DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Standardized AR Marker Integration for Heterogeneous Mobile Augmented Reality Systems with Unity3D and C++

Mayer, Felix Marcel

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Standardized AR Marker Integration for Heterogeneous Mobile Augmented Reality Systems with Unity3D and C++

Integration standardisierter AR Marker in heterogene mobile Augmented Reality Systeme mit Unity3D und C++

Author:	Mayer, Felix Marcel
Supervisor:	Klinker, Gudrun Johanna; Prof. Dr.
Advisor:	Rudolph, Linda; M.Sc.
Submission Date:	15.03.2024

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.03.2024 Mayer, Felix Marcel

Acknowledgments

I thank Janis Reisenauer for helping to create the graphical figures. I thank Paul van der Koehlen for finding credible sources. I thank Tim Braun for proofreading. I thank Dr. Peter Mayer for explaining the mathematical concept of projecting in coordinate systems to me. I thank Elena Domke for her creative input regarding the structure of the conclusion. I thank Prof. Klinker for explaining camera matrices to me. Lastly, I want to thank Linda Rudolph for being a fast, reliable, and helpful advisor when I ran into problems.

Github Copilot was used mainly to answer questions about CMake. The entire chat can be found in the supplementary material. Additionally, it generated the helper function "mirrorImageHoroizontally" needed for the image flipping talked about in 3.9

Abstract

With mobile devices becoming more and more powerful, the possibility of using them for Augmented Reality (AR) becomes more realistic. A significant aspect of AR is marker detection, and ArUco marker detection represents a robust yet flexible solution. OpenCV is a commonly used library for image processing. Among other things, it provides a fast and reliable implementation of the ArUco marker detection. Combining this with Game Engines allows for easy integration in other applications. This thesis creates a way of integrating the OpenCV implementation of ArUco into Unity3D. Specifically, it enables sending images produced by Unity3D to the OpenCV instance. This instance detects markers and does a pose estimation with ArUco markers.

Contents

1 Introduction

Nowadays, smartphones as mobile devices are common and widespread. These mobile devices have become increasingly more powerful within the last few years. This enables them to be used in computational heavy areas, like *augmented reality*. The freedom mobile devices offer by having everything augmented reality needs inside a single portable device is a big advantage.

A big part of augmented reality is determining where the camera and other objects are in relation to each other. A common solution to this problem is *fiducial markers*. *Fiducial markers* are objects inside an image and function as a point of reference. In augmented reality, they can be used to determine the position and orientation of the camera with respect to the markers. They can also be entangled with a virtual object so that it follows the image's marker. [MJA14]

The detection of *Fiducial markers* needs to be fast and reliable. In this case, reliability means that the marker will still be detected even if it is rotated, covered in shadows, or obscured in some other way. Additionally, markers need to be identified with the correct Identification (ID). This becomes important when multiple markers exist. In this thesis, the *ArUco library* for the marker generation and detection was chosen. The *ArUco library* provides a highly reliable marker detection algorithm under occlusion. A robust and well-tested implementation of the *ArUco library* is provided in the *OpenCV library*.

The *OpenCV library* is an open-source computer vision and machine learning software library. Besides ArUco, this library contains an extensive list of image-processing algorithms, which include there are algorithms for face detection, object identification, movement tracking, and image synthesis. Many well-established companies like Google, Microsoft, Intel, and many more use the OpenCV library. OpenCV also supports Windows, Linux, MacOS, and Android ([Opea]). The OpenCV library was chosen as the ArUco implementation because of its versatility and robust implementations.

Besides ArUco, OpenCV also provides algorithms for another important part of augmented reality: *Pose Estimation*. *Pose Estimation* is the process of estimating an object's 3D pose from a set of 2D point projections ([Sze22]). In this case, a pose refers to the position and orientation of the camera with respect to the marker. This allows the system to set world anchors in the virtual world or anchor a virtual object to a real-world position.

Many systems are needed to use the full potential of Augmented reality, e.g. a camera to perceive the environment, virtual objects to interact with, a Renderer to display objects, a physics system so the virtual objects can interact with each other, and many more. These systems also need to be inside a single environment and work together seamlessly to create an immersive experience. Modern *Game Engines* provide all these systems inside a single environment, making them a perfect fit. There are multiple different *Game Engines* publicly available, the common ones being Unity3D, Unreal Engine, and Godot. For this thesis, Unity3D was chosen because Unity3D is already commonly used for augmented reality and works well with Android. The only problem is that Unity3D works with C# scripting. OpenCV provides interfaces for C++, Python, Java, and MATLAB ([Opea]). This thesis will therefore use the C++ version and explore how to make it compatible with Unity's C# scripting.

This thesis aims to integrate the image-processing features, particularly the ArUco library, of OpenCV with Unity3D. The structure of the paper is as follows. Chapter 2 explains the fundamentals needed for the thesis: Compiling and Linking, C++ library types, and ArUco. Chapter 3 explores the procedure. This includes the possible ways to access C++ code from Unity's C# scripting and how to use the ArUco Library in sync with Unity3D. Chapter 5 talks about future work. Chapter 4 is the Conclusion. Chapter 6 contains practical tutorials on how to do certain setups.

2 Fundamentals

2.1 Program compiling and Linking

The "Program Compiling and Linking" section is based on [Świ22] "Chapter 6: Linking with CMake" and [Ste14] "Chapter 2: Simple Program Lifetime Stages", "Chapter 3: Program Execution".

There are five stages between writing code and executing said code.

- Writing the source code
- Compiling the source code
- Linking the source code
- Loading the program
- Executing the program

At the end of the linking stage, the program is completed and ready to be run, which are the last two stages. This section explains the necessary knowledge to understand the different types of libraries, but a lot of details will be omitted. For further and more detailed reading, see [Ste14] and [Świ22]. The compiler itself has multiple different stages. For this thesis, only the input and output of the compiler are relevant.

2.1.1 Compiler

After writing the code, it will be handed over to the compiler. *Compiling* refers to converting code written in a higher-level Programming language into a lower-level programming language. In the C/C++ case, it means transforming the translation Units (Source Code, .c/.cpp files) into binary code with metadata (object files, .o files). Object files are structured in sections that store information about the source code, as seen here 2.1:

- ELF-Header: ELF stands for *Executable and Linkable Format*. This format is the standard in Unix-based systems for every file that needs to be linked or executed.

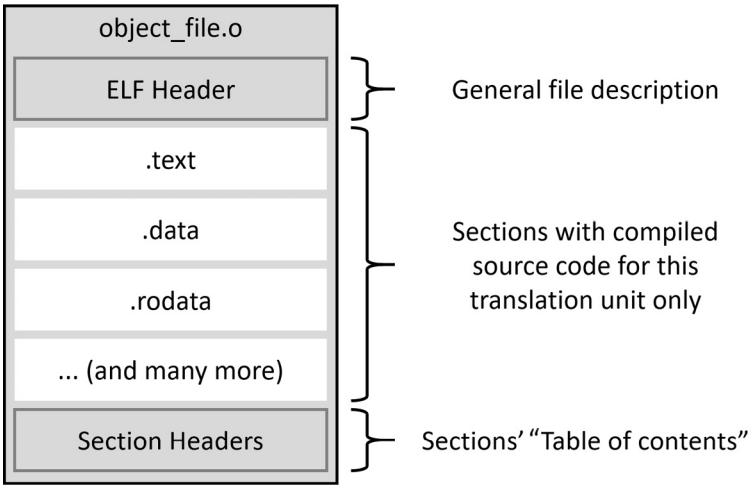

Figure 2.1: The structure of an object file; taken from [Świ22], Figure 6.1

- .text: Contains all the code of the program. It can be viewed as a list of instructions.
- .data: Contains all initialized global and static variables.
- .bss: Contains all uninitialized global and static variables.
- .rodata: Contains all constants variables (read-only data).
- Section Headers: Table of contents of the object file. It describes what sections are inside this object file and where they are.

Symbols Before proceeding, the term *Symbol* must be defined. Symbols are everything the programmer creates and gives a name to. This includes variables, functions, classes, etc. To be more precise, they are references to memory addresses. Every time the code calls the function f, it needs to know where it can find this function inside the object file. So, the compiler assigns the name f a symbol containing the address with the memory location inside the object file of said function.

If the entire program is inside a single object file, finding the symbols is not a big problem. The caller of the function and the function are inside the same .text section of the same object file, making the section's position known. If the caller and the needed symbol are inside different object files, then this becomes a problem.

The compiler compiles each source file into a separate object file. This creates the problem of unresolved references, also known as missing symbols. For example, there are two source files, A.cpp and B.cpp, and A.cpp calls the function g implemented in

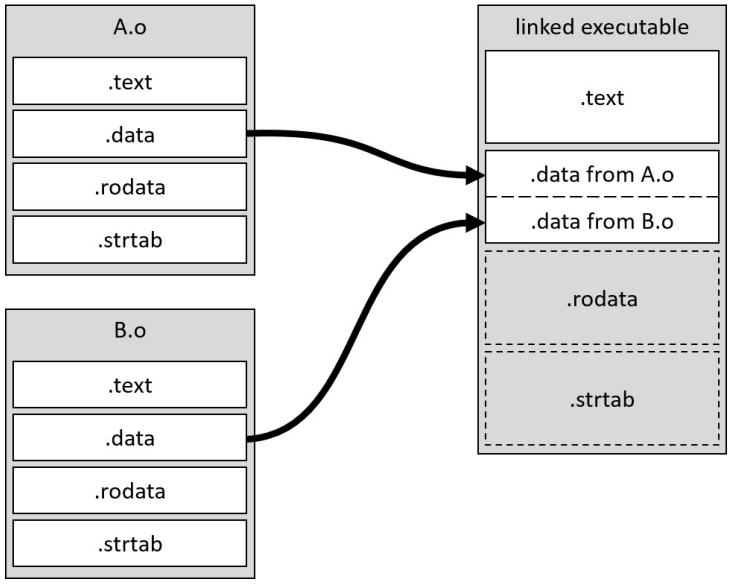

Figure 2.2: The relocation of the .data section; taken from [Świ22], Figure 6.2

B.cpp. When compiled, the compiler produces the object files A.o and B.o. But because the files are compiled separately, A.o has no idea where to find the Symbol of the function *g*. This is where the linker comes into play.

