

Verteiltes Rendering: Flexible Spezifikation und Konfiguration mittels Multimedia-Middleware

Alexander Löffler, Michael Replinger[†] und Philipp Slusallek[†]

Lehrstuhl für Computergraphik, Universität des Saarlandes

[†]auch DFKI, Forschungsbereich Agenten und Simulierte Realität

Im Stadtwald, Campus E1.1, 66123 Saarbrücken

Tel: 0681-302-3831, Fax: 0681-302-3843

E-Mail: {loeffler, replinger, slusallek}@cs.uni-sb.de

Zusammenfassung

Selbst die Rendering-Performanz der schnellsten Graphikhardware reicht heute nicht aus, auch sehr große Display-Konfigurationen zu betreiben oder im Server-Based Rendering viele kooperative Nutzer einer Szene gleichzeitig mit einem interaktiven Videostrom ihrer Ansicht zu versorgen. Andererseits bieten die meisten der bestehenden Frameworks für verteiltes Rendering nicht die Flexibilität die vielen verschiedenen Anwendungsszenarien abdecken zu können.

Die hier vorgestellte Kombination eines flexiblen Rendering-Systems für Rasterisierung und Echtzeit-Ray-Tracing (URay) mit einer verteilten Multimedia-Middleware (NMM) bietet jetzt erstmals eine einfach und sehr flexible Möglichkeit, einen großen Teil des Anwendungsspektrums abdecken zu können.

Wir stellen das Framework selbst vor und diskutieren wie auch sehr komplexe Konfigurationen unterschiedlichster Szenarien mittels einer einfachen Sprache spezifiziert und auf eine flexible Verteilungsarchitektur abgebildet werden können.

Schlüsselwörter

Verteiltes Rendering, Flexible Konfiguration, Framework, Rasterisierung und Ray-Tracing

1 Einführung

Die Qualität von interaktivem Rendering für Virtual-Reality-Anwendungen hat sich in den letzten Jahren stetig weiterentwickelt. Der Fortschritt wird vor allem von der für das Rendering eingesetzten Hardware vorangetrieben. Mit der schnellen Entwicklung von Haupt- und Grafikprozessoren eröffnen sich Anwendungsgebiete, die noch vor kurzer Zeit aufgrund ihres Rechenaufwands undenkbar gewesen wären, z.B. Echtzeit-Ray-Tracing. Trotz des Fortschritts aktueller Hardware gibt es Anwendungen, denen die Rechenleistung eines einzelnen Hosts nicht ausreicht. In diesem Fall muss auf einen Verbund mehrerer Rechner (Cluster) zurückgegriffen werden, die gemeinsam an der Berechnung arbeiten, ihre Ergebnisse über das Netzwerk koordinieren und an jene Maschinen schicken, die das Ergebnis am Ende darstellen. Auch auf den Darstellungsrechnern kann der Bedarf zur erneuten Verteilung über mehrere Rechner hinweg gegeben sein, sei es zur Realisierung einer großen, hochauflösenden Videowand aus vielen von einzelnen Rechnern betriebenen Displays (z.B. [SaHL03]), oder aber zur Überlagerung von mehreren gerenderten Bildern, wie es beispielsweise für eine klassische stereoskopische Virtual-Reality-Projektion der Fall ist.

Das *serverbasierte Rendering* ist ein weiteres Einsatzgebiet für verteiltes Rendering, bei dem man die Bildberechnung komplett auf ein entferntes Rechnercluster auslagert. Es ermöglicht Installationen, bei denen mehrere verteilte Nutzer ein Modell gleichzeitig anschauen und damit interagieren können, außerdem die Darstellung von vertraulichen Modellen, die nie den sicheren Server verlassen müssen. Eine Verteilung über das Internet macht zusätzlich serverseitige Nachbearbeitungsschritte zur Datenreduktion notwendig, z.B. eine adaptive und effiziente Codierung des Videostroms. Es sind außerdem oftmals noch weitere Operationen im Bildraum (auf Pixelebene) notwendig, bspw. Helligkeits- oder Farbanpassungen für gekachelte Displays.

All die genannten Anwendungsfälle fordern von einer Anwendung ein Höchstmaß an Einstellungen für Netzwerkverbindungen, Aufgaben- und Lastverteilung, Nachbearbeitungsschritte, Verteilung und Anordnung von Displaykomponenten.. Die Konfiguration der genannten Aspekte einer verteilten Anwendung sollte dennoch von jedermann auch ohne Programmierkenntnisse durchgeführt werden können.

Im Folgenden zeigen wir verwandte Arbeiten im Bereich des verteilten Rendering (Abschnitt 2) und stellen vier grundlegende Anwendungsszenarien vor, wie sie in typischen VR-Anwendungen auftreten (Abschnitt 3). Im darauf folgenden Abschnitt stellen wir die Architektur von URay vor, einem generischen und flexiblen Framework zur Verteilung von Computergrafik-Anwendungen im Netzwerk [RLRS08], bevor wir in Abschnitt 5 eine neue Beschreibungssprache für URay-Anwendungen vorstellen. In Abschnitt 6 wird diese Mächtigkeit und Flexibilität der Sprache anhand der zuvor vorgestellten Szenarien demonstriert. In Abschnitt 7 fassen wir die wichtigsten Punkte zusammen und beleuchten mögliche künftige Erweiterungen.

2 Verwandte Arbeiten

Im Themenfeld verteiltes Rendering gibt es diverse Arbeiten, die sich thematisch nah an URay befinden. Im kommerziellen Bereich ist z.B. die Visualisierungsplattform "VUE" von Silicon Graphics [SGI09] zu nennen, ein Softwarepaket mit dem Ziel, hochwertige visuelle Informationen zu kreieren und zu verteilen. Kernkomponente von VUE ist "RemoteVUE", das beliebige, auch hochauflösende, Pixelströme kombiniert und auf beliebige Endgeräte senden kann. Anwendungen reichen dabei von Server-Based Rendering mithilfe von SGIs Software-Renderer "SoftVUE" bis hin zu Display-Walls und HDTV-Streaming.

