Universität Bremen

Fachbereich 3: Mathematik und Informatik

Bachelorarbeit

3-Dimensionale Darstellung von Codeänderungen in Unity

Florian Garbade

11. Januar 2020

1. Gutachter: Prof. Dr. Rainer Koschke

2. Gutachter: Prof. Dr. Johannes Schöning

Florian Garbade

3-Dimensionale Darstellung von Codeänderungen in Unity

Bachelorarbeit, Fachbereich 3: Mathematik und Informatik

Universität Bremen, Februar 2020

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendete Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Bremen, den 11. Januar 2020

Florian Garbade

Abstract

This thesis deals with the extension of the SEE project of the Software Engineering Group at the University of Bremen. The goal is to illustrate the development of software projects by animating datacities in Unity like a growing city .

Inhaltsverzeichnis

1 Einleitung

Codequalität und -sicherheit spielen eine zentrale Rolle in der Informatik. Einige Programmiersprachen versuchen durch Maßnahmen wie Compiler-Warnungen, diese Aspekte zu integrieren oder sie zu erzwingen. Wenn Programme aufgrund ihrer Aufgabe oder mit der Zeit komplexer werden, wird es für Entwickler zunehmend schwieriger, die Übersicht und ein umfassendes Verständnis dafür zu behalten. Anwendungen zur automatischen Codeanalyse sollen Softwareprojekte nach Fehlern und möglichen Problemen durchsuchen und diese Informationen bereitstellen. Damit unterstützen sie Entwickler dabei, auch in komplexen Projekten die Qualität und Sicherheit umzusetzen.

Das Projekt ‚SEE' der Arbeitsgruppe Softwaretechnik an der Uni Bremen soll Informationen über den Aufbau und die Qualität, die sich aus der Codeanalyse ergeben, visuell in Form einer virtuellen Stadt darstellen. Die 3-Dimensionale Darstellung wird mithilfe der Spiele-Engine ‚Unity' umgesetzt und in der C#-Programmiersprache geschrieben. Visuell stellt SEE eine Stadt mit Bezirken und Häusern dar, die die Ordnerstruktur eines Softwareprojektes widerspiegeln. Dabei werden Ordner durch Kreise, die ihren zugehörigen Inhalt umranden, und Dateien als Häuser mit unterschiedlichen Formen und Größen dargestellt. Verbindungen zwischen zwei Häusern zeigen eine Abhängigkeit im Programmcode auf und Symbole über den Häusern stellen Informationen bezüglich des zugrundeliegenden Codes bereit.

1.1 Aufgabenstellung und Zielsetzung

Das Ziel dieser Arbeit ist es, das Projekt ‚SEE' um Animationen zu erweitern, die den Entwicklungsverlauf eines Softwareprojektes wie in einem Film darstellen. Dabei sollen die Änderungen, wie z. B. neue Dateien, von einer Projektversion zur nächsten Visualisiert werden und ein detaillierteren und nachvollziehbaren Einblick in die Entwicklung eines Projektes geben. Außerdem soll evaluiert werden, ob diese Umsetzung machbar ist und wo die Limitationen dieser Animationen hinsichtlich der Projektgröße liegen.

Dabei sind im Folgenden zwei Aufgaben umzusetzen: Um Daten für die Evaluierung und Tests der Animationen zu erhalten, wird das Projekt ‚LibVCS4j' [Sof20] um eine Export-funktion erweitert, die Informationen für SEE und die folgende Animation exportiert. Dafür wird der Verlauf eines Projektes von einer Versionsverwaltungsplatt-

form wie GitHub geladen und in ein für SEE-City verwendbares Format exportiert. Dabei werden Informationen wie die Anzahl an Codezeilen oder Dateigrößen mithilfe von Spoon[Paw+15], einer Software zur Java-Quellcodeanalyse, extrahiert und in den exportierten Verlaufsdaten abgelegt.

Im zweiten Abschnitt, dem Hauptteil der Arbeit, sollen die exportierten Verlaufsdaten in SEE geladen, aufbereitet und anschließend wie ein Film abgespielt werden. Dabei sollen Aktionen wie das Ändern, Hinzufügen oder Löschen von Dateien für die Betrachter nachvollziehbar hervorgehoben werden. Es ist hierbei nicht nur entscheidend, die Machbarkeit der Animationen zu untersuchen, sondern auch, eine flexible Architektur für eine spätere Weiterentwicklung zu generieren, die weiterentwickelt und -genutzt werden kann. Für das Layout der Daten wird das bereits in SEE existierende Ballon-Layout[1] verwendet.

1.2 Inhaltlicher Aufbau dieser Arbeit

Für ein besseres Verständnis werden in Kapitel 2 zunächst die Grundlagen von Data Cities, LibVCS4j, SEE und Unity erläutert. In Kapitel 3 werden anschließend die Anforderungen und der Aufbau der Erweiterungen erarbeitet und die daraus entstandene Implementierung wird vorgestellt. Neben dem Hauptteil, der sich mit der grafischen Darstellung von Projektverläufen beschäftigt, wird dafür in Kapitel 3.1 zunächst der Export und die Analyse von Codeprojekten behandelt, um Daten für die in 3.2 behandelten Animation zu generieren. Im Kapitel 4 werden anschließend die Ergebnisse der Implementierung und deren Funktionen beschrieben. Kapitel 5 beschäftigt sich mit der Skalierbarkeit und Performance der entworfenen Algorithmen. Zum Schluss wird in der Zusammenfassung und dem Fazit ein Rückblick auf die gesamte Arbeit gegeben, bevor im Ausblick Möglichkeiten aufgezeigt werden, um die Ergebnisse dieser Arbeit weiter zu entwickeln.

[1] Ein Algorithmus der Elemente unterschiedlicher Größe gleichmäßig in Form eines Kreises platziert

2 Grundlagen

In diesem Kapitel werden die Grundlagen der verwendeten Projekte erklärt, angefangen mit LibVCS4j, das in dieser Arbeit für den Export von Projektdaten genutzt wird. Als Einstieg in das SEE-Projekt der Softwaretechnik Arbeitsgruppe(AG) werden Data Cities vorgestellt, bevor anschließend das verwendete Dateiformat GXL betrachtet wird, das zum Speichern der repräsentierten Daten genutzt wird. Außerdem werden die von dem SEE-Projekt verwendete Spiele-Engine ‚Unity' und die Animationsbibliothek ‚ITween' vorgestellt.

2.1 LibVCS4j

Versionskontrollsysteme(VCS) wie Github dokumentieren beliebige Änderungen an Dateien und ermöglichen es, den Verlauf dieser Änderungen nachzuvollziehen. Dadurch können frühere Versionen wiederhergestellt und einzelne Änderungen rückgängig gemacht werden.

LibVCS4j ist eine in Java programmierte Bibliothek, die den Zugriff auf diese Daten von unterschiedlichen Versionskontrollsystemen wie Git oder Subversion über eine einheitliche Schnittstelle ermöglicht. Dabei ist es möglich, den Stand eines Projektes für jede dokumentierte Änderung auszulesen. Zusätzlich werden Informationen bereitgestellt, welche Dateien verändert, verschoben, hinzugefügt oder gelöscht wurden. Um komplexere Informationen wie die Anzahl an Codezeilen oder Kommentare auszulesen, wurde Spoon direkt in LibVCS4j integriert. Spoon ist eine Bibliothek für die Quellcodeanalyse von Java-Anwendungen. Sie transformiert Java-Projekte in ein eigenes Format um komplexere Parameter wie die Anzahl an Klassen und deren Komplexität berechnen zu können.

2.2 Data Cities

Es gibt mehrere Arten von Diagrammen und Graphen, um Informationen darzustellen. Data Cities oder auch Software Maps werden oft dazu genutzt, um Informationen aus Softwareanalysen in einer zwei- oder drei-dimensionalen Kartenansicht darzustellen. Sie geben die Möglichkeit, eine interaktive Benutzerschnittstelle zu schaffen, mit der auch komplexe Zusammenhänge verständlich visualisierent werden können [Lim+19].

Im Folgenden werden zwei Varianten vorgestellt, um Informationen auf mehrere Arten darzustellen.

Abbildung 2.1 Unterschiedliche Detailgrade einer Data City Quelle:[Lim+19]

In Abb. 2.1 wird in vier Stufen gezeigt, wie der Detailgrad in einer Data City durch die Gruppierung nebeneinanderliegender Elemente angepasst und so die Übersicht verändert werden kann. Die einzelnen Komponenten werden dabei durch Blöcke abgebildet, die anhand ihrer Größe, Form und Farbe unterschiedliche Eigenschaften visualisieren. Mit dieser Darstellung kann zum Beispiel ein Dateisystem abgebildet werden. In diesem Fall stellt jeder Block eine Datei dar und Ordner bestehen aus Gruppen von Blöcken. Die Größe eines Block spiegelt z. B. die Dateigröße wieder und die Farbe symbolisiert das Alter einer Datei.

Ein weiteres Beispiel ist in Abb. 2.2 zu sehen. Hier wird gezeigt, dass auch komplexere Informationen durch einzelne Blöcke leicht verständlich aufbereitet werden können. Im oberen Teil wird die Oberflächenstruktur eines Blocks von rau bis hin zu glänzend variiert. So könnten z. B. die Anzahl an Tests dargestellt werden: Ein rauer Block steht dann für wenig bis keine und ein glänzender Block für eine hohe Anzahl an Tests.
Im unteren Teil wird der Block ähnlich wie Metall mit einer rostigen, soliden oder glänzenden Oberfläche dargestellt. Hier kann z. B. Rost symbolisieren, dass ein Block Probleme oder Warnungen aufweist, wohingegen ein neutraler Block eine ausreichende Qualität

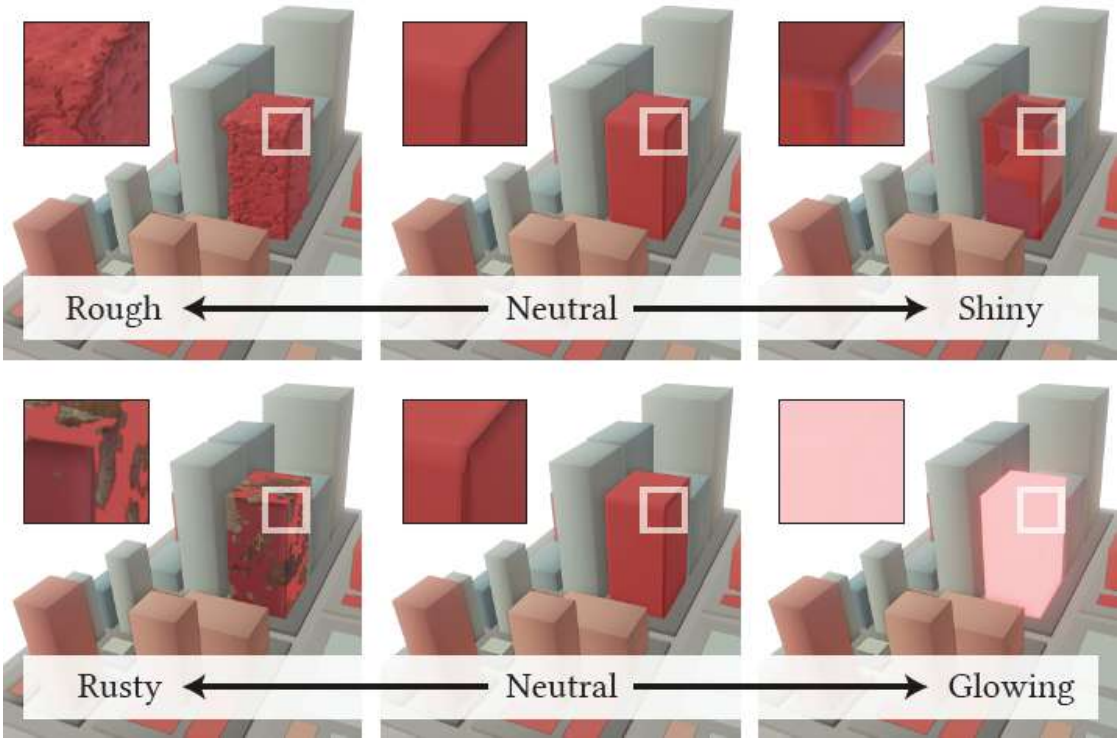

Abbildung 2.2 Oberflächenvarianten in einer Data City Quelle:[Lim+19]

darstellt und ein glänzender Block als überdurchschnittlich gut getestet gilt.

2.3 GXL

GXL oder auch Graph eXchange Language [20b] hat das Ziel, ein standardisiertes Format zum Austausch von Graphen bereitzustellen. Dabei funktioniert es unabhängig von der verwendeten Plattform und Programmiersprache und ermöglicht einen Datenaustausch zwischen unterschiedlichen Tools zur Visualisierung oder zur Analyse.

Im Folgenden werden der Aufbau und die Bedeutung der einzelnen Elemente in einer GCL-Datei vorgestellt, die zur Darstellung im SEE-Projekt verwendet werden.

Jede Datei, die einen GCL-Graphen darstellt, muss ein **gxl**- und ein **graph**-Element Listing 2.1 enthalten, das **graph**-Element bündelt dabei alle weiteren Komponenten. Das *id*-Attribut verweist dabei auf den ersten Knoten in dem Graphen.

```
1  <!DOCTYPE gxl SYSTEM "http://www.gupro.de/GXL/gxl-1.0.dtd">
2  <gxl xmlns:xlink="http://www.w3.org/1999/xlink">
3      <graph id="streamex-root" edgeids="true">
```

```
4           .
5           .
6           .
7       </graph>
8   </gxl>
```

Listing 2.1 GXL-Hauptknoten

Ein Knoten im Graphen wird durch das **Node**-Element definiert und erhält ein eindeutiges **id**-Attribut. In SEE wird mit dem **type**-Element zusätzlich angegeben, ob es sich bei dem Knoten um eine Datei oder einen Ordner handelt Listing 2.2. Im **attr**-Element *Source.Name* ist der Ordner- oder Dateiname abgelegt. Im **attr**-Element *Linkage.Name* wird der eindeutige Ordner- oder Dateipfad innerhalb des Projektes angegeben.

```
1   <node id="N0">
2       <type xlink:href="Directory"/>
3       <attr name="Source.Name">
4           <string>streamex</string>
5       </attr>
6       <attr name="Linkage.Name">
7           <string>streamex</string>
8       </attr>
9   </node>
```

Listing 2.2 GXL-Ordnerknoten

Bei einer Datei siehe Listing 2.3 werden weitere Informationen wie die Anzahl der Codezeilen (*Metric.LOC*) oder die Anzahl an Tokens (*Metric.LOC*) mit zusätzlichen **attr**-Elementen dargestellt. In diesem Fall steht *Metric.LOC* für die Anzahl an Codezeilen in der Datei.

```
1   <node id="N4">
2       <type xlink:href="File"/>
3       <attr name="Metric.Number_of_Tokens">
4           <int>1661</int>
5       </attr>
6       <attr name="Metric.LOC">
7           <int>324</int>
8       </attr>
9       <attr name="Source.Name">
10          <string>LongStreamEx.java</string>
11      </attr>
12      <attr name="Linkage.Name">
```

```
13          <string>src\main\java\javax\util\streamex\LongStreamEx.java<↩
                /string>
14      </attr>
15      <attr name="Source.File">
16          <string>LongStreamEx.java</string>
17      </attr>
18      <attr name="Source.Path">
19          <string>src\main\java\javax\util\streamex\LongStreamEx.java<↩
                /string>
20      </attr>
21  </node>
```

Listing 2.3 GXL-Dateiknoten

Verbindungen zwischen Knoten (dargestellt durch Häuser oder Blöcke in SEE), werden mit dem **edge**-Element angelegt 2.4. Das **type**-Element *Enclosing* bestimmt dabei die Hierarchie, die ähnlich wie eine Baumstruktur organisiert ist. Ein Knoten kann dadurch mehrere Unterknoten enthalten.

```
1   <edge id="E0" from="N1" to="N2">
2       <type xlink:href="Enclosing"/>
3   </edge>
```

Listing 2.4 GXL Ordnerstruktur

Mit dem **type**-Element *Clone* werden Referenzen zwischen zwei Knoten erstellt Listing 2.5. Wenn in einem Java-Projekt eine Klasse auf eine andere Klasse zugreift, wird diese Information mit einem *Clone*-Attribut dargestellt.

```
1   <edge id="E7" from="N1" to="N4">
2       <type xlink:href="Clone"/>
3   </edge>
```

Listing 2.5 GXL Referenzen zwischen Knoten

2.4 SEE

SEE wird an der Universität Bremen in der Arbeitsgruppe Softwaretechnik entwickelt. Das Ziel ist es, Informationen aus der Codeanalyse eines Softwareprojektes in einer 3-dimensionalen Umgebung darzustellen. Dabei werden die Informationen anhand von Data Cities in der Spiele-Engine ‚Unity' dargestellt. Mit der Darstellung als Data City sollen die komplexen Informationen für den Benutzer überschaubar und verständlich visualisiert werden.

