Studienarbeit

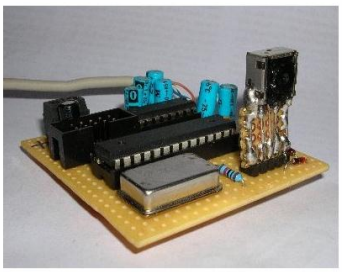
Headtracking mit Wii-Cam

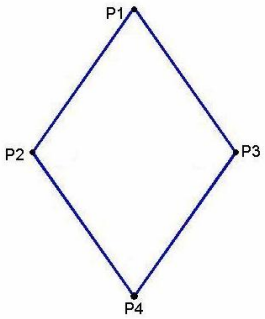
Eva Ellwardt, 203210028

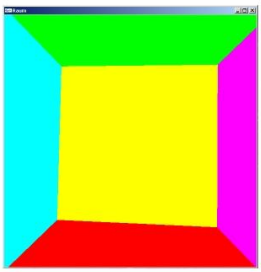
Universität Koblenz-Landau

ellwardt@uni-koblenz.de

Betreuer: Dr. Merten Joost







Inhaltsverzeichnis

1 Einleitung

Wenn man vor einem Computerbildschirm sitzt, sieht man normalerweise ein starres oder vorgefertigtes bewegtes Bild. Manchmal kann man das Bild mit Hilfe von Eingabegeräten wie Maus oder Tastatur selbst bewegen. Schön wäre es jedoch, wenn man sich einfach, indem man den Kopf bewegt, umsehen könnte. Man könnte sich dann realitätsnah in der virtuellen Welt umblicken. Das ist Inhalt dieser Studienarbeit.

1.1 Motivation

Wenn man sich im realen Raum umsieht, verändert sich das Sichtfeld bei jeder kleinsten Drehung. Bewegt man seinen Kopf vor einem Bildschirm, bleibt dieser starr. Sicher wäre es interessant sich auf dem Bildschirm so umgucken zu können wie in einem realen Raum auch.

Dieser Effekt wirkt am besten mit einem augennahen Bildschirm, Head-Mounted-Display genannt. So verliert man den Bildschirm nicht aus den Augen während man sich dreht. Normalerweise bleibt das Bild des Bildschirmes dasselbe, wenn man seinen Kopf bewegt. Nun wird sich auch das Bildschirmbild verändern, wenn man sich umschaut. Man braucht keine Maus mehr bewegen oder Tasten drücken. Man kann sich wirklichkeitsnah, mit dem Kopf, umsehen. Bei Fahr- und Flugsimulationen kann dieses Verfahren sehr hilfreich sein. Man kann sich beispielsweise umschauen oder frühzeitig Kurven einsehen, was zu

Abb. 1: Head-Mounted-Display[dis]

einer sehr realitätsnahen Ansicht führt.

1.2 Aufgabenstellung

Ziel dieser Studienarbeit ist es, die Bewegungen eines Kopfes zu erfassen und im virtuellen Raum korrekt wiederzugeben. Es soll also möglich sein, sich im virtuellen Raum genauso wie im realen Raum, „umschauen“ zu können. Für die Realisierung werden die Infrarotkamera aus der Wii-Fernbedienung[1] und Infrarot-LEDs verwendet. Die Infrarot-LEDs werden so am Kopf befestigt, dass sie im Blickfeld der Kamera liegen. Durch die Verschiebung der Punkte beim Drehen des Kopfes kann die Position des Kopfes ermittelt werden. Die Positionen (in x- und y-Werten) der Infrarot-LEDs können in Echtzeit ausgelesen werden. In einem kleinen OpenGL-Programm soll die Funktion des Projektes demonstriert werden.

[1] http://nintendo.de/NOE/de_DE/systems/technische_details_1072.html

2 Grundlagen

Seit 2006 gibt es die Videospiele-Heimkonsole Wii[2]. Hergestellt wurde sie von der Firma Nintendo. Das neue an der Wii ist der einer Fernbedienung ähnliche Controller (auch Wiimote beziehungsweise Wii-Fernbedienung genannt). Die Wii kann die Lage ihres Controllers im Raum feststellen, wodurch völlig neue Spielmöglichkeiten gegeben sind. Bei der Frage wie das genau funktioniert muss man sich die Wii-Fernbedienung einmal genauer anschauen.

Abb. 2: Nintendo Wiimote[wii]

[2]http://www.nintendo.de/NOE/de_DE/systems/ueber_wii_1069.html

2.1 Wii-Cam

In der Fernbedienung der Spielekonsole Wii sitzt eine kleine Infrarotkamera. Diese Kamera wird mit Hilfe der Sensorbar[3] dazu verwendet die Bewegung der Fernbedienung zu erfassen. So kann festgestellt werden, wo sich die Wii-Fernbedienung im Raum befindet. Für das Headtrackingprojekt wird die Infrarotkamera aus der Wii-Fernbedienung ausgebaut, weil sie als einziges Bauteil in diesem Projekt verwendet wird. Der horizontale Öffnungswinkel der Wii-Cam beträgt ungefähr 40 Grad, der vertikale etwa 30 Grad[4]. Das Kameramodul ist in der Lage, bis zu vier verschiedene Leuchtpunkte zu erfassen und deren Position zu verfolgen. Dieses „Tracking“ der Leuchtpunkte wird also vollständig von der Kamera übernommen, weswegen man sich darum nicht mehr kümmern muss. Die Leuchtpunkte dürfen prinzipiell aus einem breiten Wellenlängen-Spektrum sein. Hier (und auch in der Wii-Konsole) wird aber nur der für den Menschen nicht sichtbare Infrarotlichtbereich genutzt, was verschiedene Vorteile bietet, worauf später noch eingegangen wird.

[3] http://www.nintendo.com/wii/what/controllers#sensorbar

[4] http://www.wiimoteproject.com/wiimote-whiteboard/wiimote-camera-angles/

3 Cam-Platine

3.1 Hardware

Die in der Wiimote vorhandene Infrarot-Kamera muss, abgesehen vom Auslöten aus dem Gerät, auch noch in irgendeiner Form an einen Rechner angebunden werden. Aus der Wiimote wird noch der IR-Filter des Kameramoduls übernommen, ansonsten keine weiteren Bauteile.

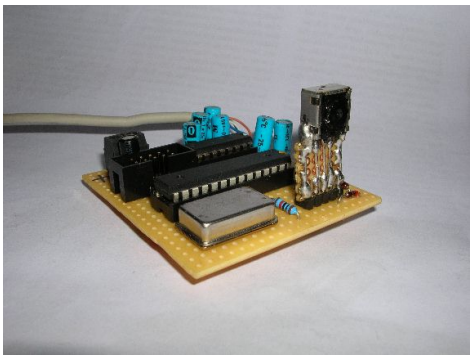


Abb. 3: Cam-Platine (ohne IR-Filter)

Die Schnittstelle der Kamera ist aber leider nicht direkt mit denen aus „gebräuchlichen“ Rechnern kompatibel, weswegen ein Adapter realisiert werden muss, um die Schnittstellen entsprechend anzupassen. Diese Arbeit wird von einem Mikrocontroller der Firma Atmel[5] erledigt. Genauer gesagt handelt es sich um einen ATmega8, ein Modell aus der AVR 8-Bit RISC Reihe. Dieser Mikrocontroller verfügt unter anderem über die gleiche Schnittstelle wie das Kameramodul, nämlich um eine I^2C-kompatible (von Atmel TWI[6] genannt). Bei der Schnittstelle für die Rechnerseite fiel die Wahl auf eine serielle RS232, da diese (obwohl schon etwas älter) noch an vielen Rechnern vorhanden ist, oder zumindest über USB-Adapter verfügbar gemacht werden kann. Ein weiterer Grund ist die Einfachheit und Robustheit dieser Schnittstelle, und dass im verwendeten Mikrocontroller ebenfalls eine serielle Schnittstelle vorhanden ist. In diesem Zusammenhang wichtig ist, dass die Schnittstellen RS232 (des Rechners) und die serielle UART

[5] http://www.atmel.com
[6] Two Wire Interface

(des Mikrocontrollers) zwar auf logischer Ebene kompatibel sind, sie aber andere Spannungslevel zur Darstellung von High- und Low-Pegeln verwenden. Um dieses Problem zu beseitigen, sitzt auf der Platine ein eigens dafür zuständiges Pegelkonverter-IC (MAX232).

