Dokumentation

Integration des C++ TUI-Framework in der HTW-CAVE mit Unity3D(C#)

Sebastian Keppler

Angewandte Informatik (M)

HTW Berlin

Independent Coursework WS 15/16

Inhalt

1. Einleitung

Im Rahmen eines Projektes in Independent Coursework soll für die CAVE der HTW-Berlin eine Schnittstelle entwickelt werden, welches es Entwicklern ermöglichen soll verschiedene Eingabegeräte in einem Anwendungsszenario einbinden und benutzen zu können. Die CAVE der HTW verwendet als primäre Anwendungsumgebung die Game-Engine Unity3d, welche als Programmiersprache C# verwendet. Zukünftige Entwicklern soll die Möglichkeit geboten werden verschiedene Ein- und Ausgabe Geräte in ein Unity3d C#-Projekt einzubinden, ohne selbst einen umfangreichen Aufwand bei der Implementierung betreiben zu müssen.

Für die Realisierung dieses Vorhabens soll das vom Fraunhofer Institut entwickelte TUI-Framework benutzt werden, um den Datenaustausch vom Ein- und Ausgabegeräten von und zu Unity3d zu ermöglichen. Ein- und Ausgabegeräte und Anwendungen können an das TUI-Framework gekoppelt werden. Das TUI-Framework verarbeitet die Daten und kann diese für andere Anwendungen zur Verfügung stellen. Das TUI-Framework selbst wurde in C++ entwickelt. [1]

Eine Unterstützung seitens des TUI-Frameworks mit Unity3d fehlt bisher.

Ziel ist es nun eine Schnittstelle zwischen dem TUI-Framework und Unity3d zu schaffen, um einen Datenaustausch zwischen den beiden Frameworks zu realisieren. Die Besonderheit liegt dabei in der Realisierung einer sprachübergreifenden Kommunikation zwischen C++ und C#.

Zunächst soll die Machbarkeit untersucht werden und recherchiert welcher Möglichkeiten zur Realisierung zur Verfügung stehen. Daraufhin sollen die Vor- und Nachteile der verschiedenen Möglichkeiten beleuchtet werden und anhand dessen und den gegebenen Anforderungen ein Vorgehen zur Implementierung beschrieben werden. Es soll außerdem detailliert beschrieben werden was bei der Implementierung zu beachten ist und welche möglichen Änderungen am TUI-Framework und in Unity3d durchgeführt werden müssen.

2. Methoden zur Sprachübergreifenden Kommunikation

2.1 Managed Code mit C++/CLR

Die Common Language Runtime(CLR) ist ein Bestandteil von Microsofts .NET Framework. Das Ziel der CLR ist es Programmcode, welcher in einer bestimmten Programmiersprache entwickelt wurde auch für andere Programmiersprachen innerhalb des CLR-Kontexts verfügbar zu machen. Dabei wird der Programmcode vom Compiler zunächst in eine Zwischensprache übersetzt auch Common Intermediate Language(CIL) oder nur Intermediated Language(IL) genannt. Die CIL wird vom CLR verwaltet weswegen es auch als managed code bekannt ist. CIL und das Konzept zur sprach- und plattformübergreifenden Anwendungsentwicklung sind Teil des Common Language Infrastruktur(CLI) Standards. Der Standard ist definiert unter ISO/IEC 23271 und ECMA-335. [2]

Das .NET Framework ist eine Implementierung des CLI-Standards. Die CLR im .NET Framework unterstützt neben C# und Visual Basic auch die funktionale Programmiersprache F# und C++/CLR. C++/CLR ist eine C++ Variante, entwickelt von Microsoft. Abbildung 1 veranschaulicht den Schematischen Aufbau der Kompilierung mit CLR in .NET Framework.

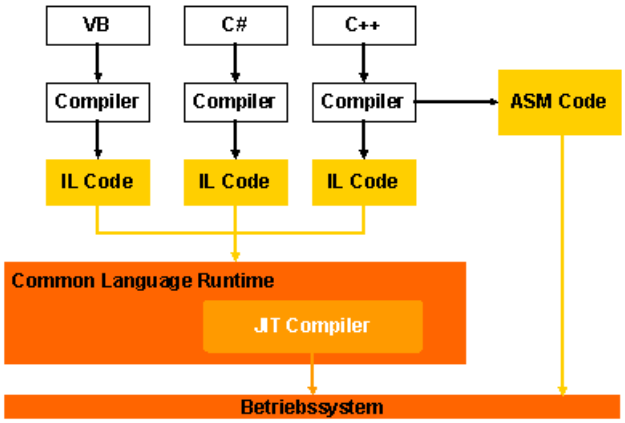

Abbildung 1 Schematischer Aufbau des Kompiliervorgangs mit CLR
[Quelle: https://i-msdn.sec.s-msft.com/dynimg/IC61662.gif]

C++/CLR ist eine eigenständige Sprache die jedoch weitestgehend die bekannten Funktionalitäten aus C++ unterstützt. Ein weiteres Merkmal ist auch die Unterstützung von in C++ geschriebenem Programmcode. Dies lässt sich auch zusammen als sogenannten mixed Code verwenden. [3]

2.2 Unmanged Code mit Nativen C++

Gegenüber dem managed Code gibt es noch als Alternative unmanaged Code mit nativen C++ Code, welcher in Form einer Shared Libary bzw. einer DLL-Datei bereitgestellt wird. Die CLR bietet einen Service der sich Platform Invoke(P/Invoke) nennt. Mit diesem Service ist es möglich Methoden aus einer unmanaged Sprache wie C++ im managed Kontext von C# aufzurufen.
Dadurch kann ein in C++ vorhandener Programmcode in C# verwendet werden, ohne dass der C++ Programmcode auf C++/CLR portiert werden muss. Dies wird auch von Unity3d unterstützt. [4]

Dafür muss jedoch sowohl in C++ als auch in C# bestimmte Definitionen vereinbart werden. In C++ muss eine API vorhanden sein auf die von C#-Seite aus zugegriffen werden kann.

Die API muss in normalem C-Code geschrieben sein damit diese in einer DLL verwendet werden kann. Dies kann man erreichen indem man die Funktionen mit `extern "C"` definiert bzw. umschlossen werden. Dadurch wird der Programmcode nicht als C++ sondern als C-Code vom Compiler gelesen.

Nachfolgend ist ein einfaches Beispiel wie eine Kommunikation zwischen C++ und C# über eine C-API realisiert wird.

Native.h

```
#pragma once

#define TESTFUNCDLL_API __declspec(dllexport)

extern "C"
{
	TESTFUNCDLL_API void printSomething(const char* str);
}
```

Es wird eine TESTFUNCDLL_API mit __declspec(dllexport) definiert, was bedeutet dass diese Funktion aus der DLL exportiert werden soll. Methoden innerhalb des `extern "C"`-Block werden als C-Funktionen deklariert und auch so aufgerufen. Die Verwendung von typischen C++-Klassen wie z.B. der string-Klasse würden während der Laufzeit zum Absturz des Programms führen.

