

Projektdokumentation für das Seminar

Entwicklung von VR-Lehr-/Lernanwendungen mit Unity

# Upload und Wiedergabe von Audiodateien

Universität Potsdam, Institut für Informatik und Computational Science,

Lehrstuhl für Komplexe Multimediale Anwendungssysteme

Alexander Gayko, Matrikelnummer 743211

## Inhaltsverzeichnis

Thema:.....	1
Motivation .....	1
Projektziel .....	1
Bestandteile.....	1
Technischer Unterbau .....	1
Exkurs: IPC .....	2
Server .....	2
Kommunikation .....	2
Webseite .....	2
Architektur.....	3
Nicht nur Töne.....	3
Befehle geben.....	3
Farbauswahl .....	4
Animationsauswahl .....	4
Audioauswahl .....	4
Unity-Assets.....	6
Verweis zum Server .....	6
Unity-Architektur.....	7
Ein konkretes Feature.....	7
Konfiguration in untiy.....	7
NuGet aber nicht!.....	8
Fehlerbehandlung? Fehlanzeige.....	8
Webseite und - Server .....	8
Installation.....	9
Anhang .....	i
Weitere Arbeiten / TODOs .....	i
API-Referenz Unity-Assets.....	ii
Glossar .....	v
Bilder .....	vii

## Thema:

In diesem Thema ist in Unity ein Avatar zu modellieren, der auf die Zusendung von **Audio-Dateien** (z.B. MP3s) von **außerhalb** der Unity-Anwendung wartet. Die Unity-Anwendung müsste also eine Schnittstelle für den Upload bzw. den Empfang von Audio-Dateien bereithalten. Zudem gehört die Implementierung einer **Webseite** für die Auswahl (vom lokalen Rechner) und den **Unity-Upload** der Audiodatei zum Umfang dieses Themas.

## Motivation

Im Kontext des wissenschaftlichen Arbeitens, gerade bei VR-Projekten ist es oft notwendig, Probanden durch VR-Umgebungen zu leiten bzw. sie zu begleiten. Dies stellt eine Herausforderung dar, da das Ziel einer VR-Umgebung in der Regel ist, die Präsenz des Nutzers in der Umgebung zu maximieren. Entsprechend sind die beiden wichtigsten Sinne für effiziente Kommunikation, Sehen und Hören, durch das HMD blockiert (um möglichst hohe Präsenz zu erzeugen), und stehen nicht direkt als Kanal für Anleitungen zur Verfügung.

Abhilfe ist hier, mittels des Gerätes (PC, Laptop, ...), auf dem die VR-Umgebung ausgeführt ist, „live“ in diese hinein zu sprechen, indem ein weiterer Audiokanal mit den Anweisungen des Anleitenden zur Verfügung gestellt wird. Da die technische Entwicklung immer mehr zu unabhängigen HMDs ohne Companion-PC geht, steht dieser Kanal in absehbarer Zukunft nicht mehr direkt zur Verfügung.

## Projektziel

Ziel dieses Projektes ist daher, prototypisch eine Möglichkeit zu schaffen, von außen Töne in eine VR-Umgebung einzubringen, die beispielsweise vor-aufgenommene Anweisungen sein können. Ich schließe an dieser Stelle das Einbringen von live-Audiosignalen (z.B. direkt aus dem Mikrofon) aus; dies muss Gegenstand weiterer Arbeiten sein.

Entwickelt werden soll ein einfach einzurichtendes optionales Feature, dass keinen großen zusätzlichen Programmieraufwand mit sich bringen darf. Ferner muss es eine Steuerungsmöglichkeit geben für die Audio-Sequenzen. Hierfür wird eine Webseite zur Verfügung gestellt, mit der ein Betreuer Audiodateien auswählen und in der VR-Umgebung abspielen kann.

## Bestandteile

Das Projekt besteht entsprechend aus zwei Bestandteilen: Eine Reihe von Assets, die der VR-Umgebung hinzugefügt werden müssen, sowie ein Webserver, der die Bedienoberfläche für die Betreuer hostet.

Als Demonstration für die im Projekt erarbeiteten Artefakte wird zudem eine simple Beispielanwendung zur Verfügung gestellt, mit der man die Interaktion ausprobieren kann. Diese Anwendung ist nicht als VR-Umgebung konzipiert, sondern eine einfache Szene mit einem Avatar, der aus der 3rd-Person-Perspektive betrachtet wird, und sich in der Nähe einer Audioquelle befindet. Diese Audioquelle spielt die Audiosequenzen ab, die der Betreuer hochlädt, und der Benutzer der Demoanwendung hört Töne.

## Technischer Unterbau

Als technische Plattform für VR-Umgebungen wird am Lehrstuhl und auch im Rahmen des Seminars Unity3d verwendet, entsprechend handelt es sich bei den Assets um in C# geschriebene Klassen sowie

deren Abhängigkeiten in Form von Assemblies. Diese Assets können in bestehende Unity-Projekte eingefügt werden. Am Beispiel des Demoprojekts wird genau das gezeigt.

### Exkurs: IPC

Für das Einbringen von Audiosignalen von außerhalb in eine Unity-Anwendung bieten sich mehrere Methoden an. Das letztendliche Ziel ist, den laufenden Anwendungsprozess dazu zu bringen, eine Audiodatei einzulesen und abzuspielen. Traditionell würde man direkte Botschaften an die Message Pump des Prozesses verwenden, beispielsweise Windows-Botschaften. Diese sind leider nicht weder plattform- noch geräteunabhängig. RPC-Mechanismen wie DCOM oder .net Remoting oder WCF wären eine weitere Variante, hierbei wäre zwar Geräteunabhängigkeit gegeben (eine Konsequenz aus der Motivation für das Projekt), allerdings blieben wir nach wie vor plattformabhängig (da .net Remoting leider nicht in .net Core implementiert ist).

Für den einfachen Fall einer Anwendung, die auf externe Signale reagieren soll, verwende ich auch gern den Named Pipes -IPC-Mechanismus und mehrere Instanzen des Prozesses. Hierbei würde eine neue Instanz der Anwendung gestartet mit dem Parameter, der in unserem Fall die abzuspielende Audiodatei bezeichnet. Dieser neue Prozess prüft dann, ob auf dem System schon ein weiterer Prozess derselben Anwendung (der „Server“) läuft, und verbindet sich als Client an dessen Named Pipe, und gibt das korrekte Abspielen-Kommando. Zwar gibt es die Möglichkeit, auf die Named Pipes anderer Rechner zuzugreifen, allerdings ist die Plattformunabhängigkeit hierbei ebenfalls fraglich, sowie einige Aspekte der Robustheit (wie sichere ich meine Kommunikation ab) müssten selbst gelöst werden.

### Server

Da ferner die Aufgabenstellung selbst eine Webseite erfordert, bietet sich für die Implementierung ein Socket-basierter und damit plattform- und geräteunabhängiger Client-Server – Ansatz, hier für HTTP, direkt an. Dieser wurde auch für die Implementierung umgesetzt.

Das letztendlich erstellte Unity-Projekt sowie die Webseite zur Steuerung bilden die beiden Benutzerschnittstellen für die Audiosteuerungstechnik. Damit die Steuerung auf Prozess- und Maschinenebene von der VR-Anwendung unabhängig ist, wird die Webseite von einem eigenen Webserver-Prozess gehostet. Dieser dient gleichzeitig als Kommunikationszentrale.

### Kommunikation

Als konkretes Kommunikationsprotokoll für die Steuersignale habe ich SignalR gewählt – eine von Microsoft erstellte und inzwischen open source gestellte Protokoll-Abstraktionsschicht für die Echtzeitkommunikation von Webanwendungen, für die es .net Implementierungen gibt. Je nach Verfügbarkeit verwendet SignalR WebSockets, Long Polling oder andere Möglichkeiten, auf Clientseite Nachrichten vom Server zu empfangen. Zudem bietet es verschlüsselte Kommunikation, eindeutige Identifizierung der Endpunkte, Gruppenverwaltung mit Single-, Multi- sowie Broadcasts und die Möglichkeit, streng typisierte Nachrichten zu senden. Hierbei werden lebende Objekte serialisiert, transportiert, und auf der anderen Seite wieder deserialisiert und weiterverwendet. Dies gilt auch für CLR-Objekte und ermöglicht so effiziente Wiederverwendung auf Assemblyebene.

