Hochschule Furtwangen - Fakultät Digitale Medien

Forschungsprojekt - Medieninformatik Master

Sommersemester 2015

FUSEE mit Blender verdengeln

Fabian Gärtner, MIM2
Sarah Häfele, MIM2
Alexander Scheurer, MIM2
Linda Schey, MIM2

Prof. Christoph Müller

11. August 2015

Abstract

Diese Arbeit soll zeigen wie und ob es möglich ist die Programmierschnittstelle von Blender die nativ nur für Python bereitsteht nach C# portiert werden kann. Und wie diese verallgemeinert werden kann um als universelle Schnittstelle auch für andere 3D-Computergrafik Software verwendbar zu sein.

Inhaltsverzeichnis

1 Fragestellung

Diese Arbeit basiert auf einem vorangehenden Forschungsprojekt, in dem die Programmierschnittstelle von Cinema 4D mit Hilfe der Software SWIG von C++ nach C# übersetzt wurde und durch Anbindung von FUSEE ein Export-Plugin realisiert wurde. In dieser Arbeit soll nun gezeigt werden wie eine ähnliche Übersetzung der Programmierschnittstelle von Blender möglich ist, mit der gleichzeitigen Betrachtung der zukünftigen Anbindung von weiteren 3D-Softwareprodukten. Die so geschaffene allgemeine Programmierschnittstelle in C# wird im weiteren Uniplug genannt.

2 Vorüberlegungen

Um die Fragestellung realistisch umsetzen zu können ist es nötig schon vorab Entscheidungen über Richtung und Umfang des Projekts zu treffen.

2.1 Plugin-Umfang

Es soll Entwicklern mit Uniplug möglich sein, ohne Kenntnis der native Programmierschnittstelle der jeweiligen 3D-Software, Plugins zu entwickeln. Hier könnten im voraus Einschränkungen getroffen werden um die Entwicklung zu vereinfachen. Zum Beispiel wäre es möglich sich auf die Erzeugung von Geometrie zu beschränken.

2.2 Plugin-Aufbau

Vorab wurden zwei Ansätze für den Aufbau von Uniplug evaluiert. Außer der schon teilweise beschrieben kompletten und am besten automatisierte Übersetzung der Programmierschnittstelle gibt es noch die Möglichkeit nativ für die jeweilige Software ein Plugin zu entwickeln das dann die zu übersetzende Schnittstelle zu Uniplug bildet und somit schon einen ersten Schritt zur Normalisierung und logischen Übersetzung bietet.

2.3 Entscheidungen

Die Entscheidung fiel auf eine möglichst vollständige Übersetzung ohne Zwischenschritt über ein Plugin. Dieser Ansatz bietet aber bei Erfolg einen wesentlich direkteren und auch komfortableren Zugriff auf Seiten von Uniplug, da jegliche logische Konvertierung im Rahmen von C# direkt als teil von Uniplug erfolgen kann.

