DHBW
Duale Hochschule
Baden-Württemberg
Mannheim

Studiengang Informationstechnik

Studienarbeit II

Verbesserung der Taktischen Spielfeldanalyse - Analyse / Auslagerung von Berechnungen auf die GPU

von:

Philipp Posovszky

Matrikelnummer: 2028068

Kurs:	TIT 10 A NS
Bearbeitungszeitraum:	25.03.2013 – 21.06.2013
Ausbildungsfirma:	Deutsches Zentrum für Luft und Raumfahrt e.V
Betreuer:	Prof. Dr. Jochem Poller

Erklärung:

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig und nur unter Benutzung angegebener Quellen und Hilfsmittel angefertigt habe.

Mannheim, den 12.06.2013 _____

Unterschrift des Verfassers

Inhaltsverzeichnis

Abkürzungsverzeichnis

CPU	-	**C**entral **P**rocessing **U**nit
CUDA	-	**C**ompute **U**nified **D**evice **A**rchitecture
FIRA	-	**F**ederation of **I**nternational **R**obot-Soccer **A**ssociation
ISO	-	**I**nternational **O**rganization for **S**tandardization
GPU	-	**G**raphics **P**rocessing **U**nit
TIGERS	-	**T**eam **I**nteracting and **G**ame **E**volving **R**obot**S**
OpenCL	-	**O**pen **C**omputing **L**anguage
OpenGL	-	**O**pen **G**raphics **L**ibrary

Abbildungsverzeichnis

Abstract

Diese Studienarbeit befasst sich mit dem Thema der Analyse eines Spielfeldes des Robocup in der Small Size League. Dabei war das Ziel, die aktuelle Methoden zur Spielfeldanalyse in der Software „Sumatra" aus Performancetechnischer und Analytischer Sicht zu verbessern. Dazu wurde die aktuelle Analyse auf die Grafikkarte ausgelagert um eine besser Performance zu erzielen. Desweitern wurde die Distanzfunktion, welche nach der Euklidischen Distanz berechnet wird, im alten Analyse-Verfahren durch die Mahlanobis-Distanz ersetzt. Dies ermöglicht es einen Miteinbezug der Geschwindigkeit eines einzelnen Roboters in das Analyseverfahren. Zusätzlich wurde noch für den Play-Entwickler einer Klasse implementiert, welche es ermöglicht Informationen aus dem Analyse-Feld abzuleiten.

1. Einleitung

1.1. Roboter Fußball

Fußball ist eine sehr beliebte Sportart in jedem Land dieser Erde. Es ist ein sehr dynamischer Wettkampf welcher schnell Entscheidungen des Einzelnen fordert, sowie Teamwork und verschiedene Strategien, welche sich immer an den Gegner anpassen müssen. Warum diesen beliebten Wettbewerb also nicht mit Roboter durchführen? Ein dynamischer Wettkampf, mit schnellen Entscheidungen und unterschiedlichen Strategien ist perfekt um KI- und Robotersysteme zu entwickeln. Dazu kommt dann noch die vorhanden Faszination am Wettkampf zwischen Mensch und Maschine. Das war auch einer der Gründungsgedanken des RoboCup:

> *„bis zum Jahr 2050 ein Team von autonomen, humanoiden Robotern zu entwickeln, das in der Lage ist, den zu diesem Zeitpunkt amtierenden menschlichen Fußballweltmeister schlagen zu können" [1]*

Der RoboCup und die Weltmeisterschaft der Federation of International Robot-Soccer Association sind die wichtigsten internationalen Wettkämpfe im Roboterfußball. Dabei werden die Roboter in sechs verschiedenen Ligen unterteilt, sowie zwei Ligen die sich nicht mit Fußball beschäftigen. Die Fußballigen unterteilt sich der Größe nach (dazu zählen Beispielsweise humanoide Roboter verschiedener Größe), sowie kleinen fahrbaren Robotern der Small Size League (siehe Abbildung 1) in der die TIGERS Mannheim agieren.

