BAUHAUS UNIVERSITÄT WEIMAR

FAKULTÄT MEDIEN
Lehrstuhl für virtuelle Realität

und

FRIEDRICH-SCHILLER UNIVERSITÄT JENA

FAKULTÄT FÜR MATHEMATIK UND INFORMATIK
Lehrstuhl für Technische Informatik

STUDIENARBEIT

„AerionInput - Ein Treiber für den GlobeFish"

Vorgelegt von:	Christian Bayer		
geboren am:	22. August 1984	in:	Aschersleben
Matrikel:	84543		
Email:	chrbayer@minet.uni-jena.de		

Betreuer:

Dipl. Des. Alexander Kulik

Dipl. Inf. (FH) Jan Hochstrate

Dr. Wolfgang Koch

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich die von mir eingereichte Studienarbeit zum Thema

„AerionInput - Ein Treiber für den Globefish"

vollkommen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Jena, den 30.11.2008

Unterschrift

Kurzfassung

Seit Jahren existieren Computerprogramme, wie zum Beispiel CAD-Programme und 3D-Spiele, mit denen versucht wird, eine dreidimensionale Welt abzubilden. Hier sind große Fortschritte bei der Darstellung der Wirklichkeit gemacht worden, jedoch ist die Navigation in diesen 3D-Welten (noch) nicht intuitiv möglich. Der GlobeFish oder Aerion füllt diese Lücke aus. Dieses Eingabegerät stellt wie die bislang existierenden Geräte die Translation elastisch zur Verfügung, die Rotation jedoch wird intuitiv und direkt abgebildet und die Drehung der Kugel wird nicht wie sonst üblich über eine elastische Bewegung errechnet. Damit können Rotation und Translation vom Benutzer einfacher eingegeben werden, da die drei Achsen voneinander getrennt sind.

Bisher wurde der Aerion mit speziellen Avango-Plugins angesteuert, die leider nur in Avango funktionierten und daher nur zu Demonstrationszwecken einsetzbar waren. Zielstellung dieser Arbeit ist es daher, einen „Treiber"[1] zu erstellen, der die Übermittlung der Daten zwischen Aerion und Anwendung übernimmt. Treiber in Anführungszeichen, da ein Treiber im engeren Sinne ja eigentlich im Kernel-Space läuft, AerionInput jedoch vollständig Userspace-basiert ist.

[1] „Treiber" in Anführungsstrichen, da ein klassischer Treiber im Kernelspace, AerionInput jedoch vollständig im Userspace abläuft. Daher ist AerionInput im engeren Sinne kein Treiber, sondern eher eine Bibliothek oder ein Framework zur Unterstützung des Aerion.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Motivation 1
1.2 Analyse des 3DConnexion-Treibers 2
1.3 Einführung in das Component Object Model (COM) 4

2 Design von AerionInput **8**
2.1 Struktur 8
2.2 Event-System 10
2.3 Die Device-Klasse 12
2.4 Die Sensor-Klasse 14
2.5 Die Keyboard-Klasse 18
2.6 Die AngleAxis-Klasse 19
2.7 Die Vector3D-Klasse 20
2.8 Die Preferences-Klasse 20
2.9 Funktionsweise des Treibers am Beispiel des Blender-Plugins 21
2.10 AerionInputConfig 26

3 Ausblick **27**

A Anhang **28**
A.1 Klassen- und Methodenreferenz AerionInput 28
A.1.1 ISimpleDevice 28
A.1.1.1 Properties 28
A.1.1.2 Methods 28
A.1.2 IKeyboard 29
A.1.2.1 Properties 29
A.1.2.2 Methods 29
A.1.3 Vector3D 29
A.1.3.1 Properties 30
A.1.3.2 Methods 30
A.1.4 IVector3D 30
A.1.4.1 Properties 30
A.1.5 functions 30
A.1.6 Preferences 31
A.1.6.1 Fields 31
A.1.6.2 Properties 31
A.1.6.3 Methods 32
A.1.7 _ISimpleDeviceEvents_DeviceChangeEventHandler 32
A.1.8 _ISimpleDeviceEvents 32
A.1.8.1 Methods 32

A.1.9 ITDxInfo 33
A.1.9.1 Methods 33
A.1.10 IAngleAxis 33
A.1.10.1 Properties 33
A.1.11 func 33
A.1.12 Sensor 34
A.1.12.1 Fields 34
A.1.12.2 Properties 35
A.1.12.3 Methods 36
A.1.13 ISensor 38
A.1.13.1 Properties 38
A.1.14 Device 38
A.1.14.1 Fields 38
A.1.14.2 Properties 39
A.1.14.3 Methods 39
A.1.15 _ISensorEvents_SensorInputEventHandler 41
A.1.16 _ISensorEvents 41
A.1.16.1 Methods 42
A.1.17 _IKeyboardEvents_KeyDownEventHandler 42
A.1.18 _IKeyboardEvents_KeyUpEventHandler 42
A.1.19 _IKeyboardEvents 42
A.1.19.1 Methods 42
A.1.20 TDxInfo 42
A.1.20.1 Methods 43
A.1.21 Keyboard 43
A.1.21.1 Fields 43
A.1.21.2 Properties 43
A.1.21.3 Methods 44
A.1.22 AngleAxis 45
A.1.22.1 Properties 45
A.1.22.2 Methods 45
A.2 GoogleEarth COM-API 45
A.2.1 COM-API 45

1 Einleitung

1.1 Motivation

Das Ziel dieser Arbeit bestand darin, eine API[1] für den Aerion zu entwickeln, die folgende Anforderungen erfüllt:

1. Möglichkeit der Verbindung des Geräts mit beliebigen Anwendungen
2. Verwendung einer API, die zu bestehenden Anwendungen mit Unterstützung für 3D-Eingabegeräte eine möglichst große Ähnlichkeit hat bzw. idealerweise binärkompatibel zu einer solchen ist
3. daraus ergibt sich die Verwendung der Windows-Plattform, da für diese die meisten Anwendungen, mit denen Aerion zusammenarbeiten soll, existieren
4. Verwendung etablierter Standards und bekannter Schnittstellen sowie bereits existierender Bibliotheken, um schnell einen Prototypen erzeugen zu können (Rapid Prototyping)

Bei genauerer Betrachtung von Punkt 2 fällt sofort die API des Marktführers 3DConnexion ins Auge, zumal diese eine Beispielimplementierung für die Anbindung einer OpenSource-Anwendung enthält, nämlich Blender. Blender ist GPL, daher musste die Anwendungsanbindung für den Treiber von 3DConnexion mindestens unter LGPL gestellt und veröffentlicht werden.

Punkt 4 gab schließlich den Anstoß, AerionInput in Visual C# zu implementieren, einer eigentlich eher ungewöhnlichen Sprache für einen „Treiber”. Die .NET-Plattform im Allgemeinen und die C#-Sprache im Besonderen bieten jedoch aufgrund ihrer starken Typisierung und der klaren Trennung zwischen sicherem und unsicherem Code die ideale Plattform, um ohne zeitraubendes Debuggen von Pufferüberläufen und Zeigern eine prototypische Implementierung eines Frameworks zu entwickeln, die tatsächlich Daten vom Aerion abholt und in eine Anwendung übergibt.

Weiterhin lag bereits eine Abstraktionsbibliothek für das USB HID-Protokoll vor, die HidLibrary.Net von Roman Reichel.[2] Diese Bibliothek wurde an der Bauhaus Universität Weimar geschrieben, um rohe HID-Reports des Aerion abfangen zu können. Diese Bibliothek ist ebenfalls in C# geschrieben und steht im Quelltext zur Verfügung.

Diese Fakten machte bei der Entwicklung des Treibers vieles leichter, da der vollständige Quelltext aller beteiligten Anwendungen und Komponenten zur Verfügung stand.

Damit blieb nur noch die Aufgabe, die „Business-Logik” zu schreiben, also ein Framework zu erstellen, das dieselbe API wie der Treiber von 3DConnexion implementiert und die Bibliothek HidLibrary.Net als Abstraktion für USB-HID Reports einbindet:

[1] API: Application Programming Interface, Anwendungsprogrammierschnittstelle
[2] HidLibrary.Net http://sourceforge.net/projects/hidlibrary/

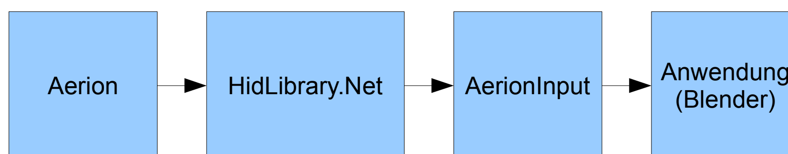

Abb. 1.1: Datenübergabeschema

Bei der Wahl der Lizenz bot sich die LGPL 2.1[3] an, eine freie Software-Lizenz, die für Bibliotheken genutzt werden kann, welche auch unfreie Programme einbinden. Da die überwiegende Mehrheit der Anwendungen, die Aerion in Zukunft unterstützen soll leider unter proprietären Lizenzen stehen (und sich das auf absehbare Zeit wohl auch nicht ändern wird), kann beispielsweise die GPL nicht verwendet werden, weil sie das Linken von unfreien Programmen verbietet.

Weiterhin wurde im Verlauf der Arbeit an diesem Projekt bekannt, dass eine API-kompatible OpenSource-Implementierung des 3DConnexion-Treibers bereits existiert, jedoch für POSIX-kompatible Plattformen wie BSD oder Linux. Eine Anpassung dieses Treibers für eine Unterstützung des Aerion ist geplant und höchstwahrscheinlich unkompliziert. Die Lücke des fehlenden Windows-Ports dieses Projekts soll geschlossen werden, indem diese Arbeit in das „Free Spacenav"-Projekt[4] integriert wird.

1.2 Analyse des 3DConnexion-Treibers

Der Treiber von 3DConnexion ist zu einer Reihe von CAD-, Design und Virtual Reality-Anwendungen kompatibel. Eine diesem Treiber ähnliche Implementierung erscheint also sehr sinnvoll. Leider ist er nicht quelloffen, so dass eine Anpassung dieses Treibers an unser Gerät nicht möglich ist. Daher blieb keine andere Möglichkeit, als einen eigenen „Treiber" zu erstellen. Je ähnlicher sich jedoch die von beiden Treibern verwendeten APIs sind, desto einfacher ist es für Software-Hersteller, die bereits Support für 3DConnexion-Geräte in ihre Programme eingebaut haben, den Aerion zu unterstützen. Da die Integration des Treibers in das „Free Spacenav"-Projekt geplant ist, wurde der Treiber mit dem Fokus auf Binärkompatibilität entwickelt.

Der Treiber von 3DConnexion besteht aus zwei Teilen. Zum einen aus einer Bibliothek, welche von einer Anwendung geladen werden kann. Die Anwendung „kennt" also den Treiber und lädt ihn „aktiv" - also von sich aus. Die Daten werden - bildlich gesprochen - aus dem Treiber „gezogen" (daher „Pull" im Bild).[5] Blender verwendet eben diese Art der Kommunikation mit dem Treiber.

[3] LGPL 2.1 http://www.gnu.org/licenses/lgpl-2.1.html
[4] Free Spacenav Projekt http://spacenav.sourceforge.net/
[5] In Wirklichkeit müssen vorher allerdings sog. Ereignis-Senken registriert werden, die Datenübergabe wird aber tatsächlich vom Treiber ausgelöst.

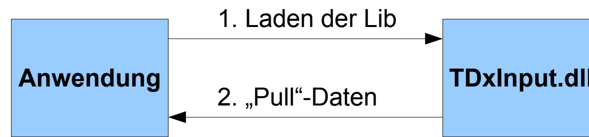

Abb. 1.2: Pull-Komponente

Die andere Möglichkeit ist die Fernsteuerung einer Anwendung. Dazu muss die Anwendung den Treiber nicht „kennen” und selbst laden. Die Daten werden also vom Treiber in die Anwendung „gedrückt” (daher „Push” im Bild). Es hat den Anschein, dass GoogleEarth nach dieselben Mechanismen zugrunde liegen. Dafür muss der Quelltext einer Anwendung nicht bekannt sein. Eine solche Fernsteuerung kann ohne Neukompilieren der Anwendung eingerichtet werden. Jedoch muss dazu die Anwendung ihre eigene Fernsteuerung per COM unterstützen.

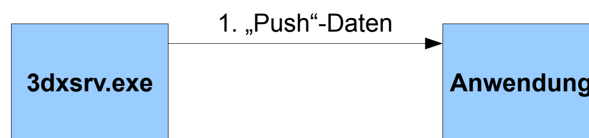

Abb. 1.3: Push-Komponente

Implementiert wurde die „Pull”-Methode, da aufgrund der Verfügbarkeit des Quelltextes von Blender eine schnelle Überprüfung der einzelnen Entwicklungsschritte möglich war. Eine Implementierung der „Push”-Komponente erscheint mit der Verfügbarkeit der „Pull”-Komponente jedoch recht einfach, da die Erstgenannte nur noch die fertigen Daten aus der „Pull”-Komponente an fernsteuerbare Anwendungen übergeben muss:

Abb. 1.4: Push- und Pull-Komponenten im 3DConnexion-Treiber TDxInput

1.3 Einführung in das Component Object Model (COM)

Das COM-Framework[6] von Microsoft ist eine native Windows Klassenbibliothek. COM ist Teil des Betriebssystems und stellt Schnittstellen und Verfahren bereit, um Objekte für objektorientierte Programmiersprachen zur Verfügung zu stellen. Dabei können einzelne Objekte instantiiert, ihre Methoden aufgerufen und Ereignisse oder Events der Objekte abgefangen werden. Die Abbildung auf die Objektorientierung „hinkt” nur an einer Stelle - Propertys oder Eigenschaften können nicht direkt ausgelesen werden, sondern müssen durch Methoden gekapselt sein. Das stellt aber kein Problem dar, da objektorientierte Programmiersprachen wie C++ oder C# intern auch mit Funktionen arbeiten, die eine (interne) Eigenschaft auslesen und auf diese ebenfalls nicht „direkt” zugreifen können. In Visual C# reicht es aus, öffentlich sichtbaren Attributen (private Attribute können ohnehin nicht exportiert werden) eine `get()`- und/oder eine `set()`-Methode zuzuordnen. Dabei handelt es sich ja um besagte Methoden, die von Visual C# auch korrekt in eine `get_PropertyName()`- bzw. `set_PropertyName()`-Funktion umgewandelt werden.