2.1.2 Linker

Object files are not executable by themselves. Even if the program is compiled from a single source file, it still lacks the program header. The linker creates this header and contains information about the program the loader requires. The linker has two tasks: *relocating* and *resolving references*.

Relocating The idea of relocating is to combine all the object files into a single executable. This drastically increases performance as the system does not have to jump between many files. Additionally, all the code and data are in the same file.

Relocating, in this case, means taking two object files and concatenating the same sections, as seen here 2.2.

Resolving references Now, the linker solves the problem of missing symbols from before. The linker will go over every object file, collect all the missing references to other binaries, try to find those binaries and then create the required symbols.

2.2 C++ Libraries Types

The "C++ Library Types" section is based on [Świ22] "Chapter 6: Linking with CMake" and [Ste14] "Chapter 4: Impact of Reusing Concept", "Chapter 5: Working with Static Libraries", "Chapter 6: Designing Dynamic Libraries: Basics".

Libraries are the answer of C++ to the question of how to efficiently reuse code. Additionally, they provide an easy way of distributing code. Theoretically, copying all the needed object files to another system is possible. However, this would require painstakingly copying every object file and keeping track of them. The idea of libraries is to store code inside a file linked to the main executable. This file can be copied to another system for distribution. In C++, there are two kinds of libraries: Static and dynamic. Note that dynamic libraries are often called shared libraries. [1]

2.2.1 Static Library

Static libraries store all the object files inside a single file and then copy them into the executable in the linking stage. Static libraries are just collections of object files. In Windows and Unix-like systems their names end with .lib (library) and .a (archive), respectively. The process is relatively simple as the linker views the library as regular object files. Static libraries are also reversible, meaning you can quickly turn the library into standard object files. When linking static libraries, the linker selectively links. Here is an example to explain what this means:

Assuming you have a static library that contains the object files A.o, B.o and C.o. You want to use code from A.o and C.o inside your program. The linker will only link A.o and C.o to your executable. B.o will not be linked as it is not needed. Note that the linker is selective; it links all of A.o and C.o even if not everything from the object files is needed.

The significant advantage of static libraries is their simplicity. The system running the executable does not need the static library, as everything it needs is already inside itself. The drawback is that the binary size of the executable is bigger.

2.2.2 Dynamic/Shared Library

Dynamic Libraries work differently than static libraries because they are not copied into the executable. While linking a dynamic library, the linker only copies the symbols

[1] In the earlier days of C/C++ programming, dynamic and shared libraries indicated different things. Shared libraries were created with the 'PIC' concept, while Dynamic libraries were not (The 'PIC' Concept will be explained later in the thesis). As the 'PIC' concept became more or less the norm, this naming difference became irrelevant, and the two names merged.

into the executable, not the actual binary code of the library. When the executable is run, the loader loads the program and the dynamic library separately. After that, the library's symbols inside the executable will be resolved. The difference to static libraries is that the dynamic library is now its own instance in the system. That means it allows multiple executables to use the code of the dynamic library while only one instance exists in the system. Dynamic libraries are very close to being an executable, the only difference being that they do not have a starting routine. Note that only the code (.text section) and other unchangeable library sections are shared. Sections like .bss and other variables will be copied for every program using the library. Dynamic libraries on Windows and Unix-like systems end with .dll (Dynamic Link Library) and .so (Shared Object), respectively.

Using dynamic libraries is favorable if multiple programs use the libraries. Additionally, dynamic libraries do not drastically increase the binary size of the executable using the library. The drawback is that it needs to be made sure that the dynamic library is present on the machine executing the program. Dynamic libraries work best if they likely exist on the system running the program. The C++ standard library implementation *libc* or device drivers are typical examples where dynamic libraries fit well.

Dynamic Libraries also can be loaded during runtime instead of being loaded before execution. This is achieved by calling *LoadLibrary*() on Windows and *dlopen()/dlsym()* on Unix-like systems. In this case, they are referred to as *Modules*.

2.2.3 PIC - Positional Independent Code

PIC stands for *Positional Independent Code*, and its invention redesigned how the loading of dynamic libraries works. Before PIC, the loader would alter the .text section of the dynamic library upon loading, known as *load-time relocation*, to fit the particular address mapping of the process. This had the problematic effect that the library was only accessible within the process that loaded the library. Each process using the library would need its own instance. This defeated a significant advantage of dynamic libraries. PIC solved that problem by adding a new section named *Global Offset Table*. It contains the offsets of all the needed symbols so that the library does not need to be mapped to a specific process. Each time at runtime, this table will be updated.

Nowadays, PIC is necessary when building dynamic libraries and is used in other programming regions. Static libraries do not need to be compiled with PIC but can be. Note that static libraries must be compiled with PIC when they are supposed to be linked in dynamic libraries. The PIC option is usually set by default when building dynamic libraries in most building systems like CMake. When compiling with gcc or clang, the compiler option is *-fpic*.

2.2.4 Name Mangling

There are some common linking problems like the *One Definition Rule* and *Dynamically Linked Duplicate Symbols* (cf. [Świ22], cp.6). A particular problem important in this thesis is *Name Mangling*, also knows as *Name Decoration* [Leab].

As discussed earlier, symbols are needed to find functions when they are called. If a programming language prohibits two functions from having the same name, this becomes easy. The name of the function becomes the symbol. If the programming language provides features like *Function Overloading* where two functions can have the same name, creating the symbols becomes more difficult. *Name Mangling* is the process of creating unique symbols for functions [Leab].

In C, it is impossible for two functions to have the same name. Thus *Name Mangling* is not required. However, C compilers are still allowed to use *Name Mangling*. For example, compilers targeted at the Microsoft Windows platform use *Name Mangling* to convey information about the calling convention that was used [Leab]. Assuming we have two functions using different calling conventions.

```
(C on Windows)
int _cdecl fOne (char x);
int _stdcall fTwo (char y);
```

The function *fOne* will get the symbol *_fOne*, and *fTwo* will get *_fTwo@1*. If no calling convention was specified, *_cdecl* will be used. For an overview of how functions will be mangled based on specific calling conventions, see [Leab].

C++ makes this process a bit more difficult by adding features like *Function Overloading*, *Classes*, and *Namespaces*. Now it becomes possible that two functions have the same name but, for example, be part of two different classes. Due to this, a lot more information is needed to mangle a function. This itself is not a problem. The big problem in all of this is that *Name Mangling* is not standardized, neither in C nor C++ ([Leab]). Different compilers, sometimes even different versions of the same compiler, produce different symbols. An example with *Function Overloading* follows.

Function Overloading Example Assuming there is a function *int countDigits(int a);* that counts the number of digits for a given integer. A similar function counting the digits for a given floating point number might also be needed. In C, this function would need a different name. However, in C++, this can be easily realized by *Function Overloading*

```
(C++)
int countDigits(int a);
int countDigits(float a);
```

This creates a problem for the Linker as there are two functions with the same name. Name Mangling solves this by changing the symbol based on the parameters. The table 2.1 shows a small example.

Compiler	void h(int)	void h(int, char)	void h(void)
GCC 3.x and higher	*_Z1hi*	*_Z1hic*	*_Z1hv*
GCC 2.9.x	*h__Fi*	*h__Fic*	*h__Fv*
Microsoft Visual C++ v6-v10	*?h@@YAXH@Z*	*?h@@YAXHD@Z*	*?h@@YAXXZ*

Table 2.1: Name Mangling Table [Wik]: shows how different compilers mangle the names of overloaded functions differently.

This can result in errors like *missing symbol* or *undefined reference* when linking code compiled with different name mangling conventions.

To work around this in C++, it is possible to use C name mangling. By using the Keyword *extern "C"* at the beginning of the declaration of a function, it ensures that the Symbol will be mangled with C linkage.

```
extern "C" int f(int a, int b);
```

Further reading can be found here: [Ste14], [Wik], [Leab], [Wik].

2.3 ArUco

The "ArUco" section is based on [Gar+14].

The *ArUco* library contains a fiducial marker system. This system is specialized for pose estimation in applications such as augmented reality. There are numerous fiducial marker systems out there 2.3.

Each one has disadvantages, e.g. a common one is the fixed number of markers. A lot of systems provide a predefined set of markers that can not be altered. This is obviously problematic when more markers are desired. But this can also be unfavorable when drastically fewer are needed. More markers mean that the inter-marker distance[2] becomes smaller, and thus the inter-marker confusion rate becomes higher. Another common problem is fixed parameters and thresholds. These make them sensitive to changing conditions and not very adaptive. More general problems are high false positive rates, low robustness, and low inter-marker distance, which makes error correction difficult. All these problems are addressed by the ArUco library.

This is an ArUco marker 2.4. ArUco markers are squares divided into a grid structure where each cell is either black or white, representing the numbers 0 or 1. The edges

[2]The inter-marker distance between two markers describes how similar they are.

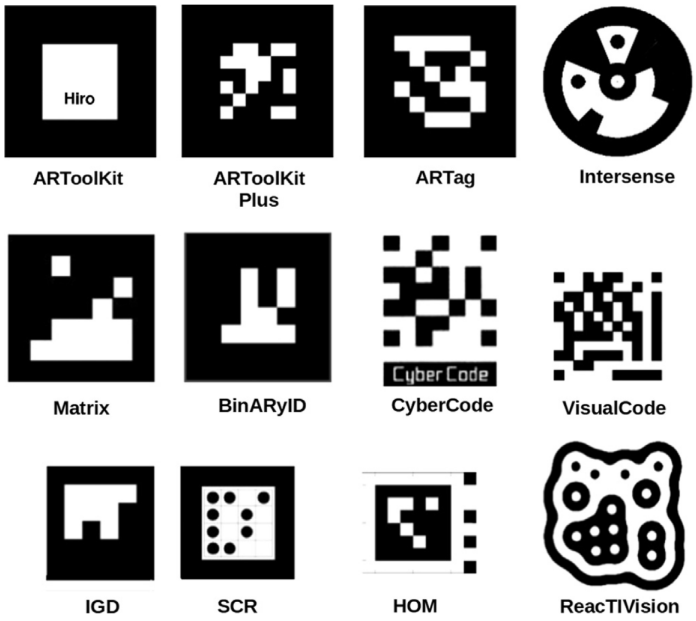

Figure 2.3: Examples of fiducial markers proposed in previous works; taken from [Gar+14], Figure 2.