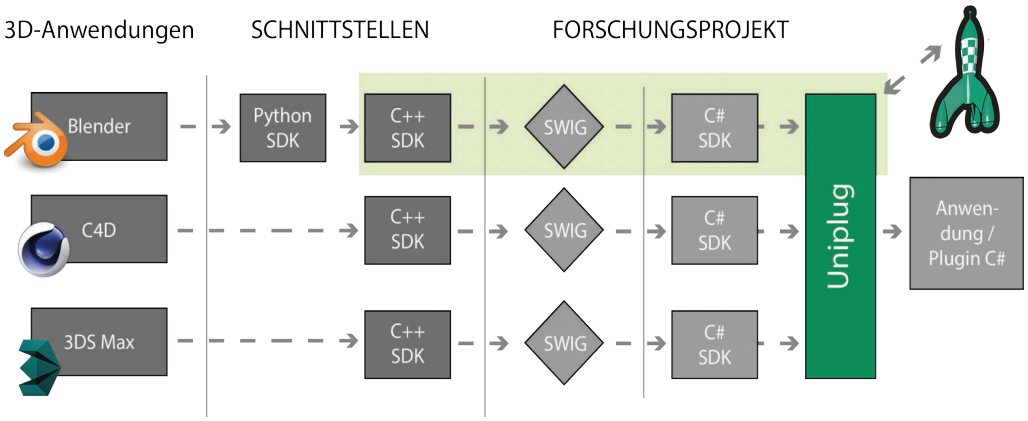


Abbildung 1: Schematische Darstellung des Aufbaus

3 Umsetzung

3.1 SWIG

Linda Schey, Sarah Häfele

Swig ist ein Programmierwerkzeug, das es ermöglicht in C/C++ geschriebenen Programmcode durch höhere Programmiersprachen anzusprechen. Neben einer Vielzahl von Programmier- und Skriptsprachen unterstützt Swig auch C# und eignet sich somit um inC++ geschrieben Programmcode mit Fusee zu verbinden. Swig wurde gewählt, da es zum einen bereits in dem Cinemea4D-Uniplug von Fusee verwendet wurde und zum anderem einem die Arbeit abnimmt einen kompletten Wrapper von Hand schreiben zu müssen, so dass man sich mittels Swig einen Wrapper generieren lassen kann. Zudem erleichtert es die Wartung von Projekten, da Updates im Code nicht von Hand und mühseliger Suche eingetragen werden müssen, sondern lediglich für die neue Generierung des Wrappers die aktuelle Headerfiles eingebunden werden müssen. Um den Wrapper zu generieren, wurde zusätzlich die Python Library eingebunden, da das Headerfile teilweise Klasse aus Python verwendet. Außerdem wurde die Fusee.Math Library referenziert, da einge Datentypen auf Fusee-Datentypen gemapt werden sollen.

3.1.1 Überblick

Swig ist nicht intuitiv verwendbar. Bei der Nutzung ist einiges zu beachten, was durch die Swig-Dokumentation nicht gleich ersichtlich ist und erst durch Trial-and-Error langsam erarbeitet werden kann (hier spielen zudem noch das Zusammentreffen von Blender und Fusee mit all ihrer Eigenheiten eine wichtige Rolle, wie später noch besser ersichtlich wird). Deshalb zunächst eine Liste mit Dingen, auf die beim Entwickeln mit Swig zu achten ist und auf deren Inhalt zum Teil später noch näher eingegangen wird.

- Den Build der Solution sowie deren Unterprojekte auf **Release** stellen: Da Blender Python als Kommunikationssprache verwendet und Python ... muss das Projekt als Release gebaut werden, da sonst ????????? Näheres zu den Buildeinstellungen in ...

- Die richtige **Build-Reihenfolge** ist unbedingt zu beachten. Zudem sollten zuvor *geswiggte* Klassen aus dem Ordner gelöscht werden, wenn an ihnen Änderungen vorgenommen wurden, um Fehler durch alte Dateien zu vermeiden. Auf die Reihenfolge soll später näher eingegangen werden (siehe ???).
- BigObj: wieso, weshalb, warum
- **Größe des Headerfiles:** Aus Sichtweise der Swig-Entwickler sind einzelne Headerfiles pro Klassen besser zu swiggen als ein großes mit allen gesammelten Klassen. Das Swiggen dauert mit einem großen File dementsprechend lang zum Bauen und tritt ein Fehler auf, muss der ganze Prozess wiederholt werden. Fehler beim Swiggen werden so erst am Ende erkannt. Zudem vermuten das Swig-Team dieses Projektes, dass weitere Komplikationen durch das zu großes File aufgetreten sind.
- Die **Unterschiede zwischen C++ und C#** müssen verstanden und beachtet werden, da SWIG nicht alle Klassen, Methoden, etc. einfach in die andere Sprache übersetzen kann. Hierzu zählen sicherlich
  - **Speicher Allokation**
  - **Garbage Collector:** Durh den Managed Code in C# können eventuell Probleme auftreten. Im unmanaged C++ -Code sind keine Vorbereitungen für einen eventuellen Garbage Collector getroffen. Wird dieser Code mit SWIG nach C# übersetzt, kann es Probleme bei Dependencies geben, dann nämlich, wenn der Garbage Collector ein Objekt entfernt, auf das ein anderes zugreifen möchte. Hier kann es zu Programmabstürzen kommen. ? Lösungsvorschläge weiter unten
  - **Default Werte:** C++ und C# gehen unterschiedlich mit Default-Werten um, wenn eine Variable nicht initialisiert wurde. C++ vergibt einen völlig beliebigen Wert, während C# meist feste Default-Werte hat. Dies ist wiederum bei der Übersetzung mit Swig zu beachten.
  - **Pointer:** Da die Anwendung C++ - in C# -Code umwandeln soll, treten hier verhäuft Fehler und Komplikationen auf. In Unterkapitel 3.1.5 soll dies eingehend behandelt werden.
- Die Spezialfälle werden in der SWIG-Doku nicht ausreichend behandelt, weswegen die Entwickler durch so genannte **Mailing Lists** kontaktiert

