6

Implementierung

Dieses Kapitel umfasst die wichtigsten Aspekte, welche die Implementierung der Scanner API sowie der notwendigen Services (Server, Simulator) betreffen. Ein vollständiges Objektmodell des Scanner-Systems kann im Anhang A gefunden werden. Wegen der gegenseitigen Abhängigkeiten der einzelnen Komponenten kann an einigen Stellen nicht verhindert werden, auf nachfolgende Kapitel zu verweisen. Dies sollte aber kein Problem für das Verständnis darstellen, wenn man sich immer vor Auge hält, dass das beschriebene Software-System aus einem Client und mehreren Servern besteht, welche über geeignete Protokolle auf einem TCP/IP-Netzwerk miteinander verbunden sind.

6.1 Scanner API

Die Scanner API (Application Programmer's Interface) ist eine objektorientierte Schnittstelle zur Steuerung eines Gesichtsscanners. Sie erlaubt den Zugriff auf alle konfigurierbaren Parameter und Klassen, die den Aufbau des Gesichtsscanners definieren. Gleichzeitig stellt sie Methoden zur Verfügung, um den Scanner zu programmieren und Bilder zu akquirieren.

Ein wichtiger Punkt ist, dass die Scanner API nicht zwischen realen und simulierten Kameras unterscheidet. Natürlich machen viele Kameraparameter, welche für die Simulation konfiguriert werden können, bei der Verwendung von CCD-Kameras keinen Sinn und können deshalb in den Konfigurationsdateien für den Realkamera-Modus vernachlässigt werden (zum Beispiel die Angabe von Clipping-Ebenen). Die Kapselung der Art der Kameras erlaubt aber, dass während des Betriebs eines Scanners zu jedem Zeitpunkt zwischen Simulation und Realkamerasystem gewechselt werden kann, ohne dass sich die Applikationsschnittstelle ändert. Wie später noch gezeigt werden wird, wurde die Unterscheidung der beiden Betriebsmodi ganz im Sinne des objektorientierten Designs erst an der letzten möglichen Stelle im Klassendiagramm realisiert.

6.1.1 Kameras

Die Klasse `Camera` repräsentiert eine einzelne Kamera. Ein Gesichtsscanner wird aus einer Menge von Kameras aufgebaut, woraus folgt, dass die Scanner API eine Liste von Kamerainstanzen verwalten muss. Der Zugriff auf ein solches Objekt geschieht über die Methode `get-Camera`, welche einen Zeiger auf die gefragte Instanz zurückliefert. Die Indizes für den Zugriff auf die Objekte werden durch die Reihenfolge des Einfügens in die Scannerkonfiguration

bestimmt. Um diese Schnittstelle anwendungsfreundlicher zu gestalten, besteht die Möglichkeit, jeder Kamera einen Namen zu geben, über den sie in der Liste der Kamerainstanzen ebenfalls gefunden werden kann.

Obwohl die Kameras den wichtigsten Bestandteil des geplanten Scanners darstellen, besitzt diese Klasse nur wenige Parameter: Den Namen des Servers, auf dem die Kamera repräsentiert wird, sowie eine Kamerakalibrierung. Letztere ist ein Objekt, das sämtliche extrinsischen und intrinsischen Kalibrierungsparameter enthält und das über die Methode `getCalibration` erhalten werden kann.

6.1.2 Kamerakalibrierung

Nirgends im Objektmodell des Scannersystems wird vom objektorientierten Design intensiver Gebrauch gemacht als im Falle der Kamerakalibrierungsklassen. Abbildung 6.1 zeigt, wie von der abstrakten Klasse `Calibration` die wiederum abstrakte Klasse `CalibrationPinholeBase` abgeleitet wird. Es wurden auch noch zwei weitere Ableitungen für Kalibrierungen von Kameras mit einfachen Linsensystem realisiert. Die Kalibrierungsklasse `Calibration` enthält hauptsächlich die extrinsischen Orientierungsdaten; die Lochkamera-Basisklasse erweitert diese um die intrinsischen Orientierungsdaten nach dem Lochkameramodell (Kapitel 5.3.1).

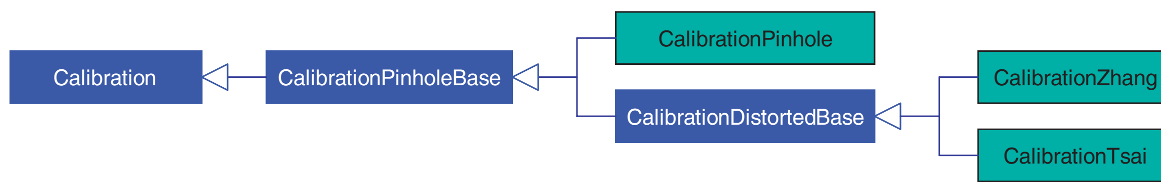

Abbildung 6.1: Objektmodell der realisierten Kalibrierungsobjekte

Für die Konfiguration der extrinsischen Orientierung stehen zwei Ansätze zur Verfügung:

- Angabe der Kameraposition sowie eines anvisierten Punktes im Raum und eines Up-Vektors, der die vertikale Ausrichtung des Bildes bestimmt.
- Angabe der Kameraposition im Raum sowie einer Rotationsachse und eines Rotationswinkels, um den die Kamera im Raum rotiert werden soll.

Da diese beiden Konfigurationsarten ineinander überführbar sind, spielt es keine Rolle, welche Art zur Konfiguration der Kameras verwendet wird. Es können auch immer die aktualisierten Daten beider Konfigurationsarten ausgelesen werden, was dadurch ermöglicht wird, dass bei jeder Änderung eines extrinsischen Parameters die Parameter der anderen Art angepasst werden. Ebenfalls immer auf dem aktuellen Stand gehalten wird die extrinsische Transformationsmatrix T_{extr}.

Die intrinsische Orientierung wird durch die im Kapitel 5.3.1 beschriebenen Parameter Brennweite und Hauptpunktverschiebung sowie die Skalierungsfaktoren definiert. Bei jeder Änderung der intrinsischen Parameter wird automatisch analog zur extrinsischen Transformationsmatrix die intrinsische Transformationsmatrix aktualisiert. Gleichzeitig werden auch die View-Transformationsmatrix $T_{view} = T_{intr} \cdot T_{extr}$, welche der Multiplikation der intrinsischen mit der extrinsischen Orientierungs-Matrix entspricht, sowie ihre Inverse bereitgestellt.

Umrechnungsfunktionen

Jedes Kalibrierungsobjekt stellt zwei Funktionen `worldToImage` und `imageToWorld` zur Verfügung, welche die Umrechnung von Weltkoordinaten nach Bildkoordinaten und umgekehrt ermöglichen. Die Umkehrfunktion `imageToWorld` kann zwar grundsätzlich nicht die richtigen Weltkoordinaten zu den Bildkoordinaten eines abgebildeten Punkts liefern, doch gibt sie einen Vektor zurück, der einen Punkt auf demjenigen Sichtstrahl definiert, auf dem sich auch der abgebildete Punkt befindet. Durch Subtraktion der Kameraposition, die sich ebenfalls auf diesem Sichtstrahl befindet, von diesem Vektor erhält man den Verbindungsvektor, der den Sichtstrahl beschreibt. Die Umrechnungen werden mittels Multiplikation des gegebenen Welt- respektive Bildkoordinatenvektors mit der View-Transformationsmatrix respektive ihrer Inversen realisiert.

