Lehrstuhl für Computergestützte Modellierung und Simulation
TUM School of Engineering and Design
Technische Universität München

Analyse der Kollaborationsplattform Speckle im Hinblick auf inkrementelle Updates von BIM-Modellen

Wissenschaftliche Arbeit zur Erlangung des Grades

Bachelor of Science (B.Sc.)

an der TUM School of Engineering and Design
der Technischen Universität München.

Betreut von Prof. Dr.-Ing. André Borrmann
Sebastian Esser M.Sc.
Lehrstuhl für Computergestützte Modellierung und Simulation

Eingereicht von Kai Dietmair

Eingereicht am 07. April 2024

Vorwort

Ich bedanke mich herzlich bei meinem Betreuer Sebastian Esser, der mit seinem umfangreichen Wissen auf dem Gebiet des Building Information Modeling entscheidend zum Gelingen dieser Arbeit beigetragen hat. Ich bedanke mich insbesondere für die investierte Zeit, die aufgebrachte Geduld, die vielen hilfreichen Tipps und die motivierenden Worte. Zudem bin ich dankbar für seinen Vorschlag für dieses höchst interessante und aktuelle Thema.

Abstract

This thesis examines the Speckle collaboration platform and its approach to implementing incremental model updates within the field of BIM. For this purpose, experiments are carried out with the respective Speckle plug-ins for a BIM authoring tool and a CAD authoring tool. This involves observing the behavior of the plug-ins as they receive various increments. During the experiments, a model is created in both the BIM and CAD software to assess the plug-ins' capabilities in modifying dimensions of model objects, as well as moving, adding, and deleting model objects. The results provide information about the advantages and current limitations of Speckle. In order to gain deeper insights into data exchange using Speckle, the data structure of the platform is analyzed and selected Speckle objects are examined and compared with each other in their serialized JSON format.

Zusammenfassung

In dieser Arbeit werden die Kollaborationsplattform Speckle und der von ihr angewendete Ansatz bei der Umsetzung inkrementeller Updates von BIM-Modellen untersucht. Dafür werden Versuche mit den jeweiligen Speckle-Plug-ins für eine BIM-Autorensoftware und ein CAD-Autorensoftware durchgeführt und deren Verhalten beim Empfangen unterschiedlicher Inkremente bzw. Daten beobachtet. Im Rahmen der Versuche wird jeweils ein Modell in der BIM- und CAD-Software erstellt und die Fähigkeit der Plug-ins getestet, die Abmessungen von Modellobjekten zu verändern sowie Modellobjekte zu verschieben, einzufügen und zu entfernen. Die Versuchsergebnisse geben Aufschluss über die Vorzüge und aktuellen Grenzen von Speckle. Um tiefere Einblicke in den Datenaustausch mittels Speckle zu erhalten, werden zusätzlich der Aufbau und die Datenstruktur der Plattform analysiert sowie ausgewählte Speckle-Objekte in ihrem serialisierten JSON-Format untersucht und miteinander verglichen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

Abkürzungen

AEC	Architecture, Engineering and Construction
API	Application Programming Interface
BIM	Building Information Modeling
BRep	Boundary Representation
CAD	Computer Aided Design
CDE	Common Data Environment
CSG	Constructive Solid Geometry
ID	Identifikation
IFC	Industry Foundation Classes
JSON	JavaScript Object Notation
SDK	Software Development Kit
UML	Unified Modeling Language
VCS	Version Control System

Kapitel 1

Einleitung

1.1 Motivation

Die Planung bebauter Umwelt zeichnet sich durch eine Vielzahl von Planern[1] unterschiedlicher Disziplinen aus. Zusammenarbeit und Kommunikation sind hier unumgänglich. In den letzten zwei Jahrzehnten wurde dabei der traditionelle Informationsaustausch in Form ausgedruckter Pläne und Dokumente zunehmend durch modellbasierte Ansätze aus dem Bereich des Building Information Modeling (BIM) ersetzt (SCHAPKE et al., 2021). Ein BIM-Modell enthält geometrische sowie semantische Informationen über ein Bauwerk und kann in verschieden Dateiformaten encodiert werden (OH et al., 2015).

Aufgrund des iterativen Charakters von Planungsprozessen kommt es häufig zu Änderungen in Modellen, die möglicherweise bereits an andere Projektbeteiligte ausgetauscht worden sind. Die gängige Praxis besteht heute darin, das veränderte Modell als neue monolithische Datei mit anderen Akteuren zu teilen. Diese müssen das neue Modell manuell auf Änderungen und Unstimmigkeiten mit ihrem eigenen domänenspezifischen Modell überprüfen, was arbeitsintensiv und fehleranfällig ist. Verbessert werden kann die Situation durch den Einsatz inkrementeller Update-Verfahren, in denen im Falle einer Änderung lediglich die Informationen über die modifizierten Objekte eines Modells übermittelt werden. (ESSER et al., 2022)

1.2 Ziel der Arbeit

Im Mittelpunkt dieser Abhandlung stehen die Kollaborationsplattform Speckle (»Speckle«, 2024) und der von ihr verfolgte Ansatz bei der Umsetzung von inkrementellen Updates für BIM-Modelle. Dabei sollen die folgenden drei Forschungsfragen beantwortet werden:

1. Warum werden inkrementelle Update-Verfahren in der Bauindustrie benötigt und welche Ansätze wurden dafür bereits entwickelt?
2. Wie sind das System und die Datenstruktur von Speckle aufgebaut und welchen Ansatz verfolgt Speckle bei der Umsetzung von inkrementellen Updates?
3. Was sind die Vorzüge und Grenzen des inkrementellen Update-Verfahrens von Speckle?

[1] Der Lesbarkeit halber werden im Rahmen dieser Arbeit nur männliche Personenbezeichnungen verwendet. Diese gelten gleichermaßen für männliche, weibliche und diverse Personen.

Das Update-Verfahren von Speckle soll hinsichtlich der Vollständigkeit der übertragenen Inkremente untersucht werden. Bei den Inkrementen handelt es sich um vier verschiedene Arten von Modelländerungen: das Ändern der Abmessungen eines Modellobjekts sowie das Einfügen, Verschieben und Entfernen von Modellobjekten. Dafür soll zunächst der Aufbau und die Datenstruktur der Speckle-Plattform untersucht werden, um anschließend mithilfe der daraus gewonnenen Erkenntnisse Versuche mit inkrementellen Modelländerungen durchführen zu können. Die Versuche sollen sowohl innerhalb der BIM-Software Revit als auch innerhalb AutoCAD, einer Software aus dem Bereich des Computer Aided Design (CAD), durchgeführt werden. Doch auch der Datenaustausch zwischen den beiden Softwareprodukten mittels Speckle soll ein Versuchsgegenstand sein.

Um noch tiefere Einblicke in die Datenstruktur von Speckle und die praktische Umsetzung inkrementeller Updates zu gewinnen, sollen die im Zuge der Versuche generierten „Speckle-Objekte“ analysiert und miteinander verglichen werden.

1.3 Aufbau der Arbeit

Nach dieser Einleitung wird im zweiten Kapitel dieser Abhandlung ein Überblick gegeben über den derzeitigen Stand der Wissenschaft und Technik in Bezug auf die Thematik. Daraus soll hervorgehen, warum inkrementelle Update-Verfahren im Bauwesen benötigt werden und welche Ansätze dafür bereits entwickelt wurden. Das darauffolgende Kapitel widmet sich dem System von Speckle und wie damit inkrementeller Datenaustausch realisiert wird. In Kapitel vier werden Versuche durchgeführt und ausgewählte Speckle-Objekte miteinander verglichen. Die daraus gewonnenen Ergebnisse werden anschließend in Kapitel fünf zusammengefasst und diskutiert, wodurch die Vorzüge und aktuellen Grenzen von Speckle ersichtlich werden. Schließlich wird im letzten Kapitel dieser Arbeit ein Fazit gezogen und ein Ausblick gegeben auf mögliche zukünftige Forschung.

Kapitel 2

Stand der Wissenschaft und Technik

2.1 Grundlagen aus Informatik und Softwareentwicklung

2.1.1 Objektorientierte Programmierung

In der objektorientierten Programmierung gibt es Klassen und Objekte. Eine Klasse stellt dabei einen nutzerdefinierten Datentyp dar und legt das Konstruktionsschema gleichartiger Objekte fest. Sie definiert, welche Attribute ein Objekt dieser Klasse aufweist und welche Methoden auf diesem Objekt ausführbar sind. Ein Attribut besitzt einen Namen, einen Datentyp und einen Wert, wobei der Datentyp wiederum eine Klasse und der Wert ein Objekt dieser Klasse sein kann. Methoden eines Objekts können auf dessen Attribute zugreifen und definieren, welche Aktionen das Objekt ausführen kann. Objekte werden oft auch als Instanzen einer Klasse bezeichnet. Ein Konzept der objektorientierten Programmierung, das im weiteren Verlauf dieser Arbeit relevant wird, ist die Vererbung. Dabei kann eine Klasse auf Basis einer gegebenen Klasse erzeugt werden und erbt somit die von der gegebenen Klasse definierten Attribute und Methoden. In der neu erzeugten Klasse können zusätzlich zu den übernommenen Attributen durchaus neue Attribute definiert werden. Um Klassen und deren Beziehungen zueinander in einem objektorientierten System auf abstrakter und strukturierter Ebene visuell darzustellen, eignet sich die Unified Modeling Language (UML). Ein wesentlicher Vorteil der objektorientierten Programmierung ist die Erweiterbarkeit bestehender Systeme. (EIRUND, 1993)

In der Programmiersprache Python definiert eine Klasse den Namen eines Attributs und optional auch dessen Datentyp, während der Wert erst beim Erstellen einer Instanz dieser Klasse definiert wird. Beispielsweise sollte eine Klasse namens *Linie* sinnvollerweise das Attribut *Länge* mit dem Datentyp „float“, dem elementaren Datentyp für Fließkommazahlen, definieren. Sobald ein Linien-Objekt erstellt wird, definiert der Anwender dann einen Wert für dessen Länge, beispielsweise 4.75. Die Maßeinheit dieser Zahl ist dabei im Vorhinein nicht klar. Es könnte sich um Millimeter, Kilometer, Meilen o. Ä. handeln. Um diese Unklarheit zu beheben, bietet es sich an, der Klasse *Linie* zusätzlich das Attribut *Maßeinheit* mit dem Datentyp „str“ hinzuzufügen. „str“ ist in Python der Datentyp für Zeichenketten wie „Meter“ oder „123abc“. Ein weiterer nützlicher, in Python vordefinierter Datentyp, ist „int“, welcher für ganze Zahlen verwendet wird.

2.1.2 Transaktionen und Nebenläufigkeitskontrollen

SCHNEIDER (2004) bezeichnet eine Transaktion als eine Folge von zusammengehörenden Operationen, die auf eine Datenbank zugreifen und deren Inhalt verändern. Greifen mehrere Nutzer zur selben Zeit unabhängig voneinander auf den selben Datenbestand zu, spricht man von Nebenläufigkeit. Eine Transaktion kann durch die sogenannten ACID-Eigenschaften charakterisiert werden, welche erstmals von HAERDER und REUTER (1983) formuliert wurden:

- Atomicity (dt. Atomarität): Eine Transaktion ist eine unteilbare Einheit, d. h. sie wird entweder vollständig oder gar nicht ausgeführt („Alles-oder-nichts"-Prinzip). Sobald eine einzelne Operation der Transaktion fehlschlägt, werden alle zuvor durchgeführten Operationen dieser Transaktion rückgängig gemacht. Man spricht dann von einem Rollback.
- Consistency (dt. Konsistenz): Eine fehlerfrei ausgeführte Transaktion überführt den Datenbestand von einem konsistenten Zustand in einen anderen konsistenten Zustand. Während der Ausführung der Transaktion kann der Datenbestand jedoch durchaus einen inkonsistenten Zustand annehmen.
- Isolation: Nebenläufig ausgeführte Transaktionen müssen isoliert voneinander ausgeführt werden, d. h. das Ergebnis muss das gleiche sein wie das von den Transaktionen in irgendeiner seriellen Reihenfolge erzeugte Ergebnis.
- Durability (dt. Dauerhaftigkeit): Die Auswirkung einer erfolgreich beendeten Transaktion auf den Datenbestand ist persistent.

Die Eigenschaften Atomarität, Konsistenz und Dauerhaftigkeit lassen sich anhand dem Beispiel einer Banküberweisung erläutern, bei der zunächst ein bestimmter Betrag von Bankkonto A abgebucht und daraufhin auf Bankkonto B gutgeschrieben werden soll. Kommt es nach der Operation der Abbuchung zu einem Fehler und die Operation Gutschrift wird nicht durchgeführt, führt das zu einer Inkonsistenz, denn die Gesamtsumme der beiden Kontobestände hat sich geändert, d. h. Geld geht verloren. Daher muss die Transaktion entweder ganz oder gar nicht ausgeführt werden und nach einem Fehler ein Rollback durchgeführt werden (Atomarität). Wurde die Banküberweisung jedoch erfolgreich durchgeführt, darf ihr Effekt auf die jeweiligen Bankkonten nicht verloren gehen (Dauerhaftigkeit). (SCHNEIDER, 2004)

Die verteilt-synchrone Bearbeitung gemeinsamer Materialien stellt eine Herausforderung für die Datenverwaltung dar (SCHAPKE et al., 2021). Denn der nebenläufige Zugriff mehrerer Nutzer auf einen gemeinsamen Datenbestand birgt die Gefahr, dass sich Transaktionen gegenseitig beeinflussen und es zu Widersprüchen oder Inkonsistenzen in den Daten kommt. Es ist die Aufgabe der Nebenläufigkeitskontrolle, diesen nebenläufigen Zugriff zu steuern und unerwünschte, inkonsistente Zustände des Datenbestandes zu verhindern (SCHNEIDER, 2004). Bei dieser Kontrolle unterscheidet man grundsätzlich zwischen zwei verschiedenen Varianten: Die pessimistische Nebenläufigkeitskontrolle vermeidet

im Vorhinein Konflikte und erlaubt nur bestimmte Änderungen, wohingegen die optimistische Nebenläufigkeitskontrolle Konflikte im Datenbestand temporär toleriert und erst im Nachhinein identifiziert und auflöst (SCHAPKE et al., 2021).

Erstere wird als pessimistisch bezeichnet, weil man hier das davon ausgeht, dass viele Konflikte auftreten. Das wichtigste Mittel der pessimistischen Nebenläufigkeitskontrolle sind Sperren. Damit werden Teile des Datenbestandes, die Gegenstand einer Transaktion sind, markiert, um sie damit anderen Transaktionen temporär vorzuenthalten. Im Gegensatz dazu geht man bei der optimistischen Nebenläufigkeitskontrolle davon aus, dass nur wenige Konflikte auftreten, die sich mit relativ geringem Aufwand auch im Nachhinein lösen lassen. Die optimistische Nebenläufigkeitskontrolle ist insbesondere in der Softwareentwicklung im Bereich der Quellcodeverwaltung verbreitet (vgl. Abschnitt 2.1.3).

Im Zusammenhang mit der optimistischen Nebenläufigkeitskontrolle sei das Prinzip der langen Transaktionen erwähnt. Lange Transaktionen sind laut AISH (2000) Transaktionen, die sich über einen längeren Zeitraum ziehen und das parallele Bearbeiten eines Datenbestandes in einem mehrköpfigen Team erleichtern sollen. Eine lange Transaktion beginnt mit der Auswahl eines Teildatensatzes des Datenbestandes durch einen Nutzer. Der Nutzer kann an diesem Teildatensatz mehrere kurze Transaktionen durchführen, die jedoch nicht öffentlich für andere Beteiligte einsehbar sind. Erst wenn der Nutzer seinen bearbeiteten Datensatz wieder in den Gesamtbestand integriert, endet die lange Transaktion. Typischerweise resultieren lange Transaktionen jedoch in einer größeren Anzahl an Konflikten, die gelöst werden müssen bei der Integration. Generell steigt nämlich die Wahrscheinlichkeit auf eine konfliktfreie Integration der jeweiligen Datensätze der Nutzer in den Gesamtdatenbestand mit der Frequenz an Integrationen (ESSER et al., 2023).

2.1.3 Version Control Systems (VCSs) am Beispiel von Git

In der Softwareentwicklung umfasst der etablierte Begriff der Versionskontrolle jede Art von Verfahren, das Änderungen an Dateien verfolgt und kontrolliert (POINET et al., 2020). Bei den Dateien handelt es sich dabei meistens um Quellcode. Mithilfe von Versionskontrolle lassen sich ausgewählte Dateien oder ein gesamtes Projekt auf einen früheren Zustand zurücksetzen, Unterschiede zwischen Dateiversionen feststellen und Verantwortlichkeiten von Projektbeteiligten festhalten. Um die Zusammenarbeit in einem Softwareentwicklerteam zu erleichtern, kommen hierfür sogenannte Version Control Systems (VCSs) zum Einsatz (FIRMENICH et al., 2005).

Es gibt mehrere Arten von VCSs, die im Folgenden kurz vorgestellt werden. Lokale VCSs stellen die einfachste Art der Versionskontrolle dar. Dabei werden alle Änderungen an den relevanten Dateien in einer einfachen Datenbank lokal auf dem Computer des Nutzers verwaltet. Um jedoch mit anderen Personen auf anderen Systemen zusammenzuarbeiten, sind lokale VCSs nicht geeignet. Daher wurden zentrale VCSs entwickelt, welche die Dateiversionen auf einem zentralen Server verwalten. Die Projektbeteiligten können diese Dateienversionen von dem Server empfangen und selbst welche hochladen. Fällt jedoch der Server aus, ist keine Zusammenarbeit mehr möglich. Falls der Server beschädigt wird

und keine entsprechenden Sicherheitskopien angefertigt wurden, ist es sogar möglich, dass der gesamte Bestand an Dateiversionen verloren geht. Um diesem Problem entgegenzuwirken, wurden verteilte VCSs entwickelt. Auch hier werden die Dateiversionen auf einem zentralen Server gespeichert, doch anstatt der aktuellsten Version der Dateien laden die jeweiligen Nutzer nun den gesamten Bestand an Versionen herunter. Auf diese Weise können die Dateiversionen im Falle eines beschädigten Servers von jedem beliebigen Nutzerrechner zurückkopiert und der Server wiederhergestellt werden. (CHACON & STRAUB, 2014)

Git ist heutzutage eines der beliebtesten Beispiele für ein verteiltes VCS (POINET et al., 2020). CHACON und STRAUB (2014) heben als größten Unterschied zwischen Git und anderen VCSs die Art und Weise hervor, wie Git seine Daten speichert. Andere VCSs speichern Dateien und daraufhin alle inkrementellen Änderungen an diesen Dateien. Git hingegen speichert jedes Mal, wenn eine Datei verändert wird, die gesamte Datei erneut ab. Ein weiteres Merkmal von Git ist die Unterscheidung zwischen „blobs“ und „trees“. Die „blobs“ repräsentieren die abgespeicherten Dateien und sind unveränderlich. Die „trees“ stellen die Ordnerstruktur dar, in der die „blobs“ abgespeichert werden. Ein „tree“ listet den Inhalt eines Ordners auf und welche Dateinamen zu welchem „blob“ gehören. Um Konflikte zwischen zwei Versionen einer Datei, die von unterschiedlichen Nutzern verändert worden ist, zu vermeiden, nutzt Git die in Abschnitt 2.1.2 erwähnte optimistische Nebenläufigkeitskontrolle. Die in der Softwareentwicklung auftretenden Konflikte basieren nämlich vor allem auf einzelnen Quelltextzeilen, welche untereinander typischerweise nur wenig vernetzt sind. Aufgrund dieses niedrigen Aggregationsgrads von Quellcode lassen sich auftretende Konflikte üblicherweise mit relativ geringem Aufwand lösen (SCHAPKE et al., 2021). Das Zusammenführen von Änderungen, die an verschiedenen Versionen derselben Datei getätigt worden sind, wird auch als Merge bezeichnet (CHACON & STRAUB, 2014).

2.2 Geometrische Modellierung von Volumenobjekten

Die Arbeit mit dreidimensionalen Geometrien ist einer der wesentlichen Bestandteile von Building Information Modeling (BORRMANN & BERKHAHN, 2021). Um die 3D-Geometrien zu beschreiben, werden in erster Linie Volumenmodelle eingesetzt, die sich aus Volumenkörpern zusammensetzen und mithilfe verschiedener Ansätze im Computer beschrieben werden können (VILGERTSHOFER, 2022).

Bei der Darstellung der Volumenkörper kann man zwischen direkten und indirekten Darstellungsschemata unterscheiden. Direkte Darstellungsschemata beschreiben das Volumen selbst, wohingegen indirekte Darstellungsschemata das Volumen nicht unmittelbar, sondern anhand der Oberflächen oder Kanten des Volumenkörpers beschreiben. Die Oberflächen beim indirekten Schema setzen sich dabei aus Teilflächen zusammen, welche geschlossen und orientierbar sein müssen. Die Geschlossenheit verhindert Löcher

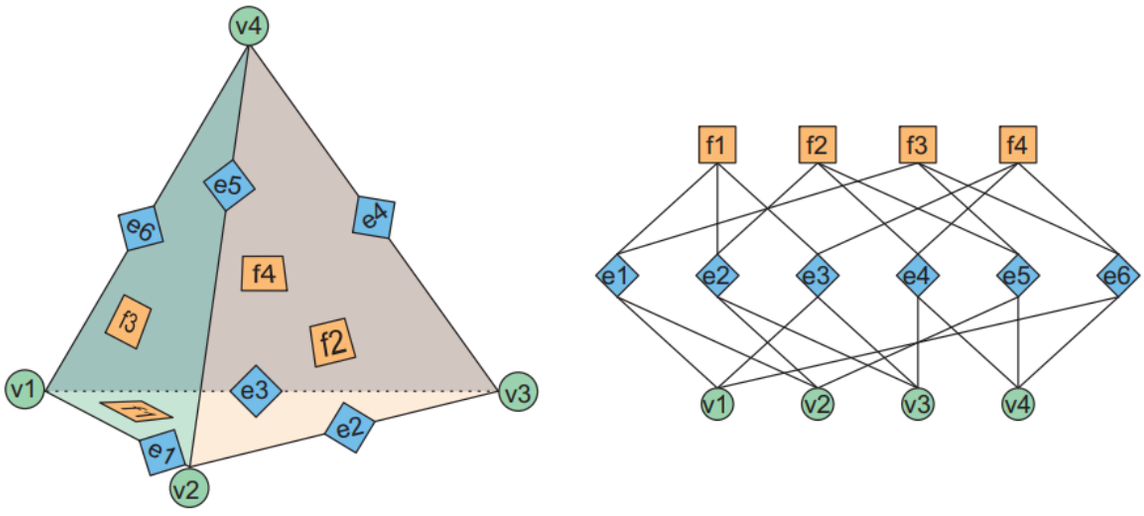
in den Teilflächen, während die Orientierung dafür sorgt, dass zwischen innerhalb und außerhalb des Volumenkörpers unterschieden werden kann. (BUNGARTZ et al., 2002)

Neben der Darstellung der Volumenkörper lässt sich zusätzlich auch noch der Prozess von deren Modellierung beschreiben. Dabei kann man die unterschiedlichen Ansätze in explizite und implizite Verfahren unterteilen (BORRMANN & BERKHAHN, 2021). Bei der expliziten Beschreibung eines Volumenkörpers wird dessen Oberfläche bzw. dessen Volumen als Endresultat eines Modellierungsprozesses beschrieben. Der Körper wird dann zum Beispiel indirekt über dessen Oberflächen beschrieben. Wird der Volumenkörper hingehen mithilfe eines impliziten Verfahrens beschrieben, so wird nicht der finale Körper aufgezeichnet, sondern dessen Konstruktionshistorie. Das heißt, die einzelnen Konstruktionsschritte, die nötig waren, um den entsprechenden Körper zu erzeugen, werden in ihrer Abfolge gespeichert. So bleibt der Modellierungsprozess nachvollziehbar und rekonstruierbar. (VILGERTSHOFER, 2022)

2.2.1 Explizite Verfahren

Die Boundary Representation (BRep) ist die am weitesten verbreitete Darstellungsform von Volumenkörpern im Computer. Die Oberfläche des Volumenkörpers wird hier mithilfe einer Hierarchie aus Knoten (engl. vertices), Kanten (engl. edges) und Flächen (engl. faces) beschrieben. Von einer Hierarchie spricht man deswegen, weil jedes Element über seine randbildenden Elemente der nächst tieferen Ebene beschrieben wird: Der Körper selbst wird über seine Oberflächen, die Flächen über ihre begrenzenden Kanten und die Kanten über ihren jeweiligen Start- und Endknoten beschrieben. Die Beziehungen der Flächen, Kanten und Knoten zueinander spiegeln sich in der Topologie des Körpers wieder, während die Geometrie des Körpers durch die Koordinaten der Knoten festgelegt wird. Die Topologie lässt sich mithilfe eines sogenannten Vertex-Edge-Face-Graphen (vef-Graph) darstellen (vgl. Abb. 2.1). (BORRMANN & BERKHAHN, 2021)

Einen stark vereinfachten Spezialfall der BRep stellt die triangulierte Oberflächenbeschreibung dar. Sie ist eines der einfachsten expliziten Verfahren zur Beschreibung eines Volumenkörpers und findet oft Verwendung bei visualisierungsnahen Anwendungen. Hierbei wird die Oberfläche des Volumenkörpers mithilfe eines Netzes aus Dreiecken beschrieben. Ebene Flächen können dadurch zwar exakt beschrieben werden, sobald die Flächen jedoch Krümmungen aufweisen, können sie nur noch approximiert werden. Die Genauigkeit der Approximation kann dabei durch Erhöhung der Dreiecksanzahl und gleichzeitiger Verringerung der Dreiecksgrößen beliebig erhöht werden. Mit zunehmender Anzahl an Dreiecken steigt jedoch auch der Speicheraufwand. (BORRMANN & BERKHAHN, 2021) Eine triangulierte Oberflächenbeschreibung wird auch als Mesh (dt. Netz) bezeichnet.



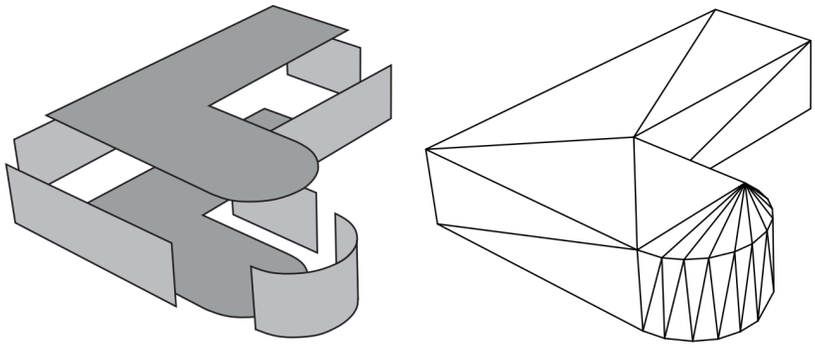
solid	faces
1	1,2,3,4

face	edges
1	1, 2, 3
2	2, 4, 5
3	1, 5, 6
4	3, 4, 6

vertex	coordinates
1	2, 0, 0
2	0, 0, 0
3	3, 0, 0
4	1, 1, 3

edge	vertices
1	1, 2
2	2, 3
3	3, 1
4	3, 4
5	2, 4
6	1, 4

Abbildung 2.1: Eine einfache BRep-Datenstruktur gefüllt mit Daten zur Beschreibung einer Pyramide. Der Vertex-Edge-Face-Graph beschreibt die Beziehungen zwischen Knoten, Kanten und Flächen und damit die Topologie des Körpers. Quelle: BORRMANN und BERKHAHN (2021)



(a) (b)

Abbildung 2.2: Ein Volumenmodell, das mittels allgemeiner Brep (a) und triangulierter Oberflächenbeschreibung (b) repräsentiert wird. Quelle: VILGERTSHOFER (2022)

2.2.2 Implizite Verfahren

Als klassisches Beispiel für die implizite Beschreibung eines Volumenkörpers nennen BORRMANN und BERKHAHN (2021) die Methode der Constructive Solid Geometry (CSG). Hierbei werden verschiedene einfache Grundkörper, sogenannte Primitive, mithilfe der mengentheoretischen booleschen Operationen Vereinigung ($\cup$), Schnitt ($\cap$) und Differenz ($\setminus$) miteinander kombiniert. Bei den Primitiven kann es sich beispielsweise um Quader,

Zylinder oder Kugeln handeln. In der Regel sind die Abmessungen dieser Grundkörper parametrisiert und können somit leicht verändert werden. Der Aufbau eines Volumenkörpers mittels schrittweiser Kombination von Primitiven kann in einem sogenannten CSG-Baum eindeutig beschrieben werden. Ein CSG-Baum ist laut ROMBERG (2005) ähnlich wie ein binärer Baum aufgebaut, wobei dessen äußere Knoten (Blätter) die Primitive und die inneren Knoten die durchgeführten Operationen repräsentieren. Viele 3D-CAD- und BIM-Systeme haben zwar das grundlegende Prinzip der booleschen Operationen übernommen, doch statt fest vorgeschriebener Primitive erlauben sie auch vom Anwender zuvor modellierte Volumenkörper als Operanden (BORRMANN & BERKHAHN, 2021). Dadurch wird der einschränkende Zwang zur Nutzung der Primitive ersetzt durch die Möglichkeit zur intuitiven Modellierung von komplexen dreidimensionalen Körpern. Im BIM-Bereich spielt das vor allem bei der Modellierung von Aussparungen und Durchbrüchen eine wesentliche Rolle, bei der vom Endnutzer erstellte 3D-Geometrien als Abzugskörper verwendet werden können. Ein wesentlicher Vorteil der CSG liegt darin, dass der Herstellungsprozess des Körpers konkret dargestellt wird und somit leicht nachvollziehbar ist. Ein Nachteil stellt jedoch die Tatsache dar, dass ein und derselbe Körper durch unterschiedliche CSG-Bäume beschrieben werden kann, wodurch der Vergleich von Volumenkörpern erschwert wird (NEUBERG, 2004).

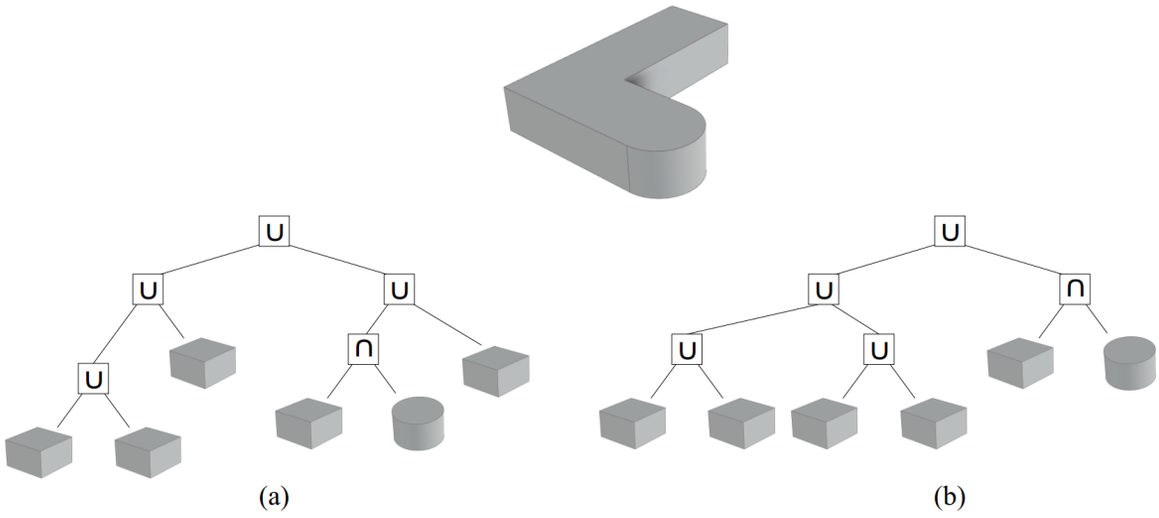


Abbildung 2.3: Der oben dargestellte Volumenkörper kann durch die in (a) und (b) dargestellten CSG-Bäume repräsentiert werden. Als Primitive werden Quader und Zylinder genutzt. Quelle: VILGERTSHOFER (2022)

Ein weiteres implizites Verfahren zur Modellierung von Volumenkörpern stellen Extrusions- und Rotationsverfahren dar. Dreidimensionale Körper werden dabei mithilfe zweidimensionaler Flächen erzeugt. Laut BORRMANN und BERKHAHN (2021) besteht das grundlegende Prinzip dieser Verfahren darin, eine zweidimensionale Geometrie entlang einer vom Anwender vorgegebenen 3D-Kurve (Pfad) zu führen, wobei der Raum, der von der 2D-Geometrie überstrichen wird, den Volumenkörper beschreibt. Wenn der Pfad eine Gerade ist, spricht man von einer Extrusion, bei gekrümmten Pfaden von einem Sweep. Beim Sweep wird im Vorfeld definiert, ob die 2D-Geometrie parallel zur ihrer Ausgangsposition oder allzeit senkrecht zum Pfad geführt werden soll. Analog dazu wird bei der Rotation die

2D-Geometrie um eine vom Anwender vorgegebene Achse rotiert und somit ein Rotationskörper erzeugt. Beim Lofting werden vom Endnutzer mehrere zweidimensionale, im Raum hintereinander liegende Geometrien erzeugt, die allesamt vom finalen Volumenkörper als Querprofile durchlaufen werden sollen. Die Formen der jeweiligen Flächen können sich dabei stark unterscheiden. Um den 3D-Körper zu erzeugen, interpoliert das CAD- oder BIM-System zwischen den 2D-Geometrien.

