Hochschule für Technik und Wirtschaft
Berlin

Beleg-Arbeit

VetoLC - Implementation einer Entwicklungsumgebung für Live Coding mit Anbindung an Python, OpenGL und Qt

Autor
Veit Heller - s0539501

Autor
Tobias Brosge - s0539713

Dozent
Sebastian Bauer

14. August 2014

Danksagung

Die Autoren danken Ihrem Dozenten Sebastian Bauer für die Möglichkeit eine solche Semesterarbeit zu produzieren.

Veit Heller möchte sich ausserdem bei seiner Freundin Alina bedanken, die lange Monologe über Dinge, die sich nicht versteht, immer wieder erträgt und sich sogar einbringt, wo sie kann; dank auch dafür, dass sie ihr Designerherz immer wieder von hässlichen Benutzeroberflächen verletzen lässt. Ausserdem darf auch seine Mutter Marion nicht unerwähnt bleiben, die ihm zu jedem Zeitpunkt zur Seite stand; nicht nur bei dieser Arbeit.

Inhaltsverzeichnis

1 Einleitung

> If debugging is the process of removing software bugs, then programming must be the process of putting them in.
>
> Edsger W. Dijkstra

Im Folgenden wird eine Live-Coding-Applikation vorgestellt, die im Rahmen einer an der Hochschule für Technik und Wirtschaft Berlin angebotenen Lehrveranstaltung des Bachelorstudiengangs Angewandte Informatik, Vertiefung Multimedia, entwickelt wurde. Das Projektziel war frei wählbar, die Anforderung war jedoch, dass es sich um eine Multimediaanwendung handelt, die Gebrauch von der Programmbibliothek Qt[1] macht und in C++, der ursprünglichen und nativen Sprache von Qt, geschrieben ist.[2]

Die Autoren entschieden sich für eine Live-Coding-Applikation,[3] da diese eine breite Einführung in die Möglichkeiten von Qt bietet und es momentan nach ihrem Wissenstand keine anderen adäquaten Tools für dieses Subset des Programmierens gibt. Ausserdem besteht bei den Autoren ein grosses Interesse an Metaprogrammierung und dem Mechanismus der Quelltextübersetzung sowie für die Sache des Live Coding. Somit schufen sie sich auch ein Werkzeug für ihre eigenen Zwecke, das jedoch auch zum freien Download auf Github angeboten wird. An dieser Stelle besteht mit einiger Wahrscheinlichkeit bereits ein Bedarf nach Definition, der in Sektionen 2.1 und 2.2 gestillt wird.

Eine aktuelle Version des Werkzeuges kann zu jedem Zeitpunkt von der Seite `https://github.com/hellerve/VetoLC` *abgerufen werden. Ebenso kann die aktuellste Version dieser Dokumentation von der Webseite* `https://github.com/hellerve/LC-Doc` *oder auf Anfrage von den Autoren bezogen werden.*

[1] http://qt-project.org/

[2] Obwohl es inzwischen verschiedenste Wrapper und Erweiterungen gibt.

[3] Für eine Definition des Begriffs wird auf Sektion 2.1 auf Seite 5 verwiesen.

1.1 Gliederung der Arbeit

Im Folgenden wird die Gliederung der Arbeit beschrieben; sie spiegelt den ungefähren Entwicklungsprozess wider:

Abschnitt 2 beschäftigt sich mit den grundlegenden Begrifflichkeiten der vorliegenden Arbeit und gibt eine kurze, nicht umfassende Einführung[4] in Qt als Bibliothek.

Abschnitt 3 behandelt die Zielsetzung des Projektes. Sie behandelt die Planung sowie die implementationsunabhängigen Vorüberlegungen der Autoren. Es wird ein Überblick über die Infrastruktur und ihr zugrundeliegende Design-Entscheidungen gegeben.

Abschnitt 4 beschreibt die Implementation und geht auf technische Details ein. Der Entwicklungsprozess und die einzelnen Teile des Programms werden erläutert. Dies ist natürlich nicht in aller Tiefe möglich, jedoch ergänzt sich die vorliegende Arbeit hierin mit der Dokumentation des Programms.