Klassen werden im COM durch ihren Namen und eine eindeutige ID, den „Globally Unique Identifier” (GUID), unterschieden. Diese 36-stelligen Zahlen werden von Windows zentral registriert und verwaltet. Da das 3DConnexion-Framework objektorientiert ist und per COM seine Schnittstellen offenlegt, muss ein binärkompatibler Treiber folglich dieselben Schnittstellen mit denselben GUIDs registrieren. Dies geschieht mithilfe von so genannten Attributen, die in eckigen Klammern in den Quelltext eingebettet werden:

```
[Attributname(Parameter)]
```

Mit dem GUID-Attribut kann man nun die GUID einer Klasse oder eines Interface festlegen:

```
[Guid("XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX")]
```

In Visual C# ist es sehr einfach, bestehende Klassen per COM zu exportieren. Ein Haken vor gleichlautender Option im Eigenschaftendialog macht alle Klassen des Projekts COM-sichtbar. Klassen, die explizit nicht per COM sichtbar sein sollen, also z.B. interne öffentliche Klassen oder Klassen, die das originale Framework nicht bereit stellt, können mit dem Attribut

```
[ComVisible(false)]
```

von der Veröffentlichung ausgeschlossen werden. Die Exposition von Klassen per COM hält jedoch unter Visual C# einige Fallstricke bereit. Zu jeder Klasse wird ein so genanntes **Klasseninterface** erzeugt, welches sämtliche Methoden der Klasse beschreibt. Auch dieses Interface hat eine GUID, die ebenfalls die GUID des 3DConnexion-Pendants tragen muss. Um die GUID für ein solches Klasseninterface von Hand anzugeben, muss das Interface dazu manuell erzeugt und der Compiler angewiesen werden, das standardmäßig erzeugte (und quasi unsichtbare) Klasseninterface nicht zu benutzen:

```
[ClassInterface(ClassInterfaceType.None)]
```

[6] COM http://www.microsoft.com/com

Eine korrekt exportierte Klasse sieht also wie folgt aus:

```
[Guid("512A6C3E-3010-401B-8623-E413E2ACC138"),
ClassInterface(ClassInterfaceType.None)]
public class AngleAxis : IAngleAxis {
...
}
```

und das entsprechende Interface so:

```
[Guid("1EF2BAFF-54E9-4706-9F61-078F7134FD35"),
InterfaceType(ComInterfaceType.InterfaceIsDual)]
public interface IAngleAxis {
..
}
```

Über COM wird eine Klasse nie direkt angesprochen, es wird immer nur über das ihr zugeordnete Interface auf sie zugegriffen!

Das Attribut

```
[InterfaceType(ComInterfaceType.InterfaceIsDual)]
```

beschreibt, dass das Interface als „Dual" exportiert werden soll. Das Interface implementiert die beiden Interfaces `IUnknown` und `IDispatch`. Diese beiden Interfaces beschreiben, wie auf Methoden der Klasse zugegriffen werden kann. Bei `IUnknown` implementierenden Klassen wird eine so genannte frühe Bindung der Methoden der Klasse durchgeführt. Frühe Bindung bedeutet, dass bei der Compile-Zeit bereits alle Typen und Einsprungadressen für die Bibliotheken bekannt sein müssen. Das funktioniert jedoch nicht bei Skriptsprachen, die keine Typen kennen, wie zum Beispiel Visual Basic Script. Dafür gibt es das `IDispatch`-Interface. Hier wird erst zur Laufzeit von einem Objekt abgefragt, welche Methoden es unterstützt. Daher spricht man bei dieser Art des Aufrufs auch von später Bindung.

Die Methoden, die ein `IUnknown`-implementierendes Interface exportiert, müssen nummeriert werden. Mit dem Attribut

```
[DispId(1)]
```

wird die Reihenfolge in einer so genannten VTable festgelegt. Auch die genaue Reihenfolge der Nummerierung muss für einen binärkompatiblen Ersatz beibehalten werden.

Jede Methode wird als Funktionszeiger übergeben. Ihr Aufruf erfolgt wie bei Funktionen üblich. Rückgabewerte der Methode können jedoch nicht wie in normalen Programmiersprachen als Resultat der Operation ausgelesen werden, denn alle COM-Funktionen haben den Typ HRESULT. Darin wird der Statuswert der Operation als 8-Bit-Zahl ausgegeben, die einen großen Bereich von möglichen Fehlern oder Resultaten abdeckt, also z.B. S_OK für Erfolg. Hat eine Funktion in einen nicht-void-Rückgabetyp, so wird eine Variable als Pointer in die Parameterliste des Aufrufs eingebettet. Die Funktion wird also mit einem Puffer aufgerufen, welcher nach dem Aufruf den tatsächlichen Rückgabewert der Funktion enthält.

In der Anwendung (wie z.B. Blender) sieht also ein Aufruf einer Funktion aus AerionInput wie folgt aus:

```
int* result = 0;
HRESULT = CComPtr->MethodName(int* result, int* par1, int* par2);
...
function(result);
...
```

Damit in der Definition der Methode klar ist, in welcher Richtung ein Parameter benutzt werden soll (also ob er Übergabe- oder Rückgabewert ist), gibt es die Attribute

```
[param: Out]
```

für einen Rückgabewert und

```
[param: In]
```

für einen Übergabewert.

Um **Ereignisse** oder **Events** zu exportieren, ist unter Visual C# folgendes zu beachten:

1. Eine Klasse kann Events nur über ein spezielles Event-Interface zugänglich machen, welches die Klasse implementieren muss.
2. Jedoch wird bei der Deklaration der Klasse nicht wie sonst üblich die Implementierung der Klasse durch das Implements-Schlüsselsymbol „:" angeben, sondern mit dem Attribut

   ```
   [ComSourceInterfaces(typeof(_ISimpleDeviceEvents))]
   ```

3. Events müssen dann in der entsprechenden Klasse deklariert werden

   ```
   public event
     _ISimpleDeviceEvents_DeviceChangeEventHandler DeviceChange;
   ```

4. Der Typ des Events oben ist `_ISimpleDeviceEvents_DeviceChangeEventHandler`. Dieser Typ muss ebenfalls COM-sichtbar sein:

   ```
   [ComVisible(true)]
   public delegate void
     _ISimpleDeviceEvents_DeviceChangeEventHandler(int reserved);
   ```

   Bei diesem Typ handelt es sich um einen so genannten Delegaten oder Funktionszeiger. Ein Delegat gibt lediglich die Signatur einer Funktion an, also den Typ des Vorgabewertes und die Typen der Parameter.
5. Die Namen obiger Delegaten müssen auch hier mit den Namen des 3DConnexion-Treibers übereinstimmen.

Eine korrekt implementierte Klasse, die Events per COM freigibt, hat folgende Gestalt:

```
[ComVisible(true)]
public delegate void
  _ISimpleDeviceEvents_DeviceChangeEventHandler(int reserved);

[Guid("82C5AB54-C92C-4D52-AAC5-27E25E22604C"),
 ComSourceInterfaces(typeof(_ISimpleDeviceEvents)),
 ClassInterface(ClassInterfaceType.None)]
public class Device : ISimpleDevice, IDisposable {
...
  private EventHandler
    m_HidControlEvents_DeviceChangeEventHandler;
...
}
```

2 Design von AerionInput

2.1 Struktur

AerionInput besteht im Gegensatz zum 3DConnexion-Treiber nur aus der „Pull"-Komponente, also der Klassenbibliothek. Diese trägt notwendigerweise den Namen des 3DConnexion-Treibers (TDxInput), um binärkompatibel zu sein, da man Klassen üblicherweise über einen Namespace anspricht. Der korrekte Name der Device-Klasse ist also z.B. TDxInput.Device. TDxInput entspricht dabei auch dem Namen, unter dem die Bibliothek registriert ist. AerionInput besteht aus den Hauptklassen

- **Device**, welche das vollständige Gerät mit ihren dazugehörigen Methoden und Ereignissen und dem Bewegungssensor sowie der Tastatur abbildet;
- **Sensor**, die den Bewegungssensor kapselt und alle relevanten Methoden und Ereignisse des Sensors bereitstellt;
- **Keyboard**, die die Tasten des Gerätes abstrahiert und Tastenereignisse und den Status der Buttons enthält und
- **TDxInfo**, welche Informationen zum Treiber ausgibt.

Weiterhin existieren die Klassen

- **AngleAxis**, die Kenndaten einer Rotation in den drei Richtungen X, Y und Z sowie einem dazugehörigen (arbiträren) Winkel speichert und
- **Vector3D**, welche eine Translation in Form eines dreidimensionalen Vektors mit entsprechender euklidischer Vektornorm darstellt.

Für alle obigen Klassen existieren die entsprechenden Interfaces, welche die öffentlich sichtbaren Methoden zur Verfügung stellen. Weiterhin gibt es noch - entsprechend der Voraussetzungen für Event-Veröffentlichung mit COM - Interfaces für die Ereignisse, die die verschiedenen Klassen feuern können. Ebenfalls für die Ereignisbehandlung existieren Delegaten, die angeben, welche Signatur eine Ereignisbehandlungsroutine oder Event Handler hat.

Diese Klassen und Interfaces haben im 3DConnexion-Treiber ihre Entsprechung.

Die Klassen **Preferences**, **functions** und **func** sind Erweiterungen der ursprünglichen Klassen, um weitere Funktionalität zur Verfügung zu stellen. Diese Klassen sind jedoch nicht öffentlich und können daher von Anwendungen nicht instantiiert werden.

package Data[AerionInput]

Device

-c_AerionDeviceType : int = 63{readOnly}
-c_AerionProductID : int = 0x2013{readOnly}
-c_AerionVendorID : int = 0x3eb{readOnly}
-c_LogitechVendorID : int = 0x46D{readOnly}
-c_SpaceExplorerDeviceType : int = 4{readOnly}
-c_SpaceExplorerProductID : int = 0xC627{readOnly}
-c_SpaceNavigatorDeviceType : int = 6{readOnly}
-c_SpaceNavigatorProductID : int = 0xC626{readOnly}
-c_SpacePilotDeviceType : int = 29{readOnly}
-c_SpacePilotProductID : int = 0xC625{readOnly}
-c_SpaceTravelerDeviceType : int = 25{readOnly}
-c_SpaceTravelerProductID : int = 0xC623{readOnly}
-c_UnknownDeviceType : int = 0{readOnly}
-m_HidControl : HidControl
-m_IHidDevice : IHidDevice
-m_Type : int
+Keyboard : Keyboard
+m_Keyboard : Keyboard
+m_Sensor : Sensor
+Sensor : Sensor
+Type : int

-ConnectAerion() : bool
-ConnectSpaceExplorer() : bool
-ConnectSpaceNavigator() : bool
-ConnectSpacePilot() : bool
-ConnectSpaceTraveler() : bool
-HidControlDeviceChange(sender : object, e : EventArgs) : void
-IsAerion(p_IHidDeviceInfo : IHidDeviceInfo) : bool
-IsSpaceExplorer(p_IHidDeviceInfo : IHidDeviceInfo) : bool
-IsSpaceNavigator(p_IHidDeviceInfo : IHidDeviceInfo) : bool
-IsSpacePilot(p_IHidDeviceInfo : IHidDeviceInfo) : bool
-IsSpaceTraveler(p_IHidDeviceInfo : IHidDeviceInfo) : bool
+Connect() : void
+Device()
+DeviceChange()
+Disconnect() : void
+LoadPreferences(preferencesName : string) : void

<<C#Delegate>> _IKeyboardEvents_KeyDownEventHandler
<<C#Delegate>> _IKeyboardEvents_KeyUpEventHandler
<<C#Delegate>>_ISensorEvents_SensorInputEventHandler
<<C#Delegate>>_ISimpleDeviceEvents_DeviceChangeEventHandler

ISimpleDevice

+Connect() : HRESULT
+Disconnect() : HRESULT
+LoadPreferences(preferencesName : BSTR) : HRESULT
+get_Keyboard(*IKeyboard) : HRESULT
+get_Sensor(ISensor) : HRESULT
+get_Type(* type : long) : HRESULT

_ISimpleDeviceEvents

+DeviceChange() : void

Keyboard

-c_Button : int = 0x3{readOnly}
-m_IHidDevice : IHidDevice
-m_ProgrammableKeys : int
-m_KeyUpList : bool"[]" [0..*]
-m_KeyNameList : string"[]" [0..*]
-m_KeyLabelList : string"[]" [0..*]
+Keys : int
+ProgrammableKeys : int
+Device : Device

+KeyDown()
+KeyUp()
-HidDataReceived(sender : object, p_HidDataReceivedEventArgs : HidDataReceivedEventArgs) : void
+GetKeyName(key : int) : string
+GetKeyLabel(key : int) : string
+IsKeyUp(key : int) : bool
+IsKeyDown(key : int) : bool
+InitKeyboard(p_Device, p_IHidDevice : IHidDevice) : void
+Keyboard()
+DestroyKeyboard() : void

IKeyboard

+GetKeyName(key : long, *name : BSTR) : HRESULT
+GetKeyLabel(key : long, *label : BSTR) : HRESULT
+IsKeyUp(key : long, *retval : Boolean) : HRESULT
+IsKeyDown(key : long, *retval : Boolean) : HRESULT
+get_Keys(*keys : long) : HRESULT
+get_Device(*ISimpleDevice) : HRESULT
+get_ProgrammableKeys(*keys : long) : HRESULT

_IKeyboardEvents

+KeyDown(keycode : int) : void
+KeyUp(keycode : int) : void

+m_Device

Sensor

-c_MaxValueRotation : double = 128.0{readOnly}
-c_MaxValueTranslation : double = 128.0{readOnly}
-c_Rotation : int = 0x2{readOnly}
-c_Translation : int = 0x1{readOnly}
-m_IHidDevice : IHidDevice
-m_Period : double
-m_XRotScaleFactor : double
-m_XScaleFactor : double
-m_YRotScaleFactor : double
-m_YScaleFactor : double
-m_ZRotScaleFactor : double
-m_ZScaleFactor : double
+Device : Device
+Period : double
+Rotation : AngleAxis
+RotationFunction : functions
+Translation : Vector3D
+TranslationFunction : functions
+XRotScaleFactor : double
+XScaleFactor : double
+YRotScaleFactor : double
+YScaleFactor : double
+ZRotScaleFactor : double
+ZScaleFactor : double

