



FACHHOCHSCHUL - BACHELORSTUDIENGANG
AUTOMATISIERUNGSTECHNIK

Entwurf eines Analysetools für den Spielverlauf bei der Eurobot

Als Bachelorarbeit eingereicht

zur Erlangung des akademischen Grades

Bachelor of Science in Engineering

von

ROBERT RATHGEBER

April 2010

Betreuung der Bachelorarbeit durch

Prof(FH) Dipl. Ing. Walter Rokitansky



Campus **Wels**

Ich erkläre ehrenwörtlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt, die den benutzten Quellen entnommenen Stellen als solche kenntlich gemacht habe und dass die Arbeit mit der vom Begutachter beurteilten Arbeit übereinstimmt.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

.....
«Vorname, Name»

.....
«Wohnort», «Datum»

Kurzfassung

In dieser Arbeit wird ein einfaches Simulationstool vorgestellt. Dieses Tool simuliert den Verfahrensweg eines mobilen Roboters, welcher für den Eurobot-Bewerb 2010 entwickelt wurde. Die Arbeit befasst sich im Wesentlichen mit den Thematiken der Datenhaltung, Grafik- und Multi-Mediaprogrammierung unter der Entwicklungsumgebung Visual Studio C#.

Im Bezug auf Datenhaltung wird auf Konzepte eingegangen, die es ermöglichen Daten in ein Projekt einzubinden. Dabei werden Möglichkeiten erläutert, die nicht den Einsatz einer Datenbank erfordern, da Datenbanken einen erhöhten Aufwand hervorrufen.

Im Kapitel Grafikprogrammierung werden Techniken vorgestellt um Animationen zu erstellen. Es wird aufgezeigt wie man auf die Qualität der Animation Einfluss nehmen kann. Des weiteren wird erläutert wie es möglich ist Videos in die Entwicklungsumgebung einzubinden.

Alle vorgestellten Thematiken werden zuerst allgemein, meist anhand einfacher Beispiele erläutert und im Anschluss auf das erstellte Projekt referenziert.

Inhaltsverzeichnis

Inhaltsverzeichnis	III
1 Motivation	1
1.1 Eurobot[1]	1
1.2 Eurobot - Thema 2010[2]	2
1.2.1 Spielelemente	2
1.3 Gesamtes Regelwerk	3
2 Projektbeschreibung	4
2.1 Anforderungen	4
3 Vorstellung des Gesamtkonzeptes	5
3.1 Form Movieplayer	5
3.2 Form Simulation	6
3.3 Class Hilfsmethode	7
3.4 Class Position	8
3.5 Class PositionList	8
4 Realisierung der Problemstellungen	9
4.1 Datenhaltung	9
4.1.1 Dateien[3]	10
4.1.1.1 Projektbezug	11
4.1.2 Initialisierungsdateien[4]	11
4.1.2.1 Klasse INI[5, 6]	12
4.1.2.2 Projektbezug	12
4.1.3 Ressourcen[7, 8]	13
4.1.3.1 Einfügen von Ressourcen	13
4.1.3.2 Verwenden von Ressourcen	13
4.1.3.3 Projektbezug	14
4.1.4 Datenbanken[9]	14
4.2 Grafikoperationen[10]	14

4.2.1	Einfache Grafikoperationen	15
4.2.2	Transformationen[11, 12]	16
4.2.2.1	Verschieben des Koordinatenursprungs	16
4.2.2.2	Rotationen	17
4.2.2.3	Hinweis	18
4.2.3	Vom Bild zur Animation	18
4.2.3.1	Die Animation	18
4.2.3.2	Double Buffering	19
4.2.4	Projektbezug	21
4.3	Implementieren eines Videoplayers	21
4.3.1	Konfiguration	21
5	Ausblick	23
	Literaturverzeichnis	24
	Abbildungsverzeichnis	25

Kapitel 1

Motivation

Es ist auf der Welt nichts unmöglich, man muss nur die Mittel entdecken, mit denen es sich durchführen lässt.

Hermann Oberth (1894 - 1989) , deutscher Physiker und Raumfahrtpionier

Die heutige technische Welt ist geprägt von der Robotertechnik. Das Einsatzspektrum von Robotern ist weit gefächert. Das Anwendungsgebiet reicht von einfachen Positionierungsaufgaben bis hin zu vollautomatisierten Fertigungslinien. Die Robotik vereint unter anderem Ingenieursdisziplinen wie Elektronik, Informatik und Maschinenbau.

Das RoboRacingTeam der Fachhochschule Wels hat es sich zum Ziel gesetzt, Studenten die Thematik der Robotik näher zu bringen und diese dafür zu begeistern. Während des Studiums besteht die Möglichkeit, in Form von Projektarbeiten mit dem RoboRacingTeam zusammenzuarbeiten. Im Zuge dieser Projektarbeiten wird jeweils in einem Team ein Roboter für einen nationalen oder internationalen Wettbewerb konstruiert.

1.1 Eurobot[1]

Die Eurobot ist aus einem nationalen französischen Wettbewerb hervorgegangen. Das RoboRacingTeam nimmt seit 2007 erfolgreich an dem internationalen Wettbewerb Eurobot teil. Die Aufgabe für den Bewerb wird jedes Jahr neu definiert. Einige Grundregeln sind fixe Bestandteile und werden nicht von Jahr zu Jahr geändert. Die Spielfeldgröße beträgt immer 300 cm x 210 cm, die Spieldauer ist auf 90 sec festgelegt und der maximale Umfang eines Roboters darf während des Spieles 1400 mm nicht überschreiten.

1.2 Eurobot - Thema 2010[2]

Das Thema der Eurobot 2010 ist „Feed the World“. Die Roboter befinden sich auf einem Feld mit Getreide, Früchte und Gemüse. Aufgabe der Roboter ist, so viele der genannten Güter wie möglich innerhalb der 90 Sekunden einzusammeln. Jeder Roboter darf alle Elemente aufnehmen. Die Punkte werden nach Gewicht vergeben. Der Roboter, der am Ende das meiste Gewicht in den Sammelbehältern hat, gewinnt das Spiel.

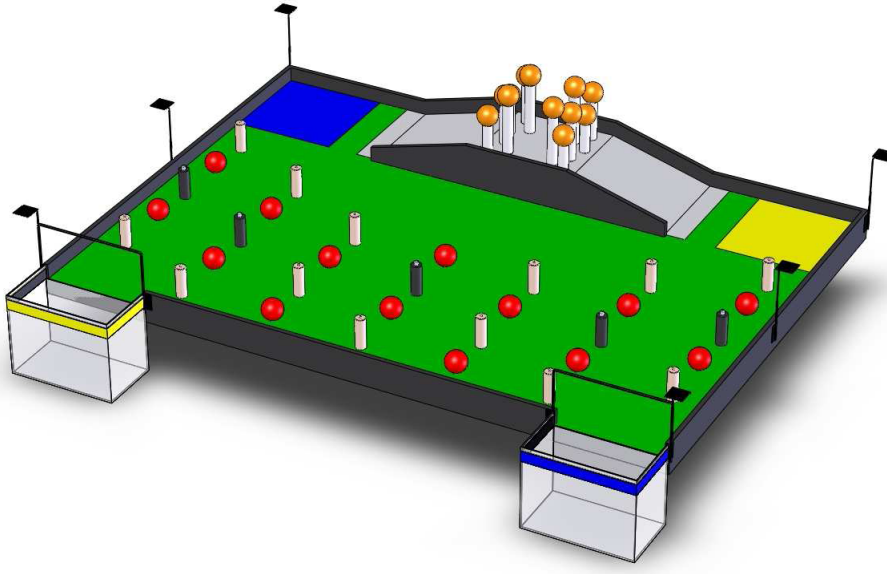


Abbildung 1.2.1: Spielfeld Eurobot 2010

1.2.1 Spielelemente

Wie bereits erwähnt, müssen verschiedene Elemente eingesammelt werden. Diese Elemente unterscheiden sich durch Gewicht, Material und Lage am Spielfeld.

1. Die Orangen sind direkt neben dem Startfeld abzuholen. Es gilt eine Rampe hochzufahren, anschließend müssen die Orangen von einem Baum gepflückt werden. Die Problematik dabei ist, dass die Orangen (sechs pro Seite) in allen drei Dimensionen zueinander versetzt sind. Die Orange ist mit 300g das schwerste Element am Spielfeld und am schwierigsten aufzunehmen.
2. Die Getreideähren sind am ganzen Spielfeld in fix definierten Abständen verteilt. Diese bestehen aus einem weißen Kunststoffzylinder und wiegen 250g. Mit einem Zapfen werden die Ähren in Bohrungen gesteckt. Es befinden sich auch schwarze Ähren auf dem Spielfeld, welche nicht aufgenommen werden können, da sie fest im Boden verschraubt sind. Die Position der schwarzen Ähren ist von Spiel zu Spiel verschieden.
3. Die Tomaten wiegen nur 150 g und bringen somit am wenigsten Punkte, dafür sind davon reichlich am Spielfeld vorhanden. Die Tomaten sind zwischen den Kornähren positioniert.