Nativ.cpp

```
#include "Native.h"

TESTFUNCDLL_API void printSomething(const char* str)
{
	printf("%s\n",str);
}
```

In der Definition der Funktion wird diese wie gewohnt Implementiert. Zu beachten ist das auch hier __declspec(dllexport), welches zuvor als TESTFUNCDLL_API definiert wurde, für die Funktion angegeben werden muss.

Demo.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.InteropServices;

namespace Demo
{
    class Program
    {
        static void Main(string[] args)
        {
            printSomething("Hi");
        }
        [DllImport("Native")]
        private static extern void printSomething(string str);
    }
}
```

Über die den DLLImport-Befehl wird die C++ DLL aufgerufen und anschließend einer C# Funktion zugeordnet. Damit die Zuordnung erfolgreich sein kann ,muss die C#-Methode genauso definiert sein wie in der C++ API. Anschließend kann die Methode im Programm verwendet werden.

Es ist darauf zu achten das die DLL im erstellt C++ DLL im selben Ordner liegt wie das auszuführende Programm bzw. die aufgerufene DLL.

2.3 Kommunikation über das UDP Netzwerkprotokolle

Ein Datenaustausch über Netzwerk mit dem UDP-Protokoll wäre die dritte Möglichkeit um eine Kommunikationsbrücke zwischen dem TUI-Framework und Unity3d realisieren. Innerhalb des TUI-Frameworks wird bereits UDP verwendet. Ein TUI-Server empfängt die Daten von einem externen Gerät oder Anwendung, verarbeitet diese und schickt diese weiter an einem TUI-Client. Die Daten kann der Client verarbeiten und anschließend bei Bedarf zurück an den Server senden. Sowohl der TUI-Server als auch der TUI-Client wurden in diesem Fall mit den Funktionen die das TUI-Framework bereitstellt implementiert. Aber auch ohne die Funktionalität des TUI-Frameworks ist es möglich die Daten in Unity3d per UDP zu empfangen. Dazu muss in Unity3d mit C# ein UDP-Client zum empfangen und senden von Daten geschrieben werden. Dieser UDP-Client kann sich dann auf die IP-Adresse und Port des TUI-Servers lauschen und die Daten empfangen.

Der TUI-Server serialisiert die Daten bevor er diese über das Netzwerk versendet. Am Unity UDP-Client würde somit ein String mit verschiedensten Werten ankommen, der dann noch deserialisiert werden muss um die Daten verarbeiten zu können.

3. Auswertung der einzelnen Methoden

3.1 Auswertung von C++/CLR

Auf dem ersten Blick erscheint C++/CLR eine gute Lösung zu bieten um die Funktionalitäten des TUI-Frameworks in Unity3d zu verwenden. Im Compiler von Visual Studio gibt es die Möglichkeit C++ mit CLR Unterstützung zu kompilieren, was mit dem */clr* Compiler-Parameter geschieht. Jedoch produziert dies für Unity3d noch keinen verwendbaren CIL-Code. Eine C++ DLL mit welche mit /clr kompiliert wurde wird in Unity3d nicht als managed Code akzeptiert werden und kann folglich nicht verwendet werden. Das liegt daran weil Unity3d ausschließlich nur sicheren CIL-Code akzeptiert. Erst wenn der Befehlsparameter abgewandelt wird in /clr:safe wird dem Compiler angewiesen sicheren CIL-Programmcode zu generieren welcher dann auch von Unity3d akzeptiert wird.

Wird das TUI-Frameworks jedoch mit /clr:safe kompiliert kommt es zu zahlreichen Fehlern. Erwähnenswert ist, dass die Fehler mit /clr:safe erst auftraten nachdem die .NET Version im C++/CLR-Projekt von der aktuellen Version v4.6.1 auf v3.5 umgestellt wurde. Dies ist nötig damit die C++/CLR Bibliothek mit Unity3d kompatibel ist. Weitere Information dazu werden unter Punkt 4.2. erläutert. Zunächst wurde versucht das Projekt mit Visual Studio(VS) 2015 Enterprise zu kompilieren, was nicht gelang. .NET v3.5 wurde eingeführt mit VS 2008. Da VS 2015 ein anderes Toolset verwendet als 2008 wurde auch versucht das Projekt auch unter VS 2008 zu kompilieren was misslang, da trotzdem zahlreiche Kompilierungsfehler auftraten.

Um diese Fehler zu beheben müssten weitreichende Änderungen an den Hauptkomponenten des TUI-Frameworks vorgenommen werden um den Quellcode kompatibel zu C++/CLR zu haben. Dies würde ein erheblicher Aufwand und Zeit benötigen um die Änderungen durchzuführen. Hinzukommt das bei solchen umfassenden Änderungen die Funktionalität des TUI-Frameworks neu getestet und evaluiert werden muss, was auch mit einem hohen Zeitaufwand verbunden wäre. Da nicht abzuschätzen war wie viel tatsächlicher Aufwand nötig ist um das TUI-Framework mit C++/CLR kompatibel zu bekommen wurde dieser Ansatz nicht weiter verfolgt..

3.2 Auswertung von Importieren von Nativen C++-Code

Dadurch das es möglich ist bereits bestehenden C++ Code zu exportieren und wieder in C# zu importieren ohne Änderungen am Quelltext vorzunehmen, erscheint dies die derzeit beste Möglichkeit TUI und Unity3d miteinander zu koppeln. Bestehende TUI-Funktionen können unverändert bleiben, was sehr für die Verwendung dieser Methode spricht. Es muss lediglich eine API entwickelt werden welche die Funktionalität zu Entwicklung eines TUI-Clients exportiert. Auf der Seite von C# müssen diese Funktionen wiederrum bekannt gemacht werden. Der Nachteil liegt jedoch in der Wartbarkeit des Programmcodes. Wenn neue Features implementiert werden oder bestehende Funktionsdeklarationen geändert werden, muss dies immer sowohl in im Programmcode als auch in der API geändert werden.

