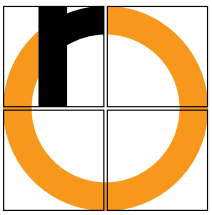
Hochschule **Rosenheim**
University of Applied Sciences



Master Thesis

CodeLeaves als neues Konzept der 3D Softwarevisualisierung und dessen Umsetzung für die Augmented Reality

Marcel Pütz
2017

Erklärung

Ich versichere, dass ich diese Arbeit selbständig angefertigt, nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benützt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Rosenheim, den 9. Oktober 2017

Marcel Pütz

Kurzfassung

here comes the abstract

Schlagworte: Softwarevisualisierung, Wald, Baum, 3D, Augmented Reality, Virtual Reality, Statische Codeanalyse, Abhängigkeiten

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

> *Virtual reality was once the dream of science fiction. But the internet was also once a dream, and so were computers and smartphones. The future is coming.*
>
> Mark Zuckerberg
> *Facebook CEO*, 2014

> *I think AR is [...] big, it's huge. I get excited because of the things that could be done that could improve a lot of lives.*
>
> Tim Cook
> *Apple CEO*, 2017

1.1 Einführung

Viele der einflussreichsten Technologieunternehmen arbeiten an der *Virtual* bzw. *Augmented Reality (VR* bzw. *AR)*. Tim Cook ist überzeugt davon, dass die AR die nächste *"big idea"* nach dem Smartphone wird [13].

Nicht nur Großkonzerne wie Apple, Facebook oder Samsung arbeiten intensiv in diesem Bereich. Ein Start-up-Unternehmen namens *Magic Leap* entwickelt eine AR Brille und wird von Investoren in Billionenhöhe unterstützt [10]. Laut einem Cover-Artikel der Zeitschrift *Wire* ist die noch unter Verschluss gehaltene Technologie den Konkurrenzprodukten allen voraus. Das Release der Hardware ist Stand heute noch nicht bekannt, aber es lässt sich ein Trend erkennen, der die nächsten Jahre viele neue Möglichkeiten eröffnen wird und möglicherweise die Digitalisierung revolutionieren könnte. Diese Arbeit wird sich mit einer Anwendung für die AR beschäftigen – der Softwarevisualisierung. Die Spezialisierung auf AR kann nach der Abgrenzung von AR und VR besser nachvollzogen werden.

VR ist eine Umgebung, in der der Betrachter vollkommen von einer computergenerierten Welt umgeben ist, die oft die reale Welt imitiert, aber auch rein fiktiv sein kann [11].

Obwohl der Begriff AR zunehmend in der Industrie Verwendung findet, entbehrt er doch einer einheitlichen Definition. In [2] wird AR als *„Variation“* von VR betrachtet. Dagegen vermittelt Milgrim in [11] ein vollständigeres Verständnis, weshalb sich die Begrifflichkeiten in dieser Arbeit daran anlehnen sollen. Nach Milgrim existieren die beiden entgegengesetzten Extreme der Realität und der Virtualität. Alles dazwischen ist die sogenannte *Mixed Reality (MR)*.

MR ist eine Umgebung, in der Elemente der realen und einer virtuellen Welt zusammen dargestellt werden [9].

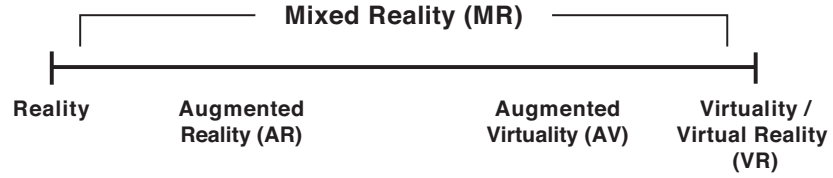


Abbildung 1.1 Abgewandelte Darstellung des Reality-Virtuality-Kontinuums aus [11]

Dieses *Reality-Virtuality-Kontinuum* ist in Abbildung 1.1 dargestellt, in dem gut zu erkennen ist, dass AR zu der Mixed Reality gehört. In den meisten Quellen wie [2, 1, 9] wird bei der AR noch die Komponente der Interaktion aufgeführt. AR kann deshalb folgendermaßen definiert werden:

Definition 1.1: AR

AR ist die Erweiterung der realen Welt durch computergenerierte Elemente, mit denen der Betrachter in Echtzeit interagieren kann.

Auch die *Augmented Virtuality*, also die Erweiterung der virtuellen Welt durch reale Elemente, gehört zur MR.

Wie viele der einflussreichsten Menschen der Technologie-Industrie, sieht Tim Cook mehr Zukunft in der AR, da, wie er in einem Interview sagt, diese Technologie nicht wie die VR die wirkliche Welt ausschließt, sondern die Realität erweitert und Teil von zwischenmenschlicher Kommunikation sein kann [13].

Wir stellen uns ein Hologramm vor, dass auf einem Konferenztisch Gestalt annimmt und ein Software-System repräsentiert. Entwickler, Projektleiter oder auch Kunden versammeln sich um den Tisch und können miteinander interaktiv die Software betrachten, evaluieren und wichtige Informationen daraus ziehen.

Dies wäre mit VR nicht möglich, da der Betrachter von der Außenwelt abgeschottet ist. Deshalb wird im Zuge dieser Arbeit mit der Stand heute am weitesten ausgereiften Technologie der AR gearbeitet – der *HoloLens* von Microsoft.

1.2 Motivation dieser Arbeit

Die Technologie der AR bietet uns viele neue Möglichkeiten. Eine Motivation dieser Arbeit ist es sich produktiv mit einer neuen, zukunftsträchtigen Technologie zu beschäftigen. Das ist jedoch nur die eine Seite. Die weitaus größere Motivation ist, die zuvor noch nicht dagewesene Zugänglichkeit und Interaktion mit dreidimensionaler *Visualisierung* auszunutzen. Visualisierung im Allgemeinen begegnet uns in vielen Bereichen unseres Lebens und nimmt eine wichtige Rolle ein.

Niemand konnte bislang unser Sonnensystem von außen betrachten. Dennoch haben wir alle eine ziemlich gute Vorstellung wie dieses aufgebaut ist. Durch die Visualisierung der Planeten und der Sonne entsteht in uns ein geistiges Abbild der Realität. Das Konzept komplexe Realitäten zu abstrahieren und zu visualisieren, um dadurch die Realität besser verstehen zu können, ist in vielen Disziplinen der Wissenschaft vertreten.

Neben Wissenschaften wie Physik, Chemie oder Biologie, nimmt Visualisierung auch besonders in der Informatik eine wichtige Rolle ein. In vielen Bereichen müssen Informationen in eine visuelle Form gebracht werden, die für das menschliche Auge besser zu lesen sind.

Gerade bei komplexen Software-Systemen ist das der Fall. Soll zum Beispiel die zu Grunde liegende Struktur einer Software Außenstehenden erklärt werden, gelingt das mit einem visuellen Modell wie einem UML-Diagramm sicherlich besser, als nur in den Source-Code zu schauen.

So wie UML-Diagramme, war die Darstellungsform der Softwarevisualisierung bislang meist zweidimensional. Mit AR wird dieser Disziplin der Visualisierung jedoch wortwörtlich ein neuer Raum an Möglichkeiten eröffnet und in diese Arbeit soll diesen Raum ausfüllen.

1.3 Zielsetzung

Für die Zielsetzung einer 3D Softwarevisualisierung in der AR sollten zunächst die allgemeinen Ziele einer Softwarevisualisierung betrachtet werden. Softwarevisualisierung ist für Diehl die „visualization of artifacts related to software and its development process“ [5]. Wird der Fokus mehr auf die Ziele, d.h. den Nutzen für den Betrachter gelegt, lässt sich Softwarevisualisierung wie folgt definieren:

Definition 1.2: Softwarevisualisierung

Softwarevisualisierung ist die bildliche oder auch metaphorische Darstellung einer Software, um dem Betrachter durch Vereinfachung und Abstraktion das bessere Verständnis oder die einfachere Analyse von Software zu ermöglichen.

In dieser Arbeit soll das neue Konzept *CodeLeaves* für eine solche Softwarevisualisie-

rung in der AR vorgestellt und im Detail ausgearbeitet werden.

Dabei soll CodeLeaves, im Vergleich zu andern 3D Softwarevisualisierungen, die Vorteile der Dreidimensionalität optimal ausnutzen.

Im Vorfeld dieser Arbeit wurden in einer Studie Metriken gesammelt, die eine gute Softwarevisualisierung bzw. CodeLeaves unterstützen sollte.

Es sollen dynamische und statische Metriken zur Erkennung von Anomalien in einer Software unterstützt werden. Ebenfalls soll die Darstellung der Struktur und der darauf abgebildeten Abhängigkeiten innerhalb einer Software möglich sein.

Durch weitere Expertengespräche sollen Userstories erstellt werden um den Mehrwert des neuen Konzepts validieren zu können.

Um diesen Anforderungen gerecht zu werden, soll für CodeLeaves ein sprachunabhängiges Datenmodell entworfen werden, dass alle geforderten Metriken unterstützt.

Der Praktische Teil dieser Arbeit soll die prototypische Entwicklung von CodeLeaves für die HoloLens sein.

1.4 Aufbau der Arbeit

Im Kapitel 2 wird das Konzept von CodeLeaves vorgestellt. Dabei wird zunächst unter Betrachtung alternativer Ansätzen begründet, wieso ein neues Konzept sinnvoll ist, um dann in Abschnitt 2.1 genauer auf das Konzept einzugehen. Die Befragung von Experten der Softwareanalyse und die daraus abgeleiteten Anforderungen an CodeLeaves in Abschnitt 2.3 schließen das erste Kapitel ab.

Das Kapitel 3 beschäftigt sich mit der Entwicklung eines geeigneten Datenmodells für CodeLeaves. Es werden vorhandene Datenmodelle auf Tauglichkeit für CodeLeaves überprüft und Rücksprache mit erfahrenen Software-Ingenieuren gehalten.

Aufbauend auf das entwickelte Datenmodell, wird das Konzept von CodeLeaves in Kapitel 4 theoretisch weiter ausgearbeitet. Darunter fällt die Positionierung der Bäume auf einer Grundfläche und die Länge, Dicke und der Winkel der einzelnen Äste. Parallel zur Theorie wird aufgezeigt, wie sich CodeLeaves in Unity für die HoloLens modellieren lässt.

Die Interaktion mit CodeLeaves soll Thema des Kapitel 5 sein. Besonders die Abhängigkeiten von Artefakten einer Software sollen mithilfe von CodeLeaves interaktiv exploriert werden können.

Das Kapitel 6 fasst die Ergebnisse der Arbeit zusammen und ein Ausblick auf zukünftige Verwendung und weiterführende Arbeiten runden die Arbeit ab.

1.5 Abgrenzung

Die Beschaffung der Informationen über eine Software soll wird hier nur am Rande betrachtet. Für die Analyse einer Software müssen statische und dynamische Informationen gesammelt und aggregiert werden. Dafür gibt es verschiedenste Tools, wie inhouse Entwicklungen der QAware mit dem *Software-EKG* und dem *QAvalidator* oder proprietäre und Open-Source-Software. Die Anbindung an CodeLeaves würde den Rahmen dieser Arbeit jedoch sprengen.

Stattdessen soll der Fokus besonders auf das grundlegende Konzept und das Frontend mit der Generierung und Interaktion des Waldes gelegt werden. Diese Arbeit stellt jedoch den Anspruch, dass mit realistischen Daten gearbeitet wird, um eine valide Einschätzung des Mehrwerts von CodeLeaves geben zu können.

2 Das Konzept CodeLeaves

2.1 CodeLeaves und die Metapher Software-Wald

"Hierarchies are almost ubiquitous [...]" [17] halten Robertson *et al.* schon 1991 bei der Visualisierung von hierarchischen Informationen fest. So auch bei der Struktur einer Software. Jede Software mit einer geschachtelten Paketstruktur ist hierarchisch und kann in einer Baumstruktur dargestellt werden. *„Bäume sind eine der wichtigsten Datenstrukturen, die besonders im Zusammenhang mit hierarchischen Abhängigkeiten und Beziehungen zwischen Daten von Vorteil sind."* [6] stellen auch Ernst *et al.* fest. Draus folgt, dass die Darstellung von Software als Baum oder auch Bäume sinnvoll ist.

Der Baum in der Informatik zeugt von einer ursprünglichen Metapher – der Baum, wie er draußen in der Natur wächst. Da dieser unbestritten dreidimensional ist, liegt eine Softwarevisualisierung für die Dreidimensionalität mit einer realitätsnäheren Interpretation der Baum-Metapher nahe. Werden die Bäume, die im zweidimensionalen meist von oben nach unten gezeichnet wird, in 3D in natürlicher Wuchsrichtung modelliert und mehrere Bäume für eine Software verwendet, entsteht eine neue Metapher: der *Software-Wald*.

CodeLeaves stellt ein Konzept dar, dass sich die Metapher des Software-Waldes zu nutze macht und wird im Folgenden auf High-Level-Ebene skizziert und danach genauer erläutert:

Konzept: CodeLeaves

1. Die Struktur der Software wird mit nach oben wachsenden Bäumen dargestellt.
2. Jedes Paket im *Root-Verzeichnis* wird als einzelner Baum auf einer Ebene – dem *Waldboden* – dargestellt.
3. Die Blätter der Bäume entsprechen den Softwareartefakten (z.B. Klassen) und können durch ihre Farbe eine beliebigen Metrik visualisieren.
4. Zwischen den Bäumen entsteht ein *Wurzelgeflecht*, was die aggregierten Abhängigkeiten zwischen den Paketen darstellt.
5. Abhängigkeiten oder Aufrufe zwischen einzelnen Softwareartefakten werden aggregiert über die Elternpakete als farbige bzw. Dicke der Äste dargestellt oder alternativ als

direkte *Spinnweben* zwischen den Bäumen.

Punkt 1 ist dafür verantwortlich, dass die Softwarevisualisierung nahe an der tatsächlichen Software ist, wie sie ein Entwickler in seiner Code-Base gewöhnt ist. Das heißt, diejenigen, die auch tatsächlich mit der Software arbeiten, finden sich aufgrund der bekannten Baumstruktur auch in der Visualisierung schnell zurecht, ohne jedes Softwareartefakt auszuwählen, um zu sehen, mit welchem sie es zu tun haben. Im Fachjargon wird hier von der *Habitability* gesprochen, also wie schnell oder gut sich ein Betrachter in einer Software oder auch deren Visualisierung „zuhause“ fühlt [19].

Punkt 2 ist weitgehend selbsterklärend. Durch Darstellung der Software in mehreren Bäumen entsteht erst der Software-Wald und die Pakete im Root-Verzeichnis als Wurzeln der Bäume zu verwenden bietet sich an. Das schließt jedoch nicht aus, dass Unterpakete als neuer Waldboden verwendet wird. Interaktion mit dem Waldboden soll Teil des Kapitel 5 sein.

Punkt 3 bedeutet, dass in CodeLeaves durch die Farbe der Blätter ein *Laubdach* entsteht, dass eine gute Übersicht über eine ausgewählte Metrik der Software gibt. Beispielsweise kann der Farbe der Blätter die Code-Coverage der einzelnen Softwareartefakte (siehe Definition 2.1) zugewiesen werden. Mit einer Skala von Grün bis Rot kann dann der Software-Wald bei guter Testabdeckung im sommerlichen Grün erstrahlen, oder bei einer weniger guten Coverage eher in den Herbst übergehen. Die Färbung des Laubdachs ist flexibel auf jegliche Metrik anwendbar, die bei der betrachteten Software zur Verfügung steht.

Definition 2.1: Softwareartefakt

Ein Softwareartefakt wird in dieser Arbeit als Überbegriff für die kleinste betrachtete Einheit der Software verstanden. Das können bei objektorientieren Sprachen typischerweise Klassen, aber auch bei feinerer Granularität einzelne Funktionen innerhalb einer Klasse sein. Auch die Betrachtung von Dateien, die mehrere Klassen enthalten können, sind denkbar.

Punkt 4 bietet einen goßen Vorteil gegenüber der Zweidimensionalität. In Abbildung 2.1 wird rechts das Prinzip des Wurzelgeflechts mit einem minimalistischen Beispiel illustriert.

Die Abhängigkeiten zwischen den Bäumen wird auf einen Blick ersichtlich. Betrachtet man die Bäume in 2D, wie es im linken Teil der Abbildung dargestellt ist, verschwinden die Abhängigkeiten hintereinander und sind nicht zuzuordnen.