wurden. Der Dialog mit den Spezialisten bringt neue Erkenntnisse, kann aber nicht alle Spezialfälle klären. Hierzu später ebenfalls mehr.

- **Syntax:** Syntaxunterschiede können ebenfalls Komplikationen hervorrufen. Da Swig hauptsächlich Strings ersetzt, muss die Struktur der jeweiligen Syntax immer beachtet werden. So muss zum Beispiel in C++ der Zeigerstern (*) direkt vor dem Variablennamen stehen, wobei in C# dieser zur Typendeklaration gehört. ???
- **Debuggen mit SWIG:** Da es kein Syntax-Highlighting für SWIG in Visual Studio gibt, ist das Finden von Fehlern im Swig-Code nicht leicht. Hierzu hat sich das Anhängen von aussagekräftigen Inline-Kommentaren direkt hinter der zu ersetzenden Swig-Variablen als nützlich erwiesen. Der Kommentar wird so in den erzeugten Code mit eingebaut und es kann explizit danach gesucht werden, um Fehler nachzuverfolgen.

Ignore Genereated

3.1.2 Aufbau des Projektes in VS

Um das von Fabi erzeugte C++ Headerfile nach C# zu wrappen wurde eine Solution aufgesetzt die aus fünf Projekten Besteht und den C++ Code nach C# wrappt. Im folgenden werden diese Projekte in der Reihenfolge wie sieh für eine erfolgreiche Generierung des C# Codes gebaut werden müssen erläutert.

CppApi
: Enthält das C++ Headerfile mit allen Klassen und Funktionen. Generiert *CppApi.dll*

SWIG
: Enthält das Swig-Interface *CppApi.i* nach dessen Vorschriften besondere Datentypen geswiggt werden (mehr dazu in 3.1.4). Generiert anhand des Headerfiles entsprechenden Wrapper in C# . Diese werden dem *Cs Wrapper* Projekt hinzugefügt.

CppWrapper
: Enthält das C++ File CppApiWrapper.cpp, das ebenfalls durch Swig

Generiert wird. Es stellt die Schnittsichtelle zwischen dem C++ Code und dem C# Code dar unter Berücksichtigung der durch Swig gemarschallten Typen und deren Umwandlungen. Des weiteren wird die *CppWrapper.dll* generiert.

CsWrapper
Generiert CsWrapper.dll die die aus dem SWIG Projekt generierten C# Klassen enthält.

CsClient_START_ME
Ist lediglich ein Test Projekt mit dem der geswiggte Code in C# getestet wird. Im Hinblick auf das Uniplug entspräche dies dem Plugin?

3.1.3 Build-Einstellungen in VS

Bigobject (C++->Command Line „/bigobj“)
Release evtl.

3.1.4 Typemaps

Was sind Typemaps und wie funktionieren sie?

