Diese Arbeit wurde vorgelegt am
Lehr- und Forschungsgebiet Theorie der hybriden Systeme

Auralisierung und Visualisierung von Windparks mittels Augmented Reality

Auralization and Visualization of Wind Farms using Augmented Reality

Masterarbeit
Software Systems Engineering

Juli 2021

Vorgelegt von Presented by	Viktor Panichkin Kreuzstrasse 17 40210 Düsseldorf Matrikelnummer: 407511 viktor.panichkin@rwth-aachen.de
Erstprüfer First examiner	Prof. Dr. rer. nat. Erika Ábrahám Lehr- und Forschungsgebiet: Theorie der hybriden Systeme RWTH Aachen University
Zweitprüfer Second examiner	Prof. Dr. rer. nat. Thomas Noll Lehr- und Forschungsgebiet: Software Modellierung und Verifikation RWTH Aachen University
Betreuer Supervisor	Dr. rer. nat. Pascal Richter Lehr- und Forschungsgebiet: Theorie der hybriden Systeme RWTH Aachen University

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich diese Masterarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Stellen meiner Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen sind, habe ich in jedem Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht. Dasselbe gilt sinngemäß für Tabellen und Abbildungen. Diese Arbeit hat in dieser oder einer ähnlichen Form noch nicht im Rahmen einer anderen Prüfung vorgelegen.

Aachen, im Juli 2021

Viktor Panichkin

Contents

Zentrales Prüfungsamt/Central Examination Office

Eidesstattliche Versicherung
Statutory Declaration in Lieu of an Oath

Panichkin Viktor | 407511

Name, Vorname/Last Name, First Name | Matrikelnummer (freiwillige Angabe) Matriculation No. (optional)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel

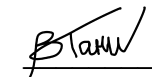
I hereby declare in lieu of an oath that I have completed the present paper/Bachelor thesis/Master thesis* entitled

Auralisierung und Visualisierung von Windparks mittels Augmented Reality /

Auralization and Visualization of Wind Farms using Augmented Reality

selbstständig und ohne unzulässige fremde Hilfe (insbes. akademisches Ghostwriting) erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

independently and without illegitimate assistance from third parties (such as academic ghostwriters). I have used no other than the specified sources and aids. In case that the thesis is additionally submitted in an electronic format, I declare that the written and electronic versions are fully identical. The thesis has not been submitted to any examination body in this, or similar, form.

Duesseldorf, 06.07.2021 | 

Ort, Datum/City, Date | Unterschrift/Signature

*Nichtzutreffendes bitte streichen

*Please delete as appropriate

Belehrung:
Official Notification:

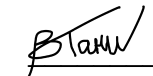
§ 156 StGB: Falsche Versicherung an Eides Statt
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

Para. 156 StGB (German Criminal Code): False Statutory Declarations
Whoever before a public authority competent to administer statutory declarations falsely makes such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment not exceeding three years or a fine.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Para. 161 StGB (German Criminal Code): False Statutory Declarations Due to Negligence
(1) If a person commits one of the offences listed in sections 154 through 156 negligently the penalty shall be imprisonment not exceeding one year or a fine.
(2) The offender shall be exempt from liability if he or she corrects their false testimony in time. The provisions of section 158 (2) and (3) shall apply accordingly.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:
I have read and understood the above official notification:

Duesseldorf, 06.07.2021 | 

Ort, Datum/City, Date | Unterschrift/Signature

1 Introduction

According to the Federal Ministry for Economics Affairs and Energy, the share of renewable resources has to enlarge up to 65 percent by 2030. However, the research conducted in 2018 postulates that the overall share of renewable resources in Germany slightly exceeds 25 percent [31]. Therefore, the 40 percent difference must be covered in the next 9 years.

Looking back in the last 10 years, the investments in renewable resources were only decreasing [32]. One of the reasons for that is the lack of interest among the population. In 2020, more than 57 percent of the respondents stated that the renewable energy topic is not interesting for them at all [33]. Hence, there exists a problem of residents' involvement in the process of alternating the renewable energy sources.

The main goal of this thesis is a transparent provision of information related to wind energy among the population. To increase the awareness of the planned wind farms, a citizen should have the information in a clear and transparent manner in one place. Besides the pros and cons, it should show the expected shadow cast, noise propagation, CO2 emission, and how the landscape will be changed visually after the wind farm would be built.

Therefore, the following plan has been defined. The starting point of the work is a spatial visualization of the potential wind farm by using augmented reality. Moreover, it is important to include noise auralization to imitate a real environment. The input data for the augmentation and auralization should be provided by the wind simulation model. The simulation model has to include the areas where potentially the new wind farms can be built. Inside these areas, we need to select the optimal position for placing a set of turbines. Afterward, the noise influence on the environment should be calculated.

The main usage scenario is: a user starts an application and points a mobile device towards the horizon. If a wind farm is planned in the direction of the pointing camera, the wind turbines are shown. To achieve this effect, the user's geographical position should be taken and synchronized with the motion sensor of a device. Once the motion tracking has been defined using six axes(three for rotation accompanying three for position), the only data is left is the planned wind turbines' GPS positions. Additionally, sound information can be provided. Utilizing the simulated noise propagation data, the sound can be added to the scene creating in order to provide a more natural user experience. As we build the spatial scene, it would be better to orient the sound with rotation. The solution should support mobile devices as the rotation movement is hardly applicable holding the large pc or even notebook. To cover as many users as possible, the application should support multiple platforms. However, the native development under two major platforms (iOS and Android) leads to the problem of maintaining two completely different codebases. Therefore, the web-based solution is an optimal way to satisfy the users without concentrating on the mobile device they are using.

1.1 Related work

Aiming to give an understanding of the augmented reality implementation, in this section we would like to describe the previous research findings and current state of the art.

One of the first successful researches was conducted in a Columbia University [12]. The scientist created a prototype of the campus touring system named as mobile augmented reality system (MARS). However, the mobility was reduced by that time as the user had to carry a large backpack with all the hardware and wore a head-tracked half-transparent display. In a static position, the result was comprehensive, though the issues came once the user started moving causing a false user pose detection and therefore wrong scene representation.

One idea for outdoor augmented reality was presented in 1999. Researchers suggested using the geographic information system (GIS) in order to improve the tracking of the user. The paper describes the improvements within the Ubicom project [27]. As the built-in GPS module can be inaccurate within two to ten meters, the determination of the user position can be improved using the video stream. The visual information taken from the camera can be combined with the gis-data by matching the respective lines, hence the improved position can be retrieved. However, there are several issues that the visual analysis has. Firstly, objects with a bizarre form such as bush are hardly recognizable. Secondly, the environmental condition can cause impediments, such as illumination change. Lastly, frequent recalculation of dynamic scenes might affect the real-time performance and cause glitching [16].

The theoretical foundations were applied and extended within the hybrid approach that allows a user to achieve near real-time experience [30]. The paper introduces the first practical implementation towards detecting visual landmarks. The main drawback of this model is that it can extract the corners of the object under various angles or lighting conditions.

As the technology was developing, the mobile device got more powerful and, hence, able to run augmented reality applications in place. First applications on the mobile devices were adding floating markers that the user could tap in order to get additional information. Nevertheless, the complex scene with multiple objects is still a problem of mobile AR. The CityViewAR project was trying to create a representation of the city. As the number of buildings grew up, performance issues appeared. One of the enhancements was to reduce the number of buildings and depict only one that was placed near the observer [23].

In the last decade, a lot of new outdoor AR apps were presented as the development process was significantly eased. However, they mainly improve the visual hints and 3d models instead of improving the user's tracking and proper overlay placing [6].

With respect to the auralization, the first substantial work was presented by Ville Pulkki in 1997 [29]. In the paper, he introduces the two important localization techniques that humans use to localize the sound, namely interaural time difference and interaural value difference. Moreover, the author introduced the most popular panning method based on two speakers located symmetrically to the median plane.

Considering browser-related solutions, the Web Audio API is one of the core technology, that allows getting a 3D perception using a simple browser. The possible use-case of this application layer is presented by Pike and Taylor [28]. They presented a PannerNode, that will be further used in this work. Additionally, they mentioned the important aspect, namely that the API is standardized. Hence, all modern browsers tend to conform to it removing the need to support and third-party library.

1.2 Outline of this work

The work is divided into several sections. In Section 1, we introduce the data obtained from the wind farm simulation model. This data is going to be used as an input for sound and augmented reality. Section 2 represents how the data is visualized with augmented reality. In Section 3, the further auralization of a wind farm is presented. Additionally, the application required a back end and a database that handle the data in the whole application. This will be described in Section 4. Finally, the project's deployment procedure using automatic deployment tools is shown. Every part of the project is going to be presented in the respective section.

2 Windfarm simulation

The electrical power can be generated by wind turbines. In order to produce enough electricity, wind turbines are usually aggregated into wind farms. This approach allows reducing the costs of installation and further support. However, while the costs are decreasing, the proximity of turbines might create a wake effect and cause an efficiency reduction. The wake effect is a process of creating turbulence that propagates the downwind used by the wind turbine. As a result, the wind speed is reduced. Consequentially, it decreases the power of the surrounding wind turbines. Therefore, the input data should be prepared beforehand, considering this notion. For that, we are going to start from the sequence diagram shown in Image-1. The diagram represents the data flow from the raw files to the final set of data after being processed with a bunch of scripts. The resulting files will be further used for the auralization and visualization. In the following subsection, a brief overview of the input data will be presented.

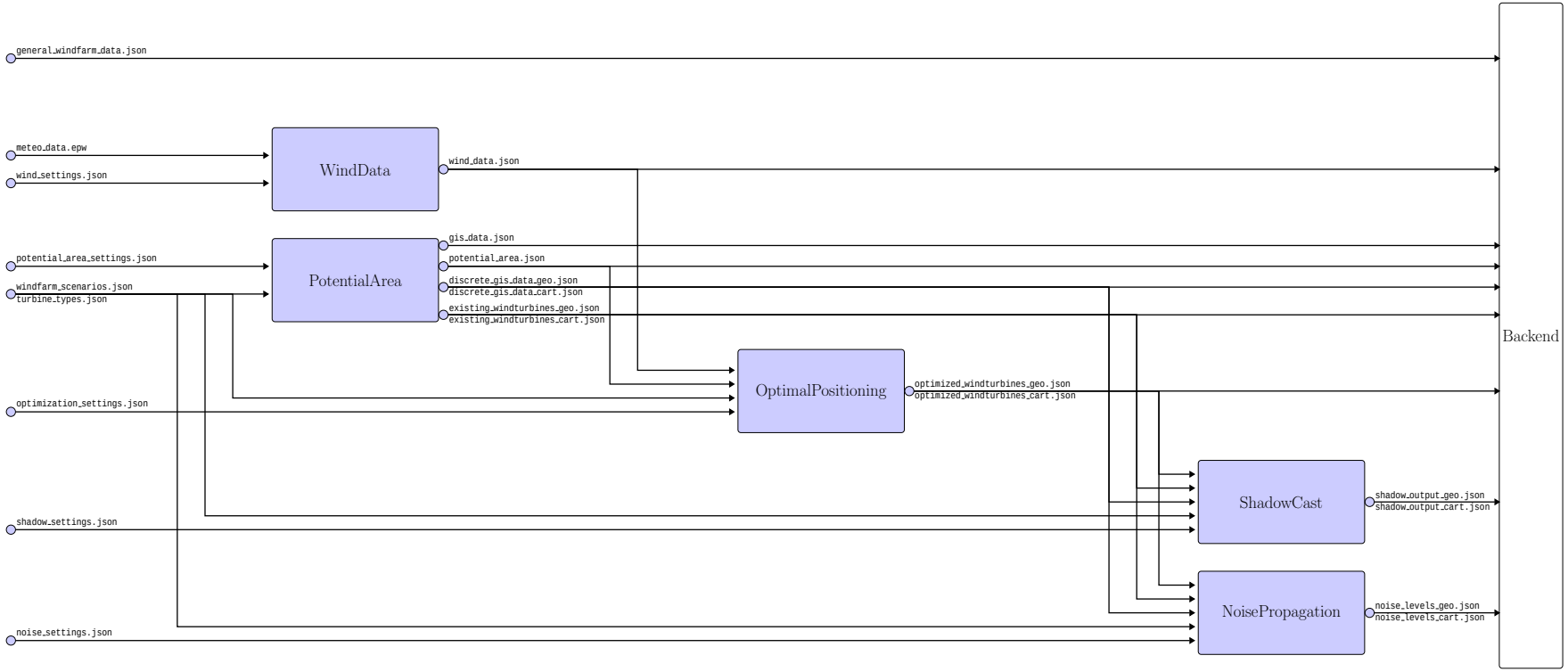


Figure 1: Wind farm simulation data-flow diagram

2.1 Wind data

One of the first task of planning the wind farm is to check the wind conditions. The wind data script uses a *meteo_data.epw* and *wind_settings.json* as input files. Meteorological data is provided by EnergyPlus Weather (EPW) file. The script provides a *wind_data.json* file as an output that contains the wind direction property, probability for observing a specific wind speed and the main wind direction property. It will be further used for the Optimal Positioning job.

2.2 Potential area

At the same time, we can execute the script that finds the potential area where the wind turbines can be placed. As an input it is required to provide a set of files. The first is *turbine_types.json*, it stores parameters for all possible turbines, including height, rotor diameter, power, etc. Secondly, we create a *windfarm_scenarios.json* file, where a set of use cases is defined by setting turbine type and minimal distance to houses. Lastly, potential area setup is completed with a settings file. The law defines the minimal distances to buildings, airports and highways. Further we will name them artifacts. However, the minimal distance to the houses can vary from 500 to 1500 meters and usually negotiated with the residents. After the program completes its execution, four files are generated. *Gis_data.json* contains all objects within the predefined boundaries. This file can be used for the further virtual reality scene building and reduces the number of queries to the third-party cartographic service. Due to the wake effect, the *existing_windturbines_*.json* provides the coordinates of the wind turbines already placed in the current region. Star symbol in the name represents two types of files the algorithm generates. It is either named as cartographic (cart) or geographical (geo). First type of files defines its own coordinate system. This has been done to ease the iteration inside the algorithm. However, for the rendering task using the location-based augmented reality the real geographical coordinates should be given as an input. Therefore, two different versions of a file have been generated. Third file is a *potential_area.json*. For every scenario, we generate a set of data with the following properties: *coords* represents the potential places for placing a windturbine, *border_points* and *border_polygons* are coordinates of the potential area boundaries and is used for the visualization. Lastly, *zone_info* provides the correspondence of the chosen coordinate system with a geographical one. These output files are further used in finding the optimum and noise pollution calculation.