-cube(p_x : double, p_max : double) : double
-HidDataReceived(sender : object, p_HidDataReceivedEventArgs : HidDataReceivedEventArgs) : void
-no_func(p_x : double, p_max : double) : double
-set_Function(p_function : functions) : func
-sin(p_x : double, p_max : double) : double
-square(p_x : double, p_max : double) : double
+DestroySensor() : void
+InitSensor(p_Device, p_IHidDevice : IHidDevice) : void
+Sensor()
+SensorInput()

ISensor

+get_Device(*ISimpleDevice) : HRESULT
+get_Rotation(*IAngleAxis) : HRESULT
+get_Translation(*IVector3D) : HRESULT
+get_Period(*period : double) : HRESULT

_ISensorEvents

+SensorInput() : void

+m_Translation

Vector3D

-m_X : double
-m_Y : double
-m_Z : double
-m_Length : double
-m_Sensor : ISensor
+Sensor : ISensor
+X : double
+Y : double
+Z : double
+Length : double

+Vector3D()

IVector3D

+get_X(*x : double) : HRESULT
+get_Y(*y : double) : HRESULT
+get_Z(*z : double) : HRESULT
+get_Length(*length : double) : HRESULT

+m_Rotation

AngleAxis

-m_X : double
-m_Y : double
-m_Z : double
-m_Angle : double
-m_Sensor : ISensor
+Sensor : ISensor
+X : double
+Y : double
+Z : double
+Angle : double

+AngleAxis()

IAngleAxis

+get_X(*x : double) : HRESULT
+get_Y(*y : double) : HRESULT
+get_Z(*z : double) : HRESULT
+get_Angle(*length : double) : HRESULT

-m_RotationFunction
-m_TranslationFunction

<<C#Delegate>>
func

-(p_x : double, p_max : double) : double

<<enumeration>>
functions

no_func
sin
square
cube

-m_Prefs

Preferences

-m_PreferencesName : string [0..*]
-m_RotationFunction : functions
-m_RotationThreshold : double [0..*]
-m_TranslationFunction : functions
-m_TranslationThreshold : double [0..*]
-m_XRotScaleFactor : double [0..*]
-m_XScaleFactor : double [0..*]
-m_YRotScaleFactor : double [0..*]
-m_YScaleFactor : double [0..*]
-m_ZRotScaleFactor : double [0..*]
-m_ZScaleFactor : double [0..*]
+PreferencesName : string
+RotationFunction : functions
+RotationThreshold : double
+TranslationFunction : functions
+TranslationThreshold : double
+XRotScaleFactor : double
+XScaleFactor : double
+YRotScaleFactor : double
+YScaleFactor : double
+ZRotScaleFactor : double
+ZScaleFactor : double

+loadConfig() : void
+saveConfig() : void

TDxInfo

-c_RevisionNumber : string = "1.0.0"{readOnly}
+RevisionNumber : string

+TDxInfo()

ITDxInfo

+RevisionNumber(*revisionNumer : BSTR) : HRESULT

Abb. 2.1: Klassendiagramm AerionInput

2.2 Event-System

Die Datenübergabe an die Anwendung erfolgt nach einem einfachen Prinzip: Die Daten werden dort abgefragt oder gepollt, wo sie produziert werden, also möglichst nah an der Hardware. Diese Aufgabe ist der größte Posten in bezug auf die Rechenzeit und wird von der HidLibrary.Net übernommen. Damit nicht noch mehr Rechenzeit durch Busy Waiting[1] bzw. Polling verschwendet wird, gibt es das so genannte Event-System. Dabei registriert sich ein Programm dafür, über ein bestimmtes Ereignis informiert zu werden, wenn es eintritt. Statt also in einem bestimmten Intervall eventuell unnötig abzufragen, ob schon Daten angekommen sind, informiert die HidLibrary.Net alle verbundenen Bibliotheken oder Anwendungen, dass Daten eingetroffen sind. Eine Anwendung, die über eingetroffene Daten informiert werden will, registriert sich für das Event „DataRecieved" und wird fortan von dieser Bibliothek mit Daten versorgt. Diese Strategie wird konsequent auch für AerionInput verwendet. Auf diese Weise bildet sich eine Event-Kette, in der jeweils ein Ereignis von der HID-Low-Level-Bibliothek bis herauf zur Anwendung „durchgereicht" wird. In eckigen Klammern steht jeweils die Anzahl der Aufrufe.

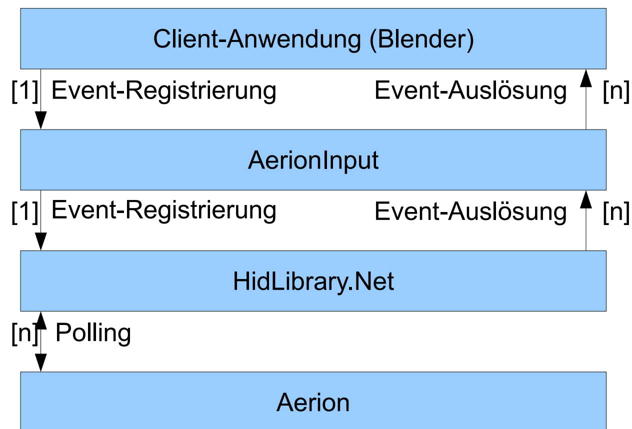

Abb. 2.2: Schichtenmodell von AerionInput

Eine Client-Applikation wie z.B. Blender kann sich bei AerionInput für folgende Ereignisse registrieren:

- **Device.DeviceChange,** um über das einstecken oder abziehen von Geräten informiert zu werden
- **Sensor.SensorInput** für Bewegungsdaten (Rotation und Translation) des Aerion
- **Keyboard.KeyDown,** um beim Niederdrücken einer Taste benachrichtigt zu werden
- **Keyboard.KeyUp,** um über das Loslassen einer Taste informiert zu werden

Den einzelnen Ereignissen müssen durch die Anwendung, die Daten von AerionInput erhalten will, mit eigenen Funktionen verknüpft werden. Soll also das Event `Device.DeviceChange`

[1] fortwährendes Abfragen der Daten in einem kurzen Zeitintervall

durch eine Client-Anwendung behandelt werden, muss diese eine spezielle Methode bereitstellen, die die Signatur des zugehörigen Event-Delegaten `_ISimpleDeviceEvents_DeviceChangeEventHandler` besitzt.

In diesem Fall wäre die Signatur der Methode `ClientApp.OnDeviceChange`:

```
public void OnDeviceChange (int reserved) {
//Behandle DeviceChange-Ereignis
[...]
}
```

Folgende Abbildung zeigt den Vorgang der Registrierung aller Ereignisse, die AerionInput anbietet, durch eine Client-Anwendung. Erwähnenswert ist dabei, dass das 3DConnexion-Blender-Plugin das Ereignis `Device.DeviceChange` nicht abfängt. Zudem ist erkennbar, dass sich AerionInput erst nach dem Aufruf der `Device.Connect()`-Methode mit `IHidDevice` aus der HidLibrary.Net verbindet. Damit wird verhindert, dass eine Anwendung sich für ein Ereignis registriert, welches gerade feuert und somit eventuell unvollständige Daten erhält.

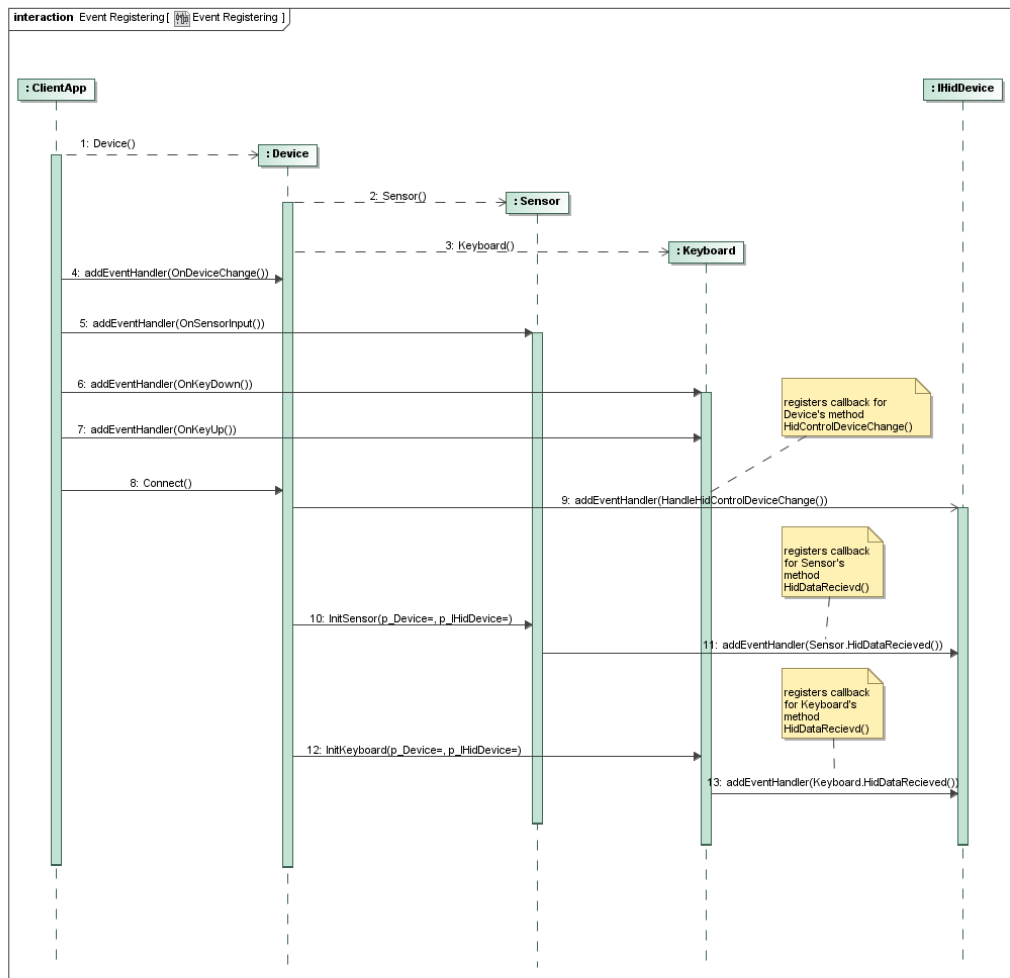

Abb. 2.3: Event Registrierung

Den umgekehrten Vorgang zeigt die nun folgende Abbildung. Die Client-Applikation ruft zuerst die `Device.Disconnect()`-Funktion auf. Diese Funktion löst dann intern schrittweise die Kette der Ereignisse zwischen AerionInput und der HidLibrary.Net wieder auf. Erst danach darf die Anwendung die Registrierung der Events aufheben.

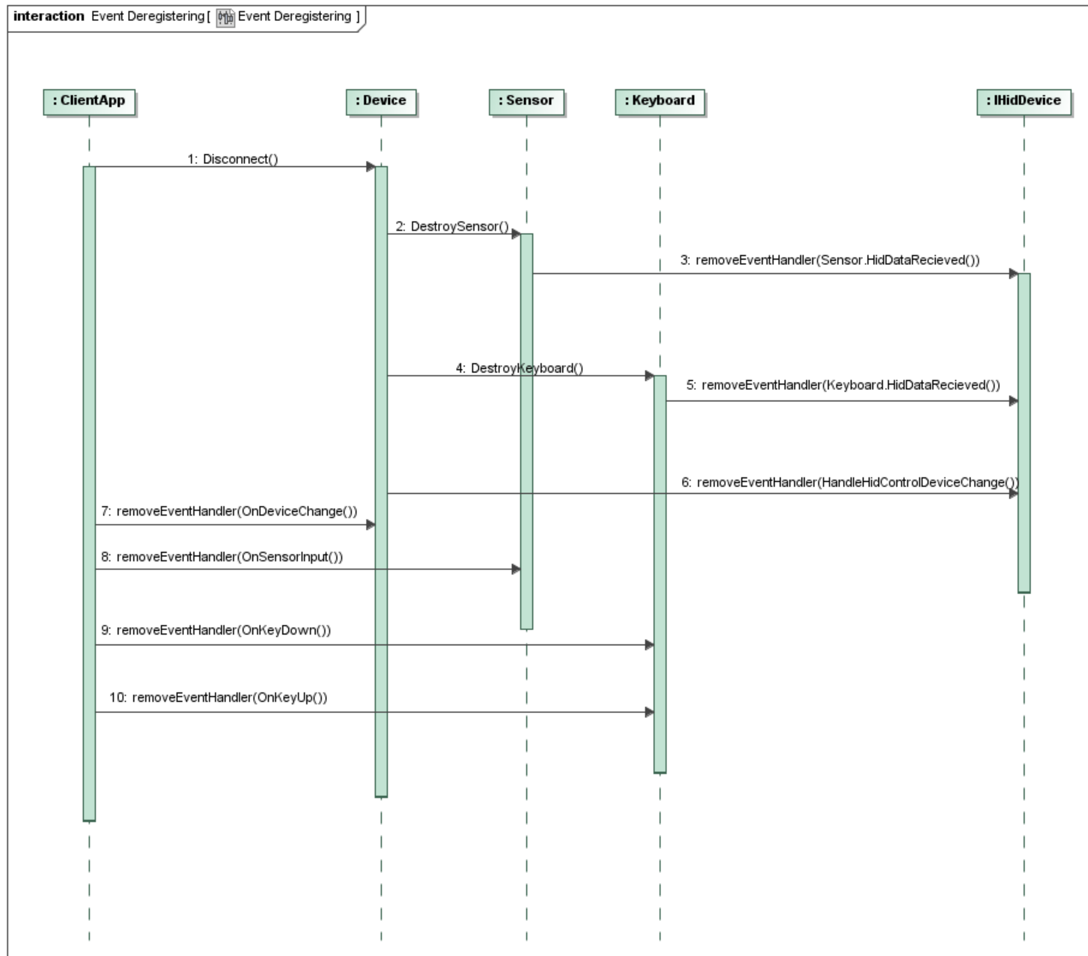

Abb. 2.4: `Event De-Registrierung`

2.3 Die Device-Klasse

Die Device-Klasse ist der Kern von AerionInput. Diese Klasse enthält ein Sensor- und ein Keyboard-Objekt, mit dem die beiden Hauptkomponenten des Geräts dargestellt werden. Device kümmert sich um die Verbindung der HidLibrary.Net mit dem Gerät über die `Device.Connect()`-Routine und ruft dazu für jedes unterstützte Gerät eine entsprechende Query-Routine (`HidControl.GetDeviceByPredicate()`) auf:

```
private bool ConnectAerion() {
  if ((m_IHidDevice = m_HidControl.GetDeviceByPredicate (
      new Predicate<IHidDeviceInfo>(IsAerion))) != null) {
    Console.Out.WriteLine("Aerion connected");
    m_IHidDevice.StartInPipeThread();
    m_Type = c_AerionDeviceType;
    return true;
  }
  else return false;
}
```

Jeder dieser Query-Funktionen wird ein Prädikat übergeben, z.B.

```
private const int c_AerionVendorID = 0x3eb;
private const int c_AerionProductID = 0x2013;

[...]

private bool IsAerion(IHidDeviceInfo p_IHidDeviceInfo) {
  return (p_IHidDeviceInfo.VendorID == c_AerionVendorID) &&
         (p_IHidDeviceInfo.ProductID == c_AerionProductID);
}
```

für den Aerion. Dieses Prädikat wird genau dann wahr, wenn die USB-VendorID und die ProductID eines angeschlossenen Geräts den Konstanten entsprechen. Auf diese Weise ist es sehr einfach, neue Geräte zu unterstützen, die ihre Daten dem HID-Protokoll für Multi-Axis-Controller entsprechend liefern. Da dies auf alle 3DConnexion-Eingabegeräte zutrifft, sprach nichts dagegen, diese Geräte ebenfalls zu unterstützen. Die entsprechenden Methoden der Device-Klasse tragen den Namen `Device.Connect_<GeräteName>()` und werden ebenfalls von obiger `Device.Connect()`-Methode aufgerufen. Soll eine Anwendung von AerionInput getrennt werden, so muss `Device.Disconnect()` aufgerufen werden. Diese Funktion trennt die Verbindung von AerionInput mit einem angeschlossenen Gerät (über HidLibrary.Net) und stoppt alle Data-Fetch-Aufgaben.

