FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Dowry Towns of Bohemian Queens - web-based 3D model viewer
Student:	Martin Púčala
Supervisor:	Ing. Jiří Chludil
Study program:	Informatics
Branch / specialization:	Web and Software Engineering, specialization Computer Graphics
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2022/2023

Instructions

Dowry Towns of Bohemian Queens is a platform for visualizing the historical environment in mobile applications using the means of augmented reality.

1. Analyze the possibilities of 3D image visualization concerning augmented reality (VRML, X3D, WebGL, applicable frameworks).
2. Perform analysis of the possibilities of the web environment in VR and AR applications.
3. Based on previous analyses, define the functional and non-functional requirements for the web application.
4. Design a prototype of the web application for displaying 3D models using methods of software engineering.
5. Implement the web application prototype.
6. Subject the resulting prototype to UX tests.

Electronically approved by Ing. Radek Richtr, Ph.D. on 16 February 2021 in Prague.

CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

F8

Faculty of Information Technology
Department of Software Engineering

Bachelor's Thesis

Dowry Towns of the Bohemian Queens - Augmented reality model visualization

Martin Púčala

May 2021
Supervisor: Ing. Jiří Chludil

Acknowledgement / Declaration

I thank my parents for their support, my supervisor Ing. Jiří Chludil for leading me through this journey and Ing. Radek Richtr, Ph.D., Ing. Petr Pauš, Ph.D. and Daniel Srb for valuable advice.

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Nesluša on June 27, 2021

.....

Abstrakt / Abstract

Táto práca sa pozerá na možnosti zobrazovania obsahu rozšírenej reality na mobilných zariadeniach v prostredí webového prehliadača. To znamená dostupnosť len obmedzených hardwarových i softwarových prostriedkov. Je implementovaný algoritmus ORB na detekciu obrazových príznakov a optimalizovaný pomocou paralelizovaného spracovania obrazu pomocou knižnice GPU.js. Bolo preskúmaných niekoľko algoritmov pre odhad polohy kamery a výpočet projekčnej matice. 3D modely sú zobrazené na obraze z kamery a sú preskúmavané možnosti ich zobrazovania tak aby sa zhodovali s orientáciou scény. Na konci sú popísané scenáre testujúce jednoduchosť používania ovládania výsledného widgetu.

Kľúčové slová: rozšírená realita, AR, ORB, perspektíva z n bodov, priama lineárna transformácia, DLT, WebGL, Three.js, GPU.js; TEX.

This thesis looks into the possibilities of displaying augmented reality content on mobile devices in the web browser environment. This means there are limited hardware and software resources available. The ORB algorithm for image features detection is implemented from scratch and optimized with the use of parallelized image processing using the GPU.js library. Several algorithms for pose estimation and projection matrix generation were collected and examined. 3D models are displayed over the camera feed and measures are examined to project it to match the scene's orientation. There are test scenarios described checking whether the controls of the resulting widget is easy to use.

Keywords: augmented reality, AR, ORB, perspective-n-point, direct linear transformation, DLT, WebGL, Three.js, GPU.js; TEX.

Contents /

Tables /

Chapter 1
Introduction

Mobile devices computational performance, mobile network speeds and prices and multimedia API possibilities in modern web browsers have come to the point where it's starting to make sense to provide tourists with new level of experiences: dig historical buildings and other structures that are no longer standing and let them be seen as if they were still in their place.

This thesis has two main goals:

- delve into the augmented reality algorithms and the options of their application in web applications
- prototype a 3D graphics viewer combined with live camera video stream

Displaying models in augmented reality means detecting the scene's pose and location and projecting models into the visible image so that they look like they belong into the scene. For this it is important to find a way to do this. This thesis looks to find algorithms for pose estimation and combine them with algorithms for image feature extraction and ways of parallel computation on the GPU to speed up this process.

Chapter 2
Analysis

Just a few decades, augmented reality was a mere sci-fi fantasy, something to be seen only in movies. The mathematics and optics has been now available for decades, but only recently pocket computers performance is becoming capable of processing camera and other sensors input and rendering output in real time.

By augmenting reality it is typically meant to enhancing the objects in the viewport with additional graphics. This means capturing and projecting captured information to the viewer with additional visible elements to the image. These added visual objects need to be projected transformed in such a way that they seem to belong into the scene.

In computer graphics, there are typically three main transformations done before projecting 3D image data - scaling, rotation and translation. When trying to to match projected image data to the image received from camera, lense distortions come into play and need to be taken into account. This is because when finding the transformations for projected models, distortions in the captured image can bring inaccuracies, resulting in projected image data being shown a bit off from expected positions.

Displaying objects as if they somehow belonged into the scene visible through the viewport on user's screen requires knowing object's position in the real world - otherwise it would feel as if it was fixed to the camera. An object's position in the real world can be read as a combination of GPS coordinates and a fixed (camera) coordinate frame for given location. Direction can be read from compass and gyroscope data.

With proper image transformations, image enhancement 3D data can be displayed.

2.1 Breaking up the problem

The problem of visualizing structures[1] using augmented reality can be broken up into several subproblems:

- determining camera position - typically by using a combination of sensors like GPS
- determining orientation - using compass and gyroscope
- loading model data with the appropriate mesh and texture versions matching weather and lighting conditions
- increasing accuracy of position and orientation using methods from photogrammetry
- model visualization transformed according to values in previous steps

[1] Already assuming the models exist in digital form and can be arbitrarily proocessed

2.2 Web as a platform

In the three decades since its invention, the World Wide Web has grown from a means for sharing scientific documents where support for embedding images was the top to a full-fledged ubiquitous multimedia platform available across all device platforms. Not limited to desktop computers and with growing performance of mobile everybody carries around in their pockets and ever-increasing range of supported technologies and a still widening range of sensors[1] it seems to be the ideal platform for presenting visual elements combined with information in text form and exploration.

2.3 Web graphics

There are many ways to display graphical data on web pages, not all of them are suitable for use of this project.

2.3.1 Images

Image support has been there in all major web browsers since the early versions of HTML and PNG, JPEG and GIF formats can today be rendered in virtually everywhere. These could be used in a very limited way using CSS 3D transforms[2] for small angles and rather surfaces.

2.3.2 SVG

Scalable Vector Graphics is best suitable for rendering 2D vector data. It can be used in an image tag and also inlined into HTML markup. 3D data could be displayed, but at the cost of modifying the document's contents and its elements' properties on every change. All that with no hardware acceleration and with all modifications being done in JavaScript code.

2.3.3 2D Canvas

For programmatically drawing geometric shapes, there is the HTML5 *canvas*[3] element. Canvas API can be used to draw basic graphic primitives as well as bitmap image data onto a drawing area occupied by the element.

2.3.4 VRML

Developed in the second half of the last decade of the 20th century, Virtual Reality Modeling Language is a file format for representing interactive 3D vector graphics mainly on the web. It was designed in the end of the 1990s and never gained bigger popularity. It was never natively supported and required a plugin installed, but today there are JavaScript libraries supporting loading and rendering of VRML files using newer 3D rendering technologies.

[1] GPS, compass, gyroscope, accelerometer, magnetometer, LiDAR in some newer devices etc.
[2] `https://developer.mozilla.org/en-US/docs/Web/CSS/transform`
[3] `https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API`

Simple VRML example[1]:

```
#VRML V2.0 utf8

Shape {
  geometry IndexedFaceSet {
    coordIndex [ 0, 1, 2 ]
    coord Coordinate {
      point [ 0, 0, 0, 1, 0, 0, 0.5, 1, 0 ]
    }
  }
}
```

2.3.5 X3D

EXtensible Three-Dimensional is a standard for declaratively representing 3D graphics, aiming to bring support for 3D data description to HTML5, similar to SVG.

Same example as above, in X3D notation[2]:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE X3D PUBLIC "ISO//Web3D//DTD X3D 3.2//EN"
  "http://www.web3d.org/specifications/x3d-3.2.dtd">

