die Rotation angenommen. Diese Werte werden allerdings gar nicht in der Matrix gespeichert, dadurch wird wiederum Platz in der Datei eingespart und diese dadurch noch kleiner.

Ist in der Konfigurationsdatei die `classic_view` ausgewählt, wird an Stelle der Texturen eine weiße Hintergrundfarbe gewählt, der Ball erhält dann zusätzlich noch einen schwarzen Rand.

9.6 Player

Die Spieler werden über ein komplexeres Shape dargestellt.

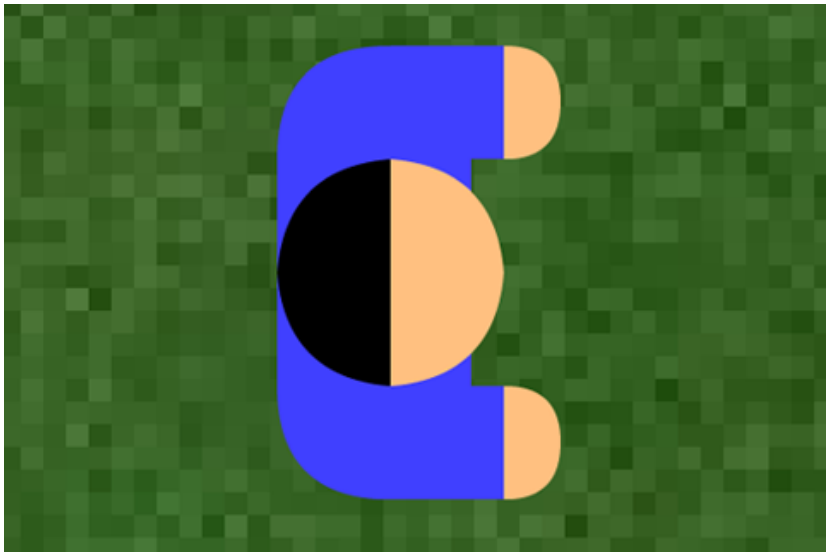

Abbildung 9.7: Ein Spieler in seiner Ausgangsposition (stark vergrößert)

Das Shape wird in mehrere Teile (*ShapeRecords*) aufgeteilt. Zuerst wird der Körper gezeichnet, dann die beiden Hände und der Kopf. Ist die `classic_view` in der Konfigurationsdatei angegeben wird nur ein Kreis gezeichnet mit der Farbe des Teams als Füllung (Abbildung 8.1).

Das ganze Shape wird über acht Punkte eines Kreises realisiert (Abbildung 9.8).

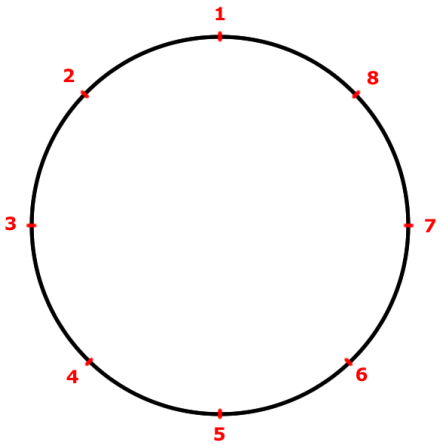

Abbildung 9.8: Punkte mit deren Hilfe gezeichnet wird

Da in einer Flashanimation immer relativ zum letzten Punkt gezeichnet wird, kann man so die Shapes leicht erstellen. Man muss beim Erstellen der Shapes nur auf die Reihenfolge achten und in welche Richtung man zeichnet. Beim Zeichnen einer Linie bzw. Kurve kann man nämlich zwei Füllungen angeben, einmal für die linke Seite der Linie und einmal für die rechte Seite.