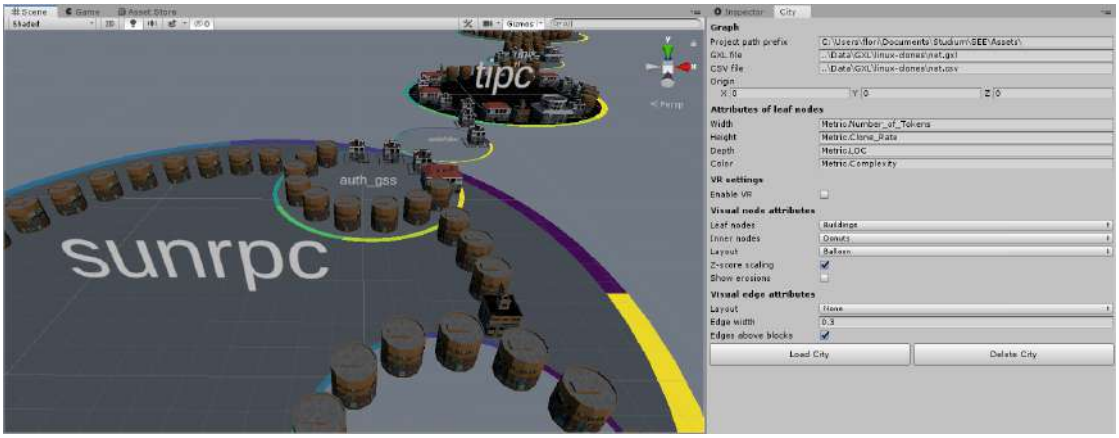

Abbildung 2.3 SEE im Unity-Editor

Um eine Data City zu erzeugen, kann im Unity-Editor (siehe Abb. 2.3) die darzustellende GCL-Datei ausgewählt werden. Zusätzlich kann die Form der einzelnen Elemente und welche Informationen diese darstellen sollen angepasst werden. In der Abbildung wird die Tiefe (hier: Depth) eines Hauses durch die Anzahl an Codezeilen (hier: Metric.LOC) bestimmt.

Abbildung 2.4 Benutzeransicht nach dem Start von SEE

Mit dem Button ‚Load City' wird eine Stadt mit den ausgewählten Einstellungen erzeugt und kann nach dem Start in Echtzeit, wie in Abb. 2.4 zu sehen ist, betrachtet werden. Der Benutzer hat dann die Möglichkeit, sich frei in der Stadt zu bewegen und sie genauer untersuchen.

Neben dieser Arbeit sind noch andere Erweiterungen in Entwicklung. Mit Virtual-Reality-Methoden soll der Nutzer noch detaillierter in die generierte Stadt eintauchen können. Leap Motion[1] soll die Interaktionsmöglichkeiten erweitern, indem die Bewegungen einer Hand von der Fingern in Echtzeit in die digitale Welt übertragen werden, sodass eine neue Form der Interaktion möglich wird. Dabei handelt es sich um ein kleines Gerät, dass Hände optisch tracken kann.

Auch das SEE-Projekt selbst wird weiterentwickelt. Zu Beginn dieser Arbeit wurden die Elemente der dargestellten Data City gleichzeitig direkt nach der Berechnung der jeweiligen Position und Form auf diese gesetzt und angepasst. Da die neuen Werte direkt angewendet wurden, gab es kein Zeitfenster um diesen Wechsel zwischen dem alten und neuen Zustand zu animieren. Daraufhin wurden die Berechnung und Anwendung der Werte getrennt, damit dazwischen zusätzliche Aktionen ausgeführt werden können.

2.5 Unity

Unity ist eine Echtzeit-Entwicklungsplattform für digitale Spiele. Sie kombiniert einen Editor zum direkten Bearbeiten von Spielszenen und die Entwicklungsumgebung Microsoft Visual Studio[2] mit der Möglichkeit, C# für die Erstellung des Programmcodes zu verwenden. Sie ermöglicht es mit einer Codebasis gleichzeitig Spiele für 25 Plattformen zu entwickeln. [20d]

Zu den Plattformen gehören nicht nur bekannte Plattformen wie IOS, Android, Windows, PS4, Xbox One und Linux. Die Spiele lassen sich auch auf Webseiten, Virtual-Reality-Brillen wie die Oculus Rift oder HTC Vive nutzen. Dabei vereint Unity Grafik-Engine, Animationen, Sound, Programmierung und Hilfsmittel wie den Asset-Store, in dem Entwickler eigene Erweiterung oder Bibliotheken kostenlos oder gegen eine Bezahlung einstellen können. Inzwischen findet Unity nicht nur Verwendung in Spielen, sondern wird auch in den Bereichen ‚Filme' und ‚Animation' genutzt und findet mithilfe von Virtual- und Augmented-Reality-Brillen Anwendung im Fertigungs- und Ingenieurswesen.

Intern nutzt Unity sogenannte GameObjects, um Objekte oder Teile wie z. B. ein Auto im Spiel zu repräsentieren. Ein GameObject kann wiederum andere GameObjects enthalten. Ein Mensch-Objekt ist z. B. in einem Auto-Objekt enthalten und orientiert seine Position an der des Autos. Das Verhalten der Spielwelt oder einzelner GameObjects kann durch

[1] https://www.ultraleap.com/
[2] https://visualstudio.microsoft.com/de/

die sogenannte MonoBehaviour bestimmt werden. Dabei handelt es sich um Programme, die an GameObjects gebunden sind und Zugriff auf diese haben. Jedes Mensch-Objekt kann z. B. eine Blinzel-MonoBehaviour enthalten, die in bestimmten Zeitabständen eine Blinzelanimation für die Augen umsetzt.

2.6 ITween

Animationen sind ein zentraler Bestandteil von Unity. Dabei geht es grundlegend darum, einen Wert wie z. B. eine Position über einen gewissen Zeitraum zu einem neuen Wert zu transformieren. Die Bibliothek ITween[3] stellt Funktionen bereit, um unterschiedliche Werte wie die Position, Größe oder Farbe von GameObjects zu animieren.

Um die Bewegung eines GameObject fünf Einheiten nach oben zu animieren, muss lediglich die neue Position (hier: **newPosition** berechnet werden, bevor diese in Verbindung mit dem GameObject mit ITween aufgerufen wird Listing 2.6.

```
1  var newPosition = gameObject.transform.position;
2  newPosition.y += 5;
3  iTween.MoveTo(gameObject, iTween.Hash("position", newPosition));
```

Listing 2.6 ITween: Start einer Animation

Um die Art der Animation anzupassen, können weitere Parameter wie die benötigte Zeit für die Animation übergeben werden.

[3] http://pixelplacement.com/itween

3 Implementierung

Im folgenden Kapitel werden das Design und dessen Implementierung im Detail erklärt. Es wird erläutert, wie die Aufgaben aus Sicht der Programmierung gelöst wurden und welche Faktoren zu den Designentscheidungen geführt haben. Dabei wird zunächst mit dem Datenexport ein Teil der Arbeit fargestellt, bevor im darauffolgenden Abschnitt der Hauptteil bezüglich der Verlaufsanimationen abgebildet wird.

Ein zentraler Punkt für das Design der Erweiterungen liegt in dem übersichtlichen Aufbau mit klar getrennten Aufgabenbereichen. Dadurch sollen besonders die spätere Weiterentwicklung in der AG Softwaretechnik ermöglicht und Änderungen am eigentlichen SEE-Projekt leicht integrierbar werden.

3.1 LibVCS4j-Export

Das erste Ziel ist der Export von GXL-Daten aus Github[1] um im Hauptteil der Arbeit die Animation umsetzen zu können. Da der Kern dieser Thesis die Darstellung der Animation ist, wird der Datenexport möglichst einfach gehalten, sodass der Zeitaufwand für die Implementierung und Anpassungen im Verlauf dieser Arbeit begrenzt wird. Die Benutzeroberfläche des Exports sollte einfach aufgebaut sein und nur notwendige Funktionen bereitstellen. Aus diesem Grund wird auf weitreichende Einstellungsmöglichkeiten verzichtet.

Die Funktionen bauen dabei auf LibVCS4j auf, um Daten über den Verlauf eines Github-Projektes zu erfassen und anschließend als GXL-Dateien für die spätere Animation zu exportieren.

3.1.1 Design

Für den Datenexport ist es entscheidend, dass er mit möglichst wenigen Nutzereingaben initiiert werden kann. Hierfür wird eine grafische Oberfläche angelegt, die durch Eingabe der URL zu einem Github-Projekt, des gewünschten Branch-Namens und der Anzahl zu exportierender Revisionen mit dem Export beginnt und dabei den aktuellen Fortschritt anzeigt. Der Benutzer muss somit nur drei Textboxen ausfüllen und startet mit einem Button das Programm.

[1] https://github.com/

Es wird nur ein Export für Java-Projekte ermöglicht, da aufgrund der Komplexität von Codeanalysen mehrere Sprachoptionen hier den Rahmen dieser Arbeit übersteigen würden. Java lässt sich dabei leicht verwenden, da LibVCS4j mit dem Java-Quellcodeanalysetool Spoon verbunden ist. Weitere Programmiersprachen sollten, ohne Anpassungen des eigentlichen Exports, leicht zu ergänzen sein.

Die erzeugten GXL-Daten sollen jeweils in einer Datei den Projektstand zu einem Zeitpunkt enthalten. Dadurch können die erzeugten GCL-Dateien auch im bisherigen SEE-Projekt verwendet werden und einen einzelnen Zeitpunkt visuell darstellen. Alle Dateien, die nicht analysiert werden können, wie z. B. Konfigurations- oder Binärdateien, werden dabei nicht vom Datenexport erfasst und somit auch nicht in den späteren Animationen dargestellt.

3.1.2 Umsetzung

Zunächst wurde der Datenexport in mehrere Ablaufschritte unterteilt, die auf der Struktur eines Projektes der VCS Github aufbauen.

RepositoryHandler:
Der RepositoryHandler stellt den Einstiegspunkt für den Datenexport dar. Hierdurch wird das ausgewählte Java-Projekt mit LibVCS4j heruntergeladen.Die einzelnen Revisionen werden mit dem RevisionHandler weiter verarbeitet und anschließend entsprechend der Reihenfolge im Projekt nummeriert abgelegt. Die GCL-Dateien enthalten dabei den Projektnamen und die Revision, die die Datei abbildet. Die Dateien werden unter dem folgenden Namen angelegt: projektname-1.gxl, projektname-2.gxl, projektname-3.gxl.

RevisionHandler:
Der RevisionHandler analysiert eine einzelne Revision, wandelt die gesammelten Informationen in das GCL-Format um und speichert sie in der vom RepositoryHandler bestimmten Datei ab. Dabei werden Ordner, die nur einen Unterordner und keine weiteren Ordner oder Dateien enthalten, in GXL als ein Knoten zusammengefasst. So wird der Ordner ‚src', der nur den Ordner ‚main' enthält, in GXL als ein einzelner Knoten ‚src/-main' erfasst, um eine tiefe Verschachtelung der Ordner mit jeweils nur einem Knoten als Inhalt zu vermeiden.

FileAnalyzer:
Innerhalb des RevisionHandler wird für jede Datei ein FileAnalyzer gesucht, der den jeweiligen Dateityp analysieren kann. Aktuell können Java-Quellcodedateien mit dem JavaFileAnalyzer analysiert werden. Die gesammelten Informationen, wie die Anzahl

der Codezeilen werden in der GXL-Datei abgelegt. Für die Analyse weiterer Dateitypen muss ein neuer FileAnalyzer implementiert und im RevisionHandler registriert werden. Der RevisionHandler erkennt anschließend die neuen Dateitypen und nutzt automatisch den registrierten FileAnalyzer.

Ablauf:
Programmintern wird für den Export zunächst eine lokale Kopie von dem ausgewählten Projekt heruntergeladen. Anschließend werden die einzelnen Versionen analysiert, in das GXL-Format umgewandelt und als nummerierte Dateien abgespeichert. Die Dateien können dann später als einzelnen Version oder für die Animation des Projektverlaufs genutzt werden.

Fehler:
Bei dem Export mehrerer Projekte kommt es zu Fehlern bei der Codeanalyse mit Spoon. Dabei kann keine Analyse durchgeführt werden, wenn zwei Dateien mit dem selben Namen in einer Projektversion vorhanden sind. Dieses Problem kann zu jeder Zeit im Exportvorgang auftreten und verhindert dabei den Export von weiteren Versionen.

Die Benutzeroberfläche wird im Kapitel 5 detaillierter vorgestellt.

3.2 Animation

Als nächstes wird das Hauptziel, die Darstellung und Animation von Projektverläufen, erarbeitet. SEE bietet bereits die Möglichkeit, einen einzelnen Graphen darzustellen. Zusammen mit den exportierten Daten sollen jetzt die Änderungen zwischen Graphen bzw. Versionen von Projekten animiert werden, um die Entwicklung von Softwareprojekten nachvollziehbar zu machen.

3.2.1 Design

Für die Animation spielt dabei die Performance eine zentrale Rolle, auf die in Kapitel 5 eingegangen wird. Im Rahmen der Umsetzung soll während der Animation möglichst wenige Berechnungen stattfinden, damit die Animationen nicht ruckeln. Trotz der Einschränkung dieser Arbeit, nur das Ballon-Layout zu nutzen, ist ein modulare Aufbau entscheidend, damit die Animations-Erweiterung in SEE integriert werden kann und weitere Layouts leicht zu ergänzen sind. Hierfür müssen Schnittstellen für einzelne Bereiche angelegt werden, die grundlegende Funktionen bereits implementieren und ausgetauscht oder angepasst werden können.

3.2.2 Umsetzung

Im Folgenden werden die wichtigsten Schnittstellen, die die Animationen und die Darstellung bestimmen, näher erläutert. Sie können je nach Bedarf ausgetauscht oder angepasst werden.

CCALoader:
Der CCALoader übernimmt das Laden des Projektverlaufes. Dafür werden mehrere GXL-Dateien geladen und in ein internes Datenmodell umgewandelt. Die Verlaufsinformationen können dann ohne weitere Verarbeitung abgerufen werden.

MultiScaler:
Um die Eigenschaftswerte wie die Anzahl der Codezeilen einzelner Knoten in einem definierten Bereich zu halten, werden Scaler benutzt. Sie berechnen eine Skalierung, mit der die Eigenschaftswerte innerhalb bestimmter Grenzwerte bleiben. Wenn z. B. drei Knoten mit den Werten 1, 2 und 4 und einen einfachen Scaler mit der Untergrenze 100 und der Obergrenze 1000 vorhanden sind, werden die Werte auf den definierten Bereich skaliert, ohne ihn zu überschreiten. Die Knoten erhalten dann in diesem Fall die skalierten Werte 1 -> 100, 2 -> 400 und 4 -> 1000. Der MultiScaler erzeugt diese Skalierung über eine beliebige Liste von Graphen. Diese Funktion ist notwendig, da unterschiedliche Versionen eines Projekts auch andere Höchstwerte für eine Eigenschaft besitzen können und so bei einer getrennten Skalierung über jede Version bzw. jeden Graphen der gleiche Eigenschaftswert unterschiedlich skalierte Werte annehmen könnte. Dadurch würde ein Knoten mit den selben Werten unterschiedlich dargestellt werden.

CCAObjectManager:
Ein CCAObjectManager beschäftigt sich mit den Erzeugen und Verwalten von GameObjects wie z. B. Häusern die für die Darstellung der Graphen benutzt werden. Damit kann die Abbildung einfach angepasst werden, indem ein anderer CCAObjectManager genutzt wird. Ein HausObjectManager könnte dann z. B. für seine Knoten Häuser erzeugen und ein AutoObjectManager würde Autos anstatt Häuser erstellen. Damit kann für jedes Layout eine eigene visuelle Darstellung angelegt oder es können für ein Layout mehrere Visualisierungen ausprobiert werden. Zusätzlich werden die erzeugten Objekte auch über Versionen hinweg verwaltet, damit sie wiederverwendet werden können und nicht bei jedem Versionswechsel erneut erzeugt werden müssen.

CCALayout:
Das CCALayout berechnet ein Layout wie das BallonLayout und legt die Werte zur späteren Darstellung ab. Damit kann das Layout für alle Versionen im Vorfeld berechnet

werden. Wenn eine Version animiert werden soll, wird das berechnete Layout einfach abgerufen. Dieses Vorgehen vermeidet Berechnungen während den Animationen, die zu Rucklern führen könnten.

CCAAnimator:
Ein CCAAnimator bietet die Funktion, ein GameObject zu einer übergebenen Position und Größe hin zu animieren. Dabei kümmert er sich darum, dass die Animation in einer eingestellten Zeit ablaufen und somit die Animationszeit frei variiert werden kann. Animationen können somit verlangsamt werden, um Änderungen genauer zu verfolgen. Jeder CCAAnimator kann eine eigene Animation implementieren: Ein SchüttelCCAAnimator würde beispielsweise ein Objekt schütteln, wohingegen ein PulsierendeBewegungCCA-Animator ein Objekt während der Bewegung pulsieren lässt. Animationen können dann durch einem anderen CCAAnimator ausgetauscht und an unterschiedliche Darstellungen angepasst werden.

CCARender:
Ein CCARender umfassst das Kernstück der Animationen. Er legt fest, welche Elemente des Graphen in welchen Situationen wie animiert werden. Ein HausCCARender kann, z. B. mit einem Aufruf ein vom ObjectManager erzeugtes Haus mit dem ShakeCCA-Animator schütteln. Gleichzeitig kann er für spezielle Fälle wie ein neu hinzugefügtes Haus eine andere Animation abspielen und so die unterschiedlichen Änderungen mit unterschiedlichen Animationen darstellen.

