Projected Augmented-Reality

Calibrating Glasses-free AR Systems

Final Report

Instructor: Dr. Edward A. Fox

Client: David Franusich

Team: Siddharth Akalwadi, Zachary Long, Shorya Malhotra, Dilan Prasad

Date: December 14, 2021

CS 4624: Multimedia, Hypertext, and Information Access

Virginia Tech, Blacksburg, VA 24061

Table of Contents

Table of Figures

Table of Tables

1. Executive Summary

Augmented and Virtual Reality are expected to have an enormous impact in the future. This is supported by Meta's (formerly Facebook) unveiling of an upcoming virtual world where users will be able to connect with others seamlessly, as if they were all in the same room. In the future, users will want to experience these Virtual Reality worlds without the need for heavy headsets that are clunky and uncomfortable. Our project eliminates the need for personal devices by bringing these mixed-reality worlds to the natural eye with the interaction of cameras, software, and projectors. In this report, we discuss a calibration system we developed for a projected augmented reality system.

Our final product required the creation of four major deliverables. Firstly, we needed a method for detecting the corners of the calibration board. This was done with a C++ script which made use of the computer vision library OpenCV to detect ArUco markers and return their relative pixel distances. Secondly, we needed a Qualisys Motion Tracking System to track the relative motion and position of physical rigid bodies within the projector's frustum. This data was then sent to our third deliverable, a Unity scene, which processed image data and adjusted the projection. Our last deliverable was this report which details how we developed our final system so that users and developers of the future can adapt it easily to fit their own use cases.

At a higher level, our specific goal in designing and developing a Projected Augmented-Reality system was to deliver a product that allows anyone, with little to no technical background and the necessary hardware and software, to get their system calibrated. Whether projecting a virtual world to meet with friends, learn from professors, or aid in rapid prototyping, our system will afford users an immersive experience to solve a broad variety of problems.

2. Introduction

The goal of this project was to design a calibration system that would allow users to dynamically display digital models on physical objects as they move within the projector's frustum in 3D space. Throughout our development process, we envisioned a system that would provide users the ability to experience Augmented and Virtual Reality worlds without the need for clunky goggles.

While this type of project has a vast range of future applications, our client had detailed near-future applications that involved both creative and technological endeavors. Our final product affords users the opportunity to put on immersive audio and visual performances and rapidly prototype user interfaces for future Augmented and Virtual reality programs.

Though this product was being developed at the behest of Dr. Edward Fox under the guidance of David Franusich, the outline was very vague. In other words, we were given a lot of freedom in terms of how we developed this system. That meant we had to try out numerous ideas and rapidly prototype them to see which ones succeeded and which ones were simply not feasible. All in all, we ended up developing a complete calibration system that coherently integrates all the necessary hardware and software requirements with the push of a single button.

One of the most interesting features of our project is the fact that our final design allows artists and programmers to display images on moving objects, giving audience members the illusion that a tangible object has 3D graphics projected on each of its sides. Providing the user with an immersive yet seamless experience was something we valued highly in the development of our project.

Every step of our development process had future users and developers in mind, and since we started our project from scratch, we made sure to thoroughly document every step of the process. So, our report begins with a brief overview of the prior work in the field of our project. From there, we delve into the necessary requirements users must meet before employing our system. After that, we delve into the technical details of our implementation and how we tested it. Then, we reflect on some problems we faced along with how we solved them. Lastly, we discuss the future applications of our project and how we see it as a steppingstone leading up to a world of glasses-free Virtual Reality.

3. Prior Work

Projected Augmented Reality is technology that directly overlays digital projections onto the real world [1]. There have been attempts to solve what our project aims for, but in different ways. Below are brief descriptions of two such recent efforts.

3.1 Lightform

Lightform, a company that has created tools for projection mapping, is one of the leaders in this industry and deals with projected Augmented Reality. This company has developed their own hardware and software that helps calibrate a projector to work with their system to project onto uneven surfaces. They also have their own portable projector that contains the whole system that can more dynamically project on static abnormal surfaces. The difference between our implementation and theirs is that they have a product that works only on static surfaces while ours is aimed to work on moving objects.

Lightform uses structured light patterns to calibrate their projector and allow the hardware to be able to capture what it sees. This allows the user to project whatever they would like on the static object. The calibration process is automated and can be completed easily, which is an essential aspect we focused on in our project. We aimed for a system tailored to user accessibility and usability due to the complicated and technical nature of our development.

More information can be found on the “Design Boom” website [2].

3.2 Projecting on Moving Objects

A research group from the University of Tokyo has developed a system for projecting images onto moving objects. Their system makes use of a series of mirrors to place the projected image onto the physical object. Using a motion sensor to track the object, the mirrors rotate to cast the projected image onto the moving object.

This project is incredibly exciting and responsive to deformation and movement. This project and the one we developed do however differ in the desired outcome. For our project, the client wanted to be able to project onto multiple objects. However, the mirror system used by the Tokyo researchers would need to be duplicated for each object in the scene and as of now can only project on a single object.

More information can be found on YouTube [3].

4. Requirements

Our project would not have been possible without the aid of three major resources: our client, software capabilities, and hardware provided by the Moss Arts Center [4].

4.1 Client

Our project was developed under the guidance of David Franusich. David works in the Institute of Creative Arts and Technology (ICAT) [5] Perform studio as a dedicated research assistant. He oversees many internal research projects that blend creativity with technology.

David Franusich hoped that when completed, our product would serve three major functions. Firstly, it would help calibrate any projector given the Unity Software [6] and a Microsoft Azure Kinect Camera [7]. Secondly, it would be able to automatically calibrate said projector to display images that match what is seen in Unity 1:1. Lastly, our project aimed to be able to project said images on objects moving in all three axes.

Upon automation of these three fundamental functions, David Franusich plans to use our system for dynamic audio-visual performances. However, this application is only for the foreseeable future. Once our system is up and running, he plans to open access to our system to other Computer Science researchers so that they may take on more intensive tasks such as rapidly prototyping user interfaces for Virtual Reality worlds. These creative applications of our product both excite and push us to develop an easy-to-use system.

4.2 Software

Before we could even begin to think about how users will interact with the system, we needed to develop robust software that automated tasks we were initially doing manually. Our final design fed Unity a C++ script that uses the Azure Camera to find ArUco markers [8] [9]. Sample ArUco markers are shown in Figure 1.1. The C++ script allowed Unity to get relative pixel distances from the markers' neighbors. From there, we fed Unity tracking objects as they entered the projector's view.

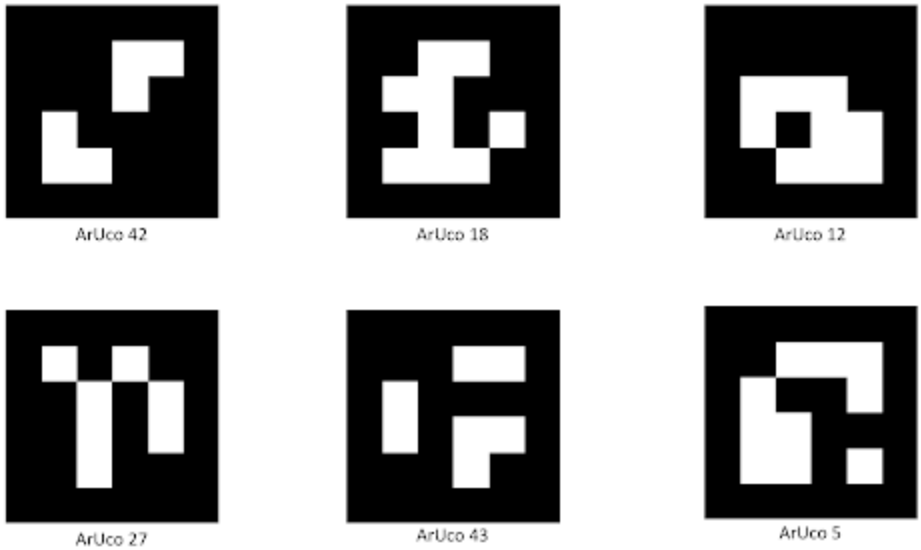

Figure 1.1: Sample ArUco Markers

A C++ computer vision library called OpenCV [10] allowed us to detect ArUco markers from captured images. Functions for finding ArUco marker offsets and scales were written in C++ using the OpenCV library and ArUco module. These functions were then compiled and exposed to Unity in a Dynamic Link Library (DLL) which served as a plugin to Unity. Unity functions then called the exposed functions from the DLL to determine how to pan and zoom the camera in Unity.

All the code that our team developed can be found at the GitHub Repository. [11]

4.3 Hardware

Our system is nowhere near real-world consumer use as the hardware that it requires is far too advanced and expensive for any individual to reasonably implement. Firstly, users need an entire Qualisys motion camera tracking system [16] that can talk with Unity to track objects entering/exiting/moving within the projector's view. This system alone costs around $40,000 and was made available to us to use by the Moss Arts Center.

Secondly, the user of our system needs an Azure Kinect Camera that can be read by the C++ script to detect ArUco markers and their relative pixel distances.

Arguably the most important part of our system's implementation is the key that ties everything together: a powerful desktop computer that can accept position data in real time from the Qualisys Motion Camera system as well as the Azure Kinect Camera. While receiving all this input data from our cameras, it requires the power to simultaneously run our Unity scene and track the data points it is receiving: all in real time.

5. Design

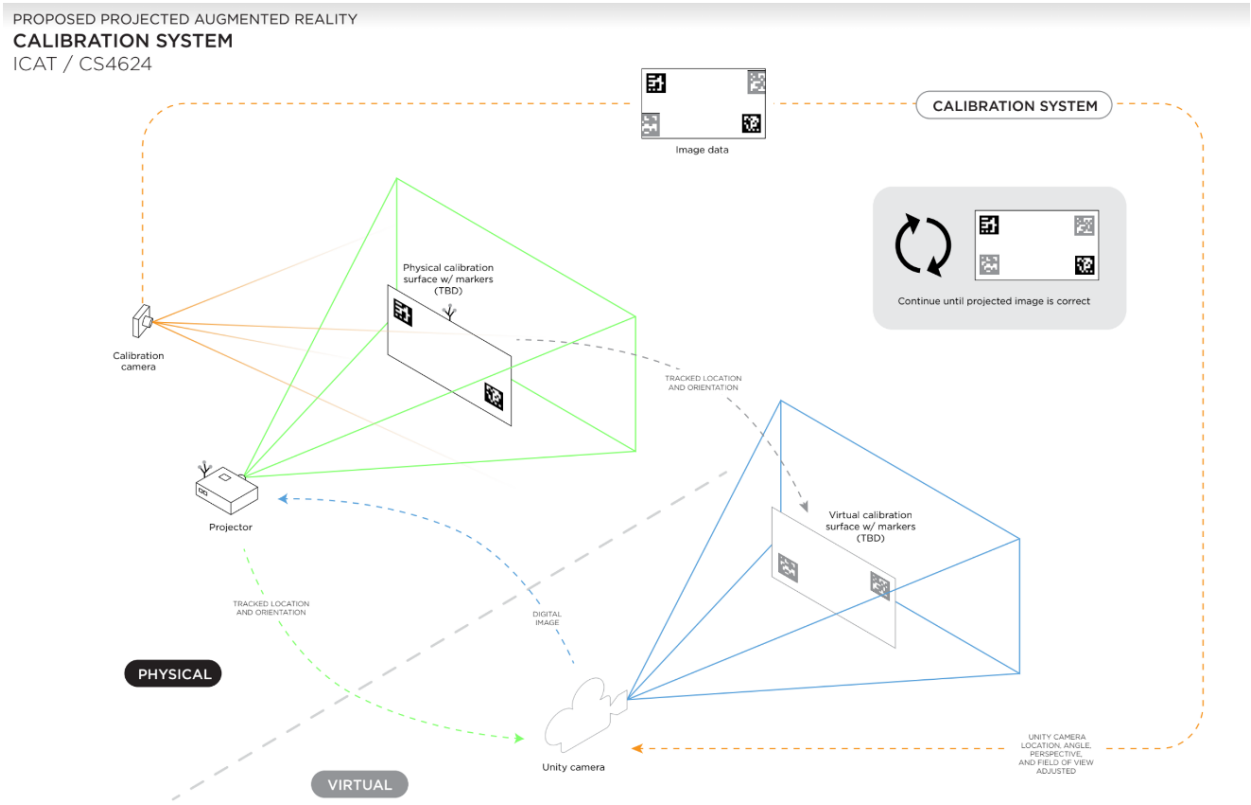

Figure 1.2: Design Overview

Figure 1.2 is a diagram illustrating how our system works. In the physical space, there is a projector and the pyramid shaped space that it can project to, called the frustum. This space is where the user can have physical objects once the system is calibrated. There are two sensor systems in place for calibration - the Microsoft Azure camera and the Qualisys motion trackers. This design requires that the projector and calibration board have been marked with reflective tape and created as objects in the Qualisys software. This allows us to send position data in real time to the other half of the project in Unity.

Our Unity project consists of a virtual camera with a similar frustum to the projector. However, this is the area defined by the horizontal and vertical fields of view that the camera can see. The calibration board is also modeled as a 3D object in Unity and moves with the position data. Our goal was to make the viewing frustum of the virtual camera (which can be changed easily) match the projector. Some variables that can change the viewing frustum among different projectors include manual lens shifting, differences in the throw type of the projector, and the intended mounting setup of the projector (which affects its projection angle). When fully calibrated, the Unity camera will have the same view as if the projector in real life was also a camera. This allows us to send this view from the Unity camera to the projector, and any virtual object would be projected exactly onto the physical counterpart.

To do the calibration step, we need ArUco markers on our physical board, and similar markers on the virtual one. The Azure camera, which is aimed towards the physical board, sends Unity a stream of video data, and we can use the OpenCV library to detect the position and orientation of these markers. If the projector and camera are not calibrated, the markers being projected from the virtual board won't line up with the markers on the physical board. Our code can see this difference in position and the orientation of these markers and continuously adjust the Unity camera's settings such as field of view and rotation until they line up. Overlapping the virtual and physical markers made it hard to detect, so our solution involved aligning the markers to a predetermined offset. For example, if the goal is to make the virtual marker project 10 inches to the left of the physical marker, the code would pan the camera to the left or right until that distance is achieved.

Once the system is calibrated, other objects can be brought into the projection area. If an image or video was applied to the 3D model in Unity, that would be projected onto the physical model. This can be used for rapid prototyping using augmented reality, or to create live props to make performances more engaging with the audience.

5.1 Panning

Figure 2.1 shows an example of a misaligned board and how the distances were calculated to move the virtual camera. In this case, the virtual board is white with ArUco markers on the top left and bottom right. Our virtual board is blue with a marker on the top right. The blue board is projecting near the physical one, but is clearly not matching the real one. Moving the board to the left and down a bit would align it properly. This can be seen with the difference in position between the physical and virtual markers, as shown by the red and green lines. If the virtual marker is higher than the physical one it is supposed to line up with, we would pan the virtual camera up. This sounds counter intuitive, but panning a camera up will cause objects in the scene to go down relative to it. However, the issue with this approach is if the virtual marker does not land on the calibration board to begin with, the code will not know which way to move and this solution will fail.

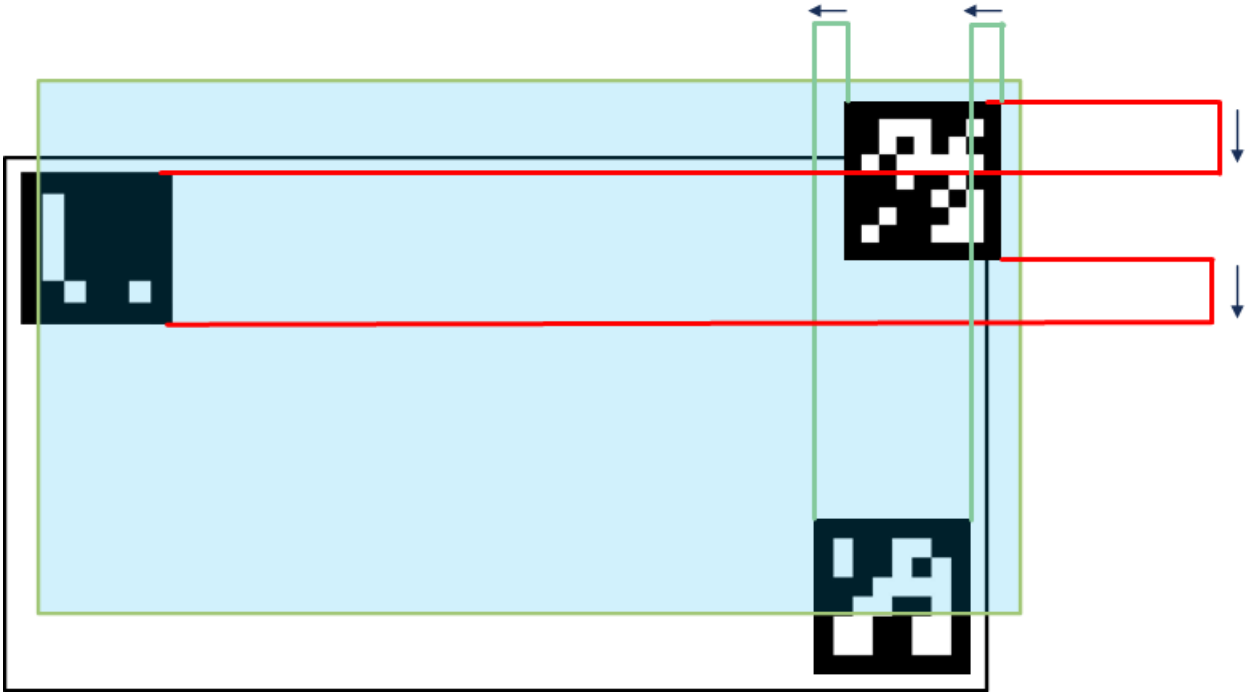

Figure 2.1: Initial Design to Handle Panning

Figure 2.2 shows a new and improved approach we took which can handle further camera angle adjustments. It consisted of a large 5x5 matrix of markers, with consecutive IDs starting in the top right corner and going down each row. Our actual image starts with ID 4 since the others were used up, so the code accounts for it by subtracting 4. Then, the row for a given ID can be found by doing an integer division of 5, and column by doing a modulus 5. Our goal was to adjust the virtual camera until the center Aruco marker at index (2,2) was on the board. If the camera angle was off by a significant amount, one of the other markers from the large image would land on the board. Using the ID of the given marker, we could tell where the virtual camera was looking and move towards the center. As shown in Figure 2.2, the virtual calibration board's original size is much smaller than the new scatter board. However, the board does not account for all angles. One potential upgrade to this system would be tiling a sphere evenly around the camera, so all angles would be accounted for.

