Studienarbeit zur Multimedialen Informationsverarbeitung

Implementierung von Entscheidungsbäumen zur Objekterkennung in Echtzeit

Sandro Esquivel

Kiel, den 14. Juni 2006

Zusammenfassung:

Gegenstand der vorliegenden Studienarbeit ist die Implementierung eines Verfahrens zur Objekterkennung, das auf dem Ansatz des Featurepoint Matching basiert und Entscheidungsbäume zur Identifizierung von Featurepoints verwendet. Die Entscheidungsbäume werden in einer Offline-Phase unter Verwendung einer Vielzahl vorliegender Kameraansichten des zu lernenden Objekts auf bestimmte Referenzpunkte trainiert, wonach die Identifizierung von Featurepoints zur Laufzeit durch die generierten Bäume mit geringem Rechenaufwand erfolgend kann. Wir interessieren uns insbesondere für das Verwenden von Videosequenzen als Lerndaten.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einführung

Die automatische Erkennung von Objekten auf Eingabebildern in Echtzeit stellt einen der Schwerpunkte der Bildverarbeitung dar. Objekterkennungsalgorithmen werden etwa zur Initialisierung von Tracking-Verfahren, zur Kalibrierung bewegter Kameras, etc. verwendet. In der Arbeitsgruppe für Multimediale Informationsverarbeitung an der Christian-Albrechts-Universität zu Kiel wird zur Zeit eine umfangreiche Bibliothek erarbeitet, welche u.a. Algorithmen zur Erkennung und zum Tracking von Objekten in Echtzeit bereitstellen soll.
Ziel dieser Studienarbeit ist es, Verfahren zur automatischen Objekterkennung durch Featurepoint Matching unter Verwendung von Entscheidungsbäumen in die bestehenden Programmbibliotheken der Arbeitsgruppe zu integrieren und anhand eines konkreten Verfahrens, welches von Lepetit, Lagger und Fua in dem Artikel *"Randomized Trees for Real-Time Keypoint Recognition"* [1] vorgestellt wurde, in der Praxis zu testen. Im Gegensatz zu den Arbeiten von Lepetit, Lagger und Fua, die im wesentlichen durch affine Verzerrungen künstlich generierte Ansichten von ebenen Objekten zum Trainieren des Matchers verwenden, sollen im Rahmen dieser Studienarbeit verschiedene Verfahren zum Erzeugen der Lernansichten getestet werden: Zum einen sollen ebenfalls Ansichten von ebenen Objekten durch affine bzw. perspektivische Verzerrungen erzeugt werden, zweitens Ansichten durch Rendern von 3D-Modellen erzeugt werden, und darüberhinaus vorliegende "echte" Kameraansichten eines zu lernenden Objekts oder einer zu lernenden Szene verwendet werden. Auf diese Weise erhoffen wir, die Methode auf allgemeinere Fälle anwenden zu können und ein realistischeres, fallspezifisches Training des Matchers zu erreichen, möglichst unter Beibehaltung der Erfolgsquote der Featurepoint-Erkennung im Einsatz unter Echtzeitbedingungen, welche die Methode in ihrer ursprünglichen Anwendung auszeichnet.
In **Kapitel 2** sollen zunächst Grundlagen aus dem thematischen Umfeld der Studienarbeit kurz umrissen werden und die verwendeten Notationen geklärt werden.
In **Kapitel 3** werden die verschieden Anwendungsszenarien vorgestellt und das vorgestellte Verfahren umrissen.
Kapitel 4 beschreibt als Hauptteil der vorliegenden Arbeit die vier wesentlichen Teile des Verfahrens und deren Implementierungsaspekte im Detail, nämlich in **4.1** das Detektieren von Featurepoints, in **4.2** die Verfahren zum Erzeugen von Viewsets, in **4.3** das Ermitteln stabiler Referenzpunkte, in **4.4** die Konstruktion und das Training von Entscheidungsbäumen zum Featurepoint-Matching und in **4.5** die Poseschätzung zur Laufzeit.
In **Kapitel 5** wird das implementierte Verfahren getestet und die Ergebnisse analysiert.
In **Kapitel 6** werden die Ergebnisse der Studienarbeit abschließend zusammengefasst.
Im **Anhang** wird eine Übersicht über die implementierten Klassen gegeben.

2 Grundlagen

Abkürzungen und Symbole

Img	8-Bit-Grauwertrasterbild
w_{Img}, h_{Img}	Breite/Höhe des Bildes Img in Pixeln
$m = (x, y)$	2D-Projektion eines Punktes in der Bildebene
$M = (X, Y, Z)$	3D-Punkt im Weltkoordinatensystem
$I(x, y)$	Grauwert im Pixel (x, y)
$\tilde{I}(x, y)$	Grauwert von (x, y) im weichgezeichneten Bild
P	Perspektivische Projektionsmatrix (innere Kameraparameter)
H	Transformationsmatrix für homogene Koordinaten
E	Posematrix (Translation und Rotation bzgl. Weltkoordinatensystem)
$K = PE^{-1}$	Projektionsmatrix einer Kamera
$\mathcal{R} = \{R_1, ..., R_N\}$	3D-Referenzpunkte
$\mathcal{C} = \{-1, 1, ..., N\}$	Referenzpunktklassen (mit Hintergrundklasse -1)
$m_R^{Img} = (x_R^{Img}, y_R^{Img})$	2D-Koordinaten des 3D-Referenzpunktes R im Bild Img
$FP[_{Img}]$	Menge der 2D-Featurepoints [im Bild Img]
$F = (x_F, y_F; \alpha_F, Q_F)$	2D-Featurepoint (x_F, y_F) mit Orientierung α_F und Güte Q_F
$F^* = (F, \ell_F)$	Klassifizierter Featurepoint F mit Label $\ell_F \in \mathcal{C}$
FPD	Allgemeiner Featuredetektor
LGD	Laplacian-Greedy-Featuredetektor
KLT	Kanade-Lucas-Tomasi-Featuretracker
$LoG(x, y)$	Laplacian of Gaussian-Wert im Pixel (x, y)
$\mathcal{P}$	Menge aller Patches (quadratische Bildausschnitte fester Größe)
$p_{x,y}^{Img}$	Patch des Bildes Img zentriert in (x, y)
p_F	Patch zum 2D-Featurepoint F (für $F \in FP_{Img} : p_F := p_{x_F, y_F}^{Img}$)
$Y_{\mathcal{R}} : FP \to \mathcal{R} \cup \{\perp\}$	Featurepoint-Matcher zu den Referenzpunkten $\mathcal{R}$
$T_{\mathcal{C}} : FP \to [0, 1]^{\mathcal{C}}$	Entscheidungsbaum für die Klassen $\mathcal{C}$
$(d_1^T(F), ..., d_N^T(F))$	Klassenverteilung zum Featurepoint F im Baum T
$D_1^Y, ..., D_N^Y$	Identifikations-Schwellwerte im Matcher Y
$\mathcal{V}_1, ..., \mathcal{V}_N \subset FP$	Viewsets (Lernansichten) der 3D-Referenzpunkte $R_1, ..., R_N$
$\mathcal{V}_{-1} \subset FP$	Viewset aller Nicht-Referenzpunkte
$\bar{\mathcal{V}} = \bigcup_{c \in \mathcal{C}} \mathcal{V}_c$	Gemischtes Viewset (Referenz- und Nicht-Referenzpunkte)
$\mathcal{V} = \bar{\mathcal{V}} \setminus \mathcal{V}_{-1}$	Viewset aller Referenzpunkte

2.1 Transformationen und Koordinatensysteme

In der Computergrafik werden geometrische Objekte üblicherweise über 3D-Punktkoordinaten bezüglich eines festen Koordinatensystems beschrieben [8], bzw. als 4D-Punktkoordinaten im entsprechenden *homogenen Koordinatensystem*. Das absolute Koordinatensystem, in welchem die realen beobachteten Objekte beschrieben werden, wird als *Weltkoordinatensystem* bezeichnet.
Wir betrachten im folgenden starre, zeitlich unveränderte Objekte. Jede Manipulation eines Objekts kann also als *euklidische Transformation*, nämlich Verschiebung oder Drehung, beschrieben werden. Solche Transformationen werden formal als lineare Abbildungen im homogenen Koordinatensystem behandelt, können also als 4×4-Matrix beschrieben werden. Die Transformationsmatrix E, durch welche ein Objekt von seiner Ausgangslage in seine aktuelle Lage und Orientierung überführt wird, sei im folgenden als *Pose(-matrix)* des Objekts bezeichnet.

2.2 Projektionen und Kameramodelle

Als Kameramodell wird im Allgemeinen das *Lochkameramodell* verwendet [8]. Eine Kamera ist durch einen Ortspunkt C (das *optische Zentrum* der Kamera), einen Vektor $\vec{v}$ (*optische Achse*), die Länge f (*Brennweite*) und Winkel α, β (die *Öffnungswinkel* der Kamera), sowie einen Rotationswinkel ϕ (*Orientierungswinkel* der Kamera) gekennzeichnet. Jeder 3D-Punkt M der aufgenommenen Szene wird auf den Schnittpunkt der Projektionsgeraden $\overrightarrow{CP}$ mit der *Projektionsfläche* der Kamera abgebildet, welche der durch die Öffnungswinkel beschränkte rechteckige Ausschnitt der Ebene mit dem Ortspunkt $C + \frac{\vec{v}}{|\vec{v}|} f$ und dem Normalenvektor $\vec{v}$ darstellt (siehe *Abb. 1*).

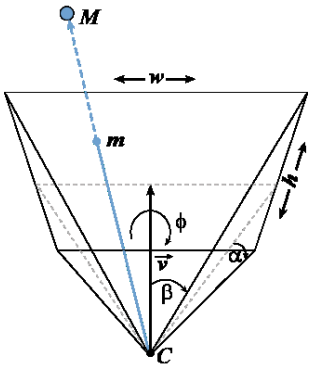


Abbildung 1: Schematische Darstellung des Lochkameramodells

Das Modell lässt sich als idealisierte Lochkamera ohne Abbildungsfehler der Linse (z.B. Distortion, siehe auch **??**) interpretieren, bei welcher C die Position der Lochblende angibt, $\vec{v}$ die Richtung der optischen Achse und f den Abstand der

Bildebene zur Lochblende, ϕ den vertikalen Drehwinkel, sowie α und β die maximalen senkrechten und horizontalen Einfallswinkel von Lichtstrahlen in die Kamera.
Die Parameter f, α und β werden als *innere Kameraparameter* oder *Kamerakalibrierung* bezeichnet und sind für jede Kamera fixiert. Die Parameter C, v und ϕ geben die *äußeren Kameraparameter* bzw. die *Kamerapose* an, also Lage und Orientierung der Kamera. Eine Kamera in Standardpose besitzt den Ortspunkt $C = 0$, die optische Achse $\vec{v} = (0, 0, 1)^T$ und die Orientierung $\phi = 0$. Eine Kamera in Nicht-Standardpose lässt sich daher auch durch eine affine Transformation der Standardpose durch $E = TR_\theta R_\psi R_\phi$ bestehend aus einer Verschiebung T um $\overrightarrow{0C}$ und Rotationen R_ϕ, R_ψ, R_θ um die Hauptachsen beschreiben.
Mathematisch beschreibt das Lochkameramodell die *perspektivische Projektion* von 3D-Punkten M des Weltkoordinatensystems auf 2D-Punkte m des Kamerabildes (siehe *Abb. 2*). Der *Hauptpunkt* (auch: *Prinzipalpunkt*) des Bildes sei hierbei derjenige 2D-Punkt in Bildkoordinaten, auf den die optische Achse abgebildet wird. Diese Projektion wird formal durch eine 3×4-Matrix $K = PE^{-1}$ mit $M = Km$ dargestellt, wobei $E \in \mathbb{R}^{4\times4}$ die Transformation von 3D-Punkten im Kamerakoordinatensystem in das Weltkoordinatensystem beschreibt (*Posetransformation*), $P \in \mathbb{R}^{3\times4}$ eine perspektivische Projektion von 3D-Punkten im Kamerakoordinatensystem in die Bildebene der Kamera. Die äußeren Kameraparameter gehen hierbei in E ein, die inneren Kameraparameter in P. Als detaillierte Referenz sei an dieser Stelle etwa auf [**?**] oder [12] verwiesen.

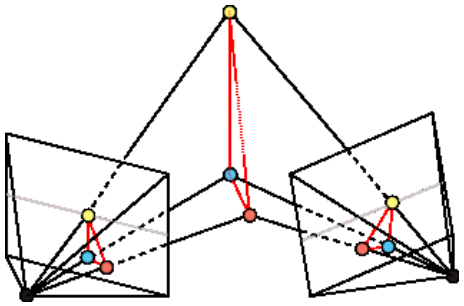


Abbildung 2: Projektionen von 3D-Punkten

2.3 Spezielle Notationen

Es sollen im folgenden die speziellen Notationen geklärt werden, die in der vorliegenden Studienarbeit verwendet werden:

Ein *Bild* Img besitze die *Höhe* h_{Img} und *Breite* w_{Img}. Wir gehen im folgenden von 8-Bit-Grauwertrasterbildern aus. Jedes *Pixel* (x, y) in Img mit $x \in \{0, ..., w_{Img} - 1\}$, $y \in \{0, ..., h_{Img} - 1\}$ besitze den Grau- bzw. *Intensitätswert*

$I(x,y) \in \{0,...,255\}$. $\tilde{I}(x,y)$ bezeichne den Wert des Pixels (x,y) in Img nach Anwendung eines GAUSS-Weichzeichners, also eines Filters, welcher durch Glättung des Bildes Störungen durch Bildrauschen reduziert. Die *Menge aller Bilder* sei durch IMG beschrieben.

Ein *Featurepoint* in einem Bild Img ist ein Bildpunkt, welcher hinreichend gut von seiner Umgebung unterschieden werden kann, also lokal auffallig ist. Die Menge aller Featurepoints [in einem Bild Img] sei als $FP[_{Img}]$ bezeichnet.
Ein *Featurepoint-Detektor* beschreibt eine Funktion $FPD : IMG \to FP$, wobei $FPD(Img) = FP_{Img}$ eine Menge von Featurepoints im Bild Img auswählt, bei einem Corner-Detektor etwa potentielle Ecken im Bild (siehe *Abb. 3*).
Ein *Featurepoint* F besitze die Koordinaten $m_F = (x_F, y_F)$ im Bild Img_F, in welchem er detektiert wurde. Je nach Detektor besitze F zusätzliche Attribute, durch welche er beschrieben wird.

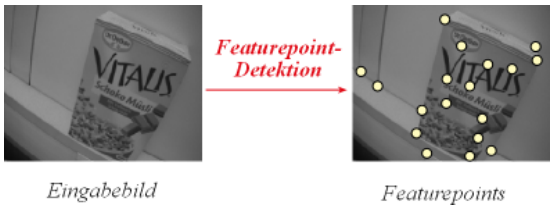


Abbildung 3: Detektion von Featurepoints

Vorliegend sei ein *Referenzobjekt*, welches später erkannt werden soll. Ein *Referenzpunkt* $R \in \mathbb{R}^3$ sei ein 3D-Punkt auf dem Referenzobjekt im Weltkoordinatensystem, der erkannt werden soll. Die *Menge der Referenzpunkte* soll im folgenden als $\mathcal{R} = \{R_1, ..., R_N\}$ bezeichnet werden. Stammen alle Referenzpunkte aus einem einzigen 2D-Referenzbild Ref, so sei die Menge der zu den Referenzpunkten gehörenden Featurepoints als FP_{Ref} bezeichnet.
Wir fixieren eine Umgebungsgröße $p_{size} \in \mathbb{N}$ und definieren als *Patch* oder *Umgebung* $p^{Img}_{x,y}$ zum Pixel (x,y) im Bild Img einen $p_{size} \times p_{size}$-Pixel-Auschnitt aus Img zentriert in (x,y). Entsprechend sei das Patch p_F zum Featurepoint $F \in FP_{Img}$ definiert als $p^{Img}_{x_F,y_F}$.
Für eine Bildmenge $Img_1, ..., Img_k$ sei das *Viewset* $\mathcal{V}_i$ (Ansichtsmenge) des Referenzpunktes $R_i \in \mathcal{R}$ definiert als die Menge aller Featurepoints in $FP_{Img_1}, ..., FP_{Img_k}$, die Ansichten des Punktes R_i darstellen (siehe *Abb. 4*). Das *Viewset aller Referenzpunkte* ist entsprechend $\mathcal{V} = \bigcup_{i=1}^{N} \mathcal{V}_i$. Die Menge aller übrigen Featurepoints in $FP_{Img_1}, ..., FP_{Img_k}$, die *keine* Ansichten der Referenzpunkte darstellen, sei das *Viewset aller Nicht-Referenzpunkte* $\mathcal{V}_{-1}$. $\bar{\mathcal{V}} = \mathcal{V} \cup \mathcal{V}_{-1}$ sei das *gemischte Viewset*.