<X3D profile="Interchange" version="3.2"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
     xsd:noNamespaceSchemaLocation=
      "http://www.web3d.org/specifications/x3d-3.2.xsd">
<Scene>
  <Shape>
    <IndexedFaceSet coordIndex="0 1 2">
      <Coordinate point="0 0 0 1 0 0 0.5 1 0"/>
    </IndexedFaceSet>
  </Shape>
</Scene>
</X3D>
```

As for now, there is the X3DOM[3] library adding X3D rendering support using JavaScript and WebGL.

The X3DOM's website provides documentation and tutorial, but the examples provided are not always properly formatted and overall the project looks like a work-in-progress (slow progress - last commit in their GitHub repository was 5 months old at the time of writing this text). The project looks still alive and for could be very useful for simple uses of 3D objects (similar to inlined SVG icons).

3D graphics files loading is supported through a third-party project's tool[4], requiring conversion to another format to be then read by X3DOM nodes. This seems to be like an extra step with no benefits.

2.3.6 WebGL

With growing complexity of 3D data and ever-increasing demand for high-speed rendering in real time, requirements push towards minimizing data transformation

[1] https://en.wikipedia.org/wiki/VRML#Example
[2] https://en.wikipedia.org/wiki/X3D#Example
[3] https://www.x3dom.org/
[4] https://doc.x3dom.org/tutorials/models/aopt/

necessary between the serialized form (as saved in a 3D graphics files) and the form consumed by 3D APIs. This push is even stronger on platforms with limited power[1] capabilities.

This led to introduction of WebGL standard based on OpenGL ES 2.0. Most browser later came with support for WebGL 2.0 based on OpenGL ES 3.0, bringing several new capabilities[2]. This effort is, however, hindered by lack of[3] support from Apple. Their Safari browsers are the default on their mobile devices and even third-party applications are forced to use their WebKit component for rendering web pages[4]. This limits what can be done using web technologies on the mobile without workarounds and often big performance impact.

As 3D models rendering is typically highly computationally expensive[5], there's been an effort to minimize required processing from serialized data to binary data. This led to the introduction of typed arrays to the ECMAScript 6 standard[6].

Today, both VRML and X3D are only supported through a JavaScript libraries that use the lower-level WebGL APIs to render the high-level 3D data descriptions. WebGL API is rather verbose and basically what it does is it loads data (vertex coordinates, triangles formed from those vertices, texture images etc.) and code (GLSL shader programs compiled from plain text source) to the GPU memory and calls these programs. For basic rendering, there are two types of shaders - vertex shaders and fragment shaders. Vertex shaders typically transform vertex positions positions from model space to camera space by applying requested transformations. Fragment shaders mostly apply colors and textures with respect to the lighting. WebGL basically is a wrapper or JavaScript adapter for the OpenGL ES API, enabling JavaScript programs access to fast parallel calculations on the GPU[7].

To avoid unnecessary layer of processing, this thesis further takes into account only the WebGL APIs and frameworks wrapping its low-level APIs, freeing programmers from the most mundane tasks.

[1] In both meanings - performance and energy.
[2] `https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API#webgl_2`
[3] Or just partial and by default not enabled.
[4] `https://www.howtogeek.com/184283/why-third-party-browsers-will-always-be-inferior-to-safari-on-iphone-and-ipad/`
[5] Directly related to the level of detail of used models and textures.
[6] `https://exploringjs.com/es6/ch_typed-arrays.html#sec_intro-typed-arrays`
[7] And not only for graphics

2.4 WebGL frameworks

As the WebGL API is quite verbose, multiple libraries exist to shield developers from having to deal with the initialization.

The following criteria were found relevant to framework selection:

- license (free and open-source preferred)
- contributors count (so that the project doesn't suddenly die off)
- most recent contribution date (to avoid getting stuck with an abandoned project dependency)
- documentation availability (what is what, how things are meant to be done etc.)

From the List of WebGL frameworks on Wikipedia, three free open-source JavaScript frameworks with easily accessible source code repository have been pre-selected and briefly compared below.

2.4.1 A-Frame

A-Frame is a declarative open-source web framework for authoring 3D and VR scenes using HTML without the need to know WebGL. It was originally developed by Mozilla VR team since 2015 and is now maintained by Diego Marcos and Kevin Ngo from Supermedium and Don McCurdy from Google. There are about 350 contributors and supported file formats are glTF, OBJ, COLLADA, PLY and Three.js JSON format.

Almost 350 people have contributed so far, with latest commits being just over a week old.

Example code from A-Frame website[1]:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello, WebVR! - A-Frame</title>
    <meta name="description" content="Hello, WebVR! - A-Frame">
    <script src="https://aframe.io/releases/0.5.0/aframe.min.js">
</script>
  </head>
  <body>
    <a-scene>
      <a-box position="-1 0.5 -3" rotation="0 45 0"
        color="#4CC3D9"></a-box>
      <a-sphere position="0 1.25 -5" radius="1.25"
        color="#EF2D5E"></a-sphere>
      <a-cylinder position="1 0.75 -3" radius="0.5" height="1.5"
        color="#FFC65D"></a-cylinder>
      <a-plane position="0 0 -4" rotation="-90 0 0" width="4"
        height="4" color="#7BC8A4"></a-plane>
      <a-dodecahedron color="#B96FD3" position="4 1 -4">
        </a-dodecahedron>
      <a-text value="Hello, A-Frame!" color="#111"
        position="0 2.5 -2" align="center"></a-text>
      <a-torus-knot color="#B96FD3" position="-4 1 -4"
        radius-tubular="0.05" p="7"></a-torus-knot>
```

[1] https://aframe-school-primitives.glitch.me/solution.html

```
      <a-sky color="#ECECEC"></a-sky>
    </a-scene>
  </body>
</html>
```

Result can be seen in figure 2.1.

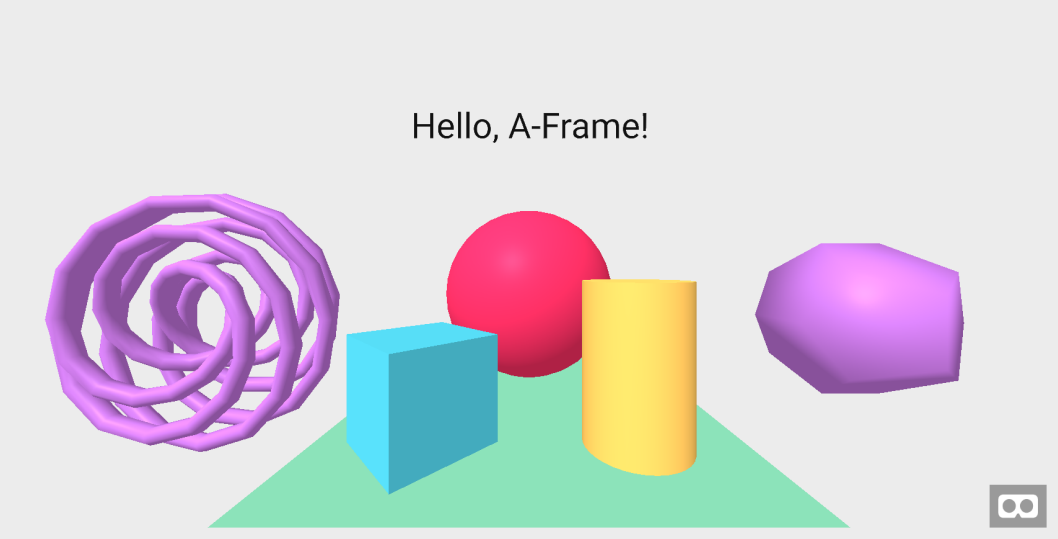

Figure 2.1. Basic A-Frame example. Screenshot of an example on project's website.

- Homepage: `https://aframe.io/`
- Source: `https://github.com/aframevr/aframe`
- Documentation: `https://aframe.io/docs/1.2.0/introduction/`
- License: MIT

The concise declarative syntax is definitely a plus for constructing 3D elements in code. On the other hand, the models to be visualized are expected to be ready and served by backend API, so this project cannot benefit from this. Still, A-Frame is built on top of Three.js and can use models with loaders available for this framework.

2.4.2 Babylon.js

Babylon.js is another open-source real-time engine for displaying 3D graphics in the browser using HTML5. First release in 2013 by Microsoft employees David Catuhe and David Rousset, with the help of artist Michel Rousseau, today it's being developed with orientation on 3D games. Contributors count has risen to about 350 since then. Beside games, it is used also in crime data visualization, medical education, product design etc. It supports STL, OBJ and glTF file formats.

Just a few contributors short of 350, latest commits are less than a week old.

- Homepage: `https://www.babylonjs.com/`
- Source: `https://github.com/BabylonJS/Babylon.js`
- Documentation: `https://doc.babylonjs.com/`
- License: Apache License 2.0

2.4.3 Three.js

Three.js is an open-source cross-browser GPU-accelerated 3D graphics library wrapping WebGL APIs. It is well-suited for visualizing 3D graphics in web browsers. It's development was started in 2010 by Ricardo Cabello (aka Mr.doob) and according to its GitHub page has gained since more than 1400 different contributors. The preferred file format is glTF, but COLLADA, FBX and OBJ are also supported.

Homepage of the project contains many examples, good API documentation and several tutorials. Further documentation can be found all around the web as this framework is the most popular one.

More than 1400 contributors with most recent commits only 2 days old.

- Homepage: `https://threejs.org/`
- Source: `https://github.com/mrdoob/three.js/`
- Documentation: `https://threejs.org/docs/`
- License: MIT

2.4.4 Comparison results

Evaluated frameworks are compared in table 2.1.

framework	license	contributors	most recent commit	docs
A-Frame	MIT	∼350	days	4/5
Babylon.js	Apache 2.0	∼350	days	4/5
Three.js	MIT	∼1400	days	4/5

Table 2.1. WebGL frameworks comparison table.

Three most popular frameworks were compared, all of them available under open source licenses, with good documentation, days-old latest contributions and hundreds of contributors. Based no these criteria, the best choice seems to be Three.js, which is also the most popular one.

2.5 3D file formats

Graphics data can be stored in textual form, binary form or mixed. Textual form's advantage is it is easily readable by humans. This can be important for debugging or interpreting data stored in the file. Textual data can take more space, especially with higher values where the text string (i.e. "123456") is longer than the binary representation (4 bytes bor 32-bit integers). This holds true even more for floating point numbers typically used in 3D graphics for vertex coordinates.

As the planned platform is the mobile browser, power - both electric and computational - is an expensive resource. 3D graphics data size depends on the amount of 3D vertices and faces and further information connected with each vertex/face (material, texture(s), normals etc.). This depends on the detail level of the stored model. Decreasing the detail level can lead to loss of visual information and thus lower visual fidelity and degraded user experience. Storing vertex and faces information in binary form can save memory and bandwidth, as well as the processing required to convert the textual data to binary form used to store values in numeric variables. Therefore, binary formats are preferred.

Having the framework selected, it would be convenient to have a loader that can load this format into the selected format, therefore mainly OBJ, FBX, COLLADA and glTF are of interest for this project. Currently, selected format doesn't matter much from the backend[1]'s perspective. As it was the case with glTF and GLB - adding another file format support was only a matter of extending several types and adding MIME types and file extensions to the backend code.

Googling for "3d file formats comparison" led to an article[2] with also a bit of background information. Additionally to the formats described there, glTF was added to the list. From the types of information listed on the page (geometry, appearance, scene, animations) only geometry (objectś shape) and appearance (colors, textures, material etc)[3] are relevant for this project.

Geometry can be saved in one of three ways:[4]

- *approximate mesh* - approximating the surface by tesselation into non-overlapping polygons, stores vertex coordinates and face normals and thus higher detail means more data to be stored
- *precise mesh* - more precisely (for the price of higher computational costs) modeling smooth surface with NURBS, storing their knots and control points
- *constructive solid geometry (CSG)* - instead of storing mesh information, shapes are created by combining primitive 3D shapes using boolean operations

To keep the performance requirements low and taking into account that visualized structure models typically are to be viewed from distance, approximate meshes should be sufficient (some details can be covered using textures and arcs and curves can be handled by using several models with different levels of detail for different scales depending on viewing distance).

[1] Will be described in the next section
[2] `https://all3dp.com/3d-file-format-3d-files-3d-printer-3d-cad-vrml-stl-obj/`
[3] Ibid.
[4] Ibid.

Based on the information above, the ideal format should:

- be free
- be open-source
- be binary
- have an existing loader for Three.js.

2.5.1 Wavefront OBJ

This plain text open source format developed by Wavefront Technologies has beed adopted by many other vendor. It stores 3D model geometry as a sets of vertex coordinates, normals, material references, texture UV coordinates and face vertex indices. Textures and material definitions are stored in extra files. The format is simple to read, but not very economical from the standpoint of processing power as it needs to be parsed and processed at load time. There is a model loader available for Three.js and Blender can also export models in this format.