2.3 Optimal Positioning

As we got the potential area, it is possible to proceed with the finding of the optimal position for the future wind turbines. In order to mitigate the wake effect, the area around the wind turbine is restricted with an ellipse. The diameters across the main wind direction and the side axis can be set up in *optimization_settings.json* file. Additionally, the knowledge of existing wind turbines should be provided. As before,

we need some wind turbine-related parameters from the *turbine_types.json* file. The *wind_data.json* is going to be used if the algorithm finds several best placements. In this case, the best orientation is taken into account according to the main wind direction. The script execution leads to generation of a *optimized_windturbines_*.json* file where the geographical coordinates will be provided. As the optimal positions for the wind turbines have been found, it is possible to calculate specific characteristics, e.g noise propagation.

2.4 Noise Propagation

In spite of the fact that the wind farms are located at least 500 meters from the residents' houses, the noise generated by the wind turbines can reach houses. Therefore, the noise propagation model has been calculated. As an input, it is required to have topography information, existing and optimized wind turbines. The script can be modified with a *noise_settings.json*. Then the main parameter is border_lines. As it was not possible to calculate the noise in every point of space, the discretization procedure has been applied. Hence, the aforementioned border_lines define the array of values that represents the sound pressure on the border of each polygon. In-between of n and n+1 border_lines the noise level is assumed to be a constant. In the auralization section, we will use this information in order to find a sound in a specific coordinate.

3 Augmented Reality

In this project GPS sensor information is used to locate the user among 3 axes. Additionally, an accelerometer is accompanying a compass to register the rotation. All together the 6-axes model can be used to identify uniquely the user's position. Based on the azimuth and the current geographical position the augmented reality scene can be built within the specific region of interest (ROI).

As the web-based solution was selected, two candidates were considered, namely Argon.js and Ar.js. Argon.js library is a successor of the well-known Argon4 web browser which core feature is augmented reality. However, downloading an additional browser to run the application is similar to installing the native application from the platform store. Therefore, the authors decided to create a standalone JavaScript library that could be imported as any other third-party dependency and facilitate the users' life. According to the documentation, [4] the main goal of the library is to provide an abstraction that allows browsers to run AR scenes.

3.1 Location-based Augmented Reality Using Argon.js

Argon.js v1.4 is the last available version. It consists of three core modules. The first is a *reality augmenter* that is responsible for adding the 3D objects to the scene. The second is a *Reality view model*. It takes care of the handling camera and gets the real environment representation. The third component is a *reality manager*. Its main duty

is to combine an artificial object within the real surroundings and to present the AR result. This separation allows a quick switch between different modes. With respect to the graphical library, Argon.js can be utilized with A-Frame or Twine. For the following example, we chose a modern A-Frame framework that is built on top of a sophisticated Three.js library.

As the main goal of the project is to create a location-based augmented reality scene where the planned wind turbines are shown around the users, we initially took a look at how the basic AR scene can be created. In Listing-1, Argon.js substitutes the core A-Frame *a-scene* tag with a custom *ar-scene*. To add an entity to the scene the *ar-geopose* need to be used. A-Frame conforms by default to the HTML format, where linked tags are nested within one another. However, there are several issues with Argon.js. Firstly, the library was in active development during 2017 and currently is not maintained anymore. Secondly, there is no control over the user's position as it is handled under the hood. Lastly, even the simplest code snippet was not possible to execute as the browser could not identify the geolocation and stopped responding regardless of the platform.

```
1  <ar-scene>
2      <ar-geopose lla="51.22417 6.79118">
3          <a-box position="10 -20 8" color="#f5f5f5"></a-box>
4      </ar-geopose>
5  </ar-scene>
```

Listing 1: Location-based AR using Argon.js

3.2 Location-based Augmented Reality Using Ar.js

Taking into consideration the cons, another solution should be found. Further, we will consider Ar.js. Ar.js provides a possibility to create a location-based augmented experience. The usage principle is similar to Argon.js. The framework aggregates the position of the motion sensor and GPS sensor. After combining location coordinates and the camera direction, Ar.js renders a scene with the objects placed on the screen. Switching to the AR mode is easy. It is only required to add a *gps-camera* and a *rotation-reader* property as depicted in Line 4 of Listing 2.

```
1  <a-scene gps-camera-debug
2           vr-mode-ui="enabled: false"
3           embedded arjs="sourceType: webcam; debugUIEnabled: false
   ;">
4      <a-camera gps-camera rotation-reader></a-camera>
5  </a-scene>
```

Listing 2: Location-based AR using Ar.js

No calibration and synchronizing is required after that. Moreover, the prompt to get access from the user's device will be automatically presented. By default, the system will try to get the coordinates of the user and the current camera direction if such per-

missions were granted. Dynamic scenes can be also created by recalculating the user's position and setting the objects via JavaScript. While creating the application, an iPhone XR, iOS 14.1 was used. During testing, the object was always rendered on the user's camera position. After research of the problem, it was defined that the framework could not get the GPS information and as a result built the scene falsely. Careful code revision and debugging did not help to solve this issue. It was decided to create a workaround: for the <a-camera> we can use *simulateLongitude*, *simulateLatitude* and *simulateAltitude* parameters. This approach is usually used for debugging purposes, but, meanwhile, can be used for production-based code. As 95 percent of the mobile browser supports Navigator API[1], the next approach has been chosen: import the geolocation module from the Navigator API, get the device position, and simulate the position for the camera by setting the respective parameters. From now, Ar.js serves only one purpose - a view library that builds a scene. The Navigator API watches the GPS position. Once it changes, an *onSuccessWatchPosition* callback is being triggered. Then it calls a function that is responsible for a scene rerendering based on the new coordinates. If the position cannot be identified, *onFailWatchPosition* callback will be called. The third argument can provide any additional configuration that would be used further.

```
1 function registerUserLocationWatcher(onUserPositionChanges) {
2
3     function onFailWatchPosition(error) {
4       console.log("Position cannot be obtained:", error);
5     }
6
7     function onSuccessWatchPosition(position) {
8         onUserPositionChanges(position);
9     }
10
11    return navigator.geolocation.watchPosition(
   onSuccessWatchPosition, onFailWatchPosition, configuration);}
```

Listing 3: Defining user's position using Navigator API

While getting the GPS position, the mobile devices usually experience an error from two to ten meters. It causes flickering of the augmented reality objects on the scene that leads to a bad user experience. However, there is no way to improve the precision. Enabling of the property *enableHighAccuracy* didn't improve the overall quality. Hence, the next approach has been designed.

3.3 Scene Stabilization Algorithm

As we build an outdoor augmented experience, where the measured distances have the value starting from five hundred meters, we can mitigate the GPS error that is, in comparison, significantly smaller. For that, we apply the following discretization algorithm. We store the current user's position as depicted with a red dot in Figure-2. At the same time, the stable zone is defined with a circle. The stable zone radius (SR)

is a configurable property and in our example equals 15 meters. As long as users locate themselves inside a circle, we assume that the position is fixed and the scene stays the same. Once the users move in the direction of the arrow and escape the stable zone, the scene is recalculated. The new user's position is stored and the new stable zone is defined. In the picture, it is represented with a green circle with a new user's position in the center. This simple algorithm solves the twinkling problem and at the same time improves the performance as the augmented reality scene is calculated only once.

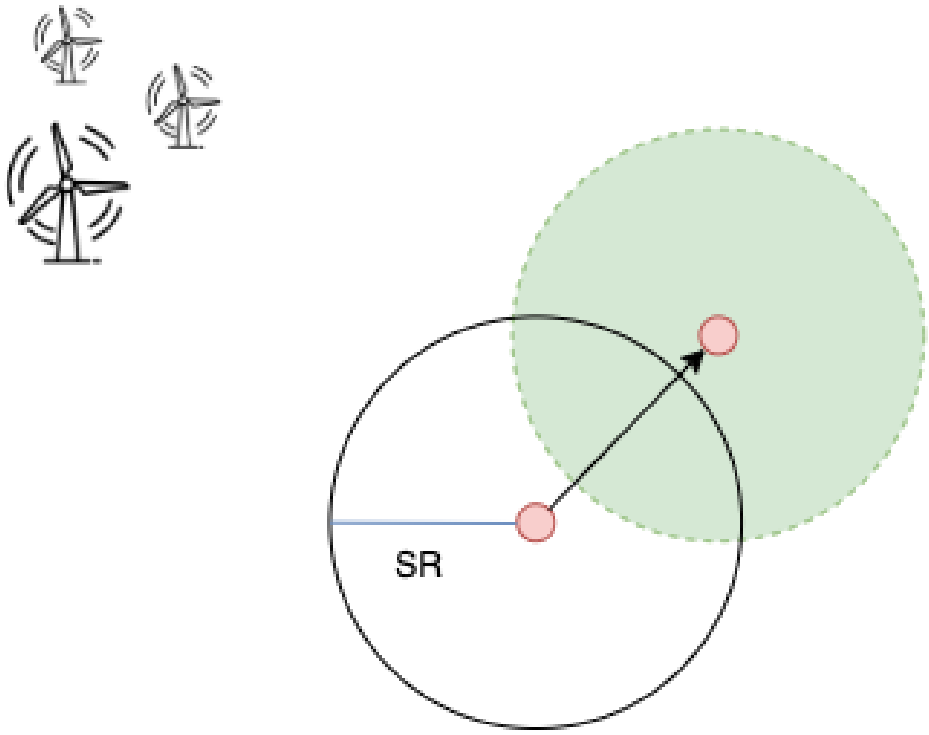


Figure 2: Applying discretization to the user's position

```
1 function onSuccessWatchPosition(position) {
2   if (!currentUserPosition.latitude || isUserOutOfZone(position.
   coords)) {
3     currentUserPosition.longitude = position.coords.longitude;
4     currentUserPosition.latitude = position.coords.latitude;
5     onUserPositionChanges(currentUserPosition);
6   }
7 }
8
9 function isUserOutOfZone(newUserCoordinates) {
10   return distanceBetweenTwoCoors(newUserCoordinates,
   currentUserPosition) > stableRadius;
11 }
```

Listing 4: Tracking if a user escapes the stable radius (SR)

With the respect to the codebase, an additional function *isUserOutOfZone* was implemented. It calculates the difference in meters between the previous and current user positions. Meanwhile, the *onSuccessWatchPosition* function was updated. It checks whether the user leaves the zone once the new coordinates are obtained and execute scene update as before. Therefore, the updated code can be found in Listing-4.

3.4 Wind turbine model rendering

To represent a wind turbine, the glTF model has been used. glTF stands for the Graphics Language Transmission Format [22]. The format was proposed by the Khronos Group in 2015. In six years, it became a standard form of sharing the 3D models and scenes. However, the provided wind turbine model was only static that does not look realistic. The task was to create a possibility to easily integrate and set up a model within the augmented scene. For that, it was decided to create a custom component using the A-Frame API.

The API should allow to set up the height of the tower, length of the rotor blades, and the rotation speed. These three arguments are identifying the component schema depicted in Listing-5. The initialization starts from the model loading. Once the model is ready, it is split into parts by name, namely tower and blade parts. Each setup* function is responsible for customizing the specific part of the model based on the input parameters.

The component utilizes *tick* method that is called on every scene render. This is a useful point where the animation can be handled. The *rotateBlades* method checks whether the model is initialized and rotates each blade in Z-plane with a specific parameter. This parameter corresponds to the speed. Finally, the component can be called everywhere by calling a one-line function: *<a-entity gltf-model=`<path_to_model_ le>` windturbine=`speed: 0.05, heightScale: 1.5, rotorScale: 2.5`></a-entity>*. This finalizes the augmentation part of this work.

```
1   export default function () {
2         AFRAME.registerComponent('windturbine', {
3           schema: {
4             speed: { type: 'number', default: 0.02 },
5             heightScale: { type: 'number', default: 1 },
6             rotorScale: { type: 'number', default: 1 },
7           },
8           init() {
9             this.el.addEventListener('model-loaded', () => {
10              const windTurbineMesh = this.el.getObject3D('mesh');
11              const towerMesh = windTurbineMesh.children.find(
    element => element.name === 'tower');
12              this.setupTower(towerMesh);
13              windTurbineMesh.children.forEach(element => {
14                this.setupRotorHeight(element);
15                this.setupBlades(element);
16              });
17              this.el.isWindModelLoaded = true;
18            });
19          },
20          setupBlades(element) {
21              // blade setup
22          },
23          setupTower(element) {
24              // tower setup
25          },
26          setupRotorHeight(element) {
27             // rotor setup
28          },
29          rotateBlades() {
30            if (!this.el.isWindModelLoaded || !this.data.
    _windTurbineBlades) return;
31            this.data._windTurbineBlades.forEach(blade => {
32              blade.rotation.z -= this.data.speed;
33            });
34          },
35          tick() {
36            this.rotateBlades();
37          }
38        });
39      }
```

Listing 5: Custom A-Frame windturbine component

4 Auralization

To provide a natural user experience, it was decided to append the wind turbine sound to the scene. According to the documentation [3], Apple forbade the autoplay and preload of the audio content. The cause was that the users might get charged for the data transfer they did not request. However, later it was also defined that the audio

and video HTML5 tags cannot also handle these two actions on their own. Therefore, the only way to play audio is to handle the user interaction. Once the user clicked the button, the audio is allowed to be played. This browser-specific behavior cannot be mitigated. Hence it was decided to implement an extra button that the user can use in order to initiate the sound on the scene.

While researching, it was defined two major problems that intervene in the scene auralization. The first problem is related to the 3D sound direction. The stereo sound does not represent where the turbines are placed. A user is only hearing a constant noise despite their rotation. Hence, it would be good to reach an effect similar to the identification of the direction from which the ambulance is coming using sound.

The second problem is defining the sound level in the specific coordinate where the user is currently located. However, due to the continuity of space, we cannot store the data for each geographical coordinate. Every problem is going to be solved and the solution will be presented in the following subsections.

4.1 Adding sound spatial effect

In order to create a spatial stereo sound, the problem was further decomposed into several sub-problems. The first task was to identify the user's rotation. Secondly, we are required to find the difference between the user's viewpoint and the sound emitter. Lastly, the sound should be played spatially.

Starting from the first problem, we will utilize the A-Frame API to track the users' rotation. A-frame provides a *rotation-reader* component. Every time the user rotates the camera, the tick function is being triggered. However, we don't need to track the rotation of each millisecond as it might drain faster the battery on the mobile devices. For that, we wrap our functionality into utility *throttle* function that will call the callback function not more often than 50 milliseconds. The Listing-6 shows how the registration has been accomplished.

```
1  AFRAME.registerComponent("rotation-reader", {
2      init: function () {
3        this.throttledFunction = AFRAME.utils.throttle(
   updateSoundPosition.bind(this, sound), 50, this);
4      },
5      tick: function () {
6        this.throttledFunction();
7      }
8  });
```

Listing 6: Tracking user's rotation using A-Frame component

Moving to the second sub-task, it is required to obtain the difference between the camera and the sound emitter. To simplify an example, we assume that only one sound emitter exists. A-Frame represents a rotation with a help of a quaternion notion. The quaternion is a four-dimensional vector, where the first three elements define a classical vector in a 3D space, while the fourth component represents the rotation around the

3D vector itself. The difference task is reduced to the task of finding an angle between two quaternions. To work with this abstraction, we need to use the core graphical library Three.js which forms the basis of A-Frame. The *Three.angleTo* method accepts two quaternions as an input and returns the difference in radians. This naive solution works well with one exception. The library defines an angle between quaternions as the smallest angle and measures it from 0 to . This leads to the problem that we cannot detect what channel will be louder, either left or right.

