Diplomarbeit

Thema: Moderne Ansätze zur Oberflächengestaltung für hardwarenahe Programmierung

I Inhaltsverzeichnis

Seitenüberschlag:

1	1
2.1	2
2.2	3
3	3
4.1	2
4.2	3
4.3	3
4.4	1
5.1	1
5.2	3
5.3	1
5.4	1
6.1	3
6.2	2

6.3 5
6.4 25
6.5 1
7 1
8 1
Summe: 62

1 Motivation

[Diese Diplomarbeit ist aus einem Bedürfnis entstanden, ein geeignete Technologie für die hardwarenahe Programmierung von Benutzeroberflächen zu verwenden. Es gibt dafür bereits eine Vielzahl von Möglichkeiten, welche jedoch mir persönlich aus unterschiedlichen Gründen nicht zusagten, da sie mir häufig zu statisch vorkamen oder verbunden waren mit einem Vendor Lock-in. Wenn man in den Bereich der Webentwicklung blickt erkennt man, welche Möglichkeiten vorhanden sind. Ich hatte das Gefühl, dass für die hardwarenahe Programmierung Platz nach oben ist in Bezug auf Dynamik und Agilität. Mit dieser Arbeit möchte ich dies unter Beweis stellen.

Meine Haupt- oder Leitthese lautet daher: Es ist möglich, bisherige Technologien zur hardwarenahe Programmierung von graphischen Oberflächen durch neue, innovative Konzepte aufzuwerten, um eine agile Entwicklung zu vereinfachen.] → überarbeiten

2 Begriffsklärungen

Klärung Begriff „modern“
[Erklärung]
→ sehr agil entwickeln; ständige Anforderungsänderungen; Entscheidungen möglichst lange hinauszögern

Klärung Begriff „hardwarenah“
[Erklärung]

→ Nicht auf Anforderungen für Bewertungskriterien eingehen. Lediglich allgemein erklären, worum es sich handelt (limitierte Rechen- und Energieresourcen, sensitives Zeitverhalten usw.)

Klärung Begriff „Prototyp“
[Erklärung]

- Unterschied Prototyp + Software

Klärung Begriff „graphische Benutzeroberfläche“
Im laufe der Diplomarbeit wird häufig der Begriff „graphische Benutzeroberfläche“ o.ä. genutzt. Hierbei wird sich ausschließlich auf Benutzeroberflächen bezogen, welche eine gewisse visuelle Ästhetik besitzen. Darunter fallen Displays, welche Inhalte pixelgenau wiedergeben können, sowie über eine Farbtiefe von mindestens 24-Bit verfügen.

Klärung Begriff „Softwarearchitekturen“

3 Vorbetrachtungen zu Möglichkeiten der Programmierung von Benutzeroberflächen

3.1 Bisherige Architekturen für die Programmierung von Benutzeroberflächen

Graphische Benutzeroberflächen sind zu einem festen Bestandteil von jedem Softwaresystem geworden. Für die Realisierung ist eine Vielzahl an verschiedenen Lösungen entstanden, welche Ähnlichkeiten aufweisen [1]. Es ist daher schwierig, einzelne Paradigmen strikt voneinander zu trennen, da Überlappungen auftreten. Dennoch sollen in diesem Kapitel typische Konzepte vorgestellt werden, welche sich historisch herausgebildet haben.

3.1.1 Formulare und Kontrollelemente

Dieses Softwarekonzept ist relativ einfach aufgebaut. Die Grundidee besteht darin, graphische Oberflächen über Formulare abzubilden, welche mehrere Kontrollelemente beinhalten. Über die Kontrollelemente werden Daten angezeigt, sowie eine Interaktion ermöglicht.

Eine Grundproblematik in diesem Konzept besteht darin, die Daten synchron zu halten. Häufig befinden sich die Daten auf der GUI, intern in der Geschäftslogik und auf einem persistenten Speichermedium, teilweise mehrfach [1]. Ein populäres Muster, um dieses Problem zu lösen ist Datenbindung. Die Idee ist Änderungen in einem Datensatz sofort zu propagieren und so an allen nötigen Stellen zu ändern. Wenn die Übertragung abgeschlossen ist, wurde der Datensatz an allen Stellen geändert [2].

Dazu ein Beispiel: Gegeben sei eine Softwareanwendung, welche aus einem Formular mit jeweils einem Kontrollelement Textfeld, Button und einem Ausgabefeld besteht. Die Funktion des Textfeldes besteht darin, eine Namenseingabe zu ermöglichen. Über den Button soll die Eingabe bestätigt werden können. Nach der Bestätigung, soll in dem Ausgabefeld der Nutzer gegrüßt werden, indem vor dem eingegebenen Namen ein Begrüßungswort gehangen wird, also beispielsweise „Hallo Anna“. In dieser Anwendung könnten laut dieser Architektur je Kontrollfeld 3 Variablen existieren: Die erste bildet den Zustand auf dem Bildschirm ab. Die Zweite bildet den internen Zustand im Programm ab und eine dritte, bildet den Zustand auf einem Speichermedium ab, z. Bsp. eine Textdatei oder eine relationale Datenbank. Letzteres könnte den Zweck verfolgen, die Daten über einen Applikationsneustart zu erhalten.

Sobald der Nutzer anfängt, den Namen in das Textfeld einzugeben wird durch die Datenbindung nicht nur der Variablenwert für die GUI-Oberfläche, sondern gleichzeitig der Variablenwert innerhalb des Programms geändert. Nach Bestätigung des Namens kann sowohl der interne Variablenwert oder der GUI-Variablenwert für das Ausgabefeld

geändert werden. Dies spielt keine weitere Rolle, weil die Änderungen in jedem Fall über das Muster Datenbindung durchgegeben werden.

Was in diesem simplen Anwendungsfall einfach aussieht, stellt sich in der Praxis jedoch als trickreich heraus, da Zyklen vermieden werden müssen. Daher falls die Daten intern geändert werden und diese Änderung an die GUI übergeben wird, darf diese nicht wiederrum die Änderung an die interne Variable weitergegeben werden. Da jedoch grundsätzlich in diesem Konzept eine bidirektionale Änderung der Daten möglich sein muss, legt die Komplexität dieses Verfahrens einen Einsatz von Software-Frameworks nahe [1].

Dabei implementieren Software-Frameworks dieses Verfahren intern häufig über das Beobachter-Muster [1]. Dies bedeutet, dass entsprechende Ereignisse ausgelöst werden, wenn sich die Variablen ändern. In den Ereignisbehandlungsroutinen muss die Änderung der jeweils redundanten Datensätzen vorgenommen werden.

Historisch gesehen wurde diese Architektur gehäuft bei Client-Server Applikationen eingesetzt, da es die Funktionalität der Änderung von Datensätzen sehr gut abbildet, insbesondere die Änderung von Datensätzen in relationalen Datenbanken. In dieser Domäne wird dieses oft benötigt [1].