Figure 2.4: Examples of markers of different sizes, n, generated with the proposed method. From left to right: n=5, n=6 and n=8.; taken from [Gar+14],Figure 3

of the marker are always black and seen as a border. The dimensions of the marker are (n+2)x(n+2), where $n > 0, n \in \mathbb{N}$. n*n is the number of bits used to represent the individual marker. The additional two cells in each dimension are the border of the marker. Squares were chosen as the shape because the four prominent corners are favorable for the pose estimation, while the inner region can be used for marker identification.

One of the big advantages of the ArUco library is the ability to automatically generate its own dictionary of markers. The number of markers generated, as well as their dimension, can be customized. The dimensions define the maximum number of markers inside a dictionary. Additionally, all markers inside a dictionary must have the same dimensions. This concept of automatic generation solves the problem of having too many or not enough markers.

Automatic Marker Generation The ArUco marker can be written as a binary nxn matrix. Before the algorithm starts, a threshold will be chosen. This threshold can be set freely for every individual generation process. The algorithm starts with an empty dictionary. Candidates for new markers are created by a stochastic process, where markers with many transitions between 0 and 1 are selected with higher probability. If the Hamming distances between a candidate and the already existing markers in the dictionary (including the rotated versions of the markers) are higher than the threshold, the candidate is added to the dictionary. Otherwise, it will be discarded. If a predefined number of candidates fail successively, the distance threshold will be reduced. This generating process creates a dictionary with a high inter-marker distance, minimizing the false positive rates.

Marker Detection The ArUco marker detection relies mostly upon already-established methods. It consists of four steps. The input image is taken as a gray-scale image.

1. *Image segmentation*: The most prominent contours of the gray-scale image are extracted.

2. *Contour extraction and filtering*: A contour extraction is performed to produce a set of image contours. Afterwards, a polygonal approximation is performed. Because the markers are rectangular, everything that is not approximated to a 4-vertex polygon will be discarded.

3. *Marker Code extraction*: An analysis of the inner region of the contour. This step tries to see if there is a marker code inside the contours.

4. *Marker identification and error correction*: This last step determines if the marker code belongs to the dictionary. This step is the only novel contribution to the marker detection process.

Through these systems, ArUco provides a robust and reliable fiducial marker system[3].

[3]The original paper about ArUco also contributes to the occlusion problem. This is done with marker boards and color masks. This is not used in this thesis and thus was omitted here

3 Procedure

3.1 Setup

The first step of integrating the OpenCV library is determining the setup.

- *Computer A Creating the Unity Code*: x64-based processor with 64-bit operating system
- *Operating System of Computer A*: Windows 11 Pro, Version 23H2, Build 22631.3007
- *Computer B Creating the Shared Library Code*: Lenovo Yoga 530-14IKB, x86-64 based processor with 64-bit operating system
- *Operating System of Computer B*: Ubuntu 22.04.4 LTS
- *Unity3D Version*: Unity3D 2022.3.14f1
- *Android Device*: LG G8s ThinQ; Android version 12; Qualcomm Snapdragon 855 processor (64bit processor, arm64 architecture)
- *OpenCV Version*: OpenCV 4.9.0

"Unity3D" will be abbreviated to "Unity" in this thesis. This thesis uses the C++ interface of OpenCV. Android devices need to be prepared for Unity to build directly to them. *Developer mode* and *USB Debugging* must be enabled. Additionally, the transfer protocol must be set to *Photo Transfer Protocol (PTP)*. When set to *Media/File Transfer Protocol* or other protocols, Unity will not detect the device. When building to an Android device, the Unity log on the computer will not display anything. *Android Logcat* is a Unity package that allows the log output of the Android device to be displayed inside Unity on the computer. Unity needs to be set to build for Android.

3.1.1 Scripting Backend

Unity provides two scripting backends to compile the project for Android: Mono and IL2CPP. The big difference between them is that Mono compiles at runtime while IL2CPP compiles everything into C++ code and then uses this C++ code to create native

binary files. In the player settings, where the scripting backend can be set, it shows that the Mono compiler is only capable of building to ARMv7 (arm32 bit). The IL2CPP can compile for ARMv7, ARM64, x86, and x86-64. As the mobile device used for this thesis runs on ARM64, IL2CPP was chosen to be used. It is possible for ARM64 models to run ARM32, but this will stop in the near future ([Cun], [Wil], [Ama]). Later in the thesis, the scripting backend would be changed to Mono because of a problem caused by IL2CPP (see 3.9)

3.2 External Code inside Unity3D: Plug-ins

This Section is based on [Docb] and [Doca]

Unity3D provides a concept named *Plug-ins* for using externally created code inside Unity3D. Plug-ins can generally be separated into two broad categories: Managed and Native.

Managed Plug-ins are *managed .NET assemblies*. With some exceptions, this means that they are written in C#. This implies that they can only access features that the .NET libraries support. Managed Plug-ins and Unity3D script code are practically the same, the only difference being that managed Plug-ins were compiled outside Unity3D. Thus, Unity3D might not have access to the source.

Native Plug-ins are platform-specific code libraries. This allows them to access features like operating system calls and third-party code libraries that would otherwise be unavailable to Unity. Native Plug-ins use a C-based call function. This means functions must be declared with C linkage to avoid Name Mangling (see ch. 3.2.4 Name Mangling). Native Plug-ins have the downside that Unity3D cannot access them as it can with managed Plug-ins. Unity3D will not be able to recognize inconsistencies. For Example, Unity3D will not notice missing Plug-ins until the program is run and an error is thrown.

Native Plug-ins are usually used when using C++ inside Unity; thus, this thesis focuses on them. There are three stages to them: Creating, Importing and Calling. Calling will be discussed in the next Section *DLLImport and P/Invoke* (3.3). Importing is a very simple process that works mostly the same for every type of native Plug-in. For this reason, there wont be a specific section on this but a small tutorial 6.0.1. The creation part will be addressed later in the thesis (3.4).

3.3 P/Invoke and DllImport: Calling C++ Code from C#

Platform Invoke, or P/Invoke, is a technology provided by the .NET framework. It allows managed code to access unmanaged code. Note that the unmanaged code

must be compiled as a dynamic library and will be loaded at runtime ([Leae]). In the Unity3D case, native Plug-ins are unmanaged code. The idea behind P/Invoke is to declare a static function *A'* inside the managed code, which corresponds to a function *A* inside the unmanaged code. The managed code only has a declaration; the actual implementation is inside the unmanaged code. The managed code needs some way to know where to find this implementation. This is the job of the Attribute *DllImport*.

DllImport *DllImport* is an attribute indicating that this function is the static entry point for an unmanaged function. Additionally, it provides the information needed to call a function from unmanaged code ([Leac]). Note that this attribute can only be given to functions. The minimal information needed is the name of the file containing the unmanaged code. For other optional parameters and their default states, see [Leac].

3.3.1 Practical Example

The following example is taken from the code of the thesis and contains small alterations for demonstration purposes.

```
C++ Code

1) extern "C" void InitOpenFrame(int width, int height){...}
2) extern "C" int GetArucoDrawing(int** rawImage){...}


C# Code

1) using System;
2) using System.Runtime.InteropServices;
3)
4) public class OpenCVManager2 : MonoBehaviour{
5)
6) private const string LIBRARY_NAME = "ArucoBridgeLibC";
7)
8) [DllImport(LIBRARY_NAME)]
9) private static extern void InitOpenFrame(int width, int height);
10) [DllImport(LIBRARY_NAME, EntryPoint = "GetArucoDrawing")]
11) private static extern int ArucoFunction(ref int[] rawImage);
12) ....
```

This example assumes that the C++ code is already compiled into a library, and the corresponding C# code should be written. Most of what is needed is contained inside the *System* and *System.Runtime.InteropServices* namespaces (C#, lines 1-2). The first step is to declare a function. This function must have the keywords *static* and *extern* keywords. *extern* tells the compiler that the function's implementation is located elsewhere. The *DllImport* provides the compiler with the said location, in this case *ArucoBridgeLibC*. *DllImport* also tells the compiler to load the unmanaged library. Note that loading the Dll will only occur on the first call to the function ([Leaa]).

To start, the first C# function *InitOpenFrame* (line 9) will be examined. The *DllImport* (line 8) only provides the location of the file containing the implementation. To find the correct C++ function, the only name of the C# function must be identical. This is because the C++ functions were compiled without name mangling (see 2.2.4). For example, using the correct name with a different return parameter will not throw an error but might cause overflows or segmentation faults. To work properly, the whole signature must be the same. The name, return parameter, number of input parameters, and type of input parameter define the signature of a function.

If it is desired for the C# function to have a different name than the C++ function, it is possible to change that. The field *EntryPoint* of the *DllImport* attribute (line 10) allows it to define the name of the C++ function while the C# function name differs.

The next paragraph addresses why the library name has no file extension. This has to do with the filename itself.

Library File Name First of all, even though the name of the attribute is ***Dll****Import* this works also on other platforms like Linux and MacOS, both of which do not have .dll files but their version of dynamic libraries. For dynamic libraries, Windows and MacOS both just use the name of the file and append the extension .dll and .mylib, respectively (aLibraryName -> aLibraryName.dll/aLibraryName.mylib). Linux adds its extension .so but also adds the word "lib" at the beginning of the name (aLibraryName -> libaLibrary.so). This thesis uses Linux-based dynamic libraries for reasons explained in a later section (3.5.2).

The official Microsoft .NET documentation on P/Invoke ([Leae]) uses the whole name of the file, which includes the extension and the leading "lib". The official Unity3D documentation on native Plug-ins ([Doca]) uses the name of the file without the extension or leading "lib". Some unofficial forum posts explicitly state that regarding Unity3D, it is necessary to only use the name without the extension or leading "lib" (one example: [oT2]). While testing the code for this thesis, both versions worked while working in Unity3D. This thesis did not test this for things outside of Unity3D.