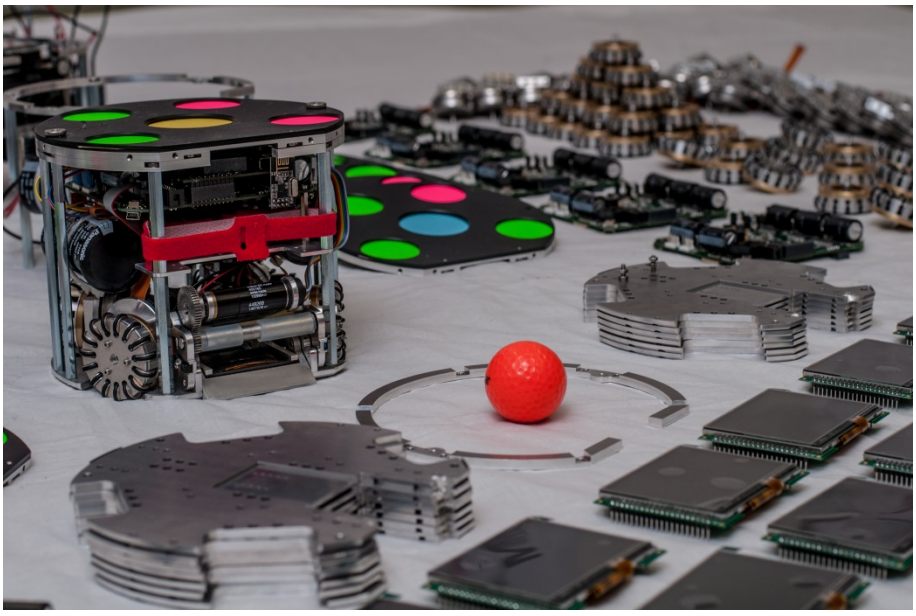

Abbildung 1: Neue Robotergeneration der Small Size League der TIGERS Mannheim in Einzelteilen

1.2. Das Team Tigers Mannheim

Die Tigers (**T**eam **I**nteracting and **G**ame **E**volving **R**obot**S**) der DHBW Mannheim sind ein Team, welches sich beim Robocup beteiligt. Die Tigers existieren seit dem Jahr 2008 und haben 2011 zum aller ersten Mal an der Roboterfußballweltmeisterschaft in der Small Size League teilgenommen. Dabei konnten schon erste Erfolge erzielt werden, im Jahre 2012 bei der Weltmeisterschaft in Mexico, haben wir nur ein Spiel 1:0 verloren und sind damit ganz knapp in der Vorrunde ausgeschieden. Im Jahr 2013 wird die WM in Eindhoven (Niederlande) ausgetragen. Dabei kommen neue Roboter, sowie eine verbesserte Software zum Einsatz.

1.3. Ziel und Anforderungen

Die WM in Mexico hat gezeigt, dass unsere Roboterplattform sehr gut funktioniert und unsere Verteidigungsstrategien eine gute Deckung des Tors erreicht haben. Jedoch waren wir nicht in der Lage genug Tore zu erzielen, was unteranderem auf Pass Ungenauigkeiten oder Wegfindungsproblemen zurück zu führen ist. Die neuen Roboter werden diese Ungenauigkeiten unteranderem durch ein verbessertes Reglungssystem ausgleichen. In der Studienarbeit „Taktische Spielfeldanalyse im Robocup mittels Rasterung des Spielfelds" von Oliver Steinbrecher und Paul Birkenkampf, wurde ein System in der Zentralsoftware Sumatra implementiert, welches es erlaubt eine Spielfeldanalyse durchzuführen.

> *„Die so gewonnen Daten sollen dann als Grundlage zur Planung und Ausführung von offensiven Spielzügen verwendet werden können. Um diese Analyse durchzuführen soll das Spielfeld gerastet werden, um somit klar definierte Bereiche zu erhalten. Den so entstanden Rechtecken soll ein Wert hinzugefügt werden, der besagt wie gut oder schlecht das Rechteck für das eigene Team ist (im weiteren Güte genannt)."[1]*

Jedoch wird das Ergebnis aus dieser Spielfeldanalyse nicht in Sumatra zur Planung von Spielzügen verwendet, da keine passende Schnittstelle bereit steht, welches konkrete Informationen aus der Güte der einzelnen Rechteecke ableitet. Das Ziel der Studienarbeit ist es nun eine solche passende Schnittstelle in Sumatra zu implementieren. Dazu wird das Spielfeldraster in einer weiten Klasse gekapselt, welche dann innerhalb eines Spielzuges abgefragt werden kann. Weiter wird die Spielfeldanalyse momentan von der CPU durchgeführt. Somit sind momentan nur Rasterungsgrößen des Spielfeldes von 4x4 bis zu 64x128 möglich, bei größeren Werten ist der Performance Einbruch in der Zentral Software „Sumatra" zu groß. Deshalb soll im Zuge der Studienarbeit im ersten Schritt, die von Oliver Steinbrecher und Paul Birkenkampf implementierten Algorithmen auf die GPU ausgelagert und verbessert werden.