3.1.2 Model View Controller

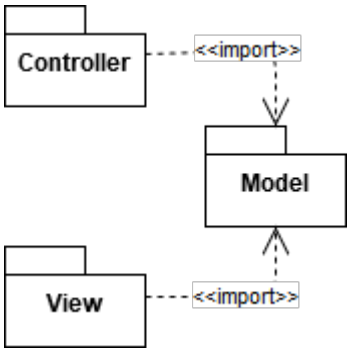
Der Model View Controller (MVC, engl. für Modell-Präsentation-Steuerung) ist ein weiteres Konzept für die Strukturierung von Software zur graphischen Oberflächengestaltung. Es ist mit eines der historisch ältesten Konzepte für die graphische Programmierung. Zurückblickend, waren in den 1970er Jahren GUI‘s keineswegs gängig. Das MVC-Konzept ist mit eines der ersten Ansätze, Benutzeroberflächenprogrammierung in großem Umfang zu betreiben. Im Laufe der Zeit, wurden mehrfach Konzepte als MVC bezeichnet, welche jedoch eher wenig mit dem Ursprung gemein hatten [1]. Deswegen ist es umso wichtiger, dieses Paradigma explizit zu besprechen, um eine Abgrenzung von anderen Konzepten zu ermöglichen.

Die Anfänge sind stark mit der Programmiersprache Smalltalk verzweigt, welche zugleich als eine der ersten objektorientierten Programmiersprachen gilt [3]. Sie geht in die 80er Jahre zurück [1]. Im folgenden Verlauf der Diplomarbeit wird der Begriff MVC auf das Konzept, welches Smalltalk verwendete, eingegrenzt.

Eine der wichtigsten Errungenschaften von MVC ist die Separierung von graphischen Elementen und Domain-Objekten. Letztere sollten in der Lage sein, komplett losgelöst von jeglichem graphischen Inhalt zu funktionieren. Sie bilden unsere Sicht auf die Welt ab. Sie müssen in der Lage sein, mehrere Präsentation-Schichten zu bedienen. In der Unix-Welt wird dies stark genutzt, da nahezu jede Applikation sowohl über GUI, als

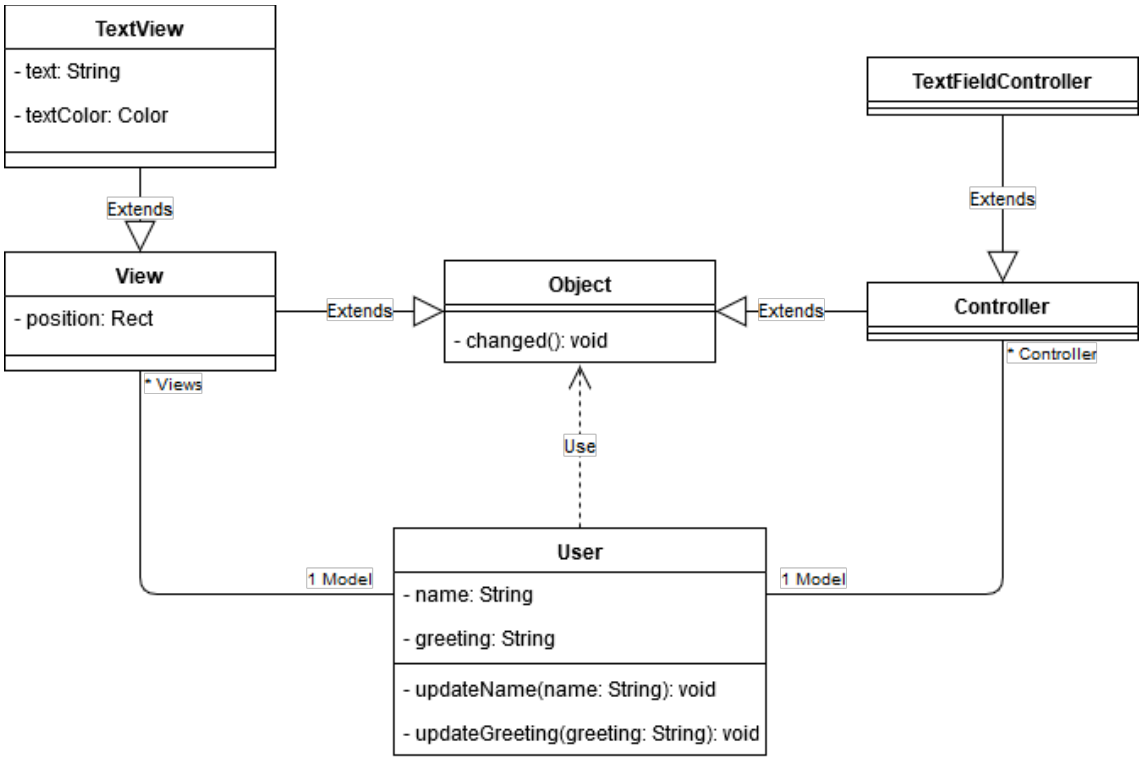
auch über Kommandozeile ansteuerbar ist. Dies ist nach aktuellem Stand immer noch der Fall.

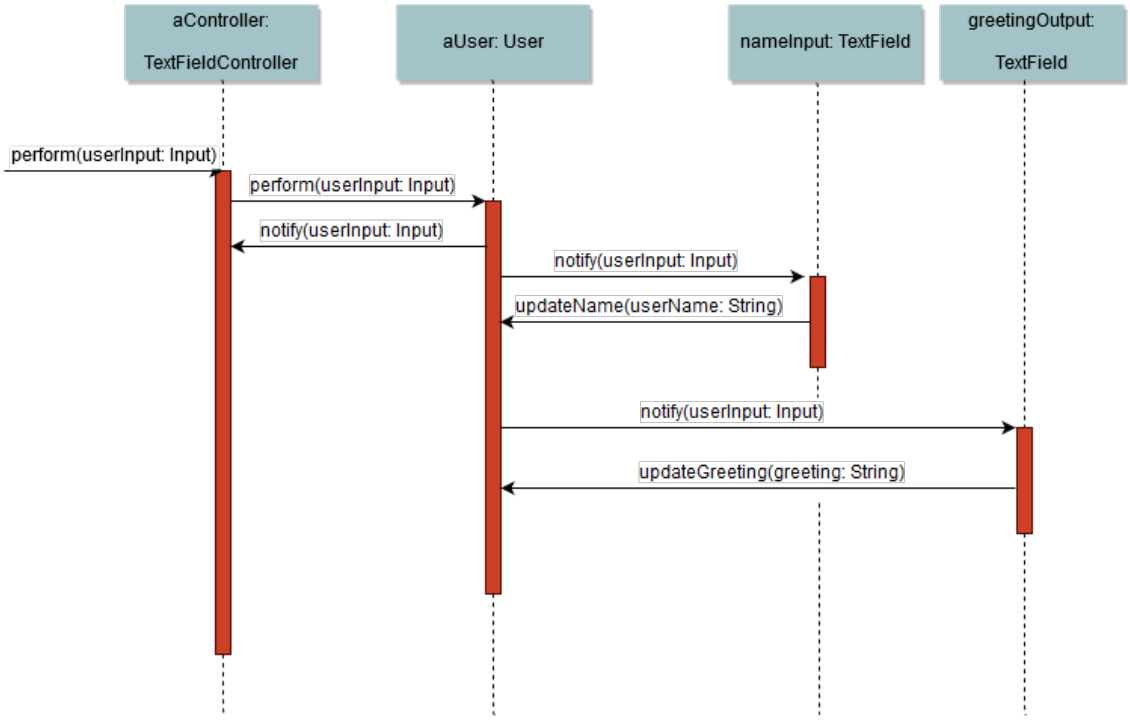
Um das Architektur-Muster im Groben zu beschreiben, werden Domain-spezifische-Objekte innerhalb des Musters als „Model“ bezeichnet. Die graphischen Elemente bilden den „View“. Die Aufgabe des „Controller“ ist es, Benutzereingaben zu verarbeiten. Dabei kennt im strengen Sinn das Model weder Controller noch View. Ebenso kennen sich Controller und View nicht. Folgende Abbildung soll dieses Zusammenspiel im groben als Paketdiagramm verdeutlichen.



