SACJ 33(1) July 2021
Research Article

Architecture and Architectural Patterns for Mobile Augmented Reality

D. Munro, A.P. Calitz, D. Vogts

Department of Computing Sciences, Nelson Mandela University, Gqeberha, South Africa

ABSTRACT
A software architecture codifies the design choices of software developers, which defines a modular organizational spine for the design of a software artefact. Different architectures may be specified for different types of artefacts, a real-time interactive artefact, for example, would have markedly different requirements to those of a batch based transactional system. The use of software architecture becomes increasingly important as the complexity of artefacts increases.

Augmented Reality blends the real world observed through a computer interface, with a computer generated virtual world. With the advent of powerful mobile devices, Mobile Augmented Reality (MAR) applications have become increasingly feasible, however the increased power has led to increased complexity. Most MAR research has been directed towards technologies and not design resulting in a dearth of architecture and design literature for MAR. This research is targeted at addressing this void.

The main requirement that a MAR architecture must meet is identified as being the efficient real-time processing of data streams such as video frames and sensor data. A set of highly parallelised architectural patterns are documented within the context of MAR that meet this requirement. The contribution of this research is a software architecture, codified as architectural patterns, for MAR.

Keywords: Architectural Patterns, Software Architecture, Augmented Reality

Categories: • **Software and its Engineering** ∼ **Designing software** • *Human-centered computing* ∼ *Ubiquitous and mobile computing*

Email:
D. Munro donaldmunro@gmail.com (CORRESPONDING),
A.P. Calitz Andre.Calitz@mandela.ac.za,
D. Vogts Dieter.Vogts@mandela.ac.za

Article history:
Received: 27 Oct 2020
Accepted: 16 May 2021
Available online: 12 July 2021

1 INTRODUCTION

Augmented Reality (AR) blends the real world, as viewed through a computer interface, with a virtual world generated by a computer, forming part of a larger set of *Mixed Realities*, described by Milgram and Kishino (1994) as a "Virtuality Continuum" (Figure 1). The related mixed reality technologies comprise Virtual Reality (VR) which provides an interactive view of an entirely virtual world and Augmented Virtuality (AV), projecting real-world objects into a virtual world.

Munro, D., Calitz, A. and Vogts, D. (2021). Architecture and Architectural Patterns for Mobile Augmented Reality. *South African Computer Journal 33*(1), 59–78. https://doi.org/10.18489/sacj.v33i1.908

SACJ is a publication of the South African Institute of Computer Scientists and Information Technologists. ISSN 1015-7999 (print) ISSN 2313-7835 (online).

The last decade has seen the rapid evolution of mobile devices incorporating multi-core Central Processing Units (CPUs), Graphics Processing Units (GPUs) and high definition displays coupled with sophisticated cameras and sensors capable of detecting position, orientation, motion as well as environmental variables, such as light and temperature. The increasing complexity of mobile devices has also resulted in the development of Operating Systems, such as Android and iOS, to control these mobile devices. The combination of hardware and system software providing access to the hardware capabilities, has enabled complex MAR applications to become not only feasible, but, given the ubiquity of mobile devices in the modern world, inevitable. Research into MAR software has proved to be a fertile new field for researchers, with copious research covering highly technical areas, such as Computer Vision for object detection and tracking, 3D graphics for realistic rendering of virtual content and probabilistic filtering and Simultaneous Localisation and Mapping (SLAM) for coordinating position and orientation between the real and virtual worlds (Billinghurst et al., 2015).

The above-mentioned research has also led to a sudden increase in interest relating to MAR, which in turn has led to the eventual commodification of MAR. Two of the larger players in the mobile device marketplace have released MAR SDKs with Google's ARCore and Apple's ARKit. MAR games have also popularised MAR in the popular software marketplace with games such as Pokémon Go and Ingress Prime introducing a wider general audience to MAR.

As is often the case with new fields in Computer Science (CS), most of the research has concentrated on solving highly technical issues, with far less being devoted to software design. As a field in CS matures, more attention needs to be bestowed on architecture and design, as the overall complexity of system implementation within the field increases, making ad-hoc implementations difficult. This is particularly the case when the maturing field becomes more accessible to a less specialised audience, who require more guidance when developing systems within the field. In MAR development, this is especially the case as the research provides a plethora of different solutions for similar or closely-related problems, documented mostly in relatively obscure academic papers, which are not always accessible to a less specialised audience.

The first step on the path towards a coherent software design is the software architecture, which provides the foundation on which the rest of the design is built. Software architecture, as discussed in Section 3, is concerned with formalising an underlying foundation on which a software artefact can be built. This entails the partition of the artefact into related components

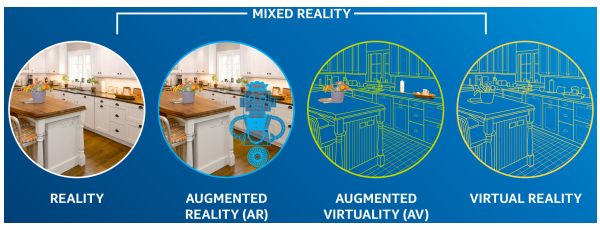


Figure 1: Mixed Reality Continuum (Milgram & Kishino, 1994). Image adapted from Valoriani (2016)

https://doi.org/10.18489/sacj.v33i1.908

based on the artefact requirements and the context in which the artefact is deployed.

The objective of this research is to provide such an architecture for MAR applications. The research involves design of an architecture which resulted in the methodology of Design Science Research (DSR) as applied to conceptual meta-artefacts, as described by Iivari (2015) being used. The methodology was combined with the research process of Nunamaker et al. (1990). In the context of the research strategies proposed by Iivari, the combined architectural patterns are a conceptual IT meta-artefact, which can be used to design and implement concrete artefacts. The sample application designed and implemented in the evaluatory section (Section 6) is an example of a concrete artefact designed using the architectural patterns.

In order to provide some theoretical background, a brief overview of the differing types of MAR will be provided in Section 2, followed by an overview of the concepts of software architecture and architectural patterns in Section 3. Related work on the design of MAR software will be described in Section 4. Section 5 will document a proposed MAR architecture, while Section 6 will present an open source realisation of the architecture as an Android application. Finally, Section 7 will provide conclusions which can be drawn from the study, and identify future research areas.