Abschnitt 5 stellt Fragen, die nach der ersten Version der Entwicklungsumgebung offen bleiben, denn die Autoren planen, daran weiterzuarbeiten.

Abschnitt 6 versucht Schlussworte zu finden. Dies ist derjenige Teil der Arbeit, der den Autoren am schwersten von der Hand ging.

[4] Eine solche umfassende Einführung müsste eine eigene Arbeit füllen, länger als diese. Für eine Referenz wird auf [EE11] verwiesen.

2 Definitionen

Im Folgenden werden einige für das Verständnis des weiteren Textes elementare Definitionen bereitgestellt.

2.1 Definition des Begriffes Live Coding

Live Coding ist eine neue und relativ interessante Disziplin der Informatik, die dem künstlerischen Potential der Programmierung von Multimedia-Anwendungen noch die Kraft der Performance-Kunst, die Kraft des Moments, eröffnet. Durch entsprechende Werkzeuge soll das Schreiben von Anwendungen in Echtzeit zur Kunstform erhoben werden. Dejaying[5] ist genauso vom Begriff eingeschlossen wie Grafikprogrammierung, wobei diese vor allem von Programmen wie Processing,[6] einer Entwicklungsumgebung für Java, dominiert wird. In der Klangprogrammierung gibt es keine solch dominante Kraft, die verschiedensten Sprachen und Umgebungen werden benutzt, von Tidal[7], einer in Haskell implementierten funktionalen Nischensprache, bis hin zu Super Collider,[8], einer mächtigen objektorientierten Sprache. Wir hoffen dass wir uns durch die Auswahl Pythons, einer objektorientierten Sprache, die jedoch auch viele funktionale Elemente enthält(`map()`, `filter()`, `lambda`, Generatoren, Closures) ein möglichst gutes Fundament erwählt haben, um beide Ansätze zumindest zu emulieren.

Durch die Erschaffung einer hinreichend komfortablen, mächtigen Entwicklungsumgebung mit einem breiten Spektrum an Möglichkeiten erhoffen sich die Autoren ein erhöhtes Interesse an der Sache des Live Coding sowie erhöhte Produktivität der kreativ involvierten Entwickler und Künstler; eine solche Plattform sollte aus ihrem Verständnis heraus von Grund auf neu geschrieben werden, da die Umformung bestehender Umgebungen meist ungewollte Kompromisse mit sich zieht, die das Projekt entweder stagnieren lassen oder in ein anderes, ungewolltes Produkt umwandeln würden.

[5] hier: das Erstellen von audiomanipulativen Programmen in Echtzeit, seit kurzem bekannt als *Algorave*.
[6] http://www.processing.org
[7] http://toplap.org/tidal/
[8] http://supercollider.sourceforge.net/

2.2 Einführung in die Programmbibliothek Qt

> More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason – including blind stupidity.
>
> William A. Wulf

Qt(ausgesprochen *kju:t*[9], wie das englische Wort *cute*) ist eine im Jahr 1991 von zwei schwedischen Entwicklern konzipierte Programmbibliothek, die plattformübergreifende Programmierung von grafischen Oberflächen, Netzwerk- und Datenbankprogrammen und Spielen ermöglicht, sequentiell und nebenläufig. Sie basiert auf dem Verständnis von Ereignissen, Signalen und Slots sowie den Erweiterungen von C++, die durch einen zusätzlichen Präprozessor, den Meta Object Compiler(moc), ermöglicht werden.

Klassen, die diese Funktionen nutzen wollen, müssen das Schlüsselwort `Q_OBJECT` einbinden und können zusätzlich zu den `private` und `public`-Teilen der Klassendefinition noch `(private/public) slot` und `signal` definieren.

