Oliver Reichmann

Using Augmented Reality to give Instructions for Building LEGO Models

Bachelor Thesis

to achieve the university degree of
Bachelor of Science

submitted to
Graz University of Technology

Supervisor
Assoc.Prof. M.Sc. PhD Ursula Augsdörfer

Co-Supervisor
Dipl.-Ing. Dr.techn. Andreas Riffnaller-Schiefer

Institute of Computer Graphics and Knowledge Visualisation

Faculty of Computer Science and Biomedical Engineering

Graz, July 2020

This document is set in Palatino, compiled with pdfLaTeX2e and Biber.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

Abstract

In the last couple of years, Augmented Reality (AR) got a lot of attention, thanks to the development of even smaller but also more powerful computers. They now easily fit into smartphones and head-mounted devices. Because of that and also because of AR games like Pokemon Go, the growth of this technology is greater than ever. In this thesis I describe how I build an application that runs on a head-mounted device, the Microsoft HoloLens. The device gives users instructions on how to build a LEGO model, without the need for using their hands to look it up in a physical booklet. To achieve that, we used a combination of Python to preprocess our input files, and Unity for building our HoloLens application. The final result is a working prototype that shows AR building instructions and is controlled via voice commands.

Contents

List of Figures

1 Introduction

AR is a powerful technology with many practical use cases, be it for teaching, gaming, giving directions or anything else. What we want to do is to use it for giving users instructions on how to build a LEGO model with the help of a head-mounted device.
The goal of our application is to give the user step by step instructions for various LEGO models, shown as realistic as possible. The manual should always be available in the users field of view, while users should be able to inspect the model from all sides and also rescale it. Additionally, the new bricks in the current step should be somehow highlighted for better understanding. We defined a few requirements that our app should meet:

Platform Picking the right platform is essential. For us, the decision was between using an Android phone and taking advantage of the camera to create the illusion of a see-trough device with additional information overlay, or using the Microsoft HoloLens, specifically designed for Augmented and Mixed Reality. The fact that an application on a head-mounted device could be controlled hands-free, made it easy for us to decide to develop our app to run on the HoloLens.

Model picker In the start menu we want an input field where we can choose a model that we want instructions for. The application should scan a directory and list all models with valid instructions and let the user pick one. Optionally it could also show a model preview and give extra information, like how many steps have to be carried out or which and how many (different) bricks are used.

Dynamically Loading No model should be loaded upfront, so that we only have to keep the steps for the current instruction in memory. Just after the model is picked, the brick meshes for just that model get loaded, ideally even just for the current step.

Step by Step Instruction For every model we want to show every step separately, highlighted for the best understanding, with bricks appearing where they should be. If there are no steps given in the input LDraw [14] file, the model should be shown brick after brick in the order they are described in the file.

Voice Input Since we want to build something while we are getting the instructions from the AR glasses, the best and easiest input we can have to go from step to step is to use voice input. With this method, the user has his hands free for the construction. We want commands to pick the model in the beginning, as well as to go to the next and previous step. In addition commands for starting and for going back to the model picker would be nice to have.

Small model To better understand what the users are building, we want to show them a small version of the model they are currently working on, that is always in their field of view. This should give an overview of how it should look in the end.

Movable/Scalable model To better see and interpret the instructions, the model geometry should be movable to place it anywhere in the room and scalable to change the size.

Step Highlighting To see the bricks used for the current step, we want to highlight them so users clearly see which ones to add where.

In the following we will first talk about what Augmented Reality is and what it can be used for. Furthermore, we will show how and with what means we implemented the above proposed application. Subsequently we evaluate it and speak about what went wrong on the way and how we overcame these problems. In the end we will come up with some things that we can change or add to further improve the app in the future.

2 Augmented Reality (AR)

When we speak about AR, we mostly think about overlaying or adding information to the real world. In most cases this happens with the help of a head-mounted device that contains a see through display. If the display is completely opaque and shields the users view from the reality, we speak about Virtual Reality (VR).
Even though it may seem like AR is a relatively new technology, the idea of it can be traced back to more than 100 years ago, way before the first computer. Back then, the american author L. Frank Baum, famous for his book "The Wonderful Wizard of Oz", described a device that can be compared to modern AR glasses. In his short story "The Master Key" from 1901, he writes about a pair of electrical glasses that show the wearer the character of the person in front of him in form of a letter over their head [2]. It took almost another 70 years until Ivan E. Sutherland proposed his idea of a head-mounted display in his paper from 1968 [24]. And even though many people think about head-mounted displays [20] when they hear about AR, it is not limited to glass-like devices. As long as some computer generated graphics are merged with the real world and are shown to the user, we can speak about Augmented or even Mixed Reality (MR).

2.1 Use Cases

AR can be used in many different situations, however, not all of them are suitable for everyday use. That is also because the hardware for more advanced displaying has to be good and is therefore still in the upper price range and hardly affordable for normal users. Furthermore, many devices are still quite big or have a limited battery capacity. Here are some usages:

Information overlay Information overlay is one of the easiest use cases, however it is still a very useful technique. The idea is that users, wearing the AR glasses or holding their smartphone, are getting information about their surroundings. A good example for that is Google Lens, an application that scans and interprets whatever is filmed with the phone's camera. The software can scan things or text, to look it up online [21], give style ideas [23] or be a tour guide in an unfamiliar city [16]. Apps that run on head-mounted devices can be advantageous, for example when driving a motorcycle or bike to give information about speed, directions, or even have a 360 degree field of view with the help of additional cameras.

The overlay is mostly non-immersive, meaning that the users do not feel like they are in a virtual world. The artifacts simply get displayed in front of the eyes or on any other display, preferably not obstructing anything else. Even in the car, a projection of information like road conditions, speed, directions or other useful information can be an area of application [26].

