



DIPLOMATERVEZÉSI FELADAT

Balogh Tamás

Mémők informatikus hallgató részére

Kiterjesztett valóság alapú szövegfordító alkalmazás Android platformon

Az elmúlt évtizedben az okostelefonok meghódították a világot, és ma már terjeszkedésük ütemét szinte csak a táblagépek képesek felülmúlni. Köszönhetően a különféle nyílt platformoknak, a felhasználók egy asztali számítógép árának töredékéért juthatnak olyan tényéymyi eszközökhöz, melyek számítási kapacitása az egy évtizeddel ezelőtti számítógépeket is túlszárnyalja.

Ez lehetőséget nyújt az alkalmazás fejlesztőknek arra, hogy a korábban csak az asztali számítógépekben használt funkcionalitást biztosítsák az okostelefon felhasználók számára. Viszont a számítási kapacitás mellett az okostelefonok számos olyan szenzorral rendelkeznek, mellyel a PC-k nem. Ezeket kombinálva az okostelefonok adta mobilitással számos új terület nyílik meg az alkalmazás fejlesztők előtt.

A feladat több ilyen új alkalmazási területbe is bepillantást nyújt, mint a számítógépes szövegfordítás, optikai karakter felismerés és a kiterjesztett valóság. Ezen területek napjainkban is aktív kutatási területek, számos ipari szereplő jelen van a különböző területeken, viszont a területeket kombináló alkalmazások száma csekély.

A hallgató feladatának a következőkre kell kiterjednie:

- A felsorolt kutatási területek megismerése, az aktuális képességek, lehetőségek bemutatása
- Mutassa be és hasonlítsa össze a célplatformra elérhető optikai karakter felismerést megvalósító komponenseket, válassza ki közülük a feladatához legmegfelelőbbet
- Szövegfordítási szolgáltatáshoz való kapcsolódás megvalósítása, a fordított szövegek gyorsítótárazását ellátó komponens implementálása
- Tervezze meg és implementálja a cél nyelvű szöveg megjelenítéséért felelős szoftver komponensét. A szoftver támogassa a Magyar-Angol, valamint az Angol-Magyar szövegfordítási irányokat
- Biztosítsa a kompatibilitást a régebbi Android verziók számára is, a lehetőségekhez mérten
- Tesztelje, és dokumentálja az elkészített szoftvert

Tanszéki konzulens: Fehér Marcell

Budapest, 2013. március 11.

Dr. Vajk István
egyetemi tanár
tanszékvezető





M Ű E G Y E T E M 1 7 8 2

Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Automatizálási és Alkalmazott Informatikai Tanszék

Kiterjesztett valóság alapú szövegfordító alkalmazás Android platformon

DIPLOMATERV

Készítette

Balogh Tamás

Konzulens

Dr. Ekler Péter, Fehér Marcell

2014. december 12.

Tartalomjegyzék

Kivonat	5
Abstract	6
Bevezető	7
1. Specifikáció	14
2. Irodalomkutatás	17
2.1. Kereskedelmi megoldások	17
2.1.1. Waygo	17
2.1.2. CamDictionary	17
2.1.3. Babelshot	18
2.1.4. Bing Translator	19
2.1.5. Word Lens	20
2.2. Akadémiai kutatások	20
2.3. Értékelés	21
3. Optikai karakterfelismerési lehetőségek	22
3.1. Elérhető megoldások	22
3.1.1. Vuuforia	23
3.1.2. Tesseract	23
3.2. A Tesseract használata	24
3.2.1. A könyvtár csatolása az Android projekthez	25
3.2.2. A könyvtár lefordítása	25
3.2.3. Nyelvi fájlok kezelése	26
4. Szövegfordító szolgáltatások	28
4.1. Webszolgáltatások	29
4.1.1. Google Translate API	29
4.1.2. Gengo API	29
4.1.3. WebserviceX - Translation Engine	29
4.1.4. Microsoft Translator	29
4.2. Autentikáció	30
4.2.1. Token frissítés	32

4.3.	Csatlakozás	32
4.3.1.	Aszinkron kommunikáció	33
4.3.2.	Broadcast események	34
4.4.	Gyorsítótárazás	36
5.	Képfeldolgozás	38
5.1.	Kamerakezelés	38
5.1.1.	A Preview megjelenítése	40
5.2.	Kapcsolódás a Tesseracthoz	42
5.2.1.	A hOCR feldolgozás	42
5.2.2.	Adatstruktúra	43
5.2.3.	Adatszűrés	44
5.2.4.	Aszinkron karakterfelismerés	45
5.3.	A Tesseract könyvtár kiegészítése	47
6.	Kiterjesztett valóság	50
6.1.	Felület	50
6.2.	Kirajzolás	51
6.2.1.	A háttér megjelenítése	52
6.2.2.	Szövegszín detektálás	52
6.3.	Főképernyő	55
6.3.1.	Navigáció	55
6.3.2.	Események	56
7.	Tesztelés	57
7.1.	Manuális tesztek	57
8.	Konklúzió	60
8.1.	Továbbfejlesztési lehetőségek	60
8.1.1.	Mondatok fordítása	60
8.1.2.	További nyelvek támogatása	60
8.1.3.	Vizuális attribútumok bővítése	61
8.1.4.	Fordítási lehetőségek kibővítése	61
	Köszönetnyilvánítás	62
	Ábrák jegyzéke	64
	Táblázatok jegyzéke	65
	Irodalomjegyzék	65
	Függelék	71
F.1.	Telepítési útmutató	71
F.2.	Felhasználói kézikönyv	73

F.2.1. Nyelvváltás	73
F.2.2. Fókuszálás	73
F.3. Tesztelési eredmények	74

HALLGATÓI NYILATKOZAT

Alulírott *Balogh Tamás*, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2014. december 12.



Balogh Tamás
hallgató

Kivonat

Az elmúlt évtizedben az okostelefonok rohamos fejlődésének lehettünk tanúi. Ennek hála, a mai telefonok és táblagépek kapacitása már megelőzi az egy évtizeddel ezelőtti számítógépeket és a köztük lévő határ folyamatosan elmosódik. A számításigényes megoldások és technológiák, melyek régebben csak asztali számítógépekben, vagy nagy számítási kapacitással rendelkező beágyazott rendszerekben voltak elérhetőek, mára az okostelefonokon is felhasználhatóak. Ezeket a technológiákat kombinálva a telefonok és táblagépek nyújtotta mobilitással és érzékelőkkel, lehetőséget sora nyílik meg az alkalmazásfejlesztők előtt.

A dolgozat egy okostelefon alkalmazás tervezését és fejlesztését mutatja be, mely ilyen technológiákat és megoldásokat ötvöz, és kiegészíti őket a telefonok és táblagépek nyújtotta új lehetőségekkel. Az alkalmazás a telefon kamerája által látott képen felismeri a szöveges tartalmakat, azokat lefordítja, majd megjeleníti a felhasználónak. A képről a szöveg tartalmát *optikai karakterfelismerés* segítségével ismeri fel, majd *számítógépes szövegfordítással* fordítja le a látott tartalmat. A lefordított szöveg és az eredeti szöveg jellemzőinek birtokában a *kiterjesztett valóság* segítségével megjeleníti a kijelzőn az eredeti képet, de felé rajzolva a lefordított szöveggel. A lefordított tartalom elfedve az eredeti nyelvű szöveget azt a hatást kelti, mintha eredetileg is a lefordított szöveget látnánk a telefon kameráján keresztül.

A dolgozat röviden bemutatja ezeket a technológiákat és értékeli a *kiterjesztett valóságot* felhasználó hasonló megoldásokat. Ezután az elérhető *optikai karakterfelismerést* támogató lehetőségek és azok alkalmazása kerül bemutatásra, majd a szövegfordítást végző komponens leírása következik. Ezután következik a kamerakép feldolgozás, valamint a *kiterjesztett valóságot* felhasználó, megjelenítő komponens bemutatása. A dolgozatot a teszteléshez használt megoldások leírása és az elkészült megoldás értékelése zárja.

Abstract

In the past decade, smartphones have showed great progress. Today, their computing capacity is higher than the desktop computers had a decade ago, and this difference is still decreasing. The computationally intensive technologies and solutions, which were only available in desktop environment or embedded systems, are accessible on smartphones today. These technologies combined with the mobility and the sensors provided by phones and tablets, allow number of opportunities for application developers.

This paper covers the design and development of an application, which combines technologies like these, and extends them with the new potentials provided by smartphones. The application recognizes the text content on the camera image, translates it, and presents the results to the user. For detection the application uses *optical character recognition*, then translates the content with *computer-assisted translation*. With the translated content and the visual attributes of the original text, the application uses *augmented reality* to display the original image with the translated text. The translated content covers the original one, in order to achieve the effect of presenting the translated text in the environment of the original content.

The paper introduces these technologies briefly and evaluates the similar solutions supported by *augmented reality*. It details the available solutions with *optical character recognition* support and the application of this technology, then the translator module is discribed. After that, the camera image processing and the *augmented reality* supported presentation module are detailed. In the end, the application testing and the evaluation of the finished solution can be found.

Bevezető

Tagadhatatlan, hogy napjaink egyik legjobban fejlődő informatikai-technológiai iránya az okostelefonok mind szoftveres, mind hardveres szempontból. Töretlen fejlődésüknek hála, számítási kapacitásuk már megközelíti, sőt esetenként túl is szárnyalja az egy évtizeddel ezelőtti asztali számítógépekét. Ez a fejlődési ütem azonban nem látszik alábbhagyni a jövőben sem, így az okostelefonok és az asztali számítógépek közötti teljesítménykülönbség egyre csökken.

Azok a technológiák, melyek régebben csak asztali környezetben kerültek felhasználásra, már elérhetőek okostelefonos, illetve táblagépes formában is. Az okostelefonok már önmagukban is hatalmas fejlesztési lehetőségeket rejtenek, köszönhetően mobilitásuknak és számos szenzoruknak (kamera, GPS, gyorsulásmérő stb.), melyekkel az asztali számítógépek nem rendelkeznek. Azonban ezeket ötvözve a számításigényes, korábban csak asztali környezetben használt technológiákkal, lehetőségek sora nyílik meg az alkalmazásfejlesztők előtt.

A platform

Az Android mobil operációs rendszer 2008-as megjelenése óta folyamatosan egyre nagyobb részesedéssel rendelkezik a mobil, és a közben megjelenő táblagép piacon is. Mára (2014 második negyedév) már a legnagyobb okostelefon platformmá nőtte ki magát, és világszerte közel 85%-os részesedéssel vezeti a mobileszköz eladási számokat [50]. Az elmúlt hat év alatt a platform jelentős fejlődésen ment keresztül, mely nem csak a felhasználók, de a fejlesztők előtt is számos lehetőséget nyitott meg. Az évek során nem csak a fejlesztéshez használható funkciók, könyvtárak bővültek, de számos új fejlesztői eszköz is elérhetővé vált. Ilyen például a közel másfél éve megjelent *Android Studio IDE* [18], melyet a feladat implementációja során én is használtam.

A többi mobil operációs rendszerrel összehasonlítva, a fejlesztők számára a legnyitottabb platformnak ez tekinthető (Linux alapokra épült, akár a teljes rendszer és az fejlesztői könyvtárak forráskódja is elérhető, mely nagy segítség lehet a fejlesztés során), ezért is szerencsés, hogy a feladatkiírásban szereplő feladatot ezen az operációs rendszeren kell megvalósítanom.

Az Android alkalmazásfejlesztői módszertan alapvetően az Android *SDK* (*Software Development Kit* [17]) használatára épít, mely tartalmazza a fejlesztők számára elérhető könyvtárakat, és lehetővé teszi, hogy ezek felhasználásával *Java* nyelvű alkalmazásokat készítsenek. Ezek a *Java* nyelvű alkalmazások, a fordítás során egy speciális *bytekódra*

fordulnak, melyet futtatás során az Android futtatókörnyezete értelmez. Eltérően a *Java Standard Edition*től, az Android saját futtatókörnyezetet használ. Bár évekig a *Dalvik Runtime* volt az Android egyetlen futtatókörnyezete, az Android 4.4 verzióval megjelent *ART* futtatókörnyezet is elérhető [36], sőt a nemrég megjelent 5.0-ás verzió már csak ezt támogatja. Bár fejlesztői szempontból is vannak különbségek a két rendszer között, természetesen munkám során mindkettőt támogatom, tekintve, hogy még mind a mai napig a telefonok nagy része a *Dalvik* futtatókörnyezettel rendelkezik, de ez a szám pár éven belül meglehetősen csökkenni fog. Tekintve, hogy az alkalmazások fejlesztése *Java* nyelven történik, a már meglévő, más *Java-alapú* környezetben fejlesztett komponensek, könyvtárak (pl. *Google GSON* [44]), melyek nem függenek az adott platformtól, Androidon is újrafelhasználhatóak.

Az Android lehetővé teszi, hogy a fejlesztők bizonyos komponenseket a futtatókörnyezetet kikerülve, magán az eszköz Linux alapú rendszerén futtassák. Ehhez az Android *NDK* (*Native Development Kit*) [16] nyújt lehetőséget, mely lehetővé teszi, hogy *C++* nyelven írt, majd magára az eszköz processzorára fordított binárisokat a *Java* nyelvű alkalmazások részeként elérjük. Bár az alkalmazások többségében erre nincs szükség, viszont előfordulhatnak olyan esetek, amikor a számítási kapacitás növelése érdekében, érdemes a futtató környezetet megkerülni, illetve olyan eset is, amikor egy már *C++* nyelvek elkészült komponenszt szeretnénk újrafelhasználni, ahogy az ezen alkalmazás fejlesztése során is történt.

Technológiai háttér

A feladatkiírásban szereplő alkalmazás, több technológia kör és kutatási terület megoldásait is felhasználja. Ezek a területek, melyek mind olyan számítási kapacitást igénylő folyamatokra épülnek, melyek felhasználása eddig csak számítógépes, esetleg speciális dedikált hardverek vagy perifériák segítségével volt megvalósítható. Az okostelefonok elterjedése tette lehetővé ezen területek mobilis felhasználását, és elterjedését a mindennapi életben. Ezen alkalmazás három ilyen technológiát is felhasznál, melyek a *kiterjesztett valóság* (**A**ugmented **R**eality), az *optikai karakterfelismerés* (**O**ptical **C**haracter **R**ecognition) és a *számítógépes szövegfordítás* (**C**omputer -**A**ssisted **T**ranslation).

Kiterjesztett valóság

Az alkalmazás által leginkább felhasznált technológia a *kiterjesztett valóság*. Bár ez a terület az okostelefonok megjelenése előtt is létezett, mégis az okostelefonok adták meg a lendületet ezen terület fejlődésének, és a köznapi életben való elterjedésének. A *kiterjesztett valóság* rendszerek többsége jellemzően valamilyen szemüveg, vagy speciális periféria használatára épült, mely képolvasásra és megjelenítésre is képes volt. Tekintve, hogy az okostelefonok ezen képességekkel is rendelkeznek, így alkalmasak voltak ezeknek a kiváltására. Bár az okostelefonos irány ma egyre elterjedtebben alkalmazott, a speciális periféria alapú vonal is aktív fejlesztés alatt van (pl. *Google Glass* [43]), habár még nem annyira terjedt el ezek mindennapi alkalmazása.

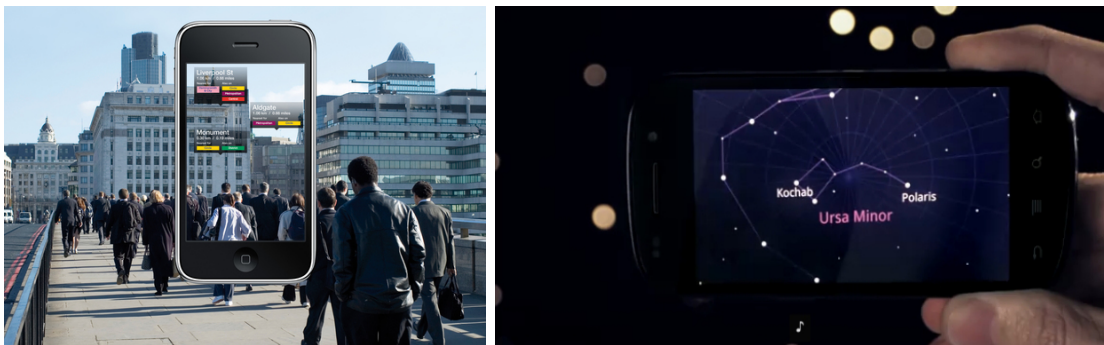
A *kiterjesztett valóság* lényege, hogy a valós világról érzékelhető információkat valami-



1. ábra. A Google Glass, a kiterjesztett valóságot támogató okoszemüveg. [6]

lyen, a világgal kapcsoltban álló, de általunk nem feltétlenül, vagy nem abban a formában érzékelhető információkkal egészítjük ki. Bár a technológiának nincs pontos definíciója, és nem is szorítkozik csak vizuális információkra, a gyakorlatban azonban ezek szinte kivétel nélkül valamilyen képi megjelenítésen alapulnak. A képalapú *kiterjesztett valóság* lényege, hogy a látható képre, melyet az okostelefonos esetben jellemzően a telefon/táblagép kamerája szolgáltat, valamilyen egyébként nem látható információt rajzolunk, mely a valósággal szoros kapcsolatban áll. Ezáltal kiterjesszük a valóságot, és azt az érzetet keltjük, mintha a telefonon keresztül olyan dolgokat látnánk, melyek egyébként nem láthatóak, de mégis kapcsolódnak valahogy a látott képhez.

Ma számos olyan okostelefonos alkalmazás elérhető, melyek valamilyen módon kihasználják a *kiterjesztett valóságot*. Bizonyos alkalmazások a GPS pozíciót és a telefon iránytűjét használják fel, ilyen például, *AcrossAir* [13], mely a kameraképre vetíti ki a néző irányában található turisztikai információkat, vagy a *Google Sky Map* [45], mely a csillagképeket jeleníti meg az ég felé tartva.



2. ábra. Az AcrossAir és a Google Sky Map használat közben. [1] [7]

Más alkalmazások pedig a kameraképet magát dolgozzák fel és az adja a többlet információ forrását. Ilyen például a *Layar* [56], mely speciális nyomtatványokhoz, kiadványokhoz ad plusz információkat, de léteznek akár játékok is, melyek a kameraképen talált jelzőkódokat felhasználva lehetővé teszik, hogy a játék háromdimenziós alakzatait például az asztalunkon jelenítsük meg (pl. *AR Defender* [51]). Természetesen a plusz információ forrása ezektől eltérhet, illetve keverhetőek a különböző megoldások is.

Jelen alkalmazás is természetesen a *kiterjesztett valóság* okostelefon alapú ágát erősíti, azon túl pedig a kameraképet feldolgozó irányba tartozik, ugyanis a fordítás forrása a képen



3. ábra. Az Layar és az AR Defender alkalmazások, a kiterjesztett valóság széleskörű felhasználásának példái. [8] [2]

található szöveg, és ennek a lefordított változata az előzőt eltakarva kerül kirajzolásra.

Optikai karakterfelismerés

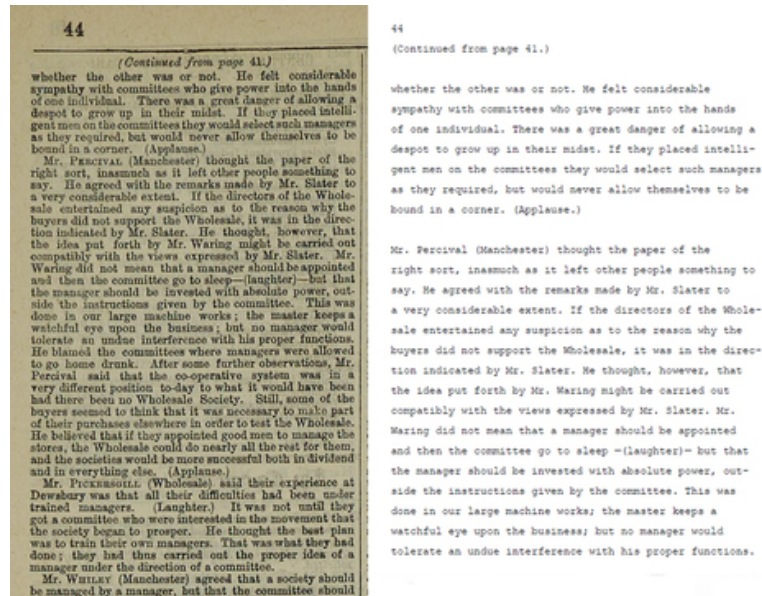
A másik terület melybe a dolgozat bepillantást nyújt, és amelyet szintén felhasznál az alkalmazás, az *optikai karakterfelismerés*. A szöveg digitális tárolásának több formája is ismeretes, viszont jelen esetben két jól elkülöníthető változat különböztethető meg. Egyfelől a szöveget lehetőség van karakteres formában tárolni, ekkor a karakterek karaktertáblázat szerinti kódja van tárolva, és ezeket lehetőség van megjeleníteni különböző méretben, betűtípussal, elrendezésben, mivel ez a formátum az informatikai rendszerek által jól kezelhető. Természetesen ez a megoldás az legelterjedtebb.

Másfelől viszont lehetőség van a szöveget képként tárolni digitálisan, ekkor a szövegről készült valamilyen fotót, esetleg a számítógép által generált képet tároljuk. Az előző esettel ellentétben, itt nem a karakterek kódja van tárolva, hanem ténylegesen a kép képpontjai (*pixel*ek). Ebben az esetben természetesen nem lehet a szöveg méretet, betűtípusát stb. állítani, illetve karakterenként módosítani a szöveget. Ez a megoldás főként azokban az esetekben használatos, amikor valamilyen kézzel írt vagy kinyomtatott dokumentumot fotózás/*scanelelés* után tárolunk.

Az első formából a másodikba triviálisan lehet konvertálni, ki kell nyomtatni, vagy a háttértárra előállítani a szöveg nyomtatott változatát. Viszont a másik irány egyáltalán nem triviális, mivel ott a *pixel*eket kell karakterekké alakítani, tehát a képről kell a karaktereket felismerni. Ezzel a problémakörrel foglalkozik az *optikai karakterfelismerés*.