Once the center marker can be recognized on the physical board, we did the same type of alignment as before with the Aruco marker. However, since the center marker landed in the center of the board, we aligned the camera differently by getting the midpoint of the physical marker's corners and moving the center of our virtual ArUco marker to it. Figure 2.3 illustrates this concept. The image itself could have also been offset, so it calibrates similarly to the old virtual board, but we found that having it at the center improved the chances of the image not being cut off and the OpenCV script recognizing it. One other small change we made was decreasing the speed at which the camera pans when the markers were close to lining up, and setting a threshold for error so the code did not keep trying to align it perfectly for a long time.

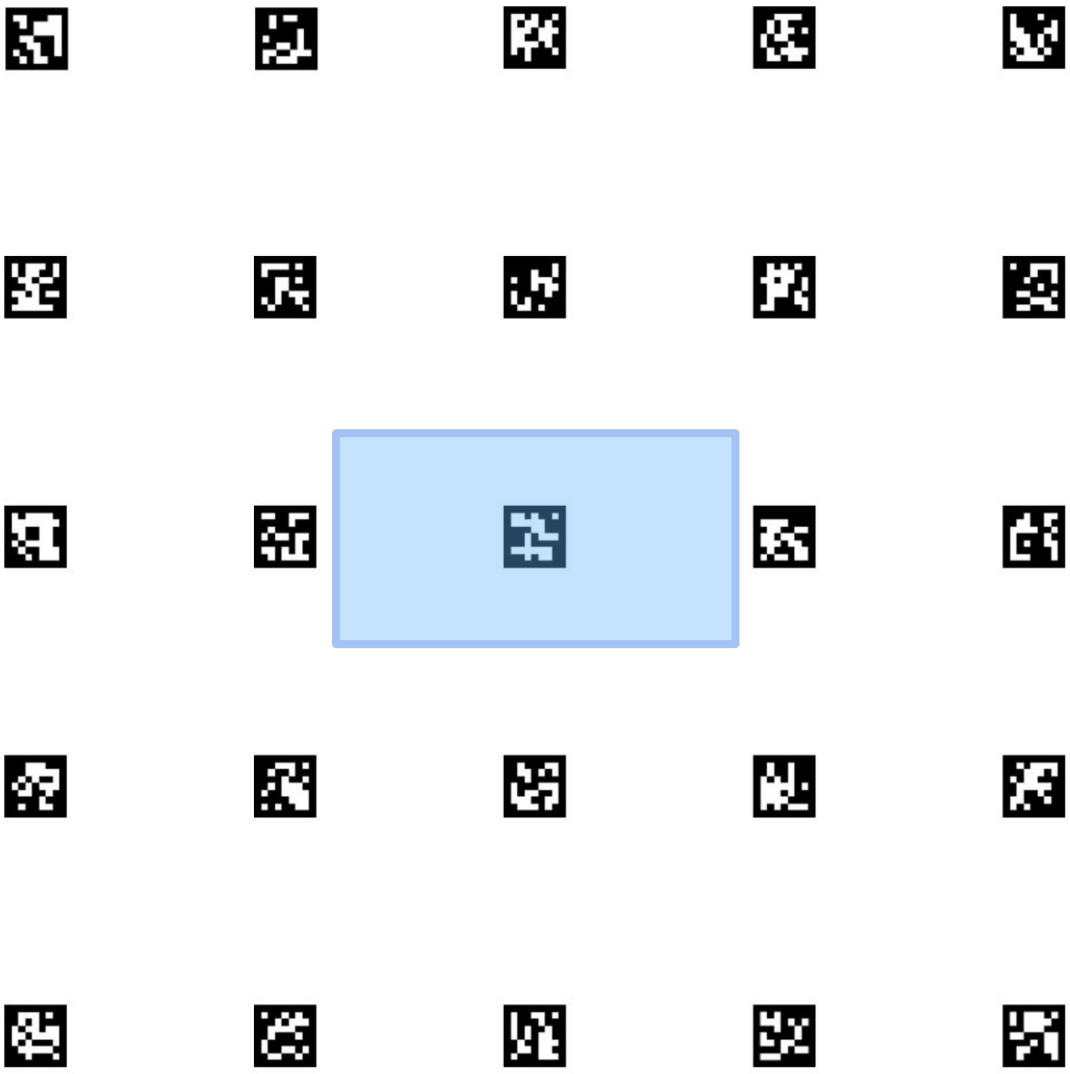

Figure 2.2: Virtual Scatter Board with Calibration Board size Comparison

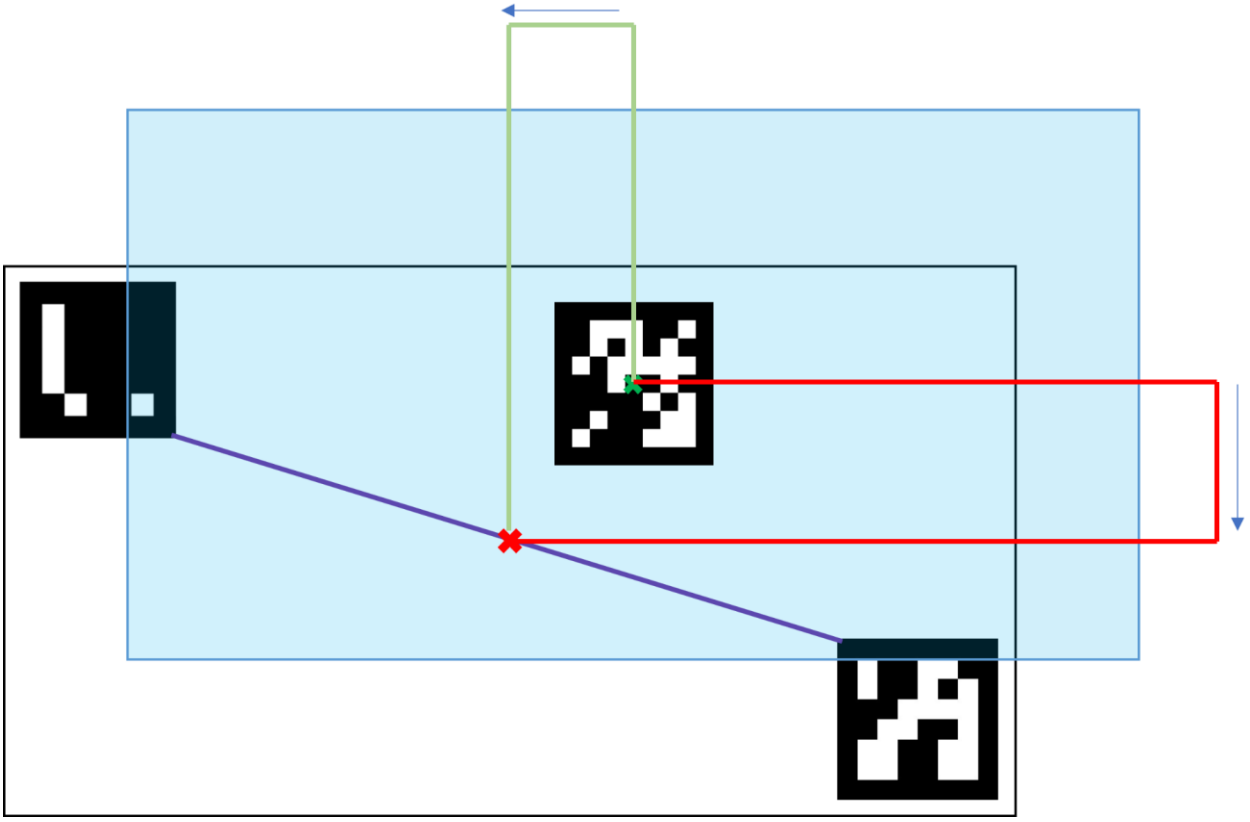

Figure 2.3: Updated Design to Handle Panning

5.2 Field of View / Scaling

Field of view (FOV) [12] defines how much the camera can see or how much the projector can project to. A short throw projector has a high FOV, meaning the angle of which light comes out of the lens is much higher. For a camera, seeing in a larger FOV angle will make objects appear to shrink since they now take a smaller area of the overall image. This FOV difference appears to have a shrinking and zooming effect of the virtual markers, as shown in Figure 2.3, but larger FOVs will also warp the image slightly. This is a key difference since simply enlarging or zooming the image will not actually align the markers, so FOV has to be set properly. This was a use case our system had to account for, so in our implementation, if our Azure camera saw that the virtual marker was larger than the physical ones, the FOV of the virtual camera was increased. The larger angle of view meant that the virtual marker took up a smaller percentage of space in the image, and appeared to shrink. We also needed a way of decreasing the FOV accordingly.

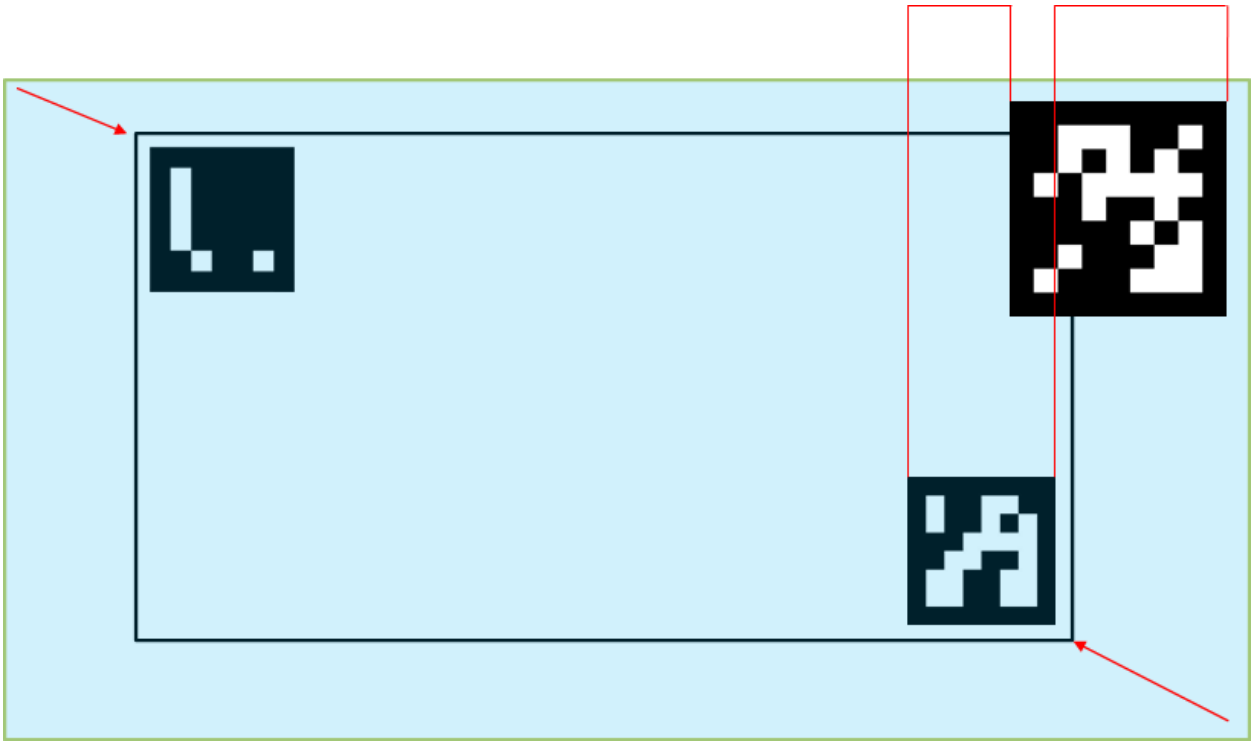

Figure 2.4: Initial Design to Handle Field of View / Scaling

Figure 2.5 shows how the new approach of the scatter board allowed us to check the relative size of the marker by drawing two diagonal lines from the physical markers. This worked out better in the case of our physical board not facing the camera exactly. One of the physical markers would be further away from the camera and therefore look smaller than the other. However, the center marker would be an average size of the two markers since it is equidistant from both. Drawing the two lines and matching up the virtual marker's corners to be on the line proved to be more accurate because in our previous implementation, the zoom would change depending on the tilt of the board, giving bad data in terms of the distance from the virtual to the physical marker.

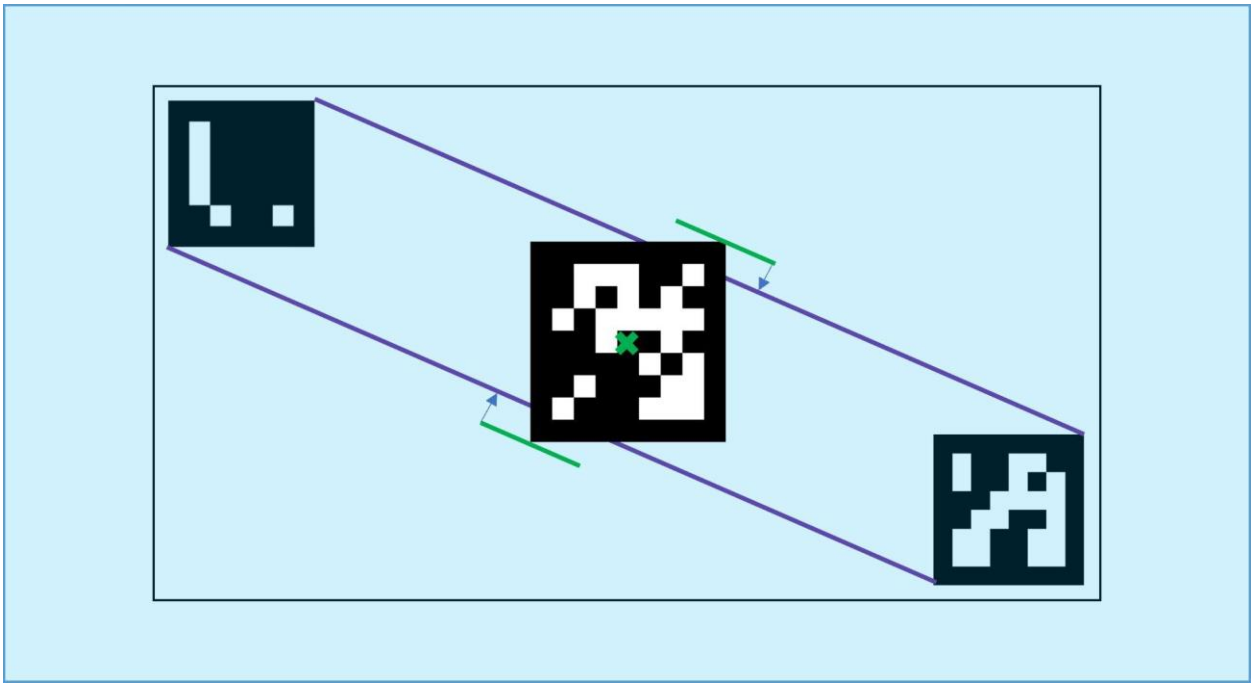

Figure 2.5: Updated Design to Handle Field of View / Scaling

6. Implementation

The calibration for the Projected Augmented Reality project began with a projector, a camera for calibration, a calibration board, and a Qualysis motion tracking setup. The projector was for projecting digital objects onto their physical counterparts. The camera was for input to the computer vision library used for calibration. The calibration board accompanied the camera and was marked with two computer vision ArUco markers. Finally, the Qualysis system tracked the position of both the projector and the calibration board. Using this setup, the goal was to adjust the settings of a digital camera in Unity such that the digital objects were properly projected on their physical counterparts.

From the camera used for calibration (the calibration camera), the video input was captured by an OpenCV function invoked by the Projected Augmented Reality Unity project. The video input contained images of the calibration board marked with ArUco markers which were then detected by the OpenCV functions.

In the Unity scene, the digital calibration board was located in the same relative position as the physical calibration board in the real space as a result of the Qualysis motion tracking system. Therefore, all that was needed was ensuring that the settings of the Unity camera, which was mapped to the position of the projector, were adjusted such that the digital and physical objects were properly mapped.

With the calibration camera, we detected both the ArUco markers on the physical calibration board and the ArUco markers from the projection of the digital calibration board. Using the differences in location and scale, between the physical and digital markers as detected by OpenCV, Unity panned and zoomed the digital camera to conduct the calibration. Once the digital and physical board were matched up within a threshold, the calibration process was finished.

7. Testing

To test our calibration setup we follow these steps:

1. Place the physical calibration board near the middle of the Perform Studio space.
2. Place the projector facing the calibration board far enough towards the center of the room so the Qualysis cameras can detect the projector.
    a. Make sure to turn the projector on, and that it is plugged into the PC.
3. Place the Azure camera anywhere facing the side of the calibration board with the ArUco markers.
    a. Make sure the camera is plugged into the PC and no other application is using it.
4. Start the Qualysis software and start a new measurement.
5. Start Unity and place the game window in the projector's view.
6. Press the play button in Unity.
7. Connect the Qualysis motion tracking to the Unity objects.

At this point the projected image should begin moving and scaling for calibration.

8. Developers Manual

Overview

In this guide you will learn how the current calibration system is working and how to build onto the existing system. The document includes information on what you need to get started, what software and hardware you will require, and information on the project files, regarding which file controls which part of the whole system. This will help you understand the current system. Along with that, in this guide you will find information about how to run the project.

To build onto this project you will find information in this document about how to test your updates as well as get information about common problems. That will help you troubleshoot problems that may occur with the whole system.

Prerequisites

To develop on the Projected AR project, you will first need to prepare the development environment. This requires the following to be set up:

Unity (Version 2019.4.23f1)

For this project you will be downloading Unity Version 2019.4.23f1. This is the core of the project and where the main scene will be running.

You will first download and install Unity Hub:

1. Go to the Unity download page [13] and click “Download Unity Hub”, as shown in Figure 3.1.

Figure 3.1: Downloading Unity Hub

2. Follow the steps in the executable that was downloaded to install Unity Hub.

Reference Unity Documentation for further help [14].