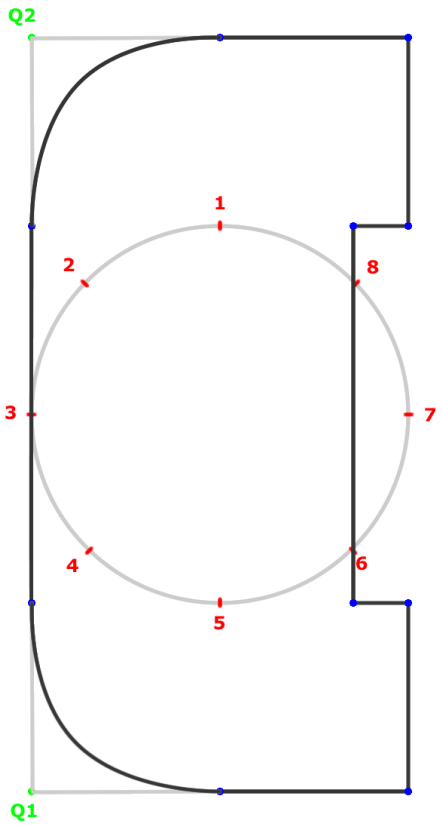

Abbildung 9.9: Der Körper eines Spielers

Um zum Beispiel den Körper zu zeichnen (Abbildung 9.9), nimmt man sich einen Anfangspunkt, im Tool wird die x-Koordinate von Punkt 3 und die y-Koordinate von Punkt 1 genommen, dann eine Linie in positiver y-Richtung (nach unten) gezeichnet, mit Hilfe der y-Koordinaten von Punkt 5 und 1 (Länge ist gleich $y_5 - y_1$). Mit Hilfe des unteren grünen Punktes Q_1 kann eine Bézierkurve gezeichnet werden, und mit der Methode `FShapeRecEdgeCurved` wird so dann erst weiter in y-Richtung und dann in x-Richtung gegangen. Alle weiteren Linien werden mit Hilfe der Punktkoordinaten gezeichnet, bis zum Schluß die letzte Kurve über den Punkt Q_2 gezeichnet wird, dann ist das *ShapeRecord* komplett.

Die Hände und der Kopf des Spielers werden auf die gleiche Weise erstellt. Man sucht sich Anfangskoordinaten und zeichnet von da an die Linien und Kurven der einzelnen Objekte. Die einzelnen *ShapeRecords* werden zu einem Shape zusammengefaßt und der Animation hinzugefügt.

Dadurch, daß man einzelne *ShapeRecords* verwendet, kann man leicht die einzelnen Füllfarben setzen. Würde man versuchen die Konturen des Spielers in ein einzelnes *ShapeRecord* zu zeichnen, käme es zu Problemen, weil man nicht eindeutig eine Füllung angeben kann, da sich einige Linien überlappen.

Auch die `Player`-Klasse hat die Methoden `set` und `remove` zum Setzen und Löschen eines Spielers, allerdings werden diese im Tool nicht verwendet, wie schon in Kapitel 4 erwähnt hat jedes Objekt eine eindeutige ID. Diese ID hat auch jeder Spieler der erstellt wird, der `FlashMaker` speichert diese in einem Array ab und setzt so selber über die ID die einzelnen Spieler in der Animation. Man muss so nur immer einen Feldspieler und Torwart pro Mannschaft erstellen die immer wieder in die Animation gesetzt werden. Im fogenden Kapitel wird nun der FlashMaker genau beschrieben und damit die Erstellung der Animation.

10 FlashMaker

Der FlashMaker besitzt neben der Methode `makeAnimation` noch einige weitere Methoden, die zuerst erläutert werden sollen. Wie im vorherigen Kapitel schon erwähnt hat der `FlashMaker` Methoden zum Setzen und Löschen von Objekten, diese werden aber nur für die Spieler genutzt.

- `set(int _id, int _layer, int _x, int _y)`
  Ein beliebiges Objekt kann durch seine ID auf eine bestimmte Position gesetzt werden. Der Layer kann außerdem angegeben werden, falls dieser vorher schon besetzt ist, wird das dortige Objekt gelöscht.

- `remove(int _layer)`
  Ein beliebiger Layer kann gelöscht werden.

- `rotate(int _id, float _angle)`
  Ein beliebiges Objekt, angegeben durch seine ID, kann um einen bestimmten Winkel `_angle` rotiert werden. Es wird dann das rotierte Objekt in der Animation abgespeichert (aber nicht angezeigt). Die Methode gibt die ID des neu entstandenen Objektes zurück.

Für einige zusätzliche Einblendungen gibt es zwei weitere Methoden, zum einen gibt es die Methode `toggleFlash`, die die Zwischeneinblendungen bei den einzelnen Szenen macht. Sie ist sehr allgemein gehalten, man kann ihr einen beliebigen Text, ein String-Array übergeben mit maximal fünf Einträgen, welche den angezeigten Zeilen entsprechen. Neben dem Text können auch noch die Dauer der Einblendung, sowie die Koordinaten angegeben werden. Zusätzlich kann auch noch ein Grad für die Transparenz angegeben werden um das Spielfeld nicht komplett zu überdecken.

Abbildung 10.1: Einblendung zu Beginn eines Spiels

Zu Beginn des Spiels wird eine Einblendung gemacht, welche Mannschaften in diesem Spiel aufeinandertreffen, und der Seitenwechsel zur Halbzeit wird auch angezeigt (Abbildung 10.1).

Als zweites gibt es noch die Methode `drawR`, die bei einer Zeitlupenwiederholung ein weißes „R“ - Abkürzung von Replay - in die rechte obere Ecke zeichnet. Neben den Koordinaten bekommt die Methode noch eine boolean `_blink` übergeben. Mit ihr kann das Blinken des „R“s gesteuert werden. Beim Aufruf der Methode wird abwechselnd `true` und `false` übergeben, dadurch wird das „R“ entweder gezeichnet oder gelöscht.