2. Analyse der aktuellen Umsetzung und Konzepte zu Verbesserung

In der aktuellen Sumatra Version wird das Spielfeld auf dem sich die Roboter bewegen in mxn Rechtecke aufgeteilt. Für jedes dieser Rechtecke wird eine Güte anhand der aktuellen Roboterpositionen berechnet. Im folgenden Abschnitt wird das aktuelle implementierte Verfahren zur Güte Berechnung erklärt und Analysiert. Auf Grund dieser Analyse werden dann Verfahren entwickelt um den Nutzen der **FieldRasterAnalyse** zu erhöhen.

2.1. Analyse des Taktischen Feldrasters

Die Güte für jedes Rechteck der Matrix mxn wird eine Güte Berechnet, welche angibt ob der Gegner das aktuelle Feld „in Besitz" hat oder wir. Im Besitz bedeutet dabei, dass der Bot welcher am nächsten steht schneller dieses Feld erreichen kann, als einer der unserer Roboter. Die daraus resultierende Formel für die Güte G eines Rechteckes ergibt sich aus der Euklidische-Distanz d zwischen dem nächsten TigersBot $tigersBot$ und gegnerischen Bot $foeBot$:

$$G = (d_{tigersBot} - d_{foeBot})$$

Anhand dieser Formel wird dann die Güte für jedes Feld berechnet, dazu wird zuerst ermittelt welcher der Bots des jeweiligen Teams am nächsten steht und daraufhin wird die Güte Formel angewandt. Aus Performance Gründen ist es in der aktuellen Version unmöglich eine Güte für eine Größere Matrix als $12x24$ Felder zu berechnen. Dies soll später durch eine Unterstützung durch die Grafikkarte verbessert werden.

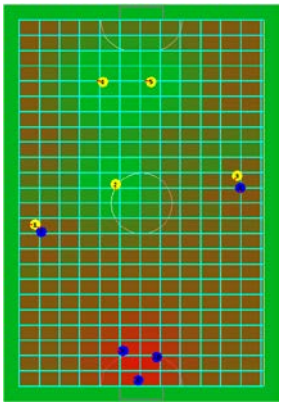

Abbildung 2: Alte Implementierung der FieldRaster Analyse [12x24 Felder]

Problematisch zur Auswertung der Analyse ist, die Güte Werte zu stark beschränkt sind. Das Intervall für die Güte geht von $[-100; 100]$ wobei -100 bedeutet, dass der Punkt für uns Eingenommen ist. Durch diesen Punkt geht jedoch die Unterscheidung von Punkten mit einer hohen oder geringen Güte verloren. In der Abbildung 2 ist die Güte für das Rechteck unter dem gelben Roboter mit der ID 2 identisch mit dem Rechteck zwischen den Robotern 4 und 5. Für einen Algorithmus ist es nun schwer den am Punkt mit der besten Güte zu finden, da es möglicherweise mehrere gibt.

Desweitern ist es anhand der Farbwahl auf der GUI schwer abzuschätzen, welcher Punkt nun Neutral ist. Das heißt vom Gegner genauso schnell angefahren werden kann, wie von uns. Theoretisch sind dies die Felder die Rot eingefärbt sind mit einer Transparenz von 50%. Das heißt für das Menschliche Auge quasi nicht auszumachen.

Die Idee auf der die Güte Formel basiert ist sehr gut, jedoch ist die Güte Berechnung zu statisch. Sie Berücksichtigt zwar die Position und wird oft genug Berechnet um die Spielsituation dynamisch wieder zu geben, jedoch Berücksichtig sie nicht die Bewegungsrichtung des Roboters. Die Entfernung von einem 1 m Punkt orthogonal zu Bewegungsrichtung wird genauso bewertet wie ein Punkt 1 m Entfernt in Bewegungsrichtung. Jedoch muss der Bot erst abbremsen und wieder in die andere Richtung bewegen, wodurch der Punkt 1m orthogonal zur Bewegungsrichtung unter dem Zeitaspekt wesentlich weiter weg ist.

Momentan reagiert nur der **PlayFinder** auf Veränderung der Güte Verteilung. Dabei ist das Problem für diesen, dass falls nur in der gegnerischen Hälfte gespielt wird und sich unser Keeper viel bewegt, variiert die Güte sehr stark.

2.2. Schnittstelle zur Datenextraktion aus dem FieldRaster

Zur sinnvollen Ableitung von Informationen aus dem **FieldRaster** werden die eigentlich berechneten Rechtecke in einer Klasse **EnhancedFieldRaster** gekapselt.