As there is no way to get the angle between 0 to 2 , where one half relates to the left channel and another half to the right channel, a new solution was proposed. The problem exists due to the fact, that quaternions have no information about the coordinate system where they are placed. While taking the difference, we perform only the mathematical operation. However, as we know the orientation of the coordinate system within the A-Frame scene, we can define the point of view and then utilize this information to find the proper rotation direction.

The algorithm is shown in Listing-7.Firstly, two vectors are taken along the vertical axis, namely the Y-axis from the A-frame coordinate system. Secondly, we multiply the quaternions by these vectors respectively to imitate the rotation. Then we project the resulting vectors to the horizontal X-Z plane. Lastly, we find the difference between two angles in a 2D space.

```
1   function getRotationQuaternion(camera, windMill) {
2   const cameraQuaternion = new THREE.Quaternion();
3   camera.object3D.getWorldQuaternion(cameraQuaternion);
4   const entityQuaternion = new THREE.Quaternion();
5   windMill.object3D.getWorldQuaternion(entityQuaternion);
6
7   const axis1 = new THREE.Vector3(0, 1, 0);
8   const axis2 = new THREE.Vector3(0, 1, 0);
9   axis1.applyQuaternion(entityQuaternion);
10  axis2.applyQuaternion(cameraQuaternion);
11
12  const angleA = Math.atan2(axis1.x, axis1.z);
13  const angleB = Math.atan2(axis2.x, axis2.z);
14  return {
15    angleBetween: THREE.Math.radToDeg(entityQuaternion.angleTo(
    cameraQuaternion)),
16    louderEar: THREE.Math.radToDeg(angleA - angleB) > 0 ? "left" :
    "right"
17  }}
```

Listing 7: Defining the direction of the noise

The resulting angle can now be measured in a range from - to . Negative values represent that the left channel should be louder, while positive value correlates with the right channel. As a result, we get the difference between two objects, namely the camera and the wind turbine, with the channel information that shows which ear will be louder. This solves the problem of identifying the sound direction and completes the second sub-task.

Initially, the third sub-task seemed to be trivial. It was decided to utilize the build-in A-frame playback component, namely *a-sound* [8]. A-sound component has a lot of parameters. It can handle the sound file, loop it, start the playback automatically, and many more. Though, the position parameter defined in the documentation can be set only by providing values in the scene coordinates system. As the location-based augmented reality utilized the geographical coordinate system, it was not possible to set the a-sound tag in the right position. Additionally, the sound component does not provide panning control.

In order to solve the last problem, we will take a look into a standalone sound library Howler.JS [21]. Howler.js is a javascript audio library that wraps the Web Audio API and provides a convenient API. Web Audio API is a standard that is currently under development [34]. Therefore the browsers are constantly required to update themselves in order to conform with the latest changes. Howler.js checks whether the functionality is available. In case it is not, the old HTML5 Audio API is utilized. With regard to our task, we would like to utilize the panner API included in the Howler.js library.

To create a simple sound object we need to get an instance of the Howler class as shown in Listing-8. The *Howl* class contains a settings object as the first argument, for instance, to set a source file. The library utilizes an event-based approach. The developer can implement a callback function that will be called, once the event is triggered. In our example, we subscribe to the *onplay* event in Line 4. To use a panner, panning attributes should be set up. Among them, *refDistance* introduces a value from which the volume will be decreased while the user is moving. The sound will be decreased linearly as it is stated in the *distanceModel* parameter. Afterward, we trigger the play method in order to start playing. It is a mandatory step as the majority of mobile browsers require manual interaction from the user, e.g button press, to start playing. To change the sound from the left to the right channel, we use a method *stereo*. The first argument is a value from the range [-1, 1], where edge values represent the 100 percent of sound to the left and right channel respectively.

```
1  const sound = new Howl({
2    src: "assets/windturbine.mp3",
3  });
4  sound.once("play", function () {
5      sound.pannerAttr({
6          refDistance: 0.75,
7          distanceModel: "linear"
8        });
9  });
10 sound.play();
11 sound.stereo(0.5);
```

Listing 8: Panning sound using Howler.js

However, the presented code snippet does not work properly in a Safari browser, where tests were conducted. The investigation leads to the following problem. Safari has no support of the *StereoPannerNode* [24] and fallback to the non-stereo regular

webkitAudioPannerNode that cannot create a stereo panning effect.

4.2 Bypassing the StereoPannerNode

As we cannot utilize the *stereoPannerNode* on the iOS platform, another workaround has been created.

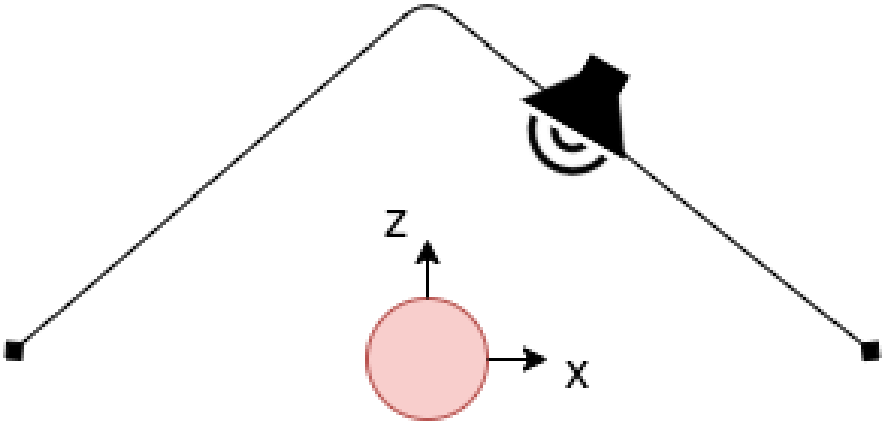


Figure 3: Imitation of StereoPannerNode by using PannerNode's position

The basic PannerNode has been implemented in Safari starting from iOS 6. The idea is simple: we can imitate the stereo panning by changing the position of the emitter in the space. Figure-3 shows the user as a red dot and the possible positioning of the speaker around the user. By changing the position, the panning effect can be achieved without manipulating the receiver. Hence, the algorithm checks whether the stereoPanner is available as depicted in Listing-9. If it is available, the panner value will be set. Otherwise, the basic PannerNode is created. The position is defined by the *setPosition* method. This method accepts 3 parameters: X, Y and Z. We set X and Z to *panWithEar* and *1-Math.abs(panWithEar)* respectively as they represent the horizontal plane.

```
1   function isWebApiPannerSupported(sound) {
2     return !!sound.audioContext.createStereoPanner;
3   }
4
5   function createPannerNode(sound) {
6     let pannerNode;
7     if (isWebApiPannerSupported(sound)) {
8       pannerNode = sound.audioContext.createStereoPanner();
9     } else {
10      pannerNode = sound.audioContext.createPanner();
11      pannerNode.panningModel = "equalpower";
12    }
13
14    sound.source.connect(pannerNode);
15    pannerNode.connect(sound.audioContext.destination);
16
17    return pannerNode;
18  }
```

Listing 9: Applying different panners based on browser-support

This concludes the three subproblems in order to play a stereo wind turbine sound within the AR scene.

4.3 Defining sound level

In this subsection, the problem of defining sound level will be presented. The sound level can be set based on the *noise_levels_geo.json* file. This file provides a set of coordinates that represents several concentric polygons. Within each polygon, the noise is assumed to be a constant. The values on the borders could be defined through border_lines settings.

As the polygons are nested, the initial task is the identification of the n and n+1 border line in-between the user is located. Using the nested property, the problem can be solved iteratively by checking whether a point is located within the polygon. As an example, the simplified problem is taken. In Image-4 two polygons depict some sound level. Within one of the polygon, a user is located and marked with a red dot.

The algorithm iterates from the smallest polygon up to the largest one. For this example, there are two iterations where the checking algorithm is applied. Once, the checking algorithm confirms that the point is located inside the polygon, the n+1 polygon is identified. Additionally, it is important to handle the edge case, when the user locates far out of the zone. This can be done easily if there is no polygon captured the point after the whole loop was executed.

4.4 Point inclusion algorithms

In the literature, there exist two algorithms that can identify whether the polygon includes the point, namely winding algorithm[35] and PNPoly[14]. Both algorithms

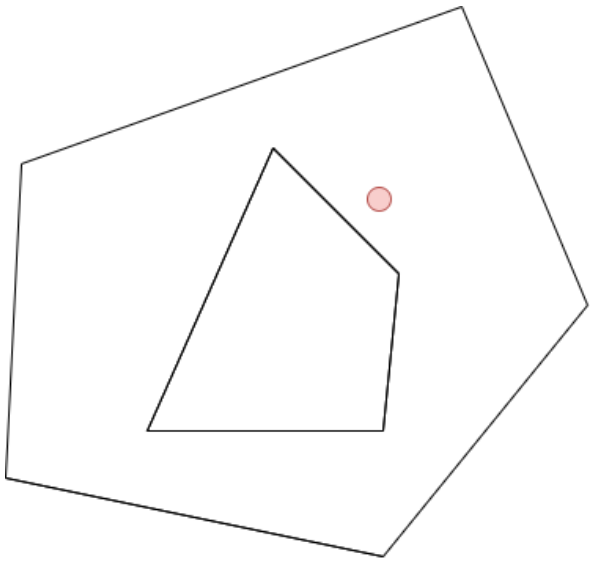


Figure 4: The exemplary problem of identifying the noise for a random user

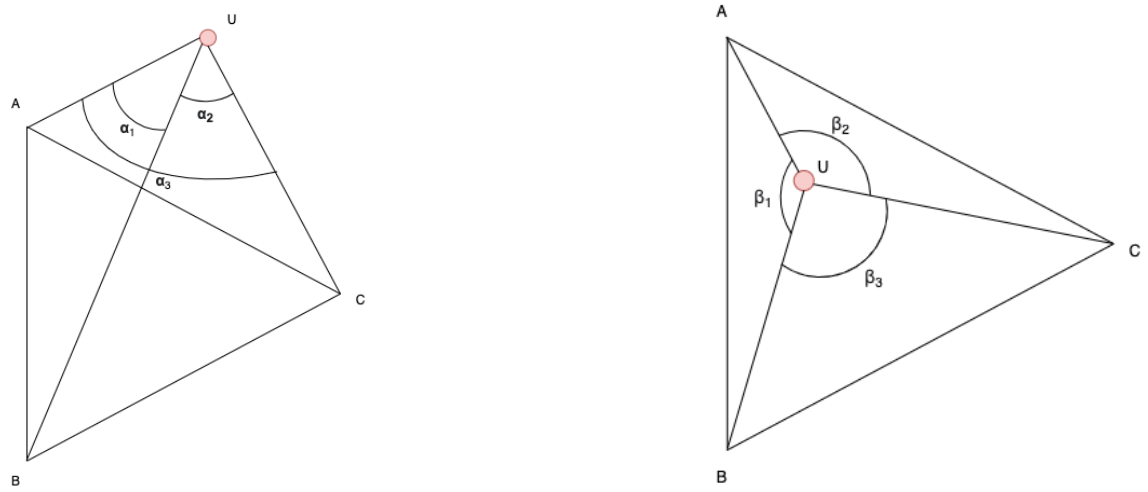
will be presented, implemented, and lastly compared. After that, the example will be continued to show the use of the algorithms

4.4.1 Winding number algorithm

A winding number algorithm was firstly mentioned in 1994 by Kevin Weiler. This algorithm belongs to the group of computational geometry algorithms and helps to identify whether the point is being captured by some polygon. There is a lot of practical application of the winding number algorithm, e.g line detector in computer vision. Initially, the algorithm has utilized the angles in order to calculate the winding number. The target point was connected with all the vertexes of the polygon, creating n line segments. These n segments create n angles. We sum up these n angles and check the value. If the value equals zero, the point is laying outside. In case the sum equals 2 , the triangle includes the points.

Two examples represent two possible cases. For the first case, there is a triangle with vertices A, B, and C. The point was placed outside as depicted in Image-5a. Three line segments that link the target point and the vertices are drawn, namely UA, UB, and UC. Consider the angles AUB, BUC and AUC marked as $_1$, $_2$ and $_3$ respectively. These angles sum up to zero as $_3$ contains both $_1$ and $_2$. As a result, it can be stated that the point is located outside of the triangle. Meanwhile, there is an opposite case shown in Image-5b. The U point is located inside the triangle ABC. The algorithm works in a similar way. There are three angles $_1$, $_2$ and $_3$. Their sum equals 2 . According to the algorithm, this means, the point is placed inside the triangle.

The winding number algorithm was not popular as the implementation included the arc-tangent calculation. That is a computationally intense task and, consequentially, it led to the fact that the algorithm was slower than pnPoly in the past.



(a) Point outside (b) Point inside

Figure 5: Angle-based winding number algorithm examples

However, one significant improvement was introduced in 2001 that removes the need of calculating the angles in order to find the winding number. The algorithm starts with the identification of the polygon wind edges. By using the horizontal ray that contains the target point, all edges that were crossed are called wind edges. The counter-clockwise path is defined on the polygon. Then, the algorithm iterates over the wind edges. Several conditions can change the winding number. If the edge is directed upwards and the point is on the left side of the edge, the winding number is increased by one. If the edge is directed downwards and the point is placed in the right half-plane of the edge, the winding number should be decreased by one. Every time the direction is checked, the orientation defines the edge itself. After all target edges are iterated, the point is considered inside the polygon if the sum does not equal zero. For explanation, the example in Image-6 is taken.