2.2.1 Ereignisse in Qt

Ereignisse werden in einer von einer Qt-Applikation(`QApplication`) begonnenen Ereignisschleife(Event Loop) verarbeitet. Passiert nichts, läuft die Schleife endlos weiter.[10] Sie wird durch ein Schliessereignis(Close Event) unterbrochen. Auch das Erzeugen neuer asynchron laufender Schleifen ist möglich(Dazu müssen bestimmte Objekte - `QThread`, `QProcess`, `QThreadpool` etc. - erstellt und die laufenden Prozesse dorthin verschoben werden).

2.2.2 Die Ereignisschleife

Wenn ein Programmierer die Ereignisschleife nach den Standardvorgaben startet, ruft er `QApplication::exec()` auf. Diese leitet ihn dann durch zwei `exec()`-Definitionen der Basisklassen zu `QtCoreApplication`, die testet, ob das Aufrufen der Schleife valide ist und dann eine Instanz der internen Klasse `QEventLoop` erstellt. Von dieser wird wiederum `exec()` aufgerufen.[11] Diese Methode ruft `processEvents()` auf, die dann wartet, bis etwas passiert. Also doch ein Fall von Busy Waiting? Nein, die Methode bricht die Operation ab, wenn nach zu langer Zeit nichts passiert. Natürlich wird die Applikation dann nicht einfach abgebrochen, sondern eine weitere Schleifenkondition inder Methode darüber getestet, doch schlägt diese fehl(ein internes atomares Flag namens exit, das zeigt, dass, nun ja, ein Beenden des Programmes nötig wurde), bricht die Ereignisschleife ab. Der beinhaltende Thread wird dann normalerweise beendet(außer man überschreibt gewisse Methoden in QThread, aber außer den Autoren dieser Arbeit braucht das heutzutage fast niemand mehr).

[9] Die Kryptik der IPA-Zeichen begeistert zumindest einen der Autoren zutiefst.

[10] Vereinfacht ausgedrückt könnte man von Busy Waiting sprechen, jedoch ist die interne Struktur der Event Loop sehr viel komplexer, siehe Sektion 2.2.2.

[11] Zur Erinnerung: Dies ist die vierte Klasse, die diese Methode implementiert!

2.2.3 Signale

Signale sind auf Ereignissen basierende Nachrichten, die von Objekten an die Ereignisschleife gereicht werden und von dort an die entsprechenden Slots versendet(durch Broadcasting an alle verbundenen Slots) werden.

2.2.4 Slots

Slots sind Routinen, die an bestimmte Signale gebunden werden können und müssen, um auf sie zu reagieren. Der Absender muss zur Definitionszeit bekannt sein, das heisst, es ist nich möglich, einfach Signale von allen oder unbekannten Objekten entgegenzunehmen.

Slots können als `private` oderm `public` deklariert sein und damit entweder interne oder externe Sichtbarkeit besitzen.

2.2.5 Der Meta Object Compiler(moc)

Der Meta Object Compiler übersetzt den nicht-C++-konformen Qt-Code in C++-Code, indem er alle Makros und die Definition von Signalen und Slots durch C++-Code ersetzt. Dieser (heute unötig) komplexe Vorgang ist der Historie der Bibliothek geschuldet, da diese in einer Zeit entstand, in der Kompilatoren die generische Programmierung durch Vorlagen(Templates) nicht gleich gut unterstützen. Dies wiederum stand in krassem Gegensatz zum plattformunabhängigen Ansatz von Qt, was bedeutete, dass man ohne dieses Konzept auskommen musste.

2.3 Einführung in OpenGL

OpenGL ist eine Spezifikation für Grafikprogrammierung mit einem ähnlichen Ziel wi Qt: Plattformunabhängigkeit. Auch soll es programmiersprachenunabhängig sein, jedoch wird die meiste Arbeit mit und an OpenGL in C++ verrichtet. Die API ist in Programmierbibliotheken sowie in Grafiktreibern implementiert und ansteuerbar.

Aus Gründen der Kürze verzichten die Autoren an dieser Stelle auf weitere, nicht unbedingt notwendige Erklärungen der OpenGL-Interna und verweisen stattdessen auf [HLSW10]. Nur noch die GLSL(OpenGL Shading Language) verdient kurze Erwähnung, da sie dem Benutzer als Skriptsprache zur Verfügung gestellt wird.