### Webseite

Um die Webseite darzustellen, habe ich mich für das momentan in Entwicklung befindliche SPA-Framework Blazor entschieden, da ich sowieso mit diesem experimentieren wollte, und das Projekt dazu eine gute Gelegenheit bot. Hiermit wird ermöglicht, weitgehend ohne JavaScript Webentwicklung zu betreiben, indem die erstellte Webanwendung als .net Assembly kompiliert und von der in WebAssembly ausgeführten Mono-Laufzeitumgebung im Browser geladen und ausgeführt wird.

## Architektur

Die Webseite sowie der Unity-Client verbinden sich als Clients mit dem SignalR-Hub, der serverseitigen Kommunikationszentrale. Sie können daraufhin Nachrichten an den Hub senden und von diesem empfangen. Für den Zweck dieses Projektes genügt es, einfache Prüfungen in den Hub einzubauen und diesen ansonsten als Broadcaster fungieren zu lassen, und die Nachrichten, die von dem jeweiligen Client nicht verstanden werden, einfach zu ignorieren. Um ein fertiges Produkt zu bauen, sollte der Server natürlich nur den Clients Nachrichten senden, die die entsprechende Verarbeitungsrolle innehaben. Sprich Audiosteuerungsnachrichten nur an den Unity-Client weiterleiten und Kontrollnachrichten nur von der Webseite akzeptieren.

Die serverseitige Modellierung von Zustand ist für dieses Projekt ebenso wenig notwendig, denn es soll zunächst ein einzelner Unity-Client gesteuert werden, und wir können davon ausgehen, dass ein gesendeter Befehl zu korrektem Verhalten des Unity-Clients führt (bzw. das testen), sodass die für den Befehl ausgeführte Änderung in der GUI der Webseite dem Zustand des Unity-Clients entspricht.

Solange es nur eine Steuerungsw Webseite gibt, gilt dies auch, wenn man mehrere Unity-Clients steuert; somit ermöglicht unser Ansatz schon mal rudimentäre Multiplayer-Funktionalität, z.B. ein gemeinsames VR-Erlebnis, dass dennoch dynamisch von außen kontrolliert werden kann, zumindest was die Soundkulisse angeht.

## Nicht nur Töne...

Und an dieser Stelle stellt sich heraus, dass der Mechanismus nicht nur in der Lage ist, Audio-Kommandos („Lade Datei xxx, mute, stop, ...“) zu senden, sondern ganz allgemein beliebige Befehle.

Wir bekommen quasi eine Fernsteuerungsmöglichkeit für nahezu beliebige Aspekte einer laufenden Unity-Anwendung, vorausgesetzt, die entsprechenden Anbindungen sind korrekt konfiguriert.

Um dies vorzuführen, habe ich die Demoanwendung sowie die unity-Assets und die Webseite auf drei steuerbare Aspekte erweitert: Farbauswahl, Animationsauswahl sowie - wie gefordert - Audioauswahl.

## Befehle geben...

SignalR unterstützt das Senden und Empfangen beliebiger Nachrichten, und Nachrichten können auf unterschiedliche Weise typisiert sein, damit ihre Semantik eindeutig wird. Der bevorzugte Ansatz wäre, Schnittstellen zu erzeugen, die die möglichen Nachrichten als Methoden deklarieren, und sich auf die stark typisierten Hubs zu verlassen, und auf diese Weise ausschließlich sinnvolle Kommunikation zuzulassen. Die Schnittstellen könnten zusätzlich noch Zustand repräsentieren, was es weiter vereinfacht, nur gute Nachrichten zu senden. Leider hatte ich keine Gelegenheit, herauszufinden, wie Hubs erzeugt werden können, die mehrere unterschiedliche Schnittstellen akzeptieren, und eine inter-hub-kommunikation erschien mir für dieses Projekt unnötig aufwändig.

Da Nachrichten aus einem Header und einer Payload bestehen (zum Beispiel: Header: SpieleAudio, Payload: Datei XXX), lässt sich auf dieser Ebene ebenso eine Eingrenzung und Absicherung der Nachrichten vornehmen. Beispielsweise würde ein Client, der eine SpieleAudio-Nachricht empfängt, auf das Vorhandensein der Datei XXX prüfen, bevor er versucht, diese abzuspielen. Ich habe entsprechend diesen Ansatz implementiert, um eine feste Nachrichtenstruktur zu erzeugen, für deren Anwendbarkeit man nur konfigurieren muss, die sich aber dennoch durch Programmierung erweitern lässt. Hierbei ist zu berücksichtigen, dass solche Programmierung sowohl im Unity-Client erforderlich ist, um das Kommando auszuführen, als auch auf der Webseite, um dem Benutzer eine komfortable Möglichkeit für die Auswahl des Kommandos zu geben (beispielsweise eine Auswahl der Audiodatei über ein html file input element). Ich nenne diese Art von Kommandos registrierte Befehle.

Als dritte Option habe ich noch allgemeine Befehle implementiert, bei denen der Empfänger im Unity-Client natürlich programmiert werden muss (zwecks Korrektheitsprüfung und Ausführung), die Webseite aber kein spezifisches UI für das korrekte Erstellen der Nachrichten bieten muss, was das Erzeugen neuer Befehle stark vereinfachen könnte (da die Kompetenzen für Unity-Entwicklung und Webentwicklung nicht zwangsläufig beide in derselben Person vorhanden sind). Hierfür würde auf der Webseite in einem Freitextfeld das Kommando „audio=XXX“ eingegeben werden. Ich bezeichne diese Form von Kommandos als unregistrierte Befehle oder schlicht Befehle.

Ich hatte zunächst nur unregistrierte Befehle implementiert, und dann registrierte Befehle eingebaut, und überlegt, die Funktionalität für unregistrierte Befehle wieder zu entfernen. Allerdings ist die Möglichkeit, unregistrierte Befehle senden zu können nicht nur für das schnelle Erstellen neuer Funktionalitäten nützlich, sondern auch dafür, die Umgebung einfach skripten zu können.

Eine Befehlsfolge wie

- Sende „color=white“
- Sende „audio=GruselGeräusch“
- Warte 1 Sekunde
- Sende „color=black“

leicht einbauen zu können, scheint mir ein nützliches Feature, das ich zwar im Rahmen des Projekts nicht selbst implementiert habe, für das man aber an der Quellcodebasis weiterarbeiten können sollte.

### Farbauswahl

Hierbei wird einem unity-Renderer eine neue Farbe zugewiesen. Diese kann als html-Farbwert #rrggbb sowie als Farbname übergeben werden. Dies ist der simpelste Befehl, den ich auch zuerst implementiert hatte. Außer dem Herausparsen der Farbe aus dem Nachrichteninhalt gibt es hier wenig interessantes zu sehen. Für die Webseite verwende ich ein html input-Element vom Typ color, dessen Wert gebunden ist und bei Änderung zu einer Nachricht führt.

In der Demoanwendung ist die Farbauswahl auf den Würfel gebunden.

### Animationsauswahl

Hierbei spielt ein unity-Animator eine Animation ab. Dies entspricht dem Wechsel des Animators in einen anderen Zustand, dessen Namen der Name der Animation ist. Hier ist klar, dass man dem Anwender der Webseite eine Liste der möglichen Animationen zur Verfügung stellen muss, diese wird über ein html select und dessen option-Elemente dargestellt und ausgewählt. Allerdings müssen die möglichen Animationen auch zur Webseite übermittelt werden. Dafür habe ich eine AddAnimations-Nachricht implementiert, mit der unsere Kommunikation nicht mehr unidirektional ist. Diese wird im Zuge der Initialisierung der Objekte an den Server gesendet, und landet via Broadcast natürlich auch beim webseiten-Client. Hier wird sie geparkt und die option-Elemente angelegt.

In der Demoanwendung verwendet ich einen aus dem Asset Store importierten avatar, „UnityChan“, dessen Animator angebunden wird.

### Audioauswahl

Das ursprüngliche Ziel des Projektes ist es, den Unity-Client eine Audiodatei abspielen zu lassen. Hierfür habe ich zunächst den einfachen Fall von Audiodateien implementiert, die mit dem unity-Programm ausgeliefert werden, und entsprechend auf der Maschine vorliegen, auf der dieses läuft:

Die Audiodatei wird als lokaler Pfad in der Nachricht („audio=c:\...“) übermittelt. Existiert sie, wird ein file:// - URL an ein unity-WWW-Objekt übergeben, und von dieser ein unity-AudioClip erzeugt, der von einer unity-AudioSource abgespielt wird.

Die WWW-Klasse innerhalb von unity ist zwar deprecated, wird also demnächst nicht mehr zur Verfügung stehen, allerdings bietet die empfohlene alternative UnityWebRequest keine einfache Methode, einen AudioClip zu erzeugen. Entsprechend muss irgendwann dazu recherchiert werden.

