UNIVERSITÄT
KOBLENZ · LANDAU

Fachbereich 4: Informatik

Template Tracking auf Basis eines Partikelfilters

Bachelorarbeit

zur Erlangung des Grades Bachelor of Science (B.Sc.)
im Studiengang Computervisualistik

vorgelegt von

Dennis Hahn

Erstgutachter: Prof. Dr.-Ing. Stefan Müller
(Institut für Computervisualistik, AG Computergraphik)

Zweitgutachter: Nils Höhner

Koblenz, im Mai 2019

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	□	□

. . . .

(Ort, Datum) (Unterschrift)

Inhaltsverzeichnis

1 Einführung 1
1.1 Motivation . . . 1
1.2 Zielsetzung . . . 1
1.3 Aufbau der Bachelorarbeit . . . 1

2 Stand der Wissenschaft 3
2.1 Kantentracking mit Partikelfilter . . . 3
2.1.1 Facetracking auf Stream Multi-Prozessoren . . . 4
2.1.2 Tacking von Bewegungen anhand von Tiefendaten . . . 5
2.1.3 3D-Model Tracking mit Partikelfilter . . . 6
2.1.4 RGB-D Tracking . . . 7

3 Grundlagen 8
3.1 Partikelfilter . . . 8
3.1.1 Bayes Filter . . . 8
3.1.2 Partikelfilter . . . 9
3.2 Cuda . . . 11
3.2.1 CUDA Architektur . . . 11
3.2.2 CUDA Programmiermodel . . . 14

4 Konzeption 15
4.1 Likelyhood-Funktionen . . . 15
4.2 Testumgebungen . . . 15
4.2.1 Virtuelle Testumgebung . . . 16
4.3 Marker . . . 16
4.4 Programmverlauf . . . 17

5 Implementation 20
5.1 Partikel und Partikelmanager . . . 20
5.2 ZED SDK . . . 20
5.3 OpenGL . . . 21
5.3.1 Marker . . . 21
5.4 Framebuffer Objects und Texturen . . . 22
5.5 Host to Device Operationen . . . 23
5.6 Sampleing . . . 24
5.7 Resampling Methods . . . 25
5.7.1 Multinomial Resampling . . . 26
5.7.2 Group-Based Resampling . . . 27
5.8 Likelyhood-Funktionen . . . 28

6 Evaluation **30**
6.1 Hardware . . . 30
6.2 Software . . . 30
6.3 Ausgangsparameter . . . 30
6.4 Versuchsaufbau . . . 31
6.5 HSV-Farbwerte . . . 31
6.5.1 Testumgebung “Karton” . . . 31
6.5.2 Optimierte Versuchsumgebung . . . 32
6.6 Normalen . . . 32
6.6.1 Testumgebung “Karton” . . . 32
6.6.2 Optimierte Versuchsumgebung . . . 34
6.7 Tiefenwerte . . . 34
6.7.1 Optimierte Versuchsumgebung . . . 34
6.8 HSV-Normalen Likelyhood . . . 35
6.9 HSV-Tiefenwerte Likelyhood . . . 35
6.10 Normalen-Tiefenwerte Likelyhood . . . 36
6.11 Kanten . . . 41
6.12 Versuche mit ZED Stereokamera . . . 41
6.13 Auflistung aller Durchschnittswerte . . . 43
6.14 Technische Daten . . . 43
6.15 Rückschlüsse . . . 45

7 Fazit **46**

Zusammenfassung

In dieser Bachelorarbeit wird ein System zur Kameratracking implementiert, dass auf Basis eines Partikelfilters arbeitet. Dazu wird ein Markertracking realisiert und anhand der Markerposition die Kameraposition errechnet. Der Marker soll mit ein Partikelfilter gefunden werden und um das zu bewerkstelligen werden mögliche Markerpositionen simuliert, auch Partikel genannt, und mit Likelyhood-Funktionen gewichtet. Fokus liegt auf der Evaluation von verschiedenen Likelyhood-Funktionen des Partikelfilters. Die Likelyhood-Funktionen wurden in CUDA umgesetzt als Teil der Implementation.

Abstract

This bachelor thesis implements a system for camera tracking based on a particle filter. For this purpose, a marker tracking is realized and the camera position is calculated based on the marker position. The marker is to be found with a particle filter and in order to accomplish this possible marker positions are simulated, also called particles, and weighted with Likelyhood-Functions. The focus lies on the evaluation of different Likelihood-Functions of the particle filter. The Likelyhood functions were implemented in CUDA as part of the implementation.

1 Einführung

1.1 Motivation

Das Bestimmen der Kameraposition ist ein immer wiederkehrendes Thema im Computersehen. Ob bei Augmented Reality Anwendungen, bei denen eine korrekte Kameraposition entscheidend ist für die Platzierung von Objekten in der künstlichen Umgebung in Proportion zum Kamerabild oder für Robotik Anwendungen in denen sich der Roboter anhand von Kamerabildern orientieren muss damit er überhaupt seine jeweiligen Aufgaben erfüllen kann.
In der Vergangenheit wurde die Kameraposition anhand von 3D-Objekten bestimmt. Aus diesen konnte man Kanten bzw. Liniensegmente bestimmen und diese einem Kamerabild anpassen, sodass sich dadurch eine Kameraposition schätzen ließ. Die Theorie für Partikelfilter gab es schon lang, bevor lauffähige Implementationen verbreitet waren. Durch die rasante Entwicklung im Bereich der Grafikhardware gewannen Partikelfilter immer mehr an Relevanz. Vor allem dank NVIDIAS CUDA Architektur wurde es Entwicklern vereinfacht General Purpose Anwendungen, die auf der GPU laufen, zu schreiben. Da die Grafikhardware mit Stream Multi-Prozessor Technologie arbeitet, sind stark parallelisierbare Algorithmen wie die Gewichtung von Partikeln eines Partikelfilters besonders geeignet für eine Implementation auf der GPU.

1.2 Zielsetzung

Zielsetzung dieser Arbeit ist die Implementation eines Tracking Systems für Kameraposition mit einem Partikelfilter anhand von verschiedenen Bewertungsfunktionen für die Berechnung der Kameraposition. Dabei soll die Kameraposition aus der Position eines zu trackenden Markers abgeleitet werden. Das Tracking mit den verschieden Likelyhood-Funktionen soll mit dem zu implementierenden Partikelfilter evaluiert werden. Des Weiteren soll geprüft werden, welche Kombinationen der jeweiligen Funktionen besonders geeignet sind für ein robustes und schnelles Kamera Tracking mit einer Stereokamera.

1.3 Aufbau der Bachelorarbeit

Die Bachelorarbeit ist gegliedert in Einführung, Stand der Wissenschaft, Grundlagen, Konzeption, Implementation, Evaluation und Fazit. Im Kapitel Stand der Wissenschaft werden zuerst grundlegende Trackingverfahren für Marker- oder Templatetracking erwähnt und vertieft Implementationen erläutert die einen Partikelfilter benutzen. Im Kapitel Grundlagen wird der Partikelfilter Algorithmus erläutert, sowie ein Überblick zu CUDA. Im

Kapitel Konzept wird die geleistete Vorarbeit für die Implementierung und Evaluation aufgezeigt. Das Kapitel Implementation beschäftigt sich mit der Implementation eines Kameratrackers mit einem Partikelfilter. Es wird zusätzlich auf die Implementationd der Likelyhood-Funktionen in CUDA eingegangen, sowie auf das Rendering in OpenGL. Im Kapitel Evaluation wird dar Implementierte Kameratracker in einer virtuellen als auch realen Umgebung evaluiert. Im selbigen Kapitel werden dann Rückschlüsse über das Verhalten der Likelyhood-Funktionen anhand der gewonnen Daten gezogen. Zum Schluss folgt ein Fazit im gleichnamigen Kapitel. Für die Struktur der Bachelorarbeit wurde sich an der Diplomarbeit von David Münch orientiert [Mue].

2 Stand der Wissenschaft

Das Kapitel beginnt mit einem kurzen Überblick über die bekanntesten Verfahren von Tracking Systemen für Kameras. Anschließend werden einige Tracking Systeme mit Partikelfilteralgorithmen präsentiert und deren Besonderheiten erläutert. Da der Fokus der Arbeit auf der Evaluation von Likelyhoodfunktionen liegt, werden nur die Algorithmen näher behandelt, die auf Partikelfiltern basieren.

Koller [Kol] stellte als Erstes ein Trackingverfahren von Kamerapositionen auf Basis eines Planaren Markers vor. Das Verfahren benutzt Liniensegmente im Bild, um den Marker mit einem erweiterten Kalman Filter zu erkennen. Dieses Verfahren wurde im ARToolKit aufgegriffen und fand in vielen Marker basierten Tracking Systemen Verwendung. Siehe z. B. [Abaw]

Das Tracken der Kamera anhand von natürlichen Eigenschaften bei dreidimensionalen Objekten ist ein weiteres Feld im Visual-Based Tracking Bereich. Besonders beliebt sind Algorithmen mit Kantenerkennung. Das wohl bekannteste darunter ist RAPiD [Har]. Die Idee bei RAPiD ist Kontrollpunkte auf einem 3D-Model zu setzen, die mit hoher Wahrscheinlichkeit mit denen aus einem Kantenbild identisch sind.

Für Template Tracking sei noch der Algorithmus von Lucas und Kanade [Luc] erwähnt, der durch Deformationen eines zweidimensionalen Templates den optischen Fluss berechnet und diesen verwendet um das Template zu tracken.

2.1 Kantentracking mit Partikelfilter

Klein und Murray [Kle] entwickelten ein Kantentrackingsystem von 3D-Objekten auf der Grundlage eines Partikelfilters. Ihr Verfahren basiert im wesentlichen auf dem CONDENSATION Algorithmus von M. Pupilli und A. Calway [Con]. Bei dem CONDENSATION Algorithmus handelt es sich jedoch um einen Kantentracker für Kameraposen, der sich auf den zweidimensionalen Raum beschränkt. Der Algorithmus von Klein erweitert diesen hingegen in die dritte Dimension. Dies wurde erreicht, indem man die von 3D-Objekten durch Selbstverdeckung nicht sichtbaren Kanten im Kantenbild entfernen konnte, die bei M. Pupilli und A. Calway beibehalten wurden und somit nur sehr simple 3D-Objekte ohne Selbstverdeckung getrackt werden konnten. Um jene Kanten erfolgreich erfassen zu können wurde der Tiefenbuffer von OpenGL eingesetzt. Das Gewicht w_i jedes Partikels gewann man über mehrere Stufen:

- Rendern des 3D-Objektes aus der Kameraposition des Partikels in den Tiefenbuffer.
- Kanten werden gezeichnet, wenn sie den Tiefentest bestehen.
- Sobel Transformation des Kamerabildes mit anschließender Optimierung der Kantenstärke
- Berechnung des Gewichts w_i durch die Likelihood-Funktion

$$w_i = Likelyhood(X_i^-)$$

Konkret handelt es sich bei der Likelyhood-Funktion (1) um folgende natürliche Exponentialfunktion

$$Likelihood(X_i^-) \propto exp(k\frac{d_i}{v_i}) \quad (1)$$

X_i^- beschreibt dabei den Partikel i bei Frame t aus einem normalisierten Partikel Set $\{X_1^- \dots X_N^-\}$. Die Anzahl aller sichtbaren Kantenpixel eines Partikels ist die Variable v und bei der Variablen d handelt um die Differenz von Kantenwerten der Partikel und den Messwerten. Der Exponent wird des weiteren mit einer Konstanten k multipliziert um zu beeinflussen wie stark die Gewichtung von besonders guten Partikeln ausfällt. Um zu verhindern, dass Partikel, die wenige Kantenpixel haben, dennoch gut bewertet werden, wurde die Division von d durch v gewählt.

2.1.1 Facetracking auf Stream Multi-Prozessoren

Lozano und Otsuka [Loz] stellten in ihrem Paper das erste Gesichtstrackingsystem vor, dass mithilfe eines Partikelfilters auf der Stream Prozessorarchitektur von NVIDIAS CUDA arbeitet. Ihr Gesichtstrackingsystem schafft eine bis zu 10-fache Beschleunigung der Rechenzeit gegenüber anderen ähnlichen Echtzeitsystemen für Gesichtserkennung und Gesichtsverfolgung in Videoaufnahmen. Das Gesichtstemplate wird aus einem 2D-Bild mit dem Viola and Jones Algorithmus [Vio] extrahiert. Das Gesichtstemplate wird weiter auf markante *feature-points* q des Gesichts reduziert. Die *feature-points* beeinhalten Position, Tiefendaten und Normalen, welche auf zwei Texturen gespeichert werden, die auch *height-map* und *normal-map* genannt werden. Durch das Reduzieren des Gesichtstemplates auf *feature-points* müssen weniger Pixel der Partikel bei der Auswertung der Likelyhood-Funktion in Betracht gezogen werden. Die Extraktion der *feature-points* stellt den ersten Schritt des Algorithmus dar und wird auf der CPU ausgeführt.

Der zweite Schritt ist es den jeweiligen Zustand x_t eines jeden Partikels zu bestimmen. Der Partikelfilteralgorithmus wird in **??** näher erläutert. Hier werden nur die von Lozano verwendeten Zustandsparameter für ihr Gesichtstrackingsystem erwähnt und die verwendete Likelyhood-Funktion.

Partikel haben den Zustand x_t^j, wobei j die ID des Partikels ist und t der Zustandszeitpunkt. Der Zustand x_t beinhaltet folgende Parameter:

$$x_t = (T_x, T_y, S, R_x, R_y, R_z, \alpha)$$

Dabei beschreibt T_x und T_y die Translation in x- und y-Richtung, S eine Skalierung, R_x, R_y und R_z eine Drehung um die x-, y-, und z-Achsen. Die α Variable verändert die Helligkeit des Partikels.

Das Gewicht eines Partikels w_i errechnet sich über die Likelyhood-Funktion in (2)

$$w_i \propto exp(k\frac{1}{\varepsilon_i} \tag{2}$$

Likelyhood-Funktion (**??**) setzt sich aus drei Teilen zusammen:

$$\varepsilon = \frac{1}{N}\sum_{m=1}^{N} p(k(J(q_m), I(p_m)))$$
$$k(J, I) = frac\alpha * I - JJ$$
$$p(k) = \frac{k^2}{k^2+1}$$

Die Funktion $J(q_m)$ ist die Intensitätsfunktionen der *feature-point* p_m des Videobildes an der Stelle $m = 1 \ldots N$ mit N als Anzahl der gesamten *feature-points*. Die Intensitätsfunktion beschreibt, wie stark ein *feature-point* in die Gewichtung des Partikels einfließt. Entsprechend gilt dasselbe für $I(p_m)$. p_m sind dabei die simulierten *feature-points* des Partikels m mit $x_t^m - 1$. Hervorzuheben an der Implementation der Likelyhood-Funktion sind die Intensitätsfunktionen, die dafür sorgen, dass markante *feature-points* besonders stark in die Gewichtung des Partikels einfließen.

2.1.2 Tacking von Bewegungen anhand von Tiefendaten

Obwohl das Tracken von Bewegungen nicht direkt mit dem Tracken einer Kameraposition vergleichbar ist, sei hier trotzdem das Paper von B. Penelle und O. Debeir [Pen] erwähnt. Aufgrund ihres Ansatzes sich nur auf Tiefendaten der Kinectkamera zu beschränken, können ihre Ergebnisse auf weitere Trackingsysteme angewandt werden, die nur Tiefendaten nutzen. Da

ihr System die Bewegung anhand von 3D-Modellen trackt lässt es sich auch auch auf das Tracken von Kamerabewegungen übertragen.

Die Versuchsumgebung bestand aus drei fest positionierten Kinectkameras und einem trackenden 3D-Model. Hierbei sei erwähnt, dass die Versuche in einer simulierten 3D-Welt stattfanden und nicht unter realen Bedingungen mit echten Kameras getestet wurden.

Ähnlich dem Verfahren von [Loz] wurde ein Template, hier 3D-Model eines Menschen, anhand von Partikel, die jeweils einen Zustand x_t mit Translationen und Rotationen beinhalten, in OpenGL transformiert und gerendert. Für das Experiment wurde die Bewegung eines menschlichen Beines simuliert, wodurch sich der Zustand x_t pro Partikel auf 15 Freiheitsgrade ausweitete. Die Tiefenbilder der Partikel wurden anschließend auf der GPU mit Einsatz von CUDA gewichtet. Für den Vergleich der Tiefendaten der Partikel $M_c(x_t)$ mit den Messdaten O_c nahm man die Funktion aus (3) .

$$\delta_x(x) = \sum_{u,v} min(|O_c - M_c(x)|, \tau) \tag{3}$$

τ dient als Schwellenwert der Funktion. Die entsprechende Likelyhood-Funktion für die Partikel definiert sich durch:

$$\begin{aligned} f_c(x) &= exp(-\frac{\delta_c(x) - \delta_c^-}{\sigma_c}) \\ \delta_c^- &= min\delta_c(x) \\ \delta_c^+ &= max\delta_c(x) \\ \sigma_c &= k\frac{\delta_c^- - \delta_c^+}{ln(f_c^-)} \end{aligned} \tag{4}$$

Die Variable σ_c soll dafür sorgen, dass die Likelyhood-Funktion stärker ausschlägt bei besonders guten bzw. schlechten Partikeln. Die Testdaten ergaben zwar, dass das Tracking funktionierte, aber noch nicht Echtzeitfähig ist ohne an Genauigkeit zu verlieren. Dafür verantwortlich gemacht wurde die hohe Zahl an Freiheitsgraden des Zustandsmodels.