Verzerrung

Eine Erweiterung der Lochkamera-Basisklasse ist die Verzerrungs-Basisklasse `CalibrationDistortionBase`. Diese abstrakte Klasse erweitert das Lochkameramodell um Linsenverzerrung und Tiefenschärfe. Die Umrechnungsfunktionen rechnen die gegebenen Koordinaten vom Lochkamera-Modell in ein verzerrtes Bildkoordinatensystem um.

Realisiert wurde die Konvertierung zwischen entzerrten und verzerrten Bildkoordinaten durch die beiden Methoden `distorted` und `undistorted`. Diese beiden Methoden verlangen im Gegensatz zu den Umrechnungsfunktionen als Parameter zentrierte Bildkoordinaten, deren Ursprung im Hauptpunkt ist. Entsprechende Konvertierungsmethoden stehen zur Verfügung, wobei jeweils unterschieden werden muss, ob das Bild, auf dem die Koordinaten basieren, verzerrte oder normale Grösse hat (vergleiche Kapitel 6.3.6).

Die Schwierigkeit bei der Verzerrung liegt darin, dass die Entzerrungsfunktion Potenzen von hohem Grad enthält (Kapitel 5.3.2) und deshalb keine eindeutige Umkehrfunktion existiert. Zur Lösung dieses Problems wird bei jeder Änderung eines intrinsischen Parameters in einer von der Verzerrungs-Basisklasse abgeleiteten Klasse ein Array mit nach dem Newton-Verfahren berechneten Werten erstellt. Die im Array enthaltenen Werte liefern zu einem verzerrten Abstand vom Bildmittelpunkt den entsprechenden entzerrten Bildradius. Die Grösse des Arrays ist auf den für den Viewport benötigten Wertebereich beschränkt. Beim Auslesen von tabellierten Werten erfolgt eine lineare Interpolation zwischen den zwei nächsten Nachbarn des gesuchten Bildradius.

Der realisierte objektorientierte Ansatz einer Distortion-Basisklasse, welche die gesamte Interpolationsfunktionalität besitzt, erlaubte nicht nur eine kompakte Implementierung der Verzerrungsfunktionen nach Tsai und Zhang, sondern ermöglicht durch Angabe einer radialen Verzerrungsfunktion und ihrer ersten Ableitung (wird für das Newton-Verfahren benötigt) weitere vom Bildradius abhängige Kalibrierungsarten beliebigen Grades. Nicht realisierbar sind mit diesem Ansatz hingegen tangentiale Verzerrungen, weil diese nicht nur vom Bildradius, sondern zusätzlich direkt von den Bildkoordinaten abhängen (Kapitel 5.3.2).

6.1.3 Lichtquellen

Lichtquellen besitzen dieselben extrinsischen Parameter wie Kameras. Es wäre daher durchaus denkbar, eine gemeinsame Basisklasse mit den extrinsischen Parametern für Lichtquellen und Kameras zu bilden, wobei erstere Licht abstrahlt und letztere Licht aufnimmt. Im vorliegenden Objektmodell wurde auf eine solche Klasse verzichtet, weil die gemeinsame Basis nur sehr eng ist, was zu einer sehr langen Vererbungslinie führen würde.

6.1.4 Drehteller

Während der Entwicklung des realen Scannersystems werden zu Beginn Experimente mit wenigen Kameras durchgeführt werden. Um dabei ein System mit wesentlich mehr Kameras nachbilden zu können, ist es notwendig, das zu scannende Objekt um die vertikale Achse zu drehen. Ein anderer Ansatz mit demselben Resultat wäre, den Scanner um das Objekt zu rotieren.

Um diese Experimente mit dem Simulationssystem nachbilden zu können, wurde eine Drehteller-Klasse implementiert, welche die virtuelle Szene um einen beliebigen Winkel drehen kann.

6.1.5 Sequenzen

Bei der Benutzung des im Kapitel 1.2 beschriebenen Hardware-Scanners wird es notwendig sein, vom selben Gesicht eine ganze Serie von Bildern mit verschiedenen Beleuchtungsbedingungen und Kameraeinstellungen zu machen.

Scannerprogramme

Als Lösung dieser beiden Anforderungen wurde in der Scanner-API eine programmierbare Schnittstelle zur Konfiguration ganzer Sequenzen von Aufnahmen implementiert. Für jeden einzelnen Sequenzschritt kann jede Kamera getrennt ein- oder ausgeschaltet werden. Zusätzlich können Parameter wie Belichtungszeit, Blendenöffnung und Verstärkungsfaktor am CCD-Ausgang definiert werden. Ebenso können die Eigenschaften aller Lichtquellen eines Scanners getrennt angesteuert werden. Als wichtigster Parameter kann hier die Lichtintensität angegeben werden, mit der die konfigurierte Lampe leuchten soll. Eine letzte Konfigurationsmöglichkeit innerhalb eines Sequenzschrittes ist der Drehwinkel des Drehtellers, welcher für jeden Schritt die Abweichung vom Startwinkel beschreibt.

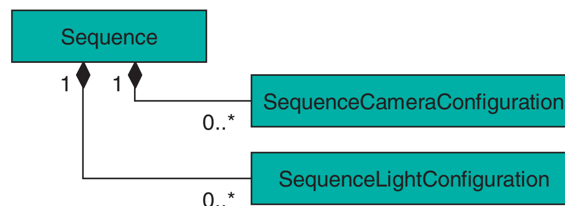

Abbildung 6.2: Eine Sequenz besteht aus je einer Matrix von Kamera- und Lichtquellen-Konfigurationen, welche für jeden Sequenzschritt die jeweilige Hardware konfigurieren

Weil die `Sequence`-Klasse serialisierbar ist, kann ein solches Scannerprogramm komfortabel in einer Datei gespeichert und somit wiederverwendet werden. Die Scanner-API stellt entsprechende Methoden zum Lesen und Speichern einer Sequenz zur Verfügung.

Ablauf

Die Ausführung eines Scannerprogramms wird durch einen endlichen Automaten gelenkt, welcher alle legalen Übergänge von einem Zustand zum nächsten beschreibt (Abbildung 6.3). Bevor eine Sequenz gestartet werden kann, müssen eine Scannerkonfiguration und ein Scannerprogramm definiert werden, was meist durch das Laden von entsprechenden Konfigurationsdateien geschieht. Zusätzlich ist es notwendig, den Client durch die Routine `connect` mit allen zu verwendenden Servern zu verbinden und mit `transmit` die Scannerkonfiguration über das Netzwerk an alle Server zu übertragen.

Sobald diese Schritte erledigt sind, kann eine Sequenz mit der Scanner-Methode `start` gestartet werden. Diese Methode überträgt das vollständige Scannerprogramm auf alle Server und