#### Au Wei - Keine mp3!

Zu diesem Zeitpunkt stellen wir fest, dass unity keine mp3-Dateien abspielen kann, zumindest nicht unter windows, zumindest nicht ohne dafür im Asset Store weitere Assets kaufen zu müssen. Ich habe mich dafür entschieden, die Audiodateien grundsätzlich nach .ogg zu konvertieren (.wav ist viel zu groß, und .ogg ist zumindest bei meinen Beispiel-Audios immernoch kleiner als .mp3). Hierfür verwende ich ffmpeg.

Zu klären ist, wo man die Konvertierung stattfinden lässt. Da ffmpeg etwa 60MB groß ist, scheidet eine Konvertierung beim Aufrufer der Webseite aus. Folglich ist zu entscheiden zwischen Server und unity-Client. Da der Unity-Client auch auf ggf. relativ schwachen HMDs laufen soll, will ich dort keine zusätzliche Belastung auslösen. Zudem ist eine Übertragung ggf. größer .wav-Dateien zum unity-Client eine unnötige Verzögerung, und sobald mehrere Clients involviert sind, würden die Dateien mehrmals übertragen und mehrmals konvertiert werden.

Folglich sollte die Konvertierung auf dem Server passieren, idealerweise beim Upload. Entsprechend habe ich auf dem Server eine Konvertierung von audiodateien nach .ogg implementiert. Hierfür habe ich eine kleine Middleware<sup>1</sup> in den Protokoll-Abarbeitungsstack eingefügt, die diese Konvertierung anwirft, indem zunächst ggf. eine aktuelle Version von ffmpeg heruntergeladen wird, und diese dann mit der ursprünglichen Datei sowie dem neuen Dateinamen mit der .ogg – Endung aufgerufen wird. Um dies zu vereinfachen, binde ich das für nichtkommerzielle Zwecke kostenlose nuget-Package „Xabe.FFmpeg“ ein.

#### Entfernte Dateien

Erfreulicherweise lässt sich der URL- Mechanismus mit WWW auch verwenden, um Dateien von einem entfernten System zu laden und abzuspielen. Statt eines file:// - URL wird dann ein http:// - URL an die WWW-Instanz übergeben.

Bleibt also nur die Frage, wie die Datei nach der Auswahl auf der Webseite auf den Server transportiert wird. Daraufhin muss der unity-Client nur noch einen URL des Servers laden. Denn es ist klar, dass der Rechner, mit dem die Webseite besucht wird, nicht zwangsläufig vom unity-Client aus erreichbar ist (und auch eher keinen Dateiserver zur Verfügung stellt).

Für diesen zweck hatte ich zunächst überlegt, die vollständige Audiodatei tatsächlich per SignalR zu übertragen, allerdings erreicht man schnell die maximalen Paketgrößen, sodass das unpraktikabel ist. Ich habe daraufhin eine Variante implementiert<sup>2</sup>, bei der die Datei in mehreren kleinen Teilen (Chunks) übertragen, und dann auf dem Server wieder zusammengesetzt wird.

Da mir dieses Vorgehen äußerst umständlich erschien, habe ich stattdessen einen Dateiupload per POST eingebaut<sup>3</sup>, sodass gewöhnliches http form method=post seitens der Webseite verwendet werden kann. Dieser erzeugt ebenso einen zufälligen Namen (ich verwende eine GUID) für die

---

<sup>1</sup> OggConverterMiddleware.cs

<sup>2</sup> ChunkedFileUploadMiddleware, leider nicht funktionsfähig; ich war mit den Versionen durcheinandergeraten.

<sup>3</sup> FileUploadMiddleware.cs

hochgeladene Datei. Leider ist es nicht so einfach, nachdem man bei einem Formular einen post-Upload durchgeführt hat, eine Nachricht vom Server zu bekommen, dass der Upload vollständig ist, und ich wollte keine JQuery-Abhängigkeit einführen. Entsprechend lasse ich das Formular in einen unsichtbaren IFrame schreiben, und warte in einer Schleife darauf, dass sich dessen Inhalt ändert.

[What goes up, must come down.](#)

Sobald der Upload also erledigt ist, löst die Webseite eine „audio=http://{server}/posted/....ogg“ – Botschaft vom Server an die Clients aus. Daraufhin will das unity-WWW – Objekt aus diesem URL den AudioClip erzeugen. Ein http Get ist erforderlich.

Damit das funktioniert, muss als letzter Schritt noch die hochgeladene und konvertierte Datei vom unity-client aus erreichbar sein. Hier funktioniert das eingebaute UseStaticFiles() von ASP.NET core nicht, sodass ich über eine weitere kleine Middleware<sup>4</sup> selbst den Filedownload zur Verfügung stellen muss.

### Audiosteuerung

Zusätzlich wurden noch einfache Kommandos für die Steuerung des Audiosignals implementiert, die keine derart komplexe Logik erfordern:

- mute               Stellt die AudioSource stumm.
- unmute           Stellt die ursprüngliche Lautstärke der AudioSource wieder her.
- stop               Beendet das Abspielen der AudioSource.

### Unity-Assets

Nachdem bei den Erläuterungen der Befehle schon die grundsätzlichen Vorgehensweisen für die Ausführung der einzelnen Kommandos dargestellt wurden, gehe ich jetzt auf die Architektur der Unity-Komponenten ein. Diese wurde mit dem Augenmerk auf möglichst geringen Umfang für notwendige Anpassungen und Erweiterungen entworfen.

Es sei ferner auf die API-Referenz im Anhang verwiesen, sodass ich hier nur auf die Funktionalitäten eingehe.

### Verweis zum Server

Die Server-Komponenten kommuniziert mit dem Webseiten-Client sowie mit unity-Clients.

Jeder unity-Client muss natürlich wissen, wie er den Server erreichen soll. Bei großen Anwendungen würde man einen DNS-Namen registrieren, und diesen hart in die Anwendung hinein kompilieren. Für kleinere Anwendungen, die flexibler auf möglicherweise unterschiedliche Server reagieren sollen, lässt man das konfigurierbar. Hierfür gibt es diverse Möglichkeiten, beispielweise manifest-Dateien, Umgebungsvariablen, Registry-Einträge, ein zentrales Repository oder die Anforderung, den Benutzer den Server zur Laufzeit auswählen zu lassen. Ich habe mich für den Kommandozeilenparameter entschieden als einfach zu implementierende und leicht zu testende Variante: Startet man die unity-Anwendung mit dem Parameter „server={servername}“, wird *servername* als Servername verwendet, ansonsten werden die in unity konfigurierten Werte genutzt.

Dieses Verhalten kann deaktiviert werden, indem die useCommandLine-Eigenschaft auf false gesetzt wird.

---

<sup>4</sup> FileDownloaderMiddleware.cs



## Unity-Architektur

Dies ist die Architektur innerhalb der Kommunikationshierarchien, mit deren Kommandos verteilt werden können: Jedes Programm, das mit dem Server kommuniziert, hat mindestens einen Client. Ein Client meldet sich beim Hub an. Ein Client kann mehrere Entitäten aufweisen. Entitäten sind Dinge innerhalb der Anwendung, die Kommandos ausführen können. Jede Entität hat Features. Ein Feature stellt eine Menge zusammengehöriger Kommandos dar. Für das Beispielprojekt ist das Mapping äußerst einfach:

Client: VRProject

Entitäten: Würfel, Audioquelle, Avatar

Features: Würfel: Farbänderung, Audioquelle: Audioplayback, Avatar: Animationen

Jeder Client, jede Entität und jedes Feature verfügen über einen Namen, mit denen sie über die Webseite angesprochen werden können. Dieser Aspekt ist noch nicht in der Webseite implementiert.

Jede Entität weiß zur Entwurfszeit in unity, welchem Client sie zugehörig ist, jedes Feature kennt seine Entität. Zur Initialisierungszeit werden die „Kinder“ ihren „Parents“ bekannt gemacht. Daraufhin wird die SignalR-Verbindung gestartet, und alle Features darüber informiert, dass sie verbunden sind. Dies ist der Zeitpunkt, wo z.B. die Animationsliste übertragen wird.

Daraufhin wartet der Client auf Kommandos, und leiten diese an die entsprechenden Entitäten und diese jene wiederum an die Features weiter, die die Befehle dann ausführen.