Außerdem muss sichergestellt werden das die Zuordnung der Funktionen in C# richtig gesetzt wird.

Es können außerdem nur Funktionen exportiert werden. Datentypen oder ganze C++ Klassen können auch nur in Form von Funktionen exportiert werden. Zusätzlich ist zu beachten, dass eine API in eine DLL nur kompiliert werden kann wenn diese C-Konform ist. Das heißt die API darf nur in C geschrieben sein und darf kein C++ Code beinhalten.

3.3 Auswertung das UDP-Netzwerkprotokoll

Eine Datenkommunikation über UDP zu realisieren wäre die einfachste zu implementierende Variante von allen bisher vorgestellten Methoden. Das Prinzip entspräche einer allgemein bekannten Server-Client Architektur wo auf der einen Seite der TUI-Server existiert und auf der anderen Seite der Unity-Client welcher die Daten empfängt und bei Bedarf zurück zum Server senden kann. In der Praxis ist dies jedoch mit Problemen verbunden.

Ein TUI-Server beginnt erst mit dem Senden der Daten sobald ein TUI-Client sich mit ihm verbunden hat. Demzufolge müsste immer erst ein C++ TUI-Client sich mit dem Server verbinden damit der Serverdaten sendet. Dieses Problem ließe sich zwar beheben, dadurch wäre der TUI-Server aber nicht mehr in der Lage sein zu bestimmen welche TUI-Clients mit ihm verbunden wären. Außerdem würde die TUI-Server Konfiguration in dem ein Datenaustausch zwischen TUI-Server und TUI-Client geregelt werden kann, obsolet werden. Ein weiterer Punkt der gegen diese Methode spricht ist das Datentypen und Funktionen des TUI-Frameworks nicht im Unity Umfeld genutzt werden können. Die Datenpakete die vom TUI-Server über UDP gesendet werden, werden in der Unity-Umgebung als String-Pakete gelesen und müssen für die weitere Verwendung noch zusätzlich in Unity vorverarbeitet werden ehe man diese effektiv nutzen kann.

3.4 Fazit zu den Methoden

Alle drei Methoden wurden untersucht und im Ansatz getestet. Als Ergebnis bietet sich die Verwendung von Nativen Bibliotheken an. C++/CLR würde zwar den Vorteil bieten das komplette TUI-Framework mit in Unity3d verwendbar zu machen. Jedoch konnten die Probleme die in 3.1 erwähnt sind trotz umfangreichen Recherchen nicht beseitigt werden.

Eine Implementierung über UDP wurde zunächst testweise umgesetzt. Erfüllte jedoch nicht den Konform einer richtigen Integration des TUI-Frameworks. Vor allem fehlen die TUI-Datentypen und eine wirkliche Verbindung

4. Technische Vorbereitungen

4.1 TUI-Framework in 64Bit kompilieren

Mit Unity 5 wird der Editor nur noch ausschließlich als 64Bit Version angeboten. DLLs die als Plugin für ein Unity-Projekt verwendet werden sollen müssen dementsprechend auch in 64Bit vorhanden sein. Das TUI-Framework aber wird standardmäßig mit 32Bit kompiliert. Damit eine Schnittstelle zwischen den beiden Frameworks funktioniert muss das TUI-Framework in 64Bit vorliegen.

Alle Projekte des TUI-Frameworks sind abhängig von der pthread Bibliothek. Die pthread-Version die derzeit noch im TUI-Framework vorhanden ist, ist jedoch noch eine 32Bit-Version. Damit TUI in der 64Bit-Architektur lauffähig ist werden entsprechende 64Bit Binaries von pthread für TUI benötigt. Der Quellcode und die Binaries stehen im Internet frei zur Verfügung.[1]

Im TUI-Framework muss die Abhängigkeit von pthreadVCE2.lib zu der 64bit Version von pthreadVC2.lib umgeändert werden und die 64bit-DLL mit im Ordner der ausführbaren Programmdateien liegen.

Für die Schnittstelle wird in erster Linie der TUIFrameworkCore, TUIFrameworkClient und TUILibTUITypes benötigt. Alle drei sind als Visual Studio Projekte vorhanden. Dort muss die Zielarchitektur auf x64 umgeändert werden. Die Reihenfolge der Kompilierung muss beachtet werden. TUIFrameworkClient und TUILibTUITypes sind beide von TUIFrameworkCore abhängig. Demzufolge muss TUIFrameworkCore zuerst kompiliert werden. Die kompilierte Bibliothek muss anschließend in den anderen Projekten mit verlinkt werden.

4.2 .NET Framework Version

Eine weitere Besonderheit die es zu beachten gilt ist die Version des .NET-Frameworks von Unity. Mit VS 2015 liegt die derzeitige Version des .NET-Frameworks bei 4.6.1. Unity3d basiert aber in der aktuellen Version 5.3.2 auf Mono welches eine offene Implementierung des .NET Standards ist. Das verwendete Mono entspricht aber auch einer älteren Version womit nur die .NET Version bis 3.5 unterstützt wird. Beim Erstellen von managed Code muss deshalb darauf geachtet werden dass die .NET Version 3.5 ausgewählt ist. [5]

5. Implementierungs Ansatz

Damit die Funktionalitäten des TUI-Frameworks in Unity3d genutzt werden kann und ein Datenaustausch von TUI zu Unity3d möglich ist, müssen die Bibliotheken des TUI-Frameworks in einer für C# verständlichen Form bereitgestellt werden. Der Datenaustausch

[1] Die 64bit Binaries für PThread sind unter https://sourceforge.net/projects/pthreads4w/ zu finden

des TUI-Frameworks erfolgt von einem TUI-Server der die Daten bereit stellt und einem TUI-Client der sich mit dem TUI-Server verbindet und die Daten vom TUI-Server empfängt. Unity3d soll hier die Rolle eines TUI-Clients einnehmen um die Daten vom TUI-Server verarbeiten zu können.

