UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Alan Durnev

Essentials of Augmented Reality Software Development Under Android Platform

Bachelor's Thesis (9 ECTS)

Supervisor: Artjom Lind, MSc
Supervisor: Amnir Hadachi, PhD

Tartu 2017

Essentials of Augmented Reality Software Development Under Android Platform

Abstract: Augmented Reality (AR) is an emerging technology. Besides entertainment, AR also is found to be used in medicine, military, engineering and other major fields of enterprise and government. Regardless of the application area, development teams usually target to achieve best performance and visual results in the AR software that they are providing. In addition, the core technology used behind a particular AR software depends a lot on resources available to the team. This means, that organizations with large resources can afford to implement AR software solutions using cutting-edge technologies build by their own engineering units, whereas ordinary companies are usually limited in time, staff and budget. Hence, forcing them to use existing market solutions - toolkits.
From this perspective, this thesis work focuses on providing the basics of working with AR toolkits. In order to succeed in building an AR software, particular toolkits are selected to be reviewed, tested and compared. Moreover, during the investigation process some essentials of the AR development under Android platform are also studied.

Keywords: Augmented reality, markerless tracking, marker-based tracking, toolkits, software, Android

CERCS: P175 Informatics, systems theory

Android platvormi põhise liitreaalsuse tarkvara arendamise algmõisted

Lühikokkuvõte: Liitreaalsus on üha enam arenev tehnoloogia. Lisaks meelelahutusele on liitreaalsus leidnud kasutust nii meditsiinis, sõjaväes, masinaehituses kui ka teistes suurtes ettevõtluse ning riigiga seotud valdkondades. Arendusmeeskondade eesmärk on saavutada võimalikult hea jõudlus ning visuaalsed tulemused nende poolt toodetavas tarkvaras sõltumata kasutuspiirkonnast. Liitreaalsuse tarkvara põhitehnoloogia sõltub väga palju meeskonnale kättesaadavatest ressurssidest. See tähendab, et paremate võimalustega organisatsioonid saavad lubada endale tipptehnoloogiaid ning oma arendusmeeskondi, mille abil on neil võimalus implementeerida uusi liitreaalsuse tarkvaralahendusi. Samal ajal on aga tavalised firmad piiratud aja, meeskonna ja raha poolest, mis omakorda sunnib neid kasutama turul olemasolevaid lahendusi - tööriistakomplekte.
Sellest lähtuvalt keskendub käesolev töö vajalikele teadmistele, mida läheb vaja erinevate liitreaalsuse tööriistakomplektide kasutamisel. Selleks, et luua edukalt valmis liitreaalsuse tarkvara, on välja valitud kindlad raamistikud, millest koostatakse ülevaade, mida

testitakse ning võrreldakse. Lisaks sellele õpetatakse uurimise käigus selgeks ka mõned põhiteadmised liitreaalsuse arendamiseks Androidi platvormi näitel.

Võtmesõnad: Liitreaalsus, markeri-põhine jälgimine, markerita jälgimine, tööristakomplektid, tarkvara, Android

CERCS: P175 Informaatika, süsteemiteooria

Contents

List of Figures

List of Algorithms

1 List of terms

Position - is a specific point defined in space and in relation to another points or system. Thus, the position is often defined by context (ie point in relation to World Geodetic System)

Landmarks - any set of predefined objects of the same type. Landmarks are predefined objects that can be relatively easily detected. **Features** are tiny landmarks (ie corners) extracted from the environment

Localization - estimating main moving object's position and orientation using data from the virtual map and sensors. **Object tracking** - a process of detecting and identifying an object, being able to do so again over time. **Target** is an object being tracked. A more complex version of object tracking is tracking of moving object. **Mapping** - constructing a virtual map using relationships between the extracted features, target and model of main moving object

SDK - Software Development Kit

JVM - Java Virtual Machine

JNI - Java Native Interface. JNI is a JVM feature (therefore Java feature) that is a default mechanism to call C and C++ written code or native libraries from the code written in Java and vice versa

NDK - Native Development Kit is a toolset that allows to use C and C++ code within an Android application, also providing the developer with libraries to manage native activities and access device hardware components, as camera. NDK should not be mixed with JNI, as NDK is an Android platform feature that uses JNI

GIT - a variation of existing version control systems

API - Application Programming Interface

ABI - Application Binary Interface

APK - Android Application Package

GUI - Graphical User Interface

Library - a collection of pre-compiled functions, classes etc...

Static library - a library that is resolved at a compile-time and is *included* in the end program

Shared library - a library that is resolved at a run-time and will be *referenced* by the program

JNL - Java Native Library

Toolkit - a set of SDKs for different platforms, sometimes packed with extra GUI tools

2 Introduction

Nowadays, we more often come up with technologies that pursue the goal of replacing or complementing the reality, such as Pokemon GO game[28] that is capable of visualizing in-game virtual characters in the real world through a mobile device. Those technologies have started emerging at the end of 20th century when first solutions and researches in that area were contributing and are named *Mixed Reality*(MR) (Fig. 1).

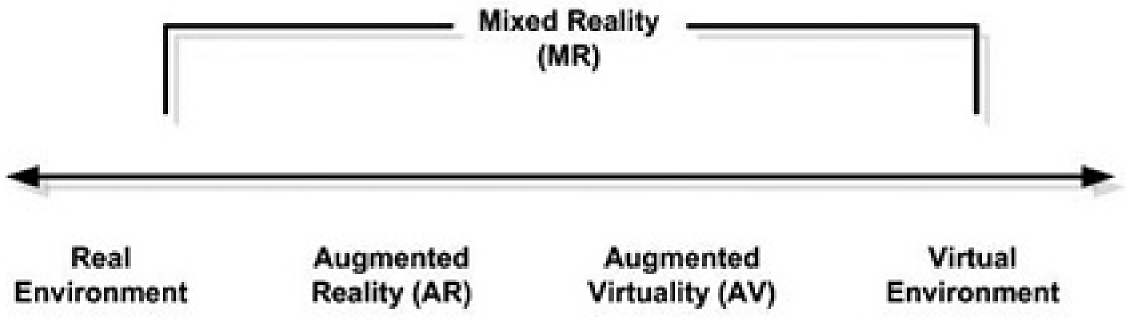

Figure 1: Mixed reality[29]

The primary research object of given thesis is *Augmented Reality*(AR), which is opposite to the *Virtual Reality*(VR) - another variation of MR. It completely encapsulates the user inside a synthetic pre-modelled environment, allowing to interact with it and keeping the real world out of sight. On the other hand, AR augments virtual objects into the real world through an intermediate device with camera. The rise on demand for mobile devices caused the vast potential of AR to be explored, as it made cheaper to complement the reality with virtual objects and became affordable for people.
AR has already been applied in many research, enterprise and entertainment areas. Some of those areas with examples are listed below[3]:

- In medicine, by visualizing medical and patient data at the same time and on the same screen (Fig. 2)
- In military, by visualizing battlefield with many annotations about the current state of the battle, information about allies and enemies
- In manufacturing, by allowing workers to visualize typical assembly procedures and measures to be taken during their work at assembly lines, thereby increasing efficiency and reducing costs with chances of making a mistake
- In entertainment, by augmenting virtual world characters and environment into the screens of devices

- In robotics, by controlling a robot or providing it with an input that can be observed by the operator, as well as by the robot itself
- In education, by visualizing abstract concepts and new information during the lecture, as professor gives a talk
- In marketing, by integrating smart advertisement into the wearable device applications and suggesting suitable restaurants, based on the location and the surroundings

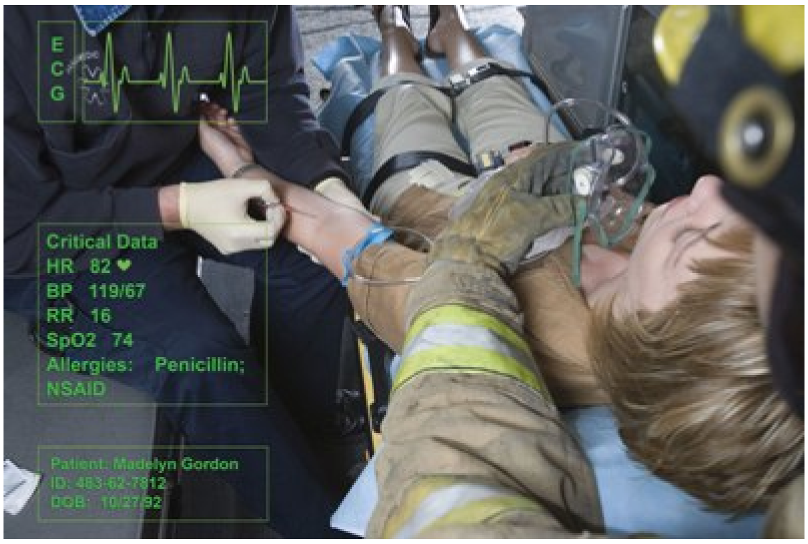

Figure 2: Medic with wearable AR glasses is able to help the injured person without being distracted by medical monitors[30]

2.1 Motivation

Most accurate- and time-critical applications rely on self-made implementations of AR engines, however in entertainment and ordinary applications different toolkits and libraries are used. It makes the development more straightforward and faster; thus, toolkits are under the scope of given thesis. The following questions have to be addressed when building competitive AR applications: "How to choose, test, compare and use the right tool for the stated needs?", "How to understand and work with tool SDK's API?", "What are the primary abstractions during developing AR application?" - the primary motivation is to answer all of those questions.

A possible solution is to define comparison metrics, to select by some criteria a number of toolkits, to review and test them. Test results are then used during comparison. In addition, developer has to be aware of toolkit SDK's API (classes, interfaces) that contains information about the abstractions. An attempt to extend the amount of information provided by abstractions, using other sources (ie toolkit's architecture) has to be made. Short analysis is provided at the end of each review chapter.
Usually, toolkits provide multiple SDKs for different platforms (Android, iOS, Unity). Due to the fact that the author owns a mobile device running Android and has minimum required knowledge about how to develop the software using Android Java, Android SDK is selected as a target SDK for the purpose of comparing the toolkits.

2.2 Structure overview

Related Work chapter gives a brief overview of solutions and open problems in AR.
AR toolkits comparison metrics and rules chapter prepares the basis for the comparison, by making the selection of toolkits, defining metrics and test flows.
ARToolKit chapter is the first toolkit to be explored and is aimed to cover needs using marker-based tracking (explained in the Related Work chapter).
Kudan chapter is the second toolkit to be explored. Kudan is different from ARToolKit as does not require any markers for tracking (further *markerless* tracking, which is explained in the Related Work chapter).
Wikitude is the final toolkit to review and participate in comparison. Wikitude's markerless tracking is reviewed.
Conclusion section briefly describes the work done, analysis of results and gained experience. Also makes sure whether stated in Motivation questions got their answers afterwards.

3 Related Work

In this chapter important aspects and existing solutions are introduced. First, types of devices used in AR are provided with description. Tracking methods are explained in detail afterwards.

3.1 Augmented Reality

Augmented reality applications are built upon on four major blocks: localization and mapping, target tracking, display technology and real-time rendering.
Nowadays, three main ways exist to implement AR. The first one that has most in common with VR - *video see-through*, then *optical see-through* and finally *projective AR*.

3.1.1 Video see-through

Video see-through represents a set of *mounted video cameras* and the *monitor* (Fig. 3). The final result is combined by merging a video shot by cameras with virtual objects (ie 2D images or 3D models) in the back-end *scene generator* and is sent to the monitor that displays it.
The key factor is video composition. The first way is to use a chroma keying technique: background of image is set to a specific, depending on the output device, color (usually green or blue), then background is decorated with virtual objects and finally background is replaced with a video and sent to the monitor.
Another approach is to combine each video frame with virtual objects using a pixel-by-pixel depth comparison, which allows both real and virtual objects to be covered by each other. Each frame is accumulated until a number of frames are gathered together and then batch of frames is sent to the monitor displaying a continuos video[2][4].

Advantages:

- Is a low-cost solution
- Uncomplicated way to manipulate virtual objects
- Brightness, contrast of virtual objects and real world objects are not affected
- Provides additional localization strategies, besides head tracking

Disadvantages:

- Resolution is lower, as depends on camera quality
- Limited field of view

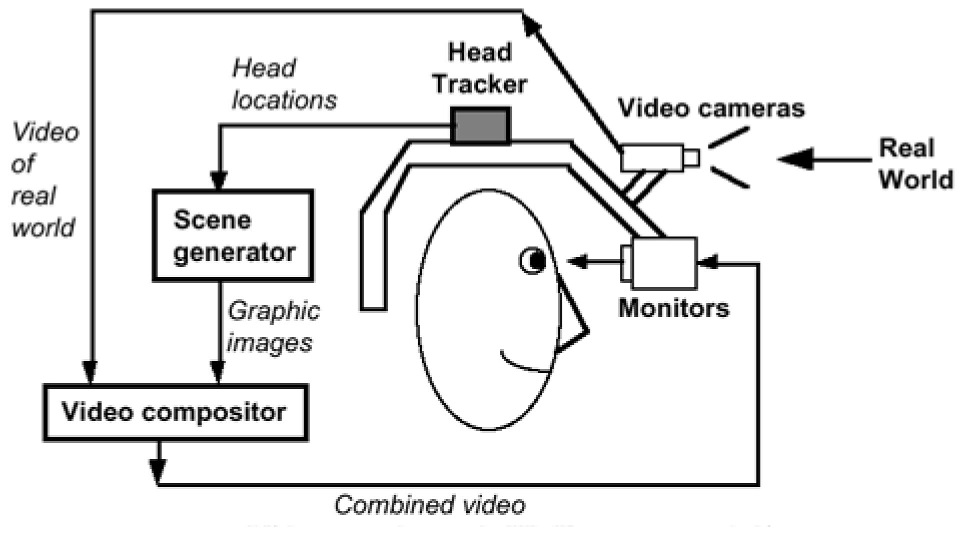

Figure 3: Video see-through[31]

- Disorientation because of difference between user's real eye and camera positions
- Time delay, as rendering requires extensive calculation

3.1.2 Optical see-through

In this case, *optical combiners* are placed in front of user's eyes (Fig. 4)(Fig. 5). Combiners, by physical nature, are both partially transmissive and reflective, so that user can observe real world through them with virtual objects being projected by the *monitors* onto the combiners and then reflected to the user's eyes. Usually, combiners are made to reflect light of certain wavelength or have a fixed percent of light that might be transmitted[3][2][4].

Advantages:

- Is a low-cost solution
- No eye-offset
- High reliability: user is able to observe the real world after emergencies and halt of hardware, which also makes it resistant to bugs and errors in software
- Resolution is not affected due to the transparency of combiners
- Time delay is minimal

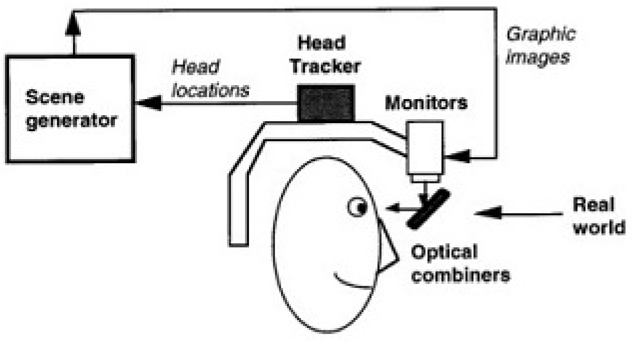

Figure 4: Optical see-through[31]

Disadvantages:

- Brightness of virtual and real world objects is affected due to the property of combiners being able to transmit and reflect light of different wavelength
- Additional devices are needed for localization
- Light coming from real world objects is combined with projected virtual objects mixing them up together

3.1.3 Projective AR

Projector-based AR users real world objects as projection surface for virtual objects (Fig.6). The result is interactable in some cases and is either a flat projected 2D object or a flat illusion of a 3D object[3][4].

Advantages:

- Can cover large surface, in order to provide a wide field of view
- Does not require any special gear to wear

Disadvantages:

- Quality of projected virtual objects depends on projection surface. Usually, projected images have low brightness which makes them applicable mostly indoors
- Requires initial calibration

Figure 5: Microsoft Hololens and Google Glass are the examples of wearable AR devices with dynamic positioning[33][34]

3.2 Display positioning

AR displays can be classified into two categories: *dynamic* and *static*. Dynamic category includes head-worn and hand-held displays, whereas static ones are often immovable dynamic displays[4].