Ablauf:
Für den gesamten Ablauf werden zunächst die GXL-Dateien mit dem CCALoader geladen und in ein SEE eigenes Graphen-Modell umgewandelt. Anschließend wird die Skalierung mit dem MultiScaler für alle geladenen Graphen berechnet. Die geladenen Graphen und die Skalierung werden dann mit dem CCAObjectManager und dem CCA-Layout zusammen genutzt, um das Layout für jeden Graphen im Vorfeld zu berechnen. Sobald alle Layouts berechnet wurden, wird der erste Graph dargestellt und der Benutzer kann damit interagieren.
Wenn der Benutzer anschließend die Animationen startet, betrachtet der CCARender alle Anpassungen zu dem folgenden Graphen und animiert die unterschiedlichen Änderungen mit einem CCAAnimator zu dem berechneten Zustand. Dieser Schritt wird wiederholt, bis alle Graphen nacheinander abgespielt wurden.

4 Ergebnisse

Im folgenden Kapitel werden die Ergebnisse der Implementierung sowie Fehler, die im späteren Verlauf dieser Arbeit entdeckt und noch nicht gelöst wurden, vorgestellt.

4.1 LibVCS4j-Export

Für den Export von Projektverlaufsdaten ist die Benutzeroberfläche siehe (Abb. 4.1) entstanden. Um hier den Verlauf eines Projektes zu exportieren, wird zunächst die URL zu einem Java-Projekt auf Github in das Feld ‚Repository adress' eingetragen. Daraufhin wird in das Feld ‚Repository-branch' der gewünschte Projekt-Branch ausgewählt, der exportiert werden soll. Als Standartwert wird hier ‚master' gesetzt. Mit dem Feld ‚Number of revisions to load' wird die Anzahl der zu exportierenden Versionen ausgewählt. Enthält das Projekt weniger Versionen, dann werden auch nur diese exportiert. Mit dem Wert, ‚0' werden hier alle verfügbaren Versionen geladen. Mit dem Button, ‚Export data' wird der Export gestartet und aktuelle Statusinformationen oder Fehlermeldungen werden im Info-Textblock angezeigt. Der Ladebalken zeigt dabei den Gesamtfortschritt an.

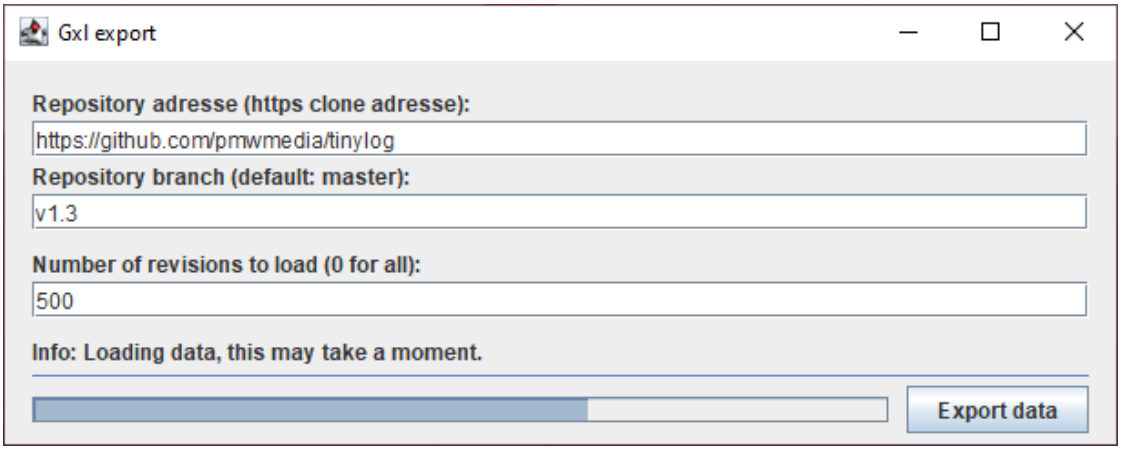

Abbildung 4.1 Programm für den Verlaufsexport

4.2 Animation

Für die Animation von Projektverlaufsdaten wurden zwei Oberflächen entwickelt. Die erste Oberfläche ist eine Erweiterung, die dem Nutzer beim Betrachten der Animationen

weitere Informationen anzeigt (siehe Abb. 4.1). In der oberen linken Ecke werden hier Informationen zur Bedienung angezeigt. Dabei kann mit den Tasten ‚L' und ‚K' in dem Verlauf in der Animation vor- und zurückgespult werden. Mit der Tabulatortaste wird der gesamte Projektverlauf animiert und erneutes Drücken bedingt einer Pause. Darunter steht die Animationszeit, die angibt, wie viele Sekunden die Animation einer Version in Anspruch nimmt. Sie kann mit den Tasten 1 bis 10 nach Bedarf und auch in der laufenden Animation variiert werden.
In der oberen rechten Ecke wird zunächst die aktuell dargestellte Version angezeigt und daneben, durch ein Schrägstrich getrennt, die Anzahl der geladenen Versionen. Darunter zeigt eine Checkbox an, ob die automatische Animation gerade aktiviert ist.

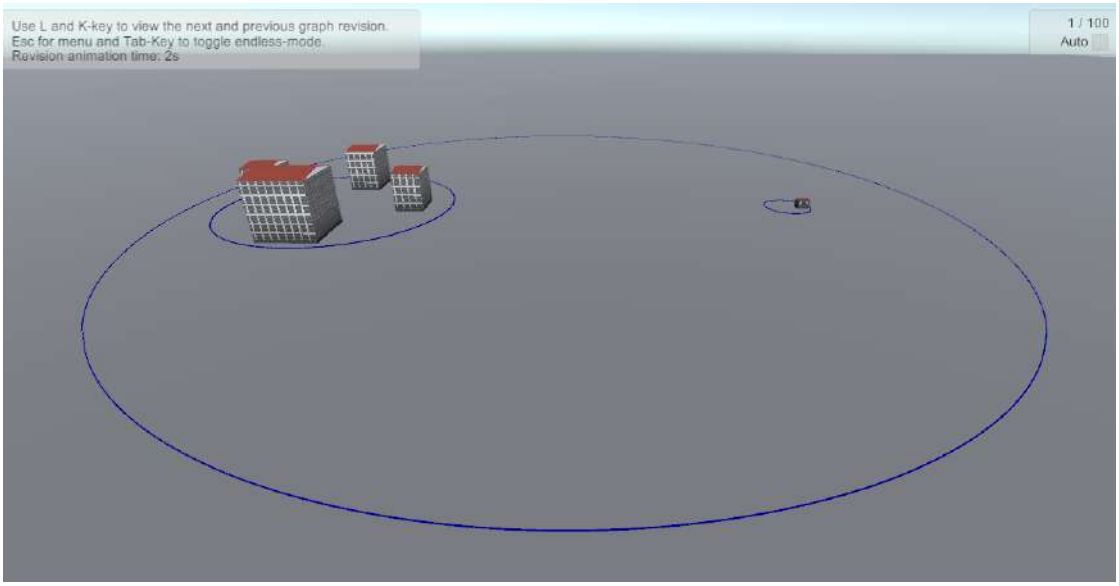

Abbildung 4.2 Ansicht nach dem Start der Animationen

Die zweite Oberfläche (siehe Abb. 4.3) kann jederzeit mit der Escape-Taste aufgerufen werden. Sobald sie geöffnet wird, werden laufende Animationen pausiert. Anschließend kann der Benutzer mit dem Dropdown-Menü in der Mitte zu einer beliebigen Version springen und die Animationen von dort fortführen. Ein erneutes Drücken der Escape-Taste oder des Close-Buttons in der oberen rechten Ecke bringt den Benutzer wieder zur vorherigen Ansicht zurück. Eine zweite Oberfläche ist hier erforderlich, da der Mauszeiger bei der ersten Oberfläche gesperrt und unsichtbar gemacht wird, um seine Bewegungen als Drehung im Spiel zu erfassen. Um das Dropdown-Menü zu nutzen, wird daher der Mauszeiger wieder freigegeben und die Drehung im Spiel blockiert.

Nachdem die automatischen Animationen in Abb. 4.1 mit der Tabulator-Taste gestartet

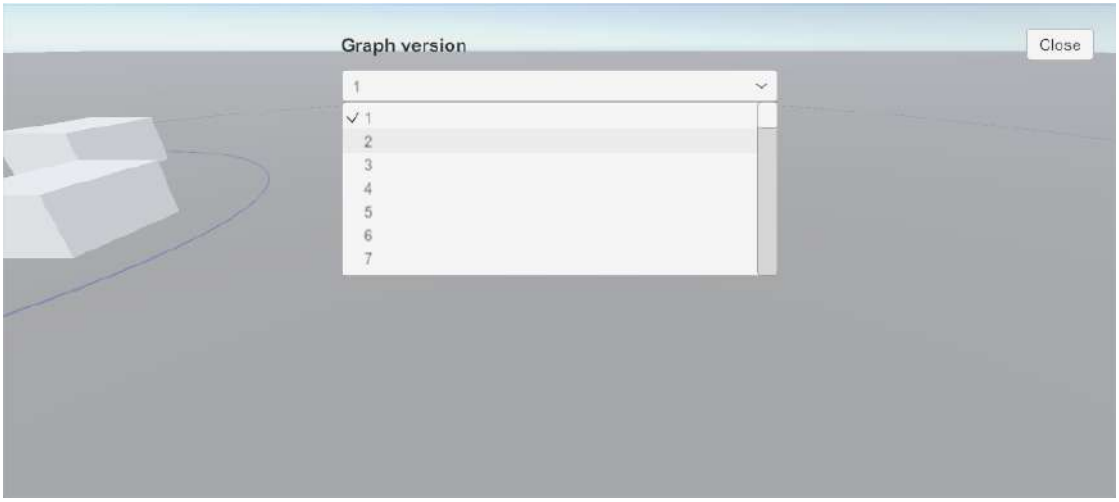

Abbildung 4.3 Ansicht zur Auswahl einer Projektversion

werden, animiert das Programm die Änderungen von einer Projektversion zur nächsten bis die letzte erreicht ist.
Unterschiedliche Änderungen werden durch individuelle Animationen hervorgehoben. Ein neuer Knoten wird von unten durch den Boden zu seine Position bewegt und wächst dabei bis zu seiner berechneten Größe.
Wenn sich der Inhalt eines Knoten ändert, beginnt er damit, sich zu schütteln. Wenn ein Knoten im Verlauf entfernt wurde, schrumpft er unter den visualisierten Boden und verschwindet. Durch neue, entfernte oder geänderte Knoten kann sich das komplette Layout ändern, daher werden die Knoten während der Animation hin zu ihrer neuen Position verschoben. In Abb. 4.4 ist die spätere Version der in Abb. 4.3 dargestellten Verlaufsdaten zu sehen. Erkennbar sind dabei die hinzugekommenen Knoten und die Verkleinerung der gesamten Darstellung.

Bei manchen Projekten kommt es während der Animationen zusätzlich zu einer fehlerhaften Darstellung. Hier überlappen sich wie in Abb. 4.5 zu sehen einige Knoten und ihre Umrandungen. Dafür können fehlerhafte Positionen oder eine falsche Skalierung die Ursache sein
In Abb. 4.6 werden einige Umrandungen so dargestellt, dass sie für den Benutzer nicht sichtbar sind oder die Darstellung wächst mit der Anzahl der Knoten so weit, dass mit dem benutzten Ballon-Layout der Betrachter nicht alle Knoten auf einen Blick erfassen kann. Dadurch können während der Animation nur einzelne Bereiche im Projektverlauf erfasst werden, was eine Gesamtübersicht erschwert. Die Probleme mit den überlappen-

Abbildung 4.4 Ansicht nach dem Start der Animationen

den Knoten und der fehlenden Umrandung konnten innerhalb dieser Arbeit nicht behoben werden und benötigen eine intensivere Auseinandersetzung mit Unity und dessen Funktionen.

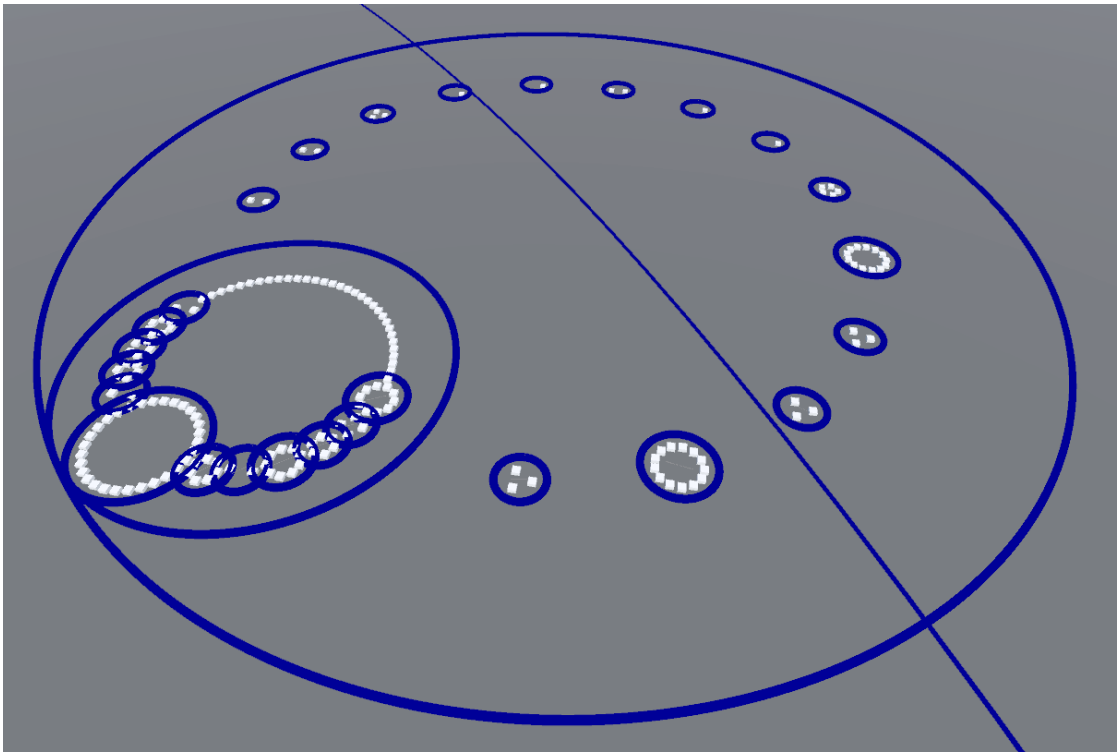

Abbildung 4.5 Überschneidungen im Layout

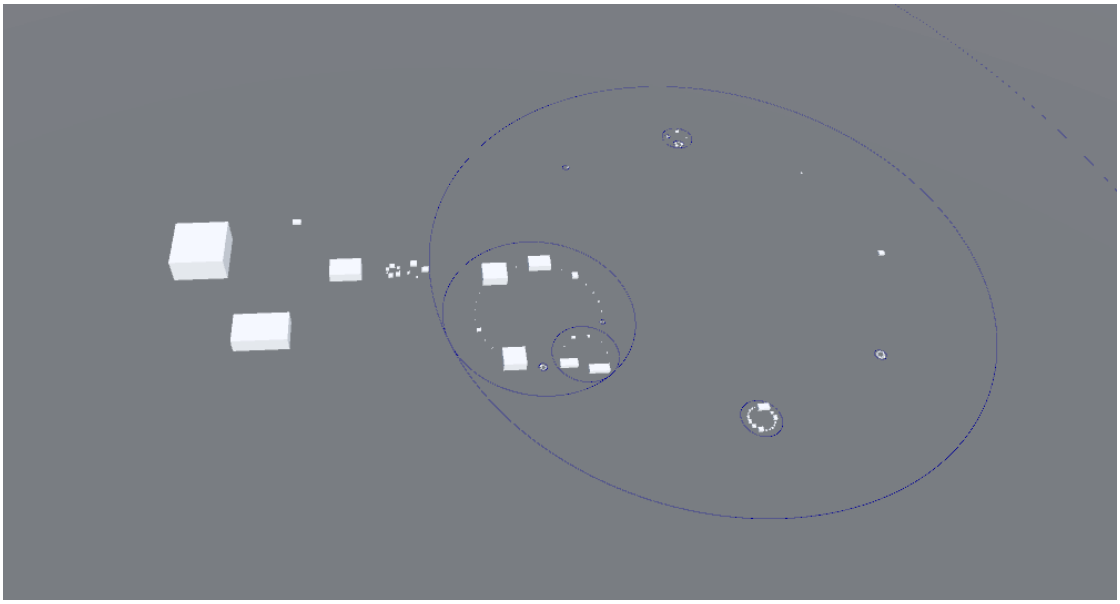

Abbildung 4.6 Fehlende Elemente

5 Evaluation

Nachfolgend sind die Ergebnisse der Evaluation in zwei Bereiche aufgeteilt. Im ersten Abschnitt wird die Performance der geladenen Graphen (siehe Tabelle 5.1) auf die benötigte Zeit zum Layout der Versionen dieser bezogen. Dabei werden die Messergebnisse mit anderen Eigenschaften eines Projektes wie der Anzahl an Dateien verglichen. Mit der Performance des Layouts ist hier die benötigte Rechenzeit gemeint, die zum Positionieren und Erzeugen der GameObjects in Unity benötigt wird.
Im zweiten Teil wird die Framerate während der Animation des Projektverlaufes untersucht und ebenfalls mit den Projekteigenschaften verglichen. Mit der Framerate wird die die Bildwiederholungsrate pro Sekunde(FPS) bezeichnet, die angibt, wie oft die betrachtete Ansicht, hier die Animationen, in jeder Sekunde neu auf dem Bildschirm gezeichnet wird.