3.3.2 Marshalling

Additionally, *Marshalling* needs to be addressed. Different programming languages have different data types. For example, C# has the type *bool* containing either *true* or *false*. By default, C++ does not have a data type for this. In practice, C++ just uses the numbers 0 and 1 to represent *true* or *false*. Another prominent example of this is Strings. C++ does not have a *string* data type like C# but has multiple different versions of how to represent a string. Additionally, it is possible that two programming languages use the same data type to represent something, but the sizes of the types differ. C++ can represent an integer with 1 byte (int8_t), 2 bytes (int16_t), 4 bytes (int32_t), or 8 bytes (int64_t). This can create a problem when sending parameters between native and managed code.

Marshalling is the process of transforming types when they need to cross between managed and native code. The runtime has default rules on how to marshall common types ([Leag]). These rules will automatically be applied unless specifically disabled. This is the reason why, in the example above, no marshalling was specified. To indicate a specific marshalling, the *MarshalAs* attribute is provided by the .NET framework. This attribute can be applied to a field, method parameter, and method return value. The following example was taken from [Leag].

```
[DllImport("somenativelibrary.dll")]
static extern int MethodA([MarshalAs(UnmanagedType.LPStr)] string parameter);
```

A complete list of marshall options, including more information, can be found here: [Leag], [Lead].

3.4 Flowchart of the Creation Process

To reiterate, the goal of this thesis is to be able to use the ArUco part of the OpenCV library in Unity on an Android build. The problem is that the OpenCV library was not compiled with C linkage (2.2.4). A native wrapper library is needed. The structure is that OpenCV will be linked into this native wrapper library, which is compiled with C linkage. This thesis discussed how to import and call native Plug-ins. The last part is the creation of them. This thesis will explore three possible paths: DLL on Android, Unity's Native Plug-ins for Android, and Android Archive Plug-ins/Android Library Projects. The following graphic provides an overview of the different paths:

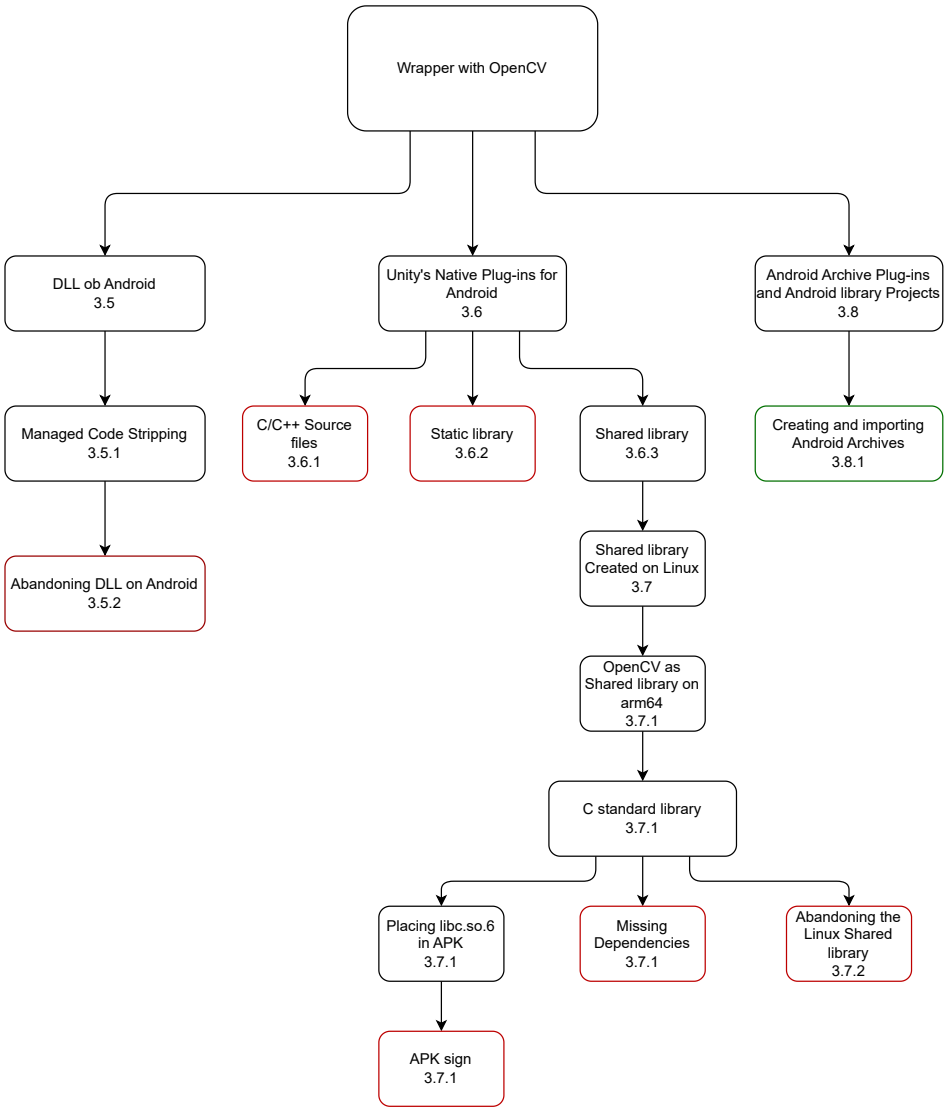

Figure 3.1: Flowchart of what led to what

3.5 DLL on Android

This is the naive approach to coping with what works on other platforms like Windows. This approach creates a .dll file, imports this as a Plug-in in Unity by placing it in the "Plugins" folder, and calls it with P/Invoke. At the moment, the dynamic library inside the .dll file will only contain a dummy function returning an integer. After that, the application can be built to an Android device (see 6.0.2).

When running the application, an error will be thrown:

```
DllNotFoundException: Unable to load DLL [...] library not found
```

This error indicates that the .dll file is not inside the application. It is possible to examine the contents of the application. The application will be stored inside a .apk file when building to Android. APK (Android package) is a file format containing Android app information required by the runtime. Android devices use this format to install apps ([Deva]). This file format is conveniently a ZIP archive ([Devd]). This means that by renaming the file and changing the extension from .apk to .zip, it is possible to unzip it and examine the contents. The .apk file has a "lib" folder containing all the libraries of the application. As foretold by the error message, the library does not exist inside the .apk file. There are multiple forum entries on the internet that have the same problem ([pau], [jim], [Ars]). The answers to these suggest that the problem is that the Unity Linker is stripping the plug-in.

3.5.1 Managed Code Stripping by the Unity Linker

The *Unity Linker* is a tool used by the *Unity Build process* for stripping managed code. The *Unity Linker* removes unused or unreachable code during the build process. This can be controlled in three different ways: *Managed Stripping Level*, *Preserve Attribute*, and a *link.xml* file. *Managed Stripping Level* is a property in the player setting and sets the general rules for how the *Unity Linker* searches. The *Preserve Attribute* can be given to individual types to ensure that they will not be stripped. *link.xml* is a file containing specific information about which files and what content of the files should be preserved. ([Mann], [Mank]).

In the case of this thesis, the *Preserve Attribute* is not usable because attributes do not exist in C++. With no success, the *Managed Stripping Level* was set to minimal, the lowest option on the IL2CPP compiler. A *link.xml* was created and placed next to the Plug-in. The *link.xml* contained this:

```
<linker>
  <assembly fullname="ardll.dll" preserve="all"/>
```

```
</linker>
```

3.5.2 Abandoning Dll on Android

Using the name of the assembly without the extension or other variations of the *link.xml* was unsuccessful. It is unlikely that *Managed Code Stripping* is the problem in this specific case. If the Plug-in was actually stripped, then the minimal option in the *Managed Stripping Level* should have solved this problem. This option makes Unity not remove any user-written code. Another indicator is the name of the concept. *Managed Code Stripping* implies that it can only affect managed code. This idea is further cemented when remembering that Unity is incapable of accessing native Plug-ins in the same way as with managed Plug-ins (see 3.2). This ability would be needed when determining which code is unused or unreachable. After this, the focus of the thesis shifted to the Android architecture to find a solution.

Android Architecture

Android is an open-source operating system often associated with Java. However, this Java association is only an API framework. Internally, Android runs on a modified version of the Linux kernel ([Devg], Fig. 3.2) written in C ([Kof89]).

As mentioned in the Fundamentals chapter 2, Linux does not use .dll files as .dll is a Windows system. On Linux, Dynamic libraries are .so files. A possible explanation of the missing .dll file might be that the Linux kernel is incapable of using the .dll file and thus Unity does not copy them into the .apk file. This explanation leaves something unexplained. When compiling with the Mono compiler, there is a folder called "Managed" (assets/bin/Data/Managed) inside the .apk file containing .dll files. This indicates that using .dll files in an Android build is possible.

The following is a hypothesis made by this thesis. The Unity documentation states that managed Plug-ins are, like native Plug-ins on Windows, also compiled into .dll files ([Manl]). It could be possible that Unity acts as some kind of container that runs the .dll files. But for this to work, Unity would need access to the .dll files. Unity only has this access if the .dll file is a managed Plug-in, meaning it was written in C#. These .dll files only exist when compiled with the Mono compiler because the IL2CPP compiler turns everything into C++ code (see 3.1.1).

There are some forum posts that confirm the hypothesis that only .dll files with C# code can be used on Android ([Cod], [Hul]). The answer on the post by [Hul] explicitly mentions that the Mono runtime functions as a container, loading and executing the .dll file. Unfortunately, neither of these posts cites any source or is answered by an official Unity developer. This means that this hypothesis still needs to be proven.

