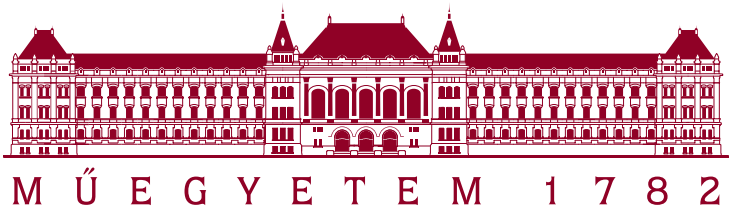


BUDAPEST UNIVERSITY OF TECHNOLOGY AND ECONOMICS
FACULTY OF MECHANICAL ENGINEERING
DEPARTMENT OF APPLIED MECHANICS

Mór Sas

BSc Thesis

Calculation of deformation of truss structures in VR enviornment

Supervisor:
Dr. Attila Kossa
associate professor

Budapest, 2024

Budapesti Műszaki és Gazdaságtudományi Egyetem
Gépészmérnöki Kar
Műszaki Mechanikai Tanszék
https://www.mm.bme.hu

SZAKDOLGOZAT-FELADAT

NYILVÁNOS

AZONOSÍTÁS

Név: **Sas Mór**		Azonosító: **72169632209**
Képzéskód: 2N-AM0	Specializáció kódja: 2N-AM0-GM-2017	Feladatkiírás azonosítója: GEMM:2025-1:2N-AM0:BHS85M
Szak: Mechatronikai mérnöki alapszak (BSc)		
Szakdolgozatot kiadó tanszék: Műszaki Mechanikai Tanszék	Záróvizsgát szervező tanszék: Műszaki Mechanikai Tanszék	
Témavezető: Dr. Kossa Attila, egyetemi docens		

FELADAT

Cím	**Rácsos szerkezet deformációjának számítása VR környezetben** Calculation of deformation of truss structures in VR enviornment
Részletes feladatok	1. Summarize the methods for calculating deformations in 3D truss structures. 2. Develop a custom application for assembling 3D truss structures in a VR environment. 3. Create a numerical solver to determine the deformation of truss structures within a VR environment. 4. Demonstrate the use of your application through selected sample cases. 5. Summarize the results in both Hungarian and English. 6. Prepare a poster summarizing the work carried out in the thesis.
Hely	A szakdolgozat készítés helye: Konzulens: ,

ZÁRÓVIZSGA

1. záróvizsga tantárgy(csoport)	2. záróvizsga tantárgy(csoport)	3. záróvizsga tantárgy(csoport)
ZVEGEMIBMIE Irányításelmélet	**ZVEGEMMBMRO** Robotmechanizmusok dinamikája	**ZVEGEÁTBM04** Áramlások numerikus modellezése

HITELESÍTÉS

Feladat kiadása: 2024. szeptember 2.		Beadási határidő: 2024. december 6.
Összeállította: Dr. Kossa Attila témavezető	Ellenőrizte: *Dr. Insperger Tamás* s.k. tanszékvezető	Jóváhagyta: *Dr. Györke Gábor* s.k. dékánhelyettes

Alulírott, a feladatkiírás átvételével egyúttal kijelentem, hogy a Szakdolgozat-készítés c. tantárgy előkövetelményeit maradéktalanul teljesítettem. Tudomásul veszem, hogy jogosulatlan tantárgyfelvétel esetén a jelen feladatkiírás hatálytalan.

……..
Sas Mór

DECLARATIONS

Declaration of individual work

I, *Mór Sas* (BHS85M), the undersigned, student of the Budapest University of Technology and Economics hereby declare that the present thesis has been prepared by myself without any unauthorized help or assistance such that only the specified sources (references, tools, etc.) were used. All parts taken from other sources word by word or after rephrasing but with identical meaning were unambiguously identified with explicit reference to the sources utilized.

Budapest, 6 Dec 2024

Mór Sas

Acknowledgement

This research was supported by the Hungarian National Research, Development and Innovation Office (FK 142457). This research was supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences.

Budapest, 2024

Mór Sas

Contents

1 Introduction **1**

1.1 Objectives and motivation 1

1.2 Background 1

1.3 Structure of the thesis 2

2 Preparing for the development and other considerations **4**

2.1 The development environment 4

2.2 Modelling the objects used in the application 4

2.3 Building blocks of a 3D project in Unity 4

2.4 Outer dependencies and third-party components 6

2.5 The hardware that I built the application for 7

3 The theoretical background of the numerical solver, its implementation and validation **9**

3.1 The method of calculating deformations of 2D truss structures 9

3.1.1 The truss element 9

3.1.2 The stiffness matrix 9

3.1.3 Truss structures 12

3.2 A simple problem 13

3.2.1 Strain and stress 17

3.3 Numerical solution using the LU factorization 17

3.4 Validation of the numerical solution 17

3.4.1 First problem 17

3.4.2 Second problem 19

3.4.3 The comparison of the analytical solution with the results of the numerical solver 20

4 Structure of the application **24**

4.1 Main functionality of the application 24

4.1.1 The basic principles of Unity 24

4.1.2 The 3D models used in the application 24

4.2 Basic concept of the code . . . 26
4.3 Basic concept of the building blocks . . . 26
4.3.1 Interfaces . . . 26
4.3.2 BaseCustomRayInteractable class . . . 27
4.3.3 BuildingBlock class . . . 28
4.3.4 The classes that inherit from BuildingBlock . . . 29
4.3.5 The final classes of placeable objects . . . 31
4.4 The base class of the tray elements . . . 32
4.4.1 The final classes of the tray elements . . . 33
4.5 Basic concept of the tools . . . 33
4.6 The solving and storing of the structure . . . 36
4.7 The colormap, the slider and the tool belt . . . 41

5 Using the application **44**
5.1 The scene . . . 44
5.2 Types of placeable objects . . . 45
5.3 The wrench tool . . . 46
5.3.1 Using the tray . . . 46
5.3.2 Placing objects on the blackboard . . . 47
5.3.3 Modifying the already placed objects . . . 48
5.3.4 Deleting objects . . . 49
5.3.5 Example . . . 49
5.4 Using the screwdriver . . . 50
5.4.1 The way the results are displayed . . . 50
5.4.2 Reading out the numerical results . . . 51
5.4.3 Looking at the stress field . . . 52
5.5 The slider . . . 53
5.6 A bigger model . . . 54

6 Summary **55**
6.1 Results . . . 55
6.2 Conclusions . . . 55

6.3 Outlook 56

Összefoglaló **59**

References **60**

A. Magyar leírás az alkalmazás használatához **61**

A.1. A virtuális környezet felépítése 61

A.2. Lehelyezhető tárgyak típusai 62

A.3. A villáskulcs használata 63

A.3.1. A tálca használata 63

A.3.2. Elemek lehelyezése a táblára 64

A.3.3. A már lehelyezett elemek módosítása 65

A.3.4. Elemek törlése 65

A.3.5. Példa szerkezet 66

A.4. A csavarhúzó használata 66

A.4.1. Az eredmények megjelenítése 67

A.4.2. A numerikus eredmények kiolvasása 67

A.4.3. A feszültségmező megtekintése 68

A.5. A csúszka 69

A.6. Egy nagyobb modell 71

1 Introduction

1.1 Objectives and motivation

The main objective was to develop an application for VR environment, where students can easily simulate the loading of truss structures, like those that are covered during classes. There has not yet been a thesis on VR technology at the department, but since it is gaining more and more ground in our world. I think it is important for our university to keep up with the new developments, therefore another purpose of the thesis was to show an example application that future students can build on.

Truss structures come up first in statics class during the first semester. At this time most students can not use big and complicated programs like Ansys to check if their results are even qualitatively good or not. At the beginning of the semester for some it is hard to get an intuition even about the direction of the reaction forces. I was willing to take on the project, because I wanted to help everybody who is starting to learn engineering to easily engage with the fundamentals at the beginning.

To achieve this goal, I tried to make the application as understandable and practical to use as I could. Another objective was to make it easy to modify the drawn structure, to be able to experiment with lots of different layouts. I think this is very important, when we meet truss structures the first time, and we are not sure what will happen, because as we experiment more and more we can get a feeling about the results. Also, it makes it easier to understand at the classes why it is happening like that. Other than the educational benefits I tried to make a fun to use application for all the students at the university.

1.2 Background

Consumer VR headsets did not enter the market a long time ago [2]. The Kickstarter campaign for the Oculus Rift started in 2012. Two years later Facebook bought up the company. In 2019 Facebook launched the Oculus Quest. At this time there were plenty of use cases, and content developed for VR headsets, so the Quest became very popular, even sold out in many locations. Just a year later the Quest 2 was released, an upgraded version of the original Quest. Since then, more companies entered the market with more and more capable headsets. For example, Apple's Apple Vision Pro was released this summer, and a direct competitor of the new Quest 3S, the Pico 4 Ultra entered the market this September. It is interesting to see the growth of the competition in this market segment, but I see it as a positive thing as competition drives development.

I worked with the Oculus Quest 2 headset, which is not the newest, as it was released 4 years ago. On the other hand, it is still a pretty capable device. A picture of the headset and its controllers can be seen in Figure 1.1. As stated in [11], it has two 1832 x 1920 pixel resolution displays built in, with 120 Hz refresh rate. Thanks to this the picture

seems totally real time and fluid when looking around. But the resolution is not enough to forget that we are wearing the headset. When looking at landscapes, or when we are close to the objects it works like a charm, but as we are getting further away from the objects their edges get very pixelated. Also, it has built-in 6 DOF head tracking, and 3D positional audio too. The Quest 2 is a standalone headset, which means that, it can run applications without being connected to any other device, thanks to a strong chipset, and a built-in battery. However, it is still possible to connect it to a PC with Oculus Link via a USB Type-C 3.0 cable. In this case the applications run on the computer using its full computational power, and the headset is only used to display the image, and to relay the inputs toward the PC. When I was developing the application I connected the Quest to my computer, because this way I could debug the application from Unity Editor.

Before starting the development I tried some VR games, and applications to get a feeling about the possibilities of the VR environment. I tried to find some VR applications that are similar to what my goal was, but I could not find any. There are some bridge building games, but they are not nearly as sophisticated as some PC games of this genre like Poly Bridge. I also tried to find some open source already existing FEM applications written in C#. I found two GitHub projects [5] and [8]. These both can solve 2D truss structures, but I decided to develop my own solver, because it would have taken more time to understand the code of these projects, than to write my own.

Figure 1.1: The Oculus Quest 2 [3]

1.3 Structure of the thesis

Chapter 2 will present the considerations on which I chose the software tools that I used. Also, these software products are introduced to the reader. The hardware that the application was developed for is presented. Furthermore, all the third-party dependencies used during the development are shown.

In chapter 3 the theoretical background of the numerical solver is presented. Starting

with the derivation of the 2D truss element, then the description of the numerical method. At the end the analytical solution is compared to the numerical one.

In chapter 4 the code design of the VR application is explained. All the way from the base classes to the interaction mechanics.

Chapter 5 gives an instruction on how to use the application. All the possible actions are explained in detail. Also, some sample structures are presented to show the capabilities of the application.

Chapter 6 gives a summary of the results, and the conclusions that can be drawn. Also, some ideas are presented for further development of the application.

2 Preparing for the development and other considerations

2.1 The development environment

When I decided to go with this project I had to choose the most suitable environment. There are two main ways of developing applications: doing it from scratch, or using some kind of 3D game engine. Regarding the complexity of an engine like that, I had to go with the second solution, because it would have been much more complicated and time-consuming to build up a 3D engine. When choosing the software, there are two main ones Unity, and Unreal. Both are game development oriented, have built-in 3D engine, rendering and physics. Unity uses the C# language, while Unreal uses C++. The latter is better for big, computationally heavy applications. Most AAA games use it because of the better, more efficient graphics. However, Unity has a much wider community for helping hobby developers. They have a forum called Unity Discussions, where anybody can ask any question concerning Unity. That was the main reason I chose Unity over Unreal for building my project. Furthermore, Unity uses Visual Studio for editing the C# scripts, from where runtime debugging can also be done even when developing for VR headsets.

2.2 Modelling the objects used in the application

Blender is the most popular 3D modelling software used for game development in Unity. It is a free, and very capable software, that can be used for creating 3D models, animations, and rendering. I went with it, because I knew if I ever need help there are lots of tutorials online. Almost all the models used in the application were made by me in Blender. The models are simple, because the application is not about the visuals, but rather about the mechanics. Also, for better performance, I baked in the lighting in Blender, so the headset does not have to calculate it in real-time. This works by creating a texture, that contains the lighting information in Blender, and then it is applied to the model in Unity. I included a

screenshot of the Blender interface in Figure 2.1.

2.3 Building blocks of a 3D project in Unity

In Unity's environment the main parts of the project are called Scenes. They are like levels in games. For my project I only used one of them, because the player cannot leave the main room.

At the beginning, the scene is an empty 3D space, with only a camera and a directional light (Figure 2.2). Everything that's inside is a game object, which is the base building

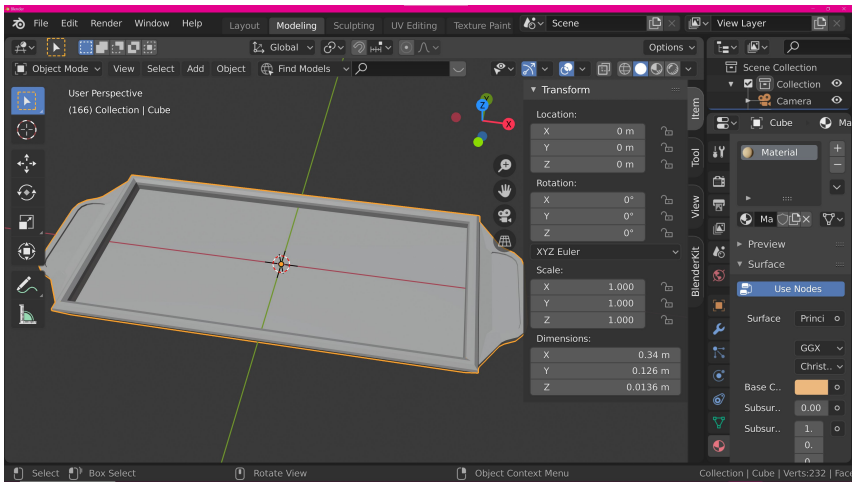


Figure 2.1: Blender's interface.

block of a scene.

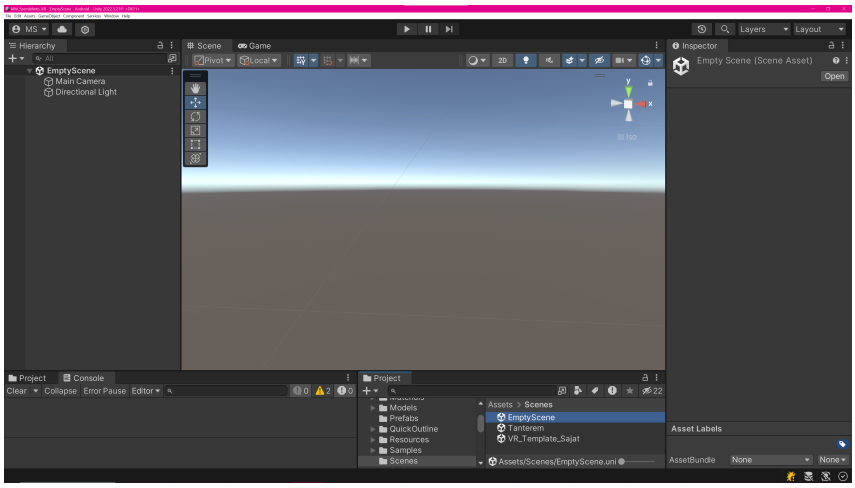


Figure 2.2: An empty scene.

Every game object has some basic properties, name, rendering layer, tag, and a transform component, for position, rotation and scale parameters. These can be seen at the Inspector window. Here it is possible to add more components to game objects, either built-in ones or self-developed scripts. These C# codes can be created in a folder as .cs files, and after being written they can be added to specific game objects, by drag and drop in the editor. Also, in the inspector window the parameters of the components can be set. This works in a way, that in the script it is possible to define SerializeField or public variables, which will show up in the editor's inspector window. Also, from the scripts it is possible to reach all other added components of the game object and modify their parameters. The inspector window of the wrench tool's game object is shown in Figure 2.3. Mesh Renderer is responsible for the visual representation of the object, Rigidbody for the physics, and the XR Grab Interactable for the interaction with the controllers. Box Collider is for the collision detection, and the Line Renderer will be the pointer of

the tool. The Wrench Class script, that I wrote is attached to the game object too.

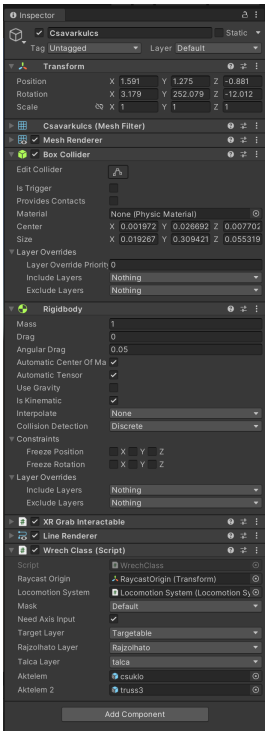


Figure 2.3: Inspector window in Unity.

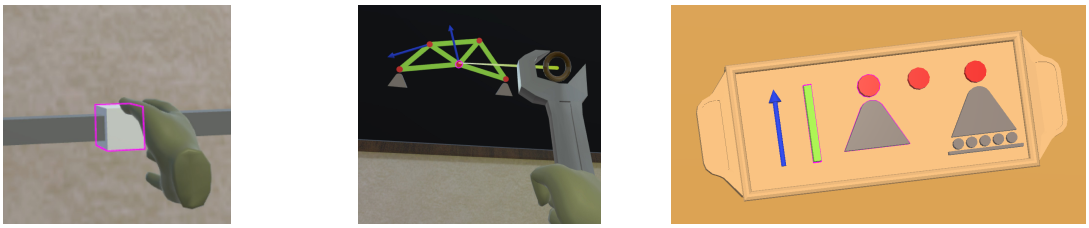
2.4 Outer dependencies and third-party components

Here I will show all third-party packages (Not written by me, nor provided by Unity) I used in my project:

- VR Hands [12]: This package contains not only the 3D model of a right and a left hand, but the animations too for the hand gestures. This makes it possible to have a smooth transition between for example an open and closed fist. The position of the hand model is determined by the trigger (for pinch position) and grip (for fist position) buttons. The positions can be seen in Figure 2.4
- Quick Outline [10]: This visual package is developed for VR environments. I use it for example to outline objects when they are hovered. It has a huge role in the user experience of the application, because it is the main feedback toward the user about their actions. Examples of how I used it can be seen in Figure 2.5.
- OpenXR [6]: This plugin makes it possible to run the application on multiple types of virtual reality devices.
- Materials: The materials with textures are not made by me, I downloaded them from the website [1].