2 MOBILE AUGMENTED REALITY CLASSIFICATION

MAR may be classified in several ways. From a hardware perspective, MAR may be implemented on a standard hand-held device, such as a smartphone or tablet with a video feed provided by a camera. Another variety of MAR is provided on wearable devices epitomised by Google Glass (2020), Microsoft HoloLens (2020) and the HTC Vive Pro (2020), although these wearable devices differ in the way they present the real-world with Google Glass and Microsoft HoloLens using see-through lenses onto which virtual content is stereoscopically projected,[1] while the HTC Vive resembles a hand-held device in that it utilises two displays in the headset to render output from two head mounted cameras.

The software implementing MAR applications can be further classified into:

- Computer Vision (CV) based applications, which use computer vision based techniques for object detection, tracking and pose determination.[2] Early CV systems used special artificial markers, known as fiducial markers for object detection and tracking. Recent and more sophisticated CV systems may also utilise SLAM to detect and redetect features, in order to dynamically build a map of the surroundings as the user moves through the world; and
- Locational MAR applications, which use device sensors such as gyroscopes, accelerometers and magnetometers combined with the use of a Global Positioning System (GPS) for outdoor localisation and WiFi/Bluetooth beacons for indoor localisation. Such applications cannot directly detect objects to augment, instead they use the location and pose to

[1] Although the Hololens uses several cameras for head and eye tracking

[2] Finding the position and orientation of the device in a common coordinate system

https://doi.org/10.18489/sacj.v33i1.908

determine when and where to display virtual content. Precise pose determination can be challenging on such sensor only systems, as the sensors are prone to noise and other errors with gyroscope drift being common and magnetometers being known for sensitivity to even small local magnetic fields, such as wrist watches. However applications which do not require precise positioning can utilise locational AR successfully, usually by using a technique known as sensor fusion to combine sensors in order to reduce errors: the popular MAR game *Pokémon Go* is an example of such an application; and

- Hybrid systems which combine CV and sensors using CV for detection/tracking and sensors combined with CV for pose determination and localisation. SLAM systems in particular are often implemented using this approach.

The main emphasis of this research is MAR implemented on a hand-held device, but most of the architectural requirements underlying the proposed design also apply to alternate types of MAR. The following section will briefly discuss architectural patterns, before Section 5 describes the application of an architectures pattern for the proposed design of an MAR architecture.

3 SOFTWARE ARCHITECTURE AND ARCHITECTURAL PATTERNS

A definition of software architecture is provided by IEEE standard 1471–2000 (Maier et al., 2001) indicating that

> The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.

An alternate definition that highlights the link between design, particularly high-level design decisions and architecture, is provided by Booch (2008):

> Architecture is design but not all design is architecture. Architecture represents the significant design decisions that shape a system, where significant is measured by cost of change.

The latter definition captures the effect of architecture on the longevity of the design, because the quality of the architecture determines the maintainability and extensibility of the software and the related costs. The costs of designing without any coherent architecture is described by Foote and Yoder (1999), in which the term describing such systems as “Big balls of mud” was first coined.

According to Booch et al. (2007), the attributes that software architectures should possess include:

- *Modularity* which entails representing the system as multiple separate modules. These modules may in turn consist of sub-modules or components;

- High *cohesivity* for components of a module meaning that the components have a common task or goal and provide interfaces for these tasks;
- Low *coupling* where modules have no unnecessary dependencies on other modules;
- *Encapsulation* of functionality by using defined interfaces for accessing modules or components; and
- *Abstraction* of the functionality of the system by using encapsulation to represent all functionality provided by the system.

An architecture describes design decisions, therefore documenting these decisions is a requirement. Software patterns provide a convenient tool for describing designs and design decisions (Ton That et al., 2012). Software design patterns emerged in the mid 1990's and were popularised in the software design community by the "Gang of Four" (Gamma et al., 1995), who in turn were inspired by earlier work by Alexander (1979), describing "real brick and mortar" architectural designs as interlinked patterns. Software patterns document recurring design problems in terms of the context in which they occur, constraints (also known as forces) that affect the design and tried and trusted solutions described as a solution space instead of a specific solution blueprint. The designer may then customise the solution to best fit the particular circumstance, with the customisation often involving the assignment of roles in the pattern to software components.

While software patterns are most frequently associated with solving lower level single issue problems, they have increasingly become popular as an architectural level design aid. Many modern software architectures are expressed as patterns, for example, LAYERED, CLIENT-SERVER, PIPES AND FILTERS and BLACKBOARD are well known and used patterns (Buschmann et al., 1996). These patterns are known as architectural patterns to distinguish them from tactical level design patterns, which target individual low-level problems. By documenting the application of architectural patterns to design a system, a record of the design decisions is created, which can be referred to in later phases of the system life cycle. It should also be noted that while patterns have become synonymous with Object-Oriented (OO) design, in reality both tactical and architectural patterns can be used in non-OO design, for example, the Linux Kernel documents several Operating System level patterns (Guy & Agopian, 2018a).

In the next section, related work and software architectures that have been applied to MAR are described. The architecture proposed by this research is then presented in Section 5.

4 RELATED WORK

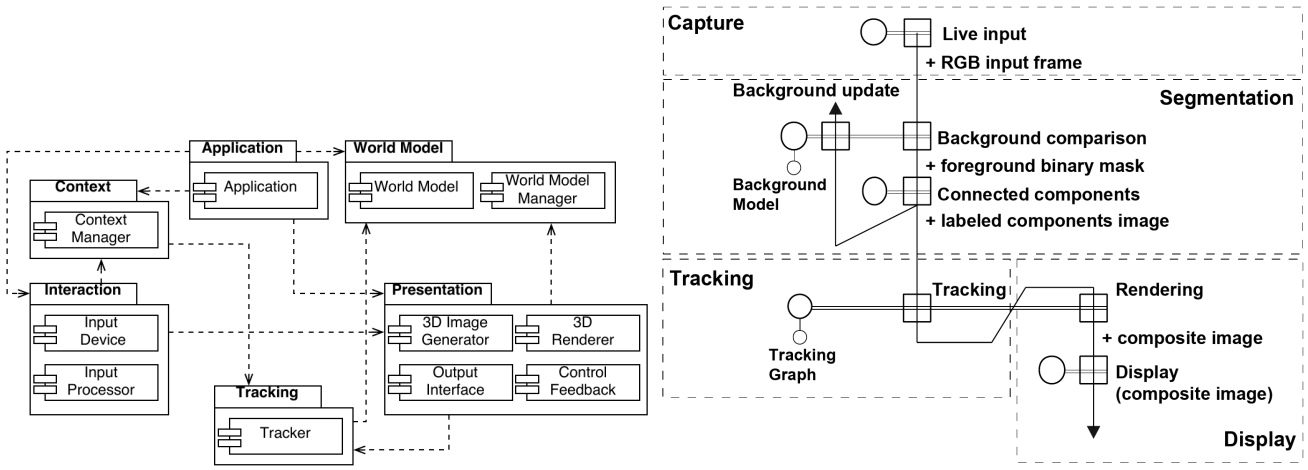
While there is much research describing the implementation of AR and MAR artefacts, most of them concentrate on demonstrating the use of AR for a particular application, for example, as a tourist guide or for surgical training or support, with very little or no description of any architecture or design.

