HOCHSCHULE FURTWANGEN UNIVERSITY HFU

Bachelorthesis

im Studiengang

Medieninformatik Bachelor

Umsetzung einer cross-plattformen Desktopapplikation mithilfe des Frameworks Electron

Referent:	Prof. Jirka Dell'Oro-Friedl
Korreferent:	Prof. Dr. Dirk Eisenbiegler
Vorgelegt am:	20.08.2018
Vorgelegt von:	Robel Teklezghi
	Matrikelnummer: 250142
	Kreuzensteinstr. 16
	78224 Singen
	robel-teklezghi@web.de

Abstract

Diese Bachelorarbeit beschäftigt sich mit der Umsetzung einer plattformunabhängigen Desktopapplikation, die mithilfe des Frameworks Electron erstellt wird.

Hierfür werden zunächst einmal die Anforderungen an die fertige Anwendung festgelegt und die Funktionsweise von Electron beschrieben. Dann werden geeignete Webtechnologien für die Umsetzung gewählt und ihre Funktion erläutert. Die Software wird modular und objekt-orientiert implementiert.

Im Anschluss werden die einzelnen Komponenten der Anwendung und ihre Funktionen beschrieben.

Die Thematik ist vor allem für Interessenten aus der Webentwicklung, die mit den vorhandenen Kenntnissen auch Desktopanwendungen entwickeln wollen, von Bedeutung.

This bachelor thesis deals with the implementation of a cross platform desktop application built using the framework Electron.

First, the requirements for the finished application are defined and the mechanism behind Electron explained. Based on this, suitable web technologies are selected for implementation and their function is explained. The software is implemented modularly and object-oriented.

Afterwards, the individual components of the application and their functions are described.

The topic is especially important for those interested in web development, who want to develop desktop apps with their existing skill set.

Inhaltsverzeichnis

Abstract I
Inhaltsverzeichnis III
Abbildungsverzeichnis VII
Abkürzungsverzeichnis IX
1 Einleitung 1
2 Anforderungen 3
3 Umsetzung 5
3.1 Electron 5
3.1.1 Node.js 6
3.1.2 Chromium 6
3.1.3 Architektur 7
3.2 TypeScript 7
3.2.1 Das Design von TypeScript 8
3.2.2 Vorteile 9
3.2.3 Alternativen 11
3.3 Angular 11
3.3.1 NgModules 12
3.3.2 Components, Template, View 12
3.3.3 Services 13
3.3.4 MVC in Angular 13
3.3.5 Dependency Injection in Angular 14
3.3.6 Angular CLI 14
3.4 Frameworks 15
3.4.1 Babylon.js 15
3.4.2 Konva.js 16

3.4.3 ngx-Electron 16
3.4.4 electron-packager 16
3.5 Einrichtung der Entwicklungsumgebung 17
3.6 Komponenten 20
3.6.1 GlobalsService 21
3.6.2 CanvasComponent 22
3.6.3 HierarchyComponent 23
3.6.4 TransformComponent 24
3.6.5 MeshesComponent 24
3.6.6 Komponenten für die Erstellung von 3D-Objekten 25
3.6.7 MaterialComponent 26
3.6.8 PlayerComponent 26
3.6.9 EditorComponent 27
3.6.10 AnimationDisplayComponent 27
3.6.11 AnimCreateComponent 28
3.6.12 AnimTimelineComponent 29
3.7 Kommunikation 30
3.7.1 Kommunikation über das IPC-Modul 31
3.7.2 Kommunikation über Datenbindung 31
3.7.3 Kommunikation über Subjects 32
4 Zusammenfassung und Ausblick 35
Literaturverzeichnis 37
Eidesstattliche Erklärung 39
A. AnimCreateComponent 41
B. AnimationDisplayComponent 42
C. AnimTimelineComponent 43
D. PlayerComponent 46

E. CanvasComponent 47

F. HierarchyComponent 50

G. BoxComponent 52

H. MeshesComponent 53

I. MaterialComponent 54

J. TransformComponent 55

Abbildungsverzeichnis

Abbildung 1: Ordnerstruktur eines neuen Angular-Projekts 15
Abbildung 2: Bestandteile der fertigen Anwendung 20
Abbildung 3: Metadaten für die Erstellung eines Singletons 21
Abbildung 4: Klassendiagramm CanvasComponent 22
Abbildung 5: Klassendiagramm HierarchyComponent 23
Abbildung 6: Klassendiagramm TransformComponent 24
Abbildung 7: Klassendiagramm MeshesComponent 24
Abbildung 8: Klassendiagramm BoxComponent 25
Abbildung 9: Klassendiagramm MaterialComponent 26
Abbildung 10: Klassendiagramm PlayerComponent 26
Abbildung 11: Klassendiagramm AnimationDisplayComponent 27
Abbildung 12: Klassendiagramm AnimCreateComponent 28
Abbildung 13: Klassendiagramm AnimTimeLineComponent 29
Abbildung 14: Timeline der Anwendung 30
Abbildung 15: Kommunikation zwischen Main- und Renderer-Prozess 31
Abbildung 16: Datenbindung in Angular 32
Abbildung 17: Funktionsweise von Subjects 33
Abbildung 18: Kommunikation zwischen allen Komponenten 34

Abkürzungsverzeichnis

AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
CCM	Chromium Content Module
CLI	Command Line Interface
CSS	Cascading Style Sheets
DOM	Document Object Model
FPS	Frames per second
GUI	Graphical User Interface
HTML	Hypertext Markup Language
IPC	Interprocess communication
IOT	Internet of Things
LOC	Lines of Code
MVC	Model View Controller
SPA	Single-page application
WebGL	Web Graphics Library

1 Einleitung

Softwareentwicklung hat sich rasant verändert. Während zu Beginn der 2000er Software meist in Form einer Desktopanwendung verfügbar war, begann sich dies mit der Zeit zu verändern. Webbrowser wurden besser und die Qualität des Internets zusammen mit dessen Reichweite ausgebaut. Später führte die Einführung von AJAX (Asynchronous JavaScript and XML) zu einer neuen Ära, in der von Software als Webanwendungen geliefert wurden. Die Anwendungen liefen unabhängig vom Betriebssystem und mussten nicht mehr installiert werden. Dann kamen Unternehmen wie Google und Facebook und die Zahl der Internetnutzer stieg immer weiter. Dadurch gewannen das Web und damit auch die genutzten Webtechnologien immer mehr an Bedeutung [1]. Webtechnologien, und vor allem JavaScript, sind heute nicht mehr auf Webanwendungen beschränkt. Es gibt JavaScript-Frameworks für die Entwicklung von Mobile-Apps [2], Robotik und IoT (Internet of Things) [3], Desktopanwendungen und vieles mehr. Für dieses Phänomen hat Jeff Atwood, Mitbegründer der Webseiten Stack Overflow und Stack Exchange, Atwood's Law formuliert [4]:

„Any application that can be written in JavaScript, will eventually be written in JavaScript. "

Ein JavaScript-Framework für die Erstellung von Desktopapplikationen, dass zurzeit an Beliebtheit gewinnt, ist das Framework Electron von GitHub. Es ermöglicht mithilfe von HTML, CSS und JavaScript native Desktop-Apps für Windows, Mac und Linux zu erstellen. Es wurde unter anderem verwendet um die Desktopanwendungen Slack, WhatsApp, WordPress, Skype, Visual Studio Code, Atom und GitHub Desktop zu entwickeln. Die Zahl der dazukommenden Anwendungen steigt ständig weiter an. [5]

In dieser Arbeit wird ein 3D-Animationsprogramm mithilfe von Electron entwickelt. Dabei wird untersucht, wie der Entwicklungsprozess mit Electron aussieht und inwieweit es sich für diese Aufgabe eignet. Dafür werden zunächst

einmal die Anforderungen an die Anwendung festgelegt. Im nächsten Schritt werden dann Electron und die anderen verwendeten Technologien vorgestellt. Im Anschluss darauf wird schrittweise die Einrichtung der Entwicklungsumgebung gezeigt. Danach werden die Bestandteile der fertigen Anwendung einzeln erläutert. Im Abschnitt „Kommunikation“ werden anschließend die verwendeten Kommunikationswege vorgestellt. Abschließend wird das Ergebnis zusammengefasst und ein Ausblick auf die Weiterentwicklung der Anwendung gegeben.

2 Anforderungen

Das Ziel ist, eine auf Webtechnologien-basierte und plattformunabhängige 3D-Animationsprogramm umzusetzen, mit der einfache Animationen erstellt werden können. Die Anwendung soll wie eine native Desktopapplikation laufen und folgende Anforderungen erfüllen:

3D-Objekte

Die Anwendung soll eine Auswahl an dreidimensionalen Figuren bieten. Diese sollen anhand von eingegebenen Parametern erstellt werden. Zudem gibt es Kurzbefehle, über die bereits parametrisierte Standardobjekte in die Szene eingefügt werden. Die Figuren sollen nach der Erstellung transformierbar sein. Hierfür sollen zwei Möglichkeiten zur Verfügung stehen:

1. Es sind Eingabefelder vorhanden, anhand derer die Transformation durchgeführt wird.
2. Die Figuren sind mithilfe von Gizmos innerhalb der Szene transformierbar.

Objekte sind nach der Transformation anhand eines Kurzbefehls wieder in den ursprünglichen Zustand zurücksetzbar. Nach dem Auswählen/Anklicken einer Figur in der Szene soll dieser für den Nutzer hervorgehoben werden. Es ist möglich ein Material zu erstellen und diesen auf ein Objekt anzuwenden.

Animation

Die Anwendung soll Schlüsselbild-Animationen erstellen, bearbeiten und abspielen können. Animiert wird die Transformation der 3D-Objekte. Die Wiedergabe-Bildfrequenz (FPS) der Animation ist einstellbar. Für das Abspielen der Animation sollen die üblichen Steuerelemente zur Verfügung stehen. Eine Animation kann gestartet, pausiert, fortgesetzt, gestoppt und zurückge-

setzt werden. Die Zeitspanne der Wiedergabe soll einstellbar sein. Für die Erstellung und Bearbeitung der Schlüsselbilder soll eine Timeline benutzt werden. Darauf sind neben den Schlüsselbildern auch deren Werte zu sehen. Animationen sollen zudem wieder löschbar sein.

Szenenhierarchie

Damit der Nutzer den Überblick nicht verliert, soll in der Anwendung die Szenenhierarchie anschaulich dargestellt werden. Über dieser Darstellung können zudem 3D-Objekte gruppiert werden.

Speichern und Laden

Der Nutzer soll die Möglichkeit haben seinen Arbeitsstand zu speichern und laden. Der Zielordner soll dabei frei wählbar sein.

Menü

Die Anwendung soll ein natives Menü besitzen, in der die wichtigsten Funktionen der Anwendung enthalten sind. Außerdem sollen diese über Tastenkürzel aufrufbar sein.

3 Umsetzung

Die Anwendung soll, wie in dem Titel der Arbeit formuliert, mit Electron erstellt werden. Da es sich letztendlich um eine Webseite handelt, können alle Webtechnologien, die für die Erstellung einer Webanwendung genutzt werden, hier angewendet werden. In dieser Arbeit wird die Anwendung aufgrund der im Abschnitt 3.2.2 genannten Vorteile in TypeScript geschrieben. Für die Gestaltung der Webseite wird das in Abschnitt 3.3 aufgeführte Webframework Angular verwendet. Es wurde anstelle der populären JavaScript-Bibliothek React gewählt, weil es zum einen bereits mit TypeScript arbeitet und zum anderen ein vollständiges Framework mit Lösungen für fast alle Aufgaben ist. Auf diese Weise werden die Zahl der benötigten Libraries, die verwaltet werden müssen, verringert. Die anderen verwendeten Frameworks werden im Abschnitt 3.4 vorgestellt.

3.1 Electron

Electron wurde 2013 unter den Namen Atom Shell von Cheng Zhao veröffentlicht. Bevor Cheng Zhao bei GitHub anfing, arbeitete er als Praktikant bei Intel an dem node-webkit-Projekt von Roger Wang. Node-webkit war eine Kombination aus Node.js und der Browser-Engine WebKit, die es möglich machte, vom Browser aus auf Node-Module zuzugreifen. Unter der Zusammenarbeit beider Leute wurde aus dem Node-Modul ein Framework zur Erstellung von Desktopapplikationen. Nach dem Ende seines Praktikums begann Cheng Zhao bei GitHub an dem Editor Atom zu arbeiten. Seine Aufgabe war es den Editor zu node-webkit zu migrieren. Nachdem Probleme auftauchten, wurde dieser Ansatz abgebrochen und stattdessen eine eigene native Shell, genannt Atom Shell, entwickelt. Die Shell, die WebKit und Node.js anders implementierte als node-webkit, wurde dann für die Entwicklung von Atom verwendet. Später erstellte Google einen eigenen Fork des WebKit namens Blink und die Node.js-Community begann sich aufzuspalten.

Dies führte dazu, dass node-webkit in NW.js und Atom Shell in Electron umbenannt wurde. Mit der Zeit wurde Electron, wie bereits in der Einleitung aufgeführt, populär und viele Anwendungen damit entwickelt [1].

Electron ist eine Kombination aus Node.js und Chromium. Dadurch ist es möglich jede Webanwendung, aber auch jede Node-Anwendung in Electron auszuführen [6].

3.1.1 Node.js

Node.js [7] ist eine asynchrone und ereignisgesteuerte JavaScript-Laufzeitumgebung zur Entwicklung von skalierbaren Netzwerkanwendungen. Node.js nutzt die JavaScript-Engine V8 von Google, welche JavaScript-Code in Maschinencode kompiliert. Dadurch ist Node.js sehr performant. Node.js stellt zudem APIs für den Zugriff auf das lokale Dateisystem, der Erstellung von Servern und das Laden von Modulen zur Verfügung. Ein weiterer Vorteil ist der Paketmanager npm [8], der bei der Installation mitgeliefert wird. Node.js hat nach eigenen Angaben das größte Ökosystem der Welt für quelloffene Bibliotheken. Somit lassen sich für fast alle Aufgabenstellungen Node-Module finden, die das Problem bereits gelöst haben oder die Arbeit erleichtern. [6, 9, 10]

3.1.2 Chromium

Chromium ist die quelloffene Version des Google Chromes. Der Großteil des Quellcodes wird von beiden geteilt. In Electron ist nicht der ganze Browser enthalten, sondern das Chromium Content Module (CCM). Dieses Modul ist der Kern des Webbrowsers. Es enthält unter anderem den HTML-Renderer Blink und die JavaScript-Engine V8. Das CCM ist für das Rendern von HTML und CSS, sowie der Ausführung von JavaScript notwendig. Es wird von Electron verwendet, um den Inhalt der Anwendung, der in Form einer Webseite vorliegt, darzustellen. [6, 11, 12]

3.1.3 Architektur

Eine Electron-Applikation hat zwei distinktive Prozesstypen: den Main-Prozess und den Renderer-Prozess. Jede Anwendung hat genau einen Main-Prozess, während es mehrere oder keinen Renderer-Prozess geben kann.

Der Main-Prozess wird beim Start der Anwendung ausgeführt. Es ist der Prozess, der das main-Skript in der package.json-Datei aufruft. Innerhalb dieses Prozesses werden die Aktivitäten, die auf Systemebene laufen, gesteuert. Dazu gehören u.a. der Lebenszyklus der Anwendung (starten, beenden, etc.), der Zugriff auf das Dateisystem, das Öffnen von nativen Dialogfenstern und das Erstellen von Menüs. Es ist zudem für die Erstellung und Steuerung von Fensterinstanzen und somit Renderer-Prozessen zuständig. Der Main-Prozess kann als ein Node.js-Prozess betrachtet werden.