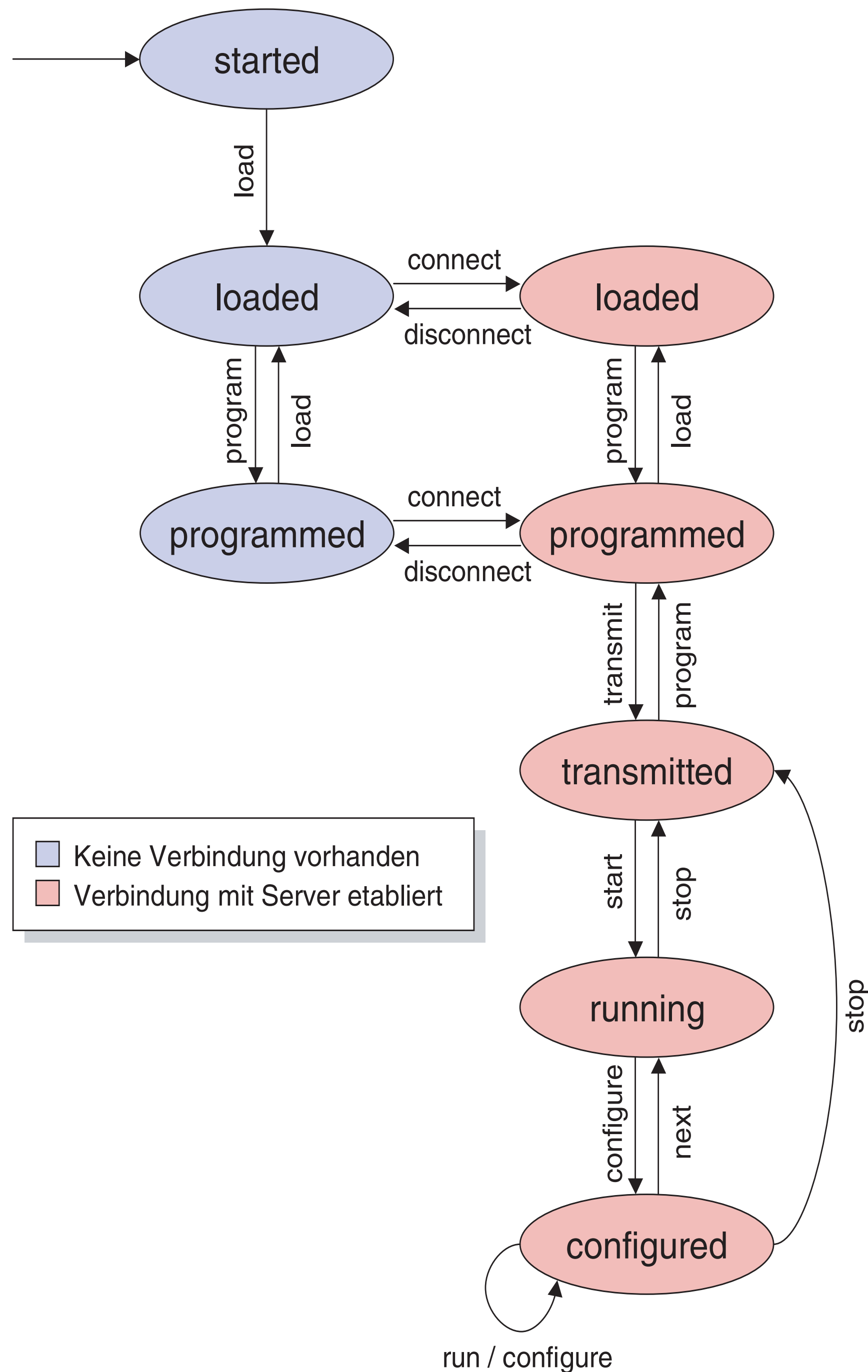

Abbildung 6.3: Zustandsdiagramm des Programmablaufs im Scanner

bereitet die Kamera-Interfaces (Kapitel 6.2.2) für die Aufnahmen vor. Bei der Verwendung von realen CCD-Kameras wäre zum Beispiel denkbar, dass in diesem Schritt die Kameras initialisiert werden. Dabei könnten auch Tests stattfinden, welche dem Client zurückmelden, ob alle Kameras funktionstüchtig sind.

Die eigentliche Abarbeitung des Scannerprogramms erfolgt durch eine Folge von Aufrufen der beiden Scanner-Methoden `configure` und `run`. Die erste Methode konfiguriert jeweils die Kameras mit den entsprechenden Parametern für den aktuellen Sequenzschritt, der zweite Aufruf lässt die aktiven Kameras mit der aktuellen Konfiguration je ein Bild aufnehmen. Der nachfolgende Sequenzschritt wird durch einen Aufruf der Methode `next` erreicht. Zu jedem

Zeitpunkt ist es zusätzlich möglich, eine sich in Abarbeitung befindliche Sequenz durch einen Aufruf von `stop` sofort zu beenden, was im Falle eines aufgetretenen Fehlers automatisch geschieht.

Im noch zu entwickelnden Scannersystem mit CCD-Kameras werden aller Voraussicht nach alle Kameras durch eine Triggerleitung mit dem Rechner, auf dem der Client läuft, verbunden sein. Dies ermöglicht eine gleichzeitige Auslösung aller Hardwareelemente mit einer Genauigkeit, wie sie bei einer reinen Netzwerklösung nicht garantiert werden könnte. Ausserdem wird der Client alle Lichtquellen direkt ansteuern, was bedeutet, dass die Server diese Aufgabe nicht übernehmen werden müssen.

6.2 Server

Weil das Rendering ein äusserst zeitintensiver Vorgang ist und weil die bei CCD-Kameras anfallenden Datenmengen enorm gross sind und deshalb der Vorgang des Übertragens der Bilder von den Kameras in den Speicher eines Rechners einige Zeit benötigt, wurde der Entscheid gefällt, sowohl die Simulation als auch die Ansteuerung der CCD-Kameras in einem Verbund von mehreren Rechnern zu verteilen.

6.2.1 Funktionale Sichtweise

Der Aspekt eines Servers als Kommunikationseinheit wird erst im Kapitel 6.4 behandelt werden. Er stellt aber gleichzeitig auch eine funktionale Einheit dar, welche die Kameras anspricht und somit eine der wichtigsten Aufgaben im ganzen System überhaupt vollbringt. Grundsätzlich ist diese funktionale Einheit sehr einfach gehalten, was im folgenden Abschnitt behandelt wird.

6.2.2 Kamera-Interfaces

Der Server instanziiert beim Lesen einer über das Netzwerk übertragenen Scannerkonfiguration für jede Kamera, die er verwaltet, ein Kamera-Interface. Dieses Interface stellt Methoden für das Ein- und Ausschalten der entsprechenden Kamera, für die Konfiguration derselben sowie das Auslösen einer Aufnahme zur Verfügung. Wann immer der Server einen Auftrag bekommt, der eine solche Aktivität verlangt, spricht er die entsprechende Methode der Kamera-Interfaces an.

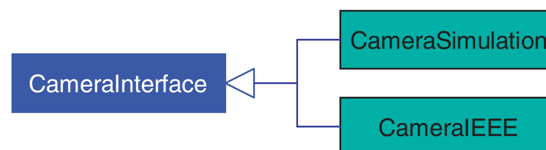

Abbildung 6.4: Realisierung der Kamera-Interfaces für die Simulation und die Hardware-Kameras

Im Objektmodell des Gesichtsscanners ist es bis zu dieser Stelle nicht notwendig zu wissen, ob schlussendlich die Bilder vom Simulator oder von einer realen Kamera stammen. Der Server erstellt die Kamera-Interfaces anhand eines Modus-Flags (ist zur Laufzeit des Servers durch eine entsprechende Meldung änderbar) entweder von der Klasse `CameraSimulation` oder `CameraIEEE`, welche beide wie in Abbildung 6.4 von der Klasse `CameraInterface` abgeleitet sind. Diese Objekte sprechen dann beim Aufruf ihrer Methoden entsprechend ihrer Abstammung entweder den Simulator oder über Hardwaretreiber die Kameras an. Nach der

Instanziierung braucht also der Server nicht mehr zu wissen, in welchem Modus er gerade arbeitet, weshalb er auch das Modus-Flag nicht mehr betrachtet.

6.3 Simulator

Kamera-Interfaces der Klasse `CameraSimulation` sprechen die Simulator-Instanz des Servers an. Eine der ersten Design-Entscheidungen dieses Systems überhaupt war, Alias für das Rendering der simulierten Bilder zu verwenden. Alias ist ein grösseres Programpaket das aus einem Modeller und mehreren Renderern und Raytracern besteht. Speziell geeignet für die hier besprochene Simulation ist Alias, weil die Renderer als Kommandozeilenprogramme angesprochen werden können, was es gleichzeitig möglich macht, die Programme aus einer Applikation heraus zu starten.