1. Abbildung: Zusammenspiel Model View Controller

Das Model kann dabei als Abbildung der realen Welt gesehen werden. Anders als in dem Formulare und Kontrollelemente-Architektur-Muster geht dieses Model nicht von Datensätzen aus, die manipuliert werden sollen, sondern von Objekten, mit welchen man in der Anwendung agiert. Als Beispiel soll der gleiche Anwendungsfall dienen, welche unter Formulare und Kontrollelemente ausgewählt wurde. Wenn man die Begrüßungsanwendung über die MVC-Architektur umsetzen würde, so könnte das Model aus einem Objekt bestehen, welche den Namen beinhaltet.


TextView
- text: String
- textColor: Color
Extends
View
- position: Rect
Extends
Object
- changed(): void
Extends
Controller
TextFieldController
Extends
* Views
* Controller
Use
User
- name: String
- greeting: String
- updateName(name: String): void
- updateGreeting(greeting: String): void
1 Model
1 Model


aController: TextFieldController
aUser: User
nameInput: TextField
greetingOutput: TextField
perform(userInput: Input)
perform(userInput: Input)
notify(userInput: Input)
notify(userInput: Input)
updateName(userName: String)
notify(userInput: Input)
updateGreeting(greeting: String)

[4]

3.1.3 MVP

3.1.4 MVVM

[1]

3.2 Vorgehensweise für die Programmierung von Benutzeroberflächen

Für die Entwicklung von Benutzeroberflächen muss sowohl das graphische Design, als auch die Programmierung erfolgen. Da die Programmierung mit hohen Kosten belegt ist, ist es eine valide Vorgehensweise, das graphische Design zu erstellen und anschließend mit der gewünschten Funktionalität hinterlegt werden. Das graphische Design kann dazu mittels geeigneten Anwendungen, z. Bsp. Photoshop [5] oder proto.io [6] erstellt und im Roh-Zustand getestet werden. Wenn das Design zusagt, kann anschließend die Implementierung erfolgen. Dazu müssen die graphischen Elemente in geeigneter Form, z. Bsp. als *.png oder *.jpg Dateien exportiert und gespeichert werden. Ebenso müssen andere Ressourcen, wie Fonts oder Animationen in geeigneter Form abgelegt und exportiert werden.

Ein Vorteil von dieser Vorgehensweise besteht darin, dass das graphische Design und die Implementation sauber getrennt voneinander erfolgen kann. Die Schnittstelle zwischen Designer und Programmierer ist an dieser Stelle deutlich durch die extrahierten Dateien gegeben. Dieser Fakt wird sich in der Diplomarbeit zunutze gemacht, sodass das graphische Design von einem Externen erstellt wird. Dadurch kann die visuelle Qualität des Endproduktes gewährleistet bleiben.

3.3 Moderne Anforderungen an die Programmierung von Benutzeroberflächen

MVP

3.4 Anforderungen durch hardwarenahe Programmierung

Hardwarenahe Programmierung weist im Vergleich zur regulären Softwareentwicklung Besonderheiten auf. Diese sind hpts. durch das Umfeld bedingt, in dem hardwarenahe Programmierung Anwendung findet. Auf typische Besonderheiten soll im Folgenden eingegangen werden.

Rechenleistung

→ Betrachtungen zum Prozessor

→ häufig 8- oder 16-bit, dieser Fall wird aber nicht betrachtet

→ Abgrenzung zu Microcontrollern an dieser Stelle

Memory,

→ Eingeschränkt

Entwicklungskosten vs. Produktionskosten

→ Trade-Off zwischen Entwicklungskosten und Produktionskosten erklären

→ Erklären, dass über diese Ansätze lediglich Entwicklungskosten gesenkt werden sollen und dadurch minimal-geringe Mehrkosten in der Produktion in Kauf genommen werden → hängt von Anzahl an ausgerollten Geräten ab

Lebensdauer

→ ...

Ausfallsicherheit

→ ...

[Garbage Collection sorgt für unvorhergesehenes Zeitverhalten → nicht benutzen]

[Möglichst selten Speicher-Allokalisierung durchführen]

[Evtl. Ausflug in Fragmentierung von heaps geben → ganz zum schluss schreiben, evtl. weglassen]

[7, S. 5]

4 Auszeichnungssprachen

4.1 Definition und Funktion von Auszeichnungssprachen

Eine Auszeichnungssprache (engl. „Markup Language“) ist eine Computersprache, welche ein Dokument um syntaktische Elemente erweitert, um eine maschinelle Verarbeitung zu ermöglichen [8].

Eine Markup Language kann nicht wie bei Programmiersprachen üblich direkt nach Maschinencode kompiliert werden. Stattdessen wird eine Dokument, welches mithilfe einer Markup Language erstellt wurde von Computerprogrammen geladen und interpretiert. Das Computerprogramm führt anhand des eingelesenen Dokumentes weitere Computerabläufe aus.

Markup Languages werden für die Verlinkung von Dokumenten untereinander, für die Strukturierung von Inhalt innerhalb eines Dokumentes, für den Datenaustausch oder für die Definition von anderen Auszeichnungssprachen eingesetzt [9]. Dabei hat jede Markup Language eigene Syntax-Elemente. Häufig kann zwischen einzelnen Syntax-Elementen unformatierter Fließtext stehen.