(a) (b) (c)

Figure 2.4: The hand model, and examples of hand gestures using VR Hands. The 3D hand model (a), the fist position (b), and the pinch position (c).



(a) (b) (c)

Figure 2.5: Examples for the usage of the Outline package. Indicating interactable objects (a), indicating hovering (b), indicating the active objects (c).

2.5 The hardware that I built the application for

Through the OpenXR library the application is able to run on several types of VR headsets, but I could only test it with my Oculus Quest 2 (Figure 1.1). Two controllers come with the headset, one for each hand. They both have 6 DOF position and location feedback to the headset. Their button layout can be seen in Figure 2.6. In general, the Grip

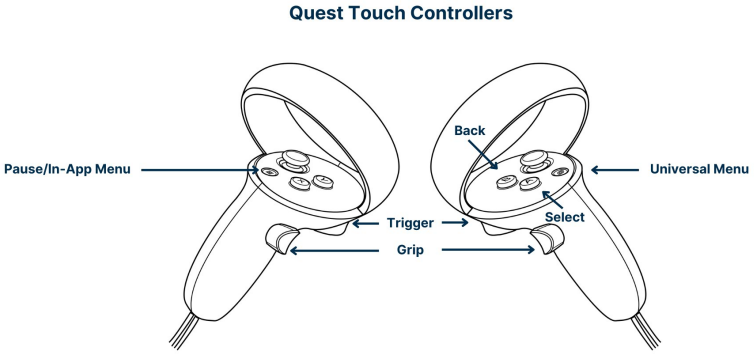


Figure 2.6: The Button layout of the Quest Touch controllers [9].

button is used for grabbing and holding onto objects, this is how I used it in my application too. In other games this button is used for the climbing mechanism. The Trigger button is used in the menu with UI elements as a universal select/click action. In games,

it is often linked to pulling the trigger of weapons. In my case this button is only live if there is a tool in the user's hand. My tools have a pointer, and the Trigger button is used like a click event on a 3D mouse. Both controllers have a clickable joystick too. They are mainly used for navigating in the 3D space. That's one part how I used them, but I added some functionality to them at the building phase of my application too. In addition, both controllers have two action buttons. Generally, the ones on the right controller are used for navigating through the levels of menus, with the A button used as select and the B button used as back. I also make use of these buttons in the building process. One is for placing specific elements and the other is for deleting them, but I will write about this in detail during Chapter 5. The last thing is the two menu buttons. With these buttons the user can pause and exit the games.

3 The theoretical background of the numerical solver, its implementation and validation

3.1 The method of calculating deformations of 2D truss structures

All the derivations are from the lecture notes of [7].

3.1.1 The truss element

Truss is the most basic element used in finite element modelling. In the 2D case it has four degrees of freedom. These are the x and y coordinates of the two endpoints. Example for a 2D Truss element is shown in Figure 3.1. This is a one-dimensional model of a

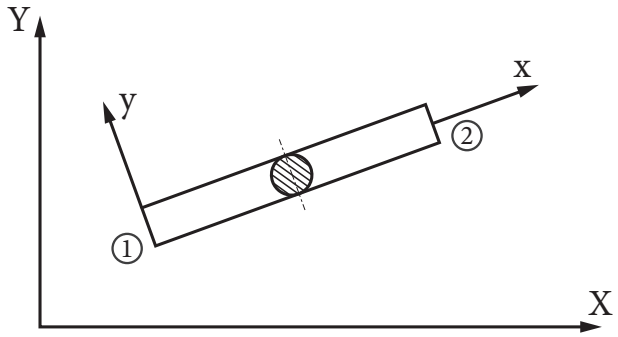


Figure 3.1: A 2D truss element.

3-dimensional rod, so some constants and physical properties are needed to make the bond between real life and the model. These are the cross-section area and the Young's modulus of the material.

3.1.2 The stiffness matrix

To determine the stiffness matrix of a single element we have to start from a drawing (Figure 3.2) with all the parameters that we will use. The global coordinate system is shown on the left, the local one is denoted with an ˜ on top of the coordinates. The length of the truss is denoted with L. The vectors $\mathbf{u_1}$ and $\mathbf{u_2}$ are the displacements in the local coordinate system of the element. They are parallel to the direction of the truss, because the truss ca not bend out from its local x coordinate axis. But the nodes are not only moving in the direction of the truss. The displacement in the global coordinate system is represented by the $\mathbf{U_1}$, $\mathbf{V_1}$, $\mathbf{U_2}$ and $\mathbf{V_2}$ vectors. The angle of the element is denoted by α.

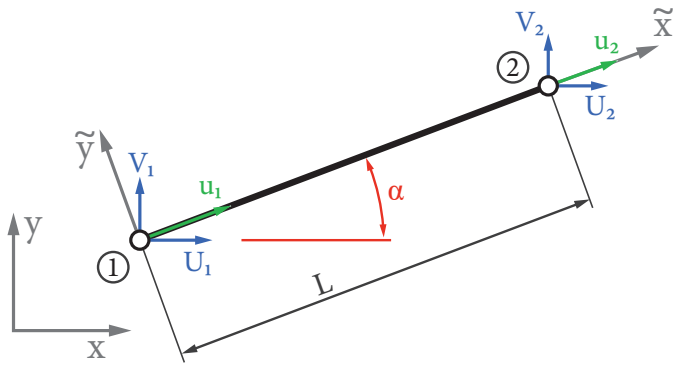


Figure 3.2: A 2D truss with all the needed values represented.

The relation between the local and global displacements can be written as

$$u_1 = U_1 \cdot \cos\alpha + V_1 \cdot \sin\alpha, \tag{3.1}$$

$$u_2 = U_2 \cdot \cos\alpha + V_2 \cdot \sin\alpha. \tag{3.2}$$

This is only true if there is just a small difference between the original, and deformed element's angle. However, this is true most of the time. Equations (3.1) and (3.2) can be combined to form a vector equation

$$\begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} \cos\alpha & \sin\alpha & 0 & 0 \\ 0 & 0 & \cos\alpha & \sin\alpha \end{bmatrix} \cdot \begin{bmatrix} U_1 \\ V_1 \\ U_2 \\ V_2 \end{bmatrix}. \tag{3.3}$$

The matrix, that makes connection between the local and global displacements is called the transformation matrix $\mathbf{T}$

$$\begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \mathbf{T} \cdot \begin{bmatrix} U_1 \\ V_1 \\ U_2 \\ V_2 \end{bmatrix}, \tag{3.4}$$

where

$$\mathbf{T} = \begin{bmatrix} \cos\alpha & \sin\alpha & 0 & 0 \\ 0 & 0 & \cos\alpha & \sin\alpha \end{bmatrix}. \tag{3.5}$$

Now let's derive the stiffness equation for the 2D truss in the local coordinate system of the element, using Figure 3.3.

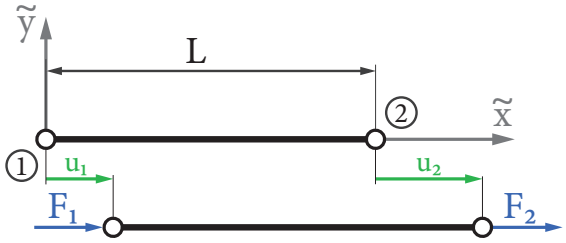


Figure 3.3: Truss element's displacement in its own coordinate system.

The length change is

$$\Delta u = u_2 - u_1. \tag{3.6}$$

Equilibrium equation:

$$F_1 + F_2 = 0. \tag{3.7}$$

The normal force from the loads is

$$N = F_2 = -F_1 = k \cdot \Delta u = k(u_2 - u_1), \tag{3.8}$$

where k is the stiffness, that can be calculated using the cross-section area, the Young's modulus of the material and the length of the element, as

$$k = \frac{A \cdot E}{L}. \tag{3.9}$$

The forces at the nodes are

$$F_1 = ku_1 - ku_2, \tag{3.10}$$

$$F_2 = -ku_1 + ku_2. \tag{3.11}$$

The two Equations (3.10) and (3.11) can be written in a matrix form as

$$\begin{bmatrix} k & -k \\ -k & k \end{bmatrix} \cdot \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} F_1 \\ F_2 \end{bmatrix}. \tag{3.12}$$

This equation can be written as: (the tildes stand for being in the local coordinate system)

$$\tilde{\mathbf{K}} \cdot \tilde{\mathbf{U}} = \tilde{\mathbf{F}}. \tag{3.13}$$

This is the stiffness equation, where $\tilde{\mathbf{K}}$ is the stiffness matrix, $\tilde{\mathbf{U}}$ is the displacement vector of the nodes and $\tilde{\mathbf{F}}$ vector represents the forces acting on the nodes.

Equation (3.4) can be substituted into Equation (3.12):

$$\frac{A \cdot E}{L} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \mathbf{T} \begin{bmatrix} U_1 \\ V_1 \\ U_2 \\ V_2 \end{bmatrix} = \begin{bmatrix} F_1 \\ F_2 \end{bmatrix}. \tag{3.14}$$

With multiplying both side of the equation with $\mathbf{T}^T$ from the left, we get

$$\frac{A \cdot E}{L} \mathbf{T}^{\mathrm{T}} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \mathbf{T} \begin{bmatrix} U_1 \\ V_1 \\ U_2 \\ V_2 \end{bmatrix} = \mathbf{T}^{\mathrm{T}} \begin{bmatrix} F_1 \\ F_2 \end{bmatrix} \tag{3.15}$$

. The right side of the equation can be expressed as

$$\mathbf{F} = \begin{bmatrix} F_1 \cdot \cos\alpha \\ F_1 \cdot \sin\alpha \\ F_2 \cdot \cos\alpha \\ F_2 \cdot \sin\alpha \end{bmatrix} = \begin{bmatrix} F_{1x} \\ F_{1y} \\ F_{2x} \\ F_{2y} \end{bmatrix}. \tag{3.16}$$

Also, expressing the $\frac{A \cdot E}{L} \mathbf{T}^{\mathrm{T}} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \mathbf{T}$ part of the left side:

$$\mathbf{K} = \frac{A \cdot E}{L} \begin{bmatrix} \cos\alpha^2 & \cos\alpha \cdot \sin\alpha & -\cos\alpha^2 & -\cos\alpha \cdot \sin\alpha \\ \cos\alpha \cdot \sin\alpha & \sin\alpha^2 & -\cos\alpha \cdot \sin\alpha & -\sin\alpha^2 \\ -\cos\alpha^2 & -\cos\alpha \cdot \sin\alpha & \cos\alpha^2 & \cos\alpha \cdot \sin\alpha \\ -\cos\alpha \cdot \sin\alpha & -\sin\alpha^2 & \cos\alpha \cdot \sin\alpha & \sin\alpha^2 \end{bmatrix} \tag{3.17}$$

. Writing back the expressed forms to Equation (3.15), we receive

$$\begin{aligned} \frac{A \cdot E}{L} \begin{bmatrix} \cos\alpha^2 & \cos\alpha \cdot \sin\alpha & -\cos\alpha^2 & -\cos\alpha \cdot \sin\alpha \\ \cos\alpha \cdot \sin\alpha & \sin\alpha^2 & -\cos\alpha \cdot \sin\alpha & -\sin\alpha^2 \\ -\cos\alpha^2 & -\cos\alpha \cdot \sin\alpha & \cos\alpha^2 & \cos\alpha \cdot \sin\alpha \\ -\cos\alpha \cdot \sin\alpha & -\sin\alpha^2 & \cos\alpha \cdot \sin\alpha & \sin\alpha^2 \end{bmatrix} \begin{bmatrix} U_1 \\ V_1 \\ U_2 \\ V_2 \end{bmatrix} = \\ = \begin{bmatrix} F_1 \cdot \cos\alpha \\ F_1 \cdot \sin\alpha \\ F_2 \cdot \cos\alpha \\ F_2 \cdot \sin\alpha \end{bmatrix}. \end{aligned} \tag{3.18}$$

It can also be written in a more compact form as

$$\mathbf{K} \cdot \mathbf{U} = \mathbf{F}, \tag{3.19}$$

where all the values are in the global coordinate system.

3.1.3 Truss structures

For truss structures an element-node assembly table has to be made, that for one element contains its number in the structure and the number of the nodes it is connected to.

These numbers belong to the whole structure, so they should not be confused with the local numbering used before.

As the stiffness matrix of a single element, the global stiffness matrix will be symmetric too. The size of the global stiffness matrix will be two times the number of nodes, because this is how there will be one equation for all possible degree of freedom in the truss structure. Using the element-node assembly table it is possible to fill the global stiffness matrix. The local stiffness values are placed to their corresponding places, based on what is the global number of the element's local 1. and 2. nodes. If the global stiffness matrix already has a value at that place, then the values should be added together.

The global displacement vector consists of two values for all nodes. And the global force vector consists of one x-axis and one y-axis direction force value. The values in these two vectors correspond to the global number of the nodes in order from top to bottom. All values, which are known from the boundary conditions should be filled. For example, if a node is fixed, its displacement values must be 0, or if there are no external forces acting on a free node, then the corresponding force values must be 0. After applying all the boundary conditions, the $\mathbf{K} \cdot \mathbf{U} = \mathbf{F}$ equation of the truss structure can be condensed, by emitting the rows, where the displacement is 0, while also emitting the corresponding columns of the $\mathbf{K}$ matrix. This way it will stay a square matrix.

The solution comes from inverting the condensed $\mathbf{K}$ matrix, and then multiplying both sides of the equation from the left with it. The results are the non-zero values of the displacement vector, which can be written back to the original stiffness equation, and by evaluating the left side, the global force vector can be gained.

3.2 A simple problem

Let's see this method in action with the help of a small problem (Figure 3.4).

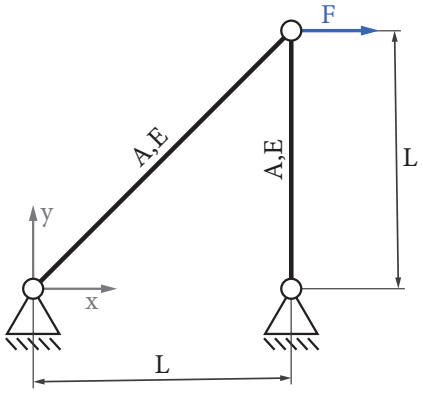


Figure 3.4: A simple problem.

First thing is to specify the sequence numbering of the nodes and elements (Figure 3.5).

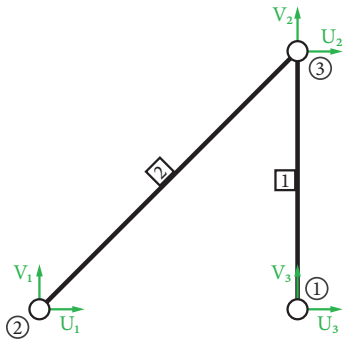


Figure 3.5: Numbering of the simple problem.

From Figure 3.5, the element-node assembly table can be seen at Table 3.1. The

Table 3.1: The element-node assembly table.

element	local node 1	local node 2
1	1	3
2	2	3

coordinates of the nodes are shown in Table 3.2. From the coordinates of Table 3.2

Table 3.2: The coordinates of the nodes in the global coordinate system.

# of node	x coordinate	y coordinate
1	0	0
2	L	L
3	L	0

we can calculate the length and the angle of the trusses (Table: 3.3). For the sake of

Table 3.3: Parameters of the two truss elements.

element	length	angle
1	L	90°
2	$\sqrt{2}L$	45°

simplicity, the cross-section area A and the Young's modulus E is the same for all the

elements. The local stiffness matrices are

$$\mathbf{K}_1 = \frac{A \cdot E}{L} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix}, \tag{3.20}$$

$$\mathbf{K}_2 = \frac{A \cdot E}{L} \begin{bmatrix} \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & \frac{-1}{2\sqrt{2}} & \frac{-1}{2\sqrt{2}} \\ \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & \frac{-1}{2\sqrt{2}} & \frac{-1}{2\sqrt{2}} \\ \frac{-1}{2\sqrt{2}} & \frac{-1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} \\ \frac{-1}{2\sqrt{2}} & \frac{-1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} \end{bmatrix}, \tag{3.21}$$

where the values are already substituted for the angles. The global stiffness matrix will be a 6 by 6 matrix, because there are 3 nodes that can move in the x and y directions:

$$\begin{aligned} \mathbf{K} &= \begin{bmatrix} \mathbf{X_{11}} & \mathbf{X_{12}} & 0 & 0 & \mathbf{X_{13}} & \mathbf{X_{14}} \\ \mathbf{X_{21}} & \mathbf{X_{22}} & 0 & 0 & \mathbf{X_{23}} & \mathbf{X_{24}} \\ 0 & 0 & \mathbf{O_{11}} & \mathbf{O_{12}} & \mathbf{O_{13}} & \mathbf{O_{14}} \\ 0 & 0 & \mathbf{O_{21}} & \mathbf{O_{22}} & \mathbf{O_{23}} & \mathbf{O_{24}} \\ \mathbf{X_{31}} & \mathbf{X_{32}} & \mathbf{O_{31}} & \mathbf{O_{32}} & \mathbf{X_{33}} + \mathbf{O_{33}} & \mathbf{X_{34}} + \mathbf{O_{34}} \\ \mathbf{X_{41}} & \mathbf{X_{42}} & \mathbf{O_{41}} & \mathbf{O_{42}} & \mathbf{X_{43}} + \mathbf{O_{43}} & \mathbf{X_{44}} + \mathbf{O_{44}} \end{bmatrix} = \\ &= \frac{A \cdot E}{L} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & -1 \\ 0 & 0 & \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & \frac{-1}{2\sqrt{2}} & \frac{-1}{2\sqrt{2}} \\ 0 & 0 & \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & \frac{-1}{2\sqrt{2}} & \frac{-1}{2\sqrt{2}} \\ 0 & 0 & \frac{-1}{2\sqrt{2}} & \frac{-1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} \\ 0 & -1 & \frac{-1}{2\sqrt{2}} & \frac{-1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & 1 + \frac{1}{2\sqrt{2}} \end{bmatrix}. \end{aligned} \tag{3.22}$$

In Equation (3.22) I tried to highlight the way that the global stiffness matrix is put together. Blue crosses represent the values from the stiffness matrix of the first element, and green circles represent the values from the stiffness matrix of the second element. The subscripts mark the exact place of the given values in the original stiffness matrices (row, column).

The displacement vector is

$$\mathbf{U} = \begin{bmatrix} U_1 \\ V_1 \\ U_2 \\ V_2 \\ U_3 \\ V_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ U_3 \\ V_3 \end{bmatrix}, \tag{3.23}$$

where the values for the first and second nodes are zero because of the boundary conditions (the two fixed nodes).

