A Modular Approach for Creating Web Applications with 3D Geology Using React and X3DOM

Tomas Fjugstad Eriksen

Master's thesis in Software Engineering at

Department of Computing, Mathematics and Physics
Western Norway University of Applied Sciences
Department of Informatics,
University of Bergen

June 2019

Supervisors: Harald Soleim, Atle Geitung and Daniel Patel

Abstract

The thesis details the work for creating a web application that allows a user to see a visualization for geological data like topography, seismic data and well measurements from the subsurface. This application uses the JavaScript web library React, and the X3DOM framework. It is built upon a previously built application using the JavaScript framework Angular instead of React, which had led to some issues. By creating the application using the React library the thesis aims to avoid these issues, and develop a more modular structure. In addition, the work in this thesis will try to further develop new and improve existing functionalities such as navigation and possibility to toggle visualization of element.

Acknowledgment

I would like to thank my supervisors Harald Soleim, Atle Geitung and Daniel Patel for assisting the master thesis, with valuable advice for writing the thesis and developing the application. I would also like to thank my friends and family for support.

Table of Contents

Table of Figures

Table of Code Listings

Glossary

API – Application Programming Interface
BVH – Bounding volume hierarchy
BVHRefiner – A X3D component that refines and loads hierarchical data dynamically.
Canvas – HTML5 canvas element
CMR – Christian Michelsen Research
Component – A JavaScript class or function that returns a React element containing a section of the UI
ComponentDidMount – A React lifecycle method that will run when a React component has been created an inserted to the DOM
ComponentDidUpdate – A React lifecycle method that will run when a React component has been updated
CSS – Cascading Style Sheet
CSV – Comma-separated values
DOM – Document Object Model
FOV – Field of view
GDAL - Geospatial Data Abstraction Library
Geometry – A graphical shape created from a set of triangles
GUI – Graphical User Interface
HTML – Hyper Text Markup Language
ISO – International Organization for Standardization
JPEG/JPG – Joint Photographic Experts Group
JSON – JavaScript Object Notation
JSX – JavaScript XML
Lifecycle methods – Methods in React that will run depending on if a component is changed, created or deleted
LOD – Level of detail
MERN Stack – Application stack using Mongo, Express, React and Node.
Modal – Dialog prompt / dialog window
OCP – Open-Closed principle
Open source – The source code is available for free for everyone to use
OpenGL – Open Graphics Library: An API used for 2D and 3D vector graphics rendering.
Performance - How quickly a web page loads necessary data
PNG – Portable Network Graphics
Props – Short for properties, is defined as a parameter when a component is defined.
Ref – Short for reference, used in React as a reference to the DOM or an instance of a component.
RGB color model – A color model with three number values representing the colors red, green and blue
RMS File – Textual file containing data for a well log
Segment/Subwell – The section between two coordinates of a *well*
Scene – A defined area within a web browser where 3D object created from the X3DOM framework can be placed
Slice – The visualization of a slice of seismic data
SOLID – An acronym representing five design principles. These are the single responsibility, open-closed, Liskov substitution, interface segregation and dependency inversion principles.
SRP – Single responsibility principle
State – A variable in React used for determining how a component is rendered, useful concept for making dynamic and interactive components.
TREE – Dataset structure that can be used with BVHRefiner

Transformation – Term in computer graphics for defining the translation, rotation and/or scale for a graphical object
UI – User Interface
URL – Uniform Resource Locator
UTM – Universal Transverse Mercator coordinate system
Viewpoint – A node in X3DOM that specify the position and orientation from where the camera is placed within a scene
VIRCOLA - Virtual CO2 Laboratory project
VRML – Virtual Reality Modeling Language
WebGL – Web Graphics Library
Well – A visual representation of data from well logs
Well logs – Files containing data gathered from drilling in the subsurface
WMTS – Web Map Tile Service
X3D – Extensible 3D Graphics
X3DOM – An open source framework and runtime for declarative 3D scenes for Web Browser
X3DOM node/element – A predefined element within the X3DOM framework that is used to represent a part or aspect within a scene, such as an object, material or light.
X3DOM tag – A tag that are used to represent and implement a particular X3DOM node
XML – Extensible Markup Language

1 Introduction

This master thesis is about the work regarding the development of a web application used for visualizing 3D geology. In particular the application can show the 3D topography of Svalbard and data gathered from the subsurface. In the development of this application two tools have primarily been used. These are the JavaScript library React and the X3DOM framework. This thesis continues from the work provided by Øystein Malt. His's work is presented in the Master Thesis "Using 3D functionality available in current web-browsers to create and visualize geological models" [1].

In Malt's work a solution for a web application that visualize 3D geology is provided. The application also used X3DOM framework. However, the application used the JavaScript framework Angular [2], instead of React. Using Angular led to some problems, such as an error that could occur when handling events [1]. To work around these problems resulted in code that was hard to maintain and extend. For these reasons this thesis will investigate if it is possible to make a better design by using React instead of Angular.

This thesis will also explore whether React can provide a more modular solution, by having the code divided into different components. By making the application and its component modular we aim to make all the components independent from each other, and easily reusable within the application, and possibly even in other application. The general idea is to have each component focused on visualizing a particular type of 3D object by using X3DOM. An example of an object could be a 3D sphere. The thesis will also offer general guidelines for how a component should be implemented.

During the research in Malt's thesis he investigated whether React could be used to create his application. He found an issue where React would ignore code, that were defined by X3DOM. This was the main reason why Angular was chosen over React in his thesis. The issue has later been fixed in version 16 of React [3], which were not available during Malt's research. The update provides a good reason and opportunity to reinvestigate if React could be used to produce a more desirable solution. The issues that occurred when using Angular, and the React update are two major reasons for the decision of why this project reexplores the possibility of using React.

There are also several other benefits for finding a solution with React. React is currently a very popular JavaScript library for front-end development [4]. This popularity allows for a more active community and a more well-defined documentation. An active community and a good documentation can make it easier to learn and find answer for common issues.

In the work for this thesis an application has been built, where some of the features from Malt's project have been rebuilt using React instead of Angular. This included the 3D topography of Svalbard, slices of seismic data and well logs. Some of them were created by a similar design, while others have been implemented in a more efficient way. In particular, the ability to navigate and toggle the visibility of elements have been improved. Finally, the thesis has explored whether volume rendering from the X3DOM framework can be used effectively to visualize geological data.

1.1 Project inception and motivation

The idea behind the work of the thesis came from Christian Michelsen Research (CMR). In addition, CMR contributed supervision for the work. The origin of the work is an earlier project that was conducted by CMR called "Virtual CO2 Laboratory project (VIRCOLA). The vision for that project is quoted below from CMR home page:

> *The vision of the Virtual CO2 Laboratory project (VIRCOLA) was to develop a data platform and methodology facilitating improved data utilization and work processes, leading to better understanding of storage capacity, injectivity and long-term confinement of CO2.*
>
> Christian Michelsen Research (2015) [5]

The researchers from the VIRCOLA project tried to use existing geological visualization solutions. However, they encountered issues with them for several reasons. These included a high learning curve and a high cost. Furthermore, the existing solutions required installation to be used. In addition, the solutions were in general more suited for detailed interpretation, rather than a simple visualization for getting an overview of the data. The researchers wanted an easy-to-use visualization tool for quickly navigating through data, rather than one meant for detailed analysis. Because of this, CMR created a 2D application that was accessible on a web browser. Creating a web solution would result in reduced costs and no longer require an installation. This application was able to visualize a 2D map of Svalbard and seismic data and well logs in the subsurface. An image of this application can be seen in Figure 1. Sometime after the initial solution, Øystein Malt made a 3D web application with Angular, where it was possible to visualize a topology of Svalbard, well logs, and slices of seismic data in 3D. However, the code needed to produce this was not optimal. Furthermore, there were significant issues concerning the ease of navigation and handling events, such as when a user clicks on an element within a scene, and an event should trigger. The application from this thesis, is based on the findings from Malt's application.

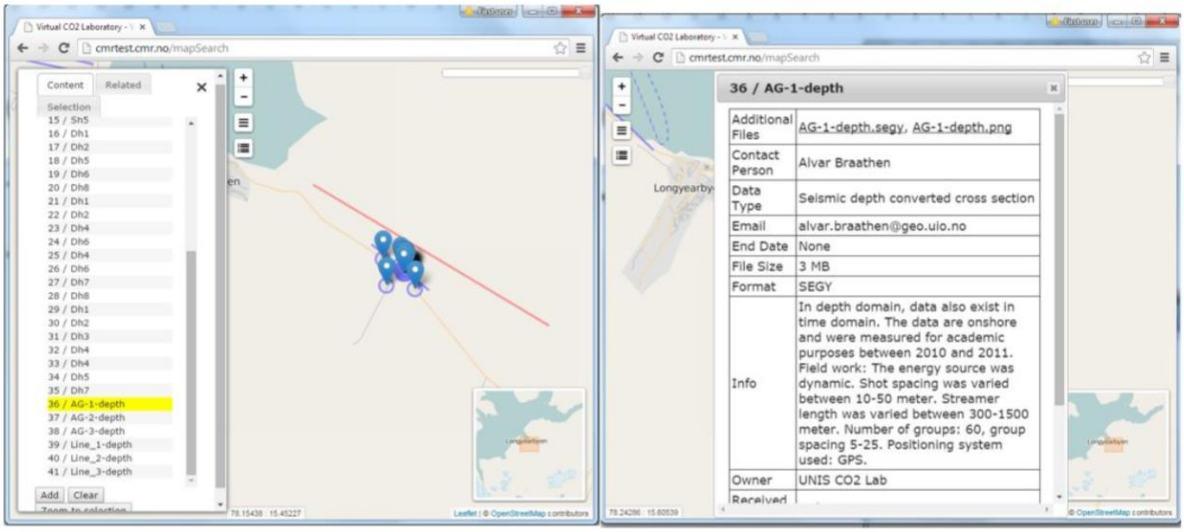

Figure 1: 2D Application from CMR's VIRCOLA Project. From: Ø. Malt. Using 3D functionality available in current web-browsers to create and visualize geological models. 2017. Master's thesis [1]

1.2 Goals

The thesis consists of a set of goals, these are listed below.

Goals:

1. Recreate a web application for visualizing the topography of Svalbard using React (instead of Angular)
2. Follow modular design patterns to make elements in the application independent and reusable
3. Develop a general structure for implementing React components that can visualize 3D objects
4. Explore possibility for volume rendering
5. Improve the navigation compared to Malt's solution
6. Improve ability to toggle elements compared to Malt's solution

The first goal consists of recreating functionalities from the web application, created by Malt. This recreation will use the React web library, instead of Angular. By following this goal, the thesis aims to research whether it is possible to develop such an application with React, and if it can provide a better solution, by fixing the issues that occurred when using Angular. Recreating the functionalities from Malt's application, would require the new application to have the ability to visualize a terrain of Svalbard, well logs, and slices of seismic data. Furthermore, it would require the ability to toggle and upload well logs and slices of seismic data.

As explained by the second goal, the application should provide a modular design. By doing this, each element in the application should be independent from each other and easy to reuse. In this context an element could reference what is needed to render a specific 3D object. This could be a simple shape such as a sphere, or something more complex such as a terrain. By making the elements independent, they could easily be extended or changed, without worrying about other unexpected changes in the application. In, addition it could make it easier to use one element in another application. By making the elements reusable, it will make it easier to define multiple instances of them. This can be useful for certain situations, for instance if the application contains a list of well logs that should be visualized.

The third goal consists of making a general guide for how an 3D object within a React application can be made. This can be useful to highlight general techniques for how visualize and interact with 3D object created from X3DOM framework. It also provides developers with an easy way for how they can make their own React components containing 3D objects.

The three last goals focus on improving and adding functionalities compared to the Malt's solution. First the thesis will explore the possibility for supporting rendering of 3D volume. This could be used to showcase a set of slices of seismic data that are closely placed together. The reason why this can be interesting is that it can be useful for a researcher to look at the information from a set of slices that are closely placed together. These slices would form a volume. A user should be able to look at each slice individually by using a slider. The two final goals focus on improving the user ability to navigate within the scene in the application, and toggle whether an element should be visualized or not. These features make it faster and easier for a user to explore all the 3D data in the application.

1.3 Research questions

Based on the thesis' goals, the thesis has identified certain topics that can provide avenues for research. These are mostly related to the second and third goals, which focus on whether the React library and X3DOM framework can provide a modular implementation and an easily useable structure for designing components containing 3D objects. The first goal delivers the opportunity for the second and third goal to be explored. Below is a list of research questions this thesis aims to answer:

- RQ1: "If possible, how can React library provide a modular approach for visualizing 3D objects and 3D geology on a web browser"

- RQ2: "What are the benefits of using React library to make components containing X3DOM elements"

- RQ3: "If any, what are the benefits of using React library together with the X3DOM framework, compared to doing it with Angular"

These questions will be discussed in chapter 6 and will be used to evaluate the results of the thesis's work. RQ1 consist of several parts, as it is used to determine if it is possible to create an application using 3D objects in general, as well as 3D geology. By creating an application using 3D geology, it would also show it would be possible with 3D objects in general. In addition, this question also focuses not only if it is possible, but how it can be achieved. RQ2 focuses on what are the benefits of achieving the desirable result of RQ1. The thesis will use RQ2 to determine whether using React led to a desirable result. RQ3 build further on RQ2 by comparing the benefits of React to the solution using Angular implemented by Malt.

1.4 Applicability beyond the thesis

This thesis provides a solution for visualization of geological data from the subsurface. Visualizations of this kind are useful in many fields including ground-water mapping, oil and gas exploration and CO_2 storage. Because of this, this project can be of interest for companies or other projects involved in the fields mentioned. In particular, the topic of CO_2 storage is interesting in relation to Svalbard, as research conducted by Longyearbyen CO_2 Labs shows that CO_2 storage is possible on Svalbard [6].

It is possibly to extracted part of the work of this thesis and used in other applications. The application consists of independent component. For example, the implementation of the visualization for slices of seismic data and well logs (for details, see 4.4 and 4.5) can easily be transferred to another React project if it consists of a scene from the X3DOM framework (see 2.1). The work from this thesis can be applied and built upon for anyone who wants a simple way to add and structure 3D objects in a web browser, without using any external plugins. This is true even if the project is unrelated to geology. Subchapter 5.4 goes into detail on a general structure for using X3DOM together with React. This subchapter also presents an example of a potential additional use.

1.5 Related works

One of the most related works for this thesis is the previous thesis that worked on the same topic, which is "Using 3D functionality available in current web-browsers to create and visualize geological models" by Malt [1]. Malt also presented his results in a report [7] and a presentation [8].

In subchapter 1.1 it was mentioned that CMR had tested a few applications for geological visualization. These are SKUA from Paradigm [9] and Petrel [10]. Both are desktop applications specifically made for visualizing subsurface data. According to Malt, CMR evaluated these two applications in an article "VIRCOLA – Review of Data And Visualization Platform" [1]. In the article CMR had the conclusion that both tools had the ability to visualize all the data from their project. However, both SKUA and Petrel had a big learning curve, was costly, required installation and were unable to run in a web browser.

One related work concerned on how to represent 3D structural geological models. This is an article "Formal representation of 3D structural geological models Computers & Geosciences" by Wang et al [11]. There was also an article by Arbelaiz et al [12]. Here the researchers have integrated functionalities for volume rendering to the X3DOM framework. It also provides an explanation for how developers can use these functionalities for volume rendering in a project. The data used in this research articles are of medical nature, rather than geological. This thesis implements some of the contribution to the X3DOM framework from this article, and research whether they can be used effectively in relation to geological setting.

There are other currently developed libraries that are working on the ability to create 3D objects within React. One of them are called *react-three-renderer* and are the using Three.js (see 2.5) [13]. This work proves that there is an interest in the ability to create 3D objects within React. However, *react-three-rendere* is not compatible with newer version of React and it does not support all the features of three.js. [13]

Articles or work exploring the possibility of using X3DOM and React together were not found, despite efforts while researching relevant literature.

1.6 Thesis outline

1. Introduction

This chapter briefly introduced the thesis and explained the motivation and goals behind it.

2. Background

This chapter goes into the concepts and tools necessary for understanding the solution in this thesis.

3. Application Overview

This chapter gives a quick overview of the features in the application and how it is presented on a web browser.

4. Solution

This chapter will go into great detail of the implementation and design for the solution of the application.

5. Results and Discussion

This chapter will discuss the results of the solution provided in the previous chapter.

6. Conclusion

This chapter will present a brief conclusion of the work of the project and provide a summary of the thesis.

7. Further Work

This chapter offers suggestion for further work that, can improve or be useful for the application.

2 Background

In this application, two tools have primarily been used. The first is X3DOM for the ability to make 3D objects, and the second one is the JavaScript Library React that are used for creating easy user interfaces (UI) on a web browser. React is also used for wrapping the code within reusable components. The following subchapters will detail these two primary tools, as well as some additional tools used in the implementation of the application. In addition, it will go into detail of some alternative tools that could have been used.

2.1 X3D and X3DOM

Extensible 3D Graphics (X3D) is a royalty-free, open-standard file format and run-time architecture, which are used to represent and communicate 3D scenes and objects [14]. X3D can render graphics that are in high quality, real-time and interactive. X3D has its origins from Virtual Reality Modeling Language (VRML), but has later been evolved into a ratified part of the ISO standard. X3D gives a system the ability to store and retrieve 3D scenes for rendering. X3D has support for extensible markup language (XML) integration, which allows it to be usable with Web Services and across different platforms. [14]

X3DOM is an open-source JavaScript framework and runtime. It is used for implementing declarative 3D scenes on a Web page. The name X3DOM is a mixture of the two abbreviation X3D and DOM (Document Object Model). X3DOM allows a specific subset of the X3D standard, which can be used as a description language for 3D content in a web page. It consists of a set of predefined nodes (also referred as elements). These nodes are referred to as an X3DOM node and represent a particular part or aspect within a scene, such as 3D geometry, material used on an object and light sources. Each node has a set of fields, that are used to define certain properties or behaviors of a node. These could be anything from the color, size, rotation, transparency and so on. These nodes are created in a very similar way as a regular HTML element, where it is defined between a start and an end tag. The fields are defined within the start tag. Just like HTML elements, a X3DOM node can be changed with DOM operations.

Within each node (between the start and end tag) other nodes can be defined, which can affect the node, that they are placed within. This structure forms a tree or a graph which is often referred to as a scene-graph. [15] Code Listing 1 shows an example of an X3DOM node, called *material* which contains a *diffuseColor* field.

```
<material diffuseColor='0 0 1'></material>
```

Code Listing 1: A X3DOM node

X3DOM is used to avoid low-level development and can hide details concerning graphics rendering in a high-level, declarative syntax. In other words, X3DOM does not require a huge amount of knowledge about computer graphics and only requires basic knowledge of HTML and DOM. It can be a helpful tool for web developers that need some graphical components in their web application [16]. This makes X3DOM more desirable for this project compared to something like three.js or OpenGL. Another benefit when using X3DOM, it that it does not need any additional plugin for a web browser. X3DOM only uses standard browser technologies like HTML5 and WebGL. Most web browser has native support for these technologies. This includes web browser such as Mozilla Firefox and Google Chrome [17]. Below there is a basic example, that displays a blue box in a 3D scene on a web page by using X3DOM in a HTML document. Code Listing 2 shows all the X3DOM nodes needed to create a box object. In addition, it the HTML documents has two references to the X3DOM framework that are required to use it. These are called "x3dom.js" and "x3dom.css" and are necessary to get access to the X3DOM framework. Both the references are found by a uniform resource locator (URL), that are placed within a script and a link HTML tag. These tags are placed inside the header of the HTML document.