1.3 Gesamtes Regelwerk

Das gesamte Regelwerk umfasst 44 Seiten, zusätzlich gibt es Ergänzungen zum Regelwerk, die so genannten FAQ. Das Regelwerk für die Eurobot 2010 kann unter der Homepage der Eurobot nachgelesen werden. <http://www.eurobot.org/eng/rules.php>

Kapitel 2

Projektbeschreibung

Im Zuge der interdisziplinären Projektarbeit im fünften Semester wurde ein einfaches Analysetool erstellt, mit dem es möglich ist, den Spielverlauf zu simulieren. Mit einer einfachen Grafikoberfläche wird das Spielfeld und die Roboter (als Rechtecke) dargestellt. Problematisch ist, dass das Spielfeld von Spiel zu Spiel verschieden sein kann. Die Verfahrwegdaten werden über eine Textdatei eingelesen, welche während dem Spiel am Roboter mitgeloggt werden. Zusätzlich soll es möglich sein, Videos vom Spiel parallel anzusehen.

Die Software soll möglichst einfach auf jedem Computer ausführbar sein, weswegen das Programm mit der Programmiersprache C# erstellt wurde. Für die Datenhaltung wird auf eine Datenbankbindung verzichtet.

2.1 Anforderungen

Daraus ergeben sich folgende Anforderungen an das Programm:

- Datenhaltung: Speichern von Benutzerinformationen und Spielfeldversionen, Einlesen von Spielinformationen
- Anpassen des Spielfeldes
- Rotationen von Grafikobjekten (Roboter)
- Darstellen einer flüssigen Simulation
- Implementieren eines Videoplayers

In der folgenden Ausarbeitung wird zuerst auf die Gesamtstruktur der Software eingegangen, im Anschluss die Realisierung der einzelnen Anforderungspunkten vorgestellt.

Kapitel 3

Vorstellung des Gesamtkonzeptes

In der folgenden Abbildung wird die Struktur des Projektes in vereinfachter Form dargestellt und die einzelnen Forms und selbst erstellten Klassen kurz erläutert.

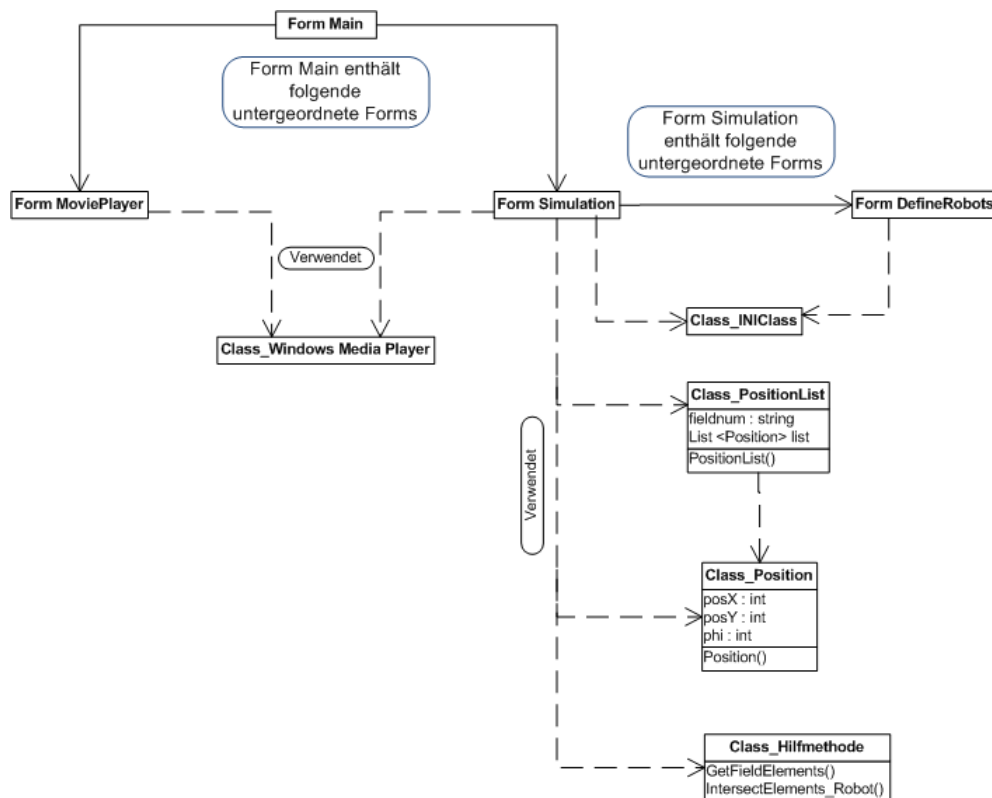


Abbildung 3.0.1: Programmstruktur

Startet man das Programm, öffnet sich die Main-Form. Diese ist der Einstiegspunkt in die Software. In diesem Einstiegspunkt kann man nun auswählen ob man die Simulationsoberfläche öffnen möchte, oder den einfachen Videoplayer verwenden will.

3.1 Form Movieplayer

Die Form Movieplayer wurde erstellt um unabhängig von der Simulation Videos ansehen zu können. Die Implementierung wird in Abschnitt 4.3 genauer besprochen.

3.2 Form Simulation

In der Form Simulation kann der Spielablauf simuliert werden. Es ist möglich parallel zur Simulation ein Video des Spiels anzusehen.

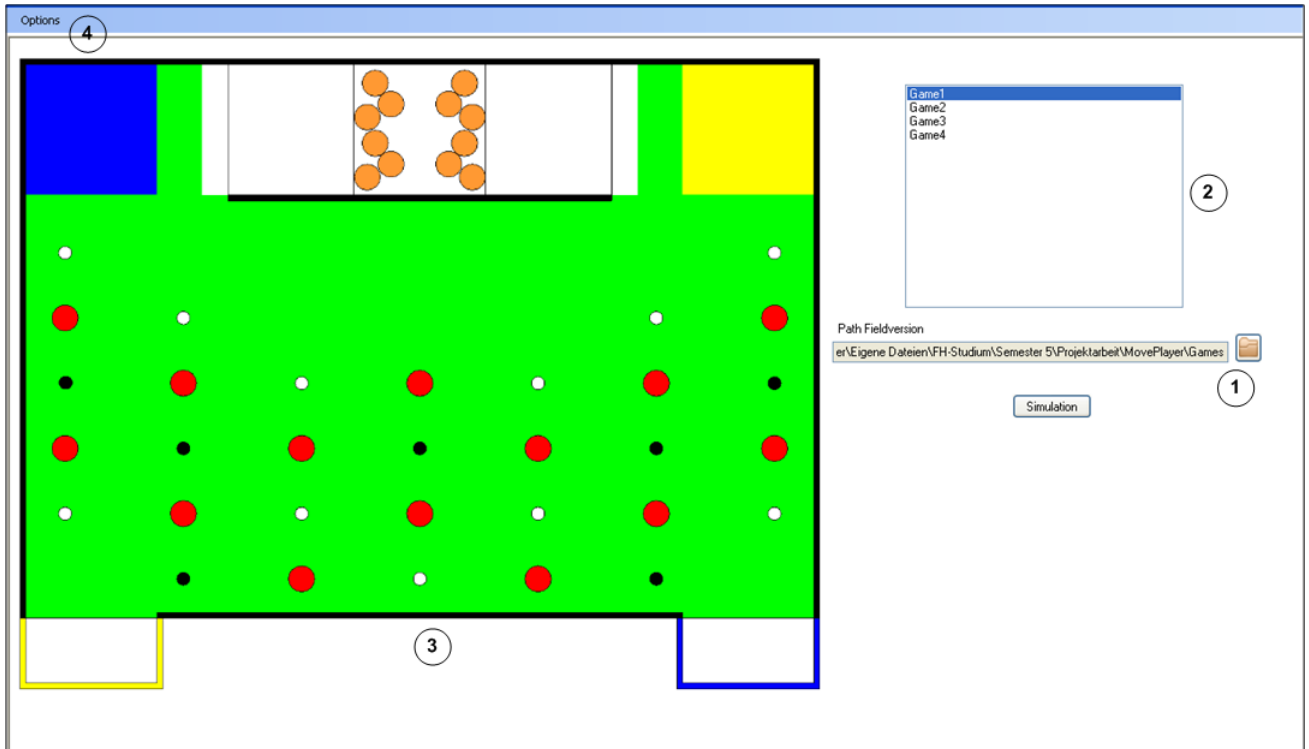


Abbildung 3.2.1: Oberfläche Simulation

1. Um eine Simulation starten zu können, muss angegeben werden, wo sich die CSV-Dateien mit den Bewegungskoodinaten befinden, welche während des Spieles am Roboter mitgeloggt wurden. Am Rechner muss dazu ein Ordner erstellt werden in dem die einzelnen Spieldaten abgelegt werden. Die Spieldaten eines Spiels müssen in Unterordnern abgelegt werden. Als Spieldaten gelten:
 - Positionsliste Roboter FH Wels
 - Positionsliste Roboter Gegner
 - Video des Spiels

Um die Simulation ausführen zu können, muss mindestens die Positionsliste des FH-Roboters im Unterordner vorhanden sein. Beim Ablegen der Dateien sind Namenskonventionen einzuhalten. Die Positionsliste unseres Roboters muss den Dateinamen „*rrtWels.csv*“ besitzen. Der gegnerische Roboter muss mit dem Dateinamen „*Gegner.csv*“ benannt werden. Ein Videofile muss mit den Buchstaben „*Vid*“ beginnen. Über das Ordnersymbol kann ein Browserdialog geöffnet werden mit dem angegeben werden kann, wo der Spielordner abgelegt ist.