Translation A case where it is useful for the generated graphics to overlay things in the real world, is when AR is used for translation, like provided by Google Translate [10]. The application uses text recognition to detect writing in a picture or video stream coming from a camera and translates it. The translation is then displayed directly over the real world text, so that the users just see text that they can understand. That can be useful while driving in a foreign country to quickly and directly understand street signs. A slightly modified approach is to use a microphone instead of a camera and use speech recognition to hear and translate the words to show the user.

Instructions What we want to do and what many AR glasses are used for, is giving instructions and showing the users what they have to do. A popular use case is giving the users directions on how to assemble a machine that they are not familiar with or, like Antifakos et al. [1] did, give instructions on how to build furniture. In that way users can have clearly shown instructions that are integrated with the real world. Ideally they are shown on a head-up display (HUD) so that the user does not have to look away from his work and also has his or her hands free for work.

Computer Games Gaming is one of the smaller fields, since here the usage of VR seems to be preferred by bigger game studios. AR is mostly used on phones with the help of the camera to simulate transparency and interacting with the real life. The advantage of games on smartphones is that nowadays the computing power often is better than on glasses. A famous one is Pokemon Go [3] from 2014, played on the smartphone. Here the phone camera and screen is used to show the real life, overlayed with graphics from the game.

Room planning/designing There are applications, among others from IKEA [12], that let you design your interior with the help of AR. They simply work on your smartphone and let you insert models of their furniture into your living room, with the help of the camera. Everything works in real time with nothing more than a smartphone.

Architecture/Construction AR can be used in many ways for showing users buildings and constructions that do not exist yet. It can help to give a better understanding on how something like a house should or will be build and can even give workers a plan on what the architect has envisioned. Delgado et al. [5] researched in their article, how augmented reality can be used in architecture, engineering and construction and found various ways on how it can be used in the process of building, for stakeholder engagement, design support, construction support or even training. One possible way of using AR in the field of archaeology is to use it for visualization of reconstructed sights. Dähne and Karigiannis [4] proposed a system to virtually rebuild ruins at historical sites and give the user a tour. AR provides visual input on how the buildings could have looked like.

TV One of the most consumed form of AR is utilized in television every day. Many TV stations want to spice up their weather forecast and show the viewer a moving and changing map with the weather showing and the presenter interacting with it. That can go from simple symbols up to fully displayed tornadoes or floods in the studio like done by IBM Max Reality [11], seen on Figure 2.1, all just to entertain the viewers. Similar methods are used when showing statistical data, for example the results of votes in different parts of the country, like ORF, national television in Austria, did it at the 2019 Austrian legislative election, seen in Figure 2.2. Pejman et al. [22] even tried to add AR artifacts to the existing TV program to give viewers additional content and tested it on a documentary.

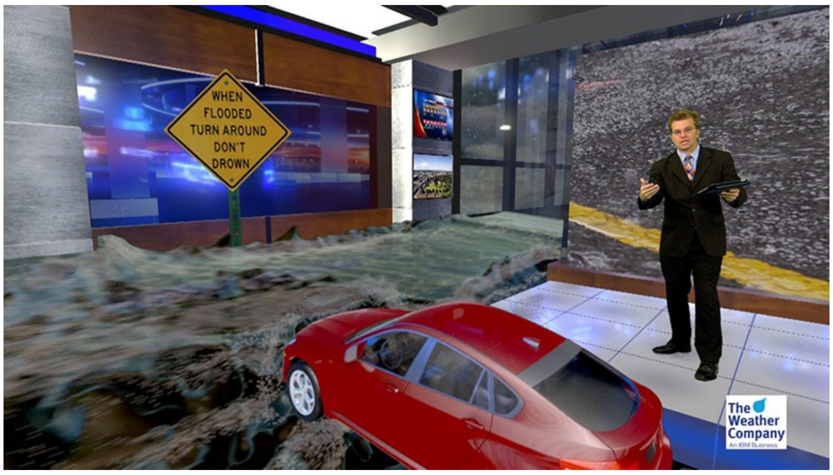


Figure 2.1: Flooding in the studio, made possible by IBM Max Reality, taken from [11].

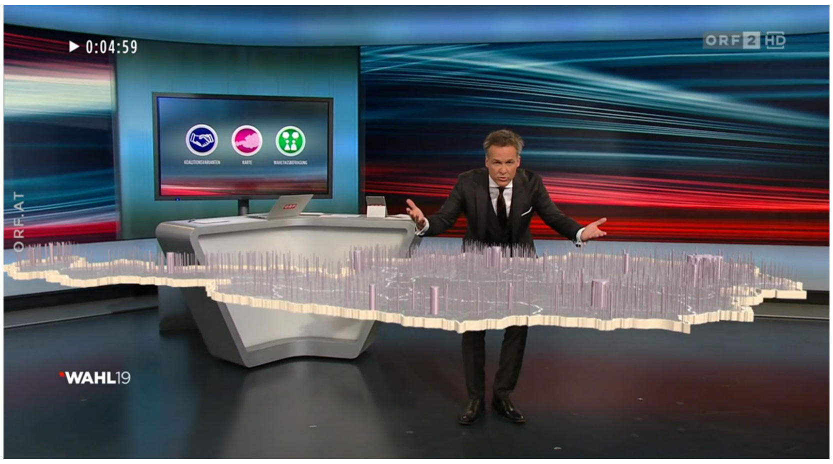


Figure 2.2: Augmented reality showing election results on TV, taken from [6].

3 Implementation

Of course there are multiple ways on how to implement our desired application, so we first take a look on what kind of technology is even available.