Very simple (and short) example borrowed and modified from an online article[1], resulting model can be seen in figure 2.2 rendered with Online 3D Viewer[2]:

```
# cube.obj
mtllib master.mtl
v 0.000000 2.000000 2.000000
v 0.000000 0.000000 2.000000
v 2.000000 0.000000 2.000000
v 2.000000 2.000000 2.000000
v 0.000000 2.000000 0.000000
v 0.000000 0.000000 0.000000
v 2.000000 0.000000 0.000000
v 2.000000 2.000000 0.000000
# 8 vertices
g front
usemtl red
f 1 2 3 4
g back
usemtl blue
f 8 7 6 5
g right
usemtl green
f 4 3 7 8
g top
usemtl yellow
f 5 1 4 8
g left
usemtl orange
f 5 6 2 1
g bottom
usemtl purple
f 2 6 7 3
# 6 elements
```

Material file then contains color definitions, in this case only the ambient, diffuse and specular components:

[1] https://www.fileformat.info/format/wavefrontobj/egff.htm
[2] https://3dviewer.net

```
# master.mtl
newmtl red
Ka 1.000000 1.000000 1.000000
Kd 1.000000 0.000000 0.000000
Ks 0.500000 0.500000 0.500000

newmtl green
Ka 1.000000 1.000000 1.000000
Kd 0.000000 1.000000 0.000000
Ks 0.500000 0.500000 0.500000

newmtl blue
Ka 1.000000 1.000000 1.000000
Kd 0.000000 0.000000 1.000000
Ks 0.500000 0.500000 0.500000

newmtl yellow
Ka 1.000000 1.000000 1.000000
Kd 1.000000 1.000000 0.000000
Ks 0.500000 0.500000 0.500000

newmtl orange
Ka 1.000000 1.000000 1.000000
Kd 1.000000 0.500000 0.000000
Ks 0.500000 0.500000 0.500000

newmtl purple
Ka 1.000000 1.000000 1.000000
Kd 1.000000 0.000000 1.000000
Ks 0.500000 0.500000 0.500000
```

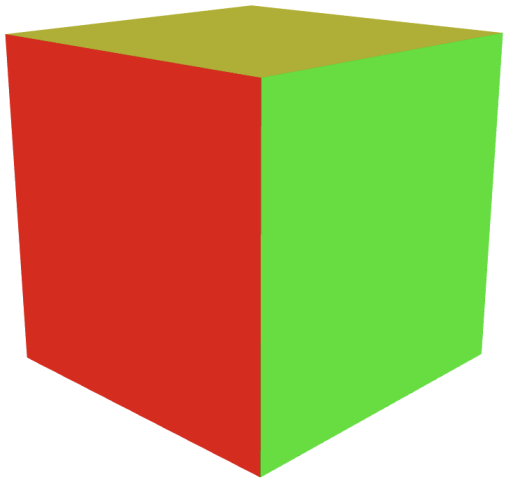

Figure 2.2. Basic 3D model example with different material applied to each face.

This format is great for playing without the need of any graphical tools, but the requirement of multiple files and textual form resulting in bigger file sizes are not ideal for more complex models.

Multiple detailed descriptions available[1] online and for example WebGL Fundamentals tutorial contains a basic instructions for its processing[2].

2.5.2 FBX (FilmBox)

Originally developed by Kaydaara for MotionBuilder[3], it is one of the main 3D exchange formats today even though it is proprietary and no official public documentation is available. There are both plain text and binary versions of the format. Blender plugin exists that implements both import and export without using the free but closed-source C++ FBX SDK.

Description is available for example on Blender Developers Blog[4] and there is also a blog post[5] about how to parse it (without sample code).

2.5.3 glTF (GL Transmission Format)

This is Three.js' preferred[6] format allowing to store binary 3D graphics data. Structure of the scene is stored as UTF-8 JSON string and buffer data like vertex coordinates is stored as binary block from the file can be loaded directly into JavaScript typed arrays and uploaded to GPU memory without any transforming step (as it is the case with OBJ or more complex document formats). As they state in the linked Three.js documentation: "... glTF is focused on runtime asset delivery, it is compact to transmit and fast to load. Features include meshes, materials, textures, skins, skeletons, morph targets, animations, lights, and cameras." For the purpose of this thesis, only meshes, materials and textures will be relevant.

glTF can be stored as a JSON file and accompanying binary data (geometry buffers, textures) in extra files (as shown in figure 2.3) at the cost of requiring additional network requests, or data can be embedded in data URIs at the cost of increasing thy payload size by one third due to base64 encoding used in data URIs, or everything can be encapsulated together in a combined GLB container as shown in figure 2.4. The benefit of combined data is the little processing required - scene structure is parsed from JSON by functions available in browsers and buffer data is read into typed arrays and sent directly to GPU with no further processing needed.

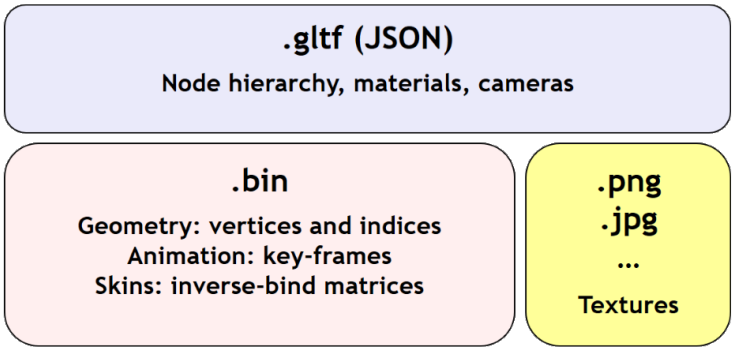

Figure 2.3. glTF overview. Image from project's GitHub page.

[1] For example https://www.fileformat.info/format/wavefrontobj/egff.htm
[2] https://webglfundamentals.org/webgl/lessons/webgl-load-obj.html
[3] https://docs.fileformat.com/3d/fbx/
[4] https://code.blender.org/2013/08/fbx-binary-file-format-specification/
[5] https://banexdevblog.wordpress.com/2014/06/23/a-quick-tutorial-about-the-fbx-ascii-format/
[6] https://threejs.org/docs/index.html#manual/en/introduction/Loading-3D-models

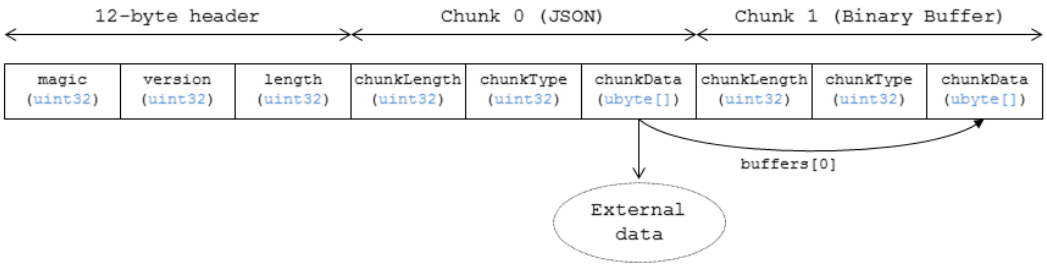

Figure 2.4. GLB overview. Image from project's GitHub page.

Blender also supports export to this format, so this adds no extra steps for the artists/content creators. Detailed public documentation is available on its GitHub page[1].

2.5.4 Comparison results

Evaluated 3D formats are compared in table 2.2. glTF comes as a winner from this comparison thanks to its free/open source nature, storing data in binary format that requires less processing when loading, and its detailed documentation.

format	open source	free	binary/text	docs
OBJ	yes	yes	text	4/5
FBX	no	yes	text/binar	3/5
glTF	yes	yes	binary+text	5/5

Table 2.2. 3D formats comparison table.

[1] https://github.com/KhronosGroup/glTF/blob/master/specification/2.0/README.md

2.6 Backend server

One part of the Dowry Towns project is also a backend server offering a REST API interface with JWT authentication. The initial version by Jindřich Máca[1] and further extended by Daniel Vančura[2] and at the time of writing, there was finished also the master thesis project by Dominik Sivák[3].

Current state of the backend server application from web frontend application's view was concisely described in a recently defended bachelor's thesis by Pavel Antoš[4], so only brief information follows, mainly about observed problems that could be tackled in future thesis projects.

2.6.1 Resources

3D models to be rendered need to be loaded depending on current location from a backend server. Besides the geographic location, there are more parameters planned to affect which textures and even model mesh is used (time period, time of the day, current weather conditions and possibly more). The Dowry Towns project's backend publishes these resources relevant for the visualizer:

The `structure` objects hold geographical location where the structure stands and a list of its available variants. Using the access information from the project of Pavel Antoš it seems like the structure resource does not support filtering by location.

For some reason the the `GET` request should support specifying a location parameter as a JSON dictionary. This seems to be worth of reevaluation as curly braces, commas and quotes are not necessary and would be escaped in the URL.

Also, the structures in the database seem to have invalid transformation matrix values set (all zeros) that if applied, would scale the models to nothing. Default value should probably be an identity matrix.

Furthermore, from previous discussions about the project requirements it sounded like there are situations when a building or a structure moves when a landslide occurs. Therefore it would probably make sense to extend the matrix with translation and add it as the fourth column.

There seems to be missing information about whether the 3 arrays in the `transformation` attribute hold columns or rows of the transformation matrix.

And as a last observed potential mistake, textures seem to be stored among models (at least so it seems from the links in in structure variants), even though there is a

[1] Accompanying thesis not published (yet)
[2] Vančura, Daniel. Věnná města českých královen - jádro. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2020.
[3] Sivák, Dominik. Věnná města českých královen - Backend administrační části. Diplomová práca. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.
[4] Antoš, Pavel. Věnná města českých královen – Webová aplikace pro schvalovací proces 3D modelů. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

separate endpoint for textures.

Having the benefit of being able to see this application for model approval process using the latest version of the REST API and its code was a great help when analyzing what requests will be required. This and the fact that the author already had some experience with the older version of the APAI greatly reduced the time necessary to analyze which objects to load for visualization.

2.6.2 Structure

The structure as a general term describes a building or a statue or some other historical object to be displayed in the real world and can have multiple variants for different historical time periods or weather conditions and so on. These variants are then stored as 3DObjects with different models, textures, assets and other parameters.

