Festlegung der verwendeten Werkzeuge

Inhalt

1 Einleitung

Nun, da die Anforderungen festgelegt sind, sind wir in der Lage Entscheidungen über die verwendeten Werkzeuge und Technologien zu treffen, welche bei dem neuen Werkzeug zur Gebäudeplanvisualisierung zum Einsatz kommen sollen. Im ersten Schritt wollen wir uns um eine Auflistung und Argumentation der verwendeten Technologien kümmern.

Nachdem zum Beispiel die verwendete Programmiersprache festgelegt ist, wird unter anderem eine Bibliothek für das Lesen von CAD-Dateien gesucht. Die verschiedenen Möglichkeiten werden gesammelt und anschließend verglichen, um die bestmöglichen Voraussetzungen für die weitere Entwicklung der Anwendung zu schaffen.

2 Auflistung und Argumentation der verwendeten Technologien

Im vorangegangenen Vergleich von CAD-Viewer Programmen haben wir zwei grobe Kategorien von Programmen betrachtet: *web-basierte* und *klassische Desktopanwendungen*. Beide Varianten können mit denselben Features aufwarten und unterscheiden sich somit lediglich in der Art der Bereitstellung: Während die webbasierten Anwendungen durch die Eingabe einer URL in der Adresszeile eines Browsers gestartet werden können, müssen die Desktopprogramme lokal auf dem System des Benutzers installiert und gestartet werden.

Anhand der nichtfunktionale Anforderung **NA1** (*Einfache und schnelle Bereitstellung/Installierbarkeit*) tendieren wir daher eher zu einer Webanwendung. Diese sind tendenziell schneller zu verwenden und leichter zu teilen. Außerdem stehen geeignete Browser praktisch jedem Endanwender zur Verfügung - selbst auf dem Smartphone.

Im Gegensatz dazu verleitet die nichtfunktionale Anforderung **NA4** (*Ruckelfreie Navigation im Gebäudeplan*) eher dazu, die Gegenseite der klassischen Desktopanwendungen zu befürworten. Webanwendungen liefern klassischerweise weniger Performance verglichen mit Desktopanwendungen. Grund dafür ist die Programmiersprache *JavaScript*, welche von den verschiedenen Browserimplementierungen mehr oder weniger schnell interpretiert wird. Allerdings gab es in den letzten Jahren deutliche Performanceverbesserungen (Rose & Hildebrand, 2015, S. 576). Beispielsweise gibt es speziell für das Rendering von Visualisierungen mittlerweile Unterstützung durch die vorhandene Grafikhardware des Benutzers, auf welche mit Hilfe der *WebGL* API zugegriffen werden kann (Rose & Hildebrand, 2015, S. 576).

Da mit der zu entwickelnden Anwendung möglichst auch äußerst komplexe und große Gebäudepläne dargestellt werden sollen, ist es sinnvoll ebenso auf der WebGL API aufzusetzen. Vorhandene JavaScript Bibliotheken wie *three.js* („three.js - JavaScript 3D library“, 2020) kapseln die WebGL API auf einer höheren Ebene, um diese schnell und einfach zu verwenden. Das ist für unser Projekt ebenfalls wichtig, da die nichtfunktionale Anforderung **NA2** eine einfache Wartbarkeit fordert. Die WebGL API ist an OpenGL ES 2.0 angelehnt und daher nur mit dem nötigen Hintergrundwissen zu verwenden („WebGL“, 2020). Die Bibliothek three.js scheint daher geeignet für das Projekt, um eine spätere gute Wartbarkeit sicherzustellen.

JavaScript ist eine dynamisch typisierte Programmiersprache. Alternativen

wie *TypeScript* von Microsoft hingegen versuchen eine statische Typisierung um JavaScript zu ermöglichen (Gao, Bird, & Barr, 2017, S. 758). Es wird oft argumentiert, dass eine statische Typisierung zu einem besseres Verständnis des Programmes, neben anderen Vorteilen, führt (Gao, Bird, & Barr, 2017, S. 758). Um die Wartbarkeit unseres Projekts weiter zu erhöhen, soll daher TypeScript als Programmiersprache verwendet werden.