AerionInput unterstützt *genau ein* angeschlossenes Gerät. Sind mehrere Geräte angeschlossen, so wird immer das zuletzt durch `Connect()` gefundene Gerät eingebunden. Die Reihenfolge ist bis dato noch hard-coded, damit ist eine sinnvolle und vorhersehbare Nutzung des Treibers mit mehreren angeschlossenen Geräten nicht möglich! Geht man daher von der Nutzung genau eines Gerätes aus, so kann dieses entweder eingesteckt oder abgezogen sein.

Das Device-Objekt stellt das Ereignis `Device.DeviceChange` zur Verfügung. Wird ein Gerät während der Laufzeit von AerionInput eingesteckt oder ein eingestecktes Gerät abgezogen, wird dieses Ereignis ausgelöst. Ein Client, der sich für dieses Ereignis registriert hat, wird davon informiert und muss seine Objektzeiger auf `Device.Sensor` und `Device.Keyboard` aktualisieren und entsprechend die Ereignis-Behandlungsroutinen dieser Objekte entfernen und neu registrieren. Durch diese Maßnahme sind alle Zeiger des Clients stets aktuell, ungeachtet dessen, ob ein Gerät tatsächlich angeschlossen ist oder nicht. Ist keines angeschlossen, so wird auch von AerionInput kein `Device.Sensor.SensorInput` oder `Device.Keyboard.Key[Up/Down]`-Ereignis ausgelöst werden (können). Ergo werden

die Ereignisbehandlungsroutinen in der Client-Applikation nie aufgerufen. Die Anwendung muss sich also nur darum kümmern, seine Device-, Sensor- und Keyboard-Zeiger und deren Events aktuell zu halten, der Rest wird durch AerionInput erledigt. Leider ist dieser Part noch nicht ausreichend getestet, da das Blender-Plugin von Haus aus das `Device.DeviceChange`-Event ignoriert.

Beim Einstecken eines Gerät wird zuerst die Methode `Device.Disconnect()` aufgerufen, welche die Verbindung aller angeschlossenen Geräte trennt. Dann wird `Device.Connect()` aufgerufen und es wird nach sämtlichen unterstützten Geräten gesucht.

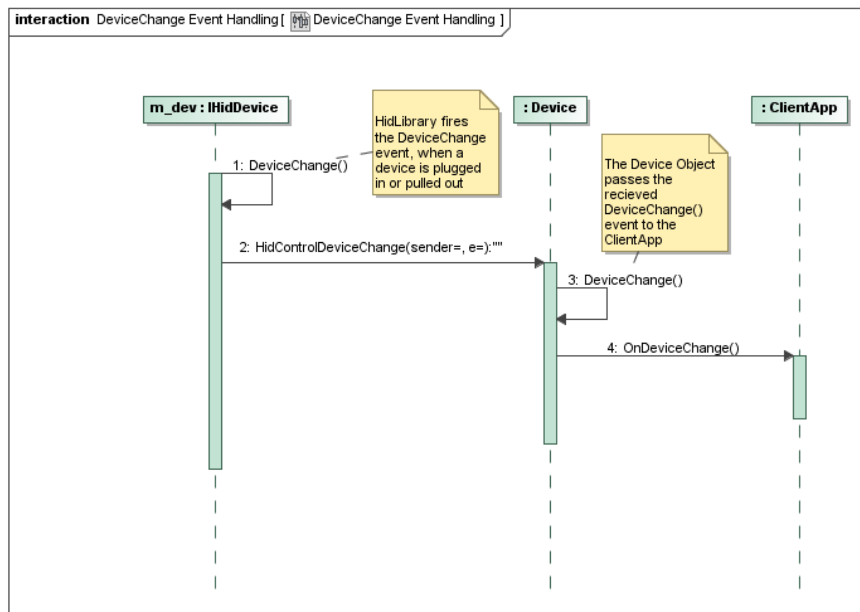

Abb. 2.5: Vorgänge beim Anstecken oder Entfernen eines unterstützen Geräts und Ablauf des `Device.DeviceChange`-Ereignisses

Die Device-Klasse enthält weiterhin ein `Preferences`-Objekt. Dieses Objekt verwaltet verschiedene Anwendungsprofile, die zur Laufzeit geladen werden können. Bei der Terminierung von AerionInput werden die Attribute des `Preferences`-Objekts in eine XML-Konfigurationsdatei gespeichert.

Die Getter-Funktionen `Device.get_Sensor()` und `Device.get_Keyboard()` liefern einen Objektzeiger zum Zugriff auf die entsprechenden Unterobjekte, ihre Methoden und Ereignisse.

2.4 Die Sensor-Klasse

Die Sensor-Klasse repräsentiert den Bewegungssensor des Aerion und kapselt Bewegungen und Drehungen der Kugel. Diese Daten werden in den untergeordneten Objekten `Sensor.Translation` und `Sensor.Rotation` als Vektoren abgelegt. Kommen von HidLibrary.Net Rotations- bzw. Translationsdaten an, werden diese vorverarbeitet. Dazu

werden die Daten als Eingabe für eine Filter-Funktion verwendet. Für Translation und Rotation kann jeweils eine eigene Funktion verwendet werden.

Mögliche Filterfunktionen sind im Einzelnen:

$$func\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \sin\begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{2.1}$$

$$func\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}^2 \tag{2.2}$$

$$func\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}^3 \tag{2.3}$$

Neue Filter können leicht hinzugefügt und implementiert werden, da diese als Funktionszeiger übergeben werden. Es muss lediglich die neue Filterfunktion definiert werden...

```
//Sensor.cs
//neue Funktion
private double newFunction(double p_x, double p_max) {
  return (newFunction((p_x / p_max), (p_x / p_max)) * p_max;
}
[...]
private func set_Function(functions p_function) {
    switch (p_function) {
      case (functions.sin):
        return sin;
      case (functions.square):
        return square;
      case (functions.cube):
        return cube;
      //neue Funktion
      case (functions.newFunction):
        return newFunction;
      default:
        return none;
    }
}
```

...und diese in die Enumeration der möglichen Filterfunktionen in der Datei `Preferences.cs` aufgenommen werden, um sie aus der Konfigurationsdatei laden zu können:

```
//Preferences.cs
public enum functions {
  none,
  sin ,
  square ,
  cube ,
  //neue Funktion
  newFunction
}
```

Erst die entsprechend vorverarbeiteten Daten werden in den Objekten `Sensor.Translation` und `Sensor.Rotation` abgelegt. Weiterhin lassen sich Skalierungsfaktoren für jede Komponente X, Y und Z von Rotation und Translation angeben. Diese werden mit der Ausgabe der jeweiligen Filter-Funktion multipliziert.

Schließlich wird überprüft, ob die erhaltenen Werte für Rotation und Translation größer als ein jeweiliger Schwellwert sind. Ist das der Fall, wird schließlich das Ereignis `Sensor.SensorInput` ausgelöst, um mögliche Clients über neu angekommene Daten zu informieren. Sowohl Schwellwert als auch Filterfunktionen lassen sich zur Laufzeit über das Konfigurationstool AerionInputConfig für jede Anwendung getrennt einstellen.

Das Event `Sensor.SensorInput` ist nur ein Benachrichtigungsereignis. Die Daten werden also nicht zusammen mit der Benachrichtigung an die Anwendung weitergegeben, sondern müssen von dieser selbst ausgelesen werden. Die Translationsdaten befinden sich dann im Sensor-Objekt gekapselt, in den Eigenschaften `Sensor.Translation` und `Sensor.Rotation`, auf welche über das Device-Objekt (`Device.Sensor`) zugegriffen werden kann. Die Translations- und Rotationsdaten sind auf ein bestimmtes Zeitfenster skaliert, bestehend aus der Differenz zwischen der Zeit, zu der das `Sensor.SensorInput`-Event letztmalig ausgelöst wurde und der aktuellen Zeit. Diese Zeit befindet sich in der Property `Sensor.Period`.

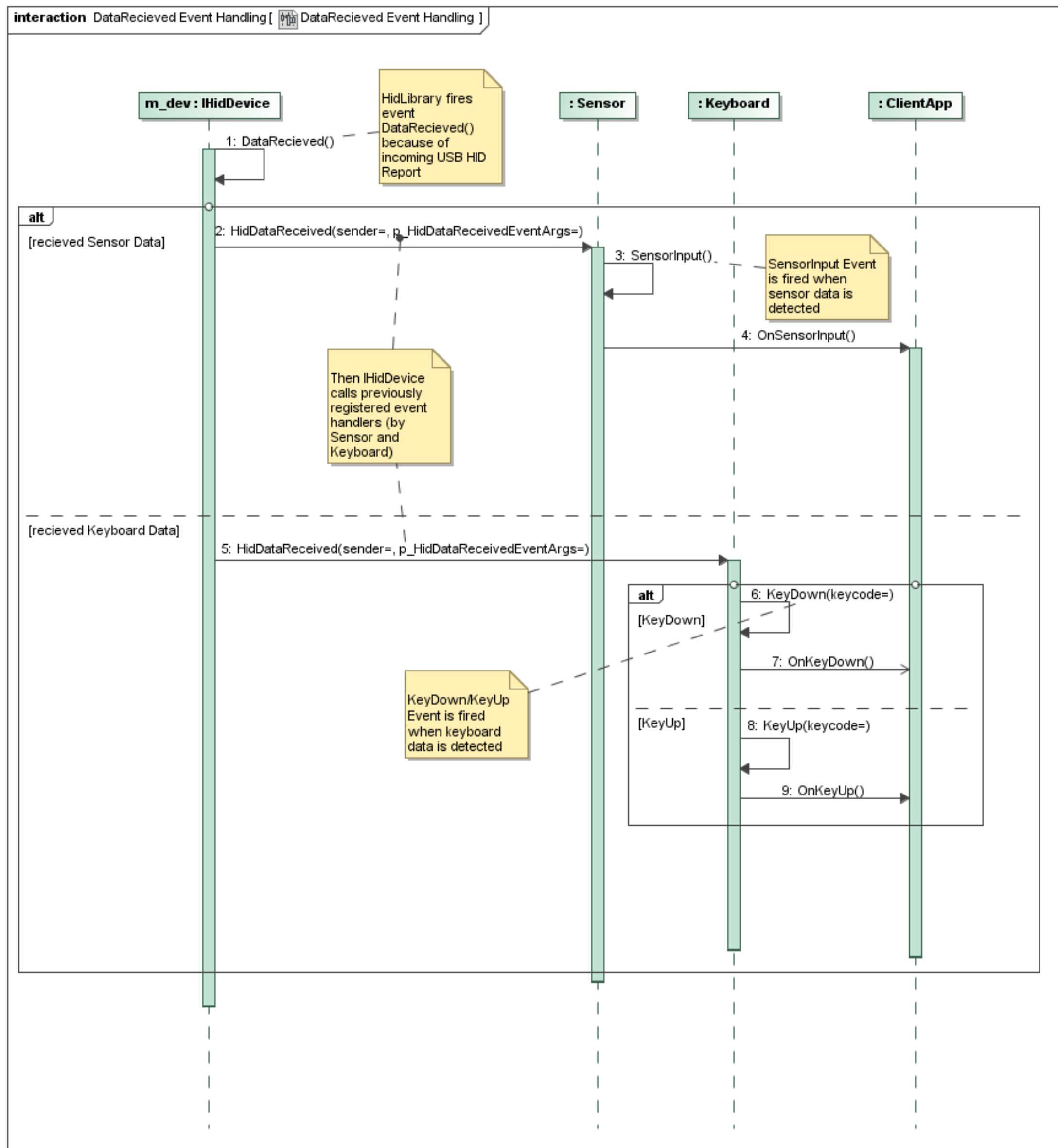

Abb. 2.6: Übertragung von Bewegungs- bzw. Tastendruck-Eingabedaten an die Anwendung

2.5 Die Keyboard-Klasse

Diese Klasse verwaltet den Status aller Buttons des Aerion. Sie hält eine Liste von Boolean-Werten für jeden Button (`m_KeyDownList`) bereit. Ist dieser Wert TRUE, so ist der entsprechende Button heruntergedrückt. Es gibt zwei weitere Listen, eine davon assoziiert Button-Nummern mit Namen, die andere mit Labels (wahrscheinlich frei wählbar). Die entsprechenden Methoden müssen für die Kompatibilität mit der 3DConnexion-API implementiert werden, jedoch gibt es über das COM-Interface keine Möglichkeit, die Labels zu verändern. Eine Anpassung muss daher über das Konfigurationstool des Treibers vorgenommen werden. Aufgrund geringer Priorität wurde diese Funktion bis dato nicht in AerionInputConfig aufgenommen.