2.6.3 3DObject

This is an object representing possible "versions" of the structure as it looked in different periods or looks in different weather conditions. The `transform_matrix` attribute needs to be extended to have 4 columns so that it can hold a full transformation matrix, including the translation vector. Also, the API should set it by default to an identity matrix with zero translation vector and never allow storing a matrix with zero scale.

2.6.4 Model

Here is where the actual 3D model geometry file is stored and should reference textures and possibly other assets (like the .MTL file for .OBJ files with material coefficients and texture references).

2.6.5 Texture

Actual model texture files, currently for some reason stored under the same endpoint as models.

2.6.6 Asset

Various other files required by the stored model, e.g. material definition file for models in Wavefront's OBJ format.

2.7 Augmented reality

For the context of this thesis let the the term *augmented reality* denote the experience of perceiving the real world extended by the means of technology. The extensions can be perceived by multiple senses, but the most typical today and the only one taken into account in this thesis is visual. This typically means projecting some spatial visual information in such a way that it gives the observer the impression it somehow *belongs* into the observed surrounding scene of the world. To evoke this impression, the projected visual objects are positioned and transformed to fit into the scene alongside the real objects of the surrounding physical world.

Proper transformations are needed to position the virtual objects in the scene next to real objects. This typically involves computing transformation matrix that `anchors` the projected objects relatively to some visually distinctive parts of the environment. Finding these distinctive features in the image feed streamed from a digital camera requires resources that are becoming widely available only in the recent years (2010s-2020s). The research in these areas though has been ongoing for decades - research papers into edge and corner detection by Moravec[1], Harris[2] and others were published in the 1980s and one of the methods for calculation of camera's relative position and orientation in respect to known positions of a set of three points in the environment that is referenced by Haralick's 1994 paper[3] supposedly originates already in 1841[4].

This thesis tries to tackle the problem by first analysing camera video stream to find distinctive points that could be tracked and extracting some characteristic information for later matching. These are then matched against a set of points with known measured positions relative to the device's geographic location. Matched points are then tracked as they move on the screen and from their known 3D position and their position in the 2D image together with beforehand known device camera calibration information transformation matrix is calculated that positions the virtual 3D models into the scene.

2.7.1 Camera

Cameras record image by measuring light intensity hitting their sensors (greatly simplified). The photo-sensitive sensor measures the light intensity it gets illuminated with on each cell of its sensor grid (ignoring Bayer matrix and assuming each pixel in the image is just one point with a grayscale intensity). The light particles landing on the sensor surface are focused by camera's lense.

Lenses are not perfect and the resulting image is typically affected by distortions. The main two types are radial and tangential distortion and when real precision is required, they need to be measured by a calibration process and then compensated.

[1] Obstacle Avoidance and Navigation in the Real World by a Seeing Robot Rover, Hans Moravec, March 1980, Computer Science Department, Stanford University (Ph.D. thesis)

[2] C. Harris and M. Stephens (1988). A combined corner and edge detector. Proceedings of the 4th Alvey Vision Conference. pp. 147–151.

[3] Review and Analysis of Solutions of the Three Point Perspective Pose Estimation Problem, (R. M. Haralick, C. N. Lee, K. Ottenberg, and M. Nolle), International Journal of Computer Vision, Vol. 13, No. 3, 1994, pp. 331-356.

[4] J. A. Grunert, Das Pothenotische Problem in erweiterter Gestalt nebst Über seine Anwendungen in der Geodäsie, Grunerts Archiv für Mathematik und Physik, Band 1, 1841, pp. 238-248.

This process requires for each pixel to compensate its shift. This can be a complex process involving some iterative calculations and for purposes of this process it is left out and ideal lenses with no distortions are assumed.

The mapping from 3D world coordinates to 2D image coordinates is called a perspective projection and can be described as

$$x = PX = K[R|t]$$

- x is the homogenous vector with 2D image coordinates
- P is the projection matrix
- K is camera's calibration matrix
- R is rotation matrix
- t is translation vector

For estimating camera's position and direction, the P matrix needs to be decomposed into K, R and t. But P should be perfectly fine when only projecting a virtual structure into the scene observed on the device's screen.

2.7.2 Moravec corner detector

Moravec's corner detector takes a patch (denoting a squared/rectangular area) of an image of given dimensions (in pixels) and tries to calculate difference of sum of squared differences (SSD) of the intensities of all points in the patch from another patch moved by some small distances u and v belonging to the patch window $W_{x,y}$.

$$SSD(x,y) = \sum_{u,v \in W_{x,y}} (I(x,y) - I(x+u, y+v))^2$$

On a flat surface this sum results in a low value for all directions (Moravec's filter only works with 4 directions - horizontal, vertical and on both diagonals). On edges, the sum of squared differences (SSD) stays small in the direction the edge and higher in direction perpendicular to the edge. For corners, the result changes significantly in all directions considered.

The measure of cornerness is defined as the smallest SSD calculated for a shift in all of the considered directions - local maxima of this value indicate corners.

The author himself points out that this is method gives false positives for points on lines/edges going not exactly in these 4 directions.[1])

2.7.3 Harris corner detector

Improving on Moravec's work, Harris and Stephens[2] employ differential calculus to help and approximated the sum of squared differences using Taylor expansion (I_x and I_y here being the partial derivatives of the light intensity function in each pixel).

$$I(x+u, y+v) \approx I(x,y) + I_x(x,y)u + I_y(x,y)v$$

and gave different points in the patch different weights according to values of a 2D Gaussian function (see figure 2.5) to make it isotropic (direction invariant) thanks to the weight decreasing with distance from the point that is being processed.

$$G(x,y) = G(x)G(y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

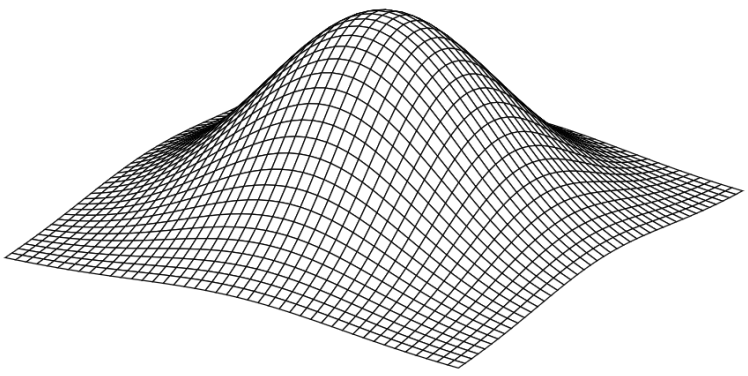

Figure 2.5. A two-variable Gaussian function.

[1] `https://en.wikipedia.org/wiki/Corner_detection`

[2] C. Harris and M. Stephens (1988). A combined corner and edge detector. Proceedings of the 4th Alvey Vision Conference. pp. 147–151.

This leads to

$$SSD(x,y) \approx \sum_u \sum_v w(u,v)(I(x,y) - I(x+u, y+v))^2$$

which can be written in matrix form as

$$SSD(x,y) \approx \sum_u \sum_v w(u,v) \begin{pmatrix} u & v \end{pmatrix} A \begin{pmatrix} u \\ v \end{pmatrix}$$

where A is the structure matrix

$$A = \begin{pmatrix} \sum_W {I_x}^2 & \sum_W I_x I_y \\ \sum_W I_x I_y & \sum_W {I_y}^2 \end{pmatrix}$$

When both A's eigenvalues are above some threshold, then that point is a corner. To avoid square root calculation, Harris and Stephens suggest instead to calculate

$$R = det(A) - \kappa \cdot trace^2(A)$$

with κ being a constant between 0.04 and 0.15. Higher R means higher cornerness factor.

2.7.4 Shi-Tomasi (a.k.a. Kanade-Tomasi) corner detector

This further improvement over Harris' filter brings more stable corners detection, Shi and Tomasi use the smaller of the eigenvalues of the same structure matrix A as the corner response score with value higher than some threshold indicating a corner:

$$R = min(\lambda_1, \lambda_2)$$

where both can be calculated by a quadratic formula

$$\lambda_{1,2} = trace(A) \pm \frac{\sqrt{trace^2(A) - 4det(A)}}{2}$$

Harris and Shi-Tomasi detectors not only detect, but also rate points according to their cornerness. The problem is, for running them on the full image is too expensive. Therefore it would make sense to prepare a narrower list of candidates, rank them by the cornerness measure and maybe only select a top N candidates. There can be multiple points evaluated as corners next to each other. To avoid possible mixup and wasting processing power on multiple adjacent pixels, each keypoint is compared with its close surroundings and only keypoints with higher corner score than their neighbours are kept. This is called *non-maximum suppression*.

2.7.5 FAST - Features from accelerated segment test

The FAST algorithm runs through a set of image points and on a circle of a given radius (see figure 2.6 from its Wikipedia page[1]) checks whether there is a continuous series of points that are either all darker or all lighter than the currently processed point. This can be done relatively fast, especially with a pre-computed list of point coordinates around the circle's circumference. Its speed makes this algorithm a good candidate for real-time image processing. On the other hand it often marks also edges and gives no information on the corner's orientation.

[1] `https://en.wikipedia.org/wiki/Features_from_accelerated_segment_test`

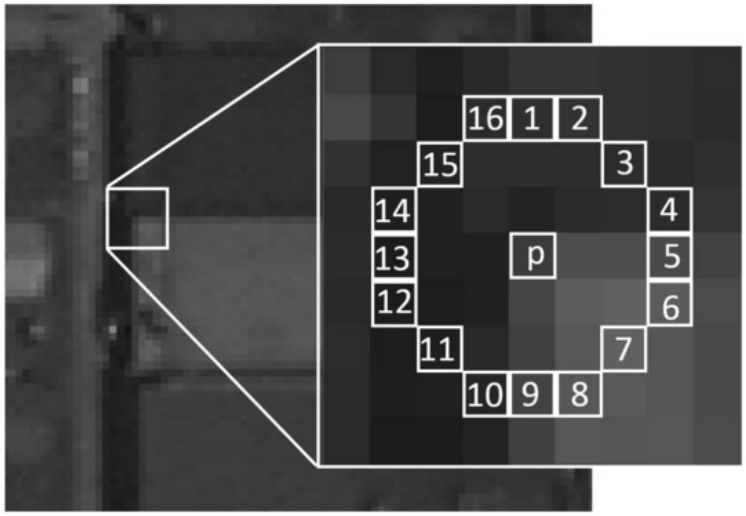

Figure 2.6. Keypoint candidate surrounding pixels examined by the FAST algorithm from its Wikipedia page.