Der Renderer-Prozess ist für die Darstellung der Anwendung zuständig und rendert HTML und CSS. Es hat Zugriff auf das DOM, die Web-API, Node.js-Modulen und einen Teil der Electron-API. Da der Renderer von Chromium verwendet wird, hat der Renderer-Prozess zudem Zugriff auf die Entwicklerwerkzeuge von Chrome. Jede Fensterinstanz hat einen eigenen Renderer-Prozess. Diese Renderer-Instanzen sind voneinander isoliert und können nur über den Main-Prozess kommunizieren. Auch das Durchführen von nativen GUI-Operationen ist im Renderer-Prozess nicht möglich, weil es mit Speicherlecks verbunden ist. [6, 5]

3.2 TypeScript

TypeScript [13] ist eine von Microsoft entwickelte Erweiterung von JavaScript. Ziel dieser Sprache ist das Erstellen von großangelegten Anwendungen mit JavaScript zu ermöglichen [14]. Microsoft hat beispielsweise das Azure Management Portal (1,2 Mio. Lines of Code (LOC)) und den Editor Visual Studio Code (300.000 LOC) in TypeScript geschrieben. Da TypeScript eine Erweiterung von JavaScript ist, sind alle JavaScript-Programme gleich-

zeitig auch TypeScript-Programme. Die Syntax von TypeScript kann in Bezug auf JavaScript in drei Bestandteile unterteilt werden:

- ECMAScript 5, auf der TypeScript basiert
- Neue ECMAScript-Features, die der Sprache hinzugefügt werden
- Zusätzliche Features, die nicht Teil des ECMAScript-Standards sind. z.B. Typ-Annotationen und ein Modulsystem

TypeScript besteht aus drei Komponenten: die Sprache, der Compiler und der Sprachservice. Die Sprache besteht aus der neuen Syntax, Schlüsselwörter und Typ-Annotationen. Der Compiler kompiliert den TypeScript-Code in JavaScript. Es zeigt Warnungen und Fehlermeldungen an, wenn es Probleme entdeckt. Zudem kann es zusätzliche Aufgaben ausführen, wie beispielsweise die JavaScript-Dateien zu einer einzigen Datei zusammenzufassen oder Source Maps generieren. Der Sprachservice stellt die Typinformation zur Verfügung. Dadurch kann es von Entwicklungswerkzeugen benutzt werden um den Entwicklungsprozess zu vereinfachen. Dazu gehören u.a. Autovervollständigung, Typ-Vorschlag und Autoformatierung. [15]

3.2.1 Das Design von TypeScript

TypeScript hat das Ziel den Entwicklungsprozess mit JavaScript zu verbessern. Es will die vorhandenen JavaScript-Praktiken unterstützen und auf die meisten vorhandenen JavaScript-Bibliotheken anwendbar sein. Dieses Ziel führt zu bestimmten Eigenschaften des Typ-Systems:

- Die Typ-Notationen des TypeScript-Programms werden nach der Kompilierung rückstandslos aus dem JavaScript-Code gelöscht
- Das Typ-System ist strukturell statt nominal (z.B. Java und C# sind nominal)
- Objekt-Typen in TypeScript können nicht nur ihre Mitglieder beschreiben, sondern auch die unterschiedlichen Wege, in der sie benutzt werden können. Dies kommt dadurch zustande, dass in JavaScript zwischen Objekten, Funktionen, Konstruktoren und Arrays der Wert

nicht unterschieden wird. Dadurch kann ein Objekt mehrere dieser Rollen haben.

- TypeScript kann den Typ herleiten und verlässt sich darauf um die Anzahl an Typnotationen, die der Programmierer explizit eingeben muss, zu minimieren.
- TypeScript ist ein Bespiel für ein graduelles Typ-System. Der Programmcode kann sowohl statische Typen als auch mit „any“ gekennzeichnete, dynamische Typen enthalten.
- In TypeScript ist „Downcasting“ möglich.
- TypeScript erlaubt unsichere Kovarianzen der Eigenschafts- und Parameter-Typen aufgrund von JavaScript-Schemas, die sich sonst nicht typisieren lassen.
- In JavaScript kann neben der üblichen Punktnotation auch über den Namen der Methode als String indiziert werden. In TypeScript ist dieses Verhalten vorhanden. Beispiel: Math.random() = Math[„random“]() [14]

3.2.2 Vorteile

TypeScript löst oder vereinfacht die folgenden Probleme von JavaScript.

Vererbung mit der Prototypenkette

Prototypenbasierte Programmierung, auch als klassenlose Objektorientierung bekannt, wird meist in interpretierte, dynamische Sprachen gefunden. JavaScript ist die bekannteste Sprache, die zu dieser Gruppe angehört. Klassenbasierte Programmierung ist jedoch viel weiterverbreitet und wird von den meisten Programmierer benutzt. Deshalb stellt TypeScript Klassen, Namensräume, Module und Interfaces zur Verfügung. Dies erlaubt Entwicklern mit ihrem vorhanden Wissen zu arbeiten, und TypeScript übernimmt die Umwandlungen in JavaScript. [15]

Dynamische Typisierung

JavaScript verwendet dynamische Typisierung. Deshalb wird während der Laufzeit der Datentyp umgewandelt, so dass Ausdrücke, die bei einer statischen Typisierung Fehler ausgeben, funktionieren. Dies hat zur Folge, dass Laufzeitfehler entstehen können, die nicht leicht festzustellen sind. TypeScript hilft bei diesem Problem, indem es während des Eintippens oder der Kompilierung Warnungen anzeigt. [15]

Management von Modulen

Die Kopplung von mehreren JavaScript-Dateien gestaltet sich nicht immer leicht. Skripte werden oft in der falschen Reihenfolge in das HTML-Dokument einfügt. Für diese Zwecke gibt es Modul-Loader. In TypeScript werden Modul-Loader standardmäßig benutzt und die erstellten Module können in ein gewünschtes Modulformat kompiliert werden. Der Code wird dabei nicht verändert. [15]

Sichtbarkeitsbereich

In JavaScript wird der Sichtbarkeitsbereich der Variablen nicht durch den Block, in der es definiert wurde, bestimmt. Es ist stattdessen in der Funktion, in der es definiert wurde sichtbar. Wenn die Variable außerhalb einer Funktion definiert wird, so ist es global sichtbar. Dies kann zu Problemen führen. Die Einführung von const und let helfen dabei dieses Problem einzugrenzen. TypeScript selbst erleichtert Sichtbarkeitsprobleme, indem es vor impliziten, globalen Variablen warnt. [15]

Fehlende Datentypen

Aufgrund der dynamischen Typisierung in JavaScript können Entwicklungswerkzeuge keine spezifische und im Kontext stehende Hilfe leisten. TypeScript hilft bei diesem Problem, indem es die Typinformation zur Verfügung stellt. [15]

3.2.3 Alternativen

Neben TypeScript gibt es noch Alternativen, die das Schreiben von einfachem JavaScript ersetzten. Am weitesten verbreitet ist Babel, ein Compiler der die neusten ECMAScript-Features zur Verfügung stellt und für den Browser lauffähigen Code erstellt. CoffeeScript ist eine Alternative, die für bestimmte Jahre beliebt war. Es ist eine andere Sprache als JavaScript. Deshalb muss JavaScript-Code in CoffeeScript-Code übersetzt werden. Dart ist eine von Google herausgegebene Sprache. Es hat vieles mit TypeScript gemeinsam. Es ist klassenbasiert, objektorientiert und bietet eine statische Typ-Überprüfung. Ursprünglich wurde Dart als JavaScript-Ersatz entwickelt und die Kompilierung zu JavaScript war nur als Übergangslösung gedacht. Aber es hat sich nicht durchgesetzt, weshalb es wahrscheinlich die Kompilierung zu JavaScript weiterhin unterstützen wird. Neben diesen genannten Alternativen ist es auch möglich, Konverter zu benutzen, sodass von den meisten Programmiersprachen heraus JavaScript-Code erstellen werden kann. [15]

3.3 Angular

Angular ist ein quelloffenes und auf TypeScript-basiertes Frontend-Webapplikationsframework von Google. Es implementiert sowohl den Kern des Frameworks als auch die optionalen Funktionalitäten als TypeScript-Bibliotheken, die in die Anwendung importiert werden können. Es vereinfacht und beschleunigt die Entwicklung von reichhaltigen, komplexen Internetanwendung. Angular ist auf eine Komponenten-basierte Architektur ausgerichtet und arbeitet zudem auch mit dem Model-View-Controller (MVC) Entwurfsmuster. Dadurch werden folgende Eigenschaften besonders betont: [17, 18]

- Erweiterbar
- Wartbar
- Testbar
- Standardisiert

3.3.1 NgModules

Angular-Applikationen sind modular aufgebaut und besitzen ein eigenes Modularität-System namens NgModules. NgModules sind Klassen, die mit @NgModule gekennzeichnet werden. Sie sind die elementaren Bauteile einer Angular-Anwendung und dienen als Container für zusammengehörendes Code. Sie enthalten components, service providers und Code-Dateien, deren Gültigkeitsbereich durch den Container definiert wird. Funktionalität kann zwischen NgModules durch importieren und exportieren ausgetauscht werden. Jede Angular-Anwendung hat ein Stammmodul namens AppModule, das für das Bootstrapping der Anwendung benutzt wird. Das Erstellen von zusätzlichen Modulen, sogenannten feature modules, ist möglich. [17]

NgModules stellen den Kompilierungskontext ihrer components bereit. Ein Stammmodul hat immer eine Stammkomponente, die während des Bootstrapping erzeugt wird und zusätzliche Komponenten laden kann. Die Angular-Bibliotheken sind auch immer NgModules. Die Metadaten von NgModules haben folgenden Aufbau: [17]

- Deklarieren der Komponenten, Direktiven und Pipes des Modules
- Exportieren die Funktionalität an andere Module
- Importieren die Funktionalität anderer Module
- Stellen Services bereit, die von den Anwendungskomponenten genutzt werden können

3.3.2 Components, Template, View

Jede Angular-Applikation hat mindestens eine Komponente, die als Stammkomponente bezeichnet wird. Diese Komponente verbindet die Komponentenhierarchie mit dem DOM. Jede Komponente definiert eine Klasse, welche die Anwendungslogik und -daten enthält. Es ist zudem mit einem HTML-Template verbunden. Dieses Template informiert Angular wie die Komponente gerendert wird. Die Syntax ist eine Mischung aus normalem HTML und spezieller Template-Syntax, welche abhängig ist von der Komponentenlogik und den Komponentendaten. Diese verändert das HTML-Dokument. Die Komponente definiert zusammen mit dem Template die View. Die Views

werden hierarchisch aufgebaut und können untereinander verschachtelt werden. Dabei ist es auch möglich, die Views von Komponenten anderer Module zu integrieren. Komponenten werden durch den @Components-Decorator identifiziert. Die Metadaten einer Komponente definieren u.a. den Selektor, der für die Referenz im HTML-Dokument und benötigt wird. Anhand dessen erkennt Angular, an welcher Stelle eine Instanz der Komponente erstellt werden muss. Die Metadaten beinhalten oder zeigen zudem auf das Template. Angular kontrolliert den Lebenszyklus der Komponenten und stellt Hook-Funktionen zur Verfügung, über die darauf zugegriffen werden kann. Im Folgenden werden die in dieser Arbeit benutzten Funktionen aufgelistet: [17]

- ngOnChanges(): reagiert auf Änderungen von Eingabewerten und wird vor ngOnInit aufgerufen
- ngOnInit(): wird nach dem Aufbau der Data-Binding einmalig aufgerufen
- ngOnDestroy(): wird vor dem Entfernen der Komponente aufgerufen

3.3.3 Services

Um Daten und Logik, die unabhängig von der View sind, zwischen den Komponenten zu teilen, werden Service-Klassen benutzt. Diese werden durch den @Injectable()-Decorator definiert. Eine Service-Klasse hat meist eine wohldefinierte, spezifische Aufgabe. In Angular wird zwischen Komponenten und Services unterschieden um die Modularität und Wiederverwendbarkeit des Codes zu verstärken. [16]

3.3.4 MVC in Angular

Das MVC-Muster wird schon seit den 1970er benutzt. Es hat sich zuerst in der serverseitigen Webentwicklung aufgrund von Werkzeug wie Ruby on Rails und ASP.NET MVC-Framework durchgesetzt. Heute wird es auch verwendet um komplexe clientseitige Webapplikationen zu erstellen. Die Prämisse dieses Musters ist „Separation of Concerns“. Das Datenmodell der

Anwendung soll von der Geschäfts- und Präsentationslogik getrennt werden. Auf die Webentwicklung übertragen bedeutet dies die Trennung in

- Daten
- Logik, die die Daten verarbeiten
- HTML-Elemente, die die Daten darstellen

Die drei Bauteile dieses Musters sind das Modell (model), die Präsentation (view) und die Steuerung (controller). Das Modell enthält die Daten und derer Logik. Auf Angular übertragen, übernehmen Services oft diese Aufgabe. Die Steuerung verwaltet sowohl das Modell als auch die Präsentation. In Angular wird diese Aufgabe von den Komponenten erfüllt. Die Präsentation ist für die Darstellung der Daten zuständig. Das entspricht in Angular den Templates. [17]

3.3.5 Dependency Injection in Angular

Dependency Injection ist ein Entwurfsmuster fürs Managen von Code-Abhängigkeiten. Diese sieht vor, dass Objekte ihre Abhängigkeiten von einer externen Instanz bekommen. Auf diese Art werden voneinander abhängige Objekte entkoppelt, und Änderungen an einem Objekt führen nicht zu einer Veränderung des anderen Objekts. In Angular wird es verwendet, um u.a. Services an die Komponenten zu übergeben. [16]

3.3.6 Angular CLI

Dieses Tool hilft beim Aufsetzen von Angular-Projekten und erleichtert den gesamten Entwicklungsprozess. Es richtet die Entwicklungsumgebung ein und kann u.a. Module, Komponenten und Services generieren. Nach dem Erstellen eines neuen Projekts sieht die Ordnerstruktur folgendermaßen aus:

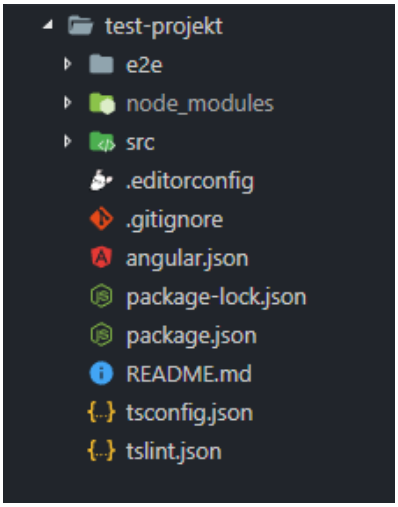


Abbildung 1: Ordnerstruktur eines neuen Angular-Projekts

Der e2e-Ordner wird für End-To-End-Tests genutzt. Im src-Ordner befinden sich die Dateien der Angular-Anwendung. In den node_modules-Ordner werden alle installierten Node-Module gespeichert. Nach dem build-Prozess kommt noch der dist-Ordner hinzu. In diesem wird die fertige und zusammengebaute Anwendung bereitgestellt. [16]

3.4 Frameworks

3.4.1 Babylon.js

Babylon.js ist ein JavaScript-Framework für das Erstellen von 3D-Anwendungen und 3D-Spielen basierend auf HTML, WebGL (Web Graphics Library), WebVR und Web Audio. Es ist ein Open Source Projekt, das von Microsoft unterstützt wird. Es beinhaltet unter anderem einen vollständigen Szenengraphen mit Lichter, Kameras, Materialien, Gitter, Animationen, Audio und Aktionen. Außerdem besitzt es noch eine Physik-Engine, Animation-Engine und Kollision-Engine. [18] Andere auf WebGL-basierte Alternativen sind Three.js, A-Frame, Whitestorm.js und PlayCanvas. In dieser Arbeit wird Babylon.js aufgrund der guten und hilfreichen Dokumentation sowie der vielen bereits implementierten Funktionen verwendet.

