FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Automatic Creation of Science and Engineering Fairs in Virtual Reality
Student:	Bc. Tomáš Bašta
Supervisor:	doc. Ing. Mgr. Petr Klán, CSc.
Study program:	Informatics
Branch / specialization:	Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2022/2023

Instructions

Realize an automatic system for science and engineering fairs in virtual reality. Data from participants will be collected via web application. Fair will be realized by a combination of immersive and collaborative telepresence.

Follow the steps below:
1. Learn the basic principles of science and engineering fairs and conferences.
2. Design a web application used for collecting conference-related assets (abstracts, posters, photos, speech recordings, 3D models).
3. Implement the web application.
4. Learn the principles of visual programming in metaverse Neos VR.
5. Design a template virtual world where the conferences will take place.
6. Create and build the template virtual world in Neos VR.
7. Connect the web application with the template virtual world.
8. Visually program the automatic generation of new virtual worlds containing the corresponding assets.
9. Test the automatic fair system by holding a testing science or engineering conference.
10. Publish the system for public use.

Electronically approved by Ing. Michal Valenta, Ph.D. on 21 December 2021 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Automatic Creation of Science and Engineering Fairs in Virtual Reality

Bc. Tomáš Bašta

Department of Software Engineering
Supervisor: doc. Ing. Mgr. Petr Klán, CSc.

May 3, 2023

Acknowledgements

I would like to express my deepest appreciation to my supervisor, doc. Ing. Mgr. Petr Klán, CSc., for his enthusiastic approach towards my thesis and for his invaluable input that helped me to produce the best result that I possibly could. A big thank you also goes to my girlfriend Marie, who lovingly kept on supporting me during the whole period of writing this thesis, even though I have probably been unbearable during this time. Next, I would like to thank my parents, Iva and Roman, for providing me with encouragement throughout my academic pursuit and for helping me to become the person I am today. Finally, I would also like to thank my Russian Blue kitten Samuel for his constant purring that gave me the emotional support that I so desperately needed.

Declaration

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

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

In Prague on May 3, 2023 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Bašta, Tomáš. *Automatic Creation of Science and Engineering Fairs in Virtual Reality*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Abstract

This thesis describes the process of developing a system for automatic creation of science and engineering fairs in virtual reality. The thesis starts by describing what the science and engineering fairs are and introduces the Regeneron International Science and Engineering Fair as an example. Afterwards, the whole system, which is composed of two parts, is designed, implemented and tested. Its first part is a web application written in Go, through which the users can define the fairs and subsequently upload their projects. The second part is a virtual world that is created in the Neos VR metaverse. This world communicates with the web application from which it loads the participants' projects, which are then displayed in three dimensions inside of the world. The resulting system is a fully functional solution that is freely available and which can be used by both individuals and institutions to automatically create their own science and engineering fairs in virtual reality.

Keywords science fairs, web application, Go, virtual reality, virtual world, Neos VR

Abstrakt

Tato práce popisuje proces vývoje systému pro automatickou tvorbu vědeckých a inženýrských konferencí ve virtuální realitě. Práce začíná vysvětlením toho co to vědecké a inženýrské konference jsou a uvádí Regeneron International Science and Engineering Fair jako příklad. Následně je celý systém, který se skládá ze dvou částí, navržen, implementován a otestován. Jeho první částí je webová aplikace napsána v jazyce Go, pomocí které mohou uživatelé konference vytvářet a následně do nich nahrávat své projekty. Druhou částí je pak virtuální svět, který je vytvořen v metaverzu Neos VR. Tento svět komunikuje s webovou aplikací, ze které načítá projekty jednotlivých účastníků a ty jsou pak ve světě prostorově zobrazovány. Výsledný systém je plně funkční řešení, které je volně k dispozici a je možné ho použít pro automatickou tvorbu vědeckých a inženýrských konferencí ve virtuální realitě, jak jednotlivci, tak i institucemi.

Klíčová slova vědecké konference, webová aplikace, Go, virtuální realita, virtuální svět, Neos VR

Contents

List of Figures

List of Tables

List of Listings

Introduction

Science conferences are events where researchers share their latest findings and breakthroughs. It is also a place where they can connect with their colleagues to get inspired, discuss the obstacles they are facing, or receive a new, unbiased point of view. Similarly, science fairs give a chance to young passionate students interested in science to taste what the work of a researcher entails. For these students, science fairs offer the chance to socialise with similarly minded peers and build lifelong lasting friendships.

In all the above stated cases, the importance of human interaction is apparent. This was not an issue when science fairs were organised in person. However, the current situation, mostly caused by the COVID-19 pandemic, made it obvious that holding events physically is not always an option. This meant that activities which were previously held solely in person had to be changed into some form of remote variation or be cancelled completely. This brought the era of home office, remote lectures and conference calls. Science fairs were also not spared from this remote mode shift. While some fairs got cancelled altogether, the rest were conducted via meeting calls or other remote methods with limited human to human interaction. This led to various issues which significantly worsened the attending experience for the contestants. Problems, such as the usage of body language, ability to focus, or human networking are all issues that are difficult to solve with the common approach of meeting calls. The technologies which can address these issues have already been developed, but as of today, are still not commonly used in conventional software solutions.

In this thesis, a system capable of addressing these issues is designed and implemented by utilising virtual reality and metaverse. Virtual reality, as well as metaverse, is, however, still a relatively new concept that most of the public is not familiar with. Because of this, any solution which would require the users to perform a complicated sequence of operations in the virtual reality is doomed to fail. For that reason, the virtual reality application is accompanied by a web application, a concept known by the majority, which will collect

assets from the users and transfer them into a virtual world. By opting for this approach, users will upload their materials in a familiar way and will not have to worry about getting them into the virtual reality itself. This way, the resulting system should be easy to use without sacrificing any essential functionality.

Chapter 1

Objectives

The overall goal of this thesis is to design, implement and test a system capable of automatic creation of science and engineering fairs in virtual reality. In order to make the implementation process simpler, a number of smaller sub-objectives were devised, which go through the process in logical succession and roughly correspond to the thesis assignment. These sub-objectives are, stated in the order they appear in the thesis, as follows:

Investigation of science and engineering fairs.
In this chapter, science and engineering fairs are examined to learn what they are and how they are usually organised. Next, the Regeneron International Science and Engineering Fair is introduced, which is currently the largest pre-college science fair held every year in the United States. Ultimately, the chapter explores how individual science fairs handled the switch to remote mode during the COVID-19 pandemic and it is evaluated whether this transformation was successful or not.

Implementation of the web application.
Based on the information gathered from the previous chapter, a web application that is capable of gathering and distributing science fair assets is designed and implemented. This chapter also briefly discusses all the used technologies and explains the reasons for choosing them. In the end, the web application is tested by unit tests and is deployed on the Internet.

Implementation of the virtual reality application.
In this chapter, the Neos VR metaverse is introduced as the main enabling technology for the virtual reality part of the system. Throughout this chapter, the most essential principles of Neos VR are explained with a special care being given to its visual programming language called LogiX. Afterwards, the virtual world inside of Neos VR is designed, implemented and published in the Content Hub for public use.

Test of the final system.
As the last step, the resulting system is checked by holding a testing science fair. Next, any potential issues that are uncovered by the testing are fixed and it is evaluated whether the system successfully fulfils the thesis assignment and if it can be used as an alternative method for holding remote science and engineering fairs.

Chapter 2

Science and engineering fairs

Science and engineering fairs, sometimes also referred to as science conferences, are competitions intended for elementary, middle, and high school students. Their focus is to get students more interested in science by giving them a task of creating a science project. Each student must find an area he is interested in, formulate a question or a hypothesis, design a research method, gather relevant data, form a conclusion and ultimately create and present a poster showcasing his results. By going through this process, which roughly corresponds to the scientific method, students can use the knowledge learned in classrooms in a real-world scenario, thus enhancing the comprehension of the subject [1]. Furthermore, it can also improve their creative thinking, the ability to work with data, and even inspire them to pursue a career in science or engineering [2].

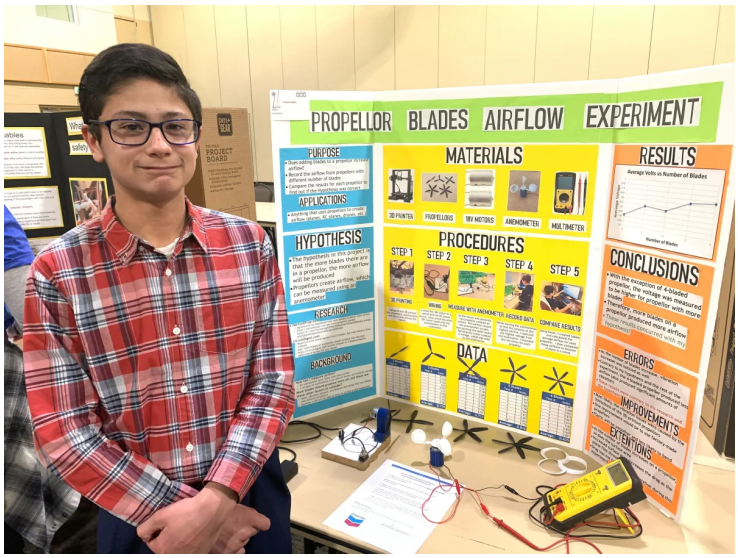


Figure 2.1: Student presenting his poster during a local science fair [3]

Science and engineering fairs are held, with some alterations, worldwide and often follow a similar pattern. The whole process usually starts at local schools where students participate in a fair by presenting their findings in the form of a poster [2], as seen in figure 2.1. This participation can be mandatory or optional for an extra credit or a better grade. Because science fairs are essentially competitions, they include a judging process where the best students advance to regional and subsequently to national or even international rounds. The biggest international science fairs include Regeneron International Science and Engineering Fair (ISEF) [4] held yearly in the United States, and the European Union Contest for Young Scientists (EUCYS) [5] which occurs every year in Europe. Because these rounds are often sponsored by big science and technological companies, they feature cash prizes, awards or scholarships for the best students. It is also possible for students to arrange internships or even secure a job in one of these companies. In addition, students can interact with each other to make like-minded friends or form groups to tackle more challenging problems.

2.1 History

The history of science fairs can be dated back to the 1930s, when the American Institute of the City of New York began uniting after-school science clubs into an association [6]. Their focus was to get young students more interested in STEM (science, technology, engineering, and mathematics) fields and to get them to experience science by doing, rather than by memorising formulas. Initially, science fairs were mostly about science demonstrations, rather than discovery and innovation projects, which define science fairs as we know them today. However, this has changed during the years 1939 and 1940, when the New York World's Fair was held [7]. This large event featured several thousand, mainly high-school, students presenting exhibits from various fields, such as chemistry, engineering, or physics. The event was met with a large public success and over 10 million visitors came to see it. Among them were also George Edward Pendray, the executive of the Westinghouse company, and Watson Davis, the director of the Science Service, which later became known as the Society for Science. Intrigued by what they saw at the World's Fair, they started discussing how to encourage more students to pursue science and ultimately started their own national science competition, called the Westinghouse Science Talent Search (STS). The Westinghouse company continued to sponsor the STS until the year 1998, when the main sponsor became Intel, which was afterwards succeeded by Regeneron in the year 2016 [8]. In 1950, Society for Science started another event called the International Science and Engineering Fair (ISEF). ISEF started as a national science fair, but quickly expanded and became an international competition, and, as of today, is the largest pre-college STEM competition in the world.

Inspired by the success of the science and engineering fairs in the United States, many local and national science fair competitions started forming all over the world. However, they did not become an integral part of the countries' education systems, as in the US, and are considered a special extracurricular activity that only a fraction of students participate in. The reason science fairs are so popular in the United States, and not so much anywhere else in the world, can be traced to the Next Generation Science Standards [9]. These standards, published in 2013 in the United States, are split into three dimensions: Cross-cutting Concepts, Science and Engineering Practices, and Disciplinary Core Ideas. Each dimension contains a list of guidelines and recommendations, with the overall idea of teaching STEM fields in context rather than in vacuum, and focusing on deep understanding and application of a smaller amount of essential ideas, rather than on a shallow understanding of numerous concepts. For example, the guidelines for the Science and Engineering Practices states, that students should understand and learn the following eight practices:

1. Asking questions and defining problems.
2. Developing and using models.
3. Planning and carrying out investigations.
4. Analysing and interpreting data.
5. Using mathematics and computational thinking.
6. Constructing explanations and designing solutions.
7. Engaging in argument from evidence.
8. Obtaining, evaluating, and communicating information.

The correspondence between these practices and science fairs is apparent, and is most likely the main reason local science fairs are a part of science curriculum in schools all over the United States.

2.2 International Science and Engineering Fair

In order to study the specifics of science and engineering fairs on a concrete example, the Regeneron International Science and Engineering Fair (ISEF) [4] is presented, which was already introduced in the previous sections. Regeneron ISEF is an international science and engineering fair organised every year by the Society for Science in the United States. ISEF is intended for students in grades 9 – 12 (ages 14 – 18) who were selected from ISEF affiliated local, regional, or national science fairs, which can be found in over 75 countries all over the world. Upon qualifying, students compete with their projects in 21

categories, each with several subcategories, such as animal sciences, biomedical engineering, mathematics, or systems software. Each year, over 1800 students qualify to compete in the event and roughly 1000 researchers and scientists volunteer to judge their efforts in a setting similar to that shown in figure 2.2. Thanks to the high number of sponsors, winners split over $6 million in prize money, awards, and scholarships, with the top prize of $75,000 awarded to the best overall project.

Figure 2.2: Finalist hall during the Regeneron ISEF [4]

ISEF is a multiple day event and usually spans from Sunday to Friday. The fair's program starts by contestants dropping off their projects, which then undergo a safety inspection and check by the Science Review Committee [10]. On Monday and Tuesday, a wide selection of events is held which the participants can attend, such as lectures, workshops and symposia. During this time, the participants are also given various opportunities to interact with each other, tour the town the fair is organised in, and prepare for the presentation of their project. Wednesday marks the start of the main event when, during multiple sessions, judges interview the contestants and judge their projects. This judging process requires the contestants to be present at the respective booth with their poster and present the project to the judge similarly, as shown in figure 2.3. The judge then, based on the established guidelines, evaluates the project on a scale of 0 – 100, taking into consideration aspects such as creativity, the oral presentation and the correctness of the scientific approach [4]. The next day, the science fair opens to the public and the special awards ceremony is held. Here, various prizes, such as scholarships

or internships, are awarded to the selected projects from the fair's sponsors. On the last day, the grand awards ceremony is held where the winners for each individual category and the overall winners are announced.

Figure 2.3: Student presenting poster to judge during the Regeneron ISEF [4]

The Regeneron ISEF offers a wonderful opportunity for young students interested in science to engage in their hobby. They obtain invaluable feedback on their projects from real researchers and might even win various cash prizes or scholarships. Thanks to the human networking aspect of the event, they can also discuss their research with the members of numerous research or educational institutions, which might help them decide what they want to do after graduating from high school. In addition, participants are given opportunities to interact with each other, which, thanks to the international nature of the event, might help them learn something new about other cultures. In the end, Regeneron ISEF is an amazing experience for all the participants, and can be the deciding factor why students decide to pursue a career in science.

2.3 Virtual science and engineering fairs

When the global pandemic of COVID-19 started in March 2020, one of the most important safety precautions was to limit physical contact between people as much as possible. This meant that most companies opted for home office and children were educated online through applications such as Google Classroom or Microsoft Zoom. Because of the increased organisational complexity and the lack of experience, numerous organisers did not follow this trend and

cancelled their science and engineering fairs for the time being. However, some of them persevered and successfully switched the fairs to a virtual variant. The individual methods of the virtual fairs differed based on the specific event, but options such as broadcasting pre-recorded speeches, describing the projects over a Zoom meeting call, or presenting the digital versions of the projects via Microsoft PowerPoint could have all been seen [11]. However, these methods also come with some disadvantages, which are mostly caused by the reduction of human representation to a mere video frame. Acts, such as using body language to further emphasise various sections of the presentation, or changing the content and form of the presentation based on the audience's expression, suddenly become hard when the human-to-human contact is removed from the events. Also, students usually consider the interaction and collaboration aspect as the most valuable feature of the science fairs [12]. With virtual science and engineering fairs, these aspects are often omitted, which might ultimately lead to a reduced number of students that will participate in these events. However, there are also some substantial advantages to holding virtual science fairs. For one, once the process of organising the event has been established, it is relatively easy to redo it in the following years. There are also no costs related to venue renting, which might notably reduce the organisational costs. Also, if the used technologies allow for it, it is often possible to account for a higher number of participants compared to physical variants, and to even make the event accessible to the public with little additional work.

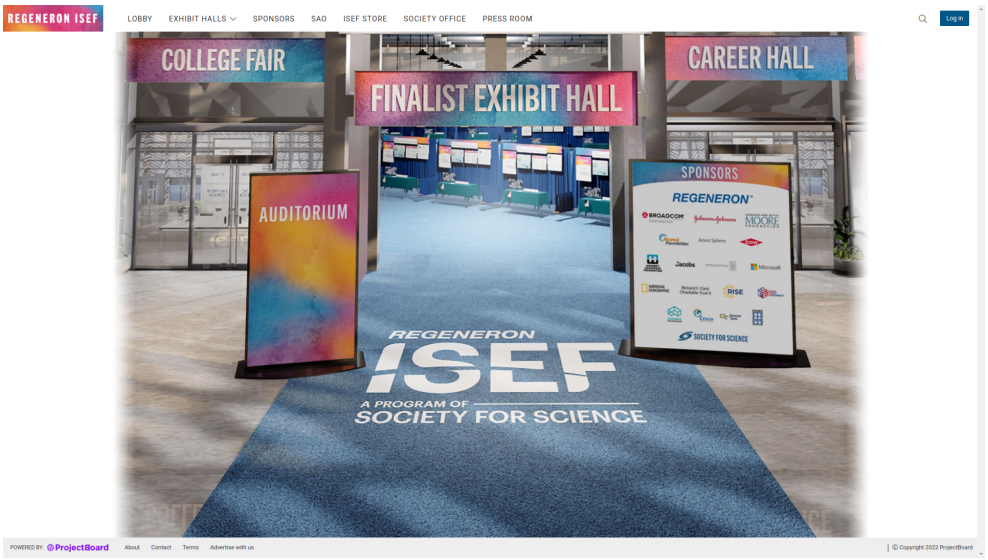


Figure 2.4: Screenshot of the Regeneron ISEF 2022 landing page

The Regeneron ISEF is one of the science and engineering fairs which has successfully adapted to the need to host the event virtually. The organisers at Society for Science have opted for an unusual approach, where they used a spe-

cial purpose product called ProjectBoard [13], which is shown in figure 2.4. ProjectBoard is a software platform that is specifically aimed at hosting virtual STEM events and student fairs, and is essentially a content management system where students upload the digital version of their project, video containing the presentation, and any other additional documents. During specific times, the judges and public can also chat with the participants via a live chat to ask them questions. ProjectBoard also gives organisers the option to customise the event's landing page and booth design, as shown in figure 2.5, to match the appearance of the real event. This solution, however, suffers from the same issues as the options stated in the previous paragraph. Here, the human-to-human interaction aspect is eliminated almost completely, except for the written chat, which might make the project's presentation difficult. It can also be noted that, while it may initially look interesting, it is not anything special and will most likely not motivate students to pursue a career in science in the long-term. Despite all that, the ProjectBoard platform was successfully used for hosting not only the ISEF but also the years 2021 and 2022 of the Canada-Wide Science Fair (CWSF) [14] and the American Junior Academy of Science (AJAS) [15] conference's poster session. This year (2023), ISEF is scheduled to be resumed as a physical event in Dallas, Texas, with a supporting virtual part hosted on ProjectBoard, where the public will browse through the participant's projects and engage with their authors through chat.

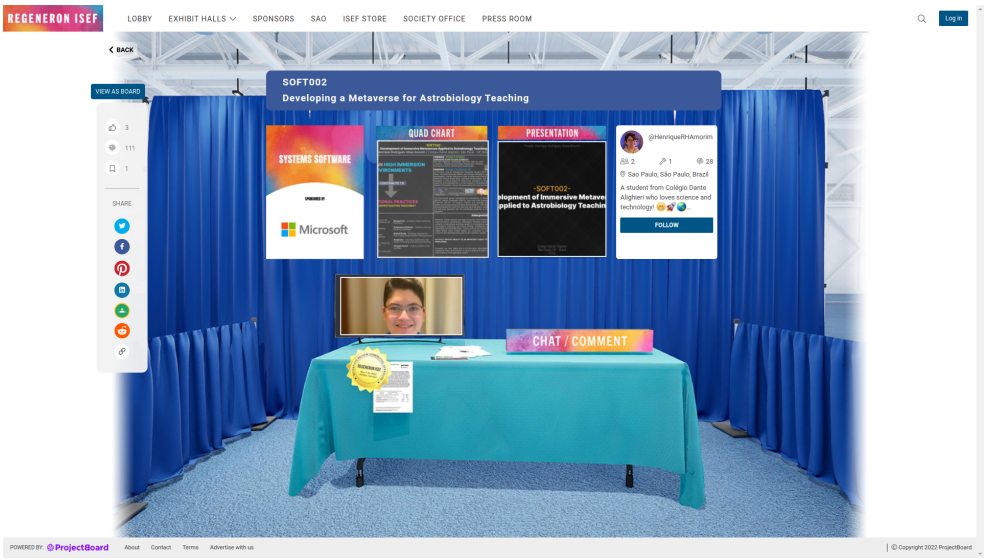


Figure 2.5: Screenshot of a student's booth from the Regeneron ISEF 2022

With the current improvement of the COVID-19 pandemic, science and engineering fairs are slowly becoming organised physically again. However, the situation is somewhat different from what it was before. Some science fairs are newly featuring virtual aspects such as allowing the public to browse

through the projects online, while others choose to continue in a hybrid mode, which is a mode where some participants are present physically and some are connected virtually. On the other hand, some organisers might have found that virtual events work the best for their specific use cases and audience, and will from now on prefer holding virtual events over physical ones. For this reason, it would be highly desirable to develop a new, more immersive system capable of addressing at least some of the above stated issues, which is the goal that this thesis is trying to achieve.