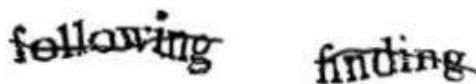
A folyamat szempontjából mindegy, hogy a kép a kamera képe, *bescannelt* kép, vagy a számítógép által készített kép (viszont az olvasás minősége ez nagyban befolyásolhatja). A módszer lényege, hogy a képet a beolvasás után, képfeldolgozási algoritmusokkal a megfelelő formára alakítja (*normalizálás*, *binarizáció* stb.), elkülöníti a szöveget az egyéb képi tartalomtól, majd kép/alakzatfelismerési módszereket használva karakterenként megpróbálja felismerni az egyes betűket. Ezeket később szavakká, mondatokká fűzi, és így áll elő a karakteres tartalom. Bizonyos megoldások a felismert szöveg mellett azok pozícióját, illetve egyéb vizuális jellemzőit (betűtípus, szín stb.) is képesek felismerni.

A több évtizedes fejlesztés ellenére, még közel sem tökéletesek ezek a technológiák, és általános esetben nem feltétlenül elfogadható az olvasási eredmény hibaráta. Mi sem bi-



4. ábra. Az optikai karakterfelismerés általános felhasználásnak egy példája. Bal oldalon az olvasott kép, jobb oldalon pedig a detektált szöveg. [9]

zonyítja ezt jobban, mint az, hogy napjainkban is *CAPTCHA*-t (Completely Automated Public Turing test to tell Computers and Humans Apart) használnak a különböző webes beviteli űrlapok védelmére. A *CAPTCHA* lényege, hogy egy olyan karaktereket tartalmazó kép tartalmát kell felismeri, melyet egy ember könnyedén elolvas, szemben egy számítógéppel, mely erre csak korlátozottan képes. Természetesen ez a kép kimondottan úgy van előállítva, hogy az optikai karakterolvasók által legnehezebben kezelhető esetek tömkelege felhasználásra kerüljön, így is nehezítve az algoritmusok munkáját. Ebből kifolyólag, az általános megoldások felhasználása során mindig figyelembe kell venni, az adott eset speciális vonásait, és azokat kihasználni a jobb eredmény elérése érdekében (pl. kontextus specifikus szűrés stb.).



5. ábra. Egy példa a CAPTCHA megoldásokban felhasznált képekre. [5]

Jelen megoldás a kamera képén látható szavakat használja fel a fordítás forrásaként. Mivel ezek az utóbbi típusba tartoznak a szövegtárolás formáját tekintve, viszont a fordításhoz a szövegekre nyilván karakteres formában van szükség, ehhez az átalakításhoz az alkalmazás az *optikai karakterfelismerést* használja fel.

Számítógépes szövegfordítás

A *számítógépes szövegfordítás* napjainkra egy eléggé elterjed technológia, mind asztali, mind okostelefonos környezetben is találkozhatunk vele. A rengeteg online és mobil szótár mellett számos olyan megoldás is elérhető mely dokumentumokból, weboldalokról olvassa be a lefordítani kívánt szövegeket, leegyszerűsítve így a fordítási procedúrát. Bár az első szá-

mítógépes fordító megoldások csak a megfelelő szavakat cserélték le egy szótár adatbázis segítségével, a ma használatos megoldások természetesen ettől jóval összetettebb algoritmust használnak. A mondatok struktúráját, a különböző mondat szerkezeteket és szóösszetételeket is figyelembe veszik a ma elterjedt megoldások.

Ez a terület is folyamatosan fejlődik, habár ma még a mindennapi életben inkább csak kiegészítő eszközként használható. A ma elterjedt megoldások, különösen a magyar nyelvről, illetve magyar nyelvre történő esetben még annyira pontatlanok, hogy mindenképpen valamilyen emberi felülvizsgálatot igényelnek.

A feladat specifikációban kitűzött funkcionalitás elérése érdekében, a képen látható szövegek lefordításához, egy *számítógépes szövegfordítást* támogató megoldást használok fel.

Tematika

Ezen bevezető után a dolgozat további struktúrája 3 részre osztható. A fejlesztés előtti kutatási és tervezése lépéseket az 1. és a 2. fejezet mutatja be. Ezután a megoldás részletes bemutatása következik a 3., 4., 5., 6. és 7. fejezetben. A dolgozatot a 8. fejezet zárja az elkészült megoldás értékelésével.

Az **1. fejezet (Specifikáció)** a feladatkiírásban szereplő előírások és követelmények értékelését, valamint azok szoftverspecifikációként való értelmezését tartalmazza.

A **2. fejezet (Irodalomkutatás)** a hasonló funkcionalitással rendelkező megoldásokat vizsgálja meg, beleértve mind a piaci termékeket, mind az akadémiai kutatásokat. A fejezetben a különböző megoldások értékelése, és a feladat specifikációban megfogalmazott követelményekkel való összehasonlítása is megtalálható.

A **3. fejezet (Optikai karakterfelismerési lehetőségek)** az Android platformon elérhető, *optikai karakterfelismerést* támogató eszközök értékelését tartalmazza. A fejezetben az általam legmegfelelőbbnek ítélt megoldás bemutatása után, annak felhasználására tér ki a fejezet, beleértve a könyvtár csatolását a szoftverhez, valamint a csatlakoztatás során fellépő szoftveres problémák megoldását.

A **4. fejezet (Szövegfordító szolgáltatások)** az elérhető *számítógépes szövegfordítási* szolgáltatásokat, és ezek összehasonlítását tartalmazza a feladat elvárásainak fényében. Ezután az általam választott szolgáltatás részletes bemutatása következik. A fejezet tárgyalja a szolgáltatáshoz való csatlakozás, és az ezzel összefüggésben lévő autentikációs és többszázalú időzíti problémák megoldását. A fejezet végén a fordítási eredmények gyorsítótárázását megvalósító megoldás bemutatása található.

Az **5. fejezet (Képfeldolgozás)** tárgyalja az alkalmazás kamerakezeléssel és kamerakép feldolgozással kapcsolatos részleteit. A fejezet részletesen bemutatja a kamerakép beolvasását, valamint továbbítását az *optikai karakterfelismerő* komponens felé. Itt kerül ismertetésre az *OCR* komponens felhasználása, időzíti és konkurenciakezelési problémáinak megoldása. Továbbá a karakterfelismerés kimenetének feldolgozása, és az alkalmazás adatmodelljének bemutatása is a fejezetben található. A fejezet a karakterfelismerést végző könyvtár kiegészítésével zárul.

A **6. fejezet (Kiterjesztett valóság)** az előzőekben bemutatott komponensek keretét,

a *kiterjesztett valóságért* felelős komponenst mutatja be. Itt kerül bemutatásra az alkalmazás felülete, nézeteinek struktúrája és felhasználói eseményei. A fejezet tárgyalja a megjelenítéssel kapcsolatos információk detektálásának megoldásait, valamint a információk megjelenítésének, illetve a *kiterjesztett valóság* hatás elérésének módját.

A **7. fejezet (Tesztelés)** mutatja be az alkalmazás manuális tesztelését támogató alkalmazásváltozatok elkészítését és felhasználását.

A **8. fejezet (Konklúzió)** tartalmazza az elkészült megoldás értékelését, az elért eredmények valamint továbbfejlesztési lehetőségek bemutatását.

A dolgozathoz csatolt **Függelékben** található az elkészült szoftvermegoldás telepítési útmutatója, felhasználói kézikönyve és az alkalmazástesztek eredményei.

1. fejezet

Specifikáció

A **Diplomatervezési feladat** c. feladatkiírás fogalmazza meg az alkalmazással és a jelen dokumentummal kapcsolatos követelményeket. Habár a leírás jól definiálja a feladat során felhasználni kívánt technikákat, valamint számos funkcionalitással kapcsolatos követelményt előír, a konkrét megvalósítás részleteit mégis magam választhattam meg. Az alkalmazás tervezése és fejlesztése előtt, ezeket a követelményeket megfelelően kell értelmezni és pontosítani, egy alkalmazás specifikáció formájában, hogy a további munkám során a követelmények és elvárások megvalósíthatóak legyenek.

A feladat értelmezése és pontosítása

A feladatkiírás előírja, hogy mely technikákat kell felhasználni az alkalmazás készítésekor. Ezek a területek a *számítógépes szövegfordítás*, az *optikai karakterfelismerés* és a *kiterjesztett valóság*, melyek a **Bevezetőben** már részletesen be lettek mutatva. A feladat ezen területek megismerése, valamint az *optikai karakterfelismeréssel* és *számítógépes szövegfordítással* foglalkozó meglévő megoldások értékelése, és a legmegfelelőbb kiválasztása. Ezen túl, feladatom még a kiválasztott megoldások csatlakoztatása a feladat implementációjához, valamint a folyamatba való integrálása.

Az *optikai karakterfelismerésért* felelős komponens feladata, hogy álló képről képes legyen magyar vagy angol nyelvű, számokat és egyéb írás jeleket nem tartalmazó szavakat felismerni. A szavak tartalmán túl a komponensnek a szavak pozícióját, sorrendjét, betűtípus jellemzőit (félkövér, dőlt stb.) és betűméretét is meg kell tudnia határozni. A komponensnek képesnek kell lennie aszinkron (a megjelenítéstől függetlenül futó) működésre, valamint képes kell legyen a találatok megfelelő szűrésére, a minél pontosabb eredmény érdekében.

A szövegfordításért felelős komponensnek képesnek kell lennie angol vagy magyar nyelvű szavakat az ellenkező nyelvre fordítania, szavanként. Továbbá a komponens feladata, hogy a fordítás eredményét megfelelően gyorsítótárazza, és a megjelenítésért felelős komponens részére bocsájtja. Hasonlóan az karakterfelismerést végző komponenshez, ennek a komponensnek is képesnek kell lennie az aszinkron működésre. Viszont nem követelmény a komponenssel szemben az internetkapcsolat nélküli (*offline*) működés.

A feladat központi eleme a *kiterjesztett valóság* megjelenítésért felelős komponens tervezése és implementálása, melyet teljesen magam készítetek el. Bár a feladat ezt nem írja elő, de én mégis követelményként specifikálom, hogy az alkalmazás a megjelenítés során, úgy jelenítse meg a lefordított szavakat, hogy azok a lehető legjobban eltakarják és utánozzák az eredeti szó vizuális jellemzőit, így elérve a lehető legjobb *kiterjesztett valóság* hatást. Ehhez a következő követelményeket kell a komponensnek kielégítenie:

1. A folyamatos kamerakép megjelenítése a felhasználói felületen.
2. A talált szavak adatainak elkérése a karakterfelismerést végző komponenstől.
3. Csatlakozás a fordítókomponenshez és a lefordított szavak elérése.
4. A szavak háttérszínének detektálása és megjelenítése.
5. A szavak betűszínének detektálása és megjelenítése.
6. A szavak egyéb vizuális adatainak (pozíció, betűméret, betűtípus jellemzők) megjelenítése.
7. A felhasználótól érkező események kezelése.

A feladat előírja a régebbi Android verziók támogatását is. Ennek érdekében az alkalmazást úgy kell elkészíteni, hogy támogassa az Android 2.2 verzióval vagy újabb rendszerrel rendelkező telefonokat és táblagépeket, beleértve a pár hónapja megjelent, legújabb 5.0-ás verziót is. Ez természetesen nem az ettől újabb verziókban megjelent funkciók elhagyását, hanem azok visszafelé történő támogatását jelenti.

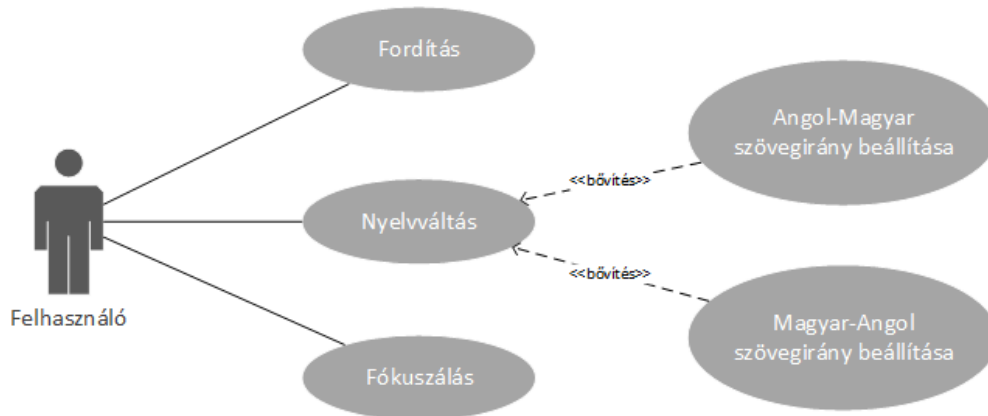
A feladat kitér az alkalmazás tesztelésére és dokumentálására is. Jelen dolgozat szolgál az alkalmazás fejlesztői dokumentációjaként, valamint a forráskód és az abban található, megfelelően formázott (*javadoc*) fejlesztői megjegyzések is csatolva vannak a dolgozat mellékleteként. Az alkalmazás felhasználói kézikönyve, illetve telepítési útmutatója a dolgozat függelékében található.

Az alkalmazást számos manuális teszttel, különböző, de direkt a megoldás erősségeire és gyengeségeire kiélezett bemenetekkel teszteltem. A tesztek eredményei a dolgozat függelékében találhatóak.

Felhasználói elvárások

A alkalmazás célja, hogy a már tárgyalt technológiákat a lehető legjobban kihasználva nyújtson, a mindennapi szövegfordító megoldásoktól eltérő, *kiterjesztett valóság* alapú szolgáltatást. Cél a technológiákban rejlő lehetőségek kiaknázása és felhasználóbarát módon történő biztosítása a felhasználók felé.

Az alkalmazásnak az elindítás után szinte azonnal (a különböző modulok betöltődése után) használhatónak kell lennie és minden beállítás opcionálisan később is elvégezhető kell legyen. Az alkalmazásnak, a lehető legjobb hatás érdekében, a felhasználók által megszokott



1.1. ábra. Az alkalmazás használati eset diagramja.

módon kell támogatnia a különböző interakciókat. A felhasználói használati eseteket a 1.1 ábra mutatja be.

Az alkalmazás fő használati esete a *Fordítás*. Az alkalmazás elindítása után ez az eset automatikusan fut, és a felhasználónak a telefont úgy kell mozgatnia, hogy a kamera képén a lefordítani kívánt szöveg jól láthatóan megjelenjen, ezután a fordítás automatikusan megtörténik és a felületen megjelennek a lefordított szavak.

A *Nyelvváltás* használati eset során a felhasználónak lehetősége van nyelvet változtatni, melynek segítségével a fordítás forrás és cél nyelvét tudja beállítani. Ezt a használati esetet bővíti ki az *Angol-Magyar szövegrány beállítása*, illetve a *Magyar-Angol szövegrány beállítása* használati esetek, melyek a megadott szövegrányt állítják be. Ezek az esetek a felhasználói felületen elhelyezett navigációs elemekkel indíthatók el.

A *Fókuszálás* használati eset során a felhasználó elindíthatja a kamera automatikus fókuszálását. Ezt a felhasználó a megjelenített kamerakép megérintésével tudja elindítani.

2. fejezet

Irodalomkutatás

Az alkalmazás tervezése és fejlesztése előtt kutatást végeztem, hogy megvizsgáljam a feladatkiíráshoz hasonló, már meglévő megoldásokat. Mind kész kereskedelmi termékeket, szolgáltatásokat, mind akadémiai kutatásokat is megnéztem, hogy ezek alapján értékelhessem, mennyire újszerű és hasznos lehet az általam elkészítendő megoldás. Valamint azért, hogy feltárjam mik azok a hiányosságok, melyek a meglévő megoldásokban nem támogatottak, viszont úgy vélem, lenne rájuk felhasználói igény. Csak olyan megoldásokat vizsgáltam, melyen támogatják a kép alapú fordítást, tehát a bemenetet valamilyen *optikai karakterfelismerési* módszerrel veszik, hiszen enélkül egy alapvetően más megközelítésű alkalmazásról lenne szó. A meglévő megoldások értékelésénél olyan szempontokat vettem figyelembe, mint hogy mennyire használja ki a *kiterjesztett valóság* technikát a lefordított szövegek megjelenítésekor, támogatja-e az *offline* fordítást, valamint az angol és a magyar nyelveket, illetve hogy milyen okostelefon és táblagép platformra érhető el.

2.1. Kereskedelmi megoldások

2.1.1. Waygo

A *Waygo* [72] egy *kiterjesztett valóság* alapú megoldás, mely kifejezetten éttermi menük fordítását támogatja. Az alkalmazás az *optikai karakterfelismerést* felhasználva elemzi a képeket, és lefordítja a képen látható szöveget. Az alkalmazás csak kínai, japan és koreai szövegek fordítását támogatja angol nyelvre, és kifejezetten ételek neveinek fordítására van kifejlesztve. Az alkalmazás támogat *offline* fordítást is, tehát nem szükséges internetkapcsolat a fordításhoz. Bár egy régebbi verzióban az alkalmazás a lefordított szövegek megjelenítését csak az eredeti szöveg megjelenítése alatt támogatta, most már elérhető egy olyan mód is, mely az eredeti szöveg felé rajzolja a lefordított szöveget (eltakarva azt), úgy hogy az eredeti szöveg színét és hátterét használja. Az alkalmazás elérhető mind Android, mind iOS operációs rendszerű okostelefonokon és táblagépeken.

2.1.2. CamDictionary

A *CamDictionary* [54] alkalmazás is az *optikai karakterfelismerést* használja fel a kamera-képen látható szövegek felismerésére. Számos nyelvet támogat, köztük az angol és a magyar



2.1. ábra. Az Waygo alkalmazás többféle kiterjesztett valóság alapú megjelenítési módot is támogat. [10]

nyelveket is. Rendelkezik szövegkiejtési, felolvasási funkciókkal is, de ezek sajnos a magyar nyelv esetén nem érhetőek el. Az alkalmazás a felismert szöveget képes *offline* is lefordítani (ehhez a Collins és Oxford szótárakat használja). Viszont szemben a feladatkiírásban leírt alkalmazással nem képes a lefordított szöveget az eredeti szöveg környezetében megjeleníteni, habár a képernyő alján megjeleníti a lefordított szöveget. Az alkalmazás támogatja a Android és az iOS okostelefon és táblagép operációs rendszereket.



2.2. ábra. Az CamDictionary alkalmazás használat közben. [4]

2.1.3. Babelshot

A *Babelshot* [3] alkalmazás is egy olyan szövegfordító alkalmazás, amely a telefon kamerájának képét használja fel a fordítás forrásául. Viszont az előző megoldásoktól és a feladatkiírásban szereplő alkalmazástól eltérően ez a megoldás nem a folyamatos kameraképet használja fel, hanem a felhasználónak kell lefotóznia egy képet, majd azon kijelölni azt a régiót, ahol a szöveg van. Ezután az alkalmazás a lefordított szöveget és az eredeti szöveget egy új képernyőn jeleníti meg, függetlenül a kameraképtől. Habár ez a megoldás is valami-

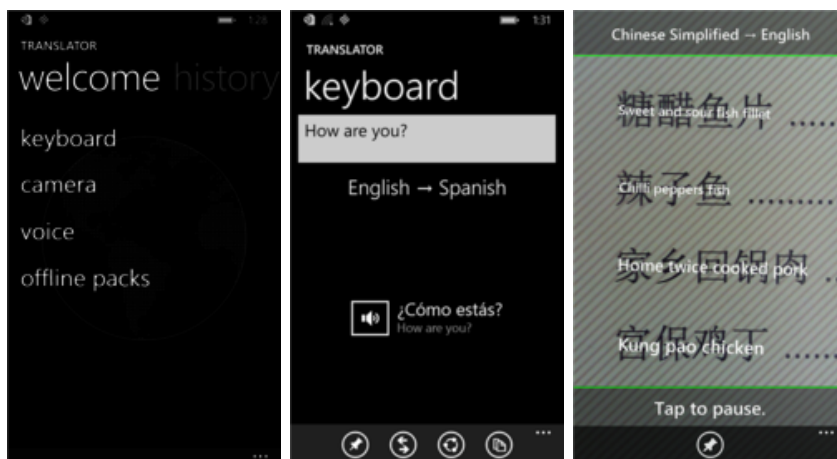
lyen *optikai karakterfelismerés*en alapul, a kép megjelenítéséhez nem használ *kiterjesztett valóság* technikákat. Az alkalmazás támogatja a magyar és angol nyelveket, valamint *offline* fordításra is képes. A *Babelshot* csak iOS-t futtató eszközökre érhető el.



2.3. ábra. A Babelshot alkalmazás felülete. [3]

2.1.4. Bing Translator

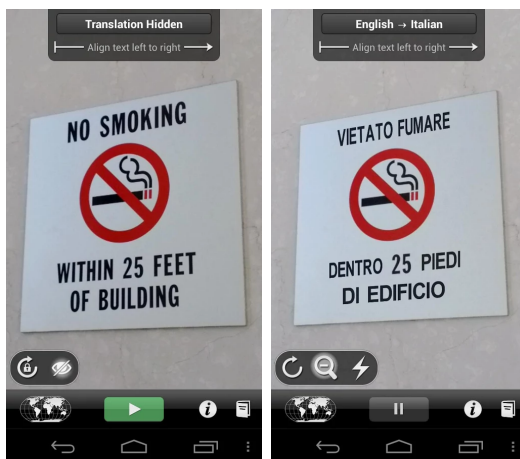
A Microsoft *Bing Translator* [57] alkalmazása, mely csak a Windows Phone operációs rendszerű telefonokra érhető el, is rendelkezik kép alapú fordító funkcióval. Az alkalmazás támogatja a hagyományos és a hangalapú fordítást is, és képes detektálni és fordítani az angol és a magyar nyelveket is. Habár az alkalmazás használ *kiterjesztett valóság* hatású megjelenítést a lefordított szöveg kijelzésére, az eredeti szöveget elhalványítja, és a lefordított szöveget ebbe a halványított mezőbe írja (mindig ugyanazzal a formázással), de mégsem éri el teljesen azt a hatást, hogy a lefordított szöveg a forrás szöveg jellemzőivel jelenjen meg, mint ami a specifikációban leírt feladat egyik követelménye. Ezen felül az alkalmazás támogatja az internetkapcsolat nélküli fordítást is.



2.4. ábra. A Bing Translator alkalmazás főbb képernyői. [57]

2.1.5. Word Lens

A *Word Lens* [11] az az alkalmazás, mely leginkább hasonlít a specifikációban leírt alkalmazásra. A telefon folyamatos képét használja a karakterolvasás bemenetül, hasonlóan a többi tárgyalt megoldás többségéhez, viszont a lefordított szavak az eredeti szavak helyén, az eredeti szó színével, betűtípusával, valamint pocizójával, torzításával/elforgatásával jeleníti meg, így nagyon jól érzékelteti azt a hatást, amit a neve is mutat, egy lencse, melyen keresztül más nyelven látod a szavakat. Az alkalmazás elérhető Android és iOS operációs rendszerű eszközökre, valamint számos nyelvet támogat, de sajnos a magyar nincs ezek között. Ezen felül *offline* működésre is képes ez a megoldás.