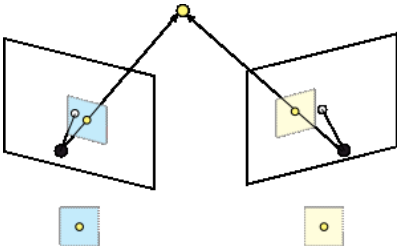


Abbildung 4: Featurepoints eines Referenzpunkts in verschiedenen Ansichten

Jedem Referenzpunkt $R_i \in \mathcal{R}$ entspreche nun eine *Klasse* $c = i$, welche die Menge aller Featurepoints des Punktes in allen möglichen Ansichten umfasst. Die Klasse c ist also im wesentlichen identisch mit $\mathcal{V}_c$ für $k \to \infty$.
Die zusätzliche *Hintergrundklasse* -1 enthalte alle Featurepoints in allen möglichen Ansichten, die *keine* Referenzpunkte darstellen, ist also identisch mit $\mathcal{V}_{-1}$ für $k \to \infty$.
Die *Menge aller Klassen* sei mit $\mathcal{C} := \{-1, 1, ..., N\}$ bezeichnet.
$F \sim c$ bedeute, dass F zur Klasse $c \in \mathcal{C}$ gehört, d.h. eine Ansicht des Referenzpunktes R_c darstellt (wenn $c \neq -1$) bzw. keinen Referenzpunkt darstellt (wenn $c = -1$).
Wir definieren die tatsächliche Klasse eines Featurepoints F - das *Label* von F - durch $\ell_F \in \mathcal{C}$, also: $\ell_F = c :\Leftrightarrow F \sim c$. Ein Viewset ist analog zu oben als *Menge von gelabelten (bereits klassifizierten) Featurepoints* zu verstehen.

Das aktuelle *Eingabebild*, in welchem zur Laufzeit das Referenzobjekt erkannt werden soll, sei mit *Input* bezeichnet.
Ein *Featurepoint-Matcher* beschreibt nun eine Funktion $Y : FP \to \mathcal{R} \cup \{\perp\}$ (siehe *Abb. 5*), wobei für ein Eingabebild *Input* mit einer möglichst großen Wahrscheinlichkeit für jeden Featurepoint $F \in FP_{Input}$ gelten soll:

$$Y(F) = \begin{cases} R_c, & \text{wenn } F \sim c,\ c \in \mathcal{C} \setminus \{-1\} \\ \perp, & \text{wenn } F \sim -1 \end{cases}$$

Ein Punkt gelte als *unidentifiziert*, wenn der Matcher $\perp$ liefert, ansonsten als *identifiziert*.

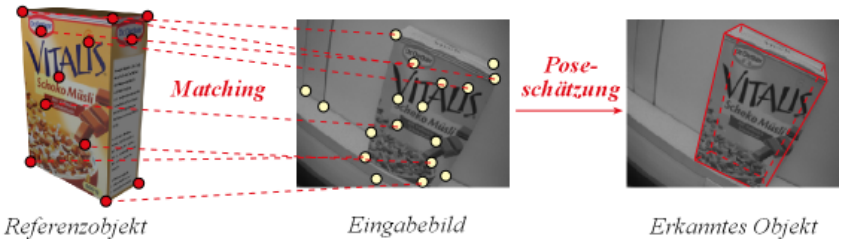


Abbildung 5: Matching von Featurepoints mit Referenzpunkten

3 Hintergrund der Anwendung

3.1 Objekterkennung durch Featurepoint Matching

Es soll zunächst kurz das Verfahren der *Objekterkennung durch Featurepoint Matching* vorgestellt werden.
Unter *Objekterkennung* versteht man in der Computergrafik das Problem, die *Pose* - also Lage und Orientierung - eines "bekannten" Objekt in einzelnen Eingabebildern oder Bildsequenzen zu erkennen. Äquivalent zu diesem Problem ist die Rekonstruktion der Kameraposition und -orientierung durch Auswertung von Kamerabildern einer "bekannten" Objektanordnung.
Wir konkretisieren das Problem wie folgt: Gegeben seien ein oder mehrere Bilder eines starren, geometrischen Objekts (*Referenzobjekt*, *Referenzbilder*), wobei die räumliche Orientierung des Referenzobjekts und die Kameraparameter der Referenzbilder als bekannt vorausgesetzt seien. Das Problem der *Poseschätzung* besteht darin, in einem beliebigen Bild (*Eingabebild*) das Referenzobjekt wiederzuerkennen, genauer: Die räumliche Lage und Orientierung (*Pose*) des Referenzobjekts aus der Vorlage (oder analog: der aufnehmenden Kamera) in dem Eingabebild abzuschätzen bzw. die Transformation, welche die Ausgangspose in die Pose im Eingabebild überführt, zu ermitteln.
Die vielversprechendsten Ansätze in diesem Bereich gehen auf den Vergleich von Merkmalspunkten in Eingabebild und Referenzbildern zurück (*Featurepoint-Matching*). Als Merkmalspunkt oder *Featurepoint* werden lokale Bildmerkmale bzw. Pixel eines Eingabebildes bezeichnet, die bezüglich eines festgelegten Erkennungsverfahren "auffällig" sind und in verschiedenen Ansichten hinreichend stabil wiedererkannt werden. Es existieren viele klassische Methoden für die Erkennung von Featurepoints (siehe *Abb. 3*); in der Regel wird ein statistisches Maß definiert, welches möglichst invariant gegenüber 3D-Projektionstransformationen, Beleuchtungsänderungen und Diskretisierungsfehlern ist. Ein Beispiel für solche Erkennungsverfahren (*Featurepoint-Detektoren*) ist etwa der Harris-*Corner-Detektor* [11], welcher *Ecken* in einem Bild detektiert - also Punkte mit hoher Variation der Intensitätsfunktion $I(x, y)$ in x- und y-Richtung. Zum Featurepoint-Matching liegen zunächst 3D-Referenzpunkte aus Objektansichten mit bekannter Pose vor. Zur Poseschätzung zur Laufzeit wird nun versucht, Featurepoints, die in einem Eingabebild detektiert wurden, mit den Referenzpunkten zu identifizieren und aus den korrespondierenden 2D-3D-Punktpaaren die Transformation der Pose zwischen den jeweiligen Bildern zu berechnen (siehe *Abb. 5*). Auch hier werden in der Regel möglichst gegenüber verschiedenen Ansichten invariante Beschreibungen der Featurepoints vorgenommen und ein statistisches Ähnlichkeitsmaß zwischen Featurepoint-Beschreibungen verwendet, um die Identifizierung vorzunehmen. Als Referenz gilt momentan das *SIFT*-Verfahren [7], welches eine skalierungsinvariantes Beschreibung von Featurepoints liefert, die eine reproduzierbare Wiedererkennung zulässt.

3.2 Klassifizierung durch Entscheidungsbäume

In der vorliegenden Arbeit untersuchen wir die Klassifizierung von Featurepoints - also die Identifizierung mit gegebenen 3D-Referenzpunkten - anhand von Entscheidungsbäumen. Eine Klasse umfasse hierbei die Menge aller Featurepoints, die einen Referenzpunkt in allen möglichen Ansichten darstellen.
Ein Featurepoint durchläuft hierbei eine Reihe aufeinanderfolgender Tests, wobei die Wahl des jeweiligen Test von dem Ergebnis des vorausgegangenen Tests abhängt. Die Testsequenz impliziert eine Baumstruktur, wobei die inneren Knoten mit unterschiedlichen Testparametern assoziiert sind, die Blätter mit dem Ergebniswert der Testsequenz. Als Ergebnis wird eine Wahrscheinlichkeitsverteilung zurückgegeben, mit welcher der getestete Featurepoint zu einer der Klassen gehört - also mit jedem Referenzpunkt identifiziert werden kann.
Ein Entscheidungsbaum T beschreibe also eine Funktion: $T : FP \rightarrow [0,1]^{\mathcal{C}}$, wobei einem Featurepoint $F \in FP_{Img}$ eine Wahrscheinlichkeitsverteilung $d \in [0,1]^{\mathcal{C}}$ zugeordnet wird. d_c wird dabei als die Wahrscheinlichkeit interpretiert, mit der F zur Klasse $c \in \mathcal{C}$ gehört, d.h. eine Ansicht des Referenzpunkts R_c darstellt.
Ein innerer Knoten v in T beschreibe einen Test $\nu : FP \rightarrow 1, ..., k_\nu$ mit k_ν Resultatklassen und besitze k_ν Kinder $v_1, ..., v_{k_\nu}$, die mit je einer Resultatklasse assoziiert sind.
Jedes Blatt v von T besitze einen Wahrscheinlichkeitsvektor $d^v \in [0,1]^{\mathcal{C}}$.
Die Klassifizierung eines Featurepoints $F \in FP_{Img}$ erfolgt durch wiederholtes Testen und Weiterreichen ausgehend von der Wurzel ρ, bis ein Blatt erreicht wird - formal durch die Rekursion:

$$T(F) = eval(F, \rho),\ eval(F, v) = \begin{cases} eval(F, v_{\nu(F)}), & \text{falls } v \text{ innerer Knoten} \\ d^v, & \text{falls } v \text{ Blatt} \end{cases}$$

Matching durch einzelne Entscheidungsbäume: Das Matching erfolgt nun durch Auswahl des Referenzpunkts R_c mit der größten Wahrscheinlichkeit d_c.
Wir beschränken uns in der vorliegenden Arbeit auf Bäume, welche ausschließlich eine bestimmte Art von Test verwenden (siehe 4.1.3), wobei nicht auszuschließen ist, dass derselbe Baum verschiedenartige Tests verwenden kann. Die Implementierung soll daher auch andere Fällen zulassen, insbesondere soll jeder Knoten beliebige Testparametertypen enthalten können.
Wir folgen im Rahmen der Studienarbeit dem Vorschlag aus [2] und fixieren einen Test, welcher Intensitäten je zweier Pixel aus der Umgebung der Featurepoints vergleicht und drei verschiedene Ergebnisse liefert, womit wir einen ternären Baum als Datenstruktur erhalten (siehe *Abb. 6*). Jeder Knoten enthalte als Testparameter die Koordinaten der zu vergleichenden Pixel relativ zu den Koordinaten und der Orientierung des betrachteten Featurepoints.

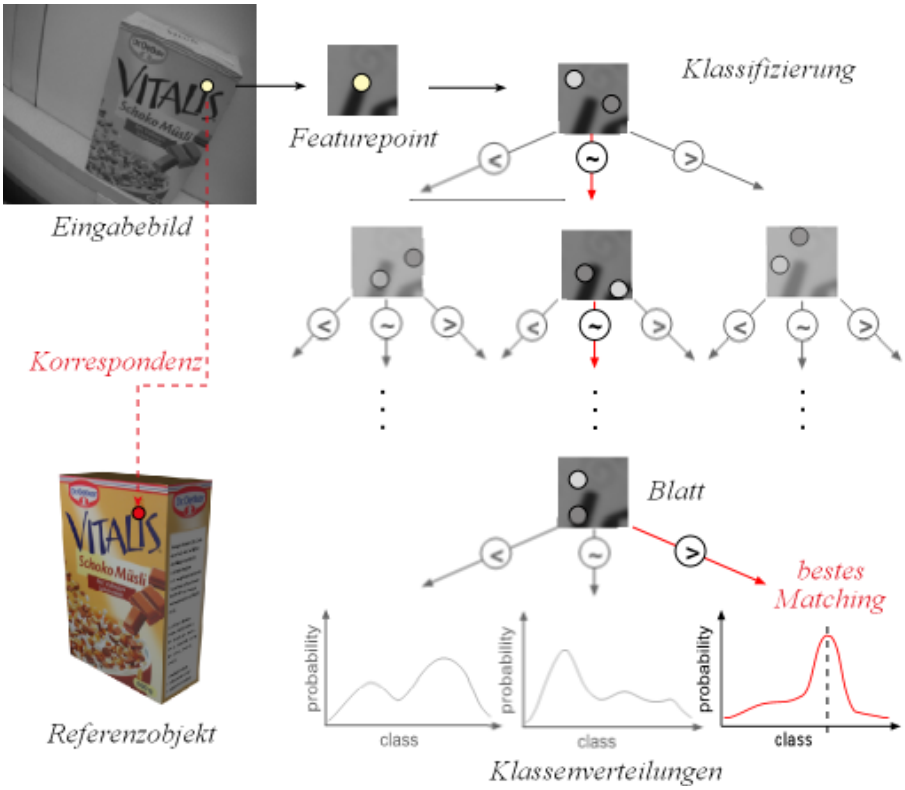


Abbildung 6: Klassifizierung von Featurepoints durch Entscheidungsbäume

Matching durch mehrere Entscheidungsbäume: Um Laufzeit und Speicher zu sparen, werden wir im folgenden mehrere suboptimale Entscheidungsbäume mit jeweils geringeren Eingabedaten erzeugen und den Mittelwert aller Bäume als Ergebnis zur Klassifizierung verwenden. Wir erhalten mit K Entscheidungsbäumen $T_1, ..., T_K$ also den Matcher $Y_{\mathcal{R}} : FP \to \mathcal{R} \cup \perp$ durch:

$$Y_{\mathcal{R}}(F) = \begin{cases} R_c, & d_c > D_c^Y \\ \perp, & \text{ansonsten} \end{cases}, \text{ wobei } c = \arg\max_{c \in \mathcal{C}} d_c,\ d_c = \sum_{i=1}^{K} d_c^{T_i} \text{ und } d^{T_i} = T_i(F).$$

$D_c^Y > 0$ ist hierbei ein festzulegender Schwellwert für jede Klasse $c \neq -1$, der angibt, ab welcher Wahrscheinlichkeit ein klassifizierter Featurepoint als ausreichend zuverlässig identifiziert gilt.

3.3 Betrachtete Anwendungsfälle

Das im Rahmen dieser Studienarbeit entwickelte Verfahren zur Objekterkennung soll in den folgenden vier Szenarien getestet werden.

- Erkennung von planaren Objekten in 3D-Szenen
- Erkennung von Objekten mit 3D-Modell
- Erkennung von Objekten oder Szenen aus Videosequenzen
- Erkennung der Kamerapose in Panoramaansichten

Wir gehen in allen Fällen davon aus, dass die beobachteten Objekte starr und unbewegt sind.

3.3.1 Erkennung von planaren Objekten in 3D-Szenen

Gegeben sei ein planares oder stückweise planares Objekt (z.B. ein Buchdeckel, eine Schachtel, etc.), dessen Pose in einem Eingabebild wiedererkannt werden soll. Als Referenzbild wird eine Frontalansicht des Objekts (bzw. jeder ebenen Objektseite) verwendet (siehe *Abb. 7*).

Abbildung 7: Erkennung von planaren Objekten in 3D-Szenen

3.3.2 Erkennung von Objekten mit 3D-Modell

Gegeben sei ein beliebig komplex geformtes Objekt, zu dem ein 3D-Modell vorliegt, dessen Pose in realen Aufnahmen wiedererkannt werden soll.
Als Referenzbilder werden eine oder mehrere Ansichten verwendet, die man durch Rendern des 3D-Modells erhält (siehe *Abb. 8*).

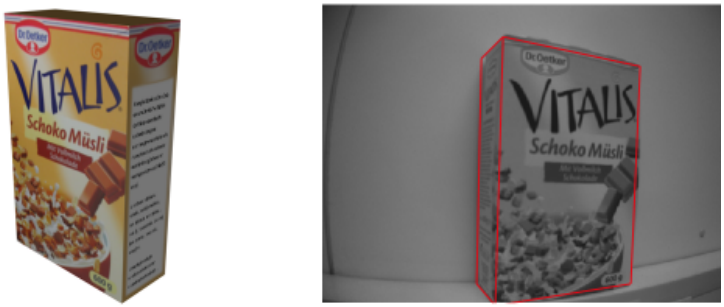


Abbildung 8: Erkennung von Objekten mit 3D-Modell

3.3.3 Erkennung der Kamerapose in Panoramaansichten

Gegeben sei eine Szene, sowie als Referenzbild eine Panoramaaufnahme, also eine perspektivische Weitwinkelprojektion der Szene, wobei die Parameter der aufnehmenden Kamera bekannt seien.
Es soll die Orientierung einer Betrachterkamera der Szene (mit bekannter Kalibrierung) anhand deren Aufnahme ermittelt werden, wobei wir für die Betrachterkamera nur beschränkte Posen zulassen werden (siehe *Abb. 9*).