2.3.1 GLSL

GLSL ist eine im Jahr 2002 von der Khronos Group vorgestellte Programmiersprache zum Programmieren von Shadern(also Texturprogrammen) auf dem Grafikprozessor. Die Syntax ist C-ähnlich, jedoch um einige Datentypen(Vektoren, Matrizen) und Standard-Funktionen erweitert und ohne Zeiger. Die für VetoLC ausgewählten Shader sind sogenannte Fragment-Shader, der die Farbe für Fragmente berechnet. Er operiert also nicht auf Vertices, das heisst er hat eine

zweidimensionale Sicht auf das zu bearbeitende Fragment.[12] Der Quelltext wird vom Grafikkarten-Treiber zu ausführbarem Code kompiliert.

[12] wobei sich, wie in den mitgelieferten Beispieldateien gezeigt, die Illusion der Dreidimensionalität relativ leicht erzeugen lässt.

3 Zielsetzung

> Walking on water and developing software from a specification are easy if both are frozen.
>
> Edward V. Berard

Eine Entwicklungsumgebung zu erschaffen, die den Ansprüchen einer Entwicklergemeinde gerecht wird, die auf hohe Verfügbarkeit und Robustheit bei gleichzeitiger Garantie der Evaluation in Echtzeit aufbaut, ist eine Aufgabe, der sich die Autoren nicht gewachsen fühlen. Kompensiert wird dies durch die offene Lizensierung, die den Benutzer dazu aufruft, selbst an dem Programm zu arbeiten. Trotzdem sind, um den Grundstein einer solchen Entwicklung zu legen, einige Vorüberlegungen nötig. Die Infrastruktur des Werkzeugs lässt sich im Nachhinein nur unter grossen Schwierigkeiten wieder ändern und meist schreibt ein engagierter Entwickler dann lieber ein neues, besseres Programm.[13]

3.1 Allgemeine Infrastruktur

War der erste Prototyp des Editors noch relativ monolithisch, wurde mit seiner Erweiterung und der Verbesserung der Funktionalität klar, dass dies keine zukunftsträchtige oder elegante Lösung war. Dazu sei auch noch gesagt, dass die erste Skizze der Infrastruktur bereits modular war, jedoch waren diese guten und wohlgeformten Ideen in der ersten Euphorie vergessen. Die letztendlich verwendete Infrastruktur sieht jener Skizze doch wieder ähnlich; ein Beweis dafür, dass Ideen meist besser sind als ihre erste Umsetzung in die Tat dies vermuten lässt.

Auf Abbildung 3.1(Seite 10) ist die grundlegende Infrastruktur des Programmes dargestellt. Alle dem Benutzer bekannten Instanzen sind weiss hinterlegt, alle ihm nicht notwendigerweise erfassbaren in einem hellem Blau. Der Kontrollfluss ist relativ klar: Der Benutzer interagiert nur mit dem Editor, dieser mit den Einstellungen, um auf diese zu reagieren und dem Backend, um Informationen über Eingaben des Benutzers weiterzugeben und auf Nachrichten der geöffneten Kontexte und des Systems zu reagieren. Das Backend wiederum ist ebenfalls lesend mit den Einstellungen verbunden. An dieser Stelle muss eine erste Anmerkung gemacht werden: die Instanz Einstellungen ist komplett ideell, das heisst es gibt *keine* ihr entsprechende Implementation.

Weiterhin ist das Backend mit dem den Code ausführendem Kontext verbunden. Diese sind nicht identisch, der Kontext ist eine asynchrone Instanz, die nach Bedarf gestartet und gestoppt wird(*d.h.* ein Thread). Dies erlaubt zum einen die weitere Benutzung des Editors, auch wenn gerade ein Programm ausgeführt wird und die Kapselung und Abstraktion. Die Steuerung des Kontextes erfolgt durch Signale durch das Backend.