Within the body of the HTML document, all the X3DOM nodes are placed. First, there is an X3DOM context, created by an *<x3d>* tag. All the other tags from the X3DOM framework must be placed inside the *<x3d>* tag. There also needs to be a scene, where all the X3DOM elements are contained. This is done by the *<scene>* tag. Inside the *<scene>* tag, there is a *<shape>* tag. This can be used to make a basic 3D object like a box or a sphere. Inside the *<shape>* tag there are two nodes one that handles the appearance of an object, and another that specify the geometry type. In this case the geometry is a box (or a cube). The appearance consists of a *material* node with the field *diffuseColor* consisting of the value '0 0 1'. This represents the RGB code for the color blue. The Figure 2 on the next page, shows the result of this code. It is also possible to put additional fields inside some of the tags in this example to specify certain attributes. For example, the *<box>* tag has a *size* field or the width and height attributes for the scene ratio. Usually a *<transform>* tag can be used to specify where in the coordinate system within the scene an object is placed. Because this tag has not been used in the example, the box has been placed in the middle of the scene with default coordinates '0 0 0'. A list of these attributes is easy to find in the X3DOM documentation.

```
<html>
  <head>
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <title>X3DOM page</title>
    <script type='text/javascript' src='https://www.x3dom.org/download/x3dom.js'>
    </script>
    <link rel='stylesheet' type='text/css'
        href='https://www.x3dom.org/download/x3dom.css'>
    </link>
  </head>

  <body>
    <h1>X3DOM Example</h1>

    <x3d width='500px' height='400px'>
        <scene>
            <shape>
                <appearance>
                    <material diffuseColor='0 0 1'></material>
                </appearance>
                <box></box>
            </shape>
        </scene>
    </x3d>

  </body>
</html>
```

Code Listing 2: X3DOM code needed for creating a box geometry within a scene

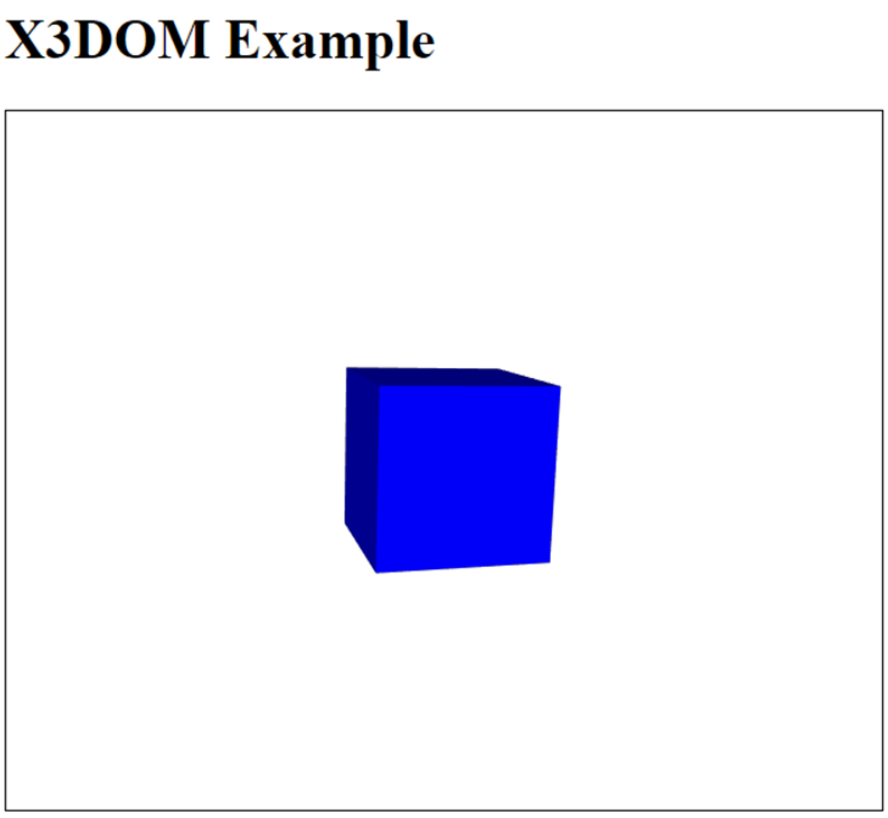

Figure 2: Result of Code Listing 2, showing a box in a web browser

2.2 React

React.js is a JavaScript library developed by Facebook [18]. It is used for making the dynamic and interactive UI for web applications. React allows a developer to create components that can easily be reused and focus on a particular task. A React component is created by either defining it as a function or a class. Making a function is a bit simpler, but a class offers more functionality. When making a React class requires to write "extends React.Component" in the declaration as shown in the Code Listing 4. Components is a way to divide the UI into independent reusable pieces. A React class is required to have a render method which is used to display elements on the webpage. The render method has a return that is written in JavaScript XML (JSX).

JSX is a syntax extension for JavaScript. It is often used in React project. It allows a variable to contain HTML-tags. In addition, JSX allows a variable, a function that return one, or conditional statements to be written within curly brackets [19]. Code Listing 3 shows an example of JSX. Here there are two variables called *name* and *greeting*. The *greeting* variable takes advantages of JSX by using a *<p>* HTML-tag and wrapping the *name* variable within curly brackets. If this element is rendered on a web browser, it would say "Hi, nice to see you Alice".

```
let name = "Alice"
let greeting = <p>Hi, nice to see you {name} </p>
```

Code Listing 3: JSX example

JSX makes it easier to work with UI elements within a React components and allows React to display more helpful error and warning messages. It also works well together with other feature of React, such as *states* and *props*.

A React class has a constructor. In the constructor it is possible to declare one or more *states* and bind methods to the component. A *state* in React is a variable in a React component, that can be changed with a *setState* method. When some change happens to a *state* the component will be re-rendered in the web application automatically. *States* are useful to make an application interactive for a user. Another feature of React is called *ref* (short for reference). A *ref* is used to access React elements or DOM nodes that were created in a render method [20]. And finally, there is *props* (short for properties), that are used to define data in another component from a parent to a child component. Unlike a *state*, *props* cannot be changed during runtime and are read-only. *Props* and *states* can also be used within a curly bracket in JSX just like the *name* variable in the Code Listing 3. If the value of a *state* changes, so would the value within the variable. Figure 3 shows an example of how the data flow can work in React. A component does not need to define *props* or contain *states*.

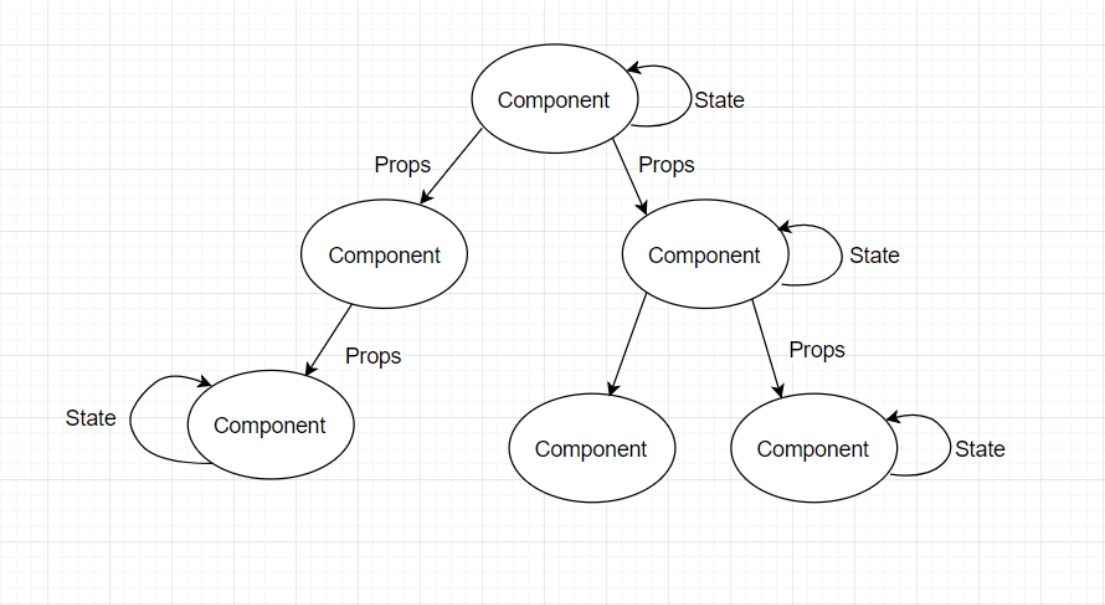

Figure 3: Dataflow in React

React components have a couple of lifecycle methods. These are methods that will run during a specific part of a component's lifecycle. They can be useful when an application needs to do a certain function at a specific moment. Example of these lifecycle methods are *componentDidMount* and *componentDidUpdate*. The *componentDidMount* method will run after the initialization, and the *componentDidUpdate* after every update.

Code Listing 4 and Code Listing 5 below shows the code of a small and simple React example. This shows how one can reuse components and define *props*. Here there are two React components *HelloMessage* and *App*. In App there are defined multiple HelloMessage components and each of them have a given name attribute. The name attribute will then be accessible as a *prop* in the HelloMessage component. The App will be rendered in a HTML file by using *ReactDOM.render()*. The result on the web browser is shown in Figure 4.

```
import React from "react";
import ReactDOM from "react-dom";

class HelloMessage extends React.Component {
    render() {
      return (
        <div>
          Hello {this.props.name}
        </div>
      );
    }
  }

class App extends React.Component {
    render() {
        return (
            <div>
                <HelloMessage name="Alice"></HelloMessage>
                <HelloMessage name="Bob"></HelloMessage>
            </div>
        );
    }
}

ReactDOM.render(<App />, document.getElementById('root'));
```

Code Listing 4: Basic React example with components and props

```
<html>

<body>
  <div id="root"></div>
</body>

</html>
```

Code Listing 5: HTML Document to render Code Listing 4

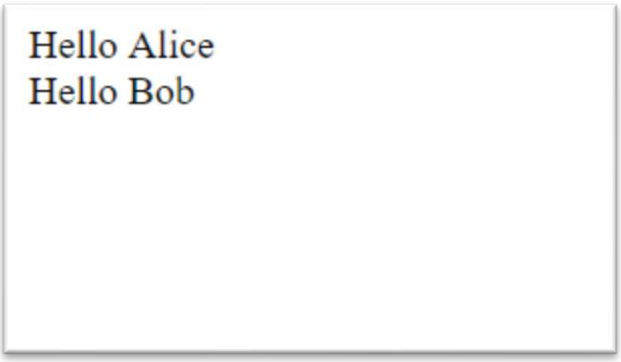

Figure 4: Result from Code Listing 4 showing the two HelloMessage in Web Browser

2.3 Combining React and X3DOM

With the benefits of React and X3DOM, in theory it should be easy to make interactive web pages with easy to use UI with reusable component-based structure, that contains 3D objects. The idea is to encapsulate all the X3DOM nodes required to produce an element inside a React component. An element can be anything from a simple box to a terrain of a landscape. To test this theory a basic example has been created. This example builds upon the code that was used to produce a 3D box geometry with X3DOM (see 2.1). The purpose is to separate all the X3DOM nodes for creating the box into its own React component and show how it can be reused. In addition, it highlights how attributes for a component can easily be defined.

The code for the example is shown in Code Listing 6 and Code Listing 7. The example consists of two React components called App and Box. Code Listing 6 shows the App component consist of a render method that is used to visualize the X3DOM scene, and the X3DOM nodes or/and React components contained within it. Inside the *scene* node two Box components are defined. Code Listing 7 contains the X3DOM nodes used for creating a box in its render method. This code will be returned in the render method where it was defined. In this example that would be inside the *scene* node in the App component. The App components consist of two Box components, which proves that the Box component are easily reusable, and can be used for making several boxes. In addition, each Box consist of two *props* called *col* and *pos*, defined with different values. These *props* are used to define a box's color and position respectively. Because each box has different values, the two boxes will be placed in different positions, and have different colors. The result of the two code listings is shown in Figure 5.

```
import React from "react";
import ReactDOM from "react-dom";
import Box from "box.js";

class App extends React.Component {
    render() {
        return (
            <x3d width='600px' height='400px'>
                <scene>

                    <Box col="0 0 1" pos="2 0 0"></Box>
                    <Box col="1 0 0" pos="-2 0 0"></Box>

                </scene>
            </x3d>
        );
    }
}

ReactDOM.render(<App />, document.getElementById('root'));
```

Code Listing 6: App component in X3DOM + React example

```
import React from 'react';

class Box extends React.Component {
    render(){
        return (
            <transform translation={this.props.pos}>
                <shape>
                    <appearance>
                        <material diffusecolor={this.props.col}></material>
                    </appearance>
                    <box></box>
                </shape>
            </transform>
        );
    }
}

export default Box;
```

Code Listing 7: Box component in X3DOM + React example

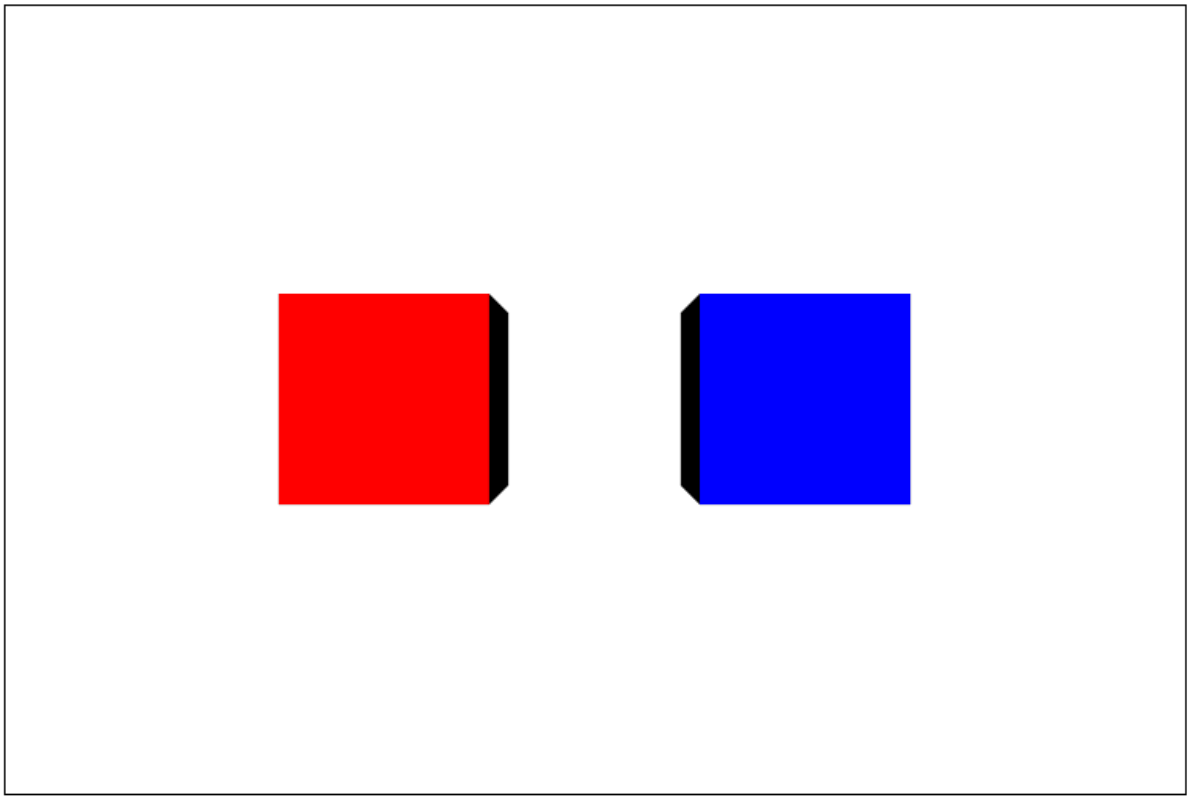

Figure 5: Result from X3DOM + React example, showing the two boxes in a web browser

The Box React component tags, need to be placed inside an external component containing the *<x3d>*, and *<scene>* tags to work (in this instance the App component). Additionally, the React component is the *<Box>* tag using an uppercase "B", and the tag with lowercase "b" is a node in the X3DOM framework. This highlights a small annoyance when using React with X3DOM, and why it exists. React requires that all X3DOM tags is written with the first letter as a lowercase character. If this requirement is not followed React will confuse the intended X3DOM tag as a potential React component. If the referenced React component does not exist or is not imported, it will result in an error.

During the research of Malt's master thesis the code from Code Listing 6 and Code Listing 7 would not have worked. At that time React would only whitelist attributes React recognized. Whitelisting means that React will only accept attributes from external libraries that it is familiar with. X3DOM was not includes on this list, which resulted in that previous editions of React, would simply ignore unrecognizable attributes. The issue has been fixed in version 16 of React [3]. This is important because React does not recognize X3DOM nodes. React will give a warning of this fact in the web browser console log, but this does not lead to problems because the desired and expected result still get rendered by React. This is a major factor in the decision to reexplore the possibility of making the application by using React.

2.4 Node.js

Node.js is an open-source and cross-platform JavaScript runtime environment [21] [22]. It can be used to create scalable network applications. In this application node.js, is used to create a web server both for the front-end, and the back-end server. An advantage of using node.js is that it makes JavaScript available to be used to write server- and client-side code. Another benefit of node.js is that a set of different packages can be used for performing specific functionality. These can easily be installed by writing the command "npm install {somePackageName}" in the command prompt within the project folder. This project makes use of a couple of these packages. Most importantly is React, which is in a package called *react*. The package contains the basic functionality for creating a React component [23]. In addition, the front-end application also uses the *react-dom*, *proj4*, *PapaParse* and *axios*. *react-dom* allows for additional features of the React library, *proj4* is used for conversion of different geospatial coordinates, *PapaParse* is used for reading and parsing of a comma-separated values (CSV) file used for an excel sheet, and *axios*, a promise-based HTTP client [24]. The back-end application uses the *mongoose*, *express* and *multer* packages. *Mongoose* allows node.js to interact with a MongoDB database [25]. *Express* is a web application framework that provides many features for a web application [26]. *Multer* allows for storing uploaded files (such as image files) in a storage folder [27].

2.5 Possible alternative approaches

In this application it has been decided that the two tools React and X3DOM should be used for several reasons. But it might be beneficial to consider other options. First one could consider another tool for creating UI. Angular is still a good option, and it might be possible to find a solution to some of the issues. However, these issues might be hard to identify, as Malt mentioned having trouble with doing this in his thesis [1].

X3DOM is not the only option one can use when working with 3D objects. One of the most popular libraries for creating a scenegraph is Three.js. Three.js uses JavaScript to define a scene and create 3D objects. Three.js make use of WebGL. Using three.js turns out to be a bit more complicated compared to use together with React, because it does not have HTML-tags. The return in a render method in a React component uses JSX, which means that all the three.js code needs to be contained within a *<script>* tag. There are a couple of libraries that tries to make tags for different function in three.js. One of these are *react-three-renderer*. This can be installed with node.js and aims to allow a three.js to be rendered with custom HTML tags, that can be used inside the react render method. According to the GitHub page for the *react-three-renderer* [13], the development of the project is moving slowly. While there are solutions that make it possible to use Three.js together with React, they are not fully supported and not compatible with the newer version of React. This along with the fact that three.js needs more code for setup, and creating simple shapes were a big factor for why three.js was not chosen for this thesis. This is one of the reasons we decided to explore X3DOM for this thesis and application, however this effort show that there is currently interest in the ability to work with a 3D scene together with React.