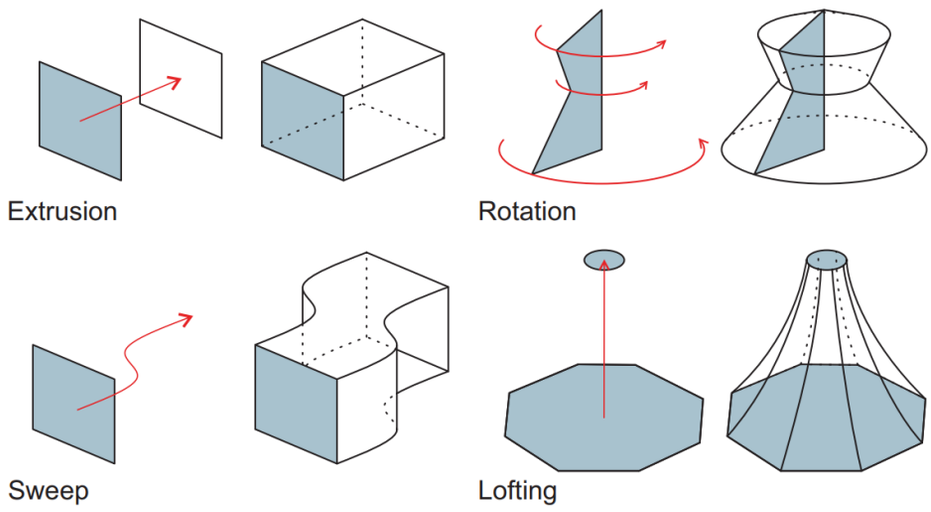


Abbildung 2.4: Extrusions- und Rotationsverfahren zum Erzeugen von Körpern aus Flächen. Quelle: BORRMANN und BERKHAHN (2021)

2.2.3 Vergleich von expliziten und impliziten Verfahren

Beim Vergleich von expliziten mit impliziten Verfahren lassen sich drei Wunschkriterien aufstellen: die Nachvollziehbarkeit der Modellierungsschritte, die einfache Modifizierbarkeit des Volumenkörpers und der reibungslose Datenaustausch.

Die Nachvollziehbarkeit der Modellierungsschritte ist bei expliziten Repräsentationen nicht gegeben, wohingegen bei impliziten Repräsentationen der Erzeugungsprozess gespeichert wird.

Was die Modifizierbarkeit angeht, ist bei expliziten Darstellungen nur das direkte Editieren der Geometrie möglich, was gegebenenfalls erheblichen Aufwand mit sich bringt. Es muss sichergestellt werden, dass durch die Veränderungen keine Fehler in der Oberflächenbeschreibung des Volumenkörpers (z.B. Lücken) entstehen, und möglicherweise muss eine große Anzahl an Kontrollpunkten editiert werden (VILGERTSHOFER, 2022). Bei impliziten Darstellungen ist die Modifizierbarkeit hingegen vergleichsweise einfach. Hier kann der Körper durch Editieren der Konstruktionsschritte modifiziert werden, was eine automatisierte Rekonstruktion des Körpers mit sich bringt. Bei komplexen Körpern kann dieser Prozess jedoch sehr rechenintensiv sein und es ist möglich, dass aufgrund eines veränderten Konstruktionsschritts die darauf folgenden Konstruktionsschritte nicht mehr fehlerfrei ausführbar sind und ebenfalls editiert werden müssen (BORRMANN & BERKHAHN, 2021).

Im Hinblick auf Datenaustausch nennen BORRMANN und BERKHAHN (2021) zunächst die geringen zu übermittelnden Datenmengen der impliziten Repräsentationen im Vergleich zu den expliziten. Jedoch führen sie als wesentliches Problem der impliziten Verfahren die schwierige Interpretierbarkeit der übermittelten Daten auf Empfängerseite an. Der Empfänger der 3D-Objekte muss Softwarewerkzeuge nutzen, die in der Lage sind, die implizit beschriebenen Modellierungsschritte korrekt zu interpretieren und in gleicher Weise auszuführen. Expliziten Repräsentationen sind hingegen einfach zu interpretieren, da lediglich die finale Geometrie und Topologie eingelesen werden muss (VILGERTSHOFER, 2022).

2.3 Building Information Modeling (BIM)

2.3.1 Allgemeine Grundlagen

Die Gebäudemodellierung mithilfe BIM hat in den letzten Jahren immer mehr Anwendung gefunden. In einer Umfrage im Jahr 2023 gaben von 723 befragten Berufstätigen im Bereich von Architecture, Engineering and Construction (AEC) 70% an, BIM bereits anzuwenden, während weitere 18% angaben, es innerhalb der nächsten fünf Jahre umsetzen zu wollen (»NBS«, 2023).

BORRMANN et al. (2021) beschreiben BIM als die durchgängige Nutzung eines digitalen Bauwerksmodells über den gesamten Lebenszyklus des Bauwerks. Der Lebenszyklus umfasst dabei den Entwurf, die Planung, die Ausführung, die Bewirtschaftung und schließlich auch den Um- bzw. Rückbau des Gebäudes. Das verwendete digitale Modell enthält neben der dreidimensionalen Geometrie der einzelnen Bauwerkselemente typischerweise auch mit der Geometrie verbundene semantische Informationen, wie beispielsweise das Material eines Bauteils oder die Beziehungen der Bauteile zueinander. Dabei werden die in Abschnitt 2.1.1 erläuterten Konzepte der objektorientierten Programmierung aufgegriffen, denn alle Bestandteile eines BIM-Modells sind Instanzen wohl definierter Objektklassen, wie beispielsweise Wand, Tür, Fenster usw. BIM hat als Ziel, den Datenaustausch innerhalb eines Bauprojekts zu verbessern und somit die Planungseffizienz zu steigern. Der Informationsaustausch basiert heutzutage überwiegend auf technischen Zeichnungen, die Bauwerksinformationen vor allem in grafischer Form wiedergeben. Die Konsistenz der verschiedenen technischen Zeichnungen kann häufig nur manuell geprüft werden und Unstimmigkeiten werden oft erst während der Bauausführung entdeckt. Wird jedoch die BIM-Methode angewendet, können Kollisionskontrollen zwischen den Fachmodellen verschiedener Gewerke durchgeführt werden, um Konflikte frühzeitig zu erkennen. Im Zusammenhang damit ermöglicht die dreidimensionale Modellierung eines Bauwerks das direkte Ableiten von konsistenten 2D-Plänen für Grundrisse und Schnitte, welche automatisch untereinander widerspruchsfrei sind. Des Weiteren erlaubt das Gebäudemodell eine präzise Mengenermittlung.

Objekte in einem BIM-Modell können auf vielerlei Hinsicht miteinander vernetzt sein, sowohl auf semantischer als auch auf geometrischer Ebene. Ein Beispiel hierfür sind die Objekte Wand und Tür. Wand und Tür gehen augenscheinlich eine semantische Beziehung miteinander ein, doch auch die 3D-Geometrien der beiden Objekte können Abhängigkeiten voneinander aufweisen. So stellt die Tür-Geometrie einen Durchbruch in der Wand dar, welcher beispielsweise mit der in Abschnitt 2.2.2 erwähnten CSG-Methode modelliert werden kann. Hierbei wird der Volumenkörper der durchbrochenen Wand erschaffen, indem der Volumenkörper der Tür von dem der Wand abgezogen wird. Somit sind die Objekte Wand und Tür auch auf geometrischer Ebene miteinander vernetzt.

Bei der technischen Umsetzung von BIM lassen sich zwei Unterscheidungen vornehmen. Die erste Unterscheidung wird vorgenommen mittels der von JERNIGAN (2008) etablierten Begriffe „BIG BIM“ und „little bim“. Bei „little bim“ erzeugt ein einzelner Planer im Rahmen seiner disziplinspezifischen Aufgabe mithilfe einer spezifischen BIM-Software ein digitales Bauwerksmodell. Das Modell wird jedoch nicht über verschiedene Softwareprodukte hinweg genutzt oder zur Koordination der Planung zwischen den beteiligten Fachdisziplinen herangezogen. Die Kommunikation zwischen den Disziplinen erfolgt weiterhin auf Grundlage von 2D-Zeichnungen. Werden jedoch von allen beteiligten Fachplanern digitale Modelle erzeugt und diese konsequent für den Datenaustausch und die Koordination untereinander genutzt, spricht man von „BIG BIM“.

Die zweite Unterscheidung wird mit den Begriffen „Closed BIM“ und „Open BIM“ vorgenommen. Dabei bezeichnet „Closed BIM“ die Erstellung des digitalen Bauwerksmodells, bei der ausschließlich Softwareprodukte eines einzelnen Herstellers eingesetzt werden. Für den Datenaustausch werden proprietäre Schnittstellen und Datenformate genutzt. Von „Open BIM“ spricht man hingegen, wenn Softwareprodukte verschiedener Hersteller zum Einsatz kommen und herstellerneutrale Formate für den Datenaustausch genutzt werden. Ein solches herstellerunabhängiges Datenformat wurde von der internationalen non-profit-Organisation buildingSMART erschaffen und heißt Industry Foundation Classes (IFC). Es beinhaltet umfangreiche Datenstrukturen, die die Beschreibung von Objekten aus nahezu allen AEC-Bereichen ermöglichen, und wurde 2013 in einen ISO-Standard überführt. Es ist jedoch anzumerken, dass die Nutzung herstellerneutraler Datenformate heutzutage noch nicht überall reibungslos funktioniert. Das ist darauf zurückzuführen, dass sowohl die Schaffung solcher Formate als auch deren Implementierung durch die Softwarehersteller technisch sehr anspruchsvoll sind. (BORRMANN et al., 2021)

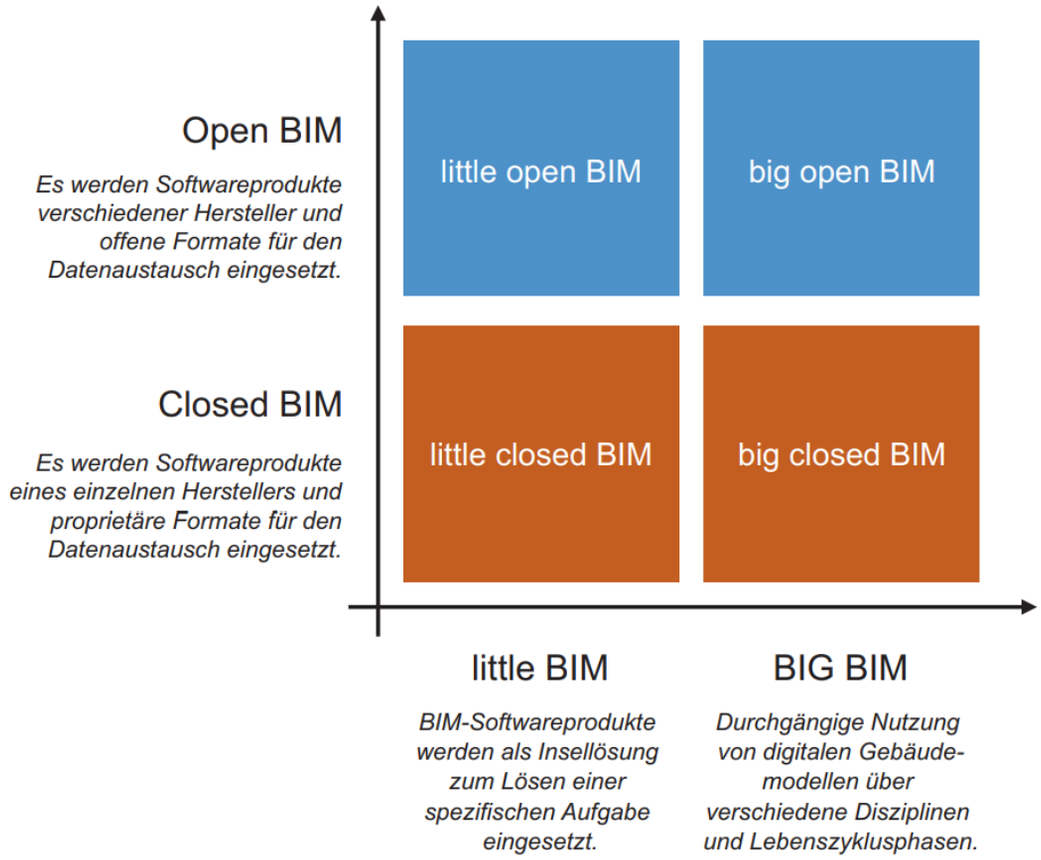


Abbildung 2.5: Die Breite des BIM-Einsatzes unterscheidet „little bim“ von „BIG BIM“. Je nachdem, ob herstellerneutrale Datenaustauschformate zum Einsatz kommen, spricht man von „Closed BIM“ oder „Open BIM“. Die Kombination dieser beiden Aspekte ergibt die hier gezeigte Matrix. Quelle: BORRMANN et al. (2021)

Um die Bauindustrie schrittweise an das Arbeiten mit BIG Open BIM heranzuführen, definiert die Norm ISO 19650-1 drei Stufen (ISO, 2018). Auf Stufe 1 existieren zwar bereits digitale 3D-Modelle für kritische Bereiche des geplanten Bauwerks, diese ergänzen jedoch lediglich die traditionellen 2D-Zeichnungen. Die Planer tauschen hierbei Daten untereinander aus, indem sie einzelne Dateien versenden.

Auf Stufe 2 werden von den Fachplanern jeweils eigene, voneinander unabhängige BIM-Modelle des geplanten Gebäudes erstellt, die regelmäßig miteinander abgeglichen werden. Auch auf dieser Stufe wird der Datenaustausch mittels dem Austausch monolithischer Dateien realisiert. Anders als auf Stufe 1 werden diese Dateien jedoch nicht mehr einzeln zwischen den jeweiligen Planern verschickt, sondern auf einer zentralen Projektplattform zusammengeführt und vorgehalten. Eine solche Plattform wird auch als Common Data Environment (CDE) bezeichnet und dient der Abstimmung und Koordinaten aller Projektbeteiligten. Daten in einer CDE werden in Informationscontainern zusammengefasst, deren Inhalte aus unterschiedlichsten Dateien wie Modellen, Plänen oder Protokollen bestehen können. Näheres zu CDEs folgt in Abschnitt 2.3.2.

Stufe 3 beschreibt das Arbeiten mit BIG Open BIM. Hier werden integrierte digitale Bauwerksmodelle verwaltet mithilfe von CDEs, die Daten nicht mehr als Informationscontainer auf Basis von monolithischen Dateien verwalten, sondern als einzelne Objekte. Die

Informationsverwaltung ist auf Stufe 3 also um einiges feingranularer als auf den vorhergehenden Stufen und erlaubt somit ein besseres Zugriffs- und Änderungsmanagement durch den Endnutzer. Aufgrund mangelnder technologischer Entwicklung hat sich Stufe 3 im Gegensatz zu Stufe 2 bislang noch nicht in der Bauindustrie etablieren können. Jedoch gibt es mehrere Ansätze, die sich dieser Herausforderung annehmen, von denen einer im Rahmen dieser Arbeit analysiert werden soll.

2.3.2 Modellbasierte Kollaboration mithilfe Common Data Environments (CDEs)

Auch ein Gebäudemodell stellt einen Datenbestand im Sinne von Abschnitt 2.1.2 dar. Hier sind Transaktionen immer dann notwending, wenn Änderungen an den Modellinhalten vorgenommen werden (AMANN et al., 2021). Bei der Kollaboration in Zusammenhang mit Gebäudemodellen existieren grundsätzlich zwei Ausprägungen: die durchgehende Verwendung eines einzigen, gemeinsamen, zentralen, monolithischen Modells auf der einen Seite und die Föderation von domänenspezifischen Modellen auf der anderen.

Um technische Inkonsistenzen und fachliche Widersprüche zu vermeiden, wird bei der Arbeit mit einem zentralen Modell meist auf die pessimistische Nebenläufigkeitskontrolle zurückgegriffen. Dabei extrahiert ein Planer einen Teildatensatz aus dem zentralen Modell, woraufhin die enthaltenen Objekte mit einer Sperre im zentralen Modell belegt werden und erst wieder von anderen Planern bearbeitet werden können, wenn der Teildatensatz zurück in das Gesamtmodell integriert wurde (SCHAPKE et al., 2021). Inwieweit Modellinhalte bei dieser Art der Nebenläufigkeitskontrolle parallel bearbeitet werden können, hängt von dem Ausmaß der Sperren ab. So könnte beispielsweise das gesamte Modell während der Bearbeitung mit einer Sperre belegt werden oder nur einzelne Elemente. Doch selbst wenn nur ein einzelnes Element des zentralen Modells gesperrt wird, kann es zu Koordinationsproblemen kommen. Beispielsweise kann ein Statiker darin gehindert werden, einen Durchbruch in einem Wandobjekt einzufügen, weil dieses bereits von einem Gebäudetechniker gesperrt worden ist, der in dieser Wand momentan eine Leitung plant. So hat sich in der Praxis gezeigt, dass die durchgehende Verwendung eines einzigen, gemeinsamen, zentralen Modells nicht zu empfehlen ist. Denn dadurch werden sowohl die zeitlich unabhängige Parallelbearbeitung von Planungsaufgaben als auch das Klären von Verantwortlichkeiten für Planungsfehler erschwert (PREIDEL et al., 2021).

Stattdessen wird die Kollaboration heutzutage meist mithilfe der Föderation von domänenspezifischen Modellen umgesetzt. Dabei werden lose zusammenhängenden Fachmodelle von den jeweiligen Fachplanern bearbeitet und in regelmäßigen Abständen zur Koordination und Konfliktbehebung zusammengeführt, um so das Gesamtmodell zu bilden (PREIDEL et al., 2021). Dies führt unweigerlich zu einer großen Anzahl an Schnittstellen, die mithilfe der optimistischen Nebenläufigkeitskontrolle koordiniert werden müssen, um so die Konsistenz und Gültigkeit des Gesamtmodells sicherzustellen. Die einzelnen Fachplaner haben hierbei zwar mehr Freiheiten bei der gleichzeitigen Bearbeitung ihrer jeweiligen Modellinhalte, doch sobald die Teilmodelle wieder in einem integrierten Modell zusam-

mengeführt werden sollen, sind umfassende Lösungsstrategien nötig, um die Konflikte zwischen den bearbeiteten Bauwerkselementen zu beheben (SCHAPKE et al., 2021).

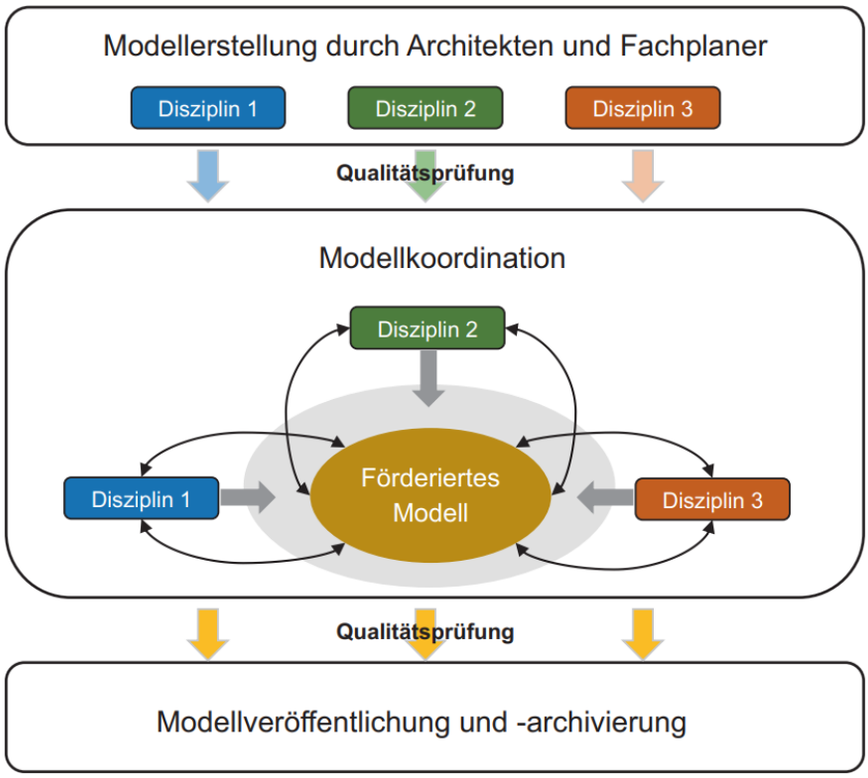


Abbildung 2.6: Prinzip der fachmodellbasierten Zusammenarbeit auf Basis eines föderierten Modells. Quelle: PREIDEL et al. (2021)

Um den hohen Anforderungen an das Informations- und Datenmanagement bei der fachmodellbasierten Zusammenarbeit gerecht zu werden, sind digitale Kollaborationsplattformen gut geeignet. In diesem Zusammenhang wird von der ISO 19650-1 (ISO, 2018) der Begriff Common Data Environment (CDE) (dt. gemeinsame Datenumgebung) definiert. Eine CDE stellt einen zentralen Raum für die Sammlung, Strukturierung, Zusammenführung, Verteilung, Verwaltung, Auswertung und Archivierung digitaler Informationen im Rahmen eines ganzheitlichen modellbasierten Projektmanagements dar. Sie speichert alle domänenspezifischen Fachmodelle und Dokumente, die für die Koordination und Durchführung eines Projekts notwendig sind. Dabei stehen alle in einer CDE gespeicherten Daten allen Projektbeteiligten zu jeder Zeit zur Verfügung („single source of truth“). Ein weiterer Vorteil einer CDE ist das reduzierte Risiko von Datenredundanz aufgrund der Zentralisierung der Datenhaltung. Eine CDE muss nicht in der Lage sein einzelne Objekte eines BIM-Modells aufzulösen. Die kleinsten Einheiten, die von einer CDE verwaltet werden, bilden sogenannte Informationscontainer. Diese enthalten eine Reihe monolithischer Dateien wie Modelle, Planunterlagen oder sonstige Dokumente. Jedoch werden zukünftige CDEs entsprechend BIM-Stufe 3 (vgl. Abschnitt 2.3.1) die Verwaltung von Informationen auf feingranularerer Ebene ermöglichen, d. h. den direkten Zugriff auf Objekte eines BIM-Modells erlauben. (PREIDEL et al., 2021) Die ISO 19650-1 spezifiziert als Kernanforderung an eine CDE, dass einem Informationscontainer ein klarer, standardisierter Zustand (Status) zugewiesen werden muss, um den Reifegrad und die

Zuverlässigkeit der darin gespeicherten Informationen zu kennzeichnen. Die möglichen Zustände lauten *in Bearbeitung*, *geteilt* und *veröffentlicht*. Weiterhin dürfen die Übergänge zwischen den jeweiligen Status erst nach der Durchführung entsprechender Prüf- und Freigabeprozesse erfolgen.

2.3.3 BIM-Softwareprodukte

Arten von BIM-Softwareprodukten

Im Bereich von BIM spielen viele verschiedene Arten von Softwareprodukten eine Rolle. Eine große Kategorie bilden dabei BIM-Autorensysteme wie »Revit« (2024), welche die Möglichkeit bieten, neue BIM-Modelle zu erschaffen und bestehende zu modifizieren. Aktuell verfügbare BIM-Autorensysteme sind primär für den Einsatz in der Neubauplanung gedacht (PETZOLD & RECHENBERG, 2021). Neben Autorensystemen existieren auch BIM-Werkzeuge, die nur einen Lesezugriff auf bestehende Modelle erlauben. Mit solchen Softwareprodukten kann ein Modell zwar eingesehen werden, aber nicht verändert oder erstellt werden. Ein Beispiel hierfür stellt »BIMvision« (2024) dar, mit welchem Modelle, die mit beliebten Autorensystemen verschiedener Herstellller im IFC-Format erzeugt worden sind, eingesehen werden können.

Autorensysteme lassen sich in parametrische und katalogorientierte Werkzeuge kategorisieren. Bei der parametrischen Modellierung werden geometrische Modelle mit Abhängigkeiten und Zwangsbedingungen versehen, sodass sie schnell und aufwandsarm an veränderte Randbedingungen angepasst werden können (BORRMANN & BERKHAHN, 2021). Als Parmeter können dabei beispielsweise die Abmessungen oder die Ausrichtung geometrischer Objekte hergenommen werden. Bislang wird die parametrische Modellierung hauptsächlich von reinen 3D-Modellierern im CAD-Bereich umgesetzt, wohingegen BIM-Autorenwerkzeuge das Konzept der Parametrik nur eingeschränkt umsetzen. BIM-Softwareprodukte nutzen häufig umfangreiche Bauteilkataloge, mit deren Hilfe der Anwender die Elemente seines Modells zusammenstellen kann. Vor allem im Hochbaubereich existieren bereits umfangreiche Bauteilkataloge und Vorlagen seitens der Softwarehersteller (SCHÄFERHOFF et al., 2021).

Revit als Beispiel für ein BIM-Autorensystem

Die BIM-Software Revit ist ein Autorensystem vom Hersteller Autodesk, das sowohl katalogorientiert ist als auch die parametrische Modellierung bis zu einem gewissen Grad umsetzt (BORRMANN & BERKHAHN, 2021). Diese Software wird für einen Großteil der Versuche im Rahmen dieser Arbeit verwendet und einige ihrer grundlegenden Konzepte sollen nun näher beleuchtet werden. Dabei basieren die nachfolgenden Erläuterungen hauptsächlich auf der Website »Revit API Docs« (2024), welche Autodesk als umfassende Informationsgrundlage in Bezug auf Revits Programmierschnittstelle, auch Application Programming Interface (API) genannt, bereitstellt. Unter anderem können sich Entwickler,

die benutzerdefinierte Plug-ins oder Anwendungen für Revit programmieren wollen, auf dieser Website über die Grundstruktur von Revit informieren.

Die Elemente des Bauteilkatalogs in Revit werden in Kategorien unterteilt, die dazu dienen die Bauteile nach ihrer Funktion oder ihrem Zweck zu gruppieren. Beispiele hierfür sind die Kategorien „Architektur“ und „Landschaftsgestaltung“. Die Kategorie „Architektur“ umfasst Bauteile, die typischerweise in architektonischen Entwürfen verwendet werden, wie beispielsweise Wände, Türen, Fenster usw. Unter der Kategorie „Landschaftsgestaltung“ hingegen finden sich Bauteile wie Zäune oder Wege, aber auch Objekte für Vegetation und Gelände. Innerhalb einer allgemeinen Kategorie wie „Architektur“ gibt es Unterkategorien, um verschiedene Arten von Bauteilen weiter zu differenzieren. So können beispielsweise Wände in die Unterkategorien „Tragende Wände“ und „Nichttragende Wände“ unterteilt werden.

Revit ist objektorientiert (vgl. Abschnitt 2.1.1). Das bedeutet, in Revit existieren Klassen, deren Instanzen gemeinsam ein Gebäudemodell bilden. Eine der wichtigsten Klassen in Revit ist *Element*, der alle Bauteile eines Revit-Modells angehören. Diese Klasse definiert u. a. die Attribute *Id* und *UniqueId*. *Id* ist eine numerische Identifikation (ID), die von Revit automatisch für jedes neu erstellte Objekt generiert wird und dieses Objekt innerhalb eines bestimmten Revit-Modells eindeutig kennzeichnet. Das Attribute *UniqueId* hingegen ist eine ID in Form einer Zeichenfolge, welche ein Objekt auch über mehrere Revit-Modelle hinweg eindeutig kennzeichnet. Auch sie wird automatisch von Revit erstellt. Im Gegensatz zu *Id* bleibt *UniqueId* jedoch auch dann unverändert, wenn das entsprechende Objekt zwischen verschiedenen Revit-Projekten verschoben wird. Die Klasse *Element* definiert noch weitere Attribute, darunter auch weitere ID-Attribute, doch im Rahmen der späteren Analysen ist hauptsächlich *UniqueId* relevant.

Eine weitere Klasse in der Revit-API ist *Transform*. Sie dient dazu, die Translation, Rotation und Skalierung eines Modellelements in Bezug auf das interne Koordinatensystem zu beschreiben. Das interne Koordinatensystem bildet die Grundlage für die kartesische Positionierung aller Elemente in einem Revit-Modell. *Transform* enthält normalerweise die x-, y- und z-Koordinate eines Bauteilelements. Daneben enthält sie auch drei Winkelangaben, die die Rotation des Elements um die x-, y- und z-Achse beschreiben. Auch die Skalierungsfaktoren des Elements entlang dieser Achsen werden in entsprechenden Werten von *Transform* festgehalten. Die Revit-API ermöglicht es Entwicklern mittels *Transform*, die Position, Größe und Ausrichtung von Objekten im Modell programmgesteuert zu ändern.

Darüber hinaus enthält Revit „Host Objects“ und „Hosted Objects“, welche eine wichtige Rolle in der Hierarchie von Bauteilen spielen. Typischerweise sind „Host Objects“ größere Bauteile (z. B. Wände), die die kleineren “Hosted Objects“ (z. B. Türen) beherbergen können. „Hosted Objects“ sind in der Regel abhängig von ihrem jeweiligen „Host Object“ und können sich dynamisch an dessen Geometrie und Position anpassen. Beispielsweise passt sich die Tiefe eines Türrahmens mithilfe geeigneter Parametrik an die Wandstärke an. Anders als ihre Namen suggerieren, existieren für „Host Objects“ und „Hosted Objects“ keine eigenen Klassen in Revit. Stattdessen wird dieses Konzept mithilfe der Beziehungen

zwischen den Modellobjekten umgesetzt, genauer gesagt mittels deren Attributen und Methoden.

Um die eben erwähnten Attribute und Beziehungen von Revit-Objekten einsehen zu können, wird im Rahmen dieser Arbeit ein Plug-in für Revit namens »RevitDBExplorer« (2024) verwendet.

2.3.4 Inkrementelle Update-Verfahren im Bauwesen

Im Gegensatz zu Programmcode handelt es sich bei BIM-Modellen nicht um einfachen Text, sondern um hoch vernetzte Strukturen (ZHU et al., 2023). Ein BIM-Modell enthält typischerweise eine Vielzahl von Objekten, die verschiedene Beziehungen miteinander eingehen, um in der Gesamtheit die Komplexität und Abhängigkeiten von Objekten der gebauten Umwelt bestmöglich computerinterpretierbar abzubilden. Beispielsweise steht das Objekt „Tür“ in Beziehung zum Objekt „Wand“. Die Herausforderung besteht also darin, eine Versionskontrolle auf Basis des einem BIM-Modell zugrunde liegenden Objektnetzwerkes auszuarbeiten. In den letzten Jahren wurden mehrere Konzepte entwickelt, die sich dieser Herausforderung annehmen.