Ein TUI-Client besteht in der Regel aus den Bibliotheken TUIFrameworkCore und TUIFrameworkClient für die Funktionalität und TUILibTUITypes um Zugriff auf die im TUI verwendeten Datentypen zu haben. Alle drei Bibliotheken sind notwendig um einen funktionierenden TUI-Client zu entwickeln. Damit in Unity3d diese drei Bibliotheken verwendet werden können wird, wie in 2.2 bereits vorgestellt, eine API benötigt die Zugang zu den Funktionalitäten zur Verfügung stellt.

Damit es übersichtlich bleibt und auch in anderen Unity3d-Projekten wiederverwendbar bleiben soll, wird die TUI/Unity-Schnittstelle auf C#-Seite zu einer DLL-Bibliothek zusammengefasst. Diese soll über den Aufruf der TUI-Frameworks API Zugang zu den Methoden des TUI-Frameworks erhalten.

Die C#-DLL soll beliebig in verschiedenen Unity3d-Projekten eingebunden werden können um auch dort nach selben Schema ein TUI-Client realisieren zu können. Abbildung 2 veranschaulicht den geplanten Aufbau.

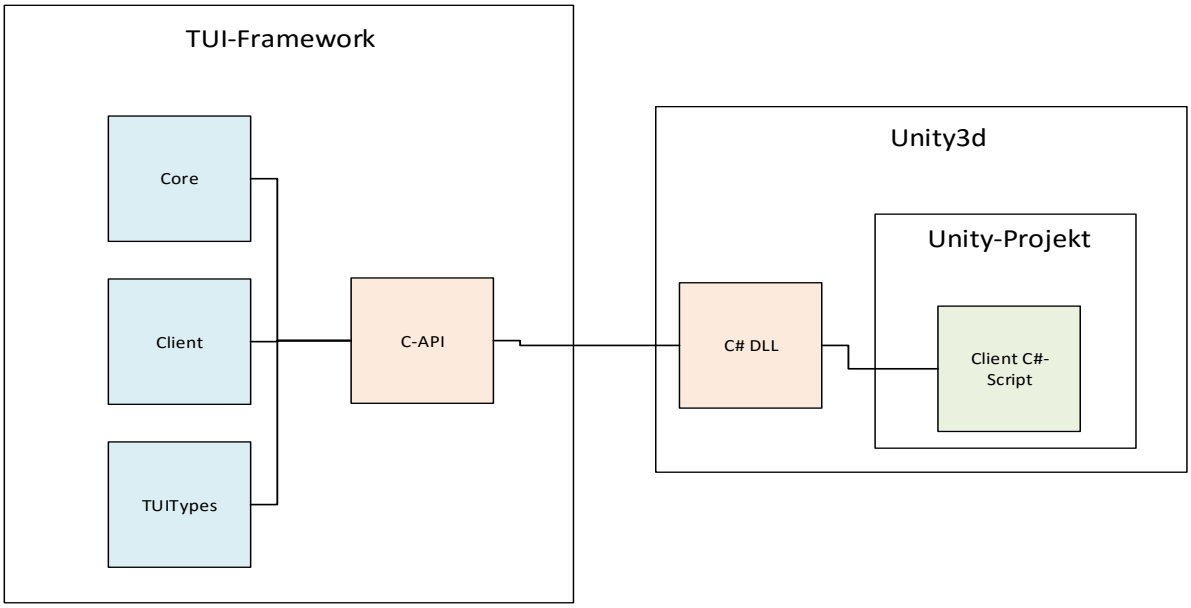

Abbildung 2 Geplanter Aufbau für die TUI/Unity-Schnittstelle

Die API des TUI-Frameworks muss Zugang zu den Funktionen aus den Drei Bibliotheken Core, Client und TUITypes bieten(hier blau markiert). Die bereitgestellten Funktionen der TUI-API müssen in der C# DLL exakt gleich benannt werden um eine richtige Zuordnung zu ermöglichen. Die C# DLL soll dann in eine beliebiges Unity3d Projekt eingebunden werden können um dort einen TUI-Client(hier grün markiert) in C# zu entwickeln. Die orange

markierten Entitäten in Abbildung 2 stellen die zu entwickelnden Komponenten dar die für eine erfolgreiche Realisierung des Projekts benötigt werden.

6. Implementierung

6.1 Implementation der TUI-API

Die TUI-API wird auf C++ Seite als DLL bereitgestellt. Damit die in der API definierten Methoden mittels DLL-Export von C# aus zugreifbar sind, mussten diese als C-Methoden deklariert werden. Neben der eigentlichen API beinhaltet die DLL auch noch 2 weitere Klassen welche in der folgenden Abbildung dargestellt sind.

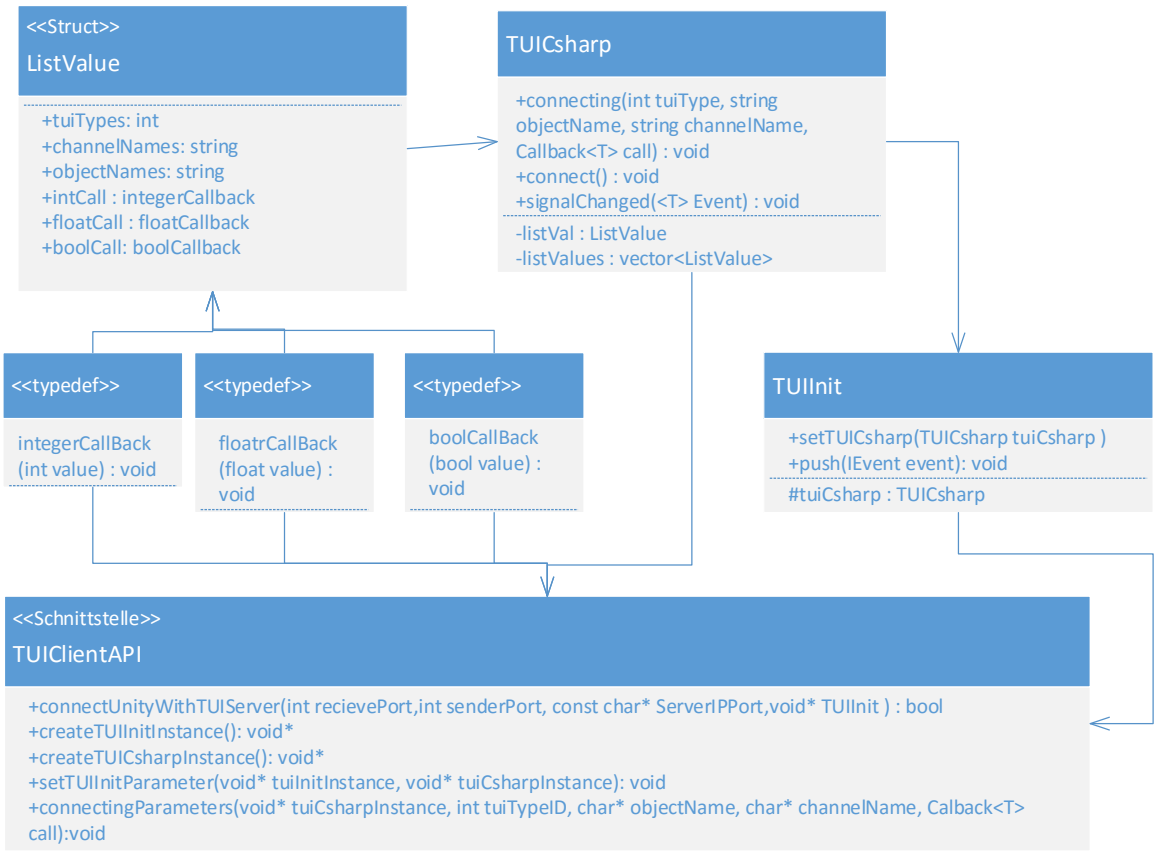

Abbildung 3 UML-Ansicht der C/C++ API des TUI-Frameworks

Die TUIClientAPI ist die Zentralle Schnittstelle zur Kommunikation mit dem TUI-Framework. Über die API können Instanzen der Klasse TUIInit und TUICsharp in C# erstellt werden. Die beiden Klassen sind für die Kommunikation innerhalb des TUI-Frameworks zuständig. C# Funktionen werden über die TUIClientAPI an Callback Funktionen in TUI übergeben. Jeder Datentyp hat seine eigene Callback-Funktion. Die Callback Funktionen werden zusammen mit den Parametern die zum Verbinden eines Channels mit dem TUI-Server in einer Struktur zusammengefasst. Die Callback Funktionen werden anschließend, sobald in TUI ein Event ausgelöst wird, in der TUICsharp-Klasse aufgerufen.

Die TUIInit-Klasse besitzt die Klasse TUICsharp als Attribut. Ein Objekt der TUIInit-Klasse wird beim Verbinden mit dem TUI-Server übergeben.

Kompiliert wird das Ganze als eine native C++ DLL.

6.2 Entwicklung der C#-DLL

Die C#-DLL beinhaltet die Schnittstelle zum TUI-Framework auf C#-Seite. Mit PInvoke kann die CLR automatisch auf die Funktionen innerhalb der erstellten TUI C++-DLL zugreifen, wenn die Funktionen gleich benannt sind und dieselben Parameter besitzen. Andernfalls muss ein Einstiegspunkt beim DLL-Import mit angegeben werden. Nachfolgend zeigt den Aufbau der C#-API die im Wesentlichen der C++ API entspricht.

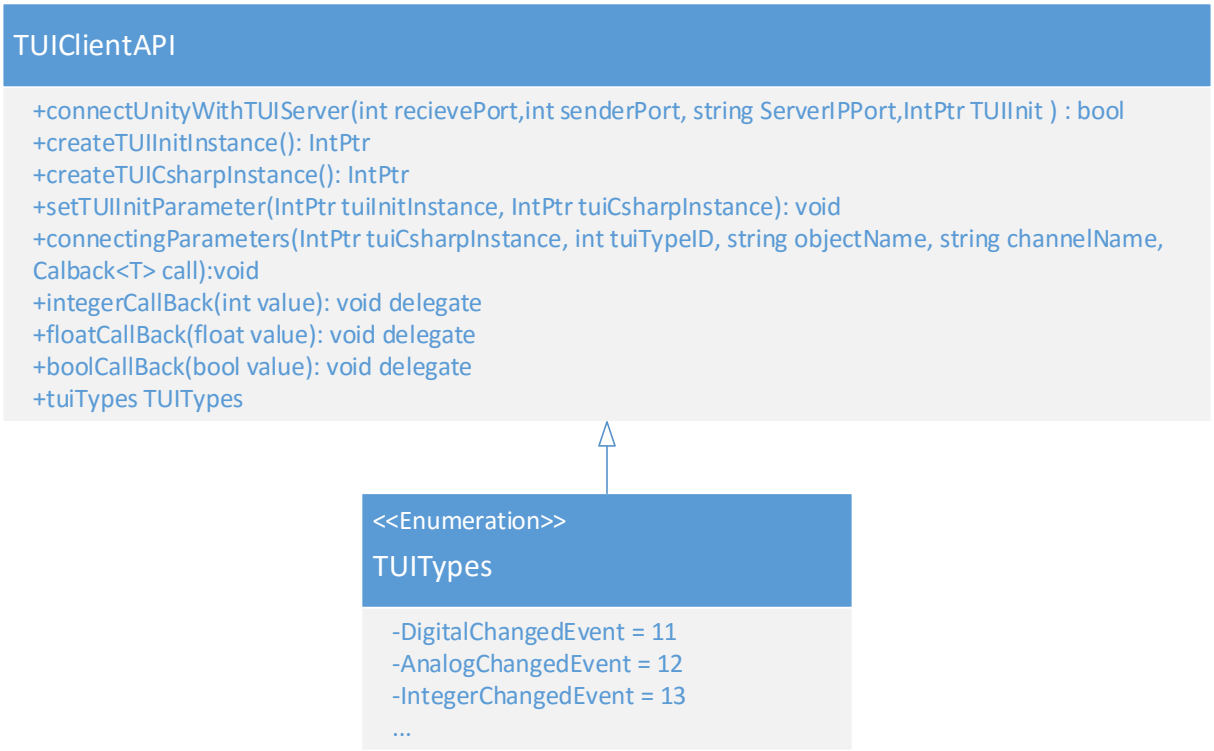

Abbildung 4 Überblick über die C#-API des TUI-Frameworks

Die TUITypes sind im einer Enumeration fest in der C#-API definiert. Der TUI-Type entspricht der TUITypeID welche im TUI-Framework fest definiert ist.

C++ Klassen die mit der C#-API erstellt wurden, werden in C# als Integer Pointer zurückgegeben. Der entsprechende Datentyp ist in C# IntPtr. Wenn Attribute in den C++ Objekten verändert werden sollen mit der C# API, muss der Pointer zur Klasse und der neue Wert mit an die C# Funktion übergeben werden. Dafür muss eine dementsprechende Funktion sowohl in der C# als auch in der C++ API vorhanden sein. Zum Beispiel wird in der Methode *createTUIInitInstance()* ein TUIInit Objekt in C++ erstellt und ein Pointer zu dem

Objekt zurückgegeben. Mit der *Funktion setTUIInitParameter(IntPtr tuiInitInstance, IntPtr tuiCsharpInstance)* wird das Objekt tuiCsharpInstance in tuiInitInstance neugesetzt.

Die Callback-Funktionen sind als Delegate in C# definiert. Delegate sind Referenzen zu Funktionen, das Äquivalent von Pointern in C++.[6]

Wie in C++ muss für jeden Dateityp ein eigener Delegate angelegt werden. Anschließend wurde das ganze Projekt mit Abhängigkeit zur C++ API-DLL als weitere DLL kompiliert.

6.3 Einbindung der C#-DLLs in Unity3d

Insgesamt sind nun 2 DLLs vorhanden. Damit Unity3d mit den DLLs arbeiten kann wird noch die pthread-DLL benötigt von der die C++DLL abhängig ist. Es werden also folgende DLLs für Unity benötigt:

pthreadVC2.DLL: Die 64Bit DLL von pthread, abhängigkeit von TUIClientAPI.DLL
TUIClientAPI.DLL: Die C++API DLL des TUI-Frameworks.
TUI-Client-Libary.DLL: Die C# API DLL des TUI-Frameworks.

Alle drei DLLs müssen in einem Plugin-Ordner innerhalb des Assert-Ordners gelegt werden. Zur Verwendung muss ein neues C#-Skript in Unity angelegt werden.

Wenn die Anwendung von Unity3d gebaut werden soll und eine .exe erzeugt wird müssen die DLL-Dateien im selben Ordner wie die .exe liegen.

Eine Beispiel Anwendung könnte in etwa wie auf der folgenden Seite aussehen:

Demo.cs