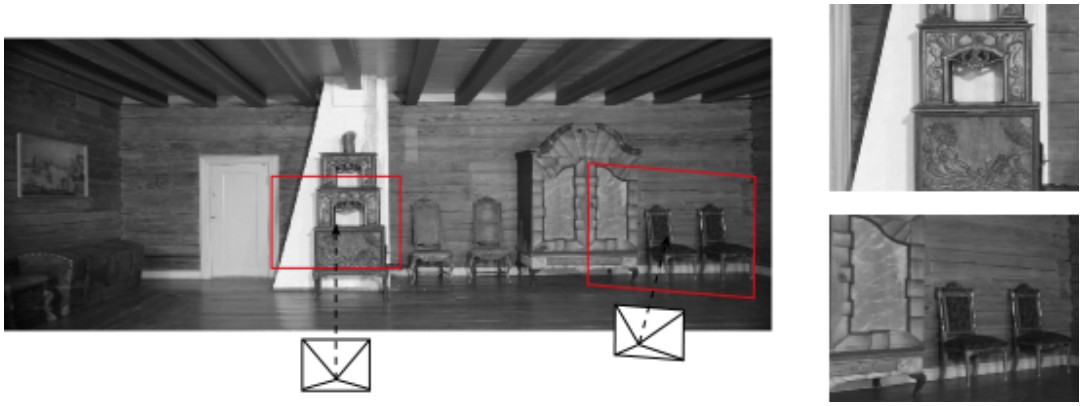


Abbildung 9: Erkennung der Kamerapose in Panoramaansichten

3.3.4 Erkennung von Objekten aus Videosequenzen

Gegeben sei ein beliebig komplexes Objekt, dessen Pose in verschiedenen Aufnahmen wiedererkannt werden, für welches allerdings kein 3D-Modell vorliegt. Statt Referenzbildern liegt mindestens eine (echte oder gerenderte) Videosequenz vor, welche das Objekt bzw. die Szene aus möglichst vielen verschiedenen Perspektiven abbildet. Die Kameraparameter in den Einzelbildern seien als bekannt oder rekonstruierbar vorausgesetzt. In unserem Fall genügt es, dass die Kamerakalibrierung und die Kamerapose im ersten Sequenzbild bekannt sind.

4 Objekterkennung durch Entscheidungsbäume

Der Hauptteil beschreibt die Implementierung des Verfahrens zur Objekterkennung durch allgemeine Entscheidungsbäume, wobei wir uns an [1] orientierten. Der Aufbau des Kapitels folgt dem chronologischen Verlauf der Studienarbeit.
Zunächst erläutern wir in 4.1 den verwendeten Algorithmus zum Detektieren von Featurepoints in Ansichten. Wir verwenden hierzu einen einfachen Detektor aus [2], der in der ersten Phase der Studienarbeit implementiert und getestet wurde. Zusätzlich gehen wir auf den verwendeten Test zum Klassifizieren von Featurepoints ein.
In 4.2 wird erläutert, wie die Erzeugung von Ansichten von Featurepoints (*Viewsets*) für die einzelnen Anwendungsfälle konzipiert und realisiert wurde.
In 4.3 gehen wir auf das Ermitteln günstiger 3D-Referenzpunkte für das Featurepoint-Matching für die einzelnen Anwendungsfälle ein.
In 4.4 wird schließlich die Realisierung der Entscheidungsbäume und des Featurepoint-Matchers, die in 3.2 theoretisch beschrieben wurden, unabhängig von den betrachteten Anwendungsfällen erläutert, wobei wir die Ergebnisse aus [1] als Grundlage unserer Arbeit verwenden.

4.1 Erkennung von Featurepoints

Essentiell für die Objekterkennung durch Featurepoint-Matching ist der verwendete Featurepoint-Detektor. Für unsere Studien verwenden wir den in [2] vorgeschlagenen Detektor unter der Bezeichnung *Laplacian-Greedy-Detektor* (*LGD*). Prinzipiell kann aber jeder beliebige ausreichend schnelle und robuste Featurepoint-Detektor verwendet werden. Hierbei ist zu berücksichtigen, dass im Gegensatz zu Trackingverfahren für das hier vorgestellte Verfahren in jedem Bild unabhängig voneinander Featurepoints detektiert werden müssen, die Laufzeit des verwendeten Detektionsalgorithmus also besonders kritisch ist.

4.1.1 Der Laplacian-Greedy-Detektor

In der ersten Phase der Studienarbeit sollte der Laplacian-Greedy-Detektor in die bestehende Programmbibliothek analog zu den bereits vorhandenen Detektoren integriert werden. An dieser Stelle soll zunächst die Funktionsweise des Detektors erläutert, die Ergebnisse der Testläufe vorgestellt und auf die Implementierung eingegangen werden.

Erkennung von Featurepoints: Der Algorithmus versucht solche Punkte zu ermitteln, die weder in einheitlichen Bereichen noch auf Kanten liegen, also etwa Ecken oder lokal abgesetzte Texturierungen. Der Algorithmus prüft dazu zunächst

jeden Punkt m im Bild Img, indem die Intensitäten (Grauwerte) je zweier gegenüberliegender Pixel m_1, m_2 auf einem Kreis C mit Radius r um m mit der Intensität $\tilde{I}(m)$ von m verglichen werden. Existiert ein Paar, dessen Intensität sich nur unwesentlich von $\tilde{I}(m)$ unterscheidet, wird der Punkt m verworfen (m liegt z.B. auf einer Kante oder in einem uniformen Bereich, siehe *Abb.10(a)*). Anderenfalls gilt m als möglicher Featurepoint (m liegt z.B. auf einer Ecke, siehe *Abb.10(b)*).
m gilt also als Featurepoint-Kandidat, wenn für alle $\alpha \in [0, \pi]$ gilt: $|\tilde{I}(m) - \tilde{I}(m + dR_\alpha)| > \tau$ oder $|\tilde{I}(m) - \tilde{I}(m - dR_\alpha)| > \tau$, wobei $dR_\alpha = (R * \cos(\alpha), R * \sin(\alpha))$ und $\tau > 0$ eine zuvor festgelegte Toleranzgrenze ist.
Da wir in der Praxis mit diskreten Punkten arbeiten, wählen wir $\alpha_1, ..., \alpha_k$ so, dass der diskretisierte Kreis um m abgetastet wird, und vergleichen auch Nachbarpixel der diametral gegenüberstehenden Pixel, um Diskretisierungsfehler zu vermeiden (siehe *Abb.10(c)*).

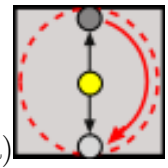
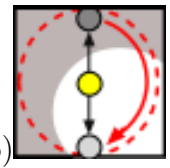
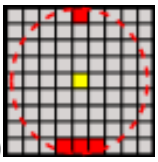
(a)  (b)  (c) 

Abbildung 10: Erkennung von Featurepoints
(a) kein Featurepoint (b) Featurepoint (c) Vergleich von Nachbarpixeln

Auswahl lokal optimaler Featurepoints: Um Häufungen von Featurepoints zu vermeiden, wird für die ermittelten Punkte wie in [2] beschrieben der *"Laplacian of Gaussian"* $\nabla^2 \tilde{I}$ (allgemein LoG) approximiert und nur die Punkte an den lokalen Optima als Featurepoints angenommen. Der LoG stellt hierbei ein isotropes Maß für die 2. Ableitung eines Bildes dar, wobei zunächst ein Gaussfilter anwendet wird, um die Sensitivität gegenüber Störungen zu vermindern (detaillierte Beschreibung siehe z.B. [13]). Durch den LoG wird also die "Geschwindigkeit" des Intensitätsabfall bzw. -anstieg in einem Punkt beschrieben; Punkte mit lokal hoher Intensitätveränderung (z.B. Eckpunkte, Kantenpunkte) werden somit bevorzugt gewählt.
Die Approximation des LoG mit $\sigma_{Gauss} = 3.5$ erfolgt während des Kreisscans effizient mit der Näherung:
$LoG(m) \approx \sum_{a \in [0,\pi]} \tilde{I}(m - dR_\alpha) + \tilde{I}(m + dR_\alpha) - \tilde{I}(m)$.
Zur Auswahl der lokalen Optima betrachten wir die 5×5-Umgebung der Featurepoint-Kandidaten und verwerfen Punkte, deren LoG-Wert in dieser nicht optimal ist. Featurepoints haben hier also einen Mindestabstand von 3 Pixeln.

Zuweisung von Orientierung und Güte: Während des Kreisscans können neben dem *LoG*-Wert weitere Informationen ermittelt werden, die zur Beschreibung der Featurepoints dienen. Wir ermitteln eine *normierte 2D-Orientierung* α zum Punkt m im Bild Img, um Rotationsinvarianz in der Ebene zu ermöglichen, sowie eine *Güte der Erkennung* $Q \in [0, 1]$.
Wir ermitteln dazu während des Kreisscan den Punkt $\hat{m}$ auf dem Kreis mit Radius r um m, der die größte Intensitätsdifferenz zu m aufweist. Der Vektor $\overrightarrow{m\hat{m}}$ legt die normierte Orientierung fest (siehe *Abb.11*). Zur Gütebestimmung des Featurepoints verwenden wir seinen *LoG*-Wert. Wir bestimmen also konkret:
$\alpha = \arg \max_{\alpha \in [0,2\pi]} |\tilde{I}(m) - \tilde{I}(m + dR_\alpha)|$ und $Q = 1/LoG(m)$.

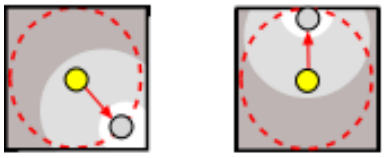


Abbildung 11: Normierte Orientierung von Featurepoints

Sortierung nach Güte: Die Menge der Featurepoints, die nach der Optimalitätsprüfung des *LoG*-Wertes übrigbleiben, werden abschließend nach Güte sortiert ausgegeben. Wir verwenden aus Effizienzgründen höchstens die N_{max} besten Featurepoints, wobei wir N_{max} zwischen 1200 und 2000 wählen.
Ein Featurepoint $F \in LGD(Img)$, den der Laplacian-Greedy-Algorithmus liefert, sei im folgenden stets durch $F = (x_F, y_F; \alpha_F, Q_F)$ beschrieben, wobei (x_F, y_F) die 2D-Koordinaten von F in Img angibt, α_F die normierter Orientierung und Q_F die Güte der Erkennung. *Abb. 12* stellt die Ausgabe des Verfahrens, sowie die *LoG*-Werte der ermittelten Featurepoint-Kandidaten dar.

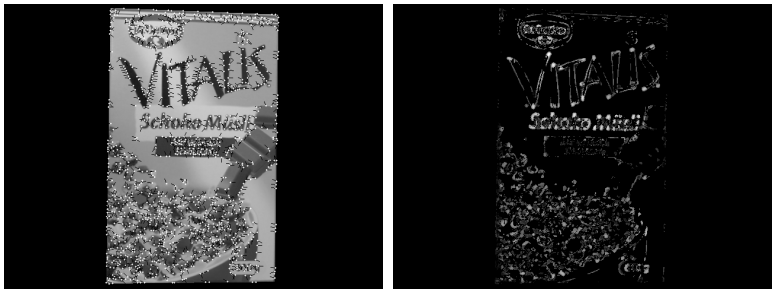


Abbildung 12: Detektierte Featurepoints und Laplacian of Gaussian-Werte

Analyse des Algorithmus: Die Vorteile des Detektors liegen in der prinzipiell einfachen Berechnung, sowie der Invarianz gegenüber Beleuchtungsänderungen und 2D-Rotationen, auf der anderen Seite ist die Detektion nicht skalierungsinvariant und ineffektiv bei schwach texturierten Objekten.
Die relativen Pixelkoordinaten des Kreises werden aus Effizienzgründen in der Implementierung vorausberechnet, wodurch für den Kreisscan nur wenige Berechnungen pro Schritt notwendig werden.

4.1.2 Verwendung anderer Featurepoint-Detektoren

Da wir uns im Entwurf des Laplacian-Greedy-Detektors an der Struktur bestehender Detektor-Klassen aus der BIAS-Bibliothek orientiert haben, können die Ergebnisse unserer Studienarbeit leicht mit anderen Detektoren reproduziert werden. In die vorliegenden Arbeit fließen allerdings keine Untersuchungen mit anderen Detektoren ein, da wir einen Test für die Klassifizierung von Featurepoints verwenden, der eine 2D-Orientierung der detektierten Featurepoints benötigt, welche bisher durch keinen bestehende Detektor attributiert wird.

4.1.3 Testen von Featurepoints durch Pixelvergleiche

Für das Testen von Featurepoints durch den Entscheidungsbaum können prinzipiell beliebige Tests der Form $\nu : FP \rightarrow 1, ..., k_\nu$ für ein $k_\nu \in \mathbb{N}$ verwendet werden, die ein Featurepoint F einer Resultatklasse $1, ..., k_\nu$ zuordnen. In unserer Implementierung wird dieser Sachverhalt durch ein Interface `IDecisionTreeTest` modelliert.
Wir verwenden den in [2] beschriebenen Test, der die Intensitäten zweier Pixel aus einer Umgebung fester Größe p_{size} von F im Bild Img_F vergleicht und einer der drei Resultatklassen *signifikant kleiner*, *signifikant größer* und *ungefähr gleich* bezüglich einer festen Toleranz τ zuordnet. Der Test besitze als Parameter dx_1, dy_1, dx_2, dy_2 die Koordinaten der zu vergleichenden Punkte relativ zur Position m_F und Orientierung α_F von F (siehe *Abb. 13*). Zur Auswertung wird vorausgesetzt, dass F das $p_{size} \times p_{size}$-Patch p_F als Deskriptor besitzt. Die Resultatklasse $\nu(F)$ ergibt sich dann durch:
$m_1 := m_F + R_{\alpha_F}(dx_1, dy_1), m_2 := m_F + R_{\alpha_F}(dx_2, dy_2)$
$\Delta\tilde{I} := \tilde{I}(m_1) - \tilde{I}(m_2)$

$$\nu(F) := \begin{cases} 1, & \text{falls } \Delta\tilde{I} < -\tau \\ 2, & \text{falls} -\tau \leq \Delta\tilde{I} \leq +\tau \\ 3, & \text{falls} \Delta\tilde{I} > +\tau \end{cases}$$

Analyse des Algorithmus: Die Vorteile des Tests liegen in der prinzipiell einfachen Berechnung (ca. $1ns$ pro Auswertung bei unoptimierter Implementierung), der geringen Parameterzahl, sowie der offensichtlichen Invarianz gegenüber Beleuchtungsänderungen und 2D-Drehungen.

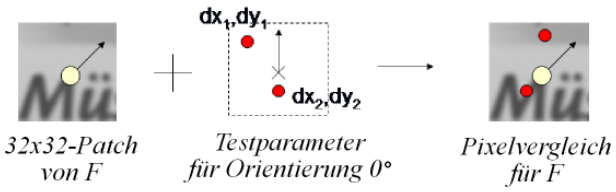


Abbildung 13: Pixelvergleichstest

Auf der anderen Seite ist der Test nicht skalierungsinvariant und ineffektiv bei schwach texturierten Objekten.

4.2 Erzeugen von Viewsets

Das *"Lernen"* der Referenzpunkte benötigt eine sehr große Menge von Featurepoints aus möglichen Eingabebildern, die Ansichten der Referenzpunkte darstellen, wobei die Klassenzugehörigkeit jedes Featurepoints bekannt sein muss. Um diese Menge, das *Viewset aller Referenzpunkte* $\mathcal{V}$, zu erzeugen, werden verschiedene Ansichten $Img_1, ..., Img_k$ des Referenzobjekt benötigt, sowie zusätzliche Informationen zu jeder Ansicht, aus welcher die 3D-Position jedes 2D-Featurepoints, der in dieser Ansicht detektiert wird, rekonstruiert werden kann, um Featurepoints, die in verschiedenen Ansichten detektiert wurden, mit den Referenzpunkten zu identifizieren.
Welche Art von Objektansichten für das Lernen verwendet wird, entscheidet über den möglichen Anwendungsbereich des Matcher. Es sollten möglichst "realistische" Ansichten verwendet werden, also solche, die eine möglichst repräsentative Teilmenge der zur Laufzeit auszuwertenden Ansichten darstellen.
Wir führen dazu eine *Viewset-Factory-Klasse* ein, welche ausgehend von einem Referenzbild mit bekannter Pose Ansichten erzeugt (siehe *Abb.19*). Die spezifischen Anwendungsfälle konkretisieren diese Klasse.
Eine Viewset-Factory Φ realisiert formal eine Funktion $\Phi(Ref, k) = (Img_1, \phi_1, ..., Img_k, \phi_k)$, wobei Ref ein Referenzbild mit bekannter Pose darstellt, $Img_1, ..., Img_k$ zufällige Ansichten fester Größe und $\phi_1, ..., \phi_k : \mathbb{R}^2 \rightarrow \mathbb{R}^3$ Transformationen von 2D-Featurepointkoordinaten in 3D-Weltkoordinaten.