After successfully downloading Unity Hub, you will now download and install Unity Version 2019.4.23f1, as shown in Figure 3.2:

1. Open Unity Hub.
2. On the left side bar click “Installs:

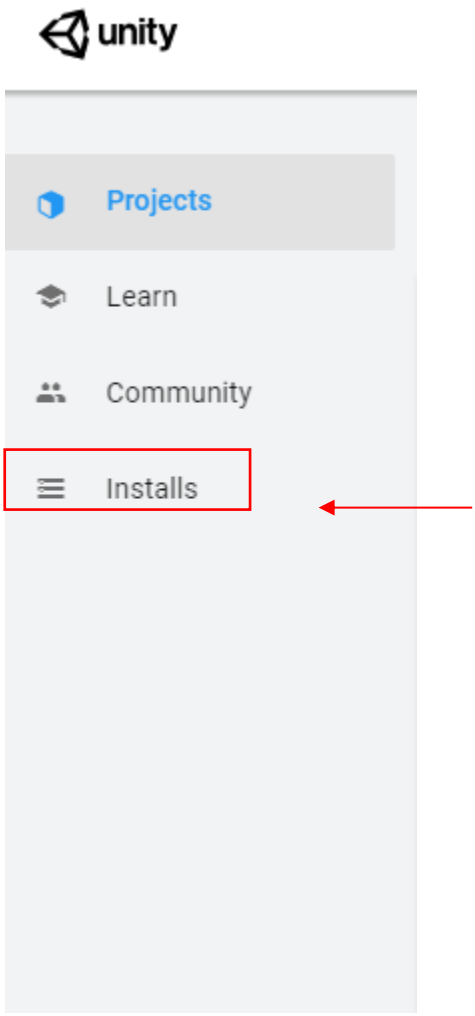

Figure 3.2: Installing Unity Version from Unity Hub

3. In the Installs tab, as shown in Figure 3.3, on the top right click “ADD”.

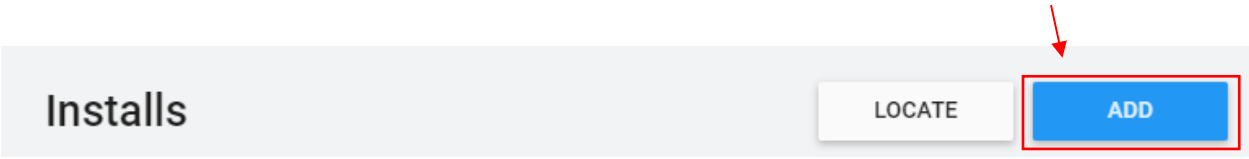

Figure 3.3: Adding new Unity Version in Unity Hub

4. A pop up will appear, as shown in Figure 3.4. Next, on the top of the pop up, click the text “download archive”.

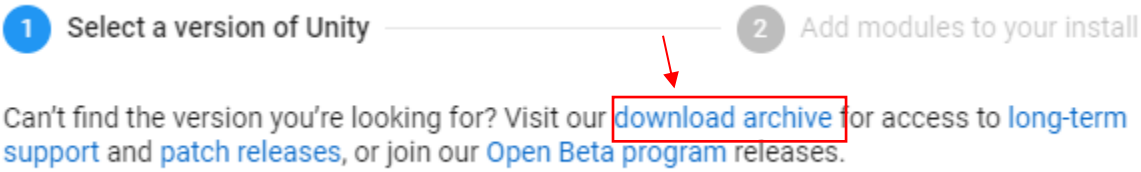

Figure 3.4: Getting Archived Version of Unity from Unity Hub

5. You will then be taken to Unity’s website, as shown in Figure 3.5, and locate version 2019.4.23f1 [15].

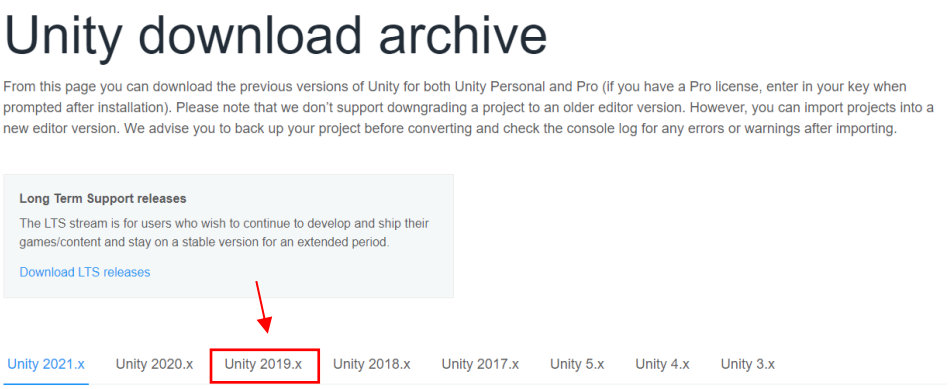

Figure 3.5: Navigating to get 2019 Version of Unity

6. Once found, as shown in Figure 3.6, click the button “Unity Hub”, as shown in Figure 3.7.

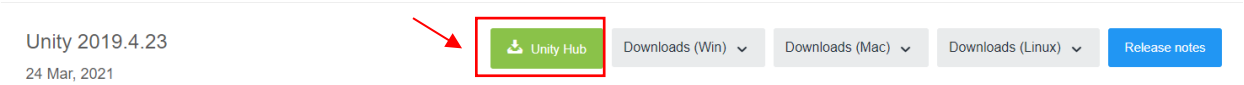

Figure 3.6: Downloading Unity Version 2019.4.23 From the Unity Website

If prompted to open in Unity Hub, as shown in Figure 3.7, click “Open Unity Hub”:

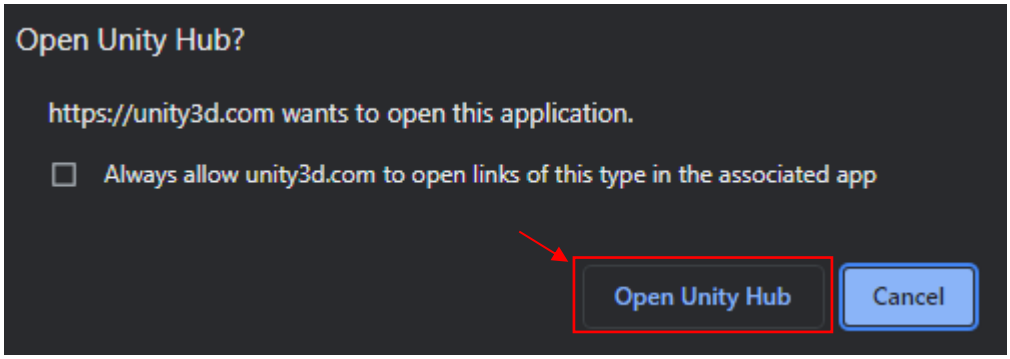

Figure 3.7: Allowing Browser to Open Unity Hub

7. Follow the steps in Unity Hub to install Unity version 2019.4.23f1.

For additional information about installing Unity or additional modules, refer to the Unity documentation [14].

Qualisys Motion Tracking System/Qualisys Track Manager

The Qualisys motion tracking system is an integral part of the current design of the project that allows the objects that are modeled in Unity to be tracked in real time around the room that contains the motion tracking equipment. With this system, we get the exact position in 3D space along with the exact rotation of the object that is visible through Unity. The equipment has been set up and integrated in the Perform Studio at the Moss Art Center. For additional information about the system visit [16] or email David Franusich about further questions at dfranusi@vt.edu.

Visual Studio

For this project, Visual Studio Community Edition will suffice, however from Virginia Tech you are also able to get Visual Studio Enterprise Edition. The following steps will be for installing Visual Studio Community Edition.

1. Visit Visual Studio Code Download link [17].
2. Under Downloads, find Community and click “Free download”, as shown in Figure 4.1.

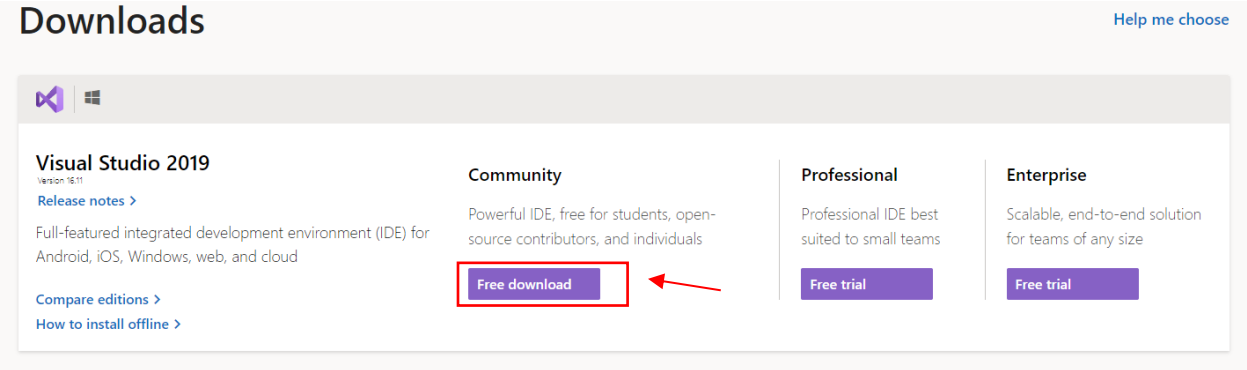

Figure 4.1: Downloading Visual Studio Installer

3. The Visual Studio Community downloader executable should now be downloading, as shown in Figure 4.2.

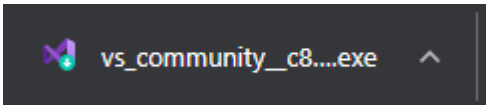

Figure 4.2: Installing Visual Studio Installer

4. Click on the executable and follow instructions to set up the software.
5. Once the setup process has begun, you will eventually be asked to select a workload. Be sure to check the “Desktop development with C++” workload, as shown in Figure 4.3.

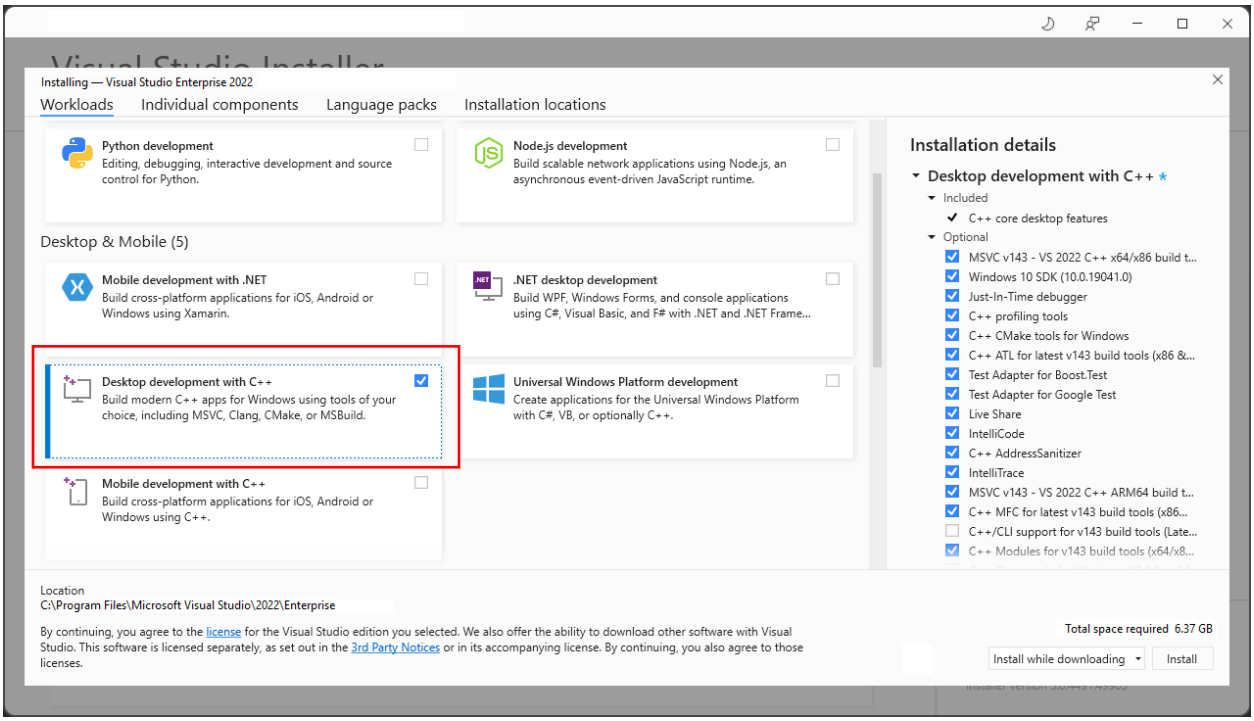

Figure 4.3: Workload Required for Visual Studio

For additional information about installing Visual Studio, visit the download link [17].

Azure Kinect SDK

The Azure Kinect SDK and Azure Kinect Viewer is being used to control the Azure camera settings, such as exposure, which may need to be altered based on lighting conditions in the room. However, if another camera is being used that is not the Azure camera, this part may not be necessary.

To download the Azure Kinect SDK, visit the Kinect SDK official documentation [7] for directions to install the SDK and the Azure Kinect Viewer.

Once downloaded, the Azure Kinect Viewer will allow you to change the camera settings. It will only be required to do so if there are issues with lighting that are causing the Azure camera to not view the ArUco markers correctly.

The only settings we end up changing is the exposure of the camera, which can be done by the following steps.

1. Select the device in the drop down and click “Open Device”, as shown in Figure 5.1.

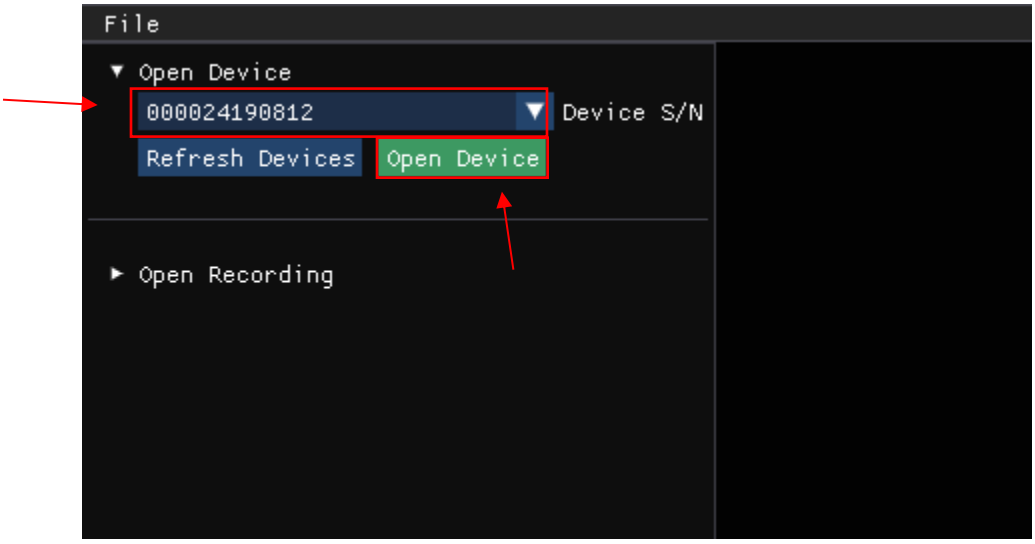

Figure 5.1: Opening Azure Camera in Azure Kinect Viewer

2. Click “Color Controls”, as shown in Figure 5.2.

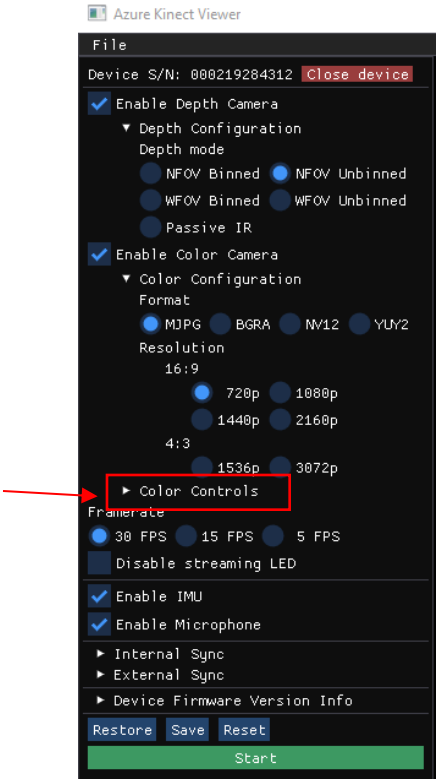

Figure 5.2: Opening Azure Camera Controls

3. Click the “A” icon next to Exposure Time to change controls to manual (“M”), as shown in Figure 5.3.

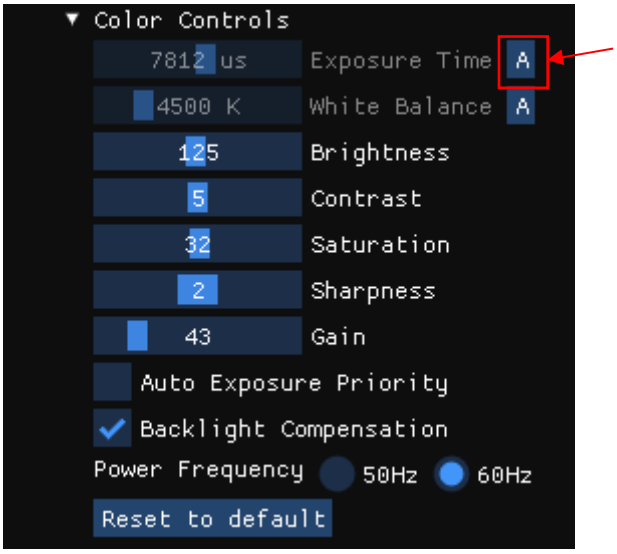

Figure 5.3: Updating Azure Camera Exposure

4. You will now be able to edit the camera exposure and see what the camera is seeing in the view, as shown in Figure 5.4.