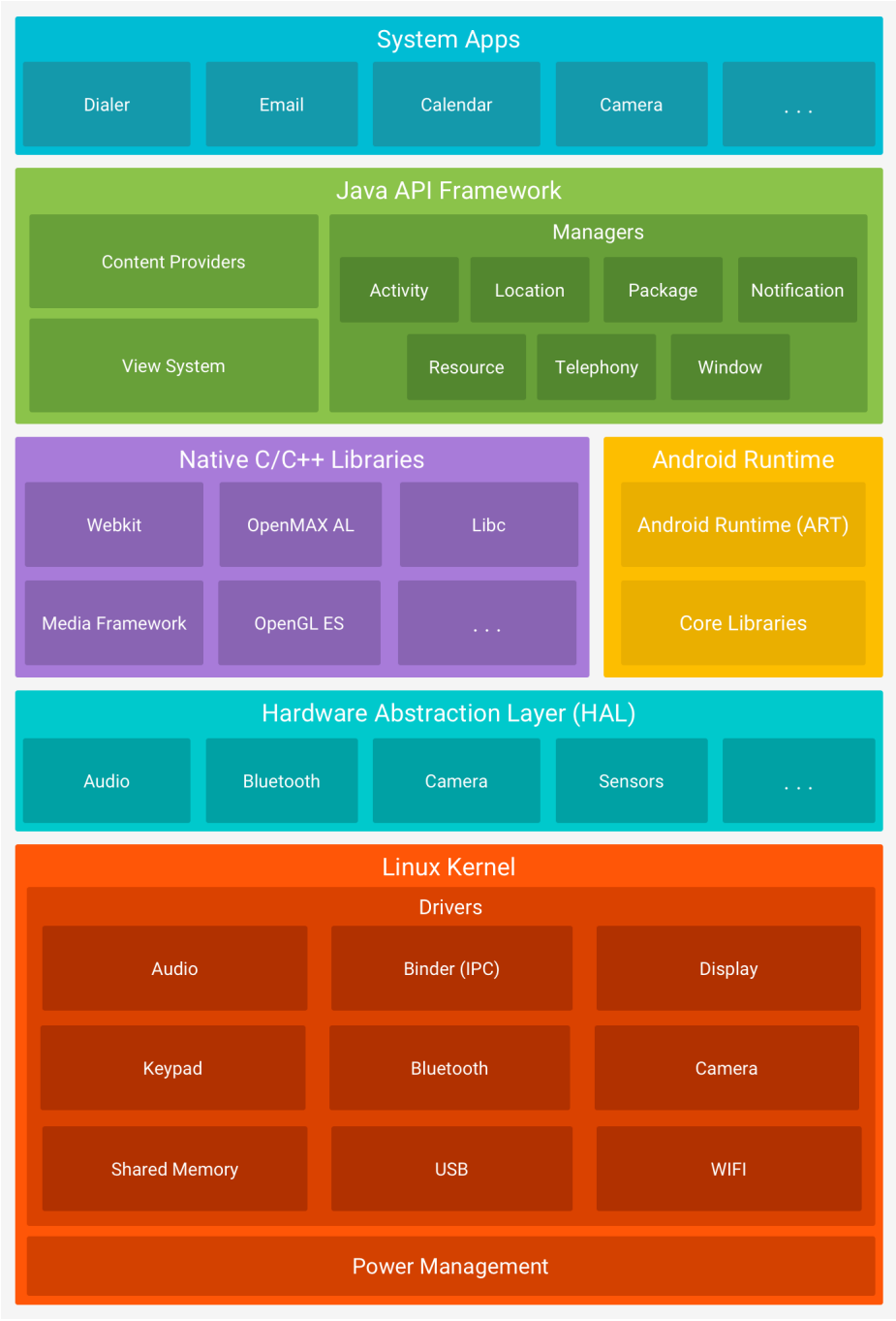

Figure 3.2: Architecture of the Android Platform, taken from [Devg]

3.6 Unity's Native Plug-ins for Android

The Unity manual contains a section about Plug-in types for Android ([Mana]). It mentions native Plug-ins specifically for Android. Native Plug-ins are again separated into 3 different types: Shared Library, Static Library, and C/C++ source files ([Manh]).

Scripting backend	Shared Library	Static Library	C/C++ source files
IL2CPP	Yes	Yes	Yes
Mono	Yes	No	No

Table 3.1: Table shows which scripting backend supports which Plug-ins type ([Manh]).

As stated previously, Unity has two scripting backends: IL2CPP and Mono. The scripting backend determines which Android native Plug-ins are compatible with Unity. The Table 3.1 shows this compatibility. At this point in the thesis, the IL2CPP compiler was used. This allows a greater variety of Plug-ins options.

Following are the Plug-in options, beginning at C/C++ source files.

3.6.1 C/C++ source files

This includes all C/C++ source files with the extensions *.c*, *.cc*, *.cpp*, and *.h* ([Manh]). These files can be directly placed in the Plug-in folder[1]. These files will be directly compiled by IL2CPP into a library and regarded as internal library code. When specifying the name of the library in the DllImport attribute, the name "__Internal" must be used ([Manc]).

The thesis tested this Plug-in type successfully with simple dummy C++ functions that return an integer. Unfortunately, it is impractical for this use case. The problem is that the OpenCV library is a complicated construct with specific toolchains for compiling. These toolchains would need to be recreated for the IL2CPP compiler. This task is beyond the scope of this thesis and may even be impossible without rewriting the OpenCV library.

3.6.2 Static Libraries

The Unity manual ([Manh]) states that static libraries (.a files) are an option. But during testing, when trying to load the library, the runtime throws an error.

```
DllNotFoundException: Unable to load dynamic Library
```

[1]It probably even works if the files are placed anywhere in the *assets* folder or in any subfolder. However, this was not tested by this thesis.

This makes sense when considering what DllImport does and what static libraries are (see 2.2). Static libraries are linked statically by the linker after compilation, while DllImport loads dynamically during runtime.

A possibility is that static libraries can only be called by a C/C++ source file. The idea was that C/C++ source files would be compiled by the IL2CPP compiler and then linked by IL2CPP. The thesis tested this only briefly but without success. Using static libraries does not work based on the reasons covered in 2.2. There could be a different way to use them, but no reason as to why it was found in the Unity Manual.

3.6.3 Shared Libraries

With the former two options deemed as not working or not practical, Shared libraries were chosen. They need to have the extension *.so*. The importing and calling of shared libraries work exactly the same as with .dll libraries on Windows. The different part is the creation. As Android is Linux-based, a logical idea is to compile the needed library into a shared library on a Linux system. Before using a big complicated library like OpenCV, a dummy library was used to test the integration of shared libraries with Unity on Android.

3.7 Shared Library Created on Linux

This section builds on the shared libraries of the last section (3.6.3) and explores ways of implementing it. This section tries to create the wrapper library on Linux and tries to link OpenCV with it. Before trying to use the OpenCV library, a dummy library will be used. The dummy library contains this code:

```
(shared.h)
extern "C" int getLibNumber(void);

(shared.cpp)
#include "shared.h"

int getLibNumber(void){
    return 8;
}
```

There are many websites explaining how to compile a shared library on Linux (see 6.0.3). The GCC and Clang compilers are usually used. However, when running the

Unity application, in the case of this setup and most likely in other setups, it still throws an error.

```
DllNotFoundException: Unable to load [...]
```

The reason for this is that the library was compiled for the wrong architecture.

Architecture of processors The architecture of processors describes their functional specifications. The architecture can be thought of as a contract between hardware and software. It specifies what functionality the processor has. With this, the software knows what it can rely on ([Devf]). There are many different architectures, and when compiling into binaries/machine Code, which C/C++ does, the code needs to be compiled for the specific architecture.

In this case, the computer compiling the library has the x86-64 architecture, while the mobile device has the arm64 (see 3.1). The term *Cross compilation* refers to the act of compiling for an architecture that differs from the architecture compiling the code. To cross-compile, specific compilers are needed. In this case, the *g++-aarch64-linux-gnu* is needed ([Pac]), which is the compiler for C++. There is also *gcc-aarch64-linux-gnu* for C. When using *gcc-aarch64-linux-gnu* for C++ code, it throws an error.

```
error trying to exec 'cc1plus'
```

After compiling the library with *g++-aarch64-linux-gnu*, the unity application works and does not throw any error. The next step is compiling the OpenCV library for the arm64 architecture.

3.7.1 OpenCV as Shared Library on arm64

OpenCV provides multiple versions and programming interfaces for different programming languages and supports a variety of platforms. OpenCV allows building the C++ library from the source and provides toolchains for different architectures. With the "aarch64-gnu" toolchain, the library can be compiled for the arm64 architecture ([Opeb], see 6.0.4). The OpenCV functions are not compiled with C linkage, making them inaccessible to the C# code. To work around this, C++ wrapper functions are needed. The wrapper functions are compiled with C linkage and get called by the C# code. Then the wrapper functions call the OpenCV code. The structure looks like this: The OpenCV static library gets linked into the dynamic library with the wrapper functions. The wrapper library gets placed in Unity as a Plug-in (3.3).

It is possible to link the OpenCV library dynamically, but because the OpenCV library is separated into single modules, it would require to always bring all the needed modules. To avoid this, the library was linked statically.

When trying to run the Unity application, the error will be thrown.

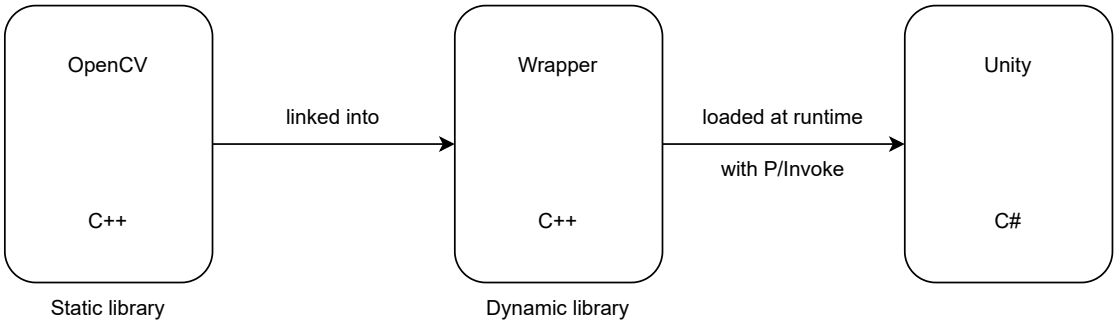

Figure 3.3: Shows the library structure

```
DllNotFoundException: Unable to load dynamic library
'libstdc++.so.6' because of 'Failed to open requested
dynamic library [...]'
```

This means that the library has a dependency on the dynamic library "libstdc++.so.6" which is not found. To understand "libstdc++.so.6" in more detail, an explanation follows.