3.4.2 Konva.js

Konva.js ist ein HTML5-Canvas-JavaScript-Framework. Es ist als Fork von KineticJS, einem JavaScript-Framework für Canvas-Interaktivität, entstanden. Konva bietet u.a. Ebenen, Verschachtelungen, Animationen, Filter, Selektoren, Events, Kollisionserkennung, Caching und eine objektorientierte API. Es wird für die Gestaltung der Timeline verwendet, weil viele benötigte Funktionen bereits vorhanden sind und diese somit keine eigene Implementierung erfordern. [19]

3.4.3 ngx-Electron

ngx-electron ist ein Angular Wrapper für Electron. Es ermöglicht die Electron API zusammen mit Angular zu benutzen, ohne dabei die Typdefinitionen in @types/electron zu verlieren. Electron ist abhängig von der window.require()-Methode, welche nur während des Renderer-Prozesses zur Verfügung steht. Wenn die Typisierungen verwendet werden, um auf die API zuzugreifen, zeigt der TypeScript-Compiler Kompilierungsfehler. Dieses Package hilft dabei dieses Problem umzugehen. [20]

3.4.4 electron-packager

Nachdem der Quellcode einer Electron-Anwendung fertig geschrieben wurde, muss es noch zu einem ausführbaren Programm verpackt werden. electron-packager ist ein Werkzeug zum Verpacken und Bündeln von Electron-Anwendungen. Damit lassen sich ganz leicht für alle drei Plattformen ausführbare Anwendungen bauen. Die Syntax sieht wie gefolgt aus:

```
electron-packager <sourcedir> <appname> --platform=<platform> --arch=<arch> [optional flags...]
```

Alternativen zu diesem Werkzeug sind electron-forge und electron-builder. [21]

3.5 Einrichtung der Entwicklungsumgebung

Als Quelltext-Editor wird Visual Studio Code von Microsoft verwendet. Dieser Editor basiert auf Electron und ermöglicht u.a. Syntaxhervorhebung, Debugging, Autovervollständigung und Versionsverwaltung. Es hat zudem einen eingebauten Terminal, welcher das Ausführen von Kommandozeilen-Befehle direkt von dem Editor ermöglicht.

Die Einrichtung der Entwicklungsumgebung für Electron an sich ist einfach. Eine minimale Electron-Anwendung besteht aus einer package.json, einer main.js und einem index.html. Bei der Verwendung mit Angular müssen jedoch einige Punkte beachtet werden, damit alles reibungslos funktioniert. Im Folgenden werden die einzelnen Schritte gezeigt, die für die Einrichtung der Entwicklungsumgebung für diese Arbeit durchgeführt werden:

1. Zunächst einmal muss Node.js auf dem Rechner installiert sein.
2. Als nächstes wird über die Kommandozeile das Angular CLI Modul installiert:

   ```
   npm install -g @angular/cli@latest
   ```

3. Ein neues Angular-Projekt wird über den folgenden Befehl erstellt:

   ```
   ng new <Projektname>
   ```

   Die CLI erstellt dann die Projektstruktur und installiert die benötigten Module.
4. In src/index.html wird der Zugangspunkt von Angular geändert um Probleme mit Electron zu vermeiden:

   ```
   <base href = "./">
   ```

5. Nun werden die im vorherigen Kapitel aufgeführten Frameworks installiert:

   ```
   cd <Projektname>
   ```

   ```
   npm install babylonjs
   ```

```
npm install konva

npm install ngx-electron

npm install electron --save-dev

npm install electron-packager --save-dev
```

Bemerkung:
Zum Zeitpunkt dieser Arbeit wurde die Babylonjs-Version „3.3.0-beta.2“ benutzt, um auf die Gizmo-Features zugreifen zu können, die in der damaligen Version 3.2 nicht enthalten waren. Außerdem werden Electron und der electron-packager als devDependencies installiert, weil diese Module nur während der Entwicklung benötigt werden.

6. Jetzt wird ein Ordner unter src/ erstellt. In diesen Ordner kommen die main.ts von Electron sowie eine tsconfig-Datei, die diesen transpiliert. Als Vorlage für den Inhalt der main.ts eignet sich die electron-quick-start-typescript-Repository von Electron [22].
7. Fürs Transpilieren wird TypeScript global installiert. Dann wird die main.js über die folgenden Befehle erstellt:

```
cd <Projektname>/src/<Electronordner>

tsc
```

8. Damit Electron in das Projekt integriert wird, muss die package.json-Datei angepasst werden. Unter dem Punkt „main“ wird der Name der Electron main.js eingefügt. In diesem Fall sieht es so aus:

```
"main": "electron-main.js"
```

9. Dann werden noch unter dem Punkt „scripts“ Skripte hinzugefügt, die zum Bauen, Starten und Verpacken der Electron-Anwendung gebraucht werden:
    a. Baut die Seite zusammen und kopiert die package.json und electron-main.js-Dateien in den dist-Ordner:

```
"build-electron": "ng build --base-href . && xcopy src\\electron\\electron-main.js dist && xcopy package.json dist"
```

Dadurch befindet sich der Quellcode der fertigen Electron-Anwendung in einem Ordner, der verpackt werden kann.

b. Zum Starten der Electron-Anwendung:

```
"electron": "electron dist"
```

c. Verpackt den Quellcode zu einer auf der Zielplattform ausführbaren Anwendung zusammen:

```
"<Skriptname>": "electron-packager dist <Anwendungsname> --platform=<Betriebssystem> --out <Zielverzeichnis> --overwrite"
```

10. In der Electron main.ts-Datei muss noch der Pfad zu der index.html-Datei angepasst werden. Da die main.js-Datei später in den dist-Ordner kopiert wird, muss der Pfad abhängig von diesem Ordner erstellt werden:

```
mainWindow.loadFile(path.join(__dirname, "<Projektname>/index.html"));
```

3.6 Komponenten

Eine Electron-Anwendung besteht wie bereits in Sektion 3.1.3 aufgeführt aus einem Main-Prozess und Renderer-Prozessen. Dementsprechend findet bei der Erstellung der Anwendung eine Trennung in diese zwei statt. Für den Main-Prozess wird eine JavaScript-Datei benötigt, die als Einstiegspunkt in die Applikation dient. Diese Datei wird in der package.json unter dem Punkt „main“ verlinkt. In dieser Datei werden alle anwendungsspezifischen Menüpunkte erstellt. Außerdem werden hier die von der Anwendung benötigten Dateien erstellt oder geöffnet. Zudem wird es für das Anzeigen von Dialogfenstern verwendet. In dieser Arbeit wird nur ein Renderer-Prozess benötigt, weil die Webseite eine mit Angular erstellte SPA (Single Page Applikation) ist. In der nachfolgenden Abbildung werden die einzelnen Bestandteile, aus der die Anwendung besteht, dargestellt.

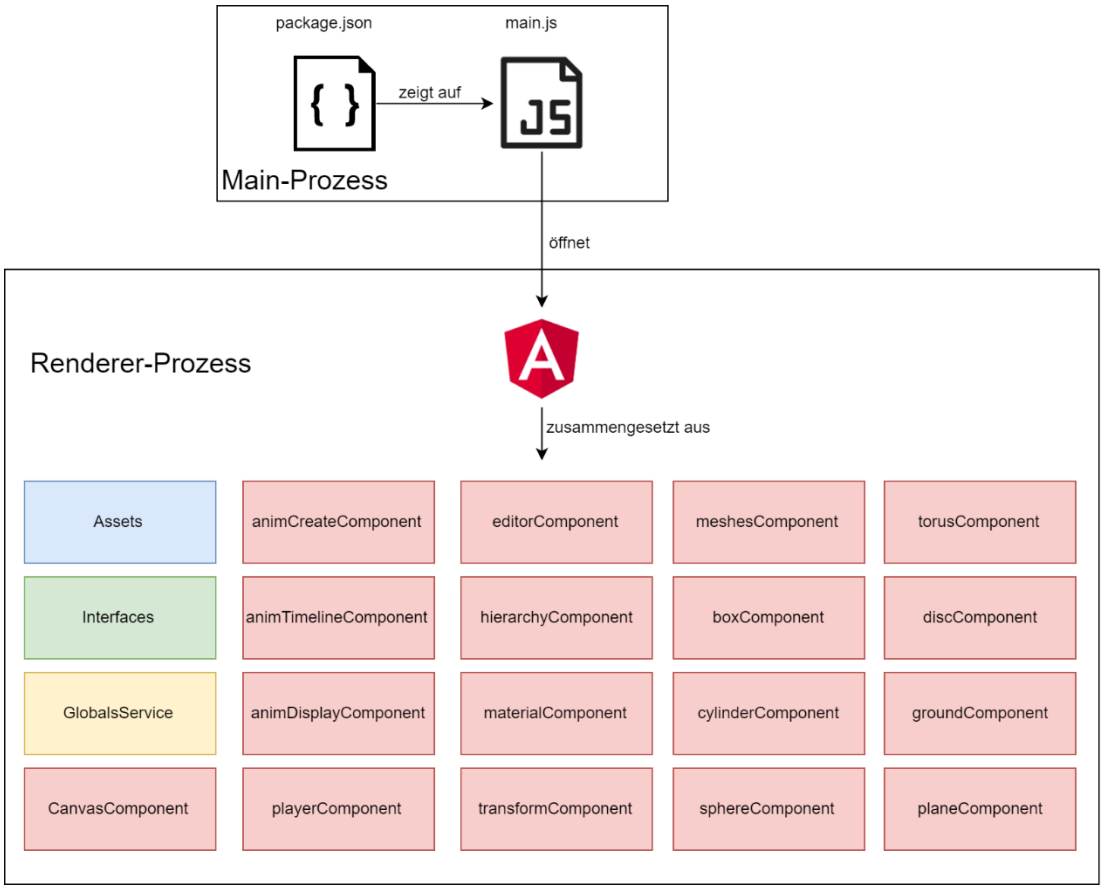


Abbildung 2: Bestandteile der fertigen Anwendung

3.6.1 GlobalsService

Für die Anwendung wird ein Service namens GlobalsService erstellt, der in allen Komponenten durch Dependency Injektion eingefügt wird. Um sicher zu gehen, dass alle Komponenten auf die gleiche Instanz dieses Dienstes zugreifen, wird dieser zu einem Singleton gemacht. Dafür wird der Service in dem Wurzelverzeichnis der Anwendung bereitgestellt, wodurch es allen Komponenten zur Verfügung steht. Dies wird durch das Einfügen von Metadaten zum Provider erreicht.

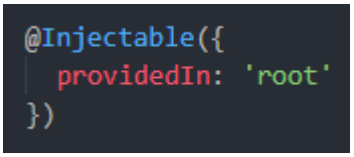


Abbildung 3: Metadaten für die Erstellung eines Singletons

Wenn der Service über der Angular CLI generiert wird, so wird dieser standardmäßig im „root“ bereitgestellt. Der GlobalsService enthält alle globalen Variablen der Anwendung. Diese sind

- Szene: Enthält die aktive Szene und ermöglicht allen Komponenten damit zu interagieren
- ipcRenderer: wird für die Kommunikation mit dem Main-Prozess benötigt
- selectedMesh: Subject zum Versenden oder Empfangen des ausgewählten Meshes
- selectedAnimation: Subject zum Versenden oder Empfangen der ausgewählten Animation
- sceneTree: Subject zum Versenden eines String-Arrays (siehe 3.6.3)

3.6.2 CanvasComponent

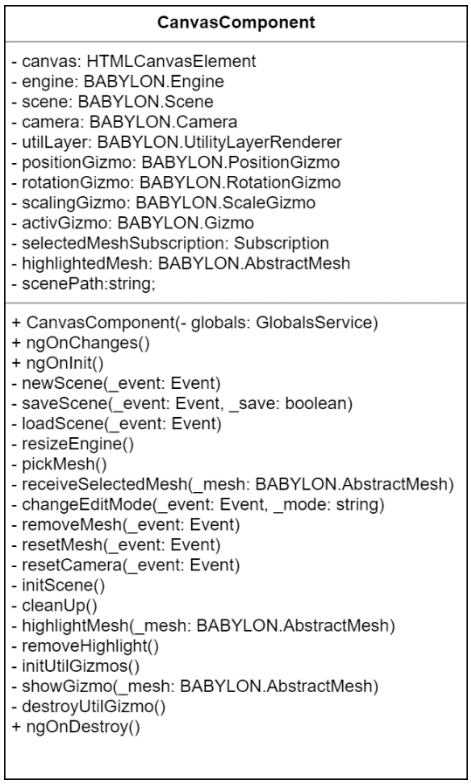


Abbildung 4: Klassendiagramm CanvasComponent

In der CanvasComponent wird der 3D-Raum dargestellt. Sie kann als die Hauptkomponente der Anwendung betrachtet werden. Damit Babylon.js die 3D-Szene rendert, benötigt sie ein Canvas-Element, in der die 3D-Engine geladen wird. Außerdem muss eine Kamera in der Szene vorhanden sein. Mithilfe der Kamera findet die Navigation durch den 3D-Raum statt. Babylon.js stellt zudem verschiedene Kameras zur Verfügung, die unterschiedliche Funktionen erfüllen. Für die Anwendung wird eine Arc-Rotate-Kamera verwendet, weil diese eine leichte Navigation durch die Szene ermöglicht. Die Methoden der CanvasComponent, die dem obigen Klassendiagramm zu entnehmen sind, können in zwei Kategorien unterteilt werden:

- Methoden, die eine Szene erstellen, speichern oder laden
- Methoden, die für die Interaktion mit der Szene da sind

3.6.3 HierarchyComponent

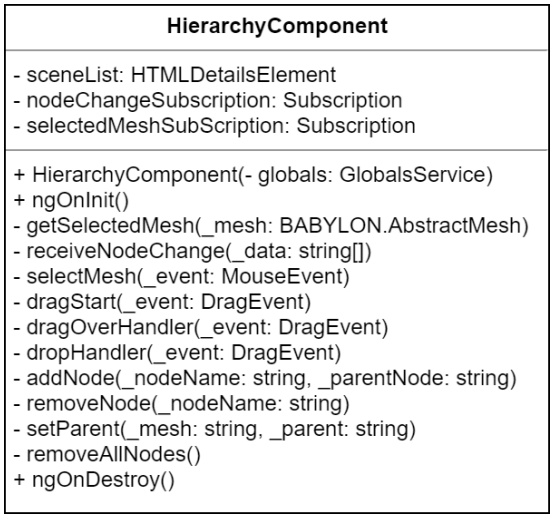


Abbildung 5: Klassendiagramm HierarchyComponent

Die HierarchyComponent ist für die Darstellung und Manipulation des Szenengraphen zuständig. Hier werden alle Objekte, die sich in der Szene befinden und mit denen interagiert werden kann, angezeigt. Durch das Anklicken des Namens kann ein Objekt in der Szene ausgewählt werden. Mithilfe von Drag & Drop können Objekte gruppiert werden. Da die Komponente über den GlobalsService Zugriff auf die aktive Szene bekommt, kann sie diese Veränderungen selber durchführen. Die Information über neu erstellte oder gelöschte Meshes bekommt sie über das sceneTree-Subject in Form eines Arrays. In diesem Array steht der Name des Meshes, sein Elternteil und ob es erstellt oder gelöscht wird.

3.6.4 TransformComponent

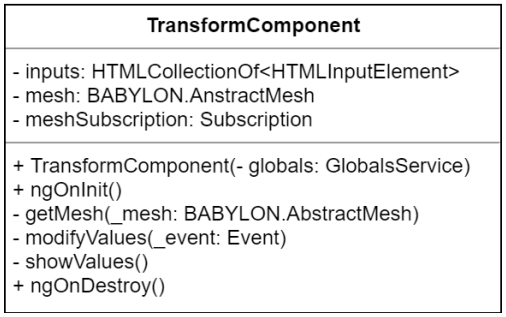


Abbildung 6: Klassendiagramm TransformComponent

Die TransformComponent ist für die Verschiebung, Rotation und Skalierung der Objekte zuständig. Da es in der CanvasComponent auch möglich ist, über Gizmos die Objekte zu transformieren, kann hier über Buttons zwischen den einzelnen Gizmos gewechselt werden.

3.6.5 MeshesComponent

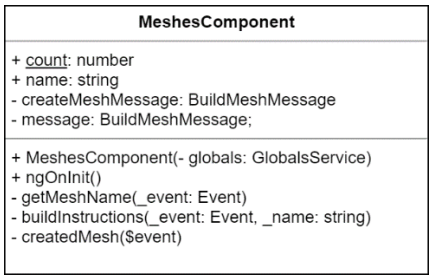


Abbildung 7: Klassendiagramm MeshesComponent

In der MeshesComponent befinden sich die Komponenten für die Erstellung der verschiedenen 3D-Objekte. Es wird zur Verwaltung dieser Komponenten benutzt. Über Events wird festgestellt, welche Komponente dargestellt werden soll, und diese wird dann angezeigt. Wenn die untergeordnete Komponente ein Objekt erstellt, wird die MeshesComponent informiert. Diese wiederum informiert dann die HierarchyComponent. Es besteht zudem die Mög-

lichkeit, über das Menü vorgefertigte Objekte zu erstellen. Dafür schickt die MeshesComponent zusätzlich noch eine fertige Bauanweisung an die gewünschte Komponente, die dann umgesetzt wird. Die MeshesComponent ist die einzige Komponente, in der alle drei Kommunikationswege benutzt werden.

