Informatik
Hauptcampus

H O C H
S C H U L E
T R I E R

Clear Up

Gruppe A:
Jonas Eckstein,
Denis Wagner

Labor Robotik Dokumentation

Betreuer: Christoph Zinnen

Trier, 14. Januar 2019

Inhaltsverzeichnis

1

Einleitung und Problemstellung

Eine Kinderspielecke ist ein angenehmer Ort zum spielen. Es wird mit Spielzeugen gespielt und einige werden auch gerne mal liegen gelassen. Das Aufräumen ist jedoch eine wichtige Aufgabe, die erledigt werden muss, sodass alle Spielzeuge wieder zurück in die Spieltruhe finden. Ein sozialer autonomer mobiler Roboter könnte diesem Problem Abhilfe schaffen und das Aufräumen kompetent autonom erledigen.

1.1 Verhalten

Der Roboter soll sich in einem Raum, wie in Abbildung 1.1 dargestellt, bestehend aus einer Spielecke und einer Spieltruhe bewegen. Innerhalb der Spielecke befinden sich zu beginn verteilt Spielzeuge, welche sich durch unterschiedliche Farben auszeichnen und anhand dieser vom Roboter erkannt werden sollen. Zu Beginn wartet der Roboter auf eine sprachliche oder visuelle Eingabe und sammelt daraufhin die entsprechenden Gegenstände ein und bringt sie in die Spieltruhe. Da es sich um eine Umgebung mit Kindern handelt, soll auf dynamische Hindernisse reagiert und entsprechend ausgewichen oder gestoppt werden.

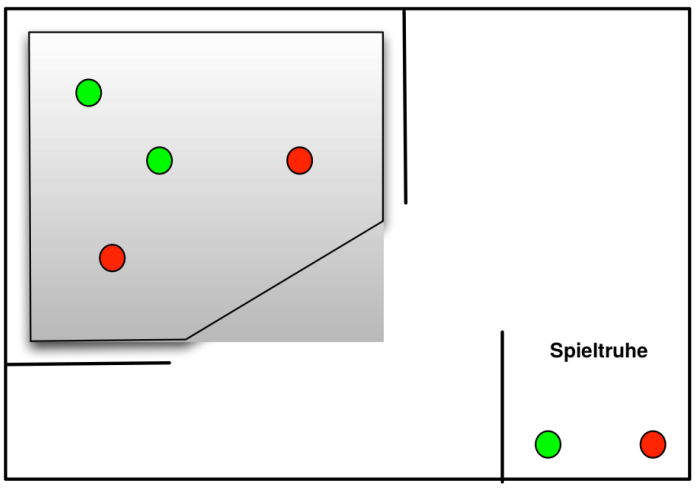

Abb. 1.1. Originales Konzept der Umgebung (Quelle: Christoph Zinnen)

1.2 Anforderungen

Das Robotik-Projekt *Clear up* beinhaltet disziplinen aus jedem Bereich der Robotik, die in der Vorlesung behandelt wurden. Die Bearbeitungsdauer betrug vier Wochen. Der Arbeitsumfang beinhalt die Programmierung eines Verhaltes auf dem Roboter Pioneer 3DX, einer Dokumentation und einer Präsentation in Form eines Plakats. Neben der allgemeinen Lösung der Aufgaben, soll das Verhalten des Roboters einige Anforderungen erfüllen:

- Die Aufgaben soll der Roboter *ökonomisch* lösen, was eine Planung der Aktionen und dessen Optimierung beinhaltet.
- Positionen sollen mithilfe korrekter Loklisierung in der Umgebung *genau* angefahren werden können, um Abweichungen und Fehler zu minimieren.
- Hindernisse sollen erkannt und sinnvoll darauf reagiert werden, um materielle und personelle Schäden zu verhindern, da es sich vor allem um einen sozialen Roboter handelt.
- Der Roboter soll seine Aufgaben außerdem zuverlässig und robust erfüllen. Dazu zählt auch eine gewisse Fehlertolleranz, sodass er sich auch bei unbekannten Situationen möglichst korrekt verhält.