## Ein konkretes Feature

Ein Feature kann als registrierter Befehl oder als gewöhnlicher Befehl implementiert sein. Für die Demoanwendung habe ich alle Features sowohl auf beide Weisen implementiert, sodass es insgesamt sechs konkrete Implementierungen gibt. Allerdings sind die eigentlichen unity-spezifischen Aspekte der Features in einer abstrakten generischen Basisklasse implementiert, sodass dieser Teil nur einmal geschrieben werden muss. Die konkrete Klasse für die Implementierung des Features erbt dann von dieser abstrakten Basisklasse, in der steht, *wie* das Feature implementiert wird. Bei dieser Vererbung wird ein generischer Parameter mitgegeben, der angibt, *wodurch* das Feature ausgelöst wird; durch einen registrierten Befehl oder einen gewöhnlichen.

Am Beispiel:

SignalRRegisteredCommandAnimationController	Konkrete Klasse. Steuert Animationen ausgelöst von Registriertem Befehl.
erbt von	
SignalRAnimationControllerBase	Weiß, wie man in unity Animationen steuert mit dem generischen Parameter
SignalRRegisteredCommandEntityController	Weiß, wie man von einer Entität registrierte Befehle entgegennimmt und ausführt.

Auf diese Art und Weise können wir das Dilemma der fehlenden Mehrfachvererbung in C# umgehen.

## Konfiguration in unity

Nachdem man die Assets dieses Projekts in ein unity-Projekt, Assets-Ordner, Skripts-Ordner importiert hat, können die einzelnen Komponenten in die Szene eingefügt werden. Es bietet sich an, ein leeres Objekt einzufügen, das den Client repräsentiert. Diesem muss ein *SignalR Connection Manager* sowie



ein SignalR Client Controller hinzugefügt werden. (bei mehreren Clients könnte jeder Client Controller evtl. ein eigenes Objekt im Szenenbaum bekommen, dies ist aber nicht notwendig).

Dem Client Controller wird nun per drag&drop der Connection Manager bekannt gemacht. Dazu zieht man die Überschrift der „Signal R Connection Manager (Script)“ – Komponente auf das Feld „Connection Manager“ des SignalR Client Controllers.

Im Connection Manager sollte noch der korrekte Weg zum Signal R Server eingetragen werden, und der hub-Name, sofern er sich vom voreingestellten unterscheidet.

Jetzt werden Entities erfasst. Dazu werden Signal R Entity-Komponenten hinzugefügt. Diese können gleich auf die entsprechenden Knoten im Szenenbaum gepackt werden, können aber auch an beliebigen anderen Stellen liegen, z.B. im vormals leeren Objekt vom vorherigen Schritt. Diese Entities müssen einen Namen bekommen, und der für sie zuständige Client Controller muss zugewiesen werden. Dies geschieht wieder per drag&drop oder mit dem kleinen ☺-Symbol rechts neben dem Eingabefeld.

Jetzt benötigt die Entity noch mindestens Controller, entweder einen SignalR Registered Command Entity Controller oder einen SignalR Command Controller, damit sie weiß, wie sie auf Befehle reagieren soll. Diesem Controller muss die Entity bekannt gemacht werden, wie eben per drag&drop oder ☺.

Im letzten Schritt kann man die Features, die von dem Controller verwendet werden, als Komponenten hinzufügen. Also beispielsweise einen „SignalR Registered Command Color Controller“. Dieser muss jetzt seinen Controller kennen lernen. Außerdem müssen noch featurespezifische Eigenschaften gesetzt werden, beispielsweise die Audioquelle, über die abgespielt werden soll.

### NuGet aber nicht!

Beim Anbinden von SignalR ist aufgefallen, dass unity leider keine NuGet-Unterstützung mitbringt. Unter dem Projekt Tools/NugetDownloader habe ich daher schnell eine Anwendung gebaut, um für ein Nuget-Paket alle benötigten Referenzen herunterzuladen. Diese sind in den unity-Assets unter lib versioniert, und müssen mit importiert werden.

### Fehlerbehandlung? Fehlanzeige

Unity-Anwendungen haben die angenehme Eigenschaft, dass sie nicht abstürzen, falls im C#-Teil Exceptions nicht abschließend verarbeitet wurden. Ich habe daher für dieses Projekt mit dem Ziel, besser lesbaren Quelltext zu produzieren darauf verzichtet, größere Anstrengungen für die Fehlerbehandlung zu unternehmen. Entsprechend gibt es nur in Unity (und eventuellen Protokollen) Fehlermeldungen, wenn beispielsweise der Server nicht läuft, und sich keine Verbindung herstellen lassen will, oder wenn Unity-Eigenschaften nicht korrekt konfiguriert sind.

### Webseite und - Server

Da der Fokus im Seminar primär auf Unity liegt, gehe ich an dieser Stelle nicht tiefer auf die Webseite ein. Soviel sei gesagt: Das Grundgerüst bildet die Blazor-Beispielanwendung, die ich um SignalR erweitere und von unnötigem entschlackt habe. Die Anwendung beinhaltet sowohl einen Webserver aus auch die von ihm gehostete Webseite, sodass hier hauptsächlich Anpassungen und weniger Neuentwicklung notwendig war. Insbesondere das JWT-Token – Erzeugen kam geschenkt und ist entsprechend auf Demo-Niveau.

Auf die zusätzliche Middleware zum Dateien hochladen, konvertieren und herunterladen bin ich oben schon eingegangen. Der SignalR-Hub verfügt über Methoden für die implementierten registrierten Befehle Audio, Color und Animation sowie die Methode Command für nichtregistrierte Befehle.

Die Demo-Webseite selbst war schon ein Chat, sodass ich hier nur Methoden anpassen und auf Broadcast einschränken musste. Insbesondere die Gruppenzuordnung, mit der man mehrere konkrete Clients ansprechen kann, ist ausgelassen worden, ebenso wie das Senden und Anzeigen des Client- und Entity-Namens.

## Installation

Die Unity-Assets müssen aus dem Ordner **Publish** in das unity-Verzeichnis kopiert werden. Nicht den **lib**-Unterordner vergessen.

In unity muss die SignalR Connection Manager Komponente den korrekten SignalR-Pfad bekommen.

Leider erfordern Webserver Zertifikate, damit das SSL funktioniert. Dies zu konfigurieren ist aufwändig und nicht Gegenstand dieser Dokumentation.

Daher habe ich SSL im Debug deaktiviert, sodass die Kommunikation über einfaches http möglich ist. Im Ordner Publish\win-x64 stehen die notwendigen Artefakte für das Ausführen des Servers zur Verfügung (für Windows kompiliert). Standardmäßig lauscht der Server auf `http://localhost:5000`. Mit dem Parameter `--urls "http://localhost:8080"` kann man ihn auf Port 8080 lauschen lassen, entsprechend auch auf anderen Ports und anderen Hostnamen.

Man sollte danach die Adresse im Browser öffnen, und auf den Punkt **Fernsteuerung** gehen. Da der Server keinen Zustand implementiert, werden die Animationen des unity-Clients sonst nicht aufgelistet, denn sie werden beim Anmelden des unity-Clients propagiert.

Jetzt können die Unity-Anwendungen gestartet werden.

Erinnerung: der Parameter `server=http://localhost:8080` lässt den unity-Client auf den soeben eingerichteten Server lauschen.

Anmerkung: Statt localhost sollten natürlich auch IP-Adressen oder tatsächliche Rechnernamen funktionieren.

Jetzt muss nur noch die Auflösung konfiguriert werden, und das Fernsteuern kann beginnen.

## Anhang

### Weitere Arbeiten / TODOs

Die hier aufgeführten Notizen sind Punkte, an denen für eine weitere Arbeit mit diesem Projekt recherchiert / programmiert werden sollte.

- Audio-Abspielen nicht mehr auf WWW-Klasse basieren lassen
- AddAnimations muss für jede Entität eine eigene Animationsliste kommunizieren bzw. diese müssen auf Server- bzw. Webseiten-Seite gemerkt werden für die Laufzeit der Anwendung
- Zustand der Clients implementieren
- Clients auf GUI-Seite unterscheidbar machen – Namen verwenden, um beim Senden der Nachrichten zu entscheiden, an welches Programm gesendet wird.
- Entitäten auf GUI-Seite unterscheidbar machen – Namen verwenden, um beim Senden der Nachrichten zu entscheiden, an welche Entität gesendet wird.
- Features auf GUI-Seite unterscheidbar machen – Nachrichten nur an Entitäten senden, die ein Feature haben.
- Auf Alternativen zum ffmpeg-Prozessaufwurf prüfen, insb. In Hinsicht auf Lizenzen
- Xabe.FFmpeg nicht mehr verwenden
- Herausfinden, wieso UseStaticFiles() im Startup des Servers nicht geht, um die Dateien in /posted zur Verfügung zu stellen.
- Clients, Features, Entitäten zur Laufzeit veränderbar machen
- Client: Abmelde-Handshake implementieren
- Explizite controller-zuordnung evtl zur Initialisierungszeit über die Baumstruktur lernen?
- Fehlerbehandlung verbessern (geht immer)
- Dokumentation / Kommentierungen verbessern (geht immer)

## API-Referenz Unity-Assets

Hier werden die relevanten Eigenschaften und Methoden der Unity-Assets erklärt.