Eine weitere Besonderheit ergibt sich für die Anzahl der Tasten an einem unterstützten Gerät. Diese ist im Moment noch fest verdrahtet auf zwei Tasten. Der Aerion hatte zur Entwicklungszeit dieses Projekts gar keine Tasten. Die Keyboard-Klasse wurde daher ausschließlich mit einem SpaceNavigator entwickelt und getestet, der aber nur über zwei Tasten verfügt. Die Realisierung von Buttons am Aerion sollte sich daher an die HID-Spezifikation halten. Die Usage Page für Buttons entspricht 0x3, das Datenbyte besteht aus acht Bits, die den Status von acht Buttons als Boolean nach dem Schema der `KeyDownList` (siehe oben) repräsentieren. AerionInput unterstützt in der Rohdatenverarbeitungsmethode `Keyboard.HidDataReceived()` im Moment maximal acht Buttons. Eine Erweiterung auf maximal 16, 32, 48... Buttons ist sehr einfach möglich, indem man den Puffer der Eingabedaten entsprechend vergrößert:

```
private void HidDataReceived
(
  object sender,
  HidDataReceivedEventArgs p_HidDataReceivedEventArgs
) {
  if (p_HidDataReceivedEventArgs[0] == c_Button) {
    try {
      byte[] buffer = {
        p_HidDataReceivedEventArgs[1],
        // Needed for more keys than 8
        // uncomment as much bytes as you have keys / 8
        /*
        p_HidDataReceivedEventArgs[2],
        p_HidDataReceivedEventArgs[3],
        p_HidDataReceivedEventArgs[4],
        p_HidDataReceivedEventArgs[5],
        p_HidDataReceivedEventArgs[6],
        */
      };
```

Bei der Verarbeitung der Tastenevents wird die Anzahl der Tasten in der Variable `m_Keys` respektiert. Eine Vergrößerung des obigen Puffers ist also gefahrlos möglich, da das BitArray nur bis zum Element `m_Keys` verarbeitet wird:

```
[...]
    BitArray bufferArray = new BitArray(buffer);
     for (int keycode = 0; keycode < m_Keys; keycode++) {
       // Fire event iff state of the button has changed
       if (bufferArray[keycode] ^ m_KeyDownList[keycode]) {
         // Fire KeyDown
         if (bufferArray[keycode]) {
           // Fire event thread save
           _IKeyboardEvents_KeyDownEventHandler
             KeyDownEventHandler = null;
           lock (m_lock) {
             KeyDownEventHandler = m_KeyDown;
             if (KeyDownEventHandler != null) {
               KeyDownEventHandler(keycode);
             }
           }
         }
       }
     }
```

Über einen Feature-Report sollte es möglich sein, die spezifische Anzahl der Buttons eines angeschlossenen Gerätes zu erfahren und in `m_Keys` abzulegen. Da jedoch noch keine Tasten am Aerion vorhanden sind, wurde hier auf eine Implementierung verzichtet, da der Nutzen für eine Demonstration der Fähigkeiten dieses Gerätes im Verhältnis zum Aufwand gering erschien. Eine prinzipielle Unterstützung von Geräten mit mehr als zwei Tasten, wie beispielsweise dem SpaceTraveller, ist sehr einfach möglich, indem `m_Keys` in diesem Fall auf 8 gesetzt wird. Da jedoch nicht vorhersehbar ist, welche Werte im BitArray an einer Stelle größer `m_Keys` stehen, wurde der sicherste Fall von zwei Tasten angenommen und hart-kodiert. Weiterhin wurde die Anzahl programmierbarer Tasten fest auf 0 gesetzt, da nicht klar ist, wie sich diese von normalen Tasten unterscheiden sollen.

Für die Keyboard-Events wird anders als für die Sensor-Events verfahren, dort wird dem Ereignisbehandler `OnKeyDown (int keycode)` gleich die gedrückte Taste als Parameter übergeben. Der Ablauf dieses Events wird in Abbildung 2.6 gezeigt.

2.6 Die AngleAxis-Klasse

Die Klasse AngleAxis repräsentiert eine Rotation im Raum als einen Vektor mit X-, Y- und Z-Achse. Dieser Vektor besitzt einen arbiträren Winkel, der seiner euklidischen Vektornorm entspricht. Setzt man den Winkel auf 0.0, so werden gleichzeitig auch die Komponenten X, Y, und Z auf 0.0 gesetzt. Ändert man eine der Komponenten, so wird gleichzeitig der Winkel neu berechnet.

2.7 Die Vector3D-Klasse

Die Klasse Vector3D repräsentiert eine Translation im Raum als einen Vektor mit X-, Y- und Z-Achse. Dieser Vektor besitzt einen Betrag, der seiner euklidischen Vektornorm entspricht. Setzt man den Betrag auf 0.0, so werden gleichzeitig auch die Komponenten X, Y, und Z auf 0.0 gesetzt. Ändert man eine der Komponenten, so wird gleichzeitig der Betrag neu berechnet. Setzt man den Betrag auf 1.0, dann werden die Komponenten so skaliert, dass der Vektor einem Einheitsvektor entspricht.

2.8 Die Preferences-Klasse

Die Klasse `Preferences` wird benötigt, um transparente Anwendungsprofile mit unterschiedlichen Filter-Funktionen, Schwellwerten und Skalierungsfaktoren für Translation und Rotation bereitzustellen. Zur Speicherung beliebig vieler unterschiedlicher Anwendungsprofile verwendet sie eine Liste, die dynamisch vergrößert werden kann. Preferences schlägt die Brücke zwischen den Anwendungsprofilen und ihrer Speicherung in einer XML-Konfigurationsdatei. Diese Klasse ist öffentlich, damit AerionInputConfig darauf zugreifen kann. Es ist jedoch nicht notwendig, diese Klasse von einer Drittapplikation aus zu benutzen.

Über die Funktion `Device.LoadProfile()` können anwendungsspezifische Profile für AerionInput geladen werden, um in jeder Anwendung mithilfe von AerionInputConfig fein-granulare Einstellungen am Bewegungsverhalten des Aerion vorzunehmen. Der obiger Funktion übergebene String kennzeichnet die Anwendung, mit der gearbeitet werden soll und deren Profil geladen wird. `Device.LoadProfile()` überprüft dazu, ob es ein Anwendungsprofil mit dem übergebenen Namen gibt und lädt in diesem Falle das Profil. Existiert kein Profil mit diesem Namen, wird eines angelegt. Hierdurch erhält eine bisher unbekannte Applikation transparent und automatisch ein eigenes Profil. In diesem Fall wird auch sofort das `Device.Preferences`-Objekt in eine XML-Konfigurationsdatei geschrieben. Diese Aufgabe sollte eigentlich beim Zerstören des Device-Objektes ausgeführt werden, was aber im Falle des Blender-Plugins als Test-Basis nicht zufriedenstellend funktionierte.

Zu beachten ist außerdem, dass Profile zur Laufzeit nicht verändert werden können. Weder bietet die COM-Schnittstelle etwaige Methoden an, noch wirken sich Änderungen durch AerionInputConfig direkt auf die laufende Instanz von AerionInput aus, sondern nur auf die XML-Konfigurationsdatei. Diese wird aber beim Aufruf der Methode `Device.Connect()`, also im Zweifelsfall nach einem Neustart der Client-Applikation neu geladen. Dann stehen in AerionInputConfig getätigte Veränderungen des Profils zur Verfügung.

Beim Aufruf der Methode `Device.Connect()` wird außerdem ein Default-Profil geladen, welches die Rohdaten eines angeschlossenen Gerätes ohne Skalierung oder Filterung an die Anwendung weitergibt.

Profilfunktion	Wertebereich
Schwellwert	Double
Filter	$f(x) = \sin(x), f(x) = x^2$
Skalierungsfaktor	-10 ... 10

2.9 Funktionsweise des Treibers am Beispiel des Blender-Plugins

Blender[2] ist eine Anwendung zur Erstellung von 3D-Animationen, die sich aus mehreren Gründen für eine Aerion-Unterstützung anbietet: Zum einen steht Blender unter der GPL und ist damit im Quelltext verfügbar. Zum anderen bietet 3DConnexion seit etwa einem Jahr ein Plugin für ihr Eingabegerät an. Der Quelltext für das Windows-Plugin steht uns zur freien Verfügung, da auch er unter der GPL steht. Im folgenden möchte ich beispielhaft dessen Quelltext erläutern, um zu klären, wie Blender die Daten von AerionInput erhält.

Das Blender-Plugin von 3DConnexion

Blender besitzt eine native Unterstützung von Eingabegeräten mit N-Freiheitsgraden (N-Degree of Freedom oder kurz NDOF) in seinem Eingabesystem „Ghost". 3DConnexion hat für die Anbindung ihrer Hardware ein Plugin erstellt, welches im Quelltext zur Verfügung steht.[3]

Das Plugin besteht aus der Klasse NDOFServer, welche die Vermittlung zwischen AerionInput und dem Blender-Eingabesystem Ghost übernimmt. Diese Klasse verfügt über Methoden, welche deren Initialisierung und Terminierung dienen sowie Sensor-Eingabedaten- und Button-Ereignisse verarbeiten. Aufgeteilt ist das Plugin in zwei Header-Files und das Hauptprogramm `3dcnxplug-win.cpp`. Die Header-Datei `3dcnxplug-stdafx.h` enthält folgende Anweisung:

```
#import "progid:TDxInput.Device" embedded_idl no_namespace
```

Diese Anweisung sorgt dafür, dass beim Laden des Plugins auch diejenige Bibliothek geladen wird, die die ProgID `TDxInput.Device` enthält. Ist AerionInput installiert, so wird AerionInput.dll geladen, indem in der Registrierung die ProgID `TDxInput.Device` nachgeschlagen und die dort vermerkte Bibliothek geladen wird. Ist statt dessen der originale Treiber installiert, wird folglich auch die Bibliothek von 3DConnexion geladen.

Über die beiden Event-Handler-Funktionen `OnSensorInput()` und `OnDevKeyDown(int keycode),` die in der Header-Datei `3dcnxplug-win.h` definiert sind, werden die beiden Methoden des Plugins für die Datenübergabe der Sensordaten und der Buttondaten exportiert:

[2] Blender-Homepage http://www.blender.org
[3] 3DConnexion Blender Plugin http://mirror.blenderbuilds.com/plugins/3DxBlender-Windows.zip

```
[...]
/** Callback invoked by the COM server
    when motion input is detected. */
HRESULT OnSensorInput();
/** Callback invoked by the COM server
    when device keys are pressed. */
  HRESULT OnDevKeyDown(int keycode);
[...]
  CComPtr<ISensor>      m_3DSensor;
  CComPtr<IKeyboard>    m_DevKeyb;
```

Hier werden ebenfalls die beiden Variablen `m_3DSensor` und `m_DevKeyb` deklariert, die Zeiger auf die bekannten Objekte Sensor und Keyboard des AerionInput für die Bewegungs- und Button-Daten bereithalten.

Im Hauptprogramm `3dcnxplug-win.cpp` wird beim Laden des Plugins die Funktion `NDOFServer::ndofInit()` aufgerufen. Diese initialisiert einen statischen `NDOFServer *sNdofServer`, welcher sich mit AerionInput verbindet und für den Datenaustausch sorgt. Dieser benutzt die wiederum statische Variable `GHOST_TEventNDOFData *sNdofStatus` als Puffer für Daten vom Eingabegerät. Dann ruft Blender die Funktion `NDOFServer::ndofOpen()` auf, die einen Pointer auf die gemeinsam von Plugin und Blender genutzte Variable `*platformData` als Parameter erwartet. Das Plugin wiederum schreibt immer dann, wenn es von AerionInput ein Ereignis erhält, die geänderten Daten für Rotation, Translation und Tastenevents in diese Variable.

Die Funktion `ndofShutdown(*device)` trennt eine etablierte Verbindung wieder. Über `NDOFServer::ndofOpen()` und die davon ausgeführte Methode `NDOFServer::initCOM()` wird das COM-Interface initialisiert und Blender bei AerionInput als Empfänger für Eingabe-Events registriert. Dazu wird zuerst die COM-Umgebung initialisiert:

```
ndofOpen() {
[...]
  initCOM() {
[...]
  hr = ::CoInitializeEx(0, COINIT_APARTMENTTHREADED |
                            COINIT_SPEED_OVER_MEMORY);
```

Dann wird als Funktionstest der Bibliothek eine nicht-initialisierte Instanz des `TDxInput.Device`-Objekts erzeugt, welches die Hauptklasse des AerionInput-Frameworks darstellt. Durch die Funktion `CoCreateInstance()` wird diese Instanz dann aber sofort wieder frei gegeben:

```
[...]
// create the COM device object
  hr = cnx_dev.CoCreateInstance(__uuidof(Device));
```

Im Erfolgsfall, also bei korrekter Registrierung der Bibliothek und aller zugehörigen Klassen, kann die Funktion `QueryInterface(&cnx_simpledev)` aufgerufen werden, die ein Objekt aus der Bibliothek zurückgibt, welches das übergebene Interface implementiert. In diesem Fall soll ein Objekt der Klasse `TDxInput.Device` instantiiert werden. Dieses Objekt implementiert das Interface `TDxInput.ISimpleDevice`, welches als Pointer übergeben wird:

```
[...]
  if (SUCCEEDED(hr)) {
    CComPtr<ISimpleDevice> cnx_simpledev;
    hr = cnx_dev.QueryInterface(&cnx_simpledev);
```

Ab hier steht eine Instanz der Device-Klasse zur Verfügung. Alle weiteren Funktionen können nun mithilfe der eigenen Methoden dieses Device-Objektes durchgeführt werden. Als erstes wird mithilfe der Getter-Funktion für `Device.Sensor` ein Pointer für das zugehörige Sensor-Objekt geholt:

```
[...]
    if (SUCCEEDED(hr)) {
      // get the actual interface to
      // the sensor and register for events
      hr = cnx_simpledev->get_Sensor(&m_3DSensor);
```

Die Funktion `__hook()` registriert die Plugin-Funktion `NdofServer::OnSensorInput()` für das Event `Device.SensorInput`. Dafür muss ein Pointer auf das entsprechende Event-Interface `_ISensorEvents` übergeben werden:

```
[...]
    if (SUCCEEDED(hr) && m_3DSensor.p)
      hr = __hook(&_ISensorEvents::SensorInput,
      m_3DSensor, &NDOFServer::OnSensorInput);
```

Um sich für Button-Events zu registrieren, ist folgendes Prozedere notwendig:

```
[...]
    // get the interface to the
    // device keys and register for events
    HRESULT hrk = cnx_simpledev->get_Keyboard(&m_DevKeyb);
    if (SUCCEEDED(hrk) && m_DevKeyb.p)
      hrk = __hook(&_IKeyboardEvents::KeyDown,
      m_DevKeyb, &NDOFServer::OnDevKeyDown);
```

Damit wird die Methode `NDOFServer::OnDevKeyDown` als Empfänger für Button-Events des AerionInput Keyboard-Objekts registriert.