Markup Languages haben in verschiedenen Domänen Anwendung gefunden. Sie werden u.a. eingesetzt für:

- das Schreiben von Textdokumente (z. Bsp. „LaTeX“ oder „Markdown“)
- das Abbilden von Websites (z. Bsp. „HTML“)
- den Datenaustausch zwischen Rechnerprogrammen (z. Bsp. „JSON“ oder „XML“)
- die Beschreibung von Serverkonfigurationen (z. Bsp. „Yaml“)
- die Definition der Metadaten einer Datei (z. Bsp. „JSON“ oder „XML“)
- die Graphische Oberflächengestaltung von Anwendungen (z. Bsp. „XAML“)

Diese Liste hat keinen Anspruch auf Vollständigkeit.

Weil Markup Languages für die Erstellung von Oberflächen eine starke Verbreitung gefunden haben, werden Sie in dieser Arbeit näher betrachtet.

4.2 Vorstellung ausgewählter Auszeichnungssprachen

4.2.1 XML

4.2.2 XAML

4.2.3 JSON

4.2.4 HTML

4.2.5 YAML

YAML (kurz für „YAML Ain't Markup Language“) ist eine benutzerfreundliche Datenserialisierungs-Sprache. Sie unterstützt verschiedene Sprachen und ist mit Focus

auf gewöhnliche native Elemente designt worden [10]. Das Attribut „benutzerfreundlich“ bezieht sich dabei auf die menschliche Lesbarkeit der Dokumente relativ im Verhältnis zu anderen Auszeichnungssprachen.

Es soll auf markante Elemente der Sprache eingegangen werden, welche in der Diplomarbeit benötigt werden. Dazu dient als Grundlage folgendes einfaches Beispiel, welches von der Website [11] übernommen wurde:

```
---
id: 731
date: 16th of May 2011
customer: 17
items:
- quantity: 5
  description: Oolong
  price: 5.98
  in-stock: true

- quantity: 2
  description: Assam
  price: 2.95
  in-stock: false
```

Über den 3 Anstrichen ‚---‘ kann Metainformationen zu dem Dokument stehen. Dies ist jedoch für diese Diplomarbeit nicht weiter relevant, weswegen darauf nicht weiter eingegangen wird. Man kann in dem Beispiel erkennen, dass ein YAML-Dokument im Kern aus 3 Elementtypen besteht: Maps, Arrays und Scalar-Werten. Ein YAML-Dokument besteht dabei aus einer Map, einem Array oder einem Scalar-Wert. Eine Map besteht aus Schlüssel-Werte-Paaren. Dabei kann jeder Schlüssel und jeder Wert wiederum eine Map, ein Array oder ein Scalar-Wert sein. In dem Beispiel sind als Schlüssel lediglich Scalar-Werte angegeben. Schlüssel werden von Werten mithilfe eines Doppelpunktes ‚:‘ voneinander getrennt. Ein Array besteht aus beliebig vielen Maps, Arrays und Scalar-Werten. Jedes Element eines Arrays wird dabei über einen Anstrich ‚-‘ angegeben. Jeder Wert einer Map und jedes Array-Element muss auf fortlaufenden Zeilen eingerückt werden. Ein Scalar-Wert besteht aus einem Ausdruck. Dieser Ausdruck kann beispielsweise ein boolscher Wert, eine Zahl oder ein Text sein.

In weiteren Kapiteln wird sich auf Map(s), Array(s) und Scalar-Werte über YAML-Map(s), YAML-Array(s) bzw. YAML-Scalar-Wert(e) bezogen. Ebenso findet eine Referenzierung von Schlüsseln und Werten einer YAML-Map über YAML-Map-Schlüssel bzw. YAML-Map-Wert statt.

Es gibt in dem YAML-Standard noch weitere grundlegende Elemente, wie beispielsweise Anker und Textblockabsätze, welche für diese Arbeit aber keine Relevanz haben und deswegen an dieser Stelle nicht weiter aufgeführt werden.

4.2.6 QML

4.3 Problematik der Typisierung bei Auszeichnungssprachen

[Kurz auf die Problematik der Typisierung eingehen → Implizite Typvergabe bei embedded Programming häufig nicht gewollt]

5 Bewertung von bisherigen Technologien zur Eignung für hardwarenahe Benutzeroberflächenerstellung

In diesem Kapitel sollen bisherige Technologien für die Programmierung von Benutzeroberflächen analysiert werden und hinsichtlich ihrer Eignung für die hardwarenahe Programmierung ausgewertet werden. Als Mittel, um dies zu bewerkstelligen wurde der Vergleich gewählt.

5.1 Bewertungskriterien

Die Bewertungskriterien werden sowohl aus Anforderungen für die Programmierung von graphischen Benutzeroberflächen, als auch aus Anforderungen für die hardwarenahe Programmierung abgeleitet. Deswegen werden diese beiden Bereich zunächst separiert betrachtet. siehe: Anforderungen an die Programmierung von Benutzeroberflächen

5.2 Anforderungen für die Programmierung von graphischen Benutzeroberflächen

- Formulare und Kontrollelemente

- MVP

- MVC

- MVVM

- No Vendor Lock-In [12]

- Open Source?

5.3 Anforderungen für hardwarenahe Programmierung

- Verzicht auf Garbage-Collector

- Portabilität für Cross-Plattform-Entwicklung

- Zugriff auf statische Resourcen

5.4 Ausgewählte Technologien

[Zu jeder Möglichkeit min. einen prominenten Vertreter auswählen, Technologien kurz vorstellen]

- QT

- WPF mit XAML

- Xamarin mit XAML

- React Native (Javascript + HTML + CSS)

5.5 Vergleich

[Punkte vergeben, jede Punktevergabe kurz begründen]

[inklusive kurzes Fazit]

6 Prototypische Entwicklung

6.1 Anforderungen an den Prototyp „XXX“

Ziel der Diplomarbeit ist es neue Ansätze für die hardwarenahe Programmierung von Benutzeroberflächen zu erforschen. Diese Ansätze sollen mithilfe eines Prototypen praxisnah veranschaulicht werden. Im Rahmen der Anforderungsanalyse konnten grundsätzlich folgende Bereich von Interesse ermittelt werden:

- Smart-Home-Anwendungen
- Anwendungen für Automaten jeglicher Art
- Anwendungen für Bürogeräte
- Mobile Anwendungen

Smart-Home-Anwendungen

Unter den Begriff „Smart-Home-Anwendungen“ wird jegliche Software gezählt, welche lokal vor Ort bei Endnutzer auf den Geräten installiert ist, die zum Zweck hat hauswirtschaftliche Abläufe zu steuern und zu automatisieren. Dazu zählt unter anderem die Kontrolle der Heizung, der Fenster, der Türen, sowie sonstigen Haushaltsgeräten, wie beispielsweise Kühlschrank, Geschirrspüler und Waschmaschine.