Die Entwicklung einer Webanwendung kann durch die Verwendung eines geeigneten Frameworks vereinfacht werden. Wir haben uns für das Framework *Angular* („Angular“, 2020) entschieden, da dieses bereits alle benötigten Komponenten für eine moderne UI mitliefert („Angular Material“, 2020) und standardmäßig mit TypeScript zu verwenden ist.

Zusammenfassend soll das Werkzeug zur Gebäudeplanvisualisierung also als **Webanwendung** entstehen. Diese soll in der Programmiersprache **TypeScript** unter Verwendung des Frameworks **Angular** verfasst werden. Zusätzlich wird **three.js** zur einfacheren Verwendung der WebGL API verwendet, womit schlussendlich die Gebäudepläne visualisiert werden.

3 Vergleich von Parserbibliotheken für CAD-Dateien

Mittlerweile haben wir auch die Programmiersprache TypeScript festgelegt. Bei der Sprache handelt es sich um ein Superscript von JavaScript, welches zum Beispiel ein statisches Typensystem einführt („Typed JavaScript at Any Scale.“, 2020). Das bedeutet auch, dass es sich im Grunde um JavaScript handelt. Somit können Softwarebibliotheken, welche eigentlich in JavaScript geschrieben worden sind, ohne Probleme verwendet werden.

Wie bereits erwähnt soll das Werkzeug zur Gebäudevisualisierung `*.dxf` Dateien lesen können. Im JavaScript-Ökosystem sehen wir uns damit mit einer Auswahl an geeigneten Bibliotheken konfrontiert. Unsere Zielsetzung für dieses Kapitel ist es, die für unsere Zwecke bestmögliche Möglichkeit zu finden.

Als erster Schritt steht eine Internetrecherche auf dem Plan, welche zu einer Auflistung von Parserbibliotheken führen soll. Bei einer Suche über `npmjs.com` sind die folgenden Bibliotheken aufgetaucht:

- **Dxf-Parser** („dxf-parser - npmjs.com“, 2020)
- **dxf** („dxf - npmjs.com“, 2020)
- **dxftoobject** („dxftoobject - npmjs.com“, 2020)
- **dxf2svg** („dxf2svg - npmjs.com“, 2020)

Wir wollen diese nach *Performance (Ladegeschwindigkeit)*, *Lizenz*, *Beliebtheit auf* `npmjs.com` (Wöchentliche Downloads[1]), *Aktualität* und der Güte der *Dokumentation* beurteilen.

Zunächst wird die Performance gemessen, indem wir für jede Alternative dieselbe CAD-Datei lesen. Das Testprogramm, welches für jeden Messdurchgang durchgeführt wird, befindet sich im Codebeispiel 1.

```
1  public static readFile(file: File): void {
2      const reader: FileReader = new FileReader();
3      reader.onload = async (e) => {
4          const dxfContentStr: string | ArrayBuffer = e.target.result;
5
6          console.log("dxf:")
7          let timer = window.performance.now();
8          const helper = new DXF.Helper(dxfContentStr);
```

[1] Jeweils abgerufen am 25.10.2020 um 15:30

```
9           console.log(helper.parsed);
10          console.log(`dxf took ${window.performance.now() - timer}ms`);
11
12          console.log("dxf2svg:");
13          timer = window.performance.now();
14          console.log(dxf2svg.parseString(dxfContentStr));
15          console.log(`dxf2svg took ${window.performance.now() - timer}ms`
    );
16
17          console.log("dxf2object:");
18          timer = window.performance.now();
19          await dxftoobject.default(dxfContentStr).then(result =>
    console.log(result));
20          console.log(`dxf2object took ${window.performance.now() - timer}
    ms`);
21
22          console.log("dxf-parser:")
23          timer = window.performance.now();
24          console.log(new DxfParser().parseSync(dxfContentStr));
25          console.log(`dxf-parser took ${window.performance.now() - timer}
    ms`);
26      };
27      reader.readAsText(file);
28  }
```

Quellcodeauszug 1: Testprogramm zum Messen der Ladegeschwindigkeiten der verschiedenen CAD-Datei Lesebibliotheken.

Wie aus Tabelle 1 ersichtlich ist, reichen fünf Messungen aus, da die Standardabweichung jeweils sehr gering ist.