Research that does briefly discuss this area tend to follow the design of Reicher (2004).

Reicher describes a Model View Controller (MVC[3]) (Krasner & Pope, 1988) based architecture comprising of several modules (Figure 2a), namely:

- **Application:** Provides a high-level interface encapsulating the rest of the modules and components;
- **Interaction:** Deals with user interaction, which would usually be seen to belong to the Controller part of MVC. For mobile device-based MAR, this primarily encompasses touch screen use, but voice-based input could also be handled;
- **Context:** Stores and shares the common information. The Context module is accessed by modules requiring common information and is in turn updated by modules producing the information;
- **Presentation:** Provides the view part in the MVC pattern rendering the augmented output. In order to do so, it needs to access other modules to obtain information required for rendering, for example, the pose can be obtained from the context module;
- **Tracking:** Handles object detection, tracking and pose updates using the camera and /or sensor data. The information is disseminated to the Context module for use by other modules; and
- **World Model:** Defines the world coordinate system.

MacWilliams et al. (2004) also documents the patterns used in the above design.



(a) MVC-based MAR high-level architecture (MacWilliams et al., 2004; Reicher, 2004).

(b) Video segmentation and tracking architecture using SAI (François, 2003).

Figure 2: Prior work in AR architectures.

A generic architecture for multimedia applications was proposed by François (2003), which can be adapted for use in a MAR setting. The architecture is known as SAI (Software Architec-

[3]MVC defines an architecture for designing interactive applications, comprising of three main components: the Model which is responsible for the state of a displayable logical entity, the View displaying a representation of the model and the Controller responding to user interaction and updating the model.

https://doi.org/10.18489/sacj.v33i1.908

ture for Immersipresence), and is based on three components, namely cells, sources and pulses. Cells are arranged in pipelines and process incoming data and forward the result to the next Cell. Data can be persistent, for example, constant data such as settings or volatile, such as video or audio frames. Persistent data are held in Sources connected to specific Cells containing persistent data for the Cell. While sources hold constant data, pulses transmit volatile data periodically. The pulse combined with a timestamp, is propagated through the pipeline of cells and each cell can extract and operate on the data in the pulse. The cell can also modify the volatile data before allowing the pulse to continue to the next cell. The pulse data itself is passed as a pointer (in process) or token (out-of-process) to avoid expensive data copying operations on memory-heavy content, such as video frames. Cell generated pulses are also used to communicate persistent data from sources to cells, these are known as active pulses.

Figure 2b illustrates a simple SAI architecture for video segmentation. Squares represent cells, while circles represent sources. Passive pulses carrying volatile data travel along single line connectors, while active pulses allowing cells to access persistent data in sources are represented by double lines. Text with a + in front denote a cell addition to the pulse. There is a feedback loop in the second Segmentation cell, which splits the pulse in two with the feedback part of the pulse updating a statistical background model used during segmentation.

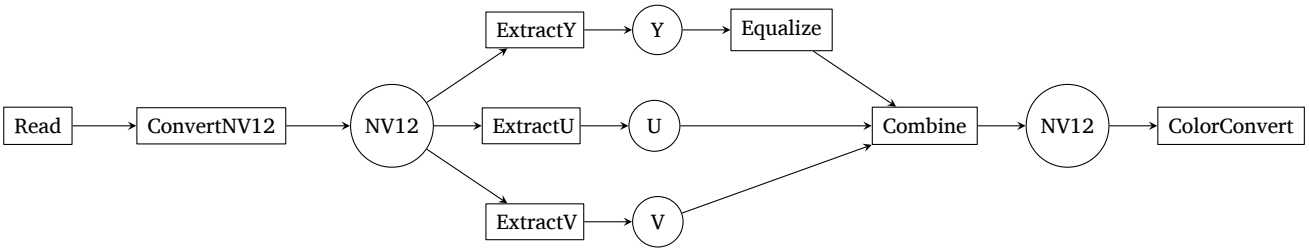


Figure 3: OpenVX graph

While not specifically targeted at MAR, OpenVX (Khronos, 2015), which is an open standard for providing acceleration for CV applications, also utilises the concept of linking data streams. OpenVX defines a set of CV functions, which are combined as nodes in a graph with data streams forming the edges between the nodes. An example of a graph diagram for a small OpenVX fragment for converting image formats is shown in Figure 3.

The architecture proposed by this research study applies to MAR artefacts that are self contained on a device. An alternate approach proposed in the early to mid 2000's, before mobile device processing power advanced to their current levels, is to move the CPU intensive tasks to more powerful cloud servers utilising a distributed system paradigm (Bauer et al., 2001; Huang et al., 2014; Piekarski & Thomas, 2001). Such an approach however, further magnifies the potential latency issues as video frames and sensor data need to be transferred to cloud servers for processing.[4] Ren et al. (2019) propose modernising this type of architecture

[4]Alternately some initial pre-processing such as feature extraction can be done locally with the preprocessed results (image features for example) transmitted to the server instead of the entire frame.

https://doi.org/10.18489/sacj.v33i1.908

by adding an extra "edge layer" situated between the device and the cloud server. The edge layers are located at the base stations of cellular networks or WiFi network access points and can therefore provide somewhat faster processing as less network hops are required, however such an approach is still dependent on network access speed and developers having access to such edge layers. It should be noted that the same requirements applicable to on-device architectures also apply to cloud-based architectures, so the architectural patterns proposed in Section 5 are also applicable to cloud or edge-based servers in conjunction with distributed system architectural patterns.