4.2.1 Synthetisieren von Ansichten für planare Objekte

Die Projektion eines planaren Objekts im Lochkameramodell kann durch eine 2D-Homographie beschrieben werden [10]. Diese kann lokal durch eine affine Transformation approximiert werden. Im Falle planarer Objekte bietet es sich aus Effizienzgründen also an, ausgehend von einer Frontansicht des zu erkennenden Objekts durch affine Transformation (Streckung, 2D-Rotation, Scherung) neue Ansichten des gesamten Objekts zu synthetisieren, da diese perspektivische Projektionen **lokal** ausreichend realistisch approximieren (siehe *Abb. 14*), wie in [2]

beschrieben. Punktkorrespondenzen mit dem Referenzbild können dann durch einfache Rücktransformation der Punkte aus den synthetischen Ansichten in das Referenzbild ermittelt werden.
Zur Stabilisierung der Erkennung können Rauschen und Störungen zu den synthetischen Ansichten hinzugefügt werden.
Im Rahmen dieser Arbeit werden stets zufällige affine Transformationen $x' = Ax$ verwendet, wobei x/x' die Pixelkoordinaten relativ zum Bildmittelpunkt in der alten/neuen Ansicht sind, $A = R_\theta R_\phi^{-1} S R_\phi$ eine 2D-Transformationsmatrix bestehend aus einer 2D-Rotation um $\theta \in [-\pi, +\pi]$ und einer direktionalen Skalierung (entspricht einer Scherung) $S = diag(s_x, s_y)$ entlang dem Winkel $\phi \in [-\pi, +\pi]$ mit Skalaren $s_x, s_y \in [0.5, 1.5]$. Man erhält neue Ansichten des Referenzbildes also durch Anwendung der Homographie-Abbildung $H = T^{-1} R_\theta R_\phi^{-1} S R_\phi T$, wobei:

$$T = \begin{pmatrix} 1 & 0 & -w/2 \\ 0 & 1 & -h/2 \\ 0 & 0 & 1 \end{pmatrix}, S = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}, R_\phi = \begin{pmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{pmatrix}, R_\theta \text{ wie } R_\phi.$$

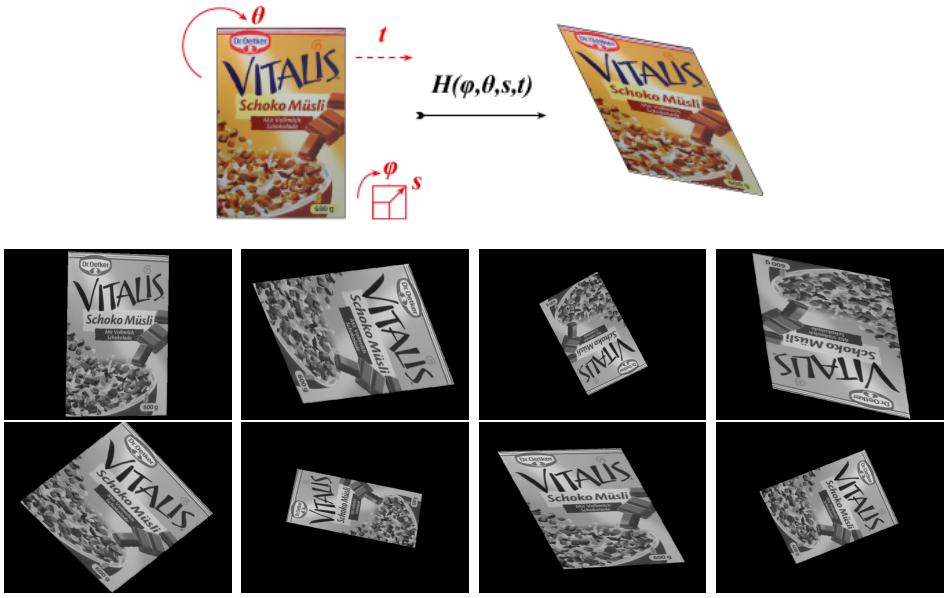


Abbildung 14: Erzeugen von Ansichten durch affine Transformation

4.2.2 Synthetisieren von Ansichten mit 3D-Modellen

Liegt ein 3D-Modell des zu erkennenden Objekts vor, können Ansichten mit Texturemapping-Techniken synthetisiert und evtl. wie oben durch Hinzufügen von Störungen nachbearbeitet werden. Das Objekt erhält hierzu zunächst eine zufällige Pose durch euklidische Transformation mit $H = T R_\gamma R_\beta R_\alpha$, wobei T eine Verschiebung, $R_\alpha, R_\beta, R_\gamma$ zufällige Rotationen um die Hauptachsen mit den Drehwinkeln $\alpha, \beta, \gamma \in [-\pi, +\pi]$ darstellen (siehe *Abb. 15*), und wird anschließend

mit einer festen virtuellen Referenzkamera K gerendert.
Konkret sei eine Referenzkamera K gegeben, welche eine Projektion der 3D-Weltkoordinaten in 2D-Bildkoordinaten darstellt (siehe 2.2), sowie Rotationswinkel α, β, γ um die x-, y- bzw. z-Achse, und eine Verschiebungsvektor (t_x, t_y, t_z). Wir erhalten die Transformationsmatrix für die homogenen Objektkoordinaten dann durch: $A = TR_\gamma R_\beta R_\alpha$, wobei:

$$H = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}, \; R_\alpha = \begin{pmatrix} 1 & 0 & 0 & 00 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \; R_\beta = \begin{pmatrix} \cos\alpha & 0 & \sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \; R_\gamma = \begin{pmatrix} \cos\gamma & -\sin\gamma & 0 & 0 \\ \sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 00 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Wir erhalten die Objektansicht dann durch Projektion mit KH. Offensichtlich stellt $E_0 H^{-1}$ die Kamerapose der Ansicht dar, wobei E_0 die Ausgangspose der Kamera sei.
Prinzipiell lässt sich aus der Kenntnis der Objektgeometrie und der jeweiligen Kamerapose der gerenderten Ansichten die 3D-Position jedes 2D-Bildpunkts ermitteln. Da das verwendete Framework diese Option allerdings nicht ohne weiteres zur Verfügung stellt, rendern wir eine 3D-Sequenz, in welcher das 3D-Modell kontinuierlich von der virtuellen Kamera umfahren wird und verfahren dann wie im Fall vorliegender Videosequenzen, wobei zu beachten ist, dass die Kamerapose für jedes Sequenzbild bekannt ist.

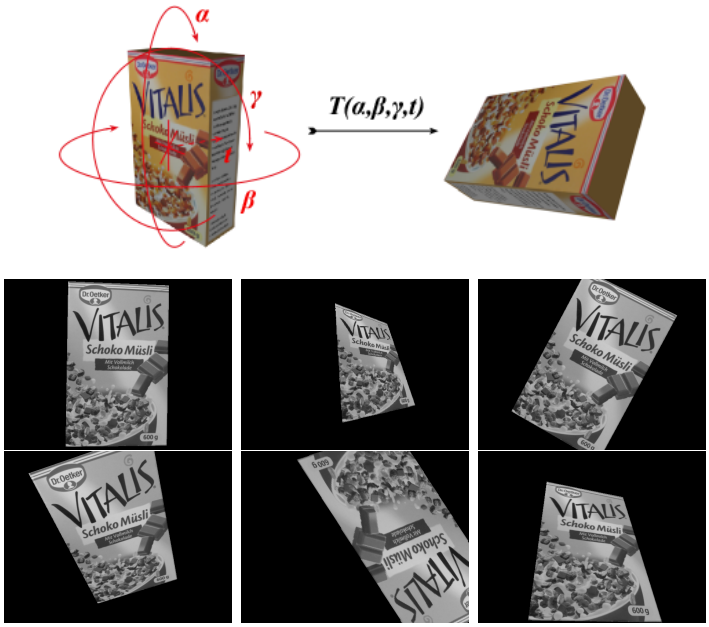


Abbildung 15: Erzeugen von Ansichten mit 3D-Modell

4.2.3 Synthetisieren von Ansichten für Panoramaansichten

Als weiterer Anwendungsfall soll die Poseschätzung in Ansichten eines Szenenpanaromas betrachtet werden. Analog zu den ersten beiden Fällen können auch hier Ansichten durch eine entsprechende Viewset-Factory generiert werden.
Die Panoramaansicht, die hier als Referenz dient, sei mit einer Kamera K_0 mit $C_0 = (0,0,0)^T$ und Fokuslänge f_0 aufgenommen und habe die Größe $w_0 \times h_0$ Pixel. K_0 besitze die Ausgangsorientierung $\phi_0 = 0$, $\vec{v_0} = (0,0,-1)^T$ (siehe *Abb. 16, links*).
Im Rahmen dieser Arbeit werden zufällige Ansichten des Szenenpanoramas mit einer Größe von 640×480 Pixeln verwendet. Die Betrachterkamera K für die generierten Ansichten sei mit $C = (0,0,0)^T$ im Ursprung fixiert und befinde sind ebenfalls in der Standardorientierung (siehe *Abb. 16, rechts*). Wir erhalten zufällige Ansichtskameras K_i durch Drehung von K. Die Raumwinkel von K werden dabei zufällig so gewählt, dass der Ansichtsmittelpunkt im Panoramabild liegt. Wir ermitteln dazu eine zufällige optische Achse $\vec{v} = \overrightarrow{CV}$ mit $V = (v_x, v_y, v_z)$, wobei $v_z = -f_0$ und $v_x, v_y \in [0, w_0 - 1] \times [0, h_0 - 1]$ zufällig gewählt werden. Die Rotationsachse $\vec{w}$ für K erhält man dann durch $\vec{w} = \frac{\vec{v} \times (0,0,1)^T}{\|\vec{v} \times (0,0,1)^T\|}$, den Rotationswinkel θ durch $\theta = \arccos(\frac{v_z}{\|\vec{v}\|})$.

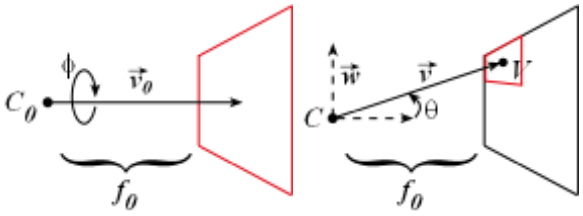


Abbildung 16: Aufnahme des Panoramabildes bzw. einer Ansicht

4.2.4 Verwenden vorhandener Ansichten mittels Tracking

Ein Anwendungsfall, der in [2] und [1] nicht behandelt wird, ist das Training der Entscheidungsbäume auf der Grundlage von Videosequenzen, die das zu lernende Objekt bzw. die zu lernende Szene aus verschiedenen Perspektiven zeigen (siehe *Abb. 18*). In diesem Fall können Ansichten also nicht synthetisiert werden, sondern es werden reale Ansichten verwendet, deren Parameter zunächst a priori unbekannt sind. Die Viewset-Factory liefert also einfach einzelne zufällig ausgewählte Sequenzbilder als Ansichten.
Problematisch ist hier das notwendige Ermitteln von Punktkorrespondenzen bzw. die Extraktion der 3D-Koordinaten der Bildpunkte jedes Sequenzbildes, die zur Identifizierung von Featurepoints aus den Ansichten mit den Referenzpunkten benötigt wird. Im Rahmen der vorliegenden Arbeit sollen vorbereitend Tracking-

Abbildung 17: Erzeugen von Ansichten aus Panoramaszenen

und Tiefenschätzverfahren angewendet werden, um diese Information aus der Videosequenz heraus zu ermitteln. Für das Tracking von Featurepoints wird eine Implementierung des KLT-Trackers (Kanade-Lucas-Tomasi Tracker [9]) verwendet (genaueres hierzu später in *Kap.4.3.4*).

Abbildung 18: Verwenden von Ansichten aus Videosequenzen

4.3 Auswahl von 3D-Referenzpunkten

Als Referenzpunkte sollen 3D-Punkte des Objekts gewählt werden, die möglichst stabil als Featurepoints erkennbar sind, also solche Punkte, die in möglichst vielen verschiedenen Ansichten durch denselben Detektor als Featurepoints erkannt werden. Um solche Referenzpunkte zu ermitteln, wird in einer ersten Phase eine möglichst große Menge von Ansichten des zu erkennenden Objekts verwendet (aus ca. 1000 Ansichten des Objekts). Zunächst werden unabhängig voneinander Featurepoints in jeder einzelnen Ansicht detektiert. Es wird vorausgesetzt, dass die 3D-Weltkoordinaten eines 2D-Featurepoints bestimmt werden können bzw. der korrespondierende Bildpunkt im Referenzbild bestimmt werden kann: Wir benutzen diese Information, um Featurepoints derselben 3D-Punkte zu zählen und diejenigen, die am häufigsten detektiert wurden, als Referenzpunkte auszuwählen.
In denjenigen Szenarien, in welchen wir die Ansichten kontrolliert aus einem Referenzbild mit Referenzkamera K_0 generieren, können wir Punkte beliebig zwischen verschiedenen Ansichten und dem Referenzbild hin- und hertransformieren. In diesen Szenarien gestaltet sich die Auswahl stabiler Referenzpunkte im Allgemeinen wie folgt:

1. Ermittle zunächst mit einem LGD Referenzpunkt-Kandidaten FP_{Ref} in dem Referenzbild Img_{Ref}. Aus diesen werden wie folgt die N stabilsten Punkte gewählt:

2. Erzeuge Ansichten $Img_1, ..., Img_k$ des zu erkennenden Objekts unter Verwendung der Viewset-Transformationen $\Phi_1, ..., \Phi_k$ des Referenzbildes, wobei $k \approx 1000$.

3. Detektiere in jedem Bild Img_i unabhängig voneinander mit dem LGD Featurepoints FP_i.

4. Zähle für jeden Referenzpunkt-Kandidaten $F \in F_{Ref}$ die Bilder Img_i, in welchen F erkannt wurde, also: $score_F := |\{i \leq k | \Phi_i(F) \in FP_i\}|$.
   Für die Identifizierung von Bildpunkten lassen wir eine Toleranz von ± 2 Pixeln zu.

5. Ordne die Featurepoints in FP_{Ref} nach ihren Wiedererkennungswerten $score_F$ und wähle die besten N Punkte als FP^*_{Ref}.

6. Verwende die Kamera K_0 des Referenzbildes, um die zu den Punkten $F \in FP^*_{Ref}$ gehörigen 3D-Punkte M zu bestimmen, und verwende diese als Referenzpunkte $R_1, ..., R_N$.

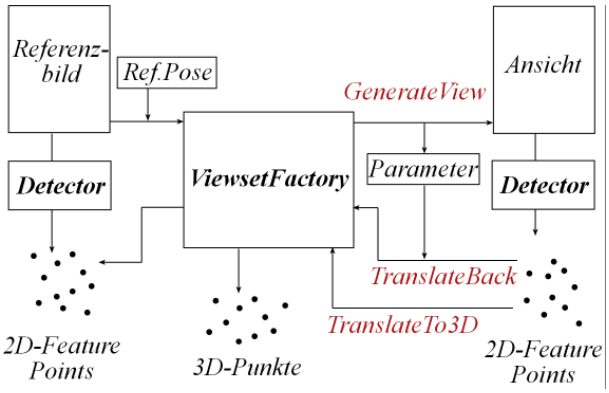


Abbildung 19: Erzeugen von Ansichten durch ViewsetFactory-Klasse

Wir erweitern die zum Erzeugen von Ansichten eingeführte *Viewset-Factory-Klasse* also um die Möglichkeit zur Identifizierung von Featurepoints, die in verschiedenen Ansichten detektiert wurden, durch Rücktransformation, sowie um die Ermittlung der 3D-Koordinaten von 2D-Punkten in den erzeugten Ansichten (siehe *Abb.19*, in der Beschreibung der Viewset-Factory werden englischsprachige Begriffe (z.B. *GenerateView*, *TranslateBack*) verwendet, da sich diese in der Implementierung an dieser Stelle wiederfinden).
Dieses Verfahren unterscheidet sich für die verschiedenen Szenarien konkret nur in der Art der für das Erzeugen von Ansichten verwendeten Transformation und der Bestimmung der 3D-Punkte. Für die einzelnen Anwendungsfälle werden diese wie folgt unterschiedlich konkretisiert.

4.3.1 Referenzpunkte bei planaren Objekten

Bei planaren Objekten werden Transformationen $H_1, ..., H_k$ des Referenzbildes als Ansichts-Transformationen verwendet. Ein Featurepoint F in einem Bild Img_i stimmt mit einem Referenzpunkt-Kandidaten $F_{Ref} \in FP_{Ref}$ überein, wenn gilt: $H_i^{-1}(F) \approx F_{Ref}$.

Zur Umrechnung von 2D- in 3D-Punkte lokalisieren wir das Referenzbild auf einer beliebigen Ebene (z.B. $z = 1$): Soll eine einfache Fläche erkannt werden (etwa die Vorderseite eines Buches, etc.), kann jeder 2D-Punkt $m = (x, y)$ im Referenzbild z.B. als 3D-Punkt $M = (x, y, 1)$ verwendet werden. Besteht das zu erkennende Objekt aus mehreren Flächen (etwa eine Schachtel, ein Buch mit Vorder- und Rückseite), werden für jede Seite unabhängig voneinander mit dem obigen Verfahren Referenzpunkte ermitteln. Die 3D-Koordinaten der jeweiligen Referenzpunkte erhält man durch anschließende Koordinatentransformation unter Kenntnis der Objektgeometrie.

4.3.2 Referenzpunkte aus 3D-Modellen

Für 3D-Objekte, für welche ein 3D-Modell vorliegt, werden die Referenzpunkte prinzipiell auf dieselbe Weise gewählt. Das Viewset wird hier durch Rendern des Modells unter euklidischen Transformationen H_i der Modellkoordinaten erzeugt (siehe *Kap. 4.2.2*). Die Kenntnis von H_i, der virtuellen Kamera K und der Objektgeometrie reicht theoretisch aus, um die zu den jeweils detektierten Featurepoints gehörigen 3D-Koordinaten zu ermitteln und Korrespondenzen zu bestimmen. Im Rahmen dieser Studienarbeit soll auf diesen Fall allerdings aus Zeitgründen nicht weiter eingegangen werden.
Es bietet sich wie bei mehrseitigen planaren Objekten an, mehrere verschiedene Ansichten für die Ermittlung der Referenzpunkt- Kandidaten zu verwenden (etwa: Aufsicht, Vorder-, Rück- und Seitenansichten).