2.5. ábra. Az *World Lens* alkalmazás használat közben [11]

2.2. Akadémiai kutatások

Bár az említett kutatási területeken egyenként igen aktív munka folyik, az ezeket kombináló akadémiai kutatások száma csekély. Mindazonáltal, elérhetőek publikációk, melyek pont a feladatkiírásban felvázolt alkalmazás problémaköreivel foglalkoznak.

A [40] publikáció egy 2001-es kutatás eredményét foglalja össze, mely jóval megelőző korát, vetíti előre egy kézi eszköz használatát, mely a kameraképen szöveget és jeleket detektál, és azt idegennyelvre fordítja, majd megjeleníti. A kutatás során fejlesztett prototípus kínai és japán szövegek fordítását támogatja angol nyelvre (bár megjegyzi, hogy bármilyen kereskedelmi fordítómegoldással kombinálható a prototípus). A megoldás egy kamerával felszerelt PDA eszközön fut, mely Windows Pocket 3.0 operációs rendszert támogat. Ez a megoldás főként abban különbözik az előzőekben bemutatottaktól, hogy az *optikai karakterfelismerési* algoritmus ténylegesen nem az eszközön fut, hanem egy háttér infrastruktúra elemen. Az eszköz a kamerakép készítésében és a lefordított szöveg megjelenítésében vesz részt. A kameraképet elküldi a *szervernek*, mely elvégzi a karakterfelismerést és a szavankénti fordítást, majd visszaküldi a kézi eszköznek. A megjelenítésnél ebben az esetben csak a szöveg pozíciója (és persze a tartalma) játszik szerepet, a színek és egyéb vizuális jellemzők nem kerülnek felhasználásra a megjelenítési folyamat során.

A [49] publikáció már egy sokkal frissebb kutatás eredménye, mely meglehetősen ha-

sonlít a specifikációban megfogalmazott alkalmazásra. A megoldás egy Maemo operációs rendszerű okostelefonra lett kifejlesztve, és ebben a kutatásban már a telefonon magán történik az *optikai karakterfelismerés*. A megoldásban az okostelefon a kamera képét dolgozza fel, majd a fordításhoz egy webes fordító szolgáltatást (Google Translate [42]) használ. A fordítás és megjelenítés nem minden látható szóra működik, hanem a felhasználónak kell kijelölni a képernyőn egy pontot és az az alatti szöveget fordítja és jeleníti meg a rendszer. Ez a megoldás már felhasználja az eredeti szöveg vizuális jellemzőit (háttér és szövegszín), valamint a karakterfelismerést egy szótár alapú megoldással szűri, így javítva a felismerés pontosságát.

2.3. Értékelés

Mint ahogy az a fejezetben bemutatásra került, mind kereskedelmi, mind akadémiai közegekben vannak kutatások és fejlesztések a feladat által érintett témákban. Mindazonáltal, olyan megoldás, mely egészében kielégítené a feladatban leírt követelményeket és funkcionalitást, nincs. Főként a magyar nyelv, mint támogatott fordítási nyelv hiánya az, ami a leginkább szembetűnő hiányosság, valamint a *kiterjesztett valóság* támogatásában is meglehetősen eltérnek az elérhető megoldások. Ezért úgy gondolom, hogy a feladatkiírásban leírt alkalmazásnak van létjogosultsága és mind kereskedelmi, mind kutatási irányban tud számottevően hozzáadni a most elérhető megoldásokhoz.

3. fejezet

Optikai karakterfelismerési lehetőségek

Az alkalmazás működését, valamint az adatfeldolgozás irányát tekintve kitüntetett helyzetben van az *optikai karakterfelismerést* végző komponens. Mind a fordítás alapjául szolgáló szövegnek, mind pedig a megjelenítésnél használt szöveg *meta-információknak* (elhelyezkedés, szín, méret stb.) ez a komponens biztosítja a forrását. Ebből kifolyólag fontos, hogy ez a komponens a lehető legtöbb információt gyűjtse, gyorsan és megbízhatóan működjön.

Az *optikai karakterfelismerés* egy hosszú múltra visszatekintő kutatási terület, számos összetett matematikai, képfelismerési és képfeldolgozási megoldáson alapszik és ma is aktív kutatások folynak ezen a területen [60]. Nyilvánvaló, hogy egy teljes értékű *optikai karakterfelismerést* végző komponens elkészítése túlmutat egy diplomamunka keretein, és több éves kutatói és fejlesztési munkát igényel. Ezért munkám során nem egy saját *OCR* megoldás készítését tűztem ki célul, hanem egy, már meglevő megoldásra építve, azt megfelelően felhasználva kívántam elkészíteni a feladatban specifikált alkalmazást. Így munkám kezdetekor kutatást végeztem az Android platformon elérhető *OCR* lehetőségekről, hogy kiválasztassam a nekem legmegfelelőbb megoldást.

3.1. Elérhető megoldások

Az Android platformon elérhető *optikai karakterfelismerést* támogató megoldások alapvetően kétféle típusba sorolhatóak. Az első ilyen típus a helyileg történő (*offline*) felismerés. Ekkor a karakterek felismerését végző algoritmusok a telefon processzorán futnak, és a felismeréshez semmilyen külső hálózati kapcsolat nem szükséges. Több ilyen megoldás is elérhető, léteznek mind ingyenes (*Vuforia* [66], *Tesseract* [47]) mind kereskedelmi eszközök (*ABBY Mobil OCR* [12], *Aspire* [37]). A másik típus pedig a távoli felismerés (*online*), amikor valamilyen távoli karakterfelismerést végző szolgáltatáshoz csatlakozik a telefon, és a karakterek felismerését végző algoritmusok nem a telefon processzorán futnak, ilyen például az *Online OCR API* [69]. Bár az első esetben a telefon hardver erőforrásai jócskán le vannak terhelve, mégis a képek feltöltésével és kommunikációval járó időtöbblettel szemben az első csoport sokkal kedvezőbb. Emellett az helyi felismerésnek nincs hálózati forgalom-

ból adódó költsége sem, ezért döntöttem úgy, hogy a helyi karakterfelismerési megoldások egyikét használom fel munkám során.

3.1.1. Vuforia

A karakterfelismeréssel foglalkozó fejlesztői lehetőségek közül elsőként a Qualcomm által fejlesztett, *Vuforia SDK*-t vizsgáltam meg. Ez a könyvtár alapvetően egy *kiterjesztett valóság* alapú szoftverek fejlesztését támogató eszköz. A szövegfelismerés mellett számos olyan funkcionalitással rendelkezik, például a *jelölő* (*marker*) felismerés, illetve a *célpontok* (*targets*) követése, melyekkel különböző *markerekre* lehet virtuálisan 3D-s objektumokat megjeleníteni. Bár ez a könyvtár sok *kiterjesztett valóságot* használó szoftver nélkülözhetetlen kelléke, sajnos a szövegfelismerésen túli többlet funkcionalitását ebben az alkalmazásban nem tudom kihasználni. A könyvtár szövegfelismerő modulja, a szövegfelismerésre mint kiegészítő funkcióra fókuszál, mely a szövegeket mint *markereket* kezelni, ezért ez nem annyira fejlett és sokrétű, mint a többi elérhető megoldás, ahol a fő cél a karakterfelismerés volt. A működés elve, hogy egy szavakból álló lista elemeit tudja felismerni, mely listát természetesen ki lehet egészíteni. Sajnos csak angol nyelvű szavakat és az angol ábécé karaktereit képes felismerni, más nyelvekkel való kiegészítése pl. magyar vagy számok felismerése sem támogatott. A *Java* nyelvű *API* mellett *C++* nyelvű *API* is létezik, valamint *Unity Engine* és *Vuforia Web Services* támogatás is elérhető, Androidon 2.2-es verzió felett használható. Bár a könyvtár nem nyílt forráskódú, használata mind oktatási mind üzleti célokra ingyenes.

3.1.2. Tesseract

A *Tesseract* egy hosszú múltra visszatekintő, kizárólag *OCR* funkcionalitással foglalkozó könyvtár. Fejlesztését 1985-ben kezdte a Hewlett Packard, majd 2005-ben vált nyílt forráskódú szoftverré. 2006 óta a Google szponzorálja a könyvtár fejlesztését. Bár kezdetben *C*-ben íródott a könyvtár, később *C++*-t használtak a fejlesztéshez, majd a teljes forráskód frissítve lett, így egy *C++* fordítóval fordítható az egész könyvtár. A könyvtár a *Leptonica* képtranzformációs és képfeldolgozást támogató könyvtárat is felhasználja.

A *Tesseract* egy mesterséges intelligencia alapú tanulási folyamatot használ, így a meglévő támogatott nyelvek mellett akár új nyelvekre, írásrendszerekre is megtanítható a könyvtár. Természetesen alapvetően is számos nyelv támogatott, köztük az angol és magyar nyelvek is, melyekre nekem szükségem van a fejlesztés során. Ellenben a többi elérhető szövegfelismerési megoldással, a *Tesseract* számos latintól eltérő ábécéjű (pl. orosz, koreai) nyelvet is támogat, valamint a jobbról balra szövegírányt használó nyelveket is, mint az arab és a héber is képes a könyvtár felismerni. A könyvtár a szavak tartalma mellett a szavak elhelyezkedését is felismeri, valamint a többi megoldástól eltérően képes a talpas, talpatlan betűtípusokat is megkülönböztetni, illetve a fix szélességű (*monospaced*) és kiskapitális (*small caps*) szövegeket is támogatja.

A *Tesseract* egyik legnagyobb hátránya, hogy nem elég alaposan dokumentált, illetve a fejlesztő részéről számos olyan többletmunkát (pl. kamerakezelés, képkonverzió) igényel,

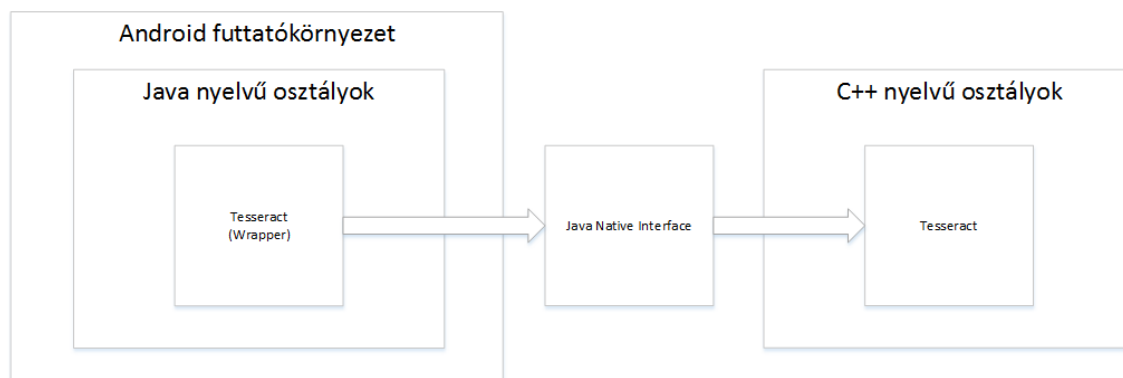
melyek pl. a *Vuforia SDK*-ban alapvetően támogatottak.

Bár számos egyedi funkcióval rendelkezik a *Tesseract*, mégis a fő ok, ami miatt ennek az *OCR* könyvtárnak a használata mellett döntöttem a nyílt forráskódja volt. A könyvtár forráskódjából számos funkció és algoritmus megérhető, valamint jól megismerhető és kihasználható a karakterolvasási folyamat. Emellett a könyvtárat bárki lefordíthatja, így igény esetén (melyre a fejlesztés során volt is példa) akár módosítható maga a könyvtár is.

Az itt bemutatott két ingyenes könyvtáron kívül természetesen léteznek még helyi karakter felismerést végző kereskedelmi megoldások mint például az *ABBY Mobil OCR* vagy az *Asprise OCR* könyvtár (amely bár Linuxra elérhető, az Androidos felhasználást nem támogatott), viszont mivel nem szerettem volna munkám során díjköteles megoldásokat használni, és az előző két ingyenes megoldást is kielégítőnek találtam, ezeket a kutatás közben nem próbáltam ki.

3.2. A Tesseract használata

Mivel a *Tesseract* könyvtár *C++*-ban íródott, így Androidon a könyvtár használata nem egy triviális feladat. Bár az *Android SDK* csak *Java* nyelven történő fejlesztést támogat, az *Android NDK* fejlesztőeszköz lehetővé teszi, hogy natív *C++* kódrészek meghívhatóak legyenek az *SDK*-val használt *Java* nyelvű osztályokból, a *Java Native Interface* [63] segítségével. Ahhoz, hogy a *C++ API Javából* jól használható legyen, a megfelelő függvényeknek a *Java* oldali és a *C++* oldali függvényeit is implementálni kell. Ez azt jelenti, hogy a *Java* oldali kódban definiálni kell üres függvényeket, melynek a megfelelői *C++*-ban implementálva vannak, és meghívják a megfelelő *C++*-os *Tesseract* függvényeket. Ez gyakorlatilag a burkoló (*wrapper*) mintát követi. Szerencsére ezt nem kell minden függvényre implementálni, mert már létezik erre megfelelő nyílt forráskódú és szabadon felhasználható könyvtár *tess-two* néven [67]. Ez a könyvtár tartalmazza magát a *Tesseract* könyvtárat, a könyvtár által felhasznált képtranzformációs könyvtárat, a *Leptonica*-t, valamint a *Java* oldali interface-t, mint Android projektet. Munkám során ezt a könyvtárat használtam fel, valamint egyes helyeken kiegészítettem a könyvtár már meglévő működését.



3.1. ábra. A *Tesseract* könyvtár elérése Android környezetben.

3.2.1. A könyvtár csatolása az Android projekthez

Az alkalmazás fejlesztése során az elmúlt évben debütált *Android Studio* fejlesztőeszközt használtam. Szemben az eddig elterjedt *Eclipse IDE*-vel, mely *Apache Ant* fordítórendszert használt, ez az eszköz a *Gradle*-t [48] használja a fordításhoz, mely egy sokkal fejlettebb és több funkcióval rendelkező rendszer. Viszont a *Tesseract* Android csatoló könyvtára még *Eclipse*-el készül, így azt megfelelően kellett konfigurálni, hogy a *Gradle* fordítóval is használható legyen.

Elsőnek egy modult hoztam létre a projektben, melybe a könyvtár forrásait és erőforrásait másoltam. Hogy ez a modul is fordítható legyen, a fordítási beállításokat tartalmazó *build.gradle*-t módosítottam [39] az alábbi módon.

```
apply plugin: 'com.android.library'

buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:0.12.2'
    }
}

android {
    compileSdkVersion 19
    buildToolsVersion "19.1.0"

    defaultConfig {
        minSdkVersion 8
        targetSdkVersion 19
    }

    sourceSets.main {
        manifest.srcFile 'AndroidManifest.xml'
        java.srcDirs = ['src']
        resources.srcDirs = ['src']
        res.srcDirs = ['res']
        jniLibs.srcDirs = ['libs']
    }
}
```

Ezután ezt a modult csatoltam az alkalmazás modulhoz. A modul fordítása két lépésben történik. Egyrészt az *C++* nyelvű implementációkat kell lefordítani, majd az Android könyvtárat, melynek a beállításait ez a fejezet tárgyalta.

3.2.2. A könyvtár lefordítása

A *C++* nyelvű implementációk, a *Tesseract* és *Leptonica* valamint a *wrapper* osztályok *C++* oldala, a projekt *jni* könyvtárában találhatóak. Mivel ezek *natív* implementációk, így az *Android SDK* fordítójától külön, az *NDK* fordítójával kell fordítani. Ellenben az *SDK* fordítójával, ami *Dalvik* (vagy újabb Android verziók esetén (4.4+) *ART*) kompatibilis *Java*

bytecode-ot fordít, az *NDK* fordítója *natív* kódra fordít, így a könyvtárakat le kell fordítani az Android által támogatott minden processzor architektúrára (*ARM*, *x86*, *MIPS*).

Ahhoz hogy a könyvtárat le tudjuk fordítani, hozzá kell adnunk a következő elérési utakat az *ANDROID_HOME* és *PATH* környezeti változóinkhoz. Ez ahhoz kell, hogy a különböző *SDK* és *NDK* binárisokat elérjük a fordítás során. A fordítás során *Ubuntu Linux BASH* parancssort használtam [67] az alábbi módon.

```
ANDROID_HOME=<SDK_HOME>;  
PATH=$PATH:<SDK_HOME>/platform-tools;  
PATH=$PATH:<SDK_HOME>/tools;  
PATH=$PATH:<NDK_HOME>;
```

Ezután pedig a következő parancsokkal lehet a könyvtárat lefordítani, a megfelelő modulban kell az *ndk-build* parancsot futtatni.

```
cd <TESSERACT_HOME>;  
cd tess-two;  
ndk-build;
```

3.2.3. Nyelvi fájlok kezelése

A *Tesseract*, hasonlóan több mesterséges intelligenciát felhasználó megoldáshoz, a tanulás során felépített információkat *tudás-fájlokban* tárolja. Mivel minden nyelv külön tanulási procedúrát igényel, így minden nyelvhez külön *tudás-fájl* tartozik. A könyvtár felhasználóinak szerencsére nem kell a könyvtárat mindig megtanítani egy-egy nyelvre, ezek a *tudás-fájlok* a könyvtár honlapjáról letölthetők, és számos nyelvet tartalmaznak, köztük a magyar és angol nyelvet is, melyre nekem az alkalmazás fejlesztése során szükségem van.

A *Tesseract* inicializálásakor mindig be kell tölteni a választott nyelv *tudás-fájlját*. Ekkor a nyelvi fájloknak a telefon tárhelyén (belső vagy külső memóriakártya) kell lennie. Viszont az alkalmazás telepítésekor a nyelvi fájlokat csak az *assets* könyvtárban lehet a telepítőbe csomagolni, ahonnan viszont nem tudja a *Tesseract* betölteni.

Ennek a problémának a megoldására készítettem egy betöltő modult, mely az adott nyelv első használatakor beolvassa a hozzá tartozó nyelvi fájlt az *assets* könyvtárból, és a telefon tárhelyére írja. A nyelvi fájlok betöltéséért és a *Tesseract* inicializálásért a *LanguageLoader* osztály felel. Mivel a betöltés és a telepítés két elkülöníthető funkció, ezért abban az esetben, ha a megfelelő fájlok nincsenek telepítve, a betöltő osztály a *Setup* osztály segítségével létrehozza a megfelelő könyvtárstruktúrát, illetve a nyelvi fájlokat beolvassa az *assets* könyvtárból, és a megfelelő helyre másolja.

```
TessBaseAPI baseApi = new TessBaseAPI();  
  
if (Setup.isInstalled(lang))  
{  
    baseApi.init(Setup.TESSERACT_PATH, lang);  
}
```

```

else
{
    Setup.createDirectories();
    Setup.copyLanguageData(context, lang);
    baseApi.init(Setup.TESSERACT_PATH, lang);
}

```

A *assets* könyvtárból történő olvasás, valamint a tárhelyre írás az alábbi módon történik.

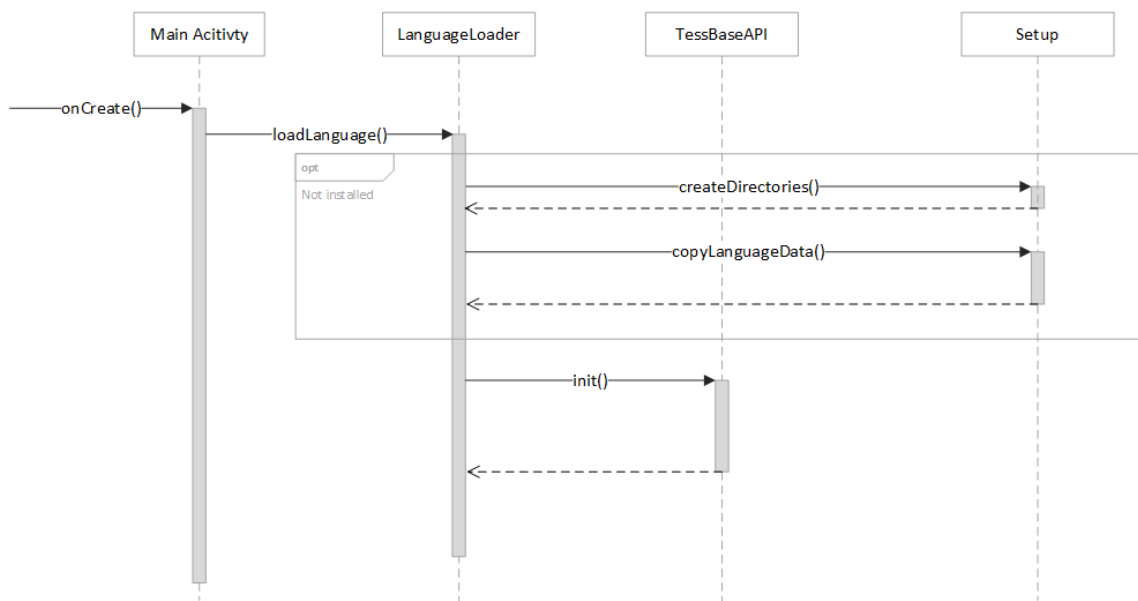
```

AssetManager assetManager = context.getAssets();
InputStream in = assetManager.open("tessdata/" + lang + ".traineddata");

File file = new File(FILE_PATH + lang + ".traineddata");
file.createNewFile();
FileOutputStream out = new FileOutputStream(file);

byte[] buf = new byte[1024];
int len;
while ((len = in.read(buf)) > 0)
{
    out.write(buf, 0, len);
}
in.close();
out.close();

```

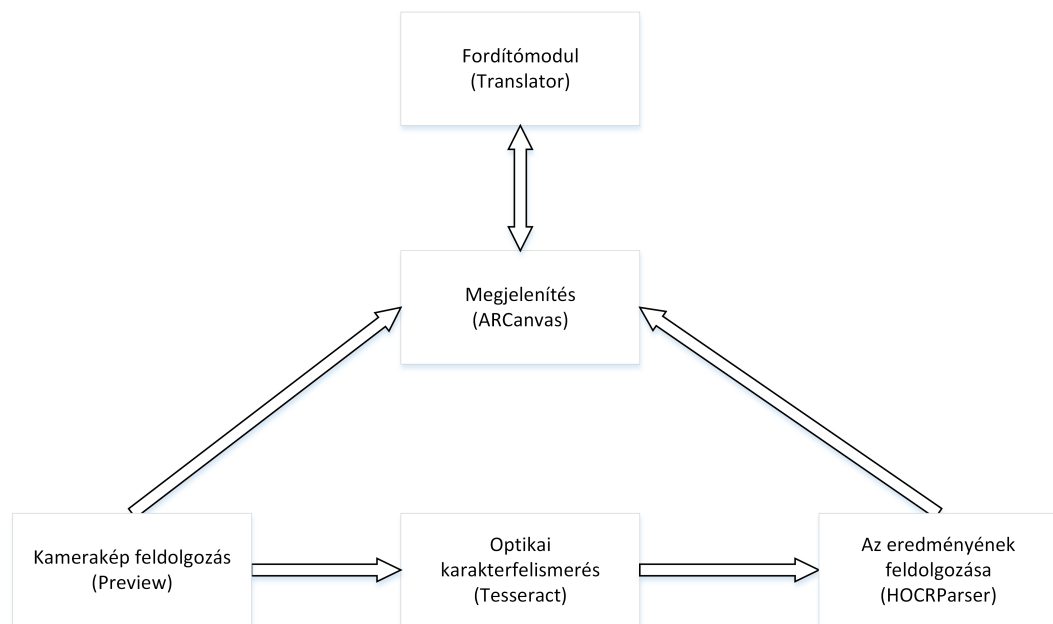


3.2. ábra. A nyelvi fájlok betöltésének és a Tesseract inicializálásának szekvencia diagramja

4. fejezet

Szövegfordító szolgáltatások

Mivel az alkalmazás alapfunkcionalitásához szorosan hozzátartozik a szövegfordítás, a fordításért felelős komponens minősége jelentősen befolyásolja az alkalmazás felhasználhatóságát. Mivel a számítógépes szövegfordítás még a karakterfelismerésnél is régebb múltra tekint vissza [61], ennek a komponensnek az elkészítésénél is valamilyen meglévő szövegfordítást nyújtó megoldásra igyekeztem támaszkodni. A fejlesztés kezdetekor több szövegfordító megoldást is megvizsgáltam, és úgy döntöttem, hogy a szövegfordító modult úgy készítem el, hogy esetlegesen lecserélhető legyen később, tehát cél volt, hogy a lehető legkevesebb függőség legyen a feldolgozó szál többi része, és a szövegfordító modul között.