Messung	**Dxf-Parser**	**dxf**	**dxftoobject**	**dxf2svg**
1	35	69	936	52
2	35	40	892	44
3	45	52	873	56
4	43	65	933	66
5	34	61	900	52
Durchschnitt	38,4	57,4	906,8	54,0
Standardabweichung	4,630334761	10,36532682	24,27673784	7,155417528

Tabelle 1: Ladegeschwindigkeitsmessungen der verschiedenen CAD-Datei Lesebibliotheken mit dem Testprogramm - in Millisekunden.

In der folgenden Tabelle 2 werden die einzelnen Kriterien für die Bibliotheken bewertet. Um eine möglichst zutreffende Einschätzung zu erhalten, wird eine Nutzwertanalyse durchgeführt. Für alle Kriterien außer Lizenz werden die Punkte 1 - 4 vergeben. Die Lizenz erhält entweder 0 (Nicht in Ordnung) oder 1 (In Ordnung). Am Ende werden die Punkte, welche sich in der Tabelle neben jeder Einschätzung in Klammern befinden, aufsummiert und die geeigneteste Bibliothek ausgewertet.

•	**Dxf-Parser**	**dxf**	**dxftoobject**	**dxf2svg**
Performance	`38,4ms` (4)	`57,4ms` (2)	`906,8ms` (1)	`54,0ms` (3)
Lizenz	MIT (1)	MIT (1)	keine Angabe (0)	MIT (1)
Beliebtheit (wöchentliche Downloads)	634 (3)	702 (4)	11 (1)	31 (2)
Aktualität (Letztes Update)	29.11.2019 (2)	16.07.2020 (4)	23.12.2019 (3)	20.03.2019 (1)
Dokumentation	Zufriedenstellend (Quellcode vorhanden) (3)	Umfangreich (Quellcode vorhanden) (4)	Wenig (Quellcode vorhanden) (2)	Wenig bis nicht vorhanden (kein Quellcode vorhanden) (1)
Summe der Punkte	13	15	7	8

Tabelle 2: Vergleich der verschiedenen CAD-Datei Parserbibliotheken

Es wird also die Bibliothek **dxf** ausgewählt, da diese die höchste Punktzahl erhalten hat. Ein Blick in die Tabelle zeigt, dass lediglich die Performance bei dieser Bibliothek geringer ist, als bei *Dxf-Parser* oder *dxf2svg*. Allerdings handelt es sich hier um einen sehr geringen Unterschied von wenigen Millisekunden, wodurch dieser nicht weiter ins Gewicht fällt.

Somit stehen schlussendlich die verwendeten Werkzeuge für die zu entwickelnde Anwendung für Gebäudevisualisierungen fest und es kann mit der Implementierungsphase begonnen werden.

Literaturverzeichnis

Angular. (2020, Oktober). Abgerufen von https://angular.io

Angular Material. (2020, Oktober). Abgerufen von Angular Material website: https://material.angular.io

dxf - npmjs.com. (2020, Oktober). Abgerufen von npm website: https://www.npmjs.com/package/dxf

dxf2svg - npmjs.com. (2020, Oktober). Abgerufen von npm website: https://www.npmjs.com/package/dxf2svg

dxf-parser - npmjs.com. (2020, Oktober). Abgerufen von npm website: https://www.npmjs.com/package/dxf-parser

dxftoobject - npmjs.com. (2020, Oktober). Abgerufen von npm website: https://www.npmjs.com/package/dxftoobject

Gao, Z., Bird, C., & Barr, E. T. (2017). To Type or Not to Type: Quantifying Detectable Bugs in JavaScript. *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 758–769. https://doi.org/10.1109/ICSE.2017.75

Rose, A. S., & Hildebrand, P. W. (2015). NGL Viewer: a web application for molecular visualization. *Nucleic Acids Research*, *43*(W1), W576–W579. https://doi.org/10.1093/nar/gkv402

three.js - JavaScript 3D library. (2020, Oktober). Abgerufen von https://three-js.org

Typed JavaScript at Any Scale. (2020, Oktober). Abgerufen von https://www.typescriptlang.org

WebGL. (2020, Oktober). Abgerufen von MDN-Web-Dokumentation website: https://developer.mozilla.org/de/docs/Web/API/WebGL_API