FIRMENICH et al. (2005) entwickelten bereits 2005 einen Ansatz, der das schon damals in der Softwareentwicklung etablierte Konzept der Versionskontrolle aufgriff und den Austausch ganzer Dateien im Falle von Planungsänderungen ablösen sollte. Sie erarbeiteten ein Vorgehen, mit welchem sich monolithische Dateien, die beispielsweise mit einer Software aus dem Bereich des CADs erstellt worden waren, in ein textbasiertes System zur Versionskontrolle abspeichern ließen. Somit konnten sie auf bereits existierende Funktionen des VCS zurückgreifen. Sie nutzten dabei die Methode der Serialisierung, welche strukturierte Objektmengen in Zeichenfolgen umwandelte.

Teile der damaligen Ideen wurden in den vergangenen Jahren von Speckle (»Speckle«, 2024) aufgegriffen. Speckle entstand aus dem Bedarf großer Planungs- und Ingenieurbüros für die Kopplung von Modellen für Datenintegration zwischen Fachanwendungen (MONDINO, 2021). Dabei handelt es sich um eine quelloffene Kollaborationsplattform für die Planung bebauter Umwelt, die die Zusammenarbeit zwischen den unterschiedlichen Beteiligten am Bauprojekt erleichtern soll (POINET et al., 2020). Speckle wurde ursprünglich 2016 am University College London von Dimitrie Stefanescu im Rahmen des InnoChain-Projektes, einem H2020 Marie Curie European Training Network, entwickelt (POINET et al., 2020). Speckles Hauptfunktionen sind Versionskontrolle, Schaffen von Kompatibilität zwischen den verschieden domänenspezifischen Softwareprodukten, Abspeichern von Autorschaft und Minimierung der zu übertragenden Datenmengen. Letzteres wird dadurch realisiert, dass im Falle einer Änderung lediglich die Informationen über die modifizierten Bestandteile eines Modells übermittelt werden müssen (POINET et al., 2020). Um den Datenaustausch zwischen den Endnutzern zu ermöglichen, nutzt Speckle Plug-ins, sogenannte Connectors, mit denen derzeit insgesamt 28 beliebte Softwareprodukte im Bereich von AEC um die nötigen Funktionalitäten erweitert werden können. Mit den Connectors lassen sich Daten in Textform mittels der in Abschnitt 2.1.2 erwähnten

langen Transaktionen an einen Speckle-Server senden und von diesem auch wieder empfangen, selbst wenn die Originalquelle bereits offline ist (POINET et al., 2020).

Abgesehen von Speckle gibt es noch andere Plattformen, die das Konzept von inkrementellen Update-Verfahren für BIM-Modelle aufgegriffen haben. Als erstes Beispiel sei hier Graphisoft's BIMcloud genannt, eine Plattform für die multidisziplinäre Zusammenarbeit im Bereich Design. Mithilfe ihrer „Delta-Server-Technologie" wird nicht die gesamte lokale Projektdatei an das zentrale, in der BIMcloud gespeicherte Datenmodell übermittelt, sondern lediglich die vorgenommenen Änderungen („Deltas"). Dabei können Projektbeteiligte Bereiche im zentralen Modell abonnieren, wodurch sie umgehend über dortige Veränderungen informiert werden und diese leicht auffinden können. Im Gegensatz zu Speckle werden die Softwareprodukte von Graphisoft jedoch kommerziell vertrieben und sind nicht quelloffen. Sie nutzen keine herstellerneutralen Datenformate, sondern beziehen sich auf die hauseigenen Formate. (»GRAPHISOFT, BIMcloud«, 2024)

ESSER et al. (2022) haben einen alternativen Ansatz ausgearbeitet, um inkrementelle Updates von BIM-Modellen zu ermöglichen. Ihre zentrale Idee besteht darin, BIM-Modelle in Form von Graphen zu repräsentieren, die zum Ziel haben, die komplexen und vernetzten Objektstrukturen innerhalb der Modelle besser abzubilden als textbasierte Ansätze. Änderungen am Modell bzw. dem entsprechenden Graphen werden mithilfe einer sogenannten Diff-Berechnung ermittelt und in Form einer Transformationsregel für den Graphen abgespeichert. Diese Transformationsregel kann anschließend an einen zentralen Server oder andere Projektbeteiligte verschickt werden, wo sie veraltete Graphen des Modells aktualisiert.

2.4 Einordnung

Wie die vorherigen Unterkapitel gezeigt haben, gibt es durchaus gute Gründe dafür, das durchgängig modellgestützte Arbeiten im Sinne von BIG Open BIM anzustreben. In der heutigen Baupraxis konnte dieses Konzept aufgrund mangelnder technologischer Entwicklung bisher noch nicht Fuß fassen. Es gibt jedoch Ansätze in Wissenschaft und Praxis, die Bemühungen in diese Richtung anstellen und inkrementell auf Objekte eines BIM-Modells zugreifen. Mehrere davon wurden bereits vorgestellt. Eines davon war das Konzept von Speckle, dessen Grundstruktur im nachfolgenden Kapitel erläutert wird. Konkret stellen sich dabei die Fragen, wie Speckle inkrementelle Updates in BIM-Modellen umsetzt, wie der Datenaustausch zwischen verschiedenen Softwareprodukten realisiert wird und wo die Vorzüge und Grenzen von Speckle liegen. Um diese Frage zu beantworten, werden mehrere Versuche mit unterschiedlichen Rahmenbedingungen und Anforderungen an Speckle durchgeführt.

Kapitel 3

Inkrementeller Datenaustausch mit Speckle

Die Informationen für dieses Kapitel wurden auf Grundlage der »Speckle Developer Docs« (2024), des »Speckle User Guide« (2024) und durchgeführter Versuche zusammengestellt. Die Speckle Developer Docs und der Speckle User Guide sind zwei Webseiten, auf denen die Entwickler von Speckle den Aufbau der Plattform dokumentieren.

3.1 Aufbau der Plattform

Speckle ist eine objektbasierte Plattform, d.h. Daten werden werden nicht als monolithische Dateien, sondern als einzelne Objekte gespeichert. Dabei lässt sich die Plattform in zwei wesentliche Bestandteile aufteilen: den Speckle-Server und die Connectors.

Die Daten bzw. Objekte werden auf einem Speckle-Server gespeichert und verwaltet. An den Server kann der Anwender Daten senden und von diesem auch wieder empfangen. Im Rahmen dieser Arbeit wird ein Server verwendet, den Speckle seinen Nutzern kostenlos zur Verfügung stellt. Bei der Organisation von Daten verfolgt Speckle einen Ansatz, der auch in der Softwareentwicklung von vielen Version Control Systems verfolgt wird. Dabei stellen sogenannte Streams die oberste Ebene der Organisation dar. Es bietet sich an, für jedes neue Bauprojekt einen eigenen Stream zu erstelllen. Beim Erstellen eines Streams wird diesem automatisch eine einzigartige, zehnstellige ID aus Hexadezimalziffern zugeordnet. Die nächsttiefere Organisationsebene stellen sogenannte Branches dar, die zur Untergliederung von Streams dienen. Beispielsweise kann der Stream eines großen Bauprojektes, das aus mehreren Gebäuden besteht, in die Branches „Gebäude A“ und „Gebäude B“ unterteilt werden. Einem Branch wird keine eigene ID zugeordnet. Auf der tiefsten Ebene der Datenorganisation befinden sich sogenannte Commits. Ein Commit beinhaltet die tatsächlichen Daten, die der Anwender an den Speckle-Server geschickt hat, und zusätzliche Informationen wie den Ersteller des Commits und den Zeitpunkt, an dem der Commit an den Server gesendet wurde. Jedes Mal, wenn Daten an den Server gesendet werden, wird ein neuer Commit erstellt und innerhalb des ausgewählten Branches gespeichert. Die Daten in einem Commit sind nach dem Hochladen nicht mehr veränderbar und können nur vom Server entfernt werden, indem der gesamte Commit gelöscht wird. Wie Streams erhalten auch Commits beim Erstellen automatisch eine einzigartige, zehnstellige ID bestehend aus Hexadezimalziffern.

Um Daten aus einer AEC-Software an den Speckle-Server zu senden und von diesem wieder in der gleichen oder einer anderen Software zu empfangen, stellt Speckle seinen Nutzern sogenannte Connectors zur Verfügung. Dabei handelt es sich um Plug-ins für beliebte Softwareprodukte im AEC-Bereich. Jedes Mal, wenn Daten bzw. Objekte mithilfe

eines Connectors an den Server gesendet werden, wandelt der entsprechende Connector die Objekte von ihrem ursprünglichen softwarespezifischen Format in Speckles eigenes softwareneutrales Format um. Beim Empfangen von Objekten vom Server geschieht die gleiche Konvertierung in umgekehrter Richtung, was die Übermittlung von Daten zwischen unterschiedlichen Softwareprodukten ermöglicht.

Sobald die Daten auf dem Speckle-Server angekommen sind, können diese mithilfe der webbasierten Schnittstelle von Speckle direkt im Browser eingesehen und verwaltet werden. Diese Schnittstelle bietet Speckle-Nutzern mehrere Funktionen. Beispielsweise werden dort hochgeladene CAD-Zeichnungen oder BIM-Objekte wie Wände, Türen usw. mittels dem 3D-Viewer von Speckle visualisiert. Der 3D-Viewer nutzt für die Visualisierung explizite Geometriedarstellungen, typischerweise Vielflächennetze basierend auf der triangulierten Oberflächenbeschreibung. Des Weiteren lässt sich mithilfe der webbasierten Schnittstelle verwalten, wer Zugriff auf einen bestimmten Stream hat und zu welchem Ausmaß die beteiligten Personen die im Stream enthaltenen Daten verändern können. POINET et al. (2020) beschreiben als weitere Funktion dieser webbasierten Schnittstelle eine interaktive Benutzeroberfläche namens „SpeckleViz“, die ihren Nutzern ein besseres Verständnis vom Datenfluss innerhalb eines spezifischen Projektes geben soll. Dabei soll graphisch visualisiert werden, wie sich das Projekt im Laufe der Zeit infolge der Beiträge unterschiedlicher Projektbeteiligter entwickelt hat. POINET et al. (2020) haben zwar das Konzept hinter „SpeckleViz“ erklärt, doch zum Zeitpunkt dieser Arbeit ist diese Benutzeroberfläche innerhalb der webbasierten Schnittstelle von Speckle nicht wiederfindbar.

Abbildung 3.1: Darstellung einer Tür, die aus dem Autorenprogramm Autodesk Revit stammt, im 3D-Viewer von Speckle

Zudem bietet Speckle seinen Nutzern drei verschiedene Software Development Kits (SDKs). SDKs sind Sammlungen von Programmierwerkzeugen und Programmbibliotheken, die Softwareentwickler dabei unterstützen, ihre eigenen Anwendungen auf Basis einer existierenden Software (in diesem Fall Speckle) zu entwickeln (»Nordic APIs«, 2023). Speckle bietet SDKs für die Programmiersprachen Python, C-Sharp und JavaScript, wobei im Rahmen dieser Arbeit ausschließlich das Python-SDK für Analysezwecke verwendet wird.

3.2 Speckles Datenstruktur

Wie in Abschnitt 3.1 bereits erwähnt, ist Speckle objektorientiert. Aus einer AEC-Software stammende Objekte, wie beispielsweise eine Revit-Wand oder eine AutoCAD-Linie, werden beim Senden an den Speckle-Server in Speckle-Objekte konvertiert. Die Speckle-Objekte können dabei auf eine spezifische Software (z.B. Revit) zugeschnitten sein oder mehrere Softwares umfassen (z.B. Geometrien im CAD-Bereich). Die softwarespezifischen Speckle-Objekte bieten jedoch typischerweise passende Attribute, die die Konvertierung in andere Softwareprodukte erlauben. Wie diese Speckle-Objekte aufgebaut und miteinander vernetzt sind, wird im Folgenden näher beleuchtet.

Ausgangspunkt fast aller Speckle-Objekte ist die Klasse *Base*. Von dieser erbt der Großteil aller weiteren Speckle-Klassen, d.h. zusätzlich zu ihren eigenen Attributen übernehmen sie alle Attribute der *Base*-Klasse (vgl. Abschnitt 2.1.1). Die *Base*-Klasse definiert insgesamt fünf Attribute, welche in Tabelle 3.1 dargestellt sind.

Tabelle 3.1: Die Attribute der *Base*-Klasse

Name	**Datentyp**	**Wert**
id	str	32-stellige ID aus Hexadezimalziffern
units	str	Maßeinheit des Speckle-Objekts
speckle_type	str	Klassenname des Speckle-Objekts
applicationId	str	ID des Objekts in seiner Ursprungssoftware
totalChildrenCount	int	Anzahl der Speckle-Objekte, die von diesem Speckle-Objekt referenziert werden

Die *id* eines jeden Speckle-Objekts ist einzigartig und wird von den Connectors auf Grundlage der restlichen Attribute des Objekts erzeugt. Sobald man eine Eigenschaft eines Objekts in seiner Ursprungssoftware verändert und das Objekt in einem Commit inkludiert, erzeugt der entsprechende Connector daraus ein vollständig neues Speckle-Objekt mit einer neuen *id*. Speckle-Objekte sind nach ihrer Erzeugung also unveränderlich. Das *id*-Attribut eines Speckle-Objektes wird im weiteren Verlauf dieser Arbeit als „Speckle-ID“ bezeichnet.

Beispiele für Klassen in Speckle sind *RevitMaterial* und *Collection*. Letztere unterscheidet sich von der erstgenannten, denn sie ist softwareunabhängig. Das bedeutet, Instanzen von *Collection* können in Commits enthalten sein, die von unterschiedlichen Connectors an den Speckle-Server gesendet wurden. Eine weitere Besonderheit ist, dass *Collection*-Objekte nicht zwingend ein entsprechendes Gegenstück in einer Ursprungssoftware benötigen und automatisch von den Connectors erzeugt werden. Dementsprechend wird das von der *Base*-Klasse geerbte Attribut *applicationId* bei Instanzen der Klasse *Collection* mit einem Nullwert belegt. Collections sind die generischsten Objekte in Speckle und werden universell von allen Connectors eingesetzt, um die in einem Commit enthaltenen

Daten hierarchisch und einheitlich zu strukturieren. Neben den von der *Base*-Klasse geerbten Attributen besitzt *Collection* drei weitere: *name*, *collectionType* und *elements*. Beachtenswert ist das Attribut *elements*, das sich in dieser Form u. a. auch in den Speckle-Equivalenten der Host Objects aus Revit wiederfindet. Dieses Attribut stellt einen Behälter für alle Speckle-Objekte dar, die von *Base* erben. Demzufolge kann es neben softwarespezifischen Speckle-Objekten wie *RevitMaterial* auch weitere Collections enthalten, was zu einer Verschachtelung von Speckle-Objekten führt. Der Aufbau von Speckles objektorientierten Datenstruktur und die oben geschilderten Beziehungen lassen sich mithilfe von UML-Diagrammen visualisieren (vgl. Abb. 3.2 und Abb. 3.3).

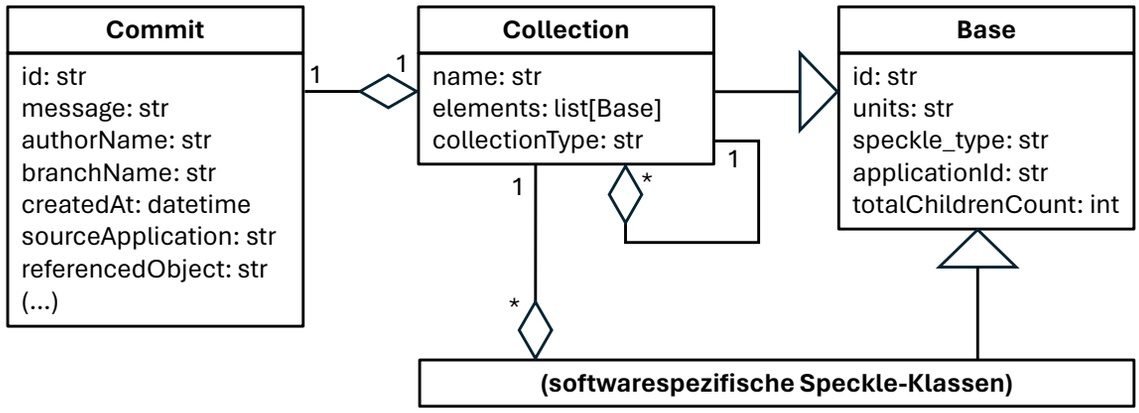


Abbildung 3.2: UML-Diagramm der Beziehungen zwischen den Speckle-Klassen *Base*, *Collection* und *Commit*

Bei näherer Betrachtung von Abb. 3.2 fällt auf, dass es eine eigene Speckle-Klasse für Commits gibt, diese jedoch nicht von der *Base*-Klasse erbt. Auf »Speckle Developer Docs« (2024) wird zwar festgehalten, dass auch die Klasse für Commits von *Base* erbt, tatsächlich ist das im verwendeten Python-SDK jedoch nicht der Fall. Näheres zu Commits folgt in Abschnitt 3.3.

Eine weitere Besonderheit des *elements*-Attributes der *Collection*-Klasse ist die Tatsache, dass dieses ein „detached property“ (dt. losgelöstes Attribut) ist. Losgelöste Attribute sind eine Funktion von Speckle, die es erlaubt, für die Werte dieser Attribute andere Speckle-Objekte als Referenzen einzusetzen. Ein Speckle-Objekt, das in einem losgelösten Attribut eines anderen Speckle-Objekts referenziert wird, ist Bestandteil des referenzierenden Objekts. Doch im Gegensatz zu den Werten der nicht-losgelösten Attribute kann dieses referenzierte Objekt auch Bestandteil anderer Speckle-Objekte sein. Trotzdem wird es nur einmal auf dem Server gespeichert und behält seine Einzigartigkeit. Der Vorteil von Referenzen wird anhand des folgenden Beispiels deutlich: In Revit wird eine Wand erstellt und in diese eine Tür eingefügt. Wand und Tür bestehen beide aus dem Material Holz. Beim Senden an den Server wandelt der Revit-Connector Tür, Wand und Holz jeweils in Speckle-Objekte um. Anstatt das Material als festes Attribut jeweils einmal im Wand- und einmal im Tür-Objekt zu speichern, speichert der Connector es nur einmal und belegt das jeweils losgelöste Attribut *material* von Wand und Tür mit einer Referenz zu dem

Material-Objekt (vgl. Abb. 3.4). Beachtenswert ist hierbei, dass das Material sowohl von dem Host Object als auch von dem Hosted Object referenziert werden kann.

Der Vollständigkeit halber wurde in Abb. 3.4 auch das Attribut *materialQuantities* eingefügt. Dieses Attribut ist nicht losgelöst und ein fester Bestandteil von gebauten Speckle-Objekten aus Revit. Sein Wert ist selbst ein Speckle-Objekt, welches neben den von *Base* geerbten Attributen u. a. die Attribute *area*, *volume* und *material* besitzt. Die Werte von *area* und *volume* geben an, wie viel Fläche und Volumen von dem Material in den entsprechenden Bauteilen (hier Wand und Tür) eingenommen wird.

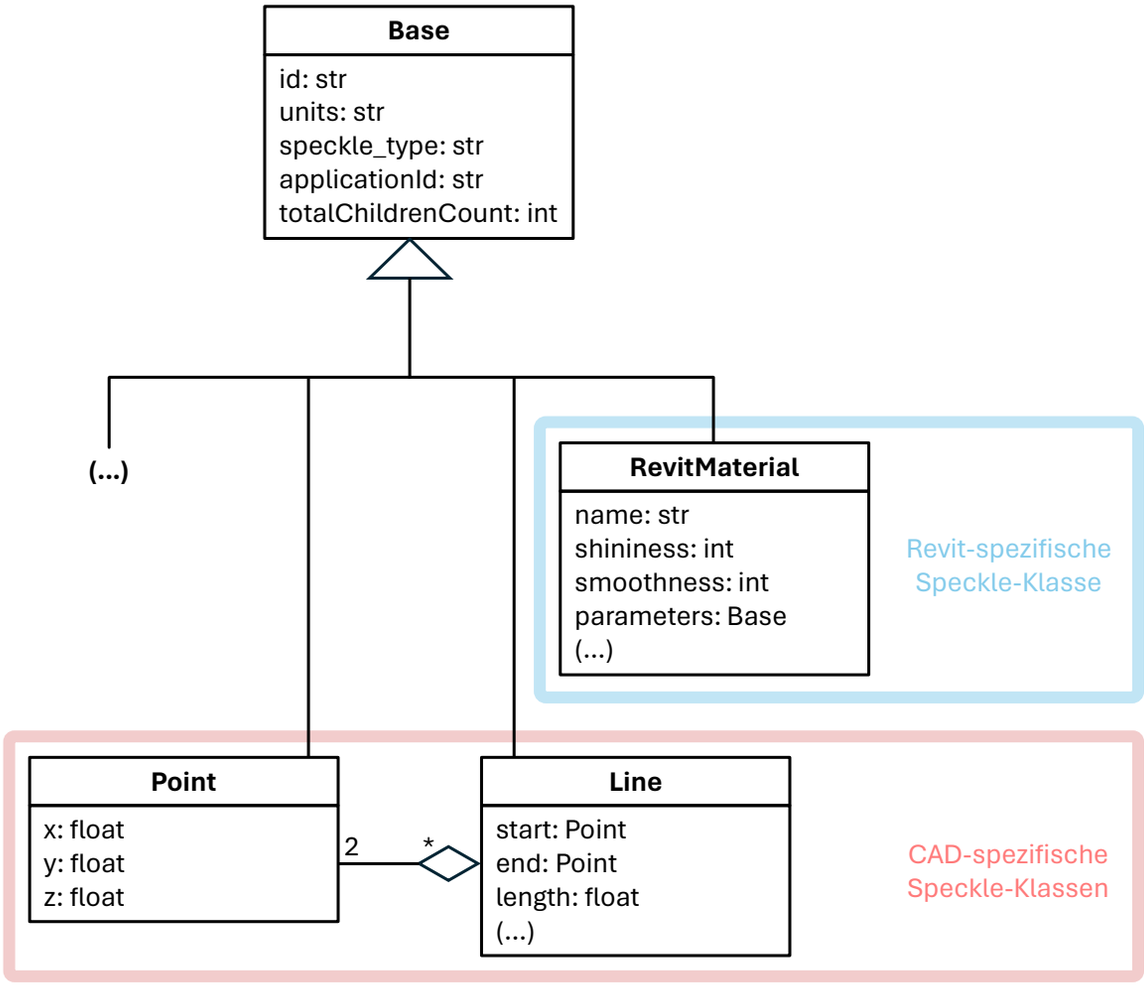


Abbildung 3.3: UML-Diagramm von *Base* und beispielhaften Speckle-Klassen, die von Base erben

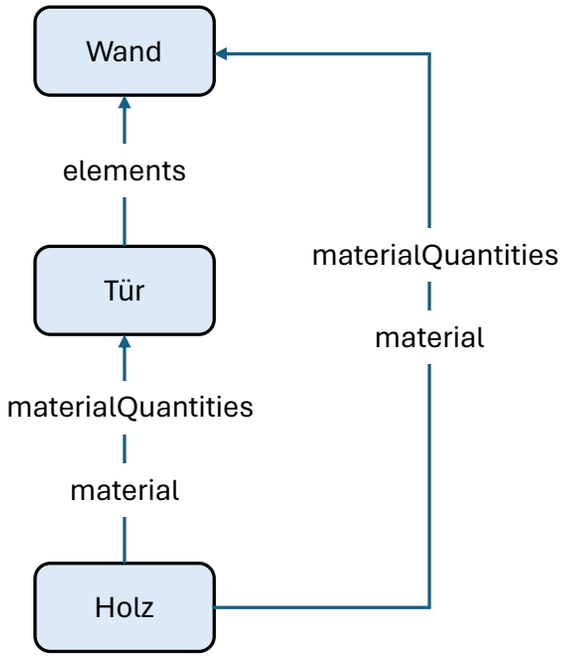


Abbildung 3.4: Vernetzung von Speckle-Objekten mittels Referenzen

3.3 Aufbau von Commits

Das wichtigste Attribut der *Commit*-Klasse ist *referencedObject*. Wie der Name bereits suggeriert, handelt sich dabei um ein losgelöstes Attribut. Der Wert von *referencedObject* ist immer ein Objekt der Klasse *Collection*, welches weitere Collections enthält. Meistens sind die für den Commit ausgewählten Objekte nämlich innerhalb mehrerer Collections verschachtelt. Der Grad der Verschachtelung hängt dabei von der Ursprungssoftware und dem konkreten Anwendungsfall ab. Beispielsweise erstellt der Revit-Connector für alle Revit-Objekte, die für den Commit ausgewählt worden sind, automatisch Collections entsprechend ihrer Bauteilkategorien in Revit. Der schematische Aufbau eines vom Revit-Connectors erstellten Commits und der darin enthaltenen Collection ist in Abb. 3.5 dargestellt.

Der AutoCAD-Connector hingegen erstellt die im Commit enthaltenen Collections basierend auf den Layern, die der Anwender zuvor in seinem AutoCAD-Projekt definiert hat (vgl. Abb. 3.6). Somit bleibt die vom Nutzer erstellte Struktur seiner Daten auch nach deren Umwandlung in das Speckle-Format erhalten.

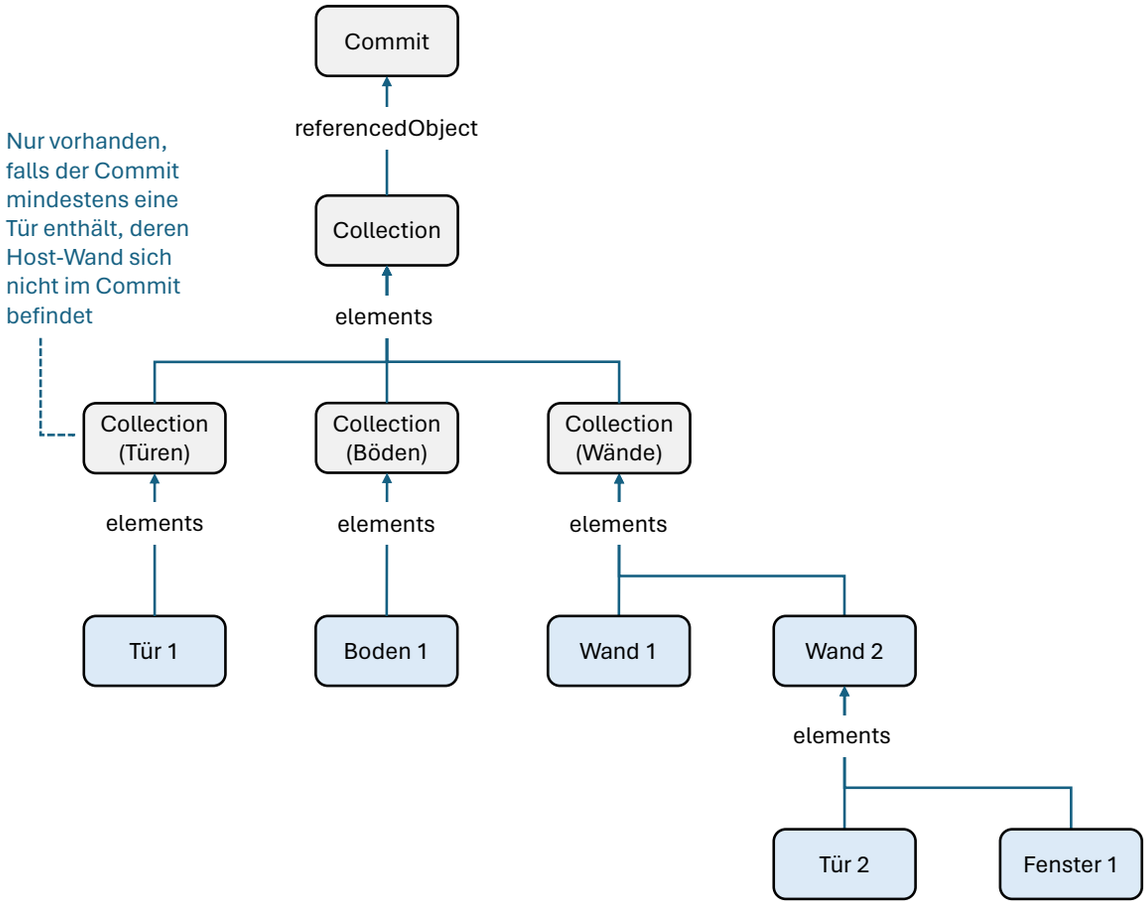


Abbildung 3.5: Aufbau eines vom Revit-Connector erstellten Commits. Angelehnt an »Speckle Developer Docs« (2024)

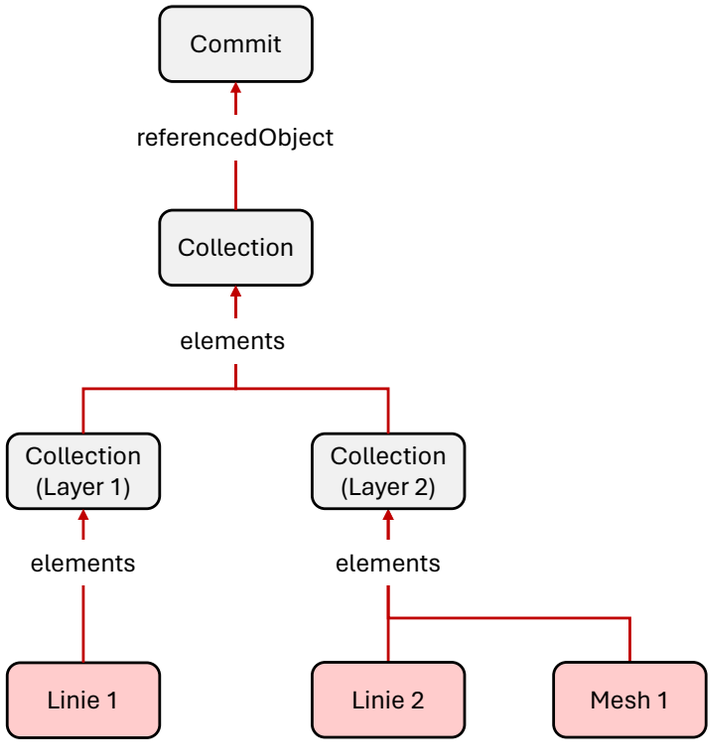


Abbildung 3.6: Aufbau eines vom AutoCAD-Connector erstellten Commits. Angelehnt an »Speckle Developer Docs« (2024)

Es ist mit Speckle nicht möglich, einen Löschen-Befehl in Form eines Commits an einen Stream zu senden, um damit beim Empfangen dieses Commits ein Objekt aus einem Modell zu entfernen. Wie in Abb. 3.5 und Abb. 3.6 zu erkennen, kann ein Speckle-Commit nur Objekte enthalten. Es ist auch nicht möglich, ein Commit-Objekt mit einer leeren Menge zu belegen, um damit ein bestehendes Objekt zu ersetzen. Des Weiteren können die Speckle-Connectors sowie das Python-SDK nur vollständige Commits empfangen. Es ist nicht möglich, einzelne Objekte aus einem Commit auszuwählen und nur die ausgewählten Objekte zu empfangen.