4.1. ábra. Az alkalmazás komponensei és a köztük lévő kapcsolat.

Mivel az alkalmazás felhasználási lehetőségeit nagyban növelheti, ha nincs szükség a szövegfordításhoz hálózati kapcsolatra, így elsőként egy helyi (*offline*) szövegfordítási megoldással akartam ezt a funkcionalitást megvalósítani. Sajnos nem találtam olyan *offline* szövegfordító megoldást, mely amellettt hogy ingyenes lenne, a magyar nyelvet is támogatná. Egy komplett szótár (kulcs-érték párok) beillesztése az alkalmazásba pedig amellettt,

hogy hatalmas tárhelyet igényelne, szerzői-jogi problémákat is felvetne, így az *offline* szövegfordítást elvettem.

Az *offline* fordítás mellett a másik megoldás természetesen az *online* fordítás. Az *online* szövegfordítás egyik nagy előnye, hogy a mobil kliensen minimális számítási kapacitást és tárhelyet igényel, hátránya viszont, hogy internetkapcsolatra van szükség a használatához. A mobil kliens a fordítandó szöveget vagy szövegrészt elküldve egy távoli szolgáltatást hív meg, majd a szolgáltatás lefordítja a szöveget és válaszul visszaadja azt. Számos ilyen szolgáltatás található az interneten, melyek ingyenes szövegfordítási funkcionalitással rendelkeznek. Ahhoz, hogy a legmegfelelőbb szolgáltatást kiválaszthassam, több ilyen megoldást is megvizsgáltam.

4.1. Webszolgáltatások

4.1.1. Google Translate API

Mivel Android fejlesztő és felhasználóként nap mint nap találkozok Google szoftverekkel, így kézenfekvő megoldásnak tűnt, a *Google Translate* [46] szolgáltatás *API*-ját (Application Programming Interface) megvizsgálnom. Bár ez a szolgáltatás évekig ingyenes volt egy bizonyos szó/hónap korlátig, sajnos a munkám kezdetekor már megszűnt ez a lehetőség, és már csak havidíj ellenében érhető el a szolgáltatás. Mivel nem szerettem volna havidíjas/-díjkötelezett szolgáltatásokra támaszkodni munkám során, így ezt a lehetőséget elvettem. Mivel a szolgáltatás webes felhasználói változatában támogatott a magyar és angol nyelv, valószínűleg az *API* is támogatja ezeket a nyelveket mint fordítási forrás és mint fordítási cél nyelv (az *API* támogatott nyelveinek lekérdezéséhez is *API Kulcs*, így előfizetés szükséges).

4.1.2. Gengo API

Hasonlóan az előző szolgáltatáshoz, a *Gengo API* [41] is nyújt szövegfordítási funkciókat, viszont ez a megoldás is díjköteles (a lefordított szavak mennyisége után kell fizetni). Habár cél nyelvként támogatott a magyar, a fordítás forrásául szolgáló nyelvek között a magyar nincs feltüntetve mint támogatott nyelv. A havidíj és a magyar nyelv hiányában ez a megoldás sem felelt meg az igényeimnek.

4.1.3. WebserviceX - Translation Engine

A *WebserviceX - Translation Engine* [74] nevű megoldása, egy ingyenes szövegfordító web-szolgáltatás. Sajnos ez a megoldás nem támogatja se forrás se cél nyelvként a magyart, így ezt a megoldást sem tudtam felhasználni.

4.1.4. Microsoft Translator

Végül a Microsoft szövegfordító megoldását a *Microsoft Translatort* [58] vizsgáltam meg. Ez az *API* havi 2 millió karakterig ingyenes, valamint mind forrás, mind cél nyelvként támogatja a magyart és az angolt. Továbbá a szolgáltatás több formában is elérhető (*AJAX*,

Név	Leírás
<i>client_id</i>	Az alkalmazás regisztrációjakor kapott azonosító.
<i>client_secret</i>	Az alkalmazás regisztrációjakor kapott kód.
<i>scope</i>	A szolgáltatás azonosítója, melyhez a hozzáférést kérjük, jelen esetben <i>http://api.microsofttranslator.com</i> .
<i>grant_type</i>	Engedély típus, jelen esetben értéke <i>client_credentials</i> .

4.1. táblázat. A hozzáférési token igénylésének paraméterei.

Név	Leírás
<i>access_token</i>	A hozzáférési <i>token</i> , melyet később fel kell használni minden kérés során.
<i>token_type</i>	A <i>token</i> típusazonosítója, jelent esetben értéke <i>SWT</i> (Simple Web Token)
<i>expires_in</i>	A hátralévő másodpercek száma, amíg a token használható. Maximális értéke 10 perc, ennyi idő után mindenképpen meg kell újítani a token.
<i>scope</i>	Megegyezik a kéréskor küldött <i>scope</i> értékkel.

4.2. táblázat. A hozzáférési token igénylés válaszána paraméterei.

HTTP, *SOAP*). Mivel a specifikáció minden követelményének megfelel, és (az általam használt keretek között) ingyenes, így ezt a megoldást választottam.

4.2. Autentikáció

A *Microsoft Translator API* használatához elsőként a *Microsoft Azure Marketplacen* [59] kellett regisztrálni az alkalmazást, így szerevez *clientId* azonosítót és *clientSecret* kulcsot, melyek a használat során azonosítják az alkalmazást. Ezután az autentikációs logika implementációjára volt szükség. Ahhoz, hogy egy kliens kéréseket küldhessen a fordítószolgáltatás felé, elengedhetetlen, hogy rendelkezzen egy aktív hozzáférési *token*nel (*access token*). Ezt a *token*t az alkalmazás indításakor mindig el kell kérni egy webes szolgáltatástól, és szükség esetén meg kell újítani. A *token* igényléséhez szükséges a *clientId* és a *clientSecret*, melyek az alkalmazást azonosítják. A *token* igénylés egy *HTTP Post* [53] kérés formájában zajlik, a megfelelő címre egy kérést kell küldeni a 4.1 táblázat által leírt paraméterekkel. A kérésre kapott válasz tartalmát a 4.2 táblázat írja le.

A kérés paraméterei mint *HTTP Post* [53] kulcs-érték párok a kérés törzsében kerülnek elküldésre. Habár Androidon lehetőség van a hálózati kommunikációt a *java.net* csomag béli *HttpURLConnection* [28] osztály segítségével is megoldani, munkám során én az egyszerűbben használható, *org.apache.http* csomagban található *DefaultHttpClient*-et [26] használok. A kérés összeállítása és elküldése az alábbi módon történik.

```
HttpClient httpClient = new DefaultHttpClient();
HttpPost httppost = new HttpPost(URL);
...
List<BasicNameValuePair> nameValuePairs = new ArrayList<BasicNameValuePair>();
nameValuePairs.add(new BasicNameValuePair(clientIdString, clientIdContent));
```

```

nameValuePairs.add(new BasicNameValuePair(clientSecretString,
    clientSecretContent));
nameValuePairs.add(new BasicNameValuePair(scopeString, scopeContent));
nameValuePairs.add(new BasicNameValuePair(gratTypeString, gratTypeContent));

httpPost.setEntity(new UrlEncodedFormEntity(nameValuePairs));

HttpResponse response = httpClient.execute(httpPost);

```

A válaszban a különböző adattagok *JSON* [52] formátumban vannak tárolva. Ahhoz, hogy a választ könnyebb legyen feldolgozni, a *Google GSON* [44] könyvtárat használtam, mely képes *Java* osztályok és *JSON* formátumú entitások két irányú konverziójára. Ehhez a *GSON* a *Java Reflection API*-t [65] használja, mely segítségével a könyvtár képes kiolvasni a különböző osztály tagváltozókat, és azokat értékkel feltölteni (még akkor is a tagváltozó *private* láthatóságú).

Elsőnek elkészítettem az *AccessToken*-t reprezentáció *Java* osztályt, majd a különböző tagváltozóit (mivel azok nem egyeznek meg a válaszban használt nevekkkel) elláttam a könyvtár által biztosított *annotációkkal* [62] az alábbi módon.

```

public class AccessToken
{
    @SerializedName("access_token")
    private String accessToken;
    ...
}

```

Majd a könyvtár segítségével a *JSON* választ feldolgoztam.

```

Gson gson = new Gson();
AccessToken accessToken = gson.fromJson(token, AccessToken.class);

```

A kérés paramétereinek értékei konstansként vannak elhelyezve az alkalmazásban, a válasz elemei közül, pedig csak az *access_token* és az *expires_in* mezők azok, melyeket később is fel kell használni a futás során. Azért, hogy ezek a későbbiekben is elérhetőek legyenek, a *SharedPreferences*-be mentem. A *SharedPreferences* [31] egy kulcs-érték párokat tároló adatbázis, mely minden alkalmazáshoz külön, csak az alkalmazás által hozzáférhető területen tárolja az elmentett adatkárokat. Az alkalmazás minden pontjáról elérhető (ahol rendelkezésre áll az alkalmazás *kontextus*), írható és olvasható. Mivel az *access_token* és az *expires_in* szöveges, illetve szám formátumú, így könnyedén ki lehet menteni, valamint beolvasni a *SharedPreferences*-ből. Az *access_token* és az *expires_in* mentése, valamint kiolvasása az alábbi módon történik.

```

SharedPreferences preferences = PreferenceManager.
    getDefaultSharedPreferences(MainActivity.this);
Editor edit = preferences.edit();

```

Név	Leírás
<i>text</i>	A fordítás forrásául szolgáló szöveg, a forrás nyelven.
<i>from</i>	Forrás nyelv
<i>to</i>	Cél nyelv

4.3. táblázat. A fordításhoz használt kérés paraméterei.

```
edit.putString("accessTokenData", accessToken.getAccessToken());
edit.putString("accessTokenExpiresIn", accessToken.getExpiresIn());
edit.apply();
```

```
SharedPreferences preferences = PreferenceManager.
    getDefaultSharedPreferences(context);
String accessToken = preferences.getString("accessTokenData", "");
String expiresIn = preferences.getString("accessTokenExpiresIn", "");
```

4.2.1. Token frissítés

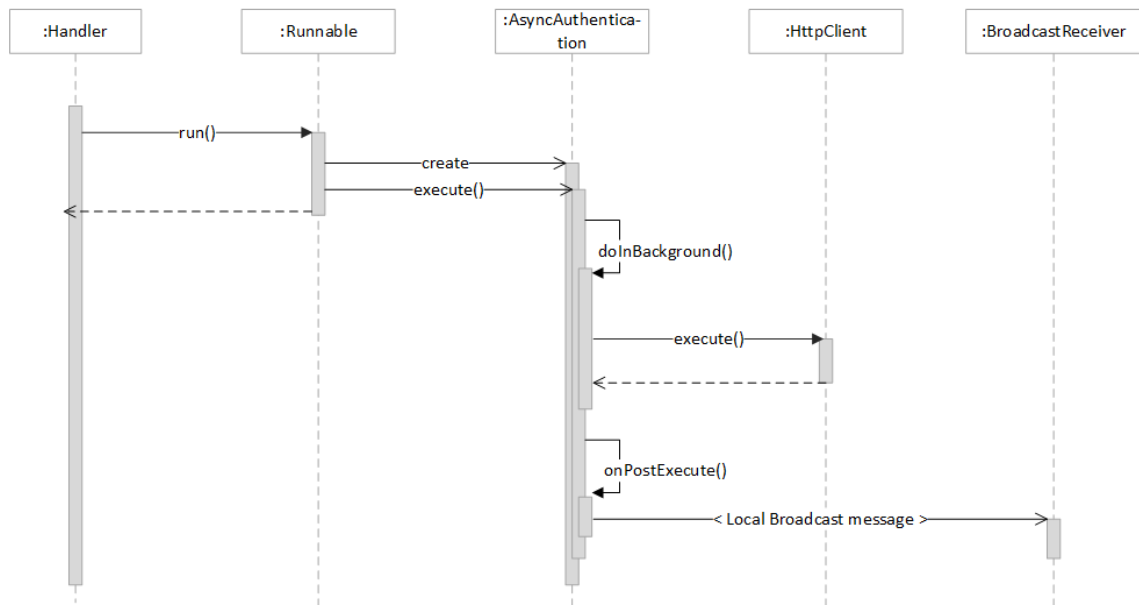
A hozzáférési *token*, egy adott időszakban érvényes, mely a kiállítás idejétől kezdődik, és az *expires_in* által jelzett ideig érvényes. A *token* csak ebben az időintervallumban használható azonosításra, így fontos, hogy mindig érvényes *token* álljon rendelkezésre a fordítás során a megfelelő működéshez. Az alkalmazás indítása után egy *token* kérés történik, mely válasza csak egy adott ideig érvényes, ezért az érvényességi időtől függően újra és újra indítani kell egy *token* kérést, és a válasz alapján frissíteni az elmentett *token* információkat. Androidon több módszer is van az ütemezett feladatokra, én a *Handler* osztályt [27] használtam erre a célra. Azért, hogy biztosan ne járjon le a *token* a kommunikáció során, a lejáratási idő előtt egy perccel, vagy ha attól kisebb a lejáratási idő, azonnal küldök egy új *token* kérés.

```
Handler handler = new Handler();
handler.postDelayed(new Runnable()
{
    @Override
    public void run()
    {
        //Start token update
        ...
    }
}, updateTokenTime);
```

4.3. Csatlakozás

Miután rendelkezésre áll a hozzáférési *token*, a *Microsoft Translator* szolgáltatás felé küldhetők a fordítási kérések. A fordítási kérést egy *HTTP Get* kérés formájában a szolgáltatás címére kell küldeni. A kérés *Get*-paramétereit írja le a 4.3 táblázat.

A kérés során a hozzáférési *token*t a kérés fejlécében, a *"Bearer "+access_token* formá-



4.2. ábra. Az hozzáférési token megújításának szekvencia diagramja.

ban kell küldeni. A *HTTP Get* kérés küldése, paraméterezése, valamint a fejléc kitöltése az alábbi módon történik.

```

HttpClient httpClient = new DefaultHttpClient();
HttpGet httpGet = new HttpGet();
...
String urlEncodedTextToTranslate = URLEncoder.encode(textToTranslate, "utf-8");
String getUrl = URL + "?" + textString + "=" + urlEncodedTextToTranslate + "&"
    + fromString + "=" + fromContent + "&" + toString + "=" + toContent;

httpGet.setURI(new URI(getUrl));
httpGet.addHeader(headerAuthorizationString, headerAuthorizationPrefix
    + accessToken);
HttpResponse response = httpClient.execute(httpGet);
  
```

A válasz sikeres fordítás esetén a lefordított szöveg, sikertelen válasz esetén *XML* formátumú hibaüzenet.

4.3.1. Aszinkron kommunikáció

Habár az *Android SDK* mind *HTTP Get*, mind *Post* kérésekre nyújt (több) megoldást, ezek minden esetben, a hívó szálon történő időigényes feladatok. Ezért ezen kéréseket nem lehet az alkalmazás főszálán (*UI Thread*) végezni, mert az alkalmazás egészét (beleértve a felhasználói felületet) blokkolná. Ezért az *Android SDK* által nyújtott többszálú fejlesztést segítő megoldást használtam, az *Async Task*-ot [19].

Az *AsyncTask* megoldás során, egy beépített, *generikus* osztályból kell származtatni saját osztályunkat. Ezen osztály nagy előnye, hogy előre definiált függvényeket nyújt a programozónak, melyek mindig a megfelelő szálon hívódnak meg. A kérés végrehajtása előtt hívódik meg az *onPreExecute(...)*, utána pedig az *onPostExecute(...)*, melyek mind-

két eseten az alkalmazás fő szálán futnak (viszont szinkronizálva vannak a mellékszálal), a *doInBackground(...)* pedig egy mellékszálán hívódik meg. Ez azért nagyon hasznos, mivel a hálózati kommunikáció során, bár a kommunikáció külön szálon kell fusson, a visszajelzéseknek mégis ezzel szinkronban kell zajlania. A visszajelzésekért felelős *onPre-* és *onPostExecute(...)* függvények azért a fő szálán futnak, mert a visszajelzések tipikusan valamilyen felhasználói felületet érintő események, és a felhasználói felülethez csak a fő szál nyúlhat (ezért is hívják sokszor angolul *Main Thread* helyett *UI Thread*-nek).

4.3.2. Broadcast események

A hálózati kommunikáció végén a kérést küldő fő szálnak az *onPostExecute(...)* függvényben kell visszaszólni. Habár kézenfekvő lenne, hogy a vissza irányú kommunikációt úgy oldjam meg, hogy az *AsyncTask* osztály tárol egy referenciát a küldő *Android Activity*-ről melynek megfelelő függvényét hívja a válasszal paraméterezve, ez a megoldás mégsem működőképes. Sajnos ez a megoldás azért nem használható, mert a küldő *Activity* a kérés alatt megsemmisülhet, például a telefon elforgatására. Előfordulhat, hogy bár a kérés indításakor él a küldő *Activity*, a kérés megérkezésekor már vagy megsemmisült állapotban van, vagy már újra létrejött. Bármelyik eset is áll fent, a referencia amit az *AsyncTask* tárol, mindkét esetben *null*, tehát nem tudunk visszaüzeni a küldőnek. Azt szerettem volna, hogy amennyiben nem él vagy háttérben van a küldő *Activity* akkor ne kapja meg senki a választ (mivel ekkorra nincs szükség rá), ha pedig újra létrejött az *Activity*, akkor az új (a kérés szempontjából ugyanolyan állapotban lévő) *Activity* kapja meg a régi kérésének választát. Ezt a legegyszerűbben az *Android SDK* által nyújtott *LocalBroadcast* megoldással oldottam meg.

Az *Android Broadcast*-ek [20] egy esemény alapú architektúrát valósítanak meg. Az eseményeket az alkalmazásból bárholnan el lehet sütni, és az eseményre feliratkozott *BroadcastReceiver*ek pedig megkapják ezt függetlenül attól hogy ki és hol indította a kérést. Ezért minden kérésre érkezett válaszkor, az *AsyncTask onPostExecute(...)* függvényében egy olyan *Broadcast* eseményt indítok, mely tartalmazza a választ. Az *Broadcast* eseményekhez, mint minden alkalmazáskomponensek közötti híváshoz, csatolható *Intent* [29] objektum, a válasz tartalmát ebben adom át az eseményt elkapó félnek.

```
Intent intent = new Intent(TRANSLATION_RECIEVED);
intent.putExtra("TextToTranslate", textToTranslate);
intent.putExtra("TranslatedText", translatedText);
LocalBroadcastManager.getInstance(context).sendBroadcast(intent);
```

Az esemény elkapását a főképernyő életciklusával szinkronban végzem. Ha a főképernyő előtérbe kerül (*onResume()* életciklus függvény) feliratkozom az eseményre, amennyiben pedig a háttérbe kerül (*onPause()* életciklus függvény) leiratkozom az eseményről. Az esemény kezelésére a *BroadcastReceiver* osztályt használom, ahol az *Intentben* átadott választ olvasom ki.

```

@Override
public void onResume()
{
    super.onResume();
    ...
    LocalBroadcastManager.getInstance(this).registerReceiver(
        translationReceiver, new IntentFilter(TRANSLATION_RECIEVED));
}

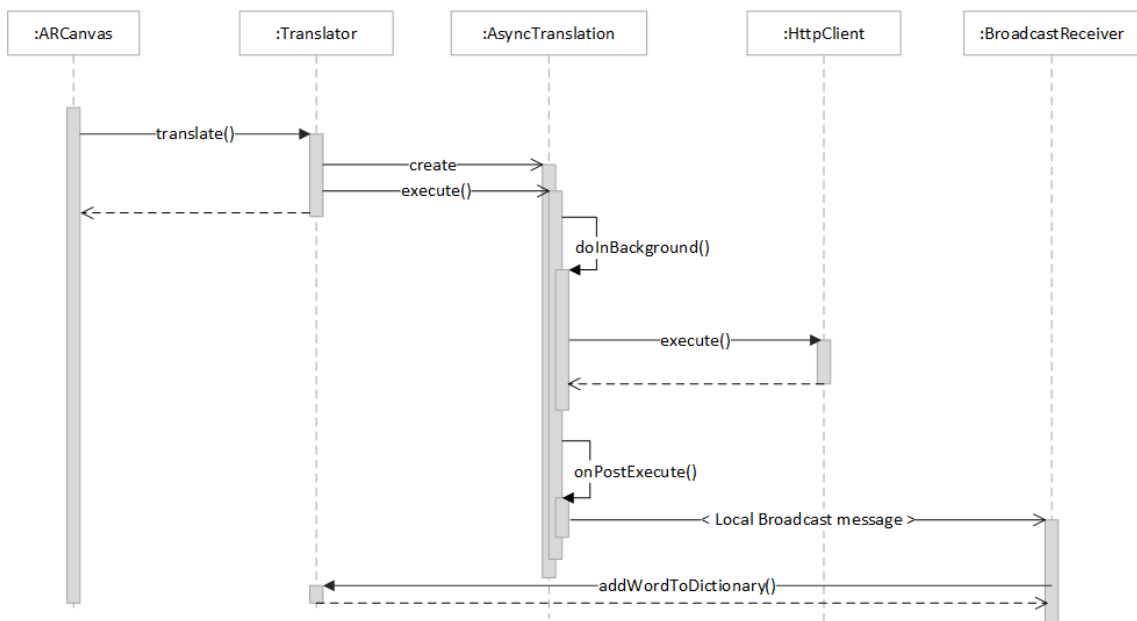
@Override
protected void onPause()
{
    LocalBroadcastManager.getInstance(this).unregisterReceiver(translationReceiver);
    super.onPause();
}

...

private BroadcastReceiver translationReceiver = new BroadcastReceiver()
{
    @Override
    public void onReceive(Context context, Intent intent)
    {
        String textToTranslate = intent.getStringExtra("TextToTranslate");
        String translatedText = intent.getStringExtra("TranslatedText");
        ...
    }
};

```

A hozzáférési *token* kérés, valamint a fordítás szekvencia diagramjai a 4.2 és a 4.3 ábrán láthatóak.