Man beachte, dass durch die Verwendung von Alias die Verwendung des Simulationssystems auf das Betriebssystem IRIX von SGI beschränkt wird, denn am Computer Graphics Lab an der ETH in Zürich sind nur solche Lizenzen vorhanden.

6.3.1 Ansteuerung von Alias

Wie im Anhang D ausführlicher beschrieben, wird Alias über sogenannte SDL (Scene Description Language) Dateien angesteuert. Der Simulator erstellt für jedes zu rendernde Bild eine solche Datei zur Definition der darzustellenden Szene und insbesondere aller Parameter für die das Bild aufnehmende Kamera und die aktiven Lichtquellen.

Um das Einbinden von Modell- und Patchdateien (vergleiche Anhänge C und D) zu vereinfachen, werden die konfigurierten Dateienamen mit einer `#include`-Anweisung eingebunden. Leider kann Alias solche Anweisungen nicht interpretieren, sondern verlangt eine einzige Steuerdatei, welche oft einen sehr grossen Umfang hat. Aus diesem Grund wird die vom Simulator erstellte Datei vor dem Aufruf des Renderers durch den C-Präprozessor bearbeitet, welcher alle eingebundenen Dateien in einer Zieldatei zusammenführt und zusätzlich allfällige Definitionen ersetzt.

6.3.2 Allgemeine Parameter

Neben den in den nächsten Abschnitten besprochenen Parametern für die Kamera und die Lichtquellen, bietet Alias noch einige Einstellungsmöglichkeiten von allgemeinem Charakter, welche den Renderer betreffen. In der folgenden Liste werden die wichtigsten davon aufgeführt.

- mit `image_format` kann das Ausgabeformat von Alias bestimmt werden. Im Simulator wird hier TIFF verwendet, weil die Bibliotheken für die Weiterverarbeitung zur Zeit nur dieses Bildformat verarbeiten können.
- `resolution` gibt die Bildauflösung des zu rendernden Bildes an.
- `xleft` und `xright` bestimmen den Bereich, in dem sich die horizontalen Bildkoordinaten bewegen.
- `ylow` und `yhigh` bestimmen den Bereich, in dem sich die vertikalen Bildkoordinaten bewegen.

6.3.3 Kamera-Parameter

In der MODEL-Sektion einer SDL-Datei werden neben den Lichtquellen und der darzustellenden Szene die Kameras instanziiert. Alias erlaubt die Konfiguration von diversen Kamera-Parametern, welche grösstenteils in den Kalibrierungsobjekten definiert sind:

- active ist immer ON, denn es werden nur aktive Kameras exportiert, inaktive Kameras werden durch den Simulator nicht behandelt.
- aov entspricht dem Öffnungswinkel des Frustums. Bei Kameras mit Linsenverzerrung ist dieser Winkel grösser als der konfigurierte, weil zur Nachbearbeitung der Verzerrung ein grösseres Bild benötigt wird.
- eye bestimmt den Punkt in der Szene, wo sich die Kamera befindet.
- far ist die hintere Clipping-Ebene.
- die Brennweite wird mit focal_length konfiguriert. Weil die Angabe in Millimetern zu erfolgen hat, wird der Meter-Wert aus der Kalibrierung entsprechend umgewandelt. Zusätzlich muss der Parameter units_to_feet angepasst werden, damit Alias nicht mit Fuss rechnet.
- motion_blur wird auf null gesetzt, weil dieser Effekt in der Simulation des Gesichtsscanners unerwünscht ist.
- near ist die vordere Clipping-Ebene.
- perspective wird immer TRUE gesetzt, weil jede realistische Kamera eine perspektivische Abbildung erzeugt.
- mit dem Parameter pix wird der Dateiname des zu erzeugenden Bildes angegeben.
- up bestimmt die vertikale Ausrichtung der Kamera.
- view gibt den Punkt in der Szene an, den die Kamera anvisiert.

Für das Rendering von Bildern für Kameras, deren Kalibrierung von CalibrationDistortedBase abgeleitet ist, werden noch weitere Parameter benötigt:

- vorausgesetzt, dass eine Fokaldistanz konfiguriert wurde, wird der Parameter auto_focus auf OFF gesetzt, damit Alias auf die gewünschte Distanz fokussiert.
- depth_of_field muss TRUE gesetzt werden, um Alias zur Berechnung von Bildern mit Tiefenschärfe zu bringen. Bei der Lochkamera wird die Tiefenschärfe ausgeschaltet.
- mit f_stop kann die gewünschte Blendenzahl angegeben werden.
- über focal_distance wird die Distanz von der Kamera angegeben, wo sich das fokussierte Objekt befindet. Wird dieser Wert auf null gesetzt, so wird Alias veranlasst, automatisch zu fokussieren.

Weitere Konfigurationsparameter wurden auf den Standardwerten belassen, da die damit erzielbaren Effekte nicht erwünscht oder durch die gegebenen Werte befriedigend abgedeckt sind.

6.3.4 Lichtquellen

Die Lichtquellen müssen in SDL-Dateien in der ersten, sogenannten Definitions-Sektion definiert werden. Dabei steuert der Simulator die folgenden Parameter:

- active ist immer ON, weil der Simulator nur aktive Lichtquellen bearbeitet.
- wenn eine ambiente Lichtquelle verwendet wird (Parameter model) so kann mit ambient_shade der Anteil des ambienten Lichts angegeben werden.
- der Parameter color erlaubt die Konfiguration der Farbe der Lichtquelle.
- decay gibt an, wie schnell die Lichtintensität mit dem Abstand von der Lichtquelle abnimmt. Dieser Wert ist immer auf 2 gesetzt, wodurch die Lichtstärke mit dem Abstand im Quadrat abnimmt.
- fog_type ist immer OFF, weil kein Nebel dargestellt werden soll.

- über `glow_type` kann bestimmt werden, ob und wie Lichtquellen in der Szene dargestellt werden sollen.
- `halo_type` ist immer `OFF`, da wir kein Halo sehen möchten.
- `intensity` beschreibt die Lichtintensität der Lampe. Über diesen Parameter wird in der Simulation nicht nur die Lichtquelle gesteuert, sondern indirekt durch die Multiplikation dieses Wertes mit der Belichtungszeit und dem Verstärkungsfaktor aus der Kamerakonfiguration auch die Empfindlichkeit der Kamera, welche selber keine entsprechende Einstellung besitzt.
- `model` ist ein konfigurierbarer Parameter und beschreibt die Art der Lichtquelle. Standardmässig wird eine Punktlichtquelle verwendet.
- `penumbra` definiert den Winkel des Schattenrandes, der als Halbschatten gerendert werden soll. Dieser Parameter steht nur bei gerichteten Punktlichtquellen (Spotlight) zur Verfügung.
- `shadow` ist true gesetzt, um realistische Schatten zu erhalten.
- `shadow_color` wird immer auf schwarz gesetzt, denn Schatten sollen keine Farbe entwickeln.

Wie bei der Kamerakonfiguration werden auch bei den Lichtquellen einige Parameter auf den Standardwerten belassen, da die damit erzielbaren Effekte nicht erwünscht sind. Darunter sind auch einige in [3] nicht dokumentierte Einstellungen, welche in einer mit Alias ausgegebenen und anschliessend analysierten SDL-Datei aufgeführt waren.