3.6.6 Komponenten für die Erstellung von 3D-Objekten

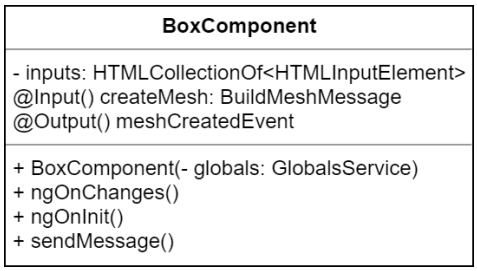


Abbildung 8: Klassendiagramm BoxComponent

Wie in dem Punkt davor erwähnt, gibt es für die Erstellung der 3D-Objekte jeweils eigene Komponenten. Diese stellen Optionen zur Verfügung, über die die Parameter der Objekte eingestellt werden. Sie haben alle den gleichen Aufbau und unterscheiden sich nur durch die Anzahl und Art der benötigten Parameter. Nach dem die Objekte erzeugt wurden, wird die MeshesComponent informiert.

3.6.7 MaterialComponent

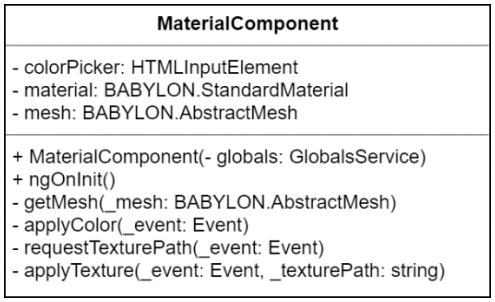


Abbildung 9: Klassendiagramm MaterialComponent

Die MaterialComponent ist für die Erstellung von Materialien zuständig. Diese werden den ausgewählten Objekten hinzugefügt. Es kann dabei zwischen Farb- und Texturmaterialien gewählt werden. Die Texturen werden über den ipcRenderer vom Main-Prozess angefordert. Bei der Erstellung von Materialien muss ein Name vergeben werden, damit diese auch nach dem Speichern der Szene wiedererkannt werden.

3.6.8 PlayerComponent

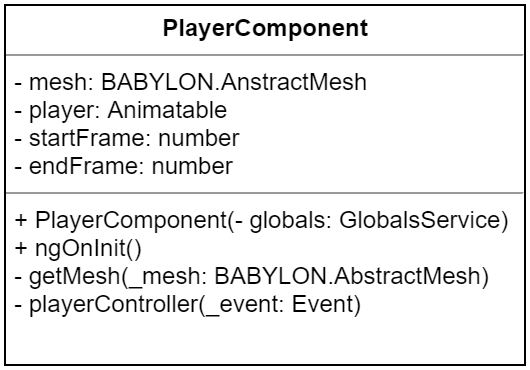


Abbildung 10: Klassendiagramm PlayerComponent

Die PlayerComponent ist zum Abspielen von Animationen da. Die verfügbaren Steuerungen sind abzuspielen, zu pausieren / fortzusetzen, zu stoppen

und zurückzusetzen. Zudem können die Start- und Endframes der Wiedergabe festgelegt werden.

3.6.9 EditorComponent

Die EditorComponent ist nur zur Einbindung und Gruppierung anderer Komponenten da. Folgende Komponenten sind darin enthalten: TransformComponent, MeshesComponent, MaterialComponent, PlayerComponent und AnimCreateComponent. Diese können in der Anwendung je nach Wunsch angezeigt oder minimiert werden.

3.6.10 AnimationDisplayComponent

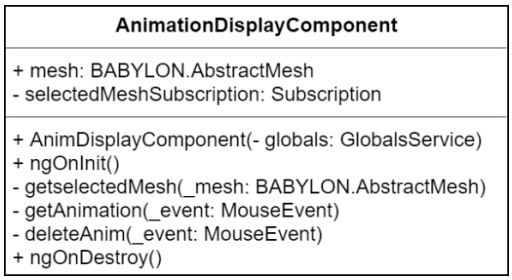


Abbildung 11: Klassendiagramm AnimationDisplayComponent

Ein Objekt kann in Babylon.js mehrere Animationen gleichzeitig haben. Damit der Überblick nicht verloren geht, wird die AnimationDisplayComponent benutzt um diese darzustellen. Durch das Aufklappen der Animation können zusätzliche Informationen betrachtet werden. Hier wird auch die Animation zur Bearbeitung in der Timeline ausgewählt und über das selectedAnimation-Subject an die AnimTimelineComponent gesendet. Außerdem können Animationen gelöscht werden.

3.6.11 AnimCreateComponent

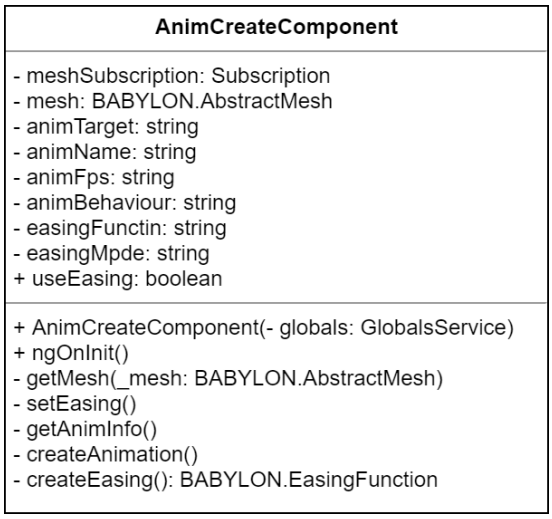


Abbildung 12: Klassendiagramm AnimCreateComponent

Die AnimCreateComponent ist für die Erstellung von Animationen zuständig. Um eine Animation zu erstellen, werden folgende Parameter eingegeben:

- Name der Animation
- Die Eigenschaft, die animiert werden soll (momentan kann zwischen Position, Rotation und Skalierung gewählt werden)
- Framerate
- Loop-Mode
- (optional) Übergangsfunktionen: beschreiben Animationsverläufe. Wenn keine ausgewählt wird, verläuft die Animation linear

Damit wird eine leere Animation erstellt und dem ausgewählten Objekt zugeordnet. Um diese Animation mit Schlüsselbilder zu füllen, muss es in der Timeline bearbeitet werden.

3.6.12 AnimTimelineComponent

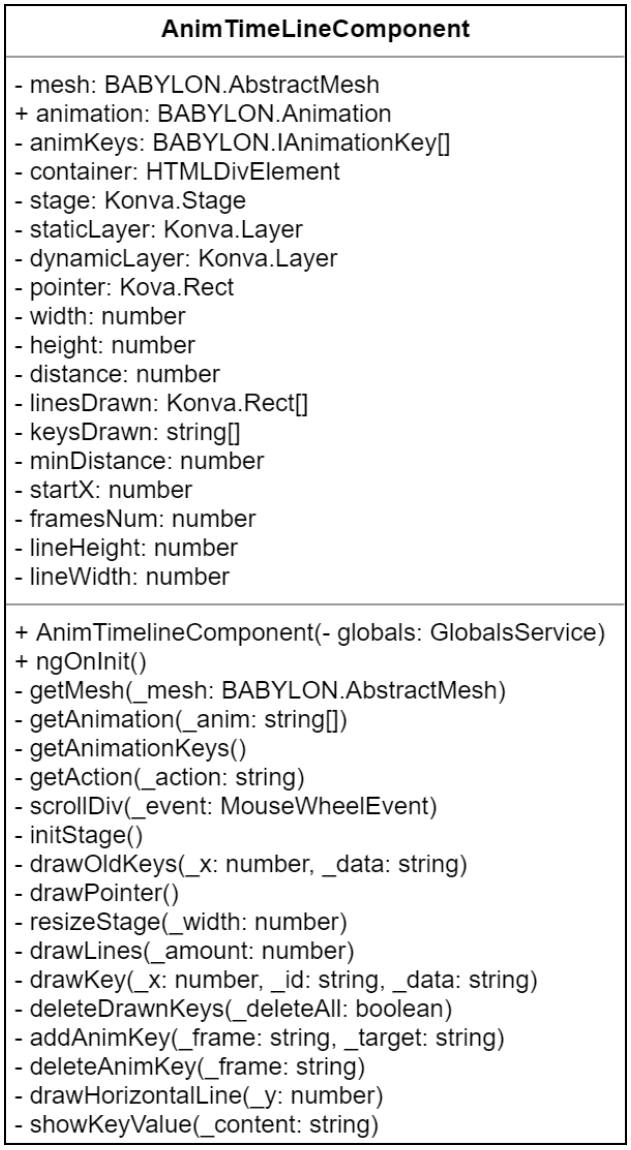


Abbildung 13: Klassendiagramm AnimTimeLineComponent

Die AnimTimelineComponent erstellt die Timeline der Anwendung und bearbeitet darin die Schlüsselbilder (Keyframe) der Animationen. Die Timeline besteht aus einem statischen und einem dynamischen Layer. In dem statischen Layer werden alle Linien sowie die Ziffern der Timeline gezeichnet. In dem dynamischen Layer werden die Symbole für die Schlüsselbilder und der rote Zeiger gezeichnet. Dadurch wird gewährleistet, dass nur die notwendigen Elemente wiederholt gezeichnet werden.

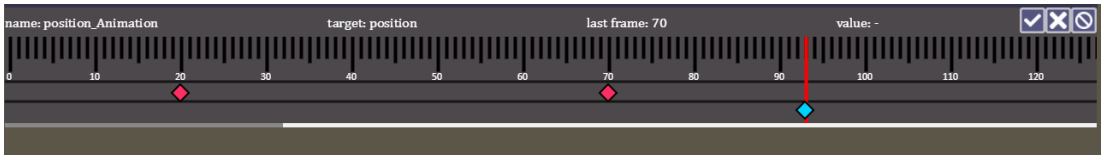


Abbildung 14: Timeline der Anwendung

Oberhalb der Timeline befindet sich die Infoleiste mit Informationen zu der ausgewählten Animation und drei Buttons. Der linke Button bewirkt, dass neu erstellte Keys der Animation hinzugefügt werden. Der mittlere Button setzt neu erstellte, aber der Animation noch nicht hinzugefügte Keys zurück, während der rechte Button die Bearbeitung der ausgewählten Animation abbricht. Der rote Zeiger wandert mit der Maus über die Timeline und wird für die Auswahl eines Frames benutzt. Die Schlüsselbilder werden durch rote Rauten für alte und blaue für neue dargestellt. Wenn mit der Maus über diese Rauten bewegt wird, kann in der Infoleiste deren Wert betrachtet werden. Zum Erstellen eines Schlüsselbilds wird auf den Zeiger geklickt. Der neue Keyframe befindet sich dann an dessen Position. Gelöscht werden Keyframe, indem auf dem gewünschten Keyframe geklickt wird. Soll ein alter Keyframe der Animation ersetzt werden, so kann am selben Frame eine neuer erstellt werden. Nach der Übernahme der Veränderungen wird dann das neue Schlüsselbild angezeigt.

3.7 Kommunikation

Die Anwendung ist aus vielen verschiedenen Komponenten zusammengesetzt. Das hat den Vorteil, dass die Anwendung leicht modifiziert oder erweitert werden kann. Aber dadurch muss die Kommunikation zwischen den Komponenten gut geregelt sein. In dieser Arbeit werden drei Kommunikationswege verwendet. Im Folgenden werden diese näher erläutert.

3.7.1 Kommunikation über das IPC-Modul

Für die synchrone und asynchrone Kommunikation zwischen dem Main- und Renderer-Prozess stellt Electron das IPC-Modul zur Verfügung. Dieses Modul besteht aus ipcRenderer und ipcMain. Über den ipcRenderer kann der Renderer-Prozess Events empfangen und senden. Auf der anderen Seite werden über den ipcMain Events im Main-Prozess empfangen und beantwortet. Soll die Kommunikation vom Main-Prozess initiiert werden, so läuft diese über den webContents-EventEmitter anstelle des ipcMain.

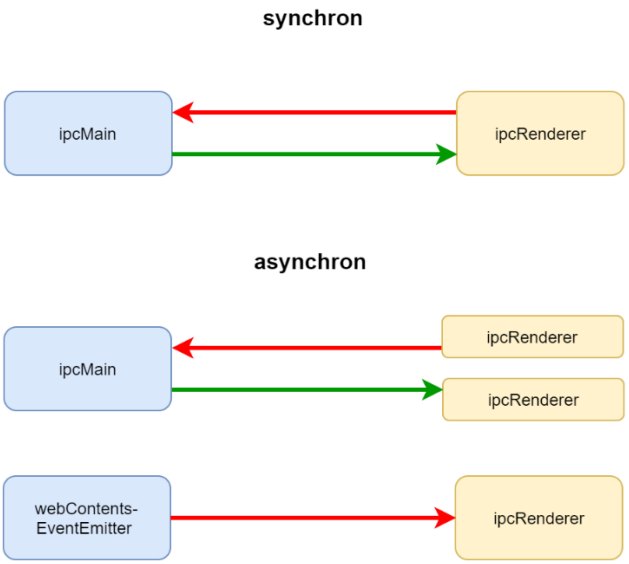


Abbildung 15: Kommunikation zwischen Main- und Renderer-Prozess

3.7.2 Kommunikation über Datenbindung

In Angular werden Property- und Event-Bindings nicht nur für den Informationsaustausch zwischen Komponenten und deren Templates benutzt, sondern auch für die Kommunikation zwischen einem Komponenten und dessen untergeordneten Komponenten. Über Property-Binding werden Daten an die Child-Komponenten übergeben. Über den @Input-Decorator kann die Child-Komponente dann darauf zugreifen. In die umgekehrte Richtung werden Daten über Event-Binding an die übergeordnete Komponente übergeben. Dafür wird das Event mit dem @Output-Decorator gekennzeichnet. Wird das Event dann versendet, kann es von der Parent-Komponente empfangen werden.

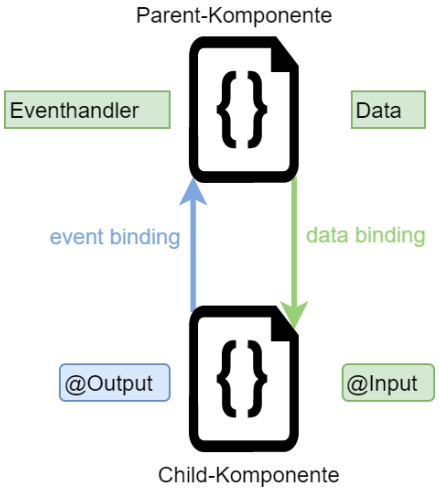


Abbildung 16: Datenbindung in Angular

3.7.3 Kommunikation über Subjects

Angular verwendet die RxJS-Bibliothek, die das Erstellen von asynchronen und Event-basierten Programmen unter Zuhilfenahme von Observable-Sequenzen ermöglicht. Diese Bibliothek ist eine Kombination aus Observer Pattern, Iterator und funktionaler Programmierung. Es enthält u.a. eine Implementierung des Observables, die von Angular benutzt wird. Ein Observable definiert eine Funktion, die erst ausgeführt wird, nachdem es durch einen Observer abonniert wurde. Solange die Funktion läuft oder der Observer sich nicht abmeldet, erhält dieser Benachrichtigungen. Es kann anstelle von Promises, Event APIs und Arrays benutzt werden. In dieser Arbeit werden Subjects anstelle von Observables verwendet. Ein Subject ist sowohl ein Observable als auch ein Observer. Dadurch kann es nicht nur auf Benachrichtigungen reagieren, sondern auch selber senden. Diese Eigenschaft ermöglicht es Subjects als Proxys zwischen einem Observable und vielen Observern zu agieren. Diese Methode wird verwendet, um die Kommunikation zwischen mehreren Komponenten, unabhängig von der Komponentenhierarchie, zu ermöglichen.