The global force vector is

$$\mathbf{F} = \begin{bmatrix} F_{1x} \\ F_{1y} \\ F_{2x} \\ F_{2y} \\ F_{3x} \\ F_{3y} \end{bmatrix} = \begin{bmatrix} F_{1x} \\ F_{1y} \\ F_{2x} \\ F_{2y} \\ F \\ 0 \end{bmatrix}. \tag{3.24}$$

Now let's get to the stiffness equation:

$$\mathbf{K} \cdot \mathbf{U} = \mathbf{F} \tag{3.25}$$

$$\frac{A \cdot E}{L} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & -1 \\ 0 & 0 & \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & \frac{-1}{2\sqrt{2}} & \frac{-1}{2\sqrt{2}} \\ 0 & 0 & \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & \frac{-1}{2\sqrt{2}} & \frac{-1}{2\sqrt{2}} \\ 0 & 0 & \frac{-1}{2\sqrt{2}} & \frac{-1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} \\ 0 & -1 & \frac{-1}{2\sqrt{2}} & \frac{-1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & 1 + \frac{1}{2\sqrt{2}} \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ U_3 \\ V_3 \end{bmatrix} = \begin{bmatrix} F_{1x} \\ F_{1y} \\ F_{2x} \\ F_{2y} \\ F \\ 0 \end{bmatrix}. \tag{3.26}$$

From the full form of the stiffness equation (Equation 3.26), with the condensation we get

$$\frac{A \cdot E}{L} \begin{bmatrix} \frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} \\ \frac{1}{2\sqrt{2}} & 1 + \frac{1}{2\sqrt{2}} \end{bmatrix} \cdot \begin{bmatrix} U_3 \\ V_3 \end{bmatrix} = \begin{bmatrix} F \\ 0 \end{bmatrix}, \tag{3.27}$$

where the only unknowns are the U_3 and V_3 displacements. This can be solved by multiplying both sides of the equation from the left with the inverse of the condensed stiffness matrix. The result of the calculation is

$$\begin{bmatrix} U_3 \\ V_3 \end{bmatrix} = \frac{L \cdot F}{A \cdot E} \begin{bmatrix} 1 + 2\sqrt{2} \\ -1 \end{bmatrix}. \tag{3.28}$$

Writing the values back to the displacement vector, we get

$$\mathbf{U} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ \frac{L \cdot F}{A \cdot E}(1 + 2\sqrt{2}) \\ -\frac{L \cdot F}{A \cdot E} \end{bmatrix}. \tag{3.29}$$

If we substitute back the full displacement vector from Equation (3.29) to the original stiffness equation (Equation 3.26), we receive the full force vector

$$\mathbf{F} = \begin{bmatrix} 0 \\ F \\ -F \\ -F \\ F \\ 0 \end{bmatrix}. \tag{3.30}$$

3.2.1 Strain and stress

From the calculated displacement it is possible to calculate the strain and stress values for each element in the structure

$$\varepsilon = \frac{u_2 - u_1}{L} = \frac{\cos\alpha \cdot (U_2 - U_1) + \sin\alpha \cdot (V_2 - V_1)}{L}, \tag{3.31}$$

$$\sigma = \varepsilon \cdot E. \tag{3.32}$$

The equations are written using the notations of Figure 3.2.

3.3 Numerical solution using the LU factorization

At first, I made a program function, that inverted the $\mathbf{K}$ matrix. In theory, it would have had to work, but at the step, when the program calculated the determinant, it ran out of the range of float variables. Float is a variable type, that takes up 4 bytes of memory and has $6-7$ significant digits. It's possible value ranges from $1.175494351 \cdot 10^{-38}$ to $3.402823466 \cdot 10^{38}$. The determinant could get greater, because I decided not to extract $\frac{A \cdot E}{L}$, because, that would have meant all the elements must have the same specific properties in the structure. However, I did not implement variable properties in the end. Switching to double variables seemed to solve the issue, but when evaluating bigger structures, the application froze multiple times. After realizing this, I decided to search for some more optimal method to solve the equation. I choose to implement the LU factorization [4]. Applying this method for Equation (3.25) means, that the $\mathbf{K}$ matrix is decomposed into one lower $\mathbf{l}$ and one upper $\mathbf{u}$ triangular matrix, whose product is $\mathbf{K}$

$$\mathbf{K} = \mathbf{l} \cdot \mathbf{u}. \tag{3.33}$$

Writing back K to the stiffness equation, we get

$$\mathbf{K} \cdot \mathbf{U} = (\mathbf{l} \cdot \mathbf{u}) \cdot \mathbf{U} = \mathbf{l} \cdot (\mathbf{u} \cdot \mathbf{U}) = \mathbf{F}. \tag{3.34}$$

The first step is to solve $\mathbf{l} \cdot \mathbf{y} = \mathbf{F}$ with forward-substitution for $\mathbf{y}$ vector. Then substituting back the result, and solve $\mathbf{u} \cdot \mathbf{U} = \mathbf{y}$ with backward-substitution for $\mathbf{U}$.

3.4 Validation of the numerical solution

After programming the solver, I had to validate its results. I did this by comparing the results of the numerical solver with analytical solutions of two problems, that I found in the example book of the Strength of Materials course [13].

3.4.1 First problem

It is numbered as 1.2 in the book. The problem in English is the following:

Example 1.2. A steel bar with a diameter of 20 mm is subjected to a circular load of 50 kN in cross-section B and a force of 40 kN in cross-section C as shown in the Figure 3.6. The modulus of elasticity of steel is 200 GPa. The tasks are, to draw the normal force diagram, calculate the end cross-section's displacement and to determine the stresses arising at each section.

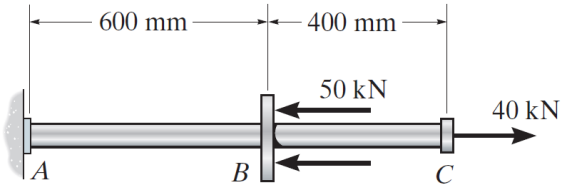


Figure 3.6: The first problem [13].

The normal force diagram can be seen in Figure 3.7.

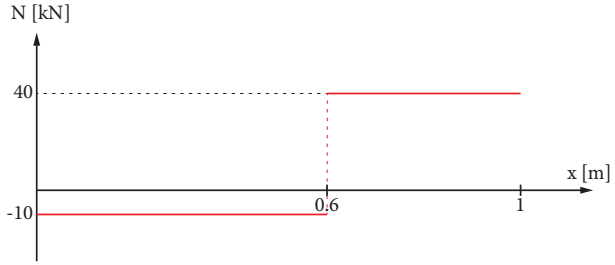


Figure 3.7: Normal force diagram.

The displacement of the cross-sections can be calculated using Hooke's law, where $A = \frac{d^2}{4} \cdot \pi = 314.159$ mm^2 is the cross-section area, $E = 200$ GPa is the Young's modulus, $L_1 = 0.6$ m is the length of the first bar and $L_2 = 0.4$ m is the length of the second bar.

$$\Delta L_1 = \frac{N_1 \cdot L_1}{A \cdot E} = -0.09549304651 \text{ mm} \tag{3.35}$$

$$\Delta L_2 = \frac{N_2 \cdot L_2}{A \cdot E} = 0.254648124 \text{ mm} \tag{3.36}$$

From the two length changes, the end cross-section's displacement is

$$\Delta L = \Delta L_1 + \Delta L_2 = 0.1591550775 \text{ mm}. \tag{3.37}$$

The stresses at the two cross-sections are

$$\sigma_1 = \frac{N_1}{A} = -31.8310155 \text{ MPa}, \tag{3.38}$$

$$\sigma_2 = \frac{N_2}{A} = 127.324062 \text{ MPa}. \tag{3.39}$$

3.4.2 Second problem

It is numbered as 7.7 in the book. The problem in English is the following:

Example 7.7. In the truss structure below (Figure 3.8), the cross-sections and material properties of the bars are the same. Determine the vertical displacement of cross-section B.

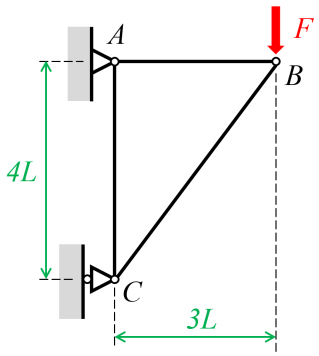


Figure 3.8: The second problem [13].

I will use the following numbering of the trusses:

- Number 1: The truss between A and B
- Number 2: The truss between A and C
- Number 3: The truss between B and C

First, let's calculate the forces at the nodes

$$\sum M_A = 0 \Rightarrow F \cdot 3L - F_C \cdot 4L = 0 \Rightarrow F_C = \frac{3}{4} \cdot F, \tag{3.40}$$

$$\sum F_x = 0 \Rightarrow -F_{Ax} + F_C = 0 \Rightarrow F_{Ax} = \frac{3}{4} \cdot F, \tag{3.41}$$

$$\sum F_y = 0 \Rightarrow F_{Ay} - F = 0 \Rightarrow F_{Ay} = F. \tag{3.42}$$

The vertical displacement of the cross-section B can be calculated using the Castigliano law:

$$f = \frac{\partial U}{\partial F} = \frac{\partial U_N}{\partial F}. \tag{3.43}$$

The normal force functions and their derivates can be written as

$$N_1 = F_{Ax} = \frac{3}{4} F \Rightarrow \frac{\partial N_1}{\partial F} = \frac{3}{4}, \tag{3.44}$$

$$N_2 = F_{Ay} = F \Rightarrow \frac{\partial N_2}{\partial F} = 1, \tag{3.45}$$

$$N_3 = -\sqrt{(\frac{3}{4}\,F)^2 + F^2} = -\frac{5}{4}\,F \Rightarrow \frac{\partial N_3}{\partial F} = -\frac{5}{4}. \tag{3.46}$$

Written back to the Castigliano law, we get

$$f = \frac{1}{A \cdot E}\left[\int_0^{3L} N_1 \cdot \frac{\partial N_1}{\partial F}\,ds + \int_0^{4L} N_2 \cdot \frac{\partial N_2}{\partial F}\,ds + \int_0^{5L} N_3 \cdot \frac{\partial N_3}{\partial F}\,ds\right], \tag{3.47}$$

by substituting back all the derived expressions:

$$f = \frac{1}{A \cdot E}\left(\frac{27}{16}\,FL + 4\,FL + \frac{125}{16}\,FL\right), \tag{3.48}$$

$$f = \frac{27\,FL}{2\,AE}. \tag{3.49}$$

With the values of $F = 20$ kN, $L = 1$ m, $A = 0.0005$ m^2, $E = 200$ GPa the result is $f = 2.7$ mm.

3.4.3 The comparison of the analytical solution with the results of the numerical solver

In my application the user can not build up structures precisely as is not designed for that. I could have made an artificial structure in the Unity editor, and then evaluate it using the headset, but that would have been rather difficult. Instead, I made a separate C# project for the validation. I copied the code of the TrussStructure2D, Node2D, Force2D and TrussElement2D classes to the new project. (I will write about these classes later in the next chapter.) After deleting the unnecessary and Unity related parts (a simple C# console application ca not use anything that is part of Unity's 3D engine), in the main function I set up the structure, that is in the first problem. A picture of the code can be seen in Figure 3.9.

First, I define a list of Node2D objects, that contains all the nodes. Then, I define the boundary conditions of the nodes as they are in the example. Next, I make a list of TrussElement2D, where the individual elements are defined by the two nodes between which they are located, their Young's modulus and cross-section area. Finally, a list of Force2D objects, where the elements are defined by the node where they are acting, and the force components in the x and y direction. When this is done, I can set up the TrussStructure2D object with the lists, and call the Solve() function for the results. After the calculations have run, the program prints out the results to the console (Figure 3.10).

```
0 references
static void Main(string[] args)
{
    //Example 1.2
    List<Node2D> nodes = new List<Node2D>{
        //Defining the nodes
        new Node2D(0,0),
        new Node2D(0.6,0),
        new Node2D(1,0)
    };
    //Defining the boundary conditions
    nodes[0].boundaryCondition = BoundaryCondition.Pinned;
    nodes[1].boundaryCondition = BoundaryCondition.xRoller;
    nodes[2].boundaryCondition = BoundaryCondition.xRoller;
    //Defining the truss elements
    List<TrussElement2D> elements = new List<TrussElement2D>
    {
        new TrussElement2D(nodes[0],nodes[1],200e9,0.000314159),
        new TrussElement2D(nodes[1],nodes[2],200e9,0.000314159),
    };
    //Defining the loads
    List<Force2D> forces = new List<Force2D>
    {
        new Force2D(nodes[1],-50000,0),
        new Force2D(nodes[2],40000,0)
    };
    //Creating the structure
    TrussStructure2D Structure = new TrussStructure2D(nodes, elements, forces);
    //Solving the structure
    Structure.Solve();
    //Printing the results
    Console.WriteLine("The resulting displacement vector:");
    foreach (double displacement in Structure.globalDisplacementVector)
    {
        Console.WriteLine(displacement*1000 + " [mm]");
    }
    foreach(TrussElement2D element in Structure.trusses)
    {
        Console.WriteLine("Stress: " + element.Stress/1000000 + " [MPa]");
        Console.WriteLine("Strain: " + element.Strain);
    }
    Console.ReadKey();
}
```

Figure 3.9: Code for the validation of the first problem.

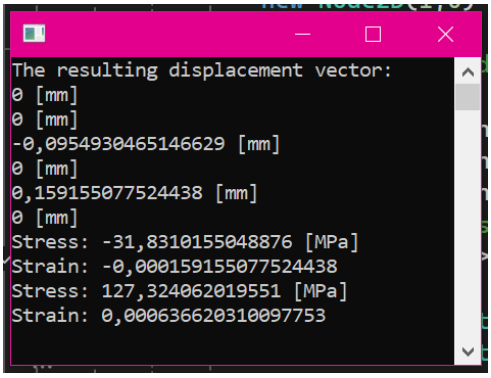


Figure 3.10: The console output for the first problem.

I repeated the same steps for the second problem too. The code can be seen at Figure 3.11 and the console output in Figure 3.12.

```
0 references
static void Main(string[] args)
{
    List<Node2D> nodes = new List<Node2D>{
        //Example 7.7 L=1m A=0.0005m^2 E=200GPa
        new Node2D(0,0),
        new Node2D(3,0),
        new Node2D(0,-4)
    };

    //Example 7.7 L=1m A=0.0005m^2 E=200GPa
    nodes[0].boundaryCondition = BoundaryCondition.Pinned;
    nodes[1].boundaryCondition = BoundaryCondition.Free;
    nodes[2].boundaryCondition = BoundaryCondition.yRoller;

    List<TrussElement2D> elements = new List<TrussElement2D>
    {
        //Example 7.7 L=1m A=0.0005m^2 E=200GPa
        new TrussElement2D(nodes[0],nodes[1],200e9,0.0005),
        new TrussElement2D(nodes[0],nodes[2],200e9,0.0005),
        new TrussElement2D(nodes[1],nodes[2],200e9,0.0005)
    };
    List<Force2D> forces = new List<Force2D>
    {
        //Example 7.7 L=1m A=0.0005m^2 E=200GPa
        new Force2D(nodes[1],0,-20000),
    };

    TrussStructure2D Structure = new TrussStructure2D(nodes, elements, forces);
    Structure.Solve();
    Console.WriteLine("The resulting displacement vector:");
    foreach (double displacement in Structure.globalDisplacementVector)
    {
        Console.WriteLine(displacement * 1000 + " [mm]");
    }

    foreach (TrussElement2D element in Structure.trusses)
    {
        Console.WriteLine("Stress: " + element.Stress / 1000000 + " [MPa]");
        Console.WriteLine("Strain: " + element.Strain);
    }

    Console.ReadKey();
}
```

Figure 3.11: Code for the validation of the second problem.

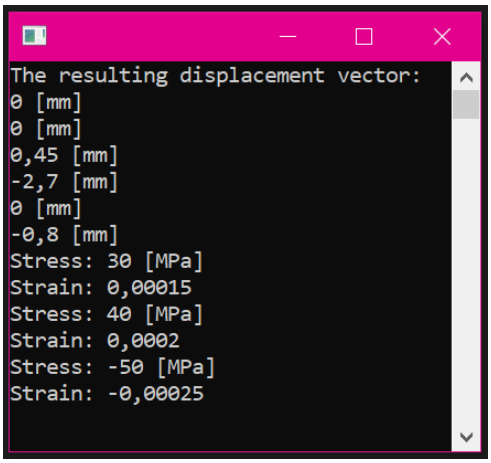


Figure 3.12: The console output for the second problem.

The results I got using numerical solver and the analytical solutions can be found in Table 3.4. The numerical solver gives the same results as the analytical solutions with a calculator, so the numerical solver works correctly. The only difference is, that while my

calculator can only display 10 digits, the computer uses double precision variables, that can store 15-16 significant digits, which it can display full length in the console.

Table 3.4: Solutions for the first (1) and second (2) example.

(1)	Numerical solver	Analytical solution
ΔL	0.159155077524438 [mm]	0.1591550775 [mm]
σ_1	-31.8310155048876 [MPa]	-31.8310155 [MPa]
σ_2	127.324062019551 [MPa]	127.324062 [MPa]

(2)	Numerical solver	Analytical solution
f	2.7 [mm]	2.7 [mm]

4 Structure of the application

4.1 Main functionality of the application

The main principle of the application is to use the wrench to create the truss structure on the blackboard. Then, when the user is ready, they can grab the screwdriver to see the deformation of the structure resulting from the applied forces. When the user finished the evaluation they can modify the structure with the wrench, and see the new results with the screwdriver in hand. So it is easy and fast to make modifications, and see how the deformations vary depending on the shape of the truss structure .

4.1.1 The basic principles of Unity

Unity handles collisions with components called colliders. These are invisible shapes, that are used to detect if two objects are touching each other. They can be set to trigger, so they do not block the movement of the objects, but they still can detect if they are touching each other. I mostly used colliders this way. Also, they are used for detecting game objects when ray casting.

Ray casting is a method, where the computer looks into a given direction, and if there is any collider in the scene along that line it gives back a reference of the game object it is attached to. It can be imagined like pointing a laser in the 3D space to search for opaque bodies.

Another key principle of unity is the prefabs. Prefabs are game objects, that are stored in the assets of the project, and can be instantiated in the scene. The instantiated objects are independent of the prefab, so if an instantiated object is modified, the prefab object and the other instantiated objects are not affected. This is very useful, because the same object can be used multiple times in the scene, and it is enough to configure the prefab object, and all the instantiated objects will be the same.

4.1.2 The 3D models used in the application

Almost all 3D models used in the application are modeled by me, the hand models are the only exception. As I mentioned before I used Blender to create, and export them to .fbx file format. The models are very simple, but they are sufficient for the visualization. Some of them are just used as decoration, like the whole room and the stand of the display. In the editor I set up the game objects that can be placed on the blackboard, and then I made prefab objects from them. Next, I will describe the most important properties of these models, that make the application work.