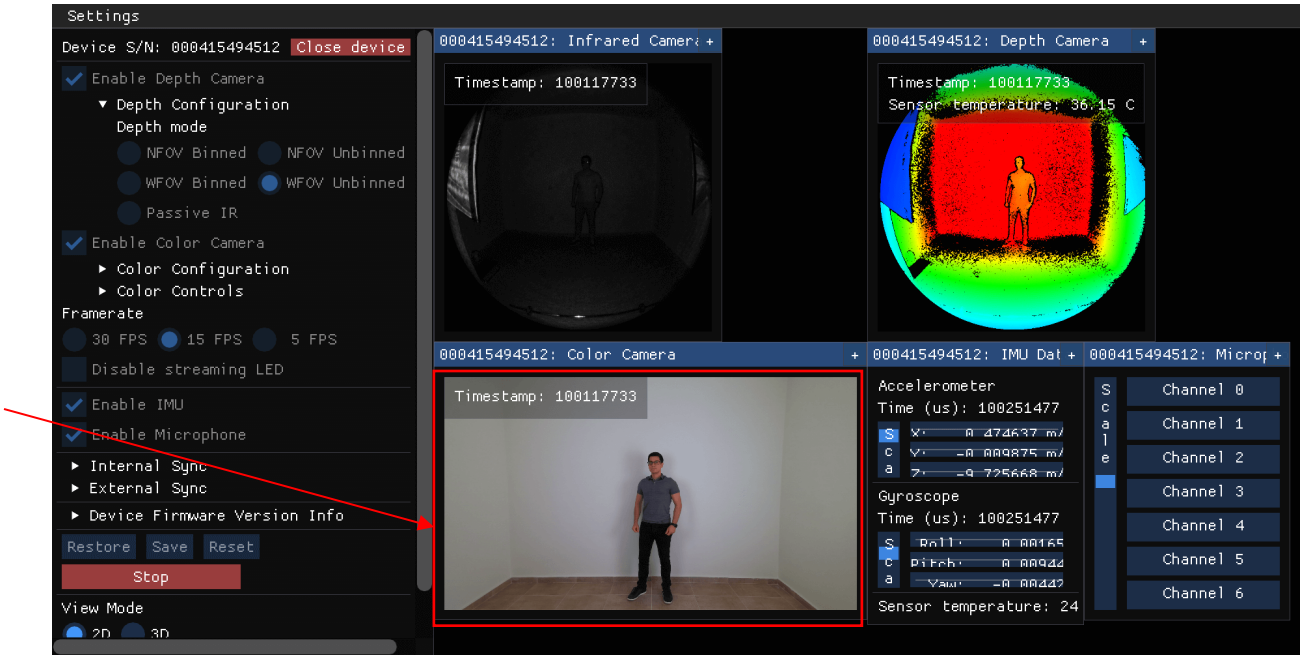

Figure 5.4: Viewing Azure Camera Exposure

> *Note: The view of what the camera is seeing (as shown in Figure 5.4) is slightly different from what the camera is seeing in Unity. Make sure to test what the camera is seeing in Unity and if it can detect ArUco markers there.*

For additional information about installing the Azure Kinect SDK, visit the official documentation [7].

Calibration Camera Plugin

Synchronizing the Unity camera and the projector is done using the calibration camera Unity plugin. This plugin is built from the OpenCV C++ methods which are exposed to Unity through compiling the C++ scripts into Dynamic Link Libraries (DLLs).

Linking Calibration Camera DLL and Unity

The calibration camera DLL and its dependency DLLs are in the Plugins folder in Unity, as shown in Figure 6.1. Besides placing the DLLs into the Plugins folder, no other actions are required to link the DLLs and Unity. To make changes to the DLLs, however, Unity will need to be restarted since Unity loads plugins from a script only once.

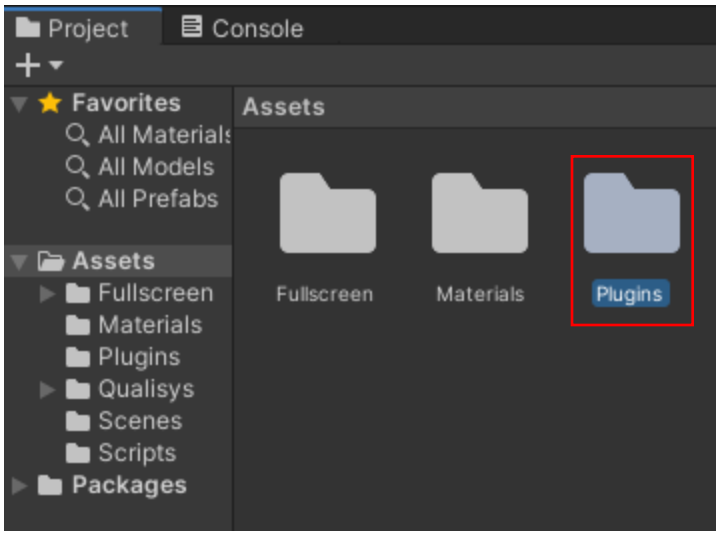

Figure 6.1: Unity Plugins Folder

Exposing C++ Functions to Unity C# Scripts

From a C# script, the desired C++ functions from the DLLs will need to be exposed. Figure 6.2 gives an example from the current Unity scripts, where *CalibrationCamera* is the name of the DLL file.

```
// Define the functions which can be called from the .dll.
internal static class OpenCVInterop
{
    [DllImport("CalibrationCamera")]
    internal static extern int Init(ref int capturePort);

    [DllImport("CalibrationCamera")]
    internal static extern int Close();

    [DllImport("CalibrationCamera")]
    internal static extern void Detect();

    [DllImport("CalibrationCamera")]
    internal static extern double xOffset();

    [DllImport("CalibrationCamera")]
    internal static extern double yOffset();

    [DllImport("CalibrationCamera")]
    internal static extern int GetSeenId();

    [DllImport("CalibrationCamera")]
    internal static extern double ScaleDifference();
}
```

Figure 6.2: Exposing DLL Functions to Unity C# Script

Building the Calibration Camera Dynamic Link Library

To build a DLL for the calibration camera, you will first need to have set up a Visual Studio project following the steps in the Visual Studio section. Having set up the Visual Studio project, follow these steps to compile the calibration camera C++ scripts.

1. In Visual Studio, set your build target to **Release x64**, as shown in Figure 6.3.

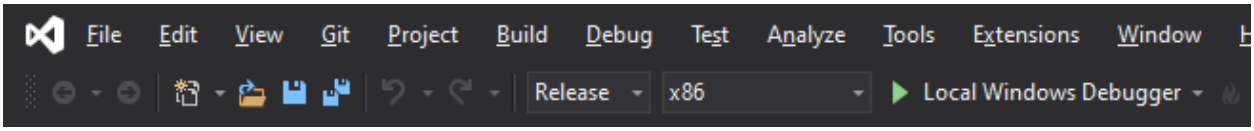

Figure 6.3: Visual Studio Release Build Target Settings

2. Click the **Build** option menu and select **Build Solution**, as shown in Figure 6.4.

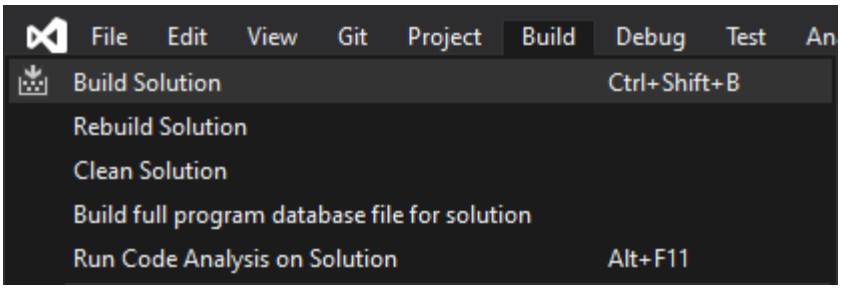

Figure 6.4: Building Visual Studio Solution

3. A DLL will be built and placed into the **x64/Release** folder at the top of the project file structure.
4. Copy this DLL to the **Plugins** folder in Unity along with any other DLL dependencies.

OpenCV

OpenCV is a robust computer vision library with many applications and modules. To identify ArUco markers, this project makes use of the OpenCV ArUco module. Currently the project makes use of the C++ OpenCV library in order to generate Dynamic Link Libraries (DLLs) that can be used as plugins in Unity. However, an implementation of the Python OpenCV library and Unity could be achievable.

To build OpenCV from source on a Windows machine please follow these steps.

Downloading the Source Code

1. Determine a file location for the **opencv** and **opencv_contrib** source code.
    a. For example: Documents/C++/Libs/
2. Download the **opencv** source code of the 4.5.3 release [18].
    a. Unzip/decompress or similar into the file location in **Step 1.**
3. Download the **opencv_contrib** source code of the 4.5.3 release [19].
    a. It is important that **opencv** and **opencv_contrib** are the same release version.
    b. Unzip into the file location in **Step 1.**

Configuring with CMake

1. Install CMake if you don't already have it [20].
2. Run CMake.
3. Click **Browse Source** and navigate to your **opencv** source code, as shown in Figure 7.1.

Figure 7.1: CMake Source Location

4. Create a **build** folder in the **opencv** folder.
    a. For example: Documents/C++/Libs/opencv/build
5. Click **Browse Build** and navigate to your **build** folder, as shown in Figure 7.2.

Figure 7.2: CMake Build Destination

6. Click **Configure** to configure.
    a. This may prompt you to specify a toolchain. If it does, just select your native toolchain.
7. Search for the parameter **OPENCV_EXTRA_MODULES_PATH** and set its value to the **modules** folder in your **opencv_contrib** source code, as shown in Figure 7.3.
    a. For example: C:/Users/iamza/Documents/C++/Libs/opencv_contrib/modules

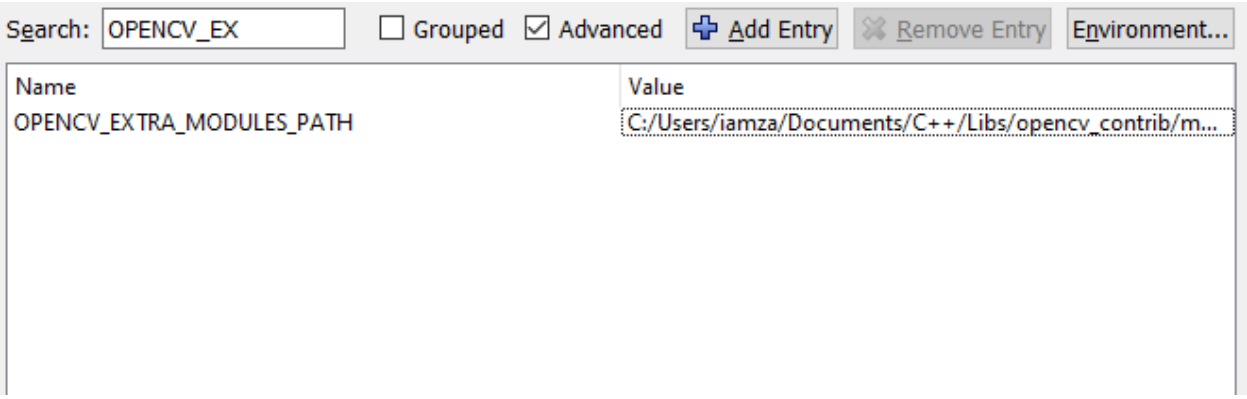

Figure 7.3: CMake Parameter Search

8. Click **Add Entry** and add the parameter **BUILD_opencv_aruco**, set the **Type** to **BOOL**, and make sure to tick the **Value** box of the parameter, as shown in Figure 7.4.

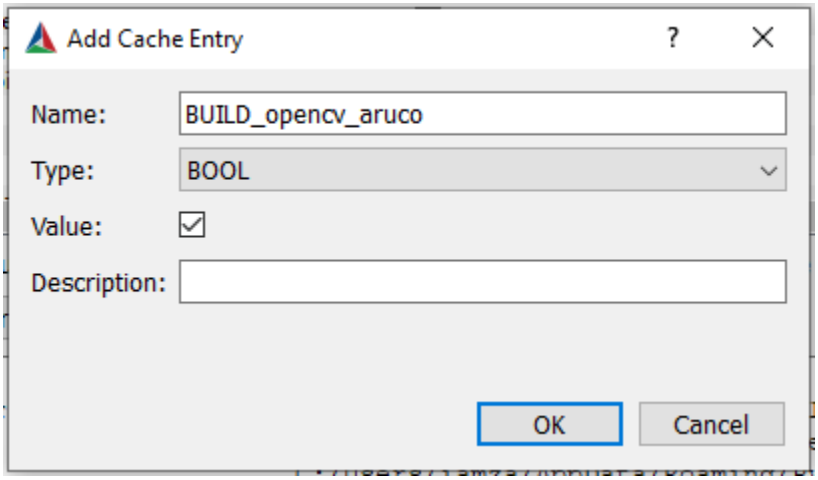

Figure 7.4: CMake New Entry for ArUco Module

a. You should be able to see **BUILD_opencv_aruco** in the parameter list, as shown in Figure 7.5.

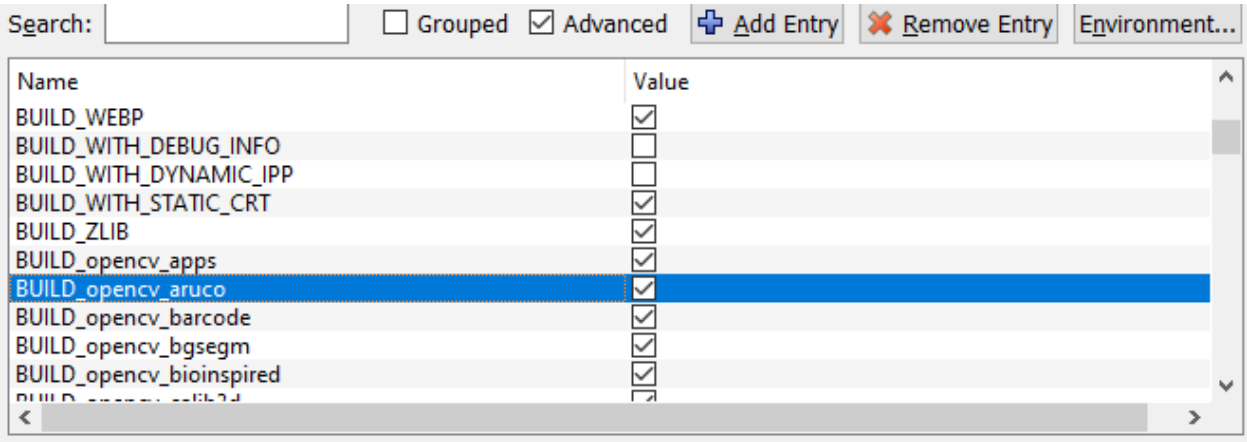

Figure 7.5: CMake Parameter List

9. Click **Generate**

a. You'll need to select a generator [21]. For Windows we suggest (and we used) the Visual Studio generator. If you are on MacOS you might be able to use Visual Studio too, but maybe you'll use something else. Our generator was specifically **Visual Studio 16 2019.**

Building OpenCV

Visual Studio Install

Install instructions if you built with Visual Studio:

1. Navigate to your **build** folder. You should see a solution file called **OpenCV.sln.**
2. Open **OpenCV.sln** with Visual Studio.
3. Build the project with **x64** for both **Debug** and **Release**, as shown in Figures 8.1 and 8.2.
    a. This may take upwards of an hour to finish.

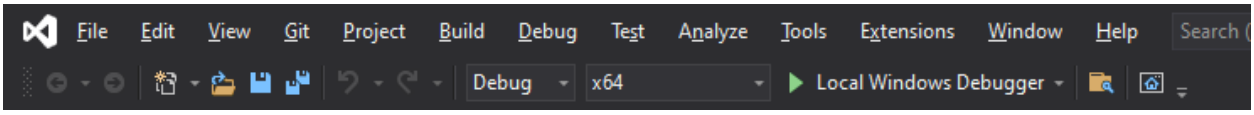

Figure 8.1: Visual Studio Set to Debug Target Build

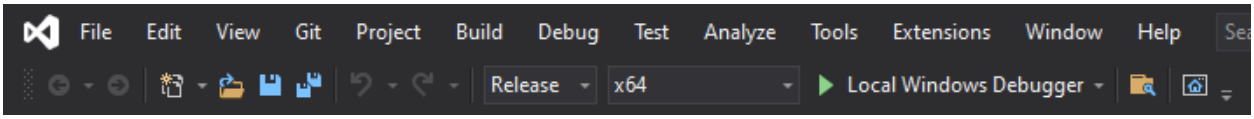

Figure 8.2: Visual Studio Set to Release Target Build

Linux Install

Install instructions if you are building with Linux (and likely Unix)

1. Navigate to your **build** folder and run the command:
    a. make -j(number of cores in your machine)
    b. example: make -j8
2. Run the command
    a. make install

Setting Up Path Variables

Note: These are all steps we know work and are needed for Windows, so if you are not on Windows your OpenCV might already work or you might require other steps.

1. Create a directory to serve as a permanent home for OpenCV.
    a. This step may be unnecessary since you might be able to just do the following steps with the **build** folder, but we'd suggest following them for simplicity.
    b. We'd also recommend putting them somewhere that won't have admin restrictions.

c. For example: C:\Users\iamza\OpenCV\OpenCV4.5.3\

2. Copy the **bin** folder, the **lib** folder, and the **include** folder from the **build** folder, as shown in Figure 9.1.