Während das VUE-System stark auf Endanwender ausgerichtet ist, und daher auf sehr hohem Abstraktionslevel konfigurierbar ist, finden sich im akademischen Bereich mehrere Low-Level-Frameworks für die eigentliche Verteilung des Rendering-Prozesses. Das Stream-Processing-Framework "Chromium" [HHN+02] ist einer der bekanntesten Vertreter dieser Gattung und ermöglicht verteiltes Rendering über mehrere Hosts hinweg indem es die systemeigene OpenGL-Bibliothek [Open09] durch eine eigene ersetzt, die ankommende OpenGL-Kommandos ins Netzwerk verteilt. Das Equalizer-Framework [EiPa07] basiert ebenfalls ausschließlich auf OpenGL, erweitert den Ansatz von Chromium aber, indem es höherrangige Kommandos über das Netzwerk verteilt und somit Bandbreite einspart. Außerdem erlaubt Equalizer den Betrieb von gekachelten Displays.

Während VUE zur Konfiguration seiner Szenarien einen grafischen Editor bietet, verwenden Chromium und Equalizer je ein reines Textformat zur Beschreibung einer Rendering-Installation. Die Beschreibungssprache, die wir in Abschnitt 5 für URay entwickeln werden, ist im Ansatz ebenfalls textuell, unterscheidet sich aber von Chromium und Equalizer klar durch ihre Grammatik, die sich mehr an eine objektorientierte Programmiersprache anlehnt als an ein reines Datenmodell. Durch Introspektion erlaubt URay ferner die Verwendung eigens definierter Komponenten in der Anwendungsbeschreibung, was bei den anderen Verteilungs-Frameworks nicht der Fall ist.

3 Anwendungsszenarien

Die im Folgenden diskutierten Szenarien und die an sie gestellten Anforderungen ergeben sich aus den typischen Anwendungen aus den Bereichen VR und Rendering.

Verteiltes Rendering (AS1)

Beim *verteilten Rendering* wird nicht nur ein einziger Rechner, sondern ein Cluster von Rechnern zum Berechnen der Bilder verwendet. Die Hauptanforderung (R1.1), die sich hierdurch ergibt, ist eine Möglichkeit, beliebige Rechner innerhalb eines Netzwerkes zum Rendering oder für die Darstellung der Bilder zu verwenden. Insgesamt erfordert

dies den Einsatz einer *Multimedia-Middleware*, welche die Übertragung von Bilddaten sowie den netzwerktransparenten Zugriff auf verteilte Objekte ermöglicht.

Als weitere Anforderung (R1.2) ergibt sich aus dem Szenario (AS1) die Möglichkeit, unterschiedliche Rendering-Technologien, wie sowohl Ray-Tracing als auch Rasterisierung zu verwenden. Um die Unabhängigkeit der Anwendungen von speziellen Implementierungen zu gewährleisten, darf ein Umschalten des Renderers aber keine Änderungen an den Anwendungen nach sich ziehen, die das Framework verwenden.

Da das verteilte Rendering insbesondere auch für das Berechnen von hochauflösenden Szenen, wie z.B. für Landschaftssimulationen, geeignet ist, ergibt sich aus (AS1) ebenfalls die Anforderung (R1.3), mehrere Bildschirme für die Darstellung der Bilder verwenden zu können. Teile der berechneten Bilder sollen dabei auf unterschiedlichen Bildschirmen dargestellt werden, die zusammen eine Videowand oder eine VR-CAVE ergeben. Hierdurch ergibt sich die Anforderung (R1.4) an eine verteilte Synchronisierung, um die einzelnen Teilbilder eines Bildes zeitgleich darzustellen.

Multi-View-Rendering (AS2)

Das zweite Anwendungsszenario umfasst *Multi-View-Rendering*, wobei mehrere Ansichten innerhalb einer Szene gleichzeitig berechnet werden. Dies ist insbesondere bei der Berechnung von Stereo-Bildern bis hin zu VR-Installationen, wie z.B. einer CAVE [CSD+92], erforderlich. Als Anforderung (R2.1) ergibt sich hieraus die Möglichkeit, beliebige Ansichten einer Szene synchronisiert zu berechnen und auf unterschiedlichen Bildschirmen darzustellen.

Post-Processing (AS3)

Das dritte Anwendungsszenario ermöglicht das *Post-Processing* gerendeter Bilder. Dies ist insbesondere für Anwendungen erforderlich, bei denen die berechneten Bilder über eine Netzwerkverbindung mit limitierter Bandbreite, wie z.B. dem Internet, übertragen werden müssen, oder Szenendaten an sich nicht herausgegeben werden dürfen und daher in einer sicheren Umgebung berechnet werden müssen. Gerade für industrielle Rendering- und VR-Anwendungen, bspw. im Automobilsektor, ist es essentiell, Kunden und Lieferanten interaktiven Zugang auf 3D-Modelle ihrer Produkte zu ermöglichen ohne die 3D-Modelle selbst herauszugeben.

Es ergeben sich daraus zwei weitere wichtige Anforderungen an das Framework: Zum einen soll die Möglichkeit bestehen, berechnete Bilder nachträglich zu bearbeiten, um z.B. adaptives Tonemapping, Farb- und Helligkeitskorrekturen vorzunehmen, oder die Daten vor der Übertragung über eine Netzwerkverbindung zu komprimieren (R3.1). Zum anderen sollte die Möglichkeit gegeben sein, unterschiedliche Transportprotokolle für die Übertragung der Bilddaten selektieren zu können, um bspw. den Einsatz einer verschlüsselten Verbindung (z.B. via SSL) zu ermöglichen oder verschiedene Streamingprotokolle (z.B. HTTP oder RTP) zu verwenden (R3.2).