6.3.5 High Dynamic Range Imaging

Für die Realisierbarkeit von CCD-Effekten wie Blooming, welche durch zu lange Belichtungszeiten und dadurch entstehende Überbelichtung verursacht werden, wurde in Anlehnung an die im Kapitel 2.2 besprochene Arbeit [8] die Simulation von Bildern mit hohem dynamischem Bereich implementiert. Voraussetzung für solche Bilder ist ein mehrmaliges Rendern derselben Szene mit unterschiedlicher Belichtungszeit. Weil, wie oben erläutert, in Alias die Belichtungszeit über die Lichtquellenintensität gesteuert werden muss, wird folgerichtig dieser Parameter für die einzelnen Durchgänge variiert.

Nach dem Rendern eines ersten Bildes wird jedes Pixel betrachtet und entschieden, ob seine Intensität zu hoch ist, um richtig belichtet zu sein. Dies ist genau dann der Fall, wenn Pixel mit maximal möglicher Farbintensität gefunden werden (eventuell mit Toleranz), denn dann ist es nicht möglich zu sagen, ob deren Intensität korrekt ist, weil Alias Punkte mit wesentlich stärkerer Belichtung auf den für Standard-RGB-Werte nutzbaren 8bit-Bereich normieren muss. Dadurch wird in den allermeisten Fällen bei solchen Bildpunkten Information verloren gegangen sein.

Um diese verlorene Farbintensitäts-Information doch noch gewinnen zu können, wird jeweils ein nächster Rendering-Vorgang gestartet, bei welchem die Lichtstärke aller Lichtquellen halbiert wird, bis keine überbelichteten Pixel mehr gefunden werden können. Damit diese zusätzliche Farbinformation der nachträglich gerenderten Bilder mit dem ersten Bild kombiniert werden kann, muss aber die Farbtiefe des Bildes angepasst werden. Dies entspricht dem Vorgehen beim *High Dynamic Range Imaging*

In Abbildung 6.5 wird als Beispiel die Berechnung des grossen Wertebereichs eines einzelnen Pixels demonstriert. Vom ersten bis zum dritten Durchgang ist der Bildpunkt jeweils überbelichtet, was durch die von der Farbtiefe abhängige maximal mögliche Farbintensität festgestellt werden kann. Erst nach dem vierten Rendering-Durchgang mit einem Achtel der ursprünglich

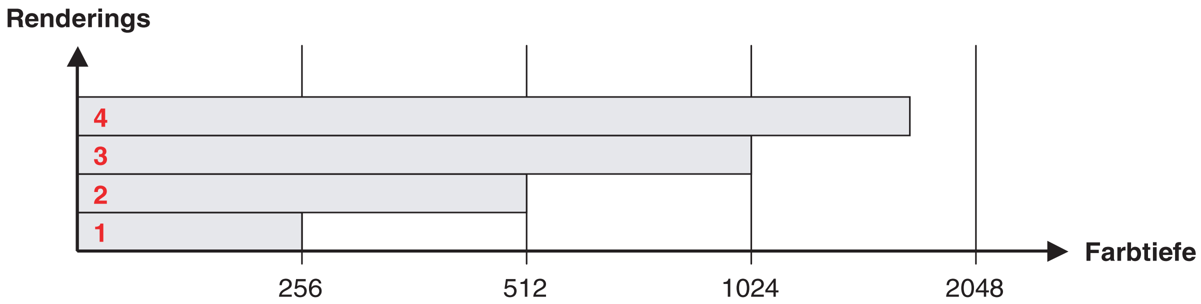

Abbildung 6.5: Erst im vierten Rendering-Durchlauf kann die effektive Farbintensität des mit logarithmischer Farbtiefe dargestellten Bildpunktes exakt bestimmt werden

konfigurierten Belichtungszeit wird von Alias ein nützlicher 8bit-Intensitätswert 218 zurückgegeben. Dieser Intensitätswert I muss jetzt nach folgender Formel 6.1 in die beim vierten Durchgang $(n = 4)$ mögliche Farbtiefe von 2048 skaliert werden, damit er sich von denjenigen früherer Renderings unterscheiden und den korrekten Wert 1744 darstellen kann:

$$I' = 2^{n-1} \cdot I \qquad \forall n > 0 \tag{6.1}$$

Damit die so entstehenden Bildpunkte mit einem grossen Wertebereich gespeichert werden können, wird die Bildtiefe für jede Grundfarbe auf 16bit erhöht. Es sind also RGB-Werte von 0 bis 65535 erlaubt. Durch diese Bildtiefe wird gleichzeitig auch die maximal mögliche Anzahl Renderings für die Berechnung des grossen Wertebereichs auf acht Durchgänge limitiert. Sämtliche folgende Nachbearbeitungsschritte basieren auf solchen 16bit-Bildern. Erst vor dem Speichern in eine Datei wird das Bild auf 8bit zurückkonvertiert.

6.3.6 Einfügen von Artefakten

Ein wichtiger Teil der Simulation ist, die gerenderten Bilder durch das Einfügen von Kamera-Artefakten realistischer erscheinen zu lassen. Sämtliche dafür benutzte Morpher-Klassen werden von der Bibliothek `aberration` zur Verfügung gestellt.

Morpher

Für das Einfügen von Artefakten in gerenderte Bilder wurde eine abstrakte Morpher-Klasse erstellt. Neben der Definition der engen Morpher-Schnittstelle mit der Funktion `process` stellt sie zwei statische Methoden für die Umwandlung von 8bit-Bildern in solche mit 16bit pro Pixel und zurück zur Verfügung. Letztere werden benötigt, um Bilder mit grossem Wertebereich darstellen zu können (Kapitel 6.3.5).

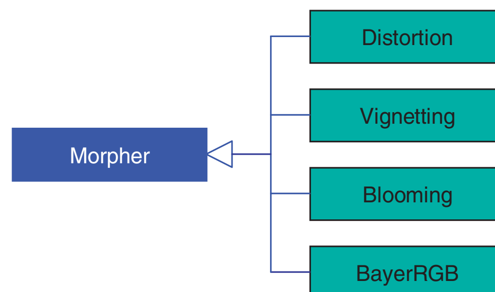

Abbildung 6.6: Abstrakte Morpher-Klasse und die davon abgeleiteten Artefakte-Klassen

Die davon abgeleiteten Artefakte-Klassen arbeiten somit auf Bildern mit grossem Wertebereich, weil dieser einige Artefakte erst ermöglicht (zum Beispiel Blooming). Die Methode `process` nimmt ein solches Bild als Argument entgegen und bearbeitet es entsprechend des unterstützten Artefakts. Anschliessend gibt die Methode das veränderte Bild zurück. Es muss dabei unbedingt beachtet werden, dass die zurückgegebene Bildinstanz unter Umständen nicht mit derjenigen des Arguments übereinstimmt. Ein Beispiel, wo dies nicht der Fall ist, ist die Verzerrungs-Morpherklasse, welche den durch das übergebene Bild belegten Speicher freigibt und eine neue Bildinstanz mit unterschiedlicher Bildgrösse generiert. Allfälligerweise noch vorhandene Zeiger auf die übergebene Bildinstanz werden folglich ungültig.

Im folgenden werden die unterstützten Artefakte-Klassen in der Reihenfolge besprochen, wie sie im Simulator angewandt werden.

Verzerrung

Wie bereits im Abschnitt 6.3.3 erwähnt, ist es für das Einfügen von Verzerrung in ein Bild notwendig, ein Bild mit einem grösseren Viewport und somit auch einem grösseren Blickwinkel als konfiguriert zu rendern. Begründet werden kann dies damit, dass durch diesen Effekt das ganze Bildfeld verzerrt wird.