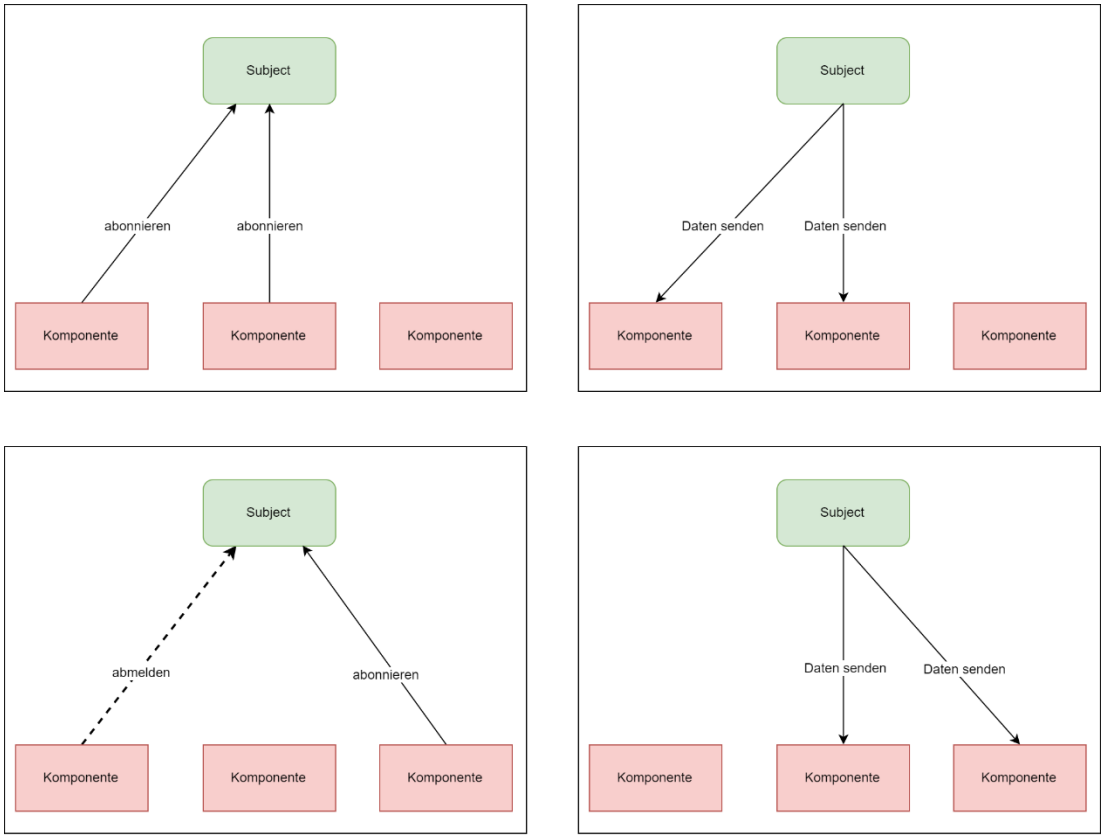


Abbildung 17: Funktionsweise von Subjects

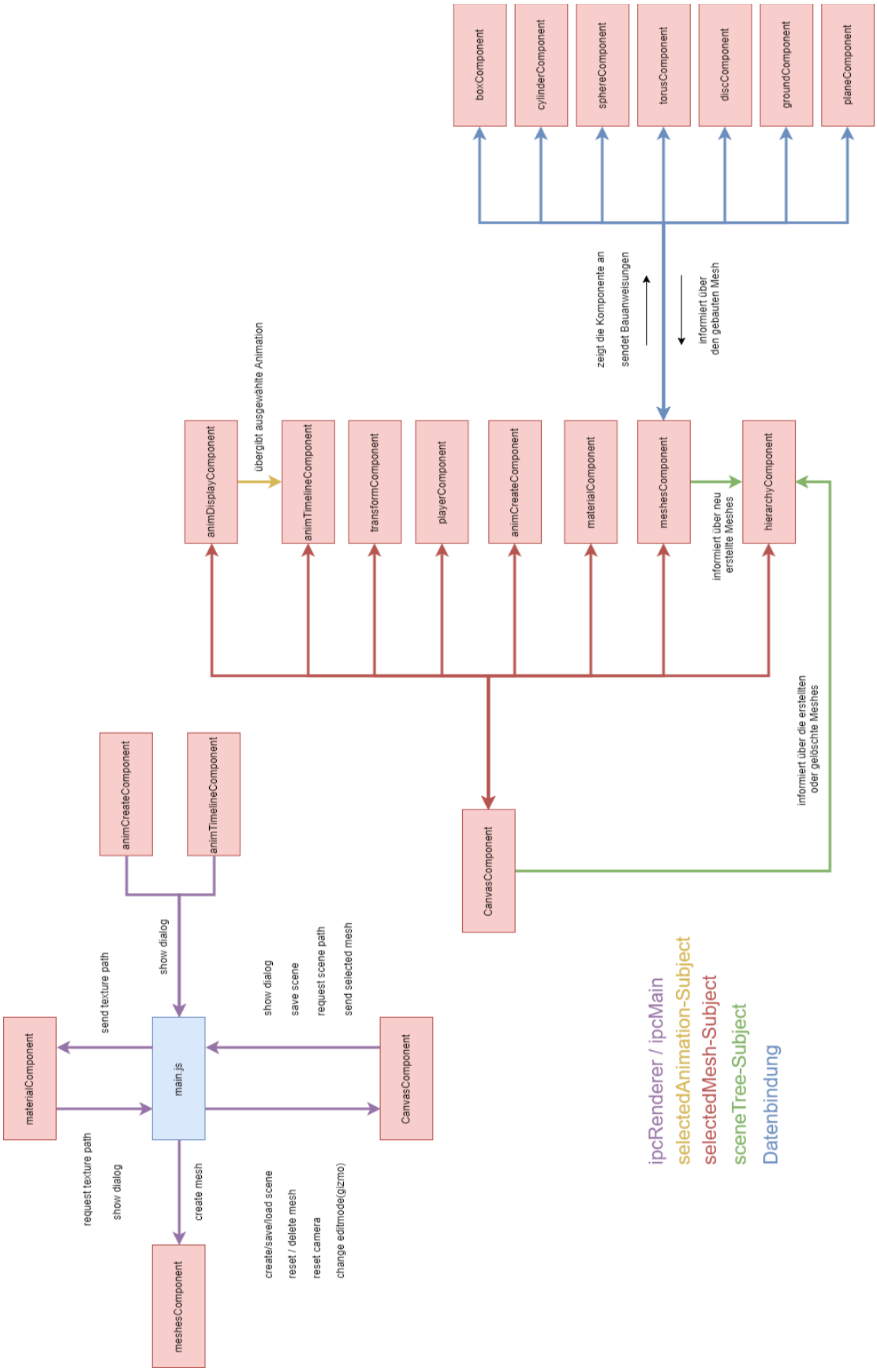


Abbildung 18: Kommunikation zwischen allen Komponenten

4 Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde eine Desktopapplikation mithilfe von Webtechnologien und dem Framework Electron entwickelt. Die Anwendung kann 3D-Objekte erstellen, transformieren und animieren. Sie hat Zugriff auf das Dateisystem und kann Dateien speichern oder laden. Sie besitzt ein anwendungsspezifisches Menü und zeigt Dialogfenster an. Durch den Komponenten-basierten Aufbau lässt sich die Applikation leicht erweitern oder modifizieren. Die Anwendung fühlt sich wie eine native Desktopapplikation an und wirkt nicht wie eine Webseite. Es ist plattformunabhängig und läuft auf Windows, Linux und Mac. Electron hat sich für die Aufgabenstellung als sehr geeignet herausgestellt. Der Entwicklungsprozess mit Electron ist unkompliziert und nachvollziehbar. Es ist gut dokumentiert und einsteigerfreundlich. Im Rahmen einer zukünftigen Arbeit können noch folgende Funktionen oder Aufgaben umgesetzt werden.

Die Anwendung kann momentan nur einfache Objekte erstellen. Babylon.js bietet aber die Möglichkeit, 3D-Objekte aus bestimmten Dateiformaten oder anderen 3D-Grafiksuiten wie Blender, 3DSMax und Maya zu importieren. Dies könnte benutzt werden, um auch komplexere Figuren zu animieren und bei Bedarf benötigte Funktionalitäten zu ergänzen. Außerdem kann es um weitere Animationsoptionen ergänzt werden wie z.B. Pfadanimationen oder Morphing.

Aufgrund der Nutzung von Babylon.js lässt sich die Anwendung leicht zu einer Spiel-Engine erweitern, sodass Web-basierte Spiele erstellt werden können. Durch Electron könnten diese Spiele dann wie normale PC-Spiele auf dem Rechner gespielt werden und vor allem auch auf allen drei Plattformen. Diese Aufgabenstellung kann auch in zwei geteilt werden. Eine Aufgabe beschäftigt sich mit der Umsetzung einer Spiel-Engine und die andere mit der Umsetzung eines PC-Spiels mit Electron.

Ein weiterer Punkt, der behandelt werden kann, ist die Usability und das Design der Anwendung. Zum jetzigen Zeitpunkt ist die Applikation schlicht gehalten, lässt sich aber aufgrund der Architektur leicht umgestalten.

Literaturverzeichnis

[1] P. B. Jensen, Cross-platform desktop applications, Shelter Island, NY: Manning Publications, 2017.

[2] Apache Cordova: https://cordova.apache.org/, Abruf: 20.8.2018

[3] Johnny-Five: http://johnny-five.io/, Abruf: 20.8.2018

[4] Blog Codinghorror: https://blog.codinghorror.com/the-principle-of-least-power/, Abruf: 20.8.2018

[5] Electron: https://electronjs.org/, Abruf: 20.8.2018

[6] C. Griffith und L. Wells, Electron: From Beginner to Pro, Berkeley, CA: Apress, 2017.

[7] Node.js: https://nodejs.org/en/, Abruf: 20.8.2018

[8] NPM: https://www.npmjs.com/, Abruf: 20.8.2018

[9] M. Jasim, Building cross-platform desktop applications with Electron, Birmingham, UK: Packt Publishing, 2017.

[10] A. Mead, Learning Node.js development, Birmingham, UK: Packt Publishing, 2018.

[11] Chromium Blog: https://blog.chromium.org/2013/04/blink-rendering-engine-for-chromium.html, Abruf: 20.8.2018

[12] The Chromium Projects: https://www.chromium.org/, Abruf: 20.8.2018

[13] TypeScript: https://www.typescriptlang.org/, Abruf: 20.8.2018

[14] R. Jones, Hrsg., ECOOP 2014 -- Object-oriented programming, Bd. 8586, Heidelberg: Springer, 2014.

[15] S. Fenton, Pro TypeScript, Berkeley, CA: Apress, 2018.

[16] Angular: https://angular.io/, Abruf: 20.8.2018

[17] A. Freeman, Pro Angular, Milton Keynes, UK: Apress, 2017.

[18] Babylon.js: https://www.babylonjs.com/, Abruf: 20.8.2018

[19] Konva.js: https://konvajs.github.io/, Abruf: 20.8.2018

[20] Ngx-electron: https://github.com/ThorstenHans/ngx-electron, Abruf: 20.8.2018

[21] Electron-packager: https://github.com/electron-userland/electron-packager, Abruf: 20.8.2018

[22] Electron TypeScript Quickstart: https://github.com/electron/electron-quick-start-typescript/blob/master/src/main.ts, Abruf: 20.8.2018

Eidesstattliche Erklärung

Ich versichere, dass ich die vorstehende Arbeit selbständig verfasst und hierzu keine anderen als die angegebenen Hilfsmittel verwendet habe. Alle Stellen der Arbeit, die wörtlich oder sinngemäß aus fremden Quellen entnommen wurden, sind als solche kenntlich gemacht.

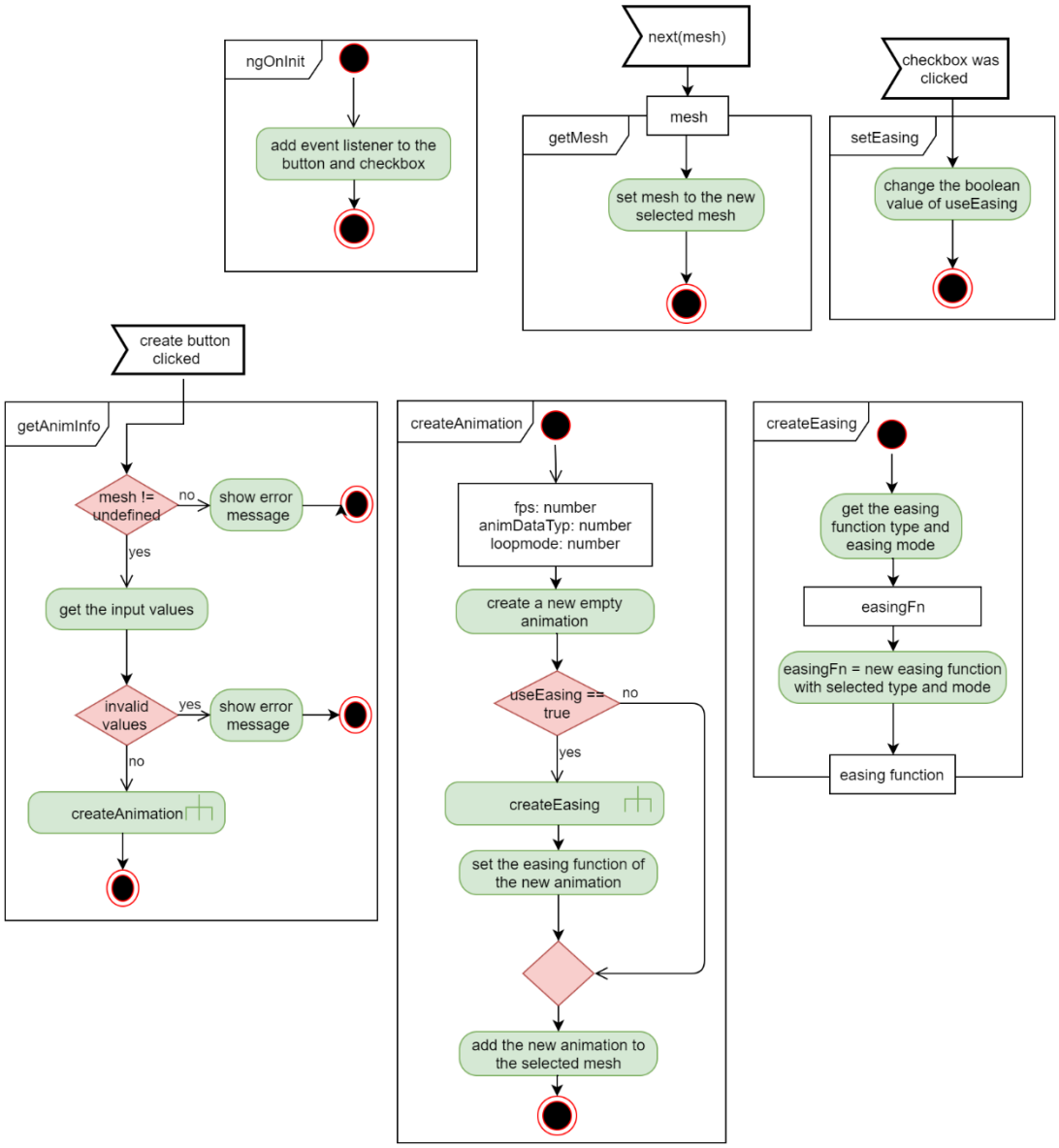
Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt oder an anderer Stelle veröffentlicht.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben kann.