3 Application Overview

The application consists of a front-end and a back-end, both of which are using node.js. The front-end uses the tools React and X3DOM, to make the elements that will be shown to the user on a web browser. The project consists of a set of React components, each one focusing on the rendering of a particular element or performing some task. The elements could for instance be a 3D object, such as a box or a sphere.

In the web browser a user can see a 3D scene, and buttons to the right of the scene. The scene contains 3D objects, which includes the visualization of 3D topography of Svalbard, slices of seismic data and well logs. The slices and well logs also have a sphere that are placed near. This sphere is called a toggle and is used to toggle between whether an element should be visualized or not. This is useful for turning off geometry that is occluding other features. In, addition when a user presses on an element, a modal dialog prompt will open with some information concerning the element. The buttons on the side offer additional functionality, such as toggling elements, navigation and uploading files. The uploaded files will result in adding additional slices, or well logs to the scene. The user also can navigate in the scene, with the mouse or the keyboard. The data used in this project such as the 3D the topography map of Svalbard, seismic data, and well logs is provided by CMR. Figure 6 shows the application running in a Google Chrome web browser. This is the screen a user will see when they first load the web page.

The back-end consists of a server, that is used to handle requests from the front-end project to store or present data from the database. It also defines the models of different object that is used to define the corresponding table or schema in the database.

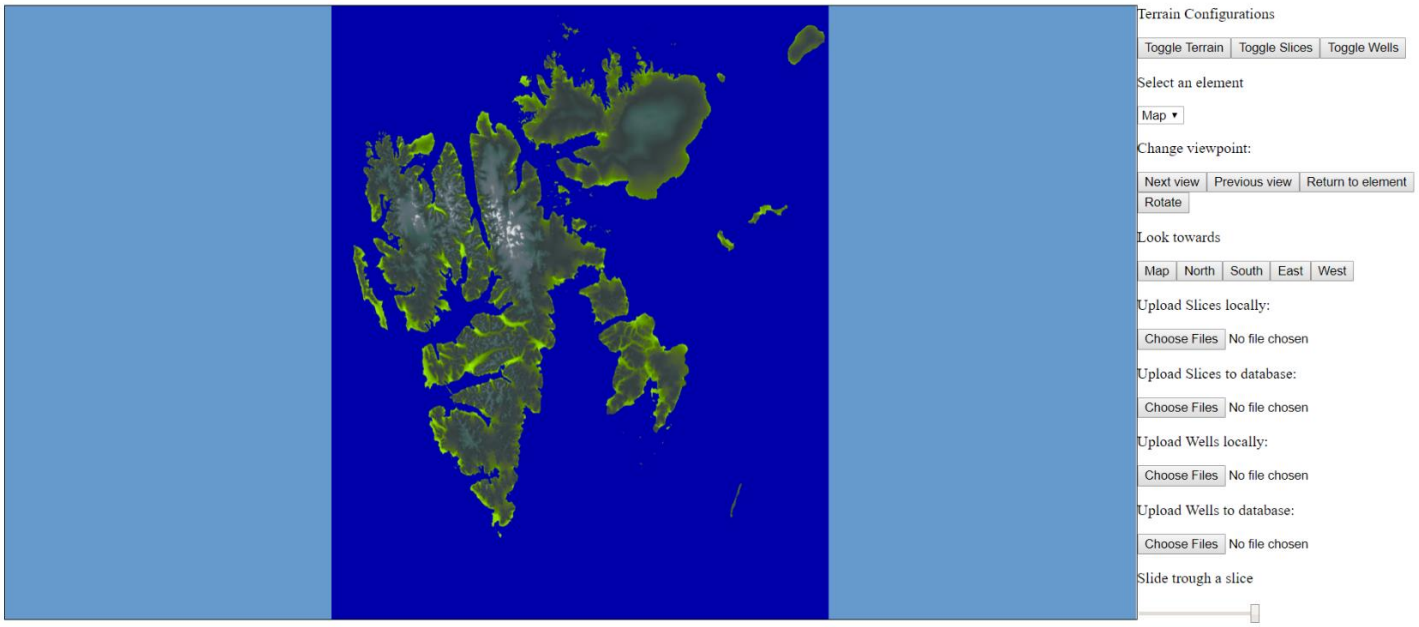

Figure 6: Application running in web browser

4 Solution

This chapter will go into details on how the application in this thesis has been design and implemented. The application consists of three parts: the front-end, back-end and a database. The front-end handles what is displayed in the graphical user interface (GUI) in a web browser. The front-end consists of a set of React components. All the React component are built from the ground up for this project. Most of the components uses X3DOM nodes for creating 3D objects. All the nodes used in the project are a part of the X3DOM framework. The back-end is used to connection between the front-end and the database. It is used as a middle step for transportation of data.

The purpose for the application is to give a user the ability to easily look at geological data gathered from beneath the surface. The data comes in two types, which are called slices of seismic data (referred as “slice” from this point) and well logs. A *slice* consists of an 2D image that provides an easy way to look at useful information concerning seismic data. A well log is a file containing a set of properties and UTM coordinates. The visualized representation of a well log is referred to as a “well” in this thesis.

UTM coordinates is a way to provide indirect information about latitude and longitude. This can be used to determine the real-world location of an object. In this project UTM coordinates are used in a process to determine where a *well* or a *slice* is placed in relation to the terrain. To achieve this the UTM coordinates needs to be converted to a local coordinate system.

It is important for a user to be able to understand where the data is located. For this reason, a terrain will be visualized in the GUI. The terrain will help a user to understand where the *slices* and *wells* are located in the real world. This project uses data provided from CMR. The data has been gathered from the beneath the surface of Svalbard. As a result of this the terrain is the topography of Svalbard. An issue when working with visualization of a large set of data, is that each element can obstruct each other. To avoid the issue a user should be able to toggle which elements they want to be visualized, and which elements they want to be hidden.

It can also be useful for a user to be able to see additional information concerning *slices* and *wells*. This information includes the element’s name, a description of the element, and related articles or literature. To access this information a user can click on the element to open a modal dialog prompt. Finally, the user has the ability to upload *slices* and well logs. This allows a user to look at their own data and share it with others.

The React components implemented for this application are:

- App
- Terrain
- Slice
- Well
- UploadSlices
- UploadSlicesDB
- UploadWells
- UploadWellsDB
- Modal
- Toggle
- Volume
- Navigation
- SlicesDB
- WellsDB

Figure 7 shows the structure of some of the React components within the Application. The boxes in this figure represent the different components. The arrows signify which components are defined by others, e.g. Navigation is defined by the App component. At the highest level is the App component which defines all the component directly within itself or indirectly through another component. The App component defines, the Navigation, Terrain, UploadSlices and UploadWells. The UploadSlices component defines a set of Slice components. Likewise, the UploadWells component defines a set of Well components. Each of the Slice and Well components define a Modal component and a Toggle component. The thesis will go into further details for each of the components in some of the following subchapters.

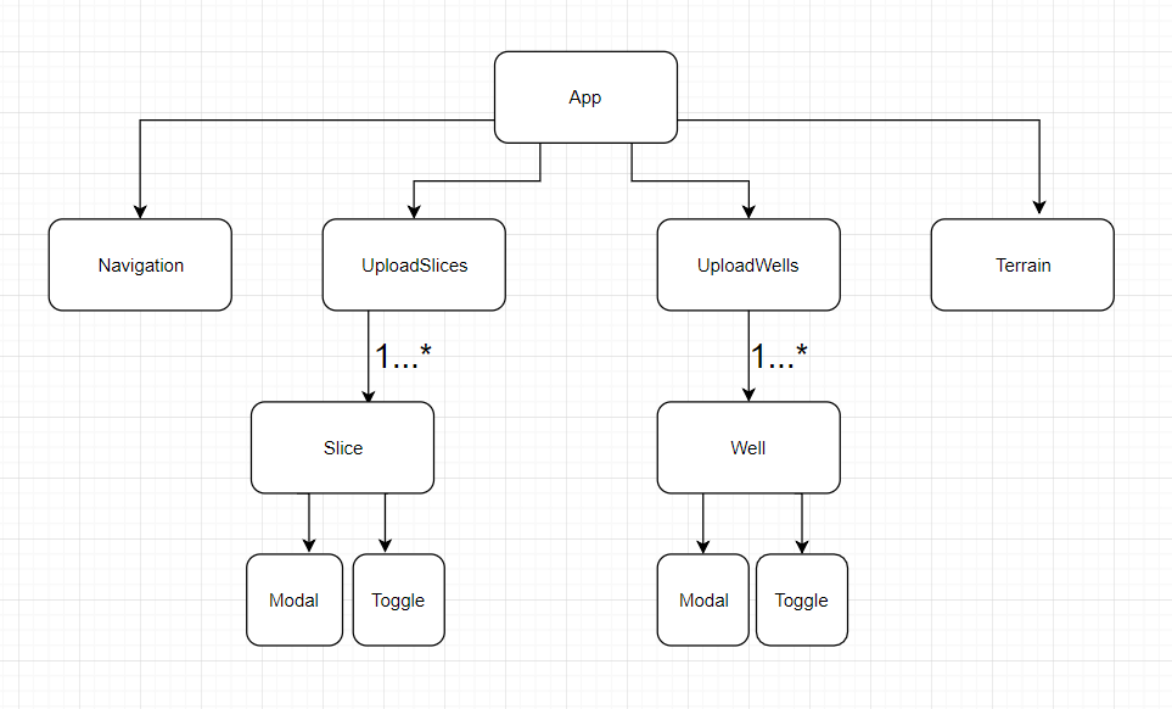

Figure 7: Application component structure

4.1 Setup and application structure

The application follows a structure called MERN stack. MERN is short for the four tools that are used to build the structure [28]. These are MongoDB, Express, React and Node.js. Figure 8 shows the MERN Stack architecture. This consist of a front-end application, and a back-end server and the database. The back-end server is used to connect data submitted from a user in the front-end, to be stored in the database. It is also responsible for showing requested data by the front-end from the database.

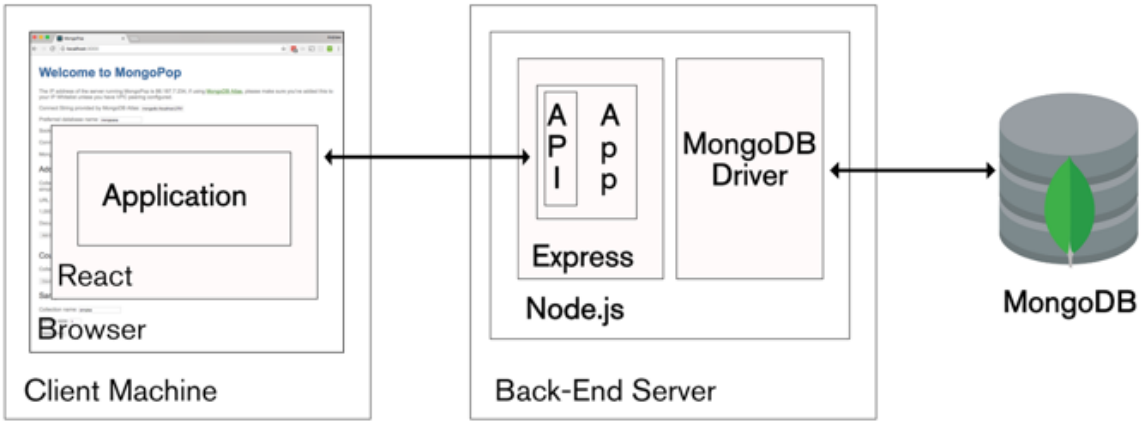

Figure 8: MERN Stack architecture, image from MongoDB [28]

Both the front-end and the back-end runs on local servers by using Node.js. The front-end application will run on port 8080 (if it is currently available, if not it will go to the port with the next number that is available), when typing the command "npm start" in the command prompt from the project folder. The front-end uses the npm package *webpack*. Webpack will detect when code in the front-end is changed and rebuilds the application automatically. This makes development of the application easier, because one does not need to restart the application whenever the code is changed. The application can also be connected to a local mongo database. This can be started by typing the command "mongod" in the command prompt. To start the back-end, type the line "nodemon server" in the back-end folder of the project in a separate command prompt. The back-end is running on localhost:4000, by default. The source code for the application and the backend can be found at a public GitHub repository [29]. To use MongoDB one can, download the "MongoDB Community Server" from MongoDB's official web page [30].

4.2 App component

As mentioned in the beginning of this chapter the App component is the highest leveled component in the application. It contains all the other components in the application directly or indirectly (a component is defined within a component defined in app).

The App component's purpose is to render the *x3d* and *scene* X3DOM nodes. These two nodes are necessary for using the other X3DOM nodes in the X3DOM framework. Because of this all the React components that contains X3DOM nodes must be placed within a *scene* node. The App component provide a good location for the declaration of the *scene* node, as it is the component on the highest level. All React components that uses X3DOM nodes must be defined within a *scene* node, otherwise they will not function as expected. Code Listing 8 shows the *scene* placed within the render method of the App component. There are four React component defined within the *scene* node. These are Terrain, UploadWells, UploadSlices and Navigation. In addition, the *scene* contains one X3DOM node called *background*. This node determines the background color of the scene, which in this case is light blue.

```
render() {

    return (
        <div>
            <x3d width={width} height={height} id="x3d_context">
                <scene>
                    <background skycolor="0.4 0.6 0.8"></background>

                    <Terrain name="Svalbard" render={this.state.terrainVisible}>
                    </Terrain>

                    <UploadSlices showSlices={this.state.showSlices}> </UploadSlices>
                    <UploadWells showWells={this.state.showWells}> ></UploadWells>

                    <Navigation></Navigation>

                </scene>
            </x3d>
```

Code Listing 8: The scene within the App component

In addition to the scene, the App component will render UI elements that are placed outside the *scene* and *x3d* nodes. These elements can be seen on the right side in Figure 6 from chapter 3. The UI elements includes buttons used for toggling the visualization of elements within the scene, uploading elements and navigation control. In addition, there is also a drop-down menu used for navigation.

4.3 Terrain component and BVHRefiner

This application can visualize the terrain of Svalbard. The terrain provides users of the application with a reference point that allows users to see for themselves where certain elements are placed in relation to the real world.

The terrain is built from a X3DOM node called *BVHRefiner*. According to the X3DOM documentation, the *BVHRefiner* is a node that has the ability to refine and load hierarchical data dynamically during runtime [31]. It loads data from a set of folders containing image files. *BVHRefiner* loads different images based on the current level of detail (LOD) of the terrain. The LOD determines how much detail the terrain has. The LOD will increase or decrease based on the user's view position in relation to the terrain. If the user view position is close to the terrain, the BVHRefiner will use more image data for rendering the terrain, that will in turn make it more detailed. However, the part of the terrain which is outside the user field of view (FOV), will not be rendered.

The *BVHRefiner* supports two dataset folder structures, which are WMTS and TREE. This project uses the WMTS dataset structure. Figure 9 below shows a general WMTS folder structure. The *BVHRefiner* support three different datasets, these are displacement data, subsurface texture data and normal data. Each of the dataset has its own WMTS. This structure consists of a folder for each level of detail, simply named from 0 to n. Each level of detail consists of subfolders explaining the columns of the matrix. The number of columns needed for level 0 is one. The number given for every other level is 2^n, where n is the level. Each of these subfolders consist of image files, the number of files in each folder is the same as the number of columns for that level. The *BVHRefiner* node contains many fields describing how the terrain should be rendered.

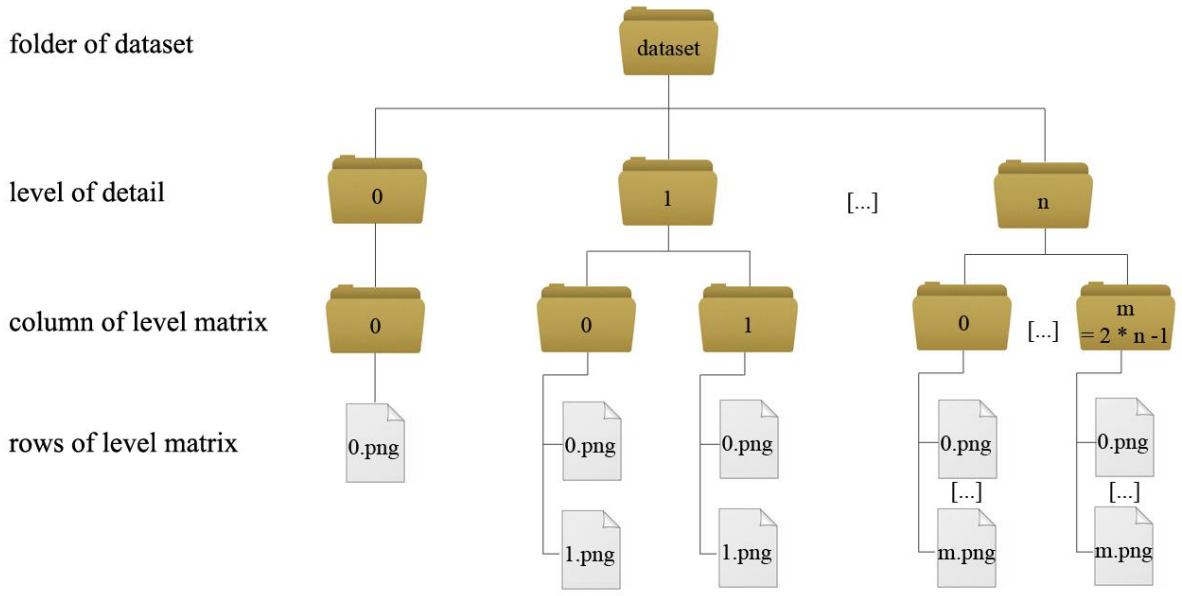

Figure 9: WMTS Folder Structure, image from X3DOM Documentation on BVHRefiner [31]

These field include *maxDepth*, *minDepth*, *interactionDepth*, *size*, *subdivision*, *factor*, *maxElevation*, *elevationURL*, *textureURL*, *normalURL*, *elevationFormat*, *textureFormat*, *smoothLoading* and *mode*. The *elevationURL* and *textureURL* should have values that define the path were the files needed to determine the elevation and texture of a terrain. The *elevationFormat* and *textureFormat* is used to define which file type the image files are. usually PNG or JPEG file. The *maxDepth* and *minDepth* fields are used to define the maximum and minimum depth of the terrain, i.e. the number of LOD the terrain should use. *InteractionDepth* are the maximum depth during user interaction with the scene. *smoothLoading* is used to determine how much time the application can use to switch between the different LOD assets. The *size* field is simply the size of the terrain, and *subdivision* is the resolution of a rendered tile. The size attribute is defined in the *componentDidMount* method. The reasoning for this is the same as for the event handler discussed in subchapter 4.6 below. The ratio of the size chosen for the terrain, matches the corresponding location on Svalbard. *Mode* is used to determine if the terrain should be visualized in 2D or 3D.

Finally, *factor* field is defined as number used to determine from when the different LOD should be rendered. How high this number is, will affects the balance between the quality of the terrain and the performance of the application. In Figure 10 and Figure 11 below one can see the difference between two LOD. These images are taken from the exact same position in the application, but the *BVHRefiner*'s *factor* variable has a different value. In this application there are only two LOD, and the difference is not that noticeable. If the *BVHRefiner* had the assets to support more LODs, the difference would be more noticeable. The two figures look similar, but if one would look closely one can see that Figure 10 on the left appears more blurry compared to Figure 11. Because the position of the camera is a short distance from the terrain, one should choose a value for the *factor* variable that gives the same result as seen in Figure 11.