Dabei werden zehn Projekte betrachtet, die bereits in der Arbeitsgruppe Softwaretechnik zur Codeanalyse genutzt werden. Jedes Projekt wird zusätzlich jeweils mit Häusern und mit Blöcken für die Darstellung der Knoten untersucht. Die Blöcke stellen für Unity einfache Objekte dar und sollten dabei weniger Rechenleistung benötigen als Häuser mit mehr Details.
Zuletzt werden die gesammelten Daten zusammengefasst sowie Probleme bei der Messung dargestellt.

5.1 Testaufbau

Für den Testaufbau wurden alle Messungen auf dem gleichen Computer durchgeführt. Er ist mit einem Intel-i5-Prozessor, mit vier-Kernen bzw. Threads und einem Basistakt von 3.40 GHz ausgestattet. Daneben sind 16 GB Arbeitsspeicher verbaut, von denen 4 GB im Leerlauf belegt werden, und eine Nvidia GTX 680 Grafikkarte mit 2 GB RAM verbaut. Als Betriebssystem wird Windows 10 benutzt, auf dem Unity in der Version 2019.2.15f1 installiert ist. Vor jeder Messung wurden alle anderen Programme beendet und geprüft, sodass keine rechenintensiven Anwendungen im Hintergrund aktiv sind.

Für die Tests werden nur grundlegende Informationen wie die Anzahl an Codezeilen benutzt, ohne eine detaillierte Codeanalyse durchzuführen. Die Performance der Animation soll nicht durch den Aufwand der Codeanalyse beeinflusst werden.
Um einen Projektverlauf zu untersuchen, werden bei jedem Test 1000 aufeinander folgen-

de Versionen eines Projektes mit einer eingestellten Animationszeit von einer Sekunde für jede Version durchlaufen.

In Tabelle 5.1 werden die verwendeten Projekte aufgelistet, die bereits in der AG Softwaretechnik zur Codeanalyse genutzt werden. Es handelt sich dabei jeweils um Java-Projekte. Das Projekt Dozer ist z. B. eine Mapping Bibliothek, die Daten von einer Java-Klasse in eine andere kopiert, wodurch Datenstrukturen einfacher organisert werden können. Neben ihrem Namen werden auch die Anzahl an Dateien, Ordnern und Codezeilen, die in den 1000 Versionen gezählt wurden, gezeigt. Eine genauere Einsicht, wie viele solcher Komponenten in einer Version enthalten sind, werden in den Stammdaten-Diagrammen im Anhang unter A.5 aufgeführt.

Projekt	**Dateien**	**Ordner**	**LOC**
eureka	89 901	20 492	18 188 890
Agrona	119 104	19 181	21 581 668
cucumber-jvm	120 006	38 741	4 664 528
Aeron	133 712	29 474	18 921 667
aurora	135 808	30 936	22 570 684
avian	152 123	12 890	9 060 921
error-prone	238 627	30 090	19 777 033
drjava	241 371	22 102	57 172 392
cyclops-react	245 630	56 567	29 600 404
dozer	693 124	97 409	54 387 158
eclipse-collections	2 508 900	397 408	366 917 870

Tabelle 5.1 In der Evaluation verwendete Projekte

In Abb. 5.1 werden die Stammdaten von eureka aufgezeigt. Sie umfassen grundlegende Informationen zum Projekt, bezogen auf den Verlauf. Die x-Achse stellt dabei die Version, hier Revision genannt, dar und zeigt, wie sich die Werte im Projektverlauf entwickeln. Auf der y-Achse sind die Anzahl der Dateien mit einer blauen und die Anzahl

der Ordner mit einer orangenfarbene Linie im Graphen dargestellt. Auf der y-Achse wird mit der gelben Linie die Anzahl an Codezeilen verdeutlicht.
Die Stammdaten aller weiteren Testprojekte wurden in der gleichen Form visualisiert und sind in Anhang A.5 zu finden.
In dem Projekt eureka fallen Revisionen auf, in denen im Projektverlauf zum Teil keine Dateien oder Ordner gezählt wurden, wie Abb. 5.1 verdeutlicht. Ob in dem Projekt dabei in einzelnen Revisionen alle Dateien gelöscht wurden oder es beim Export mit LibVCS4j Probleme gab, konnte in dieser Arbeit nicht geklärt werden. Da diese Situation jedoch nur bei eureka vorzufinden ist, wird von einer Anomalie ausgegangen, die bei keinem weiteren Projekt Auswirkungen zeigt und somit kein Einfluss auf die weiteren Messung hat.

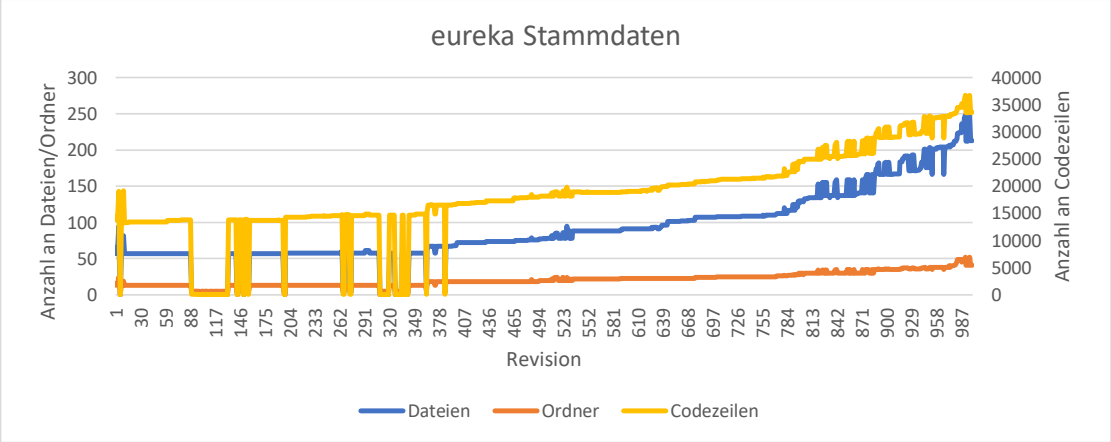

Abbildung 5.1 Stammdaten von eureka

5.2 Layout

Für die Messung des Layout wird die benötigte Zeit für jede Version in einem Projektverlauf einzeln gemessen[20c]. Im Layout werden dabei zuerst die GameObjects in Unity erstellt, bevor ihre Form abhängig von der vorher berechneten Skalierung angepasst wird. Sobald sie die passende Form und Größe haben, werden ihre Positionen berechnet und abgelegt. Damit sind das Layout und die Messung für eine Version abgeschlossen.
Für das Layout der darauf folgenden Version werden bereits existierende GameObjects wiederverwendet oder, falls nötig neu erzeugt. Daraufhin wiederholt sich der eben be-

schrieben Ablauf, bis alle Versionen berechnet wurden.

In Abb. 5.2 werden die Messergebnisse des Layouts von eureka dargestellt. Dabei werden die Werte aus Unity von den Messungen mit Blöcken und von denen mit Häusern für einen Vergleich in einem Graphen dargestellt. Die x-Achse stellt dabei die Version, hier ebenfalls Revision genannt, dar. Auf der y-Achse wird die benötigte Rechenzeit in Millisekunden abgetragen. Für Blöcke wird eine blaue, für Häuser eine orangenfarbene Linie benutzt.
Die Layout-Zeiten aller weiteren Testprojekte wurden in der gleichen Form visualisiert und sind in Anhang A.5 zu finden. In Tabelle 5.2 werden zusätzlich die Mittelwerte der Layout-Zeiten für Blöcke und Häuser bezogen auf das jeweilige Projekt aufgelistet.

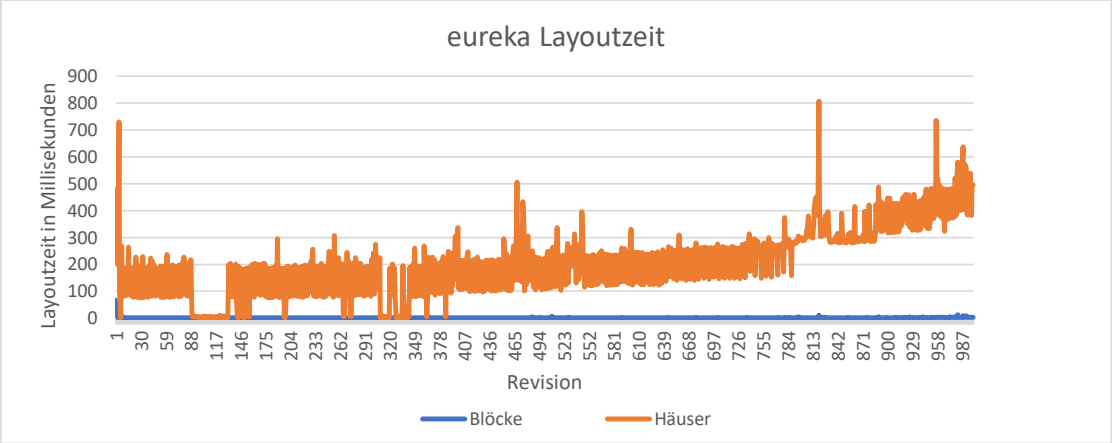

Abbildung 5.2 Rechenzeit für das Layout von eureka

Auswertung:
In den Messungen zeigt sich, dass mit der steigenden Anzahl an Dateien und Ordnen auch die benötigte Rechenzeit zunimmt. Dies war bereits zu erwarten, da mit mehr Komponenten auch die Anzahl der zu erzeugenden GameObjects in Unity steigt.
Dabei zeigt der große Zeitunterschied zwischen Häusern und Blöcken bei dem Layout, dass Unity weitaus mehr Zeit benötigt, um komplexere GameObjects wie die verwendeten Häuser zu erzeugen. Für das Layout mit Blöcken fällt dabei nur ein Bruchteil der Zeit an. Bei dem Projekt dozer mit fast 700 000 Dateien und 100 000 Ordnern benötigt das Layout mit Häusern etwa 150 mal mehr Zeit als mit Blöcken.

Im Vergleich dazu beeinflussen die Anzahl an Codezeilen(LOC) die Rechenzeit weniger. Sie wachsen in mehreren Projekten zwar ähnlich zu den Dateien, steigen jedoch an manchen Stellen stärker an als Dateien oder Ordner, ohne einen signifikanten Einfluss auf die Layout-Zeit zu haben. Diese Beobachtung wird in Abb. A.5 und A.6 deutlicher. Hier steigen die LOC ab Revision 800 stärker an als die Dateien und Ordner. Die Layout-Zeit orientiert sich dabei jedoch eher an dem Zuwachs der Dateien und Ordner. Ähnlich verhält es sich in Abb. A.17, in der ab Revision 800 die LOC spontan ansteigen, ohne einen wahrnehmbaren Einfluss auf die Layout-Zeit in Abb. A.18 zu nehmen.

Weiterhin fällt auf, dass die Layout-Zeit bei den Häusern im gesamten Verlauf zwischen den Versionen Schwankungen ausgesetzt ist, dabei wurde hier durch den konstanten Anstieg an Dateien und Ordnern ein ebenso konstanter Anstieg der Layout-Zeit erwartet.

Projekt	**Dateien**	**Ordner**	**LOC**	**Layout-Zeit Blöcke**	**Layout-Zeit Häuser**
eureka	89 901	20 492	18 188 890	1,29	210,41
Agrona	119 104	19 181	21 581 668	1,16	245,96
cucumber-jvm	120 006	38 741	4 664 528	2,15	202,28
Aeron	133 712	29 474	18 921 667	1,42	296,99
aurora	135 808	30 936	22 570 684	1,42	320,82
avian	152 123	12 890	9 060 921	1,28	329,51
error-prone	238 627	30 090	19 777 033	2,56	503,63
drjava	241 371	22 102	57 172 392	2,63	550,25
cyclops-react	245 630	56 567	29 600 404	3,19	602,01
dozer	693 124	97 409	54 387 158	11,12	1645,52

Tabelle 5.2 Zusammenfassung der erfassten Layout-Zeiten in Sekunden

5.3 Framerate

Für die Messung der Framerate werden bei der Animation jeder Version die kleinste, durchschnittliche und höchste FPS gemessen[20a]. Dafür werden 1000 Versionen im Automatikmodus [1] mit einer Animationszeit von 0,5 Sekunden abgespielt. Vor dem Start der Animationen werden fehlende GameObjects erzeugt und nicht mehr benötigte entfernt. Die restlichen werden wiederverwendet. Dieser Ablauf wiederholt sich anschließend, bis alle Versionen animiert wurden.

In Abb. 5.3 und Abb. 5.4 werden die gemessenen FPS während der Animation dargestellt. In beiden Diagrammen stellt die x-Achse die Version, hier Revision genannt, dar. Auf der y-Achse werden die gemessenen FPS gezeigt. Für den niedrigsten Wert wird eine blaue, für den durchschnittlichen Wert einer Version eine orangefarbene und für den höchsten gemessenen Wert eine graue Linie verwendet. Bei allen weiteren Testprojekten werden die FPS in der gleichen Diagrammdorm dargestellt. Sie sind in Anhang A.5 zu finden. In Tabelle 5.3 werden die Mittelwerte der Ergebnisse bezogen auf das jeweilige Projekt aufgelistet. Werte mit ‚B:' stehen für FPS mit Blöcken, wohingegen solche mit ‚H:' Häuser implizieren. ‚Min', ‚Mittel' und ‚Max' stehen für den niedrigsten, durchschnittlichen und höchsten FPS-Wert.

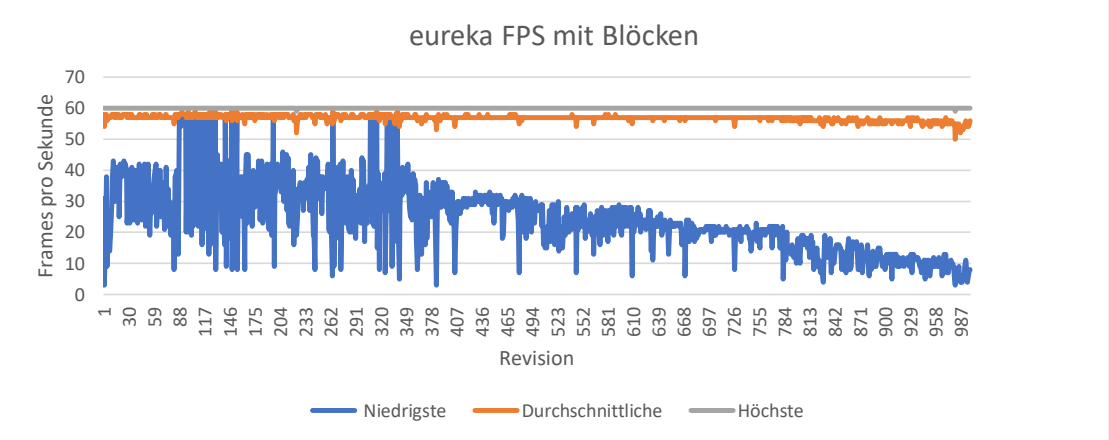

Abbildung 5.3 Framerate mit Blöcken für eureka

[1] Im Automatikmodus wird nach der Animation einer Version direkt die nächste Version animiert, bis es keine nächste mehr gibt.

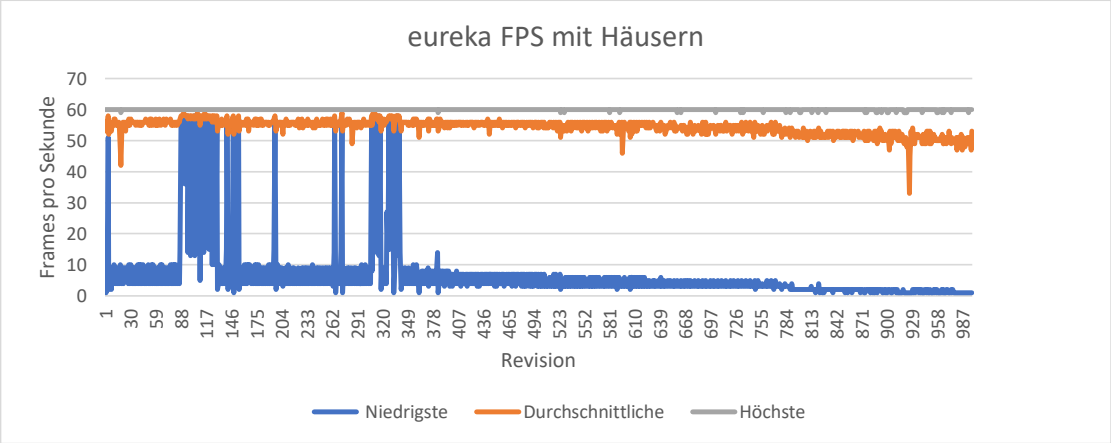

Abbildung 5.4 Framerate mit Häusern für eureka

Auswertung:
Die Resultate zeigen, dass die Messwerte für die durchschnittlichen FPS nahe an den Höchstwerten liegen. Dies zeigt, dass niedrigere FPS seltener auftreten und dadurch der Durchschnitt höher ausfällt. Dieses Verhalten passt zu der Implementierung, da hier am Anfang der Animationen die notwendigen GameObjects erzeugt werden und dadurch der Rechenaufwand zu dieser Zeit steigt. Beim Abspielen der Animationen fällt analog dazu auf, dass beim Start der Animationen einer Version ein kurzes Ruckeln auftritt.