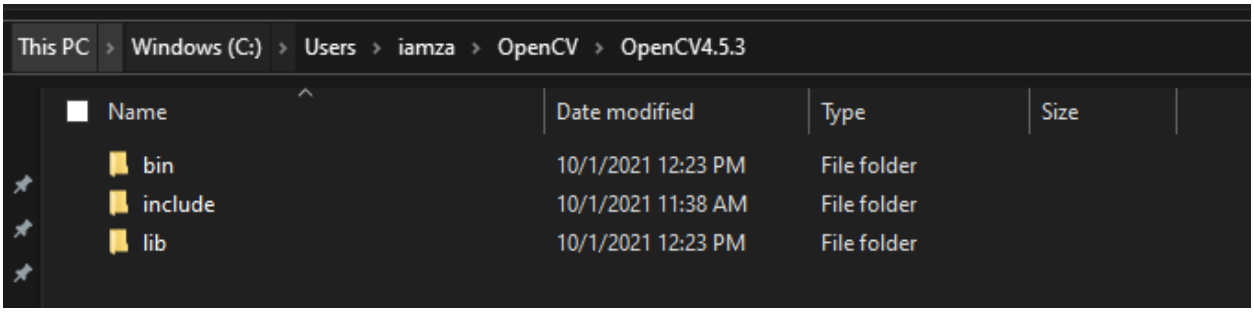

Figure 9.1: OpenCV File Location

3. Now we need to create an environment variable for our **OpenCV** folder, as shown in Figure 9.2.

a. Control Panel > System > Advanced system settings

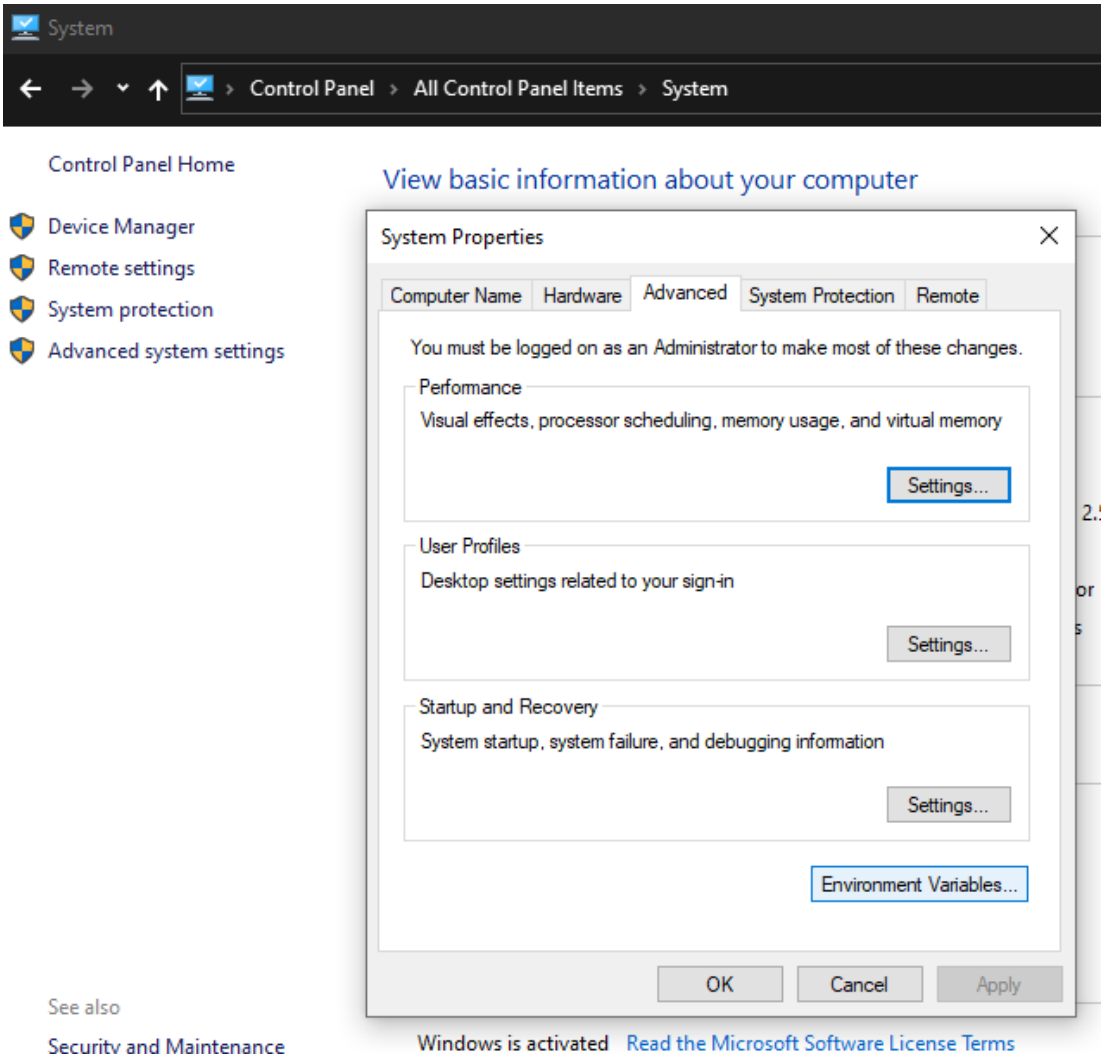

Figure 9.2: System Properties Window

4. Click **Environment Variables.**

5. In the **System variables**, click **New** and add the variable **OPENCV_DIR** with the value of the path to your **OpenCV** folder, as shown in Figure 9.3.

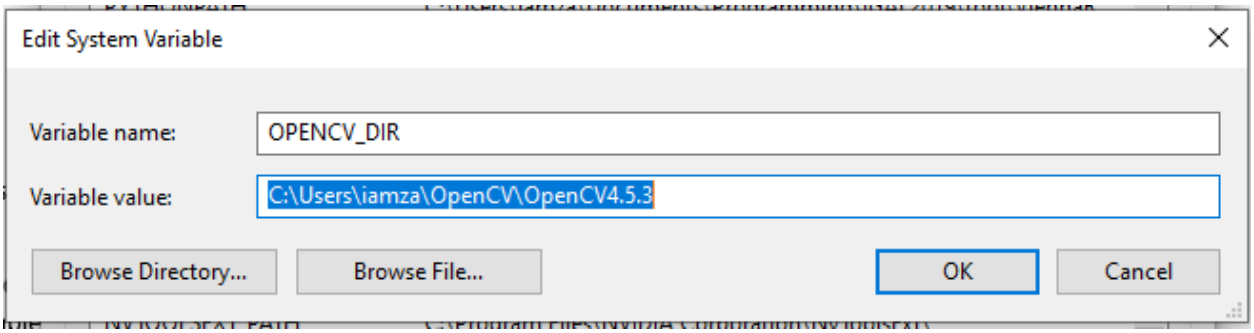

Figure 9.3: Creating New System Variable for OpenCV Directory

a. MAKE SURE **NOT** TO PUT A ';' AT THE END OF THIS

6. In the **User variables**, add **%OPENCV_DIR%\bin** to the end of your **PATH** variable.
    a. This one can have a ';'

Setting Up a Visual Studio Project for OpenCV

All Configuration Settings

1. Create a new empty C++ project in Visual Studio.
2. Create a new .cpp script called whatever you want.
    a. Ours is called Source.cpp; the name prefix doesn't matter.
3. Go to **Project** > **[ProjectName] Properties** to open the **Property Pages.**
4. Set the **Configuration** to **All Configurations** and the **Platform** to **x64**, as shown in Figure 10.1.

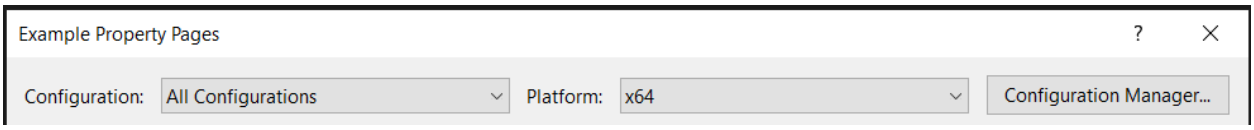

Figure 10.1: Setting Configuration in Visual Studio Property Pages

5. Under **C/C++** > **General**, add **$(OPENCV_DIR)\include** to the **Additional Include Directories**, as shown in Figure 10.2.

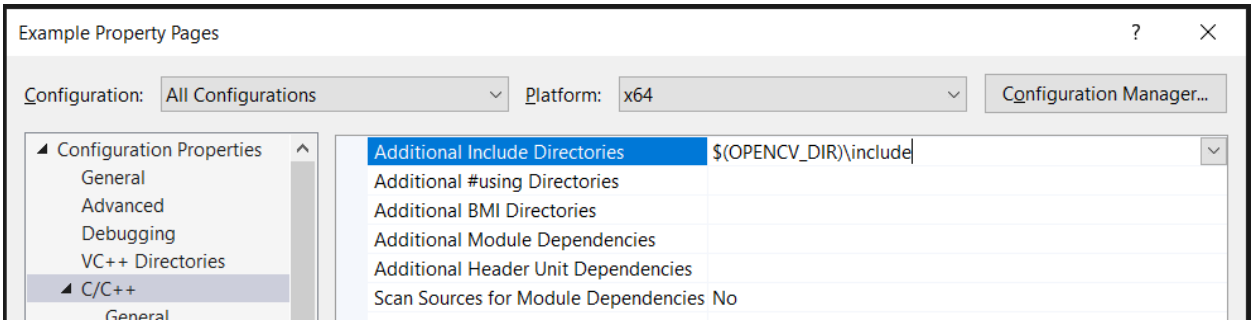

Figure 10.2: Adding Include Directories to C++ Properties

Debug Settings

We have set it up so that targeting a debug build will make an executable that will run locally on your machine. For a DLL to export to Unity, check out the **Release Settings**.

1. Set the **Configuration** to **Debug**, as shown in Figure 10.3.

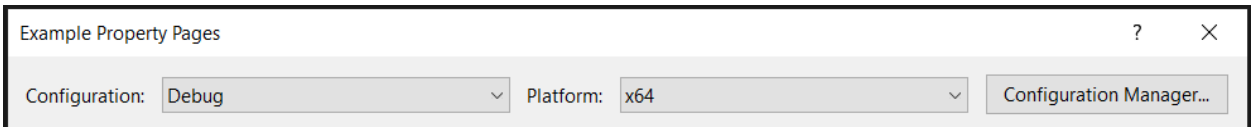

Figure 10.3: Setting Configuration to Debug in Visual Studio Property Pages

2. Under **Configuration Properties** > **General** change **Configuration Type** to **Application (.exe)**, as shown in Figure 10.4.

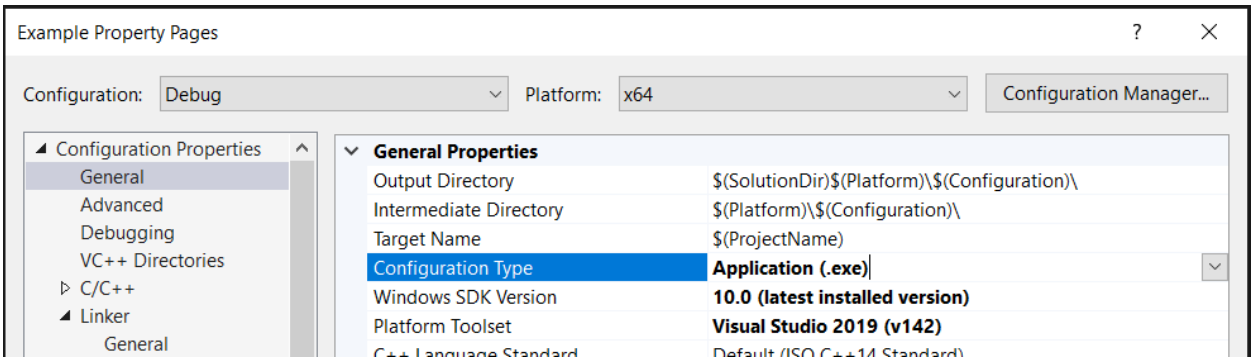

Figure 10.4: Setting Debug Configuration Type to Application

3. Under **Configuration Properties** > **Advanced** change **Target File Extension** to **.exe**, as shown in Figure 10.5.

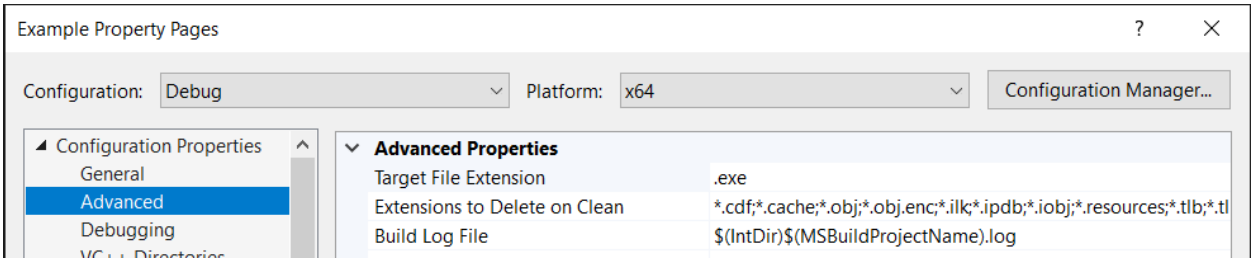

Figure 10.5: Setting Debug Target File Extension to .exe

4. Under **Linker** > **General**, add **$(OPENCV_DIR)\lib\Debug** to the **Additional Library Directories**, as shown in Figure 10.6.

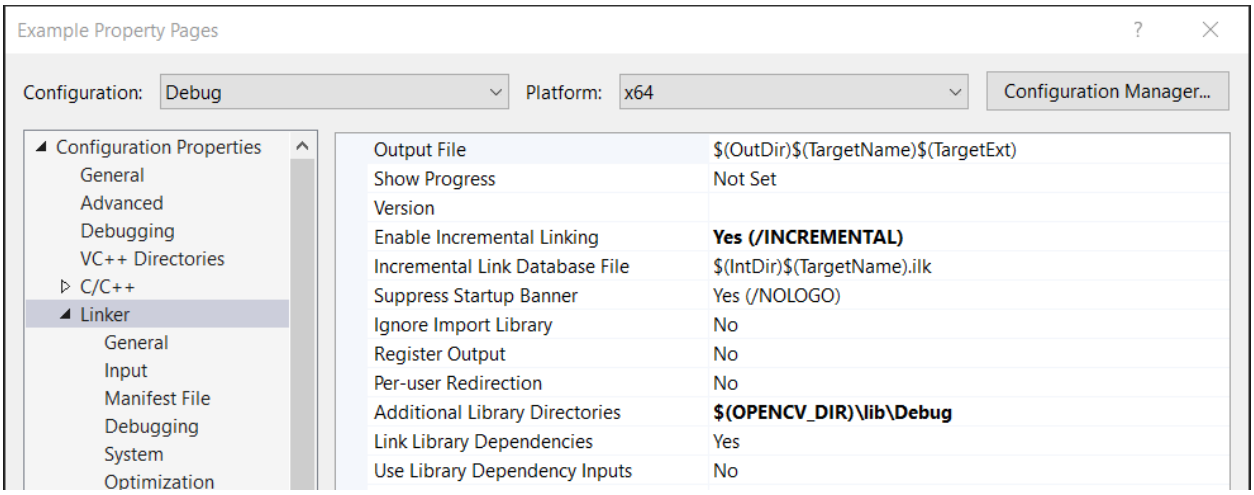

Figure 10.6: Setting Debug Additional Library Directories

5. Under **Linker** > **Input**, add the following **Dependencies**

    a. opencv_aruco453d.lib
    opencv_core453d.lib
    opencv_highgui453d.lib
    opencv_videoio453d.lib

    b. Make sure the *453d.lib matches the version of OpenCV you installed; you can check this by going to your **$(OPENCV_DIR)\lib\Debug** folder, as shown in Figures 10.7 and 10.8.

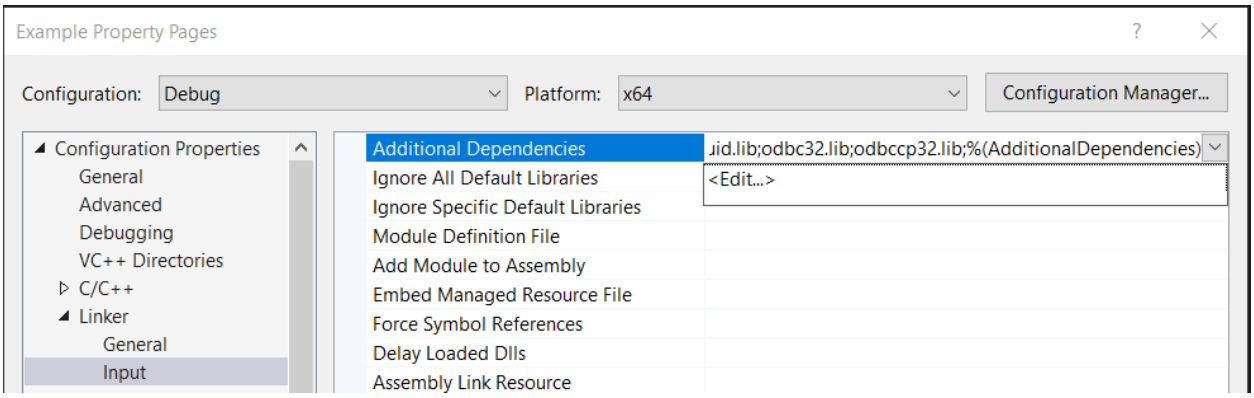

Figure 10.7: Setting Debug Additional Dependencies

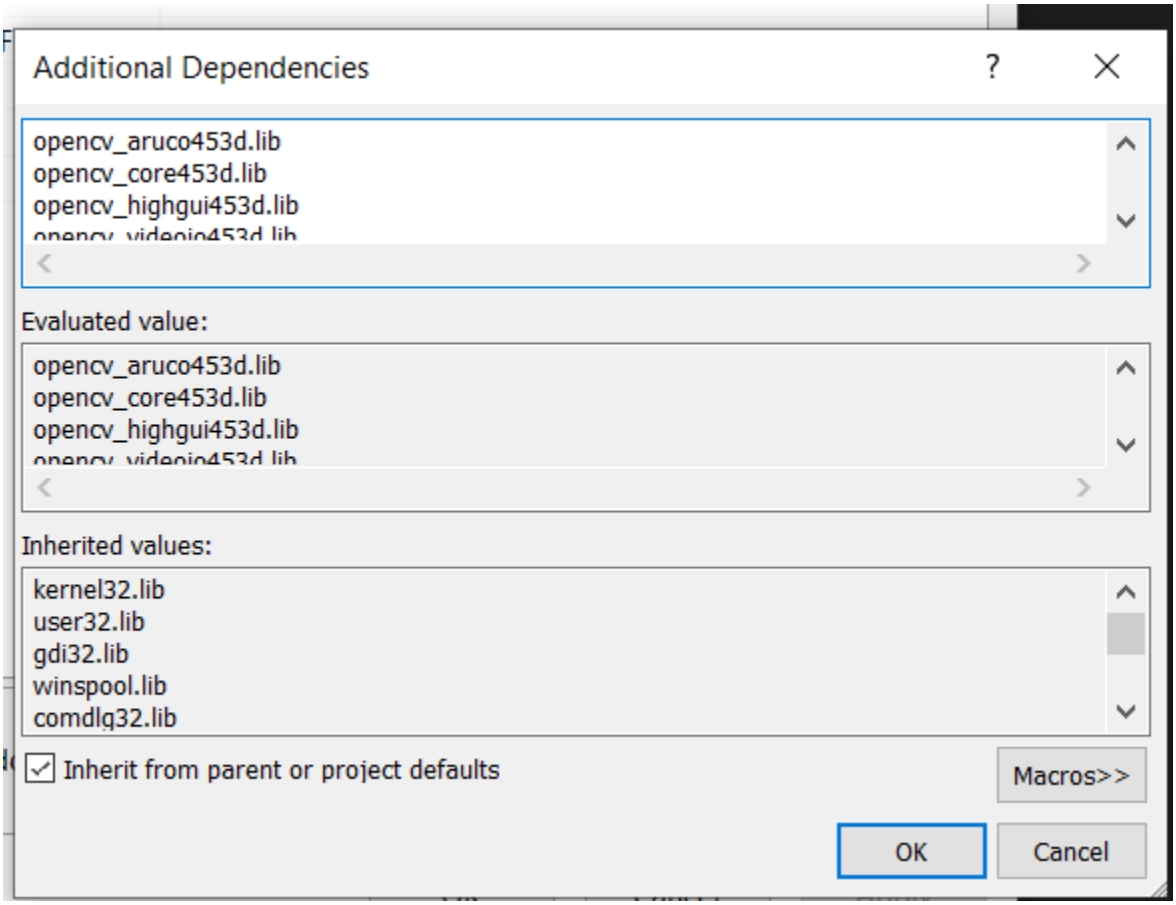

Figure 10.8: Additional Dependencies Window with Debug Dependencies

6. Click **Apply** at the bottom of the **Property Pages** window.

Release Settings

Targeting a release build will produce a DLL that can be exported and used elsewhere. You will need to include all other DLLs for the dependencies of your DLL.