3.1 Available Hardware

There are currently multiple AR devices on the market, with many of them being some sort of glasses but with the increasing power of smartphones, AR applications also became available for them. That may be the easiest and widely available form. There are even concepts for motorcycle helmets with integrated displays to give the driver real time information while on the road [13].
One of the better known head-mounted device is the Microsoft HoloLens [18] which runs a Windows operating system (OS) and is therefore relatively easy to use. That is also the reason why we used it in our solution. Another device that attracted some attention a few years ago is the Google Glass [8] which is, compared to the HoloLens, much lighter and smaller. But also AR on smartphones is, thanks to ever rising performance, wide spread. One powerful application for it is e.g. Google Lens [9].

3.2 Available Software

Every device has, depending on the vendor, a different OS and therefore also different applications that can be used. For example, the Google Glass uses Glass OS which is based on Android and also has an API for developers to make their own apps. The HoloLens on the other hand, uses the Windows

Mixed Reality platform which is part of Windows 10. That means it can run Universal Windows Platform apps that can be loaded directly from the Windows store and has a Holographic API for 3D applications.

3.3 Features

Most of the available AR glasses have some integrated features that enhance the usability of the device. How good they work of course depends on the vendor. Many have an integrated camera, either to just take photos/videos or, like it is the case with the HoloLens, to track the room and hand movements for controlling. Another feature that can be used for that is eye tracking. For that the device needs a camera on the inside, which unfortunately makes it even bigger. Gesture input is another common technique to interact with the device.

3.4 Used Technologies

For realizing our project, we picked a wide range of both soft- and hardware.

3.4.1 Software

LDraw LDraw [14] is a collection of software tools to model LEGO designs, created by James Jessiman. Today it is maintained and further developed by the community. From this tool collection we just use the file formats .ldr and .mpd (Multi-Part Document) and the large library of parts for the models. The .ldr file tells us all the steps and which bricks are used in this phase. Additionally to the file name of the new brick, it also tells us the color and orientation in space. In the brick files we have information about lines, triangles, quads, and in most of them also references to other files, so that common structures do not have to be defined in every file but can be shared. For parsing that means we can work with recursion. There is also much more additional information in these files, which we mostly ignore since it is irrelevant to us.

Python For the preprocessing of the model step files we used Python 3.8.1 [7], an easy and powerful scripting language. We use it to read the .ldr or .mpd model files and translate them to meshes that we save in .obj files to later load the model in Unity.

Wavefront OBJ/MTL This file format is used for the step models, since Unity can easily read and understand it without the need for us to write our own parser. It is one way to describe 3D meshes. The files are saved in plain text so humans can read it. To describe a model, the file first specifies vertices and then faces that reference them. Additionally, it is also possible to define vertex normals and texture coordinates. Since the LDraw format does not provide textures but only the colors of the bricks, we are also using the material template library (.mtl) to specify the colors/materials of the objects.

Unity/C# Unity [25] is an application for creating games and applications. We use it to build our program for the HoloLens, where we load our generated .obj files for each step and show them to the user. C# is the language used in Unity to write additional scripts.

Mixed Reality Toolkit for Unity (MRTK) This toolkit [19] is a Unity framework, developed and distributed by Microsoft, and is especially designed for use with the HoloLens, also distributed by the same company. It provides many useful functions and scripts for building and interacting with apps in AR.

Windows Since we are using the HoloLens, which runs a Windows OS, we decided it would also be best to use a Windows PC to code and build our Unity application. Also the preprocessing which is done with Python was coded on Windows even though it also runs on Linux.

3.4.2 Hardware

HoloLens 1 The HoloLens is one of the available 3D AR glasses and is developed by Microsoft. Currently the 2nd version is available but for our project we used the 1st generation. It uses a common, popular OS, Windows, which has many good maintained tools to develop applications for it.

3.5 Application

The goal of the project is to create an application for the HoloLens that gives the user instructions for building a LEGO model step by step. The app should let the user select a model and display the bricks for each step in Augmented Reality on the glasses. For that we use a two step approach. First we preprocess the input files and then build and compile our Unity app that uses the created model files. The plan is to use the instructions from the input file, that we got from the LDraw Model Repository [15]. Most of the models from there have information about the bricks that we need per step. As a backup, in case there are no instructions, we just create a step for every brick.

3.5.1 Step 1: Preparing the Model

To show the model step after step, we first have to parse the given model file. We are working with the open source model description file format LDraw, which provides us with .ldr files. These describe lines, triangles and quads. The problem is that Unity, the application we will use to display the models, does not understand this file format. Therefore we are using the

Python scripting language to read the .ldr files and write out a model in the Wavefront OBJ file format for each step. Additionally we are writing a file for the whole model. To represent the color of the models, the Material Template Library format (.mtl), a companion file format to .obj, is used.

Input

We have two options on how to describe our input. The Multi-Part Document (.mpd) and the LDraw (.ldr) file format. Our parser works with both of them and can create a valid set of input files for the second step.

MPD The Multi-Part Document is a file that comes from the Official Model Repository (OMR) [15] and describes a brick set, released by LEGO. It has a standardized file name with information about the set name and number and in principle just contains one or multiple models in the .ldr format. They in return refer to multiple .dat files that describe the bricks.

LDR The LDraw file describes the orientation and coordinates of multiple bricks of a model in 3D space. It consists of multiple lines, each one starting with an integer that describes the kind of line. There are 6 different line types:

0 Commands or META command like information about new steps.
1 Sub-file reference and their orientation and coordinates.
2 Line between two points, typically edges of bricks.
3 Filled triangle between three points.
4 Filled quadrilateral between four points, defined in either clockwise or counter-clockwise winding order.
5 Optional line between two points, that is only shown from an certain perspective.

