

Virtual Room

Entwicklung einer Anwendung für visuelle Kommunikation und Interaktion in einem virtuellen Raum

Bachelorthesis

Studiengang:

Informatik

Autoren:

Marc Wacker, Cédric Renggli, Roman Kühne

Betreuer:

Marcus Hudritsch

Experte:

Dr. Federico Flueckiger

Datum:

19.06.2014

Abstract

Computergenerierte Welten werden dank neuer Technologien stetig realistischer. Neue Ein- und Ausgabegeräte stärken dabei das Gefühl, sich in einer virtuellen Welt zu befinden. Eines dieser Eingabegeräte ist die Microsoft Kinect, eine Kamera, welche die Körperhaltung und Position einer Person im Raum erkennen kann. Ein weiteres innovatives Gerät ist die Oculus Rift. Das auf dem Kopf montierte, brillenartige visuelle Ausgabegerät besitzt einen integrierten Bildschirm, der das gesamte Sichtfeld abdeckt. Zudem verfügt die Oculus Rift über Orientierungssensoren, mit denen es möglich ist, ein virtuelles Bild zu generieren, das der Blickrichtung des Trägers entspricht. Mit ihrem grossen Sichtfeld ermöglicht die Oculus Rift eine perfekte stereoskopische Darstellung, die den Eindruck entstehen lässt, dass der Betrachter sich *in* der virtuellen Welt befindet. Positionsänderungen des Körpers, insbesondere des Kopfes, führen jedoch zu keiner Aktualisierung des Bildes und verschlechtern dadurch das Gefühl, ein Teil der virtuellen Umgebung zu sein.

Die Erfassung durch die Kinect ermöglicht eine virtuelle Nachbildung des eigenen Körpers. Die in diesem Projekt durchgeführte Erweiterung in Kombination mit der Oculus Rift lässt eine Person komplett in eine virtuelle Welt eintauchen. Die Möglichkeit, die Bewegungen des eigenen virtuellen Körpers räumlich und zeitlich korrekt sehen zu können, führt zu einer starken, unterbewussten Identifikation mit diesem.

Durch die implementierte Übertragung von mehreren Personen in einen gemeinsamen virtuellen Raum wird eine reale Begegnung nachgebildet. Dabei können sich diese Personen an unterschiedlichen geographischen Standorten befinden. Diese Personen wird zusätzlich die Möglichkeit geboten, gemeinsame Objekte betrachten und manipulieren zu können. Als mögliche Implementation treten zwei Personen in einem Brettspiel gegeneinander an. Die generische Herangehensweise im Projekt ermöglicht eine einfache Entwicklung von weiteren Interaktionsanwendungen.

Inhaltsverzeichnis

1. Einleitung	5
1.1 Ausgangslage	5
1.2 Ziel	6
1.2.1 Visualisierung des eigenen Skelettes	6
1.2.2 Zusammenführen von mehreren Personen	6
1.2.3 Interaktion im virtuellen Raum	6
1.2.4 Visuelle Verbesserungen (Rigged Mesh Animierung)	7
1.3 Projektplanung	7
2. Architektur	9
2.1 Systemarchitektur	9
2.2 Client	9
2.2.1 Input Daten	10
2.2.2 Datenverarbeitung	12
2.2.3 Output der Daten	12
2.3 Server	13
2.3.1 Player / Observer Verwaltung	13
2.3.2 Datenverarbeitung / Simulation	14
3. Entwicklungsumgebung	15
3.1 Libraries / SDKs	15
3.1.1 SLProject	15
3.1.2 Kinect for Windows SDK v1	15
3.1.3 Kinect for Windows SDK v2	15
3.1.4 RakNet	15
3.1.5 Assimp	16
3.2 MS Visual Studio 2012 Projekt	16
3.2.1 SVN	16
3.2.2 Projekt Struktur	16
3.2.3 Ordnerstruktur	17
3.2.4 Visual Studio Konfigurationen	17
4. Implementation	19
4.1 Anpassungen SLProject	19
4.1.1 Ausgangslage	19
4.1.2 Änderungen an SLSce	20
4.1.3 Änderungen an SLSce	20
4.1.4 Änderungen an SLIn	20
4.1.5 Manipulation der aktiven Kamera	21
4.2 Skelett Visualisierung	22
4.2.1 Kinect Daten	22
4.2.2 Skelett Daten	23
4.2.3 Daten aufbereiten	24
4.2.4 Daten-Darstellung	25
4.2.5 Zusammenfassung	26
4.3 Client / Server Kommunikation	27
4.3.1 Grundfunktionalität	27
4.3.2 Common Library	27
4.3.3 Serverseitige Implementation	31
4.3.4 Netzwerk Test-Client	33
4.3.5 Virtual Room Client	34

4.4 Game Implementationen	39
4.4.1 Color Sphere	40
4.4.2 Board Game Pattern	41
4.4.3 Turn Table	50
4.5 Rigged Mesh	51
4.5.1 Was ist ein rigged Mesh	52
4.5.2 Skinning	52
4.5.3 Importer	53
4.5.4 Kinect Data Binding	57
4.5.5 Verbesserungen / andere Ansätze	58
4.5.6 Verwendete Modelle	59
4.5.7 Verwendung von Blender	59
5. Tests & Messungen	60
5.1 Minimale Szene	60
5.1.1 Beispiel: „weisser Punkt“	60
5.2 Kinect Daten	60
5.3 Netzwerk	61
5.3.1 Antwortzeit vom Server	61
5.3.2 Erhalten der Datenpakete	62
5.3.3 Ergebnisse	62
6. Schlussfolgerung / Fazit	63
6.1 Interaktion / Immersion	63
6.2 Vision	64
7. Glossar	65
8. Abbildungsverzeichnis	66
9. Tabellenverzeichnis	67
10. Literaturverzeichnis	67
11. Anhang	68

1. Einleitung

Dieses Dokument wurde im Rahmen der Bachelor Thesis „Virtual Room“ erstellt. Zusätzlich zur Projektbeschreibung wird detailliert auf die Architektur und Implementation eingegangen. Das ist wichtig, weil die Möglichkeit auf ein Folgeprojekt bestehen könnte. Als Erstes werden die groben Zusammenhänge der einzelnen Komponenten beschrieben, womit der Funktionsumfang ersichtlich wird. In einem weiteren Teil befinden sich die Detailinformationen zu den einzelnen Komponenten.

1.1 Ausgangslage

Im Rahmen dieser Bachelor Arbeit wurde auf zwei verschiedene Hardware Komponente zurückgegriffen. Als Erstes kam die Oculus Rift der Firma Oculus VR zum Einsatz. Die Rift ist ein auf dem Kopf getragenes visuelles Ausgabegerät, auch als Head Mounted Display (HMD) bezeichnet. Wie es der englische Name schon suggeriert, wird dabei ein kleiner Bildschirm mit einer helmartigen Befestigung unmittelbar vor den Augen positioniert. Durch eine Abtrennung wird sichergestellt, dass linkes und rechtes Auge ausschliesslich Informationen von der ihnen zugewiesene Hälfte des Bildschirms erhalten. Durch diese Technologie wird eine möglichst reale stereoskopische Sichtweise auf virtuelle Objekte gewährleistet. Unter stereoskopischem Sehen versteht man das Vorhandensein einer glaubhaften Tiefenwahrnehmung durch beidäugiges Betrachten von Objekten. Anders als bei gängigen 3D Filmen, wie wir sie aus den Kinos kennen, sind die Augen bei der Rift komplett von der Außenwelt abgeschottet. Dank dieser Vorgehensweise kann eine möglichst grosse Immersion geschaffen werden. Unter Immersion versteht man ein für das Gehirn glaubhaftes Eintauchen in eine virtuelle Welt. Einfacher ausgedrückt: Eine Person mit der Rift hat das Gefühl sich tatsächlich an einem anderen Ort zu befinden. Ein weiteres Ziel dieser Art von virtuellen Brillen ist es das gesamte Sichtfeld eines Menschen abzudecken. Dadurch kann sich eine Person in der computergenerierten Umgebung frei umsehen. Um dies zu erreichen, müssen die fiktiven Objekte basierend auf der aktuellen Blickrichtung des Menschen dargestellt werden. Die dafür notwendige Ausrichtung des Kopfes wird in der Rift mittels verschiedener Sensoren errechnet und kann somit für eine korrekte Projektion auf den Bildschirm verwendet werden.

Die Idee einer vollständigen Immersion, anhand eines auf dem Kopf getragenen visuellen Ausgabegerätes, wurde bereits früh in den Siebziger durch die Pioniere Ivan Sutherland (Massachusetts Institute of Technology) und Raymond Goertz (Argonne National Laboratory) anhand eines Prototypen demonstriert [10]. Die damalige Grösse des Gerätes, die Genauigkeit der Sensoren, die Auflösung des Bildschirms und die Kosten für eine solche Technologie verhinderten es, ein massentaugliches Produkt auf den Markt bringen zu können. Erst im März 2013 schaffte es die Firma Oculus VR all diese Schwachpunkte mit ihrer ersten Version der Brille Rift für interessierte Entwickler abzudecken. Der Hype um dieses Produkt ist seither riesig und nimmt auch nicht ab. Dies beweist vorwiegend die Ankündigungen von Konkurrenten ebenfalls ähnliche Produkte auf den Markt bringen zu wollen.

Ein wesentlicher Schwachpunkt der Rift ist es einzig den Kopf drehen zu können. Simulierte Bewegungen in der virtuellen Umgebung, wie das Laufen, welche jedoch real nicht stattfinden, bringen bei den meisten Menschen das Gehirn in ein ziemliches Durcheinander. Dies kann bei schnellen Abläufen, wie zum Beispiel das Treppenlaufen, zu Übelkeitserscheinungen führen. Um diesem Effekt entgegen wirken zu können, wurde die Kinect von Microsoft als zweite Hardware Komponente eingesetzt.

Die Kinect ist eine Kamera, welche zusammen mit der Spielkonsole Xbox 360 im November 2010 erstmals verkauft wurde. Anhand von diesem Gerät können Spieler jeweils ihren eigenen Körper für die Steuerung von Charakteren in diversen Games verwenden. Die 3D Kamera erfasst dafür den Raum samt sich darin befindenden Personen sowie deren relativen Entfernung zur Kinect. Durch interne Berechnungen liefert der Sensor nicht nur die Position eines Spielers, sondern ermittelt ebenfalls den Standort von einzelnen Gelenken wie der Kopf, das Knie, die Schulter oder Hände.

Diese Daten können in Zusammenhang mit der Rift reale Bewegungen vom eigenen Körper möglichst exakt in die virtuelle Umgebung abbilden. Die Bewegungsfreiheit sollte dadurch ausschliesslich durch die Limitierung der Kinect eingeschränkt werden. Zusätzlich kann der eigene Körper in der virtuellen Welt dargestellt werden. Dies hat zum Vorteil, dass sich die Person auf eingeschränktem Raum betrachten und frei bewegen kann. Dabei kann das Heben des Beines oder das Beugen des Ellenbogens ohne grosse Verzögerung direkt in einer simulierten Umgebung betrachtet werden. Anhand von diesem Zusatz sollte das Gefühl einer vollständigen Immersion verstärkt werden.

Durch die Projekt 2 Arbeiten sind wir bereits mit der Rift und der Kinect vertraut. Es sind minimale Applikationen vorhanden, welche die Anbindungen für die Entwicklung beinhalten. Außerdem wird das SLProject [3] als Grundlage für die Darstellung benutzt.

1.2 Ziel

Die Grundidee ist es, einen virtuellen Raum zu kreieren, in dem sich eine Person frei bewegen und umsehen kann. Zusätzlich sollen sich in dieser virtuell eingeschränkten Umgebung mehrere Personen begegnen können. Die Personen können sich dabei an unterschiedlichen geografischen Standorten befinden. Der Datenaustausch soll via ein Netzwerk über einen zentralen Server erfolgen. Damit dies erreicht werden kann, haben wir die folgenden Teilziele definiert:

1. Visualisierung des eigenen Skelettes
2. Zusammenführen von mehreren Personen
3. Interaktion im virtuellen Raum
4. Visuelle Verbesserungen (Rigged Mesh Animierung)

1.2.1 Visualisierung des eigenen Skelettes

Als Erstes müssen die Daten der Oculus Rift und der Kinect für die korrekte Darstellung in der Rift kombiniert werden. Anschliessend kann ein einfaches Skelett anhand der Daten der Kinect in der virtuellen Umgebung abgebildet werden. An der Kopfposition wird die Kamera hinzugefügt, welche die Sichtweise basierend auf den Sensordaten der Rift übernimmt. Das Skelett wird mit einfachen Objekten wie Kugeln und Zylindern analog einem dreidimensionalen Strichmännchen dargestellt. Mit dieser Kombination soll es möglich sein, sich möglichst real in der virtuellen Welt zu bewegen und seine eigenen Körperbewegungen mit zu verfolgen. Durch den eingeschränkten Erfassungsbereich der Kinect wird die reale Bewegungsfreiheit eingeschränkt. Um dies virtuell ebenfalls abbilden zu können, wird die virtuelle Welt als kleiner geschlossener Raum repräsentiert.

1.2.2 Zusammenführen von mehreren Personen

Damit sich mehrere Personen in einer virtuellen Welt gegenüberstehen können, müssen die Skelett-Daten ausgetauscht werden. Dazu soll ein zentraler Server erstellt werden, welcher für das Sammeln und Verteilen der Daten verantwortlich ist. Mit der Visualisierung des Skelettes einer anderen Person und insbesondere durch die Übertragung der Bewegungen wird eine neuartige Kommunikation auf visueller Ebene ermöglicht.

1.2.3 Interaktion im virtuellen Raum

Neben dieser neuen Art sich virtuell zu begegnen sollen Interaktionsmöglichkeiten implementiert werden. Alle beteiligten Personen sollen die Möglichkeit bekommen, eine gemeinsame Struktur zu beeinflussen und so zum Beispiel in einem einfachen Brettspiel gegeneinander anzutreten.

1.2.4 Visuelle Verbesserungen (Rigged Mesh Animierung)

Dieses Projekt kann praktisch endlos visuell, wie auch funktionell erweitert werden. In diesem letzten Teil geht es darum, optionale Ideen umzusetzen. Dazu gehört zum Beispiel die Implementation und Animation von einem virtuellen menschenähnlichen 3D Objekt, ein sogenanntes Mesh, an der Stelle des einfachen Skelettes. Mit diesem Vorhaben könnte der Eindruck sich tatsächlich in der virtuellen Umgebung zu befinden und andere reale Personen zu begegnen noch zusätzlich gestärkt werden.

1.3 Projektplanung

Für die Projektplanung dienen die Teilziele als Grundlage. Wir haben einen groben Zeitplan mit drei Meilensteinen erstellt. Diese definieren die wichtigsten Arbeiten, wodurch wir den Projektfortschritt im Auge behalten können. Außerdem dient es der Aufgabenzuteilung innerhalb unserer Projektgruppe.

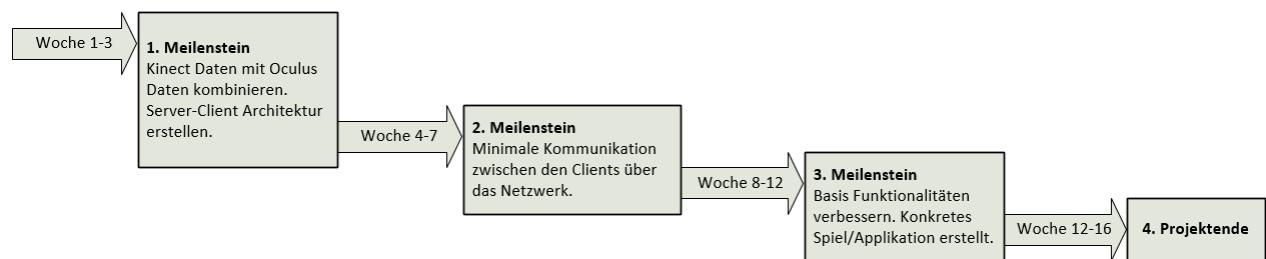


Abbildung 1: Projektplan

Wir teilen die Arbeiten für jeden Meilenstein in möglichst unabhängige Teilgebiete auf. Dies ist bei einer dreier Gruppe wichtig, damit möglichst effizient gearbeitet werden kann.

1. Meilenstein

- SLProject anpassen: Damit wir das SLProject für unsere Projektarbeit verwenden können, braucht es einige Verbesserungen, Anpassungen und Erweiterungen. Außerdem muss die gesamte Projektstruktur in unserer Entwicklungsumgebung konfiguriert werden.
- Kinect-Oculus Rift: Aus den von der Kinect und Oculus Rift gewonnenen Daten soll ein Skelett erstellt werden. Das entspricht unserem ersten Teilziel.
- Server-Client: Es soll die Grundfunktionalitäten für den Datenaustausch zwischen dem Server und Client erstellt werden. Das brauchen wir um das zweite Teilziel zu erreichen.

2. Meilenstein

- SLProject anpassen: Dieses Arbeitsgebiet begleitet uns durch die gesamte Projektarbeit. Einige von unseren Erweiterungen sollen als Fixes in das SLProject integriert werden.
- Skelett: Die gewonnenen Daten von der Kinect und Oculus Rift müssen möglichst einfach abgelegt werden. Somit können diese über das Netzwerk mit den anderen Clients ausgetauscht werden. Außerdem sollen die Skelett-Daten in einer hierarchischen Struktur abgelegt werden.
- Server-Client: Die Netzwerk Implementation soll für die konkreten Anforderungen erweitert werden. Es muss möglich sein, Skelett Daten auszutauschen. Das entspricht unserem zweiten Teilziel.

3. Meilenstein

- Interaktion: Implementation von einem konkreten Spiel / Applikation. Dort soll den Benutzern die Möglichkeit geboten werden, auf eine gemeinsame Struktur Einfluss zu nehmen.
- Server-Client: Verbesserung der Server Architektur.
- Mesh Import: Damit wir mehr haben als nur ein einfaches Skelett, soll zusätzlich ein Mesh importiert und animiert werden.

4. Projektende

Die letzten vier Wochen haben wir als Reserve eingeplant. Dort können in Rückstand geratene Teile fertig implementiert werden. Außerdem werden in dieser Zeit noch allfällige Fehler in der Software behoben.

Aus dieser Planung entsteht schlussendlich folgende Arbeitsteilung.

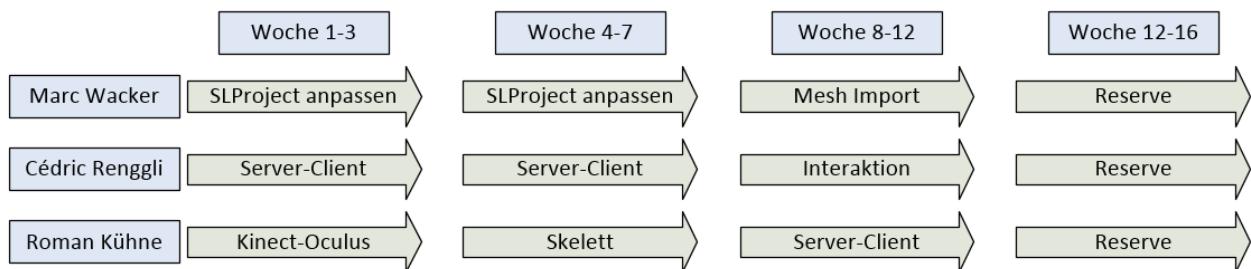


Abbildung 2: Arbeitsteilung

In dieser Abbildung ist das jeweilige Hauptarbeitsgebiet ersichtlich. Natürlich gibt es zwischen durch immer wieder kleinere Arbeiten, welche erledigt werden müssen.

Daraus ergibt sich auch ein gewisser Zyklus. Nach jedem Zusammenfügen von zwei oder mehreren Teilen muss der momentane Zustand der Software getestet werden. Gefundene Fehler werden behoben und der nächste Schritt kann geplant werden.

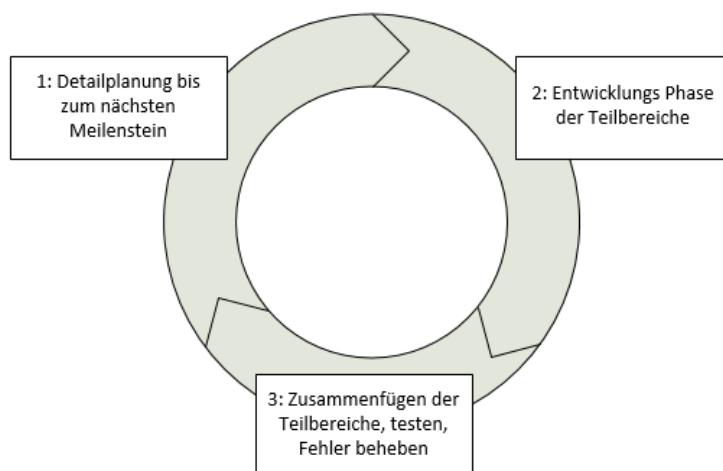


Abbildung 3: Projekt Zyklus

2. Architektur

Dieser Teil der Dokumentation beschreibt die Gesamtarchitektur sowie einen detaillierten Einblick in die einzelnen, vorwiegend hardwaremässig, verwendeten Komponenten.

2.1 Systemarchitektur

Die nachfolgende Gesamtübersicht erläutert die einzelnen Komponenten des Projekts etwas genauer. Nebst der verwendeten Hardware, sind auf dem Diagramm auch die zur Kommunikation verwendeten Schnittstellen aufgezeigt.

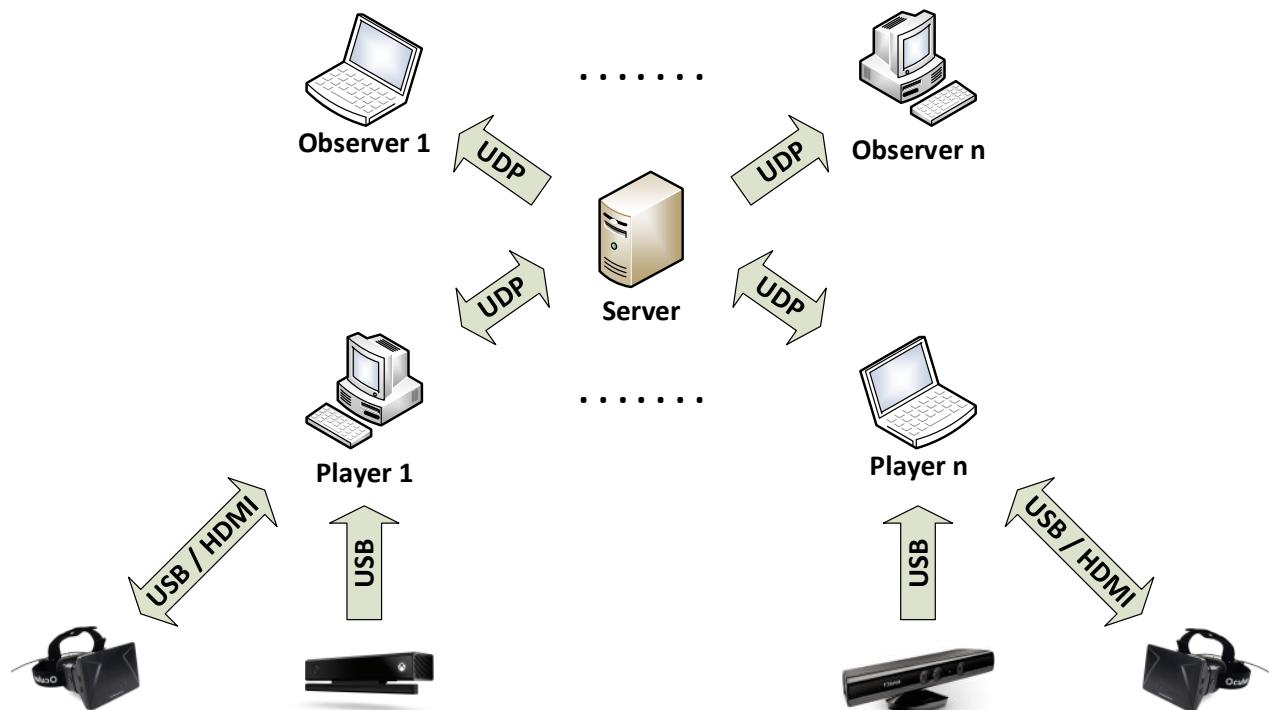


Abbildung 4: Systemübersicht

Aus der Systemübersicht ist ersichtlich, dass für die Realisierung eine Client / Server Architektur verwendet wurde. Dabei kann jeder Client entweder die Rolle eines Players oder Observers annehmen. Die beiden zusätzlich verwendeten Hardware Komponenten, die Kinect Kamera von Microsoft und die Rift VR Brille von Oculus, kommen dabei nur beim Client, respektive nur bei einem Client vom Typ Player zum Einsatz. Server und Client sind jeweils über das Netzwerk verbunden. Dies kann entweder das lokale LAN oder das Internet sein. Einzig der verwendete UDP-Port (auf dem Server konfigurierbar) muss auf den Firewalls geöffnet sein.

2.2 Client

Wie bereits erwähnt, handelt es sich beim Client entweder um einen Player oder einen Observer. Während der Player aktiv an Veränderungen im virtuellen Raum durch den eigenen Körper oder sonstige Interaktionen beteiligt ist, bekommt der Observer jeweils nur den aktuellen Zustand in regelmässigen Abständen vom Server mitgeteilt. Somit kann dieser Typ von Client die gesamte Szene aus diversen Perspektiven veranschaulichen.

Der Prozess auf einem Client kann in drei Schritte unterteilt werden.

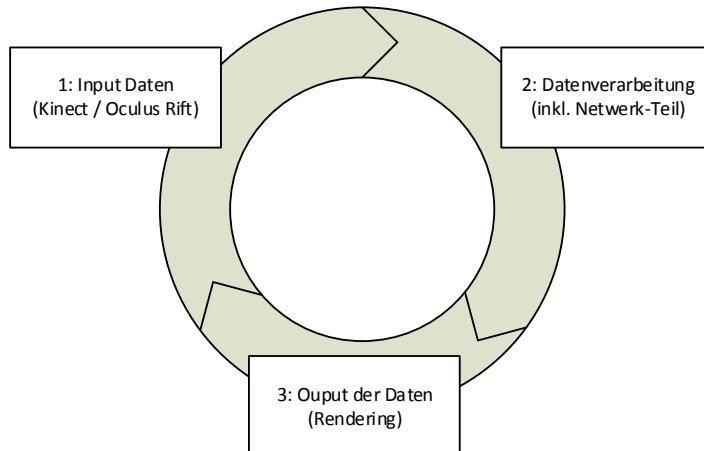


Abbildung 5: Prozess auf dem Client

Sequenziell und etwas detaillierter:

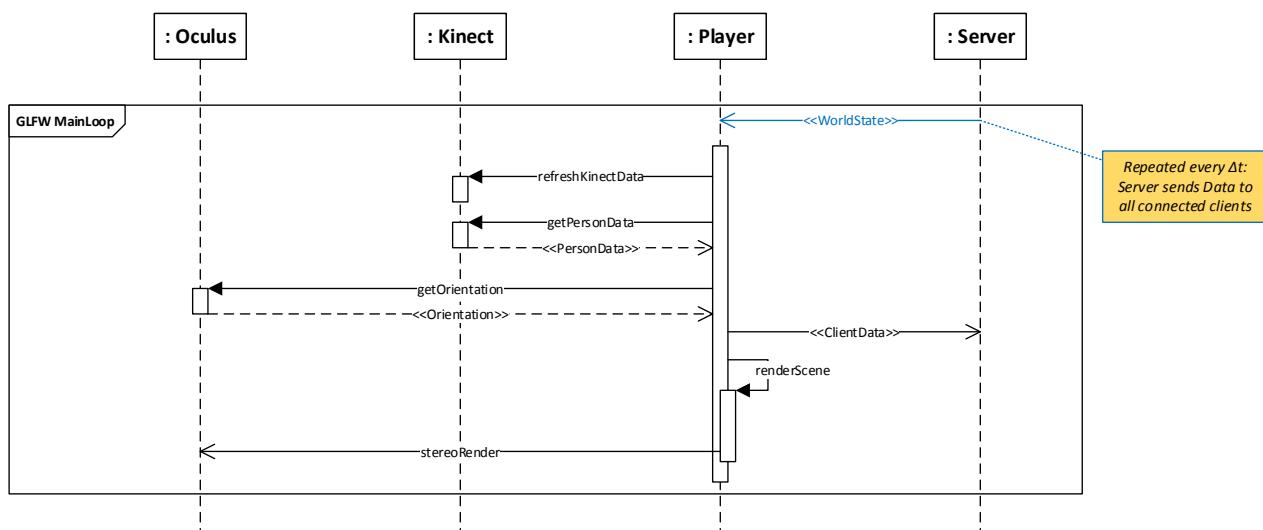


Abbildung 6: Sequenzieller Ablauf auf dem Client

2.2.1 Input Daten

Als Input Daten werden solche betrachtet, welche aktiv an Veränderungen der Szene beitragen. Dabei haben Änderungen an der Sichtweise, zum Beispiel durch die Maus oder die Tastatur, keine direkten Auswirkungen auf die Szene. Somit besitzt der Client als Observer keine Input Daten.

2.2.1.1 Kinect

Anhand der Kinect von Microsoft wird der Körper der Person vor der Kamera stehend erfasst. Dabei werden die Positionen der Gelenke und die Rotation der einzelnen Knochen in angemessene Datenstrukturen geliefert.

Kinect 1:

Beim Start des Projekts wurde ausschliesslich mit der ersten Version der Kinect gearbeitet. Diese wurde durch Microsoft für die Xbox 360 im November 2010 erstmals lanciert. Ein Infrarot Projektor

und eine horizontal versetzte Infrarot Kamera erstellen anhand von einfacher Triangulation ein Tiefenbild in VGA Auflösung (640 x 480 Pixel). Neben dieser Tiefenkamera erfasst ein RGB-Objektiv Farbbilder in derselben Auflösung.

Um auf diese Bilder zugreifen zu können, kursieren im Internet verschiedene Software Development Kits (SDKs). Zu erwähnen sind hauptsächlich die beiden Libraries OpenNI und das Kinect for Windows SDK. Ersteres wird von PrimeSense, dem Hersteller des Tiefensensors für die Kinect, entwickelt. Das Kinect for Windows SDK erschien erst über ein halbes Jahr nach dem Verkaufstart der Xbox 360 im Juni 2011. Im Rahmen dieses Projekts wurde anhand des Microsoft Produkts entwickelt. Gründe dafür waren unter anderem die bereits vorhandenen Skelett Informationen im SDK, die Notwendigkeit auch bei OpenNI die Microsoft Treiber auf einer Windows Maschine installieren zu müssen und die Hoffnung, dass keine grosse Wechsel bei der neuen Kinect auftreten würden.

Dem Client dienen keine Tiefen- oder RGB Bilder als Input Daten, sondern bereits berechnete Skelett Informationen. Das verwendete SDK bietet die Möglichkeit anhand von einer Kinect bis zu sechs Personen gleichzeitig zu erkennen, und dabei bei zwei davon alle effektiven Gelenk- und Knochen-Informationen abzufragen.

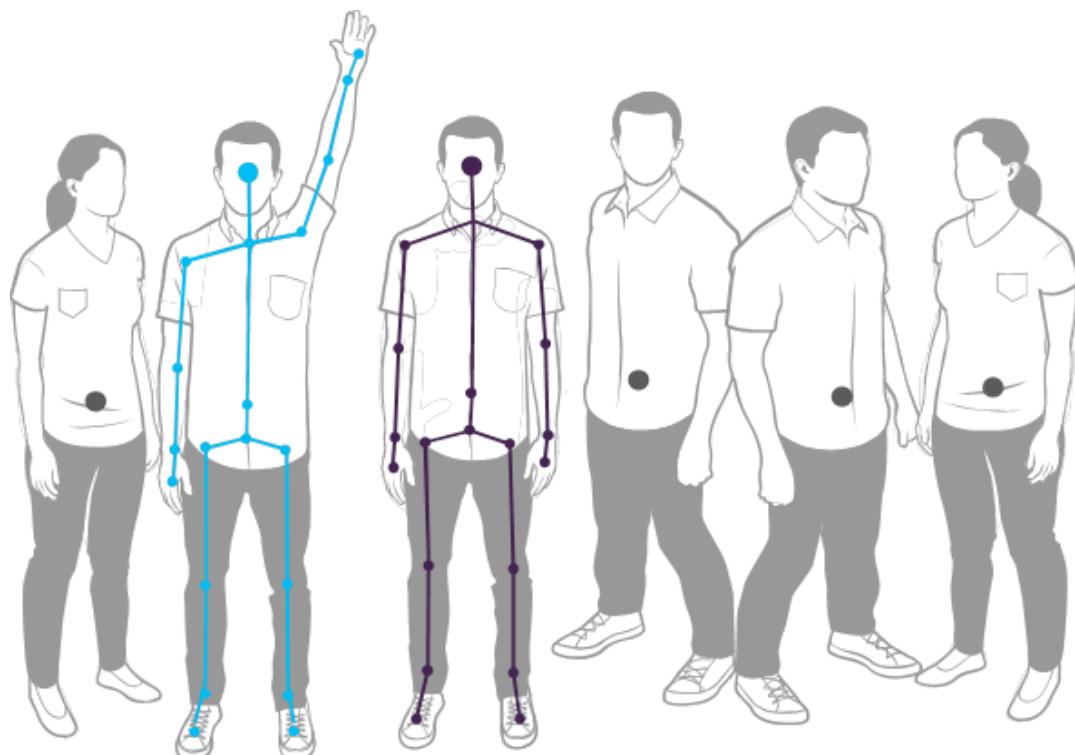


Abbildung 7: Kinect for Windows SDK: Skeletal Tracking [1]

Das SDK berechnet die Daten selbstständig anhand der durch die Kamera gelieferten Tiefen- und RGB Bilder.

Die Input Daten der Kinect werden über einen normalen USB 2 Anschluss geliefert. Durch die Verwendung des Microsoft Treibers und SDK beschränkt sich die Nutzung der Kinect auf Windows ab der Version 7, 32 oder 64 Bit.

Kinect 2:

Obwohl die Eventualität bestand, war am Anfang des Projekts noch nicht sicher, ob mit der neuen Kinect v2 im Rahmen dieses Projekt gerechnet werden konnte. Glücklicherweise sandte Microsoft per 8. April 2014 eine Developer Preview Version samt Kinect for Windows v2 SDK Preview der BFH per Post zu.