Kollaboratives Rendering (AS4)

Das letzte Anwendungsszenario umfasst *kollaboratives Rendering*, das in einer beliebigen Kombination der Anwendungsszenarien (AS1) - (AS3) eingesetzt werden kann. Idealerweise sollte ein Framework die Umsetzung von großen Kontroll-Zentren mit Videowänden ermöglichen, mit denen sich aber auch mobile Geräte verbinden können, um gleichzeitig nur einige bestimmte Ausschnitte zu empfangen und darzustellen. Die wesentliche Anforderung (R4.1) an das Framework ist es, die gemeinsame Nutzung der berechneten Bilder über mehrere Clients hinweg zu ermöglichen, um so den Rendering-Aufwand zu minimieren. Weiterhin muss das Framework die Integration unterschiedlicher Ansätze für die Interaktion zwischen mehreren Benutzern ermöglichen.

4 Die URay-Architektur

Das URay Framework baut auf zwei grundlegenden Komponenten auf: Dem Real-Time-Szenengraph (RTSG) [GRHS08] und der netzwerkintegrierten Multimedia-Middleware (NMM) [LWRS08].

RTSG basiert auf einer strikten Trennung des 3D-Szenengraphen und der Implementierung von spezifischen Rendering-Komponenten. Hierdurch ist das Rendering unabhängig vom Szenengraphen, sodass auch neue Rendering-Technologien in RTSG integriert werden können (R1.2). RTSG stützt sich auf den ISO-Standard X3D [X3D08], der von den meisten Modellierungsanwendungen unterstützt wird.

Aufbauend auf RTSG wird NMM als zweite Technologie für die Umsetzung von URay verwendet. NMM ist hierbei vollständig für die verteilte Multimedieverarbeitung zuständig und verwendet das Konzept eines *verteilten Flussgraphen*. Die Knoten des Graphen stellen je eine Verarbeitungs-Komponente dar, z.B. das Komprimieren oder Darstellen eines Bildes. Die Kanten des Graphen repräsentieren eine Verbindung zwischen zwei Knoten und sind für die Datenübertragung zuständig. NMM ist netzwerktransparent und kann so auch Knoten auf entfernten Rechnern verwenden und konfigurieren (R1.1). Die Auswahl eines bestimmten Transportprotokolls kann dabei automatisch oder explizit von der Anwendung durchgeführt werden (R3.2). Darüber hinaus bietet NMM bereits die Möglichkeit die Verarbeitung und Wiedergabe der Medienströme auf mehreren Rechnern und Bildschirmen zu synchronisieren (R1.3). NMM unterstützt hierbei als einzige Multimedia-Middleware bereits drei grundlegende Anforderungen an das URay-Framework.

4.1 Der URay-Flussgraph

Um die beschriebenen Anwendungsszenarien zu realisieren, basiert das URay-Framework auf den folgenden Knoten, die z.B. zu dem in Bild 1 zu sehenden Flussgraphen verbunden werden können um (AS1) zu realisieren:

- **ManagerNode**: Der einzige Quell-Knoten im URay-Flussgraphen ist dafür zuständig, die anfallenden Berechnungen auf die Rechner eines Clusters zu verteilen. Der in URay verwendete Ansatz besteht darin, auf jedem Rechner nur einen Ausschnitt eines Bildes zu berechnen. Der ManagerNode verschickt hierfür Nachrichten an verfügbare Rendering-Knoten, wodurch die Last dynamisch auf diese Knoten verteilt wird.
- **RenderNode**: Der Rendering-Knoten berechnet mit Hilfe von RTSG aus einer 3D-Szenenbeschreibung ein Teil des 2D-Bilds. Die Information, welchen Ausschnitt er rendern soll, bekommt er vom Manager-Knoten.
- **TileAssemblyNode**: Dieser Knoten empfängt die berechneten Bildausschnitte von beliebig vielen Rendering-Knoten und setzt diese zu einem einzigen Bild zusammen. Jeder TileAssembly-Knoten leitet sein produziertes Bild an einen Display-Knoten weiter und empfängt nur jene Teilbilder als Eingabe, die der folgende Knoten für seinen entsprechenden Bildausschnitt benötigt.
- **DisplayNode**: Der Display-Knoten stellt eine Senke des URay-Flussgraphen dar und zeigt eintreffende Bilder synchronisiert an. URay ermöglicht es hierbei mehrere Display-Knoten zu spezifizieren, wobei je ein Knoten entweder das gesamte berechnete Bildes oder nur eine Teilansicht anzeigt.

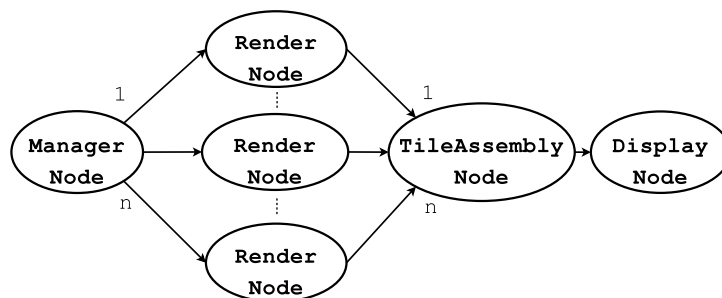


Bild 1: Dieser Flussgraph veranschaulicht das verteilte Rendering innerhalb des URay-Frameworks mit einem Display-Knoten und n Rendering-Knoten, die automatisch auf im Netzwerk verfügbare Rechner verteilt werden.

4.2 Verarbeitungsblöcke

Damit das URay-Framework eine einfache Umsetzung von unterschiedlichen Rendering- und VR-Anwendungen ermöglicht, wird der entsprechende Flussgraph, der die Bilder einer 3D-Szene berechnet, bearbeitet und präsentiert, in mehrere *Verarbeitungsblöcke* aufgeteilt. Diese Verarbeitungsblöcke können dann, je nach Anforderung der jeweiligen Anwendung, miteinander verbunden oder erweitert werden (siehe Bild 2). Die eigentliche Entwicklung von Anwendungen wird dadurch auf die Kombination und Konfiguration bestimmter Verarbeitungsblöcke beschränkt und muss sich nicht mit den Details der Implementierung beschäftigen.