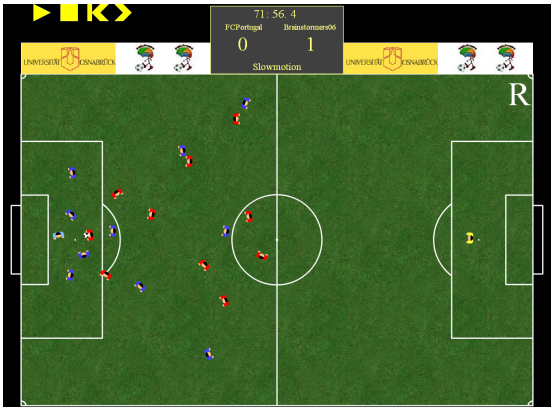

Abbildung 10.2: Das „R“ wird während einer Zeitlupenwiedrholung angezeigt

In der Methode `makeAnimation` wird die Animation zusammengestellt, das Vorgehen dabei soll nun genau betrachtet werden. Die Klasse `FlashMaker` bindet alle anderen Klassen ein, außerdem wird die `FObjCollection flashObjects` erstellt die als globale Variable auch in den anderen Klassen genutzt werden kann, in `flashObjects` werden alle Objekte abgespeichert.

Die Methode `makeAnimation` bekommt den Namen der Logdatei, der zu erstellenden Flashdatei und der Konfigurationsdatei übergeben. Aus der Konfigurationsdatei werden zu Beginn die Maße der Animation ausgelesen, sowie das Verhältnis der Seiten des Spielfeldes und die Hintergrundfarbe der Animation.

Mit `flashObjects.AddFObj(new FCTSetBackgroundColor(color)` wird der Animation eine Hintergrundfarbe hinzugefügt.

Als nächstes werden einige Maße anhand der Animationsgröße berechnet. Es wird berechnet wie groß das Spielfeld ist, wie groß das Display und die Bandenwerbung sind. Das Spielfeld erhält die maximal mögliche Größe, die Größe des Display und der Bandenwerbung werden dementsprechend angepaßt.

Mit den berechneten Größen läßt sich nun das Spielfeld zeichnen, außerdem wird das Display erstellt.

```
DrawField(cr, xMin, yMin, xMax, yMax);

Display display (xMin + 2 * advertWidth,  // int _xMinDisplay
                 MARGIN,                  // int _yMinDisplay
                 xMax - 2 * advertWidth,  // int _xMaxDisplay
                 yMin);                   // int _yMaxDisplay

display.setScore(0, 0); // short _score, int _side
display.setScore(0, 1); // short _score, int _side
```

Zusätzlich wird ein initialer Spielstand auf das Display geschrieben.

Es folgt die Erstellung der einzelnen Banden. Für jede der in der Konfigurationsdatei stehenden Bilddateien wird ein `Adverts`-Objekt erstellt und in einer Liste gespeichert. Beim Zeichnen der Banden ist es wichtig, jeweils die richtige x-Koordinate anzugeben, die y-Koordinate ist immer dieselbe. Die Methode `draw` der Klasse `Adverts` (Kapitel 9.4) zeichnet ein erstes Mal die Banden.

Nach dem die Banden erstellt wurden, werden nun die eigentlichen Spielobjekte erstellt, die verschiedenen Bälle und die einzelnen Spieler.

Die verschiedenen Bälle (Kapitel 9.5) werden in einem Vektor abgespeichert. Bei den Spielern (Kapitel 9.6) wird nur die ID gespeichert. Dazu wird ein mehrdimensionales Array `playerSpriteID[2][2][32]` erstellt. Die erste Stelle ist für die beiden Teams, die zweite unterscheidet zwischen Torwart und Feldspieler. Die dritte Stelle ist für die einzelnen Richtungen in die ein Spieler blicken kann.

Als erstes werden für jede Mannschaft ein Feldspieler und ein Torwart mit den vorher eingelesenen Farben erstellt, bei der Erstellung wird auch übergeben, ob die Spieler im klassischen Modus dargestellt werden soll, oder nicht.

```
Player player0(teamColor0, classicView);
playerSpriteID[0][0][0] = player0.getID();

Player goalie0(goalieColor0, classicView);
playerSpriteID[0][1][0] = goalie0.getID();

Player player1(teamColor1, classicView);
playerSpriteID[1][0][0] = player1.getID();

Player goalie1(goalieColor1, classicView);
playerSpriteID[1][1][0] = goalie1.getID();
```

Nach dem die einzelnen Sprites[9] erstellt wurden, können nun die um einen Winkel rotierten Sprites erstellt werden. Dazu wird die Methode `rotate` benutzt, die ein

[9] Vom Hintergrund unabhängiges Grafikobjekt

beliebiges Objekt rotieren kann und als neues Objekt der Animation hinzufügt. Die Methode gibt die neue ID zurück, die im Array gespeichert wird. Die Anzal der rotierten Objekte wird auf 32 pro erstelltem Spieler beschränkt um den Speicherbedarf für die Animation nicht unnötig zu vergrößsern.

```
for (int angle = 1; angle < 32; angle++) {
  float alpha = (float)angle * PI / 16.0f;
  playerSpriteID[0][0][angle] =
        rotate(playerSpriteID[0][0][0], alpha);
  playerSpriteID[0][1][angle] =
        rotate(playerSpriteID[0][1][0], alpha);
  playerSpriteID[1][0][angle] =
        rotate(playerSpriteID[1][0][0], alpha);
  playerSpriteID[1][1][angle] =
        rotate(playerSpriteID[1][1][0], alpha);
}
```

Alle Objekte für die Animation sind erstellt worden, bevor nun die Erstellung der Animation beginnt, müssen noch zwei wichtige Schritte gemacht werden.