2.7.6 ORB - Oriented FAST and rotated BRIEF

The ORB algorithm[1] combines the FAST algorithm for finding keypoints and the BRIEF algorithm for describing the patch surrounding that keypoint found by FAST.

The BRIEF (Binary Robust Independent Elementary Features) algorithm is a member of the binary descirptors family known for their speed when compared to the descriptors based on histogram of oriented gradients. It stores the keypoint patch description as a binary vector of the results of comparison of pixel intensities of random image pairs (e.g. figure 2.7, illustrative image from a introductory series[2]). It is then, of course, important to use the same list of coordinates when comparing two patches/their descriptros.

FAST nor BRIEF originally have no support for rotational invariance. This is added by finding the so called *intensity centroid* described by Rudin[3]. The idea is to find the angle the patch is rotated from its canonic orientation. This is done by using *tangens* inverse of the horizontal and vertical patch moments. This angle is then quantized into one of 30 angular steps and a lookup table is pre-calculated for each combination of the quantized patch orientation angle and every one of the patch pixel pair coordinates. This saves later calls to expensive *cos* and *sin* functions.

Having descriptor binary vectors, keypoints from an image or a single video frame can be matched to another set of descriptors to find correspodences. Correspodence pairs can be used for tracking keypoints changing locations between frames or for recognizing patches representing corners of physical objects with known measured positions in the real world.

[1] Rublee, Ethan & Rabaud, Vincent & Konolige, Kurt & Bradski, Gary. (2011). ORB: an efficient alternative to SIFT or SURF. Proceedings of the IEEE International Conference on Computer Vision. 2564-2571. 10.1109/ICCV.2011.6126544.

[2] `https://medium.com/data-breach/introduction-to-brief-binary-robust-independent-elementary-features-436f4a31a0e6`

[3] P. L. Rosin. Measuring corner properties. Computer Vision and Image Understanding, 73(2):291 - 307,1999.

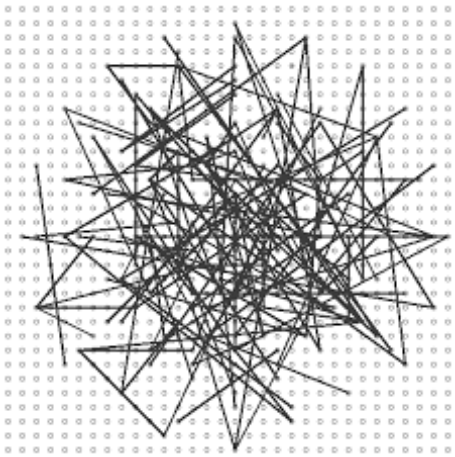

Figure 2.7. Randomized BRIEF pixel pairs.

Matching is done by simply finding keypoint pairs with the smallest Hamming distance of their binary descriptor vectors. This basically assumes that when two keypoints have patches of pixels around them with most of the relative light intensities similar, then there is a good chance that this is the same keyping in two images. This however leads to false positive matches on shapes and/or surfaces with regular repetitive patterns (like staircase banisters, balustrades, floor tiles, but also possibly grass or tree leaves etc). One of possible ways to deal with this can be the RANSAC algorithm, where translation and/or rotation transformation is computed that correctly maps most of the matching points. Those whose mapping does not work for most correspodences are then marked as outliers and are eliminated from further processing.

The most reliable algorithm for finding matches is brute-force matching when every descriptor is compared with all the others. This however brings a great performance penalty where the complexity grows to at least $O(n^2)$ when the circumstances allow using a combination of special instructions like *popcount*[1] with *xor* for counting different bits. In the JavaScript environment this would require preparing and loading compiled WebAssembly modules and to be used for this purpose. Faster alternative to brute-force matching is the FLANN algorithm using hierarchical data structures. The prices for higher speed here are less accurate results according to Pavel Kříž[2]. Šefčík also mentions cost of building the data structure, which would be worth of consideration. This could make sense in case of matching currently tracked keypoints against a keypoints dataset larger in order(s) of magnitude.

The ORB algorithm has been chosen due to its relatively good results[3][4] and low performance demands, which is important for its aimed use in real-time on mobile devices.

[1] `https://en.wikipedia.org/wiki/Hamming_weight`

[2] Kříž, Pavel. Urban scene recognition and editing II.. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

[3] Rublee, Ethan & Rabaud, Vincent & Konolige, Kurt & Bradski, Gary. (2011). ORB: an efficient alternative to SIFT or SURF. Proceedings of the IEEE International Conference on Computer Vision. 2564-2571. 10.1109/ICCV.2011.6126544.

[4] Šefčík, Jan. Rozpoznávání a editace urbanistické scény. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2020

This project uses the same set of generated 256 pixel pair coordinates is used as in the *Speedy-vision.js*[1], where its author adopted the list from the *OpenCV*[2] project.

[1] https://github.com/alemart/speedy-vision-js
[2] https://github.com/opencv/opencv

2.8 Pose estimation

Camera position and direction can be calculated from at least 4 point correspodences between 3D points in the world and their projections to the image. The goal is to obtain a rotation matrix R and a translation vector t or a 3×4 matrix $[R|t]$ combining them. There are methods that can be used with both calibrated and uncalibrated cameras and also methods for estimating the camera's calibration information from several images of objects with known dimensions. Another division can be to closed-form and iterative methods. Closed-form methods produce results by combining mathematical functions and operations as a finite list of steps. Iterative methods need an initial guess and continue adjusting it towards a good-enough solution. Whether a solution is good enough is measured by a so-called error function that, without knowing the best solution, can still say whether the current guess is closer or further from it. One more differentiating criterium could be required point correspodences between the world and the image plane.

The idea this thesis was heading towards was to get a video from the camera stream and after some pre-processing find distinct keypoints using the ORB (FAST + BRIEF + Harris or Shi-Tomasi + scale and orientation invariance modifications) algorithm due to its lower performance requirements (+ attempt to implement parallelization of this process using the GPU.js library to benefit from mobile devices' GPU power). Then, based on device navigation sensor's location, get relevant (by direction and distance) distinct keypoints, their descriptors and reference locations from the server. Then, matching keypoints from the camera and from the server obtain list of correspodences and to find an algorithm that would then produce a projection matrix that could be used to project the virtual structure models to the image. During the analysis of this problem, several other arised - calculating lighting and shadows and occlusion of the projected model. These were not even attempted to tackle.

Several methods methods were evaluated, some implemented from scratch (Grunert), some attempted but failed (DLT from 3D-2D correspodences), several converted from C++ (with minor complications solved along the way - like switched order of the matrices produced by SVD) from a great resource[1] discovered too late in the process. Many other are mentioned in one of the papers[2] discovered.

Further more (like Dementhon's POSIT) are mentioned in the thesis by Pavel Kříž[3].

There are are projects like js-aruco[4] and AR.js[5] that showcase solutions to problems similar to this project's, but simpler in the sense that they rely on detection of pre-defined markers in the camera stream. AR.js also has a demo with a so called NFT (non-fiducial tracking) marker.

[1] `https://visp-doc.inria.fr/doxygen/camera_localization/`

[2] S. Li, C. Xu and M. Xie, `A Robust O(n) Solution to the Perspective-n-Point Problem,` in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 34, no. 7, pp. 1444-1450, July 2012, doi: 10.1109/TPAMI.2012.41.

[3] Kříž, Pavel. Urban scene recognition and editing II.. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

[4] `https://github.com/jcmellado/js-aruco`

[5] `https://github.com/AR-js-org/AR.js`

Description of these algorithms is left out due to time pressure of the near deadline, therefore the author can only suggest to leave out the Grunert's algorithm and similar other in detail described and compared in Haralick's 1994 paper[1] due to its complexity both cognitive and computational (goniometric functions, division, 4th degree polynomial). The DLT algorithm seems to be a good starting point for 6 non-planar point correspodences (6 points in the image that can be pretty accurately matched to 6 points known in the world) or 4 planar correspodences (4 points in the image with 4 known points in the world where the world points lie all in one plane), only a working implementation (or linear algebra knowledge to implement it) of the SVD algorithm is needed.

Originally there was the idea of having to compute the camera's location was key for accurately positioninig the virtual models. For this, calibration information for each camera (each device can have multiple) would be necessary. There are projects like calibdb.net that even tried to collect this information and let users calibrate their cameras or just read the parameters for their device in case it already was in their database (they even wrote scientific papers[2][3] about it). Later, with evaluation of some of the solutions published with code and testing data (mentioned above), there came an important realization - the calibration information is not required (at least not as separate values or matrix) and all that is important is the projection matrix. There are procedures how to extract the estimated calibration data (for example prof. Cyrill Stachniss in his lectures for University of Bonn, published freely on YouTube explains it in his lecture on Direct Linear Transformation), but for the purposes of merely projecting virtual models to the scene, just the compound matrix produced by DLT is enough.

Contrary to the initial assumptions, the exact geographical coordinates do not matter for the purposes of this project. It probably has not much sense to display buildings in locations further than several tens or hundreds of meters. According to the Earth Curvature website[4] at 1 kilometer distance, 8 centimeters of Earth's curvature can be observed. If one looked at a screen with height of 50 centimeters from a distance of 50 centimeters, the tangent of this angle is 1. That means that (assuming a non-scaling lense) at a distance, maximum height of an object to fit on the screen would be equal to the distance, i.e. at 1000 meters a 1000 meters tall object would fit the screen. 8 centimeters on such a distance does not translate to even a single pixel. With a 4K display one pixel would mean about 26 centimeters. On a device with a Full HD resolution of about 1080x1920 (in portrait mode), a single pixel at this distanc is about 52 centimeters. As the typical sightseeing scenario considers even much shorter distances, Earth can be safely assumed to be flat and GPS coordinates and compass direction can be taken into account only as a guide to from which angular section of the space in front of the device and for what distance it makes sense to return points

[1] Review and Analysis of Solutions of the Three Point Perspective Pose Estimation Problem, (R. M. Haralick, C. N. Lee, K. Ottenberg, and M. Nolle), International Journal of Computer Vision, Vol. 13, No. 3, 1994, pp. 331-356.

[2] Rojtberg, Pavel, and Arjan Kuijper. `Efficient pose selection for interactive camera calibration.` 2018 IEEE International Symposium on Mixed and Augmented Reality (ISMAR). IEEE, 2018.