2. Wurde der Spielordner erfolgreich angegeben, erscheinen in der Listbox die Unterordner der einzelnen Spiele. Wird ein Spiel ausgewählt, werden die Positionslisten geladen. Jetzt ist es möglich, mit dem Button „*Simulation*“ den Ablauf der Simulation zu starten.

3. Die Simulation wird in einem Panel dargestellt und der Hintergrund automatisch auf die Spielfeldversion eingestellt.
4. In der Menüleiste ist es unter Optionen möglich, die Größe der Roboter einzustellen. Hierfür öffnet sich ein kleines Unterform in dem die Maße definiert werden können. Die Daten bleiben auch nach dem Beenden des Programms gespeichert und müssen nicht jedes mal erneut definiert werden. Eine weitere Funktion ist „*Split Window*“. Ist im Spielordner ein Videofile vorhanden, wird über diese Funktion ein Split-Container aufgezogen, und es erscheint ein Videofenster.

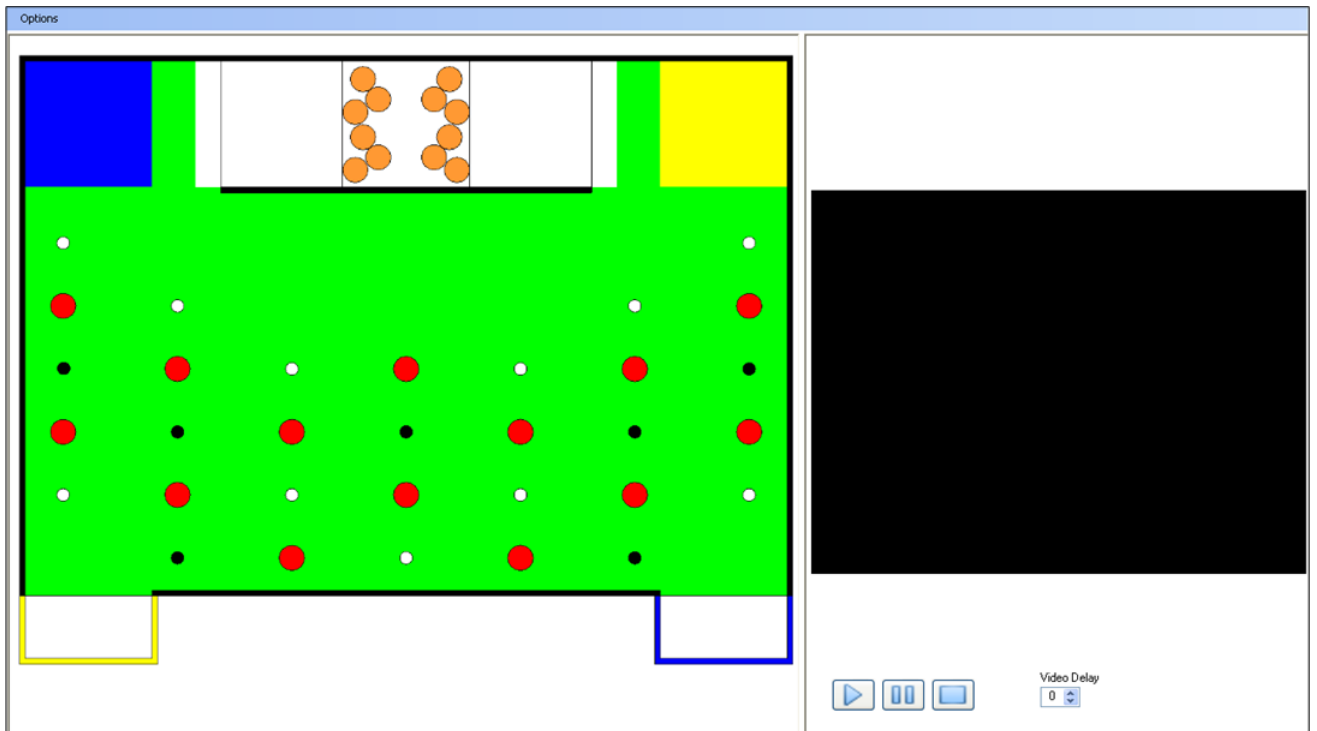


Abbildung 3.2.2: Form Simulation mit Videofenster

Der Splitcontainer wird über die Ordernavigation gezogen, der Button „*Simulation*“ ist verschwunden. Die Simulation wird nun über das Videofenster gestartet. Es kann vorkommen, dass der Start des Spieles am Video einige Sekunden später beginnt. Dieser Offset kann mit dem „*Video Delay*“ korrigiert werden. Die Eingabe entspricht einem Offset in Sekunden. Der Splitcontainer verschwindet, wenn die Simulation zu Ende ist, oder die Simulation von Hand gestoppt wird.

3.3 Class Hilfsmethode

Die Klasse Hilfsmethode enthält zwei Methoden, die zum Betrieb der Simulation erforderlich sind. Dabei handelt es sich um folgende Methoden:

- `public static Rectangle[] GetFieldElements(double scaleFaktor)`

Über diese Methode werden die auf dem Feld vorhandenen Feldelemente eingelesen, das heißt die Position und Größe von Tomaten und Getreideähren. Die Daten werden aus einer Ressource eingelesen. In Abschnitt 4.1.3 werden Ressourcen genauer behandelt. Die

Methode gibt diese Daten als ein Array von Rechtecken zurück. Der übergebene Wert „*scaleFaktor*“ wird für die Skalierung der Maße benötigt, da die Daten in der Ressource in den realen Abmaßen abgelegt sind.

- `public static void IntersectElement_Robot(Rectangle[] allElements, Rectangle robot1, Rectangle robot2, ref Rectangle[] intersectElements)`

Während der Simulation bewegen sich die Roboter am Spielfeld. Fährt ein Roboter über ein Feldelement soll dieses verschwinden. Diese Methode wertet aus ob die Roboter über ein Element gefahren sind. Dazu werden sowohl die Positionsdaten der Roboter als Rechtecke, als auch das Array mit den Feldelementen übergeben. Die Methode prüft, ob eine Rechtecksüberschneidung zwischen den Robotern und den Feldelementen vorliegt. Wenn dies zutrifft wird das betroffene Feldelement in das Rechteckarray „*intersectElements*“ eingetragen.

3.4 Class Position

Die Klasse Position dient zur einfachen Handhabung einer Roboterposition auf dem Spielfeld. Die Position ist gegeben durch:

- Position in der X-Richtung: `posX`
- Position in der Y-Richtung: `posY`
- Winkelorientierung: `phi`

Auf die Membervariablen selbst kann nicht direkt zugegriffen werden. Diese müssen über den Konstruktor der Klasse gesetzt werden, und sind über die Eigenschaften der Klasse abrufbar.

3.5 Class PositionList

In der Klasse PositionList können die gesamten Verfahrspositionen für einen Roboter abgelegt werden. Diese Klasse wird ebenfalls über den Konstruktor definiert. Der Konstruktor ist wie folgt definiert:

- `public PositionList(string path)`

Dem Konstruktor ist ein Pfad zu übergeben. Dies ist jener Ablagepfad, wo die CSV-Datei für die Roboterpositionen abgelegt ist. Mit dem Aufruf des Konstruktors wird eine Liste mit Abfahrpositionen erstellt. Zusätzlich existiert die Membervariable „*feldnum*“. Diese Variable gibt an, um welche Spielfeldversion es sich handelt. Die Spielfeldversion ist in der ersten Zeile der CSV-Datei definiert und wird ebenfalls vom Roboter bereitgestellt. Während des Spieles werten Kameras aus, welche Spielfeldkonstellation vorliegt. Die Positionsliste und die Variable „*feldnum*“ können ausschließlich über die entsprechenden Eigenschaften abgerufen werden.