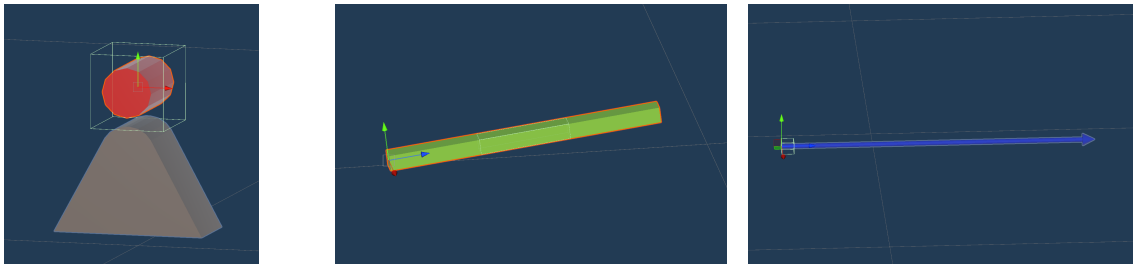
All different nodes have a red cylinder as the parent object, whose coordinate system is attached to the middle of the cylinder. This is crucial to be able to place them precisely

without any offset needed, also they are rotated around this point. The semi-free node and the fixed node have some additional objects, that make them look like the usual notation. These 3D models are the child objects of the red cylinder, this way if the red cylinder is moved or rotated, they move and rotate with it. This setup is beneficial, because this way it is enough to manipulate the red cylinder, and I don't have to worry about the complementary objects. A box collider in trigger mode is attached to the red cylinder, which is slightly bigger than the cylinder itself, so the user can hit the node with the pointer easily.

The 3D model of the truss is a simple square prism. Its coordinate system is attached to the end of the square prism, in a way, that the z axis points along the direction of the truss. This is important, because the length of the truss will be determined by the scale of the object along the z axis. For this to work one more condition has to be met. The original length of the square prism (when the scale value is 1) has to be exactly 1 unit long. The truss has a box collider in trigger mode too. It covers the middle third of the prism.

The 3D model of the force consists of two parts, a shaft and a head. The shaft is a simple cylinder, and the head is a cone. It is arranged in a way, that both parts are the child objects of an empty game object. The coordinate system of the empty game object is set up in a way, that the origin is at the beginning of the shaft and the z axis points towards the direction of the shaft, towards the head. Here just like in the case of the truss, the length of the shaft is set to be 1 unit long, when its scale is 1. A box collider roughly the size of the head is attached to the parent object. It would have been easier to attach the collider to the head, but the head is a child object, and the ray cast would give back the reference of the head, not the parent object's. This would have made it a lot harder to adjust the position of the force, so I attached the collider to the parent object. The only downside of this is, that I have to move to collider separately to where the head is.

These models are shown in Figure 4.1. All of them have a collider, an Outline component, and also a unique script attached to them.



(a) **(b)** **(c)**

Figure 4.1: The fixed node, the truss and the force models with their origins and colliders. The fixed node(a), the truss (b), the force (c).

4.2 Basic concept of the code

At my first attempt I didn't use the principles of object-oriented programming. I started programming the project with a single wrench tool, which controlled everything in the scene. I developed the application like this until the structure building worked, but at that point I realized I cannot continue programming that way, because the code got extremely complicated, and even very small modifications could totally ruin its functionality. So, I reworked the whole code with an object-oriented mindset.

I tried to reuse as much code as possible, through inheritance. Also, I built up the new structure, so that not the scripts of the tools are controlling the other game objects, rather the scripts of the tools only call functions of the modified object, and then the selected object modifies itself based on the passed parameters. This way, the scripts of the tools are much simpler as they don't have to determine what type of object the user tries to interact with. The tool simply sees if the objects are interactable or not, and then the objects themselves decide what should happen.

4.3 Basic concept of the building blocks

4.3.1 Interfaces

First, I made several interfaces, which I later implemented into the classes. These are:

- IHoverable: All game object which can be interacted with implements this interface. It contains two boolean properties: one stores the state of being hovered at the moment, the other stores if the object is selected. Also contains four functions, one for the event when the user starts hovering, and one for the end of hovering, the other two are for the events when the object gets selected or deselected.
- IDeleteable: This interface is for all objects that can be deleted from the blackboard. Only contains one Delete function.
- ITrayInteractable: All game objects on the tray inherit this interface. It contains a function that returns a GameObject, whose value is prefab object of the newly selected tray element.
- IMoveable: It has a property called moving, which stores if the object is moved at the moment. Also, it contains three functions. One which is called at the start, one at end of the moving of an object. The third one will be called every frame while the object is moved along.
- IOverplaceable: This interface is inherited by the object types which can be placed on the joints (trusses and forces). It has four functions, three are similar to the functions of the IMoveable interface's functions just for placing the objects. The fourth one can be called to redraw the object.

4.3.2 BaseCustomRayInteractable class

I have an abstract base class called BaseCustomRayInteractable, which inherits from the IHoverable interface, and Unity's Monobehaviour. The Monobehaviour class comes with three very important Unity specific functions [14]:

- Awake: This function is called before anything has happened.
- Start: This function is called before the first frame.
- Update: This function gets called in all frames.

The BaseCustomRayInteractable class is for all the objects which can be interacted with a pointer. It is called a RayInteractable, because the interaction is implemented via ray casts. This base class has:

- An Outline type variable, which will store the Outline component of the game object that the script is attached to.
- Two boolean type properties:
  - IsHighlighted, for storing if the object is highlighted at the moment.
  - IsHovered, for storing if the object is hovered at the moment.
- Some basic functions, which are all virtual, so they can be overridden in the derived classes:
  - For initialization:
    - Awake function: To initialize the Outline variable.
    - Start function: To set the default value of the properties, and to disable the Outline at start.
  - For the implementation of the hovering:
    - HoverEnter: This function is called if the user hovers over the object. It turns on the outline.
    - HoverExit: This function is called if the user stops hovering over the object. It turns off the outline, unless the object is selected.
    - SelectObject: This function is called if the user selects the object. It turns on the outline until the object is deselected.
    - DeselectObject: This function is called if the user deselects the object. It turns off the outline, unless the object is hovered.

  Note: These functions store the state of the highlighting in the IsHighlighted and IsHovered properties.

4.3.3 BuildingBlock class

The BuildingBlock class inherits from the BaseCustomRayInteractable class, and implements the IDeleteable and IMoveable interfaces. It is the base class for all the objects that can be placed on the blackboard. It has the following properties:

- moving: A boolean property, which stores if the object is being moved at the moment.
- information: A string property, which stores the text that will be displayed on the in-game display, if the object is selected with the screwdriver.
- neighbours: A list of GameObjects, which stores the objects this game object connects to.

And it has the following functions:

- Awake, and Start function, which expand the functions of the base class, to initialize the new properties too.
- Update: If the moving property is true, it calls the UpdateMovement function every frame.
- UpdateMovement: This function ray casts at the direction of the pointer, and if it hits the blackboard, it calls the Move function with the hit position, and the input of the joystick.
- Move: This function will move the object, but here it is an abstract function, that all children must implement in its own way.
- MovingStart: This function is called when the object is selected to be moved. It sets the moving property to true.
- MovingEnd: This function is called when the object is deselected. It sets the moving property to false.
- AddNeighbour: This function adds a game object to the neighbours list.
- RemoveNeighbour: This function removes a specific game object from the neighbours list, if it contains that.
- DeleteObject: This function deals with the event when the object is deleted. However, here it is an abstract function, as all children must implement it in their own way.
- SnapAngle: This function snaps an arbitrary 2D vector to the nearest 90-degree angle and returns the snapped angle.

- SnapDirection: This function is almost the same as the SnapAngle function, but it snaps the vector to the nearest 45-degree angle and returns the snapped direction as a 2D vector.

Note: The Snap functions are used, when the user wants some object to look at a specific direction. For example, to place a roller node, that moves on the y-axis, or if the user wants a truss to be exactly horizontal or vertical.

- GetOriginalMaterial: This function's job is to store the original material of the object in a list made of MaterialObjectInfo type variables, which is a struct I specifically made for this purpose.

Note: A MaterialObjectInfo type variable stores a material and a game object variable pair.

- SetOriginalMaterial: This function restores the original materials from the list made by the GetOriginalMaterial function.

- SetMaterial: This function's input is a material, and it sets that material to the object and its children.

Note: The last three functions are needed when the deformed shape is displayed. In that case the original objects turn grey, and the original materials are needed when the user gets back to editing mode by releasing the screwdriver, to color back the structure.

4.3.4 The classes that inherit from BuildingBlock

All the different objects that can be placed on the blackboard have its own class, which inherits from the BuildingBlock class, but not directly. There are some classes between the BuildingBlock and the final classes, which are GeneralPlaceable, the base class for all the different types of nodes and GeneralOverPlaceable, the base class of the truss and force classes. (It is called GeneralOverPlaceable, because the truss and force can only be placed over an already existing node.)

They are still abstract classes, but they also implement some of the abstract functions that are still not implemented in the BuildingBlock class.

GeneralPlaceable defines a new function called Dof, which returns a boolean array. This function will be used at the evaluation, to get the boundary conditions of the nodes. (If the node is fixed in a direction or not.) One value for x and one for y translation. The functions that are overridden:

- Move: It checks if the joystick is pushed in any direction, and if it is then the object is rotated in that direction. Furthermore, the Move function sets the position of the

object to the hit point, and calls the Redraw function of all of its neighbours. This is needed, because the neighbours must be updated, when a node is moved.

- DeleteObject: This function calls all of its neighbours' DeleteObject functions, and then it destroys its own game object. (If a node is deleted, all the trusses and forces connected to it must be deleted too.)
- AddNeighbour: This function adds a game object to the neighbours list, and then calls the added object's AddNeighbour function with itself. This is needed because neighbouring is a two-way relationship.

GeneralOverPlaceable defines one more boolean property called placing, that stores if the object is being placed at the moment. It overrides these functions:

- Awake: The Awake function is expanded to initialize the placing property.
- Update: The Update function is expanded to call the UpdateBuilding function, if the placing property is true.
- AddNeighbour: This function simply adds the game object it receives as a parameter to the neighbours list.
- DeleteObject: This function calls the RemoveNeighbour function of all its neighbours with its own game object, and then destroys it.

Furthermore it defines some new functions:

- PlacingStart: This function is called when the placing of the object was started. Is sets up the variables for the placing action, and turns on the outline of the object.
- PlacingEnd: This function is called when the user finishes placing the object. Here it is just an abstract function.
- UpdateBuilding: This function is called every frame while the object is being placed. It works similarly to the UpdateMovement function, but it calls the Build function, if the ray cast hits the blackboard instead of the Move function.
- Build: This function takes two 3D vectors as an input (the desired start and end point of the object that is placed), and it moves the object between these two points. But here it is just an abstract function.
- Redraw: This function is to update the position of GeneralOverPlaceable objects. This function is abstract at this level too.

4.3.5 The final classes of placeable objects

The classes that inherit from the GeneralPlaceable class are:

- FreeNode: This class is for the free nodes. Its Dof function returns a boolean array, whose values are all true.
- SemiFreeNode: This class is for the semi-free (roller) nodes. Its Dof function returns with a true and a false value. The order depends on whether the node can roll along the x or y axis.
- FixedNode: This class is for the fixed nodes. Its Dof function returns with two false values.

Note: All of them override the inherited Dof function. Also, the SemiFreeNode and the FixedNode classes override the GetOriginalMaterial function, because the original definition does not store the materials of the child objects, but these game objects have child objects, whose materials must be stored too.

The Force and Truss classes inherit from the GeneralOverPlaceable class.

- Force: This class implements a new property called size, which only has a getter function and returns the z scale of the shaft of the Force object. Also, it has a boolean variable called snap, which stores if the user has used the joystick to snap the direction of the force to a 45-degree angle during the moving action. This is needed, because otherwise the user would have to hold the joystick until the force is placed, and in that case, when the moving action has ended, depending on which hand was used the user would immediately start moving or rotating in the 3D space.
  - The functions that are overridden:
    * Start: The Start function is expanded to initialize the snap property.
    * MovingStart: The MovingStart function is expanded to set the snap property to false.
    * MovingEnd: The MovingEnd function is expanded to set the snap property to false.
    * Move: This function checks if the joystick is pushed in any direction, and if it is, then turns on the snap mode. If the snap mode is off, then the Build function is called with the received hit point and the position of the first neighbour. If the snap mode is on, then the Build function is called with the position of the first neighbour and a point, that is as far from the first neighbour as the hit point is from the first neighbour, but in the snapped direction. But, if the joystick input is not big enough, then the SnapDirection function returns null. In this case, instead of calling the

Build function it is easier to call the SetMagnification function with the distance between the first neighbour and the hit point.

    * Build: First it sets the rotation of the force object to match the vector pointing from the start point to the end point. Then it sets the scale of the shaft to be the distance between the two points. Finally, it sets the position of the head and the collider to the end point. Also, when setting the position of the object it uses the OffsetPosition function, to bring the force a bit towards the user.
    * PlacingEnd: This function resets the outline of the object, and sets the placing property to false.
    * GetOriginalMaterial: This function is overridden, because the force object has child objects, whose material must be stored too.

  – It also has two new functions:

    * SetMagnitude: It is like the Build function, but it only changes the magnitude of the force. It changes the scale of the shaft, and then adjusts the position of the head and the collider.
    * OffsetPosition: This function sets the input 3D vector just a bit forward, and returns it. It is used to offset the position of the force object, so that in the 3D world it does not overlap with the truss objects.

- Truss: This class only overrides some functions. These functions are:

  – Move: If the joystick is pushed in any direction, then the object is rotated to be parallel with the snapped direction. Also, it calls its second neighbour's Move function, with the point where it should be after the rotation.

  – Build: Sets the position of the truss to the position of the first neighbour, sets the rotation of the truss to look from the first neighbour to the second, and sets the scale of the truss to be the distance between the two neighbours.

  – Redraw: Calls the Build function with the position of the first neighbour and the position of the second neighbour.

  – PlacingEnd: Resets the outline of the object, and sets the placing property to false. Furthermore, checks if when the placing has ended the pointer pointed to a node object. If it did, then it calls the AddNeighbour function of the node object with itself, and calls the redraw function of itself. Otherwise the placing failed and it deletes itself.

4.4 The base class of the tray elements

The base class of the tray elements is called GeneralTrayElement. It inherits from the BaseCustomRayInteractable class, and implements the ITrayInteractable interface. There

are two variables, one stores if the actual type is the selected one, and a GameObject variable, that can be set from the editor. It stores the reference to the corresponding prefab, that will be placed on the blackboard if this is the active object type. It only has a basic Start function for the initialization.

4.4.1 The final classes of the tray elements

Two classes inherit from the GeneralTrayElement class:

- TrayPlaceable: For the different types of nodes on the tray.
- TrayOverPlaceable: For the truss and force objects on the tray.

These classes are very similar to each other. They both expand the inherited Start function, with gathering all the objects in the scene, that are of the same type as they are. This is key to the functionality of selecting only one element of a type at a time on the tray. They also implement the GetSelectedObject function, whose return value is the prefab object (the aforementioned variable that has to be set from the editor) of the selected type. Also, this function makes sure, that only one object of the same type is selected at a time by setting the selected property of the other objects to false.

4.5 Basic concept of the tools

There are two tools. Both of them have a specific class in their scripts specifying their behavior. They have the same base class CustomRayInteractorForGrabbable. It has only one property called inhand of boolean type, which stores if the tool is in the user's hand at the moment. Also, it has some variables that can be configured from the editor, from which the most important is the transform property of the origin of the ray casts.

The CustomRayInteractorForGrabbable class has the following functions:

- Awake: The Awake function initializes the used variables, and finds all tools in the scene.
- Start: This functions sets up the events of the XRBaseInteractable to trigger the Grabbed, UnGrabbed, Select and Deselect functions. Grabbed and Select functions will be called when the user picks up the tool, UnGrabbed and Deselect functions will be called when the user releases it.
- Grabbed: This function sets the inhand property to true.
- UnGrabbed: This function sets the inhand property to false.
- Select: This function sets up the other functions, to be called when specific actions are performed on the controller's buttons. (When they are pushed or released.)

- Deselect: Basically resets everything that was set up in the Select function.
- Update: If the tool is active the Update function ray casts in the direction of the pointer every frame and calls the HoverEnter and HoverExit functions of the objects, that were hit. This is how the hovering is implemented.
- SendHaptics: This function sends haptic feedback to the controller. It is called every time if an action is performed.

Note: The next functions each correspond to a specific button action on the controller. They are all abstract functions at this level.

- SecondaryActionStarted: This function is paired with the secondary button press on the controller.
- TriggerActionStarted: This function is paired with the trigger button press on the controller.
- TriggerActionEnded: This function is paired with the trigger button release on the controller.
- PrimaryActionStarted: This function is paired with the primary button press on the controller.
- PrimaryActionEnded: This function is paired with the primary button release on the controller.

Two classes inherit from the CustomRayInteractorForGrabbable class:

- WrenchClass: This class, as the name suggests is for the wrench tool.
- ScrewdriverClass: This class is for the screwdriver tool.

The WrenchClass has some additional LayerMask type variables, that can be set from the editor. These are used to filter the objects, that the wrench interacts with. But, it has no additional functions, only implements the remaining abstract ones:

- SecondaryActionStarted: This function makes a ray cast too, then if it hits an IDeleteable object calls the DeleteObject function of the object that was hit.
- TriggerActionStarted: A ray cast is performed, and if it hits:
  - A node object, it starts moving the object, by calling its MovingStart function.
  - The blackboard, it instantiates a new node object of the active type (on the tray), at the hitpoint.
- TriggerActionEnded: If any object was moved, it calls its MovingEnd function.

- PrimaryActionStarted: A ray cast is performed, and if it hits a node object, it places the active overplaceable tray element, and calls its PlacingStart function.
- PrimaryActionEnded: If any object was under placing, it calls its PlacingEnd function. If the object was placed successfully, it calls the SendHaptics function, for haptic feedback.

The ScrewdriverClass has much more going on. The Update function is modified, so the tool can only interact with the objects that represent the result. This is simply implemented by setting the tag of the result objects to "result", and then checking the tag of the hit object in the Update function. The other functions are:

- Select: The Select function is expanded. It collects all the placeable objects in the scene and initializes a StructureSolverAndAnalyzer type variable with them. It creates a BackgroundWorker, so the calculations can be done on a separate thread. This way the application does not freeze while the calculations are running. After setting up the BackgroundWorker's main function to be the bgw_DoWork function, and the bgw_RunWorkerCompleted function to be called after the calculations are done, it starts the BackgroundWorker. More on the calculator later. Also, it sets up the new SecondaryActonEnded function to be triggered by the secondary button release.
- Deselect: The Deselect function is expanded too, as this function has to reset what the expanded Select function did.