[3] Rojtberg, Pavel, and Felix Gorschlüter `calibDB: enabling web based computer vision through on-the-fly camera calib` 2019 Proceedings of the 24th International Conference on 3D Web Technology. ACM, 2019.

[4] `https://earthcurvature.com/`

with known coordinates to be matched with keypoints observed on the screen.

Each sightseeing location (a town or its part) can have assigned its own origin and have structures positioned in respect to that origin. After matching GPS location of the device with location of some sightseeing site and a spatial angle, i.e. where to the device's screen is oriented, a list of keypoints and its descriptors can be pre-selected (location and distance with relation to keypoint's zoom level pyramid location) and returned to the device. As the locations of existing building and especially their visually distinct features (corners, window and door corners, roof corners etc) and the location of the virtual 3D models are expressed in respect to the same reference coordinate system, projection matrix from observed points can be then used to project also the virtual models that need to stored properly scaled.

It can be expected to observe a reprojection error, i.e. the projected virtuaal model to be slightly shifted. This can be probably remedied with some iterative optimization methods like Levenberg–Marquardt or by stochastic least squares gradient descent.

2.9 Parallel computation in the browser using GPU.js

Image processing requires lots of similar computations that differ only by a few parameters and are independent from each other. The computations themselves are not necessarilly complex, but there may be many of them. Many independent computations can be slow when done one by one in sequence (serial processing) and the process can be sped up by parallel processing. CPUs are well able of parallel computation using multiple processes and thread in one process. Even in modern browsers this is possible using WebWorkers[1] and soon even lower-level and more parallelized SIMD computations will become available through SIMD support coming to WebAssembly[2].

JavaScript is known to be single-threaded and delegating work to WebWorkers might not be enough for high amounts of data. On the other hand, GPUs are well known to be designed for exactly this kind of data processing. There is today ubiquitous WebGL, the newer WebGL2 that is currently not supported by mobile browsers from Apple (experimental support of some features can be enabled manually, but this is not user-friendly) and at least partial support for WebGPU is starting to appear in the newest browser versions. Conveniently, GPU.js provides an abstraction and a façade to different versions of the WebGL API.

One, two or three levels of for-loops can be hidden by creating functions that process each of the iterations independently and ideally in parallel. These functions are referred to as *kernels* - like the convolution kernels, these are called on each of the processed elements. There is a list[3] of mathematical functions known from JavaScript that can be used in these kernel functions. The only noticable complication is that the kernel functions are transpiled (transparently to the user/programmer) to GLSL and thus standard browser debugging tools are not usable. Partial support is available with GPU.js instace created with parameter *mode* set to `'cpu'`, but it was not exactly straightforward (different error messages in different places, result objects seem to have different APIs etc.).

There also is support for Node.js that could help with some computations on the backend part, but that is beyond the scope of this thesis.

A simple example would be conversion to grayscale where red, green and blue color components are combined with certain weights into a single value expressing color intensity typically displayed as a level of gray somewhere between black and white.

```
const gpu = new GPU();

const grayscaleKernel = gpu.createKernel(function(image) {
  const pixel = image[this.thread.y][this.thread.x];
  return 0.299 * pixel[0] + 0.587 * pixel[1] + 0.114 * pixel[2];
}, {
  output: [width, height],
});

const grayscaleImage = grayscaleKernel(canvas);
```

[1] `https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API`
[2] `https://v8.dev/features/simd`
[3] https://github.com/gpujs/gpu.js/#supported-math-functions

This function parsed and transpiled to GLSL and called on all pixels between [0, 0] and [widht, height] and its result is written in the same position to the output array/texture.

Kernel functions take arguments of various types[1] like numbers, arrays, images, canvas elements or even video element (although this option was not successfully managed by the author to work and had to be worked around through copying video frames to a canvas). Although each of these functions (or, rather, a single run of this function) only returns a single value or a tuple (or sets a pixel color for kernels with graphical output to a canvas), these are then arranged as an one-, two- or three-dimensional array. This result can then be processed in JavaScript or used as input for parallel processing by another kernel function.

The bottleneck to computation speed is sending API calls to the GPU and data transfer between the CPU memory and the GPU memory. To avoid the data transfer penalty, GPU.js provides the pipelining feature where one kernel's result is stored in a texture and passed directly to another kernel as its input, keeping the data in the GPU memory.

[1] https://github.com/gpujs/gpu.js/#argument-types

Chapter 3
Design

The resulting prototype should be able to start and stop displaying device camera's video stream and display a given variant of given structure loaded from the Dowry Town's API. Communication with the backend API requires sending a token with every request as one of the request's headers, but not keeping an authenticated session with login/logout as in the application for accepting the 3D models.

The user interface of this widget is imagined as very simple and minimalistic: a simple image placeholder that would open full-screen (depending on runtime platform's support) with video stream and a specified 3D model loaded from the backend that would display with the video stream in the background.

3.1 Requirements

3.1.1 Functional requirements

- **FR 3D model listing** - viewer lists available modles and their variants.
- **FR Displaying 3D model** - 3D models will be rendered in web browser window.
- **FR 3D model variants switching** - viewer allows user to switch between available model variants.
- **FR Loading models over network** - viewre will be able to load 3D models and their assets over network using HTTP RESTful API.
- **FR Loading progress indicator** - while loading model and referenced assets, there will be a visual indicator showing how much has already been downloaded and how much is still remaining before model can be displayed.
- **FR Model scaling, rotation and translation** - viewer will support standard operations like displayed object rotation, scaling and panning.
- **FR Fullscreen support** - model viewer will be switchable between in-page widget and fullscreen modes.
- **FR Displaying model metadata** - user will be able to display model information like file format, file size, vertex count, textures count, assets list, author name and creation/modification date.

3.1.2 Non-functional requirements

- **NFR Platform** - model viewer will be a JavaScript web application running in a web browser, meaning both desktop and mobile versions of Chrome, Firefox and Safari browsers current at the time of thesis completion. APIs tend to change sometimes, so no guarantee can be made that the app will be runnable in future browser versions.
- **NFR Reliability and stability** - stability will depend on the host platform and correct input. Effort will be made to avoid only obvious crashes in typical cases, the aim of this prototype is not to be bulletproof.
- **NFR Model rendering performance** - speed will be limited by used platform, browser vendors keep competing in improving JavaScript engine performance, but are still limited by single thread. Only the rendering itself (offloaded to used framework) will take advantage of GPU calculations. Optimizations as calculations on GPU are left for possible future improvements.
- **NFR Dowry Towns backend API support** - widget will support Dowry Towns project API for loading 3D models and their versions for different levels of detail of meshes and textures
- **NFR Mouse, touch and keyboard control** - focused widget will be able to accept commands by mouse, touch controls and keyboard.
- **NFR Comfortable and easy use** - model viewer will use controls that are intuitive for users of other 3D software for basic rotation, scaling and rotation using mouse and touch controls. Keyboard controls usage help will be easily accessible.

3.2 Wireframe

The viewer itself is envisioned to be used as a widget on a web page. In a basic form, it should be a clear affordance inviting the user to activate the immersion into the AR experience.

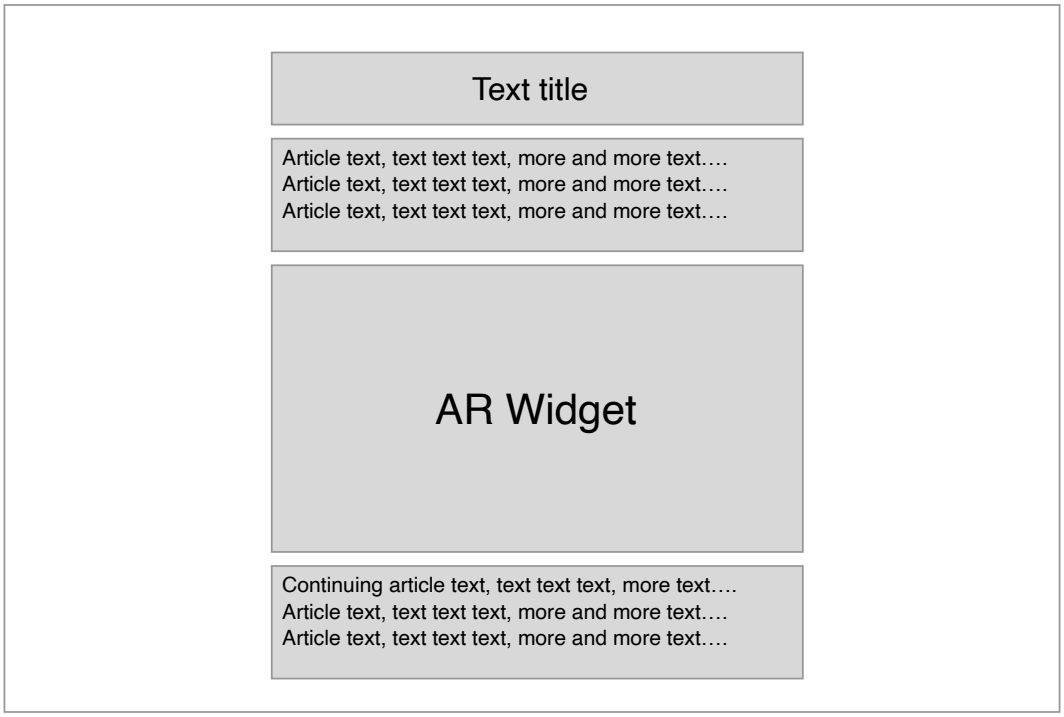

Figure 3.1. AR Widget Wireframe

Several variants were considered - AR Widget being a static placeholder (as seen in the wireframe above) that would either change to a video element with user's real-time camera stream, or toggle fullscreen with the video stream, or contain the video stream immediately, without requiring user interaction.

The version with no require user interaction was abandoned immediately as disturbing and intrusive on one side, and technically impossible on the other as at least Apple's Safari browser requires user interaction leading to video playback. Displaying video stream in the small area of the widget would make sense performance-wise as smaller image requires less computing power, on the other hand it would leave the possibility of unwanted with links and other elements on the web page - and most users would pinch-zoom the element anyway. That leaves the fullscreen switch regime - user reading the text can - should they be interested in more information and ideally with their attention attracted by a placeholder informational image - toggle the immersive fullscreen video with rendered 3D model, fixed to the correct location.