3.2.1 Dynamic positioning

Dynamic positioning gives a massive benefit in allowing users to move around with the device and giving more valuable location-unbounded experience (Fig.5). From the other side, this freedom has a significant trade-off - those devices are disconnected from the source of power for some time, which means that after a while, user must return for a recharge. Nowadays, however, devices that belong to that category are lighter and own an integrated longer-living battery, decreasing a weight importance of given trade-off.

3.2.2 Static positioning

Static positioning, in contrast to dynamic one, does not have serious trade-offs, as by default it is meant to be immovable, which makes it perfect for exhibitions and offices (Fig.6).

3.3 Markerless tracking

Three main building blocks of markerless tracking are introduced. Each of *Simultaneous Localization and Mapping*(SLAM), *Parallel Tracking and Mapping*(PTAM) and *Dense Tracking and Mapping*(DTAM) are methods[1] working on solving tracking problems. The detail they all have in common, is the goal to reconstruct virtual environment (virtual map with virtual camera inside it) that would model real world with real camera

[1]Sets of algorithms and solutions

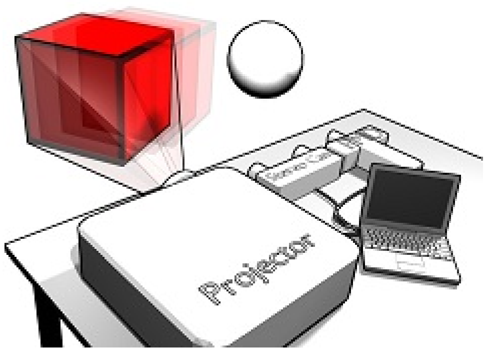

Figure 6: This is the projective AR sketch that shows its basic principle[32]

moving inside. While the camera is moving in the real world, it performs *localization* using data from sensors, computer vision methods and statistics. The second task is *mapping* of the real environment.
In AR applications, cooperation of those tasks allows to track an arbitrary place of the real world (target), making it possible to detect and identify the target again from another camera's position, orientation.

3.3.1 Simultaneous Localization and Mapping

SLAM refers to the problem of finding the position and orientation of device equipped with sensors relative to its surroundings, while mapping the structure of the environment at the same time and maintaining it (virtual map). Research of SLAM method has started in 1986 by Smith and Cheesman[1]. The popularity and grow in interest to SLAM method is related to the emergence of indoor applications, as mostly indoor places are often located beyond the GPS localization error radius, which makes it an inappropriate sensor for particular usage[6].
The goal is to reproduce the virtual map of an unknown environment, while using information about current position given same partly constructed virtual map.
Initially, problem was simple, as robot had to construct a 2D virtual map and localize itself using wheel and laser sensors. Laser, being a range-only sensor, is suitable for extracting corners from the real world and measuring the distance to them. As a result, those robots were showing good results indoors full of different objects. Next when, the cameras became an affordable type of sensors, robot was able to construct a 3D virtual map and perform localization using wheel and camera (bearing-only) sensors. In order to navigate in the real world, artificial signs were manually integrated into the environment - visual landmarks. Colorful squares were often selected as visual landmarks, making it possible for the most simple methods in computer vision to detect and identify them. Nowadays, computer scientists search for a method to replace visual

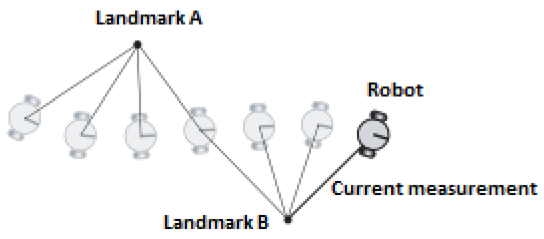

Figure 7: While the object moves through the environment, it performs measurements on landmarks it observes, which results in updating the virtual map it stores in memory[35]

landmarks with something else to completely eliminate any need in manually complementing the real world with additional objects. This is how a new branch of SLAM - *Visual SLAM*(VSLAM) appeared, allowing to use a set of non-predefined, still common-type features (ie corners, lines) excluding the need of using visual landmarks.

Advantages:

- SLAM method has the least complex solution that would satisfy today needs in markerless tracking
- Works reasonably well for small number of distinct landmarks

Disadvantages:

- Landmark dependent quadratic complexity of time and memory
- Requires distinct landmarks
- Computation overhead, as two potentially separate tasks are computed sequentially

3.3.2 Visual odometry

In navigation, odometry data is received from movement registering sensors and encoders in order to estimate the position of the moving object. A good example is robot's wheel rotary encoder that measures rotation of the wheel. *Visual odometry* (VO) has same goal but different hardware and methods (primarily computer vision) performing position and traveled trajectory estimation through analysis of a sequence of camera frames. VO is much more profitable in cost and weight, compared to the usual odometry due to the usage of camera. However, the camera still is a bearing-only sensor and provides no data about range, in contrast to expensive lasers which results in additional complex computations.

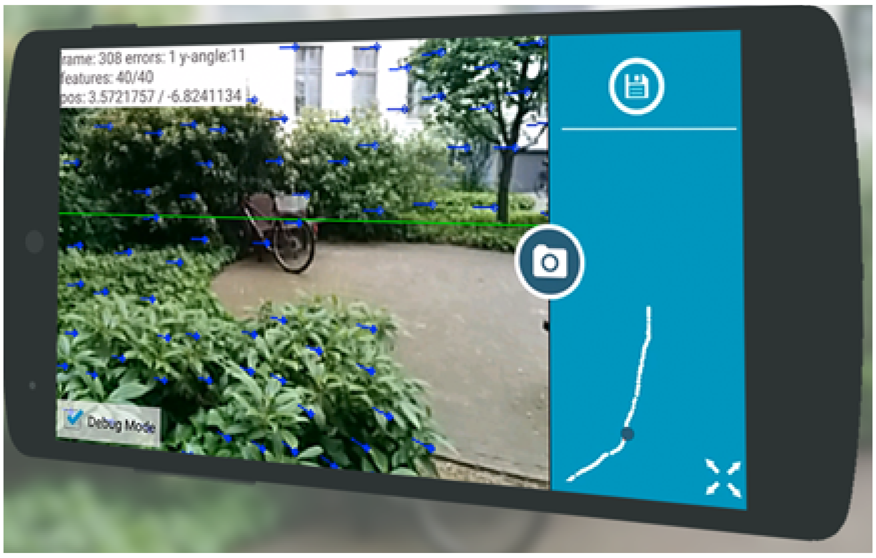
```
rame: 308 errors: 1 y-angle:11
eatures: 40/40
pos: 3.5721757 / -6.8241134
```

Figure 8: An example of a VO application. The left part of the screen visualizes the real world with extracted features of blue color. Some of them during the motion have changed the position (some blue features also have a connected blue line that reflects the translation vector), which was registered and influenced the trajectory on the right part of the screen to be updated[40]

Disadvantage of VO is absence of functionality to memorize places it has visited. VO is able to construct trajectory, but in case it visits the same place twice, it is not be able to understand that, as VO does not maintain any past information about it. Everything VO knows is a set of frames from current time step, to be able to estimate trajectory.

3.3.3 Visual Simultaneous Localization and Mapping

VO is the building block of VSLAM, as the later uses the idea of it. But, there are significant differences between end implementations. VO is aimed to localize the position and trajectory of moving object at a particular time step (locally), whereas VSLAM does it globally, as it maintains the virtual map of the real environment and keeps relations between the virtual camera[2] and other virtual map objects[3].
At the very moment, when the application running VSLAM is turned on, it knows almost nothing neither about it surroundings nor about its position in relation to them. It is reasonable to take a look at the high-level algorithm of how given system should

[2]Virtual camera is an object of virtual map and models position, orientation and trajectory of real camera
[3]Extracted features or manually registered objects (ie some place from real world)

behave further till terminating:

Algorithm 1: VSLAM algorithm's pseudocode

```
if not initialized then
    initialize;
end
while running do
    read sensor data (accelerometer, gyroscope) and using virtual camera's past
      position, orientation in combination with a new information, estimate
      camera's new position, orientation (current state);
    extract features;
    wait for pair frame;
    extract features;
    match features on both frames (all features hold a descriptor that would allow
      to identify it by comparing descriptor to other features' descriptors);
    if no matches then
        perform map correction, as some features had to have matches;
    else
        perform virtual map check to control whether some of the extracted
          features are already registered in the virtual map;
        if some features extracted are not yet registered in the map then
            register extracted features that had no matches in the virtual map;
        else
            assign to each new registered feature a probability of detecting them
              from virtual camera current state;
        end
        run uncertainty correction process for rest of registered features;
    end
end
```

It is important to understand, that camera is a central term. Devices with camera (AR glasses, mobile devices, robots) are objects that really move in the real world. In given thesis, author abstracts away from them and defines camera as an object that owns position, orientation (further *state*) and trajectory. Camera is provided with the sensor data.

3.3.4 Parallel Tracking and Mapping

PTAM method is one of the variations of VSLAM proposed by Klein and Murray in 2007[9] and its main idea is to split localization and mapping into separate tasks to be processed in parallel. The *localization thread* matches features and estimates the camera position for every new frame. The *mapping thread* iteratively updates the position of

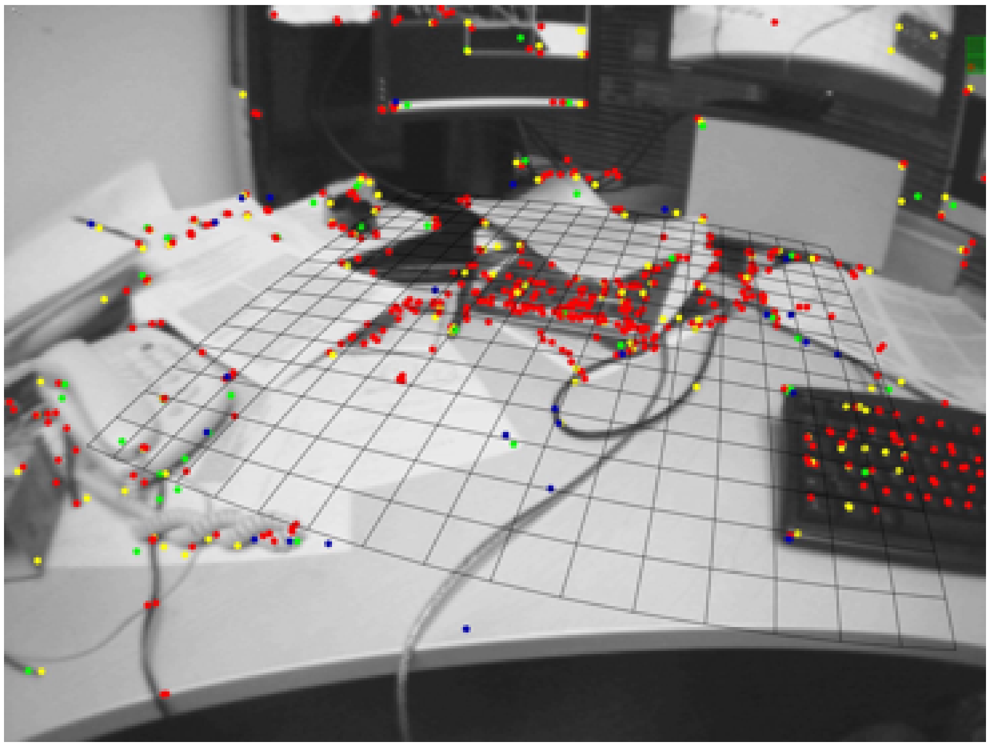

Figure 9: Application running PTAM draws features registered in the virtual map and visible from current camera's state[9]

registered features, that compose the virtual map, accurately as possible[9].
An important quality of PTAM method is that the mapping is performed only when there are free resources available on the mapping thread. This allows the localization thread to follow the camera in real-time regardless of the complexity of the scene, decreasing the load on the processing units. Once the camera goes static in an already mapped environment, the mapping thread is given some resources to analyze state and new information, in order to update the existing virtual map.

Advantages:

- PTAM method has a better performance efficiency solution than VSLAM

Disadvantages:

- Implementing PTAM is significantly harder, due to concurrency

3.3.5 Dense Tracking and Mapping

DTAM is a method for real-time camera localization and map reconstruction, which uses pixel methods instead of feature extraction. Pixel methods make use of all the data on

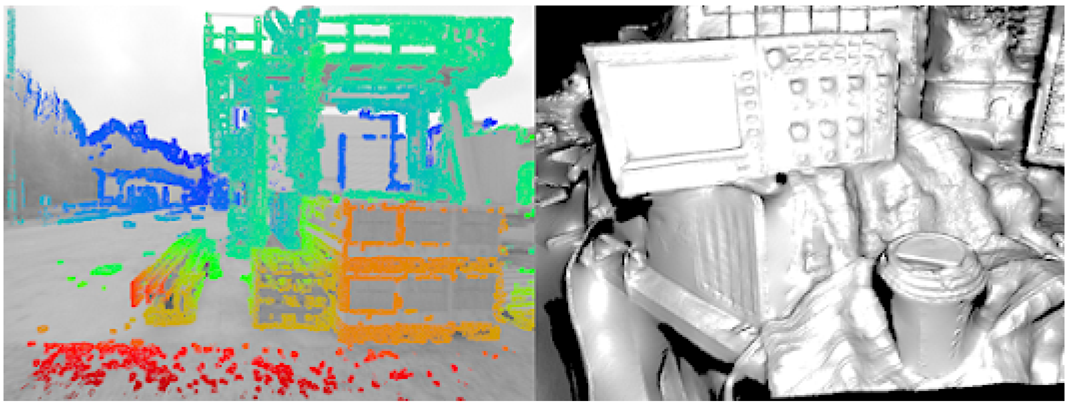

Figure 10: A Semi-dense and dense maps[36]

the image. While the camera is changing its position, it uses every single frame as an input for the Multi-view Stereo Reconstruction, that is able to accurately build the 3D shape of the scene by a number of sets of frames made from different positions[27]. This scene 3D model is not only used for mapping, but also for camera localization.
Due to the recent availability of powerful *General-purpose computing on graphics processing units*(GPGPU) processing[4] made this method implementation possible.

Advantages:

- DTAM achieves a goal of the highest quality among others

Disadvantages:

- Implementation of DTAM is harder, because of the need to create a dense 3D scene model
- GPGPU supporting hardware is required

3.4 Marker-based tracking

Marker (Fig. 11) is an easily detectable predefined sign-object in the real environment. A marker based approach easily solves the problem of aligning virtual objects in the real environment with visual markers, that are detectable with common computer vision methods[5] and do not require explicit work to be done in localization and mapping. Once

[4]The use of GPU, to perform application computations, traditionally handled by the CPU, leaving the CPU mostly with some coordination-and-management tasks and trying to achieve maximum possible level of parallelism for given set of computations and tasks in general

[5]Image processing, pattern recognition and other techniques

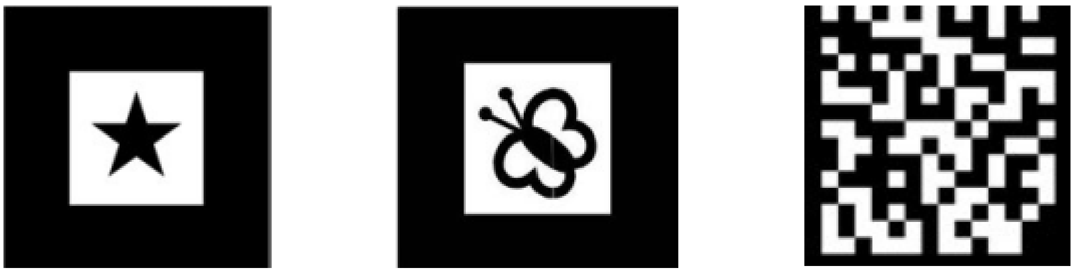

Figure 11: All markers have one thing in common: an unique black and white pattern[37]

detected, it defines the correct state of the camera. The ease of use and an extensive list of marker-based toolkits made this method very popular among AR application developers.

3.5 Conclusion

In this chapter, theory behind AR was provided. Developer must be able to distinguish a variety of device type's that can run AR applications. Typically, they have a camera sensor, navigation sensors and a screen to display virtual content over the real world. Additionally, developer must have in mind an approximate conception regarding methods used for implementing marker-based and markerless tracking engines. Altogether, making the process of working with AR abstractions less complicated. With that knowledge, it is possible to proceed to the process of selecting toolkits, defining metrics for tests, testing, reviewing and comparing them.

4 AR toolkits test metrics and rules

In this chapter target toolkits are selected and metrics for testing them are specified. Test results are later used during comparison. The description and general implementation of the test application are introduced in the end.