5 PROPOSED SOFTWARE ARCHITECTURE FOR MAR

The architecture of a MAR system must reflect the primary requirements and constraints that pertain to the design of the system. The user's perception of a MAR system is almost entirely dependent on how responsive it is, thus the primary requirement for MAR is the processing of streams of video and sensor data in real-time, therefore the choice of an architecture optimal for processing data streams in real-time is essential.

Streams of data are traditionally processed using the PIPES AND FILTERS[5] architectural pattern (Buschmann et al., 1996). Filters encapsulate processing steps and filters can execute concurrently with other filters, when data are available. The data are passed through pipes between filters. The pipes are typically implemented as queues. Synchronisation is effected by blocking on the queue when the pipe is empty or full. Filters can also provide incremental output to the next filter to maximise throughput. Examples of pipeline architectures include the 3D graphics rendering and compiler parsing pipelines.

A recent alternative to PIPES AND FILTERS is the PARALLEL PIPELINE pattern (McCool et al., 2012), which also supports fully parallel filters that can process more than one data item from the stream at a time. Implementation for this pattern is more complicated and must allow for out-of-sequence data.

The Pipes and Filters architecture is a good fit for traditional streamed data applications, such as 3D graphics or compilers, however it has some drawbacks when processing real-time or interactive streams, namely:

- Data are passed as raw bytes between filters resulting in both speed and storage overheads;
- Large architectures having multiple dependent pipelines cannot synchronise the output from different pipelines at logical barrier points where the pipelines merge;
- When two stages can act on the same data independently there is unnecessary latency, while the first stage processes the data. For example, if P1, P2 and P3 are stages with P2 and P3 dependent on P1 but not on each other:
  $\boxed{P1} \rightarrow \boxed{P2} \rightarrow \boxed{P3} \rightarrow \boxed{P4}$

[5]The notational convention of using SMALL CAPS to denote patterns follows from the convention used in Buschmann et al. (2007a, 2007b).

https://doi.org/10.18489/sacj.v33i1.908

then P3 has to wait until either P2 completes, or if P2 supports incremental output then at least until after P2 has started producing output. If P4 is only dependent on P2, then it too has to wait for P3; and

- There is no feedback support, so the output from a filter cannot change the processing done by another, for example, user interaction cannot modify filter behaviour.

For interactive real-time architectures, moving away from a strict pipeline topology would appear to provide a solution. The pipeline can be replaced with a Directed Acyclic Graph (DAG), with the nodes of the graph corresponding to filters and the edges to pipes. This DAG based solution provides a more flexible alternative, while retaining the underlying chained structure of a pipeline, as DAGs can be decomposed into chains of nodes (Chen, 2007).

The TASK GRAPH pattern (Miller, 2010), models multiple atomic tasks with dependencies between the tasks as a DAG, with nodes representing tasks, while edges represent dependencies and predecessor output. The graph can be defined either at compile time or, as a variation, at runtime and is documented as a computational pattern as opposed to an architectural one. For the purposes of providing an architecture for MAR, an architectural variant of this pattern is documented in Subsection 5.2.1. As previously mentioned, the data streams involved in MAR can comprise large quantities of data, which would be slow to move between tasks and would use lots of memory, which may be in short supply on mobile devices that lack virtual memory capability. An architectural specialisation of the SHARED RESOURCE pattern (Ortega-Arjona, 2003) can be combined with the architectural TASK GRAPH pattern to create a base software architecture for MAR. This version of the SHARED RESOURCE is documented in Subsection 5.2.2.

This base architecture does not preclude other MAR modules from having module specific sub-architectures interacting with the task components through predefined interfaces. For example, the rendering/interaction module may choose to use a MVC related architecture, with the MVC components interacting with task nodes to obtain or update the system state.

As previously mentioned in Section 2, MAR can be classified by hardware type, that is, hand-held or wearable and by the use of CV techniques versus location sensors for pose and localisation. The architecture proposed applies directly in the wearable case as these devices also use camera streams (in the case of the Hololens (2020) there are 6 cameras and a time-of-flight depth sensor), in addition to an accelerometer, gyroscope and magnetometer. Locational MAR does not rely on camera video streams, however the sensors provide streams of data at high frequency, thus also necessitating efficient stream processing, although the size of the stream content is much smaller.

While patterns concern themselves with design level decisions, some discussion of possible implementation approaches and technologies may be useful as examples. This discussion is especially relevant here, as implementing task graphs efficiently depends on being able to utilise a high degree of parallelism. Subsection 5.1 will briefly describe some possible implementation technologies and provide a simple example illustrating a particular implementation.

https://doi.org/10.18489/sacj.v33i1.908

5.1 Implementation Tools

Historically, parallel programming used low level software threads that were mapped onto hardware threads by either the OS or a threading package using pre-emptive time slicing. A modern high-level alternative is to define potentially parallel units of execution as tasks and let a task scheduler determine optimal task-to-thread allocation, based on the current status of available processors. The tasks themselves can then be implemented as lightweight non-pre-emptive threads or fibres, thereby avoiding the expense of pre-empting hardware threads. Several of the technologies described in this Subsection use this approach, however it is entirely possible to implement a task graph using old fashioned threads. It will however be both more complex and, more often than not, sub-optimal.

Threading Building Blocks (TBB) (Voss et al., 2019) is an open source C++ task-based template library for parallel programming using multi core CPUs. TBB has also been ported to mobile OSs such as Android and iOS. TBB directly supports task graphs through its flow graph template class. Several predefined node types are supported (Intel TBB, 2014), while new node types can also be created by composition and inheritance of the predefined node types. Edges between nodes can be specified with a concurrency limit, which designates the number of tasks the node can run in parallel.