4.3.3 Referenzpunkte aus Panoramaszenen

Vorliegend ist eine Panoramaansicht, die mit einer Kamera K erzeugt wurde. Verschiedene Ansichten des Panoramas werden wie oben beschrieben durch Projektionen $K_1, ..., K_k$ mit zufälligen Parametern synthetisiert. Wir erhalten die 3D-Koordinaten M eines Featurepoints F aus dem Bild Img_i durch Rückprojektion:

1. Berechne zuerst den (normierten) Rückprojektionsstrahl $\vec{w}$ von F bezüglich Kamera K_i: $\vec{w} = K_i^{-1}(F)$.

2. Berechne M als den Schnittpunkt von $\vec{w}$ mit der Bildebene $z = f_0$ der Referenzkamera K_0: $M = C_i + \frac{\vec{w}}{w_z} \cdot f_0$.

Ob ein Featurepoint F in einem Bild Img_i mit einem Referenzpunkt-Kandidaten $F_{Ref} \in FP_{Ref}$ übereinstimmt, lässt sich dann durch Projektion des zu F gehörigen 3D-Punkts M auf die Bildebene der Referenzkamera überprüfen: $K_0(M) \approx F_{Ref}$. Es ist zu beachten, dass wir deutlich mehr Referenzpunkte benötigen, als in den vorigen Fällen, da die einzelnen Ansichten nur kleine Ausschnitte aus dem gesamten Referenzbild darstellen.

4.3.4 Referenzpunkte aus Videosequenzen

Sollen Videosequenzen des Objekts als Trainingsmaterial verwendet werden, ist die Kamerapose jeder Ansicht a priori **nicht** bekannt. Damit ist für jeden 2D-Bildpunkt die entsprechende 3D-Information zunächst **nicht** vorhanden, und wir kennen keine Korrespondenzen zwischen Bildpunkten. In diesem Fall wird zunächst ein (bereits in der Programmbibliothek implementiertes) Verfahren zum Featurepoint-Tracking verwendet, um 3D-Koordinaten von Featurepoints zu ermitteln und Korrespondenzen zwischen Featurepoints in verschiedenen Ansichten herzustellen.
In diesem Kapitel soll nur ein kurzer Überblick über das Verfahren gegeben werden, soweit es für unsere Arbeit notwendig ist. Für eine detaillierte Beschreibung siehe [10] bzw. [9].
Wir ermitteln die stabilsten 3D-Punkte und stellen gleichzeitig die Information zum Klassifizieren von Featurepoints für die folgenden Phasen zur Verfügung:

1. Vorausgesetzt wird eine Bildsequenz $Img_0, ..., Img_k$ mit nur geringen Unterschieden in der Kamerapose aufeinanderfolgender Bilder. Die Kamerakalibierung sowie die Kameraprojektion K_0 des ersten Bildes sei zur Initialisierung des Trackers bekannt.

2. In der Tracking-Phase werden Featurepoints durch die Sequenz hindurch verfolgt [9] und die 3D-Koordinaten identischer Featurepoints durch Triangulation geschätzt:

   a) Detektiere mit dem LGD in Img_0 intiale Featurepoints und tracke diese unter Verwendung eines KLT-Trackers durch die Sequenz hindurch. Wir erhalten sogenannte *(2D-)Trails*, also Sequenzen von 2D-Koordinaten identischer Punkte in aufeinanderfolgenden Bildern.

   b) Füge jeweils bei Verlust von Trails mit dem LGD ausreichend viele neue Featurepoints hinzu.

   c) Rekonstruiere gleichzeitig die 3D-Informationen jedes Trails durch Triangulation.

3. Nach Abschluss der Tracking-Phase können Featurepoints aus verschiedenen Ansichten durch Vergleich mit den vorhandenen Trails identifiziert werden, und ihre 3D- Koordinaten sind bekannt. Damit stehen alle Daten zum Klassifizieren von Featurepoints und zum Ermitteln der 3D-Referenzpunkte wie in den obigen Fällen zur Verfügung, und die Viewset-Factory-Klasse kann für diesen Fall wie oben erweitert werden (siehe *Abb.20*):

   a) Detektiere nach Abschluss des Trackings in jedem Bild Img_i unabhängig voneinander mit dem LGD Featurepoints F_i.

b) Zähle für jeden Trail t die Bilder Img_i, in welchen der entsprechende Punkt m_i^t von t in Img_i durch den LGD detektiert wurde, also: $score_t := |\{i \leq k | m_i^t \in FP_i\}|$.

c) Ordne die Trails nach ihren Wiedererkennungswerten $score_t$ und wähle die 3D-Punkte der besten N Trails als Referenzpunkte aus.

d) Klassifiziere abschließend alle Featurepoints in $FP_0, ..., FP_k$ durch Vergleich mit den Referenztrails, also setze für alle $F \in FP_i$: $label_F := c$, falls $F \approx m_i^{t_c}$, $label_F := -1$ anderenfalls (siehe *Abb. 22*). Die Menge $\bar{\mathcal{V}}$ der gelabelten Featurepoints kann später für das Erzeugen und Trainieren der Bäume verwendet werden.

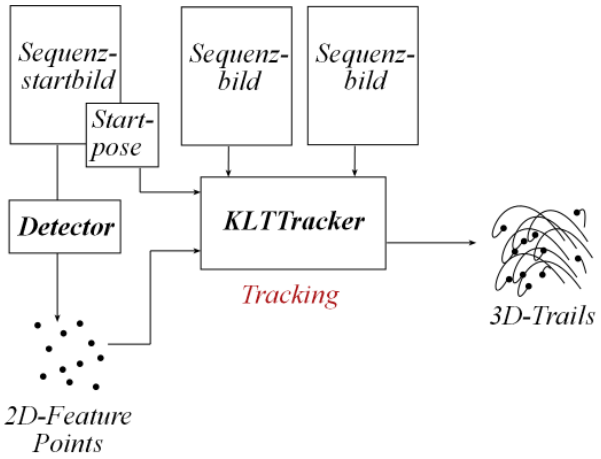


Abbildung 20: Erzeugen von 3D-Trails durch Tracking in Sequenzen

4.3.5 Zusammenstellen klassifizierter Featurepoints

Im vorigen Schritt wurden N 3D-Punkte auf dem zu erkennenden Objekt als Referenzpunkte ausgewählt. Für die Erzeugung und das anschließende Training der Entscheidungsbäume und des Matchers werden im folgenden jeweils klassifizierte Featurepoints F^* aus verschiedenen Ansichten $Img_1, ..., Img_k$ des Objekts benötigt. Die Menge der klassifizierten Featurepoints aus den Ansichten werde als Viewset $\mathcal{V}$ bezeichnet. Ein klassifizierter Featurepoint $F^* = (F, \ell_F)$ - implementiert durch die Struktur *LabelledFeaturePoint* - besitze das Label $\ell_F = c$, wenn F eine Ansicht des 3D-Referenzpunkts R_c darstellt bzw. $\ell_F = -1$, wenn F keinem Referenzpunkt entspricht.
Für die ViewsetFactory-Klassen (a)-(c), die Ansichten synthetisieren, erfolgt das Klassifizieren von Featurepoints durch Rücktransformation in das Referenzbild und Vergleich mit den Referenzpunkten. Für die Identifizierung lassen wir dabei eine Toleranz von ± 2 Pixeln zu.

Für die ViewsetFactory-Klasse (d), die Videosequenzen verwendet, erfolgt das Klassifizieren von Featurepoints durch Vergleich mit den Koordinaten der Referenztrails, die in der vorherigen Trackingphase ermittelt wurden, in der entsprechenden Ansicht.
Wir erhalten ein Viewset $\mathcal{V}$ also durch Detektieren von Featurepoints in verschiedenen generierten Ansichten bzw. Sequenzbildern $Img_1, ..., Img_k$ und anschließendes Klassifizieren der Featurepoints in $FP_{Img_1}, ..., FP_{Img_k}$ durch die entsprechende Viewset-Factory-Klasse (siehe *Abb.21* und *Abb.22*). Im folgenden sollen gemischte Viewsets, die auch Featurepoints F mit $\ell_F = -1$ enthalten, seien explizit als $\bar{\mathcal{V}}$ bezeichnet, reine Viewsets von Referenzpunkten dagegen als $\mathcal{V} = \bigcup_{c \in \mathcal{C}} \mathcal{V}_c$.
Um unseren Test anwenden zu können, wird an die klassifizierten Featurepoints (F, ℓ_F) des erzeugten Viewsets jeweils das zugehörige Patch p_F^{Img}, also ein quadratischer Ausschnitt fester Größe p_{size} aus der Umgebung des Punktes, als Beschreibung angehängt. Wir verwenden $p_{size} = 32$.
Gemischte Viewsets, die auch Featurepoints F mit $\ell_F = -1$ enthalten, seien explizit als $\bar{\mathcal{V}}$ bezeichnet.

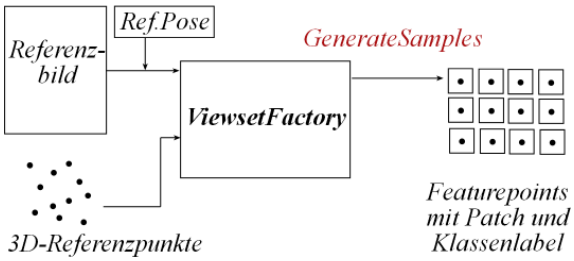


Abbildung 21: Erzeugen klassifizierter Featurepoints durch ViewsetFactory-Klasse

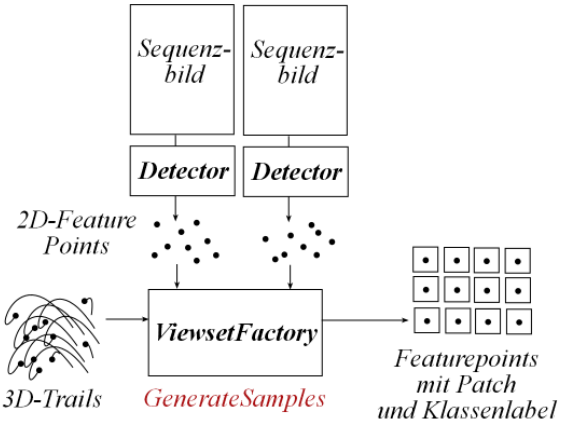


Abbildung 22: Erzeugen klassifizierter Featurepoints durch Tracking

4.4 Konstruktion der Entscheidungsbäume

Im folgenden sollen die Strukturen zur Realisierung von Entscheidungsbäumen entwickelt und die Konstruktion und das Trainieren das Matchers beschrieben werden, wobei wir die Beschreibungen in [2] und [1] umsetzen. Die besondere Zielsetzung der Implementierung soll hierbei die *Erweiterbarkeit* der Entscheidungsbäume durch beliebige Testmethoden sein.

Spezifikation der Entscheidungsbäume: Die Klassenstruktur orientiert sich streng an der formalen Beschreibung in Kapitel 3.2. Ein Baum ist mit einer Menge von Referenzpunkten $\mathcal{R}$ bzw. $N = |\mathcal{R}|$ Klassen sowie mit einem Test ν assoziiert. Innere Knoten des Baums werden mit beliebigen Testparametern assoziiert, während Blattknoten mit Klassenverteilungen assoziiert sind, die in der Konstruktionsphase generiert werden (siehe auch Implentierungshinweise im Anhang).

Da unsere Spezifikation vorsieht, Entscheidungsbäumknoten mit beliebigen Testparametern zu verknüpfen, verwenden wir Templates zur Beschreibung der Klassen, die einen prinzipiell beliebigen Parametertyp *ParamType* verwenden. Für den oben beschriebenen Pixelvergleichstest erhalten wir damit ein Klassendiagramm wie in *Abb. 23*.

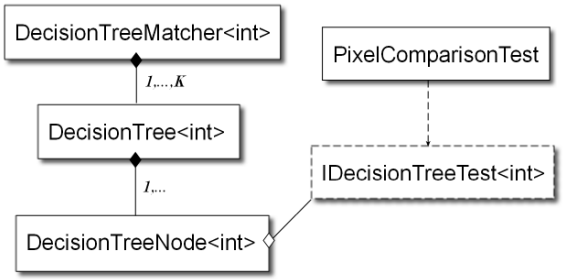


Abbildung 23: Klassendiagramm der Entscheidungsbäume für Parametertyp `int`

Konstruktion und Training: Liegen klassifizierte Featurepoints $\mathcal{V} = \{(F_0, \ell_0), ..., (F_s, \ell_s)\}$ aus genügend Ansichten als Konstruktionsviewset vor, können nun Entscheidungsbäume konstruiert werden, welche das Konstruktionsviewset in möglichst eindeutige Teilmengen partitionieren. Anschließend können die Klassenverteilungen in den Blättern durch ein weiteres, gröseres Viewset a posteriori mit geringem Aufwand ermittelt werden. In einem letzten Schritt wird die Fehlerrate des Matchers, der den Mittelwert aller Entscheidungsbaumes zur Klassifizierung verwendet, durch eine weitere Trainingsphase reduziert.

4.4.1 Konstruktion einzelner Bäume

Zur Konstruktion einzelner Entscheidungsbäume gemäß [2] werden die Testparameter in den inneren Knoten so fixiert, dass eine Menge repräsentativer Ansichten

der Referenzpunkte (*Konstruktions-Viewset*) bestimmter Größe nach einem festgelegten Gütekriterium optimal partitioniert, also "unterschieden" wird.
Zur Konstruktion einzelner Entscheidungsbäume setzen wir ein Viewset $\mathcal{V}_{create} = \{(F_0, \ell_0), ..., (F_s, \ell_s)\}$ von klassifizierten Featurepoints F_i mit $\ell_i \neq -1$ aus ca. $k_{create} = 100$ Ansichten voraus. Beachte: $s = |\mathcal{V}_{create}| \leq 100N$. Gegeben sei außerdem ein Test ν für den Entscheidungsbaum. Wir erzeugen T von der Wurzel ausgehend und wählen die Parameter für ν so, dass die Trainingsmenge möglichst eindeutig in Teilmengen partitioniert wird. Für jeden Nachfolger der Wurzel verfahren wir rekursiv mit der entsprechenden Teilmenge der Trainingsmenge genauso, bis die Konstruktionsmenge in einem Knoten ausreichend geordnet oder klein genug ist, oder eine vorgegebene Maximaltiefe d_{max} erreicht wurde.

Auswahl der besten Testparameter: Als Gütekriterium für die zu wählenden Testparameter wird in [2] der *"erwartete Informationsgewinn"* (*"expected gain in information"*) vorgeschlagen, welcher im Wesentlichen die "Geordnetheit" der Teilmengen, die der Test mit diesen Parametern erzeugt, beschreibt. Als Maß für die "Geordnetheit" einer Menge $\mathcal{S}$ von klassifizierten Featurepoints dient die *negative Entropie* $-E(\mathcal{S})$ bezüglich der Klassen der Featurepoints:

$$\text{Neg. Entropie } -E(\mathcal{S}) := \sum_{c=1}^{N} prop_c(\mathcal{S}) \log_2(prop_c(\mathcal{S})),$$
wobei $prop_c(\mathcal{S}) := \frac{1}{|\mathcal{S}|}|\{F \in \mathcal{S} | \ell_F = c\}|$ der Anteil von Featurepoints der Klasse c in $\mathcal{S}$ sei.

Es gilt: $-E(\mathcal{S})$ ist minimal, wenn in $\mathcal{S}$ alle N Klassen genau gleich oft vorkommen, nämlich: $-E(\mathcal{S}) = \log_2(\frac{1}{N+1}) < 0$ (schlechtester Fall), sowie: $-E(\mathcal{S})$ ist maximal, wenn in $\mathcal{S}$ nur eine einzige Klasse auftritt, nämlich: $-E(\mathcal{S}) = 0$ (bester Fall).
Der erwartete Informationsgewinn $\Delta E(\nu, \mathcal{S})$ des Tests ν ergibt sich als der gewichtete Mittelwert der negativen Entropien der durch ν erzeugten Teilmengen:

$$\text{Erwarteter Informationsgewinn } \Delta E(\nu, \mathcal{S}) := -\frac{1}{|\mathcal{S}|} \sum_{i=1}^{k_\nu} E(\mathcal{S}_i),$$

wobei der Test ν die Trainingsmenge $\mathcal{S}$ in k_ν Teilmengen $\mathcal{S}_1, ..., \mathcal{S}_{k_\nu}$ partitioniert. Für eine Teilmenge $\mathcal{S}$ des Viewsets in einem Blatt v werden zunächst verschiedene zufällige Testparameter gewählt und $\Delta E(\nu, \mathcal{S})$ jeweils ausgewertet. Es werden dann diejenigen Parameter für ν gewählt, die $\Delta E(\nu, \mathcal{S})$ maximieren. Man kann also davon ausgehen, dass die Featurepointmengen, die in den Blättern resultieren, größtenteils nach Klassen geordnet sind. Wir zählen die Featurepoints in den Blättern und erhalten daraus einen provisorischen Klassenverteilungsvektor d jedes Blattes.