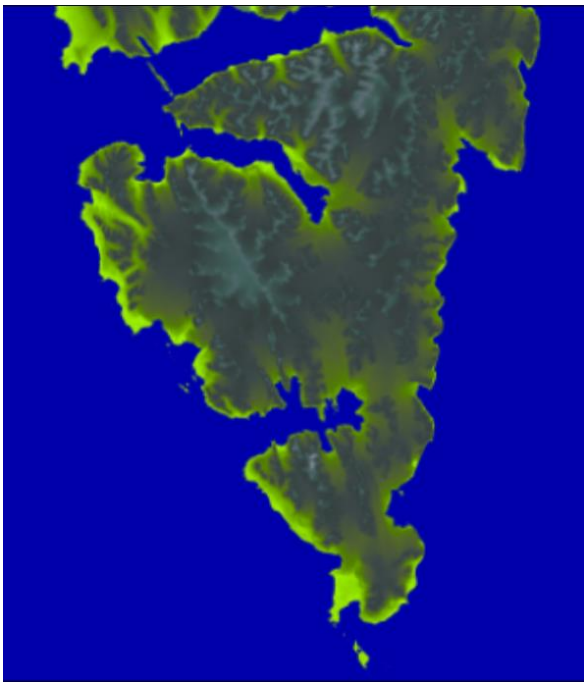

Figure 10: Low LOD

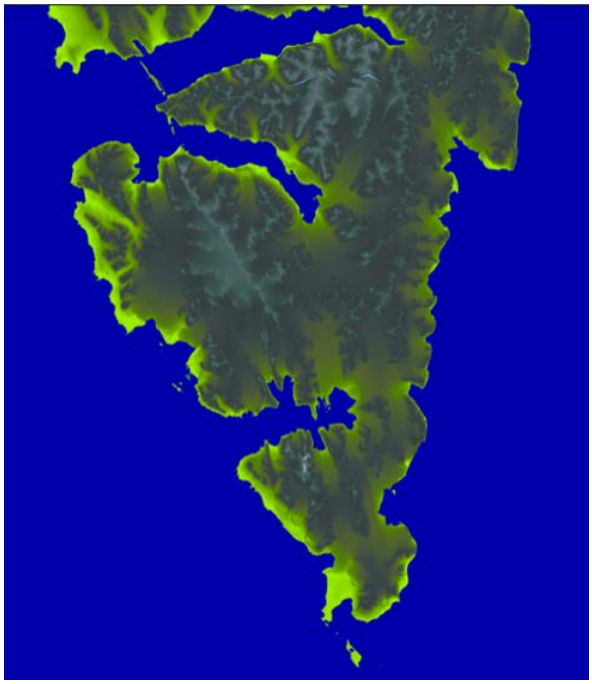

Figure 11: High LOD

The user also has the ability to toggle the terrain on and off by pressing a button. This button is placed in the App component. The App component has a *state* called “terrainVisible” that will change between true and false for each time the button is pressed. The *state* is defined as a *prop* in the terrain declaration. This *prop* will be used to determine if the render parameter in *BVHRefiner* is true or false. If this *prop* is not defined, the default value will be true. The *BVHRefiner* also has a *solid* filed set to false, this ensure that the terrain is visible from both sides. This is useful, because most of the elements in the application are placed beneath the terrain.

The height-map images used to create the elevation of the terrain with the *BVHRefiner* node was originally extracted from a GeoTIFF file of Svalbard. This file came from the Norwegian Polar Institute [32]. As explained by Malt [1] the extraction was performed using the tool Geospatial Data Abstraction Library (GDAL). The same tool was also used to generate textures for the terrain. Figure 12 shows the entire terrain of Svalbard from above. This should be the start orientation, when the web page is visited. Figure 13 shows the terrain from ground level. The figure also illustrates more clearly the difference in elevation.

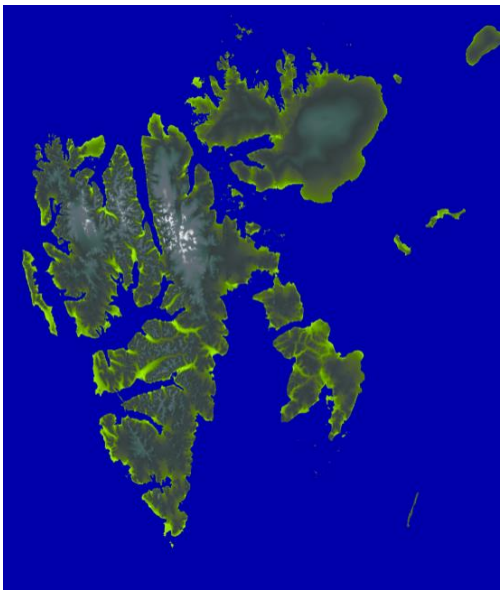

Figure 12: The terrain of Svalbard looked at from above

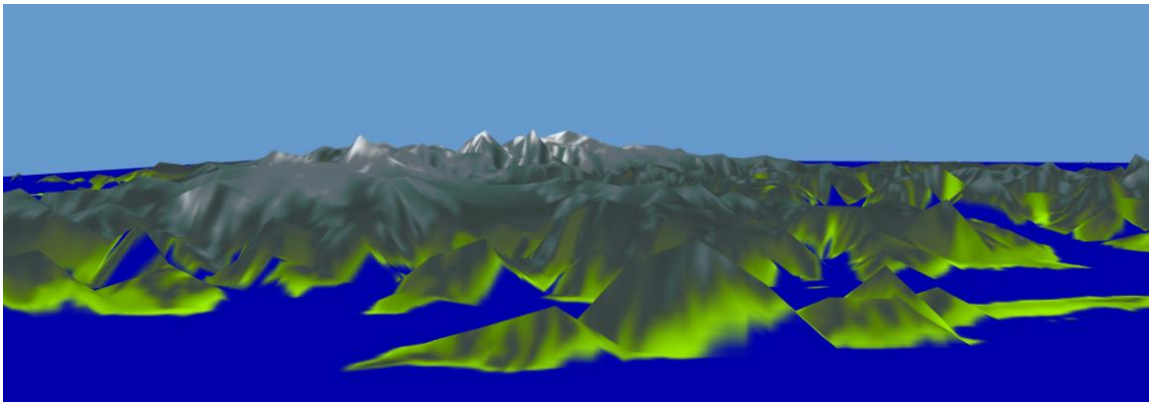

Figure 13: The terrain of Svalbard looked at ground level

4.4 Slice component

The purpose of a Slice component is to visualize a *slice* at the correct location. A *slice* is rendered in the scene as a X3DOM *box* geometry. The box has been made thin and a texture is added on the side surfaces of the box. This gives the *slice* the appearance of a 2D Image placed inside the scene. A *billboard* X3DOM node was considered to be used instead, however then the texture on the backside of the *slice* would have been mirrored.

A JavaScript class called *SliceModel*, has been used to contain all attributes concerning a *slice*. This class is shown in Code Listing 9. As seen in the code listing the *SliceModel* contains variables for the *slice*'s name, description, longitudes, latitudes, depth and related articles.

```
export default class SliceModel {
    constructor(name, description, imageUrl, start_e, start_n,
    end_e, end_n, start_depth, end_depth, articles) {
        this.name = name;
        this.description = description;
        this.imageUrl = imageUrl;
        this.start_longitude = start_e,
        this.start_latitude = start_n,
        this.end_longitude = end_e,
        this.end_latitude = end_n,
        this.start_depth = start_depth,
        this.end_depth = end_depth,
        this.article = articles;
    }
}
```

Code Listing 9: SliceModel JavaScript class

When a Slice component is defined it has been given a SliceModel as a *prop*. This *prop* is used to determine certain values in a *slice*. The *imageUrl* is used as the texture for the *slice*. To determine the location and shape of the *slice* the values for longitude, latitude and depth needs to be converted. They are first converted into UTM coordinates, and then to the local coordinate system. The UTM coordinates give an accurate description of the real location of the *slice*, but the UTM coordinates needs to be converted into local coordinates for the application to display the *slices* in the right position in relation to the terrain. A consequence of this is that the image used as a texture on the *slice* will not have the correct ratio. Figure 14 shows two *slices* in the application.

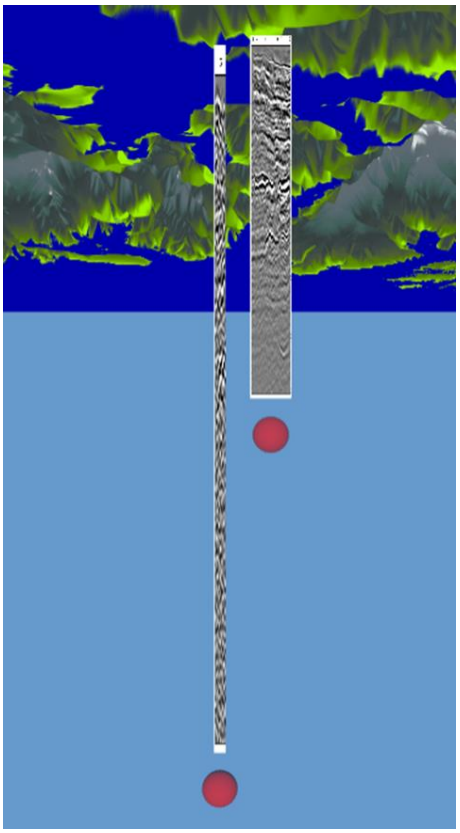

Figure 14: Two slices in the scene

Finally, the Slice component define a Toggle and Modal component, see subchapters 4.7 and 4.8 for details concerning these components. In the definition the Slice component will define some variables as *props* that can be used by the Toggle and Modal component.

4.5 Well component

The intension of the Well component is to visualize a *well* based on data from a well log. A Well log is a textual file which contain data for a *well* gathered from drilling in the ground beneath the surface. The file contains a set UTM coordinates and information about different properties concerning a *well*. The area between two UTM coordinates forms a line, which will be referred to as a *segment* or *subwell* in this thesis. All the *segments* together show the path the well log has been drilled. Each *segment* represents a certain depth of the *well*.

The Well component visualizes the *wells* as a 3D object. Each *segment* of the *well* is visualized as a parallelepiped (usually a right square prism). The *segment* is visualized as a parallelepiped because it is easy to spot from any view angle. In addition, if two coordinates are not placed directly beneath each other, the parallelepiped still allows the different *segments* to look connected. All the *segment* together forms the 3D visualization of the *well*. Each *segment* is given a color based on the value of a property. Figure 15 shows two pictures of one *well*. For each of the picture a different property has been selected to be visualized. The two properties consist of different values for each *segment* that results in different colors for the visualization of the *well*. Figure 16 shows an example of a *well* that are have not been drilled directly downwards.

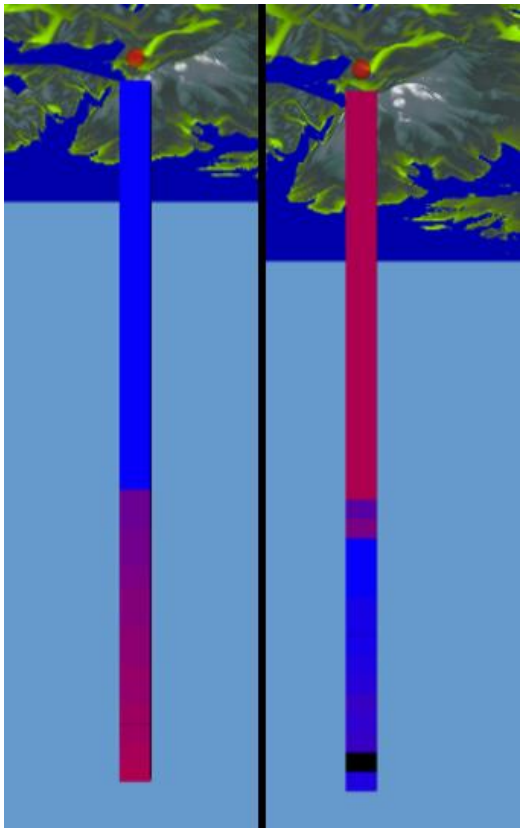

Figure 15: A well visualized with two different properties

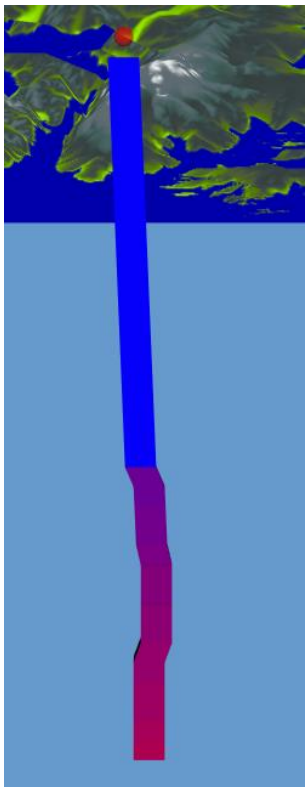

Figure 16: Example of a well, that has segments that have not been drilled directly downwards

The properties in the well log details information about the surrounding geology and the drilling operation. Examples of such properties can be the temperature, pressure or P-wave velocity. For all of the properties a numerical value is provided for each and every *segment* in the *well*.

Although a *well* can have multiple properties, only one can be visualized at any given time. The user can toggle between the different properties of the *well*. Each *subwell* is given a color based on values for the property currently chosen. The color is based on the what the value is in relation to the maximum and minimum values for that property. The *subwell* with the highest value will be colored red, while the one with the lowest value is blue. The other *subwells*' color will be somewhere on the color line between blue and red, as seen in Figure 17. A *subwell* will get the color black if the value for a property is invalid. The color line is a gradient made from the *createLinearGradient* method on the context of the canvas element. Two colors are added on the gradient. These are blue and red. The line will interpolate in the area between these colors. If the value of the property is larger than the maximum value of the gradient, the corresponding *subwell* will be red.

Figure 17: Color spectrum for a well

To store the variables concerning a *well*, two JavaScript classes has been implemented. They are called *WellModel* and *SubWellModel*. The *WellModel* contains all the variables a *well* need, while the *SubWellModel* contains all the variable a *segment* of the *well* need. A *WellModel* has an array of *SubWellModel* objects. Code Listing 10 and Code Listing 11 shows the *WellModel* and *SubWellModel* classes.

```
export default class WellModel {
    constructor(name, des, subwells, properties, article, start_e, start_n, depth){
        this.name = name;
        this.description = des;
        this.subwells = subwells;
        this.properties = properties;
        this.article = article;
        this.longitude = start_e;
        this.latitude = start_n;
        this.depth = depth;
    }
}
```

Code Listing 10: WellModel JavaScript class

When a Well component is defined it has been given a *WellModel* object as a *prop*. The Well component uses the *prop* for the visualization of a *well*. The *well* is created from a list containing *wells* coordinates, with x, y and z position values. The area between two coordinates forms a line. In order to convert the line into a parallelepiped (representing a *subwell*). To do this we add eight more coordinates, which are placed from a specified value away from the two originals coordinates. Each of the original coordinates has a coordinate that has this value on the x-axis, z-axis and both the x- and z-axis. Each of the vertices contains a color. All the 8 vertices in the *subwell*, is given the same color. This is done for every *subwell* in a *well*. Figure 18 shows a model that highlights where the eight new coordinates will be placed based on the two original coordinates. The two original coordinates are the two points (0 0 0) and (0 1 0). The "x" value in the figure represent a given width from the center of the point.

```
export default class SubWellModel {
    constructor(x, y, z, x2, y2, z2, propertyValues) {
        this.x = x;
        this.y = y;
        this.z = z;
        this.x2 = x2;
        this.y2 = y2;
        this.z2 = z2;
        this.propertyValues = propertyValues;
    }
}
```

Code Listing 11: SubWellModel JavaScript class

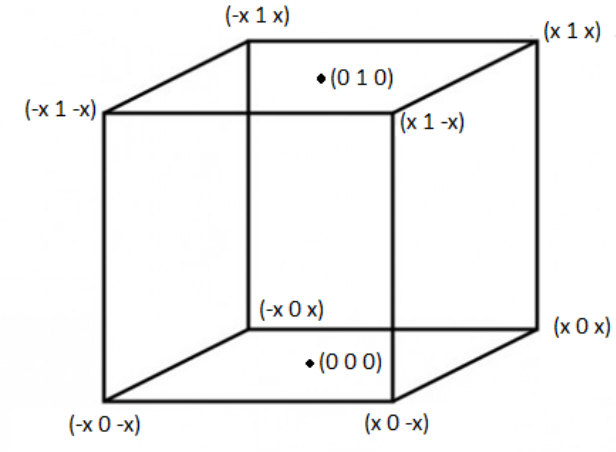

Figure 18: Shows the value of the coordinates for a subwell

To render the *well* the X3DOM node *IndexedFaceSet* is used. An *IndexedFaceSet* is an 3D object that contains a list of polygons. The *IndexedFaceSet* node has a field called *coordIndex*. This field explains how to the faces are used to create a polygon. The application uses the coordinates from the *well* to create the faces. Every *subwell* should have six faces, and every face are made from four of the eight coordinates the *subwell* consist of. Inside the *IndexedFaceSet* node there are two other X3DOM node. These are *Coordinate* and *Color*, which have a field containing a list of all the coordinates and colors in the *well* respectively. The field for color is determined by a *state* called *currentColors*. This *state* will change based on the currently displayed property of the *well*.

Figure 19 shows a mockup of the *subwell* that would render if the *cordIndex* field has the value: ”0 1 2 3 -1, 4 5 6 7 -1, 0 4 5 1 -1, 1 5 6 2 -1, 2 6 7 3 -1, 4 0 3 7 -1”. The numbers from 0 to 7 represents the eight vertices, and -1 is used to separate each face. The four numbers between “-1” represent a face. The order of the numbers is important to render the *subwell* correctly.

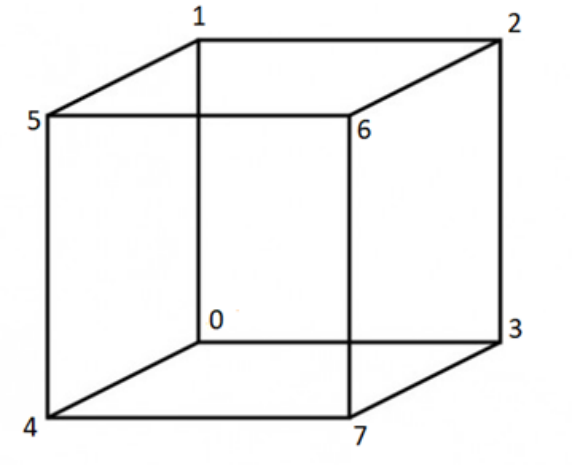

Figure 19: Shows the vertices used to created six faces for a subwell

4.6 Handling events on X3DOM tags

In this application, it is a desired functionality that an event gets triggered when a user clicks on a 3D object. This event could be opening a dialog prompt or changing the color of the 3D object. To do this an eventListner need to be added to an X3DOM tag. This includes interactivity with the 3D objects that are created from X3DOM tags. This subchapter describes a technique that allows some functionality to happen when an object created from X3DOM tags are clicked on.

Usually within HTML tags, one can define an onClick method. This method will execute when the element created from the tag is pressed on with the mouse pointer. However, X3DOM requires that the eventListener is added after the X3DOM tags has rendered in the web browser. This is mentioned in the X3DOM documentation "The call of the onclick function is handled by x3dom by directly calling the callback function, since the addEventListener method needed to be overwritten. No page-wide onclick events are thrown, so attaching a listener to this object is only possible after." and "Alternatively, wait until the page is fully loaded and the document.onload event was fired." [33]. Because of this limitation an alternative approach must be applied to perform this action.