Anschließend wird die Methode `Device.Connect` bzw. dessen COM-Inkarnation `cnx_simpledev->Connect()` aufgerufen, um AerionInput anzuweisen, nach angeschlossenen Geräten zu suchen und die Event-Quellen „scharf" zu schalten:

```
[...]
        if (SUCCEEDED(hr) && SUCCEEDED(hrk)
          && cnx_simpledev->Connect() == S_OK)
          err = 0;
```

Wird das Ereignis Device.SensorInput ausgelöst, so erfolgt die Übertragung der Bewegungsdaten von AerionInput an Blender in der Funktion `NDOFServer::OnSensorInput()`:

```
HRESULT NDOFServer::OnSensorInput() {
  static double ndofData[7];
[...]
  CComPtr<IAngleAxis> pRot;
  CComPtr<IVector3D> pTra;
  double len, ang;

  m_3DSensor->get_Translation(&pTra);
  m_3DSensor->get_Rotation(&pRot);
  pTra->get_Length(&len);
  pRot->get_Angle(&ang);
```

Die Variable `m_3DSensor` ist der Zeiger auf das aktuelle Sensor-Objekt, das den Zugriff auf die Bewegungsdaten für Translation und Rotation kapselt. Über `m_3DSensor->get_Translation(&pTra)` und `m_3DSensor->get_Rotation(&pRot)` werden jeweils Translation und Rotation ausgelesen. Dabei wird die Rotation in einen COM-Zeiger auf ein AngleAxis-Objekt übertragen und die Translation in einen solchen Pointer auf ein Vector3D-Objekt.

Vor der Übertragung an Blenders Ghost-Interface und damit der Ausführung der Bewegung wird überprüft, ob sich der Betrag der Translations- oder Rotationskomponenten überhaupt verändert hat. Dies geschieht über die Methoden `pTra->get_Length(&len)` bzw. `pRot->get_Angle(&ang)`, die jeweils die Vektornorm der entsprechenden Vektoren ausgeben.

Das Double-Array `ndofData[7]` enthält die X-, Y- und Z-Komponenten der sechs Freiheitsgrade und im siebten Feld die Zeit, auf die Translations- und Rotationsdaten skaliert sind. Die Funktion `m_3DSensor->get_Period(&ndofData[6])` holt diese Daten von AerionInput ab:

```
[...]
  m_3DSensor->get_Period(&ndofData[6]);
  sNdofStatus->changed = 1;
  sNdofStatus->delta = ndofData[6] - sNdofStatus->time;
  sNdofStatus->time = ndofData[6];
```

Die X-, Y- und Z-Komponenten von Translation und Rotation werden aus dem Array `ndofData[]` gelesen und an die Struktur `GHOST_TEventNDOFData sNdofStatus` übergeben:

```
[...]
  // Check if the cap is still displaced
  if (len > 0.0 || ang > 0.0) {
    pTra->get_X(&ndofData[0]);
    pTra->get_Y(&ndofData[1]);
    pTra->get_Z(&ndofData[2]);
    pRot->get_X(&ndofData[3]);
    pRot->get_Y(&ndofData[4]);
    pRot->get_Z(&ndofData[5]);
```

```
sNdofStatus->tx = ndofData[0];
sNdofStatus->ty = ndofData[1];
sNdofStatus->tz = -ndofData[2];
```

Bei der Translation wird der von AerionInput erhaltene Vektor direkt an Ghost übertragen. Für die Rotation wird anders verfahren: Die neuen Winkelwerte werden mit dem vorher geholten Winkelmaß `ang` multipliziert:

```
[...]
    sNdofStatus->rx = (GHOST_TInt16)(ndofData[3] * ang);
    sNdofStatus->ry = (GHOST_TInt16)(ndofData[4] * ang);
    sNdofStatus->rz = (GHOST_TInt16)(ndofData[5] * ang);
```

Wenn sich die Cursor-Position nicht verändert haben sollte, gelangen wir in den else-Zweig obiger Bedingung:

```
[...]
  else {
  // zero out axisData so that 3d motion stops.
    memset(&sNdofStatus->tx, 0, sizeof(GHOST_TInt16) * 6);
  }
```

Hier nun werden einfach alle Achsen-Felder der Struktur `sNdofStatus` auf „0” gesetzt, damit stoppt im Blender-Fenster die Bewegung.

Schließlich wird dem gerade aktiven Fenster folgende Nachricht geschickt, wobei diese nur in die Nachrichtenwarteschleife eingereiht wird und die Plugin-Ausführung nicht unterbrochen wird. Wenn der Blender-Hauptprozess rescheduled[4] wird, werden alle anstehenden Fenster-Nachrichten verarbeitet und die Bewegung im Blender-Fenster sichtbar.

```
[...]
  PostMessage(GetActiveWindow(), WM_BLND_NDOF_AXIS, NULL, NULL);
```

Ist das Blender-Fenster nicht aktiv, so sollen dort auch keine Daten verarbeitet werden, da es ja gar nicht den Fokus hat. Die Konstante `WM_BLND_NDOF_AXIS` zeigt Blender, dass es sich um ein Achsen-Bewegungsereignis handelt. Diese Window-Nachricht ist eine frei definierte User-Nachricht, wie in der Datei `GHOST_Types.h` nachzulesen ist:

```
[...]
#define WM_BLND_NDOF_AXIS WM_USER + 1
```

Entsprechendes gilt auch für das Button-Ereignis:

```
[...]
#define WM_BLND_NDOF_BTN WM_USER + 2
```

Um eben dieses Button-Klick-Ereignis abzufangen, geht das Plugin folgendermaßen vor:

[4]to schedule: Sinngemäß: Bekommt die Ressource Prozessor zum Rechnen wieder zugewiesen

```
HRESULT NDOFServer::OnDevKeyDown(int keycode) {
[...]
  sNdofStatus->changed = 2;
  sNdofStatus->buttons = keycode;
  PostMessage(GetActiveWindow(), WM_BLND_NDOF_BTN, NULL, NULL);
```

Hier wird der Status `sNdofStatus->changed` auf 2 gesetzt, um anzuzeigen, dass es sich um ein Button-Ereignis handelt. Der Keycode des Buttons[5] wird übergeben und wieder mit `PostMessage()` eine Nachricht an das aktive Fenster geschickt, welches eine Änderung der Buttons anzeigt.

Will man den Aerion in einer Applikation unterstützen, so kann man nach diesem Muster vorgehen, um die grundlegenden Funktionen des Aerion benutzen zu können. Im Blender-Plugin fehlt eine Unterstützung des `Device.DeviceChange`-Events. Da diese jedoch nicht essentiell[6] ist, um den Aerion zu benutzen, wurde auf eine Erweiterung des Plugins im Rahmen dieser Arbeit verzichtet. Eine beispielhafte Implementierung kann jedoch exakt so aussehen wie die oben beschriebenen Funktionen für die Unterstützung des SensorInput und KeyDown-Events.

2.10 AerionInputConfig

Dieses Zusatzprogramm existiert, um Anwendungsprofile mit jeweils verschiedenen Schwellwerten, Filterfunktionen und Skalierungsfaktoren für Rotation und Translation verwalten zu können. Beim erstmaligen Aufruf der Funktion Device.LoadProfile() wird durch AerionInput pro Anwendung ein Profil angelegt, welches sofort in einer XML-Konfigurationsdatei gespeichert wird. AerionInputConfig ist ein Editor für diese Konfigurationsdatei, dessen Änderungen jedoch nicht zur Laufzeit übernommen werden können. Dafür muss die betreffende Anwendung geschlossen und neu gestartet werden.

Der Ausbau von AerionInputConfig zu einer "Push-Komponente", welche aktiv Daten an eine Anwendung übergibt, ist relativ unkompliziert möglich. Die Klassen von AerionInput können nativ instantiiert werden, es muss kein Umweg über COM genommen werden. Jedoch müssen dann für jede Anwendung, die aktiv von AerionInputConfig ferngesteuert werden soll, eigene Funktionen geschrieben werden. Jede Anwendung muss für diese "Push-Komponente" explizit unterstützt werden. Da die API für die Anwendungsfernsteuerung bestimmter Anwendungen aber mitunter gar nicht öffentlich verfügbar ist, stellt diese Komponente gerade für das bei dieser Arbeit betriebene Rapid Prototyping (schnelle Prototypenentwicklung, um frühzeitig Machbarkeit zu verifizieren) keinen guten Einstiegspunkt dar. Daher sollte eine entsprechende "Push-Komponente" erst jetzt entwickelt werden, da mit der "Pull-Komponente" bereits eine funktionsfähige Infrastruktur vorhanden ist. Ein guter Anknüpfungspunkt dazu wäre die Anwendung GoogleEarth, deren API offenliegt. Sie ist zumindest grundlegend dokumentiert und erscheint für eine Aerion-Unterstützung geeignet.

[5] Nummerierung der am Gerät vorhandenden Buttons, entspricht Button-Nummer

[6] Ein Abziehen des Aerion zur Laufzeit von Blender führt lediglich dazu, dass man den Aerion vor einem Blender-Neustart nicht mehr benutzen kann. Ein Absturz ließ sich nicht provozieren.

3 Ausblick

Mit dieser Arbeit wurde dem Aerion eine Anbindung an Standard-Applikationen ermöglicht. Durch die hohe Kompatibilität mit dem Treiber von 3DConnexion ist eine Einbindung in weitere Anwendungen zusätzlich zu Blender realisierbar. Dies wurde bereits durch eine modifizierte[1] Version von GPUCast[2] von Andreas Schollmeyer demonstriert. Ebendiese Kompatibilität machte die Nutzung von AerionInput mit 3DConnexion-Hardware als freien Ersatz für den proprietären Treiber möglich. Da zum Testen entsprechende Hardware zur Verfügung stand, wurde die Unterstützung dafür kurzerhand ebenfalls implementiert, jedoch aufgrund des Fokus auf den Aerion nicht en detail getestet. Der Autor vermutet Schwierigkeiten mit der so genannten „Tilt"-Bewegung bei der Verwendung von 3DConnexion-Hardware, da sich diese Bewegung schwer bis gar nicht vollziehen ließ.

Die vorliegende Arbeit wurde nach Fertigstellung veröffentlicht und in das „Free Spacenav"-Projekt[3] integriert. Der jeweils aktuelle Quelltext steht ab sofort im dortigen SVN zur Verfügung. Weiterhin findet sich dort auch ein Installer für die End-User-Runtime-Bibliothek inklusive Blender-Plugin (ohne Quelltexte). Damit ist eine Nutzung von Aerion in Blender auch für Nicht-Entwickler möglich.

Todo

- Das Event Device.DeviceChange benötigt noch zusätzliches Testing, eine Quick'n'Dirty-Implementierung hat leider nicht auf Anhieb funktioniert.
- Nicht nur für unterschiedliche Anwendungen, auch für unterschiedliche Hardware sollten distinkte Profile unterstützt werden.
- LED an 3DConnexion-Hardware bei `Device.Connect()`-Funktion einschalten.
- Feature Report implementieren, um Anzahl der Buttons zur Laufzeit zu bestimmen.
- Implementierung einer „Push-Komponente" ähnlich wie 3dxserv.exe
- Als letzten Schritt das Projekt vollständig in C++ neuimplementieren, um von .NET unabhängig zu sein und das Projekt mit quelloffenen Compilern erstellen zu können.

Diese Aufgaben werden nicht mehr Teil dieser Arbeit, sondern Gegenstand von Hiwi-Tätigkeiten für das GlobeFish/Aerion-Projekt oder freiwilliger Arbeit sein.

[1] Nurbs Casting https://trac.medien.uni-weimar.de/trac/gpucast
[2] GPUCast www.gpucast.org
[3] Free Spacenav Projekt http://spacenav.sourceforge.net/

A Anhang

A.1 Klassen- und Methodenreferenz AerionInput

A.1.1 ISimpleDevice

COM interface exposing the methods of Device. Forced to use the same GUID as the original driver interface does. Provides both interface implementations (Dual), IDispatch and IUnknown for early and late binding.

A.1.1.1 Properties

Sensor Returns an interface pointer to the associated Sensor object. For COM interop this must be marshaled explicitly to Interface. The DispId attribute states the vtable entry of the method.

Keyboard Returns an interface pointer to the associated Keyboard object. For COM interop this must be marshaled explicitly to Interface. The DispId attribute states the vtable entry of the method.

Type Returns type of the connected device. The DispId attribute states the vtable entry of the method.

IsConnected Returns true if a device is connected with the driver. The DispId attribute states the vtable entry of the method.

A.1.1.2 Methods

Connect Tries to find connected devices. For proper operation this method must be called before getting data from the device. The DispId attribute states the vtable entry of the method.

Disconnect Disconnects already connected devices. Before the client application shuts down, this method should be called in order to disconnect event handlers and save application profiles. The DispId attribute states the vtable entry of the method.

LoadPreferences(System.String) Loads the application profile in argument preferencesName. As usual in a COM environment the stings are of type BSTR, so marshal strings that way. The DispId attribute states the vtable entry of the method.

A.1.2 IKeyboard

COM interface exposing the methods of Keyboard. Forced to use the same GUID as the original driver interface does. Provides both interface implementations (Dual), IDispatch and IUnknown for early and late binding.

A.1.2.1 Properties

Keys Returns number of keys the device offers. The DispId attribute states the vtable entry of the method.

ProgrammableKeys Returns number of programmable keys the device offers. The DispId attribute states the vtable entry of the method.

Device Returns an IDispatch pointer to the parent Device object associated with the Sensor. For COM interop compatibility, the object must be marshaled to IDispatch. (See Blender Plugin) The DispId attribute states the vtable entry of the method.

A.1.2.2 Methods

GetKeyLabel(System.Int32) Returns the Label of the supplied key, if any. As usual in a COM environment the stings are of type BSTR, so marshal strings that way. "In" states that the argument will be passed to the structure. The DispId attribute states the vtable entry of the method.