Im Vergleich zu der steigenden Anzahl an Dateien und Ordnern bleiben die Werte der durchschnittlichen und maximalen FPS-Rate nahezu konstant zwischen 50 und 60 FPS. Die niedrigsten FPS dagegen sinken durch die steigende Anzahl an Dateien und Ordnern auf Werte nahe Null wie in Abb. A.19 und Abb. A.20 zu sehen ist. Dabei fallen die FPS bei Häusern bereits ab Revision 150 auf nahe Null, wohingegen die Animationen für Blöcke selbst bei der letzten Revision noch bei 10 FPS liegen.

Erst in den Testprojekten cyclops-react und dozer fallen die durchschnittlichen und maximalen FPS-Werte ab Revision 750 und 500 unter 50 FPS und unterliegen vor allem bei der Verwendung von Häusern starken Schwankungen.

Projekt	**Dateien**	**Ordner**	**H: Min**	**H: Mittel**	**H: Max**	**B: Min**	**B: Mittel**	**B: Max**
eureka	89 901	20 492	7	54	60	24	57	60
Agrona	119 104	19 181	3	54	60	17	56	60
cucumber-jvm	120 006	38 741	6	53	60	19	56	60
Aeron	133 712	29 474	4	53	60	14	55	60
aurora	135 808	30 936	7	53	60	3	53	60
avian	152 123	12 890	7	53	60	20	56	60
error-prone	238 627	30 090	2	50	60	11	54	60
drjava	241 371	22 102	2	50	60	13	55	60
cyclops-react	245 630	56 567	3	44	55	10	27	31
dozer	693 124	97 409	0	17	28	3	33	50

Tabelle 5.3 Zusammenfassung der erfassten Frameraten

5.4 Ergebnisse der Evaluation

Auf Basis der Messungen und den entsprechenden Resultaten lassen sich mehrere Rückschlüsse ziehen. Zunächst ist deutlich geworden, dass die Implementierung nachvollziehbare Ergebnisse für die Animationen liefert. Die Layout-Zeit und Frameraten zeigen dabei die bisherige Performance und die Möglichkeiten, auch größere Projekte wie dozer zu animieren. Besonders die unterschiedlichen Messungen für Blöcke und Häuser zeigen, dass die Implementierung noch Potential zur Weiterentwicklung hat. Hier kann eine vereinfachte Darstellung der Häuser und Optimierungen im Zusammenhang mit Unity die Performance von Häusern an die von Blöcken angleichen.

Die Projekte dozer und eclipse-collections stellen dabei auch die Grenzen in Bezug auf die Anzahl an Komponenten (Dateien und Ordner) der aktuellen Implementierung dar. Hier brechen im Verlauf von dozer die FPS ein, wodurch eine gleichmäßige Animation nicht möglich ist. Das Projekt eclipse-collections kann bereits nicht mehr geladen werden.

Unity hängt sich hier beim Layout auf und belegt dabei den kompletten Arbeitsspeicher.

6 Zusammenfassung und Fazit

Als Abschluss der entworfenen Arbeit werden die Ergebnisse noch einmal in Kapitel 6.1 dargestellt. In Abschnitt 6.2 werden anschließend Möglichkeiten zur Erweiterung und Verbesserung dieser Arbeit vorgestellt.

6.1 Rückblick

In dieser Arbeit wurden die Grundlagen von Unity und Data Cities erläutert und das SEE-Projekt wurde um Animationen erweitert, die den Verlauf eines Softwareprojektes ähnlich wie in einem Film abspielen können. Bei der Implementierung gab es dabei mehrere Hindernisse, die aufgrund der fehlenden Erfahrung mit Unity mehr Zeit in Anspruch genommen haben als im Vorfeld erwartet. Zusätzlich wurde auch ein Programm entwickelt, dass Projektverlaufsdaten aus Java-Projekten von Github exportiert und diese für die weitere Animation oder zur Anzeige in SEE abspeichert. Der Aufbau der Implementierung macht es dabei einfach, weitere Darstellungsformen, Animationsarten und Programmiersprachen hinzuzufügen und deren Leistungsdaten zu evaluieren.

Während der Evaluation wurden auch die Grenzen des verwendeten Ballon-Layouts deutlich und wie sie die Übersicht der Darstellung negativ beeinträchtigen haben. Anhand der Ergebnisse der Evaluation wurde gezeigt, dass das SEE-Projekt dafür geeignet ist, mit Animationen und weiteren Funktionen die Veranschaulichung von Daten aus der Codeanalyse zu verbessern. Die Resultate verdeutlichen auch, dass der erste Kontakt mit Unity und die damit einhergehende, fehlende Erfahrung maßgeblich die Implementierung und die entsprechenden Leistungswerte beeinflusst.

6.2 Ausblick

Da das SEE-Projekt aktiv weiterentwickelt wird, wäre es sinnvoll, die Animation um weitere SEE-Funktionen zu erweitern. Dazu gehört die Visualisierung von Code-Smells und die Möglichkeit, Einstellungen für die Animationen über eine Benutzeroberfläche konfigurieren zu können. Dabei ist es auch unerlässlich, die Performance zu betrachten und sie gegebenenfalls zu verbessern, damit auch mehr und komplexere Informationen darstellbar werden. Dafür bietet sich eine detaillierte Untersuchung der Animation an, die überprüft, an welchen Stellen zeitintensive Prozesse ablaufen und wie diese optimiert werden können.

Bei der Evaluation wurde bereits die Berechnung des Layout als rechenintensiver Prozess erkannt. Hier könnte es eine Trennung von der Layout-Berechnung und der Erstellung von GameObjects den Nutzern erlauben, das Layout im Vorfeld zu berechnen und das Resultat in einer Datei abzuspeichern. Zu Laufzeit müssten dann nur noch die GameObjects erzeugt und in ihre gewünschte Form transformiert werden.

Für die Performance wäre auch die Implementierung und Evaluierung anderer Layouts hilfreich. Diese Arbeit nutzt das Ballon-Layout, das anschließend mit den Ergebnissen weiterer Layouts verglichen werden könnte, um den Einfluss dieser Eigenschaften auf die Performance zu bewerten.

A Anhang

A.1 Abbildungsverzeichnis

A.2 Tabellenverzeichnis

A.3 Quellcodeverzeichnis

A.4 Abkürzungsverzeichnis

FPS Bildwiederholungsrate pro Sekunde
AG Arbeitsgruppte
VCS Versionskontrollsystem
LOC Anzahl an Codezeilen

A.5 Literatur

[20a] *FPS messen in Unity*. *https://answers.unity.com/questions/64331/accurate-frames-per-second-count.html*. 2020 (Stand 02.02.2020).

[20b] *Graph eXchange Language*. *http://www.gupro.de/GXL/index.html*. 2020 (Stand 02.02.2020).

[20c] *Rechenzeit messen in Unity*. *http://blog.ctrlxctrlv.net/en/unity-function-execution-time/*. 2020 (Stand 02.02.2020).

[20d] *Unity Gameengine*. *https://unity.com/de*. 2020 (Stand 02.02.2020).

[Lim+19] Daniel Limberger, Willy Scheibel, Jürgen Döllner und Matthias Trapp. »Advanced Visual Metaphors and Techniques for Software Maps«. In: Sep. 2019. ISBN: 978-1-4503-7626-6. DOI: *10.1145/3356422.3356444*.

[Paw+15] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera und Lionel Seinturier. »Spoon: A Library for Implementing Analyses and Transformations of Java Source Code«. In: *Software: Practice and Experience* 46 (2015), S. 1155–1179. DOI: *10.1002/spe.2346*. URL: *https://hal.archives-ouvertes.fr/hal-01078532/document*.

[Sof20] Universität Bremen Arbeitsgruppe Softwaretechnik. *LibVCS4j*. *https://github.com/uni-bremen-agst/libvcs4j*. 2020 (Stand 02.02.2020).

