

Unity 3D Server for CAVE Rendering

Bachelorthesis

Studiengang:	Informatik
Autor:	Julien Villiger, Daniel Inversini
Betreuer:	Prof. Urs Künzler
Auftraggeber:	cpvr Lab BFH
Experten:	Harald Studer
Datum:	01.01.2016

Management Summary

Die vorliegende Bachelorarbeit behandelt die Programmierung eines Unity Plugins sowie die Konfiguration und Inbetriebnahme des Multi-Screen Rendering Setups des CAVEs der BFH.

Die Arbeit verfolgt das Ziel, neben der bereits vorhandenen Cluster-Rendering Lösung neu einen Unity 3D Render-Server in Betrieb zu nehmen und dank Unity einem breiten Publikum zur Verfügung zu stellen. Somit kann der schnell wachsenden Verbreitung von Unity Rechnung getragen werden und der CAVE der BFH ist über eine neue Plattform ansprechbar.

Das Plugin als Kernkomponente übernimmt das gesamte Rendering für die Seitenwände des CAVEs, die Kommunikation mit dem bereits vorhandenen Trackingsystem von WorldViz, die Implementierung verschiedener VR-Eingabegeräte und weitere Aufgaben wie das Behandeln von GUI-Elementen, sekundären Kameras und Systemeinstellungen.

Plug & Play soll auch für die Verwendung des Plugins seine Gültigkeit haben. Die gesamte Funktionalität soll per Drag & Drop in ein neues oder bereits bestehendes Unity Projekt übernommen werden können. Bestehende Einstellungen und Funktionalität eines Spiels sollen nicht überschrieben werden, lediglich erweitert.

Der Hauptbestandteil der Umsetzung ist die virtuelle Abbildung der Komponenten. Mit Hilfe von Unity wird die Weiterverarbeitung und Interpretation vereinfacht und ist somit Basis für sämtliche Manipulationen der Applikation. Die Erweiterung der Projektion auf mehrere Leinwände erfolgt mittels Generierung weiterer virtueller Kameras, die sich einerseits der Logik der Applikation anpassen und andererseits Inputs vom Benutzer über das Trackingsystem entgegennehmen. Gleichzeitig wird mit Hilfe dieser erstellten Kameras, polarisierten Beamern und polgefilterten Brillen eine stereoskopische Projektion ermöglicht.

Um einer generischen Lösung gerecht zu werden, stehen etliche Einstellungseinstellungen zur Verfügung. So können beispielsweise zusätzliche Kameras individuell platziert oder die Darstellung der GUI-Elemente auf eine CAVE-Leinwand fixiert werden. Weiter sind die Buttons auf dem WorldViz Wand, dem primären Inputgerät, frei zuordenbar, um der eigenen Applikation zu entsprechen. Falls auf Wunsch nur spezifische Achsen bei der Weiterverarbeitung der Devices beachtet werden sollen, können diese auch frei ein- und ausgeschaltet werden. Die Viewfrustum-Transformation, welche basierend auf der Position des Benutzers im CAVE berechnet wird, gewährleistet eine realistische Perspektive im virtuellen Raum und lässt den Benutzer in die künstliche Welt eintauchen.

Abgerundet wird die Arbeit mit einem extra erstellten Demospiel, welches die Möglichkeiten des CAVEs zusammen mit Unity, dem Plugin und dem Trackingsystem demonstriert. Zusätzliche kleine Demos zeigen weitere Anwendungsfälle wie Simulationen oder sonstige Demonstrationen auf. Dank der Einfachheit des Plugins können erstellte 3D Modelle innert kurzer Zeit hautnah erlebt werden, was auch für andere Abteilungen der BFH von grossem Nutzen sein kann. Um die Inbetriebnahme des Plugins für Drittanwender zu vereinfachen, wird eine Schritt-für-Schritt Anleitung zur Verfügung gestellt.

Inhalt

Management Summary	2
1 Einführung	5
1.1 Vorarbeiten Projekt 2	5
2 Infrastruktur	6
2.1 Trackingsystem WorldViz	8
2.2 Devices	8
2.3 Unity Server	10
2.4 Audio	10
2.5 Video Matrix Switch	10
3 Architektur der Komponenten	11
3.1 Unterteilung Module	13
3.2 Sequenzdiagramm	13
4 Stereoskopie	15
4.1 Kameraeinstellungen	15
4.2 GUI Kameras	15
4.3 Cursor Kameras	15
4.4 Sekundäre Kameras	16
4.4.1 VR Namespace Unity	16
4.5 Mosaic Settings	17
5 Immersion	19
5.1 Frustum allgemein	19
5.2 CAVE XXL Frustum	20
5.3 General Projection Matrix Frustum	21
5.4 Vergleich CAVE XXL und General Projection Matrix	22
6 Devices	23
6.1 Wand	23
6.2 Eyes	27
6.3 Gamepad	28
7 VRPN	29
7.1 Verwendung mit PPT Studio WorldViz	29
7.2 Datenverarbeitung im Unity	29
7.3 Datenveredlung im Unity	30
8 Unity Plugin	32
8.1 Konfiguration	32
8.2 Aufgabenverteilung	35
8.3 Deployment	41
8.4 Troubleshooting	46
8.5 Performance Verbesserungen	46
9 Warping	49
10 Demo Apps	50
10.1 Shooting Gallery	50
10.2 Model-Viewer	56
10.3 App Drittpartei	57
11 Testing	Fehler! Textmarke nicht definiert.
11.1 PPT-Studio und VRPN	Fehler! Textmarke nicht definiert.
11.2 Wrapping VRPN in C# und Unity	Fehler! Textmarke nicht definiert.
11.3 UnityPlugin Projekt als Debugger	Fehler! Textmarke nicht definiert.
11.4 Performance Tests Unity Server	Fehler! Textmarke nicht definiert.
12 Abbildungsverzeichnis	62
13 Tabellenverzeichnis	63
14 Quellcodeverzeichniss	63
15 Glossar	64

16 Literaturverzeichnis	64
17 Anhang	65
18 Selbständigkeitserklärung	66

1 Einführung

1.1 Vorarbeiten Projekt 2

Im Rahmen des vorgängigen Projekts, die Projekt 2 Arbeit, wurden verschiedene Methoden geprüft wie eine Integration von Unity in den CAVE erfolgen kann.

In einem ersten Schritt erfolgte eine Prüfung der bereits verwendeten Frameworks (Equalizer, Chromium) und der Software MiddleVR, die den Einsatz von Unity in einem CAVE deutlich vereinfachen sollte. Die dadurch entstehende Abhängigkeit der Software, die Anforderungen an die Unity -Applikationen (etliche Adaptionen am Code waren jeweils notwendig) und die nicht gebrauchte Fülle an Features waren der Grund, wieso eine eigene Umsetzung eines Unity-Plugins erfolgen sollte.

Bezüglich der Infrastruktur wurden folgende Methoden analysiert und bewertet:

1. Mehrere Hosts / Mehrere GPUs / Mehrere Unity Instanzen
2. Ein Host / Mehrere GPUs / Mehrere Unity Instanzen
3. Ein Host / Mehrere GPUs / Eine Unity Instanz
4. Ein Host / Mehrere GPUs / Eine Unity Instanz mit Mosaic¹ Treiber

Basierend auf Prototypen der verschiedenen Systeme und theoretischer Abklärungen konnte sich Variante 4 (**Ein Host / Mehrere GPUs / Eine Unity Instanz mit Mosaic Treiber**) klar hervorheben. Die dadurch erreichte hohe Flexibilität, die obsolete Synchronisierung und die gute Performance durch Verteilung der Last auf verschiedene GPUs waren ausschlaggebend.

Mosaic ist eine Technologie von Nvidia, um mehrere Graphikkarten über den Treiber zu verlinken und auf einem Desktop darzustellen. Windows wird ein einzelner Output vorgegaukelt, auch wenn physisch mehrere GPUs mit multiplen Ausgängen angeschlossen sind.

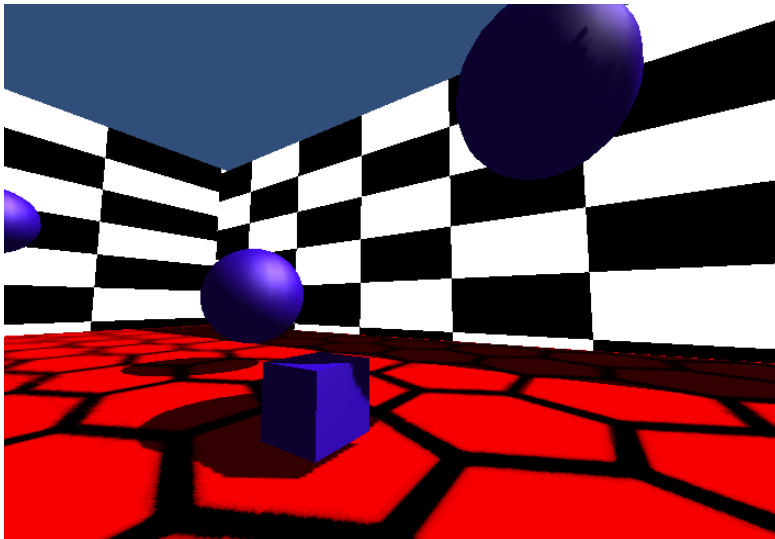


Abbildung 1: Prototyp 1 Projekt 2

¹ Nvidia Mosaic - <http://www.nvidia.com/object/nvidia-mosaic-technology.html>

Kommentiert [vi1]: Fehlt da noch die genauere Beschreibung (laut Feedback Künzler)?

Kommentiert [DI2R1]: Das wäre eigentlich mit dem block oben erschlagen



Abbildung 2: Prototyp 2 Projekt 2

2 Infrastruktur

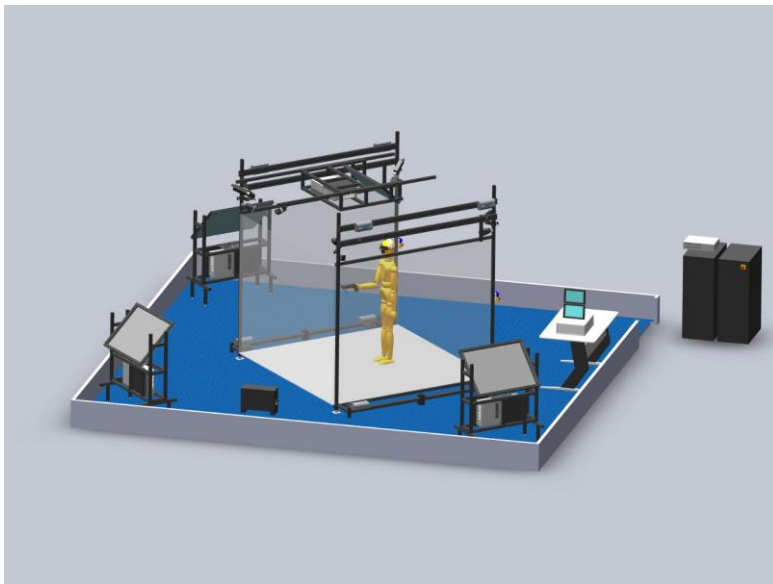


Abbildung 3: CAVE BFH

Der virtual reality haptic CAVE der BFH bietet Entwicklern und Forschern ein mächtiges Instrument, um hochrealistische immersive Applikationen zu bauen. Der CAVE ist eine kubische Konfiguration mit vier Leinwänden (Links, Front, Rechts, Boden) die von je 2 Projektoren bestrahlt werden. Um eine realistische 3D Stereoprojektion zu erschaffen, sind bei den Projektoren Polfilter angebracht und die Benutzer tragen entsprechend eine Brille mit Polfilter.

Alle Komponenten des CAVE und dessen Abhängigkeiten werden in der folgenden Grafik dargestellt:

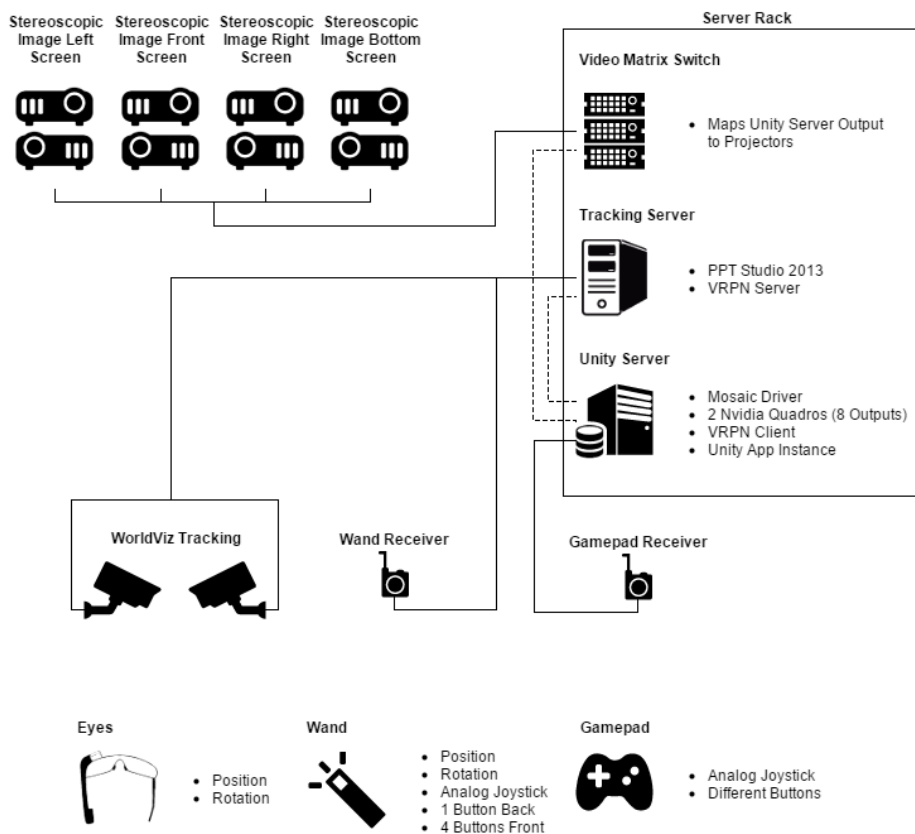


Abbildung 4: Infrastruktur CAVE

2.1 Trackingsystem WorldViz

Das Trackingsystem von WorldViz wurde integriert, um Positionen und Rotationen von verschiedenen Devices im CAVE zu erfassen. Die von den 10 WorldViz Kameras übermittelten Daten werden vom PPT Studio 2013 zentral auf einem Server interpretiert, d.h. es werden Punkte im Raum und dessen Rotation der Devices berechnet, und können bei Bedarf über das VRPN-Protokoll abgefragt werden.

2.2 Devices

• Eyes

Die Eyes von WorldViz sind Brillen mit Polfilter und zwei montierten Infrarot-Trackern. Somit lassen sich die Position des Kopfes und dessen Rotation auf zwei Achsen (Yaw und Roll) bestimmen. Die dritte Achse (Pitch) kann mit lediglich zwei Trackern nicht ermittelt werden. Dazu wären mindestens drei LEDs oder ein Gyrometer notwendig.



Abbildung 5: PPT Eyes, Quelle: www.worldviz.com

- **Wand**

Der Wand von WorldViz ist das primäre Eingabegerät. Nebst zwei Infrarot-Tackern, welche für die Positions- und Rotationsbestimmung verwendet werden, ist ein Gyrometer integriert, um noch präziser Drehungen feststellen zu können. Dadurch wird auch die fehlende Rotationsachse (fehlende bei den Eyes) kompensiert und es können Yaw, Roll und Pitch ermittelt werden.

Als Inputs dienen ein analoger Joystick, vier Buttons auf der Vorderseite sowie ein Button auf der Rückseite.

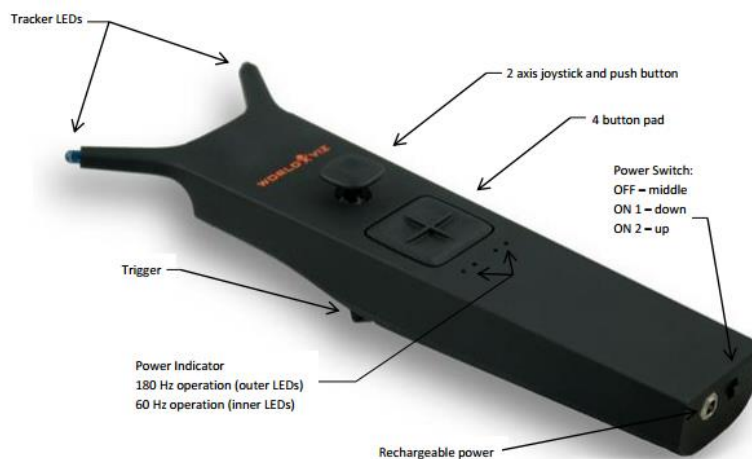


Abbildung 6: PPT Wand, Quelle: www.worldviz.com

- **Gamepad**

Ein weiteres Inputgerät ist ein handelsübliches Gamepad. Frei von jeglichem Tracking wird einzig die Unity-Applikation gesteuert.



Abbildung 7: Gamepad, Quelle: www.androidrundown.com

2.3 Unity Server

Der leistungsstarke Unity Server ist der Knotenpunkt des gesamten Systems. Auf diesem Rechner läuft die Unity-Applikation mit dem konfigurierten UnityPlugin, welches die Trackingdaten vom Trackingserver abgreift und das korrekte Rendering in der Unity-Applikation für die Projektoren Aufteilung sicherstellt. Mittels Mosaic, einem speziellen Treiber von Nvidia der die Aufteilung auf mehrere Grafikkartenausgänge übernimmt, werden alle Projektoren korrekt für die stereoskopische Projektion angesprochen.