3.4 Serialisierung und Deserialisierung

Unter Serialisierung versteht man in der Informatik das Konvertieren von Objekten in ein speicherbares, transportierbares und sequenzielles Datenformat (»C++ FAQ«, 2024). Das Gegenstück dazu ist die Deserialisierung, bei der ein Objekt aus diesem Format rekonstruiert wird. Im Fall von Speckle wird die Aufgabe der Serialisierung und Deserialisierung von den Connectors übernommen. Nachdem der Anwender die gewünschten Objekte in seinem AEC-Softwareprojekt für seinen nächsten Commit ausgewählt hat, wandelt der entsprechende Connector die Objekte zunächst in Speckle-Objekte um. Daraufhin serialisiert er die Speckle-Objekte in das menschenlesbare Datenaustauschformat JavaScript Object Notation (JSON), um sie in dieser Form an den Server zu senden und dort zu speichern. Beim Empfangen eines Commits vom Server werden diese Schritte in umgekehrter Reihenfolge durchlaufen: Der Connector empfängt die JSON-Darstellungen der Speckle-Objekte, deserialisiert diese und wandelt sie anschließend in Objekte der entsprechenden AEC-Software um. In der JSON-Repräsentation eines Speckle-Objekts werden alle dessen Attribute und zugehörigen Werte gelistet.

Ein zusätzliches Attribut, das nicht im eigentlichen Speckle-Objekt aber in dessen JSON-Serialisierung enthalten ist, ist *__closure*. In diesem Attribut werden alle Speckle-IDs der Objekte aufgelistet, die von dem betrachteten Speckle-Objekt referenziert werden. Somit entspricht die Anzahl der in *__closure* aufgelisteten Speckle-IDs dem Wert von *totalChildrenCount*. Jede der referenzierten Speckle-IDs besitzt eine zugehörige ganze Zahl, die angibt, wie viele Ebenen an Referenzen durchlaufen werden müssen, bis das referenzierte Speckle-Objekt erreicht wird. Somit ist ein Speckle-Objekt aus der Sicht von Git (vgl. Abschnitt 2.1.3) „blob“ und „tree“ zugleich. Denn es enthält die nötigen Informationen um den vollständigen Graph aller von ihm referenzierten Speckle-Objekte zu konstruieren. Beispielsweise kann das Attribut *__closure* einer Speckle-Wand, die ursprünglich aus einem Revit-Modell stammt, die Speckle-ID einer Speckle-Tür enthalten. Da das Tür-Objekt ausgehend vom Wand-Objekt nach nur einer Referenzierung erreicht wird, besitzt die Speckle-ID der Tür den Wert 1. Jedoch referenziert das Tür-Objekt selbst u. a. ein Speckle-Objekt names *definition*, welches selbst wiederum ein Speckle-Objekt referenziert, das die explizite Mesh-Darstellung der Tür enthält. Dementsprechend befinden sich in *__closure* der Speckle-Wand auch die Speckle-ID von *definition* mit dem zugehörigen Wert 2 und die Speckle-ID der Mesh-Darstellung mit dem Wert 3.

3.5 Transports

Transports sind ein weiteres erwähnenswertes Konzept in Speckle. Transports geben einem Entwickler die Kontrolle darüber, wo und wie er seine Speckle-Objekte speichert. Neben dem Speckle-Server können Speckle-Objekte nämlich auch anderweitig gespeichert und abgerufen werden, beispielsweise in SQLite- oder MongoDB-Datenbanken. Die von Speckle zur Verfügung gestellten SDKs bieten Entwicklern neben bereits implementierten Transports auch die Möglichkeit, benutzerspezifizierte Transports zu programmieren. Auch die Speckle-Connectors verwenden Transports im Hintergrund, um die Datenübertragung zwischen dem Speckle-Server und der entsprechenden AEC-Software zu handhaben. Beispielsweise speichern die Connectors mithilfe von Transports beim Empfangen von Speckle-Objekten die IDs der neu in der AEC-Software erzeugten Objekte. Beim Speichern werden diese IDs den Speckle-IDs der empfangenen Speckle-Objekte zugeordnet.

Die *applicationId* eines Speckle-Objekt entspricht immer der ID des Objektes in seiner Ursprungssoftware. Wird durch Empfangen dieses Speckle-Objekts jedoch ein neues Objekt in der Ursprungssoftware erzeugt, stimmen ID des neu erzeugten Objekts und *applicationId* des Speckle-Objekts nicht mehr überein. Aufgrund der oben erwähnten Zuordnung der Speckle-ID zu der neu erzeugten ID in der Usprungssoftware kann das Speckle-Objekt, wenn es erneut vom Connector empfangen wird, jedoch trotzdem das neu erzeugte Objekt in der AEC-Software verändern.

3.6 Revit-Connector

Der konkrete Aufbau und die Funktionsweise der Connectors auf Quellcodeebene sind nicht Gegenstand dieser Arbeit. Dennoch werden im Folgenden einige grundlegende Konzepte erläutert, die für die Verwendung des Revit-Connectors relevant sind. Der Revit-Connector ist ein Plug-in für die BIM-Autorensoftware Revit, das den Datenaustausch zwischen Revit und dem Speckle-Server ermöglicht. Dieses Plug-in bietet eine Benutzeroberfläche in Revit, mit der Anwender steuern können, welche Daten an den Server gesendet bzw. von diesem empfangen werden sollen. Mit dem Revit-Connector lassen sich also Commits erstellen und auch wieder empfangen.

Beim Senden erstellt der Revit-Connector für alle im Commit inkludierten Bauteilelemente entsprechende Speckle-Objekte und verschachtelt diese in Collections entsprechend ihrer Bauteilkategorien in Revit (vgl. Abb. 3.5). Alle diese erstellten Speckle-Objekte besitzen u. a. das Attribut *displayValue*. Dieses Attribut wird dafür benutzt, die entsprechenden Objekte in Softwareprodukten geometrisch darzustellen, die das Objekt nicht nativ darstellen können. Es beinhaltet typischerweise eine explizite Geometriedarstellung, genauer gesagt ein Mesh basierend auf der triangulierten Oberflächenbeschreibung (vgl. Abschnitt 2.2.1). Beispielsweise können CAD-Autorensysteme wie AutoCAD BIM-Objekte wie Wände oder Türen nicht von sich aus darstellen. Hier dient *displayValue* als Rückfallgeometrie. Auch

der 3D-Viewer von Speckles webbasierter Schnittstelle nutzt für die Visualisierung der Objekte die in *displayValue* enthaltene Geometrie.

Darüber hinaus lassen sich mittels der Benutzeroberfläche verschiedene Optionen für das Senden von Objekten auswählen. Beispielsweise werden mit der Senden-Option „Everything“ alle im Revit-Modell enthaltenen Objekte in den zu sendenden Commit inkludiert. Mit der Option „Selection“ lassen sich hingegen einzelne Objekte an den Server senden. Daneben existieren noch andere Optionen, die die Auswahl der zu sendenden Objekte nach Bauteilkategorien, Ansichten oder sonstigen Kriterien ermöglichen.

Auch beim Empfangen von Objekten vom Speckle-Server lassen sich mittels der Benutzeroberfläche verschiedene Optionen auswählen. Diese Optionen lauten „Update“, „Create“ und „Ignore“. Bei allen drei Optionen sollen empfangene Speckle-Objekte, die noch kein Äquivalent in dem aktuellen Revit-Modell besitzen, als neue Revit-Objekte ins Modell eingefügt werden. Inwieweit das tatsächlich möglich ist, ist Gegenstand der Versuche in Abschnitt 4.3.4. Die Empfangsoptionen unterscheiden sich jedoch, sobald ein empfangenes Speckle-Objekt bereits ein Äquivalent im aktuellen Revit-Modell besitzt. Wird die Option „Ignore“ gewählt, wird das aktuell im Revit-Modell existierende Objekt nicht verändert und das empfangende Speckle-Objekt ignoriert. Auch bei der „Create“-Option bleibt das bereits existierende Revit-Objekt unverändert, jedoch wird ein weiteres Revit-Objekt basierend auf dem empfangenen Speckle-Objekt erzeugt. Wird „Update“ gewählt, so wird das existierende Objekt im Revit-Modell an das empfangene Speckle-Objekt angepasst. Die „Update“-Option ist die Standardoption des Revit-Connectors und wird auch im Rahmen der Versuche zum Revit-Connector in Kapitel 4 verwendet.

Der Abgleich eines empfangenen Speckle-Objekts mit einem bereits existierenden Objekt im Revit-Modell basiert auf dem in Abschnitt 3.5 angesprochenen Konzept von Transports und der Zuordnung der Speckle-ID zu der ID eines Revit-Objektes. Wie in Abschnitt 2.3.3 erläutert, besitzen Objekte in Revit mehrere IDs. Der Revit-Connector nutzt für die Erzeugung der *applicationId* eines Speckle-Objekts ausschließlich die *UniqueId* des entsprechenden Revit-Objekts. Um die empfangenen Inkremente, sprich die Modelländerungen, auf das Revit-Modell anzuwenden, führt der Revit-Connector also einen ID-basierten Abgleich von nativen Revit-Objekten und empfangenen Speckle-Objekten durch.

3.7 AutoCAD-Connector

In diesem Unterkapitel werden grundlegende Konzepte des AutoCAD-Connectors erläutert, die für dessen Verwendung von Relevanz sind. Dafür werden zunächst einige Grundzüge der Software AutoCAD vom Hersteller Autodesk in wenigen Worten zusammengefasst. Wie ihr Name bereits suggeriert, handelt es sich bei AutoCAD um eine CAD-Software und nicht um eine BIM-Software. Daher lassen sich mit AutoCAD nur geometrische 2D- und 3D-Objekte erzeugen, die keinerlei semantische Informationen beinhalten. Auch das in Abschnitt 2.3.3 angesprochene Konzept von „Host Objects“ und „Hosted Objects“ existiert somit nicht in AutoCAD. Die erzeugten Geometrien lassen sich sogenannten

Layern zuordnen, welche vom Anwender erstellt werden können und der Strukturierung der Zeichnung bzw. des 3D-Modells dienen. Elemente in AutoCAD können mithilfe der Layer logisch gruppiert und verwaltet werden. Beispielsweise können die Eigenschaften und die Sichtbarkeit aller Elemente einer Layer zentral über die Layer verändert werden.

3D-Geometrien in AutoCAD können grundsätzlich auf zwei verschiedene Arten dargestellt werden: als 3D-Volumenkörper oder als Vielflächennetze. 3D-Volumenobjekte sind durch geschlossene Oberflächen begrenzt und haben ein inneres Volumen. Sie können mittels impliziter Verfahren, wie beispielsweise der Extrusion oder der CSG-Methode, erstellt werden. Vielflächennetze haben hingegen kein inneres Volumen und können Löcher oder offene Bereiche enthalten. Weiterhin können Objekte in AutoCAD zu sogenannten Blöcken zusammengefasst werden. Bei der Definition eines Blockes können diesem eigene Attribute zugewiesen werden.

Der AutoCAD-Connector ist ein Plug-in für AutoCAD und ermöglicht den Datenaustausch zwischen AutoCAD und dem Speckle-Server. Wie der Revit-Connector bietet auch der AutoCAD-Connector eine Benutzeroberfläche, mit der der Anwender steuern kann, welche Objekte an den Speckle-Server gesendet bzw. von diesem empfangen werden sollen. Bei der Auswahl der Elemente aus der 2D-Zeichnung bzw. dem 3D-Modell, die in Form eines Commits an den Server gesendet werden sollen, bietet der Connector drei verschiedene Optionen: „Everything“, „Selection“ und „Layers“. Die beiden Erstgenannten unterscheiden sich in ihrer Funktion nicht von den gleichnamigen Optionen des Revit-Connectors. Mithilfe der Option „Layers“ können ausgewählte Layers an den Speckle-Server gesendet werden. Wie bereits in Abb. 3.6 dargestellt, erstellt der AutoCAD-Connector die in einem Commit-Objekt enthaltenen Speckle-Collections auf Basis der AutoCAD-Layer, der die gesendeten Objekte angehören. Beim Empfangen von Objekten bietet der AutoCAD-Connector nur zwei Optionen: „Create“ und „Update“. Diese sind äquivalent zu den entsprechenden Empfangsoptionen des Revit-Connectors.

Kapitel 4

Versuche

4.1 Ablauf und Gegenstand der Versuche

In diesem Kapitel werden Versuche mit den Speckle-Connectors für eine BIM-Autorensoftware (Revit) und eine CAD-Autorensoftware (AutoCAD) durchgeführt, um das Verhalten des jeweiligen Connectors bei der Übermittlung von unterschiedlichen Inkrementen zu beobachten. Dabei werden an beide Connectors jeweils die folgenden vier Anforderungen gestellt: das Modifizieren, Verschieben, Einfügen und Entfernen von Objekten in einem BIM-Modell bzw. einem CAD-Volumenmodell beim Empfangen eines Commits vom Speckle-Server. Unter Modifizieren wird dabei das Ändern der Objektabmessungen verstanden. Sowohl in der BIM-Software als auch in der CAD-Software wird ein Modell erstellt und ausgewählte Modellbestandteile in verschiedenen Kombinationen mithilfe des entsprechenden Speckle-Connectors in Form von Commits an den Speckle-Server gesendet. Daraufhin werden diese Commits mittels der Connectors wieder in der BIM- bzw. der CAD-Software empfangen und die Auswirkungen der einzelnen Commits auf das jeweilige Modell beobachtet. Doch auch das Verhalten der beiden Connectors beim Empfangen von Objekten, die vom jeweils anderen Connector erstellt worden sind, wird beobachtet. Anschließend werden die Speckle-Objekte, welche von den Connectors innerhalb der Versuche generiert worden sind, hinsichtlich ihres Aufbaus und ihrer Unterschiede zueinander untersucht.

Gegenstand der Versuche sind somit nicht der Aufbau und die Funktionsweise der Connectors auf Quellcodeebene, sondern deren Verhalten und die von ihnen generierten Speckle-Objekte. Wie die Connectors auf der Quellcodeebene auf Transports zugreifen, Speckle-Objekte erstellen und mit der jeweiligen AEC-Software interagieren, wird nicht näher betrachtet.

4.2 Extraktion und Datenfluss der serialisierten Speckle-Objekte

Der Aufbau der Speckle-Objekte und ihre Unterschiede zueinander werden anhand der JSON-Serialisierungen der Objekte analysiert. Es muss jedoch zunächst ein Weg gefunden werden, um auf diese Serialisierungen zuzugreifen. Zum aktuellen Stand bietet die webbasierte Schnittstelle von Speckle nämlich keine Möglichkeit, die rohen JSON-Repräsentationen der auf dem Server gespeicherten Speckle-Objekte einzusehen. Deswegen wird dafür das von Speckle zur Verfügung gestellte Python-SDK verwendet. Aufbauend auf diesem SDK wird ein Python-Code geschrieben, mit dem sich die serialisierten Speckle-Objekte extrahieren lassen. Der Code findet sich im digitalen Anhang dieser Arbeit.

Es gibt zwei Möglichkeiten, um ein Speckle-Objekt in seiner serialisierten Form zu extrahieren. Die erste Möglichkeit besteht darin, das JSON-Objekt direkt über seine Speckle-ID zu extrahieren. Die Alternative dazu ist die indirekte Extraktion des JSON-Objektes über den Speckle-Commit, in dem es enthalten ist. Dabei wird zuerst der Commit mittels seiner eigenen ID und der ID des übergeordneten Streams vom Server empfangen. Danach werden alle im Commit enthaltenen Speckle-Objekte in ihrer serialisierten Form extrahiert. Schließlich wird aus diesen Speckle-Objekten das Gesuchte mithilfe seiner ID herausgefiltert. Das Resultat beider Varianten ist der exakt gleiche JSON-String. Dieser ist jedoch noch unformatiert, d. h. er besteht aus einer einzigen langen Zeichenkette. Um serialisierte Speckle-Objekte im Rahmen dieser Arbeit übersichtlich darstellen zu können, werden diese mithilfe der Website »FreeFormatter.com« (2024) formatiert. Um die einzelnen JSON-Serialisierungen miteinander zu vergleichen, wird die Website »jsondiff.com« (2023) verwendet, welche semantische Unterschiede zwischen zwei ausgewählten JSON-Objekten hervorhebt.

Alle betrachteten Speckle-Objekte haben ihren Ursprung in Objekten, die in einer Autorensoftware aus dem AEC-Bereich erstellt worden sind. Die entsprechenden Connectors serialisieren die Speckle-Objekte und senden sie an den Speckle-Server. Es gibt aber nun zwei verschiedene Wege, auf denen Speckle-Objekte im Rahmen dieser Arbeit vom Speckle-Server empfangen werden. Auf dem Ersten werden die Speckle-Objekte von einem Speckle-Connector empfangen und in das native Datenformat der entsprechenden BIM- oder CAD-Software konvertiert. Der zweite Weg besteht aus dem Empfangen und der darauffolgenden Extraktion der Speckle-Objekte in ihrem JSON-Format mithilfe des geschriebenen Python-Codes.

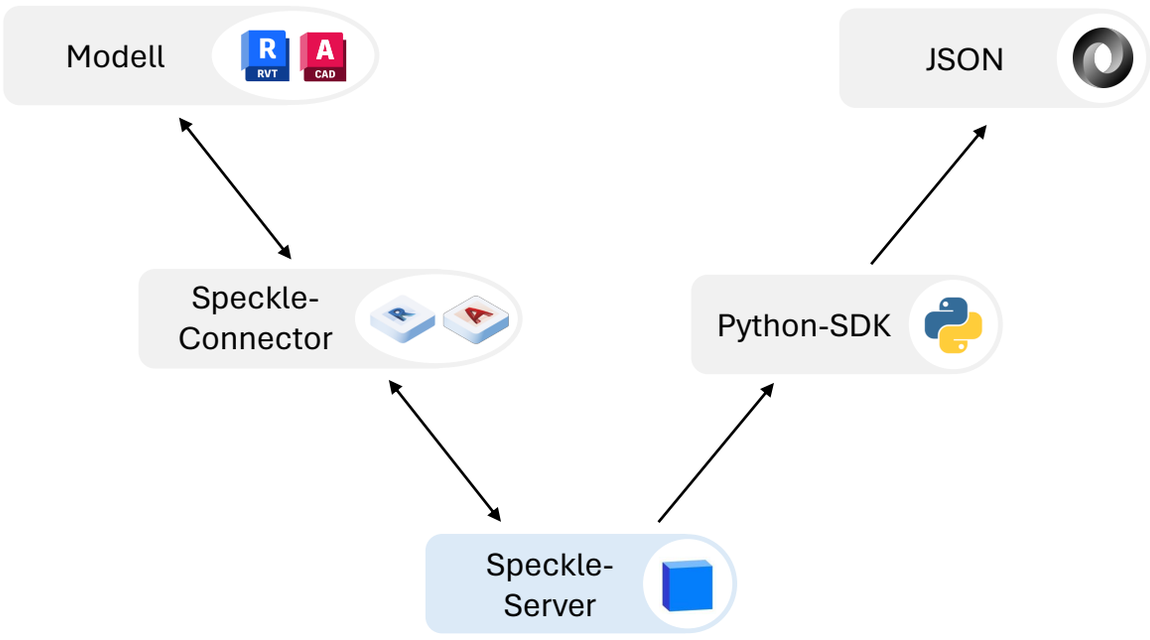


Abbildung 4.1: Datenfluss der Speckle-Objekte im Rahmen dieser Arbeit

4.3 Inkrementelle Updates innerhalb von Revit

4.3.1 Vorbereitung der Versuche

In den nachfolgenden Versuchen soll untersucht werden, inwiefern Speckle-Commits dafür genutzt werden können, bestehende Revit-Modelle zu verändern. Bei den Veränderungen handelt es sich um das Modifizieren, Verschieben, Einfügen und Löschen von Host Objects (Wänden) sowie von Hosted Objects (Türen). Im Rahmen dieser Arbeit werden Wände auch als Host Objects bezeichnet, selbst wenn diese keine Hosted Objects beherbergen. Es werden die in Tabelle 4.1 gelisteten Programmversionen genutzt.

Tabelle 4.1: Übersicht über die benutzten Programmversionen für die Versuche innerhalb von Revit

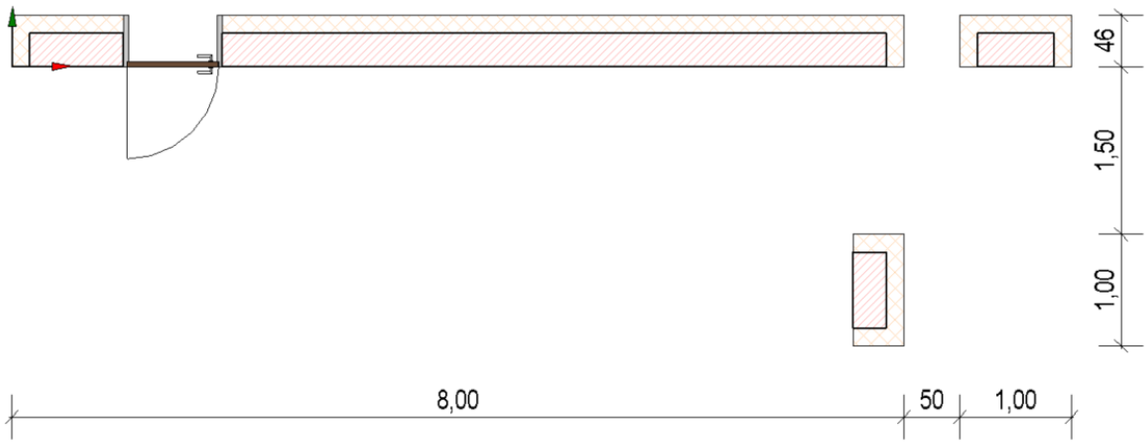
Programm	**Version**
Revit	2024.1
RevitDBExplorer	2024 - v2.2.2
Speckle-Connector für Revit	2.17.0
Speckle	2.17.13

Die Versuche wurden sowohl innerhalb einer Revit-Datei als auch zwischen zwei verschiedenen Revit-Dateien durchgeführt. Im Falle der zwei unterschiedlichen Revit-Dateien wurde die erste genutzt für die Erstellung des ursprünglichen Commits, der das Ausgangsmodell enthielt. Dieser Commit wurde in der zweiten Datei empfangen, in der daraufhin die entsprechenden Änderungen am Modell vorgenommen und dann als Commit auf den Server geladen wurden. Die Änderungen wurden anschließend in der ersten Datei empfangen. Das Vorgehen mit zwei Revit-Dateien sollte das Szenario mehrerer am Projekt beteiligter Akteure simulieren. Die Ergebnisse entsprachen jedoch exakt denen aus der Variante mit nur einer Datei, weshalb im Folgenden der Übersichtlichkeit wegen die Versuche anhand von nur einer Revit-Datei dargestellt werden.

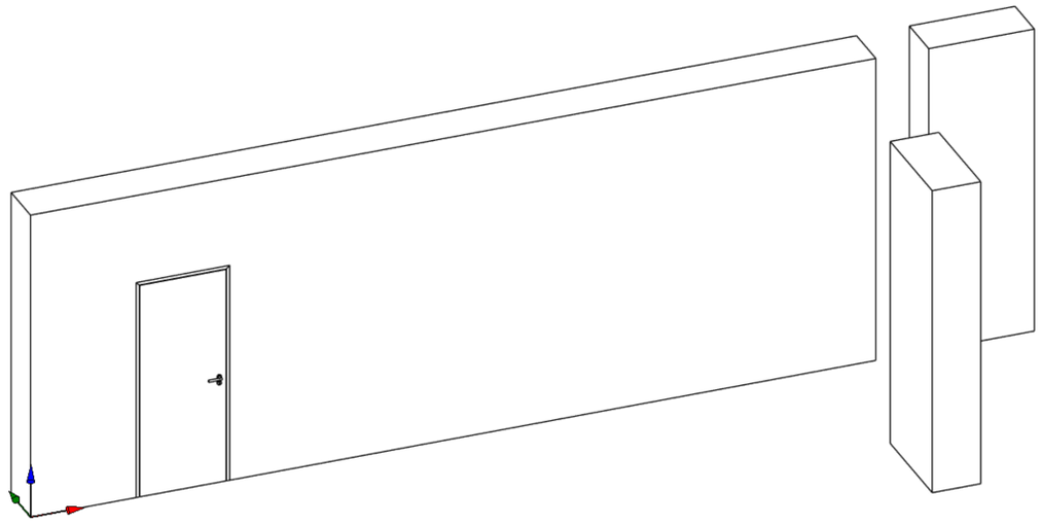
Ausgangspunkt der jeweiligen Versuchen ist immer dasselbe in Revit erstellte BIM-Modell, welches auf Grundlage der revitinternen Vorlagedatei „BIM Architektur und Ingenieurbau (vereinfacht)“ erstellt worden ist. In dieser Vorlage sind bereits eine Reihe voreingestellter Bauteilfamilien enthalten, die häufig in Architektur- und Ingenieurprojekten verwendet werden, wie beispielsweise Wände, Türen, Fenster, Träger usw. Als Längenmaßeinheit für das verwendete Modell wird der Meter eingestellt.

Das Revit-Modell besteht aus drei nichttragenden Wänden der Art „Basiswand : Ziegel+WD hart 300+160 : R6“. Sie alle haben eine Höhe von drei Metern und eine Breite von $0,46$ Metern, die sich aus $0,3$ Metern Mauerwerk und $0,16$ Metern Wärmedämmung zusammensetzt. Eine der Wände ist acht Meter lang und wird im weiteren Verlauf als Hauptwand bezeichnet. Die restlichen zwei Wände besitzen jeweils eine Länge von nur

einem Meter und werden nachfolgend als Hilfswände bezeichnet. In die Hauptwand ist mit einem Abstand von einem Meter zum linken Wandende eine Tür eingefügt worden. Bei der Tür handelt es sich um eine „TU DF 1 - Rahmenstock flächenbündig : ML“ mit einer Breite von $0,885$ Metern und einer Höhe von $2,135$ Metern. Die Positionen der Wände zueinander und in Bezug auf den internen Ursprung sind in Abb. 4.2 dargestellt.



(a) Grundriss



(b) 3D-Ansicht

Abbildung 4.2: Revit-Ausgangsmodell

Um dem Leser die Nachvollziehbarkeit der im Rahmen der Versuche durchgeführten Modelländerungen zu erleichtern, werden in den folgenden Abbildungen Farben benutzt. Die Bedeutungen der jeweiligen Farben sind in Abb. 4.3 beschrieben.

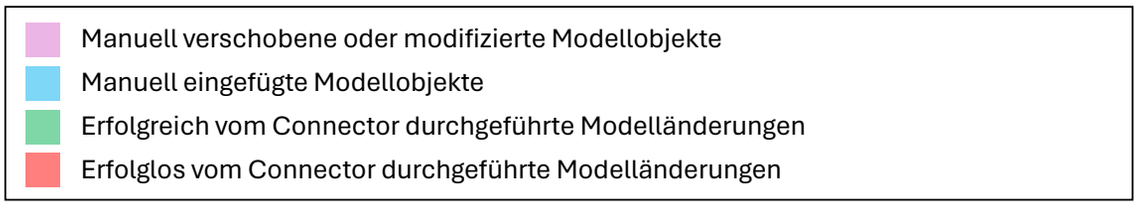


Abbildung 4.3: Farblegende

Auf dem Speckle Server wird ein neuer Stream erstellt. Innerhalb dieses Streams werden die in Tabelle 4.2 aufgelisteten Commits auf den Speckle-Server geladen.

Tabelle 4.2: Übersicht über alle vom Revit-Connector erstellten Commits

enthaltene Objekte	**Bezeichnung**
Hauptwand, Tür	(Hauptwand, Tür)-Commit
Hauptwand ohne Tür darin	(Hauptwand)-Commit
Tür	(Tür)-Commit
Hauptwand, Hilfswand in y-Richtung	(Hauptwand, Hilfswand)-Commit
Hilfswand in y-Richtung	(Hilfswand)-Commit
beide Hilfswände	(Hilfswände)-Commit
auf vier Meter verkürzte Hauptwand	(kurze Wand)-Commit

4.3.2 Versuch 1: Objekte modifizieren

Modifizieren von Host Objects

Zunächst wird der Übersicht halber die Tür aus dem Ausgangsmodell entfernt. Wie in Tabelle 4.2 bereits aufgeführt, wird die Hauptwand auf vier Meter verkürzt und in einem Commit auf den Speckle-Server hochgeladen. Anschließend wird die Hauptwand manuell wieder auf acht Meter verlängert und der Commit in das Revit-Modell empfangen. Das Ergebnis ist in Abb. 4.4 dargestellt. Die Wand konnte mithilfe des Commits erfolgreich auf 4 Meter verkürzt werden.

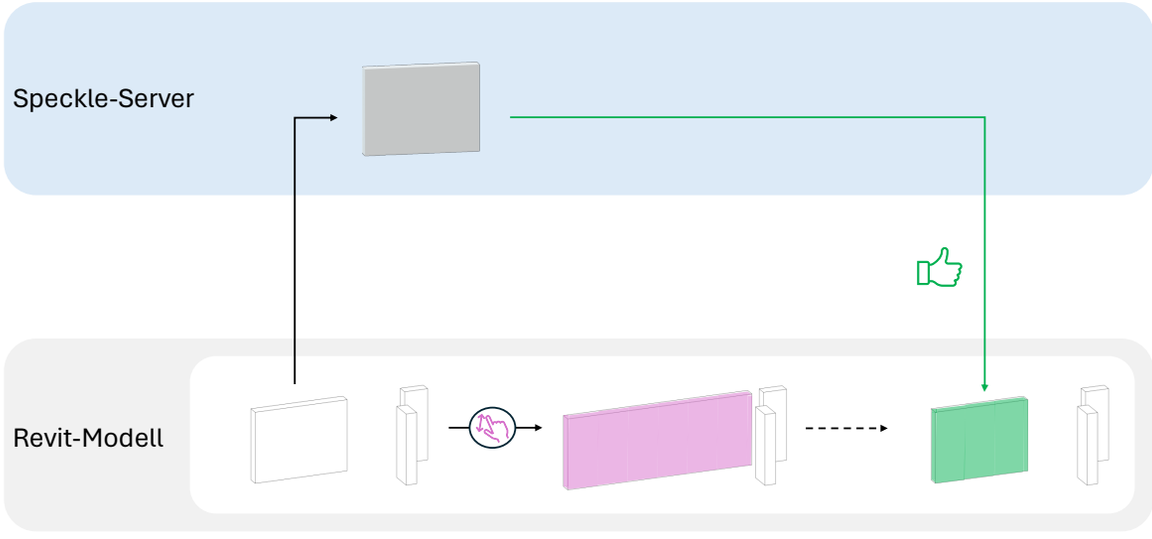


Abbildung 4.4: Modifizieren der Hauptwand durch Empfangen des (kurze Wand)-Commits

Modifizieren von Hosted Objects

Nun soll getestet werden, ob auch das Modifizieren von Hosted Objects mittels des Revit-Connectors möglich ist. Dafür wird die Tür manuell auf eine Breite von zwei Metern vergrößert und sowohl die x- als auch die y-Hilfswand aus Gründen der Übersichtlichkeit aus dem Revit-Modell entfernt. Als Erstes wird der (Tür)-Commit empfangen. Wie in Abb. 4.5 zu sehen, wird die Tür dadurch nicht auf ihre ursprüngliche Breite zurückgesetzt. Als Nächstes wird der (Hauptwand, Tür)-Commit empfangen. Auch hier bleibt das Modifizieren der Tür mittels des Revit-Connectors erfolglos (vgl. Abb. 4.6).