Es wird die Variable `slowmotion` aus der Konfigurationsdatei ausgelesen, ist diese auf `true` gesetzt werden die Frames herausgesucht, bei denen Tore gefallen sind. Dazu wird einmal die komplette Logdatei eingelesen und der passende *Playmode* herausgesucht, jedes so gefundene Tor wird in einem Array gespeichert. Auch die Anzahl der gefundenen Tore wird gespeichert, außerdem wird für den weiteren Verlauf die höchste Zeitmarke gespeichert.

```
OpenLogFile(_inFile); // read slowmotion points
while (smEnabled && (chunkMode = ReadLogFileChunk())) {
  if ((chunkMode == PM_MODE) && (smTime >= 0)) {
    switch(getPlaymode()) {
      case 14 : sm[smLength++] = maxTime;
           break;
      case 15 : sm[smLength++] = maxTime;
           break;
    }
  }
  if (chunkMode == SHOW_MODE) {
    maxTime = getTime();
  }
}
CloseLogFile();
```

Beim Einlesen der Datei werden auch zusätzlich andere Felder besetzt, so dass man erstmals die Teamnamen auf das Display zeichnen kann

```
display.setTeam(getTeamName(0), 0);
display.setTeam(getTeamName(1), 1);
```

Als zweiter Schritt werden die einzelnen Szenen aus der Konfigurationsdatei ausgelesen, sofern es welche gibt. Die Informationen werden in drei unterschiedlichen Arrays abgespeichert, einmal die Startpunkte, die Endpunkte und als drittes der zur Szene passende Text. Dabei ist zu beachten, dass man die Einblendung zur Halbzeit nur einbaut, falls überhaupt keine Szenen angegeben sind und so das komplette Spiel verarbeitet wird.

Nun sind alle Vorbereitungen zur Erstellung der Animation abgeschlossen. Als nächstes werden die Steuerelemente der Animation hinzugefügt, falls die passende Variable in der Konfigurationsdatei auf `true` gesetzt ist.

```
bool drawButtons = false;
cr.get("SWF::buttons", drawButtons);
if (drawButtons) DrawButtons();
```

Dann kommt die erste Einblendung, die die Namen der spielenden Mannschaften anzeigt (Abbildung 10.1). Falls die Steuerelemente gezeichnet wurden wird nun eine *Action* an die Animation angefügt.

```
if (drawButtons) {
  FCTDoAction* stop = new FCTDoAction();
  stop->AddAction(new FActionStop());
  flashObjects.AddFObj(stop);
  flashObjects.AddFObj(new FCTShowFrame()); // next frame
}
```

Mit der `FActionStop` wird die Animation angehalten, sie kann nun über die Steuerelement weiter gesteuert werden, sind keine Steuerelemente vorhanden, läuft die Animation von alleine weiter.

Falls Szenen angegeben wurden, wird nun über die Methode `toggleFlash` die erste Einblendung gemacht, die den Szenentext und die aktuelle Spielminute anzeigt. Dabei wird die Variable `startframe` auf den entsprechenden Wert gesetzt. Die Variable wird dann in der Variablen `endframe` gespeichert.

In einer Schleife werden nun nach und nach die einzelnen Blöcke aus der Logdatei ausgelesen, dabei werden die drei Modi `SHOW_MODE`, `PM_MODE` und `TEAM_MODE` verarbeitet. Das geschieht solange bis die höchste Zeitmarke `endframe` oder eben das Ende der Szene erreicht ist. Wird `endframe` erreicht, ist die Animation beendet. Falls der Endpunkt einer Szene erreicht ist, wird die Variable `startframe` auf den nächsten Szenenstart gesetzt. Es wird eine neue Einblendung mit den aktuellen Szenendaten gemacht und die Schleife startet mit den neuen Daten von vorne bis

die Szene wieder beendet ist, bei der letzten Szene wird das Ende dieser Szene in der Variablen `endframe` gespeichert, so dass die Animation nicht einfach bis `maxtime` weiter durchläuft.

Der `SHOW_MODE` (Kapitel 2.4) wird nur innerhalb der Zeitindizes der Variablen `startframe` und `endframe` verarbeitet. Es werden dann die einzelnen Objekte neu gezeichnet.