Für den Prototyp sind Smart-Home-Anwendungen interessant, welche über eine graphische Oberfläche mit dem Endnutzer interagieren.

Anwendungen für Automaten jeglicher Art

Unter „Automaten jeglicher Art“ fallen sämtliche Automaten, welche nach Anbindung an ein Softwaresystem direkt nutzbar sind und über einen vordefinierten Funktionsumfang verfügen. Darunter fallen beispielsweise Fahrkartenautomaten, Getränkeautomaten und Selbstbezahlungsautomaten.

Anwendungen für Bürogeräte

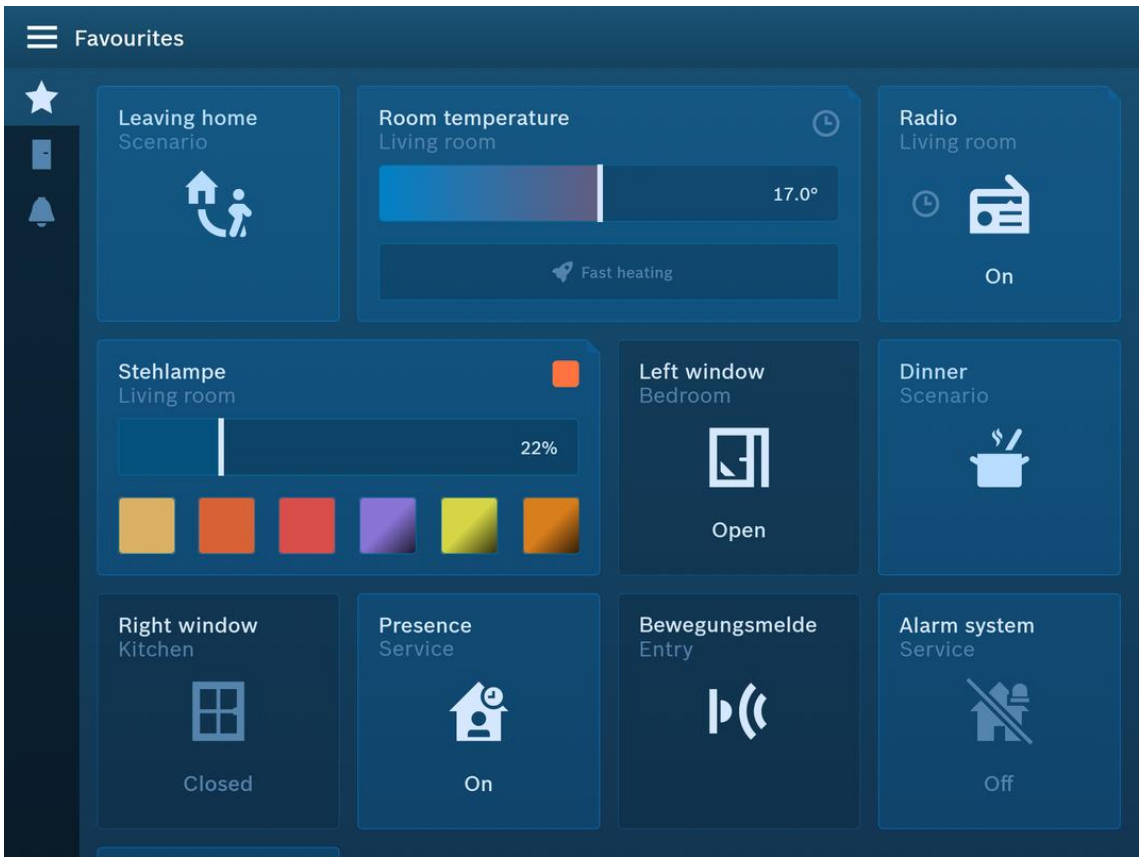
Mit Bürogeräten werden Geräte für den Bürobedarf abgedeckt. Darunter fallen Drucker und Scanner.

Mobile Anwendungen

Mobile Anwendungen sind Anwendungen für Geräte, welche ohne erheblichen Mehraufwand im Alltag mitgeführt werden können. Darunter fallen Smartphones und Smartwatches. Nicht dazu zählen Notebooks.

Alle Bereiche eint, dass stets limitierte Hardware zu berücksichtigen ist. Der Prototyp soll exemplarisch häufig vorkommende Funktionalität in der Benutzerführung abbilden. Dementsprechend müssen funktionale und nicht-funktionale Anforderungen festgelegt werden.

Um eine möglichst genaue Abbildung der Anforderungen aus der Praxis zu gewährleisten, wurde als Grundlage ein Screenshot der Applikation "Bosch Smart Home" von der Robert Bosch GmbH für Android gewählt, siehe Abbildung 2. Dieser Screenshot wird aktuell benutzt, um die Applikation im Google Play Store zu vertreiben und deckt damit sowohl das Feld „Mobile Applikationen“, als auch „Smart-Home-Anwendungen“ ab.



2. Abbildung: Screenshot der Android-Applikation "Bosch Smart Home" von der Robert Bosch GmbH

Die Anforderungen wurden anhand des Screenshots festgelegt.

6.1.1 Funktionale Anforderungen

/101/ Overlay:
Der Prototyp besteht aus einem Overlay. Innnerhalb des Overlays müssen Titelleiste, Navigationsbar und Inhalt angeordnet werden.

/102/ Titelleiste:

Die Titelleiste muss aus einem Symbol bestehen und einem Textbanner. Das Symbol muss klickbar sein. Bei Klick muss in der Konsole eine Benachrichtigung erfolgen. (Wenn es vom Umfang her noch gestreckt werden soll, könnte man sich hier noch zusätzliche Späße ausdenken.)

/103/ Navigationsbar:
Die Navigationsbar muss aus einem Tab "Favoriten" und weiteren Tabs bestehen. Der Tab "Favoriten" ist bei Start der Software ausgewählt. Jeder Tab ist durch ein unterschiedliches Symbol dargestellt. Bei Klicken eines Tabs muss sich die Titelleiste entsprechend ändern, sowie der Inhalt angepasst werden. Es muss visuell markiert sein, welcher Tab ausgewählt ist.

/104/ Inhalt:
Der Inhalt ist leer, solange ein anderer Tab wie "Favoriten" ausgewählt ist. Falls der Tab "Favoriten" in der Navigationsbar ausgewählt ist, so besteht der Inhalt aus einem Scrollview, welcher ein Grid-Layout beinhaltet. In dem Gridlayout müssen unterschiedlich große Kacheln angeordnet werden.

/201/ Kacheln [TODO]:
[An dieser Stelle würde ich die Kacheln, welche man auf dem Screen sieht definieren, sowie eine Kachel "Add to favorites", welche bei klick ebenso eine Meldung in der Konsole wirft.]