Anmerkung: Unity verwendet nicht die Eigenschaften von Klassen, um den Inspector zu befüllen, sondern public Fields. Diese beginnen zudem per Konvention mit einem Kleinbuchstaben.

Die hier entwickelten Assets verwenden intern aber normale Properties

Wenn keine Basisklasse angegeben wurde, ist MonoBehaviour die Basisklasse.

### SignalRConnectionManager

Stellt die Server-Bindung zentral zur Verfügung

Unity-Properties:

signalRServer	string	voreingestellte Serveradresse
hub	string	Name des hubs auf dem Server
useCommandLine	bool	Soll der „server=...“ Parameter ausgewertet werden?

Properties

UsedSignalRServer	string	benutzter SignalR-Server
-------------------	--------	--------------------------

### SignalRClientController

Implementiert einen Client.

Unity-Properties:

connectionManager	SignalRConnectionManager	Bindung zu Serveradresse und Hub
clientName	string	Name des clients
signalREnabled	bool	Soll sich der Client verbinden?

Properties

EntityControllers	List<SignalREntityController>	Die Entitäten des Clients, init-Füllung
-------------------	-------------------------------	---

### SignalREntity

Stellt eine Entität dar.

Unity-Properties

entityName	string	Name der Entität
clientController	SignalRClientController	Der für die Entität zuständige Client

### SignalREntityController

Abstrakte Basisklasse für die Steuerung von Entitäten

Unity-Properties:

remoteControlEnabled	bool	Soll diese Entität gesteuert werden können
entity	SignalREntity	die gesteuerte Entität
signalREnabled	bool	Soll sich der Client verbinden?

Properties

FeatureControllers	List<SignalRUnityControllerBase>	Die Features der Identität
--------------------	----------------------------------	----------------------------

### SignalREntityController<TRegisterEventArgs> : SignalREntityController

Generische abstrakte Basisklasse für die Steuerung von Entitäten

Da Unity keine Properties für den Inspector unterstützt, kann kein sauberes IOC vorgenommen werden, indem mit gettern und settern gearbeitet wird. Daher werden die Entitäten und ihre Features über eine generisch typisierte Initialisierungsmethode miteinander verbunden. TRegisterEventArgs ist der Parametertyp, der bei der Feature-Initialisierung verwendet wird.

#### Abstrakte Methoden

Init	Action<object, EventArgs<TRegisterEventArgs>> onInitFeature	
------	---	--

SignalRCommandEntityController : SignalREntityController<(string Command, string Value)>  
Steuert eine Entität, die mit normalen Befehlen arbeitet.

#### Events

ExecuteCommand	EventHandler<EventArgs<(string Command, string Value)>>	Ereignis, an dass die Features gebunden werden. Wird ausgelöst, wenn ein Befehl ausgeführt werden soll.
----------------	---	---

SignalRRegisteredCommandEntityController : SignalREntityController<HubConnection>  
Steuert eine Entität, die mit registrierten Befehlen arbeitet.

#### Events

ExecuteCommand	EventHandler<EventArgs<HubConnection>>	Ereignis, mit dem die Features registriert werden, wird ausgelöst, bevor die SignalR-Verbindung geöffnet wurde.
----------------	--	---

#### SignalRUnityFeatureBase

Abstrakte Basisklasse für die unity-spezifischen Aspekte eines Features.

#### Methoden

Connected (virtual)	HubConnection hubConnection	Benachrichtigung, dass der SignalR-Hub verbunden ist.
---------------------	-----------------------------	---

SignalRUnityFeatureBase<TEntityController, TInitFeatureArgs> : SignalRUnityFeatureBase  
Generische abstrakte Basisklasse für die unity-spezifischen Aspekte eines Features

TEntityController ist der Typ des Entity-Controllers für das Feature

TInitFeatureArgs ist der Parametertyp für die Initialisierungs-Events

#### Unity-Properties

entityController	TEntityController	Der Controller, der dieses Feature steuert
------------------	-------------------	--

#### Methoden

InitFeature		ruft Init des entityControllers.
OnInitFeature(abstrakt)	object sender, EventArgs<TInitFeatureArgs> args	das Initialisierungs-Feature. Den Kontext bestimmt der konkrete TEntityController-Typ.

SignalRAudioBase : SignalRUnityFeatureBase<TEntityController, TArgs> :

SignalRUnityFeatureBase<TEntityController, TInitFeatureArgs>

Generische abstrakte Basisklasse für das Abspielen von Audioquellen

#### Unity-Properties

audioSource	AudioSource	Audioquelle, von der aus Töne gespielt werden.
-------------	-------------	--

## Eigenschaften

SoundFile	string	Gibt die abgespielte Datei an. (ruft PlayAudio im Setter)
Muted	bool	Stummschalten der Audioquelle

## Methoden

PlayAudio	string url	Lässt die Audioquelle den Clip aus url abspielen.
HandleAudioCommand	string audioCommand	Wertet den Inhalt der Nachricht aus.
StopAudio		Stoppt die Audioquelle

SignalRRegisteredCommandAudioController :

SignalRAudioBase<SignalRRegisteredCommandEntityController, HubConnection>

## Methoden

OnInitFeature (override)	object sender, EventArgs<HubConnection> args	Dispatch von „Audio“-Nachrichten an HandleAudioCommand
-----------------------------	---	---

## Glossar

SPA – Single Page Application

SignalR – Echtzeitkommunikationsmechanismus für Webanwendungen

HTTP – Hypertext Transfer Protocol – Protokoll für das WWW, das auf TCP und IP aufbaut; ab Sitzungsebene im ISO OSI Schichtenmodell

WWW – World Wide Web – Der Webseiten -Teil des Internets.

HMD – Head Mounted Device – VR-Brillen, Kopfhörer, Sensoren usw., das auf dem Kopf getragen wird.

C# (C Sharp) – Multiparadigmen-Programmiersprache, mit der u.A. in Unity3d entwickelt werden kann

Unity3d – Spiele-Engine, auch für VR-Anwendungen. Verwendet u.a. .net Core als Logikengine.

.net Core – Plattformunabhängige Neuimplementierung des .net Frameworks. Eine weitere .net Ausprägung

.net Framework – managed Runtime mit Unterstützung für viele Sprachen mit einheitlichem Typmodell. Windows-Bestandteil, monolithisch, hohe Abwärtskompatibilität. Erste .net Ausprägung.

VR – Virtuelle Realität

RPC – Remote Procedure Call. Eine Möglichkeit für IPC über Rechner hinweg

IPC – Interprozesskommunikation

DCOM – Distributed Component Object Model. Ein für RPC geeigneter nativer Ansatz, um Anwendungen in kleinere wiederverwendbare Komponenten aufzuteilen.

Blazor –SPA-Framework, mit dem Webseiten einschließlich der hintelegten Logik in C# erstellt werden können. Hierbei wird die Anwendung vom nach WebAssembly kompilierten Mono-Framework im Browser ausgeführt.

WebAssembly – Strikte JavaScript-Variante, die als Laufzeitumgebung für andere Anwendungen dienen kann.

Mono – Plattformunabhängige .net Ausprägung. Unterstützt .net Standard

.net Standard – Vereinheitlichte API-Oberflächendeklaration für (fast) alle .net-Ausprägungen. Ermöglicht Wiederverwendung von kompilierten Assemblies in unterschiedlichen .net Ausprägungen.

Assembly – kompilierter Teil einer Anwendung. Besteht aus CIL-Anweisungen und Metadaten

CIL – Common Intermediate Language. Das Kompilationsziel für .net Anwendungen

GUI – Graphical User Interface.

ffmpeg – freies plattformunabhängiges Werkzeug zum Bearbeiten und Konvertieren von Mediendateien

.ogg – Format für komprimierte Audiodateien, freie Implementierung

.mp3 – Format für komprimierte Audiodateien, proprietär (Fraunhofer)

.wav – Waveform-Format, unkomprimierte Audiodateien. Sehr groß.

URL – Uniform Resource Locator



ASP NET core – Webframework für .net, in dessen Rahmen Blazor und SignalR funktionieren. Gut konfigurierbarer Middleware-Stack.