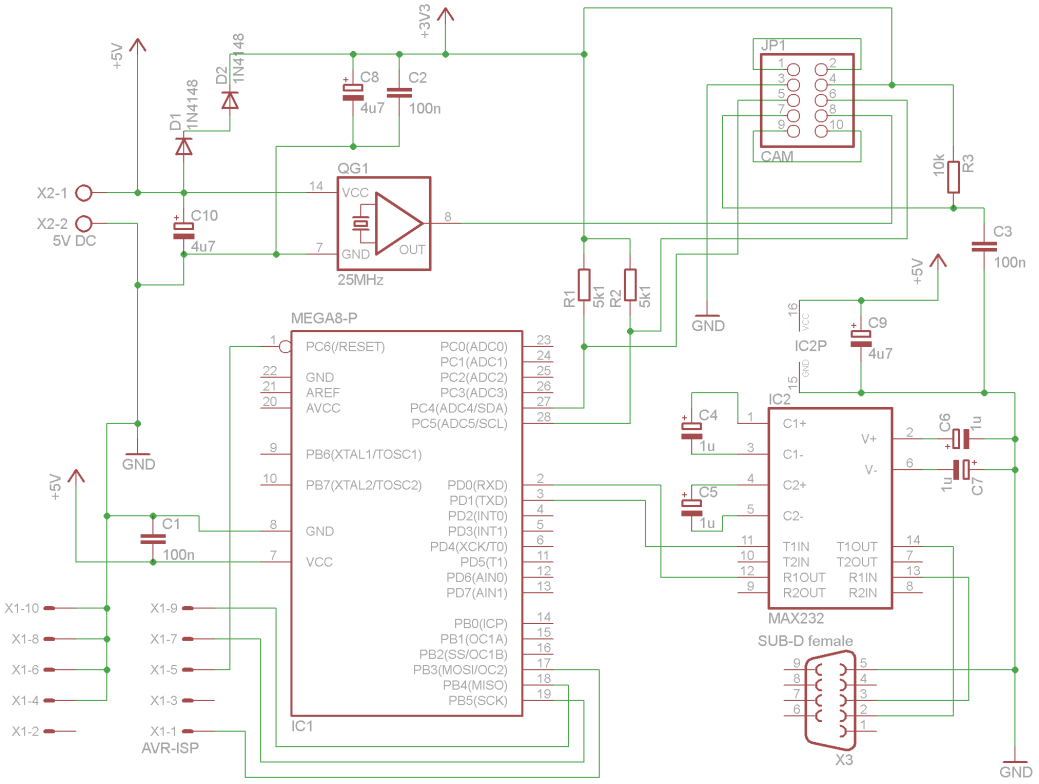


Abb. 4: Schaltplan der Platine

Abgesehen von den bereits aufgeführten Bauteilen (ATmega8, MAX232, Kameramodul) befinden sich auf der Platine nur noch eine ISP-Schnittstelle für den ATmega8, ein Quarzoszillator und ein paar Standardbauteile. Die ISP-Schnittstelle verwendet hierbei eine von Atmel empfohlene Belegung (siehe Abb. 5).[Atm10]

Erwähnenswert sind eventuell noch die Dioden D1 und D2 in Abb. 4, die aus den 5V DC des Platinenanschlusses etwa die vom Kameramodul

benötigten 3,3V DC erzeugen. Obwohl der Mikrocontroller mit 5V DC betrieben wird (und somit auch seine I²C-Schnittstelle), entsteht bei der Kommunikation mit dem Kameramodul (3,3V I²C) kein Problem.

Der Grund dafür liegt in der Spezifikation dieses Schnittstellentyps, die vorsieht, dass die beiden beteiligten Leitungen SCL (Takt) und SDA (Daten) nur durch Pullup-Widerstände (R1 und R2 in Abb. 4) auf High-Pegel gezogen werden. Die beteiligten Geräte erzeugen also einen Low-Pegel, indem sie die entsprechende Leitung auf Masse ziehen, und einen High-Pegel, indem sie die Leitung „loslassen“ (hochohmiger Zustand).

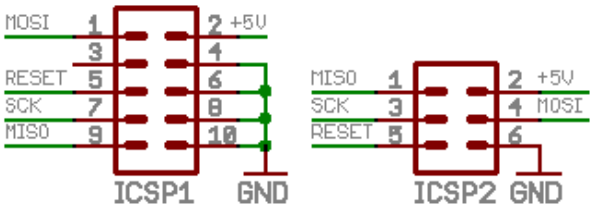


Abb. 5: Anschlussbelegung ISP-Buchse

Indem die Pullup-Widerstände nun an 3,3V angeschlossen werden, liegen die Pegel für das Kameramodul im Optimum, reichen aber auch für den Mikrocontroller mit 5V-I²C noch, um sicher einen High-Pegel erkennen zu können.

Abb. 4 zeigt den Schaltplan der benötigten Platine (erstellt mit Layout-Editor EAGLE[7]), wonach auch der Prototyp (siehe Abb. 3) angefertigt wurde. Das serielle Anschlusskabel für den Rechner ist beim Prototyp bereits fest angelötet, es kann aber ebenso eine 9-polige SUB-D-Buchse benutzt werden. Für Nachbau oder Modifikation befindet sich im Anhang und auf beiliegendem Datenträger ein Vorschlag für ein Platinenlayout (ebenfalls erstellt mit EAGLE), welches sich der Übersicht halber sehr am Layout des Prototyps orientiert, welcher nur auf einer Lochraster-Platine realisiert wurde. Ebenso ist der Schaltplan auf beiliegendem Datenträger als EAGLE-Quelldatei

[7] http://www.cadsoft.de

enthalten.

Obwohl nicht direkt auf der Cam-Platine vorhanden, werden in diesem Kapitel auch kurz die Infrarot-LEDs beschrieben, da sie ebenfalls zur Hardware gehören. Prinzipiell können alle IR-LEDs verwendet werden, wobei man darauf achten sollte, dass sie einen hohen Durchlassstrom besitzen, weil sie dadurch heller sind, und somit die Reichweite und Zuverlässigkeit des kompletten Aufbaus erhöht wird.

Eine noch wichtigere Eigenschaft der LEDs ist jedoch ein hoher Abstrahlwinkel, damit bei einer Drehung des Kopfes möglichst lange ein brauchbares Signal von der Kamera bereitgestellt werden kann. Die Eigenschaften der hier verwendeten IR-LEDs sind aus Tab. 1 ersichtlich[8].[led09]

Hersteller	Typ	Abstrahlwinkel	Wellenlänge	U_{typ}	I_{max}
Osram	CQY99	50°-60°	950nm	1,4V	100mA

Tab. 1: Daten der IR-LED CQY99

3.2 Software

Da die im Mikrocontroller verwendeten Schnittstellen UART und TWI bereits durch die Hardware realisiert werden, muss nur noch wenig durch die Software erledigt werden. Dazu gehört das Initialisieren der Schnittstellen, des Kameramoduls und die Kommunikation mit den beiden Partnern „Rechner“ und „Kameramodul“. Kompiliert wurden die Sourcen mit WinAVR[9] in der Version 4.3.0.

In diesem Kapitel werden Ausschnitte des Quelltextes erläutert, der komplette Quelltext befindet sich jedoch im Anhang und auf beiligendem Daten-

[8]Angabe zum Abstrahlwinkel differiert je nach Datenblatt

[9]http://winavr.sourceforge.net

träger.

```
1 void twiInit(void){
2   TWBR=10; //approx. 80khz twi-clock
3 }
```

Listing 1: aus twi.c

Zuerst initialisiert die Software die beiden verwendeten Schnittstellen UART und TWI. Bei der TWI-Schnittstelle wird dazu nur ein Wert eingestellt, der die Taktung des Busses bestimmt (siehe Listing 1). Berechnet werden kann die Taktung mithilfe folgender Formel:

$$Clock_{TWI} = \frac{Clock_{CPU}}{16 + 2(TWBR) \cdot 4^{TWPS}}$$

In diesem Fall ist *TWBR* auf 10 gesetzt, das *TWPS*-Register hingegen bleibt unberührt, was zu einem *TWPS*-Wert von 1 führt. Mit einer Taktung des Mikrocontrollers (und somit der CPU) von 8 MHz ergibt sich die TWI-Frequenz:

$$Clock_{TWI} = \frac{8\text{MHz}}{16 + 20 \cdot 4} = 83.\overline{3}\text{kHz}$$

Darüber hinaus muss nur noch die (festverdrahtete) TWI-Adresse des Kameramoduls angegeben werden (siehe Listing 2).

```
1 #define TWI_SLAVE_ADDRESS 0xB0 //wii-ir-cam
```

Listing 2: aus twi.h

Bei der UART muss etwas mehr konfiguriert werden. In diesem Projekt wird die übliche Konfiguration „8n1“ verwendet, also 8-Bit-Datenübertragung, keine Parität, 1 Stoppbit. Wichtig ist, dass man auf Rechner- und Mikrocontrollerseite die gleiche Baudrate verwendet, die man in der *uart.h* einstellen kann (im Prototyp auf 38.400 Baud gesetzt, siehe Listing 3).

```
1 #define BAUD 38400
2 #define BAUD_SETTING F_CPU/16/BAUD-1 //calc baudrate
```

Listing 3: aus uart.h

Weiterhin wird die UART dahingehend konfiguriert, dass sie zum Senden (Punktdaten, Fehlersequenz) und zum Empfangen (Datenanfrage vom Rechner) genutzt werden kann. Das Empfangen der Daten ist interruptgesteuert. Tritt nun ein solcher Interrupt auf, was in der Regel nur dann passiert, wenn der angeschlossene Rechner aktuelle Punktdaten haben möchte, wird die globale Variable *request* beschrieben (siehe Listing 4).

```
1  //sends 1 byte <data> over UART
2  void UARTsend(uint8_t data){
3      while(!(UCSRA&(1<<UDRE))){;}
4      UDR=data;
5  }
6
7  //isr for uart receive
8  ISR(USART_RXC_vect){
9    request=UDR;
10 }
```

Listing 4: aus uart.c

Wird die Anfrage als legitim ausgewertet, kann die UART nun die Punktdaten mithilfe der *UARTSend()*-Funktion aus Listing 4 an den Rechner senden.