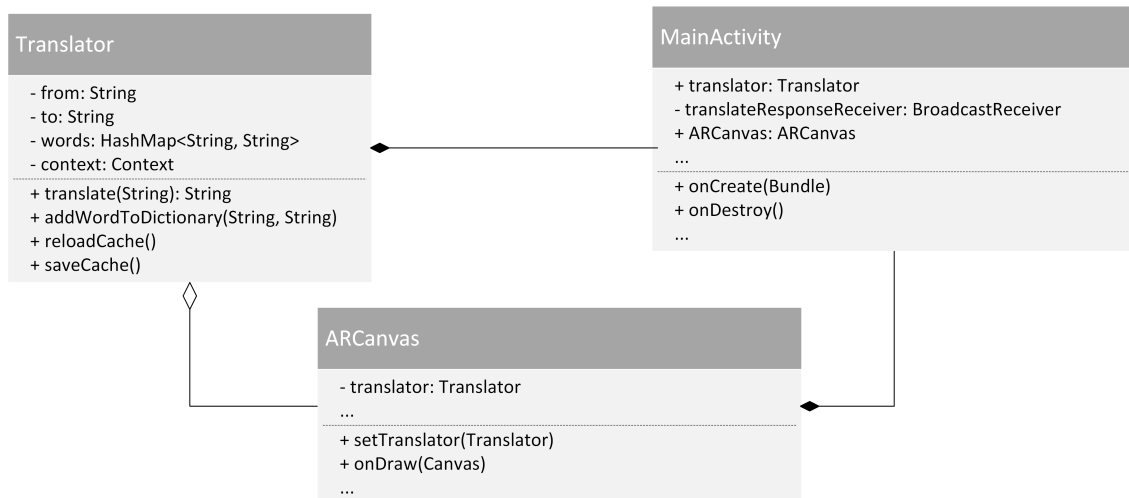
4.3. ábra. A fordítás szekvencia diagramja.

4.4. Gyorsítótárazás

Mivel a fordítás alapvető feltétele az internet kapcsolat, így a kapcsolat teljes hiányában nem képes működni ez a megoldás. Mindazonáltal előfordulhat, hogy bizonyos időre nem elérhető az internetkapcsolat, esetleg egye-egy kérésre nem érkezik válasz, és ezek az esetek kezelhetők. Az ilyen esetek kezelésére, valamint a felesleges hálózati kommunikáció csökkentése érdekében a fordításért felelős modult elláttam egy gyorsítótárazást megvalósító megoldással.

A gyorsítótárazás vagy *cachelés* jelen esetben kétfeléppen történhet, a háttértárban vagy a memóriában. A memóriában történő *cachelés* során nem kell háttértár olvasást végezni minden egyes kérésnél, viszont az alkalmazás bezárása után a memória cache tartalma törlődik. Ellenben a háttértáron történő gyorsítótárazáskor minden kérés előtt olvasni kell a tárból, ami pedig eléggé időigényes, viszont az alkalmazás bezárása után is megmarad az állapota. Mivel mindkét megoldásnak megvannak a maga előnyei és hátrányai, ezért én egy vegyes megoldást használtam. A megoldásom alapvetően egy memóriában történő *cachelés*, így a kérésekkor elég csak a memóriából olvasni, viszont az alkalmazásmemória törlése előtt (alkalmazás megsemmisülés - *onDestroy()* életciklus függvény) a memória *cache* tartalmát kiírom a háttértárra, majd az alkalmazás újbóli elindításakor (alkalmazás létrejötte - *onCreate()* életciklus függvény) visszaolvasom a gyorsítótár tartalmát a memóriába.

Az adatok háttértárra írásához a már bemutatott *SharedPreferences* használtam, annyi különbséggel, hogy a szópárok által alkotott *StringSet*-et (a párok összefűzését és szétbontását nekem kell elvégezniem) kell elmentenem és beolvasnom, minden nyelv pár esetén külön. A nyelvek jelölésére azért van szükség, mert egy szó lehet több nyelv eleme is (pl. ember, eleven, van, is stb.). A szavakat pedig azért fűzöm össze, nem pedig a kulcsok és értékek halmazát mentem külön külön (csak a *String*ek halmazát lehet menteni *SharedPreferences*-be, tömbjét, listáját stb. nem), mert így azok a szópárok, melyeknél két szóhoz ugyanaz a jelentés tartozik, elvesznének (mert a halmazban nem lehet két ugyanolyan elem).



4.4. ábra. Az fordítás és a gyorsítótárazás során használt főbb osztályok egyszerűsített statikus struktúra diagramja.

A gyorsítótárazást, valamint a fordítás kérését a *Translator* osztály foglalja magában. Mivel egyszerre egy időben csak egy fordítási irány támogatott, a *Translator* osztályból elég egy példány, mely tárolja a cél és forrás nyelvet, viszont a szavaknál már nem kell ezt külön jelölni (csak a háttértárra mentéskor). A szópárokat egy *HashMap<String, String>* objektumban tárolom, így könnyen kereshető az eredeti szó alapján a lefordított szó.

Az osztály rendelkezik egy *translate(String word)* függvénnyel. Amennyiben a szó megtalálható a *HashMap*-ben, a függvény visszatér a lefordított szóval, ha nem, elindít egy fordítási kérést.

A függvény rendelkezik egy publikus *addWordToDictionary(String from, String to)* függvénnyel, melyet a fordítást kezelő (fő) *Activity* hív, miután megkapta a fordítás választ tartalmazó *Broadcast* eseményt. Az osztály fejlesztése során feltételeztem, hogy ugyanahhoz a forrás nyelvű szóhoz a fordítószolgáltatás mindig ugyanazt a cél nyelvű szót adja, tehát alternatív fordításokat nem tárol az osztály, mivel azok nem fordulhatnak elő.

5. fejezet

Képfeldolgozás

Az előző fejezetekben bemutatott komponensek mind az információfeldolgozási folyamat egy-egy kitüntetett lépését valósítják meg. Ahogy a szövegfordítást végző modul támaszkodik a karakterfelismerést végző megoldásra, úgy a karakterfelismerésnek is elengedhetetlen, hogy rendelkezzen valamilyen adatforrással. A karakterfelismerést végző komponens egy speciális formátumú képet tud feldolgozni, és mivel a megoldás egy általános karakterfelismerést végző könyvtárra (*Tesseract*) épít, így a forrásadatok biztosítása teljes mértékben az én feladatomban volt.

Az alkalmazásspecifikációban megfogalmazott követelmények között szerepel, hogy a lefordított szöveget a kiterjesztett valóság érzetét keltve kell a képernyőn elhelyezni. Ezért, hogy a megjelenített információk a lehető legfrissebbek legyenek, a karakterfelismerés bemenetéül a kamera által látott pillanatnyi képet kell biztosítani folyamatosan. Ezért azt a megoldást, amikor egy fotót készít a felhasználó, majd csak és kizárólag azt a képet elemezzük, elvettem, és helyette a folyamatos kamerakép feldolgozása mellett döntöttem. Ennek a lényege, hogy a kamera által pillanatnyilag látott képet kirajzoljuk a telefon képernyőjére, majd ugyanezt a képet feldolgozva, rárajzoljuk az képernyőre a lefordított szavakat, helyenként elfedve az aktuális kameraképet. Így a felhasználó mindig a legfrissebb információkat látja, és csak a feldolgozási és kirajzolási procedúra késleltetését érzékeli. Mivel ez a lépéssorozat egy állandóan ismétlődő folyamat egy ciklusa, a feldolgozási idő nagy mértékben meghatározza a felhasználói élményt, így az egyes lépések időtartama kritikus tényező, melyet a tervezés során figyelembe kellett vennem.

5.1. Kamerakezelés

Mivel a feldolgozási folyamat forrása a telefon kamerájának a képe, így annak a beolvasása, illetve megjelenítése a feldolgozási folyamat során a leghamarabb történik. Bár sok telefon rendelkezik mind hátlapi, mind előlapi kamerával, a *kiterjesztett valóság* hatás elérésére nyilvánvalóan csak a hátlapi kamera használható. Bár a mai telefonok kamerái (főleg a hátlapi kamerák) már a digitális fényképezőket is túlszárnyaló felbontással rendelkeznek, a feldolgozási folyamat során mégsem használható ki ez az előny teljesen. Sajnos a folyamatos kép érdekében egy kisebb felbontású, rosszabb minőségű képet kell kérnünk a

hátlapi kamerától. A telefon képernyőfelbontása a megjelenített kép szempontjából ad alsó korlátot, mivel cél, hogy a megjelenített kép ne legyen rosszabb minőségű, mint a telefon képernyőfelbontása. Továbbá a fényképkészítés a telefonon egy hosszadalmasabb folyamat (kb. 1 másodperc), viszont jelen esetben egy folyamatos kameraképre (*Preview Image*) [23] van szükség, mely a folyamatos képolvasás és így a rövidebb mintavételezésre szánt idő miatt rosszabb minőségű. Ahogy az elkészített kép esetében, a *Preview* kép esetében is több felbontás támogatott, amely készülékenként eltérő lehet. Ahhoz, hogy a már említett felbontáskorlátot a lehető legjobban kielégítsem, a telefon által biztosított legnagyobb felbontású *Preview* képet használok fel. A telefon kamerájának elérése (a *Camera* [21] osztály reprezentálja), a támogatott *Preview* formátumok lekérdezése, és a megfelelővel történő inicializálás a következő módon történik.

```
Camera mCamera = Camera.open();
List<Size> sizesPrev = mCamera.getParameters().getSupportedPreviewSizes();

Size highest=null;
for (Size size : sizesPrev)
{
    if(highest==null)
    {
        highest=size;
    }
    else
    {
        if (highest.width<size.width && highest.height<size.height)
        {
            highest=size;
        }
    }
}

Parameters parameters = mCamera.getParameters();
parameters.setPreviewSize(highest.width, highest.height);
parameters.setFocusMode(Camera.Parameters.FOCUS_MODE_AUTO);
mCamera.setParameters(parameters);
```

Ahhoz, hogy a kapott kép a lehető legélesebb legyen, be kell állítani azt is, hogy a kamera hogyan fókuszáljon. Bár az újabb Android verziókban támogatott a folyamatos automatikus fókusz (*FOCUS_MODE_CONTINUOUS_PICTURE*), a régebbi verziókkal való kompatibilitás miatt ezt a megoldást nem használtam. Helyette a normál auto fókuszot használtam, mely során az automatikus fókuszálást a fejlesztőnek kell elindítania. Az automatikus fókuszálás indítására a legkézenfekvőbb megoldás, ha a felhasználó a képernyőre nyomva indítja el. Ez a legfelső nézeten való kattintás eseménykezelőjében, a kamerán hívott automatikus fókusz kérésével idézhető elő.

```
arCanvas.setOnClickListener(new View.OnClickListener()
{
    @Override
    public void onClick(View view)
```

```

{
    mCamera.autoFocus(new Camera.AutoFocusCallback()
    {
        @Override
        public void onAutoFocus(boolean b, Camera camera)
        {
        }
    });
}
});

```

5.1.1. A Preview megjelenítése

A hátlapi kamera képének beolvasásáért és megjelenítésért az általam készített *Preview* osztály felel. A *Preview* osztály az Android beépített *SurfaceView* [34] osztályából származik, így gyakorlatilag egy teljes értékű Android nézet. Amint a nézet létrejön, csatlakozik a telefon kamerájához és elkezd kéri tőle folyamatosan a legfrissebb képet. A megjelenített képet egy *SurfaceHolder* (felület tároló) [33] osztály tartalmazza, melyet a *SurfaceView* ősoosztály biztosít. Ahhoz, hogy mindig a megfelelő *Holder*-be kerüljön a megjelenített kép, a *Holder* által definiált *callback* eseményekre kell feliratkozni. Itt biztosított az, hogy a *Holder* létrejöttkor csatlakódjon hozzá a kamera, mint a *Preview* kép forrása, a megsemmisítésekor pedig lecsatlakódjon róla a kamera. Ha a *Holder* paraméterei megváltoznak (elfordítás, háttérbe kerülés), akkor mindig újra el kell indítani a *Preview* kép kérését a kamerától. A kép folyamatos frissítéséről, és *Holder* számára elérhetővé tételéről a már említett *Camera* osztály gondoskodik. A *Holder* és a *Camera* csatlakozása az alábbi módon történik a *Preview* osztály konstruktorában.

```

Holder mHolder = getHolder();
mHolder.setSizeFromLayout();
mHolder.addCallback(this);
mHolder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);

```

A *Holder* eseményeinek kezelését a *Preview* osztály által megvalósított *SurfaceHolder.Callback* interface függvényi biztosítják.

```

@Override
public void surfaceChanged(SurfaceHolder surfaceholder, int format, int w, int h)
{
    mCamera.startPreview();
}

@Override
public void surfaceCreated(SurfaceHolder holder)
{
    ...
    mCamera.setPreviewDisplay(holder);
    ...
}

@Override

```

```

public void surfaceDestroyed(SurfaceHolder arg0)
{
    ...
    mCamera.stopPreview();
    ...
    mCamera.release();
}

```

A *Holder* mellett a *Camera* osztály is biztosít különböző eseményeket, melyek a *Preview* kép feldolgozásakor hívódnak meg. Mivel a *Holder* és *Camera* közötti kapcsolat átlátszó (nem a fejlesztőnek kell kiolvasni a kamerából a képet, és betölteni a *Holder*-be, mert ezt az említett osztályok megoldják), így ebben az esetben nem férék közvetlenül hozzá a *Preview* képhez. Azért, hogy elérjem a *Preview* képet, a *Camera* osztály *PreviewCallback* interfacet valószínűsítem meg, majd csatolom a kamerához mint eseménykezelőt.

```

Camera.PreviewCallback previewCallback = new Camera.PreviewCallback()
{
    public void onPreviewFrame(byte[] data, Camera camera)
    {
        ...
    }
}
mCamera.setPreviewCallback(previewCallback);

```

Ebben az eseménykezelőben már hozzáférhető a *Preview* kép, mint függvényparaméter (*data* byte tömb). Ezt a képet használok fel a *Tesseract* számára nyújtott adatforrásként. Ez a byte tömb a kép adatait *YUV* képformátumban tárolja [38], mely egy multimédia környezetben elterjedt formátum, viszont a képfeldolgozó megoldások általában nincsenek felkészítve erre a formátumra. A *Tesseract* sem képes *YUV* formátumú képek kezelésére (mivel a képtranszformációs könyvtár, a *Leptonica* ezt nem támogatja), ezért át kell konvertálnom a képet egy olyan formátumba, melyet támogat a *Leptonica* könyvtár, hogy azt a képfeldolgozó komponens kezelni tudja. Jelen esetben ez a *JPEG* formátum [55]. A konverzió során, a kapott *Preview* képből egy kisebb felbontású képet készítek a folyamat. Erre azért van szükség, mert az *optikai karakterfelismerés* hossza függ a kép felbontásától is. Mivel a legnagyobb *Preview* kép, ami a telefonon elérhető, általában jóval nagyobb attól, mint ami a karakterfelismeréshez szükséges, ezért azt le kell kicsinyíteni. Ennek a mértékét tapasztalati úton határoztam meg. A kicsinyítés mértéke négyszeres mind a kép magasságát, mind a kép szélességét nézve, így a *pixelek* száma gyakorlatilag tizenhatszor kevesebb. A konverzió után a kapott *Bitmap* már egyszerűen átadható a *Tesseract*-nak.

```

Size pictureSize = camera.getParameters().getPreviewSize();

YuvImage yuvimage = new YuvImage(data, ImageFormat.NV21, pictureSize.width,
    pictureSize.height, null);
ByteArrayOutputStream baos = new ByteArrayOutputStream();
yuvimage.compressToJpeg(new Rect(0, 0, pictureSize.width, pictureSize.height),
    90, baos);

```

```
byte[] jdata = baos.toByteArray();
Bitmap bitmap = BitmapFactory.decodeByteArray(jdata, 0, jdata.length);
Bitmap resized = Bitmap.createScaledBitmap(bitmap, pictureSize.width / 4,
    pictureSize.height / 4, true);
```

5.2. Kapcsolódás a Tesseracthoz

A *Tesseract* karakterfelismerő egy példányát a már bemutatott *LanguageLoader* osztály hozza létre a megfelelő nyelvvél. Amint ez betöltődött, küldhetők feléje a kérések. A karakterfelismerés első lépésként a már átkonvertált képet kell átadni a könyvtárnak, majd meghívni a beolvasást végző függvényeket. A *Tesseract* kétféle módszert is biztosít a feldolgozásra, melyek a feldolgozás mikéntjében nem különböznek, viszont a visszaadott adatstruktúrában eltérnek.

Az első esetben a visszaadott adatmodell, a *C++*-os *Tesseract* modell osztályok egy részének *Java* oldali leképezése. Ilyenkor az adatok egy *iterátorral* érhetők el egy speciális struktúrában. Sajnos ez az adatstruktúra nem egy-egy megfeleltetése a *C++* adatstruktúrának, így nem érhető el egyszerűen minden adat (például adott szóhoz tartozó befoglaló négyzet szavanként), melyet a könyvtár kiolvas.

A másik lehetőség az eredmények visszaadására a *hOCR* szabvány [71]. A *hOCR* szabvány egy *XML* alapú, szöveges leírást definiál, mely az *optikai karakterfelismerést* végző szoftverek szabványos kimenetét adja meg. Ebben az adatstruktúrában több adatmező, általánosabb elrendezésben érhető el, mint az első esetben. Egy mintafelismerés eredményét a következő példa struktúra tartalmazza.

```
<div class='ocr_page' id='page_1' title='image " "; bbox 0 0 1920 1080; ppageno 0'>
  <div class='ocr_carea' id='block_1_1' title="bbox 0 0 1920 1080">
    <p class='ocr_par' dir='ltr' id='par_1_1' title="bbox 277 0 1920 1080">
      <span class='ocr_line' id='line_1_1' title="bbox 277 0 1920 1080;
        baseline 0.012 -509">
        <span class='ocrx_word' id='word_1_1'
          title='bbox 277 405 697 572; x_wconf 93'
          lang='eng' dir='ltr' >
            Test
          </span>
        ...
      </p>
    </div>
  </div>
```

5.2.1. A hOCR feldolgozás

Ahhoz, hogy a *hOCR* alapú a módszert használjam, egy saját értelmezőt kellett implementálnom ehhez a kimenethez, és így lehetőségem nyílt a saját adatmodell használatára is. Az értelmező *hOCRParser* egy *XML* feldolgozó, mely *DOM* alapú megközelítést követ [73]. Szemben a *SAX* módszerrel [70], mely *XML tag*-enként halad, és eseményeket definiál

bizonyos *tag*-ekre, a *DOM* módszer egy objektum modellt épít a *XML* struktúra alapján, és azt lehet bejárni a feldolgozás során. Azért a *DOM* alapú feldolgozót használtam, mert tipikusan a *hOCR* válaszok mindig valamilyen rövidebb válaszok, tehát nem kérdés, hogy elfér-e a válasz és az objektum modell a memóriában, mely ellenben a *SAX* mellett szóló egyik érv lenne. Továbbá a *DOM* módszer sokkal egyszerűbb, rugalmasabb és átláthatóbb programozói modellt követ, így ezt választottam.

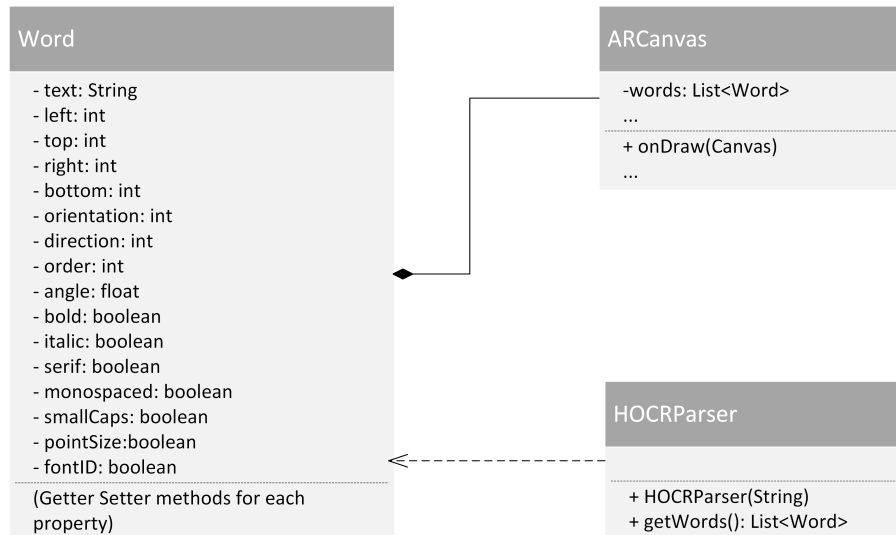
Felmerült még továbbá, a *reflekcio* alapú feldolgozó könyvtárak használata (pl. *simplexml* [68]). Ez a megoldás hasonló a *GSON*-hoz, viszont *XML* és *Java* osztályok egy-egy megfeleltetését végzi, így nem kell a fejlesztőnek a feldolgozást implementálnia. Viszont azért nem emellett a megoldás mellett döntöttem, mert a *reflekcio* egy időigényes folyamat, és nem szerettem volna a képfeldolgozási ciklust ezzel az idővel is növelni. Továbbá mivel a *hOCR* struktúra független az általam használt adatmodelltől, így a *reflekcio*s esetben, egy külön adatmodell kellett volna létrehoznom a *reflekciónál* használt osztályokkal (melyek a *hOCR* struktúrától függenek), és azt kellett volna konvertálnom a saját adatstruktúrámba.