All React components have a set of lifecycle methods, such as *ComponentWillMount, componentDidMount* and *componentDidUpdate*. These methods run automatically based on the what happens to a component. From the documentation of X3DOM it is mentioned that an eventListener should be added after the object is rendered. To ensure this, the method *componentDidMount* can be used, as it runs after the initial rendering of a component, which includes all the X3DOM tags needed for the element the component should render. Alternatively, the *componentDidUpdate* method, can be used. However, then the application needs to control that the eventListner only gets added once or removed before adding a new one.

To add an eventListner to a X3DOM tag used in the components render method, within the *componentDidMount* method, some kind of reference is needed. React provides this with a concept called *refs*. The *refs* should be created within the constructor of the component and should be used within a desired X3DOM-tag as a reference. In Code Listing 12 on the next page a generic example shows how this can be done one the X3DOM-tag *<shape>*. The result of this code will be that, when pressing on the object formed by the *<shape>* tag (which could for example be a 3D sphere within the scene) the method "someMethod" will run.

```
constructor(){
    this.someRef = React.createRef();

    this.someMethod = this.someMethod.bind(this);
}

render() {
    return (
      <x3d>
        ...
        <shape ref={this.someRef}>
            ...
        <shape>
        ...
      </x3d>
    )
}

someMethod(){ ... }

componentDidMount(){
    this.someRef.current.onclick = () => this.someMethod();
}
```

Code Listing 12: Using a ref to add an eventListner to a X3DOM tag

4.7 Toggle component

The Toggle component purpose is to give a user the ability to toggle whether an element should be visualized or not. The Toggle component visualize a small red semi-transparent sphere which is called a *toggle*. This is done by using the X3DOM *sphere* geometry node, which is implemented in a similar way as the box in Code Listing 2 (see 2.1). The Toggle component is defined by the element that should be toggled. In this application, the component could either be a Slice or Well component. The *toggle* is placed very near the element it toggles. Figure 20 shows a *toggle*.

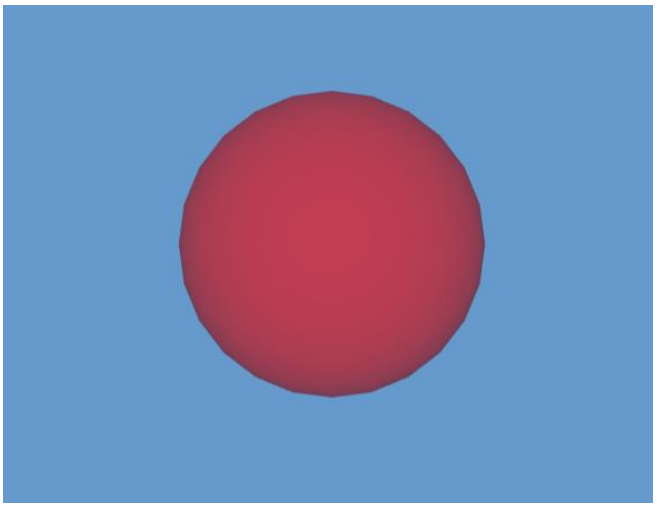

Figure 20: A toggle

The toggle gets a *ref* to an element, defined as a *prop* when it is created within the element's component. This *ref* is used to create an event handler (described in subchapter 4.6). When pressing on the toggle, the element will change between being visible or not in the scene. In the code the element's component has a *state* called *isVisible*. This *state* can either be true or false. When pressing on the toggle this value will change between the two values. The value determines the render property of the element's component X3DOM *shape* node. Alternatively, a user can toggle the element to not be visualized, by pressing the right mouse button on the element instead. This option is only possible if the element is currently visualized.

4.8 Modal component

The purpose of the Modal component is to display a modal dialog prompt. This window should open when a user clicks on a *slice* or a well log. The dialog prompt should contain additional information for the element that was pressed on. The information can for example include the name, description and related articles for an element. To close the dialog prompt a user can press on the close button or outside the dialog prompt.

Figure 21 shows the modal dialog prompt for a *slice*. The *slice* has the name "AG-1-depth", some description just below, that details where data has been conducted and how it has been calculated. Beneath the description is an image. The image is the same as the texture used for the *slice*. In the dialog prompt the image has a ratio that makes it easier to analyses the image. In addition, a user can press on the image. This action will result in opening a new tab in the web browser. This tab shows the URL for the image, which makes it larger and shows it in the correct ratio. Finally, the dialog window contains a link for related articles.

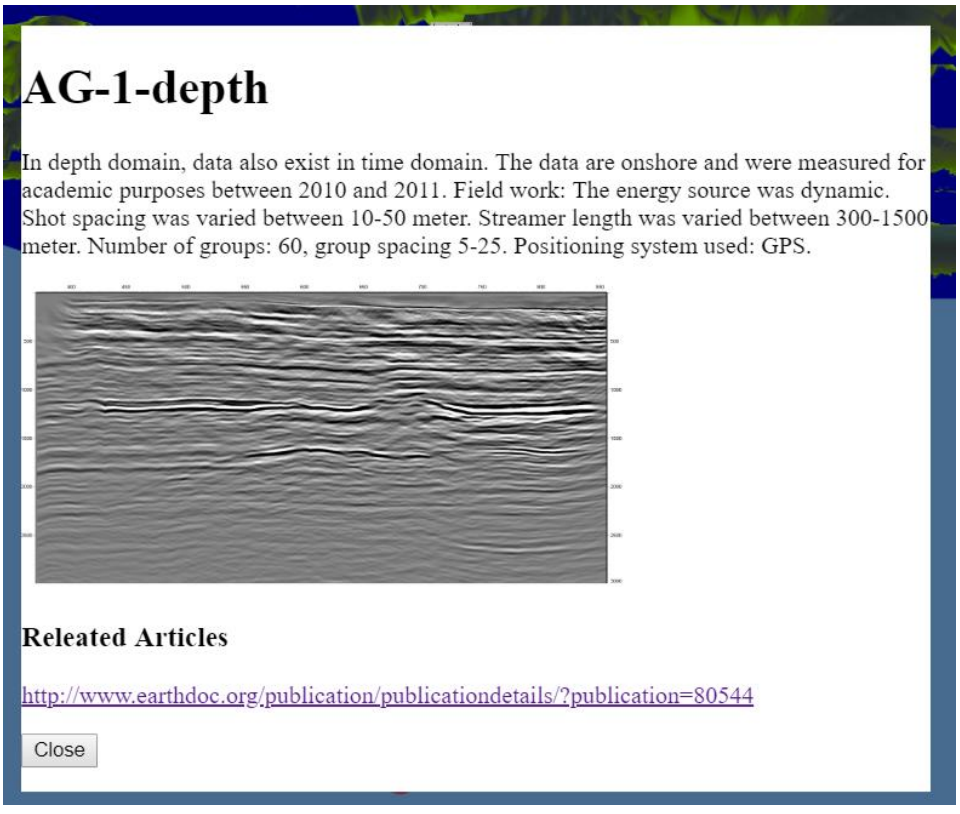

Figure 21: A dialog promt for a slice

Figure 22 show a dialog prompt for a *well*. Just like a *slice* it contains a name, description and related articles. However, it does not contain an image. Instead it has a drop-down list of properties (referenced to as visualization types in the figure). When a user selects a property the color of the *well* will change to reflect the values of that property. The differences between the dialog prompts for each element, shows a big opportunity for flexibility.

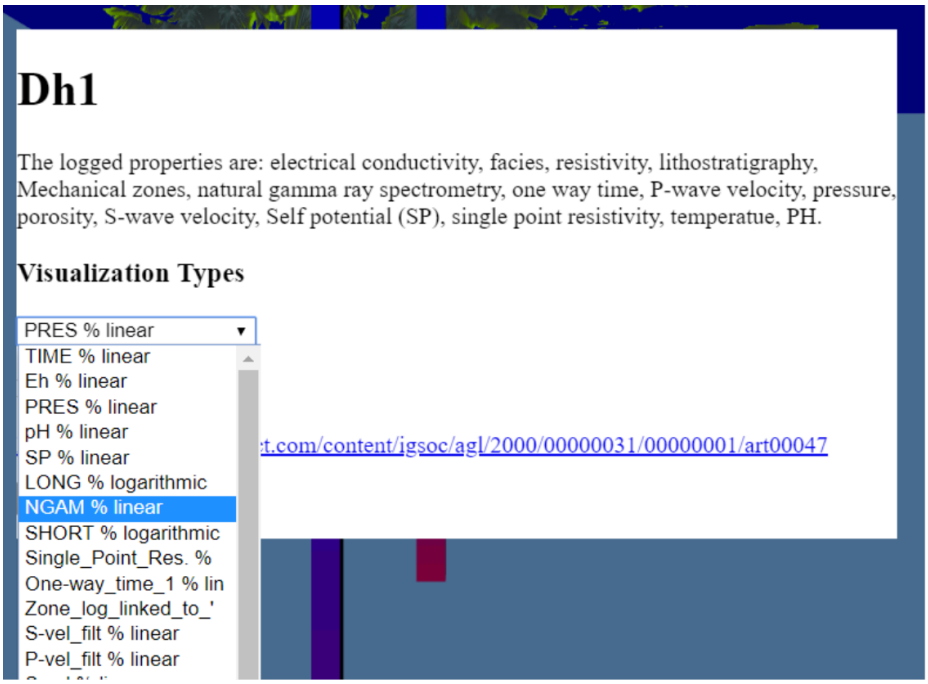

Figure 22: Dialog prompt for a well

4.9 Components for uploading *slices*

In the application a user has the ability to upload one or more *slices*. When a user uploads a set of *slices*, they should appear within the scene of the web browser. This feature allows a user to see their own data or share their data with other users. There are two React components that can be used to upload *slices*. They are called UploadSlices and UploadSlicesDB component. The two components are implemented similarly but has some key differences. The UploadSlices component reads all the uploaded data and uses it to define a list of Slice components. The UploadSlicesDB only needs to read the data from the excel file. Instead of defining a list of Slice components, it sends the data from the excel file alongside the uploaded image files to the back-end server by using *axios*. The back-end server will then store the data in the database. Unlike UploadSlices, the UploadSlicesDB component dose not define a set of Slice components. This is handled by a separate component called SlicesDB. These two options give a user the choice whether they want to share the data, or just observe it on their own.

The user can upload a set of *slices* in to the application, by pressing on the "Choose File" button beneath "Upload Slices locally" or "Upload Slices to database", shown in Figure 23. A user must upload an excel sheet contained in a CSV file and at least one image file. If they do not follow this requirement the application will give the user an error message that explains the issue. Figure 24, Figure 25 and Figure 26 shows three different error messages that can occurred based on what a user has not uploaded. The image file can either be a PNG or a JPG file.

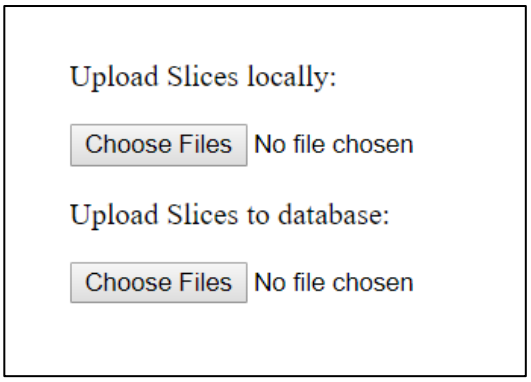

Figure 23: Buttons for uploading slices

Figure 24: Error message for missing excel file

Figure 25: Error message for missing image file

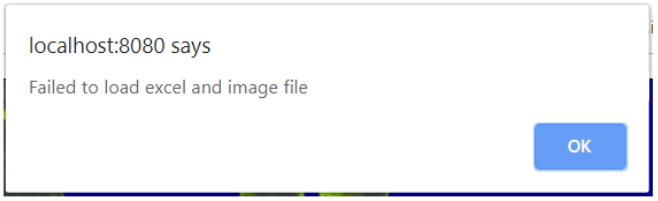

Figure 26: Error message for missing image and excel files

CMR has used Microsoft Excel spreadsheets to store data connected to the *slices*. This data includes information such as a name, description, related articles, longitude and latitude. Because of this the application need to be able to read and parse data from the spreadsheet. To read and parse the date from the sheets the PapaParse framework is used. A benefit with the spreadsheet is that it allows a user to upload multiple *slices* at the same time. The CSV file consist of an excel sheet containing data for a set of *slices*. The first row in the excel sheet contains all the variables a *slice* can have. Each of the following rows represents a *slice*, and the value for all its variables. One of the columns in the excel sheet is called "additional_files". This contains a reference to an image file. For a specific *slice* to be visualized the image file referenced in this column must be uploaded together with the excel file. A user can upload multiple *slices* at once if the sheet contains multiple rows, and multiple image files are uploaded.

Figure 27 on the next page shows a snippet of the excel sheet. This shows some (but not all) the variables for five *slices*. The depth values are used to determined how deep the *slices* should be visualized. For instance, the two *slices* shown in Figure 14 are the two first *slices* in this sheet. Both *slices* have the same "start_depth", but one of them has twice as large "end_depth". This result in that one of the *slices* is twice as long compared to the other one.

	A	B	C	D	E	F
1	data_type	dataset_name	info	additional_files	start_depth	end_depth
2	Seismic depth	AG-1-depth	In depth dom	AG-1-depth.png	0	100
3	Seismic depth	AG-2-depth	In depth dom	AG-2-depth.png	0	200
4	Seismic depth	AG-3-depth	In depth dom	AG-3-depth.png	0	300
5	Seismic depth	Line_1-depth	In depth dom	Line_1_utm.png	0	400
6	Seismic depth	Line_2-depth	In depth dom	Line_2_utm.png	0	400

Figure 27: Snippet of excel sheet for slices

To use the image for a *slice* as a texture on the *slice* it needs to be referenced as a URL. In the UploadSlices component a FileReader is used to get a URL representing the image. The FileReader can read the image files and return it as a URL.

When uploading the files, it is important to consider the order in which the files are read. The image files should be read first, because the excel file contains references to the images. If no excel files and/or no image files are uploaded, the application will give the user a message of what is missing for the uploading process.

All the variables from the uploaded data, will be used to create a *SliceModel* object for each *slice*. In the UploadSlicesDB component these objects are sent to the back-end server. In the UploadSlices component, each *SliceModel* is defined as a *prop* called *sliceProp* for each corresponding Slice component. In Code Listing 13 the render method for the UploadSlices is shown. Here a *listOfSlices* array defines a Slice component for each *slice* in the array. The array is defined in a different part of the component. The UploadSlicesDB component render method consist only of a HTML input element.

```
render(){
    return (
        <React.Fragment>
            {listOfSlices.map((sliceModel,index) => (
                <Slice key={index} index={index} sliceProp={sliceModel}></Slice>
            ))}
        </React.Fragment>
    );
}
```

Code Listing 13: Shows how the UploadSlices component creates a list of Slice component

4.10 Components for uploading *wells*

This application allows a user to upload *wells*. The process of uploading *wells* is very similar to the process of uploading *slices*. There are two components that handles uploaded data for *wells*. They are *UploadWells* and *UploadWellsDB* component. The differences between these two components are largely the same as the differences between UploadSlices and UploadSlicesDB described above, except that UploadWellsDB needs to read all the data from the uploaded files.

The user can upload a set of *wells* into the application, by pressing on the "Choose File" button beneath "Upload Wells locally" or "Upload Wells to database", shown in Figure 28. A user must upload an excel sheet contained in CSV file and at least one well log. If they do not follow this requirement the application will give the user an error message that explains the issue. Figure 29 and Figure 30 shows two different error messages that can occurred based on what a user has not uploaded.

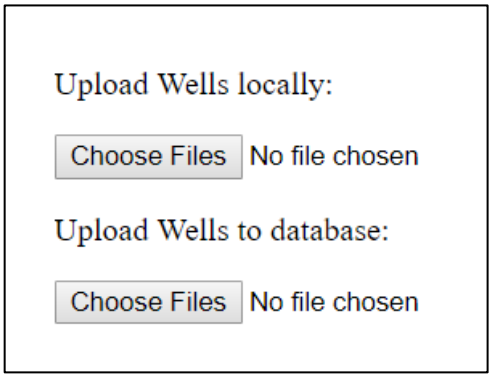

Figure 28: Buttons for uploading wells

Figure 29: Alert message waring for missing excel/csv file

Figure 30: Alert message warning for missing rms file

The uploaded data from the excel sheet and well log are stored in a *WellModel* or *SubWellModel* objects. The UploadWells component creates a list of Well components for all the *Well* that was uploaded. When each Well components is defined it has been given a *prop* that consist of a *WellModel* object. The UploadWellsDB component send the data to the back-end server instead.

4.11 Navigation

An important aspect of an application is how a user can navigate through it. The point of the navigation is to make it easy for a user to see what they want. In particular, this subchapter will explain how a user can move within the scene. The application has an React component called Navigation that handles many of the options for navigating within a scene. X3DOM supports default navigation by using the mouse. This allows a user to have full control over the scene's location and orientation. For this application, this option turned out to be a bit overwhelming for a user. It was common for a user to reach unhelpful orientations, for instance orientations that were off-center. Because of this some of the navigation has been limited, to ensure a better user experience. X3DOM has a node called *NavigationInfo*, that contains value for how the navigation works in the scene. An X3DOM can have multiple *NavigationInfo* nodes, but only one can be active at any given time. The node has a field

called *explorationmode*. If the *NavigationInfo* node in not defined, the scene will automatically create a node of this kind with the *explorationmode* field set to the value “any”. When defining a *NavigationInfo* node for this application the *explorationmode* field’s value has been set to “pan”. This will restrict the mouse control to only be able to move along the x, y and z axis from the current viewpoint. Which means the user cannot change the orientation or rotation of the scene with the mouse. The *NavigationInfo* node also has other fields that can be defined. An example of this is *speed* that determines how fast the movement is.

Even though the user’s ability to change the orientation with the mouse has been restricted, there are still options to change it. In X3DOM a viewpoint node can be used to determine the camera’s position and orientation. Similar to the *NavigationInfo* node, only one of them can be active at any given time. In the Navigation component, five viewpoint nodes are defined. All of these viewpoints have a fixed value for five different orientation that cannot be change during runtime. The first viewpoint has an orientation that will make the camera look down towards the terrain. This would be like looking down from a helicopter. X3DOM will automatically set the first define viewpoint as the active viewpoint. The four other viewpoints define orientation that would resemble a person view standing on the terrain. The four viewpoints are facing towards north, south, west and east. A user can change between these viewpoints by pressing buttons in the application. These are “Map”, “North”, “South”, “West”, and “East”. Figure 31 shows these buttons in the application. In addition, the user can press the an “Rotate” button, which will rotate between the four grounded viewpoints.

Figure 31: Navigation Buttons

The application supports other forms of navigation. One of which is by using the keyboard. The keyboard button that are supported is “W”, “A”, “S”, “D”, “Q”, “E”, “R” and Shift keys. The Navigation component has a method that will run when there is an input from the keyboard. “W”, “A”, “S”, “D”, “E” and “Q” are used to change the position of the view. The position of the five viewpoints mentioned above, are determined based on React *states* for each of the three axes. The “R” key can be used to reset the position, back to the original *state*. The Shift key has the same function as the rotate button mentioned above.

In the application there are a drop-down list, that contains the name of all the elements currently in the application. When pressing on the element, the viewpoint will change to the position such that the element is placed in the center of the scene. The element’s viewpoint is defined in the element’s component. When the application starts, the only element available is the terrain, called map. When a user uploads *slices* or *wells*, the dropdown menu will update to include the newly added element. Figure 32 shows the drop-down list when the application has a few elements added.