```
using UnityEngine;
using System.Net;
using System.Threading;
using System;
using UnityEngine.UI;
using TUIClientUnity_Lib;

public class Demo : MonoBehaviour {

    // start from unity3d
    public void Start()
    {
       Init();
    }

    public void Update()
    {
    }

    private void init()
    {
        receiveThread = new Thread(
            new ThreadStart(ReceiveData));
        receiveThread.IsBackground = true;
        receiveThread.Start();
    }

    private void ReceiveData()
    {
        int rPort = 8998;
        int sPort = 8999;
        string serverIP = "127.0.0.1:7999";
        // Erstellt eine TUI C#-Instanz und speichert diese als IntPtr
        IntPtr tuiUnityTest = TUIClientLibary.createTUICsharpInstance();

        // Erstellt eine TUIInit-Instanz und speichert diese als IntPtr
        IntPtr tuiUnityInit = TUIClientLibary.createTUIInitInstance();

        // Erstellt eine Unity C#-Instanz und übergibt diesen die TUI C#-Instanz
        TUIUnity tuiUnity = new TUIUnity(tuiUnityTest);

        // Setzt TUI C#-Instanz als Member für TUIInit-Instanz
        TUIClientLibary.setTUIInitParameter(tuiUnityInit, tuiUnityTest);

        Debug.Log(TUIClientLibary.connectUnityWithTUIServer(rPort, sPort, serverIP,
tuiUnityInit));
    }

}
```

Zunächst wird mit

```
using TUIClientUnity_Lib;
```

Die C#-API DLL eingebunden.
In der Start()-Methode wird bei Programmstart die Methode *Init()* aufgerufen die ihrerseits einen Thread startet. Das Starten eines separaten Threads ist notwendig da sonst das TUI-Framework den Editor blockieren würde, was dann nur noch mit beenden des Unity-Prozesses im Task-Manager behoben werden kann. In der *RecieveData()*-Methode des Threads wird folgender Code ausgeführt:

```
// Erstellt eine TUI C#-Instanz und speichert diese als IntPtr
 IntPtr tuiUnityTest = TUIClientLibary.createTUICsharpInstance();