Felmerülhet a kérdés, hogy ebben az esetben miért vettem el a *reflekcio* alapú megoldásokat, amikor a fordítószolgáltatás modulban, az autentikáció választ szintén a *reflekcio*t használó *GSON*-al dolgozom fel. Szemben a szövegfordítással, mely aszinkron módon zajlik, a karakterfelismerés a feldolgozási folyamattal szinkronban, a lánc részeként fut, tehát minden idő amit a feldolgozás eltölt késlelteti az eredményt. Továbbá a szövegfordítás során, csak az autentikációs folyamat végén kell *JSON* feldolgozást végezni, mely általában csak alkalmazásindításkor, vagy használat közben 10 percenként egyszer aszinkron módon történik meg. Az autentikációs választól eltérően, a szövegfordítás válasza, melyre folyamatosan szükség van a futás során (minden nem *cached* szó esetén), pedig szöveges formában van továbbítva, így azt nem kell feldolgozni használat előtt.

5.2.2. Adatstruktúra

A saját adatstruktúrámba az 5.1 ábrán látható. Az *ARCanvas* osztály (mely következő fejezetben kerül részletesen bemutatásra) tartalmazza az aktuálisan látható szavak listáját. A *Word* osztály egy szót reprezentál, és minden szóhoz tárolja a szót befoglaló négyzet point-jait (melyek a beolvasott kép koordináta rendszerében értendők), tárolja a szót szöveges formátumban, valamint a különböző jellemzői is, úgymint félkövér, dőlt, talpas, talpatlan, fix szélességű és kiskapitális. Tárolja továbbá az irányt, az elfordítás szögét, a sorrendet, valamint a betűtípus azonosítóját és a betűméretet is (szintén az olvasott kép koordináta rendszerében). Az *ARCanvas* mindig csak az egy ciklusban olvasott szavakat tárolja, ciklusonként törölődik a szavak listája (így mindig csak a legfrissebb információkat jelenítjük meg)

A *hOCR* feldolgozás során, minden elemen végig megy a feldolgozó, és amennyiben olyan elemet talál mely *ocrx_word* osztállyal rendelkezik, akkor a hozzá tartozó attribútumok és tartalom alapján feltölti a szó adatait. Az elem tartalmát kivéve, mely az olvasott szöveg, a *title* attribútum kerül még kiolvasásra, mely szóközökkel elválasztva tárolja a szóhoz tartozó befoglaló négyzet (*Bounding Box*) pozícióját, valamint a konfidencia szintet, mely



5.1. ábra. Az optikai karakterfelismerés feldolgozása után kapott adat modell egyszerűsített statikus struktúra diagramja.

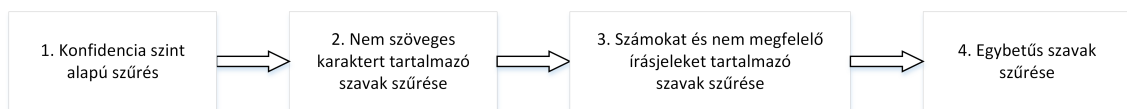
azt jelzi, hogy a talált szó milyen valószínűséggel lett megfelelően beolvasva.

A title attribútum felépítésére egy példa:

```
title='bbox 277 405 697 572; x_wconf 93'
```

5.2.3. Adatszűrés

A szavak szűrése is a feldolgozó egység feladata. Ez úgy történik, hogy minden talált szó esetében, a szó tartalma alapján egy szűrési folyamat eldönti, hogy le kell-e fordítani a szöveget. Sajnos a *Tesseract* nagyon sok használhatatlan információt is visszaad, mert a szöveges tartalmak mellett egyéb érdektelen, de a képen rajta lévő adatot is szöveggként értelmez esetenként, ami számos értelmetlen szöveget és karaktert eredményezhet, ezért a felhasználói élmény szempontjából fontos, hogy azokat az információkat, melyek valószínűleg nem tartalmaznak tényleges szöveges tartalmat, de a *Tesseract* mégis betűknek olvassa, ne jelenítse meg az alkalmazás a felhasználói felületen, és ne is próbálja lefordítani.



5.2. ábra. Az karaktfelismerés során talált adatok szűrésének folyamata.

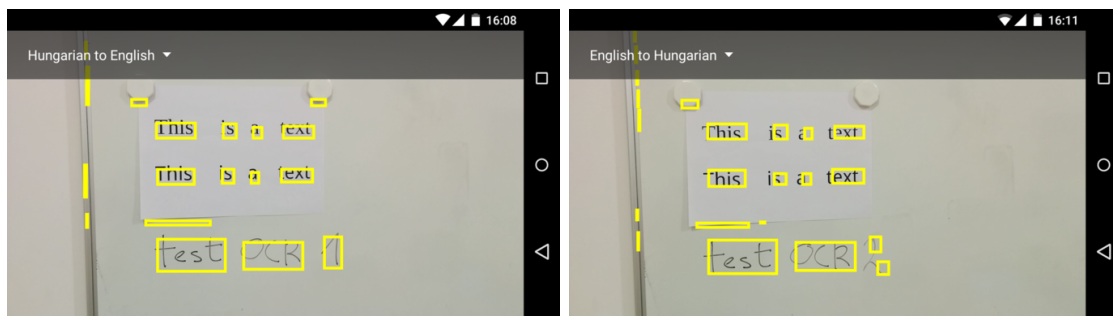
A szűrés során első lépésként (1) a konfidencia szint alapú szűrés történik. Ez az attribútum egy, a *Tesseract* által vissza adott érték, mely a tesztjeim alapján elég jól közelíti, hogy mennyire megfelelően lett kiolvasva az érték. A szűrési feltétel a 40% -os konfidencia szint,

minden olyan szót, mely ettől kisebb szinttel rendelkezik eldob a szűrő. Ez a küszöbérték egy általam választott állandó, mely a *Tesseract* használata során elég jó közelítésnek tűnt. Bár a szűrő implementációja során elsőnek ezt a szűrőt a szűrési lánc végére tettem, később előrébb mozgattam, mivel a későbbi szűrési lépések által eldobott elemek nagy részét már ez a feltétel is kiszűri, és mivel általában ez egy tágabb szűrési feltételként értelmezhető, így általánosságban kevesebb szűrési lépésre van szükség, ha ezt a lépést a lánc elejére teszem.

Ezután a nem szöveges karaktereket tartalmazó szavakat szűröm (2). A szűréshez itt egy speciális *reguláris kifejezést* használok, mellyel eldönthető, hogy egy adott szó tartalmaz-e nem latin (magyar és angol betűtől eltérő) karaktert. Amennyiben a szó mégis tartalmaz ilyen karaktert, úgy nem csak a karaktert, de magát a szót is kiszűröm.

```
Pattern p = Pattern.compile("[^\\u00C0-\\u1FFF\\u2C00-\\uD7FF\\w]");
boolean hasSpecialChar = p.matcher(text).find();
```

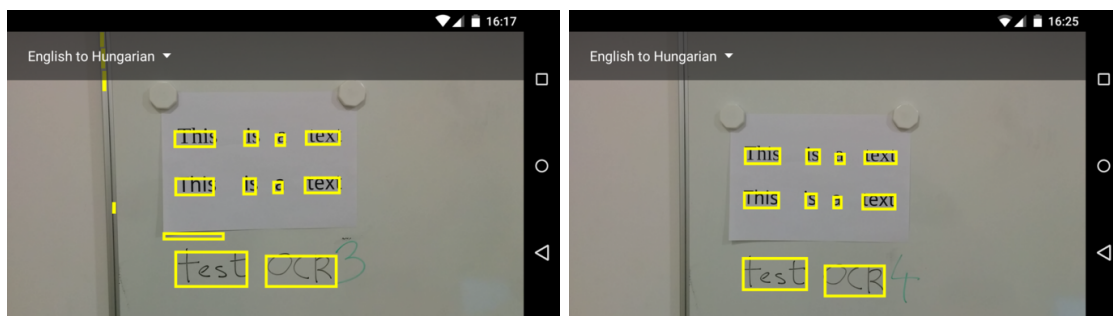
A következő szűrési lépésben (3) kiszűröm az olyan szavakat melyek számokat, illetve mondatvégi (kérdőjel, pont, felkiáltójel) és összekötő írásjeleket (vessző, kötőjel) is tartalmaznak. A legutolsó szűrési lépésben (4) pedig az egybetűs szavakat szűröm. Erre azért van szükség, mert a különböző nem szöveges alakzatokat sokszor egy-egy betűnek érzékeli a karakterfelismerő (pl. a lap sarkra egy nagy L betű stb.), viszont mind a magyar, mind az angol nyelvben az értelmes egybetűs szavak száma csekély. Ezért minden olyan egybetűs szót, mely nem a kis és nagy A, nagy I (angolul én), vagy kis/nagy S (s magyar összekötő) is kiszűrök a válaszból. A különböző szűrési lépéseket az 5.3 és a 5.4 ábra demonstrálja.



5.3. ábra. Az optikai karakterfelismerés eredménye szűrés nélkül (bal), valamint az első szűrési lépés alkalmazásával (jobb).

5.2.4. Aszinkron karakterfelismerés

A fentebb bemutatott karakterfelismerési folyamat egy hosszadalmas műveletsor (akár több másodperc nagyságrendű is lehet, a *Preview Image* felbontásától és tartalmától függően), valamint ez a folyamat állandóan ismétlődik, tehát az alkalmazás használata során az idő nagy részében ez a folyamat fut. Ez a folyamat nem futhat az alkalmazás fő szálán (*UIThread*), mivel akkor a felhasználói felületet hosszabb időre is blokkolná. Viszont nem futhat teljesen függetlenül a fő száltól sem, mivel az egész megjelenítési folyamat függ attól,



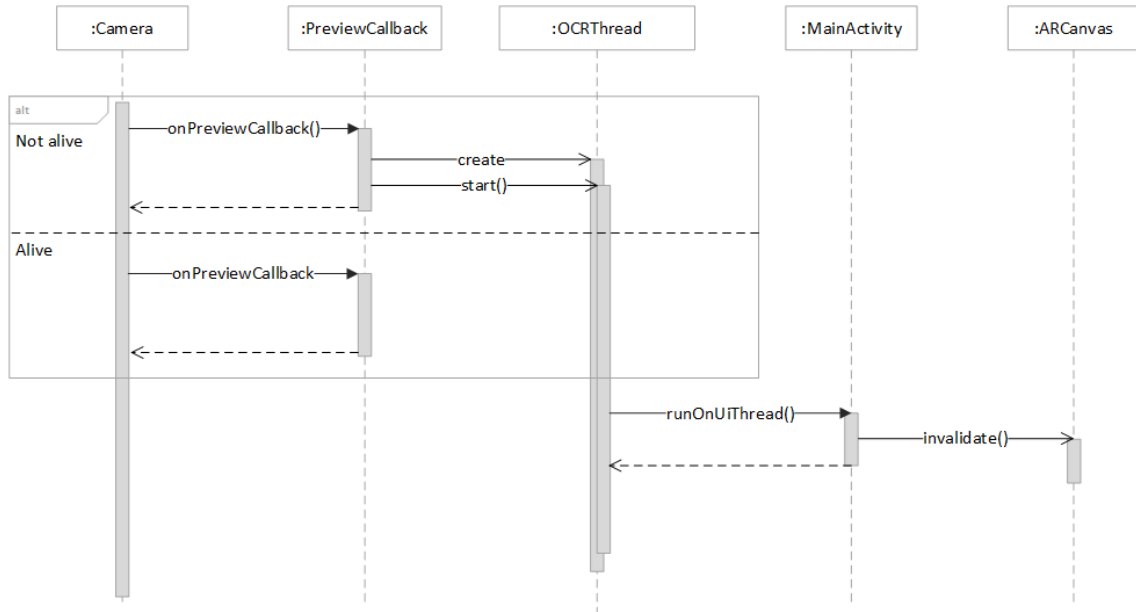
5.4. ábra. Az optikai karakterfelismerés eredménye az első kettő szűrési lépéssel (bal), valamint a teljes szűrési láncsal (jobb).

hogy mikor érkezik meg feldolgozás eredménye. Valamint nem lehet egymás mellett több szálát indítani, mivel a *Tesseract* egyszerre több szálból való hívása nem támogatott. A probléma kiküszöbölésére a következő módszert használtam.

A karakterolvasási folyamat során, a kamerától érkező *Preview Image callback* eseménykezelőben egy új mellékszálát indítok. Ebben a mellékszálban található a már bemutatott konverzió, karakterolvasás, *hOCR* kimenet feldolgozás, szűrés, valamint végül a talált információk átadása az *ARCanvas*-nek. Ezt a mellékszálát egy Java *Thread*-el [64] valósítottam meg, melyről folyamatosan tárolok egy referenciát, a *Preview* osztályban. Amikor az első kép megérkezik a kamerától elindítom ezt a szálát a megfelelő bemenettel, majd eltárolom a referenciáját, és elindítom a feldolgozást. Mivel a mellékszálban történő feldolgozás alatt új kép is érkezik a kamerától, ezért az újabb képek érkezésekor megvizsgálom, hogy létezik, illetve él-e (a *Thread* osztály *isAlive()* függvénye) az a szál amire a referencia mutat, és ha igen, akkor a meglévő kamara képet eldobom, tehát egy futó karakterolvasási folyamat során keletkező többlet képek elvesznek (a felületen természetesen megjelenik, csak feldolgozásra nem kerül). Abban az esetben, ha a feltétel nem igaz, akkor újra elindítom a szálát a kapott kameraképpel.

Ezen módszer egyik problémája, hogy a karakterolvasási folyamat végén az *ARCanvas* osztályt értesíteni kell arról, hogy változott a megjeleníteni kívánt tartalom, és rajzolja újra magát annak megfelelően. Viszont az *ARCanvas* újrarajzolását, mint minden Android nézet esetén, csak a fő szál kezdeményezheti. Ennek a megoldására egy referenciát tárolok a főképernyőről, melynek van egy speciális *runOnUiThread(...)* metódusa, mely lehetővé teszi, hogy egy mellékszálról kódot futtassunk a főszálban. Ennek segítségével a folyamat végén értesítem az *ARCanvas*-t az újrarajzolásról.

```
arCanvas.words = words;
activity.runOnUiThread(new Runnable()
{
    @Override
    public void run()
    {
        arCanvas.invalidate();
    }
});
```



5.5. ábra. Az aszinkron optikai karakterfelismerési folyamat egyszerűsített szekvencia diagramja.

5.3. A Tesseract könyvtár kiegészítése

A *hOCR* szabvány nem definiál olyan kiegészítő információkat, mint például szó sorrend vagy betűtípus jellemzők (félkövér, talpas stb.). Mivel én ezt a válasz formátumot használok fel, szerettem volna ilyen információkkal kiegészíteni a *hOCR* kimenetet, hogy a *Tesseract* által egyébként beolvasott, de a *JNI* kérés során vissza nem adott információkat is el tudjam érni. Ehhez változásokat kellett eszközölnöm a *Tesseractban*, melyet tekintve, hogy a könyvtár nyílt forráskódú és magam fordítottam le, könnyedén megtehettem.

A *JNI* csatoló felületet nem kellett módosítanom, mivel a *Java* és *C++* közötti kapcsolatot megvalósító függvényeken nem szerettem volna változtatni, csak a *hOCR* struktúrába szerettem volna több információt beszúrni. Ahhoz, hogy ezt megtehessem, a *Tesseract C++* nyelvű forrásában (*jni* könyvtár), az *src/api/baseapi.cpp*-ben található *GetHOCText()* függvényt kellett módosítanom. A függvény a *Tesseract C++ API* hívásait felhasználva lefuttatja a képfelismerést, majd végig iterál az eredményen, és ez alapján felépíti a *hOCR* kimenetet, melyet visszaad a függvény végén. Elsőnek az orientációval kapcsolatos információkat olvastam ki az iterátor aktuális eleméből, és adtam hozzá a *hOCR* kimenetet reprezentáló *C++* karakterlánchoz.

```

tesseract::Orientation orientation;
tesseract::WritingDirection direction;
tesseract::TextlineOrder order;
float deskew_angle;
char buffer[200];

res_it->Orientation(&orientation, &direction, &order, &deskew_angle);

sprintf(buffer, " orientation='%d' direction='%d' order='%d' angle='%.4f'",
        orientation, direction, order, deskew_angle);

```

```
hocr_str+=buffer;
```

Majd a betűtípussal kapcsolatos információkat, félkövér, dőlt, aláhúzott, fix szélességű, talpas, kiskapitális valamint betűméret és betűtípus azonosító attribútumokat olvastam ki, és adtam hozzá a *hOCR* válaszhoz.

```
bool bold, italic, underlined, monospace, serif, smallcaps;
int pointsize, font_id;
(void) res_it->WordFontAttributes(&bold, &italic, &underlined,
                                  &monospace, &serif, &smallcaps,
                                  &pointsize, &font_id);

if (bold)
{
    hocr_str += " bold='true' ";
}
else
{
    hocr_str += " bold='false' ";
}

...

char fontBuffer[200];
sprintf(fontBuffer, " pointsize='%d' fontid='%d' ", pointsize, font_id);
hocr_str+=fontBuffer;
```

A többletinformációkat a *ocrx_word* osztályú elemek attribútumaiként vettem fel, és ezután a szövegben található félkövér szöveget jelző * tagek* hozzáadását távolítottam el. A könyvtárat ezután a 3.2.2 fejezetben leírt módon újra kellett fordítani, és a könyvtár ezután megfelelt az igényeimnek. Habár ezzel a módosítással a *hOCR* kimenet már nem felel meg a szabványnak (mivel az általam hozzáadott módosítások nem szabványosak), de mivel máshol nem tervezem ezt a módosított könyvtárat felhasználni, ezért ez nem jelent problémát, viszont a többletinformációk miatt ez a kompromisszum számomra elfogadható. Az kiegészített működés által generált *hOCR* minta kimenet a következő.

```
<div class='ocr_page' id='page_1' title='image " "; bbox 0 0 1920 1080; ppageno 0'>
  <div class='ocr_carea' id='block_1_1' title='bbox 0 0 1920 1080'>
    <p class='ocr_par' dir='ltr' id='par_1_1' title='bbox 277 0 1920 1080'>
      <span class='ocr_line' id='line_1_1' title='bbox 277 0 1920 1080;
        baseline 0.012 -509'>
        <span class='ocrx_word' id='word_1_1'
          title='bbox 277 405 697 572; x_wconf 93'
          lang='eng' dir='ltr'
          orientation='0' order='2'
          angle='-0.0000' bold='false'
          italic='false' monospace='false'
          serif='true' smallcaps='false'
          pointsize='200' fontid='283' >
            Test
          </span>
        ...
```

```
</p>  
</div>  
</div>
```

Bár a *JNI* csatlakozás *Java* oldali implementációját nem kellett módosítanom, a *Java* oldalon a *HOCRParser* osztályt természetesen ki kellett egészítenem a többletinformációk beolvasásával és adatmodellbe írásával.

6. fejezet

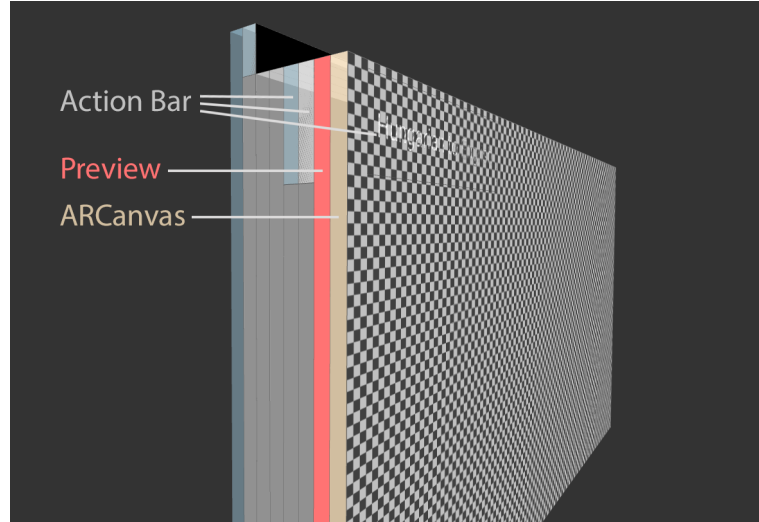
Kiterjesztett valóság

Habár a korábbi fejezetekben bemutatott komponensek mind szerves részét képezik az alkalmazásnak, a legfontosabb, a felhasználhatóságot és az alkalmazás felhasználói élményét leginkább érintő komponens a *kiterjesztett valóság* érzetét megvalósító elem. Ez a komponens az alkalmazás alapkonceptiójának szerves része, annyira, hogy még a dolgozat címében is külön fel lett tüntetve ez a technológia. Habár munkám során több esetben is támaszkodtam meglévő megoldásokra, programkönyvtárak formájában, valamint a *kiterjesztett valóság* támogatására több könyvtár is létezik (pl. a *Vuforia* [66] kimondottan ilyen), ennek a funkciónak a megvalósítását mégis teljesen magam készítettem el. Azért döntöttem így, mert így teljes egészében kontrollálhatom a megjelenítés minden részletét, és úgy szabhatom testre a működést, hogy az előzőleg bemutatott komponenseket megfelelően használhassam. A következő fejezetben az alkalmazás felületének főbb elemeit, azok működését, valamint a kiterjesztett valóság modult fogom bemutatni, miközben külön ki-
térek a megjelenítési folyamat egyes lépéseire.