4.1 Requirements for the selection of the toolkit

In order to test and compare toolkits, an important question must be raised: "What are the nowadays and future needs for AR and how do they differ from the past ones?". Limited by the performance of the available hardware and software, the requirements in the past were mostly dedicated to the idea of marker-based tracking. Today, the market is full of relatively powerful wearable devices with built-in camera allowing to run AR applications and making it possible to research both marker-based and markerless tracking. However, future requirements are mostly focused on the markerless tracking, as the only difference between marker and visual landmark is that the first one is a more advanced variation of the second one, still manually placed in the real world for tracking. In addition the following options are found critical for toolkit to have:

- Period for testing
- An Android Java SDK
- A comfortable API to work with rendering[6], if possible

Taking all of it into account, author proposed to make a selection of three camera-based toolkits:

- ARToolKit is the marker-based-only AR toolkit that satisfies the requirements for the past and today solutions in camera-based tracking
- Kudan AR (further *Kudan*) is the AR toolkit that supports both marker-based and markerless tracking and satisfies the requirements for today and future needs. Kudan's markerless tracking is reviewed
- Wikitude SDK (further *Wikitude*) is an enterprise AR toolkit that supports both marker-based and markerless tracking. Analogically to Kudan, only the markerless tracking of Wikitude is reviewed

[6]Rendering is the process of generating a result image (drawing an image) from a 2D/3D model stored in binary in any type of computer memory

4.2 Test metrics

In order to figure out what is the most suitable toolkit for the development of further provided test application, author decided to assign a weight value for each metric:

- In case of simple[7] qualitative metrics:

Algorithm 2: Simple qualitative metric points calculation

```
if condition is met then
    toolkit receives 1 point;
toolkit receives 0 point;
```

- In case of composite (metric controls C > 1 conditions) qualitative metrics:

Algorithm 3: Composite qualitative metric points calculation

```
forall met conditions do
    toolkit receives 1/C points;
```

- In case of quantitative metrics:

Algorithm 4: Simple quantitative metric points calculation

```
if at least one toolkit got result more than 0 then
    toolkit that got the best result R receives 1 point;
    any other toolkit that got result P receives R/P points;
```

Also, if undefined behavior is detected, then author is obliged to take some of the points off (amount of removed points depends on the level of severity and is up to the author).

The primary metrics are qualitative and are related to the observable behavior of particular toolkit's tracking and rendering engine. In order to provide an explanation for the choice of a particular metric, a reason is provided. The reason is represented as a potential use case from an AR application where user is able to augment the detailed virtual model of some building (Fig. 12):

[7] Metric controls one condition

Figure 12: User (ie architect) is able to rotate and move the device, in order to observe the model of the building from different angles and distances[39]

- General behavior of rendered virtual cube object (further object). Object should be stable and must have no random convulsions on camera movement (Simple):
    1. Specify the target to be tracked
    2. Place the camera, so that object would appear on the screen (be visible)
    3. Hold camera still
    4. Register the result
    5. Expectation: object does not move. Reason: randomly moving virtual model can ruin user's AR experience. S/he will not be able to keep focus researching the virtual model while it moves
- Behavior during fast[8] camera shifts (Simple)
    1. Specify the target to be tracked
    2. Place the camera, so that object would appear on the screen
    3. Start moving camera fast along all 6 degrees of freedom, still keeping the trackable target in the sight of camera
    4. Register the result
    5. Expectation: object is visible all the time during the camera shifts. Reason: If user has to take a look at the virtual model's different parts in hurry, then the virtual model must behave according to the expectations

[8]Around 3 camera shifts per second. Approximate measurements are made with the help of a mobile device's timer and might have a non-critical error in it

- Behavior on being very close and far away (relatively to the room size)[9] from the object (Composite: very close + far away):

    1. Specify the target to be tracked
    2. Place the camera, so that object would appear on the screen
    3. Move the camera close to the object as possible, keeping it visible
    4. Register the result
    5. Expectation: object is visible and grows in size without loss of quality. Reason: in order to observe virtual model's some part (ie building's window) in detail, then by moving camera towards this part it should be enlarged without quality loss
    6. Move the camera far away from the object as possible
    7. Register the result if maximum available distance is achieved or object disappeared.
    8. Expectation: object is visible from the state of the maximum available distance (2.6m). Reason: if user moves the camera away from the virtual model, in order to observe the entire virtual model from the distance, then virtual model must persist the scale of all it's composite parts and keep the quality

- Behavior on camera rotating around the object (Composite: 4 cube faces):

    1. Specify the target to be tracked
    2. Place the camera, so that object would appear on the screen
    3. Move with the camera to the object's first face
    4. Register the result
    5. Move with the camera to the right to the object's second face
    6. Register the result
    7. Move with the camera to the right to the object's third face
    8. Register the result
    9. Move with the camera to the right to the object's fourth face
    10. Register the result
    11. Expectation: object always remains visible. Reason: virtual model must be observable from all sides, so that user could have an idea of how the modelled object potentially might look like in the real world

[9]Around 2.6m. Approximate measurements were made with the help of a retractable flexible rule and might have a non-critical error in it

- Behavior on camera turning away from and back to the object (Composite: slow + fast):

    1. Specify the target to be tracked
    2. Place the camera, so that object would appear on the screen
    3. Slowly move the camera to the right, till object has disappeared
    4. Register the result
    5. Expectation: object stays visible until it is out of camera's sight. Reason: if user is interested in the virtual model's part that can be observed from the current state only partially, then virtual model must not disappear
    6. Slowly return camera to the start position
    7. Register the result
    8. Expectations: object is visible. Reason: if by return to the start position, virtual model is not visible anymore, then AR application has lost the target. As a result, user must spend additional time on the application restart
    9. Move the camera to the right fast, till object has disappeared
    10. Register the result
    11. Expectations: object stays visible until it is out of camera's sight. Reason: reason mentioned above can be also applied here
    12. Return camera to the start position fast
    13. Register the result
    14. Expectations: object stays visible until it is out of camera's sight. Reason: reason mentioned above can be also applied here

- Behavior on walking away to another room with camera and turning back to the object (Simple):

    1. Specify the target to be tracked
    2. Place the camera, so that object would appear on the screen
    3. Move with camera to the another room
    4. Return back to the room, where trackable target is
    5. Register the result
    6. Expectation: by return, the object must be visible. Reason: if user is working on multiple projects at the same time, then s/he must be able to switch working places without restarting the application

Other qualitative metrics are:

- Existence of examples and sample projects (Simple):
    1. Select an appropriate sample project to integrate test application
    2. Check a variety of sample projects, left after
    3. Register the result, based on author's experience. Reason: for the developer, who just started working with an AR toolkit, a number of example projects is important to master the basics of it
- Existence of toolkit overview and well-structured Java SDK API documentation (Composite: overview + API documentation):
    1. Research high-level architecture of the toolkit
    2. Research abstractions related to the toolkit
    3. Make changes to the source code of the sample project during the integration of the test application, if required. Find and research SDK's API
    4. Register the result, based on author's experience. Reason: any developer, who is developing AR applications should be familiar with the tools

Other metrics are quantitative and numerically express:

- Amount of the most important[10] code lines (the less, the better) that are spent on preparing rendering[11] (ie loading 3D models or 2D textures, creating virtual objects and applying various transformations with effects to them) (Simple):
    1. Detect the most crucial parts regarding rendering preparation, during test application integration. Place the shortened (without comments, newlines and extra configuration details) source code into the appendix
    2. Calculate amount of code lines
    3. Register the result
- Amount of the most important code lines that are spent on configuring tracking[12] (initializing and managing tracking) (Simple):
    1. Test provided for rendering preparation can be also applied here

[10] An attempt to drop the supporting and detailed source code was made, in order to eliminate the process of complete API research

[11] In review chapters is denoted as Rendering

[12] In review chapters is denoted as Tracking

- Complexity of development and integrating test application's source code into the sample project in hours (Simple):
    1. Review the toolkit
    2. Register the result
    3. Prepare the test application using the toolkit
    4. Register the result

Metrics that are not mentioned:

- Complexity of setting project up and its configuration. The reason behind that, is that in order to eliminate issues related to configuration, all possible source code changes are done in a suitable toolkit's sample project
- Complexity of toolkit review. The reason behind, is that mentioned toolkits are reviewed only from the perspective of tracking and rendering. There are many additional possibilities and functionalities, uncovered under the scope of given thesis

In order to achieve reliable test results, author prepared a feature rich environment for it (Fig. 13)

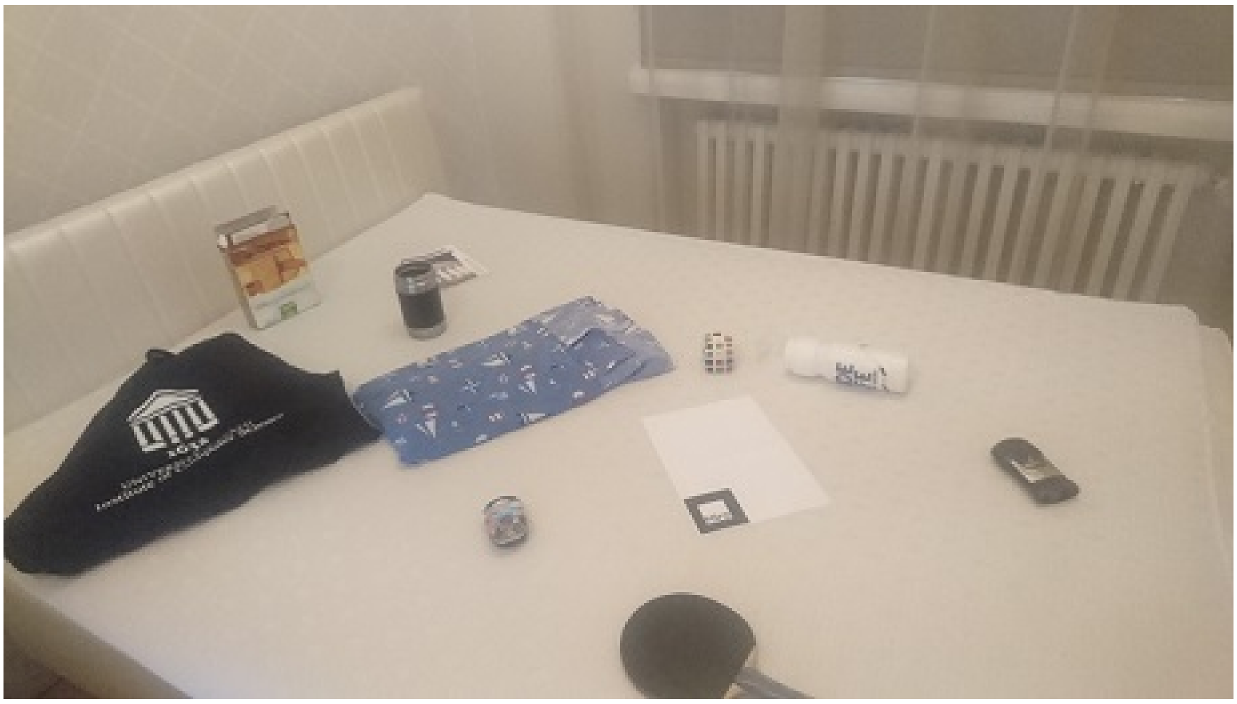

Figure 13: White surface with different objects on it represents the working surface, which is located inside the room with constant lighting (no shadows)

4.3 Test application

Test application must be simple, yet detailed enough to be able to perform all possible test actions related to the desired metrics.

4.3.1 Architecture of test application

When the application has started, the user is suggested to enter one short word and launch the tracking activity afterwards. Application generates a texture, consisting of a light-blue background and a white word on it, the user entered. Then, application loads or constructs the 3D model of cube (further *model*), applies the 2D texture (further *texture*) or material to all of its faces, adds created virtual object to the 3D world of the target[13] and starts tracking the target. Toolkit's tracking engine notifies when target is detected and toolkit's rendering engine draws a virtual object. Afterwards, the virtual object appears on the screen. The more detailed flow of the tracking activity:

Algorithm 5: Tracking activity's flow

```
while application running do
    select a suitable place for target (target defines the world where virtual object
      resides);
    start tracking mode;
    while in tracking mode do
        if interrupt triggered by user actions is detected then
            end tracking mode;
        perform required tracking engine operations;
        send updated data to the rendering engine;
        perform rendering;
```

4.3.2 Implementation of test application

The general implementation of texture creation is provided in Appendix A.

[13]In case of marker-based tracking, the target is a marker and user must physically select a place for it. Whereas, with markerless tracking, target is an arbitrary place of real world chosen by the user through device

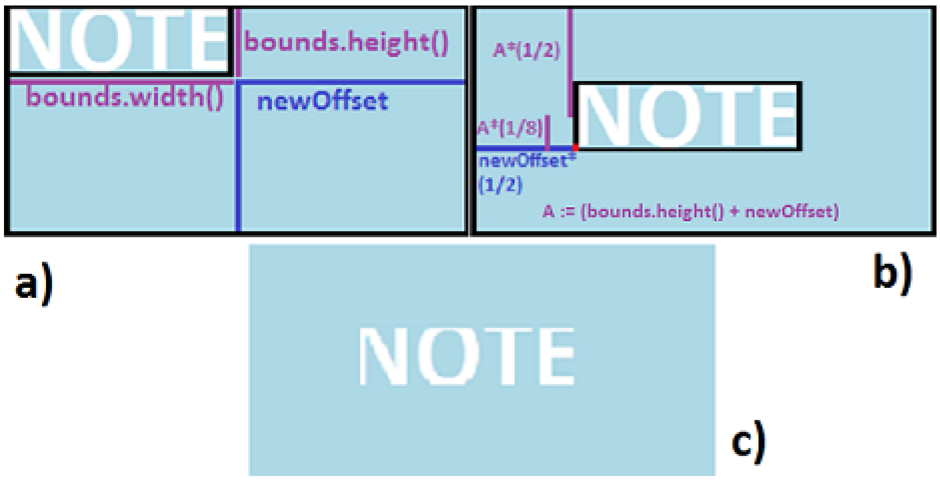

Figure 14: Calculations performed during texture image file generation (Appendix A). a) Initial placement of text b) Image has a tiny red rectangle in the left-bottom corner of the black border rectangle - an origin in relation to which calculations are performed to place the text as required c) Final result for given .png file creation. When the generated rectangle-form texture is applied to the cube model, it is additionally shrinked to achieve the form of a square

Figure 14 provides a number of images with calculations and explanations.

4.4 Conclusion

When toolkits are selected, metrics are defined and goals are set it is possible to proceed to an actual implementation and testing. Following chapters provide general, architectural and development information of particular toolkits.

5 ARToolKit

In this chapter ARToolKit library is introduced using Android platform. In order to avoid reader's misconception about Android Java definitions and concepts, in the following chapter some of them are explained in detail.

5.1 Introduction to ARToolKit

At the moment of writing this thesis, ARToolKit is the most well-known open-source marker-based AR toolkit. It is widely used in AR applications and has already achieved 650,000 downloads only on SourceForge[14] since its public release in 2004[10].
ARToolKit allows to develop AR applications under a number of major platforms: iOS, Android, Linux, Windows, MAC OS X. SDKs are compiled for a fast development start. ARToolKit has a primary focus set on wearable devices (by supporting OpenGL ES[15], GPS, compass and automatic camera calibration utilities)[10].

5.2 ARToolKit's architecture

Before the development start, it is useful to explore high-level architecture, in order to understand how its components are connected and cooperate with each other.
Fig.15 illustrates the three main components (further *modules*) of ARToolKit library[11]:

ARToolKit Core: This is a core module that consists of native C static libraries that handle low-level functionality, like rendering. Those libraries cooperate with other external core libraries (ie computer vision libraries). Core module is used to build a shared library

ARToolKitWrapper: Next module is a native C++ shared library that provides a limited API as a set of functions to the core module that manages ARToolKit application lifecycle, such as:

- initialization
- configuration
- registering marker
- retrieving transformation matrix

[14]A web service that gives access to a centralized public storage of free and open-source software products and projects

[15]OpenGL for Embedded Systems is a subset of OpenGL API made specifically for embedded systems, like mobile devices

- cleaning up on exit

JVM JNI mechanism registers those API functions as JNL, then connects those native implementations with java native method declarations in **NativeInterface.class** API endpoint class and adds it as an import to other classes under ARBaseLib for access.