Im Anschluss an diese Schnittstelleninitialisierung und -konfiguration muss nun auch noch das Kameramodul initialisiert werden. Das Modul erwartet dazu nach [Ele08] eine Sequenz an Daten(-bytes), die in *initCam()* (siehe Listing 5) gesendet werden.

```
1 //sends init sequence to cam
2 void initCam(void){
3   cam_data[0]=0x30;
4   cam_data[1]=0x01;
5   twiSendChars(cam_data, 2);
6
```

```
7     cam_data[0]=0x30;
8     cam_data[1]=0x08;
9     twiSendChars(cam_data, 2);
10
11    cam_data[0]=0x06;
12    cam_data[1]=0x90;
13    twiSendChars(cam_data, 2);
14
15    cam_data[0]=0x08;
16    cam_data[1]=0xC0;
17    twiSendChars(cam_data, 2);
18
19    cam_data[0]=0x1A;
20    cam_data[1]=0x40;
21    twiSendChars(cam_data, 2);
22
23    cam_data[0]=0x33;
24    cam_data[1]=0x33;
25    twiSendChars(cam_data, 2);
26
27    delay_ms(100);
28  }
```

Listing 5: aus main.c

Nach dieser Initialisierungssequenz ist das Kameramodul nun dahingehend konfiguriert, dass es nach Empfangen des Befehl-Bytes 0x36 aktuelle Punktdaten in Form von 16 Bytes bereitstellt, die vom Mikrocontroller (als TWI-Master fungierend) gespeichert werden. Ausgelöst wird dieser Vorgang durch *getCamData()* (siehe Listing 6), wodurch die Daten in einem Array gespeichert werden.

```
1  //sends point-data-request to cam and receives data
2  void getCamData(uint8_t *data){
3    twiRecv(data, 16); //get 16 bytes from cam
4  }
```

Listing 6: aus main.c

Nun, da man zu beliebigen Zeitpunkten aktuelle Punktdaten aus dem Kameramodul anfordern kann, und per Interrupt Anfragen vom Rechner erhält, bleibt der eigentlichen Hauptschleife (siehe Listing 7) nicht mehr viel zu tun.

Intuitiv könnte man ja nun abwarten, bis der Rechner eine Datenanfrage sendet, dann aktuelle Punktdaten vom Kameramodul anfordern und diese anschließend senden. Dieses Vorgehen bedeutet aber logischerweise für jede Anfrage des Rechners eine unnötige Verzögerung, bis er seine Daten erhält.

```
1  while(1){
2    if(request==0xAA){ //check for data-request
3      request=0; //reset request
4      for(i=1;i<13;i++) UARTsend(cam_data[i]);
5    }
6    twiRecv(cam_data, 16); //get 16 bytes from cam
7  }
```

Listing 7: aus main.c

Aufgrund dessen geht die Software etwas anders vor. Sie holt mit maximaler Geschwindigkeit laufend die aktuellen Punktdaten aus der Kamera, unabhängig davon, ob überhaupt eine Anfrage eines Rechners vorliegt. Sollte jedoch ein Rechner eine Anfrage gesendet haben, so kann das erkannt werden, weil die zugehörige ISR[10] die Variable *request* gesetzt hat. Ist das der Fall, können nun umgehend aktuelle Punktdaten zum Rechner geschickt werden. Auffällig ist, dass nur 12 Byte zum Rechner gesendet werden, obwohl immer 16 Bytes vom Kameramodul abgerufen werden. Das liegt daran, dass die letzten 4 Bytes keine Punktinformationen enthalten und somit nicht von Interesse sind.

[10]Interrupt Service Routine

4 Headtracking

4.1 Definition der Begriffe

Die möglichen Bewegungen, die mit einem Kopf ausgeführt werden können, sind Neigen, Nicken und Verneinen. Im folgenden werden die Begriffe definiert, damit danach mit ihnen gearbeitet werden kann.

- Neigen beschreibt die Bewegung eines Kopfes die beim Kippen nach rechts und links ausgeführt wird.
- Verneinen beschreibt die Bewegung eines Kopfes die beim "Nein"-Sagen ausgeführt wird.
- Nicken beschreibt die Bewegung eines Kopfes die beim „Ja"-Sagen ausgeführt wird.

Tracking nennt man die Verfolgung und Erfassung von bewegten Objekten. So werden bestimmte Punkte als Tracking-Punkte festgelegt und beobachtet. Es gibt zwei Dinge, die man mit der Hilfe von Tracking herausfinden kann: Die Lage des Objektes im Raum und den Verlauf der Bewegung. Bei dieser Studienarbeit liegt der Schwerpunkt darauf, die Lage des Kopfes bzw. die Sichtrichtung des Kopfes zu bestimmen. Beim Headtracking werden die Trackingpunkte am Kopf befestigt, so dass sie sich beim Bewegen des Kopfes mitbewegen.

4.2 Idee und Anforderungen

Für das Tracking braucht man Markierungen am Objekt. Anhand dieser Markierungen ist es möglich die Lage des Objektes (in unserem Fall der Kopf) im Raum zu erfassen. Es gibt verschiedene Arten von Markierungen. Für dieses

Projekt werden Infrarot-LEDs als Markierungspunkte verwendet und in einem bestimmten Muster am Kopf befestigt. Infrarot-LEDs als Markierungen zu verwenden ist sinnvoll, da man sie sowohl tagsüber als auch nachts verwenden kann. Am Tag können die Markierungen nicht von anderen Lichtquellen gestört oder mit diesen verwechselt werden, wie das bei „normalen“ LEDs der Fall wäre. Man muss aber bedenken, dass jedes Objekt, das Wärme abstrahlt (und somit auch Infrarot-Strahlung), die Funktion des Aufbaus stören kann. Als Beispiel sei hier ein der Kamera gegeüber liegendes, sonnenbestrahltes Fenster, oder eine Glühlampe mit hoher Leistung im Blickwinkel der Kamera genannt.

5 Realisierung

5.1 Einführung

Als erstes analysieren wir, auf welche Weise sich die einzelnen Infrarot-LED-Markierungen beim Nicken, Neigen und Verneinen bewegen. Als Hilfe wird angenommen, dass die x-Achse parallel zu den Schultern des Menschen verläuft, die y-Achse senkrecht nach oben zeigt und die z-Achse vorne aus dem Modell hinausführt. Der Normalenvektor steht senkrecht auf der Ebene der ersten 3 LEDs und beschreibt somit die Blickrichtung (siehe Abb.6). Beim Neigen bleibt die Blickrichtung des Kopfes gleich, die Position allerdings