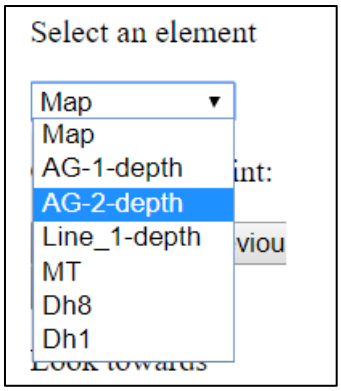

Figure 32: Drop-down list containing viewpoints to elements

In addition, some features of X3DOM has been disabled to ensure a better user experience. The first feature is when a user double clicks on the left mouse button the camera would move to the position in the scene where the mouse pointer currently is. This was removed because a user could accidentally do this when trying to click on some of the elements in the scene. The second feature is that X3DOM has some shortcut keys that are useful for development. These buttons have been disabled to avoid interference with the navigation keys used for this application, and to avoid a user accidentally activating them. For instance, the "D" key would usually open the developer log for X3DOM. This could be confusing for a user. Additionally, each element in the application, such as the *wells* or *slices* has been given a viewpoint X3DOM node.

A user can also change the viewpoint by pressing the next or previous viewpoint buttons. X3DOM stores all the viewpoints defined in the scene in a list and it can be used to find the next or previous viewpoint based on the placement of currently active viewpoint in the list. To be able to do this the Navigation component needs a reference to the *x3d* node in the App component. The easiest way to do this is to give the *x3d* node an id value. With this id the Navigation component can get access next or previous viewpoint by using the method *e.runtime.nextView()* or *e.runtime.prevView()*, where *e* is the X3D context found from the id.

There is also a button called "Return to element" to go back to the element with the currently selected viewpoint. This can be useful if a user has moved around in the scene by using the mouse. This is possible because when navigating with the mouse the current viewpoint will not be modified. However, this showcases a bit of a problem with the application, as there is no way to recognize the current camera position when using the mouse for navigation. By comparison, when pressing on one of the keyboards buttons the active viewpoint will change to the previously used viewpoint from the Navigation component. Each button press will alter the position value of the active viewpoint. The consequence of this is that if a user press on the keyboard button right after they have used the mouse for navigation, the camera will be moved from the position of the viewpoint rather than the current position of the camera. Figure 33 shows the buttons for these three functionalities, as well as the rotate button mentioned earlier.

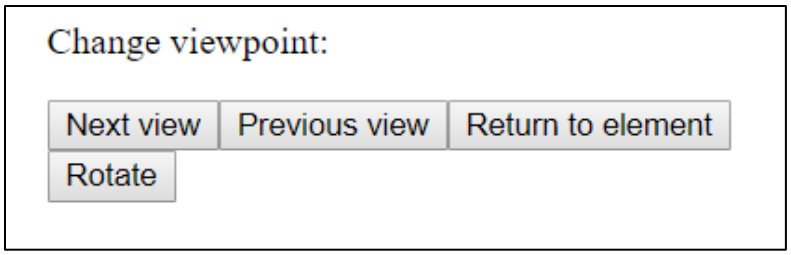

Figure 33: Buttons for changing viewpoints

Finding existing viewpoints like by using the navigation UI buttons work fine for the most part. However, since the viewpoint is related to a specific element, one need to be careful to not alter the viewpoint because then the application will lose the original data for that viewpoint. This can happen because of a bug within the X3DOM framework itself. If a user changes the viewpoint too quickly during the transition animation that occurs when the camera moves between viewpoints, the bug can happen. The bug can result in that one of the viewpoint's values are changed. To avoid this the application disables all the UI navigation buttons for a short time during the transition animation.

4.12 Volume data and rendering of multiple slices

The application explores the possibility to visualize volume data from existing nodes within the X3DOM framework for geological data. In X3DOM volumetric data is a set of 2D texture images that are aligned in a row [34]. All the texture will together create a 3D space in the shape of a box. The height and width of a texture details the xy-plane, while the number of textures is used to define a z-axis. Within X3DOM this process is referred to as volume rendering. An example of what volume rendering can be used for are a representation of a human torso as shown in one of the example in the X3DOM documentation [35]. This is seen in Figure 34.

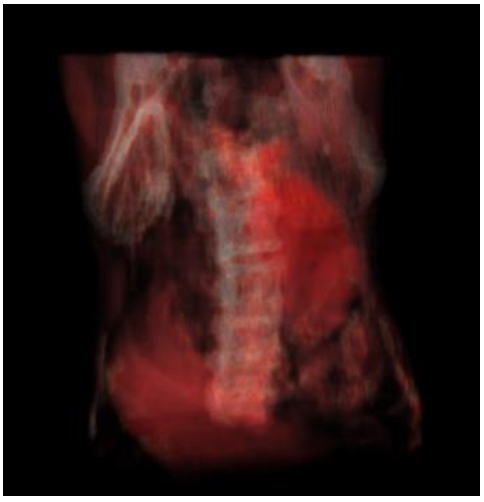

Figure 34: Shows a representation of a human torso visualized by using X3DOM volume rendering

For this application, the idea is to show multiple slices of seismic data that are closely placed next to each other. Each slice should be shown as an image texture. Rather than using the texture to create a 3D shape, a user should be able to toggle which texture image they want to look at. A user should be able to use a slider to decide which slice they want to look at. The slider will correspond to the order in which the slices are aligned in the volumetric area. For instance, if the slider is located in its leftmost position the slice at the lowest z-position should be shown. This slider can be seen in Figure 35.

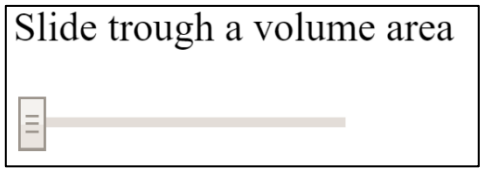

Figure 35: Shows the slider used to select a slice

CMR has not given any data for this kind of data. Because of this the examples in this subchapter only shows a concept of how this could be used. Volume rendering is explored because it can often be useful to compare data from slices that are near each other. This subchapter details two different approached to visualize multiple slices, each having their own pros and cons. The first approach uses existing X3DOM nodes that have the purpose of creating volume data. The second approach extends the Slice component so it consists of several images, representing different slices of seismic data. The images used for visualizing these two approached comes from an example on the home page for the X3DOM framework.

The first approach uses the *VolumeData*, *MPRVolumeStyle* and *ImageTextureAtlas* X3DOM nodes. Code Listing 14 shows a how the three nodes can be used to create volume. The *VolumeData* consist of a field called *dimensions*. The field is used to define the height, width and depth of a 3D area where the volumetric data can exist. Within the *VolumeData* node there needs to be a X3DOM node that is used for specifying the visual rendering style of the VolumeData. This approach uses the *MPRVolumeStyle* for the rendering operation. The reason for this is that *MPRVolumeStyle* was the only style, that had support to only look at one image used for the *VolumeData*. The *MPRVolumeStyle* node contains a field called *positionLine* that can be used to determine which image should be shown.

```
<volumedata dimensions='2.0 2.0 2.0'>
    <imagetextureatlas url="resources/Volume/kansai_pawr_20120726175907.png"
        numberOfSlices="64" slicesOverX="8" slicesOverY="8">
    </imagetextureatlas>
    <mprvolumestyle positionLine={this.props.pos}>

    </mprvolumestyle>
</volumedata>
```

Code Listing 14: Volume Data

In addition to the *MPRVolumeStyle* node, the *VolumeData* contains the *ImageTextureAtlas* node. An image texture atlas is an image that contains collection of smaller images. The smaller images are placed uniformly in rows and column of the same height and width. Each image has its own unique index value. The *ImageTextureAtlas* node contains four fields called *url*, *numberOfSlices*, *slicesOverX* and *slicesOverY*. The *url* field specify the location of the image texture atlas. The *numberOfSlices* field specify how many smaller images the image texture atlas contains. Finally, the *slicesOverX* and *slicesOverY* fields are used to define how many rows and columns are used to contain all the images. These fields are important to define because the texture atlas need this information to know where the images should be separated. Figure 36 show the *ImageTextureAtlas* that is used in Code Listing 14 and also provides an example of how an *ImageTextureAtlas* can look like. The *ImageTextureAtlas* consist of 64 smaller images. Each of these images represents a slice, and has a given index based on its position in the *ImageTextureAtlas*. The slice that will be shown in the scene, will be the slice that has the same index number as the value used in the *positionLine* field in the *MPRVolumeStyle* node. Because of this the *positionLine* field should have a value between 0 and 63.

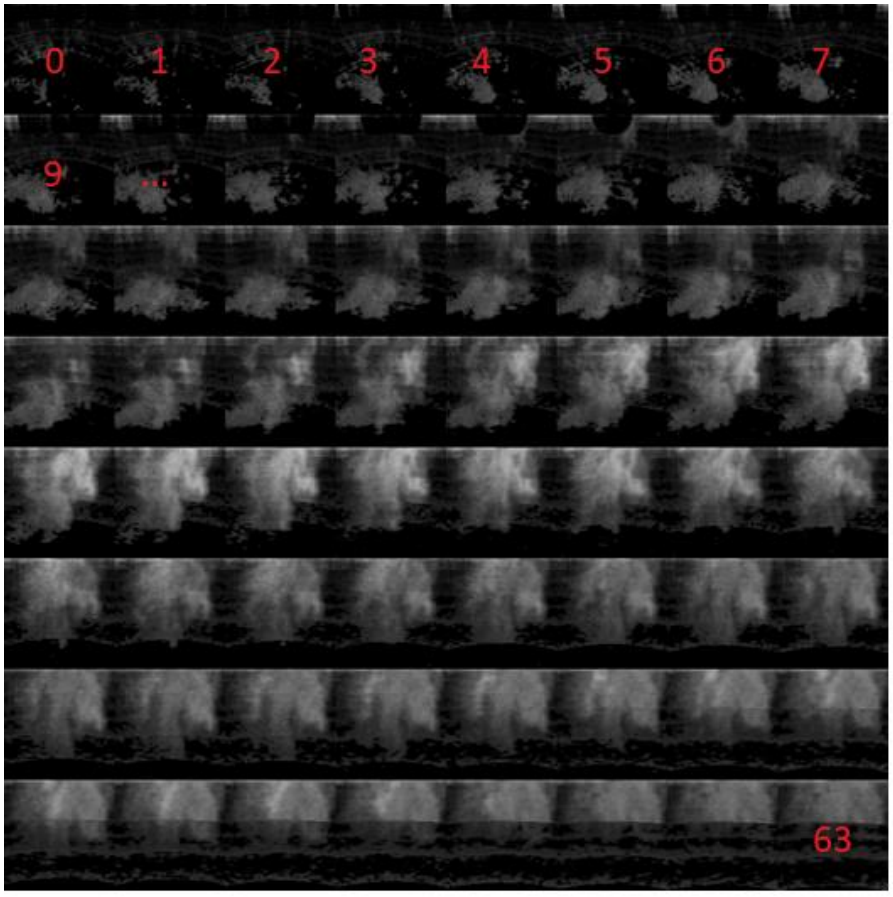

Figure 36: Example of a texture atlas containing 64 smaller images. Some of the images has a number showing their index

A user should be able to determine the value of the positionLine, by using a slider. The slider should have the range between lowest and largest number index in the *ImageTextureAtlas* (which in this case is 0 and 63). Figure 37 and Figure 38 shows two different slices and the corresponding slider position.

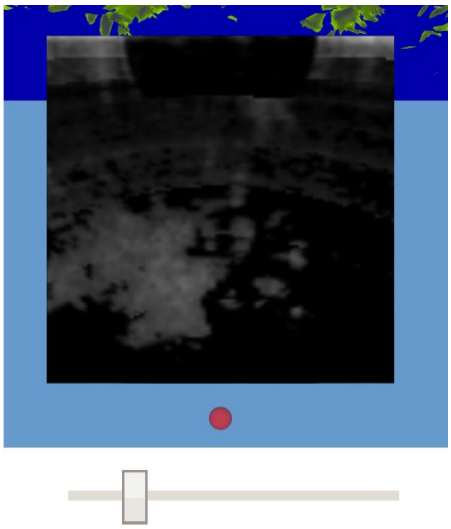

Figure 37: Shows the slice corresponding to an id from a slider

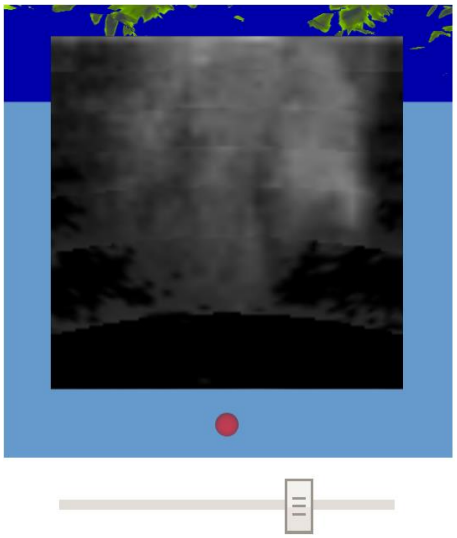

Figure 38: Shows another slice with a different id corresponding to a different slider position

A problem that was noticed by using this approach is that some 3D objects like the slices, will be hidden behind the invisible volume area. Figure 39 shows an example of this issue. Here some parts of the slices are obstructed by the invisible volume area that is placed in front of them. Attempting to find a solution for this problem was the reason for exploring the alternative approach for this task.

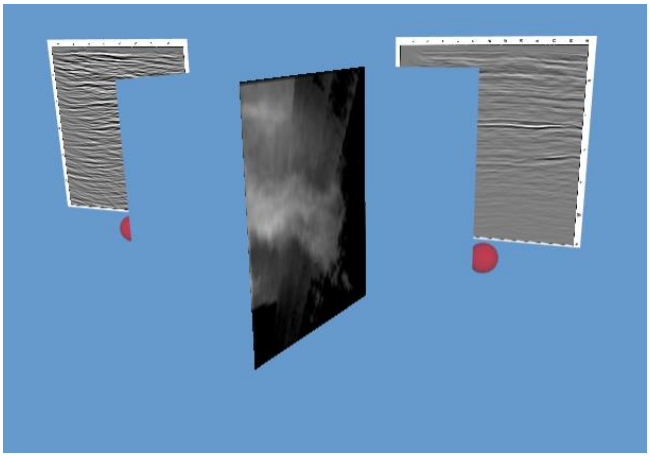

Figure 39: Volume Area hides two slices behind it

The second approach is to extend the Slice Component mentioned in subchapter 4.4. In this approach all the images from the volume dataset are stored as separate image files instead of an *ImageTextureAtlas*. The images are placed in a folder and has the indexed number as its name. The Slice component uses a new *prop* called *currentSlice* to determine which images should be used as a texture for a slice. Additionally, the *SliceModel* object that a slice uses contains a list of image URLs instead of just one. The value of the *currentSlice prop* is determine from the value of a slider with the range between 0 and the number of images for the Slice component. A user should be able to use the slider. This approach archives the same result but requires a large collection of files. This approach does not have the problem with some elements being hidden.

4.13 Database and handling of data

This application uses a MongoDB database to store data [36]. The database can store three types of data. These are well logs, *slices* and reference to the location of image files. For each of them the back-end consists of a model, that is used to create a schema/table that can be stored in the database. These schemas are made in simple JavaScript Object Notation (JSON) format, which is supported by MongoDB. JSON is text syntax that can easily be used to exchange data between a browser and a server [37]. The back-end consists of a local server, that is used to connect to the MongoDB database.

Figure 40 show the route for when a user uploads data to the application, to when the user can look at the uploaded data. The arrows represent the direction the data is sent to. First a user can upload the data from the UI in the web browser. After the data has been uploaded by the user, it will be handled by the UploadSlicesDB and UploadWellsDB component (see subchapters 4.9 and 4.10 for more information). These components will post the data to a back-end server.

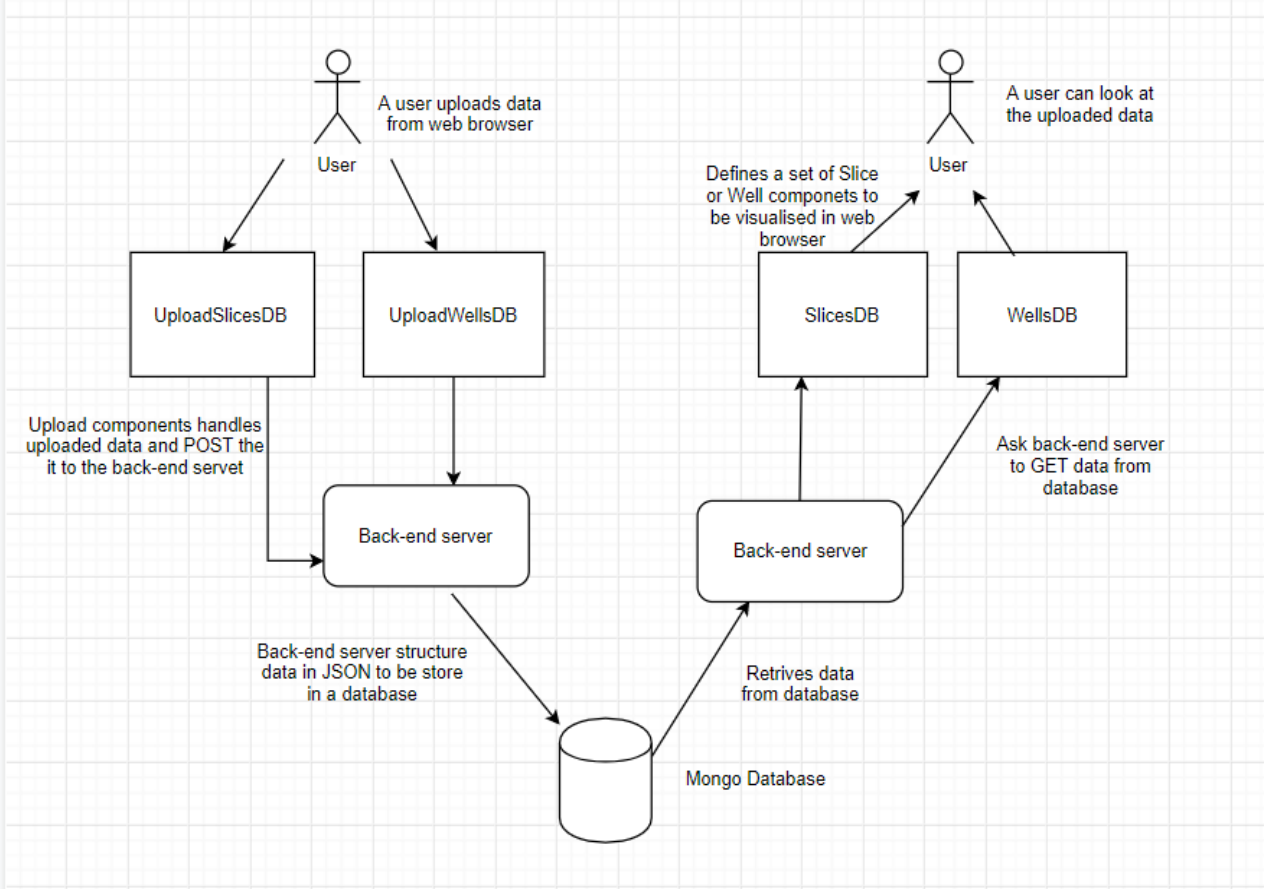

Figure 40: Shows the route from when data is uploaded by a user, to when a user can look at it

The back-end server retrieves the posted data that a user has uploaded. Then it creates a schema with the data, that can be stored in a database. There is a schema for each of the three type of data. They are called *wellSchema*, *sliceSchema* and *imageSchema*. The back-end server can also retrieve the data back from a database, to the front-end. For this to happen it needs a get request from the front-end. These operations are done by the express framework in the back-end, and axios in the front-end.