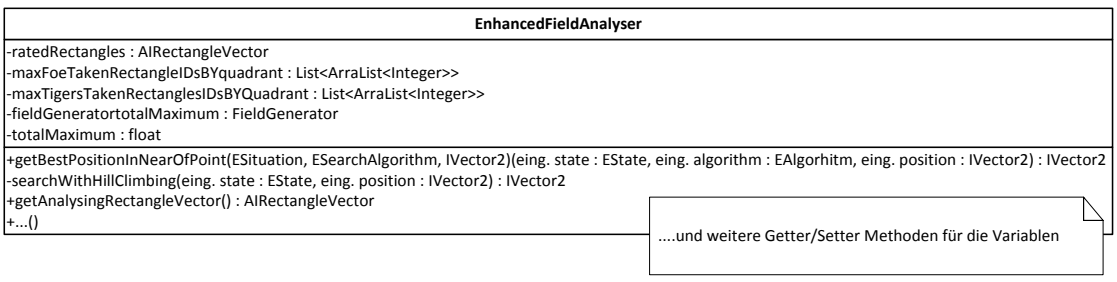

Abbildung 3.: Klassendiagramm für die EnhancedFieldAnalyser-Klasse

In dieser Klasse stehen mehrere Methoden zum Informationsgewinn bereit. Zum einen eine Methode *getMaxFoeRectanglesIDs()* (Foe durch Tigers für günstige Rechtecke ersetzen) welche alle IDs der günstigsten oder ungünstigen Rechtecke zurückgibt, diese können auch für jeden Quadranten einzeln zurückgegeben werden, bei mit übergeben der Quadranten Nummer als Parameter *getMaxFoeRectanglesIDs(1).* Eine Methode *getMaxRectangleValue()* welche den betragsmäßig höchsten Wert für den Visualizer zurückgibt, womit dieser die Farbgebung der zu zeichnenden Rechtecke skalieren kann. Als wichtigstes gibt es eine Methode *searchNextOptimalField(EAlgorithm, EState, limit)* wonach nach einem *EState.Free* oder *EState.Taken* Rechteck gesucht werden kann. Mit EAlgorithm kann zwischen verschiedenen Algorithmen ausgewählt werden, momentan ist nur ein einfacher Hill-Climbing-Algorithmus implementiert, der solange dem Gradienten Anstieg folgt, bis er das Optimalste Feld erreicht hat. Das *limit* gibt dabei an wie oft der Algorithmus das Feld weiter gehen kann. Dies soll dem Play-Entwickler ermöglichen nur in einem bestimmten Radius nach dem Optimum zu suchen. In ungünstigen Fällen könnte sonst ein Feld auf der ganz anderen Seite des Spielfeldes gewählt werden, da ein stetiger Gradienten Anstieg möglich ist. Zukünftig sollen weitere Algorithmen implementiert und getestet werden.

2.3. Verbesserungen des Algorithmus

Zur Verbesserung des Algorithmus wird zuerst die Visualisierungsproblematik angepasst. Dazu wird nichtmehr zwischen Rot mit und Transparenz unterschieden, sondern jedes Team bekommt eine eigene Farbe (Rot für TigersBots und Blau für Gegnerische Bots). Neutral ist zukünftig der Bereich, an dem keine Farbe eingezeichnet wird bzw. der die Güte 0 besitzt. Bei der Visualisierung werden jedoch auch alle Werte Oberhalb einer bestimmten Grenze abgeschnitten, im Hintergrund bleiben diese aber auch für die Software erhalten. Somit sorgt ein Rechteck mit einem extremen Maximum nicht dazu, dass alle Felder im Vergleich zu diesem gegen Null laufen und somit Transparent eingezeichnet werden.

Die Formel zu Güte Berechnung wird insofern angepasst, dass die Distanzrechnung nicht mehr die Euklidische-Distanz benutz wird. Anstatt dessen wird die Mahalanobis-Distanz verwendet. Im Gegensatz zu der Euklidische-Distanz bildet diese projiziert auf einen 2Dimensionalen Raum keinen Kreis, sondern eine Ellipse. Wird diese Ellipse nun an dem Bewegungsvektor eines Roboters ausgerichtet, ergibt sich der gewünscht Effekt, dass Punkt orthogonal zur Bewegungsrichtung weiter weg sind als Punkt in Bewegungsrichtung.