1. Set the **Configuration** to **Release**, as shown in Figure 10.9.

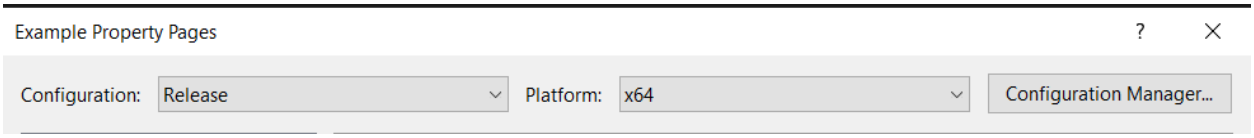

Figure 10.9: Setting Configuration to Release in Visual Studio Property Pages

2. Under **Configuration Properties** > **General** change **Configuration Type** to **Dynamic Library (.dll)**, as shown in Figure 10.10.

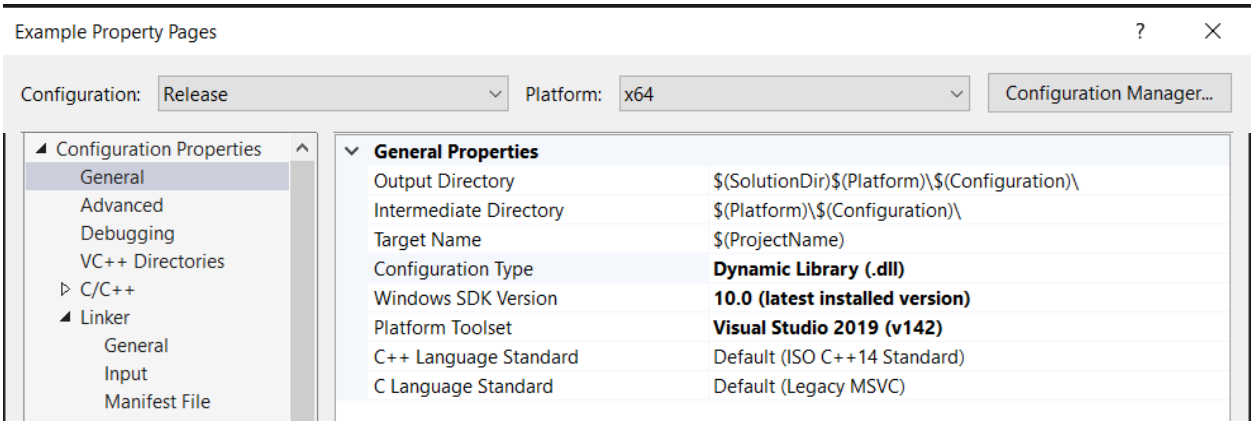

Figure 10.10: Setting Release Configuration Type to Dynamic Library

3. Under **Configuration Properties** > **Advanced** change **Target File Extension** to **.dll**, as shown in Figure 10.11.

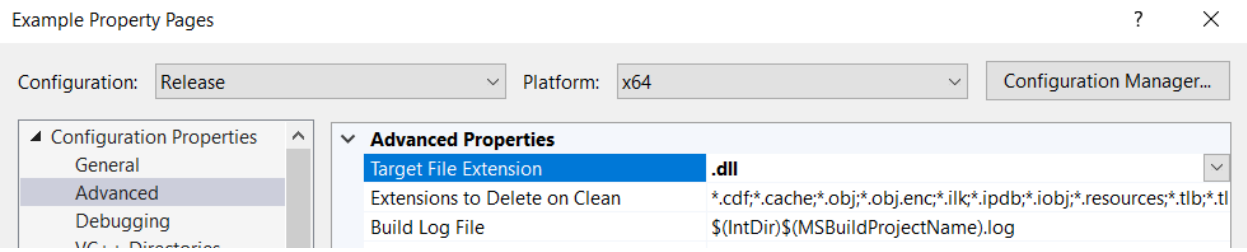

Figure 10.11: Setting Release Target File Extension to .dll

4. Under **Linker** > **General**, add **$(OPENCV_DIR)\lib\Release** to the **Additional Library Directories**, as shown in Figure 10.12.

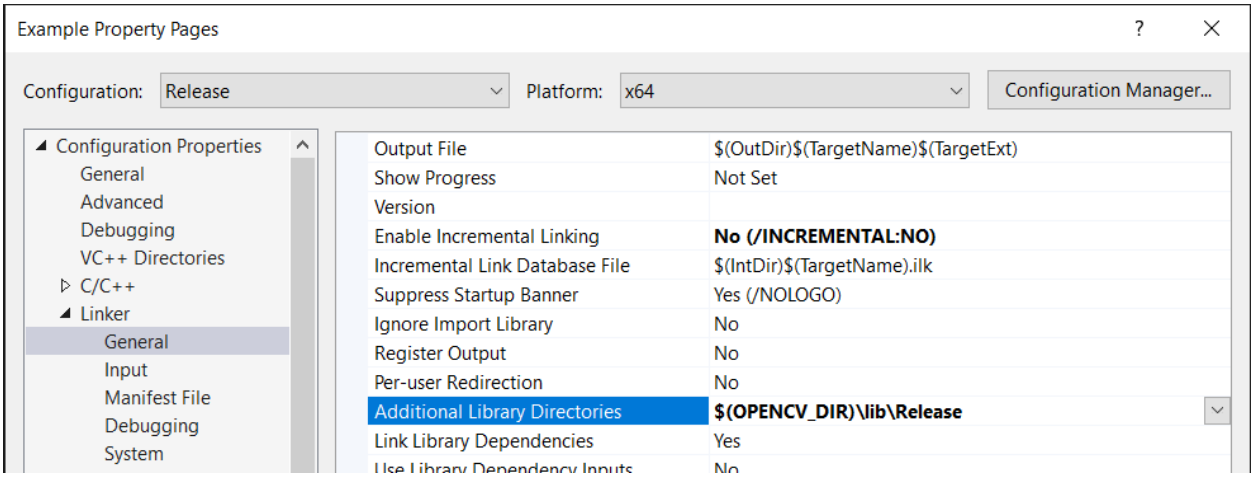

Figure 10.12: Setting Release Additional Library Directories

5. Under **Linker** > **Input**, add the following **Dependencies**, as shown in Figure 10.13.

    a. opencv_aruco453.lib
    opencv_core453.lib
    opencv_highgui453.lib
    Opencv_videoio453.lib

    b. Make sure the *453.lib matches the version of OpenCV you installed, you can check this by going to your **$(OPENCV_DIR)\lib\Release** folder, as shown in Figure 10.14.

    c. Also notice that the difference between the Debug and Release library files is the *d.lib

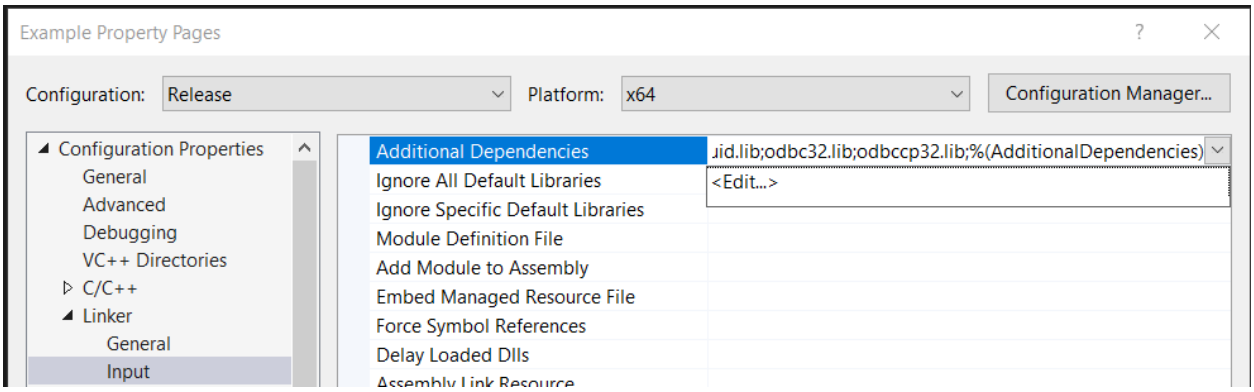

Figure 10.13: Setting Release Additional Dependencies

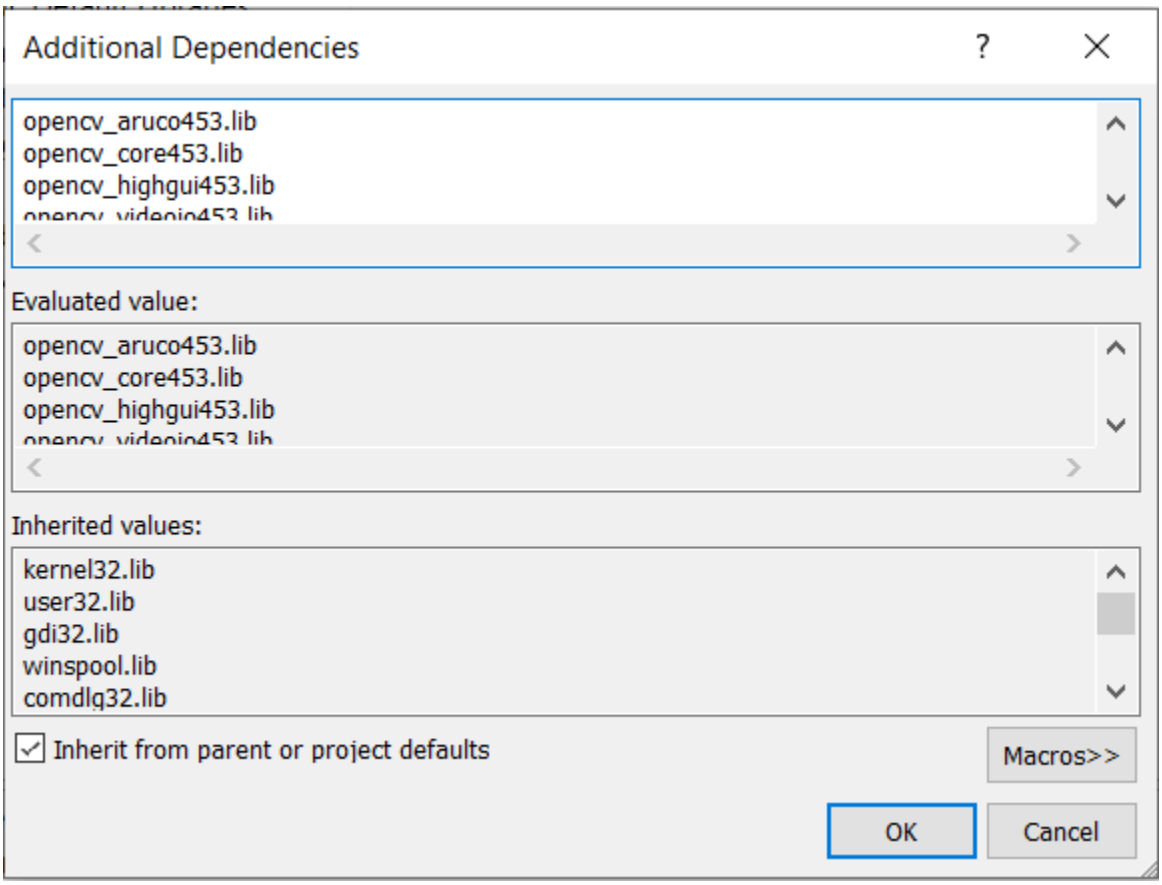

Figure 10.14: Additional Dependencies Window with Release Dependencies

6. Click **Apply** at the bottom of the **Property Pages** window.

Project files

The project files are contained in 3 separate parts [11]. There is a Unity project that contains the scene as well as the C# scripts, there is a Visual Studio project that contains the OpenCV scripts in C++, and there is the Qualisys project that contains the setup for the Qualisys system.

Unity Project

Our Unity project has one scene. This scene contains a virtual camera as well as a calibration board model. Clicking the “connect” button after the project has been started will link the positions of the objects to the data from the Qualisys system. This section goes into detail on how you can modify the project for different setup needs.

Folder Structure

The Unity project is a collection of files primarily managed by Unity. The Assets folder, however, is the location for the various plugins, scripts, and models created by the user. Table 1 outlines all the folders and files contained within the Assets folder.

```
ProjectedARSystem.zip/ProjectedAR-CS4624-main/ProjectedAR-Unity
|
+- Assets
   |
   +- FullScreen (FullScreen plugin)
   +- Materials
      |
      +- Projected AR Calibration board
         (Material for digital calibration board)
      +- Shotgun Markers (Material for scatter board)
   |
   +- Plugins
      |
      +- CalibrationCamera.dll (OpenCV C++ functions DLL)
      +- … (Other important DLLs)
   |
   +- Qualysis (Qualysis plugin)
   +- Scenes
      |
      +- Calibration.unity (Calibration scene)
   |
   +- Scripts
      |
      +- Calibrator.cs (Script for calibrating the Unity camera)
      +- CameraSetup.cs
         (Script for panning and zooming the Unity camera)
   |
   +- shotgun board.skp (scatter board model)
   +- texture board.skp (digital calibration board)
|
+- Library
+- Logs
+- Packages
+- ProjectSettings
```

Table 1: Unity Project File Structure

Scripts

There are two scripts in the Unity project. The script called Calibrator.cs handles the calibration process in Unity. In Calibrator.cs, the Update function pans and zooms the camera based on position and scale offsets of the ArUco markers retrieved from the OpenCV functions. The second script is called CameraSetup.cs. This script provides the functions and controls for rotating the camera and adjusting the camera's FOV. Together these scripts comprise the majority of the calibration process.

Creating New Assets (programs and file types)

Creating new assets for our project is quite simple. Unity takes in a variety of 3D files, and we have chosen to import using Sketchup (.skp) files. Unity uses the .fbx filetype and recommends using it if possible. However, Unity can also import automatically by converting the files from the third-party software listed in the Unity documentation [22]. Simple assets such as boxes and planes can also be made in Unity directly.

Uploading/Inserting New Assets

To import an asset, you can click on the asset tab on the top left and select “import new asset”, as shown in Figure 11.1. You can also drag and drop files directly into the Unity Project panel. We recommend creating materials in the Unity project though, and not adding it to the 3D model prior to import. We found that simple colors work well with Unity and can be used as placeholders for actual materials.

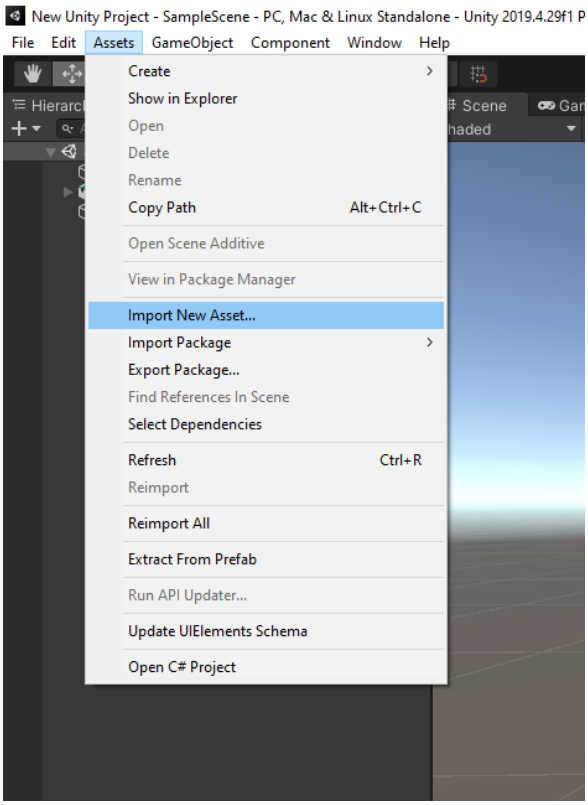

Figure 11.1: Importing new Assets

Handling Assets (adding the asset to the scene)

To add an asset to a scene, simply drag and drop it from the assets folder to the scene hierarchy panel. All position coordinates are relative to the parent, or the world if there is no parent component. For us, we have an object with the Qualisys tracking script. If we import a board and it ends up sideways, we cannot manually set the transform components (position, rotation, scale) since starting the script would begin changing it to what Qualisys sends. So we add the calibration board as a child component to the object that has tracking, and we can rotate or shift the board so that it lines up with where it is in real life. This discrepancy should technically not be there, but setting the wrong axis in Qualisys or the 3D modeling software can make the object orient to the wrong direction.