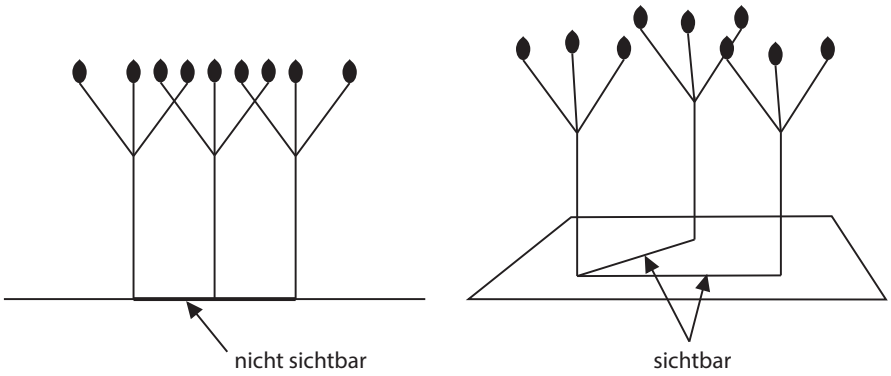
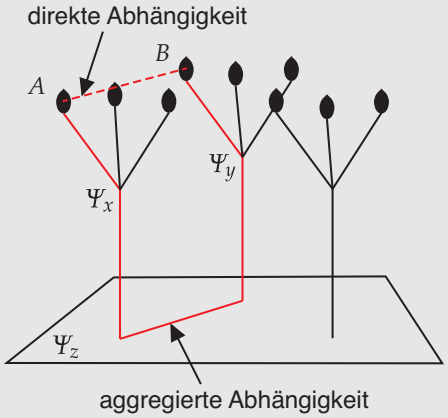


Abbildung 2.1 Vorteil der Dreidimensionalität bei der Darstellung von Abhängigkeiten

Punkt 5 heißt, dass die Abhängigkeiten einer Software sehr übersichtlich auf die Struktur der Software abgebildet werden können, ohne diese zu beeinflussen. Für das besser Verständnis bedarf es einer Definition der Aggregation von Abhängigkeiteän.

Definition 2.1: Aggregation von Abhängigkeiten

Bei der Aggregation von Abhängigkeiten zwischen Softwareartefakten werden die Abhängigkeiten nicht direkt dargestellt, sondern über die Eltern-Pakete geleitet. Seien x, y Pakete und Softwareartefakt $A \in x$ besitze eine Abhängigkeit zu Softwareartefakt $B \in y$, dann geht die Abhängigkeit von A zu einem zusätzlich Konstrukt Ψ_x, das das Pakets x repräsentiert. Angenommen x und y befinden sich zudem im Paket z, dann geht die aggregierte Abhängigkeit entweder direkt von Ψ_x zu Ψ_y, oder weiter über Ψ_z zu Ψ_y und schließlich B.



Auf der rechten Seite der Definition 2.1 ist eine aggregierte Abhängigkeit am Beispiel von CodeLeaves zu sehen. Die Ψ in CodeLeaves sind die Knoten der Bäume, an denen die Äste zusammen laufen und im Spezialfall des Root-Verzeichnisses, der Waldboden.

Bei mehreren Abhängigkeiten zwischen Nachbar-Paketen, überlagern sich die aggregierten Abhängigkeiten zwangsläufig. Falls in unserem Beispiel eine weitere Abhängigkeit von Paket x zu Paket y bestünde, würden sich die Abhängigkeiten von Ψ_x bis Ψ_y überlagern. Daraus ergibt sich, dass nicht jede aggregierte Abhängigkeit ohne Interaktion zwangsläufig eindeutig zuzuordnen ist.

Wird aber bei jeder Kante, sei es ein Ast, Stamm, oder Wurzel, die Anzahl an

überlagernden Abhängigkeiten als Dicke der Kante dargestellt, bekommt der Betrachter eine gute Übersicht über die Gesamtheit der Abhängigkeiten. Durch Interaktion mit einzelnen Kanten, oder sogar des ganzen Waldbodens, soll eine fein granulärere Analyse der Abhängigkeiten möglich sein.

Das zweite Element von Punkt 5 sind die Spinnweben. Damit lassen sich die Abhängigkeiten direkt darstellen. Um bei großen Software-Systemen aber die Übersicht über den Wald nicht zu verlieren, sollten diese mithilfe der Interaktion des Nutzers flexibel aktivierbar sein.

Nachdem das grundlegende Konzept von CodeLeaves verstanden ist, muss eine Begriffsdefinition ein Einheitliches Verständnis der verwendeten Komponenten schaffen. Die eingeführten Bezeichnungen dienen dem Verständnis der verwendeten Datenmodelle im nächsten Kapitel, sowie die entwickelten Zeichenalgorithmen in Kapitel 4.

Die Struktur von CodeLeaves baut stark auf der Datenstruktur eines klassischen Baumes auf – *„the most important nonlinear structures that arise in computer algorithms“*[**knuth1973fundamental**] – die zum Beispiel in [**knuth1973fundamental**, **gumm2009einfuehrung**, 6] definiert wird.

Da aber die Struktur von CodeLeaves zum Teil von der klassischen Definition abweicht und teilweise neue Bezeichnungen benötigt, betrachten wir daher einige Definitionen zur Baumstruktur, um diese an die Gegebenheiten von CodeLeaves anzupassen und führen neue Begriffe ein.

2.2 Begriffsklärung

- Ein *Baum* (engl.: *tree*) besteht aus einer Menge von *Knoten*, die so durch *Kanten* verbunden sind, dass keine Kreise auftreten (Abgewandelt aus [**gumm2009einfuehrung**, 6]).
- Ein *Knoten* beinhaltet Informationen zu sich selbst und hat 0 bis *n Kinder* (auch Nachfahren genannt, engl.: *childs* oder *descendant*).
- Ein Knoten mit keinen Kindern wird als *Blatt* (engl.: *leaf*) bezeichnet. Alle anderen Knoten heißen *innere Knoten* (engl.: *innerNode*) [**gumm2009einfuehrung**].
- Kinder des selben Knotens werden *Geschwister* (engl.: *siblings*) genannt.

Bei der klassischen Definition eines Baumes wird bei dem Knoten ohne Elternteil von der „Wurzel“ gesprochen. Im Modell von CodeLeaves ist der unterste Knoten des Baumes jedoch noch durch eine Kante mit dem Waldboden verbunden. Darüber hinaus überschneidet sich die Bezeichnung von „Wurzel“ mit den Verbindungen zwischen den einzelnen Bäumen, die bei der klassischen Definition nicht existieren. Deshalb wird für diesen speziellen Knoten eine neue Bezeichnung eingeführt.

- Der unterste Knoten eines Baumes wird als *Kronenansatz* (engl.: *crown base*) bezeichnet.
- Alle Knoten bis auf den Kronenansatz haben genau einen Knoten als *Elternteil* (auch Vorfahre genannt, engl.: *parent* oder *ancestor*).

Nachdem in den meisten Fällen Bäume in 2D nach unten wachsend dargestellt werden, was wahrscheinlich auf die Tatsache zurück zu führen ist, dass handschriftliche Diagramme tendenziell nach unten wachsend gezeichnet werden [**knuth1973fundamental**], wird bei Knoten auch oft von einer Tiefe gesprochen. Diese Bezeichnung wird beibehalten.

- Die *Tiefe* eines Knotens gibt an, wie viele Kanten er vom Kronenansatz aus entfernt liegt (Abgewandelt aus [6]).
- Die *Höhe* (engl.: *height*) eines Knoten beschreibt die maximale Tiefe aller Nachfahren.
- Alle Knoten mit der gleichen Höhe befinden sich auf der selben *Ebene*

Alle nachfolgenden Kanten und Knoten eines Knotens A werden in der Literatur unterschiedlich bezeichnet. Es ist die Rede von „Teilbaum“ [6] oder auch „Unterbäumen“ [**gumm2009einfuehrung**]. Wir definieren dafür einen dem 3D Modell besser entsprechenden Begriff.

- Ein *Ast* (engl.: *branch*) sind alle nachfolgenden Kanten und Knoten des Knotens A ausschließlich des Knotens A selbst.

Bisher wurden Bezeichnungen aus einschlägiger Literatur verwendet oder abgeändert. Betrachten wir nun Elemente von CodeLeaves, die so nicht in der klassischen Definition einer Baumes vorkommen.

- Der Kronenansatz ist durch die Kante namens *Stamm* (engl.: *trunk*) mit den *Waldboden* (engl.: forest floor) verbunden.
- Der Schnittpunkt zwischen Stamm und Waldboden nennen wir *Stammbasis* (engl.: *trunk base*). Dieser ist jedoch kein Knoten und beinhaltet auch keine Informationen.
- Ein *Wald* besteht aus einer disjunkten Menge an Bäumen und dem Waldboden.
- Ein Waldboden eines Waldes mit n Bäumen besitzt 0 bis $n!$ *Wurzeln* (engl. *roots*), die jeweils zwei Stammbasen miteinander verbinden.

2.3 Vergleich mit anderen Konzepten

Im Vorfeld dieser Arbeit wurde in [14] evaluiert, was eine gute Softwarevisualisierung ausmacht und unterstützen sollte. Ausgehend davon, wurden vorhandene 3D Visualisierungen und andere mögliche Konzepte miteinander verglichen. Die Ergebnisse dieser Untersuchung, soll in Folgenden vorgestellt werden.

Um herauszufinden welchen Mehrwert sich Nutzer einer Softwarevisualisierung von dieser versprechen, wurde eine Umfrage in der QAware GbmH durchgeführt. Die QAware ist ein Projekthaus mit den Kerngeschäften Diagnose, Sanierung, Exploration und Realisierung von Software [15]. Durch die Erfahrung in Projekten für namhafte Kunden,

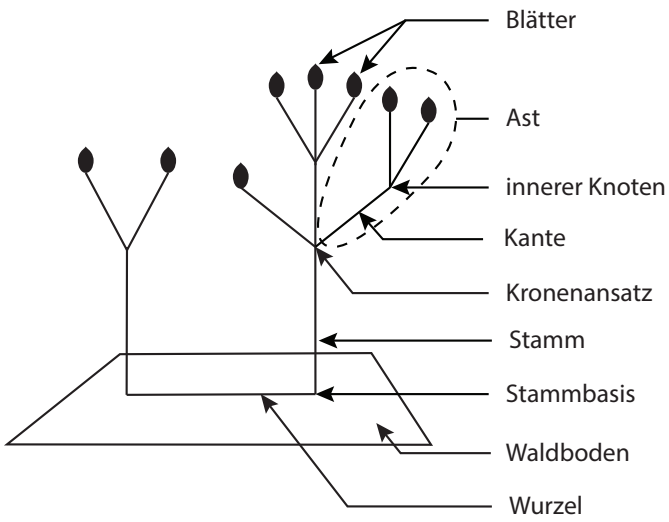


Abbildung 2.2 Bezeichnungen

zeichnen sich die Mitarbeiter durch fundiertes Wissen und Expertise aus. Es wurden insgesamt 22 Mitarbeiter mit unterschiedlichen Rollen in der Softwareentwicklung befragt.

In Abbildung 2.2 ist zu sehen, wie oft welche Metriken genannt wurden. Alle Metriken lassen sich in die drei Kategorien der Softwarevisualisierung aus [5] einordnen und sind farbig entsprechend gruppiert.

Statik sind die Informationen, die ohne die Ausführung der Software generiert werden können [5]. Darunter fallen die Metriken, die als Zahl zu jedem Softwareartefakt zugeordnet werden können und damit zueinander in Relation gesetzt werden können. Genannt wurden LOC, Komplexität, Coverage und Code-Violations. Letzteres sind beispielsweise Verletzungen von vereinbarten *Code-Conventions*. Die Informationen, die komplizierter zu visualisieren sind, stellen die Struktur und die Abhängigkeiten dar. Aus Abbildung 2.2 geht hervor, dass diese Informationen gleichzeitig am meisten von Interesse sind.

Dynamik beschreibt die Informationen, die zur Laufzeit einer Software generiert werden können [5]. Besonders oft wurden Ausführungszeiten von Softwareartefakten und Anzahl von Aufrufen genannt. Damit sind beispielsweise *Bottlenecks* identifizierbar. Auch die Darstellung der Laufzeitfehler einer fehlerhaften Software sind für deren Analyse wichtig. Die Resourcen-Auslastung ist dabei auch hilfreich, wirkt sich jedoch wenig auf das 3D Modell der Softwarevisualisierung aus, da diese Informationen parallel zur eigentlichen Software existieren.

Evolution beschreibt den zeitlichen Verlauf einer Software und stellt den Entwicklungsprozess in den Vordergrund [5]. Beispielsweise kann die Entwicklung statischer Metriken verfolgt werden. Mit der Evolution eines Themas ist gemeint, dass anhand

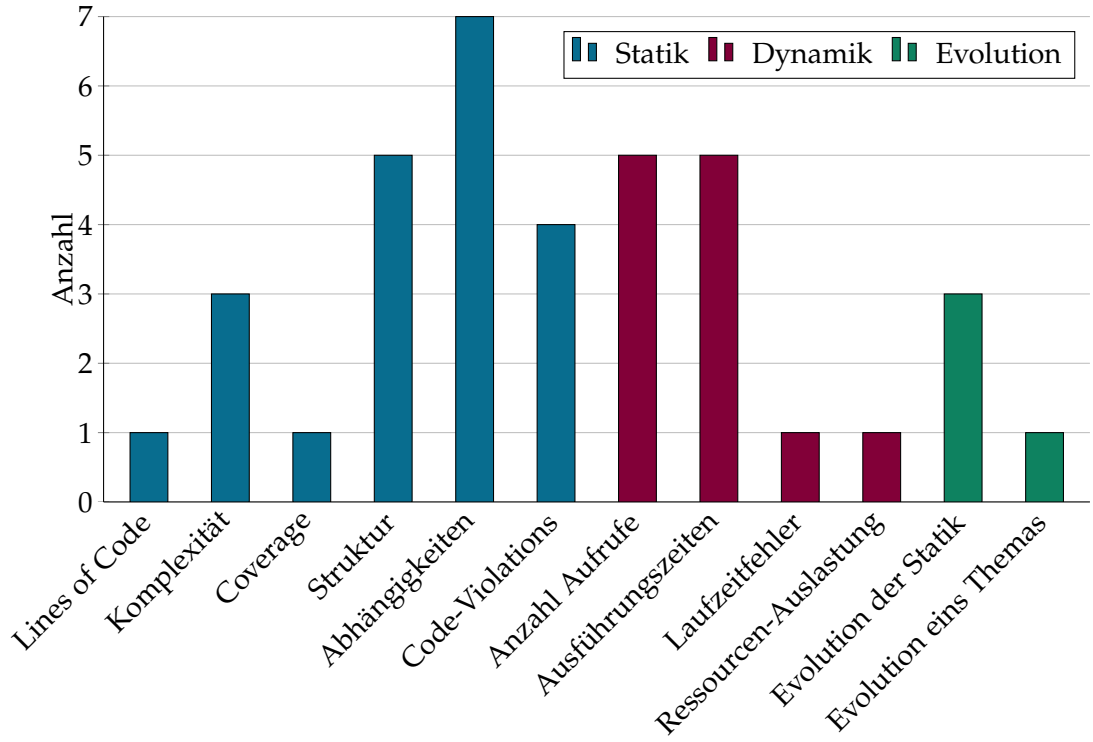


Abbildung 2.3 Von Mitarbeitern der QAware gewünschte Informationen

die Entwicklung eines bestimmten Themas nachverfolgt werden kann.

Aus den von den Mitarbeiten gewünschten Informationen und weitern Rahmenbedingungen wurde in [14] folgende Kriterien aufgestellt, anhand derer vier verschiedene Modelle der 3D Softwarevisualisierung bewertet wurden.

- Statische Metriken (z.B. Komplexität)
- Struktur
- Abhängigkeiten
- Dynamik (Primär Ausführungszeiten und Anzahl der Aufrufe)
- Evolution
- Habitability (vgl. Kapitel 2 Punkt 1)
- Drilldown
- Technische Machbarkeit

Bei dem Kriterium Drilldown wurde bewertet, wie gut eine Visualisierung ihre Informationen von High-Level, bis hin zu Details darstellen kann.

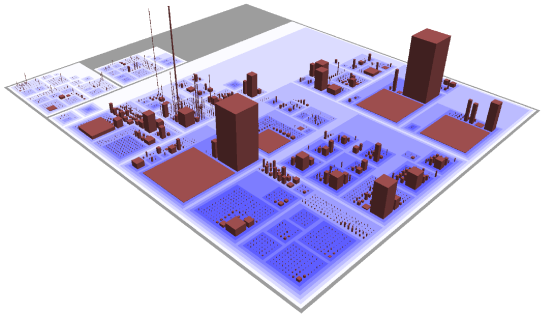
Bei der technischen Machbarkeit wurde berücksichtigt, ob eine existierende Softwarevisualisierung für die HoloLens verwendbar ist. In Abbildung 2.3 sind die untersuchten Alternativen abgebildet, darunter auch ein erster Entwurf von CodeLeaves.