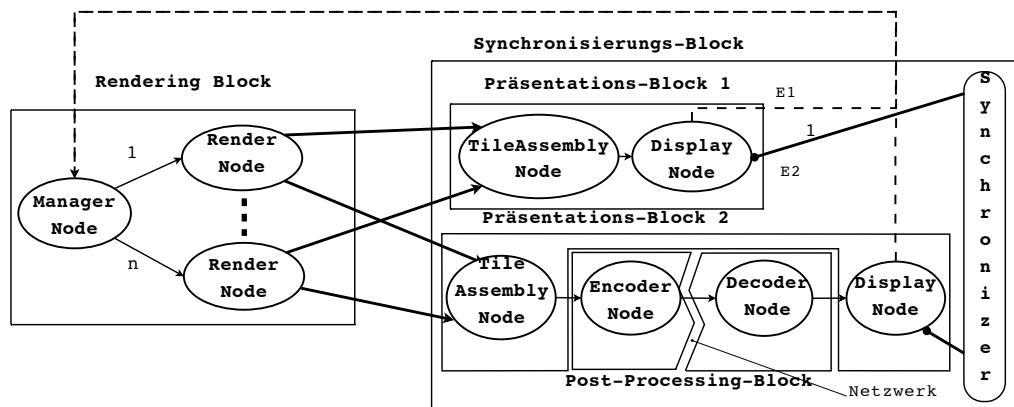


Bild 2: Aufbau einer URay-Anwendung, die gerenderte Bilder synchronisiert auf zwei Bildschirmen anzeigt. Die über den zweiten Präsentationsblock dargestellten Bilder werden vor einer Netzwerkübertragung zusätzlich komprimiert.

Rendering-Block

Der wichtigste Verarbeitungsblock, der in allen URay-Anwendungen enthalten ist, ist der *Rendering-Block*. Dieser Verarbeitungsblock beinhaltet alle NMM-Komponenten, die für das Rendering einer 3D-Szene erforderlich sind. Im Detail beinhaltet dieser Verarbeitungsblock einen Manager-Knoten, sowie mindestens einen Rendering-Knoten. Als Parameter bekommt dieser Block die Anzahl der zu benutzenden Rechner (optional alle verfügbaren). Er erzeugt dann intern die notwendige Anzahl an Rendering-Knoten und verteilt sie automatisch auf mehrere Rechner im Netzwerk, um eine maximale Performance zu erreichen. Darüber hinaus bietet er jedoch der Anwendung auch eine Schnittstelle, durch die eine explizite Verteilung der Knoten auf bestimmte Rechner vorgenommen werden kann, falls das erforderlich ist.

Präsentationsblock

Jeder Rendering-Block ist mit mindestens einem Präsentationsblock verbunden, der die berechneten Teilbilder wieder über einen TileAssembly-Knoten zu einem Bild zusammensetzt und über einen Display-Knoten anzeigt. Da die Informationen über den Bildausschnitt, der dargestellt werden soll, Teil des Verbindungsformates zwischen dem Rendering und dem Präsentationsblock ist, ist jedem Rendering-Knoten innerhalb des Rendering-Blocks bekannt, welche Teilbilder an einen bestimmten Präsentationsblock geschickt werden müssen. Hierbei ist jedem Präsentationsblock eine *Kamera* zugeordnet, wobei standardmäßig die in der Szene definierte Kamera verwendet wird. Der Block kann einen *Offset* zu dieser Kamera spezifizieren, der wiederum beliebig viele Offsets zugeordnet werden können.

Mit diesem Ansatz lassen sich sowohl Multi-User- (AS4) als auch Multi-View-Szenarien (AS2) spezifizieren. Bei Multi-User-Szenarien wird jedem Anwender eine eigene Kamera zuordnet, wobei Navigationsoperationen des Anwenders direkt auf die

Kamera angewendet werden. Multi-View-Szenarien werden realisiert, indem man jedem Präsentationsblock einen Offset zur selben Kamera zuordnet. Da Navigationsoperationen auf die Kamera angewendet werden, bleiben die Offsets automatisch bestehen.

Post-Processing-Block

Um die Anforderungen von (AS3) zu erfüllen und die berechneten Bilder nachträglich zu bearbeiten, bietet das URay-Framework die Möglichkeit an, in einen Präsentationsblock eine beliebige Anzahl von Post-Processing-Blöcken hinzuzufügen. Ein Post-Processing-Block selbst kann dabei beliebig viele interne Knoten, wie z.B. einen Knoten zum Anpassen der Helligkeit, oder Encoder- und Decoder-Knoten, enthalten. Diese internen Knoten, werden dabei zwischen den TileAssembly-Knoten bzw. den Senke-Knoten des zuletzt hinzugefügten Post-Processing-Blocks sowie den Display-Knoten des Präsentationsblocks gehängt. Damit keine Abhängigkeiten zwischen unterschiedlichen Post-Processing-Blöcken entstehen, entspricht das Ein- und Ausgangsformat des Quell- bzw. Senke-Knoten eines Post-Processing-Blocks, wieder unkomprimierten Videodaten.

Verbundblock

Um unterschiedliche aber auch komplexe Konfigurationen zu vereinfachen, unterstützt das URay-Framework sog. *Verbundblöcke*, was auf dem *Kompositum-Entwurfsmuster* [GHJV04] basiert. Hierbei können mehrere Verarbeitungsblöcke vom selben Typ zu einem Verbund-Block zusammengeschlossen werden, der sich für die Anwendung wie ein einziger Verarbeitungsblock des jeweiligen Typs verhält. So können z.B. mehrere Post-Processing-Blöcke zusammengefasst werden, um die Daten für ein bestimmtes mobiles Gerät anzupassen, was die Konfiguration der Anwendung deutlich vereinfachen kann.