2.1.3 3D-Model Tracking mit Partikelfilter

In [Ped] wurde ein 6-DoF Model-Based Tracking für Festkörper vorgestellt, das anhand eines Partikelfilters auf Kantenbildern arbeitet ähnlich wie in [Kle]. Um die korrekte 6-DoF Ausrichtung eines Festkörpers zu bestimmen, muss der View-Space alle sechs Freiheitsgrade von Translationen und Rotationen abdecken. Ein solches Unterfangen ist äußerst rechenaufwendig.

Durch die Implementation eines Partikelfilters konnte der zu rendernde View-Space eingeschränkt werden. Für die Eingabe benötigt der Ansatz von [Ped] lediglich ein akkurates 3D-Model des zu trackenden Festkörpers. Die Position der Kamera wurde pro Partikel simuliert und das Model wurde anhand der simulierten Kameraposition gerendert. Im Folgenden wurden die Texturen zu binären Kantenbildern transformiert und danach gewichtet. Die Likelyhood-Funktion für die Gewichtung der Partikel war eine binäre AND Operation auf jedem Pixel der Kantenbilder. Anstatt in einem Renderdurchgang alle Partikel zu rendern wurden bis zu fünf Durchgänge benutzt, um ein finales Partikel Set zu erhalten. Im jeden Durchgang fand ein Sampling, Gewichtung und Resampling statt. Die Varianz pro Durchgang wurde stetig reduziert, um beim ersten Durchgang eine möglichst breite Verteilung zu erhalten und stetig die Verteilung einzuengen um an Präzesion zu gewinnen. Aus der Evaluation der Arbeit ging hervor, dass Tracking mit einem Partikelfilter besonders robust gegenüber teilweisen Verdeckungen der zu trackenden Festkörper ist. Jedoch muss ein aufwendiges Postprocessing der Texturen und Messwerte stattfinden, um ungewollte Kanten zu entfernen

2.1.4 RGB-D Tracking

Choi und Christensen [Cho] erreichten mit ihrer Implementation eines Partikelfilters für RGB-D Kameradaten für das Tracken von Festkörpern echtzeitfähige Laufzeiten mit sehr guten Ergebnissen. Anstelle einer Kamera mit lediglich einem Sensor für Farben wie sie unter anderem in [Ped] für Kantendetektion benutzt wurde, kam eine RGB-D Kamera zum Einsatz, die auch Tiefendaten messen konnte.

Mit dem zusätzlichen Tiefenkanal ergaben sich neue Messdaten für Bilder, die bei Farbkameras ohne weiteren Sensoren nicht gegeben waren. In ihrer Implementierung des RGB-D Trackingsystems beschränkten sich Choi und Christensen für die Likelyhood-Funktion auf die Parameter:

- 3D-Punkt Koordinaten
- Farbe der Punkte
- Normalen der Punkte

Im Gegensatz zu der Arbeit von Lenz [Ped] erreichte man hier eine Optimierung des Partikelfilters durch CUDA. Um den Engpass der beim Mapping von in OpenGL gerenderten Partikel nach CUDA zu vermeiden, wurde die Partikel stattdessen Off-Screen in ein FBO gerendert. Dadurch gelang es den Engpass beim Mapping zu umgehen und die Likelyhoodberechnung auf der GPU auszuführen.

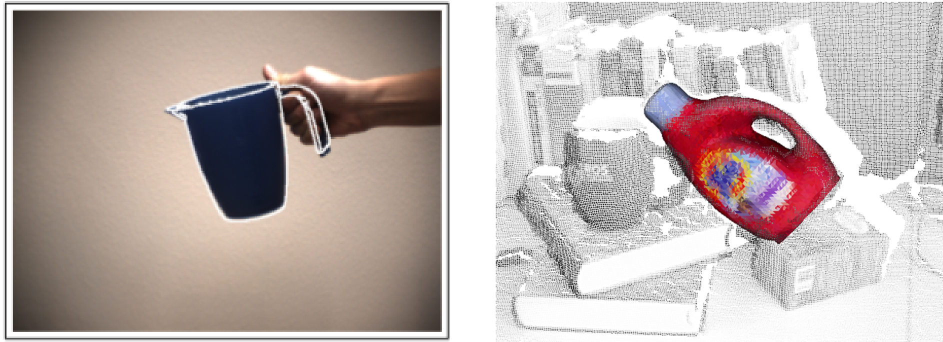


Figure 1: Links: Partikelfiltertracking anhand von Edges, Rechts: Partikelfiltertracking mit einer RGB-D Kamera

3 Grundlagen

3.1 Partikelfilter

Der Partikelfilter ist eine nicht-parametrische Version des Bayesfilters, welcher auf den Grundlagen der Bayesischen Logik, benannt nach den englischen Mathematikers Thomas Bayes, entstand. Um den nachfolgenden Partikelfilter Algorithmus in Abschnitt 3.1.2 besser nachzuvollziehen, zu können, wird in folgenden Abschnitt 3.1.1 zuerst der Bayesfilter erläutert.

3.1.1 Bayes Filter

Der Bayesfilter errechnet anhand eines Zustands x_t, Regulierungsdaten u_t und Messdaten z_t zum Zustandszeitpunkt t eine Verteilungsfunktion $bel(x_t)$ [Thu]. In Listing 1 wird der Bayesfilter in Pseudocode aufgeführt.

```
Algorithm Bayes_filter(bel(x_t-1), u_t, z_t){
for(i = 0; i < x_t.size(); i++){
bel_overline(x_t.at(i));
bel(x_t.at(i));
}
return bel(x_t)
}
bel(x_t){

mean( p(z_t , x_t) * bel_overline(x_t) )
}
bel_overline(x_t){

integral( p(x_t , u_t, x_t-1 ), bel(x_t-1)
}
```

Listing 1: Bayes Algorithmus Pseudocode

Man kann an der Eingangsvariable $bel(x_t{-}1)$ erkennen, dass der Bayesfilter-Algorithmus die Ausgangsvariable $bel(x_t)$ zum Zustandszeitpunkt t rekursiv über den vorangegangenen Zustandszeitpunkt $t-1$ aus $bel(x_t-1)$ errechnet. Die Verteilung $\overline{bel}(x_t)$ in Listing 1 wird auch als *Prediction* des Bayesfilters bezeichnet. Im ersten Schritt wird $\overline{bar}(x_t)$ über das Integral eines Produktes von der Verteilung $bel(x_t-1)$ und der Wahrscheinlichkeit $p(x_t|u_t, x_t-1)$ integriert.
Die Funktion $p(x_t|u_t, x_t{-}1)$ beschreibt die Wahrscheinlichkeit für den Übergang von Zustand $x_t{-}1$ in den Zustand x_t , welcher durch die Regulierungsvariable u_t zum Zeitpunkt t herbeigeführt werden könnte. Die Regulierungsvariablen u_t kann man sich z.b. als eine Aktion wie die Bewegung der Kamera vorstellen. Es handelt sich somit um Annahmen über mögliche Zustandsänderungen. Der zweite Schritt, auch *Measurement update* genannt, wird die gesuchte Verteilung$bel(x_t)$ berechnet. Dafür wird erneut ein Produkt gebildet, dass aus der Wahrscheinlichkeitsfunktion $p(z_t|x_t)$ und der aus dem ersten Schritt gewonnene Annahme $\overline{bar}(x_t)$ besteht. Die Wahrscheinlichkeitsfunktion $p(z_t|x_t)$ beschreibt die Wahrscheinlichkeit, dass die Messung z_t mit dem Zustand x_t übereinstimmt. Somit ergibt das Produkt von $p(z_t|x_t)$ und $\overline{bar}(x_t)$ die gesuchte Verteilung $bel(x_t)$. Das Integral der resultierenden Verteilung muss allerdings nicht zwangsläufig eine Wahrscheinlichkeitsverteilung sein. Deswegen wird das Produkt zusätzlich normiert.
Die Verteilung $bel(x_t)$ beschreibt, wie wahrscheinlich es ist, dass die angenommene Verteilung $\overline{bar}(x_t)$ durch den Einfluss der Regulierungsvariablen u_t auch nach einer möglichen Messung aus $p(z_t|x_t)$ übereinstimmt.

3.1.2 Partikelfilter

Der Partikelfilter ist eine alternative Umsetzung des Bayesfilter für nichtlineare Filterverfahren [Thu]. Da es kein triviales Problem ist die Verteilungsfunktion des Bayesfilters zu bestimmen, bedienen sich Partikelfilter des Monte Carlo Ansatzes [Ray]. Die Idee ist Verteilungen stochastisch anzunähern nach dem Prinzip des Gesetzes der Großen Zahlen. Die Verteilung $bel(x_t)$ aus Abschnitt 3.1.1 wird durch eine Menge aus Zuständen approximiert, die aus der Verteilung $bel(x_t-1)$ gezogen werden sollen und anschließend mit einem ein Rauschen gestreut werden, ähnlich der Regulierungsvariablen u_t aus dem Bayesfilter Algorithmus. Das Rauschen wird durch Zufallswerte repräsentiert, die in der Regel aus einer Wahrscheinlichkeitsverteilung wie der Normalverteilung zufällig entnommen werden. Die Menge an Zuständen werden auch Partikel X_t genannt. Der Vorteil einer solchen Annäherung ist, dass man sich bei der Modellierung eines Problems nicht mehr auf lineare Funktionen beschränken muss.

Um zu erreichen, dass die neue Verteilung $bel(x_t)$ durch die Streuung der Partikel weiterhin in Relation mit ihrer vorherigen Verteilung $bel(x_t{-}1)$

steht, werden Partikel bewertet. Die Bewertung übernimmt eine sogenannte Likelyhood-Funktion. Anschließend wird eine neue Menge von Partikel anhand ihrer Bewertung konstruiert. Wenn man den Bayes-Algorithmus betrachtet entspricht die *Prediction* der Gleichung (5) aus [Thu] der Streuung der Partikel mit

$$x_t^m \sim p(x_t|u_t, x_t^m - 1)$$

. Im Folgenden auch als *Sampling* bezeichnet.

$$\overline{bel}(x_t) = \int p(x_t|u_t, x_t - 1)bel(x_t - 1)dx_t - 1 \tag{5}$$

Bei der Funktion $p(x_t|u_t, x_t^m - 1)$ handelt es sich um die *State-Transition*-Funktion, einer Zufallsverteilung mit u_t als Regulierungsfaktor. Die Variable $m \in \{1 \dots M\}$ beschreibt die Position eines Partikels des Partikelsets X_t.

Das *Measurement update* des Bayesfilers setzt sich beim Partikelfilter aus zwei Schritten zusammen.

- Die Partikel werden anhand einer Likelyhood-Funktion und einer Messung z_t verglichen und gewichtet.

  $$w^[m]_t = p(z_t|x_t^m)$$

  Daraus bekommt man eine neue Menge $\overline{X_t}$ mit dem Tupel x_t^m, w_t^m Im weiteren Text nur noch als *Likelyhood* bezeichnet.

- Das Set $\overline{X_t}$ wird normalisiert und es werden Partikel anhand ihrer Gewichtung gezogen und zum neuen Set X_t hinzugefügt. Das Ziehen wird solange wiederholt bis X_t genau so viele Partikel hat wie $\overline{X_t}$. Das Ziehen eines neuen Sets von Partikeln in Anbetracht ihrer Gewichtung aus dem Partikel Set $\overline{X_t}$ wird als *Resampling* bezeichnet.

Dank dem Resampling arbeitet der Algorithmus nach dem Prinzip *survival of the fitest*. Die gut bewerteten Partikel bleiben im Set und die schlecht bewerteten fallen (hoffentlich) raus. Dadurch wird auch die zweite Gleichung des Bayesfilters (6) erfüllt.

$$(bel)(x_t) = \eta p(z_t|x_t^m - 1)\overline{bel}(x_t) \tag{6}$$

Da die Partikel aus dem Set $X_t - 1$ gezogen werden, behält der Algorithmus seine rekursive Form und das Resampling zwingt die Partikel in ihre ursprüngliche Verteilung zurück. Das Resampling kann man auf verschiedene Arten umsetzen. Eines der gängigsten Verfahren ist das Multinomial Resampling das in Abschnitt 5.7.1 näher erläutert wird.

3.2 Cuda

Seit den 2000er hat die Rechenleistung und Speicherbandbreite von Grafikkarten (GPU's) konstant zugenommen und die der CPU's nahezu exponential überholt. Siehe dazu Abbildung 2 für die Floating-Point Operationen auf der GPU und CPU und Abbildung 3 für die Speicherbandbreite über die Jahre 2003 bis 2015 [Cuda]. Dadurch wuchs der Wunsch diese Rechenleistung auch außerhalb von Grafikanwendungen zu nutzen. Obwohl es auch schon vor NVIDIAS CUDA möglich war die Grafikkarte für *general-purpose computing* auf der GPU (GPGPU) mit API's wie OpenGL oder DirectX einzusetzen, musste man seine Anwendungen dennoch als Grafikanwendungen maskieren.

Am 08. November 2006 hat NVIDIA die Geforce 8 Serie mit integrierter CUDA Architektur. Mit CUDA führte NVIDIA eine Platform ein, die Programmieren ermöglicht die hohe Rechenleistung und Speicherkapazität von Grafikkarten direkt über eine high-level C ähnlicher Sprache anzusprechen. Der CUDA Compiler NVCC kann C/C++ Code sowohl für die CPU, als auch C und mit Einschränkungen C++ Code für die GPU kompilieren. Dadurch ist man nicht mehr auf Shadersprachen wie z. B. GLSL von OpenGL angewiesen, um die Grafikkarte anzusprechen.

3.2.1 CUDA Architektur

Die Grafikkarten von NVIDIA mit CUDA Unterstützung bauen auf einer sogenannten Single Instruction, Multiple Threads (SIMT) Architektur auf. So sind mehrere Stream Multi-Prozessoren verbaut, die jeweils weitere Stream-Prozessoren beinhalten [Brow]. Wieviele Prozessoren onboard sind, ist von Grafikkarte zu Grafikkarte unterschiedlich.

Die Stream Multi-Prozessoren haben den IEEE Standart für single-precision floating-point Arithmetik [Cuda2]. Zusätzlich verfügen die GPU's eine Global Memory Einheit auf die jeder Prozessor Lese- und Schreibzugriff hat. Weitere Speicher auf der GPU sind die Constant-, Texture- und Shared-Memory auf die, wegen ihrer Eigenheiten, von welchem Block bzw. Thread sie angesprochen werden können, erst im Laufe des Kapitels eingegangen wird. Ein Model der CUDA Hardware ist in Abbildung 4 aufgeführt und wurde entnommen aus dem Paper von [Yang] Wegen der Skalierbarkeit des CUDA Programmiermodels müssen sich Softwareentwickler nicht direkt darum kümmern, auf welcher Prozessoreinheit der GPU der CUDA Code ausgeführt wird. Man hat es hingegen mit einer Abstraktion in Grids, Blocks und Threads zu tun, die dann jeweils von CUDA automatisch auf die zugrundeliegenden Multiprozessoren aufgeteilt werden. In Abbildung 5 ist eine Illustration der Aufteilung in Grids, Blocks und Threads.