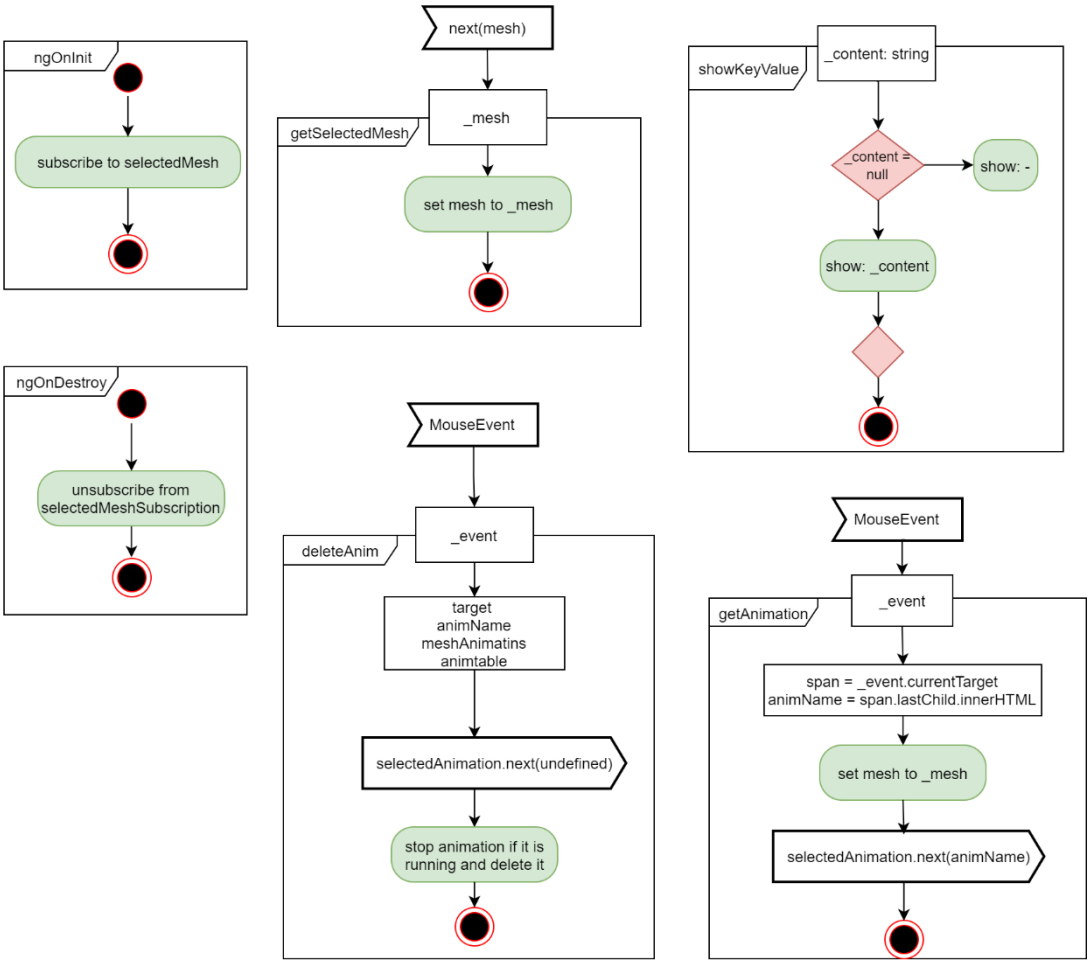
_____

Singen, den 20.08.2018 Robel Teklezghi

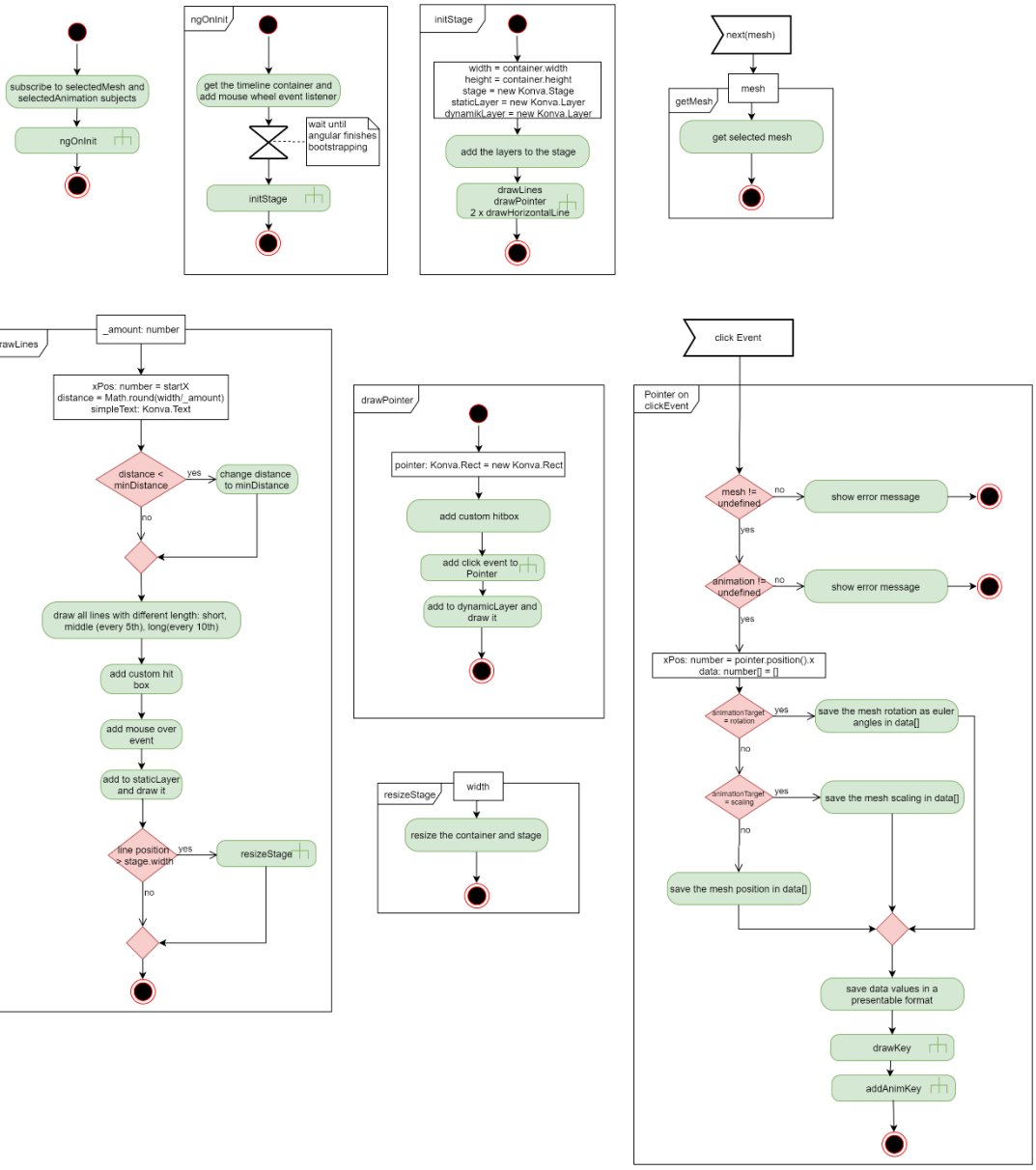
A. AnimCreateComponent

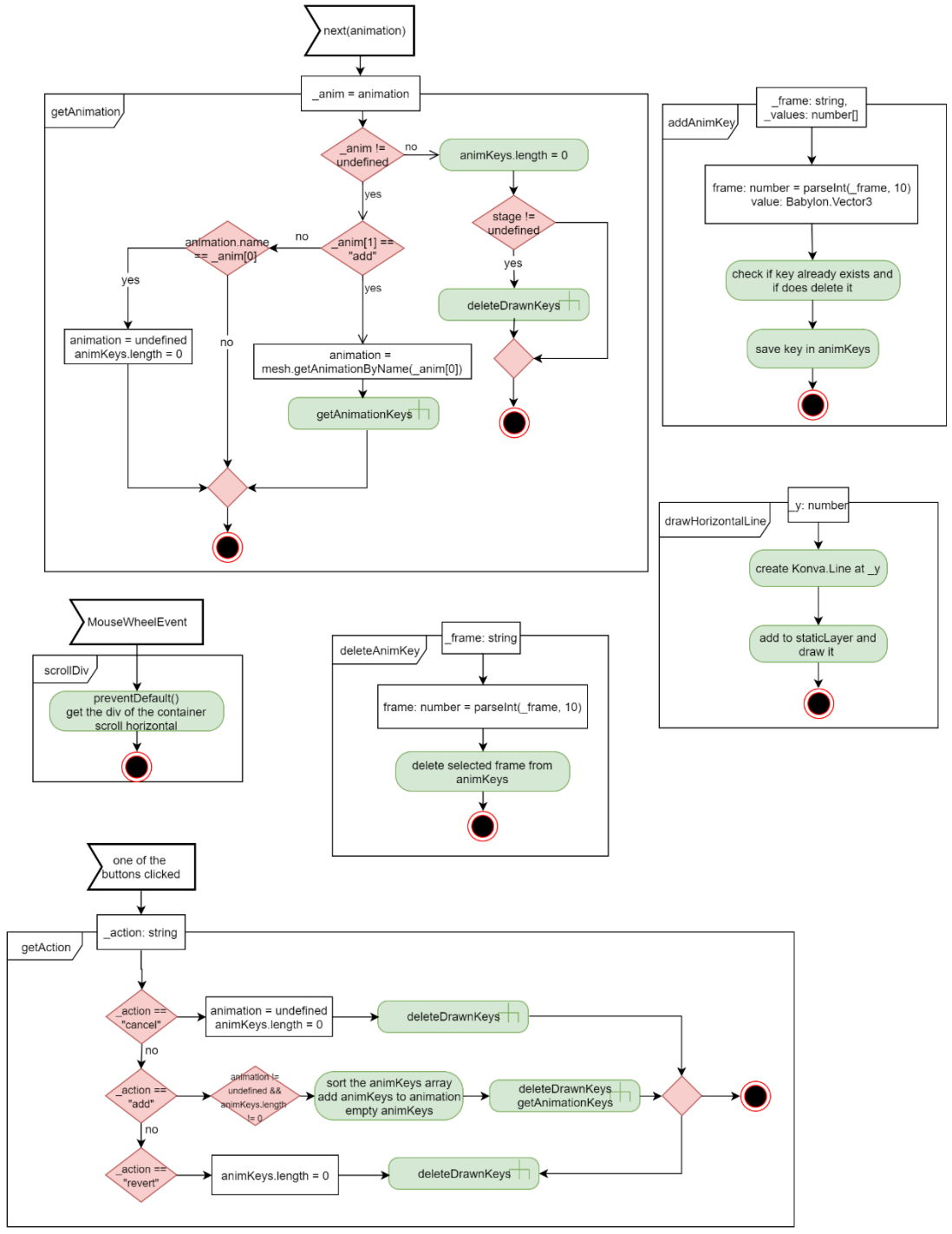


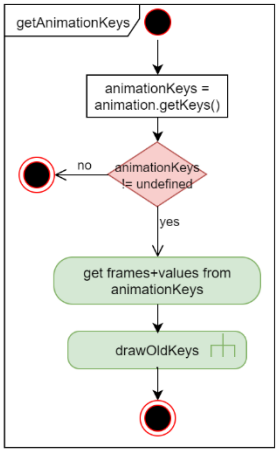
B. AnimationDisplayComponent

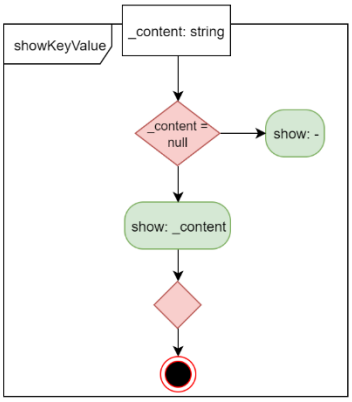


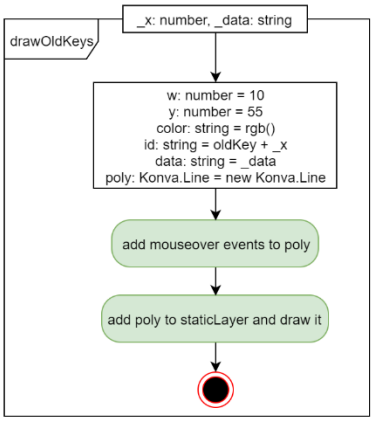
C. AnimTimelineComponent

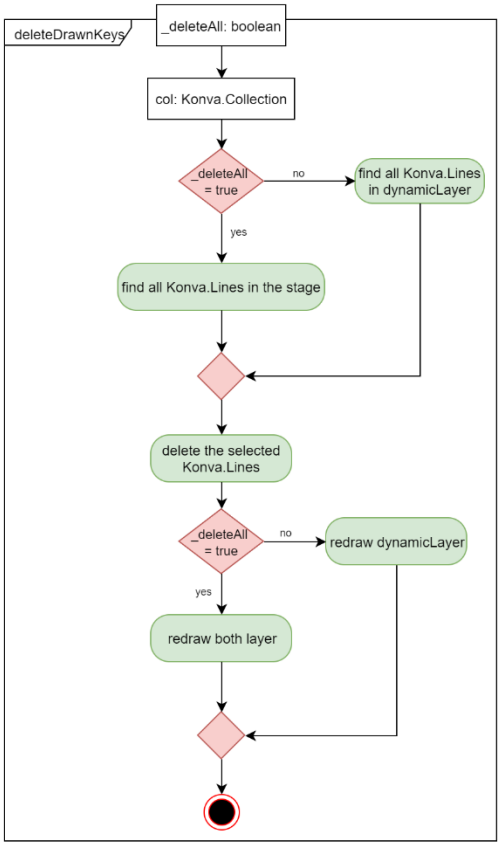


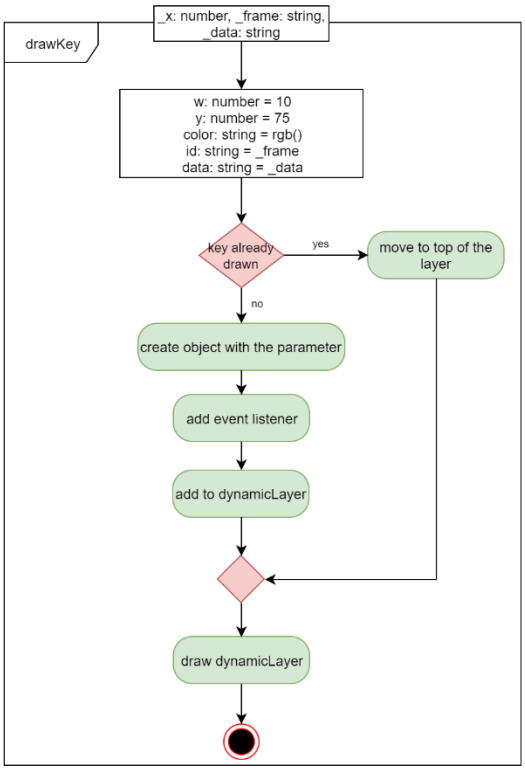

next(animation)
_anim = animation
getAnimation
_anim != undefined
no
animKeys.length = 0
yes
stage != undefined
yes
deleteDrawnKeys
animation.name == _anim[0]
no
_anim[1] == "add"
yes
yes
no
animation = undefined
animKeys.length = 0
animation = mesh.getAnimationByName(_anim[0])
getAnimationKeys
addAnimKey
_frame: string, _values: number[]
frame: number = parseInt(_frame, 10)
value: Babylon.Vector3
check if key already exists and if does delete it
save key in animKeys
drawHorizontalLine
_y: number
create Konva.Line at _y
add to staticLayer and draw it
MouseWheelEvent
scrollDiv
preventDefault()
get the div of the container
scroll horizontal
deleteAnimKey
_frame: string
frame: number = parseInt(_frame, 10)
delete selected frame from animKeys
one of the buttons clicked
getAction
_action: string
_action == "cancel"
animation = undefined
animKeys.length = 0
deleteDrawnKeys
no
_action == "add"
animation != undefined && animKeys.length != 0
sort the animKeys array
add animKeys to animation
empty animKeys
deleteDrawnKeys
getAnimationKeys
no
_action == "revert"
animKeys.length = 0
deleteDrawnKeys


getAnimationKeys
animationKeys = animation.getKeys()
animationKeys != undefined
no
yes
get frames+values from animationKeys
drawOldKeys