C Standard Library

The standard library provides definitions for the entities and macros inside the language. Both C and C++ have their own standard library, but the C++ standard library also includes their own version of parts of the C standard library. The reason for this is that C code can still be compiled with C++ compilers ([cpp]). Even though the standard library defines its content, it does not implement it. There are multiple different implementations, each with advantages and disadvantages. The important libraries for this thesis are: *libc.so.6*, *libstdc++*, *libc++*, and *libm.so.6*.

libc.so.6, often referred to as glibc, is a C standard library implementation created by GNU. This implementation is commonly found in Unix-based systems, namely Ubuntu ([Pag]). *libstdc++* is the same but for the C++ standard library ([gnu]). *libc++* is a C++ standard library implementation created by llvm ([llv]). This implementation is the one used by Android ([Devb]). *libm* refers to the math library. This library provides elemental mathematical functions and floating point environment routines. *libm.so.6* is the GNU implementation ([Uni]).

Missing Dependencies

With this knowledge, the missing dependency problem can be addressed. The OpenCV library, and thus also the wrapper library, uses *libstdc++* as the standard C++ library. Android does not provide this library[2], as it uses *libc++*. The simple solution is to statically link the library. This has the advantage that there is only one file that needs to be transferred. The GNU compiler provides the link option "-static-libstdc++" for this ([GNU]). Now the *libm.so.6* dependency is missing showing that *libstdc++* is not the only missing dependency, just the first that comes up. It is possible to see a dynamic library's dependencies (see T6.0.5). When doing so, five dependencies show up: *libstdc++.so.6*, *libm.so.6*, *libgcc_s.so.1*, *libc.so.6*, and *ld-linux-aarch64.so.1*. All five of these need to be present in Unity for the application to work.

Statically linking *libm.so.6* and *libc.so.6* as *libm.a* and *libc.a* did not work. The linker does not throw any error, but when looking at the dependencies, they are both still linked dynamically. The next idea was to tell the compiler to link the entire project statically. The GNU compiler provides the link option "-static" to compile the whole project statically ([GNU]). Linking the wrapper library with this option throws an error.

```
final link failed: bad value
```

(see supplementary material 7 for the entire error).

Note that this did not happen with the dummy library 3.7 from the previous section. The reason for this is that the dummy library does not depend on any external code, not even the standard library.

It is unclear why manual linking *libm.so.6* did not throw any error. However, there is a possible theory on why both ideas did not work which can be seen as part of the following error message.

```
relocation R_AARCH64_ADR_PREL_PG_HI21 against symbol
‘_dl_debug_state’ which may bind externally can not
be used when making a shared object; recompile with -fPIC
```

As mentioned in Section 2.2.3, static libraries need to be compiled as positional independent code to be linked into dynamic libraries. The theory is that *libm.a* and *libc.a* have not been compiled with -fpic and thus cannot be linked into dynamic libraries.

A possible solution could be to get the source code and compile it with -fpic. However, this was never done in this thesis because the error message also hinted at a different problem:

[2]Android provides a file called "libstdc++.so", which is not a full C++ standard library implementation. This file should not be confused with the GNU libstdc++. See [Devb] for more information.

```
opencl_core.cpp:[...]: warning: Using 'dlopen' in
statically linked applications requires at runtime
the shared libraries from the glibc version used for
linking.
```

The *opencl_core.cpp* is a file from OpenCV. This means that even when linking *libc.a*, *libc.so.6* would still be needed inside Unity. This made the focus change from statically linking to bringing the shared library to Unity.

Placing libc.so.6 in the APK

When placing the *libc.so.6* file inside the Unity Plug-in folder, Unity does not recognize it as a shared library. After some testing, it became clear that Unity can only recognize a shared library with the extension ".so". Because of this, Unity does not copy the file into the .apk file. The solution is to manually copy the needed files into the .apk file after Unity is finished building.

Unity provides callbacks that will be called at certain times in the pipeline of creating the application. *OnPostProcessBuild* will be executed after building the player ([Manj], [Manm]). This can be used to edit the .apk file before it is deployed on the Android device. However, when deploying the .apk file, the error "CommandInvokationFailure: Unable to install APK to device" will be thrown. The reason for this is that the APK signature is invalid.

APK Signing An APK needs to be signed to be installed on an Android device. There are two types of signing: Debug and Custom. Debug signing is a default signing method that allows the application to be run on Android devices but not to be published. Custom signing allows the application to also be published ([Deve], [Manb]). When *OnPostProcessBuild* will be called, the APK has already been signed by Unity. Editing the file will result in the signature to be invalid, making it not possible to deploy on an Android device.

Unity provides another callback: *OnPostGenerateGradleAndroidProject*. This is called after the Android Gradle project is generated but before the building and eventual signing begins ([Mani]). Unity explicitly specifies that this callback can be used to modify or move files before Gradle builds the application ([Manf]). Using this callback also ends in another error.

```
UnauthorizedAccessException: Access to the path
'[...]/OpencvTestWoky/Library/Bee/Android/Prj/IL2CPP/Gradle/unityLibrary'
is denied.
```

The error suggests that the program does not have the right to access the needed folders. Moving the entire project to a different, unprotected location does not have any effect.

The Unity Manual mentions that it is possible to export the Unity project as a Gradle project and import that into Android Studio. This grants greater control over the build pipeline or the ability to modify the Android manifest ([Manb], [Mane]). It could be possible that this allows the addition of the *libc.so.6* file without throwing an error. However, this was never tested in this thesis as this approach was abandoned due to many reasons.

3.7.2 Abandoning the Linux Shared Library Approach

Following is the list of steps to make the OpenCV library accessible in Unity. Steps 1-3 were researched and tested by this thesis. The other steps are assumptions based on the insights gained in the previous section.

1. Get the source code of the OpenCV library.
2. Cross-Compile the library with the aarch64 toolchain.
3. Create a dynamic wrapper library that links the OpenCV library.
4. Create the Unity project.
5. Export the project to Android Studios.
6. Add the wrapper library to the project.
7. Build, sign, and deploy the project with Android Studios

It becomes clear that this approach is very convoluted, especially with certain setups. In the case of the thesis, the Unity part is done on Windows, and the GNU aarch64 compiler needs a Unix system. This was realized by having a second computer with Ubuntu. Additionally, the eventual library does not consist of one file but multiple: The wrapper library itself and at least three dependency libraries. Every new dynamic dependency adds a new file to keep track of. On top of all that, there are unofficial forum posts that state that Gradle is incapable of packaging a library that does not match the naming pattern *lib*.so* ([Dan]). This still needs to be proven, but if it is true, it would make this whole approach impossible. For these reasons, the approach of using a shared library created on Linux was abandoned.

3.8 Android Archive Plug-ins and Android Library Projects

Android libraries are Android app modules that compile into *Android archives* (.aar files) instead of APKs (.apk files). They include everything they need to build an app and can be used as a dependency for Android apps ([Devc]). Android archives are contained in a single file, while Android libraries are directories with a specific structure. Unity recommends using Android archives if the distribution of the Plug-in is desired ([Mang]). For this reason, the thesis chose them over Android libraries. The basic idea of this approach is to have a module containing the wrapper code depending on another module containing the OpenCV library.

3.8.1 Creating and Importing Android Archives

This section is based on [Devc], [Voib], and [Voia]

Android archives are created as modules in Android Studio. This Android Studio project is not actually used but is more of a vessel to create the modules needed. First, the OpenCV library needs to be imported as a module. OpenCV provides a prebuild version of this module that can be imported ([Opee], click the Android button). After this, the new wrapper module needs to be created. Android Studio provides the option to add all the needed C++ files automatically. Now the dependencies need to be set so that the wrapper library can use the OpenCV library. *<All Modules>* and *app* need to depend on OpenCV. This allows the wrapper library to access the OpenCV library while coding. To use the OpenCV library while executing, it still needs to be linked inside the CMake, and the Gradle needs to be given the location of the OpenCV implementation. The last thing is to include the *jni.h* header file inside all the source files that use OpenCV. After building the module, the .aar file can be found at "[Path to the Android Studio directory]/*ModuleName*/build/outputs/aar". This file can be placed inside the Unity "Plugins" folder. The name of the .aar file will probably differ from the name of the .cpp source file. The name of the Plug-in specified in the DllImport attribute needs to be exactly the same as the name of the .cpp source file. When running the application, no error is thrown. The .aar file can also be used on different Architectures: arm64-v8a, armabi-v7a, x86, and x86_64. This can be seen by unzipping the .aar file.

For a full and detailed tutorial, see 6.0.6.

3.9 Using ArUco in Unity

The code implemented by the thesis is meant as a proof of concept. It shows the basic capability of detecting ArUco markers, estimating their pose, and drawing coordinate systems into them.

The idea is that the Unity code fetches the camera image and calls the wrapper function, giving a reference to the image as the parameter. The wrapper function uses this image to detect the markers. Additionally, it draws red circles around the corners of rejected candidates and green ones around the accepted candidates. After this OpenCVs pose estimation function *solvePnP* is used to calculate the poses of the markers. Lastly, the resulting rotation and translation vectors produced by the pose estimation are used to draw coordinate system frame axes onto the markers.

The combination of ArUco and Unity creates some difficulties.