3.1.5 Besondere Datentypen

Besondere Datentypen sind jene, die nicht zu den Standard Datentypen in C# gewrappt werden, oder gar keine Standarddatentypen sind. Hier sind das spezielle Typen zu Darstellung von Vektoren oder Matrizen im 3-dimensionalen Raum. Dafür wurden verschiedene Datentypen ausprobiert. Zu Beginn wurde begonnen einen 3-dimensionalen Vektor als Feld mit drei Elementen darzustellen. Dafür wurde im Swig-Interface CppApi.i ein extra Typemap angelegt, das diese dreielementige Feld in den Fusse Datyntyp float3 mapt. Da hier aber Probleme auftraten, da Felder in C++ nicht als Rückgabe wert einer Funktion zulässig sind, sondern nur als Pointer, C# aber keine Pointer unterstützt, wurde die der Datentyp bei der Erstellung des Headerfiles in den Datentyp std::array<float, 3> geändert. Auch für diesn Datentyp wurde zunächst ein entsprechendes Typemap erstellt, das teilweise funktionierte. Jedoch wurde bei der Verwendung eines std::array<> als Parameter einer Funktion übergeben

dann ein Ecxeption geworfen, da für std::array NULL bzw. 0 zurückgegeben werden kann, was bei einer Standard-Exceptio in Swig der Fall ist. So wurde wieder der Datentyp geändert, diesmal wurde ein extra Struct definiert, das die Eigenschaften eines »Plain old Datatype« aufweist, so dass anstatt eines Objektes auch NULL zurück gegeben werden kann. Über die Swig-User-Mailingliste kam letztendlich der Hinwei,d as im Typemap festzulegen ist, das für der Parameter »out« außerdem festlegen kann, was als »null« zurückgeben wird, hier ein leeres Objekt des Strucs. Dies Lösung hätte auch für das std::array funktioniert, so das nicht erst ein extra neuer Datentyp eingeführte werden müsste. Es wurde aber beim Struct geblieben, da im Hinblick auf das Ziel ein Uniplug zu erstellen, man so bei der Erstellung des Headefiles, von den unterschiedlichen Modellierungsprogrammen erst die verschiedenen Datentypen die Vektoren oder Matrizen darstellen in einen einheitlich C++ Typ umwandeln kann, so das hier das Typemap für jedes Modellierungsprogramm das selbe wäre. Datentypen die im aktuellen *CppApi.i* Interface mit Typemaps nach C# und dem entsprechenden Typ in Fusee gemapt werden sind:

- FVector2 nach Fusee.Math.float2
- FVector3 nach Fusee.Math.float3
- FVector4 nach Fusee.Math.float4
- VFLOAT16 nach Fusee.Math.float4x4

- std::vector, std::map

Zeigertypen können in C# nur im unsafe-Kontext verwendet werden. Zudem unterliegen sie weiteren Einschränkungen (Boxing und Unboxing wird nicht unterstützt und sie erben nicht von object). Diese und weitere Gründe bewegen dazu, mit SWIG aus den c++-Pointern andere, entsprechende Typen in C# zu generieren. Dazu müssen die Objekte, die hinter den betreffenden Klassen stehen, analysiert werden. Was repräsentiert der Objekttyp in Blender? Wie ist dieses Konzept in Fusee umgesetzt? Evtl hier noch ein Beispiel mit Fusee Float3 oder so. Danach typemaps dazu

–> nicht nullbar, wird aber wie Pointer behandelt. Exception-Fehler usw.

3.1.6 Das SWIG Interface File

Das SWIG Interace File CppApi.i enthält neben den oben genannte Typmaps noch weitere . string map(mit problemen)&vector ignore-Regex

3.1.7 GC

3.2 Logische Übersetzung

4 Fazit und Ausblick

Technische Umsetzung wahrscheinlich möglich allerdings sehr hoher Aufwand bei Wartung und Anbindung neuer Software. Logische Umsetzung sehr Komplex, je höher die Betrachtungsweite an Software ist.