A.6 Evaluation Graphen

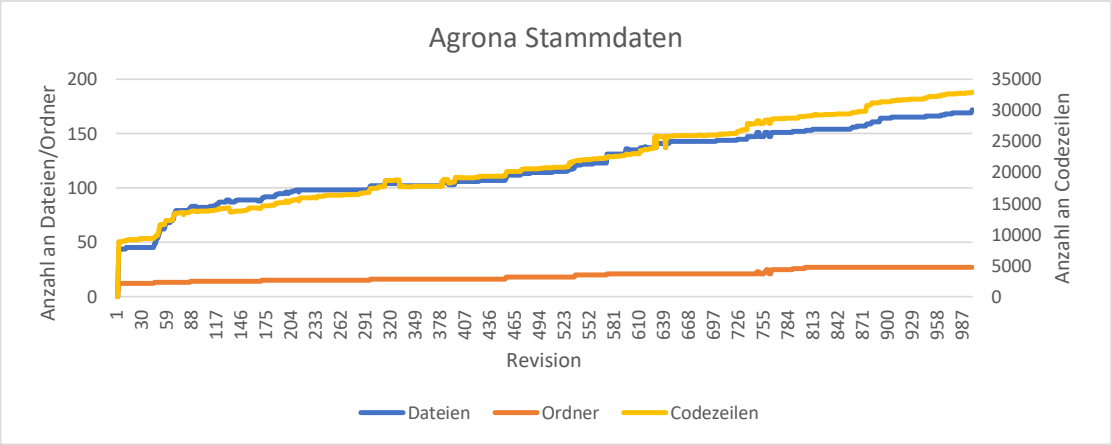

Abbildung A.1 Stammdaten von Agrona

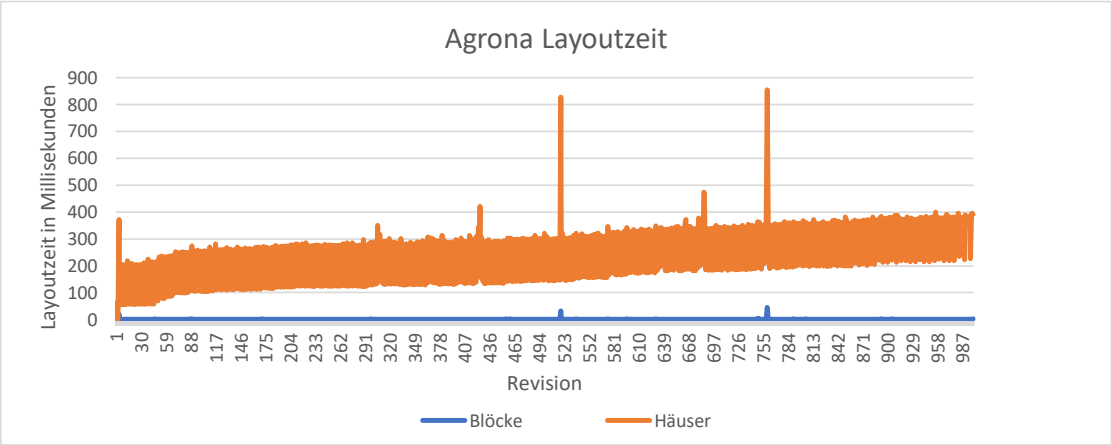

Abbildung A.2 Rechenzeit für das Layout von Agrona

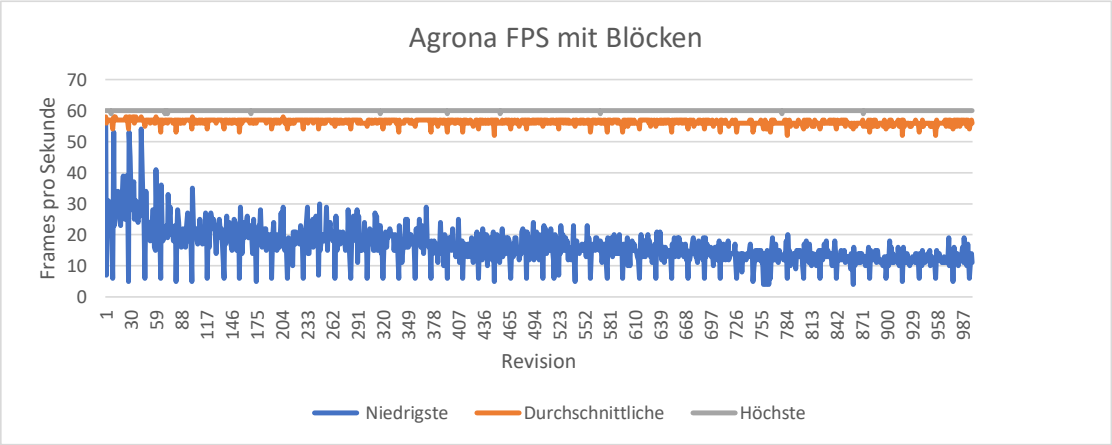

Abbildung A.3 Framerate mit Blöcken für Agrona

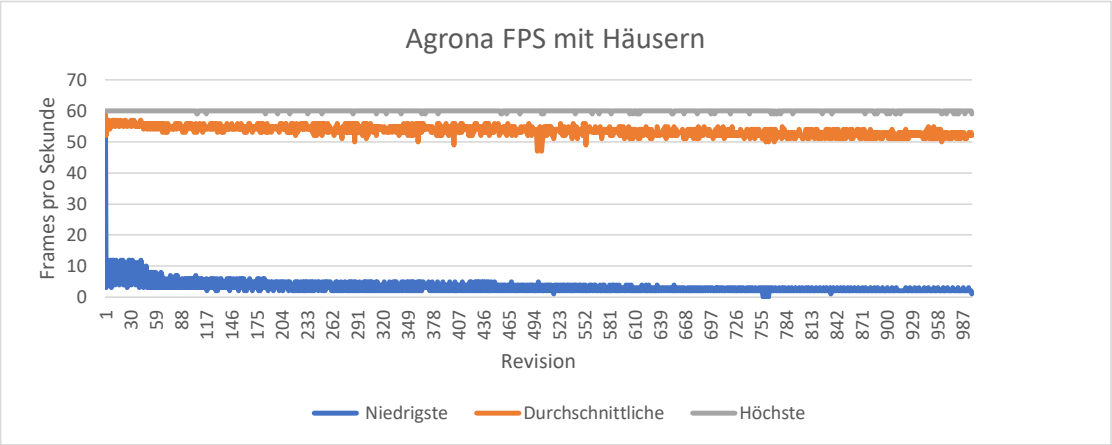

Abbildung A.4 Framerate mit Häusern für Agrona

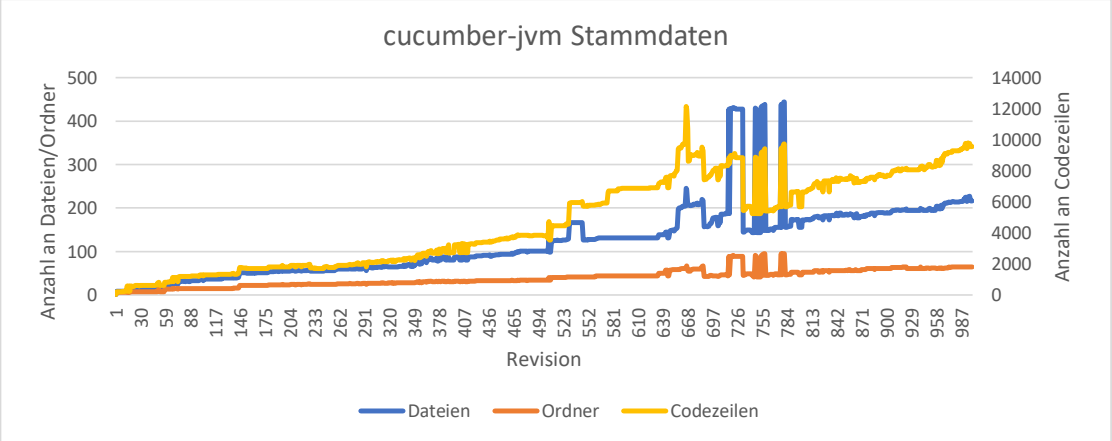

Abbildung A.5 Stammdaten von cucumber-jvm

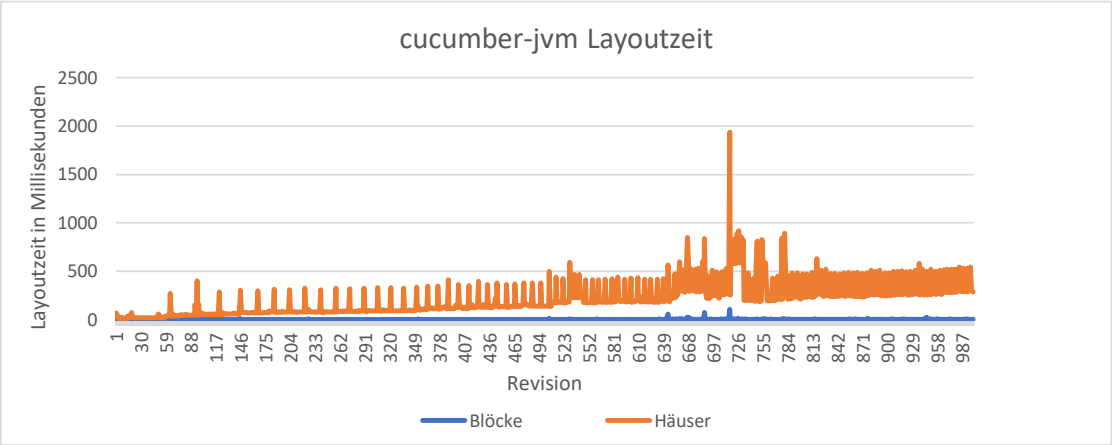

Abbildung A.6 Rechenzeit für das Layout von cucumber-jvm

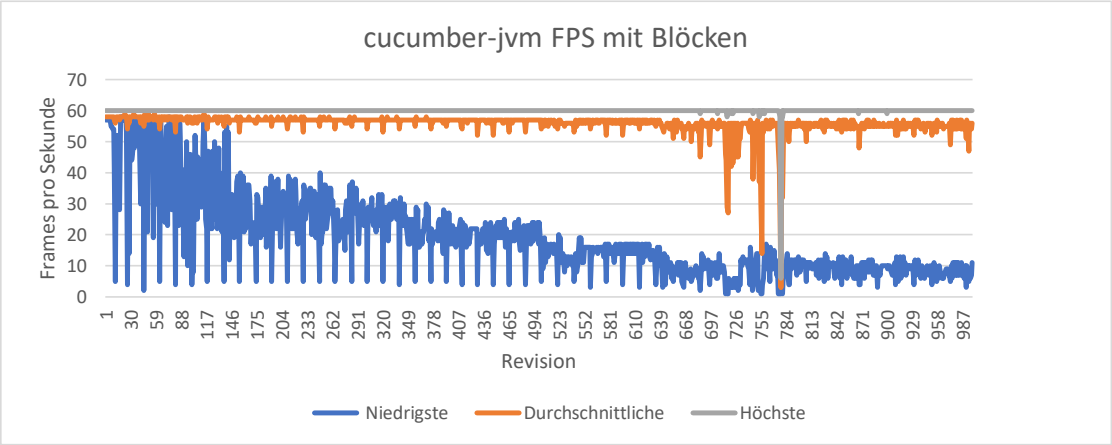

Abbildung A.7 Framerate mit Blöcken für cucumber-jvm

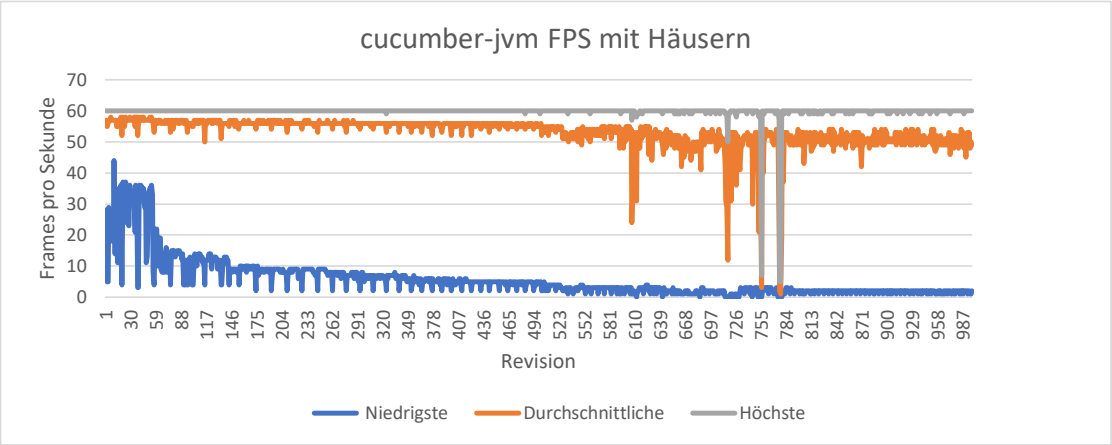

Abbildung A.8 Framerate mit Häusern für cucumber-jvm

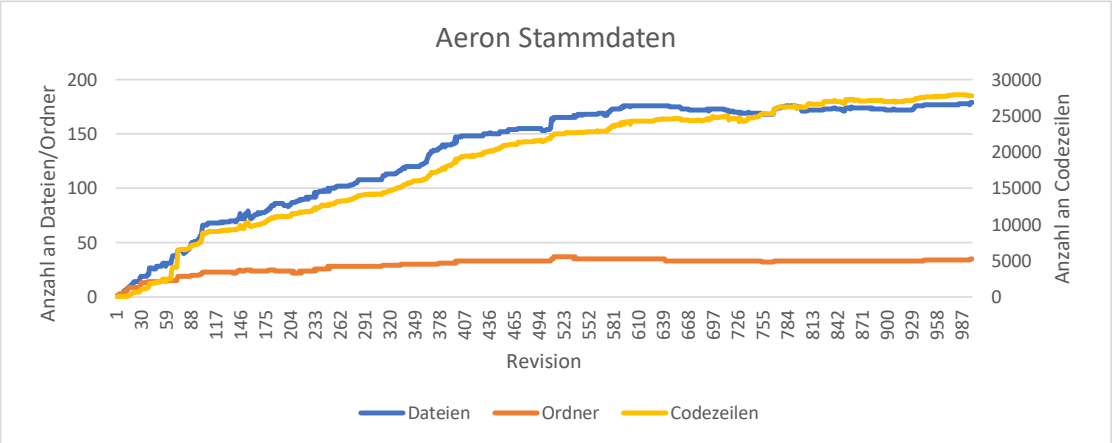

Abbildung A.9 Stammdaten von Aeron

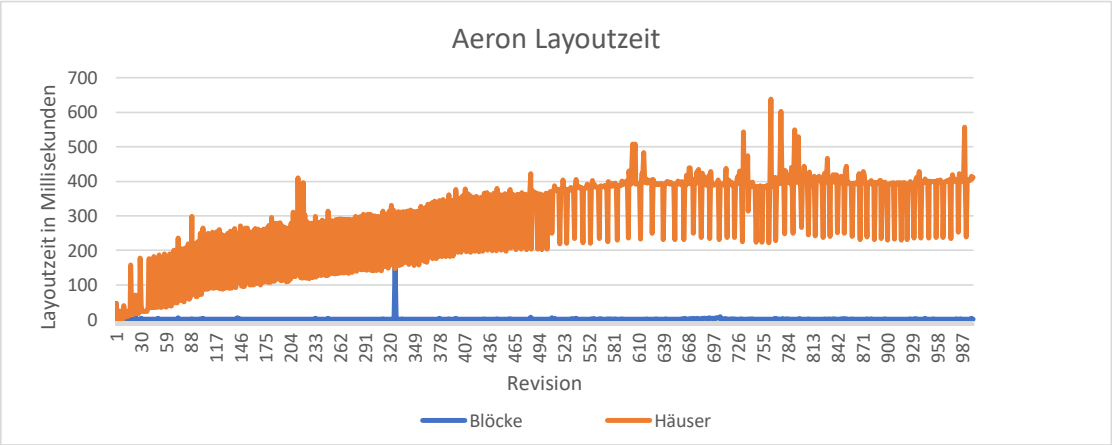

Abbildung A.10 Rechenzeit für das Layout von Aeron

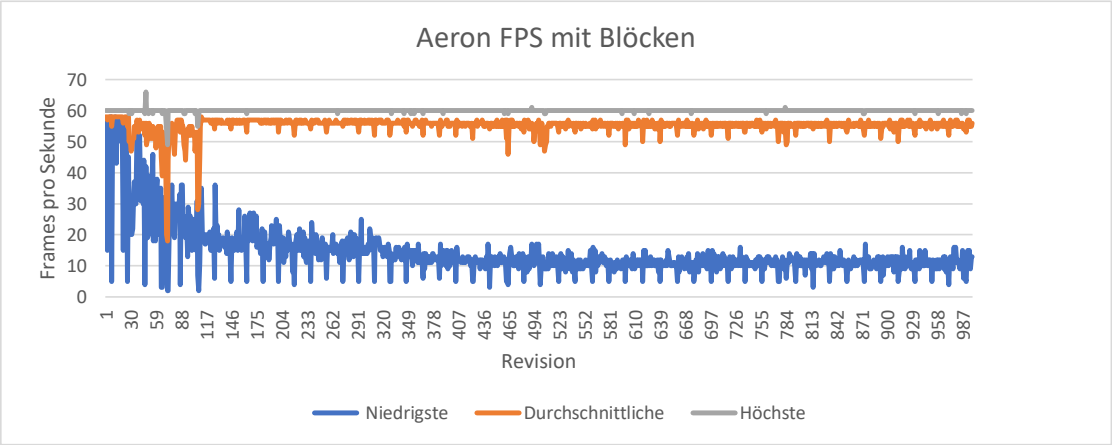

Abbildung A.11 Framerate mit Blöcken für Aeron

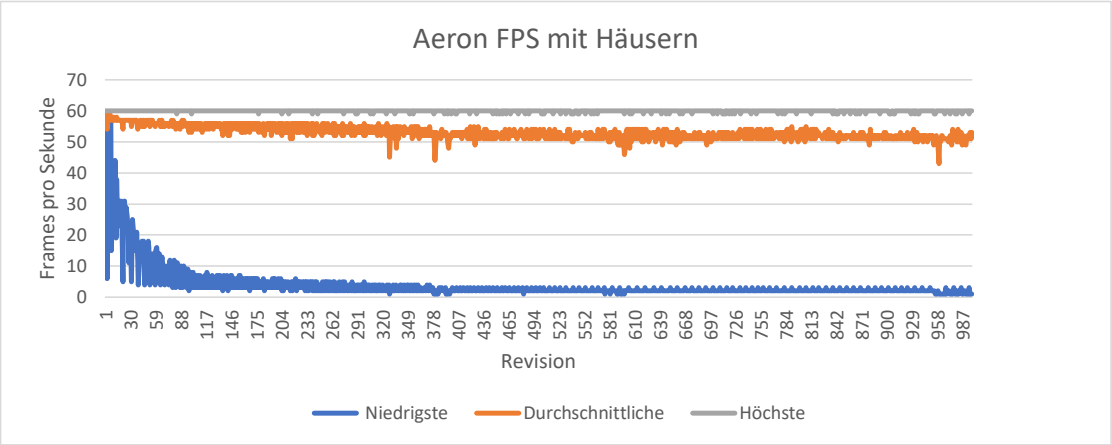

Abbildung A.12 Framerate mit Häusern für Aeron