2.4 Audio

Um die Immersion weiter zu steigern, ist ein 3D Soundsystem mit vier Lautsprechern in Betrieb.

2.5 Video Matrix Switch

Weil parallel weitere Clients in Betrieb sind und Bilddaten für den CAVE liefern können, wird ein Video Matrix Switch eingesetzt, um die verschiedenen Inputquellen auf die 8 bestehenden Projektoren abbilden zu können.

3 Architektur der Komponenten

Damit das Unity Plugin einwandfrei läuft, ist eine enge Zusammenarbeit zwischen realen und virtuellen Komponenten vonnöten. Beispielsweise haben die Inputdevices Wand und Eyes ein virtuelles Pendant, um dessen Eigenschaften besser verwerten und weiterverarbeiten zu können. Zusätzlich ermöglicht dieses Konzept ein sauberes Debugging und eine ansprechende Visualisierung.

Die folgende Grafik zeigt die Verschmelzung der realen und virtuellen Welt mit je einem halben CAVE und deren spezifischen Komponenten auf.

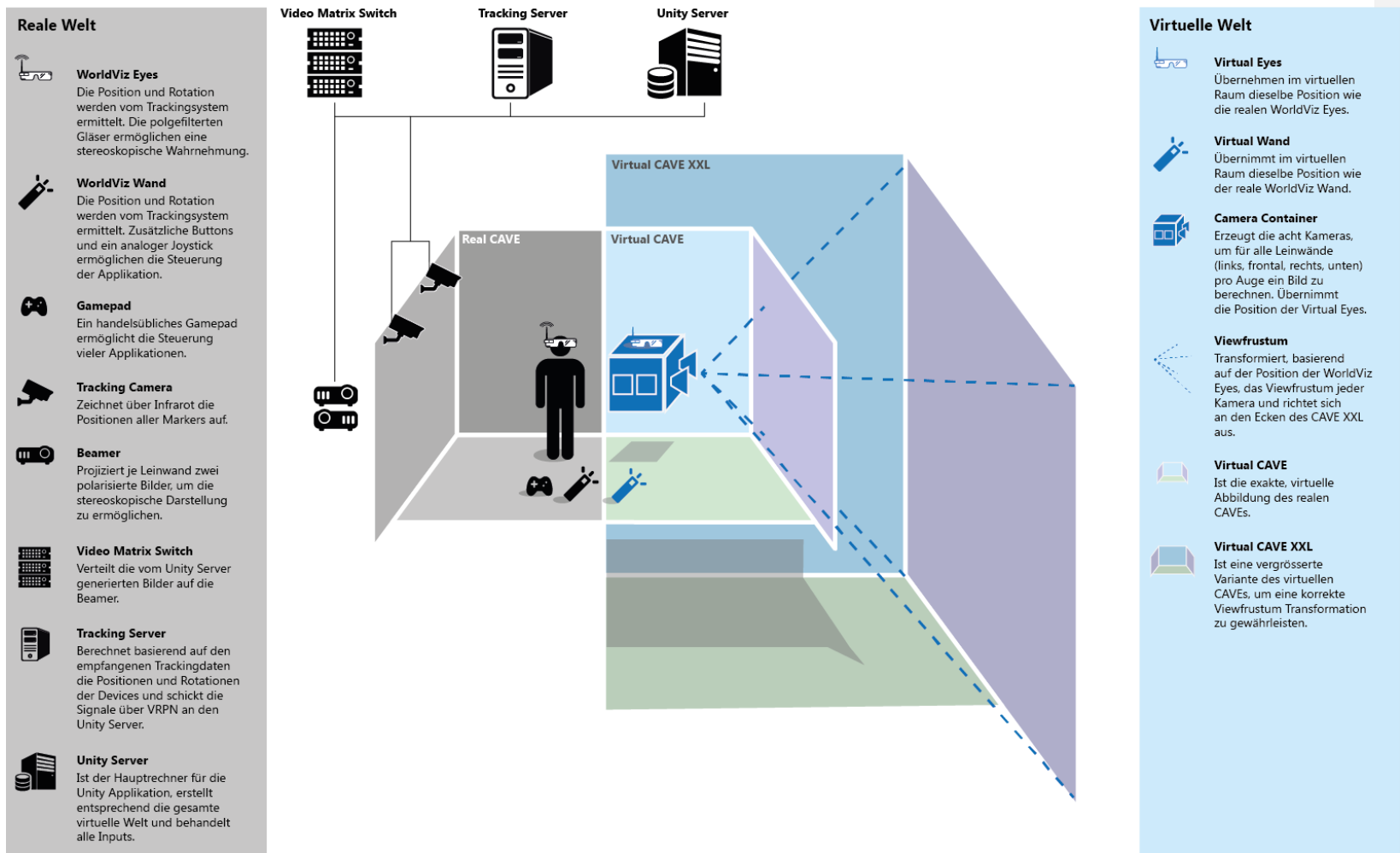


Abbildung 8: Übersicht der Komponenten

3.1 Unterteilung Module

Das modulare Konzept der objektorientierten Programmierung wurde streng eingehalten. Jede einzelne Komponente verfügt über einen klar abgegrenzten Aufgabenbereich und hat möglichst wenige Verknüpfungen zu anderen Modulen. Diese für das menschliche Denken intuitive Aufteilung erleichtert die Programmierung enorm und vereinfacht den Einstieg und die Einarbeitung in den Code für Aussenstehende.

3.2 Sequenzdiagramm

Folgendes Diagramm zeigt ein Update, also eine Abfolge für die Berechnung eines Frames, welches Unity durchführt. Die physikalischen Komponenten Wand und Eyes werden nicht dargestellt, jedoch das virtuelle Pendant in Unity. Der Benutzer macht im CAVE eine Bewegung, die vom Trackingsystem wahrgenommen und mittels Unity Plugin weiterverarbeitet wird um schlussendlich ein Bild auf den Projektoren anzuzeigen.

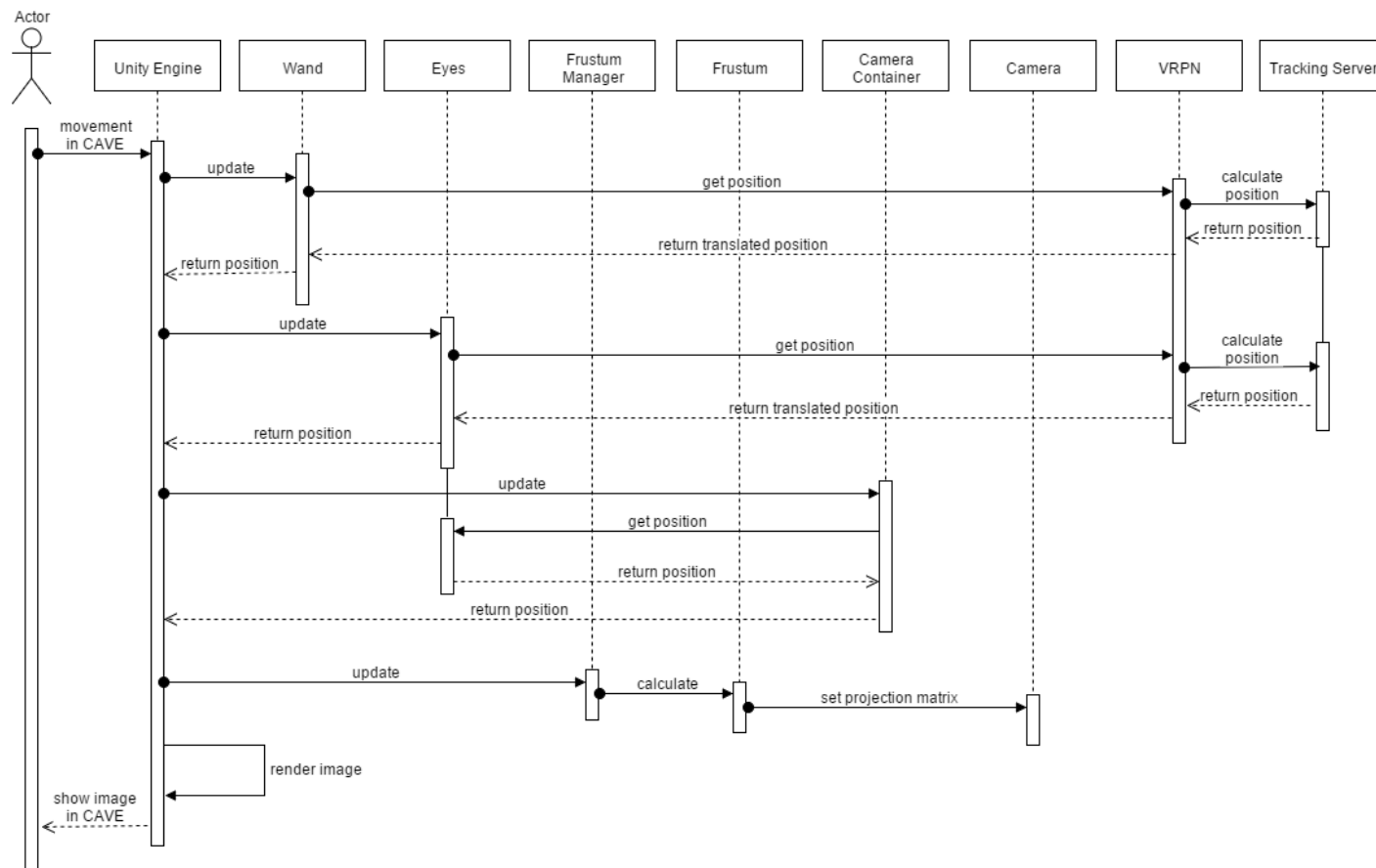


Abbildung 9: Sequenzdiagramm

4 Stereoskopie

4.1 Kameraeinstellungen

Für jede Seitenwand des CAVes werden mehrere Kameras instanziiert und dem CameraContainer hinzugefügt:

- 3D Render Kamera für das linke Auge
- 3D Render Kamera für das rechte Auge
- GUI Render Kamera, aktiv falls sich GUI Elemente auf dieser Seite befinden
- Cursor Kamera, aktiv falls sich der Cursor auf dieser Seite befindet

Die 3D Render Kameras werden wie folgt instanziiert:

- An der Position der Hauptkamera (übernommen von der Applikation)
- Rotiert um jeweils 90, 0, -90 Grad um Y für die Seitenwände
- Rotiert um 90 Grad um X für den Boden
- Verschieben um die halbe Augendistanz für den passiven Stereoeffekt

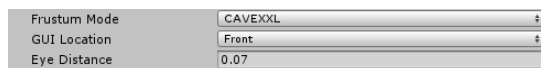


Abbildung 10: Einstellungen Augendistanz

```
Left = new CameraInfo
{
    Cam = _cameraLeftLeft,
    CamGUI = API.Instance.Cave.CaveSettings.GUILocation == BasicSettings.Sides.Left ?
        Instantiate(_cameraLeftLeft) : null,
    CamCursor = Instantiate(_cameraLeftLeft),
    Offset = new Vector3(0f, 0f, -(API.Instance.Cave.CaveSettings.EyeDistance / 2))
},

Right = new CameraInfo
{
    Cam = _cameraLeftRight,
    CamGUI = API.Instance.Cave.CaveSettings.GUILocation == BasicSettings.Sides.Left ?
        Instantiate(_cameraLeftRight) : null,
    CamCursor = Instantiate(_cameraLeftRight),
    Offset = new Vector3(0f, 0f, +(API.Instance.Cave.CaveSettings.EyeDistance / 2))
}
```

Quellcode 1: Kamerainstanzierung

4.2 GUI Kameras

Der Benutzer kann die Position der GUI-Elemente auf eine CAVE-Seite festlegen. Beim initialisieren der Applikation werden sämtliche 2D-Elemente gesucht und so im virtuellen Raum platziert, dass die Darstellung direkt auf der CAVE-Leinwand erfolgt und kein Tiefeneffekt entsteht. Gleichzeitig werden spezielle GUI-Kameras für das linke sowie das rechte Auge instanziiert, die einzig dazu da sind, GUI-Elemente zu rendern. Diese zusätzlichen Kameras sind notwendig, damit keine 3D-Objekte die Sicht auf die UI-Elemente nehmen.

4.3 Cursor Kameras

Ähnlich wie für die GUI-Elemente werden auch für den Cursor spezielle Kameras erstellt, jedoch insgesamt 8 Stück (pro Seite je eine Kamera pro Auge). Je nachdem wohin der Wand zielt, also wo der Cursor dargestellt werden soll, aktivieren sich diese Cursor Kameras und stellen einen Windowscursor

dar. Der Systemcursor kann nicht verwendet werden, weil der nur einmal vorhanden ist und somit nur für ein Auge angezeigt werden könnte. Deshalb wird die Position bestimmt und mit Hilfe von Unity zwei Kopien des Cursors dargestellt. Gleichzeitig wird der richtige Cursor ausgeblendet, damit kein Konflikt entsteht.

Falls ein Custom Cursor gewünscht ist, kann dieser dem Plugin angegeben werden und das entsprechende Sprite wird angezeigt.

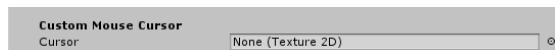


Abbildung 11: Custom Cursor

4.4 Sekundäre Kameras

Je nach Applikation kann es vorkommen, dass neben einer Hauptkamera (deren Viewport an die Wände des CAVes projiziert werden), noch mehrere sekundäre Kameras vorhanden sind. Dies können eine Minimap, ein Querschnitt eines Körpers, eine Aussenkammer oder auch nur Rückspiegel bei einem Fahrzeug sein.



Abbildung 12: Minimap

Das Plugin löst diese Anforderung wie folgt:

- Der Benutzer kann entscheiden, auf welche Seite er die sekundäre Kamera zuordnen will.
- Es müssen lediglich all die gewünschten Kameras dem Plugin übergeben werden.

4.4.1 VR Namespace Unity

Unity verfügt seit Version 5.1 über einen VR Namespace. Dieser wird verwendet, um externe Plugins/SDKs, wie das der Oculus Rift, abzugrenzen. Mit diesem Namespace möchte Unity folgende Ziele erreichen:

- Vermeiden von Konflikten der Plugins untereinander
- Mehrere VR Geräte brauchen nicht mehr mehrere Plugins (Reduzierung Aufwand)
- Neue SDKs der Hersteller können mit älteren Versionen der Spiele nicht kompatibel sein

Das Basis-API unterstützt aktuell nur folgende Features:

- **Automatisches Stereodisplay**

Es ist nicht mehr nötig, wie bisher zwei Kameras zu instanzieren, dies wird von Unity übernommen. Das funktioniert jedoch nur für alle Kameras, die nicht auf eine Textur gerendert werden.

- **Head-Tracking als Input**

Dieser Input wird analog dem umgesetzten Unity Plugin behandelt.

Da dieses Feature erst mit Version 5.1 (Release Juni 2015) verfügbar war und es im Moment nur über Basisfähigkeiten verfügt, wird dies nicht verwendet.

(Quelle: <http://docs.unity3d.com/Manual/VROverview.html>)

4.5 Mosaic Settings

Mosaic ist eine Technologie von Nvidia, um mehrere Graphikkarten über den Treiber zu verlinken und auf einem Desktop darzustellen. Windows wird ein einzelner Output vorgegaukelt, auch wenn physisch mehrere GPUs mit multiplen Ausgängen angeschlossen sind.

„Die NVIDIA® Mosaic™ Mehrbildschirm-Technologie dient zur einfachen Skalierung jeder Anwendung auf mehrere Bildschirme, und das ohne Softwareanpassungen oder Leistungseinbußen. Durch die Mosaic Technologie werden Mehrbildschirm-Konfigurationen vom Betriebssystem als einzelner Bildschirm wahrgenommen.“

(Quelle: <http://www.nvidia.de/object/nvidia-mosaic-technology-de.html>)

Die Mosaic Einstellungen des Unity Servers sind wie folgt:

- 1280px auf 1024px Auflösung pro Ausgang
- Eine 2 auf 4 Verteilung der acht Bildschirme
- Gesamtauflösung von 5120px auf 2048px

Karte 1 Ausgang 1 Position 0;0	Karte 1 Ausgang 2 Position 0;1	Karte 2 Ausgang 1 Position 0;2	Karte 2 Ausgang 2 Position 0;3
Karte 1 Ausgang 3 Position 1;0	Karte 1 Ausgang 4 Position 1;1	Karte 2 Ausgang 3 Position 1;2	Karte 2 Ausgang 4 Position 1;3

Tabelle 1: Mosaic Setting schematisch

Kommentiert [DI3]: Bilder von Settings UnityServer

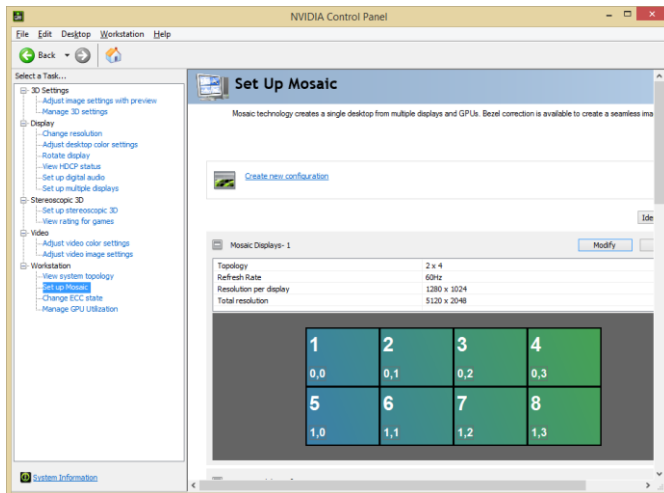


Abbildung 13: Mosaic Setting Unity Server

5 Immersion

5.1 Frustum allgemein

Für eine optimale Projektion muss das bereits vorhandene Frustum der Applikation angepasst werden. Das Frustum wird im CAVE direkt bestimmt durch die geometrischen Eigenschaften der Wände des CAVES (initial) und wird durch die Position des Benutzers im CAVE aktualisiert (runtime). Ganz allgemein bildet das Frustum das 3D Bild der Computergraphik auf einen zweidimensionalen Bildschirm ab.