Chapter 3

Web application

Utilising the newly acquired information about the science and engineering fairs and conferences, this chapter describes the development process of the first part of the system for the automatic creation of these types of events in virtual reality. This part is realised through a simple web application, which will work as a content management system that can be used by the science fair organisers to specify the details of the event they are holding. Next, users interested in participating in the event will upload their submissions to the event, which will contain their project's poster, abstract, audio recording of their speech and possibly any additional images or three-dimensional (3D) models. Then, when the date of the event comes, the event organiser will log into a special virtual world, which is developed later in this thesis, which will communicate with this web application to automatically create the environment for the science fair by importing the participants' assets.

3.1 Analysis

To specify the requirements for the web application, the gathered knowledge about science and engineering fairs from the previous chapter 2 was used. Based on this knowledge, the core requirements were specified, which made sure that the resulting application will be able to address all the usual demands of the fairs. This core was then further expanded by including the requirements from the thesis assignment and by additional ones that were gathered upon a discussion with the supervisor, who is the intended user of the system. The final collection of the requirements also contains a few that were not explicitly requested, but are generally considered an industry standard for this type of application. To formally state the requirements, a combination of functional requirements, non-functional requirements, and use cases was used, which are all commonly used for the requirements engineering process.

3.1.1 Functional requirements

Functional requirements define the functionality which the system should have to fulfil its purpose. They should contain a name along with a brief description of that specific functionality. Usually, it is also desirable to include an indication of the requirement's importance and difficulty of implementation, based on which the selection of the implemented requirements is made. However, since this web application is relatively simple and does not feature that many requirements, this evaluation step was skipped, because all the requirements will be implemented. Next, to make it possible to address individual requirements, each of them will also contain a unique identifier in the form of *FRx*, where *FR* stands for functional requirement and x is the sequence number of that specific requirement. The functional requirements that will be implemented in this thesis are:

- **FR1 - Management of user accounts**
  Because it is required to link the ownership of events and submissions to the users, a simple user management must be implemented. This will include the creation of a new account and logging in and out of the account.

- **FR2 - Management of events**
  The events will form the core of the web application. Logged-in users must have the ability to create new events by specifying the basic information such as the name, description or date of the event. It should also be possible to define what assets, such as the poster, audio, images, or models, are required to participate in that specific event. Furthermore, logged-in users must be able to edit and delete their own events.

- **FR3 - Management of submissions**
  Logged-in users must be able to upload submissions for the individual events. The submission content will differ based on the event's settings, but it should be possible to add both texts and files as a part of the submission. Upon submitting, the author must also be able to edit and delete his submission.

- **FR4 - Accepting common formats for different asset types**
  Users must be given the option to upload their assets in multiple different formats. The system should not force them to use a single proprietary format, which is otherwise rarely used.

- **FR5 - Specific purpose single person events**
  Since the aim of the whole system is to generate a virtual world with the uploaded assets, it should be possible to create a specific type of events where only one person will participate. Therefore, it should be possible to allow one participant to have multiple submissions per single event.

This will allow users to utilise the system for the creation of virtual galleries and presentation displays, besides the science and engineering fairs.

- **FR6 - Accessing submission data through VR application**
  In order to access the required information through the virtual reality application, the web application must be able to supply the information in a standardised machine-readable format.

3.1.2 Non-functional requirements

Non-functional requirements do not specify the functionality of the system, but rather the attributes and constraints that are required from the system. They often describe attributes such as speed, reliability, or security, which is why they usually dictate the architecture of the system. When defining the non-functional requirements, it is important to always specify the metric of the requirement, which can later evaluate whether the requirements were fulfilled or not. Similarly to functional requirements, an identifier will be assigned to each requirement in the form of *NFRx*, where *NFR* stands for non-functional requirement and x is the sequential number of that specific requirement. For this web application, the following non-functional requirements were specified:

- **NFR1 - User facing web application**
  The application will be accessible through the Internet from everywhere and will feature a graphical user interface through which the users will interact with it.
- **NFR2 - Data persistence**
  The application's data and submissions' assets will be persistently stored in a way where restarts or upgrades of the application do not result in a loss of data.
- **NFR3 - Encrypted communication**
  Because the application will require the users to login via password, the communication between the application and the users has to be encrypted.
- **NFR4 - Resilience**
  While it would be optimal, applications are rarely made with no bugs. For this reason, the application will contain a fail-safe mechanism that will recover the application in case any error occurs during the request handling instead of crashing the application.

3.1.3 Use cases

Use cases are formal descriptions of the smallest sequences of actions that achieve a common goal within the system. The usual content of the use case

differs based on the project's needs. In this thesis, a simple representation of the use case was selected that contains its name, the executing actor, sequence of steps, and, optionally, the requirements needed to execute the use case. In this context, the actor can be understood as the user, which will usually perform the specific use case. To later use them in the use cases, the following three actors are defined:

- **Guest** - non-logged in user accessing the system via browser
- **Member** - logged in user accessing the system via browser
- **VR application** - virtual reality application accessing the system via a standardised protocol

Besides the above stated attributes, a unique identifier was also assigned to each use case in the form of *UCx*, where *UC* stands for use case and x is the sequential number of that specific use case. The use cases that will get implemented in this thesis are:

- **UC1 - Register**
  Actor: Guest
  Requirements: An account with the same email does not exist.
  Sequence: This use case starts when the actor is on the initial page of the application.
  1. Actor clicks the *Login* button at the top of the page.
  2. System displays the login page.
  3. Actor clicks the *Register here* hyperlink.
  4. System displays the registration page.
  5. Actor fills his email address, name and two matching passwords.
  6. Actor clicks the *Register* button.
  7. System creates the account in the database and redirects the actor back to the login page.

- **UC2 - Login**
  Actor: Guest
  Requirements: An account with the email and password already exists in the database.
  Sequence: This use case starts when the actor is on the initial page of the application.
  1. Actor clicks the *Login* button at the top of the page.
  2. System displays the login screen.
  3. Actor fills his email address and password.
  4. Actor clicks the *Login* button.
  5. System changes the actor's role from Guest to Member and redirects the actor back to the initial page.

- **UC3 - Logout**
  Actor: Member
  Sequence: This use starts on the initial page of the application.
  1. Actor clicks the *Logout* button at the top of the page.
  2. System changes the actor's role from Member to Guest and redirects the actor back to the initial page.

- **UC4 - Create event**
  Actor: Member
  Requirements: The date of the event is not in the past, and an event with the same name does not exist.
  Sequence: This use case starts on the initial page of the application.
  1. Actor clicks the *Events* hyperlink at the top of the page.
  2. System displays a visual representation of the created events.
  3. Actor clicks the button for the creation of a new event.
  4. System displays a form for the creation of a new event.
  5. Actor fills the name, description, and date of the event. He can also specify the maximum number of participants, maximum number of submissions per participant, and the type of assets the event will allow, or leave these to their default preset values.
  6. Actor clicks the *Create* button.
  7. System creates the event in the database and redirects the actor to the detail page of the created event.

- **UC5 - Display event details**
  Actor: Member/Guest
  Sequence: This use case starts on the initial page of the application.
  1. Actor clicks the *Events* hyperlink at the top of the page.
  2. System displays a visual representations of the created events.
  3. Actor clicks on the event he is interested in.
  4. System displays the detail page for the selected event.

- **UC6 - Edit event**
  Actor: Member
  Requirements: The actor is the creator of the event.
  Sequence: This use case starts on the detail page of the event.
  1. Actor clicks the button for the editing of the event.
  2. System displays a form containing the current details of the event.
  3. Actor changes the details to the desired state.
  4. Actor clicks the *Update* button.
  5. System updates the event in the database and redirects the actor back to the event detail page, which will contain the revised information.

- **UC7 - Delete event**
  Actor: Member
  Requirements: The actor is the creator of the event.
  Sequence: This use case starts on the detail page of the event.
  1. Actor clicks the button for the deletion of the event.
  2. System removes the event from the database, along with any connected submissions and assets, and redirects the actor to the lists of the already created events.

- **UC8 - Create submission**
  Actor: Member
  Requirements: The number of current participants is lower than the event's capacity and the number of submissions for the actor is lower than the event's maximum submission allowance.
  Sequence: This use case starts on the detail page of the event.
  1. System displays a submission form on the event detail page.
  2. Actor fills the name of the submission and uploads all assets that are marked as *required* by the event.
  3. Actor clicks the *Submit* button.
  4. System creates the submission in the database and redirects the actor back to the event detail page.

- **UC9 - Edit submission**
  Actor: Member
  Requirements: The actor is the creator of the submission.
  Sequence: This use case starts on the detail page of the event.
  1. Actor clicks the button for the editing of the submission.
  2. System displays a form containing the current details of the submission.
  3. Actor changes the details, removes unwanted assets, and uploads additional assets.
  4. Actor clicks the *Update* button.
  5. System updates the submission in the database, deletes removed assets and redirects the actor back to the event detail page.

- **UC10 - Delete submission**
  Actor: Member
  Requirements: The actor is the creator of the submission.
  Sequence: This use case starts on the detail page of the event.
  1. Actor clicks the button for the deletion of the submission.
  2. System removes the submission from the database along with any connected assets and redirects the user back to the event detail page.

- **UC11 - Verify user credentials**
  Actor: VR application
  Requirements: The provided credentials are valid.
  Sequence: This use case starts when a user attempts to login in the VR application.
  1. Actor sends an email and password to the system.
  2. System verifies the credentials and sends the identifier for that user to the actor.

- **UC12 - Get events for user**
  Actor: VR application
  Requirements: The user for that specific identifier exists.
  Sequence: This use case starts when the VR application successfully authenticates the user.
  1. Actor sends an identifier of a user to the system.
  2. System sends a list of events created by that specific user to the actor.

- **UC13 - Get data for event**
  Actor: VR application
  Requirements: The event for that specific identifier exists.
  Sequence: This use case starts when the user clicks the event initialisation button in the VR application.
  1. Actor sends an identifier of the event to the system.
  2. System sends the data of the event, along with the event's submissions data, to the actor.

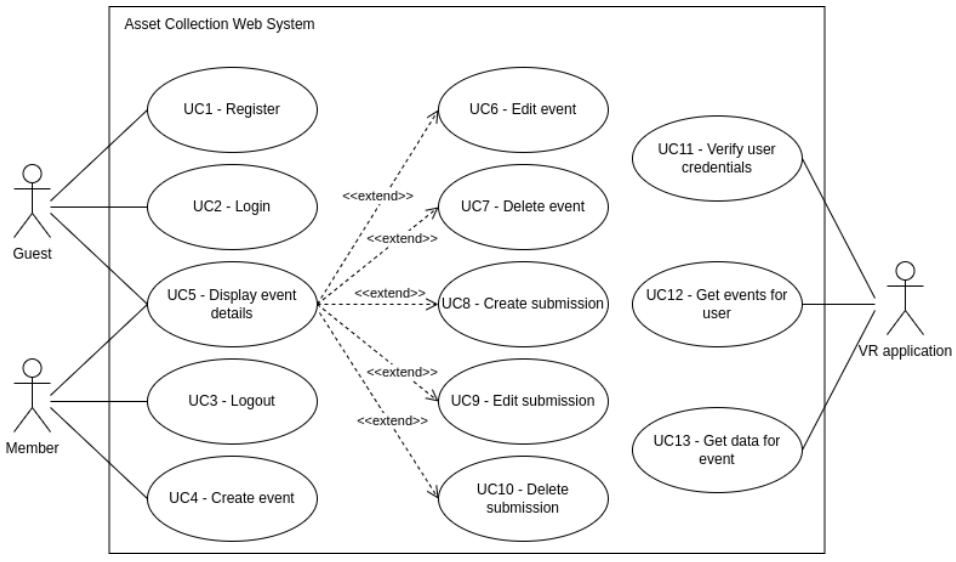


Figure 3.1: UML use case diagram

The visual representation of all use cases and their executing actors can be seen in figure 3.1, which shows the Unified Modelling Language (UML) use case diagram. From this diagram, it can be observed that all the previously defined actors were used and that all use cases are executed by at least one actor. It can further be observed that UC6 – UC10 are extensions of UC5, which the diagram represents by the use of the «extend» relation. This means that upon executing UC5, the actor can optionally continue with any of the extending use cases, provided the requirements are met.

3.1.4 Functional requirements fulfilment

The fulfilment of individual functional requirements by the use cases can be seen in table 3.1. This table, which lists use cases in rows and functional requirements in columns, shows which use cases satisfy which functional requirements. It can be concluded that all functional requirements are satisfied, because each column contains at least one check mark. Similarly, since all rows contain at least one check mark, it means that all use cases serve a purpose within the system and that none of them fall out of the previously defined scope of the system.

	FR1	FR2	FR3	FR4	FR5	FR6
UC1	✓					
UC2	✓					
UC3	✓					
UC4		✓			✓	
UC5		✓				
UC6		✓			✓	
UC7		✓				
UC8			✓	✓	✓	
UC9			✓	✓	✓	
UC10			✓			
UC11	✓					✓
UC12		✓				✓
UC13		✓	✓			✓

Table 3.1: Fulfilment of functional requirements by use cases

3.2 Design

This section discusses various design decisions, which are essential for the successful implementation of the web application. These design decisions are done while keeping technology agnosticism in mind. This way, the design produced in this section will not limit the implementation to any specific

technology and it will be possible to select the most suitable one later during the implementation process.

3.2.1 Architecture

For architecture, the three-tier architecture was selected, which is the most common when developing web applications [16]. In this architecture, the system is split into three distinct tiers, based on their functionality, which often run on separate devices. The bottom tier is called the data tier and is used to retrieve and persistently store the application data. This can be any type of relational database management system (RDBMS) such as PostgreSQL [17] and MySQL [18], or a NoSQL database server such as MongoDB [19] and Apache Cassandra [20]. The middle tier, called the application tier, contains the application's core and handles the business logic specific to that given application. In addition, it also updates data in the database by communicating with the data tier through its Application Programming Interface (API). A wide range of programming languages can be used to implement the application tier, but Python [21], Java [22], and Go [23] are among the most commonly used ones [24]. The last tier is the presentation tier, which displays the data to the user in a readable way. It is also used to gather the input from the user and delegate it to the application tier, which then performs the relevant computations. This tier is the most standardised out of the three and essentially all web applications use a combination of Hypertext Markup Language (HTML), Cascading Style Sheets (CSS) and JavaScript [25] to implement this tier.

3.2.2 Domain

Domain describes what entities the web application will contain, and what type of relations will be between them. It also describes what attributes of which type each entity contains. Based on the functional requirements FR1, FR2, and FR3, specified in the analysis subsection 3.1.1, three initial entities were defined. These are: *User*, containing the login information and references for created events and submissions. *Event*, containing basic information such as name, description, date, and references to linked submissions. And *Submission*, containing the references to uploaded assets. In addition, an entity called *File* was also defined, which holds asset related data such as its type and name. Besides these, two auxiliary entities were also added called *SubmissionFiles* and *ModelAsset*, which do not have any data on their own, and are only used to hold references to individual files. These two entities are intended to be embedded into the corresponding entities and are used to simplify the data organisation.

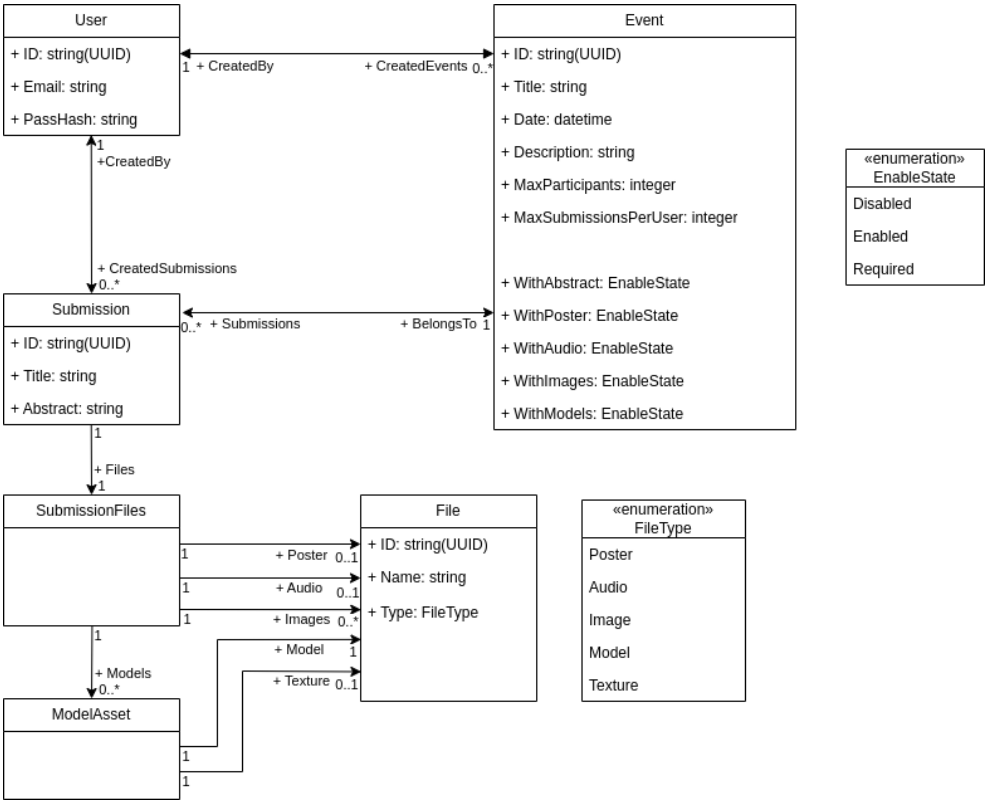


Figure 3.2: UML domain model

The final representation, shown as an UML domain model, can be seen in figure 3.2. This diagram shows the individual entities, together with the names of their attributes. Each attribute also contains a data type that will represent it in the final application. This design uses only a few data types, which are present in most programming languages and database engines. These are *strings*, which represent a text of an unspecified length, *datetime*, which describes a specific point in time, and *integers*. Furthermore, two custom enumeration data types were also defined. The first one, called *EnableState*, can be *Disabled*, *Enabled*, or *Required* and is used to specify the submission format for that given event entity. The second one, called *FileType*, can be *Poster*, *Audio*, *Image*, *Model*, or *Texture*, which corresponds to all the different asset types that the web application will accept.

Besides the names and data types of the attributes, the model also contains the information about the attribute's visibility. This is expressed by the plus sign (+) for public attributes and the minus sign (−) for private attributes. The diagram also describes the direction of the relation by using single or bi-directional arrows and the cardinality of the respective relations. In this context, cardinality refers to the number of values for that specific relation. For example, the relation between *submission* and *event* is bi-directional, with

cardinality 1 in the direction from the submission to the event. This means that for each submission, there is exactly 1 event in the submission's public attribute *BelongsTo*. Looking at the relation from the other direction, the event entity will have a public attribute called *Submissions*, which will contain any number of submissions, ranging from zero to infinity.

To represent the identifiers (IDs) of the entities, version 4 of the Universally Unique Identifier (UUID) will be used, which is a randomly generated 128-bit number expressed as a string in a predefined format. UUIDs were designed to be unique across time and space, and the standard [26] defines 5 different versions of them. These versions differ in the values that the algorithm uses for the generation of the UUID, rather than in the format of the actual identifier. For this application, the UUID of version 4 was selected, which generates the variable bits either randomly or pseudo-randomly. This means that while it is technically possible that two same UUIDs will be generated, this probability is so close to zero that it is often neglected. Also, since all the information is generated randomly, the UUID of version 4 does not hold any information about when or where it was generated, which is the characteristic of the other versions of the UUIDs. Because of this, the UUIDs of version 4 are the most commonly used, which is why this version was selected for this application.

3.2.3 User interface

Another design decision that can be made without anticipating any specific technology is the organisation of the user interface (UI). The use cases, which are intended for human users (UC1–UC10), already specify some of the required buttons and screen transitions. All this information can be collected and visualised as a screen transition diagram, which can be seen in figure 3.3. This diagram is not standardised and does not use the UML notation, but similar diagrams are often produced during the software engineering process. In this diagram, the individual screens are represented as rectangles, with arrows depicting the transitions between them. Each transition also contains a description of an event or trigger that is connected to that specific transition. The diagram should be read from the circle, labelled *Entry*, which depicts the starting point of the screen transition flow. In the diagram, there are also three rectangles that have a thicker border than the rest. These screens are special in the sense that there will be a hyperlink referring to them in the application's menu bar. This means that it will be possible to transition to them from any other page from within the application. However, to keep the diagram as simple as possible, these arrows were omitted. Another set of arrows, which were also omitted, are the return arrows from the *form* pages. Each form will feature a mechanism that will make it possible to cancel the data input and return to the invoking page. In addition, when the user provides invalid data into any form, he will be kept on the page with the form and an error message will be shown, which will describe the occurred issue.