3.3 Model displaying activity diagram

The process of structure model loading from the initiation by user interaction, through loading the 3D geometry mesh and textures, including loading progress indicator, is visible in this image. For the case of simplicity, model URL composition from model's id of the selected structure's variant was left out.

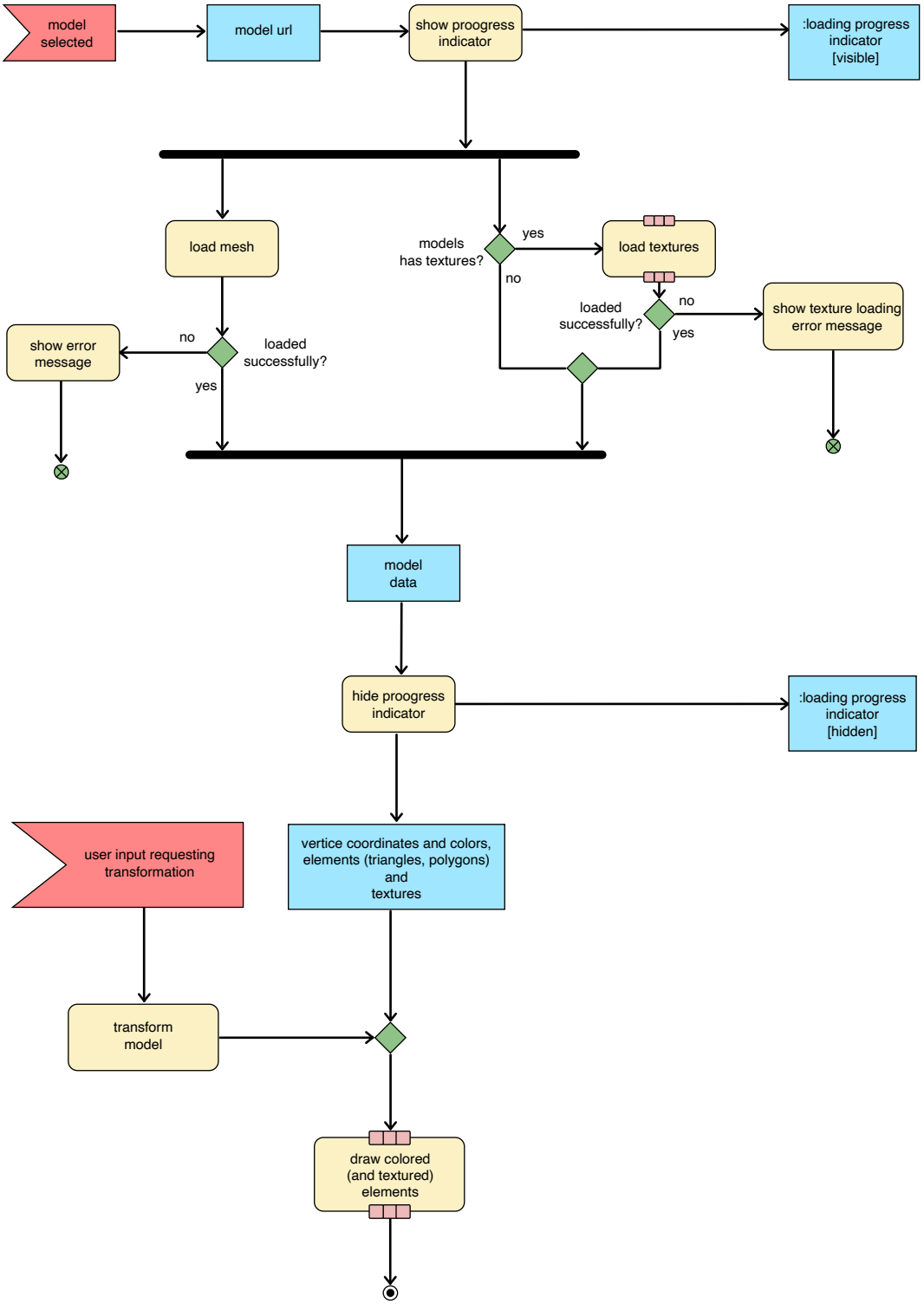

Figure 3.2. Activity diagram of displaying a model in the app

3.4 Components

The AR Widget is planned as a compound component. In the prototype, it should allow selecting from the whole database of structures and their variants, in the ideal final version the structure selection should be automatic, based on user's geographic location and current weather conditions at the location, structure variant should be selectable only where it makes sense - e.g. in case the building substantially changed it looks and is displayed using different geometry.

Components the AR Widget is composed of are planned as follows:

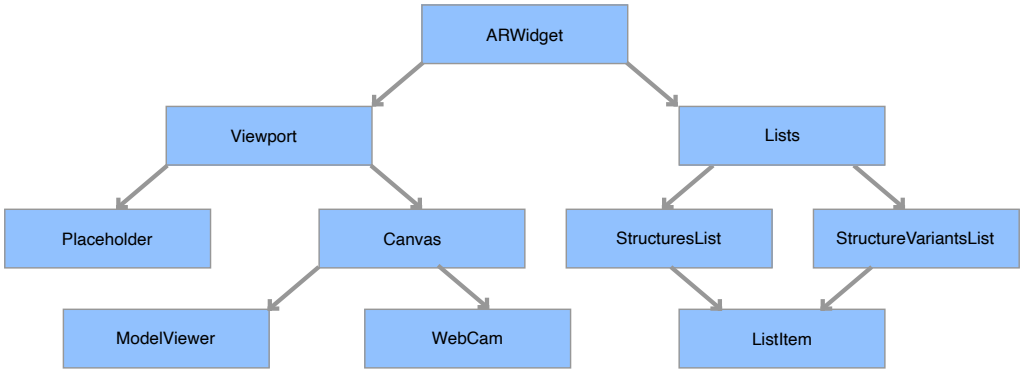

Figure 3.3. Class diagram

Chapter 4
Implementation

The viewer is a JavaScript project bundled by Webpack bundler and using yarn for dependencies management. Webpack bundler allows the developer to focus on the development and not care about paths of projects dependencies. The main HTML file only references the application entry script and Webpack takes care of the rest. Webpack's output is a single (for the sake of simplicity - it be configured for multiple files) where all project files and dependencies are bundled into a single JavaScript files glued together as function bodies.

Technologies except for Three.js were chosen by the author due to at least partial familiarity as the deadline was too close for better analysis.

4.1 Git

Git has been used as the version control as today's de facto standard (not only) in the open source world. There are free services available like for example GitHub or BitBucket on the web where developers can freely host their project's source code and also open source projects like GitLab that let anybody with relevant skill set host a similar service on their (be it virtual or physical) hardware.

Initializing a version control repository in the project's directory is done by running:

```
git init .
```

Very useful feature of distributed version control systems is the possibility to keep project's sources in several separate repositories and exchange code between them. These separate repositories are called *remotes* in Git and having created a repository, one can add its URL to local project's configuration as:

```
git remote add origin \
  git+ssh://git@github.com/martinpucala/vmck-viewer.git
```

The simplest and ideal workflow would be just committing change sets to the repository and pushing them to the remote repository as a backup:

```
git add file1 file2 file3
git commit -m 'Add new files x.js and y.js providing feature XYZ'
git push origin HEAD
```

Git also supports (several types of) branching, allows reordering commits, copying commits between branches, editing new and even older commits, reordering them and so on. Description of these features is beyond the scope of this thesis and thankfully there is great documentation, free and paid courses online or taught at the university.

4.2 Yarn

Yarn is a faster alternative to *NPM* as a dependencies manager for JavaScript and TypeScript projects.

Adding and removing dependencies to project is quite straightforward:

```
yarn add webpack
yarn remove webpack
```

Dependencies used only during the development (like the dev server watching and reloading code as necessary, linters watching syntax and project's coding style) are installed with the `-D` switch:

```
yarn add -D webpack webpack-cli webpack-dev-server
```

There is also the option of configuring script aliases where running a simple command like:

```
yarn start
```

runs a command specified in projects configuration file named `package.json`. This is typically used for monitoring changes in project's directory to trigger re-bundling the code and reloading page in the browser to save the developer from dull repetitive steps of doing this manually after each change. Another benefit is that especially with bigger projects his allows the bundler to re-process only the files that changed since previous processing. This is similar to the `make` command, but it also saves the time of re-running the bundler (which itself is interpreted code), re-building the dependency graph, loading the files etc. - because they are still loaded im memory in their processed form.

Yarn creates and manages a file with the versions of currently installed dependencies, this is called `yarn.lock` and it should not be changed manually, yet it should be committed as to keep the same package versions everywhere across different project members and computers (developers, production, testing environment etc.).

4.3 Webpack

The JavaScript ecosystem is evolving fast. From the times where the purpose of scripts on web pages was to open and close menus on hover it situation has matured to today's state of complex projects running both on backend and in web browsers. Complex projects are typically built on top of frameworks and multiple utility libraries. During the years, several bundlers and standards for JavaScript modules have evolved (e.g. AMD, CommonJS)[1]. Instead of adding a `<script>` HTML tag or even several of them every time a new dependency is added to the project, bundlers allow specifying how project's code should be bundled with its dependencies - it takes project's entry point script and recursively its imported dependencies, wraps them into functions, assigns them unique identifiers and combines them into a single bundle file that contains all the code and can be loaded by the page as a single file.

Building project into a bundle is done using command:

```
webpack --mode=development
```

[1] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules`

Project's code base directory can be watched for changes and rebuilt after each save using the `--watch` switch:

```
webpack --watch --mode=development
```

Webpack also provides a built-in development web server:

```
webpack server --mode=development
```

All these commands typically are saved with script aliases like *build*, *watch* or *start* in the project's *package.json*.

4.4 Babel

Babel is a tool used to transpile latest EcmaScript and JSX language features into plain JavaScript to make the bleeding edge available to developers as soon as possible and still keep it runnable in older browsers. The configuration used was adopted from a tutorial article[1] about minimal Webpack+Babel configuration for React.

4.5 React.js

React is probably today's most widespread frontend framework. Released in 2013 by Facebook, it is today quite mature, yet still evolving. Maybe the biggest change in recent years is the move from class components to functional components with hooks. Probably its main goal and benefit for developers is the declarative description of components as projects clearly divided into components are typically easier to maintain and especially navigate even for programmers new to a project, where one can expect at least some known structure. The workflow in React projects is simple - a component is given some initial properties and is expected to render itself i.e. put together a markup written in JSX markup language which is a HTML-like markup language for describing component's output. More complex components typically inside use other components, keeping the code complexity from growing too much. In functional components, hooks are a mechanism for handling asynchronous data updates. React has a very good and quite verbose documentation explaining its concepts on its website[2].