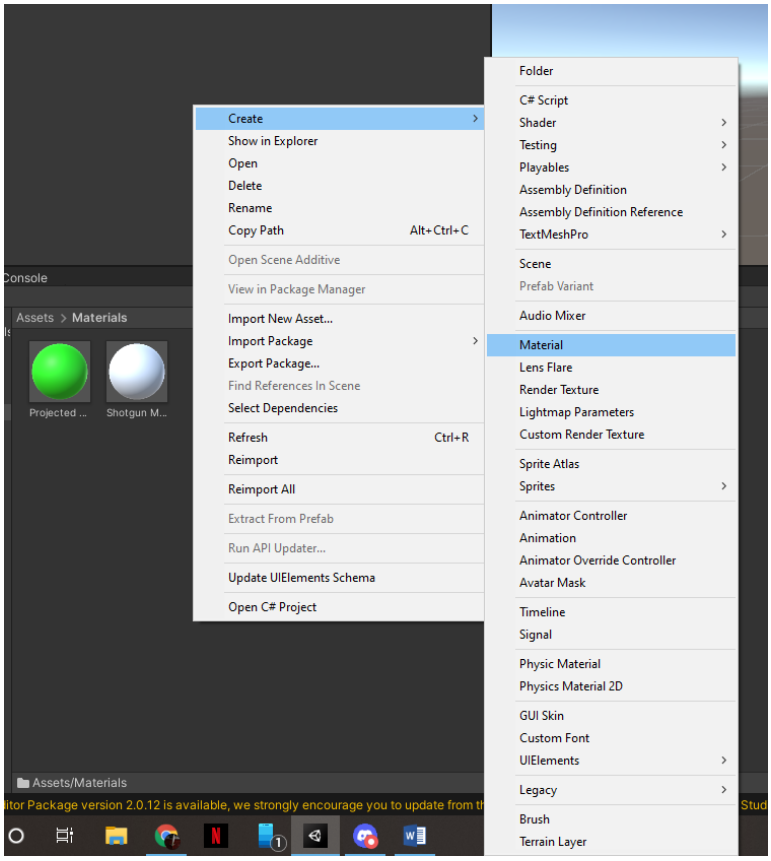

Figure 11.2: Creating a new Material

Materials

All of our materials are in the materials subfolder for organizing but this isn't necessary. After importing a 3D object, Unity will automatically try to import the simple materials too. If you try to apply a material to an object in Unity, it will cover it fully. But if you drag and drop a material onto a specific color on that object, only that area will be replaced. This is what we did for the calibration board, so that the image texture is only on one face. To create a material, left click in the project plane and select create new material, as shown in Figure 11.2.

We then set the albedo (which controls the base color map of a surface) of the material to our custom image and fiddled around with the tiling to match. We dragged and dropped the material on the part of the object in the scene that we wanted, as shown in Figure 11.3. Additional informative materials can be found on the Unity documentation.

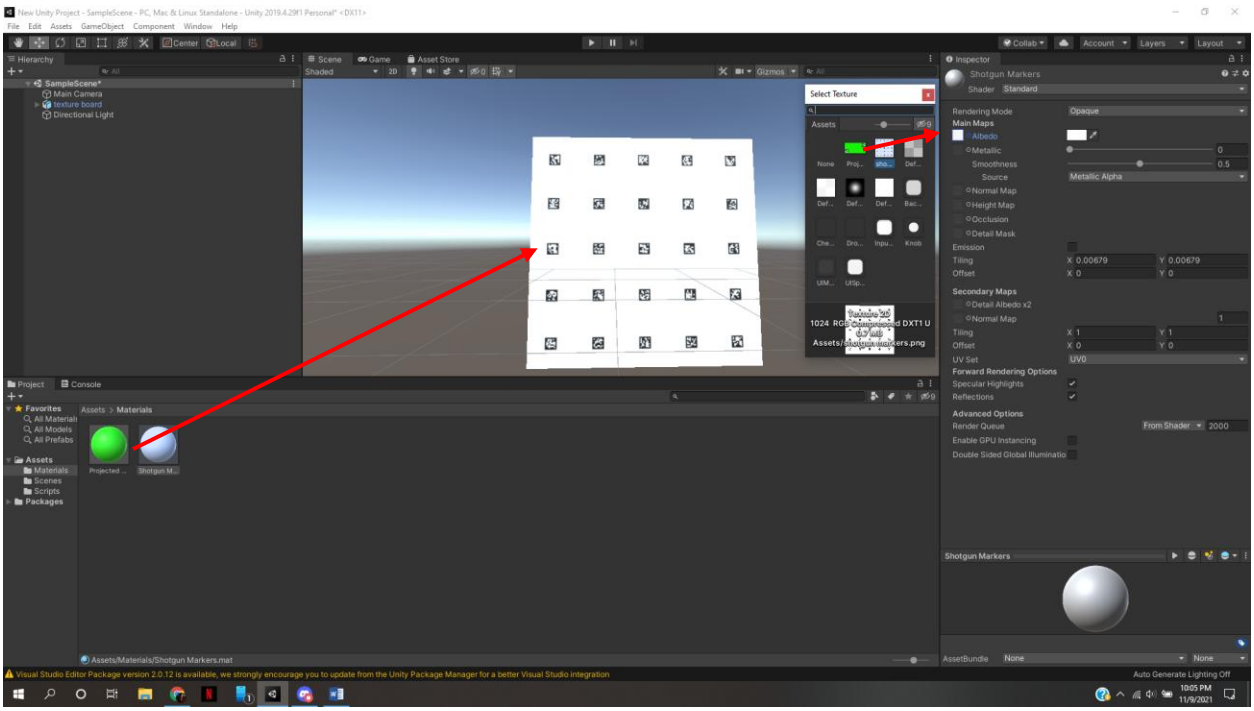

Figure 11.3: Setting Material Albedo and Dragging Material to Object Surface

Visual Studio Project

The Visual Studio Project contains the C++ script which is compiled into the DLL plugin for Unity. This file is called Source.cpp. In this file, there are several functions used for calibration and debugging.

Folder Structure

Within the GitHub project is a folder called CalibrationCamera [11]. This folder contains the Visual Studio project used to write and compile the OpenCV C++ code. The Source.cpp file is the file containing all the exposed C++ functions. The x64 folder contains the debug and release builds. The debug build is an executable which runs the main function located in Source.cpp. The release build is a Dynamic Link Library (DLL) which is used to expose the C++ functions in Unity. Table 2 outlines the files located in this folder.

```
ProjectedARSystem.zip/ProjectedAR-CS4624-main/CalibrationCamera
|
+- x64
   |
   +- Debug
      |
      +- CalibrationCamera.exe (Local debug OpenCV C++ executable)
   |
   +- Release
      |
      +- CalibrationCamera.dll (DLL of exposed OpenCV C++ functions)
|
+- CalibrationCamera.vcxproj (Visual Studio project file)
+- Source.cpp (C++ file with all OpenCV functions)
```

Table 2: Visual Studio Project File Structure

C++ Functions

The main function in Source.cpp is used for debugging. Building and running the Visual Studio project in Debug mode will allow you to run and test OpenCV locally. The main function and other C++ functions can be called and run this way.

The Source.cpp file also contains functions that are exposed from the DLL with the unique external method signature. Using this method signature, functions can be called from Unity. These functions conduct the necessary calculations to determine marker offset and scale differences. For further details on the C++ functions, see the documentation in Source.cpp.

Qualisys Project

The Qualisys project consists of a single file that contains the setup for motion tracking the objects. In the file you will be able to start the motion tracking system and "see" what the motion tracking cameras are seeing. Along with that you will be able to model new points to track as you add new models to the project. For additional information about the system visit the specified link [16] or email David Franusich about further questions at dfranusi@vt.edu.

The website linked above also has tutorials to follow that may be useful, as shown in Figure 12.

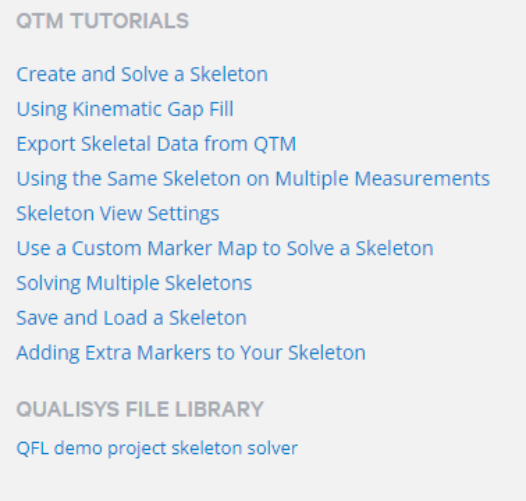

Figure 12: Qualisys Track Manager Tutorials

Running the Project

Running the project involves running the Qualisys project and running Unity.

1. Open Qualisys project
    a. Open Qualisys Track Manager.
    b. Open the project file for the motion tracking system.
    c. On the top left click the leftmost icon to start up the system, as shown in Figure 13.1.

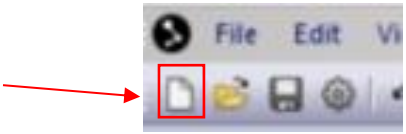

Figure 13.1: Starting Qualisys Motion Tracking System

You will now have the Qualisys system running and the objects set up in the project will now be tracking.

To display the 3D view of what the system is seeing, click “3D” on the bar as shown in Figure 13.2 (the image below says “2D” as the view is currently in 3D, however if you are in the 2D view, that option will say “3D” to go to the 3D view).

Figure 13.2: Setting Qualisys View to 3D

2. Open Unity Project

    a. Open Unity Hub.

    b. Open the Unity project file.

    c. Once loaded and with Qualisys Track Manager running in the background, press play on Unity, as shown in Figure 13.3.

Figure 13.3: Playing Unity Scene

    d. Click “Connect” to connect to the Qualisys Motion Capture system, as shown in Figure 13.4.

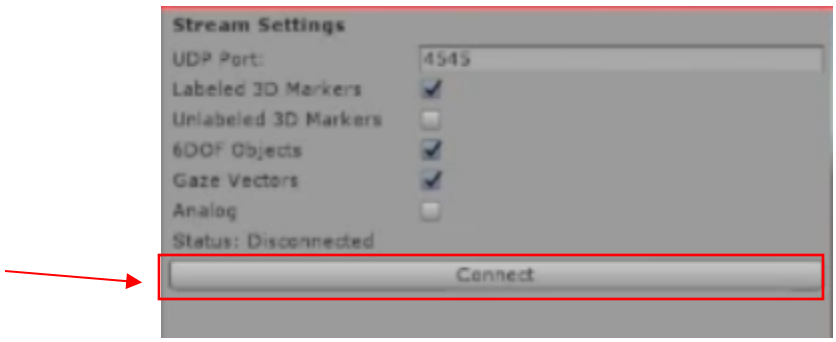

Figure 13.4: Connecting Unity to Qualisys System

You are now running the project and as you move the object that is being tracked in the physical world, you will see the same actions taking place in Unity.

9. User Manual

Qualisys Project

The Qualisys project is there to track the objects throughout the room to know the object's exact position and rotation. This will be used to calibrate the board to know where it is positioned with respect to the projector, as well as track the board to project images onto it.

Setup

1. Open Qualisys Track Manager
2. Open the project file for the motion tracking system
3. On the top left click the leftmost icon to start up the system

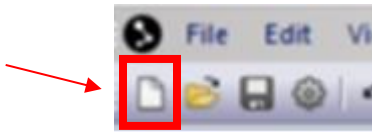

Figure 14.1: Starting Qualisys Motion Tracking System

You will now have the Qualisys system running, and the objects set up in the project will now be tracking.

Unity Project

The Unity project is the backbone of this system. It contains all the scripts and the scene with the objects that are required to calibrate the whole system. You will interact with this project to calibrate the system as well as update any textures you want projected.

Setup

1. Open the project
    a. Open Unity Hub
    b. Open the Unity project file titled ProjectedAR-Unity
2. Position the calibration board, camera, and projector
    a. Position the board in front of the lens of the projector as shown in Figure 14.2, where the board fits completely inside the board's projection. Along with that, position the camera so that the lens of the camera faces the board, and the camera can view the board in its entirety.

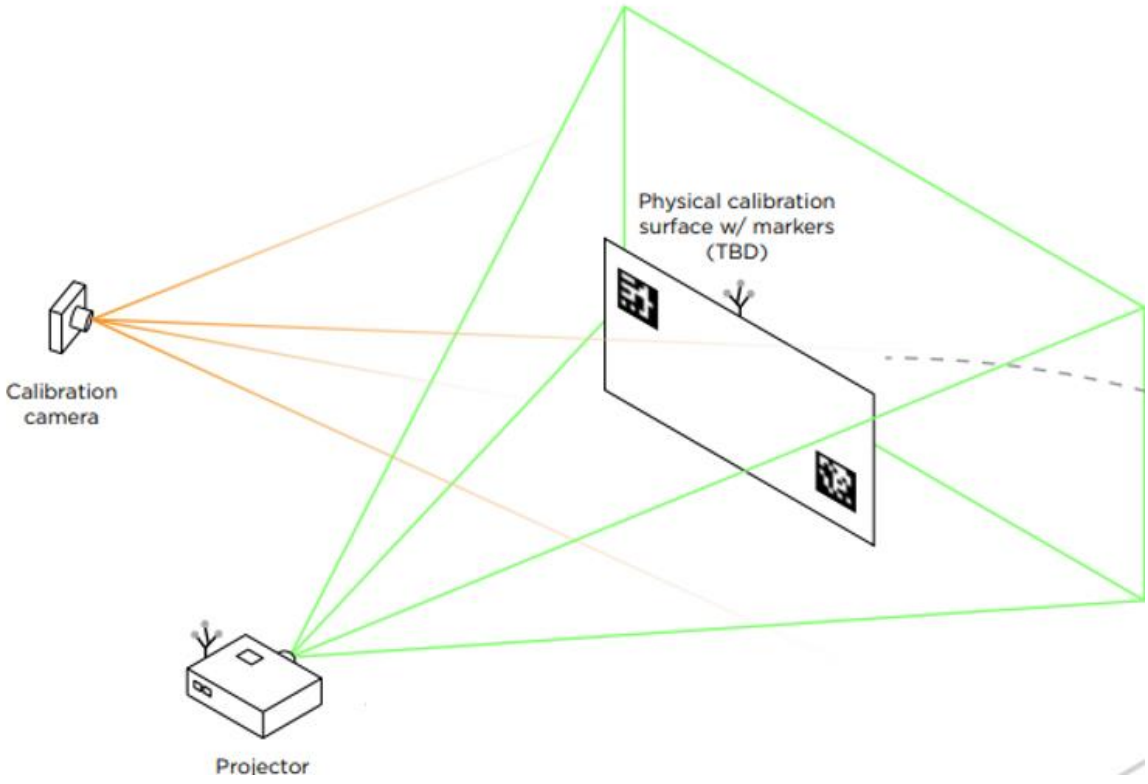

Figure 14.2: Calibration Board, Projector and Calibration Camera Setup

How to Run Calibration

1. Once loaded and with Qualisys Track Manager running in the background, press play on Unity, as seen in Figure 14.3.

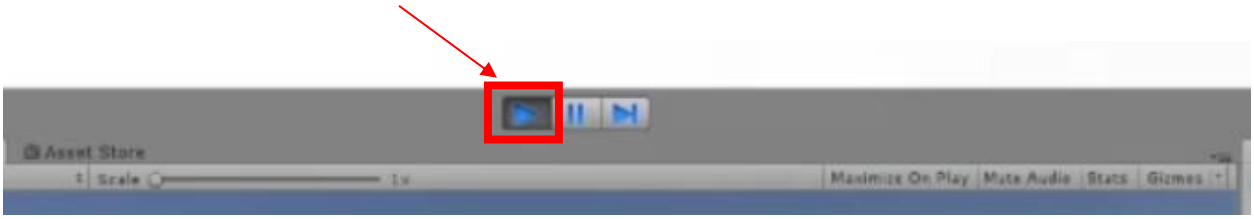

Figure 14.3: Playing Unity Scene

2. Click “Connect” to connect to the Qualisys Motion Capture system, as seen in Figure 14.4.

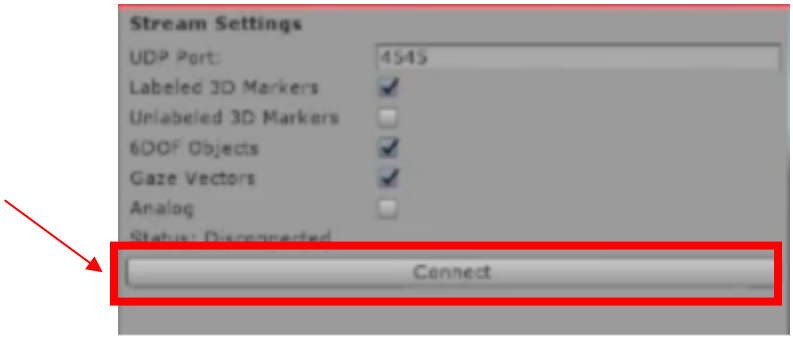

Figure 14.4: Connecting Unity to Qualisys System

The project is now running, it will automatically begin calibrating using the scatter board. Once the position and scale of the center marker meet the set thresholds, the calibration will finish with a message in the Console window, as seen in Figure 14.5. When the calibration is finished, you should be able to move objects around in the room and have them properly projected onto.

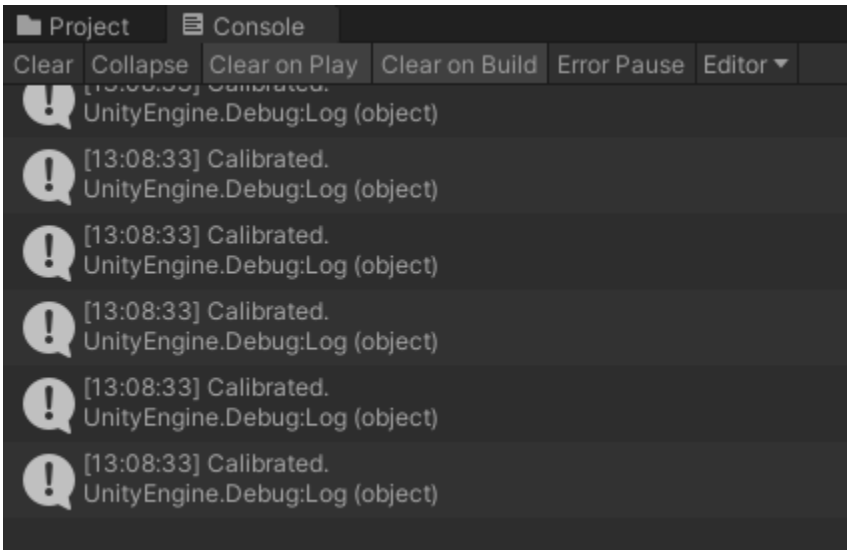

Figure 14.5: Calibrated Console Output

If the calibration process seems to be stuck in a loop, the threshold or sensitivity for panning and zooming can be adjusted to assist the program, as seen in Figure 14.6. Sensitivity changes the speed of panning or zooming, while the threshold is the upper limit for the absolute value of the zoom and panning difference from perfectly calibrated under which the code will stop.