If the data is stored in the database, the *slices* and *wells* will be placed in the scene when the application has been loaded in the web browser. The application consists of two React Components that gathers data form the database. They are called SlicesDB and WellDB and are used for defining Slice and Well components based on the data gathered. They have an axios get request in both the *componentDidMount* and *componentDidUpdate* methods. The *componentDidMount* method will make sure that the data from the database gets shown, when the web browser after the initial loading or reloading of the web page. While the *componentDidUpdate* method will make sure that when some update happens within the component it will get the most recent data from the database. This means that when a user uploads element, the newly created element would appear right away, and there is no need for a page reload. While there currently is no function within the application that allows for deletion of elements, the elements will disappear from the web browser during runtime if they are deleted manually from the database. This is possible because the get axios request is placed within the *componentDidUpdate* React lifecycle method. This shows that the application can delete elements without the axios request having any knowledge on how the element was removed from the database. For this reason, functionality for deletion could successfully be placed anywhere within the application.

4.14 Structures for React components

This subchapter goes into detail of how the components in the application works together, if one would want to check each component in more detail see the subchapter for that specific component. Figure 7, from earlier in the chapter, presented an overview for how some of the components are structured. This structure only showed one of the uploading approaches for each type of element. This subchapter will show the structure for the other approach used when uploading components. It will also provide examples of how some components can be used in a different structure from the one used in this application. The different approaches highlight how the components can be reused for different situations.

Figure 41 shows the structure of the component used when a set of *slices* or *wells* are uploaded to a database. Because the *UploadSlicesDB* and *UploadWellsDB* does not need to render X3DOM nodes, they do not need to be placed within the scene. In this case approach the SlicesDB and WellDB has the responsibility to define the Slice and Well components that should be rendered. Code Listing 15 shows the scene inside the render method of the App component.

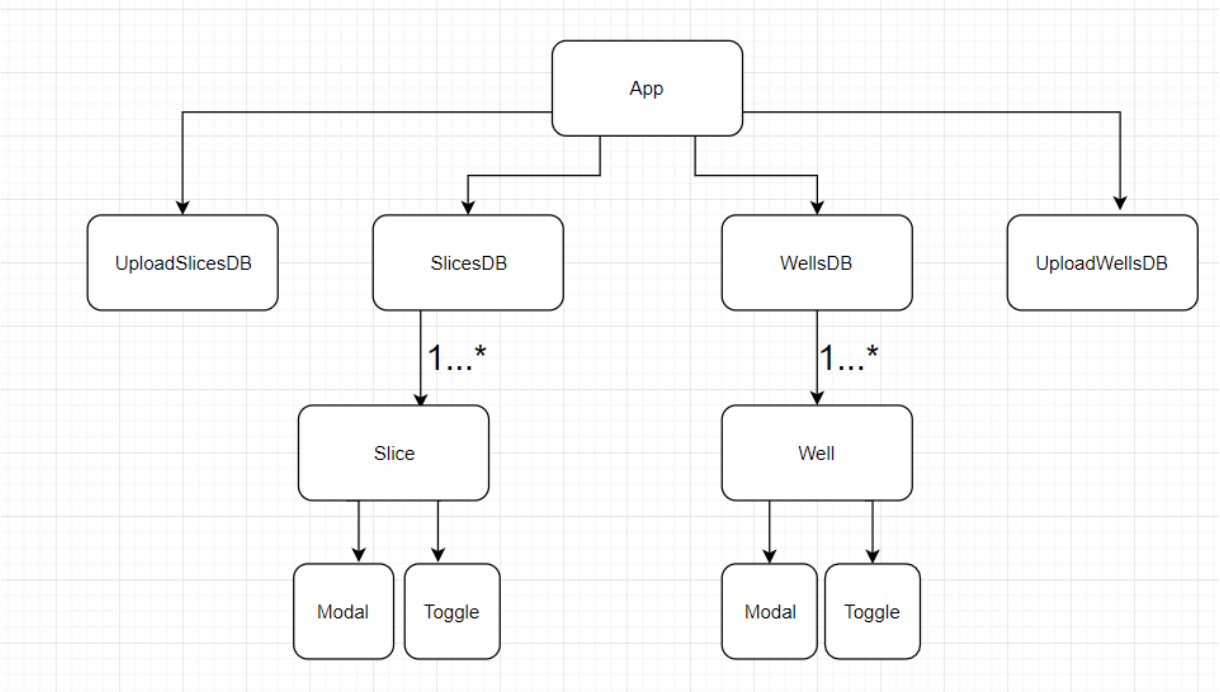

Figure 41: Component structure for Slices and Wells upload to a database

```
render() {

    return (
        <div>
            <x3d width={width} height={height} id="x3d_context">
                <scene>
                    <background skycolor="0.4 0.6 0.8"></background>
                    <viewpoint id="over" position='0 0 1000' description="Map"/>

                    <Terrain name="Svalbard" render={this.state.terrainVisible}>
                    </Terrain>

                    <SlicesDB></SlicesDB>
                    <WellsDB></WellsDB>

                    <Navigation></Navigation>

                </scene>
            </x3d>
```

Code Listing 15: Shows the rendering of the scene from the render method in the App component for uploading elements to a database

Figure 42 shows an alternative way to use the Well and Slice components. This example consists of two Slice components and two Well components. These components are defined directly within the *scene* in the App component. In this example the application consists of four predetermined elements, rather than let a user be able to upload their own elements. This shows that the Slice and Well components can be used in many different scenarios and be defined within different components. The only thing that is required is that the Slice component has a *SliceModel* object as a *prop*, that consist of the required values for rendering a *slice*. Likewise, the Well component needs a *WellModel* object. Code Listing 16 shows the render method placed inside an alternative App component.

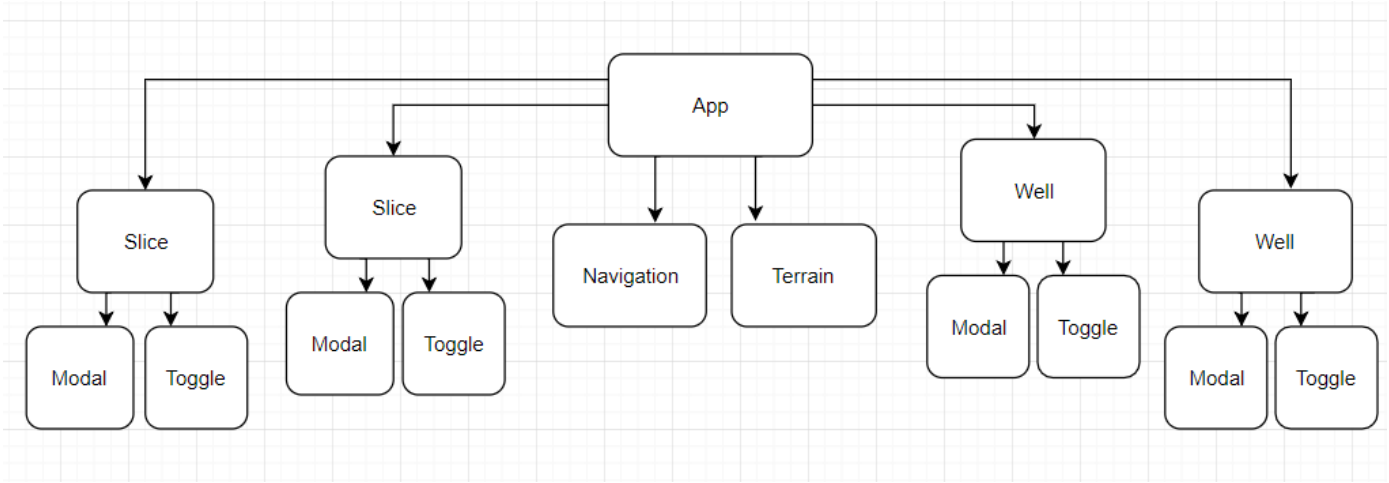

Figure 42: Alternative possible way to use components

```
render() {
    return (
        <div>
            <x3d width={width} height={height} id="x3d_context">
                <scene>
                    <background skycolor="0.4 0.6 0.8"></background>
                    <viewpoint id="over" position='0 0 1000' description="Map"/>

                    <Terrain name="Svalbard" render={this.state.terrainVisible}>
                    </Terrain>

                    <Slice slice={someSliceModel1}></Slice>
                    <Slice slice={someSliceModel2}></Slice>
                    <Well well={someWellModel2}></Well>
                    <Well well={someWellModel2}></Well>

                    <Navigation></Navigation>

                </scene>
            </x3d>
```

Code Listing 16: Shows the rendering of the scene from the render method in the App component for alternative structure

5. Result and Discussion

This chapter will detail the results achieved during the work on the thesis. The chapter will also discuss the decisions behind the implementation for the application created in this thesis. It will discuss the principles used to make the application more modular, as well as compare it to the implementation to the previously made application by using Angular. Finally, the chapter will provide a technique for how a developer can implement a React component that can be used to visualize a 3D object created from X3DOM nodes.

5.1 Result

In the work of this thesis an application has been created, by using the React and X3DOM tools. This application contains a scene that is able to visualize the 3D topography of Svalbard, and various data gathered from drilling beneath the surface. A user can upload their own data to the application. This data can either be shown only during the duration of a web page being open or be stored in a database. The application also allows a user to navigate with the mouse, keyboard and UI buttons. In addition, a user can toggle whether elements within the scene should be visualized or not, and get additional information concerning an element by clicking on it. Furthermore, the possibility of volume rendering has been explored, and two approaches has been assessed, with different pros and cons. This project did not have access to any resources concerning volumetric data for geology, so these approaches only show a concept of how it may be done.

The first approach is using the X3DOM nodes for volume rendering got all the information it needs from one image file, using by dividing them using an *ImageTextureAtlas*. This could be a benefit, if the application would give the user an opportunity to upload their own file, as they do not need to worry about missing certain files during the upload. On the other hand, an issue using this method was spotted. The area of the volume box defined by X3DOM could sometimes hide certain other elements placed behind it (in relation to the user's viewpoint) in the scene, even though the defined area is invisible. This is not an issue with the other approach, because it does not define an area of volume, but rather only renders the current selected *slice*. However, this approach uses, a set of image files, that needs to have a name that can be identified based on user interaction. If a user should have the ability to upload this, it would require a more complicated system to store them correctly. Volume data could also be created from a set of data values instead of given by images, especially the dataset is large. Neither of these approaches support this feature.

5.2 Modular approach

One of the goals for this thesis is to explore the possibility to implement the application in a modular fashion. The advantages of this is the main focus of this subchapter. To achieve these benefits, the application needs to follow design patterns that are known to increase the modularity of the design.

The concept of modularity in software design is to separate the different responsibilities of an application into a set of smaller part [38] [39]. Each of these parts is referred to as a module and ideally should handle just one responsibility.

The advantages of designing an application using modularization can lead to: [38] [39]

- Less code
- Reusable code
- Easier to manage within a team
- More understandable code
- Errors should be easier to identify
- Code can be used across different applications

In this application the code has been divided into several different React Components, which serves the same idea as a module. Each of these components focuses on one particular task, such as the code needed to render one type of 3D object. The application follows the Single responsibility principle [40], which is the first principle of SOLID which was first introduces in "Design Principles and Design Patterns" by Robert C. Martin [41]

The single responsibility principle (SRP) are mostly followed within the components in the application. First and foremost, the application consists of components focusing on visualizing a corresponding element such as the *slice* or *well*. The Slice component could easily have worked with additional responsibility, such as toggling the *slice*, calculating their positions, opening and closing their dialog prompt. However, these responsibilities have been separated to their own components or classes to ensure the SRP, which leads to more modularity within the application. There is however always room for improvements, for instance currently the coloring calculation for the *wells* is handled by the Well component. This could, and probably should, be separated to ensure more modularity within the application. As shown by the application's ability to calculate positions separately this should probably be possible to do, without any major design changes.

By dividing the code into components, they can easily be reused in other part of the application, which is desirable. For instance, if the application did not have the *well* component, the code would need to consist of multiple instances of the same code to create them within the App component. This leads to another benefits which is that the application contains less code. During the development of this application, it has been easy to identify errors. For example, if the project failed to run, it would specify in the console on the web browser in which component an error did occurred. If the application runs, but one of the elements did not render as expected, the fault usually was in the component related to that element.

Another principle of the SOLID used during development was the open-closed principle [42] (OCP). The idea behind this principle is that “Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification” [42]. In this project the entities would be the React components. A component is considered open when you can add functionality, but at the same time, the changes should not lead to a change in other part of the application that uses them. The components in this application follow this principle because during development the storing and creation of certain component changed during development. An example of this is the development of the Slice component.

Two important concepts to consider when designing a modular application are the cohesion and coupling [38] [39]. Coupling is a term used to determined how dependent on the different modules in application are on each other [43]. To ensure that the that the modules are independent from each other, the coupling between them should be low. This can also be referred to as loose coupling. The different modules in this application still needs some coupling to send or define parameter to each other. This is called data coupling and is considered as a loose form of coupling [43]. Cohesion is a term used to determined how well the different modules focuses on their particular task or responsibility [44]. An application has high or strong cohesion if each of its modules focused only on a particular task. Two types of cohesion that are considered strong are object cohesion and functionality cohesion [44]. The application follows the object cohesion pattern as each React component is focused on creating one particular type of 3D object. The application has some instances of functionality cohesion as certain functionalities, such as the UTM coordinates converter that are separated from the components. However, the usage of functionality cohesion could be improved, such as with the coloring calculation mentioned before.

The application created for this thesis follow these two principles for the most part. The application has high cohesion because each React component works as its own module. An example of this is the Slice component, that focuses on visualizing a *slice*. The application follows the low coupling principle by not being restricted by the other components in the application. This is shown in subchapter 4.14, where an alternative approach for how the components can be structured are presented. In particular, it shows that one does not need the components for uploading files to create a Slice or Well component. In addition, the application can reuse certain components that are useful for different scenarios. An example of this is the Toggle component, that can be used to create a toggle for both the Slice and Well components. There is however one restriction concerning the coupling in this application, which are that the component using the X3DOM framework needs to be located within a *scene* node. If X3DOM tags are placed outside a *scene*, it will not render and could potentially lead to an error. To reduce this problem, the *scene* should be easily accessible for all components that renders X3DOM code. Because of this the *scene* node is located in the App component, which also serves as the highest level for the application. Alternatively, the *scene* node could be placed within the HTML document itself. All the components should be able to work on any *scene* from any React application, if the file has access to the component either by having the component in the same file or by importing it from another one.

5.3 Comparison between Angular and React solutions

This subchapter goes into detail how the React application differs compared to the application made by Malt, using Angular. Firstly, this subchapter will go into details on decisions concerning the design of the application. These changes do not necessarily benefit from using React, rather they are different approaches to solve already existing working task. Secondly, the chapter will go into detail of issues that occurred during development of the Angular project, and how React can be used to avoid them. Finally, the chapter will go into detail on additional created functionalities that were not present in the previous application.

5.3.1 Design decisions

In this thesis some of the features of the solution from the Angular project has been rebuilt. When doing this, the design decisions for the application has been reevaluated to determine whether they should change or not. First up are the terrain and *slices*, which has been visualized by using the same set of X3DOM nodes as in the Angular project. With terrain this process is done by the *BVHRefiner*. A *slice* is visualized by using the *shape* node and *box* geometry node and has an image used as a texture.

However, when it comes to visualizing a *well* the two applications have very different approaches. In the Angular application a *well* was visualized by a set of cylinders. Each of the cylinders was a 3D object. The cylinders were created by using the X3DOM *cylinder* geometry node. The cylinders serve a similar role as what was called a *subwell* or a *segment* in the React application. In the React application a *well* consist of one 3D element formed by the *IndexedFaceSet* node. This might be a little more complicated to implement, as the values must be placed in a very specific order to be visualized correctly. The benefit of this approach is that it allows for more options when visualizing a *well*.

One example of this is the possibility to visualize a *well* from a well log that contains UTM coordinates that are placed diagonally from each other. To do this with a cylinder, it would require a way to calculate the rotation of a cylinder and ensured that the top and bottom face of a cylinder is in parallel, with the other cylinders in different rotations. It is possible to define a rotation of a cylinder by using the orientation filed in a *transform* X3DOM node wrapped around a cylinder X3DOM node. However, it does not provide a way to ensure that the base of one cylinder is aligned with the top of another cylinder with a different rotation. A mock-up of this problem can be seen in Figure 43. The issue here is that some parts of the cylinders will merge into each other, while other parts will leave a hole. Figure 44 displays the *well* when using the *IndexedFaceSet*. With this approach the two segments do not have the same issues, and the *well* looks smoothly connected.

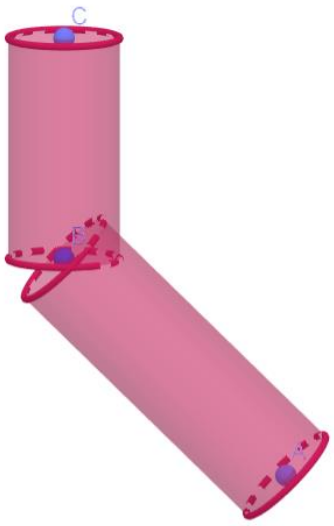

Figure 43: Mock-up of cylinders

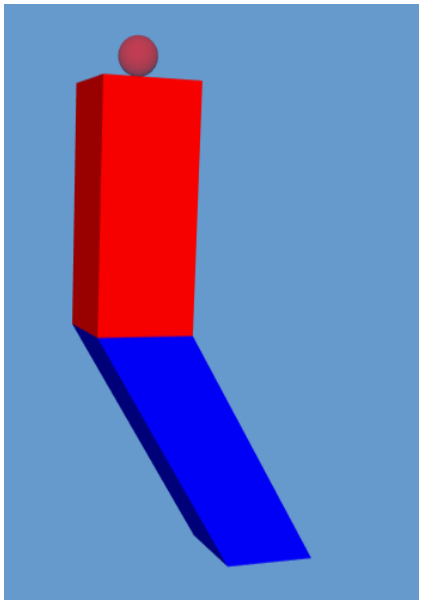

Figure 44: A well with a diagonal segment

The approach for toggling elements is also a bit different. In the Angular application, the user had the ability to toggle between whether an element should be rendered or if a substitute sphere should be rendered. In the React application this has been changed to each element having a relatively small toggle in the form of a sphere close to the given element. Additionally, a user can toggle off the visualization of a *slice* or *well*, by pressing on it with the left mouse button. Naturally, this is only possible when the element is currently visible.

5.3.2 Using React to resolve issues encountered with Angular

During the development of Malt's solution, several issues occurred. The reasons for these issues could be traced to problem with trying to make the Angular and X3DOM frameworks work together.

One of the biggest inconveniences when using these two frameworks together, is that it is not possible for non-X3DOM elements to be used within the X3DOM nodes. For instance, one cannot wrap X3DOM nodes within a div-tag, and then place it within a X3DOM *scene* node.