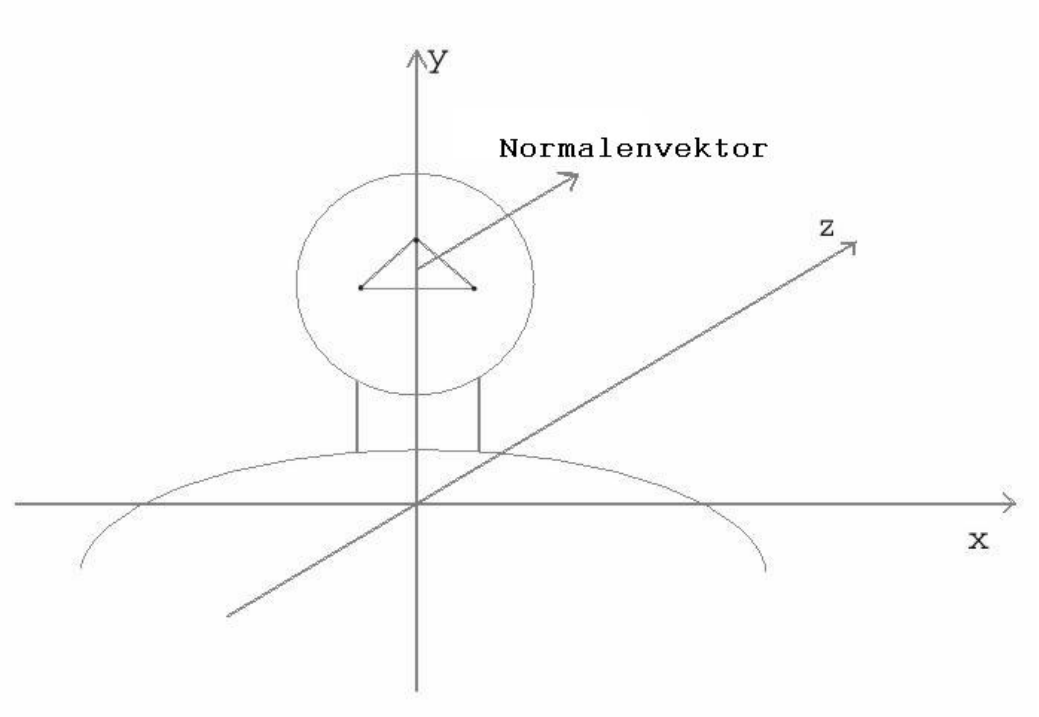


Abb. 6: Modellansicht

nicht. Es wird eine Rotation um die z-Achse ausgeführt. Der Normalenvektor zeigt nun also immernoch in die gleiche Richtung und befindet sich auch

noch auf der gleichen Ebene, seine Position hat sich allerdings verändert. Beim Nicken ändert sich sowohl die Position des Normalenvektors als auch die Blickrichtung. Die Ebene kippt, es wird also eine Rotation um die x-Achse ausgeführt. Beim Verneinen dreht sich der Normalenvektor um die y-Achse. Betrachtet man die LEDs, sieht man, dass sie sich beim Nicken und Verneinen optisch verschieben. Je nachdem in welche Richtung die Bewegung ausgeführt wird, kommen sich die LED-Punkte näher oder entfernen sich von einander. Nur beim Neigen bleiben die Abstände der LEDs aus der Sicht der Kamera gleich.

5.2 Lösungsansätze und Probleme

Als erster Lösungsansatz liegt nun nahe, die Drehungen der Bewegungen Nicken und Verneinen über den Abstand der LEDs zueinander zu bestimmen. Der tatsächliche Abstand zwischen den einzelnen LEDs bleibt natürlich immer gleich. Wir blicken allerdings zunächst senkrecht auf die LEDs. Die Kamera bleibt die ganze Zeit am selben Ort. Daher kommt es einem aus der Sicht der Kamera so vor, als ob sich die Abstände zwischen den LEDs bei den Kopfbewegungen Nicken und Verneinen verändern. Der Grad der Drehung wird dabei folgendermaßen berechnet:

$$grad = acos\left(\frac{abstand}{ausgangsabstand}\right) \cdot \frac{180}{\pi}$$

abstand ist dabei der neu gemessene Abstand und *ausgangsabstand* der alte Abstand. Der Faktor $\frac{180}{\pi}$ ist dabei nötig, um den von *acos* im Bogenmaß gelieferten Winkel in Gradmaß umzurechnen. Führt man nun die entsprechenden Bewegungen aus und berechnet die Winkel mit der angegebenen Formel, entstehen allerdings Fehler. Diese kommen daher, dass sich beispielsweise

bei der Bewegung Nicken auch der Abstand zwischen der Kamera und den LEDs verändert. Somit ist dieses Verfahren für unsere Anwendung unbrauchbar. Auch wenn man mehrere Bewegungen zusammen ausführt kommt es zu Fehlern, da sich die für eine Bewegung relevanten Abstände fast immer auch bei anderen Bewegungen verändern.

5.2.1 Anordnung der LEDs

Wenn man das Projekt mit 3 LEDs auf einer Ebene realisiert, treten ebenfalls Probleme auf. Über den Abstand der verschiedenen LEDs zueinander kann bestimmt werden, wie groß der Winkel ist, um den gedreht wurde. Auf den ersten Blick scheint mit 3 LEDs alles gut zu funktionieren.

Dann stellt sich allerdings das Problem heraus, dass man nicht bestimmen kann, in welche Richtung (rechts/links bzw. oben/unten) der Kopf gedreht wurde, da sich der Abstand der LEDs in beide Richtungen gleich verringert bzw. vergrößert. Man kann mit 3 LEDs also den Winkel der Drehung herausfinden, nicht aber die Richtung der Nicken-/Verneinen-/Neigenbewegung.

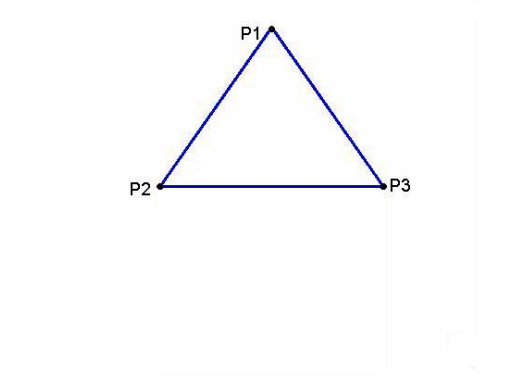


Abb. 7: Anordnung der 3 LEDs

Um nun die Richtung, in die gedreht wurde, zu bestimmen, muss eine vierte LED angebracht werden.

Diese LED wird in der Tiefe angebracht (nicht auf der gleichen Ebene der 3 anderen LEDs). Durch den Abstand dieser Vergleichs-LED zu den anderen LEDs, kann nun die Richtung bestimmt werden, in die der Kopf bewegt wurde.

Die vierte LED wird unterhalb des Dreiecks der drei anderen LEDs angebracht. Aus dem Dreieck der drei Haupt-LEDs wird eine Raute. Die untere LED ist die neu angebrachte vierte LED. Sie steht etwa zwei Zentimeter heraus, ist somit also näher an der Kamera als die anderen drei LEDs. Die Rautenform wurde gewählt, damit die herausstehende LED die anderen LEDs nicht so leicht überdecken kann.

Mit der vierten LED kann nun festgestellt werden, in welche Richtung sich der Kopf bei den einzelnen Bewegungen bewegt. Dadurch kann dann ermittelt werden, ob das Vorzeichen der Gradzahl positiv oder negativ ist. Vor der jeweiligen Drehung wird der Abstand der einzelnen Punkte zum Punkt P4 ermittelt. Bei jeder Drehart (Neigen, Nicken, Verneinen) ist ein anderer Abstand relevant. Nach der Drehung wird der neue Abstand zwischen dem entsprechenden Punkt und P4 ermittelt und mit dem alten Abstand zwischen diesen beiden Punkten verglichen. Ist der Abstand nach der Drehung größer als vorher, wird die Gradzahl negativ. Ist der neue Abstand kleiner als der alte Abstand, ist die Gradzahl positiv.

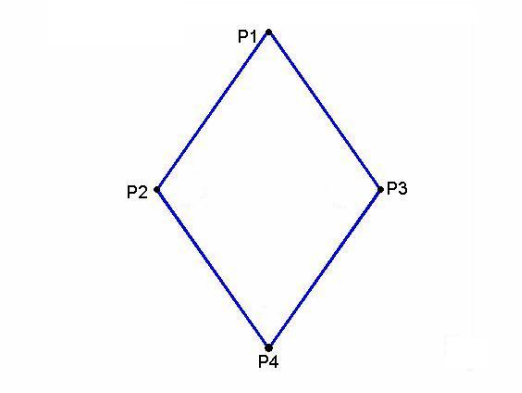


Abb. 8: Anordnung der 4 LEDs

- Beim Verneinen wird der Ausgangsabstand zwischen den x-Werten von P2 und P4 mit dem Abstand nach der Drehung verglichen.
- Beim Nicken wird der Ausgangsabstand zwischen den y-Werten von P1 und P4 mit dem Abstand nach der Drehung verglichen.
- Beim Neigen wird der Ausgangsabstand zwischen den y-Werten von P2

und P4 mit dem Abstand nach der Drehung verglichen.

Die Entscheidung in welcher Richtung Plus und in welcher Minus ist, wurde am Koordinatensystem der Kamera festgelegt (siehe Abb.9).