For an example of a task graph architecture, see Figure 5 in Section 6. An excerpt from the code to create the task graph illustrated in Figure 5 using TBB is listed below:

```
source_node<uintptr_t> backSrc{graph, backSourceNode, false};
source_node<uintptr_t> frontSrc{graph, frontSourceNode, false};
function_node<uint64_t, uint64_t, rejecting> frontDetect
 {graph, 1, [this] (uint64_t seqno) -> uint64_t
   { return (*frontDetectorNode)(seqno); } };
function_node<uint64_t, uint64_t, rejecting> frontTrack
  {graph, 1, [this] (uint64_t seqno) -> uint64_t
    { return (*frontTrackerNode)(seqno); } };
function_node<uint64_t, uint64_t, rejecting> backDetect
  {graph, 1, [this] (uint64_t seqno) -> uint64_t
    { return (*backDetectorNode)(seqno); } };
function_node<uint64_t, uint64_t, rejecting> backTrack
  {graph, 1, [this] (uint64_t seqno) -> uint64_t
    { return (*backTrackerNode)(seqno); } };
multifunction_node<uintptr_t,RouterOutputTuple> rearRoute
  {graph,1,rearRouterNode};
multifunction_node<uintptr_t,RouterOutputTuple> frontRoute
  {graph,1,frontRouterNode};
function_node<uint64_t, uint64_t, rejecting> render{graph, 1,
  [this] (uint64_t seqno) -> uint64_t { return (*renderNode)(seqno); } };

make_edge(backSrc, rearRoute );
make_edge(frontSrc, frontRoute );
```

https://doi.org/10.18489/sacj.v33i1.908

```
make_edge(output_port<0>(rearRoute), backDetect);
make_edge(output_port<1>(rearRoute), backTrack);
make_edge(output_port<2>(rearRoute), render);
make_edge(output_port<0>(frontRoute), frontDetect);
make_edge(output_port<1>(frontRoute), frontTrack);
```

FastFlow (Aldinucci et al., 2017) shares many features with TBB, particularly in supporting a task-based model for data streaming and data parallelism. FastFlow differs somewhat from TBB in task graphs creation as it supports this through combining tasks into task farms or by pipeline composition.

5.2 Architectural Patterns

The proposed architectural patterns for MAR, namely the ARCHITECTURAL TASK GRAPH and SHARED RESOURCE patterns are documented in this Subsection. The context, problem and solution are specialised for use in an architectural MAR setting. The architectural patterns described here can also be integrated into a larger Pattern Language (PL), as described by Munro (2020).

5.2.1 The Architectural Task Graph Pattern

Context: Defining an architecture for MAR capable of processing multiple data streams in real-time.

Problem: The chief requirements for MAR are real-time interactivity and augmentation that is in both spatial and temporal registration with the physical world. The realisation of these requirements are dependent on rapid and efficient processing of streams of data, such as video frames and sensor readings. The streams are usually periodic and high frequency, and as in the case of video frames, may also comprise large data items comprising many bytes. The streams are usually processed by different components, which extract information from the streams or transform the data from the stream. Some components depend on the output of other components, either as information extracted from the stream or changes made to the data. The probable existence of multiple data streams, combined with the possibility that the output from some MAR components may be split into inputs for several descendent components and later recombined means a simple pipeline is not practicable.

Solution: Structure the design components responsible for handling and transforming data streams into tasks and place them in a DAG, where the nodes are tasks and the edges are data stream dependencies along which the data stream can flow. The component lifetimes are assumed to match that of the artefact being designed, so the graph exists for the lifetime of the artefact and defines its architecture, with the tasks being architectural components.

As streamed data packets can be large, for example video frames, it is undesirable and inefficient to repeatedly copy the data between nodes. A better solution is to combine this pattern with the SHARED RESOURCE pattern (See Subsection 5.2.2) and pass a key or token

representing the data between the nodes while gaining access to the actual data through a shared resource object or method.

The technology being implemented (as described in Subsection 5.1) should support task parallelism and the components should be designed so as to be able to support processing the next item in the stream, as soon as the current one is complete in order to be able to maximise throughput. Implementing technologies that use task-based scheduling are preferred to those that require low-level threading. If low-level threading is used, then care must be taken in assigning logical threads in components to physical hardware threads to avoid over-subscription of logical to physical threads, which leads to inefficient time sliced scheduling amongst the logical threads.

It is also possible to combine other forms of parallelism into the graph. For example, a task node may utilise a scan or map-reduce pattern or execute an OpenVX graph within the component. To fully utilise available parallelism, the component could combine the use of multiple GPU cores with a CPU core (the combination of CPU and GPU processing is known as heterogeneous computing).

Nodes and edges are the structural elements with nodes representing tasks and edges dependencies with data flow. While the functional task is the primary node in the DAG, various other node types exist in various implementations, or can be implemented using lower level primitives. These nodes provide a means to synchronise and buffer graph flow. Some examples include:

- Data generation and processing nodes, such as source nodes (for example a node that obtains a camera image), functional task nodes, multifunctional task nodes that can output results to multiple descendent nodes selectively and broadcast nodes that copy their input to multiple descendants;
- Buffering nodes, including standard buffer, ring buffer, queue, priority queue and sequencer nodes, which resynchronise output based on an incoming sequence number; and
- Synchronisation nodes, such as a join node that receives input from multiple nodes but only transmits to its successor when all input nodes have data available and the limiter node that stops outputting data when a given count is reached until the counter is reset.

The critical path through the DAG, which is determined by the time taken by tasks in paths in the graph determines the maximal throughput. It is possible to work around the limitations of slower components by allowing them to “skip turns”, that is not process for every periodic iteration, such as video frames. For example, if it is known that an object detector is slower than other components, then the detector node can be set to have a parallelism of two and have one thread doing the detection, while the second one simply passes the received token on to the next task until the main thread completes its current detection. The obvious disadvantage is that the detection may be several frames behind the other components, but this is frequently acceptable given that inter-frame changes are normally quite minimal.

https://doi.org/10.18489/sacj.v33i1.908

5.2.2 The Shared Resource Pattern

Context:

- When defining a STRUCTURAL TASK GRAPH, the size of the data that will have to be passed between task nodes is too large to be efficiently copied; and
- When results computed from data streams need to be shared amongst all nodes.

Problem:

- The default means of communicating data between nodes in a STRUCTURAL TASK GRAPH is by message passing between nodes, however multimedia data such as images tend to consume large amounts of memory space and are slow and inefficient to copy when implementing task graph message passing; and
- When the results computed using the streams by one task should be available to other nodes in the graph, regardless of their position in the graph relative to the node that computed the result.

Solution: Use the SHARED RESOURCE pattern (Ortega-Arjona, 2003) to provide shared data access to components acting as nodes in a STRUCTURAL TASK GRAPH. Data consistency is enforced by the shared resource component, while ensuring correct sequencing is left either to the nodes or the design. For example, if a task updates a frame and there is a possibility of out-of-sequence access causing inconsistencies, the designer could specify that the updated frame should be stored separately from the original.