Die Mahalanbois-Distanz berechnet sich anhand des Ortsvektors des Rechteckes $\vec{x}$ und des Ortsvektors des Roboters $\vec{y}$ wie folgt:

$$d(\vec{x},\vec{y}) = \sqrt{(\vec{x}-\vec{y})^T S^{-1}(\vec{x}-\vec{y})}$$

Um die globalen Ortsvektoren in das lokale Bezugssystem des Richtungsvektors $\vec{v}$ zu bringen, müssen diese noch mit einer Homogenen Matrix verschoben und rotiert werden.

$$transAndRot(\alpha, t_x, t_y) = \begin{pmatrix} \cos\alpha & -\sin\alpha & t_x \\ \sin\alpha & \cos\alpha & t_y \\ 0 & 0 & 1 \end{pmatrix}$$

Der Winkel α wird dabei aus dem Skalar Produkt des Geschwindigkeitsvektors gebildet und die Translation t_x, t_y wird aus dem Ursprungspunkt des Geschwindigkeitsvektors ermittelt. Wichtig ist nun noch S die Kovarianz Matrix. Mit der Kovarianz Matrix lässt sich die Form der Ellipse bestimmen, für große Geschwindigkeiten soll diese schmaler und länger werden. Deshalb wird eine Abhängigkeit zur Geschwindigkeit gewählt:

$$S = \begin{pmatrix} 10 + vS & 0 \\ 0 & 10 - vS \end{pmatrix}$$

Der Wert 10 wurde dabei empirische ermittelt und gibt die grundlegende Ausbreitung der Ellipse an. Im Fall das die Kovarianz Matrix eine Einheitsmatrix darstellt, entspricht die Mahalanobis-Distanz der Euklidsichen-Distanz. Was bei dieser Wahl der Matrix S für den Fall das die Geschwindigkeit $vS = 0$ gegeben ist. Falls ein Matrix-Element <=1 wird, wird es auf 1 gesetzt. [3]

Als Nachteil der Berechnungen mit der Mahalanobis-Distanz resultiert allerdings, dass Punkt in negativer Bewegungsrichtung immer noch gleichwertig zu Punkten in Positiver Bewegungsrichtung sind, dabei sollten diese theoretisch noch schlechter bewertet werden als seitliche Punkte.

Eine Leistungssteigerung wird durch eine Auslagerung des Algorithmus auf die Grafikkarte erreicht, welche im nächsten Kapitel erläutert wird.

3. Auslagerung von Berechnungen auf eine GPU

Grafikkarten wurden bis jetzt immer für Berechnung von Grafiken verwendet, unteranderem von Spieleszenen in Computerspielen. In den letzten Jahren hat sich aber eine neue Anwendungsmöglichkeit für Grafikkarrten erschlossen, das Parallelrechnen. Ermöglicht wird dies durch diverse Schnittstellen die es erlauben Programmcode direkt auf der Grafikkarte auszuführen. Dazu zählt die Schnittstelle CUDA die es ermöglicht auf NVIDIA Grafikkarten zu rechnen, oder OpenGL, OpenCL etc. Für das Projekte

Sumatra und die das Problem, dass man mit CUDA an NVIDIA Grafikkarten gebunden ist. Nur OpenCL und OpenGL infrage. OpenCL arbeitet auch Problemlos auf verschiedenen Betriebssystemen und auch auf der CPU selbst (falls kein OpenCL-Gerät bereitsteht). Für OpenCL gibt es eine Java Bibliothek, die alle Funktionen von OpenCL integriert hat. Somit muss auch keine Schnittstelle zu einer anderen Programmiersprache implementiert werden.

Zur Leistungssteigerung unserer Software, sollen schrittweiße geeignete Berechnungen auf die GPU ausgelagert werden. Als erster Algorithmus hat sich dazu die **FieldRasterAnalyse** angeboten, da diese für eine große Matrix viele Berechnungen tätigt. Wichtig dabei ist, dass diese Berechnungen der einzelnen Elemente unabhängig voneinander Ausgeführt werden könne.

3.1. Parallelisierung der Algorithmen

Zur Parallelisierung eines Algorithmus muss dieser in die Programmiersprache OpenCL C portiert werden. OpenCL C basiert dabei auf der Syntax ISO C99 hat aber ein paar Einschränkungen, z.B. ist es nicht möglich mehrdimensionale Arrays primitiver Datentypen an einen OpenCL Kernel zu übergeben. Der Kernel selbst hat keinen Rückgabewert, man muss bei der Initialisierung ein Pointer mitgeben der auf den Speicherplatz des Zielarrays verweist. Rekursionen sind nicht möglich, Pointer auf Funktionen sind nicht erlaubt, sowie ist die Länge von Arrays nicht variabel. Vorsichtig sollte man zudem mit dem schreiben in lokalen Speicher sein. Innerhalb des Kernels werden verschiedene Speicherbereiche unterschieden. Es existiert der globale Speicher auf den jede Instanz des Kernels zugreifen kann, der konstante Speicher ist für nichtveränderbare Werte vorgesehen. Der lokale Speicher gilt immer für kleine Arbeitsgruppen von Kernelinstanzen. Als letztes gibt es noch den privaten Speicher, der ausschließlich für eine Kernelinstanz gilt. Um Probleme beim schreiben und synchronisieren von lokalen Speicher zu vermeiden, wird ausschließlich privater und globaler Speicher verwendet.