Es gibt mehrere Möglichkeiten ein Frustum aufzubauen, die folgenden zwei Methoden wurden umgesetzt:

- Frustum mit einer Projektionsfläche XXL (Seitenwand des CAVE virtuell ~30m entfernt, Dimensionen beibehalten)
- Frustum mit der genauen Projektionsfläche der CAVE-Wand und einer Off-Axis Projektion

Diese unterschiedlichen Möglichkeiten haben sich aus dem Adaptieren des bereits vorhandenen Frustums ergeben, den eigenen Ansätzen und Überlegungen und der Adaptierung der generalisierten Projektionsmatrix des Ur-CAVES.

Kommentiert [DI4]: Braucht noch viele fusszeilen

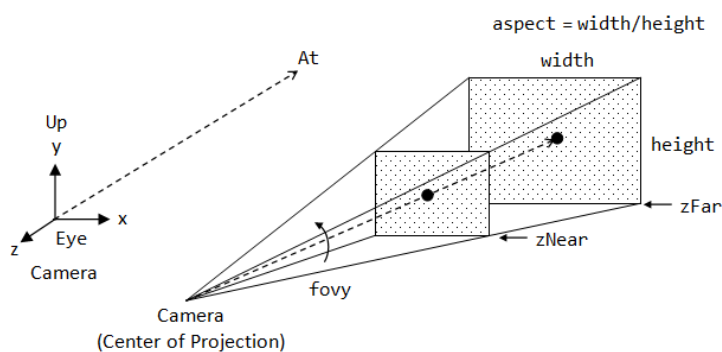


Abbildung 14: Frustum, Quelle: www.stackoverflow.com

Ein Frustum wird in Unity wie folgt dargestellt. Sichtbar hier ist, dass die Seitenwand des CAVE XXL die Plane bildet, woran das Frustum aufgespannt wird.

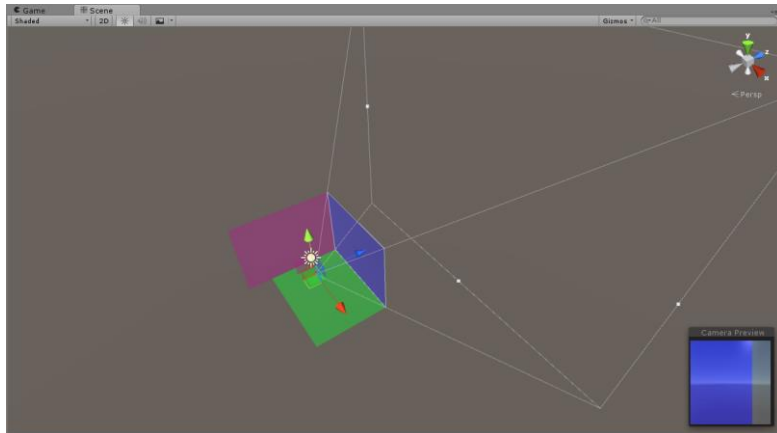


Abbildung 15: UnityPlugin Frustum 1

Mit allen 8 Kameras ergibt sich folgendes Bild:

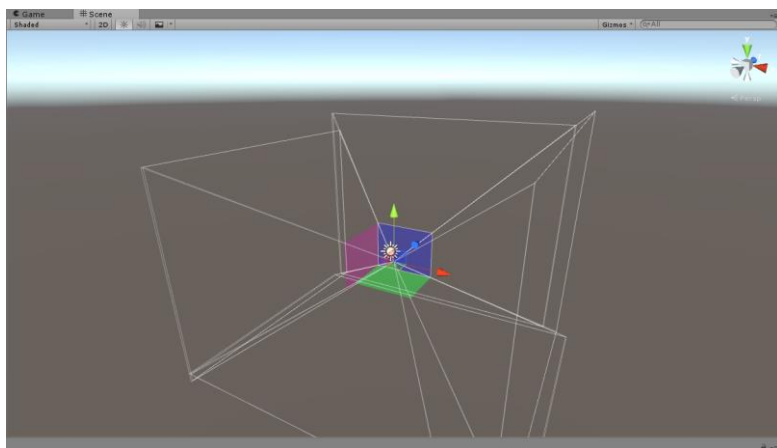


Abbildung 16: UnityPlugin Frustum 2

5.2 CAVE XXL Frustum

Falls das Frustum auf die realen CAVE Seitenwand projiziert wird, vergrößert sich das Field of View (FoV) wenn man sich der Wand nähert. Die Objekte werden dann Verzogen, wenn ein FoV von Beispielsweise 120 Grad angezeigt wird, obwohl die Projektionsfläche dieselbe bleibt.

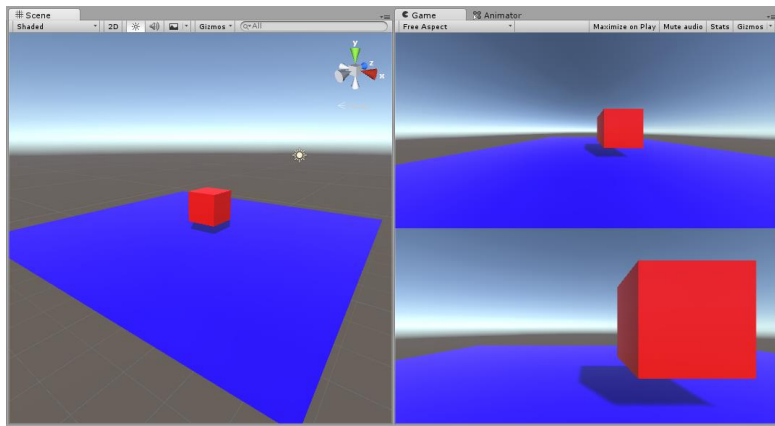


Abbildung 17: Vergleich FoV 120 & 60 Grad

Um diesen Effekt abzuschwächen, wird in diesem Betriebsmodus nicht die Seitenwand des CAVEs verwendet um das Frustum aufzuspannen, sondern die Seitenwand des CAVE-XXL. Dies hat folgende Vorteile:

- Es kann die «Generalized Perspective Projection» von Robert Kooima verwendet werden, welche 2009 publiziert wurde
- Da sich der Betrachter der Seitenwand des CAVE-XXL nie gross nähert, tritt der oben genannte Effekt nicht auf.
- Die Frustums überschneiden sich nicht, noch haben sie Lücken.

Kommentiert [DI5]: Verweis fehlt noch

5.3 General Projection Matrix Frustum

In diesem Betriebsmodus wird das FoV nach der Berechnung des Frustums noch angeglichen.

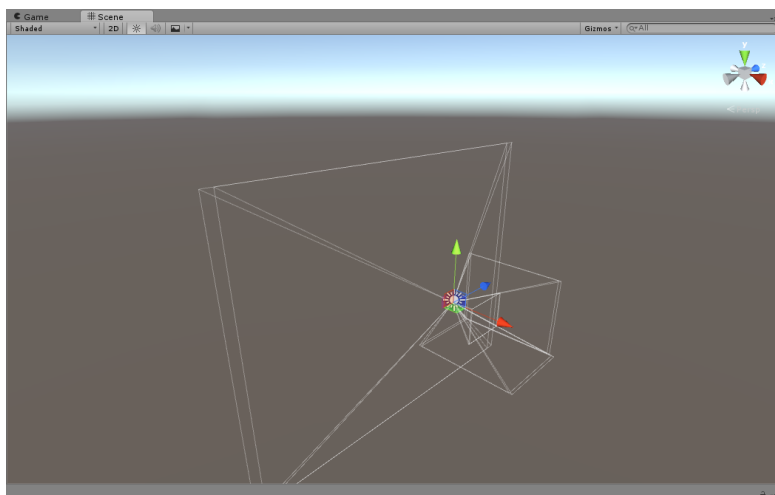


Abbildung 18: UnityPlugin Frustum GGM

Es ist sichtbar, dass hier unterschiedlich grosse Frustums in Verwendung sind, je nachdem wie nahe an der CAVE Wand man sich befindet. Die FoVs werden aufgrund des Verhältnisses von Höhe und Breite des Frustums nachgeglichen.

Kommentiert [DI6]: Hier noch graphik mit gizmos um das FOV zu zeichnen + noch etwas mathematik

5.4 Vergleich CAVE XXL und General Projection Matrix

Grundlegend basieren beide Berechnungen auf der «GPP» von Koomia. Das Frustum wird anhand einer Plane im 3D Raum aufgespannt.

Vorteile CAVE XXL

- Der Benutzer nähert sich der Seitenwand XXL nie so stark, dass ein unrealistisches FoV erzeugt wird
- Die Vergrößerung aller Distanzen erzeugt eine grössere Genauigkeit
- Weniger Berechnung nötig, da nicht noch das FoV nachberechnet werden muss, und dies dann die bereits berechnete Projektionsmatix ajustiert

Vorteile General Projection Matrix

- Konstrukt CAVE XXL nicht notwendig
- Anpassung FoV nach verbreiteter und akzeptierter Methode

Da der CAVE XXL für weitere Features (Raycast, 2D GUI) verwendet wird, halbieren sich die Vorteile der General Projection Matrix. Zusätzlich ergibt das Frustum nach CAVE XXL ein besseres 3D Bild. Somit wird standardmässig diese Methode verwendet. Zu Demonstrationszwecken sind aber beide Methoden vorhanden.

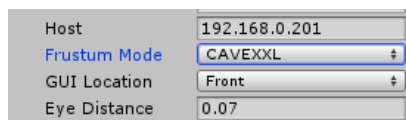


Abbildung 19: Frustum Mode

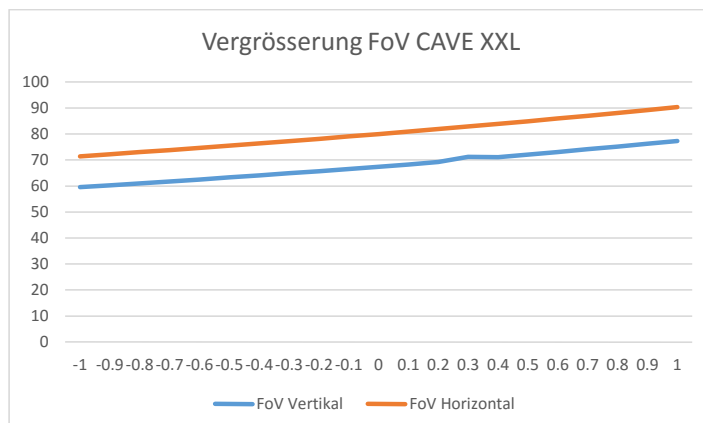


Abbildung 20: Frustum CAVE XXL FoV Anpassung

6 Devices

6.1 Wand

Die Interpretation des Wands ist ein zentrales Element für die Verwendung des CAVEs mit Unity. Er dient dazu, Objekte in der virtuellen Welt zu bewegen oder zu rotieren.

- **Virtueller CAVE**

Damit die Verarbeitung der vom Wand gelieferten Informationen vereinfacht werden können und eine visuelle Darstellung möglich ist, wurde ein virtueller CAVE in Unity erstellt, welcher dieselben Dimensionen wie der reale CAVE der BFH hat. Diese genaue Adaptierung ist möglich, weil in Unity die verwendete Grösseneinheit einem Meter in der realen Welt entspricht.

Implementiert wurde das mittels einem Prefab, welches einmalig in der Hierarchie liegt.

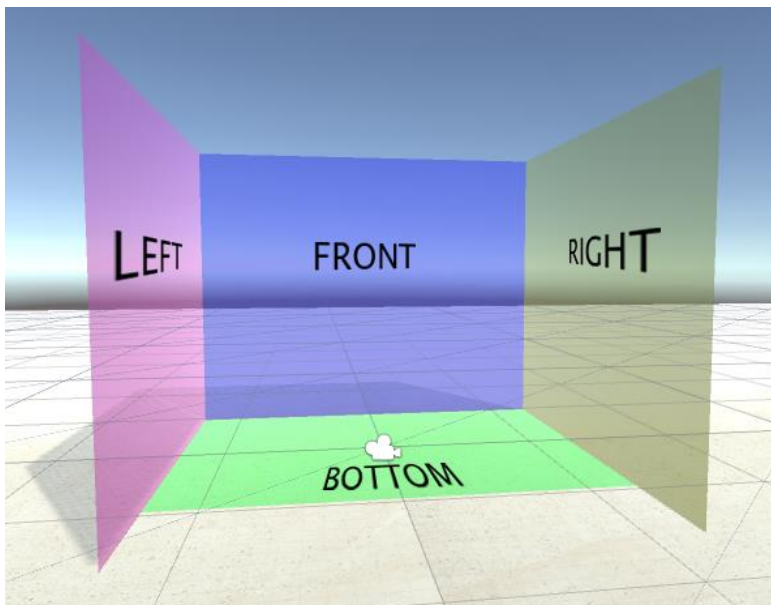


Abbildung 21: Unity Plugin, virtueller CAVE

Zusätzlich zum Cave sind auch die beiden Devices Eyes und Wand als virtuelle Objekte in der Hierarchie der Applikation.

Der virtuelle CAVE übernimmt die Position und Rotation der Hauptkamera der Applikation. Die Korrelation zwischen CAVE und Wand / Eyes kann nur bestehen, wenn sich der Spieler in der virtuellen Welt auch im virtuellen CAVE befindet. Die Hauptkamera wird rein durch die Unity-Applikation gesteuert und das Plugin hat keinerlei Einfluss darauf, um den Spielmechanismus nicht zu stören. Damit aber die Checks, wie sich die Devices im CAVE befinden, nach wie vor gemacht werden können, verändert der CAVE die Position und Rotation analog der Hauptkamera.

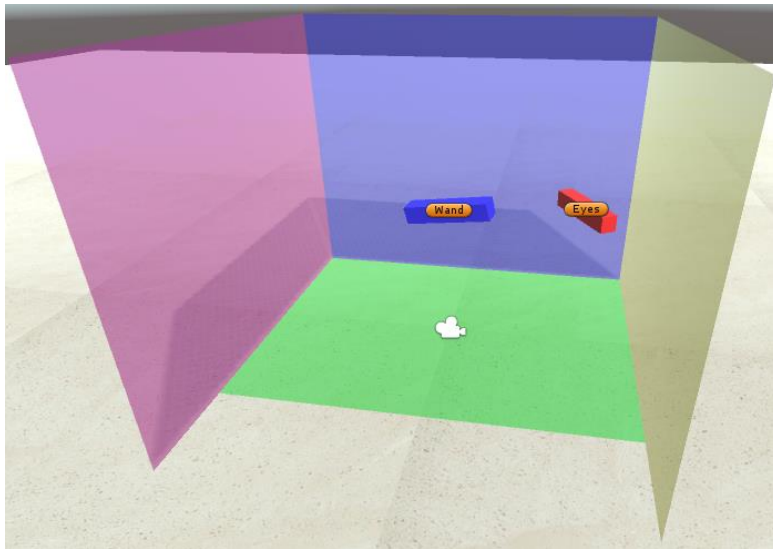


Abbildung 22: Virtueller Wand und Eyes

- **Position und Rotation**

Die realen Positions- und Rotationsveränderungen werden direkt über VRPN auf die Objekte abgebildet und ermöglichen somit ein einfaches Auslesen dieser Daten über das API.

```
var wandRotation = API.Instance.Wand.transform.rotation;
```

Quellcode 2: Zugriff über das API

Das Plugin lässt jedoch zu, einzelne Achsen bei der Positions- und Rotationsfestlegung auszuschliessen. Hierzu wird die Position, bzw. Rotation, vor der Berechnung des neuen Frames zwischengespeichert und auf die blockierten Achsen auf den ursprünglichen Wert zurückgesetzt.

Statt eine Achse komplett zu blockieren, lässt sich auch die Sensibilität adaptieren. Der Standardwert ist 1, was bedeutet, dass im realen Cave eine Verschiebung von einem Meter genau einem Meter in der virtuellen Welt entspricht. Wird die Sensibilität jedoch unter 1 gesetzt, bewegt sich der Wand langsamer und es wird eine kleinere Bewegung virtuell ausgeführt.

- **Mousecursor**

Um möglichst generisch die Steuerung der Applikationen übernehmen zu können, ist es unerlässlich, die Mausposition mittels Wand setzen zu können, weil das häufig das primäre Inputgerät ist. Wird auf dieser Ebene des Betriebssystems bereits Hand angelegt, entfallen spezifische Mappings auf Applikationslevel um die Steuerung übernehmen zu können.

Dazu wird ermittelt, wohin der Wand zielt. Die Verlängerung der x-Achse, also eine Gerade definiert durch die Rotation des Wands, kann einen Schnittpunkt mit einer Leinwand des CAVES haben.

