Universität Regensburg

Philosophische Fakultät III
Sprach- , Literatur- und Kulturwissenschaften
Institut für Information und Medien, Sprache und Kultur (I:IMSK)
Lehrstuhl für Medieninformatik

Interaktionstechniken und -technologien
Modul: MEI-M32.1+2
SS2019
Leitung: Hahn Jürgen

ArucoFootball-Project

Jonas Puchinger
Matr.-Nr.: 1812110
1. Semester M.Sc. Medieninformatik
E-Mail: jonas.puchinger@stud.uni-regensburg.de

Daniel Schmaderer
Matr.-Nr.: 1696560
3. Semester M.Sc. Medieninformatik
E-Mail: daniel.schmaderer@stud.uni-regensburg.de

Maximilian Kögl
Matr.-Nr.: 1807588
1 Semester M.Sc. Medieninformatik
E-Mail: maximilian.koegl@stud.uni-regensburg.de

Abgegeben am 02.08.2019

Content

1 Motivation and Use Case

Normally, a football coach is standing in front of a tactics table with colored magnets and a marker and is trying to explain the new tactics for an upcoming match to his team. While he does that, it is hard for the team members to see the whole board, because the coach is usually directly in front of the board handling the magnets. It is also hard for each team member to memorize their specific magnet and its location. To help the coach to teach his tactics to the football team, we implemented „ArUcoFootball“, a modern approach to tactic boards in augmented reality.

With „ArUcoFootball“ the football coach can show the tactics using a beamer to project camera input of markers on a wall. The players can see a 3D-model of themselves and their number on those markers. With this, the football coach does not have to stand in front of the team and can move the markers to explain different tactics. He also has the possibility to change players with a control device and put new players on the field. The models represent each player and with their numbers on the model, it is easier to know where they are.

2 Implementation

The implementation of „ArUcoFootball“ requires different steps and aspects that are explained in the following. The application is written in Python 3.7.3 using the PyQt5 framework. The user interface of the application is divided into two parts, the camera window, where the user can see his placed models rendered onto the markers, and a control window, where the user can set and modify the models associated with certain markers.

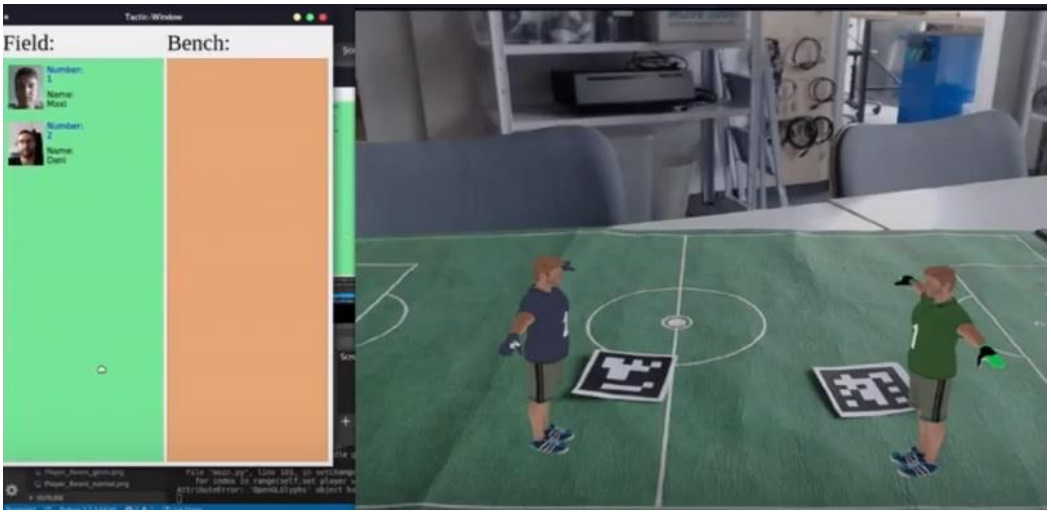

Illustration 1: Screenshot of the UI with two rendered models and markers on the right side and the drag and drop lists on the left side with two players set on the field.

2.1 ArUco marker detection

An „ArUco“ marker is a square with a black border and an inner binary matrix that stores the ID of the marker. We use an online marker generator to create markers [1]. For the recognition of the markers, we use the Open Source Computer Vision Library, commonly known as OpenCV. The corresponding Python module is named cv2. With functions provided by this module, we are able to detect the individual markers, their IDs, corners, and rotation as well as translation vectors of the markers in said corners [2]. First, marker candidates in a calibrated video or image feed are determined by their black border. After that, markers are compared to a dictionary of binary matrices of a certain size. In our implementation, this matrix has a size of 6x6 bits per marker matrix. The data stored in this matrix decides the ID of a given marker. Through this ID, the marker can be associated with a model to display, more on that later.