When using SHARED RESOURCE in conjunction with STRUCTURAL TASK GRAPH, nodes from the task graph play the role of the sharer components, while the shared resource object or API plays the role of the SHAREDRESOURCE in the pattern as described by Ortega-Arjona (2003). In order to adhere to the principle of programming to an interface, the SHAREDRESOURCE component should export a standardised EXTERNAL INTERFACE (Buschmann et al., 2007b), which would support pluggability and maintainability and minimise dependencies.

The locking that may be necessary to ensure data consistency will impact performance and could even be the source of bugs. In some cases lock-free algorithms for some data structures, such as queues may lead to improved performance although performance gains for lock-free structures are not guaranteed and require real-life benchmarking.

6 EVALUATION

In order to evaluate the architecture, an Android application was designed using the architectural patterns defined in Subsection 5.2 and implemented mainly in C++ using the Android NDK. The application source is available at https://github.com/donaldmunro/ARArch.

As mentioned in Section 5, processing data streams, especially video streams in real-time is the prime requirement, therefore the application was designed to support video streams from multiple cameras. For example, Android devices having multiple cameras have recently become increasingly common, although not all devices allow individual addressing of the cameras using the Android camera API.

https://doi.org/10.18489/sacj.v33i1.908

In addition to the video streams, the application also provides sensor data streaming from accelerometers, gyroscopes and fused gravity/rotation sensors provided by devices through the Android sensor API. These sensor streams are less demanding as the size of the data is much smaller than video frames, however they typically occur at high frequency when used in a MAR setting, where orientation updates need to be real-time. An example of a MAR requirement for real-time sensor data streams is Visual-Inertial SLAM (Chang et al., 2018), where Computer Vision (CV) is fused with sensor data to map the environment in real-time.

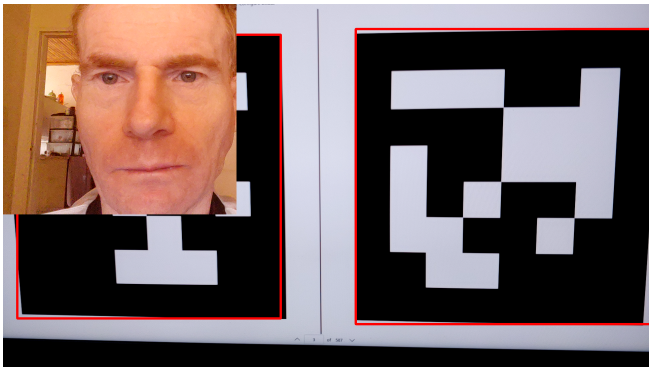


Figure 4: Sample application performing facial recognition and April Tag detection on front and rear video streams.

The primary goal of the application is the evaluation and benchmarking of a MAR architecture so it does not implement a full AR application, however, in order to simulate the kind of tasks performed by an AR application, some CV tasks are included. In particular, an April Tags (Wang & Olson, 2016) detector is used to detect April Tags, (a modern version of the fiducial markers used in early AR) in the rear camera(s) feeds, while facial detection is implemented for the front camera. For rear cameras, the April Tag bounding boxes (BB) are rendered, for front-only camera streams, detected faces are rendered with a BB whilst processing both rear and front video streams, a face detected using the front camera is overlayed over the rear camera frame, which is rendered with BBs for detected April Tags (Figure 4). The device tested was a LG G7 ThinQ, which has two individually addressable rear cameras and a single front camera. The G7 is a relatively modern (July 2018) device, which was high-end, however not quite flagship level when it was introduced. As indicated in the benchmark, it is able to maintain close to maximal frame rates across all benchmarks when using this architecture, the most up-to-date high-end devices should be able to produce even better results.

The overall architectures are defined by the ARCHITECTURAL TASK GRAPH pattern described in Subsection 5.2.1, combined with the SHARED RESOURCE pattern (Subsection 5.2.2) (for the rest of this section the SHARED RESOURCE realised in the design will be referred to as the repository). The application defines several test architectures, including a monocular architecture for a single rear or front camera, a stereo version for dual rear cameras, a dual monocular version for a single rear and front camera and finally, a rear stereo for two cameras and a monocular front camera. Unfortunately, the final configuration could not be tested

https://doi.org/10.18489/sacj.v33i1.908

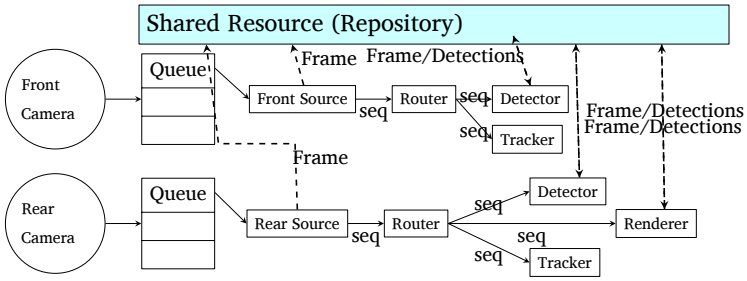


Figure 5: An example architecture for separate detector/trackers for rear and front video streams.

as the test device did not seem to support more than two simultaneous video streams at the hardware level.[6]

An example architecture for the case of single rear and front cameras is illustrated in Figure 5. The circular nodes on the left specify the Android camera handling code, which is currently implemented in Kotlin[7] combined with RenderScript[8] (or optionally, CPU) based conversion from the hardware native YUV video format to RGB required for CV-based processing (see Munro (2020) for a more in depth description of utilising Renderscript for YUV conversion documented as an Android specific idiom).

The frames from the camera handler are placed in an NDK level queue using Java Native Interface (JNI) to interface with the C++ code. The source nodes are the first part of the task graph and are responsible for removing frames from the queues, combining the frames with identification information and placing them in a collection maintained by the repository. For stereo configurations, the camera source nodes feed into a join node which marks the incoming frames as stereo combinations in the repository. The source nodes (or join node in the stereo case) pass the frame information (but not the data) to the router node, which routes the frame information on to other nodes based on configuration information and current destination node state. For example, detections may take some time so the detector should not receive any further frames until it completes the current detection.