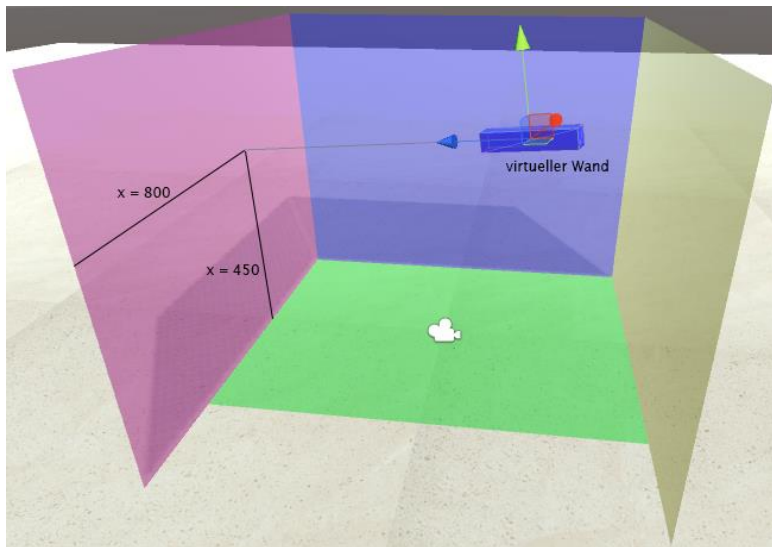


Abbildung 23: Unity Plugin, Schnittpunkt Wand / CAVE

Der Schnittpunkt wird von Unity mit einem Raycast ermittelt.

```
var fwd = transform.TransformDirection(Vector3.forward);

Ray ray = new Ray(transform.position, fwd);
RaycastHit hit;

if (Physics.Raycast(ray, out hit, 100))
{
    Vector3 localSpaceHitPoint = hit.transform.worldToLocalMatrix.MultiplyPoint(hit.point);
    ...
}
```

Quellcode 3: Raycast

Ausgehend vom virtuellen Wand wird ein sogenannter Ray geschossen, welcher wahlweise nach der ersten Kollision abbricht und das getroffene Hit-Objekt zurückgibt oder durch sämtliche Colliders weiterfliegt und alle Ergebnisse als Returnwert liefert.

Der exakte Schnittpunkt auf der getroffenen Fläche liefert nun die benötigten Informationen, um den Mousecursor auf Betriebssystemebene festzulegen. Zunächst muss aber noch unterschieden werden, welche Leinwand betroffen ist. Das Mapping funktioniert so, dass bei einem Auftreffen auf die linke Leinwand der Cursor im ersten, oberen Achtel des Bildschirms festgelegt wird und der Cursor auf den zweiten, oberen Achtel mit Hilfe des GUI-Systems dupliziert wird, damit bei der stereoskopischen Projektion beide Augen den Cursor sehen. Bei der Front-Leinwand dasselbe mit dem 3. und 4. Achtel. Findet der Schnittpunkt auf der rechten oder unteren Leinwand statt, wird der Cursor in der unteren Hälfte des Betriebssystems platziert. Weitere Informationen zur Aufteilung der Viewports auf dem Unity-Server mittels UnityPlugin werden im Kapitel „Unity Plugin“ behandelt.

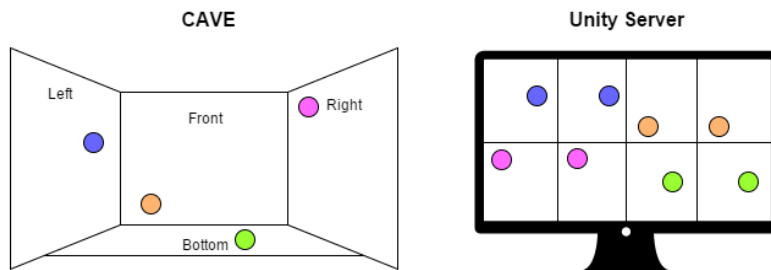


Abbildung 24: Zuordnung Schnittpunkt CAVE und Unity Server

Auf dem Unity Server ist somit jeweils der erste Punkt die reale Position des Cursors und der zweite, gleichfarbige Punkt eine Duplikation für das rechte Auge. Der reale Cursor wird jedoch nie dargestellt, weil die Kopie, welche mittels GUI-System dargestellt wird, immer einen zeitlichen Versatz aufweist und für den Benutzer ein Störfaktor darstellt. Aus diesem Grund wird der Cursor zwar platziert, damit sämtliche über den Cursor laufenden Inputs nach wie vor funktionieren, jedoch ausgeblendet und angezeigt wird eine Cursor-Grafik.

- **Buttons**

Der Wand verfügt über verschiedene Inputs. Nebst dem analogen Joystick ist ein Button auf der Rückseite und vier weitere Buttons auf der Vorderseite angebracht. Zusätzlich lässt sich der Joystick nach unten drücken.

Dem Benutzer soll die Freiheit haben zu entscheiden, welche Aktionen beim Drücken dieser Buttons ausgeführt werden und damit wiederum eine möglichst grosse Bandbreite von Applikationen abgedeckt werden können, ist für jeder Wand-Input eine Keyboard- oder Mauseingabe auswählbar.

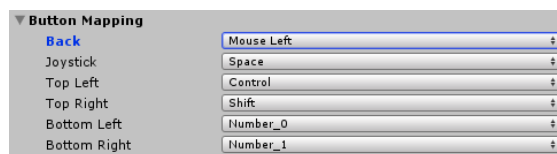


Abbildung 25: Wand Button Zuordnung

Die Auswahlmöglichkeiten beschränken sich auf eine erstellte Liste von enums, die direkt an einen virtuellen Keycode der `InputSimulator`-Bibliothek geknüpft sind. Wird nun also beim Wand ein Button betätigt, wird diese Information über das VRPN-Protokoll ans `UnityPlugin` geliefert und mittels dem `InputSimulator` ein Tastendruck simuliert. Der grosse Vorteil hier ist wiederum, dass Unity nicht unterscheiden kann, ob dieser Tastendruck von einer Tastatur kommt oder im `UnityPlugin` initialisiert wurde. Die Abfrage in der Applikation, ob eine Taste gedrückt wurde, kann also wie gewohnt über die `Input`-Klasse gemacht werden und braucht keine spezielle, vom `UnityPlugin` abhängige Implementierung. Das untenstehende Codesnippet zeigt, wie auf herkömmliche Weise in Unity das betätigen der Leertaste abgefragt wird und auch weiterhin mit dem umgesetzten `UnityPlugin` Gültigkeit hat.

Kommentiert [JV7]: Fussnote

```

if (Input.GetKeyDown(KeyCode.Space))
{
    // Applikationsspezifischer Code beim Betätigen der Leertaste
}

```

Quellcode 4: Unity Tastaturabfrage

Eine Ausnahme bilden die Mausklicks. Aus Sicherheitsgründen ist es nicht gestattet, mittels InputSimulator Maus-Inputs zu simulieren. Deshalb musste die `user32.dll` zugezogen werden, welche erlaubt, Maus-Events zu senden. Dies geschieht wiederum auf Betriebssystem-Level und hat entsprechend keinen Einfluss auf die Interpretation in der Unity-Applikation.

Kommentiert [JV8]: Fussnote

Weil, basierend auf der Unity-Architektur, die Button-Abfrage bei jedem Frame geschieht, würden die Inputs einmal pro gerendertem Frame simuliert werden. Das hätte zur Folge, dass selbst bei einem kurzen Klick von weniger als einer Sekunde, der Input mehrfach ausgeführt werden würde. Deshalb werden beim Auslösen desselben Buttons mittels Coroutinen mehrfache Ausführungen blockiert.

- **Joystick**

Weiter verfügt der Wand über einen analogen Joystick, welcher sich stufenlos auf zwei Achsen bewegen kann. Über die API des UnityPlugins lassen sich diese Werte einfach mittels Delegates auslesen. Es muss lediglich eine Funktion definiert werden, welche bei einem Joystick-Update aufgerufen wird. Der folgende Beispielcode zeigt, wie ein Objekt basierend auf den Joystick-Werten bewegt wird.

```

void Start()
{
    // Register
    API.Instance.Wand.OnJoystickAnalogUpdate += OnJoystickUpdate;
}

private void OnJoystickUpdate(float x, float y)
{
    Vector3 posNew = transform.position;
    posNew.x += x / 10f;
    posNew.z += y / 10f;

    transform.position = posNew;
}

```

Quellcode 5: Joystick Handling

Die Joystick-Werte sind Teil der VRPN-Daten, welche der Trackingserver über das PPT Studio 2013 liefert.

6.2 Eyes

Die Eyes von WorldViz sind notwendig für das stereoskopische Sehen und die Positions- sowie Rotationsbestimmung des Anwenders.

- **Position und Rotation**

Mit Hilfe zweier Infrarot-Trackern werden die Position im Raum und die Rotation auf zwei Achsen (Yaw und Roll) im PPT Studio 2013 aufbereitet und über VRPN an das UnityPlugin übermittelt. Eine Erfassung der dritten Rotationsachse (Pitch) kann ohne weiteren Tracker nicht erfolgen. Im Gegensatz zum Wand verfügen die Eyes auch nicht über ein Gyrometer.

Im Cave-Prefab des Plugins sind die Eyes ebenfalls als Objekt in der Hierarchie und übernehmen die Werte vom Trackingsystem. Über das API ist somit ein einfacher Zugriff möglich. Die selektive Fixierung und Adaption der Sensibilität funktionieren analog dem Wand.

6.3 Gamepad

Das direkt am Unity Server angeschlossene Gamepad ist nicht konfigurierbar und wird bewusst als Standard-Input belassen, um gängige Applikationen steuern zu können. Der Input-Manager von Unity deckt diese Art von Devices bereits sehr gut ab.

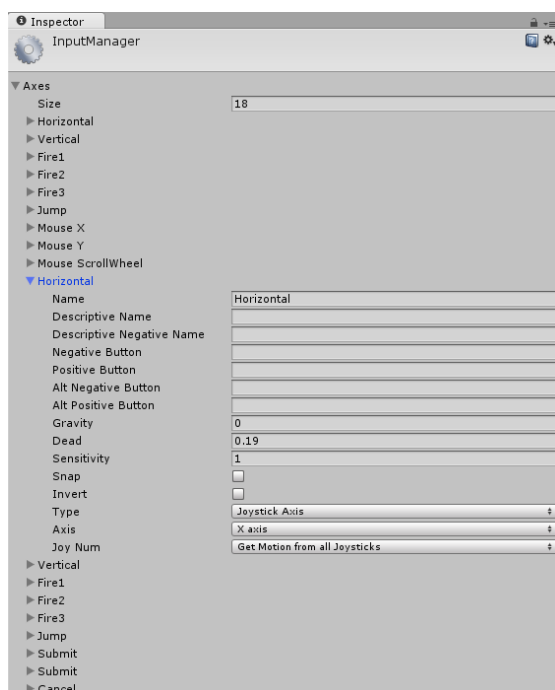


Abbildung 26: Inputmanager Unity

7 VRPN

Virtual-Reality Peripheral Network (VRPN) ist eine Klassenbibliothek und ein Server-Interface für Client-Programme und physikalische Geräte, welche in einem VR-System verwendet werden. Das Ziel von VRPN ist es, ein allgemeines Interface für Eingabegeräte wie Motion-Tracking, Joysticks, und weitere bereitzustellen.

7.1 Verwendung mit PPT Studio WorldViz

Das PPT Studio stellt seine Geräte über verschiedene Ausgänge zur Verfügung, es hat auch einen eingebauten VRPN Server.

Dieser liefert über verschiedene Trackernamen die gewünschten Informationen.

```
#define Machine Address of PPT machine
hostname = 'localhost'
#define markerID of Wand in PPT
markerID = 3
#create a tracker object for the 6DOF data
tracker = vrpn.addTracker('PPT0@'+ hostname, markerID-1)
#create analog device for the joystick
analogDev = vrpn.addAnalog('PPT_WAND%d@%s:%d' % (markerid, 8945))
#create button device for the buttons
buttonDev = vrpn.addButton('PPT_WAND%d@%s:%d' % (markerid, hostname, 8945))
```

Abbildung 27: Beispielverbindung VRPN

7.2 Datenverarbeitung im Unity

Wir haben uns folgender Wrapperbibliothek bedient, welche wir über das offizielle Git-Repository von VRPN gefunden haben:

- Offizielles Git-Repository
<https://github.com/vrpn/vrpn/wiki>
- UnityWrapper
<https://github.com/arviceblot/unityVRPN>

Diesen Sourcecode wurde in unser Git-Repository verlinkt, mit CMake kompiliert und als DLL in unserem Asset eingefügt.

Im Unity wurden dann ein kurzer Wrapper geschrieben, welche den managed Code der Library zur Verfügung stellt:

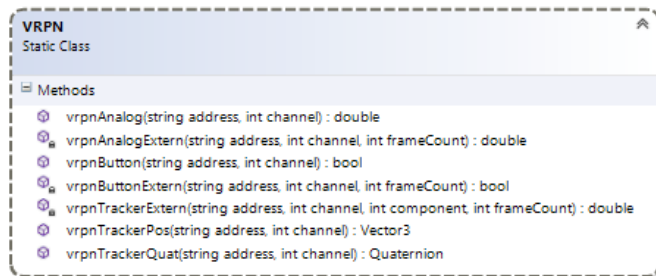


Abbildung 28: VRPN Wrapper C#

Die Weiterverarbeitung dieser Trackerdaten übernehmen dann die Klassen der Objekte, namentlich Wand und Eyes. Im Updatezyklus werden diese Daten ausgelesen, verarbeitet (Anwendung eines Smoothing-Filters) und zugewiesen (Rotation und Position).

```
private void HandlePosition()
{
    if (API.Instance.Cave.WandSettings.TrackPosition)
    {
        // Position
        var posOri = transform.localPosition;
        var pos = VRPN.vrpnTrackerPos(
            API.Instance.Cave.WandSettings.WorldVizObject + "@" +
            API.Instance.Cave.Host, API.Instance.Cave.WandSettings.Channel);

        if (_usePositionSmoothing)
        {
            Vector3 filteredPos = Vector3.zero;
            Vector3 filteredVelocity = Vector3.zero;
            OneEuroFilter.ApplyOneEuroFilter(pos, Vector3.zero, posOri,
                Vector3.zero, ref filteredPos, ref filteredVelocity,
                _posJitterReduction, _posLagReduction);
            pos = filteredPos;
        }

        // Block Axis
        if (API.Instance.Cave.WandSettings.PositionAxisConstraints.X)
            pos.x = posOri.x;
        if (API.Instance.Cave.WandSettings.PositionAxisConstraints.Y)
            pos.x = posOri.z;
        if (API.Instance.Cave.WandSettings.PositionAxisConstraints.Z)
            pos.x = posOri.z;

        transform.localPosition = pos;
    }
}
```

Quellcode 6: VRPN-Positionshandling

7.3 Datenveredlung im Unity

Da unter Umständen ein etwas unruhiger Input über das VRPN geliefert wird, wurde ein Lowpass Filter hinzugefügt. Dieser kann aktiviert, deaktiviert und mit zwei Parametern eingestellt werden. Der „1€ Filter“, welcher zum Einsatz kommt beschreibt sich wie folgt:

The 1€ filter ("one Euro filter") is a simple algorithm to filter noisy signals for high precision and responsiveness. It uses a first order low-pass filter with an adaptive cutoff frequency: at low speeds, a low cutoff stabilizes the signal by reducing jitter, but as speed increases, the cutoff is increased to reduce lag. The algorithm is easy to implement, uses very few resources, and with two easily understood parameters, it is easy to tune. In a comparison with other filters, the 1€ filter has less lag using a reference amount of jitter reduction.

(Quelle: <http://crystal.univ-lille.fr/~casiez/publications/CHI2012-casiez.pdf>)

Der 1€-Filter ist nur auf 2D ausgelegt, wurde jedoch um eine Dimension erweitert. Sowohl bei tiefen wie auch bei hohen Frequenzen und schnellen Bewegungen wurden optimale Resultate erzielt. Aufgrund dieser guten Resultate kam dieser Filter schlussendlich zum Einsatz. Folgende Grafik zeigt für die verschiedenen Geschwindigkeitsintervalle die durchschnittliche Distanz zwischen dem gefilterten und der aktuellen Cursorposition.

(Quelle: <http://crystal.univ-lille.fr/~casiez/publications/CHI2012-casiez.pdf>)

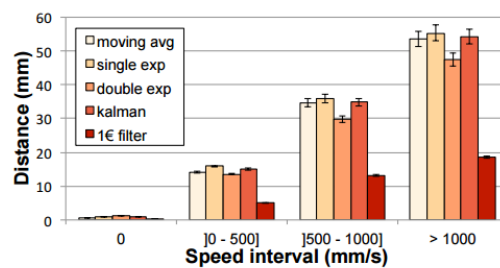


Abbildung 29: 1€ Smoothing Vergleich

8 Unity Plugin

8.1 Konfiguration

Um eine möglichst weite Bandbreite von Unity-Applikationen abdecken zu können, werden etliche Einstellungsmöglichkeiten zur Verfügung gestellt. Diese gliedern sich in fünf relevante Sektionen.

- **Wand**

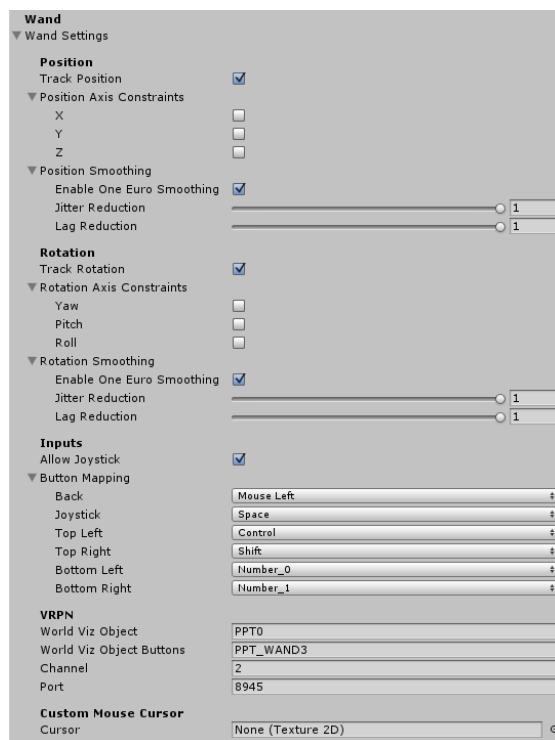


Abbildung 30: Unity Plugin, Settings Wand

Position

Falls die Position des Wands in der aktuellen Applikation unerheblich ist, kann die an dieser Stelle deaktiviert werden. Somit übernimmt der virtuelle Wand, welcher im Prefab liegt, keine Translationen vom realen Wand. Ist diese Option aber aktiviert, besteht die Möglichkeit, achsenabhängig die Sensibilität einzustellen. Das heisst, bei einer hohen Sensibilität auf der y-Achse vollführt der virtuelle Wand eine grosse Bewegung im Vergleich zu der realen Bewegung im CAVE. Gegenteilig, wird der Regler unter 0 gestellt, ist die virtuelle Bewegung kleiner als die reale Bewegung. Zusätzlich können einzelne Achsen auch komplett deaktiviert werden.

