Hochschule Furtwangen - Fakultät Digitale Medien

Forschungsprojekt - Medieninformatik Master

Sommersemester 2015

FUSEE Universal Plugin für 3D-Modellierungssoftware

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

Prof. Christoph Müller

3. September 2015

Abstract

Diese Arbeit soll zeigen wie und ob es möglich ist, die Programmierschnittstelle der 3D-Grafiksoftware Blender, die nativ nur für Python bereitsteht, nach C# zu portieren. Zudem soll untersucht werden, wie diese Schnittstelle verallgemeinert werden kann, um als universelle Schnittstelle auch für andere 3D-Computergrafik-Software verwendbar zu werden.

Inhaltsverzeichnis

1 Fragestellung

Alexander Scheurer

Diese Arbeit basiert auf einem vorangehenden Forschungsprojekt, in dem die Programmierschnittstelle der 3D-Grafiksoftware Cinema 4D mit Hilfe des Programmierwerkzeugs SWIG von C++ nach C# übersetzt und durch Anbindung von FUSEE[1] ein Export-Plugin realisiert wurde. In dieser Arbeit soll nun gezeigt werden, wie eine ähnliche Übersetzung der Programmierschnittstelle der 3D-Grafiksoftware 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.

[1] An der Hochschule Furtwangen entwickelte 3D-Echtzeit-Engine (siehe `http://fusee3d.org`).

2 Vorüberlegungen

Alexander Scheurer

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 nativen 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-Arten

Um abschätzen zu können, welche Funktionen in das Uniplug mit aufgenommen werden müssen, muss vorher überlegt werden, welche Plugin-Arten ein Plugin-Entwickler eventuell programmieren möchte. Ein 3D-Grafikprogramm ist recht vielseitig und bringt daher viele Möglichkeiten für Erweiterungen mit sich. Hierzu zählen sicherlich Import- und Export-Möglichkeiten für individuelle Datentypen. Dies ist besonders in Hinblick auf FUSEE interessant. Hierbei ist wichtig zu überlegen, welche Daten in der exportierten Datei gespeichert werden sollen und in welchem Format sie weitergegeben werden müssen. Dies hängt natürlich von den Programmen ab, die diese weiterverwenden sollen. Hier sind nicht nur Objektgrößen wichtig, sondern eventuell auch Hierarchiebäume, Lichtberechnungen, Animationen und vieles mehr.

Weitere Plugins könnten 3D-Programme durch verschiedene Funktionen erweitern. Hier wären neue Mesh-Operationen, neue Standardobjekte oder andere Zugriffe auf den Szenengraph denkbar. Interessant könnten weitere Parameter oder Objekttypen sein.

Bei der Überlegung muss immer bedacht werden, dass Daten in verschiedenen Programmen unterschiedlich gespeichert und gehandhabt werden. Allein die in der Projektgruppe gesammelten möglichen Plugin-Funktionen stellen eine schier endlose Fülle an Möglichkeiten dar, die alle ihre eigenen Komplikationen

mit sich bringen. Hier wurde schnell klar, dass nicht alle Funktionen gleich berücksichtigt werden können.

2.3 Plugin-Aufbau

Vorab wurden zwei Ansätze für den Aufbau von Uniplug evaluiert. Außer der schon teilweise beschriebenen kompletten und am besten automatisierten 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. Weitere Informationen zu den einzelnen Verfahren finden sich in Abschnitt 3.1.

2.4 Entscheidungen

Die Entscheidung fiel auf eine möglichst vollständige Übersetzung ohne Zwischenschritt über ein Plugin. Dieser Ansatz bietet 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.

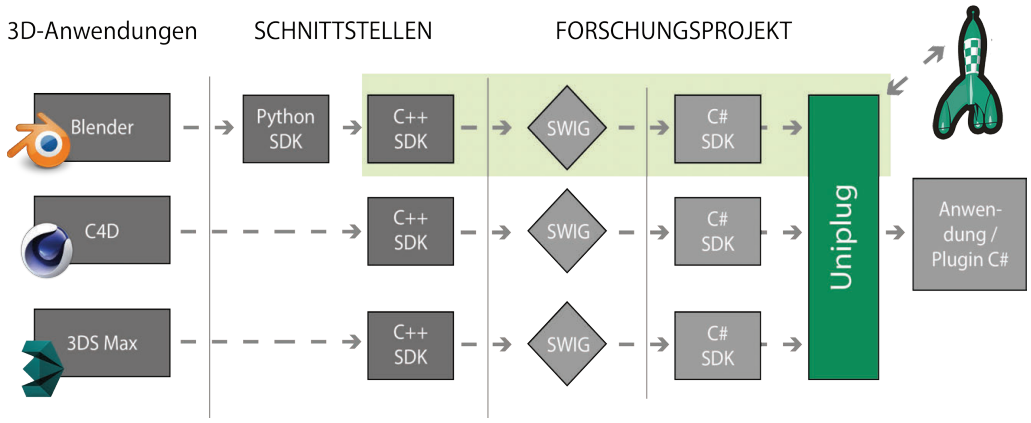


Abbildung 1: Schematische Darstellung des Aufbaus

3 Umsetzung

3.1 C++-API für Blender: die Header-Datei

Fabian Gärtner

3.1.1 Vorüberlegung

Da Blender im Gegensatz zu 3D-Modellierungsprogrammen wie Cinema4D oder 3ds Max nur über eine Python-API, nicht aber über eine C++ -API verfügt, musste zunächst eine Möglichkeit gefunden werden, Blender dennoch mittels C++ anzusteuern. Zwar ist Blender quelloffen, sodass hier über Modifizierungen des Quelltexts pluginartige Erweiterungen vorgenommen werden könnten, dies hätte aber zur Folge, dass Nutzer des Uniplugs gezwungen wären, eine bestimmte Blender-Version zu verwenden oder gar Blender selbst zu bauen (siehe dazu den in Abschnitt 3.1.2.1 beschriebenen Aufwand). Um diese Problematik zu vermeiden, wurden im Laufe des Projektes verschiedene Methoden evaluiert, mit denen von C++ aus die Python-API von Blender aufgerufen werden kann.

Eine der ersten Versuche bezogen sich dabei auf das Python-Modul *ctypes*, das es erlaubt, in Python externe C- bzw. C++ -DLLs zu laden und die in den DLL implementierte Funktionalität auszuführen. Dies stellte sich allerdings schnell aus mehreren Gründen als ungeeignet für das Projekt heraus. Zum einen hätte das System hauptsächlich nur einseitig funktioniert, das heißt von Python aus hätten C++ -Funktionen aufgerufen werden können, aber keine Python-Funktionen von C++ aus. Einzig über Callbacks, die aber letztlich auch wieder von Python heraus gesetzt und aufgerufen werden müssen, wäre dies möglich gewesen. Zum anderen aber hätte bei jeder Kommunikation auch eine Typumwandlung stattfinden müssen, da ctypes mit speziellen Datentypen arbeitet, die nicht implizit in C-Datentypen konvertiert werden können.

Ein ganz allgemeines weiteres Problem war die Tatsache, dass zunächst keine Liste der Funktionen der Blender-API verfügbar war. Diese wird aber benötigt, damit SWIG daraus entsprechende C# -Funktionen erzeugen kann. Über die Python-Funktion *dir()* hätte zwar ein geladenes Plugin über alle Typen und deren Funktionalitäten iterieren können, dies wäre aber weder effektiv noch

zielführend gewesen. Dann wären nämlich die jeweiligen Funktionen, nicht aber deren Parameter bzw. die Typen der Parameter bekannt gewesen. Zur Lösung dieses Problems beschäftigte sich ein erster Versuch damit, die im Blender-Quelltext (bzw. insbesondere im Quelltext des Tools *MakesRNA*) vorhandenen Dateien zu parsen, da diese jegliche später in der Blender-API verfügbare Funktionalität in einer speziellen Syntax und durch Erweiterung von Python deklarieren. Listing 1 zeigt dies beispielhaft anhand einer Property der Klasse *Object*. Deklariert werden hier Typ, Beschreibung und andere Eigenschaften. Ein erstes auf regulären Ausdrücken basiertes Tool zeigte allerdings Probleme bei Strukturen, deren Deklaration von der Regel abwich. Alles in allem wäre diese Vorgehensweise daher zu aufwändig und zu fehleranfällig gewesen.

```
1    prop = RNA_def_property(srna, "location", PROP_FLOAT,
     PROP_TRANSLATION);
2    RNA_def_property_float_sdna(prop, NULL, "loc");
3    RNA_def_property_editable_array_func(prop, "
     rna_Object_location_editable");
4    RNA_def_property_ui_text(prop, "Location", "Location of the
     object");
5    RNA_def_property_ui_range(prop, -FLT_MAX, FLT_MAX, 1,
     RNA_TRANSLATION_PREC_DEFAULT);
6    RNA_def_property_update(prop, NC_OBJECT | ND_TRANSFORM, "
     rna_Object_internal_update");
```

Listing 1: Deklaration der Property *location* der Klasse *Object* in *rna_object.c*

3.1.2 MakesRNA

Stattdessen wurde daher das in C geschriebene Tool MakesRNA genauer untersucht, da dieses die angesprochenen Dateien nicht nur oberflächlich parst, sondern direkt ausführt und so eine Reihe von weiteren Quelltext-Dateien erzeugt, die beim späteren Bau von Blender Python um eben jene Funktionalität erweitern. Daraus ergaben sich zwei Erkenntnisse: Zum einen zeigte sich so, dass Python relativ simpel um eigene C++ -Module erweitert werden kann[2] (dies ist auch der Hauptzweck des MakesRNA-Tools) und zum anderen, dass ein Umbau des MakesRNA-Tools das gewünschte Ziel, nämlich eine Liste aller in Blender vorhandenen API-Strukturen zu erzeugen, ermöglichen würde. Die Idee war daher, mit Hilfe dieses Tools eine »simple« Header-Datei zu erzeugen, in der

[2]Siehe hierzu auch `https://docs.python.org/3.5/extending/index.html`

jegliche Blender-API-Klassen, -Methoden, -Properties, etc. in C++ deklariert sind und die diese Aufrufe durch Einbindung der Python-Header-Datei mittels Python-Funktionen wie *PyObject_GetAttrString* oder *PyObject_CallMethod* an Python und damit an Blender weiterreicht. So sollte also, vereinfacht gesagt, bei Ausführung des Tools automatisiert ein C++ -Wrapper für die Blender-API erstellt werden. Da das MakesRNA-Tool aber Teil des kompletten Blender-Projekts ist und, wie eben erwähnt, benötigt wird, um Blender selbst zu bauen, wurde von der Forschungsgruppe festgelegt, dass dieses Tool lediglich durch einen speziellen FUSEE-Modus erweitert werden soll, ohne dabei die eigentliche Funktionalität des Tools im normalen Einsatz zu stören.

Im Folgenden nun zunächst eine Anleitung, um den Quelltext von Blender und damit auch den Quelltext von MakesRNA zu beziehen. Anschließend folgt dann eine nähere Erläuterung der während des Forschungsprojekts vorgenommenen Erweiterungen zur Erreichung der angestrebten Funktionalität.

3.1.2.1 Bezug des Quelltextes

Blender verwendet Git[3] zur Versionierung seines Quelltextes. Allerdings werden zum Bau der Unterprojekte eine Vielzahl an externen Bibliotheken benötigt. Da diese mit einer Gesamtgröße von rund 10 Gigabyte nicht mit Git versioniert werden können, verwendet Blender zusätzlich zum eigenen Git-Server auch einen SVN-Server[4]. Entsprechend komplex ist es, das Projekt und alle Abhängigkeiten korrekt zu beziehen. Hinzu kommt, dass anschließend mittels CMake zunächst die entsprechenden Projektdateien für den gewünschten Compiler erzeugt werden müssen. Um diesen kompletten Prozess zu vereinfachen, wurde daher von der Forschungsgruppe zunächst ein einfaches Shell-Script erstellt, das alle benötigten Dateien herunterlädt und anschließend eine Solution für Visual-Studio 2013 mittels CMake erzeugt. Des Weiteren wurden zwei Repositories auf GitHub angelegt. Das Repository *blender*[5] ist ein Fork vom offiziellen Repository inklusive aller Änderungen, die im Rahmen dieses Forschungsprojektes vorgenommen wurden. Dieses Repository ist als Submodule in das Repository *MotherBase*[6] eingebunden, das neben dem eben erwähnten Skript auch die benötigten Ordner und eine aktuelle Version von CMake bereitstellt.

[3] `https://git.blender.org/gitweb/`
[4] `https://svn.blender.org/svnroot/bf-blender/trunk/`
[5] `https://github.com/InformatischesQuartett/blender`
[6] `https://github.com/InformatischesQuartett/MotherBase`

Es sind daher nun nur noch wenige einfache Schritte notwendig, bevor Blender (und die Unterprojekte wie MakesRNA) gebaut werden kann beziehungsweise können. Als erster Schritt muss ein Git- und ein SVN-Tool installiert werden und die Pfade zu diesen Tools müssen der Umgebungsvariablen *PATH* hinzugefügt werden. Anschließend kann das MotherBase-Repository bezogen und über den Befehl *./make.sh* aus der Git-Konsole heraus die »Installation« gestartet werden. Das Submodule blender wird selbstständig geupdated, ebenso werden mittels SVN alle notwendigen Abhängigkeiten heruntergeladen oder aktualisiert und anschließend über CMake die Solution erstellt. Dieser Prozess kann aufgrund der Größe der Dateien einige Zeit in Anspruch nehmen. Anschließend sollte im Ordner *cbuild* die Solution *Blender.sln* liegen, die nun als x64-Release gebaut werden könnte. Zum regelmäßigen Aktualisieren des Forks stehen die beiden Dateien *update1.sh* und *update2.sh* zur Verfügung. Diese sollten in dieser Reihenfolge über die Git-Konsole ausgeführt werden. Da es zu Merge-Konflikten kommen könnte, ist der Updateprozess zweigeteilt. Nach Ausführen von update1.sh können so erst mögliche Konflikte behoben werden und anschließend das Update fortgesetzt werden. Abschließend aktualisieren die Skripte automatisch die beiden Repositories und pushen die Änderungen selbstständig mit Datumsangabe in der Commit-Message.

3.1.2.2 FUSEE-Modus im Projekt MakesRNA

Innerhalb der Blender.sln befindet sich das Projekt MakesRNA, wobei die für das Forschungsprojekt notwendigen Änderungen hauptsächlich an der Datei *makesrna.c* vorgenommen wurden. Hier wurden zwei neue Kommandozeilenparameter *--fusee* und *--nocomment* eingeführt, die das Tool in den FUSEE-Modus versetzen. Alle relevanten Funktionen im Quelltext des MakesRNA-Projektes fragen den aktuellen Modus ab und erzeugen so ggf. statt den vielen verschiedenen für Blender notwendigen Dateien nur noch eine einzige Header-Datei, die wie oben erwähnt die Blender-API wrappt. Diese Datei kann auch direkt von C++ aus (also unabhängig von FUSEE und Uniplug) verwendet werden. Daher handelt es sich hier bereits um eine vollständige C++ -API für Blender, die all das ermöglicht, was auch aus Python mittels Blender-API möglich wäre.

Im Folgenden nun eine genauere Übersicht über alle Unterschiede und vorgenommen Änderungen an Datentypen, Funktionen, etc. im Vergleich zum normalen Modus des MakesRNA-Tools.

3.1.2.3 Datentypen

Da die von MakesRNA üblicherweise erstellen Dateien in C-Syntax geschrieben sind, werden im normalen Modus keine C++ -Datentypen verwendet. Des Weiteren werden hier normalerweise mittels #define-Direktiven eigene Datentypen definiert, die spezielle Speichermechanismen implementieren. Da dies für dieses Forschungsprojekt und für das Uniplug irrelevant ist und die Header-Datei im Vergleich zu C modernere C++ -Syntax verwenden kann, wurden Änderungen an den Datentypen vorgenommen. In aller Regel werden Basistypen verwendet (*int, bool, float*) mit Ausnahme von Arrays, *Collections* und Strings in Form von *char*-Pointern. Letztere wurden durch *std::strings* ersetzt. Für Arrays werden im normalen Modus in der Regel Pointer eingesetzt (bspw. zur Rückgabe von Array-Daten bei Funktionen). Da aber auf Pointer verzichtet werden sollte, um Speicherlecks zu vermeiden, war zunächst geplant, diese durch das in C++ bekannte *std::array<>* zu ersetzen. SWIG hat allerdings Probleme std::array (und im allgemeinen Nicht-POD-Strukturen) korrekt nach C# zu übersetzen und im späteren Prozess daraus die von FUSEE bekannten Datentypen *float2, float3*, usw. zu generieren. Daher deklariert das MakesRNA-Tool für alle möglichen Array-Arten (int-, float-, bool-Arrays mit Längen 1, 2, 3, etc.) eigene Datentypen im Stil *VFLOAT3* (entspricht *std::array<float, 3>*, bzw. *float[3]* bzw. *float3*). Dazu werden im Vorfeld alle Properties und alle Parameter von Funktionen analysiert und die entsprechend notwendigen Datentypen mittels #define-Direktive zu Beginn der Header-Datei deklariert. Siehe dazu Listing 2. Die entsprechenden Structs verfügen über ein *data*-Feld, das das eigentliche Array darstellt und über Operator-Overloading für den Subscript-Operator ([ ]), um mit der gewohnten Syntax auf die Daten des Arrays zuzugreifen. Dies hat auch zum Vorteil, dass in der gesamten Header-Datei weiterhin mit der gewohnten Array-Syntax gearbeitet werden kann und lediglich die Datentypen durch die selbstdeklarierten Datentypen ersetzt werden mussten. Der Name diese Datentypen weist den Nutzer dabei gleichzeitig auf die Anzahl an gespeicherten Werten hin, was insbesondere dadurch relevant ist, dass noch nicht für alle Vector-Datentypen ein entsprechendes FUSEE-Äquivalent existiert. Die beiden Funktionen *get_value* und *set_value* sind hauptsächlich für die spätere Übersetzung nach C# gedacht, wo diese dann vom in C# definierten Subscript-Operator aufgerufen werden.

```
#define DEFINE_VECTOR_POD(sname, stype, slength)\
    struct V##sname##slength {\
    private:
        stype data[slength];\
    public:\
        stype get_value(const int idx) { return data[idx]; }\
        void set_value(const int idx, stype value) { data[idx]
   = value; }\
        stype& operator[] (const int idx) {return data[idx];}\
    };

DEFINE_VECTOR_POD(FLOAT, float, 2);  // etc.
```

Listing 2: Deklaration der Vektordatentypen mittels #define

Dynamische Arrays (teilweise Pointer) wurden in *std::vector<>* geändert. Diese kann SWIG selbstständig in C# -Listen übersetzen. Damit erfüllen sie sowohl in C++ als auch in C# die Aufgabe, eine dynamische Menge an Daten abspeichern zu können. Collections hingegen (in der Blender-API Ansammlungen von Key-Value-Paaren mit eigenem Speichermanagement) wurden in *std::maps<string, *>* geändert. Diese können von SWIG später in C# -Dictionaries übersetzt werden, sodass bspw. Objekte in der 3D-Szene durch ihren Namen abgefragt werden können. Dazu muss SWIG allerdings eine Liste an verwendeten Key-Value-Paaren zur Verfügung gestellt werden. Daher wird im FUSEE-Modus zusätzlich eine Interface-Datei für SWIG erstellt, in der alle std::maps-Typen aufgelistet sind.

Ein weiter Sonderfall sind Enum-Typen. Da Python bzw. die Blender-API Enum-Werte nur in Form von Strings zurückliefert und entsprechend Strings bei der Zuweisung von Enum-Typ-Variablen erwartet, dies aber für Programmierer natürlich ein Problem darstellt, werden in der Header-Datei Konvertierungsfunktionen, die Enum-Werte in Strings und Strings in Enum-Werte konvertieren, deklariert. Ein Beispiel für solche Konvertierungen ist in Listing 3 zu finden. Hier werden zwei std::maps angelegt, die jeweils entweder ein String annehmen und ein int zurückliefern oder umgekehrt ein int annehmen und einen entsprechenden String zurückliefern. Dass diese Funktionen einen numerischen Wert und nicht gleich einen Enum-Typ zurückliefern, liegt am Template-Prinzip, das der std::maps-Klasse zugrunde liegt. Der Compiler legt zur Compilezeit für jedes std::map der Form *std::map<std::string, example_enum>* eine konkrete Klasse an. Dies heißt zwangsläufig, dass für jeden neuen Enum-Typ eine solche Klasse

angelegt werden müsste (und zwar jeweils einen für die Hin- und Rücktransformation). Bei ersten Tests endete dies in Compilezeiten von mehreren Minuten, teilweise auch in Speicherüberläufen. Daher wird nun stattdessen immer nur std::string und int als Key-Value-Paare verwendet und durch implizites und explizites casten (letzteres mit *static_cast<example_enum> enum_value*) bei Bedarf der entsprechende Enum-Typ erzeugt.

```
1        std::map<std::string, int>
    create_string_to_smoked_type_items() {
2            return { { "SMOKEDENSITY",
    smoked_type_items_SMOKEDENSITY }, /* ... */ };
3        };
4
5        std::map<int, std::string>
    create_smoked_type_items_to_string() {
6            return { { smoked_type_items_SMOKEDENSITY, "
    SMOKEDENSITY" }, /* ... */ } };
7        };
8
9        std::map<std::string, int> string_to_smoked_type_items
    = create_string_to_smoked_type_items();
10       std::map<int, std::string> smoked_type_items_to_string
    = create_smoked_type_items_to_string();
```

Listing 3: Konvertierung von Enum-Typen in Strings und umgekehrt

Damit nun nicht bei jeder einzelnen Übersetzung wiederum die Map neu erzeugt werden muss, wurden die entsprechenden Variablen als Klassenattribute angelegt und eine entsprechende Map-Erzeugungsfunktion implementiert, sodass diese Maps beim Erzeugen der Klasse erstellt werden. Schön wäre es hier zwar gewesen, die Attribute als *static-const*-Variablen zu deklarieren, da sie dann zum einen nach der Initialisierung nicht mehr geändert werden könnten und zum anderen klassen- statt objektspezifisch wären, dies ist aber in der jetzigen Form nicht möglich. Zum einen darf der C++ -Compiler laut den Spezifikationen keinen impliziten Copy- und Zuweisungs-Konstruktor anlegen, wenn nicht-statische Variablen als *const* deklariert wurden. Beim Erzeugen von Objekten werden die Werte aller Variablen des alten Objektes in die des neuen kopiert. Da dies bei Konstanten logischerweise nicht geht, werden in einem solchen Fall implizite Konstruktoren aus der Klasse entfernt. Zum anderen handelt es sich bei std::maps nicht um Basistypen. Daher sind diese Felder aktuell lediglich als gewöhnliche Felder angelegt.

3.1.2.4 Stukturen

Sowohl die Deklarationen als auch die Implementierungen werden in der Header-Datei vorgenommen. Sofern dies nicht anders notwendig ist (bspw. bei Abhängigkeiten zu anderen Klassen), erfolgt die Implementierung zeitgleich zur Deklaration. Im Folgenden nun eine Übersicht über gängige Programmierstrukturen und ihre Implementierung in der C++ -Header-Datei mittels MakesRNA.

Klassen Die meisten der im MakesRNA-Tool definierten Klassen werden deklariert. Die Idee hinter diesem System ist, dass bei Erzeugung eines Python-Objektes durch Aufruf einer entsprechenden Blender-API-Funktion gleichzeitig auch ein Objekt einer äquivalenten C++ -Klasse erzeugt wird, das auf das Python-Objekt verweist und Funktionsaufrufe delegiert. Lediglich einige Basisklassen (bspw. die Klasse RNA_Pointer), die nur für das interne Management zuständig sind, aber keine für den Nutzer relevante Funktionalität implementieren, werden ignoriert. Stattdessen wurde eine gemeinsame Basisklasse *pyObjRef* angelegt, die sich um Speicherung der Referenz und um das Speichermanagement kümmert. Python-Funktionen geben in aller Regel Pointer auf Objekte der Klasse *PyObject* zurück, die anschließend weiter verarbeitet werden können (bei Sequenzen z.B. durch die Abfrage der einzelnen Items, die wiederum Pointer auf PyObject-Objekte sind). Je nach internem Python-Typ können diese auch in C-Basistypen konvertiert werden. Der Pointer auf diese Objekte wird in der Property *pyobjref* gespeichert, sodass jede C++ -Klasse auf ihr aktives Äquivalent in Python verweisen kann. Da Python ein Referenzzähler-System verwendet, um den Speicher der nicht mehr referenzierten PyObject-Objekte freizugeben, muss sich auch die Header-Datei um die korrekte Referenzzählung kümmern. Der Aufruf der Python-Funktionen erhöht in der Regel den Referenzzähler für das zurückgegebene PyObject. Werden nun in der Header-Datei neue Objekte erzeugt, die jeweils das gleiche PyObject referenzieren, müssen auch hier die Referenzzähler erhöht werden. SWIG bspw. verwendet bei der späteren Übersetzung nicht nur den Standardkonstruktor, sondern ruft ebenso sowohl den Kopierkonstruktor als auch den Zuweisungskonstruktor auf. Daher müssen alle drei Konstruktorarten in der Header-Datei explizit deklariert werden, sodass in allen drei Fällen auch der Referenzzähler des referenzierten Objektes erhöht wird. Da aber alle Klassen direkt oder indirekt von pyObjRef erben, ist es ausreichend, wenn diese Fuktionalität in der pyObjRef-Klasse implementiert ist. Ebenso ist in dieser Klasse ein Destruktor deklariert, der den Referenzzähler

bei Bedarf verringert. Lediglich, wenn Python zu diesem Zeitpunkt nicht mehr initialisiert ist (PyInitialized ist dann *null*), also insbesondere wenn Blender gerade beendet wird, wird dieser Schritt übersprungen, da es ansonsten zu Access-Violations käme. Der Speicher wird in diesem Fall sowieso gerade durch Windows freigegeben.

Zudem wird in der Header-Datei eine zusätzliche Klasse *pyUniplug* implementiert, die den Einstiegspunkt zur Verwendung der Blender-C++ -API definiert. Hier wird im Konstruktor das Blender-Python-Modul geladen und die Referenz darauf gespeichert. Die Funktion *context* liefert dann ein *Context*-Objekt zurück, über das auf jegliche Blender-API-Funktionalität zugegriffen werden kann. Dieses Verhalten entspricht dem aus der Blender-Python-Konsole bekannten *from bpy import ** und *bpy.context.**. Abkürzungen zu Subtypen wie *context().scene()* könnten hier ebenfalls deklariert werden. Des Weiteren steht in der pyUniplug-Klasse eine *print*-Funktion zur Verfügung, die die mittels String-Parameter übergebene Nachricht als print-Befehl an *PyRun_SimpleString* weitergibt und so ein Debugging aus C++ oder C# ermöglicht.

Properties Properties liegen als Setter- und Getter-Funktionen vor. Die Getter-Funktionen geben den Aufruf an die Funktion *PyObject_GetAttrString* weiter, die einen Pointer auf ein PyObjekt und die abzufragende Property als String entgegennimmt. Das Ergebnis ist wie bereits beschrieben wiederum ein Pointer auf ein PyObjekt. Dieses kann dann je nach Rückgabewert entweder in einen C++ -Basistyp oder durch Aufruf der entsprechenden Konstruktoren und durch Weitergabe der Referenz an die dadurch erzeugten Objekte in ein Objekt einer in der Header-Datei deklarierten Klasse konvertiert werden. Bei den Settern wird entsprechend umgekehrt ein Basistyp oder ein Objekt angenommen und dieses zunächst in ein PyObject konvertiert und anschließend durch die Funktion *PyObject_SetAttrString* wiederum mit Referenz und Property-Name als String an Python weitergereicht.

Da diese Konvertierungen und Aufrufe sich grundsätzlich über mehrere Zeilen erstrecken (bei Konvertierung in ein Array bis zu acht Zeilen inkl. Speichermanagement) ergeben sich bei mehr als 1000 Klassen mit teilweise mehreren dutzend Properties Unmengen an sich wiederholendem Code. Daher wurde hierfür und für die im folgenden Abschnitt beschriebenen Funktionen ein System aus #define-Direktiven definiert (siehe Listing 4), das die Implementierung jeglicher Getter- bzw. Setter-Funktionen auf eine Zeile (siehe Listing 5) und damit die

Dateigröße stark reduziert. Die Namen der #define-Direktiven wurde dabei so gewählt, dass deren Funktion direkt aus dem Namen geschlossen werden kann. So ergibt sich eine optimale Mischung aus Effizienz und Code-Verständlichkeit.

```
#define PRIMITIVE_TYPES_GETTER(stype, sconv, sidentifier)\
    PyObject *val = PyObject_GetAttrString(pyobjref,
    sidentifier);\
    stype resval = sconv;\
    Py_CLEAR(val);\
    return resval;
```

Listing 4: Beispiel für die Vereinfachung durch #define-Direktiven

```
int queue_count() {
    PRIMITIVE_TYPES_GETTER(int, PyLong_AsLong(val),
        "queue_count")
}
```

Listing 5: Beispiel für die Implementierung einer Getter-Funktion

Funktionen Funktionen stellten mitunter die größte Herausforderung dar. Dies hat mehrere Gründe. Zum einen können Funktionen in Python mehrere Rückgabewerte besitzen. Das MakesRNA-Tool markiert dazu ein oder mehrere Parameter der Funktion mit einem speziellen Ausgabe-Flag. Dieses Flag wird im FUSEE-Modus verwendet, um herausfinden, bei wie vielen und insbesondere bei welchen Parametern es sich eigentlich um Rückgabewerte handelt. Am einfachsten wäre es nun gewesen, diese Parameter beispielsweise als Call-By-Reference- oder Call-by-Pointer-Parameter zu definieren. Dies hätte aber zu Problemen bei Parametern mit Default-Werten geführt, da diese am Ende der Parameterdeklaration stehen müssen. Auch wäre die Verwendung in C# möglicherweise komplexer gewesen. Daher wird im FUSEE-Modus entweder, wenn es sich nur um ein Rückgabewert handelt und die Funktion selbst als *void*-Funktion deklariert ist, dieser entsprechend als Return-Value definiert und der Parameter entfernt, oder aber, wenn es zwei oder mehr Rückgabewerte sind, automatisiert ein Struct angelegt, das alle Rückgabewerte bündelt und die entsprechenden Parameter entfernt. Es werden zudem nur Value-Typen, und nicht wie ursprünglich bei MakesRNA Pointer, auf Objekte verwendet und zurückgegeben. Dies vereinfacht das Speichermanagement vor allem in Hinblick darauf, dass diese Funktionen später in C# verwendet werden sollen. Daher

haben alle Funktionen nur noch Call-by-Value-Parameter und Value-Typen als Rückgabewerte und können entsprechend benutzerfreundlich verwendet werden.

Ein weiteres Problem bei der Implementierung der Funktionen ist, dass jegliche Parameter (Basistypen, Array, Objekte, etc.) und später auch jegliche Rückgabewerte in PyObject-Objekte bzw. wiederum zurück in Basistypen oder andere Objekte konvertiert werden müssen. Existieren mehrere Rückgabewerte, so wird das automatisch angelegte Struct mit den konvertierten Daten gefüllt und zurückgegeben. Wird das Objekt einer Klasse zurückgegeben, so wird das entsprechende Objekt erzeugt und der Pointer auf das PyObject gespeichert. Da dies teilweise zu sehr langen Implementierungen führt, wird, wie bei den Properties im letzten Abschnitt beschrieben, mittels #define-Direktiven die redundanten Code-Teile gebündelt. Damit kann die Implementierung von Funktionen auf wenige Zeilen gekürzt werden.

Die Implementierung der Funktionen erfolgt, wenn möglich, direkt bei der Deklaration innerhalb der Klassendeklaration. Verwendet eine Funktion allerdings ein Objekt einer Klasse, die zu diesem Zeitpunkt noch nicht deklariert wurde, erfolgt die Implementierung erst nach der Deklaration aller Klassen. Das MakesRNA-Tool entscheidet dynamisch, wann dies notwendig ist und wann nicht. Auch hierdurch wird die Dateigröße optimiert.

Kommentare Das MakesRNA-Tool definiert bereits für viele Klassen, Properties, Funktionen, Enums, etc. Hilfstexte, die in Blender als Tooltips in der GUI oder für die Doku der Blender-API[7] verwendet werden. Im FUSEE-Modus werden diese Texte direkt als Kommentare an den entsprechenden Stellen in der Header-Datei verwendet. Diese sind im *javadoc*-Stil[8] angelegt, sodass mit Hilfe von *Doxygen* oder anderen Programmen eine umfangreiche Dokumentation für die Header-Datei erzeugt werden könnte. Dies kann die Entwicklung eines Plugins in C# als auch eines C++ -Plugins vereinfachen. Da die Kommentare aber etwa 1 MB bei der Gesamtgröße der Header-Datei ausmachen, können diese auch deaktiviert werden, indem MakesRNA mit dem Argument *--nocomment* gestartet wird.

[7] `https://www.blender.org/api/blender_python_api_2_75_3/`
[8] `http://www.oracle.com/technetwork/articles/java/index-jsp-135444.html`

3.2 SWIG

Linda Schey, Sarah Häfele

SWIG[9] ist ein Programmierwerkzeug, das es ermöglicht in C/C++ geschriebenen Programmcode durch andere Programmiersprachen anzusprechen. Da C# von SWIG neben vielen anderen Programmier- und Skriptsprachen unterstützt wird, eignete es sich somit auch um in C++ geschrieben Programmcode mit FUSEE zu verbinden. SWIG wurde gewählt, da es zum einen bereits in einem früheren Projekt im Rahmen von FUSEE entwickelten Cinema-4D-Uniplug verwendet wurde und zum anderen die Arbeit abnimmt, einen kompletten Wrapper von Hand schreiben zu müssen, da man sich mittels SWIG einen Wrapper generieren lassen kann. Zudem erleichtert es die Wartung von Projekten, da Updates im C++ -Code nicht von Hand eingetragen werden müssen, sondern lediglich für die neue Generierung des Wrappers die aktuellen Header-Dateien eingebunden werden müssen. Um besondere Datentypen zu übersetzen, die beispielsweise in den Standard APIs von C++ und C# nicht vorhanden sind oder in einen anderen (optimierten) Datentyp umgewandelt werden sollen, müssen für diese in einem SWIG-Interface File spezielle Vorschriften definiert werden. Um den Wrapper zu generieren, muss zusätzlich die Python-Bibliothek eingebunden werden, da die Header-Datei teilweise Python-Klassen verwendet. Außerdem wird die Fusee.Math-Library referenziert, da einige Datentypen auf Fusee-Datentypen gemappt werden.

3.2.1 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 spielt zudem noch das Zusammentreffen von Blender und FUSEE mit all ihren Eigenheiten eine wichtige Rolle, wie später noch besser ersichtlich wird). Deshalb zunächst eine Liste mit Aspekten, auf die beim Entwickeln mit SWIG zu achten ist und auf deren Inhalt zum Teil im weiteren Verlauf noch näher eingegangen wird.

- Die richtige **Build-Reihenfolge** ist unbedingt zu beachten. Zudem sollten zuvor *geswiggte* Klassen bzw. die entsprechenden Dateien gelöscht

[9] `http://swig.org/`

werden, wenn an ihnen Änderungen vorgenommen wurden, um Fehler durch alte Dateien zu vermeiden. Auf die Reihenfolge soll später näher eingegangen werden (siehe Abschnitt 3.2.2).

- **Größe der Header-Datei:** Aus Sichtweise der SWIG-Entwickler sind einzelne Header-Dateien pro Klassen besser zu swiggen als eine große Datei mit allen gesammelten Klassen. Das Swiggen dauert mit einem großen File dementsprechend lang. Tritt ein Fehler auf, muss der gesamte Prozess wiederholt werden. Fehler beim Swiggen werden so erst am Ende erkannt.

- 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 und Pointer:** Da die Anwendung C++ - in C# -Code umwandeln soll, können Fehler bei der Speicherbelegung auftreten. Um dies so weit als möglich ausschließen zu können, wurde auf die Verwendung von Pointern verzichtet.
  - **Garbage Collector:** Durch 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.
  - **Default-Werte:** C++ und C# gehen unterschiedlich mit Default-Werten um, wenn eine Variable nicht initialisiert wurde. Bei C++ ist kein Default-Wert spezifiziert, sodass der Wert von der Implementierung und Laufzeit abhängt. C# vergibt dagegen immer einen Default-Wert. Dies ist wiederum bei der Übersetzung mit SWIG zu beachten. Dieses Problem entstand beim Übersetzten von Klassen, die nicht dem Schema eines PODs entsprechen (vgl. Abschnitt 3.2.3).

- Spezialfälle werden in der SWIG-Doku nicht ausreichend behandelt, weswegen die Entwickler durch sogenannte **Mailing Lists** kontaktiert wurden. Der Dialog mit den Spezialisten brachte neue Erkenntnisse, konnte jedoch nicht alle Spezialfälle klären.

- **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.

3.2.2 Aufbau des Projektes in VS

Um die C++ -Header-Datei nach C# zu wrappen wurde eine Solution aufgesetzt, die aus sieben Projekten besteht. Im Folgenden werden diese Projekte kurz erläutert.

CppApi
: Enthält die C++ -Header-Datei mit allen Klassen und Funktionen. Generiert *CppApi.dll*.

SWIG
: Enthält drei SWIG-Interfaces (*CppApi.i, POD_Mapping.i und uniplug_blender_api_swig.i*) nach deren Vorschriften besondere Datentypen geswiggt werden (mehr dazu in Abschnitt 3.2.3). Anhand der Header-Datei werden die entsprechenden Wrapper in C# generiert. Diese werden dem *CsWrapper*-Projekt hinzugefügt.

CppWrapper
: Enthält die C++ -Datei CppApiWrapper.cpp, die ebenfalls durch SWIG generiert wird. Sie stellt die Schnittstelle zwischen dem C++ -Code und dem C# -Code dar, unter Berücksichtigung der durch SWIG gemarshallten Typen und deren Umwandlungen. Des Weiteren wird die Datei *CppWrapper.dll* generiert.

CsWrapper
: Generiert CsWrapper.dll, welche die aus dem SWIG-Projekt generierten C# -Klassen enthält.

CsClient
: Dieses Projekt ist das Plugin, das der Entwickler schreiben kann. Die Init-Funktion wird von Blender aus bei der Initialisierung oder Ausführung

des Plugins aufgerufen. Im vorliegenden Projekt liegt hier ein einfaches Beispiel-Plugin vor.

ManagedBridge
Brücke zwischen dem C# -Plugin und dem C++ -Plugin für Blender. Es handelt sich dabei um ein Managed-C++ -Projekt, welches das C# -Plugin referenziert, dessen Funktionen aufruft und die entsprechenden Funktionsaufrufe exportiert, sodass diese von C++ aus aufgerufen werden können.

BlenderPlug
Das Projekt BlenderPlug ist das eigentliche C++ -Modul, das in Blender als Python-Erweiterungs-Modul geladen werden kann und eine Init-Funktion bereitstellt, um über die ManagedBridge hinweg auf die C# -Funktionalität zuzugreifen. Der relativ kurze C++ -Quelltext dieses Programms richtet sich nach den Vorgaben von Python[10] und muss mittels Python-Compiler kompiliert werden. Ein entsprechendes Script und eine entsprechende Konfigurationsdatei (*setup.py*) werden im Post-Build-Prozess aufgerufen. Die daraus entstehende pyd-Datei (letztlich eine C++ -DLL) wird dann zusammen mit allen anderen Projektdateien in den Unterordner *Fusee* im Addon-Ordner im Blender-Hauptverzeichnis und in das Blender-Hauptverzeichnis selbst kopiert. In Blender kann dann in der Python-Konsole über *from fusee import uniplug* und *uniplug.init()* die Init-Funktion des Plugins ausgeführt werden. Siehe hierzu auch das angehängte Tutorial zur Erstellung eines eigenen Plugins.

3.2.3 SWIG-Interface Files

Da in diesem Projekt mit Blender bzw. FUSEE spezifischen Datentypen gearbeitet wird, die weder in der Standard C++ noch in der C# API vorkommen, müssen diese im SWIG-Interface behandelt werden. Neben diesen projektspezifischen Datentypen müssen auch andere durch Anweisungen angepasst werden, sodass sie später in C# verwendbar sind. In drei Interface Files sind mehrere sogenannte Typemaps und Mappingvorschriften definiert (allgemein kurz erklärt im Abschnitt »Typemaps« in diesem Paragraph 3.2.3). *POD_Mapping.i* enthält eine Typemap für die Übersetzung von besonderen Datentypen (siehe

[10] `https://docs.python.org/3/extending/extending.html`

Abschnitt »Besondere Datentypen« in diesem Abschnitt 3.2.3). Das *CppAPi.i* ist gemischt, hier werden kleine Typemaps definiert, sowie verschiedene Defines, die nur über wenige Zeilen gehen und kein einzelnes Interface File benötigen. Im File *uniplug_blender_api_swig.i* werden die Typemaps für die projektspezifischen Klassen der Header-Datei angewendet.

Typemaps SWIG selbst enthält bereits Code zum Übersetzen von Datentypen von einer Programmiersprache in eine andere Programmiersprache. Jedoch kann es vorkommen, dass sich Datentypen anders verhalten sollen oder dass, wie in diesem Fall, eigene Datentypen vorliegen, für die SWIG keine Übersetzungsvorschriften bereithält. Hierzu können unter anderem so genannte eigene Typemaps erstellt werden, welche die Regeln verändern oder ganz neu definieren. Verschiedene Arten von Typemaps behandeln dabei unterschiedliche Szenarien, so konvertiert *%typemap(in)* beispielsweise einen Wert aus der Zielsprache nach C. *%typemap(out)* geht dagegen in die andere Richtung[11] . Da bei diesem Projekt viele Klassen automatisch gewrappt werden müssen, durften die Typemaps keine Einzelfälle beschreiben und mussten allgemein für jeden Fall einsetzbar sein. Trotzdem soll an dieser Stelle vor Updates im 3D-Grafikprogramm gewarnt werden: Zwar sollten neue Funktionen durch die Header-Datei automatisch aufgenommen und beim nochmaligen Swiggen übersetzt werden, doch kann es vorkommen, das komplett neue Datentypen noch nicht durch bestehende Typemaps erfasst werden. Diese müssen dann neu bedacht und angelegt werden. Im Folgenden werden einige der für das Projekt relevanten Datentypen näher beleuchtet.

Arrays SWIG mappt Arrays von C++ nach C# standardmäßig als Pointer, so dass auf der C# -Seite dann nicht einfach mit einem klassischen C# -Array gearbeitet werden kann, sondern mit einem *SWIGTYPE_p_int* oder *SWIGTYPE_p_double*. Mit dem vordefinierten Interface *arrays_csharp.i* gibt es nur die Möglichkeit, Arrays namentlich mit dem Befehl *%apply* in ein C# -Array zu mappen. Das ist für ein Projekt dieser Größe aber nicht passend, da es extrem aufwendig ist, die Namen aller Arrays herauszusuchen und in das Interface einzutragen, da dies nicht automatisierbar ist. Deshalb werden für Integer-, Float- und Boolean-Arrays Angaben zum Typemapping gemacht, sodass sie,

[11] Weitere Informationen zu Typemaps: `http://www.swig.org/Doc1.3/Typemaps.html`

unabhängig von Größe und Namen, zu einem C# -Array und umgekehrt umgewandelt werden (siehe Listing 6). Aktuell sind jedoch keine Arrays mehr in der Header-Datei enthalten, da sie alle durch den jeweilig entsprechenden Datentyp (VFLOAT3, VFLOAT4, VINT3, ...) abgedeckt werden.

```
1 %typemap(cstype, out="$csclassname") int[ANY] "int[]"
2 %typemap(csin) int[ANY] " $csinput"
3 %typemap(imtype) int[ANY] "int[]"
```

Listing 6: Beispiel Arraymapping

std::map In der Header-Datei treten viele Key-Value Paare vom Typ *std::map<string, int>*, *std::map<int, string>* oder *std::map<string, DATA_TYPE*[12]*>* auf. Um dieses Konstrukt in ein äquivalentes in C# zu konvertieren, wird das *std_map.i* Standard-Interface in das Interface eingebunden. Das ermöglicht, diese Key-Value Paare in einen Datentyp zu übersetzen, der die selben Eigenschaften wie ein C# Dictionary besitzt. Der Name des Datentyps wird im Interface definiert und ist dann in C# unter diesem Namen zu finden. Der so definierte Datentyp funktioniert wie ein Dictionary (siehe Listing 7, Zeile 3). Um ein Key-Value Paar, das einen beliebigen Datentyp enthält, nach C# mappen zu können, muss für jeden Datentyp ein eigenes Template erstellt werden. Um hier zumindest ein wenig Copy-Paste-Arbeit zu sparen, wurde per *%define* ein Template definiert (siehe Listing 7, Zeile 5), das einen Datentyp entgegennimmt, der dann zu einem Key-Value Paar zusammengefügt wird. Dieses Template muss dann für jeden Datentyp einmal aufgerufen werden (siehe Listing 7, Zeile 9).

[12]Beliebiger Datentyp

```
%include "std_map.i"

%template(String_Int_Map)  std::map<std::string, int>;

%define %std_templates(DATA_TYPE)
%template(String_ ## DATA_TYPE ## _Map) std::map<std::string,
    UniplugBL::DATA_TYPE>;
%enddef

%std_templates(Property);
%std_templates(Function);
%std_templates(EnumPropertyItem);
```

Listing 7: std::map

std::vector In der Header-Datei treten drei Fälle des *std::vector<T>* auf. Diese können mit wenig Aufwand unter Verwendung des *std_vector.i* Interfaces einfach gemappt werden (siehe Listing 8)

```
%include "std_vector.i"

%template(IntVector) std::vector<int>;
%template(FloatVector) std::vector<float>;
%template(DoubleVector) std::vector<double>;
```

Listing 8: std::vector

ignore-Regex Einige Funktionen der Header-Datei müssen ignoriert werden. Dafür stellt SWIG die sogenannte *%ignore* Anweisung zur Verfügung, mit der man jede beliebige Funktion oder ganze Klassen ignorieren kann, das heißt, sie werden nicht in den Wrapper übernommen. Die Syntax lautet: Zuerst wird die Ignore-Anweisung geschrieben und darauf folgend der Name der zu ignorierenden Funktion. Beispielsweise *%ignore examplefunction;*. Die Funktion *examplefunction* würde im Wrapper dann nicht existieren. Hier ist der Fall jedoch etwas speziell, da nicht eine bestimmte Funktion ignoriert werden soll, sondern alle Funktionen, die ein bestimmtes Namensschema aufweisen. Um das zu erreichen, wird Patternmatching mit Regulären Ausdrücken angewandt. Für die Regulären Ausdrücke wird die Perl-Syntax (PCRE) verwendet. Um ein Pattern zu erkennen und zu ignorieren, weicht die Ignore-Syntax aber von der

Obengenannten ab. Hier muss mit der *%rename*-Anweisung, einem Regulären Ausdruck und dem Zusatz eines *$ignore*-Value gearbeitet werden, da *%ignore* eigentlich nur ein Sonderfall der *%rename* Anweisung ist. Listing 9 zeigt wie alle Funktionen, die entweder *sting_to* oder *to_string* enthalten, ignoriert werden.

```
1 %rename("%(regex:/(\w*)string_to(\w*)/$ignore/)s") "";
2 %rename("%(regex:/(\w*)to_string(\w*)/$ignore/)s") "";
```

Listing 9: inoreregex

Besondere Datentypen Die Header-Datei enthält Datentypen, die Arrays, Vektoren oder Matrizen im 3-dimensionalen Raum repräsentieren, die auf die entsprechenden Datentypen in FUSEE gemappt werden sollen. Dafür wurden verschiedene Datentypen ausprobiert. Der erste Versuch einfache Arrays zu verwenden, wurde wieder verworfen, da diese nur als Pointer zurückgegeben werden können und dies, wie in Kapitel 3.1 beschrieben, vermieden werden sollte. Darauf hin wurden diese Datentypen in der Header-Datei auf den Datentyp *std::array<>* geändert. Leider gab es auch in diesem Fall Probleme den Datentyp zu mappen, da seine Eigenschaften nicht denen eines POD entspricht und es so Probleme mit Übergabeparametern dieses Typs gibt, da bei Exceptions dieser Typ nicht auf NULL bzw. 0 gecastet werden kann. Deshalb wurden in der Header-Datei Datentypen definiert, die für sämtliche int-, bool- und float-Arrays verwendet werden. Für diese Art von Datentypen wurden Typemaps geschrieben und auf das entsprechende Äquivalent in Fusee gemappt. Die Typemaps wurden mit der *#define*-Anweisung definiert, sodass nicht jeder Datentyp eine eigene Typemap benötigt, sondern diese ähnlich wie bei *std::map<>* teilweise automatisch übersetzt werden und ein Teil an Tipparbeit und Codezeilen gespart werden kann. Das *out*-Typemap (siehe Listing 10) weist eine Besonderheit auf, die es theoretisch ermöglichen würde, auch nicht PODs zu übersetzen, da im Falle einer Exception nicht NULL bzw. 0 zurückgegeben wird, sondern ein neu erzeugtes leeres Objekt des Datentyps. Das wird mit folgender SWIG-Anweisung erreicht »*%typemap(out, null="TYPE_NAME()") TYPE_NAME*« (vgl. Listing 10). Diese Lösung wurde jedoch erst sehr spät gefunden, sodass letztendlich bei den neu definierten Datentypen das VFLOAT Prinzip behalten wurde, da so auch alle auftretenden Arrays abgedeckt werden und auf das oben beschriebe Arraymapping verzichtet werden kann.

```
%define %fusee_pod_typemaps(DATA_TYPE, TYPE_NAME, RESULT)
    %ignore "operator []";
    %ignore TYPE_NAME;
    %typemap(cstype, out="RESULT") TYPE_NAME, TYPE_NAME *
        "RESULT"
        %typemap(imtype, out="RESULT") TYPE_NAME, TYPE_NAME *
        "RESULT /* imtype */"
    %typemap(ctype, out="TYPE_NAME") TYPE_NAME, TYPE_NAME *
        "TYPE_NAME /* ctype */"
        ...
    %typemap(out, null="TYPE_NAME() /* out (null) */")
   TYPE_NAME
    %{
        $result = $1;
    %}
    %typemap(in) TYPE_NAME
    %{
        $1 = *((TYPE_NAME *)&($input));
    %}
    %typemap(csin) TYPE_NAME, TYPE_NAME *
        "$csinput"
    %typemap(csdirectorin, pre="") TYPE_NAME, TYPE_NAME *
        "$iminput"
    %typemap(csdirectorout) TYPE_NAME, TYPE_NAME *
        "$cscall"
    %typemap(csvarin) TYPE_NAME, TYPE_NAME *
    %{
            ...
    %}
    %typemap(csvarout) TYPE_NAME, TYPE_NAME *
    %{
            ...
    %}
%enddef

%include "POD_Mapping.i"
%fusee_pod_typemaps(float, UniplugBL::VFLOAT3, Fusee.Math.
   float3);
```

Listing 10: Stark vereinfachtes Beispiel des Typemaps für VFLOAT

3.3 Logische Übersetzung

Alexander Scheurer

Neben der rein programmatischen Übersetzung der Blender API nach C# muss auch für Uniplug eine eigene API-Struktur geschaffen werden, gegen die zukünftige Benutzer von Uniplug bzw. Plugin-Entwickler programmieren sollen. Diese Struktur muss allgemein gehalten werden, damit sie auch die Anbindung weiterer 3D-Softwareprodukte erlaubt. Da die Forschungsgruppe im technischen Bereich eine vollständige Übersetzung der Blender-API anstrebt, muss auch hier eine ebenso vollständige Lösung gefunden werden. Eine vollständige Lösung bedeutet, dass nicht nur der Umgang mit 3D-Modellen erreicht werden soll, sondern auch der mit einzelnen Dreiecken, Materialien, Animationen, GUI-Elementen, sowie Lichtern und Kameras.

Da Austauschformate, die zwischen verschiedenen 3D-Softwareprodukten operieren, genau dieses Problem bereits gelöst haben, folgt nun eine Betrachtung eben jener. Einfache Formate wie *Wavefront OBJ* sind hierfür nicht ausreichend, da dieses unter anderem keine Animationen unterstützt.

Eines der neusten Austauschformate ist *Open Game Engine Exchange* (OpenGEX)[13], das im Dezember 2013 von Eric Lengyel veröffentlicht wurde. Der Aufbau von OpenGEX beinhaltet alle relevanten Konstrukte mit Ausnahme von GUI-Elementen, wobei hier die des jeweiligen Plugins in der entsprechenden 3D-Software gemeint sind, somit ist es nicht verwunderlich, dass ein Austauschformat diese nicht vorsieht.

[13] `http://opengex.org/`

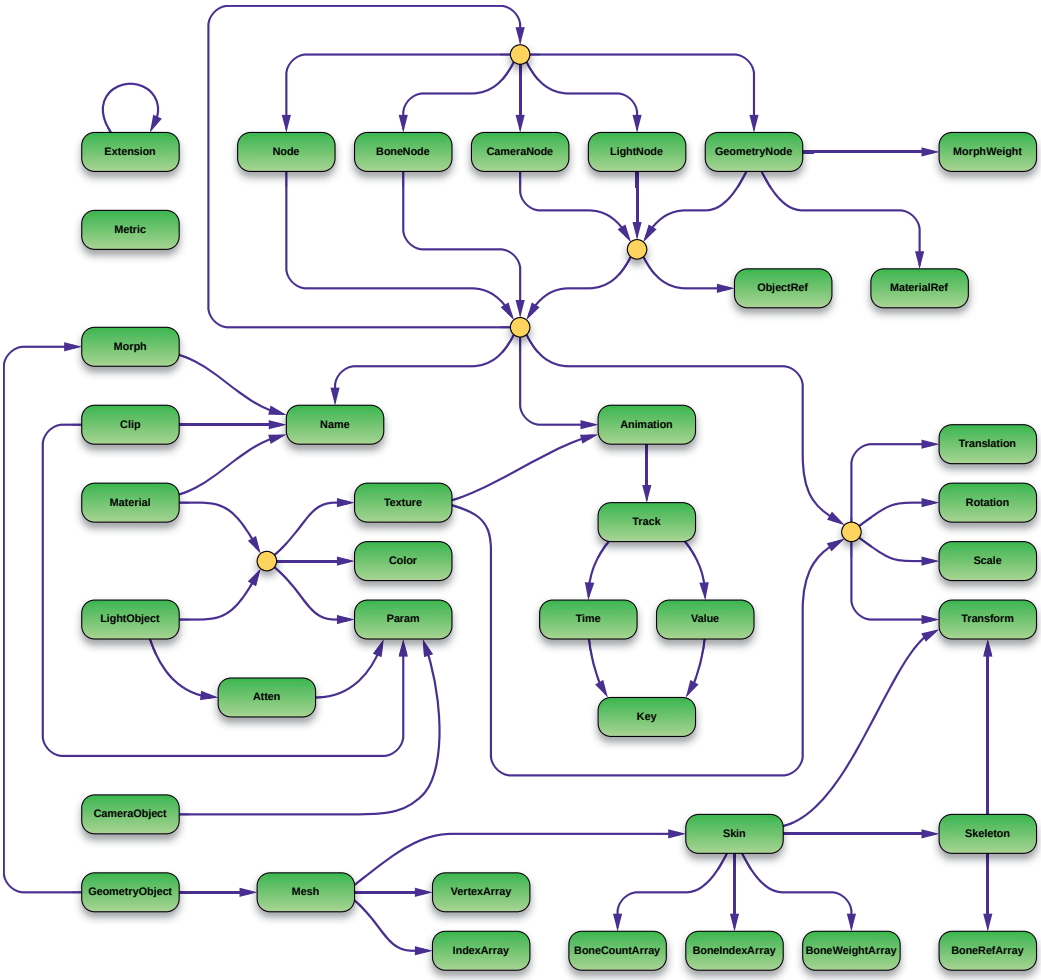


Abbildung 2: OpenGEX Struktur

4 Fazit und Ausblick

Alexander Scheurer

Die Ergebnisse des Forschungsprojektes zeigen, dass es auf dem gezeigten Weg und mit den technischen Gegebenheiten prinzipiell möglich ist, eine Kommunikation zwischen Blender und C# zu schaffen. Allerdings hat sich die Einschätzung der in den Vorüberlegungen getroffenen Entscheidungen sehr stark verändert.

Der Implementierungs- und Wartungsaufwand für mehrere 3D-Softwareprodukte ist sehr viel höher als erwartet, was in den zuvor dargelegten Problemen mit SWIG (3.2) und allgemein am Umfang der Funktionen der Softwareprodukte begründet liegt.

Die Forschungsgruppe empfiehlt für weitergehende Versuche der Entwicklung eines Uniplugs, die zu übersetzenden Funktionen weitergehend zu evaluieren und nativ für die jeweilige 3D-Software ein Plugin als Zwischenschicht für die Übersetzung zu entwickeln.

4.1 Aussicht

Fabian Gärtner

Aktuell werden die in Abschnitt 3.1.2.4 beschriebenen Kommentare noch nicht nach C# übernommen. Solange das eigentliche Uniplug, das die Schnittstellen verschiedener APIs vereinheitlichen soll, in Entwicklung ist, sollte daher automatisiert eine Doku erstellt werden, die die Entwicklung eigener Plugins vereinfacht. Ebenso wäre dies für Entwickler hilfreich, die, abseits des Uniplugs, die durch dieses Projekt entstandene C++ -API für Blender nutzen möchten.