Dabei besteht mit der neuen Version des SDKs ebenfalls die Möglichkeit die Skelett Daten für eine oder mehrere Personen auszulesen. Zusätzlich zu weiteren Gelenken, erkennt die Kinect v2 den Hand-Status (geöffnet / geschlossen / auf ein Objekt zeigend), Gesichtszüge, Emotionen, Brillenträger, usw. Die verwendeten Daten wurden jedoch auf dieselben wie bei der ersten Version der Kinect reduziert. Durch die Time of Flight Technologie (ToF) für die Tiefenbild Erstellung und eine höhere Auflösung, bei sowohl dem Tiefen- wie auch dem RGB-Bild (bei selber Frequenz), erhofften wir uns etwas bessere Ergebnisse durch die Anbindung der neuen Kinect v2.

Die vorhandene Kinect 2 Developer Preview Hardware und Software ist nur für Windows 8 und 64 Bit OS vorhanden. Zusätzlich werden Daten nur übertragen, wenn die Kamera über einen USB 3 Anschluss angeschlossen ist.

2.2.1.2 Oculus Rift

Als Zweites Input Device für einen Player dient das Oculus Rift Dev Kit. Die Daten der VR-Brille von Oculus können über das mitgelieferte Oculus Rift SDK [2] ausgelesen werden. Eine detailliertere Beschreibung der Brille und des im SDK enthaltene und verwendete LibOVR wurde im Rahmen der Vorarbeit „Projekt 2: Oculus Rift“ (siehe Anhang II) bereits beschrieben.

Als Input Daten der Rift dient lediglich die aktuelle Rotation des Kopfes der Person, welche die Brille trägt. Diese Rotation wird als Quaternion über eine USB 2.0 Schnittstelle geliefert.

2.2.2 Datenverarbeitung

Die Datenverarbeitung auf dem Client beinhaltet zwei Komponenten. Dabei werden zum einen die durch die zwei Input-Quellen gewonnene Daten für die direkte Darstellung und das Versenden über das Netzwerk verarbeitet. Dafür werden die Informationen in eine gemeinsame hierarchische Datenstruktur abgelegt. Dieser Schritt wird wiederum ausschliesslich vom Player durchgeführt. Als Zweites werden die Daten vom Server über das Netzwerk empfangen. Diese Daten sind schon im Sinne der Anzeige aufbereitet und können mit minimalem Aufwand von Player und Observer direkt visualisiert werden. Vor der effektiven Anzeige der Daten wird der vom Player verarbeitete Input dem Server über das Netzwerk übermittelt.

2.2.3 Output der Daten

Die Ausgabe der Szene wird je nachdem ob eine Oculus Rift am PC angeschlossen ist oder nicht in einem Mono- (keine Rift angeschlossen) oder einem Stereo-Rendering (mit Rift) Prozess durchgeführt. Mit dieser Unterscheidung kann sowohl ein Player wie auch ein Observer ohne die VR-Brille gestartet werden. Die Szenen für den Player wird jeweils aus der Ich Perspektive angezeigt. Der Observer bekommt über die Nummerntasten die Möglichkeit zwischen den Sichten der verschiedenen Player, einer bewegbaren Third-Person-View Kamera, und einer animierten Third-Person-View Kamera zu wechseln.

2.2.3.1 Oculus Rift

Die Spezifikationen für das Stereo-Rendering sowie der Aufbau der Displays in der Oculus Rift wurden im Rahmen der Vorarbeit „Projekt 2: Oculus Rift“ (siehe Anhang II) beschrieben.

Die VR-Brille wird über einen normalen Display-Output Anschluss wie HDMI, VGA oder DVI angeschlossen.

2.3 Server

Der Server ist für die Verwaltung der Clients und die Berechnung des World State (Zustand der Szene) zuständig. Dabei empfängt die Applikation die Daten der Player, simuliert Änderungen an der Welt und sendet danach den Zustand an alle Clients (Players und Observers). Dieser Zyklus folgt einer vordefinierten Tickrate.

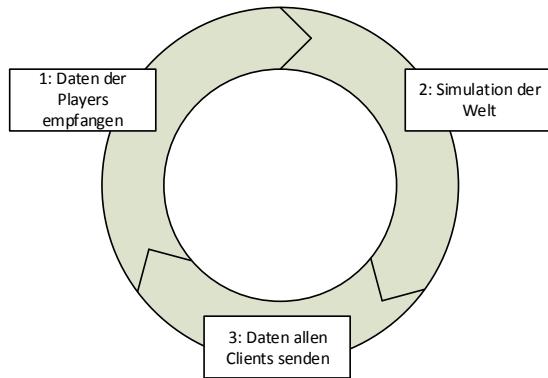


Abbildung 8: Prozess auf dem Server

2.3.1 Player / Observer Verwaltung

Eine der Kernaufgaben des Servers ist die Verwaltung aller verbundenen Clients. Dabei wird beim Starten des Servers jeweils eine maximale Anzahl verbundener Player und Observer definiert. Wird ein Client gestartet, versucht dieser eine Verbindung zum angegebenen Server (mittels IP-Adresse und Port) herzustellen. Dabei gibt der Client noch an, ob er als Player oder Observer fungieren wird. Bekommt der Server eine solche Anfrage, wird überprüft ob die maximale Anzahl des geforderten Typen schon erreicht ist. Tritt dieser Fall ein, wird die Verbindungs-Anforderung abgelehnt. Andererfalls wird der Client samt Typ registriert und die Verbindung wird akzeptiert. Von diesem Zeitpunkt an bekommt der Client den World-State nach jedem Simulations-Zyklus, siehe Abbildung 8, über das Netzwerk zugesandt. Wird ein neuer Player registriert, wird der Server dem Client eine spezifische Position und Orientierung (als Rotation) innerhalb des virtuellen Raums zuweisen. Zusätzlich wird für den Player ein bestimmtes Modell zur Anzeige bestimmt. Diese Informationen werden dem Client in einem Bestätigungs-Paket mitgeteilt.

Die Verwaltung der Clients beinhaltet ebenfalls das Schliessen der offenen Verbindungen. Dafür gibt es grundsätzlich drei verschiedene Szenarien:

- Der Server wird beendet: Alle offenen Verbindungen werden geschlossen.
- Ein Client wird korrekt beendet: Vor dem Schliessen des Programms führt der Client ein Disconnect durch. Dadurch wird auf dem Server wieder ein Player oder Observer Platz frei.
- Ein Client wird unerwartet beendet oder getrennt: Nach einem definierten Timeout wird auf dem Server automatisch ein Disconnect des Clients simuliert. Dabei kann der Server den Platz für einen anderen Player oder Observer freigeben.

Nachfolgend ein möglicher sequenzieller Ablauf mit zwei sich verbindenden Clients (ein Observer und ein Player):

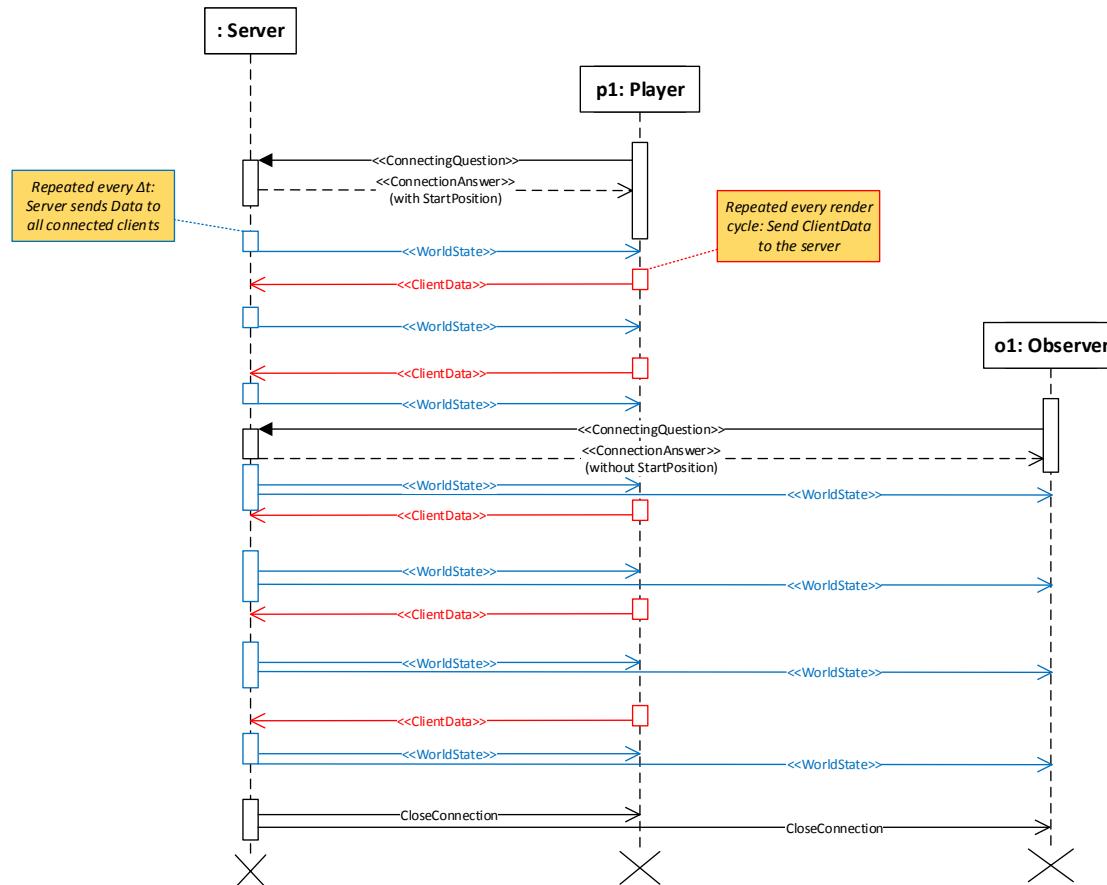


Abbildung 9: Sequenzieller Ablauf - Client Verwaltung

Aus der obigen Abbildung ist ebenfalls der Unterschied der beiden Client-Typen beim Senden von Daten zum Server ersichtlich.

2.3.2 Datenverarbeitung / Simulation

Wie in Abbildung 8 ersichtlich, wird in einem zyklischen Vorgang auf dem Server zwischen dem Empfangen der Client Daten und dem Versenden des World States, eine Datenverarbeitung resp. Simulation durchgeführt. Diese umfasst im Wesentlichen folgende Tätigkeiten:

- Personen- und Rotations-Daten der jeweiligen Player anhand der Client Paket Daten aktualisieren
- Simulation basierend auf dem gestarteten Server-Typen
- Warten um eine konstante Tickrate auf dem Server zu erreichen

Die Simulation ändert die Szene meist nur wenn Personen-Daten (aktuelle Gelenk-Positionen) sich an gewissen Stellen der Szene befinden (Hit-Detection). Dafür müssen jedoch die erhaltenen relativen Positionen der Gelenke anhand der Start-Position und -Rotation des Players an den korrekten Ort in der Welt verschoben / rotiert werden. Zusätzliche Veränderungen an der Szene können durch aktive Simulationen wie das Fallen einer Kugel in einem Board-Game bewirkt werden.

Das Warten vor dem Senden der Welt-Daten zum Client um eine konstante Tickrate zu erreichen hat mehrere Gründe. Zum einen würde der Server ohne das Warten mit einer zu hohen CPU-Last laufen. Zusätzlich könne saubere, flüssige Bewegungen von Objekten, wie das Fallen von Kugeln, nur so erreicht werden. Zusätzlich kann der Client nur mit einer begrenzten Framerate rendern. Daher muss

der Server ebenfalls nicht mit einer höheren Tickrate als 16ms (entspricht etwas mehr als 60 Frames pro Sekunde [FPS]) arbeiten.

3. Entwicklungsumgebung

Im Rahmen dieser Bachelor Arbeiten wurden diverse Tools und Libraries eingesetzt. Dieses Kapitel gibt einen Überblick darauf und zeigt ebenfalls die grundlegende Projektstruktur.

3.1 Libraries / SDKs

Zum Realisieren des Projektes wurden mehrere Libraries und Frameworks verwendet. Nachfolgend werden diese kurz samt Funktionalität vorgestellt.

3.1.1 SLProject

SLProject ist ein plattformunabhängiges Rendering-Framework mit Szenengraph. SL steht für „Scene Library“, und die Hauptfeatures sind die Darstellung und Organisation von verschiedenen Szenen. Des Weiteren bietet das Framework Ray-und Path Tracing Funktionalität, um die Szenen realistisch zu rendern. Das Projekt wurde ursprünglich an der FHNW entwickelt, und wird momentan an der BFH weiterentwickelt. Die Schnittstelle zur Oculus Rift SDK wurde bereits in einem früheren Projekt implementiert, und ist ein fester Bestandteil des SLProjects.

Für den Virtual Room wurden vor allem der Szenengraph und das Rendering durch OpenGL verwendet. Damit das SLProject unseren Anforderungen entspricht, haben wir einige Änderungen vorgenommen. Diese sind im Kapitel 4.1 Anpassungen SLProject beschrieben.

Um aus den C++ Klassen und Funktionen eine geeignete Dokumentation automatisch erstellen zu lassen, wurde im SLProject auf das Tool Doxygen von Dimitri van Heesch gesetzt [11]. Dabei müssen die Kommentare gewisse Regeln entsprechen. Anschliessen kann die Ausgabe automatisch in den Formaten HTML, Latex, RTF, Man oder XML erfolgen. Um diese Möglichkeit im Rahmen unserer Arbeit weiterhin nutzen zu können, wurde ebenfalls auf eine Doxygen konforme Kommentierung des Codes geachtet.

3.1.2 Kinect for Windows SDK v1

Um die Kinect zu verwenden, muss zuerst ein Treiber installiert werden. Dieser wird beim Download des SDKs mitgeliefert. Das SDK dient als Interface zur Kamera. In der SDK werden bereits viele wichtige Berechnungen vorgenommen. Für dieses Projekt werden die berechneten Skelett Daten verwendet.

3.1.3 Kinect for Windows SDK v2

Die Kinect v2 ist offiziell noch nicht erhältlich. In diesem Projekt wird deshalb mit der Entwicklerversion gearbeitet. Wie auch für die Kinect v1 muss zuerst ein Treiber installiert werden. Die Verwendung der SDK unterscheidet sich zur v1, aber beinhaltet die gleichen Grundfunktionalitäten. Die Kinect v2 SDK ist nur ab Windows 8 und im 64-Bit Modus kompatibel.

3.1.4 RakNet

RakNet ist eine C++ Open Source, Cross Plattform Netzwerk Enging, welche vorwiegend von Game-Developer verwendet wird. Dabei wird der Fokus hauptsächlich auf eine hohe Performance und einfache Integrierbarkeit gelegt. Nebst den ausgereiften Low-Level Funktionen, bietet die Library auch

High-Level Funktionalitäten wie das Replizieren von Game Objekte. Im Rahmen des Projekts wurde vorwiegend auf die robuste Kommunikations-Schnittstelle gesetzt.

3.1.5 Assimp

Die „Open Asset Import Library“ ist eine Open Source Library für das Laden von verbreiteten 3D-Formaten in einer einheitlichen Datenstruktur. Viele Formate unterstützen ganze Szenen mit mehreren Objekten und Animationen. Ausserdem bietet Assimp die Möglichkeit importierte Daten in einem Postprocessing-Schritt anzupassen, zum Beispiel grosse Modelle in mehrere kleinere aufzuteilen, oder das Generieren von Normalen. In diesem Projekt wird Assimp zum Importieren von einem rigged Mesh verwendet.

3.2 MS Visual Studio 2012 Projekt

Das Projekt wurde unter Windows (7, 8, 8.1) entwickelt und getestet. Andere Plattformen wurden in der Entwicklung nicht berücksichtigt. Das Zusammenführen der einzelnen Komponente wurde schliesslich in Visual Studio 2012 gemacht. Das Projekt kann grundsätzlich in 32- und 64-Bit erstellt werden. Es gibt nur kleine Einschränkungen, welche bereits bei den jeweiligen Libraries beschrieben wurden.

3.2.1 SVN

Für die Versionisierung der Projektdateien wird SVN verwendet. Das SLProject verfügt bereits über ein Repository bei Google Code [3]. Deshalb wurde in diesem Repository ein neuer Branch vom aktuellen Stand des SLProjects erstellt. Für die Verwaltung der Dateien in der lokalen Entwicklungsumgebung haben wir TortoiseSVN [5] (zum Verwalten im Windows Explorer) und AnkhSVN [6] (Visual Studio Erweiterung) verwendet.

3.2.2 Projekt Struktur

Das finale Virtual Room Projekt besteht aus mehreren Komponenten, welche in unterschiedlichen Visual Studio Projekte unterteilt wurden. Es folgt eine Liste der Projekte aus der Virtual Room Visual Studio Solution.

3.2.2.1 Lib-RakNet

Beinhaltet den unveränderten RakNet Quellcode. Wir kompilieren alle Open Source Libraries selbst, um keine unnötigen DLLs zu verwenden.

3.2.2.2 CG-External

Enthält alle externen Libraries, welche von SLProject verwendet werden. Assimp wurde ebenfalls in dieses Projekt integriert, da es stark zum Szenengraphen gehört. Die Importfunktionalität wurde von uns direkt im SLProject implementiert.

3.2.2.3 Lib-SLProject

Eine angepasste Version von SLProject mit etwas mehr API Funktionalitäten.

Abhängigkeiten: CG-External.

3.2.2.4 Lib-VirtualRoomCommon

Beinhaltet alle Klassen, welche von Client und Server in gleichem Masse benötigt werden. Gemeinsame Datentypen und Netzwerkpakete machen den Hauptteil von diesem Projekt aus.

Abhängigkeiten: Lib-SLProject, Lib-RakNet.

3.2.2.5 VirtualRoom

Die Client Implementation wurde im Virtual Room Projekt untergebracht. Input von Kinect und Oculus Rift, sowie die Darstellung der Daten geschehen hier unter Verwendung vom SLProject als Framework.

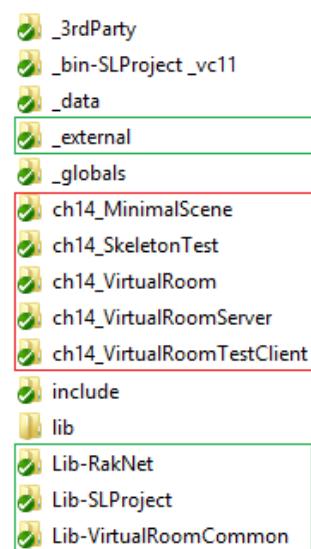
Abhängigkeiten: Lib-VirtualRoomCommon.

3.2.2.6 VirtualRoomServer

Die Server Implementation, welche sich um die Simulation der Applikationen kümmert, befindet sich im gleichnamigen Projekt.

Abhängigkeiten: Lib-VirtualRoomCommon.

3.2.3 Ordnerstruktur



Im Ordner „_3rdParty“ befinden sich alle externen Libraries, welche nicht im Projekt CG-External benötigt werden. Konkret sind das die beiden Libraries Kinect for Windows SDK und RakNet.
Die verwendeten Texturen und Modelle sind im „_data“ Ordner untergebracht.
Die im SLProject benötigten externen Libraries befinden sich im Ordner „_external“.
In dem „_globals“ Ordner befinden sich die unveränderten globalen Headerfiles der externen Libraries vom originalen SLProject.
Das „include“ Verzeichnis enthält die Headerfiles vom SLProject.

Die Verzeichnisse in der roten Box entsprechen dem jeweiligen Projekt im Visual Studio. Dort befinden sich alle C++ Dateien. Der Ordner „_bin-SLProject_vc11“ ist das Output Verzeichnis für die kompilierten ausführbaren Dateien.

Die Verzeichnisse in den grünen Boxen entsprechen den vier Library Projekte. Beim Kompilieren wird jede erstellte Library im Ordner „lib“ abgelegt.

Abbildung 10: Ordnerstruktur

3.2.4 Visual Studio Konfigurationen

- Die Abhängigkeiten der einzelnen Projekte sind in den Solution Properties konfiguriert: „Solution->Properties->Common Properties->Project Dependencies“.
- Die Kinect v2 SDK ist erst ab Windows 8 kompatibel. Unter älteren Windows Versionen dürfen deshalb keine Referenzen zu dieser SDK existieren. Damit die Kinect v2 verwendet werden kann, muss in den Client Projekt Properties die Präprozessor Definition „ISWIN8“ hinzugefügt werden. Bei nicht kompatiblen Windows Versionen muss diese Definition zwingend entfernt werden, ansonsten kann das Projekt nicht kompiliert werden.

- Die Client Implementation vom Virtual Room kann über Kommandozeile Argumente gewisse Konfigurationen entgegennehmen. Der erste Parameter definiert den Client Typ. Es gibt die Typen „Player“ (Typ: 0) und „Observer“ (Typ: 1). Der zweite Parameter ist die IP-Adresse vom Server. Werden die Parameter nicht gesetzt, werden die Standard Parameter 0 und localhost verwendet. Im Visual Studio können diese Parameter direkt gesetzt werden:
„ch14_VirtualRoom->Properties->Configuration Properties->Command Arguments“
- Die Server Applikation kann ebenfalls über Kommandozeile Argument gestartet werden. Dabei gibt der erste Parameter die zu startende Game Implementation an. Die Implementation ist an die Game Typen gekoppelt: 0 – Leere Game Szene, 1 – Color Sphere Game, 2 – Tic Tac Toe, 3 – 3D Vier-Gewinnt und 4 – Turn Table. Als zweites Argument kann spezifiziert werden, wie viele Spieler in den Virtual Room gesetzt werden sollen. Die beiden Brettspiele Tic Tac Toe und das Vier-Gewinnt würden eine allfällige Anzahl ignorieren, da bei diesen Games immer exakt zwei Spieler gegeneinander antreten. Werde keine Parameter dem Server als Argumente übergeben, startet der Server die leere Game Szene mit zwei Personen.

4. Implementation

In diesem Kapitel wird im Detail auf die Implementationen von diesem Projekt eingegangen. Es soll dazu dienen alle Abhängigkeiten und technischen Aspekte zu verstehen. Gerade der Client-Seitige Teil ist stark an das SLProject [3] gebunden. Deshalb muss ein Basiswissen über das SLProject [3] vorhanden sein, um alle Abschnitte gänzlich zu verstehen.

4.1 Anpassungen SLProject

Für dieses Projekt mussten mehrere Anpassungen am SLProject vorgenommen werden. Diese Anpassungen sind nötig, um allen Projekt Anforderungen gerecht zu werden.

4.1.1 Ausgangslage

Der Aufbau vom SLProject zur Zeit dieser Arbeit sieht wie folgt aus:

1. External: Externe Libraries.
2. TextureMapping: Ein Projekt welches die OpenGL Wrapper-Klassen verwendet aber keinen Szenengraph.
3. Final: Hauptprojekt mit Szenengraph und mehreren Szenen zum Durchschalten.

Die angesprochenen OpenGL Wrapper-Klassen befinden sich in einem Ordner auf der obersten Ebene. Die Szenengraph und Rendering Implementationen befinden sich im Final Projekt. Das Final Projekt hat jedoch bereits eine Menge an Beispielszenen und Navigation implementiert, was es stark erschwert das SLProject direkt als Framework zu verwenden.

Ein erstes Ziel war es also das SLProject so anzupassen, dass es als Framework verwendbar ist. Um das Framework zu benutzen, soll die Funktion `SLScene::onLoad` implementiert werden. In dieser Funktion wird die Szene gebaut. Zusätzlich wird eine Klasse erstellt, welche von der `SLSceneView` erbt. In dieser Klasse können alle wichtigen Funktionen überschrieben werden, um in die Grundfunktionalitäten des Frameworks einzugreifen.

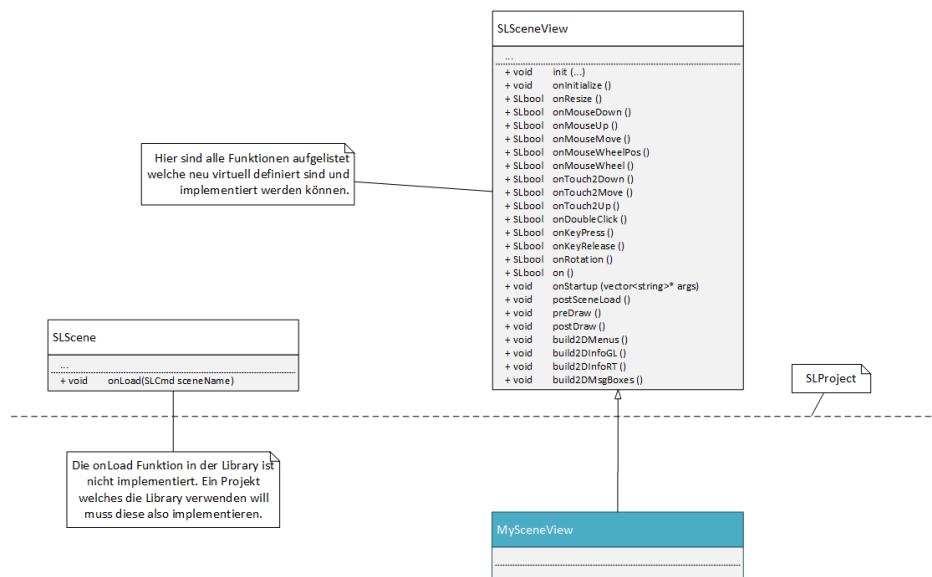


Abbildung 11: SLProject Anpassung

4.1.2 Änderungen an SLSce

Um eigene Szenen im Projekt „Final“ einzufügen, kann man in der `SLSce::onLoad` Funktion eine Szene generieren. Wir entfernen also die Implementation von `onLoad` aus unserer SLProject Library, was nun bedeutet, dass ein Projekt welches SLProject erweitern will die `onLoad` Funktion von `SLSce` implementieren muss.

4.1.3 Änderungen an SLSceView

Die zweite Änderung war es `SLSceView` abstrakt zu machen. `SLSceView` implementiert den Hauptteil des Szenengraphen, den Rendering-Loop, das Frustumculling sowie auch Inputevents werden von der Klasse verarbeitet. Es wurden also alle Inputeventhandler zu virtuellen Funktionen umgewandelt. Die Render Funktion `onPaint` wurde nicht virtuell gemacht, jedoch wurden folgende virtuellen hook Funktionen eingebaut:

- `onStartup` – wird nach der Initialisierung von `SLSceView` aufgerufen.
- `postSceneLoad` – wird nach `SLSce::onLoad` aufgerufen.
- `preDraw` – wird vor `updateAndDraw3D` aufgerufen.
- `postDraw` – wird nach `updateAndDraw3D` aufgerufen.

Die Parameter `name`, `screenWidth`, `screenHeight` und `dotsPerInch` wurden vom Konstruktor zur `init` Funktion verschoben, um eine simplere Vererbung zu ermöglichen. Die `init` Funktion erhält zusätzlich noch eine Liste der Kommandozeile Argumente, falls eine Applikation diese nutzen will.

4.1.4 Änderungen an SLInterface

In `SLInterface.h` befinden sich die C Funktionen, welche `SLSceView` und `SLSce` instanziieren, und Systemevents an diese weiterleiten. Die Instanziierung der `SLSceView` Klasse passiert in der Funktion `sllInit`. Damit `sllInit` die korrekte Subklasse von `SLSceView` instanziert wurde der direkte Aufruf des Konstruktors mit einer „extern“ Funktion ersetzt, welche einen Pointer auf `SLSceView` zurück gibt.

```
extern SLSceView* getSceneView();
```

Ein neues Projekt, welches SLProject verwenden will, muss diese Funktion also implementieren und eine Instanz der verwendeten `SLSceView` Subklasse zurückgeben:

```
class MySceneView : public SLSceView
{
    //...
}
```

```
SLSceView getSceneView()
{
    return new MySceneView;
}
```

4.1.5 Manipulation der aktiven Kamera

Die `SLCamera` Klasse erbt ebenfalls von `SLShape` und kann in den Szenengraphen gehängt und animiert werden. Wenn diese Kamera jedoch aktiv ist, werden jegliche Änderungen auf dieser Kamera ignoriert. Um die Möglichkeit zu geben, dass auch die aktive Kamera verändert wird, passen wir die View-Matrix der Kamera an, falls die Parent Node der Kamera geändert wurde. Dies hat den Nachteil, dass eine Kamera, welche sich in einer animierten Node befindet, nicht mehr gesteuert werden kann. Um diese Funktionalität ebenfalls zu ermöglichen, müsste eine andere Lösung gefunden werden.

Zudem wurde die Ordnerstruktur für das Projekt angepasst. Viele Headerfiles waren exklusiv im Final Projekt, diese sind jedoch Kernklassen von SLProject wie Nodes oder gar die Szene selbst. Damit diese Klassen nutzbar sind, wurden die Headerfiles welche sich im Final Projekt befanden in einen Ordner „include“ auf oberster Ebene verschoben. Die Headerfiles, welche sich im Ordner „globals“ befinden, wurden an ihrer Position belassen. Es besteht jedoch die Möglichkeit diese ebenfalls in den „include“ Ordner zu verschieben, um das Projekt einheitlich zu halten. Die finale Ordnerstruktur wurde bereits im Kapitel 3.2.3 Ordnerstruktur beschrieben.

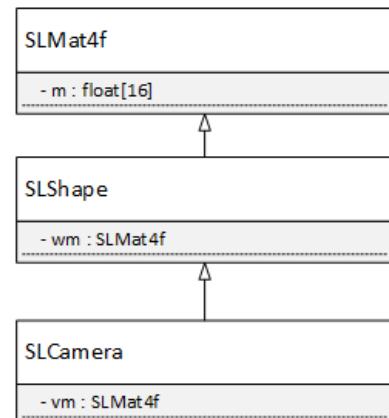


Abbildung 12: Kamera - Klasse

4.2 Skelett Visualisierung

Dieser Abschnitt beschreibt die Visualisierung des Skelettes. Vom Auslesen der Skelett Rohdaten von der Kinect, bis zur Darstellung am Bildschirm, sind einige Schritte nötig. Im folgenden Klassendiagramm sind alle wichtigen Klassen dargestellt, welche in diesem Prozess involviert sind. Diese Architektur kann in drei Teile getrennt werden. Der blaue Teil repräsentiert die aktuelle Scene und ist das Kernstück. Der grüne Teil ist für das Auslesen der Rohdaten aus der Kinect verantwortlich. Im orangen Teil werden die Daten von der Kinect verarbeitet, bevor sie zur Darstellung des Skelettes verwendet werden.

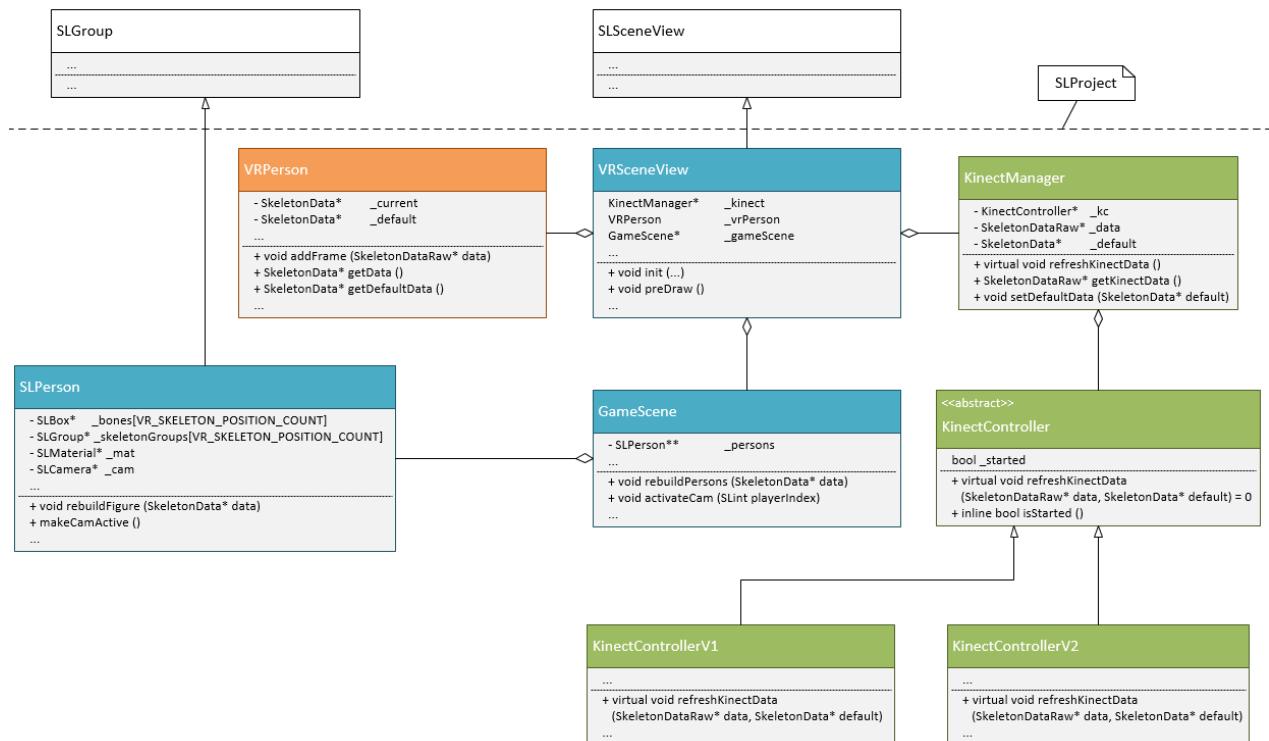


Abbildung 13: Klassendiagramm, Skelett Visualisierung

4.2.1 Kinect Daten

Die Klasse `KinectManager` dient als Schnittstelle zur Scene. Die Hauptaufgabe ist das Laden des richtigen Kinect Controllers. Ausserdem müssen über diese Klasse die aktuellen Kinect Daten geholt werden. Im Konstruktor wird der Kinect Controller erstellt. Dort wird entschieden ob die Kinect V1 oder V2 verwendet werden soll. Es gibt verschiedene Gründe warum jede Version einen eigenen Kontroller hat. Der Hauptgrund sind die völlig verschiedenen SDKs. Beide Versionen verwenden eine eigene SDK, welche sich komplett unterscheidet. Die Art und Weise wie das Kinect Device initialisiert werden muss, wie auch das Zugreifen auf die Daten, ist deutlich anders. Ausserdem ist die Kinect 2 nur ab Windows 8 kompatibel. Deshalb kann Programmcode, welcher auf die SDK zugreift nur ab Windows 8 kompiliert werden. Um dieses Problem zu beheben, ist der Kinect 2 Teil mit einem „#ifdef ISWIN8“ umgeben. Ist „ISWIN8“ bei den Präprozessor Definitionen vorhanden, wird die Kinect 2 verwendet. Kann diese nicht initialisiert werden (z.B. Ist nicht angeschlossen), oder ist „ISWIN8“ nicht definiert wird die Kinect 1 verwendet.