Weiter kann auf Wunsch die Interpolation (Smoothing) deaktiviert werden, es wird aber empfohlen, diese Option aktiv zu halten.

Rotation

Auch die Rotation kann auf Wunsch komplett deaktiviert werden. Ebenfalls ein Smoothing ist standardmässig aktiv.

Inputs

Der Wand verfügt über mehrere Buttons, die aus einer umfassenden Auswahlliste auf die Applikation abgebildet werden können. Zusätzlich kann der Joystick aktiviert oder deaktiviert werden.

Custom Mouse Cursor

Falls im Spiel ein spezifischer Cursor Verwendung findet, kann die Textur hier angegeben werden, damit sie für beide Augen gerendert wird. Ansonsten wird der normale Windows-Cursor angezeigt.

VRPN

Der VRPN-Block wird für die Anmeldeinformationen beim VRPN-Server verwendet. Diese müssen nur bei einer Umstellung des PPT-Studios (auf dem Tracking-Server) adaptiert werden.

- Eyes

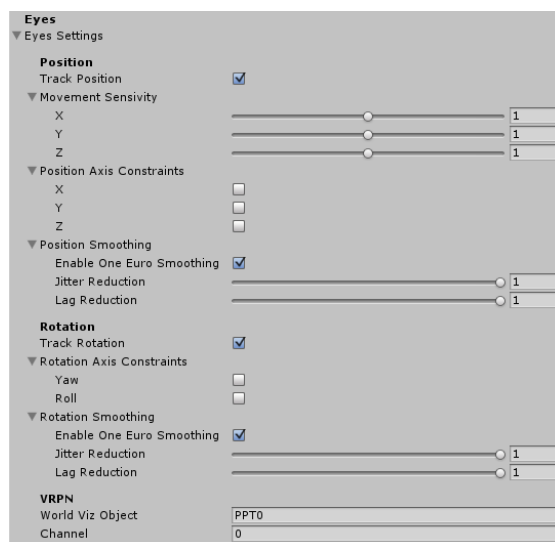


Abbildung 31: Unity Plugin, Settings Eyes

Position

Falls die Position der Eyes in der aktuellen Applikation unerheblich ist, kann die an dieser Stelle deaktiviert werden. Somit übernehmen die virtuellen Eyes, welche im Prefab liegen, keine Translationen von den realen Eyes. Ist diese Option aber aktiviert, besteht die Möglichkeit, achsenabhängig die Sensibilität einzustellen. Das heisst, bei einer hohen Sensibilität auf der y-Achse vollführen die virtuellen Eyes eine grosse Bewegung im Vergleich zu der realen Bewegung im CAVE. Gegenteilig, wird der Regler unter 0 gestellt, ist die virtuelle Bewegung kleiner als die reale Bewegung. Zusätzlich können einzelne Achsen auch komplett deaktiviert werden.

Weiter kann auf Wunsch die Interpolation (Smoothing) deaktiviert werden, es wird aber empfohlen, diese Option aktiv zu halten.

Rotation

Auch die Rotation kann auf Wunsch komplett deaktiviert werden. Ebenfalls ein Smoothing ist standardmässig aktiv.

VRPN

Der VRPN-Block wird für die Anmeldeinformationen beim VRPN-Server verwendet. Diese müssen nur bei einer Umstellung des PPT-Studios (auf dem Tracking-Server) adaptiert werden.

- **Sekundäre Kameras**

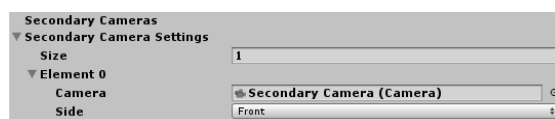


Abbildung 32: Unity Plugin, Settings Sekundäre Kameras

Möglicherweise werden für die Applikation nicht nur eine Hauptkamera, sondern auch eine oder mehrere sekundäre Kameras parallel gerendert. Eine beliebige Anzahl an Kameras können hier angegeben werden und auf welcher Seite der Leinwand die Kamera erscheinen soll.

- **Cave**

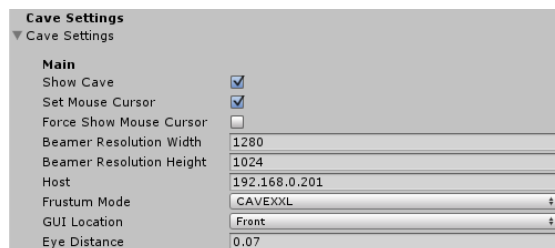


Abbildung 33: Unity Plugin, Settings Cave

In dieser Sektion in den meisten Fällen keine Einstellungen vorgenommen werden. Möglicherweise will der Benutzer aber die GUI-Elemente auf einer anderen Leinwand darstellen.

- **System**

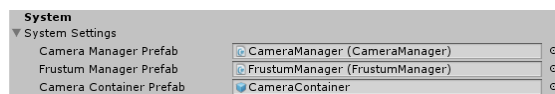


Abbildung 34: Unity Plugin, Settings System

Die zugewiesenen Prefabs werden beim Initialisieren des Plugins instanziiert und müssen nicht adaptiert werden.

Das CameraContainer Gameobject beinhaltet alle Kameras, die sich jeweils der Hauptkamera unterordnen und die Bildaufteilung für die verschiedenen Beamer übernimmt.

Einstellungen müssen nur bei grundlegenden Veränderungen gemacht werden und empfiehlt sich nur für erfahrene Benutzer.

8.2 Aufgabenverteilung

Der zentrale Knoten des Plugins ist die Klasse „CaveMain“. Hier sind alle Einstellungsmöglichkeiten, die Referenzen auf sämtliche Objekte und die Geometrie der virtuellen CAVes gespeichert.

Ein direkter Zugriff auf verschiedene Komponenten sollte grundsätzlich nicht erfolgen, dafür wird ein API als Schnittstelle für Unity-Applikationsprogrammierer zur Verfügung gestellt.

Das untenstehende Klassendiagramm zeigt sämtliche Klassen und deren wichtigsten Assoziationen auf.

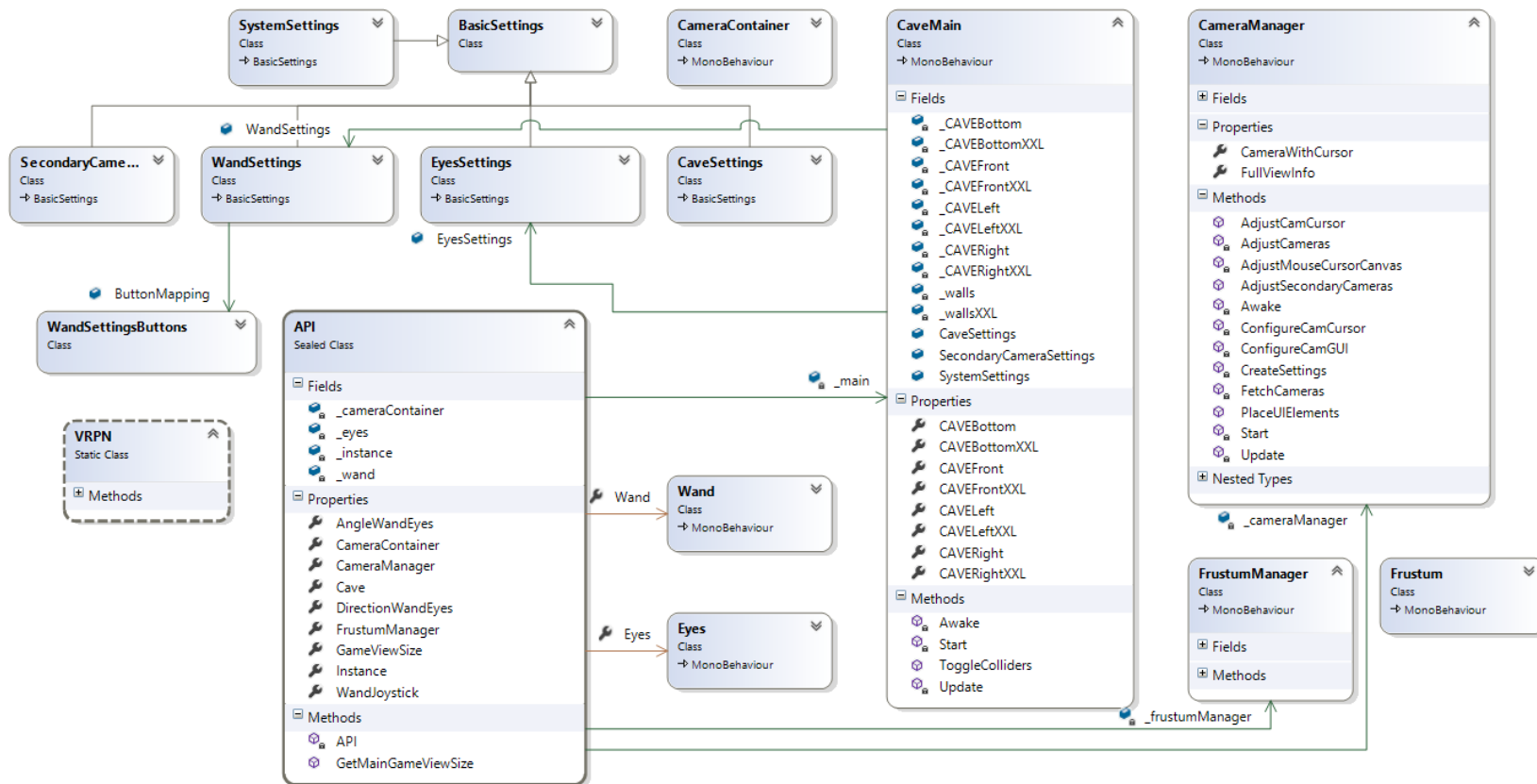


Abbildung 35: Unity Plugin, Klassendiagramm

Die wichtigsten Klassen und deren Methoden werden folgend kurz beschrieben.

- **CaveMain.cs**

Das ist der zentrale Knoten des Plugins. In der Unity-Hierarchie sind sämtliche Plugin-relevanten Objekte Child-Elemente dieser Instanz. Wichtige Management-Klassen werden während des Startvorgangs vom CaveMain instanziiert und die Geometrie des virtuellen Caves wird hier zusammengetragen. Der Zugriff auf die benutzerspezifischen Einstellungen, welche in Model-Klassen ausgelagert sind, erfolgt über diese Stelle. Je nach Einstellung, ob der CAVE zu Debug-Zwecken dargestellt werden soll, deaktiviert diese Klasse sämtliche Renderers.

Methode	Aufgabe
Awake()	Instanziiert CameraManager, FrustumManager und CameraContainer
Start()	<ul style="list-style-type: none"> • Erstellt eine Kollektion der Cave-Transforms • Deaktiviert auf Wunsch sämtliche Renderer • Deaktiviert die Collider des virtuellen Caves
Update()	Übernimmt die Position sowie Rotation der Hauptkamera.
ToggleColliders()	Schaltet die Collider der CAVE-Wände ein, aus.

Tabelle 2: CaveMain Methoden

Enum	Aufgabe
FrustumMode	Auswahl für CAVEXXL und GPP_Kooima. Steuert wie das Frustum berechnet wird

Tabelle 3: CaveMain Enums

- **Eyes.cs**

Die virtuellen Eyes interpretieren die Daten des Trackingsservers über VRPN und beachten die benutzerspezifischen Einstellungen des Plugins. Nach dieser Bearbeitung wird das in der Hierarchie liegende Eyes-Objekte aktualisiert, damit mittels API dessen Werte ausgelesen werden können.

Methode	Aufgabe
Start()	Speichert etliche über das API erreichbare Daten in privaten Variablen.
Update()	Führt in jedem Frame die Methoden „HandlePosition“ und „HandleRotation“ aus.
HandlePosition()	<ul style="list-style-type: none"> • Position von VRPN übernehmen • Smoothing durchführen • Blockierte Achsen zurücksetzen • Neue Position setzen
HandleRotation()	<ul style="list-style-type: none"> • Rotation von VRPN übernehmen • Smoothing durchführen • Blockierte Achsen zurücksetzen • Neue Rotation setzen

Tabelle 4: Eyes Methoden

- **Wand.cs**

Der Wand interpretiert analog den Eyes die Daten des Trackingsservers über VRPN und aktualisiert unter Berücksichtigung der benutzerspezifischen Einstellungen das Wand-Objekt. Der Wand ist aber noch für etliche weitere Aufgaben zuständig. Alle Inputs, die der Wand liefert, werden ebenfalls an dieser Stelle bearbeitet. Das umfasst das Setzen der Cursor-Position und dessen eventuell vorhandenen Cursor-Textur, die Bereitstellung der Joystick-Daten und die Input-Simulation der verschiedenen Buttons.

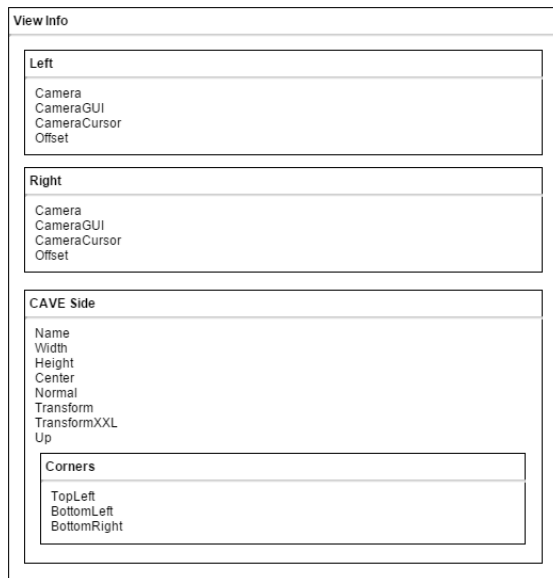
Methode	Aufgabe
Start()	Speichert etliche über das API erreichbare Daten in privaten Variablen und setzt, falls vom Benutzer gewünscht, eine spezifische Cursor-Textur.
Update()	Führt in jedem Frame die Methoden „HandlePosition“, „HandleRotation“, „HandleButtons“, „HandleJoystick“ und „SetCursor“ aus.
HandlePosition()	<ul style="list-style-type: none"> • Position von VRPN übernehmen • Smoothing durchführen • Blockierte Achsen zurücksetzen • Neue Position setzen
HandleRotation()	<ul style="list-style-type: none"> • Rotation von VRPN übernehmen • Smoothing durchführen • Blockierte Achsen zurücksetzen • Neue Rotation setzen
HandleButtons()	<ul style="list-style-type: none"> • Empfängt die Button-Inputs über VRPN • Simuliert die im Inspector zugewiesenen Inputs (Tastatur sowie Maus) • Stellt sicher, dass derselbe Input nicht versehentlich mehrmals hintereinander ausgeführt wird
HandleJoystick()	<ul style="list-style-type: none"> • Empfängt die Joystick-Inputs über VRPN • Führt die registrierten Delegates aus mit der aktuellen Joystick-Position
SetCursor()	<ul style="list-style-type: none"> • Raycast auf den virtuellen Cave • Leinwand ermitteln • Cursor-Kamera adaptieren • 2D-Position auf Screen ermitteln • Cursor-Position setzen • Cursor-Position des Duplikats setzen

Tabelle 5: Wand Methoden

- **CameraManager.cs**

Das Aufgabenspektrum des CameraManagers befasst sich mit allen Tasks, die in Verbindung mit einer Kamera stehen. Vorgängig wird hier das grundlegende Setting der multiplen Kamera-Konstruktion für das stereoskopische Sehen vorgenommen und alle relevanten Informationen zu den CAVE-Seiten werden in Structs gespeichert.

Jede Seite des CAVEs, also 4 Stück, enthält Daten über die Kameras je Auge, die Abmessungen der Leinwand, die Eckpunkte der Leinwand usw. Diese Daten finden während dem gesamten Prozess Verwendung, hauptsächlich jedoch bei der Frustumberechnung. Folgend sind die verwendeten Structs.



Quellcode 7: CameraManager DataStruct

Methode	Aufgabe
Awake()	Setzt CaveMain als parent.
Start()	Verhindert das Rendern von UI-Elementen auf der Hauptkamera und führt die Methoden „FetchCameras“, „CreateSettings“, „AdjustCameras“, „AdjustSecondaryCameras“ und „PlaceUIElements“ aus.
Update()	Ermittelt, auf welcher Kamera sich der Systemcursor befindet.
FetchCameras()	Speichert die Referenzen auf die verschiedenen Kameras.
CreateSettings()	Füllt die oben abgebildeten Structs mit Daten ab und speichert sie in einem Dictionary.
AdjustCameras()	Übernimmt die Grundeinstellungen der Hauptkamera, macht spezifische Einstellungen je Seite und positioniert die Viewports.
AdjustSecondaryCameras()	Konfiguriert die vorhandenen sekundären Kameras.
PlaceUIElements()	Konfiguriert die GUI-Kameras und platziert die Canvas.
AdjustCamCursor()	Die für den Cursor zuständigen Kameras werden je nach Cursorposition ausgerichtet und ein- / ausgeschaltet.