Kapitel 4

Realisierung der Problemstellungen

Wie schon in den Anforderungen definiert wurde, ergaben sich bei diesem Projekt verschiedenste Problemstellungen. Im folgenden Abschnitt wird auf diese Problemstellungen eingegangen und Lösungsansätze präsentiert.

4.1 Datenhaltung

Oft wird von einer Software gefordert Daten zu verwalten, bereitzustellen oder zu speichern. Daten sind aber nicht gleich Daten. Hier sind verschiedene Unterscheidungen zu treffen:

1. Datentyp

In der heutigen virtuellen Welt existieren eine Unmenge an verschiedenen Daten. Bei der Datenverwaltung muss man sich Gedanken darüber machen, welche Art von Daten man verwalten will. Handelt es sich um einfache Zahlentypen, Texttypen oder um komplexere Daten wie Bilder. Nicht jede Form der Datenhaltung kann mit allen Datentypen operieren.

2. Umfang der Daten

Es ist zu beachten welchen Umfang die Datenmenge annimmt. Es besteht einerseits die Möglichkeit, dass eine Software nur wenige Daten sichern muss, andererseits kann aber auch der Fall auftreten, dass die Datenanzahl mit der Verwendung laufend anwächst.

3. Ändern von Daten

Es ist selten der Fall, dass die Daten immer dieselben sind. Informationen sollten vom Benutzer auch geändert werden können.

4. Zusammenhang der Daten

Man sollte beachten ob verschiedene Datensätze miteinander im Zusammenhang stehen. Das heißt, dass ausgehend von einem Datensatz Rückschlüsse auf einen anderen Datensatz gezogen werden müssen.

5. Aufwand für die Datenhaltung

Ein sehr wichtiges Kriterium für die Auswahl welche Art der Datenhaltung man wählt ist der Aufwand der Realisierung. Abhängig von den oben angeführten Kriterien sollte man

den Aufwand, der zur Realisierung und auch für den späteren Betrieb notwendig ist, nicht außer Acht lassen.

Ausgehend von den angegebenen Kriterien gilt es zu entscheiden, welche Form der Datenhaltung man wählt. Im folgendem Abschnitt wird auf verschiedene Möglichkeiten eingegangen, wie Daten abgelegt werden können.

4.1.1 Dateien[3]

Die wahrscheinlich älteste Möglichkeit Daten zu sichern, ist das Ablegen in einer Datei. Die Entwicklungsumgebung von C# stellt hierfür Klassen zur Verfügung, die im Namespace „*System.IO*“ definiert sind. Es ist sowohl möglich Daten aus einer Datei zu lesen als auch in eine Datei zu schreiben.

1. Die Klasse StreamWriter

Mit der Klasse StreamWriter wird ein schreibender Zugriff auf eine Datei geöffnet. Ein Aufruf kann wie folgt aussehen:

```
StreamWriter datei = File.CreateText(@"C:\temp\file.txt");
datei.WriteLine("Dies ist ein Eintrag");
datei.Close();
```

Abbildung 4.1.1: Codebeispiel StreamWriter

In diesem Codebeispiel wird ein StreamWriter mit dem Namen „*datei*“ erstellt. Mit dem Aufruf „*File.CreateText(string path)*“ wird eine Datei erstellt, in die hineingeschrieben wird. Existiert bereits eine Datei mit dem gleichen Namen, wird diese überschrieben. Mit „*datei.WriteLine(string text)*“ wird der Text in den Klammern in die Datei hinausgeschrieben. Um sicherzustellen, dass der Text in die Datei geschrieben wird, muss ein abschließendes Close erfolgen.

2. Die Klasse StreamReader

Mit der Klasse StreamReader wird ein lesender Zugriff auf eine Datei geöffnet. Ein Aufruf kann wie folgt aussehen:

```
StreamReader datei = File.OpenText(@"C:\temp\file.txt");
string text = datei.ReadLine();
datei.Close();
```

Abbildung 4.1.2: Codebeispiel StreamWriter

Mit dem Aufruf „*File.OpenText(string path)*“ wird ein schreibender Zugriff auf die angegebene Datei geöffnet, anschließend eine Zeile aus der Datei ausgelesen und auf eine Stringvariable geschrieben. Es wäre nun möglich die Datei Zeile für Zeile auszulesen, indem man den Befehl „*datei.ReadLine()*“ erneut ausführt. Man kann aber auch die Datei in einem Schritt komplett auslesen, in dem man die Methode „*datei.ReadToEnd()*“ aufruft.

Es gibt natürlich weit aus mehr Möglichkeiten im Bezug auf Filehandling, an dieser Stelle wird aber nicht weiter auf diese Thematik eingegangen und auf die entsprechende Fachliteratur verwiesen.

Die Vorteile von Dateien sind, dass ein einfacher Schreib- bzw. Lesezugriff durch wenige Codezeilen realisiert werden kann. Die Dateien werden extern abgelegt und können somit auch außerhalb eines Programms angesehen und verändert werden. Dies kann sowohl seine Vorteile, als auch seine Nachteile haben. Das Problem ist, dass einfache Textdateien, in die laufend hineingeschrieben wird, schnell unübersichtlich werden. Nach Informationen muss gesucht werden.

4.1.1.1 Projektbezug

Das Analysetool greift auch auf externe CSV-Dateien zu. Wie bereits erwähnt werden in den CSV-Dateien die Verfahrspositionen der Roboter übergeben. CSV-Dateien sind strukturierte Textdateien in denen einzelne Datenfelder durch Trennzeichen getrennt werden. Als Trennzeichen wird das Semikolon verwendet, ein neuer Datensatz ist getrennt durch einen Zeilenumbruch.

In der Klasse „*PositionList*“ wird beim Aufruf des Konstruktors die CSV-Datei eingelesen und in eine Positionsliste übertragen. Dies wurde folgendermaßen realisiert:

```
public PositionList(string path)
{
    StreamReader file = new StreamReader(path);
    string helptext = string.Empty;
    string [] helparray;
    Position helpPos;
    this.Fieldnum = file.ReadLine();

    do
    {
        helptext = file.ReadLine();
        helparray = helptext.Split(';');
        helpPos=new Position(Convert.ToInt32(helparray[0]),
            Convert.ToInt32(helparray[1]),Convert.ToInt32(helparray[2]));
        this.List.Add(helpPos);
    } while (file.Peek() >=0);
}
```

Abbildung 4.1.3: Generieren der PositionList

Die CSV-Datei wird Datensatzweise also Zeile für Zeile in einer Schleife ausgelesen. In der Schleife wird der Datensatz durch ein Stringsplitt aufgeteilt und als Position in die Positionsliste hinzugefügt. Über den Aufruf „*file.Peek*“ in der While-Bedingung wird abgefragt ob das Ende der Datei erreicht ist.

4.1.2 Initialisierungsdateien[4]

Initialisierungsdateien (kurz: INI-Datei) sind Textdateien, in denen Benutzereinstellungen abgespeichert werden können. Der Vorteil dabei ist, dass das Ablegen von Daten in diese Datei strukturiert erfolgt. INI-Dateien können in Sektionen unterteilt werden. Will man zum Beispiel für mehrere Forms Benutzereinstellungen abspeichern, kann für jede Form eine Sektion definiert werden. Wichtig dabei ist, dass Sektionsnamen eindeutig sind, also nur einmal vorkommen. In einer Sektion werden Werte mit einem Key abgelegt.

4.1.2.1 Klasse INI[5, 6]

Die Entwicklungsumgebung stellt für INI-Dateien im Namespace „*Kernel.32*“ Funktionen für Initialisierungsdateien bereit. Grundsätzlich können über die Funktionen „*GetPrivateProfileString*“ und „*WritePrivateProfileString*“ INI-Dateien ausgelesen und geschrieben werden. Eine relativ einfach zu verwendende Klasse wird auf der Internetfachseite http://www.codeproject.com/KB/cs/cs_ini.aspx bereitgestellt.