// Erstellt eine TUIInit-Instanz und speichert diese als IntPtr
 IntPtr tuiUnityInit = TUIClientLibary.createTUIInitInstance();

// Erstellt eine Unity C#-Instanz und übergibt diesen die TUI C#-Instanz
 TUIUnity tuiUnity = new TUIUnity(tuiUnityTest);

// Setzt TUI C#-Instanz als Member für TUIInit-Instanz
 TUIClientLibary.setTUIInitParameter(tuiUnityInit, tuiUnityTest);

 Debug.Log(TUIClientLibary.connectUnityWithTUIServer(rPort, sPort, serverIP,
tuiUnityInit));
```

Zunächst werden zwei C++ Objekt mit *createTUICsharpInstance()* und *createTUIInitInstance()* erstellt. Beide werden jeweils in einem IntPtr gespeichert.
Dann wird eine C#-Objekt erstellt der als Parameter das TUICsharp Objekt übergeben wird.
Anschließend wird das TUICsharpObjekt in dem TUIInit-Objekt neu gesetzt mit setTUIInitParameter()

Anschließend erfolgt die Verbindung mit dem TUI-Server mit connectUnityWithTUIServer().

Das C#-Objekt, in dem Beispiel von der TUIUnity-Klasse, kann wie folgt aussehen:

TUIUnity.cs

```
using System;
using TUIClientUnity_Lib;
using UnityEngine;

public class TUIUnity
{

    public TUIUnity(IntPtr tuiUnityTest)
    {
        // Verbindet sich mit dem Channel
        TUIClientLibary.connectingParameters(tuiUnityTest,
            (int)TUIClientLibary.TUITypes.IntegerChangedEvent,
            "TUIUnity",
            "DeltaX",
            new TUIClientLibary.integerCallback(this.getXValue));

        TUIClientLibary.connectingParameters(tuiUnityTest,
            (int)TUIClientLibary.TUITypes.IntegerChangedEvent,
            "TUIUnity",
            "DeltaY",
            new TUIClientLibary.integerCallback(this.getYValue));

        TUIClientLibary.connectingParameters(tuiUnityTest,
            (int)TUIClientLibary.TUITypes.DigitalChangedEvent,
            "TUIUnity",
            "LeftMouseButton",
            new TUIClientLibary.boolCallback(this.getleftMouse));

    }

    public void getXValue(int value)
    {
        Debug.Log("XValues " + value);
    }

    public void getYValue(int value)
    {
        Debug.Log("YValues " + value);
    }

    public void getleftMouse(bool value)
    {
        Debug.Log("Left Mouse " + value);
    }