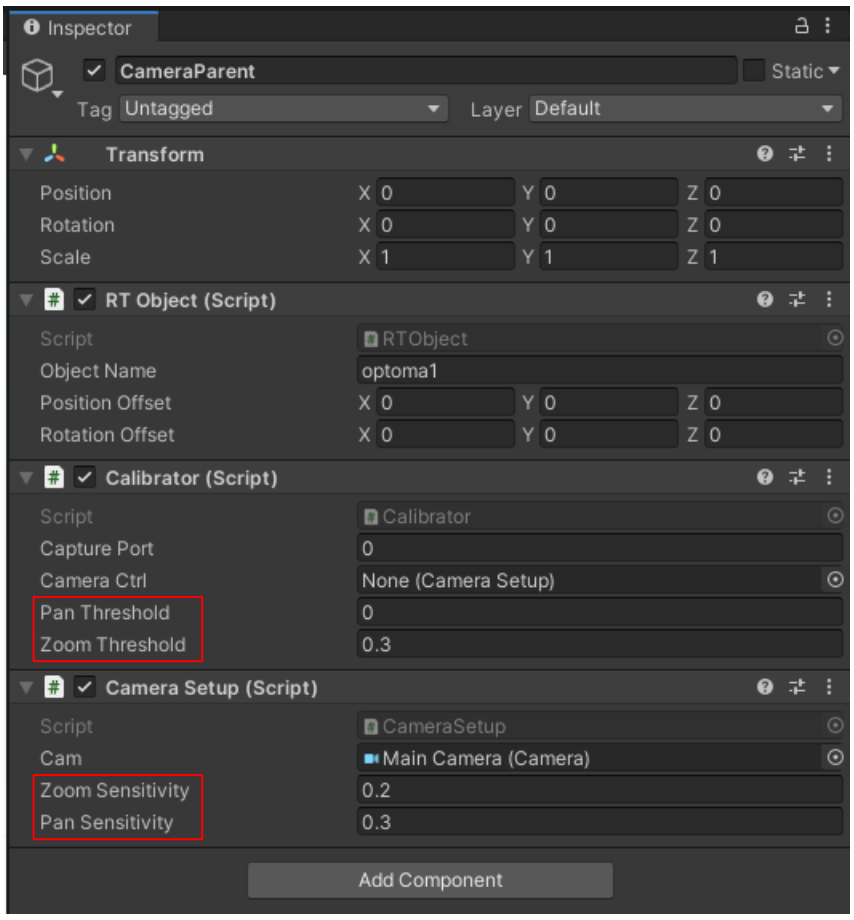

Figure 14.6: Adjusting Calibration Thresholds and Sensitivity

10. Timeline and Schedule

10.1 Timelines

The following figures detail the progress of our project throughout our development. At the beginning of the semester, we outlined a schedule with milestones that provided us with clear goals for our development. The initial timeline can be seen in Figure 15.1.

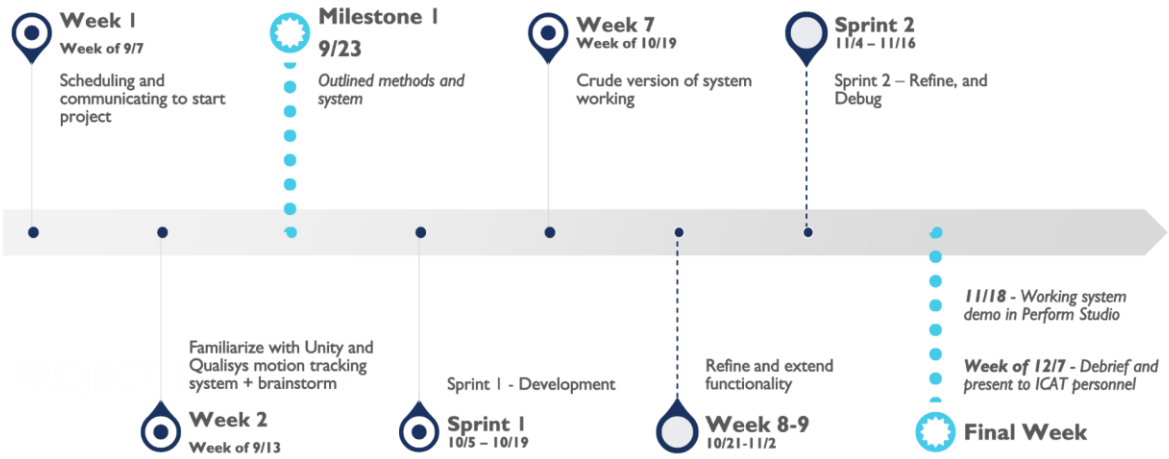

Figure 15.1: Initial Timeline

Figure 15.2 shows the state of the project by the beginning of November. At that point the team was still developing a crude version of the calibration system, which put us behind schedule. In Section 10.2, we outline some of the challenges which caused the team to fall behind.

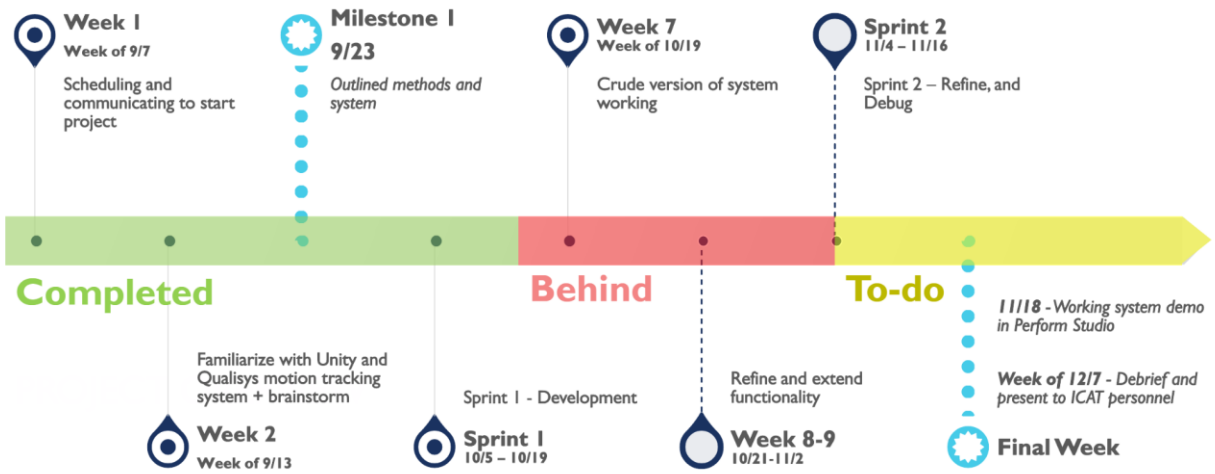

Figure 15.2: Timeline Update 11/03/2021

By the end of the semester, the team achieved the goals planned at the start of the semester. The finished timeline can be seen in Figure 15.3. A summary of the completed project can be found in Section 10.4.

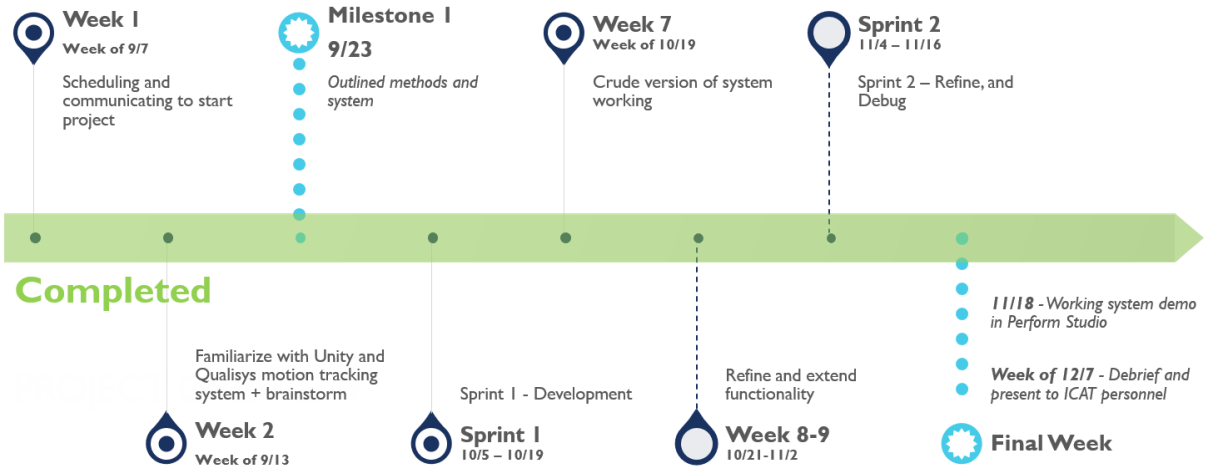

Figure 15.3: Final Timeline Update

10.2 Problems

Building OpenCV

Working with new technologies always introduces new challenges and learning. Working with OpenCV was no different. OpenCV was an integral part of the project and any issues regarding it affected the entire team. Due to a lack of up to date and official documentation for building and installing OpenCV from source, initial development was slow.

Almost 3 weeks were spent learning how to build the C++ OpenCV library from source. This included multiple attempts to build OpenCV and having to restart when libraries were unable to link in the compiler. In addition to building OpenCV, we had to learn how to associate the ArUco module with the base OpenCV library since they were not packaged together by default. Coordinating the components of OpenCV and learning C++ library linking severely delayed the start of our implementation.

OpenCV and Unity Integration

Once OpenCV was installed and development could begin, the team moved towards integrating the C++ OpenCV scripts and Unity. Testing OpenCV locally from C++ was trivial and demonstrated effective ArUco marker detection. To detect markers from Unity, the C++ functions needed to be exposed to the C# scripts of Unity.

Unity supports linking external libraries with the use of plugins. On initial loading of a Unity project, Unity reads and links any Dynamic Link Libraries (DLLs) found in the Plugins folder. Using this plugin functionality, we compiled our C++ OpenCV scripts into DLLs and copied them to Unity. At this step, however, we ran into a number of issues.

One of the immediate issues was that Unity could not load the calibration camera DLL we had compiled. With a vague error regarding not being able to load the DLL, we had trouble resolving and finding the exact fix. While the team was able to eventually address these errors on their personal machines, finding the solution to the issue had to be replicated on the Perform Studio PC. Spending time DLL error fixing in the Perform Studio wasted the limited time we had each week in the studio, again slowing development in the starting month of the project.

ArUco Marker Detection

Another problem we ran into involved the effects of lighting on OpenCV's ability to detect the ArUco markers. Detecting at least 3 ArUco markers was integral to the calibration process. What the team found was that projecting on the physical calibration board would cause the Azure camera to adjust its exposure if the background and foreground had large lighting differences. Increasing exposure, however, washed out almost all of the ArUco markers, leaving none detected.

Adjusting the Azure camera settings required releasing the camera from Unity's use and handing it over to the Azure SDK. Adjustments could be made in the Azure SDK, but to see if the adjustments were helpful, we had to release the camera from the Azure SDK and recapture it with Unity. While not a complicated process, it only further impacted how the team made use of the limited time we had in the Perform Studio.

10.3 Solutions

Building OpenCV

While OpenCV took a while to finally build and install, we made sure to document all of the steps. Doing research and trying out solutions did not always yield a working solution, but did teach us important lessons about how libraries are linked in C++. These lessons helped future attempts to build OpenCV until ultimately we were able to put together a complete guide and successfully build, install, and run OpenCV.

The steps for all things OpenCV are included in the Developer Documentation.

OpenCV and Unity Integration

To resolve the DLL issues we faced when importing the OpenCV code to Unity, we made sure to copy over other DLL files from OpenCV. A given DLL may use other DLLs that are dependencies, and they must be located in the same directory in order to properly link. By copying over all of the DLLs from our OpenCV bin folder to the Unity Plugins folder, we were able to resolve these dependency issues.

Restarting the machine and Unity to ensure system environment variables were up to date assisted in preventing further issues.

ArUco Marker Detection

There was unfortunately no guaranteed solution to this problem. However, we did record lighting conditions and Azure camera settings which worked well and allowed the team to develop without external issues. We wanted our system to be more responsive to different lighting conditions, but, at the conclusion of this semester, adjusting lighting settings until the markers were visible was the only feasible solution.

10.4 Current State of the Project

As of the end of this semester, we completed a number of important steps. For calibration, we constructed a physical board which displays two ArUco markers to be detected by the OpenCV library. Complimenting the physical calibration board, a digital calibration board was modeled and textured for Unity. Finally, we modeled a scatter board of ArUco markers for Unity. These three boards were used to calibrate the Unity camera and projector.

The team manually calibrated the Unity camera and recorded these settings to assist with development. The Unity camera can now be calibrated automatically, but the manual calibration allowed for rapid setup, development, and testing.

With the use of OpenCV, the team was able to detect physical and virtual ArUco markers and determine their offsets. By exposing the OpenCV methods for marker detection to Unity, we were able to detect the ArUco markers from Unity. Using the exposed methods, we also wrote a C# script which panned the camera based on the locations of the markers.

Actual calibration was implemented in the following steps. First, the physical calibration board was placed in the throw of the projector. The scatter board was then displayed by the projector and the calibration script attempted to center and scale the center marker. In its final form, the script first attempted to move the

center scatter board ArUco marker to the center of the physical board. From there, the script aimed to adjust the scale of the marker, so it is the same size as the physical markers already present on the board. The script alternates between panning and zooming until position and scale values meet a certain threshold.

11. Future Work

Our final calibration system could use improvements in accuracy. Our scatter board approach was useful for guaranteeing a marker was seen; however, using a single point to calibrate the projected image does not provide the most accuracy. The team found that after calibrating with the scatter board, switching to the digital calibration board revealed the projected image was still not fully aligned.

We suggest, to reduce this inaccuracy, that a two-step calibration process be implemented. First, the scatter board would be used to center and scale the Unity camera with the accuracy as seen currently. Secondly, the scatter board should be disabled and replaced with the digital calibration board, providing two ArUco markers for calibration. This second step is similar to the initial calibration design and our hope is that it will further reduce the error in calibration, making for a better fitting projection.

Additional future work that can be considered (but has not yet been looked into) include:

- Make performances more interactive and allow for real time animation to be done on stage.
- Bring the real world and video games together by projecting scenes from the video games onto objects around you.
- Collaborate on designing on a model such as a car with a team in real time. This can reduce costs of modeling as the process can be done on the computer and projected onto the physical model in real time.
- Adapt this calibration system to have a projector in every house that can project certain things on certain objects. For example, while you are in the kitchen, you can get recipes projected onto wherever you are looking.

12. Acknowledgements

We would like to give thanks to two individuals in particular. Without them, this project would not have been possible. Our client, David Franusich, was always generous with his time despite having to manage several other internal projects overseen in the ICAT department. Our professor, Dr. Edward Fox, helped us throughout the project as well by giving us meaningful and insightful advice along the way.

Dr. Edward A. Fox
Department of Computer Science
fox@vt.edu

David Franusich
Institute for Creativity, Arts, and Technology
dfranusi@vt.edu

13. References

[1] George White. (2018, January 25). Projected Augmented Reality. Retrieved December 12, 2021 from https://cantina.co/projected-augmented-reality/

[2] Murray-Nag, B. (2017, March 29). 'Lightform' augmented reality projector covers inanimate objects with trippy digital patterns. Retrieved September 2, 2021 from https://www.designboom.com/technology/lightform-ar-scanner-03-28-2017/

[3] Watanabe, Y *et al*. (2021). Projecting Images onto a Moving Object. Retrieved October 10, 2021 from http://www.vision.ict.e.titech.ac.jp/projects/dynaflashv2/index.html

[4] Virginia Polytechnic Institute and State University. (2021). Moss Arts Center. Retrieved December 7, 2021 from https://artscenter.vt.edu/

[5] Virginia Polytechnic Institute and State University. (2021). Institute for Creativity, Arts, and Technology. Retrieved December 7, 2021 from https://icat.vt.edu/

[6] Unity Technologies. (2021). Power better collaboration and creativity. Retrieved September 19, 2021 from https://unity.com/

[7] Microsoft. (2021, March 04). Azure Kinect Sensor SDK. Retrieved September 29, 2021 from https://docs.microsoft.com/en-us/azure/kinect-dk/sensor-sdk-download

[8] Francisco J. Romero-Ramirez, Rafael Muñoz-Salinas, Rafael Medina-Carnicer. (2018). "Speeded up detection of squared fiducial markers". Image and Vision Computing, vol 76, pages 38-47.

[9] S. Garrido-Jurado, R. Muñoz Salinas, F.J. Madrid-Cuevas, R. Medina-Carnicer. (2016). "Generation of fiducial marker dictionaries using mixed integer linear programming". Pattern Recognition: 51, 481-491.

[10] OpenCV. (2021). OpenCV. Retrieved September 9, 2021 from https://opencv.org/

[11] Siddharth Akalwadi, Zachary Long, Shorya Malhotra, Dilan Prasad. (2021). ProjectedAR-CS4624 GitHub Repository. Retrieved September 24, 2021 from https://github.com/Masterfishy/ProjectedAR-CS4624

[12] Unity Technologies. (2021). Camera Manual. Retrieved September 18, 2021 from https://docs.unity3d.com/Manual/class-Camera.html

[13] Unity Technologies. (2021). Download Unity. Retrieved September 19, 2021 from https://unity3d.com/get-unity/download

[14] Unity Technologies. (2021). Installing the Unity Hub. Retrieved September 10, 2021 from https://docs.unity3d.com/Manual/GettingStartedInstallingHub.html

[15] Unity Technologies. (2021). Unity Download Archive. Retrieved September 9, 2021 from https://unity3d.com/get-unity/download/archive

[16] Qualisys AB. (2021). Qualisys Track Manager (QTM). Retrieved September 9, 2021 from https://www.qualisys.com/software/qualisys-track-manager/

[17] Microsoft. (2021). Download Visual Studio. Retrieved September 3, 2021 from https://visualstudio.microsoft.com/downloads/

[18] OpenCV. (2021, July 05). OpenCV 4.5.3: "Open Source Computer Vision Library". Retrieved September 22, 2021 from https://github.com/opencv/opencv/tree/4.5.3

[19] OpenCV. (2021, July 04). OpenCV_Contrib: "Repository for OpenCV's extra modules". Retrieved September 23, 2021 from https://github.com/opencv/opencv_contrib/tree/4.5.3

[20] Kitware. (2021). Build with CMake, Build with Confidence. Retrieved September 23, 2021 from https://cmake.org/

[21] Kitware. (2021). Cmake-generators. Retrieved December 7, 2021 from https://cmake.org/cmake/help/latest/manual/cmake-generators.7.html

[22] Unity Technologies. (2021). Model File Formats. Retrieved September 20, 2021 from https://docs.unity3d.com/2019.4/Documentation/Manual/3D-formats.html

[23] Mountainborn, T. (2021). Creative development using Unity and Unreal. Retrieved September 23, 2021 from https://thomasmountainborn.com/category/unity-tutorials/