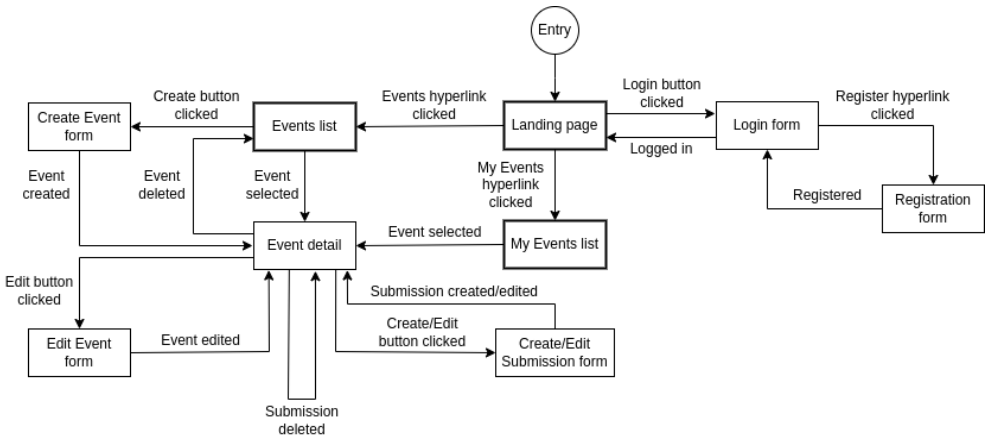


Figure 3.3: Screen transition diagram

3.3 Implementation

In this section, the technologies used for implementing the web application are described and the reasons for choosing them are briefly stated. Starting with the data tier, the implementation process is described in the order of the three-tier architecture, which should make the explanations easy to follow. After the data tier subsection, the application and presentation tier subsections will follow, where numerous backend and frontend technologies, which were used to create the final web application, are explained.

3.3.1 Data tier

For the data tier, the initial decision that had to be made was to choose either classical RDBMS or some form of a NoSQL database. The main advantage of RDBMSs is that they are well established in the industry and have been continuously developed for the past 30 years. They also feature a powerful querying mechanism called Structured Query Language (SQL), which makes it possible to get any combination of data from the database efficiently. However, the main disadvantage of RDBMSs lies in their inability to scale properly. This is caused by the necessity of having all the database's data on a single machine, meaning that if the machine's capacity runs out or it cannot keep up with the computations, the whole database needs to be moved to a new, more performant machine. This scaling approach is called *vertical scaling* or *scaling up* and is considered inferior to other approaches since there is a physical limit of how performant a single machine can be. Because of this reason, RDBMSs are not appropriate for applications where a big amount of data is expected.

The other option is a NoSQL database. The main difference between RDBMS and a NoSQL database lies in the data storing method. In RDBMS, the individual entities are stored in rows and the entities' attributes are stored

in columns. This means that there must exist a clearly defined scheme, which is usually prescribed by the stored entity, that specifies the number, names, and data types of that database's columns. For NoSQL databases, one of the most popular data storing models is a document database, which stores whole entities as strings in a format that is easily processed by machine, such as the JavaScript Object Notation (JSON) or its binary equivalent. Other popular NoSQL data storing models include key-value stores, which only store a single value for a specific key, or graph databases, which model the relations between the entities as a graph, therefore allowing the user to query the database by utilising graph algorithms. NoSQL databases also do not force the user to use a specific database scheme, so it is possible to save objects with different attributes into a single database. This makes the use of a NoSQL database much simpler, but at the cost of a slightly less powerful querying mechanism compared to SQL. However, a big advantage of NoSQL databases lies in the support of a concept called *sharding*. Sharding refers to a technique where the database's data is split across multiple devices, which are referred to as *shards*, to improve database's capacity, security, and performance. This concept allows the database to balance the load by redirecting different requests to different shards, which has a positive impact on the performance, because it is possible to fulfil multiple requests concurrently. It also gives the database the option to *scale horizontally*, such that it is possible to improve the database's performance and capacity by adding new shards. This means that NoSQL databases are often easier to use and scale, and offer different data models, which might be more appropriate for the application's use case. They, however, do not offer as powerful and universal querying methods as SQL and should be only used in applications where complex queries are not required.

MongoDB

The database engine that was selected for the application's data tier is called MongoDB [19]. MongoDB is a NoSQL document database which stores whole entities in binary JSON (BSON). The reason for choosing a NoSQL document database is that it will be easy to transfer the entities between the application and the database, since it will not require any relational mapping. MongoDB also supports sharding, which, even though it will most likely not be required, will allow the application to scale horizontally and serve multiple users concurrently in case the need to do so ever arises. In addition, MongoDB's proprietary query language allows querying the database by attributes or embedded documents, which is entirely sufficient for this web application. Also, MongoDB is one of the most popular document databases, comes with high-quality documentation, and offers official support for 12 programming languages including Java, Python and Go, which makes it ideal for almost any type of project.

MongoDB offers 3 products which are: Community Edition, Enterprise Advanced, and Atlas [19]. Community Edition can be downloaded and self-hosted for free and offers all the essential functionality that is needed from a database server. Enterprise Advance is a product aimed at larger companies, which adds features such as enterprise level engine encryption, data auditing, dedicated technical support and requires a payment that differs based on the company size and functionality requirements. MongoDB's latest product, called Atlas, features a fully managed cloud database. With Atlas, developers do not have to worry about the infrastructure on which the database server runs. They only configure the properties of the database and specify the number of shards, and then Atlas takes care of the rest. As for the pricing model, it works similarly to other cloud services, where the user pays only for the used space and the number of operations. This means that it is possible to start with minimal investment and slowly increase the resources as required. Atlas also features a free tier with 512 megabytes of storage, which can be used for smaller projects with no time limits.

Because the web application will not be data intensive, the free tier of MongoDB Atlas was selected for the data tier. This should provide the application with enough data storage for the initial stage, with the option of upgrading it to a bigger capacity tier in the future. Upon registering into Atlas, the system created 3 database shards in the *us-central1* region, and gave the user the access to the developer portal. Through this portal, it was possible to configure the database's properties, check various metrics, or set up triggers that would execute an action every time a specific condition was met. The created database was also assigned an Uniform Resource Locator (URL), which could be used to connect to it either via the command line utility or through any of the provided programming language drivers.

3.3.2 Application tier

For the application tier, the main decision lies in the programming language for the application's backend. Most present day programming languages are already mature enough and offer the tools needed for web application development in the form of networking functions within the standard library. In addition, the ecosystems of the languages often offer web development frameworks, such as Spring [27] for Java or Django [28] for Python. These frameworks can make the web development process much easier when compared to the standard library, because they can solve common issues related to web applications, such as authorisation, request routing, and HTML rendering. The languages can also differ in whether they are statically or dynamically typed. In statically typed languages, variable types are known during compile type and cannot change during the lifetime of the application. On the other hand, in dynamically typed languages, the types are checked during runtime, which might introduce some stability and performance issues into the application.

However, for this type of application, the performance difference would be so small that it would most likely not be recognisable. In the end, the programming language decision for this type of simple web application does not matter and should be solely based on the developer's preference and experience.

Go

The Go [23] programming language was used as the main language for implementing the web application's backend. This language was chosen because the author had the most experience with it and felt confident that it can be used to create the required application. Go, sometimes referred to as Golang to avoid ambiguity, is a statically typed programming language released by Google in 2009. Go was created to be fast and powerful, while being simple and easy to learn. By looking at the code, Go can appear similar to C or C++, thanks to its use of pointers and curly brackets. It also contains some quality of life improvements that are commonly seen in higher-level programming languages, such as automatic garbage collection or the use of packages. In addition, Go also features a built-in concurrency support and extensive standard library with an elaborate networking part, which makes it an ideal candidate for web applications. Currently, Go is being most often used for cloud services and DevOps applications, but web development is also a common use case.

The work on the application was started by making a new Go project and creating the domain entities that were defined in the domain model shown in figure 3.2. Go contains all the common types, such as `string` or `int`, while also allowing users to create custom types by using the `type` keyword. These custom types can either rename existing types and further extend them, or create completely new composite types by using the `struct` keyword. This feature was used to create the enumeration entity *EnableState* for which the following three values were defined: *Disabled*, *Enabled*, and *Required* by using the `var` keyword. A custom composite type called *Event* was also created, which contains all the fields defined in the domain model. To represent the date of the event, the *Time* object from the *time* standard library was used, which describes a specific point in time along with the time zone information. To represent the ID of the entity, a custom embedded object from the *repository* package was used, which is defined in the *repository* folder within the project, that creates a version 4 UUID based on the definition specified in [26]. The last step was to represent the relations to *User* and *Submission* entities. Here, while it is possible to reference the object by using a pointer value, it is desirable to avoid this approach because it makes the implementation more complex and introduces issues such as circular dependency. For this reason, only the IDs of other entities were stored inside of the objects, which were then used to load the referenced entities from the database if required. This keeps the implementation straightforward and makes it possible to reference other objects by simple types, such as `string`, or a list of `strings`, which in

Go is defined as `[]string`. The final definition of the *Event* composite type along with the *EnableState* custom type can be seen in listing 3.1.

```
import (
    "scifairvr/repository"
    "time"
)

type EnableState string

var EnableStateDisabled EnableState = "Disabled"
var EnableStateEnabled EnableState = "Enabled"
var EnableStateRequired EnableState = "Required"

type Event struct {
    repository.ID             `validate:"required,uuid"`
    Title         string      `validate:"required"`
    Date          time.Time   `validate:"required"`
    Description           string
    MaxParticipants       int `validate:"required,gte=1,lte=16"`
    MaxSubmissionsPerUser int `validate:"required,gte=1,lte=24"`

    CreatedByUUID   string     `validate:"required,uuid"`
    SubmissionsUUID []string

    WithAbstract EnableState   `validate:"required"`
    WithPoster   EnableState   `validate:"required"`
    WithAudio    EnableState   `validate:"required"`
    WithImages   EnableState   `validate:"required"`
    WithModels   EnableState   `validate:"required"`
}
```

Listing 3.1: Event structure definition

To make the entity validation process easier, the go-playground/validation library [29] was used. With this library, it is possible to define validation tags for individual attributes, which can then be checked before saving the entity into the database. The library offers a wide range of checks, such as checking for a specific string format, or comparing the attributes between themselves. In this application, mainly the *required* check was used, which makes sure that the attributes contain a non-zero value, together with the *uuid* check, which checks that the string corresponds to a valid UUID. In addition, the *gte* and *lte* checks were also used to make sure that the integer satisfies the specified upper and lower bound. By opting for this approach, the requirements for

the attributes are defined in a single location, which removes the need for any code duplication, while also making sure that the same validation rules are enforced everywhere within the application.

After creating all the entities from the domain model, the *Repository* and *Ider* interfaces were defined, which are shown in listing 3.2. The *Ider* interface is simple and prescribes a single function *GetId* that returns a `string` containing the ID of the object. *Repository* is slightly more complex and aims to provide an abstraction over the numerous possible storage types. It works with any entity that satisfies the *Ider* interface and allows it to be *Created*, *Updated* or *Deleted* within that specific storage type. In addition, it also provides three different querying functions that either return all the entities within the storage, all the entities that contain a specific value in a given field, or a single entity based on its ID. Furthermore, all functions return an `error`, which is Go's built-in type used to inform the user that the operation has failed. Go also supports returning multiple values from functions, so it is possible to return an *error* in case of a failure and a *value* in case of a success as seen in the *Get*, *GetAll* and *GetByField* functions.

```
type Ider interface {
    GetId() string
}

type Repository[T Ider] interface {
    Get(id string) (*T, error)
    GetAll() ([]*T, error)
    GetByField(field string, value string) ([]*T, error)

    Create(entity *T) error
    Update(entity *T) error
    Delete(entity *T) error
}
```

Listing 3.2: Repository interface definition

Interfaces in Go work similarly as in Java, in the sense that they are a set of function definitions that aim to achieve some common goal. They do not contain any function implementations and therefore cannot be used on their own. To use them, an implementation, in the form of an object implementing all the interface's functions, must be created first. The advantage of referring to an interface rather than to a concrete object lies in the fact that it is possible to swap the interface's implementations as required and to do so even during runtime. In the case of this application, two classes that implement the *Repository* interface were created. One is an in-memory storage, which stores all the data in the system's operating memory and does not save the

data persistently. This implementation was used in tests to make sure that the individual parts of the application work properly without the need to burden the production database. The second implementation uses MongoDB's official driver to connect to the MongoDB instance specified by the *mongoURI* environmental variable. This implementation is used in the production version of the application to save the data persistently in the MongoDB Atlas database.

Gin

While Go has an amazing standard networking library capable of producing any web application on its own, a web framework was used, because it can address the most common issues related to web application development to make the implementation easier. For this purpose, the Gin [30] web framework was selected, which is currently the most popular web framework for Go, according to GitHub repository stars. This framework provides functionality, such as request routing, HTML rendering, or middleware support, which will all be very useful during the development process and will help to limit boilerplate code. Gin also offers high-quality documentation with code examples for various use cases, and an active community, which frequently adds new functionality into the framework by creating new, optional packages.

One of the most important features of the Gin framework is the request router, implemented by the `gin.Engine` struct. With this router, it is possible to define request handlers for specific combinations of URL patterns and Hypertext Transfer Protocol (HTTP) methods. In addition, it is also possible to specify middleware functions that will get executed either before or after the handler function to provide some additional functionality. By default, Gin creates the router with two middleware functions attached. The first is a logger middleware, which outputs the details of every incoming request, informing about values such as current time, requested URL, used HTTP method, or returned HTTP status code. The second is a recovery middleware, which recovers the application in case any critical error occurs in the handler during the request processing. By using this middleware function, the application will not crash even if some internal error occurs, which will have a positive impact on the application's stability. In addition, two additional custom middleware functions were defined that are located in the *middleware* folder within the project. First is the *LogInternalError*, which specifically logs error details for any request that results in HTTP status code 500, which is used to describe internal server error. The second one is called *Sessions*, which manages the sessions of the currently active users and enables them to perform authorised operations upon logging in the application.

Once the request router and all the middleware functions were ready, the handlers for the individual combinations of URLs and HTTP methods were defined, which are based on the use cases defined in subsection 3.1.3. To make the code simple to orient in, handlers with common goals were grouped

together. By using this approach, three distinct categories emerged, which correspond to functional requirements FR1, FR2, and FR3 that were previously defined in subsection 3.1.1. To implement these categories, three custom Go objects called *UserController*, *EventController*, and *SubmissionController* were defined, which are all located in the *controllers* folder within the project. To specify the URL patterns, Gin supports a combination of static strings and dynamic variables, which are denoted by a colon sign. This allows to define patterns similar to `/events/:event_id/`, which will match URLs such as `/events/1/` and `/events/08e0cfcb-a129-423d-9a19-f6e8438021dd/` and will make it possible to obtain the provided ID in the handler by referring to the *event_id* URL parameter. As for the utilised HTTP methods, only the GET and POST methods were used, because standard HTML does not offer a simple way to send anything else. The main difference between these two methods is that GET is almost exclusively used for data querying and should not carry any additional data in the request. On the other hand, POST requests often contain data that is sent together with the request, which is then used by the application to update resources in the database. This makes the GET method ideal for listing the entities, whereas POST for creating, updating, or deleting them. A list of all the implemented handlers, along with the corresponding controller, URL pattern, HTTP method, and content type of the response can be seen below.

Handler: LoginGet
Controller: UserController
URL pattern: /login/
HTTP method: GET
Content type: HTML

Handler: RegisterGet
Controller: UserController
URL pattern: /register/
HTTP method: GET
Content type: HTML

Handler: LoginPost
Controller: UserController
URL pattern: /login/
HTTP method: POST
Content type: HTML

Handler: RegisterPost
Controller: UserController
URL pattern: /register/
HTTP method: POST
Content type: HTML

Handler: LoginAPI
Controller: UserController
URL pattern: /api/login/
HTTP method: POST
Content type: JSON

Handler: Logout
Controller: UserController
URL pattern: /logout/
HTTP method: POST
Content type: HTML

Handler: All
Controller: EventController
URL pattern: /events/
HTTP method: GET
Content type: HTML

Handler: Create
Controller: EventController
URL pattern: /events/
HTTP method: POST
Content type: HTML

Handler: My
Controller: EventController
URL pattern: /events/my/
HTTP method: GET
Content type: HTML

Handler: MyAPI
Controller: EventController
URL pattern: /api/events/my/
HTTP method: GET
Content type: HTML

Handler: Detail
Controller: EventController
URL pattern: /events/:event_id/
HTTP method: GET
Content type: HTML

Handler: Update
Controller: EventController
URL pattern: /events/:event_id/
HTTP method: POST
Content type: HTML

Handler: Delete
Controller: EventController
URL pattern: /events/:event_id/delete/
HTTP method: POST
Content type: HTML

Handler: DetailAPI
Controller: SubmissionController
URL pattern: /api/events/:event_id/submissions/
HTTP method: POST
Content type: JSON

Handler: Create
Controller: SubmissionController
URL pattern: /events/:event_id/submissions/
HTTP method: POST
Content type: HTML

Handler: Update
Controller: SubmissionController
URL pattern: /events/:event_id/submissions/:submission_id/
HTTP method: POST
Content type: HTML

Handler: Delete
Controller: SubmissionController
URL pattern: /events/:event_id/submissions/:submission_id/delete/
HTTP method: POST
Content type: HTML

Apart from the above stated handlers, the final implementation also contains some additional ones, such as one for serving static files or displaying the *Not Found* page, which were omitted to keep the list in accordance with the previously defined use cases. Each handler is implemented as a Go func-

tion that takes a single argument, which is a pointer to the `gin.Context`. Context is another very important object within the Gin framework that contains information about the currently processed request along with methods, which can format the response in various formats, such as JSON or HTML. It can also be used to obtain the parameters from the URL or parse data from the POST request's body. An example of the *Detail* handler, which is a part of the *EventController* and handles GET requests that map to the `/events/:event_id/` URL, can be seen in listing 3.3. In this handler, an event with the ID corresponding to the *event_id* URL parameter is queried from the events repository. In case no event is found, the handler calls the *NotFound* function, which returns the HTTP status code *404: Not Found*, and renders the *Not Found* page to the user. If the event is found, the handler continues by loading relevant render information, obtaining the user details from the session, and filling the event with submissions by calling the *loadSubmissionsForEvent* function. In case an error occurs during the submission loading, a human readable error page is shown to the user and the handler returns, otherwise the status code *200: OK* is returned and the event detail page is shown to the user.

```
import(
    "net/http"
    "github.com/gin-gonic/gin"
)

func (ec *EventController) Detail(c *gin.Context) {

    e, err := ec.eventCore.GetById(c.Param("event_id"))
    if err != nil {
        NotFound(c)
        return
    }

    rInfo := rInfoEventDetail(e, ec.userCore)
    usr, _ := getUserFromSession(c, ec.userCore, rInfo)
    ok, _ := loadSubmissionsForEvent(c, e, usr,
                ec.submissionCore, ec.userCore, &rInfo)
    if !ok {
        return
    }

    renderHTML(c, http.StatusOK, rInfo)
}
```

Listing 3.3: Event detail handler implementation

Docker

To simplify the development process even further, a few supporting technologies were used. First of them is Docker [31], which solves issues related to software portability. Without Docker, each machine needs to have a specific set of development tools and libraries in order for the application to run properly. These libraries often create unnecessary clutter on the host machine and may even cause severe issues, such as software version conflicts. Docker solves these problems by packaging the application along with all its dependencies into a *container*. The packaged container can then be run on any machine which has Docker installed, without needing to worry about the machine's operating system or configuration. In addition, Docker creates a new separate runtime environment for each container, which is by default isolated from the machine's devices. In this way, Docker also functions as a security mechanism because it prevents the containerised application from interfering with any running processes, filesystems or network interfaces.

To run a container, an *image* needs to be created first. Image contains the container's filesystem with the application, dependencies, and all additional binaries that are planned to be used. It also contains the container's configuration, such as the default command to run, environment values, and other metadata. There are multiple methods of defining images, but creating them via a *Dockerfile* is the most common one. Dockerfile is a plain text file that contains a list of instructions specifying how the image should be made. There is a wide range of commands that can be put into the Dockerfile, but `COPY`, for copying files, or `RUN`, for running commands, are among the most common ones. Other commands that are often used are: `WORKDIR`, which sets the working directory for subsequent commands, `EXPOSE`, which informs the container about which ports should be exposed, and `CMD`, which defines the command to be run after the container starts. Last important command, which is mandatory and has to be the first within the Dockerfile, is the `FROM` command, which specifies the base image on which the new image will be built on. While it is possible to start with an entirely new image by typing `FROM scratch`, most images will need some additional functionality and it therefore makes sense to start with some predefined base image. Popular choices for general use are *Ubuntu* or *Alpine Linux*, but a wide range of special purpose images can be found on Docker Hub [32], which is Docker's official image repository. Docker Hub contains, among others, officially maintained images for operating systems, databases, programming languages, and utility software. Upon registering, it is also possible to upload created images in order to share them with other developers.