All types, apart from 0, contain at least the 3D coordinates (X,Y,Z) of one point, all of them also contain an integer that describes the color of the item. This color has to be looked up in a dictionary because the number says nothing about the red-green-blue (RGB) values. Line type 1 defines a homogeneous transformation matrix, with which we have to multiply all points in the referenced file.

We do not parse line type 5, since it only represents optional lines that are shown depending on the perspective. We do not care about them, we only want the minimum so we can still understand the model but do not have any unnecessary overhead.

Output

As output we get multiple files that we will need in the second step. Most of them are Wavefront Object files (.obj file ending), additionally we also create one normal text file (.txt) that contains the name of all available models and a file with the colors. We get the following 3 files, some of them can also occur more often, depending on how many steps or bricks we have.

Steps The most important thing we want to extract from our .ldr file is a model of every step. For that we used the .obj file format which contains our vertices and surfaces between them. The more steps we have, the more of these files we get.

Colors To give the models the same color as in the .ldr files, we have to use the .mtl file format, so for every model we parse, we create one color.mtl file. Since it contains a dictionary of the LDraw colors (every color number is represented as a material), we just need to write one for all step models.

Small models For showing the small model at the side, we write an extra .obj, which contains all the step files copied together, so we get the final model mesh. That is not the optimal solution, since we also create points and triangles, that are not seen inside of the finished model but it is simple and fast in creation. Also many points get defined multiple times, even if they are used from various items.

Preprocessing

As mentioned earlier, we decided to use Python for preparing the model. It is very simple to read and write files with it and is also object-oriented, a feature we can use to our advantage, after all we are dealing with multiple bricks. We create our own brick class that has all features that we get from our input: color, position, orientation and file name. Additionally, every brick object also has lists of all its lines, triangles, quads and eventually sub-bricks. These things are again represented with their own classes.

Build Assets

For the second step, we have to prepare our models and build asset bundles. To do that, we have to write a build script that can later be started in the Unity editor. Unfortunately we can not build them directly without opening the editor since we have to manually add our meshes to the respective packages. In our script, that has to be in the folder `Assets/Editor/` of our Unity project, we define the output path and with `BuildAssetBundles(string outputPath, BuildAssetBundleOptions assetBundleOptions, BuildTarget targetPlatform)` we build our asset bundles for the `targetPlatform`, in our case `BuildTarget.WSAPlayer` since applications on the HoloLens are Windows Store Apps.

3.5.2 Step 2: Unity App

In the second step, we want to build an application for the HoloLens, where we can pick a model to be shown step by step. To do so, we used Unity to make and build the app and Visual Studio to debug it while running on both the HoloLens emulator and the actual AR device.
The app structure is simple, it has just one scene with an interface for picking the model. For that there is a drop down menu that gets filled at run-time with all models we have. This information is in the file `models.txt` which gets updated every time we prepare a new model with our Python script. Additionally, there is some text for the user, as we can see in Figure 3.1.
After picking the model, this interface disappears and the first step of the picked instruction gets shown. Furthermore, a small preview of the finished model appears in the upper right corner. Both these meshes get loaded at run-time, depending on which instruction the user picks. In the background, the app loads all of the steps in advance and sets all step meshes up to the current visible and all others invisible.

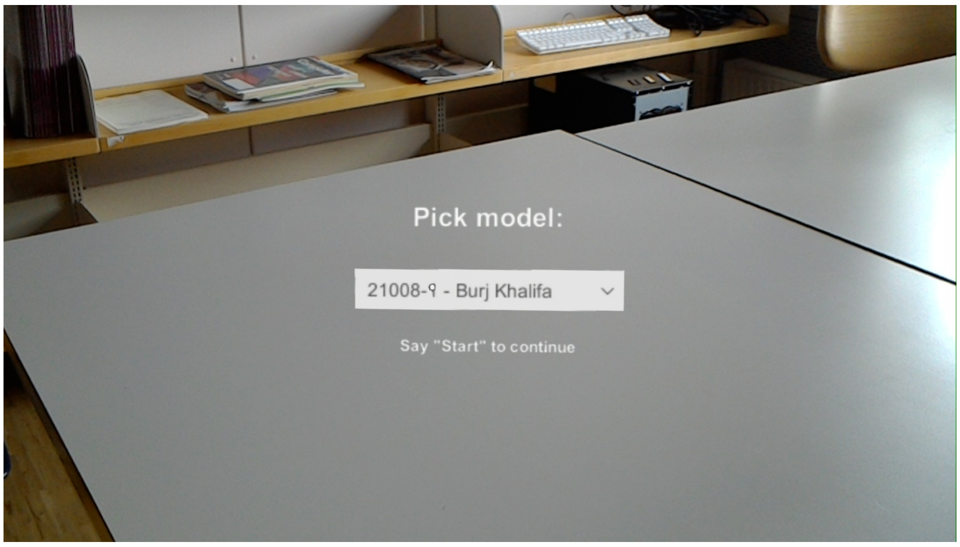


Figure 3.1: Start screen.

Voice Commands

For controlling our instructions, we wanted to use voice commands. For this, we made use of the MRTK Voice Input. We first had to define our keywords and then add a response to these words later on. Since we load our model at runtime, we also want to add the responses later on in our C# script. Only two of the keywords could be added in advance, the one to start our picked model in the beginning and the one for going back to the model selector. The “Start” keyword is the only one we need at the menu screen and it gets disabled once the model is picked. In Table 3.1 we can see our available words and the information what they trigger and when they are added. We decided to add even more keywords than what was planned in the beginning to make the model handling easier.