ARBaseLib: ARBaseLib module is the final building block and primarily a connection point between developer's application and core native functions, conveniently encapsulating and persisting good modularity. Any action that requires core native functions should be submitted to ARBaseLib classes that, in turn, delegate them further in a more appropriate low-level way. ARBaseLib consists of three primary classes:

- **ARToolKit.class** - an instance of given class is a singleton that provides access to native functions, doing error checking and type conversions
- In Android, activity is a single focused component of application, the user can interact with. Every activity is provided with a window that becomes a base for GUI implemented in the *Activity.class* subclass. ARToolKit's **ARActivity.class** is an extended activity fulfilled with more details and functions, yet not enough to have application up and running alone (is declared abstract[16])
- Finally, a result has to be also rendered. An instance of ARToolKit's **ARRenderer.class** makes OpenGL ES function calls for rendering. AR application consists of two views stacked one upon another inside a *FrameLayout.class* view group:
    - Surface view that displays each frame that comes from the camera
    - An OpenGL surface view with a transparent background that holds rendered virtual objects

  The final view is combined from them, resulting in virtual objects overlaying real world objects in real-time. Final view's content is sent to the mobile device's display (further *screen*).

5.3 Developing test application

ARToolKit has a GIT repository[17] that provides the beginner with sample projects. ARSimpleProj for Android has been chosen to be a skeleton project for integrating test application (Appendix E).

[16]In Java language, any class that is declared abstract cannot be instantiated
[17]https://github.com/artoolkit/artoolkit5/

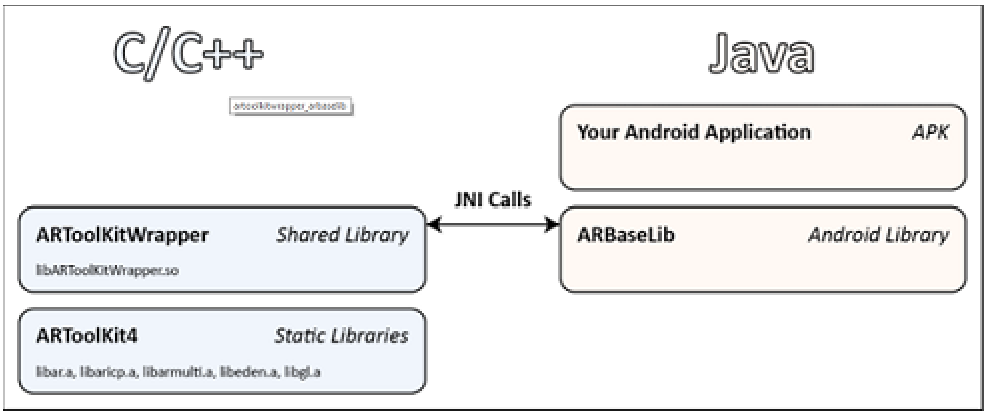

Figure 15: ARToolKit4(ARToolKit Core), ARToolKitWrapper and ARBaseLib are the three main components of ARToolKit application, that is the 4th component using them[11]

Hiro pattern (Fig. 17) is used for tracking keeping virtual object above it even when camera's state (position and orientation) is changed. Whereas, for rendering ARToolKit provides a raw OpenGL ES context which requires presence of skills in computer graphics area. Another option is to perform model loading and rendering through ARToolKit's native interface. Despite the fact, that provided possibilities are acceptable, author of given thesis focuses on libraries and frameworks that would simplify the rendering process on Android platform. The solution was found in an ARToolKit helper library, that connects JPCT[18] with ARBaseLib producing an *ArJpctBaseLib*[19] that leaves tracking unchanged to ARBaseLib, but rendering becomes more compact and comfortable due to the Java API. ArJpctBaseLib is added in form of .aar file[20] and is used as a module dependency for main module of the project. Further some important moments regarding development using ARToolKit are provided.

ARSimpleApplication.class (specific to sample project)**:** in Android library, *Application.class* is the class for maintaining application's global state and providing context. Singleton instance of it additionally performs during run-time a performance critical operation: it copies from the .apk[21] file's filesystem .../assets/ directory into the application specific cache on the primary memory[22], so that every next call that requires assets[23] takes less time. When the application is removed or the device is out of memory,

[18] Android and Java 3D engine

[19] https://github.com/plattysoft/ArToolKitJpctBaseLib

[20] Extended variation of java archive .jar file for Android. In addition to .class files it includes resources

[21] Is the package file format used by the Android OS to distribute and install Android application

[22] Internal storage

[23] Asset is any file of any type (ie .xml, .json, .jpg) the developer wants to include into his/her application.

the cache is cleared.
In order to avoid future misunderstanding in difference between .../res/ and .../assets/ directories:

- the files inside .../res/ directory define the run-time data structure and properties of it (ie using XML[24] markup language). However, media content can be also placed there

- Resources placed in .../res/ directory are easily accessible at run-time through *R.class*, which is created and compiled by Android SDK compiler itself, whereas .../assets/ directory has raw data, that has to be read manually

5.3.1 Tracking and rendering

ARToolKit's tracking and rendering source code is provided in Appendix B. In case of using ArJpctBaseLib, actual rendering process is hidden behind API, but preparing virtual objects for it is on developer's side.

- **ArJpctSimpleActivity.class** (specific to sample project) - an instance of given class aids in rendering preparation and starting marker-based tracking. By extending *ArJpctActivity.class* it is forced to override method *ArJpctSimpleActivity#populateTrackableObjects*. The goal of last one is to register marker and virtual objects that are rendered on successful marker detection and identification

- **TrackableObject3d.class** - an instance of given class represents a trackable predefined target (further *marker*). Marker must be detected and identified by the ARToolKit's tracking engine. Each marker's pattern file[25] is loaded by specifying a config-string: <single;path_to_pattern_file;pattern_width> - which represents properties separated by a semicolon

- **Object3D.class** - an instance of given class represents a virtual object. After applying the texture to the model of virtual object, it is ready to acquire a bond with marker. All virtual objects must have a marker parent in order to be rendered[26]. The model's file with extension *.obj* is a common format for

Assets are stored in .../assets/ directory, being the part of .apk file's filesystem

[24]Extensible Markup Language

[25]A pattern file is a special pattern recognition file, received after training process on input picture of marker, in order that ARToolKit could detect and identify the initial marker under different angles and distances

[26]Center of parent becomes the origin of the 3D coordinate space. All algebraic and geometric calculations relating child virtual objects are performed relatively to the origin

describing model's geometry[27].

While the application works, ARToolKit's tracking engine tries to detect every real world object that reminds any of the registered markers. Once detected, an identification process starts that makes sure which one of registered markers is exactly detected. After marker is identified, application is able to calculate transformation matrix to be applied to prepared virtual objects during rendering. Previous calculation is possible, due to the difference in how marker must look from specific distance, angle (prior information from pattern file) and how it really looks from camera's current state (Fig.16).

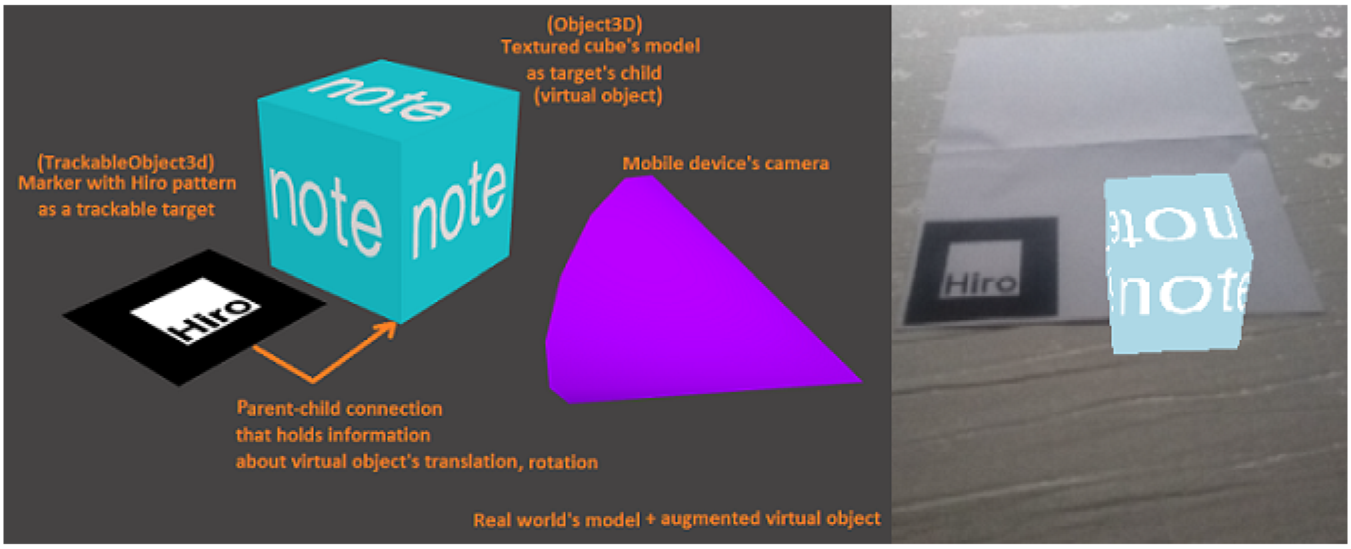

Figure 16: Model of real world with augmented virtual object

5.3.2 Build preparation

Before build process, one explicit thing has to be done. ARBaseLib's NativeInterface.class has a static function *NativeInterface#loadNativeLibrary* that finds recursively and loads specified shared libraries at run-time from the *jniLibs*[28] directory. ARToolKit provides scripts *build.sh* and *build_native_examples.sh*. These scripts put into each subproject they find a jniLibs directory (Fig. 18) that has many architecture specific subdirectories, each containing required wrapper shared library compiled for different ABIs (using pre-compiled C both static and preliminary shared libraries located

[27]The model file is taken from the public resource: https://clara.io/view/085783aa-6482-4ad0-bbdb-c2827f5bcf61/ and is also used in further reviews

[28]jniLibs is a special directory in Android Studio project that contains CPU architecture specific native object code and is auto-uploaded into the device

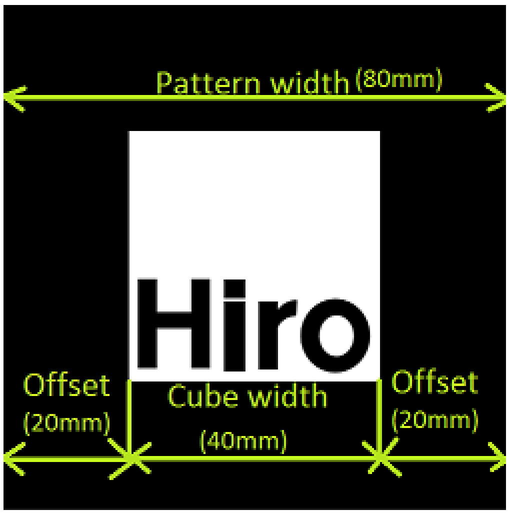

Figure 17: The square hiro pattern has width of 80mm[38]

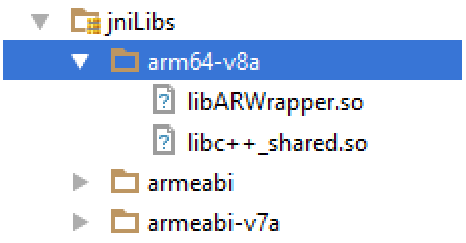

Figure 18: jniLibs directory has many architecture specific subdirectories

in the same directory with scripts). Devices that run Android OS might have different CPUs which might have different architecture forcing architecture specific C and C++ source code to be compiled into different object code. During run-time an appropriate shared library is chosen and taking into account the fact that the application is tested on the mobile device with CPU architecture ARMv8-A. As a result, the shared library from the arm64-v8a subdirectory is loaded during run-time.

At this point, the project is ready to be built into .apk file, transported to the device and installed there.

5.4 ARToolKit results

Here the author provides the test results of running the integrated test application according to the predefined metrics.

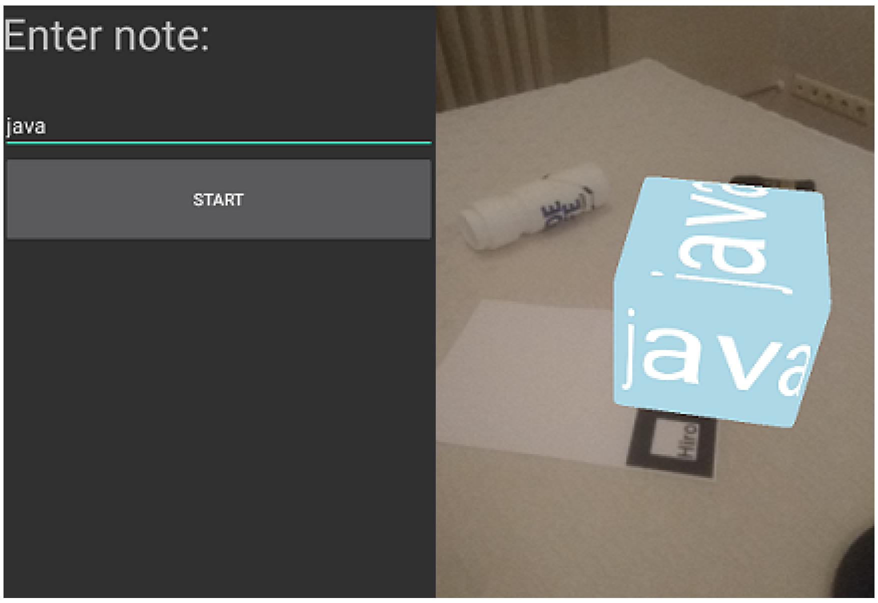

Figure 19: Application start

1. **General behavior of rendered virtual cube object (further object):** object had only slight convulsions, resulting in a penalty of 20% from 1p. Result: 1p*(1-0.2) = *0.8p*
2. **Behavior during fast camera shifts:** object was constantly disappearing and appearing back again. Result: *0p*
3. **Behavior on being very close to the object:** expected behavior (Fig. 20). Result: (1/2)p = *0.5p*

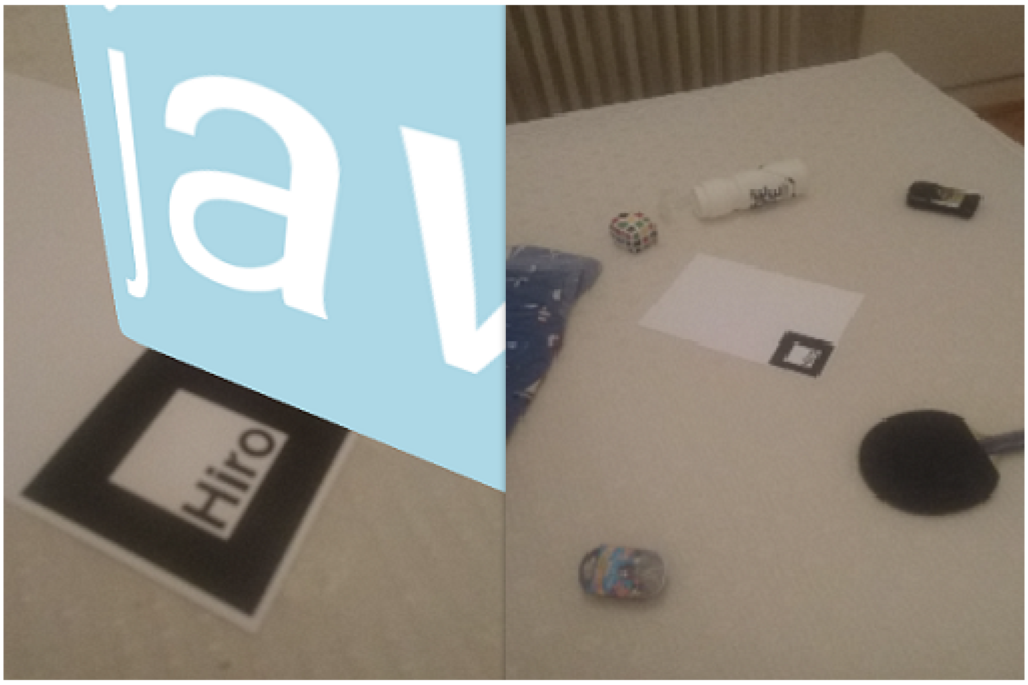

Figure 20: Behavior on being very close and far away (relatively to the room size) from the object

4. **Behavior on being far away (relatively to the room size) from the object:** marker was not detected from the state with distance of 1.3m[29] from it, which is half of the maximum available distance (2.6m). This does not satisfy the requirement. (Fig. 20). Result: *0p*

5. **Behavior on camera rotating around the object:** expected behavior (Fig. 21). Result: (1/4)p*4 = *1p*

[29]Approximate measurements are made with the help of a retractable flexible rule and might have a non-critical error in it