GetKeyName(System.Int32) Returns the Name of the supplied key, if any. As usual in a COM environment the stings are of type BSTR, so marshal strings that way. "In" states that the argument will be passed to the structure. The DispId attribute states the vtable entry of the method.

IsKeyDown(System.Int32) Returns true if the supplied key is pressed down at the time. The DispId attribute states the vtable entry of the method.

IsKeyUp(System.Int32) Returns true if the supplied key is up at the time. The DispId attribute states the vtable entry of the method.

A.1.3 Vector3D

Provides translational data in three dimensions and a length (euclidean norm). Implements the IVector3D interface which exposes its methods to COM. Because of that the ClassInterfaceType must be "None", otherwise the CLR provides another class interface (and does not use IVector3D). The class is forced to use the same GUID as the original CoClass does in order to be accessible by binary-only clients.

A.1.3.1 Properties

X Represents the X component of the translation.

Y Represents the Y component of the translation.

Z Represents the Z component of the translation.

Length Represents the length of the translation (euclidean length of the three components). If set to 0.0 all three components are set to 0.0, if set to 1.0 the components are scaled to be a unit vector.

A.1.3.2 Methods

Constructor: Vector3D Vector3D constructor

A.1.4 IVector3D

COM interface exposing the methods of Vector3D. Forced to use the same GUID as the original driver interface does. Provides both interface implementations (Dual), IDispatch and IUnknown for early and late binding.

A.1.4.1 Properties

X Gets or sets the x component of the vector. The DispId attribute states the vtable entry of the method. param: In states that the argument will be passed to the structure.

Y Gets or sets the y component of the vector. The DispId attribute states the vtable entry of the method. param: In states that the argument will be passed to the structure.

Z Gets or sets the z component of the vector. The DispId attribute states the vtable entry of the method. param: In states that the argument will be passed to the structure.

Length Gets or sets the euclidean length of the vector. The DispId attribute states the vtable entry of the method. param: In states that the argument will be passed to the structure.

A.1.5 functions

Enumeration of possible functions in order to save numeric values to the config file (and not delegates..)

A.1.6 Preferences

Represents a configuration profile on a per-application basis. In order to load another profile just use the Setter of PreferencesName. If a profile of this name exists it will be loaded, if not a new default profile of this name will be created. This class is an extension of the orginal API, there is no such class exposed to COM. Thatswhy the Attribute ComVisible(false). So this is our implementation of the LoadPreferencesName() call of Device. The documentation says nothing about that so we can only guess the real functionality. This is our "educated" guess so ;)

A.1.6.1 Fields

path Contains the exact path (with file name) to the XML config file.

A.1.6.2 Properties

RotationFunction Gets or Sets the filter function which shall be applied to the rotational input data.

index Gets or Sets the list index of the current profile.

TranslationFunction Gets or Sets the Translation Filter Function of the current profile. Possible values: sin, cos, square, cube, none.

PreferencesName Gets or Sets the Name of the current profile. When a name is set which is not in the list holding all known profiles, it will be created.

Count Gets the count of stored profiles.

XScaleFactor Gets or Sets the scaling factor the Translation X value will be multiplied with.

YScaleFactor Gets or Sets the scaling factor the Translation Y value will be multiplied with.

ZScaleFactor Gets or Sets the scaling factor the Translation Z value will be multiplied with.

XRotScaleFactor Gets or Sets the scaling factor the Rotation X value will be multiplied with.

YRotScaleFactor Gets or Sets the scaling factor the Rotation Y value will be multiplied with.

ZRotScaleFactor Gets or Sets the scaling factor the Rotation Z value will be multiplied with.

RotationThreshold Gets or Sets the Rotation threshold value.

TranslationThreshold Gets or Sets the Translation threshold value.

A.1.6.3 Methods

ConstructPath Creates the path variable to the XML config file. Usually this is "C:\Documents and Settings\%username%\Application Data\AerionInput\config.xml". If directories do not exist, they will be created.

loadConfig Loads the XML config file stored in "path" and parses the config data to its local variables. If anything goes wrong, all fields will be initialized with defaults. So if file is malformed, a new valid config file will be created.

saveConfig Writes all preferences into XML config file stored in "path". Must be called explicitly in order to save values for the next session. Overwrites any existing config file with the values in this object.

Constructor: Preferences Calls ConstructPath() to have a valid path string

A.1.7 _ISimpleDeviceEvents_DeviceChangeEventHandler

Delegate declaration for DeviceChange event handler. Only states the return type of the event which is void for COM events and the signature of the event. This is implicitly COM visible and needed for COM to determine the signature of the event.

reserved: Not yet defined functionality.

A.1.8 _ISimpleDeviceEvents

COM Events publisher interface forced to use the same GUID as the original driver interface does. Must be IDispatch Interface for late binding, not standard dual.

A.1.8.1 Methods

DeviceChange(System.Int32) The DeviceChange event is fired whenever a device is plugged in or pulled off. The reserved argument has no use for now.

Parameters **reserved:**

A.1.9 ITDxInfo

COM interface exposing the methods of TDxInfo. Forced to use the same GUID as the original driver interface does. Provides both interface implementations (Dual), IDispatch and IUnknown for early and late binding.

A.1.9.1 Methods

RevisionNumber Returns the revision number of the driver. As usual in a COM environment the stings are of type BSTR, so marshal strings that way. The DispId attribute states the vtable entry of the method.

A.1.10 IAngleAxis

COM interface exposing the methods of AngleAxis. Forced to use the same GUID as the original driver interface does. Provides both interface implementations (Dual), IDispatch and IUnknown for early and late binding.

A.1.10.1 Properties

X Gets or sets the x component of the vector. The DispId attribute states the vtable entry of the method. param: In states that the argument will be passed to the structure.

Y Gets or sets the y component of the vector. The DispId attribute states the vtable entry of the method. param: In states that the argument will be passed to the structure.

Z Gets or sets the z component of the vector. The DispId attribute states the vtable entry of the method. param: In states that the argument will be passed to the structure.

Angle Gets or sets the angle of the vector (arbitrary units but euclidean norm). The DispId attribute states the vtable entry of the method. param: In states that the argument will be passed to the structure.

A.1.11 func

Filter function delegate. If you want to define a new filter function, use this signature for it (input value x and maximum value max). Then process the input value through your function (e.g. f(x)=x^4) and return this value. Possible extensions can be an accumulation of function values and returning a median of the last 5 values for example.

p_x:

p_max:

Return Value:

A.1.12 Sensor

Represents the motion sensor of the device. Encapsulates movements and twists of the cap or ball of the device. Holds data in the AngleAxis (for Rotation) and Vector3D (for Translation) members which are accessible trough COM. Processes (accelerate and multiply with constant) raw Input data. Fires SensorInput event whenever new Input is detected. Implements the ISensor interface which exposes its methods to COM. Because of that the ClassInterfaceType must be "None", otherwise the CLR provides another class interface (and does not use IVector3D). The class is forced to use the same GUID as the original CoClass does in order to be accessible by binary-only clients.

A.1.12.1 Fields

c_MaxValueTranslation Maximum value for translation data used for scaling with filter function. THIS MUST BE ADJUSTED IF ANY DEVICE SENDS HIGHER VALUES!! At the time this value is far from optimal for 3DConnexion Devices but seems to work properly. These devices offer 11 bit precison, however setting MaxValue to 2048 causes the sensor not to move. Will digg in that later.

c_MaxValueRotation Maximum value for translation data used for scaling with filter function. THIS MUST BE ADJUSTED IF ANY DEVICE SENDS HIGHER VALUES!! At the time this value is far from optimal for 3DConnexion Devices but seems to work properly. These devices offer 11 bit precison, however setting MaxValue to 2048 causes the sensor not to move. Will digg in that later.

c_Translation Usage page for Translation Reports

c_Rotation Usage page for Rotation Reports

m_IHidDevice IHidDevice object of HidLibrary which raises events when Hid Reports are recieved.

IsDisposed IDisposeable implementation help variable.

m_HidDeviceEvents_HidDataReceivedEventHandler HidDataReceived event handler member variable storing the function pointer (delegate) to the method m_HidDevice will call back if a HidDataReceived is detected.

m_lock Binary semaphore for locking event access.

A.1.12.2 Properties

XScaleFactor Gets or Sets the X translation scaling factor which is mulitiplied with the raw input data from the Sensor before passed to the client. This property is an extension of the orginal API, which is not exposed to COM. Thatswhy the Attribute ComVisible(false).

YScaleFactor Gets or Sets the Y translation scaling factor which is mulitiplied with the raw input data from the Sensor before passed to the client. This property is an extension of the orginal API, which is not exposed to COM. Thatswhy the Attribute ComVisible(false).

ZScaleFactor Gets or Sets the Z translation scaling factor which is mulitiplied with the raw input data from the Sensor before passed to the client. This property is an extension of the orginal API, which is not exposed to COM. Thatswhy the Attribute ComVisible(false).

XRotScaleFactor Gets or Sets the X rotation scaling factor which is mulitiplied with the raw input data from the Sensor before passed to the client. This property is an extension of the orginal API, which is not exposed to COM. Thatswhy the Attribute ComVisible(false).

YRotScaleFactor Gets or Sets the Y rotation scaling factor which is mulitiplied with the raw input data from the Sensor before passed to the client. This property is an extension of the orginal API, which is not exposed to COM. Thatswhy the Attribute ComVisible(false).

ZRotScaleFactor Gets or Sets the Z rotation scaling factor which is mulitiplied with the raw input data from the Sensor before passed to the client. This property is an extension of the orginal API, which is not exposed to COM. Thatswhy the Attribute ComVisible(false).

TranslationFunction Sets the Translation filter function which takes the raw input data from the Sensor and passes it to this function as argument. The result of this will be passed to the client. This property is an extension of the orginal API, which is not exposed to COM. Thatswhy the Attribute ComVisible(false).

RotationFunction Sets the Rotation filter function which takes the raw input data from the Sensor and passes it to this function as argument. The result of this will be passed to the client. This property is an extension of the orginal API, which is not exposed to COM. Thatswhy the Attribute ComVisible(false).

TranslationThreshold Gets or Sets the Translation Threshold. The Vector3D.Length attribute must be > TranslationThreshold to be passed to the client. This property is an extension of the orginal API, which is not exposed to COM. Thatswhy the Attribute ComVisible(false).

RotationThreshold Gets or Sets the Rotation Threshold. The AngleAxis.Angle attribute must be > RotationThreshold to be passed to the client. This property is an extension of the orginal API, which is not exposed to COM. Thatswhy the Attribute ComVisible(false).

Period Gets the time frame the values of Translation and Rotation are scaled to.

Translation Gets lengths of movement for all 3 axes.

Rotation Gets angles for all 3 axes.

Device Gets the parent Device object associated with the Sensor.

A.1.12.3 Methods

set_Function(TDxInput.functions) Converts numeric function types to function pointers for filter processing. Used to convert the enum type coming from the XML config to delegate.

HidDataReceived(System.Object,HidLibrary.HidDataReceivedEventArgs) Handles HidDataReceived events and processes incoming raw data from the Device. Fills m_Translation and m_Rotation objects with the filtered and scaled raw data. At the end fire SensorInput to notify clients.

Parameters **sender:**

p_HidDataReceivedEventArgs:

none(System.Double,System.Double) Null filter. Output = Input.

Parameters **p_x:** input value

p_max: maximum value the function is scaled to

Return Value **Return Value:**

sin(System.Double,System.Double) Sinus filter. Output = sin(Input) [Shifted to origin]

Parameters **p__x:** input value

p__max: maximum value the function is scaled to

Return Value **Return Value:**

cube(System.Double,System.Double) Cube filter. Output = Input*Input*Input [Shifted to origin]

Parameters **p__x:** input value

p__max: maximum value the function is scaled to

Return Value **Return Value:**

square(System.Double,System.Double) Sqaure filter. Output = Input*Input [Shifted to origin]

Parameters **p__x:** input value

p__max: maximum value the function is scaled to

Return Value **Return Value:**

InitSensor(TDxInput.Device,HidLibrary.IHidDevice) Binds the HidDataReceivedEvent to HidDataReceived() and sets parent Device object. This method is an extension of the orginal API, which is not exposed to COM. Thatswhy the Attribute ComVisible(false).

DestroySensor Unbinds the HidDataReceivedEvent from HidDataReceived(). This method is an extension of the orginal API, which is not exposed to COM. Thatswhy the Attribute ComVisible(false).

Constructor: Sensor Constructor which inits the TranslationFunction and RotationFunction attributes.

Finalize Destructor supporting IDisposeable implementation.

Dispose Suppress garbage collection of this object, take care of it yourself.

Dispose(System.Boolean) Clean up managed and unmanged ressources by hand.

Parameters **Disposing:**

A.1.13 ISensor

COM interface exposing the methods of Sensor. Forced to use the same GUID as the original driver interface does. Provides both interface implementations (Dual), IDispatch and IUnknown for early and late binding.

A.1.13.1 Properties

Translation Holds lengths of movement for all 3 axes. For COM interop this must be marshaled explicitly to Interface. The DispId attribute states the vtable entry of the method.

Rotation Holds angles for all 3 axes. For COM interop this must be marshaled explicitly to Interface. The DispId attribute states the vtable entry of the method.

Device Returns an IDispatch pointer to the parent Device object associated with the Sensor. For COM interop compatibility, the object must be marshaled to IDispatch. (See Blender Plugin) The DispId attribute states the vtable entry of the method.

Period Holds the time frame the values of Translation and Rotation are scaled to. The DispId attribute states the vtable entry of the method.

A.1.14 Device

Main device class representing the entire NDOF device encapsulating Keyboard and Sensor. Indirectly implements the _ISimpleDeviceEvents_Event interface via the ComSourceInterfaces attribute in order to properly expose events to COM. Implements the IDisposeable interface which helps to properly dispose this object and its events. Implements the ISimpleDevice interface which exposes its methods to COM. Because of that the ClassInterfaceType must be "None", otherwise the CLR provides another class interface (and does not use IAngleAxis). The class is forced to use the same GUID as the original CoClass does in order to be accessible by binary-only clients.

A.1.14.1 Fields

m_prefs Object which encapsulates LoadPreferencesName() functionality.

m_HidControl Used to communicate with the sigleton instance of HidControl for searching devices and registering DeviceChange events.

m_IHidDevice Object of HidLibrary used to raise events when Hid Reports are recieved.

m_HidControlEvents_DeviceChangeEventHandler DeviceChange event handler member variable storing the function pointer (delegate) to the method m_HidControl will call back if a DeviceChange is detected.

m_lock Binary semaphore for locking event access.