2

Lösungsansatz

Am Anfang befindet sich der Roboter in einer vorgegebenen Startposition und wartet auf ein visuelles Startsignal. Dieses wird dem Roboter gegeben, indem ihm ein farbiger Gegenstand vor die Kamera gehalten wird. Der Roboter erkennt die Farbe und begibt sich daraufhin zu einem vorgegebenen Punkt, welcher sich im Spielbereich befindet. Zur Pfadplanung soll eine einfache Rastarisierung und der A* Algorithmus verwendet werden. Zur Bestimmung der derzeitigen Position sollen außerdem die Sensoren (Laser und Sonar) verwendet werden, die genaue Methode und Umsetzung war uns hierbei jedoch noch nicht klar. Ist er dort angekommen, soll er sich durch drehen umsehen und mit der Kamera nach Gegenständen suchen, welche die selbe Farbe haben, wie der Gegenstand, der als Startsignal verwendet wurde. Findet er einen solchen Gegenstand, soll der Roboter seinen Gripper öffnen und auf den Gegenstand zufahren, bis dieser sich innerhalb des Grippers befindet. Dabei soll er über die Kamera mit dem Strahlensatz eine Abstandsabschätzung vornehmen und seine Geschwindigkeit entsprechend anpassen. Befindet sich der Gegenstand in seinem Gripper, soll der Roboter den Gegenstand aufnehmen und seine Pfadplanung verwenden, um zu einem Ablagebereich zu fahren, welcher ebenfalls vorgegeben ist. Dort angekommen soll er den Gegenstand wieder ablegen. Dabei soll er darauf achten, Gegenstände, die er dort bereits abgelegt hat, nicht zu überfahren. Nachdem der Roboter den Gegenstand abgelegt hat, fährt er wieder zum Spielbereich und sucht nach weiteren Spielzeugen. Findet er keine weiteren Gegenstände, soll er zurück zu seiner Startposition fahren und auf ein neues Startsignal warten.

3

Durchführung (tatsächliche Umsetzung)

Für die programmiertechnische Umsetzung haben wir uns dazu entschieden, einen großen Zustandsautomaten zu verwenden, welcher den Programmablauf steuert. Er besteht aus insgesamt 11 Zuständen, welche in Kapitel 3.1 genauer beschrieben sind.
Aus Zeitgründen verwendet unser Programm zur Navigation anstelle eines eigenen Algorithmus, wie es ursprünglich geplant war, den *ArPathPlanningTask*. Als Zielpunkte sind dabei in der *Map* drei Punkte vorgegeben: der Startpunkt (*home*), der Spielbereich (*playarea*), sowie der Ablagebereich (*chest*).
Zur Objekterkennung wird *ACTS* verwendet. Hierfür werden verschiedene Farben in verschiedenen *Channels* trainiert. Das Programm selber wird den Roboter so steuern, dass er alle Gegenstände einsammelt, welche in dem selben *Channel* erkannt werden, wie das Startsignal. Dabei vermeidet er, Gegenstände einzusammeln, welche durch einen anderen *Channel* erkannt werden.

3.1 Zustände

Der Zustandsautomat verwendet insgesamt 11 Zustände, um das Verhalten des Roboters zu steuern. Ein Zustandsdiagramm ist in Abbildung 3.1 zu finden. Dabei sind im Zustandsdiagramm jedoch nur 9 Zustände aufgeführt, da die anderen zwei Zustände einen Sonderfall darstellen und für die Übersicht nicht wichtig sind. Nachfolgend sind die 11 Zustände genauer beschrieben.

- *WaitForColorState:* Der Roboter prüft, ob verschiedene Vorbedingungen erfüllt sind. So muss beispielsweise eine Verbindung zu *ACTS* möglich sein. Außerdem prüft er, ob der Laser bereits aktiviert ist. Ist dem nicht so, versucht er ihn zu aktivieren. Ist eine der Bedingungen nicht gegeben, wechselt er in den *TerminationState*. Sind alle Vorbedingungen erfüllt, durchläuft der Roboter immer wieder alle *Channels* von *ACTS* und wartet auf einen *Blob*, welcher eine gewisse Mindestgröße besitzt. Findet er einen solchen *Blob*, setzt er diesen *Channel* als *Zielchannel* und wechselt in den *GoToPlayAreaState*.

- *GoToPlayAreaState:* Der Roboter liest das Goal *playarea* aus der *Map* aus und benutzt den *DriveToTargetState*, um sich zu diesem zu begeben. Dort angekom-

men, wechselt er in den *SearchToyState*.