Preview

On the top right corner we added a small replica of the finished model. To prevent the user to loose this preview, it is fixed to the camera object. That way, it will always stay in the same place. Unfortunately, that makes it

Keyword	Response	Added at runtime
Start	Starts the instructions for selected model	
Back	Ends the instructions and goes back to the model picker	
Next	Shows the next step	X
Last	Shows the last step	X
Drop	Enables Gravity so that the model can be placed	X
Come Back	Brings the model back in the users view and disables gravity	X

Table 3.1: Usable keywords to control the application

impossible to focus and therefore grab or modify it to have a better look at it from all angles. To solve this, we wrote and added a small C# script to make it rotate. This is the very simple code that we used:

```
using UnityEngine;
public class RotatingObject : MonoBehaviour
{
  public float xAngle, yAngle, zAngle, speed;

  void Update()
  {
      transform.Rotate(xAngle*speed, yAngle*speed, zAngle*speed);
  }
}
```

We see that we have 4 public variables, x-,y- and z-angle and the speed. The angles can be set to by how many degrees times speed the model should be rotated. They can be changed at runtime from any other script.

Highlighting

Highlighting is an essential point in our application for better recognition of the bricks of the current step. The first approach was to let the bricks of the current step emit light, which turned out to be not that bright. To solve that, we changed it up a little bit and simply change the color of the material. We let it pulse between the original color and very light gray. That helps the user to clearly see which parts are new, while also seeing the original indented color. To every step model we added a script in which we where saving the original colors before we started our application. We have a public boolean that gets set to true if the step is the current one and in the Update() function we check for that variable:

```
void Update()
{
  if (activeStep)
  {
    int i = 0;
    foreach (Material mat in materials)
    {
      mat.color = Color.Lerp(startColors[i],
                             Color.white * 0.8f,
                             Mathf.PingPong(Time.time * 0.2f, 1));
      i++;
    }
  }
  else
  {
    int i = 0;
    foreach (Material mat in materials)
    {
      mat.color = startColors[i];
      i++;
    }
  }
}
```

`Color.Lerp((Color a, Color b, float t)`[1] linearly interpolates between the first and second parameter (original color and very light gray) by t, `Mathf.PingPong(float t, float length)`[2] returns a value that increments and decrements between 0 and length. The first parameter t has to be a self-incrementing one, like the time.
Another possible way would be to change the shader for the current step to one that highlights the edges or changes the color in another way.

3.6 Workflow

Currently, to get an instruction of a model of our choice we have to pursue the following steps:

1. Start the Python script with following command:
   `python main.py path/to/model/file`.
2. Open the project in Unity.
3. Navigate to the folder `Assets/Models/name_of_model` and add all the models in that folder to a new asset package with the name of the model at the bottom of the inspector pane.
4. Build the app in Unity.
5. Open the build project in Visual Studio and deploy it to the HoloLens.
6. Start the application on the HoloLens.

[1] https://docs.unity3d.com/ScriptReference/Color.Lerp.html
[2] https://docs.unity3d.com/ScriptReference/Mathf.PingPong.html

4 Evaluation

We found that it is possible to build our application for the HoloLens that works as intended but we still had to overcome a few problems and faced some challenges. These are the difficulties we had to deal with:

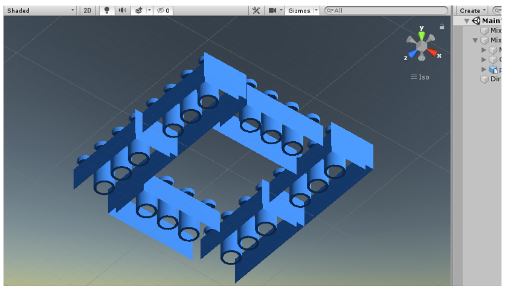
Wrong orientation matrix This problem affected us right in the beginning. For getting all our bricks at the right place, we have to transform all the points that we read from the input. For that, every brick file that gets added per step has its own transformation matrix defined. We now have to transform all the points inside of the new file to have the bricks where the file wants them to be. The error we made was that we also have to transform the new transformation matrices inside of the file and not just the points. The result was that the points and triangles in the first layer of files would be in the right position but any others would be wrong.

Color To get the color of the models right was difficult. The .ldr file defines the color at the beginning of every file, line, triangle or quad with help of a single integer. This number then has to be looked up to find the actual color. Some of the numbers also define other behaviors, like to take the same color as the brick one layer above, or the complementary color.

The fist approach was to define vertex colors, to give every point in the .obj file a RGB value. That works great if we work with programs that can interpret and parse the vertex color like MeshLab [17], which we used to test the output of the Python script. The problem emerged when we started with the second step and tried to load our objects in Unity to build our application. The Unity .obj parser does not understand vertex colors, so to interpret them we would have to use a custom shader.

The other option we have is to use a material library file (.mtl) and write all the colors/materials in there, instead of defining them per vertex. Now we just have to write a separate file with all the colors and define which ones we are using in front of the triangle lines in the .obj file. With that, Unity can create the materials itself to add them to the models.

Shading The way the shader, that is added to a loaded .obj mesh in Unity, defines the outside of a face, is to take the direction of the normal of it and then just colors this side. The other side of it is see-trough. The problem we have is that in the preprocessing script we have no normals and therefore also do not know which side is outside or inside. That means we can not write them in the .obj files.
Unity now takes the points in the order they are given in the file to calculate the normal of each face. That does not always work as we can see in Figure 4.1a. To solve that we can either write our own shader that colors both sides of the faces, or we take the easier way and define every face twice. We switch the order of the points that they are defined by and so effectively switch the normal. The outcome can be seen in Figure 4.1b. Even though we have a little overhead now (every polygon is defined twice), it saved us time with writing a shader and from possible problems that we could have gotten with shading both sides of the faces.


(a) Shading of front side of polygones.