As before, there is a triangle ABC and the point U. The horizontal ray crosses edges AB and AC. As the edges are considered counter-clockwise, the target edges are AB and CA. The first step checks the edge AB, the direction is downwards and the point is located on the left. Hence, the winding number does not change. Next, the edge CA is taken. Its direction is upwards and the point is located on the right. Therefore, the winding number is incremented by one. As there are no more edges crossed by the ray and the winding number is not equal to zero, it implies that the point lies inside the triangle. The implementation of the improved algorithm is presented in Listing-10. Before the main function will be explained, the *ndCrossing* function needs to be introduced. As an input, it receives two points that are parts of the edge and the target point. After calculation, it returns the x coordinate of the point created by crossing the horizontal ray crossing the U point and the line segment of the polygon. By that,

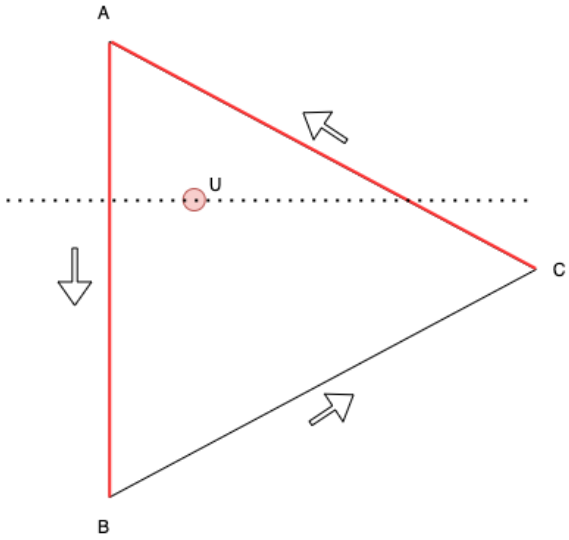


Figure 6: Ray-based winding number algorithm

we can check the downwards and upwards direction of the polygon.

```
1   function windingAlgorithm(target, polygon) {
2
3     function findCrossing(pointI, pointJ, target) {
4       return (pointJ.x - pointI.x) * (target.y - pointI.y) - (
    target.x - pointI.x) * (pointJ.y - pointI.y);
5     }
6
7     let windingNumber = 0;
8
9     for (let i = 0, j = polygon.length - 1; i < polygon.length; j =
     i, i += 1) {
10      const pointI = {x: polygon[i][0], y: polygon[i][1]};
11      const pointJ = {x: polygon[j][0], y: polygon[j][1]};
12      if (pointI.y <= target.y) {
13        if (pointJ.y > target.y && findCrossing(pointI, pointJ,
    target) > 0) {
14          windingNumber++;
15        }
16      } else if (pointJ.y <= target.y && findCrossing(pointI,
    pointJ, target) < 0) {
17        windingNumber--;
18      }
19    }
20    return !!windingNumber;
21  }
```

Listing 10: Ray-based winding number algorithm implementation

Then, there is an iteration over all edges. While doing that, the presented conditional statements are being checked in order to calculate the winding number. Finally, the

windingNumber is cast to Boolean and the result is returned.

4.4.2 PnPoly algorithm

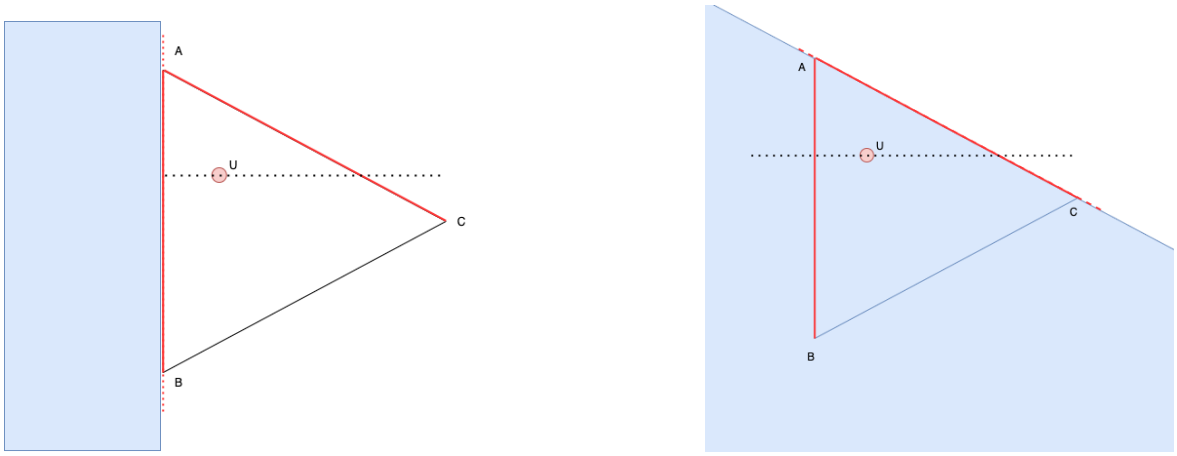
Another algorithm was created by W. Randolph Franklin and is known as pnPoly. The main task of it is the same. It allows identifying whether a specific point lies inside or outside of the polygon. The algorithm contains three steps. Initially, the horizontal ray cuts the plane through the target point. The total sum of edges being crossed before the ray reached the target point is calculated. In case the number is odd, then the point is inside the polygon, otherwise, it is outside. The Listing-11 shows the implementation of the algorithm.

```
1   function isCoordinateInPolygon(target, polygon) {
2
3     function isTargetVerticallyInBetween(pointI, pointJ, target) {
4       return ((pointI.y > target.y) !== (pointJ.y > target.y));
5     }
6
7     function isTargetInLeftHalfPlane(pointI, pointJ, target) {
8       return target.x < ((pointJ.x - pointI.x) * (target.y - pointI
    .y) / (pointJ.y - pointI.y) + pointI.x);
9     }
10
11    let isInside = false;
12    for (let i = 0, j = polygon.length - 1; i < polygon.length; j =
     i, i += 1) {
13      const pointI = {x: polygon[i][0], y: polygon[i][1]};
14      const pointJ = {x: polygon[j][0], y: polygon[j][1]};
15
16      const isIntersected = isTargetVerticallyInBetween(pointI,
    pointJ, target)
17        && isTargetInLeftHalfPlane(pointI, pointJ, target);
18
19      if (isIntersected) {
20        isInside = !isInside;
21      }
22    }
23    return isInside;
24  }
```

Listing 11: PnPoly algorithm implementation

The algorithm requires two additional functions, namely *isTargetVerticallyInBetween* and *isTargetInLeftHalfPlane*. The first function checks whether the vertical coordinate of the target point is located in-between two different points with coordinates *pointI.y* and *pointJ.y*. Meanwhile, the function does not check which point is located higher. The only goal is to identify the target edges of the polygon. The second function checks whether the point lies in the left half-plane. The function is similar to the *ndCrossing* while calculating the winding number. The algorithm iterates over all edges. Each edge is checked initially whether it is a target one. Once it is the target

edge, the left half-plane property is being checked. If that is the case, the value of the *isIntersected* variable will set to true. Lastly, we toggle the variable *isInside* in order to satisfy the odd-even constraint.



(a) Left half-plane for edge AB (b) Left half-plane for edge AC

Figure 7: Left half-plane identification for triangle edges

As an example, we take the same triangle as shown in Image-6. We have three edges in the order A, B, and C. The first iteration of the for loop takes the edge AB. Y-component of the A vertex is bigger than the Y-component of the target U point, whilst point B is vertically lower than U. This means that the edge AB is a target one. Secondly, we check the left half-plane created by the edge. As it is shown in Image-7a, the point is not included in the left half-plane of the edge AB, therefore the variable *inInside* is not inverted. The iteration continues with the edge AC. It is a target edge and at the same time, the left half-plane created by this edge covers the point. This follows that the variable *inInside* is toggled to true. The third edge CB is not considered as a target one. As a result, it will not influence on the final result. The for loop ends and algorithms finish with a return value true as the U point is enclosed within the triangle ABC.

4.4.3 Comparison of the Winding number and PnPoly algorithms

After presenting both algorithms, they are going to be compared. As the interface they are providing is the same, the *testBench* function wrapper has been created. Inside the Node.JS performance module [13] is used. This module was firstly integrated in 2017 as part of the standard Node.JS build. The main advantage is the support of high-resolution timestamps that can be utilized for performance testing. Some parts of the API will be explained with an example shown in Listing-12. The function can be initialized with the number of runs to avoid accidental outliers. Secondly, the test generates polygons with a random number of vertexes. JavaScript allows passing the function as an argument that is the third argument. To control the timing, the PerformanceObserver class is instantiated. It can subscribe to the events and run a

callback function once an event is being triggered. In this example, we would like to save the execution time after each run. All timing will be saved in the *timeResults* array for further processing. Meanwhile, the polygon generator is initialized. For every run, we generate a custom polygon and a random target point. Once the initialization is completed, the mark is placed in Line 13. Marks store the timestamp when they were called. After that the algorithm executes and the mark is set. Having two marks, the difference is measured and, subsequently, recorded to the timeResults array. To neglect the outliers, as a comparative metric a median was selected.

```
1   function testBench(runsCount, pointsCount, algorithm) {
2     const timeResults = [];
3     const observer = new PerformanceObserver(currentRun => {
4       timeResults.push(currentRun.getEntries()[0].duration);
5     });
6     observer.observe({entryTypes: ["measure"]});
7
8     const generator = new PolygonGenerator({numOfPoints:
    pointsCount, minCoordVal: 0, maxCoordVal: 20});
9
10    for (let i = 0; i < runsCount; i++) {
11      const polygon = generator.getPolygonCoord();
12      const target = generator.getRandomPoint();
13      performance.mark("singleRun:start");
14      algorithm(target, polygon);
15      performance.mark("singleRun:end");
16      performance.measure(`Run: ${i}`, "singleRun:start", "
    singleRun:end");
17    }
18    return calculateMedian(timeResults);
19  }
```

Listing 12: Testing script to compare Winding number and PnPoly algorithms

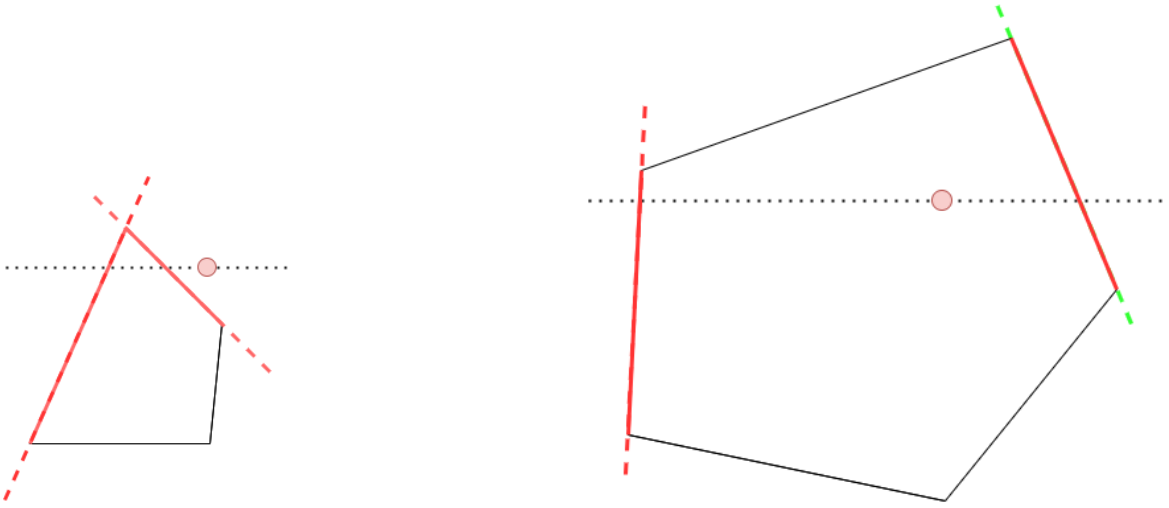
Having the testing function, the comparison can be conducted. The algorithms are tested while increasing the number of polygons' vertexes by a power of ten. For the polygons with 100000 vertexes, the execution time difference is less than 1 ms. According to the results presented in Table-1, the algorithms have the same cyclomatic complexity. The execution time is less than 1ms for both algorithms while checking point inclusion within polygons having 10000 vertexes.

	10	100	1000	10000	100000	1000000
Winding number	0.01052	0.01008	0.07538	0.81352	11.91704	190.32163
PnPoly	0.01058	0.01343	0.07708	0.82176	12.06002	191.432

Table 1: Comparison of Winding Number and PnPoly algorithms

As we have two similar algorithms, the application for the simplified problem depicted in Figure-4 will be shown. Consider the PnPoly algorithm for that. The first inner polygon contains two target edges that are crossed with a horizontal line. However, left half-planes, created with the extension of edges, do not cover the point.

Therefore, the variable *isInside* is toggled zero times and return false value. The point is not part of the polygon shown in Image-8a. The algorithm checks the next polygon as shown in Image-8b. As before, there are two target edges which vertices are placed in-between the user. The extension of the first edge again creates a left half-plane that does not cover the target red point. Though, the second target edge which extension is marked with green dashed lines, satisfies both conditions and flips the variable *isInside*. As there is no target edge anymore, the algorithm resolves with identified polygon where the user is located.



(a) First iteration (b) Second iteration

Figure 8: Application of PnPoly algorithm for simplified problem

This ends the subsection of finding the noise value inside the specific polygon of the user's position. However, after testing this approach in the real application, the a problem appeared. Despite the user moves around the scene, the noise stays constant within the same zone. Secondly, due to the exponential nature of the decibel scale, the user can get a false perception. Once they cross the boundary between the n and n+1 polygon, the decibel value can increase instantly and cause an amplitude jump many times. Due to the continuity of space, it is not possible to calculate the noise for every coordinate where the user potentially can be located. To solve the problem, the interpolation of the noise value between the zones should be implemented.

4.5 Sound interpolation for user's coordinate

The simplest solution that can be implemented is linear interpolation. As shown in Image-9, the radius vector is built from the center up to the largest polygon through the user's position U. A and B are found that are the crossing points of the radius vector, and the n and n+1 polygon that encloses the user's coordinate. After that, the factor value is calculated based on the value AU and AB. As the values in points

A and B are defined by polygons, the noise in point U can be interpolated using the noise difference between A and B multiplied by the factor value.

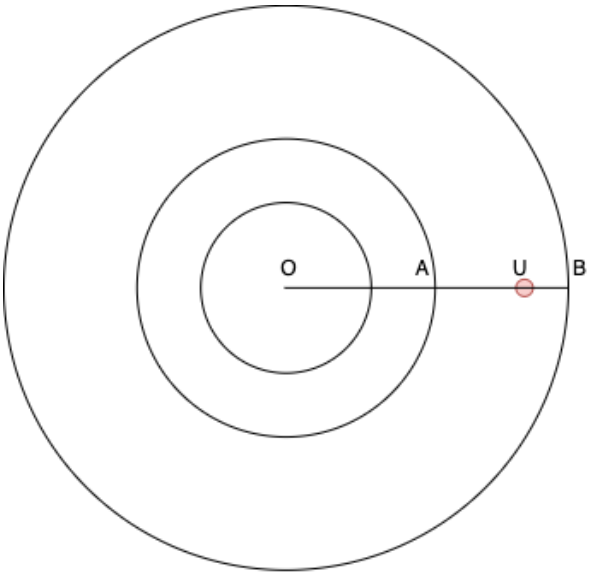


Figure 9: Linear interpolation of the noise

However, this algorithm has an important restriction, namely, the polygons must be smooth and, in the best case, create a set of concentric circles. The problem is that the radius vector is orthogonal towards the borderlines. However, The orthogonal projection from the target point to the polygon edge will not be the closest path and, consequentially will not allow the calculation of the interpolation property.