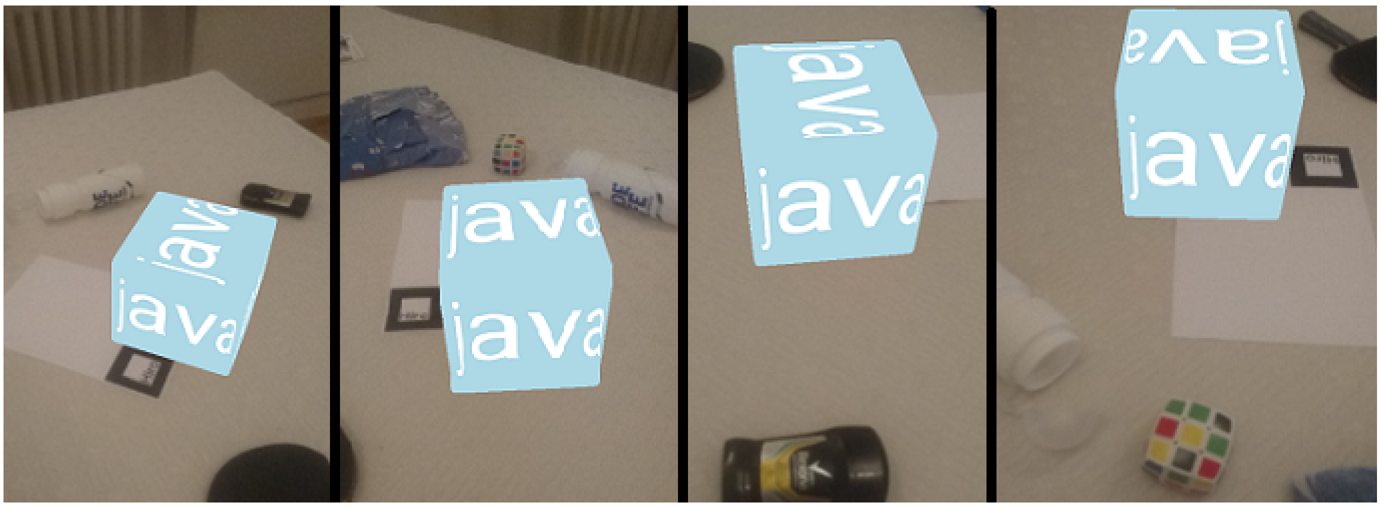

Figure 21: Rotating camera around the object

6. **Behavior on camera turning away from and back to the object (slow) :** if only a half of the marker could be detected, object was not rendered, resulting in a penalty of 50% (Fig. 22). Result: (1/2)p*0.5 = *0.25p*

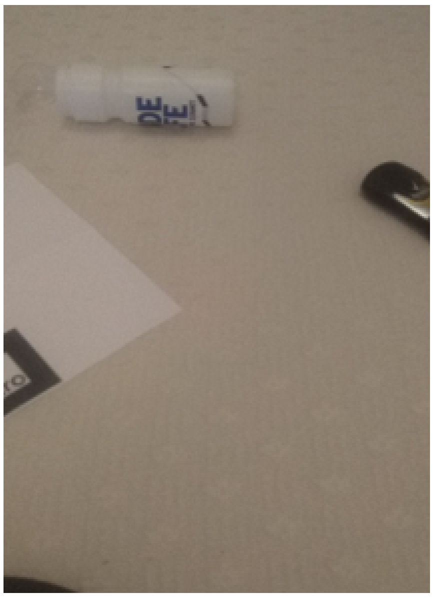

Figure 22: Tracking engine is not able to detect marker by the half of it

7. **Behavior on camera turning away from and back to the object (fast) :** expected behavior. Result: (1/2)p = *0.5p*

8. **Behavior on walking away to another room with camera and turning back to the object:** if by return of the camera back to the room the marker is still there - object would be rendered. This results in an unfair comparison in relation to the markerless tracking toolkits, as they are obliged in this situation to perform complex computations to achieve object rendering without marker by return to the room. Result: *0p*

9. **Existence of examples and sample projects:** there are a lot of sample projects provided by the ARToolKit[30]. Result: *1p*

10. **Existence of toolkit overview:** a lot of main and additional information about the product is provided. Result: *1p*

11. **Existence of well-structured Java API documentation:** was not detected for ARToolKit 5. Result: *0p*

12. **Amount of (the most important) code lines that are spent on preparing rendering:** 15 lines of code (further *loc*) (Kudan has 15 loc, Wikitude has 34 loc). Result: *1p*

13. **Amount of (the most important) code lines that are spent on configuring tracking:** 10 loc (Kudan has 13 loc, Wikitude has 23 loc). Result: *1p*

14. **Complexity of development and integrating test application's source code into the sample project in hours:** 4 hours for all routines (Kudan has 2 hours). Result: 2/4 = *0.5p*

Total: **7.55p**

5.5 Conclusion

This chapter made a detailed overview of how to work with an extended version of ARToolKit avoiding raw OpenGL ES context during rendering. In addition, high-level architecture of ARToolKit was covered.
Despite the fact, that ArJpctBaseLib significantly reduces amount of time and code spent on rendering by providing Java API for preparing-only routines, it is not an official project of ARToolKit and is developed independently by a small open-source team. As a reason of that, it is not capable of supplying fixes and new functionality with pace of ARToolKit teams. Any potential problem with accuracy, performance and system design may appear any time and cause decrease in usability and maintainability leading to loss of audience and profit as a result.

[30]https://github.com/artoolkit/artoolkit5/

According to the test result, ARToolKit has an expected behavior if it is not that far away from the marker (ARToolKit results, test 4). Also, marker could be still detected by the tracking engine, if small part of it was out of camera's sight. Otherwise, virtual object would not be rendered (ARToolKit results, test 6). From the perspective of user's AR experience, in order to observe the augmented virtual objects, user must keep the marker in the sight of the camera. If marker is immovable, then user's experience is bound to particular place, as a result. On the other hand, user is able to leave the place, where marker is positioned and return to it later. Marker-based tracking assures that even if the marker was moved to another place, virtual content related to it is rendered as before. This leads to the conclusion that ARToolKit would be a suitable toolkit to develop applications related to:

- Museum exhibitions: ie augmenting the model of some building from Roman Empire age
- Basic AR games: ie board games, where battlefield can be observed by all players
- Architectural bureaus: ie augmenting a not yet finished model of some building, in order to control how it looks from a particular side
- Training and education: ie markers can be placed over important tools for mechanical engineers, like assembly lines. On marker detection critical information about the tool is provided

6 Kudan

In this chapter a VSLAM-based Kudan toolkit[31] is explored using Android platform.

6.1 Introduction to Kudan

Besides marker-based tracking, Kudan also provides support for markerless one using self-implemented VSLAM-based engine which makes this product highly valuable in industry of AR. Kudan with its AR SDKs (Android, iOS, Unity), KudanCV SDK (Android, iOS), GUI tools provides the developers with stable and comfortable platform for AR development.

6.2 Kudan's architecture

AR SDK provides an API to perform various forms of tracking and rendering preparation. AR SDK depends on KudanCV which is a core engine written in C++ that consists of computer vision and tracking modules and can be also integrated with other platforms and third-party renderers[18]. KudanCV is an enhanced and performance-tuned OpenCV, as it has got a lot of performance-critical places in source code manually rewritten in Assembly Language. Hence, achieving high-level of optimization in combination with thread-safe concurrent environment[17].

6.3 Developing test application

Kudan has a GIT repository[32] that provides the beginner with demo projects. KudanArbiTrackTutorial for Android has been chosen to be a skeleton project for integrating test application (Appendix E).
Organizational aspects of developing, maintaining and deploying application under Android platform were covered during the review of ARToolKit, therefore any kind of precise explanation is further skipped on purpose, leaving only the most crucial aspects of Kudan. Code is provided in the Appendix C (tracking and rendering)

6.3.1 Rendering

AR SDK comes with a built-in rendering engine hidden behind a Java API for its preparation. Kudan defines all virtual objects that belong to the virtual map, constructed and updated by Kudan's VSLAM-based tracking engine (further *Kudan's VSLAM* or *tracking engine*), as nodes of different kind (ie *ARImageNode.class*,

[31] https://www.kudan.eu/kudan-2d-3d-recognition/
[32] https://github.com/kudan-eu

ARModelNode.class) to better manipulate them on high-level abstractions. Each type of node also has a set of attributes configurable through a number of methods. In order to properly construct virtual objects for rendering, following classes aid the developer with that task:

- **ARModelImporter.class** - an instance of given class allows to load a model from file with a special *.jet* format[33] (to avoid compression and loss of model's quality as a result[19]) and create a model node to represent a virtual object
- **ARModelNode.class** - an instance of given class is an object representation of the model described in the .jet asset file. The model node consists of many meshes which represent a single drawable entity. In order to style or apply effects (ie so that model could reflect more light) to the model, material must be provided to each mesh. **ARMeshNode.class** is an object representation of mesh
- **ARLightMaterial.class** - an instance of given class is an object representation of the material that defines the behavior of the model with the surrounding virtual environment resulting in different effects under different light conditions (ie light intensity) during rendering. Additionally, Kudan's material encapsulates texture data used in model's styling. Usually, material's file comes along with the model's file, but in case of Kudan, material of the model is defined programmatically[19]
- **ARTexture2D.class** - an instance of given class is an object representation of model's texture and is coupled with the material
- **ARImageNode.class** - an instance of given class in an object representation of image

6.3.2 Tracking

Kudan's tracking engine is equipped with the ability to track an arbitrary place of real world without using markers which differs from ARToolKit, that required a preregistered marker and would not perform any rendering before detecting one. However, a compulsory step, forcing user to select an area of real world and approve it as a target by tapping the screen, is added to the application's flow. Once chosen, Kudan's VSLAM starts its work, by maintaining a virtual map of real environment and performing localization to estimate virtual camera's state. All virtual objects also become eligible for rendering on detection and do not require target to be detected first. Following classes own a significant role in tracking initialization and management:

[33]For conversion from common format to the required one, author used Kudan's conversion GUI tool

- **ARGyroPlaceManager.class** - instance of given class is a sensor-only (GYROscope sensor) tracker that manages state of the target the user is about to approve. The central abstraction introduced by Kudan during target positioning is *virtual floor's plane* - is an imaginary plane of floor (in this case floor can be interpreted differently: room's floor, table's surface etc...) the target belongs to. Target can only move along the virtual plane and by rotating the real camera the orientation and position of it is changed. Distance to the virtual plane also aids in achieving realistic scale, so that any virtual object with configured dimensions would be rendered approximately to the scale[20]. The distance from the camera to the floor is by default set to 1.5m, but according to the task may be changed[34] (Fig. 23)
- **ARArbiTrack.class** - instance of given class natively communicates [35] with KudanCV's VSLAM. Once user approves state of the target, the next goal of VSLAM is to create an empty virtual map, add virtual representation of real camera (virtual camera that models the state of the real camera) with predefined virtual plane to it and lock the registered target to some point on the virtual plane. After virtual map creation, rest of the tracking process starts:
  - Feature extraction
  - Feature registration
  - Virtual map maintenance
  - Virtual camera's localization

  ARArbiTrack also provides an abstraction of *world* inside virtual map it covers. World contains virtual objects to be rendered on virtual camera detecting them. World is an analogy for scene used in 3D engines[36] (Fig. 24)

[34]In case of a comparison, a floor is the working surface and distance from camera to it was calculated using following formula: authorsHeight (1.76m) - deviceHoldingOffset (0.2m) - distanceFromFloorToSurface (0.52m) = distanceFromCameraToSurface (1.04m). Approximate measurements are made with the help of a retractable flexible rule and might have a non-critical error in it

[35]Through calling native methods

[36]A 3D space to hold and manipulate models, textures, materials, lighting

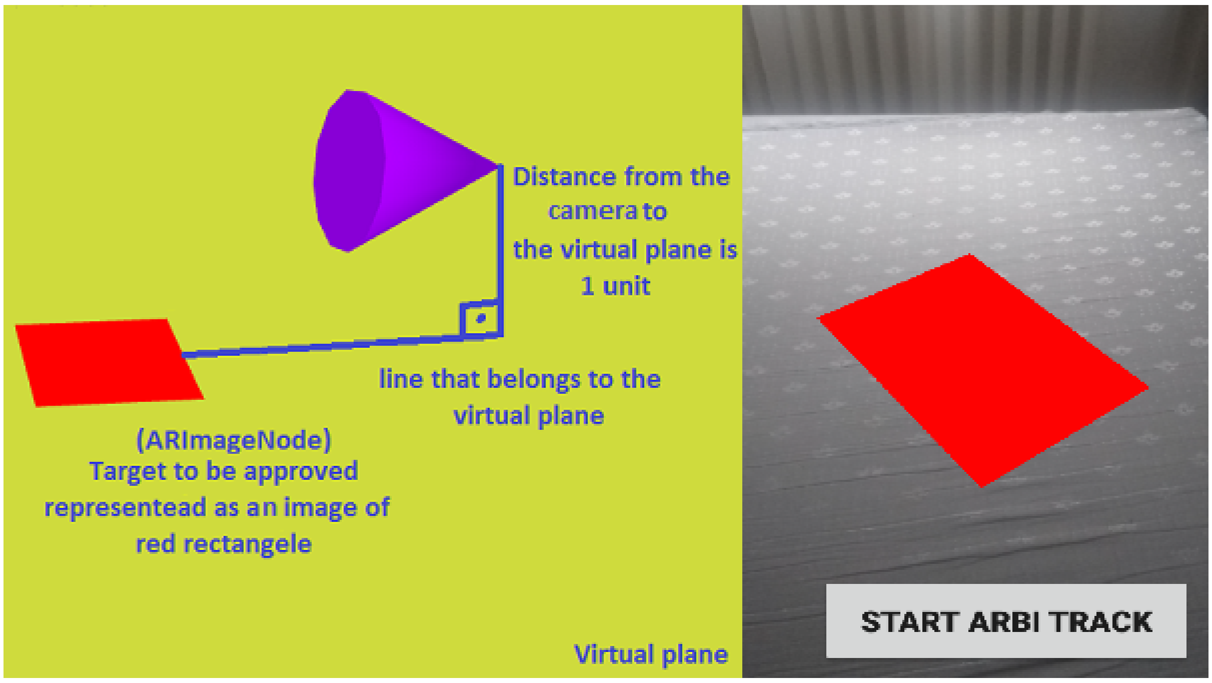
Distance from the camera to the virtual plane is 1 unit
line that belongs to the virtual plane
(ARImageNode) Target to be approved representead as an image of red rectangele
Virtual plane
START ARBI TRACK

Figure 23: After the note texture was generated, the user is offered to select an area of the real world to become a target and be tracked. The scale between real world and virtual map that represents it becomes 1 unit = 1.04m according to author's calculations

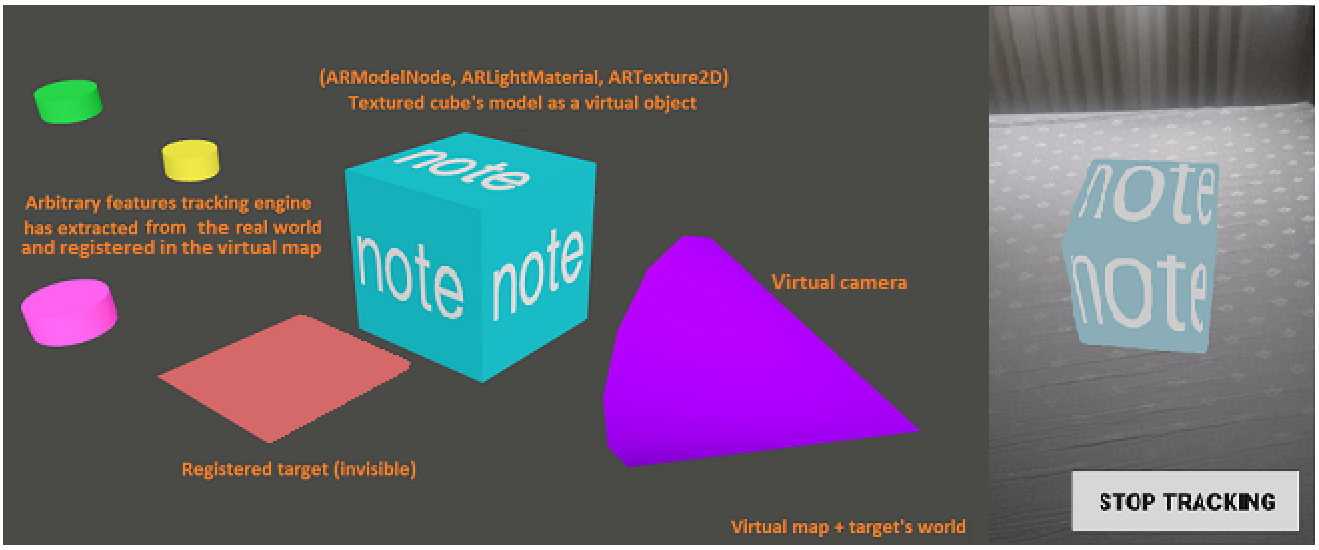
(ARModelNode, ARLightMaterial, ARTexture2D)
Textured cube's model as a virtual object
Arbitrary features tracking engine has extracted from the real world and registered in the virtual map
Virtual camera
Registered target (invisible)
Virtual map + target's world
STOP TRACKING

Figure 24: In Kudan, maintained virtual map is merged with target's world, so that virtual camera is able to render virtual object even if target is out of sight. Figure also shows presence of some extracted features registered in the virtual map (human is also able to distinguish them in real world: straight lines, corners etc... - but sometimes it is hard, as they are usually tiny)

6.4 Build preparation

Kudan Java SDK's .aar file comes with a pre-compiled library for ARMv7 CPU architecture. From past experience, it is known that the test mobile device owns a CPU with architecture of ARMv8-A, which is compatible with object files compiled for ARMv7 architecture[21].