Um nun die Daten zu holen, gibt es zwei wichtige Funktionen. Durch den Aufruf der Funktion `refreshKinectData` werden die aktuellen Daten der Kinect ausgelesen. Diese Funktion sollte nur einmal pro Rendervorgang aufgerufen werden. Die ausgelesenen Daten werden in der Membervariable `_data` gespeichert. Über die Funktion `getKinectData` kann auf diese Daten zugegriffen werden.

4.2.2 Skelett Daten

In den Kontroller Klassen befinden sich alle SDK Abhängigkeiten. Die Kinect muss initialisiert werden, und anschliessend werden kontinuierlich die Skelett Daten ausgelesen. Für das Auslesen der Daten wird vom Kinect Manager jeweils die Funktion `refreshKinectData` aufgerufen. Die Kinect kann mehrere Personen gleichzeitig erkennen. Für diese Applikation werden aber lediglich die Daten von einer Person benötigt. Deshalb werden nur die Skelett Daten von der Person, welche als Erstes von der Kinect erkannt wurde, genommen.

4.2.2.1 Skelett Daten von der Kinect

Die Skelett Daten werden in einer Liste von Knochen/Gelenken geliefert. Diese Liste enthält pro Knochen/Gelenk die Position, die Rotation, den „Tracking State“ und die Information über den Elternknochen. Die Liste der Kinect 2 SDK ist ein wenig anders aufgebaut und verfügt ausserdem über mehr Gelenke als die Kinect 1. In der folgenden Abbildung ist der hierarchische Aufbau der Skelette zu sehen.

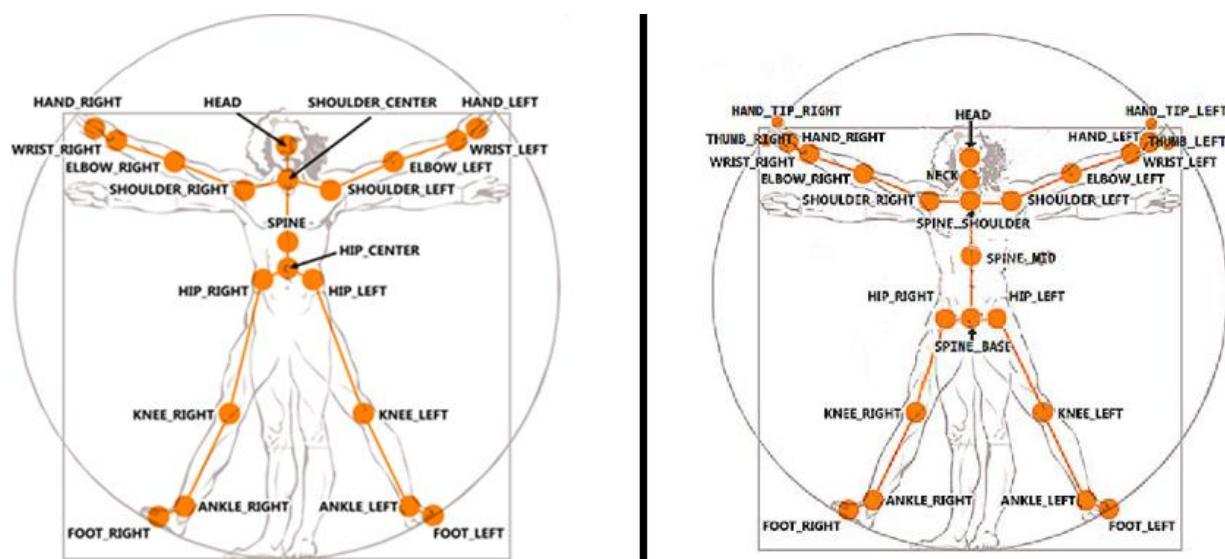


Abbildung 14: links Kinect 1, rechts Kinect 2, Quelle: Kinect for Windows SDK: Skeletal Tracking [1]

Bei der Kinect 2 verfügt das Skelett über mehr Gelenke an den Händen und einem zusätzlichen Gelenk zwischen dem Kopf und dem Schulterzentrum. Ausserdem sind die Positionen, der im Bauchbereich befindenden Gelenke, unterschiedlich.

Beispiel „KNEE_RIGHT“:

- Position: Dabei handelt es sich um die relative Position von der Kinect aus gesehen. Die Kinect selber hat die Position (0, 0, 0).
- Rotation: Die Rotation beschreibt den Knochen, welcher sich zwischen dem Gelenk („KNEE_RIGHT“) und seinem Elterngelenk („HIP_RIGHT“) befindet. Also ist es eigentlich die Rotation am Elterngelenk.
- Tracking State: Jedes Gelenk kann drei verschiedene Zustände haben. Komplett erkannt („tracked“), nur teilweise erkannt, oder gar nicht erkannt. Dieser Zustand kann zum Filtern der Daten verwendet werden.
- Elterngelenk: Dabei handelt es sich lediglich um den Listenindex des Elterngelenkes.

4.2.2.2 SkeletonDataRaw

Die von der Kinect erhaltenen Daten werden in der Datenstruktur `SkeletonDataRaw` gespeichert. Diese dient als einheitliche Struktur der Rohdaten für die Kinect 1 und 2. Sie besteht aus den folgenden drei Werten:

- „`position`“: Die Position vom "root" Gelenk. Damit wird der Standort des Skelettes definiert. Dabei handelt es sich um „HIP_CENTER“ bzw. „SPINE_BASE“.
- „`headRotation`“: Dabei handelt es sich um ein Quaternion, welches die Rotation des Kopfes beschreibt. Diese Rotation wird für die Kamera verwendet. Diese Rotation wird im Skelett Kontroller nicht gesetzt. Dies geschieht erst in einem späteren Schritt in der Klasse `VRPerson`. Dazu mehr im nächsten Kapitel.
- „`bones`“: Das ist eine Liste mit allen vorhandenen Knochen der Kinect 2. Im Kontroller der Kinect 1 werden die Knochen, welche nur in der Kinect 2 vorhanden sind, leer gelassen. Für jeden Knochen werden folgende Daten gespeichert: Elternknochen Index, relative Rotation, absolute Rotation, Länge und der Zustand.

4.2.3 Daten aufbereiten

Die von der Kinect gewonnenen Skelett Daten werden nicht direkt für die Anzeige verwendet. Diese werden zuerst in der Klasse `VRPerson` verarbeitet. Die Hauptaufgabe ist das Speichern der Daten über eine gewisse Zeit. Die Skelett Daten der Kinect werden über die Funktion `addFrame` dieser Klasse übergeben. Dort werden die gelieferten Daten in eine mehrdimensionale Struktur abgelegt. Diese Struktur dient als Grundlage für den Einbau von Filtern. Nach jedem hinzufügen von neuen Rohdaten aus der Kinect, werden die neuen effektiven Daten für die Darstellung berechnet. Über die Funktion `getData` können die aktuellen Daten für die Darstellung ausgelesen werden.

4.2.3.1 Daten hinzufügen / filtern

Beim Hinzufügen der Daten wird bereits ein einfacher Filter angewendet. Jeder Knochen wird einzeln in die Struktur gespeichert. Dies geschieht aber nur wenn der Knochen den Zustand „tracked“ hat. Damit sollen extreme Positionswechsel (Zuckungen) von Knochen verhindert werden. Weil wenn der Knochen nicht „tracked“ ist, wird die Position/Rotation oft ungenau geliefert.

Eine weitere Filterung geschieht durch das Sortieren der Daten. Die Daten werden nach ihrem Wert sortiert. Somit ist es einfach möglich kleine und/oder grosse Extremwerte zu ignorieren. Oder einfach immer den Median zu verwenden. Momentan wird lediglich der neuste Wert (mit dem Zustand „tracked“) verwendet. Wir haben bei Versuchen festgestellt, dass beim Verwenden von älteren Daten eine gewisse Verzögerung bei den Bewegungen festzustellen ist. Die Verbesserung der Qualität der Bewegungen war zu klein und die Verzögerung zu gross.

Um solche Filter effektiv einzusetzen, müsste man zusätzlich noch eine Berechnung in die Zukunft einbauen. Dabei gilt zu beachten, dass bei schnellen Bewegungen, mit Richtungswechseln, eine Berechnung in die Zukunft sehr ungenau werden kann. Leider hatten wir keine Zeit mehr solche Filter zu implementieren und zu testen. Weitere Informationen zum Thema Filterung von Skelett Daten gibt es auf der Seite der „Kinect for Windows SDK“ [1].

Knochenlänge

Damit die Körpergrösse möglichst realistisch wirkt, werden die Positionsdaten der Gelenke von der Kinect verwendet. In einem ersten Schritt haben wir die Knochenlängen bei jedem Update der Daten neu berechnet. Dabei haben wir festgestellt, dass die Knochenlängen stark variieren können. Die Ursache sind fehlerhafte Positionsdaten von einzelnen Gelenken. Die Veränderung von Knochenlängen wirkt extrem unnatürlich, deshalb haben wir die Berechnung angepasst. Die Knochenlängen werden über einen definierten Zeitraum gesammelt. Die effektive Knochengrösse wird anschliessend aus dem Durchschnitt dieser gesammelten Daten berechnet.

Positionierung

Die gelieferten Positionsdaten der Gelenke sind relativ zur Kinect. Wie bereits beschrieben hat die Kinect selber die Position (0, 0, 0). Damit das Skelett im Virtual Room auf dem Boden steht, wird der

tiefste Punkt (rechter oder linker Fuss) verwendet. Die Höhe des Skelettes wird genau so berechnet, damit dieser tiefste Punkt genau auf dem Boden ist. Der tiefste Punkt wird über längere Zeit berechnet, damit es möglich ist vom Boden abzuspringen. Wird immer der aktuell tiefste Punkt zur Berechnung verwendet, klebt das Skelett zu jeder Zeit am Boden, auch wenn die Person nur kurz hoch springt.

Default Daten

In der Klasse `VRPerson` ist auch ein Default Datensatz gespeichert. Dieser kann über `getDefaultValueData` geholt werden. Zusätzlich wird dieser Datensatz auch bei `getData` zurückgegeben, falls die Kinect keine Daten liefert. In der folgenden Abbildung ist das Default Skelett visualisiert.

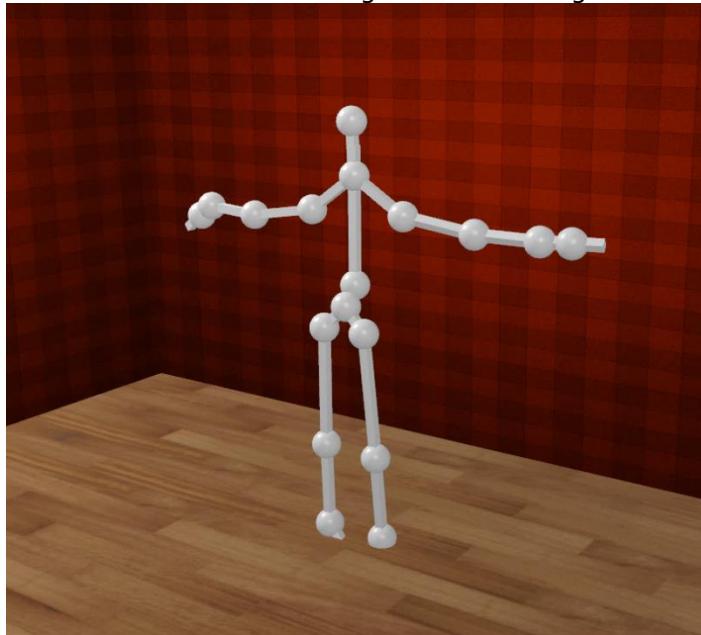


Abbildung 15: Default Skelett

Diese Daten werden auch für die über das Netzwerk verbundenen Personen verwendet. Sind diese Personen nicht verbunden, werden sie als Default Skelette dargestellt. Das Default Skelett wird in jeder Situation als Platzhalt verwendet, solange keine realen Kinect Daten vorhanden sind.

4.2.4 Daten-Darstellung

Für die Darstellung von einem Skelett ist die Klasse `SLPerson` verantwortlich. Diese erbt von der Klasse `SLGroup` und ist somit ein fester Bestandteil vom Szenengraph. Für jedes Skelett in der Szene wird ein eigenes `SLPerson` Objekt erzeugt. Beim Initialisieren (`BuildFigureGroup`) werden alle benötigten Objekte erstellt. Das Skelett wird hierarchisch zusammen gebaut. Jeder Knochen wird als `SLGroup` repräsentiert. An jeder Gruppe hängt eine `SLSphere` (Gelenk) und eine `SLBox` (Knochen). Die `SLBox` wird mit der Länge 1 initialisiert, damit sie möglichst einfach auf die erforderliche Knochenlänge skaliert werden kann. In einem ersten Schritt haben wir statt einer Box einen Zylinder verwendet, da die Knochen eher rund als eckig sind. Damit für Testzwecke alle Rotationen ersichtlich sind (beim Zylinder ist die Rotation entlang der Achse nicht ersichtlich), wird eine Box verwendet.

Beim Kopf wird eine `SLCamera` hinzugefügt. Falls eine Oculus Rift angeschlossen wird das Bild entsprechend stereoskopisch gerendert. Die Kamera einer Person kann über die Funktion `makeCamActive` aktiviert werden.

`rebuildFigure`

Diese ist die zentrale Funktion dieser Klasse. Sie muss zum Aktualisieren der Darstellung aufgerufen werden. Normalerweise passiert das vor jedem render Durchgang. Dabei werden die Matrizen der Knochengruppen aktualisiert. Zusätzlich wird das Skelett vom rigged Mesh aktualisiert. Dazu mehr im Kapitel 4.5 Rigged Mesh.

4.2.5 Zusammenfassung

Der gesamte Prozess vom Auslesen der Kinect Daten bis zum Aktualisieren des dargestellten Skelettes wird bei jedem render Durchgang durchlaufen. Dieser Ablauf wird jeweils in der `VRSceneView` in der Funktion `preDraw` durchgeführt. Dieser Vorgang wird im folgenden Diagramm noch einmal verdeutlicht.

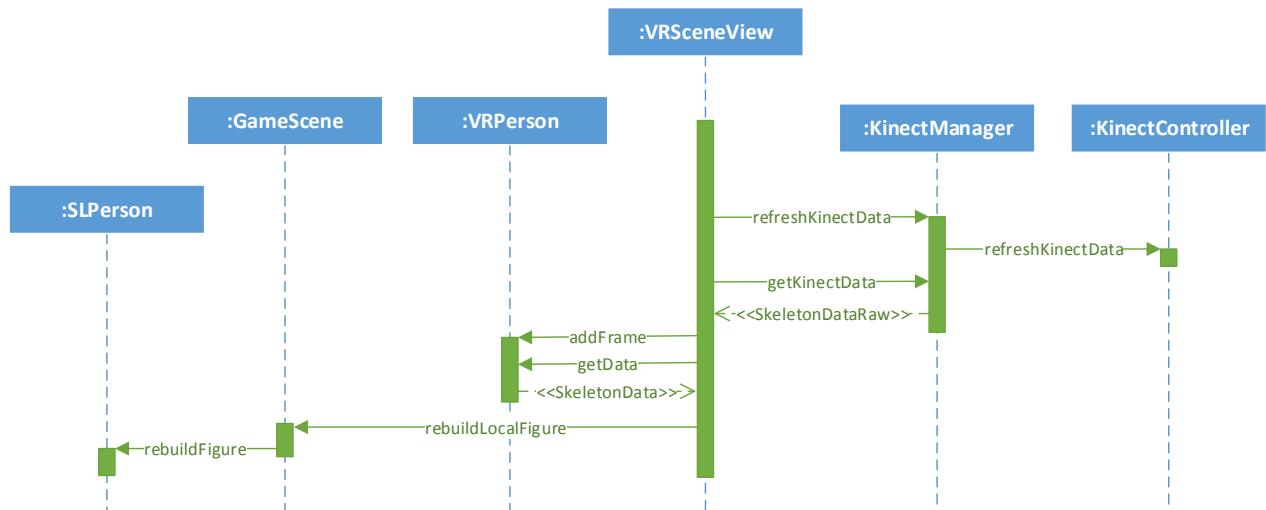


Abbildung 16: Skelett Daten aktualisieren

4.3 Client / Server Kommunikation

Wie bereits im Kapitel 2 Architektur beschrieben, wurde im Rahmen dieses Projekts auf eine Client / Server Architektur gesetzt. Ziemlich zu Beginn musste dafür ein Grundkonzept samt Demo-Implementation für die Machbarkeit auf die Beine gestellt werden. Dabei standen unterschiedliche Punkte wie das zu verwendende Protokoll, allfällige unterschiedliche Implementationsmethoden und der potenzielle Einsatz von Libraries zur Diskussion.

4.3.1 Grundfunktionalität

Nach kurzen Recherchen im Internet und Rücksprache mit dem Betreuer wurde der Grundsatz beschlossen, die Netzwerk Engine RakNet [4] für den Aufbau, die Verwaltung und die Kommunikation um, und über das Netzwerk zu verwenden. Obwohl nur ein kleiner Bruchteil der Funktionalitäten im Rahmen des Virtual Room Projekts verwendet wurde, verhalf uns die Software diverse Netzwerk-Technische Schwierigkeiten, wie unterschiedliche System-Architekturen (little-endian, / big-endian), als gelöst zu betrachten.

4.3.1.1 UDP oder TCP

Trotz der Verwendung von RakNet für die Verbindungskontrolle und das Senden, resp. Empfangen von Paketen, stellte sich eine bei Netzwerkanwendungen immer aufkommende Frage: Sollte beim verwendeten Protokoll auf UDP (User Datagram Protocol) oder auf TCP (Transmission Control Protocol) gesetzt werden. Dabei wurde nach einigen Recherchen ersichtlich, dass bei Game-Anwendungen (oder Simulationen) meist UDP zum Einsatz kommt. TCP wird öfters beim Versenden von Dateien oder komplett zusammenhängenden Daten verwendet.

Der Grund dafür liegt hauptsächlich darin, dass UDP Pakete direkt über das Netzwerk ohne grosse Überprüfungen versendet werden. Dadurch können die Daten im Gegensatz zu TCP ohne grosse Verzögerung an den Client gesendet werden. Dies ist bei Games und Simulationen für einen möglichst kleinen Lag von sehr hoher Wichtigkeit. Dies bringt jedoch auch einige Nachteile mit sich. Zum einen gibt es keine Garantie über die sichere Ankunft der Daten, noch müssen sich die erhaltenen Daten in der korrekten Reihenfolge befinden. RakNet bietet jedoch einige Lösungsansätze, um diesen Nachteile entgegen zu wirken.

4.3.1.2 Lib-RakNet

Wie bereits in der Projektstruktur beschrieben, musste für die Verwendung von RakNet ein eigens dafür vorgesehenes Projekt angelegt werden. Anhand von diesem konnte die Philosophie vom SLProject, immer wenn möglich den Sourcecode einer Abhängigkeit selber zu kompilieren, weiterverfolgt werden.

4.3.2 Common Library

Das Projekt *Lib-VirtualRoomCommon* beinhaltet diejenigen Komponenten, welche sowohl vom Server wie auch vom Client verwendet werden. Dabei handelt es sich um gemeinsam verwendete Typen, Enumerationen sowie Grundfunktionalitäten zum Versenden und Empfangen von Paketen. Die Definition dieser Pakete ist ebenfalls im gemeinsamen Projekt vorhanden.

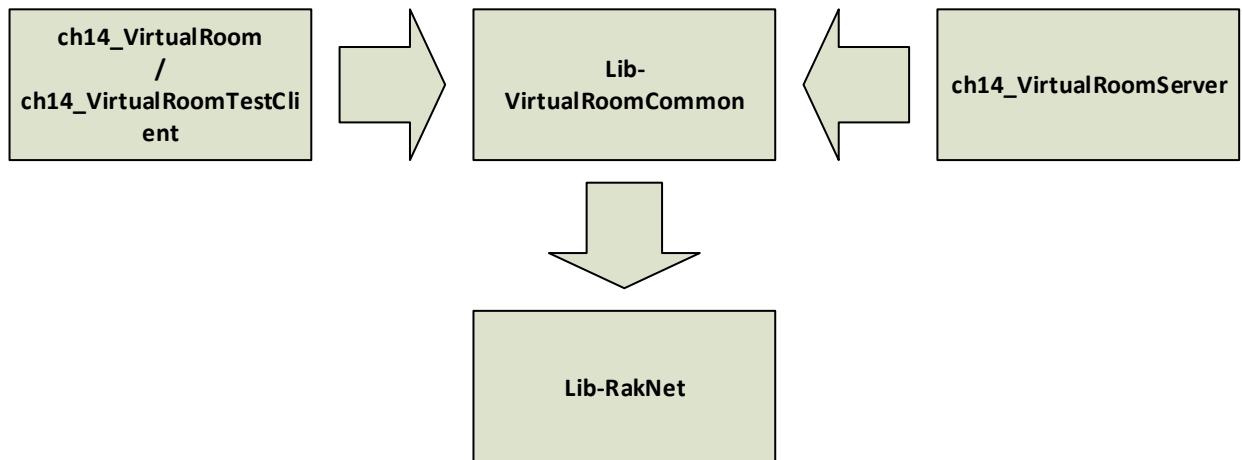


Abbildung 17: Projekt-Abhängigkeiten

Die Library kann zusätzlich als drei einzelne Teile betrachtet werden.

- Die Netzwerk-Klassen
- Projekt übergreifende Konstante
- Gemeinsam verwendete Typen

4.3.2.1 Netzwerk – Klassen

Diese Klassen entsprechen einer Kapselung für die zu versendende und zu empfangende UDP Pakete. Dabei wird jeweils der Inhalt eines solchen Pakets über die `serializeMessage` Funktion in ein Bit-Stream der Bibliothek RakNet geschrieben. Dieser Stream kann dann über RakNet Funktionen direkt einer System Adresse über das definierte UDP-Protokoll versendet werden. Dabei können noch weitere Parameter wie das interne Erzwingen einer Ankunftskontrolle angegeben werden.

Wie in der nächsten Abbildung ersichtlich, stammen alle Daten-Paket Klassen von der gemeinsamen Basis Klasse `BaseMessage` ab. Dabei besitzt jedes Paket neben einer Message-ID noch einen Erzeugungs-Zeitstempel. Diese beiden Werte werden immer als Erstes in den `RakNet::BitStream` geschrieben und auch wieder gelesen. Anhand der Message-ID kann die Aufgabe des Auslesens aus dem `BitStream` der jeweiligen Klassen übergeben werden. Die entsprechende Methode, und weitere Hilfsmethoden, sind in der Hilfsklasse `NetworkUtils` im Namespace `VirtualRoom::Network` enthalten.

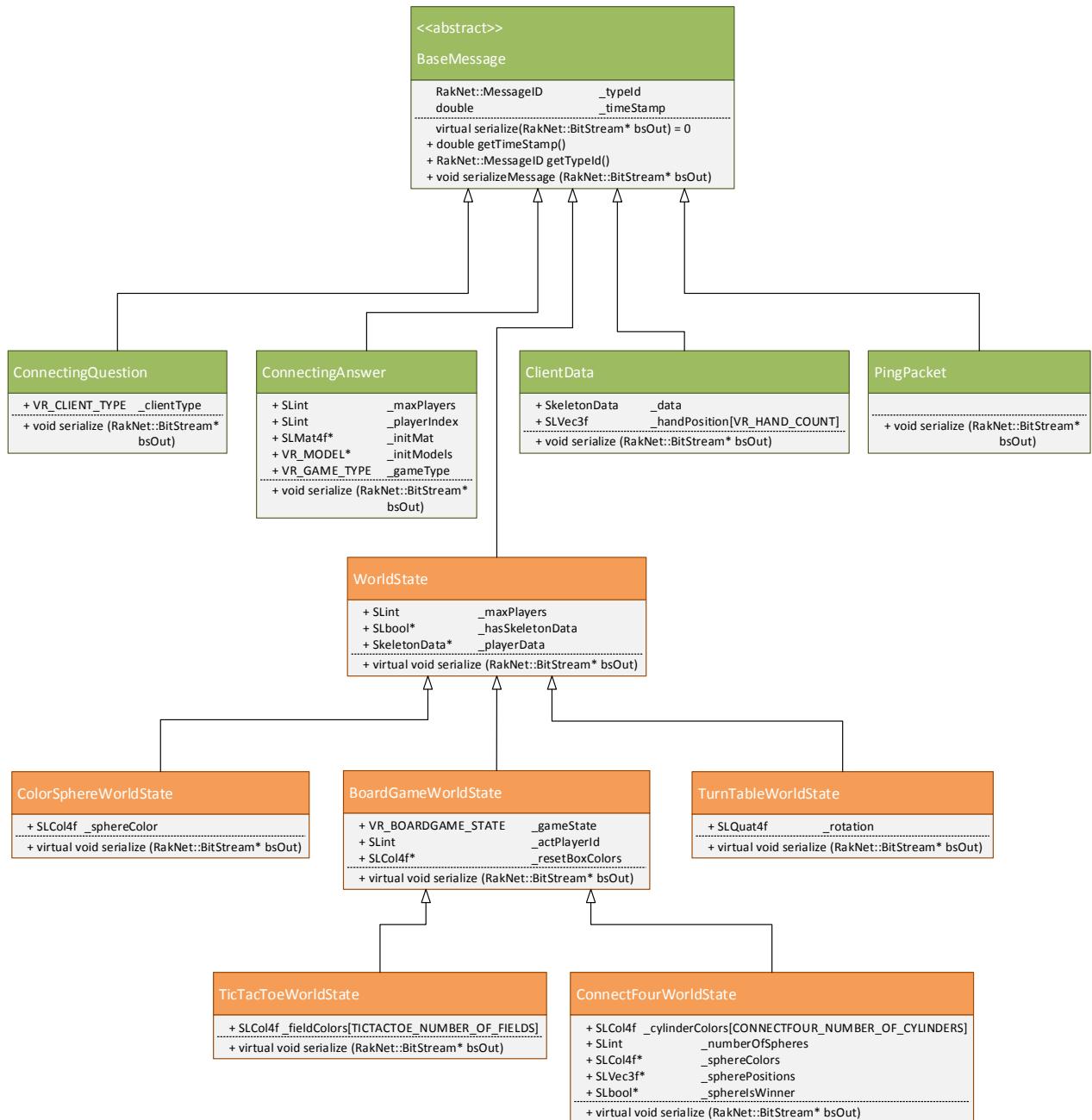


Abbildung 18: Common - Netzwerk Klassen

Die grünen Klassen befinden sich im `VirtualRoom::Network` Namespace. Die orangen ihrerseits sind direkt im `VirtualRoom` Namespace enthalten. Die einzelnen Attribute, vor allem der Subklassen vom Welt-Zustand (`WorldState`), sind in den Game Implementation beschrieben.

4.3.2.2 Konstanten

Die Konstanten in der Common Library entsprechen fixen Werten, welche sowohl vom Client wie auch vom Server verwendet werden. Dabei handelt es sich um feste Größen bei den Brettspielen (Board Games), welche vom Client für die Anzeige und vom Server für die Berechnungen verwendet werden.

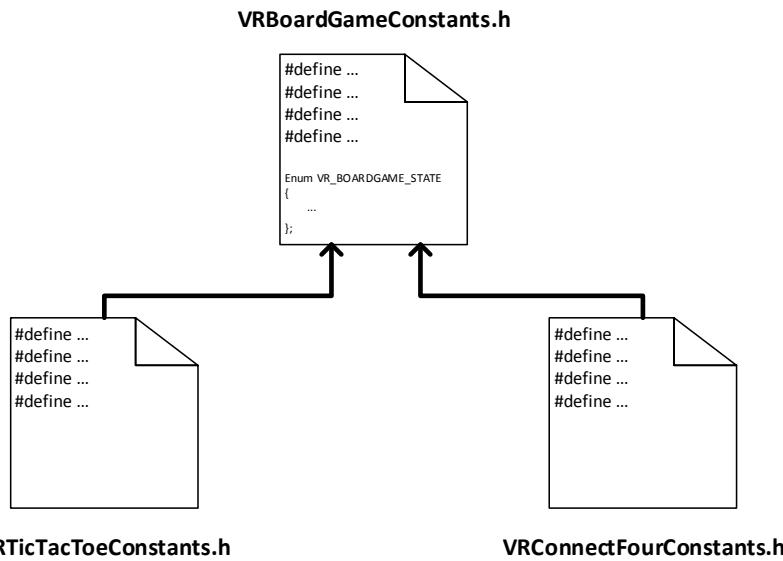


Abbildung 19: Common - BoardGame Konstanten

4.3.2.3 Typen

Die Typen in Form von Enums und Structs sind allesamt in der Datei [VRTypes.h](#) definiert. Die Enums werden für die gezielte Adressierung der einzelnen Gelenke und Knochen verwendet. Zusätzlich sind Client- und Game-Typen definiert.

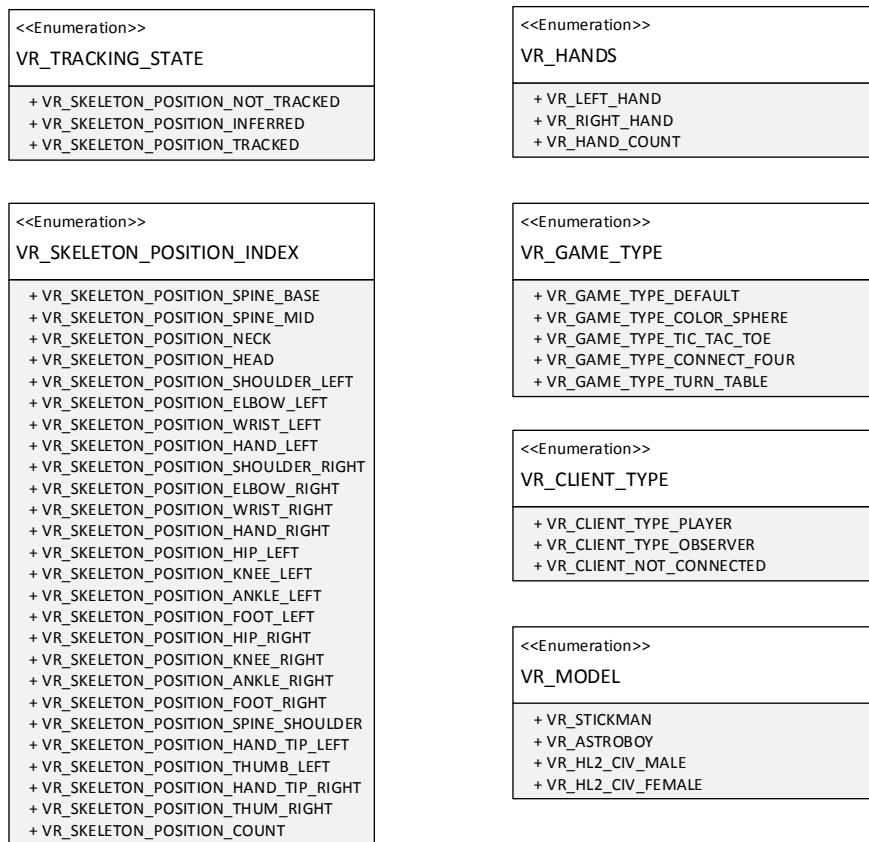


Abbildung 20: Common - Enumerations

Neben diesen Aufzählungen sind in der [VRTypes](#) Headerdatei ebenfalls die verwendeten Typen für die Datenkapselung definiert. Diese werden für den Transfer der Skelett-Daten verwendet. Dabei gibt es zwei Fassungen. Einerseits eine Raw-Datenstruktur, welche die effektiven von der Kinect gelieferten Werte in eine Datenstruktur kapselt. Die zweite Version ist etwas schlanker, ohne den Tracking Status pro Knochen.

<<Struct>> SkeletonBone <hr/> + VR_SKELETON_POSITION_INDEX parent + SLQuat4f rotation + SLQuat4f absoluteRotation + SLfloat length	<<Struct>> SkeletonData <hr/> + SLVec3f position + SLQuat4f headRotation + SkeletonBone bones[VR_SKELETON_POSITION_COUNT]
<<Struct>> SkeletonBoneRaw <hr/> + VR_SKELETON_POSITION_INDEX parent + SLQuat4f rotation + SLQuat4f absoluteRotation + SLfloat length + VR_TRACKING_STATE state	<<Struct>> SkeletonDataRaw <hr/> + SLVec3f position + SLQuat4f headRotation + SkeletonBoneRaw bones[VR_SKELETON_POSITION_COUNT]

Abbildung 21: Common – Structs

4.3.3 Serverseitige Implementation

Beim Server handelt es sich um eine reine Konsolen-Applikation. Diese läuft in einem Endlos-Modus, wobei für saubere Simulationen auf eine konstante Tickrate geachtet werden musste. Die Anwendung kreiert beim Start die gewünschte Server-Instanz.

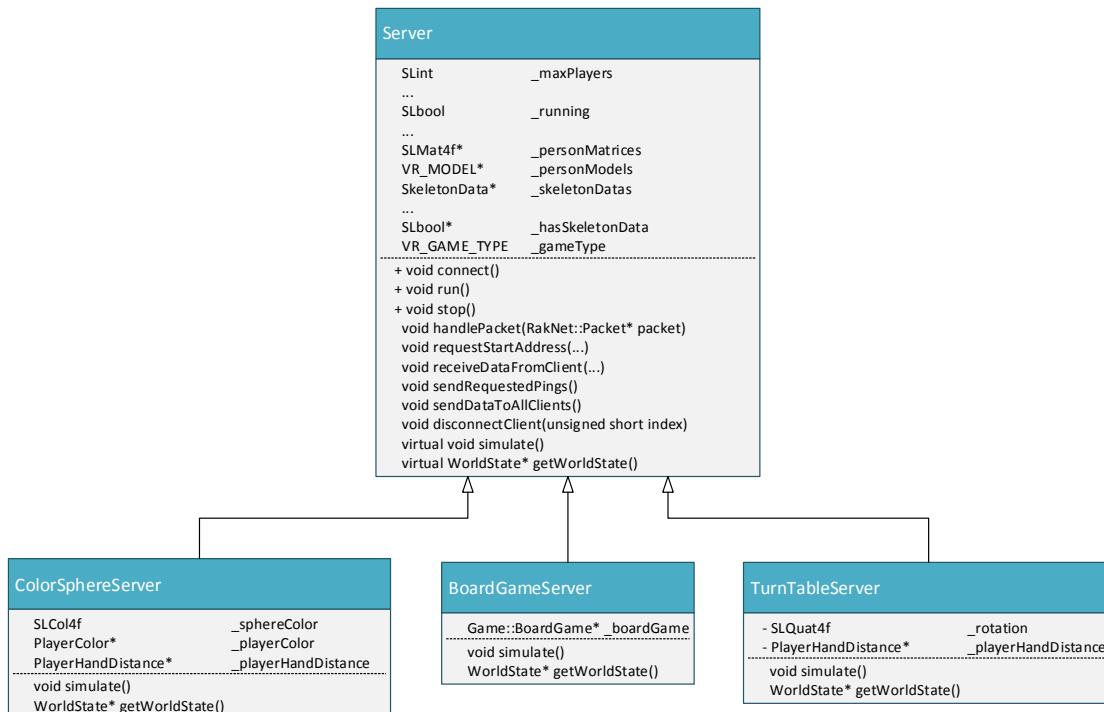


Abbildung 22: Server - Klassen

Detaillierte Beschreibungen aller im Rahmen dieser Arbeit implementierten Server Instanzen sind im nachfolgenden Kapitel 4.4 Game Implementationen aufgeführt.