}
```

Auch in dieser Klasse muss zunächst wieder die C#-API eingebunden werden. Im Konstruktor der Klasse werden die Channels mit dem TUI-Server verbunden. TUIObject-Typename und Channelname müssen exakt mit denen in der TUI-Server XML Konfiguration übereinstimmen. Als letzter Übergabeparameter wird die C# Funktion übergeben die als Callback in C++ von TUI ausgeführt werden soll. In diesem Fall werden X und Y-Bewegungen, sowie die linke Maustaste vom TUI-Server abgefangen und zum Unity-C# Client geschickt.

7. Funktionstest

Als Testsystem wurde ein Asus Zenbook mit Intel I7 Prozessor, 10GB Arbeitsspeicher und 256GB SSD verwendet. Das verwendete Betriebssystem ist Windows 10 Education. Programmiert und Debuggt wurde in Visual Studio 2015 Enterprise für C++ und C#. Unity wurde in der Version 5.3.1 verwendet.

Als Funktionstest wurde eine Beispielanwendung unter Unity entwickelt, welche die ΔX und ΔY-Werte der Maus vom TUI-Server empfängt und damit einen Würfel rotieren lässt.

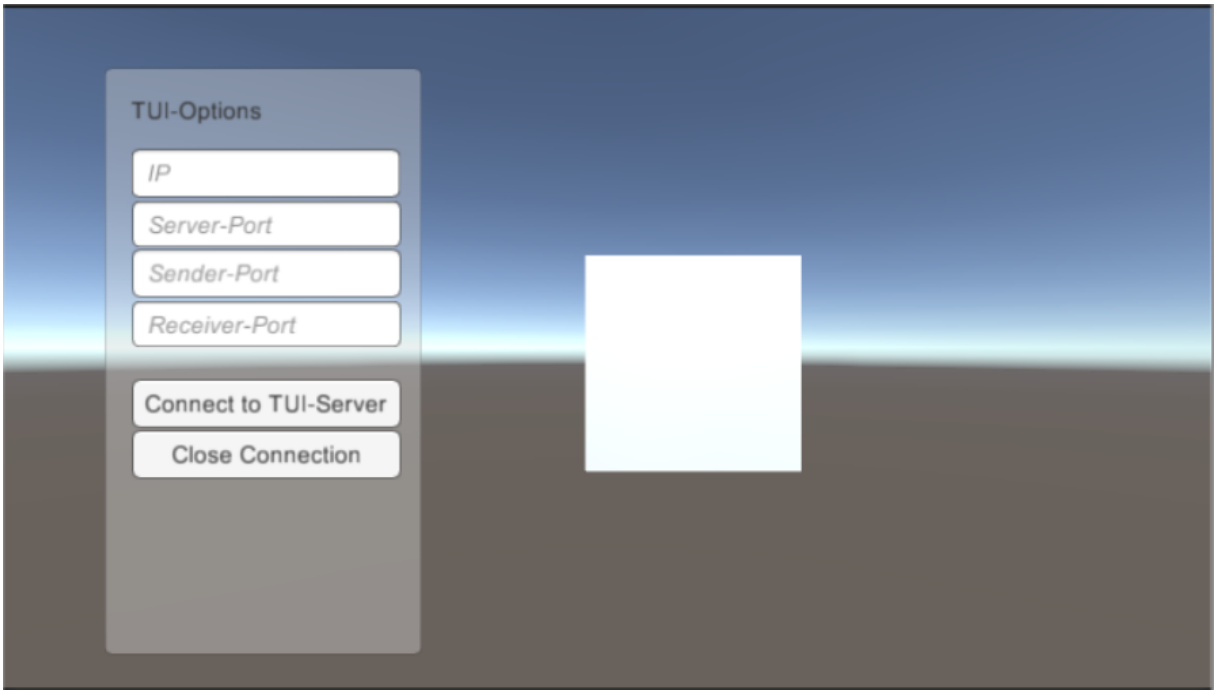

Abbildung 5 Screenshot der Test-Anwendung in Unity3d

Der TUI-Server muss mit einer entsprechenden Konfiguration gestartet werden. Die nachfolgende Abbildung zeigt die Server-Konfiguration im Konfigurations-Editor, welcher mit im TUI-Framework enthalten ist. Mouse1 ist ein TUI Device. Damit wird der TUI-Server angewiesen das Maus-Plugin in der TUI-Architektur zu laden um Maus-Daten empfangen zu können. Alle Maus-Daten werden im MouseChannel gekapselt und zum MSP-Type MouseDemux1 geschickt wo die Daten aufgeteilt werden. Für den Test wurden nur die ΔX und ΔY-Werte der Maus benötigt weswegen diese mit TUIObjectType TUIUnity verbunden sind.

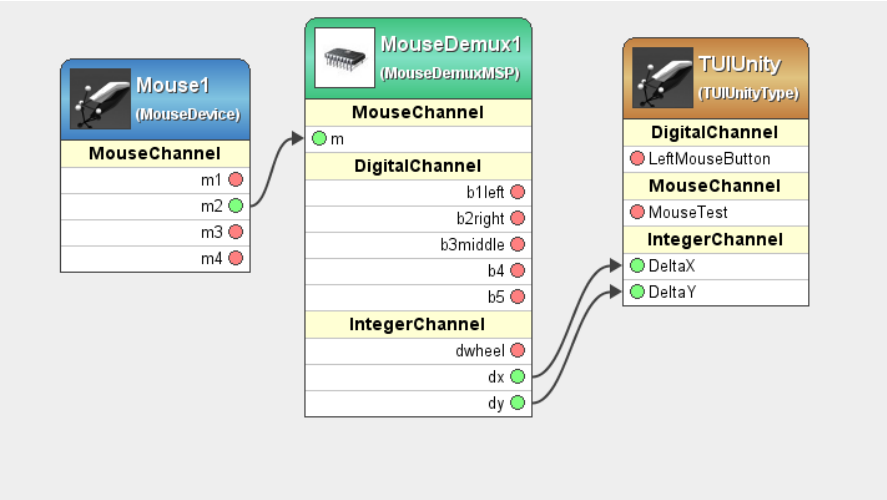

Abbildung 6 TUI-Server Konfiguration für das Unity-Beispiel

Die dargestellte Konfiguration lässt sich als XML-Datei abspeichern und wird dem TUI-Server als Parameter mit übergeben.

Über das User Interface wurden die Parameter zur Verbindung des TUI-Servers eintragen. Eine Meldung auf dem TUI-Server zeigt eine erfolgreiche Verbindung von der Unity3d-Anwendung mit dem TUI-Server.

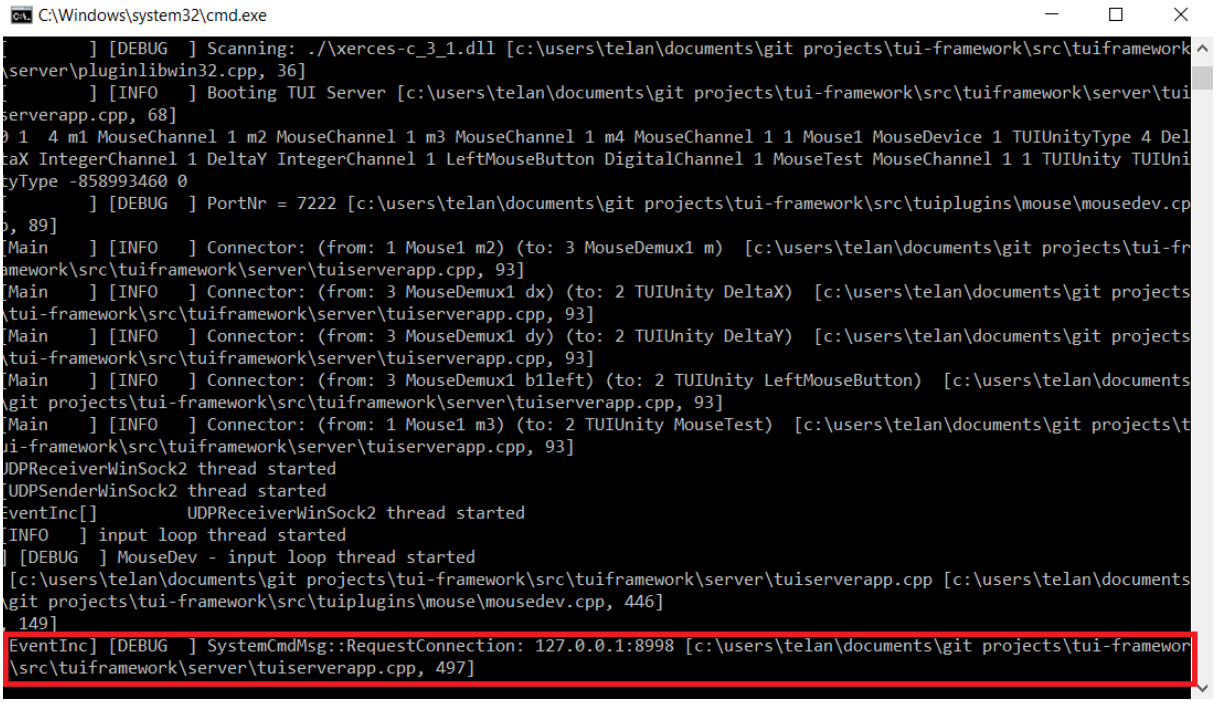

Abbildung 7 Ausgabe des TUI-Servers der rot markierte Bereich zeigt das Unity sich mit dem TUI-Server verbunden hat.

Damit der TUI-Server Maus-Bewegungen empfangen kann wird noch der im TUI-Framework enthaltene Mouse Server benötigt damit die Maus-Daten nicht zum Betriebssystem sondern zum TUI-Server gesendet werden.

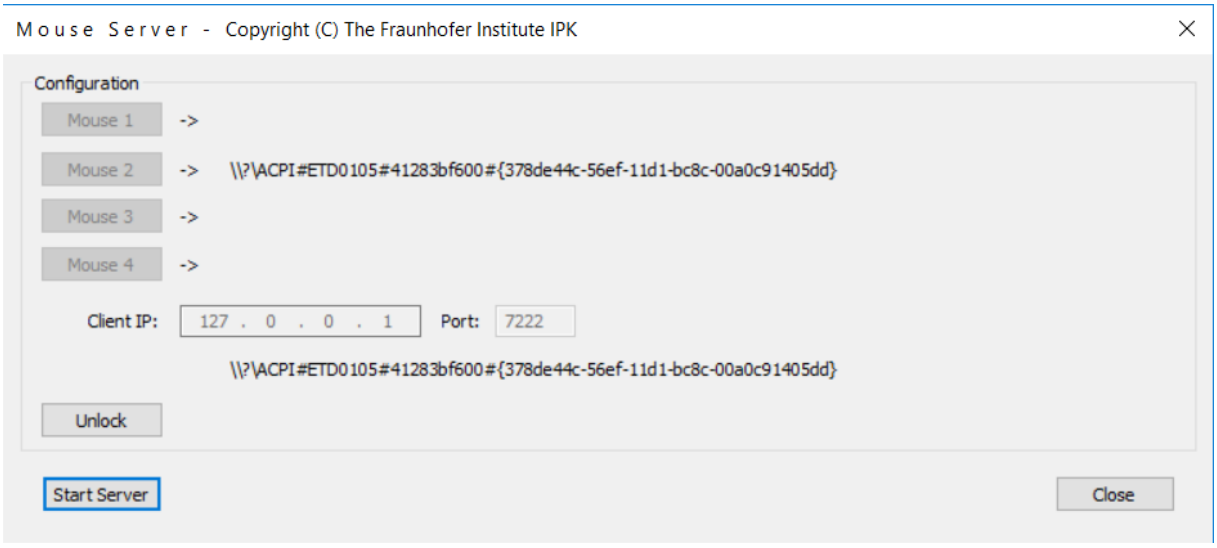

Abbildung 8 Interface des Mouse Servers. Die Maus ist auf Position 2 eingestellt und ClientIP entspricht der des TUI-Servers. Als Port muss der Port angegeben werden der in der TUI-Server Konfiguration unter Mouse1 definiert ist.

Wenn der Server gestartet ist, ist es nicht mehr möglich mit dem Betriebssystem mit der Maus zu interagieren. Die Maus-Daten werden nun direkt zum TUI-Server gesendet der die Daten weitersendet zur Unity3d-Anwendung.

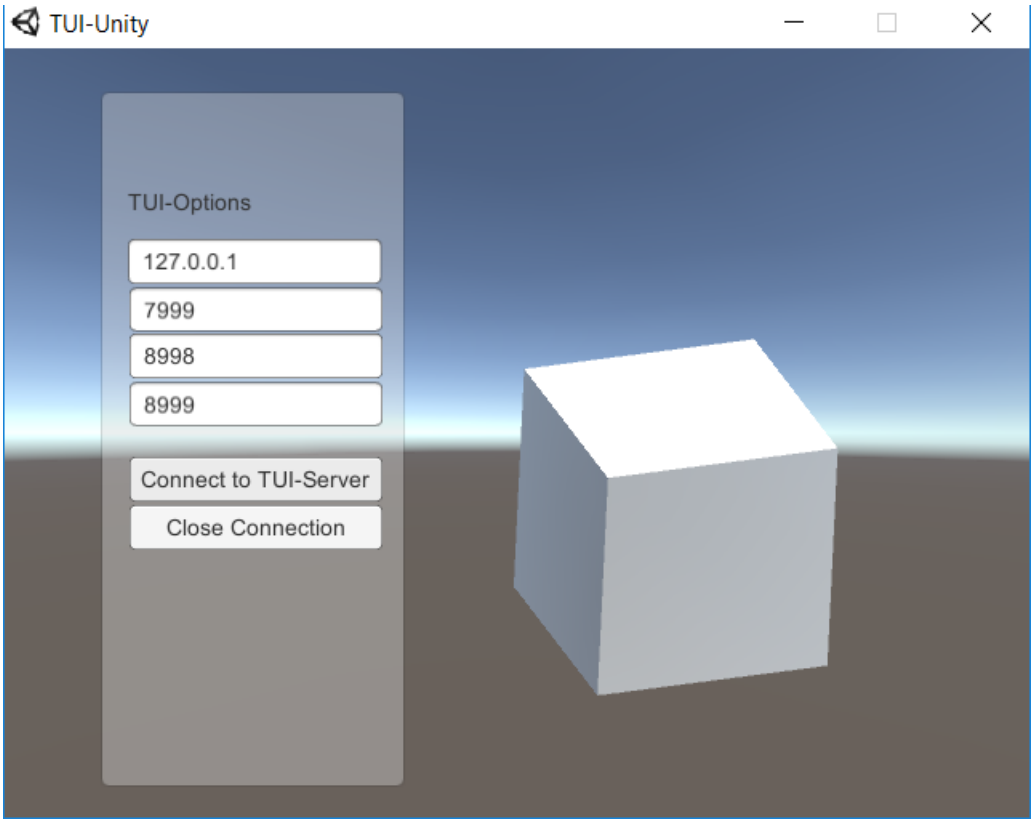