Algorithmus zur Konstruktion von Entscheidungsbäumen: Gegeben sei das Viewset $\mathcal{V}_{create}$, ein Test mit t Ergebnisklassen, sowie die Wurzel ρ des Baums. Starte mit $CreateTreeRecursive(\mathcal{V}_{create}, \rho, 0)$.

```
CreateTreeRecursive(S, v, d):
  // create leaf and stop if set S is good enough or max depth is reached
  E = negative entropy for set S
  if d == maxDepth or E > limitE then
    // create initial class distribution for this leaf
    for every featurepoint F in S do
      v.totalNumber++
      v.numberOfClass[F.label]++
    end for
    break
  end if

  // select best test parameters otherwise
  bestdE = -Infinity
  if d > 0 then k = 100*d else k = 10
  for i = 1 to k do
    select random test parameters Param for test
    // partition S into subsets using random test parameters
    create empty subsets S[1], ..., S[t]
    for every featurepoint F in S do
      result = test F with parameters P
      add F to subset S[result]
    end for
    dE = expected gain in information for subsets S[1], ..., S[t]
    if dE > bestdE then bestParam = Param, bestdE = dE
  end for
  // set best test parameters for this node
  v.testParam = bestParam
  // partition set S into subsets using best test parameters
  create empty subsets S[1], ..., S[t]
  for every featurepoint F in S do
    result = test F with parameters v.testParam
    add F to subset S[result]
  end for
  // build children nodes recursively with each subset
  for j = 1 to t do
    CreateTreeRecursive(S[j], v.childNode[j], d+1)
  end for
```

Analyse der Konstruktion von Entscheidungsbäumen: Die Konstruktion ist umso aufwendiger, je größer das Konstruktionsviewset ist und je mehr Tests in jedem einzelnen Knoten geprüft werden müssen. Um Speicher und Rechenzeit zu sparen, darf das Viewset nicht zu groß gewählt werden, wir müssen also eine größere Suboptimalität des konstruierten Baums in Kauf nehmen. Verwenden wir ein Viewset aller Referenzpunkte aus $\mathcal{V}_{create}$ aus k_{create} Ansichten, werden insgesamt in jeder Ebene des Baums max. $k_{create}N$ Featurepoints betrachtet, da $|\mathcal{V}_{create}| \leq k_{create}|\mathcal{R}| = k_{create}N$. Prüfen wir in der Tiefe d n_d zufällige Testparameter, so benötigen wir insgesamt im pessimalen Fall $\chi = \sum_{i=0}^{d_max} n_d k_{create} N$ Testauswertungen. Wir verwenden $n_0 = 10$ und $n_d = 100d$, $d_max = 10$, sowie $N \approx 200$ und erhalten $\chi \approx 1.000.000 \cdot k_{create}$. Da eine Testauswertung ca. $1ns$ benötigt, ist die Anzahl der Ansichten für das Konstruktionsviewset auf den Bereich $k_{create} \approx 100$ beschränkt, um eine Laufzeit in der Größenordnung 100 sec nicht wesentlich zu überschreiten.

4.4.2 Ermitteln der Klassenverteilungen

Nachdem die Baumstruktur fixiert ist, wird der Entscheidungsbaum verwendet, um eine größere repräsentative Menge von Featurepoints zu klassifizieren und die Klassenverteilungen in den Blättern anhand dieser zu präzisieren. Hierzu wird ein weiteres, größeres Viewset $\mathcal{V}_{distr}$ von Referenzpunkten aus $k_{distr} = 1000$ Ansichten verwendet. Zu jedem einzelnen Featurepoint aus $\mathcal{V}_{distr}$ wird das resultierende Blatt ermittelt und der dazugehörige Klassenverteilungsvektor entsprechend angepasst. Es ist damit zu rechnen, dass wir auf diese Weise realistischere Klassenverteilungen erhalten, allerdings zu Lasten von deren Eindeutigkeit.

Analyse des Ermitteln der Klassenverteilungen: Die Schätzung der Klassenverteilungen benötigt - im Gegensatz zur Baumkonstruktion - mit einem Viewset $\mathcal{V}_{distr}$ aus k_{distr} Ansichten der Größe $\mathcal{V}_{distr} \leq k_{distr}N$ lediglich $|\mathcal{V}_{distr}|d_{max}$ Testauswertungen (für jeden Featurepoint in $\mathcal{V}_{distr}$ max. d_{max} Auswertungen), also mit obigen Werten max. $k_{distr} \cdot 2ms$. Da die Auswertung eines Featurepoints unabhängig von der anderer Featurepoints ist, können wir ebenfalls nacheinander jeweils kleinere Viewsets zum Updaten der Klassenverteilung verwenden, wodurch der benötigte Speicher beliebig klein gehalten wird. Wir können k_{distr} daher problemlos im Bereich $1000 - 10.000$ wählen, um eine möglichst repräsentative Menge aller Referenzpunktansichten zu berücksichtigen.

Algorithmus zum Ermitteln der Klassenverteilungen: Gegeben sei das Viewset $\mathcal{V}_{distr}$, ein Test mit t Ergebnisklassen. Wir ermitteln die Klassenverteilungen mit $UpdateClassDistributions(\mathcal{V}_{distr})$ oder mit aufeinanderfolgenden Aufrufen

$UpdateClassDistributions(\mathcal{V}^1_{distr})$, ..., $UpdateClassDistributions(\mathcal{V}^k_{distr})$, wobei $\mathcal{V}^1_{distr}, ..., \mathcal{V}^k_{distr}$ eine beliebige Zerlegung des Viewsets $\mathcal{V}_{distr}$ sei.

```
UpdateClassDistributions(S):
  for every featurepoint F in S do
    // find leaf for featurepoint F
    v = root
    while v is no leaf do
      result = test F with parameters v.testParam
      v = v.childNode[result]
    end while
    v.totalNumber++
    v.numberOfClass[F.label]++
  end for
```

4.4.3 Konstruktion des Matchers

Wir können die Qualität des Matchings erhöhen, indem mehrere, suboptimale Entscheidungs bäume verwendet werden, um das Matching durchzuführen.
Wir erhalten also einen Featurepoint-Matcher $Y_{\mathcal{R}}$ aus K einzelnen Entscheidungsbäumen $T_1, ..., T_K$ durch Generieren aller $T_1, ..., T_K$ unabhängig voneinander. Die Klassifizierung eines Featurepoints geschieht durch Mittelwertbildung der Ergebnisvektoren aller Bäume wie in 3.2 formal erläutert.
Obwohl wir prinzipiell zulassen, dass jeder Entscheidungsbaum einen eigenen Test verwendet, verwenden wir im Rahmen der Studienarbeit einen globalen Test für alle Entscheidungsbäume eines Matchers.

Analyse der Konstruktion des Matchers: Die obere Schranke für die Laufzeit der Konstruktion eines Matchers mit K Entscheidungsbäumen (mit identischen Parametern) ergibt sich durch $K \cdot time_{Tree}$, wobei $time_{Tree}$ die Laufzeitschranke der Konstruktion eines einzelnen Baums ist.
Der zusätzliche Speicherbedarf für die Konstruktion ist identisch mit dem zusätzlichen Speicherbedarf für die Konstruktion eines einzelnen Baums.

Algorithmus zum Klassifizieren von Featurepoints: Gegeben seien Bäume $T_1, ..., T_K$ mit einem gemeinsamen Test ν, sowie ein Featurepoint F.

```
ClassifyFeaturepoint(F):
  // calculate class distribution d as mean result of all trees
  d = vector of size N
  for i = 1 to K do
    // find leaf in i-th tree for featurepoint F
    v = root of i-th tree
```

```
  while v is no leaf do
    result = test F with parameters v.testParam
    v = v.childNode[result]
  end while
  for c = 1 to N do
    d[c] += (v.numberOfClass[c] / v.totalNumber) / K
  end for
end for
// find class with highest probability
bestD = 0, class = -1
for c = 1 to N do
  if d[c] > bestD then bestD = d[c], class = c
end for
// use identification threshold to decide if F has been identified
if bestD > D[c] then return class else return -1
```

4.4.4 Weiteres Training des Matchers

In einem letzten Schritt soll die Identifikationsgüte des Matchers, den wir aus einzelnen Entscheidungsbäumen konstruieren, durch weiteres Training verbessert werden. Insbesondere soll der Matcher trainiert werden, beliebige Featurepoints von Referenzpunkten zu unterscheiden. Aus diesem Grund benötigen wird hier im Gegensatz zu den vorigen Phasen auch Ansichten von Nicht-Referenzpunkten zum Lernen.
Zur Bestimmung der Identifikationsschwellenwerte ist für jede Klasse $c \in \mathcal{C} \setminus \{-1\}$ das folgende Optimierungsproblem zu lösen:
Minimiere $D_c \in [0, 1]$ so, dass für alle $F \in \bar{\mathcal{V}}_{train}$ gilt:

$$(*)\ \mathbb{P}[\ell_F = c | Y_{\mathcal{C}}(F) = c, d(F)_c > D_c] > 0.9.$$

Das Lösen dieser Minimierungsaufgabe ist nicht trivial. Wie das Problem in der Implementierung des Verfahrens durch Lepetit und Fua [4], die uns als Referenz dient, gelöst wird, wird in [2] und [1] nicht beschrieben. Im Rahmen dieser Studienarbeit sollen die Schwellenwerte der Einfachheit halber adaptiv ermitteln werden: Wir erhalten die Identifikationsschwellenwerte $D_1, ..., D_N$ für jede Klasse $c \in \mathcal{C} \setminus \{-1\}$ durch fortschreitende Inkrementierung mit einer festen Schrittweite ΔD, bis die Bedingung $(*)$ erfüllt ist. Erfüllt kein D_c die Bedingung, wählen wir D_c so, dass die rechte Seite von $(*)$ zumindest maximiert wird.
Insbesondere ist hier der Speicherbedarf kritisch, da im Gegensatz zu den vorigen Schritten auch Nicht-Referenzpunkte in das Trainingsviewset mit eingehen (siehe **Analyse**).

Algorithmus zum Bestimmen der Identifikationsschwellwerte: Gegeben sei das **gemischte** Viewset $\bar{\mathcal{V}}_{train} = \mathcal{V}_1 \cup ... \cup \mathcal{V}_N \cup \mathcal{V}_{-1}$, sowie Bäume $T_1, ..., T_K$ mit ei-

nem gemeinsamen Test ν. Wir ermitteln die Schwellwerte $D_1, ..., D_N$ unabhängig voneinander durch das oben beschriebende Verfahren mit $EstimateIdentificationThresholds(\bar{\mathcal{V}}_{train})$

```
EstimateIdentificationThresholds(S):
  // separate featurepoints from S into subsets according
  // to their classification results
  create empty subset S[c] for each class c
  for each F in S do
    result = ClassifyFeaturepoint(F)
    add F to S[result]
  end for

  // estimate threshold for each class c
  for c = 1 to N do
    bestInlier = 0, bestD = 0
    D[c] = 0
    while D[c] < 1
      countIdenitified = 0
      countInlier = 0
      for each F in S[c] do
        result = ClassifyFeaturepoint(F)
        if result == c then
          countIdentified++
          if F.label == c then countInlier++
        else
          remove F from S[c]
        end if
      end for
      percentInlier = countInlier / countIdentified
      if percentInlier > bestInlier then
        bestInlier = percentInlier
        bestD = D[c]
        if bestInlier > 0.9 then break
      end if
      D[c] = D[c] + stepwidth
    end while
    D[c] = bestD
  end for
```

Analyse der Bestimmung der Identifikationsschwellwerte: Das Verfahren benötigt für jede Klasse im pessimalen Fall $1/\Delta D$ Iterationen zum Terminieren. In allen Iterationen werden ingesamt max. $|\bar{\mathcal{V}}_{train}|/\Delta D$ Klassifizierungen durchgeführt, also max. $|\bar{\mathcal{V}}_{train}|d_max/\Delta D$ Testauswertungen. Es gilt $|\bar{\mathcal{V}}_{train}| \leq k_{train}N_{max}$,

wobei N_{max} die maximale Anzahl an Featurepoints ist, die der LGD zurückgibt. Mit $k_{train} = 500, N_{max} = 2000, d_{max} = 10$ und $\Delta D = 0.005$ erhalten wir also eine pessimale Laufzeit von $2000\,\text{sec} \approx 30\,\text{min}$.
Der Speicherbedarf für das Viewset ist mit den obigen Werten $|\bar{\mathcal{V}}_{train}|mem_F$, wobei $mem_F \approx 1$ KByte den Speicherbedarf für einen Featurepoint samt Deskriptor (hier: ein 32×32-Patch) angibt, im pessimalen Fall also $k_{train}N_{max}mem_F \approx 1$ GByte.

4.5 Objekterkennung zur Laufzeit

Abschließend kann aus den 2D-3D-Korrespondenzen, die der Matcher liefert, die Posetransformationsmatrix E_{Input} der Kamera K_{Input} des Eingabebildes geschätzt werden [8], wobei zu berücksichtigen ist, dass fehlerhafte Korrespondenzen vorliegen können.
Wir verwenden eine bereits vorhandene Implementierung des POSIT-Algorithmus [5] in Kombination mit dem RANSAC-Verfahren [6] zur Poseschätzung.

4.5.1 Poseschätzung durch den POSIT-Algorithmus

Der POSIT-Algorithmus [5] wird verwendet, um aus 6 gegebenen 2D-3D- Korrespondenzen, die der Matcher liefert, die Pose des Eingabebildes zu schätzen. Durch Kombination mit dem RANSAC-Verfahren [6] erhalten wir eine hinreichende Robustheit der Schätzung, obwohl der Matcher zum Teil falsche Korrespondenzen zurückgibt.
Nach dem Matching sind 2D-3D-Punktpaare $(m_1, M_1), ..., (m_n, M_n)$ bekannt, wobei $m_1, ..., m_n$ die 2D-Bildkoordinaten der im Eingabebild $Input$ detektierten Featurepoints, die durch $Y_{\mathcal{R}}$ identifiziert wurden, sind, sowie $M_1, ..., M_n$ die 3D-Koordinaten der jeweils durch Y zugeordneten Referenzpunkte. Ein gewisser Anteil der Korrespondenzen ist fehlerhaft, etwa durch Ähnlichkeiten von Hintergrundpunkten mit gelernten Referenzpunkten.

- RANSAC wählt jeweils zufällig 6 Punktpaare aus und berechnet durch Anwendung des POSIT-Algorithmus eine Posematrix E aus diesen.
- Anschließend wird die Pose auf die übrigen Punktpaare angewendet und die Inlierrate aller Punkte unter E berechnet.
- Ist die Pose zu genügend Punktpaaren kompatibel (min. 50% Inlier), wird E zurückgegeben, ansonsten wird das Verfahren mit 6 anderen, zufällig ausgewählten Punktpaaren wiederholt.
- Wurde nach einer maximalen Anzahl von Iterationen keine passende Pose gefunden, terminiert der Algorithmus ohne Ergebnis, die Poseschätzung ist in diesem Fall fehlgeschlagen.

5 Ergebnisse und Auswertung

Abschließend sollen die Ergebnisse unsere Testfälle mit den implementierten Methoden vorgestellt und interpretiert werden. Wir verwenden als Testbild, wenn nicht anders erwähnt, ein Graustufenbild der Größe 640×480. Alle Rechenoperationen wurden auf einem ??Angaben über den Rechner?? ausgeführt. Für die Offline-Phase wurde dabei die Debug-Version der BIAS/MIP-Bibliothek verwendet, während wir für die zeitkritischen Anwendungen die Release-Version verwenden.

5.1 Ergebnisse für den Detektor

Laufzeit: Zunächst testen wir die Laufzeit für das Detektieren von max. $N_{max} = 2000$ Featurepoints mit dem LaplacianGreedy-Detektor abhängig von der Grösse des Bildes. Da die Laufzeit des Detektors offensichtlich von der potentiellen Anzahl detektierter Featurepoints und damit vom Texturiertheitsgrad des Eingabebildes abhängt, benötigen wir für jede Bildgröße Testbilder mit ähnlichem Texturiertheitsgrad, um vergleichbare Ergebnisse zu erhalten. Aus diesem Grund verwenden wir Mosaike desselben Bildes zum Testen (siehe *Abb. 25*).