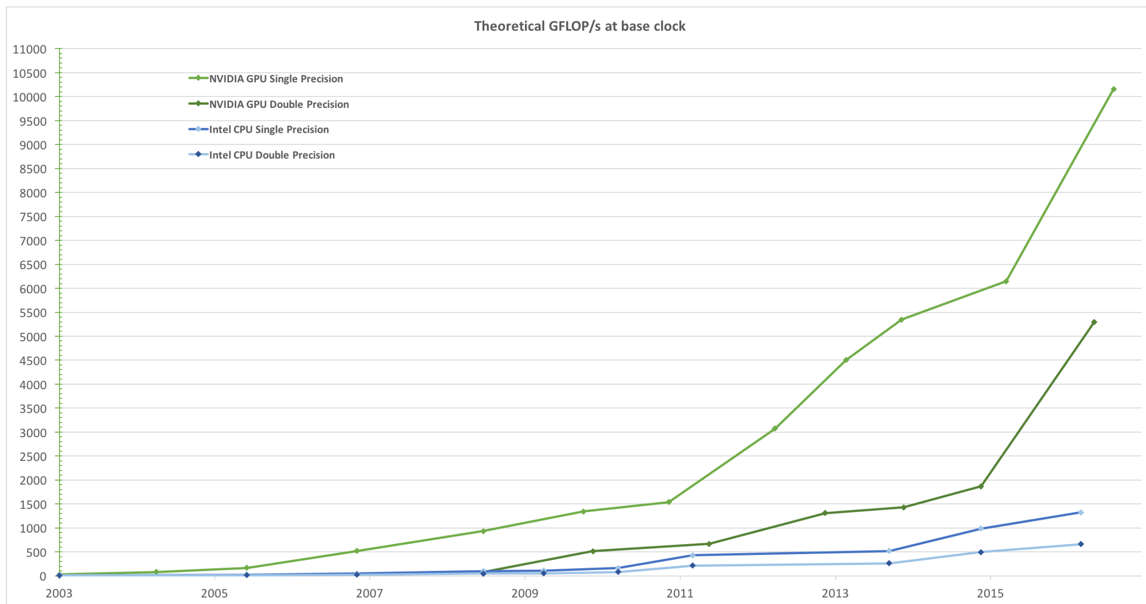


Figure 2: Floating-Point Operationen pro Sekunde auf der GPU und CPU

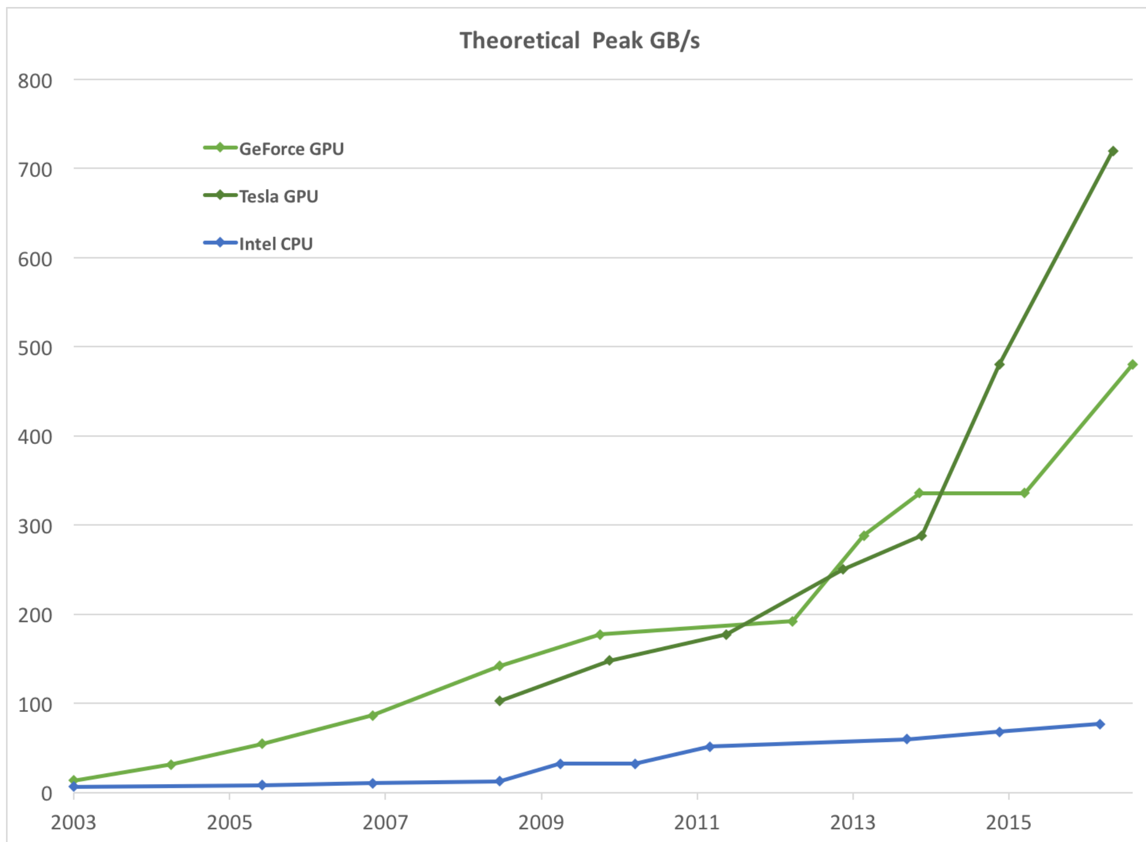


Figure 3: Speicherbandbreite für GPU und CPU

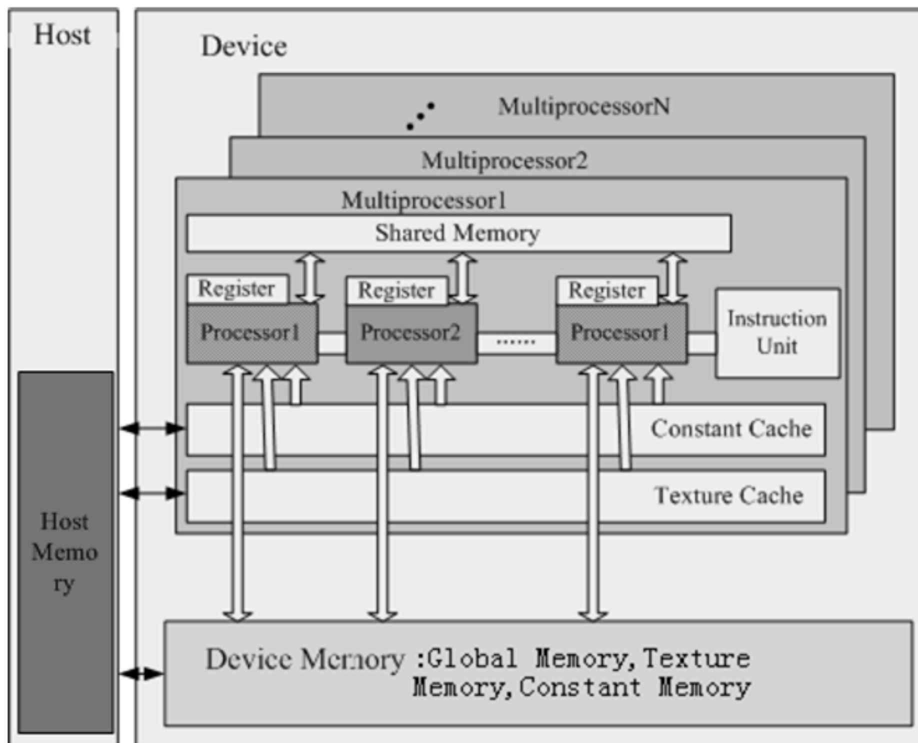


Figure 4: CUDA Hardware Model

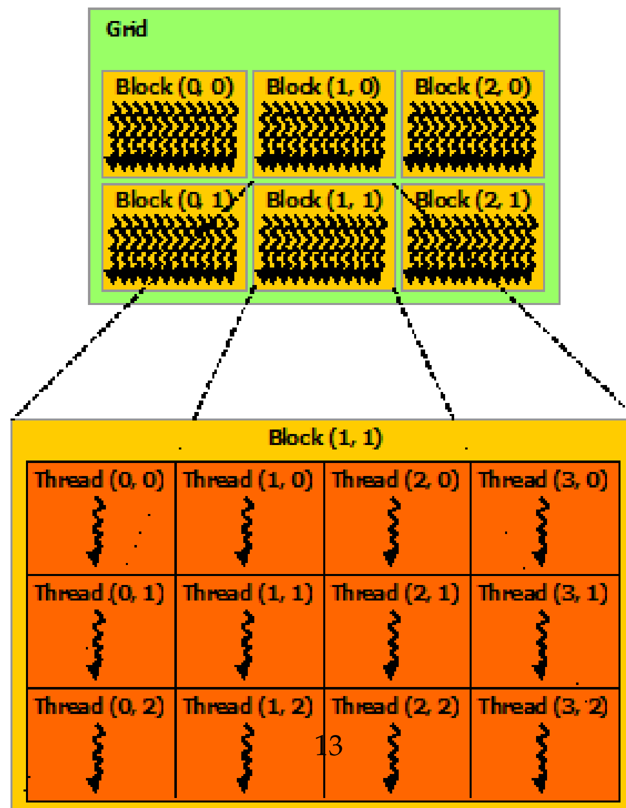


Figure 5: Illustration von einem Grid mit 6 Blocks und jeweils 12 Threads pro Block

3.2.2 CUDA Programmiermodel

CUDA's Programmiermodel ist auf ein hohes Maß an Parallelität ausgelegt. Um die Leistung der Stream Multi-Prozessoren auszunutzen, aber trotzdem die low-level Hardware Deteils zu verbergen, bedient sich CUDA einer Abstraktion in Grids, Blocks und Threads. Grids haben drei Dimensionen und können in $(2^3 1) - 1$ Blocks in x-Richtung und 65535 Blocks in y- und z-Richtung aufgeteilt werden [Cuda]. Blocks können weiter in Threads aufgeteilt werden und haben ebenfalls drei Dimensionen. Jeweils 1024 in x- oder y-Richtung und 64 in z-Richtung. Es bleibt dem Programmierer offen auch tatsächlich alle Dimensionen auszunutzen oder sich auf bestimmte Kombinationen zu beschränken. Siehe Abbildung 5.

Bevor eine C Funktion, auch Kernel genannt, überhaupt auf der GPU ausgeführt werden kann, muss sie zuerst vom Host aufgerufen werden. Beim Host handelt es sich hierbei um den Code, der auf der CPU läuft. Das Pendant zum Host auf der GPU nennt sich Device. Der Aufruf eines Kernels erfolgt über den Syntax **«<...»>** , welcher vor einer Kernelfunktion steht. In diese Klammern kommt dann jeweils die Aufteilung der Threads auf Blocks und Grids und optional den einzelnen Blocks zugewiesener Shared-Memory. Auf das Shared-Memory können die Threads ähnlich dem Global-Memory zugreifen, jedoch nur die Threads aus dem zugehörigen Block. Der Vorteil dabei ist, dass die Zugriffe ca. 40x schneller sind als auf Global-Memory [Yang]. Um eine Funktion als Kernelfunktion zu deklarieren, erwartet CUDA, dass man das Keyword **__global__** vor den Funktionskopf setzt. Ähnlich verhält es sich bei Variablen, die dem Shared-Memory zugewiesen werden sollen. Diese erhalten das Keyword **__shared__** .

Um in einem Kernel auf die einzellnen Threads zuzugreifen, verfügt jeder Thread über eine einzigartige ID. Diese ID bekommt man durch die CUDA-C spezifische Variable **threadIdx** . Ebenfalls vorhanden ist die Variable **blockIdx** für den Zugriff auf die einzigartigen Block-ID's. Die Variablen entsprechen ihrer jeweiligen Aufteilung der Blocks und Threads in die drei Dimensionen. Durch die Kombination von Thread- und Block-ID's gelingt ein spezifischer Zugriff auf einen bestimmten Thread innerhalb eines Blocks. Da Threads parallel und unabhängig voneinander ablaufen, es aber für viele Applikationen dennoch erforderlich ist, dass Threads koordiniert auf bestimmte Speicherbereiche zugreifen, um komplexe Probleme zu lösen, bietet CUDA die **__syncthreads()** Funktion für die Synchronisation von Threads innerhalb eines Blocks. Nach dem Aufrufen dieser Funktion werden Threads innerhalb eines Blockes gezwungen auf alle anderen Threads aus dem gleichen Block zu warten bis jene ebenfalls zum Aufruf von **__syncthreads()** kommen. An dieser Stelle wird die Relevanz des Shared-Memorys auch klarer, abseits von den beschleunigten Zugriff-

szeiten. Da __syncthreads() nur die Threads innerhalb ihres zugehörigen Blockes synchronisiert, können Threads auch nur durch ihren dem Block zugeordneten Shared-Memory kooperieren. Kernels, die lediglich die Global-Memory der GPU nutzen können zu Seiteneffekten führen beim Lesen und schreiben, da durch die Unabhängigkeit der parallel ablaufenden Threads ein erwartungskonformer Zugriff nicht sichergestellt werden kann.

Ein typischer CUDA Programmverlauf, entnommen aus [Yang], gestaltet sich wie folgt:

- Daten von Host-Memory auf Device-Memory kopieren.
- Host steuert den Kernel Aufruf:
  a Threads, Blocks und Shared-Memory zuweisen.
  b (Optional) Daten von Global-Memory zur Shared-Memory übertragen
  c Kernel ausführen
- Bearbeitete Daten vom Device- auf Host-Memory kopieren

4 Konzeption

4.1 Likelyhood-Funktionen

Die zu evaluierenden Likelyhood-Funktionen wurden anhand der entsprechenden Bilddaten der Stereokamera gewählt. Konkret handelt es sich dabei um Farbwerte, Tiefenwerte und Normalen von Flächen. Zusätzlich wurden auch Kantenbilder mit dem Sobel Operator erstellt für Likelyhood-Funktionen auf Kantenbildern. Des Weiteren wurden Likelyhood-Funktionen getestet, die aus Kombinationen der vorher genannten Daten bestehen. Es wurden folgende Kombinationen gewählt:

- Farb- und Tiefenwerte
- Farbwerte und Normalen
- Tiefenwerte und Normalen

4.2 Testumgebungen

Der Fokus der Arbeit liegt auf der Evaluation von Likelyhood-Funktionen von Partikelfiltern zum Tracking der Kameraposition. Es wäre eine Möglichkeit gewesen bei der Implementation sich lediglich auf eine Testumgebung für die Stereokamera zu beschränken. Um jedoch die spezifischen Eigenheiten der Likelyhood-Funktionen, unabhängig von den Limitationen die eine Kameraaufnahme mit sich bringt, betrachten zu können wurde zusätzlich eine virtuelle Testumgebung konzipiert.

4.2.1 Virtuelle Testumgebung

Als Versuchsumgebung wurde ein halboffener Raum konstruiert, der ab hier als "Karton" referenziert wird. Die Abmessungen der Wände betragen 60 x 40 cm. Die Initiale Pose der Kamera beginnt bei der Position (0, 0, 2) und ist 12 cm von der hintersten Wand entfernt. Der Marker wurde an der Position (-1, 1, -6) platziert. Zwischen Marker und Kamera liegt ein Hindernis von 4 cm Breite und einer Höhe von 60 cm. Die Position des Hindernisses liegt bei (-2.5, 0, -4). Abbildung eine zeigt ein Rendering der Testumgebung "Karton".

Figure 6: Rendering der Testumgebung "Karton"

4.3 Marker

Um die Kameraposition überhaupt tracken zu können müssen die Partikel gewichtet werden. Die Gewichtung erfolgt anhand von Messwerten der Stereokamera. Da die Stereokamera primär zweidimensionale Bilddaten wie Farben liefert, muss der Zustand des Partikels ebenso mit vergleichbaren Datentypen repräsentiert werden. Aus diesem Grund wurde ein Marker verwendet. Der Marker erfüllt den Zweck eines messbaren Bildele-

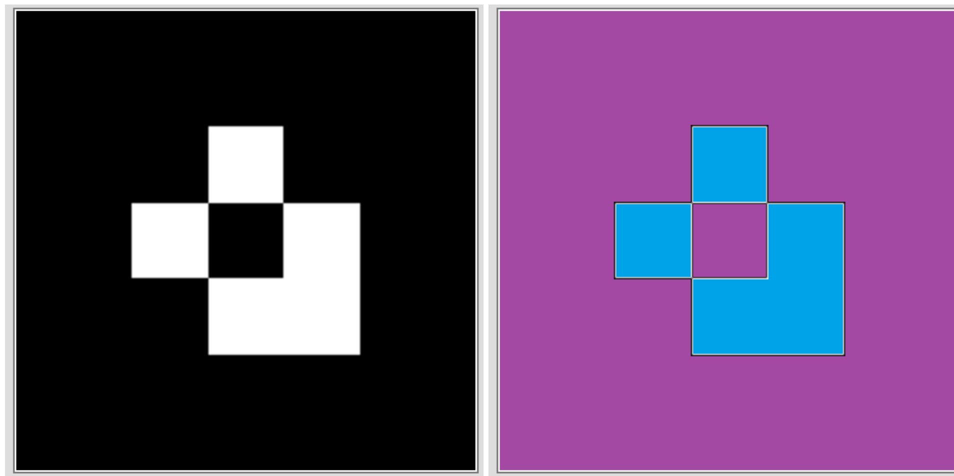


Figure 7: Links: Schwarzweiß Marker. Einsatz: Kantenbilder, Rechts:Gefärbter Marker, Einsatz: HSV-Farbbilder

ments in einer Kameraaufnahme anhand dessen Position in der Welt die Kameraposition bestimmt werden kann. Abbildung 7 zeigt die verwendeten Marker.

4.4 Programmverlauf

Der Programmverlauf orientiert der Implementation an der Funktionsweise eines in Abschnitt 3.1.2 vorgestellten Partikelfilters. Abbildung 8 stellt exemplarisch den Programmverlauf mit der Stereokamera und einem Partikelfilter für Farbwerte und Normalen dar. Bei der Implementation mit einer virtuellen Kamera ändert sich lediglich der Input für den Partikelfilter. Statt Aufnahmen der Stereokamera zu benutzen wird eine virtuelle Umgebung gerendert und als Input für den Partikelfilter verwendet.

Bei der Initialisierung werden die Partikel, die ZED Kamera und ein GLFW Fenster initialisiert. Zusätzlich ein Frame Buffer Object (FBO) generiert und jeweils die verwendeten Texturen dem FBO zugewiesen. Für die ZED Kamera genügt es eine OpenGL Textur zuzuweisen, da das ZED SDK die Aufnahmedaten in einem der ZED Kamera spezifischen Format speichert und man auf diese ähnlich einer OpenGL Textur zugreifen kann. Die Textur für die Kamera wird im späteren Programm zur visualisierung des Kamerabildes für den Benutzer auf den Bildschirm gerendert. Auf die einzelnen Datentypen wird im Detail erst in Abschnitt 5 eingegangen.