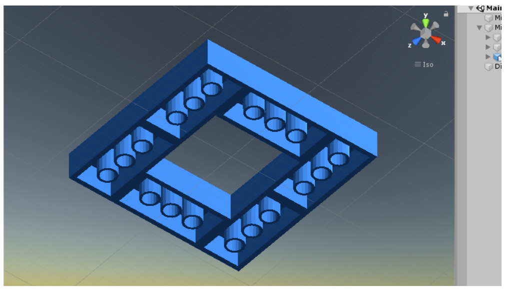
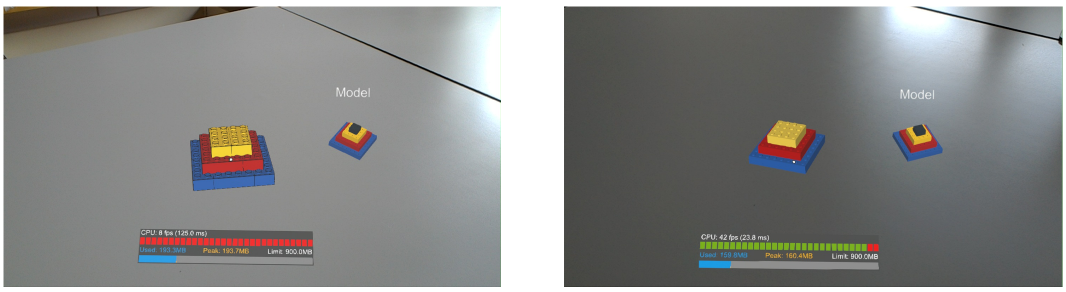

(b) Shading of both polygon sides.

Figure 4.1: Shading with one sided vs. double sided polygons.

Lighting To make the the model clearly visible we decided to add an extra directional light to the scene. We could also just use ambient lighting but this was just not bright enough. We also want to take advantage of shadows to bring out the features of the models.
To always have the light from the front and do not have a darker side, the added light is attached to the camera. In this way the light always comes from the users head.

Lines Another big problem we have with the .obj format is that the Unity parser does not know what to do with the lines defined between two vertices. All the models just have an uniform color and we can not distinguish between different bricks.
One fix for that would be using texture mapping but since we do not have texture pictures and do not want to create them for every single brick we are using, this is not optimal. A possible solution that we were trying, is to write all our lines to a separate file that we were parsing manually. In our C# script that we are using to load the current model, we are reading the lines file, a simple plain text file, and create and add all the lines. Unfortunately, that is a large overhead, even with a simple model we do not get more than 10 frames per second like we can see in Figure 4.2a. Since there is not really any other good way, the best is to not add lines. With shadows turned on we can still recognize most of the parts and we even get decent performance as seen in Figure 4.2b.

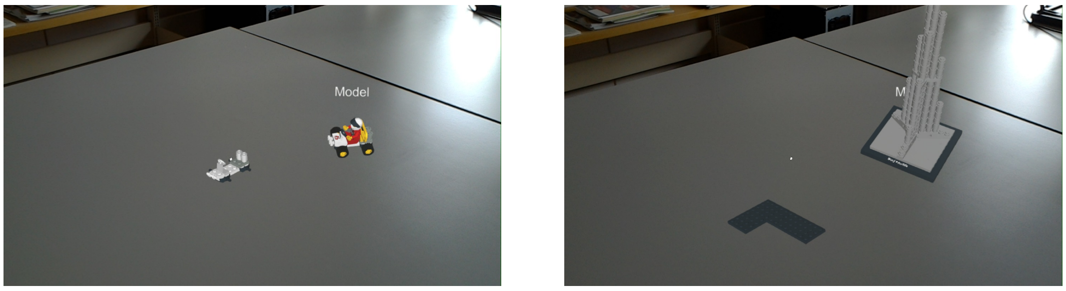


(a) Added lines with bad performance.

(b) Decent performance without lines.

Figure 4.2: Performance difference when adding lines to the model.

Scaling of preview Scaling of the model is normally not a problem since the input files have the measurements in real life and the model that gets shown on the HoloLens can be scaled by the user. The only real issue we have is the preview in the upper right corner. It has a fixed size and moves with the user so it will always stay at the same place. If we now have a bigger figure, like for example a whole building that is 30 centimeters tall, it can happen that parts of it will be out of the field of view, in case it is not scaled down as seen in Figure 4.3 (the actual field of view when wearing the HoloLens is smaller as on the figures). Other, smaller models will be, when scaled down, too small to see. The best approach we have is to define a bounding box around the model and scale the model according to that.