Bei jeder Zeitmarke werden die Spieler und der Ball neu gezeichnet.

```
int alpha = 0;
for (int i = 0; i < 23; i++) {
  if (isVisible(i)) {
    float angle = getAngle(i);
    newX = getX(i);
    newY = getY(i);
    if (timer>=3000) factor = -1;

    int x = (int)((xMin + xMax)/2 +
             newX * (xMax - xMin)/2 / fieldMeasureX * factor);
    int y = (int)((yMin + yMax)/2 +
             newY * (yMax - yMin)/2 / fieldMeasureY * factor);
```

Zuerst wird überprüft, ob die Spieler oder der Ball sichtbar sind, dann werden die Koordinaten ausgelesen und in die Animationskoordinaten umgewandelt, die Variable `factor` dreht die Koordinaten für den Seitenwechsel zur Halbzeit. Ist das Objekt der Ball wird aus den Koordinaten ein Rollwinkel gestimmt, da der Ball keinen eigenen Winkel hat.

```
    if (i == 22) {
      float a = newX - oldX;
      float b = newY - oldY;
      if (a == 0)
        angle = PI / 2;
      else
        angle = 2 * atan(b / a);

      if (!((oldX == newX) && (oldY == newY))) {
        ball[ballCounter%4].set(x, y, angle);
        ballCounter++;
      }
      oldX = newX;
      oldY = newY;
    }
```

Die Koordinaten werden in den Variablen `oldX` und `oldY` zwischengespeichert. Bei den Spielern muss kein Winkel berechnet werden, der aus der Logdatei ausgelesene Winkel wird in eine ganze Zahl zwischen 0 und 31 umgewandelt, damit der richtige Sprite aus dem Array `playerSpriteID` herausgesucht werden kann. Die Methode `set` bekommt außerdem noch den Layer und die Position des Spielers übergeben.

```
      else {
        int goalie = isGoalie(i);
        int team = i / MAX_PLAYER;
        if (factor == -1)
          angle += PI;
        alpha = (int)(angle / PI * 16.0f) & 31;
        set(playerSpriteID[team][goalie][alpha], 9000 + i, x, y);
      }
    } // end if(isVisible())
  } // end for
```

Wie in Kapitel 9.4 schon gezeigt werden alle zehn Sekunden (100 Frames) die Banden in der Animation gewechselt. Außerdem wird für jeden Frame die neue Zeitmarke auf das Display geschrieben. Danach wird der Frame gezeichnet.

```
display.setTime(timer); // set timestamp on display
flashObjects.AddFObj(new FCTShowFrame());
```

Sind die Zeitlupen für die Animation aktiviert und einer der vorher abgespeicherten Punkte (ein Tor) wird erreicht, wird ein neuer `startframe` gesetzt und der Zeitlupenmodus `smEnabled` aktiviert, die Schleife wird dann mit dem Frame `startframe` neu begonnen.

```
          if (sm[smCounter] == timer &&
              smCounter < smLength && smEnabled) {
            smCounter++;
            Reset();
            startframe = timer - smTime;
            timer = 0;
            slowmotion = true;
            break;
          }
```

Ist die Zeitlupe aktiviert wird mit der Methode `drawR` ein weißes „R“ in die rechte obere Ecke des Spielfeldes gezeichnet und ein zusätzlicher Frame eingefügt. Mit dem zusätzlichen Frame wird die Zeitlupe simuliert.

```
if (slowmotion && smEnabled) {
    display.setPlaymode(-1); // show slowmotion on display
    if ((timer % 2) == 0)
      blink = true;
    else
      blink = false;

    drawR(xMax - 800, yMin, xMax, yMin + 1000, blink);

    if (sm[smCounter - 1] == timer) {
      slowmotion = false;
    }
  }
```

Wird der gespeicherte Punkt wieder erreicht wird die Zeitlupe wieder deaktiviert.

Ist der Timer bei der Halbzeit angekommen wird die Anzeige zur Halbzeit eingeblendet, außerdem findet so ein Seitenwechsel statt, das heißt die Daten der Teams müssen auf dem Display auch die Seiten wechseln.

Abbildung 10.3: Einblendung zur Halbzeit

```
display.setScore(getTeamScore(1), 0);
display.setScore(getTeamScore(0), 1);

if (firstTimeSideChange) {
  display.setTeam(getTeamName(1), 0);
  display.setTeam(getTeamName(0), 1);
  firstTimeSideChange = false;
}
```

Wird beim Einlesen der Modus `PM_MODE` gefunden, wird der neue Spielzustand auf das Display geschrieben.

```
display.setPlaymode(getPlaymode());
```

Beim `TEAM_MODE`, der nur in der Logdatei geschrieben wird, wenn ein Tor fällt, wird das neue Spielresultat auf das Display geschrieben.

```
if (timer >= 3000) {
  display.setScore(getTeamScore(1), 0);
  display.setScore(getTeamScore(0), 1);

  if (firstTimeSideChange) {
    display.setTeam(getTeamName(1), 0);
    display.setTeam(getTeamName(0), 1);
    firstTimeSideChange = false;
  }
}
else {
  display.setScore(getTeamScore(0), 0);
  display.setScore(getTeamScore(1), 1);
}
```

Auch hier muss darauf geachtet werden, in welcher Halbzeit sich das Spiel gerade befindet.