4.3.3.1 Server-Start

Der Start der Applikationen, also die `main` Methode, befindet sich in der Datei `VRServerStarter.cpp`. In dieser Methode wird anhand des ersten Command Line Arguments der entsprechende `VR_GAME_TYPE` ermittelt. Basierend auf dieser Information wird eine Instanz einer Subklasse von `Server` erstellt. Die beiden darauffolgend verwendeten Funktionen `connect` und `run` sind nur in der Basis-Klasse implementiert. Nebst dem Erzeugen der Server-Instanz wird in der Startdatei ein Ctrl-Handler implementiert, welcher es ermöglicht den Server über die Tastenkombination Ctrl-C sauber herunterzufahren. Bei der Instanziierung des Servers muss angegeben werden, wie viele Player und Observer sich jeweils maximal gleichzeitig verbinden dürfen. Basierend auf der Player-Anzahl werden die jeweiligen Start-Positionen und Orientierungen (ein Kreis bildend mit einem fixen Abstand zum Ursprung und darauf blickend) errechnet. Zusätzlich wird jeder Position ein fixer Model-Typ (`VR_MODEL`) zugewiesen. Aktuell ist die Verteilung der Modelle so, dass jeder gerade Startplatz einem männlichen und jeder ungerade Startplatz einem weiblichen Half-Life 2 Model entsprechen soll.

Connect:

Dieser Aufruf startet den Server, sodass über die RakNet Funktionalitäten der Socket (IP und Port Kombination) reserviert wird.

Run:

Bei diesem Aufruf startet der Server seine eigentliche Funktion. Dabei wird der Server immer, solange er nicht gestoppt wurde (Ctrl-Handler), die nachfolgenden Schritte ausführen:

- Startzeit ermitteln
- Alle Pakete über die RakNet Library abrufen und jedes direkt abarbeiten
- Die in der instanzierten Sub-Klasse implementierte `simulate` Funktion ausführen
- Allfällige Ping-Requests beantworten
- Für konstante Tickrate: $\text{Tick-Rate} - (\text{Endzeit} - \text{Startzeit})$ warten
- World-State an alle Clients versenden

Das Abarbeiten der empfangenen Pakete wird anhand der Message-ID im Paket unterschiedliche Funktionen ausführen.

Wird der Server gestoppt, fällt die Funktion aus dem oben beschriebenen Loop und fährt alle offenen Verbindungen über eine `CloseConnection` Funktion von RakNet herunter.

4.3.3.2 Client-Connect / -Disconnect

Obwohl UDP als Protokoll Connection-Less ist (keine aktive Verbindung nötig), bietet RakNet die Möglichkeit Verbindungen aufzubauen und diese gezielt oder erzwungen über Timeouts auch wieder zu beenden. Dafür ist das Verhalten identisch, sprich wenn eine Verbindung wegen einem Timeout beendet wird, simuliert RakNet den Empfang eines Disconnect Pakets.

Beim Verbindungsaufbau sendet der Client ein `ConnectingQuestion` Paket an den Server. In diesem Paket ist der Typ des Clients als `VR_CLIENT_TYPE` enthalten. Bei der Initialisierung des Servers wurde spezifiziert, wie viele Player und Observer maximal gleichzeitig verbunden sein können. In der `requestStartAddress` Methode wird auf dem Server die Anfrage des Clients bearbeitet. Ist die maximale Anzahl Player oder Observer bereits erreicht, wird die Verbindung zum Client über die RakNet Funktionalität automatisch wieder geschlossen. Meldet sich ein neuer Player, wird diesem der nächste freie Startplatz zugewiesen. Dieser zugewiesene Platz wird nebst allen Startpositionen, - Rotationen und dem pro Platz zugewiesenen Model in einem `ConnectingAnswer` Paket dem Client zugesandt. Zusätzlich wird dem Client für das Erstellen der korrekten Szene der Game-Type im Paket

angegeben. Handelt es sich beim verbindungs-aufbauenden Client um einem Observer, werden dieselben Informationen wie beim Player gesendet. Einzig eine Startposition muss dem Observer nicht übermittelt werden, und wird daher auf einem Default-Wert „-1“ gesetzt. Nachdem das Antwortpaket versendet wurde, muss auf dem Server vermerkt werden, dass der neue Client von jetzt an nach jedem Loop den Welt-Zustand zugesandt werden soll.

Wie bereist beschrieben, handelt es sich beim Schliessen einer Verbindung immer um den Erhalt eines Disconnet-Pakets. Dabei muss neben der Freigabe des Startplatzes bei einem Player, einzig noch vermerkt werden, dass dem Client keine Welt-Zustände mehr zugesendet werden müssen. Dabei bekommt automatisch auch wieder ein weiterer Observer die Möglichkeit sich zu verbinden.

4.3.3.3 Daten Empfang / Ping-Requests

Der Daten-Empfang eines Players ist sehr simple. Pro Player-Platz im virtuellen Raum wird vermerkt ob Skelett-Daten vorhanden sind oder nicht. Falls nicht, werden diese beim Empfangen gesetzt, ansonsten überschrieben, sodass immer nur die aktuellsten Daten pro Player-Platz auf dem Server gespeichert werden. Nebst den Skelett Daten werden die effektiven Handpositionen pro Player für Interaktionen innerhalb der Game Implementationen gespeichert.

Zusätzlich zu Daten-Pakete kann dem Server auch ein Ping-Paket gesendet werden. Dieses Paket beinhaltet keinen Inhalt. Beim Abarbeiten wird lediglich vermerkt, welcher Client eine Ping-Anforderung gestellt hat. Diesen Clients wird vor dem Warten für die konstante Tickrate ein gleichermaßen leeres Ping-Paket zurückgesendet. Diese Pakete wurden für die Performance- und Latenz-Tests verwendet.

4.3.3.4 Simulation / World-State Versand

Für die Simulation von Veränderungen an der Welt wird die virtuelle `simulate` Funktion auf dem Server ausgeführt. Dadurch kann jede Game Implementation seine eigene Änderungs-Funktion implementieren. Die Funktion der Basis-Klasse muss keine Änderung durchführen, da sich neben den Player nichts Weiteres im Raum befindet, was verändert werden könnte. Die Veränderung der Player (Position des gesamten Skelettes, sowie der einzelnen Gelenke) wird durch das Abarbeiten der Client-Daten bereits durchgeführt.

Dadurch, dass der Welt-Zustand nur durch jede Game Implementation selbst bekannt ist und manipuliert wird, muss die Ermittlung des World-States für das Versenden an alle Player und Observer ebenfalls an die jeweiligen Server Sub-Klassen delegiert werden. Dabei wird die virtuelle Funktion `getWorldState` aufgerufen. Diese Funktion wird einen Zeiger auf ein `WorldState` Klasse zurückgeben. Dieses Paket wird dann noch zusätzlich mit allen aktuellen Skelett-Informationen ausgestattet (pro Player). Danach wird das Paket allen verbundenen Clients über die im Abschnitt 4.3.2.1 Netzwerk - Klassen beschrieben `BitStream` Klasse und RakNet Funktionalitäten versendet.

4.3.4 Netzwerk Test-Client

Um die grundsätzliche Netzwerk-Kommunikation verifizieren zu können, wurde in einem ersten Schritt ein einfacher Netzwerk-Test-Client erstellt. Beim Projekt `ch14_VirtualRoomTestClient` handelt es sich wie beim Server um eine reine Konsolen-Applikation. Diese Anwendung verwendet die im finalen Client implementierte Klasse `NetworkClient`.

```

NetworkClient

- RakNet::RakPeerInterface* _peer
- RakNet::SystemAddress _systemaddress
- bool _connected
- string _serverAddress
- VR_CLIENT_TYPE _clientType
- SLTimer _timer
- double _pingStartTime
- double _lastPingTime
- double _lastDataPacketReceived
- SLint _numberOfReceivedDataPackets

+ void connect()
+ void waitForConnect()
+ void sendConnectingQuestion()
+ void waitForConnectingAnswer(Network::ConnectingAnswer** connectingAnswer)
+ bool receiveServerData(RakNet::BitStream** bsIn)
+ void sendData(SkeletonData& data, SLVec3f& leftHand, SLVec3f& rightHand)
+ bool isConnected()
+ void sendPingPacket()
+ double lastDataPacketReceived()
+ SLint numberOfReceivedDataPackets()
+ double lastPingtime()
+ SLTimer* getTimer()

```

Abbildung 23: NetworkClient Klasse

In der oben beschriebenen Klasse befinden sich alle benötigten Funktionen für den Aufbau der RakNet Verbindung (Methode `connect` und `waitForConnect`), die konkrete Anfrage mit dem eigenen Client-Typ (`sendConnectingQuestion` und `waitForConnectingAnswer`), das Senden der Client-Daten an den Server (`sendClientData`), das Empfangen des Welt-Zustands (`receiveServerData`) und weitere Hilfsfunktionen für die Messungen von Latenzzeiten (Pings und Zeitpunkt / Anzahl erhaltene Daten-Pakete).

Da der Test-Client keine effektiven Daten von der Kinect und, oder Rift erhält, werden Default Skelett-Daten an den Server geschickt. Um eine gewisse Auslastung der Konsolen-Applikation zu simulieren, wird am Ende jedes Loops ein `Sleep` aufgerufen. Wird die Verbindung geschlossen (Timeout oder bedingt durch den Server), wird die Konsolen-Applikation automatisch beendet.

4.3.5 Virtual Room Client

Wie bereits unterer 4.1 (Anpassungen SLProject) beschrieben, setzt das Projekt `ch14_VirtualRoom` auf das SLProject als Framework. Konkret bedeutet dies, dass die beiden Funktionen `SLScene::onLoad` und `getSceneView` implementiert werden müssen.

`SLScene::onLoad`:

Die Funktion ist in der Datei `SLScene_onLoad.cpp` implementiert. Sie erstellt den eigentlichen Raum, welcher unabhängig des Game-Implementation Inhaltes oder der Personen immer dasselbe Erscheinungsbild garantieren soll. Dafür wurden vier Wände und eine Decke mit ein und derselben Textur, einen Boden mit einer anderen Textur sowie zwei entfernte Lichtquellen in die Scene gesetzt.

Abbildung 24: `SLScene::onLoad` - Leerer Raum

getSceneView:

Die zweite zu implementierende Funktion soll einen Pointer auf eine `SLSceneView` Instanz zurückgeben. Durch die konkrete Verwendung der drei Funktionen `onStartup`, `postSceneLoad` und `preDraw` wurde eine eigene Klasse `VRSceneView` erstellt. Ein Pointer auf eine Instanz von dieser Klasse wird zurückgegeben.

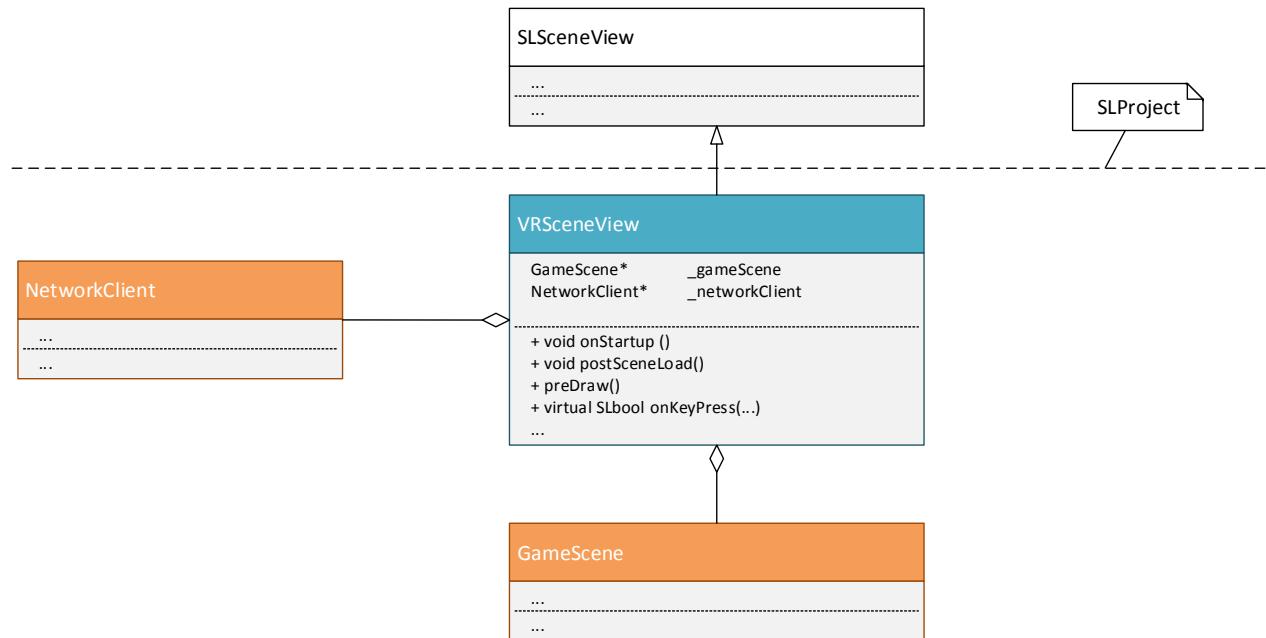


Abbildung 25: VRSceneView Klasse

Wie aus der obigen Abbildung ersichtlich, besitzt die `VRSceneView` eine Referenz auf eine `GameScene`. Subklassen von dieser werden dann bei den konkreten Game Implementation dafür zuständig sein, die initialen Objekte im Raum zu erstellen, und basierend auf den erhaltenen World-State Pakete zu aktualisieren. Die Erzeugung der Personen im Raum und deren Aktualisierungs-Vorgang ist jedoch über alle `GameScenes` gleichermassen zu implementieren, und wurde daher in die Basis-Klasse verschoben.

4.3.5.1 VRSceneView

Die Klasse stellt gewissermassen die Implementation vom SLProject als Framework dar. Dabei kommen unterschiedliche Funktionen zum Einsatz. Nachfolgend sind die wichtigsten samt Beschreibung aufgelistet.

onStartup:

Die Funktion wird einmalig beim Starten der Applikation aufgerufen. Dabei werden die beim Ausführen des Programms übergeben Kommando-Zeile Argumente in einem String-Vector als Argumente der Funktion übergeben. Diese Argumente werden als Erstes über die eigene Funktion `initCommandLineArguments` ausgewertet.

NR	Verwendung	Typ	Default – Wert
1	Server Name oder IP-Adresse	String	localhost
2	Client – Typ als Zahl	VR_CLIENT_TYPE	VR_CLIENT_TYPE_PLAYER

Tabelle 1: Virtual Room Client - Kommando-Zeile Argumente

Anschliessend wird, falls es sich beim Client um einen Player handelt, der `KinectManager`, wie im Abschnitt 4.2.1 beschrieben, initialisiert. Danach wird versucht anhand der `NetworkClient` Methoden

eine Verbindung zum ermittelten Servernamen oder IP-Adresse mit der Angabe des Client-Typen aufzubauen. Gelingt dies, werden die erhaltenen initial Matrices und Model Angaben aller Personen in Klassen-Variablen zwischengespeichert. Zusätzlich muss der eigene Index zwischengemerkert werden. Konnte keine Verbindung aufgebaut werden und der Client wurde als Player gestartet, wird einfach eine einzelne Person mit dem vermerkten Index darauf in den Raum gesetzt. Somit ist eine allfällige Bewegung im virtuellen Raum trotzdem möglich. Bei fehlender Verbindung kann eine Game Implementation jedoch auf keinen Fall geladen werden, da die Berechnungen für Veränderungen an der Welt anders als diejenigen am eigenen Körper immer vom Server durchgeführt werden. Gründe für ein Fehlschlagen beim Verbindungsaufbau können folgende sein:

- Der Server läuft nicht
- Der Servername oder IP-Adresse wurde falsch angegeben
- Der verwendete UDP-Port **60000** (momentan überall fix als Konstante programmiert) wurde blockiert
- Der Server hat bereits zu viele verbundene Clients vom gewünschten Typ

postSceneLoad:

Die Funktion wird aufgerufen nachdem die `SLScene::onLoad` Methode und weitere SLProject spezifische Initialisierungs-Tasks durchgelaufen sind. In dieser Methode wird der Rest der Szene hinzugefügt. Zum einen sind dies zwei freie Kameras, welche der Observer ansteuern kann. Zum anderen lädt jede implementationsspezifische `GameScene` Instanz eigenen Inhalt in die Szene. Welche Instanz von `GameScene` erstellt wird, entspricht dem im `ConnectingAnswer` erhaltene `VR_GAME_TYPE`. Ist keine Verbindung vorhanden und folglich auch nie eine `ConnectingAnswer` erhalten worden, wird direkt eine Instanz von `GameScene` erstellt. Dies ist notwendig, da in dieser Basis-Klasse die Person erstellt und verarbeitet wird.

preDraw:

Vor dem eigentlichen Rendern der Szene müssen zwei Hauptätigkeiten durchgeführt werden:

1. Aktualisieren der Daten der Kinect und der Rift
2. Datenverarbeitung im Sinne vom Empfangen der Netzwerk-Daten, die Aufbereitung der Skelett-Daten und das Senden der aufbereiteten Daten an den Server

Diese Aktionen müssen jeweils nur in Abhängigkeit des Verbindungsstatus und je nach Client-Typ ausgeführt werden. Das nachfolgende Diagramm sollte dieses Verhalten etwas näher erläutern.

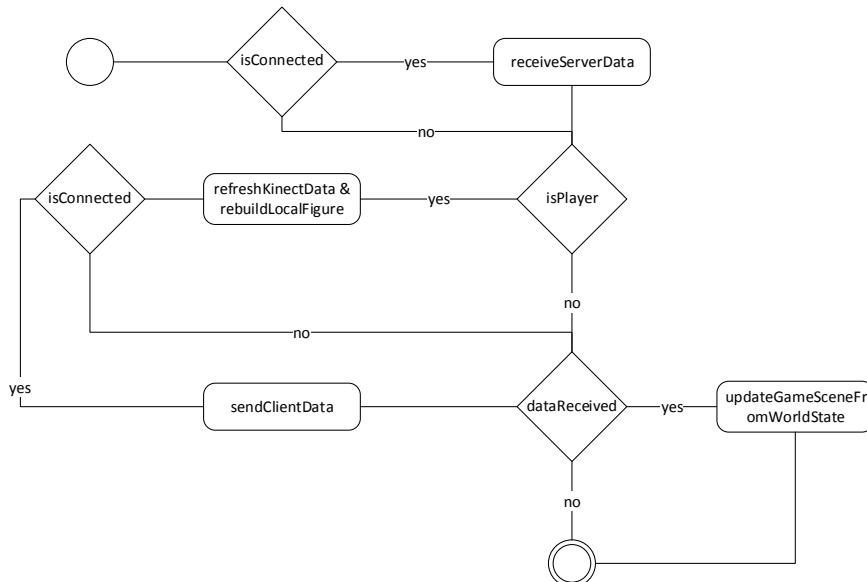


Abbildung 26: Ablauf `VRSceneView::preDraw`

Alle Schritte bis auf die Aktualisierung der Szene wurden bereits in den vorherigen Kapiteln etwas näher beschrieben. Basierend auf den empfangenen Daten wird auf der in der `postSceneLoad` Funktion erstellten `GameScene` Instanz die virtuelle Funktion `updateFromWorldState` aufgerufen. Die Funktion benötigt zum einen den `RakNet::BitStream` mit dem erhaltenen World-State. Zusätzlich werden Default-Daten aus der `VRPerson` mitgegeben. Beim Erhalten der Server-Daten ist zu erwähnen, dass die dazugehörige `NetworkClient` Funktion alle neuen Pakete durchgeht und dabei einfach nur den Inhalt des letzten Daten-Pakets berücksichtigt. Betrachtet man eine andere Client-Server Architektur, zum Beispiel einer Game-Engine, wird beim Erhalten der Daten immer eine Interpolation der Inhalte basierend auf Zeitstempeln durchgeführt. In diesem Projekt wurde mit Absicht darauf verzichtet. Hauptgrund dafür sind die sehr kleinen Datenmengen, welche hin und her gesendet werden. Zusätzlich ist der Rechenaufwand auf dem Server, wegen den klein gehaltenen Szenen, sehr klein, sodass der Server mit einer hohen Tickrate problemlos senden kann. Sollten jedoch Performance Einbussen bemerkt werden, wäre eine Interpolation sicherlich ein wichtiges und momentan fehlendes Feature.

Im obigen Diagramm nicht aufgezeigt, ist die Kamera Loop Funktionalität, welche bei einem Observer aktiviert werden kann. Eine genauere Beschreibung folgt unter 4.3.5.3 Keyboard Events.

4.3.5.2 GameScene

Die `GameScene` Klasse wird verwendet, um den Inhalt unseres virtuellen Raums zu initialisieren und manipulieren. Dabei wird je nach Game Implementation in der `postSceneLoad` Funktion von `VRSceneView` eine Instanz der Basis- oder Sub-Klasse erstellt.

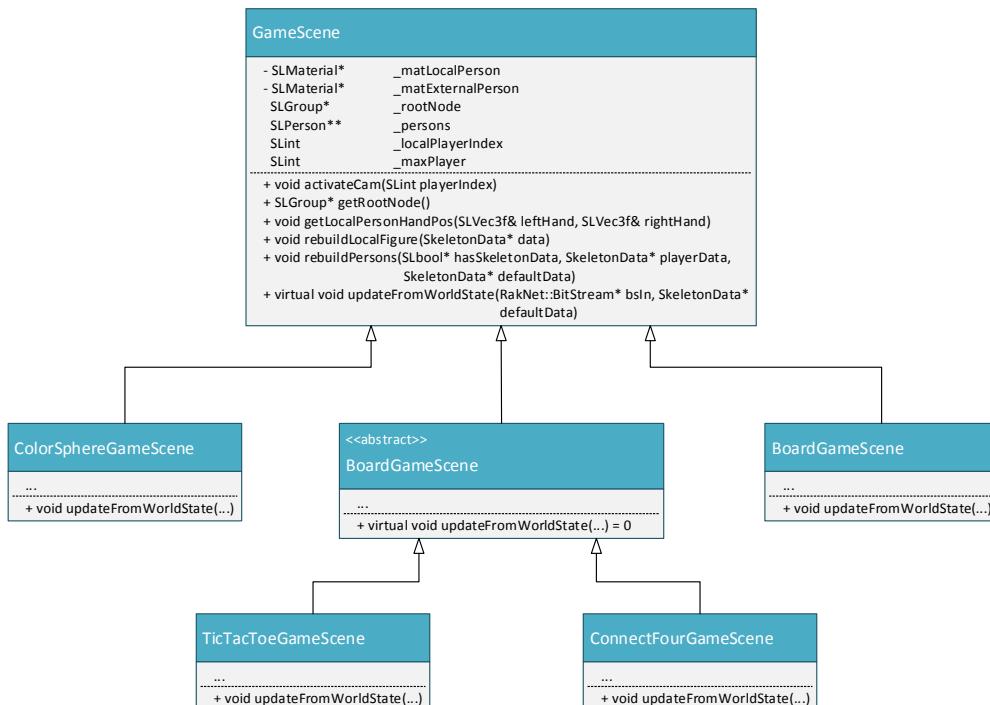


Abbildung 27: Klassendiagramm, GameScene

Der Konstruktor der jeweiligen Klasse ist dafür zuständig, dass der gewünschte Inhalt in die `_rootNode` (eine `SLGroup`) kommt. In dieser Gruppe befindet sich der gesamte Inhalt des virtuellen Raums. Da alle `GameScenes` zwingend die Personen darstellen müssen, werden diese im Konstruktor der Basis-Klasse als `SLPerson` Instanzen initialisiert. Die Anzahl, deren `VR_MODEL` Typs und Position wurden vorgängig aus der Antwort beim Erstellen der Verbindung zum Server ermittelt (oder Standard Daten falls keine Verbindung). Da jede `SLPerson` eine Kamera enthält, wird beim Erstellen der `GameScene` die entsprechende Personen-Kamera direkt aktiviert. Die eigene Person wird aus einer

Index-Variablen ermittelt. Diese Variable wird nur bei einem Client vom Typ Player gesetzt, da der Observer an keine fixe Person gebunden ist.

rebuildLocalFigure:

Diese Funktion ruft direkt die `rebuildFigure` Funktion auf der eigenen `SLPerson` Instanz auf. Welches die eigene ist, wird aus den oben beschriebenen Variablen ermittelt.

rebuildPersons:

Diese Methode ist dafür zuständig alle Personen, außer die eigene, basierend auf den erhaltenen Paket-Daten zu aktualisieren. Dabei wird pro `SLPerson` im Raum überprüft ob für diese Daten vorhanden sind. Falls ja, dienen diese als Basis für die `rebuildFigure` Funktion in der `SLPerson`. Ansonsten werden die angegebenen Default-Daten verwendet.

updateFromWorldState:

Der virtuellen Funktion wird der aus dem letzten Daten-Paket ausgelesene `BitStream` übergeben. Jede `GameScene` Implementation weiß, welchen Sub-Typ von `WorldState` der Server als Welt-Zustand gesendet hat. Dadurch kann die `update` Funktion anhand des `BitStreams` den Konstruktor der entsprechenden Klasse aufrufen, wobei dieser die Daten wieder korrekt einliest. Da die Basis-Klasse von `GameScene` nur die Personen in den virtuellen Raum stellt, werden diese Personen spezifische Daten in der `WorldState` Instanz übermittelt. Die Personen werden dann über die `rebuildPersons` Funktion aktualisiert.

4.3.5.3 Keyboard Events

Auf dem Client wurden zusätzlich einige eigene Tastatur Events abgefangen. Dafür wurde die `onKeyPress` Funktion der `SceneView` überschrieben. Wichtig: Damit die nicht eigens abgefangen Events weiter abarbeitet werden, muss am Schluss der Funktion die eigentliche Basis-Funktion aufgerufen werden.

Knochenlänge verändern:

Über die Tasten Komma (ASCII: 44) und Punkt (ASCII: 46) können die ursprünglich berechneten Knochenlängen (nur bei der Skelett Darstellung als `VR_STICKMAN`, nicht als Mesh) jeweils um 10% verlängert resp. verkürzt werden. Diese Veränderung wird über die Klasse `VRPerson` durchgeführt und hat folglich nur Auswirkung auf die selbst ausgelesenen Kinect Daten.

Kamera wechseln:

Handelt es sich beim Client um einen Observer, bekommt der Nutzer der Anwendung die Möglichkeit die Szene aus verschiedenen Ansichten zu betrachten. Da der Observer im Gegensatz zum Player keiner Person zugewiesen ist, wird beim Erstellen der `GameScene` auch keine Kamera direkt aktiviert. Innerhalb der `postSceneLoad` Funktion wurden jedoch zwei freie Kameras der Szene hinzugefügt. Dabei ist die Erste der beiden zu Beginn für einen Observer aktiviert. Diese kann auch frei mit der Maus und den Tasten WASD bewegt werden. Die zweite Kamera kreist anhand einer Animation oberhalb der Szene, immer ins Zentrum zeigend. Der Observer kann nun über die Nummern auf der Tastatur die verschiedenen Personen-Sichten und die zwei freien Kameras ansteuern. Dabei folgt das Abfangen der Tastatur Events bei n Personen im Virtual Room folgendem Muster.

Taste	Aktion
1	1-te <i>SLPerson</i> Kamera aktivieren
2	2-te <i>SLPerson</i> Kamera aktivieren
...	...
n	n-te <i>SLPerson</i> Kamera aktivieren
n+1	Erste frei bewegbare Kamera aktivieren
n+2	Zweite freie mit einer Animation ausgestattete Kamera aktivieren
n+3	Automatischer Loop über alle <i>SLPerson</i> Kamera und den zwei freien Kameras aktivieren

Tabelle 2: Tastatur-Events - Kamera Wechsel für den Observer

Bei aktivem Loop wird immer am Ende der *preSceneLoad* Funktion überprüft, ob auf die nächste Kamera geschaltet werden soll. Der Loop beginnt immer bei der ersten *SLPerson* Kamera. Ist der Zyklus jedoch bereits gestartet, hat die Taste n+3 keine Wirkung.

4.4 Game Implementationen

Dieser Teil der Dokumentation beschreibt wie vorgegangen werden soll, um ein eigenes „Game“ zu implementieren. Im Rahmen der Arbeit wurden alle Implementationen als Game bezeichnet obwohl nur die beiden Board-Games als Spiel betrachtet werden, im Sinne davon, dass zwei Spieler gegeneinander antreten. Der Begriff „Game“ bezieht sich darauf, dass die Implementationen nicht einer reinen Simulation entsprechen, sondern die Benutzer anhand von Interaktionen aktiv am Geschehen teilhaben können.

Folgende Schritte müssen für eine eigene Implementation durchgeführt werden:

1. Erweitern des *VR_GAME_TYPE* Enums um einen weiteren Typen (Lib-VirtualRoomCommon)
2. Erstellen einer Sub-Klasse von *WorldState* (Lib-VirtualRoomCommon)
3. Kreieren der Sub-Klasse von *Server* (ch14_VirtualRoomServer). **Wichtig:** Dem Parent-Konstruktor muss den beim Punkt 1 erstellten Typen angegeben werden. Zusätzlich müssen die beiden Funktion *simulate* und *getWorldState* zwingend überschrieben werden
4. Erweitern des Switch-Blockes in der *main* Methode der *VRServerStarter* Datei (ch14_VirtualRoomServer). Dabei wird anhand des spezifizierten Typen eine Instanz der neu erstellen Server-Klasse erstellt.
5. Kreieren einer Sub-Klasse von *GameScene* (ch4_VirtualRoom). **Wichtig:** Die Funktion *updateFromWorldState* muss zwingend überschrieben werden.
6. Erweitern des Switch-Blockes in der Funktion *postSceneLoad* der Klasse *VRSceneView* (ch14_VirtualRoom). Dabei wird ebenfalls anhand des im Punkt 1 definierten Typen eine Instanz der neu erstellen Sub-Klasse im Punkt 5 erstellt.

Diese sechs Schritte stellen das Konstrukt für eine neue Game Implementation dar, und wurden auch bei allen vier Beispiel Implementationen exakt so durchgeführt. Die Game-Spezifische Funktionalität wird nun in die folgenden Funktionen programmiert:

Klasse	Funktion	Zweck
Subklasse von Server (Punkt 3)	Konstruktor	Erzeugen der verwendeten mathematischen und sonstigen Daten-Strukturen als Repräsentation der Welt
Subklasse von Server (Punkt 3)	simulate	Veränderungen an den im Konstruktor erzeugten Daten-Strukturen z.B. durch Hit-Detections basierend auf den Positionen der Hände einzelner Personen (in der Basis-Klasse gespeichert).
Subklasse von Server (Punkt 3)	getWorldState	Erzeugen einer Instanz der in Punkt 2 definierten Sub-Klasse von <i>WorldState</i> . Diese Klasse mit allen relevanten

		Welt-Status Informationen vorgängig ergänzen, und in der Methode bestücken. Instanz zurückgegeben.
Subklasse von GameScene (Punkt 5)	Konstruktor	Erzeugen der für die Szene weiteren notwendigen Szenengraph Objekte. Diese müssen, falls durch die Server-Anwendung manipulierbar, auch zu einem späteren Zeitpunkt abrufbar sein (Pointer speichern). Zusätzlich müssen die Objekte in den <code>_rootNode</code> gehängt werden
Subklasse von GameScene (Punkt 5)	updateFromWorldState	Als Argument wird der <code>RakNet::BitStream</code> übergeben. Anhand von diesem kann eine Instanz der in Punkt 2 erstellten Sub- Klasse von <code>WorldState</code> kreiert werden. Anschliessend muss zwingen die Basis-Funktion <code>rebuildPersons</code> aufgerufen werden! Danach können die im Konstruktor erstellten Szenengraph Objekte anhand der erhaltenen Daten entsprechend manipuliert werden.

Tabelle 3: Eigenes Game - Zu implementierende Funktionen

4.4.1 Color Sphere

Bei der einfachsten Beispiel-Implementation war das Ziel in die Mitte des Raumes eine Sphäre zu platzieren. Die Farbe ist initial schwarz. Anschliessend soll sich die Farbe ab einer gewissen Hand-Distanz zum Zentrum der Kugel verändern. Die Farbe wird zufällig pro Person und Hand definiert. Anschliessend wird diese linear zur effektiven Distanz auf die Kugelfarbe aufaddiert. Bei vielen Spielern kann es somit schnell der Fall sein, dass die Kugel Weiss wird.

Für die Color Sphere Applikation wurden gemäss den oben definierten Sechs Punkten der Datentyp `VR_GAME_TYPE_COLOR_SPHERE` und die Klassen `ColorSphereWorldState`, `ColorSphereServer` sowie `ColorSphereGameScene` erstellt.

ColorSphereWorldState:

Die einzige sich wechselnde Eigenschaft der game-spezifischen Szene ist die Farbe der Kugel. Daher wurde in der Klasse nur einen Wert vom Typ `SLCol4f` definiert.

ColorSphereServer:

Der Konstruktor der Klasse generiert zufällige pro Person im Raum eine Farbe für jeweils die linke und die rechte Hand.

Die `simulate` Methode setzt die Sphärenfarbe auf Schwarz und iteriert anschliessend über alle Personen. Dabei wird jeweils anhand der Distanz von linker und rechter Hand zu einem definierten Punkt, die vorgängig definierte Farbe ab einem gewissen Minimum linear interpoliert, und auf die Farbe der Sphäre addiert.

Die `getWorldState` Methode erzeugt eine Instanz der `ColorSphereWorldState` Klasse. Anschliessend wird die einzige zu setzende Eigenschaft, die aktuelle Farbe der Sphäre, gesetzt.

ColorSphereGameScene:

Der Konstruktor der `GameScene` muss lediglich die Sphäre in den `_rootNode` setzen. Der Standort der Kugel muss für die korrekte Funktionsweise dieselbe sein, wie die Position, welche bei der oben beschriebenen `simulate` Funktion für die Berechnung verwendet wurde! Der Server verändert nur die Farbe der Kugel, daher wird als Referenz nur ein Pointer auf das Material der Kugel gespeichert. Dabei ist zu erwähnen, dass durch die SLProject Architektur Eigenschaften eines Materials problemlos nachträglich verändert werden können. Es kann jedoch nicht ohne grösseren Aufwand das Material eines bestehend Objekts im Szenengraph komplett neu gesetzt werden.