CodeCity

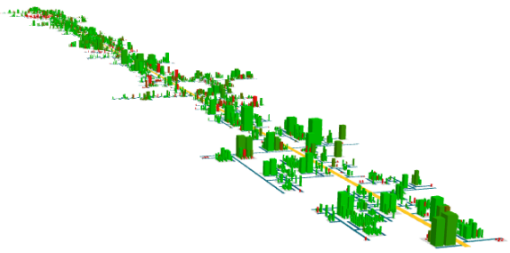
2007 stellten Wettel et al. CodeCity vor, die mithilfe der *Stadt-Metapher* dreidimensionale Städte visualisiert, in denen Klassen als Gebäude und Pakete als Stadtviertel dargestellt werden [19, 20, 21]. Für die Breite und Tiefe der Gebäude wurde für die Anzahl der Attribute (engl. *number of attributes (NOA)*) und für die Höhe die Anzahl der Methoden (engl. *number of methods (NOM)*) der visualisierten Klasse gewählt.

Die CodeCity ist als Konzept sehr durchdacht, bietet durch die Metapher gute Habitability und unterstützt die Darstellung der Evolution. Auch soll nach Wettel et al. die CodeCity die Analyse von Software im Vergleich zu herkömmlichen Analyse-Werkzeugen signifikant verbessern [21].

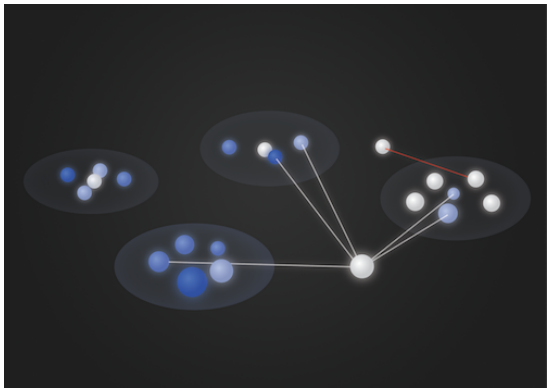
Jedoch unterstützt CodeCity keine Dynamik und die Abhängigkeiten sind nur als direkte Verbindungen darstellbar, was bei größeren Software-Systemen sehr unübersichtlich wird. Die verfügbaren statischen Metriken sind begrenzt und vor allem ist die Technologie Stand heute nicht mehr produktiv einsetzbar [14].



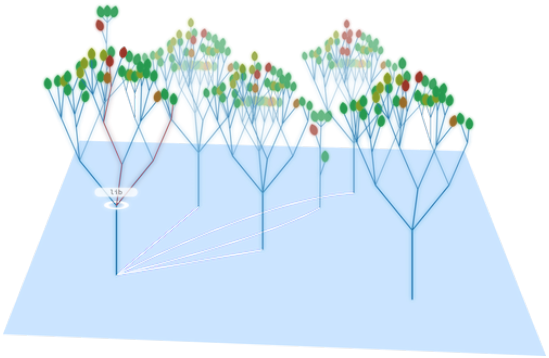
(a) CodeCity von ArgoUML [20]



(b) SoftVis3D mit Evostreet Layout



(c) Entwurf eines CodeUniverse



(d) Erster Entwurf von CodeLeaves

Abbildung 2.4 CodeLeaves und alternative Modelle

SoftVis3D

SoftVis3D greift das Konzept der CodeCity auf und visualisiert als Plugin für SonarQube[1] Projekte direkt im Browser. Durch die direkte Anbindung an SonarQube, ist SoftVis3D hoch konfigurierbar und kann alle Metriken darstellen, die auch in SonarQube zur Verfügung. Neben dem *District-Layout*, wie es in der CodeCity verwendet wird, unterstützt SoftVis3D darüber hinaus auch des *Evostreet-Layout*, das ursprünglich in [18] für die Evolution einer Software entworfen wurde. In diesem Layout, wie es in Abbildung 2.3b zu sehen ist, werden Pakete als Straßen dargestellt.

Die Evolution wird von SoftVis3D trotz Evostreet-Layout jedoch nicht unterstützt. Bei Die Abhängigkeiten wurde in früheren Versionen aggregiert dargestellt. Dafür wurden Pakete im Distrikt-Layout in übereinander liegenden Ebenen abgebildet und für Ψ (vgl. Definition 2.1) ein Hilfsgebäude benutzt, dass zu der darüberliegenden Ebene führt. Dadurch ging jedoch die Stadt-Metapher und die Übersichtlichkeit verloren. Die Dynamik kann in SoftVis3D ebenfalls nicht visualisiert werden. Die verwendete Technologie ist zwar mit *WebGL* für den Browser State of the Art, aber für die HoloLens aktuell noch nicht sinnvoll einsetzbar [14].

CodeUniverse

Im Zuge der der Studie [14] wurde eine weitere Metapher evaluiert. Ähnlich wie in der Arbeit [8, 3], wird die Software als Universum dargestellt. Die Softwareartefakte in Paketen gruppieren sich als Sterne in Galaxien. Statische Metriken können dann als Farbe und Größe der Sterne widergespiegelt werden. So können „weiße Zwerge“ bis hin zu „roten Riesen“ entstehen.

Das CodeUniverse ist für statische Metriken gut geeignet. Auch die Evolution ist mit der Entstehung von neuen Sternen und Galaxien gut vorstellbar. Die Struktur der Software ist zwar mit der Gruppierung der Sterne gegeben, aber weniger offensichtlich wie andere Konzepte. Bei der Visualisierung der Abhängigkeiten stößt das CodeUniverse aber an seine Grenzen. Durch direkte Verbindungen zwischen den Sternen lassen sich zwar Abhängigkeiten darstellen, aber bei großen Software-Systemen würde das schnell im Chaos enden. Auch in [3] wird beschrieben, dass eine übersichtliche Darstellung von Abhängigkeiten nur durch deren Aggregation erreicht werden kann. Deshalb wird in [3] ein Konzept entworfen, dass die Softwareartefakte mit einem *„hierarchischem Netz“* verbindet. Dieses ist nichts anderes als die vorhandene Baumstruktur der Software und hat mit der Metapher des Universums auch nichts mehr zu tun. Folglich wären wir wieder bei dem neuen Konzept CodeLeaves angelangt.

Vorteile von CodeLeaves gegenüber anderen 3D Softwarevisualisierungen

Die betrachteten Alternativen und weitere, haben gemein, dass sie zum einen Struktur, Dynamik und Evolution nicht vereinen. Zum anderen können Abhängigkeiten oder dynamische Aufrufe zwischen Softwareartefakten nicht ohne Verlust der Übersichtlichkeit angezeigt werden. CodeLeaves soll alle drei Kategorien der Softwarevisualisierung

[1] SonarQube ist eine open-source Plattform für statische Code-Qualität, `https://www.sonarqube.org/`

unterstützen und ist bei der Visualisierung der Struktur und der Abhängigkeiten den Alternativen überlegen. Durch die Baumstruktur, wie er auch in der Code-Base vorhanden ist, wird die Paket-Struktur eins zu eins wiedergegeben. Die aggregierten Abhängigkeiten lehnen sich an die Struktur an und beeinflussen diese nicht negativ. Durch das Wurzelgeflecht und die Spinnweben wird die Dreidimensionalität optimal ausgenutzt.

2.4 Anforderungen an CodeLeaves

Die Umfrage, die in Abschnitt 2.2 vorgestellt wurde, ergab einen gutes Stimmungsbarometer über die Wünsche im Bezug auf zu visualisierend Informationen. In diesem Abschnitt soll genauer auf die Anforderungen an einen Software-Wald eingegangen werden. Dazu wurden von der QAware zwei Experten der Softwareanalyse zu diesem Thema befragt, um Userstories für CodeLeaves zu sammeln.

Aus der dynamischen Analyse wurde der promovierende Performance-Analyst F. Lautenschlager befragt. Dieser erarbeitete im Zuge seiner Dissertation eine hochperformante Zeitreihendatenbank zur Speicherung und Auswertung von dynamischen Daten einer Software und ist Experte auf diesem Gebiet.

Als Experte der statischen Analyse wurde J. Weigend befragt. Dieser ist Chefarchitekt, Geschäftsführer und Mitgründer der QAware. Er studierte Informatik mit Schwerpunkt "Verteilte Systemeän der Hochschule Rosenheim und hält dort seit 2001 Vorlesungen [16].

Im Folgenden werden die Gespräche mit den beiden Experten unter Anwendung der Methoden aus [4] in Userstories und in den Tabellen **??** – **??** zu den dazugehörige Akzeptanzkriterien zusammengefasst. Zum Großteil überschneiden sich die Userstories mit der Umfrage aus Abschnitt 2.2, was für eine gute Übereinstimmung der eingeholten Informationen spricht.

Userstory 1: Dynamische Metriken

Als *Performance-Analyst*
möchte ich *Funktionsaufrufe, deren Antwortzeiten, Laufzeitfehler und Resourcen-Auslastung visualisiert haben*,
um *das Laufzeitverhalten der Software in einem bestimmten Zeitraum explorativ bewerten zu können.*

Tabelle 2.1 Akzeptanzkriterien zu Userstory 1

1	Antwortzeiten und Laufzeitfehler können als Metrik auf die Farbe der Blätter angewandt werden.

Fortsetzung auf nächster Seite...

Tabelle 2.1 – *Fortsetzung von vorheriger Seite*

2	Wenn ich ein Blatt auswähle, dann kann ich die genaue Zahl der gerade betrachteten Metrik sehen.
3	Wenn ich ein Blatt auswähle, kann ich sehen wohin von dieser Klasse Aufrufe hin- bzw. eingehen.
4	Die Dicke einer Verbindung (d.h. ein Stück Ast oder Wurzel) zeigt mir die Anzahl der Aufrufe zwischen Anfang und Ende der Verbindung.
5	Wenn ich eine Verbindung anklicke, dann kann ich sehen, welche Klassen von dieser Verbindung betroffen sind.
6	Wenn ich eine Verbindung anklicke, dann kann ich sehen in welche Richtung die meisten betroffenen Verbindungen gehen.
7	Wenn ich den Wald als ganzes auswähle, dann kann ich den Zeitraum auswählen, auf die sich die dargestellten Daten beziehen.

Userstory 2: Zustand der Software

Als *Systemverantwortlicher*
möchte ich *auf einen Blick den Zustand meiner Software erkennen*,
um *bei Bedarf agieren zu können*.

Tabelle 2.2 Akzeptanzkriterien zu Userstory 2

1	Wenn ich die Laubfarbe des Waldes betrachte, dann möchte ich eine möglichst gute Gesamtübersicht über die ausgewählte Metrik haben.
2	Wenn ich die Laubfarbe des Waldes betrachte, dann kann ich Ausreißer der ausgewählten Metrik gut erkennen.
3	Die Struktur der Bäume ist möglichst übersichtlich und hat keine Überlappungen.

Userstory 3: Kopplung und Struktur

Als *Softwarearchiteckt*
möchte ich *die Kopplung und Struktur einer großen Software sehen*,
um *die Zusammenhänge zu verstehen*.

Tabelle 2.3 Akzeptanzkriterien zu Userstory 3

1	Die Paketstruktur entspricht der Struktur der Bäume.
2	Wenn ich die Verbindungen zwischen einzelnen Bäumen oder Verzweigungen betrachte, dann erkenne ich durch die Dicke, wie stark die Pakete aneinander gekoppelt sind.
3	Wenn ich ein Blatt auswähle, dann kann ich die ein- bzw. ausgehenden Abhängigkeiten hervorheben.
4	Wenn ich ein Blatt auswähle, dann kann ich Abhängigkeiten direkt als Spinnweben darstellen.
5	Wenn ich eine Verbindung auswähle, dann kann ich die von den Abhängigkeiten betroffenen Klassen hervorheben.
6	Wenn ich eine Verbindung auswähle, dann kann ich sehen wie viele Abhängigkeiten in welche Richtung fließen.
7	Wenn ich auswähle, dass zyklische Abhängigkeiten hervorgehoben werden, können diese direkt als Spinnweben im gesamten Wald dargestellt werden.

Userstory 4: Code-Qualität

Als *Softwarearchitekt*
möchte ich *Code-Qualität auf die Struktur abbilden können*,
um *Anomalien und Qualitätsdefizite zu erkennen*.

Tabelle 2.4 Akzeptanzkriterien zu Userstory 4

1	Statische Metriken der Code-Qualität, wie z.B. Code-Coverage, kann auf die Farbe der Blätter angewandt werden.
2	Wenn ich die Laubfarbe des Waldes betrachte, dann kann ich Ausreißer der ausgewählten Metrik gut erkennen.

Userstory 5: Code-Qualität

Als *Softwarearchitekt*
möchte ich *einen intuitiven Drilldown haben*,
um *die Zusammenhänge auf verschiedener Pakte-Ebene zu verstehen.*

Tabelle 2.5 Akzeptanzkriterien zu Userstory 5

1	Wenn ich eine Verbindung auswähle, dann kann ich das repräsentierte Paket als neuen Waldboden festlegen.
2	Wenn ich die Laubfarbe des Waldes betrachte, dann kann ich Ausreißer der ausgewählten Metrik gut erkennen.

Bei der Entwicklung der Datenmodelle im nächsten Kapitel fließen die Userstories und Akzeptanzkriterien mit ein.

3 Datenmodell

3.1 Trennung in Schichten

Das Datenmodell für CodeLeaves nimmt einen zentralen Baustein ein, auf den sich die Implementierung stützt. Bei dem Entwurfsmuster sind einige Prinzipien zu beachten. Wie Buschmann et al. in [7] beschreibt, ist das Pattern *Layers* ein Grundprinzip bei Software-Architekturen. Durch die *Separation of concerns* werden einzelnen Schichten voneinander getrennt und können so später leichter ausgetauscht werden:

"Using semi-independent parts [...] enables the easier exchange of individual parts at a later date." [7]

Backend

Gerade bei der Visualisierung von abstrakten Daten, wie es eine Software ist, ist eine Schichtentrennung besonders wichtig. CodeLeaves hat das Ziel eine beliebige objektorientierte Software darstellen zu können. Da aber nicht für jede Programmiersprache CodeLeaves angepasst werden soll und die Informationen über die Software aus unterschiedlichen Quellen stammen können, ist hier die erste Schicht *Backend* sinnvoll. Im Backend Layer werden die Daten über eine Software aggregiert, seien es statische oder dynamische Informationen und in ein Sprach-agnostisches Format gebracht. Die Konnektoren, d.h. die Verbindungen zu den verschiedenen Datenquellen, können dann leicht ausgetauscht werden.

Wie in Abschnitt **??** bereits festgehalten, soll der Schritt des Transfers der realen Software in deren Repräsentation jedoch nicht Schwerpunkt dieser Arbeit sein. Das Datenmodell ist hingegen sehr wohl zu definieren, um die Daten darstellen zu können.

Application logic

Die Einteilung in Schichten geht nach der Transformation der Software in ein Meta-Modell aber noch weiter. CodeLeaves könnte neben Software prinzipiell auch jegliche anderen, hierarchisch strukturierten Informationen darstellen. Für die interne Datenhaltung wird deshalb in der Schicht *Application logic* das Datenmodell weiter abstrahiert und in ein Format gebracht, dass sich stark an der Baumstruktur der Informatik orientiert. So könnten später auch leicht gänzlich andere Daten angebunden werden.

Presentation

Die letzte Schicht ist die der *Presentation*. Das Datenmodell für hierarchische Informationen muss für die Darstellung der Bäume weiter verarbeitet werden. Zum Beispiel

muss für die rekursive Generierung der Bäume die Höhe der Knoten bekannt sein. Nur so kann das Paket mit der tiefsten Schachtelung auch sinnvoll als Verlängerung des Stammes dargestellt werden. Auch die Aggregation der Verbindungen zwischen einzelnen Blättern findet in diesem Schritt statt, sodass in den Zeichenalgorithmen, mit denen die Bäume in Unity generiert werden, keine fachliche Logik mehr benötigt wird. Das entstehende UI-Datenmodell wird nur zum Rendern in Unity verwendet und ist unveränderlich.

Das bedeutet auch diese Schicht könnte wieder ausgetauscht werden und unterschiedliche Zeichenalgorithmen könnten unterstützt werden.

In Abbildung 3.1 ist die Beziehung der Datenmodelle in den drei Schichten abgebildet.