Nachdem abschließend OpenGL spezifische Parameter gesetzt wurden, beginnt die Render Loop. Beim Sampling wird der Zustand der Partikel

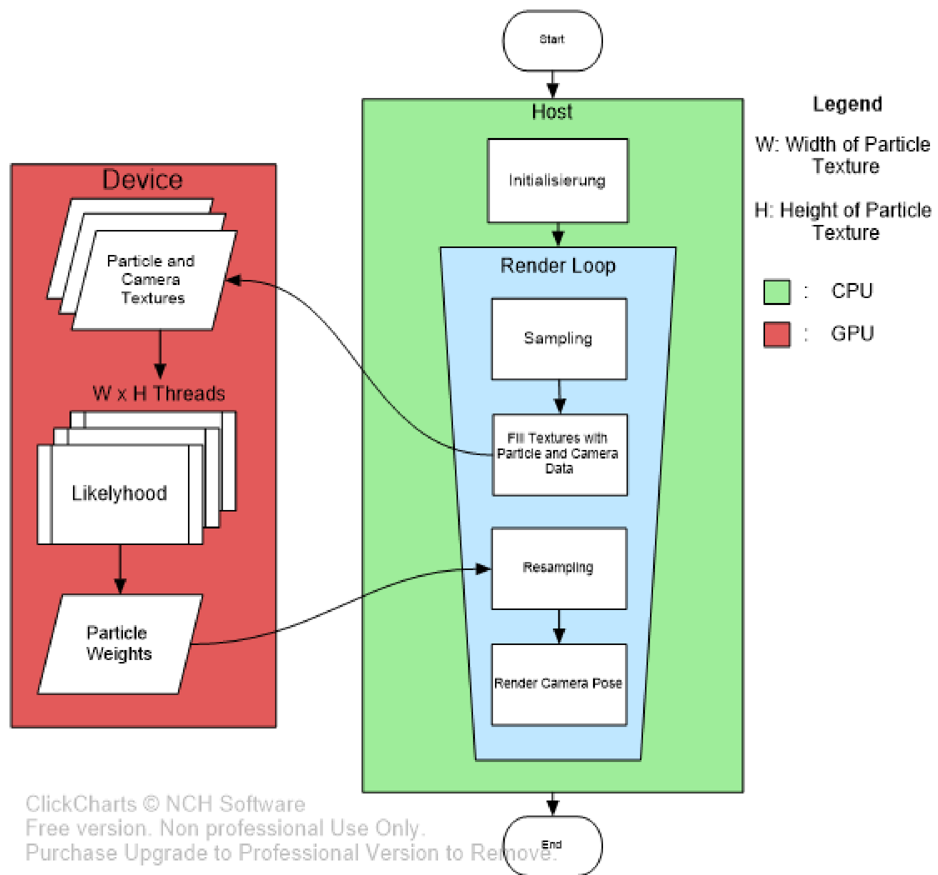


Figure 8: Flow Chart

durch eine Wahrscheinlichkeitsverteilung gestreut. Der Zustand eines Partikels beschreibt die Ausrichtung der Kamera mit den sechs Freiheitsgraden für Translationen und Rotationen.

Anschließend wird der Marker pro Partikel aus den gesampelten Kamerapositionen in die dazugehörigen Texturen in den Framebuffer gerendert. Ebenso wird das Kamerabild abgefangen. An dieser Stelle müssten die Texturen für CUDA referenziert, auf Arrays gemappt, Host Daten auf das Device übertragen werden ect. Es sei der Übersichtlichkeit halber in Abb. 8 darauf verzichtet worden und wird erst in Abschnitt 5 weiter ausgeführt. Zudem wird der Aufruf der Kernels nicht explizit genannt. Dies soll die Aufteilung für Host Prozesse in die Grüne Box und für Device Prozesse in die rote Box implizieren.

Nach dem Aufruf des Kernels befindet man sich im Flowchart auf dem Device (GPU). Jeder Thread bearbeitet ein Pixel der Partikel Textur. Die Pixel werden mit den korrespondierenden Pixeln aus dem aufgenommenen Kamerabild verglichen und anhand einer Likelyhood-Funktion bewertet. Die Gewichte der einzelnen Pixel werden aufaddiert und dem jeweiligen Partikel zugeordnet. Die Partikel Texturen setzten sich aus den gerenderten Partikeln zusammen. Jeder Partikel hat einen eindeutigen Texturbereich, weshalb das Gewicht eines jeden Pixels eindeutig einem Partikel zugeordnet werden kann.

Im Resampling Schritt wird nach der Definition aus 3.1.2 ein neues Partikel Set aus dem im vorherigen Schritt gewichteten Partikel Set erstellt. Die für das Resampling genutzten Verfahren werden in Abschnitt 5.7 vorgestellt.

Als letztes wird der Mittelwert der Zustände der resampelten Partikel gebildet. Dann wird ein einzelner Partikel gebildet durch den Mittelwert der Zustände auf den Bildschirm gerendert. Dieser Partikel stellt die getrackte Markerposition dar. Aus der Modelview Matrix des gemittelten Partikels wird durch Invertierung jener Matrix die Kameraposition bestimmt. Zusätzlich wird das aufgenommene Kamerabild gerendert um die Genauigkeit des Trackings für den Benutzer zu visualisieren.

5 Implementation

5.1 Partikel und Partikelmanager

Partikel werden in der Anwendung als C++ *sructs* repräsentiert, genannt ParticleStruct. Das ParticleStruct beinhaltet das Gewichtung bzw. weight des Partikels und eine 4x4 Modelmatrix für den Zustand des Partikels. Die Modelmatrix beschreibt die Position und Rotation des Markers zum Zeitpunkt t in der Welt mit den sechs Freiheitsgraden

$$X_t = (x_t, y_t, z_t, \alpha_t, \beta_t, \gamma_t)$$

.

Neben dem Partikel wurde eine Klasse PartikelManager implementiert. Diese Klasse übernimmt die Verwaltung des Partikelfilters. In ihr wird das Partikelset im Container vom Typ std::vector gespeichert, dass aus den Partikeln des Typs ParticleStruct besteht. Des weiteren beinhaltet die Klasse Funktionen für das Sampling und Resampling der Partikel, sowie Hilfsfunktionen wie das Normalisieren der Gewichte. Darüber hinaus übernimmt der PartikelFilter die Aufgabe der Initialisierung. Abhängig von der zuvor gewählten Likelyhood-Funktion für den Partikelfilter werden bei der Initialisierung des PartikelManagers die benötigten Shader kompiliert, die FBO's generiert und die passende Renderschleife gewählt. Dafür wird der Benutzer beim Start des Programms aufgefordert über die Konsole einen Modus für den Partikelfilter auszuwählen.

Die Modi der entsprechen Likelyhood-Funktionen können aus Abschnitt 4.1 entnommen werden.

5.2 ZED SDK

Das ZED SDK in Kombination mit der ZED Stereokamera wurde für die Aufnahmen genutzt. Es wurde ein ZED Cam Objekt instanziiert und während jedes Farmes bekam dieses den Befehl das aktuelle Kamerabild an die GPU zu schicken. Das Kamerabild wird als ein internes Matrixformat von ZED gespeichert. In CUDA können diese Matrizen genauso wie Texturen verwendet werden und man kann auf sie über den CUDA Befehl tex2D zugreifen. Es wurden sowohl Farbbildaufnahmen aus dem linken Objektiv der Kamera aufgenommen an CUDA weitergereicht, als auch die Normalen und Tiefendaten. Damit die Partikel aus der gleichen Perspektive gerendert werden konnten, wurde die Projection Matrix der Partikel mit den ZED Stereokamera Intrinsics modifiziert.

5.3 OpenGL

5.3.1 Marker

Beim Marker handelt es sich um ein planares Objekt ähnlich einer Postkarte. Diesen gilt es in einer virtuellen Umgebung zu repräsentieren. Dafür wurde die Klasse Marker erstellt, die Vertices eines Quads und dessen Textur für OpenGL enthält. Um die verschiedenen Positionen des Markers in der Welt zu visualisieren wurde eine Rendermethode implementiert. Die Rendermethode erhält als Input das Partikel Set. Pro Partikel wird ein Marker mit der Modelmatrix des Partikels gerendert. Damit die Marker sich nicht gegenseitig verdecken, sondern einzeln für sich betrachtet werden können, verschiebt man den Viewport für jeden Partikeln. Der Viewport für das Rendering kann mit der OpenGL Methode glViewport festgelegt werden. Als Parameter erwartet die Methode die x- und y-Koordinaten des Viewports sowie Breite und Höhe des Viewports. Die Renderschleife der Marker ist in Listings 2 als Pseudocode gelistet.

```
renderPartikels( vector<ParticleStruct> &particelSet ) {

glBindFramebuffer(GL_FRAMEBUFFER, framebuffer);

        //OpenGl Parameters//

int particleNum = 0;

for(int k = 0; k < textureColumns; k++) {
        for (int j = 0; j < textureRows; j++) {
                glViewport(VIEWPORT_WIDTH * j, VIEWPORT_HEIGHT*k,
        VIEWPORT_WIDTH ,VIEWPORT_HEIGHT);
        shader.use();
        glm::mat4 model = glm::mat4(1.0f);
        shader.setMat4("model", particleSet.at(particleNum).model);

                //Set View and Projection Matrix and draw Marker
        // ... //
        particleNum++;
        }
    }
};
```

Listing 2: Renderschleife Partikel

Die doppelte for-Schleife mit den Greznen *textureColums* und *textureRows* sorgt für die Anordnung der Viewports entlang der im FBO gespeicherten Textur. Auf die generierten Texturen und deren Struktur wird im Abschnitt FBO‘s eingegangen, sowie die Bedeutung von *textureColums* und *textureRows*.

5.4 Framebuffer Objects und Texturen

Framebuffer Objects (FBO) sind benutzerdefinierte Framebuffer, die nicht zwangsweise auf den Bildschirm rendern so wie es der Default Framebuffer von OpenGL macht [OpenGL Wiki] . Die Möglichkeit *off-screen* zu rendern wird oft für *post-processing* genutzt. Beim *off-screen Rendering* wird die Szene statt auf den Bildschirm auf eine oder mehrere Textur/en gerendert. Diese Technik wurde verwendet um die simulierten Marker auf Texturen zu rendern. Die gewonnen Texturen beinhalten Daten für Farben, Normalen und Tiefenwerten. Da die Texturen auf der Grafikkarte in einem speziellen Texturspeicher gespeichert werden und CUDA über Funktionalität verfügt, um auf diese schnell zuzugreifen, eignen sich Texturen sehr gut um große Datenmengen auf die Grafikkarte zu übertragen.

Für die Verwaltung der FBO's und ihrer Texturen wurde eine Klasse GLManager implementiert. Die Klasse GLManager hat drei Arten von Methoden:

a Generieren der FBO's

b Generieren der Texturen

c Registrieren der Texturen für CUDA

Die Breite und Höhe einer Textur für Partikel ergibt sich aus der Anzahl der Partikel und Höhe und Breite des Viewports mit dem die Partikel gerendert werden. Wie so eine Textur aussehen könnte kan man Abbildung 9 entnehmen. Die Höhe und Breite des Viewports ist eng verbunden mit der Auflösung des Kamerabildes. Das heißt, dass der Viewport der Auflösung der verwendeten Kameraaufnahmen entsprechen sollte um einen einfachen Zugriff von Pixeln im Kamerabild und den korrespondierenden Pixeln aus der Textur der Partikel zu ermöglichen. Eine weitere Limitation beim Generieren der Texturen ist die MAX_TEXTURE_SIZE von OpenGL. Grafikkarten die OpenGL 4.1+ unterstützen haben eine MAX_TEXTURE_SIZE von 16384 x 16384 [OpenGL Wiki] . Aus diesem Grund muss die Auflösung so gewählt werden, dass die gewünschte Partikelanzahl überhaupt auf eine Textur passt. GLManager hat für das Generieren der Textur die Methode *arrangeTexture()*. Je nach Anzahl der Partikel wird eine Textur mit den folgenden Abmessungen erstellt:

$$Texture_Width = Viewport_Width * particleRows$$

$$Texture_Height = Viewport_Height * particleColumns$$

Die Variablen $particleRows$ und $particleColums$ beschreiben wie viele Partikel in einer Reihe bzw. Zeile in die Textur gerendert werden sollen und

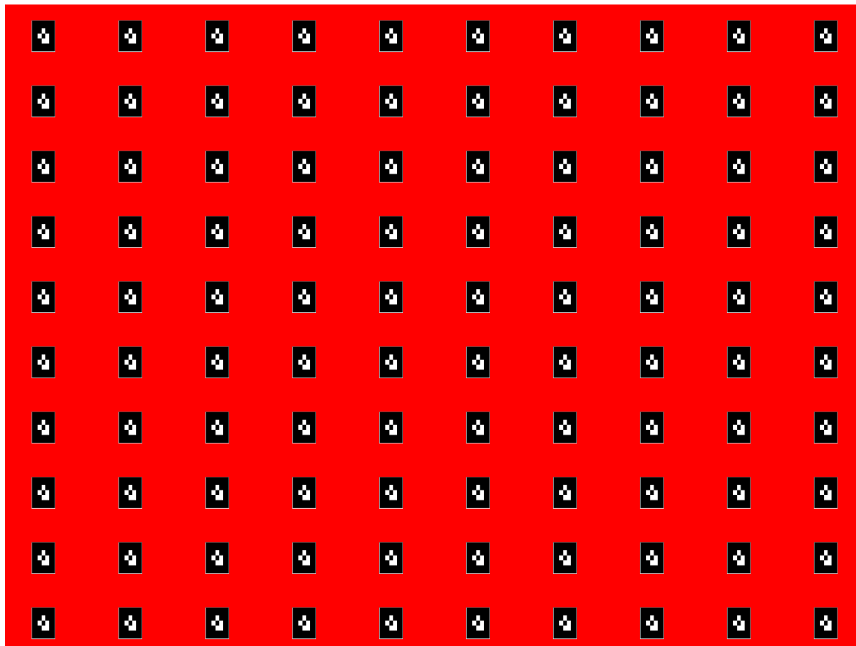


Figure 9: Textur mit 100 Partikeln und einem Viewport von 320 x 240 Pixeln pro Partikel

ergeben sich aus der gesamten Anzahl an Partikeln.

Für die Likelyhood-Funktionen für HSV-Farben, Normalen und Tiefenwerten wurde ein einziges FBO benötigt. Marker und die virtuelle Testumgebung wurden ebenfalls in ein seperates FBO gerendert, um die Aufnahme eines Kamerabildes zu simulieren. Beim Einsatz der Stereokamera mit dem ZED SDK wurde nur eine OpenGL Textur benötigt, um das Kamerabild auf den Bildschirm zu rendern. Würde man auf das Rendern der Stereokameraufnahmen verzichten, könnte gänzlich auf OpenGl Operationen verzichtet werden, da man auf die Texturen vom ZED SDK direkt über CUDA zugreifen kann. Um ein Katenbild der gerenderten Partikel zu generieren wurden zwei weitere FBO's in die Implementation integriert. Die beiden FBO's wurden genutzt für das Postprocessing mit einem Blur und Sobel Shader. Anschließend wurde das Kantenbild mit einem Shader für Dilatation in den Framebuffer in den die ursprüngliche Partikeltextur gerendert wurde gerendert.

5.5 Host to Device Operationen

Generell gilt wie schon in Abschnitt 3.2.2 beschrieben, alle Daten, die auf dem Host vorhanden sind und in CUDA weiterverwendet sollen auf den Speicher der GPU übertragen werden. Für diesen Zweck wird zuerst Spe-

icher auf der GPU allokiert und anschließend mit den von CUDA bereitgestellten Funktionen, die in Abschnitt 3.2.2 genannt wurden, auf das Device kopiert werden. Bei der Implementation handelt es sich hierbei um das Array von Partikel und ein weiteres Array um nachzuverfolgen wie oft die Likelyhood-Funktion vom Device aufgerufen wurde. Will man die Texturen in CUDA weiter nutzen, muss man sie zuerst registrieren. CUDA bietet für diesen Zweck die Methode *cudaGraphicsGLRegisterImage*. Die weiteren Operationen werden mittels des Codebeispiels aus Listings 3 erklärt. In Zeile 1 wird das Format der Textur festgelegt. In Zeile 7 muss die registrierte Textur an eine CUDA Resource gebunden werden, um anschließend in Zeile 9–12 gemapt zu werden. Wenn alle Kernels durchlaufen wurden müssen die Texturen wieder von CUDA gelöst werden, damit OpenGL wieder auf diesen Speicherbereich schreiben kann. Dieser Prozess muss jedes Mal wiederholt werden für jeden Durchlauf der Renderschleife.

```
texture<uchar4, cudaTextureType2D,
cudaReadModeNormalizedFloat> texRef;

void host(cudaGraphicsResource_t &texSr,
 vector<PartikelStruct> &particels, std::vector<float> &norm){

		cudaGraphicsResource_t resource = texSrc;

		cudaGraphicsMapResources( 1, &resource,0 );
		cudaArray* srcTexArray;
		cudaGraphicsSubResourceGetMappedArray(
		&srcTexArray, resource, 0, 0 );
		cudaBindTextureToArray( texRef, srcTexArray );

		// Allokieren des Speichers fuer particels und norm
		// Kopieren auf das Device
		// Einteilen der Bloecke und Threads
		// Kernel Aufruf
		// particles mit neuer Gewichtung auf Host kopieren

		cudaUnbindTexture( texRef );
		cudaGraphicsUnmapResources( 1, &resource ) ;

};
```

Listing 3: Host to Device

5.6 Sampleing

Die Streuung der Partikel wurde mit zwei Normalverteilungen realisiert. Jeweils eine für Translationen und eine für Rotationen der Partikel. Die Normalverteilungen werden im Programm mit der Bibliothek <random> generiert. Als Parameter der Normalverteilung dienen ein Mittelwert und eine Varianz. Pseudocode in Listings XY beinhaltet die implementierte

Sampling Methode.

```
void sampling(vector<particleStruct> &particleSet,
int particleAmount) {

    float x, y, z, alpha, beta, gamma;

    default_random_engine generator{
    static_cast<long unsigned int>(time(0))};
    normal_distribution<float> distTranslation(0.0f, 0.05f);

    default_random_engine generatorRotation{
    static_cast<long unsigned int>(time(0))};
    normal_distribution<float> distRotation(0.0f, 0.5f);

    for (int i = 0; i < particleAmount; i++) {

        x = distTranslation(generator);
        y = distTranslation(generator);
        z = distTranslation(generator);
        alpha = distRotation(generatorRotation);
        beta = distRotation(generatorRotation);
        gamma = distRotation(generatorRotation);

        // translate
        particleSet.at(i).model = translate(partikel.at(i).model,
        particleSet.at(i).translation + glm::vec3(x,y,z) );

        // rotate at origin
        particleSet.at(i).model = rotate(particcleSet.at(i).model,
        radians(alpha),vec3(1.0f, 0.0f, 0.0f));
        particleSet.at(i).model = rotate(particleSet.at(i).model,
        radians(beta), vec3(0.0f, 1.0f, 0.0f));
        particleSet.at(i).model = rotate(particleSet.at(i).model,
        radians(gamma), vec3(0.0f, 0.0f, 1.0f));

        // translate to origin
        particleSet.at(i).model = translate(
        particleSet.at(i).model, -particleSet.at(i).translation );

        particleSet.at(i).translation += vec3(x,y,z);
    }
};
```

Listing 4: Sampling Algorithmus der implementation

5.7 Resampling Methods

Für das Resampling wurden zwei verschiedene Verfahren gewählt. Einmal das Group-Based Resampling in Abschnitt 5.7.2 und als Alternative das Multinomial Resampling in Abschnitt 5.7.1. Das Multinomial Resampling

wurde gewählt, da es ein typisches Resampling Verfahren ist, dass die Bedingung erfüllt Partikel unvoreingenommen aus dem Set auszuwählen. Beim Group-Based Resampling wird diese Bedingung dagegen nicht erfüllt. Partikel beim Group-Based Resampling werden in Gruppen unterteilt und es wird ein anderes Auswahlverfahren für jede Gruppe genommen.