Die bereitgestellte Klasse „*INI*“ besitzt einen Konstruktor, in dem der Pfad der INI-Datei angegeben wird.

```
public IniFile(string INIPath)
{
    path = INIPath;
}
```

Abbildung 4.1.4: Konstruktor Class INI

Anschließend kann in die INI-Datei über die Methode „*IniWriteValue*“ geschrieben werden. Dazu ist eine Sektion anzugeben, der Name der Sektion kann frei gewählt werden. Weiters muss ein Key und der zu speichernde Wert angegeben werden.

```
public void IniWriteValue(string Section, string Key, string Value)
{
    WritePrivateProfileString(Section, Key, Value, this.path);
}
```

Abbildung 4.1.5: Methode IniWriteValue

Zum Lesen einer INI-Datei wird der Methodenaufruf „*IniReadValue*“ verwendet. Dem Methodenaufruf muss die Sektion und der Key übergeben werden. In der Methode wird die INI-Datei mit dem Aufruf „*GetPrivateProfileString*“ ausgelesen. Dieser Methode wird zusätzlich zur Sektion und dem Key ein DefaultString übergeben. Dieser wird auf „*temp*“ geschrieben, falls die Methode keinen gültigen Wert beim Auslesen ermitteln kann. Wurde ein gültiger Wert gefunden, wird dieser ebenfalls auf „*temp*“ gespeichert. Die vorletzte Zahl beschreibt die Buffergröße, als letztes wird der Pfad übergeben, wo die INI-Datei zu finden ist. Als Rückgabewert besitzt die Methode „*GetPrivateProfileString*“ einen Integerwert, welcher angibt, wie viele Zeichen in den Buffer geschrieben wurden.

```
public string IniReadValue(string Section, string Key)
{
    StringBuilder temp = new StringBuilder(255);
    int i = GetPrivateProfileString(Section, Key, "Not Found", temp, 255, this.path);
    return temp.ToString();
}
```

Abbildung 4.1.6: Methode IniReadValue

4.1.2.2 Projektbezug

Im Projekt wurde die oben beschriebene Klasse verwendet, um Benutzereinstellungen, wie zum Beispiel Größenangaben der Roboter, zu speichern. In der Form „*DefineRobots*“ können die Ab-

maße der Roboter definiert werden. Beim Aufruf der Form wird das INI-File aus dem Hauptprogramm mitübergeben und die voreingestellten Abmaße aus der INI-Datei gelesen. Beim Speichern werden die Werte zurück in die Datei geschrieben.

```

IniFile forminfo;

public DefineRobots(IniFile ini)
{
    InitializeComponent();
    forminfo = ini;

    tbxLengthRRT.Text = ini.IniReadValue("Robots", "lengthRRT");
    tbxHeightRRT.Text = ini.IniReadValue("Robots", "heightRRT");
    tbxLengthEnemy.Text = ini.IniReadValue("Robots", "lengthEnemy");
    tbxHeightEnemy.Text = ini.IniReadValue("Robots", "heightEnemy");
}

private void btnSave_Click(object sender, EventArgs e)
{
    forminfo.IniWriteValue("Robots", "lengthRRT", tbxLengthRRT.Text);
    forminfo.IniWriteValue("Robots", "heightRRT", tbxHeightRRT.Text);
    forminfo.IniWriteValue("Robots", "lengthEnemy", tbxLengthEnemy.Text);
    forminfo.IniWriteValue("Robots", "heightEnemy", tbxHeightEnemy.Text);
}

```

Abbildung 4.1.7: Define Robots

4.1.3 Ressourcen[7, 8]

Eine weitere Möglichkeit der Datenhaltung ist das Verwenden von Ressourcen. Sind Daten wie Bilddateien, Zeichenketten oder Sounddateien während der Entwicklung bekannt, kann man diese direkt in das Projekt über Ressourcen einbinden. Diese Ressourcen werden in Ressourcendateien definiert. Dies kann von Vorteil sein wenn man eine Programm mehrsprachig gestalten möchte oder die Oberfläche während der Laufzeit geändert werden soll.

4.1.3.1 Einfügen von Ressourcen

Zum Hinzufügen einer Ressource muss zuerst der Projektmanager-Explorer expandiert werden. Unter dem Knoten *Properties* kann die *Ressources.resx-Datei* geöffnet werden. In der Symbolleiste des Ressourcen-Designers lassen sich über die Schaltfläche „Ressourcen hinzufügen“ Zeichenfolgen oder Bilder einfügen. Zeichenfolgen-Ressourcen werden direkt in der Ressourcendatei *Ressources.resx* gespeichert. Bilder, Symbole, Audio- und Textdatei werden im automatisch angelegten Projektunterverzeichnis *Ressources* abgelegt und in *Ressources.resx* als Verweis eingetragen.

4.1.3.2 Verwenden von Ressourcen

In der Quelldatei *Ressources.Designer.cs* hat Visual Studio eine Klasse *Ressources* im Namespace des Projekts (`<Projektnamespace>.Properties`) angelegt. Diese Klasse definiert für jede Ressource eine gleichnamige statische Nur-Lesen-Eigenschaft, über diese man auf die Ressource zugreifen kann. Der Zugriff kann wie folgt aussehen:

```
button1.Image = PrjNamespace.Properties.Resources.ResourceManager.GetObject (picture);  
  
//oder  
  
button1.Image = PrjNamespace.Properties.Resources.picture;
```

Abbildung 4.1.8: Zugriff auf Ressource

4.1.3.3 Projektbezug

Im Projekt „Move Simulator“ ist gefordert, dass der Spielfeldhintergrund zur Laufzeit, abhängig von der benötigten Spielfeldversion, geändert werden kann. Dazu sind die einzelnen Bilder der Feldversionen als Ressourcen hinterlegt. Wird eine neue Spieldatei ausgewählt, wird die Feldversion aus den Positionlisten ausgelesen und das gleichnamige Bild aus den Ressourcen als Hintergrundbild in das Zeichenpanel eingefügt.

```
bmpGround =  
    new Bitmap ( (Bitmap) (MovePlayer.Properties.Resources.ResourceManager.GetObject (posRrtWels.Fieldnum)) );
```

Abbildung 4.1.9: Laden eines Hintergrundes aus den Ressourcen

4.1.4 Datenbanken[9]

Eine der wohl mächtigsten und wichtigsten Möglichkeiten zur Datenverwaltung sind die Datenbanken. In der Datenverarbeitung sollen Daten permanent verfügbar sein. Daten, die einmal eingegeben wurden, sollen zu einem späteren Zeitpunkt abgerufen und bearbeitet werden können. Mit Datenbanken besteht die Möglichkeit verschiedene zusammenhängende Datensätze kombiniert auszuwerten.

Um eine Datenbank realisieren zu können benötigt man zusätzliche Datenbanksysteme. Einfache Systeme sind Access oder MySQL. Um eine Datenbank effektiv verwenden zu können muss man dazu eine Datenbank modellieren. Das bedeutet man muss Datentabellen mit deren Variablen definieren. Es müssen Zusammenhänge zwischen den einzelnen Datentabellen aufgestellt werden. Bereits das Modellieren einer Datenbank kann viel Zeit in Anspruch nehmen, es muss auch der reibungsfreie Betrieb gewährleistet werden. Dazu sind Wartungen und Sicherungen der Datenbank erforderlich.

Deshalb sollte man sich vor der Implementierung die Frage stellen, ob der Aufwand für eine Datenbank wirklich gerechtfertigt ist. Der angestrebte Zweck sollte mit minimalem Aufwand verwirklicht werden.

Aus diesem Grund wurde beim Projekt „Move Simulator“ auf eine Datenbank verzichtet.

4.2 Grafikoperationen[10]

GDI – das Graphics Device Interface – ist die wichtigste Schnittstelle zur freien Programmierung von sichtbaren Elementen einer Anwendung mit grafischer Benutzeroberfläche, Erzeugung und Manipulation von Bildern in verschiedenen Formaten. DOTNET liefert eine Anzahl von Klassen,

die zur Programmierung von zweidimensionalen grafischen Elementen, Manipulation von Bitmap-Grafiken und der Darstellung von Texten dienen. Die Namespaces, in denen diese Klassen zu finden sind, sind unter dem Begriff GDI+ zusammengefasst.

Der wichtigste Namespace zum Zeichnen von Grafiken ist „System.Drawing“. In diesem Namespace befinden sich grundlegende Typen und Klassen mit denen man sehr einfach Formen wie Kreise, Rechtecke oder auch Bilder zeichnen und verschieben kann.