Note: The next functions which are triggered by the controller's buttons can only affect objects, with the tag "result".

- TriggerActionStarted: A ray cast is performed, and if it hits a result object, its information is displayed on the in-game display.
- TriggerActionEnded: This time it has no real functionality, but it is needed to be implemented, because the base class has it.
- PrimaryActionStarted: First this function finds the minimum and maximum stress values in the structure, then it calls all the element's ShowStress function with the found values. Also, writes out on the display the minimum and maximum stress values, and calls the colormap's SetValues function, with the minimum and maximum values.
- PrimaryActionEnded: Calls each element's EndShowStress function, to set their color back to the original. Also, calls the colormap's Erase function.
- SecondaryActionStarted: This function is almost the same as the PrimaryActionStarted function, but it calls the ShowAbsoluteStress function of the elements, and calls the colormap's SetValues function only with the maximum absolute stress value.

- SecondaryActionEnded: Without any real functionality it is just like the PrimaryActionEnded function.

Note: The next functions are all new functions, that are not in the base class.

- bgw_DoWork: This function runs on a separate thread. It calls the StructureSolverAndAnalyzer's Solve function.
- bgw_RunWorkerCompleted: This function runs after the DoWork function is done. It calls the BuildDeformedStructure function.
- TurnGrey: This function turns the structure grey, by calling the SetMaterial function of each element.
- BuildDeformedStructure: This function builds up the deformed structure. It calls the CreateInstance function of all the separate parts of the structure, with the magnification and a z value, that is a bit forward from the original structure in the global coordinate system. (This way the deformed structure does not overlap with the original one. The magnification is needed to make the deformations visible.)
- SetMagnification: This function sets the value of the magnification variable for the visualization of the deformations.

4.6 The solving and storing of the structure

I have spent a lot of time trying to figure out how to do this part of the application. The main problem came from the fact that, when the structure is built up and ready to be evaluated, all the information that is needed to execute the finite element analysis is scattered among all the objects on the blackboard. At the end, I settled on a method, where I collect all the different types of placeable objects into lists of their types, and then convert their information into some custom classes only developed for the ease of the calculations. These classes are the Node2D, TrussElement2D and Force2D. I made an interface called IInstantiable that they all implement. It has three functions:

- CreateInstance: This function will be used to build up the deformed structure. And it has a GameObject return type, as it will return a reference to the instantiated gameo bject.
- DataToString: This function returns a string with all the data about the object. This string will be displayed on the in-game display when an object is selected with the screwdriver.
- RoundToThreeSignificantFigures: This function rounds the input value to three significant figures. It is used, when the DataToString function composes the string. This function is defined here, so all child classes can use it.

I also created two enums, to make the code more readable:

- ForceType: This enum can have two values, Outer and Reaction. The Outer value is for the forces that are applied to the structure by the user. The Reaction value is for the forces exerted by supports in response to loads applied to the structure.
- BoundaryCondition: This enum can have four values, Free, xRoller, yRoller and Pinned. It is for the boundary conditions of the nodes.

The implementation of the classes:

- Node2D:
  - The Node2D class has the following properties:
    - boundaryCondition: Stores the boundary condition of the node in a BoundaryCondition type variable.
    - X: Stores the x coordinate of the node.
    - Y: Stores the y coordinate of the node.
    - xDisplacement: Stores the x displacement of the node, after the calculations.
    - yDisplacement: Stores the y displacement of the node, after the calculations.
    - displacement: This property has only a getter function, which returns a 3D vector with the x and y displacements of the node. (The z value is 0.)
  - The Node2D class has the following functions:
    - position: This function returns a 3D vector with the x and y coordinates, of the node. The z value is set to 0, unless it is provided as an input for the function. (This is possible with operator overloading.)
    - CreateInstance: This function instantiates a node prefab object, whose type depends on the value of the boundaryCondition. It is created at the point that the position function returns, plus the magnification times the displacement vector. The tag of the instantiated object is set to "result", and its information is set to the return value of the DataToString function.
    - DataToString: This function returns a string with the data of the node. It is formatted to be displayed on the in-game display.
- TrussElement2D:
  - The TrussElement2D class has the following properties:
    - Node1: Stores the reference to the first Node2D object, that the truss connects to.

    * Node2: Stores the reference to the second Node2D object, that the truss connects to.
    * YoungsModulus: Stores the Young's modulus of the truss. The default value is 210 GPa.
    * CrossSectionalArea: Stores the cross-sectional area of the truss. The default value is $1 \cdot 10^{-5}$ m^2.
    * Length: Stores the length of the truss. This is basically the distance between the two nodes.
    * Stress: Stores the value of the stress in the truss. It gets its value after the calculations.
    * Strain: Stores the value of the stress in the truss. It gets its value after the calculations too.
  - The TrussElement2D class has the following functions:
    * CreateInstance: This function instantiates a truss prefab object between the two nodes. The tag of the object is set to "result", and its information property is set with the return value of the DataToString function. A reference of the created object is stored in a variable, so that later modifications can be done on it. (For example, when changing the color.)
    * ShowStress: This function changes the color of the truss object to represent the stress in the truss. This is done by first initializing a new white material then changing the color of the material to the desired one, depending on the stress of the truss, and the maximum and minimum stress values in the whole structure. After that, the material is set to the truss object with the SetMaterial function.
    * ShowAbsoluteStress: This function operates the same way as the ShowStress function, but it uses the absolute value of the stress and a different color range.

    Note: When the color of the trusses change the program has to determine appropriate color, that should be displayed. For this I used HSV color space, because here the hue value is the only thing that has to be changed. Simply, the program sets the hue value between two predefined values on a linear scale at the same place where the stress value is between the maximum and minimum shown stress values.
    * EndShowStress: This function sets the material of the truss object back to the original.
    * DataToString: This function returns a string with the data of the truss. It is formatted to be displayed on the in-game display.

- Force2D:
  - The Force2D class has the following properties:

        * forceType: Stores the type of the force in a ForceType type variable.
        * node: Stores the reference to the Node2D, where the force is applied.
        * strength: Stores the magnitude of the force.
        * XForce: Stores the x component of the force.
        * YForce: Stores the y component of the force.

      Note: The strength property is calculated from the XForce and YForce properties. If any of the XForce or YForce properties is changed, the strength property is automatically updated too.

        * direction: This property has only a getter function, which returns a 3D vector with the x and y components of the force. (The z value is 0.)

    - The Force2D class has the following functions:
        * DataToString: This function returns a string with the data of the force. It is formatted to be displayed on the in-game display.
        * CreateInstance: This function instantiates a force prefab object. It is created at the point, that the position function of the node returns plus the displacement of the node times the magnification. The direction of the force is set to match the direction property.

The TrussStructure2D class contains the calculator in the application. Let's see its structure:

- The TrussStructure2D class has the following properties:
    - nodes: A list of Node2D objects.
    - trusses: A list of TrussElement2D objects.
    - forces: A list of Force2D objects.
    - globalDisplacementVector: A double array, that stores the displacements of the nodes after the calculations.
    - globalForcesVector: A double array, that stores the forces acting on the structure after the calculations.
- The TrussStructure2D class has the following functions:
    - FixedDofs: This function returns a boolean array, which stores the boundary conditions of the nodes. It gives false if the node can move in the direction, and true if it is fixed in that direction.
    - LUdcmpMethod: This function separates the global stiffness matrix into a lower and an upper triangular matrix, using the LU decomposition method.
    - FwdSubstitution: This function solves the equation system with the forward substitution method.

  - BwdSubstitution: This function solves the equation system with the backward substitution method.
  - FillValues: This functions adds the reaction forces to the forces list, and calculates the stresses and strains for the TrussElement2D-s in the trusses list.
  - MergeForceVectors: This function merges the applied forces into a single vector, that can be used in the calculations.
  - CalculateStiffnessMatrix: This function calculates the stiffness matrix of a single truss element.
  - MergeStiffnessMatrices: This function calls the CalculateStiffnessMatrix function for all the trusses in the trusses list, and merges the results into a single global stiffness matrix.
  - CondenseStiffnessMatrix: This function as the name suggests condensates the global stiffness matrix, by removing the rows and columns that correspond to the fixed degrees of freedom.
  - CondenseForceVector: This function condensates the global force vector, by removing the rows that correspond to the fixed degrees of freedom.
  - SolveLinearSystem: This function solves the linear system of equations, with the LUdcmpMethod, FwdSubstitution and BwdSubstitution functions.
  - MultiplyMatrixVector: This function multiplies a matrix with a vector, and returns the result. It is used to get the full force vector.
  - GetFullDisplacementVector: This function expands the results of the calculations to form the structures full displacement vector.
  - Solve: This function is the main function of the class. It calls all the other functions in the right order, to solve the structure. It follows the steps described in Chapter 3.1, with the only difference, that instead of inverting the stiffness matrix, it uses the LU decomposition method from Chapter 3.3.

The main purpose of the StructureSolverAndAnalyzer class is to set up a TrussStructure2D object, from the received lists and to be ready to call its Solve function. It has the following structure:

- The StructureSolverAndAnalyzer class has the following properties:
  - structure: A TrussStructure2D object.
- The StructureSolverAndAnalyzer class has the following functions:
  - GetIsolatedNodes: This function returns a list of Node2D objects, that are not connected to any truss element.
  - DeleteNodes: This function deletes the nodes in the input list from the structure's nodes list.

- PopulateStructure: This function converts the information of objects in the 3D space to useable data for the calculations. After the function ran, the nodes, trusses and forces lists are filled with the converted data.
- Solve: This function calls the Solve function of the structure object.
- The constructor of the class: Here I have to mention this too. Its inputs are three lists, one with all the objects placed on the blackboard. The constructor deletes the isolated nodes from the structure by calling the DeleteNodes function with the return value of the GetIsolatedNodes function. After that, it calls the PopulateStructure function.

4.7 The colormap, the slider and the tool belt

All of these objects have their own classes. But before I go into the details of these classes, I have to write about the UI elements in Unity. UI elements are different from the 3D objects, as they are only two-dimensional. This is why they can only be placed on a Canvas object. There are many ways someone can set up a Canvas, but I used them in world space. This way the canvas can be placed in the 3D world. This is how I could achieve for example, to display information on the in-game display.

The description of the classes:

- Colormap:
  - This class inherits from Unity's Monobehaviour class and has no properties. Only some variables storing the UI elements that are needed to be set from the editor.
  - The Colormap class has the following functions:
    - Start: This function only calls the Erase function, to set everything to the default state, which is invisible.
    - SetValues: This function is operator overloaded, it can be called on two double values, or just on a single one.
      - If it gets two values, then it enables and sets the minimum and maximum values of the colormap to the input values. Furthermore, if the value zero is between the minimum and maximum values, then it enables the zero stress marking too, and places it at the right position on the color map.
      - If it gets only one value, then it enables and sets the maximum value to the input value, and also enables the zero stress marking at the beginning of the colormap.
    - Erase: This function sets all the elements of the colormap to be invisible, by disabling them.

- Slider:
  - This class inherits from Unity's Monobehaviour class and has no properties. It has some variables that have to be set from the editor: the rail's transform, the UI text element and the reference to the screwdriver's game object.
  - The Slider class has the following functions:
    * Awake and Start: These functions set up the events of the XRBaseInteractable to trigger the HoverEnter, HoverExit, OnGrab and OnRelease functions. Also, they set the slider to its default position and the text to display the default value.
    * Update: If the slider is grabbed, it updates the value of the slider, also checks if the position of the marking cube is in the bounds of the rails. If it is not, then it sets the value of the slider's position to the nearest bound. In the end it updates the text to display the value of the slider.
    * OnHoverEnter: This function enables the outline component.
    * OnHoverExit: This function disables the outline component, if the slider is not grabbed.
    * OnGrab: This function sets the slider's material color to red, and sets the grabbed variable to true.
    * OnRelease: This function sets the slider's material color back to the original, and sets the grabbed variable false. Also, sets the magnification value of the screwdriver's class to the value of the slider, and starts the WhenDisable coroutine.

  Note: IEnumerators are functions that can pause their execution between frames, and then continue where they left off at the next one, or wait for a specified amount of time. They are very useful when dealing with time dependent tasks, like animations. IEnumerators can be called with the StartCoroutine function by giving them to it as a parameter.
    * WhenDisable: This function is an IEnumerator with a 3D vector input, the position where the knob of the slider should be moved to, when it is released. Basically, it creates a smooth animation, where the knob returns to its place on the rail in 0.5 seconds with a constant speed.

- ToolBelt:
  - This class inherits from Unity's Monobehaviour class and has no properties. It has some variables, that have to be set from the editor: two lists, one for the tools and one for the slots where they can be stored, and a reference to the main camera.
  - The ToolBelt class has the following functions:
    * Awake: To set up some variable to be used later.

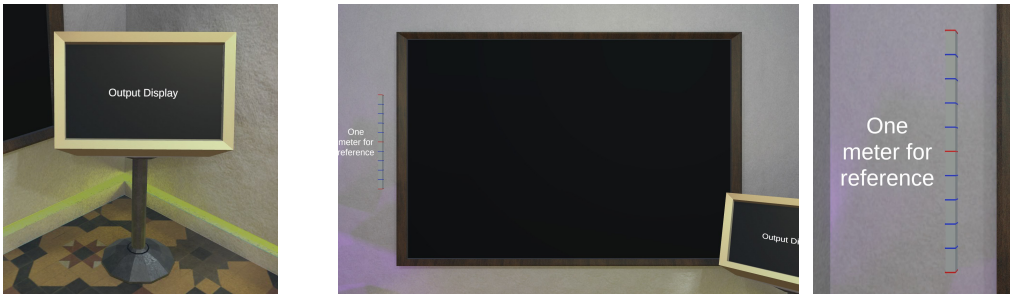
* Start: Calls the AlignToolCoroutine function.
* Update: Checks if the in hand properties of the tools are true, and if it is not, then it places them back to their slots. It is important to call this function every frame, because the tool belt moves with the user, and the tools must move with it.
* AlignToolCoroutine: I had to implement this IEnumerator, because sometimes I got a bug, when the forward direction of the tool belt did not align with the forward direction of the camera. This function aligns the forward direction of the tool belt with the forward direction of the camera after waiting for one second.

5 Using the application

5.1 The scene

As I wrote it before, there is only one scene in the application. This 3D space is where everything happens. From the building of the structure to the evaluation of the results. Useful fix objects in the scene:

- Display: The properties of specific elements will be displayed on the display during the evaluation.
- Blackboard: This object serves a role very similar to a real life blackboard in a classroom. Everything will be drawn on this.
- Meter Stick: It is in the scene, so the user has some size reference. It is exactly 1m long, and has marks every 10 centimetres.



(a) **(b)** **(c)**

Figure 5.1: Non movable, but useful objects. The ingame display (a), the blackboard (b), and the meter stick (c).

Now let's see what interactable objects can be found here:

- Wrench: With this tool the user can build up the structure.
- Screwdriver: The screwdriver is for the evaluation of the structure, to display the properties of individual components of the deformed structure, and to show the stress field.
- Tray: For all the objects that can be placed on the blackboard, a sample takes place on the tray. This serves the purpose of choosing the active building elements.
- Magnification slider: With this slider it is possible to set the magnification multiplier for the displacement.

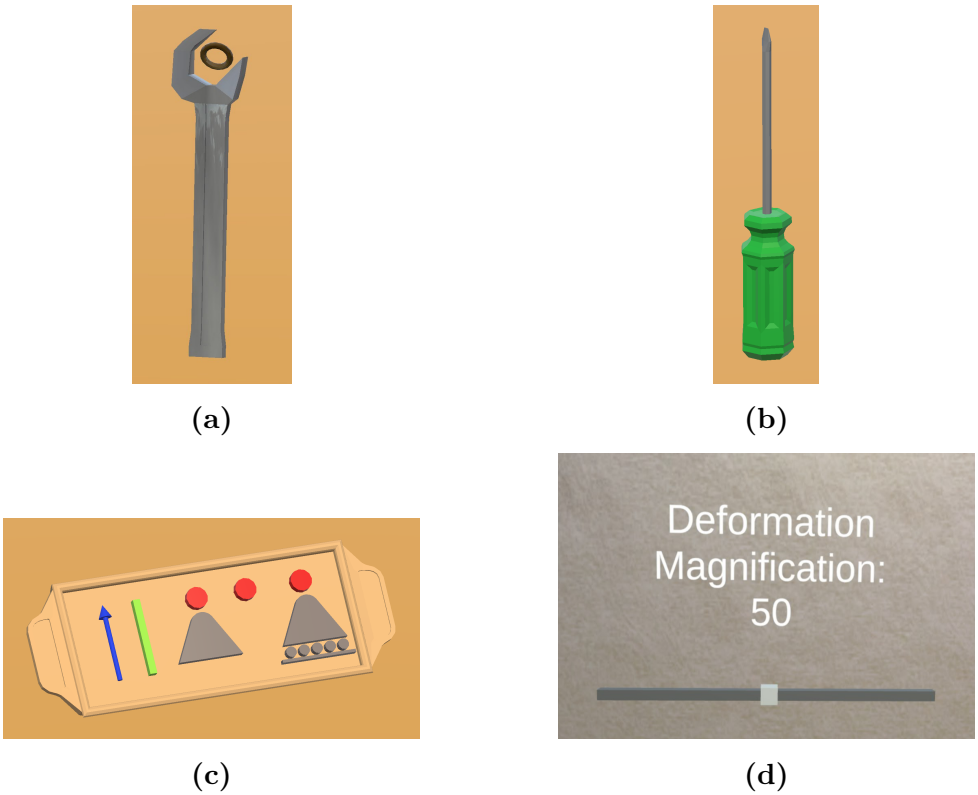


Figure 5.2: The interactable objects. The wrench (a), the screwdriver (b), the tray (c), the slider (d).

5.2 Types of placeable objects

The placeable objects can be divided into two groups. A group with the objects that can be placed independently to any other object, and another group in which they can only be placed on objects from the former group. The first set contains the three types of the nodes, shown in Figure 5.3:

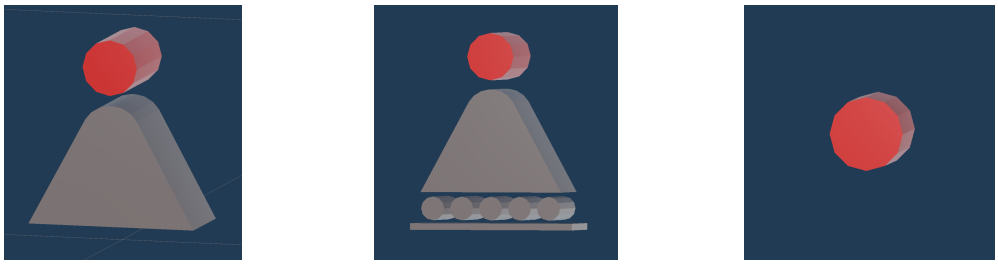
- Pinned node: No displacement is allowed.
- Roller node: Displacement is only allowed in the x or y direction.
- Free node: The structure can move in any direction.