5.7.1 Multinomial Resampling

Für das Multinomial Resampling wurde sich am Algorithmus aus [Li] orientiert. Aus dem normalisierten Partikelset wird ein neues Set an Partikeln Q_t erstellt nach folgender Regel:

$$Q_t^1 = w_t^1$$

$$Q_t^m = Q_t^{m-1} + w_t^m$$

Dabei ist $m \in \{1 \dots M\}$ und M die gesamte Anzahl an Partikeln im Set. Das Gewicht jedes Partikel Q_t^m beträgt somit sein eigenes Gewicht aus der Likelyhood-Funktion und die akkumulierten Gewichte aller Partikel, einschließlich Q_t^{m-1}. Aus diesem Set werden nun Partikel durch ein Zufallswert zwischen $(0, 1]$ ausgewählt und in das finale Set $\overline{X_t}$ geschrieben. Dieser Vorgang wird solange wiederholt bis $\overline{X_t}$ ebenfalls M Partikel hat. Durch das Akkumulieren der Partikelgewichte bekommt man gewichtete Bereiche aus denen die Partikel ausgewählt werden. Siehe Abbildung 10 für eine visuelle Darstellung des Sets Q_t entnommen aus [Li].

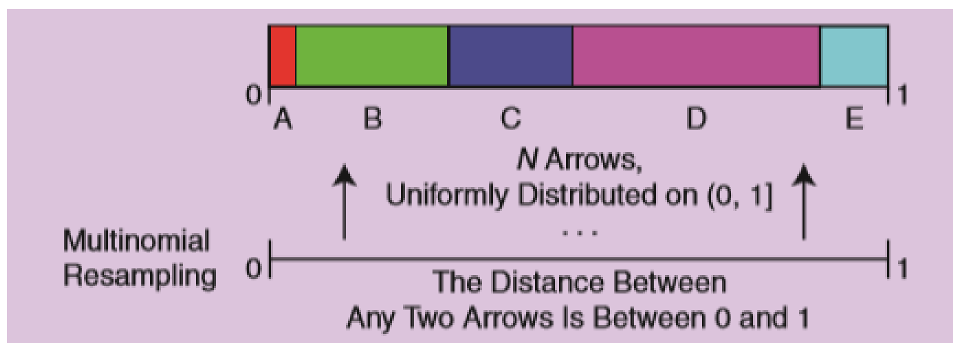


Figure 10: Multimodial Resampling Darstellung

Partikel mit hohen Gewichten nehmen größere Bereiche ein und werden öfter gewählt als Partikel mit kleinen Gewichte. Dennoch wird aus der ganzen Spanne des Partikelsets ausgewählt was für eine höhere Varianz bei der Verteilung des Partikelsets $\overline{X_t}$ sorgt.

Anstatt ein neues Partikelset an akkumulierten Gewichten zu erstellen wurde der Algorithmus mit einer while-Schleife umgesetzt, die Gewichte

solange aufsummiert bis sie größer als eine generierte Zufallszahl sind. Zusätzlich wird in der while-Schleife ein Index j jeweils um 1 erhöht. Beim Abbruch der Schleife wird der Partikel an der Stelle X^{j-1} gewählt, was dem oben beschriebenen Multinomial Resampling entspricht. Siehe dazu listings 5 für die Implementation in Pseudocode.

```
vector<ParticleStruct> resampling_Multimodial(
vector<ParticleStruct> &particelSet ,int particleAmount) {

    default_random_engine generator{
    static_cast<long unsigned int>(time(0))};
    uniform_real_distribution<float> dist(0.0f,1.0f);
    vector<ParticleStruct> newParticleSet;

    for (int i = 0; i < particelAmount; i++) {
        float roll = dist(generator);
        float sum = 0.0f;
        int j = 0;
        while(sum<roll && j<particelAmount-1){

            sum += particelSet.at(j).weight;
            j++;
        }
        newParticleSet.push_back(particleSet.at(j -1));
    }
    return newParticleSet;
};
```

Listing 5: Multinomial Resampling

5.7.2 Group-Based Resampling

Bei der Implementation eines Group-Based Resamplingverfahrens wurde sich am Artikel [Li] für das Threshold/Group-Based Resampling orientiert. Zunächst wurde das Partikelset X_t anhand der Gewichte in absteigender Reihenfolge sortiert und in zwei Gruppen G_best und G_rest aufgeteilt. Die Gruppe G_best besteht dabei aus den Partikeln $G_best = \{X_t^1, X_t^2, \ldots X_t^n\}$ und demzufolge die Gruppe G_rest aus den Partikeln

$G_rest = \{X_t^{n+1}, X_t^{n+2}, \ldots X_t^m\}$ mit n als ein beliebiger Grenzwert der zwischen 1 und $m-1$ liegt. Nach der Gruppierung werden Partikel aus der Gruppe G_best zufällig ausgewählt bis sich ein neues Set $\overline{X_t}$ ergibt. Siehe für die Implementation Listings 6.

```
vector<ParticleStruct> resampling_Group(
vector<ParticleStruct> &particelSet ,int particleAmount) {

    sort(particelSet.begin(), particleSet.end(), compare);

    float n = 0.1f;
    default_random_engine generator;

```

```
    uniform_int_distribution<int> dist(0,particleSet.size() * n );

    vector<ParticelStruct> newParticleSet;

    for(int i = 0; i<particelAmount; i++){
        newParticleSet.push_back(particleSet.at(dist(generator)));
    }

    return newParticleSet;

};
```

Listing 6: Group-Based Resampling

5.8 Likelyhood-Funktionen

Die Bewertung der Partikel wurde in CUDA realisiert. Für jede Likelyhood-Funktion wurde ein Kernel geschrieben. Die Struktur eines Kernels für die Likelyhood-Funktionen kann man Listings 7 entnehmen.

```
__global__ void kernel_likelyhood(PartikelStruct *particleSet,
int moduloX, int moduloY, int rows, int colums, float alpha) {

unsigned int x = threadIdx.x+blockIdx.x*blockDim.x;
unsigned int y = threadIdx.y+blockIdx.y*blockDim.y;

unsigned int xIndex = 0;
unsigned int yIndex = 0;

float weight = 0.0f;
float4 tex;
float4 zedTex;

int xMod = x % moduloX;
int yMod = y % moduloY;

pixel = tex2D(texRef_particles,x,y);
measurement = tex2D(texRef_measurement,xMod,yMod);

xIndex = (rows*x) / (blockDim.x * gridDim.x);
yIndex = (colums*y) / (blockDim.y * gridDim.y);
yIndex = yIndex*rows;

weight = likelyhood(pixel, measurement);
atomicAdd(&particleSet[xIndex + yIndex].weight, weight);

};
```

Listing 7: Likelyhood-Kernel

Wie in Abschnitt 4.1 beschrieben hat jede Art von Bildinformationen eine eigene Likelyhood-Funktion. Diese Funktionen sollen beschreiben,

wie wahrscheinlich es ist, dass der simulierte Zustand mit einer Messung übereinstimmt. Bei der Implementation der Likelyhood-Funktionen diente als Basis die natürliche Exponentialfunktion. Sie bietet sich besonders gut an, wegen ihrem schnellen Anstieg bei großen Werten im Exponenten bzw. rapiden Abfall bei niedrigen Werten im Exponenten. Außerdem besitzen Exponentialfunktionen die Eigenschaft $x^0 = 1$. Diese Eigenschaft ist hilfreich, da es sich bei den implementierten Funktionen im Exponenten um Distanzfunktionen handelt, welche bei identischen Werten null werden und deshalb oft vorkommen. Negiert man den Exponenten bekommt man Werte im Intervall von $(0, 1]$. Das bedeutet, dass Partikel, die stark von der Messung abweichen ein Gewicht bekommen, der gegen 0 konvergiert. Die Kombination der Likelyhood-Funktionen von HSV Farbwerten, Normalen und Tiefenwerten sind der folgenden Gleichung zu entnehmen.

$$likelyhood = exp^{-\lambda_c * f_{HSV}} * exp^{-\lambda_n * f_{normals}} * exp^{-\lambda_d * f_{depth}} \quad (7)$$

Dabei sind $\lambda_c, \lambda_n, \lambda_d$ Faktoren, die die Sensibilität der zugehörigen Funktionen bestimmen. Werden nicht alle drei der Terme aus (7) zur Gewichtung benutzt, werden die nicht genutzten auf null gesetzt.

- HSV-Farbwerte

  $$f_{HSV} = ||HSV_{pixel} - HSV_{measurement}||$$

  Die HSV-Funktion beinhaltet die euklidische Distanz zweier Vektoren HSV_{pixel} und $HSV_{measurement}$.

- Flächennormalen

  $$f_{normals} = cos^{-1}\left(\frac{Normal_{pixel} \bullet Normal_{measurement}}{|Normal_{pixel}| * |Normal_{measurement}|}\right)$$

  Bei der Funktion für Flächennormalen wird die Distanz über den Winkel der beiden Normalen $Normal_{pixel}$ und $Normal_{measurement}$ zueinander bestimmt.

- Tiefenwerte

  $$f_{depth} = |Depth_{pixel} - Depth_{measurement}|$$

  Die Funktion für Tiefenwerte beschreibt die Distanz zwischen zwei reellen Zahlen.

- Kanten

  $$likelyhood_edge = exp^(- \lambda(\frac{weight_binr}{v}))$$

  Die Likelyhood-Funktion für Kantenbilder ist die Division von$weight_binr$ mit v. Dabei ist die Variable $weight_binr$ die Anzahl aller Pixel eines

Partikels, die mit der Vergleichstextur der Kameraaufnahme identisch sind. Die Variable v beinhaltet die Anzahl aller Kantenpixel in der Vergleichstextur. Der Quotientenwert wird wie schon bei den anderen Funktionen mit λ multipliziert und negiert. Lambda dient dabei der Einstellung der Sensibilität für die natürliche Exponentialfunktion.

6 Evaluation

6.1 Hardware

Für die Evaluation wurde ein Acer Aspire E-17 Laptop verwendet. Der verbaute Prozessor des Aspire E-17 ist eine Intel Core i5-5200U CPU. Der Arbeitsspeicher entspricht 8 GB DDR3 L RAM. Die verwendete Grafikkarte ist eine NVIDIA GeForce 940M mit 2 GB VRAM.

Als Aufnahmegerät wurde die ZED 2K Stereo Kamera von Stereolabs verwendet.

6.2 Software

Die Applikation wurde in C++ mit dem Visual Studio 14 Compiler und CUDA Version 9.2 implementiert. Als API's kamen OpenGL 3.3 mit GLFW als Window Manager, die Single Header Bibliothek STB Image und das ZED SDK zum Einsatz. Für die Evaluation wurde die Programme Nvidia Visual Profiler und Nvidias Nsight Graphics benutzt.

6.3 Ausgangsparameter

Alle Versuche wurden mit 600 Partikeln durchgeführt und einen Viewport von 320 x 240 Prixeln pro Partikel und 400 Partikeln und einem Viewport von 320 x 180 Partikeln für die Stereokamera. Eine höhere Partikelanzahl würde zwar die Diversität der Verteilung erhöhen, aber die Einbuße in der Performance wurden als zu signifikant eingestuft. Ab 900+ Partikeln läuft die Anwendung unter 9 Frames pro Sekunde. Der λ Parameter der Likelyhood-Funktionen aus Abschnitt 5.8 wurde initial auf den Wert 15 gesetzt. Im Laufe der Evaluation wurde dieser als Teil der Optimierung individuell für die Likelyhood-Funktionen angepasst. Welche Likelyhood-Funktionen betroffen waren wird in den Abschnitten für die zugehörige Likelyhood-Funktion erwähnt. Alle Versuche werden sowohl mit dem Group-Based Resampling, als auch dem Multinomial Resampling, durchgeführt. Gestartet wird immer mit dem Group-Based Resampling. Alle Partikel werden aus der initialen Position (0, 0, 0) gesampelt.

6.4 Versuchsaufbau

Für die virtuelle Umgebung "Karton" wurden zwei Versuche pro Likelyhood-Funktion durchgeführt. Ein Versuch bei dem eine realistische Kamerafahrt (1) simuliert wurde, ohne den Marker zu verdecken. Siehe Abbildungen 12 und 14 für eine Sequenz von Kamerafahrt (1). Der zweite Versuch simulierte eine schnelle horizontale Kamerafahrt (2). Während der Kamerafahrt (2) wird die Sicht der Kamera auf den Marker blockiert um das Verhalten des Trackings bei Verdeckungen evaluieren zu können. Zusätzlich werden die Likelyhood-Funktionen mit nur einer der drei Bildinformationen, heißt HSV-Farbwerte, Normalen oder Tiefenwerte, in einem Best-Case-Szenario getestet. Dabei werden weitere Versuche mit einer Versuchsumgebung durchgeführt, bei der die HSV-, Normalen- oder Tiefenwerte besonders gut abschneiden könnten. Versuch (1) dauert 18.0Sek und Versuch (2) 10.0Sek. Bei beiden Versuchen wird die relative Distanz von Marker und simulierten Partikeln gemessen, sowie der Winkel zwischen ihren Normalen. Die Distanz dient der Dokumentation für die Entfernung zwischen Marker und Partikel. Der Winkel der Normalen beschreibt die Abweichung der Rotationen. Alle versuche wurden 20 mal wiederholt um eine Durchschnittliche Abweichung zu berechnen. Damit das Tracking besser zu veranschaulichen ist, wurde auf die Umrechnung der Kameraposen verzichtet, sondern nur Markerposen simuliert.

Um Missverständnisse zu vermeiden wird an dieser Stelle erwähnt, dass die umgangssprachliche Formulierung "ein Marker wurde gefunden" als Umschreibung dient für die Klassifizierung, dass die Position und Rotation des simulierten Partikels zum Marker unter 10mm liegt für die Pose und der Winkel der Normalen von Partikel und Marker unter 15 Grad ist. Zusätzlich muss eine klare Bewegung zum Marker hin feststellbar sein.

6.5 HSV-Farbwerte

6.5.1 Testumgebung "Karton"

Das Tracking anhand von HSV-Farbwerten mit Group-Based Resampling erwies sich bei der Kamerafahrt (1) als robust. Der Marker konnte innerhalb 2sec gefunden werden. Im Durchschnitt betrug die Abweichung 1.65mm in x-Richtung, 1.21mm in y-Richtung und 4.91mm in z-Richtung. Bei stillstehender Kamera verbessern sich die x- und y-Werte um einen Faktor 10. Die Abweichung in z-Richtung hingegen bleibt nahezu unverändert. Der Winkel zwischen Partikel und simulierten Marker lag im Durchschnitt bei 7.8 Grad. Solange die Kameradrehungen in einem Winkel unter 20 Grad blieben, verbesserten sich die Abweichung auf 2.6 Grad.

Die Werte der Kamerafahrt (2) liegen bei 5.14mm in x-Richtung, 2.46mm in y-Richtung und 10.1mm in z-Richtung. Der durchschnittliche Winkel lag bei 7.67 Grad. Die die schlechten Durchschnittswerte ergeben sich aus Ungenauigkeiten, die aufgrund von teilweiser bis vollständiger Verdeckung des Marker verursacht wurden.