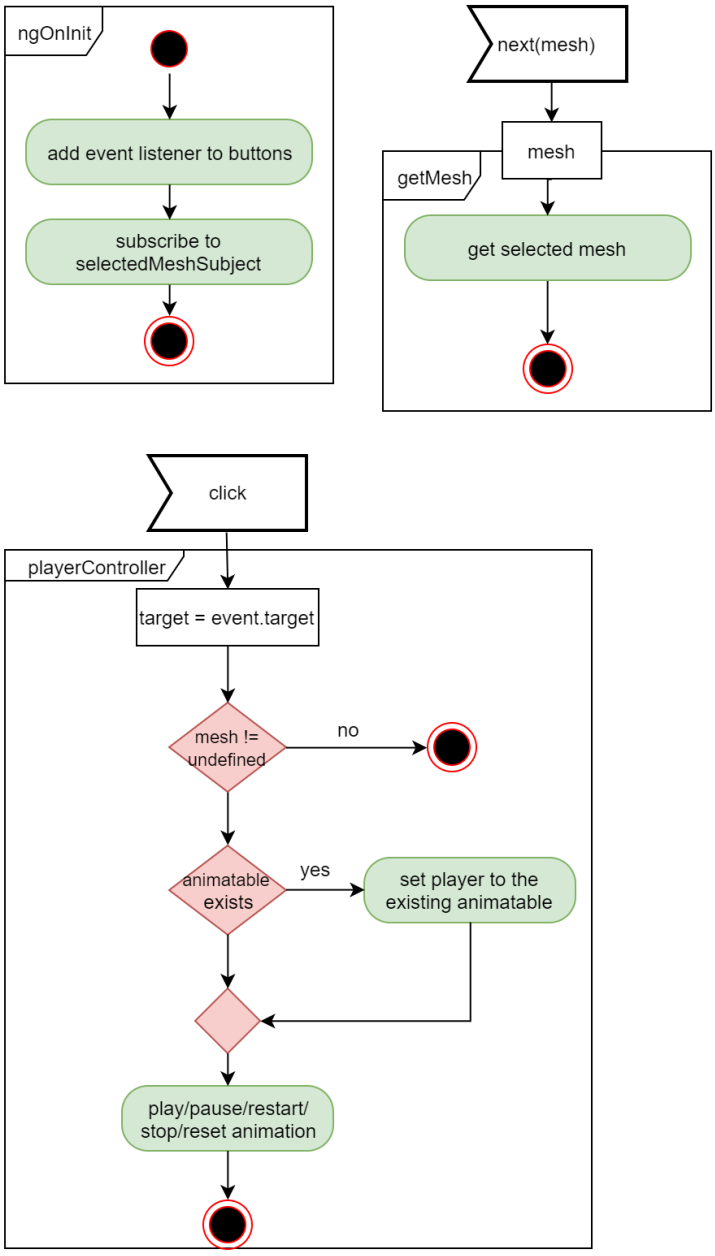

showKeyValue
_content: string
_content = null
show: -
show: _content


drawOldKeys
_x: number, _data: string
w: number = 10
y: number = 55
color: string = rgb()
id: string = oldKey + _x
data: string = _data
poly: Konva.Line = new Konva.Line
add mouseover events to poly
add poly to staticLayer and draw it

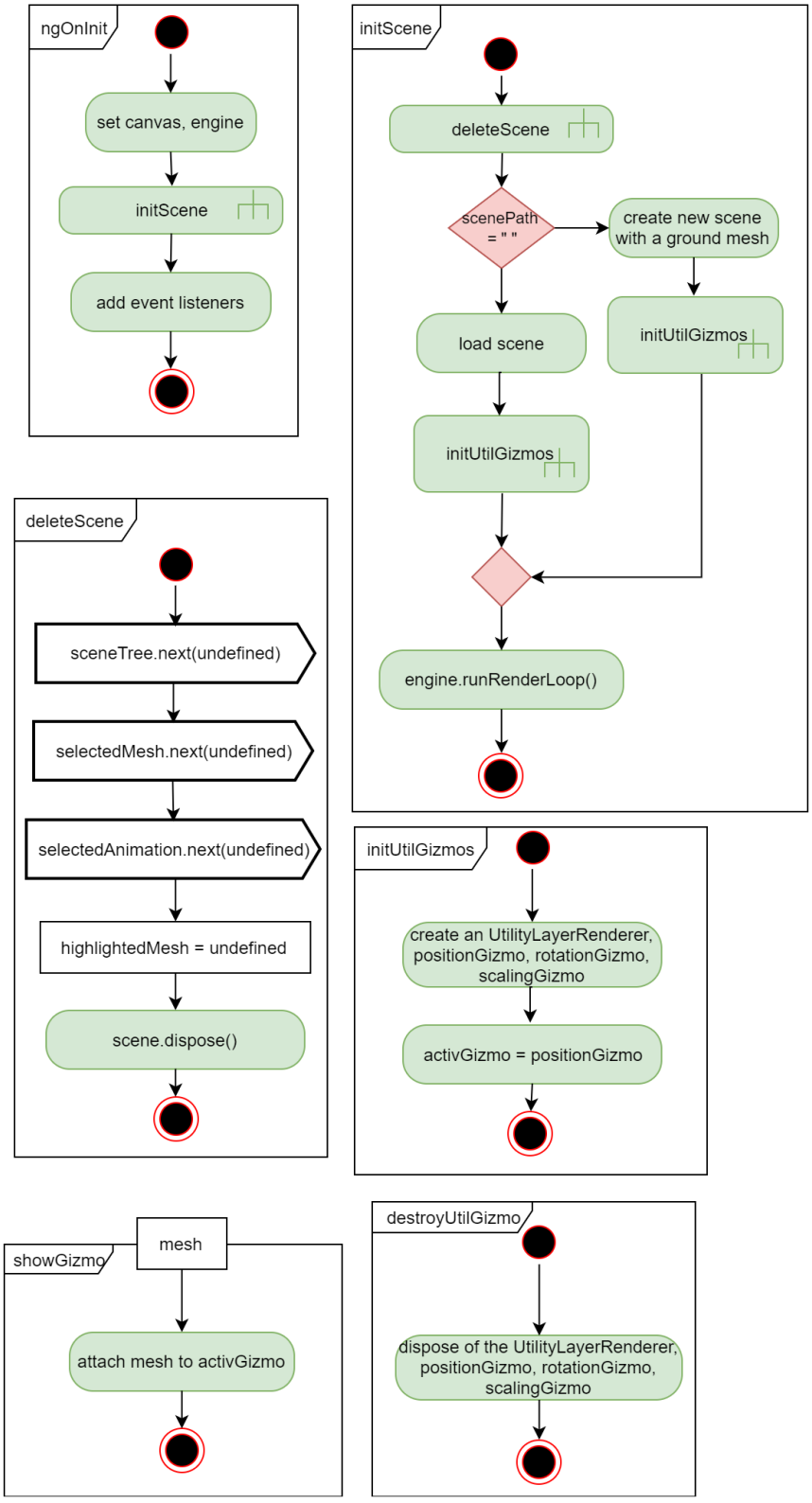

deleteDrawnKeys
_deleteAll: boolean
col: Konva.Collection
_deleteAll = true
no
find all Konva.Lines in dynamicLayer
yes
find all Konva.Lines in the stage
delete the selected Konva.Lines
_deleteAll = true
no
redraw dynamicLayer
yes
redraw both layer

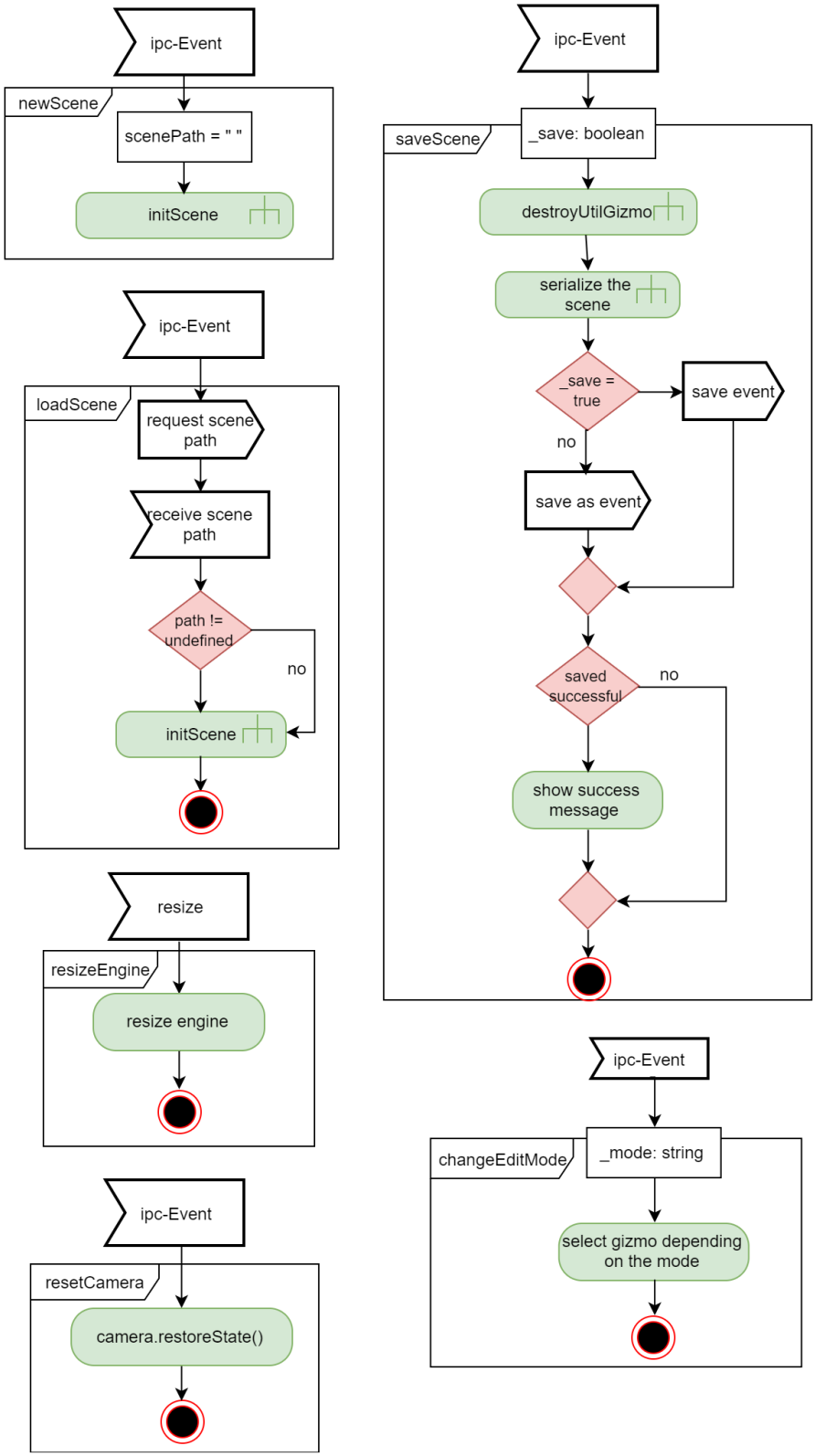

drawKey
_x: number, _frame: string, _data: string
w: number = 10
y: number = 75
color: string = rgb()
id: string = _frame
data: string = _data
key already drawn
yes
move to top of the layer
no
create object with the parameter
add event listener
add to dynamicLayer
draw dynamicLayer

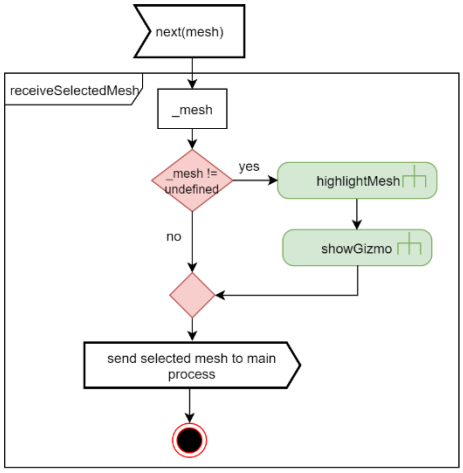
D. PlayerComponent

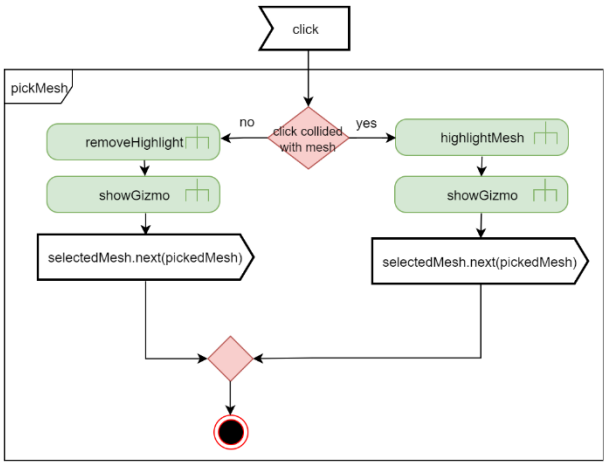


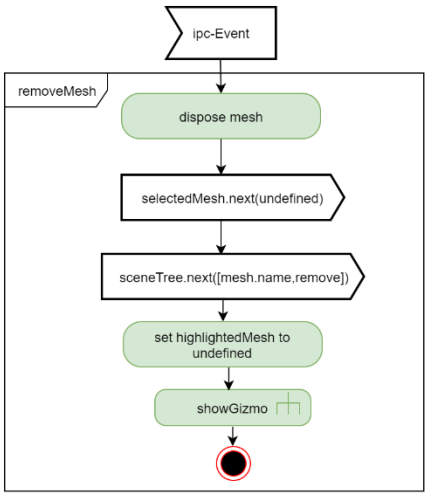
E. CanvasComponent

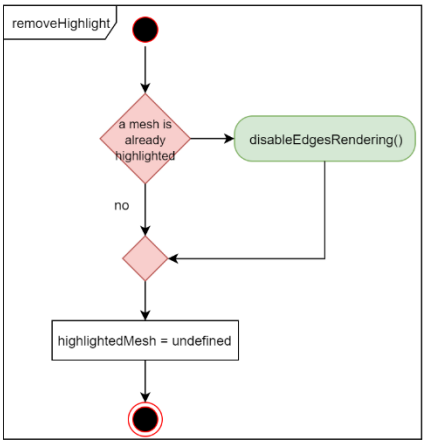


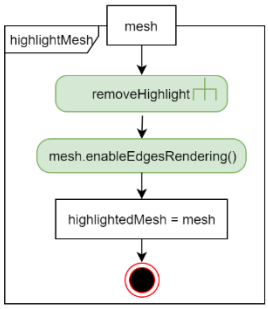

ipc-Event
newScene
scenePath = " "
initScene
ipc-Event
loadScene
request scene path
receive scene path
path != undefined
no
initScene
resize
resizeEngine
resize engine
ipc-Event
resetCamera
camera.restoreState()
ipc-Event
saveScene
_save: boolean
destroyUtilGizmo
serialize the scene
_save = true
save event
no
save as event
saved successful
no
show success message
ipc-Event
changeEditMode
_mode: string
select gizmo depending on the mode

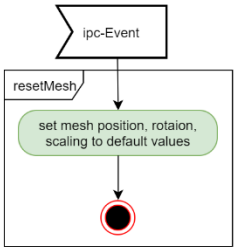

next(mesh)
receiveSelectedMesh
_mesh
_mesh != undefined
yes
highlightMesh
no
showGizmo
send selected mesh to main process

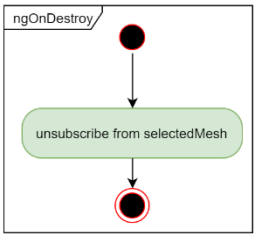

click
pickMesh
no
click collided with mesh
yes
removeHighlight
highlightMesh
showGizmo
showGizmo
selectedMesh.next(pickedMesh)
selectedMesh.next(pickedMesh)


ipc-Event
removeMesh
dispose mesh
selectedMesh.next(undefined)
sceneTree.next([mesh.name,remove])
set highlightedMesh to undefined
showGizmo