Darüber hinaus können auch spezialisierte Verbundblöcke realisiert werden, um bestimmte Operationen automatisch auf mehrere Verarbeitungsblöcke anzuwenden. Ein Beispiel für einen spezialisierten Verbundblock ist der *Synchronisierungs-Block*, der eine beliebige Anzahl von Präsentationsblöcken enthalten kann und deren Display-Knoten mit der selben Synchronisierungs-Komponente von NMM verbindet, so dass die Darstellung der Bilder synchron erfolgt (R1.3). Darüber hinaus ist der Präsentationsblock selbst ein spezialisierter Verbundblock, der das Hinzufügen einer beliebigen Anzahl von Post-Processing-Blöcken unterstützt. Weiterhin werden über Verbundblöcke auch globale Bildoperationen realisiert, die das gesamte gerenderte Bild betreffen, wie z.B. Tonemapping. Hierbei wird der komplette Frame erst zusammengesetzt, dann die globale Bildoperation darauf angewendet, und danach die passenden Bildausschnitte an die zugehörigen Präsentationsblöcke weitergeleitet.

5 Eine Beschreibungssprache für URay-Anwendungen

Um die beschriebenen Verarbeitungsblöcke von URay möglichst einfach kombinieren und konfigurieren zu können, definieren wir eine eigene Syntax für URay-Konfigurationsdateien, sog. *Rendering-Graph-Descriptions* (RGDs). Mit dem Kommandozeilenwerkzeug "renderclic" können die so definierten Rendering-Flussgraphen ohne weitere notwendige Schritte abgespielt werden. Sowohl die Syntax der RGDs als auch die "renderclic"-Applikation sind von den Möglichkeiten zur Flussgraph-Spezifikation und der "clic"-Applikation innerhalb von NMM inspiriert [LWRS08].

Die Konfigurationssyntax und somit auch der RGD-Parser innerhalb des URay-Frameworks, auf dem auch die "renderclic"-Applikation basiert, verwenden als Grundlage eine kontextfreie Grammatik G, die wir im Folgenden Schritt für Schritt entwickeln werden. Wir verwenden zu deren Beschreibung die Backus-Naur-Form [WiMa92] und stellen per Konvention sämtliche Nichtterminalsymbole in spitzen Klammern dar (`<symbol>`), alle Terminalsymbole dagegen in Anführungszeichen ("`symbol`"). Bezeichner sind englischsprachig. Um schwer verständliche Rekursionen zu vermeiden, verwenden wir die folgenden Standard-Kurzschreibweisen: a) Symbole in eckigen Klammern ("`[]`") sind optional, und b) ein einem Symbol nachstehendes "+" erlaubt eine Wiederholung, d.h. das Symbol tritt mindestens einmal auf, darf aber auch mehrfach auftreten.

Wie in Abschnitt 4.2 bereits erwähnt, besteht eine URay-Anwendung stets aus genau einem Rendering-Block, der letztendlich in einem oder mehreren Präsentationsblöcken mündet. Da Präsentationsblöcke wiederum in Verbund-Blöcken geschachtelt sein können, entsteht der folgende einfache Ansatz für G, unter Verwendung von `<uray_graph>` als Startsymbol:

```
<uray_graph> ::= <rendering_block> "|" <composite_block>
<rendering_block> ::= "RenderingBlock"
<composite_block> ::= <identifier> "{" <composite_block>+ "}" |
<presentation_block>
<presentation_block> ::= "PresentationBlock"
```

Das Symbol `<identifier>` leiten wir an dieser Stelle nicht weiter ab, es kennzeichnet einen gültigen Bezeichner nach den Regeln aktueller Programmiersprachen, d.h. eine Zeichenkette ohne Sonder- und Leerzeichen, die mit einem Buchstaben beginnt. Als Symbol einer seriellen Verbindung zwischen zwei Blöcken wählten wir das aus der Unix-Welt bekannte Pipe-Symbol ("`|`") zur Darstellung des Informationsfluss. Die einfachste gültige RGD, die man bisher aus G erzeugen kann, ist daher diese:

```
RenderingBlock | PresentationBlock
```

Im Gegensatz zu seriell verbundenen Blöcken werden parallel verbundene Blöcke (d.h. mehrere gleichberechtigte Blöcke, die alle an den selben Vorgängerblock angeschlossen sind, in der RGD-Syntax ohne Verbindungssymbol hintereinander gestellt, müssen allerdings immer in einem umschließenden Verbundblock stehen. Eine Schachtelung von

zwei Präsentationsblöcken innerhalb eines Verbundblocks (hier ein Synchronisations-Block, der all seine Präsentationsblöcke synchronisiert darstellt) sieht dann folgendermaßen aus:

```
RenderingBlock | SyncBlock { PresentationBlock PresentationBlock }
```

Die erstellte Grammatik zielt bisher lediglich auf die *Kombination* von Blöcken, d.h. den Fluss der gerenderten Bilder ab. Eine wesentliche Komponente bei der Definition von RGDs ist aber die *Konfiguration* von einzelnen Blöcken um z.B. einem Rendering-Block eine Szene, oder einem Präsentationsblock seine Auflösung zuzuweisen.

Um die Erweiterbarkeit von Verarbeitungsblöcken und gleichzeitig eine enge Bindung der Konfiguration an die tatsächliche Implementierung zu gewährleisten, verwenden wir eine C-ähnliche Syntax für die Spezifikation von *Methodenaufrufen* für einzelne Blöcke. Dies ermöglicht neben einer intuitiven Definition des Rendering-Flusses auch eine Konfiguration der Blöcke unter Verwendung der identischen Methodensignaturen wie im C++-Quelltext.

Wir integrieren Methodenaufrufe in unsere Grammatik G indem wir gültigen Signaturen ein Dollarzeichen ("\$\$") voranstellen, und erweitern die bisherige Definition von Rendering- und Präsentationsblöcken:

```
<rendering_block> ::= "RenderingBlock" [ <method>+ ]
<composite_block> ::= <identifizier> [ <method>+ ] "{" <composite_block>+ "}" |
  <presentation_block>
<presentation_block> ::= "PresentationBlock" [ <method>+ ]
<method> ::= "$" <identifizier> "(" <arguments> ")"
```

Hier verzichten wir erneut auf eine feinere Ableitung der Nichtterminale `<identifizier>` für den Methodenbezeichner und `<arguments>` für eine aus Zeichenketten oder Zahlen bestehende Argumentliste. Eine um Methodenaufrufe erweiterte RGD könnte dann z.B. so aussehen, um Szene und Auflösung am jeweiligen Block zu definieren:

```
RenderingBlock $setSceneURL("file:///home/uray/scenes/box.wrl")
| PresentationBlock $setResolution(1024, 768)
```

Die Erweiterung der Blöcke um optionale Methoden ermöglicht so eine generische Schnittstelle für die Konfiguration sämtlicher Verarbeitungsblöcke, seien dies vordefinierte Blöcke oder eigene Erweiterungen. URay verwendet Introspektion um zum Zeitpunkt des Parsens einer RGD die angegebenen Methoden und Argumente mit tatsächlich implementierten Schnittstellenmethoden zu vergleichen und eine direkte Fehlerbehandlung zu ermöglichen.