All type of nodes have a red cylinder. The rotation axis, around which the nodes can turn is at the symmetry axis of the cylinders. Also, the nodes can be selected via this red cylinder only.

In the other group there are the objects shown in Figure 5.4:

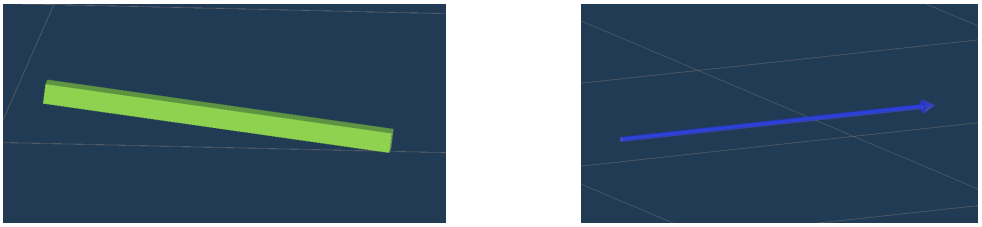
- Truss: This is a green square prism, that represents the connection of two nodes with a truss. It can be selected at the middle third of its length.

- Force: This is a blue arrow that's length represents the magnitude of the force acting on a node. Forces can be selected at the tip, around the arrow.



(a) **(b)** **(c)**

Figure 5.3: The types of nodes. Fixed node (a), roller node (b), and free node (c).



(a) **(b)**

Figure 5.4: The truss (a), and the force (b) objects.

5.3 The wrench tool

When the user takes up the wrench the pointer of the object will activate itself. This pointer is used for all the actions that can be performed with this tool. A picture of the wrench in hand can be seen in Figure 5.5.

5.3.1 Using the tray

All the mentioned placeable objects can be found on the tray. All the time there are two active elements, one of each from the aforementioned groups. The active element can be changed if the user points at the object that they want to use, and then pushes the trigger button. After this happened the new object will be highlighted. It is possible to independently select the active element from the two groups. Also, the Tray can be grabbed too. A comfortable way of building is by holding the tray in one hand, while using the wrench with the other hand. A picture of the tray while the application is running can be seen in Figure 5.6.

Figure 5.5: The wrench in hand.

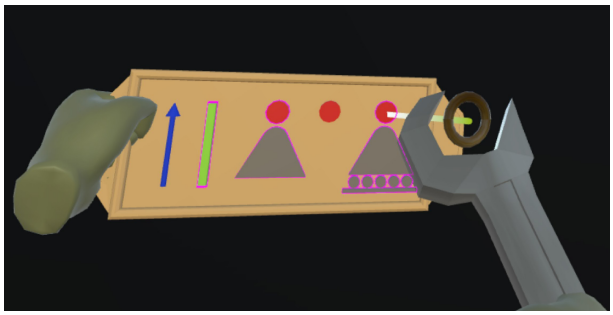


Figure 5.6: The tray in use.

5.3.2 Placing objects on the blackboard

If the user points to the blackboard and presses the trigger button, then the selected type of node is placed at the point where the pointer hits the blackboard. A Truss object can only be placed between two nodes, by hovering over one, pushing and holding down the primary button while moving the pointer to another node. During this operation thanks to some animation, for the user it seems like, that they are stretching the truss to reach the other node. Finally, when the button is released the Truss element is placed between the two nodes. If at the moment, when the button is released the pointer is not pointing to any node, the placing fails and the Truss is not going to be placed. Forces can only be placed on nodes too, with a similar method to the placement of trusses. The only difference is that the placement does not have to end on another node. Always the selected object (Truss or Force) is placed with the primary button.

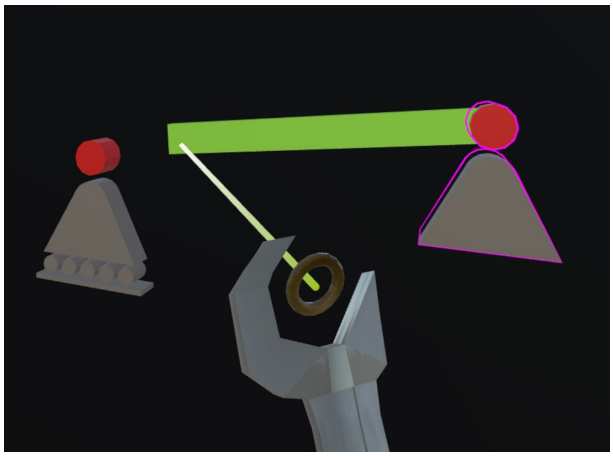


Figure 5.7: Placing a truss

5.3.3 Modifying the already placed objects

If the user hovers over an already placed object with the pointer, it gets highlighted indicating that it's ready for the modification. It always starts with pushing, and holding down the Trigger button, but different objects can be modified in different ways:

- Node: As pointer is moved, the node is moved on the blackboard. Also, all the attached Trusses and Forces are moved with the node. Another functionality is implemented with the joystick. If it is pushed enough the program calculates the closest direction in a 90° division of the circle, and rotates the nodes around, to the desired direction. This functionality was included to be able to add a Roller node both in the x and y directions.

- Truss: This time nothing will happen if the pointer is moved, only the joystick has some effect. Here, the direction of the joystick is snapped to the closest direction in a 45° division of the circle. The selected Truss will turn in this direction around the node from which it was stretched out when placed. All the connected objects on the other side will update their positions based on the change.

- Force: The end of the force vector will move with the pointer. This way both the magnitude and the direction of the Force can be changed freely. But if the user rotates the joystick the direction of the Force will snap to the corresponding 45° division of the circle. Now only the distance of the pointer from the node matters, as the length of the vector will be equal to this, the exact location of the pointer has no effect. Once the joystick was pushed the only way to go back to free editing is to release the Trigger, and select the object again.

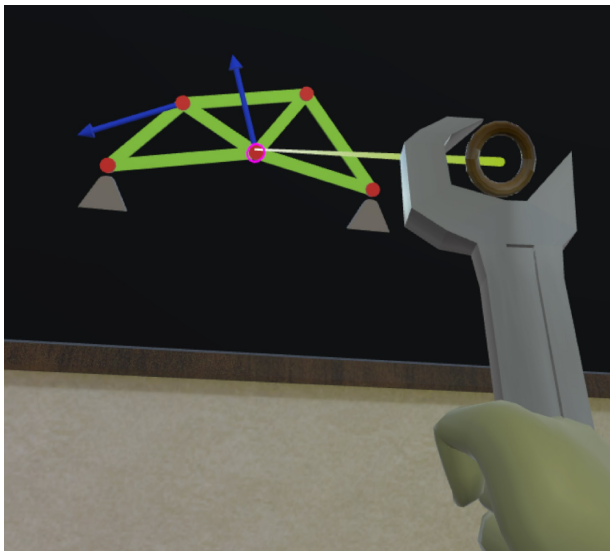


Figure 5.8: Moving a node.

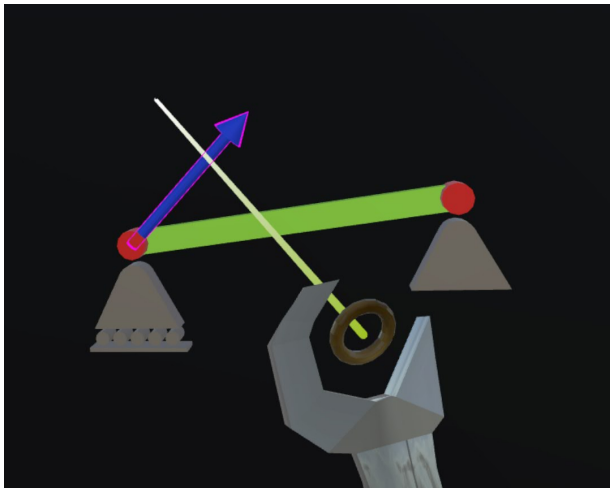


Figure 5.9: Snapping a force.

5.3.4 Deleting objects

Any object can be deleted, that has been placed on the blackboard. This can be done by hovering over them, and pressing the secondary button on the controller. In case of the Truss and Force, only the selected object is deleted, but if the user deletes a node, all the connected objects will be deleted too.

5.3.5 Example

In Figure 5.10 an example of a truss structure can be seen. This is, where the next step is to evaluate the structure.

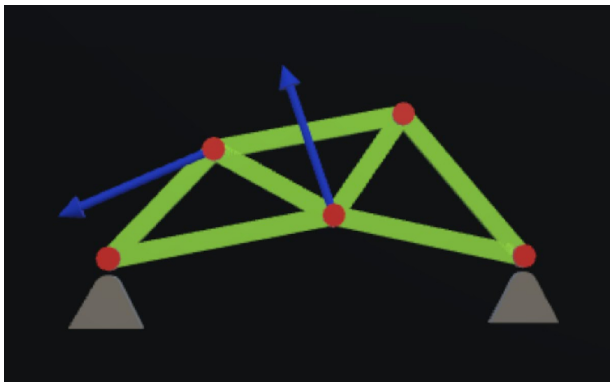


Figure 5.10: Example of a truss structure.

5.4 Using the screwdriver

This tool is used for the evaluation of the structure. When the user picks up the screwdriver the calculation automatically starts, and similarly to the wrench the pointer activates. The display will write out "Ready" when the program finished with the calculations. Here I have to mention, that the program can operate on any structure, that the user drew on the board, but results will only be true if the structure was statically determinate, or over constrained. The other cases are not handled yet, just to a point that the application does not freeze.

5.4.1 The way the results are displayed

After the calculation is done, the original structure turns grey, and the deformed one is displayed in front of the undeformed. New Forces are added to the structure, with an orange tint. They show the reaction forces acting on the structure. Figures 5.11 and 5.12 show an example of the deformed structure.

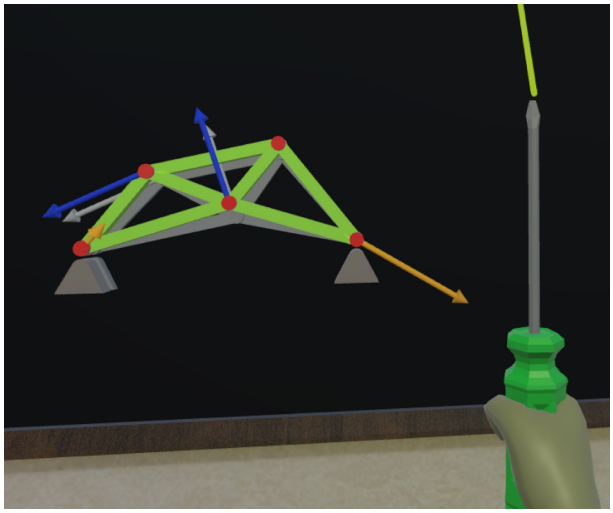


Figure 5.11: The structure after picking up the screwdriver.

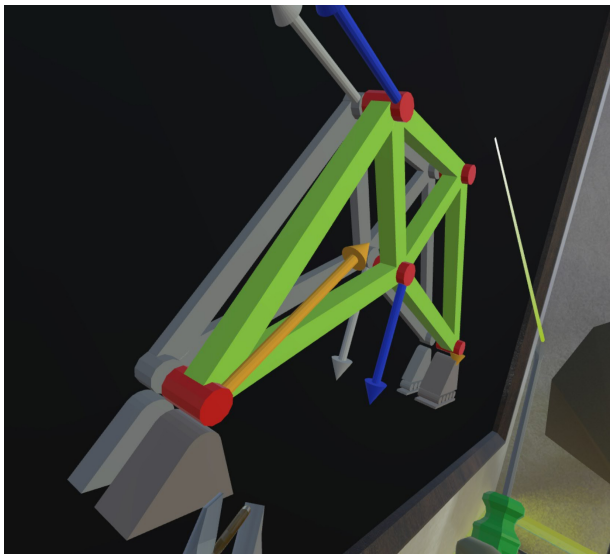


Figure 5.12: The deformed structure in front of the original.

5.4.2 Reading out the numerical results

If the user points to an object of the deformed structure, and presses the Trigger button, the information of the selected object will be written out on the display. This is different in all types of objects:

- Nodes: It writes out the location of the node, the boundary condition (fixed degrees of freedoms), and the displacement of the node if it is not pinned.
- Trusses: The display shows the length of the truss element, the physical parameters (cross-section area, and young's modulus), and the strain with the resulting tension.
- Forces: For the forces the display shows their overall magnitude, and also separately the x and y components.

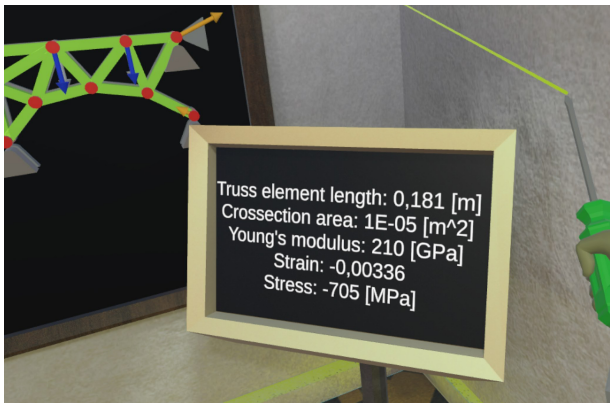


Figure 5.13: The display showing the properties of a truss.

5.4.3 Looking at the stress field

By holding down the primary button all the trusses change color according to their stress levels. Also, on the wall a color scale appears, with the minimal, and maximal stress shown at the two ends of the scale, and another label indicates the color corresponding to the zero stress level. In this mode, the actual value of the stress is considered with red color for compression, and blue color for tension (the color gradient is continuous from blue to red). An example of this functionality can be seen in Figure 5.14.

Figure 5.14: The structure showing the stress.

If the secondary button is held down, then the absolute value of the stress in the trusses is shown. Now the color gradient goes from green representing zero stress, to red representing the maximal stress value. An example of showing the absolute stress can be seen in Figure 5.15. After the user has looked at the results, they can take the wrench back in hand, modify the structure and see how the modifications effect the results using the screwdriver.

Figure 5.15: The structure showing the absolute stress.

5.5 The slider

The text of the slider represents the magnification of the deformations occurring while the user holds the screwdriver. The magnification is needed, because the deformations are so small, that without it, they would be barely visible. The value of the slider can be adjusted between 0 and 100. The user does not need any tool to adjust the value of the slider, simply the small cube can be grabbed and moved around until the text shows the desired value, then the cube can be released. An example of adjusting the slider can be seen in Figure 5.16.

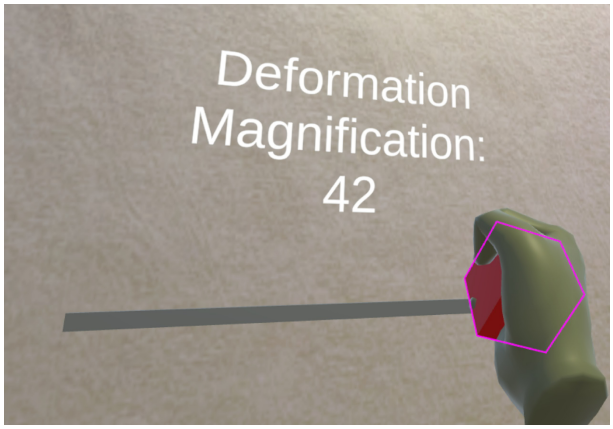


Figure 5.16: Adjusting the slider.

5.6 A bigger model

The results of a bigger model can be seen in Figure 5.17 and A.17.

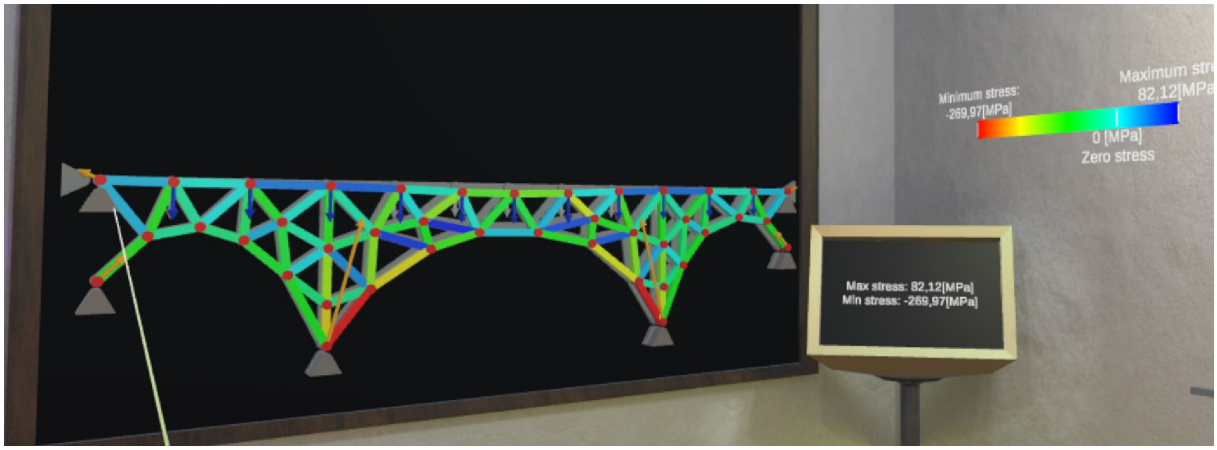


Figure 5.17: Stresses in the model of a bridge.

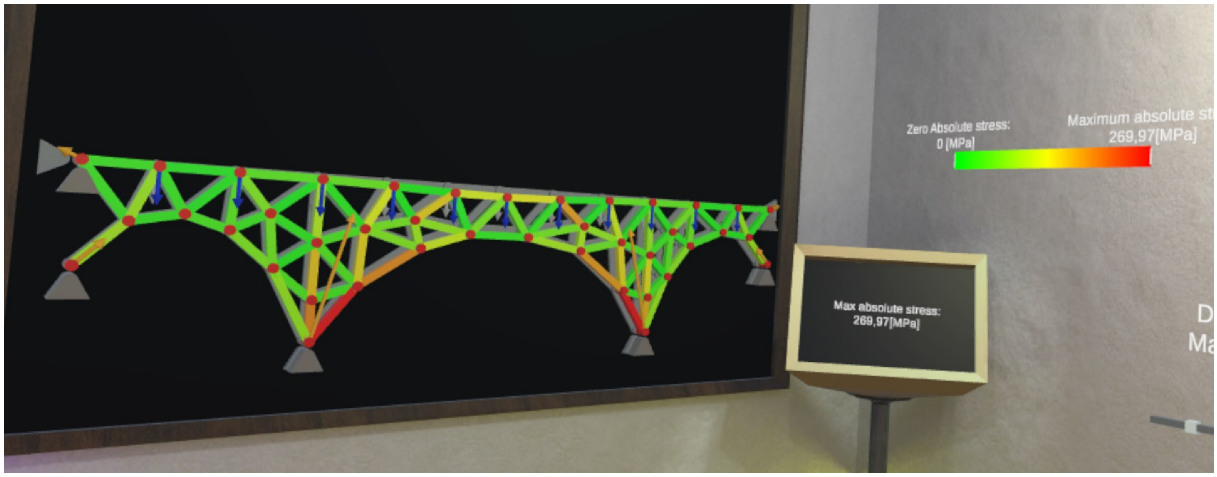


Figure 5.18: Absolute stresses in the model of a bridge.