Leaving home - Einfache Meldung in der Konsole
Room temperature - Leiste für temperature. Wenn fast heating aktiviert wird, steigt temperatur simulationsmäßig langsam an.
Radio - on/off switch
Stehlampe - Scrolleiste für Helligkeit in %. Farbauswahl.
Left window - open/close switch, evt. mit verzögerung
Dinner - Einfache Meldung in der Konsole
Right window - open/close switch
Presence - on/off switch
Bewegunsmelder - statisches feld
Alarm system - on/off switch

6.1.2 Qualitätsanforderungen

/301/ Visuelle Aufbereitung:
Der Prototyp muss ansprechend visuell aufbereitet sein.

6.1.3 Rahmenbedingungen

6.1.3.1 Technisch/ Technologische Rahmenbedingungen

/401/ Statische Favoriten:
Die Favoriten dürfen für den Prototyp als statisch angenommen werden und müssen nicht dynamisch geladen werden.

/402/ Portabilität:
Die Portabilität der zugrunde liegenden Programmiersprache soll gewährleistet bleiben.

/403/ Hardwarenahe Programmierung:
Die Programmierung muss mit einer hardwarenahen Programmiersprache, wie „C“ oder ähnlich umgesetzt werden.

6.1.3.2 Rechtliche Rahmenbedingungen

6.1.3.3 Organisatorische Rahmenbedingungen

6.2 Lösungskonzepte bzgl. der Technologie

In diesem Kapitel sollen die Lösungskonzepte bzgl. der genutzten Technologie für den Prototypen vorgestellt werden.

6.2.1 Verwendete Technologien und Software-Bibliotheken

Um das Rad nicht neu zu erfinden, wurde für den Prototypen auf verschiedene Technologien und Software-Bibliotheken zurückgegriffen. Diese sollen in diesem Kapitel vorgestellt werden.

6.2.1.1 CMake
CMake ist eine Kollektion an Werkzeugen, welche erstellt wurde um Software zu kompilieren, zu testen und zu archivieren. CMake ist dabei Compiler- und Plattform-unabhängig und unter einer Open-Source-Lizenz verfügbar [13].

Für das Projekt wurde CMake Version „3.12.1“ verwendet für jenen Zweck. Dadurch kann eine IDE- und Plattform-Unabhängigkeit gewährleistet bleiben. Ebenso können über CMake verschiedene Code-Generierungsskripte in den Compilierungsprozess integriert werden. Diese Funktionalität wurde ebenso genutzt.

6.2.1.2 Imps Engine, Expat, SFMT
Die Imps Engine ist eine Sammlung von Software-Bibliotheken für die Plattform-unabhängige Entwicklung von Applikationen in der Programmiersprache „C“ [14]. Sie ist verfügbar unter einer Open-Source-Lizenz, der MIT-Lizenz.

Die Imps Engine hat als interne Abhängigkeiten Expat und SFMT, welche durch die Benutzung vererbt werden, aber keine weitere Verwendung im Quellcode des

Prototypen finden. Die beiden Bibliotheken sollen nur kurz angerissen werden. Expat ist eine C-Software-Bibliothek um XML-Dokumente zu parsen, siehe [15]. SFMT (SIMD oriented Fast Mersenne Twister, wobei SIMD für „Single Instruction Multiple Data“ steht) ist eine C-Software-Bibliothek für die Erzeugung von Pseudo-Zufallszahlen [16]. Dazu wurde eine Variante des Mersenne Twister (MT) implementiert, welche von Mutsuo Saito und Makoto Matsumoto im Jahr 2006 vorgestellt wurde. Es verspricht eine bessere Performance im Vergleich zu ähnlichen Zufallsgeneratoren [17]. Die Generierung von Zufallszahlen ist für die Diplomarbeit jedoch nicht weiter Relevant, weswegen auf den Algorithmus nicht weiter eingegangen wird.

6.2.1.3 OpenCV

OpenCV (Open Source Computer Vision Library) ist ebenso unter einer Open-Source-Lizenz, der BSD-Lizenz verfügbar [18]. Sie wurde genutzt, um das Laden von Bilddateien zu ermöglichen. Die Software-Bibliothek ist für die rechnergestützte Erkennung von Bildern optimiert, wodurch sie wesentlich umfangreicher ist und nur ein minimaler Teil des Umfangs genutzt wurde. Es besteht die Möglichkeit, den Umfang des Prototypen zu reduzieren, indem das Laden der Bilddateien durch eine andere Software-Bibliothek unterstützt wird. Jeglicher Quellcode dieser Funktionalität, welcher ersetzt werden müsste, befindet sich dabei unter „deps/opencv-glue“.

6.2.1.4 OpenGL und GLEW

OpenGL ist ein Graphikkarten Standard für hohe graphische Qualität und Performance [19]. Er wird genutzt, um graphische Elemente darzustellen. Um die Entwicklung zu vereinfachen, werden Hilfsobjekte aus der Imps Engine genutzt.

GLEW (OpenGL Extension Wrangler Library) ist eine Plattform-unabhängige Open-Source C/C++ Software-Bibliothek für das Laden von OpenGL-Erweiterungen. Sie ist dafür geeignet, zur Laufzeit zu bestimmen welche OpenGL Versionen unterstützt werden und welche Erweiterungen verfügbar sind und wurde für diesen Zweck genutzt [20].

6.2.1.5 CLion, MinGW, Windows 8.1

Für die Entwicklung wurde die IDE „CLion“ von JetBrains genutzt, siehe [21]. CLion ist eine kommerzielle Anwendung, spezialisiert für die Programmierung in C und C/C++, inklusive nativem Support für CMake. Die Versionsnummer lautet „2018.2.3“.

Als Compiler wurde MinGW verwendet, siehe [22], Version „w64 6.0“. MinGW beinhaltet den Debugger „MinGW-w64 GDB“ mit Versionsnummer „8.1“.

Die Plattform-Umgebung ist Windows 8.1. Aufgrund der Tatsache, dass CMake verwendet wurde ist es kein Problem, IDE, Compiler oder Plattform auszuwechseln. Lediglich bei einer Änderung der Plattform müssen Windows-spezifische Teile angepasst werden, beispielsweise für das Laden von Bildern. Diese Anpassungen sind

für Android und iOS bereits in der Imps Engine enthalten und müssen lediglich korrekt verlinkt werden.

6.2.2 Objektorientierte Programmierung in C

Der Prototyp repräsentiert die praktische Anwendung von objektorientierter Programmierung (OOP) in der Programmiersprache C als hardwarenahe Programmiersprache. Dabei bietet die Programmiersprache C native keine Unterstützung für OOP [23].

Es ist festzuhalten, dass OOP nicht an syntaktische Elemente gebunden ist, sondern lediglich ein Programmierparadigma darstellt, nach welchem Daten und zugehöriger Code bzw. Funktionen gekapselt werden [24]. Die Objekte kommunizieren dabei über Funktionen miteinander.