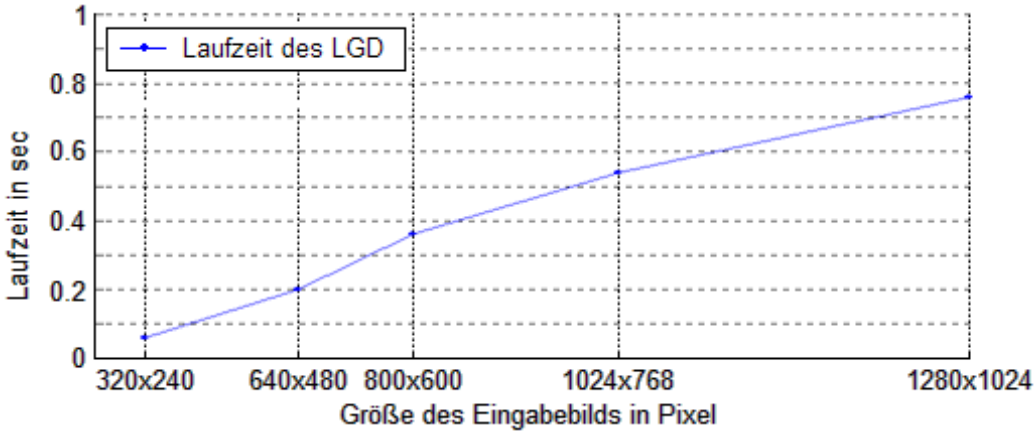


Abbildung 24: Laufzeit des LaplacianGreedy-Detektor

Die Laufzeit des Detektors (siehe *Abb. 24*) verhält sich (bei konstantem Texturiertheitsgrad) etwa linear zur Größe des Bildes, wobei wir hier einen Wert von ca. $0.65 ns/Pixel$ erhalten. Um einen Einsatz in Echtzeit zu ermöglichen besteht hier noch Optimierungsbedarf, wobei etwa auf eine hardwarenähere Implementierung des Detektionsalgorithmus zurückgegriffen werden kann.

Abbildung 25: Testbilder für den LaplacianGreedy-Detektor

5.2 Ergebnisse für das Erzeugen der Bäume

Als nächstes prüfen wir den Aufwand für das Erzeugen der Lerndaten und das Erzeugen und Trainieren der Entscheidungsbäume. Während die Laufzeiten hier unkritisch sind, da das Erzeugen offline geschieht, stehen hier Speicheraspekte im Vordergrund.

Laufzeitergebnisse für das Erzeugen von Viewsets: Die Laufzeit für das Erzeugen von (gemischten) Viewsets ergibt sich pro Ansicht im wesentlichen aus der Laufzeit des Erzeugens der Ansicht, des Detektors und des Klassifizierens, sowie aus der max. Anzahl zu detektierender Featurepoints.
Die durchschnittlich gemessenen Zeiten für das Erzeugen von (gemischten) Viewsets aus jeweils k Ansichten der Größe 640×480 Pixel mit jeweils max. 2000 Featurepoints pro Ansicht durch affine Verzerrungen eines Referenzbilds der Größe 640×480 (Anwendungsfall (a)), perspektivische Projektion eines Referenzbilds der Größe ca. 3600×2400 (Anwendungsfall (c)) bzw. aufgrund von Videosequenzen der Bildgröße 640×480 mittels Tracking (Anwendungsfall (d)) sind wie folgt:

Anwendungsfall	Zeit für $k = 100$	Zeit/Ansicht
Erzeugen durch Homographien	80 sec	0.8 sec/Ansicht
- davon Erzeugen der Ansichten	50 sec	0.5 sec/Ansicht
Erzeugen durch Projektion	4.9 min	2.9 sec/Ansicht
- davon Erzeugen der Ansichten	4.4 min	2.6 sec/Ansicht
Erzeugen durch Tracking	30 sec	0.3 sec/Ansicht
- davon Erzeugen der Ansichten	< 10 sec	< 0.1 sec/Ansicht
Initialisierung beim Tracking	3.2 min	1.9 sec/Ansicht

Erläuterung:

- Bei dem Ermitteln von Viewsets aus Videosequenzen mittels Tracking fällt einmalig die Zeit für das Initialisieren der Viewset-Factory (i.e. das Erzeugen der Featurepoint-Trails durch Tracking) zusätzlich an. Das Erzeugen

von zufälligen Ansichten erfolgt hier quasi ohne Zeitaufwand durch zufällige Auswahl einzelner Sequenzbilder.
Vergleichbares lässt sich für den Fall synthetisierter Ansichten erreichen, wenn Ansichten einmalig generiert und für die spätere Verwendung mit Angaben über die zugehörigen Posen bzw. Transformationen gespeichert werden.

- Die Laufzeit für das Erzeugen gemischter Viewsets ist für synthetisierte Ansichten unabhängig von der Anzahl der Klassen, da wir eine Implementierung gewählt haben, welche die Klassifizierung einzelner Featurepoints in den Lernansichten in konstanter Laufzeit erlaubt. Für Ansichten aus Videosequenzen müssen Featurepoints mit allen Referenztrails verglichen werden, das Klassifizieren in den Lernansichten ist also linear abhängig von N. Für die Tests wurde $N = 200$ gewählt.

Speicherbedarf für Viewsets: Der Speicherbedarf für ein (gemischtes) Viewset $\mathcal{V}$ aus k Ansichten ergibt sich aus der Größe eines Featurepoints mit Deskriptor mem_F - in unserem Fall ein Patch der Größe 32×32 - und der maximalen Anzahl N_{max} an Featurepoints, die in einem Bild detektiert werden. Wir haben $mem_F \approx 1$ kByte und damit für $N_{max} = 2000$ einen pessimalen Speicherbedarf von $kN_{max}mem_F = 2k$ MByte für $\mathcal{V}$. Für $k = 1000$ erhalten wir damit 2 GByte im pessimalen Fall, wobei diese Größenordnung während der Testphasen tatsächlich erreicht wurde und deutlich über den üblichen Kapazitäten liegt! Wir beschränken die Größe des Trainingsviewsets $\mathcal{V}_{train}$ daher auf max. $k_{train} = 500$ Ansichten.

Laufzeitergebnisse für das Erzeugen der Bäume: Die Laufzeit für das Erzeugen und Lernen der Bäume ergibt sich aus der Größe der verwendeten Viewsets, der maximalen Baumtiefe und der Laufzeit des verwendeten Tests - unabhängig von der anwendungsfallspezifischen Viewset-Factory.
Als durchschnittlich gemessene Zeiten für das Erzeugen und Lernen von 20 Bäumen der maximalen Tiefe 10 und das Trainieren des Matchers mit $k_{create} = 100$, $k_{distr} = 1000$, $k_{train} = 500$ und $N = 200$ Klassen erhalten wir unabhängig vom betrachteten Anwendungsfall:

	Durchschnittliche Zeit	Erwartete pessimale Zeit
Erzeugen einzelner Bäume	< 1 min	1.5 min
Klassenverteilung/Baum	< 10 sec	< 10 sec
Training des Matchers	< 15 min	30 min
Dauer gesamt für 20 Bäume	≈ 30 min	67 min

Die erwarteten pessimalen Zeiten ergeben sich aus den theoretischen Abschätzungen der Laufzeit in 4.4.

5.3 Ergebnisse für das Matching:

Wir testen den Matcher zum einen für die Objekterkennung einer Müslischachtel abhängig von der Art der für das Lernen verwendeten Ansichten, wobei wir folgende Lernansichten unterscheiden:

1. Lernansichten durch affine Transformationen erzeugt.
2. Verwendung einer aus einem 3D-Modell gerenderten Videosequenz zum Lernen.
3. Verwendung einer echten Videosequenz zum Lernen.

Davon unabhängig testen wir den Matcher für die Poseerkennung in Panoramaansichten.

5.3.1 Ergebnisse für die Objekterkennung:

Wir testen das Matching für die Objekterkennung der Schachtel jeweils mit zwei Eingabebildsequenzen. In der ersten Sequenz befindet sich das Referenzobjekt vor einem weitestgehend neutralen Hintergrund (*einfache Sequenz*, siehe *Abb. 26, links*), in der zweiten befindet es sich in einem komplexen Szenenaufbau (*komplexe Sequenz*, siehe *Abb. 26, rechts*).

Abbildung 26: Testsequenzen für das Matching

Das Matching für die Panoramaszene testen wir in einer Sequenz von zufällig generierten Panoramaansichten. Wir verwenden jeweils $N = 200$ Referenzklassen, $K = 20$ Bäume der maximalen Tiefe 10 und detektieren max. 2000 Featurepoints in jedem Eingabebild. Für das Erzeugen der Bäume, Ermitteln der Klassenverteilungen und Training des Matchers verwenden wir je $k_{create} = 100, k_{distr} = 1000$ und $k_{train} = 500$ Objektansichten.
Für die Erzeugung von Ansichten der Schachtel verwenden wir zum einen affine Verzerrungen (Anwendungsfall (a)), zum anderen eine künstlich generierte

Videosequenz eines 3D-Modells der Schachtel mit Tracking sowie eine echte Videosequenz mit Tracking (beides Anwendungsfall (d)).
Wir erhalten für die Erkennung der Schachtel bei Erzeugen der Viewsets durch affine Verzerrungen (a) eine durchschnittliche Inlierrate von 85% in der einfachen Sequenz bzw. 70% in der komplexen Sequenz (siehe *Abb. 27*). Der höhere Anteil von Outliern in der zweiten Sequenz kommt dabei durch die größere Anzahl von Featurepoints auf Hintergrundobjekten zustande, die den Referenzpunkten zum Teil ähnlich sehen.
Für die Erkennung der Schachtel aufgrund der gerenderten Videosequenz (d) ergibt sich eine durchschnittliche Inlierrate von 80% in der einfachen Sequenz bzw. 66% in der komplexen Sequenz. Man beachte, dass die Trainingsequenz lediglich ca. 400 Bilder umfasst, die sich zum Teil nur geringfügig unterscheiden. Hier ist festzustellen, dass der Spielraum der Erkennung eingeschränkter ist als für den oben beschriebenen Fall; sobald das Objekt eine Pose einnimmt, welche im Lernvideo nicht auftritt, gelingt die Erkennung deutlich schlechter.
Für die Erkennung der Schachtel aufgrund der realen Videosequenz (d) erhalten wir eine durchschnittliche Inlierrate von 90% in der einfachen Sequenz bzw. 33% in der komplexen Sequenz. Es ist zu berücksichtigen, dass wir hier die einfache Sequenz zum Lernen verwendet haben. Wir stellen fest, dass der Spielraum der Erkennung noch deutlich eingeschränkter ist als im zuvor betrachteten Fall.
Die durchschnittliche Laufzeit eines Matching-Zyklus beträgt für alle Beispiele 0.25 sec, wir erreichen also eine Echtzeiterkennung mit ca. 4 Hz.

Abbildung 27: Feature Matching für die Müslischachtel zur Laufzeit

5.3.2 Ergebnisse für Panoramaansichten:

Für die Poseerkennung in den Panoramaansichten verwenden wir perspektivische Projektionen einer Referenzansicht der Größe ca. 3600×2400 Pixel, wie für Abwendungfall (c) beschrieben, sowie Zielansichten der Größe 640×480 Pixel. Wir matchen hier gegen $N = 250, 500, 1000$ Referenzpunkte und erhalten eine durchschnittliche Inlierrate von $15\%, 30\%, 33\%$ (siehe *Abb. 28* für $N = 250$). Die Wiedererkennung ist also nicht ausreichend gut, um eine erfolgreiche Poseschätzung zu erlauben. Problematisch ist für diesen Fall die geringe Diche der Referenzpunkten in jeder einzelnen Ansicht. Da die einzelnen Ansichten nur ca. 1/30 des Gesamtbildes ausmachen, ist selbst bei günstiger Verteilung der Referenzpunkte im Bild mit nur je $N/30$ Referenzpunkten in jeder Ansicht zu rechnen. In der Trainingsphase ist aus dem selben Grund das Verhältnis von Hintergrundpunkten zu Referenzpunkten zu unausgewogen, um eine effektive Gewichtung der Referenzpunkte zu erlauben.
Die durchschnittliche Laufzeit eines Matching-Zyklus beträgt für dieses Beispiel unabhängig von der Anzahl der Klassen ebenfalls ca. 0.25 sec, wir erhalten also auch hier eine Erkennungsfrequenz von 4 Hz.

Abbildung 28: Featurepoint Matching in Panoramaansichten zur Laufzeit

6 Zusammenfassung

Zunächst wurden auf der Grundlage von [2] konkrete Definitionen für die betrachteten Datenstrukturen und Algorithmen erarbeitet, sowie die Anwendungsfälle spezifiert, für welche das Verfahren konkretisiert werden soll.
Im weiteren haben wir uns der Reihe nach mit dem Entwurf, der Implementierung und dem Testen der Klassen für den LaplacianGreedy-Detektor, allgemeine Entscheidungsbäume und Entscheidungsbaumtests sowie für den konkreten Fall des Pixelvergleichstests beschäftigt.
In einem weiteren Schritt wurde das Konzept der Viewset-Factory erarbeitet, um eine möglichst leicht erweiterbare Basis für das Erzeugen von Viewsets für das Lernen der Entscheidungsbäume umzusetzen. Die Klasse wurde für die Anwendungsfälle (a) Lernen mit Homographien, (c) Lernen von Panoramaansichten und (d) Lernen mit Videosequenzen realisiert, wobei letztere auf das Tracking-Framework zurückgreift und daher gesondert behandelt wurde.
Gleichzeitig wurden die Implementierungen für die betrachteten Anwendungsfälle getestet, um günstige Parameter zu ermitteln und die Effizienz zu überprüfen. Ein besonderes Interesse galt hierbei der Laufzeit der implementierten Algorithmen, da Echtzeiteinsatz eine der Zielsetzungen war. Leider musste festgestellt werden, dass insbesondere die Detektion noch nicht vollständig echtzeitfähig ist, was zum Teil an der unoptimierten Implementierung der Testmethoden für den Detektor festzumachen ist. Hier kann, etwa durch hardwarenähere Implementierung, noch optimiert werden. Das Featurepoint Matching zur Laufzeit scheint dagegen vielversprechend schnell zu sein, wozu eine weitere Optimierung der verwendeten Testmethoden sicherlich noch beitragen kann.
Es wurde darüberhinaus festgestekkt, dass die Laufzeiten über den Werten des Referenzprogramms [4] liegen, wobei die Erweiterbarkeit und Konfigurierbarkeit der darin verwendeten Strukturen in Frage zu stellen ist. An dieser Stelle sei erwähnt, dass das auf [4] zu Verfügung gestellte Demoprogramm offensichtlich andere Datenstrukturen verwendet (binäre statt ternärer Bäume), als im Paper beschrieben und womöglich weiter modifiziert wurde.
Der Zeitaufwand für die Offline-Phase resultiert aus der Zeit fr das Erzeugen von Viewsets. Diese hängt im Fall synthetisierter Ansichten maßgeblich von den verwendeten Abbildungsverfahren aus der BIAS-Bibliothek ab. Werden Videosequenzen als Lernmaterial verwendet, ist der Zeitaufwand sehr gering, allerdings wird hier einmalig das Tracking (und eventuell Nachbesserung, also Entstörung, Entzerrung, ...) der Lernsequenz notwendig, was sich recht aufwendig gestalten kann, und die Menge der Lernansichten ist - im Gegensatz zum Fall synthetisierter Ansichten - recht beschränkt.
Ein anderes Problem stellte der Speicherbedarf des Verfahrens dar. Um ein Viewset aus 1000 Ansichten mit jeweils max. 1000 Featurepoints mit einer Beschreibung von ca. 1 KByte im Speicher vorzuhalten, wie es in der Trainingsphase notwendig wird, ist ein Mindestspeicher im Bereich von 1 GByte notwendig! Die-

ses Problem konnte nur durch Beschränkung des Trainingsviewsets auf max. 500 Ansichten zunächst gelöst werden. Für die Baumkonstruktions bzw. das Lernen der Klassenverteilungen ist der Speicheraufwand unproblematisch, da hier Viewsets aus 100 Ansichten mit jeweils max. N Featurepoints genügen, woraus sich ein Speicherbedarf im Bereich von nur $10 - 100$ MByte ergibt.
Für die Objektdetektion stellten wir wie erwartet fest, dass die Güte der Erkennung stark von der Realitätsnahe sowie der Varianz und Anzahl der Lernansichten abhängt. Dementsprechend erhielten wir die größere Erfolgsquote bei Verwendung von umfangreichen, gerenderten 3D-Sequenzen als Lernmaterial. Da der Matcher lediglich aus der Lernphase bekannte Punkte zu unterscheiden lernt, ist die Erfolgsquote umso geringer, je mehr unbekannte Objekte im Bild erscheinen.
Für die Poseschätzung in Panoramaansichten stellte sich die direkte Anwendung des Verfahrens als ineffizient heraus, da die Anzahl der Klassen sehr groß gewählt werden muss, um zu gewährleisten, dass in jeder Ansicht genug Referenzpunkte für die Poseschätzung detektiert werden.
Abschließend lässt sich der Ansatz zum Lernen von Entscheidungsbäumen als zeitaufwendig in der Offline-Phase, aber echtzeitnah in der vorliegenden Implementierung und damit vielversprechend für Echtzeitverfahren bewerten, insbesondere auch für die (Re-) Initialisierung von Echtzeit-Trackingverfahren. Positiv ist die Möglichkeit, verschiedenartigste Lerndatenmengen zu verwenden, interessant die flexible Einsatzmöglichkeit der Entscheidungsbäume. Die Verwendung effizienterer Tests für die Entscheidungsbäume wäre ebenfalls noch zu untersuchen. Da die maßgebliche Zielsetzung der Implementierung auf Erweiterbarkeit und Flexibilität abzielte, sollte dieses einfach zu realisieren sein.