The Dockerfile that was written to create the image for the Go application can be seen in listing 3.4. It starts by defining the *golang:1.20-alpine* as the base image, which is an image based on the *Alpine Linux* that contains a version 1.20 of the Go programming language, and by specifying the */app* folder

as the working directory. It continues by copying the *go.mod* and *go.sum* files into the image, which contain information related to used 3rd party libraries, and by downloading them to satisfy the application's dependencies. Afterwards, all other files from the local folder are copied into the image and the application is built as the *goAPP* binary. While it might be argued that the initial copying of the *go.mod* and *go.sum* files is unnecessary, since the `COPY . .` command will also copy these files, this approach was used because of caching. The way the caching works in Docker is that for every Dockerfile command, for which the resource did not change, a cached value will be used. This means that if the application's source code changes, but no 3rd party libraries are modified, the cached results of the `COPY go.mod go.sum ./` and `RUN go mod download` commands are used, which would not be possible if the `COPY` commands would not be separated. The shown Dockerfile also uses an approach called *multi-stage* build, which is another optimisation technique. Because the Go installation is needed only to build the application and not to run the resulting binary, it is possible to split the build process into multiple stages, where each stage uses a different base image. Thanks to this, it is possible to start with the *golang:1.20-alpine* image to build the application and continue with a simpler *alpine:latest* image to which the built binary is copied, along with additional static files, and then executed. By using the multi-stage build, the size of the produced image is only 28 megabytes compared to the 620 megabytes produced by the conventional build.

```
FROM golang:1.20-alpine AS build
WORKDIR /app

COPY go.mod go.sum ./
RUN go mod download
COPY . .
RUN go build -o /goAPP


FROM alpine:latest AS run
WORKDIR /

COPY --from=build /goAPP .
COPY /resources /resources
COPY /html /html

EXPOSE 80 443
CMD [ "/goAPP" ]
```

Listing 3.4: Dockerfile for the Go application

Git

The second tool which was used to help with the development process is Git [33]. Git is an open source version control system, which tracks the project's history and allows the developers to bring back past versions of all files within the project. Git does this by storing the changes of all files, within a specific editing session, into an object called *commit*. By saving only the changed lines of the affected files, commits do not require a lot of space while still allowing to revert any file to any point in time by applying the changes in the reverse order. This also means that Git works the best with plain text files, such as source code or configuration files, which are edited line by line, and does not perform well when used with binary files where individual lines do not have a meaning on their own. Git is also often used as a collaboration tool, because it allows multiple developers to edit files at the same time. This feature is, however, not realised by locking the file or by broadcasting the changes between the developers at all times, as other collaboration tools often do. This functionality is achieved by a process called *merging*, which occurs every time someone publishes a commit that edits a file that has already been edited by someone else in the meantime. During merging, the developer is presented with a list of conflicting lines, which are lines that were edited by both commits, and has to decide which lines to keep in the result. Once the merging is confirmed, the conflicting lines are appropriately updated by the newly created merge commit, which can then be distributed to the other developers.

Git, on its own, works only locally and does not deal with file distribution over the network. However, there are multiple companies that offer free private Git repositories with various limitations. For this purpose, the faculty's GitLab [34] was used, because it offers the most generous limits and is free for CTU's students. Thanks to GitLab, the project was uploaded online, which made it possible to work on it from anywhere, while also serving as an online backup tool in case of any problems. GitLab also allows creating *pipelines*, which are sequences of steps to be run on the code every time a new commit is uploaded to the repository. These sequences often contain actions such as testing the code, and subsequently building it and deploying it to the production server. Pipelines for GitLab are defined in a file called *.gitlab-ci.yml*, which is placed together with other files in the repository. The pipeline definition, which was used for this project, can be seen in listing 3.5. In this pipeline, two stages are defined: *test*, which runs the unit tests for the Go files, and *build*, which checks if the code can be successfully built. Because the pipelines are run inside of containers, they require a definition of the base image, which will be used to run the container. Here, similarly to the Dockerfile, the *golang:1.20* image is used, which will provide all the tools needed for testing and building the Go application. To keep the testing process separated from the production MongoDB Atlas database, the pipeline was set up such that it runs a local

MongoDB instance, which is then propagated into the tested application via the *mongoURI* environment variable. Apart from running the automated unit tests by using the `go test` command, the pipeline also formats the code via the `go fmt` utility and checks for any warning by using the `go vet` utility. In case both the `fmt` and `vet` utilities are satisfied and all the unit tests pass, the pipeline runs the second stage where the application is built to make sure that there are no errors which would prevent it from compiling.

```
image: golang:1.20

stages:
  - test
  - build

test:
  stage: test
  services:
    - mongo:latest
  variables:
    mongoURI: "mongodb://mongo:27017"
  script:
    - go fmt $(go list ./...)
    - go vet $(go list ./...)
    - go test -race -cover -timeout 30s $(go list ./...)

build:
  stage: build
  script:
    - mkdir -p bin/
    - go build -o bin/ ./...
```

Listing 3.5: Gitlab pipeline definition

3.3.3 Presentation tier

For the presentation tier, the technology choices were fairly clear since the set of HTML, CSS, and JavaScript [25] is currently the standard for creating user facing web interfaces. These technologies are commonly used in unison with one another, because each of them addresses a different aspect of the user interface. HTML is a markup language that defines what objects should the user's browser render on the web page. HTML does this by defining *elements*, which are semantical blocks enclosed by *tags* that are represented by angle brackets. Tags usually come in pairs, where the element's content is enclosed by the opening and closing tag, which specifies the semantic type for

that element. A simple example is the `<p>Hello World</p>` element, which defines a paragraph of text, indicated by the `p` tag, containing the `Hello World` string. Besides paragraphs, HTML contains tags that can describe text with different types of semantic meaning, such as headers, hyperlinks, ordered or unordered lists, and more. In addition to text elements, HTML can be also used to create tables, media objects, or even interactive elements, such as forms and buttons, through which the user can interact with the web page.

HTML is used to define only the content and type of the objects to be rendered on the page. It does not concern itself with the element's visual appearance and provides all the elements unstyled. To define the visual properties of elements, the CSS language is used. By using CSS, it is possible to alter the visual appearance of the HTML elements by specifying various visual attributes, such as colour, background, or border. It also allows defining of the spacing between the individual elements by setting their *margins* and *paddings*. Once a CSS style is defined, it can be applied to a group of HTML elements by the use of *selectors*. CSS defines several selectors, which can be further combined to describe any combination of elements. The most common ones can select a single element based on its unique ID, select a group of elements that share a common class, or select all elements for a specific HTML tag.

Both HTML and CSS offer various methods to achieve basic levels of interactivity. With HTML, it is possible to create forms to obtain data from the user or define buttons, which the user can use to navigate between the pages of the application. On the other hand, CSS allows defining conditional styles where elements might change their colour once the user hovers over them with the cursor, or change the element's positioning based on the user's screen size and orientation. For anything more complex than this, the JavaScript programming language has to be used. JavaScript can be used to add new HTML elements into the page or change attributes of the existing ones dynamically, without the need to refresh the page. JavaScript can also perform more advanced functions, such as sending HTTP requests to other web applications, which enable a completely new set of possibilities. Similarly to programming languages in the application tier, there currently exist many JavaScript frameworks, which provide commonly used functionality for the presentation tier and allow the developers to produce cleaner and more easy-to-read code. The most widely used JavaScript framework is React [35], which has been created by Meta, that allows the developers to create reusable JavaScript components. While it was originally thought that the application would use some JavaScript framework, it later became obvious that it would only increase the complexity of the project while providing little to no benefit. This is mainly caused by the fact that the Go programming language contains a HTML templating library, which can generate static HTML pages with variable content. Thanks to this library, the project contains almost zero JavaScript and therefore did not need to use any JavaScript framework.

Go templates

To provide the dynamic content for the HTML pages, the html/template [36] package was used, which is a part of the Go standard library. With Go templates, it is possible to write classic HTML code that is enriched by special control structures, which are enclosed by two curly brackets and are unique to the template package. The package offers some of the most common control structures known from programming languages, such as `if/else` conditions or the `range` loop, which iterates over all elements of a list. In addition, when working with an instance of a Go object, it is also possible to access its public fields, or call its methods and use the obtained values as content or attributes for HTML elements. The package also automatically escapes all values, which prevent attackers from performing code injection attacks, which is a type of attack where the attacker attempts to change the way a computer program works by injecting it with malicious code.

An example of the HTML code, which uses the Go template's special control structures, can be seen in listing 3.6. This code is a part of the *events.html* page, which is responsible for rendering a list of all the events to the user. This example uses the `range` control structure, which loops over a list of event objects that is provided to the template in the *events* variable. In the loop's body it creates a *card* element for each event, which will contain the event's title, date in a human-readable format, and shortened description, along with a hyperlink that will redirect the user to the detail page for that specific event. The rest of the presentation tier's HTML pages are done similarly, where the dynamic values are provided by the Go application to the HTML templates, which are then compiled into pure HTML by substituting the template's control structures with concrete values.

```
<div class="row">
{{ range .events }}
  <div class="col">
    <div class="card">
      <div class="card-body">
        <h5>{{ .Title }}</h5>
        <h6>{{ .DateHuman }}</h6>
        <p>{{ .DescriptionShort }}</p>
        <a href="{{ $.eventsEndpoint }}{{ .GetId }}/">Detail</a>
      </div>
    </div>
  </div>
{{ end }}
</div>
```

Listing 3.6: Go HTML template example

Bootstrap & Font Awesome

The last two technologies that were used for the presentation tier are Bootstrap [37] and Font Awesome [38], which both helped to improve the visual appearance of the application. The issue with using only the HTML and CSS is that all the elements are, by default, unstyled. This means that in order to define standardly looking elements, developers must write a lot of CSS code, which needs to be rewritten over and over across every project. Bootstrap is a CSS framework that aims to solve this issue by defining a collection of commonly used styled components. Besides components, it also defines various utility and layout styles, which can be used to organise the components into containers, rows, columns, or even grids. Bootstrap does all of this by defining a set of CSS classes, which can be applied to the HTML elements to achieve the desired look and functionality. It is also possible to apply multiple classes on a single element to achieve multiple visual or functional effects at the same time. An example of using the Bootstrap classes can be seen in the previous listing 3.6. Here, the *row* and *col* classes on the *div* HTML element define the layout of the embedded elements and the *card* and *card-body* classes are used to create a styled *card* component to achieve a nice visual design.

Font Awesome is an icon library that offers thousands of icons, which can improve the visual appearance of the website. Each icon is also available in multiple styles, which gives the developers the option to select the one which matches the overall look and feel of the application the most. Besides different styles, Font Awesome also provides different colour and size variants for each icon, and even defines a few simple animations that can be used with no additional CSS code. In the application, the Font Awesome icons were used to replace some buttons with icons to make the user interface more intuitive, and to define the website's *favicon*, which is the icon that is displayed in the browser's address bar next to the website's name.

3.4 Testing

The goal of every developer is to produce software that will satisfy users' requirements and function correctly in every situation. This goal is, however, almost impossible to achieve in reality, because it is not possible to think of everything that might happen during the code's execution. Yet, it is possible to reduce the number of errors by properly testing all aspects of the developed software. Thanks to the fact that the application uses a managed database service in the database tier, the responsibility for the database is shifted to the provider of the service, which meant that there was nothing to test. As for the presentation tier, the most severe issues are often related to the inability of the users to navigate through the website easily and perform all the required tasks successfully. This means that the target user is needed to test the presentation tier accurately. For now, the user testing of the presentation tier

is skipped, because a user testing session will occur once the whole system is finished. During this testing session, the whole system will be checked by holding a testing science fair, which will verify that all parts of the system work correctly along with the web application's presentation tier. This leaves only the application tier, which is a part that will be focused on in this section.

There are various ways of testing the source code of the application tier, with *unit tests* being the most common one. Unit tests are automated tests that verify the correct functioning of the code by separately checking each unit, which in this situation refers to functions and methods. To do so, unit tests attempt to execute every single line of the code, which can help to identify most common issues such as `null` pointer exceptions, unreachable parts of code, errors in the code's logic, and more. Once a test is defined and executed, the ratio of executed lines of code to the total number of lines of code is calculated and is presented to the user in percentages as the *code coverage*. Usually, it is desirable to strive for the highest code coverage possible, but it is often unnecessary to aim for perfect coverage as the creation of tests becomes unproportionally difficult and provides little informational value once it gets close to 100%. For non critical software, the code coverage of over 80% is considered a good goal to aim for [39]. It is, however, important to note that while higher code coverage can help to find more errors in the code, having perfect coverage does not guarantee that the code is completely free of errors. This is caused by the fact that there are still some variable parts, such as, for example, the user's input, where an unexpected and untested value can produce an error within the code.

The creation of unit tests in Go is easy thanks to the testing [40] package from the standard library and the `go test` utility that is built into the language. The testing package prescribes a specific file naming standard for test files, which must contain the *_test* suffix. In addition, the testing function that performs the unit test must be called *TestXxx*, where *Xxx* is the name of that specific test, and take exactly one argument in the form of a `testing.T` pointer. In this project, the testing files were included in the same folders as the source code files and matched their filenames such that it is obvious which source code file is tested by which test file. This means that, for example, the *controllers/events.go* source code file is tested by the *controllers/events_test.go* test file, *models/validation.go* is tested by the *models/validation_test.go* test file, and so on. An example of a simple test from the *repository/inmemory_test.go* test file can be seen in listing 3.7. This example shows the *TestInMemoryCreate* unit test, which verifies that the *Create* method of the *InMemory* object performs as expected. In this test, a new InMemory object is created, and it is made sure that it is empty and does not contain the *o1* object. Then, the o1 object is created within the InMemory object by using the Create method and the InMemory is checked again to make sure that, this time, the o1 object is present inside of it. In addition to the testing package, a 3rd party assert [41] library was used, which made it possi-

ble to define the test requirements in the form of assert statements. The assert library offers a wide range of different assertion statements to choose from, such as checking if two values are equal, checking if an array has the expected length, or checking if an object is or is not `nil`, which is Go's built-in type for undefined values that corresponds to the `null` type in other programming languages. By using the assert library, it was possible to reduce boilerplate code, which is otherwise required when working only with the testing package, and produce unit tests that are easy to read and comprehend.

```
import (
        "testing"
        "github.com/stretchr/testify/assert"
)

func TestInMemoryCreate(t *testing.T) {

        im := NewInMemory[StringIntObject]()
        assert.NotNil(t, im)
        assert.Len(t, im.objects, 0)

        x, err := im.Get(o1.GetId())
        assert.NotNil(t, err)
        assert.Nil(t, x)

        err := im.Create(o1)
        assert.Nil(t, err)
        assert.Len(t, im.objects, 1)

        x, err := im.Get(o1.GetId())
        assert.Nil(t, err)
        assert.Equal(t, o1, x)
}
```

Listing 3.7: Go TestInMemoryCreate unit test

After defining the unit tests for all the major functions, code coverage of over 80% was reached for all the packages. The accomplished coverage for the respective packages can be seen in table 3.2. This table shows that the project was able to achieve the perfect coverage of 100% for 7 out of the 10 packages, which means that each line of code out of these packages was executed at least once. Furthermore, thanks to the fact that unit tests can be executed and evaluated automatically, the GitLab repository was set such that it runs the unit tests every time a new commit is pushed into the repository, which was already shown in the pipeline definition in listing 3.5. By using

this approach, the GitLab always sends an error message whenever the newly uploaded changes to the code break some already functional behaviour. This made it possible to identify and fix these issues quickly and efficiently because it was always clear when they happened and which commit introduced them into the code.

Package	Code coverage
scifairvr	83.9%
scifairvr/controllers	84.5%
scifairvr/html	100%
scifairvr/middleware	100%
scifairvr/models	100%
scifairvr/models/event	100%
scifairvr/models/file	100%
scifairvr/models/submission	100%
scifairvr/models/user	92.9%
scifairvr/repository	100%

Table 3.2: Go packages code coverage

3.5 Deployment

With the application successfully implemented and tested, the last step was to deploy it to the cloud to make it accessible to the users and the VR application. There is currently a wide range of cloud infrastructure providers out of which the Amazon Web Services (AWS) [42], Microsoft Azure [43], and Google Cloud [44] are the biggest ones based on their market share. All three providers offer free trials during which the developer can use various products, up to certain limits, for free. Upon registering, the developers are also given some initial free credits, which are valid through the period of the trial, that can be used to test additional paid products. The application's requirement for the cloud provider was to publish a containerised web application online and to connect to it from anywhere. Since this is a common use case, all the providers offer tools that are required to achieve this goal, which all share the same functionality. Ultimately, the Google Cloud was picked thanks to its generous free tier offering. While all three providers make it possible to deploy a containerised application for free, only Google Cloud offers this option with no time limit. Both Microsoft Azure and AWS limit this offer to 12 months, after which the billing period starts no matter if the application stays within the free offering limits or not.

With Google Cloud, it was possible to create the simplest instance of a virtual machine, called *e2-micro*, which offers 0.25 core of a virtual central processing unit (CPU), 1 gigabyte of random access memory (RAM), and persistent hard drive with 30 gigabytes of storage, for free. Even though the

provided resources are relatively small, they will suffice for the initial phase of running the application. As with other cloud providers, Google Cloud makes it easy to migrate the application to a more performant variant of the virtual machine in case the demand will increase and the current machine will be unable to keep up. Also, in case any other cloud provider introduces a better deal, it is also possible to migrate the application there, thanks to the fact that the implementation does not depend on any Google Cloud specific solution.

In order to run the container with the application on the virtual machine, the Docker image had to be published first. Google Cloud offers a product, which is specifically intended for this purpose, called Artifact Registry. Artifact Registry offers 500 megabytes of image storage as a part of the free tier, which was, thanks to the fact that the image's size was reduced by using the multi-stage build, fully sufficient. Once the image was uploaded to the registry, the virtual machine was set in a way where it automatically downloaded the image and started the container every time the VM was booted on. This option was available as a part of the virtual machine creation setting, which meant that it would persist even if the VM were to be restarted or migrated. When specifying the container details, the *mongoURI* environment variable was set, to make sure that the application connects to the MongoDB Atlas database, and a mounted directory was defined to which the uploaded assets will be stored into. Once the whole configuration of the VM was defined and confirmed, Google Cloud created the virtual machine in the *us-central1* region, and gave it an ephemeral Internet Protocol (IP) address, where the VM was reachable. A screenshot of the Google Cloud console showing the running virtual machine, along with all the related actions, can be seen in figure 3.4.

With the virtual machine set up like this, it was possible to connect to the provided IP to verify that everything works properly. Yet, there was still a bit of fine tuning required to make the application production ready. The IP address, which was currently assigned to the VM, was ephemeral, which means that it could change every time the virtual machine got restarted. Luckily, Google Cloud allows the developers to change an ephemeral IP address into a static one for free, meaning that the virtual machine will always be reachable on that address even if it gets restarted or migrated to a different physical server. By using a static IP address, the application was accessible at a fixed location, but the problem still was that IP addresses are hard to remember and users are not accustomed to using them when accessing websites. For this reason, the FreeDNS [45] website was used, through which the *scifairvr.my.to* domain was registered. FreeDNS is a domain name system (DNS), which handles the translation of domain names the users write in the browser's address bar into IP addresses. With FreeDNS, it was possible to set the translation rule for the *scifairvr.my.to* domain such that each request got forwarded to the static IP address of the virtual machine running on the Google Cloud platform.

The final step was to secure the communication between the user and the

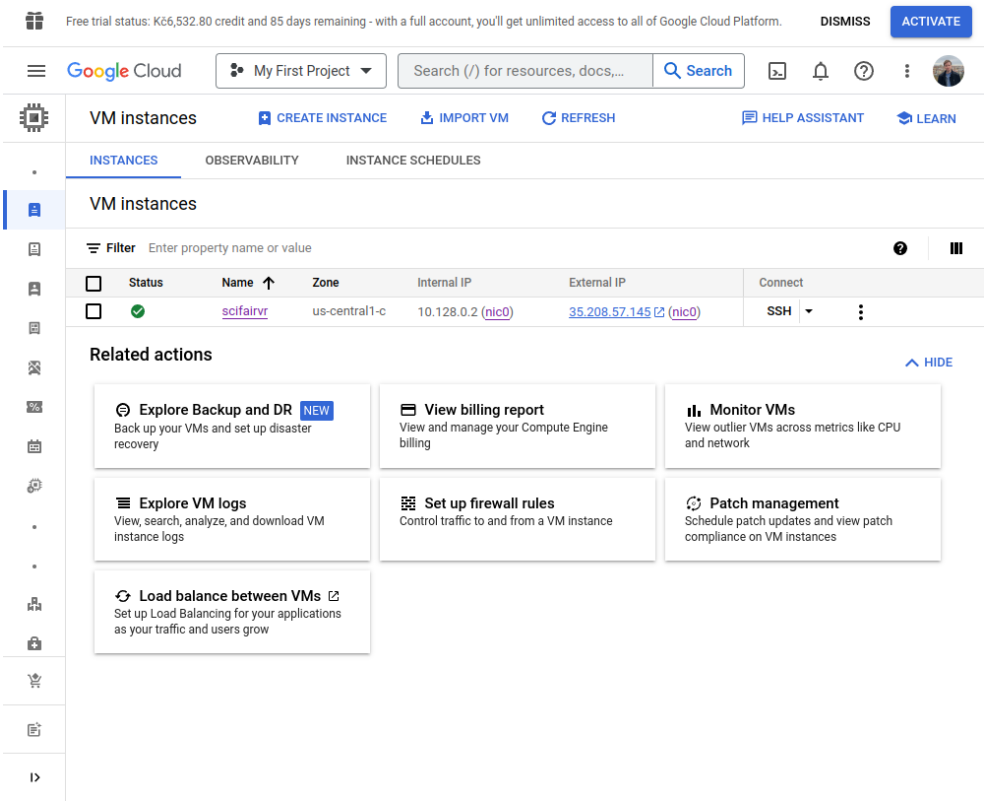


Figure 3.4: Google Cloud Console with a running virtual machine

web application. Until now, the application was using the HTTP protocol to send the data, which does not protect the communication against a potential attacker in any way. This behaviour is highly undesirable, because the application asks the users to log in to perform various operations. This would result in the users' passwords being sent as plain texts through the Internet, which would be considered a great threat from the security perspective. The solution to this problem is to use the encrypted variant of the HTTP protocol called Hypertext Transfer Protocol Secure (HTTPS). HTTPS encrypts the HTTP communication by utilising the Transfer Layer Security (TLS) protocol. In order to work properly, the TLS protocol requires a certificate that describes the identity of the website, which has to be signed by a trustworthy certificate issuer. There are currently multiple certification authorities capable of creating free TLS certificates out of which the Let's Encrypt [46] is the most popular one. By using Let's Encrypt, it was possible to issue a signed certificate for the *scifairvr.my.to* domain, which was then uploaded to the virtual machine with the application. This gave the users, as well as the virtual reality application, the ability to communicate with the web application in a secured way by using HTTPS.