6.5 Kudan results

Here the author provides the test results of running the integrated test application according to the predefined metrics.

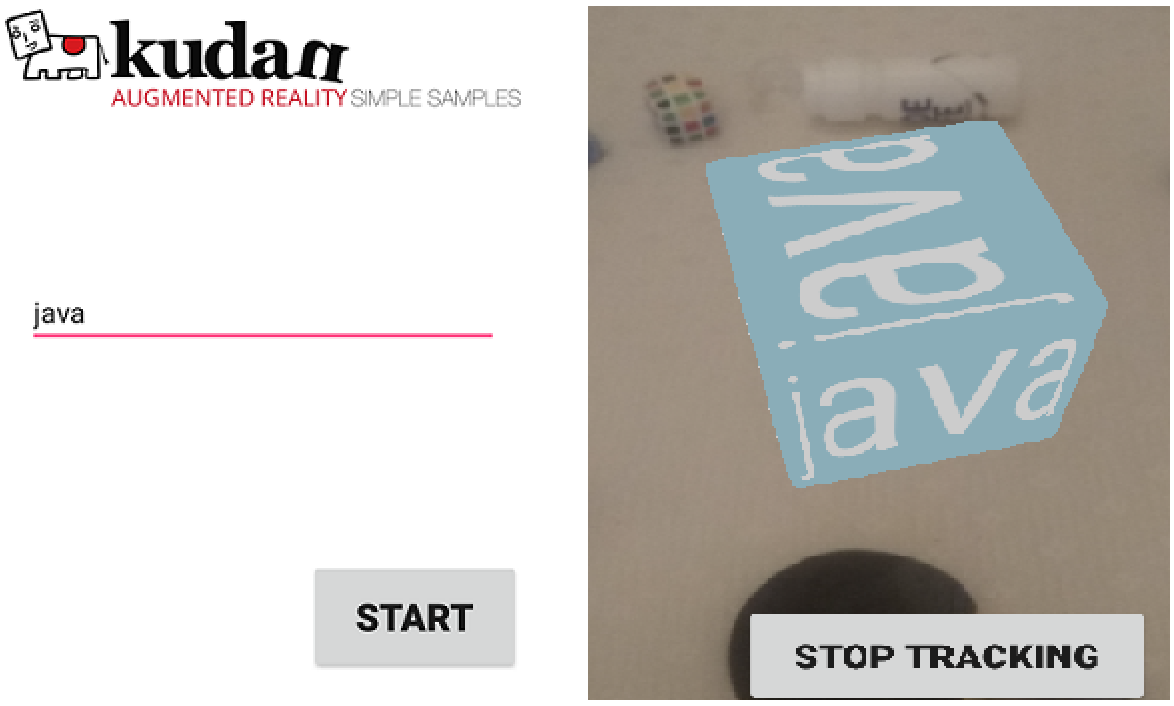

Figure 25: Application start

1. **General behavior of rendered virtual cube object (further object):** object had no convulsions. Result: *1p*

2. **Behavior during fast camera shifts:** object started changing its position and scale (Fig. 26). Result: *0p*

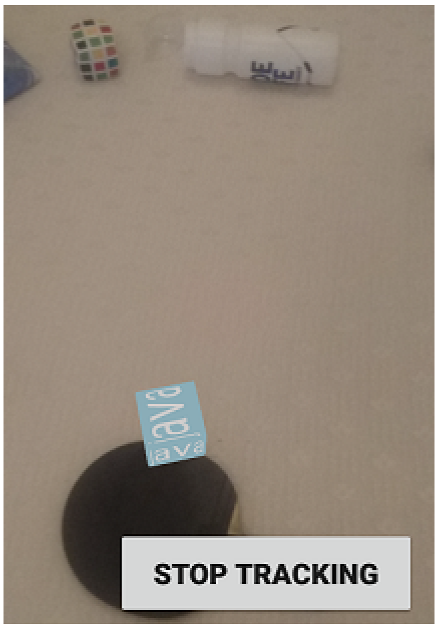

Figure 26: Behavior during fast camera shifts

3. **Behavior on being very close to the object:** expected behavior (Fig. 27). Result: (1/2)p = *0.5p*

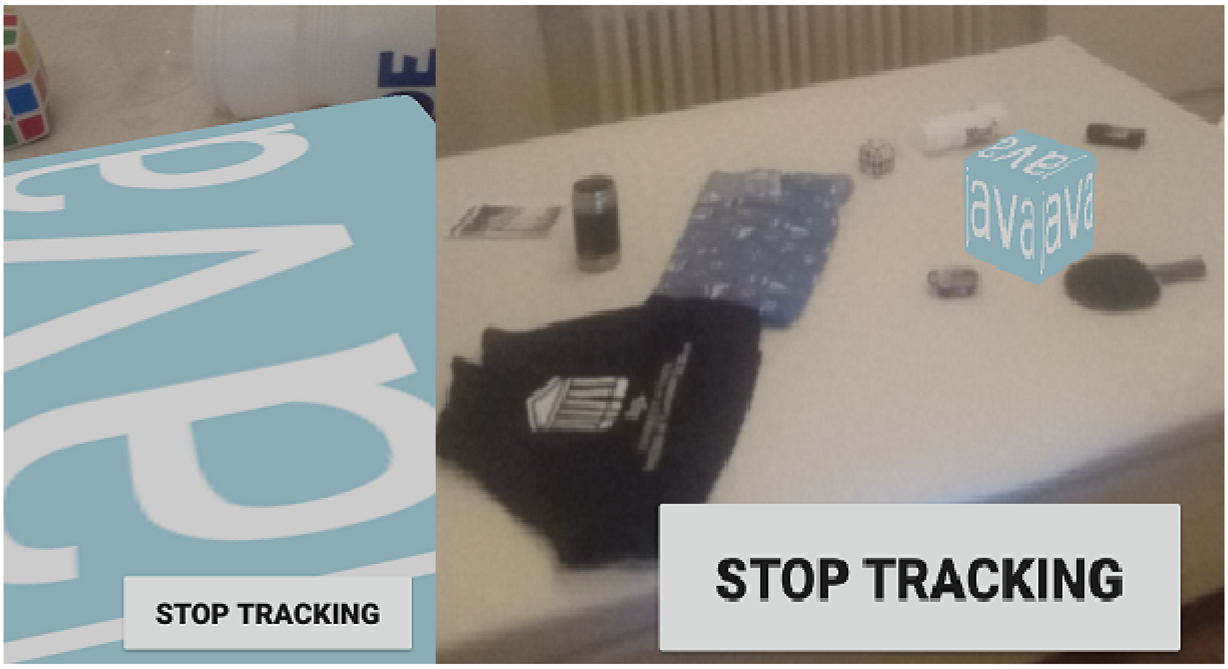

Figure 27: Behavior on being very close and far away (relatively to the room size) from the object

4. **Behavior on being far away (relatively to the room size) from the object:** expected behavior (Fig. 27). Result: (1/2)p = *0.5p*
5. **Behavior on camera rotating around the object:** expected behavior (Fig. 28). Result: (1/4)p*4 = *1p*

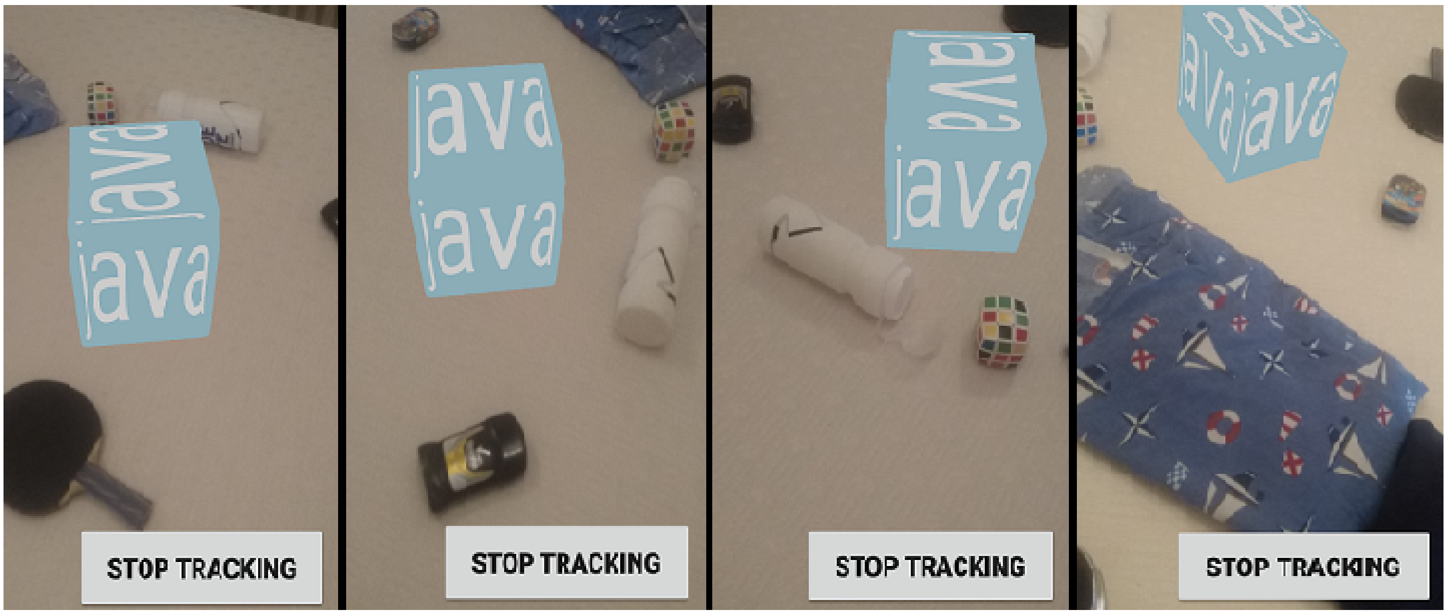

Figure 28: Rotating camera around the object

6. **Behavior on camera turning away from and back to the object (slow) :** expected behavior. Result: (1/2)p = *0.5p*
7. **Behavior on camera turning away from and back to the object (fast) :** object disappeared. Result: *0p*
8. **Behavior on walking away to another room with camera and turning back to the object:** object disappeared. Result: *0p*
9. **Existence of examples and sample projects:** there are only a few sample projects provided by the ARToolKit, but they are useful[37]. Penalty of 30% is applied. Result: 1p*(1-0.3) = *0.7p*
10. **Existence of toolkit overview:** amount of toolkit related information is minimal, but has a wiki page. Penalty of 50% is applied. Result: 1p*0.5 = *0.5p*
11. **Existence of well-structured Java API documentation:** exists[20]. Result: *1p*
12. **Amount of (the most important) code lines that are spent on preparing rendering:** 15 loc (ARToolKit has 15 loc, Wikitude has 34 loc) . Result: *1p*

[37] https://github.com/kudan-eu/

13. **Amount of (the most important) code lines that are spent on configuring tracking:** 13 loc (ARToolKit has 10 loc, Wikitude has 23 loc). Result: 10/13 = *0.77p*

14. **Complexity of development and integrating test application's source code into the sample project in hours:** 2 hours for all routines (ARToolKit has 4 hours). Result: *1p*

Total: **8.47p**

6.6 Conclusion

This chapter made an overview of how to work with Kudan. To start with, it was noticed that Kudan's public resources lack information regarding its architecture and system description. Author had to search, filter and put up together the information retrieved from other Kudan's sources, like wiki[38] and blog[39]. About twice a month posts regarding AR, tracking methods and SLAM, VSLAM in particular appear on the blog. Information placed there was found useful, as helped in making some concepts regarding markerless tracking clear[16].
Kudan's wiki, Android SDK's API and some information regarding its tracking engine aided the author in describing abstractions, met during test application's integration. Those abstractions reflect author's understanding of Kudan and are based on mentioned supporting material. Illustrations were also provided.
Test application's integration was accomplished in relatively short time, due to the availability of effectively made sample project. During testing, it was figured out that Kudan has weaknesses related to cameras' fast movement (Kudan results, tests 2 and 7). The disappointment faced, was found in an inability of Kudan to render virtual object when switching the rooms (Kudan results, test 8). During this test, author was moving the camera slowly, trying to extract as many features as possible, in order to enrich the virtual map with registered features. Unfortunately, it has not changed the end result. Ability to maintain a large virtual map is important for VSLAM-based tracking.
This leads to the conclusion that Kudan would be a suitable tool to develop indoor applications for the specific room:

- Interior design: ie augmenting virtual model of furniture, in order to figure out whether it suits the room's style

- Entertainment: ie AR games that are playable inside the specific room

[38]https://wiki.kudan.eu/Main_Page
[39]https://www.kudan.eu/blog/

7 Wikitude

In this chapter a VSLAM-based Wikitude toolkit[40] is explored. Initially Wikitude was oriented to B2C[41] business model focusing on development of their Wikitude application. Later it was changed to B2B[42] model focusing on Wikitude SDK, AR engine and AR tools (ie tool for converting model file to the correct format [43] and online studio for creating AR content[44]).

7.1 Wikitude's architecutre

Wikitude SDK provides both Native and JavaScript APIs on a variety of different platforms (Unity, Android, iOS, Titanium[45] etc...) for both markerless and marker-based tracking. Besides indoor tracking, Wikitude SDK is capable of providing outdoor AR experience[24]. Previously reviewed libraries had an architecture overview, so does Wikitude.
The Native API is completely different from the native API concept used before, as Wikitude SDK's Native API is the API with the development language of the target SDK (ie Android Java). This allows to develop Wikitude SDK based AR applications avoiding using APIs not based on development's primary programming language, style and pattern. Every application is built above those API's either straight or using intermediate-connection plugin, module or component (ie Unity with Unity Plugin and Titanium platform with Titanium module for JavaScript API)[22]. Using Wikitude SDK, developer gets a high-level control over configuring tracking and preparing rendering. Both APIs make use of a Wikitude's Plugins API, which in turn gives access to some original and 3rd party Java, ObjC[46] or C++ written plugins each having an unique identifier and extending initial Plugins API base class *Plugin.class*. Plugins integrate custom computer vision, AR and camera functionality [23]. Plugins' activity is integrated into the application's lifecycle after being registered by the Wikitude's Computer Vision engine (CV engine).
Both Native and JavaScript APIs are built above the CV engine native C++ layer. A Plugins API also holds a connection with CV Engine for more flexible custom 3rd party plugin development. The responsibility of CV Engine includes:

1. OpenGL ES rendering

[40]https://www.wikitude.com/wikitude-slam/
[41]Business to Client
[42]Business to Business
[43]Wikitude 3D Encoder
[44]Wikitude Studio
[45]A platform for creating both mobile and desktop cross-platform applications using JavaScript, HTML and CSS
[46]Objective-C language for writing software for iOS and OSX operating system

2. 3D recognition[47] and tracking
3. 2D recognition and tracking
4. 2D cloud recognition[48]
5. Plugin management

CV Engine further makes use of hardware abstractions (ie camera) and optimizations (GPU, CPU) provided by the platform's operating system. In case of Android Platform many hardware optimizations are made for the main CPU architectures: ARMv7 and ARMv8-A.