Die letzten fehlenden URay-Komponenten zur vollständigen externen Beschreibung von Rendering-Graphen sind Post-Processing-Blöcke. Diese werden nun in der Grammatik in einen neu zu definierenden Rumpf des Präsentationsblocks integriert, analog zu ihrer tatsächlichen Position im darunterliegenden Multimedia-Flussgraphen. Die ei-

gentliche Funktionalität eines Post-Processing-Blocks (z.B. Helligkeitsanpassung oder Videocodierung) kann nun auf zwei Arten erfolgen: entweder durch Implementierung eines eigenen Post-Processing-Block-Typs in C++ und anschließender Verwendung in der RGD, oder aber durch direkte Spezifikation eines inneren NMM-Flussgraphen.

Um letzteres zu ermöglichen, definieren wir einen speziellen Rumpf für einen Post-Processing-Block, der intern direkt zur Verarbeitung an NMM weitergereicht wird. Wir erweitern G also wie folgt, um Post-Processing-Subgraphen innerhalb von Präsentationsblöcken zu ermöglichen:

```
<presentation_block> ::= "PresentationBlock" [ <method>+ ] [ <presentation_body> ]
<presentation_body> ::= "{" <postprocessing_graph> "}"
<postproc_graph> ::= <postprocessing_block> [ "|" <postprocessing_block> ]
<postproc_block> ::= "PostProcessingBlock" [ <method>+ ]
                  [ "[" <nm_flowgraph> "]" ]
```

Der gesamte NMM-Flussgraph, also der Inhalt des Nichtterminals `<nm_flowgraph>` welches in eckigen Klammern einem Post-Processing-Block anhängt, wird ohne direkte interne Fehlerbehandlung durch URay direkt an NMM weitergereicht. Der URay-Parser verarbeitet dann lediglich das Ergebnis dieser Operation, d.h. gibt entweder den Fehler aus oder integriert den resultierenden Flussgraph in den URay-Render-Graph. Beispielsweise kann man jetzt den folgenden Präsentationsblock mit integriertem Post-Processing-Block spezifizieren:

```
PresentationBlock $setSceneURL("file:///home/uray/scenes/box.wrl")
{
  PostProcessingBlock [ BrightnessNode $setMultiplier(0.95) ]
}
```

Somit kann man die gesamte URay-Architektur extern durch Spezifikation einer RGD-Datei beschreiben und konfigurieren. Im folgenden Abschnitt wird die soeben hergeleitete Syntax anhand weitererführender Beispiele verdeutlicht.

6 Realisierung der Anwendungsszenarien in URay

Dieser Abschnitt erklärt, wie die in Abschnitt 3 beschriebenen Anwendungsszenarien auf Basis der in Abschnitt 5 definierten Syntax für Render-Graph-Descriptions spezifiziert werden können. Gleichzeitig werden die wichtigsten Konfigurationsmöglichkeiten für die vordefinierten URay-Verarbeitungsblöcke erläutert.

6.1 Verteiltes Rendering (AS1)

Die einfachste Konfiguration für das mit URay realisierbare verteilte Rendering besteht aus einem Rendering-Block und nur einem Präsentationsblock. Der einzelne Präsentationsblock definiert dabei genau ein Ausgabefenster auf genau einem Host.

RGD 1 beschreibt verteiltes Rendering einer VRML-Szene "test.wrl" auf den beiden Hosts "host1" und "host2", und eine Darstellung des gerenderten Videostreams auf dem einzelnen Host "display_host".

```

RenderingBlock $addLocation("host1")
                $addLocation("host2")
                $setRenderingEngine("RTfact")
                $setSceneURL("file:///home/uray/models/box.wrl")
| PresentationBlock $setDisplayLocation("display_host")
                   $setResolution(1024, 768)

```

RGD 1: Verteiltes Rendering mit einem Darstellungs-Host

Der Rendering-Block dieser Konfiguration wird mit den Netzwerk-Namen der Hosts konfiguriert, die für das Rendering verwendet werden sollen. Außerdem wird eine Szenendatei angegeben und pro Rendering-Block ein Renderer definiert. Im Beispiel wird der Renderer "RTfact" [GeSI08] verwendet, der via RTSG bereits in URay integriert ist. Für die eindeutige Konfiguration des Präsentationsblocks benötigt URay den Namen des zugeordneten Hosts, der zur Darstellung verwendet werden soll, außerdem die Auflösung des Fensters, in das gerendert werden soll.

Eine Erweiterung des bisher vorgestellten Szenarios ist das verteilte Rendering, bei dem das Ergebnis auf *mehrere* Ausgabegeräte verteilt wird, wie z.B. für eine Videowand. Hier wird an den zuvor definierten Rendering-Block ein weiterer Präsentationsblock angehängt. Die Synchronisation der Ausgabe auf sämtliche Bildschirme wird dabei durch einen speziellen Verbundblock vom Typ "Sync-Block" erreicht, der alle beteiligten Präsentationsblöcke umschließt. RGD 2 beschreibt dieses Szenario, bei dem die Ausgabe synchron auf die beiden Hosts "display_host1" und "display_host2" erfolgt.

```

RenderingBlock // ... wie in RGD 1
| SyncBlock
{
  PresentationBlock $setLocation("display_host1")
                   $setResolution(512, 768) // Default-Offset hier: (0, 0)
  PresentationBlock $setDisplayLocation("display_host2")
                   $setResolution(512, 768)
                   $setPixelOffset(512, 0)
}

```

RGD 2: Verteiltes Rendering mit zwei Darstellungs-Hosts (Videowand-Installation)

Für jeden Präsentationsblock kann ein relativer Pixelversatz (Offset) innerhalb des gerenderten Bildes definiert werden, der gekachelte Szenarien, bei denen jedes Anzeigefenster nur einen Teil des Gesamtbilds anzeigt, erst ermöglicht. URay berechnet in diesem Falle die Gesamtgröße des zu rendernden Bildes als Vereinigung sämtlicher konfigurierten Kacheln.