The detector nodes perform the April Tag or facial detection by using the frame sequence to retrieve the frame data from the repository, performing the detection and then updating detection information in the repository. For stereo operations, detection is performed on both frames. In the current implementation, the tracker node is a no-operation node, but in a full AR application, it would be used to perform tracking, which is usually assumed to be faster and less resource intensive than detection. Finally the renderer node combines frame data and detection information to render the output using a concrete renderer implementation, which

[6]It was possible to configure the cameras, but initiating the video streams resulted in hardware abstraction layer (HAL) errors in the Android system logs.

[7]Recent updates to the NDK do include some C++ camera support, but the support is not as yet fully comprehensive

[8]An Android specific GPGPU API providing some of the functionality provided by GPU APIS such as OpenCL or CUDA

https://doi.org/10.18489/sacj.v33i1.908

Architecture	FPS
Monocular Rear	30.147
Monocular Front	30.135
Stereo Rear	30.064
Front & Rear Monocular	30.061

Table 1: Benchmark frame rates for different architectures (averaged over 10 runs).

implements an abstract renderer interface.

The test implementation uses a Vulkan-based renderer as Vulkan allows a fully multi-threaded approach, as opposed to OpenGL, where all rendering must be done on the main thread. However OpenGL rendering is also supported by placing rendering information in a queue which can then be rendered on the main thread. The Vulkan renderer is a simple implementation, which is all that is necessary for the test application, but the architectural framework can also support scenegraph style rendering using Google's Filament 3D framework (Guy & Agopian, 2018b) for more advanced material-based rendering (See https://github.com/donaldmunro/Bulb for a C++ scenegraph adaption for Filament that can be used with the architectural framework).

The application was benchmarked using the maximum standard frame rate setting of 30fps of the Android Camera2 API, unfortunately the newer high speed *CameraConstrainedHighSpeedCaptureSession* mode appears to only support direct output to a device display surface and not programmatic capture. All benchmarks were also performed at the maximum resolution of 1920×1080, with the Monocular Rear and Stereo rear also performing April Tag detection, the Monocular Front performing face detection and the Front & Rear Monocular performing both types of detection. The results are summarised in Table 1. The performance scaled well across multiple video streams maintaining consistently high frame rates while performing CV detection tasks.

7 CONCLUSIONS

Designing MAR systems is complex due to MAR being a synthesis of many individually complex technologies, such as 3D computer graphics, computer vision and mobile device programming. In Section 3, the importance of utilising a coherent software architecture rather than resorting to figurative "Big ball of mud" represented by the absence of any architecture was highlighted. It would then seem reasonable to assume the existence of architectures applicable to MAR design. In reality, the relative paucity of research into MAR design and architecture is evident when searching the literature, with the majority of MAR researchers concentrating on innovating new technologies and improving the many existing technologies that contribute to MAR, most references date back to the early-to-mid 2000's (MacWilliams et al., 2004; Reicher, 2004). Also the number of developers interested in MAR development is increasing, due to the power and ubiquity of mobile devices, and many of these developers do not have the spe-

cialised knowledge that MAR researchers do, thereby increasing the requirements for design guidance when developing MAR applications. In light of this, the importance of providing a viable modern architecture for MAR design is increasing.

The objective of this research has been to fill this void by proposing an architecture based on two architectural patterns that can be applied to MAR design. The main requirement that applies to MAR system architecture was identified as being the efficient processing in real-time of multiple data streams, with the concomitant requirement for a high degree of parallelism in the design. As a result of this analysis, two existing tried and tested patterns that are well suited to meeting the aforementioned requirements and constraints, were adapted and documented for architectural use in a MAR setting. Finally, in order to evaluate the architectural patterns, an application was successfully developed and evaluated. As the evaluation was conducted using an older non-flagship device, it also demonstrated the applicability of the architecture across a wide range of candidate hardware. The source of the evaluation software is open source and available for testing or adaption (see Section 6 for source URLs).

Because the underlying nature of MAR will continue to involve real-time processing of data-streams, the proposed architecture should remain applicable in the foreseeable future as new technology will introduce more data streams and better parallelisation capabilities for implementations of the architecture. For example 3D point-cloud streams from depth sensors may become commonplace, while increased CPU/GPU cores and the use of Field-Programmable Gate Arrays (FPGA) for accelerating neural network object detection may improve parallelisation capabilities.

Future work could include:

- Applying the architecture as part of a wider pattern language for MAR; and
- Extending the evaluation to a wider range of software artefact types, for example, utilising the architecture within an OO framework for MAR.

While this architecture has been specified with MAR in mind, it should also be possible to apply it to other fields requiring real-time processing of large data streams, for example, many multimedia applications have similar requirements.

References

Aldinucci, M., Danelutto, M., Kilpatrick, P. & Torquati, M. (2017). Fastflow: High-Level and Efficient Streaming on Multicore. In S. Pllana & F. Xhafa (Eds.), *Programming Multi-core and Many-core Computing Systems* (pp. 261–280). John Wiley; Sons, Inc. https://doi.org/10.1002/9781119332015.ch13

Alexander, C. (1979). *The Timeless Way of Building*. Oxford University Press.

Bauer, M., Bruegge, B., Klinker, G., MacWilliams, A., Reicher, T., Riss, S., Sandor, C. & Wagner, M. (2001). Design of a Component-Based Augmented Reality Framework. *Proceedings of the International Symposium on Augmented Reality (ISAR)*, 45–54. https://doi.org/10.1109/ISAR.2001.970514

Billinghurst, M., Clark, A. & Lee, G. (2015). A Survey of Augmented Reality. *Foundations and Trends in Human-Computer Interaction*, *8*(2–3), 73–272. https://doi.org/10.1561/1100000049

Booch, G. (2008). Architectural Organizational Patterns. *IEEE Software*, *25*(3), 18–19. https://doi.org/10.1109/MS.2008.56

Booch, G., Maksimchuk, R., Engle, M., Conallen, J. & Houston, K. (2007). *Object-oriented analysis and design with applications*. Pearson Education. http://doi.org/10.1145/1402521.1413138

Buschmann, F., Henney, K. & Schmidt, D. (2007a). *Pattern oriented software architecture: On patterns and pattern languages*. John Wiley & Sons.

Buschmann, F., Henney, K. & Schmidt, D. (2007b). *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing*. John Wiley & Sons.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. & Stal, M. (1996). *Pattern-oriented Software Architecture: A System of Patterns*. John Wiley & Sons, Inc.