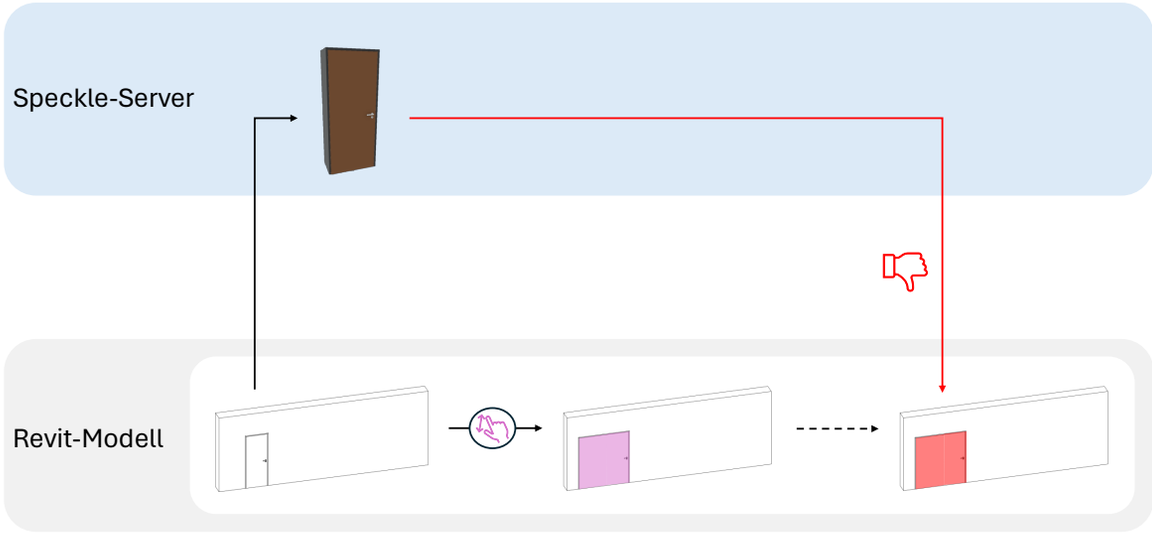


Abbildung 4.5: Modifizieren der Tür durch Empfangen des (Tür)-Commits

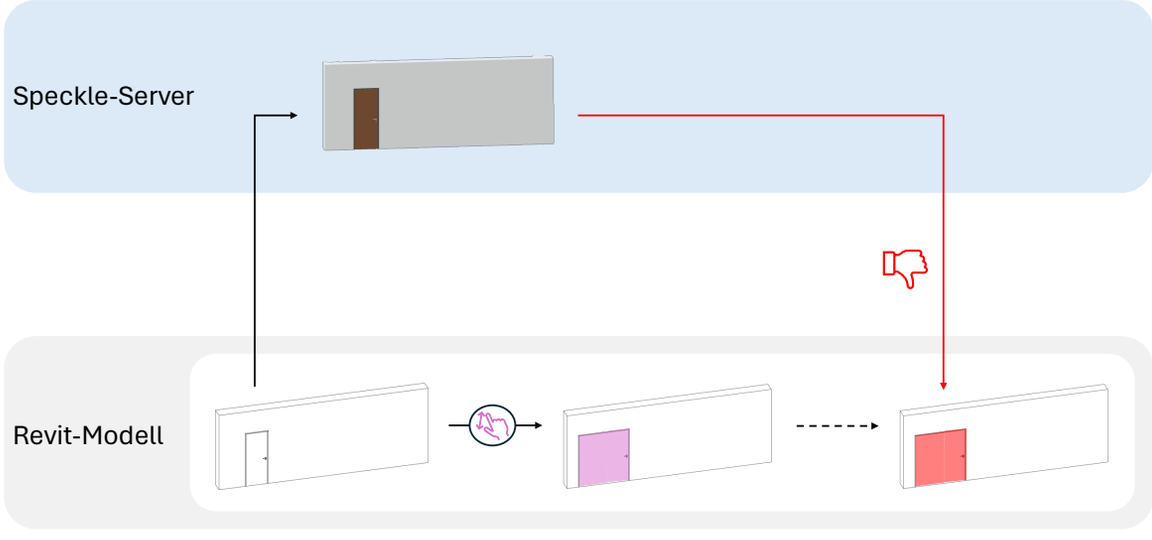


Abbildung 4.6: Modifizieren der Tür durch Empfangen des (Hauptwand, Tür)-Commits

4.3.3 Versuch 2: Objekte verschieben

Verschieben von Hosted Objects

Das Hosted Object Tür wird innerhalb des Host Objects Hauptwand um fünf Meter in positive x-Richtung verschoben. Die beiden Hilfswände werden nicht benötigt und somit der Übersicht halber aus dem Ausgangsmodell entfernt. Wie in Abb. 4.7 und Abb. 4.8 zu sehen, können anschließend sowohl der (Tür)-Commit als auch der (Hauptwand, Tür)-Commit dafür benutzt werden, die Tür an ihre ursprüngliche Position in der Hauptwand zurückzuverschieben.

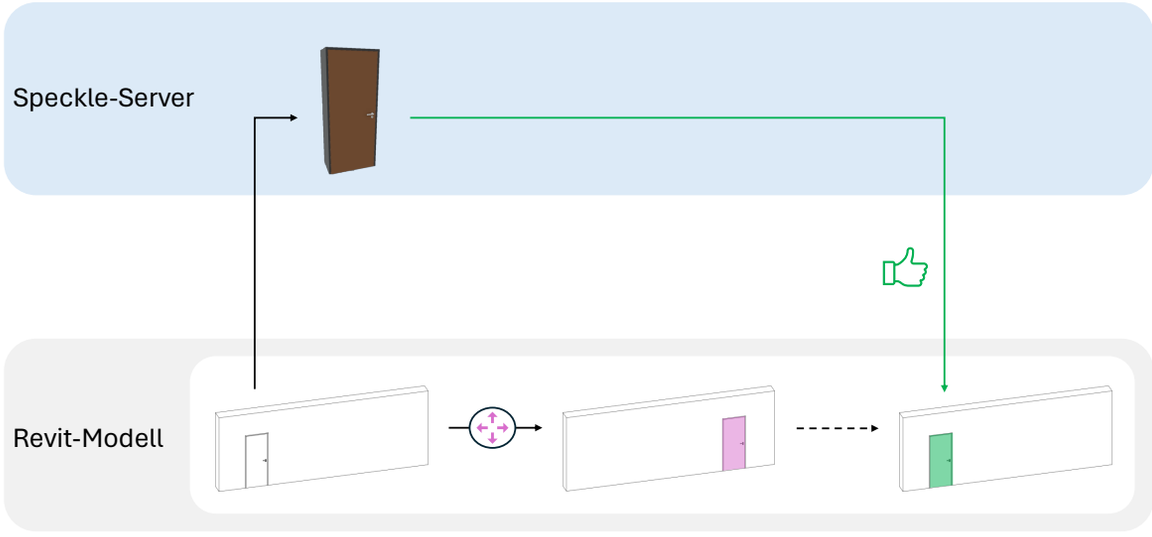


Abbildung 4.7: Verschieben der Tür durch Empfangen des (Tür)-Commits

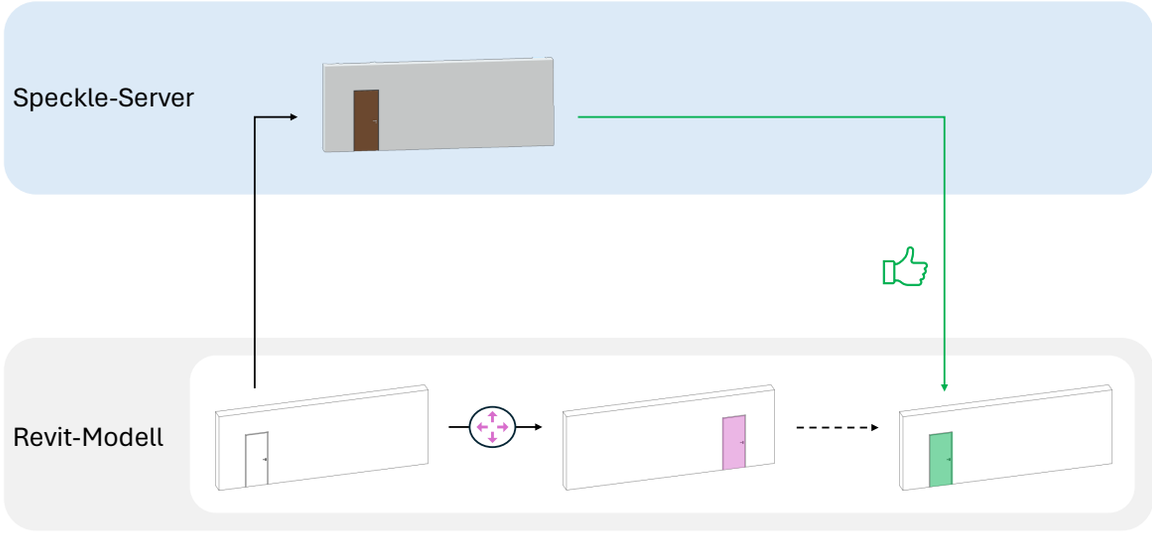


Abbildung 4.8: Verschieben der Tür durch Empfangen des (Tür, Wand)-Commits

Verschieben von Host Objects

Um das Verschieben von Host Objects innerhalb von Revit mithilfe von Speckle-Commits zu untersuchen, wird nun die Hilfswand in y-Richtung betrachtet. Dazu werden die Tür und die Hilfswand in x-Richtung aus dem Ausgangsmodell entfernt und die y-Hilfswand um drei Meter in negative y-Richtung verschoben. Daraufhin wird der (Hilfswand)-Commit empfangen, was dazu führt, dass die y-Hilfswand wieder an ihre ursprüngliche Stelle im Revit-Modell zurückverschoben wird (vgl. Abb. 4.9).

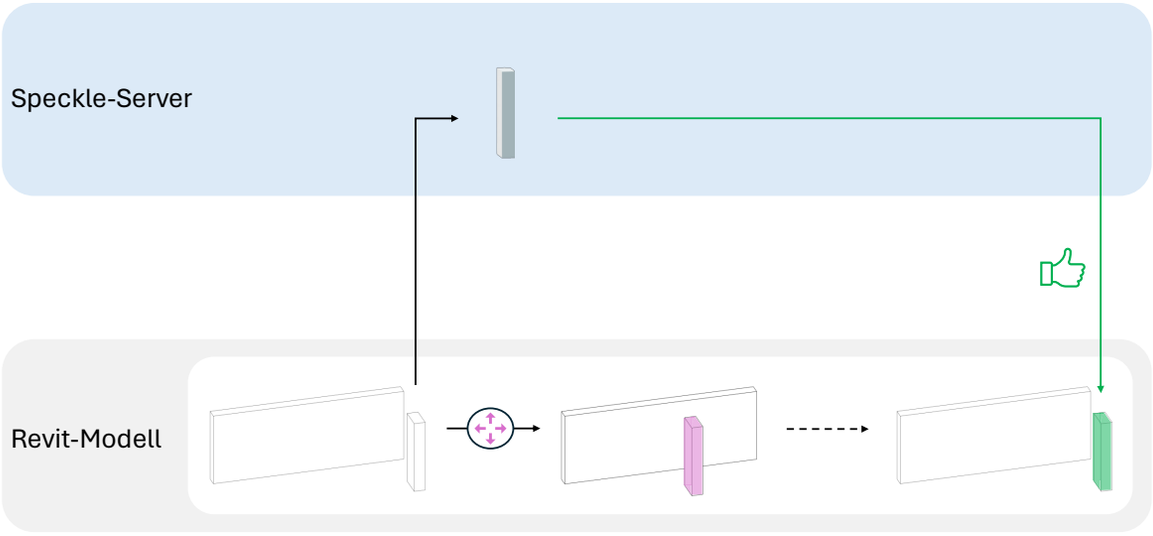


Abbildung 4.9: Verschieben der y-Hilfswand durch Empfangen des (Hilfswand)-Commits

4.3.4 Versuch 3: Objekte einfügen

Einfügen von Hosted Objects

Die beiden Hilfswände werden hier nicht benötigt und somit aus dem Ausgangsmodell entfernt. Im Anschluss wird auch die Tür aus dem Modell entfernt und daraufhin der (Tür)-Commit empfangen. Im Zuge dessen wird die Tür jedoch nicht in das Revit-Modell eingefügt, die Wand bleibt leer (vgl. Abb. 4.10). Daraufhin wird der (Hauptwand, Tür)-Commit empfangen. Wie in Abb. 4.11 zu sehen, ergibt sich dadurch ein anderes Resultat: Die Tür wird erfolgreich in das Revit-Modell eingefügt. Diese Versuche wurden sowohl mit der „Update“- als auch mit der „Create“-Option des Revit-Connectors durchgeführt, führten jedoch beide Male zum gleichen Ergebnis in Bezug auf das Einfügen der Tür. Mit der „Create“-Option wurde jedoch beim Empfangen des (Hauptwand, Tür)-Commits eine zusätzliche Hauptwand in das Modell eingefügt, die die alte überlagerte.

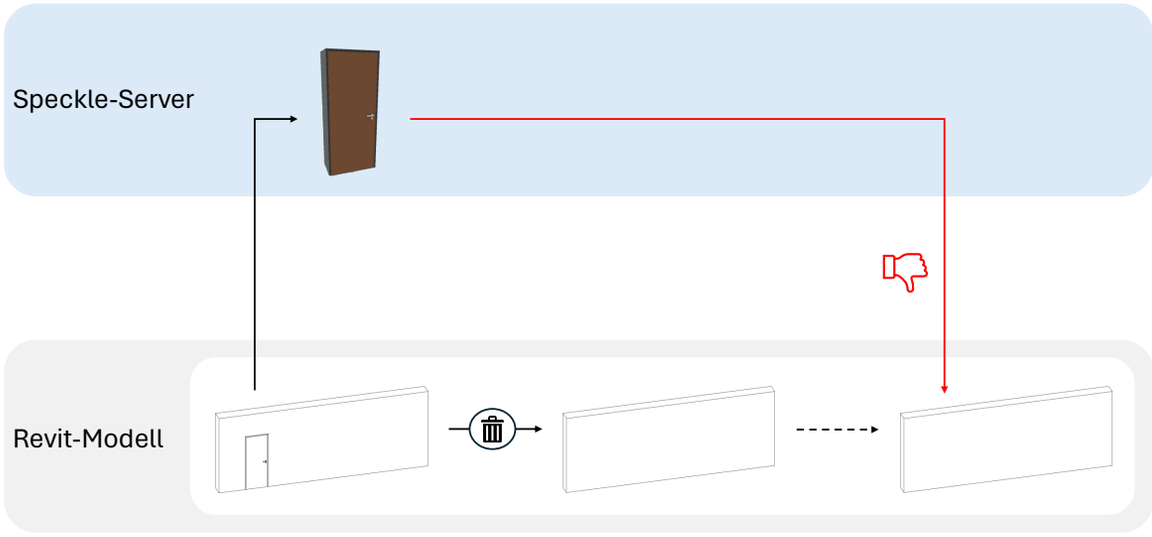


Abbildung 4.10: Einfügen der Tür durch Empfangen des (Tür)-Commits

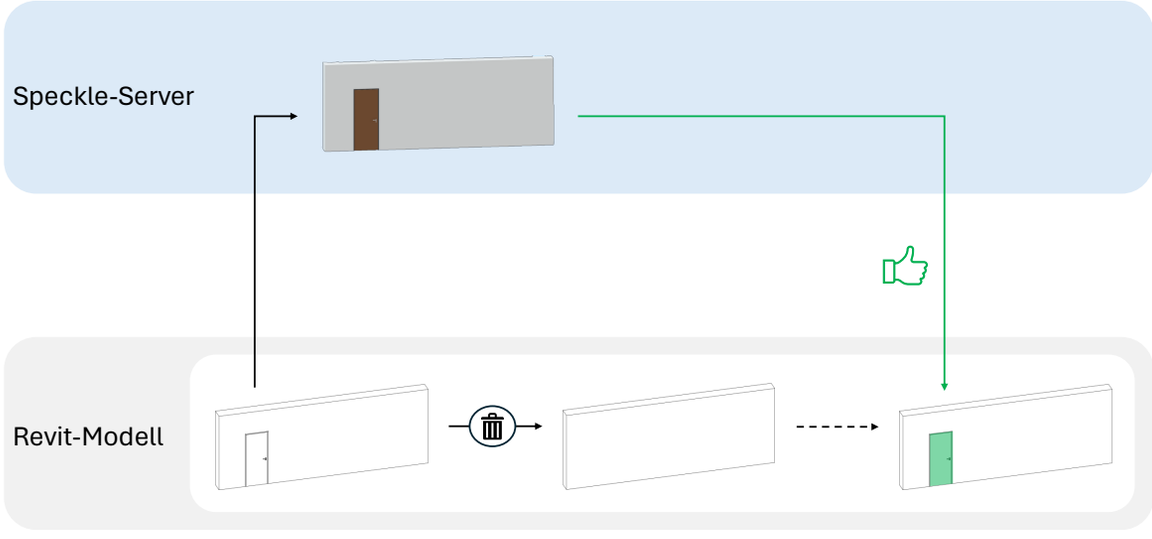


Abbildung 4.11: Einfügen der Tür mithilfe des (Tür, Wand)-Commits

Einfügen von Host Objects

Nun soll die Fähigkeit des Revit-Connectors, Host Objects in das Revit-Modell einzufügen, untersucht werden. Wieder wird sowohl die Tür als auch die x-Hilfswand aus Gründen der Übersichtlichkeit aus dem Ausgangsmodell entfernt. Danach wird auch die Hilfswand in y-Richtung aus dem Modell entfernt und anschließend der (Hilfswand)-Commit vom Speckle-Server empfangen. Das Empfangen des Commits hat zur Folge, dass die Wand wieder in das Modell eingefügt wird (vgl. Abb. 4.12).

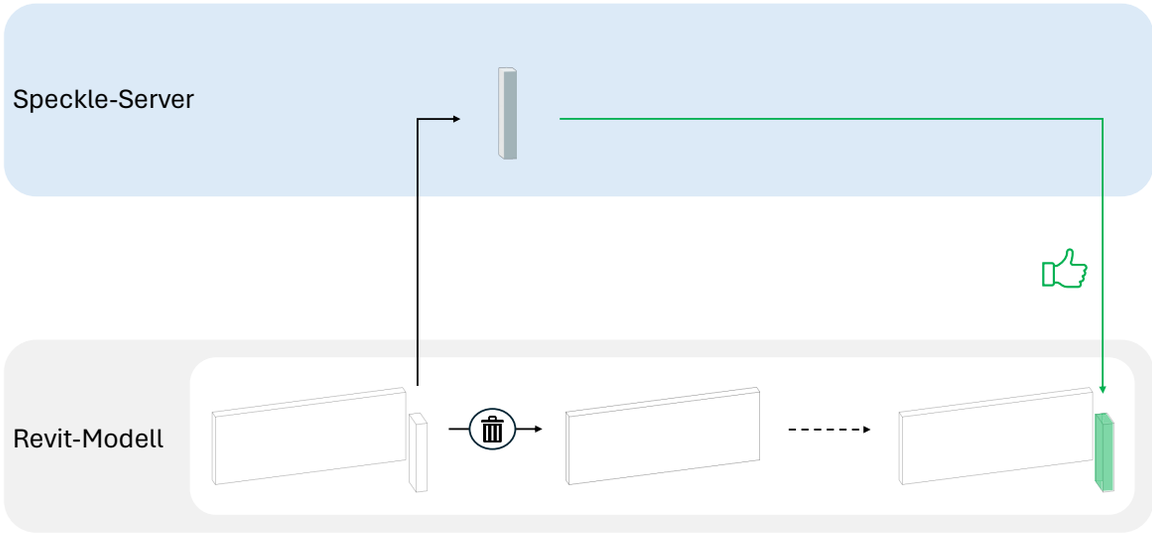


Abbildung 4.12: Einfügen der y-Hilfswand durch Empfangen des (Hilfswand)-Commits

4.3.5 Versuch 4: Objekte entfernen

Im letzten Versuch zum Revit-Connector soll dessen Fähigkeit getestet werden, Host Objects und Hosted Objects durch das Empfangen von Commits aus einem Revit-Modell zu entfernen. Damit ein Objekt mittels eines Commits entfernt werden kann, muss dieses zunächst in das Revit-Modell eingefügt worden sein. Für das Einfügen stehen nach dem letzten Versuch nun zwei Möglichkeiten zur Verfügung:

1. Manuelles Einfügen des neuen Objektes im Revit-Modell
2. Einfügen des neuen Objektes durch Empfangen eines Speckle-Commits, der das Objekt und im Falle eines Hosted Objects auch dessen Host Object enthält

„Neu“ bedeutet hier nicht, dass das eingefügte Objekt äußerliche Unterschiede zu dem bisher im Ausgangsmodell verwendeten Objekt aufweist oder an einer anderen Position in das Modell eingefügt wird. Die neu eingefügten Objekte sind von der gleichen Art und Form wie die bisher verwendeten. Jedoch besitzen diese neuen Objekte in Revit andere *UniqueIds* als die alten Objekte. Die zweite Möglichkeit kann auf das Szenario eines weiteren Projektbeteiligten übertragen werden, der sein veraltetes lokales Revit-Modell (z.B. die Hauptwand) auf den aktuellen Stand bringen will und daher das neueste Inkrement (z.B. eine Hilfswand) vom Speckle-Server empfängt.

Entfernen von Hosted Objects

Wie bereits in Abschnitt 3.3 erwähnt, kann ein Commit in Speckle keinen „Löschen-Befehl“ enthalten. Um eine Tür zu löschen, muss daher ein Umweg über deren Host-Wand genommen werden. Die Tür und die beiden Hilfswände werden manuell aus dem Ausgangsmodell entfernt. Die nun „leere“ Hauptwand wird daraufhin selektiert und in Form

des (Hauptwand)-Commits auf den Speckle-Server geladen. Daraufhin wird eine neue Tür manuell in das Revit-Modell eingefügt und anschließend der (Hauptwand)-Commit vom Speckle-Server empfangen. Es ist festzustellen, dass mit Empfangen dieses Commits die Tür nicht aus dem Modell entfernt wird (vgl. Abb. 4.13). Daraufhin wird die Tür nicht manuell in das Revit-Modell eingefügt, sondern indem der (Hauptwand, Tür)-Commit empfangen wird. Anschließend wird wieder der (Hauptwand)-Commit empfangen. Wie in Abb. 4.14 zu sehen, führt auch hier das Empfangen des (Hauptwand)-Commits nicht dazu, dass das Tür-Objekt aus dem Revit-Modell entfernt wird.

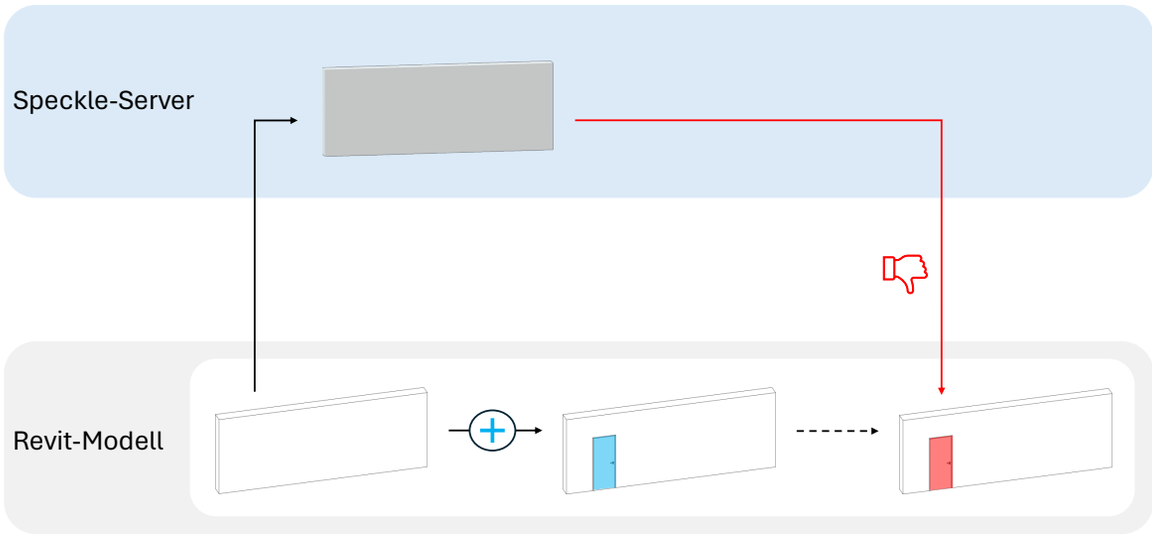


Abbildung 4.13: Manuelles Einfügen der Tür und anschließendes Empfangen des (Hauptwand)-Commits

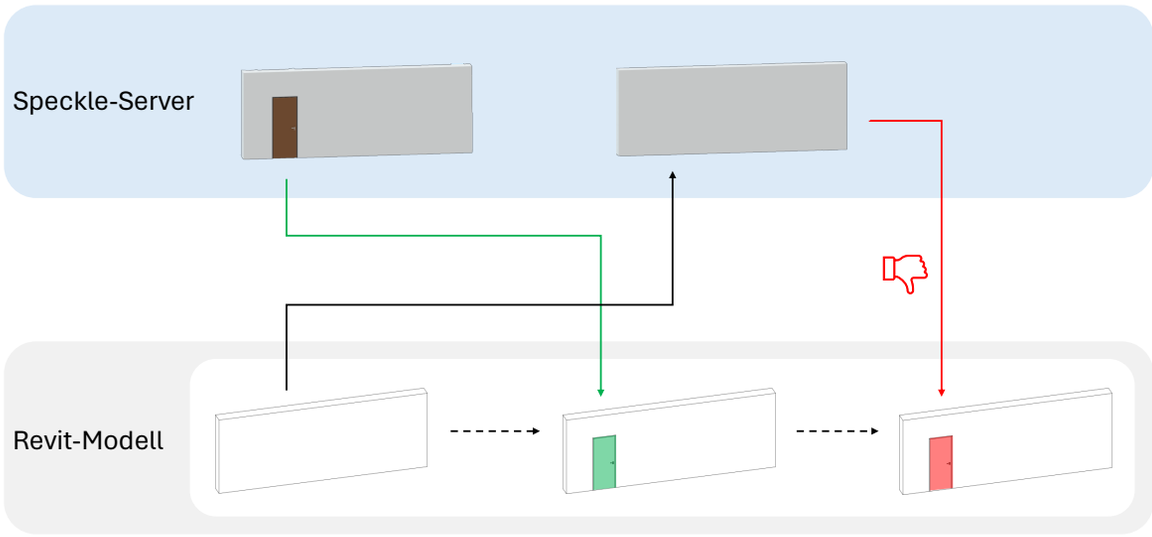


Abbildung 4.14: Einfügen der Tür per Commit und anschließendes Empfangen des (Hauptwand)-Commits

Durch Empfangen der leeren Hauptwand allein kann also weder eine manuell eingefügte Tür noch eine vom Speckle-Server empfangene Tür aus dem Revit-Modell entfernt werden. Infolgedessen wird ein anderer Commit gewählt, mit dem die Tür aus dem Modell entfernt

werden soll. Genauer gesagt wird anstatt dem (Hauptwand)-Commit jetzt der (Hauptwand, Hilfswand)-Commit empfangen. Wie in Abb. 4.15 zu sehen, kann eine manuell in das Revit-Modell eingefügte Tür auch durch diesen Commit nicht aus dem Modell entfernt werden. Wird die Tür jedoch durch das Empfangen eines Commits eingefügt und anschließend der (Hauptwand, Hilfswand)-Commit empfangen, wird die Tür erfolgreich aus dem Modell entfernt (vgl. Abb. 4.16).

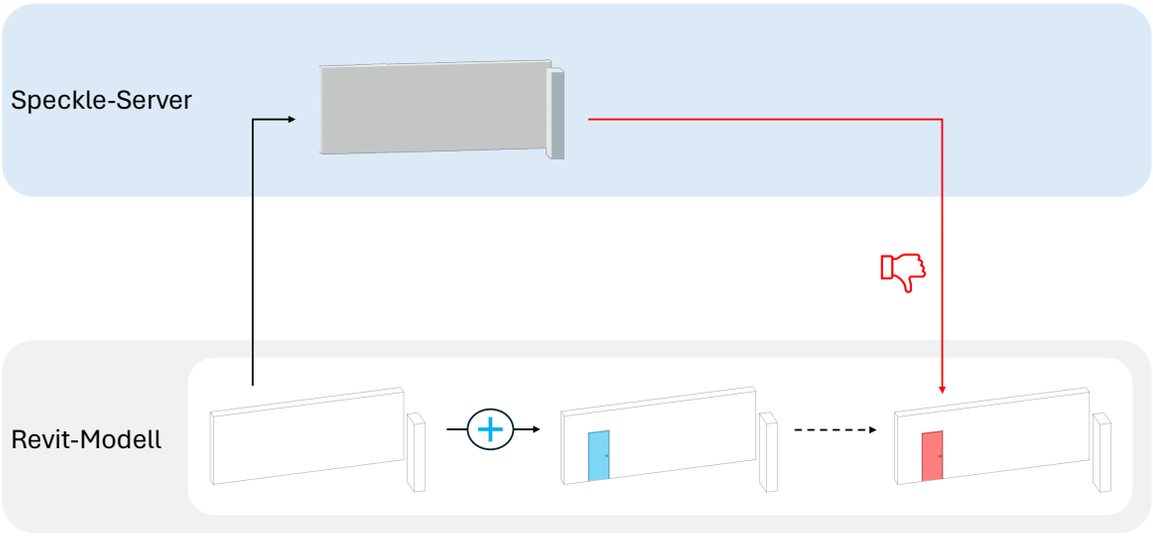


Abbildung 4.15: Manuelles Einfügen der Tür und anschließendes Empfangen des (Hauptwand, Hilfswand)-Commits

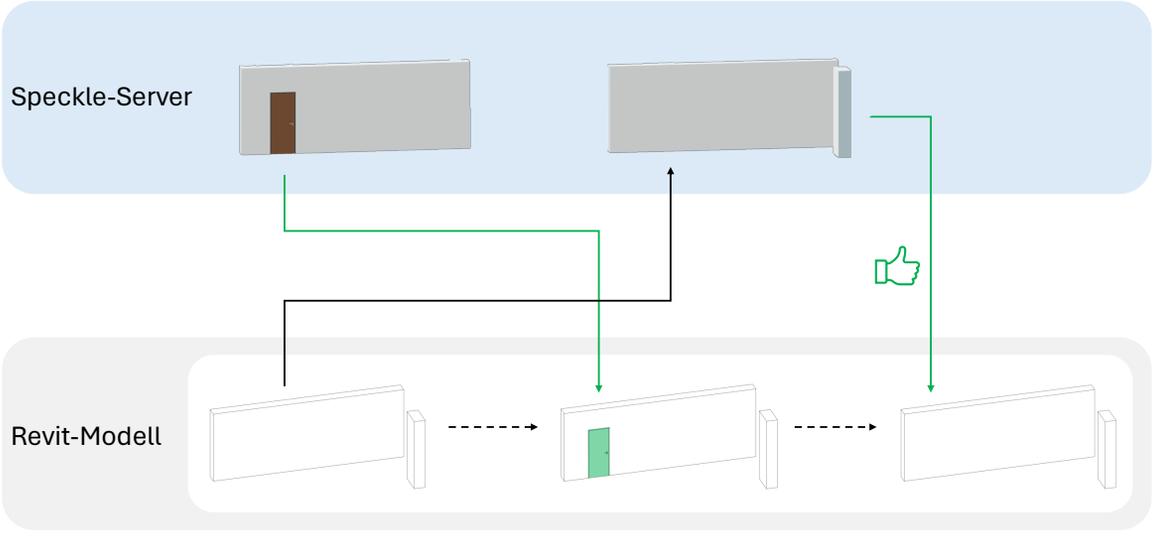


Abbildung 4.16: Einfügen der Tür per Commit und anschließendes Empfangen des (Hauptwand, Hilfswand)-Commits

Interessant ist weiterhin, was passiert, nachdem der (Hauptwand)-Commit empfangen und somit die Tür nicht aus dem Modell entfernt worden ist. Ab diesem Zeitpunkt kann die Tür nicht mehr per Empfangen eines Commits aus dem Revit-Modell entfernt werden, selbst wenn dieser neben der Host-Wand noch weitere Objekte enthält (vgl. Abb. 4.17). Ein Entfernen der Tür ist nur noch manuell möglich.