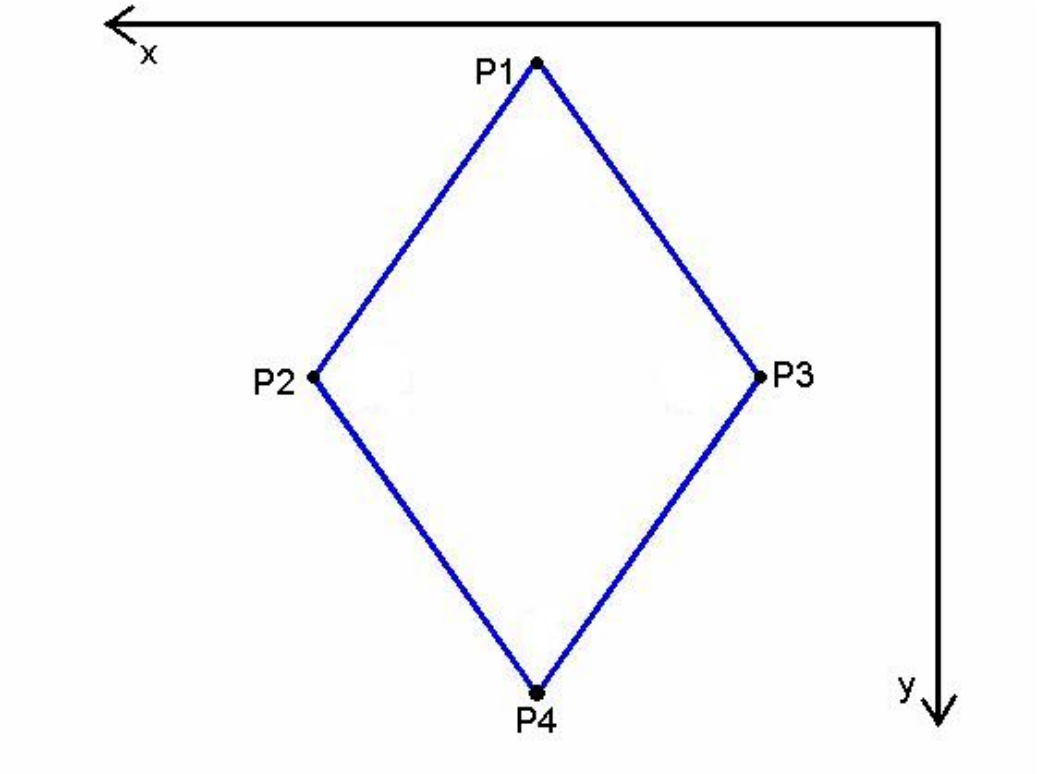


Abb. 9: Die LEDs im Koordinatensystem der Kamera

Insgesamt ist dieser erste Lösungsansatz aber nicht zu gebrauchen, da das Problem der sich gegenseitig bedingenden Bewegungen so nicht eindeutig lösbar ist. Weiterhin wird die translatorische Bewegung, die beispielsweise beim Nicken auftritt, oder wenn der Anwender sich anderweitig auf die Kamera zu- oder von ihr wegbewegt, nicht berechnet, was bei Tests zu weiteren (teilweise großen) Abweichungen geführt hat. Trotzdem sind wichtige Erkenntnisse aus diesem ersten Vorgehen erkannt worden, die auch für die endgültige Lösung benutzt werden konnten.

5.2.2 Algorithmus von DeMenthon

Um diese Probleme zu beheben wird der Algorithmus von DeMenthon zur Hilfe genommen. Dieser Algorithmus erstellt eine 3×3 Rotationsmatrix R mit folgenden Elementen:

$$R = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix}$$

Mit R und den folgenden Formeln[Cra89] kann man nun die einzelnen Rotationen berechnen:

$$yachse = atan2\left(-r_{31}, \sqrt{r_{11}^2 + r_{21}^2}\right)$$

$$zachse = atan2\left(\frac{r_{21}}{cos(yachse)}, \frac{r_{11}}{cos(yachse)}\right)$$

$$xachse = atan2\left(\frac{r_{32}}{cos(yachse)}, \frac{r_{33}}{cos(yachse)}\right)$$

Genaueres zur Funktion des Algorithmus ist in der Beschreibung des Quelltextes zu finden.

5.3 Implementation

In diesem Abschnitt werden Teile des Quellcodes erläutert. Der gesamte Quellcode befindet sich auf dem beigelegten Datenträger. Bevor man das Programm startet sollte gewährleistet sein, dass die Kamera möglichst senkrecht auf die von P1, P2 und P3 aufgespannte Ebene zeigt. Nur so können die LEDs, aufgrund ihrer Lage, den Punkten P1, P2, P3 und P4 zugeordnet werden.

5.3.1 cam.c

```
1  int updatePoints(unsigned int *dst)
2  {
3     extern unsigned int points[8]; //[p1x, p1y, p2x, p2y, p3x, p3y, p4x, p4y]
4                                    //  0    1    2    3    4    5    6    7
5     getCamData();
6     convertData(dst);
7
8     if((points[1]==1023)||(points[3]==1023)||
9        (points[5]==1023)||(points[7]==1023))
10    {
11       printf("\n\nEin oder mehrere Punkte sind nicht mehr im Sichtfeld der "
12              "Kamera!\nBitte bringen Sie die LEDs in eine andere Position"
13              "!");
14       return 1;
15    }
16    else return 0;
17 }
```

Listing 8: aus cam.c

In der Funktion *updatePoints* werden laufend die Punktdaten aktualisiert. Dazu werden zwei weitere Funktionen aufgerufen, *getCamData* und *convertData*.

getCamData sendet hierzu eine Anfrage zur Kamera, woraufhin diese 12 Bytes mit aktuellen Punktdaten übermittelt (siehe Listing 9).

```
1  //requests point-data, stores 12 byte to <cam_data>
2  void getCamData(void)
3  {
4     extern unsigned char cam_data[12]; //raw data from cam
5     char request=0xAA; //cam-command for request
6     write(com_fd, &request, sizeof(request)); //request data
7     //delay_ms(1); //wait for cam to react (1ms)
8     read(com_fd, cam_data, 12); //store data to <cam_data>
9  }
```

Listing 9: aus cam.c

Dieser Datenstrom mit den Punktdaten muss aber erst noch in eine leichter zu verarbeitende Form konvertiert werden, da die Kamera, um weniger Daten übertragen zu müssen, diese etwas anders zusammenstellt, als man das

vielleicht zunächst erwarten würde. Der Grund dafür ist, dass jede Koordinate von jedem Punkt den Wert 1023 annehmen kann. Da dieser Wert nicht in ein Byte passt, müsste man nun eigentlich 2 Bytes für jede Koordinate verwenden, was in 16 zu übertragenden Bytes enden würde. Das ist aber unnötig, wie das Übertragungsformat der Kamera zeigt.

Byte 1	Byte 2	Byte 3
Bits 7..0 vom X-Wert	Bits 7..0 vom Y-Wert	Bits 9..8 vom X-Wert Bits 9..8 vom Y-Wert

Tab. 2: Datenformat der Koordinaten eines Punktes

Wie man Tabelle 2 entnehmen kann, werden die 10-Bit-Koordinaten von jedem Punkt nun auf 3 Bytes aufgeteilt, was für 4 Punkte die 12 übertragenen Bytes erklärt.[Ele08] Die Umrechnung dieses Formats übernimmt *convertData*.

```
1 //converts received cam-data from <cam_data> to point x-y-values <*dst>
2 void convertData(unsigned int *dst)
```

Listing 10: aus cam.c

Gelangt eine LED (oder mehrere) aus dem Sichtfeld der Kamera, so muss dieser Fehler abgefangen werden. Mit einer Fehlerabfrage in den Zeilen 8-16 in Listing 8 wird sichergestellt, dass alle Punkte im Sichtbereich der Kamera sind. Ansonsten wird eine Fehlermeldung ausgegeben. Wenn die y-Werte der Punkte den Wert 1023 annehmen, sind sie nicht mehr im Sichtfeld der Kamera. Ist dies der Fall, erscheint im Konsolenfenster die Fehlermeldung „Ein oder mehrere Punkte sind nicht mehr im Sichtfeld der Kamera! Bitte bringen Sie die LEDs in eine andere Position!". Sobald wieder alle Punkte

im Sichtbereich der Kamera liegen, läuft das Programm wie gewohnt weiter.

```
1 void sortPoints(unsigned int *points)
```

Listing 11: aus cam.c

Nun sollen die eingelesenen Koordinaten den richtigen Punkten zugeordnet werden. Das heißt, dass die Punkte immer gleich benannt werden müssen, auch wenn mal eine oder mehrere LEDs aus dem Sichtbereich der Kamera geraten sind. Dem Array *points* werden die Koordinaten der LEDs von der Kamera in der Form *(x1,y1,x2,y2,x3,y3,x4,y4)* übergeben. Die Koordinaten sollen nun so sortiert werden, dass beispielsweise x1 und y1 auch unserem Punkt P1 entsprechen. Die Koordinaten sollen den Punkten so zugeordnet werden, dass sie unserer Nummerierung der LEDs entsprechen (siehe Abb.8). Es werden vier Zeiger erzeugt, die auf den jeweiligen x-Wert des entsprechenden Punktes zeigen. Durch if-Schleifen wird festgestellt, welche Werte zu welchem Punkt gehören. Bei der Sortierung der Punkte werden zunächst die x-Werte verglichen. Das Koordinatenpaar mit dem geringsten x-Wert wird P3 zugeordnet. Das Koordinatenpaar mit dem größten x-Wert gehört zu P2. Nun müssen noch die y-Werte der einzelnen Werte verglichen werden. Das Koordinatenpaar mit dem kleinsten y-Wert ist P1. Das Koordinatenpaar mit dem höchsten y-Wert wird P4 zugeordnet.