With the deployment process completed and the security issue taken care of, the work on the web application was done. A screenshot of the website accessed through the HTTPS on the *scifairvr.my.to* domain can be seen in figure 3.5. This marks the first half of the automatic science fair generation system done, which leaves only the virtual reality application left to be implemented, which is further described in the following chapter.

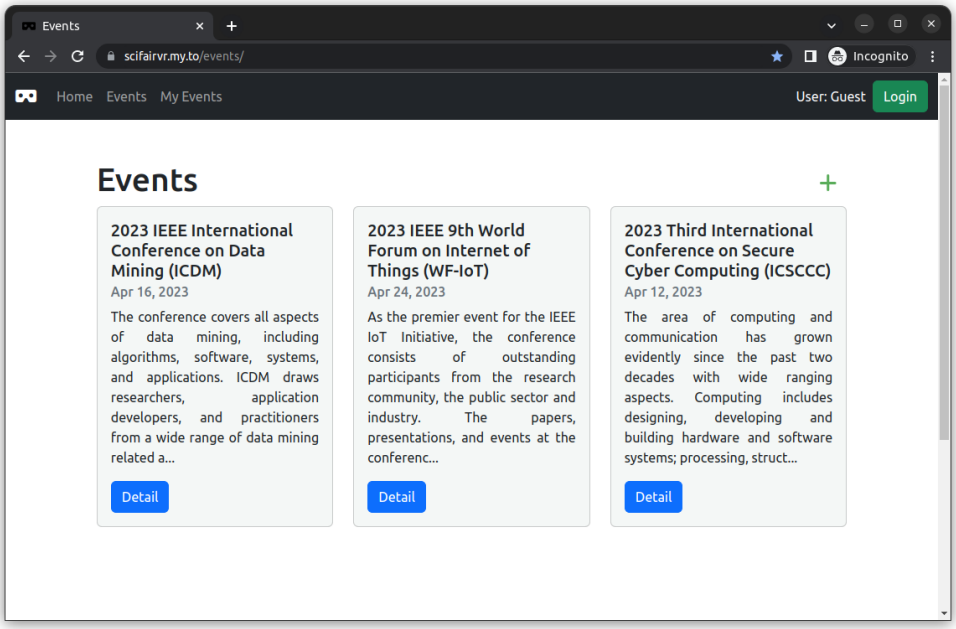


Figure 3.5: Finished web application accessed on the *scifairvr.my.to* domain

Chapter 4

Virtual reality application

In this chapter, the design and implementation process of the virtual reality application is described. This application will handle the automatic generation of the science and engineering fairs and allow the users to connect to the generated worlds to attend the fairs in virtual reality. To obtain the participants' assets to use for the world generation, the VR application will communicate with the web application, which was developed in the previous chapter, and transfer the assets into the virtual reality for the users to see and use.

Because of the fact that the development of a virtual reality application differs from that of a web application, this chapter will be structured differently. It will begin by introducing the technology that will be used throughout the whole implementation, called Neos VR [47]. This technology, which is also specified as a requirement in the thesis's assignment, will make it possible to create the virtual world, as well as program any dynamic behaviour, completely inside of the virtual reality. Following the Neos VR introduction, will be the design and implementation sections during which the system will be created and published for public use. This will finalise the automatic fair creation system and prepare it for the final chapter that will feature the testing process of the whole system.

4.1 Neos VR

Neos VR [47], sometimes referred to simply as Neos, is a social virtual reality platform with an emphasis on immersion, social interaction, and creative freedom. With Neos, users can create their own virtual worlds, fill them with numerous 3D models, create new objects with a predefined behaviour, or even program their own custom behaviour by using the built-in visual programming language. In addition, non-developer users can use Neos to explore a vast universe of different worlds created by other users, and, thanks to the integrated networking capabilities, do so together with friends. For this reason, Neos is also often used as a collaboration tool, where people can meet without wor-

rying about the physical distance between them. This makes Neos VR an ideal tool for the creation of the science and engineering fair system, since the participants will be able to connect to it from anywhere in the world by using only their computer and, optionally, a virtual reality headset.

The development in Neos is substantially different from other methods commonly used for creating virtual reality applications. Currently, the most frequently used tools when developing for virtual reality are game engines, such as Unity [48] and Unreal Engine [49]. In these tools, the developer has to implement all the general functionality, such as communication with the VR headset, user movement and physics, and networking, before he may start working on the actual virtual reality application. Also, all of this implementation is done exclusively by using the mouse and keyboard, and the virtual reality headset is used only as a display device to make sure that everything works properly. Neos VR takes a different approach to the development process, where the whole implementation occurs within the virtual reality, with an option of switching to a desktop mode if required. Neos also takes care of all the essential functionality, meaning that the developer can focus only on the important part, which is the actual development of the application. In addition, thanks to the fact that Neos is an actively developed product, developers do not have to worry about supporting the latest virtual reality equipment as this functionality is managed by the Neos development team. The final advantage of developing with Neos is that once the application is finished, it can be uploaded to the Content Hub, where other users can browse the published worlds and easily access them. This removes the need to distribute the application via the Internet, which can often be difficult because users are often cautious when downloading executable files from new, untrusted sources.

4.1.1 World creation

To start creating virtual worlds in Neos VR, the developer has to download and install it first, which can be done either through the official site [47] or by using a supported software distribution tool such as Steam [50]. Once Neos is successfully installed, the users can start creating the worlds straight away or register a user account first. With a Neos account, each user acquires access to 1 gigabyte of personal storage, which can be used to save worlds and assets in the cloud to make them accessible from anywhere.

Upon starting Neos, users are greeted with a dashboard similar to the one shown in figure 4.1, which serves as the main control element of the application. Through the dashboard, users can create new worlds, access the *Content Hub*, or get to know the controls of Neos. At the bottom, the users can access other sections of the dashboard that offer additional functionality. These include the *Worlds* section, where the user can join active worlds hosted by other users, *Inventory*, which offers a selection of tools and items created by the developers and the community, *File Browser*, which can be used to

import external files into Neos, and more. To create a new world, the user needs to press the *Create New World* button, which is located in the *Home* section on the dashboard. Upon pressing it, Neos displays the user a world creation dialog. Here, it is possible to specify the name of the world, set whether the world will be publicly available to other users or not, and choose the world's starting template, which adjusts the initial skybox and material of the ground. Once the selection is confirmed, the user is teleported into the new world, which he may start modifying by using the basic tools, by spawning items from the inventory, or by importing external models from the computer.

Figure 4.1: Neos VR dashboard

4.1.2 Basic tools

Neos offers a wide range of basic tools, called *tooltips*, that can be used to create new objects, or modify various properties of the existing ones. These tools are located in the *Essential Tools* folder within the inventory and need to be spawned and equipped before they can be used. Simple tools include brushes that can draw lines in the air, similar to real world 3D pens, or create various geometrical shapes with modifiable colour and material. An example of a more complex tool can be the *LightTip*, which creates a new point, directional or spot light of the specified intensity, or *GlueTip*, which can glue several objects into a single one to make the manipulation with the objects easier. Most of the tools also offer a *context menu* that contains

additional settings and working modes for that specific tooltip. With the help of the context menu, it is, for example, possible to specify the type of the cast shadows for the LightTip, or set whether the GlueTip will also bake the meshes of the glued objects together or not.

Another tool that is essential when creating in Neos is the *MaterialTip*, which is used to change the objects' materials. To use this tool, a material, in the form of a *material orb*, has to be loaded into the tool first. With the tool loaded with the orb, it is then possible to change the material of any object within the world by clicking on it with the trigger button of the VR headset's controller or with the left mouse button if using the desktop mode. As for the materials, the inventory within Neos is full of predefined ones that represent numerous variants of wood, ground, glass, and many more. By using the tool's context menu, it is also possible to create new physically based materials with custom colours, textures or blend modes. In addition, it is also possible to define a *normal map* for the material, which, upon applying on an object, will render the model with additional spatial details that are not present in the model's mesh. The MaterialTip, loaded with a grass material, together with a few predefined material orbs from the inventory, can be seen in figure 4.2.

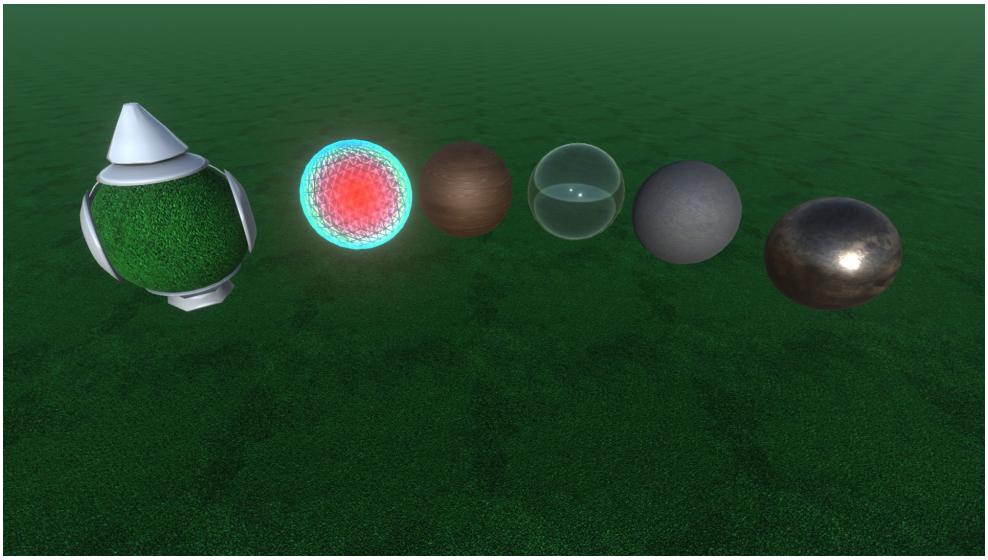


Figure 4.2: Loaded MaterialTip with additional predefined material orbs

The last tool that needs further introduction is the *DevToolTip*. This tool offers the most versatile usage by allowing the user to create any type of object in Neos. This includes objects that can be created by using specialised tools, such as lights or materials, as well as other types of objects that do not have a specific tool, such as colliders, particle effects, or UI elements. The DevToolTip can also be used to position, rotate and scale objects more precisely. In Neos, most of the objects can be picked up and dropped off

wherever the user needs them. It is also possible to rotate the object by grabbing it with the controller and rotating the hand. The issue is that neither of these methods is very precise and always modifies the object's position or rotation in all axes at once. With the DevToolTip, it is, however, possible to position, rotate, or scale the object on a single axis at a time, which results in a much finer control over the object's properties. This can be done by grabbing the axis gizmo, which is shown in figure 4.3 and appears around the object by selecting it with the DevToolTip, and moving it in the desired direction.

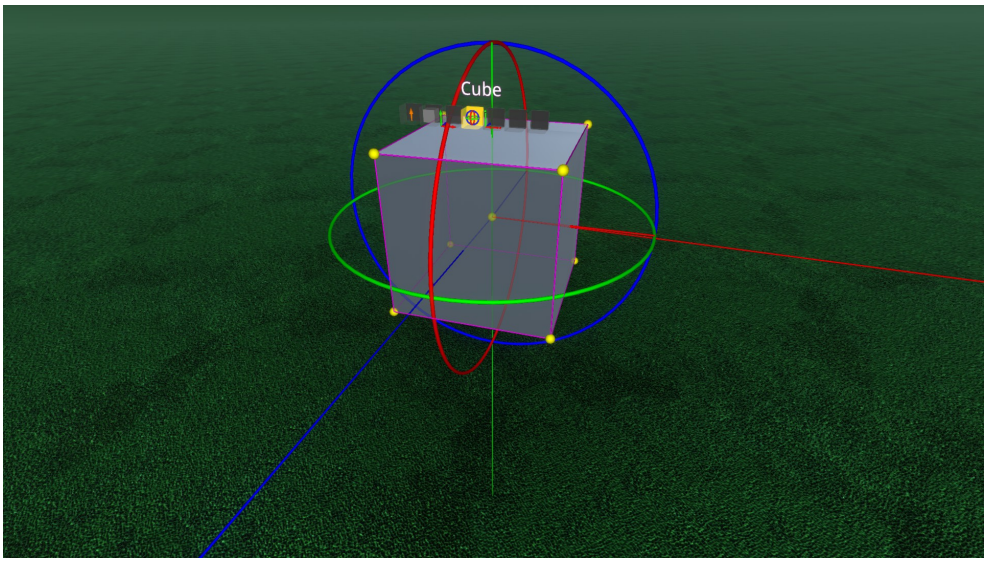


Figure 4.3: Cube object with a rotation gizmo

4.1.3 Defining object's properties

The last capability of the *DevToolTip* is to open the *Scene Inspector*, which lists all the objects inside the current world. In Neos, objects are referred to as *slots*, where each slot holds the same attributes. These attributes are *name*, *parent* slot reference, *tag*, *position*, *rotation*, *scale*, *order offset*, and information, whether the slot is *active* and *persistent* or not. The most important ones out of these are the position, rotation, and scale, which are represented by triplets, where each component specifies the property on either the X, Y, or Z axis. Neos uses a system of local coordinates, meaning the values in position, rotation, and scale are always described in relation to the parent slot. This also means that every time a transformation is applied to the parent, corresponding transformations are also applied to all of its children. This gives the user the ability to create objects that are composed of multiple parts, where it is possible to grab the parent object and move it somewhere and have the child objects follow it by keeping their relative position.

However, specifying properties such as the name and position of the slot is not enough to have anything displayed on the screen. In order for the slots to have any kind of functionality, they need to have a *component* attached to them that will handle that specific behaviour. Apart from sharing the common *persistent* and *enabled* attributes, each component contains a different set of properties which are based on the component's type. By using components, it is possible to define visual properties of the slots by specifying their meshes or materials, as well as physical properties, such as colliders or various physical interactions. In addition, there are also components that can make the slots behave dynamically by making them move or even perform animations. According to Neos Wiki [51], there are currently over 500 components, each having a different behaviour and use case, which can be used to make almost anything by adding suitable components and adjusting their properties.

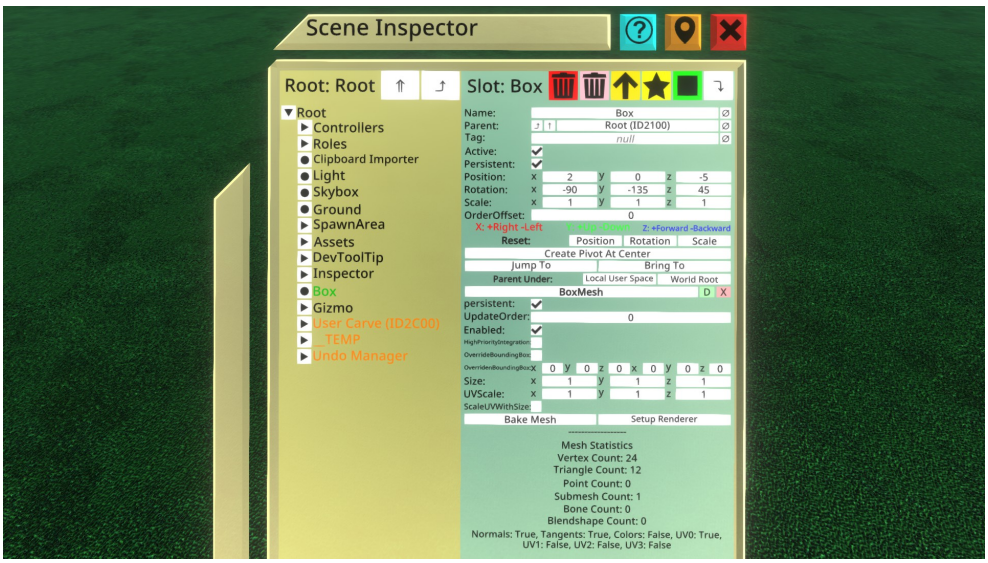


Figure 4.4: Scene Inspector

The *Scene Inspector*, which is currently displaying the details of the *Box* slot, can be seen in figure 4.4. The left portion of the inspector, indicated by the yellow background, displays all the objects inside of the current world. It can be observed that while there are objects that would be expected to be here, such as the *ground*, *skybox*, or *light*. There are also internal objects such as the *user*, *gizmo*, and even the *inspector* itself. This goes on to show that everything in Neos is a slot and is handled the same way. The left side also shows the hierarchy of the world with all the objects being parented under the *Root* slot. To change the hierarchy, it is possible to either set a reference to a new parent in the slot's *parent* attribute or to drag and drop the slot under the intended parent. The right side, indicated by the turquoise background, displays the details of the selected box slot. It includes

all the common properties from the above paragraphs, like the slot's name and position, with shortcuts for resetting the attributes or re-parenting the object under the root slot. In addition, there is also one component attached to the box slot called *BoxMesh*. BoxMesh is one of the predefined mesh components, which can be used to define any rectangular mesh by setting the *size* attribute. To edit any value in the inspector, it is possible to click on the input field to invoke a virtual keyboard and replace the current value of the attribute with the desired one. The inspector also contains additional control elements, located above the slot's name attribute, that can be used to further interact with the currently selected slot. By clicking the green square button, it is possible to create a copy of the currently selected slot, while the yellow arrow and star can be used to create an empty slot as the selected slot's parent or child, respectively. Finally, the two trash cans are used to delete the slot and differ in whether the slot's assets are preserved or not.

4.1.4 Visual programming

Sometimes the functionalities of the predefined components are, however, not fully sufficient and a custom solution is required. For these situations, Neos offers its own visual programming language called *LogiX*. In LogiX, each element of the language is represented by a block, called a *node* in Neos, with inputs and outputs of specific types. By interconnecting these with the inputs and outputs of other nodes, it is possible to create visual algorithms and programs with functionality similar to those produced by conventional programming languages. LogiX contains all the common data types, such as *booleans*, *strings*, *integers* and *floating-point numbers*, while also providing more complex ones that might not be present in other programming languages, such as *time* or *colours*. Since everything in Neos is represented in three dimensions, it is often necessary to address points in space or objects' rotations, which cannot be easily expressed by a single value. For this reason, LogiX also provides built-in support for *pairs*, *triples*, and even *quadruples* of common data types that greatly simplify the work with these kinds of values.

To use LogiX, the LogiX tooltip from the inventory is required. With the LogiX tool equipped, it is possible to open the *node browser* that offers all the LogiX nodes grouped into distinct categories by using the context menu. Out of these, the categories that include the most important nodes are *actions*, which contain nodes for writing values or references to variables, *flow*, with nodes for flow control like loops or conditional statements, and *input*, which contains data types with adjustable values to be used as inputs for other nodes. Apart from these, LogiX also contains a wide range of nodes for specific mathematical or physical computations, or nodes for working with 3D models and their textures. Some nodes also support overloading, meaning that it is possible to connect multiple different data types to the node's inputs. This concept is often utilised by nodes that accept floating-point numbers by

allowing the user to input integers as well, but there are also nodes that can change their behaviour completely based on the type of the provided input. For example, the "+" node does an addition when given a numerical input, but in the case of a string input, it performs a string concatenation. LogiX also uses a system of colour coding the data types, so it is always apparent what types are being exchanged between the nodes. This is achieved by representing each data type with a unique colour, out of which the red for strings, green for integers and light blue for floating-point numbers are the most often seen ones.

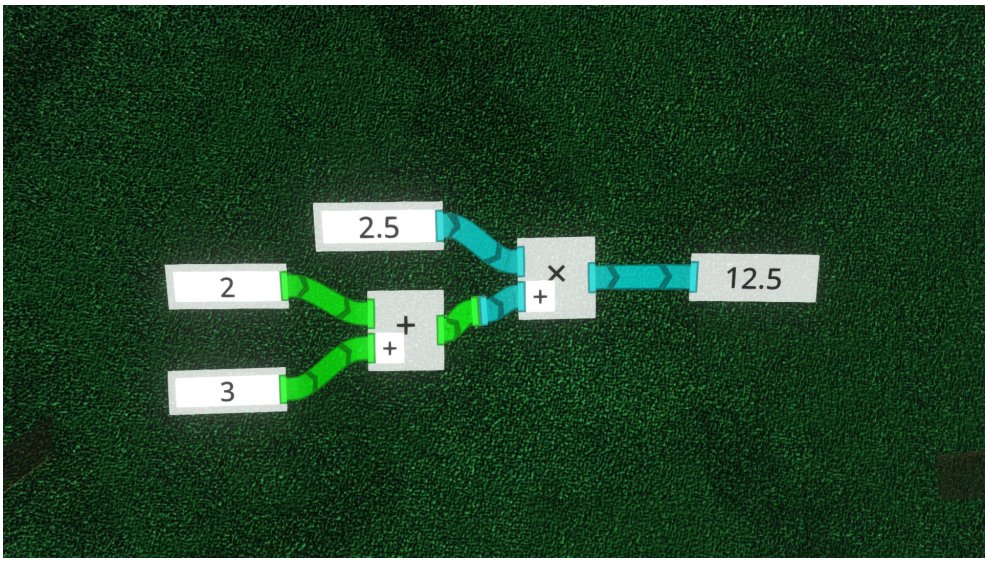


Figure 4.5: Arithmetical expression in LogiX

The LogiX tooltip can also be used to interconnect the nodes by drawing wires between the nodes' inputs and outputs. A simple LogiX program that represents the arithmetical expression $(2 + 3) \times 2.5$ can be seen in figure 4.5. This expression is read from left to right and begins by declaring two integers and adding them together by using the "+" node. The output of this operation, together with an additional floating-point number, is then used as an input for the "×" node, which performs multiplication. Finally, the output of the "×" node is displayed to the user as the result of the computation. This figure also shows an example of overloading the "+" node, which, upon creation, expects floating-point inputs. However, after being provided with an integer input, it changes into the integer variant. The "×" node also features another concept, called *type casting*, where the inputted integer is converted into a floating-point number because the node does not offer an overloaded variant for a combination of integers and floating-point numbers.