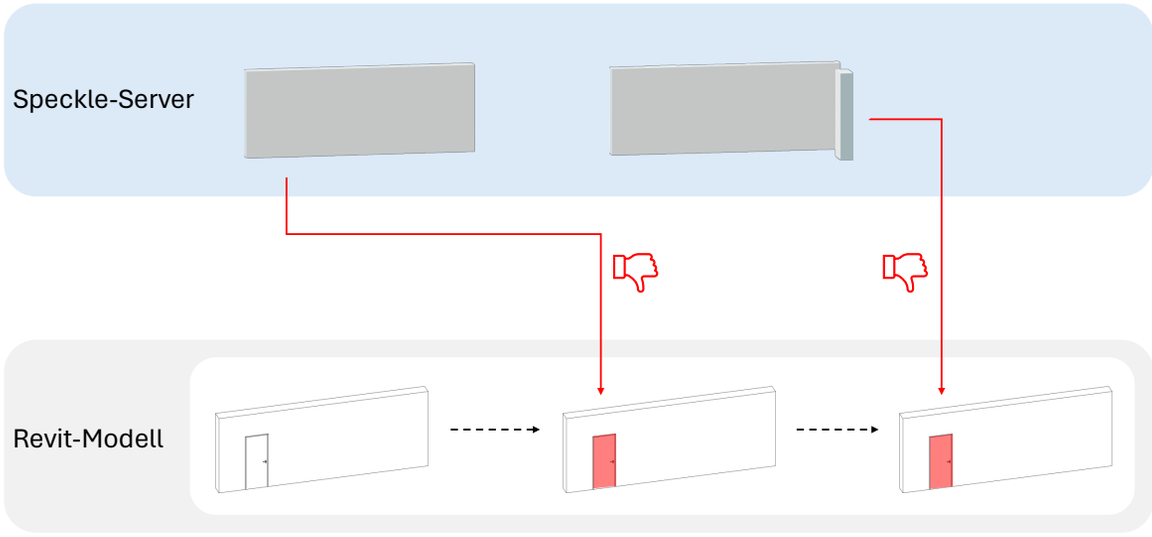


Abbildung 4.17: Verhalten nach Empfangen des (Hauptwand)-Commits

Entfernen von Host Objects

Auch für das Entfernen von Host Objects existiert kein "Löschen-Befehl", der in einem Commit an den Speckle-Server geschickt werden könnte. Welche Objekte ein Commit enthalten muss, um ein Host Object aus einem Revit-Modell zu entfernen, und unter welchen Bedingungen dies möglich ist, soll nun untersucht werden.

Aus Gründen der Übersichtlichkeit wird die Tür aus dem Ausgangsmodell entfernt. Zuerst wird versucht, die manuell ins Revit-Modell eingefügte Hauptwand aus dem Modell zu entfernen. Sowohl das Empfangen einer einzelnen Hilfswand als auch das Empfangen beider Hilfswände führt nicht dazu, dass die Hauptwand aus dem Modell entfernt wird (vgl. Abb. 4.18). Danach wird die Hauptwand manuell aus dem Modell entfernt und mittels Empfangen des (Hauptwand)-Commits erneut in die Revit-Datei eingefügt. Eine auf diese Weise eingefügte Hauptwand kann durch Empfangen des (Hilfswand)-Commits immer noch nicht aus dem Modell entfernt werden. Wird jedoch der (Hilfswände)-Commit empfangen, wird die Hauptwand erfolgreich aus dem Modell gelöscht (vgl. Abb. 4.19). Beim Entfernen von Host Objects hat der Revit-Connector also ähnliche Schwierigkeiten wie beim Entfernen von Hosted Objects. Anders als beim Entfernen von Hosted Objects kann die Hauptwand jedoch nach Empfangen eines Commits, der nur ein Objekt enthält, immer noch per Empfangen eines Commits aus dem Modell entfernt werden.

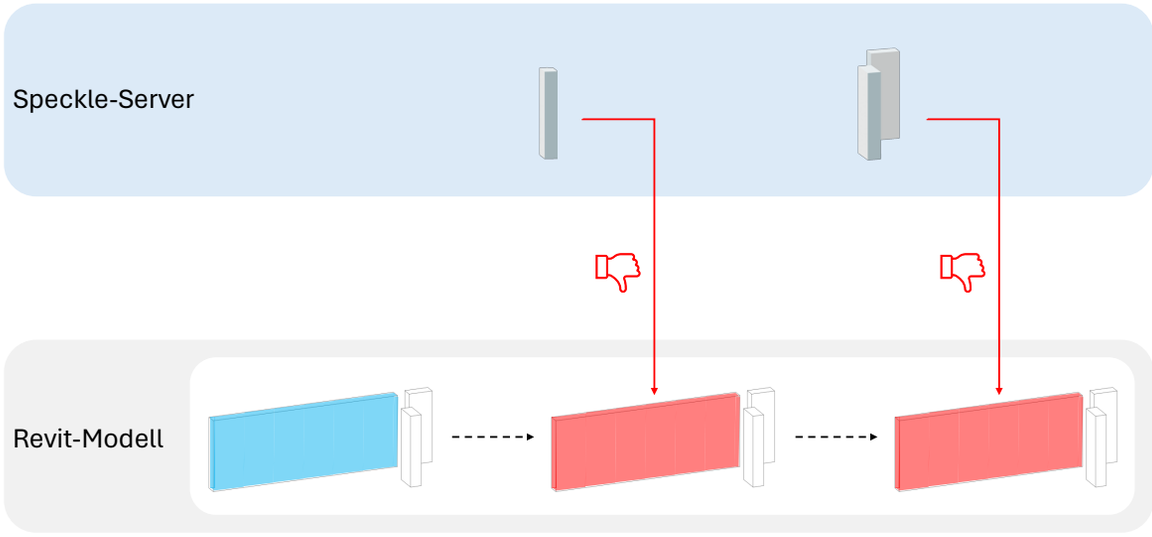


Abbildung 4.18: Manuelles Einfügen der Hauptwand und anschließendes Empfangen des (Hilfswand)-Commits und des (Hilfswände)-Commits

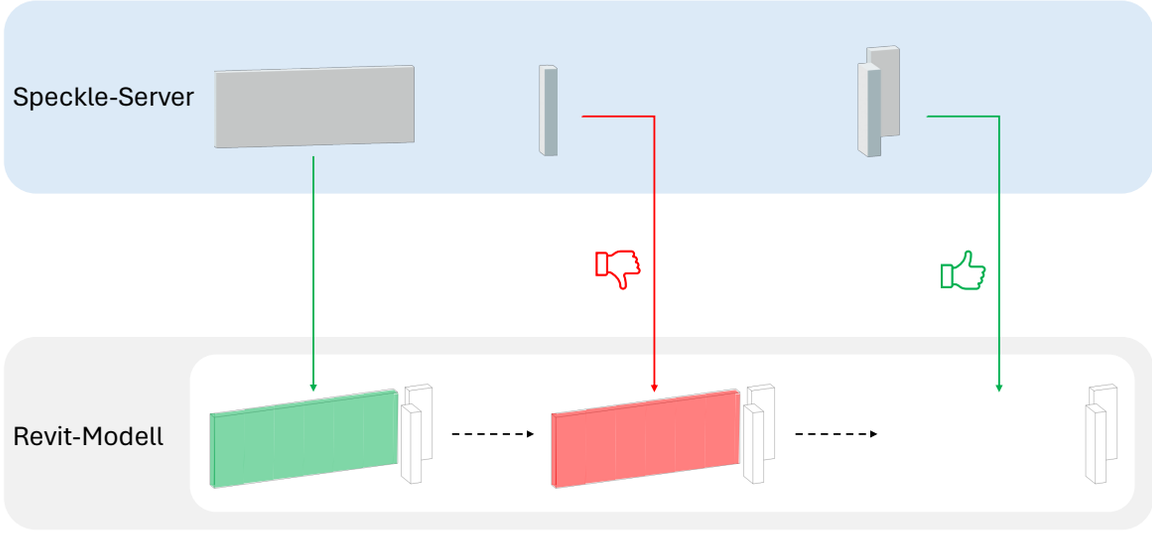


Abbildung 4.19: Einfügen der Hauptwand per Commit und anschließendes Empfangen des (Hilfswand)-Commits und des (Hilfswände)-Commits

4.3.6 Vergleich der vom Revit-Connector erstellten Speckle-Objekte

Speckle-Objekte von Hosted Objects mit unterschiedlichen Koordinaten

Zuerst sollen die JSON-Serialisierungen von zwei Speckle-Objekten verglichen werden, die aus dem selben Objekt an unterschiedlichen Positionen im Revit-Modell entstanden sind. Dafür wird das Tür-Objekt betrachtet, welches an seiner ursprünglichen Position im Ausgangsmodell bereits im (Tür)-Commit auf den Server geladen wurde. In Abschnitt 4.3.3 wurde dieses Tür-Objekt im Revit-Modell innerhalb der Hauptwand um fünf Meter in positiver x-Richtung verschoben. Die verschobene Tür wird nun in Form eines weiteren Commits an den Server gesendet und deren JSON-Serialisierung extrahiert. Die obersten Ebenen der formatierten JSON-Repräsentationen der ursprünglichen Tür und der verschobenen Tür sind in Algorithmus A.2 und Algorithmus A.3 dargestellt. Beim Vergleich der beiden Speckle-Objekte in ihrer serialisierten Form lassen sich insgesamt zehn Unterschiede feststellen:

Vier davon sind zurückzuführen auf die endliche Maschinengenauigkeit bei der Rechnung mit Gleitkommazahlen. Drei der Rundungsunterschiede liegen in den jeweiligen Volumenangaben der drei verwendeten Türmaterialien Metall, Lack und Holz. Ein weiterer Rundungsunterschied findet sich in dem Eintrag des *transform*-Attributs, der den y-Wert des Tür-Objektes in Bezug auf das interne Koordinatensystem des Revit-Modells widerspiegelt.

Weitere fünf Unterschiede zwischen den beiden Serialisierungen stammen von den unterschiedlichen Speckle-ids, die vom Revit-Connector beim Hochladen jeweils für das Tür-Objekt selbst, die drei *materialQuantities* der zuvor genannten Materialien und *transform* erstellt worden sind.

Der letzte und relevanteste Unterschied liegt in dem Eintrag von *transform*, der den x-Wert des Tür-Objektes repräsentiert. In der JSON-Darstellung des ursprünglichen Tür-Objektes hat der Eintrag den Wert 1.4424999999999937, während er in der JSON-Darstellung des verschobenen Tür-Objektes den Wert 6.442499999999991 annimmt. Die Differenz der beiden Werte beträgt fünf. Hier spiegelt sich also die Distanz von fünf Metern wider, um die die Tür im Revit-Modell verschoben worden ist. Auffällig ist, dass die Werte keine ganzen Zahlen sind, selbst wenn man die Maschinengenauigkeit außer Acht lässt. Der Wert des ursprünglichen Tür-Objektes beträgt nicht eins, obwohl die Tür mit einem Abstand von exakt einem Meter zum linken Wandende in die Wand eingefügt worden ist. Das liegt daran, dass sich dieser Positionswert nicht auf den linken Rand der Tür, sondern auf deren Mitte bezieht. Wie in 4.3.1 bereits erwähnt, ist die Tür $0,885$ Meter breit. Somit errechnen sich die beiden Werte zu $1 + \frac{0,885}{2} = 1,4425$ und $1 + \frac{0,885}{2} + 5 = 6,4425$.

Speckle-Objekte von Host Objects mit unterschiedlichen Abmessungen

Als nächstes soll untersucht werden, in welcher Hinsicht sich veränderte Abmessungen eines Revit-Objektes auf die JSON-Darstellung des entsprechenden Speckle-Objekts aus-

wirken. Daher soll die Hauptwand aus dem Ausgangsmodell mit ihrer verkürzten Variante aus Abschnitt 4.3.2 verglichen werden. Im Gegensatz zu oben werden im Folgenden nur noch die relevanten und maßgebenden Unterschiede aufgeführt und Rundungsfehler infolge der endlichen Maschinengenauigkeit nicht mehr beachtet. Die obersten Ebenen der jeweiligen JSON-Darstellungen sind in Algorithmus A.4 und Algorithmus A.5 dargestellt, wobei die im Weiteren erwähnten Unterschiede in rot hervorgehoben sind.

An erster Stelle sei erwähnt, dass sich das Attribut *baseLine* bei den beiden Wandobjekten unterscheidet. *baseLine* ist selbst ein Speckle-Objekt, das unter anderem die Attribute *start*, *end* und *length* besitzt. *start* und *end* sind ihrerseits wieder Speckle-Objekte und gehören der Speckle-Klasse *Point* an. Daher besitzen sie jeweils die Attribute *x* und *y*, welche ihre Koordinaten in Bezug auf das interne Koordinatensystem des Revit-Modells darstellen. Bei dem Speckle-Objekt der ursprünglichen Wand besitzt die x-Koordinate von *start* den Wert 0, wohingegen sie bei dem Speckle-Objekt der verkürzten Wand den Wert 2 annimmt. Auch die x-Koordinaten der jeweiligen Endpunkte unterscheiden sich: Der Endpunkt der ursprünglichen Wand besitzt den x-Wert 8 und der der verkürzten Wand den x-Wert 6. Somit ergibt sich das Attribut *length* von *baseLine* bei der langen Wand zu 8 und bei der verkürzten zu 4. Das entspricht den Abmessungen, die bei der Erstellung der Hauptwand in Abschnitt 4.3.1 bzw. deren Verkürzung in Abschnitt 4.3.2 gewählt wurden.

Ein weiterer Unterschied zwischen den beiden Speckle-Objekten lässt sich im Attribut *materialQuantities* feststellen. Wie in Abschnitt 3.2 bereits erwähnt, geben die Attribute *area* und *volume* von *materialQuantities* an, wie viel Fläche und Volumen das entsprechende Material in dem Bauteil einnimmt. Die Wände haben jeweils eine Höhe von drei Metern und eine Breite von $0,46$ Metern, die sich aus aus $0,3$ Metern Mauerwerk und $0,16$ Metern Wärmedämmung zusammensetzt (vgl. Abschnitt 4.3.1). Dementsprechend nehmen die beiden Materialien in der langen Wand eine Fläche von $8 \cdot 3 = 24$ ein und in der kurzen Wand eine Fläche von $4 \cdot 3 = 12$. Das Mauerwerk in der langen Wand besitzt ein Volumen von $8 \cdot 0,3 \cdot 3 = 7,2$, während es in der kurzen Wand nur $4 \cdot 0,3 \cdot 3 = 3,6$ Kubikmeter einnimmt. Die Wärmedämmung besitzt in der langen Wand ein Volumen von $8 \cdot 0,16 \cdot 3 = 3,84$ und in der verkürzten Wand ein Volumen von $4 \cdot 0,16 \cdot 3 = 1,92$ Kubikmetern.

Speckle-Objekte von Host Objects mit und ohne enthaltenem Hosted Object

In diesem Abschnitt wird untersucht, inwiefern sich die JSON-Serialisierung der Hauptwand mit darin enthaltener Tür von der der Hauptwand ohne Tür unterscheidet. Dafür werden die Serialisierungen der beiden Wandobjekte, die im (Hauptwand)- und (Hauptwand, Tür)-Commit enthalten sind, miteinander verglichen. Die serialisierte Wand aus dem (Hauptwand)-Commit ist bereits in Algorithmus A.4 dargestellt und die JSON-Darstellung der Wand, die die Tür enthält, ist in Algorithmus A.1 einsehbar. Auch bei diesem Vergleich werden nur die relevanten Unterschiede betrachtet.

Der maßgebende Unterschied liegt im Attribut *elements* der beiden Wandobjekte. Während *elements* bei der leeren Hauptwand mit einem Nullwert belegt ist, enthält dieses Attribut bei

der Wand mit Tür eine Referenz zum Speckle-Objekt der Tür. Diese Referenz besteht aus der Speckle-ID der Tür, die zusammen mit der Hauptwand im (Hauptwand, Tür)-Commit auf den Speckle-Server geladen wurde.

Das Attribut *materialQuantities* beinhaltet auch bei diesem Vergleich Unterschiede zwischen den beiden JSON-Repräsentationen. Denn die enthaltene Tür verringert die Flächen und Volumina, die von den Wandmaterialien eingenommen werden. So reduziert die $0,885$ breite und $2,135$ hohe Tür die Fläche der Wandmaterialien von 24 auf $24 - 0,885 \cdot 2,135 = 22,11$ Quadratmeter und beispielsweise das von der Wärmedämmung eingenommene Volumen von $3,84$ auf $3,84 - 0,885 \cdot 0,16 \cdot 2,135 = 3,54$ Kubikmeter.

Ein weiterer Unterschied lässt sich im jeweiligen Attribut *totalChildrenCount* der beiden Speckle-Objekte feststellen. Bei der Wand ohne Tür besitzt dieses Attribut den Wert 8 und bei der Wand mit Tür den Wert 22. Wie in Abschnitt 3.4 erwähnt, geben diese Werte an, wie viele andere Speckle-Objekte von dem jeweiligen Speckle-Objekt referenziert werden. Das Wand-Objekt, das das Tür-Objekt enthält, referenziert im Vergleich zum Wand-Objekt ohne Tür zusätzlich das Tür-Objekt selbst und die Speckle-Objekte, die vom Tür-Objekt referenziert werden.

4.4 Inkrementelle Updates innerhalb von AutoCAD

4.4.1 Vorbereitung der Versuche

In den nächsten Versuchen soll der AutoCAD-Connector und dessen Fähigkeit, bestehende AutoCAD-Modelle zu inkrementell zu verändern, untersucht werden. Bei den Inkrementen handelt es sich um die gleichen Veränderungen, welche bereits Gegenstand der Versuche in Abschnitt 4.3 gewesen sind: das Modifizieren, Verschieben, Einfügen und Löschen von Objekten. Die dafür benutzten Programmversionen werden in Tabelle 4.3 gelistet. Da AutoCAD als reine CAD-Software nur mit Geometrien arbeitet und diesen keine Semantik beimisst, entfällt in den folgenden Versuchen die Unterscheidung zwischen Hosted Objects und Host Objects. Analog zu den Erläuterungen in Abschnitt 4.3.1 wurden auch diese Versuche sowohl innerhalb von nur einer AutoCAD-Datei als auch zwischen zwei AutoCAD-Dateien durchgeführt. Auch hier unterschieden sich die Ergebnisse der beiden Varianten nicht, weshalb die folgenden Erläuterungen anhand einer einzelnen AutoCAD-Datei erfolgen.

Tabelle 4.3: Übersicht über die benutzten Programmversionen für die Versuche innerhalb von AutoCAD

Programm	**Version**
AutoCAD	T.181.0.0 AutoCAD 2023.1.4
Speckle-Connector für AutoCAD	2.17.0
Speckle	2.17.13

Obwohl AutoCAD keine BIM- sondern eine CAD-Software ist, wird das Ausgangsmodell für die folgenden Fallstudien dem Modell aus Abschnitt 4.3.1 nachempfunden. Dabei werden sowohl für die drei Wand-Objekte als auch für das Tür-Objekt jeweils 3D-Volumenkörper erstellt, die die selben Abmessungen aufweisen wie die erstellten Objekte im Revit-Modell. Somit entspricht der Grundriss des AutoCAD-Modells dem des Revit-Modells, welcher bereits in Abb. 4.2a dargestellt ist. Aus Gründen der Verständlichkeit wird der Volumenkörper, der die Tür repräsentiert, von nun an als „Türquader“ bezeichnet, während die restlichen Quader als „Hauptwandquader„ und „Hilfswandquader“ beschrieben werden. Hier sei jedoch ausdrücklich darauf hingewiesen, dass diese Namensgebung lediglich der Unterscheidbarkeit im Rahmen der folgenden Ausführungen dient und nicht aus der Datenstruktur von AutoCAD stammt. Dabei ist anzumerken, dass der Hauptwandquader keine Aussparung für den Türquader aufweist, sondern einen tatsächlichen Quader im geometrischen Sinn darstellt. Der Türquader befindet sich somit in dem Hauptwandquader. Zur besseren Übersichtlichkeit wird der Türquader im Modell in brauner Farbe dargestellt, wodurch sich dieser von den grau dargestellten Wandquadern abhebt. Als visueller Stil des 3D-Modells wird „Röntgen“ und als Einheit Meter gewählt.

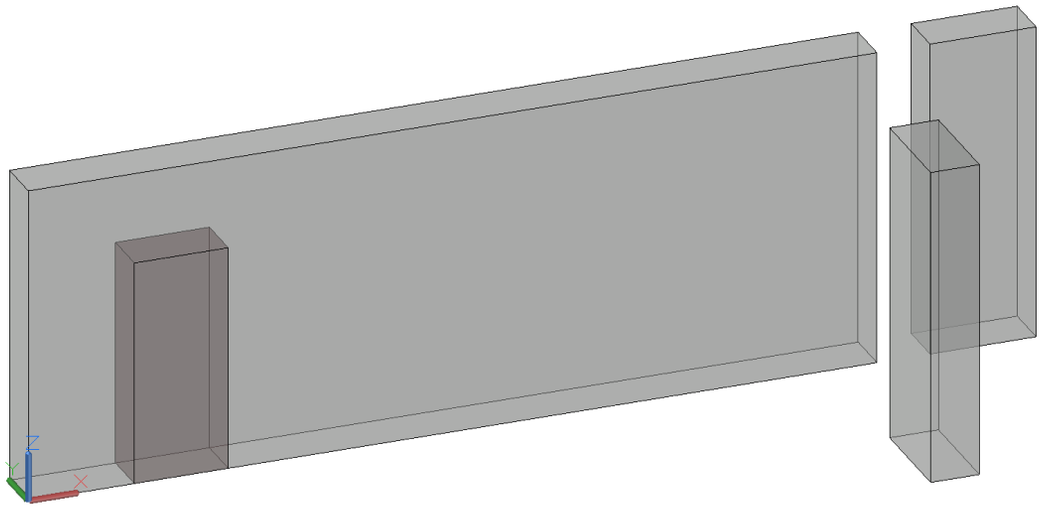


Abbildung 4.20: AutoCAD-Ausgangsmodell in der 3D-Ansicht mit Koordinatenachsen

Auf dem Speckle-Server wird ein neuer Stream für das AutoCAD-Modell erstellt, innerhalb dessen mehrere Commits hochgeladen werden (vgl. Tabelle 4.4). Da die vier im Modell enthaltenen Quader keinerlei Beziehungen zueinander aufweisen und innerhalb von AutoCAD gleichwertig sind, reicht es aus, die folgenden Versuche nur für den Türquader durchzuführen. Dieser ist repräsentativ für alle 3D-Geometrien in AutoCAD. Dementsprechend fällt die Anzahl der für die Versuche benötigten Commits geringer aus als in Abschnitt 4.3.1. Auch die Versuche mit dem AutoCAD-Connector werden mit Abbildungen visualisiert, deren farbliche Kennzeichnug der aus Abschnitt 4.3 entspricht. Somit gilt die in Abb. 4.3 dargestellte Farblegende auch für die nachfolgenden Abbildungen.

Tabelle 4.4: Übersicht über alle vom AutoCAD-Connector erstellten Commits

enthaltene Objekte	**Bezeichnung**
Türquader	(Türquader)-Commit
alle drei Wandquader	(Wandquader)-Commit

4.4.2 Objekte modifizieren, verschieben und einfügen

Die Versuche bezüglich dem Modifizieren, Verschieben und Einfügen von Geometrieobjekten werden analog zu den Versuchen in Abschnitt 4.3 durchgeführt. Zuerst wird das Modifizieren mittels eines Speckle-Commits getestet. Dabei wird zuerst die Länge des Türquaders manuell von $0,885$ auf 2 Meter vergrößert und im Anschluss der (Türquader)-Commit empfangen. Wie in Abb. 4.21 zu sehen, erhält der Quader durch das Empfangen des Commits wieder seine ursprünglichen Abmessungen. Um das Verschieben von AutoCAD-Objekten mithilfe von Speckle-Commits zu testen wird der Türquader analog zur Tür in Abschnitt 4.3.3 um fünf Meter in positiver x-Richtung verschoben. Nach Empfangen des (Türquader)-Commits wird er erfolgreich an seine Ursprungsposition zurückverschoben (vgl. Abb. 4.22). Auch das Einfügen des Türquaders mittels des (Türquader)-Commits ist möglich. Im Gegensatz zu dem entsprechenden Versuch mit dem Hosted Object in Revit muss dafür keine dazugehörige „Wand“ bzw. ein weiterer Quader im Commit enthalten sein (vgl Abb. 4.23). Auffällig ist, dass bei allen durchgeführten Versuchen der im Commit enthaltene Türquader nicht als 3D-Volumenkörper in AutoCAD empfangen wird, sondern als Vielflächennetz (vgl. Abb. 4.24). Der Quader, den es zu modifizieren, verschieben oder einzufügen gilt, wird also in seiner Grundstruktur verändert.

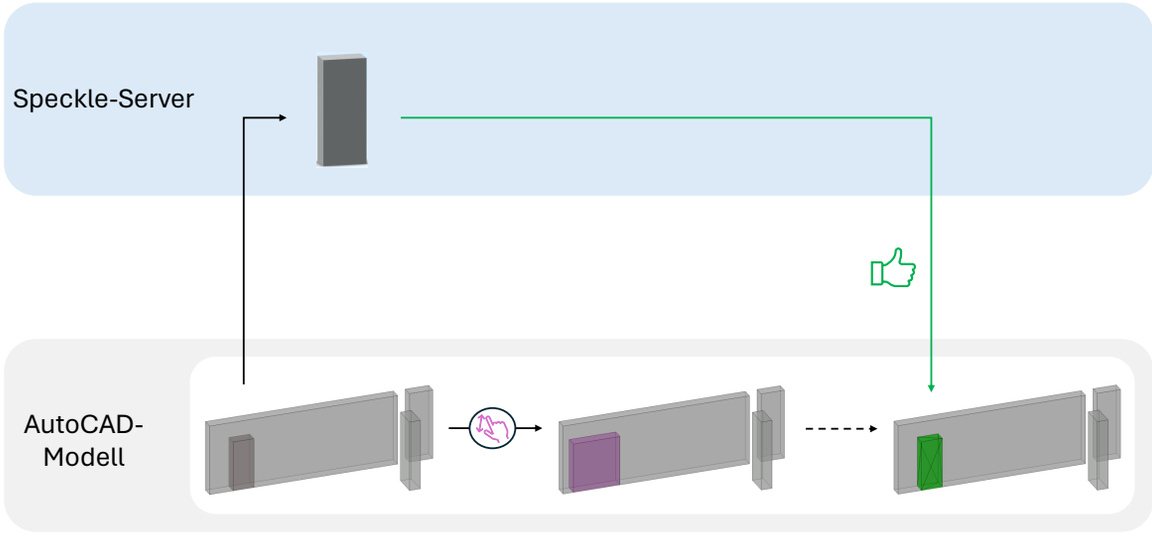


Abbildung 4.21: Modifizieren des Türquaders durch Empfangen des (Türquader)-Commits

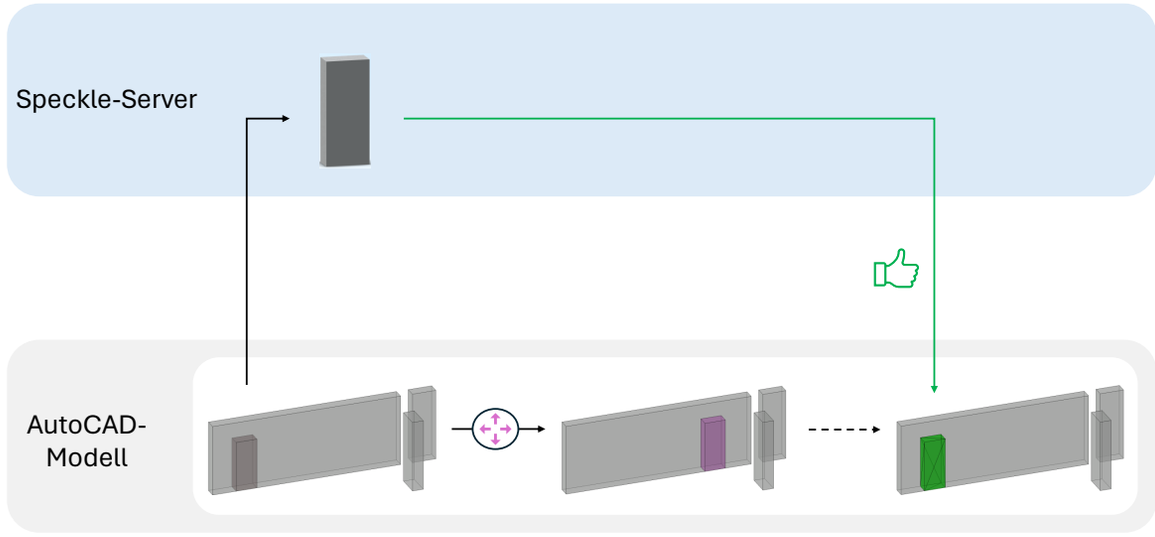


Abbildung 4.22: Verschieben des Türquaders durch Empfangen des (Türquader)-Commits

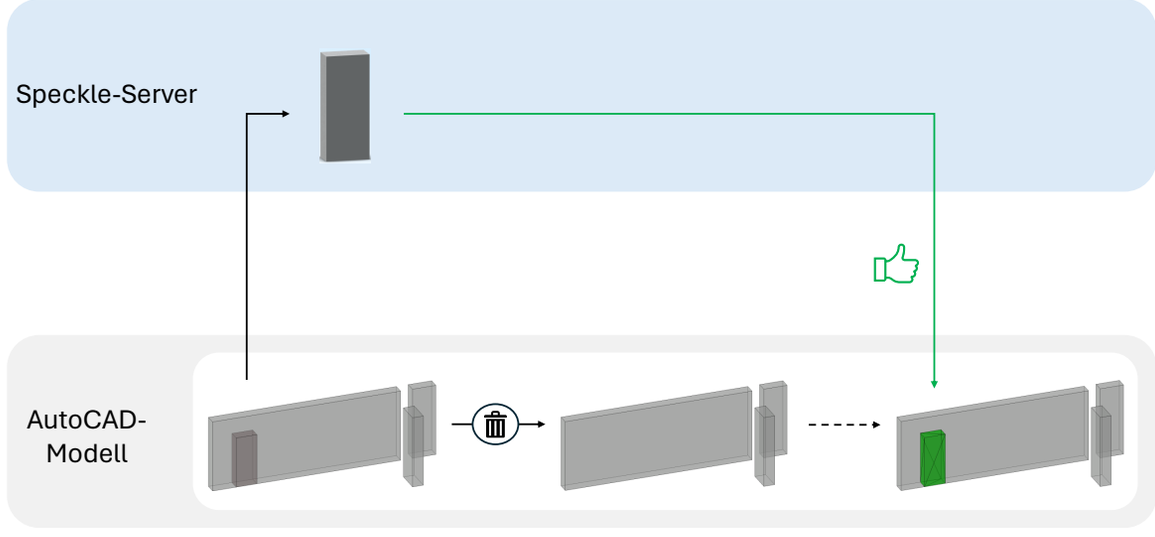


Abbildung 4.23: Einfügen des Türquaders durch Empfangen des (Türquader)-Commits

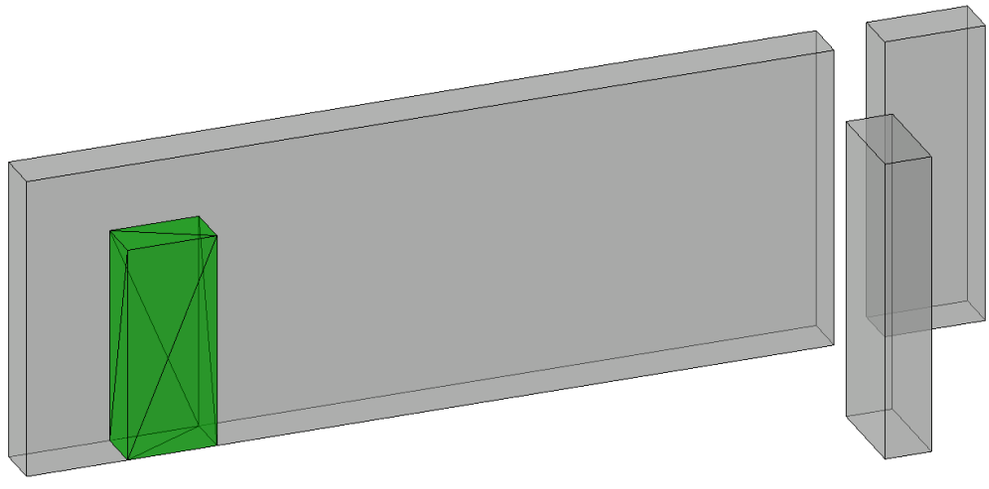


Abbildung 4.24: Der Türquader ist in AutoCAD nach dem Empfangen des (Türquader)-Commits kein 3D-Volumenkörper mehr, sondern ein Vielflächennetz aus Dreiecken.