A.1.14.2 Properties

Sensor Returns the associated Sensor object.

Keyboard Returns the associated Keyboard object.

Type Returns type of the connected device.

IsConnected Returns true if a device is connected with the driver.

A.1.14.3 Methods

IsAerion(HidLibrary.IHidDeviceInfo) Predicate which queries p_IHidDeviceInfo.VendorID and p_IHidDeviceInfo.ProductID and returns true if an Aerion device is connected.

Parameters **p_IHidDeviceInfo:**

Return Value **Return Value:** boolean

IsSpaceExplorer(HidLibrary.IHidDeviceInfo) Predicate which queries p_IHidDeviceInfo.VendorID and p_IHidDeviceInfo.ProductID and returns true if a SpaceExplorer device is connected.

Parameters **p_IHidDeviceInfo:**

Return Value **Return Value:** boolean

IsSpaceNavigator(HidLibrary.IHidDeviceInfo) Predicate which queries p_IHidDeviceInfo.VendorID and p_IHidDeviceInfo.ProductID and returns true if a SpaceNavigator device is connected.

Parameters **p_IHidDeviceInfo:**

Return Value **Return Value:** boolean

IsSpaceTraveler(HidLibrary.IHidDeviceInfo) Predicate which queries p_IHidDeviceInfo.VendorID and p_IHidDeviceInfo.ProductID and returns true if a SpaceTraveler device is connected.

Parameters **p_IHidDeviceInfo:**

Return Value **Return Value:** boolean

IsSpacePilot(HidLibrary.IHidDeviceInfo) Predicate which queries p_IHidDeviceInfo.VendorID and p_IHidDeviceInfo.ProductID and returns true if a SpacePilot device is connected.

Parameters **p_IHidDeviceInfo:**

Return Value **Return Value:** boolean

ConnectAerion Connects the AerionInput main instance with a plugged NDOF-device by passing a predicate. Instructs m_IHidDevice to handle incoming data and sets the global Type of the device.

Return Value **Return Value:** true if device is connected

ConnectSpaceExplorer Connects the AerionInput main instance with a plugged NDOF-device by passing a predicate. Instructs m_IHidDevice to handle incoming data and sets the global Type of the device.

Return Value **Return Value:** true if device is connected

ConnectSpaceTraveler Connects the AerionInput main instance with a plugged NDOF-device by passing a predicate. Instructs m_IHidDevice to handle incoming data and sets the global Type of the device.

Return Value **Return Value:** true if device is connected

ConnectSpaceNavigator Connects the AerionInput main instance with a plugged NDOF-device by passing a predicate. Instructs m_IHidDevice to handle incoming data and sets the global Type of the device.

Return Value **Return Value:** true if device is connected

ConnectSpacePilot Connects the AerionInput main instance with a plugged NDOF-device by passing a predicate. Instructs m_IHidDevice to handle incoming data and sets the global Type of the device.

Return Value **Return Value:** true if device is connected

HidControlDeviceChange(System.Object,System.EventArgs) Handles the HidControlDeviceChange event. First disconnects the current device from the driver, then calls Connect() which looks for new devices. Fires DeviceChange event in order to notify clients.

Parameters **sender:**

e:

LoadPreferences(System.String) Loads the application profile in argument preferencesName and updates all relevant fields. For now save the config file, as the Blender plugin does not disconnect (and thereby save the config file) correctly.

Connect Look for connected supported devices and register for HidControl.DeviceChange event to be notified if devices are connected. If a supported device is found init the Sensor and the Keyboard. Then load initial "default" profile.

Disconnect Disconnect a connected device. Stop the HidDevice processing input data, and deregister from HidDevice.DeviceChange event handler. Save the config file.

Constructor: Device Default constructor needed by COM to create objects.

Finalize Destructor supporting IDisposeable implementation.

Dispose Suppress garbage collection of this object, take care of it yourself.

Dispose(System.Boolean) Clean up managed and unmanged ressources by hand.

Parameters **Disposing:**

A.1.15 _ISensorEvents_SensorInputEventHandler

Delegate declaration for SensorInput event handler. Only states the return type of the event which is void for COM events and the signature of the event. This is implicitly COM visible and needed for COM to determine the signature of the event.

A.1.16 _ISensorEvents

COM Events publisher interface forced to use the same GUID as the original driver interface does. Must be IDispatch Interface for late binding, not standard dual.

A.1.16.1 Methods

SensorInput The SensorInput event is fired whenever the Sensor recieves valid data from the device.

A.1.17 _IKeyboardEvents_KeyDownEventHandler

Delegate declaration for KeyDown event handler. Only states the return type of the event which is void for COM events and the signature of the event. This is implicitly COM visible and needed for COM to determine the signature of the event.

keyCode: KeyCode of the button which is pressed down.

A.1.18 _IKeyboardEvents_KeyUpEventHandler

Delegate declaration for KeyUp event handler. Only states the return type of the event which is void for COM events and the signature of the event. This is implicitly COM visible and needed for COM to determine the signature of the event.

keyCode: KeyCode of the button which pulled up.

A.1.19 _IKeyboardEvents

COM Events publisher interface forced to use the same GUID as the original driver interface does. Must be IDispatch Interface for late binding, not standard dual.

A.1.19.1 Methods

KeyDown(System.Int32) The KeyDown event is fired whenever a key is pressed. The keyCode argument contains the number of the key that fired.

Parameters **keyCode:**

KeyUp(System.Int32) The KeyUp event is fired whenever a key is released. The keyCode argument contains the number of the key that fired.

Parameters **keyCode:**

A.1.20 TDxInfo

Provides information about the used Revision of the driver. Implements the ITDxInfo interface which exposes its methods to COM. Because of that the ClassInterfaceType must be "None", otherwise the CLR provides another class interface (and does not use ITDxInfo). The class is forced to use the same GUID as the original CoClass does in order to be accessible by binary-only clients.

A.1.20.1 Methods

RevisionNumber Returns the revision number of the driver. It is not implemented as a classic getter method in the original driver but as a simple constant return function.

Return Value **Return Value:** constant string c_RevisionNumber

Constructor: TDxInfo TDxInfo constructor

A.1.21 Keyboard

Represents the keys of the device. Holds boolean array with the state of all keys. Fires KeyUp and KeyDown events whenever Keys are pressed and released. Implements the IKeyboard interface which exposes its methods to COM. Because of that the ClassInterfaceType must be "None", otherwise the CLR provides another class interface (and does not use IVector3D). The class is forced to use the same GUID as the original CoClass does in order to be accessible by binary-only clients.

A.1.21.1 Fields

c_Button Usage page for Button Reports

m_KeyDownList BitArray containing one bit (0 = up, 1 = down) for each key of the device.

m_IHidDevice IHidDevice object of HidLibrary which raises events when Hid Reports are recieved.

IsDisposed IDisposeable implementation help variable.

m_HidDeviceEvents_HidDataReceivedEventHandler HidDataReceived event handler member variable storing the function pointer (delegate) to the method m_HidDevice will call back if a HidDataReceived is detected.

m_lock Binary semaphore for locking event access.

A.1.21.2 Properties

Keys Returns number of keys the device offers.

Device Returns the parent Device object associated with the Sensor.

ProgrammableKeys Returns number of programmable keys the device offers.

A.1.21.3 Methods

HidDataReceived(System.Object,HidLibrary.HidDataReceivedEventArgs) Handles HidDataReceived events and processes incoming raw data from the Device. Fills BitArray m_KeyDownList with key press data. At the end fire KeyDown or KeyUp to notify clients.

Parameters **sender:**

p__HidDataReceivedEventArgs:

GetKeyName(System.Int32) Returns the Name of the supplied key, if any.

GetKeyLabel(System.Int32) Returns the Label of the supplied key, if any.

IsKeyUp(System.Int32) Returns true if the supplied key is up at the time.

IsKeyDown(System.Int32) Returns true if the supplied key is pressed down at the time.

InitKeyboard(TDxInput.Device,HidLibrary.IHidDevice) Binds the HidDataReceivedEvent to HidDataReceived() and sets parent Device object. TODO: Send feature report to device to query number of keys. This method is an extension of the orginal API, which is not exposed to COM. Thatswhy the Attribute ComVisible(false).

DestroyKeyboard Unbinds the HidDataReceivedEvent from HidDataReceived(). This method is an extension of the orginal API, which is not exposed to COM. Thatswhy the Attribute ComVisible(false).

Constructor: Keyboard Default constructor needed by COM to create objects.

Finalize Destructor supporting IDisposeable implementation.

Dispose Suppress garbage collection of this object, take care of it yourself.

Dispose(System.Boolean) Clean up managed and unmanged ressources by hand.

Parameters **Disposing:**

A.1.22 AngleAxis

Provides rotational data in three dimensions and an angle in arbitrary units (euclidean norm). Implements the IAngleAxis interface which exposes its methods to COM. Because of that the ClassInterfaceType must be "None", otherwise the CLR provides another class interface (and does not use IAngleAxis). The class is forced to use the same GUID as the original CoClass does in order to be accessible by binary-only clients.

A.1.22.1 Properties

X Represents the X component of the rotation

Y Represents the Y component of the rotation

Z Represents the Z component of the rotation

Angle Represents the angle of the rotation (euclidean length of the three components). If set to 0.0 all three components are set to 0.0, too.

A.1.22.2 Methods

Constructor: AngleAxis AngleAxis constructor

A.2 GoogleEarth COM-API [1]

GoogleEarth[2] ist ein 3D-Programm zur Betrachtung von Satellitenbildern. Da man sich hier auf einer virtuellen Erde im dreidimensionalen Raum bewegen kann, bietet sich die GlobeFish-Maus zur Vereinfachung der Navigation an.

Für die Steuerung von GoogleEarth bieten sich verschiedene Wege an. Zum einen ist das die Simulation von Standard-Eingabegeräten (Maus, Tastatur, Joystick) und zum anderen wird eine COM-API angeboten. Eine API für Plugins existiert allerdings nicht.

Ein externer Prozess, welcher die Eingabedaten der Globefish-Maus verarbeitet und an GoogleEarth weiterleitet, ist daher in jedem Fall erforderlich.

A.2.1 COM-API

Das COM-Interface für GoogleEarth ist Bestandteil der normalen Installation und wird normalerweise auch während der Installation registriert. Die Beschreibungsdatei der Schnittstelle (IDL) und die Dokumentation dieser kann direkt von Google bezogen werden[3].

[1] Auszugskapitel von Oliver Spindler aus vorangegangenem Dokument zum Prototypenseminar
[2] GoogleEarth http://earth.google.com/intl/de/
[3] GoogleEarth COM API http://earth.google.com/comapi/

Die COM-API setzt sich aus verschiedenen Schnittstellen zur Steuerung von GoogleEarth zusammen. Die wichtigste ist die `IApplicationGE` - Schnittstelle.

Sie ist der Einstiegspunkt der API und bietet Methoden zur Steuerung der Kamera-Sicht in GoogleEarth.

Dabei sind vor allem die folgenden Methoden interessant:

```
HRESULTGetCamera (
   [in] BOOL considerTerrain,
   [out, retval] ICameraInfoGE **pCamera
)
```

Über diese Methode wird die aktuelle Instanz der Kamera-Sicht in GoogleEarth abgefragt.

```
HRESULTSetCamera (
   [in] ICameraInfoGE *camera,
[in] double speed
)
```

Diese Methode ändert die Position der Kamera mit der angegebenen Geschwindigkeit.

```
HRESULTSetCameraParams (
   [in] double lat,
   [in] double lon,
   [in] double alt,
   [in] AltitudeModeGE altMode,
   [in] double range,
   [in] double tilt,
   [in] double azimuth,
   [in] double speed
)
```

Diese Methode ändert wie `SetCamera` die Position der Kamera mit der angegebenen Geschwindigkeit. Sie soll aber gegenüber der `SetCamera`-Methode einen Performance-Vorteil durch weniger COM-Aufrufe haben.

Die Kamera-Sicht wird mit Hilfe eines Fokus-Punktes auf der Erde und Kamera-Parametern, welche relativ zu diesem Fokus-Punkt definiert sind, gesteuert.

Die folgende Aufzählung gibt einen Überblick über die verfügbaren Parameter der `ICameraInfoGE-`
Schnittstelle. In Klammern sind abweichende Variablennamen der `SetCameraParams`-Methode vermerkt.

Verfügbare Parameter:

- `FocusPointLatitude (lat)`
  - Double-Wert zwischen -90 und +90
  - gibt den Breitengrad des Fokus-Punktes an
  - -90 entspricht dem Südpol, 90 dem Nordpol
- `FocusPointLongitude (lon)`

  - Double-Wert zwischen -180 und 180
  - gibt den Längengrad des Fokus-Punktes an
  - 0 entspricht dem Nullmeridian

- `FocusPointAltitude (alt)`
  - Double Wert
  - Höhe des Fokus-Punktes in Metern

- `FocusPointAltitudeMode (altMode)`
  - Enumeration
  - RelativeToGroundAltitudeGE Höhe wird relativ zum Gelände gesetzt
  - AbsoluteAltitudeGE Höhe wird absolut (Meereshöhe) gesetzt

- `Range`
  - Double-Wert = 0
  - Abstand der Kamera zum Fokus-Punkt in Metern

- `Tilt`
  - Double Wert zwischen 0 und 90
  - Neigung der Kamera
  - ein Wert von 0 entspricht der vertikalen Draufsicht auf den Fokus-Punktes
  - ein Wert von 90 entspricht der horizontalen Ansicht des Fokus-Punktes

- `Azimuth`
  - Double Wert zwischen -180 und 180
  - Azimutwinkel der Kamera
  - bei einem Wert von 0 befindet sich der Nordpol oben auf dem Bildschirm
  - bei einem Wert von 180 befindet sich der Südpol oben auf dem Bildschirm

Abbildungsverzeichnis

1.1 Datenübergabeschema 2
1.2 Pull-Komponente 3
1.3 Push-Komponente 3
1.4 Push- und Pull-Komponenten im 3DConnexion-Treiber TDxInput 3

2.1 Klassendiagramm AerionInput 9
2.2 Schichtenmodell von AerionInput 10
2.3 Event Registrierung 11
2.4 `Event De-Registrierung` 12
2.5 Vorgänge beim Anstecken oder Entfernen eines unterstützen Geräts und Ablauf des `Device.DeviceChange`-Ereignisses 14
2.6 Übertragung von Bewegungs- bzw. Tastendruck-Eingabedaten an die Anwendung 17