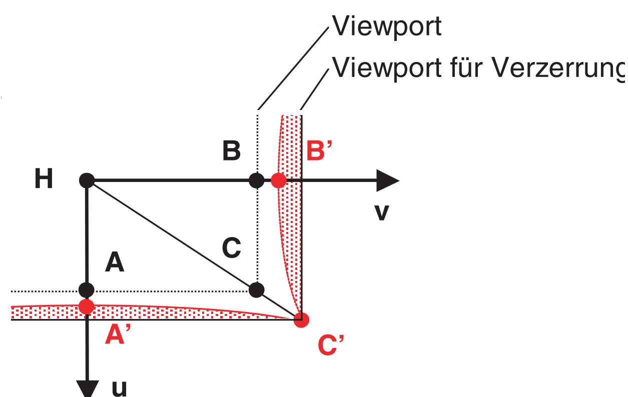

Abbildung 6.7: Vergleich der Viewport-Grösse aus der Konfiguration und derjenigen, welche für das Einfügen von Verzerrung benötigt wird

Ein Beispiel anhand der Abbildung 6.7 veranschaulicht obige Aussage: Der Bildpunkt C' wird in einem verzerrten Bild an der Stelle C abgebildet, also radial zu nahe beim Hauptpunkt. Dasselbe gilt auch für die Bildpunkte A' und B', und natürlich auch alle anderen, über das gesamte Bild verteilten Punkte. Wird ein Rastermuster so verzerrt, so entsteht der Eindruck eines Kissens (Abbildung 7.1), weshalb diese Variante der Verzerrung Kissen-Effekt genannt wird. Gerade umgekehrt ist es beim Fass-Effekt, wo die Bildpunkte radial zu weit entfernt vom Hauptpunkt abgebildet werden (Kapitel 4.1.1).

Die Grösse des zu rendernden Bildes wird also durch die Verzerrung beeinflusst. Die Klasse `CalibrationDistortedBase` stellt eine Methode zur Verfügung, welche den für die Berechnung der Verzerrung benötigten Viewport zurückgibt. Der Simulator fragt diese Grösse ab und lässt Alias ein Bild der entsprechenden Grösse rendern.

Die Artefaktmethode `process` der Morpher-Klasse `Distortion` erstellt zuerst ein Bild der konfigurierten Grösse. Anschliessend geht eine Schleife durch dieses Bild und berechnet mit der Verzerrungsfunktion die entsprechenden Koordinaten im übergebenen, unverzerrten Bild. Weil diese berechnete Koordinate meistens keiner Ganzzahl entspricht, werden die Farben der

im gerenderten Bild gefundenen Nachbarpunkte interpoliert. Zusätzlich ist auf Wunsch vier- und neunfaches Oversampling verfügbar.

Vignetting

Ebenso wie die Verzerrung existiert der Effekt des Vignettings (Kapitel 4.1.5) nur bei Kameras mit einer Kalibrierung, die von `CalibrationDistortedBase` abgeleitet ist, denn für das Auftreten dieses Effekts müssen Linsen sowie eine Apertur in der Kameraoptik vorhanden sein. Er ist abhängig vom Winkel θ zwischen der Hauptachse $\mathbf{m}$ der Kamera und dem Bildstrahl $\mathbf{b}$. Dieser Winkel kann mit folgender Formel berechnet werden:

$$\theta = \operatorname{acos}\left(\frac{\mathbf{m} \cdot \mathbf{b}}{|\mathbf{m}| \cdot |\mathbf{b}|}\right) \tag{6.2}$$

wobei $\mathbf{m}$ und $\mathbf{b}$ für einen Bildpunkt (x, y) definiert sind als

$$\mathbf{m} = \mathbf{lookAt} - \mathbf{position} \text{ und } \mathbf{b} = imageToWorld(x, y) - \mathbf{position} \tag{6.3}$$

Die Vektoren $\mathbf{lookAt}$, $\mathbf{position}$ und die Funktion `imageToWorld` stammen aus einer von `CalibrationDistortedBase` abgeleiteten Kamera-Kalibrierung. Der Faktor für die Abschwächung der Strahlung gegenüber derjenigen beim Hauptpunkt wurde im Kapitel 4.1.5 beschrieben als

$$E' = E_0 \cdot k_V \cdot \cos(\theta)^4 \text{ mit } k_V = \frac{A}{f^2} \tag{6.4}$$

wobei A die Aperturfläche und f die Brennweite der Kamera ist.

Dieser Faktor muss nun auf einen RGB-Farbwert übertragen werden. Zum besseren Verständnis dieser Intensitätsanpassung kann der Farbwert zuerst nach [14, 4] ins HSV-Farbmodell umgerechnet werden. Dieses in Abbildung 6.8 gezeigte Modell unterstützt im Gegensatz zum RGB-Modell mit dem Value-Parameter V direkt die gesuchte Lichtintensität. Eine Abschwächung derselben entspricht also in Abbildung 6.8 einer Verschiebung eines Farbpunktes nach unten. Damit ergibt sich folgende Gleichung für die HSV-Werte im Zielbild:

$$\begin{aligned} H' &= H \\ S' &= S \\ V' &= V \cdot k_V \cdot \cos(\theta)^4 \end{aligned} \tag{6.5}$$

Dieser Farbwert würde anschliessend nach [14, 4] wieder ins RGB-Farbmodell zurückgerechnet und im Bild gespeichert.

Weil dabei offensichtlich der Farbton nicht geändert wird (H und S bleiben konstant), kann dieser Effekt auch durch eine Skalierung aller RGB-Werte mit dem Vignetting-Faktor realisiert werden:

$$\begin{aligned} R' &= s_V \cdot R \\ G' &= s_V \cdot G \text{ mit } s_V = k_V \cdot \cos(\theta)^4 \\ B' &= s_V \cdot B \end{aligned} \tag{6.6}$$

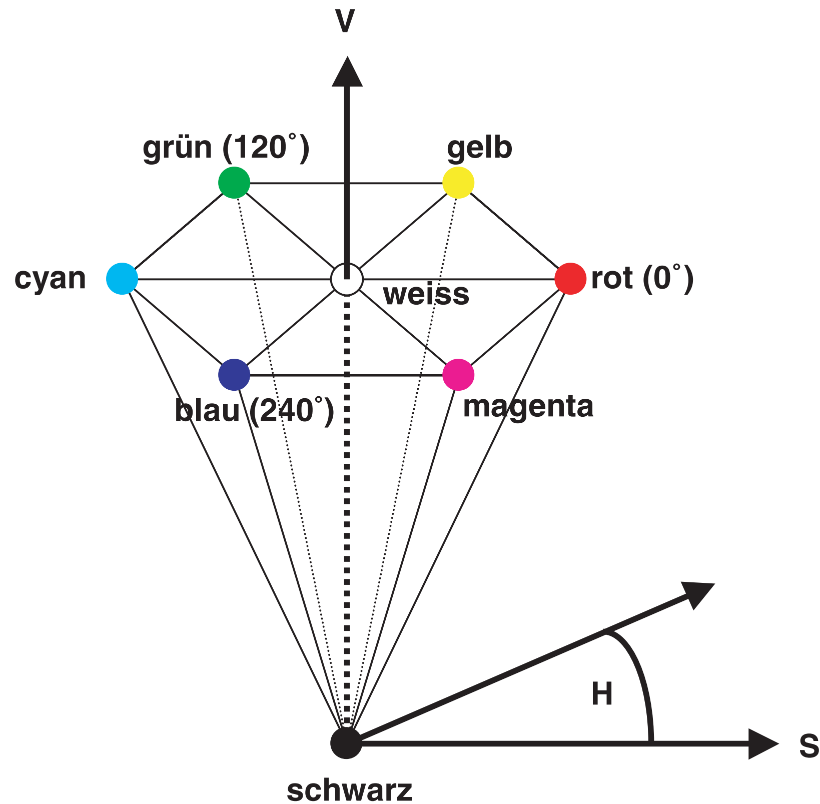

Abbildung 6.8: HSV-Farbmodell, definiert Farben als Farbton H (0..360°), Sättigung S (0..1) und Helligkeit (0..1) nach [5].