4.4.3 Objekte entfernen

Beim Entfernen von Objekten treten die größten Unterschiede auf im Vergleich zu den Versuchen mit dem Revit-Connector. Analog zu den Versuchen in Abschnitt 4.3.5 kann ein manuell in die AutoCAD-Datei eingefügtes Objekt nicht mittels eines Speckle-Commits entfernt werden (vgl. Abb. 4.25). Doch im Gegensatz zu dem entsprechenden Versuch mit dem Revit-Connector kann mithilfe des AutoCAD-Connectors auch kein Objekt aus dem AutoCAD-Projekt entfernt werden, welches dort zuvor mithilfe eines Speckle-Commits eingefügt wurde (vgl. Abb. 4.26). Das Entfernen ist selbst mit Commits, die mehr als ein Objekt enthalten, nicht möglich. Generell ist es im Rahmen aller mit dem AutoCAD-Connector durchgeführten Versuchen und Recherchen innerhalb dieser Arbeit nie gelungen, ein 2D- bzw. 3D-Objekt mithilfe eines Speckle-Commits aus einer AutoCAD-Datei zu entfernen.

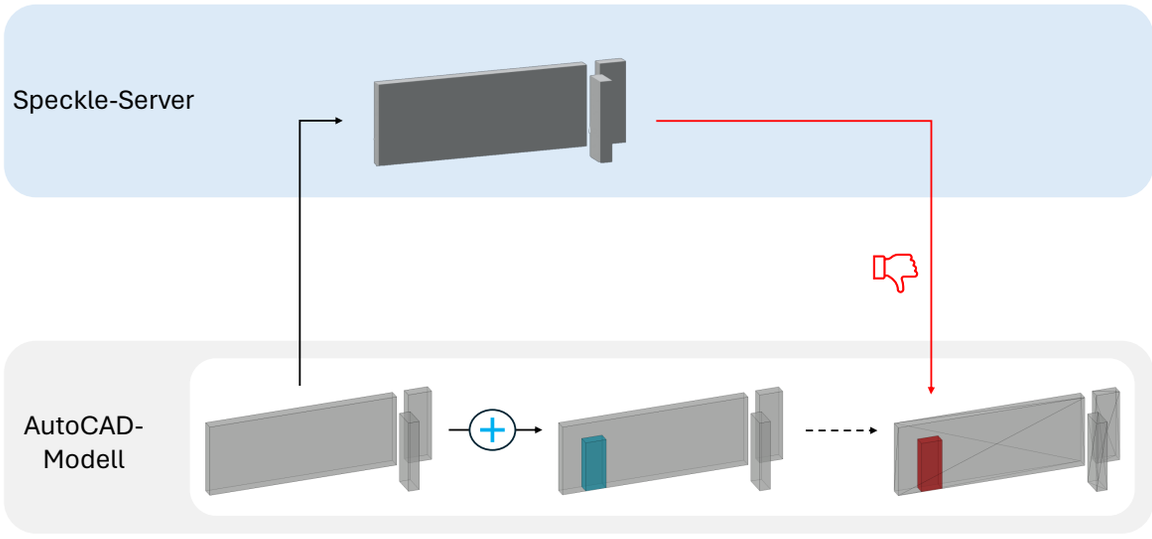


Abbildung 4.25: Manuelles Einfügen des Türquaders und anschließendes Empfangen des (Wandquader)-Commits

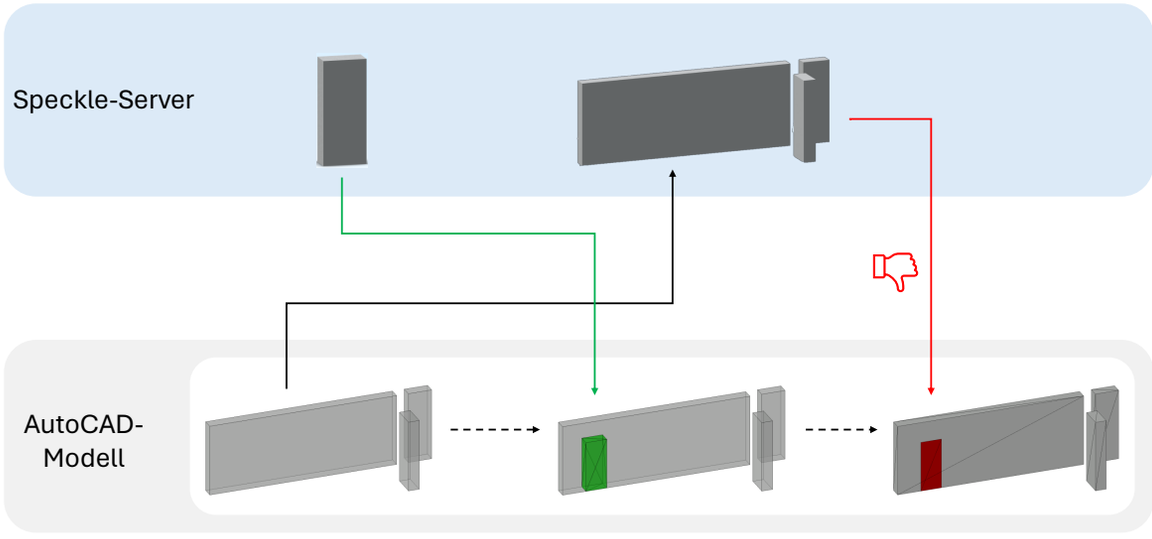


Abbildung 4.26: Einfügen des Türquaders per Commit und anschließendes Empfangen des (Wandquader)-Commits

4.4.4 Vergleich der vom AutoCAD-Connector erstellen Speckle-Objekte

In diesem Unterkapitel werden die JSON-Serialisierungen von zwei 3D-Volumenkörpern aus AutoCAD mit unterschiedlichen Abmessungen miteinander verglichen. Dafür werden die serialisierten Speckle-Objekte von Hauptwand- und Türquader betrachtet, welche in Algorithmus A.6 und Algorithmus A.7 dargestellt sind. Auch bei diesem Vergleich werden Unterschiede aufgrund von Rundungsfehlern infolge der endlichen Maschinengenauigkeit oder unterschiedlicher Speckle-IDs vernachlässigt und nur die relvanten Unterschiede aufgeführt.

Zunächst sei erwähnt, dass die beiden Speckle-Objekte unterschiedliche Werte im Attribut *applicationId* besitzen, denn sie wurden auf Grundlage unterschiedlicher Objekte im AutoCAD-Modell erstellt und haben demzufolge unterschiedliche IDs in ihrer Ursprungssoftware.

Ein weiterer Unterschied findet sich im Attribut *area*, dessen Wert die Gesamtoberfläche des jeweiligen Mesh-Objektes angibt. Der Wert errechnet sich bei dem 8 Meter langen, $0,46$ Meter breiten und 3 Meter hohen Hauptwandquader zu $2 \cdot (8 \cdot 3 + 8 \cdot 0,46 + 3 \cdot 0,46) = 58,12$ Quadratmetern, während der Türquader lediglich eine Oberfläche von $2 \cdot (0,885 \cdot 2,135 + 0,885 \cdot 0,46 + 2,135 \cdot 0,46) = 6,56$ Quadratmetern besitzt. Analog dazu unterscheiden sich auch die Werte des jeweiligen Attributs *volume*, denn das Volumen des Hauptwandquaders beträgt $8 \cdot 3 \cdot 0,46 = 11,04$ Kubikmeter und das des Türquaders $0,885 \cdot 2,135 \cdot 0,46 = 0,87$ Kubikmeter.

Unterschiede lassen sich auch in den jeweiligen Werten der Attribute *xSize* und *zSize* feststellen. Beide sind selbst ein Speckle-Objekt und ein Wert des Attributs *bbox*. In den Attributen *start* und *end* von *xSize*, *ySize* und *zSize* sind die Koordinaten der diagonal gegenüberliegenden Eckpunkte eines Quaders gespeichert. Dementsprechend unterscheidet sich *ySize* bei Hauptwand- und Türquader nicht, wohl aber *xSize* und *zSize*. In der Serialisierung des Hauptquaders betragen *start* und *end* von *xSize* 0 und 8, während *start* und *end* von *zSize* die Werte 0 und 3 besitzen. Die jeweiligen Startwerte stimmen mit dem Ausgangsmodell überein, denn die linke vordere Ecke des Hauptwandquaders befindet sich im AutoCAD-Modell im Koordinatenursprung. In der Serialisierung des Türquaders hingegen nehmen *start* und *end* von *xSize* die Werte 1 und $1,885$ an und die von *zSize* die Werte 0 und $2,135$. Der Startwert von *zSize* ist bei Hauptwand- und Türquader also gleich. Auch das stimmt mit dem Ausgangsmodell überein, denn der Türquader wurde auf gleicher Höhe mit einem Abstand von einem Meter zum linken Ende des Hauptwandquaders eingefügt.

4.5 Datenaustausch zwischen Revit und AutoCAD mittels Speckle

Im Folgenden soll untersucht werden, inwieweit der Datenaustausch zwischen einem BIM- und einem CAD-Autorenwerkzeug mittels der entsprechenden Speckle-Connectors

realisiert werden kann. Im Rahmen dieses Versuches stammen sowohl die BIM-Software als auch die CAD-Software vom selben Hersteller: Autodesk.

4.5.1 Objekte aus AutoCAD in Revit

Als Erstes werden alle Volumenkörper des AutoCAD-Ausgangsmodells, welches in Abschnitt 4.4.1 erstellt worden ist, unter Anwendung des AutoCAD-Connectors in einem Commit an den Speckle-Server geschickt. Anschließend wird eine neue Datei in Revit erstellt, in der dieser Commit mithilfe des Revit-Connectors empfangen wird. Das daraus resultierende Revit-Modell ist in Abb. 4.27 dargestellt. Die Objekte, die ursprünglich als 3D-Volumenkörper in AutoCAD erstellt worden sind, werden in Revit allesamt als Mesh-Objekte ohne jegliche Semantik empfangen.

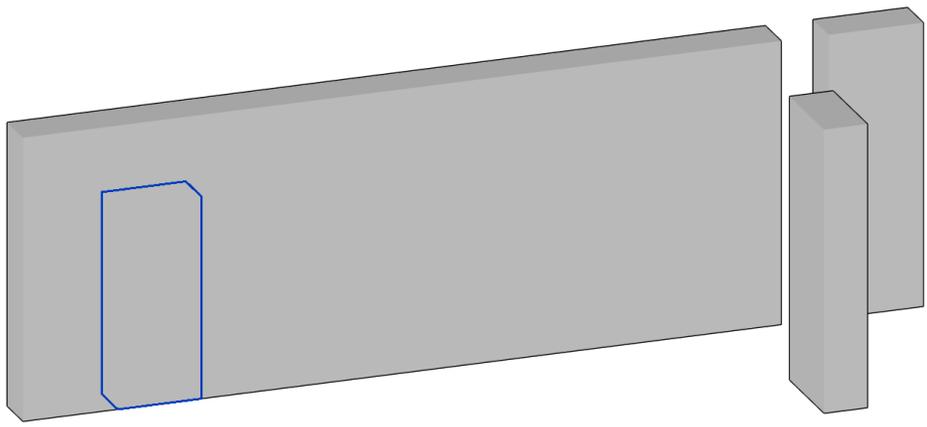


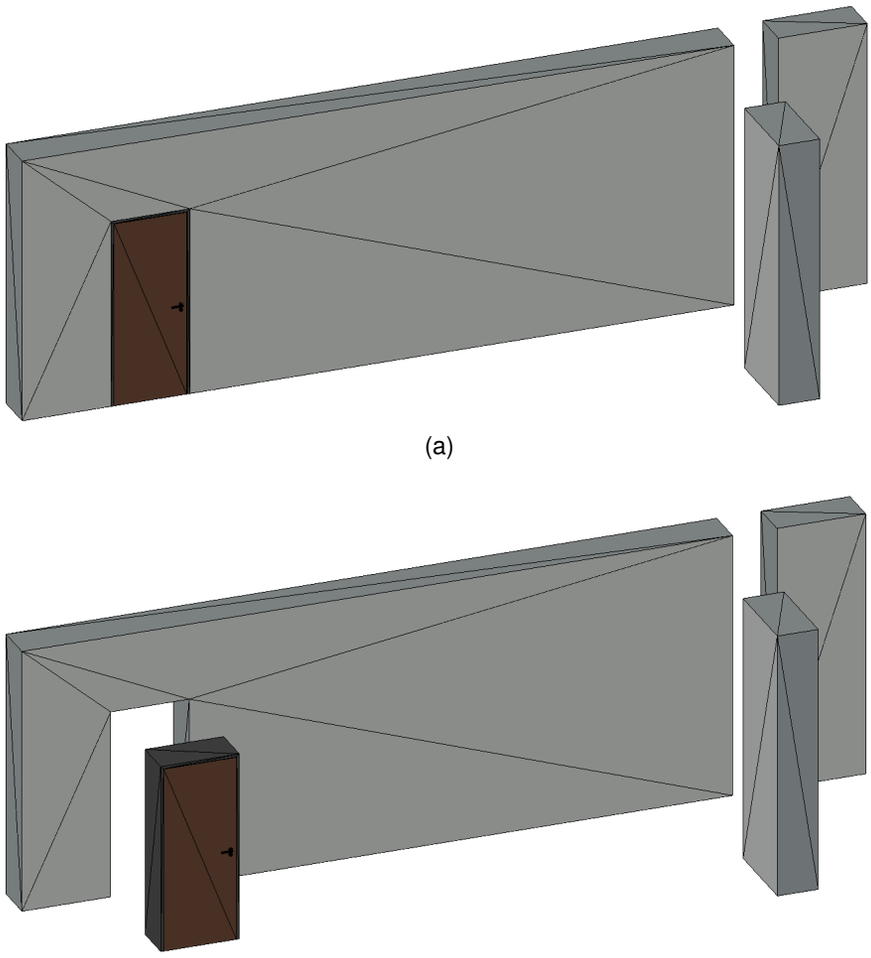
Abbildung 4.27: Durch Empfangen des AutoCAD-Modells entstehen in Revit vier Mesh-Objekte. Das Türquader-Mesh wurde manuell blau umrandet, da es ansonsten nicht sichtbar wäre.

4.5.2 Objekte aus Revit in AutoCAD

Zuerst wird das gesamte in Abschnitt 4.3.1 erstellte Ausgangsmodell mittels des Revit-Connectors in einem Commit auf den Speckle-Server geladen. Daraufhin wird eine neue AutoCAD-Datei erstellt und darin der soeben erstellte Commit mithilfe des AutoCAD-Connectors vom Speckle-Server empfangen. Dadurch entsteht ein AutoCAD-Modell, welches in Abb. 4.28 zu sehen ist. Dieses Modell enthält zwei verschiedene Arten von geometrischen Objekten. Die drei Wände, die ursprünglich in der Revit-Datei erstellt worden waren, wurden als Vielflächennetze in AutoCAD empfangen. Die Tür hingegen wurde als Block mit dem Namen „TU DF 1 - Rahmenstock flächenbündig - ML - 885 x 2135“ empfangen. Dieses Block-Objekt besteht aus drei verschiedenen triangulierten Oberflächenbeschreibungen, welche den Türrahmen, das Türblatt und die Türklinke darstellen (vgl. Abb. 4.29).

Weiterhin wurden für die verschiedenen Bauteilkategorien der empfangenen Objekte Layer in der AutoCAD-Datei angelegt. In diesem Fall wurde nur ein Layer names „Walls“ erstellt, dem sowohl alle Vielflächennetze als auch das Block-Objekt angehört. Nichtsdestotrotz

ist der Block ein unabhängiges Objekt im AutoCAD-Modell und kann beliebig im Modell verschoben werden (vgl. Abb. 4.28b). Würde eine weiterer Block durch Empfangen des (Tür)-Commit (vgl. Tabelle 4.2) in das AutoCAD-Modell eingefügt werden, würde diese automatisch einem eigens erstellten Layer names „Doors“ zugeordnet werden.



(a)

(b)

Abbildung 4.28: AutoCAD-Modell, das durch Empfangen des Revit-Ausgangsmodells entstanden ist, in der 3D-Ansicht

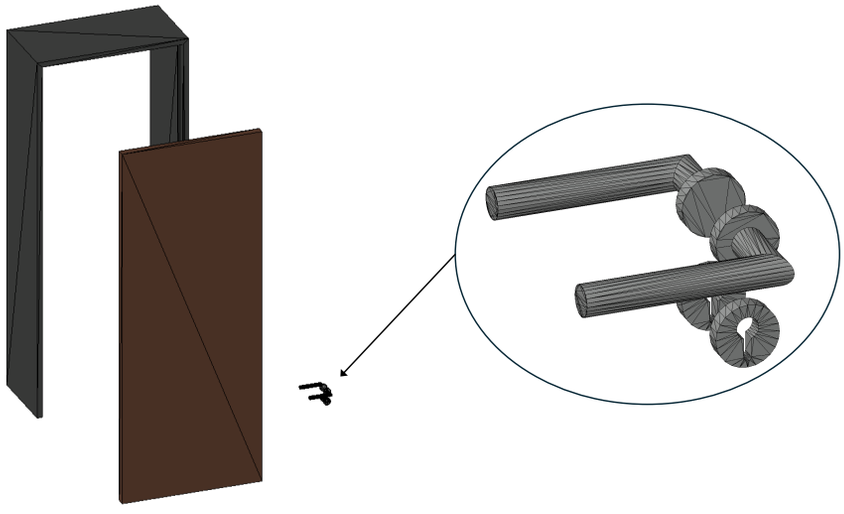


Abbildung 4.29: In seine drei Bestandteile zerlegter Block

Kapitel 5

Diskussion der Ergebnisse

5.1 Ergebnisse des Revit-Connectors

Die durchgeführten Versuche haben gezeigt, dass der Revit-Connector von Speckle in vielen Fällen die erwarteten Ergebnisse zeigt und inkrementelle Updates des Revit-Modells erfolgreich umsetzt. Vor allem im Hinblick auf Host Objects in Revit-Modellen ist das Modifizieren, Verschieben und Einfügen mithilfe des Revit-Connectors sehr intuitiv. Auch das Verschieben von Hosted Objects im Modell ist leicht mithilfe des Connectors umsetzbar.

Grenzen des Revit-Connectors existieren aber in Situationen, bei denen Hosted Objects modifiziert, eingefügt und gelöscht werden sollen. Vor allem das Löschen von Objekten ist bisher noch unintuitiv, auch im Hinblick auf Host Objects. Abgesehen davon muss man als Anwender des Revit-Connectors bedacht vorgehen bei der Erstellung von Commits. Wie die Versuche gezeigt haben, hängen die Art und das Ausmaß der durch den empfangenen Commit hervorgerufenen Änderungen im Modell von den Objekten ab, die im Commit enthalten sind. Beispielsweise entscheidet die Anzahl der enthaltenen Objekte darüber, ob Objekte aus dem Revit-Modell gelöscht werden oder nicht. Während in Abschnitt 4.3.4 das Empfangen einer einzelnen Wand dazu geführt hat, dass diese in das Modell eingefügt wurde, führte in Abschnitt 4.3.5 das Empfangen der selben Wand und einer weiteren dazu, dass das gesamte restliche Modell gelöscht wurde.

Ergebnisse bezüglich Hosted Objects

Im Rahmen aller mit dem Revit-Connector durchgeführten Versuche ist es nie gelungen, die Abmessungen eines Hosted Objects durch Empfangen eines Commits zu verändern. Sowohl das Modifizieren mittels eines Commits, der nur das Hosted Object enthält, als auch mittels eines Commits, der zusätzlich noch das zugehörige Host Object enthält, blieb stets erfolglos. Das Verschieben von Hosted Objects funktioniert hingegen äußerst intuitiv. Dafür reicht es aus, einen Commit vom Speckle-Server zu empfangen, der das zu verschiebende Hosted Object an der gewünschten Stelle im Host Object enthält. Soll ein Hosted Object jedoch in das Revit-Modell eingefügt werden, so muss der dafür empfangene Commit neben dem einzufügenden Hosted Object auch dessen Host Object beinhalten. Enthält der Commit nur das Hosted Object, bleibt das Revit-Modell nach dem Empfangen unverändert. Hinsichtlich des Entfernens von Hosted Objects aus einem Revit-Modell lässt sich festhalten, dass das Entfernen mittels Empfangen eines Commits nur möglich ist, wenn neben dessen leerem Host Object auch noch mindestens ein weiteres

Objekt in dem Commit enthalten ist und das Hosted Object bereits per Commit in das Revit-Modell empfangen worden ist. Ist nur das leere Host Object im empfangenen Commit enthalten, bleibt das Revit-Modell unverändert und das Hosted Object kann daraufhin nur noch manuell aus dem Modell entfernt werden.

Ergebnisse bezüglich Host Objects

Für das Modifizieren, Verschieben und Einfügen von Host Objects in einem Revit-Modell ist es ausreichend, einen Commit zu empfangen, der nur das Host Object mit den gewünschten Abmessungen an der gewünschten Position im Modell enthält. Somit unterscheiden sich Host Objects von Hosted Objects hinsichtlich dem Modifizieren und dem Verschieben. In Bezug auf das Entfernen von Host Objects aus einem Revit-Modell tritt das gleiche Phänomen auf wie bei Hosted Objects: Das Entfernen mittels des Revit-Connectors ist nur möglich, solange der empfangene Commit mehr als ein Speckle-Objekt enthält und das zu entfernende Objekt zuvor bereits in einem Commit vom Speckle-Server empfangen worden ist. Diese Feststellung ergibt sich aus dem Ergebnis des Versuchs zum Entfernen von Host Objects in Abschnitt 4.3.5 und deckt sich mit dem Versuchsergebnis zum Modifizieren von Host Objects in Abschnitt 4.3.2. Beim Modifizieren der Hauptwand war im (kurze Wand)-Commit nur ein Objekt enthalten, wodurch die beiden Hilfswände nach Empfangen des Commits im Revit-Modell verblieben. Wäre neben der kurzen Hauptwand auch noch eine der beiden Hilfswände im Commit enthalten gewesen, wäre die andere Hilfswand nach Empfangen des Commits aus dem Modell entfernt worden.

Hypothesen für das Entfernen von Objekten aus einem Revit-Modell

Entsprechend der Ergebnisse aus Abschnitt 4.3.5 lassen sich zwei Hypothesen aufstellen, die das Verhalten des Revit-Connectors im Hinblick auf das Entfernen von Objekten aus einem Revit-Modell mittels des Empfangens von Speckle-Commits beschreiben. Sie gelten auch, wenn das Revit-Projekt nach dem Einfügen des Objekts (manuell oder per Commit) gespeichert und neu geöffnet worden ist. Die Hypothesen lauten:

1. Objekte, die durch das Empfangen eines Commits in das Revit-Modell eingefügt worden sind, können durch einen anderen Commit, der mehr als nur ein Objekt enthält, wieder aus dem Modell entfernt werden. Dabei spielt die zeitliche Reihenfolge, in der der „Einfügen-Commit“ und der „Entfernen-Commit“ auf den Speckle-Server hochgeladen worden sind, keine Rolle.
2. Objekte, die manuell in das Revit-Modell eingefügt worden sind, können nicht durch Empfangen eines Commits entfernt werden. Nachdem diese neu eingefügten Objekte jedoch in einem Commit auf den Speckle-Server hochgeladen und wieder empfangen worden sind, können auch sie per Speckle-Commit aus dem Revit-Modell entfernt werden.

Im Rahmen dieser Arbeit wurde am 18.01.2024 eine neues Thema auf der Website „Speckle Community Forum“ eröffnet, das das Entfernen von Objekten aus einem Revit-Modell mithilfe des Revit-Connectors behandelte.[2] Dort wurde unter anderem auch das Problem angesprochen, dass ein Hosted Object nach dem Empfangen eines Commits, der nur dessen leeres Host Object enthält, nur noch manuell aus dem Modell entfernt werden kann. Göker, der derzeitige Lösungsarchitekt von Speckle, antwortete am 14.02.2024 auf die dort gestellten Fragen mit „I believe this is not the expected behaviour.“. Bis zum Abgabezeitpunkt dieser Arbeit wurden keine weiteren Antworten innerhalb dieses Themas veröffentlicht, die Aufschluss geben könnten über die Ursache der beobachteten Phänomene.

5.2 Ergebnisse des AutoCAD-Connectors

Um ein Objekt mittels des AutoCAD-Connectors in einem AutoCAD-Modell zu modifizieren, zu verschieben oder neu in das Modell einzufügen, muss ein Commit vom Speckle-Server empfangen werden, der das Objekt mit den gewünschten Abmessungen an der gewünschten Position enthält. Falls es sich bei dem Objekt, das im AutoCAD-Modell modifiziert oder verschoben werden soll, jedoch um einen 3D-Volumenkörper handelt, so wird das Objekt mit Empfangen des Commits in seiner Grundstruktur verändert: Der Volumenkörper wird zu einem Vielflächennetz. Auch ein Objekt, das als 3D-Volumenkörper an den Speckle-Server gesendet worden ist, wird beim erneuten Empfangen als Vielflächennetz in das AutoCAD-Modell eingefügt. Der Connector konvertiert also 3D-Volumenkörper, die aus AutoCAD stammen, in Mesh-Speckle-Objekte. Er ist jedoch nicht in der Lage, aus diesen Mesh-Objekten die 3D-Volumenkörper zu rekonstruieren. Streng genommen könnte man also behaupten, dass das Modifizieren, Verschieben und Einfügen eines 3D-Volumenkörpers mittels dem AutoCAD-Connector nicht funktioniert.

Das Entfernen von Objekten aus einem AutoCAD-Modell durch Empfangen eines Speckle-Commits scheint mit der aktuellen Version des AutoCAD-Connectors, unmöglich zu sein. Sowohl manuell als auch per Commit in das Modell eingefügte Objekte können weder mittels einem Commit, der nur ein Objekt enthält, noch mittels einem Commit, der mehrere Objekte enthält, aus dem Modell entfernt werden. Auch dieses Problem wurde in dem Thema angesprochen, welches auf dem „Speckle Community Forum“ eröffnet und bereits in Abschnitt 5.1 erwähnt wurde, doch bis zum Abgabezeitpunkt dieser Arbeit gab es keine Antwort darauf.

Was den Datenaustausch von Revit nach AutoCAD mittels der beiden Connectors angeht, wird ein Teil der im Revit-Modell enthaltenen Semantik in Form von Layern in das empfangene AutoCAD-Modell übertragen. Um die Speckle-Objekte, die ursprünglich in der BIM-Software Revit erzeugt worden sind, geometrisch darzustellen, greift der AutoCAD-Connector auf das Attribut *displayValue* der jeweiligen empfangenen Speckle-Objekte zu. Mit den darin gespeicherten triangulierten Oberflächenbeschreibungen lassen sich

[2] https://speckle.community/t/deleting-hosted-elements-e-g-doors-windows-in-revit-via-speckle/8744/8

auch Speckle-Objekte in einer CAD-Software darstellen, die ihren Ursprung in einer BIM-Software haben.

5.3 Vorzüge und Grenzen von Speckle

Vorzüge von Speckle

Die durchgeführten Versuche haben gezeigt, dass Speckle durchaus in der Lage ist, inkrementelle Updates erfolgreich zu realisieren. Das gilt sowohl für inkrementelle Updates innerhalb eines BIM-Autorensystems als auch für inkrementelle Updates innerhalb einer CAD-Software. Doch vor allem die Analyse ihrer Datenstruktur zeigt die Vorteile der Kollaborationsplattform Speckle auf. An erster Stelle sei die Erweiterbarkeit von Speckle erwähnt. Die quelloffene Plattform ermöglicht es ihren Nutzern mittels entsprechender APIs und SDKs, die Funktionalität von Speckle zu erweitern und an benutzerspezifische Anforderungen anzupassen. Beispielsweise können eigene Connectors programmiert oder benutzerspezifische Speckle-Klassen mit eventuell als lösbar gekennzeichneten Attributen erstellt werden. Anders als starre Datenstrukturen vordefinierter Modellschemata wie IFC (MONDINO, 2021), die auf Vollständigkeit setzen, setzt Speckle somit auf Erweiterbarkeit und lebt von seiner aktiven und offenen Entwicklungsgemeinschaft.

Ein Merkmal, das Speckles Datenformat mit IFC teilt, ist die Herstellerneutralität. Speckle-Objektklassen sind so konzeptioniert, dass ihre Instanzen in die AEC-Softwareprodukte, für die ein entsprechender Speckle-Connector exisitiert, übertragen werden können. Auch die Connectors sind ihrerseits darauf ausgelegt, softwarespezifischen Objekte in das herstellerneutrale Format von Speckle zu konvertieren. Speckle ist somit ein alternativer Ansatz zu IFC, was den herstellerneutralen Datenaustausch in der Baubranche angeht.

Das Konzept der losgelösten Attribute und die damit verbundenen Referenzierungen verringern die zu übertragenden Datenmengen und sorgen für eine höhere Übertragungsgeschwindigkeit. Denn anstatt ein Speckle-Objekt, das Bestandteil von anderen Speckle-Objekten ist, mehrmals auf dem Speckle-Server speichern zu müssen, kann es nur einmal gespeichert werden und von den darauf aufbauenden Speckle-Objekten referenziert werden.

Ein weiterer Vorzug von Speckle besteht darin, dass es mithilfe des 3D-Viewers in der webbasierten Schnittstelle von Speckle möglich ist, die übertragenen Inkremente bzw. die veränderten Objekte direkt zu visualisieren. Zusammen mit der Unterteilung eines Bauprojektes in Streams, Branches und Commits wird es dem Anwender somit erleichtert, die Übersicht über die ausgetauschten Daten zu behalten.

Grenzen von Speckle

Die Versuche in Kapitel 4 haben jedoch auch aufgezeigt, wo aktuell noch die Grenzen von Speckle liegen. Es zeigen sich vor allem Limitationen beim Entfernen von Objekten

aus BIM- oder CAD-Modellen. Im Gegensatz zu VCSs in der Softwareentwicklung wie Git oder dem graphenbasierten Ansatz für inkrementelle Updates von BIM-Modellen von ESSER et al. (2022) ist es mit Speckle nicht möglich, intuitiv einen „Löschen-Befehl“ als Inkrement zu versenden. Um ein Objekt erfolgreich aus einem Revit-Modell zu entfernen, müssen stattdessen immer mindestens zwei Objekte im empfangenen Commit enthalten sein. Das Entfernen von Objekten aus AutoCAD-Modellen ist hingegen gar nicht möglich.

Weiterhin kann es zu Inkonsistenzen kommen zwischen der visuellen Darstellung der Commit-Objekte im 3D-Viewer von Speckle und der Visualisierung der Objekte in der AEC-Software, in die der Commit empfangen wurde. Beispielsweise wird eine Wand, die im Revit-Modell eine Tür beherbergt und ohne die Tür an den Speckle-Server gesendet wird, im 3D-Viewer als Wand mit Durchbruch dargestellt. Wird dieses Wand-Objekt jedoch vom Speckle-Server in eine andere Revit-Datei empfangen, wird dort eine vollständige Wand ohne Druchbruch erzeugt. Auch dieses Phänomen könnte zu Verwirrung auf Seiten des Endanwenders führen.