Andere Herrangehensweise wählen ... Zwischenlayer nativ für die Software zugeschnitten, dann Übersetzung nach C# und Anbindung zu Fusee.

FUSEE Universal Plugin für 3D-Modellierungssoftware

Untersucht werden soll die Machbarkeit einer Schnittstelle, die es ermöglicht, universell einsetzbare Plugins für 3D-Anwendungen mit C# und Fusee zu entwickeln. Dabei soll auch analysiert werden, welche Anforderungen ein solches Plugin erfüllen muss und welche Möglichkeiten sich dadurch für den Entwickler ergeben.

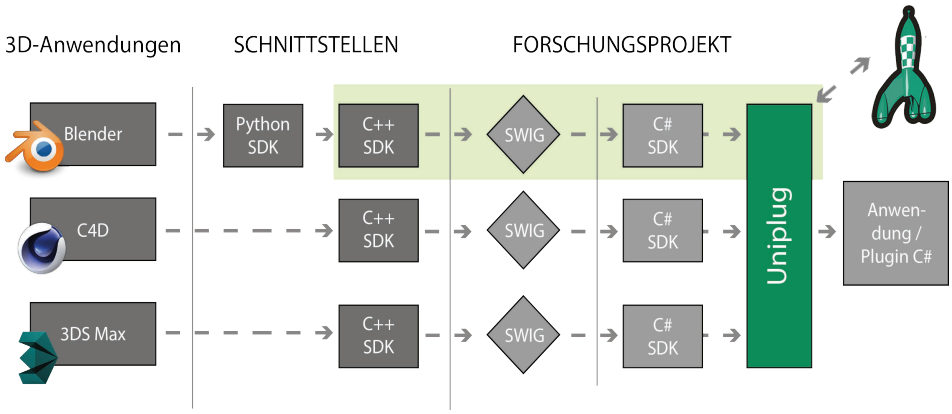
Methodik & Vorgehensweise

Zunächst wird das Vorhaben an der 3D-Anwendung Blender exemplarisch umgesetzt und getestet. Anschließend wird von diesem Spezial- auf den Allgemeinfall geschlossen, indem die Gemeinsamkeiten zwischen verschiedenen Modellierungsprogrammen analysiert werden.

Organisation & Management

Versioning Control mit GitHub und SVN. Aufteilung in kleinere Expertengruppen und wöchentliche interne Meetings. Austausch mit den Entwicklern der verwendeten Softwarepakete. Dokumentation der Arbeiten im Quelltext und in Form eines wissenschaftlichen Papers.

Architektur & Umsetzung



Um aus C# auf die Funktionalitäten der 3D-Anwendungen zugreifen zu können, die in der Regel in C++ implementiert sind, muss mit dem Programmierwerkzeug SWIG eine C#-Schnittstelle generiert werden. Im speziellen Fall von Blender ist es zudem notwendig, eine zusätzliche C++-Schicht zu entwickeln. Der so generierte Code wird in einer Bibliothek (Uniplug) zusammengeführt, sodass dem Entwickler einheitliche Funktionen zur Entwicklung von Plugins für 3D-Anwendungen mit C# und Fusee zur Verfügung stehen.

Bisherige Erkenntnisse

Ein solches „Uniplug" ist machbar und würde eine effiziente Entwicklung von Plugins für 3D-Anwendungen ermöglichen. Da aber jede Modellierungssoftware spezielle Anforderungen stellt, steigt der Implementierungsaufwand mit der Anzahl der unterstützten Anwendungen. So verfügt Blender beispielsweise über keine C++-Schnittstelle, Cinema4D hingegen ist nicht quelloffen und daher schwerer anzupassen. Andere Ansätze oder die Beschränkung auf Kernfunktionalitäten könnten den Aufwand verringern.

Forschungsprojekt Prof. Christoph Müller, MIM SoSe 2015:
Linda Schey, Fabian Gärtner, Sarah Häfele, Alexander Scheurer

HOCHSCHULE FURTWANGEN UNIVERSITY HFU