Bayer-Pattern

Nachdem alle optischen Artefakte auf das Bild angewandt worden sind, werden im Simulator noch die im CCD-Chip auftretenden Effekte berücksichtigt. Falls in der Kalibrierung so konfiguriert, wird als erstes das Bayer-Pattern (Kapitel 4.2.1) auf das Bild angewandt. Das bedeutet, dass für jedes Pixel im Bild immer zwei der RGB-Werte auf den Wert der dritten Farbe gesetzt werden. Wird also beispielsweise ein grüner Bildpunkt bearbeitet, so werden die roten und blauen Farbwerte der grünen Intensität gleichgesetzt. Daraus resultiert ein Grauwertbild. Dieser Effekt wird normalerweise nur verwendet, wenn eine Ein-Chip-Kamera simuliert werden soll.

Welche Farbwerte angepasst werden müssen, kann durch Evaluation des letzten Bits in den Koordinaten des Bildpunktes einfach berechnet werden (Abbildung 6.9). Aus Kompatibilitätsgründen mit verschiedenen Kameraherstellern wurden zwei Varianten des Bayer-Pattern implementiert, welche sich durch eine horizontale Verschiebung um eine Pixelposition unterscheiden.

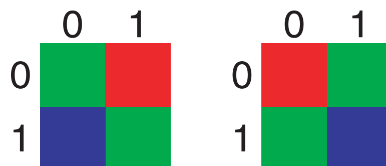

Abbildung 6.9: Vier Pixel eines CCD-Chips mit Bayer-Muster-Anordnung für zwei Varianten des RGB-Modells, sowie Indizes, welche durch Evaluation des letzten Bits der Bildkoordinaten berechnet werden können.

Blooming

Voraussetzung für das zum Schluss angewandte Blooming sind Pixel, welche zu stark belichtet wurden. Dies ist genau dann der Fall, wenn in einem Bild mit grossem Wertebereich Pixel vorhanden sind, deren Farbintensität grösser als der von einem 8bit-RGBA-Bild fassbare Bereich ist. Physikalisch bedeutet das, dass die Kapazität eines CCD-Elements nicht alle Ladungen fassen kann, weshalb ein Teil der Elektronen auf benachbarte Elemente überspringen (Kapitel 4.2.2).

Insgesamt werden in der Implementierung zwei unterschiedliche Arten unterstützt, wie die Elektronen auf Nachbar-Elemente überspringen können:

- auf CCD-Elemente in den Nachbarspalten.
- auf CCD-Elemente entlang der Ausleserichtung des Chips.

In der Implementierung wird das Überspringen auf die Nachbarspalten zuerst eingefügt. Unter

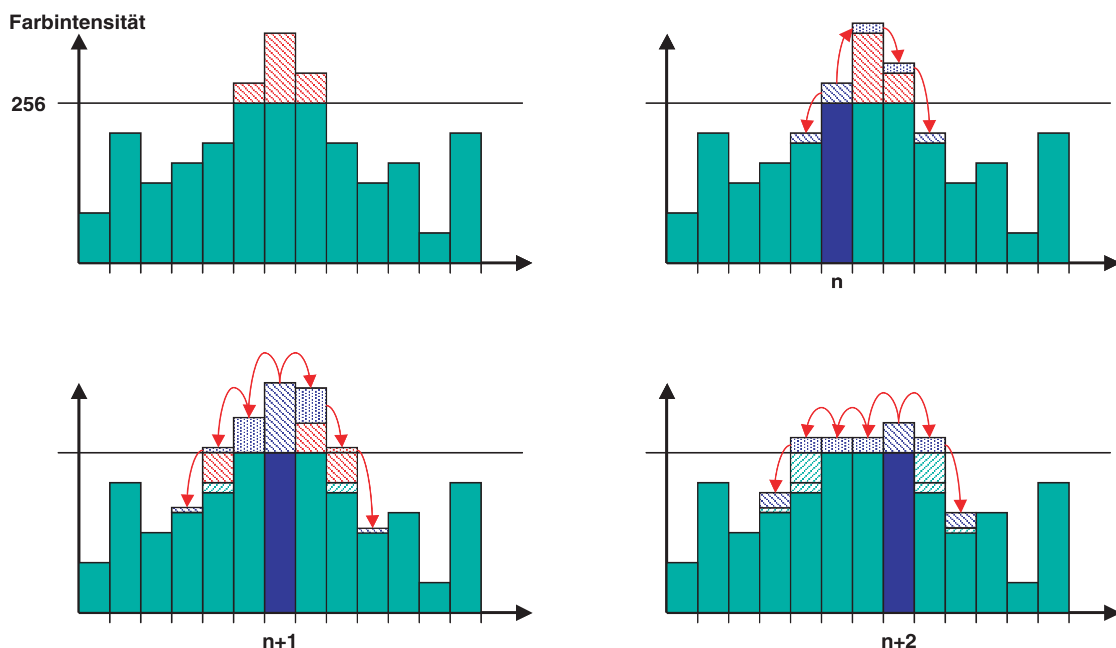

Abbildung 6.10: Darstellung der Applikation von Blooming bei überbelichteten Bildpunkten. Bearbeitet wird jeweils das blau dargestellte Pixel

der Annahme, dass dieser Effekt seltener auftritt als Blooming entlang der Ausleserichtung, wurde eine Modellierung mit einer Gauss'schen Verteilungskurve realisiert:

$$I'_{\Delta}(I_{\Delta}, x) = I_{\Delta} \cdot e^{\sigma^2 x^2} \tag{6.7}$$

Diese Funktion gibt für einen Bildpunkt im Abstand x vom betrachteten Pixel an, welche zusätzliche Intensität er von jenem bekommt, weil Elektronen über die Grenzen der Auslesespalten springen. Diese so verteilte Intensität ist derjenige Anteil der Gesamtintensität des gerenderten Pixels, welcher durch die Begrenzung der Farbtiefe nicht gespeichert werden kann. σ ist eine empirisch gesuchte Konstante.

Abbildung 6.10 demonstriert, wie sich Blooming in der Ausleserichtung des CCDs auswirkt. Die Darstellung links oben zeigt die Ausgangslage nach dem Rendern und dem Einfügen von Intensitätsanteilen, welche von den Nachbarpixeln kommen. Drei Pixel in der CCD-Zeile sind

überbelichtet und verursachen somit diesen unerwünschten Effekt. Die restlichen Bilder zeigen, wie bei jedem überbelichteten Bildpunkt in der Abarbeitungsreihenfolge von links nach rechts der Anteil der Lichtintensität über der durch das schlussendlich gewünschte 8bit-Bild definierten Farbtiefe je zur Hälfte auf die Nachbarpixel verteilt wird. Findet dieser Anteil beim Nachbarpixel keinen oder nicht genügend Platz, so wird er oder entsprechend der überflüssige Teil davon weiter geschoben, bis entweder alle Anteile verteilt werden konnten, oder der Bildrand erreicht ist. Beim Bildrand gehen die überflüssigen Intensitätsanteile verloren. Im Beispiel resultieren am Schluss fünf Pixel mit maximaler Intensität sowie zwei weitere, deren Intensität zugenommen hat. Insgesamt sind vier Bildpunkte vom Effekt betroffen.