7 Anhang: Implementierung

Abschließend soll eine Übersicht über die implementierten Klassen, deren Entwurf in 4 motiviert wurde, und über ihre Funktionalität gegeben werden. Wir hoffen, dass diese Referenz die Verwendung der Klassen in zukünftigen Projekten vereinfacht.

7.1 Die Programmbibliotheken BIAS und MIP

Zunächst soll die Position der entworfenen Klassen innerhalb der Programbibliothek MIP verdeutlicht werden:

```
FeatureDetector
|
+- LaplacianGreedy                  Laplacian-Greedy-Featurepoint-Detektor

Matcher2D
|
+- DecisionTree
   |
   +- DecisionTree                  einzelner Entscheidungsbaum
   +- DecisionTreeNode              Entscheidungsbaumknoten
   +- DecisionTreeMatcher           Featurepoint-Matcher
   |
   +- IDecisionTreeTest             Interface für Featurepoint-Tests
   +- PixelComparisonTest           Pixelvergleichstest
   |
   +- ViewsetFactoryBase            Abstrakte Viewset-Factory
   +- ViewsetFactoryHomography      Factory für affine Transformationen
   +- ViewsetFactoryPerspective     Factory für perspektivische Projektionen

TrackingFramework
|
+- Modules                          Module für das Tracking-Framework
   |
   +- TFMFeatureDetectorLaplacian   Detektion von Featurepoints
   +- TFMCreateDecisionTrees        Erzeugen von Bäumen aus Videosequenzen
   +- TFMDecisionTreeMatcher        Matching mit Entscheidungsbäumen
```

7.2 Die Klasse `LaplacianGreedy`

Die Funktionalität des Laplacian-Greedy-Detektors orientiert sich an derjenigen bereits bestehende Detektoren (siehe *Abb.29*). Die Methode `Detect()` detektiert Featurepoints in einem 8-Bit-Graustufenbild vom Typ `BIAS::Image<unsigned char>` nach optionalem Weichzeichnen mit einem Gauss-Filter, und wählt aus den ermittelten Punkten optional mit der geschützten Methode *FilterByLaplacianGreedy()* die bzgl. ihrem *Laplacian of Gaussian* lokal optimalen Punkte aus. Die ermittelten Featurepoints werden mit normierter 2D-Orientierung und Erkennungsgüte attributiert, nach ihrer Güte sortiert und als Vektor von `FeaturePoint`-Objekten zurückgegeben.
Die Klasse verfügt außerdem über Methoden zur grafischen Darstellung von Featurepoints samt Orientierung in einem Bild bzw. der LoG-Werte, die in einem Bild ermittelt wurden.

Konfiguration von Instanzen: Die Toleranzschwelle `Tolerance_` für die Erkennung und der Radius `Radius_` des Scankreises können zwischen 1 und `MAX_TOLERANCE` bzw. `MAX_RADIUS` gewählt werden und sind standardgemäß mit `STD_TOLERANCE` bzw. `STD_RADIUS` vorbelegt.
Weichzeichnung und Filterung durch den LoG-Wert können für eine Instanz optional deaktiviert werden. Zu beachten ist bei der Verwendung der Klasse:

Klasse `LaplacianGreedy`

`+ static const unsigned int STD_TOLERANCE = 10`
`+ static const unsigned int MAX_TOLERANCE = 255`
`+ static const unsigned int STD_RADIUS = 7`
`+ static const unsigned int MAX_RADIUS = 32`
`- unsigned int Radius_`
`- unsigned int Tolerance_`

`+ LaplacianGreedy()`
`+ LaplacianGreedy(unsigned int r, unsigned int t)`
`+ unsigned int Detect(BIAS::Image<unsigned char> &image,`
`vector<FeaturePoint> &vecFP, int *mapLoG)`
`+ void SetScanRadius(unsigned int r)`
`+ void SetTolerance(unsigned int t)`
`+ void UseLoGFilter(bool b)`
`+ void UseSmoothing(bool b)`
`- unsigned int FilterByLaplacianGreedy(unsigned int width, height,`
`vector<FeaturePoint> &vecFP,`
`const int *mapLoG)`

Abbildung 29: Diagramm der Klasse `LaplacianGreedy`

- Die Erkennunggüte $Q \in [0, 1]$, die der Detektor einem Featurepoints zuweist, ist optimal für $Q = 0$ (hoher Wiedererkennungwert) und pessimal für $Q = 1$ (kein Wiedererkennungswert).
- Das Array zur Speicherung der LoG-Werte muss vor Aufruf von `Detect` für ein Bild der Größe $w \times h$ Pixel mit der Größe $w \cdot h$ angelegt werden.

7.3 Die `ViewsetFactory`-Klassen

Wie in Kapitel 4.2 motiviert, kapseln wir die Verwaltung von Ansichten für das Erzeugen und Trainieren der Entscheidungsbäume - also das Erzeugen von Ansichten und das Ermitteln von Korrespondenzen zwischen Featurepoints und Referenzpunkten in den erzeugten Ansichten - in einer *ViewsetFactory*-Klasse. Eine abstrakte Basisklasse `ViewsetFactoryBase` übernimmt hierbei die wesentlichen Aufgaben, die von der konkreten Art der Erzeugung von Ansichten unabhängig ist - nämlich Auswahl stabiler 3D-Referenzpunkte und das Erzeugen von Viewsets (= Lernmengen klassifizierter Featurepoints) - während fallspezifische Aufgaben - also das Erzeugen zufälliger Ansichten, die Rücktransformation von Featurepoints in das Referenzbild bzw. in 3D-Weltkoordinaten - durch entsprechende Erweiterungen der Basisklasse umgesetzt werden.
Für unsere Anwendungsfälle verwenden wir konkrete ViewsetFactory-Klassen `ViewsetFactoryHomography` (Erzeugen von Ansichten durch affine Transformation für planare Objekte) und `ViewsetFactoryPerspective` (Erzeugen von Ansichten durch perspektivische Projektion für Panoramaansichten). Das Erzeugen von Viewsets für vorliegende Videosequenzen ist im TrackingFramework-Modul `TFMCreateDecisionTrees` realisiert.

Basisklasse `ViewsetFactoryBase`

`+ virtual void GenerateView()`
`+ virtual void NextRandomParameters()`
`+ virtual void TranslateFeaturePoints(vector<FeaturePoint> &features,`
`vector<BIAS::HomgPoint2D> &newPos)`
`+ virtual void Get3DPoint(BIAS::HomgPoint3D &point3D,`
`BIAS::HomgPoint2D &point2D)`
`+ virtual void Get2DPoint(BIAS::HomgPoint2D &point2D,`
`BIAS::HomgPoint3D &point3D)`

`+ int GenerateRandomSamples(unsigned int size,`
`vector<FeaturePoint> &features,`
`vector<LabelledFeaturePoint> &samples, ...)`
`+ void GetLastView(BIAS::Image<unsigned char> &img)`
`...`

Abbildung 30: Diagramm der Basisklasse `ViewsetFactoryBase`

7.4 Die `DecisionTreeTest`-Klassen

Wir definieren allgemeine Featurepoint-Tests für Entscheidungsbäume über eine Schnittstelle `IDecisionTreeTest<ParamType>` (siehe *Abb.31*). Der Parametertyp eines Tests ist durch `ParamType` festgelegt.
Ein konkreter Test verfügt neben der Testauswertung `Evaluate()` und Angaben zur Anzahl der Resultatklassen `CaseCount_` und der maximalen Parameterzahl `ParamCount_` über eine Methode `NextRandomParams()` zum Ermitteln zufälliger Parameter für die Baumkonstruktion, sowie Methoden zum Lesen und Schreiben von globalen Testparameter aus/in Dateien.
Als Beispiel wird das Interface `IDecisionTreeTest` mit dem Parametertyp `int` durch den Pixelvergleichstest `PixelComparisonTest` implementiert.

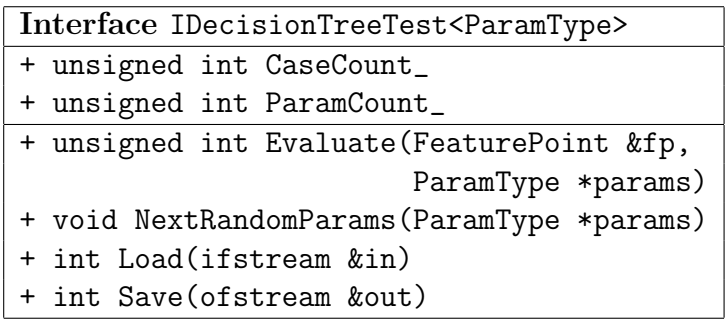


Abbildung 31: Diagramm der Schnittstelle `IDecisionTreeTest`

7.5 Die Klassen `DecisionTree` und `DecisionTreeMatcher`

Wir definieren einzelne Entscheidungsbäume unabhängig von der Art der Tests durch eine Templateklasse `DecisionTree<ParamType>` (siehe *Abb.33*). Einzelne Baumknoten sind entsprechend durch `DecisionTreeNode<ParamType>` implementiert (siehe *Abb.32*), wobei innere Knoten mit Testparametern in Form eines Array vom Typ `ParamType` ssoziiert sind, Blätter mit Klassenverteilungsvektoren `FeatureClasses[]`. Indem wir in den Blättern nicht direkt die Wahrscheinlichkeiten d_c speichern, sondern die Gesamtzahl aller Features `FeatureCount` und die Anzahl der Features pro Klasse `FeatureClasses[c]`, lässt sich die Klassenverteilung zu jedem Zeitpunkt mit weiteren gelabelten Featurepoints updaten.
Knoten sind nach ihrer Instanziierung zunächst undeklariert und werden erst durch Aufruf der Methoden `CreateInnerNode()` bzw. `CreateLeaf()` als innere Knoten bzw. Blätter deklariert.
Ein Matcher ist kanonisch durch eine Templateklasse `DecisionTreeMatcher<ParamType>` definiert, welche mehrere Entscheidungsbaum-Instanzen akkumuliert (siehe *Abb.34*).
In der gegenwärtigen Realisierung teilen sich alle Entscheidungsbäume eines Matcher denselben Test und dieselbe Menge von Referenzpunkten, die Implementierung lässt eine Erweiterung allerdings einfach zu.

Die Konstruktion und das Training erfolgt wie in Kapitel 4.4 beschrieben durch die Methoden `DecisionTree::Build()`, `DecisionTree::EstimateDistributions` und `DecisionTreeMatcher::TrainIdentification()`, welche jeweils ein Viewset in Form eines Vektors von klassifizierten Featurepoints (`LabelledFeaturePoint`) als Parameter erhalten.
Zum Speichern und Laden von Entscheidungsbäumen benutzen wir ein ähnliches Dateiformat wie in [4], wobei wir einzelne Bäume in separaten Dateien speichern, welche in der Datei des Matchers referenziert werden. Auf diese Weise können einzelne Bäume einfach ausgetauscht oder hinzugefügt werden.
Im Rahmen der Studienarbeit testen wir lediglich Entscheidungsbäume mit dem oben definierten Pixelvergleichstest (siehe Kollaborationsdiagramm 23). Prinzipiell lassen sich aber beliebige Tests verwenden, welche die Schnittstelle `IDecisionTreeTest` verwenden, wobei der Parametertyp `ParamType` der Templateklassen gemäß dem Testparametertyp gewählt werden muss.

Klasse `DecisionTreeNode<ParamType>`

`+ unsigned int Depth_`
`+ unsigned int ChildCount_`
`+ DecisionTreeNode<ParamType>* Children_[]`
`+ ParamType TestParams_[]`
`+ unsigned int FeatureCount_`
`+ unsigned int FeatureClasses_[]`

`+ unsigned int CreateInnerNode(...)`
`+ unsigned int CreateLeaf(...)`
`+ bool IsLeaf()`

Abbildung 32: Diagramm der Klasse `DecisionTreeNode`

7.6 Integration in das `TrackingFramework`

Abschließend integrieren wir die entworfenen Komponenten LaplacianGreedy-Detektor, Konstruktion der Entscheidungsbäume und Anwenden des Featurepoint-Matchers zur Poseschätzung in das TrackingFramework der MIP-Bibliothek. Hierzu binden wir die Klassen in entsprechende Module `TFMDetectorLaplacian`, `TFMCreateDecisionTrees` und `TFMDecisionTreeMatcher`, wobei wir uns an bestehenden Modulen orientieren. Insbesondere sind alle Module durch XML-Parameterdateien konfigurierbar.
Dem Aufruf des Moduls `TFMCreateDecisionTrees` zum Erzeugen von Entscheidungsbäumen aus Videosequenzen muss eine Szenenrekonstruktion, konkret: das Tracking von Featurepoints durch KLT und die Rekonstruktion der 3D-Koordinaten der Featuretrails durch Triangulation vorausgehen, wozu die Module `TFMCorrespondenceGeneratorKLT` und `TFMTriangulationSimple` verwendet werden.

Klasse DecisionTree<ParamType>
+ static const int UNIDENTIFIED = -1 - DecisionTreeNode<ParamType> Root_ - IDecisionTreeTest<ParamType>* Test_ - vector<BIAS::HomgPoint3D>* ReferencePoints_ - unsigned int ClassCount_ - unsigned int MaxDepth_ ...
+ int ClassifyFeaturePoint(FeaturePoint &fp, double *distr) + void Build(vector<LabelledFeaturePoint> &samples) + void EstimateDistributions(vector<LabelledFeaturePoint> &samples) + int Load(string filename) + int Save(string filename)

Abbildung 33: Diagramm der Klasse DecisionTree

Klasse DecisionTreeMatcher<ParamType>
- DecisionTree<ParamType>* Trees_[] - IDecisionTreeTest<ParamType>* Test_ - vector<BIAS::HomgPoint3D>* ReferencePoints_ - double IdThresholds[]_ ...
+ int ClassifyFeaturePoint(FeaturePoint &fp, double *distr) + void TrainIdentification(vector<LabelledFeaturePoint> &samples) + DecisionTree<ParamType>* GetDecisionTree(unsigned int k) + int Load(string filename) + int Save(string filename, string treefiles[]) ...

Abbildung 34: Diagramm der Klasse DecisionTreeMatcher

Nach Aufruf des Moduls TFMDecisionTreeMatcher zum Featurepoint-Matching von Eingabebildern kann die Pose abschließend durch das Modul TFMExtendedPoseEstimatorNumeric (Implementierung des POSIT-Algorithmus) geschätzt werden.

Literatur

[1] Vincent Lepetit, Pascal Lagger, Pascal Fua: *"Randomized Trees for Real-Time Keypoint Recognition"*, Conference on Computer Vision and Pattern Recognition, San Diego, CA, Juni 2005.

[2] Vincent Lepetit, Pascal Fua: *"Towards Recognizing Feature Points Using Classification Trees"*, in: *"Technical Report IC/2004/74"*, Ecole Polytechnique Federale de Lausanne, September 2004.

[3] Vincent Lepetit, Julien Pilet, Pascal Fua: *"Point Matching as a Classification Problem for Fast and Robust Object Pose Estimation"*, Conference on Computer Vision and Pattern Recognition, Washington, DC, Juni 2004.

[4] Homepage des Computer Vision Laboratory - Ecole Polytechnique Federale de Lausanne (*http://cvlab.epfl.ch/research/augm/detect.html*)

[5] D. de Menthon, L.S. David: *"Model-Based Object Pose in 25 Lines of Code"*, in: *"International Journal of Computer Vision, Vol. 15"*, S.123-141, 1995.

[6] M. A. Fischler, R. C. Bolles: *"Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography"*, in: *"Communications of the ACM, Vol. 24 (6)"*, S.381-395, 1981.

[7] David G. Lowe: *"Distinctive Image Features from Scale-Invariant Keypoints"*, in: *"International Journal of Computer Vision, Vol. 60 (2)"*, S.91-110, 2004.

[8] Richard Hartley, Andrew Zisserman: *"Multiple View Geometry in Computer Vision"*, 2000.

[9] Carlo Tomasi, Takeo Kanade: *"Detection and Tracking of Point Features"*, in: *"Carnegie Mellon University Technical Report CMU-CS-91-132"*, April 1991.

[10] M. Pollefeys, L. Van Gool, M. Vergauwen, F. Verbiest, K. Cornelis, J. Tops, R. Koch: *"Visual Modeling with a Hand-held Camera"*, in: *International Journal of Computer Vision, Vol. 59 (3)*, S.207-232, 2004.

[11] C. Harris, M. Stephens: *"Combined Corner and Edge Detector"*, in: *"Proceedings of the 4th Alvey Vision Conference"*, S.147-151, 1988.

[12] Michael Bender, Manfred Brill: *"Computergrafik"*, S.10-39, 2003.

[13] B. Jahne, H. Haubecker, P. Geibler: *"Handbook of Computer Vision and Applications"*, 1999.