Die Schleife läuft durch bis die Zeitmark `endframe` erreicht wird, dann werden zum Schluss alle Spielobjekte vom Feld gelöscht.

```
if (timer == endframe) {
  display.setTime(timer); // next time element
  for (int i = 0; i < 23; i++) {
    if (i == 22)
      ball[ballCounter%4].remove();
    else
      remove(9000 + i);
  }
}
```

Zum Schluss wird eine weitere *Action* an die Animation angefügt, damit nicht automatisch wieder an den Anfang der Animation gesprungen wird.

```
FCTDoAction* stop2 = new FCTDoAction();
stop2->AddAction(new FActionStop());
flashObjects.AddFObj(stop2);
flashObjects.AddFObj(new FCTShowFrame()); // next frame
```

Mit dem Aufruf der Methode `CreateMovie` wird dann die Animation in die angegebene Datei abgespeichert, dabei werden auch die Größe der Animation, sowie die Bildwiederholrate angegeben. Die so entstandene Flashdatei kann dann mit dem Flash Player aufgerufen und betrachtet werden.

Teil IV
Resümee

11 Fazit

Im vierten Teil der Arbeit wurde die Applikation vorgestellt, welche das Ziel der Arbeit, eine geeignete Visualisierung der Fußballsimulationen zu erstellen, erfüllt. Mit dem Tool kann man sehr schnell und leicht Flashanimationen erstellen. Diese Animationen können in einem Webbrowser betrachtet werden, den jeder internetfähige Computer heutzutage hat.

Die Animationen werden durch das Tool mit möglichst wenigen verschiedenen Objekten erstellt, dadurch wird die Flashdatei sehr klein gehalten. Es wird das *Place/move/remove*-Modell des gewählten Vektorgrafikformats Flash genutzt um einzelne Objekte mehrfach an verschiedene Stellen zu positionieren und von dort zu verschieben oder wieder zu löschen.

Die gewählte Vogelperspektive bei der Animation ist notwendig, da in der Logdatei keine Informationen über irgendwelche Höhen stehen, dadurch ist es nur sehr schwer möglich das Spiel in einer simulierten 3D-Umgebung ablaufen zu lassen. Die Vogelperspektive bietet allerdings den besten Überblick über das gesamte Spiel, da man so immer die komplette Spielfläche in der Animation betrachten kann.

Die Applikation wird bereits von einigen Teams[10] genutzt. Es kommt jetzt auf das Feedback der Benutzer an, welche Wünsche, Anregungen etc. man noch in das Tool einarbeiten kann. Aus diesem Grunde ist die Applikation modular programmiert, sie ist so leichter zu erweitern und zu verändern. Zusätzlich wird der Quellcode der Applikation veröffentlicht, damit jeder die Applikation auch nach seinen eigenen Wünschen anpassen kann.

Als weiteres Ausgangsmedium für die Simulationen war ein Videoformat gedacht. Die Programmmierung einer solchen Applikation ist allerdings aufwändig. Wenn man direkt aus den Daten der Simulation ein Video erstellen möchte, muss zunächst für jede Zehntelsekunde (soviele Daten sind vorhanden) ein Einzelbild erstellt werden und diese müssen dann anschließend zu einem Video zusammengestellt werden. Der Umweg über Flash ermöglichzt es, diese vor allem für den Computer aufwändige Arbeit zu umgehen. Mit bereits vorhandenen Tools, für Windows wie für Linux, lassen sich die Animationen am Bildschirm abfilmen. da gibt es natürlich auch verschiedene Möglichkeiten und die besseren Programme unter Windows kosten ca. €140. Diese sind dann aber auch sehr umfangreich, schnell und leicht zu bedienen. Mit umfangreich sind die vielen Auswahlmöglichkeiten, wie Auflösung, Video-Codec und Qualität gemeint, die es bei freien Tools nicht immer gibt. Ein Qualitätsmerkmal ist auch, ob man mit dem Computer bei einer laufenden Konvertierung auch weiterarbeiten kann oder nicht. Bei einigen Programmen, darf man das Fenster mit der Animation nicht bewegen. Einige Testversionen von kostenpflichtigen Programmen befinden sich auf der CD-Rom (Anhang B).

Im abschliessenden Kapitel soll nun noch ein kleiner Ausblick auf mögliche Erweiterungen und Ideen für weitere mit dieser Arbeit zusammenhängende Projekt gegeben werden.

[10] http://www.ni.uos.de/index.php?id=779

12 Ausblick

Adobe und vorher auch Macromedia stellen seit Flash 4 kein eigenes SDK mehr zur Verfügung und andere SDKs setzen die SWF-Spezifikationen teilweise fehlerhaft um. Um ein gutes und funktionierendes API für Flash zu haben, ist es nötig ein eigenes zu programmieren. Man kann dann die Programmiersprache des API wählen, allerdings sollte man es für die populärsten Programmiersprachen zur Verfügung stellen, dazu zählt neben C/C++ natürlich auch Java. Gerade für multimediale Anwendungen ist eine Zusammenarbeit von Java und Flash sinnvoll, da beide plattformunabhängig sind und einen objektorientierten Ansatz haben, bei Flash über die einzelnen Tags realisiert. In der Community java.net ([Web java.net]) gibt es schon einige Projekte die Java und Flash zusammenbringen, zum Beispiel durch einen Java Flash Player.