Tabelle 6: CameraManager Methoden

Enum	Aufgabe
CameraDepths	Steuert auf welchem Layer sich die Kamera befindet. Sodass bspws das GUI nicht durch das Spiel verdeckt wird.

Tabelle 7: CameraManager Enums

- **CameraContainer.cs**

Dieser Container beinhaltet alle dynamisch, vom UnityPlugin generierten Kameras, um sie zentral verschieben zu können. Zur Laufzeit wird geprüft, ob eine andere Kamera als Hauptkamera definiert wurde und verschiebt sich entsprechend dorthin.

Die Position CameraContainer-Objects ist direkt an die Position der Eyes gebunden. Und weil die Sensibilität, also in welchem Verhältnis sich die virtuellen Eyes zu den realen Eyes verschieben, eingestellt werden können, muss diese Berechnung hier geschehen. Beim Setzen der Position wird eine einfache Vektormultiplikation durchgeführt.

Methode	Aufgabe
Start()	Die Bewegungssensibilität wird zwischengespeichert.
Update()	<ul style="list-style-type: none"> • Prüfung, ob der Container noch an der Hauptkamera angehängt ist und eventueller Parentwechsel. • Setzen der Position basierend auf den Eyes sowie Sensibilität beachten.

Tabelle 8: CameraContainer Methoden

- **FrustumManager.cs**

Der FrustumManager sammelt die benötigten geometrischen CAVE-Daten und die dazugehörigen Kameras, um mittels Frustum-Klasse das auf die Kameras abzubildende Viewfrustum zu berechnen.

Methode	Aufgabe
Awake()	Setzt CaveMain als parent.
Update()	Nimmt die vom CameraManager zusammengetragenen Daten und bereitet sie für die Frustum-Berechnung auf.

Tabelle 9: FrustumManager Methoden

- **Frustum.cs**

Berechnet für die ihm angegebene Kamera das Viewfrustum, basierend auf der Eyes-Position, den drei Eckpunkten einer CAVE-Seite sowie der Near- und Farplane. Siehe dazu Kapitel 4.1

- **VRPN.cs**

Diese Klasse übernimmt die Schnittstelle zwischen C# (Unity) und C++ (VRPN). Alle Daten, die vom Trackingserver aufbereitet und übers VRPN-Protokoll verschickt werden, werden hier empfangen und in C# Datentypen umgewandelt, um die Verwendung zu vereinfachen.

- **API.cs**

Grundsätzlich kann der gesamte Sourcecode des Plugins eingesehen und verändert werden. Um die Verwendung jedoch zu vereinfachen und die eigene Anwendung applikationsspezifisch mit dem Plugin zu verknüpfen, werden gewisse Werte, Berechnungen und Objekte in der API zur Verfügung gestellt. Mittels Singleton-Pattern wird sichergestellt, dass die Verwaltung der besagten Properties zentral an einem Ort geschieht und dort abgegriffen werden können.

Der untenstehende Code der Klasse API zeigt, wie beispielsweise der Wand gesucht, referenziert und zurückgegeben wird. Ebenfalls werden gewisse Berechnungen, in diesem Beispiel der relative Winkel zwischen dem Wand und den Eyes, direkt berechnet und können abgefragt werden.


```

public sealed class API
{
    static readonly API _instance = new API();

    public Wand Wand
    {
        Get
        {
            if(_wand == null)
            {
                return _wand = GameObject.Find("WorldVizWand")
                    .GetComponent<Wand>();
            }

            return _wand;
        }
    }

    ...

    public Quaternion AngleWandEyes { get { return Quaternion
        .Inverse(Eyes.transform.rotation) * Wand.transform.rotation; } }

    public static API Instance
    {
        get
        {
            return _instance;
        }
    }

    ...
}

```

Quellcode 8: API

Der Zugriff auf das API erfolgt, weil es sich um ein Singleton-Pattern handelt, über die selber erstellte Instanz.

```

var angle = API.Instance.AngleWandEyes;

```

Quellcode 9: Zugriff über das API

8.3 Deployment

Das entwickelte Unity Plugin muss möglichst unkompliziert und rasch in die gewünschte Applikation integriert werden können. Um das zu erreichen, wird der gleiche Ansatz wie das Deployment über den integrierten Unity Asset Store gewählt. Dazu werden sämtliche Verzeichnisse und Dateien in eine .unitypackage-Datei gepackt und können in jedes beliebige Projekt importiert werden.

Um diese Bündelung der Dateien zu erreichen, gibt es in Unity den Befehl, ein Package zu exportieren.

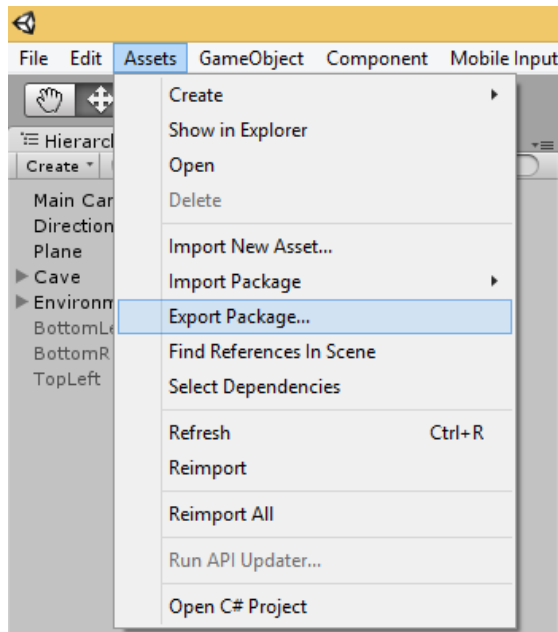


Abbildung 36: Unity Plugin, Export Package

Beim anschließenden Popupmenu alle Assets auswählen und den Export starten.

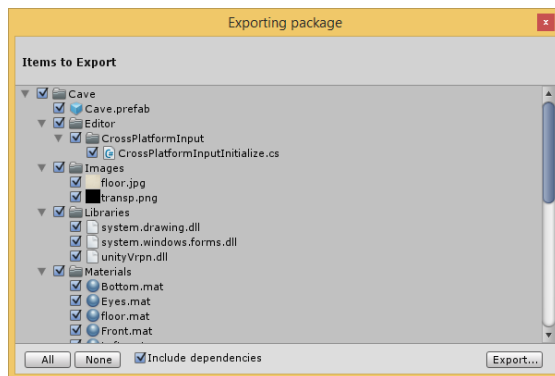


Abbildung 37: Unity Plugin, Export Package Selection

Die Verwendung des Plugins mit sämtlichen Abhängigkeiten erfolgt in wenigen Schritten:

1. Import des Plugins

Abhängig der Unity-Version kann der Import zu Problemen führen. Sind gewisse, bereits existierende Files in Verwendung, werden die Klassen / Prefabs / usw. nicht überschrieben, sondern lediglich hinzugefügt (CaveMain 1.cs, CaveMain 2.cs usw.). Darum empfiehlt sich, den Import manuell über den Explorer vorzunehmen und nicht die exportierte .unitypackage-Datei zu verwenden.

Kommentiert [DI9]: Hier noch der Fehler mit den Missing Tags erwähnen. Da diese im ProjectSettings gespeichert sind, und nicht im Asset

Als erstes muss das Unity-Projekt geschlossen werden, damit alle Dateien und Verzeichnisse über Schreibrechte verfügen. Falls bereits eine andere Version des Plugins ins Projekt integriert wurde, den Ordner „Cave“ im Assets-Verzeichnis löschen und die neue Version reinkopieren.

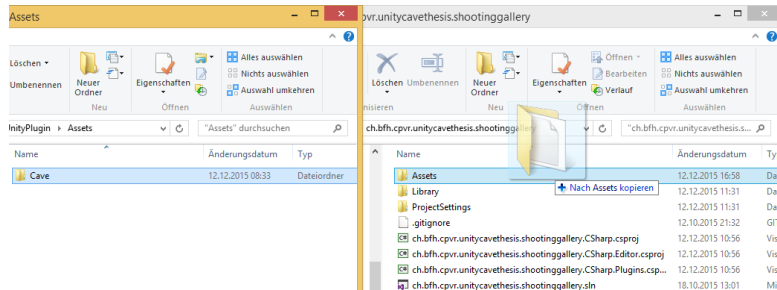


Abbildung 38: Kopieren des UnityPlugins

Alternativ kann der Asset Import-Mechanismus von Unity verwendet werden. Diese Methode, auch wenn sie elegant erscheinen mag und ursprünglich so angedacht war, führt leider oft zu einem Fehlverhalten und es wird an dieser Stelle abgeraten, das Asset auf diese Weise zu importieren.

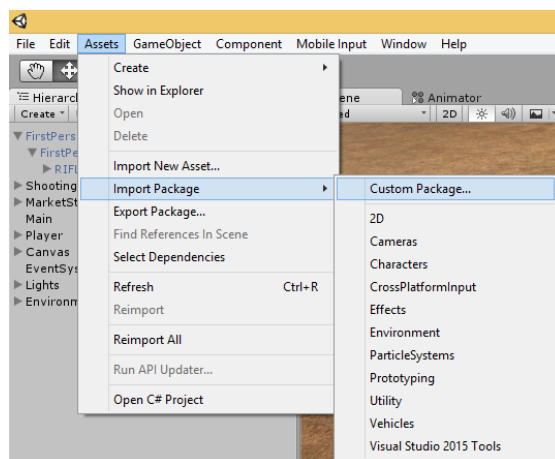


Abbildung 39: Unity Plugin, Import Package

Beim anschließenden Popupmenu sind alle Assets auszuwählen und mit „Import“ zu bestätigen.

2. Cave-Prefab

In einem zweiten Schritt gilt es, das Cave-Prefab, welches sich direkt im Verzeichnis „Cave“ befindet, an einer beliebigen Stelle in der Szene zu platzieren.

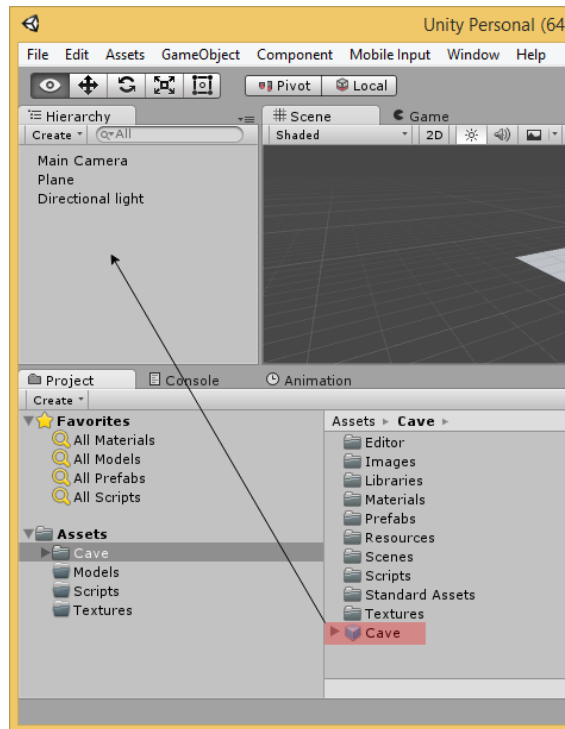


Abbildung 40: Hinzufügen Cave-Prefab

3. API Kompatibilitätslevel

Weil das UnityPlugin sich weiterer DLLs bedient, muss bei den Player-Settings das „Api Compatibility Level“ auf „.NET 2.0“ (ohne Subset) gesetzt werden. Damit wird die Verwendung externer DLLs ermöglicht.

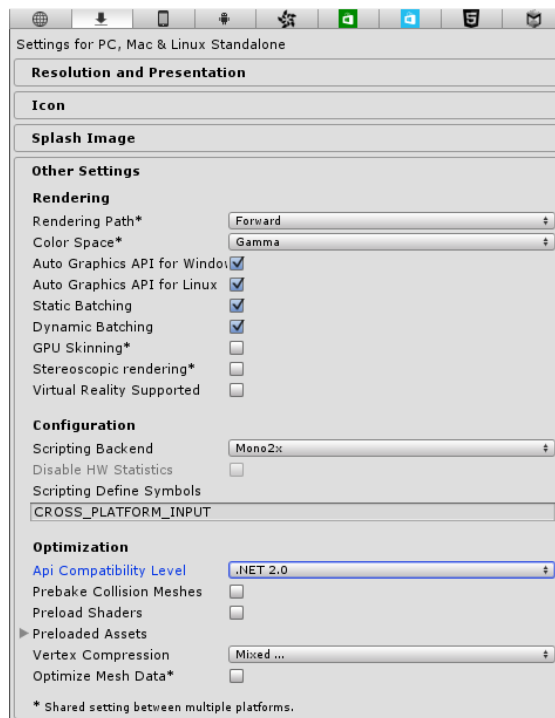


Abbildung 41: Player Settings, Api Compatibility Level

4. Plattform Einstellungen

Um sicherzustellen, dass die zusätzlichen DLLs für die Zielplattform exportiert werden, im Projekt zu den DLLs navigieren (Cave -> Libraries) und bei allen DLLs die entsprechenden Häkchen setzen.



Abbildung 42: Platform Settings

5. Einsatzbereit

Nach diesen einfachen Schritten wurde das Plugin erfolgreich in die Applikation eingebettet und kann verwendet werden.

8.4 Troubleshooting

- **Fehlende Verlinkung Prefabs**

Es besteht die Möglichkeit, dass die Verbindung zu den Prefabs vom CameraManager, FrustumManager und CameraContainer unter den System Settings beim Kopieren verloren geht. In diesem Fall direkt in Unity das entsprechende Prefab aus dem Ordner „Cave / Prefabs“ an das dafür vorgesehene Property hängen.

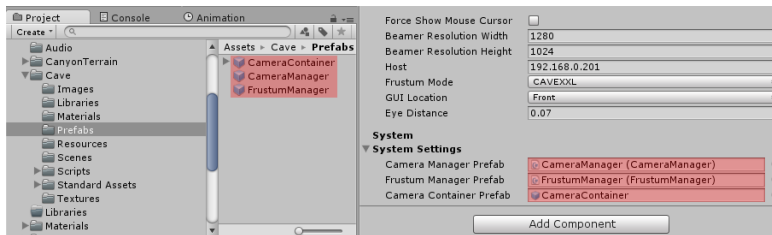


Abbildung 43: Zuweisung Prefabs

- **Falsche Architektur**

Falls beim Exportieren Fehler auftreten sollten, muss die Architektur möglicherweise auf 64bit angepasst werden.

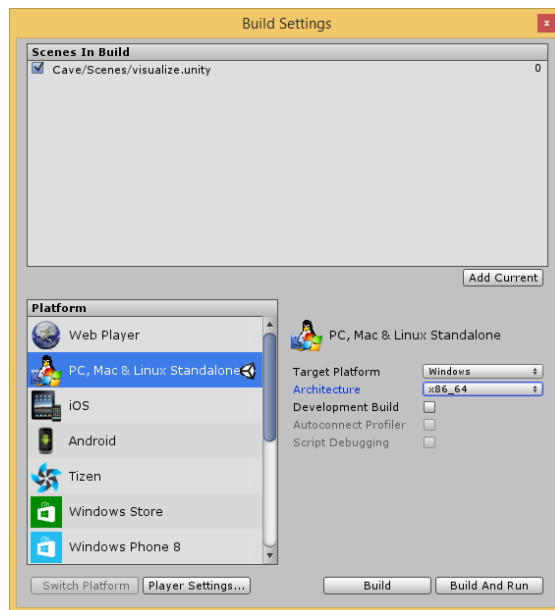


Abbildung 44: Export 64bit

8.5 Performance Verbesserungen

Um eine möglichst gute Performance zu erreichen, wurde bereits während der Entwicklung darauf geachtet, Scripts nicht unnötig auszuführen. Dennoch wurde zum Schluss der Code nochmals analysiert und geprüft, wo Verbesserungen durchgeführt werden können.

Im folgenden Profiling-Screenshot ist deutlich zu sehen, dass fast sämtliche Ressourcen nur noch für die Berechnung der Kameras verwendet werden. Alle anderen Aufgaben sind weitgehend optimiert und haben keinen markanten Einfluss auf den Spielfluss. Das relativ kostspielige Rendering aller Kameras lässt sich nicht umgehen und ist ein wichtiger Bestandteil des Plugins.

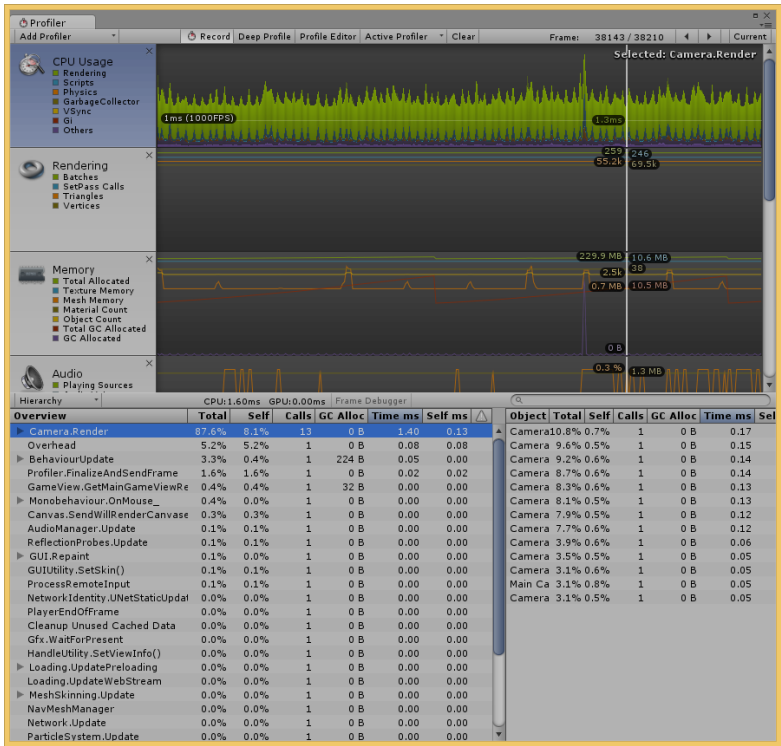


Abbildung 45: Performance Profiling

Die folgenden Codeänderungen wurden im Zuge der Performanceanalyse durchgeführt.