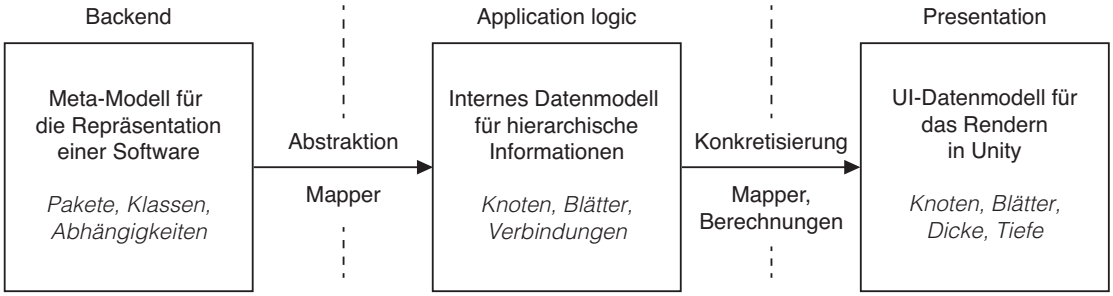


Abbildung 3.1 Schichten der Datenstrukturen

3.2 Software Meta-Modell

Vorhandene Meta-Modelle

CodeCity verwendet als Meta-Modell für Software das sogenannte *FAMIX*-Modell in Version 2.1. Dieses wird bis heute von *Moose*, dem Software-Analyse-Tool, auf dem CodeCity basiert, protegiert und ist heut in Version 3.0 verfügbar und in [**merrill1916moose**] beschrieben. Das Meta-Modell ist sehr umfangreich und beinhaltet 51 verschiedene Entities. Es wird in einem eigenen Dateiformat namens MSE, vergleichbar mit JSON, gespeichert. Da aber dynamische Informationen nicht im FAMIX-Modell enthalten sind, eignet sich das Format nicht um in CodeLeaves verwendet zu werden.

Ein Meta-Modell, das dynamische und statische Informationen vereint, existiert nach heutigem Wissensstand nicht.

Für eine reine dynamische Analyse ist das Tool *ExplorVis* sehr ausgereift [**fittkau2017software**]. Das im nachfolgende entwickelte Meta-Modell ist von Aspekten aus dem FAMIX- und dem ExplorVis-Modell inspiriert. Dabei soll das Modell aber möglichst simpel gehalten werden und nur das beinhalten, was CodeLeaves auch darstellen können soll.

Entwicklung eines eigenen Modells

Um uns vor Augen zu führen, welche Informationen dargestellt werden können sollen, rufen wir uns die Userstories aus Kapitel 2 in Erinnerung.

Aus den Userstories 1 und 4 ergeben sich folgende Informationen, die auch die Blattfarbe und damit auf jede Klasse angewandt werden können soll:

- Statische Metriken zur Code-Qualität
- Antwortzeiten
- Laufzeitfehler

Eine *Metric*-Entity ist also ein wichtiger Teil des Meta-Modells und wird mit einer *Class*-Entity assoziiert. Mit einem Kompositum-Pattern mit der Class- und *Paket*-Entity lässt sich die Struktur der Software beschreiben. Damit ist ein Teil von Userstory 3 abgedeckt.

Aus Userstory 1 und 3 lassen sich folgende mögliche Verbindungen ableiten:

- Statische Abhängigkeiten
- Dynamische Aufrufe

Beide Verbindungen lassen sich im Meta-Modell als *Assoziation*-Entity von einem Startpunkt zu einem Endpunkt modellieren. Statische Abhängigkeiten können sich unterscheiden. Es können Importe, Vererbungen, oder statische Aufrufe bzw. Zugriffe sein.

Durch die Expertengespräche ging hervor, dass sowohl bei Metriken wie Laufzeitfehler, als auch bei Verbindungen wie dynamische Aufrufe ein Bezug zum Code wichtig ist. Demnach müssen die Entities Association und Metric mindestens ein *Codesnippet* beinhalten wo zum Beispiel der Fehler in der Klasse aufgetreten ist.

Association associated with class?

Neben der Struktur, Metriken und den Verbindungen einer Software geht aus Userstory 1 hervor, dass es auch gilt die Ressourcen-Auslastung zu visualisieren. Diese Information wird parallel zu den bisher eingeführten Entities in der *Utilization*-Entity gespeichert.

Damit sind bereits fast alle Informationen adressierbar. Es fehlt lediglich noch, dass der Wald auch zu unterschiedlichen Zeitpunkten bzw. Zeiträumen dargestellt werden kann. Dafür werden alle Informationen als Zeitreihe in einer *Snapshot*-Entity gespeichert. So kann in der Zeit zurück gegangen werden und CodeLeaves kann somit auch die Evolution von Software unterstützen.

In Abbildung **??** sind die Entities des entwickelten Meta-Modells zu sehen.

Bei Betrachtung des Modells fällt auf, dass sich unter der Class-Entity noch die *Method*-Entity befindet. Diese hat die gleichen Assoziationen wie de Class-Entity. Je nachdem was als Softwareartefakt festlegt wird, kann auf Klassen-Ebene aufgehört werden, oder aber noch weiter bis auf Methoden-Ebene vorgegangen werden. Die Assoziationen von der Class- und der Method-Entity sind gleich. Jede Metric- und Association-Entity lässt sich einer Class-Entity zuordnen, kann aber bei feinerer Granularität auch einer Method-Entity zugeordnet werden.

Im Rahmen dieser Arbeit wird als Softwareartefakt eine Klasse festgelegt. Das Modell wäre mit der Mehtod-Entity aber leicht auf Methoden-Ebene erweiterbar.

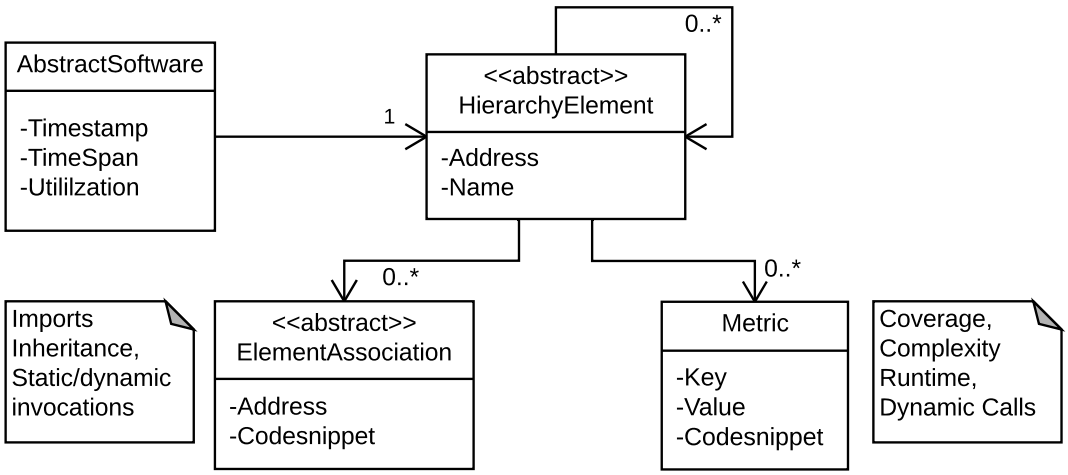


Abbildung 3.2 Meta-Modell einer Software

3.3 Internes Datenmodell

3.4 UI-Datenmodell

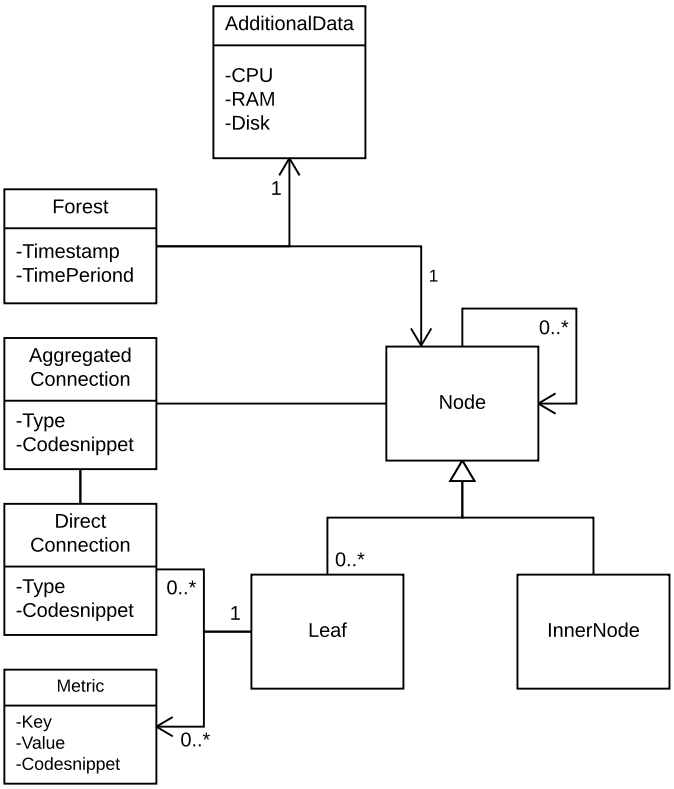


Abbildung 3.3 Internes Datenmodell

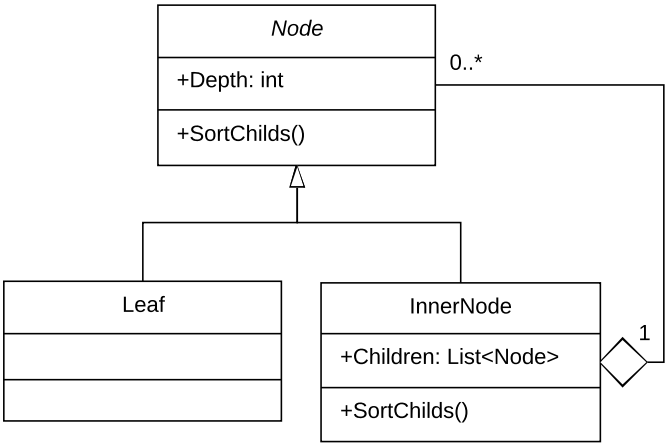


Abbildung 3.4 UI-Datenmodell

4 Modellierung

4.1 Grundlegender Ansatz

CodeLeaves steht und fällt mit der korrekten und übersichtlichen Darstellung der Baumstruktur. Durch Akzeptanzkriterium ist die Struktur der Bäume so zu generieren, dass so viele Blätter wie möglich gleichzeitig im Blickfeld des Betrachters ist. Auf der anderen Seite gilt es wegen Akzeptanzkriterium die Struktur übersichtlich und deswegen mit so wenig Überschneidungen wie möglich zu generieren.

link auf Akzeptanz

link auf Akzeptanz

In Abschnitt **??** und **??** werden deshalb zwei Algorithmen vorgestellt, die in Verbindung eine solche Darstellung der Baumstruktur ermöglicht. Für das Rendern in 3D müssen zunächst 3D Objekte gegeben werden, die die Knoten und Blätter repräsentieren. Dies wurde mithilfe der professionellen 3D Modellierungs-Software Maya[1] von Audodesk modelliert.

3D Objekte

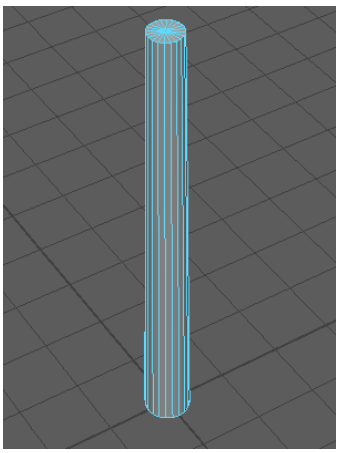
Die Bäume für CodeLeaves kommen mit den Objekten Kante und Blatt aus. Ein innerer Knoten wird der Baum Metapher folgend immer durch die Kante zum Elternteil bzw. zum Waldboden dargestellt. Ein Blatt - im Sinne der Begriffsdefinition von Abschnitt **??** - wird demnach vom dem dazugehörigen Blatt-Objekt und dem Kanten-Objekt zum Elternteil repräsentiert.

Kanten-Objekt Eine Kanten-Objekt ist ein vertikal ausgerichteter, nach oben leicht zulaufender Zylinder was die Abbildung **??** veranschaulicht. Die Pivots zur Position, Rotation und Skalierung wurden so gewählt, dass eine Manipulation des Objekts immer um den Mittelpunkt der Grundfläche der Kante erfolgt.

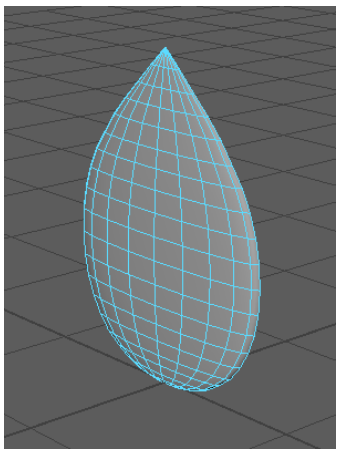
Blatt-Objekt Das Blatt-Objekt wurde in Anlehnung an ein echtes Blatt modelliert und ist in Abbildung **??** zu sehen.

Das Blatt-Objekt hat aufgrund der flachen Form die Eigenschaft, dass die sichtbare Fläche bei seitlicher Betrachtung deutlich geringer ist, als von vorne. Wir stellen uns einen Baum vor, bei dem der Betrachter nur die Kanten der Blätter sieht. Dadurch lassen sich die Farben der Blätter nur schwer miteinander vergleichen. Dies sollte aber aus jeder Postion möglich sein.

[1] https://www.autodesk.de/products/maya/overview



(a) Kante



(b) Blatt

Abbildung 4.1 3D Objekte für den Prototyp

Eine Abhilfe könnte ein Blatt-Objekt sein, dass in allen Dimensionen gleich dick ist. Dies würde aber eher an eine Knospe erinnern und schaut wenig natürlich aus. Deshalb wird dem Blatt-Objekt in Unity eine Script hinzugefügt, dass das Blatt zur Laufzeit immer in die Richtung des Betrachters rotieren lässt. Ein 3D-Objekt mit einer solchen Verhalten wird *Billboard* genannt. Mithilfe des Billboard-Verhaltens ist gewährleistet, dass der Nutzer von jedem Winkel aus immer eine optimale Übersicht über das Laubwerk besitzt.

Beschaffung von realistischen Beispieldaten

Für die Generierung der Struktur im Prototyp wurde mit einem Beispielprojekt gearbeitet. Das Projekt lautet *Air* und ist eines der größten Software-Projekte der QAware und wird seit rund 5 entwickelt.

In Tabelle **??** sind hinsichtlich der Struktur Eckdaten von Air angegeben.

Tabelle 4.1 Eckdaten des Beispielprojekts Air

Bäume	Knoten	Blätter	innere Knoten
40	6736	5373	1363

Alle Projekte von QAware werden in SonarQube überwacht. SonarQube bietet eine Web-API, durch die die Struktur eines Projekts und unterstützte statische Metriken abgefragt werden können. Diese API wird im Prototyp genutzt um die benötigten Informationen für Struktur der Software und Farben der Blätter abzufragen. Die Struktur wird zur Laufzeit rekursiv zusammengebaut, da SonarQube immer nur die direkten Kinder eines Softwareartefakts liefert.

Das Format von SonarQube wird dabei schon in das generische Software-Meta-Modell von CodeLeaves transferiert, was durch andere Daten später ergänzt werden kann. Da bei Air sehr viele HTTP-Requests entstehen, wird das entstandene Meta-Modell lokal in eine Datei geschrieben, sodass dieses als Beispiel-Datensatz für den Prototypen dient. Prinzipiell können aber auch dynamisch andere Projekte geladen werden.

Das Datenmodell des Backends wird in das Datenmodell der Application logic und weiter in das UI-Datenmodell transferiert.

Nun liegt eine Struktur vor, die es gilt zu Bäumen zusammen zu bauen. Der Grundlegende Ablauf des Algorithmus zur Generierung eines Baumes mit Kronenansatz K und dessen Kindern $\{(K_i) \mid i = 1, 2 \ldots n\}$ lässt sich wie folgt beschreiben:

(i) Instanziiere eine Kante E als Stamm des Baums.
(ii) Falls Knoten K ein Blatt ist, füge ein Blatt an E an.
(iii) Falls Knoten K ein innerer Knoten ist ist, finde für jedes Kind K_i eine passende Position, füge eine Kante E_i zwischen E und K_i an, setze K gleich K_i und E gleich E_i und und geht zu (ii).

Wie die Kinder eines Knotens sinnvoll verteilt werden, soll im Folgenden erarbeitet werden.

4.1.1 Verteilung von Kindern eines Elternteils

Ein wichtiger Bestandteil der Modellierung von Bäumen ist die Verteilung der Kinder eines Knotens. Es stellt sich zunächst die Frage auf Grundlage welcher geometrischen Figur die Kinder verteilt werden. In Frage kommt eine Kugel und eine Kreisfläche.

Verteilung auf einer Kugel

Werden die Kinder gleichmäßig auf der Oberfläche einer Kugel verteilt, wird der 3D Raum maximal ausgenutzt. Das Ergebnis käme einem *Fractal tree* sehr nahe. Diese Bäume werden rekursiv durch die Neigung der Aststücke konstruiert. Ein solcher Fractal tree ist in Abbildung **??** dargestellt. Das Konzept ließe sich problemlos in die Dreidimensionalität übertragen. Das Problem dabei ist, dass bei diesem Ansatz die Aststücke auch rekursiv verkürzt werden. Dadurch wird gewährleistet, dass die Äste nicht in den Boden wachsen.