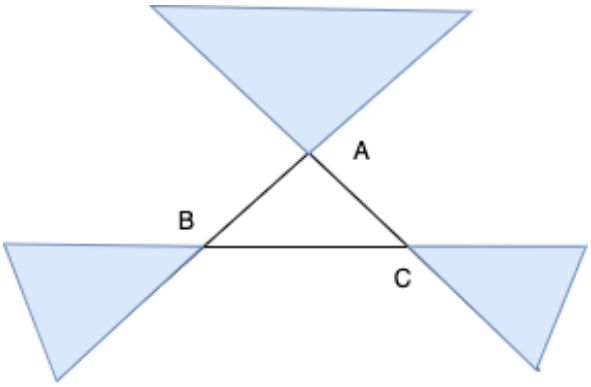


Figure 10: Edge case for orthogonal line interpolation

As an example an Image-10 is drawn. There is a triangle ABC that represents the borderlines of one polygon from the noise model. In case a user is located in one of the blue-filled areas, the orthogonal projection to the edge will not be the closest path. In fact, the line segment should be built to one of the closest vertexes. The localization of the blue area can be quite a hard task as all edges of each polygon should be considered. Hence, another algorithm instead of linear interpolation should be used.

4.6 Triangle-based interpolation

One of the possible ways to interpolate the noise value in a user's position can be a triangle-based interpolation. The algorithm can be divided into three major sub-tasks:

1. Triangulate a polygon
2. Localize the target triangle
3. Interpolate the point value using the vertices from the target triangle.

4.6.1 Polygon triangulation

Triangulation is the process of splitting a set of points on a set of triangles. Meanwhile, edges from these points are not crossing and the amount of the edges is maximal. In this paper, one specific triangulation sub-type will be used, namely Delaunay triangulation.

To name the triangulation as Delaunay triangulation it should conform to the important property. For every circle around a triangle created by triangulation, there are no points from the set inside the aforementioned circle. The implication that can get from this type of triangulation, that the angles for all triangles are maximized such that sliver triangles are avoided. To build a Delaunay triangulation, several concepts should be considered beforehand.

The first is checking whether a point is placed inside a circumcircle that is drawn around some triangle. One of the ways is a determinant calculation [15]. If there are three points A, B, C that are ordered in a counterclockwise direction, the determinant will be position only in a case if a target point D lies inside the circumcircle around the triangle ABC. This check can confirm or reject the main requirement of the Delaunay triangulation.

The second concept is a flipping algorithm [5]. Consider a polygon with four edges ABCD such that depicted in Image-11. There is two possible triangulation. The first can imply two triangles ABC and BCD with a common edge BC. The second one can be created by adding edge AD and, consequentially, divide the polygon into two triangles, namely ACD and ABD. As it can be seen, the first triangulation is a Delaunay triangulation as there are two circumcircles depicted in green around the triangles ABC and BCD that do not contain the fourth point. The circle around triangles ACD and ABD is shown in a red dotted line and implies that this triangulation does not conform to the Delaunay restriction. If initially the wrong triangulation has been selected, the edge inside the quadrangle can be flipped and as a result, satisfies the Delaunay triangulation requirement. Apart from the calculation of the determinant, another simpler checking property can be used. If the sum of the two opposite angles of a quadrangle is equal or less than 180°, the splitting edge between these angles will create a Delaunay triangulation.

The third concept that will be needed to create a Delaunay triangulation is a visible convex hull. For some point, the edge of the polygon is called visible, if the two line segments created from the point to the edge vertexes do not cross the polygon.

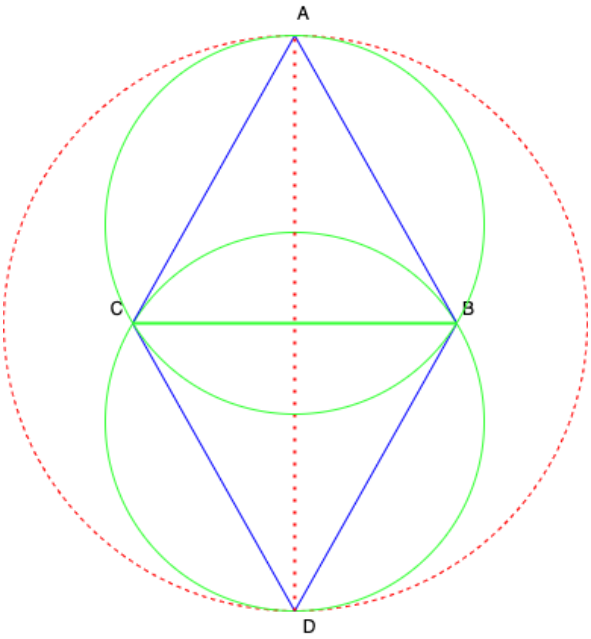


Figure 11: Flipping algorithm in a Delaunay triangulation

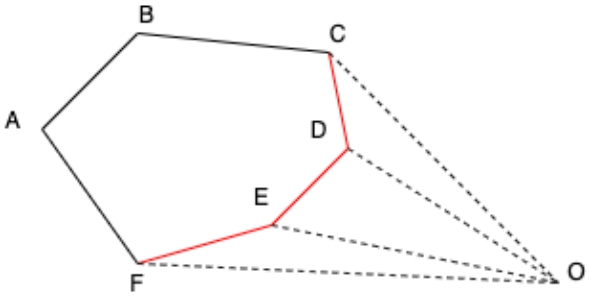


Figure 12: Visible convex hull in a Delaunay triangulation

Consider a polygon ABCDEF in the Image-12. Assuming there is a task of identifying the visible convex hull from point O. The edges CD, DE and EF are visible from point O as the line segments OC, OD, OE, and OF do not cross the polygon ABCDEF. However, there is a need to check for each point whether the line segment crosses the polygon. Therefore, some heuristics should be applied. We sort the points across some line, for instance, the y-axis. For the O point, the closest one is point D. The cross-product of two vectors is checked, namely OD and DC. Subsequently, the cross-product for the vectors OC and CB are checked. The sign of the cross-product is changed, which means that point C is the last visible vertex from point O. Similar iterative approach can be conducted towards the edge DE, EF, and FA. The sign change will indicate that the vertex F is the last visible vertex from O. Hence, the edges CDEF are considered as visible from point O.

Having three separate definitions, the sweep-line algorithm [2] of building the Delau-

nay triangulation can be presented. The algorithm was initially developed in 2005 by Borut Alik. The main concept is that the set of points is going to be sorted and then is processed step-by-step. Firstly, the points are sorted along some line, for instance, the y-axis. The first three dots are connected and hence, create the first triangle. For every further point, the algorithm finds the visible convex hull as were described in statement two. Based on every visible edge and the current point, new triangles are added to the triangulation. The next step should satisfy the Delaunay triangulation invariant. Therefore, the algorithm checks quadrangles that contain the newly created visible edge. Once the requirement is not satisfied, the visible edge changes according to the flipping algorithm. The process continues recursively for all subsequent quadrangles, as the change of the current edge might lead to the case that another triangle does not satisfy the Delaunay invariant anymore.

The complexity of the algorithm is quadratic as the recursion can require at step i at most i-1 recursive calls. As the triangulation will be built only once during the server startup, the real-time experience for a particular user will not suffer as the data will be prepared upfront.

To simplify the development, in the current project the Delaunay triangulation is calculated with a help of the Delaunator project [18]. In order to avoid intense computation on the end-user devices, it was decided to implement the functionality on the back end side with a further API provision.

```
1  async function getTriangulation() {
2     const noiseFile = await getFileCompletely(config.ar.
   noiseFileName);
3     const triangulations = {};
4     Object.keys(noiseFile).forEach(scenarioId => {
5       const {coords, noise} = getTriangulationForScenario(noiseFile
   [scenarioId]);
6       triangulations[scenarioId] = {
7         delaunay: new Delaunator(coords),
8         noise
9       };
10    });
11    return triangulations;
12  }
```

Listing 13: Precomputing of the Delaunay triangulation based on noise file

Before a server startup, the *getTriangulation* function is called. The function is shown in Listing-13. It fetches the *noiseFile* from the MongoDB database. For each scenarioId, the Delaunay triangulation is calculated. Lastly, it is stored in an object with a scenarioId as a key. It allows further to access the needed triangulation in a constant time when a user requests the interpolated noise value.

As an example, the problem depicted before is taken into account. The triangulation procedure resolves with a figure shown in Image-13. It contains two borderlines as before, the user is marked with a red dot. Additionally, the figure is split with the orange line segments that create a Delaunay triangulation.

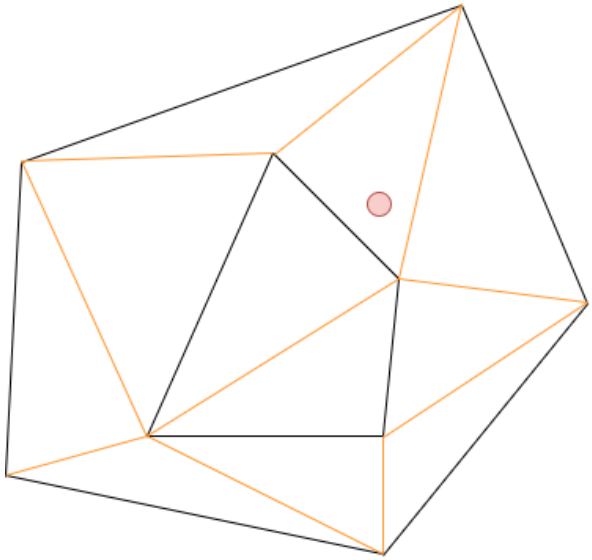


Figure 13: Example of the Delaunay triangulation

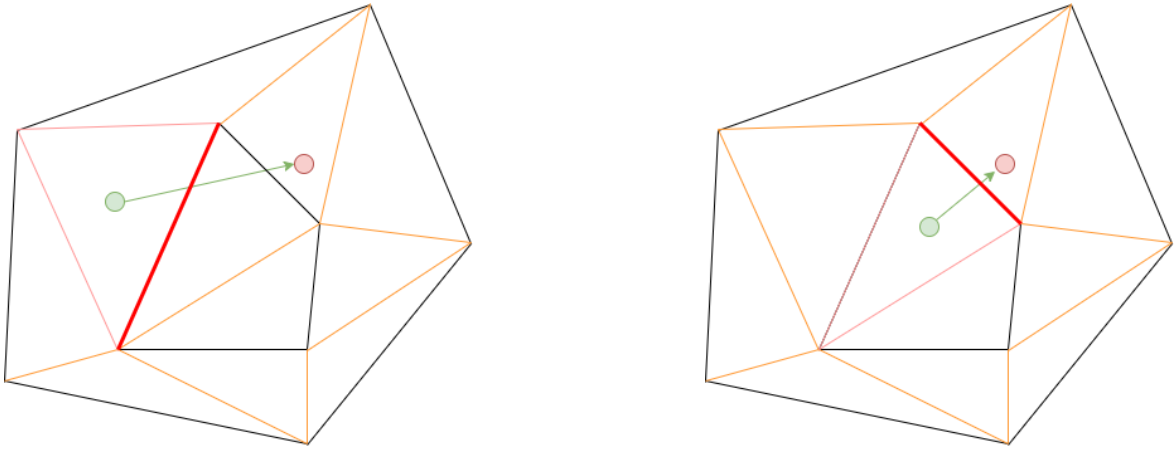
4.6.2 Localization of the target triangle

The next problem is the localization of the triangle where the user is located. The naive solution can be an iterative checking of every triangle. However, it is not an optimal way and in the worth case can be $O(n)$. This localization procedure needs to speed up. This is due to the fact, that a user's location is defined only during request and hence, it cannot be calculated during the server startup.

For that, the movement between i and i+1 triangle should conform to some heuristic. Hence, the walk and jump algorithm [7] was selected. Initially, the algorithm selects a random triangle to start with. It checks whether the target point lies within this triangle. If the answer is yes, the algorithm returns the current triangle. Otherwise, it takes any point within the current triangle, for instance, the triangle centroid. The vector starting from the centroid to the target point is built. In the next step, the intersection between the vector and the triangle edge is identified. Once the edge is identified, the algorithm can step into the next adjacent triangle which shares this common edge. The iteration continues until the triangle that contains the target point is found.

To show the process, the previous example is used again. Assuming that the algorithm selects the triangle on the left part of the Image-14a depicted with the red edges. It checks whether the user is placed inside the triangle. As it is not the case, the algorithm continues and adds a centroid marked with a green dot. The vector from the green to red mark is built. The bold red edge represents the first edge crossed by the vector within the current triangle. The adjacent triangle is being processed next. Again, Image-14b shows that it does not contain the point. Therefore the procedure of finding the adjacent triangle based on a vector from the centroid to the user's location is repeated. The adjacent triangle is taken and finally, it contains the user.

In this example, it is required to check three triangles in comparison to the total



(a) First iteration (b) Second iteration

Figure 14: Target triangle localization process

number of triangles that equals to twelve. The cyclomatic complexity of the algorithm is $O(n^{\frac{1}{4}})$.

The code in Listing-14 depicts the localization algorithm. The main part is an eternal while loop. However, it will stop in one of two cases, either the target triangle has been found or the *adjacentsTriangleEdges* variable will be negative. The second case means that the user is located out of the triangulated zone and the triangle covers the user's point cannot be found.

The edges are stored with a help of half edges concepts. Instead of having one shared edge with vertexes A and B, the data structure stores two half-edges AB and BA. As before, we take one triangle, check the point inclusion with *isInsideTriangle* function. Having centroid and user's location, we apply *getIntersectionEdge* function that returns the index in a range 0-2 of the edge that was crossed by the line segment. The adjacent edge is found with the help of half-edges. The initial vertex of the new triangle is set. The algorithm has one peculiarity, namely the case when the user is located exactly on the edge. This case leads to the infinite loop as the algorithm will step between two adjacent triangles that share the common edge. To avoid such behavior, the cache was introduced. It stores two previous iterations of the algorithm. If the vertexes of the triangles are repeated, the point is on the edge and the triangle is again returned.

```
1  function findTriangle(triangulation, targetPosition) {
2     let triangleStartingVertex = 0;
3     let iteration = 0;
4     const cachedVertexes = [];
5     while (true) {
6       const triangle = getTriangle(triangulation.delaunay,
    triangleStartingVertex);
7       if (isInsideTriangle(triangle.coords, targetPosition)) {
8         return triangle;
9       }
10      const trianglePoint = getTriangleCenter(triangle.coords);
11      const intersectedEdgeIndex = getIntersectionEdge(
    trianglePoint, targetPosition, triangle.coords);
12      const adjacentTriangleEdge = triangulation.delaunay.halfedges
    [getEdgesOfTriangle(triangleStartingVertex)[intersectedEdgeIndex
    ]];
13      if (adjacentTriangleEdge === -1) {
14        throw new Error("User is out of the noise polygons");
15      }
16      triangleStartingVertex = Math.floor(adjacentTriangleEdge / 3)
    ;
17
18      // checking case when a target point lays on the edge of
    triangle and we step between two adjacent triangles
19      if (cachedVertexes.includes(triangleStartingVertex)) {
20        return triangle;
21      }
22      cachedVertexes[iteration++ % 2] = triangleStartingVertex;
23    }
24  }
```

Listing 14: Implementation of the walk-and-jump algorithm

4.6.3 Interpolation within the target triangle

Once the target triangle has been found, the interpolation can be calculated. The simplest approach of calculating the value within the triangle can be an identification of the closest vertex.