Die `updateFromWorldState` Methode erzeugt eine Instanz der `ColorSphereWorldState` Klasse anhand des mitgelieferten `RakNet::BitStreams`. Anschliessend wird die `rebuildPersons` Funktion der Basis-

Klasse aufgerufen. Zum Schluss muss nur die ambiente und diffuse Farbe des Materials der Sphäre anhand der `SLColor` Information aus dem Paket aktualisiert werden.

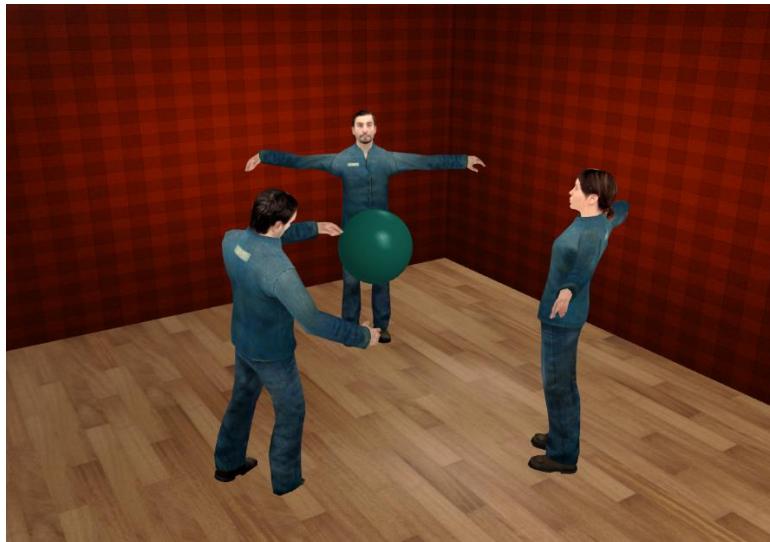


Abbildung 28: Color Sphere Game

4.4.2 Board Game Pattern

Um ein Brettspiel möglichst einfach implementieren zu können, wurde ein eigenes Pattern als Game-Implementation eingeführt. Dabei nutzen die beiden Brettspiele Tic Tac Toe und ein 3D Vier-Gewinnt dieses Pattern. Wichtig: Das Pattern beschränkt sich auf Brettspiele mit zwei Spielern.

Das Pattern hat keinen dedizierten `VR_GAME_TYP`. Dafür wurde sowohl auf dem Server, wie auf dem Client die beiden Sub-Klassen von `Server (BoardGameServer)` und `GameScene (BoardGameScene)` implementiert. Ebenfalls wurde eine Sub-Klasse von `WorldState (BoardGameWorldState)` angelegt. Die Idee dieses Pattern ist es die eigentliche Spiellogik wie das Abwechseln der beiden Spieler, das Auswählen eines Spielfeldes, das Beenden und Neu-Starten des Spiels, gemeinsame Anzeigen und Daten nur einmal definieren zu müssen.



Abbildung 29: Board Game Pattern – Reset-Boxes und Sieger Text

BoardGameWorldState:

Die über alle Brettspiele gemeinsamen Eigenschaften sind in der Paket-Klasse repräsentiert. Das ist zum einen eine Variable vom Typ `VR_BOARDGAME_STATE`, welche angibt, ob das Spiel am Laufen ist, falls beendet ob ein Gewinner feststeht, oder das Spiel unentschieden ausgegangen ist. Zudem ist die ID des Spielers, welcher aktuell an der Reihe ist, angegeben. Zu guter Letzt ist für jede Reset-Box die Farbe spezifiziert. Diese Boxen erscheinen jeweils für jeden Spieler, sobald das Spiel beendet ist.

BoardGameServer:

Der Konstruktor der Klasse muss zwingend mit einem Pointer auf eine `BoardGame` Instanz aufgerufen werden. Dabei wird im Konstruktor eine Referenz auf die Personen Matrizen (in der Basis Klasse abgefüllt) der `BoardGame` Instanz übergeben. Eine detaillierte Beschreibung der `BoardGame` Klassen ist unter 4.4.2.1 aufzufinden.

Die `simulate` Methode prüft als Erstes auf der `BoardGame` Instanz ob eine aktive Simulation am Laufen ist. Falls ja, wird die Funktion `doSimulateStep` aufgerufen. Läuft aktuell keine Simulation, werden Player Aktionen durchgeführt. Ist das Spiel noch nicht entschieden, wird die Funktion `play` auf der `BoardGame` Instanz aufgerufen (mit den jeweiligen Hand Positionen des Spielers, welcher aktuell an der Reihe ist). Ist das Spiel beendet, wird anhand der Hand Positionen für jeden Spieler die Funktion `handleRest` auf dem `BoardGame` aufgerufen. Das Zurücksetzen und Neustarten des Spiels kann folglich durch jeden Spieler parallel durchgeführt werden.

Die `getWorldState` Methode übergibt die Aufgabe direkt der Instanz von `BoardGame`, indem die voll virtuelle Funktion `getBoardState` aufgerufen wird.

BoardGameScene:

Die [GameScene](#) beinhaltet alle gemeinsamen Komponenten eines Brettspiels. Dabei handelt es sich um die Reset-Boxen und den Schriftzug, sobald das Spiel beendet ist. Zusätzlich wird der aktuelle Status des Spiels gespeichert. Dieser wird verwendet, um die Boxen zum richtigen Zeitpunkt anzuzeigen.

Der Konstruktor initialisiert jeweils das Material der Reset-Boxen. Da einmal sichtbar nur jeweils die Farbe für die Selektion der Boxen verändert werden soll, muss darauf eine Referenz gespeichert werden. Das Material besitzt standardmäßig eine Textur, damit ersichtlich ist, dass die Box für ein Zurücksetzen zuständig ist.

Die Methode [updateBoardGameFromWorldState](#) wird später von den Subklassen von [BoardGameScene](#) aufgerufen. Dabei aktualisiert die Methode einerseits die Personen in der Basis Klasse [GameScene](#). Zum anderen werden die Farben der Reset-Boxes gesetzt (auf dem Material).

Die zweite Funktion [changeGameState](#) wird ebenfalls auf den Subklassen aufgerufen. Basierend auf dem aktuellen Status des Spiels werden die Reset-Boxen und der Schriftzug mit dem Sieger entweder aus dem Szenengraph entfernt oder aber korrekt eingefügt. Beim Entfernen wird ebenfalls eine voll virtuelle Funktion für das Zurücksetzen der restlichen szenenspezifischen Objekte aufgerufen.

4.4.2.1 Board Game Klassen

In diesem Kapitel wird die im [BoardGameServer](#) angesprochene Klasse etwas detaillierter erläutert. Nachfolgend ein Klassen-Diagramm für die Übersicht.

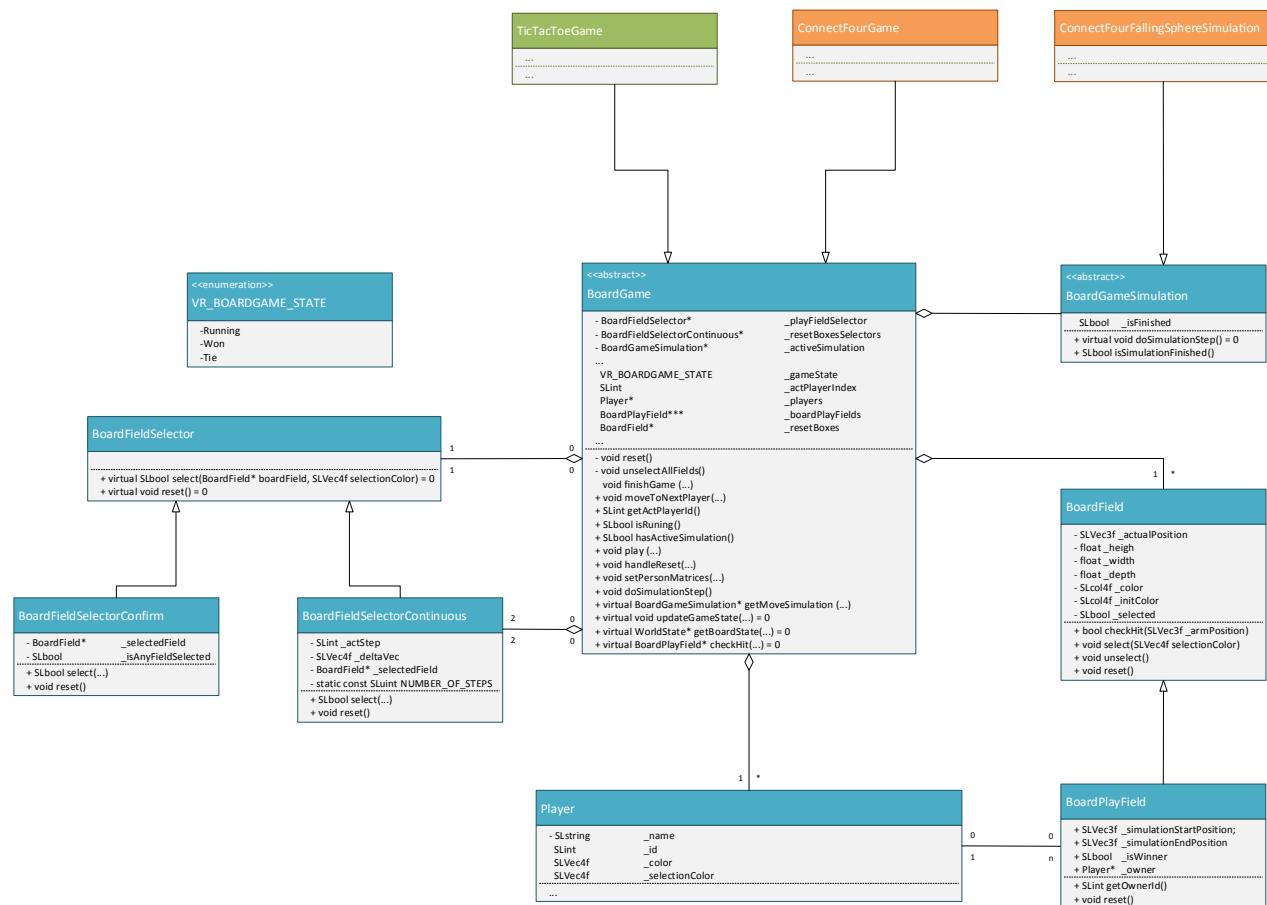


Abbildung 30: Klassendiagramm – BoardGame

Die blauen Klassen entsprechen dem Grundgerüst. Für die spezifische Implementation sind dann nur noch die grüne für das Tic Tac Toe, und die orangen für die 3D Vier-Gewinnt (Connect-Four auf English) notwendig.

BoardGame:

Diese Klasse stellt den Kern der Logik dar. Dabei wird nebst allen Spielfeldern (Instanzen von `BoardPlayField`) und den Spielern (Instanz von `Player`) auch pro Spieler eine Reset-Box (Instanzen von `BoardField`) gespeichert. Für die Definition der Eigenschaften der Spielfelder ist jedoch die konkrete Implementation (also Subklasse von `BoardGame`) zuständig. Nur diese weiss exakt wie viele Felder das Spiel benötigt, und wo diese positioniert sind. Ebenfalls in dieser Klasse vermerkt ist ein Verweis auf einen `BoardFieldSelector` (für die Selektion eines einzelnen Feldes) und pro Reset-Box ein Verweis auf einen `BoardFieldSelectorContinuous`. Der `_playFieldSelector` muss ebenfalls durch die konkrete Implementation definiert werden.

Die Methode `play` wird wie bereits beschrieben durch den `BoardGameServer` mit der Angabe der Hand-Positionen des aktuell sich an der Reihe befindenden Spielers aufgerufen. Die Methode ermittelt als Erstes, welches Feld selektiert wurde. Dabei wird die voll virtuelle Funktion `checkHit` aufgerufen. Der Grund liegt darin, dass je nach Brettspiel Implementation eine andere Selektion erwünscht sein kann. Anschliessend wird auf dem spezifizierten `_playFieldSelector` die Funktion `select` mit dem ermittelten Spielfeld, auch falls keines ermittelt wurde (NULL Referenz), aufgerufen. Diese Funktion gibt erst dann True zurück, wenn das Feld definitiv selektiert wurde. Die beiden Klassen `BoardFieldSelectorConfirm` und `BoardFieldSelectorContinuous` stellen dafür zwei verschiedene Selektionsarten dar. Wurde das ermittelte Feld definitiv selektiert, wird auf diesem Feld der Besitzer anhand des aktuell sich an der Reihe befindenden Spielers gesetzt. Anschliessend wird ermittelt, ob eine Simulation notwendig ist. Hierfür wird die Funktion `getMoveSimulation` aufgerufen. Standardmäßig wird sie einen NULL Pointer zurückgeben. Die virtuelle Funktion kann jedoch falls erwünscht eine eigene Simulation setzen. Zu guter Letzt wechselt das Spiel zum nächsten Spieler mit der Funktion `moveToNextPlayer`.

Die Funktion `handleReset` wird ebenfalls durch den `BoardGameServer` aufgerufen. Dabei wird auf dem pro Spieler definierten Rest-Box-Selektor, die Funktion `select` mit entweder einem NULL Argument oder aber falls der Player seine Hand in der Box positioniert, mit dieser Reset-Box (als `BoardField`) als Argument aufgerufen. Gibt diese Methode True zurück, bedeutet es, dass das Spiel zurückgesetzt werden soll. Dafür wird die Funktion `reset` aufgerufen. Diese Funktion ruft die `reset` Funktion auf allen Reset-Boxen (Felder und Selektoren), dem Spielfeld-Selektor und allen Spielfeldern auf.

Nachdem ein Spielfeld definitiv selektiert wurde, muss der Spieler gewechselt werden. Dabei wird die Funktion `moveToNextPlayer` aufgerufen. Diese Funktion muss jedoch zusätzlich überprüfen, ob sich der Spielstatus geändert hat. Dafür wird die voll virtuelle Funktion `updateGameState` aufgerufen. Da jedes Spiel nach seinen eigenen Regeln läuft, weiss auch nur diese, ob das Spiel nun beendet ist oder nicht. Der Gewinner, falls es einen gibt, entspricht dann jeweils der aktuellen Player ID.

BoardFieldSelector:

Der Selektor ist dafür zuständig ein Feld auszuwählen. Dabei gab es zwei grundsätzliche Ideen.

1. Um ein Feld auszuwählen, sollte der Player einmal seine Hand in das Feld legen, sie wieder wegnehmen und anschliessend exakt in dasselbe Feld legen. Nach dem ersten Hineinfassen sollte sich das Feld blau färben (= ausgewählt). Nach dem Bestätigen sollte das Feld die Farbe des Spielers annehmen. Die Funktionalität wurde in der Klasse `BoardFieldSelectorConfirm` implementiert.
2. Als Alternative ist in der Klasse `BoardFieldSelectorContinuous` eine Methode implementiert, wobei ein Spieler eine gewisse Zeit seine Hand in ein Feld legen soll, ohne dieses zu verlassen. Dabei färbt sich das Feld langsam mit der Farbe des Spielers bis das Feld selektiert ist.

4.4.2.2 Eigene Brettspiel Implementation

Aus der Beschreibung der Board-Game-Klassen ist nun ersichtlich, dass für eine eigene Board Game Implementation einige Schritte durchgeführt werden müssen:

1. Erweitern des `VR_GAME_TYPE` Enums um einen weiteren Typen (Lib-VirtualRoomCommon).
2. Erstellen einer Sub-Klasse von `BoardGameWorldState` (Lib-VirtualRoomCommon).
3. Kreieren der Sub-Klasse von `BoardGame` (ch14_VirtualRoomServer). **Wichtig:** Dem Parent-Konstruktor muss den beim Punkt 1 erstellten Typen angegeben werden. Zusätzlich müssen die Funktionen `updateGameState`, `getBoardState` und `checkHit` zwingend überschrieben werden. Alternativ kann die `getMoveSimulation` Methode ebenfalls überschrieben werden.
4. Erweitern des Switch-Blockes in der `main` Methode der `VRServerStarter` Datei (ch14_VirtualRoomServer). Dabei wird anhand des spezifizierten Typen eine Instanz von `BoardGameServer` mit der neu erstellten `BoardGame` Instanz als Argument erstellt.
5. Kreieren einer Sub-Klasse von `BoardGameScene` (ch4_VirtualRoom). Wichtig: Die Funktionen `updateFromWorldState` und `resetSpecificGame` müssen zwingend überschrieben werden.
6. Erweitern des Switch-Blockes in der Funktion `postSceneLoad` der Klasse `VRSceneView` (ch14_VirtualRoom). Dabei wird ebenfalls anhand des im Punkt 1 definierten Typen eine Instanz der neu erstellten Sub-Klasse im Punkt 5 erstellt.

In den neu zu kreierenden Klassen wird die Logik in den folgenden Funktionen aufgeteilt:

Klasse	Funktion	Zweck
Subklasse von <code>BoardGame</code> (Punkt 3)	Konstruktor	Setzen der Spielereigenschaften (Farbe / Name / Selektionsfarbe). Dem Basis-Konstruktor wird eine Referenz auf ein <code>BoardFieldSelector</code> angegeben. Zusätzlich werden dem Basis-Konstruktor die Dimensionen des Spielfeldes angegeben. Achtung: Das Pattern bezieht sich nur auf 3-dimensionale Spielfelder. Werden nur zwei davon verwendet, muss die Anzahl Levels (Z-Indices) auf 1 gesetzt werden. Der Konstruktor kann nun selber die Grösse aller Felder inkl. deren Positionen definieren. Dabei wird auf das 3D Array <code>_boardPlayFields</code> zugegriffen. Die erste Dimension entspricht dem Level (Z-Index). Die zweite der Spalte (Y-Index) und die dritte Dimension entspricht der Zeilenangabe (X-Index).
Subklasse von <code>BoardGame</code> (Punkt 3)	<code>updateGameState</code>	Der Funktion wird der Index in allen drei Dimensionen (Z, Y und X) des Feldes angegeben, welches zuletzt bearbeitet wurde. Anhand von diesem soll die Brettspiel Implementation herausfinden, ob das Game abgeschlossen ist. Trifft dies zu, muss die Funktion unterscheiden zwischen einem Unentschieden oder ob es durch einen Spieler gewonnen wurde.
Subklasse von <code>BoardGame</code> (Punkt 3)	<code>getBoardState</code>	Erzeugen einer Instanz der in Punkt 2 definierten Sub-Klasse von <code>BoardGameWorldState</code> . Diese Klasse mit allen relevanten Welt-Status Informationen vorgängig ergänzen und in der Methode bestücken. Wichtig: Die Daten der Basis-Klasse (<code>GameState</code> / <code>ActPlayerId</code> / <code>ResetBoxColors</code>) müssen ebenfalls gesetzt werden. Instanz dem Aufrufer zurückgeben.
Subklasse von <code>BoardGame</code> (Punkt 3)	<code>getMoveSimulation</code> (optional)	Optional kann diese Funktion überschrieben werden. Die Funktion gibt eine Instanz einer Sub-Klasse von <code>BoardGameSimulation</code> zurück. Diese Simulation wird bei jedem Selektieren (nicht auswählen, sondern bestätigen) erzeugt.
Subklasse von <code>BoardGameScene</code> (Punkt 5)	Konstruktor	Erzeugen der für die Szene weiteren notwendigen Szenengraph Objekte. Diese Objekte müssen, falls durch die Server-Anwendung manipulierbar, auch zu einem späteren Zeitpunkt

		erreichbar sein (Pointer speichern). Zusätzlich müssen die Objekte in den <code>_rootNode</code> gehängt werden.
Subklasse von <code>BoardGameScene</code> (Punkt 5)	<code>updateFromWorldState</code>	Als Argument wird der Funktion der <code>RakNet::BitStream</code> übergeben. Anhand von diesem kann eine Instanz der in Punkt 2 erstellten Sub- Klasse von <code>BoardGameState</code> kreiert werden. Anschliessend muss zwingen die Basis-Funktion <code>updateBoardGameFromWorldState</code> aufgerufen werden! Danach können die im Konstruktor erstellten Szenengraph Objekte anhand der erhaltenen Daten entsprechend manipuliert werden. Falls sich der <code>gameState</code> geändert hat, muss der Schlusstext definiert werden. Anschliessend wird die Basis-Klassen Funktion <code>changeGameState</code> für die Darstellung der Reset-Boxen aufgerufen.
Subklasse von <code>BoardGameScene</code> (Punkt 5)	<code>resetSpecificGame</code> (optional)	Führt Aktionen wie das Entfernen von weiteren Objekten aus dem Szenengraphen, nebst den Reset-Boxen und dem Text, beim zurücksetzen des Spiels.

Tabelle 4: Eigenes Brettspiel - Zu implementierende Funktionen

4.4.2.3 Tic Tac Toe

Als erstes einfaches Brettspiel wurde das Kultspiel Tic Tac Toe implementiert. Das Spiel sollte die gesamte Funktionalität mit dem Auswählen und Bestätigen von Feldern, sowie einen Spielablauf demonstrieren.

Für das Spiel wurde der Daten-Typ `VR_GAME_TIC_TAC_TOE` und die Klassen `TicToeWorldState`, `TicToeGame`, sowie `TicToeGameScene` erstellt.

TicToeWorldState:

Einige brettspielspezifische Eigenschaft ist beim Tic Tac Toe die Farbe jedes einzelnen Spielfeldes. Aus diesem Grund wird für die Übertragung ein Array von `SLCol4f` serialisiert.

TicToeGame:

Im Konstruktor werden die jeweiligen Farben der Spieler zugewiesen. Zudem werden die Feldergrößen und Positionen definiert. Da die Gamescene dieselben Felder für den Szenengraphen definieren muss, wurde in der Common-Lib eine Datei `VRTicToeConstants.h` mit diversen Konstanten angelegt.

Die `updateGameState` Methode überprüft ob sich das zuletzt selektierte Feld horizontal, vertikal oder diagonal mit Feldern vom selben Spieler befindet. Ist dies der Fall, wird das Spiel mit dem Status `VR_BOARDGAME_STATE WON` beendet. Wurde keine Winner-Kombination gefunden, wird überprüft ob sich noch nicht selektierte Felder auf dem Spielfeld befinden. Falls nicht, wird das Spiel mit `VR_BOARDGAME_STATE TIE` beendet. Anderer falls wird das Spiel fortgesetzt.

Die `getWorldState` Methode erzeugt eine Instanz von `TicToeWorldState`. Pro Spielfeld wird die Farbe des besitzenden Spielers abgespeichert. Hat das Feld keinen Besitzer, wird die aktuelle Spielfeld Farbe entnommen. Dies ist für die Selektierung eines neuen Feldes notwendig. Die `BoardGame` übergreifenden Eigenschaften müssen ebenfalls gesetzt werden.

Als Letztes muss noch die `checkHit` Methode implementiert werden. Für die Überprüfung wird über alle Spielfelder iteriert. Dabei wird auf jeder `BoardPlayField` anhand der Funktion `checkHit` für die linke und die rechte Hand überprüft, ob sich diese in der Box befindet oder nicht. Die `checkHit` Funktion in der `BoardPlayField` Klasse kennt die Position und die Dimension des Feldes und kann daher selber diese Überprüfung durchführen.

TicTacToeGameScene:

Der Konstruktor der Klasse erstellt das eigentlich sichtbare Spielbrett. Dabei werden Referenzen auf das Material jeder Box, welches ein Spielfeld darstellt, zwischengespeichert. Die `SLBox` Objekte werden in den `_rootNode` eingefügt. Damit die Anzeige auf dem Client mit den Berechnungen auf dem Server übereinstimmen, werden alle Größen und Distanzen aus der `VRTicTacToeConstants.h` Datei in der Common-Lib verwendet.

Die `updateFromWorldState` Methode erzeugt eine Instanz der `TicTacToeWorldState` Klasse anhand des mitgelieferten `RakNet::BitStreams`. Danach wird als Erstes auf der Basis-Klasse `BoardGameScene` die Funktion `updateBoardGameFromWorldState` aufgerufen. Anschliessend wird für jedes Feld die ambiente und diffuse Farbe des Materials basierend auf den Daten im erhaltenen Paket aktualisiert. Zum Schluss muss, falls sich der `GameState` geändert hat, die Funktion `changeGameState` in der Basis-Klasse aufgerufen werden.



Abbildung 31: Tic Tac Toe Spiel - Links: Am Laufen, Feld selektiert - Recht: Gewonnen

4.4.2.4 3D Vier-Gewinnt

Als zweite Brettspiel Implementation kam uns das Spiel 3D Vier-Gewinnt in den Sinn. Das Spiel eignet sich optimal für die Verwendung der Virtual Room Anwendung. Das Spielbrett hat 4×4 Zylinder, auf welche jeweils 4 Kugeln passen. Jeder Spieler kann abwechselungsweise eine Kugel auf einen der Zylinder platzieren, wobei die Kugel dann auf den untersten freien Platz fällt. Ziel des Spiels ist es, vor dem Gegner eine vierer Reihe mit den eigenen Kugeln entweder horizontal, vertikal oder diagonal und dass in allen drei Dimensionen zu bilden. Die Schwierigkeit liegt dabei im dreidimensionalen Denken und der verschiedenen Sichtweisen aus unterschiedlichen Blickwinkeln.

Die Implementation des Spiels lehnt sich an die im Punkt 0 beschriebene Anleitung. Dabei wurde jedoch eine kleine Modifikation an der Denkweise durchgeführt. Als Spielfeld wird ein möglicher Platz einer Kugel betrachtet. Ausgewählt wird im Game ein Zylinder und nicht ein Spielfeld. Die Berechnungen für die Hit-Detection werden vom Server selber durchgeführt und auch die Farben der Zylinder werden eigens ermittelt.

Für das Spiel wurde der Daten-Typ `VR_GAME_TYPE_CONNECT_FOUR` und die Klassen `ConnectFourWorldState`, `ConnectFourGame`, `ConnectFourFallingSphereSimulation`, sowie `ConnectFourGameScene` erstellt.

ConnectFourWorldState:

Da die Selektierung nicht direkt über die Felder, sondern über die Zylinder erfolgt, werden die Farben aller Zylinder dem Client übertragen. Zusätzlich wird die Anzahl Kugeln angegeben. Pro Kugel muss neben der Position in der Welt und deren Farbe, noch ein boolescher Wert spezifiziert werden, welcher angibt, ob es sich bei der Kugel um eine Sieges-Kugel handelt oder nicht. Dieser Wert wird verwendet, um die Sieger-Kombination auf dem Client aufleuchten zu lassen.

ConnectFourGame:

Im Konstruktor werden die jeweiligen Farben der Spieler zugewiesen. Zudem werden die Zylindergrößen und Positionen definiert. Da die Gamescene dieselben Werte für den Szenengraphen definieren muss, wurde in der Common-Lib eine Datei `VRConnectFourConstants.h` mit diversen Konstanten angelegt. Neben den Zylindern müssen die Positionen und Größen der `BoardPlayField` Objekte spezifiziert werden. Dabei entspricht jedes Spielfeld einer möglichen Position einer Kugel. Neben diesen beiden Eigenschaften muss zwingend auch die Simulations-Start und Simulations-End Position definiert werden. Anhand von diesen wird das Herunterfallen einer Kugel nachgeahmt.

Die `updateGameState` Methode überprüft ob sich die zuletzt hinzugefügte Kugel (zuletzt selektiertes Feld) horizontal, vertikal oder diagonal nur mit Feldern vom selben Spieler befindet. Ist dies der Fall, wird das Spiel mit dem Status `VR_BOARDGAME_STATE WON` beendet. Wurde keine Winner-Kombination gefunden, wird überprüft ob sich noch nicht selektierte Felder auf dem Spielfeld befinden. Falls nicht, wird das Spiel mit `VR_BOARDGAME_STATE TIE` beendet. Andererfalls wird das Spiel fortgesetzt. Wichtig: Wurde bei der Überprüfung vier siegende Felder gefunden, werden auf diesen `BoardPlayFields` die Eigenschaft `_isWinner` auf True gesetzt.

Die `getWorldState` Methode erzeugt eine Instanz von `ConnectFourWorldState`. Zum einen wird pro Zylinder die Farbe ermittelt. Da jedoch vom `BoardGame` Pattern her nur `BoardFields` selektiert werden, wird die Farbe jeweils aus den vier Feldern pro Zylinder ermittelt. Ist eins dieser Felder selektiert, wird dessen Farbe als Zylinderfarbe verwendet. Als Nächstes wird in der Funktion die Anzahl Kugeln errechnet. Dabei entspricht jedes `BoardField` mit einem Besitzer einer Kugel. Als Farbe wird die Besitzerfarbe kopiert und die Position entspricht der Spielfeld-Position. Die `BoardGame` übergreifenden Eigenschaften werden ebenfalls gesetzt.

Wie bereits beschrieben, muss beim Hinzufügen einer Kugel eine Simulation stattfinden. Dafür wurde die Methode `getMoveSimulation` überschrieben. Die Methode gibt dabei eine Referenz auf Instanz der Klasse `ConnectFourFallingSphereSimulation` zurück. Der Klasse wird eine Referenz auf das aktuell ausgewählte Spielfeld übergeben.

Zu guter Letzt musste in der `ConnectFourGame` Klasse noch die Funktion `checkHit` überschrieben werden. Diese Funktion überprüft als Erstes, ob sich eine der Hände im obersten Viertel eines Zylinders befindet. Trifft dies zu, wird das erste Feld von unten in diesem Zylinder, welches keinen Besitzer hat, zurückgegeben.

ConnectFourFallingSphereSimulation:

Die Simulation bekommt als Argument ein `BoardField`. Dieses besitzt eine aktuelle, eine simulations-start und -end Position. Die aktuelle Position wird dem Client übermittelt. Somit wird im Konstruktor die aktuelle Position auf die simulations-start Position gesetzt. Anschliessend wird basierend auf der Zielstufe, die Anzahl Simulations-Schritte und ein delta Vektor berechnet.

Bei jedem `doSimulate` Aufruf wird nun für die errechnete Anzahl Simulations-Schritte die aktuelle Position um den Delta Vektor addiert. Beim letzten `doSimulate` Aufruf wird die aktuelle Position des Feldes auf die simulations-end Position gesetzt.



Abbildung 32: 3D Vier-Gewinn - Simulation der herunterfallenden Kugel

ConnectFourGameScene:

Der Konstruktor erstellt das leere sichtbare Spielbrett. Dabei handelt es sich um einen Sockel mit 4x4 Zylinder darauf positioniert. Damit die Anzeige auf dem Client mit den Berechnungen auf dem Server übereinstimmt, werden alle Größen und Distanzen wie beim Tic Tac Toe aus der `VRConnectFourConstants.h` Datei in der Common-Lib entnommen. Da die Zylinder in der Farbe wechseln können, muss pro Objekt die Referenz auf das Material zwischengespeichert werden. Zusätzlich werden alle Materialien der maximalen Anzahl `SLSpheres` erstellt und zwischengespeichert. Der Sockel und die Zylinder werden in den Szenengraph eingefügt. Der Client speichert sich die aktuelle Anzahl Kugeln auf dem Spielbrett. Diese wird initial auf 0 gesetzt.

Die `updateFromWorldState` Methode erzeugt eine Instanz der `ConnectFourWorldState` Klasse. Danach wird als Erstes die Funktion `updateBoardGameFromWorldState` aufgerufen. Anschliessend wird für jeden Zylinder die ambiente und diffuse Farbe des Materials basierend auf den Daten im erhaltenen Paket aktualisiert. Anhand der momentanen Anzahl Kugeln auf dem Spielbrett und der erhaltenen Anzahl aus dem Paket, wird ermittelt ob neue dazu gekommen sind. Trifft dies zu, muss die Farbe und Position aller bestehenden Kugeln aktualisiert werden, da die Reihenfolge geändert haben könnte. Der ambiente und diffuse Anteil wird aus dem Daten Paket entnommen. Falls es sich bei der Kugel um eine Sieger-Kugel handelt, wird zusätzlich noch die Emission für ein Leuchten der Sphäre gesetzt. Neue Kugeln werden mit der korrekten Position und Farbe erstellt und in den Szenengraphen eingefügt. Zum Schluss muss, falls sich der `GameState` geändert hat, die Funktion `changeGameState` in der Basis-Klasse aufgerufen werden.

Da beim Zurücksetzen des 3D Vier-Gewinn Spiels zusätzliche Aktionen auf dem Client ebenfalls durchgeführt werden müssen, wurde die Funktion `resetSpecificGame` implementiert. Dabei werden alle momentanen Sphären aus dem Szenengraph entfernt.

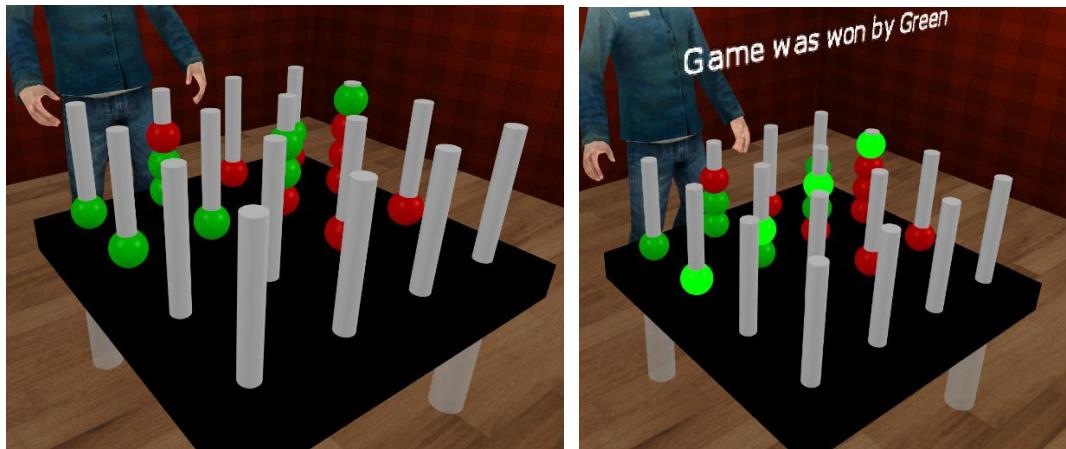


Abbildung 33: 3D Vier-Gewinnt - Links: Am Laufen, Zylinder selektiert - Recht: Gewonnen

4.4.3 Turn Table

Als letzte Beispiel Applikation wurde eine Turn Table (Drehtisch) implementieren. Die Idee dabei ist, dass alle Player im Raum eine detaillierte Sicht auch ein beliebiges Mesh haben, welches auf einem drehbaren Tisch in der Mitte des Raums platziert wird. Dabei ist die Komplexität im Vergleich zu den Brettspielen wieder um einiges reduziert worden. Zu erwähnen ist auch, dass das eigentliche Mesh nicht über das Netzwerk versendet wird. Einzig die Rotation im Raum wird verändert. D.h. die *GameScene* auf dem Client lädt beim Starten das Mesh in den Raum. Das initiale Versenden des Meshes vom Server aus könnte als Erweiterung implementiert werden und würde etwas mehr an Flexibilität anbieten.