7.2 Developing test application

Wikitude provides sample projects along with the SDK[49] for the required platform. Project related to instant tracking (for both APIs) has been chosen to be a skeleton project for integrating test application.
The Native API has no support of comfortable 3D rendering as provides a raw OpenGL ES context requiring many algebraic and geometry transformations to be done manually. However, JavaScript API has an internal 3D rendering functionality that avoids usage of OpenGL ES, therefore JavaScript API is used for comparison. Unfortunately, JavaScript API is able to load model, material and texture all at once and only from the file with special *.wt3*[50] format. As a result, loading texture from file during run-time is not possible and for comparison a predefined cube model with texture is prepared by the author and converted to .wt3 format (with the help of Wikitude's Encoder).
Also, author decided to show the complexity of working with the raw OpenGL ES context by implementing a shortened variation of the initial test application. The reason behind that, are the values returned from the CV engine during tracking which lead to wrong behavior of virtual object in case of rotating camera around it. Misbehavior was figured out practically, during application testing.

7.3 JavaScript API

Primary abstractions of working with JavaScript API on Android SDK are Architect World, Architect View and Architect Activity:

[47] Using predefined image/model/pattern for tracking
[48] All calculations are performed on the remote recognition server
[49] https://www.wikitude.com/download/
[50] A compressed file format by Wikitude for describing models, optimized for fast loading and handling by the CV engine

- **Architect World** - is interchangeably used with the term 'AR experience in WEB', as encapsulates configurable information related to models, tracking and other resources associated with AR inside an abstract world of HTML and JavaScript files[25]. It is developed similarly to a web application to enable a cross-platform AR. It also holds a communication with Android SDK through *ArchitectView.class*[51] and *ArchitectJavaScriptInterfaceListener.class*[52]. In project is represented as a *World.JSclass* instance in the JavaScript source code with supporting HTML files

- **Architect View** - is in response for rendering and communication with the Architect World. Additionally, stores application's state, provides application context and establishes native communication with CV engine. In project's source code is represented as an *ArchitectView.class*

- **Architect Activity** - it handles the rest of work on configuring, event listening and maintaining the application. In project's source code is represented as an *AbstractArchitectCamActivity.class*

Before development start, author had an opinion that switching rendering preparation and tracking configuration environment from Java to JavaScript would be a sufficient trade-off. After making changes to the source code and testing the application, it was figured out that it was rather a positive experience and should be treated more as a web application development. Code is provided in the Appendix D (tracking and rendering).

7.3.1 JavaScript API rendering

Following JavaScript API classes are common in preparing basis for rendering:

- **ImageResource.JSclass** - an instance of given class loads an image from file

- **ImageDrawable.JSclass** - encapsulates the instance of ImageResource and uses it during rendering

- **Model.JSclass** - loads a model from the file with format .wt3 and represents it, also allowing to configure some attributes of the model (ie rotation, translation)

7.3.2 JavaScript API tracking

Despite the fact, that Wikitude's and Kudan's tracking engines differ internally, they still have same basic concept of VSLAM in common. User must be able to select an area of

[51]In case SDK is calling the Architect World
[52]In case Architect World is calling SDK

the real world as a target to be tracked before starting tracking. In case of Kudan, virtual objects are part of the virtual map from the very start of the tracking process. However, during Wikitude's tracking created virtual map is free of virtual objects and only contains virtual camera and registered features. User is able to add virtual objects, manually by tapping the screen during tracking process.

- **InstantTracker.JSclass** - an instance of given class represents connection with markerless tracking algorithm[26] based on VSLAM method provided by the CV engine. Without specifying the target, it concentrates on extracting and registering arbitrary features, maintaining the virtual map and carrying out localization of the virtual camera
- **InstantTrackable.JSclass** - an instance of given class represents a target image (also visible during tracking), virtual objects that are connected with it and an algorithm to perform tracking (InstantTracker). Listeners are also registered there, to listen for events from CV engine's tracking module (further *tracking engine*)

7.4 Native API

If in case of JavaScript API, developer focuses of configuring rendering and tracking in the JavaScript file of Architect World, then with Native API the developer besides manual configuration is additionally responsible for defining all of the OpenGL ES rendering steps using data from tracking engine. Native API gives the developer more freedom, but also makes the development more complex. However, JavaScript API is less rich in functionality, but keeps the development simple and compact. Code for Native API is not provided in the thesis for two reasons: 1. It is not participating in the comparison 2. It has an enormous amount of lines of code (about 177 loc only for rendering cube). However, author provides a link for the BitBucket repository in Appendix E (WikitudeNativeDemoRepo), where source code author has integrated into the sample project is provided.

- **InstantTrackingActivity.class** (specific to sample project) - an instance of given activity class allows to setup rendering component, listen for events on detecting the virtual object by the tracking engine and providing received data to the rendering component (further *renderer*)

Must be mentioned, that Kudan and Wikitude's JavaScript API were able to detect virtual objects and did not require target for that. Target was simply a start point to build a virtual map from. In case of Wikitude's Native API, virtual objects are rendered only then when the target is detected by the tracking engine which means that those virtual objects belong to the target.

7.4.1 Native API rendering

Wikitude and ARToolKit (ARToolKit's ARBaseLib structure) have similar flow on displaying virtual objects to the screen in case of OpenGL ES context:

1. Create an OpenGL ES renderer, which might reference a number of virtual objects constructed and prepared for rendering
2. Create view for displaying received from camera frame of the real world
3. Create view (*GLSurfaceView.class*) for rendering virtual objects on
4. Stack render view above the frame view in the view group (*FrameLayout.class*), so that it would look like virtual object is augmented into the real world
5. Both have a background job running, that would make calls to renderer to start rendering process with period of 0.033333 (0.03333 seconds per frame = 30 FPS[53]) which is time passed between two calls

The test application's virtual object is a cube, model of which was previously loaded from the file, but now the model must be programmatically constructed and prepared using OpenGL ES.
In real life, when photographer is willing to take a photo of the building's other side, a photographer's movement to required position is necessary as the only other option would be to rotate the building to face the needed side to the photographer which is impossible due to many physical factors. In computer graphics, the photographer is a static camera and the illusion of camera movement is caused by transformations applied to the virtual objects, as with computers there are no crucial factors that would disallow to do that.
Cube has six faces, each face is split into 2 polygon triangles each having 3 sides (further *vertices*). The total amount of triangles is 6 * 2 = 12, therefore total amount of vertices is 12 * 3 = 36. Trio of vertices defines a triangle and how it is positioned, oriented. Fragment is in response for the style and defines color for each pixel inside the triangle (Fig. 29). Texture is also split and applied to fragments. Given calculations are done in manually written shaders[54] as a part of rendering pipeline. Transformations are applied to the virtual object's vertices. Transformation and virtual object's vertices are represented as matrices, product of which would be a new matrix defining virtual object's vertices with applied transformation.
States of virtual map's known objects (target, virtual objects, virtual camera and registered features) are all important for building correct transformation matrix. It allows to render the virtual object connected with the detected target properly from the

[53]Frames Per Second
[54]Special helper programs written in C like language - GLSL

perspective of virtual camera's current state, achieving maximum realistic AR experience.

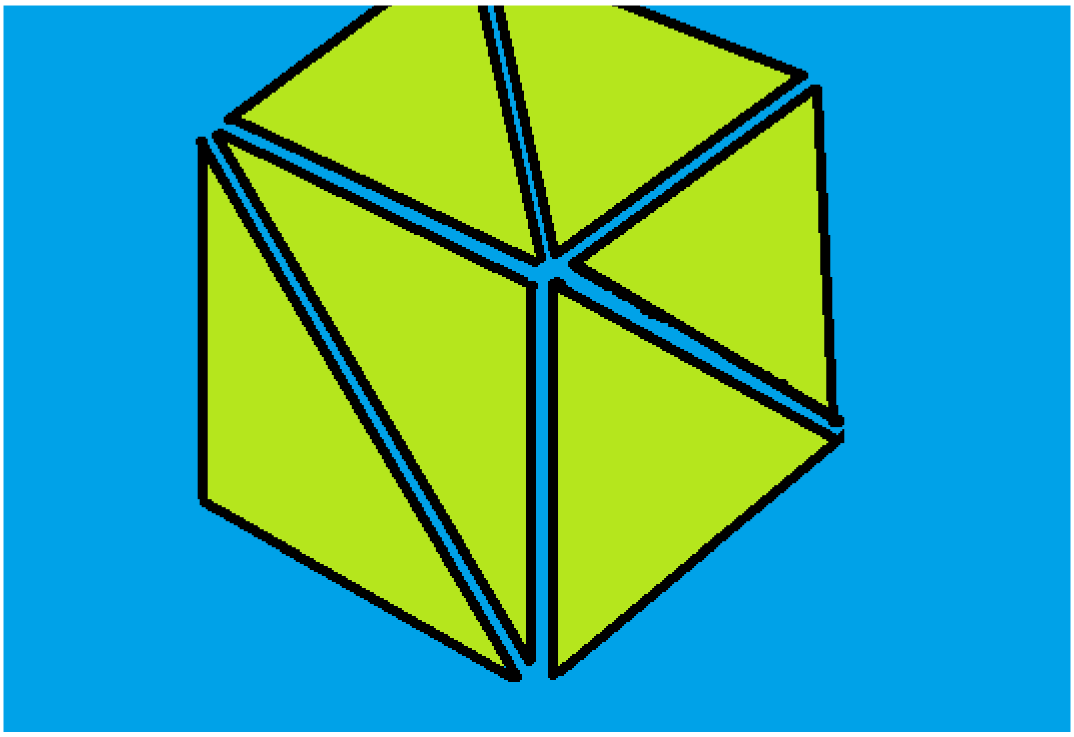

Figure 29: Black lines are vertices which make up a triangle polygon. Polygon consists of units that give it a color - fragments (green pixels)

7.4.2 Native API tracking

While the tracking algorithm works, it constantly performs localization to figure out virtual camera's state and mapping to update virtual map's known objects previously estimated states. Once the target is detected by the virtual camera, tracking engine performs calculations based on it's current state to produce transformation matrix. The result matrix is returned as an event that is later detected by the event listener from the InstantTrackingActivity which supplies it further to the renderer which references virtual objects. In order to get the latest transformation matrix from tracking engine, proper configuration has to be done and listeners have to be set:

- **WikitudeSDK.class** - an instance of given class creates the tracker, natively connects it to the tracking engine and registers a listener (InstantTrackingActivity) for it to receive new data. Once target is detected, the CV engine supplies to the renderer an instance of class implementing **Target.interface** that gives access to an encapsulated newly constructed transformation matrix from the last virtual camera's state. On next render cycle, renderer will use this data to perform drawing of referenced virtual objects to the render view (GLSurfaceView)

7.5 Build preparation

Wikitude also comes with a pre-compiled C++ library that suits test device.

7.6 Wikitude results

Here the author provides the test results of running the integrated test application according to the predefined metrics.

1. **General behavior of rendered virtual cube object (further object):** object had no convulsions. Result: *1p*

2. **Behavior during fast camera shifts:** expected behavior. Result: *1p*

3. **Behavior on being very close to the object:** object disappeared (Fig. 30). Result: *0p*

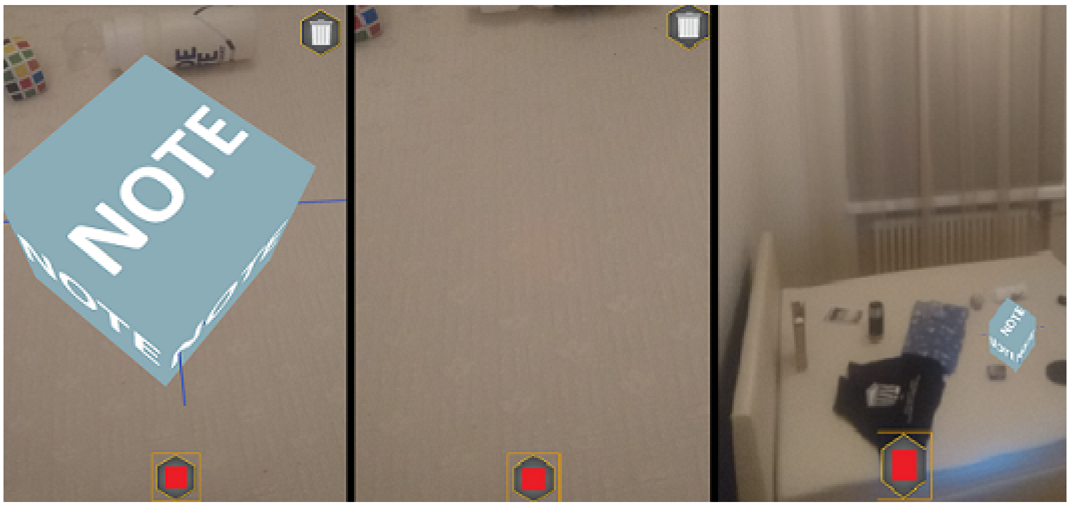

Figure 30: Object is not rendered even if camera goes slightly closer to it

4. **Behavior on being far away (relatively to the room size) from the object:** expected behavior (Fig. 30). Result: (1/2)p = *0.5p*

5. **Behavior on camera rotating around the object:** object would not render on focusing the back face (Fig. 31). Result: (1/4)p*3 = *0.75p*

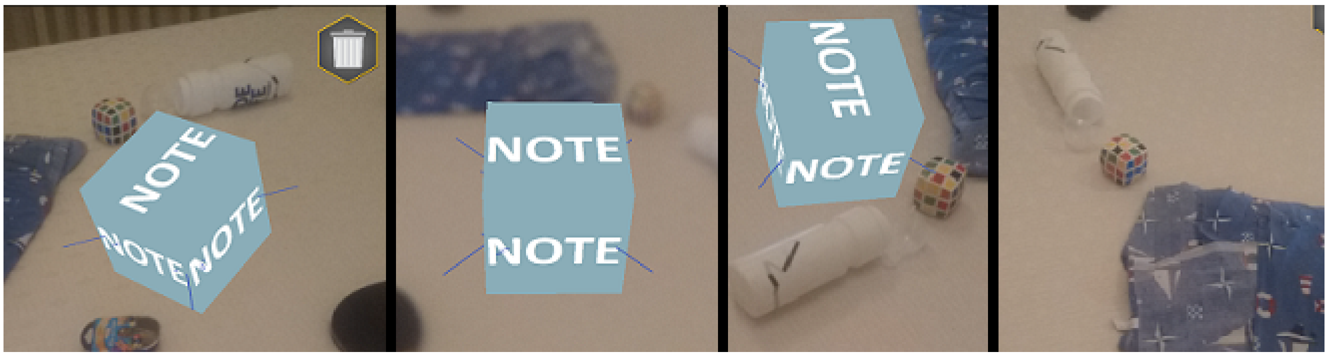

Figure 31: Object was not rendered when camera focused the back face of it

6. **Behavior on camera turning away from and back to the object (slow) :** expected behavior. Result: (1/2)p = *0.5p*

7. **Behavior on camera turning away from and back to the object (fast) :** expected behavior. Result: (1/2)p = *0.5p*

8. **Behavior on walking away to another room with camera and turning back to the object:** expected behavior. Result: *1p*

9. **Existence of examples and sample projects:** there are a lot of sample projects provided by the Wikitude[55]. Result: *1p*

10. **Existence of toolkit overview:** a lot of main and additional information about the product is provided. Result: *1p*

11. **Existence of well-structured Java API documentation:** exists[56]. Result: *1p*

12. **Amount of (the most important) code lines that are spent on preparing rendering:** 34 loc (ARToolKit has 15 loc, Kudan has 15 loc). Result: 15/34 = *0.44p*

13. **Amount of (the most important) code lines that are spent on configuring tracking:** 23 loc (ARToolKit has 10 loc, Kudan has 13 loc). Result: 10/23 = *0.44p*

14. **Complexity of development and integrating test application's source code into the sample project in hours:** 0.5 of an hour. Due to the fact that texture loading at run-time is not supported by the Wikitude, author had to construct models manually. Fairness is important, that is why this result is not able to participate in the comparison. Result: *0p*

[55]Examples that are a part of the SDK
[56]http://www.wikitude.com/external/doc/documentation/latest/Reference/Android%20Native%20SDK%20API/index.html

Total: **9.13p**

7.7 Conclusion

This chapter made a detailed overview of how to work with the Wikitude. Both JavaScript API and Native API of Wikitude's Android SDK were studied. Primary abstractions were provided that reflect the author's view of Wikitude's AR high-level abstract world. Public resources and theory behind VSLAM aided in that.
At the very beginning, author was planning to integrate test application using the Native API, in order to keep Android Java as a primary language and show off how complicated it is to work with OpenGL ES. The goal was achieved and a virtual object with custom texture was rendered, but with a significant trade-off. The behavior of the virtual object was unexpected on rotating the camera around it. Virtual object would not take the required form, always showing only three faces. It was figured out, that problem comes from the transformation matrix returned by the tracking engine. Author was not able to perform any actions related to the data returned by the tracking engine, as it is out of scope of given thesis.
As a result, JavaScript API was chosen to participate in the comparison. Problem with JavaScript API is that it does not support the run-time loading of textures. Author had to manually prepare a model with texture for testing. During testing, it was figured out that Wikitude had only one significant weakness: virtual object was disappearing on the camera being close to it (Wikitude results, test 3). Also, an interesting fact is that the virtual object was not rendered from one out of four camera states (Wikitude results, test 5). This test does not state, that Wikitude is not capable of rendering virtual object on full rotation around it all the time. Rest of the tests were passed by the Wikitude achieving competitive end results.
Taking all of that into account, Wikitude would be a suitable toolkit to develop a variety of different applications, including the ones mentioned for the other toolkits.