Dennoch konnte der Marker bei Rückschwenkung der Kamera wieder gefunden werden und das Tracking fortgesetzt werden.

Beim Multinomial Resampling konnte der Marker erst innerhalb 6sec Sekunden gefunden werden. Dieser Umstand lässt sich auf die breiter gestreute Verteilung zurückführen, die durch ein unvoreingenommenes Ziehen aus dem Partikelset entsteht. Dadurch wird der Marker zwar langsamer getrackt, dafür gibt es weniger Jittering. Die Durchschnittswerte sind 2.23mm Abweichung in x-Richtung, 2.99mm in y-Richtung, 8.43mm in z-Richtung und einem durchschnittlichen Winkel von 3.8 Grad. Im Versuch mit Kamerafahrt (2) haben sich die Durchschnittswerte kaum verschlechtert. Die Abweichung erhöhte sich nur geringfügig um 0.6mm für die x-Richtung und 0.3mm für y- und z-Richtung. Der Winkel blieb nahezu unverändert. Auch hier wurde der Marker kurzzeitig verloren, konnte aber in unter 20 Frames wieder gefunden werden.

6.5.2 Optimierte Versuchsumgebung

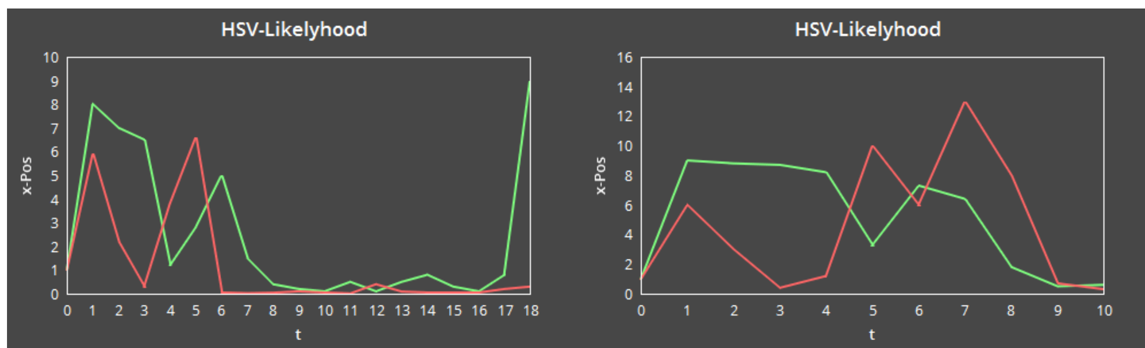
Für eine optimierte Versuchsumgebung wurde der "Karton" entfernt und lediglich der Marker alleine gerendert mit einem Hintergrund, dessen Farbe die Komplementärfarbe des Markers hat. Erwartungsgemäß haben sich die Durchschnittswerte nochmal verbessert. Für die x- und y-Richtung liegt die Abweichung unter 1.0 mm. Die Abweichung in z-Richtung hat den Wert 3.31 mm und für die Rotation beträgt der Winkel 3.3 Grad. Der Marker konnte innerhalb 1.0Sek gefunden werden.

6.6 Normalen

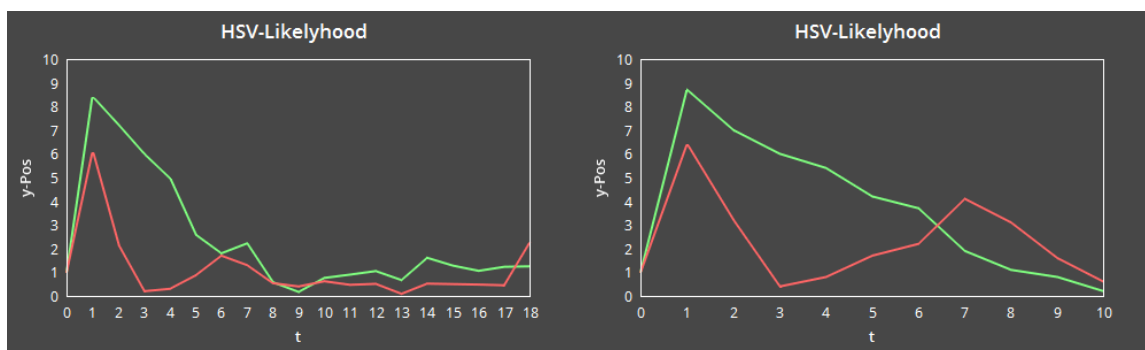
6.6.1 Testumgebung "Karton"

Die Ergebnisse beim Tracking mit Normalen ergab schlechte Resultate. Bei beiden Kamerafahrten konnte der Marker nicht gefunden werden. Da die Normale des Markers identisch mit der Normalen der im Hintergrund verwendete Wand ist, konnte nicht zwischen den Normalen der Wand und denen des Markers bei der Gewichtung differenziert werden.

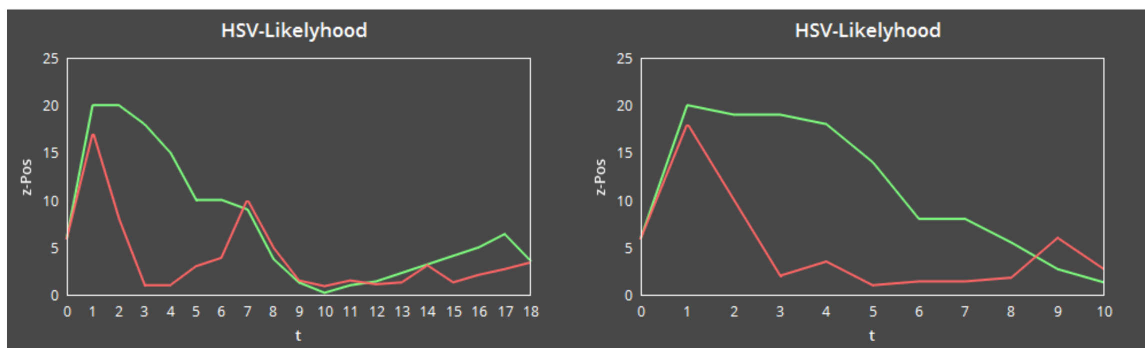
Auch eine Drehung des Markers um 45 Grad und somit auch der Normalen des Markers machte keinen Unterschied. Hier liegt eine Schwäche der Implementation. Da nur der Bereich des simulierten Partikels in die Gewichtung einfließt und nicht der volle Bildbereich, kann es passieren,



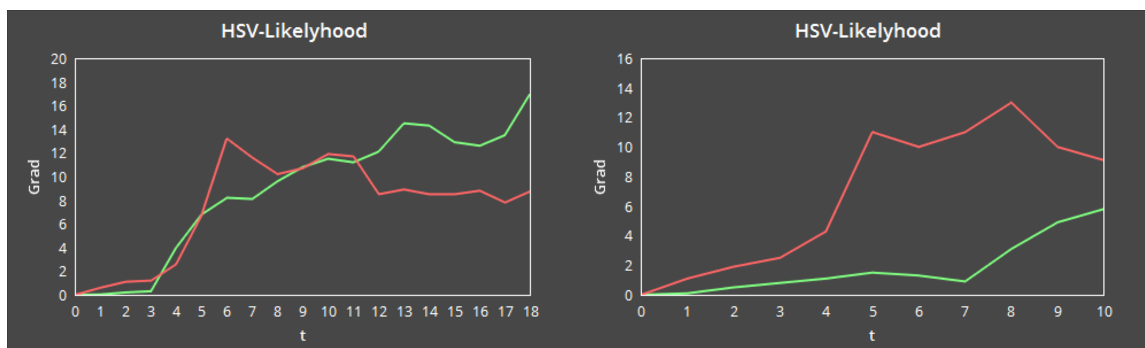
(a)



(b)



(c)



(d)

Figure 11: Hsv-Likelyhood mit Kamerafahrt (1) Links und Kamerafahrt (2) Rechts. Rot ist der Partikelfilter mit Group-Based Resampling und Grün der Partikelfilter mit Multimodial Resampling. (a) x-Abweichung in mm, (b) y-Abweichung in mm, (c) z-Abweichung in mm über, (d) Winkel-Abweichung in Grad über die Zeit t in Sekunden

dass der Partikelfilter seine Suche in einem Bereich anfängt, der sich der initialen Verteilung des Partikelsets sehr ähnelt. Folglich kommen die Partikel aus diesem Bereich nicht mehr raus. Eine Lösung dafür wäre ein mehrschichtiges Sampling in dem über mehrere Durchläufe die Partikel gestreut werden und bei jedem Durchlauf die Varianz sich verringert, wie in [Kle] umgesetzt wurde.

6.6.2 Optimierte Versuchsumgebung

Als optimierte Versuchsumgebung wurde nur der Marker gerendert. Die Hintergrundfarbe wurde so gewählt, dass sie die Werte einer Normalen hat, die um -45 Grad um die y-Achse gedreht ist. Der Marker wurde ebenfalls um 45 Grad gedreht. Somit hebt sich der Marker deutlich vom Hintergrund ab. In dieser Testumgebung konnte der Marker erfolgreich innerhalb von 3.0Sek gefunden werden.

Während der Kamerafahrt (1) wurde das Tracking ungenau. Die Abweichung in z-Richtung betrug 18.1 mm. Eine Erklärung dafür wäre, dass je weiter zurück der Partikel ist, desto weniger Pixel der Textur werden in Betracht gezogen und verfälschen die Gewichtung dahingehend, dass Partikel mit hohen z-Werte in negative z-Richtung besser bewertet werden als Partikel die nah am Marker bleiben und deshalb eine größere Fläche für Abweichungen bieten.

6.7 Tiefenwerte

Auch das Tracking mit Tiefenwerten ist stark fehleranfällig. In Testumgebung (1) mit Kamerafahrt (1) konnte der Marker nicht gefunden werden. Wegen des geringen Abstandes von nur 4.0 cm des Markers zur Wand hebt sich der Marker kaum vom Hintergrund ab. Erhöht man jedoch die Abstände zwischen Hintergrund und Marker erweisen sich Tiefenwerte als sehr robust beim Tracking. Das suggerieren die Daten aus der optimierten Versuchsumgebung für Tiefenwerte.

6.7.1 Optimierte Versuchsumgebung

Passt man die Testumgebung (1) an indem man den Hintergrund auf 55.0 cm entfernt vom Marker setzt, verbessern sich die Ergebnisse signifikant. Der Marker wird innerhalb von 4.0 Sekunden vollständig erkannt und bleibt auch während der Kamerafahrt sehr robust mit wenig Jittering. Die durchschnittliche Abweichung liegt bei 1.48 mm in x-Richtung, 1.91 mm in y-Richtung und 4.58 mm in z-Richtung. Der durchschnittliche Winkel beträgt 4.33 Grad.

Nahm man, als Resampling Verfahren das Multimodial Resampling wurde der Marker nicht gefunden und verblieb weitestgehend in seinem Startzustand. Dieser Umstand könnte den Gewichten geschuldet sein, die alle recht nah beieinander liegen und nur wenige etwas besser bewertet sind als andere. Somit werden viele mittelmäßige Partikel gezogen anstatt wie beim Group-Based Resampling auch bei ähnlichen Gewichten trotzdem die besten erwischt werden.

6.8 HSV-Normalen Likelyhood

Die Kombination von Hsv-Farbwerten und Normalen wies sehr gute Ergebnisse auf. Entgegen der Annahme, dass die Normalen das Tracking in der Testumgebung "Karton" erschweren würden haben sich nicht bestätigt. Die Abweichung von Partikel zu Marker in Kamerafahrt (1) sind 1.35 mm in x-Richtung, 1.01 mm in y-Richtung und 3.65 mm in z-Richtung. Der Winkel hat sich gegenüber dem Tracking mit Hsv-Farbwerten ohne Normalen um 7.3 Grad verbessert und lag im Durchschnitt bei 0.49 Grad. Die guten Ergebnisse aus Kamerafahrt (1) konnten in Kamerafahrt (2) nur teilweise repliziert werden. Die durchschnittliche Abweichung betrug 4.79 mm in x-Richtung, 2.39 mm in y-Richtung und 10.7 mm in z-Richtung. In x- und z-Richtung schlagen die Werte stark aus. Die Verschlechterung der Abweichung in x-Richtung lässt sich durch die Bewegung entlang der horizontalen erklären. Die Verschlechterung der z-Werte kann man auf die Verdeckung des Marker zurückführen und einem ähnlichen Umstand wie in Abschnitt 6.6.2. Stabil blieb hingegen die Abweichung beim Winkel. Diese betrug im Durchschnitt nur 0.56 Grad.

Relativ schlechte Ergebnisse wurden mit dem Multimodial Resampling erzielt. Der Marker konnte erst nach 6.0Sek gefunden werden. Im Vergleich dazu wurden 2.0Sek beim Group-Based Resampling gebraucht. Durchschnittliche Abweichungen waren 3.19 mm in x-Richtung, 2.56 mm in y-Richtung, 8.93 mm in z-Richtung und einem Winkel von 1.29 Grad.

Kamerafahrt (2) lieferte folgende Durchschnittswerte für Abweichungen: 3.41 mm in x-Richtung, 4.1 mm in y-Richtung, 11.4 mm in z-Richtung und einem Winkel von 1.6 Grad.

Der Parameter λ wurde auf den Wert 5.0 für die Likelyhood der Normalen verringert wegen den schlechten Ergebnissen aus 6.6.

6.9 HSV-Tiefenwerte Likelyhood

Bei den Versuchen mit Kamerafahrt (1) in der "Karton" Testumgebung wurden sehr gute Durchschnittswerte erzielt. Lediglich in der Rotation litt die Kombination von Hsv-Farbwerten und Tiefenwerten. Ebenfalls war

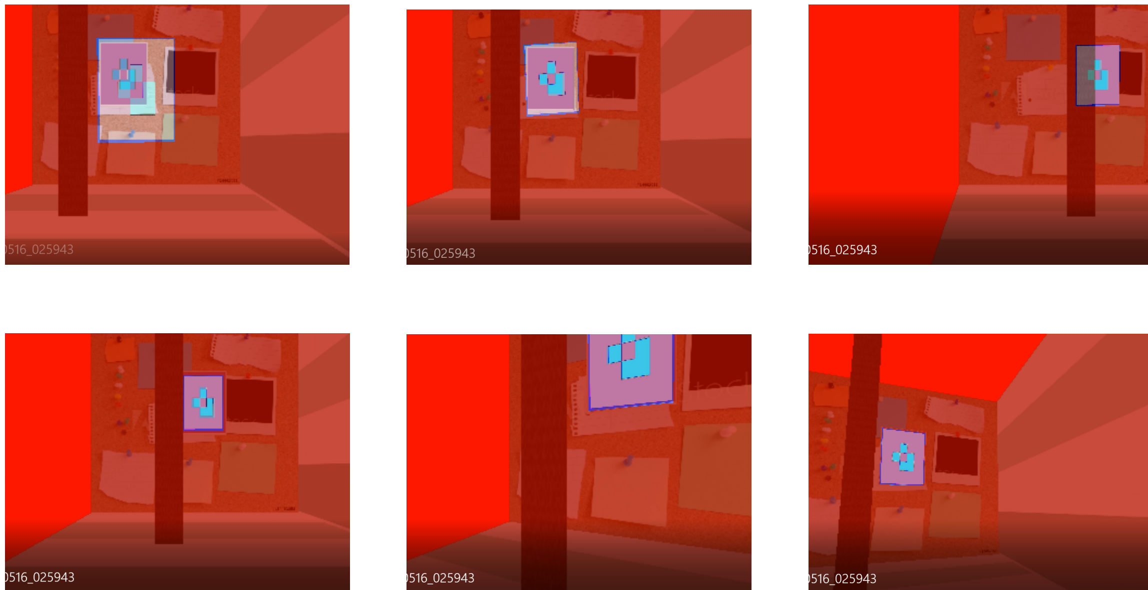
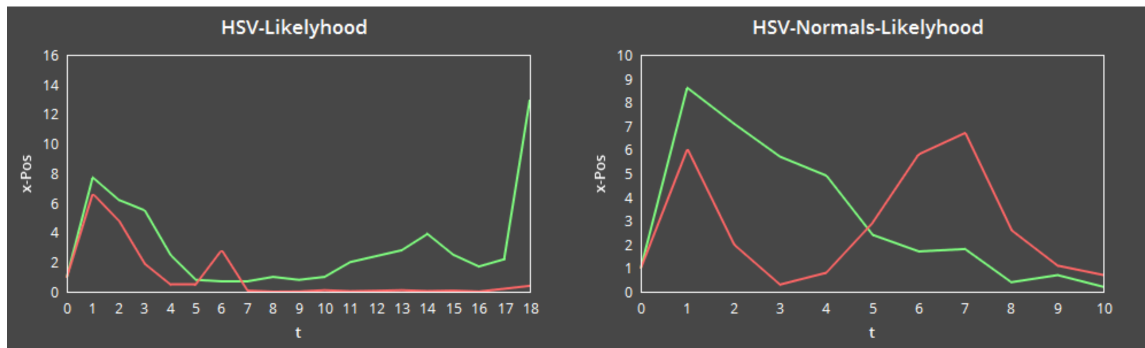


Figure 12: Kamerafahrt (1) mit dem kontrastreichen Marker und der HSV-Normalen-Likelyhood. Farben sind sehr rötlich, weil die Textur des Partikels und der Umgebung übereinandergelegt wurden

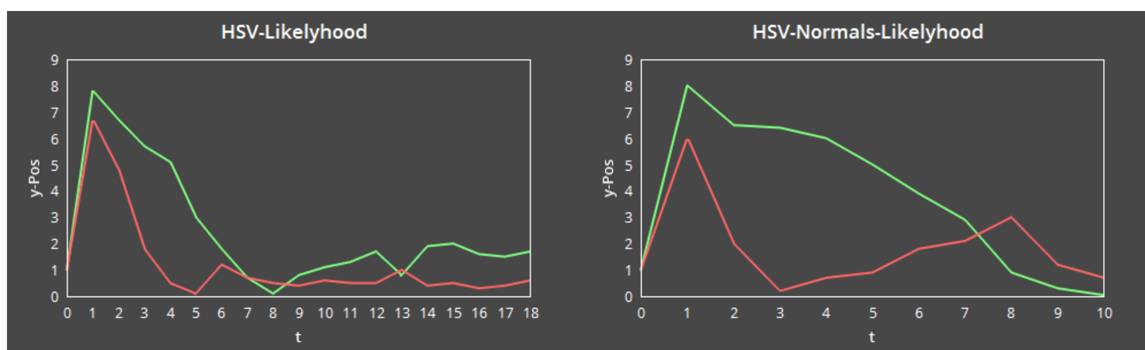
bei der Kamerafahrt (2) ein robustes Tracking möglich. Sowohl bei Kamerafahrt (1) als auch bei Kamerafahrt (2) konnte der Marker innerhalb 3.0Sek gefunden werden. Jittering war bei beiden Kamerafahrten gering. Dagegen wird der Marker oft verloren, wenn Multimodial Resampling aktiv ist. Da das Multimodial Sampling moderate Ergebnisse beim Tracking nur mit Hsv-Farbwerten erzielte, wurde der λ-Wert der Likelyhood-Funktion für Tiefenwerte auf 3 reduziert. Dennoch blieb die Verfolgung des Markers unterdurchschnittlich.