6.1. Felület

A *kiterjesztett valóság* megjelenítésért felelős komponens szervesen kapcsolódik az alkalmazás felületéhez. Az alkalmazás egyetlen Android *Activity*-ből áll [15], melynek felülete több egymást elfedő rétegből épül fel. Az egymást elfedő rétegek elkészítéséhez *RelativeLayout* elrendezést használtam [30], amiben az összes tartalmazott nézet a képernyő tetejéhez van igazítva, és a felület leírásban egymás után definiált elemek egymást ugyanebben a sorrendben fedik el. A felület felépítését a 6.1 ábra szemlélteti.

A *kiterjesztett valóság* elemeit megjelenítő nézet az *ARCanvas* osztály, mely az Android *View* osztályából származik [24], éppen ezért rendelkezik az Android nézetek megfelelő tulajdonságaival, elhelyezhető felületi elemként, valamint a megfelelő eseménykezelő függvények is használhatóak rajta. Ezek közül az osztály az *onDraw(Canvas canvas)* függvényt valósítja meg. Ez az a *callback* függvény, melyet akkor hív meg a rendszer, amikor az adott nézetet megkéri, hogy rajzolja ki magát. A paraméterként átadott *Canvas* osztály egy *festővászon* [22], melyre a különböző nézeti elemek kerülnek kirajzolásra. Ennek megfelelően az *ARCanvas* osztály ezen függvényében történik meg a megjelenítési szabályoknak



6.1. ábra. A felület felépítését szemléltető háromdimenziós ábra.

megfelelő kirajzolás.

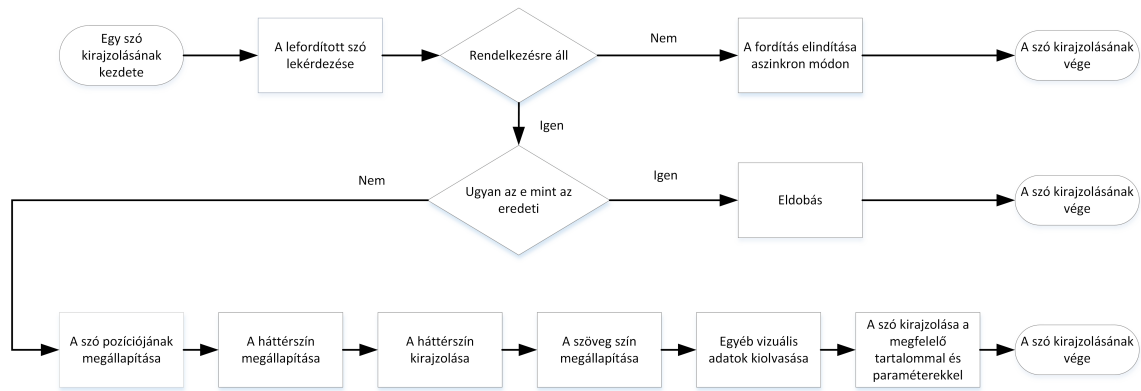
A kirajzolás során az első lépés a meglévő adatok transzformációja. Mint ahogy az már a 5.2.2 fejezetben bemutatásra került, az *ARCanvas* tartalmazza az *optikai karakterfelismerés* kimenetétül előálló *Word* objektumok listáját. Viszont az ezek által tartalmazott mértékek mind a kamera által nyújtott, majd átkonvertált és átméretezett *Preview Image* koordináta-rendszerben értelmezendők, hiszen az a kép szolgált a karakterolvasás forrásául. Ennek megfelelően, a megjelenítés előtt át kell transzformálni a különböző mértékegységeket a forráskép koordináta-rendszeréből a *festővászon* által rajzolható területre. Ez úgy történik, hogy a vízszintes mértékegységeket a vízszintes arányokkal, a függőleges mértékegységeket pedig a függőleges arányokkal kell szorozni. Az alábbi képlet mutatja egy képpont vízszintes és függőleges koordinátáinak transzformációját.

$$pixel_X^{AR} = pixel_X^{OCR} * \frac{width_{Canvas}}{width_{Preview}}$$

$$pixel_Y^{AR} = pixel_Y^{OCR} * \frac{height_{Canvas}}{height_{Preview}}$$

6.2. Kirajzolás

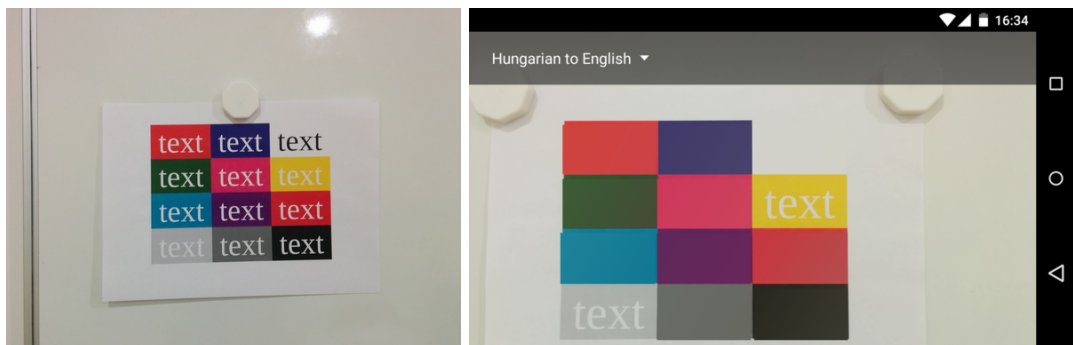
A kirajzolás során a szavak listáján végigiterál a folyamat, és minden egyes szót külön-külön dolgoz fel. Elsőként az *ARCanvas* kapcsolódik a *Translator* osztályhoz és lekérdezi az adott szóhoz tartozó lefordított szót. Amennyiben ez nem áll rendelkezésre, vagy rendelkezésre áll, de ugyanaz a szó az idegen nyelven is, akkor az adott szót kihagyja, mivel ebben az esetben nem történik fordítás, így nincs is értelme mit kirajzolni, a képen lévő eredeti szót felesleges ugyanazzal, vagy egy üres szóval elfedni.



6.2. ábra. Egy felismert szó kirajzolásának folyamata.

6.2.1. A háttér megjelenítése

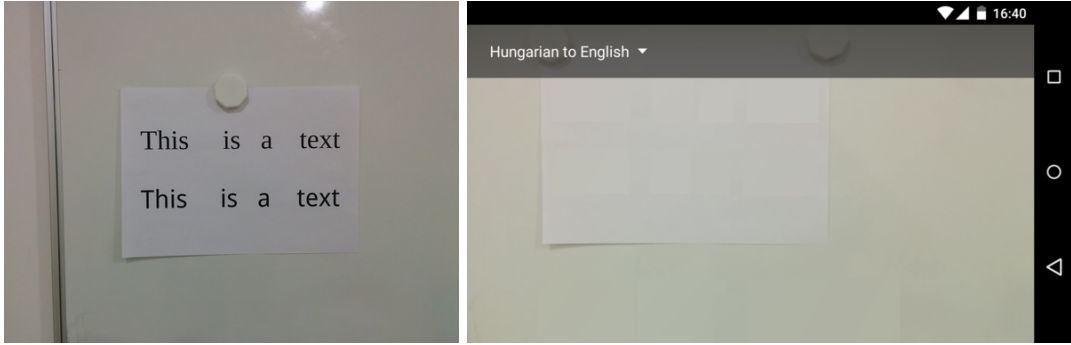
Ezután a rajzolás kezdődik, és a rajzolás első lépése a képen lévő szó eltüntetése. Ez úgy történik, hogy a szó pozícióját tartalmazó befoglaló négyzethez (*bounding box*) egy fix margót adok hozzá minden oldalról, hogy biztosan elfedje a meglévő szöveget, és ebbe a nézetbe egy kitöltő színt rajzolok. Azért, hogy ez kitöltő szín a lehető legjobban fedje el a szöveget a képen, és az eltüntetés érzetét keltse, egy a befoglaló négyzet által eltakart háttér színein alapuló megoldás használok. A kitöltő szín, egy lineáris színátmenet két szín között, melyek a befoglaló és a margó által lefedett négyzet bal felső és jobb felső sarkán lévő színek. (Ezek megtalálásához természetesen a *PreviewImage* koordináta-rendszerébe kell visszatranszformálni a pontok pozícióját). Ennek szemléltetése a 6.3 és a 6.4 ábrán található, melyen az eredeti illetve az eltüntetett szó látható. Természetesen azokat a szövegeket melyeket az optikai képfelismerő komponens nem tudott felismerni, vagy a szűrés során eldobásra kerültek nem is lehet elfedni (mint ahogy az a 6.3 ábrán észre is vehető).



6.3. ábra. Az háttér megjelenítés tesztelésének bemenete (bal), valamint a megjelenített háttér, szöveg nélkül (jobb).

6.2.2. Szövegszín detektálás

A folyamat következő lépése a szöveg színének detektálása. Bár a szöveg attribútumainak értelmezése főként a karakterfelismeréssel áll kapcsolatban, a szöveg színét sajnos nem képes detektálni a *Tesseract*. Ez azért van, mert a felismerési folyamat egyik első lépése a kép szürkeárnyaltos formátumra konvertálása, illetve *binarizációja*, mely során a kép szín



6.4. ábra. Az általános angol szöveg (bal), valamint hozzá tartozó megjelenített háttér, szöveg nélkül (jobb).

információi elvesznek. Ezért ennek a detektálására egy saját módszert használtam. Ennek lényege, hogy a szöveg színét a háttértől legjobban eltérő színként azonosítja az algoritmus.

Ez úgy történik, hogy a befoglaló négyzet minden pixelén végigiterál az algoritmus, és összehasonlítja az összes pixelt a bal felső sarokban (befoglaló négyzet és margó bal felső sarka, tekinthető háttérszínnel) található színnel. Minden egyes RGB színcsatornára megnézi [38], hogy mennyi az az érték amivel eltér a háttér adott RGB csatornájától, és ezen eltérések abszolút értékét összegzi, és ezt veszi a háttértől való távolságnak.

$$difference_{red} = |currentPixel_{red} - background_{red}|$$

$$difference_{green} = |currentPixel_{green} - background_{green}|$$

$$difference_{blue} = |currentPixel_{blue} - background_{blue}|$$

$$difference_{total} = difference_{red} + difference_{green} + difference_{blue}$$

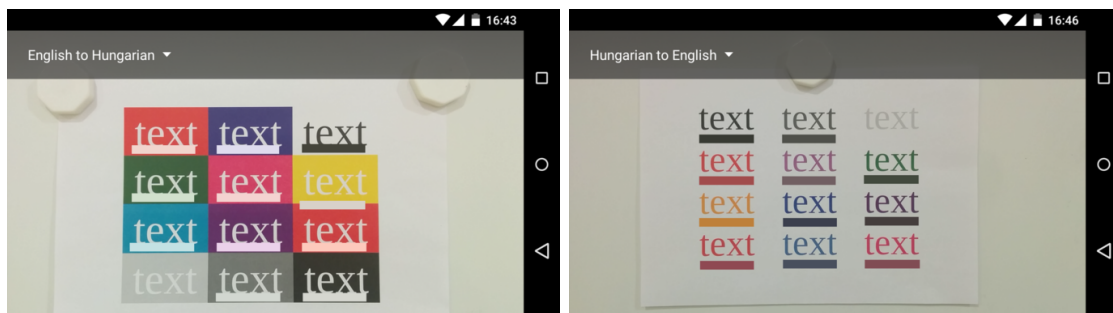
Az iteráció során tárolja az eddig talált legtávolabbi színt az algoritmus, és amint ettől távolabbi színt talál (mely nem feltétlenül ugyanabban az irányban van távolabb, hanem abszolút távolságban), akkor arra frissíti ezt az értéket. Ebből kifolyólag az algoritmus lépésszáma a pixelek számának lineáris függvénye, valamint a memória szükséglete konstans. Az alábbi kódrészlet mutatja be az algoritmus vázát, a színdetektáció eredményét pedig a 6.5 és 6.6 ábra demonstrálja, ahol demonstrációs célból a felismert szövegszínnel vannak aláhúzva a megfelelő szövegek.

```
for (int i = word.getTop(); i < word.getBottom(); i++)
{
    for (int j = word.getLeft(); j < word.getRight(); j++)
    {
        int currentPixelColor=bitmap.getPixel(i,j);
        //Calculate difference values
        ...
        if (differenceTotal>differenceFarest)
        {
            farestColor = currentPixelColor;
```

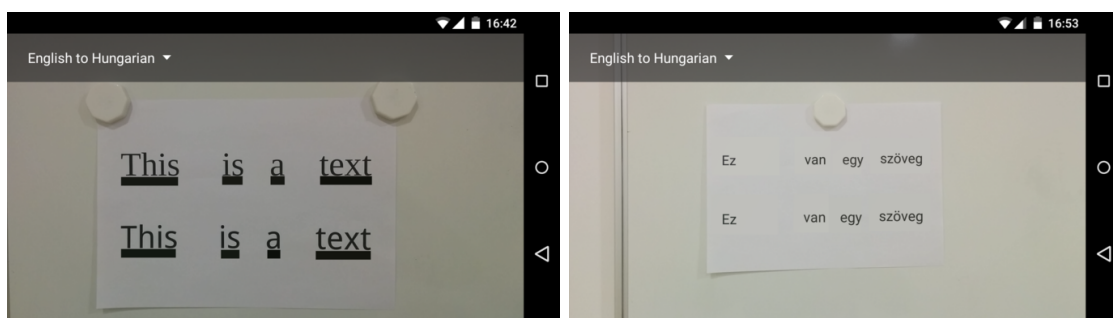
```

    }
  }
}

```



6.5. ábra. A szövegszín detektálás eredménye színes háttér (bal), illetve színes szöveg esetén (jobb).



6.6. ábra. Egy általános angol szöveg szövegszíne (bal), illetve ugyanennek a lefordított, szövegszínhelyes megjelenítése (jobb).

A folyamat következő lépése a karakterfelismerés során detektált attribútumok alkalmazása. A szöveg méretét is át kell transzformálni a *Canvas* koordináta-rendszerébe, mely a már ismertetett megoldástól annyiban tér el, hogy egyszerre kell figyelembe venni a hosszanti és szélességi arányokat, ezért a transzformáció ebben az esetben úgy történik, hogy a két irányban vett arányok átlagának 55%-át veszem mint skálázási faktort. Ez az szám egy tapasztalati úton kialakult érték, mely elég jól közelíti a valós betűméretet.

$$fontSize^{AR} = fontSize^{OCR} * \left(\frac{\frac{width_{Canvas}}{width_{Preview}} + \frac{height_{Canvas}}{height_{Preview}}}{2} \right) * 0.55$$

A szöveg *Canvas*re történő rajzolása során a kirajzolás paramétereit a *Paint* objektum tárolja [25]. A *Paint* objektum *setTypeface(...)* metódusával állítható be a betűtípus, mely létrehozása a *Typeface* osztály [35] segítségével történik. Ezen osztály létrehozásakor (*create(...)*) megadható egy betűtípus család, illetve egy betűtípus stílus attribútum. A stílus attribútum tartalmazza a félkövér (*bold*), dőlt (*italic*) illetve mindkettőt tartalmazó beállítást, a család pedig lehet talpas (*serif*), talpatlan (*sans-serif*) vagy fix szélességű (*monospaced*). A megfelelő paraméterek beállítása során a lefordított szöveget a paraméterezett *Paint* segítségével a *bounding box* bal alsó sarkától kezdve kirajzolom az alábbi módon.

```

paintText.setTypeface(Typeface.create(family, style));
paintText.setColor(farestColor);
paintText.setTextSize(fontSizeAR);
canvas.drawText(translatedText, boundingBox.left, boundingBox.bottom, paintText);

```

6.3. Főképernyő

Az alkalmazás főképernyője a már bemutatott *Preview* és *ARCanvas* elemeken túl még tartalmaz egy fontos elemet, mely a felület leírásában explicit nem szerepel. Ez az elem az *Action Bar* [14], melyet az Android rendszer ad arra, hogy az adott képernyő legfőbb funkcióit ezen keresztül érje el a felhasználó. Azért, hogy a *kiterjesztett valóság* érzetét még jobban erősítsem, ez a nézet elfedi a már említett két nézetet, és részben átlátszó háttere miatt, a kamera képre rajzolódik rá. Ezt úgy érem el, hogy az alkalmazás stílusában engedélyezem, hogy az *Action Bar* elfedhesse az alkalmazás felületet és így a már említett két nézetet (*windowActionBarOverlay* attribútum), majd a hátterét egy félig átlátszó színátmenetre állítom.

A feladatspecifikációban fontos előírás, hogy az alkalmazás támogassa az Android 2.2 és afeletti verziókat. Viszont az *Action Bar* mint beépített felületi elem, csak az Android 3.0 verzióban jelent meg, így az alatt nem használható ez a beépített komponens. Szerencsére az probléma hamar nyilvánvaló lett, és már elérhető a hivatalos *AppCompat* könyvtár [32], mely lehetővé teszi ennek a komponensnek a használatát Android 3.0 alatti verziókban is, a megoldásomban én is ezt használatom.

6.3.1. Navigáció

A főképernyő *Action Bar*ján található legördülő menü használatával biztosított a felhasználó számára a szövegfordítási irányok közötti váltás. Ahhoz, hogy ezt engedélyezzem, az *Action Bar*t kell megfelelően konfigurálni, majd beállítani a listát feltöltő adaptert a megfelelő értékekkel.

```

final ActionBar actionBar = getSupportActionBar();
...
actionBar.setNavigationMode(ActionBar.NAVIGATION_MODE_LIST);
ArrayAdapter<String> adapter = new ArrayAdapter<String>(getBaseContext(),
    android.R.layout.simple_spinner_dropdown_item, new String[]
    {getString(R.string.hunToEng), getString(R.string.engToHun)});

```

Ezután a navigációs *callback* függvényét kell definiálni. Ez a függvény akkor hívódik, amikor a felhasználó kiválaszt egy elemet a listából, és tartalmazza többek között a kiválasztott elem pozícióját. Ennek az eseménynek a hatására le kell állítani a kamerakép folyamatos olvasását és az optikai karakterfelismerést. Majd újra inicializálni kell a megfelelő nyelvű *Tesseract* példányt, és a kiválasztott nyelvekkel paraméterezett *Translator* osztályt. A *callback* függvény definiálása után az adapterrel együtt csatolni kell az *Action*

Barhoz.

```
ActionBar.OnNavigationListener navigationListener = new ActionBar.  
OnNavigationListener()  
{  
    @Override  
    public boolean onNavigationItemSelected(int pos, long id)  
    {  
        //Refresh components with the new settings  
    }  
}  
actionBar.setListNavigationCallbacks(adapter, navigationListener);
```

6.3.2. Események

Az alkalmazás használata során explicit módon a felhasználó két eseményt tud kiváltani. Az egyik ezek közül a már bemutatott nyelvváltás, a másik pedig a fókuszálás, mely a képernyő bármely részére (kivéve az *Action Bar*) történő kattintáskor aktiválódik.

Ezen kívül azonban számos olyan esemény történhet az alkalmazás használata során, melyet bár nem szándékosan, vagy nem a megszokott módon vált ki a felhasználó, az alkalmazás szempontjából mégis fontos. Nyilvánvalóan ilyen esemény a kamerakép megváltozása, hiszen ennek használatára alapvetően épít az alkalmazás, ezt az eseményt a *Camera* és a *Preview* osztály, a már bemutatott módon kezeli.

Ilyen esemény még a telefon elfordulása is, mely kezelése az alkalmazásban le van tiltva. Alapvetően az Android felületek elfordulás hatására megsemmisülnek, majd újra felépülnek. Mivel egy ilyen megsemmisülés és felépülés idővel jár, így elég zavaró lehet a felhasználónak, valamint számos többlet implementációs feladatot igényel a megvalósítása. Ezért jobbnak láttam, ha az alkalmazás csak egy nézeti irányt támogat, és a telefon elfordításának eseményére sem fordul el. A felhasználás szempontjából, mint sok más kamerát felhasználó megoldásnál, a legkézenfekvőbb irány a fekvő nézet (*landscape orientation*), mivel általában az olvasni kívánt szövegek sokkal szélesebbek, mint magasabbak, így ezen megoldás is a fekvő nézetet támogatja.

Továbbá ilyen implicit esemény az idő eltelése, ami interakció hiányában a képernyő háttérvilágítás kikapcsolását eredményezheti. Ez abban az esetben nagyon zavaró, ha a felhasználó éppen mozgatja a telefont a szöveg felé vagy olvassa a lefordított szöveget, és a képernyő eközben kikapcsol. Azért, hogy ez a viselkedés elkerülhető legyen, a főképernyőt úgy kell konfigurálni, hogy amíg az előtérben van, addig a képernyő háttérvilágítás ne kapcsoljon ki. Ezt a viselkedést az képernyő definíciójánál használt *keepScreenOn* attribútummal kapcsoltam be.

7. fejezet

Tesztelés

Mindamellett, hogy mai szoftverfejlesztési módszertanok és a fejlesztői gyakorlat is nagy hangsúlyt fektet a tesztelésre, a feladatkiírás is előírja, hogy teszteljem az elkészített szoftvert. Alkalmaztam tesztek mind a komponens fejlesztési ciklusok végén, mind az elkészült szoftver egészének tesztelésekor is.

Habár napjainkban a szoftverfejlesztés számos területén elterjedt az automatizált tesztelés, úgy gondolom, hogy a mobilszoftverek automatizált tesztelése, még nem eléggé kiforrott. Számos eszköz elérhető ezen a területen, mely egy-egy típusú tesztet támogat, viszont általános megoldást, mely lehetővé tenné minden mobilszoftver közel teljes mértékű automatizált tesztelését, nem találtam. Az alkalmazás tesztelése során, az egységek tényleges működésének teszteléséhez, szinte folyamatosan, akár egy teszt alatt is változtattam a teszteléshez használt bemenetet, hogy egy-egy speciális esetet előidézzek. Valamint az alkalmazás működése és az eredmények alapján is befolyásoltam a teszt bemeneteket. Ilyen jellegű tesztek készítése automatizált esetben számos plusz munkával járt volna, viszont úgy gondolom, hogy az alkalmazás minőségének növeléséhez nem járult volna hozzá jelentősen. Ezért döntöttem úgy, hogy automatizált tesztek helyett manuális tesztek végzek, viszont ezeket fejlesztői szinten is támogatom.

7.1. Manuális tesztek

A manuális tesztek esetén a tesztet nem egy számítógép, hanem egy ember (általánosságban egy tesztelő, jelen esetben magam) végzi el. Habár ezen tesztek elvégzése és kiértékelése nem nélkülözheti az emberi tényezőt, mégis a manuális teszt nem feltétlenül jelenti azt, hogy nem igényelne fejlesztői teendőket. Ezen szoftver esetén is ez a helyzet, mivel a manuális tesztek támogatásához számos fejlesztést végzettem.