DNS – Domain Name System

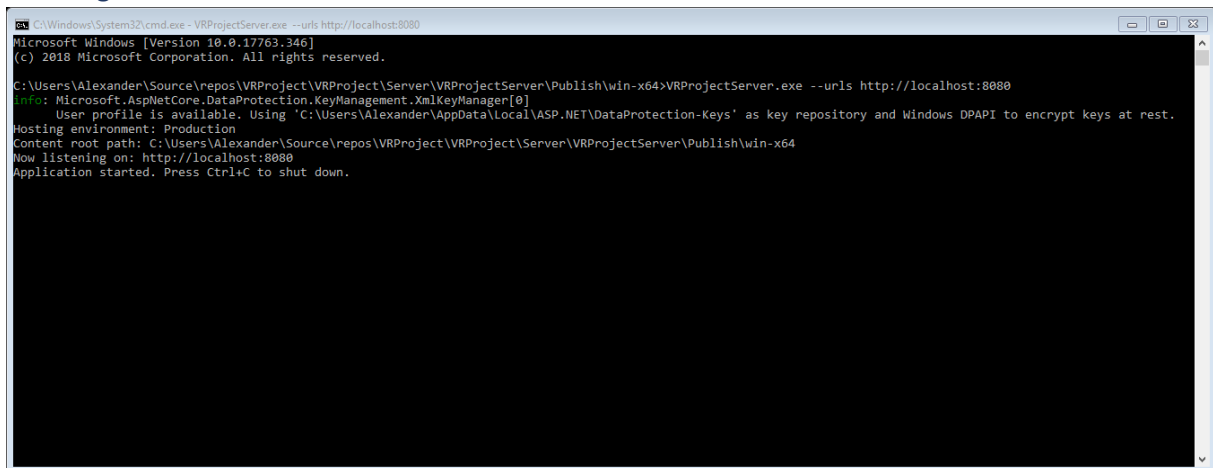
NuGet – Paketmanager für .net

SSL – Secure Socket Layer

## Bilder

Im Folgenden ein Ablauf durch eine typische Fernbedienungssitzung

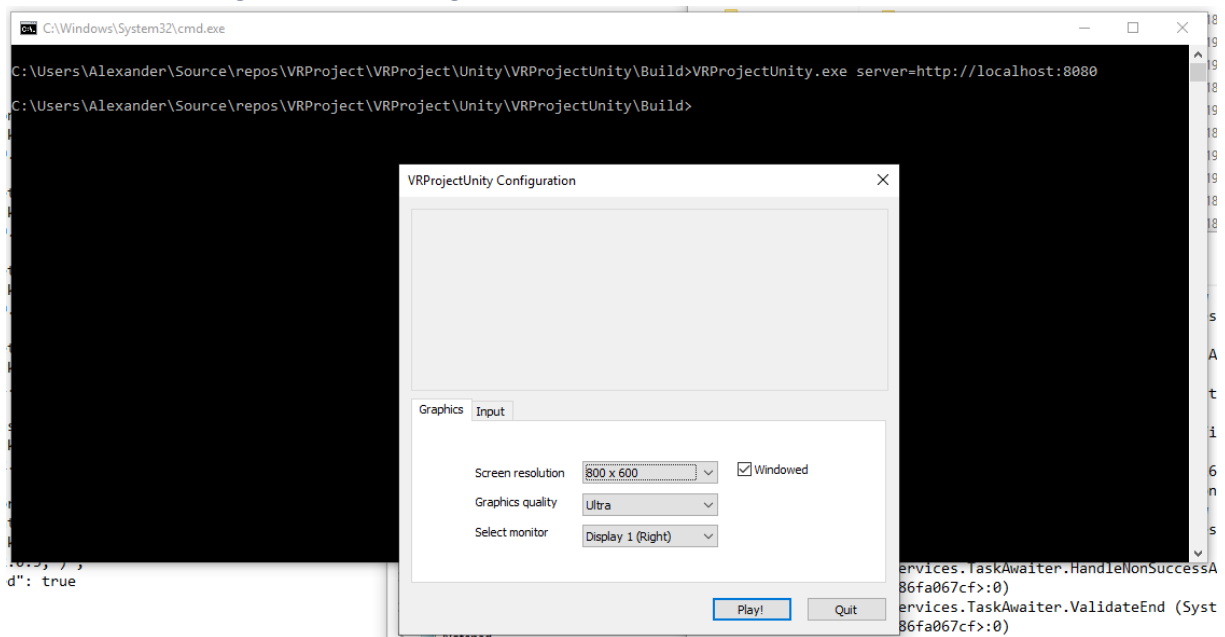
### Server gestartet



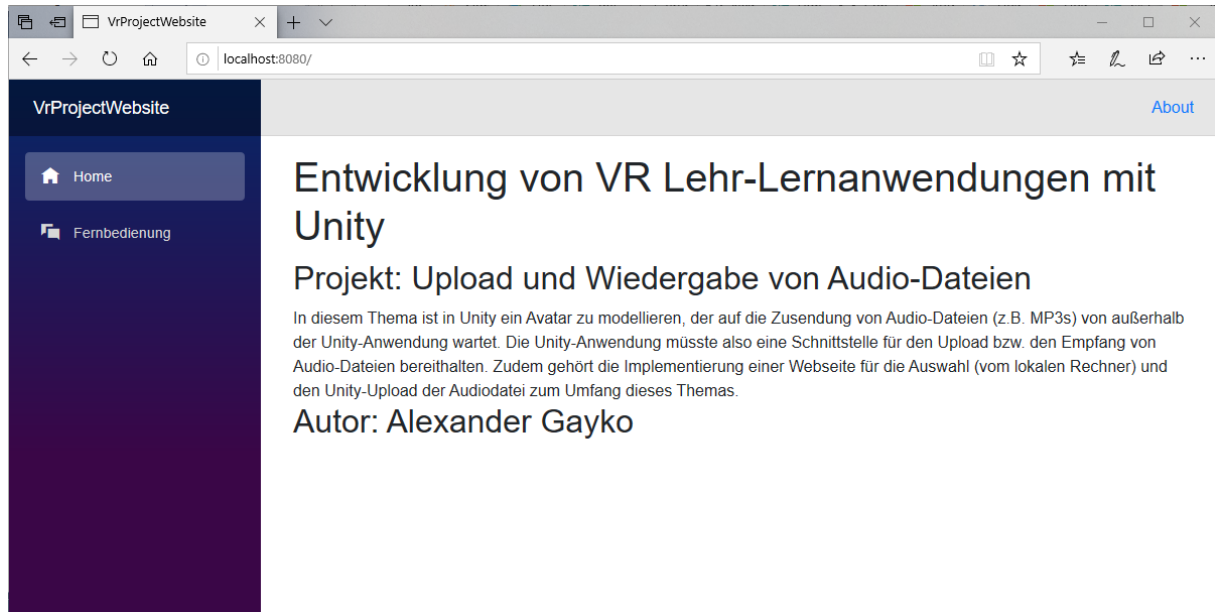
```
C:\Windows\System32\cmd.exe - VRProjectServer.exe --urls http://localhost:8080
Microsoft Windows [Version 10.0.17763.346]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\Alexander\Source\repos\VRProject\VRProject\Server\VRProjectServer\Publish\win-x64>VRProjectServer.exe --urls http://localhost:8080
info: Microsoft.AspNetCore.DataProtection.KeyManagement.XmlKeyManager[0]
      User profile is available. Using 'C:\Users\Alexander\AppData\Local\ASP.NET\DataProtection-Keys' as key repository and Windows DPAPI to encrypt keys at rest.
Hosting environment: Production
Content root path: C:\Users\Alexander\Source\repos\VRProject\VRProject\Server\VRProjectServer\Publish\win-x64
Now listening on: http://localhost:8080
Application started. Press Ctrl+C to shut down.
```