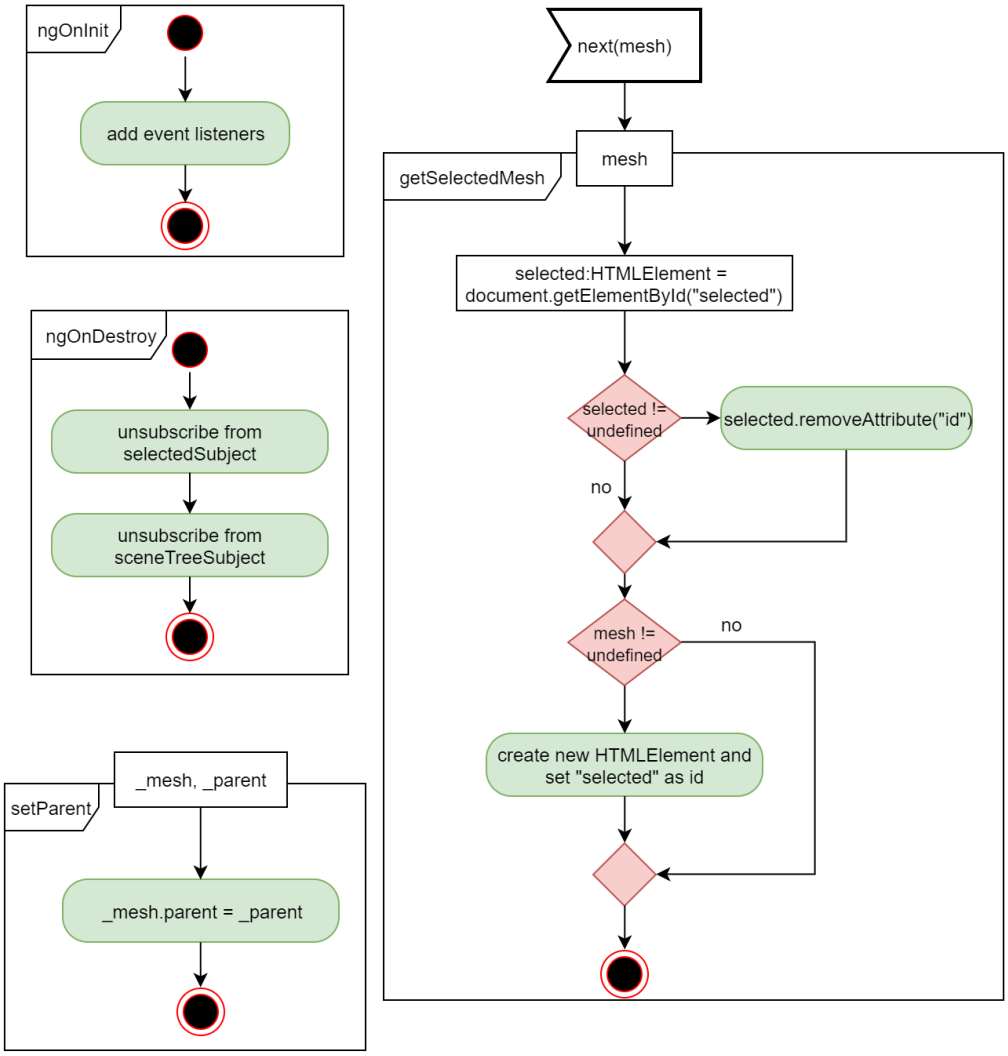

removeHighlight
a mesh is already highlighted
disableEdgesRendering()
no
highlightedMesh = undefined

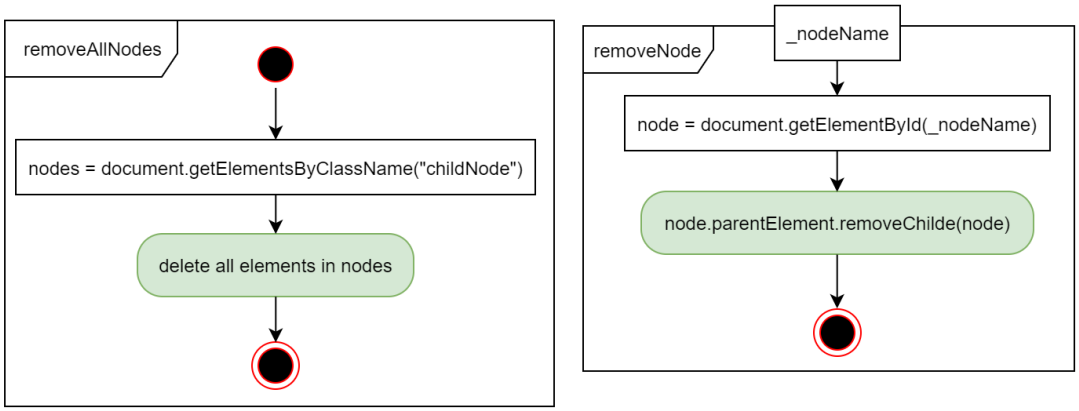

highlightMesh
mesh
removeHighlight
mesh.enableEdgesRendering()
highlightedMesh = mesh

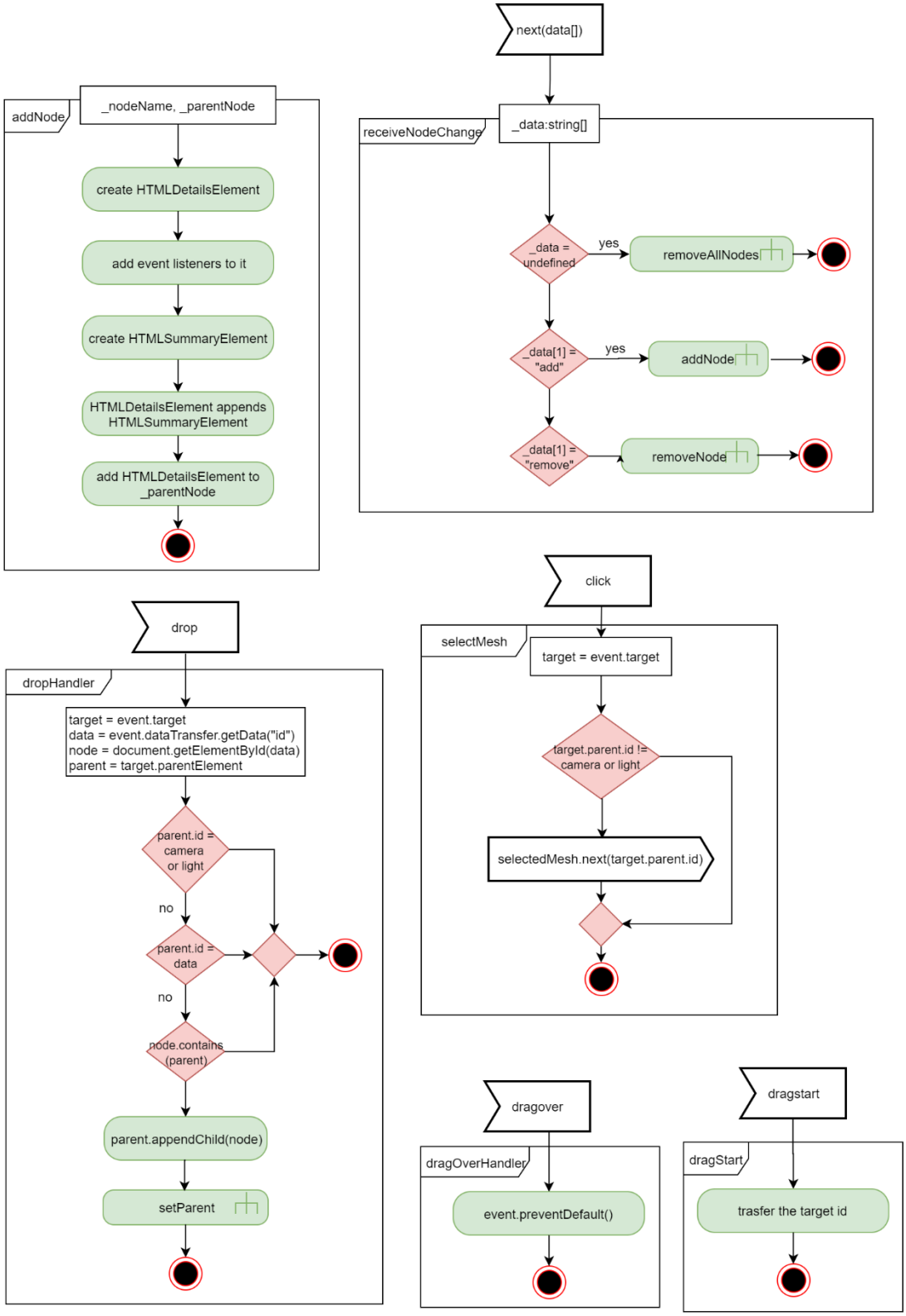

ipc-Event
resetMesh
set mesh position, rotaion, scaling to default values


ngOnDestroy
unsubscribe from selectedMesh

F. HierarchyComponent

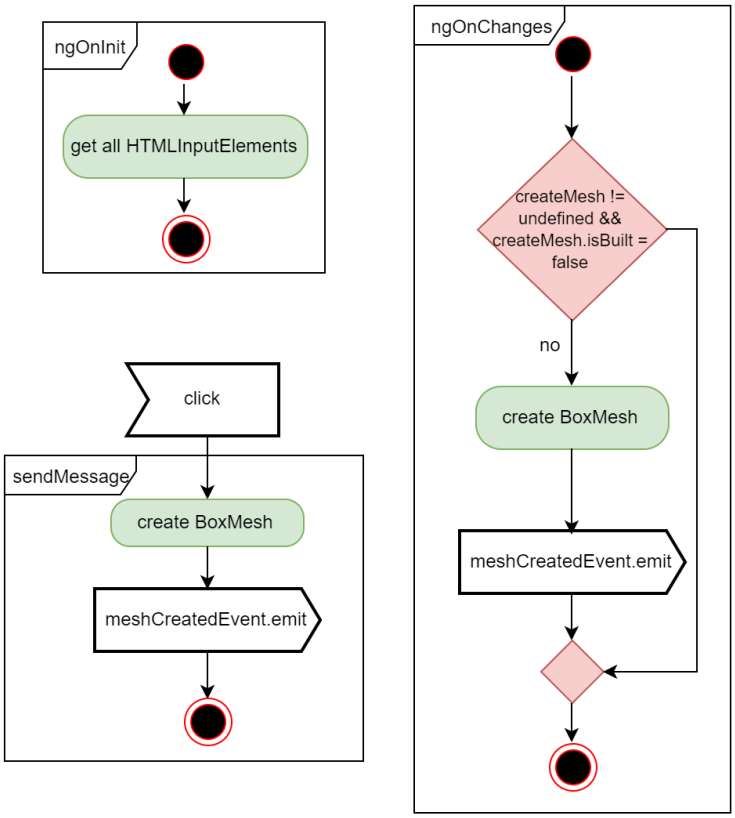




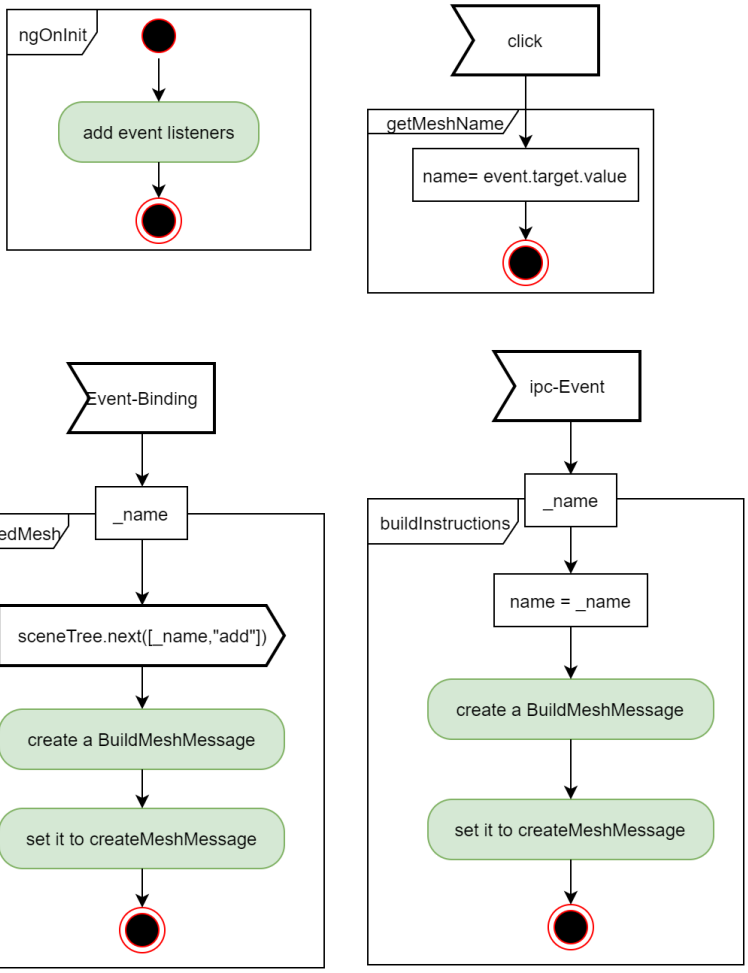

addNode
_nodeName, _parentNode
create HTMLDetailsElement
add event listeners to it
create HTMLSummaryElement
HTMLDetailsElement appends HTMLSummaryElement
add HTMLDetailsElement to _parentNode
next(data[])
receiveNodeChange
_data:string[]
_data = undefined
yes
removeAllNodes
_data[1] = "add"
yes
addNode
_data[1] = "remove"
removeNode
drop
dropHandler
target = event.target
data = event.dataTransfer.getData("id")
node = document.getElementById(data)
parent = target.parentElement
parent.id = camera or light
no
parent.id = data
no
node.contains (parent)
parent.appendChild(node)
setParent
click
selectMesh
target = event.target
target.parent.id != camera or light
selectedMesh.next(target.parent.id)
dragover
dragOverHandler
event.preventDefault()
dragstart
dragStart
trasfer the target id

G. BoxComponent

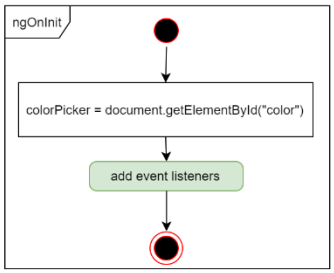
(gilt auch für Cylinder, Disc, Ground, Plane, Sphere und Torus)

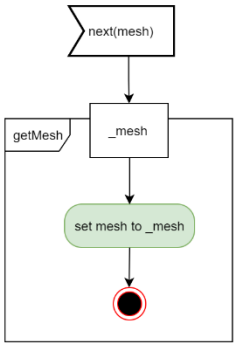


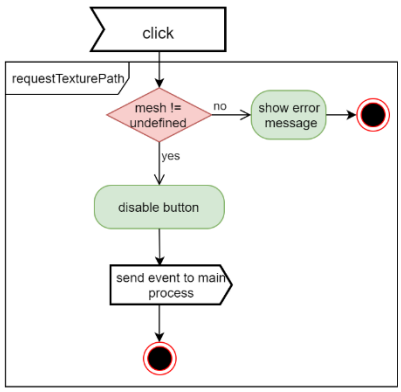
H. MeshesComponent

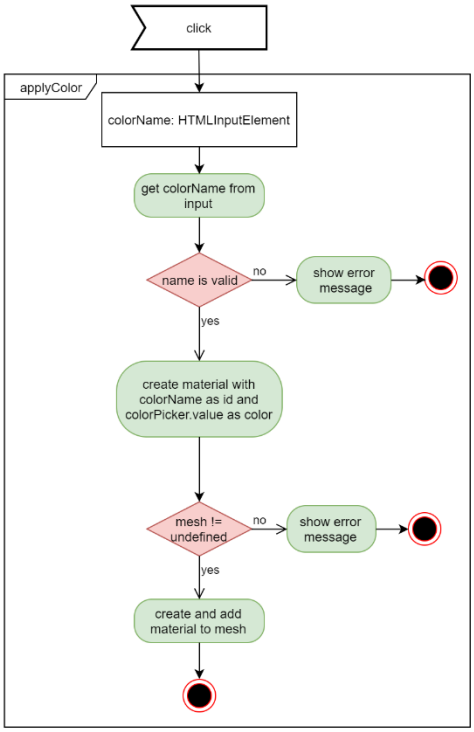


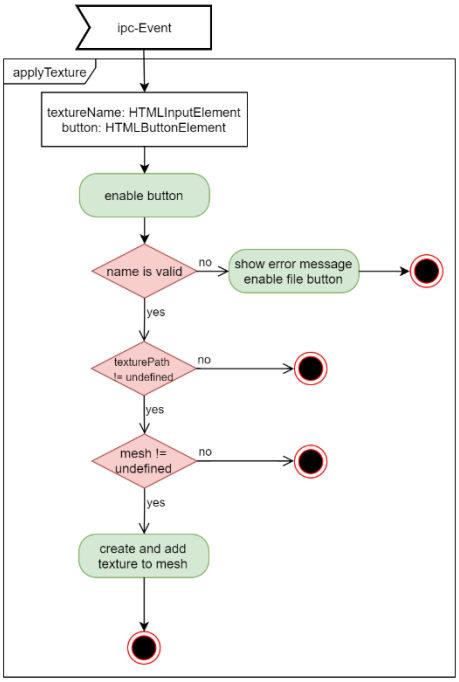
I. MaterialComponent











J. TransformComponent