In der Realität ist die Infrastruktur sehr viel komplexer und verzweigter, doch die grundlegenden Komponenten sind diejenigen, die in Abbildung 3.1 zu sehen

[13] So entstand zumindest die Idee der Autoren zu ihrem Editor.

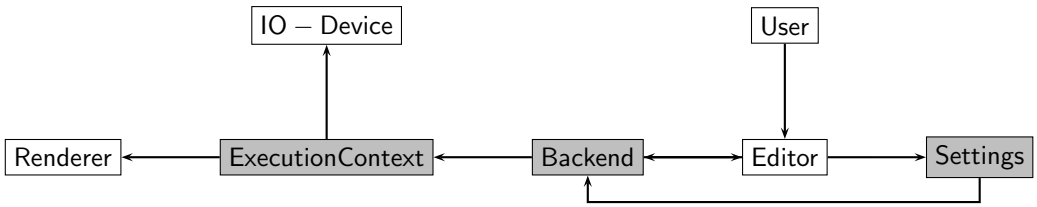

Abbildung 1: Eine Übersicht über die Infrastruktur des Editors.

sind; meist sind die eigentlich zusammenhängenden Ideen noch in Unterprobleme aufgespalten und gekapselt.

3.2 Entscheidungen im Vorfeld und ihre Begründung

Über Modularität und Kapselung lässt sich streiten, in vielen Fällen gab es durchaus zweierlei Wege, die Infrastruktur anzulegen, doch war eine Entscheidung vonnöten. Trotzdem fühlen die Autoren sich verpflichtet, Auskunft über die Gedanken zu geben, die hinter dem Aufbau stecken.

So stellt sich die Frage, ob die Einstellungen tatsächlich vom Backend zu trennen sind. In der Ansicht der Autoren sind sie das, da sie Anweisungen des Benutzers sind, die zwar Anweisungen des Backends an den Editor entsprechen, jedoch nicht wirklich aus dem Arbeitsgang des Programmes stammen, sondern quasi in dieses injeziert werden. Desweiteren ist eine so einfache Kapselung kostengünstig und für die Wartbarkeit von Bedeutung. So wird dem aufmerksamen Leser des Quelltextes aufgefallen sein, dass neben dem Backend auch ein SettingsBackend existiert, das mit dem Backend in Verbindung steht. Dies ist eine sehr einfache, der Lesbarkeit jedoch zuträgliche Trennung und nur einer von einigen Punkten, in dem der kleine Überblick der Realität nicht ganz getreu wird.

Ebenso ist der ausführende Kontext(im Überblick englisch betitelt), eigentlich ein Teil des Backends, jedoch läuft dieser asynchron und unabhängig vom Backend und steht mit diesem nur über `Signals` in Verbindung. Dies ist infrastrukturell geschickter, da es eine vom Hauptprogramm unabhängige Fehlerbehandlung im Kontext ermöglicht und diesem zusätzlich mehr CPU-Zeit zur Verfügung stellen sollte.[14]

Abschliessend möchten die Autoren noch zugeben, dass sämtliche Infrastruktur in einem Versuch-und-Fehler-Arbeitsprozess entstand; dies lässt sich leicht anhand der Commit-Beschriftungen(und deren Inhalt) im Versionsrevisionswerkzeug(git) ablesen. Von einer Studie derselben raten die Verantwortlichen jedoch heftigst ab, da sich ein Moloch der schlechtmöglichsten Praktiken abzeichnet(so wurde zum Beispiel über den ganzen Erarbeitungsprozess nur eine einzige Branch bearbeitet).

[14]Dies ist eine blosse Vermutung, ungetestet und höchstwahrscheinlich falsch.

4 Implementation

Die Beschreibung der Implementation ist nicht ganz einfach und es lohnt sich, die mithilfe von Doxygen erstellte Klassendokumentation zu Rate zu ziehen. Glücklicherweise sind sämtliche Komponenten jedoch mehr oder weniger aussagekräftig dokumentiert, sodass ein geneigter Leser sich schnell zurecht finden sollte. Auch findet sich im Programm selbst eine Art Handbuch, dass unter der Überschrift `Über VetoLC` lokalisiert werden kann. Die Besprechung der Implementation erfolgt bestandteilweise, sodass die konzeptuellen Teile nicht notwendigerweise in Abhängigkeit von Klassendefinitionen betrachtet werden.