Marshalling the Image input from Unity to ArUco In Unity, the image gathered from the camera is an array of pixels, Color32[] to be specific. Color32 is a struct that represents an RGBA Color and has a size of 32bit. Each part of the RGBA is stored as a byte ([Mand]). Structs are value types that encapsulate data ([Leaf]). That means in arrays, their data is simply lined up. An example follows to make this more understandable.

```
Color32 ar[] = new Color[2];
ar[0] = new Color32(255, 12, 0, 76);
ar[1] = new Color32(213, 0, 23, 25);

Contents of ar (size of each number = 8bit):
255,12,0,76,213,0,23,25
```

This becomes helpful as Color32 does not exist in C++[3] while integers with 8bits do exist. The resulting P/Invoke functions look like this.

```
(C#)
[DllImport(LIBRARY_NAME)]
private static extern int GetArucoDrawing(ref Color32[] rawImage);

(C++)
int GetArucoDrawing(unsigned char** rawImage);
```

Destroyed Pointer by Scripting Backend Calling the wrapper function with the image pointer as a parameter destroys the pointer. Before calling, the length of the array is equal to the number of pixels. After calling, the length of the array is one. This first index is the same as before, and trying to access beyond the first index reveals

[3] It is possible to create this struct also in C++ and then fill it with the Input. This was not done because this data structure is never needed in the C++ code

that the rest is completely changed. As mentioned previously (see 3.1.1), the IL2CPP compiler was used as the scripting backend. However, this destroyed the pointer. The research done in this thesis could not find an explanation for this. From this point on the Mono compiler was used as scripting backend.

CV_8UC4 to CV_8UC3: Changing Colors Unity represents its images always with RGBA, which has the type *CV_8UC4*. When trying to use the ArUco functions with an *CV_8UC4* image, ArUco states that it is only capable of using RGB (CV_8UC3) or gray-scale (CV_8UC1). To use the ArUco function, the image needs to be converted. OpenCV provides a function for this: CvtColor ([Opec]). The problem is that this function relocates the image. Thus whatever is done with it inside the C++ code will have no effect on the image C# has. It is possible to work around this by copying to the image and performing cvtColor on this copy. This copy can be used by the ArUco functions to produce the location of the corners of the markers. Because the images are copies of each other, the marker corner will be in the exact same space. Now the corners can be drawn into the original video.

Image flipping Unity and OpenCV place the first pixel of the pixel array on different parts of the screen. OpenCV places the first pixel at the top left corner and then continues to the right. For the next row, it starts at the left border again. Unity places the first pixel at the bottom left corner and then continues to the right. For the next row, it starts at the left border again. This results in the images being flipped, and thus ArUco will not recognize any markers. To solve this, the image needs to be flipped horizontally before and after the call for the wrapper function.

The dictionary used for testing was the predefined "DICT_6x6_250". For detection, the default detector parameters were used ([Oped]). The pose estimation also used the default parameters ([Oped], [Opef]). In this specific application, marker detection and pose estimation do not interact with Unity directly. They do not deviate from their normal use case. For this reason does the thesis not go into detail about them (For more information: [Oped]).

If all of these things are dealt with, then the application detects and estimates the pose of the marker accurately (3.4). The big problem here is the performance. On the device specified in the setup (3.1), the application only reaches about 3 frames per second. There is a demonstration video in the Supplementary material (7).

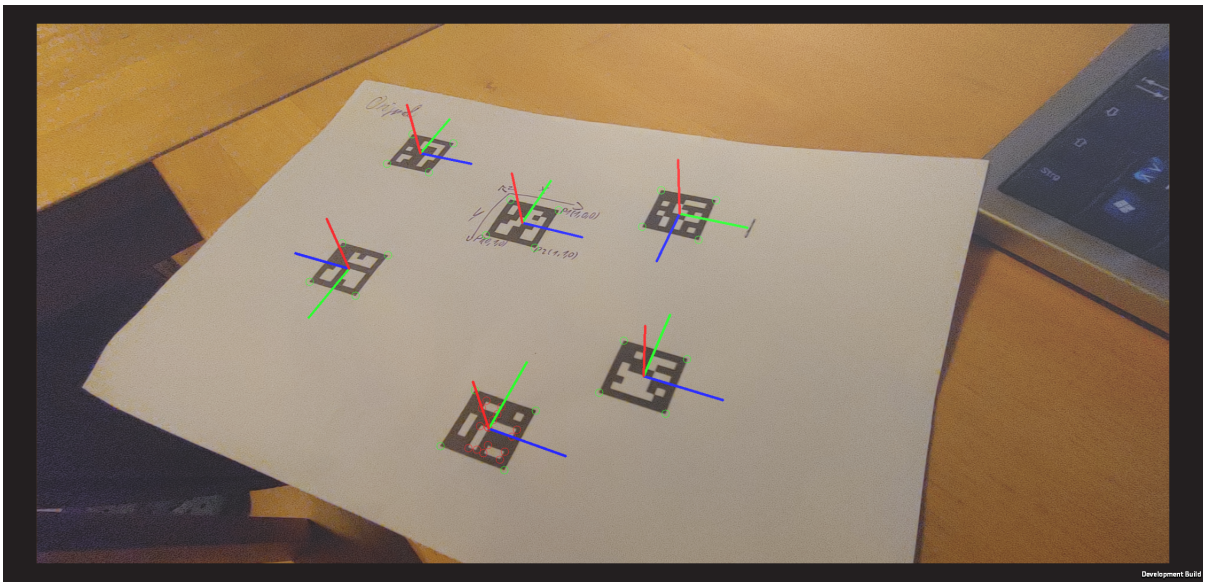

Figure 3.4: Demonstration of the ArUco detection and pose estimation

4 Conclusion

This thesis researched how to make the ArUco part of the OpenCV library accessible for Unity on an Android build. Unity uses C# for scripting, while OpenCV is written in C++. The thesis first explained the fundamentals needed to understand the procedure. These cover the basics of what the Compiler and Linker do in the C++ language, what types of C++ libraries exist, and what the ArUco marker detection is.

The procedure explains the possible ways to make external C++ accessible for Unity. Unity provides the Plug-in concept for this. There are three stages to using a Plug-in: Creating, Importing, and Calling. Plug-ins can be separated into two types: Native and Managed. The specifics of the three Plug-in stages depend on the type used. Native Plug-ins were chosen as the most promising. Native Plug-ins need a particular way to call them, which is P/Invoke and DllImport. Because different programming languages have different data types, a way is needed to convert them. This process is called Marshalling. Importing Native Plug-ins means placing them inside the "Plug-ins" folder. There are different ways to create a native Plug-in. Creating the native Plug-in as a .dll file does not work as Android is Linux-based, and DLL is a Windows concept. Unity has four ways to use C++ code for Android: C++ Source Files, Static libraries (.a files), Shared libraries (.so files), and Android Archives (.aar files). C++ Source Files are impractical for the given use case. Static libraries (.a files) do not work. Shared libraries (.so files) and Android Archives (.aar files) both work. For Shared libraries it is important that they are compiled for the architecture of the Android device. The OpenCV library can not be used as a Plug-in directly because the functions are not compiled with C linkage. To solve this the library is enclosed inside a wrapper library. Compiling the OpenCV library from the source with the aarch64-gnu toolchain creates a missing dependencies problem. E.g. the library *libc.so.6* can neither be statically linked nor be brought with. OpenCV provides a precompiled version of the library that works with Android Archives. This enables Unity to use the OpenCV library.

Unity needs to send the C++ code a pointer containing the image. The pointer will be destroyed when building the Unity project with the IL2CPP compiler as the scripting backend. The Mono compiler needs to be used. Unity and OpenCV store images differently. It must be flipped horizontally before sending the image from Unity to OpenCV.

This thesis showed that using the ArUco part of the OpenCV library is possible.

5 Further Research

This thesis is an exploration of possible concepts and examines their validity. It focuses on making the ArUco part of the OpenCV library accessible. Regarding functionality, the thesis code detects markers, estimates their poses, and draws coordinate systems onto them. The next step for the detected marker would include affecting game objects in Unity. This would involve using the rotation and translation vectors created by the code of the thesis to rotate and move objects in Unity. These vectors must be returned to the C# code for this to work. This creates the next task.

Data generated from C++ needs to be sent back to C#. Data put on the stack will be deleted after returning from the function. Allocating memory on the heap without freeing it will create a memory leak. A solution would be to allocate memory inside C#, which will be filled by C#. This way, the C# garbage collector will deal with it. The disadvantage is that it limits how many markers can be detected. Another solution could be to allocate memory in C++, send it to C#, copy it, and then send it back to C++, where it can be freed. Maybe there is a way of freeing memory in C#, which was not found during the research of this thesis.

Improving the program's performance is also crucial for its use in real-life scenarios. It stands to question whether the poor performance is due to the specific mobile device used by this thesis or if the methodology creates these performance issues.

Lastly, the ability to use the ArUco marker boards is of interest. OpenCV provides the functionality of ArUco marker boards. They are a common approach to improving the robustness of a marker system ([Gar+14]).

6 Tutorials

This chapter contains small tutorials and helpful links for specific tasks.

6.0.1 Importing a Native Plug-in

https://docs.unity3d.com/Manual/android-native-plugins-import.html

6.0.2 Building to Android with Unity

https://docs.unity3d.com/Manual/android-BuildProcess.html

6.0.3 Creating shared libraries on Linux

https://www.cprogramming.com/tutorial/shared-libraries-linux-gcc.html

6.0.4 OpenCV Cross Compilation

https://docs.opencv.org/4.x/d0/d76/tutorial_arm_crosscompile_with_cmake.html

6.0.5 Displaying the Dependencies of Dynamic Library

Executing the command "readelf –dynamic library.so" in the terminal.

6.0.6 Creating an Android Archive

This consists of two steps: Importing the OpenCV library as a module and creating the wrapper module. https://www.youtube.com/watch?v=imTTaZTSVQk between 0:00 and 3:30 explains how to import the OpenCV Library. https://www.youtube.com/watch?v=mrBsn7JDf28 between 0:00 and 3:40 explains how to create the wrapper module.

7 Supplementary Material

The final working version of the Unity project and Android Studio can be found in this GitHub repository:
https://github.com/Wookie20/BA.git

List of Figures

List of Tables

Bibliography

[Ama] R. Amadeo. *The Snapdragon 8 Gen 2 brings Wi-Fi 7, sticks with some 32-bit support*. https://arstechnica.com/gadgets/2022/11/the-snapdragon-8-gen-2-brings-wi-fi-7-sticks-with-some-32-bit-support/. Accessed: 2024-03-14.

[Ars] Arsyl_Games. *Unity DLL plugins for android*. https://forum.unity.com/threads/unity-dll-plugins-for-android.892189/. Accessed: 2024-03-7.