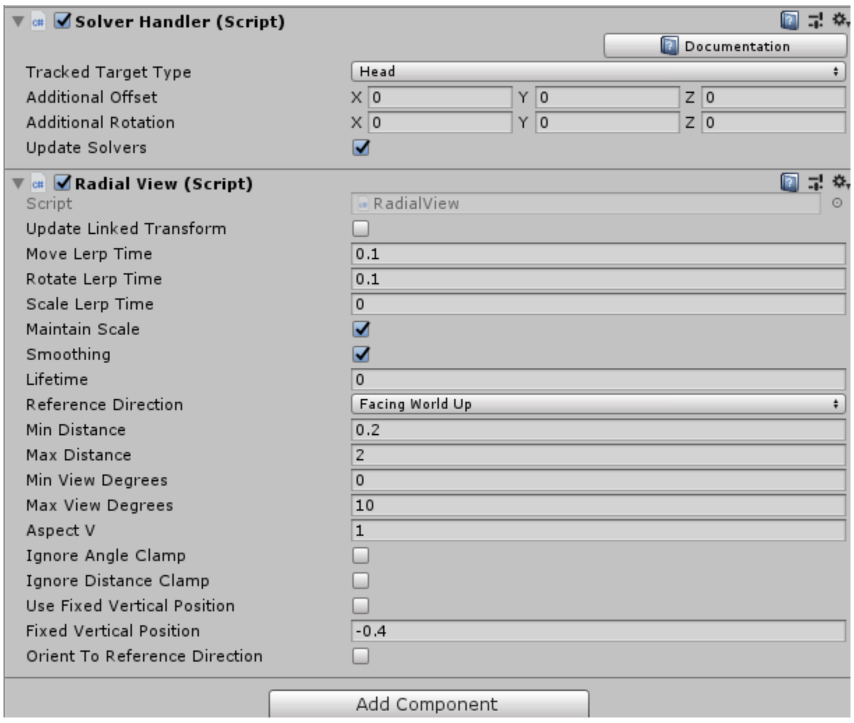


(a) Good scaling with a small enough model. (b) Bad scaling with a too big model.

Figure 4.3: Preview scaling of two different sized models.

Parsing We had serious problems on multiple occasions, when we were trying to parse the files, simply because of the fact that the model description files we are using are from the official model repository [15]. Even though the files there should be all the same and follow the specification from the LDraw website, they still slightly differ. The fact that not every model description has defined steps, or some just have them in the sub-files, makes it hard to find out all individual steps and also which bricks are used when.

Model Picker Billboard On the start screen, where we pick our model, we have a canvas with user interface (UI) elements on it. With our first design, once the user looked away from it, the canvas with the text and the drop down box stayed where it was. When returning back to it from one instruction, it was where the app first booted up. But we want it to behave more like a billboard that follows the users gaze. For that, we had to add a Solver Handler script that tracks the head movement and Radial View script that defines the parameters like max and min view degrees and min and max distance of the canvas, pictured in Figure 4.4.



Solver Handler (Script) ✓

Tracked Target Type	Head
Additional Offset	X 0 Y 0 Z 0
Additional Rotation	X 0 Y 0 Z 0
Update Solvers	✓

Radial View (Script) ✓

Script	RadialView
Update Linked Transform	☐
Move Lerp Time	0.1
Rotate Lerp Time	0.1
Scale Lerp Time	0
Maintain Scale	✓
Smoothing	✓
Lifetime	0
Reference Direction	Facing World Up
Min Distance	0.2
Max Distance	2
Min View Degrees	0
Max View Degrees	10
Aspect V	1
Ignore Angle Clamp	☐
Ignore Distance Clamp	☐
Use Fixed Vertical Position	☐
Fixed Vertical Position	-0.4
Orient To Reference Direction	☐

Figure 4.4: Scripts added to the model picker to make it move along with the user.

Runtime model adding Even though the idea in the beginning was to implement runtime adding from new models, it turned out to be harder and more complicated. At first we fixed the model to the app, depending on the chosen model in the beginning, we loaded a prepared scene. That worked but was not optimal since we want the models loaded at runtime.
The second approach was to put every model we have in the resources folder in the application, so that the model files would be part of the build and can be loaded at execution. The problem here is that all models and files for them have to be written before building the app

and once built we can not add any new LEGO builds. Every model .ldr file has to be preprocessed with the Python script and the created files saved in the build directories before we can build our application for the HoloLens.
Another way to load resources at time of execution of an already built app in Unity, is to use asset bundles[3]. These can be loaded from everywhere on the file system where the app runs, or even better, from a web server. The problem is the building of these packages. While before Unity 5 it was possible to build them via a C# script, with Unity 5 it changed and is now only possible in the editor.
For the user that means, for creating asset packages with newly preprocessed models, it is necessary to open Unity and create the bundles there and then copy it to the HoloLens. That is almost the same amount of work as if we just recompile the Unity app and deploy it to the HoloLens again. In the end we implemented a mix form, we are building asset bundles "per hand" and put them in the streaming asset folder. Files from there get copied to the device and will be at the same location where we could store assets from web requests (more on that in Chapter 5.3). From there we can load all our resources.

4.1 Conclusion

In conclusion we can say that even though we had some problems along the way, we were able to fix them in a reasonable way to make our application work and fulfill the requirements. We came up with a working prototype app that runs on the HoloLens but we also found the limitations of our implementation. We saw that the color/material loading in Unity still has problems, that the insertion of lines is, in terms of frames per second, not tolerable and that large models do not work with reasonable performance.

[3]https://docs.unity3d.com/Manual/AssetBundlesIntro.html

5 Outlook

Our instruction workflow works well for most of the models but there are also many things or features that could be improved, optimized or added.

5.1 Optimizations

Following things are working fine in our application but could be optimized for a smoother and more efficient execution:

Unseen faces Most of the meshes given in the model repository have faces defined that we do not need in our application. This is because in the .ldr or .mpd files, every brick is described entirely. If we take for example a pyramid that has four layers, we would not need all the bottom faces and insides of the bricks on the top 3 layers. Just the outside, the hull of the model would be sufficient. What makes it difficult, is how to find out which faces to neglect without looking at each set before and deciding by hand.

5.2 Additions

Some things that can possibly be added to expand the functionality:

Brick detection This addition would be useful if users have a pile of bricks and do not know which of them they really need for the model. The idea is to implement an algorithm that detects the bricks required for the current step out of the pile and highlights it.

Model detection The idea is similar to the last point, but with the difference that the application should detect the whole model we are currently building and overlay it with our instructions. If the users move their head or the already connected bricks, the overlay should of course stay on the model.

Bricks per step To have a better understanding of what parts are used, an idea would be to have an overview of which individual bricks are used each step and to show them somewhere on the screen, similar to the small model we already have.