Für die Turn Table wurde der Daten-Typ *VR_GAME_TYPE_TURN_TABLE* und die Klassen *TurnTableWorldState*, *TurnTableServer*, sowie *TurnTableGameScene* erstellt.

TurnTableWorldState:

Als einzige sich wechselnde Eigenschaft beim Turn Table ist die Ausrichtung des Tisches (und des darauf liegenden Meshes). Dafür reicht es ein einzelnes Quaternion im Paket dem Client zukommen zu lassen.

TurnTableServer:

Der Konstruktor muss keine weiteren Aufgaben durchführen, da die initiale Ausrichtung einer 0 – Rotation entspricht.

Die *simulate* Methode wird jeweils pro Person für die linke und die rechte Hand-Position eine Hit-Detection mit der Tischplatte durchführen. Da die Tischplatte auf dem Server nicht modelliert wird, werden die Berechnungen anhand von fixen Werten jeweils mit einem Minimum und Maximum in y-Richtung und einer maximalen Distanz in den zwei Dimensionen X und Z zur Y-Achse durchgeführt. Pro linke Hand über alle Player wird die Rotation in der *simulate* Methode um 1 Grad im Gegenuhrzeigersinn verändert. Pro rechte Hand über alle Player um 1 Grad im Uhrzeigersinn. Dadurch wird sich der Tisch schneller drehen, falls mehrere Player gleichzeitig dieselbe Hand in den Tisch legen. Andererseits bleibt der Tisch unbewegt, falls linke und rechte Hand einer Person zur selben Zeit im Tisch liegen.

Die *getWorldState* Methode erzeugt eine Instanz der *TurnTableWorldState* Klasse. Anschliessend wird die momentan berechnete Rotation als Variable gesetzt.

TurnTableGameScene:

Der Konstruktor erstellt den Drehtisch anhand eines Sockels und der Tischplatte. Dabei haben die beiden Zylinder unterschiedliche Materialien. Der gesamte Tisch entspricht einer *SLGroup*, welche zu einem späteren Zeitpunkt im Raum rotiert werden muss. Daher wird eine Referenz zu dieser Gruppe als Pointer Variable gespeichert. Der Inhalt auf dem Tisch entspricht einer weiteren Gruppe. Immer noch im Konstruktor wird ein vordefiniertes Mesh (zur Veranschaulichung der „Teapot“) in diese

Inhalt-Gruppe importiert. Durch die Szenengraphen Architektur wird beim Anwenden einer Rotation auf der Tisch-Gruppe auch der gesamte Inhalt rotiert.

Die `updateFromWorldState` Methode erzeugt eine Instanz der `TurnTableWorldState` Klasse anhand des mitgelieferten `RakNet::BitStreams`. Anschliessend wird wie bei allen Implementationen die `rebuildPersons` Funktion der Basis-Klasse aufgerufen. Als Letztes muss einzig die im Paket gespeicherte Rotation als solche direkt auf der Tisch-Gruppe gesetzt werden (nicht darauf multiplizieren, sondern überschreiben).

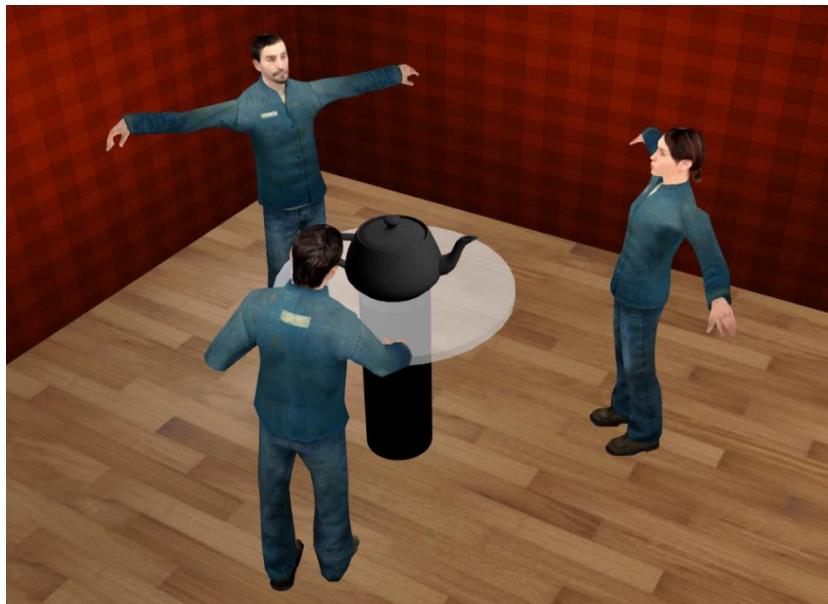


Abbildung 34: Turn Table mit dem „Teapot“ als Mesh

4.5 Rigged Mesh

Für die Darstellung des Spielers in der Applikation gab es verschiedene Möglichkeiten. Wir zogen zwei Methoden in Betracht: Die Erste davon wäre ein Spielermodell aus der Punktewolke der Kinect zu generieren.



Abbildung 35: Punktewolke Bild der Kinect, Quelle: Kinect 2 Tech Demo

Die Zweite, ein traditionellerer Ansatz, ist die Verwendung eines sogenannten rigged Meshes, wie es auch in den meisten 3D-Spielen der Fall ist. Wir entschieden uns für die zweite Variante aus folgenden Gründen:

1. Weniger fehleranfällig, ein dynamisches Mesh aus einer Punktewolke flackert oft stark.
2. Daten auch hinter der Kamera vorhanden, bei der Punktewolke fehlen jegliche Daten, welche für die Kamera verdeckt sind.

Die erste Methode wäre jedoch eine mögliche Ergänzung für das Projekt in der Zukunft.

4.5.1 Was ist ein rigged Mesh

Ein rigged Mesh ist ein Mesh, welches von einem Skelett verformt werden kann.



Abbildung 36: Rigged Mesh mit sichtbarem Skelett in der Modelling-Software Blender

Solche Modelle werden von Artists kreiert und mit den nötigen Gelenken, sogenannten Bones oder auch Joints versehen. Das Kreieren dieses Skelettes wird als Rigging bezeichnet.

4.5.2 Skinning

Nachdem das Skelett für ein Mesh kreiert ist, kommt der Skinning-Prozess. Skinning bedeutet in diesem Fall das Zuweisen von Vertices zu Bones (ein Vertex ist ein einzelner Punkt in einem Gitternetz-Modell). Ein einzelner Vertex kann von mehreren Bones mit unterschiedlicher Gewichtung beeinflusst werden, wobei die Summe dieser Gewichtungen 1 betragen sollte. Im Normalfall wird ein Vertex von einem einzelnen Bone beeinflusst. Bei Übergängen zwischen zwei Bones, wie zum Beispiel beim Ellbogen werden Punkte oft auch von zwei beeinflusst, was dazu führt dass sich ein Knick zwischen den Knochen bildet. Wären Vertices strikt nur einem Bone zugewiesen, so könnte es in Bereichen wie eben dem Ellbogen zu Überlappungen des Meshes kommen, was die Glaubwürdigkeit stark schwächen kann.

In der unteren Abbildung sind die Gewichtungen für den Oberarmknochen farblich hervorgehoben. Rot bedeutet 100% Einfluss, Blau 0%.

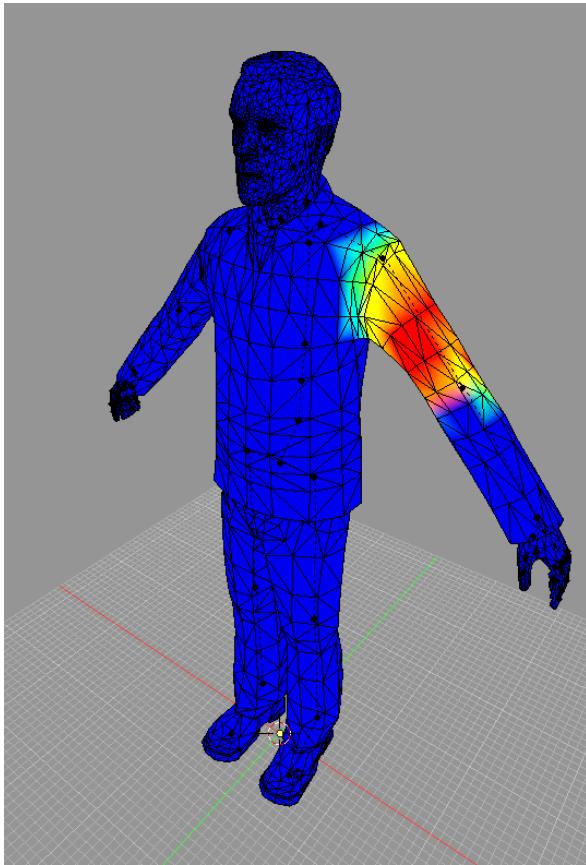


Abbildung 37: Bone Gewichte visualisiert, rot = 1, blau = 0

4.5.3 Importer

Um Modelle, welche mit Modellierungs-Software erstellt wurden, nutzen zu können, müssen wir die Daten in die Virtual Room Applikation importieren. Es gibt eine grosse Anzahl an Fileformaten, welche 3D-Modelle beschreiben und die genutzt werden könnten. Wir brauchen ein Fileformat, welches die Hierarchie einer Szene beschreiben kann, sowie die Skelette der Meshes. Wir entschieden uns die Open Source Library Assimp zu verwenden, da sie bereits eine grosse Anzahl an Fileformaten unterstützt und in eine einfache Datenstruktur laden kann.

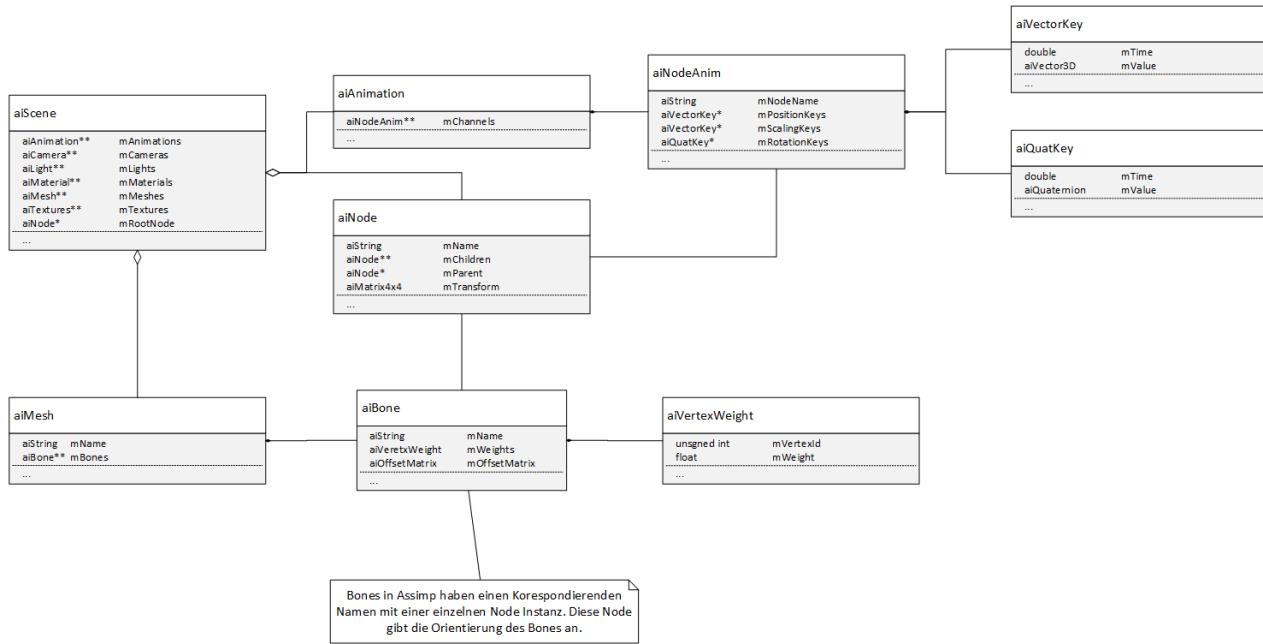


Abbildung 38: Assimp Import Library Daten Strukturen für das Laden von Szenen

Assimp lädt alle Elemente einer Szene in eine `aiScene`-Instanz. Diese Instanz enthält einen Szenengraph und Referenzen auf alle Meshes, Materialien, Texturen, Kameras, Lichter und Animationen. Für unsere Zwecke benötigen wir nur Meshes, Materialien, Texturen und den Szenengraphen. Wenn ein rigged Mesh geladen wird, verfügt dieses über eine Liste von `aiBones`. Die Namen dieser Bones korrespondieren mit einem Namen einer `aiNode` im Szenengraphen.

SLImporter ist die Wrapper-Klasse um Assimp, welche importierte Daten in die SLProject Datenstrukturen schreibt. Der Import Prozess läuft wie folgt ab:

1. Wir laden alle Materialien und Texturen in SLProject Datenstrukturen
2. Wir laden alle Nodes des Szenengraphen in einen Baum von ***SLBoneNodes***
3. Wir laden alle Meshes
 - a. In diesem Schritt greifen wir auf die Materialien und Texturen zurück und setzen die Materialien des Meshes direkt.
 - b. Bone Referenzen, falls vorhanden, werden ebenfalls direkt dem Mesh hinzugefügt. Diese Referenzen zeigen auf den im zweiten Schritt importierten ***SLBoneNode***-Baum.
4. Wir traversieren den ***SLBoneNode***-Baum ein weiteres Mal, um die relativen Pose Matrizen zu finden (diese Matrizen beschreiben die Pose des Meshes in welcher es geriggt wurde).
5. Als Letztes importieren wir Animationen, falls vorhanden.

Der 5. Schritt wurde in diesem Projekt nicht implementiert, da es grundlegende Änderungen am vorhandenen Animationssystem im SLProject vorausgesetzt hätte, und Keyframe-Animationen wurden nicht benötigt. Jedoch wurde zum Testen der rigged Mesh Klasse ein solches Animationssystem in einem externen Projekt implementiert. Als Implementationshilfe diente hierzu das Buch „Real-Time 3D Character Animation with Visual C++“ [8].

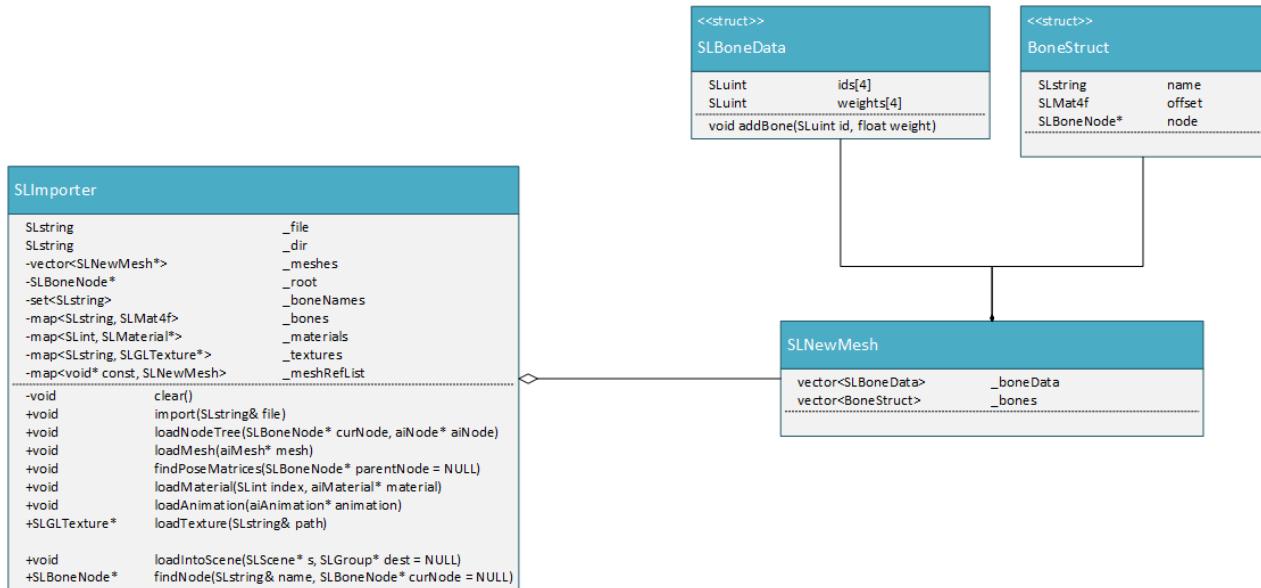


Abbildung 39: SLImporter, Wrapper-Klasse um Assimp.

4.5.3.1 Mesh Klasse

SLNewMesh ist eine zweite Mesh-Implementation im SLProject, welche in einer nächsten Iteration die alte Mesh Klasse ersetzen soll. Zur Zeit der Abgabe war diese Klasse jedoch noch nicht soweit fertiggestellt, um die alte komplett abzulösen zu können. Die **SLNewMesh** Klasse bietet die Möglichkeit **SLBoneNodes** auf Vertices zu binden (Skinning). Die dafür benötigten Bones und Bone-Gewichte werden im **SLImporter** beim Laden des Meshes gesetzt.

Die Mesh-Klasse unterstützt bis zu vier Bones pro Vertex, in den meisten Fällen werden höchstens zwei Bones pro Vertex vorhanden sein, dies sollte also genügend Freiheit bieten. Jedoch könnte diese Anzahl zu einem späteren Zeitpunkt erhöht werden, da die meisten Modelling-Programme kein Limit vorgeben.

4.5.3.2 Bones

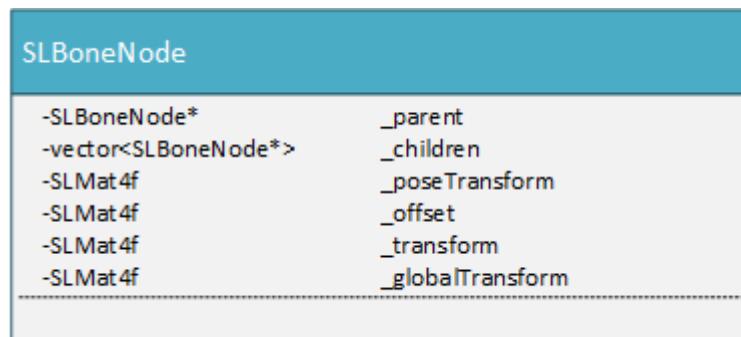


Abbildung 40: SLBoneNode Klassen-Member

Ein Bone ist ein Teil eines Baumes und besitzt eine relative Transformation. Aus diesen hierarchischen relativen Transformationen wird die globale Transformation berechnet. Soweit funktionieren die Bones wie normale Nodes eines Szenengraphen. Bones besitzen jedoch noch eine zusätzliche Matrix, die Offset Matrix. Die Offset Matrix ist die Inverse der globalen Transformation im Mesh-Raum. Sie dient dazu die Vertices welche von einem Bone beeinflusst werden in den korrekten Bone-Raum zu

verschieben, um die Transformation auf die Vertices anzuwenden. Die finale Bone-Matrix bildet sich also aus dem folgenden Produkt: `_globalTransform * _offset;`

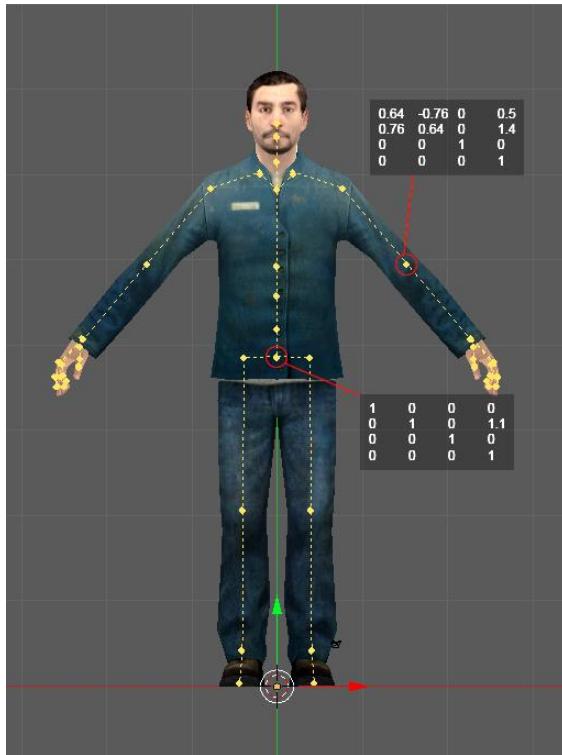


Abbildung 41: Absolute Bone Transformationen im Mesh-Raum, die Inverse ist die Offset-Matrix

4.5.3.3 Shader

Die finale Vertex Transformation wird in einem Vertex-Shader gemacht. Wir erlauben ein Maximum von vier Bones pro Vertex. Pro Mesh wird eine Liste von Bone-Matrizen erstellt (`u_bones`) welche die Bone-Transformationen in Weltkoordinaten enthält. Eine mögliche Optimierung wäre Meshes welche die gleichen Skelette verwenden nacheinander zu rendern, um die Liste `u_bones` nur einmal setzen zu müssen.

```

in      ivec4 a_boneIds;
in      vec4 a_boneWeights;

const    int   MAX_BONES = 100;
uniform  mat4 u_bones[MAX_BONES];

void main(void)
{
    mat4 boneTransform =
        u_bones[a_boneIds.x] * a_boneWeights.x
        +
        u_bones[a_boneIds.y] * a_boneWeights.y
        +
        u_bones[a_boneIds.z] * a_boneWeights.z
        +
        u_bones[a_boneIds.w] * a_boneWeights.w;

    //...
}

```

Ein wichtiger Unterschied bei diesem Vertex-Shader zu den andern Shader in SLProject ist der Gebrauch des „in“-Keywords. SLProject verwendet „attribute“ und „varying“ Keywords um möglichst viele Versionen von GLSL zu unterstützen. Dies ist bei diesem Vertex Programm nicht möglich, da `ivec4` als Vertex Attribut Typ nicht unterstützt werden vor GLSL-Version 1.3. Alternativ könnte man die

Bone-Ids ebenfalls als vec4 angeben und zu int casten, diese Variante könnte aber zu Performance-Einbussen führen.

4.5.3.4 Frustum Culling für rigged Meshes

Bei rigged Meshes muss beachtet werden, dass das Frustum-Culling, also das Nichtrendern von Objekten welche nicht im Sichtbarkeitsbereich der Kamera sind, nicht mehr mit einer simplen Axis-Aligned-Bounding-Box (AABB) gelöst werden kann. Die Vertices werden nach Bone Gruppen unterschiedlich transformiert und die AABB des Meshes kann an einem ganz anderen Ort sein als das sichtbare Modell. Die einfachste Lösung hierfür ist es den Bones selbst Bounding-Boxen zu geben. Dafür muss jedoch beachtet werden welche Vertices von diesem Bone beeinflusst werden, und diese Vertices müssen bei der Berechnung der Bounding-Box miteinbezogen werden.

4.5.4 Kinect Data Binding

Die Daten der Kinect müssen nun auf das Skelett des importierten Meshes angewendet werden. Bei diesem Schritt gibt es zahlreiche mögliche Fehlerquellen. Im unteren Bild ist einer der Unterschiede der beiden Skelette gut ersichtlich. Die Knochen, welche die Hüfte mit den Beinen verbinden, haben einen unterschiedlichen Winkel.

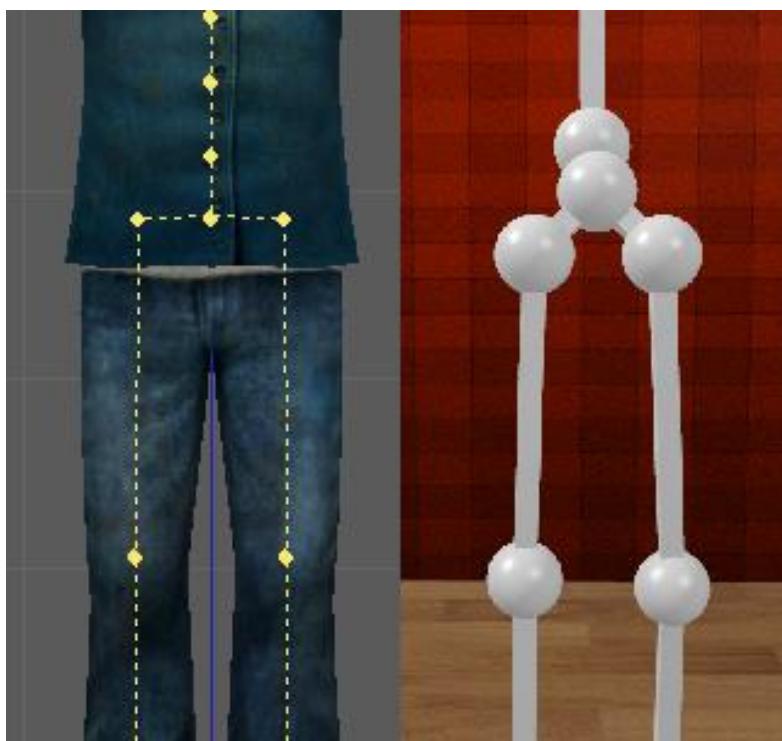


Abbildung 42: Unterschiedliche Bone Hierarchy bei Kinect und Mesh Skelett

Die Klasse [VRSkeletonKinectBinding](#) kümmert sich um das Mapping von Kinect Bones zu Mesh-Skelett-Bones. Das Mapping setzt die Rotationen der Knochen im relativen Raum, ein Knochen mit einer Null-Rotation hat also die gleiche Ausrichtung wie sein Eltern-Knochen. Die Klasse führt eine Map in der jeder gebundene Kinect Knochen auf einen Knochen im Mesh-Skelett zeigt. Für jedes Binding besteht die Möglichkeit eine Offset und eine Spaceconversion Matrix ([KinectToSkeletonSpace](#)) zu setzen. Die Offset-Matrix korrigiert Unterschiede, wie sie im oberen Bild ersichtlich sind. Beim Beispiel der Beine wird eine Korrektur von ungefähr 45° angewendet. Die [KinectToSkeletonSpace](#) Matrix kümmert sich um anders liegende Knochen. Die Knochen im Kinect Skelett liegen auf der Y-Achse, ein geladenes Modell garantiert jedoch nicht, dass die Knochen ebenfalls auf der Y-Achse liegen. Jeder individuelle

Knochen kann eine beliebige Ausrichtung besitzen, und braucht somit eine für ihn abgestimmte Matrix.

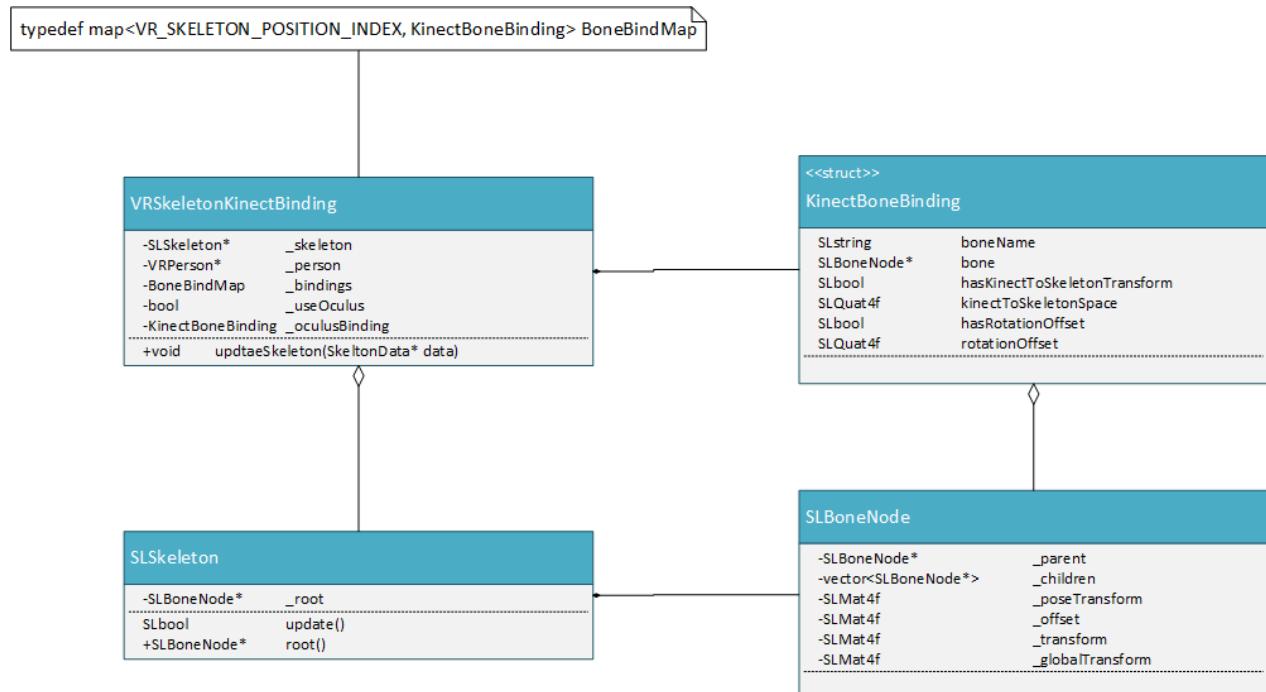


Abbildung 43: VRSkeletonKinectBinding

Dieses Binding vom Kinect- zum Mesh-Model ist gemäss der obigen Struktur erfolgreich für die erste Version der Kinect implementiert. Leider reichte die Zeit nicht, um ebenfalls eine Implementation für die zweite Entwickler Version der Kinect durchzuführen. Da die Basis-Hierarchie für die neue Version leicht abgeändert wurde (unterschiedliche Rotationen), konnten die Daten nicht analog der Kinect v1 übernommen werden.

4.5.5 Verbesserungen / andere Ansätze

4.5.5.1 Kinect Binding in World Space

Anstatt relative Rotationen der Kinect Bones auf die Mesh Bones zu binden, könnte man absolute Rotationen benutzen. Dieser Ansatz wurde getestet und verworfen, aber es besteht immer noch die Möglichkeit, dass es eine andere Lösung gibt. Der Vorteil wäre, dass Unterschiedliche relative Rotationen von Kinect Skelett zu Mesh Skelett keine Rolle mehr spielen, da alles im World Space stattfindet. Dadurch würde die Möglichkeit bestehen, dasselbe Binding für Kinect v1 und v2 zu verwenden. Aus zeitlichen Gründen konnte diese Methode nur oberflächlich mit mässigen Ergebnissen getestet werden.

4.5.5.2 Kinect X-Rotation Ungenauigkeiten

Sobald das erste rigged Mesh mit Kinect-Daten lief, fiel ein grosser Ungenauigkeitsbereich der Kinect auf: Die X-Achsen-Rotation (die Arme liegen auf dieser, wenn man in einer T-Pose steht) weist ein Flackern auf. Dieses Flackern fiel uns bei der Zylinderfigur natürlich nicht auf. Es fehlte leider die Zeit um dieses Flackern zu beheben.

4.5.5.3 Debugging mit Keyframe Animationen

Während der Entwicklung mussten wir sehr häufig Tests mit der Kinect durchführen. Ein grosses Problem bei komplexen Input-Geräten wie der Kinect ist die Inkonsistenz bei verschiedenen Durchführungen. Wenn man ein binäres Input-Gerät wie eine Tastatur testet, kann der Input leicht simuliert werden. Die Daten der Kinect können nicht so einfach reproduziert werden. Für die meisten Testzwecke funktionierte dieser Ansatz zwar, war aber trotzdem manchmal mühsam und zeitaufwendig.

Ein besserer Ansatz wäre gewesen, dennoch Keyframe-Animationen in das Projekt einzubauen und Kinect-Testdaten als Keyframes aufzunehmen. Somit hätte man mit konsistenten Daten arbeiten können und hätte nicht einmal eine Kinect dafür anschliessen müssen.

4.5.6 Verwendete Modelle

Die finalen Modelle welche im Virtual Room verwendet werden stammen von dem 2004 herausgebrachten Spiel „Half-Life 2“. Es wurde eine weibliche und eine männliche Version eines generischen „Citizen“-Modell verwendet. Auf der Suche nach guten Modellen für unsere Zwecke untersuchten wir einige Gratis-Modelle, aber fanden keine generischen Menschen, welche über ein gutes Skelett verfügten. Wir entschieden uns deshalb für die Verwendung von diesen professionell erstellten Modellen.

4.5.7 Verwendung von Blender

Für das Bearbeiten der 3D-Modelle wurde die open source Modelling-Software Blender [9] verwendet. Die Einarbeitung in dieses Programm nahm mehr Zeit in Anspruch als vorerst erwartet. Was beim Anschauen der Zahlreichen verschiedenen rigged Meshes auffiel, ist dass Bone-Orientierung und Aufbau des Skelettes im Allgemeinen stark variieren kann. Diese Orientierungen werden nach Präferenz des Modellers gewählt. Verschiedene Modelling-Programme können zudem auch andere Resultate produzieren.

5. Tests & Messungen

Das Testen der Implementationen ist ein wichtiger Teil der Arbeit. Über das gesamte Projekt ergaben sich drei Haupttestgebiete. Zum einen brauchte eine minimale Szene, um einfache Grafik Fehler zu finden und zu beheben. Zusätzlich wurde ein Projekt erstellt, welches die Kinect Daten direkt als Skelett visualisiert. Der dritte wichtige Bereich ist der Netzwerkteil. Damit dieser möglichst einfach getestet werden kann, wurde ein minimaler Test Client implementiert.

5.1 Minimale Szene

Die minimale Szene (ch14_MinimalScene) befindet sich in der Visual Studio Solution. Dieses Projekt enthält die minimalen Implementationen, um das SLProject als Framework zu verwenden. Diese Szene dient als Testumgebung bei Anpassungen am SLProject. Es kann einfach ermittelt werden, ob die Grundfunktionalitäten wie gewünscht ausgeführt werden.

Der zweite grosse Einsatzbereich war das Beheben von aufgetauchten Grafikfehlern. Im Verlauf unserer Projektarbeit tauchten verschiedenste solche Fehler auf. Diese minimale Testumgebung ist ideal zum Reproduzieren der Fehler.

5.1.1 Beispiel: „weisser Punkt“

In unserem Projekt tauchte plötzlich ein weisser Punkt auf.