4.1 Editor(GUI)

Die Oberfläche ist möglichst einfach gestaltet. Neben Task-, Tool- und Statusbar gibt es lediglich noch ein fast den ganzen Editor ausfüllendes Eingabefenster mit Zeilenangabe und Syntax-Hervorhebung. Die Programmierung dieses Elements stellte den eher gestalterischen Teil der Arbeit dar, da es hier viel um Benutzbarkeit und Intuitivität der Oberfläche ging. Das ist wohl auch der Teil, der den Autoren am schlechtesten gelang, da deren kreativen Fähigkeiten in diesem Bereich eher zurückhaltend ausgeprägt sind. Jedoch gelang es schliesslich, ein einigermassen brauchbares, einfaches Design zu erarbeiten, das sich auf allen unterstützen Plattformen recht ähnlich ausmacht.[15]

4.2 Backend

Das Backend kommuniziert indirekt mit den Editor-Instanzen über eine virtuelle `IInstance`-Klasse. Dies bewerkstelligt, dass die Funktionalität leicht erweitert werden kann und das Backend auch über zum Beispiel Sockets, die von dieser Klasse erben, ansprechbar ist. Dies war ein erster Versuch in Richtung der angedachten LAN-Funktionalität, bei der mehrere Benutzer simultan am selben Code bzw. dem selben Backend arbeiten können; jedoch ist dies noch nicht implementiert und es würde den Rahmen der Semesterarbeit sprengen, dies noch zu implementieren.[16] Kommuniziert wird über Qt-typische `Signals` beziehungsweise über Funktionsaufrufe. Dies ist bei bidirektionaler Kommunikation zweier Klassen mittels Qt oftmals nötig, wenn auch nicht unbedingt ästhetisch.

Auf die Kommunikationsklasse wird nicht weiter eingegangen, da sie einfach ein kleines Detail der internen Kommunikation ist, die nur Funktionen bereitstellt, um die Infrastruktur generisch zu halten.

4.3 Settings

Die Einstellungen existieren so nicht, doch wie bereits in Sektion 3.2 erwähnt, existier für sie ein getrenntes Backend. Dieses ist nur für das Holen und Setzen von Einstellungen zuständig, die dann in einer Hash-Liste gespeichert werden.

[15] Die Version unter OS X verwendet auch die globale Tabbar, was einen eigenen Funktionsaufruf benötigt. Die Autoren halten das im Kontext einer sogenannten plattformunabhängigem GUI-Bibliothek für schlechtes Design, jedoch ist dies nicht der einzige Moment, in welchem sie über nur scheinbare Plattformunabhängigkeit stiessen.

[16] Die Autoren behalten es sich jedoch vor, dies in Zukunft noch zu tun, wenn Zeit und Muse dies zulassen.

Dies ist wahrscheinlich nicht die effizienteste Lösung, jedoch ist es sehr einfach und expressiv, damit zu arbeiten.[17] Die Klasse steht nur mit dem eigentlichen Backend in Kontakt.

4.4 Execution Context

Der ausführende Kontext ist das Element, das der meisten Erklärung bedarf und gleichzeitig jenes, das die meisten Konzepte bündelt. Er ist das Kernstück des Editors, das das Programm überhaupt multimedial werden lässt. Es können, so wie Editor-Instanzen, beliebig viele nebeneinander existieren, jedoch kann immer nur ein Kontext pro Editor geöffnet werden. Er implementiert eine abstrakte Basisklasse und ist zugleich Subklasse der virtuellen Klasse `QThread` und findet seine abschliessende Implementation in den Klassen `GlLiveThread`, `PyLiveThread`, `PySoundThread` und `QSoundThread`, wobei jede dieser Klassen wiederum einen Interpreter beheimatet. Alle Interpreter laufen demnach in eigenen Threads ab. Den einzelnen Interpretern soll im Folgenden nachgegangen werden.