The algorithm is called the nearest-neighbor method. Image-15 shows a sample triangle with the vertexes ABC. Zone A, B, and C within the triangle correspond to the closest vertexes respectively. However, the problem is still present. Considering the noise value of the A point is 35db and the B point is 40db. Once the user steps across the border, the value increases momentarily by 5db and, consequently, does not create a smooth auralization.

Instead of setting the value from one vertex, it would be good to have a contribution of all three vertexes of the target triangle towards the noise value in the user's coordinate. The closer a user to the vertex is, the larger the impact of the vertex should be. As a metric, the inverse distance can be used. Having the coordinates ABC, the distances D_a, D_b, D_c can be calculated applying the Pythagorean theorem as shown in

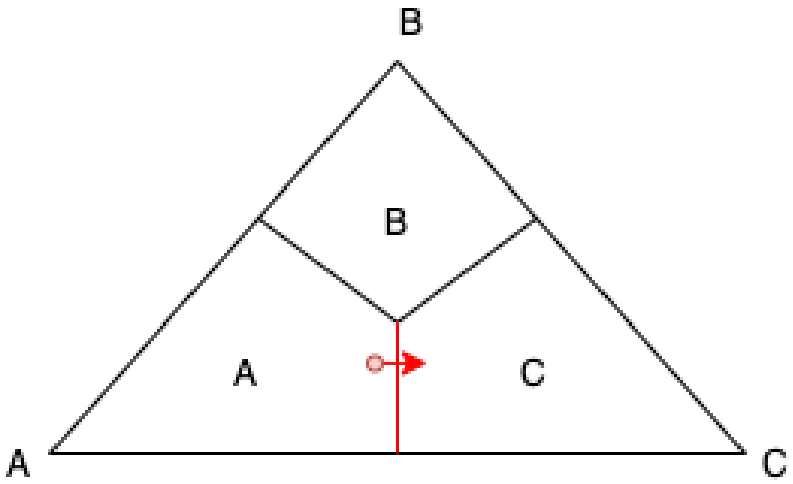


Figure 15: Nearest-neighbour interpolation in a triangle

the Equations-1, 2 and 3 respectively.

$$D_a = \sqrt{(Y_u \quad Y_a)^2 + (X_u \quad X_a)^2} \tag{1}$$

$$D_b = \sqrt{(Y_u \quad Y_b)^2 + (X_u \quad X_b)^2} \tag{2}$$

$$D_c = \sqrt{(Y_u \quad Y_c)^2 + (X_u \quad X_c)^2} \tag{3}$$

Having the distances calculated, the weights are just the inverted values of each distance from points A, B, and C. The final noise value is calculated as the weighted sum among 3 vertexes and can be found using the Equation-4

$$Noise_X = \frac{\frac{Noise_A}{D_a} + \frac{Noise_B}{D_b} + \frac{Noise_C}{D_c}}{\frac{1}{D_a} + \frac{1}{D_b} + \frac{1}{D_c}} \tag{4}$$

However, the linear interpolation works well if only the point is not located close to the edges. Consider the following edge case as shown in Image-16. There is a triangle ABC and the user is marked with a red dot and U point. The impact of the B point should be zero as the point lies exactly on the AC edge. However, the distance AU and AC will be bigger than distance B. Indeed, the weighting factor of the noise value from vertex B will be the largest among all vertexes that should not be the case. Therefore the linear approach does not suit a value interpolation within the obtuse triangle.

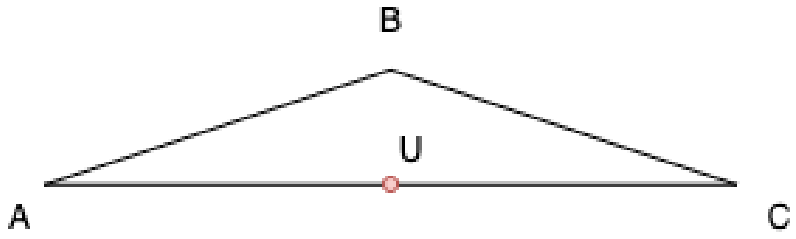


Figure 16: Linear interpolation failure while a user is on the edge

For the edge-case scenario, the shift towards the barycentric coordinates system [36] was made. The barycentric coordinates also utilize the concept of weighting the target point. The basis for it is a system of equations as shown in Equation-5. The first equation calculates the X component based on three vertices. The second equation computes the Y-component, while the third equation conforms to the fact, that the factors sum up to one. Having a system of three equations with three undefined variables, namely W_a, W_b and W_c the equation has only one solution. The coordinates X and Y have the same behavior as the noise value. Hence, once the factors are found, the sum of weighted components will be the interpolated noise value.

$$\begin{cases} X_u = W_a X_a + W_b X_b + W_c X_c \\ Y_u = W_a Y_a + W_b Y_b + W_c Y_c \\ W_a + W_b + W_c = 1 \end{cases} \tag{5}$$

The approach of the calculation of the weights can be found in Listing-15. The *approximateNoiseIn* function takes the vertex coordinates, user's coordinate, and the noise map as an input. The key-value map was taken in order to speed up the process of getting noise values for the specific vertex based on its index. The *calculateWeights* function is calculating the factors for the barycentric coordinates. Lastly, the value is obtained and will further be returned to the user.

```
1   function approximateNoiseIn(targetTriangleCoords,
    targetPositionCoords, noise) {
2     const [v1, v2, v3] = targetTriangleCoords.map(coordinate =>
    coordinate.join("_"));
3     const [w1, w2, w3] = calculateWeights(targetTriangleCoords,
    targetPositionCoords);
4     return w1 * parseInt(noise.get(v1)) + w2 * parseInt(noise.get(
    v2)) + w3 * parseInt(noise.get(v3));  }
5
6   function calculateWeights(targetTriangleCoords,
    targetPositionCoords) {
7     const [v1, v2, v3] = targetTriangleCoords;
8     const [x, y] = targetPositionCoords;
9     const w1 = ((v2[1] - v3[1]) * (x - v3[0]) + (v3[0] - v2[0]) * (
    y - v3[1])) /
10      ((v2[1] - v3[1]) * (v1[0] - v3[0]) + (v3[0] - v2[0]) * (v1[1]
     - v3[1]));
11    const w2 = ((v3[1] - v1[1]) * (x - v3[0]) + (v1[0] - v3[0]) * (
    y - v3[1])) /
12      ((v2[1] - v3[1]) * (v1[0] - v3[0]) + (v3[0] - v2[0]) * (v1[1]
     - v3[1]));
13    const w3 = 1 - w1 - w2;
14    return [w1, w2, w3]; }
```

Listing 15: Triangle-based interpolation using barycentric coordinates

Concerning the performance, the complete query from front end to the back end that calculates the noise in the user's coordinates takes around 17ms. In production, the triangulation operates around 1200 triangles. The localization process iterates over 10 triangles. The complete processing time is relatively small and hence, this approach can be used for the real-time server-based interpolation of the noise for the user. This ends the auralization section.

5 Back end

After creating visualization and auralization, the application should be served to the end-user. Moreover, the input files, that have been generated as part of the wind farm simulation, should also be accessible through the back end.

As we need a listening HTTP server, the *Node.JS* server was selected as the core back-end technology. For the routing purpose, the *express.js* library was taken. The data that we get from the offline computation is usually JSON files, therefore the document-oriented database was chosen, namely, MongoDB. By default, MongoDB is a schema-less database. However, it might lead to the problem of loading unknown data from itself during the runtime and cause a crash. To avoid such a behavior *mongoose* has been added. *Mongoose* is an object-document model (ODM), that helps to manage the data object. It introduces object schemas for validation and references between different collections. A brief overview of each back-end technology will be given further.

5.1 Express

To communicate with the frontend, the back end should support the HTTP protocol with a set of methods and handle the request according to the defined route paths. Moreover, it should serve static files, e.g images or, in our case, JSON files. All these requirements can be satisfied within the framework called Express.

Express is a javascript library created with a help of a large contributing community that runs an HTTP server. Apart from the requirements mentioned above, it can be used together with the HTML template builders. Template builder loads the data and places it inside the HTML markup such that the user gets the preprocessed view without additional requests to the back end. The framework can handle cookies, sessions, and headers that are usually used for user authorization. The main object is named as *app* and is available across the whole application.

Express uses the concept of middlewares. A middleware is a function that contains a *request*, *response* and *next* objects as input arguments. The request contains the request headers, query parameters, and the route that was asked by the user. In the response object, the output is expected to be provided. The *next* object serves for the sequence control purpose and allows the library to delegate the work for the next middleware [10]. Every route is registered in the router with the respective processing function (handler). When the user initiates a request, the framework iterates over all registered routes. Once it finds one, it dispatches the task to the handler. The main advantage of this middleware concept is the possibility to introduce a custom middleware in-between that performs a custom job, e.g. checking user permission. If the check is successful, the execution flow continues to the next middleware. However, once the number of routes increases, the query parsing will need to iterate over all register routes inside every middleware. Hence, this might lead to performance degradation.

In this project, the back end was split into several layers. The layering approach helps to reduce the coupling and increase cohesion. The coupling notion shows to which extend two layers are related to one another. If small changes in one layer require the changes in another layer - the layers and highly coupled. In the meantime, cohesion represents the responsibility of a class. If some class is responsible for many things, the cohesion is low as the class is not focused on a single specific task.

In the project, the Server layer is responsible for creating a *mongoDB* connection and starting the listening server. The application logic is separated and located in the app.js file. This file is responsible for registering middlewares such that logging the app status, decoding initial JSON, parsing cookies, and serving static files. Every handler is defined in a separate file to keep a controller small and simple. The service layer provides an access to the specific service, such as a logger or a class to work with GridFS storage. To customize the application behavior, such as database connection, logging storage path, and application HTTP port, the configuration file was added.

5.1.1 Routes

The first route group is *"/o inedata"*. These routes handle the loading and reading of the files. The set of requests is defined in Table-2. In order to get all files with the corresponding metadata, a basic GET request should be used. In the response, the array of all available files will be returned to the requesting party. In case of the file list is empty yet, the status code 404 will be sent.

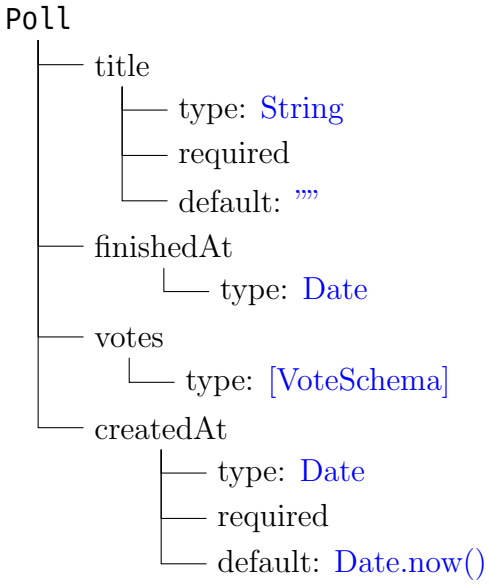
If the specified file is going to be read, the *leName* will be part of the query path. The file will be loaded chunk by chunk due to the internal storing structure described in the GridFs subsection [5.3]. The file could be uploaded via a POST request. The API implements the FormData interface. FormData is based on a javascript object where the key represents the form name and the value corresponds to the actual data.

In this application, the validator checks whether the form has an *o ine-data* name. The value is an array that allows uploading multiple files simultaneously. If the requesting party would like to use a custom FormData name, the *inputHTMLName* property can be customized in the configuration file. Finally, in order to delete a file, the object id should be provided as a part of the query string. This metadata can be found while getting the information about all files. If the deletion is successful, the status code *200 - OK* will be returned. In case the file does not exist the request will end with the *204 - No Content* HTTP code.

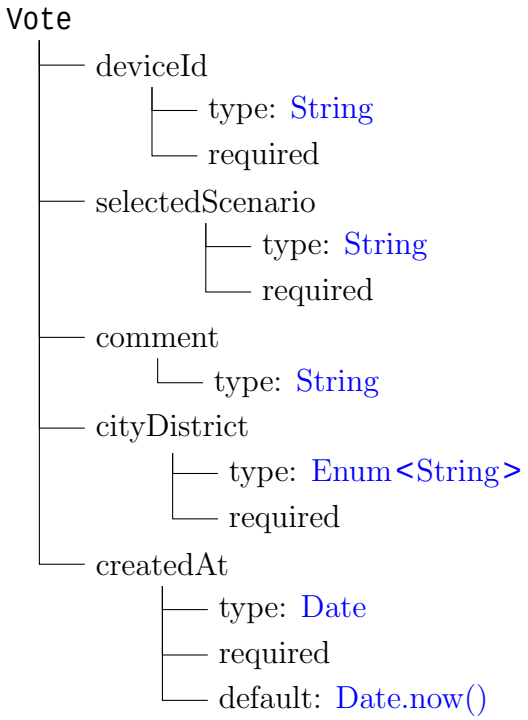
The second group is a poll route. To get all available polls, the GET query should be sent to the *poll/* path. All available queries will be retrieved. To query a specific poll, the *poll/:pollId* path can be used. If the poll with such an identifier does not exist - the 400 status code is returned. To create a new poll, the POST query is available. The body of the request must conform to the poll validation schema. Unless the correct body is sent, the status code 400 Bad Request is returned. The *errors* object contains the validation error and meaningful message what the query is lacking. If the user wants to vote, the *poll/vote* POST request should consist of the following body: *f*pollId: <id of the poll>, vote: VoteInstance*g*. VoteInstance can be created from Vote Schema described below. If the vote object does not conform to the protocol, the corresponding *errors* object will be returned containing all validation issues.

While voting, we are trying to avoid requesting a user's GPS position due to data privacy concerns. However, during voting phase, we need to know at least the district where the user lives. Therefore, the district enumeration can be set up in a config file, namely by setting a poll.cityDistrictsEnum field. To get all votes in a poll - query the *"poll/votes"* path. The body requires *pollId*. Optionally, the votes can be filtered by sending *votesIds* array. If the identifier match - the respective vote is returned.