Deswegen ist es grundsätzlich kein Problem, OOP in C durchzuführen. Es ist sogar eine Tatsache, dass populäre Bibliotheken in C, wie beispielsweise POSIX Threads [25] einen objektorientierten Aufbau verfolgen, siehe dazu die Spezifikation [26]. Man kann erkennen, dass Funktionen zu einem bestimmten Datentyp, wie „pthread_mutex_t“, über die Namensgebung, in diesem Falle „pthread_mutex*“, gruppiert und auf diese Weise an den Datentyp gekapselt werden. Daher ist diese Bibliothek mindestens teilweise von einem objektorientierten Aufbau geprägt.

Das Paradigma OOP bietet jedoch nicht nur die Kapselung von Daten und Funktionen, sondern auch erweiterte Konzepte und Muster, wie beispielsweise Vererbung oder bzw. das Beobachter-Muster. Um diese Funktionalität bereitzustellen, wurde die Imps Engine verwendet. Über eine Reihe von Annotationen in den Kommentaren können reguläre C-Structs mit Objektfunktionalität aufgewertet [27]. Dies wurde für die Diplomarbeit genutzt.

Todo: Header und C Datei example. Vererbung und Beobachter-Muster am Beispiel erklären

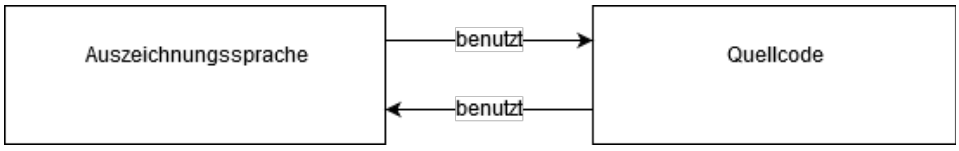
Da OOP in der Programmiersprache C nicht primärer Forschungsgegenstand ist, sondern lediglich für die Lösungskonzepte Anwendung findet, soll an dieser Stelle nicht weiter darauf eingegangen werden.

6.2.3 Verwendung einer Auszeichnungssprache

Es soll für die graphische Oberflächengestaltung eine Auszeichnungssprache angewendet werden. Als Grundlage soll der YAML-Standard dienen. Dieser wird weiter eingeschränkt.

Das Ziel ist es, über die Auszeichnungssprache eine Objekterstellung zu ermöglichen. Dazu soll das YAML-Dokument in C-Code übersetzt werden und innerhalb des Programms aufgerufen werden können, um die spezifizierte Oberfläche zu erstellen.

Dabei soll innerhalb des YAML-Dokumentes keine Referenz, auf den aufzurufenden Code durchgeführt werden. Dies impliziert, dass Funktionsreferenzen, wie beispielsweise die onClick-Methode eines Buttons nicht direkt verlinkt werden können. Hintergrund dazu ist folgender: Bei allen analysierten bisherigen Lösungen, z. B XAML oder QML kann eine gegenseitige Referenzierung zwischen Auszeichnungssprache und Quellcode, siehe Abbildung



6.2.4 Definition der Auszeichnungssprache

Folgende Grammatik wird festgelegt und näher erläutert:

Dokument := *YAML-Map-Schlüssel['__generate__'] : Objekt-Typ, (YAML-Map-Schlüssel : Element)**
Objekt-Typ := *Name*
Name := *\w[\w\d]**
Element := *(Objekt | Array | Struktur | Konstanter-Ausdruck)*
Objekt := *(YAML-Map-Schlüssel['__type__'] : Objekt-Typ)?,*
(YAML-Map-Schlüssel['__ref__'] : Objekt-Referenz)?,
*(YAML-Map-Schlüssel : Element)**
| YAML-Map-Schlüssel['__resource__'] : Resource-Provider,
*(YAML-Map-Schlüssel : (Objekt | Struktur | Konstanter-Ausdruck))**
Resource-Provider := *Name*
Objekt-Referenz := *Name*
Array := *YAML-Array[Array-Objekt]*
Array-Objekt := *(YAML-Map-Schlüssel : Element)**
Struktur := *(YAML-Map-Schlüssel : Konstanter-Ausdruck)** | *YAML-Array[Konstanter-Ausdruck]* | *Konstanter-Ausdruck*
Konstanter-Ausdruck := *Zeichenkette* | *Nummerischer-Ausdruck* | *Boolscher-Ausdruck* | *Konstante*
Zeichenkette := *Name*
Nummerischer-Ausdruck := *\d[\d\s\+\-*\/]**
Boolscher-Ausdruck := *true* | *false*
Konstante := *Name*

Die Verarbeitung der Auszeichnungssprache erfolgt kontextorientiert, dies bedeutet, abhängig davon in welchem Kontext der Parser sich befindet sind unterschiedliche Elemente zulässig und werden anders interpretiert. Jeder grammatikalische Ausdruck repräsentiert dabei einen Kontext. Teilweise können mehrere unterschiedliche Kontexte

auf einen grammatikalischen Ausdruck folgen. In diesem Fall wird der weitere Kontext sowohl anhand der Syntax des YAML-Dokumentes, als auch der dahinter liegenden Objekte und Strukturen bestimmt. Da die Grammatiksprache keinem Standard entspricht, werden kurz die einzelnen grammatikalischen Ausdrücke erklärt. Jede Zeile beschreibt dabei einen Ausdruck.

Dokument := *YAML-Map-Schlüssel['__generate__'] : Objekt-Typ, (YAML-Map-Schlüssel : Element)**

Ein Dokument besteht aus einer YAML-Map. Die YAML-Map muss dabei aus einem Schlüssel „__generate__“ bestehen, welcher auf ein Objekt-Typ referenziert. Anschließend können beliebig viel

Objekt-Typ := *Name*
Name := *\w[\w\d]**
Element := *(Objekt | Array | Struktur | Konstanter-Ausdruck)*
Objekt := *(YAML-Map-Schlüssel['__type__'] : Objekt-Typ)?,*
(YAML-Map-Schlüssel['__ref__'] : Objekt-Referenz)?,
*(YAML-Map-Schlüssel : Element)**
| YAML-Map-Schlüssel['__resource__'] : Resource-Provider,
*(YAML-Map-Schlüssel : (Objekt | Struktur | Konstanter-Ausdruck))**
Resource-Provider := *Name*
Objekt-Referenz := *Name*
Array := *YAML-Array[Array-Objekt]*
Array-Objekt := *(YAML-Map-Schlüssel : Element)**
Struktur := *(YAML-Map-Schlüssel : Konstanter-Ausdruck)** | *YAML-Array[Konstanter-Ausdruck]* | *Konstanter-Ausdruck*
Konstanter-Ausdruck := *Zeichenkette* | *Nummerischer-Ausdruck* | *Boolscher-Ausdruck* | *Konstante*
Zeichenkette := *Name*
Nummerischer-Ausdruck := *\d[\d\s\+\-*\/]**
Boolscher-Ausdruck := *true* | *false*
Konstante := *Name*

6.2.5 Objekterstellungs-Injektion

6.2.6 Referenzierung von Objekten in der Auszeichnungssprache von außerhalb

6.2.7 Typüberladung

6.2.8 Resource-Provider Konzept

6.2.9 Schritte zum kompilieren des Prototypen

Um den Prototypen auszuführen, muss sämtlicher Code kompiliert werden. Dies erfolgt vereinfacht dargestellt in 3 Schritten.