Chang, C., Zhu, H., Li, M. & You, S. (2018). A Review of Visual-Inertial Simultaneous Localization and Mapping from Filtering-Based and Optimization-Based Perspectives. *Robotics*, *7*, 45. https://doi.org/10.3390/robotics7030045

Chen, Y. (2007). Decomposing DAGs into Disjoint Chains. In R. Wagner, N. Revell & G. Pernul (Eds.), *Proceedings of the 18th International Conference on Database and Expert Systems Applications* (pp. 243–253). Springer-Verlag. https://doi.org/10.1007/978-3-540-74469-6

Foote, B. & Yoder, J. (1999). Big Ball of Mud. In Foote, B. and Harrison, N. and Rohnert, H. (Ed.), *Pattern Languages of Program Design 4* (pp. 29–37). Addison-Wesley Longman Publishing Co., Inc.

François, A. (2003). Software Architecture for Computer Vision. In G. Medioni & S. B. Kang (Eds.), *Emerging Topics in Computer Vision* (pp. 585–653). Prentice Hall PTR.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc.

Google. (2020). Google Glass 2 Specifications. https://www.google.com/glass/tech-specs

Guy, R. & Agopian, M. (2018a). Physically Based Rendering in Filament. https://lwn.net/Articles/336224/

Guy, R. & Agopian, M. (2018b). Physically Based Rendering in Filament. https://google.github.io/filament/Filament.html

HTC. (2020). HTC Vive Pro Specifications. https://www.vive.com/eu/product/vive-pro

Huang, Z., Li, W., Hui, P. & Peylo, C. (2014). CloudRidAR: A Cloud-Based Architecture for Mobile Augmented Reality. *Proceedings of the 2014 Workshop on Mobile Augmented Reality and Robotic Technology-Based Systems*, 29–34. https://doi.org/10.1145/2609829.2609832

Iivari, J. (2015). Distinguishing and Contrasting Two Strategies for Design Science Research. *European Journal of Information Systems*, *24*(1), 107–115. https://doi.org/10.1057/ejis.2013.35

Intel TBB. (2014). TBB Node Types. https://www.threadingbuildingblocks.org/docs/help/index.htm#tbb_userguide/Task-Based_Programming.html

Khronos. (2015). OpenVX Standard. https://www.khronos.org/openvx/

Krasner, G. E. & Pope, S. T. (1988). A Cookbook for Using the Model-View Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, *1*(3), 26–49. https://www.ics.uci.edu/~redmiles/ics227-SQ04/papers/KrasnerPope88.pdf

MacWilliams, A., Reicher, T., Klinker, G. & Bruegge, B. (2004). Design Patterns for Augmented Reality Systems. *Proceedings of the IUI-CADUI'04 Workshop on Exploring the Design and Engineering of Mixed Reality Systems - MIXER 2004*. http://ceur-ws.org/Vol-91/paperE4.pdf

Maier, M. W., Emery, D. & Hilliard, R. (2001). Software Architecture: Introducing IEEE Standard 1471. *Computer*, *34*(4), 107–109. https://doi.org/10.1109/2.917550

McCool, M., Reinders, J. & Robison, A. (2012). *Structured Parallel Programming: Patterns for Efficient Computation* (1st). Morgan Kaufmann Publishers Inc. https://doi.org/10.1145/2382756.2382773

Microsoft. (2020). HoloLens 2 Specifications. https://www.microsoft.com/en-us/hololens/hardware

Milgram, P. & Kishino, F. (1994). A Taxonomy of Mixed Reality Visual Displays. *IEICE Transactions on Information and Systems*, *E77-D*(12), 1321–1329. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.102.4646&type=pdf

Miller, A. (2010). The Task Graph Pattern. *Proceedings of the 2010 Workshop on Parallel Programming Patterns*. https://doi.org/10.1145/1953611.1953619

Munro, D. (2020). *Patterns and Pattern Languages for Mobile Augmented Reality* (PhD Thesis). Department of Computing Sciences, Nelson Mandela University, Port Elizabeth. https://donaldmunro.github.io/thesis/Patterns_and_Pattern_Languages_for_Mobile_Augmented_Reality.pdf

Nunamaker, J., Chen, M. & Purdin, T. (1990). Systems Development in Information Systems Research. *Journal of Management Information Systems*, *7*(3), 89–106. 10.1080/07421222.1990.11517898

Ortega-Arjona, J. L. (2003). The Shared Resource Pattern. An Activity Parallelism Architectural Pattern for Parallel Programming. *Proceedings of the 10th Conference on Pattern Languages of Programming, PLoP*. https://www.researchgate.net/profile/Jorge_Ortega-Arjona/publication/272419733_The_Shared_Resource_Patterns_An_Activity_Parallelism_Architectural_Pattern_for_Parallel_Programming/links/5509b4c70cf26198a63959f5/The-Shared-Resource-Patterns-An-Activity-Parallelism-Architectural-Pattern-for-Parallel-Programming.pdf

Piekarski, W. & Thomas, B. H. (2001). Tinmith-Evo5 - An Architecture for Supporting Mobile Augmented Reality Environments. *Proceedings IEEE and ACM International Symposium on Augmented Reality*, 177–178. https://doi.org/10.1109/ISAR.2001.970530

Reicher, T. (2004). *A Framework for Dynamically Adaptable Augmented Reality Systems* (Dissertation). Technische Universität München. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.676.8644&type=pdf

Ren, J., He, Y., Huang, G., Yu, G., Cai, Y. & Zhang, Z. (2019). An Edge-Computing Based Architecture for Mobile Augmented Reality. *IEEE Network*, *33*(4), 162–169. https://doi.org/10.1109/MNET.2018.1800132

Ton That, T. M., Sadou, S. & Oquendo, F. (2012). Using Architectural Patterns to Define Architectural Decisions. *Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture*, 196–200. https://doi.org/10.1109/WICSA-ECSA.212.28

Valoriani, M. (2016). Introduction to Mixed Reality with Hololens. https://www.slideshare.net/MatteoValoriani/etna-dev-2016-introduction-to-mixed-reality-with-hololens

Voss, M., Asenjo, R. & Reinders, J. (2019). *Pro tbb: C++ parallel programming with threading building blocks* (1st). Apress. https://doi.org/10.1007/978-1-4842-4398-5

Wang, J. & Olson, E. (2016). AprilTag 2: Efficient and Robust Fiducial Detection. *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. https://doi.org/10.1109/IROS.2016.7759617