2.2 Pose estimation

The next step is to estimate the pose for every marker that gets detected. To do so the calibration parameters for the camera need to determined. These calibration parameters are the camera matrix and the coefficients of the distortion. In our case, we used the default calibration images and code from the OpenCV

documentation to calculate the parameters [3]. With the parameters, corners, rotation and translation vectors the pose of each ArUco marker can be estimated. The local coordinate systems for all detected markers are established. This enables the correct placement of the 3D models on those markers in terms of size and orientation of the model.

2.3 Model rendering and placement

As a model, we used a free 3D "Football Player" model in the Wavefront OBJ (.obj) format [4]. Unnecessary parts of the model were deleted to decrease the size. Further, the model was adjusted in terms of size and orientation. We also reworked the original material of the model and created four different ones to distinguish individual player models. The actual parsing and loading of the models and their respective textures were implemented through objectloader.py based on the OBJFileLoader, which uses pygame and OpenGL as imports [5]. The objectloader.py takes the football player objects and their mtl files - including textures for the object - and prepares them for the rendering process with OpenGL. It is important that only one model per file is allowed and diffuse color and texture are used [5]. The models are placed onto the markers with the help of the estimated coordinate system from the previous step, and estimated parameters for the models themselves [6].

2.4 Interaction techniques

There are three different techniques to interact with „ArUcoFootball". Those are pointing on the graphical user interface or camera output window, drag and drop to assign individual player models to certain markers, and the option to undo or redo the setting of player models.

2.4.1 Pointing with Wiimote

At first, there is pointing with a Wiimote. This is done by first connecting to the Wiimote via its MAC address [7]. After that, the Wiimote needs to be pointed at a physical board equipped with four infrared LED lights, which are positioned at four corners and powered via USB. With this, the Wiimote can detect those lights

and track them. The points are projectively transformed to map them from their source coordinate system in the real environment onto the coordinate system of the screen [8]. This is accomplished by calculating two different matrices, one that transforms the source coordinates to an identity matrix, and another one that transforms this identity matrix to the destination matrix of the screen. These two matrices are multiplied in the order identity_to_source times source_to_identity to receive the transformed x and y coordinates. The calculated coordinates are then dehomogenized by dividing each x and y through the corresponding calculated z coordinate. With the computed x and y coordinates we can set the mouse cursor the point defined by them. In the code, this functionality is provided by PyQt5. It enables the mapping of the cursor position on a global screen level or a local widget level. Furthermore, the shape of the cursor icon can be adjusted. We use this to indicate the drag and drop functionality in our application.

2.4.2 Drag and drop

For the drag and drop function, we made custom list widget items and used an already existing drag and drop code for PyQt4 [9], which we adjusted for PyQt5. The custom list widget items contain a picture of the player with a specific scaled height, the name of the player and their player-number. Those custom widgets can be dragged and dropped from one list widget to another. The two list widgets are directly placed next to each other in the user interface of our application. Via mouse or Wiimote a user is able to select player list items and drag them assign them to the list, he chooses to.

2.4.3 Undo and Redo

The undo and redo functionality is implemented by hand, not via the implemented Qt functions. This was done, because of the way the drag and drop were implemented and enclosed from other code encroachment. To enable the undo and redo functions a signal to the list widgets was created, which fires every time an item in the list gets changed. With this, the data from every custom widget item within the two list widgets gets copied separately and saved as a tuple in an array. This also increments a counter. Excluded from this counter are the initial

custom list widget items, which get added at application startup. If the undo function gets triggered, the counter is decremented and loads the saved data from the previous state. The redo function increments the counter and loads the prior state. To load the states all lists must be cleared and every custom widget has to be loaded again. When reloading the custom widget items, they are not added to the stack array. If the user performs an undo-action and continues to change players afterward with drag and drop, the array used to store the states will be set to delete all states which were redoable.

2.5 Program structure

Our starting point is the main.py file. We set here the necessary variables and initialize everything we need. At first, we create a wiiModel from our WiiModel.py which uses the wiimote.py file to connect to the Wiimote with the given MAC address. With the drag.py will our main window, drag and drop, and undo/redo functions be set as the next step. Then we load our player models and initialize OpenGL, which uses the objloader.py for the models and player.py for different player classes. After this, we initialize a thread with the wiiModel to get the Wiimote input and change the cursor position accordingly. To change the cursor position right, the wiiModel uses the ProjectiveTransformation.py file to compute where to point. Afterward, we start another thread which handles the OpenGL loop for model rendering, marker detection, and the webcam. The aruco_detection.py and the webcam.py files are used there for this purpose. At last, we execute our Qt main window loop. This has to be done lastly because Qt requires to be run in the main thread.