4.2.1 Einfache Grafikoperationen

Unter Visual Studio können sehr einfach Grafikobjekte gezeichnet und verschoben werden. Dies soll anhand eines Beispiels demonstriert werden. In einer Formanwendung wird ein Panel eingefügt. Beim Betätigen eines Buttons soll ein Rechteck gezeichnet werden, welches jeweils 10 Maßeinheiten aus dem Ursprung verschoben und 100 mal 60 Maßeinheiten groß ist. Der Code schaut wie folgt aus.

```
private Graphics zeichnung;
Rectangle rect = new Rectangle(10, 10, 100, 60);

private void button1_Click(object sender, EventArgs e)
{
    zeichnung = zeichenpanel.CreateGraphics();
    zeichnung.FillRectangle(Brushes.Red, rect);
}
}
```

Abbildung 4.2.1: Zeichnen eines Rechteckes

Zuerst wird ein Grafikobjekt (Graphics zeichnung) definiert auf dem die Zeichenoperationen ausgeführt werden. Das Rechteck wird nach den Angaben definiert. Beim Event „button1_Click“ wird das Grafikobjekt „zeichnung“ mit dem Panel „zeichnenpanel“ verknüpft. Im Anschluss wird ein rotes Rechteck gezeichnet. Das Ergebnis würde so aussehen:

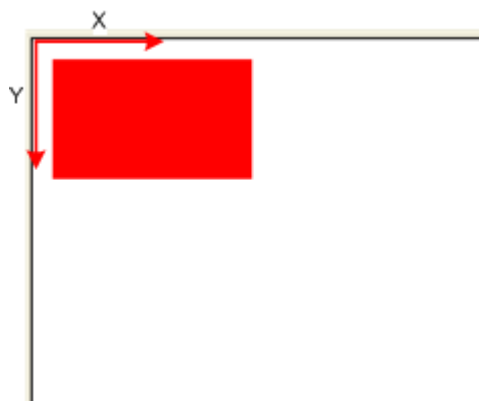


Abbildung 4.2.2: Panel mit Rechteck

Betrachtet man das Ergebnis aus Abbildung 4.2.2 wird schnell klar dass das Koordinatensystem des Panels in der linken oberen Ecke liegt, der Ursprungspunkt des Rechteckes befindet sich ebenfalls an der linken oberen Ecke. Die positive Zählrichtung für die Y-Koordinate zeigt nach unten. Viele Anwendungen beziehen ihren Ursprung aber auf die linke untere Ecke. Um dies zu

erreichen, kann man versuchen mit einfacher Mathematik die Koordinaten umzurechnen, oder dies mit Transformationen zu realisieren. Dazu mehr im nächsten Abschnitt.

4.2.2 Transformationen[11, 12]

Mit Transformationen können verschiedene Verschiebungen im Raum bzw in der Ebene realisiert werden. Transformationen erfolgen in X/Y-Richtung oder um einen Drehwinkel. Mit einer Transformation kann auch eine andere Skalierung erreicht werden. Das Graphics-Objekt verwaltet dazu eine Instanz der Klasse Matrix, die über die Eigenschaft Transform erreicht werden kann. In diese Eigenschaft kann ein eigens erzeugtes und initialisiertes Matrix-Objekt geschrieben werden um eine spezielle Transformation zu erreichen. Transformationsmatrizen können sehr komplex ausfallen, deshalb ist es möglich Transformationen direkt anzusprechen. In der folgenden Ausarbeitung werden drei Transformationen besprochen:

- TranslateTransform
- ScaleTransform
- RotateTransform

In Abschnitt 4.2.1 stellte sich das Problem, dass der Ursprung nicht in der linken unteren Ecke lag. Dies kann mit Transformationen einfach korrigiert werden. Dazu verwendet man die Transformationen „*Translate Transform*“ und „*Scale Transform*“.

4.2.2.1 Verschieben des Koordinatenursprungs

Um den Koordinatenursprung richtig zu stellen müssen zwei Operationen ausgeführt werden. Zuerst muss der Ursprung des Koordinatensystems in die linke untere Ecke verschoben werden. Dazu verwendet man den Befehl „*Translate Transform*“ . Als zweiten Schritt muss die Zählrichtung für die Y-Koordinate invertiert werden. Hierfür wird der Befehl „*Scale Transform*“ verwendet. Auf das Beispiel aus Abschnitt 4.2.1 umgelegt würde das folgende Abänderung ergeben:

```
private void button1_Click(object sender, EventArgs e)
{
    zeichnung = zeichenpanel.CreateGraphics();
    zeichnung.TranslateTransform(0, zeichenpanel.Size.Height);
    zeichnung.ScaleTransform(1.0F, -1.0F);
    zeichnung.FillRectangle(Brushes.Red, rect);
}
```

Abbildung 4.2.3: Verschieben des Koordinatenursprungs

TranslateTransform bewirkt ein Verschieben des Ursprungs. Im Codebeispiel wird der Ursprung in der Y-Koordinate um die Panelhöhe nach unten verschoben. Danach wird die positive Zählrichtung der Y-Koordinate mit dem Befehl ScaleTransform umgedreht. Im Aufruf von ScaleTransform muss ein Skalierungsfaktor für die X und Y-Koordinate angegeben werden. Da die Zählrichtung der X-Koordinate nicht verändert werden muss, ist dieser Skalierungsfaktor Eins.

Für Y wird als Skalierungsfaktor minus Eins angegeben, dies bewirkt die Umkehrung der Zählrichtung. Es können auch Skalierungsfaktoren ungleich Eins angegeben werden, was einen Einfluss auf die Größe der Zeichnung hat.

Führt man den geänderten Code aus, erhält man folgendes Ergebnis:

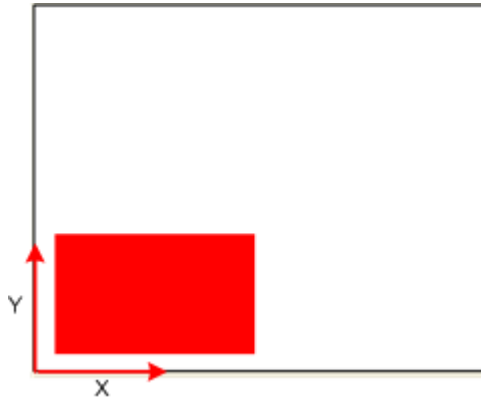


Abbildung 4.2.4: Verschieben des Koordinatenursprungs-Ergebnis

Der Koordinatenursprung liegt jetzt wie gewünscht in der linken unteren Ecke.

4.2.2.2 Rotationen

Eine weitere wichtige Transformation ist die Rotation. Mit Hilfe des Befehls „RotateTransform“ können Rotationen ausgeführt werden. Es ist zu beachten, dass Rotationen um den Koordinatenursprung ausgeführt werden. Die Rotation kann um einen beliebigen Winkel erfolgen, wobei positive Winkel im Uhrzeigersinn drehen und negative entgegen.

Hierzu ein kleines Beispiel. Der Koordinatenursprung wird in die Panelmitte gesetzt, ebenso ein Rechteck. Danach erfolgt eine Drehung um 45°. Der Code dazu lautet wie folgt:

```
Rectangle rect = new Rectangle(-50, -30, 100, 60);

private void button2_Click(object sender, EventArgs e)
{
    zeichnung = zeichenpanel.CreateGraphics();
    zeichnung.TranslateTransform(zeichenpanel.Width/2, zeichenpanel.Height/2);
    zeichnung.RotateTransform(45);
    zeichnung.FillRectangle(Brushes.Red, rect);
}
}
```

Abbildung 4.2.5: Code Rotation

Daraus resultiert folgendes Ergebnis:

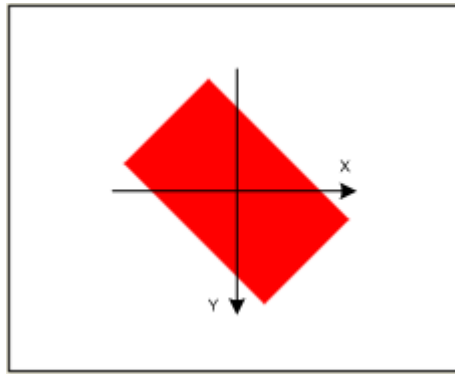


Abbildung 4.2.6: Rotation um den Koordinatenursprung

Wie erwartet, wurde das Rechteck um 45° gedreht. Genau genommen müsste das Koordinatensystem mitgedreht werden, da das gesamte Koordinatensystem gedreht wird.

4.2.2.3 Hinweis

Alle Koordinatentransformationen werden durch Multiplikation mit der internen Transformationsmatrix kombiniert. Sind Skalierungen und Rotationen beteiligt, hängt der Effekt aufeinander folgender Transformationen von ihrer Reihenfolge ab. Transformationen können zurückgesetzt werden indem man den Befehl „*ResetTransform*“ anwendet.

4.2.3 Vom Bild zur Animation

In den vorangegangenen Abschnitten wurde behandelt wie man einfache Grafikoperationen durchführt und diese durch Transformationen beeinflussen kann. Als nächsten Schritt soll gezeigt werden wie man Animationen erstellt und wie die Qualität der Animation beeinflusst wird.

4.2.3.1 Die Animation

Wie eine Animation funktioniert kann am Prinzip des Daumenkinos erklärt werden. Man nimmt Bilder aus einer Bewegungssequenz auf und blättert diese schnell durch. Liegen die Einzelbilder der Bewegung nahe beisammen und ist die Geschwindigkeit mit der man diese Bilder durchblättert schnell genug, so glaubt man eine Bewegung zu sehen. Dies ist unter dem Begriff des Stroboscopheneffekt bekannt. Beträgt die Taktgeschwindigkeit des Bildaufbaus etwa 16Hz, so verschmelzen die Einzelbilder zu einem kontinuierlichen Ablauf.