The above example, which only displays the result in the world, might be sufficient for demonstration, or people interested in visually programming

mathematical expression, but the main advantage of LogiX lies in the ability to alter the properties of objects. By using the LogiX tooltip, it is possible to extract any slot or component from the inspector and use its properties as either the inputs or outputs for the LogiX nodes. This enables the user to load the values from any slot or its components, perform calculations on them and write them back into an attribute of the same slot or a different one. This way, it is possible to define any custom behaviour of the objects without being limited by the predefined components. An example of this approach can be seen in figure 4.6. In this example, the "+" node is used to perform string concatenation on a string constant `Hello`, which is defined through a LogiX node, and the *Name* attribute of the *Box* slot. The result of this concatenation is then written into the *Text* property of the *TextRenderer* component, which renders the given string on the screen.

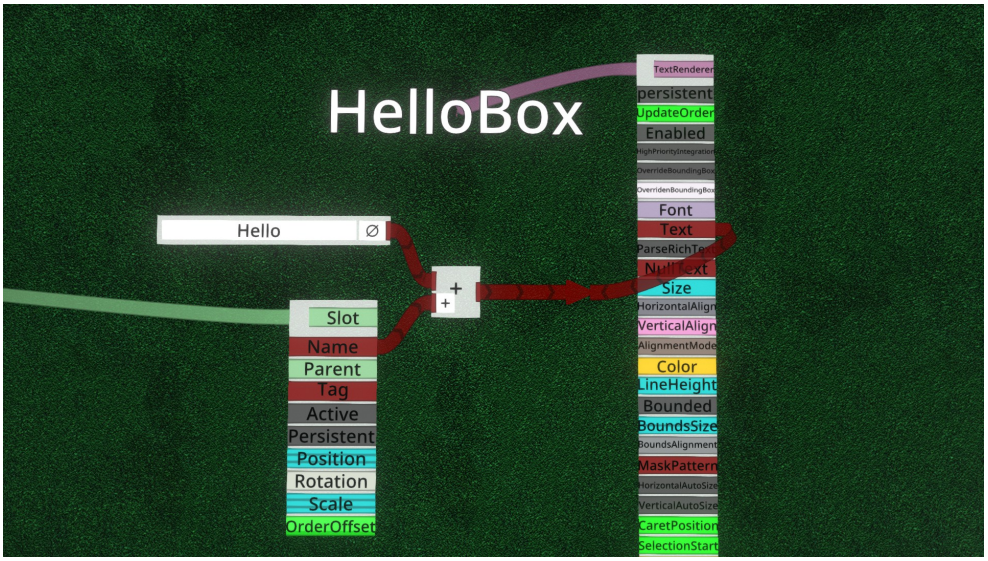


Figure 4.6: Using LogiX to modify the object's properties

In LogiX, there are two categories of nodes that differ in when they execute. The nodes that belong to the first category execute the operation given by its type all the time. This means that every time the input values change, the outputs immediately change as well to reflect the new inputs. These nodes are often easier to use, but might increase the world's complexity because a lot of unnecessary computations are executed in every frame. The previously shown examples in figures 4.5 and 4.6 both belong to this category, but they would not, thanks to their simplicity, result in a noticeable performance drop. The nodes belonging to the second category perform one time operations and need to be triggered in order for the computations to happen. To send the trigger signal, a built-in *impulse* type is used, which is always the first input value that the nodes of this category accept. In addition, these nodes also

produce an impulse as the first output value, which is triggered once the node has finished its execution, that can be used to chain multiple nodes one after another to create a sequence of actions. Because of the additional need to send the impulses to execute the actions, these nodes are often more troublesome to work with, but they offer the ability to control when and how often will the calculations trigger, which, if used correctly, can result in a much better overall performance.

The final functionality of the LogiX tooltip is to *pack* and *unpack* the nodes. Once the user has finished the creation of a LogiX program, it is no longer desirable for the nodes to be freely floating in the world. By using the LogiX tool, it is possible to hide the visual representations of the nodes, which is a process referred to as packing the nodes. During the packing, the nodes are parented under the selected slot and the nodes' visuals are deleted. This means that the packed nodes still exist within the world and execute the actions the same way as before, with the only difference of no longer being visible to the user. If, for any reason, it is needed to make the nodes appear again, the tooltip can also perform the reverse operation called unpacking, during which the nodes' visuals are brought back into the world.

4.2 Design

In this section, the virtual world is designed along with all the parts that are needed to achieve the desired functionality. For the design process, there is no need to be technology agnostic, as with the web application, since the technology for the implementation is already given by the assignment. For this reason, the world will be designed by using concepts available in Neos, which should make the subsequent implementation process go smoothly. Compared to the web application, there are also no distinct parts that could be handled completely on their own, because all the implementation will occur in a single virtual world inside of Neos. However, it is still possible to define a few elements of the world, with distinct functionalities, that can be designed and implemented independently of others.

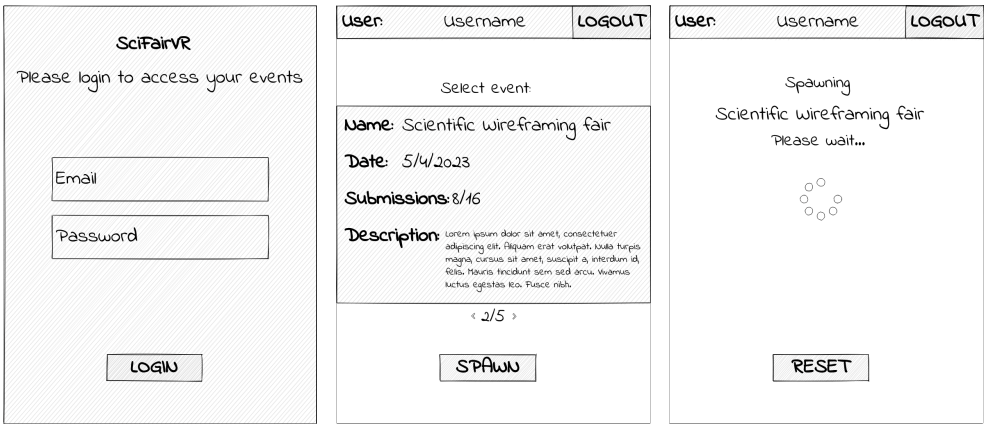
4.2.1 Environment

The environment will be created in such a way, where, upon joining the world, the participants will get the impression of attending a real science or engineering fair. Main source of the inspiration will be the ISEF's finalist hall, which was previously shown in figure 2.2, with its blue booths and carpets. The main elements of the world will be a control panel, through which the user will initialise the event defined in the web application, and participants' booths, which will showcase all the assets that were uploaded to the web application as part of the submissions. To adhere to the ISEF style, the world will contain one main aisle, which will be lined with the booths from both sides. This will

allow the participants and spectators to walk through this aisle while being able to look at the posters and interact with their creators, similar to the real event.

4.2.2 Control panel

The control panel will serve as the main element that will manage the world's content. Upon entering the world, the person who created the event in the web application will be asked to login through the control panel. Once successfully logged in, the control panel will present the user all of his created events, together with basic information such as its name, date, current number of submissions, and description. From these events, the user will be able to select the one he is interested in and click the *spawn* button to create the participant's booths, along with their respective assets. Because the creation process will not be instant, since it is necessary to transfer all the assets between the web application and Neos and position them correctly inside of the world, the panel will also contain a visual representation of the status of the ongoing spawning process.



(a) Login screen (b) Event selection screen (c) Spawning screen

Figure 4.7: Control panel wireframes

To get a better idea of how the control panel should look, wireframe designs for the individual screens were created, which can be seen in figure 4.7. These wireframes were made by the Uizard [52] wireframing tool, which is normally used to create wireframes for web and mobile applications. However, thanks to the fact that Neos offers all the UI elements commonly found in these types of applications, it was possible to create the wireframes for the control panel in the same way. The first wireframe 4.7a shows the *login* screen, where the user inputs his email and password exactly the same way as in the web application. Upon clicking the *login* button, the user's credentials will get verified by the web application and the user will be displayed the *event selection* screen

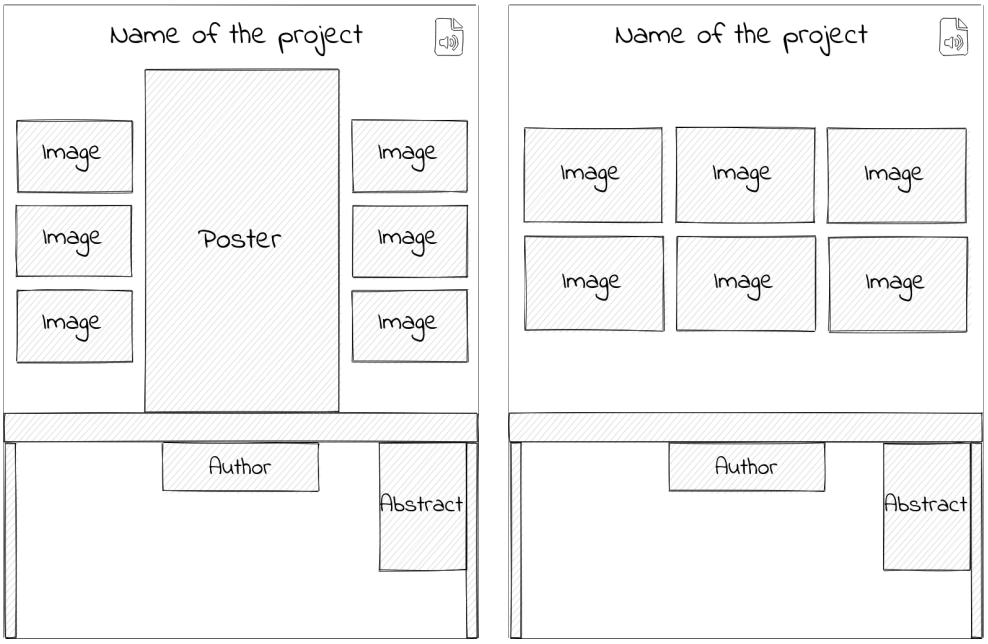
shown in wireframe 4.7b. On this screen, the user will be able to list through the events that he created via the web application and spawn the one he is interested in. Once the spawning process gets initiated by the user, the control panel will display the *spawning* screen, which is shown in wireframe 4.7c, that informs the user about the current status of the spawning process. Through this screen, it will also be possible to reset the world, which will remove all the participants' booths and redirect the user back to the event selection screen, making it possible to initialise a different event without the need to create a new instance of the world. In addition, both the event selection screen and the spawning screen will contain a user panel, which will display the name of the logged in user, and the *logout* button, which will logout the user from the control panel and take him back to the initial login screen.

4.2.3 Booths

Similarly to the environment, the booths will be modelled by adhering to the ISEF style. For each submission, the system will generate a booth with the uploaded assets, while also displaying the submission's name and author. Visually, the booth will be composed of a backplate, to which the poster and images will be attached, and a table, which will provide the authors with a place to display their uploaded models. Next to the table, there will be a small area, where the author will be able to stand in order to present his project to the other participants. In addition, it will also be possible to read the abstract or listen to the audio file with the speech, provided that it was uploaded as a part of the submission. This will enable the participants to visit the world and learn about the projects even if their respective authors will not be present in the world.

As with the control panel, wireframes of the booths were also created, which can be seen in figure 4.8. Because the web application allows to specify what type of content can be uploaded as part of the submissions, it was important to create a booth design, which would look good for all combinations of asset types. For this reason, two designs, which differ depending on whether there is a poster present in the submission or not, were created. The first wireframe shown in subfigure 4.8a places the poster on the table, while making it as big as possible, to highlight its importance. The additional images are then scaled down and arranged around the poster, because their purpose is only to accompany the poster and provide further context. On the other hand, if there is no poster, the images are scaled up and positioned into the central area of the booth, as shown in subfigure 4.8b. This type of booth will probably not be that useful in the case of science and engineering fairs, but it can be utilised to create virtual galleries or other types of presentation displays. Besides the poster and images, the booths will contain playable audio, located at the top right corner, and abstract, which will be attached to the table mimicking the appearance of an A4 paper. In case that neither of these

two will be present in the submission, their absence in the booth should not introduce any weird appearance, which is why there was no need to create a specific booth variant for them. The last type of assets, which cannot be seen in the wireframes, are the models, which will also provide an additional context for the poster. The models will be placed on the table, which is why they are not visible in the wireframes.



(a) Booth with poster (b) Booth without poster

Figure 4.8: Booth wireframes with variant with or without poster

4.3 Implementation

In this section, the implementation process of the virtual world inside of Neos is described. Apart from stating the most important aspects of the world, a few interesting implementation details are also highlighted, which might provide further insight into how the world functions. This section will contain all the elements described in the previous design section, but in a slightly different order. This change should make the text easier to understand, as each subsection will always focus on one topic without relying on concepts that have not been introduced yet.

4.3.1 JSON parser

One of the biggest challenges during the implementation was to create a completely custom JSON parsing solution. Through LogiX, it is possible to send

either the HTTP GET or POST requests, but there are currently no built-in solutions for parsing JSON or any other machine-readable format to process the server's response. Luckily, LogiX offers a wide selection of nodes for working with strings, which made it possible to create a custom implementation of JSON parsing mechanism. Thanks to these nodes, it was possible to implement functions that would return the value for a specific key, loop through the elements of a list, remove the surrounding quotation marks from a string, and a few others that were used in a range of specific use cases.

To help with the creation process, the LogiX *blueprint* utility from the public inventory was used. The blueprint is essentially a plate to which it is possible to stick LogiX nodes to keep the implementation nicely aligned. By using the blueprints, it is possible to create reusable blocks of code that can be duplicated to use the same functionality in multiple places. In addition, it is also possible to save the created blocks in the inventory for later use. There is, however, one issue connected to an extensive use of the blueprints which is that it might result in a decreased performance of the world. This is caused by the fact that blueprints themselves are objects that were created in Neos by utilising a big amount of LogiX nodes. Because of this, the blueprints were used to only create the reusable blocks, and pure LogiX was used for everything else.

An example of one of the JSON parsing LogiX blocks can be seen in figure 4.9. This example shows the *GetValueForKey* blueprint that takes the *JSON* and *key* and returns the *value* for that given key provided it exists. In this example, the blueprint takes the `{"name":"John", "age":30, "car":null}` JSON and the `age` key, and returns `30`, which is the corresponding value. This blueprint is one of the simpler ones that was created in order to parse the server's responses, but the one that was probably used the most. It works by finding the key's substring in the JSON by using the *Index Of String* node to determine the beginning of the value and then by finding the first ending symbol that is represented by a comma, square ending bracket, or curly ending bracket. Once it finds both the bounding indexes, it returns the inner substring as the result. However, because of the way the ending symbols are checked, this implementation does not work properly if the value for the queried key is a composite object or a string containing commas. To address these issues, special variants of this blueprint were created, which return the correct output, but at the cost of an increased complexity and a higher number of LogiX nodes. The reason why the GetValueForKey blueprint was not completely replaced by these "improved" blocks is, that while the original block does not return the correct output for every type of input, it behaves correctly for integers, UUIDs, and lists of non-object values. These types of values were the ones that had to be queried the most often, which is why the GetValueForKey blueprint was sufficient most of the time. Also, by opting for this approach, it was possible to limit the complexity of the implementation while keeping the number of LogiX nodes to a minimum.

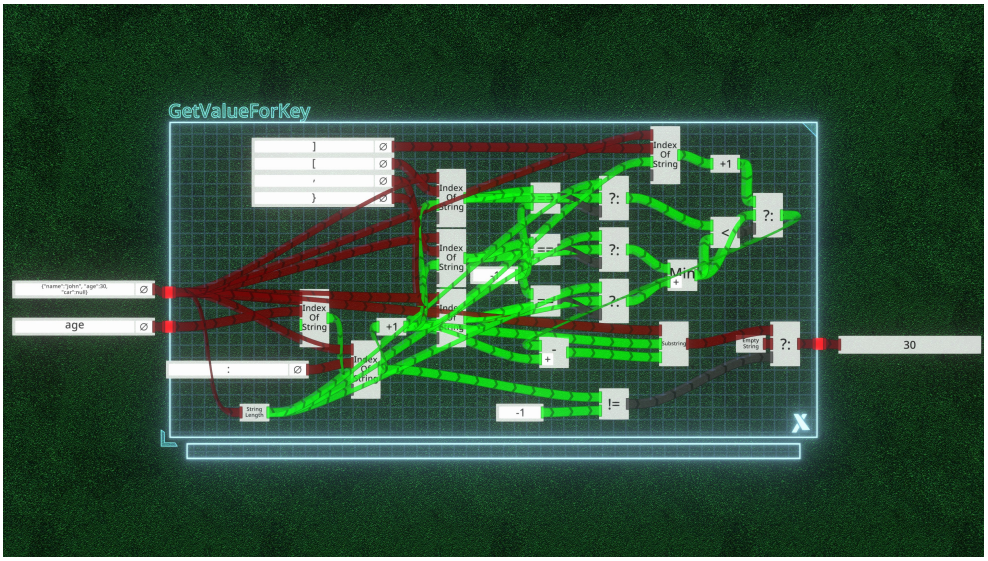
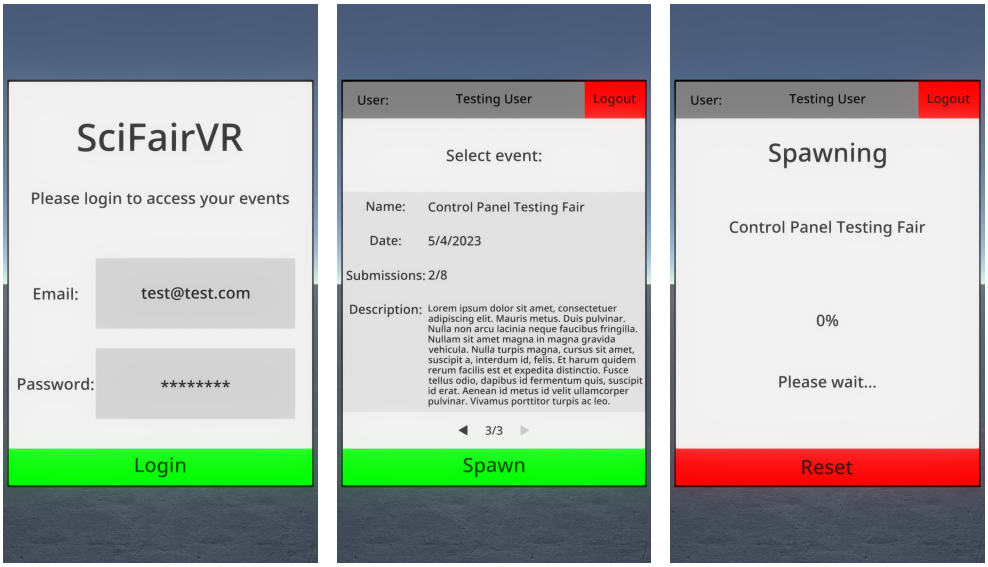


Figure 4.9: GetValueForKey LogiX blueprint

4.3.2 Control panel

With the JSON parsing blueprints created, it was possible to start making the functional elements of the world. First, the control panel was created, through which the user will be able to interact with the world, by following the wireframe designs shown previously in figure 4.7. To define the UI of the panel, the *UIX* components were used, which Neos uses to render interactive elements of the user interface, such as the content of the scene inspector or the node browser for the LogiX tool.

The most essential part of each user interface, which uses the UIX components, is the *Canvas* component that defines the dimension of the UI along with the accepted user interaction methods. After defining the main Canvas component that is shared for the whole panel, the *login*, *event selection*, and *spawning* screens were created. To define the layout of the pages, Neos offers the *HorizontalLayout* and *VerticalLayout* UIX components that specify the positions of their child objects by aligning them either horizontally or vertically. These components position the UI elements such that each occupies the same amount of space, which might not always be the desired behaviour. However, by using the *LayoutElement* component, it is possible to define the ratios of their height and width relative to one another, which makes it possible to change the sizing of the individual elements. Once the layout of the screens was created, the graphical components, which are used to render images and texts, as well as the interactive components, such as buttons and text fields, were added into the screens. After defining all the UI elements, the appearance shown in figure 4.10 was achieved, which is close to that of the wireframes from the design section.



(a) Login screen (b) Event selection screen (c) Spawning screen

Figure 4.10: Control panel

By using the UIX components, it was possible to interact with the control panel by changing the text in the input fields or clicking the buttons. Upon clicking the buttons, there was, however, no action other than a slight colour change to indicate the click. In addition, there was also no mechanism to switch between the screens of the control panel other than manually selecting the active one in the inspector. The functionality of each page was specified by creating LogiX programs, which utilised the previously created JSON parsing blueprints. This custom functionality was then connected to the control panel in such a way that it executed every time the user pressed any of the buttons.