6 Summary

6.1 Results

I could finish all the major tasks that I planned to. I gave my headset to multiple people to test my application, and after explaining the controls they could use it mostly without any help. The only thing that was a little bit confusing for the first time users was, when to use the trigger and primary buttons. Maybe I could have done a one button mode, where the user can only use the trigger to interact with the application. Selecting the action type that they want to perform, and then after all the work is done, selecting another action and so on. This way it would even possible to use only one hand, with a pointer. No tools needed. Otherwise, for the people who are used to VR, the controls seemed very intuitive.

The tools and the tray can be used with both hands. To make this possible I had to implement a system that can distinguish between the two controllers, so the program can take separate actions based on which controller is used. The wrench tool is used to build up the structure, and the screwdriver is used to evaluate it. I implemented a tool belt too, so every tool can be easily accessed at any time.

The class hierarchy that I developed can store all the parameters of the truss structure, and also the results of the calculations. These are not only the displacements of the nodes, but also the resulting strain and stress values in the trusses and the reaction forces. After the evaluation, the program can build up the deformed structure in the 3D space, with the added reaction forces from the data, that was stored in instances of the classes.

The application can be used to simulate the loading of truss structures, and to check the results of the calculations. The user can easily modify the structure, and the loads, to see how the results change. Also, the user can look at the parameters, and the result of each individual part of the structure, thanks to a text box that is displayed in the VR environment. The resulting stress field can be displayed on the deformed structure. This is implemented by changing the color of the trusses, based on the stress value. Also, a colormap appears with a scale to give meaning to the colors.

6.2 Conclusions

While I was working on the application I gained a deeper understanding of the Unity engine, and the steps of VR development. I learned how the interaction with objects is handled in VR, and how I can make my own ray interactor script from scratch. Along the way I also learned how to use Unity's new input system. When I was preparing the 3D models I learned how to use Blender for modelling and rendering too. And most importantly, how to export the models and baked in textures to Unity, with the correct scale, and orientation. Another big part of the development was to learn about C#

programming language, and how it can be used in Unity. I think I used almost all the features of the language somewhere in my application.

6.3 Outlook

The application is about a very broad topic, so it can be further developed in many ways. But the most important is to keep it user-friendly as it is for facilitating the learning process of the students. While making the application I tried to set up its structure, so that it can be easily extended with new features. I hope, that in the future other students will continue the development of the application. Some of my ideas for further development are:

- A major shortcoming of the application is that it can not decide if a structure is statically indeterminate or not. It would be useful, if the solver could decide if it is the case, and inform the user in some way, so they can modify the structure to make it statically determinate.
- Implementing more types of elements to extend the range of problems that can be handled with the application. For example by adding the beam element bendings could be calculated too. This would provide the opportunity to add new types of loads, like moments and distributed forces.
- Another step forward could be to extend the existing application, so it can handle 3D structures. A smaller part of this would be to extend the 2D classes to store the 3D data. The bigger part would be to implement a way the user can draw the 3D structure in the VR environment. I think this would make the application more immersive, and fun to use, as the user could walk around, or even inside the structure.
- The application could be extended with a way to save and load the structures. This way the user could save the structure, and the results of the calculations, and load it back later. This would be very useful for the students, as they could save their work, and continue it later. Also, this way there could be some educational premade structures, that the users could load, and experiment with.
- A mode could be added to the application, which would work like a bridge building game. There would be fixed nodes, and loads, that the user can not delete, and then with only using free nodes, the user would have to build a structure that does not exceed a given stress value anywhere, or the buckling of the trusses could be the limit. But for this to work the application somehow would need to implement second-order moment of different cross-sections too.
- Another way to develop further the application could be to make it possible to set the positions of the nodes and the magnitude of the forces more precisely. This way

any given structure could be simulated. For this to work a numerical input has to be implemented, for example in the form of a slider or a virtual keyboard.

Összefoglaló

A feladatom az volt, hogy fejlesszek egy VR alkalmazást, amivel szimulálni lehet rácsos tartók alakváltozásait erő terhelések hatására. Még nem született szakdolgozat VR technológiával kapcsolatban a tanszéken, de mivel egyre nagyobb teret nyer a világunkban, fontosnak tartom, hogy az egyetem is felvegye a lépést. Ezért a dolgozatnak az is célja volt, hogy mutasson egy alkalmazási példát, amire későbbi hallgatók is építhetnek.

Az alkalmazásban egy mutatóval ellátott villáskulcs szerszámmal lehet felépíteni a szerkezetet egy táblára. Le lehet helyezni csuklós támaszt, görgős támaszt és közönséges csuklót is. Ezeket lehet összekötni rudakkal. A rácsostartóra lehet terheléseket adni koncentrált erő formájában. Az alkalmazásban lehetőség van minden lehelyezett elem törlésére, és áthelyezésére is. A felhasználó a szerkezet felépítése után egy csavarhúzó szerszám kézbe vételével indítja el a numerikus megoldót. Az alkalmazás kiszámítja a deformációkat, fajlagos alakváltozásokat és az ezekből adódó feszültségeket a rudakban. Az eredményeket a felhasználó egy virtuális kijelzőn minden egyes lehelyezett elemre megtekintheti. Továbbá lehetősége van a feszültségmező megjelenítésére is a deformált szerkezetben színskála segítségével.

Az alkalmazás fejlesztése során kiemelt szempont volt, hogy egy könnyen megtanulható és magától értetődően használható felületet hozzak létre. Ez azért volt fontos, mert az alkalmazás elsősorban egy oktatási segédprogramnak készült, amivel a hallgatók könnyebben elsajátíthatják a rácsostartók számításának alapjait. Ennek érdekében olyanra készítettem az alkalmazást, hogy a megrajzolt szerkezetet az eredmények megtekintése után könnyen lehessen módosítani, így kísérletezve, hogy különböző változtatások hogyan hatnak a végeredményre. A feladat elkészítésénél figyeltem arra, hogy egy könnyen bővíthető struktúrát hozzak létre, hogy a jövőben más hallgatók is folytathassák az alkalmazás fejlesztését.

A feladat elvégzéséhez a Unity játékfejlesztő szoftvert használtam. Először felállítottam a VR környezetet, majd miután sikerült összekötnöm az Oculus Quest 2 headsetet a számítógéppel, elkezdtem a szoftverfejlesztést. Mielőtt még nekiállhattam volna kódolni, először meg kellett terveznem a program struktúráját, hogy minél több kódot tudjak többször is használni öröklésen keresztül. Ezt követően elkészítettem a 3D modelleket a Blender nevű szoftver segítségével. A modellek textúrázását is ebben a szoftverben végeztem el. A fejlesztés során készítettem egy saját ray interactor scriptet is, ami lehetővé teszi, hogy a felhasználó egy mutató segítségével a VR környezetben kapcsolatba tudjon lépni különböző tárgyakkal. Miután sikerült eljutnom oda, hogy fel lehetett építeni a szerkezetet, elkészítettem a numerikus véges elemes megoldót a programhoz. Itt a több numerikus módszer közül a felső és alsó háromszögmátrixra bontás módszerét választottam a kapott lineáris egyenletrendszer megoldására. Viszont nem volt elég a számolót elkészíteni, írnom kellett egy programrészt az adatok előkészítéséhez, és a megfelelő alakra hozásához. Ehhez létrehoztam egy osztály hierarchiát, amiben minden felhasznált építőelemnek megvan a maga osztálya, ahol el lehet tárolni. Miután végzett a program

a számításokkal felépíti a deformált szerkezetet a táblára. A számolt adatok alapján kirajzolva a reakcióerőket is. Még a feszültségek megjelenítését valósítottam meg, amit a rudak színének változtatásával értem el. Minden kódot C# nyelven írtam, mert a Unity főként ezt a programozási nyelvet támogatja.

References

[1] *AmbientCG* [n.d.].
URL: *https://docs.ambientcg.com/license/*

[2] Barnard, D. [2024], 'History of vr – timeline of events and tech development', *VirtualSpeech* .

[3] Castle, K. [2020], 'Oculus quest 2 review', *Rock paper shotgun* .

[4] Chunawala, Q. [2024], 'Fast algorithms for solving a system of linear equations', *Baeldung* .

[5] Ibrahim, M. S. [2021], 'Fealite2d'.
URL: *https://github.com/FEALiTE/FEALiTE2D*

[6] Khronos [n.d.], *OpenXR plugin*.
URL: *https://docs.unity3d.com/Packages/com.unity.xr.openxr@1.13/manual/index.html*

[7] Kossa, D. A. [2023], 'Végeselemes módszer alapjai'.

[8] M.A., E. [2022], 'Brieffiniteelement.net'.
URL: *https://github.com/BriefFiniteElementNet/BriefFiniteElement.Net*

[9] *Meta Quest 2 User Guide* [n.d.].
URL: *https://learn.gienc.org/course/meta-quest-2-user-guide/meta-quest-2-user-guide*

[10] Nolet, C. [2018], 'Quick outline'.
URL: *https://github.com/chrisnolet/QuickOutline*

[11] *Oculus Quest 2 tech specs* [n.d.].
URL: *https://www.meta.com/quest/products/quest-2/tech-specs/*

[12] Stringer, D. [2022].
URL: *https://www.youtube.com/watch?v=ijcn-mIJL5s*

[13] *Szilárdságtan Példatár* [2022].

[14] *Unity Documentation* [n.d.].
URL: *https://docs.unity3d.com/2022.2/Documentation/Manual/UnityManual.html*

A. Magyar leírás az alkalmazás használatához

A.1. A virtuális környezet felépítése

Az alkalmazás egyetlen virtuális térben fut, amely a szerkezet építésétől az eredmények kiértékeléséig minden tevékenységnek helyet ad. A környezetben található állandó elemek a következők:

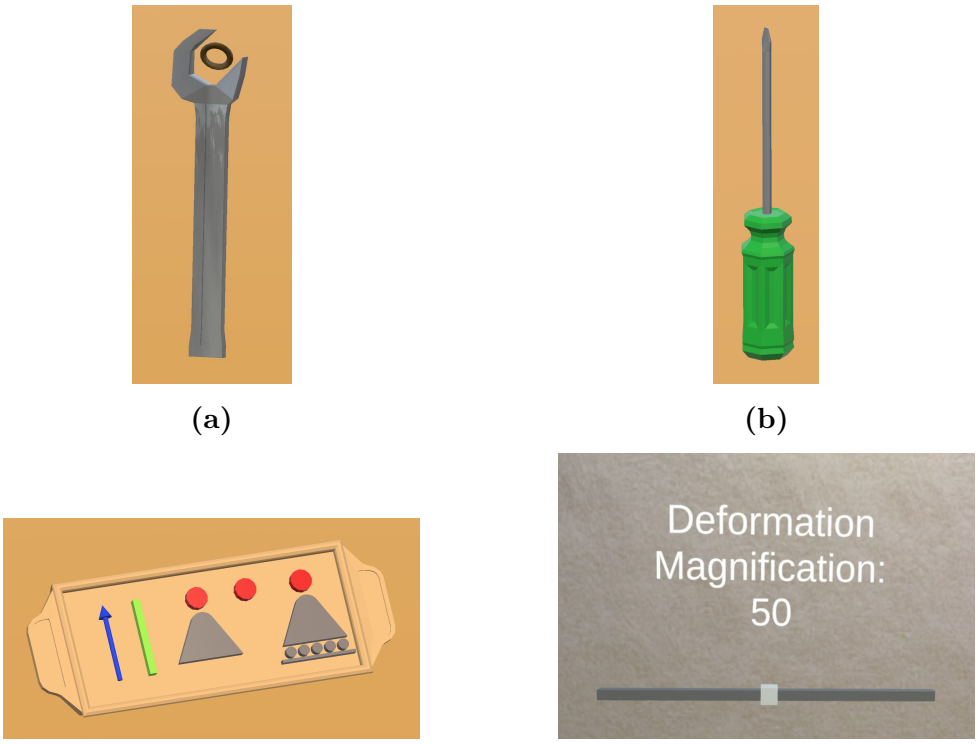
- Kijelző: Ezen jelennek meg az egyes szerkezeti elemek tulajdonságai és a számítási eredmények a kiértékelés során.
- Tábla: Az osztálytermi táblához hasonló felület, amely a szerkezet építésének alapját képezi. Minden rajzolási művelet ezen történik.
- Méterrúd: A szerkezet méretezését segítő eszköz. Pontosan 1 méter hosszúságú, és 10 centiméteres beosztásokkal van ellátva a pontos méretezés érdekében.

(a) **(b)** **(c)**

A.1. ábra. A virtuális tér alapvető elemei: Kijelző (a), tábla (b) és méterrúd (c).

A felhasználó által kezelhető eszközök a következők:

- Villáskulcs: A szerkezet építésére szolgáló alapvető eszköz, amellyel a felhasználó létrehozhatja és módosíthatja a szerkezeti elemeket.
- Csavarhúzó: A szerkezet mechanikai viselkedésének vizsgálatára szolgáló eszköz. Segítségével megjeleníthető a deformált állapot és a feszültségmező.
- Tálca: A táblára helyezhető összes elem mintapéldányát tartalmazza. Itt választhatók ki az aktív építőelemek.
- Nagyítási csúszka: Az elmozdulások láthatóságának javítására szolgál, a deformációk mértékének skálázását teszi lehetővé.



(a) **(b)**

(c) **(d)**

A.2. ábra. Az interaktív eszközök: Villáskulcs (a), csavarhúzó (b), tálca (c) és nagyítási csúszka (d).

A.2. Lehelyezhető tárgyak típusai

A táblára helyezhető objektumok két fő csoportba sorolhatók. Az első csoportba tartoznak azok az elemek, amelyek önállóan is elhelyezhetők a táblán, míg a második csoportba tartozó elemek csak az első csoport elemeire helyezhetők rá.

Az első csoport három különböző típusú csuklót tartalmaz, amelyek a A.3 ábrán láthatók:

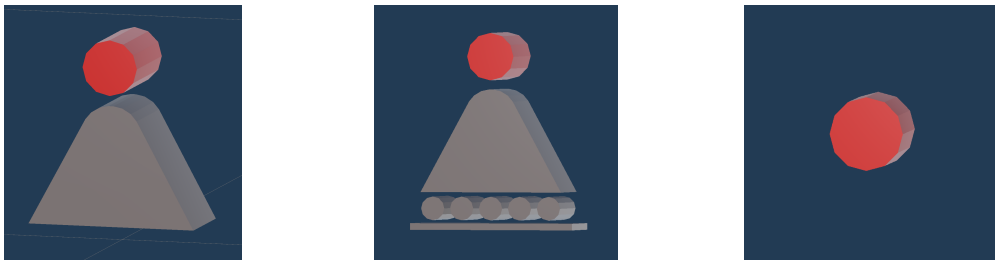
- Csuklós támasz: Ez a típus nem enged meg elmozdulást egyik irányban sem.
- Görgős támasz: Ennél a típusnál az elmozdulás csak egy meghatározott irányban lehetséges.
- Szabad csukló: Ez a típus minden irányban szabadon elmozdulhat.

A csuklók mindegyike rendelkezik egy piros hengerrel, amelynek forgástengelye egyben a csukló forgástengelye is. A csuklók kijelölése kizárólag ezen a piros hengeres részen lehetséges.

A második csoportba tartozó elemeket a A.4 ábra mutatja be:

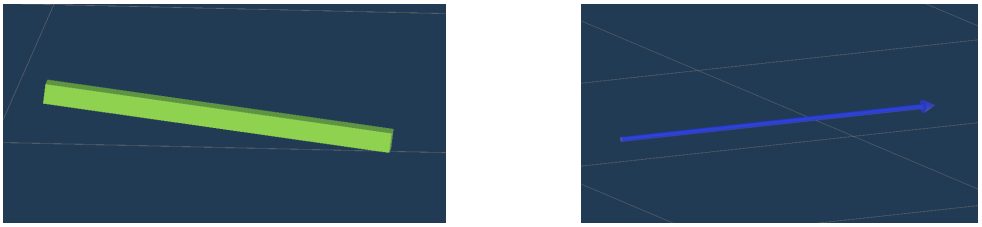
- Rúdelem (truss): Zöld színű, négyzet keresztmetszetű rúd, amely két csukló között hoz létre kapcsolatot. A rúd hosszának középső harmadánál lehet kijelölni.

- Erővektor: Kék színű nyíl, amelynek hossza az erő nagyságát reprezentálja. Az erővektorokat a nyíl hegyénél lehet megfogni és módosítani.



(a) **(b)** **(c)**

A.3. ábra. A csuklók fajtái. Csuklós támasz (a), görgős támasz (b), szabad csukló (c).



(a) **(b)**

A.4. ábra. A rúdelem (a) és az erő (b) modell.

A.3. A villáskulcs használata

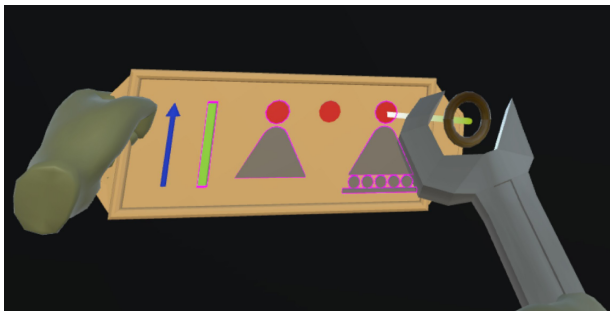
A villáskulcs felvételekor annak mutatója automatikusan aktiválódik. Ez a mutató szolgál minden művelet végrehajtására, amit ezzel a szerszámmal végezhetünk. A A.5 ábrán egy kézben tartott villáskulcs látható.

A.3.1. A tálca használata

A tálcán az összes lehelyezhető elem mintapéldánya megtalálható. A rendszer mindig két aktív elemet tart nyilván, egyet-egyet mindkét korábban említett csoportból. Az aktív elem megváltoztatásához a felhasználónak a kívánt elemre kell mutatnia, majd meg kell nyomnia a trigger gombot. A kiválasztás után az új aktív elem kijelölődik. A két csoportból az elemeket függetlenül lehet kiválasztani.