[Cod] CodeSmile. *DllNotFound for .dll file in Android build*. https://forum.unity.com/threads/dllnotfound-for-dll-file-in-android-build.1489696/. Accessed: 2024-03-8.

[cpp] cppreference. *C++ Standard Library*. https://en.cppreference.com/w/cpp/standard_library. Accessed: 2024-03-11.

[Cun] E. Cunningham. *Improving app security and performance on Google Play for years to come*. https://android-developers.googleblog.com/2017/12/improving-app-security-and-performance.html. Accessed: 2024-03-14.

[Dan] DanAlbert. *How to link libraries with non so endings in NDK . e.g.libc.so.6*. https://github.com/android/ndk/issues/1341. Accessed: 2024-03-12.

[Deva] A. Developer. *Application fundamentals*. https://developer.android.com/guide/components/fundamentals. Accessed: 2024-03-9.

[Devb] A. Developer. *C++ library support*. https://developer.android.com/ndk/guides/cpp-support. Accessed: 2024-03-11.

[Devc] A. Developer. *Create an Android Library*. https://developer.android.com/studio/projects/android-library. Accessed: 2024-03-12.

[Devd] A. Developer. *Reduce your app size*. https://developer.android.com/topic/performance/reduce-apk-size. Accessed: 2024-03-9.

[Deve] A. Developer. *Sign your app*. https://developer.android.com/studio/publish/app-signing. Accessed: 2024-03-12.

[Devf] A. Developer. *What do we mean by architecture?* https://developer.arm.com/documentation/102404/0201/What-do-we-mean-by-architecture-. Accessed: 2024-03-10.

[Devg] A. Developers. *Platform architecture*. https://developer.android.com/guide/platform. Accessed: 2024-03-6.

[Doca] U. Doc. *Native Plug-ins*. https://docs.unity3d.com/Manual/NativePlugins.html. Accessed: 2024-02-16.

[Docb] U. Doc. *Plug-ins*. https://docs.unity3d.com/Manual/Plugins.html. Accessed: 2024-02-16.

[Gar+14] S. Garrido-Jurado, R. Muñoz-Salinas, F. Madrid-Cuevas, and M. Marín-Jiménez. "Automatic generation and detection of highly reliable fiducial markers under occlusion." In: *Pattern Recognition, 47(6), 2280-2292* (2014).

[GNU] GNU. *Options for Linking*. https://gcc.gnu.org/onlinedocs/gcc/Link-Options.html. Accessed: 2024-03-11.

[gnu] gnu. *GCC, the GNU Compiler Collection*. https://gcc.gnu.org/. Accessed: 2024-03-11.

[Hul] D. Hulme. *Why would an APK contain DLL files?* https://android.stackexchange.com/questions/195326/why-would-an-apk-contain-dll-files. Accessed: 2024-03-8.

[jim] jimmycrazyskills. *How do I include added dll's to a android apk build?* https://discussions.unity.com/t/how-do-i-include-added-dlls-to-a-android-apk-build/133343. Accessed: 2024-03-7.

[Kof89] M. Kofler. *Linux – Installation, Konfiguration, Anwendung; 3 Auflage*. Page 240. Addison-Wesley, 1989.

[Leaa] M. Learn. *Consuming Unmanaged DLL Functions*. https://learn.microsoft.com/en-us/dotnet/framework/interop/consuming-unmanaged-dll-functions. Accessed: 2024-03-6.

[Leab] M. Learn. *Decorated names*. https://learn.microsoft.com/en-us/cpp/build/reference/decorated-names?view=msvc-170. Accessed: 2024-03-9.

[Leac] M. Learn. *DllImportAttribute Class*. https://learn.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.dllimportattribute?view=net-8.0. Accessed: 2024-03-4.

[Lead] M. Learn. *MarshalAsAttribute Class*. https://learn.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.marshalasattribute?view=net-8.0. Accessed: 2024-03-4.

[Leae] M. Learn. *Platform Invoke (P/Invoke)*. https://learn.microsoft.com/en-us/dotnet/standard/native-interop/pinvoke. Accessed: 2024-03-4.

[Leaf] M. Learn. *Structure types*. https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/struct. Accessed: 2024-03-13.

[Leag] M. Learn. *Type marshalling*. https://learn.microsoft.com/en-us/dotnet/standard/native-interop/type-marshalling. Accessed: 2024-03-4.

[llv] llvm. *libc++" C++ Standard Library*. https://libcxx.llvm.org/. Accessed: 2024-03-11.

[Mana] U. Manual. *Android Plug-ing types*. https://docs.unity3d.com/Manual/android-plugin-types.html. Accessed: 2024-03-9.

[Manb] U. Manual. *Build your application for Android*. https://docs.unity3d.com/Manual/android-BuildProcess.html. Accessed: 2024-03-12.

[Manc] U. Manual. *Call native plug-in for Android code*. https://docs.unity3d.com/Manual/android-native-plugins-call.html. Accessed: 2024-03-9.

[Mand] U. Manual. *Color32*. https://docs.unity3d.com/ScriptReference/Color32.html. Accessed: 2024-03-13.

[Mane] U. Manual. *Export an Android project*. https://docs.unity3d.com/Manual/android-export-process.html. Accessed: 2024-03-12.

[Manf] U. Manual. *How Unity builds Android applications*. https://docs.unity3d.com/Manual/how-unity-builds-android-applications.html. Accessed: 2024-03-12.

[Mang] U. Manual. *Introducing Android Library Projects and Android Archive plug-ins*. https://docs.unity3d.com/Manual/android-library-project-and-aar-plugins-introducing.html. Accessed: 2024-03-12.

[Manh] U. Manual. *Introducing native plug-ins for Android*. https://docs.unity3d.com/Manual/android-native-plugins-introducing.html. Accessed: 2024-03-9.

[Mani] U. Manual. *IPostGenerateGradleAndroidProject.OnPostGenerateGradleAndroidProject*. https://docs.unity3d.com/ScriptReference/Android.IPostGenerateGradleAndroidProje
OnPostGenerateGradleAndroidProject.html. Accessed: 2024-03-12.

[Manj] U. Manual. *IPostprocessBuildWithReport.OnPostprocessBuild*. https://docs.unity3d.com/ScriptReference/Build.IPostprocessBuildWithReport.OnPostprocessBuild.html. Accessed: 2024-03-12.

[Mank] U. Manual. *Managed code stripping*. https://docs.unity3d.com/Manual/ManagedCodeStripping.html. Accessed: 2024-03-7.

[Manl] U. Manual. *Managed plug-ins*. https://docs.unity3d.com/Manual/UsingDLL.html. Accessed: 2024-03-8.

[Manm] U. Manual. *PostProcessBuildAttribute*. https://docs.unity3d.com/ScriptReference/Callbacks.PostProcessBuildAttribute.html. Accessed: 2024-03-12.

[Mann] U. Manual. *The Unity linker*. https://docs.unity3d.com/Manual/unity-linker.html. Accessed: 2024-03-7.

[MJA14] M. Ma, L. C. Jain, and P. Anderson. *Virtual, Augmented Reality and Serious Games for Healthcare 1*. Page 457. Springer Heidelberg New York Dordrecht London, 2014.

[Opea] OpenCV. *About*. https://opencv.org/about/. Accessed: 2024-03-8.

[Opeb] OpenCV. *Cross compilation for ARM based Linux systems*. https://docs.opencv.org/4.x/d0/d76/tutorial_arm_crosscompile_with_cmake.html. Accessed: 2024-03-10.

[Opec] OpenCV. *cvtColor*. https://docs.opencv.org/3.4/d8/d01/group__imgproc__color__conversions.html#ga397ae87e1288a81d2363b61574eb8cab. Accessed: 2024-03-14.

[Oped] OpenCV. *Detection of ArUco Markers*. https://docs.opencv.org/4.9.0/d5/dae/tutorial_aruco_detection.html. Accessed: 2024-03-14.

[Opee] OpenCV. *Release*. https://opencv.org/releases/. Accessed: 2024-03-12.

[Opef] OpenCV. *solvePnP*. https://docs.opencv.org/4.9.0/d9/d0c/group__calib3d.html#ga549c2075fac14829ff4a58bc931c033d. Accessed: 2024-03-14.

[oT2] oT2. *DllNotFoundException on Android*. https://stackoverflow.com/questions/44027714/dllnotfoundexception-on-android. Accessed: 2024-03-6.

[Pac] U. Packages. *Package: g++-aarch64-linux-gnu (4:9.3.0-1ubuntu2)*. https://packages.ubuntu.com/focal/g++-aarch64-linux-gnu. Accessed: 2024-03-10.

[Pag] M. Page. *libc(7)*. https://man7.org/linux/man-pages/man7/libc.7.html. Accessed: 2024-03-11.

[pau] paulpizzi. *Using dll-Library in Unity3D for Android Build*. https://stackoverflow.com/questions/59771713/using-dll-library-in-unity3d-for-android-build. Accessed: 2024-03-7.

[Ste14] M. Stevanovic. *Advanced C and C++ Compiling*. Apress L. P, 2014.

[Świ22] R. Świdziński. *Compiling C++ Sources with CMake*. Packt Publishing, 2022.

[Sze22] R. Szeliski. *Computer Vision - Algorithms and Applications, Second Edition*. Springer International Publishing AG, 2022.

[Uni] Unix. *libm Man Page*. https://www.unix.com/man-page/linux/3lib/libm. Accessed: 2024-03-11.

[Voia] Voitanium. *Open CV for Unity Android - Face Detection*. https://www.youtube.com/watch?v=mrBsn7JDf28. Accessed: 2024-03-13.

[Voib] Voitanium. *OpenCV for Android (Java)*. https://www.youtube.com/watch?v=imTTaZTSVQk&t=147s. Accessed: 2024-03-13.

[Wik] Wikipedia. *Name Mangling*. https://en.wikipedia.org/wiki/Name_mangling. Accessed: 2024-01-29.

[Wil] P. Williamson. *Pushing the Boundaries of Performance and Security to Unleash the Power of 64-bit Computing*. https://newsroom.arm.com/news/pushing-the-boundaries-of-performance-and-security-to-unleash-the-power-of-64-bit-computing. Accessed: 2024-03-14.