5.3 Possible Changes

Changes that may improve the application:

Input files One potential fix for our problem with the runtime model loading would be to use another file format for the preprocessing output and application input. Preferably one that can be imported directly to Unity and works better than the .obj parser but still without having to write it on our own.

Model preview The currently implemented way on how to show the preview model is to copy together all the step models in the preprocessing step and then load this new model. By not optimizing and deleting unseen vertices and faces inside of the mesh, this step is unnecessary. We could just load all the step meshes together and display them as one object.

Runtime loading of model At the moment, the asset bundles that contain the models get saved and loaded from the streaming assets folder because it gets copied to the final build location. The idea behind this is to let the application get the assets from a web server, download it to the streaming assets folder and load it from there. To do so, we could make a web request every time the model picker is opened and check if there are any new models on the server. That could even go so far to just get the model file list every time and just download the picked model to keep the web traffic as low as possible.

Bibliography

[1] Stavros Antifakos, Florian Michahelles, and Bernt Schiele. “Proactive Instructions for Furniture Assembly.” In: *UbiComp*. Vol. 2498. Aug. 2002.

[2] L. Frank Baum. *The Master Key. An Electrical Fairy Tale Founded Upon the Mysteries of Electricity*. ebook. Urbana, Illinois: Project Gutenberg, 1996. URL: http://www.gutenberg.org/ebooks/436 (visited on 07/15/2020).

[3] The Pokemon Company. *Pokemon Go*. URL: https://www.pokemongo.com/en-us (visited on 07/23/2020).

[4] Patrick Dähne and John N. Karigiannis. “Archeoguide: System Architecture of a Mobile Outdoor Augmented Reality System.” In: *Proceedings of the 1st International Symposium on Mixed and Augmented Reality*. 2002, pp. 263–264.

[5] Juan Manuel Davila Delgado et al. “A research agenda for augmented and virtual reality in architecture, engineering and construction.” In: *Advanced Engineering Informatics* 45 (2020), p. 101122.

[6] Karl Fluch. *Österreich, flach gelegt: Die Wahlgrafiken des ORF*. Sept. 30, 2019. URL: https://www.derstandard.de/story/2000109270202/oesterreich-flach-gelegt-die-fliegenden-wahlgrafiken-des-orf (visited on 07/19/2020).

[7] Python Software Foundation. *Python*. URL: https://www.python.org (visited on 07/23/2020).

[8] Google. *Google Glass*. URL: https://www.google.com/glass/start (visited on 07/23/2020).

[9] Google. *Google Lens*. URL: https://lens.google.com (visited on 07/23/2020).

[10] Xinxing Gu. *Google Translate's instant camera translation gets an upgrade*. July 10, 2019. URL: https://blog.google/products/translate/google-translates-instant-camera-translation-gets-upgrade/ (visited on 07/15/2020).

[11] IBM. *IBM Max Reality*. URL: https://www.ibm.com/products/max-reality (visited on 07/19/2020).

[12] IKEA. *IKEA launches IKEA Place, a new app that allows people to virtually place furniture in their home*. Oct. 12, 2019. URL: https://newsroom.inter.ikea.com/news/ikea-launches-ikea-place--a-new-app-that-allows-people-to-virtually-place-furniture-in-their-home/s/f5f003d7-fcba-4155-ba17-5a89b4a2bd11 (visited on 07/15/2020).

[13] Jarvish. *Jarvish AR Helmet Kickstarter*. URL: https://www.kickstarter.com/projects/308263621/jarvish-the-smartest-motorcycle-helmet-ever-made (visited on 07/23/2020).

[14] LDraw.org. *LDraw*. URL: https://www.ldraw.org (visited on 07/23/2020).

[15] LDraw.org. *LDraw Official Model Repository*. URL: http://omr.ldraw.org (visited on 07/23/2020).

[16] Katie Malczyk. *Let Google be your holiday travel tour guide*. Dec. 13, 2019. URL: https://blog.google/products/maps/let-google-be-your-holiday-travel-tour-guide/ (visited on 07/15/2020).

[17] MeshLab. *MeshLab*. URL: https://www.meshlab.net/ (visited on 07/23/2020).

[18] Microsoft. *Microsoft HoloLens*. URL: https://www.microsoft.com/en-us/hololens (visited on 07/23/2020).

[19] Microsoft. *Mixed Reality Toolkit for Unity*. URL: https://microsoft.github.io/MixedRealityToolkit-Unity/README.html (visited on 07/23/2020).

[20] Paul Milgram and Fumio Kishino. "A Taxonomy of Mixed Reality Visual Displays." In: *IEICE Trans. Information Systems* vol. E77-D, no. 12 (Dec. 1994), pp. 1321–1329.

[21] Rajan Patel. *Google Lens: real-time answers to questions about the world around you*. May 8, 2018. URL: https://blog.google/products/google-lens/google-lens-real-time-answers-questions-about-world-around-you/ (visited on 07/15/2020).

[22] Pejman Saeghe et al. "Augmenting Television With Augmented Reality." In: *Proceedings of the 2019 ACM International Conference on Interactive Experiences for TV and Online Video*. TVX '19. Association for Computing Machinery, 2019, pp. 255–261.

[23] Kelly Schaefer. *Get outfit inspiration with style ideas in Google Lens*. Oct. 3, 2019. URL: https://blog.google/products/google-lens/get-outfit-inspiration-style-ideas-google-lens/ (visited on 07/15/2020).

[24] Ivan E. Sutherland. "A head-mounted three dimensional display." In: *AFIPS '68 (Fall, part I)*. 1968, pp. 757–764.

[25] Unity Technologies. *Unity*. URL: https://www.unity.com (visited on 07/23/2020).

[26] WayRay. *WayRay Embedded Holographic AR Display*. URL: https://wayray.com/embedded (visited on 07/23/2020).