2.6 Hardware setup

There are some hardware solutions required to successfully use this project.
The first one is a working camera, that is connected to the computer the program runs on. During testing, we used different webcams, which all gave similar and good results. The second requirement is a tracking element to enable the pointing functionality with the Wiimote. We used a wooden board with four infrared LEDs, but this is interchangeable. The LEDs could also be attached to the screen

of the computer used to run the application or an external screen. Choosing the right tracking element depends on the situation the application is used in. In the use case of a tactics board for a football team, a bigger screen is preferable, but only in combination with an external tracking device such as the wooden board, because otherwise the screen would be covered by the user doing the pointing.

2.7 Pitfalls and Workarounds

While implementing the application, there were some pitfalls and caveats that occurred. These are discussed in the following, as knowing about them and possible workarounds or solutions make alternative and future implementations easier.

As already mentioned, during development we tried different cameras. They all provided similar results, but using a camera with a high resolution gives the best picture and makes marker detection even better. It further proved to be practical to use an external camera instead of an integrated webcam, because the added flexibility makes it much easier to adjust the camera setting to the given presentation view.

Another point we initially underestimated was the quality of the ArUco markers. It is vital for the detection of the markers, that they are printed with high enough dpi and a big enough size. Further, it proved to be very useful to attach the markers to a small board to stabilize them and prevent them from getting wrinkles and dog-ears. It is also interesting to note, that the size of the rendered models adapts to the size of the physical markers, so choosing the correct size for a given presentation setting is important. The detection of the markers also is improved by the physical background they are placed on. It proved practical to use a unicolored background with no lines or patterns on it, which also provides enough contrast to the markers.

When using the application on slightly different technical setups during development, there often occurred a problem with different versions of the aruco module of cv2. The command aruco.estimatePoseSingleMarkers(corners, 1, self.mtx, self.dist) returns either two or three parameters, depending on the version. Using the wrong destructuring assignment broke the program at runtime.

The solution for this is to adjust the destructuring to the given technical setup. We solved this problem by checking for the length of the tuple the mentioned command returns and destructuring accordingly.

One problem occurred when using the program in a virtual machine. While implementing the pointing with the Wiimote, we used the code in the corresponding section of this report. However, setting the cursor only works, if the input option "mouse integration" of the virtual machine is toggled off.

Another point of contention is the light setup for a presentation with "ArUco Football". Marker detection is enhanced is enough light is present to ensure high contrast between the markers and the background. The tracking devices for the wiimote on the other hand operates best if low light is present. Choosing a balanced setup is therefore vital to running the application in the best way. We found that separating the presentation area and the tracking device made this task a lot easier.

A big problem is the long loading time of the 3D-models, that are rendered onto the markers. Our implementation uses OBJ models with MTL textures. The models are parsed and loaded by an objectloader taken from the project pygame at application startup [6]. Due to using a somewhat complex model with different textures, startup times are long. We tried to pre-load the parsed models, but this brought no performance improvements. We, therefore, suggest using simple models and textures. The problem with the complex models also ties in with the general performance problems of the application. Rendering and displaying complex models causes the application to lag, especially when more models are rendered at the same time. This is also amplified when running the application in a virtual machine. Input lag is also caused when connecting the computer to an external screen. This makes pointing more difficult and less intuitive to use. Although we were not able to test it, we suggest using a native OS instead of a virtual machine to reduce lag and performance issues.

We also encountered a bug, that caused our application to crash when using different thumbnail images in the list items used to represent players. The bug was not encountered if the same image was used for every player. We used a try and

catch block with an exception to prevent our program from crashing if the bug occurs but we were not able to reliably reproduce or fix it.

3 Alternative and future Use Cases

For this project, we focused on a specific use case, namely an augmented reality tactics board for football. However, with a little adjustment, the project can be used for different scenarios. A great alternate use case would be to switch our models of football players to models of different objects and furniture. This enables users to create and rearrange certain architectural settings without the use of specialized software. On top of that, the AR component of the application makes it easy to place the models in a real environment, which enhances the immersion of designing a real room. Alternative themes for models are designing an outdoor space, tactic boards for other team-based sports, or games where player models are presented in augmented reality.

Bibliography

[1]https://tn1ck.github.io/aruco-print/

[2]https://docs.opencv.org/4.1.0/d5/dae/tutorial_aruco_detection.html

[3]https://docs.opencv.org/master/dc/dbb/tutorial_py_calibration.html

[4]https://clara.io/view/59a3fbf4-286a-492a-a341-ca6edccd0e7c

[5]https://www.pygame.org/wiki/OBJFileLoader

[6]https://rdmilligan.wordpress.com/2015/10/15/augmented-reality-using-opencv-opengl-and-blender/

[7]https://github.com/RaphaelWimmer/wiimote.py

[8]https://math.stackexchange.com/questions/296794/finding-the-transform-matrix-from-4-projected-points-with-javascript

[9]https://stackoverflow.com/questions/41595014/dragndrop-custom-widget-items-between-qlistwidgets