Für CodeLeaves wären immer kleiner werdende Kanten wenig sinnvoll, da so eine Interaktion mit den kleiner werdenden Kanten und Blättern kaum möglich wäre. Auch die Überschneidungen, die in Abbildung **??** zu sehen sind, wären bei größeren Bäumen nur durch Neigung der Kanten nicht zu vermeiden. Die Kollisionen, die dadurch im 3D Raum entstünden, würden sich negativ auf die Übersichtlichkeit der Bäume auswirken und müssen deswegen vermieden werden.

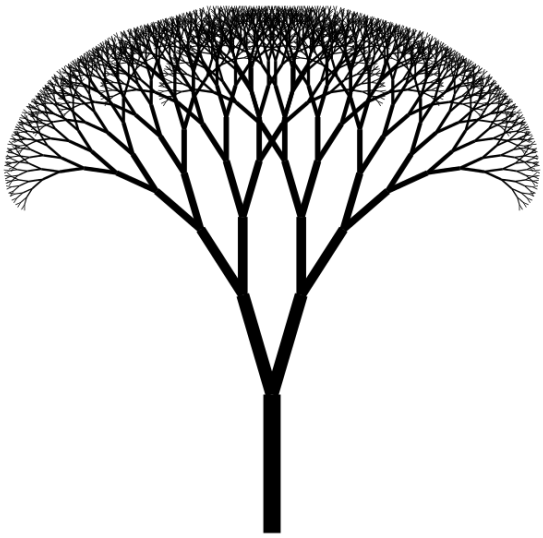


Abbildung 4.2 Fractal tree [**rocchini2017fractal**]

Verteilung auf einer Kreisfläche

Werden die Kinder stattdessen auf gleicher Höhe auf einer Kreisfläche überhalb des Elternteils positioniert, muss die Länge der Kanten entsprechend angepasst werden. Dadurch entstehen drei Vorteile.

Erstens befinden sich Software-Artefakte mit der gleichen hierarchischen Tiefe auch auf gleicher Höhe, was sich positiv auf die Übersichtlichkeit auswirkt.

Zweitens wird das Laubdach primär am Ende des Baumes auf einer Ebene angezeigt, was aus der Vogelperspektive eine gute Betrachtung der Metriken verspricht und gleichzeitig von der Seite nicht den Blick auf die Struktur des Baumes versperrt.

Drittens kann durch die Längenanpassung der Kanten Kollisionen von Ästen vermieden werden. Der Abstand zischen zwei Geschwister muss so groß gewählt werden, dass die Kanten der Nachfahren beider Geschwister nicht miteinander kollidieren.

Der Prototyp von CodeLeaves verwendet aus genannten Gründen eine Kreisfläche für die Verteilung von Kinder eines Knotens.

4.2 Verteilung von Blättern mit dem Sonnenblumen-Algorithmus

Betrachten wir nun zunächst den einfachen Fall eines inneren Knotens, der nur Blätter als Nachfahren besitzt. D.h. die Verteilung kann gleichmäßig erfolgen und muss die Breite nachfolgender Knoten nicht mit einbeziehen.

Dieser Fall tritt in realen Softwareprojekten häufig auf. In Air existieren 1154 solcher Knoten, was 85% aller inneren Knoten ausmacht. Nur $0,4\%$ aller inneren Knoten enthalten Blätter **und** weiter innere Knoten. Die Verteilung der Kinder eines Knotens mit inneren Knoten, wird im **??** behandelt.

Es wird ein Algorithmus gesucht, bei dem Punkte auf einer Kreisfläche gleichmäßig verteilt werden.

Dazu liefert uns die Natur ein schönes Beispiel. Die kleinen inneren Blüten der Sonnenblume, die sogenannten Röhrenblüten, die nach Befruchtung die Sonnenblumenkerne ausbilden, nutzen den Platz innerhalb der gelben Zungenblüten optimal aus [**zimmermann2017sonnenblume**]. Die spiralförmige Anordnung ist nicht nur ästhetisch, sondern folgt auch einem genauen Muster.

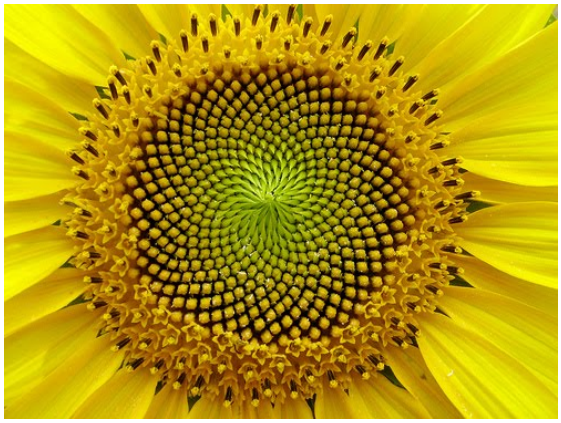


Abbildung 4.3 Spiralförmige Anordnung der Röhrenblüten einer Sonnenblume [**blender2017howto**]

Der Goldene Winkel

Ausgehend von dem Mittelpunkt der Sonnenblume ist jede nachfolgender Blüte um rund 137.5 um den Mittelpunkt rotiert. Dieser sogenannte *Goldener Winkel* entsteht durch die Teilung des Vollkreises durch den *Goldenen Schnitt*. Der goldene Schnitt ist ein Teilungsverhältnis das oft bei Größenverhältnissen von einfachen geometrischen Figuren vorkommt und ist wie folgt definiert:

$$\Phi = \frac{a}{b} = \frac{a+b}{a} = \frac{1+\sqrt{5}}{2} \approx 1.618 \tag{4.1}$$

Wird der Vollwinkel durch den Goldenen Schnitt geteilt, entsteht folgender Winkel:

$$\frac{2\pi}{\Phi} \approx 222.5° \tag{4.2}$$

Die Ergänzung zum Vollwinkel ist der Goldene Winkel:

$$2\pi - \frac{2\pi}{\Phi} = \frac{2\pi}{\Phi^2} \approx 137.5° \tag{4.3}$$

Dieser hat die Eigenschaft, dass er aufaddiert nie den selben Winkel ergibt. Diese Eigenschaft machen sich viele Pflanzen zu Nutze und kann auch in CodeLeaves Anwendung finden.

Berechnungen für einen neuen Knoten

Mit dem Vielfachen des goldenen Winkels ist der Winkel φ gegeben, um den eine neue Kante um die y-Ache rotiert wird. Für die Rotation und Skalierung einer Kante sind aber noch andere Größen zu berechnen.

Wird eine Kante mit der Länge l an einen Knoten angefügt, um die x-Achse mit dem Winkel θ geneigt und anschließend um die y-Achse mit den Winkel φ rotiert, entsteht ein Kugelkoordinatensystem mit der Position des Elternteils als Ursprung O, der y-Achse als Polachse, der Position des n-ten Kindknoten als Punkt P, dem *Polarwinkel* θ und dem *Azimutwinkel* φ[12].

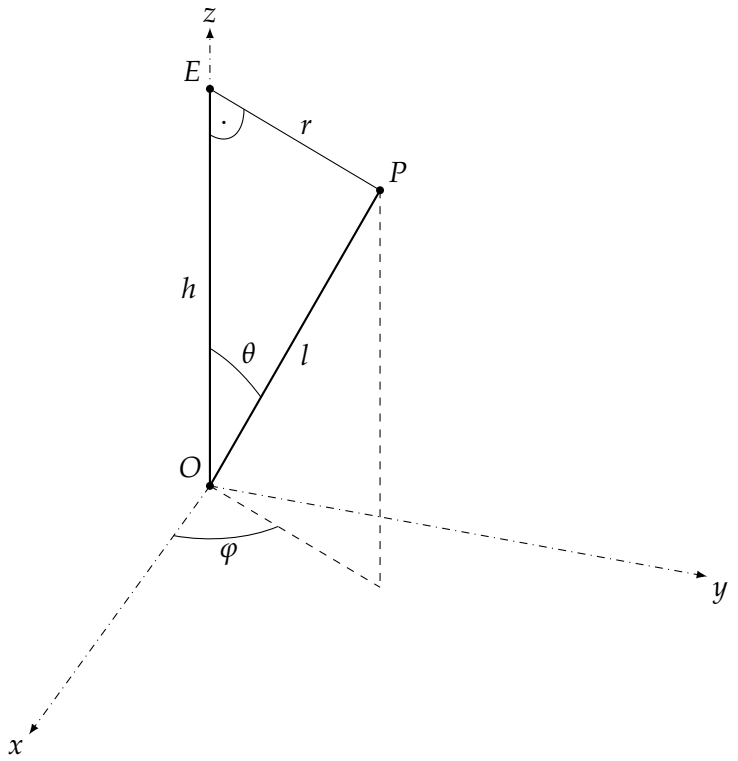


Abbildung 4.4 Kugelkoordinatensystem mit den zu berechnenden Größen für das Hinzufügen eines neuen Knotens

Diese Größen sind in Abbildung **??** dargestellt, wobei die Höhe h als Standardhöhe einer Kante gegeben gilt und E mit $(0, h, 0)$ die Position des 0-ten Kindes ist.

Der Abstand d_n zwischen dem Punkt E und P berechnet sich für den n-ten Knoten nach [**vogel1979better**] mit

$$d_n = c\sqrt{n}, \tag{4.4}$$

wobei c eine Konstante ist und den Abstand der Kindknoten untereinander beeinflusst.

Die Rotation um die y-Achse φ ist wie oben hergeleitet das n-te Vielfache vom Goldenen Winkel:

$$\varphi = \frac{2\pi \cdot n}{\Phi^2} \tag{4.5}$$

Der Polwinkel θ lässt sich mithilfe von h und r berechnen.

$$\theta = tan^{-1}\left(\frac{r}{h}\right) \tag{4.6}$$

Für die Positionierung des Kindknotens muss der Punkt P von den Kugelkoordinaten (l, θ, φ) in das kartesische Koordinatensystem umgerechnet werden. Dies wird durch folgende Formel erreicht:

$$\vec{P} = \begin{pmatrix} \sin(\varphi) \cdot \sin(\theta) \cdot l \\ \cos(\theta) \cdot l \\ \cos(\varphi) \cdot \sin(\theta) \cdot l \end{pmatrix} \tag{4.7}$$

Damit ist alles für den Algorithmus gegeben, der Kinder eines Knotens mit gleichmäßigem Abstand auf eine gegebene Höhe verteilt und die Kanten entsprechend rotieren und skalieren kann

In Unity und C# ist dieser Algorithmus in vereinfachter Form in Listing 4.1 zu sehen ist.

Listing 4.1: Hinzufügen von Kinder eines Knotens

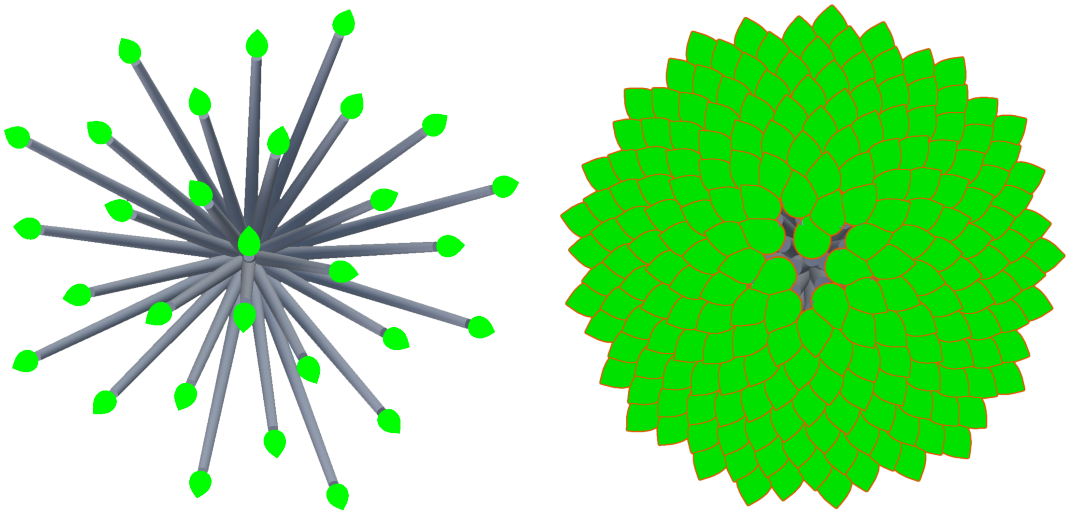
```
for (var i = 0; i < node.Children.Count; i++)
{
    var d = TreeGeometry.CalcDistance(i);
    var phi = TreeGeometry.CalcPhi(i);
    var theta = TreeGeometry.CalcTheta(DefaultEdgeHeight, r);
    var l = TreeGeometry.CalcEdgeLength(DefaultEdgeHeight, theta);
    var pVec = TreeGeometry.CalcNodePosition(l, theta, phi);

    AddEdgeObject(l, theta, phi);
    AddEmptyNodeObject(pVec);
}
```

Das Ergebnis ist in Abbildung **??** zu sehen. Auf der linken Seite (**??**) sind 30 Blätter verteilt worden. Die Ähnlichkeit zur Verteilung der Röhrenblüten der Sonnenblume wird auf der rechen Seite (**??**) mit der Extremsituation mit 200 Blättern und kleinem c sichtbar.

4.3 Verteilung innerer Knoten mit Circle-Packing

Die Verteilung mit dem Sonnenblumen-Algorithmus beruht auf der Tatsache, dass die Kinder des Knotens mit gleichmäßigem Abstand zueinander verteilt werden können.



(a) 30 Blätter mit $c = 0.05$ **(b)** 200 Blätter mit $c = 0.01$

Abbildung 4.5 Sonnenblumen-Algorithmus zur Verteilung von Blättern

Dies ist bei inneren Knoten, die wiederum innere Knoten als Kinder besitzen, nicht gegeben. In diesem Fall muss die Breite der Kinder mit berücksichtigt werden, sodass die Äste der einzelnen Kinder nicht kollidieren.

Für die Verteilung der inneren Knoten mit einer minimalen Höhe von 1, führen wir ein weiteres Merkmal ein.

Definition 4.1: Kreis und Radius eines inneren Knotens

Der Radius eines inneren Knotens ist der Radius des Kreises, aus dem alle Nachfahren des Knotens von oben betrachtet nicht hinaus ragen.

Bei einem inneren Knoten, dessen Kinder nach dem Sonnenblumen-Algorithmus verteilt wurden, kann dessen Radius leicht berechnet werden. Der Radius r des inneren Knoten ist gleich dem Abstand a_n des n-ten Kindes, der nach Formel **??** mit $a_n = c\sqrt{n}$ berechnet werden kann.

Werden nun alle inneren Knoten so verteilt, dass sich von oben betrachtet die Kreise der Kinder nicht überschneiden, sind alle Äste eines Baumes kollisionsfrei verteilt und alle Blätter sind von oben sichtbar.

Das Problem der Verteilung von inneren Knoten kann demnach auf 2D reduziert werden.

Es wird ein Algorithmus gesucht, der Kreise unterschiedlicher Größe auf möglichst kleinem Raum überschneidungsfrei positioniert.

Diese Aufgabenstellung wird vom sogenannten *Circle-Packing* gelöst.

Dazu hat Wang in [**wang2006visualization**] 2006 einen Algorithmus entworfen, der seitdem von vielen adaptiert worden ist. Zum Beispiel in der populären JavaScript Library *d3.js* zur Datenvisualisierung verwendet Bostock, der Schöpfer von d3.js diesen Algorithmus [**bostock2017abetter**].

Der Algorithmus muss noch an die Gegebenheiten von CodeLeaves angepasst werden. Bei Wang's Version startet der Algorithmus bereits mit drei tangentialen Kreisen. Der Mittelpunkt, um den zusätzliche Kreise positioniert werden, befindet sich zwischen den drei initialen Kreisen. Bei CodeLeaves sind aber nicht zwangsläufig bei jedem Knoten mindestens drei Kindknoten zu verteilen und als Mittelpunkt für die Verteilung der Kreise ist der Mittelpunkt des 0-ten Kreises sinnvoll, da sich so immer ein Hauptstamm für einen Baum ergibt.

Ist nur ein Knoten zu verteilen, wird der Mittelpunkt seines Kreises von oben gesehen (in der XZ-Ebene) in den Ursprung gelegt. Der Kreis eines zweiten Knotens wird in einem zufälligen Winkel tangential zum Kreis des ersten Knotens positioniert. Bereits bei einem dritten Kreis kann Wangs's Algorithmus Anwendung finden.

Es wird eine sogenannte *Front-Chain* verwendet, eine zyklische Liste, die alle Kreise enthält, an die potentiell ein neuer Kreis angefügt werden könnte. Die Front-Chain ist in Abbildung **??** als dickere Linie dargestellt. Bei der Positionierung des ersten bzw. des zweiten Kreises werden diese zur Front-Chain hinzugefügt.