The LogiX functionality for the *login* page shown in subfigure 4.10a is simple. Upon pressing the *login* button, the user's email and password are collected from the input fields and sent as a body of an HTTP POST request to the web application's `/api/login/` endpoint. This endpoint then verifies the received user information with the data from the database and returns either the HTTP status code *200: OK* along with user's ID encoded as JSON, or a different status code with an error that further describes the occurred issue. This means that it is sufficient to only check the response's status code and, based on it, parse either the value for the *userID* or *error* key by using the *GetValueForKey* blueprint. In case the request results in an error, the error description is displayed on the screen and the user is given a chance to fix the issue and try again. Otherwise, the parsed user's ID is saved into a variable for later use and the user is shown the event selection screen.

The functionality of the *event selection* page shown in subfigure 4.10b is the most complex out of the three screens. Its execution starts right after

the screen gets activated by the login page and begins by sending an HTTP GET request to the `/api/events/my/` endpoint with the user's ID supplied as a URL parameter. This endpoint then returns a list of all the events created by the user with the provided ID, encoded as a JSON. To represent the data of each returned event, a template UI object, which has the necessary UIX components to render the event's name, date, description, and the number of submissions, was created. In addition, a LogiX procedure that initialises this template was also created, which listens for the JSON representation of the event, from which it extracts the respective values and writes them into the UIX components to render them on the screen. This implementation is shown in figure 4.11 and can be visually split into two parts. The left portion parses the individual values from the JSON by using the previously defined *GetValueForKey* functionality and the right side transforms the parsed values into the required format and writes them into the *Text* UIX components' *Content* fields to make them appear on the screen. This event UI template is then duplicated for each event that is returned as a part of the response from the endpoint and initialised with the obtained JSON. Apart from visualising the event details, this screen also contains some additional LogiX functionalities that were implemented separately. This includes the logout process, which only clears the variable with the user's ID and changes the active screen to the login page, and the event selection control, which lists through the loaded events by clicking either the left or right arrow below the event's details. The final functionality of this screen is triggered upon pressing the *spawn* button at the bottom of the page, which initiates the spawning process for the currently selected event, while also changing the active screen to the spawning page.

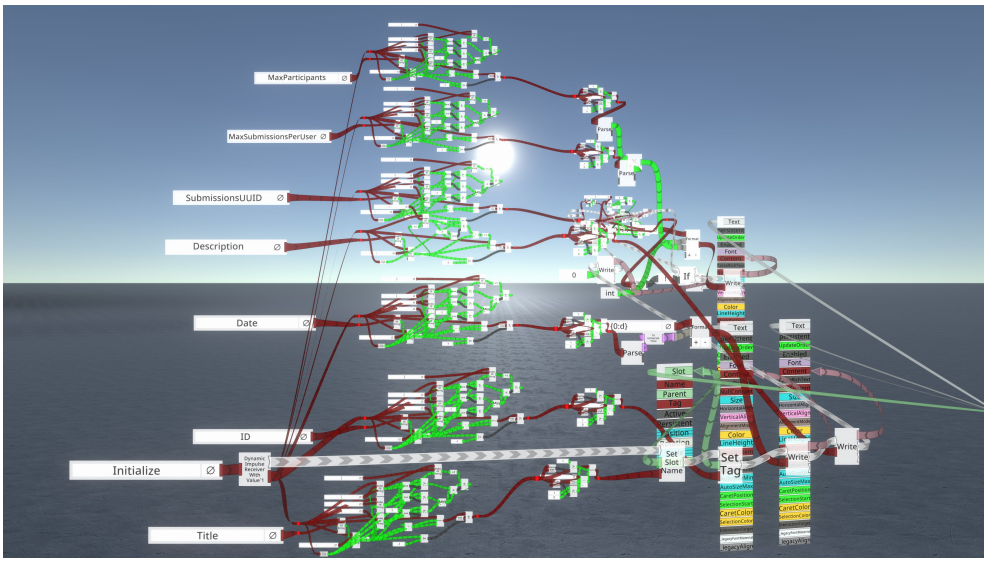


Figure 4.11: Event UI template initialisation functionality

The LogiX implementation for the *spawning* page shown in subfigure 4.10c is the least complicated out of the three. During the event's spawning process, the booths trigger two types of impulses. *AssetLoading*, which is triggered once the asset's loading process starts and *AssetLoaded*, which is triggered once the loading process finishes. To keep track of the current loading progress, the program defines two integer variables, one for each impulse type, which it increments every time it receives the respective impulse signal. Based on these two variables, it then calculates the current spawning progress by dividing the number of loaded assets by the number of all assets and expresses the result as percentages on the screen. Besides displaying the spawning status, it also allows the user to logout the same way as the event selection page and to reset the current event by clicking the *reset* button at the bottom of the screen. By resetting the event, all the event's booths are removed from the world and the active screen is changed to the event selection page, which gives the user the ability to spawn a different event without having to reset the whole world.

4.3.3 Booths

The booths represent the most essential part of the world, because they will display the assets that the participants uploaded to the web application. Similarly to the control panel, the creation of the booths began by creating the static models by following the design wireframes that were shown previously in figure 4.8. To create the models, the assets from the Neos inventory were used, together with the help of some of the basic tools. The inventory contained the 3D models of the table and curtains, which were then modified to match the ISEF's style by changing their material by using the material tooltip. Next, the developer tooltip was used to create simple cuboids, which were then used as the booth's backplate and as the dividers between the individual booths. In addition, a text object, which shows the name of the submission, and a label, which displays the author's name, were also created and placed in their respective places inside of the booth.

Once all the static parts of the booths were finished, the dynamic objects, which will visualise the assets loaded from the web application, were created. First, the poster object was defined, which is the most important out of all the assets, because it represents the foundation of the participant's project. Usually, to make image files appear in Neos, users have to find the required file through the built-in *file browser* and go through the importing procedure. Because this process contains manual steps and it cannot be automatically executed even through LogiX, a different approach had to be found. To render the images, Neos uses the *StaticTexture2D* component, which contains an *URL* attribute. This attribute usually links to the URL of the image inside of the Neos server, but it is also possible to supply a custom URL to render any image from the Internet. This made it possible to display the posters by substituting the value of the URL attribute with the address of the file from

the web application. Once the file was loaded from the web application and displayed through the StaticTexture2D component, all that was required was to define a simple procedure in LogiX that properly scaled the image while keeping its aspect ratio and placed it into the designated space inside of the booth.

With the poster template created, a similar functionality was defined for the additional images. The process of showing the images was exactly the same as with the poster, the only difference was the subsequent placement of the image objects. According to the design wireframes from figure 4.8, two types of image layouts were created that differed based on whether the submission does contain a poster or not. In case the submission does contain a poster, the images are placed alternatingly in two columns next to the poster. Otherwise, the images are placed in a grid layout in the centre of the booth to maximise the usage of space. Furthermore, a slight modification was also made to the scaling LogiX procedure, which limited the maximum size of each respective image to prevent it from expanding outside of the booth area.

In the case of the audio files, the importing suffered from the same issues as did the images. Luckily, it was also possible to solve them in a similar way. To represent the audio, the *StaticAudioClip* component was used, which contains the *URL* attribute from which it loads the audio data. As for the control of the audio clip, there was no need to implement any custom solution, because Neos comes with its own audio player. Through this player, it is possible to play, pause, or stop the audio. In addition, it also allows the users to change the volume of the currently played track or specify various looping options. Because this player contained all the required functionality, there was no need to modify its visuals or implementation in any way. The only issue that was connected with the usage of this player was that it incorrectly displayed the audio's length for audio files that were loaded from the Internet. Thankfully, Neos provides a good selection of LogiX nodes aimed at working with audio, so it was possible to create a simple LogiX program that fixed this issue and made sure that the audio's length gets displayed properly.

The final asset type were the models and their textures. The original idea was to use the *StaticMesh* component through which it is possible to load the object's mesh from a *URL*. However, this component expects a URL that points to a `.meshx` file, which is an internal type of file used by Neos to define meshes. This meant that changing the component's URL to a file from the web application would not work, because these files are in the standardised formats such as `.obj`, `.fbx`, or `.gltf`. Because of this, it was not possible to import the models automatically, since it was necessary to have Neos convert them into the internal `.meshx` representation by manually going through the import dialogue. To make this process as easy as possible for the users, a visual representation of the model files was created, which links to the assets uploaded to the web application by using the *StaticBinary* component. Upon loading, these file representations are placed on the table where they await

the participants, who can then initialise them by clicking and going through the importing dialogue. In case the user has also uploaded a texture for the model, a material tool loaded with the uploaded texture will be placed right next to the model's file. By using this material tool, the user will be able to apply the texture on the imported model by clicking on it. An example of the uploaded files and the prepared material tools can be seen in figure 4.12. This figure also shows the models that were obtained by going through the import dialogue and by using the provided material tools to apply the textures.

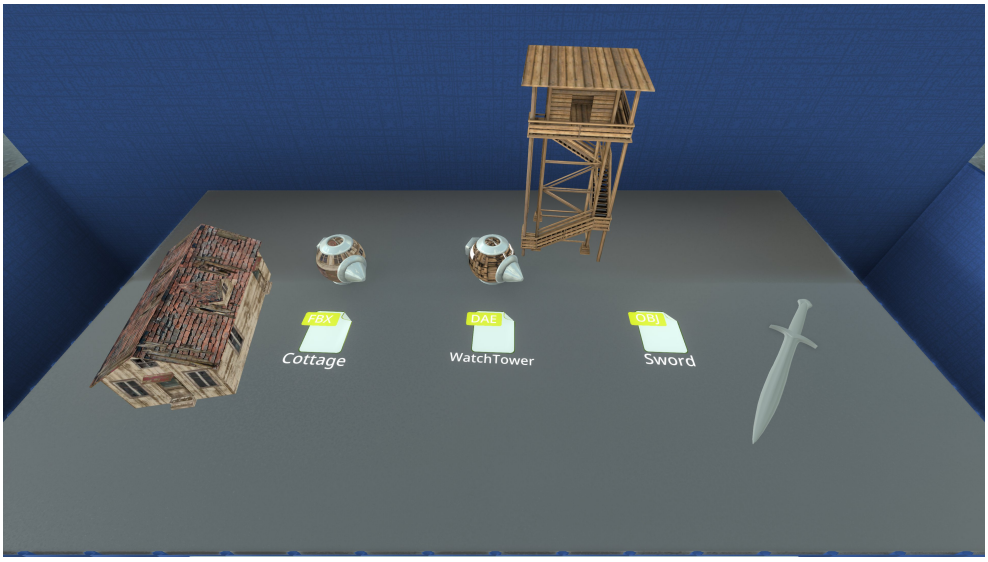
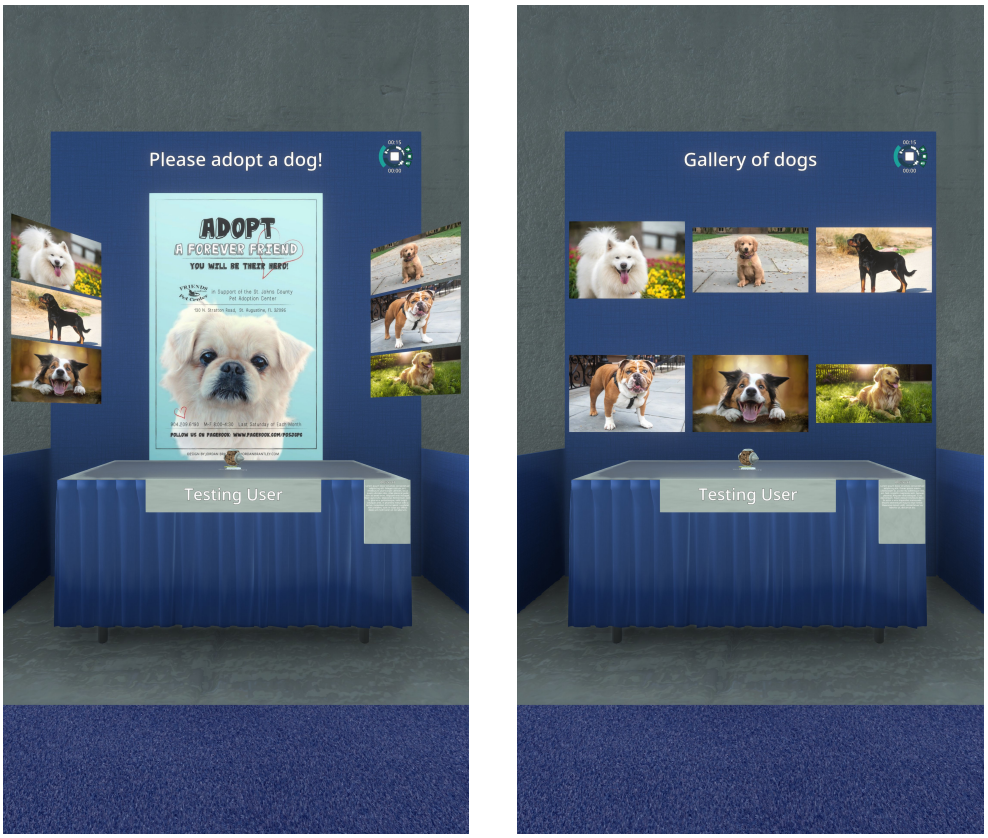


Figure 4.12: Model files and textures placed on the booth's table

After processing all the asset types, all that was left to place into the booth was the abstract. This was probably the easiest part, because the web application provides the abstract as a plain text rather than a file. Thanks to this, it was possible to supply it directly to the *TextRenderer* component, which displayed it on the screen. Visually, the abstract object was made to look similar to a paper, which was then attached to the front side of the booth's table. To make the text easily readable for the participants, the abstract object was supplied with the *Grabbable* component, which makes it possible to grab and move up close in order to read it. Furthermore, the *Snapper* and *SnapTarget* components were also used to align the abstract properly back into its original place once the user is done with the reading and places it back on the table. The finished booths can be seen in figure 4.13. The subfigure 4.13a shows the booth that contains both the poster and the images. In this type of booth, the poster is placed in the centre as the main element, whereas the images are placed around it to provide additional context. The layout of the images in this type of booth was also made to be slightly rotated towards the middle of the booth to make them easier to see while looking at the poster.

The second subfigure 4.13b shows the booth where the poster is absent and only the images are present. In this case, the images are stretched to fill the maximum amount of space, which might be useful for the creation of virtual galleries. In addition, the booths in both subfigures contain labels with the name of the project and the author, a playable audio at the top-right corner, a file for the model together with its texture placed on the table, and the abstract attached to the front corner of the table.



(a) Booth with poster (b) Booth without poster

Figure 4.13: Booths implementation with variant with or without poster

4.3.4 Environment

Once all the functional elements of the world were done, the next task was to create the world's environment. Instead of containing the world inside of some building, the world was created as an open space with sufficient lighting to make sure that the participants will be able to read all the texts with no issues. The ground was made by creating a circular plate with a concrete material on which a blue carpet was put to make it appear similar to the ISEF. At the end of the carpet, the control panel was placed and the world's spawning point

was set such that every user will face the control panel directly upon joining the world. Next, an information board was created and positioned right next to the control panel, which informs the users about the world's purpose and describes how to use it. To finish the visuals of the world, palm trees were also added to fill the empty spaces and to evoke the atmosphere of attending a real science fair somewhere in the United States. The resulting environment, shown from the bird's-eye view, can be seen in figure 4.14.

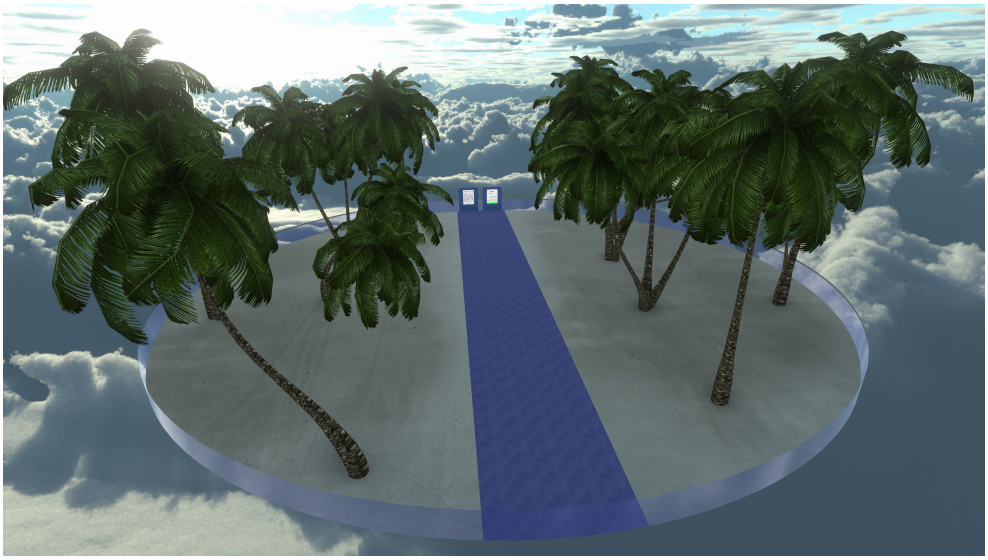


Figure 4.14: World's environment

In case some institution will want to use this system to hold their own science or engineering fair, they will probably want to organise it in their own custom environment. This environment will most likely be in the form of a 3D scan or a 360° photograph of a building or an assembly hall in which the fairs are usually held. Therefore, by keeping the world's default environment as simple as possible, it will be easy to create these personalised variations of the world, because the number of required changes will be minimal.

4.3.5 Finishing up

To finish up the implementation, all the functional elements were connected to work together. This included triggering the spawning process for the booths upon clicking the *spawn* button and subsequently removing them upon clicking the *reset* button on the control panel. Once the implementation was fully functional, the work was finalised by publishing the world through the Content Hub under the name *SciFairVR*. The world is currently freely accessible to anyone, and it is possible to open it either from the world browser option on the dashboard or to find it under the published worlds in the Content Hub.

Chapter 5

Testing the system

With the implementation of both the web application and the virtual reality application done, the final task that remains is to test the entire system. In this chapter, the whole process of creating the science fair through this automatic generation system is performed, starting by registering a user account in the web application all the way until the final initiation of the participants' booths in the virtual reality application. Throughout this chapter, it is also evaluated if the system successfully fulfils the requirements defined at the beginning of this thesis, and stated whether the system is capable of being used by organisers to host virtual science and engineering fairs with no complications.

5.1 Web application

The testing process was initiated by accessing the web application by typing the `https://scifairvr.my.to/` URL into Google Chrome's address bar. Once the page loaded, the *login* button was pressed to navigate to the *login* form and then, by clicking the *register here* hyperlink, the *registration* form was accessed. Here, the application asks the user to input an email address, name, and two identical passwords for validation purposes. To check how the application behaves if given invalid data, first an incorrect email address, followed by two passwords that did not match, was input into the form. In both cases, the application correctly recognised that the input is not valid and displayed an error message, which described the reason why the registration failed. After providing the correct information, the application redirected the user back to the login page with a success message, which stated that the user account had been created. Here, the login form was tested by inputting incorrect user data, which always resulted in a failed login request and an error notification. Only after providing the correct user information that was previously used during the registration process did the login request finish successfully and the user was brought back to the initial landing page.

Afterwards, the *events* page was opened through the *events* hyperlink from the menu bar and then the *create event* dialogue was invoked by clicking the "+" button. Here, a name, date, description, maximum number of participants, and maximum number of submissions per participant were input into the form. To make it possible to test all the asset types, the abstract, audio, images, and models submission assets were set to enabled, and the poster to required. Then, after clicking the *create* button, the event got successfully created, and the user was taken to its *detail* page. To check that the created event got correctly linked to the user account, the *my events* page was accessed, which contained the newly created event as can be seen in figure 5.1. The event creation dialogue was also tested by supplying incorrect values, like setting an event date in the past, attempting to create an event with an already existing name, or inputting a number that is out of the allowed bounds. The application had correctly rejected all these creation requests while displaying an error message similar to the ones on the login and registration pages.

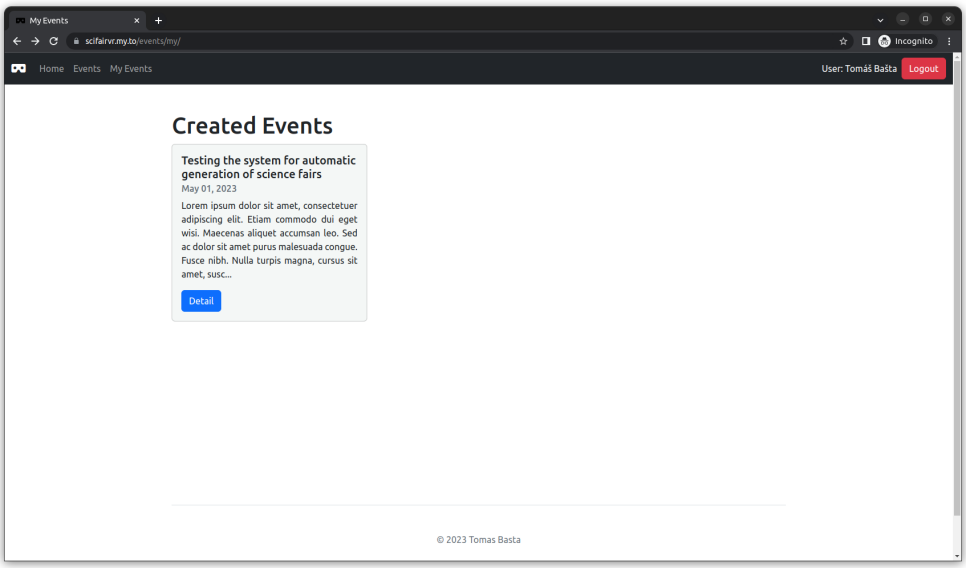


Figure 5.1: List of created events

With the event created, its detail page was accessed where it was possible to add submissions to it. Through the *submission upload* dialogue, eight different submissions were created, out of which the first two can be seen in figure 5.2. During the uploading process, the application rejected all submissions that did not contain any poster. This was correct behaviour since the poster asset type was previously set as required during the creation of the event. In order to test booths with various layouts, the submissions were created in a way where all of them contained a different combination of the asset types. As for the respective assets, files with different file extensions were uploaded

to verify that all of them will get displayed correctly inside of Neos. These file extensions included the `.jpg`, `.jpeg` and `.png` for the images, textures and posters, `.wav`, `.ogg`, and `.flac` for the audio files, and `.obj`, `.fbx`, `.dae`, and `.gltf` for the 3D models. The editing of the submission had also been tested by both adding an asset into the submission and removing an asset from the submission to make sure that it works properly.