6.4 Kommunikation

Damit die im Netzwerk als Client auftretende Scanner API die Aufträge für die Kamera-Interfaces an die Kameraserver verteilen kann, wird eine Kommunikationsschicht benötigt. Die folgenden Abschnitte beschreiben die Realisierung des verwendeten, auf Meldungen basierenden Netzwerprotokolls. Die für die Kommunikation implementierten Klassen werden von der Bibliothek `connect` zur Verfügung gestellt.

6.4.1 Objektserialisierung

Für die Konfiguration der Scanner API und der Kameraserver galt es zu Beginn dieser Diplomarbeit, ein Konzept zur Speicherung und zum Auslesen von Konfigurationsdateien sowie für die Übertragung derselben Daten über das Netzwerk zu den Servern zu entwickeln. Voraussetzung war, dass die Dateien von Menschen editier- und lesbar sein müssen. Ein binäres Format kam deshalb von Beginn weg nicht in Frage. Als mögliche Formate wurden XML oder das in Windows verwendete INI-Dateiformat in Betracht gezogen. XML unterstützt im Gegensatz zu den INI-Dateien zwar den objektorientierten Gedanken durch mögliche Schachtelung, doch erschien die Syntax schlecht lesbar und schreibintensiv bei manueller Erstellung. Mangels der Unterstützung der Darstellung von Objekten wurde auch das INI-Format verworfen. Die Entscheidung fiel schliesslich auf ein eigenes, an die C++ Syntax angelehntes Format, welches detailliert im Anhang C.1 besprochen wird.

Für das Erreichen einer Objektserialisierung wurde eine Basisklasse `Dumpable` mit einer `read` und einer `dump` Methode entwickelt. Von dieser Klasse werden alle konfigurierbaren Klassen abgeleitet. Die einzelnen Klassen erweitern jeweils die vererbten Serialisierungs-Routinen um die Möglichkeit, ihre eigenen Parameter lesen und schreiben zu können. Wann immer notwendig wird der Aufruf in der Vererbungslinie nach oben weitergeleitet, damit auch die vererbten Parameter gelesen und geschrieben werden können.

Die Steuerung der Objektserialisierung erfolgt in einer Parser-Klasse. Sie stellt Methoden für das Lesen und Schreiben von ganzen Dumpable-Klassen sowie der verschiedensten Datentypen zur Verfügung. Gleichzeitig prüft sie auch die syntaktische Korrektheit der geparsten Daten respektive schreibt die Daten in der durch die Syntax vorgegebenen Form.

Der Parser arbeitet mit Tokens, wobei jede syntaktische Einheit ein solches darstellt. Wird zum Beispiel eine Wertzuweisung gelesen, so liest der Parser erst das Token Variablenname, anschliessend das Gleichheitszeichen, gefolgt vom Wert und dem abschliessenden Strichpunkt. Jeder gelesene Variablenname wird an die zu konfigurierende Klasse übergeben, welche dann entsprechend dem erwarteten Datentyp die passende Leseroutine des Parsers aufruft.

6.4.2 Netzwerk

Die Übertragung von Daten über das Netzwerk erfolgt meldungsbasiert über Sockets, also unter Verwendung des TCP/IP-Protokolls [17]. Die unterstützten Meldungen haben einen einfachen, dadurch aber auch flexibel einsetzbaren Aufbau:

```
int32 messageNumber
int32 messageSize
char  messageData[messageSize]
```

Die Meldungsnummer beschreibt die Informationen, welche die Meldung trägt. Sie kann entweder einen Befehl repräsentieren oder auch einen Ergebniswert bei einer Rückmeldung. Zusätzlich muss immer die Grösse der Meldungsdaten angegeben werden, welche auf null gesetzt wird, falls keine Nutzdaten mitgeführt werden. Andernfalls gibt diese Grösse die Anzahl Bytes der nachfolgenden Meldungsdaten an. Somit hat jede Meldung einen Umfang von mindestens acht Bytes.

Zusätzlich zu den Nutzdaten schicken einige Meldungen im Anschluss an eine Message ein serialisiertes Objekt über das Netzwerk. In diesem Fall soll die Grösse der gesendeten Daten nicht angegeben werden, denn die empfangende Parser-Instanz muss einfach die ganze zu konfigurierende Klasse einlesen, bis das syntaktische Ende empfangen wird. Ausserdem wird der Parser nicht automatisch durch das Meldungsobjekt gestartet (vergleiche nächster Absatz). Beim Schreiben wird automatisch darauf geachtet, dass speziell am Ende der Objektserialisierung keine überflüssigen Zeichen ausgegeben werden, womit sich der Empfänger darauf verlassen kann, dass nach der terminierenden geschweiften Klammer (Anhang C.7) keine weiteren Daten vom Socket gelesen werden können.

Das Senden und Empfangen von Meldungen erledigen die Instanzen der Klasse `Message` selbständig, wenn sie von einem Client- oder Server-Objekt den entsprechenden Aufruf zusammen mit dem zu verwendenden Socket bekommen, welcher die Verbindung darstellt. Beim Senden schreibt sich die Meldung auf den gegebenen Socket, beim Empfangen werden analog die benötigten Daten vom Socket gelesen. Dadurch müssen weder Client noch Server das Meldungsprotokoll kennen.

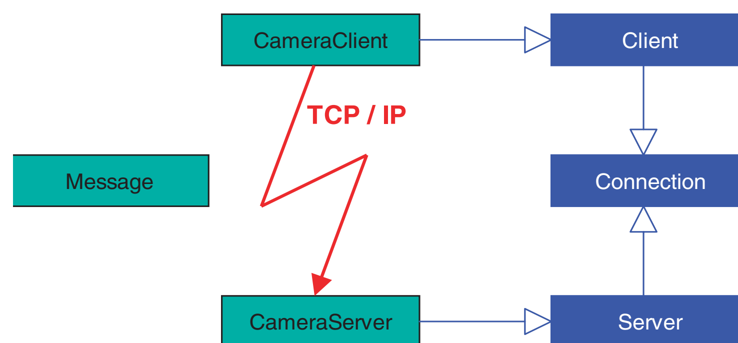

Abbildung 6.11: Sowohl der Kameraclient als auch der Kameraserver stammen von einer Connection-Klasse ab. Beide kommunizieren über ein TCP/IP-Netzwerk auf Meldungsebene

Die Client/Server-Umgebung wurde so aufgebaut, dass ein Client eine Verbindung zu einem Server aufnehmen und ihm Meldungen schicken respektive von dort solche empfangen kann. Der Server empfängt ankommende Meldungen und gibt sie an eine Instanz der im folgenden Abschnitt beschriebenen Handler-Klasse zur Bearbeitung weiter.

6.4.3 Handler

Jede Server-Instanz besitzt ein Handler-Objekt, dem alle empfangenen Meldungen sofort zur Bearbeitung übergeben werden. Diese Objekte repräsentieren also die Server-Logik, indem sie den Inhalt von Meldungen analysieren und die entsprechenden Funktionen auslösen. In der vorliegenden Implementierung werden jeweils die Clients über den Erfolg oder allenfalls bei der Bearbeitung aufgetretene Fehler informiert, indem jede Meldung durch den Handler beantwortet wird.

Während der Initialisierung des Handlers, was beim Start des Servers geschieht, wird auch die benötigte Simulator-Instanz aufgebaut (Kapitel 6.3). Die Erstellung der Kamera-Interfaces geschieht erst, wenn die Scanner-Konfiguration übertragen wird.