5.3.2 main.c

```
1 double **ReadObjectFile(char *name, int *nbPts)
```

Listing 12: aus main.c

In der Funktion *ReadObjectFile* werden die LED-Koordinaten aus der *object.txt* eingelesen und zurückgegeben. In der *object.txt* wird das LED-Objekt beschrieben. Die Anzahl der verwendeten LEDs und ihre Position in relativen Koordinaten wird dort definiert. Es werden die x-, y- und z-Werte der einzelnen LEDs benötigt. Als Einheit werden Millimeter empfohlen. Dies ist aber nicht zwingend notwendig. Durch die Verwendung der *object.txt* ist es sehr einfach das LED-Objekt zu verändern. Wenn man sich also für ein anderes LED-Objekt entscheidet, trägt man die neuen Koordinaten der LEDs einfach in die *object.txt* ein und kann dann damit ohne weitere Anpassungsmaßnahmen im Programm arbeiten.

```
1 void getObjectData(char *objectName, TObject *object)
```

Listing 13: aus main.c

In der Funktion *getObjectData* werden die Daten von *ReadObjectFile* in eine structure geschrieben.

```
1 void getImageData(TObject *object, TImage *image)
```

Listing 14: aus main.c

getImageData schreibt die aktuellen Punktdaten in eine structure. Mit *imageCenter* wird ein virtueller Mittelpunkt im Objekt erzeugt. Damit das Koordinatenkreuz im Fokus der Kamera liegt, wird es in den Mittelpunkt des Bildes verschoben. Dies ist für die Berechnung der Translation in der *posit.c* nötig. Ohne diesen virtuellen Mittelpunkt würde auch eine Translation angezeigt, wenn das LED-Objekt im Brennstrahl der Kamera liegt, was hier aber unerwünscht ist.

```
1 void refreshImage(void)
2 {
```

```
3     int error=0;
4     //refresh image data with new cam data
5     while(updatePoints(points)) error=1;
6     if(error) sortPoints(points); //sort to define
7  }
```

Listing 15: aus main.c

In der Funktion *refreshImage* wird das Bild mit den neuen Kameradaten aktualisiert. Solang der Rückgabewert der Funktion *updatePoints* (siehe Listing 8) den Wert *1* hat, wird die Variable *error* auf *1* gesetzt. Wenn *error=1* ist, werden die Punkte neu sortiert. Somit werden die Punkte nur neu sortiert, wenn ein oder mehrere Punkte aus dem Blickfeld der Kamera gelangt sind, was viel Rechenzeit spart.

```
1  void calcDegree(double rotmat [3][3])
2  {
3     yachse = atan2(-rotmat[2][0],
4                    sqrt(pow((rotmat[0][0]),2)+pow((rotmat[1][0]),2)));
5     zachse = atan2((rotmat[1][0])/cos(yachse), (rotmat[0][0])/cos(yachse));
6     xachse = atan2((rotmat[2][1])/cos(yachse), (rotmat[2][2])/cos(yachse));
7     //xachsegrad = xachse*(180/M_PI); //not needed for openGL part
8     //yachsegrad = yachse*(180/M_PI); //converts from radian to degree
9     //zachsegrad = zachse*(180/M_PI);
10 }
11
```

Listing 16: aus main.c

Die Aufgabe der Funktion *calcDegree* ist es, aus der Rotationsmatrix die Bewegungen in Richtung der einzelnen Achsen zu berechnen. Die Drehbewegungen um die einzelnen Achsen werden in Bogenmaß berechnet. Für diese Berechnungen wird der Arcustangens2 verwendet.

```
1  void refreshCam(TObject *wiiObject, TCamera *wiiCamera, TImage *wiiImage)
```

Listing 17: aus main.c

In *refreshCam* werden die aktuellen Kameradaten geholt und in eine structure geschrieben. Mit Hilfe von *POSIT* werden die Rotationsmatrix und der Translationsvektor berechnet. Danach wird noch die Drehung in die einzelnen Richtungen in Bogenmaß ermittelt.

```
1 int main(void)
```

Listing 18: aus main.c

main setzt den Mittelpunkt der Koordinaten und die focal length. Als guter Wert für die Brennweite haben sich 1280 Pixel herausgestellt[PV]. Danach wird die *object.txt* eingelesen und der Rang der Matrix bestimmt. Anhand des Ranges kann nun bestimmt werden ob das Objekt zu flach ist. Ist das Objekt zu flach, kann nicht gut mit ihm gearbeitet werden. Man sollte in so einem Fall ein anderes Objekt wählen. Nun wird die Kamera gestartet und die Funktionen *updatePoints* und *sortPoints* aufgerufen. Die Punktdaten aus der Kamera werden also eingelesen und sortiert. *roomControl* stellt sicher, dass die aktuelle Kontrolle über das Programm bei OpenGL liegt.

5.3.3 posit.c

```
1 void POS(TObject object, TImage image, TCamera *camera)
```

Listing 19: aus posit.c

Die Funktion *POS* berechnet den Translationsvektor und die Rotationsmatrix und gibt diese wieder zurück. Im ersten Schritt wird ein angenäherter Wert berechnet. Die Position des Objektes im Raum wird durch das Lösen linearer Systeme gefunden. Diese Schritte werden solange wiederholt bis keine Verbesserung mehr erkennbar ist. Allerdings liefert dieser Algorithmus in den meisten Fällen nur Annäherungen.

```
1 long GetImageDifference(TImage image)
```

Listing 20: aus posit.c

In der Funktion *GetImageDifference* wird die Summe der Differenz der alten und neuen Bildpunkte errechnet und zurückgegeben.

```
1 void POSIT(TObject object, TImage image, TCamera *camera)
```

Listing 21: aus posit.c

Mit Hilfe des *POSIT*-Algorithmus kann man, anhand von Punkten, die Position eines Objektes im Raum bestimmen. Die *POSIT* iteriert über die Ergebnisse aus der *POS*-Funktion. Somit kann die Position des Objektes Schritt für Schritt immer genauer bestimmt werden. Die „korrekten“ Bildpunkte werden mit Hilfe von skalierten orthografischen Projektionen gefunden. Die skaliert orthografische Projektion ist eine Mischform aus der orthogonalen Projektion und der perspektivischen Projektion. Bei der skaliert orthografischen Projektion werden die Punkte des Objektes als ersten orthogonal auf eine virtuelle Bildfläche projiziert. Danach werden die Punkte auf der virtuellen Bildfläche noch einmal perspektivisch projiziert.

Die *svd.c* wurde komplett von Daniel DeMenthon[DeM03] übernommen. In der *svd.c* werden die benötigten Arrays, Vektoren und Matrizen deklariert. Außerdem befinden sich dort einige Funktionen zur Ausgabe der Vektoren und Matrizen.

6 Demonstrationsprogramm

6.1 Grundidee

Teil der Studienarbeit ist außerdem eine kleine Anwendung, mit der man die Funktionalität des Programmes testen kann. Hierfür wurde in OpenGL ein einfacher Raum erstellt. In diesem Raum kann man sich nun „umsehen“.

6.2 OpenGL

OpenGL[ope] steht für Open Graphics Library. OpenGL ist also eine freie Grafik Bibliothek, mithilfe derer man 3D Objekte darstellen kann. Außerdem wurde für das Demonstrationsprogramm GLUT[glu] Version 3.7 verwendet. GLUT steht für OpenGL Utility Toolkit und ermöglicht den Zugriff auf die grafische Oberfläche und Eingabegeräte.