Mit einem aktuellen API kann man aber auch neue Features von Flash, genauer gesagt von ActionScript in die Animationen der Fußballsimulation einbinden. Durch ActionScript sind weitere Gestaltungsmöglichkeiten gegeben, die Animation interaktiver zu gestalten oder weitere Darstellungsformen dynamischer zu gestalten, als Beispiel sei die Zeitlupenwiederholung genannt, die man während eines Spieles an- und ausschalten könnte.

Die weitere Möglichkeit, eine 3D-Animation aus den 2D-Daten zu erstellen, ist zwar sehr schwierig, da wie vorhin erwähnt keinerlei Informationen über Höhen der Spieler und des Balles vorhanden sind, aber vielleicht kann man diese fehlende Koordinate irgendwie berechnen oder schätzen, für die Spieler ist es sicherlich am Einfachsten eine einheitliche Figur zu erstellen und die Flughöhe des Balles könnte man aus der Geschwindigkeit und der zurückgelegten Entfernung berechnen.

Flash ist allerdings für eine 3D-Simulation nicht so gut geeignet, da es ausschliesslich für 2D-Grafiken entwickelt wurde. Die 3D-Objekte müssten erst aufwändig durch 2D-Objekte simuliert werden und passend in die Szene eingesetzt werden, das läßt aber den Speicherbedarf der Animation in die Höhe schnellen. Außerdem gäbe es dann nur einen festen Blickwinkel, was für eine 3D-Simulation auch eher ungeeignet ist.

Es gibt aber genug Möglichkeiten eine 3D-Animation zu erstellen, angefangen bei OpenGL bis hin zu den SDKs, mit denen bekannte Spielehersteller ihre 3D-Welten generieren. Zum Beispiel ist Crystal Space 3D [Web CS3D] ein freies und portables 3D Game Development Kit, welches allerdings nicht nur für die Spieleentwicklung geeignet ist, sondern auch für generelle 3D-Visualierungen. Diese 3D-Simulation könnte man dann natürlich auch für eine geeignete Darstellung der RoboCup Soccer 3D Simulationen nutzen. Sie hat den Vorteil, dass sie auf den Betrachter realistischer wirkt, da man dort dann simulierte Figuren sieht, welche einem realen Spieler sehr nahe kommen können und außerdem kann sich der Be-

trachter in der Animation frei bewegen, er kann den Blickpunkt der Kamera frei einstellen.

Teil V
Anhang

A Spielmodi

In Tabelle 9 befinden sich alle Spielmodi der 2D-Simulation mit einer kurzen Erläuterung, in Tabelle 10 befinden sich die zusätzlichen Spielmodi der 3D-Simulation. Die 3D-Simulation wird zwar in der Arbeit nicht behandelt, der `LogFileReader` (Kapitel 7) ist aber trotzdem in der Lage die Zustände zu verarbeiten.

Variable	**Serverstring**	**Erläuterung**
`PM_Null`		Kein definierter Zustand
`PM_BeforeKickOff`	before_kick_off	Vor dem Anstoß des Spieles
`PM_TimeOver`	time_over	Das Spiel ist zuende
`PM_PlayOn`	play_on	Das Spiel läuft weiter
`PM_KickOff_Left`	kick_off_l	Anstoß linke Seite
`PM_KickOff_Right`	kick_off_r	Anstoß rechte Seite
`PM_KickIn_Left`	kick_in_l	Einfurf linke Seite
`PM_KickIn_Right`	kick_in_r	Einwurf rechte Seite
`PM_FreeKick_Left`	free_kick_l	Freistoß linke Seite
`PM_FreeKick_Right`	free_kick_r	Freistoß rechte Seite
`PM_CornerKick_Left`	corner_kick_l	Ecke linke Seite
`PM_CornerKick_Right`	corner_kick_r	Ecke rechte Seite
`PM_GoalKick_Left`	goal_kick_l	Abstoß linke Seite
`PM_GoalKick_Right`	goal_kick_r	Abstoß rechte Seite
`PM_AfterGoal_Left`	goal_l	Tor auf linker Seite
`PM_AfterGoal_Right`	goal_r	Tor auf rechter Seite
`PM_Drop_Ball`	drop_ball	Schiedsrichterball
`PM_OffSide_Left`	offside_l	Abseits linke Seite
`PM_OffSide_Right`	offside_r	Abseits rechte Seite