4.4.1 GlLiveThread

Der GlLiveThread beheimatet den Renderer, der in GLSL geschriebene Fragment Shader entgegennimmt und versucht, diese zu kompilieren und auf der Grafikkarte auszuführen. Seine Hauptarbeit besteht also darin, den Kontext für die Ausführung bereitzustellen, das heisst ein OpenGl-spezifisches Setup und ein `QPaintDevice`. Da es nun auch möglich ist, auf externe Aktionen zu reagieren(Mausposition und Audio-Output des PCs), müssen auch diese von der Klasse erfasst werden.

4.4.2 PyLiveThread

Dieser Thread ist für den rudimentären Python-Interpreter zuständig. Im Prinzip tut er nur wenig: er initialisiert den Interpreter, übergibt ihm den zu verarbeitenden Programmtext und formatiert das Ergebnis, wobei er auf eventuell auftretende Exceptions im Interpreter Rücksicht nimmt.

4.4.3 PySoundThread

Ähnlich wie der normale Python-Thread funktioniert der sound-spezifische, jedoch läuft dieser in einer Schleife, um immer wieder Audio-Daten zu erstellen(in 8-Kilobyte-Paketen). Der wesentliche Unterschied zwischen diesem und dem vorgenannten Kontext besteht darin, dass der Code des Benutzers nur am Anfang ausgeführt wird und danach eine bibliotheksinterne Funktion, die einen Generator darstellt(quasi eine Funktion mit internem Status). Somit läuft dies ähnlich dem Streaming-Prinzip, das vom Grafik-Renderer ja auch zumindest emuliert wird(auch jener läuft in einer potentiell endlosen Schleife).

[17] Die Autoren glauben dem Streben nach Optimierung, sind jedoch auch der Ansicht, dass Entwicklerzeit bei lokalen Desktop-Anwendungen sehr viel teurer ist als CPU-Zeit. Daher achten sie auf die Komplexität ihrer Algorithmen, jedoch nur an kritischen Stellen auf die Komplexität verschiedener Datenstrukturen.

4.4.4 QSoundThread

Jener Thread ist im Moment noch nicht korrekt implementiert und liefert, egal welcher Input gegeben wird, immer nur eine kurze Mitteilung zurück, die besagt, dass alles in Ordnung sei. Hier wird die QML[18]-Engine ansetzen, die die Interpreter-Funktionalität um eine Qt-interne Sprache erweitern soll.

4.5 Zusammenfassung

Zusammenfassend lässt sich, zumindest nach Ansicht der Autoren, doch von einem relativ modularem Design sprechen, das in die drei Hauptkomponenten Editor, Backend und ausführender Kontext aufspaltbar ist, wobei der letztgenannte austauschbar ist und es ermöglicht, verschiedene Technologien einzusetzen. Dieses Design machte die Erstellung des Editors in seinem momentanen Zustand erst möglich.

[18] Qt Meta Language/Qt Modeling Language: Auf JSON basierende Modeling-Sprache.

5 Offene Fragen

Die vorliegende Implementation ist gewissermassen als Alpha-Version anzusehen. Sie ist an manchen Stellen noch unsauber und nicht sehr geschickt; teilweise werden die Instanzen nicht sauber aufgeräumt und die Audio-Programmierung läuft nur rudimentär. Die Grafikprogrammierung wiederum ist schon als ein fast fertiges Produkt anzusehen, hier bleibt nur noch der Renderer, der aufgeräumt und vielleicht auf mehrere Klassen aufgeteilt werden könnte. Dieser Unterschied in der Betriebsbereitschaft liegt wohl darin begründet, dass die Grafik auf OpenGL aufbaut, wohingegen die Audioprogrammierung von Grund auf neu geschrieben wurde.