Malt commented on this in his thesis:

> *The main issue when using X3DOM inside a web framework that creates custom HTML elements in the form of components, is that non-X3DOM elements can't be used inside the X3DOM scene. This causes X3DOM to fail at reading the scene and produce an error, since X3DOM only understands its own syntax. Because of this, components created with a web framework, that do not have a name which match the name of an X3DOM node, cannot be used to wrap X3DOM elements inside the scene.*
>
> Using 3D functionality available in current web-browsers to create and visualize geological models, Ø. Malt (2017) [1], p. 27

However, with newer versions of React this is no longer an issue. React gives the opportunity to contain X3DOM nodes wrapped within a React *fragment* tag, which can be placed within a *scene* node. A *fragment* in React is a pattern that can be used to wrap several groups of children nodes, without adding the *fragment* itself as an node to the DOM [45]. In particular the React *fragment* is used to wrap all the X3DOM tags used within the render method of a React component. To work around this problem Malt created an attribute selector to be able to change values of the fields in the X3DOM nodes. This resulted to a more complex implementation. It also reduced the application's ability to be modular because the Angular component is depending on the attribute selector. Specifically, it becomes less modular because the components need to handle both the attribute selectors and the visualization of the element itself. Furthermore, the attribute selector had to be wrapped around in an X3DOM *<transform>* tag, rather than simply referring to a custom-made React component with any name. This also leads to an increase readability with the React solution compared to the Angular project. According to the React documentation, there are plans to allow for onClick attribute for the React *fragment* tags in future version of React [45]. This could potentially make it easier to add event handlers that can affect X3DOM elements in the future.

Another problem that was noticed during the development of the Angular project, was that the eventListner could lead to errors in the application. This was suspected by Malt to be because the eventListner was added before the component was loaded. [1] In the beginning of the development of the React application, a similar issue was encountered. An event would not be triggered by an *onClick method* that was added directly inside a X3DOM tag. The reason for this can be explained by looking at the documentation for X3DOM as mentioned in 4.6. However, React supports a feature called refs, where one could reference an rendered node. This allowed for the possibility to add an event handler after the initialization of the X3DOM scene and objects. This is done by using the React lifecycle method *componentDidMount* (see 4.6 for a general implementation of this). When using this approach, there were no longer problems with events not happening when expected. Both problems listed above answer RQ2, as they exemplify two good benefits of using React, that can be used to avoid the issues mentioned above.

5.3.3 Additional functionalities

In addition to remaking and improving the features discussed above, some new ones have been explored. Firstly, the image used for a *slice* has been added to the *slice*'s dialog prompt. This allows the user to get a better look at the details of the image. Additionally, a user can press on the image, which would result opening an additional tab in the web browser. This tab consists of the URL to the image clicked on and presents the user to the image in its original resolution. In addition, the React application allows a user to toggle the terrain. Furthermore, a user can press on a button called "Toggle Slices" and "Toggle Wells". Unlike the dedicated *toggles* for each element, these buttons allow the user a straightforward way to hide all elements of the chosen type, including the *toggles* for the individual elements. This strengthens the user's ability to decide what they want to see. The benefit of this option is particularly noticeable whenever there is a high density of elements. To many active elements on the screen can make it harder to see and interact with the desired element.

Compared to the earlier solution, the navigation has been greatly improved to ensure a better user experience. This includes the ability to use navigation buttons in the web application, as well as a keyboard. Finally, two approaches for simulating volume rendering has been implemented. These allow a user to slide through a set of images that forms a volume.

5.4 General structure for a React component containing X3DOM nodes

This subchapter goes into detail of general guidelines when creating a React component containing X3DOM code. These guidelines are followed when creating the basic example shown in subchapter 2.3, as well as more complex component such as *slice* and *well* mentioned in subchapters 4.4 and 4.5 respectively. In Code Listing 17 on the next page, a general and generic structure of a React component is shown. Firstly, the component should import React, which will be used when defining the class as an extension. The component contains a constructor, render method, React lifecycle methods (such as *componentDidMount*) and a set of method made for this component specifically. In this example these methods are listed as "someMethod".

The constructor can be used to initial the components *states* and binds the methods. In addition, the *ref* should also be initialized in the constructor. It is possible to not do this, but the documentation of React strongly encourages it. For example, one could define the *ref* as string value, but the React documentation considers this as a legacy approach and it will likely be removed in future updates of React [20]. Binding methods in React allows one to reference them with "this" within the component, which make them easier to access.

The methods should be used to change or initialize the visualization of the component. They will often result in changing the *state* of the component. This is done by using *this.setState()* method. The React lifecycle should be used to set a property of the component at an appreciate time. For instance, the eventListner should be added in the *componentDidMount* method as explained in subchapter 4.6. If the components need to perform some calculations before the rendering of the component, this can be done in the componetWillMount. This is done in the *well* component because the component needs to determine the color of each *subwell* based on its properties.

The additional methods can be whatever is necessary for a given component. They can be run from anywhere within the components, such as when a keyboard or UI button is pressed, or if a certain value is true. The methods can be placed within the React Lifecycle methods or other methods in the application.

The render method consists of a return. The definition of a component will be replaced by the tags defined within the return. These tags should eventually be returned to a HTML document, where they are used for the creation of the web page. The render method's return supports JSX. This is a good fit for writing the X3DOM code, because the JSX syntax is supported in HTML after it has been returned. In addition, it allows for altering the properties in the X3DOM-tags. This can be done by encapsulating *props* and *states* in brackets. Finally, the component is exported, so that is can be used in other places in the application.

```
import React from 'react';

class SomeComponent extends React.Component {
    constructor(props) {
    super(props);

    this.state = {
      someState1 = {insertInitialState},
      someState2 = {insertInitialState}
        };

    this.elementRef = React.createRef();

    this.someMethod = this.someMethod.bind(this);
    this.someMethod2 = this.someMethod2.bind(this);
    }

    render() {
        return (
            <React.Fragment>
                {Write X3DOM Code here}
                {Create another react component}
            </React.Fragment>
        );
    }


    componentDidMount(){
        this.elementRef.current.onclick = () => this.someMethod();
    }

    someMethod() {
        ...
    }

    someMethod2() {
        ...
    }

}
export default SomeComponent;
```

Code Listing 17: General structure for a React component with X3DOM code

This structure can be used to create more than just elements within the field of geology. For example, this structure could be used to create a 3D humanoid robot. The robot could contain one high-level component for its entire body and component for each different kind of limb. The body should contain all the components for different limbs (directly, or indirectly through another component). A component could consist of several of the same component. For instance, the body could contain two arms components, each of which could contain five finger components. One could also make some function happen when pressing the different part of the body. For instance, pressing on the arm could move it 45° upwards or downwards.

6 Conclusion

In chapter 5 the result and the procedures of the implementation in the application was discussed. This chapter will use this discussion to conclude and evaluate whether the result is satisfactory. The conclusion will be determined based on if the goals for the thesis has been achieved and whether it can provide answers for the research questions. These goals and research questions were introduced in the subchapter 1.2 and 1.3 in the introduction.

The first goal of this project was to create a web application that could visualize the topography of Svalbard and data gathered from drilling beneath the surface of Svalbard, by using the React library and X3DOM framework. This goal has been achieved as shown in the solution detailed in chapter 4. In addition, the project was conducted to see whether it could fix some of the issues present in the previous solution that used the Angular framework. As discussed in chapter 5, several of issues were avoidable when using React. These issues include errors concerning event handlers and not being able to wrap the X3DOM nodes.

The React application is working without any major or noticeable problems on both the Google Chrome and Mozilla Firefox web browsers. Furthermore, most of the functionalities from the earlier application are still present, as well as some new ones. Among these new additions are an improved ability to navigate within a scene and toggle elements, which fulfills the fifth and sixth goals of the thesis.

The second goal in the thesis aims to design the application in a more modular way compared to Malt's solution. The discussion in chapter 5 details how the components are modular, by being reusable and independent. This has been possible by following design principles such as the SRP in SOLID. This is shown by being able to use the components in different part of the application without doing any changes to a component. In addition to achieving the second goal, this gives an answer for the first part of RQ1. This part concerns whether if it is possible to create 3D object and 3D geology in a modular way. The answer to this is "yes", as shown by the implementation of the solution. Furthermore, chapter 5 provides an example of how a React component containing X3DOM nodes can be build. This structure can be used to create component for any 3D object created from X3DOM, which includes elements used in 3D geology. This is supported by the fact that the implementation of the React application, follows the same structure for all its components. This answer the second part of RQ1, that consist of questioning how 3D objects and 3D geometry could be implemented in a modular way by using React.

The possibility of this leads to a couple of benefits, including the ability to easily divide 3D objects into different reusable React components. This goes back to RQ2, where the benefits of these components are put into question. The benefits of React components are that they are easy to manage, reuse and provide a general structure for how it can be used, as shown in chapter 5. As well as being a benefit for RQ2, the general structure provided here also give a solution for the third goal. As detailed earlier, the structure works for all the elements and examples discussed in this thesis. Moreover, an example for a use case outside the scope of this thesis is provided. Having a general structure is a huge benefit, as it leads to a simple-to-use approach that can be used in many scenarios. The React components are easy to manage because their function is to do one specific task and nothing more. When looking at the React components within a *scene* node in the App component, it is easy to understand what parts the application consists of, and where the corresponding code is handled.

The fourth goal of the thesis was to explore the possibility of volume rendering within the field of 3D geology. Two approaches were proposed, but as this project lacked access to real data of this type further work is needed to determine whether these approaches would be fully functional. While X3DOM offered a lot of options for visualizing volume data, most of them were dedicated to medical data. A feature wanted for visualizing seismic data is a slider to look at specific slices forming a volume. Only one of the volume styles offered by the X3DOM framework consisted of functionalities that could support showing individual part of the volumetric dataset with a slider. This approach had an issue where it could obstruct the visibility of other elements in the application. The goal was to determine the possibility, which seems probable. The idea certainly seems to merit further investigation, but certain restrictions have been identified that could make it challenging.

The third and final research question, asked if there were any benefits to using React compared to Angular. In this thesis several solutions have been presented for problems that were experienced when using Angular. In that sense, the solutions provided in this thesis show a benefit for using React, as it uses techniques not available in Angular. It is, however, not possible to conclude that these issues are necessarily unsolvable when using Angular. For now, React is the only web library that can guarantee the ability to create event handlers on X3DOM nodes without risk of errors, and the ability to use React *fragments* that allows the X3DOM code to be divided into different modules.

7 Further Works

This thesis has achieved it goals, however a project can always be improved further. This chapter will present some suggestion in how the application could be further developed. The application can be extended by adding support for more ways to visualize elements.

Currently the files used to create the 3D topography of Svalbard is stored locally in the application. The reason for this, is because the process requires a large set of files used and the files need to be placed in a very particular structure. The application could easily use a different set of files to create another terrain. During development another terrain was used for testing which worked as expected. However, this might not align with the conversion of coordinates. This is important for placing other elements correctly in relation to the terrain. Ideally a user could be able to upload their own terrain, however this might be challenging to implement because of the conversion of coordinates would be different between different terrains. To achieve this the implementation of the conversion would need to be improved, and some way to store and gather the files needs to be implemented. Alternatively, the application could contain a set of predefined terrains, and the user can choose between them before rendering a scene. If this were to be implemented, one might need to consider which of the other elements is related to the different terrains, to make sure that only relevant elements get shown.

Another improvement could be to add access control to the application. For example, one might not want all users of the application to have the ability to upload data to the database. To achieve this a login functionality for admin users could be implemented. The application already features two approaches for uploading data. One way to implement this could be by splitting the two approaches for uploading between different kind of users. For example, the admin user could be able to upload files to the database, while a regular user could only upload to the currently open web page. In addition, functionality for a user to delete or edit elements could also be added. This should be fairly straightforward to implement because elements already can be deleted during runtime manually from a database.

It might also be beneficial to keep an eye on updates to the X3DOM and React libraries to see if new opportunities for improving the application arise. Finally, the application is currently only available on a local server. To make the application easily accessible to the users it needs to be put on an online server.

8. Bibliography

[1] Ø. Malt, “Using 3D functionality available in current web-browsers to create and visualize geological models.,” 07 June 2017. [Online]. Available: http://bora.uib.no/handle/1956/16264. [Accessed 25 March 2018].

[2] Angular, "Angular," [Online]. Available: https://angular.io/. [Accessed 25 March 2018].

[3] D. Abramov, "DOM Attributes in React 16," Facebook, 08 September 2017. [Online]. Available: https://reactjs.org/blog/2017/09/08/dom-attributes-in-react-16.html?utm_source=reactnl&utm_medium=email. [Accessed 29 April 2019].

[4] Medium Corporation, "React vs Angular vs Vue.js—What to choose in 2019? (updated)," 16 March 2018. [Online]. Available: https://medium.com/@TechMagic/reactjs-vs-angular5-vs-vue-js-what-to-choose-in-2018-b91e028fa91d. [Accessed 16 May 2019].

[5] CMR, "Virtual CO2 Laboratory - VIRCOLA," [Online]. Available: https://www.cmr.no/projects/10445/virtual-co2-laboratory-vircola/. [Accessed 31 October 2018].

[6] M. Ryen, "Vellykket lagringsforsøk på Svalbard," CLIMT, 8 April 2016. [Online]. Available: http://www.climit.no/no/Sider/Vellykket-lagringsfors%C3%B8k-p%C3%A5-Svalbard.aspx. [Accessed 29 April 2019].

[7] Ø. Malt, "Visualizing 3D geology in web browsers using X3DOM," 26 November 2017. [Online]. Available: https://ojs.bibsys.no/index.php/NIK/article/view/428. [Accessed 25 March 2018].

[8] Ø. Malt, "Visualizing 3D geology in web browsers using X3DOM," 2017. [Online]. Available: https://docs.google.com/presentation/d/1IMShuJCIhRqaD9uSG1AZ_NDhhQfgY72-7-WtdxemLWg/edit#slide=id.g2143ace4a2_0_50. [Accessed 25 March 2018].

[9] Paradigm, "SKUA-GOCAD," [Online]. Available: http://www.pdgm.com/products/skua-gocad/. [Accessed 7 November 2018].

[10] Schlumberger, "Petrel E&P Software Platform," [Online]. Available: https://www.software.slb.com/products/petrel. [Accessed 7 November 2018].

[11] Z. Wang, H. Qu, Z. Wu, H. Yang and Q. Du, "Formal representation of 3D structural geological models," *Computers & Geosciences*, pp. 10-23, May 2016.

[12] A. Arbelaiz, A. Moreno, L. Kabongo and A. García-Alonso, "X3DOM volume rendering component for web content," 2016.

[13] GitHub, "react-three-renderer," [Online]. Available: https://github.com/toxicFork/react-three-renderer. [Accessed 7 Januar 2019].

[14] web3D Consortium, "What is X3D?," [Online]. Available: http://www.web3d.org/x3d/what-x3d. [Accessed 31 October 2018].

[15] X3DOM, "Basic X3D Concepts: Nodes, Components and Profiles," [Online]. Available: https://doc.x3dom.org/gettingStarted/basicX3D/index.html. [Accessed 16 April 2019].

[16] X3DOM, "Background: What is X3DOM, and what can it do for me?," [Online]. Available: https://doc.x3dom.org/gettingStarted/background/index.html. [Accessed 31 October 2018].

[17] X3DOM, "Browser support," [Online]. Available: https://www.x3dom.org/contact/. [Accessed 2019 April 29].

[18] Facebook, "React - A JavaScript libary for building user interfaces," [Online]. Available: https://reactjs.org/. [Accessed 12 May 2018].

[19] Facebook, "Introducing JSX," [Online]. Available: https://reactjs.org/docs/introducing-jsx.html. [Accessed 7 May 2019].

[20] Facebook, "Refs and the DOM," [Online]. Available: https://reactjs.org/docs/refs-and-the-dom.html. [Accessed 31 October 2018].

[21] Node.js, "About Node.js," [Online]. Available: https://nodejs.org/en/about/. [Accessed 16 April 2019].

[22] Node.js, "Introduction to Node.js," [Online]. Available: https://nodejs.dev/. [Accessed 16 April 2019].

[23] npm, "react," [Online]. Available: https://www.npmjs.com/package/react. [Accessed 16 April 2019].

[24] GitHub, "axios," [Online]. Available: https://github.com/axios/axios. [Accessed 16 April 2019].

[25] mongoose, "mongoose," [Online]. Available: https://mongoosejs.com/. [Accessed 16 April 2019].

[26] Express, "Fast, unopinionated, minimalist web framework for Node.js," [Online]. Available: https://expressjs.com/. [Accessed 16 April 2019].

[27] npm, "multer," [Online]. Available: https://www.npmjs.com/package/multer. [Accessed 16 April 2019].

[28] A. Morgan, "Introducing the MEAN and MERN stacks," Mongo, 26 January 26 2017. [Online]. Available: https://www.mongodb.com/blog/post/the-modern-application-stack-part-1-introducing-the-mean-stack.

[29] T. F. Eriksen, "Visualizing 3D geology with React and X3DOM," GitHub, 24 May 2019. [Online]. Available: https://github.com/tomfjug/Visualizing-3D-geology-with-React-and-X3DOM. [Accessed 24 May 2019].

[30] MongoDB, "MongoDB Download Center," [Online]. Available: https://www.mongodb.com/download-center/community. [Accessed 23 April 2019].

[31] X3DOM, "Refining and Loading hierarchical data dynamically," [Online]. Available: https://doc.x3dom.org/tutorials/largeModels/BVHRefiner/index.html]. [Accessed 22 November 2018].

[32] Norwegian Polar Institute, "Terrengmodell Svalbard (S0 Terrengmodell)," [Online]. Available: https://data.npolar.no/dataset/dce53a47-c726-4845-85c3-a65b46fe2fea. [Accessed 7 March 2019].

[33] X3DOM, "Picking Objects," [Online]. Available: https://doc.x3dom.org/tutorials/animationInteraction/picking/index.html. [Accessed 22 November 2018].

[34] web3D Consortium, "Volume rendering component," [Online]. Available: http://www.web3d.org/documents/specifications/19775-1/V3.3/Part01/components/volume.html. [Accessed 22 November 2018].

[35] X3DOM, "Examples," [Online]. Available: https://www.x3dom.org/examples/. [Accessed 22 November 2018].

[36] MongoDB, "What is MongoDB?," [Online]. Available: https://www.mongodb.com/what-is-mongodb. [Accessed 16 April 2019].

[37] w3schools, "JSON - Introduction," [Online]. Available: https://www.w3schools.com/js/js_json_intro.asp. [Accessed 14 May 2019].

[38] K. Wendt, "Software Design: Modularity," 2017. [Online]. Available: https://www.coursera.org/lecture/software-processes/software-design-modularity-xH6BK. [Accessed 18 May 2019].

[39] C. Larman, "GRASP: Designing Objects with Responsibilities," in *Applying UML and Patterns*, Upper Saddle River, Prentice Hall PTR, 2005, pp. 316-318.

[40] objectmentor, "SRP: The Single Responsibility Principle," [Online]. Available: https://web.archive.org/web/20150202200348/http://www.objectmentor.com/resources/articles/srp.pdf. [Accessed 10 May 2019].

[41] R. C. Martin, "Design Principles and Design Patterns," 2000. [Online]. Available: https://web.archive.org/web/20150906155800/http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf. [Accessed 10 May 2019].

[42] objectmentor, "The Open-Closed Principle," [Online]. Available: https://web.archive.org/web/20150905081105/http://www.objectmentor.com/resources/articles/ocp.pdf. [Accessed 10 May 2019].

[43] K. Wendt, "Software Design: Coupling," 2017. [Online]. Available: https://www.coursera.org/lecture/software-processes/software-design-coupling-BNSI0. [Accessed 18 May 2019].

[44] K. Wendt, "Software Design: Cohesion," 2017. [Online]. Available: https://www.coursera.org/lecture/software-processes/software-design-cohesion-LWhWM. [Accessed 18 May 2019].

[45] Facebook, "Fragment," [Online]. Available: https://reactjs.org/docs/fragments.html. [Accessed 16 April 2019].