Validation Schemas



`title`	The name of the poll
`finishedAt`	Time when the poll should be closed
`votes`	Array of votes in current poll
`createdAt`	The timestamp when the poll was created



`deviceId`	The unique identificator of the device
`selectedScenario`	Scenario the user mostly liked
`comment`	Optional comment string
`cityDistrict`	Enum of district of the current city
`createdAt`	The timestamp when the poll was created

Method	Url	Description	Success response
GET	/offlineData	get all files loaded in GridFS	code 200, array with a list of files
GET	/offlineData/:fileName	get specific file by name	code 200, requested file
POST	/offlineData	create a specific file	code 201, *f*"message": "loaded"*g*
DELETE	/offlineData/:fileId	remove file via specific id.	code 200
GET	/poll	get all polls	code 200
GET	/poll/:pollId	get specific poll by id	code 200, requested poll
POST	/poll	create new poll	code 201, *f*"pollId": "some poll Id"*g*
POST	/poll/vote	vote in a poll	code 200, updated poll
POST	/poll/votes	get votes with specific ids	code 200, filtered votes

Table 2: Offline data routes

5.2 Mongoose

Mongoose utilizes two main abstractions: a schema and a model. The schema is a blueprint of the future structure of the respective document. Mongoose supports default JavaScript types and adds a definition for the collections such as *Array* or *Mixed* for an object. Before the usage, the schema should be converted in the respective model. Model is a core class in the mongoose that introduces the binding between the schema and the respective MongoDB collection.

The instance of the model is a document. In order to create a new entry in the database the method save is available on the instantiated model entry. Create, read, update, delete (CRUD) operations are also available as the methods for the model. There are some differences in names, hence the find method corresponds to read, findByIdAndUpdate, and findByIdAndDelete for the updating and deletion respectively. For advance usage, the model contains a watch method that can be used for tracking the changes of the specific collection. Once the changes are applied, the callback function is going to be triggered. Mongoose is well customizable, the developer can introduce its own query functions by patching *Model.methods* object.

5.3 GridFS

To handle large files (around 90 MB), such as *gis_data.json*, the GridFS storage has been created [26]. GridFS is a specification that presents a way how the files over 16 MB can be stored in the database. To avoid performance issues during the reading phase, a single document is divided into several smaller parts (chunks). Every chunk requires 255 KB of the disk size by default. The only exception is the final chunk that might be smaller. GridFS utilizes two different types of data: chunks data and files metadata. The MongoDB driver will merge the required chunk when they are requesting, therefore the file can be loaded fully. There is also a possibility to start reading from the custom index. This can be used, for instance, to play the video content from a specific minute. The main benefit of the GridFS solution is a sequential sending of data without the need to load the whole file in the memory. Therefore the requesting party can start the process the data without the need to wait for the download of the complete file.

As far as wind farm simulation successfully completed its execution, sixteen produced files can be loaded to the back-end through the simple administrative page as depicted in Image-17. Several files can be uploaded simultaneously. All files are validated and should correspond to the predefined naming. Additionally, if the data changes and the new version should be upload, there is no need to delete the last version. An admin can simply upload the new file that will overwrite the old one keeping the files up-to-date.

Choose files No file chosen Upload data

filename	length(B)	uploadDate	md5	contentType	
general_windfarm_data.json					
wind_data.json	2015	2021-07-03T10:19:50.599Z	4f81e6e318e9f791749779bb59eb0076	application/json	✖
gis_data.json					
potential_area.json					
discrete_gis_data_geo.json					
discrete_gis_data_cart.json					
wind_settings.json					
existing_windturbines_cart.json					
optimized_windturbines_geo.json					
optimized_windturbines_cart.json					
shadow_output_cart.json					
shadow_output_geo.json					
noise_levels_cart.json					
noise_levels_geo.json					
turbine_types.json	570	2021-07-03T10:19:50.598Z	43ed72cbe2c40e07ecfddd83e6f8b72a	application/json	✖
windfarm_scenarios.json	188	2021-07-03T10:19:50.599Z	96b4aedab7155b7e9285cf654e51c918	application/json	✖

Figure 17: Simple admin page for loading simulation data

6 Deployment

In order to deploy the application, the deployment phase is split into three sections, namely MongoDB deployment, Node.JS server deployment and Docker-based deployment. As the product is non-commercial, it would be good to find a free-of-charge solution.

6.1 Node.js deployment

In order to deploy a Node.JS application, Heroku hosting has been taken. Heroku is a platform as a service (PaaS) solution that handles containers within itself. Moreover, it supports dozens of adds-on for server monitoring, deployment, and supervision. For instance, Heroku provides a way to control Postgres, Redis, and Apache Kafka itself. The cloud management is app-centric [20], meaning that the developer is not responsible for the load-balancing, routing, and cloud scaling. The platform handles these challenges automatically. Developers are required to provide the codebase using one of 6 different server-based languages, including Node.JS. The management can be conducted either via a web-based interface - Heroku Dashboard or Heroku CLI. One working node is conventionally called dyno. The platform provides 900 computational hours per month for free. That is enough to have one app that will be available 24/7. Apart from automatic scaling, the platform support both horizontal and vertical scaling. While the first approach is useful to handle more parallel work, the latter one can improve the performance for the computationally intense task as the memory or CPU of one dyno is increased.

The further deployment process will be covered using the Heroku CLI as the commands are less possibly to change in the future in comparison to the user interface in the Heroku Dashboard. The first step is a creation of a free user account. The platform requires user data and a credit card to prove the credibility of the customer. Though a user will be charged only if the free hours are utilized. The second step is an installation of the Heroku CLI [19]. There are several operating systems that are supported, including popular macOS, Windows, or Ubuntu.

Once the CLI is installed, the deployment process contains no more than five steps as shown in Listing-16. The user requires to log in as shown in Line-1. They will be redirected to the browser. In case there exist an active session, there is no need to enter the credentials manually. Line-6 shows the next step where the remote repository is set on Heroku's behalf. Here the user can choose the name of the app, for instance, *ar-windfarm*. While deployment, the AR part of the app is being built with a webpack in the cloud. Therefore it's required to have the development dependencies installed. Otherwise, the webpack will not be initialized and the deployment process fails. To avoid this, the environment variable *NPM_CONFIG_PRODUCTION* should be set to false. Heroku Platform tries to optimize the server usage, therefore, especially for a free account, there exists a sleep policy. If the app does not get any requests within thirty minutes, it goes to sleep mode. The next request will wake the dyno app again, though it requires more time, usually around forty seconds. All further requests will be

handled faster. As we possess 900 hours for a month, it is possible to keep the server awake by making self-calls. This functionality is added to the app and can be turned on via an environment variable *NO_SLEEP_HEROKU* set to true.

The project contains several subprojects, including Data Preparation and virtual reality application. Obviously, they do not need to be deployed to the Heroku platform. For that, the *git subtree* command will be used. It takes only the folder *Backend* and pushes the result to the Heroku remote repository.

The Heroku Platform identifies the Node.js app by the package.json file in the root directory. Additionally, the default running script for the Node.js application is *npm start*. It is also possible to change the entry point by setting the Procfile. In this project, the script for running the server is defined under the command *npm run start:server*, hence the Procfile is written as shown in Line-14. After a while, the frontend app will be built with a help of a webpack and the result is put in the server/public/ar folder. There are the static files we will serve for the end-user. As far as the build is available, the platform runs the entry point script. The site can be opened automatically in a browser by running the command *heroku open*.

```
1   heroku login
2   Opening browser...
3   Logging in... done
4   Logged in as panichkinvictor+1@gmail.com
5
6   heroku git:remote -a ar-windfarm
7   set git remote heroku to https://git.heroku.com/ar-windfarm.git
8
9   heroku config:set NPM_CONFIG_PRODUCTION=false NO_SLEEP_HEROKU=
    true
10  Setting NPM_CONFIG_PRODUCTION, NO_SLEEP_HEROKU and restarting ar-
    windfarm... done, v7
11  NO_SLEEP_HEROKU:       true
12  NPM_CONFIG_PRODUCTION: false
13
14  echo "web: npm run start:server" > Procfile
15
16  git subtree push --prefix Backend heroku master
17
18  heroku open
```

Listing 16: Node.js deployment via Heroku CLI

6.2 MongoDB deployment

While working with the application, the MongoDB database was introduced in order to store files produced during the simulation phase. The free-of-charge add-on was provided by the Heroku platform. There was a collaboration with the MLab service. This service provided a free-of-charge MongoDB database that includes 500MB space. However, in October 2020 the add-one was deprecated and it was needed to search for another solution.

MongoDB Atlas[25] is a cloud-based on-demand service that provides a MongoDB database. The provider operates all over the world and is hosted on Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure. The usual billing plans start from 57 dollars a month for the dedicated server. Once the development team requires a multi-region solution, the price starts from 95 dollars a month. Though, for small projects, the service provides free-of-charge shared MongoDB cluster. The RAM in this case is going to be shared among all users. It is possible to get up to 512MB of storage without any payment. On average, space to store files for the simulation job equals 250MB, therefore the MongoDB Atlas option can be utilized in this project.

To start with, the registration should be done. Again, it is required to enter some data and bind the card information for possible future billing. While creating an account, a user can select the hosting platform and the region where the shared instance will be deployed. It is wise to select the instance closer to the potential end-users as the latency will be reduced. For an additional cost, MongoDB Atlas can handle regular backups or conduct database sharding. Lastly, the name of the cluster should be selected. It will be part of the connection URI at the end of the set up process. After several minutes, the database is available as shown in Image-18. To provide secure access, the connection can be checked with the help of username and password, X.509 Certificate, or Identity and Access Management credentials provided by AWS. On top of that, extra settings such as read-write policy or cluster restrictions can be applied. For the current application, the connection will be established via a username and password.

The service restricts the IP addresses from which the connection can be set up. However, while deploying on the Heroku Platform, a user does not have a static IP address. After each deployment, the IP address will be randomly selected. The solution is to set up a strong password and allow access from all possible IP addresses, namely 0.0.0.0/0. This can be done in the *Network access* tab.

6.3 Docker-based deployment

Apart from the deployment to the cloud services, it was decided to containerize an application if there will be a need to quickly deploy the application on the other platform. Docker is a powerful solution that can create an isolated environment whether the app is stored. The main advantage of the platform, that it allows creating an image that can be easily transferred from one system to another. The image can be used as a blueprint for a running container instance. Initially, Docker should be downloaded and installed [9]. The next step is to set up a configuration file as shown in Listing-17. As the Node.js application will be running within our container, the first line describes fetching the image of Node.js version 10.24. If the image is already stored locally, it will be reused. In the next step, the working directory is initialized. If it does not exist, the docker will create the folders itself recursively. In Line 5 the whole folder from the local machine is copied to the current working direction followed by the installation script. Command *COPY* accepts two parameters, namely input directory from

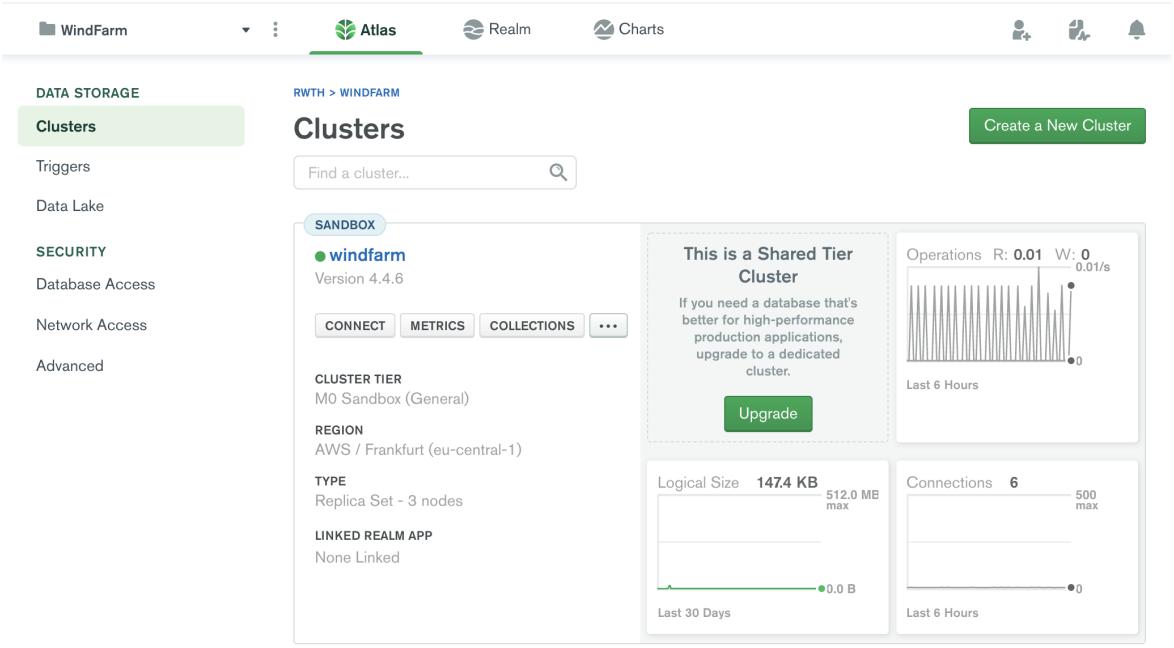


Figure 18: MongoDB Atlas dashboard

the computer and output one within the container. Moving one folder level deeper, npm dependencies for the AR app should be installed. As far as the AR application is built, the script returns to the *Backend/* folder and ends with a command *CMD*. *CMD* command defines the starting script once the container is instantiated from the image.

```
1   FROM node:10.24
2
3   WORKDIR /Backend
4   COPY . .
5   RUN npm install
6
7   WORKDIR ./AR
8   RUN npm install
9   RUN npm run build
10
11  WORKDIR ../
12
13  CMD npm run start:server
```

Listing 17: Docker Configuration

After setting the configuration the image can be built with a command: *docker build {no-cache -t backend:v1*

This command cleans up the state and builds the image from scratch avoiding cached images. The name of the file is provided by the setting -t option. Following the image creation, an app can be started with a help of *docker run* command. However, the app

will not run correctly. The problem is that MongoDB is neither part of the container, nor deployed as a standalone app. The containers contain only the AR app and the back-end Node.js application. To handle multiple container deployments, the docker-compose will be presented.

Docker-compose is a utility that is included within the docker platform. It allows to deploy multiple containers and handle the sequence of such deployments. Docker-compose can be set up via a docker-compose.yaml file. The structure of it is shown in Listing-18. There is multiple version of this utility, hence every file starts with the specific version the script should handle. The next section is services. A service is an atomic docker container that will be created by the docker-compose script. The image can be provided with an *image* key as shown in Line 6 or build at first by the setting *build* path as listed in Line 14. Containers are isolated, hence it is not possible to access the network outside. To allow that, the ports property is defined. The first value shows the outbound port and the second is the inbound port within the container itself.

```
1   version: '3'
2   services:
3     mongoDB:
4       image: mongo:latest
5       ports:
6         - 27017:27017
7       volumes:
8         - mongodb:/data/db
9     backend:
10      command: npm run start:server
11      build: .
12      ports:
13        - 3000:3000
14      depends_on:
15        - mongoDB
16      environment:
17        - MONGO_URL=mongodb://mongoDB:27017/
18        - MONGO_SSL=false
19        - SSL=false
20        - PORT=3000
21      image: app-backend-image
22    reverse:
23      image: nginx:latest
24      ports:
25        - 443:443
26      volumes:
27        - ./nginx/config/:/etc/nginx/
28        - ./nginx/certs/:/etc/ssl/certs/nginx/
29      depends_on:
30        - backend
31    frontend:
32      command: npm run start:server
33      build:
34        context: ../Frontend
35      ports:
36        - 5000:5000
37      environment:
38        - PORT=5000
39      image: app-frontend-image
40  volumes:
41    mongodb:
```

Listing 18: Docker-compose configuration

On every build, the containers will start up from scratch based on the image. However, as we are using the database and storing the data in it, it would be good to have persistence across multiple deployments. Therefore, the *volumes* property is set up. Line-8 shows how the MongoDB container is attached to the MongoDB volume. The order of deployment is important as the server should connect only to the existing database. The *depends_on* key identifies the containers that should start up before the current container is instantiated. Within the node application, the connection URI to the database is used. The environment variable can solve the problem of dynamic

allocation of containers such that we can use the container name as the reference. Optionally, it is possible to set up the containers' names for the readability purpose.