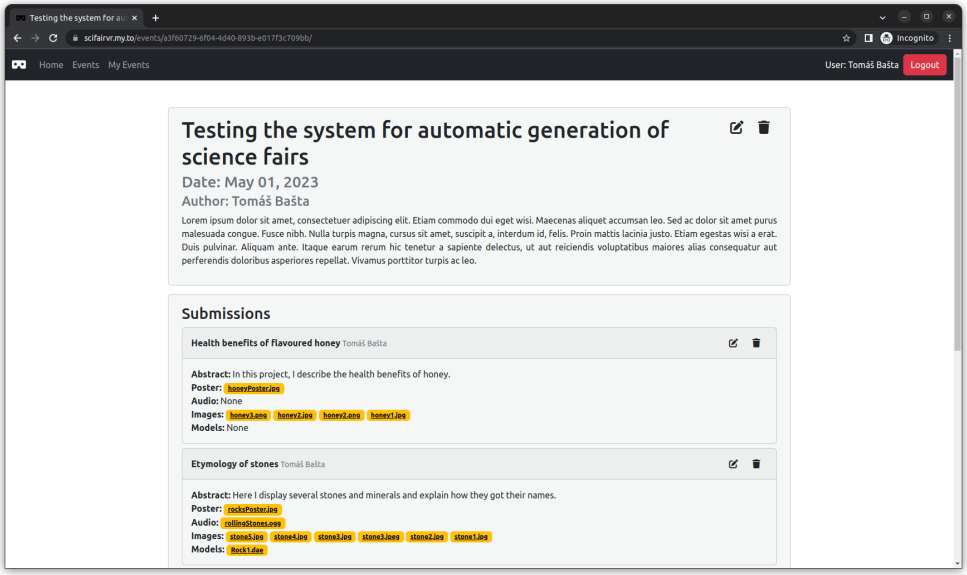
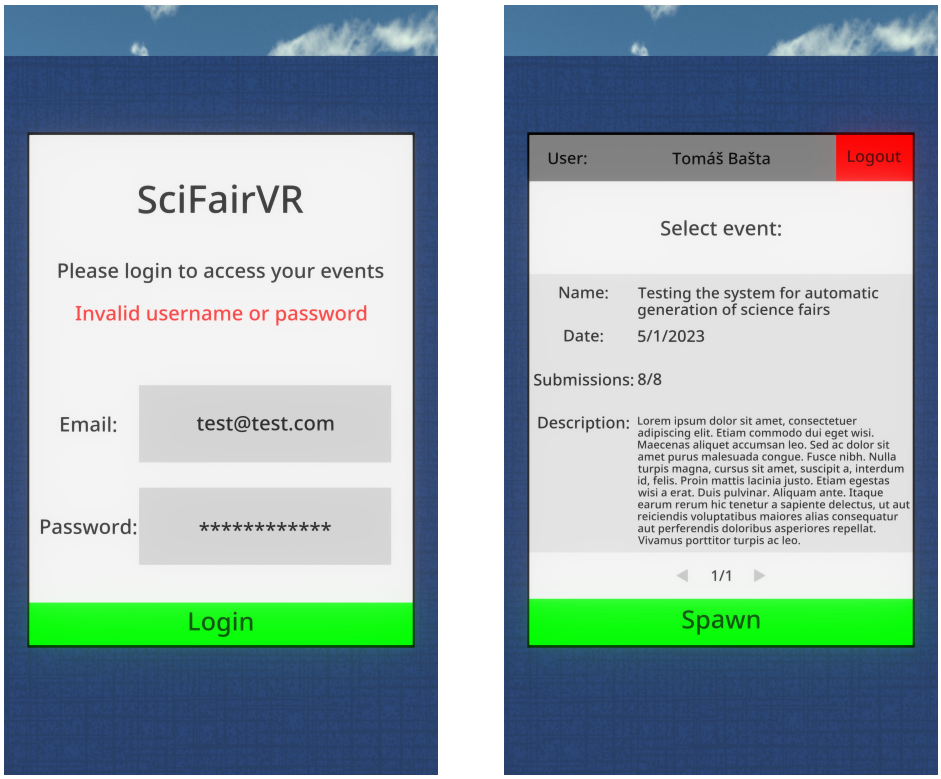


Figure 5.2: Event detail with uploaded submissions

During the testing of the web application, there were no major issues that would crash the application or otherwise prevent the user from correctly executing the use cases defined in subsection 3.1.3. The only issues that were found were related to a few hanging success and error messages, which were easily fixed once the main cause of the problem was identified. Apart from allowing the user to execute all the previously defined use cases, the application also satisfies all the functional and non-functional requirements from subsections 3.1.1 and 3.1.2. Also, to make sure that the web application's UI is intuitive, the thesis supervisor, which represents the intended user of the system, was invited to join in for a testing session. During this session, the supervisor went through the basic process of creating a user account, logging into the application, creating an event, and uploading the submissions with no complications. From this result, it can be concluded that the web application was designed and implemented properly and that it correctly fulfils its main purpose of collecting the fair-related assets.

5.2 Virtual reality application

With the web application tested, all that remained was to test the virtual reality application to make sure that the whole system works properly. The test was started by opening Neos and finding the *SciFairVR* world inside of the list of all the published worlds by typing its name into the search field. Then a private session of the world was created, and the user was, after a small loading period, transported into the world. Here, the user was spawned such that he immediately faced the instructions and the control panel, which displayed the login screen. To test the login process, several combinations of invalid user information were input into the login screen's text fields. Similar to the web application, the control panel successfully rejected all these requests and always displayed an error message that informed the user that the information is invalid. Only after inputting the data that was previously registered through the web application did the control panel accept the login request and show the user the event selection screen that contained the previously created event. The login screen, which shows a failed login request, together with the event selection screen, which shows the event created through the web application, can be seen in figure 5.3.



(a) Failed login request (b) Event from the web application

Figure 5.3: Control panel testing

With the event selection screen opened, the event was initialised by clicking the *spawn* button. Afterwards, all the eight booths appeared along the main aisle with placeholder assets. These assets were then automatically loaded and scaled one by one, which was a process that took around a minute and was visualised through the progress indicator on the control panel. Once the last asset got loaded and placed within the booth, a bell sound was played that informed the user that the spawning process had finished. Throughout this process, all the submissions' assets got loaded into the world, with the only exception of the 3D models. These were supplied into the world in the form of clickable files that were placed on the booths' tables and had to be imported before they could be used. After spawning all the models by manually going through the importing dialogues and applying textures on them with the provided material tools, the models were ready to be used. With all the assets present in the world, the event's initiation process was completed, and the world was ready to be joined by the fair's participants. The resulting world, filled with the booths that contained the submissions that were previously uploaded through the web application, can be seen in figure 5.4. From the control panel, it was then possible to remove the currently spawned event by clicking the *reset* button, which removed all the booths and set the active screen back to the event selection page. Afterwards, the last step was to log out of the control panel by clicking the *logout* button, which brought the user back to the initial login screen.

Figure 5.4: Virtual world with spawned booths

During the testing of the virtual world, one big issue was uncovered, which prevented the user from spawning events that had more than four submissions. The cause of this issue was that LogiX, which takes care of creating the booths

based on the submission data returned from the web application's API, was too slow to process all the submissions at once. The solution was to introduce an artificial delay into the algorithm, which forced it to wait half a second after the creation of every booth. This way, the program always has enough time to process each individual submission before advancing to the next one. With the issue fixed like this, it was possible to spawn the testing event that contained eight submissions, with no problems. However, to make sure that this solution truly solved the issue and did not just make it work for this particular event with eight submissions, a different testing event was created that contained the maximum number of submissions, which is currently set to 24. Thankfully the fixed implementation worked properly even for this event, which meant that no further change was necessary.

Similar to the testing of the web application, the thesis supervisor was invited to join the testing of the virtual reality application to make sure that the control panel's UI is intuitive and that the world functions correctly when used simultaneously by multiple users. During this testing, the supervisor successfully opened the world, logged into the control panel by using the previously registered user information from the web application, and initiated the spawning process of an event. During this process, a strange issue occurred where the final booth's poster loaded on only one computer, which caused the shown progress to get stuck on 75%. Only after resetting the event and spawning it again did the final poster load on both computers and the progress indication reached 100%. To find the cause of this issue, the virtual world was tested on one more computer where it exhibited a similar behaviour of getting stuck on a certain percentage. However, after giving it a few minutes, the stuck asset eventually loaded and the spawning process reached 100%. From this behaviour, it can be concluded that on some computers, the asset loading process takes longer than on others, which is most likely caused by various external factors that do not relate to Neos, such as the configuration of the device or the computer's network interface controller.

Overall, the testing of the virtual reality application can also be considered being a success. The virtual world correctly displays all the submission assets uploaded through the web application and works properly, even when used by multiple users simultaneously, even though the loading times might be longer for some users. Visually, the main aisle and the booths, which were shown in figure 5.4, look rather similar to the real world ISEF finalist hall that was shown at the beginning of the thesis in figure 2.2. Ultimately, the delivered system that is composed of both the web and virtual reality applications offers a viable alternative to currently used methods for hosting virtual science and engineering fairs. By using this system, even users that do not possess a high expertise in virtual reality can host virtual science and engineering fairs without the need to worry about importing the participants' assets. In addition, the participants will also benefit from this solution thanks to the added engagement that comes from the immersion aspect of virtual reality.

Conclusion

The goal of this thesis was to create a system for automatic creation of science and engineering fairs in virtual reality. This task was accomplished by creating two separate applications, each with a specific purpose. The first application was implemented in the form of a web application written in Go that allows the event organisers to define events through a simple web interface. Afterwards, the event's participants can use this application to upload their submissions by supplying the project's poster together with any assets that they might want to use during the project's presentation.

The second part of the system is the virtual reality application, which was realised as a virtual world in the Neos VR metaverse. This world communicates with the developed web application to load the uploaded assets, which it then uses to automatically generate the participants' booths that look similar to those commonly found in real world science and engineering fairs. The organisers and participants can then all join this prepared world and start presenting the projects without having to worry about manually transferring all the assets into the virtual reality.

The resulting system offers a fully functional solution for hosting remote science and engineering fairs that does not require the event organisers to be experienced users of virtual reality. Yet, by using the system, virtual reality will help the participants to focus more easily and give them the ability to express themselves better when compared to the conventional methods of presenting the projects through conference calls.

Currently, both the web application and the virtual world inside of Neos VR is freely available to anyone wanting to try the system, with no limitations. Going forward, the system is intended to be used for organising the science and engineering fairs that are a part of the *Scientific thinking* subject taught here at CTU. Furthermore, it is planned to offer the system to various Czech and international institutions that organise student fairs, to give it an additional use besides its utilisation at the university.

Bibliography

1. KOOMEN, Michele Hollingsworth; RODRIGUEZ, Elizabeth; HOFFMAN, Alissa; PETERSEN, Cindy; OBERHAUSER, Karen. Authentic science with citizen science and student-driven science fair projects. *Science Education*. 2018, vol. 102, no. 3, pp. 593–644. ISSN 0036-8326. Available from DOI: 10.1002/sce.21335.
2. MCCOMAS, William F. Science fair. *The Science Teacher*. 2011, vol. 78, no. 8, pp. 34–38. ISSN 0036-8555.
3. *Student projects wow at STEM fair* [online]. Farmington (Utah) [visited on 2023-03-16]. Available from: https://www.davis.k12.ut.us/district/district-news/~board/news-stories-2022/post/student-projects-wow-at-stem-fair.
4. *Regeneron ISEF* [online]. Washington, D.C.: Society for Science & the Public, (c)2023 [visited on 2023-03-19]. Available from: https://www.societyforscience.org/isef/.
5. *EUCYS 2023: European Contest for Young Scientists — 12-17 September 2023, Square, Brussels* [online]. (c)2023 [visited on 2023-03-20]. Available from: https://eucys2023.eu/.
6. GRINNELL, Frederick. Reinventing Science Fairs. *Issues in Science and Technology*. 2020, vol. 36, pp. 23–25. ISSN 1938-1557.
7. TERZIAN, Sevan G. The 1939-1940 New York World's Fair and the transformation of the American science extracurriculum. *Science Education*. 2009, vol. 93, no. 5, pp. 892–914. ISSN 0036-8326. Available from DOI: 10.1002/sce.20329.
8. *Society for Science* [online]. Washington, D.C.: Society for Science & the Public, (c)2023 [visited on 2023-03-18]. Available from: https://www.societyforscience.org/.

9. NGSS LEAD STATES. *The Next Generation Science Standards: Executive Summary* [online]. Washington, D.C. [visited on 2023-03-19]. Available from: http://www.nextgenscience.org/sites/default/files/Final%20Release%20NGSS%20Front%20Matter%20-%206.17.13%20Update_0.pdf.

10. *Intel International Science and Engineering Fair 2013 Program*. Washington, D.C.: Society for Science & the Public, 2013.

11. *Hosting a Virtual Science Fair during COVID-19* [online]. Newton (Illinois) [visited on 2023-03-20]. Available from: https://classroomcompletepress.com/blogs/community-buzz/hosting-a-virtual-science-fair-during-covid-19.

12. SHEPHERD, Marshall. *The Coronavirus Pandemic Forces Virtual Science Fairs - And That's Okay* [online]. New York: Forbes Media LLC., (c)2023 [visited on 2023-03-20]. Available from: https://www.forbes.com/sites/marshallshepherd/2021/01/27/the-coronavirus-pandemic-forces-virtual-science-fairs-and-thats-okay.

13. *ProjectBoard* [online]. Toronto, (c)2023 [visited on 2023-03-20]. Available from: https://projectboard.world/.

14. *Edmonton 2023: Youth Science Canada — Youth Science Canada* [online]. Youth Science Canada, (c)2023 [visited on 2023-03-20]. Available from: https://youthscience.ca/science-fairs/cwsf/edmonton-2023/.

15. *AJAS - National Association of Academies of Science* [online]. National Association of Academies of Science, (c)2023 [visited on 2023-03-20]. Available from: https://www.academiesofscience.org/ajas.php.

16. *What is three-tier architecture?* [Online] [visited on 2023-03-26]. Available from: https://www.ibm.com/topics/three-tier-architecture.

17. *PostgreSQL* [online]. The PostgreSQL Global Development Group, (c)1996-2023 [visited on 2023-03-26]. Available from: https://www.postgresql.org/.

18. *MySQL* [online]. Oracle, (c)2023 [visited on 2023-03-26]. Available from: https://www.mysql.com/.

19. *MongoDB: The Developer Data Platform* [online]. MongoDB, Inc., (c)2023 [visited on 2023-03-26]. Available from: https://www.mongodb.com/.

20. *Apache Cassandra* [online]. The Apache Software Foundation, (c)2009-2023 [visited on 2023-03-26]. Available from: https://cassandra.apache.org/_/index.html.

21. *Python* [online]. Python Software Foundation, (c)2001-2023 [visited on 2023-03-26]. Available from: https://www.python.org/.

22. *Java* [online]. Oracle, (c)2022 [visited on 2023-03-26]. Available from: https://www.java.com/en/.

23. *The Go Programming Language* [online]. Google [visited on 2023-03-26]. Available from: https://go.dev/.

24. *11 Most In-Demand Programming Languages* [online] [visited on 2023-03-26]. Available from: https://bootcamp.berkeley.edu/blog/most-in-demand-programming-languages/.

25. *Web Design and Applications* [online]. W3C, (c)2016 [visited on 2023-03-26]. Available from: https://www.w3.org/standards/webdesign/.

26. LEACH, Paul J.; SALZ, Rich; MEALLING, Michael H. *A Universally Unique IDentifier (UUID) URN Namespace* [RFC 4122]. RFC Editor, 2005. Request for Comments, no. 4122. Available from DOI: 10.17487/RFC4122.

27. *Spring* [online]. VMware, Inc., (c)2023 [visited on 2023-03-29]. Available from: https://spring.io/.

28. *Django: The web framework for perfectionists with deadlines* [online]. Django Software Foundation, (c)2005-2023 [visited on 2023-03-29]. Available from: https://www.djangoproject.com/.

29. *GitHub: go-playground/validator* [online]. GitHub, Inc., (c)2023 [visited on 2023-03-30]. Available from: https://github.com/go-playground/validator.

30. *Gin Web Framework* [online]. Gin Team, (c)2022 [visited on 2023-03-31]. Available from: https://gin-gonic.com/.

31. *Docker: Accelerated, Containerized Application Development* [online]. Docker Inc., (c)2023 [visited on 2023-04-04]. Available from: https://www.docker.com/.

32. *Docker Hub Container Image Library* [online]. Docker Inc., (c)2023 [visited on 2023-04-05]. Available from: https://hub.docker.com/.

33. *Git* [online] [visited on 2023-04-06]. Available from: https://git-scm.com/.

34. *GitLab FIT* [online]. GitLab B.V., (c)2023 [visited on 2023-04-06]. Available from: https://gitlab.fit.cvut.cz/.

35. *React* [online]. Meta Open Source, (c)2023 [visited on 2023-04-07]. Available from: https://react.dev/.

36. *Go Packages: template package* [online]. Google [visited on 2023-04-07]. Available from: https://pkg.go.dev/html/template.

37. *Bootstrap: The most popular HTML, CSS, and JS library in the world* [online]. Bootstrap team [visited on 2023-04-07]. Available from: https://getbootstrap.com/.

38. *Font Awesome* [online]. Fonticons, Inc. [visited on 2023-04-07]. Available from: https://fontawesome.com/.

39. *What is code coverage?* [Online]. Atlassian, (c)2023 [visited on 2023-04-09]. Available from: https://www.atlassian.com/continuous-delivery/software-testing/code-coverage.

40. *Go Packages: testing package* [online]. Google [visited on 2023-04-09]. Available from: https://pkg.go.dev/testing.

41. *Go Packages: assert package* [online]. Google [visited on 2023-04-09]. Available from: https://pkg.go.dev/github.com/stretchr/testify/assert.

42. *Cloud Computing Services: Amazon Web Services (AWS)* [online]. Amazon Web Services, Inc., (c)2023 [visited on 2023-04-09]. Available from: https://aws.amazon.com/.

43. *Cloud Computing Services: Microsoft Azure* [online]. Microsoft, (c)2023 [visited on 2023-04-09]. Available from: https://azure.microsoft.com/.

44. *Cloud Computing Services: Google Cloud* [online]. Google [visited on 2023-04-09]. Available from: https://cloud.google.com/.

45. *FreeDNS: Static DNS subdomain and domain hosting* [online]. Joshua Anderson, (c)2001-2023 [visited on 2023-04-10]. Available from: https://freedns.afraid.org/.

46. *Let's Encrypt* [online]. San Francisco: Internet Security Research Group (ISRG) [visited on 2023-04-10]. Available from: https://letsencrypt.org/.

47. *Neos Metaverse* [online]. Solirax CoreDev s.r.o [visited on 2023-04-12]. Available from: https://neos.com/.

48. *Unity Real-Time Development Platform: 3D, 2D, VR & AR Engine* [online]. Unity Technologies, (c)2023 [visited on 2023-04-12]. Available from: https://unity.com/.

49. *Unreal Engine: The most powerful real-time 3D creation tool* [online]. Epic Games, Inc., (c)2004-2023 [visited on 2023-04-12]. Available from: https://www.unrealengine.com/.

50. *Steam* [online]. Valve Corporation, (c)2023 [visited on 2023-04-12]. Available from: https://store.steampowered.com/.

51. *Neos Wiki: Components* [online] [visited on 2023-04-13]. Available from: https://wiki.neos.com/index.php?title=Category:Components.

52. *Uizard: App, Web, & UI Design Made Easy* [online]. Uizard Technologies, (c)2023 [visited on 2023-04-19]. Available from: https://uizard.io/.

Appendix A

Acronyms

3D 3-Dimensional

AJAS American Junior Academy of Science

API Application Programming Interface

AWS Amazon Web Services

BSON Binary JSON

CPU Central Processing Unit

CSS Cascading Style Sheets

CWSF Canada-Wide Science Fair

DNS Domain Name System

EUCYS European Union Contest for Young Scientists

FR Functional Requirement

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

ID Identifier

ISEF International Science and Engineering Fair

JSON JavaScript Object Notation

NFR Non-Functional Requirement

RAM Random Access Memory

RDBMS Relational Database Management System

SQL Structured Query Language

STEM Science, Technology, Engineering, and Mathematics

STS Science Talent Search

TLS Transfer Layer Security

UC Use Case

UI User Interface

UML Unified Modelling Language

URL Uniform Resource Locator

US United States

UUID Universally Unique Identifier

VM Virtual Machine

VR Virtual Reality

Appendix B

Contents of enclosed CD

- `readme.txt` ..... the description of the CD contents
- `src` ..... the directory of source codes
  - `web` ..... the directory of web application source codes
  - `thesis` ..... the directory of thesis source codes
- `text` ..... the directory of text files
  - `DP_Bašta_Tomáš_2023.pdf` ..... the thesis text in PDF format