6.2.9.1 Generierung der Funktionalität für Objektorientierung

6.2.9.2 Generierung der Objekt-Erstellungsalgorithmen von YAML

6.2.9.3 Kompilierung des C-Codes inklusive generiertem Code

6.3 Lösungskonzepte bzgl. des Prototyps

6.3.1 Grober Softwareentwurf mittels MVP

6.3.2 Feinentwurf Model

6.3.3 Feinentwurf View

6.3.4 Feinentwurf Presenter

6.4 Implementation

6.4.1 Ausgewählte Basisklassen

6.4.2 Eleganter Aufbau der Hierarchie in YAML

Attribute geschickt sortieren, sodass schöner Aufbau entsteht.

6.4.3 Implementation der jeweiligen Konzepte mit ausgewählten Codebeispielen (Kapitel aufsplitten je Konzept)

→ spezifisch schwerpunkte setzen, bei welchen Konzepten Probleme in der Implementation aufgetreten sind. Wenn alles glatt gelaufen ist eher nicht darauf eingehen.

6.5 Validierung

[Überprüfen, ob Rahmenbedingungen erfüllt worden sind]

7 Open-Source-Aktivitäten und Abschließende Bemerkungen des Projektes

8 Quellenverzeichnis

[1] „GUI Architectures“, *martinfowler.com*. [Online]. Verfügbar unter: https://martinfowler.com/eaaDev/uiArchs.html. [Zugegriffen: 14-Sep-2018].

[2] dotnet-bot, „Data Binding Overview“. [Online]. Verfügbar unter: https://docs.microsoft.com/en-us/dotnet/framework/wpf/data/data-binding-overview. [Zugegriffen: 15-Okt-2018].

[3] „The Early History of Smalltalk“. [Online]. Verfügbar unter: http://gagne.homedns.org/~tgagne/contrib/EarlyHistoryST.html. [Zugegriffen: 11-Okt-2018].

[4] „Organizing Presentation Logic“, *martinfowler.com*. [Online]. Verfügbar unter: https://martinfowler.com/eaaDev/OrganizingPresentations.html. [Zugegriffen: 14-Sep-2018].

[5] „Photo Editing Software - Photo Editor for Online, Mac & PC | Adobe“. [Online]. Verfügbar unter: https://www.adobe.com/products/photoshopfamily.html. [Zugegriffen: 09-Nov-2018].

[6] „Proto.io - Prototypes that feel real“. [Online]. Verfügbar unter: https://proto.io/. [Zugegriffen: 09-Nov-2018].

[7] M. Barr, *Programming Embedded Systems in C and C++*. O’Reilly Media, Inc., 1999.

[8] „Markup Language Definition“. [Online]. Verfügbar unter: https://techterms.com/definition/markup_language. [Zugegriffen: 04-Okt-2018].

[9] „Auszeichnungssprache :: ML (markup language) :: ITWissen.info“. [Online]. Verfügbar unter: https://www.itwissen.info/Auszeichnungssprache-markup-language-ML.html. [Zugegriffen: 04-Okt-2018].

[10] „YAML Ain’t Markup Language (YAML™) Version 1.2“. [Online]. Verfügbar unter: http://yaml.org/spec/1.2/spec.html. [Zugegriffen: 12-Nov-2018].

[11] „XML, JSON und YAML im Vergleich“, *predic8.com*. [Online]. Verfügbar unter: http://www.predic8.de/xml-json-yaml.htm. [Zugegriffen: 12-Nov-2018].

[12] „Trends in der modernen Software-Entwicklung“. [Online]. Verfügbar unter: https://www.microsoft.com/germany/techwiese/know-how/trends-in-der-modernen-software-entwicklung.aspx. [Zugegriffen: 14-Sep-2018].

[13] „CMake“. .

[14] *Contribute to impsapps/impsengine development by creating an account on GitHub*. impsapps, 2018.

[15]: *herb: Expat library: Fast streaming XML parser written in C; in the process of migrating from SourceForge to GitHub: libexpat/libexpat*. Expat development team, 2018.

[16] *SIMD-oriented Fast Mersenne Twister. Contribute to MersenneTwister-Lab/SFMT development by creating an account on GitHub*. MersenneTwister-Lab, 2018.

[17] „SIMD-oriented Fast Mersenne Twister (SFMT)“. [Online]. Verfügbar unter: http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/. [Zugegriffen: 12-Nov-2018].

[18] „OpenCV library“. [Online]. Verfügbar unter: https://opencv.org/. [Zugegriffen: 12-Nov-2018].
[19] „OpenGL - The Industry Standard for High Performance Graphics“. [Online]. Verfügbar unter: https://www.opengl.org/. [Zugegriffen: 12-Nov-2018].
[20] „GLEW: The OpenGL Extension Wrangler Library“. [Online]. Verfügbar unter: http://glew.sourceforge.net/. [Zugegriffen: 12-Nov-2018].
[21] „CLion: A Cross-Platform IDE for C and C++ by JetBrains“, *JetBrains*. [Online]. Verfügbar unter: https://www.jetbrains.com/clion/. [Zugegriffen: 12-Nov-2018].
[22] „MinGW | Minimalist GNU for Windows“. [Online]. Verfügbar unter: http://www.mingw.org/. [Zugegriffen: 12-Nov-2018].
[23] „C99 Standard PDF Entwurf“. [Online]. Verfügbar unter: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n897.pdf. [Zugegriffen: 12-Nov-2018].
[24] „Was ist OOP?“ [Online]. Verfügbar unter: https://www.dev-insider.de/was-ist-oop-a-677737/. [Zugegriffen: 13-Nov-2018].
[25] „pthreads(7) - Linux manual page“. [Online]. Verfügbar unter: http://man7.org/linux/man-pages/man7/pthreads.7.html. [Zugegriffen: 13-Nov-2018].
[26] „<pthread.h>“. [Online]. Verfügbar unter: http://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html. [Zugegriffen: 13-Nov-2018].
[27] „ImpsEngine: Annotations documentation“. [Online]. Verfügbar unter: https://impsapps.github.io/impsengine-doc/d9/d3b/md_docs__annotations-_documentation.html. [Zugegriffen: 13-Nov-2018].

II Abbildungsverzeichnis

III Tabellenverzeichnis

IV Abkürzungsverzeichnis

GUI
YAML
MIT-Lizenz
SFMT
BSD
OOP (objekt orientierte programmierung)