- **Komponenten suchen**
Bei etlichen Modulen bestehen Verbindungen zu anderen Objekten. Es wurde sichergestellt, dass sämtliche Referenzen nicht mehrmals neu geholt werden, sondern alle Verknüpfungen in der Startphase gemacht werden.
- **Deaktivieren Rendering Hauptkamera**
Weil das Plugin eigene Kameras erstellt und die Darstellung der ursprünglichen Kamera obsolet macht, indem die neuen Viewports den alten Viewport verdecken, wird das Rendering dieser Kamera nach der Konfigurationsphase deaktiviert.
- **UI-Kameras orthografische Projektion**
Weil die Berechnung einer orthografischen im Vergleich zu einer perspektivischen Projektion einfacher ist, wird die Projektion für alle UI-Kameras auf orthografisch gesetzt. Dies ist möglich, weil die Perspektive keinen Einfluss auf die Darstellung der 2D-UI-Elemente hat.
- **FixedUpdate statt Update**

Unity stellt eine Methode namens `FixedUpdate()` zur Verfügung, die im Vergleich zu `Update()` nicht so häufig wie möglich (also bei jedem berechneten Frame) ausgeführt wird, sondern nur zu fix definierten Zeitpunkten. Standardmässig sind das 30 Ausführungen pro Sekunde. Alle Berechnungen vom Wand, den Eyes und dem Viewfrustum werden in dieser `FixedUpdate`-Methode gemacht. Der Unterschied ist für den Benutzer nicht spürbar und es können somit etliche Berechnungen vermieden werden.

- **Wand Raycast**

Der Wand führt in gewissen Zeitabständen Raycasts aus, um den Punkt auf dem virtuellen CAVE, in welche er ausgerichtet ist, zu bestimmen. Die Distanz, wie weit dieser Raycast in die virtuelle Welt geschossen werden soll, kann bestimmt werden. Im Falle der CAVE-Schnittpunktsuche ist eine maximale Distanz von den räumlichen Gegebenheiten bereits gegeben und kann somit beschränkt werden.

- **Clipping Planes**

Wie weit, bzw. nah, sich die Clipping Planes befinden, wird von der Unity Applikation festgelegt und darf vom Plugin nicht beeinflusst werden. Jedoch für die UI-Kameras ist die maximale Distanz nur bis zum CAVE XXL und kann entsprechend verkleinert werden, um eine bessere Performance zu erreichen.

9 Warping

Bei einer stereoskopischen Projektion mit zwei unterschiedlichen Beamern muss für eine vollständige Immersion die Kalibrierung dieser zwei Output Geräte sehr genau sein. Aufgrund verschiedener Umstände (Wärmeausdehnung, Freiheitsgrade durch Gelenke und Spiegel) wäre es wünschenswert, den gerenderten Output noch zu warpen, um eine bessere Übereinstimmung beider Bilder auf einer Leinwand zu erreichen.

„Warping (von englisch warp = verformen, verzerren) von Bildern gehört in der Computergrafik zu den bildbasierten Techniken. Falls zu einem Bild die dazugehörigen Tiefenwerte existieren, ist es mittels der Warping-Gleichung möglich, das Bild von einem anderen Blickpunkt zu betrachten. Das Verfahren ist echtzeitfähig, bringt jedoch einige Artefakte wie beispielsweise Aufdeck- oder Verdeckungsfehler mit sich.“

(Quelle: https://de.wikipedia.org/wiki/Image_Warping)

In Unity ist dies durch einen Shader oder über die Projektionsmatrix der Kamera möglich. Da im CAVE die Projektionsmatrix bereits für die Frustumtransformation neu berechnet wird, empfiehlt sich hier die Verwendung des Shaders.

Im folgenden Beispiel ist ein einfacher Shader verwendet worden, um eine homographische Transformation der Kameraperspektive durchzuführen.

(Quelle: <https://github.com/chiragraman/Unity3DProjectionMapping>)

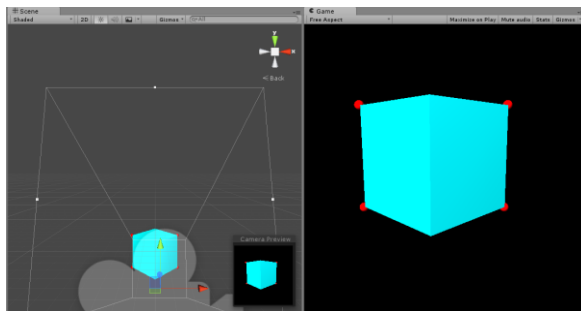


Abbildung 46: Initialposition Warpingbeispiel

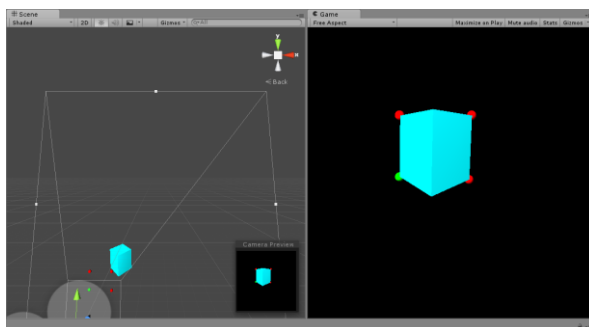


Abbildung 47: Verkleinerung Perspektive Warping

10 Demo Apps

10.1 Shooting Gallery



Abbildung 48: Shooting Gallery Ingame

Das Setting dieses Demospiels ist eine Schiessbude im Wilder Westen Stil, wie sie auf einem Jahrmarkt anzutreffen ist. Die Galerien mit den abzuschliessenden Zielen verteilen sich rund um den Spieler. Mit Hilfe des Head Trackings kann sich der Spieler in der gesamten Szenerie umschauen, Bewegungen ausführen und die Objekte aus verschiedenen Perspektiven betrachten. Das Wand-Device steuert das Gewehr, um die Zielobjekte anzuvisieren und abzuschliessen. Die Buttons des Walls werden gebraucht um das Gewehr abzufeuern.

- **Umgebung**
Die Landschaft ist ein Model aus dem Asset Store von Unity mit riesigen Ausmessungen. Um die abzuschliessenden Ziele platzieren zu können, Hindernisse zu schaffen und der Szenerie Leben einzuhauchen, wurden verschiedene Marktstände, Heukarren, Büsche und Zäune eingefügt.

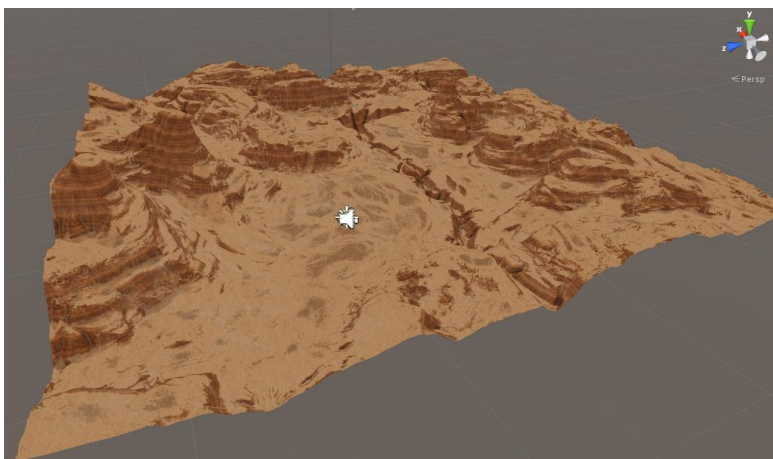


Abbildung 49: Desert Model, Quelle: Unity Asset Store

- **Gewehr**

Das Model des Gewehrs ist ebenfalls aus dem Unity Asset Store. Die Enten und Zielscheiben werden mit der Cursorposition anvisiert, welche durch das UnityPlugin vom Wand gesetzt wird.

Rotation

Das Gewehr dreht sich entsprechend dessen Position. Die Rotation wird zweierlei beeinflusst.

a) Rotation der Eyes

Basierend auf der Rotation der Eyes auf der y-Achse (Yaw) und der z-Achse (**Roll**) dreht sich auch das Gewehr im Spiel. Somit wird ermöglicht, dass sich der Benutzer des CAVes drehen kann und das Gewehr immer in seine Blickrichtung zielt. Neigt er den Kopf leicht auf eine Seite, übernimmt das Gewehr ebenfalls diese Manipulation rollt sich auf die Seite.

b) Relativer Winkel zwischen Eyes und Wand

Zusätzlich zu der Blickrichtung, welche mittels Eyes festgestellt wird, folgt das Gewehr der aktuellen Cursorposition. Dazu wird der relative Winkel zwischen dem Wand und den Eyes berechnet und darauf basierend erfolgt eine zusätzliche Rotation des Gewehrs.

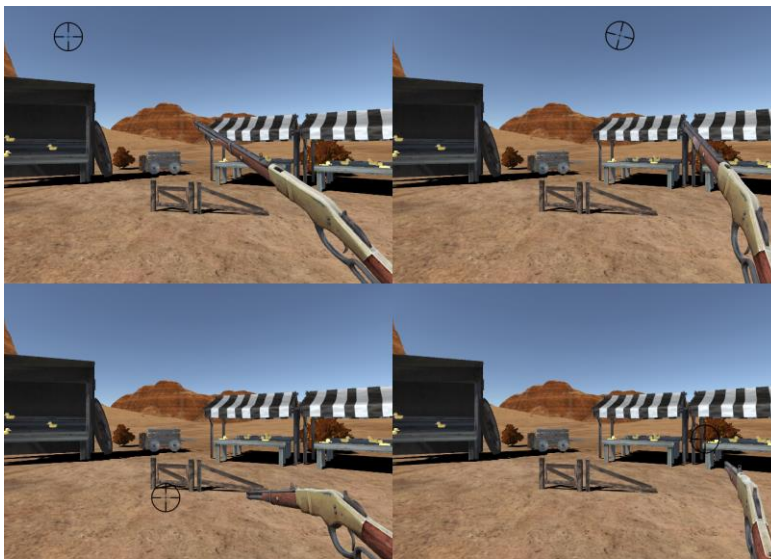


Abbildung 50: Shooting Gallery, Rotation des Gewehrs

Schuss

Dort wo sich der Cursor auf der 2D-Ebene befindet, wird auch präzise der Schuss in der 3D-Welt auftreffen. Dazu wird ein Raycast mit der Ausrichtung der Kamera an der Position des Cursors in die Umgebung geschossen und geschaut, welches Objekt als erstes im Wege steht.

Beim Auftreffen dieses virtuellen Schusses wird dem Ziel mitgeteilt, ob nun eine Interaktion erfolgen soll oder nicht. Zusätzlich wird ein Partikeleffekt, welcher den Einschuss verdeutlicht, an der getroffenen Stelle erzeugt. Ein weiterer Raucheffekt wird bei der Flinte direkt gezeigt.



Abbildung 51: Shooting Gallery, Rauch

Audio

Verschiedene Audiofiles wurden integriert, um das Spielerlebnis zu verbessern. Folgende Ereignisse provozieren einen Audioeffekt:

- Feuern des Gewehrs
- Treffen einer Ente
- Treffen der Zielscheibe

• Zielscheibe

Eines der beiden abzuschliessenden Ziele ist eine Holzkonstruktion mit drei Zielscheiben, welche sich zu zufälligen Zeitpunkten nach oben, bzw. nach unten klappen und somit angreifbar, bzw. nicht angreifbar werden.

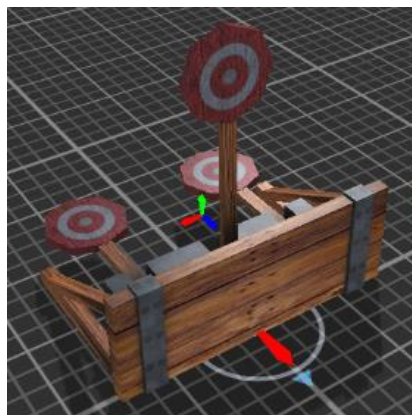


Abbildung 52: Shooting Gallery, Zielscheibe

Die Animationen sind im FBX-Model gespeichert und werden mittels einem AnimationController ausgelöst. Zu zufälligen Zeitpunkten wird eine der Show-States aktiviert um die Animation abzuspielen. Trifft während einer gewissen Zeitspanne kein Schuss die Zielscheibe, aktiviert sich der Hide-State und geht anschliessend zurück in den Idle-State. Im Gegenzug, trifft der Spieler auf die Zielscheibe, wird die Animation beim Hit-State abgespielt und es werden Punkte gutgeschrieben. Zudem wird als Audiofeedback ein entsprechender Sound gehört. Anschliessend fängt die Sequenz von vorne an.

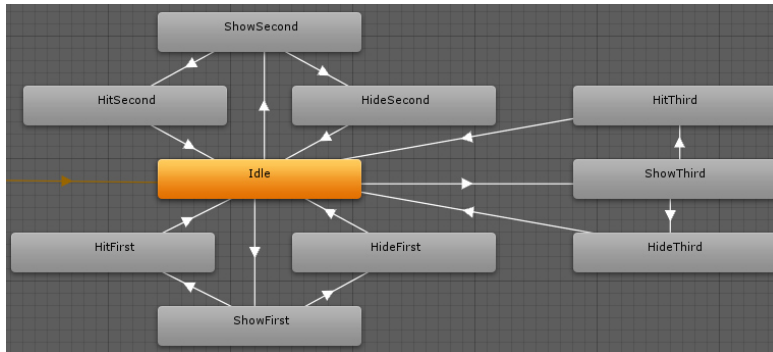


Abbildung 53: Shooting Gallery, Zielscheibe Animation Controller

- **Ente**

Das zweite abzuschliessende Objekt ist eine Gummiente, die sich auf einer festgelegten Bahn mit zufälliger Geschwindigkeit nach vorne und hinten bewegt.



Abbildung 54: Shooting Gallery, Ente

Am Anfang und Ende der Bahn befinden sich jeweils Trigger, die ausgelöst werden, sobald sich das Mesh der Ente damit überschneidet. In diesem Moment ändert sich die Bewegungsrichtung der Ente und eine neue Geschwindigkeit wird zufällig zwischen einem definierten Bereich gewählt.

Bei einem Treffer wird der Winkel zwischen der aktuellen Kamera und der Ente berechnet und entsprechend ein Impuls auf die Ente angewandt, damit sie in die korrekte Richtung davonfliegt. Sobald die Ente keinen Kontakt mit der Oberfläche mehr hat auf der sie sich bewegt, wird ein Event losgeschickt um nach einer gewissen Zeitspanne die Position und Rotation wiederherzustellen und Punkte gutzuschreiben. Bei einem Treffen ist zudem ein Audiofeedback (Quaken) zu hören.

Weil das gefundene Model der Ente für die Anwendungszwecke viel zu detailliert war, musste aus Performancegründen mit Blender eine Reduktion der Faces erfolgen. Von ehemals 7160 sind noch 1288 Faces übrig geblieben. Dank der Textur und der Distanz, die zwischen dem Spieler und der Ente liegt, fällt der Unterschied nicht auf.

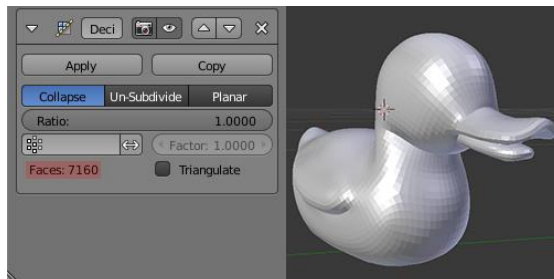


Abbildung 55: Shooting Gallery, Ente mit vielen Details

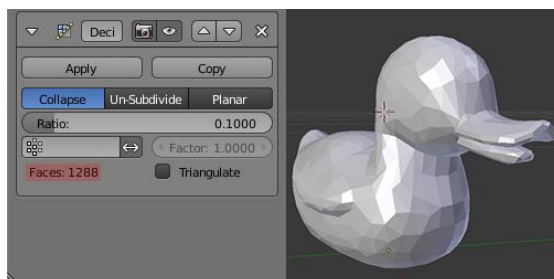


Abbildung 56: Shooting Gallery, Ente mit reduzierten Details

- **Punktesystem**

Nebst der bereits verstrichenen Zeit ist in einer Ecke als UI-Element die erspielte Punktzahl sichtbar. Die beiden abzuschliessenden Ziele (Ente und Zielscheibe) stellen öffentliche, statische Delegates (Events) zur Verfügung, mittels denen andere Objekte Methoden registrieren können. Wird also beispielsweise eine beliebige Ente abgeschossen, wird die Methode aufgerufen.

Die Klasse, welche für die Punkteberechnung und -darstellung zuständig ist, registriert entsprechend eine Methode, welche als Parameter die erspielten Punkte erhält. Somit kann dort sauber Buch geführt werden über den aktuellen Punktestand und ihn als GUI-Element anzeigen.