- *SearchToyState:* Der Roboter sucht in diesem Zustand nach Spielzeugen im Spielbereich. Dazu dreht er sich zuerst nach links, bis er durch seinen Laser die linke Wand des Spielbereichs vor sich erkennt. Anschließend dreht er sich langsam nach rechts. Findet seine Kamera dabei ein Spielzeug zum aufräumen (das heißt, im *Zielchannel* wird von *ACTS* ein *Blob* erkannt), wechselt er in den *FocusOnColorState*. Findet er keines, sondern erkennt mit seinem Laser die rechte Wand des Spielbereichs, wechselt er in den *GoToHomeState*.

- *FocusOnColorState:* Der Roboter sucht mit Hilfe von *ACTS* nach dem größten *Blob* im *Zielchannel*. Er versucht, diesen durch neigen der Kamera sowie durch drehen des Roboters diesen in der Mitte der Kamera zu zentrieren. Ist dies geschehen, ist der Roboter auf den Gegenstand ausgerichtet und er wechselt in den *GoToToyState*.

- *GoToToyState:* Der Roboter ist nun auf ein Spielzeug ausgerichtet und fährt seinen *Gripper* runter und öffnet ihn, während er beginnt das Spielzeug anzufahren. Mithilfe des Strahlensatzes wird die Entfernung zu dem Objekt berechnet und daraufhin die Geschwindigkeit angepasst. Wenn Das Spielzeug durch die äußere Lichtschranke des *Grippers* bricht, reduziert er seine Geschwindigkeit auf ein Minimum um rechtzeitig zum Halten zu kommen, um das Spielzeug nicht wegzuschieben und wechselt danach in den Zustand *TakeToyState*. Bei der Anfahrt an das Spielzeug überprüft der Roboter mithilfe des Lasers die Entfernung zu anderen Objekten vor sich und stoppt entsprechend, sodass dynamische Objekte und Wände nicht angefahren werden. Auch überprüft der Roboter Spielzeuge von der falschen Farbe, die eventuell vor einzusammelnden Spielzeugen liegen und stoppt auch dann, um das falsche Spielzeug nicht stattdessen einzusammeln. Dies wurde umgesetzt, indem mithilfe der Kamera und *ACTS* ein Rechteck am unteren Rand des Kamerabildes aufgespannt und geprüft wurde, ob sich ein *Blob* falscher Farbe darin befindet, sofern es kein *Blob* richtiger Farbe gibt, dessen Y-Koordinate des *Blob-Mittelpunktes* vor dem falschen *Blob* liegt. Durch die Perspektive der Kamera würde der rechteckige Fahrtweg zwar in einem Trapez resultieren, jedoch wurde das Trapez aufgrund der besonders niedrigen Höhe aber höheren Breite als Rechteck approximiert. Wurde ein falsches Spielzeug erkannt, wartet er einige Sekunden lang, bevor er zum Home-Point fährt und damit die Aufräumaktion für diese Farbe abbricht (Eigentlich war hier ein Umfahren des Hindernisses geplant, dieses wurde aus Zeitgründen jedoch nicht fertig gestellt).

- *TakeToyState:* In diesem Zustand wird der *Gripper* geschlossen und anschließend angehoben. Es wird auf das vollständige Schließen des *Grippers* gewartet, bevor das Spielzeug angehoben wird, damit das Objekt nicht runterfallen kann. Wurde der *Gripper* vollständig hochgefahren, welchselt er in den Zustand *Go-*

ToChestState.

- *GoToChestState:* Der Roboter liest das Goal *chest* aus der *Map* aus und benutzt den *DriveToTargetState*, um sich zu diesem zu begeben. Dort angekommen, wechselt er in den *PutToyState*.

- *PutToyState:* Der Roboter merkt sich die Anzahl der bereits abgelegten Spielzeuge und variiert anhand dessen die Entfernung, die er zur Wand (in der Spieltruhe) haben muss, um nicht mit den bereits abgelegten Spielzeugen zu kollidieren. Er fährt zunächst auf die Wand zu und hält an, wenn er nah genug ist. Anschließend fährt er den *Gripper* runter und wartet darauf, bis der *Gripper* vollständig unten ist. Nach dem vollständigen Öffnen des *Grippers* fährt er rückwärts aus der Kiste heraus und überprüft mithilfe der Sonarsensoren, ob sich ein Gegenstand im Bereich hinter des Roboters befindet und stoppt, wenn er eines erkennt. Nach erfolgreichem Verlassen der Spieltruhe wechselt er in den Zustand *GoToPlayAreaState*.