Genau mit dem gleichen Prinzip kann man Animationen realisieren. Dazu benötigt man nur Bildmaterial, das man im Panel nacheinander abspielt. Die Abspielgeschwindigkeit wird durch einen Taktgeber vorgegeben.

Als Beispiel wird ein einfacher Sekundenzeiger realisiert. Der Zeiger wird durch eine dicke Linie dargestellt, welche jede Sekunde um sechs Grad weitergedreht wird.

```

Pen stift = new Pen(Color.Black, 10);

private void btnStart_Click(object sender, EventArgs e)
{
    zeichnung = zeichenpanel.CreateGraphics();
    zeichnung.TranslateTransform(zeichenpanel.Width / 2, zeichenpanel.Height / 2);
    zeichnung.DrawLine(stift, 0, 0, 0, -80);
    timer1.Start();
}

private void timer1_Tick(object sender, EventArgs e)
{
    zeichnung.Clear(Color.White);
    zeichnung.RotateTransform(6);
    zeichnung.DrawLine(stift, 0, 0, 0, -80);
    zeichenpanel.Update();
}

```

Abbildung 4.2.7: Umlaufender Sekundenzeiger

Zuerst wird ein Zeichenstift definiert mit dem der Strich für den Zeiger gezeichnet wird. Beim Betätigen eines Startbuttons, wird ein Graphicsobjekt mit einem in das Form eingefügten Panel verknüpft und der Koordinatenursprung in die Panelmitte verschoben. Als Taktgeber fungiert ein Timer, der mit einer Taktgeschwindigkeit von einer Sekunde eingestellt wurde. Jede Sekunde wird das Koordinatensystem um sechs Grad weitergedreht und der Sekundenzeiger neu gezeichnet.

Das Ergebnis aus diesen Codezeilen ist bereits recht brauchbar, dennoch ist auf den ersten Blick zu erkennen, dass der Zeiger in schrägen Positionen unscharf ist und feine Kanten aufweist. Der Grund dafür ist, dass das Anzeigegerät, sprich der Monitor, mit Pixeln arbeitet. Aus diesem Grund können Übergänge nicht scharf dargestellt werden. Diesem Problem kann man durch Einfärben der Pixel an den Übergängen entgegenwirken. Unter C# ist dieses Verfahren als Anti-Aliasing definiert, was aber standardmäßig deaktiviert ist, da dieses Verfahren eine erhöhte Rechenbelastung hervorruft. In C# kann dieses Verfahren durch eine Codezeile aktiviert werden, die wie folgt lautet:

```
zeichnung.SmoothingMode = System.Drawing.Drawing2D.SmoothingMode.AntiAlias;
```

Abbildung 4.2.8: Anti-Aliasing

Zum Einschalten des Anti-Aliasing muss die Eigenschaft *SmoothingMode* des Grafikobjektes gesetzt werden. Dies erfolgt über die Enumeration *System.Drawing.Drawing2D.SmoothingMode.AntiAlias*.

Nach erneutem Ausführen sind die Übergänge wesentlich glatter. Ein weiteres Problem tritt auf, wenn die Rotationsgeschwindigkeit des Zeigers erhöht wird. Setzt man den Timerwert auf 50ms stellt sich ein unangenehmes Flackern ein. Das Flackern entsteht, wenn Zeichenoperationen kurz hintereinander durchgeführt werden. Um dies zu vermindern kann man die Technik des Double Buffering verwenden.

4.2.3.2 Double Buffering

Beim Double Buffering zeichnet man nicht direkt in den Gerätekontext, sondern erstellt sich im Arbeitsspeicher eine Kopie des Gerätekontextes, auf dem man alle Zeichenoperationen ausführt. Wenn alle Zeichenoperationen durchgeführt sind, wird der Hilfsspeicher zurück in den Gerätekontext kopiert.

Es gibt zwei Möglichkeiten Double Buffering zu realisieren. Eine Möglichkeit besteht darin die Eigenschaft *DoubleBuffered* des Panels auf true zu setzen. Die Eigenschaft *DoubleBuffered* ist protected. Deshalb muss eine eigene Klasse erstellt werden, nach der die Klasse Panel abgeleitet wird, anschließend kann die Eigenschaft auf true gesetzt werden. Im folgenden Codeausschnitt wird dies demonstriert.

```
public partial class myPanel : Panel
{
    public myPanel()
    {
        base.DoubleBuffered = true;
    }
}
```

Abbildung 4.2.9: MyPanel

Nachdem die Klasse abgeleitet wurde steht in der Toolbox das abgeleitete Panel zur Verfügung und kann in die Entwurfsansicht hineingezogen werden. Der Nachteil bei dieser Form des Double Buffering ist, dass nur Zeichenoperationen innerhalb des Paint-Events doppelt gepuffert werden können. Möchte man außerhalb des Paint-Events Double Buffering betreiben, ist diese Funktion selbst zu erstellen.

Wie das aussehen kann wird am Beispiel des rotierenden Zeiger vorgestellt. Im Vergleich zu den anderen Codebeispielen werden alle Grafikoperationen in den Timer gezogen.

```
Pen stift = new Pen(Color.Black, 10);
Graphics zeichnung;
BufferedGraphicsContext bgc;
BufferedGraphics bg;
int winkel = 0;

private void timer1_Tick(object sender, EventArgs e)
{
    zeichnung = zeichenpanel.CreateGraphics();
    bgc = BufferedGraphicsManager.Current;
    bg = bgc.Allocate(zeichnung, zeichenpanel.ClientRectangle);

    bg.Graphics.SmoothingMode = System.Drawing.Drawing2D.SmoothingMode.AntiAlias;
    bg.Graphics.Clear(Color.White);|
    bg.Graphics.TranslateTransform(zeichenpanel.Width / 2, zeichenpanel.Height / 2);
    bg.Graphics.RotateTransform(winkel);
    bg.Graphics.DrawLine(stift, 0, 0, 0, -80);
    bg.Render();

    bg.Dispose();
    zeichnung.Dispose();
    winkel += 6;
}
```

Abbildung 4.2.10: Double Buffering

Im Timer-Event wird wie gehabt zuerst das Grafikobjekt *zeichnung* mit dem Zeichenpanel verknüpft. In den nächsten Codezeilen wird eine Kopie des Gerätekontextes erstellt. Weiters werden alle Zeichenoperationen wie in den vorangegangenen Beispielen durchgeführt. Gezeichnet wird aber nicht mehr auf das Grafikobjekt *zeichnung* sondern in den Puffer *bg*. Zum Abschluss wird mit dem Befehl *bg.Render()* der Pufferinhalt in den Zielgerätekontext übertragen und der Puffer entleert.

4.2.4 Projektbezug

Die Analysesoftware MoveSimulator verwendet alle Techniken die in den vorhergehenden Abschnitten präsentiert wurden. Der Koordinatenursprung für die Simulation befindet sich in der rechten oberen Ecke, weswegen der Koordinatenursprung geändert werden muss. Mit den Methoden Translate- und RotateTransform werden die Roboter, die als Bitmap dargestellt sind, auf dem Spielfeld positioniert. Die Zeichenoperationen erfolgen doppelt gepuffert, da ansonsten der Bildaufbau erheblich flackern würde.

In Abbildung 4.2.11 wird ein Codefragment zur Positionierung eines Roboters gezeigt.

```
bg.Graphics.ResetTransform();
bg.Graphics.TranslateTransform((Single)panBattleground.Width, 0);
bg.Graphics.ScaleTransform((Single)(-1.0F), (Single)1.0F);
bg.Graphics.TranslateTransform((Single)(offsetGround / scaleFactor),
                               (Single)(offsetGround / scaleFactor));
bg.Graphics.TranslateTransform((Single)(posRrtWels.List[aktPosition].PosX / scaleFactor),
                               (Single)(posRrtWels.List[aktPosition].PosY / scaleFactor));
bg.Graphics.RotateTransform(posRrtWels.List[aktPosition].Phi);
bg.Graphics.DrawImage bmpRrtWels, Convert.ToSingle(-bmpRrtWels.Width / (2)),
                      Convert.ToSingle(-bmpRrtWels.Height / (2)));
```

Abbildung 4.2.11: Positionierung eines Roboters

Der angeführte Codeausschnitt stammt aus dem Timer-Event, in welchem alle Zeichenoperationen durchgeführt werden. Ein Bestandteil davon ist die Positionierung eines Roboters. Die Initialisierung des Zeichenpuffers erfolgt in einem anderen Codeteil. Wie zu erkennen ist, erfolgen alle Zeichenoperationen auf diesem Puffer. Auf diesen Codeausschnitt folgen noch weitere Zeichenoperationen, die hier aber nicht aufgeführt werden, da dies den Umfang sprengen würde. Nachdem alle Zeichenoperationen ausgeführt wurden, erfolgt das Kopieren des Puffers in den Zielgerätekontext.