As the server is going to be used to serve multiple applications, the reverse proxy has been configured. The reverse proxy allows hiding the servers themselves and routes requests to the specific server. In this example, it was selected to use an Nginx as a free and open-source technology with a substantial community [17]. The simplest way to use Nginx is to set the image parameter. It will be fetched automatically during the installation process and deployed. As the configuration, it is required to set a *nginx.conf* file.

```
events {
  worker_connections  1024;
}
http {
 server {
    listen 443 ssl;
    server_name ar.soest.renergi.de;
    include common.conf;

    location / {
        proxy_pass http://backend:3000/;
    }
 }

  server {
     listen 443 ssl;
     server_name soest.renergi.de;
     include common.conf;

     location / {
         proxy_pass http://frontend:5000/;
     }
  }
}
```

Listing 19: Nginx configuration file

There are two sections in the Nginx configuration file, namely events and http. The first is events with the *work_connections* attribute. This parameter represents the maximum number of connections one Nginx process can handle simultaneously. The second section is responsible for handling HTTP connections. The server directive defines a virtual route that can handle requests. For this project, it was decided to give a user access to the augmented reality app via *ar* sub-domain. The location directive with the *proxy_pass* attribute defines the internal server that can handle requests. The names for the server corresponds to the names defined in the docker-compose file. This allows to avoid unrestricted access outside the localhost directly to the server and provides a possibility to get the query only through the reverse proxy. A similar section is defined for the first-level domain to serve another project.

After the configuration is set up, the *docker-compose up {detach* script can be run.

It creates both containers with the aforementioned order. Once the process is finished, the application is available under https://soest.renergi.de

7 Conclusion

In this thesis, the problem of auralization and visualization of Wind Farms using Augmented Reality has been solved. First, the main problem was described, namely, the lack of information provided to the citizen in the potential area of the wind farm placement.

Therefore it was decided to create an augmented reality application that can give an understanding of the potential landscape changes once the wind farm is built. Additionally, using the sound model in the potential area, the auralization of the area has been performed. Users can get a noise perception with the respective amplitude based on their current location. Moreover, rotation influences the sound balance between the right and left headsets.

In the beginning, the augmentation task has been solved. Two popular libraries were compared, namely Argon.js and Ar.js. After pros and cons, it was considered to choose the Ar.js due to the possibility to use location-based augmented reality. Due to inaccurate GPS signal, the discretization algorithm has been proposed. The users' position is considered to be the same once they are located in the circle with a specific radius, e.g. fifteen meters. If they escape from the stable zone, the scene is rerendered. Subsequently, the custom A-Frame component has been implemented to customize the wind turbine model for the different scenarios, controlling tower height, blade length, and rotation speed.

This followed with the chapter that described the auralization problem. The rotation-reader that takes the data from the motion sensors was used. Subscribing to the rotation-reader emitter, the sound is adjusted every 50ms. The first sub-task was to identify the angle between the sound emitter (wind turbine) and the user rotation. This has been solved by the usage of quaternions. Projecting two vectors to the X, Z plane, the two-dimensional angle was calculated. The angle is in a range of [- ;] with the edge values representing 100 percent gain of the left or right channel respectively. The transformation of the number value to the balanced sound requires creating a workaround as the StereoPannerNode is not available under iOS. The solution was based on an idea to virtually move an emitter around the scene, while the user stays at the same place.

The next problem was the identification of the sound level in the user's position. The sound model data provides only concentric polygons where the amplitude is known on the borders. The naive solution utilized the point inclusion algorithms. Two algorithms were considered, implemented, and compared, namely the Winding number algorithm and PnPoly. They both show similar performance regardless of the number of points. Therefore, all polygons are considered starting from the smallest to the largest in order to find the first inclusion and, consequently, identify the noise in the complete area.

Despite the simplicity of this approach, it has two major problems. The first one is

related to the size of the polygons. Once the user is located in the same polygon the sound level amplitude is stable, even while the user is moving. Secondly, once a user crosses the borderline between polygons the amplitude can increase rapidly due to the non-linear nature of the dB scale.

This problem has been approached using interpolation. The polygon has been triangulated using the Delaunay triangulation. The next step was a localization of the triangle where the user is currently located. For that, the walk-and-jump algorithm has been considered. Lastly, the value within the triangle has been interpolated using three vertexes within the barycentric coordinate system. The selection of such a specific coordinate system was caused by getting the wrong value, once the triangles are obtuse or the user is located near the triangle's edge.

In the production environment, the polygons were split into 1200 triangles. The localization requires 10 iterations at most, checking one triangle in every step. To save mobile resources, it was decided to migrate the solution to the back end and precompute the triangulation data during the server start-up. After this tuning, the overall request to the back end takes around 17ms that allows providing a real-time experience for the end-user at any point of space. To avoid the repeated sound level update, the value is also recalculated only if the user escapes the stable circular zone using the same discretization algorithm as in augmented reality.

As far as the main goals were achieved, the back end has been added. It is mostly used for accessing the simulation wind model data, serving the AR application, and preprocessing Delaunay triangulation. MongoDB is selected as a database as it is flexible with respect to the structure and includes GridFS storage, which allows the handling of large files.

Last but not least, the application has been containerized using docker. This allows to ease the deployment to the end-server and avoid potential environment differences. Moreover, the Nginx has been added as a reverse proxy. As a result, serving several applications from one server became possible preventing the exposure of the back-end to the public.

Finally, future work has been proposed including object occlusion detection and computer vision integration in order to improve scene stability.

8 Future work

While working on the augmentation and auralization, three additional problems appeared that are required substantial research.

The first problem is related to the augmented reality. Despite the fact, that the model is bound to the user's latitude and longitude, the model continues to "shake". This is caused by the sensitivity of mobile sensors. The possible direction of solving the problem might be working with the sensors directly or analyzing the scene using computer vision. The latter approach can be started with horizon detection where the base of the wind turbine will be bind. Later, other artifacts can be detected.

This also can help to solve the second problem, namely occlusion detection. Occlu-

sion is a blocking of the augmented reality content with real environmental object, for instance, the building that stands in front of the user. The problem gets complicated due to the outdoor AR. The approach that might solve the issue is a combination of the GIS data and the raycasting in the direction of the user's sight.

Lastly, due to the browser's restriction, the sound balancing using the basic PannerNode is not optimal. Once the stereoPannerNode is available in all modern browsers, the problem will be solved by removing the deprecated code. Still, for now, it might be possible to find a better solution for handling balance in the earphones, including solutions that work with the native device API instead of browser one.

References

[1] Alexis Deveria. Navigator api: geolocation browser support, 2018. URL https://caniuse.com/mdn-api_navigator_geolocation. last accessed on 12.04.2021.

[2] Borut alik. An efficient sweep-line delaunay triangulation algorithm. *Comput. Aided Des.*, 37(10):1027–1038, September 2005. ISSN 0010-4485. doi: 10.1016/j.cad.2004.10.004. URL https://doi.org/10.1016/j.cad.2004.10.004.

[3] Apple. ios-specific considerations, 2012. URL https://developer.apple.com/library/archive/documentation/AudioVideo/Conceptual/Using_HTML5_Audio_Video/Device-SpecificConsiderations/Device-SpecificConsiderations.html. last accessed on 12.04.2021.

[4] Blair MacIntyre. Argon.js Documentation. https://docs.argonjs.io/, 2017. Online; accessed 2021-04-10.

[5] Jesus A. De Loera, Jorg Rambau, and Francisco Santos. *Triangulations: Structures for Algorithms and Applications*. Springer Publishing Company, Incorporated, 1st edition, 2010. ISBN 3642129706.

[6] Federico Debandi, Roberto Iacoviello, Alberto Messina, Maurizio Montagnuolo, Federico Manuri, Andrea Sanna, and Davide Zappia. Enhancing cultural tourism by a mixed reality application for outdoor navigation and information browsing using immersive devices. 364:012048, jun 2018. doi: 10.1088/1757-899x/364/1/012048. URL https://doi.org/10.1088/1757-899x/364/1/012048.

[7] Luc Devroye, Christophe Lemaire, and Jean-Michel Moreau. Expected time analysis for delaunay point location. *Computational Geometry*, 29(2):61–89, oct 2004. doi: 10.1016/j.comgeo.2004.02.002. URL https://doi.org/10.1016%2Fj.comgeo.2004.02.002.

[8] Diego Marcos, Don McCurdy. A-frame *j* sound component, 2015. URL https://aframe.io/docs/1.2.0/components/sound.html. last accessed on 12.04.2021.

[9] Docker Inc. Docker *j* get started, 2013. URL https://www.docker.com/get-started. last accessed on 23.05.2021.

[10] Expressjs organisation. Express - Fast, unopinionated, minimalist web framework for node. https://github.com/expressjs/express, 2016. Online; accessed 2021-04-04.

[11] Federal Ministry for Economics Affairs and Energy. Renewable Energy Sources Act overview. https://www.bmwi.de/Redaktion/EN/Dossier/renewable-energy.html, 2017. Online; accessed 2021-04-04.

[12] Steven Feiner, Blair Macintyre, Tobias Höllerer, and Anthony Webster. A touring machine: Prototyping 3d mobile augmented reality systems for exploring the urban environment. volume 1, pages 74–81, 12 1997. doi: 10.1007/BF01682023.

[13] OpenJS Foundation. Performance measurement apis in node.js, 2017. URL https://nodejs.org/api/perf_hooks.html. last accessed on 09.05.2021.

[14] Randolph Franklin. Pnpoly - point inclusion in polygon test, 1970. URL https://wrf.ecse.rpi.edu/Research/Short_Notes/pnpoly.html. last accessed on 02.05.2021.

[15] Leonidas Guibas and Jorge Stolfi. Primitives for the manipulation of general subdivisions and the computation of voronoi. *ACM Trans. Graph.*, 4(2):74–123, April 1985. ISSN 0730-0301. doi: 10.1145/282918.282923. URL https://doi.org/10.1145/282918.282923.

[16] Wei Huang, Min Sun, and Songnian Li. A 3d gis-based interactive registration mechanism for outdoor augmented reality system. *Expert Systems with Applications*, 55, 02 2016. doi: 10.1016/j.eswa.2016.01.037.

[17] Igor Sysoev. Nginx *j* high performance load balancer, web server, reverse proxy, 2004. URL https://www.nginx.com/. last accessed on 12.06.2021.

[18] MapBox Inc. A javascript library for delaunay triangulation, 2017. URL https://github.com/mapbox/delaunator. last accessed on 12.05.2021.

[19] James Lindenbaum, Adam Wiggins, Orion Henry. The heroku command line interface (cli), 2007. URL https://devcenter.heroku.com/articles/heroku-cli. last accessed on 23.05.2021.

[20] James Lindenbaum, Adam Wiggins, Orion Henry. The heroku platform, 2007. URL https://www.heroku.com/platform. last accessed on 23.05.2021.

[21] James Simpson. Howler.js documentation, 2016. URL https://howlerjs.com/. last accessed on 12.04.2021.

[22] Khronos Group. gltf *j* overview, 2015. URL https://www.khronos.org/gltf/. last accessed on 12.06.2021.

[23] Gun Lee, Andreas Duenser, Seungwon Kim, and Mark Billinghurst. Cityviewar: A mobile outdoor ar application for city visualization. pages 57–64, 11 2012. ISBN 978-1-4673-4663-4. doi: 10.1109/ISMAR-AMH.2012.6483989.

[24] MDN. Stereopannernode specification, 2020. URL https://developer.mozilla.org/en-US/docs/Web/API/StereoPannerNode#browser_compatibility. last accessed on 12.04.2021.

[25] MongoDB Inc. Global multi-cloud database, 2009. URL https://www.mongodb.com/cloud/atlas1. last accessed on 23.05.2021.

[26] MongoDB Inc. GridFS — MongoDB Manual. https://docs.mongodb.com/manual/core/gridfs/, 2020. Online; accessed 2021-04-04.

[27] Stelian Persa and Pieter Jonker. On positioning for augmented reality systems. In *Proceedings of the 1st International Symposium on Handheld and Ubiquitous Computing*, HUC '99, page 327–329, Berlin, Heidelberg, 1999. Springer-Verlag. ISBN 3540665501.

[28] Chris Pike, Peter Taylour, and Frank Melchior. Delivering object-based 3d audio using the web audio api and the audio definition model. In Samuel Goldszmidt, Norbert Schnell, Victor Saiz, and Benjamin Matuszewski, editors, *Proceedings of the International Web Audio Conference*, WAC '15, Paris, France, January 2015. IRCAM.

[29] Ville Pulkki. virtual sound source positioning using vector base amplitude panning. *journal of the audio engineering society*, 45(6):456–466, june 1997.

[30] Miguel Ribo, Peter Lang, Harald Ganster, Markus Brandner, Christoph Stock, and Axel Pinz. Hybrid tracking for outdoor augmented reality applications. *Computer Graphics and Applications, IEEE*, 22:54 – 63, 12 2002. doi: 10.1109/MCG.2002.1046629.

[31] Statista. Renewable energy in Germany. https://www.statista.com/study/59612/renewable-energy-in-germany/, 2018. Online; accessed 2021-04-04.

[32] Statista. Investments in renewable energy plants in Germany from 2000 to 2019. https://www.statista.com/statistics/583526/investments-renewable-energy-plants-germany/, 2019. Online; accessed 2021-04-04.

[33] Statista. Interest in renewable energy in Germany in 2020. https://www.statista.com/statistics/1099204/renewable-energy-interest-by-gender-germany/, 2020. Online; accessed 2021-04-04.

[34] W3C. Web audio api *j* editor's draft, 2021. URL https://webaudio.github.io/web-audio-api/. last accessed on 12.04.2021.

[35] Kevin Weiler. *An Incremental Angle Point in Polygon Test*, page 16–23. Academic Press Professional, Inc., USA, 1994. ISBN 0123361559.

[36] Lewis Van Winkle. Interpolating in a triangle, 2016. URL https://codeplea.com/triangular-interpolation. last accessed on 12.05.2021.