6.10 Normalen-Tiefenwerte Likelyhood

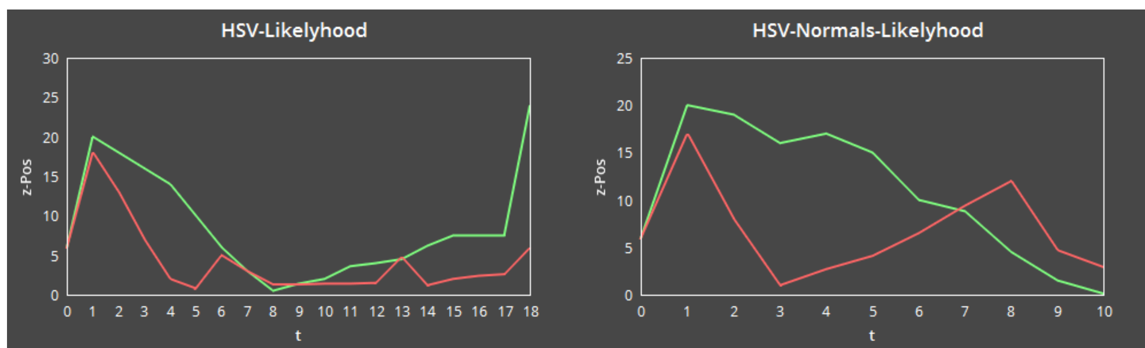
Die schlechtesten Ergebnisse erzielte die Kombination Normalen und Tiefenwerte. Der Marker wurde nur in 2 von 20 Versuchen bei der Kamerafahrt (1) mit Group-Based Resampling gefunden. Kamerafahrt (2) erwies sich als zu schnell. Bevor der Marker gefunden werden konnte, begann die Bewegung. Während der ganzen Laufzeit von 10.0Sek konnte der Marker nicht getrackt werden. Stattdessen konzentrierten sich die Partikel auf zufällige Bereiche ohne Marker und nahmen immer höhere z-Werte an, was ein gutes Zeichen ist, dass das Tracking gescheitert ist. Mit Multimodial Resampling war ein Tracking bei beiden Kamerafahrten nicht möglich.



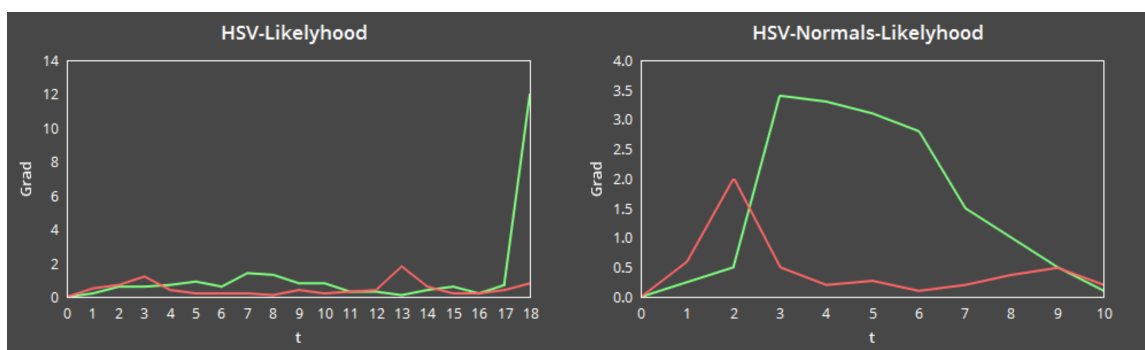
(a)



(b)



(c)



(d)

Figure 13: Hsv-Normalen-Likelyhood mit Kamerafahrt (1) Links und Kamerafahrt (2) Rechts. Rot ist der Partikelfilter mit Group-Based Resampling und Grün der Partikelfilter mit Multimodial Resampling. (a) x-Abweichung in mm, (b) y-Abweichung in mm, (c) z-Abweichung in mm über, (d) Winkel-Abweichung in Grad über die Zeit t in Sekunden

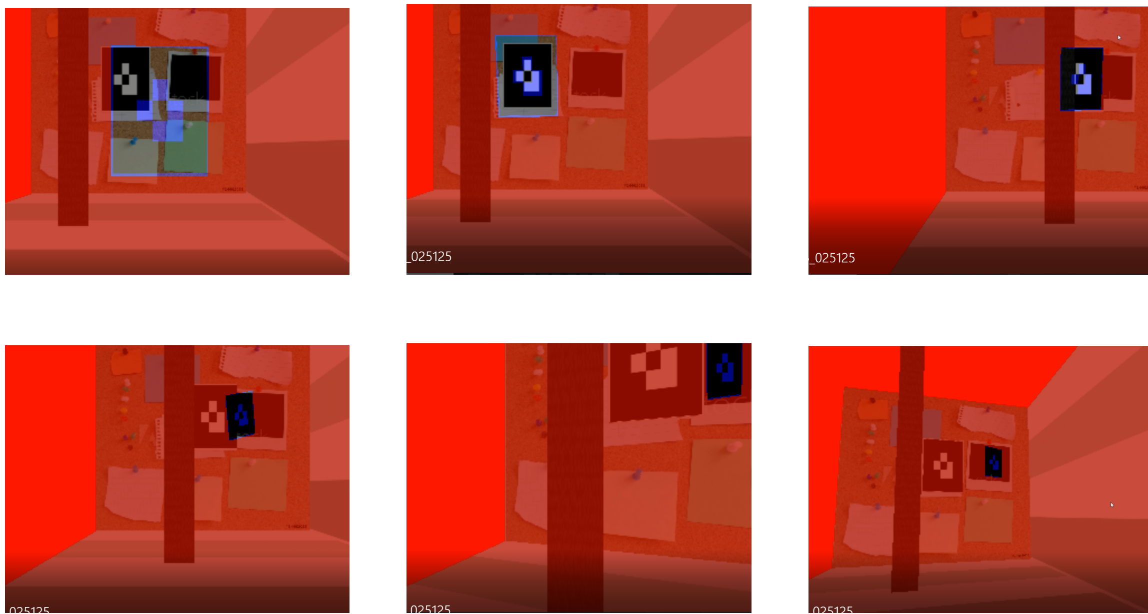


Figure 14: Kamerafahrt (1) mit dem schwarzen Marker und der HSV-Tiefen-Likelyhood. Als negatives Beispiel zu Abbildung 9 in der der Marker erfolgreich getrackt wurde. Das Tracking versagte wegen dem kontrastarmen schwarzen Marker.

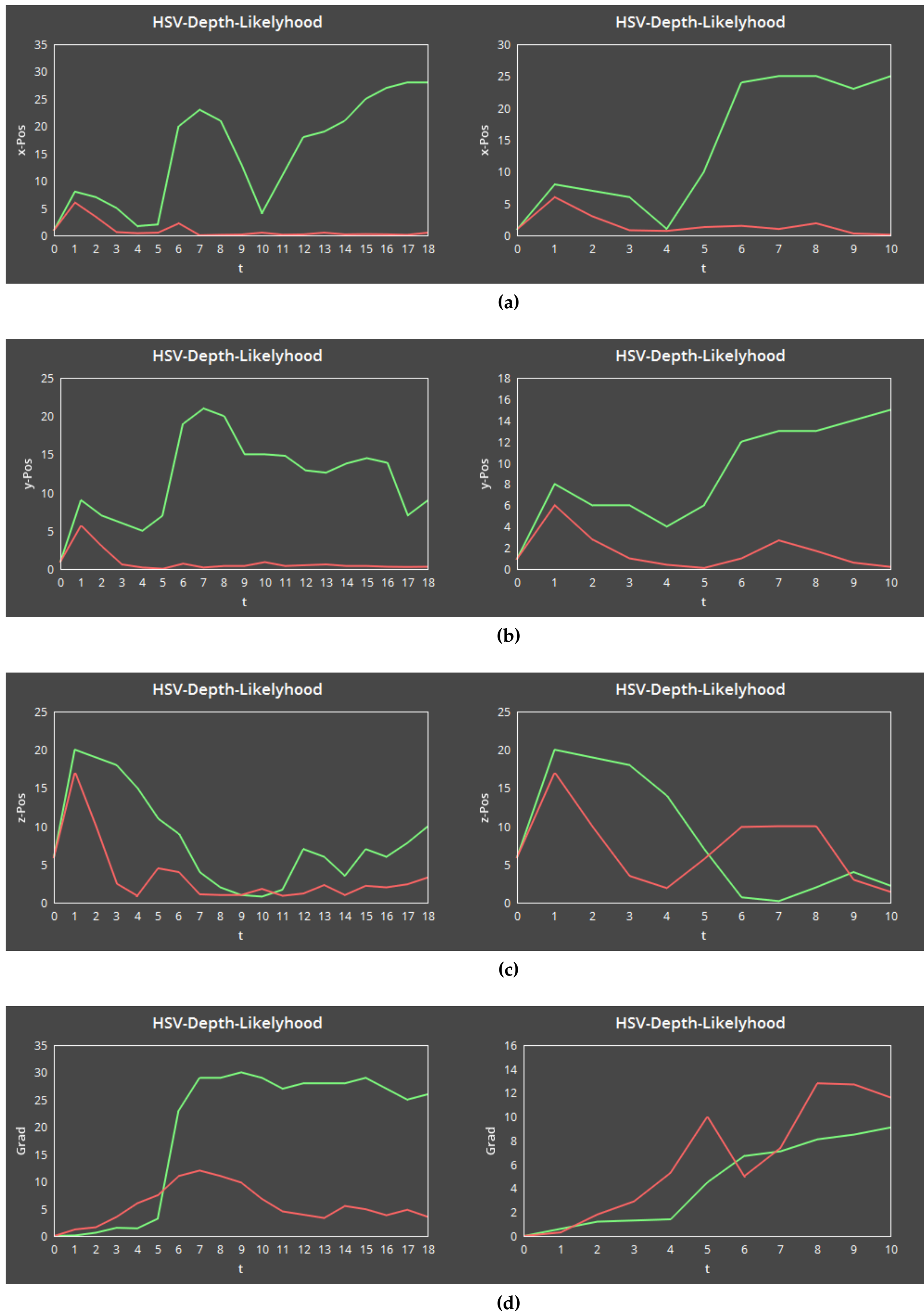
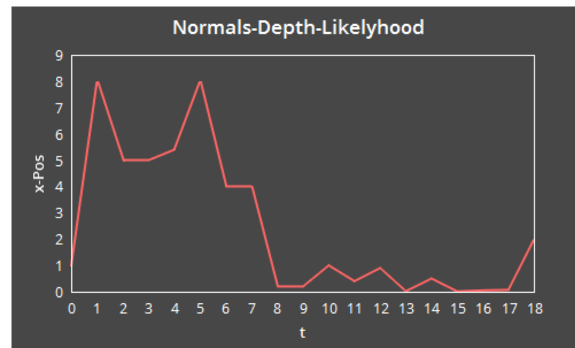
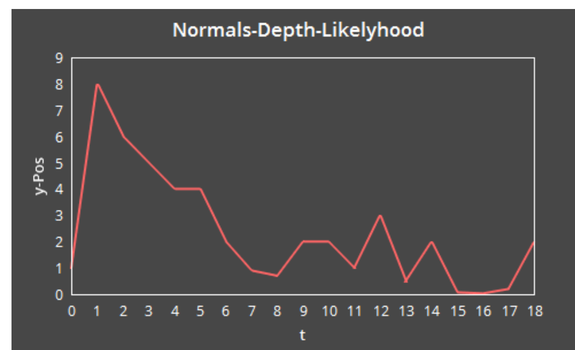


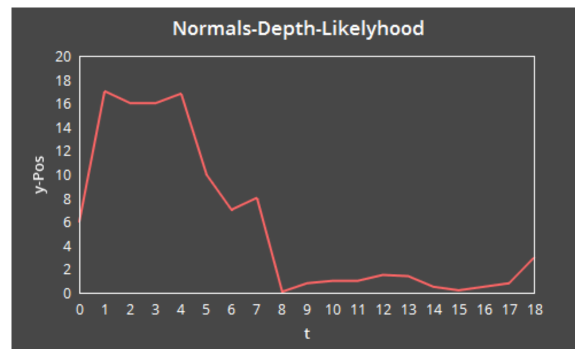
Figure 15: Hsv-Tiefenwerte-Likelyhood mit Kamerafahrt (1) Links und Kamerafahrt (2) Rechts. Rot ist der Partikelfilter mit Group-Based Resampling und Grün der Partikelfilter mit Multimodial Resampling. (a) x-Abweichung in mm, (b) y-Abweichung in mm, (c) z-Abweichung in mm über, (d) Winkel-Abweichung in Grad über die Zeit t in Sekunden



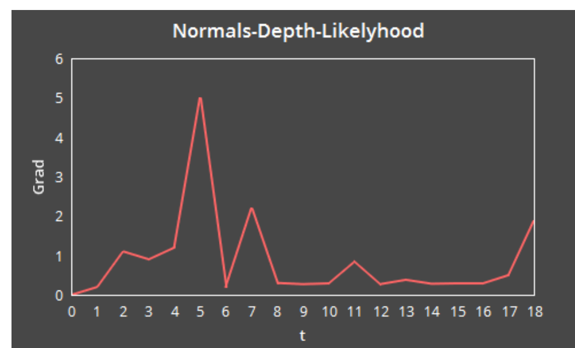
(a)



(b)



(c)



(d)

Figure 16: Normalen-Tiefenwerte-Likelyhood mit Kamerafahrt (1) Links. Rot ist der Partikelfilter mit Group-Based Resampling und Grün der Partikelfilter mit Multimodial Resampling. (a) x-Abweichung in mm, (b) y-Abweichung in mm, (c) z-Abweichung in mm über, (d) Winkel-Abweichung in Grad über die Zeit t in Sekunden

6.11 Kanten

An dieser Stelle seien noch die Versuche mit Kantenbildern erwähnt. Die Ergebnisse aus den Versuchen mit Kantenbildern lieferten vergleichbar kontroverse Ergebnisse. In der "Karton' Testumgebung konnte der Marker nicht gefunden werden, da sich die Partikel auf größere Flächen mit ähnlichen Kantenstrukturen wie der Marker sie hat verteilten. Demzufolge muss ein aufwendigeres Preprozessing des Vergleichbildes vorgenommen werden aus dem möglichst viele irrelevante Kanten entfernt werden.

Ein möglicher Ansatz dafür würden die Tiefenwerte liefern, mit denen man anhand eines Threshold alle Bildelemente im Hintergrund entfernen könnte. Des Weiteren erwies sich ein Partikelfilter auf Kantenbildern mit einem Multimodial Resampling deutlich besser als mit einem Group-Based Resampling. Unter optimalen Bedingungen konnte der Marker schnell getrackt werden und konnte im Vergleich zu den schmalen Varianzbereichen bei den anderen Likelyhood-Funktionen deutlich breitere Varianzintervalle verwerten. Durch diesen Umstand würde die Likelyhood-Funktion mit Kanten besonders von einer hohen Anzahl an Partikeln profitieren.