4.3 Implementieren eines Videoplayers

Die Implementierung eines Videoplayers ist sehr schnell zu realisieren, in nur wenigen Minuten kann man einen Videoplayer in seine eigenen Forms einbauen. Es gibt viele Bibliotheken, die es ermöglichen Videos abzuspielen. Für viele benötigt man jedoch etwas mehr Hintergrundwissen über Video-Streaming und deren Programmierung.

Für einfache und schnelle Anwendungen bietet Microsoft auf der Internetseite <http://msdn.microsoft.com> den Windows Media Player SDK zum Download an. Diese Installationsdatei enthält Bibliotheken die es ermöglichen den Windows Media Player in Visual Studio einzubinden. Nach der Installation sind die Verweise AxWMPLib und WMPLib hinzuzufügen, im Anschluss ist der Windows Media Player in der Toolbox verfügbar.

4.3.1 Konfiguration

Der MediaPlayer muss aus der Toolbox nur auf einer Formoberfläche aufgezogen werden. Es erscheint die bekannte Media Player-Oberfläche. Unter den Eigenschaften des Objektes kann zum

Beispiel die Steuerelementausstattung bestimmt oder Lautstärkeneinstellungen vorgegeben werden. Im Modus *full* sind alle Steuerelemente wie Buttons und Lautstärkeneinstellung vorgegeben, es muss im Programmcode nur noch der Videopfad definiert werden. Der Modus *none* stellt nur das Videofenster zur Verfügung, Abspielbefehle müssen im Code selbst geschrieben werden.

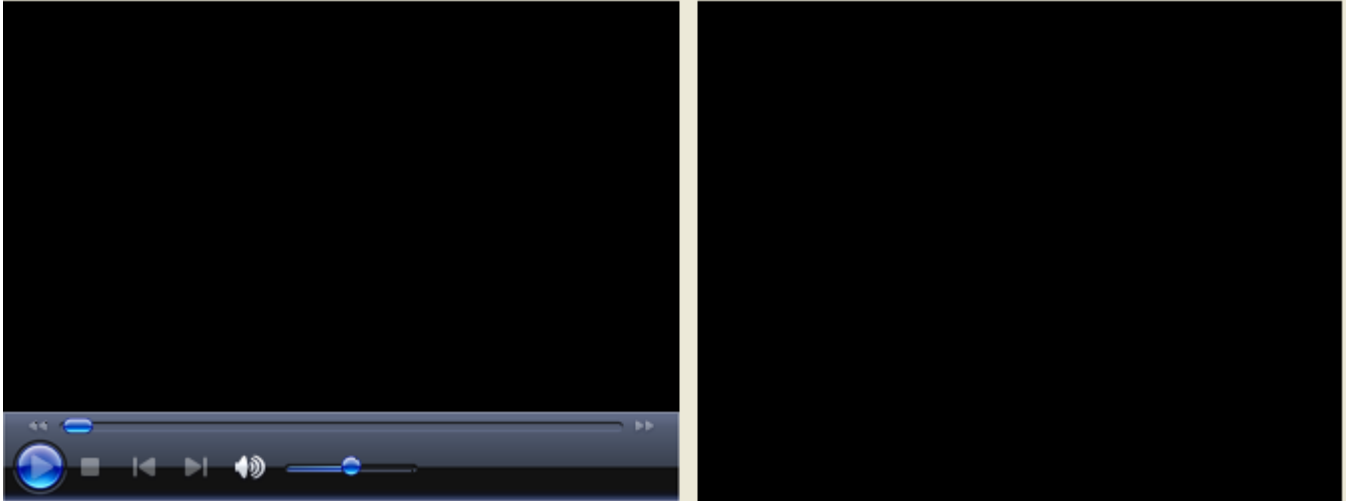


Abbildung 4.3.1: Modus *full* und *none*

In der folgenden Tabelle werden die wichtigsten Methoden bzw. Eigenschaften des Windows Media Players aufgelistet. Das Kürzel *wmp* steht für den Namen mit dem der Player im Programmcode angesprochen wird.

Aufruf	Funktion
wmp.URL	Eigenschaft auf die der Pfad des Videos verwiesen wird
wmp.Ctlcontrols.currentPosition	Eigenschaft für die aktuelle Videoposition
wmp.Ctlcontrols.play()	Methode zum Starten des Videos
wmp.Ctlcontrols.pause()	Methode zum Anhalten des Videos
wmp.Ctlcontrols.stop()	Methode zum Beenden des Videos
wmp.Ctlcontrols.fastForward()	schneller Vorwärtslauf
wmp.Ctlcontrols.fastReverse()	schneller Rückwärtslauf
wmp.settings	Settings enthält Untereigenschaften zum Einstellen der Lautstärke oder Stummsschaltung

Tabelle 4.3.1: Methoden bzw. Eigenschaften des Windows Media Players

Darüber hinaus kann natürlich noch mehr realisiert werden, wie zum Beispiel das Erstellen einer Playlist, auf das an dieser Stelle aber nicht weiter eingegangen wird.

Kapitel 5

Ausblick

Das Projekt *MoveSimulator* dient in erster Linie zur Analyse der Verfahrswege. Es würde sich aber anbieten das Projekt mit weiteren Funktionen zu versehen. Ausgehend von den Positionslisten könnte der maximal zurückgelegte Weg der Roboter ausgewertet werden. Aus dem maximalen Weg könnte eine Durchschnittsgeschwindigkeit ermittelt werden. Weiters wäre es möglich auszuwerten in welchen Spielfeldbereichen sich der Gegner hauptsächlich aufgehalten hat. Auch die Grafikaufbereitung kann noch adoptiert werden. Es wäre möglich den Verfahrsweg in einem eingeschränkten Bereich nachleuchten zu lassen.

Literaturverzeichnis

- [1] (2010, Februar). [Online]. Available: <http://www.eurobot.org/>
- [2] (2009, September) Feed the world - rules 2010. [Online]. Available: <http://www.eurobot.org>
- [3] S. S. Dirk Louis, *CSharp 2008 - Das Entwicklerbuch, Seite 509 - 511*. Microsoft Press Deutschland.
- [4] Initialisierungsdatei. [Online]. Available: <http://de.wikipedia.org/wiki/Initialisierungsdatei>
- [5] Ini-class. [Online]. Available: http://www.codeproject.com/KB/cs/cs_ini.aspx
- [6] Registry functions. [Online]. Available: <http://msdn.microsoft.com/en-us/library/ms724875%28VS.85%29.aspx>
- [7] P. Monadjemi, *Route 66 to .NET, Seite 79 - 83*. Markt+Technik Verlag, 2003.
- [8] S. S. Dirk Louis, *CSharp 2008 Das Entwicklerbuch, Seite 594 - 596*. Microsoft Press Deutschland.
- [9] A. Herbolsheimer, *Datenbank Programmierung, Seite 13-15*, ser. Programmers Choice. Addison Wesley Verlag, 2002.
- [10] J. M. F. Andreas Maslo, *.Net Framework Developers Guide, Seite 508 und folgende*. Markt+Technik Verlag, 2002.
- [11] J. Bayer, *Das CSharp Codebook, Seite 649 -652*. Addison-Wesley Verlag, 2003.
- [12] S. S. Dirk Louis, *CSharp 2008 Das Entwicklerbuch, Seite 839 -843*. Microsoft Press Deutschland.

Abbildungsverzeichnis

1.2.1 Spielfeld Eurobot 2010	2
3.0.1 Programmstruktur	5
3.2.1 Oberfläche Simulation	6
3.2.2 Form Simulation mit Videofenster	7
4.1.1 Codebeispiel StreamWriter	10
4.1.2 Codebeispiel StreamWriter	10
4.1.3 Generieren der PositionList	11
4.1.4 Konstruktor Class INI	12
4.1.5 Methode IniWriteValue	12
4.1.6 Methode IniReadValue	12
4.1.7 Define Robots	13
4.1.8 Zugriff auf Ressource	14
4.1.9 Laden eines Hintergrundes aus den Ressourcen	14
4.2.1 Zeichnen eines Rechteckes	15
4.2.2 Panel mit Rechteck	15
4.2.3 Verschieben des Koordinatenursprungs	16
4.2.4 Verschieben des Koordinatenursprungs-Ergebnis	17
4.2.5 Code Rotation	17
4.2.6 Rotation um den Koordinatenursprung	18
4.2.7 Umlaufender Sekundenzeiger	19
4.2.8 Anti-Aliasing	19
4.2.9 MyPanel	20
4.2.10 Double Buffering	20
4.2.11 Positionierung eines Roboters	21
4.3.1 Modus <i>full</i> und <i>none</i>	22