6.2 Multi-View-Rendering (AS2)

Zusätzlich ermöglicht URay, *mehrere Blickpunkte* in die dargestellte Szene zu definieren, und synchron zu rendern. Dafür werden die entsprechenden Präsentationsblöcke in einen Verbundblock vom Typ "MultiViewSyncBlock" integriert. Dieser Block definiert eine für all seine Präsentationsblöcke gültige Kamera. Jeder enthaltene Präsentationsblock definiert dann nur noch einen Offset zu der gemeinsamen Kamera. RGD 3 zeigt die Umsetzung von einfachem Stereo-Rendering, bei der jedes Auge durch einen Präsentationsblock mit fixem Offset zur Hauptkamera repräsentiert wird.

```

RenderingBlock // ... wie in RGD 1
| MultiViewSyncBlock $setCameraPosition(0, 0, 0)
                    $setCameraDirection(0, 0, -1)
{
  PresentationBlock $setDisplayLocation("display_host1")
                    $setResolution(1024, 768)
                    $setCameraPositionOffset(-0.05, 0, 0) // Linkes Auge
  PresentationBlock $setDisplayLocation("display_host2")
                    $setResolution(1024, 768)
                    $setCameraPositionOffset(0.05, 0, 0) // Rechtes Auge
}

```

RGD 3: Stereo-Rendering (Powerwall-Szenario)

6.3 Rendering mit Nachbearbeitung (AS3)

In einem weiteren Beispiel soll die Möglichkeit zur Nachbearbeitung gerendeter Videostreams gezeigt werden. Hierzu stellen wir uns erneut ein VR-Stereo-Szenario vor, bei dem jedoch die einzelnen Bilder der Projektoren einen Nachbearbeitungsschritt durchlaufen um Helligkeit und Farbe aneinander anzugleichen. RGD 4 redefiniert im Folgenden das Powerwall-Szenario von RGD 3, fügt aber noch Post-Processing-Blöcke innerhalb der Präsentationsblöcke ein, die durch einen reinen NMM-Flussgraphen das zu präsentierende Bild in Helligkeit, bzw. Farbe modifizieren:

```

RenderingBlock // ... wie RGD 3
| SyncBlock
{
  PresentationBlock // ... wie RGD 3, linkes Auge
  {
    PostProcessingBlock [ BrightnessNode $setBrightness(0.94) ] // NMM-Flussgraph
  }
  PresentationBlock // ... wie RGD 3, rechtes Auge
  {
    PostProcessingBlock
    [
      ColorCorrectionNode $setMultipliers(1, 1.07, 0.9) // NMM-Flussgraph
    ]
  }
}
}

```

RGD 4: Powerwall-Installation mit Nachbearbeitung der gerenderten Bilder.

6.4 Kollaboratives Rendering (AS4)

Das explizite Setzen von mehreren Views in die identische Szene via Blickpunkt, Blickrichtung und vertikalem Öffnungswinkel der Kamera ermöglicht nicht nur Stereo-Anwendungen wie in Abschnitt 6.2 beschrieben, sondern auch Installationen, in denen mehrere User die gleiche Szene sehen und sich darin bewegen, sog. *kollaborative Szenarien*. Obwohl die URay-Architektur mehrere Szenen berücksichtigen kann, bezieht sich die Kollaboration in dieser Arbeit nur auf eine einzige Szene mit mehreren Benutzern, nicht auf mehrere verbundene Szenen.

Kollaborationen über das Internet oder andere Netzwerke mit geringer oder stark schwankender Bandbreite sowie hoher Latenz erfordern eine Komprimierung des gerenderten Datenstroms, z.B. durch Verwendung eines geeigneten Videocodecs. Analog zu RGD 4 ist es möglich, Encoding und Decoding des Videostroms innerhalb eines Post-Processing-Blocks als reine NMM-Komponenten unterzubringen. RGD 5 zeigt ein Szenario für serverbasiertes Rendering, bei dem auf dem Server "server" die Zusammensetzung der Teilbilder und eine Codierung des Videostroms via DivX-Codec erfolgt. Der Client "client" decodiert den empfangenen Videostrom wieder und stellt ihn dar.

```
...
| PresentationBlock $setTileAssemblyLocation("server")
                    $setDisplayLocation("client")
                    $setResolution(1024, 768)
  {
    PostProcessingBlock
    [
      DivXEncodeNode $setLocation("server")
                    $setBitrate(2000000) // entspricht 2 MBit/s
      ! DivXDecodeNode $setLocation("client")
    ]
  }
}
```

RGD 5: Serverbasiertes Rendering mit Videocodierung und -decodierung

7 Zusammenfassung und Ausblick

In dieser Arbeit wurde das URay-Framework vorgestellt. Durch die Verwendung der flexiblen Rendering-Architektur RTSG sowie NMM für die Multimedia-Verarbeitung wird eine beispiellose Flexibilität erreicht, wodurch alle wesentlichen Aspekte eines Rendering-Systems parallelisiert und im Netzwerk verteilt werden können: User-Eingabe, Rendering, Post-Processing, Präsentation und Synchronisierung.