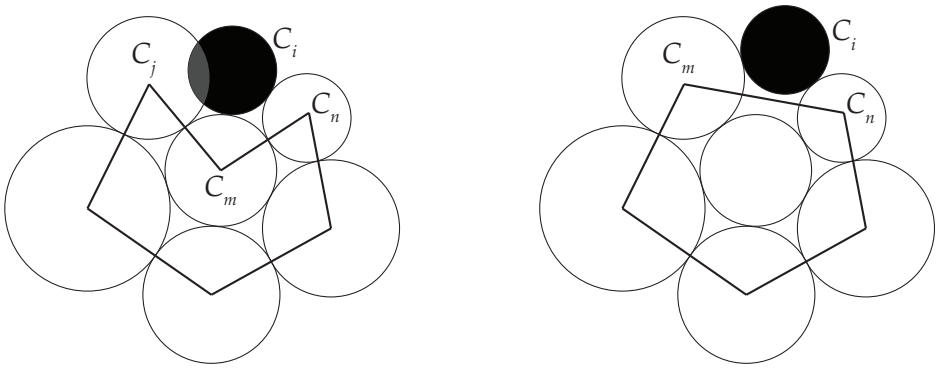
Im Ganzen ist der leicht modifizierte Algorithmus für eine Menge von Kreisen $\{C_i \mid i = 1, 2 \cdots n\}$ im Folgenden beschrieben:

Abgewandelte Form von Wang's Circle-Packing-Algorithmus (nach [wang2006visualization])

(i) Falls $n >= 1$ setzte den Mittelpunkt von C_0 in den Ursprung und füge C_0 zur Front-Chain hinzu.
(ii) Falls $n >= 2$ setzte den Mittelpunkt von C_1 so, dass C_1 in einem zufälligen Winkel tangential zu C_0 ist und füge C_1 zur Front-Chain hinzu.
(iii) Für $1 < i < N$ suche C_m, der Kreis dessen Mittelpunkt am nächsten zum Ursprung ist. C_n ist der Kreis in der Front-Chain nach C_m.
(iv) Berechne den Mittelpunkt von C_i so, dass er tangential zu C_m und C_n ist.
(v) Suche einen Kreis C_j der sich mit C_i überschneidet.
(vi) Falls der C_j nicht existiert, setze C_i gleich C_{i+1} und gehe zu (iii).
(vii) Falls C_j in der Front-Chain näher an C_m als an C_n ist, lösche alle Kreise aus der Front-Chain, die zwischen C_j und C_n liegen. Setze C_m gleich C_j und gehe zu (iv).
(viii) Falls C_j in der Front-Chain näher an C_n als an C_m ist, lösche alle Kreise aus der Front-Chain, die zwischen C_m und C_j liegen. Setze C_n gleich C_j und gehe zu (iv).

In Abbildung **??** ist ein Ausschnitt aus dem Algorithmus visualisiert. Auf der linken Seite (**??**) ist Schritt (iv) dargestellt, wobei C_j existiert und und näher an C_m, als an C_n liegt. Demnach werden alle Knoten zwischen C_j und C_n aus der Front-Chain entfernt und C_i wird erneut platziert. Das Resultat ist auf der rechten Seite (**??**) zu sehen.

Werden die Knoten vor der Durchführung des Algorithmus nach deren Radien sortiert, befindet sich der Knoten mit dem größten Kreis genau über dem Stamm



(a) Platzieren eines neuen Kreises

(b) Anpassung der Front-Chain und erneutes Platzieren

Abbildung 4.6 Wang's Circle Packing Algorithmus (nach [**bostock2017abetter**])

und nach außen hin werden die Äste kleiner, was einem natürlichem Wuchs am nächsten kommt. Durch die zufällige Positionierung des 1-ten Knotens, wächst der Baum nicht bei jedem Knoten „in die gleiche Richtung" was das natürliche Bild des Waldes ebenfalls fördert.

Der oben beschriebene Algorithmus setzt in (ii) und in (iv) die Berechnung eines Mittelpunktes voraus, sodass dessen Kreis tangential zu einem bzw. zwei weiteren Kreisen ist.

Bei dem 1-ten Kreis mit Radius r_1 und Mittelpunkt M_1, der nur tangential zu dem 0-ten Kreis mit r_0 und $M_0 = (0, 0)$ ist und mit dem zufälligen Winkel Polarwinkel α platziert wird, ergibt sich für die XZ-Ebene folgende Koordinate:

$$\vec{M_1} = \begin{pmatrix} x = \cos(\alpha) \cdot (r_1 + r_2) \\ y = \sin(\alpha) \cdot (r_1 + r_2) \end{pmatrix} \tag{4.8}$$

Bei allen weitern Kreisen C_i ist die Berechnung komplexer. Der Sachverhalt ist in Abbildung **??** abgebildet. Gesucht ist M_3. Gegeben sind die Mittelpunkte M_1 und M_2 sowie die Radien der drei Kreise.

M_3 befindet sich genau an einem Schnittpunkt der beiden Kreise $\{X \in XZ \mid \overline{M_1X} = r_1 + r_3\}$ und $\{X \in XZ \mid \overline{M_2X} = r_2 + r_3\}$, die in Abbildung **??** gestrichelt veranschaulicht sind. Diese Schnittpunkte lassen sich mithilfe von Vektorrechnung wie folgt errechnen:

$$\vec{M_{3_{1/2}}} = \vec{M_1} + \vec{D_1} \pm h \cdot \vec{e} \tag{4.9}$$

Dabei ist e ein zu $M_2 - M_1$ orthogonaler Einheitsvektor, für h gilt

$$h = \sqrt{r_1^2 - |d_1|^2} = \sqrt{r_2^2 - |d_2|^2} \tag{4.10}$$

und D_1 lässt sich durch Umformen der Gleichung **??** mit

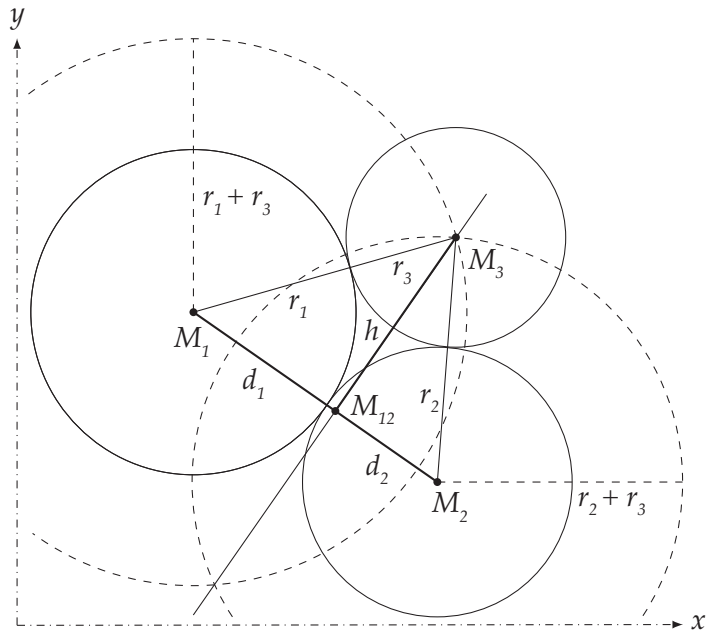


Abbildung 4.7 Berechnung der Position eines Kreises, tangent zu zwei anderen Kreisen

$$\vec{D}_1 = \frac{1}{2} \cdot \left(\frac{r_1^2 - r_2^2}{|\vec{M}_2 - \vec{M}_1|} + 1 \right) \tag{4.11}$$

darstellen.

Mit diesen Gleichungen kann der Circle-Packing Algorithmus für Kinder eines Knotens angewandt werden. Für die Positionierung der Knoten im dreidimensionalen Raum kommt für die Y-Koordinate jeweils noch die Länge der Kante des 0-ten Geschwisters hinzu. Die Kanten der Geschwister müssen entsprechend deren Position angepasst werden

Knoten mit gemischten Kindern

Für den Fall, dass ein Paket innere Knoten **und** Blätter enthält, kann der Circle-Packing-Algorithmus ebenfalls angewandt werden. Dafür muss lediglich einem Blatt ein Radius zugewiesen werden. Mit der Verwendung der Konstante c aus dem Sonnenblumen-Algorithmus als Radius, haben die Blätter bei beiden Verteilungs-Algorithmen den gleichen Abstand zueinander. Lediglich die Ausbreitung der Blätter um den Ursprung erfolgt weniger natürlich, als beim Sonnenblumen-Algorithmus.

Hierarchische Anwendung

Für die Anwendung auf mehreren Ebenen ist nach der Verteilung von Kindern eines Knotens K auf Ebene n die Berechnung des Radius von K nötig, dass in Ebene $n-1$ die Generation von K verteilt werden kann.

Der Radius des Kreises von Knoten K (mit Mittelpunkt an der Position von K), der alle Kreise der Kinder von K umschließt und mindestens zu einem Kreis der Kinder tangential ist, ist der Abstand vom 0-ten Kind zum n-ten Kind zuzüglich des Radius des n-ten Kindes. Diese Methode ist nicht der kleinste umschließende Kreis, reicht für die Generierung von überschneidungsfreien Bäume jedoch aus. Wir nennen den Kreis dieser Methode den *größten umschließenden Kreis*.

Für die vollständige hierarchische Anwendung des Circle-Packing wird für jeden Knoten rekursiv das Circle-Packing durchgeführt und anschließend der eigene Radius gesetzt. Abbruchbedingung für die Rekursion ist, dass der Knoten ein Blatt, oder ein innerer Knoten mit ausschließlich Blättern ist. In letzterem Fall greift der Sonnenblumen-Algorithmus.

Das heißt der Algorithmus fängt mit dem Knoten mit der größten Tiefe an und arbeitet sich über Geschwister und Elternteile bis hin zum Kronenansatz fort, bis alle Knoten eines Baumes verteilt sind.

Verteilung von Bäumen

Die Verteilung der einzelnen Bäume kann durch das Circle-Packing sehr leicht umgesetzt werden. Nachdem der Kronenansatz einen Radius besitzt, ist damit auch die Radius des ganzen Baumes bekannt. Damit können die Bäume beliebig platziert werden, ohne dass sich deren Äste überschneiden würden. Es wäre zum Beispiel eine Verteilung in einem Gitter möglich, was dann an eine Baumschule mit Baumreihen erinnern würde. Es ist jedoch auch ganz einfach möglich die Bäume ebenfalls mit Circle-Packing anzuordnen. Damit entsteht ein runder Wald und die Bäume nehmen möglichst wenig Platz ein.

Mit den beiden beschriebenen Algorithmen können Bäume garantiert überschneidungsfrei generiert und verteilt werden. Ein Auszug aus einem so entstanden Wald ist in Abbildung **??** zu sehen. Die Kreise der Knoten wurden als dunkle Schattierungen visualisiert, um das Circle-Packing zu veranschaulichen. Der Waldboden ist in blau dargestellt. Aus der Vogelperspektive ist gut zu erkennen, wie die Blätter innerhalb ihrer Eltern mithilfe des Sonnenblumen-Algorithmus sehr gleichmäßig verteilt werden.

Auch das Circle-Packing ist gut zu sehen. Jedoch lässt sich feststellen, dass durch die Wahl größten umschließenden Kreises Platz "verschenkt"wird. Besonders in der Mitte der Abbildung wird dies durch die große Fläche, in denen keine weiteren Knoten platziert sind, deutlich.

Um das zu umgehen müsste anstelle des größten umschließenden Kreises, der minimal umschließende Kreis verwendet werden. Dies ist ein weiteres mathematisches Problem und als Teil des *Appollonius* Problem bekannt [**dergiades2007soddy**].

Vielleicht noch auf appollonius kreise eingehen

Für den Prototyp ist die Umsetzung des kleinsten umschließenden Kreises für CodeLeaves aber nicht essentiell. Entscheidend ist, dass die Struktur der visualisierten Software ohne Überschneidungen dargestellt wird, was auch mit dem größten umschließenden Kreis möglich ist. Die Umstellung auf den kleinsten umschließenden Kreis ist deshalb eine weiterführende Aufgabe.

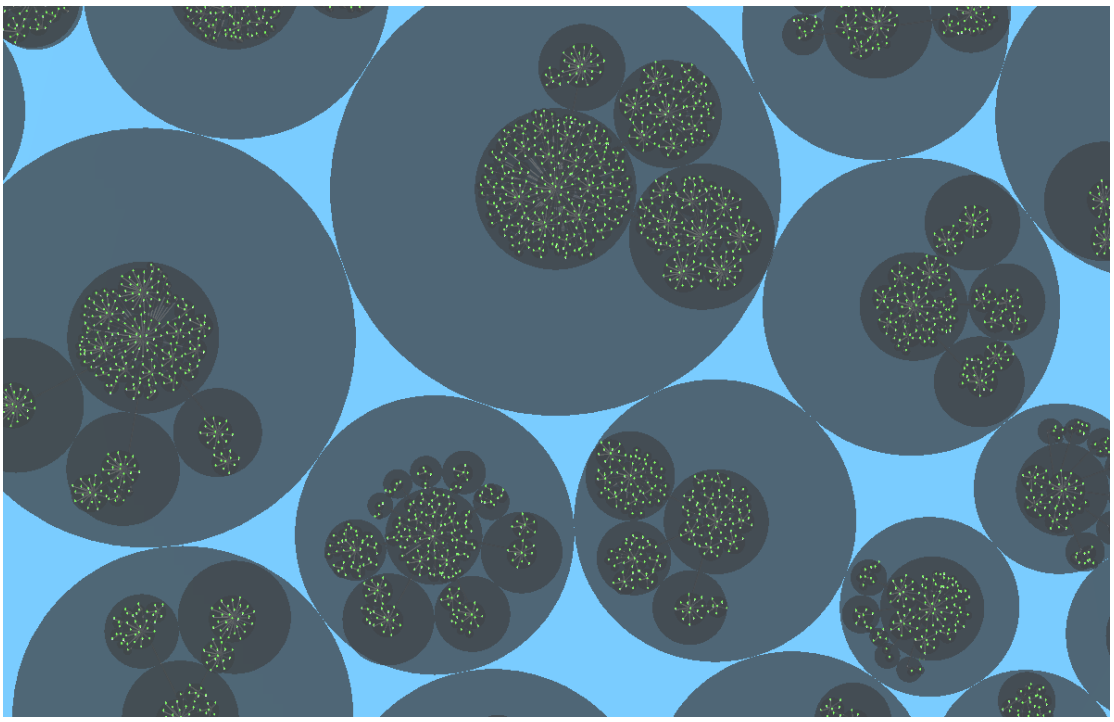


Abbildung 4.8 Circle-Packing in CodeLeaves

4.4 Verwendete Algorithmen in der Praxis

Für die Praxis in Unity ist bei der Verteilung der Bäume noch zu beachten, dass deren Radien erst verfügbar sind, nachdem der Circle-Packing-Algorithmus für alle Bäume vollständig durchlaufen wurde. Das heißt, dass bei der Generierung der einzelnen Bäume noch nicht bekannt ist, an welcher Position die Bäume letztendlich gesetzt werden müssen und zunächst zum Beispiel an der gleichen Stelle generiert werden müssen. Das bedeutet wiederum, dass bei asynchroner Programmierung und großen Projekten wie Air zunächst die Bäume überlagernd gerendert werden und dies auch durchaus für einige Sekunden für den Nutzer sichtbar ist, bevor sich die Bäume verteilen.

Bei Air dauert die Generierung der 40 Bäume im Unity Editor auf einer Hardware mit 2.3 GHz Intel Core i7 CPU und 16 GB 1600 MHz DDR3 RAM bei 10 Messungen zwischen 6,5 und 6,8 Sekunden. Auf der HoloLens stehen keine genauen Messdaten zur Verfügung, die Generierung dauert aber noch etwas länger.

Soll vermieden werden, dass der Nutzer zunächst die überlagernde Generierung mit ansieht, dürften die Bäume erst sichtbar werden, nachdem alle Bäume fertig generiert und verteilt worden sind. Jedoch ist zu beachten, dass währenddessen dem Benutzer angezeigt wird, dass der Aufbau des Waldes im Hintergrund läuft. Mehr zu diesem Thema ist im nächsten Kapitel zu finden

Ein weiterer zu beachtender Punkt ist, dass bei beschriebener Vorgehensweise einzelne Blätter theoretisch auf die gleiche Position gelangen könnten. Grund dafür ist, dass

der Radius eines Knotens mit ausschließlich Blätter gleich a_n gesetzt wurde. Daraus resultiert, dass bei zwei Geschwister solcher Knoten deren n-ten Kinder genau auf dem Schnittpunkt der Kreise liegen könnten. In diesem Fall würden sich die Blätter überschneiden. In der Praxis ist dies jedoch sehr unwahrscheinlich, besonders wenn für den Azimutwinkel des Sonnenblumen-Algorithmus ein zufälliger initialer Wert gesetzt wird, auf den dann ein Vielfaches des goldenen Winkel aufaddiert wird. In unserem Beispielprojekt mit 5373 Blättern, trat dieser Fall nicht auf.