Tabelle 9: Spielmodi der 2D-Simulation

Variable	**Serverstring**
PM_PK_Left	penalty_kick_l
PM_PK_Right	penalty_kick_r
PM_FirstHalfOver	first_half_over
PM_Pause	pause
PM_Human	human_judge
PM_Foul_Charge_Left	foul_charge_l
PM_Foul_Charge_Right	foul_charge_r
PM_Foul_Push_Left	foul_push_l
PM_Foul_Push_Right	foul_push_r
PM_Foul_MultipleAttacker_Left	foul_multiple_attack_l
PM_Foul_MultipleAttacker_Right	foul_multiple_attack_r
PM_Foul_BallOut_Left	foul_ballout_l
PM_Foul_BallOut_Right	foul_ballout_r
PM_Back_Pass_Left	back_pass_l
PM_Back_Pass_Right	back_pass_r
PM_Free_Kick_Fault_Left	free_kick_fault_l
PM_Free_Kick_Fault_Right	free_kick_fault_r
PM_CatchFault_Left	catch_fault_l
PM_CatchFault_Right	catch_fault_r
PM_IndFreeKick_Left	indirect_free_kick_l
PM_IndFreeKick_Right	indirect_free_kick_r
PM_PenaltySetup_Left	penalty_setup_l
PM_PenaltySetup_Right	penalty_setup_r
PM_PenaltyReady_Left	penalty_ready_l
PM_PenaltyReady_Right	penalty_ready_r
PM_PenaltyTaken_Left	penalty_taken_l
PM_PenaltyTaken_Right	penalty_taken_r
PM_PenaltyMiss_Left	penalty_miss_l
PM_PenaltyMiss_Right	penalty_miss_r
PM_PenaltyScore_Left	penalty_score_l
PM_PenaltyScore_Right	penalty_score_r

Tabelle 10: Zusätzliche Spielmodi der 3D-Simulation

B Inhalt der CD-Rom

arbeit	**Diplomarbeit**
arbeit/latex	Latex-Quellen und Grafiken
arbeit/pdf	PDF-Version
arbeit/flash	Flash-Version
applikation	**Applikation der Diplomarbeit**
applikation/quellen	Quelldateien der Applikation
applikation/binaries	Ausführbare Dateien
daten	**Anwendungsdaten der Applikation**
daten/logdateien	Logdateien des RoboCup Soccer Server
daten/animation	Flashanimationen der Logdateien
dokumente	**Zusätzliche Dokumente**
software	**Zusätzliche Software**
software/converter	Flash2Video-Konvertierer
software/flash	Flash Player und Flash SDK
software/robocup	Software des RoboCup

C Literaturverzeichnis

Aufgrund der Aktualität der in dieser Diplomarbeit eingesetzten Techniken sind viele Informationen noch nicht in Büchern zu finden. Daher beziehen sich die meisten Literaturverweise auf Websites, deren Erreichbarkeit und Inhalt sich schnell ändern können. Zum Zeitpunkt der Erstellung dieser Diplomarbeit waren alle Quellen unter den hier aufgeführten Links erreichbar.

Literatur

[Adb2005] **Adobe Systems**
Macromedia Flash (SWF) and Flash Video (FLV) File Format Specification Version 8
Adobe Systems, 2005

[Cpp2000] **Bjarne Stroustrup**
Die C++-Programmiersprache
Addison-Wesley; Auflage: 4. Aufl. (15. Mai 2000)

[Make79] **Stuart I. Feldman**
Dake-A Program for Maintaining Computer Programs
Journal Software - Practice and Experience, 1979

[RC1993] **Alan Mackworth**
On Seeing Robots; Kapitel 1, S. 1-13
World Scientific Press, 1993

[RC1995] **Hiroaki Kitano, Minoru Asada, Yasou Kuniyoshi, Itsuki Noda, and Eiichi Osawa**
RoboCup: The Robot World Cup Initiative. In Proc. of IJCAI-95 Workshop on Entertainment and AI/Alife, S. 19-24, 1995.

[RC2001] **Itsuki Noda et al.**
RoboCup Soccer Server
http://www.robocup.org, 2001

[Web CS3D] **Crystal Space 3D**
http://www.crystalspace3d.org/main/Main_Page

[Web FHist] **Adobe Flash History**
http://de.wikipedia.org/wiki/Adobe_Flash#Geschichte

[Web FPoll] **Adobe Flash Player Version Penetration**
http://www.adobe.com/products/player_census/flashplayer/version_penetration.html

[Web java.net] **java.net - The Source for Java Technology Collaboration**
http://java.net/

[Web NI UOS] **Neuroinformatics Group, Universität Osnabrück**
http://www.ni.uos.de/

[Web RCSS] **The RoboCup Soccer Simulator**
http://sserver.sourceforge.net

[Wol2002] **Sascha Wolter**
Flash MX
Galileo Press GmbH, Bonn 2002

Erklärung

Hiermit erkläre ich, dass ich die Diplomarbeit selbstständig angefertigt und keine anderen Quellen und Hilfsmittel außer den in der Arbeit angegebenen benutzt habe.

Rheine, den 16.03.2007 .....

Tobias Schwegmann