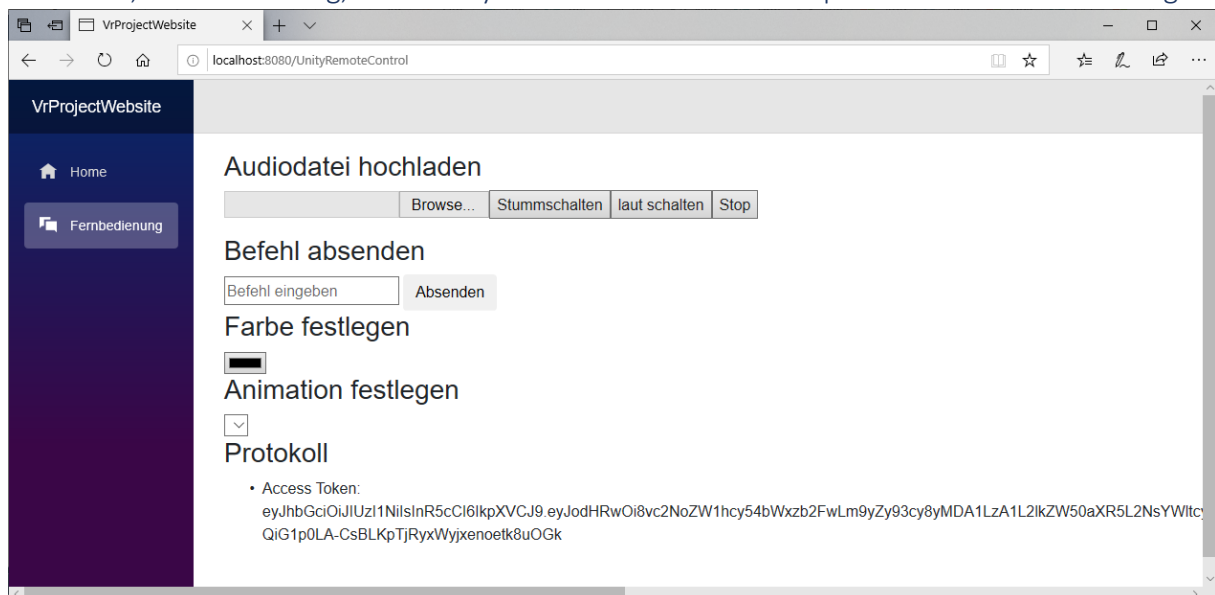
### Demo-Anwendung Start und Konfigurationsbildschirm



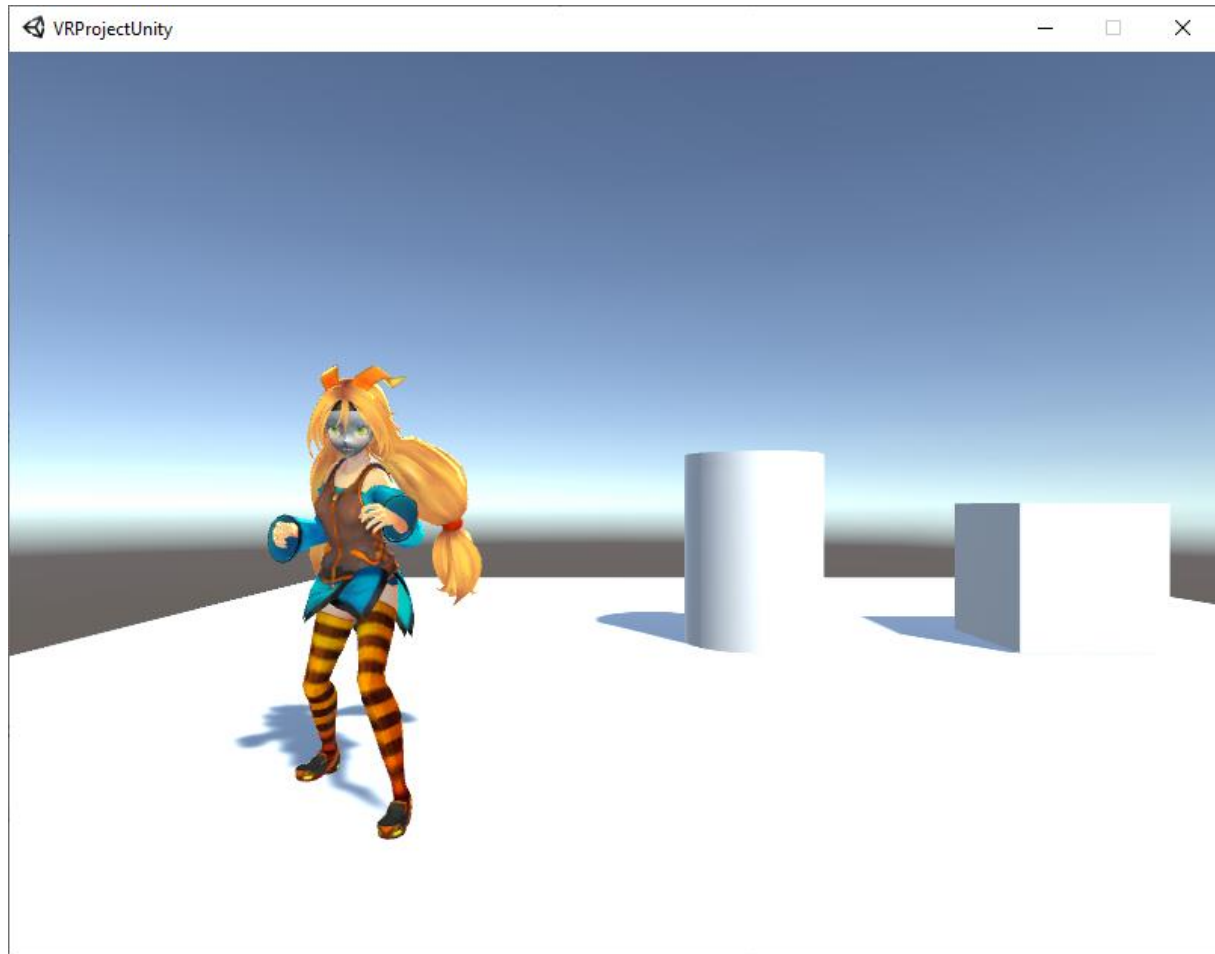
## Webseite, Begrüßungsbildschirm



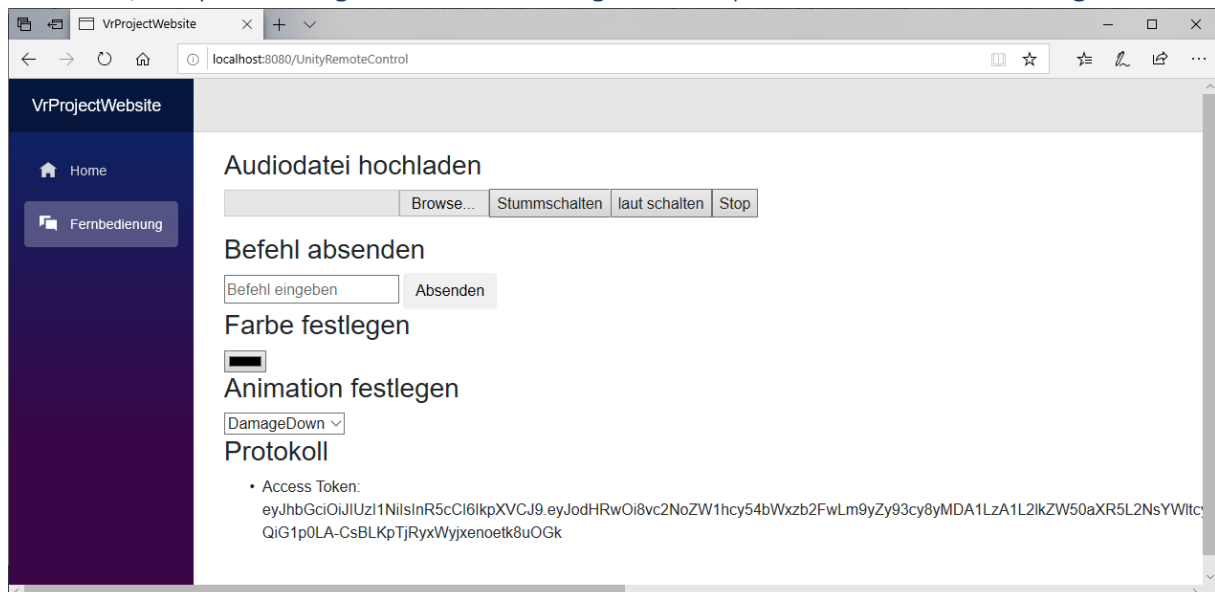
## Webseite, Fernbedienung, ohne unity-Client. Beachte: leeres Input unter Animation festlegen