6.3 Implementation

In diesem Abschnitt werden wiederum Teile des Quellcodes erläutert. Der gesamte Quellcode befindet sich auf dem beigelegten Datenträger. Als Compiler wurde der GNU C Compiler in der Version 3.4.2 verwendet.

```
1 void display(void)                    //here's where we do all the drawing
```

Listing 22: aus raum.c

In der Funktion *display* wird der Raum gezeichnet. Der Raum besteht aus sechs Flächen, die in verschiedenen Farben gezeichnet werden, damit sie besser unterschieden werden können.

```
1 int init(void)
```

Listing 23: aus raum.c

Die Funktion *init* initialisiert die OpenGL-Parameter. Hier wird unter anderem auch festgelegt, dass eine perspektivische Projektion vorgenommen werden soll, damit ein realistischer Eindruck ensteht.

```
1  void idle(void)
2  {
3     int i;
4     const int meanValue=10;
5     double xMid=0.0, yMid=0.0, zMid=0.0;
6     extern double xachse, yachse, zachse;//degree-values of movement
7     extern TObject wiiObject;//descripton of LED layout
8     extern TImage  wiiImage; //current image from wii-cam
9     extern TCamera wiiCamera;//descripton of virtual cam
10    for (i=0;i<meanValue;i++)//calc mean value to prevent jitter
11    {
12       refreshCam(&wiiObject, &wiiCamera, &wiiImage);
13       xMid+=xachse; yMid+=yachse; zMid+=zachse;
14    }
15    xMid/=meanValue; yMid/=meanValue; zMid/=meanValue;
16    glLoadIdentity();
17    gluLookAt(0.5f,0.5f,0.8f, //here i am
18              0.5f-yMid*3, //turn left/right
19              0.5f-xMid*3, //turn up/down
20              -1.0f,
21              0+zMid*3,1,0); //mod. up-vector to simulate head pitch
22    glutPostRedisplay();
23 }
```

Listing 24: aus raum.c

In der Funktion *idle* werden die Kameradaten aktualisiert und das Bild der neuen Position des Kopfes angepasst. In den Zeilen 10-15 werden die aktuellen Kameradaten übergeben und die Mittelwerte der einzelnen Achsenbewegungen errechnet. Dieser Vorgang macht das Bild stabiler. Mit der *gluLookAt* in den Zeilen 17-21 wird die Kamera gesetzt. Die ersten drei Koordinaten beschreiben, wo sich die Kamera gerade im Raum befindet. Die vierte, fünfte und sechste Koordinate geben den Punkt im Raum an, auf den die Kamera zeigt. Dieser Punkt wird auch Centerpunkt genannt. Die letzten drei Koordinaten beschreiben den UP-Vektor der Kamera. Er bestimmt die vertikale Ausrichtung der Kamera. Vom x-Wert des Centerpunktes wird der Mittel-

wert der y-Achse subtrahiert. Bei einer Drehung um die x-Achse verschiebt sich der fokussierte Punkt in der y-Achse. Ebenso wird vom y-Wert des Centerpunktes der Mittelwert der x-Achse subtrahiert. Eine Drehung um die y-Achse verursacht eine Verschiebung des fokussierten Punktes entlang der x-Achse. Die Drehung um die z-Achse verursacht, dass der up-Vektor zur Seite geneigt wird. Der Mittelwert wird mit der Zahl 3 multipliziert um die Drehungen zu verstärken.

7 Fazit

Abschließend möchte ich nochmals die Problemstellung, die Vorgehensweise und das Ergebnis meiner Studienarbeit zusammenfassen. Ziel der Studienarbeit war es, mit der Kamera aus der Wii-Fernbedienung die Kopfbewegungen eines Menschen anhand von Infrarot-LEDs bestimmen zu können. Daraus sollte dann die Blickrichtung des Menschen ermittelt und in einem kleinen Demonstrationsprogramm dargestellt werden. Besondere Herausforderung dabei war das mathematische Problem zur Errechnung der Kopfbewegung. Der erste Ansatz scheiterte, da sich die einzelnen Bewegungen gegenseitig bedingten. Deshalb wurde als Lösungshilfe der *POSIT*-Algorithmus von DeMenthon hinzugezogen. Er macht es möglich von einem bekannten Objekt die Rotationsmatrix zu bestimmen. Daraus können dann die einzelnen Drehungen des Kopfes berechnet werden. Der OpenGL-Raum ist natürlich nur ein Beispiel für eine Anwendung des Programms. Man könnte es auch für viele andere Anwendungen benutzen, wie beispielsweise für Flugsimulationen. Außerdem wird die Möglichkeit geboten, das LED-Objekt zu ändern und gegebenfalls zu optimieren. Durch die zusätzliche Ausgabe der Translation ist dieses Programm außerdem sehr gut für Weiterentwicklungen geeignet.

Literatur

[Atm09] Atmel: *Datenblatt AVR ATmega8*, 2009. [http://atmel.com/dyn/resources/prod_documents/doc2486.pdf].

[Atm10] Atmel: *AVR Hardware Design Considerations*, 2010. [http://www.atmel.com/dyn/resources/prod_documents/doc2521.pdf].

[Cra89] Craig, John J.: *Introduction to Robotics*. Addison-Wesley, 2. Auflage, 1989.

[DD95] Daniel DeMenthon, Larry Davis: *Model-Based Object Pose in 25 Lines of Code*, 1995. [http://www.cfar.umd.edu/~daniel/daniel_papersfordownload/Pose25Lines.pdf].

[DeM03] DeMenthon, Daniel: *POSIT*, 1993 - 2003. [http://www.cfar.umd.edu/~daniel/Site_2/Code.html].

[dis] [http://www.itwissen.info/bilder/head-mounted-display-hmd-foto-vr-realities-dot.png].

[Ele08] Elektor: *Den Hot-Spots auf der Spur - Mega88 verfolgt Infrarotquellen*, 2008. Ausgabe 10/2008, Seite 42 ff.

[glu] [http://www.opengl.org/resources/libraries/glut/].

[led09] *Datenblatt Infrarot-LED CQY99*, 2009. [http://www.ic-online.cn/IOL/datasheet/cqy99_809937.pdf].

[Mül08] Müller, Prof. Dr. Stefan: *Folien „Virtuelle Realität und Augmented Reality“*, 2007/2008. [http://userpages.uni-koblenz.de/~cg/ws0708/vrar/vrar_ws0708.zip].

[ope] [http://www.opengl.org/].

[PV] Phong Vuong, Dr. Gregorij Kurillo, Dr. Ruzena Bajcsy: *Dr. Oliver Kreylos' Wiimote Tracking Algorithm and its Limitations*. [http://phongvuong.com/wordpress/wp-content/uploads/2009/11/wiimotePaper1.pdf].

[wii] [http://www.nintendo.co.jp/n10/e3_2006/wii/img_con/photo_controller.jpg].

[ZX03] Zhiang Xiang, Roy A.Plastock: *Computergrafik*. mitp-Verlag, 1. Auflage, 2003.