Falls der in Java geschrieben Algorithmus nur mit primitiven Datentypen arbeitet, kann dieser nahezu 1 zu 1 auf die GPU ausgelagert werden. Falls dies nicht der Fall ist, müssen die Java-Objekte auf Arrays aus primitiven Datentypen umgewandelt werden. Diese Umwandlung kann unter Umständen dafür sorgen, dass es ineffizient wird den Algorithmus auf die Grafikkarte auszulagern. Nach der Umwandlung muss man nur noch dafür Sorge tragen, dass in dem OpenCL C-Code jede Kernel-Instanz nur ein Teil berechnet, welcher unabhängig vom Rest ist. Ein Beispiel für eine einfache unabhängige Berechnung ist die Vektoraddition.

Für die Berechnung des **FieldRasters** nach der alten und neuen Methode musste zuerst einmal die Umwandlung getätigt werden. Dazu wurden z.B. die Roboterpositionen nicht als Vector2f übergeben, sondern alle Positionen zusammen in ein Array geschrieben. Dabei beschreiben immer zwei folgende

Wertepaare einen Vektor. Auch die Matrix der Rechtecke wird statt als 2 Dimensionales Arrays als 1 Dimensionales Array übergeben und innerhalb des Kernels dem entsprechen interpretiert. Im Anhang II.a befindet sich der parallele Algorithmus zu der Güteberechnung für das alte **FieldRaster**.

Als Grenzen für die Implementierung des Algorithmus auf der GPU wurde anhand von Test mit einer NVIDIA Geforce GT 555M eine maximale Größe des Rasters von 64x128 Feldern ermittelt. Bei höheren Rastergrößen ist eine schnellere oder mehrere Grafikkarte notwendig. Jedoch ist eine größere Aufteilung fast nicht sinnvoll, da die Rechtecke dann wesentlich kleiner werden als die Roboter selbst es sind. Mit 64 x 128 Feldern steht ein einzelner Roboter schon auf Minimum 4 Rechtecken gleichzeitig.

Da der **PlayFinder** mit dem neuen größeren Spielfeld ein zu großen Performanceeinbruch erleidet. Denn dieser berechnet sich aus seiner Datenbank mithilfe eines eigenen **FieldAnalyser** die Spielfelder die über die Zeit angefallen sind, regelmäßig neu. Dieser benötigt zudem auch nicht so ein großes Spielfeld. Die Lösung ist es, einen separaten **FieldGenerator** zu implementieren, welcher ein kleineres Analysing-Feld ermöglicht. Dazu wurde der **FieldGenerator** von seiner jetzigen Implementierung als Singelton gelöst und kann nun öfters Instanziiert werden, um genau zu sein einmal für *Metis* und einmal für den *PlayFinder*.

4. Fazit und Ausblick

Im Laufe der Studienarbeit wurden die Analysefähigkeiten der Software Sumatra verbessert. Dazu wurde zum einen die Performance durch eine Auslagerung der Algorithmen auf die Grafikkarte erhöht, siehe Abbildung 3. Zum anderen wurde ein Algorithmus implementiert, welcher mit der Mahalanobis-Distanz arbeite. Durch diesen Algorithmus ist es nun möglich die Bewegung des Roboters besser zu analysieren, da dieser zusätzlich zur Position noch die Geschwindigkeit des Roboters einbezieht. Dieser Algorithmus legt auch den Grundstein für eine Zukunftsanalyse. Anhand der Geschwindigkeit und der Richtung soll zukünftig mit einbezogen werden an welcher Stelle sich der Gegner in 1 bis 2 Sekunden befindet, bzw. das Analyse-Feld dann in diese Richtung verlagert.

Anhand der neuen Schnittstellen Klasse zum **FieldRaster**, können nun besser Informationen aus dem **FieldRaster** abgeleitet werden. So werden die Punkte mit der größten Gegnerdeckung, geringsten Gegnerdeckung und vieles weiter angegeben und kann innerhalb von Spielzügen verwendet werden.

Durch die Verbesserungen im **FieldRaster** kann das Spielfeld nun dynamischer Analysiert werden und ermöglicht es den Playentwicklern auf mehr Informationen über das aktuelle Geschehen zuzugreifen. Theoretisch ist eine noch detaillierte Berechnung des Spielfeldes möglich, jedoch hat diesen kein

größerer Nutzen. Aktuell sind die Rechtecke schon so klein, das ein Roboter immer auf 3-4 Rechtecken gleichzeitig steht.