Duck.cs

```
public delegate void DelegateHit(int points);
public static event DelegateHit OnHit;

public void Hit()
{
    if (!_alreadyHit)
    {
        _alreadyHit = true;
        OnHit(POINTS);
    }
}
```

UIPoints.cs

```
Duck.OnHit += OnHit;

void OnHit(int points)
{
    _points += points;
    _text.text = "Punktzahl: " + _points.ToString();
}
```

Quellcode 10: Shooting Gallery, Ente und Punkte

10.2 Model-Viewer

Da der CAVE nicht nur zum Spielen gedacht ist, zeigt diese zweite Demo noch ein weiteres Anwendungsgebiet. Es handelt sich um mehrere Operationssäle, welche begangen werden können.

Die gesamte Umgebung ist statisch, einige Personen haben jedoch vordefinierte Animationen.

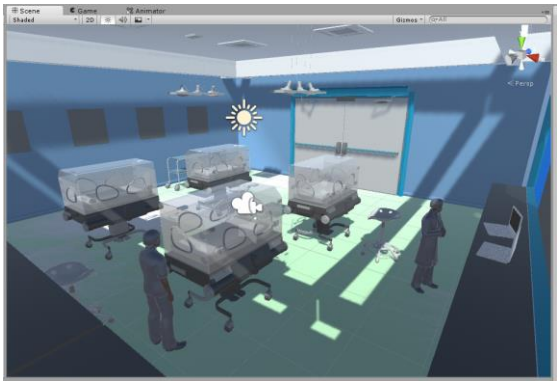


Abbildung 57: Model-Viewer OP Raum 1

• Modelle

Es werden Modelle von Dosch verwendet, welche bereits im Besitz der BFH sind.

(Quelle: <http://www.doschdesign.com/>)

Es folgt eine Auflistung der verfügbaren Personen und Räume. Auf die Detailliste aller Equipments wird hier Verzichtet. Weitere Informationen dazu sind im Anhang.

Person	Infos	
Surgeon	Person a1	19 Actions, 2 Sounds
SurgeonAssistant	Person a2	4 Actions
TOA	Person a3	6 Actions
TOA	Person a4	4 Actions
OPattendant	Person a5	8 Actions
Anesthetist	Person a6	7 Actions

Tabelle 10: Personen Modelviewer

Raum	Infos	
MedicalRoom01	CT Raum mit Kontrollraum	
MedicalRoom02	Zahnarzt	
MedicalRoom03	Zahnarzt schlicht	
MedicalRoom04	Bettzimmer	
MedicalRoom05	Babystation	
MedicalRoom06	Zahnarzt mit Vorzimmer	
MedicalRoom07	Patientendoppelzimmer	
MedicalRoom08	Quadratisches Zahnarztzimmer	
MedicalRoom09	Psychiater	
MedicalRoom10	Fitnessstudio	

Tabelle 11: Räume Modelviewer

• Szenen

Für jeden verfügbaren Raum ist eine vordefinierte Szene erstellt, in welcher mit dem WAND navigiert werden kann.

- **Hauptmenu**

Die Szenen können über ein statisches Menu geladen werden.

10.3 App Dritte Partei

Dieser Anwendungsfall soll aufzeigen, dass nicht nur konkret auf das umgesetzte Unity Plugin hin erarbeitete Applikationen problemlos verwendet werden können. Ausgewählt wurde das mit Unity 5 ausgelieferte Beispielprojekt namens „Standard Assets Example Project“.

Die Integrierung des Plugins verlief einwandfrei und das Spiel konnte erfolgreich in 3D im CAVE gespielt werden. Alle Elemente, zum Beispiel die Anzeige der UI Elemente auf einem spezifischen Screen, verursachen keinerlei Schwierigkeiten.

Der nachfolgende Screenshot zeigt die Aufteilung der Viewports nach dem Aktivieren des Plugins.

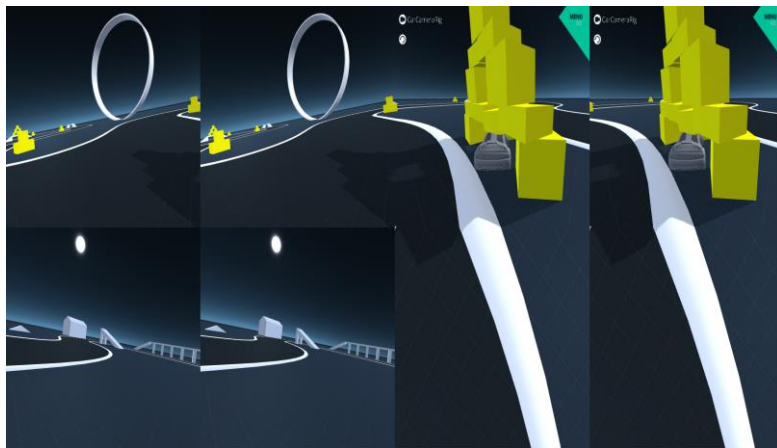


Abbildung 58: Standard Assets Example Project mit Unity Plugin

11 Testing

Dieses Kapitel beinhaltet die Tests der Prototypen, welche für verschiedene Module erstellt wurden, das Testen des Plugins mit der eigenen Testszene, sowie Performancetests der gesamten Anwendung.

11.1 PPT-Studio und VRPN

Die Funktionalität des internen VRPN Servers des PPT Studios kann mit einem einfachen Konsolclient überprüft werden, welcher die Trackerdaten ausgibt. Das Programm «vrpn_print_devices.exe» von vrpn.org liefert diese Informationen.

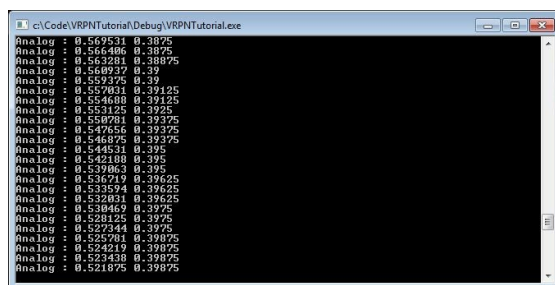


Abbildung 59: VRPN Debug Ausgabe

(Quelle: <http://www.cs.unc.edu/Research/vrpn/index.html>)

11.2 Wrapping VRPN in C# und Unity

Die Funktionalität der kompilierten DLL des VRPN Projekts sowie die .net Wrapperklasse wurde Anhand eines kleinen Unity Testprojektes überprüft. Dies verwendete die erhaltenen Trackerdaten direkt als Positions- und Rotationsangaben der Kamera. Weiter ist eine Plane im Raum sichtbar, welche die Ausrichtung im Unity bekanntgibt, damit dies dann mit der realen Trackingposition überprüft werden kann.

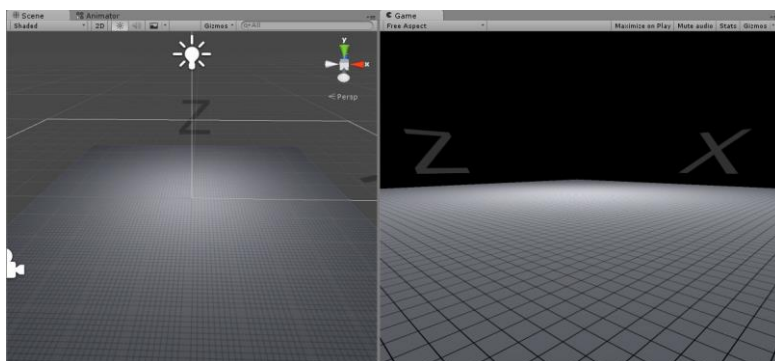


Abbildung 60: VRPN Prototyp

```

public class Tracker : MonoBehaviour
{
    public string host = "localhost";
    public string obj;
    public int channel = 0;

    public bool trackPosition = true;
    public bool trackRotation = true;

    // Update is called once per frame
    void Update()
    {
        if (trackPosition)
        {
            transform.position = VRPN.vrpnTrackerPos(obj + "@" + host, channel);
        }
        if (trackRotation)
        {
            transform.rotation = VRPN.vrpnTrackerQuat(obj + "@" + host, channel);
        }
    }
}

```

Quellcode 11: VRPN Prototyp

11.3 UnityPlugin Projekt als Debugger

Das Unity-Plugin kommt mit einer Testszene, mit welcher folgende Elemente getestet werden können:

- Stereoskopie Einstellungen der Kameras
- Sekundäre Kameras
- GUI Elemente
- Mousecursor
- Verhalten der Frustumtransformation bei Bewegungen im CAVE.

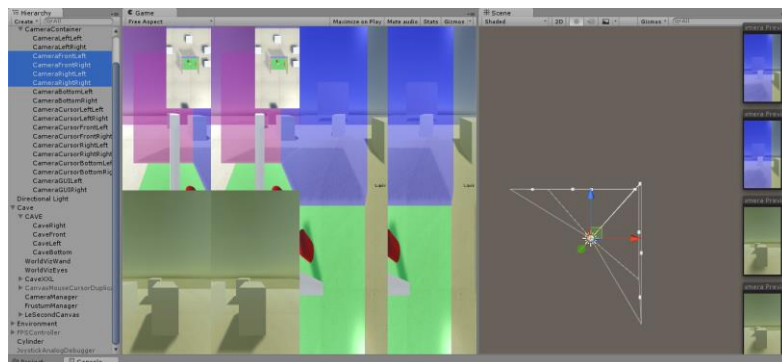


Abbildung 61: CAVE Testszene

Auf Abbildung 60 sind die Frustums von vier Kameras sichtbar, dass die Aufspannung der Projektion korrekt ist. Eine sekundäre Kamera ist auf die linke Projektionsseite gemappt, sodass auch dieser Output direkt im Plugin überprüft werden kann. Zusätzlich können die Frustums noch über Gizmos von Unity angezeigt werden.

Die Simulationen der Bewegung übernimmt ein Debugscript, welches bei Bedarf auf den Komponenten aktiviert werden kann. So führt die Komponente eine Bewegung im Raum durch zusammen mit einer Rotation.

```

void Update()
{
    TimeCounter += Time.deltaTime * speed;
    float x = Mathf.Abs(Mathf.Cos(TimeCounter) * width);
    float y = Mathf.Abs(Mathf.Sin(TimeCounter) * height);
    float z = 0.1f;

    transform.position = new Vector3(x, y, z);
    transform.rotation = new Quaternion(x, y, z, 1f);
}

```

Quellcode 12: Debugmover

11.4 Performance Tests Unity Server

Die Graphik Performance des Unity Servers ist von zentraler Bedeutung. Falls das Plugin durch die zusätzlichen Kameras und Scripts zu Ressourcenintensiv wird, sollte eine Ressourcenanalyse durchgeführt werden. Aktuelle Tests mit aufwändigen Berechnungen zeigen folgendes Bild:

Kommentiert [DI10]: Muss überarbeitet werden falls neue Screenshots bessere Werte liefern, oder sonst löschen

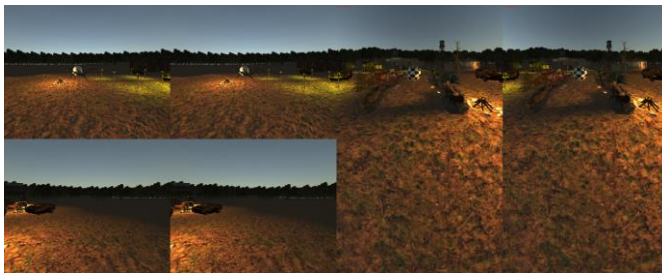


Abbildung 62: Bowlingdemo AdvGameDev

00.5 ms (33 fps)

Abbildung 63: FPS Bowlingdemo

Gemessen wurden konstant 33-35 Frames. Für die Ressourcenanalyse siehe Kapitel 11.5. Weniger Ressourcenintensive Anwendungen erreichten deutlich höhere Frameraten.

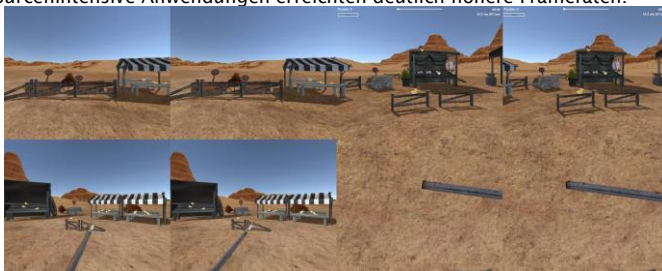


Abbildung 64: Shootinggallery Demospiel

00:09
10.3 ms (97 fps)

Abbildung 65: FPS Shootinggallery

12 Abbildungsverzeichnis

Abbildung 1: Prototyp 1 Projekt 2	5
Abbildung 2: Prototyp 2 Projekt 2	6
Abbildung 3: CAVE BFH	6
Abbildung 4: Infrastruktur CAVE	8
Abbildung 5: PPT Eyes, Quelle: www.worldviz.com	9
Abbildung 6: PPT Wand, Quelle: www.worldviz.com	9
Abbildung 7: Gamepad, Quelle: www.androidrundown.com	10
Abbildung 8: Übersicht der Komponenten	12
Abbildung 9: Sequenzdiagramm	14
Abbildung 10: Einstellungen Augendistanz	15
Abbildung 11: Custom Cursor	16
Abbildung 12: Minimap	16
Abbildung 13: Mosaic Setting Unity Server	18
Abbildung 14: Frustum, Quelle: www.stackoverflow.com	19
Abbildung 15: UnityPlugin Frustum 1	20
Abbildung 16: UnityPlugin Frustum 2	20
Abbildung 17: Vergleich FoV 120 & 60 Grad	21
Abbildung 18: UnityPlugin Frustum GGM	21
Abbildung 19: Frustum Mode	22
Abbildung 20: Frustum CAVE XXL FoV Anpassung	22
Abbildung 21: Unity Plugin, virtueller CAVE	23
Abbildung 22: Virtueller Wand und Eyes	24
Abbildung 23: Unity Plugin, Schnittpunkt Wand / CAVE	25
Abbildung 24: Zuordnung Schnittpunkt CAVE und Unity Server	26
Abbildung 25: Wand Button Zuordnung	26
Abbildung 26: Inputmanager Unity	28
Abbildung 27: Beispielverbindung VRPN	29
Abbildung 28: VRPN Wrapper C#	30
Abbildung 29: 1€ Smoothing Vergleich	31
Abbildung 30: Unity Plugin, Settings Wand	32
Abbildung 31: Unity Plugin, Settings Eyes	33
Abbildung 32: Unity Plugin, Settings Sekundäre Kameras	34
Abbildung 33: Unity Plugin, Settings Cave	34
Abbildung 34: Unity Plugin, Settings System	34
Abbildung 35: Unity Plugin, Klassendiagramm	36
Abbildung 36: Unity Plugin, Export Package	42
Abbildung 37: Unity Plugin, Export Package Selection	42
Abbildung 38: Kopieren des UnityPlugins	43
Abbildung 39: Unity Plugin, Import Package	43
Abbildung 40: Hinzufügen Cave-Prefab	44
Abbildung 41: Player Settings, Api Compatibility Level	45
Abbildung 42: Platform Settings	45
Abbildung 43: Zuweisung Prefabs	46
Abbildung 44: Export 64bit	46

Abbildung 45: Performance Profiling	47
Abbildung 46: Initialposition Warpingbeispiel	49
Abbildung 47: Verkleinerung Perspketive Warping	49
Abbildung 48: Shooting Gallery Ingame	50
Abbildung 49: Desert Model, Quelle: Unity Asset Store	51
Abbildung 50: Shooting Gallery, Rotation des Gewehrs	51
Abbildung 51: Shooting Gallery, Rauch	52
Abbildung 52: Shooting Gallery, Zielscheibe	52
Abbildung 53: Shooting Gallery, Zielscheibe Animation Controller	53
Abbildung 54: Shooting Gallery, Ente	53
Abbildung 55: Shooting Gallery, Ente mit vielen Details	54
Abbildung 56: Shooting Gallery, Ente mit reduzierten Details	54
Abbildung 57: Model-Viewer OP Raum 1	56
Abbildung 58: Standard Assets Example Project mit Unity Plugin	57
Abbildung 58: VRPN Debug Ausgabe	58
Abbildung 59: VRPN Prototyp	58
Abbildung 60: CAVE Testszene	59
Abbildung 61: Bowlingdemo AdvGameDev	60
Abbildung 62: FPS Bowlingdemo	60
Abbildung 63: Shootinggallery Demospiel	60
Abbildung 64: FPS Shootinggallery	60

13Tabellenverzeichnis

Tabelle 1: Mosaic Setting schematisch	17
Tabelle 2: CaveMain Methoden	37
Tabelle 3: CaveMain Enums	37
Tabelle 4: Eyes Methoden	37
Tabelle 5: Wand Methoden	38
Tabelle 6: CameraManager Methoden	39
Tabelle 7: CameraManager Enums	39
Tabelle 8: CameraContainer Methoden	40
Tabelle 9: FrustumManager Methoden	40
Tabelle 10: Personen Modelviewer	56
Tabelle 11: Räume Modelviewer	56

14Quellcodeverzeichnis

Quellcode 1: Kamerainstanziierung	15
Quellcode 2: Zugriff über das API	24
Quellcode 3: Raycast	25
Quellcode 4: Unity Tastaturabfrage	27
Quellcode 5: Joystick Handling	27

Quellcode 6: VRPN-Positionshandling	30
Quellcode 7: CameraManager DataStruct	39
Quellcode 8: API	41
Quellcode 9: Zugriff über das API	41
Quellcode 10: Shooting Gallery, Ente und Punkte	55
Quellcode 11: VRPN Prototyp	59
Quellcode 12: Debugmover	60

15Glossar

Todo

16Literaturverzeichnis

todo

17Anhang

Todo

18Selbständigkeitserklärung

Ich bestätige, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der im Literaturverzeichnis angegebenen Quellen und Hilfsmittel angefertigt habe. Sämtliche Textstellen, die nicht von mir stammen, sind als Zitate gekennzeichnet und mit dem genauen Hinweis auf ihre Herkunft versehen.

Ort, Datum:

Unterschrift: