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

23. 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.

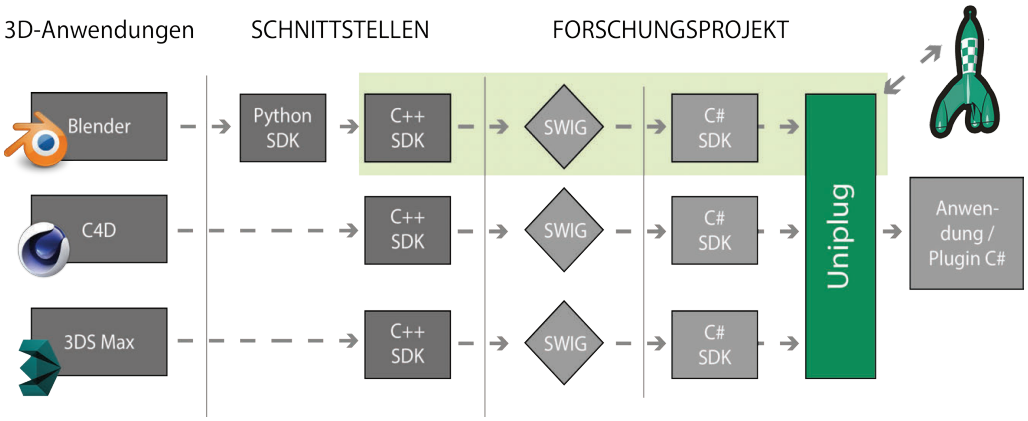


Abbildung 1: Schematische Darstellung des Aufbaus

3 Umsetzung

3.1 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 beschrieben Aufwand). Um diese Problematik zu vermeiden, wurden im Laufe des Projekts 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 war ein erster Versuch, 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[1] (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

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

der 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[2] 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[3]. 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*[4] 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*[5] eingebunden, das neben dem eben erwähnten Skript auch die benötigten Ordner und eine aktuelle Version von CMake bereitstellt.

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

Es sind daher nun nur noch wenige einfache Schritte notwendig, bevor Blender (und die Unterprojekte wie MakesRNA) gebaut werden können. Als erster Schritt muss ein Git- und ein SVN-Tool installiert werden und die Pfade zu diesen Tools müssen der Umgebungsvariable *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-Projekts fragen den aktuellen Modus ab und erzeugen so ggf. statt den vielen verschiedenen für Blender notwendigen Dateien nur noch eine einzige Header-Datei erzeugt, 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 die 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 nicht mehr geändert werden könnten nach der Initialisierung 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 Objekts in die des neuen Objekts 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 Es werden die meisten im MakesRNA-Tool definierten Klassen 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* der 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), dass die Implementierung jeglicher Getter- bzw. Setter-Funktionen auf eine Zeile reduziert (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, dass 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[6] 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[7] 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.

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

3.2 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.

3.2.1 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 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 3.2.5 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

3.2.2 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 3.2.4). 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?

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

3.2.3 Build-Einstellungen in VS

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

3.2.4 Typemaps

Was sind Typemaps und wie funktionieren sie?

3.2.5 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.

3.2.6 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

3.2.7 GC

4 Fazit und Ausblick

Trotz dem, dass es der Forschungsgruppe bisher nicht gelungen ist eine Kommunikation zwischen Blender und C#zu schaffen schätzt die Forschungsgruppe die technischen Gegebenheite so ein das es auf dem gezeigten Weg prinzipiell trotzdem möglich ist. Allerdings hat sich die Einschätzung der in den Vorüberlegungen getroffenen Entscheidungen sehr stark verändert.

Der Implementierung- und Wartungsaufwand für mehrere 3D-Softwareprodukte ist sehr viel höher als erwartet, was in den zuvor dargelegten Problemen mit SWIG (s. **SWIG-Referent**) begründet liegt. SWIG ist (noch?) nicht dazu geeignet eine Übersetzung für **X-Dröfzigzausendstel** Interfaces durchzuführen.

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

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.

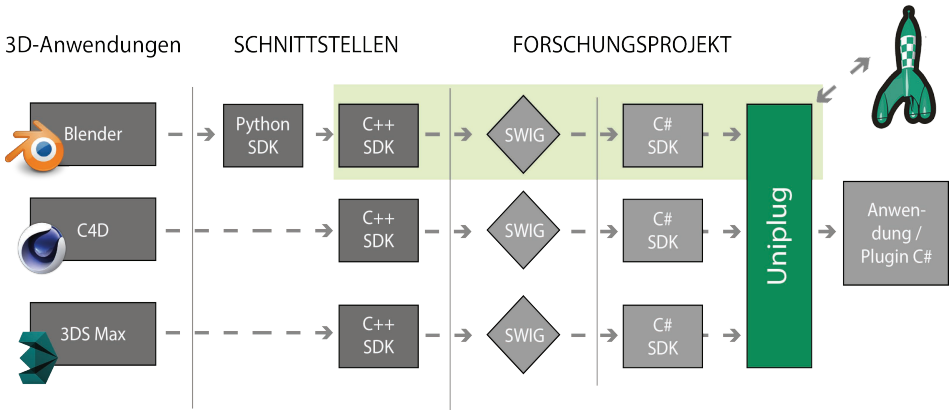
Methodik & Vorgehensweise

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