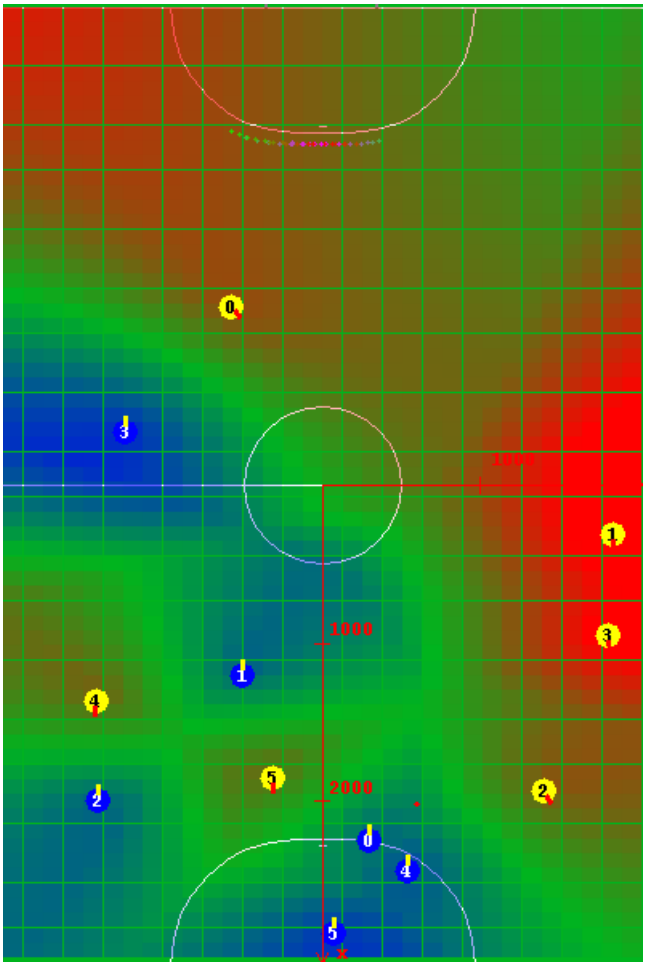

Abbildung 4: Güte Berechnung auf der Graifkkarte [32x64 Felder]

I. Quellenverzeichnis

1. ELEKTRONIKPRAXIS: *Bremer Studenten wurden Europameister bei den RoboCup German Open 2009.* http://www.elektronikpraxis.vogel.de/index.cfm?pid=4800&pk=185583&sector=2 Abruf: 25.05.2013

2. Oliver Steinbrecher, Paul Birkenkampf, Studienarbeit, Taktische Spielfeldanalyse im Robocup mittels Rasterung des Spielfelds, 2012

3. P.C. Mahalanobis: On the generalised distance in statistics. In: Proceedings of the National Institute of Science of India. Vol. 2, Nr. 1, 1936

II. Anhang

a. Programmcode – StandardFieldRasterAnalyse in OpenCL

```
__kernel void fieldRaster_v1(__global const int *botPos,
                    __global const int *parameter,
                    __global float *goal)
        {

         /* Parameter Beschreibung:
        paramet[0]=m (rows)
        parameter[1]=n (colums)
        parameter[2]=k (Laenge des Arrays von botposition)
        parameter[3]=botOurCount (Laenge des Arrays von botposition)
        botPos[n]: gibt die ID des Arrays an in dem ein Bot steht
     goal: ist das Ziel-Array welches parallel erarbeitet werden soll, jedes
     work-item berechnet 1-stelle*/

            int gid = get_global_id(0);
              goal[gid]=0.0f;

            int m=parameter[0];
            int n=parameter[1];
            int k=parameter[2];
            int botOurCount=parameter[3];

            float shortestTiger = MAXFLOAT;
              float shortestFoe = MAXFLOAT;
            for(int i=0;i<k;++i)
            {

              /* ID des Rechtecks */
                      int recID = botPos[i];
                      /* m koordinate des Rechtecks */
                      int rec_n = 0;
                      /* n koordinate des Rechteks */
                      int rec_m = recID;
                      if (recID >= n)
                      {
                              rec_n = recID / n;
                              rec_m = recID - (rec_n * n);
                      }
                      /* m koordinate des goalArray */
                      /* n koordinate des goalArray */
                      int goalArray_n = 0;
                      int goalArray_m = gid;
                      if (gid >= n)
                      {
                              goalArray_n = gid / n;
                              goalArray_m = gid - (goalArray_n * n);
                      }

               int x = rec_m - goalArray_m;
                      int y = rec_n - goalArray_n;

             /* Berechnen des Abstandes von goalArray und recPos*/
             float value=(sqrt((float)(x*x+y*y)));

                      if (i < botOurCount)
```

```
                        {
                                if (value < shortestTiger)
                                {
                                        shortestTiger = value;
                                }
                        } else
                        {
                                if (value < shortestFoe)
                                {
                                        shortestFoe = value;
                                }
                        }
                goal[gid] = shortestFoe - shortestTiger;
                }
};
```