A Anhang

A.1 Layout Cam-Platine

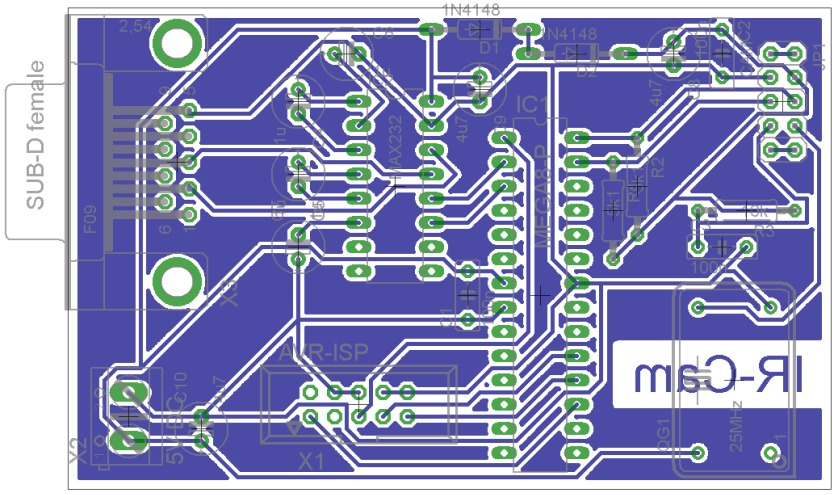


Abb. 10: Layout der Platine

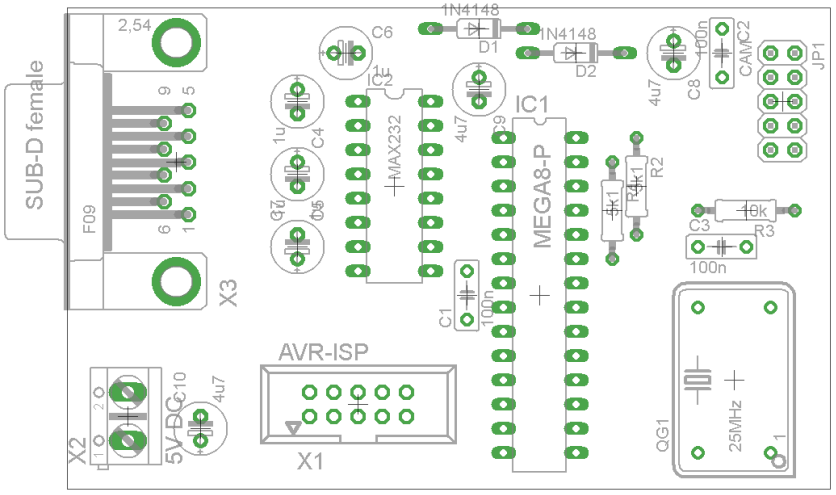


Abb. 11: Bauteile der Platine

A.2 Quelltext Cam-Platine

```
#include <avr/io.h>
#include <util/delay.h>
#include "uart.h"
#include "twi.h"

uint8_t cam_data[16];
volatile uint8_t request=0x00;

//delay of <count> * 1ms
void delay_ms(uint16_t count){
   while (count!=0){
      _delay_ms(1.0);
          count--;
   }
}

//error-routine, sends 0xFF, which can't be point-data
void error(void){
  UARTsend(0xFF);
  while(1){;}
}

//sends init sequence to cam
void initCam(void){
  cam_data[0]=0x30;
  cam_data[1]=0x01;
  twiSendChars(cam_data, 2);

  cam_data[0]=0x30;
  cam_data[1]=0x08;
  twiSendChars(cam_data, 2);

  cam_data[0]=0x06;
  cam_data[1]=0x90;
  twiSendChars(cam_data, 2);

  cam_data[0]=0x08;
  cam_data[1]=0xC0;
  twiSendChars(cam_data, 2);

  cam_data[0]=0x1A;
  cam_data[1]=0x40;
  twiSendChars(cam_data, 2);

  cam_data[0]=0x33;
  cam_data[1]=0x33;
  twiSendChars(cam_data, 2);

  delay_ms(100);
}

//sends point-data-request to cam and receives data
```

```
53 void getCamData(uint8_t *data){
54   twiRecv(data, 16); //get 16 bytes from cam
55 }
56
57 int main(void){
58 uint8_t i;
59   delay_ms(100); //power-up-delay
60   UARTinit(BAUD_SETTING);
61   twiInit();
62   initCam();
63   sei();
64   while(1){
65     if(request==0xAA){ //check for data-request
66       request=0; //reset request
67       for(i=1;i<13;i++) UARTsend(cam_data[i]);
68     }
69     twiRecv(cam_data, 16); //get 16 bytes from cam
70   }
71   while(1){;}
72   return 0;
73 }
```

Listing 25: main.c

```
1  #include <avr/io.h>
2  #include <avr/interrupt.h>
3
4  #define BAUD 38400
5  #define BAUD_SETTING F_CPU/16/BAUD-1 //calc baudrate
6
7  extern volatile uint8_t request;
8
9  void UARTinit(uint16_t setting);
10 void UARTsend(uint8_t data);
```

Listing 26: uart.h

```
1  #include "uart.h"
2
3  //inits UART, 8,N,1, no parity
4  void UARTinit(uint16_t setting){
5     UBRRH=(uint8_t)(setting>>8);
6     UBRRL=(uint8_t)setting;
7     UCSRC|=(1<<URSEL)|(1<<UCSZ1)|(1<<UCSZ0);
8     UCSRB|=(1<<RXCIE)|(1<<TXEN)|(1<<RXEN);
9  }
10
11 //sends 1 byte <data> over UART
12 void UARTsend(uint8_t data){
13    while(!(UCSRA&(1<<UDRE))){;}
14    UDR=data;
15 }
```

```
16
17 //isr for uart receive
18 ISR(USART_RXC_vect){
19   request=UDR;
20 }
```

Listing 27: uart.c

```
1  #include <avr/io.h>
2  #include "uart.h"
3
4  #define TWI_SLAVE_ADDRESS 0xB0 //wii-ir-cam
5
6  extern void delay_ms(uint16_t count);
7  extern void error(void);
8
9  void twiInit(void);
10 void twiSendChars(uint8_t *data, uint8_t size);
11 void twiRecv(uint8_t *dst, uint8_t size);
```

Listing 28: twi.h

```
1  #include "twi.h"
2
3  void twiInit(void){
4    TWBR=10; //approx. 80khz twi-clock
5  }
6
7  //sends <size> bytes from <*data> over TWI
8  void twiSendChars(uint8_t *data, uint8_t size){
9  uint8_t i;
10   TWCR=(1<<TWINT)|(1<<TWSTA)|(1<<TWEN); //send start cond.
11   while(!(TWCR&(1<<TWINT))){;} //wait for being sent
12   if((TWSR&0xF8)!=0x08) error(); //check for error
13
14   TWDR=TWI_SLAVE_ADDRESS; //add nothing cause write-command
15   TWCR=(1<<TWINT)|(1<<TWEN); //send slave address
16   while(!(TWCR&(1<<TWINT))){;} //wait for being sent
17   if((TWSR&0xF8)!=0x18) error(); //check for error
18
19   for(i=0;i<size;i++){
20     TWDR=data[i]; //load data
21     TWCR=(1<<TWINT)|(1<<TWEN); //send data
22     while(!(TWCR&(1<<TWINT))){;} //wait for being sent
23     if((TWSR&0xF8)!=0x28) error(); //check for error
24   }
25   TWCR=(1<<TWINT)|(1<<TWSTO)|(1<<TWEN); //send stop cond.
26   while(!(TWCR&(1<<TWSTO))){;} //wait for bus release
27   delay_ms(10);
28 }
29
30 //sends point-data-request 0x36 to cam and
```

```
31 //receives <size> bytes and stores to <*dst>
32 void twiRecv(uint8_t *dst, uint8_t size){
33 uint8_t i=0x36; //command for cam to prepare point-data
34   twiSendChars(&i,1);
35   delay_ms(1);
36   TWCR=(1<<TWINT)|(1<<TWSTA)|(1<<TWEN); //send start cond.
37   while(!(TWCR&(1<<TWINT))){;} //wait for being sent
38   if((TWSR&0xF8)!=0x08) error(); //check for error
39
40   TWDR=TWI_SLAVE_ADDRESS+1; //add 1 cause read-command
41   TWCR=(1<<TWINT)|(1<<TWEN); //send slave address
42   while(!(TWCR&(1<<TWINT))){;} //wait for being sent
43   if((TWSR&0xF8)!=0x40) error(); //check for error
44
45   for(i=0;i<size-1;i++){
46     TWCR=(1<<TWINT)|(1<<TWEN)|(1<<TWEA); //send command
47                                          //TWEA=1 -> more bytes
48         while(!(TWCR&(1<<TWINT))){;} //wait for being sent
49         if((TWSR&0xF8)!=0x50) error(); //check for error
50         dst[i]=TWDR;
51   }
52   TWCR=(1<<TWINT)|(1<<TWEN); //send command,
53                              //TWEA=0 -> no more bytes
54         while(!(TWCR&(1<<TWINT))){;} //wait for being sent
55         if((TWSR&0xF8)!=0x58) error(); //check for error
56         dst[size-1]=TWDR;
57
58   TWCR=(1<<TWINT)|(1<<TWSTO)|(1<<TWEN); //send stop cond.
59   while(!(TWCR&(1<<TWSTO))){;} //wait for bus release
60 }
```

Listing 29: twi.c

A.3 Aufbau

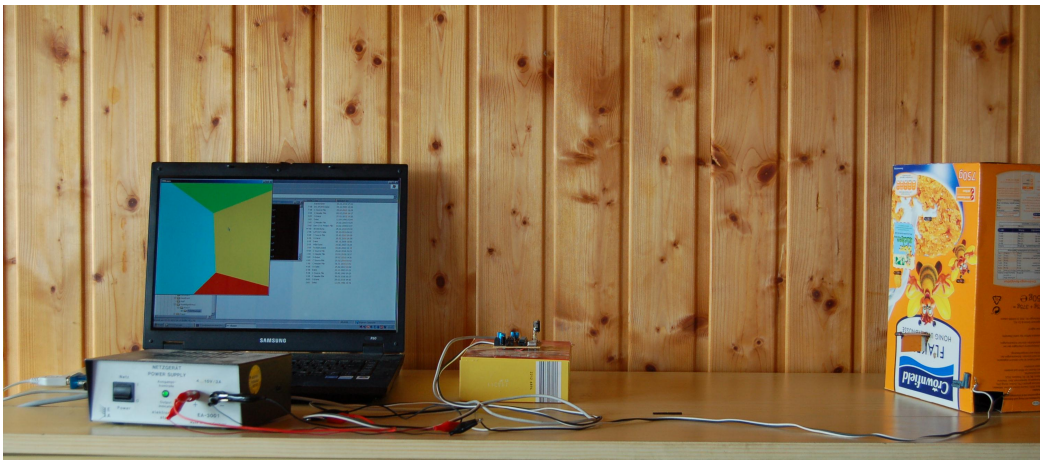


Abb. 12: Aufbau