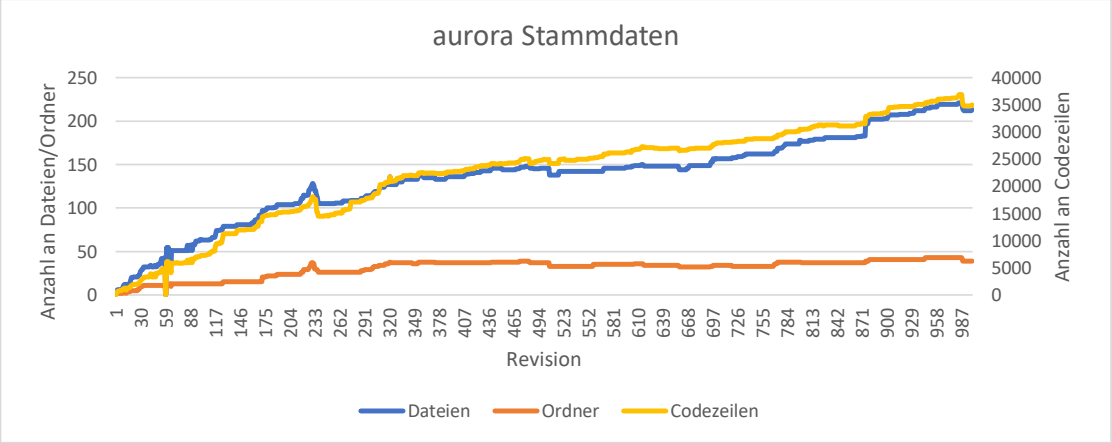

Abbildung A.13 Stammdaten von aurora

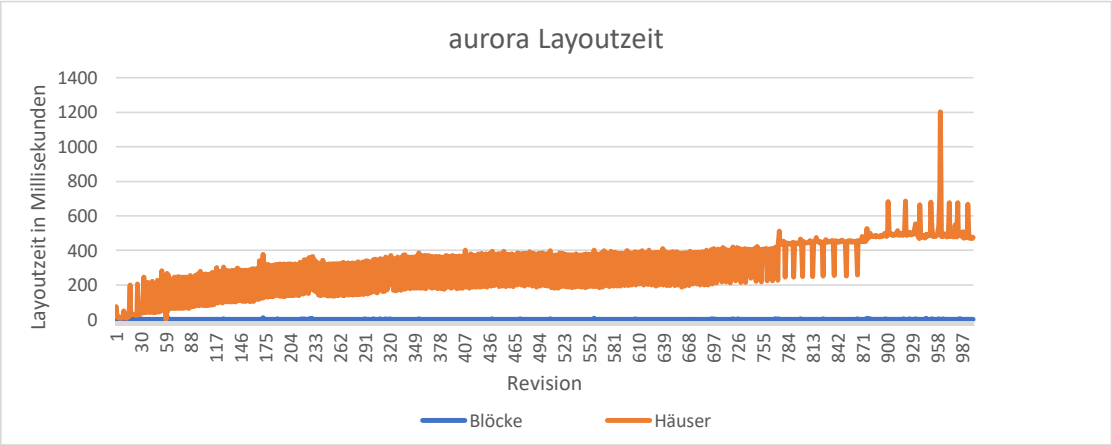

Abbildung A.14 Rechenzeit für das Layout von aurora

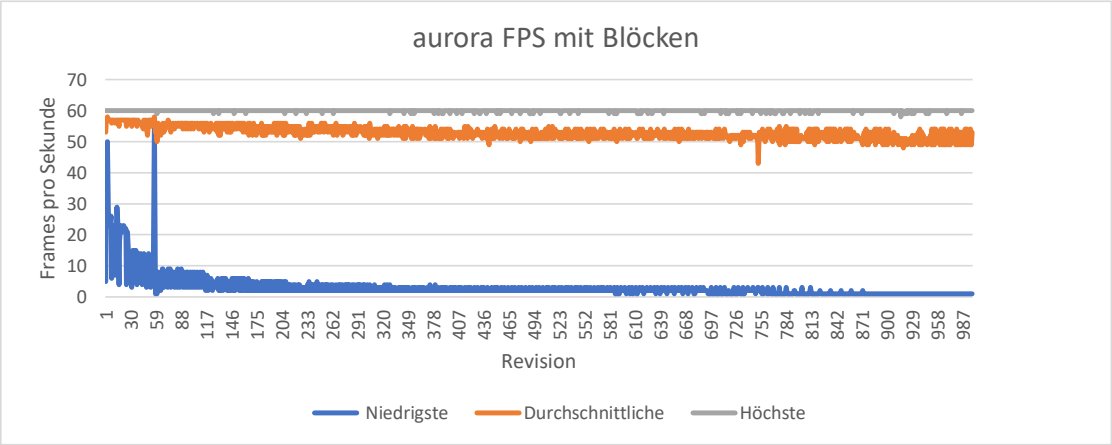

Abbildung A.15 Framerate mit Blöcken für aurora

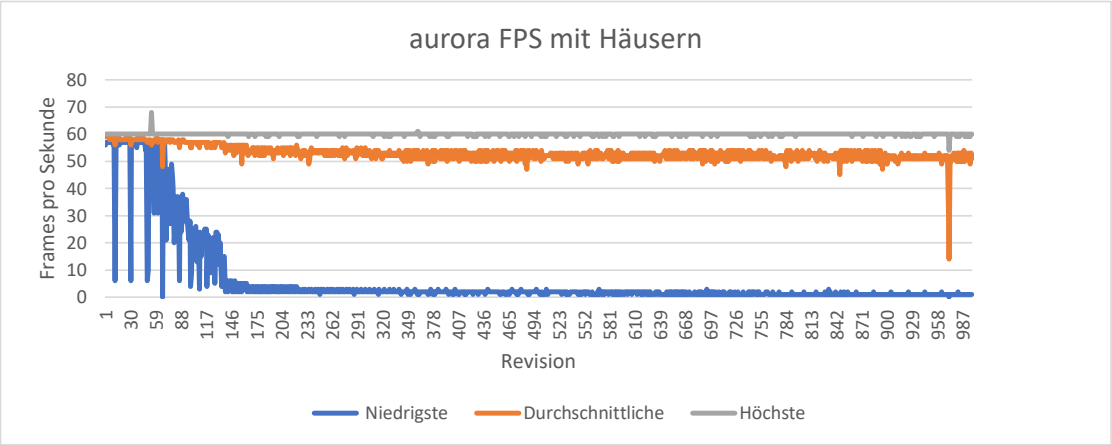

Abbildung A.16 Framerate mit Häusern für aurora

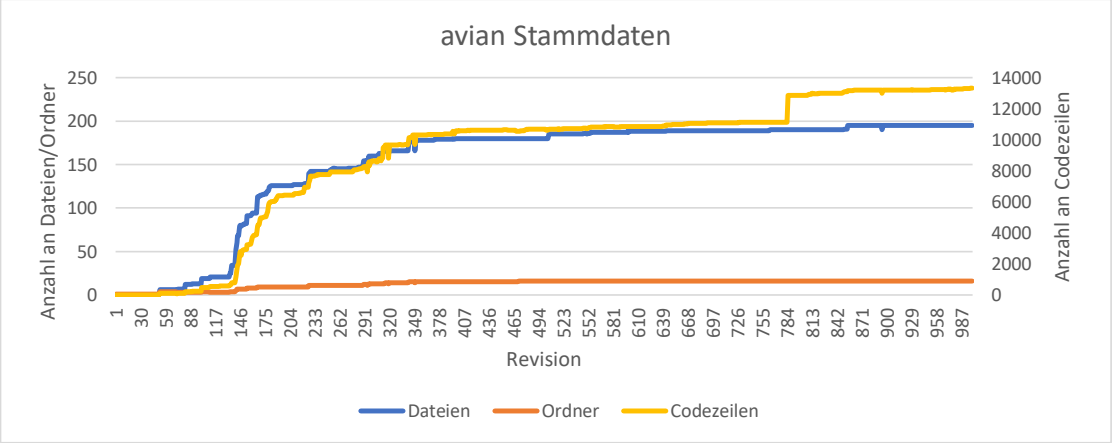

Abbildung A.17 Stammdaten von avian

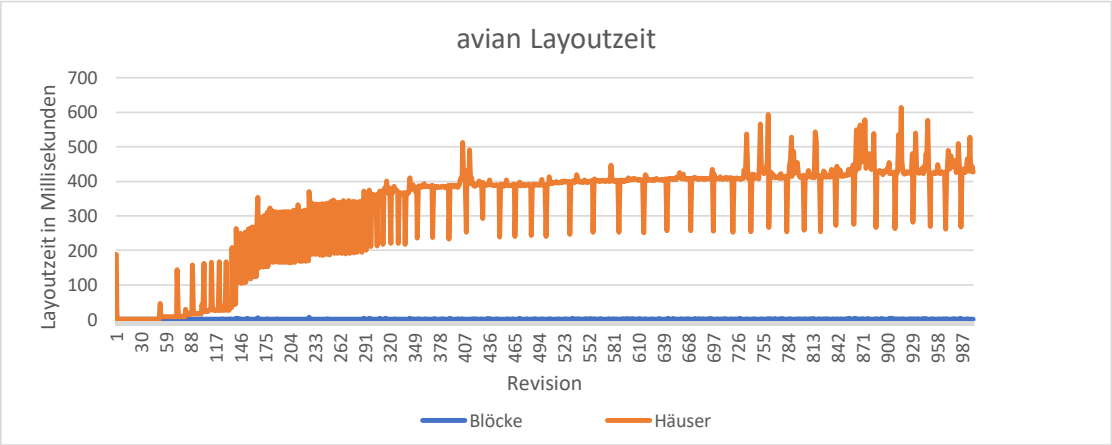

Abbildung A.18 Rechenzeit für das Layout von avian

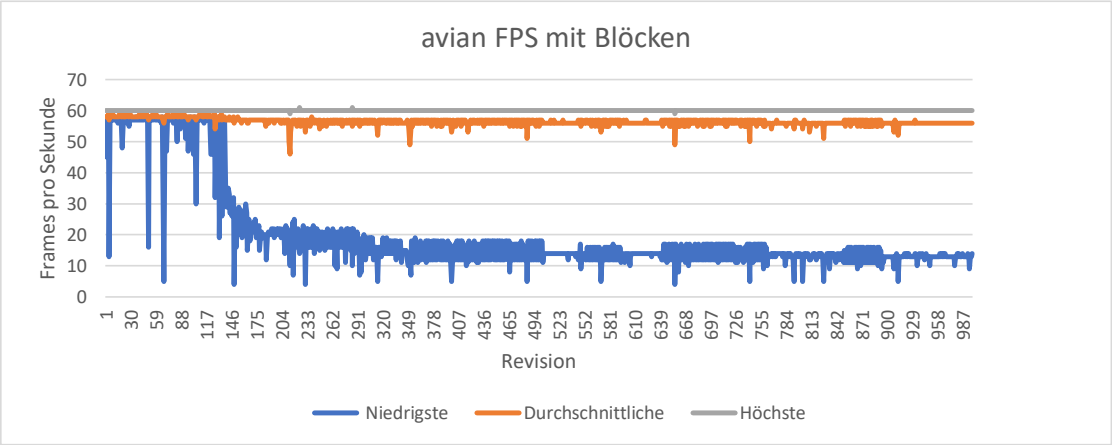

Abbildung A.19 Framerate mit Blöcken für avian

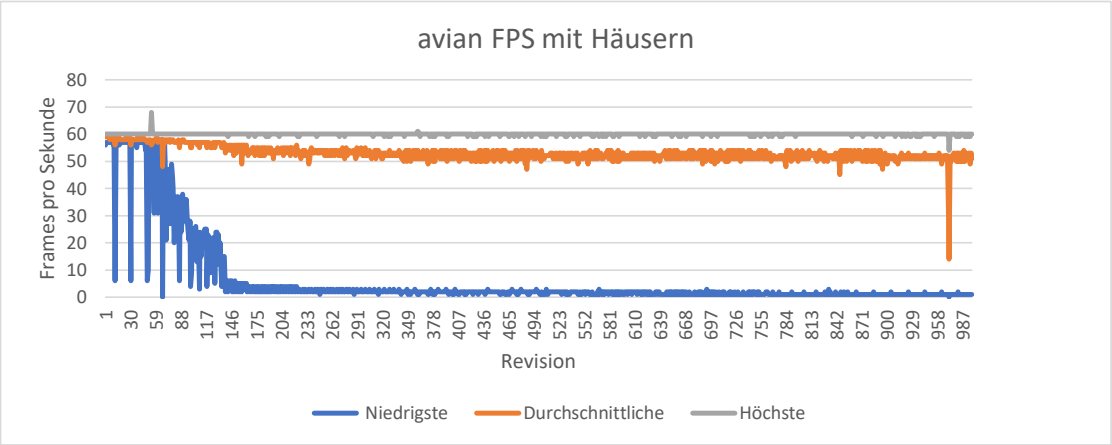

Abbildung A.20 Framerate mit Häusern für avian

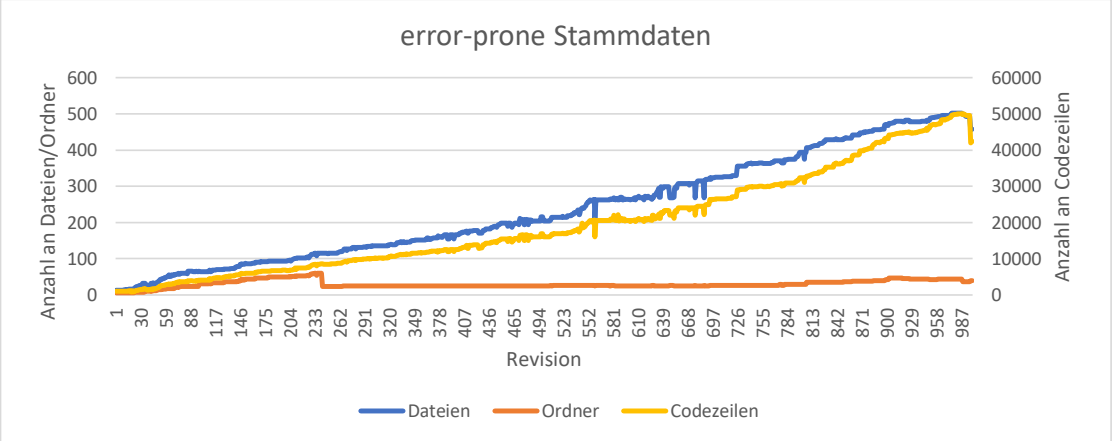

Abbildung A.21 Stammdaten von error-prone

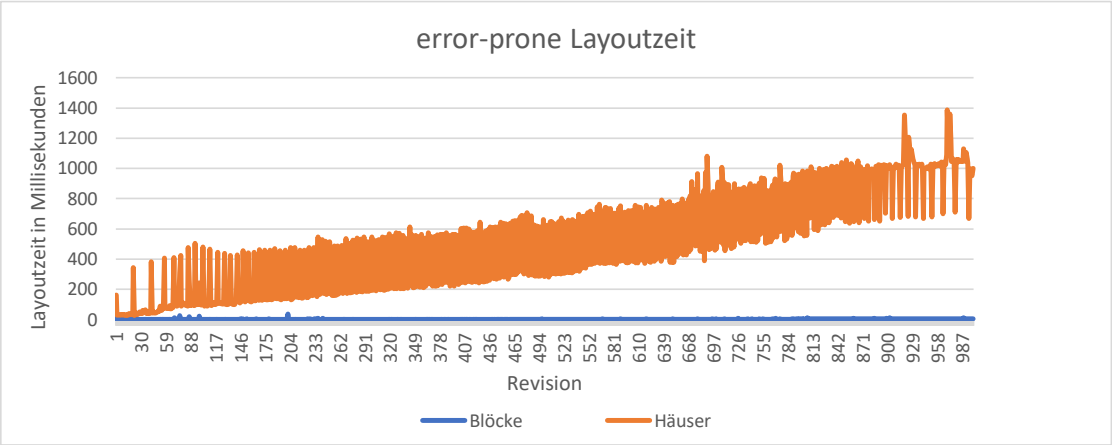

Abbildung A.22 Rechenzeit für das Layout von error-prone

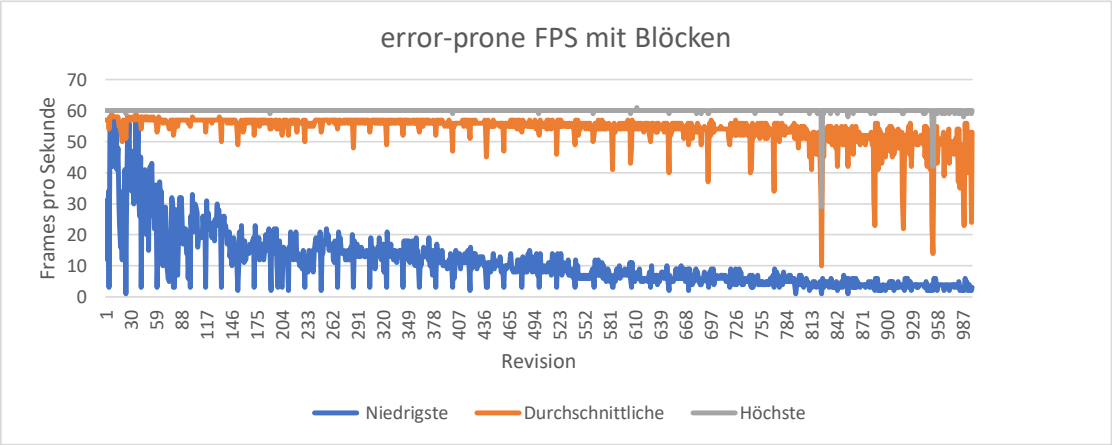

Abbildung A.23 Framerate mit Blöcken für error-prone

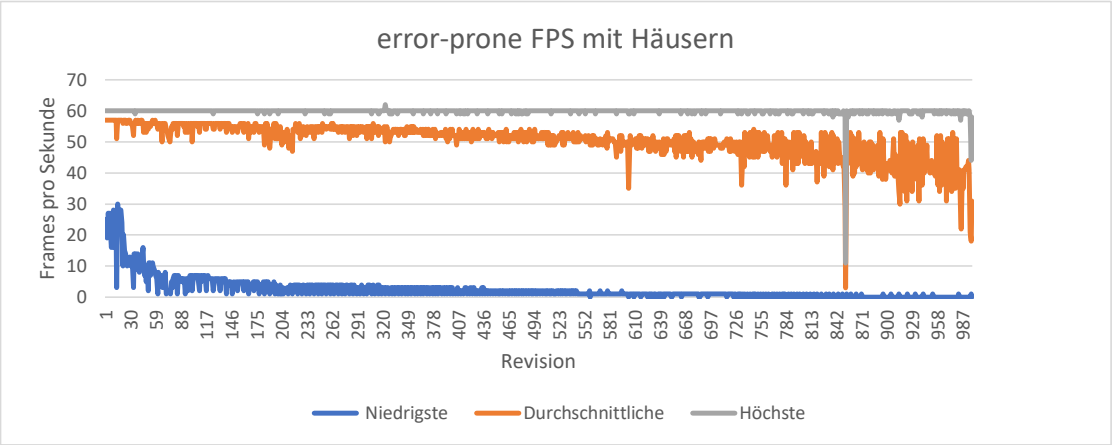

Abbildung A.24 Framerate mit Häusern für error-prone

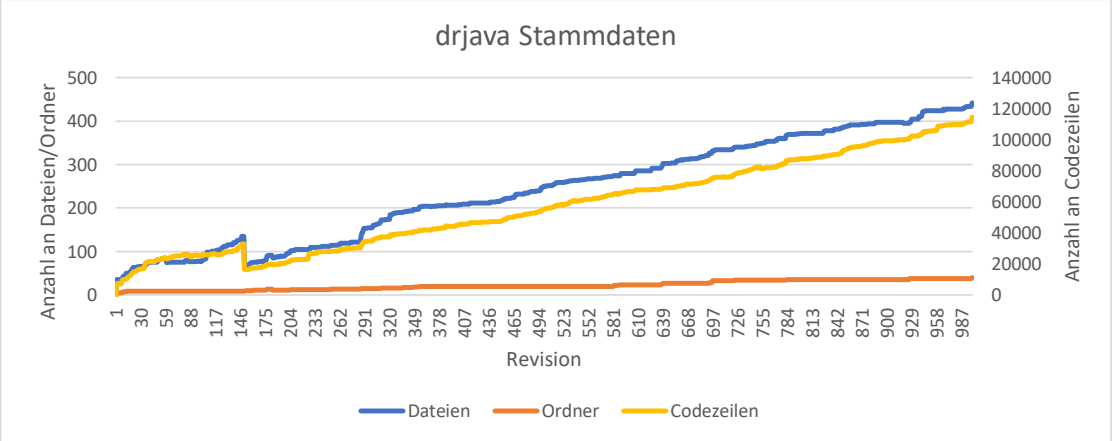