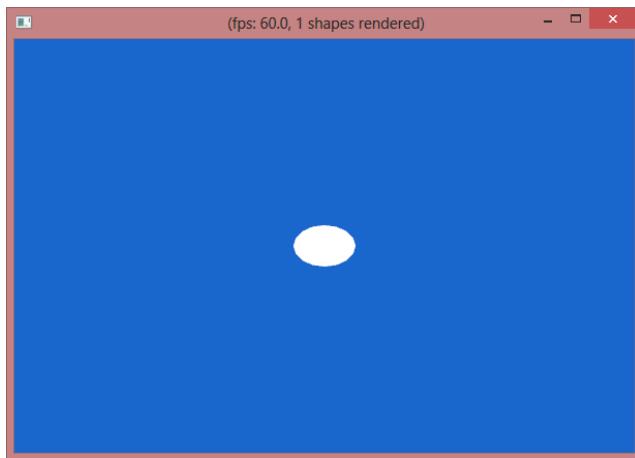


Abbildung 44: Weisser Punkt Fehler

Dieser weisse Punkt erschien, sobald die Kamera von den Szenenobjekten weggeschwenkt wurde. Zuerst kam der Gedanke, dass in unserer komplexer Szene irgendwo ein Fehler geschieht. Als dieser Fehler jedoch auch in der minimalen Szene auftauchte, musste es sich um einen Grundsatzfehler des SLProjects handeln. Diverse Tests mit verschiedenen Szenen waren notwendig. Szenen mit einer Lichtquelle, ohne Lichtquelle, mit nur einer Box oder mit einer Kugel.

Schlussendlich konnte das Problem eingeschränkt werden. Der Fehler lag bei der Verwendung von Materialien. Falls beim Rendern immer nur das gleiche Material verwendet wurde, und die Kamera keine Szenenobjekte im Bild hatte, kam es zu diesem Fehler.

5.2 Kinect Daten

Wie bereits im Kapitel 4.2 Skelett Visualisierung beschrieben, gibt es bis zur Darstellung von einem Skelett in der View einige Schritte. Damit die Auswirkungen von einzelnen Änderungen getestet werden können, wurde das Projekt ch14_SkeletonTest erstellt. Dieses Projekt beinhaltet, ähnlich wie

die minimale Szene, die erforderliche Implementation für die Verwendung des SLProjects als Framework. Zusätzlich ist der gesamte Teil zum Darstellen der Kinect Daten enthalten. In einem ersten Schritt mussten viele Tests durchgeführt werden, damit die Kinect Daten korrekt in der View dargestellt wurden. Nur die Gelenke als Kugel zu rendern war kein Problem. Dazu mussten lediglich die Positionen verwendet werden. Um jedoch die Knochen dazu zu rendern, galt es zuerst herauszufinden, wie die gelieferten Rotationsdaten der Kinect zu interpretieren sind. Anschliessend mussten diese Rotationen korrekt auf die Boxen, welche die Knochen darstellen, angewendet werden. Dieser Schritt zog einige komplizierte Berechnungen mit sich. Das Hauptproblem lag vor allem an unterschiedlichen Definitionen der Rotationsachsen zwischen der Kinect und dem SLProject.

Zum Testen der verschiedensten Funktionalitäten in der Klasse `VRPerson` eignet sich dieses Projekt auch sehr gut. In der `VRPerson` werden die Kinect Daten aufgearbeitet. Die Auswirkung dieser Aufarbeitung kann kaum gemessen werden. Man muss die Applikation laufen lassen und schauen ob sich das visualisierte Skelett mit den gemachten Änderungen anfühlt.

5.3 Netzwerk

Der Netzwerkteil ist ein wichtiger Bestandteil dieser Arbeit. Bei der Entwicklung der Netzwerkarchitektur wurde der in Punkt 4.3.4 beschriebene Netzwerk Test-Client verwendet. Dort konnten die Grundfunktionalitäten und die verschiedenen Netzwerkpakete getestet werden. Viel wichtiger war das Testen der fertigen Applikation. Welchen Einfluss hat die Netzwerkverbindung. Die meisten Tests führten wir in einem lokalen Netzwerk aus, oder die beiden Applikationen Server und Client liefen gar auf dem gleichen Computer. In dieser Umgebung gab es nie Probleme. Deshalb haben wir das System unter verschiedenen Bedingungen getestet.

1. Server und Client laufen auf dem gleichen Computer.
2. Server und Client befinden sich im gleichen Netzwerk.
3. Server und Client befinden sich in zwei verschiedenen Netzwerken in der Schweiz. Die Verbindung wird über das Internet hergestellt.
4. Der Client verbindet sich via Mobile-Netzwerk über das Internet auf den Server. Dies soll eine möglichst schlechte Verbindung simulieren.

5.3.1 Antwortzeit vom Server

Wir haben zwei wichtige Messwerte definiert, die uns interessieren. Der erste Messwert ist die Antwortzeit vom Server. Das heißt, wie lange dauert es, bis die vom Client gesendeten Daten auf dem Server simuliert werden und zurückkehren.

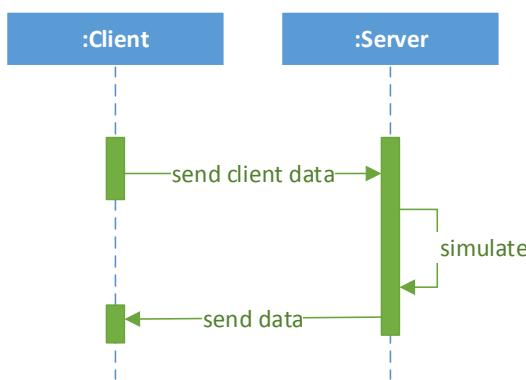


Abbildung 45: Definition Server Antwortzeit

Dieser Messwert ist wichtig für die Interaktion. Der Spieler sendet immer seine Positionsdaten an den Server. Auf dem Server werden die Berechnungen anhand der aktiven Applikation durchgeführt. Manipuliert der Spieler aufgrund seiner Position das Geschehen, sendet der Server diese Veränderungen an die Clients. Berührt nun ein Spieler zum Beispiel eine Box, ändert dieses ihre Farbe. Diese Berührung wird auf dem Server berechnet. Dieser schickt anschliessend die Information zum Client, dass die Box die Farbe ändern soll. Ist nun die Antwortzeit vom Server zu gross, gibt es zwischen der Berührung und der Verfärbung eine ungewollte Verzögerung.

5.3.2 Erhalten der Datenpakete

Der Server sendet mit einer konstanten Tickrate Datenpakete an die Clients. Wir haben diese Tick Rate auf 60 Pakete pro Sekunde eingestellt. Somit sollte der Client bei einer maximalen Zeichnungsrate von 60 Bildern pro Sekunde, für jedes Zeichnen ein Datenpaket vom Server erhalten. Bei den Netzwerktests wollen wir überprüfen, ob der Client wirklich konstant Datenpakete empfängt.

5.3.3 Ergebnisse

Wir haben für alle vier Netzwerk Konfigurationen einen Test für die Antwortzeiten und den Erhalt der Datenpakete laufen lassen. Dieser Test wurde bei einer Zeichnungsrate von ca. 60 Bildern pro Sekunde über eine Zeitspanne von insgesamt 1000 Zeichnungen durchgeführt. Der Test kann zu jeder Zeit im ch14_VirtualRoom mit der Taste „Z“ gestartet werden. Die Rohdaten werden anschliessend in eine csv Datei geschrieben.

In der folgenden Tabelle sind die Ergebnisse zur Antwortzeit des Servers:

Netzwerk Konfiguration	AVG	MIN	MAX
Gleicher Computer	0.029 s	0.015 s	0.037 s
Gleiches Netzwerk	0.032 s	0.015 s	0.066 s
Via Internet	0.048 s	0.032 s	0.121 s
Via Mobile - Internet	0.118 s	0.081 s	0.186 s

Tabelle 5: Antwortzeiten vom Server

Die Tests zur Ermittlung der Antwortzeiten haben unser Gefühl bestätigt. Bei den Tests hatten wir nie den Eindruck, dass auf unsere Eingaben verzögerte Server Antworten folgten. Selbst im schlechtesten Fall über das Mobile-Netzwerk liegt die Antwortzeit unter einem Fünftel einer Sekunde.

Bei der Datenrate entsprachen die Ergebnisse ebenfalls den Erwartungen. Der Client hat bei jeder Netzwerkbedingung konstant Datenpakete erhalten. Es gab einzelne Zeichendurchgänge, wo keine Datenpakete empfangen wurden, jedoch nie zwei hintereinander.

6. Schlussfolgerung / Fazit

Wir haben uns für diese Bachelor Arbeit viel vorgenommen. Die von uns gesetzten Ziele waren ambitioniert. Zusätzlich war die Gruppengröße von drei Personen eine Herausforderung. Es musste eine gute Mischung aus Arbeitsaufteilung und Zusammenarbeit stattfinden, damit wir das Ziel erreichen konnten. Diese Mischung haben wir sehr gut gefunden. Wir hatten von Anfang an eine gute Arbeitsaufteilung und kamen schnell voran. Es dauerte nicht lange, bis der erste Prototyp mit dem visualisierten Skelett fertiggestellt war. Auch die Mehrspieler Fähigkeit über das Netzwerk war kurz um integriert.

Die Grundfunktionalitäten konnten wir somit erstaunlich schnell umsetzen. Jedoch brauchten wir mehr Zeit als erwartet für die Überarbeitung der Implementationen. Wir wollten die Klassen- und Datenstrukturen generisch halten, damit eine Weiterentwicklung möglichst einfach ist. Eine solche Architektur zu entwerfen ist mit einem gewissen Aufwand verbunden. Dennoch konnten wir praktisch alle Arbeiten im von uns definierten Zeitrahmen fertigstellen. Einzig für das Importieren und Animieren von einem Mesh brauchten wir deutlich mehr Zeit als geplant. Der Grund dafür war, dass wir den Aufwand deutlich unterschätzt hatten. Vor allem das Animieren durch die Kinect Daten stellte sich als komplizierter heraus als angenommen. Doch dank der eingeplanten Reserve reichte es für die Umsetzung.

Während der gesamten Projektarbeit stiessen wir auf erstaunlich wenige Stolpersteine. Die Anwendung konnte kontinuierlich und schnell ausgebaut werden. Dies verdankten wir vor allem unserer guten Zusammenarbeit innerhalb der Gruppe.

6.1 Interaktion / Immersion

Die Menschheit war schon immer bestrebt die Realität abzubilden. Höhlenmalerei ist die älteste nachgewiesene Form von solchen Abbildungen. Später wurde versucht, durch perspektivische Zeichnungen die Realität möglichst aus der Betrachter Sicht darzustellen. Bereits in den 1980er Jahren sprach Jaron Lanier von virtueller Realität. Er war an der Entwicklung von ersten VR-Handschuhen und HMDs beteiligt. Durch den Einsatz von solchen technischen Geräten soll dabei eine möglichst starke Interaktion und Immersion entstehen.

Durch die Abdeckung von einem besonders grossen Sichtfeld und einem perfekten Stereo-Bild ist die Oculus Rift ein revolutionäres HMD. Wird die Rift nun mit der Kinect kombiniert, kann eine Person mit all ihren Bewegungen in eine virtuelle Welt übertragen werden.

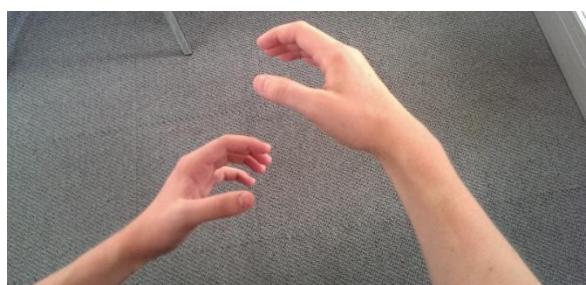


Abbildung 46: Reale Welt

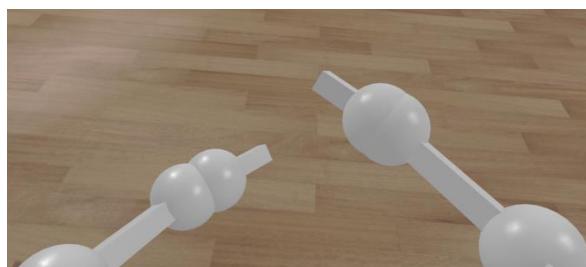


Abbildung 47: Virtuelle Welt, Skelett

Dieses Projekt hat gezeigt, dass bereits eine einfache visuelle Rekonstruktion der Personen eine grosse Wirkung hat. Dadurch, dass alle Bewegungen praktisch in Echtzeit übertragen werden, beginnen wir uns mit dem Avatar zu identifizieren.



Abbildung 48: Virtuelle Welt, Mesh

Ein Mesh verstärkt diese Identifikation zusätzlich. Die Kinect liefert keine Gelenkinformationen der einzelnen Finger, deshalb hat die Hand immer die gleiche Form. Beim Testen unserer Arbeit ist uns aufgefallen, dass wir unsere Hand immer halb offen hatten. Der Grund dafür war die starke Identifikation mit der virtuellen Abbildung von uns selber. Weil wir in der virtuellen Welt die Hand offen haben, überträgt unser Gehirn dies automatisch auf unsere reale Hand. Das heisst, wir haben durch Kombination der Kinect und der Rift eine so starke Immersion geschaffen, dass wir unser Gehirn täuschen konnten.

6.2 Vision

Diese Teleportation von einer realen Person in eine virtuelle Welt hat sicher Zukunft. Kombiniert mit dem Output Device der Rift ist die Realitätsnähe erstaunlich. Nicht ohne Grund hat Facebook die Firma Oculus VR aufgekauft. Diese Technologie kann die Zukunft bei sozialen Netzwerken werden. Ausserdem könnte eine solche Technologie für global wirkende Unternehmen interessant sein. Sitzungen via Telefon oder Skype können nicht mit den herkömmlichen Sitzungen in einem Sitzungsraum verglichen werden. Die Körpersprache spielt bei der Ausdrucksweise eines Menschen eine grosse Rolle. Doch wir sind überzeugt, dass mit dieser Virtualisierung diese Lücke geschlossen werden kann. Noch fehlen einige technische Mittel, damit die Qualität bei geringem Preis gut genug ist. Aber bereits in wenigen Jahren könnte man so weit sein.

Bezogen auf unsere Projektarbeit haben wir viele Ideen, wie diese weiterentwickelt, und verbessert werden kann. Gerade die Animation von unserem Körper mit dem Skelett und Mesh kann verbessert werden. Dazu gibt es verschiedene Lösungsansätze. Am Einfachsten wäre das Hinzufügen von zusätzlichen Filtern in der momentanen Struktur. Damit kann erreicht werden, dass es weniger Zuckungen gibt, und die Positionen genauer dargestellt werden. Eine weitere Möglichkeit wäre komplett auf die berechneten Positionsdaten aus der Kinect zu verzichten. Durch den Einsatz von mehreren Kameras könnte direkt aus den Tiefenbildern ein pro Person eigenes 3D-Modell erzeugt werden. Somit wäre auch das Problem der Animation der Meshes nicht mehr vorhanden.

7. Glossar

HMD	Bei einem HMD (Head Mounted Display) handelt es sich um ein visuelles Ausgabegerät, welches auf dem Kopf getragen wird.
Immersion	Unter Immersion versteht man ein für das Gehirn glaubhaftes Eintauchen in eine virtuelle Welt.
Szene	In der 3D-Computergrafik ist eine Szene ein virtuelles räumliches Modell. Sie beschreibt alle Objekte und deren Eigenschaften sowie die Platzierung eines Betrachters.
Szenengraph	Der Szenengraph ist die hierarchische Gruppierung der Objekte, welche sich in einer Szene befinden.
Rendern	Beim rendern wird ein Bild der Szene erzeugt. Dieses Bild repräsentiert alle Szenen-Objekte aus der Sicht des Betrachters.
World-Matrix	Mit einer World-Matrix wird die Position und Lage der einzelnen Objekte im Raum beschrieben. Jedes Objekt besitzt eine individuelle Matrix.
View-Matrix	Durch die View-Matrix wird die Position und Ausrichtung des Betrachters beschrieben.
Quaternion	Ein Quaternion ist eine komplexe Zahl, die eine Rotation im dreidimensionalen Raum beschreiben kann.
Vertex	Ein Vertex ist die Ecke von einem Polygon. In der Computergrafik werden den Vertices neben der Position (3D-Vector) auch noch weitere Eigenschaften wie Farbe und Transparenz zugeordnet.
World-State	Der World-State beschreibt den Zustand der Scene. Der World-State kann zwischen verschiedenen Benutzern ausgetauscht werden, damit alle eine einheitliche Szene haben.
Mesh	Ein Mesh ist ein Polygongitter, das die Darstellung von 3D-Objekten beschreibt.
Punktwolke	Punktwolken bestehen aus einer Ansammlung von dreidimensionalen Positionsdaten. Durch eine Vielzahl von solchen Positionen kann ein Objekt beschrieben werden.
Textur	Texturen sind meistens zweidimensionale Bilder, die auf die Oberfläche von einem virtuellen Objekt übertragen werden. Damit soll eine Erhöhung der Details erreicht werden, ohne die Komplexität der Geometrie zu verändern.
Shader	In einem Shader befindet sich Programmcode, der zur Erzeugung von 3D-Effekten dient. Dieser Code wird auf der Grafikkarte ausgeführt.
Keyframe-Animation	Eine Keyframe-Animation besteht aus mindestens zwei Keyframes. Ein Keyframe beschreibt den Zustand von einem Objekt. Die Animation erfolgt durch das Interpolieren zwischen den einzelnen Keyframes.

8. Abbildungsverzeichnis

Abbildung 1: Projektplan	7
Abbildung 2: Arbeitsteilung	8
Abbildung 3: Projekt Zyklus	8
Abbildung 4: Systemübersicht	9
Abbildung 5: Prozess auf dem Client	10
Abbildung 6: Sequenzieller Ablauf auf dem Client	10
Abbildung 7: Kinect for Windows SDK: Skeletal Tracking [1]	11
Abbildung 8: Prozess auf dem Server	13
Abbildung 9: Sequenzieller Ablauf - Client Verwaltung	14
Abbildung 10: Ordnerstruktur	17
Abbildung 11: SLProject Anpassung	19
Abbildung 12: Kamera - Klasse	21
Abbildung 13: Klassendiagramm, Skelett Visualisierung	22
Abbildung 14: links Kinect 1, rechts Kinect 2, Quelle: Kinect for Windows SDK: Skeletal Tracking [1]	23
Abbildung 15: Default Skelett	25
Abbildung 16: Skelett Daten aktualisieren	26
Abbildung 17: Projekt-Abhängigkeiten	28
Abbildung 18: Common - Netzwerk Klassen	29
Abbildung 19: Common - BoardGame Konstanten	30
Abbildung 20: Common - Enumerations	30
Abbildung 21: Common – Structs	31
Abbildung 22: Server - Klassen	31
Abbildung 23: NetworkClient Klasse	34
Abbildung 24: SLSce::onLoad - Leerer Raum	34
Abbildung 25: VRSceneView Klasse	35
Abbildung 26: Ablauf VRSceneView::preDraw	36
Abbildung 27: Klassendiagramm, GameScene	37
Abbildung 28: Color Sphere Game	41
Abbildung 29: Board Game Pattern – Reset-Boxes und Sieger Text	41
Abbildung 30: Klassendiagramm – BoardGame	43
Abbildung 31: Tic Tac Toe Spiel - Links: Am Laufen, Feld selektiert - Recht: Gewonnen	47
Abbildung 32: 3D Vier-Gewinn - Simulation der herunterfallenden Kugel	49
Abbildung 33: 3D Vier-Gewinnt - Links: Am Laufen, Zylinder selektiert - Recht: Gewonnen	50
Abbildung 34: Turn Table mit dem „Teapot“ als Mesh	51
Abbildung 35: Punktewolke Bild der Kinect, Quelle: Kinect 2 Tech Demo	51
Abbildung 36: Rigged Mesh mit sichtbarem Skelett in der Modelling-Software Blender	52
Abbildung 37: Bone Gewichte visualisiert, rot = 1, blau = 0	53
Abbildung 38: Assimp Import Library Daten Strukturen für das Laden von Szenen	54
Abbildung 39: SLImporter, Wrapper-Klasse um Assimp.	55
Abbildung 40: SLBoneNode Klassen-Member	55
Abbildung 41: Absolute Bone Transformationen im Mesh-Raum, die Inverse ist die Offset-Matrix	56
Abbildung 42: Unterschiedliche Bone Hierarchie bei Kinect und Mesh Skelett	57

Abbildung 43: VRSkeletonKinectBinding	58
Abbildung 44: Weisser Punkt Fehler	60
Abbildung 45: Definition Server Antwortzeit	61
Abbildung 46: Reale Welt	63
Abbildung 47: Virtuelle Welt, Skelett	63
Abbildung 48: Virtuelle Welt, Mesh	64

9. Tabellenverzeichnis

Tabelle 1: Virtual Room Client - Kommando-Zeile Argumente	35
Tabelle 2: Tastatur-Events - Kamera Wechsel für den Observer	39
Tabelle 3: Eigenes Game - Zu implementierende Funktionen	40
Tabelle 4: Eigenes Brettspiel - Zu implementierende Funktionen	46
Tabelle 5: Antwortzeiten vom Server	62

10. Literaturverzeichnis

[1] Kinect for Windows SDK – Skeletal Tracking

Url: <http://msdn.microsoft.com/en-us/library/hh973074.aspx> [Aufgerufen im Februar 2014]

[2] Oculus Rift SDK

Url: <http://developer.oculusvr.com> [Aufgerufen im Februar 2014]

[3] SLProject

Url: <http://code.google.com/p/slproject/> [Aufgerufen im Februar 2014]

[4] RakNet

Url: <http://www.jenkinssoftware.com/> [Aufgerufen im Februar 2014]

[5] TortoiseSVN

Url: <http://tortoisevn.net/> [Aufgerufen im Februar 2014]

[6] AnkhSVN

Url: <http://ankhsvn.open.collab.net/> [Aufgerufen im Februar 2014]

[7] Assimp

Url: <http://assimp.sourceforge.net/> [Aufgerufen im März 2014]

[8] Real-time 3D Character Animation with Visual C++

Never, Nik, Focal Press, 2002, ISBN-10: 0240516648

[9] Blender

Url: <http://www.blender.org/> [Aufgerufen im März 2014]

[10] Wikipedia Head-Mounted Display

Url: http://de.wikipedia.org/wiki/Head-Mounted_Display [Aufgerufen im April 2014]

[11] Doxygen

Url: <http://www.stack.nl/~dimitri/doxygen/> [Aufgerufen im Februar 2014]

11. Anhang

Der Anhang zur Bachelor Arbeit ‘Virtual Room’ besteht aus den nachfolgend aufgelisteten Dokumenten. Der Source-Code der Anwendung inkl. der zu installierenden MSIs für die Treiber sind auf der beigelegten CD enthalten.

- I. Selbständigkeitserklärungen
- II. Dokumentation der Projektarbeit 2 „Oculus Rift“

Anhang I



Erklärung der Diplandinnen und Diplanden Déclaration des diplômé-e-s

Selbständige Arbeit / Travail autonome

Ich bestätige mit meiner Unterschrift, dass ich meine Bachelor Thesis selbständig durchgeführt habe. Alle Informationsquellen (Fachliteratur, Besprechungen mit Fachleuten, usw.), die wesentlich zu meiner Arbeit beigetragen haben, sind in meinem Arbeitsbericht im Anhang vollständig aufgeführt.

Par ma signature, je confirme avoir effectué ma thèse de Bachelor de manière autonome. Toutes les sources d'information (littérature spécialisée, discussions avec spécialistes etc.), qui m'ont fortement aidées dans mon travail, sont intégralement mentionnées dans l'annexe de ma thèse.

Name/Nom, Vorname/Prénom

Renggli, Cédric

Datum/Date

12.06.14

Unterschrift/Signature

C. Renggli

Dieses Formular ist dem Bachelor Thesis Bericht beizulegen.

Ce formulaire doit être joint au rapport de la thèse de Bachelor.



Erklärung der Diplandinnen und Diplanden Déclaration des diplômé-e-s

Selbständige Arbeit / Travail autonome

Ich bestätige mit meiner Unterschrift, dass ich meine Bachelor Thesis selbständig durchgeführt habe. Alle Informationsquellen (Fachliteratur, Besprechungen mit Fachleuten, usw.), die wesentlich zu meiner Arbeit beigetragen haben, sind in meinem Arbeitsbericht im Anhang vollständig aufgeführt.

Par ma signature, je confirme avoir effectué ma thèse de Bachelor de manière autonome. Toutes les sources d'information (littérature spécialisée, discussions avec spécialistes etc.), qui m'ont fortement aidées dans mon travail, sont intégralement mentionnées dans l'annexe de ma thèse.

Name/Nom, Vorname/Prénom

Wacker, Marc

Datum/Date

12.06.14

Unterschrift/Signature

M. Wacker

Dieses Formular ist dem Bachelor Thesis Bericht beizulegen.

Ce formulaire doit être joint au rapport de la thèse de Bachelor.



Erklärung der Diplandinnen und Diplanden

Déclaration des diplômé-e-s

Selbständige Arbeit / Travail autonome

Ich bestätige mit meiner Unterschrift, dass ich meine Bachelor Thesis selbständig durchgeführt habe. Alle Informationsquellen (Fachliteratur, Besprechungen mit Fachleuten, usw.), die wesentlich zu meiner Arbeit beigetragen haben, sind in meinem Arbeitsbericht im Anhang vollständig aufgeführt.

Par ma signature, je confirme avoir effectué ma thèse de Bachelor de manière autonome. Toutes les sources d'information (littérature spécialisée, discussions avec spécialistes etc.), qui m'ont fortement aidées dans mon travail, sont intégralement mentionnées dans l'annexe de ma thèse.

Name/Nom, Vorname/Prénom

Kühne, Roman

Datum/Date

12.06.14

Unterschrift/Signature

R. Kühne

Dieses Formular ist dem Bachelor Thesis Bericht beizulegen.

Ce formulaire doit être joint au rapport de la thèse de Bachelor.

Anhang II

Projekt 2 Arbeit 2013

Oculus Rift



Fachbereich	Informatik
Modul	Projekt 2
Studierende	Roman Kühne
Betreuer	Marc Wacker Marcus Hudritsch

Inhaltsverzeichnis

1	Ziel des Projektes	3
2	Oculus Rift	4
2.1	Technische Grundlagen	4
2.1.1	Aufbau	4
2.1.2	LibOVR	4
2.2	Rendering	5
2.2.1	Stereo	5
2.2.2	Korrektur Verkrümmung	8
2.2.3	Skalierung	9
2.3	Implementation	10
2.3.1	SLOculus	10
2.3.2	Setzten der Matrizen	11
2.3.3	Verkrümmung Shader	12
3	Projektarbeit	13
3.1	Minimalanwendung	13
3.1.1	Ziel	13
3.1.2	Übersicht und Features	13
3.1.3	Tastenkürzel	14
3.1.4	Implementation	14
3.1.4.1	Rift Kontroller	14
3.1.4.2	Pipeline	14
3.1.4.3	Linsenkrümmung als Postprozess	14
3.1.4.4	Anwendung der Zwischentextur	15
3.2	Generisches Post-Processing	15
3.2.1	Warum	15
3.2.2	Erster Prototyp	15
3.2.2.1	Klassen	15
3.2.2.1.1	PostEffect	15
3.2.2.1.2	PostProcManager	16
3.2.2.1.3	AttachmentPool	16
3.2.2.2	Probleme	16
3.2.2.2.1	Nicht generisch für mehrere Viewports	16
3.2.2.2.2	Szene und Gui getrennt	16
3.2.3	Per Viewport Post Processing	16
3.2.3.1	Unterschiede zur ersten Version	17
3.2.3.2	Probleme	17
3.2.3.3	Architektur	18
3.2.3.4	FBOs vs Attachments	18
3.3	Integration SLProject	19
4	Fazit	20

1 Ziel des Projektes

Das erfolgreiche Kickstarter Projekt Oculus Rift hat in 2012 Virtual Reality in den Mainstream gerückt. Gerade der tiefe Preis macht die Oculus für die Massen zugänglich.

Ziel dieses Projekts ist die Einarbeitung in die Oculus und OpenGL. Es soll eine minimale OpenGL Applikation geschrieben werden welche mit der Oculus funktioniert, um die Grundlagen in möglichst kleiner Form zu haben.

Des Weiteren soll das Projekt als Vorbereitung auf die Bachelor Thesis dienen, welche eine konkrete Applikation für die Rift finden soll.

Ausserdem soll die Oculus in das SLProject integriert werden. Dabei muss beachtet werden, dass die Implementation auf verschiedenen Betriebssystemen laufen muss.

2 Oculus Rift

2.1 Technische Grundlagen

In dieser Arbeit wird die erste Entwicklerversion der Oculus Rift verwendet. Der Hauptbestandteil ist ein 7 Zoll grosser LCD Bildschirm. Dieser verfügt über eine Auflösung von 1280 x 800 Pixel. Dieser Bildschirm wird mit fixierten Linsen ergänzt. Die eingebauten Sensoren bilden den zweiten wichtigen Teil der Oculus. Sie beinhaltet ein Gyroskop, Magnetometer und ein Beschleunigungssensor. Die Sensor Informationen werden mit 1000Hz ausgelesen. Diese Sensordaten werden benötigt um die Kopfrotation zu bestimmen.

2.1.1 Aufbau

Der Bildschirm und die Linsenhalterung sind in ein fixes Gehäuse eingebaut. Der Bildschirm ist in der Hälfte durch eine dünne Trennwand unterteilt. Dadurch werden die einzelnen Sichtfelder der Augen klar getrennt.

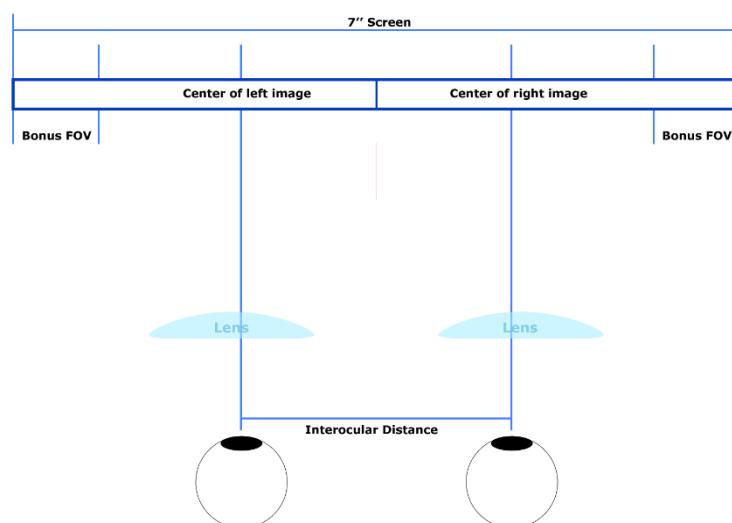


Abbildung 1: Schemaskizze

2.1.2 LibOVR

Die LibOVR ist die Oculus SDK. Diese dient als Schnittstelle zur Oculus in einem c++ Projekt. Durch diese erhalten wir alle wichtigen Informationen, welche von der Oculus benötigt werden.

Durch die Informationen aus der SDK kann das Bild für die Oculus berechnet werden. Dazu gehören die statischen Werte Bildschirmgrösse, Linsenabstand, Krümmungsfaktoren, etc. Dadurch kann eine Hardware unabhängige Implementation erstellt werden.

Ein weiterer wichtiger Bestandteil ist die „SensorFusion“. Daraus werden die Kopfrotation und die momentane Rotationsgeschwindigkeit ausgelesen. Dieses Resultat stammt von den Daten der Sensoren.

2.2 Rendering

Eine Szene für die Oculus zu rendern ist nicht ganz einfach. Es braucht einige Schritte um das korrekte Bild zu erhalten.

2.2.1 Stereo

Als erstes muss die Szene in Stereo gerendert werden, eine Hälfte des Bildes pro Auge. Beim Tragen der Brille sieht das Auge nur seine jeweilige Hälfte des Bildschirmes. Das bedeutet dass die gesamte Szene zweimal gerendert werden muss.

Im Gegensatz zum rendern von TV-Stereo braucht es für die Oculus keine asymmetrische Projektion. Die Projektionsachsen sind parallel zueinander. Damit ist dieser Vorgang dem rendern ohne Stereo ziemlich ähnlich. Es muss lediglich die Kamera für das jeweilige Auge ein wenig verschoben werden.

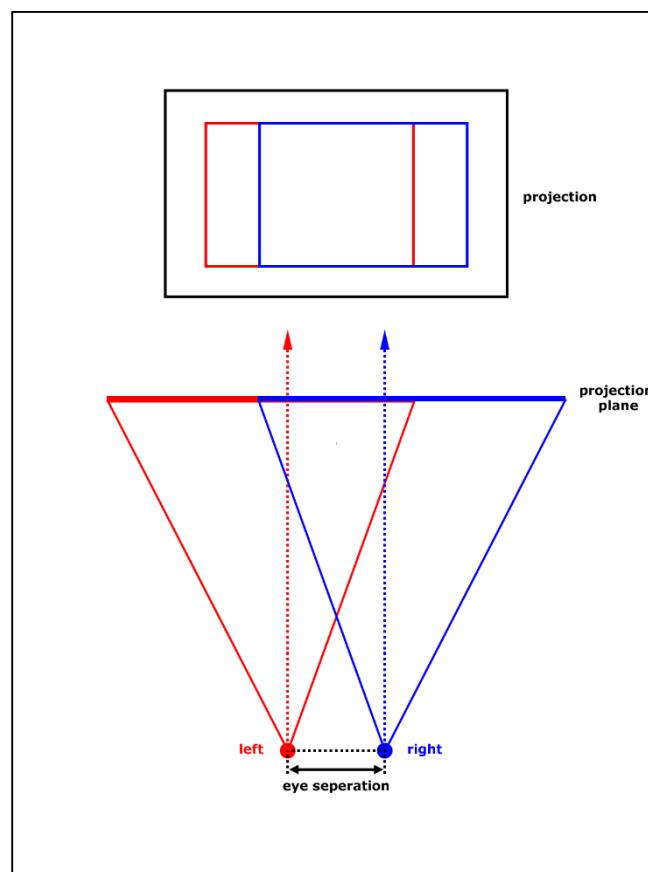


Abbildung 2: Oculus-Stereo

Um die Projektionsmatrix zu berechnen sind folgende Parameter nötig, welche aus der LibOVR gelesen werden.

Name	Beschreibung
HScreenSize, VScreenSize	Physikalische Dimensionen des Bildschirmes in Meter.
EyeToScreenDistance	Distanz vom Auge zum Bildschirm. Dieser Wert wird benötigt um den optimalen Winkel für das „fov“ zu berechnen.
LensSeparationDistance	Physikalischer Abstand zwischen den beiden Linsen Zentren.
InterpupillaryDistance	Distanz zwischen den beiden Augenmittelpunkten. Dieser Abstand ist für jede Person individuell. Er kann mit dem Oculus Konfigurationsprogramm bestimmt werden.
HResolution, VResolution	Die Auflösung des Bildschirmes.