Abbildung 9 Die Maus-Daten werden von Unity empfangen und lassen den Würfel rotieren

Es sei angemerkt dass die Unity3d-Anwendungen die Eigenschaft haben die 3D-Szene nur zu aktualisieren wenn diese den Fokus besitzen. Durch das Starten des Mouse Server erhält der Mouse Server den Fokus während die Unity3d-Anwendung in den Hintergrund gerückt wird. Erst wenn der Mouse-Server beendet wurde und die Unity3d-Anwendung den Fokus zurück erhält beginnt der Würfel zu rotieren.

8. Fazit

Mit der geschaffenen Schnittstelle können müssen zukünftig nur die DLL-Dateien vom TUI-Server in den Assert Ordner von Unity3d reinkopiert werden. Mit den DLL-Dateien kann dann in Unity3d ein TUI-Client, mit der bereitgestellten API, implementiert werden. Damit kann dann vom Unity3d-Projekt unabhängig das TUI-Framework genutzt werden. Die DLL-Dateien lassen sich auch ohne Unity3d in ein normalen C#-Projekt verwenden womit ein universeller Einsatz möglich ist unter anderem auch in die CAVE.

Zum Zeitpunkt des Verfassung dieser Dokumentation befindet sich die C#/C++ TUI-API noch auf dem Stand eines Prototypen. Bei der Entwicklung sind Fehler aufgefallen die im TUI-Framework noch behoben werden müssen. Darunter zählt das Trennen der Verbindung zum TUI-Server welche derzeit nicht funktionstüchtig ist. Wird im Unity-Editor die laufende Anwendung beendet bleibt die UDP-Verbindung aber erhalten, wenn die Anwendung erneut gestartet wird und versucht wird sich mit dem TUI-Server zu verbinden beendet sich der Unity-Editor.

Bisher ist eine Datenkommunikation nur mit den primitiven Datentypen int, float und boolean möglich. Zukünftig soll es, neben weiteren Datentypen, auch möglich sein komplexere Datenstrukturen des TUI-Frameworks in C# nutzbar zu machen. Des Weiteren ist es derzeit nur möglich Daten vom TUI-Framework in C# ausgeben zu lassen. Das Versenden von Daten von C# zum TUI-Framework ist derzeit noch nicht möglich. Zukünftig soll es möglich sein Daten die in Unity3d verarbeitet wurden zurück zum TUI-Server zu senden.

Quellen

[1] Israel, J. H., Belaifa, O.(2013), *TUI-Framework zur Intergation begreifbarer Objekte in interaktive Systeme*. Mensch & Computer 2013 – Workshopband (S. 201-206). Bremen, Deutschland

[2] International Organization for Standardization(2012), *Common-Language Infrastructure – ISO 23271* http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=58046 [Zuletzt aufgerufen am 23.03.2016]

[3] Kerr, K. (2004*), C++: The Most Powerful Language for .NET Framework Programming* https://msdn.microsoft.com/en-us/library/ms379617%28v=vs.80%29.aspx [Zuletzt aufgerufen am 28.03.2016]

[4] Unity3d 5.3 API-Reference(2016), *Plugins* http://docs.unity3d.com/Manual/Plugins.html [Zuletzt aufgerufen am 28.03.2016]

[5] Hauwert R. (2014), *The Future of Scripting in Unity* http://blogs.unity3d.com/2014/05/20/the-future-of-scripting-in-unity/ [Zuletzt aufgerufen am 28.03.2016]

[6] Microsoft C# Reference (2015), *Delegat* https://msdn.microsoft.com/de-de/library/900fyy8e.aspx [Zuletzt aufgerufen am 28.03.2016]