Bisher muss das *BlenderPlug*-Plugin des Weiteren noch manuell initialisiert werden. Es ist aber möglich, über ein direkt in Python geschriebenes Plugin, das unter dem Namen *__init__.py* ebensfalls in den Unterordner Fusee im Blender-Verzeichnis gelegt wird, das Plugin entweder automatisch auszuführen oder Kontextmenüeintäge anzulegen. Über diese kann die im Plugin implementierte Funktionalität ausgeführt werden. Mehr Informationen hierzu finden sich in der Blender-Doku[14]. Je nach Aufbau könnten so mehrere Plugins im Uniplug

[14] `https://www.blender.org/api/blender_python_api_2_75_3/info_tutorial_addon.html`

gebündelt werden (Exportfunktionalität, Szenenfunktionalität, etc.) oder ein Event-System (Aufruf einer speziellen Funktion beim Start vom Blender, Aufruf einer Funktion beim Beenden von Blender, etc.) aufgebaut werden. Dies sollte in Zukunft ebenso weiter verfolgt werden wie die Entwicklung von Beispielplugins und insbesondere eines zur C4D-Variante des Uniplugs vergleichbaren Exportplugins, das es erlaubt, die Szene als FUSEE-Web-Anwendung zu exportieren und so im Web anzuschauen.

Ein Problem ist auch, dass alle DLL-Dateien (*CsWrapper.dll, CsClient.dll*, etc.), die für die korrekte Ausführung der in C# implementierten Plugin-Funktionalität benötigt werden, im Hauptverzeichnis von Blender liegen müssen, da sie sonst nicht gefunden werden. Hier müsste der Assembly-Suchpfad durch entsprechende Programmierung oder Konfiguration angepasst werden. Dadurch wäre es ausreichend, dass alle Dateien im Unterordner Fusee im Addon-Unterordner des Hauptverzeichnisses liegen.

Des Weiteren sollte das System an das System angepasst werden, das aktuell bereits beim Uniplug für Cinema4D zum Einsatz kommt. Hier wird bspw. ein Ordner nach C# -Plugins durchsucht und diese per Reflection ausgeführt, sodass nicht nur ein einziges C# -Plugin verwendet werden kann. Ebenso sollte die Nutzung auf anderen Betriebssystemen vorbereitet werden, indem, wie auch beim C4D-Uniplug, Mono als alternative .NET-Runtime ermöglicht wird. Das automatische Aufrufen von .NET-Funktionen aus einer C++ -DLL heraus, wie es hier von BlenderPlug nach ManagedBridge gemacht wird, ist aktuell nur unter Windows möglich.

In C# ist es möglich für Properties sogenannte Auto-Setter und -Getter zu verwenden, sodass Anweisungen wie *pyUniplug.Context().Scene()...* zu *pyUniplug.Context.Scene...* vereinfacht werden können. Dies ist zwar hauptsächlich eine Aufgabe von SWIG, möglicherweise sind aber bereits Vorbereitungen in der C++ -Header-Datei notwendig, sodass SWIG solche Auto-Setter und -Getter erstellen kann. Denkbar wäre bspw. das Anlegen von (in C++) ungenutzten »Dummy«-Feldern innerhalb der Klassen, sodass bei der Verarbeitung durch SWIG diese als Properties und nicht als Funktionen erkannt und um den Aufruf der eigentlichen Setter- und Getter-Funktionen (deren Access-Modifier im gleichen Schritt auf *private* geändert werden können) erweitert wird.

Zwangsläufig muss durch die längere Prozesskette von C# über SWIG und C++ nach Python ein Performanceverlust in Kauf genommen werden. Der Flaschenhals ist dabei aber weniger im Bereich SWIG zu erwarten, da Aufrufe von C# in exportiere C/C++ -Funktionen letztlich nur wenig Mehraufwand bedeuten,

sondern im speziellen durch den Aufruf von Python-Funktionen in der Header-Datei. Hier wird zusätzlich zu den Konvertierungen der PyObject-Objekte in Basistypen und umgekehrt auch immer wieder mit String-Operationen gearbeitet. Zudem ist insbesondere beim allerersten Initialisierungsaufruf des Uniplugs eine merkliche Verzögerung von 1 - 2 Sekunden messbar, da hier zunächst im Hintergrund die .NET-Runtime initialisiert werden muss. Weitere Aufrufe der in C# implementierten Uniplug-Funktionen sind dann aber zumindest nicht merklich verzögert. Dennoch sollten hier einige präzise Messungen vorgenommen werden, um allgemeine Aussagen über den Performanceverlust zu treffen und so herauszufinden, ob diese im Alltag relevant sind. Es ist in der Regel zwar nicht davon auszugehen, dass mit den mit FUSEE/Uniplug geschriebenen Plugins zeitkritische Aufgaben bearbeitet werden müssen, denkbar wäre es aber dennoch. Auf Basis der sich dadurch ergebenden Erkenntnisse, wären eventuell weitere Optimierungen (bspw. das Cachen der Ergebnisse von Python-Funktionsaufrufen, etc.) möglich.

How to create your very own FUSEE plugin for Blender:

> 1. Install these four programs:

   a. Microsoft VisualStudio 2013 b. Python 3.4 (x64)
   c. SWIG for Windows 3.0.6 d. Blender 2.72 (x64)

> 2. Create and set system environment variables:

   a. Open: System Properties -> Advanced system settings -> Advanced -> Environment Variables

   b. Add the path to your SWIG and Python folders to the PATH variable -> e.g.: D:\swigwin-3.0.6\;D:\Python34\;

   c. Add a new variable BLENDER_ROOT and set it to the path of your Blender version folder -> e.g.: D:\Blender Foundation\Blender\2.72\

   d. Add a new variable PYTHON_ROOT and set its value to the path of your Python folder -> e.g.: D:\Python34\

   e. Add or change the value of the VS100COMNTOOLS variable to the path of your VisualStudio 2013 folder
   -> e.g.: C:\Program Files (x86)\Microsoft Visual Studio 12.0\Common7\Tools\

> 3. Open the solution SwigBlender.sln and change the build mode to Release x64. If you don't have read and/or write permissions for your Blender folder you have to run VisualStudio as administrator.

> 4. Build projects 01_CppApi, 02_Swig, and 03_CppWrapper in that order. Use "Project only" when building.

> 5. Add all files in \SwigIt\CsWrapper\Generated to the corresponding folder in the 04_CsWrapper project in VisualStudio.

> 6. Build projects 04_CsWrapper, 05_CsClient, 06_ManagedBridge, and 07_BlenderPlug in that order. Again, use "Project only".

> 7. Set 07_BlenderPlug as the starting project. Open the project properties and go to "Debugging". Set the command property value to the path of your blender-app.exe and the working directory property value to the path of your Blender folder.
   -> e.g.: D:\Blender Foundation\Blender\blender-app.exe
   and D:\Blender Foundation\Blender\

> 8. Click on the start button. Blender should open. Then, switch to the Scripting view (see upper menu, next to "Help").

> 9. You should see a black console. Type:

```
from fusee import uniplug
uniplug.init()
```

   This executes the code written in the CsClient project.

> 10. That's it! You can now start to add your own code to the CsClient.cs. All you have to do is to rebuild the CsClient and BlenderPlug projects and start Blender again (don't forget to close Blender before building).

   If you need more than one function (i.e. more than uniplug.init()) you can modify the Main.cpp of the BlenderPlug project and/or the two files ManagedBridge.h and ManagedBridge.cpp and rebuild the corresponding projects. Please do not change the uniplug_blender_api.h unless you know what you're doing.

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.

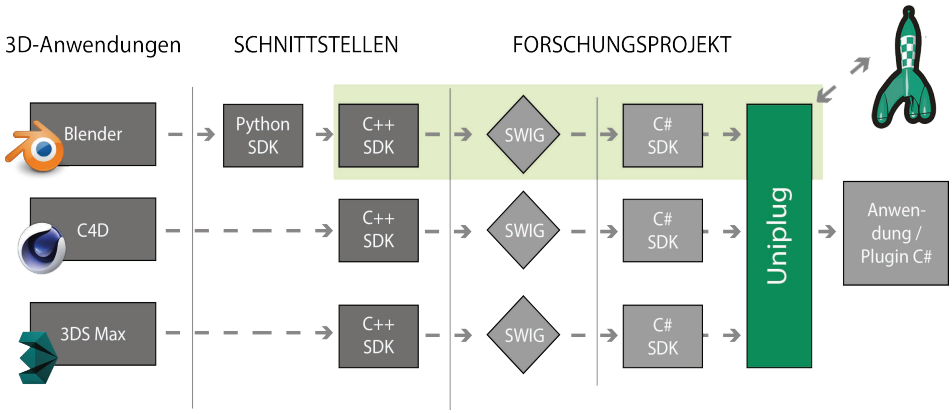
Methodik & Vorgehensweise

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