Abbildung A.25 Stammdaten von drjava

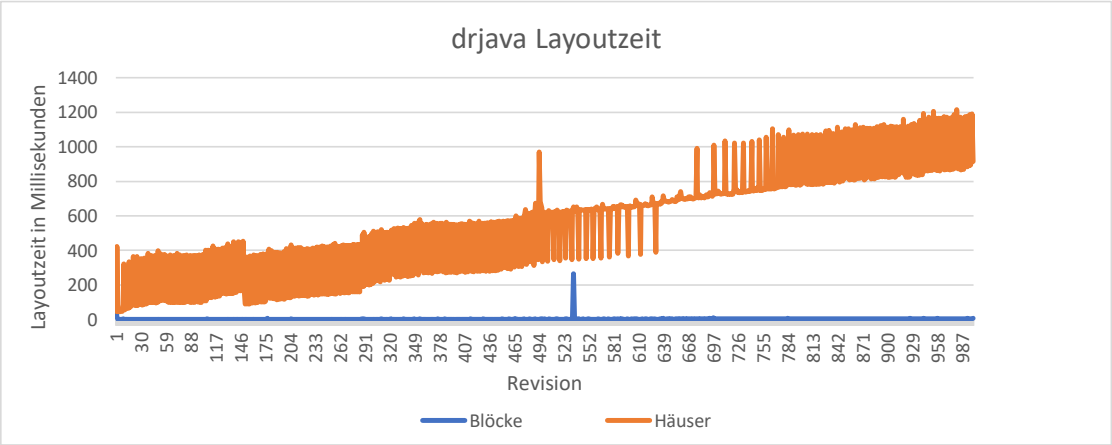

Abbildung A.26 Rechenzeit für das Layout von drjava

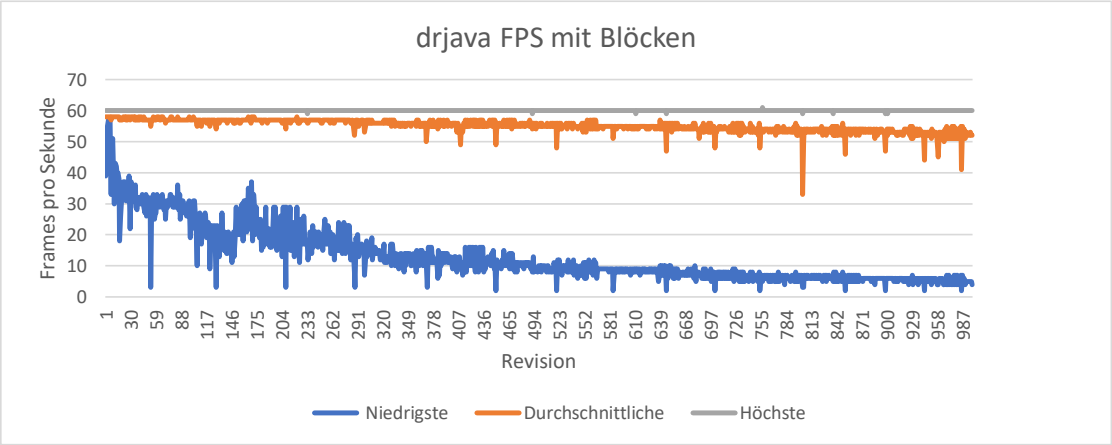

Abbildung A.27 Framerate mit Blöcken für drjava

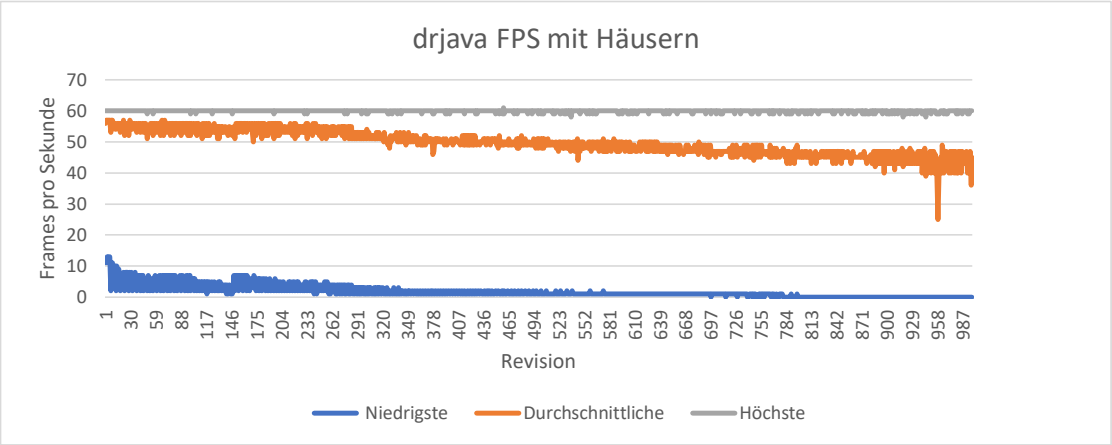

Abbildung A.28 Framerate mit Häusern für drjava

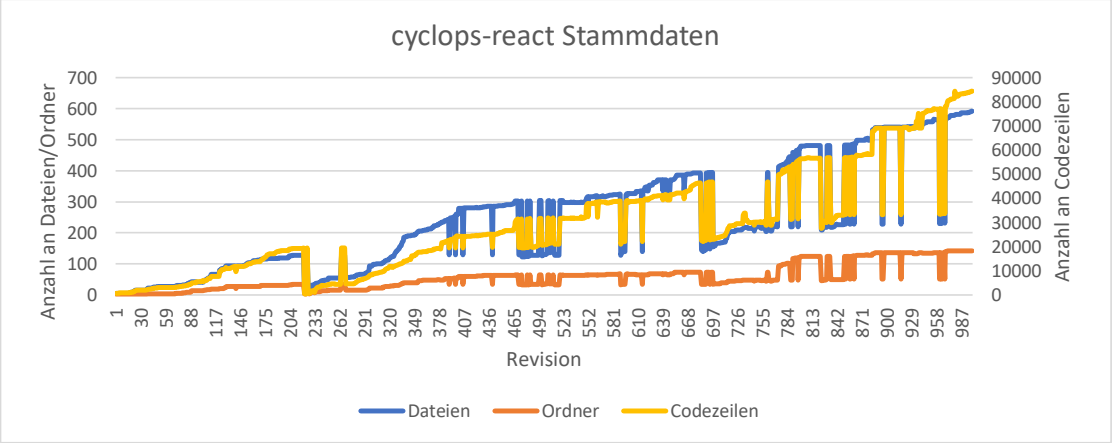

Abbildung A.29 Stammdaten von cyclops-react

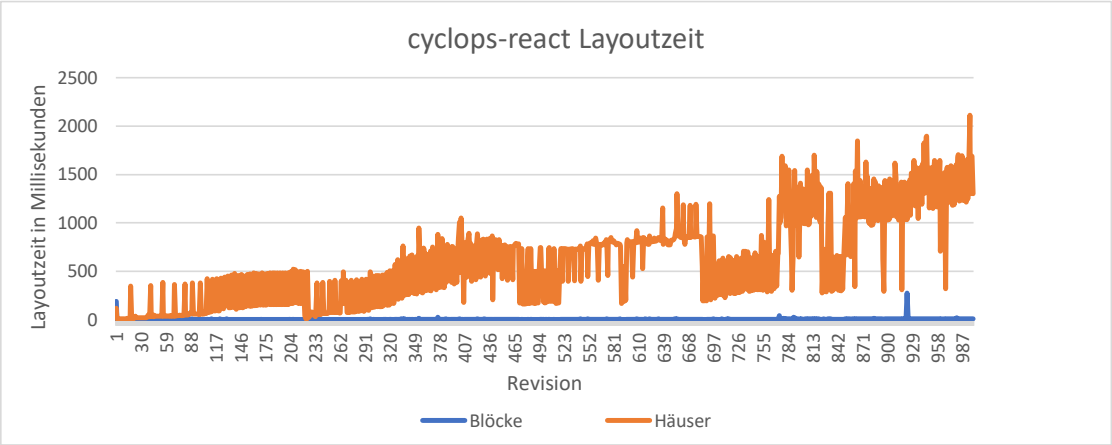

Abbildung A.30 Rechenzeit für das Layout von cyclops-react

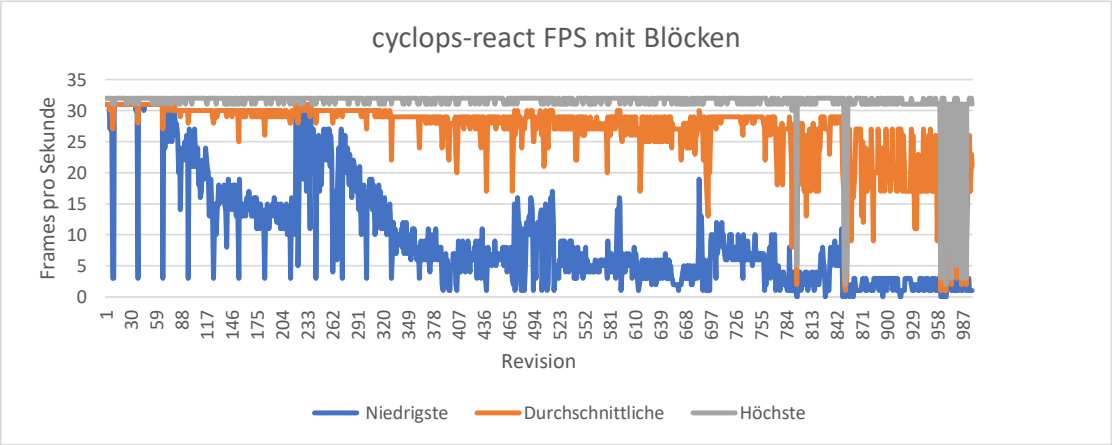

Abbildung A.31 Framerate mit Blöcken für cyclops-react

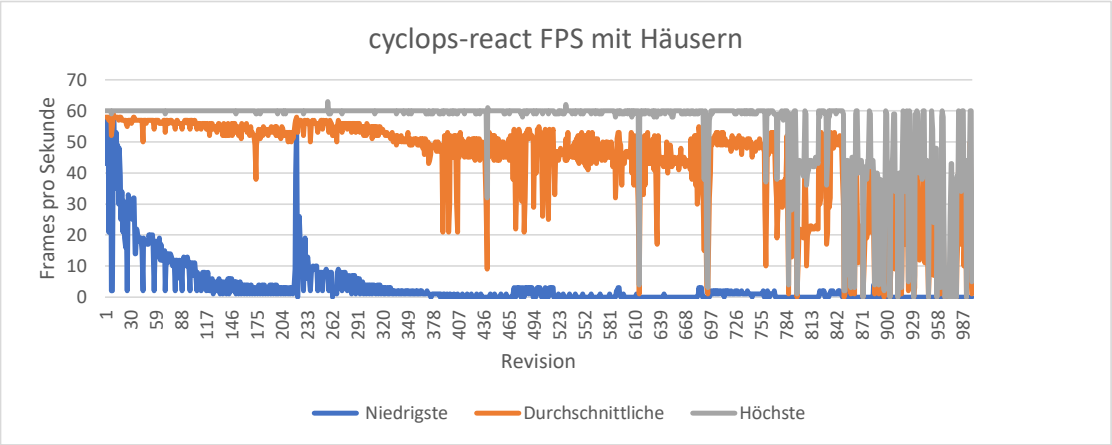

Abbildung A.32 Framerate mit Häusern für cyclops-react

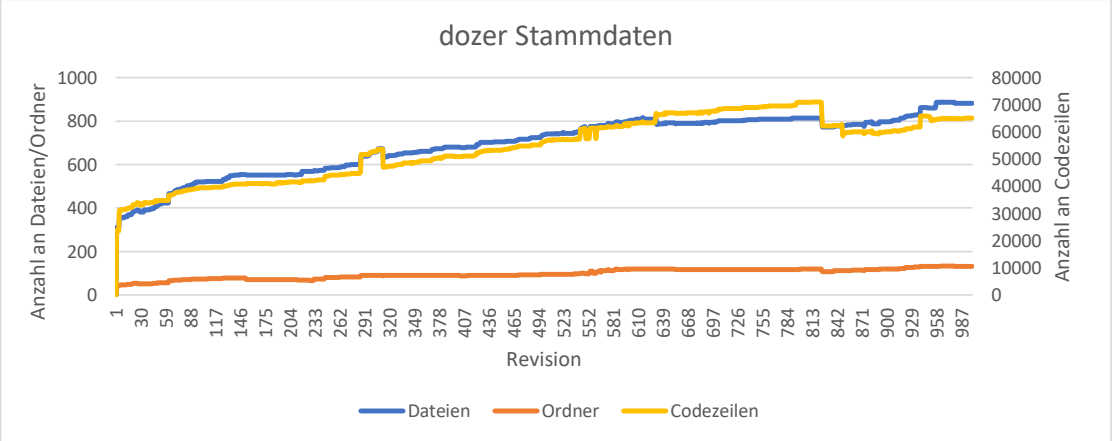

Abbildung A.33 Stammdaten von dozer

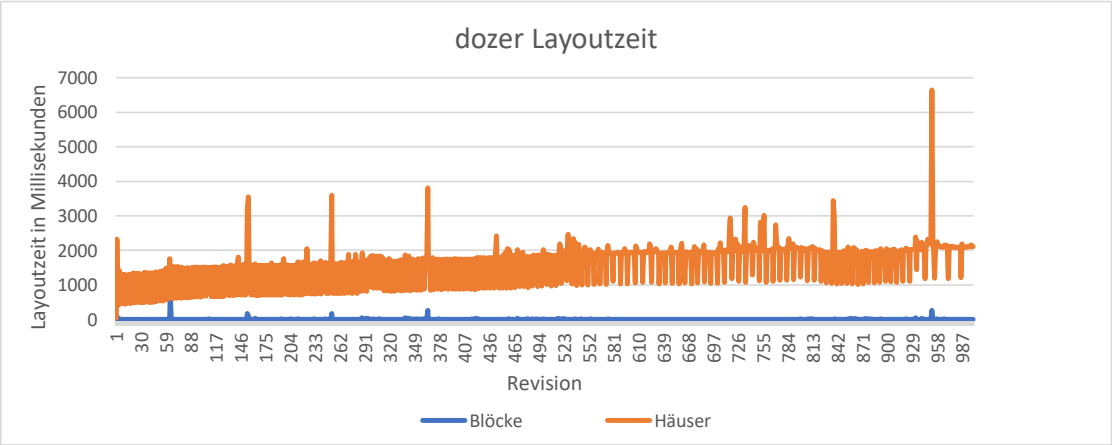

Abbildung A.34 Rechenzeit für das Layout von dozer

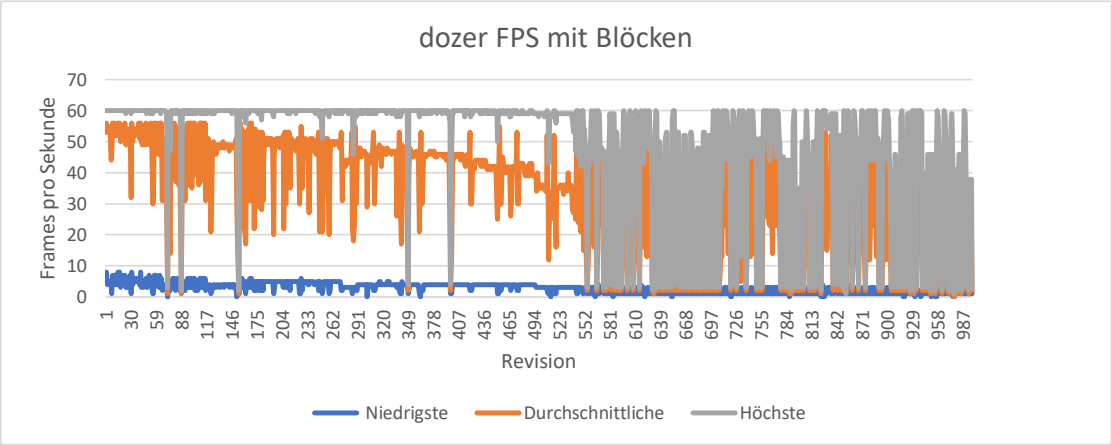

Abbildung A.35 Framerate mit Blöcken für dozer

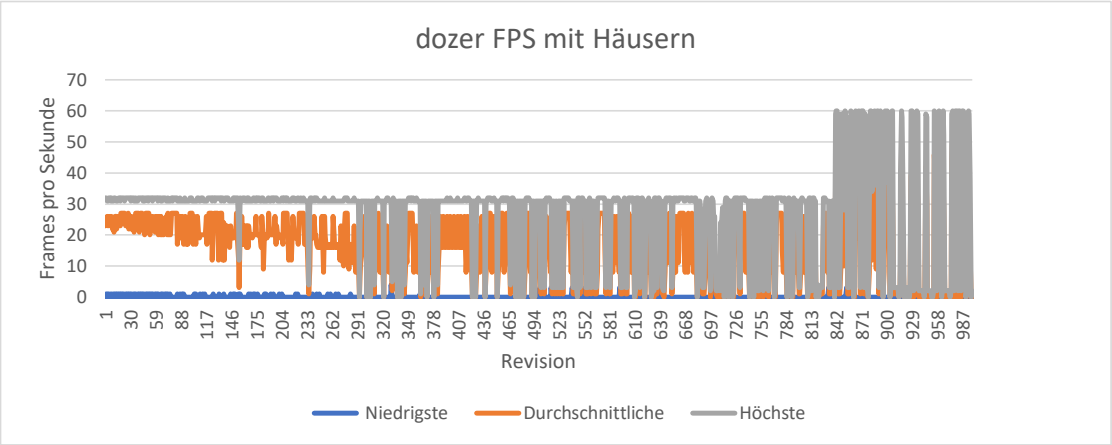

Abbildung A.36 Framerate mit Häusern für dozer

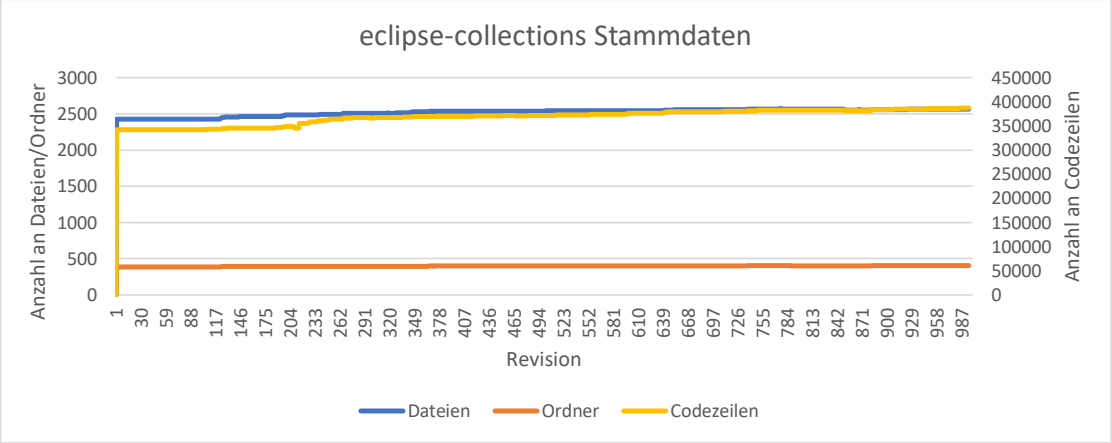

Abbildung A.37 Stammdaten von eclipse-collections