8 Conclusion

To remind, at the beginning author has set the goal to answer following questions: "How to choose, test, compare and use the right tool for AR application development?", "How to understand and work with tool SDK's API?", "What are the primary abstractions during developing AR application?". Author managed to achieve that goal, by answering all of the stated in the Motivation questions. Reviews provide a high-level architecture overview, important aspects of working with particular toolkit Android SDK's API, primary abstractions offered by the author (based on a variety of toolkit's public resources and author's experience), development details and testing results.
Received test results finally allow to compare the toolkits with each other. To recap, in total, ARToolKit had 7.55p, Kudan had 8.47p and Wikitude had 9.13p. According to the numbers, Wikitude would be the most suitable toolkit to integrate the test application into. However, should be recalled that Wikitude JavaScript API did not achieve completely the goal of the defined test application's integration, as does not support the run-time texture loading. Most of the points it received from a strong visual performance. An opinion of author exists, that Kudan would be a more suitable toolkit for the integration, as not only satisfied the needs for it, but also had an acceptable visual behavior and informational basis.
Concluding the thesis, selection of the toolkit depends on the type and the scale of the problem. All toolkits were provided with potential areas to be used in. Wikitude had the broadest field to be applied in. However, ARToolKit and Kudan are also important.

9 References

References

[1] Randall C.Smith, Peter Cheesman. *On the Representation and Estimation of Spatial Uncertainty*.
The International Journal of Robotics Research. 1986.

[2] Ronald T. Azuma. *A Survey of Augmented Reality*.
In Presence: Teleoperators and Virtual Environments. 1997.

[3] R. Silva, J. C. Oliveira, G. A. Giraldi. *Introduction to Augmented Reality*.
National Laboratory for Scientific Computation. 2003.

[4] D.W.F. van Krevelen and R. Poelman. *A Survey of Augmented Reality Technologies, Applications and Limitations*.
The International Journal of Virtual Reality. 2010.

[5] Khalid Yousif, Alireza Bab-Hadiashar, Reza Hoseinnezhad. *An Overview to Visual Odometry and Visual SLAM: Applications to Mobile Robotics*.
Intelligent Industrial Systems. 2015.

[6] Cesar Cadena, Luca Carlone, Henry Carrillo, Yasir Latif. *Past, Present, and Future of Simultaneous Localization And Mapping: Towards the Robust-Perception Age*.
IEEE Transactions on Robotics. 2016.

[7] Stephen Tully, George Kantor, Howie Choset. *A Single-Step Maximum A Posteriori Update for Bearing-Only SLAM*.
Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence. 2010.

[8] Niko Sünderhauf, Peter Protzel. *Towards a Robust Back-End for Pose Graph SLAM*.
IEEE Intl. Conf. on Robotics and Automation. 2012.

[9] Georg Klein, David Murray. *Parallel Tracking and Mapping for Small AR Workspaces*.
Proceedings of the 2007 6th IEEE and ACM International Symposium on Mixed and Augmented Reality. 2007.

[10] ARToolKit team. *About ARToolKit*.
http://artoolkit.org/. Last Accessed: 10.05.2017

[11] ARToolKit team. *Developing with ARToolKitWrapper and ARBaseLib*.
https://artoolkit.org/documentation/doku.php?id=4_Android:android_developing.
Last Accessed: 10.05.2017

[12] Android team. *Android Platform Guide*.
https://developer.android.com/guide/index.html?. Last Accessed: 09.05.2017

[13] Urassl. *Structure from Motion-classical realisation*
https://habrahabr.ru/post/228525/. Last Accessed: 02.04.2017

[14] Andrew J. Davison, Ian D. Reid, Nicholas D. Molton and Olivier Stasse.
MonoSLAM: Real-Time Single Camera SLAM.
IEEE Transactions on Pattern Analysis and Machine Intelligence. 2007.

[15] Rodrigo Munguia, Antoni Grau. *Monocular SLAM for Visual Odometry: A Full Approach to the Delayed Inverse-Depth Feature Initialization Method*.
Mathematical Problems in Engineering. 2012.

[16] Kudan team. *Scale in Simultaneous Localisation and Mapping*.
https://www.kudan.eu/kudan-news/scale-simultaneous-localisation-mapping/. Last Accessed: 20.04.2017

[17] Kudan team. *KudanCV vs. OpenCV for AR*.
https://www.kudan.eu/kudan-news/kudancv-vs-opencv-ar/. Last accessed: 05.05.2017

[18] Kudan team. *Kudan framework* https://wiki.kudan.eu/Framework. Last Accessed: 10.05.2017

[19] Kudan team. *Kudan 3D models*.
https://wiki.kudan.eu/3D_Models. Last accessed: 03.05.2017

[20] Kudan team. *Kudan Android Java API documentation*.
https://wiki.kudan.eu/apidocs/AndroidDocs/index.html. Last Accessed: 10.05.2017

[21] Peter Harris(ARM team). *ARMv8 backwards compatibility with ARMv7*.
https://community.arm.com/processors/f/discussions/4798/armv8-backwards-compatibility-with-armv7. Last Accessed:
20.04.2017

[22] Wikitude team. *Wikitude SDK Architecture*
https://www.wikitude.com/external/doc/documentation/latest/android/gettingstartedandroid.html#getting-started. Last Accessed: 10.05.2017

[23] Wikitude team. *Preview on the wikitude SDK Plugins API*.
https://www.wikitude.com/blog-preview-on-the-wikitude-sdk-plugins-api/. Last Accessed: 10.05.2017

[24] Wikitude team. *Wikitude SDK features*.
http://www.wikitude.com/products/wikitude-sdk/. Last Accessed: 10.05.2017

[25] Wikitude team. *Augmented Reality Wikitude SDK*.
https://components.xamarin.com/gettingstarted/com.wikitude.xamarin.component. Last Accessed: 09.05.2017

[26] Wikitude team. *Wikitude JavaScript API*.
http://www.wikitude.com/external/doc/documentation/latest/Reference/JavaScript%20API/index.html. Last Accessed: 10.05.2017

[27] Richard A. Newcombe, Steven J. Lovegrove and Andrew J. Davison. *DTAM:Dense Tracking and Mapping in Real-Time*.
IEEE International Conference on Computer Vision. 2011.

[28] *Pokemon GO*.
http://www.pokemongo.com/. Last Accessed: 10.05.2017

[29] *Mixed Reality*.
http://elatewiki.org/index.php/Mixed_Reality. Last Accessed: 08.05.2017

[30] *Virtual Imaging for the Medical Industry*.
https://www.mdtmag.com/article/2013/04/virtual-imaging-medical-industry. Last Accessed: 08.05.2017

[31] *Augmented Reality*.
http://csis.pace.edu/~marchese/DPS/Lect3/dpsl3.html. Last Accessed: 10.04.2017

[32] *Types of augmented reality*.
https://augmentedrealityresearch.wordpress.com/2011/10/11/types-of-augmented-reality/. Last Accessed:
15.04.2017

[33] *Microsoft Hololens*.
http://virtualnyeochki.ru/obzoryi/microsoft-hololens-obzor-innovaczionnyix-ochkov-dopolnennoj-realnosti-ot-microsoft. Last Accessed:
15.04.2017

[34] *Google Glass*.
http://iseekmate.com/976-istoriya-sozdaniya-i-funkcii-google-glass.html. Last Accessed: 15.04.2017

[35] Stephen Tully, George Kantor, Howie Choset. *A Single-Step Maximum A Posteriori Update for Bearing-Only SLAM*.
Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence. 2010.

[36] Kudan team. *Different Types of Visual SLAM Systems*. https://www.kudan.eu/kudan-news/different-types-visual-slam-systems/. Last Accessed: 08.05.2017

[37] Sanni Siltanen. *Theory and applications of marker-based augmented reality*. http://www.vtt.fi/inf/pdf/science/2012/S3.pdf. Last Accessed: 10.05.2017

[38] *Hiro pattern*. https://github.com/artoolkit/artoolkit5/blob/master/doc/patterns/Hiro%20pattern.pdf. Last Accessed: 10.02.2017

[39] *Augmented reality for architects*. http://www.augment.com/augmented-reality-architecture/. Last Accessed: 10.05.2017

[40] *Mobile Visual Odometry and Ego-Motion Estimation for Virtual Walkthroughs* https://angel.co/projects/123834-mobile-visual-odometry-and-ego-motion-estimation-for-virtual-walkthroughs?src=more_projects. Last Accessed: 10.05.2017

10 Appendix A: Note texture creation code

```
/* A more problem specific version of
 * stackoverflow.com/questions/9973048/convert-text-to-image-file-on-android
 */
private String createNote(String text) {
   text = text.trim().replace("\n", "").replace("\r", "").toLowerCase();
   /* TextPaint represents transformations to be performed on the
    * styled-by-default text in order to
    * make it look according to the needs.
    */
   final TextPaint textPaint = new TextPaint() {
       {
         //Style text
         setColor(Color.WHITE);
         setTextAlign(Paint.Align.LEFT);
         setTextSize(200f);
         setAntiAlias(true);
       }
   };
   // Following border rectangle defines bounds of the text
   final Rect bounds = new Rect();
   textPaint.getTextBounds(text, 0, text.length(), bounds);
   // Manually figured out in practice offset. May vary.
   int newOffset = 150;
   /* The RGBA(Red-Green-Blue-Alpha(Transparency)) bitmap
    * as a light-blue colored background.
    * Use 4x8 bit ARGB format for better quality
    */
    final Bitmap bmp = Bitmap.createBitmap(bounds.width()+newOffset,
                    bounds.height()+newOffset,
                    Bitmap.Config.ARGB_8888);
    bmp.eraseColor(Color.rgb(173,216,230));
    /* Canvas based on bitmap in order to perform drawing
     * over encapsulated bitmap
     */
    final Canvas canvas = new Canvas(bmp);
    // Draw the text the way it would be close to the center as possible
    canvas.drawText(text,
           newOffset/2,
            (bounds.height() + newOffset)/2 + (bounds.height() + newOffset)/8,
            textPaint);
    //Temporary directory
    File outputDir = getApplicationContext().getCacheDir();
    File outputFile = null;
    try {
       outputFile = File.createTempFile("generated_note_texture",
                                          ".png", outputDir);
       FileOutputStream stream = new FileOutputStream(outputFile);
       bmp.compress(Bitmap.CompressFormat.PNG, 85, stream);
       /* Bitmap object contains data in native memory and
        * it takes a few cycles to be garbage collected,
        * so in case of application where many Bitmap
        * objects are created, without manual recycling
        * it might result in performance decrease or crash
        */
       bmp.recycle();
       stream.close();
   } catch (FileNotFoundException e) {
       e.printStackTrace();
   } catch (IOException e) {
       e.printStackTrace();
   }
   return outputFile.getAbsolutePath();
}
```

11 Appendix B: ARToolKit code

11.1 Rendering

```java
private Object3D getCube() throws IOException {
  TextureManager.getInstance().addTexture("tex",
    new Texture(
      new FileInputStream(getIntent().getStringExtra("noteAbsPath"))));
  int scale = 30;
  Object3D object3D = Loader.loadOBJ(
                          getAssets().open("test-cube.obj"),
                          null, scale)[0];
  object3D.setTexture("tex");
  object3D.rotateX((float) Math.PI/2);
  object3D.rotateY((float) Math.PI);
  object3D.rotateZ((float) Math.PI);
  object3D.setOrigin(new SimpleVector(300, -50, scale*4));
  return object3D;
}
```

11.2 Tracking

```java
protected void populateTrackableObjects(List<TrackableObject3d> list) {
  try {
    TrackableObject3d tckobj = new TrackableObject3d(
                                   "single;Data/hiro.patt;80",
                                   getCube());
    list.add(tckobj);
  } catch (IOException e) {
    e.printStackTrace();
  }
}
```

12 Appendix C: Kudan code

12.1 Rendering

```
private  ARModelNode modelNode;
private  void addModelNode() {
  ARModelImporter modelImporter = new ARModelImporter();
  modelImporter.loadFromAsset("test-cube.jet");
  modelNode = modelImporter.getNode();
  ARTexture2D texture2D = new ARTexture2D();
  texture2D.loadFromPath(getIntent().getStringExtra("noteAbsPath"));
  ARLightMaterial material = new ARLightMaterial();
  material.setTexture(texture2D);
  material.setAmbient(0.8f, 0.8f, 0.8f);
  for (ARMeshNode meshNode : modelImporter.getMeshNodes()) {
    meshNode.setMaterial(material);
  }
  modelNode.scaleByUniform(30f);
}
```

12.2 Tracking

```
public void setupArbiTrack() {
  ARImageNode targetImageNode = new ARImageNode("target.png");
  targetImageNode.scaleByUniform(0.1f);
  targetImageNode.rotateByDegrees(90, 1, 0, 0);
  ARGyroPlaceManager gyroPlaceManager = ARGyroPlaceManager.getInstance();
  gyroPlaceManager.initialise();
  gyroPlaceManager.getWorld().addChild(targetImageNode);
  gyroPlaceManager.setFloorDepth(-(176F - 20 - 52F));
  ARArbiTrack arbiTrack = ARArbiTrack.getInstance();
  arbiTrack.initialise();
  arbiTrack.setTargetNode(targetImageNode);
  arbiTrack.getWorld().addChild(modelNode);
}
```

13 Appendix D: Wikitude code

13.1 Rendering

```
var World = {
    init: function initFn() {
        this.createOverlays();
    },
    createOverlays: function createOverlaysFn() {
        var crossHairsRedImage =
            new AR.ImageResource("assets/crosshairs_red.png");
        var crossHairsRedDrawable =
            new AR.ImageDrawable(crossHairsRedImage, 1.0);
        var crossHairsBlueImage =
            new AR.ImageResource("assets/crosshairs_blue.png");
        var crossHairsBlueDrawable =
            new AR.ImageDrawable(crossHairsBlueImage, 1.0);
    ...
    addModel: function addModelFn(xpos, ypos) {
        if (World.isTracking()) {
            var model = new AR.Model("assets/models/test-cube.wt3", {
                scale: {
                    x: 0.7,
                    y: 0.7,
                    z: 0.7
                },
                translate: {
                    x: xpos,
                    y: ypos
                },
                rotate: {
                    z: 45
                },
            })
            allCurrentModels.push(model);
            this.instantTrackable.drawables.addCamDrawable(model);
        }
    },
    ...
};
```

13.2 Tracking

```
...
    this.tracker = new AR.InstantTracker({
        onChangedState:  function onChangedStateFn(state) {},
        deviceHeight: 1.04,
        onError: function(errorMessage) {
            alert(errorMessage);
        }
    });
    this.instantTrackable = new AR.InstantTrackable(this.tracker, {
        drawables: {
            cam: crossHairsBlueDrawable,
            initialization: crossHairsRedDrawable
        },
        onTrackingStarted: function onTrackingStartedFn() {},
        onTrackingStopped: function onTrackingStoppedFn() {},
        onTrackingPlaneClick: function onTrackingPlaneClickFn(xpos, ypos) {
            World.addModel(xpos, ypos);
        },
        onError: function(errorMessage) {
            alert(errorMessage);
        }
    });
},
...
World.init();
```

14 Appendix E: other resources

14.1 Modelling

ClaraIo - models for comparison and examples provided by the author are created (or downloaded) using given online software. Author's profile link is https://clara.io/user/alanduUT

14.2 Bitbucket

Provided repositories store the toolkits' sample projects with integrated test application. They are explicitly made private, in order to retain the toolkit owner's rights. Author is able to provide the access for the educational purpose only, if requested:

- ARToolKit demo - https://bitbucket.org/Alandur/artoolkitdemorepo
- Kudan demo - https://bitbucket.org/Alandur/kudandemorepo
- Wikitude Native API demo - https://bitbucket.org/Alandur/wikitudenativedemorepo