A tálca maga is megfogható és mozgatható. A szerkezet építése során praktikus az egyik kézben a tálcát tartani, míg a másik kézben lévő villáskulccsal történik az építés. A tálca használatát szemlélteti a A.6 ábra.

A.5. ábra. Kézben tartott villáskulcs.

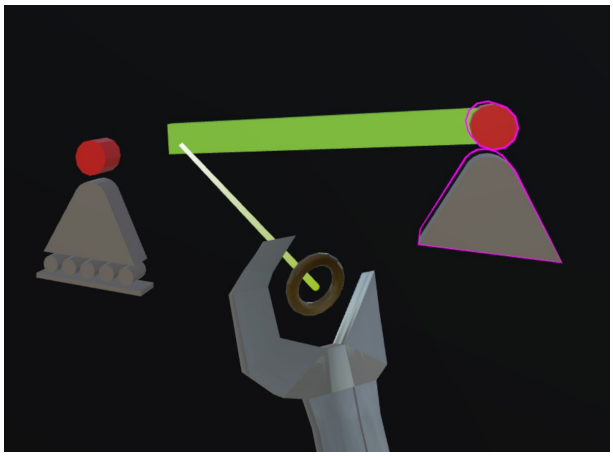


A.6. ábra. A tálca használat közben.

A.3.2. Elemek lehelyezése a táblára

A táblára mutató villáskulcs trigger gombjának megnyomásával a kiválasztott csuklótípus a mutatott helyre kerül. A rúdelem (truss) elhelyezése két lépésből áll: először egy meglévő csuklóra kell mutatni és a primary gombot lenyomva tartva a mutatót egy másik csuklóhoz kell vinni. Az építés folyamata során egy animáció azt a hatást kelti, mintha a rudat nyújtanánk a két csukló között. A gomb felengedésekor a rúd véglegesen a helyére kerül, feltéve hogy a mutató egy csuklón van. Ellenkező esetben a művelet sikertelen és a rúd nem épül meg.

Az erők elhelyezése hasonló módon történik, azzal a különbséggel, hogy az építésnek nem szükséges egy másik csuklón végződnie. A primary gomb minden esetben az aktuálisan kiválasztott objektum (rúd vagy erő) lehelyezését végzi.



A.7. ábra. Rúdelem építése.

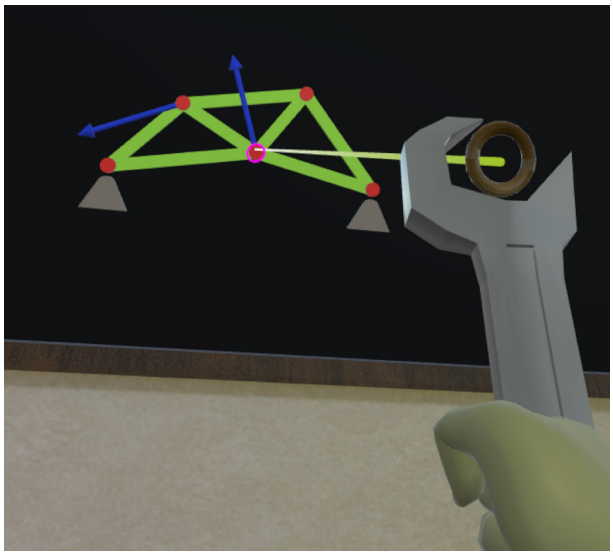
A.3.3. A már lehelyezett elemek módosítása

Ha a felhasználó a mutatóval egy már lehelyezett elemre mutat, az elem kijelölődik, jelezve, hogy készen áll a módosításra. A módosítás mindig a trigger gomb lenyomásával kezdődik, és amíg folyik a módosítás, addig lenyomva kell tartani. A különböző objektumok különböző módon módosíthatók:

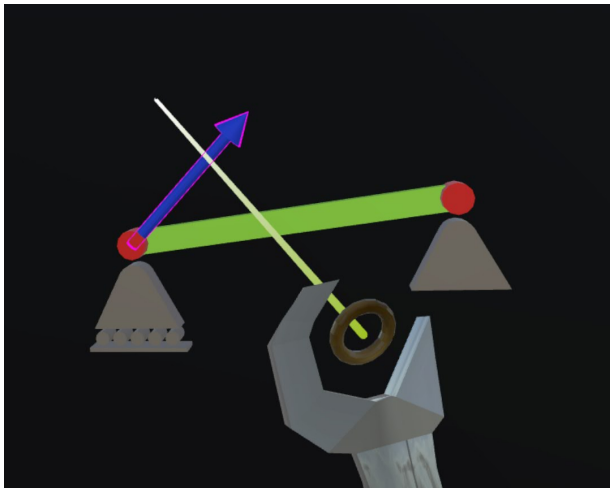
- Csomópont: A mutató mozgásával a csukló is mozog a táblán. Minden csatlakoztatott rúd és erő követi a csomópont mozgását. A joystick használatával további funkció érhető el: ha kellően eltoljuk, a program megkeresi a legközelebbi 90°-os irányt, és ebbe forgatja a csuklót. Ez a funkció teszi lehetővé a görgős támasz x és y irányú beállítását.
- Rúdelem: A mutató mozgása itt nem eredményez változást, csak a joystick használata vált ki hatást. A joystick által mutatott irány a legközelebbi 45°-os osztáshoz igazodik. A kiválasztott rúd ebbe az irányba fordul el a rögzítési pontja körül. A rúd másik végéhez kapcsolódó elemek automatikusan követik ezt a változást.
- Erővektor: Az erővektor vége szabadon követi a mutató mozgását, így az erő nagysága és iránya tetszőlegesen állítható. A joystick használata esetén az erő iránya a legközelebbi 45°-os osztáshoz igazodik, és csak a mutatónak a csomóponttól mért távolsága határozza meg a vektor hosszát. A szabad szerkesztési módba való visszatéréshez a felhasználónak el kell engednie a triggert, majd újra ki kell választania az erőt.

A.3.4. Elemek törlése

Az elemek törlése a menü gomb megnyomásával történik, miközben a mutató a törölni kívánt elemre mutat. A rúd és az erő esetében csak a kiválasztott elem törlődik, míg egy csukló törlésekor az összes hozzá kapcsolódó elem is automatikusan törlésre kerül.



A.8. ábra. Egy csomópont mozgatása.



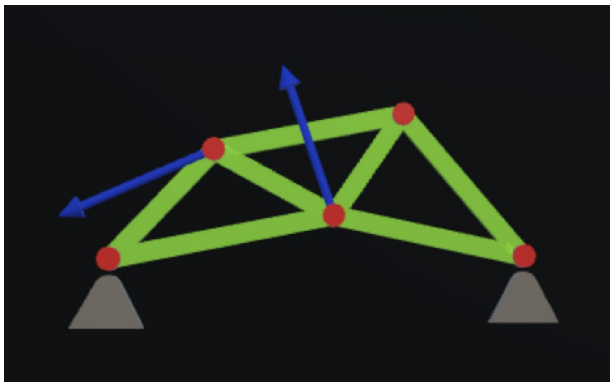
A.9. ábra. Erő illesztése egy irányhoz.

A.3.5. Példa szerkezet

Az A.10 ábrán egy példa rácsos szerkezet látható. Ennél a pontnál következhet a szerkezet kiértékelése.

A.4. A csavarhúzó használata

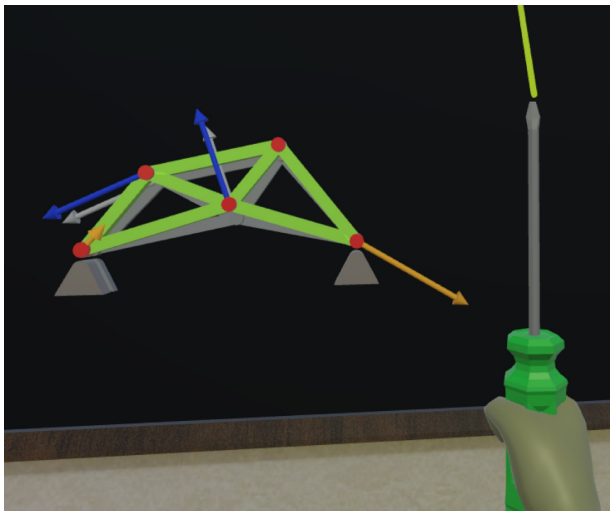
Ez az eszköz a szerkezet kiértékelésére szolgál. Amikor a felhasználó felveszi a csavarhúzót, a számítás automatikusan elindul, és hasonlóan a villáskulcshoz, a mutató aktiválódik. A kijelző kiírja, hogy "Ready", amikor a program befejezte a számításokat. Fontos megjegyezni, hogy bár a program bármilyen szerkezeten futtatható, az eredmények csak akkor lesznek helyesek, ha a szerkezet statikailag határozott vagy túlhatározott. A többi esetben, csak a program stabilitása van biztosítva.



A.10. ábra. Példa rácsos szerkezet.

A.4.1. Az eredmények megjelenítése

A számítás befejezése után az eredeti szerkezet szürkévé válik, és előtte megjelenik a deformált szerkezet. A szerkezetre új, narancssárga árnyalatú erők kerülnek, amelyek a reakcióerőket reprezentálják. Az A.11 és A.12 ábrák szemléltetik a deformált szerkezetet.

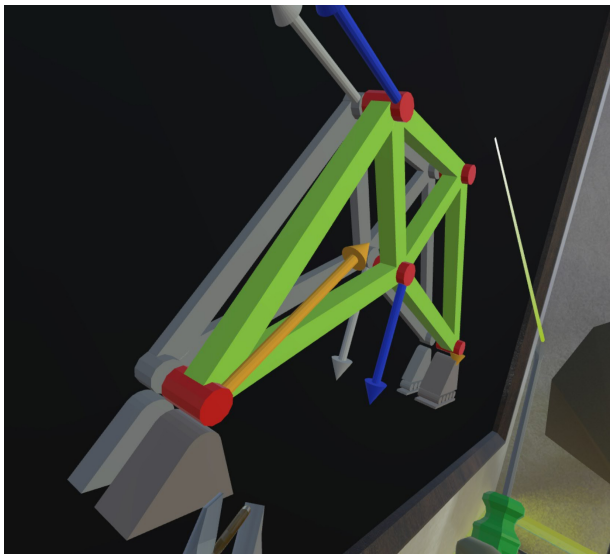


A.11. ábra. A szerkezet a csavarhúzó felvétele után.

A.4.2. A numerikus eredmények kiolvasása

A deformált szerkezet egyes elemeinek részletes adatai a következőképpen tekinthetők meg:

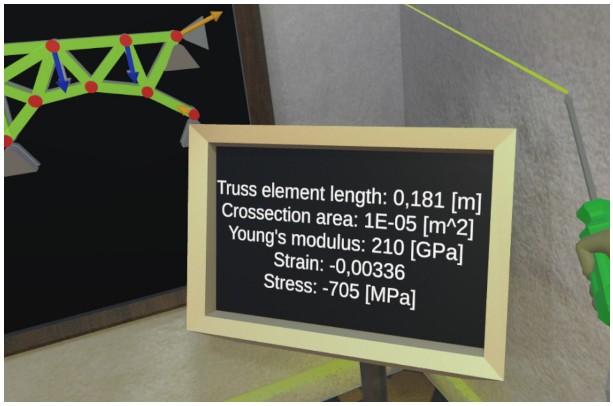
- Csomópontok esetén: A rendszer megjeleníti a csomópont koordinátáit, a rögzített szabadságfokokat, valamint a szabad irányokban történt elmozdulásokat.
- Rúdelemeknél: A kijelzőn láthatóvá válnak a rúd geometriai és anyagi jellemzői (hossz, keresztmetszet, rugalmassági modulus), valamint a számított alakváltozás



A.12. ábra. A deformált szerkezet az eredeti előtt.

és feszültség értékek.

- Erőknél: Megjelenítésre kerül az erő teljes nagysága és komponensekre bontott értéke.



A.13. ábra. A kijelző egy rúdelem tulajdonságait mutatja.

A.4.3. A feszültségmező megtekintése

A primary button lenyomva tartásával az összes rúdelem színe megváltozik a feszültségszintjüknek megfelelően. A falon megjelenik egy színskála, amelynek két végén a minimális és maximális feszültség látható, és egy másik címke jelzi a nulla feszültségszinthez tartozó színt. Ebben a módban a tényleges feszültségérték számít, a piros szín a nyomást, a kék szín a húzást jelzi (a színátmenet folyamatos a kéktől a pirosig). Ennek a funkciónak a példája látható az A.14 ábrán.

Ha a secondary button lenyomva van, akkor a rúdelemekben lévő feszültség abszolút értéke jelenik meg. Most a színátmenet zöldtől (ami a nulla feszültséget jelzi) pirosig

A.14. ábra. A szerkezet a feszültséget mutatja.

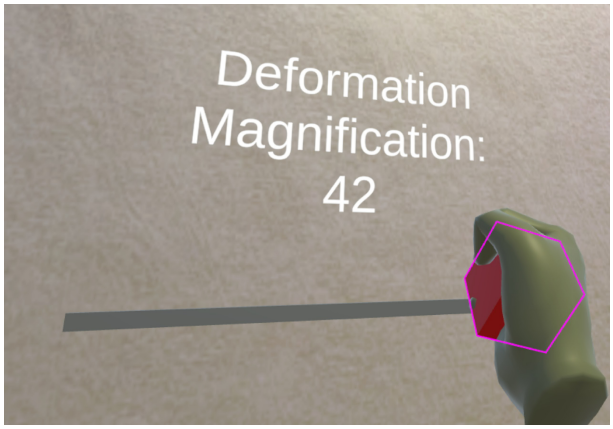
(ami a maximális feszültséget jelzi) terjed. Az abszolút feszültség megjelenítésének példája látható az A.15 ábrán. Miután a felhasználó megnézte az eredményeket, újra kézbe veheti a villáskulcsot, módosíthatja a szerkezetet, és a csavarhúzóval megnézheti, hogyan befolyásolják a módosítások az eredményeket.

A.15. ábra. A szerkezet az abszolút feszültséget mutatja.

A.5. A csúszka

A csúszka szövege az elmozdulások nagyítását jelzi, miközben a felhasználó a csavarhúzót tartja. A nagyítás szükséges, mert az elmozdulások olyan kicsik, hogy nélküle alig lenné-

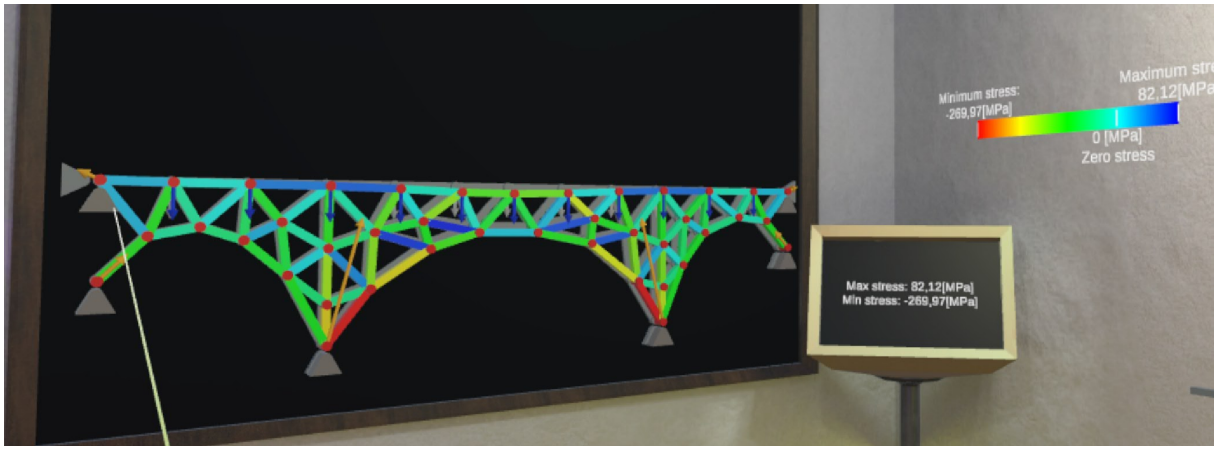
nek láthatóak. A csúszka értéke 0 és 100 között állítható. A felhasználónak nincs szüksége semmilyen eszközre az érték beállításához, egyszerűen megfoghatja és mozgathatja a kis kockát, amíg a szöveg a kívánt értéket mutatja, majd elengedheti a kockát. A csúszka beállításának példája látható az A.16 ábrán.



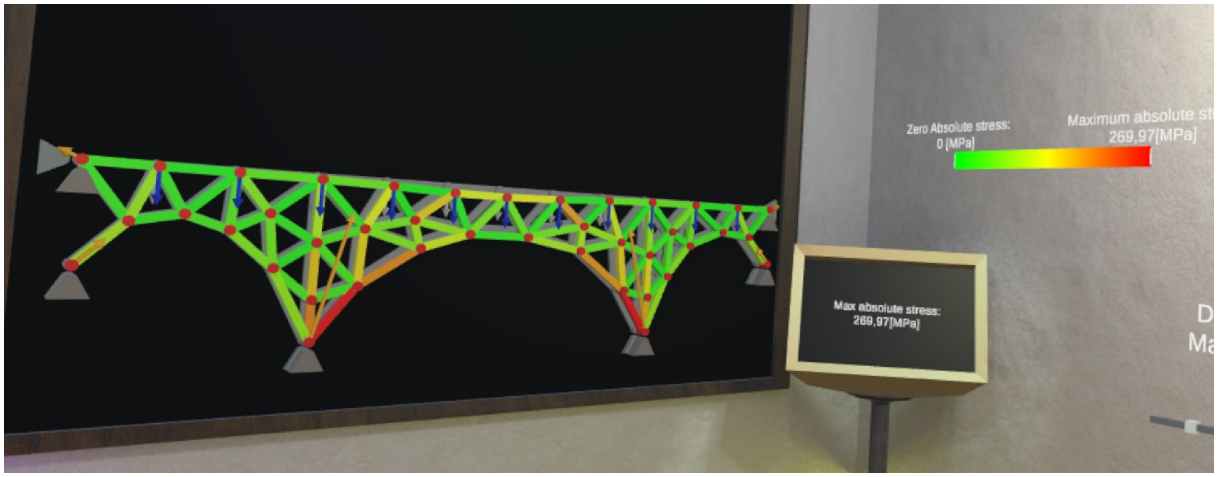
A.16. ábra. A csúszka beállítása.

A.6. Egy nagyobb modell

Egy nagyobb modell eredményei láthatók az A.17 és A.18 ábrákon.



A.17. ábra. Feszültségek egy híd modelljében.



A.18. ábra. Abszolút feszültségek egy híd modelljében.