Darüber hinaus ist aus der Analyse der Speckle-Datenstruktur und der durchgeführten Versuche hervorgegangen, dass der Datenaustausch stets mittels dem Versenden und Empfangen vollständiger Speckle-Objekte realisiert wird. Beim grahenbasierten Ansatz für inkrementelle Updates werden hingegen anstatt vollständiger Objekte noch feingranularere Inkremente versendet. Genauer gesagt werden nur die Veränderungen der jeweiligen Objekte übermittelt. Zum Beispiel ändert sich bei einer Tür, die innerhalb eines Revit-Modells um fünf Meter in x-Richtung verschoben wird, im Endeffekt nur ein einziger Attributwert (vgl. Algorithmus A.2 und Algorithmus A.3). Doch anstatt nur diesen veränderten Wert zu übermitteln, erstellt der Revit-Connector ein völlig neues Speckle-Objekt und sendet dieses an den Server. Was Speicherbedarf, Übertragungsgeschwindigkeit und Datenredundanz angeht, existiert hier also noch Spielraum für Optimierung.

Ein weitere Limitation von Speckle besteht aktuell noch in dem Zusammenführen von Teilmodellen. Während in der Softwareentwicklung der Merge üblicherweise zentral in dem VCS durchgeführt werden kann, ist es nicht möglich, unterschiedliche (Teil-)Modelle eines Gebäudes zentral auf dem Speckle-Server zusammenzuführen. Stattdessen müssen diese über einen entsprechenden Connector in eine AEC-Autorensoftware empfangen werden und dort manuell auf Konflikte untersucht und vereinigt werden.

Kapitel 6

Fazit und Ausblick

Fazit

In dieser Abhandlung wurde die Kollaborationsplattform Speckle und der von ihr verfolgte Ansatz bei der Umsetzung inkrementeller Updates für BIM-Modelle analysiert. Anhand der durchgeführten Versuche und Analysen wurden sowohl Vorzüge als auch Grenzen von Speckle ersichtlich.

Speckle bietet großes Potential hinsichtlich Versionskontrolle und Datenmanagement von Gebäudemodellen und ist auf dem Weg, ein Version Control System für BIM-Modelle zu werden. Inkremente, wie das Verschieben oder Einfügen von Bauteilen, lassen sich größtenteils intuitiv mittels Speckle von einem digitalen Gebäudemodell zum nächsten übertragen. Auch das Übermitteln veränderter Bauteilabmessungen funktioniert weitestgehend gut. Jedoch bestehen derzeit noch Probleme beim Versenden von Inkrementen, die ein Bauteil aus einem Gebäudemodell entfernen sollen. In der BIM-Software Revit können solche Inkremente nur über Umwege empfangen werden und in der CAD-Software AutoCAD ist das Entfernen von Objekten mittels empfangener Inkremente momentan gar nicht möglich.

Als digitale Kollaborationsplattform im AEC-Bereich liegt die Frage nahe, ob Speckle eine CDE entsprechen BIM-Stufe 3 darstellt. Speckle weist viele Charakteristika auf, die der einer CDE nach BIM-Stufe 3 entsprechen. Anstatt Informationscontainer zu verwalten, die monolithische Dateien enthalten, erlaubt die Speckle-Plattform seinen Nutzern den feingranularen Zugriff auf ausgewählte Objekte. Dementsprechend beschreiben POINET et al. (2020) Speckle als „distributed CDE“. Jedoch definiert die ISO 19650-1 als Kernanforderung an eine CDE, dass die von ihr verwalteten Informationen mittels formaler Status klar gekennzeichnet werden müssen hinsichtlich ihres Reifegrads und ihrer Zuverlässigkeit. Da Speckle keinerlei derartige Kennzeichnung von Objekten vorsieht, kann Speckle nicht als CDE im Sinne der ISO 19650-1 bezeichnet werden.

Bei einer Unterscheidung zwischen „Closed BIM“ und „Open BIM“ lässt sich Speckle eindeutig dem Bereich des „Open BIM“ zuordnen. Speckle-Objekte stellen ein herstellerneutrales Datenformat dar, welches den Datenaustausch zwischen Softwareprodukten verschiedener Hersteller ermöglicht. Voraussetzung dafür ist jedoch, das die jeweiligen Softwareprodukte über entsprechende Speckle-Connectors verfügen, mit deren Speckle-Objekte empfangen und gegebenenfalls auch erzeugt werden können. Was die Unterscheidung zwischen „little bim“ und „BIG BIM“ angeht, fällt Speckle in den Bereich des „BIG BIM“. Fachmodelle, die mittels unterschiedlicher Autorensysteme erzeugt worden

sind, lassen sich mithilfe der Speckle-Connectors und dem Speckle-Server unter den beteiligten Planern austauschen und zur Koordination nutzen. Somit verfolgt Speckle das Konzept des „BIG Open BIM“ und kann der BIM-Stufe 3 nach ISO 19650-1 zugeordnet werden.

Ausblick

Die Versuche im Rahmen dieser Arbeit wurden mit einer CAD-Software sowie einer BIM-Autorensoftware durchgeführt. Beide Softwareprodukte stammen jedoch vom Hersteller Autodesk. Um Speckles Potenzial hinsichtlich des herstellerneutralen Datenaustauschs zu analysieren, könnte es Gegenstand zukünftiger Forschung sein, Versuche mit Speckle-Connectors für Softwareprodukte unterschiedlicher Hersteller durchzuführen. Beispielsweise könnte untersucht werden, bis zu welchem geometrischen und semantischen Detaillierungsgrad Objekte aus der BIM-Software Revit in die BIM-Software Archicad vom Hersteller Graphisoft übertragen werden können und vice versa.

Dies könnte kombiniert werden mit der Durchführung einer konkreten Fallstudie im Zuge eines (fiktionalen) BIM-Projektes. Als Forschungsfrage würde sich beispielsweise anbieten: „Vereinfacht die Verwendung von Speckle die Zusammenarbeit in einem interdisziplinären Team bei der Planung eines Einfamilienhauses?“ Eine solche Fallstudie könnte Aufschluss geben über praxisrelevante Aspekte bei der Nutzung der Speckle-Plattform als zentralen Ort für den Datenaustausch und die Koordination zwischen verschiedenen beteiligten Planern, wie Statikern, Architekten oder Gebäudetechnikern, die jeweils domänenspezifische Softwareprodukte benutzen.

Weiterhin wurde in dieser Arbeit der konkrete Aufbau und die Funktionsweise der Speckle-Connectors auf Quellcodeebene außer Acht gelassen. Interessant wäre daher auch eine Untersuchung der jeweiligen Quellcodes, die die Gründe für das beobachtete Verhalten der Connectors im Rahmen der durchgeführten Versuche darlegen könnte. Insbesondere das Verhalten des Revit-Connectors beim Empfangen von Commits, die nur ein Objekt enthalten, im Vergleich zum Empfangen von Commits mit mehreren Objekten, stellt einen interessanten Untersuchungsgegenstand auf Quellcodeebene dar. Jedoch ist zu beachten, dass die Speckle-Connectors konstant weiterentwickelt werden und somit die in dieser Abhandlung erarbeiteten Ergebnisse in Zukunft möglicherweise an Aktualität verlieren werden. Des Weiteren könnte mit einer Analyse der Quellcodes festgestellt werden, bis zu welchem Detaillierungsgrad die Connectors Objekte, die in einer AEC-Software erstellt worden sind, in Speckle-Objekte umwandeln. In dieser Arbeit wurden nur die bereits serialisierten Speckle-Objekte analysiert und miteinander verglichen, doch für den Vergleich von nativen Objekten in einer Software und Speckle-Objekten könnte eine mögliche Forschungsfrage lauten: „Werden die in einem Objekt enthaltenen Informationen bei dessen Konvertierung aus dem softwarespezifischen Format in das neutrale Speckle-Format vollständig erhalten?“

Anhang A

JSON-Serialisierungen der Speckle-Objekte

Die Werte mancher Attribute der nachfolgenden JSON-Serialisierungen wurden aus Gründen der Übersichtlichkeit durch {...} ersetzt. Die vollständigen JSON-Dateien sind zusammen mit dem benutzten Python-Code und den verwendeten Modellen im digitalen Anhang einsehbar.

Algorithmus A.1: oberste Ebene des serialisierten Speckle-Objektes der Hauptwand, die die Tür enthält, mit ausgeklappten „elements“- und „materialQuantities“-Attributen


```
{
  "id": "085fc3c47d95e2c59a73ec8a43249f4f",
  "type": "Ziegel+WD hart 300+160",
  "level": {...},
  "units": "m",
  "family": "Basiswand",
  "height": 3.0000000000000004,
  "flipped": false,
  "baseLine": {...},
  "category": "Wände",
  "elements": [
    {
      "referencedId": "d034e0416987ea4a44d31b18dbffa7a9",
      "speckle_type": "reference"
    }
  ],
  "topLevel": {...},
  "__closure": {...},
  "elementId": "2509202",
  "topOffset": 0,
  "worksetId": "0",
  "baseOffset": 0,
  "parameters": {...},
  "structural": false,
  "displayValue": {...},
  "phaseCreated": "Phase 01",
  "speckle_type": "Objects.BuiltElements.Wall:Objects.BuiltElements.Revit.
      RevitWall",
  "applicationId": "b1501889-ad53-4335-9882-80bbca2b0580-00264992",
  "builtInCategory": "OST_Walls",
  "isRevitLinkedModel": false,
  "materialQuantities": [
    {
      "id": "8c09c9af97a3af27600ae1c53c98e4b6",
      "area": 22.11052500000001,
      "units": "m",
      "length": 7.999999999999999,
```

```
        "volume": 6.633157500000001,
        "material": {
          "referencedId": "a66b8fd0618bd240f7c70d4011dbd50a",
          "speckle_type": "reference"
        },
        "speckle_type": "Objects.Other.MaterialQuantity",
        "applicationId": null,
        "totalChildrenCount": 0
      },
      {
        "id": "b1fc463334c3a63fcc9047d94adf3535",
        "area": 22.11052500000001,
        "units": "m",
        "length": 7.999999999999999,
        "volume": 3.5376840000000023,
        "material": {
          "referencedId": "dcc2a520248e72c34eb6b989e55ab49b",
          "speckle_type": "reference"
        },
        "speckle_type": "Objects.Other.MaterialQuantity",
        "applicationId": null,
        "totalChildrenCount": 0
      }
    ],
    "totalChildrenCount": 22,
    "revitLinkedModelPath": "...\\Test_Wand_03.rvt"
}
```

Algorithmus A.2: oberste Ebene des serialisierten Speckle-Objektes, das auf Grundlage der Tür an ihrer ursprünglichen Position im Revit-Modell erzeugt worden ist, mit ausgeklapptem „transform“-Attribut


```
{
  "id": "90f3d1d5c92a391f63d6f238ad1fc6e7",
  "level": {...},
  "units": "m",
  "category": "Türen",
  "mirrored": false,
  "__closure": {...},
  "elementId": "2510287",
  "transform": {
    "id": "8e8f4f73d180d51b7325c80720ebb0db",
    "units": "m",
    "matrix": [
      -1,
      -1.615452956788033E-15,
      0,
      1.4424999999999937,
      1.615452956788033E-15,
      -1,
      0,
      0.22999999999999765,
      0,
      0,
      1,
      -1.1099388075308527E-14,
      0,
      0,
      0,
      1
    ],
    "speckle_type": "Objects.Other.Transform",
    "applicationId": null,
    "totalChildrenCount": 0
  },
  "worksetId": "0",
  "definition": {...},
  "parameters": {...},
  "handFlipped": true,
  "phaseCreated": "Phase 01",
  "speckle_type": "Objects.Other.Revit.RevitInstance",
  "applicationId": "eeccab4d-55f4-4d7f-b325-cccc573dd4ff-00264dcf",
  "facingFlipped": true,
  "renderMaterial": {...},
  "builtInCategory": "OST_Doors",
  "isRevitLinkedModel": false,
  "materialQuantities": {...},
  "totalChildrenCount": 13,
  "revitLinkedModelPath": "...\\Test_Wand_03.rvt"
}
```

Algorithmus A.3: oberste Ebene des serialisierten Speckle-Objektes, das auf Grundlage der verschobenen Tür im Revit-Modell erzeugt worden ist, mit ausgeklapptem „transform“-Attribut


```
{
  "id": "4c39e0b1c4e9b68f06eaf18a7a761058",
  "level": {...},
  "units": "m",
  "category": "Türen",
  "mirrored": false,
  "__closure": {...},
  "elementId": "2510287",
  "transform": {
    "id": "abd1d1f80571a4c2f5f68742cf940b17",
    "units": "m",
    "matrix": [
      -1,
      -1.615452956788033E-15,
      0,
      6.442499999999991,
      1.615452956788033E-15,
      -1,
      0,
      0.22999999999998955,
      0,
      0,
      1,
      -1.1099388075308527E-14,
      0,
      0,
      0,
      1
    ],
    "speckle_type": "Objects.Other.Transform",
    "applicationId": null,
    "totalChildrenCount": 0
  },
  "worksetId": "0",
  "definition": {...},
  "parameters": {...},
  "handFlipped": true,
  "phaseCreated": "Phase 01",
  "speckle_type": "Objects.Other.Revit.RevitInstance",
  "applicationId": "eeccab4d-55f4-4d7f-b325-cccc573dd4ff-00264dcf",
  "facingFlipped": true,
  "renderMaterial": {...},
  "builtInCategory": "OST_Doors",
  "isRevitLinkedModel": false,
  "materialQuantities": {...},
  "totalChildrenCount": 13,
  "revitLinkedModelPath": "...\\Test_Wand_03.rvt"
}
```

Algorithmus A.4: oberste Ebene des serialisierten Speckle-Objektes der ursprünglichen Hauptwand ohne Tür mit ausgeklapptem „baseLine“- und „materialQuantities“-Attribut


```
{
  "id": "3e76c70961dce638d7b2646c592f939e",
  "type": "Ziegel+WD hart 300+160",
  "level": {...},
  "units": "m",
  "family": "Basiswand",
  "height": 3.0000000000000004,
  "flipped": false,
  "baseLine": {
    "id": "51a42ae063f6a0be649bddcc4a0ef92d",
    "end": {
      "x": 7.999999999999998,
      "y": 0.22999999999998705,
      "z": 0,
      "id": "f38c3e3e610a6ffa92228d67fc4788df",
      "bbox": null,
      "units": "m",
      "speckle_type": "Objects.Geometry.Point",
      "applicationId": null,
      "totalChildrenCount": 0
    },
    "area": 0,
    "bbox": null,
    "start": {
      "x": 3.7146236030703003E-16,
      "y": 0.22999999999999995,
      "z": 0,
      "id": "d749598da240b1e650eea8f4f9cfb3e7",
      "bbox": null,
      "units": "m",
      "speckle_type": "Objects.Geometry.Point",
      "applicationId": null,
      "totalChildrenCount": 0
    },
    "units": "m",
    "domain": {
      "id": "3c11d8d90ada7ea233bc107e31637ab6",
      "end": 26.246719160104977,
      "start": -3.5544320787374045E-15,
      "speckle_type": "Objects.Primitive.Interval",
      "applicationId": null,
      "totalChildrenCount": 0
    },
    "length": 7.999999999999998,
    "speckle_type": "Objects.Geometry.Line",
    "applicationId": null,
    "totalChildrenCount": 0
  },
  "category": "Wände",
  "elements": null,
  "topLevel": {...},
```

```
  "__closure": {...},
  "elementId": "2509202",
  "topOffset": 0,
  "worksetId": "0",
  "baseOffset": 0,
  "parameters": {...},
  "structural": false,
  "displayValue": {...},
  "phaseCreated": "Phase 01",
  "speckle_type": "Objects.BuiltElements.Wall:Objects.BuiltElements.Revit.
      RevitWall",
  "applicationId": "b1501889-ad53-4335-9882-80bbca2b0580-00264992",
  "builtInCategory": "OST_Walls",
  "isRevitLinkedModel": false,
  "materialQuantities": [
    {
      "id": "7852d3a3f2230cf7b4505bfee8293b0f",
      "area": 24.000000000000004,
      "units": "m",
      "length": 7.999999999999998,
      "volume": 7.1999999999999975,
      "material": {
        "referencedId": "a66b8fd0618bd240f7c70d4011dbd50a",
        "speckle_type": "reference"
      },
      "speckle_type": "Objects.Other.MaterialQuantity",
      "applicationId": null,
      "totalChildrenCount": 0
    },
    {
      "id": "dce8bb85ee96ee33e0a1cf8ecbc4c0e5",
      "area": 24.000000000000004,
      "units": "m",
      "length": 7.999999999999998,
      "volume": 3.8400000000000003,
      "material": {
        "referencedId": "dcc2a520248e72c34eb6b989e55ab49b",
        "speckle_type": "reference"
      },
      "speckle_type": "Objects.Other.MaterialQuantity",
      "applicationId": null,
      "totalChildrenCount": 0
    }
  ],
  "totalChildrenCount": 8,
  "revitLinkedModelPath": "...\\Test_Wand_03.rvt"
}
```

Algorithmus A.5: oberste Ebene des serialisierten Speckle-Objektes der verkürzten Hauptwand mit ausgeklapptem „baseLine“- und „materialQuantities“-Attribut


```
{
  "id": "fd4d8bb74427d5e10792f0b3b80bcb47",
  "type": "Ziegel+WD hart 300+160",
  "level": {...},
  "units": "m",
  "family": "Basiswand",
  "height": 3.0000000000000004,
  "flipped": false,
  "baseLine": {
    "id": "fb6a2b7ca025288042d0b61ecb81dd61",
    "end": {
      "x": 5.999999999999999,
      "y": 0.22999999999999024,
      "z": 0,
      "id": "b90c38c0f9a3025b40eba7ed4014c03a",
      "bbox": null,
      "units": "m",
      "speckle_type": "Objects.Geometry.Point",
      "applicationId": null,
      "totalChildrenCount": 0
    },
    "area": 0,
    "bbox": null,
    "start": {
      "x": 1.9999999999999991,
      "y": 0.2299999999999967,
      "z": 0,
      "id": "58447de4330d0a67db0da66b4f7ee382",
      "bbox": null,
      "units": "m",
      "speckle_type": "Objects.Geometry.Point",
      "applicationId": null,
      "totalChildrenCount": 0
    },
    "units": "m",
    "domain": {
      "id": "28899f679a999ec73972f940c4ef8238",
      "end": 19.685039370078737,
      "start": 6.5616797900262425,
      "speckle_type": "Objects.Primitive.Interval",
      "applicationId": null,
      "totalChildrenCount": 0
    },
    "length": 4,
    "speckle_type": "Objects.Geometry.Line",
    "applicationId": null,
    "totalChildrenCount": 0
  },
  "category": "Wände",
  "elements": null,
  "topLevel": {...},
```

```
  "__closure": {...},
  "elementId": "2509202",
  "topOffset": 0,
  "worksetId": "0",
  "baseOffset": 0,
  "parameters": {...},
  "structural": false,
  "displayValue": {...},
  "phaseCreated": "Phase 01",
  "speckle_type": "Objects.BuiltElements.Wall:Objects.BuiltElements.Revit.
      RevitWall",
  "applicationId": "b1501889-ad53-4335-9882-80bbca2b0580-00264992",
  "builtInCategory": "OST_Walls",
  "isRevitLinkedModel": false,
  "materialQuantities": [
    {
      "id": "4dc4dd58b7664de55b649d64501de7bf",
      "area": 12.000000000000002,
      "units": "m",
      "length": 4,
      "volume": 3.5999999999999988,
      "material": {
        "referencedId": "a66b8fd0618bd240f7c70d4011dbd50a",
        "speckle_type": "reference"
      },
      "speckle_type": "Objects.Other.MaterialQuantity",
      "applicationId": null,
      "totalChildrenCount": 0
    },
    {
      "id": "a3e7772811f87224fabccfa7468a2f7b",
      "area": 12.000000000000002,
      "units": "m",
      "length": 4,
      "volume": 1.9200000000000002,
      "material": {
        "referencedId": "dcc2a520248e72c34eb6b989e55ab49b",
        "speckle_type": "reference"
      },
      "speckle_type": "Objects.Other.MaterialQuantity",
      "applicationId": null,
      "totalChildrenCount": 0
    }
  ],
  "totalChildrenCount": 8,
  "revitLinkedModelPath": "...\\Test_Wand_03.rvt"
}
```

Algorithmus A.6: oberste Ebene des serialisierten Hauptwandquaders mit ausgeklapptem „bbox“-Attribut


```
{
  "id": "6d040755ee3d6f6dbe16879e6a96c5a4",
  "area": 58.120000000000005,
  "bbox": {
    "id": "7b293a979876d842ee99c8f83d7f4a09",
    "area": 0,
    "bbox": null,
    "units": "m",
    "xSize": {
      "id": "bca5cfc12f3189cc37c71e533f1b511b",
      "end": 8,
      "start": 0,
      "speckle_type": "Objects.Primitive.Interval",
      "applicationId": null,
      "totalChildrenCount": 0
    },
    "ySize": {
      "id": "4dfcd8a6bd301b0b6949053d41d1cdc3",
      "end": 0.46,
      "start": 0,
      "speckle_type": "Objects.Primitive.Interval",
      "applicationId": null,
      "totalChildrenCount": 0
    },
    "zSize": {
      "id": "c98d1c77dd5043fe18ae73a11eb311e9",
      "end": 3,
      "start": 0,
      "speckle_type": "Objects.Primitive.Interval",
      "applicationId": null,
      "totalChildrenCount": 0
    },
    "volume": 11.040000000000001,
    "basePlane": {...},
    "speckle_type": "Objects.Geometry.Box",
    "applicationId": null,
    "totalChildrenCount": 0
  },
  "faces": {...},
  "units": "m",
  "colors": [],
  "volume": 11.040000000000001,
  "vertices": {...},
  "__closure": {...},
  "displayStyle": {...},
  "speckle_type": "Objects.Geometry.Mesh",
  "applicationId": "5ec0d8feae70d42c2456b225e21d6a87-2B6",
  "textureCoordinates": [],
  "totalChildrenCount": 2
}
```

Algorithmus A.7: oberste Ebene des serialisierten Türquaders mit ausgeklapptem „bbox“-Attribut


```
{
  "id": "2708d64d523755e601a9281b4f414983",
  "area": 6.55735,
  "bbox": {
    "id": "5ab8212c2718b01ed138ee6e28bf6f24",
    "area": 0,
    "bbox": null,
    "units": "m",
    "xSize": {
      "id": "1fff51d076ec48d49b7d2433863674a6",
      "end": 1.8849999999999998,
      "start": 0.9999999999999999,
      "speckle_type": "Objects.Primitive.Interval",
      "applicationId": null,
      "totalChildrenCount": 0
    },
    "ySize": {
      "id": "4dfcd8a6bd301b0b6949053d41d1cdc3",
      "end": 0.46,
      "start": 0,
      "speckle_type": "Objects.Primitive.Interval",
      "applicationId": null,
      "totalChildrenCount": 0
    },
    "zSize": {
      "id": "d098d3fee109929e4d7d1767a8c78d16",
      "end": 2.135,
      "start": 0,
      "speckle_type": "Objects.Primitive.Interval",
      "applicationId": null,
      "totalChildrenCount": 0
    },
    "volume": 0.8691584999999998,
    "basePlane": {...},
    "speckle_type": "Objects.Geometry.Box",
    "applicationId": null,
    "totalChildrenCount": 0
  },
  "faces": {...},
  "units": "m",
  "colors": [],
  "volume": 0.8691585,
  "vertices": {...},
  "__closure": {...},
  "displayStyle": {...},
  "speckle_type": "Objects.Geometry.Mesh",
  "applicationId": "5ec0d8feae70d42c2456b225e21d6a87-2D2",
  "textureCoordinates": [],
  "totalChildrenCount": 2
}
```

Literatur

AISH, R. (2000). Collaborative Design using Long Transactions and "Change Merge". *Proceedings of the 18th International Conference on Education and Research in Computer Aided Architectural Design in Europe (eCAADe)*, 107–111. https://doi.org/10.52842/conf.ecaade.2000.107

AMANN, J., ESSER, S., KRIJNEN, T., ABUALDENIEN, J., PREIDEL, C., & BORRMANN, A. (2021). BIM-Programmierschnittstellen. In A. BORRMANN, M. KÖNIG, C. KOCH & J. BEETZ (Hrsg.), *Building Information Modeling* (S. 263–290). Springer. https://doi.org/10.1007/978-3-658-33361-4_13

BIMvision. (2024). Zuletzt aufgerufen am 31. März 2024. https://bimvision.eu/de/

BORRMANN, A., & BERKHAHN, V. (2021). Grundlagen der geometrischen Modellierung. In A. BORRMANN, M. KÖNIG, C. KOCH & J. BEETZ (Hrsg.), *Building Information Modeling* (S. 35–51). Springer. https://doi.org/10.1007/978-3-658-33361-4_2

BORRMANN, A., KÖNIG, M., KOCH, C., & BEETZ, J. (2021). Die BIM-Methode im Überblick. In A. BORRMANN, M. KÖNIG, C. KOCH & J. BEETZ (Hrsg.), *Building Information Modeling* (S. 1–31). Springer. https://doi.org/10.1007/978-3-658-33361-4_1

BUNGARTZ, H.-J., GRIEBEL, M., & ZENGER, C. (2002). Geometrische Modellierung dreidimensionaler Objekte. In H.-J. BUNGARTZ, M. GRIEBEL & C. ZENGER (Hrsg.), *Einführung in die Computergraphik* (S. 55–112). Vieweg+Teubner Verlag. https://doi.org/10.1007/978-3-663-01614-4_2

C++ FAQ. (2024). Zuletzt aufgerufen am 2. April 2024. https://isocpp.org/wiki/faq/serialization

CHACON, S., & STRAUB, B. (2014). Pro Git: everything you need to know about Git (2. Aufl.). Apress.

EIRUND, H. (1993). Motivation und Einführung. In H. EIRUND (Hrsg.), *Objektorientierte Programmierung* (S. 11–28). Vieweg+Teubner Verlag. https://doi.org/10.1007/978-3-322-89217-1_1

ESSER, S., VILGERTSHOFER, S., & BORRMANN, A. (2022). Graph-based version control for asynchronous BIM collaboration. *Advanced Engineering Informatics*, *53*, 101664. https://doi.org/10.1016/j.aei.2022.101664

ESSER, S., VILGERTSHOFER, S., & BORRMANN, A. (2023). Version control for asynchronous BIM collaboration: Model merging through graph analysis and transformation. *Automation in Construction*, *155*, 105063. https://doi.org/10.1016/j.autcon.2023.105063

FIRMENICH, B., KOCH, C., RICHTER, T., & BEER, D. (2005). Versioning structured object sets using text based Version Control Systems. *Proceedings of the 22nd CIB-W78 Conference on Information Technology in Construction*, 105–112.

FreeFormatter.com. (2024). Zuletzt aufgerufen am 28. Januar 2024. https://www.freeformatter.com/json-formatter.html#before-output

GRAPHISOFT, BIMcloud. (2024). Zuletzt aufgerufen am 28. Februar 2024. https://graphisoft.com/solutions/bimcloud#workhybrid

HAERDER, T., & REUTER, A. (1983). Principles of transaction-oriented database recovery. *ACM Computing Surveys*, *15*(4), 287–317. https://doi.org/10.1145/289.291

ISO. (2018). ISO 19650-1:2018: Organisation und Digitalisierung von Informationen zu Bauwerken und Ingenieurleistungen, einschließlich Bauwerksinformationsmodellierung (BIM) - Informationsmanagement mit BIM - Teil 1: Begriffe und Grundsätze. https://www.iso.org/standard/68078.html

JERNIGAN, F. E. (2008). Big BIM little BIM: the practical approach to building information modeling: integrated practice done the right way! (2. Aufl.). 4Site Press.

jsondiff.com. (2023). Zuletzt aufgerufen am 28. Januar 2024. https://www.jsondiff.com/

MONDINO, D. (2021). BIM im architektonischen Entwurf. In A. BORRMANN, M. KÖNIG, C. KOCH & J. BEETZ (Hrsg.), *Building Information Modeling* (S. 381–391). Springer. https://doi.org/10.1007/978-3-658-33361-4_19

NBS. (2023). Zuletzt aufgerufen am 28. März 2024. https://www.thenbs.com/digital-construction-report-2023/

NEUBERG, F. (2004). Ein Softwarekonzept zur Internet-basierten Simulation des Ressourcenbedarfs von Bauwerken (Diss.). Technische Universität München. https://mediatum.ub.tum.de/601065

Nordic APIs. (2023). Zuletzt aufgerufen am 7. April 2024. https://nordicapis.com/what-is-the-difference-between-an-api-and-an-sdk/

OH, M., LEE, J., HONG, S. W., & JEONG, Y. (2015). Integrated system for BIM-based collaborative design. *Automation in Construction*, *58*, 196–206. https://doi.org/10.1016/j.autcon.2015.07.015

PETZOLD, F., & RECHENBERG, B. (2021). BIM und Bauen im Bestand. In A. BORRMANN, M. KÖNIG, C. KOCH & J. BEETZ (Hrsg.), *Building Information Modeling* (S. 507–532). Springer. https://doi.org/10.1007/978-3-658-33361-4_26

POINET, P., STEFANESCU, D., & PAPADONIKOLAKI, E. (2020). Collaborative Workflows and Version Control Through Open-Source and Distributed Common Data Environment. *Proceedings of the 18th International Conference on Computing in Civil and Building Engineering*, 228–247.

PREIDEL, C., BORRMANN, A., EXNER, H., & KÖNIG, M. (2021). Common Data Environment. In A. BORRMANN, M. KÖNIG, C. KOCH & J. BEETZ (Hrsg.), *Building Information Modeling* (S. 335–351). Springer. https://doi.org/10.1007/978-3-658-33361-4_16

Revit. (2024). Zuletzt aufgerufen am 31. März 2024. https://www.autodesk.de/products/revit/overview?term=1-YEAR&tab=subscription

Revit API Docs. (2024). Zuletzt aufgerufen am 31. März 2024. https://www.revitapidocs.com/

RevitDBExplorer. (2024). Zuletzt aufgerufen am 31. März 2024. https://github.com/NeVeSpl/RevitDBExplorer

ROMBERG, R. (2005). Gebäudemodell-basierte Strukturanalyse im Bauwesen (Diss.). Technische Universität München. https://mediatum.ub.tum.de/601078

SCHÄFERHOFF, G., JÄPPELT, U., & BLUME, T. (2021). BIM im Verkehrswasserbau. In A. BORRMANN, M. KÖNIG, C. KOCH & J. BEETZ (Hrsg.), *Building Information Modeling* (S. 687–700). Springer. https://doi.org/10.1007/978-3-658-33361-4_35

SCHAPKE, S.-E., BEETZ, J., KÖNIG, M., KOCH, C., & BORRMANN, A. (2021). Prinzipien und Techniken der modellgestützten Zusammenarbeit. In A. BORRMANN, M. KÖNIG, C. KOCH & J. BEETZ (Hrsg.), *Building Information Modeling* (S. 309–333). Springer. https://doi.org/10.1007/978-3-658-33361-4_15

SCHNEIDER, M. (2004). Transaktionen und Concurrency Control. In M. SCHNEIDER (Hrsg.), *Implementierungskonzepte für Datenbanksysteme* (S. 133–192). Springer. https://doi.org/10.1007/978-3-642-55888-7_5

Speckle. (2024). Zuletzt aufgerufen am 1. Februar 2024. https://speckle.systems/

Speckle Developer Docs. (2024). Zuletzt aufgerufen am 1. Februar 2024. https://speckle.guide/dev/

Speckle User Guide. (2024). Zuletzt aufgerufen am 1. Februar 2024. https://speckle.guide/

VILGERTSHOFER, S. (2022). Kopplung von Graphersetzung und parametrischer Modellierung zur Unterstützung des modellbasierten Entwerfens und der Erstellung mehrskaliger Modelle (Diss.). Technische Universität München. https://mediatum.ub.tum.de/1687268

ZHU, J., WU, P., & LEI, X. (2023). IFC-graph for facilitating building information access and query. *Automation in Construction*, *148*, 104778. https://doi.org/10.1016/j.autcon.2023.104778