Einige Dinge sind noch zu tun, sowohl für die vorgegabenen Anforderungen als auch für selbstgestellte. So sind im Moment folgende Fehler bekannt:

1. Die Audio-Ausgabe funktioniert bisher nur unter Windows. Grund ist ein insuffizienter Prozessor für den Audio-IO.
2. Das Laden von Bildern vom Fragement Shader aus ist fehlerhaft. Der Grund ist bis dato nicht bekannt(es werden Bilder geladen, jedoch die falschen).
3. Die Syntax-Hervorhebung wird nur bei Editorstart angepasst.
4. Das Laden von neuen Designs führt ohne das Schliessen des Einstellungsfensters unter gewissen Umständen zu Darstellungsproblemen bei einigen Widgets.

Hierbei ist zu beachten, dass das dritte Problem nur deshalb besteht, da die Autoren noch keine elegante Lösung gefunden haben, um das Backend mit dem Highlighter kommunizieren zu lassen, da diese über vier Klassenverbindungen miteinander verbunden sind, die alle durchlaufen werden müssten. Diese Redundanz gefällt nicht. Das vierte Problem wiederum könnte ein Qt-internes Problem sein, da es nicht durch `repaint()` oder `update()` beeinflusst wird.

Desweiteren decken die Tests zwar alle `public`-Schnittstellen des Programms ab, könnten aber sehr viel mehr mit GUI-Elementen interagieren. Hier müssen die Autoren sich noch mit dem gegebenen Bezugssystem(in jenem Fall `QTest`) auseinandersetzen und dies verbessern. Nach Meinung der Autoren gibt es jedoch keine Speicherlecks(Valgrind zeigt lediglich noch Speicherlecks in Fremd-Code an).

Die Autoren sind sich darin einig, dass sie das Projekt weiter betreuen möchten. Nachdem die Kanten geglättet sind, sollte ein neues, zeitgemässes Design geschaffen und hernach QML als weitere Skriptsprache eingeführt werden. Auch den Editor durch das Anbieten einer skriptbasierten Pluginschnittstelle erweiterbar zu machen, ist ein Gedanke. Weitere Funktionalitäten sind zu diesem Zeitpunkt noch nicht geplant.

6 Schlussworte

> Most good programmers do programming not because they expect to get paid or get adulation by the public, but because it is fun to program.
>
> Linus Torvalds

Am Ende bleibt den Autoren nur zu sagen, dass das Projekt sehr lehrreich war. Durch die recht freie Arbeit konnten sie den eigenen Interessen in der Programmierung nachgehen und trotzdem produktiv sein,[19] und so versuchten sie, möglichst viele neue Konzepte zu erarbeiten und sich in verschiedenen Bereichen der multimedialen Gestaltung von Software ein gewisses Grundwissen zu erarbeiten.[20]

Und am Ende besteht die Hoffnung, dass ein Programm geschaffen wurde, das mehr Menschen als nur den Fabrikanten eine Freude machen wird. Doch bevor eine eventuelle Bewerbung des Fabrikats stattfinden kann, wollen sie die Software-Qualität und das Design auf eine professionellere Ebene bringen. Da nun der Prototyp steht, kann man sich um die Rafinesse der Details kümmern.

Ein weiteres Mal danken die Autoren Tobias Brosge und Veit Heller ihrem Dozenten Sebastian Bauer für die Unterstützung und Offenheit gegenüber ihren Ideen und dem etwas gewundenen Entwicklungsprozess; die Gedanken mussten sich erst noch setzen, reifen und mit der Realität abgeglichen werden.

[19] Denn ein gewisser Druck besteht bei universitären Arbeiten immer.

[20] So ist dies zum Beispiel der erste mit Latex gesetzte Text, den die Autoren je erarbeitet haben.

Literatur

[EE11] Alan Ezust and Paul Ezust. *Introduction to Design Patterns in C++ with Qt, 2nd Edition.* Prentice Hall, 9 2011.

[HLSW10] Nicholas Haemel, Benjamin Lipchak, Graham Sellers, and Richard S. Wright. *OpenGL Super Bible, Fifth Edition.* Addison Wesley, 2010.