5 Interaktionskonzept

5.1 Interaktion in der AR

5.1.1 Eingabemöglichkeiten

In den frühen 70er-Jahren wurden erste Modelle der Computermaus entwickelt und mit Apple's Lisa aus dem Jahr 1983 wurde sie zum Markterfolg. Heute ist die zeigerbasierte Interaktion mit Computer nicht mehr weg zu denken.

Neben der zeigerbasierten Eingabe hat sich mit dem Einzug von Smartphones und Tablets die touchbasierte Eingabe etabliert. Diese kann für Monitor basierte AR verwendet werden. Bei dieser Art von Mit dem AR-Kit von Apple ab der iOS Version 11 sind solche Applikationen einfach zu realisieren.

Für die AR mit Head Mounted Displays (HMD) Devices, wie es die HoloLens ist und die wie der Name schon sagt, vom Nutzer getragen werden, ergeben sich neue Möglichkeiten mit Anwendungen zu interagieren. Auch wenn z.B. die HoloLens die Eingabe durch eine Bluetooth-Maus unterstützt, bieten sich in AR zusätzliche Eingabemöglichkeiten an.

Es ergeben sich folgende Eingabemöglichkeiten für HMD-Devices:

1. Blickrichtung
2. Gesten
3. Sprache
4. Controller

Blickrichtung

Bei der HoloLens wird die Blickrichtung mit *Gaze* bezeichnet und ist ein elementarer Bestandteil der Bedienung der HoloLens [**windows2017interaction**]. Der Gaze wird im Normalfall mithilfe eines Cursers visualisiert, der der Kopfbewegung des Nutzers folgt und sich realen und virtuellen Objekten anschmiegt.

Gesten

Nachdem bei HDM-Devices Hologramme im realen Raum platziert werden, ist Gestensteuerung eine intuitive Eingabemöglichkeit. Beispielsweise muss die Möglichkeit der Auswahl, Platzierung, Skalierung oder Rotation von Hologrammen ermöglicht werden.

Allgemein kann zischen zwei Arten von Gesten unterschieden werden.

Diskrete Gesten sind solche, in denen die Ausführung der Geste einen binären Status besitzt. D.h. die Ausführung der Geste ist die Information selbst und trägt keine weiteren Informationen. Diese Gesten lassen sich mit einem Klick einer Computermaus vergleichen.

Kontinuierliche Gesten sind solche, bei denen das Ausmaß der Geste eine Rolle spielt. Das Ausmaß bestimmt die Größe der Ausgabe. Vergleichen wir diese Gesten mit einer Computermaus, wäre das die Mausbewegung.

Theoretisch sind beliebig viele Gesten denkbar. Zum Beispiel könnte für das Rotieren von Objekten die Rotation der Hände verwendet werden. Mit Produkten wie zum Beispiel Kinetic[1] können solche Gesten auch erkannt werden. Bei der HoloLens sind die unterstützen Gesten jedoch stark begrenzt. Es existieren insgesamt drei verschiedene Gesten, die von der HoloLens als solche identifiziert werden können. Die sogenannte *Bloom*-Geste ruft das Windows Menü auf. Sie wird ausgeführt, indem der Nutzer alle Fingerspitzen zusammen führt und dann die Hand öffnet und eine aufgehende Blume imitiert.

Als weitere diskrete Geste kann der *Air Tap* verwendet werden. Dieser ist in Abbildung **??** veranschaulicht. In Verbindung mit dem Gaze können so Hologramme angeklickt werden.

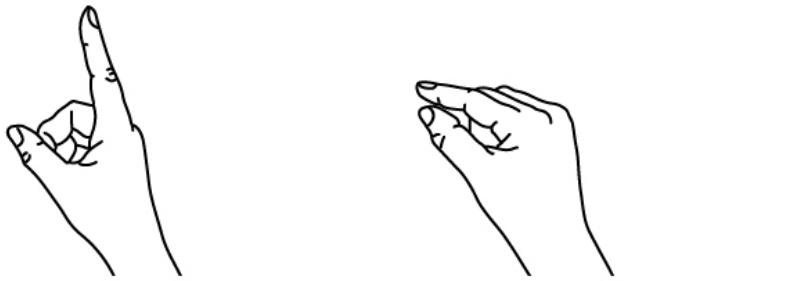


Abbildung 5.1 Air Tap **[windows2017gesture]**

Mit der *Manipulation* Geste stellt Microsoft eine kontinuierliche Geste zur Verfügung. Wird ein AirTap gehalten und die Hand anschließend bewegt, so kann als Ausgabe die Positionsänderung der Hand verwendet werden. Damit ist beispielsweise ein Drag and Drop von Hologrammen möglich.

Die HoloLens kann zwischen linker und rechtem Hand unterscheiden. Das bedeutet, dass auch eine Interaktion mit der Verwendung von beiden Händen möglich ist.

[1] `https://developer.microsoft.com/de-de/windows/kinect`

Sprache

Neben den Gesten kann auch durch Sprachsteuerung mit Hologrammen interagiert werden. In einer Applikation können beliebige Sprachbefehle definiert werden, die dann von der HoloLens automatisch erkannt werden.

Da die Interaktion mit Gesten auf Dauer für die Arme ermüdend sein kann, bietet sich an, die Spracherkennung zur Unterstützung der Gestensteuerung zu verwenden. Beide Eingabemethoden können auch gut miteinander kombiniert werden. So kann z.B. mithilfe von Sprachbefehlen zwischen unterschiedlichen Interaktionsmodi umgeschalten werden. Dadurch können unterschiedliche Interaktionen wie Skalieren und Rotieren mit der gleichen Geste durchgeführt werden.

Controller

Die vierte Möglichkeit der Interaktion mit HMD-Devices sind Controller. Die meisten VR-Headsets sind nur mit solchen Controllern zu bedienen. In der aktuellen Version unterstütz die HoloLens jedoch kein Controller. Bei anderen Windows Mixed Reality Geräten wie die *Immersive* Headsets von Acer und HP[2] kann mit den *Motion* Controllern, ähnlich wie bei Gaming Konsolen, mit verschiedenen Tasten unterschiedliche Eingaben tätigen. Bei der HoloLens steht lediglich der *Clicker* zur Verfügung, ein kleines Bluetooth-Gerät, dass durch physisches Klicken den Air Tap ersetzen kann. Bei Dauerhafter Eingabe von Air Tap Gesten ist der Einsatz vom Clicker angenehm, da man die Hand nicht im Sichtfeld der HoloLens halten muss.

5.1.2 Herausforderungen

Bei der Entwicklung von AR-Applikationen sind einige Herausforderungen zu beachten, die auch für CodeLeaves von Relevanz sind. Im Zuge der Entwicklung von der HoloLens-Anwendung HoloStudio hat der Senior Holographic Designer Ghaly in [**windows2017casestudy3**] eine Case Study veröffentlicht. Mitunter traten die im Folgenden betrachteten Herausforderungen auf.

Hologramme außerhalb des Sichtfelds

Da Hologramme im ganzen Raum, in dem sich der Betrachter befindet, platziert werden können, ist nicht immer gewährleistet, dass sie sich auch im Blickfeld des Betrachters befinden. Besonders bei der aktuellen Entwickler-Version der HoloLens ist das Sichtfeld, in dem Hologramme angezeigt werden können, mit rund 30 Grad [**czerulla2017microsoft**] gering.

Eine Variante dies zu umgehen, ist es *Tag-along*-Objekte zu verwenden. Diese Objekte bleiben immer im Blickfeld des Betrachters. Schaut der Nutzer in eine andere Richtung, wird das Objekt neu positioniert, sodass es wieder im Blickfeld gelangt. Innerhalb des Blickfelds bleiben Tag-along-Objekte aber stationär, sodass der Nutzer mit dem Gaze

[2] `https://developer.microsoft.com/en-us/windows/mixed-reality/immersive_headset_hardware_details`

weiterhin mit ihnen interagieren kann. Das Windows-Startmenü der HoloLens ist ein solches Tag-along-Objekt.

In der Case Study aus [**windows2017casestudy3**] wurde festgestellt, dass die Probanden sich von solchen Tag-along-Objekten bedrängt fühlen und instinktiv versuchen davon weg zu kommen.

Eine andere Möglichkeit ist es die Hologramme stationär im Raum zu platzieren und die Aufmerksamkeit des Nutzers gezielt zu lenken. Das kann mit verschiedenen Techniken erreicht werden. Eine davon ist die Verwendung von *Spacial Sound*, was bedeutet, dass Geräusche ebenfalls im Raum platziert werden können und vom Nutzer dreidimensional mit Richtung und Entfernung warngenommen werden können. Dies bietet sich vor allem bei Spielen mit bewegten Charakteren an. Bei statischem Content wie CodeLeaves, sind Geräusche nicht besonders intuitiv.

Es kann auch mit zusätzlichen Objekten gearbeitet werden, die dem Nutzer darauf hinweisen, wo sich etwas befindet, was seine Aufmerksamkeit erfordert. Solche Hilfs-Objekte nennen wir *Direction Indicators*. Diese können unterschiedliche Gestalt annehmen. Bei anderen HoloLens-Applikationen sind Pfeile, Licht oder Denkblasen im Einsatz.

UI von anderen Objekten verdeckt

Ein Problem, das in vielen HoloLens-Applikationen auftritt, ist dass UI Elemente von anderen Hologrammen, die näher am Nutzer positioniert sind, verdeckt werden.

Das Problem ist in Abbildung **??** illustriert. Darin wird in CodeLeaves das Menu zu Interaktion mit dem Wald der „Move“-Button von einem Stamm verdeckt.

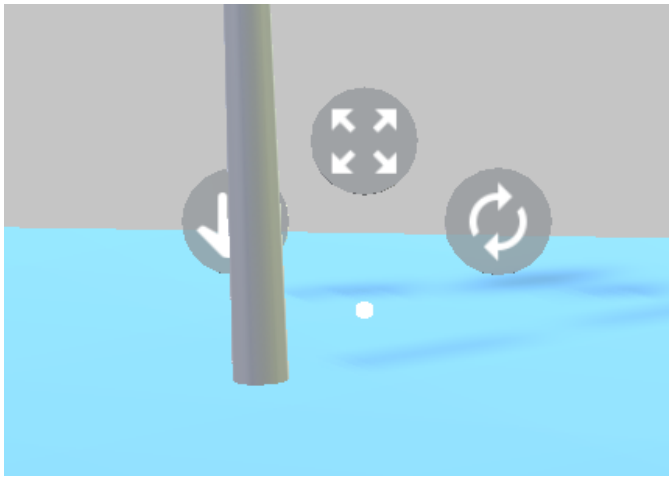


Abbildung 5.2 Verdeckung eines Buttons

Analog zum 2D Popups ist es natürlich möglich UI Elemente vor allen anderen Elementen zu platzieren. In 3D bedeutet das, dass das UI Element vor dem Hologramm erscheinen muss, dass sich am nächsten am Nutzer befindet. Die UI Elemente würde dadurch vom eigentlichen interagierten Objekt losgelöst werden, was eine wenig intuitive Interaktion darstellt. Dies kann aber zu Folge haben, dass der Nutzer irritiert

wird, weil er mit etwas weiter weg interagiert und die UI deutlich weiter vorne erscheinen kann.

Eine Abhilfe kann ein *Shine-through*-Effekt schaffen, wie er bei HoloStudio Verwendung findet. Dabei bleiben UI Elemente an dem Hologramm angeheftet, mit dem interagiert wird, jedoch scheinen sie durch davor liegende Hologramme durch.

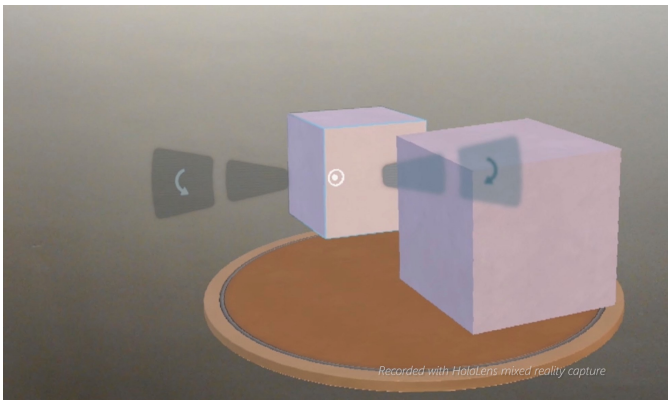


Abbildung 5.3 Shine-through-Effect von UI Elementen in HoloStudio [**windows2017casestudy3**]

Diese Methode funktioniert jedoch schlecht, wenn die UI Elemente Texte sind und von davor liegenden Texten verdeckt werden. Dadurch werden beide Texte unleserlich. Dieses Problem tritt z.B. zwangsläufig bei der Beschriftung von Blättern in CodeLeaves auf und lässt sich nur vermeiden, indem einer der beiden Texte verschwindet oder zumindest verblasst.

Die betrachteten Herausforderungen von 3D Interaktion und deren mögliche Lösungsansätze finden sich im Interaktionskonzept von CodeLeaves wieder und werden in Abschnitt **??** aufgegriffen.

5.2 Reaktive UI

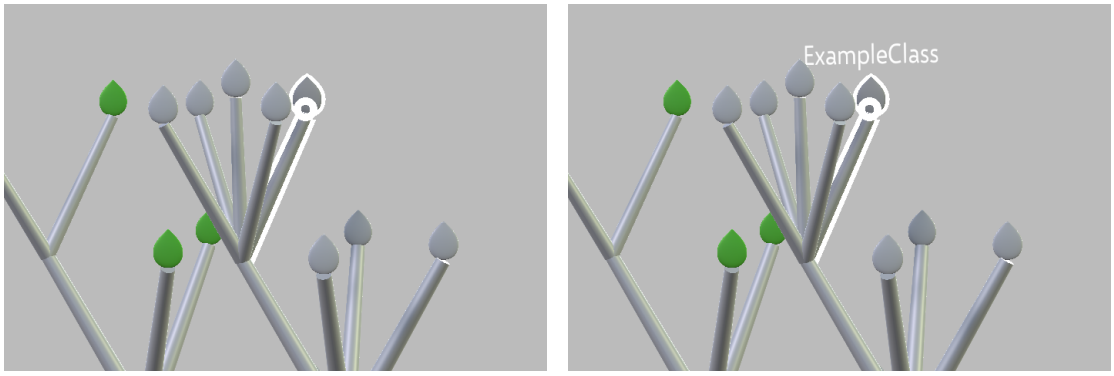
5.2.1 Motivation

Eine der offensichtlichen Interaktionen mit CodeLeaves ist das Anklicken von Knoten, seien es Blätter oder die dazugehörigen Kanten. Der Nutzer erwartet damit nähere Informationen zu einem Paket oder einer Klasse zu erfahren. Betrachten wir zunächst eine sehr einfaches Szenario:

> *Wenn der Nutzer auf ein Blatt klickt, soll daraufhin der Name der repräsentierten Klasse über dem Blatt erscheinen.*

Diese Interaktion ist in Abbildung **??** veranschaulicht.

Bei der Generierung des Baumes wird zu jedem Blatt ein Label hinzugefügt, in dem der Klassenname gespeichert werden kann. Das Anzeigen Namens ist demnach kein Problem und würde wie folgt funktionieren:



(a) Fokussiertes Blatt **(b)** Angeklicktes Blatt

Abbildung 5.4 Anzeige des Klassennamens durch Klick auf ein Blatt

Listing 5.1: Direkte Manipulation (Negativbeispiel)

```
public class NodeInputHandler : MonoBehaviour, IInputClickHandler
{
    public void OnInputClicked(InputClickedEventData eventData)
    {
        // Get label of currently clicked node
        var labelObject = GetLabelObject(gameObject);

        // Toggle active
        labelObject.SetActive(!labelObject.activeSelf);
    }
}
```

Im traditionellen Ansatz könnte also auf das Ereignis des AirTaps auf einfachste Weise reagiert werden. Wird das Blatt angeklickt, wird das Label-Objekt des Blatts aktiviert und der Name wird damit sichtbar. Entsprechend kann bei erneutem Klick das Label-Objekt wieder deaktiviert werden. Wird die Businesslogik aber komplexer, kommt dieser Ansatz schnell an seine Grenzen.

Eventuell könnte sich zum Beispiel folgende Einstellung in der Applikation als sinnvoll erweisen:

> *Der Nutzer kann einstellen, ob bei einem Klick auf ein Blatt der Klassenname oder die Zahl der gerade visualisierten Metrik angezeigt wird.*

Woher weiß die Funktion jetzt was angezeigt werden soll?