Um die Initial Projektionsmatrix zu bestimmen muss zuerst das „fov“ berechnet werden. Den optimalen Winkel wird aus der realen Bildschirmhöhe und dem Abstand vom Bildschirm zum Auge berechnet.

$$\alpha_{fov} = 2 \arctan \left(\frac{VScreenSize}{2 \times EyeToScreenDistance} \right)$$

Zusätzlich zu diesem Winkel braucht es noch das Seitenverhältnis für eine Bildschirmhälfte. Dieses Verhältnis lässt sich einfach aus der Auflösung berechnen.

$$a = \frac{HResolution}{2 \times VResolution}$$

Durch diese zwei Werte kann nun die Initiale Projektionsmatrix bestimmt werden.

Wie in der Abbildung 1 ersichtlich ist befinden sich die Linsen nicht genau im Zentrum der jeweiligen Hälfte, sondern näher zu der Mitte. Die Projektionsmatrix muss deshalb zum Linsenzentrum, und nicht zum Bildschirmhälfte-Zentrum verschoben werden. Alle nötigen Angaben, um diese Verschiebung zu berechnen, können aus der LibOVR ausgelesen werden.

$$projectionCenterOffset = 4 \times \left(\frac{\left(\frac{HScreenSize}{4} - \frac{LensSeparationDistance}{2} \right)}{HScreenSize} \right)$$

Die Initiale View Matrix beinhaltet die Position der Kamera und die Rotation der Oculus. Die Verschiebung für das jeweilige Auge ist durch den Augenabstand gegeben.

$$viewCenterOffset = \frac{InterpupillaryDistance}{2}$$

Die gerenderte Szene mit dieser Konfiguration sieht dann folgendermassen aus:

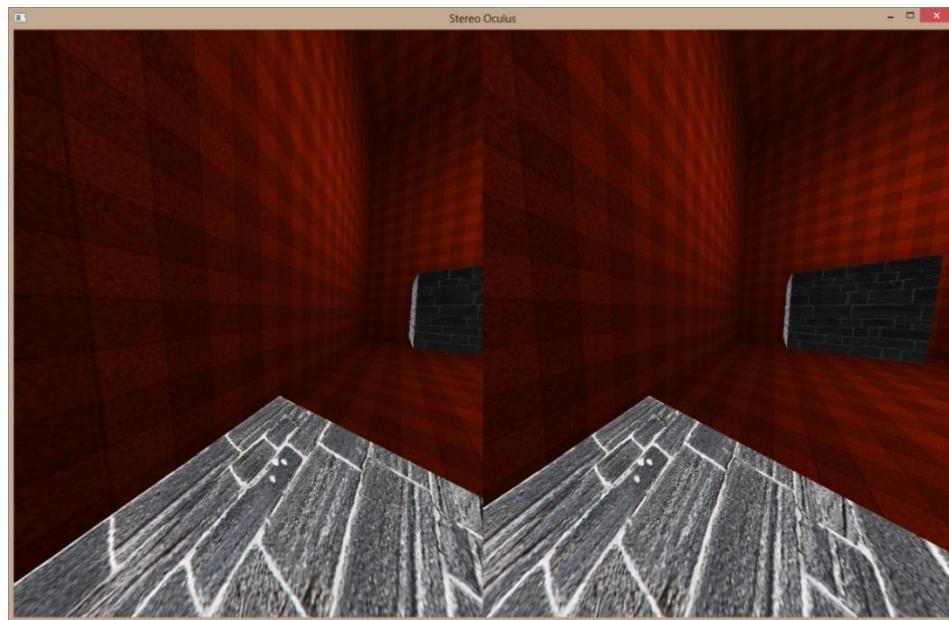


Abbildung 3: Stereo

Dabei ist das Oculus-Stereo-Rendering gut ersichtlich. Das rechte Auge sieht rechts mehr als das linke und umgekehrt. Nun fehlt aber noch ein wichtiger Teil. Wegen dem Einsatz von Linsen wird das Bild gekrümmmt. Dieser Krümmung muss entgegengewirkt werden.

Fazit:

Die Berechnungen für das Stereo-Rendering werden möglichst auf die physikalische Gegebenheit der Oculus abgestimmt. Das „fov“ entspricht dem maximalen „fov“, welches in der Oculus physikalisch Möglich ist. Die View Matrix(pro Auge) wird genau zum Augenmittelpunkt verschoben. Die Projektionsmatrix orientiert sich an der Mitte der Linsen.

2.2.2 Korrektur Verkrümmung

Durch den Einsatz der Linsen wird das „fov“ vergrössert. Dies hat eine Kissenverzerrung(Pincushion distortion) zur Folge. Um dieser Verzerrung entgegenzuwirken wird dem Bild eine tonnenförmige Verzerrung (Barrel distortion) hinzugefügt.

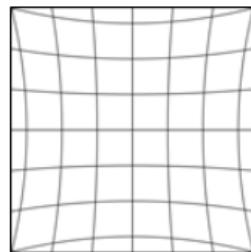


Abbildung 4: Pincushion Distortion

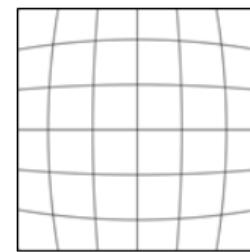


Abbildung 5: Barrel Distortion

Die Stärke der Verzerrung wird durch den Abstand zum Linsenzentrum bestimmt. Dazu wird folgende Funktion angewendet:

$$f(r) = k_0 + k_1 r^2 + k_2 r^4 + k_3 r^6$$

Die vier Faktoren k_0-k_3 bestimmen die Verzerrung. Diese Faktoren können aus der LibOVR ausgelesen werden. Wichtig ist das bestimmen des korrekten Zentrums für die Berechnung. Es muss vom Linsenzentrum ausgehende Verzerrt werden, dies entspricht aber nicht dem Zentrum der Bildhälfte.

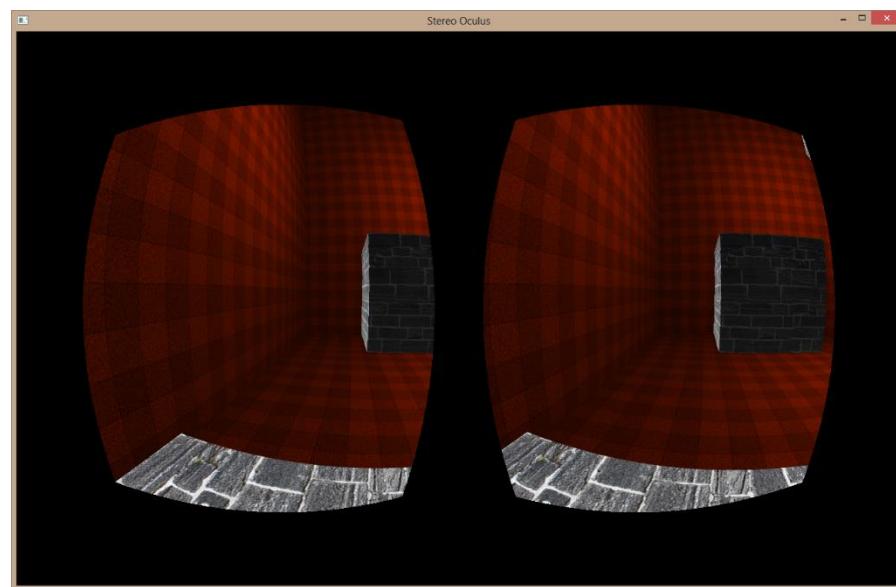


Abbildung 6: Stereo mit Verzerrung

Durch das Verzerren wird das Bild gegen die Verzerrungsmitte zusammengezogen. Somit entsteht für einen Teil des Bildes eine „Leere“. Damit trotzdem die gesamte

Bildgrösse ausgenutzt werden kann, muss die Eingangstextur grösser sein als das eigentliche Bild.

2.2.3 Skalierung

Um wie viel diese Eingangstextur sein muss ist durch die Verzerrungsfaktoren und das Seitenverhältnis gegeben.

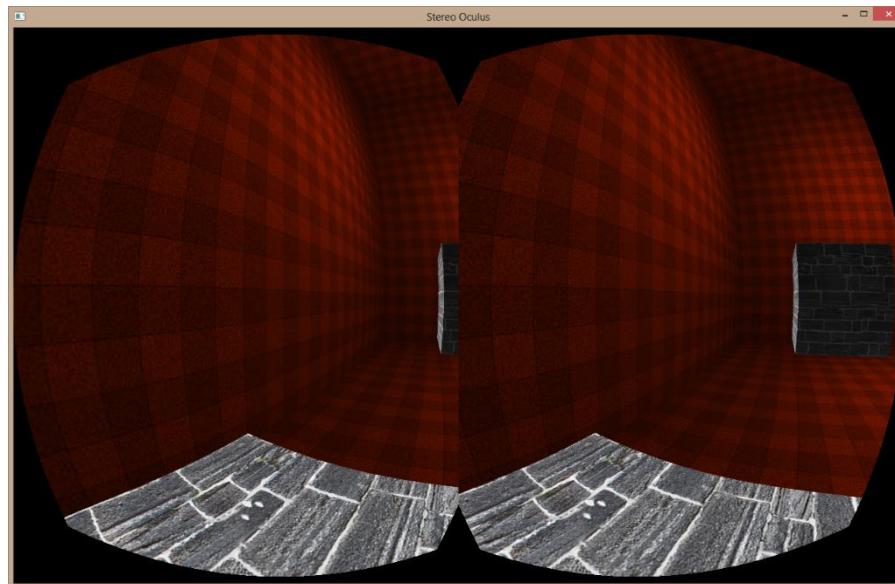


Abbildung 7: Stereo mit Verzerrung und Skalierung

Dies ist schliesslich das Bild, welches in der Oculus dargestellt wird. Gut zu sehen ist auch die Asymmetrische Verzerrung. Da das Linsenzentrum näher zur Mitte ist, nimmt die Verzerrung gegen aussen zu.

2.3 Implementation

2.3.1 SLOculus

Die Klasse SLOculus ist die Schnittstelle zur LibOVR. In dieser Klasse könne alle Parameter, welche für das rendern benötigt werden ausgelesen werden.

```
//-----
/*!
initialize the connection to the oculus. Sets defaults values if the oculus is
not connected
*/
void SLOculus::init()
{
..
}
```

Diese Funktion wird zum Start einmalig aufgerufen. Sie initialisiert die Verbindung zur Oculus.

```
//-----
/*!
returns the oculus orientation
*/
SLOquat4f SLOculus::orientation()
{
..
}
```

Beim rendern von jedem Frame wird über diese Funktion die momentan Rotation der Oculus geholt.

```
//-----
/*!
calculates the factors for stereo rendering
*/
void SLOculus::updateRenderFactors(int width, int height)
{
    float DistortionFitX = -1.0f;
    float DistortionFitY = 0.0f;

    //distortion offset
    float lensOffset = Info.LensSeparationDistance * 0.5f;
    float lensShift = Info.HScreenSize * 0.25f - lensOffset;
    lensViewportShift = 4.0f * lensShift / Info.HScreenSize;

    //projection offset
    float viewCenter = Info.HScreenSize * 0.25f;
    float eyeProjectionShift = viewCenter - Info.LensSeparationDistance * 0.5f;
    projectionCenterOffset = 4.0f * eyeProjectionShift / Info.HScreenSize;

    //Scale
    float stereoAspect = 0.5f * float(width) / float(height);
    float dx = DistortionFitX - lensViewportShift;
    float dy = DistortionFitY / stereoAspect;
```

```

float fitRadius      = sqrt(dx * dx + dy * dy);
scale               = distortionFn(fitRadius)/fitRadius;

//yfov
float percievedHalfRTDistance = (Info.VScreenSize / 2) * scale;
yFov = 2.0f * atan(percievedHalfRTDistance/Info.EyeToScreenDistance);
yFov = yFov * 180 / Math<float>::Pi;
}

```

In dieser Funktion werden alle wichtigen Parameter die für das Stereorendering benötigt werden berechnet.

2.3.2 Setzten der Matrizen

Viewmatrix

```

_riftRotation = SLOculus::orientation().toMat4();

_viewLeftMat   = _viewRightMat = _riftRotation * _viewCenterMat;
_viewLeftMat   = SLMat4f(halfIPD, 0.0f, 0.f) * _viewLeftMat;
_viewRightMat  = SLMat4f(-halfIPD, 0.0f, 0.f) * _viewRightMat;

```

Die Verschiebungen der Kamera sind in der „_viewCenterMat“ gespeichert. Zu dieser Matrix wird die Rotationsmatrix von der Oculus dazu multipliziert, und bildet damit die Grundlage für die Matrizen der Augen. Diese zentrale Matrix wird nun noch um die Hälfte des Augenabstandes für das jeweilige Auge verschoben.

Projektionsmatrix

```

_centerProjection.perspective(SLOculus::yFov, (double)_halfWidth/(double)_height,
0.01f, 100.0f);

_leftProjection = _rightProjection = _centerProjection;
_leftProjection = SLMat4f(SLOculus::projectionCenterOffset, 0.0f, 0.f) * _leftProjection;
_rightProjection=SLMat4f(-SLOculus::projectionCenterOffset,0.0f, 0.f) * _rightProjection;

```

Die Initiale Projektionsmatrix ist abhängig vom berechneten "fov" und dem Seitenverhältnis des Bildschirmes. Die Faktoren für das „Clipping“ sind fix.

Mit diesen Matrizenkonfigurationen kann nun das Stereobild für die Oculus gerendert werden. Dazu wir die Scene in eine Textur gerendert und anschliessend vom Verkrümmung Shader bearbeitet.

2.3.3 Verkrümmung Shader

In einem Fragment Shader wird die Textur verzerrt, um dem bereits beschriebenen Problem der Linsen entgegenzuwirken.

```

uniform sampler2D      u_sceneBuffer;
uniform float          u_lensCenterOffset;
uniform vec4            u_hmdWarpParam;
uniform vec2            u_scale;
uniform vec2            u_scaleIn;

varying vec2           vTexCoord;

vec2 lensCenter = vec2(0, 0.5);
vec2 screenCenter = vec2(0.25, 0.5);

// Scales input texture coordinates for distortion.
vec2 hmdWarp(vec2 in01)
{
    vec2 theta = (in01 - lensCenter) * u_scaleIn; // Scales to [-1, 1]
    float rSq = theta.x * theta.x + theta.y * theta.y;
    float rSq2 = rSq * rSq;
    vec2 rvector = theta * (u_hmdWarpParam.x + u_hmdWarpParam.y * rSq +
        u_hmdWarpParam.z * rSq2 +
        u_hmdWarpParam.w * rSq2 * rSq);
    return lensCenter + u_scale * rvector;
}

void main()
{
    if(vTexCoord.x > 0.5) {
        screenCenter.x = 0.75;
        lensCenter.x = 1.0 - u_lensCenterOffset;
    }
    else {
        lensCenter.x = 0.0 + u_lensCenterOffset;
    }

    vec2 tc = hmdWarp(vTexCoord);
    //any tests the bool vector if any parameter is true (are we outside of
the screen?)
    if(any(bvec2(clamp(tc,screenCenter-vec2(0.25,0.5),
screenCenter+vec2(0.25,0.5)) - tc)))
    {
        gl_FragColor = vec4(0, 0, 0, 1);
        return;
    }

    gl_FragColor = texture2D(u_sceneBuffer, tc);
}

```

3 Projektarbeit

3.1 Minimalanwendung

3.1.1 Ziel

Um die Funktionsweise der Rift übersichtlich und einfach demonstrieren zu können, war unser erstes Ziel eine Minimalapplikation zu schreiben, welche Headtracking erlaubt. Diese Applikation haben wir in C++ und OpenGL implementiert. Für die Mathematik relevanten Klassen verwendeten wir SLProject und für das Window-Handling GLFW 3.0.

3.1.2 Übersicht und Features

Die finale Version des Minimalprojekts stellt einen Raum mit einer Seitenlänge von 20 m dar. Der Raum ist mit 7 unterschiedlich kleineren Würfeln gefüllt, welche in den Größen von 10 cm bis zu 4 m variieren. Der Benutzer startet auf einem 2 m grossen Würfel mit einem 10 cm Würfel direkt vor ihm. Hier ist sehr schön die Größenwahrnehmung zu erleben.

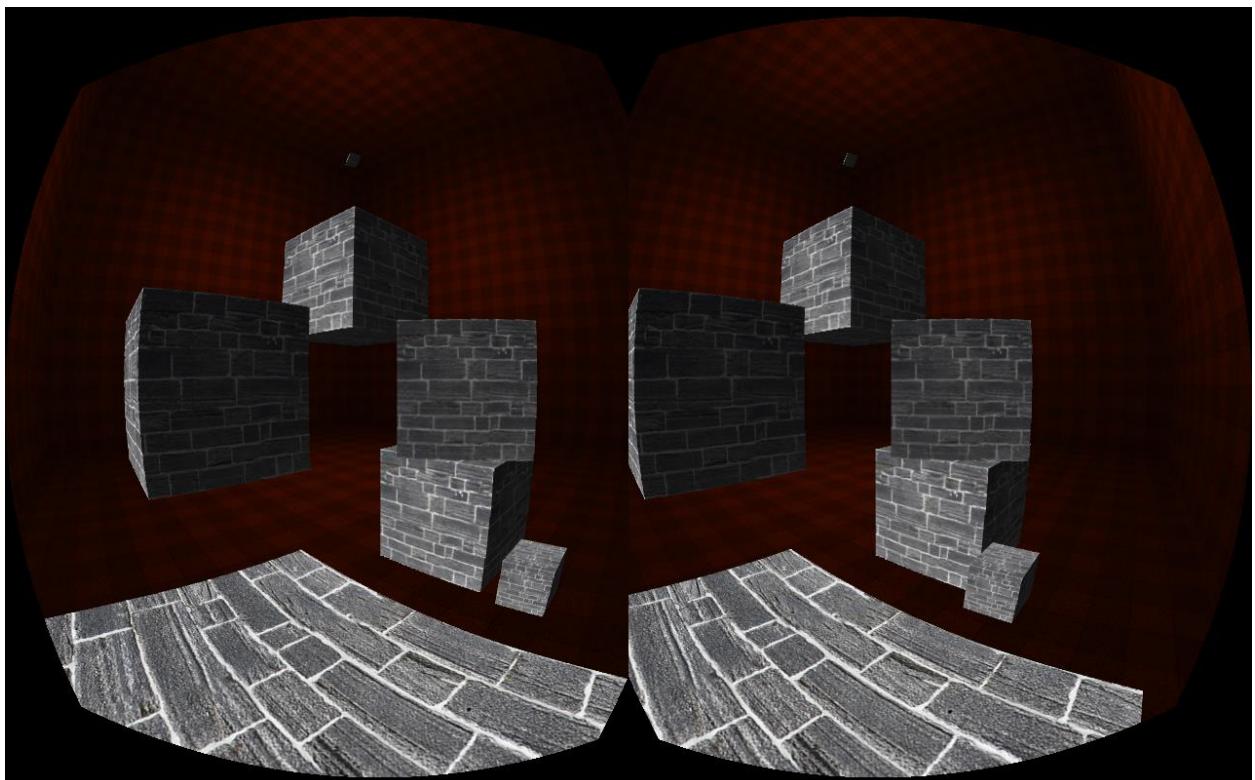


Abbildung 8: Größenwahrnehmung

3.1.3 Tastenkürzel

- W:** Vorwärts
- A:** Links
- S:** Rückwärts
- D:** Rechts
- Q:** Runter
- E:** Hoch
- I:** Info in Konsole ausgeben
- P:** Animation pausieren/starten
- H:** Projektionsmatrix Translation verkleinern
- J:** Projektionsmatrix Translation vergrössern
- K:** Augendistanz vergrössern
- L:** Augendistanz verkleinern
- R:** Ausgangseinstellungen wiederherstellen

M1: X und Y Rotation der Kamera kontrollieren

3.1.4 Implementation

3.1.4.1 Rift Kontroller

Der Rift Kontroller ist ein C++ Wrapper der die wichtigsten Funktionen der OVR SDK beinhaltet. Mit dem Kontroller können wir die Orientierung und das Field of View (FOV) der Rift auslesen.

3.1.4.2 Pipeline

Die Renderpipeline der Minimalapplikation sieht wie folgt aus:

1. View Matrix mit der momentanen Rift Orientierung aktualisieren.
2. Szene für linkes und rechtes Auge in Zwischentextur rendern.
3. Von Zwischentextur mit Verkrümmung auf den Bildschirm rendern.

Inputevents werden durch GLFW gepollt.

Warum wir eine Zwischentextur benötigen wird im nächsten Punkt erklärt.

3.1.4.3 Linsenkrümmung als Postprozess

Wegen der benötigten Linsenkrümmung welche in Kapitel 2 behandelt wurde, können wir die Scene nicht direkt in den Standard Framebuffer rendern. Der Verkrümmungseffekt ist ein lokaler Pixeloperator wie auch Unschärfe oder Kantenerkennung. Für die meisten lokalen Operatoren muss auf Pixeldaten ausserhalb der Zielpixelposition zugegriffen werden. Auf solche Daten kann erst in einem zweiten Renderschritt zugegriffen werden. Der Verkrümmungseffekt muss also als Postprozess laufen.

3.1.4.4 Anwendung der Zwischentextur

OpenGL bietet sogenannte Framebuffer Objects (FBOs). Ein FBO ist ein alternativer Framebuffer wie auch der Bildschirm einer ist. FBOs besitzen verschiedene Attachments für Texturen. Diese Attachments sind 0 bis N Farbtexturen, ein Tiefenattachment, Stencilattachment und eine Kombination aus Tiefen und Stencilattachment.

Um die Szene in unsere Zwischentextur rendern zu können, benötigen wir einen FBO mit einem Farb- und einem Tiefenattachment. Der Bildschirm besitzt bereits eine Tiefentextur, wenn man aber einen FBO mit Tiefentextur benutzen will, dann muss dieser auch ein Tiefenattachment besitzen. OpenGL bietet sogenannte Renderbuffer Objects. Ein Renderbuffer kann wie eine normale Textur ebenfalls als FBO Attachment verwendet werden. Renderbuffer sind optimierte Datenstrukturen genau für den Gebrauch mit FBOs und können einen Geschwindigkeitsvorteil gegenüber Texturen aufweisen.

Die Zwischentextur beinhaltet die gerenderten Szenen beider Augen nebeneinander. Demnach muss die Textur die gleiche oder grössere Auflösung wie die finale Bildschirmauflösung haben. Wir rendern erst die Szene für beide Augen in die Zwischentextur. In einem zweiten Schritt rendern wir ein Viereck über den ganzen Bildschirm und mappen die Zwischentextur darauf. In diesem zweiten Schritt wenden wir nun die im Kapitel 2 angesprochene Verkrümmung an.

3.2 Generisches Post-Processing

3.2.1 Warum

Das finale Rift Projekt soll SLProject als Grundlage verwenden. In der Minimalapplikation haben wir bereits gesehen, dass die Linsenkrümmung als Postprozess implementiert werden muss. Eine generische Implementation für Postprozess-Effekte wäre also eine Möglichkeit die Rift in SLProject zu unterstützen und gleichzeitig andere Effekte wie zum Beispiel Tiefunschärfe zu erlauben.

3.2.2 Erster Prototyp

3.2.2.1 Klassen

Die erste Version der Post Processing Implementation differenzierte zwischen zwei verschiedenen Effekten. Effekte welche auf die Szene angewendet werden (z.B. Tiefunschärfe) und Effekt welche auf die Szene und das GUI angewendet werden (z.B. Linsenverkrümmung für die Rift).

3.2.2.1.1 PostEffect

Beinhaltet Effekt spezifische Daten (Effekt File, Skallierung, Grösse etc.). Diese Klasse erbt von SLGLShaderProg, beim instanziieren eines Effektes muss der Name eines Fragmentshaders angegeben werden. Ein Effekt hat eine execute Funktion

welche eine angegebene Input-Textur mit dem Fragmentshader in einen angegebenen Output rendern (Bildschirm oder nächste Textur).

Effekte können aus mehreren Stages bestehen z.B. ein gaussischer Weichzeichnungsfilter kann in X und Y-Teil aufgespalten werden. Dafür braucht es einen Zwischenschritt und man muss zwischen mehreren Zwischentexturen wechseln.

3.2.2.1.2 PostProcManager

Beinhaltet zwei Listen mit PostEffect Pointern, eine Liste für GUI-Effekte und eine zweite für die Szenen-Effekte. Der Manager kennt die Auflösung, welche die Initiale Input-Textur haben muss, wenn alle individuellen Effekt-Skalierungen berücksichtigt werden.

3.2.2.1.3 AttachmentPool

Der AttachmentPool kümmert sich um das Textur-Sharing. Die Effekte benötigen Zwischentexturen zwischen denen gewechselt werden kann. Wenn mehrere Effekte in einer Liste dieselben Größen haben, kann Memory gespart werden in dem diese die gleichen Texturen verwenden.

Bei dieser ersten Implementation werden Effekte auf den ganzen Bildschirm angewendet. Wenn eine Szene stereoskopisch Side-by-Side gerendert werden muss, müssen spezielle Effekte verwendet werden. Positiv ist jedoch, dass der AttachmentPool selbst Resize-Events erhalten kann, und somit die Effekte sich nicht um das Vergrössern und Verkleinern der Zwischentexturen kümmern müssen.

3.2.2.2 Probleme

3.2.2.2.1 Nicht generisch für mehrere Viewports

Wenn wir eine Szene in Stereo oder gar in mehreren speziellen Viewports rendern wollen, kann es zu Problemen mit den Vollbildeffekten kommen. Ein Effekt wie z.B. Tiefenunschärfe würde dann die Kanten zwischen den Effekten verwischen, und die Effekt-Files bräuchten speziellen Code für diese Fälle.

3.2.2.2.2 Szene und Gui getrennt

Zwei separate Listen zu unterhalten für Szenen und Gui Effekte geht ebenfalls in eine nicht generisch Richtung.

3.2.3 Per Viewport Post Processing

Um eine wirklich generische Implementation zu bieten, haben wir eine zweite Version implementiert. Die zweite Version funktioniert im Grunde gleich wie die erste mit ein paar kleinen Unterschieden.

3.2.3.1 Unterschiede zur ersten Version

1. Das Resizen der Attachments wird nicht mehr automatisch erledigt, sondern von den Effekten selbst.
2. PostProcManager wurde entfernt und mit EffectList ersetzt.
3. Effekte werden nun auf bestimmten Viewport-Bereichen ausgeführt anstatt auf den ganzen Bildschirm.

Das bedeutet zusammengefasst, dass für jeden zusätzlichen Viewport eine Post Prozess Liste ausgeführt werden muss. Unsere Implementation instanziert eine Effekt Liste für jede Kamera. Bei einer Stereo Kamera wird die gleiche Effekt Liste zweimal ausgeführt. Dies erlaubt es uns auch ein GUI mit einer Orthogonalen Kamera über die Szene zu rendern, und somit immer noch eine Trennung der Kameras zu behalten.

3.2.3.2 Probleme

Das Trennen von GUI und Szenen Effekten funktioniert in unserer Implementation, jedoch ist es teurer als eine nicht generische Implementation. Im Falle der Rift wäre der Renderablauf wie folgt:

1. Die Scene wird für das linkes Auge gerendert.
2. Post Prozesse werden angewendet und in den Bildschirm Framebuffer gerendert.
3. Punkt 1 und 2 werden für das rechte Auge wiederholt.
4. GUI wird für das rechte Auge gerendert.
5. Post Prozesse werden angewendet und in den Bildschirm Framebuffer gerendert.
6. Punkt 4 und 5 werden für das rechte Auge wiederholt.

Der angewendete Post Prozess ist in unserem einfachen Beispiel lediglich die Linsenverkrümmung. Das ganze könnte aber wesentlich einfacher sein, wenn wir die Stereo Szene mit dem GUI zusammen rendern würden, bevor wir die finalen Effekte anwenden würden.

Wir konnten zwar zum Ende des Projektes eine fertige Version des generischen Post Processings implementieren, jedoch waren wir nicht zufrieden mit den Kompromissen welche wir machen mussten. Wir halten uns die Möglichkeit offen, das Post Processing in einer generischen Art und Weise ins SLProject einzubauen, jedoch wollen wir dies erst tun, wenn wir die grösseren Probleme lösen können. Der Projektumfang ist auch ohne generisches Post Processing gross genug und wird vorerst nicht weiter verfolgt.

3.2.3.3 Architektur

Die Momentane Architektur unseres generischen Post Processing Systems.

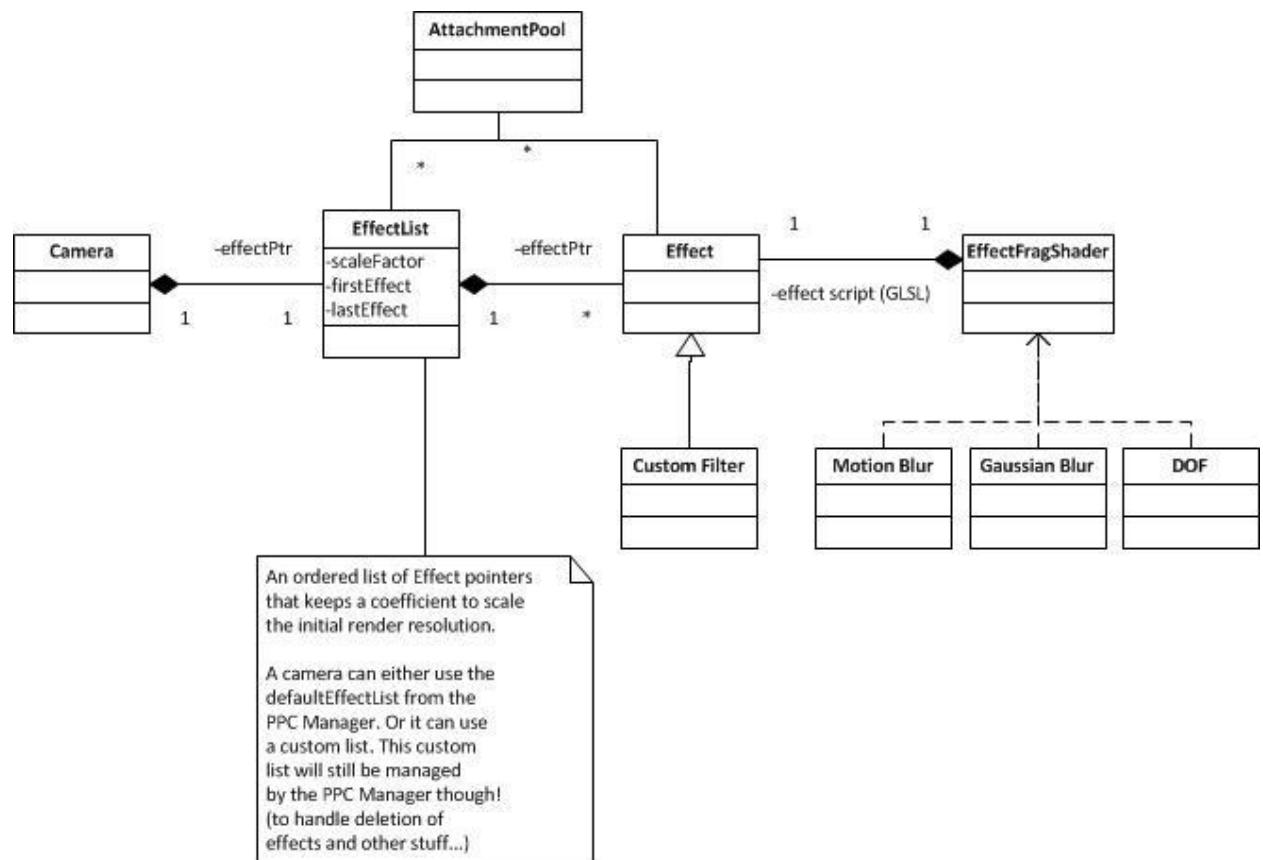


Abbildung 9: Post Processing Architektur

3.2.3.4 FBOs vs Attachments

Bei der Entwicklung des generischen Post Processings galt es eine Entscheidung zwischen zwei Arten von „Render to Texture“ in OpenGL zu treffen. Siehe „Anwendung der Zwischentextur“ für eine Erklärung von FBOs und FBO Attachments.

Was ist die optimalere Implementation? Verwenden wir einen separaten FBO für jede Zwischentextur oder verwenden wir einen einzelnen FBO und wechseln dessen Attachments?

Bei unseren Recherchen fanden wir Argumente für beide Varianten.

Nach Tests mit 300 Zwischentexturen auf zwei Maschinen kamen wir zum Schluss, dass beide Varianten ca. die gleiche Leistung erbringen.

3.3 Integration SLProject

Für die finale Rift-Integration ins SLProject haben wir uns für einen nicht generischen Ansatz entschieden. Wegen den im letzten Kapitel angesprochenen Problempunkten war dies die vorerst beste Lösung. Wenn alle Unklarheiten gelöst sind ist der Einbau von generischem Post Processing ins SLProject lediglich ein kleiner Aufwand.

Für die Implementation wurde eine neue Klasse zu SLProject hinzugefügt: SLOculus. Diese Klasse enthält alle Funktionalitäten welche für den Gebrauch der Rift benötigt werden: Polling der Orientierung, Polling sonstiger Oculus SDK Daten und Rendern in Zwischentextur und zurück in den Bildschirm mit der Linsenverkrümmung. Die Implementationsdetails wurden bereits in Kapitel 2.3 und 3.1 erläutert.

4 Fazit

Die Arbeit mit der Oculus Rift über diese ersten sechs Monate war sehr lehrreich. Ein Grossteil der Zeit wurde mit Recherchen über Rendertechniken und Funktionsweise der Rift verbracht. Es gab viele Bereiche zum Einlesen und Erfahrungen sammeln. Eine erste funktionierende Applikation für die Rift war schnell geschrieben, jedoch gab es Unklarheiten bei einigen Implementationsdetails. Wir verbrachten also auch einige Zeit damit andere Software-Beispiele zu analysieren.

Neben der simplen Rift Implementation arbeiteten wir an generischem Post Processing. Diese Arbeit ging schnell auch in Richtung von Deferred Rendering, welches sehr eng mit Multipass-Rendertechniken zusammenhängt. Durch diese Arbeit informierten wir uns auch stark über Deferred Rendering und diverse Post Process und Render-Pipeline Implementationen.

Die erste Hälfte dieses Projektes war eine gute Erfahrung und wir sind sehr gut auf das weiterführende Projekt vorbereitet. Wichtig wird es sein, dass wir nicht versuchen zu generische Lösungen zu schreiben, sondern vorerst Funktionierenden Code zu liefern.