- *GoToHomeState:* Der Roboter liest das Goal *home* aus der *Map* aus und benutzt den *DriveToTargetState*, um sich zu diesem zu begeben. Dort angekommen, wechselt er in den *WaitForColorSignalState*.

- *DriveToTargetState:* Der DriveToTargetState entspricht nicht komplett einem klassischen Zustand in einem Zustandsautomaten, da sein Verhalten von einigen globalen Variablen abhängt, welche er verwendet, um mit anderen Zuständen zu kommunizieren. Der DriveToTargetState wird verwendet, um den *ArPathPlanningTask* zu steuern und in den Zustandsautomaten zu integrieren. Wird er aufgerufen, liest er aus globalen Variablen eine Zielposition aus und gibt dem *ArPathPlanningTask* den Auftrag, dorthin zu navigieren. Hat der *ArPathPlanningTask* seine Navigation beendet, wird in einer globalen Variable nachgelesen, in welchen Zustand nun gewechselt werden soll. Außerdem wird gespeichert, ob die Navigation erfolgreich war.

- *TerminationStateState:* Der TerminationState sollte im normalen Betrieb nicht aufgerufen werden. Er wird benutzt, um den Zustandsautomaten abzuschalten und somit die Ausführung des Programms zu beenden. Wird in einem der anderen Zustände ein Fehler erkannt, welcher nicht korrigierbar ist, wird in den TerminationState gewechselt.

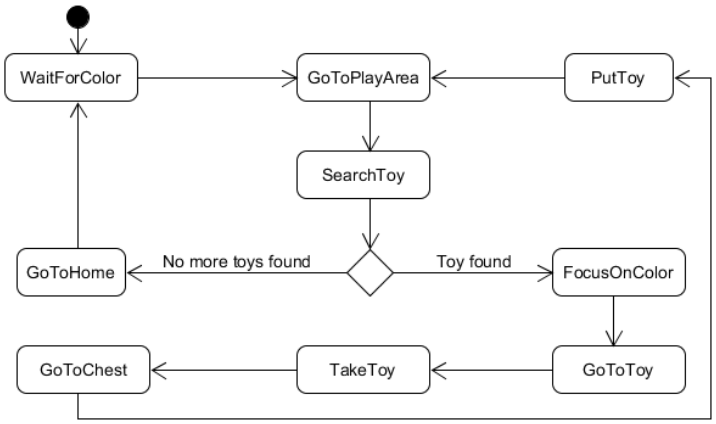

Abb. 3.1. Aufbau des Zustandsautomaten. Die Zustände *DriveToTargetState* sowie *TerminationState* fehlen in dieser Abbildung, da sie die Übersichtlichkeit beeinträchtigen würden. Sie sind jedoch in Kapitel 3.1 genauer beschrieben.

3.2 Probleme bei der Entwicklung

Im Laufe der Entwicklung traten mehrere Probleme auf, welche schwieriger zu lösen waren als ursprünglich vermutet. Im Folgenden werden die Probleme aufgeführt, die uns am meisten Zeit gekostet haben.

ACTS erkennt Gegenstände nicht zuverlässig

Das Farberkennungstool ACTS soll die Bilddaten, welche von der Kamera kommen, analysieren und darin Flächen erkennen, die einer trainierten Farbe entsprechen. Leider funktioniert das nicht immer wirklich gut. So wird oft irgendwo im Hintergrund eine ähnliche Farbe ebenfalls erkannt oder Objekte werden nur aus einem gewissen Winkel, also unter bestimmten Lichtverhältnissen erkannt. Unter anderem für die Abstandsabschätzung ist eine möglichst genaue Erfassung eines Gegenstands jedoch wichtig.
Unser erster Lösungsansatz war es, die Bibliothek *OpenCV* zu verwenden. Da es dort allerdings zu Problemen kam, welche in Unterkapitel 3.2 beschrieben werden, haben wir weitere Lösungen erarbeitet.
So haben wir eigene Gegenstände mit in das Labor gebracht, welche von den Farben her so gewählt sind, dass sie möglichst keine Ähnlichkeit mit anderen Gegenständen im Raum haben. Außerdem sind sie rund, wodurch Änderungen an der Beleuchtung nicht mehr ganz so starke Auswirkungen haben. Des Weiteren haben wir die Kamerasteuerung so umgeschrieben, dass sie nicht mehr zu weit nach oben schauen kann. Dadurch können wir stärker kontrollieren, welche Gegenstände die Kamera überhaupt sehen kann und so Gegenstände ausschließen, die gegebenenfalls Probleme verursachen könnten. Das Herunterlassen der Rollladen hat ebenfalls geholfen, da die Lichtverhältnisse im Labor nun nicht mehr vom

Wetter oder der Tageszeit abhängig sind. Da diese Lösung nun für uns gut funktioniert, haben wir dann auch auf den Einsatz von *OpenCV* verzichtet, obwohl wir das Problem mit den Bilddaten in der Zwischenzeit ebenfalls gelöst haben.

Beim Abfragen von Informationen aus ACTS können Blobs verloren gehen

Da ACTS asynchron zu unserem Programm läuft kann es beim Auslesen zu unerwünschten Effekten kommen. Es kann vorkommen, dass ACTS beispielsweise zu einem Zeitpunkt mehrere Blobs erkennt. Unser Programm filtert diese, indem es eine gewisse minimale Größe eines solchen Blobs erwartet, ansonsten werden diese Blobs verworfen. Durch den parallelen Ablauf zwischen unserem Programm und ACTS kann es nun aber vorkommen, dass, obwohl ein Gegenstand immer vor der Kamera liegen bleibt, er für einige wenige Programmzyklen nicht erkannt wird. Dieses Problem tritt beispielsweise auf, wenn ACTS zum Beispiel zwei Blobs erkennt und wir den ersten Auslesen, dieser jedoch ein eher kleiner Blob ist und verworfen wird (was auch korrekt ist, da die vielen kleinen Blobs durch zum Beispiel durch Bildrauschen entstehen). Wenn nun, bevor wir den zweiten Blob abfragen, ACTS seine Blobs aktualisiert und nur noch den einen, großen Blob erkennt, rutscht dieser Blob in ACTS auf Position 1. Fragen wir nun nach dem zweiten Blob, gibt ACTS uns keinen weiteren Blob zurück. Dadurch haben wir in diesem Zyklus den großen Blob praktisch übersehen.
Um dieses Problem zu lösen ignoriert unser Programm es, wenn einige wenige Zyklen ein Objekt nicht mehr von ACTS erkannt wird. Erst wenn es eine gewisse Zeit lang nicht mehr erkannt wird, gehen wir davon aus, dass das Objekt nicht mehr vor der Kamera zu sehen ist.

Schlechte Sensorerausstattung hinter dem Roboter

Vor dem Roboter liefern die Lasersensoren zuverlässige Ergebnisse. Jedoch kann hinter dem Roboter nicht auf sie zugegriffen werden. Dort mussten wir die Sonarsensoren verwenden, um im Zustand *PutToyState* dynamische Hindernisse hinter dem Roboter zu erkennen, wenn er versucht rückwärts aus der Spielkiste herauszufahren. Große Hindernisse, die sich geradlinig hinter dem Roboter befinden können gut erkannt werden, jedoch existieren Punkte etwas seitlich vom Roboter, die genau zwischen zwei Sonarsensoren liegen, die folglich nicht abgefragt werden können. Falls beispielsweise ein Bein in genau diesem Bereich liegt, würde dieses nicht erkannt werden und der Roboter fährt gegebenenfalls gegen das Bein oder fährt über den Fuß, was unter allen Umständen zu vermeiden sein sollte. Ein Lösungsansatz wurde dafür noch nicht gefunden, da sich der Laser oder die Kamera nicht nach hinten ausrichten lässt. Würde sich der Roboter sofort drehen, könnte das Spielzeug eventuell beschädigt werden.

Integration von OpenCV

Um eine bessere Spielzeugerkennung zu erhalten, wurde OpenCV anstelle von ACTS in Betracht gezogen. OpenCV hat den Vorteil, dass Bilddaten in unter-

schiedlichen Farbräumen analysiert werden können. Der HSV-Farbraum bietet sich besonders an, da variierende Lichtverhältnisse keinen so großen Enfluss haben, wie im RGB-Farbraum in dem ACTS arbeitet. Des Weiteren konnte eine stabilere Erkennung erreicht werden, wenn die Spielzeuge als Bälle definiert wurden. Die in OpenCV integrierte Hough Transformation erkennt Kreise innerhalb eines Bildes, wodurch andere gleichfarbige Elemente in der Umgebung nicht erkannt werden, wenn sie keine Kreisform aufweisen. Also wurde Farberkennung mit einer einfachen Form von Objekterkennung verbunden. Jedoch konnte OpenCV nicht auf dem Roboter zum Laufen gebracht werden, da die Kamera des Roboters kein eigenständiges Gerät, sondern einen Videostream darstellt (womit das die bisherige Programm noch nicht umgehen konnte). Die Verwendung des Videostreams und die Integration des Programms in das Roboterverhalten hätten einen zu großen Zeitaufwand bedeutet, wodurch die Spielzeugerkennung weiterhin auf OpenCV basiert.

Mangelnde C++ Erfahrung sowie unbekannte, teils schlecht dokumentierte Hardware

Ein weiteres großes Problem für uns war unsere mangelnde Erfahrung mit C++ sowie die für uns unbekannte Hardware und dessen teilweise schlecht dokumentierte Code-Bibliothek. Vor allem am Anfang des Projektes ist viel Zeit dadurch verbraucht worden, Fehler wie Segmentation Faults zu finden und zu beheben, die durch unsere mangelnde Erfahrung mit C++ entstanden sind. Auch war es teilweise nicht immer einfach, herauszufinden, wie gewisse Dinge auf dem Roboter funktionieren, da die Dokumentation der Bibliotheken, die den Roboter steuern, nicht alle Fragen beantwortet haben. Als Beispiel ist hier der *ArPathPlanningTask* zu nennen. In der Dokumentation wurde zwar erwähnt, dass mit dem starten der Pfadplanung alle ArActions (also auch unser Zustandsautomat) deaktiviert werden, jedoch nicht, wie man sie am Ende wieder reaktivieren sollte. Diese Probleme haben sich mit wachsender Erfahrung von selbst gelöst, da wir gelernt haben, diese Fehler zu vermeiden.

4

Zusammenfassung und Ausblick

In den vier Wochen der Entwicklungszeit sind ca. 130 Stunden für die Entwicklung des Roboterverhaltens investiert worden. Das Ergebnis dieser Entwicklungszeit ist ein Roboter, der die Anforderungen aus Unterkapitel 1.2 erfüllt und die in Unterkapitel 1.1 beschriebenen Aufgaben mithilfe eines Zustandsautomaten und 11 miteinander arbeitenden Zuständen in vielen Fällen robust und zuverlässig erledigt. Jedoch existieren einige Randfälle, mit denen der Roboter nicht korrekt umgeht und eventuell Fehler verursacht.

Wie schon in Unterkapitel 3.2 beschrieben, führen wechselhafte Lichtverhältnisse dazu, dass ACTS die Spielzeugfarben nicht gut erkennen kann, wodurch die Zuverlässigkeit des Roboters erheblich beeinträchtigt wird. Als Ersatztechnologie haben wir uns mit OpenCV beschäftigt und auch eine anfängliche Implementierung einer Kreiserkennung (da die Spielzeuge in unserem Szenario Bälle sind) gebündelt mit Farberkennung angefertigt, die in das Programm eingebunden werden könnte. Aus Zeitgründen konnte diese jedoch nicht fertiggestellt werden.

Zustände müssen sich nicht um die Fahrtplanung selber kümmen, jedoch werden sie eventuell nach einer gescheiterten Fahrtplanung aufgerufen, wodurch sie auch für die Fehlerbehandlung bei einer fehlerhaften Planung oder Fahrt zuständig sind. Unsere Zustände reagieren alle mit einer Rückfahrt zum *home*-Ziel. Dies ist nicht die optimale Lösung, zumal die Rückfahrt selber auch fehlschlagen kann. Allerdings könnten bessere Fehlerbehandlungen als Verbesserungen recht einfach aufgrund unserer Programmstruktur eingebaut werden.

In dem Zustand *PutToyState* aus Unterkapitel 3.1 wird zur Zeit noch darauf vertraut, dass Spielzeuge, die einmal in die Spielkiste gelegt wurden, diese nicht verlassen und neue Spielzeuge alle hintereinander in der Spielkiste in einer Reihe angeordnet werden. Eine Verbesserung für diesen Zustand könnte sein, dass mithilfe der Kamera alle Spielzeuge erkannt werden und so dynamisch zu entscheiden, wo künftige Spielzeuge abgelegt werden sollen.