4.6 Three.js

Three.js has been the most popular 3D graphics framework on the web for several years already. It has loaders for various 3D formats and provides very useful abstractions to the low level WebGL API. This helps creating simple projects using 3D graphics on the web in just a few lines.

Integrating Three.js and React code is well described in two articles[3][4] by Alexander Solovyev. Written in 2019, however, some adjustments were necessary to make the ideas work with functional React components and hooks.

[1] https://www.robinwieruch.de/minimal-react-webpack-babel-setup
[2] https://reactjs.org/docs/
[3] https://codeburst.io/react-16-three-js-integration-tips-2019-b6afe19c0b83
[4] https://blog.bitsrc.io/starting-with-react-16-and-three-js-in-5-minutes-3079b8829817

Endless loops in JavaScript cause the page they are running on become unresponsive. Furthermore, there's no point in re-rendering animated scene on a page that is not visible (tab begin in the background, for example). Rendering in loop is therefore done using the `requestAnimationFrame`[1] function that schedules the function it is given as its argument to be called only just before the frame needs to be rendered and pauses when the tab with the page it is running in gets into background (different tab in the same window is activated).

4.7 Web camera

`react-webcam` provides web camera access to React applications. It supports video constraints for specifying camera and desired resolution and enabling and disabling camera.

4.8 Full-screen toggle

Full-screen toggling is handled using the `react-fulls-screen'` package. It allows hooking to other components' events and switch a chosen component to to full-screen mode and notify whenever the fulls-screen mode is entered or exited.

4.9 OrbitControls

Three.js package contains an example with intuitive controls. After pressing a mouse button or touching device's touch screen, dragging causes rotation around the axis perpendicular to the lines running from `OrbitControls`' *target* point and the points of previous and current contact of the pointer. Rotation direction matches the pointer direction, at least on shorter distances (meaning it does not work like rotating with a handle).

The constructor takes reference to the camera and the element on which it is supposed to hook its event handlers.

```
setControls(new OrbitControls(camera, element))
```

Storing the instance in a component state variable ensures is remembered at initialization time and reused on subsequent renders of the component and not re-created on every re-draw.

4.10 TrackballControls

Same as the `OrbitControls` package, but seems more intuitive.

[1] `https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame`

4.11 Model loader

Three.js tool set provides 3D model loaders for several formats. Having the database of projects backend server full of .OBJ models, the `OBJLoader` is the ideal. Unfortunately, the .OBJ format needs its accompanying .MTL files with material descriptions containing values like material's ambient, diffuse and specular coefficients and, more importantly, texture file names. Testing database only contained .OBJ files and after randomly picking several, all of them had textures assigned in the database, but none of them had .MTL files assigned (not even texture coordinates were found in the few models examined). Had this been discovered earlier by the author, measures could have be taken like preparing and uploading models with textures.

There was a slight complication with the authentication and loading resources from through REST API in combination with the `OBJLoader`: the API needs each request to be accompanied by the `Authentication` header set to a valid token, otherwise the server responds with 401 Unauthorized response. To mitigate this, objects had to be loaded manually, response read as a `Blob`[1], converted to object URLs using `window`'s `createObjectURL()`[2] method and only then passed to `OBJLoader`.

With models referencing textures this will get complicated further by the fact that the `OBJLoader` loads model's dependencies automatically - and those requests will be missing the `Authentication` header. This can be fixed using Three' `LoadingManager`[3] with a good example of this in the documentation.

4.11.1 Loading progress indicator

A simple component for displaying the progress of model loading has been implemented, taking an input of 0 to 1. Numeric value renders a simple loading progress bar.

[1] `https://developer.mozilla.org/en-US/docs/Web/API/Blob`
[2] `https://developer.mozilla.org/en-US/docs/Web/API/URL/createObjectURL`
[3] `https://threejs.org/docs/index.html#api/en/loaders/managers/LoadingManager`

Chapter 5
User Testing

There are several simple scenarios to be tested to find out whether the AR widget is intuitive enough for general use. All of them should be ideally done with four or five participants and in a location points in the environment that are recognized and stored in the project's database. All user are assumed to be keen on sightseeing tourism and be routine smartphone users. There should be ideally two people besides the participant. First, a facilitator guiding the user through the process and instructing him according the testing scenario, but avoiding other interaction or helping (unless the user gets completely stuck - which would be a good indicator of a usability problem and the facilitator should intervene only to get to the next point). Second person (could be replaced by a camera recording user's interactions and maybe facial expressions to catch confusion - testing participant must be informed that he/she is being recorded and agree with it) is only a silent observer, taking notes during the process to prevent the need of re-watching the video recording for evaluation purposes (necessary in case of only a facilitator is present or when a closer look at some issues is need).

Test scenarios:

- entering AR visualization
- leaving AR visualization
- selecting structure variant from the list
- rotating structure
- zooming structure

5.0.1 Entering AR visualization

Estimated duration time:
less than 1 minute

Test reason:
Testing the affordance - whether it is obvious to users and really invites them to open it.

Starting point:
Page is scrolled so that the clickable placeholder is visible.

Ending point:
ARWidget is open in fullscreen mode.

Test participant instructions:

- Could you find a way to start the structure visualization?

Expectation
Potentially after some up and down scrolling, the user should find and tap or click the widget's trigger.

5.0.2 Leaving AR visualization

Estimated duration time:
less than 1 minute

Test reason:
Testing fullscreen mode closing - whether users can find a way to get out of the fullscreen mode back to visited site.

Starting point:
Fullscreen mode is open, video stream is displaying the camera video feed.

Ending point:
User exits the fullscreen mode and is back on the visited page.

Test participant instructions:

- Could you now exit back to the page ?

Expectation
User finds how to close the fullscreen mode within a few seconds.

5.0.3 Selecting structure variant from the list

Estimated duration time:
1 minute

Test reason:
Testing whether different structure variant is easy to find for users.

Starting point:
Fullscreen mode is open with structure that has multiple variants available.

Ending point:
User opens structure variants list and selects a structure.

Test participant instructions:

- Could you try to select different version of this structure?

Expectation
User notices structure variants list and after a few seconds selects one of the variants.

5.0.4 Rotating structure

Estimated duration time:
less than 1 minute

Test reason:
Testing controls intuitivity. It should be easy to find out how to rotate the displayed structure.

Starting point:
Fullscreen mode is open, structure is displayed.

Ending point:
User presses mouse button or touches the screen and drags in the direction they want the structure to be rotated.

Test participant instructions:

- Would you know how to rotate the structure if you wanted to see it from the side?

Expectation
It should not take more than a few seconds of thinking for the user to try rotating the displayed structure.

5.0.5 Zooming structure

Estimated duration time:
less than 1 minute

Test reason:
Testing controls intuitivity. It should be easy to find out how to scale/zoom the displayed structure.

Starting point:
Fullscreen mode is open, structure is displayed.

Ending point:
User presses mouse button or touches the screen witch two fingers and pinch-zooms to change the scale of the displayed structure.

Test participant instructions:

- How would you zoom in or out the structure you see?

Expectation
This can be potentially tricky in case the user is not an advanced user. For frequent smartphone users this should be a familiar gesture.

Chapter 6
Conclusion

Leaving aside the fact that the code is only partially complete and far from was envisioned by the author when starting working on this project, it got much further than originally expected or imagined - and yet, it still didn't get as far as it was hoped for. Having to collect and sort out relevant information with only a very superficial overview made the estimates and planning rather challenging. Even with the great advantage of prior works on this project, each steps forward was preceded by many leading nowhere - not even into this text (real-time Canny and Sobel edge detectors and hours of computer vision lectures, for example) - on the hand because was not deemed directly useful (or simply not understood enough to be of any use), on the other hand because the effort to save time and eventually find a path that would lead in the next steps that can be seen towards the goal.

All this work, however, is not not visible at the time of writing this and is only attached as work-in-progress code that is not used in the prototype, which, from user's standpoint looks very basic and trivial without the automatic positioning in the video stream. One could really say the visible part is only the cap of and iceberg in this case.

Having to implement the algorithms from scratch without depending on computer vision frameworks years in development proved to be even more time consuming than expected. Still, the know-how amassed during the research invested lead to the point where finishing the work-in-progress code submitted with this thesis (using the overview and references collected herein) should lead to a working application and open wide area of possibilities and projects that require nontrivial time and effort to put together and grasp.

Both goals are partially fulfilled, the UI prototype, although with bugs, demonstrates the vision where it is imagined should lead after the pose estimation part is finished - opening an article at a sightseeing site and directly from there be able to see - be it historical structures as in this case, or commercial information in a restaurant, or some additional information in books, magazines, restaurant menus(!) etc.

If nothing else, then then merely the realization, that one can read and comprehend published scientific papers was enlightening. What more, suddenly even the often by-many-hated mathematical subjects suddenly found their use when reading how only decades (years even) great minds used it to describe solutions to problems unimaginable to solve without (Moravec, Harris and Shi-Tomasi corner detectors, ORB feature detection and description, projection, systems of linear equations...). Problems like, especially with visual demonstrations, could certainly attract people scared off by the raw and abstract ways of teaching in the mainstream today.

It was certainly naive to expect to come from zero to a working application like envisioned here. Still, the author believes it would be worth repeating. Taking risks, failing, trying again and still pushing wild fantasies forward towards reality (with the ever growing strength of amassed information and experience).

Even after finishing this prototype, there is still lots of work left for the future. Be it already in the text mentioned lighting estimation (calculated from geographical location, date and time and position of the Sun and current weather or maybe from the gradients of smooth surfaces in the surrounding environment), shadows (both on the projected model and cast by it, each for different light source types), estimating correct occlusion of the projected and real-world structures, accuracy of the projection can be worked on using various optimization techniques (Levenberg-Marquardt, probably not-so-fast stochastic gradient descent) minimizing the reprojection error. Even just measuring and evaluating the effectiveness of used algorithms under different lighting and weather conditions could maybe be extended to be enough work for a standalone thesis.

After finishing parallelized ORB and the, there is the challenge of implementing - now its patent is expired and doors are open even for its commercial use - the SIFT algorithm runnable on GPUs. For now, hoping it would still do some help with defending this thesis, the next steps seem to be continuing work on features and fixing bugs here.