b. Programmcode Mahlanobis-Distanz in OpenCL

```
__kernel void fieldRaster_v3(__global const int *botPos,
                             __global const float *botSpeed,
                __global const int *parameter,
                __global float *dstArray)
   {

      /* Parameter Beschreibung:
   parameter[0]=m (rows)
   parameter[1]=n (colums)
   parameter[2]=k (Laenge des Arrays von botposition)
   parameter[2]=ourBot  (wie viele der übgerbeen bots gehören uns)
   botPos[n]: gibt die ID des Arrays an in dem ein Bot steht

   goal: ist das Ziel-Array welches parallel erarbeitet werden soll
   */

       int gid = get_global_id(0);
      dstArray[gid]=0.0;

       int m=parameter[0];
       int n=parameter[1];
       int k=parameter[2];
       int ourBot=parameter[3];

         float shortestTiger = MAXFLOAT;
         float shortestFoe = MAXFLOAT;
         for (int i = 0; i < 12; i++)
         {
                 float goalArray_n = 0;
                 float goalArray_m = gid;
                 if (gid >= n)
                 {
                         goalArray_n = gid / n;
                         goalArray_m = gid - (goalArray_n * n);
                 }

                 // Geschwindigkeitsvektordes Roboters
                 float vX = botSpeed[2 * i];
                 float vY = botSpeed[(2 * i) + 1];
                 float vS = (float) sqrt((vX * vX) + (vY * vY));

                 /* ID des Rechtecks in dem der Bot steht */
                 /* n koordinate des Rechteks */
                 /* m koordinate des Rechtecks */
                 int recID = botPos[i];
```

```
		float rec_n = 0;
		float rec_m = recID;
		if (recID >= n)
		{
			rec_n = recID / n;
			rec_m = recID - (rec_n * n);
		}

		if (vS > 0.0001)
		{
		 float a = (float) acos(vX / (sqrt((vX * vX) + (vY *
		 vY))));
// Fallunterscheidung ob Negativ oder Positv Rotiertwerden muss
			if (vY > 0)
			{
				a = -a;
			}
			float rotMatrix[9];
			rotMatrix[0]=(float) cos(a);
			rotMatrix[1]=(float) -sin(a);
			rotMatrix[2]=-rec_n;
			rotMatrix[3]=(float) sin(a);
			rotMatrix[4]=(float) cos(a);
			rotMatrix[5]=-rec_m;
			rotMatrix[6]=0;
			rotMatrix[7]=0;
			rotMatrix[8]=1;

			float temp = rec_n;
			rec_n = (rotMatrix[0] * rec_n) + (rotMatrix[1] *
			rec_m) + (rotMatrix[2] * 1);
			rec_m = (rotMatrix[3] * temp) + (rotMatrix[4] *
			rec_m) + (rotMatrix[5] * 1);
			temp = goalArray_n;
			goalArray_n = (rotMatrix[0] * goalArray_n) +
			(rotMatrix[1] * goalArray_m) + (rotMatrix[2] *
			1);
			goalArray_m = (rotMatrix[3] * temp) +
			(rotMatrix[4] * goalArray_m) + (rotMatrix[5] *
			1);
		}
		// --------

		if (vS <= 0.0001)
		{
			vS = 0.0001f;
		}

		float s_neg[4];
		s_neg[0]=10 + (10 * vS);
		s_neg[1]=0;
		s_neg[2]=0;
		s_neg[3]=10 - (10 * vS);
		if(s_neg[3]<=1)
		 s_neg[3]=1;

		float rVec[2];
		rVec[0]=rec_n - goalArray_n;
		rVec[1]=rec_m - goalArray_m;

		float x = (rVec[0] * s_neg[0]) + (rVec[1] * s_neg[2]);
		float y = (rVec[0] * s_neg[1]) + (rVec[1] * s_neg[3]);

		float temp = (float) sqrt((x * rVec[0]) + (y * rVec[1]));
```

```
                        if (i < ourBot)
                        {
                                if (temp < shortestTiger)
                                {
                                        shortestTiger = temp;
                                }
                        } else
                        {
                                if (temp < shortestFoe)
                                {
                                        shortestFoe = temp;
                                }
                        }
        }
        dstArray[gid] = shortestFoe - shortestTiger;
};
```