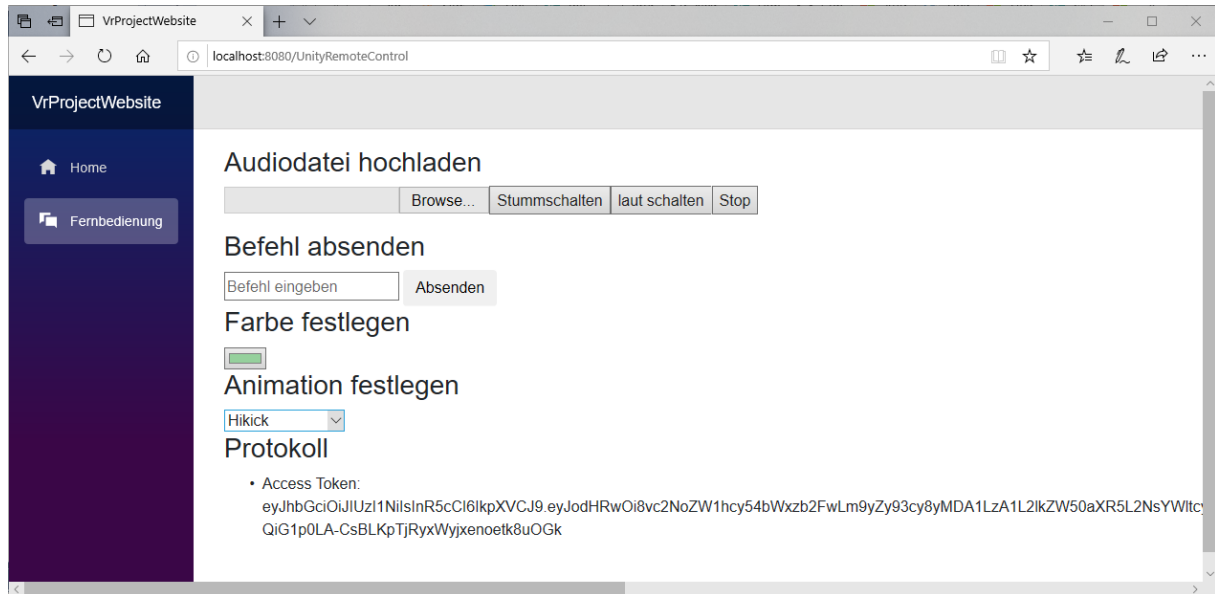
Demoanwendung gestartet



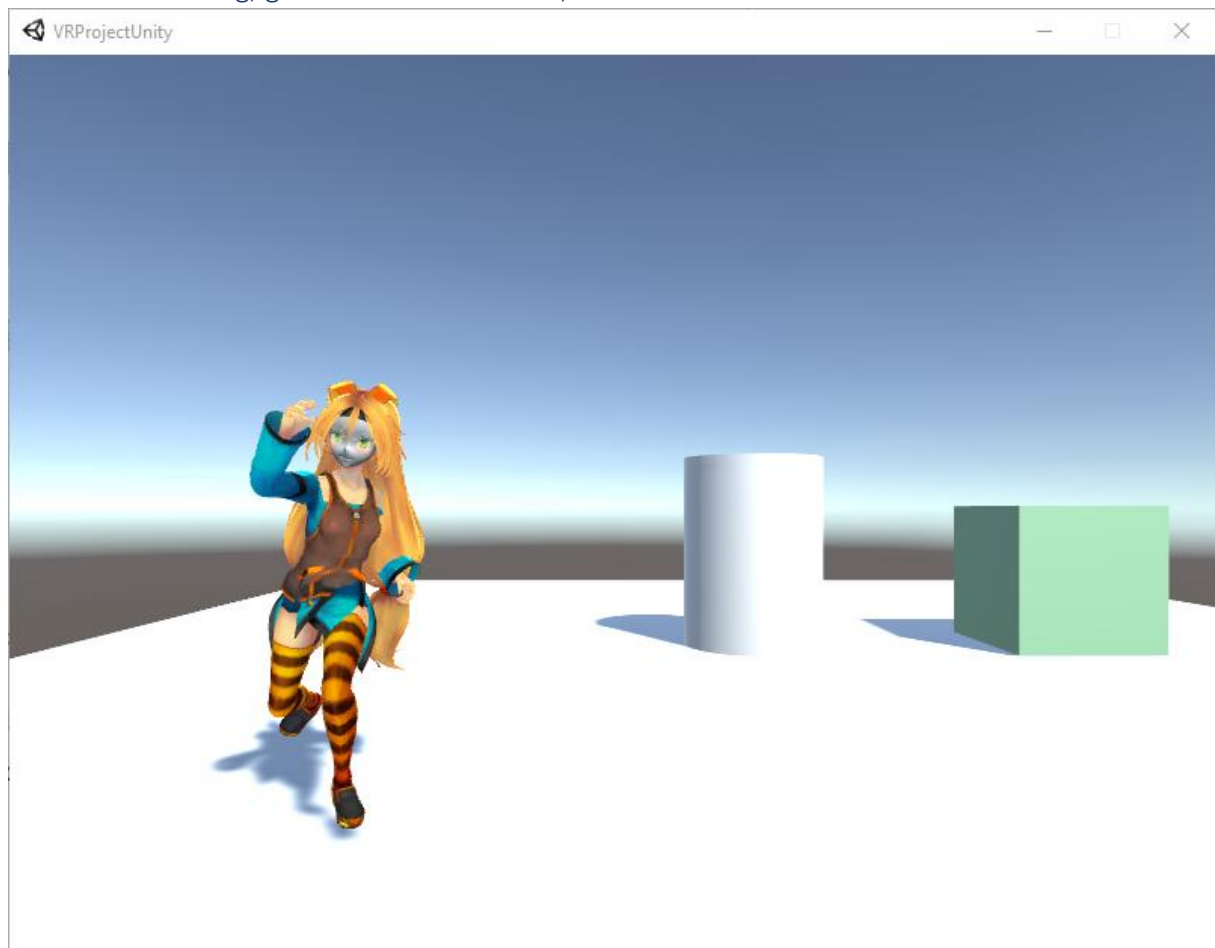
Webseite, unity-Client angemeldet. Beachte: gefülltes input unter Animation festlegen



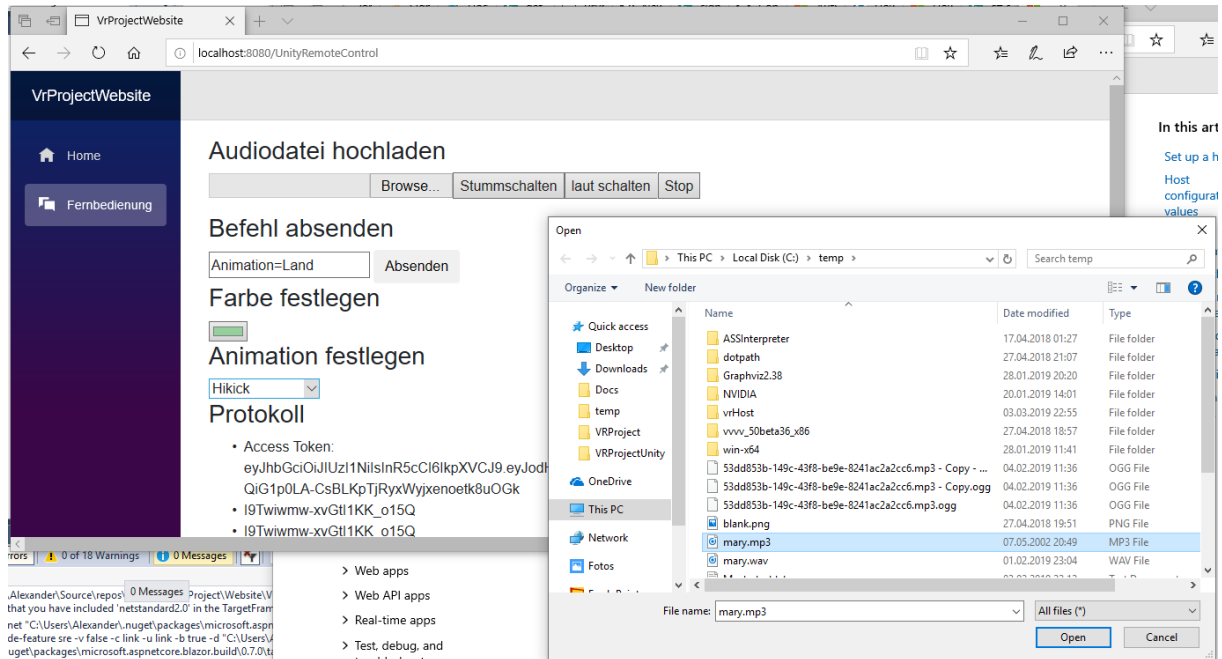
## Webseite, Geänderte Würfelfarbe, Animation ausgewählt



## Demo-Anwendung, geänderte Würfelfarbe, in Animation



## Webseite, Auswahl Audiodatei, eingegebener Befehl



## Webseite, nach Audioauswahl