6.12 Versuche mit ZED Stereokamera

Der im Konzept vorgestellte Versuchsaufbau konnte nicht nachgestellt werden. Stattdessen wurde in einer skalierten Version der Testumgebung (1) gearbeitet. Die Stereokamera befand sich 2.0 m von einer weißen Wand entfernt und eine ausgedruckte Version der zwei Marker wurde in Abständen zwischen 50. cm und 1.0 m zwischen Kamera und Wand platziert. Zunächst sollte der Marker ohne Kamerabewegungen gefunden werden. Falls der Marker gefunden wurde, wurde die Stereokamera stetig um 45 Grad horizontal gewendet. Es sollte festgestellt werden wie robust das Tracking auch bei Bewegungen ist. Die Ergebnisse aus den Versuchen der verschiedenen Likelyhood-Funktionen legen nahe, dass die Implementation noch Schwierigkeiten beim Auffinden des Markers hat. Lag der Marker mittig im Bild konnte, er in der Regel gefunden werden, da die Streuung der Partikel ebenfalls an der Position (0, 0, 0) beginnt. Verschob man den Marker um 30.0 cm in eine der x-, y- oder z-Achsen wurde der Marker nur in 6 von 20 Versuchen gefunden beim Tracking mit HSV-Farbwerten und der Kombination von HSV- und Tiefenwerten. Tiefenwerte alleine konnten den Marker in 3 von 20 Versuchen finden. Bei der Kombination von HSV-Farbwerten und Normalen waren es 7 von 20 Versuchen. Bei Bewegungen der Kamera oder Rotation des Markers stieg die Erfolgsquote auf 9 von 20 Versuchen an. Vergleicht man die Ergebnisse von HSV-Farbwerten mit den Ergebnissen der Kombination von HSV-Farbwerten und Normalen, so sieht man, dass vor allem die Farbwerte für ein erfolgreiches Tracking ver-

antwortlich sind und die Normalen lediglich bei Rotation das Tracking sensibilisieren. Da planare Marker unter realistischen Bedingungen nicht im Raum schweben, sondern auf ebenfalls planare Oberflächen platziert werden, wird die Relevanz der Normalen und Tiefenwerte für planare Marker an dieser Stelle infrage gestellt.

Bei der Kameradrehung um 45 Grad wurden die Marker schnell verloren, wenn die Bewegung nicht langsam genug durchgeführt wird. Die enge Varianz der Verteilungsfunktion kann innerhalb den zur Laufzeit erreichten 15 FPS nicht genug Diversität erzeugen, um schnelle Positionswechsel zu simulieren. Wurde die Varianz für Translationen erhöht, litt darunter die Genauigkeit des Trackings oder führte dazu, dass die Position der Partikel und somit der errechneten Kameraposition stark variierte und es zu hohem Jittering kam. Als Schlussfolgerung hätte eine Stufenweise Dekrementierung der Varianz, ähnlich wie es in [Cho] umgesetzt wurde, die Ergebnisse voraussichtlich verbessert. Überraschenderweise schnitt das Multinomial Resampling besser ab als das Group-Based Resampling. Dies wäre zurückzuführen, dass es unter realen Bedingungen wichtig ist die Diversität der Verteilung hochzuhalten und ein guter Kompromiss zwischen der Geschwindigkeit wie schnell ein Marker verfolgt werden kann und wie erfolgreich der Marker überhaupt gefunden wird. Abbildung 17 zeigt zwei Versuche mit der Stereokamera. In beiden Beispielen kann man sehen wie ungenau der Marker ist. Bei leichten Kammeradrehungen wird der Marker schnell verloren und nur selten wiedergefunden.

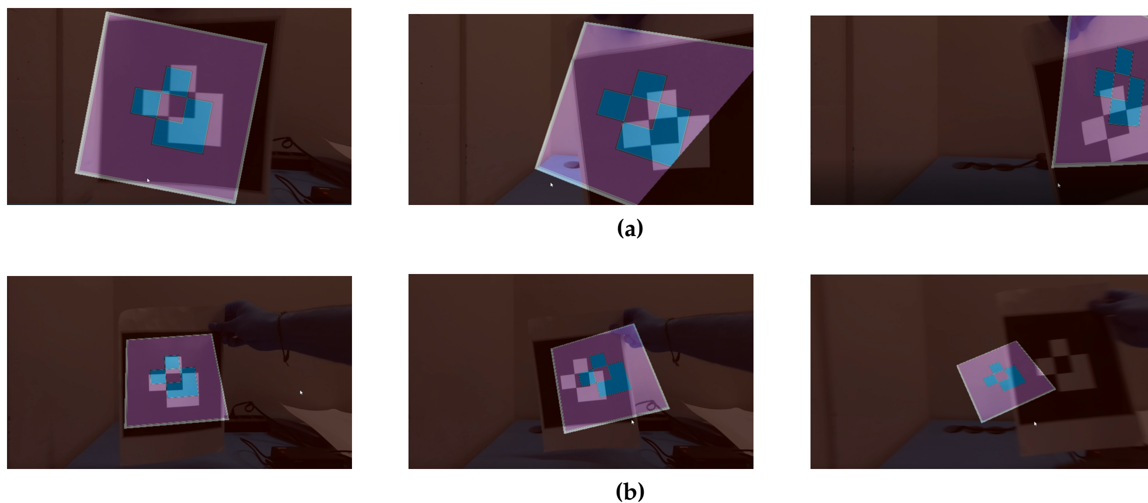


Figure 17: Tracking mit der Stereokamera anhand von HSV-Normalen-Likelyhood. (a) Zeigt eine Bewegung des Markers nach Links. (b) Kameradrehung.

6.13 Auflistung aller Durchschnittswerte

Kamerafahrt (1) Group-Based Resampling				
Likelyhood	X	Y	Z	Winkel
HSV	1.65mm	1.21mm	4.91mm	7.8 Grad
Normalen	/	/	/	/
Tiefenwerte	/	/	/	/
HSV und Normalen	1.35mm	1.01mm	3.65mm	0.49 Grad
HSV und Tiefenwerte	1.2mm	0.83mm	2.61mm	5.28 Grad
Normalen und Tiefenwerte	6.62mm	3.18mm	4.46mm	1.35 Grad
Kamerafahrt (2) Group-Based Resampling				
Likelyhood	X	Y	Z	Winkel
HSV	5.14mm	2.46mm	10.1mm	7.67 Grad
Normalen	/	/	/	/
Tiefenwerte	/	/	/	/
HSV und Normalen	4.79mm	2.39mm	10.7mm	0.56 Grad
HSV und Tiefenwerte	2.42mm	1.63mm	6.06mm	4.43 Grad
Normalen und Tiefenwerte	/	/	/	/
Kamerafahrt (1) Multimodial Resampling				
Likelyhood	X	Y	Z	Winkel
HSV	2.23mm	2.99mm	8.43mm	3.8 Grad
Normalen	/	/	/	/
Tiefenwerte	/	/	/	/
HSV und Normalen	3.19mm	2.35mm	8.93mm	1.29 Grad
HSV und Tiefenwerte	3.89mm	2.95mm	8.59mm	5.41 Grad
Normalen und Tiefenwerte	/	/	/	/
Kamerafahrt (2) Multimodial Resampling				
Likelyhood	X	Y	Z	Winkel
HSV	2.83mm	3.29mm	8.43mm	3.8 Grad
Normalen	/	/	/	/
Tiefenwerte	/	/	/	/
HSV und Normalen	3.41mm	4.1mm	11.4mm	1.6 Grad
HSV und Tiefenwerte	5.18mm	4.99mm	17.59mm	6.11 Grad
Normalen und Tiefenwerte	/	/	/	/

6.14 Technische Daten

In diesem Abschnitt werden Technische Einzelheiten zur Performance und Speicherverbrauch aufgelistet. Dabei handelt es sich um Durchschnittswerte die mit Nvidia Nsight Graphics und Nvidia Visual Profiler gesammelt wurden für die Performance der GPU. Für die Erhebung der CPU Daten wurde der Windows Task Manager benutzt. Zwischen den Implementationen, die entweder mit einer Textur des Partikelsets arbeiten (HSV-Normalen-Tiefen

Partikelfilter) und den Implementationen mit zwei Texturen (Kombinierte Partikelfilter) gab es nur marginale Unterschiede bei der Performance zur Laufzeit. Die Implementation, die auf Kanten arbeitet beanspruchte die GPU um ganze 30 mehr als die anderen Implementationen. Die deutlich höhere Auslastung kann man auf die vermehrten Fragmentshader Aufrufe zum Generieren des Kantenbildes zurückführen.

Hsv-Normalen-Tiefen Partikelfilter	
CPU Auslastung	30%
GPU Auslastung	6%
CPU Arbeitsspeicher	98.4MB
GPU Speicher	0.7/2.0GB
FPS	15
Kernel Laufzeit	27.82MS
Zeitspanne zwischen Kernelaufrufen	110MS
RegisterGraphics Laufzeit	910.54MS
Host to Device memCopy Laufzeit	30us
Device to host memCopy Laufzeit	22.1us
Kombinierte Partikelfilter	
CPU Auslastung	31%
GPU Auslastung	6%
CPU Arbeitsspeicher	99.2MB
GPU Speicher	0.7/2.0GB
FPS	15
Kernel Laufzeit	27.56MS
Zeitspanne zwischen Kernelaufrufen	132MS
RegisterGraphics Laufzeit	1.2S
Host to Device memCopy Laufzeit	29.7us
Device to host memCopy Laufzeit	22.8us
Kanten-Likelyhood Partikelfilter	
CPU Auslastung	35%
GPU Auslastung	36%
CPU Arbeitsspeicher	361.4MB
GPU Speicher	1.1/2.0GB
FPS	13
Kernel Laufzeit	35.26MS
Zeitspanne zwischen Kernelaufrufen	311MS
RegisterGraphics Laufzeit	890MS
Host to Device memCopy Laufzeit	29.7us
Device to host memCopy Laufzeit	32.4us

6.15 Rückschlüsse

Aus den Ergebnissen der Versuche mit den drei Likelyhood-Funktionen von Hsv-Farbwerten, Normalen und Tiefenwerte und deren Kombination miteinander lassen sich folgende Rückschlüsse ziehen.

- Farben reichen aus um einen planaren Marker zu tracken solange der Marker kontrastreich genug ist zum Hintergrund.
- Normalen und Tiefenwerte sind stark von der Umgebung abhängig. Da planare Marker im Anwendungsfall oft auf flachen Oberflächen platziert werden, ist es schwer den Marker von der Oberfläche anhand von Normalen oder Tiefenwerten zu unterscheiden.
- Normalen helfen den bei Rotationen die Partikel stabil zu halten, wie die Ergebnisse der Kombination aus HSV Farben und Normalen suggerieren.
- Tiefenwerte veringern das Abdriften der Partikel in x- oder y-Richtung, wie die Versuche mit der Kombination von HSV Farben und Tiefenwerten nahelegen.
- Tiefenwerte erweisen sich als sehr robust, wenn Marker und Hintergrund ausreichend Abstand zueinander haben. Im Vergleich zu Farbwerten sind Tiefenwerte auch geeignet den Marker zu tracken, wenn der Abstand zur Kamera groß ist und die Schärfe des Bildes darunter leidet.
- Kantentracking ist stark vom Preprocessing abhängig. Werden viele Kanten im Bild generiert, passen sich die Partikel der größtmöglichen Fläche an die eine ähnliche Form wie der Marker hat.
- Die teilweise befriedigenden Ergebnisse mit dem Multinomial Resampling unter realen Bedingungen suggerieren, dass ein Verfahren mit einem *Annealed Particle Filter* wie es in [Ped] und [Kle] vorgestellt wurde, besser für reale Bedingungen geeignet sind.
- Normalen und Tiefenwerte lohnen sich eher zum Tracken von 3D-Objekten, die auch mitten im Raum stehen können, als planare Marker, die in der Regel ebenfalls nah am Hintergrund platziert werden.
- Die Auslastung der GPU legt nahe, dass die Kernelfunktionen noch weiter ausgelastet werden können, der limitierende Faktor jedoch der Speicher auf der GPU darstellt. Um mehr Partikel rendern zu können sollten die Partikel über mehrere Texturen verteilt werden und pro Renderschleife mehrere Kernels gleichzeitig starten, die diese bewerten, anstatt nur einen Kernel, der auf einer großen Textur arbeitet.

7 Fazit

Das Ziel der Arbeit war es einen Trackingsystem für Marker zu implementieren, dass auf Basis eines Partikelfilters läuft und unterschiedliche Likelyhood-Funktionen zu evaluieren, die sich aus Daten ergaben, die die Stereokamera berechnen konnte. Die Implementation mit der Stereokamera erwies sich als nicht robust genug, konnte dennoch moderate Ergebnisse liefern.

Für die Gewichtung wurden die Likelyhood-Funktionen auf der GPU ausgeführt und mit CUDA implementiert. Aus der Evaluation ging hervor, dass vor allem der Speicher und die größe der Partikeltexturen entscheidend sind für eine gute Performance. Da nur der Global Speicher in der Implementation genutzt wurde, wäre eine mögliche Optimierung den Local Speicher der Blöcke zu nutzen wie in [Yang] nahegelegt wird.

Um bessere Daten erheben zu können wurde eine virtuelle Umgebung implementiert, die die Positionen des Markers und der Partikel miteinander direkt vergleichen kann. Damit konnte man bessere Aussagen treffen, wann und an welchen Parametern das Tracking scheitert oder besonders robust ist.

Aus der Evaluation ging hervor, dass besonders Farbwerte einen starken Einfluss haben bei der initialen Lokalisation des Markers. Normalen und Tiefenwerte benötigen neben guter Beleuchtung, die auch bei HSV-Farbwerten vorausgesetzt werden, auch eine geeignete Umgebung für ein robustes Tracking. So profitiert die Likelyhood-Funktion für Tiefenwerte von Entfernungen ab einem Meter und mehr zwischen Marker und Hintergrund. Normalen hingegen können eingesetzt werden, um das Jittering bei Rotationen zu reduzieren. Naheliegend wäre eine Optimierung durch die Nutzung der Local Memory auf der GPU, sowie mehr Partikel, die auf Texturen verteilt werden. Mit diesen Optimierungen wäre eine Implementation für 3D-Objekte denkbar, da die Evaluation naheliegt, dass weder HSV-Farbwerte, Normalen oder Tiefenwerte gänzlich abhängig sind von der Form des zu trackenden Models. Tatsächlich leidet des Tracking mit Normalen und Tiefenwerten an der planaren Struktur des Markers und würde von einer komplexeren Form des Markers profitieren, da der Partikel mehr Normalen hätte, die sich von ebenen Hintergrundflächen abheben könnten.

References

[Abaw] *Accuracy in Optical Tracking with Fiducial Markers: An Accuracy Function for ARToolKit*, D. F. Abawi, J. Bienwald und R. Dorner, 2004 ISMAR Proceedings of the 3rd IEEE/ACM International Symposium on Mixed and Augmented Reality

[Brow] *A Framework for 3D Model-Based Visual Tracking Using a GPU-Accelerated Particle Filte*, J. Anthony Brown und David W. Capson, 2011 IEEE Transactions on Visualization and Computer Graphics

[Cho] *RGB-D Object Tracking: A Particle Filter Approach on GPU* , Changhyun Choi und Henrik I. Christensen, 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems

[Con] *Real-Time Camera Tracking Using Known 3D Models and a Particle Filter*, Mark Pupilli und Andrew Calway, 2006 18th International Conference on Pattern Recognition (ICPR'06)

[Cuda] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#introduction , 16.05.2019, 13:00 GMT+1

[Cuda2] *CUDA by Example: An Introduction to General-Purpose GPU* , Jason Sanders und Edward Kandrot, 2010, Seite 7.ff

[Har] *Tracking with Rigid Objects*, C. Harris, 1992 MIT Press

[Kle] *Full-3D Edge Tracking with a Particle Filter*, Georg Klein und David Murray, 2006 British Machine Vision Conf. (BMVC)

[Kol] *Real-time vision-based camea tracking for augmented reality applications*, D. Koller, G. Klinker, E. Rose, D. Breen, R. Whitakr und M. Tuceryan, 1997 ACM Symposium on Virtual Reality Software and Technology

[Li] *Resampling Methods for Particle Filtering: Classification, implementation, and strategies*, Tiancheng Li, Miodrag Bolic, und Petar M. Djuric, 2015 IEEE Signal Processing Magazine

[Loz] *Real-time Visual Tracker by Stream Processing*,Oscar Mateo Lozano und Kazuhiro Otsuka, 2008

[Luc] *An iterative image registration technique with an application to stereo vision*, B. Lucas und T. Kanade, 1981 International Joint Conference on Artificial Intelligence

[Mue] *6-DoF Particle Filter-based Tracking of Arbitrarily Shaped Objects*, David Münch, 2010 Faculty of Informatics Institute for Anthropomatics

[OpenGL Wiki] khronos.org/opengl/wiki/Framebuffer_Object , https://www.khronos.org/opengl/wiki/Textures__more, 16.5.2019 13:00 GMT-1

[Ped] *6-DoF Model-based Tracking of Arbitrarily Shaped 3D Objects* , Pedram Azad1, David Münch, Tamim Asfour, Rüdiger Dillmann, 2011 IEEE International Conference on Robotics and Automation

[Pen] *Human Motion Tracking for Rehabilitation using Depth Images and Particle Filter Optimization*, Benoıt Penelle und Olivier Debeir, 2013 2nd International Conference on Advances in Biomedical Engineering

[Ray] *INTRODUCTION TO MONTE CARLO SIMULATION* , Samik Raychaudhuri, 2008 Winter Simulation Conference

[Thu] *Probalistic Robotics* ,S. Thurn, W. Burgard und D. Fox, 2006 Massachusetts Institute of Technology

[Vio] *Robust real-time face detection*, Viola, P. und Jones, M. , 2004 International Journal of Computer Vision

[Yang] *Parallel Image Processing Based on CUDA*, Zhiyi Yang, Yating Zhu und Yong Pu , 2008 IEEE