Das eigentliche Problem bei der beschriebenen Situation ist, dass Businesslogik und UI eng gekoppelt wurde. Dies sollte tunlichst vermieden werden. Bei einem Klick auf einem Blatt sollte nicht die UI entscheiden was zu tun ist, sondern die Businesslogik. Die Entkopplung von UI und Logik, ist mit reaktiver Programmierung möglich.

Was ist reaktive Programmierung? Staltz beschreibt es in [**staltz2016introduction**] als „Programmierung mit asynchronen Datenströmen." Dies trifft den Kern. Es geht darum auf Änderungen von Werten zu reagieren, die durch *Streams* propagiert werden.

Streams sind nichts anderes als *Observables* die bei jeder Änderung ihrer Werte allen registrierten *Subscribers* den neuen Wert mitteilen.

Betrachten wir in Abbildung **??** konkret das Beispiel den Text, der im Label eines Blatts steht.

Abbildung 5.5 Datenstrom des Labels

Über die Zeit hinweg, kann sich durch die Businesslogik der darzustellende Text verändern. Im Beispiel soll der Klassenname, dann vielleicht Laufzeitfehler, Coverage und dann wieder der Klassenname dargestellt werden.

Bei der Generierung des Baumes muss dann lediglich der Label-Wert observiert und der Text entsprechend gesetzt werden.

5.2.2 Verwendung in Unity

In Unity steht für reaktive Programmierung das Open-Source-Projekt *UniRx*[3] zur Verfügung, das die Funktionalität der reaktiven .NET Library *Rx* für Unity zugänglich macht. UniRx implementiert z.B. *ReactiveProperties*, mit denen eine einfache Verwendung des Observer-Pattern möglich ist. Darüber hinaus bietet die API hilfreiche Stream-Funktionen wie filter, map, merge oder join.

In unserem Beispiel von oben wird der Text eines Blatts des UI-Models zu einer ReactiveProperty und bei jeder Änderung wird der Text des Labels angepasst. Auch das aktivieren des Labels wird über eine ReactiveProperty realisiert. So ist zum Beispiel das deaktivieren von allen Labels gleichzeitig leicht möglich.

In Listing **??** wird gezeigt, wie eine ReactiveProperty für den Status `isSelected` eines UI-Knotens definiert wird. Im `TreeBuilder` (Zeile 9) wird das Label bei Aktualisierung von `isSelected` entsprechend aktiviert.

Listing 5.2: Observieren von Werten

```
1  public abstract class UiNode
2  {
3      public ReactiveProperty<bool> isSelected { get; set; }
4      ...
5  }
6
7  public class TreeBuilder {
```

[3] https://github.com/neuecc/UniRx

```
8       ...
9       node.IsSelected.Subscribe(label.SetActive);
10      ...
11  }
12
13  public class NodeInputHandler : MonoBehaviour, IInputClickHandler
14  {
15      public void OnInputClicked(InputClickedEventData eventData)
16      {
17          Store.Dispatch(NodeClickAction(GetNodeId(gameObject)));
18      }
19      ...
20  }
```

Im *NodeInputHandler* wird nun nicht mehr direkt auf die UI zugegriffen, sondern es wird ein Funktion aufgerufen, die sich darum kümmert, dass das UI-Model editiert. Hier kommt zusätzlich Redux[4] ins Spiel.

5.2.3 Redux

Mit UniRx kann auf Änderung eines Models in der UI reagiert werden. Redux ist ein Konzept, das noch ein Schritt weiter geht. Die Grundidee ist, dass der komplette Status einer App in einem *State* gespeichert wird. Der State kann durch *Reducer*, reinen Funktionen transformiert werden. Diese wiederum haben als Input wohl definierte *Actions*, welche durch Interaktion mit der UI *dispatched* werden. Der State, die Reducer und die Möglichkeit Actions zu versenden sind Inhalt des *Stores* [**abramov2017redux**].

Diesem Prinzip folgenden ist auch die Interaktion von CodeLeaves aufgebaut. Dadurch wird einen nachvollziehbaren Datenfluss garantiert und eine technische Basis für ein Interaktionskonzept geschaffen.

5.3 Interaktion mit CodeLeaves

CodeLeaves kann ohne Interaktion die Struktur einer Software, eine Metrik und Verbindungen zwischen Softwareartefakten darstellen. Damit kann bereits einen Überblick über die Software geschaffen werden. Der wirklichen Mehrwert von CodeLeaves wird aber erst erreicht, wenn mit dem Software-Wald interagiert wird. Die intuitive Bedienung von CodeLeaves ist demnach essentiell für eine gute User Experience und damit ein tieferes Verständnis der visualisierten Informationen.

5.3.1 Interaktion mit dem gesamten Wald

Grundlegend muss der gesamte Wald einige *Basisinteraktionen* der AR unterstützen, bevor der Nutzer überhaupt mit CodeLeaves sinnvoll arbeiten kann. Diese sind:

[4] http://redux.js.org/

- Platzieren
- Skalieren
- Rotieren

Das Platzieren ist von Anfang an notwendig um den Wald im Raum dahin zu setzten, wo es für den Nutzer am meisten Sinn macht. Bequemerweise kann das z.B. auf einem Tisch sein, oder wenn ein solcher nicht zur Verfügung steht auch auf dem Fußboden. Wenig hilfreich wäre es aber wenn er über dem Benutzer schwebt.

Das Skalieren ist auch unterlässlich. Gerade bei großen Systemen kann auch der Wald sehr groß werden. Daher muss die Größe des Waldes auf die Gegebenheiten der Räumlichkeiten angepasst werden können.

Ist der Wald einmal platziert und skaliert, möchte der Nutzer sich den Wald von allen Seiten anschauen können. Er kann natürlich darum herumlaufen, aber vielleicht reicht der Platz dazu nicht aus, oder ein Drehen des Waldes selbst ist bequemer. Daher müssen die Basisinteraktionen im 3D Raum ermöglicht werden.

Bounding Box

Das Repository „Mixed Reality Design Labs“ bietet für genau diese Interaktionen eine Lösung. Die „Bounding Box“, wie sie in Abbildung **??** zu sehen ist, rendert ein 3D Rahmen um das zu manipulierende Objekt. Das Platzieren des Objekts kann mit einer Manipulation auf den gesamten Bereich der Bounding Box erreicht werden. Mit den kleinen Würfeln an den Ecken der Bounding Box kann skaliert werden und die kleine Kugeln auf den vertikalen Kanten der Bounding Box ist für das Rotieren vorgesehen.

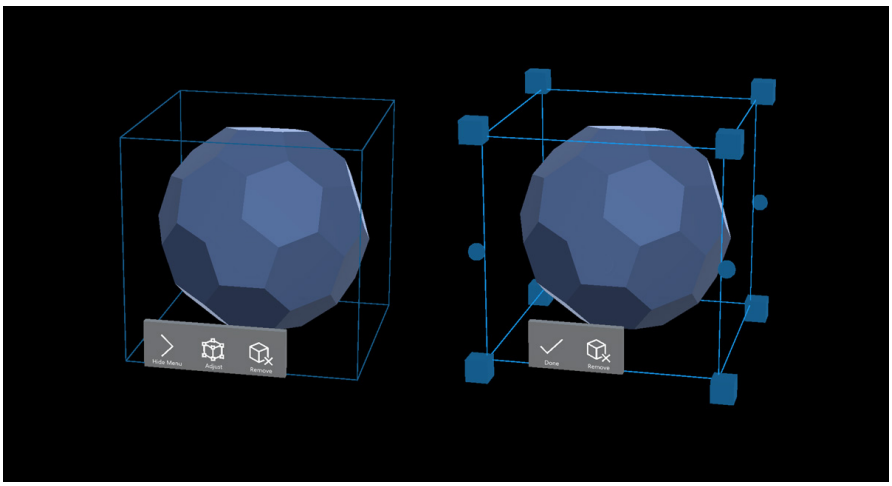


Abbildung 5.6 Bounding Box zum Platzieren, Skalieren und Rotieren von Objekten

Dieser Ansatz hat den Vorteil, dass dem Nutzer ein Interface angeboten wird, das ihm aus dem Zweidimensionalen wohl bekannt ist. In den meisten Applikationen mitunter in den Bereichen Text- und Bild- oder Grafikverarbeitung ist die Bounding Box Standard. In die Dreidimensionalität übertragen kennt der Nutzer das Prinzip und weiß was er für eine gewünschtes Ziel zu tun hat.

Interaktionsmodi

Für CodeLeaves ist dieser Ansatz jedoch aus einem einfachen Grund schwierig. Der Wald muss nicht zwangsläufig in seiner Gänze im Blickfeld des Nutzers sein. Angenommen der Nutzer interessiert sich vor allem für ein bestimmten Baum, möchte er diesen genauer betrachten und vergrößert den Wald soweit, dass andere Bäume aus dem Sichtfeld verschwinden. Möchte er dann den Wald wieder verkleinern, müsste er erst soweit zurück gehen, bis er die Bounding Box nutzen kann. Das ist aber aufgrund der Räumlichkeiten vielleicht gar nicht möglich und wenn doch nicht besonders benutzerfreundlich.

Eine Alternative muss daher gefunden werden. Es stellt sich die Frage, was immer im Sichtfeld des Nutzers ist und den Wald für den Nutzer als Ganzes repräsentiert. Der Waldboden als Antwort auf diese Frage leuchtet ein. Er ist auch bei näherer Betrachtung einzelner Teile von CodeLeaves immer präsent und da alles aus ihm wächst ist eine Interaktion mit ihm für die Basisinteraktionen intuitiv.

Damit steht das Ziel der Interaktion fest, jedoch können mit der begrenzten Anzahl an Gesten nicht alle Basisinteraktionen mit dem gleichen Ziel bedient werden. Daher wird bei einem AirTap ein Kontextmenü dargestellt, mit dem der Nutzer zwischen den drei verschiedenen Basisinteraktionen wählen kann. Eines davon ausgewählt, kann der Nutzer mit fokussiertem Waldboden die Manipulation Geste nutzen, um den Wald wie gewünscht zu transformieren.

5.3.2 Das Kontextmenü

Das Kontextmenü, mit dem die verschiedenen Interaktionsmodi für dem gesamten Wald ausgewählt werden können, lässt sich aber nicht nur für die Interaktion mit dem Waldboden anwenden. Auch für andere fokussierte Objekte ist eine vom Kontext abhängige Auswahl von möglichen Funktionen allgemeingültig sinnvoll. In herkömmlichen 2D User Interfaces ist dies mit einem Rechtsklick zu vergleichen. In 3D kann um den Punkt, den der Nutzer beim Aufruf des Kontextmenüs fokussiert hatte, das Kontextmenü als Buttons in einer kreissegmentförmigen Anordnung dargestellt werden.

Die kreissegmentförmige Anordnung der Buttons hat den Vorteil, dass der Interaktionspunkt für das öffnen des Kontextmenüs im Zentrum des Kreises ist und daher eine Verbindung mit dem fokussieren Objekt schafft. Darüber hinaus sind alle Buttons vom Cursor gleich weit entfernt und daher schnellstmöglich mithilfe des Gaze erreichbar.

Die Distanz des Kontextmenüs ist den Umständen entsprechend anzupassen. Ist der fokussierte Punkt bereits näher als die Distanz, in der eine Darstellung von Interaktionselementen empfohlen wird[5], sollte das Kontextmenü nicht noch näher an dem Nutzer platziert werden. Bei größerem Abstand sollte der Abstand zum Kontextmenü auf die empfohlene Distanz reduziert werden, sodass der Nutzer noch komfortabel mit dem Menü interagieren kann.

In dem Fall, dass das Kontextmenü direkt an der fokussierten Position erscheint, muss das Kreissegment des Kontextmenüs so angepasst werden, dass die Buttons nicht

[5] Die Distanz, in der es empfohlen ist Hologramme zu platzieren, liegt bei 2,0 Metern [**windows2017interaction**]

in das fokussierte Objekt hinein ragen. Für den Waldboden ist daher ein nach oben orientiertes Kreissegment eine gute Wahl. Würde aber zum Beispiel mit einer Wand interagiert werden, sollte das Kontextmenü von der Wand weg orientiert sein.

Ein Beispiel für ein solches Kontextmenü mit zuvor fokussierten Waldboden, ist in Abbildung **??** zu sehen.

(a) Kontextmenü aus Sicht des Nutzers **(b)** Kontextmenü von der Seite

Abbildung 5.7 Kontextmenü im Fall der Interaktion mit dem Waldboden

5.3.3 Das App-Menü

5.3.4 Manipulations-Indikatoren

6 Zusammenfassung und Ausblick

Literatur

[1] Ronald Azuma u. a. „Recent advances in augmented reality“. In: *IEEE computer graphics and applications* 21.6 (2001), S. 34–47.

[2] Ronald T Azuma. „A survey of augmented reality“. In: *Presence: Teleoperators and virtual environments* 6.4 (1997), S. 355–385.

[3] Michael Balzer u. a. „Software landscapes: Visualizing the structure of large software systems“. In: *IEEE TCVG*. 2004.

[4] Mike Cohn. *User stories applied: For agile software development*. Addison-Wesley Professional, 2004.

[5] S. Diehl. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer Berlin Heidelberg, 2007. ISBN: 9783540465041.

[6] H. Ernst, J. Schmidt und G. Beneken. *Grundkurs Informatik: Grundlagen und Konzepte für die erfolgreiche IT-Praxis - Eine umfassende, praxisorientierte Einführung*. Springer Fachmedien Wiesbaden, 2016. ISBN: 9783658146344.

[7] Buschmann Frank, Henney Kevlin und C Schmidt Douglas. *Pattern Oriented Software Architecture Volume 5: On Patterns and Pattern Languages*. 2007.

[8] Hamish Graham, Hong Yul Yang und Rebecca Berrigan. „A solar system metaphor for 3D visualisation of object oriented software metrics“. In: *Proceedings of the 2004 Australasian symposium on Information Visualisation-Volume 35*. Australian Computer Society, Inc. 2004, S. 53–59.

[9] Hirokazu Kato und Mark Billinghurst. „Marker tracking and hmd calibration for a video-based augmented reality conferencing system“. In: *Augmented Reality, 1999.(IWAR'99) Proceedings. 2nd IEEE and ACM International Workshop on*. IEEE. 1999, S. 85–94.

[10] Kevin Kelly. „The untold story of magic leap, the world's most secretive startup“. In: *Recuperado de https://www.wired.com/2016/04/magic-leap-vr* (2016).

[11] Paul Milgram u. a. „Augmented reality: A class of displays on the reality-virtuality continuum“. In: *Photonics for industrial applications*. International Society for Optics und Photonics. 1995, S. 282–292.

[12] Lothar Papula. „Mathematik für Ingenieure und Naturwissenschaftler Band 3–Vektoranalysis, Wahrscheinlichkeitsrechnung, Mathematische Statistik, Fehler- und Ausgleichsrechnung, 6“. In: *Aufl. Braunschweig: Vieweg* (2001).

[13] David Phelan. *Apple CEO Tim Cook: As Brexit hangs over UK, 'times are not really awful, there's some great things happening'*. 2017. URL: http://www.independent.co.uk/life-style/gadgets-and-tech/features/apple-tim-cook-boss-brexit-uk-theresa-may-number-10-interview-ustwo-a7574086.html#gallery (besucht am 30. 05. 2017).

[14] Marcel Pütz. „Softwarevisualisierung für Virtual Reality". Masterseminar. Hochschule Rosenheim, 2017.

[15] QAware GmbH. *IT-Probleme lösen. Digitale Zukunft gestalten.* 2017. URL: http://www.qaware.de/leistung/#leistung-realisierung (besucht am 28. 03. 2017).

[16] QAware GmbH. *Johannes Weigend - Chefarchitekt, Geschäftsführer und Mitgründer*. 2017. URL: http://www.qaware.de/unternehmen/johannes-weigend/ (besucht am 01. 07. 2017).

[17] George G Robertson, Jock D Mackinlay und Stuart K Card. „Cone trees: animated 3D visualizations of hierarchical information". In: *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM. 1991, S. 189–194.

[18] Frank Steinbrück. „Consistent Software Cities: supporting comprehension of evolving software systems". Diss. Cottbus, Brandenburgische Technische Universität Cottbus, 2013.

[19] R. Wettel und M. Lanza. „Program Comprehension through Software Habitability". In: *15th IEEE International Conference on Program Comprehension (ICPC '07)*. 2007, S. 231–240. DOI: 10.1109/ICPC.2007.30.

[20] R. Wettel und M. Lanza. „Visual Exploration of Large-Scale System Evolution". In: *2008 15th Working Conference on Reverse Engineering*. 2008, S. 219–228. DOI: 10.1109/WCRE.2008.55.

[21] Richard Wettel, Michele Lanza und Romain Robbes. „Software systems as cities: A controlled experiment". In: *Proceedings of the 33rd International Conference on Software Engineering*. ACM. 2011, S. 551–560.