Darüber hinaus haben wir gezeigt, dass durch die Bereitstellung und Kombination von Verarbeitungsblocken innerhalb des URay-Frameworks die Entwicklung selbst von komplexen Anwendungen stark vereinfacht wird, wodurch neue Anwendungsszenarien

Zeit- und Kosteneffektiv umgesetzt und evaluiert werden können. Durch die Verwendung der ebenfalls in dieser Arbeit vorgestellten Anwendung "renderclie" müssen hierfür nicht einmal mehr neue Anwendungen programmiert werden. Stattdessen reicht es aus die Verbindung der erforderlichen Verarbeitungsblöcke sowie deren Konfiguration über eine einfache Grammatik innerhalb einer Textdatei zu beschreiben. So lassen sich z.B. mit Hilfe des Werkzeugs "renderclie" alle in Abschnitt 3 beschriebenen Anwendungsszenarien mit wenigen Textzeilen realisieren. Natürlich haben Anwendungen auch direkten Zugriff auf alle Konfigurationen, die wir hier vorgestellt haben und können diese zur Laufzeit beliebig ändern.

Zukünftige Arbeiten werden darauf ausgelegt sein, "renderclie" um eine graphische Benutzungsschnittstelle zu erweitern, so dass Verarbeitungsblöcke auch einfach zur Laufzeit hinzu geschaltet, rekonfiguriert oder wieder entfernt werden können. Darüber hinaus werden für kollaborative VR-Anwendungen geeignete Interaktionsparadigmen evaluiert und in URay integriert.

Literatur

- [CSD+92] CRUZ-NEIRA, C.; SANDIN, D.J.; DEFANTI, T.A.; KENYON, R.V.; HART, J.C.: The CAVE: Audio Visual Experience Automatic Virtual Environment. In *Communications of the ACM*, 35(6), ACM Press, 1992.
- [EiPa07] EILEMANN, S.; PARAJOLA, R.: *The Equalizer Parallel Rendering Framework*. Technical Report IFI 2007.06, Department of Informatics, University of Zürich, 2007.
- [GHJV04] GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J.: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley, 2004.
- [GRHS08] GEORGIEV, I.; RUBINSTEIN, D.; HOFFMANN, H.; SLUSALLEK, P.: Real Time Ray Tracing on Many-Core-Hardware. In *Proceedings of the 5th INTUITION Conference on Virtual Reality*, 2008.
- [GeSI08] GEORGIEV, I.; SLUSALLEK, P.: RTfact: Generic Concepts for Flexible and High Performance Ray Tracing. In *IEEE/Eurographics Symposium on Interactive Ray Tracing 2008*, Aug 2008.
- [HHN+02] HUMPHREYS, G.; HOUSTON, M.; NG, R.; FRANK, R.; AHERN, S.; KIRCHNER, P.D.; KLOSOWSKI, J.T.: Chromium: a Stream-Processing Framework for Interactive Rendering on Clusters. In *SIGGRAPH '02: Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, ACM Press, 2002.
- [LWRS08] LOHSE, M.; WINTER, F.; REPLINGER, M.; SLUSALLEK, P.: Network-Integrated Multimedia Middleware (NMM). In *MM '08: Proceedings of the 16th ACM International Conference on Multimedia*, ACM Press, 2008.
- [Open09] *Khronos Group: OpenGL*. <http://www.opengl.org>.
- [RLRS08] REPLINGER, M.; LÖFFLER, A.; RUBINSTEIN, D.; SLUSALLEK, P.: *URay: A Flexible Framework for Distributed Rendering and Display*. Technical Report 2008-01, Universität des Saarlandes, Saarbrücken, 2008.
- [SaHL03] SANDSTROM, T.A.; HENZE, C.; LEVIT, C.: The Hyperwall. In *Proceedings of IEEE Coordinates and Multiple Views in Exploratory Visualization*, 2003.
- [SGI09] *Silicon Graphics: VUE - Visual User Experience*. <http://www.sgi.com/vue>.

[WiMa92] WILHELM, R.; MAURER, D.: *Übersetzerbau: Theorie, Konstruktion, Generierung*. Springer, 1992.

[X3D09] *X3D Spezifikation*. <http://www.web3d.org/x3d/specifications>.

Autoren

Alexander Löffler absolvierte ein Fachhochschulstudium "Digitale Medien" an der FH Kaiserslautern sowie ein Masterstudium "Visual Computing" an der Universität des Saarlandes. Während seiner Studien beschäftigte er sich vornehmlich mit Computergrafik und 3D-Szenengraphen im Virtual-Reality-Kontext, außerdem mit den grafischen Aspekten verteilter Multimedia-Middleware. Seit 2008 ist er wissenschaftlicher Mitarbeiter am Lehrstuhl für Computergrafik bei Prof. Dr.-Ing. Philipp Slusallek und dort u.a. für die Integration von Szenengraphen und Verteilungsstrategien in aktuelle Virtual-Reality-Systeme zuständig.

Michael Replinger studierte Informatik an der Universtät des Saarlandes. Seit 2003 ist er wissenschaftlicher Mitarbeiter und Promotionsstudent am Lehrstuhl für Computergrafik bei Prof. Dr.-Ing. Philipp Slusallek und leitet seit 2006 das Projekt Netzwerk-Integrierte Multimedia Middleware (NMM). Im Rahmen seiner bisherigen Forschungstätigkeiten hat er Arbeiten in den Bereichen verteilte Multimedia- und Rendering-Architekturen sowie parallele Systeme publiziert.

Prof. Dr.-Ing. Philipp Slusallek ist seit 2008 Wissenschaftlicher Direktor am Deutschen Forschungszentrum für Künstliche Intelligenz (DFKI), wo er den Forschungsbereich "Agenten und Simulierte Realität" leitet. Seit 1999 ist er darüber hinaus Professor am Lehrstuhl Computergraphik an der Universität des Saarlandes und seit 2004 Gründungssprecher des dortigen Kompetenzzentrums Informatik. Bevor er 1999 ins Saarland kam, war er Visiting Assistant Professor an der Stanford Universität in den USA. Er studierte Physik in Frankfurt und Tübingen (Diplom) und promovierte in Informatik an der Universität Erlangen. Seine Forschungsinteressen liegen in der Integration von Computergraphik und Künstlicher Intelligenz, bei der Nutzung von Many-Core-Hardware zum Rendering sehr komplexer und hoch realistischer Modelle, sowie in verteilten Anwendungen für Multimedia und Future-3D-Internet.