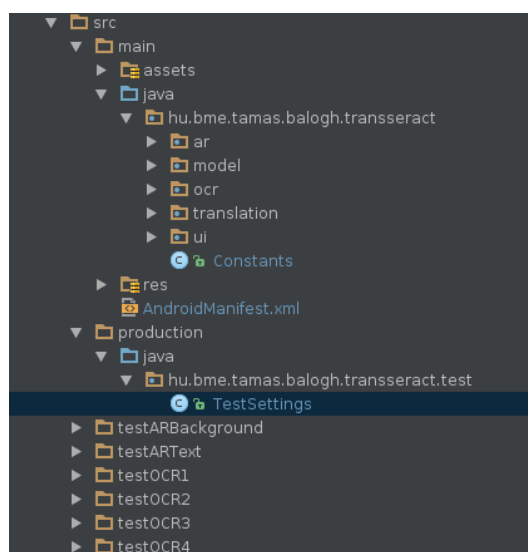
Habár a manuális tesztek elvégzésekor többségében a szoftver teljes funkcionalitását teszteltem a felhasználóknak szánt formában, bizonyos tesztekhez szükség volt arra, hogy a szoftver egy speciális, módosított változatát futtassam, mely az eredeti funkcionalitástól eltér, és kiemel olyan tényezőket, melyek a tesztelés során fontosak, viszont a felhasználóknak szánt verzióban ezek nem figyelhetők meg elég alaposan.

Azért, hogy ilyen speciális tesztváltozatokat készíthessek, a már említett *Gradle* fordító

rendszer egyik új funkcióját, a *Build Flavors*-t (*fordítási sajátosságok*) használtam fel. Ez a funkció lehetővé teszi, hogy a már megszokott fordítási típusokon túl (*Build Types*), egy újabb dimenzió mentén különítsük el a fordítás során kihasznált eltéréseket. Ez azt jelenti, hogy definiálhatóak különböző *Flavor*-ök a fordítást leíró *build.gradle* fájlban, majd ezután létre lehet hozni a projekt alap mappáján (*main*) túl, a *Flavor* nevével megegyező mappákat.

```
...
android
{
    ..
    productFlavors
    {
        production {}
        testOCR1 {}
        testOCR2 {}
        testOCR3 {}
        testOCR4 {}
        testARBackground {}
        testARText {}
    }
    ...
}
```

A fordítás során ezután kiválasztható, hogy éppen melyik *Flavor*-t szeretnénk használni. A fordítás során, a projekt alap mappájában lévő forrásokat (beleértve az erőforrásokat is), valamint a kiválasztott *Flavor* mappájában lévő forrásokat összefűzi a fordító, és így készíti el a megfelelő változatot. Ez a módszer azért nagyon jó, mert így a különböző változatok eltéréseit, a változatok saját mappájában definiálhatjuk, majd mindig az adott *Flavor*-höz tartozót használhatjuk fel a fordítás során.



7.1. ábra. A *Build Flavor*-ökkel kiegészített könyvtárstruktúra.

Név	Leírás
<i>testOCR1</i>	Az <i>optikai karakterfelismerés</i> szűrési beállításainak tesztelése, csak a találatok megjelenítése szűrés nélkül.
<i>testOCR2</i>	Az <i>optikai karakterfelismerés</i> szűrési beállításainak tesztelése, csak a találatok megjelenítése a szűrési lánc első lépésével (nem támogatott karakterek).
<i>testOCR3</i>	Az <i>optikai karakterfelismerés</i> szűrési beállításainak tesztelése, csak a találatok megjelenítése a szűrési lánc első két lépésével (nem támogatott karakterek, számok és egybetűs szavak).
<i>testOCR4</i>	Az <i>optikai karakterfelismerés</i> szűrési beállításainak tesztelése, csak a találatok megjelenítése a teljes szűrési láncot futtatva.
<i>testARBackground</i>	A <i>kiterjesztett valóság</i> beállításainak tesztelése, csak a szavak háttérének megjelenítése.
<i>testARText</i>	A <i>kiterjesztett valóság</i> beállításainak tesztelése, csak a szavak színének megjelenítése a szavak alatti aláhúzással.
<i>production</i>	A specifikációban leírt működést tesztelő változat (egyben a végső változat is).

7.1. táblázat. A manuális tesztek során használt konfigurációk.

Ezt a funkcionalitást én úgy használtam fel, hogy létrehoztam egy *TestSettings* osztályt, mely csak statikus konstansokat tartalmazott, benne egy adott teszt változat beállításával. Minden változatra, beleértve a végső változatot is (*production*) létrehoztam ezt az osztályt, viszont minden esetben a statikus konstansok értéke eltért, a teszt elvégzéséhez szükséges beállításoknak megfelelően. Ezután a programkódban ellenőrizhettem az aktuális tesztelési beállításokat, és ettől függően befolyásolhattam a különböző folyamatok működését (főként a szűrési és a megjelenítési funkcionalitás módosult ennek függvényében). Ezután a fordítás során kiválasztott változat beállításával, már futtathattam a manuális teszthez szükséges konfigurációt. A teszt konfigurációk leírását a 7.1 táblázat tartalmazza.

8. fejezet

Konklúzió

A diplomaterv feladatom elkészítése során, miután a feladatban leírt technikákat megismertem, részletesen specifikáltam a szoftverrel szemben támasztott követelményeket és elvárásokat a feladatkiírás alapján. Megvizsgáltam a már meglévő megoldásokat, ezután áttekintettem az egyes komponensekben felhasználható lehetőségeket, a különböző komponenseket egyenként megterveztem, majd implementáltam azokat. Ezután a különböző komponenseket csatoltam egymáshoz, megoldottam az ekkor fellépő kommunikációs, szinkronizációs és egyéb integrációs problémákat. Majd a felhasználói felületet illesztettem a meglévő háttérfolyamatokhoz és végül teszteltem, valamint dokumentáltam az alkalmazást. Természetesen az implementáció során is számos tesztet végeztem, ám ezek során nem az alkalmazás működésének az egészét vizsgáltam, hanem csak az éppen fejlesztés alatt álló komponens bizonyos paramétereit. A tesztek alapján kijelenthetem, hogy a feladatkiírásban megfogalmazott, valamint a specifikációban általam pontosított követelményeket az alkalmazás kielégíti.

8.1. Továbbfejlesztési lehetőségek

8.1.1. Mondatok fordítása

Habár az előre lefektetett követelményeket kielégíti az alkalmazás, és rendelkezik az előírt funkcionalitással, az alkalmazás képességei még számos helyen kibővíthetőek. Ilyen lehetőség például, a szavankénti fordítás helyett, opcionálisan a mondatonkénti fordítás támogatása. Ehhez viszont nem csak a fordításért felelős komponenst kell részben átalakítani, hanem a megjelenítésért felelős megoldást is. A mondatonkénti fordítás esetén például előfordulhat, hogy több, illetve kevesebb szóból áll a lefordított szöveg mint az eredeti. Ezért a megjelenítés során ezt figyelembe kell venni, például az összes szót befoglaló négyzetek keresésével, vagy további szavak elhelyezésével olyan helyekre, ahol nem takarnak el információt.

8.1.2. További nyelvek támogatása

Hasznos fejlesztési irány lehet a további nyelvek támogatása. Habár a fordító és a karakterfelismerő komponens is képes lenne további nyelvek kezelésére, a képfeldolgozás során

használt szűrési algoritmust, illetve a megjelenítéshez használt komponenst is át kellene alakítani, hogy további nyelveket tudjon támogatni. Az *optikai karakterfelismerés* kimenetét szűrő megoldás jelenleg csak az angol és magyar nyelvek szabályai szerint szűri a találatokat. A további nyelvek támogatáskor célszerű lenne a többi támogatott nyelv szabályait egyenként megvizsgálni és a fordítási nyelvtől függő (futás közben változtatható) szűrési módszereket alkalmazni.

8.1.3. Vizuális attribútumok bővítése

A felhasználói élményt, a megjelenítés során használt további vizuális attribútumokkal lehetne leginkább növelni. Ilyen például a szavak elforgatásának mind a kamerakép síkjában, mind térben való detektálása, majd ettől függően történő megjelenítése. Így, ha egy szöveget nem pont szemből olvas a felhasználó, akkor a megjelenítés sem szemből történne. Nyilván ezt csak egy bizonyos mértékig lehetne használni, hiszen egy adott szög felett már nem ismerhető fel elég pontosan a szöveges tartalom. Viszont mivel általánosságban egy kisebb szögű eltérésről van szó, ez a felhasználó számára még inkább növelné a *kiterjesztett valóság* érzését. Továbbá, nem csak a betűtípus jellemzőinek detektálása, de magának a betűtípusnak a megtalálása és felhasználása a megjelenítés során is nagyban növelné a felhasználói élményt.

8.1.4. Fordítási lehetőségek kibővítése

A fordítókomponens jelen megoldásban nyelvi szövegfordítást, tehát a szavak egyik nyelvről a másikra történő fordítását támogatja, mivel a feladatkiírás ezt írja elő. Mindazonáltal, úgy gondolom, hogy a fordítást másképp is lehetne értelmezni, ennek a komponensnek az továbbfejlesztésével. Ez azt jelenti, hogy nem csak egyik nyelvről a másikra történő fordítást lehetne támogatni, hanem bármely olyan megoldást is, ami egy adott karakterlánc egy másik karakterlánccá történő átalakítását követeli meg.

Ilyen ötlet például, a számok fordítása, mely segítségével, lehetőség nyílna különböző számrendszerek (decimális, bináris, hexadecimális stb.), különböző számokat jelző karakterek (arab számok, római számok, hindu-arab számok stb.), illetve bármely számokat számokba konvertáló megoldás (pénzváltás, mértékegységváltás) megvalósítására *optikai karakterfelismerés* és *kiterjesztett valóság* segítségével. Valamint ez az irány nem csak számok közötti, hanem szövegek közötti, viszont nem tényleges fordítással járó konverzióra is kiterjeszthető. Egy ilyen megoldás például a különböző írásrendszerek (arab, orosz, kínai, székely-magyar rovásírás stb.) használatát oktató alkalmazások készítésére is alkalmas lehet. Az ilyen megoldások *optikai karakterfelismeréssel* és *kiterjesztett valósággal* történő megvalósítására csak csekély számú alkalmazás képes, és úgy gondolom, hogy számos potenciál rejlik ezekben a lehetőségekben.

Köszönetnyilvánítás

Ezúton is szeretném megköszönni konzulenseimnek, Dr. Ekler Péternek és Fehér Marcellnek a diplomaterv elkészítése során nyújtott lelkiismeretes segítségüket és hasznos tanácsaikat. Köszönettel tartozom számos hallgatótársamnak, köztük Román Tamásnak, Tóth Dánielnek és Vágó Tamásnak a dolgozat elkészítése során nyújtott segítségükért. Továbbá szeretném megköszönni Daku Istvánnénak a lektorálást.

Ábrák jegyzéke

1.	A Google Glass, a kiterjesztett valóságot támogató okosszemüveg. [6]	9
2.	Az AcrossAir és a Google Sky Map használat közben. [1] [7]	9
3.	Az Layar és az AR Defender alkalmazások, a kiterjesztett valóság széleskörű felhasználásának példái. [8] [2]	10
4.	Az optikai karakterfelismerés általános felhasználásnak egy példája. Bal oldalon az olvasott kép, jobb oldalon pedig a detektált szöveg. [9]	11
5.	Egy példa a CAPTCHA megoldásokban felhasznált képekre. [5]	11
1.1.	Az alkalmazás használati eset diagramja.	16
2.1.	Az Waygo alkalmazás többféle kiterjesztett valóság alapú megjelenítési mód is támogat. [10]	18
2.2.	Az CamDictionary alkalmazás használat közben. [4]	18
2.3.	A Babelshot alkalmazás felülete. [3]	19
2.4.	A Bing Translator alkalmazás főbb képernyői. [57]	19
2.5.	Az World Lens alkalmazás használat közben [11]	20
3.1.	A Tesseract könyvtár elérése Android környezetben.	24
3.2.	A nyelvi fájlok betöltésének és a Tesseract inicializálásának szekvencia diagramja	27
4.1.	Az alkalmazás komponensei és a köztük lévő kapcsolat.	28
4.2.	Az hozzáférési <i>token</i> megújításának szekvencia diagramja.	33
4.3.	A fordítás szekvencia diagramja.	35
4.4.	Az fordítás és a gyorsítótárazás során használt főbb osztályok egyszerűsített statikus struktúra diagramja.	36
5.1.	Az optikai karakterfelismerés feldolgozása után kapott adat modell egyszerűsített statikus struktúra diagramja.	44
5.2.	Az karaktfelismerés során talált adatok szűrésének folyamata.	44
5.3.	Az optikai karakterfelismerés eredménye szűrés nélkül (bal), valamint az első szűrés lépés alkalmazásával (jobb).	45
5.4.	Az optikai karakterfelismerés eredménye az első kettő szűrés lépéssel (bal), valamint a teljes szűrés láncsal (jobb).	46

5.5. Az aszinkron optikai karakterfelismerési folyamat egyszerűsített szekvencia diagramja.	47
6.1. A felület felépítését szemléltető háromdimenziós ábra.	51
6.2. Egy felismert szó kirajzolásának folyamata.	52
6.3. Az háttér megjelenítés tesztelésének bemenete (bal), valamint a megjelenített háttér, szöveg nélkül (jobb).	52
6.4. Az általános angol szöveg (bal), valamint hozzá tartozó megjelenített háttér, szöveg nélkül (jobb).	53
6.5. A szövegszín detektálás eredménye színes háttér (bal), illetve színes szöveg esetén (jobb).	54
6.6. Egy általános angol szöveg szövegszíne (bal), illetve ugyanennek a lefordított, szövegszínhelyes megjelenítése (jobb).	54
7.1. A Build Flavor-ökkel kiegészített könyvtárstruktúra.	58

Táblázatok jegyzéke

4.1.	A hozzáférési token igénylésének paraméterei.	30
4.2.	A hozzáférési token igénylés válaszáinak paraméterei.	30
4.3.	A fordításhoz használt kérés paraméterei.	32
7.1.	A manuális tesztek során használt konfigurációk.	59

Irodalomjegyzék

- [1] AcrossAir Image (2014 december 10.). http://acrossair.com/_ui/images/acrossair_nearesttube.jpg.
- [2] AR Defender Image (2014 december 10.). <http://www.igoiphone.com/wp-content/uploads/2010/09/005-AR-Defender.jpg>.
- [3] Babelshot (photo translator) (2014 december 10.). <https://play.google.com/store/apps/details?id=com.intsig.camdict>.
- [4] CamDictionary (2014 december 10.). <https://play.google.com/store/apps/details?id=com.intsig.camdict>.
- [5] CAPTCHA Sample (2014 december 10.). <http://upload.wikimedia.org/wikipedia/commons/thumb/b/b6/Modern-captcha.jpg/290px-Modern-captcha.jpg>.
- [6] Google Glass Image (2014 december 10.). http://upload.wikimedia.org/wikipedia/commons/b/be/Google_Glass_with_frame.jpg.
- [7] Google Sky Map Image (2014 december 10.). <http://emosqueira.files.wordpress.com/2010/12/polarisnexus.jpg?w=593>.
- [8] Layar Image (2014 december 10.). <http://www.opengardensblog.futuretext.com/wp-content/uploads/2012/06/layar-interactive-print.jpg>.
- [9] Optical Character Recognition Sample (2014 december 10.). <http://www.townswebarchiving.com/wp-content/uploads/2010/09/OCR-sample-comparison.jpg>.
- [10] Translator, Dictionary - Waygo (2014 december 10.). <https://play.google.com/store/apps/details?id=com.waygoapp.waygo>.
- [11] Word Lens Translator (2014 december 10.). <https://play.google.com/store/apps/details?id=com.questvisual.wordlens.demo>.
- [12] ABBYY. ABBYY Mobile OCR Engine (2014 december 10.). <http://www.abbyy.com/mobileocr/>.
- [13] AcrossAir. AcrossAir (2014 december 10.). <http://www.acrossair.com/>.

- [14] Android Developers. Action Bar (2014 december 10.). <http://developer.android.com/guide/topics/ui/actionbar.html>.
- [15] Android Developers. Activities (2014 december 10.). <http://developer.android.com/guide/components/activities.html>.
- [16] Android Developers. Android NDK (2014 december 10.). <https://developer.android.com/tools/sdk/ndk/index.html>.
- [17] Android Developers. Android SDK (2014 december 10.). <https://developer.android.com/sdk/index.html?hl=i>.
- [18] Android Developers. Android Studio (2014 december 10.). <https://developer.android.com/sdk/installing/studio.html>.
- [19] Android Developers. AsyncTask (2014 december 10.). <http://developer.android.com/reference/android/os/AsyncTask.html>.
- [20] Android Developers. BroadcastReceiver (2014 december 10.). <http://developer.android.com/reference/android/content/BroadcastReceiver.html>.
- [21] Android Developers. Camera (2014 december 10.). <http://developer.android.com/guide/topics/media/camera.html>.
- [22] Android Developers. Canvas and Drawables (2014 december 10.). <http://developer.android.com/guide/topics/graphics/2d-graphics.html>.
- [23] Android Developers. Controlling the Camera (2014 december 10.). <http://developer.android.com/training/camera/cameradirect.html>.
- [24] Android Developers. Creating a View Class (2014 december 10.). <http://developer.android.com/training/custom-views/create-view.html>.
- [25] Android Developers. Custom Drawing (2014 december 10.). <http://developer.android.com/training/custom-views/custom-drawing.html>.
- [26] Android Developers. DefaultHttpClient (2014 december 10.). <http://developer.android.com/reference/org/apache/http/impl/client/DefaultHttpClient.html>.
- [27] Android Developers. Handler (2014 december 10.). <https://developer.android.com/reference/android/os/Handler.html>.
- [28] Android Developers. HttpURLConnection (2014 december 10.). <http://developer.android.com/reference/java/net/HttpURLConnection.html>.
- [29] Android Developers. Intent (2014 december 10.). <http://developer.android.com/reference/android/content/Intent.html>.

- [30] Android Developers. RelativeLayout (2014 december 10.). <http://developer.android.com/guide/topics/ui/layout/relative.html>.
- [31] Android Developers. Saving Key-Value Sets (2014 december 10.). <http://developer.android.com/training/basics/data-storage/shared-preferences.html>.
- [32] Android Developers. Support Library Features (2014 december 10.). <https://developer.android.com/tools/support-library/features.html>.
- [33] Android Developers. SurfaceHolder (2014 december 10.). <http://developer.android.com/reference/android/view/SurfaceHolder.html>.
- [34] Android Developers. SurfaceView (2014 december 10.). <http://developer.android.com/reference/android/view/SurfaceView.html>.
- [35] Android Developers. Typeface (2014 december 10.). <http://developer.android.com/reference/android/graphics/Typeface.html>.
- [36] Android Source. ART and Dalvik (2014 december 10.). <http://source.android.com/devices/tech/dalvik/index.html>.
- [37] Aspire Software. Java OCR and Barcode Recognition (2014 december 10.). <http://asprise.com/royalty-free-library/java-ocr-api-overview.html>.
- [38] Equasys. Color Formats (2014 december 10.). <http://www.equasys.de/colorformat.html>.
- [39] Eranga Bandara. Tesseract with andoird and gradle (2014 december 10.). <https://coderwall.com/p/eurvaq/tesseract-with-andoird-and-gradle>.
- [40] V. Fragoso, S. Gauglitz, S. Zamora, J. Kleban, and M. Turk. Translatar: A mobile augmented reality translator. In *Applications of Computer Vision (WACV), 2011 IEEE Workshop on*, pages 497–502, Kona, HI, USA, January 5–7 2011.
- [41] Gengo. Translation API by Gengo (2014 december 10.). <http://gengo.com/developers/>.
- [42] Google. Google Fordító (2014 december 10.). <https://translate.google.com/>.
- [43] Google. Google Glass (2014 december 10.). <https://www.google.com/glass/start/>.
- [44] Google. Google Gson (2014 december 10.). <https://code.google.com/p/google-gson/>.
- [45] Google. Google Sky Map (2014 december 10.). <http://www.google.com/mobile/skymap/>.
- [46] Google. Google Translate API (2014 december 10.). <https://cloud.google.com/translate/docs>.

- [47] Google Code. tesseract-ocr (2014 december 10.). <https://code.google.com/p/tesseract-ocr/>.
- [48] Gradleware. Gradle - Build Automation Evolved (2014 december 10.). <http://www.gradle.org/>.
- [49] I. Haritaoglu. Scene text extraction and translation for handheld devices. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, pages 408–413, 2001.
- [50] IDC Corporate USA. Smartphone OS Market Share, Q3 2014 (2014 december 10.). <http://www.idc.com/prodserv/smartphone-os-market-share.jsp>.
- [51] int13. AR Defender (2014 december 10.). <http://www.ardefender.com/>.
- [52] ECMA International. The JSON Data Interchange Format (2014 december 10.). <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>.
- [53] Internet Engineering Task Force. Hypertext Transfer Protocol – HTTP/1.1 (2014 december 10.). <http://tools.ietf.org/html/rfc2616>.
- [54] INTSIG. CamDictionary (2014 december 10.). <http://www.intsig.com/en/products/camdictionary>.
- [55] JPEG.ORG. Overview of JPEG (2014 december 10.). <http://www.jpeg.org/jpeg/index.html>.
- [56] Layar. Layar (2014 december 10.). <https://www.layar.com/>.
- [57] Microsoft. Bing Fordító (2014 december 10.). <http://www.bing.com/translator/phone/>.
- [58] Microsoft. Microsoft Translator (2014 december 10.). <http://msdn.microsoft.com/en-us/library/dd576287.aspx>.
- [59] Microsoft. Microsoft Translator (2014 december 10.). <https://datamarket.azure.com/dataset/bing/microsofttranslator>.
- [60] Mori, S. ; Suen, C.Y. ; Yamamoto, K. Historical review of OCR research and development. In *Proceedings of the IEEE*, pages 1029–1058, July 1992.
- [61] Olivia Craciunescu and Constanza Gerding-Salas and Susan Stringer-O’Keeffe. Machine Translation and Computer-Assisted Translation: a New Way of Translating? <http://translationjournal.net/journal/29computers.htm>, 2004.
- [62] Oracle. Annotations (2014 december 10.). <https://docs.oracle.com/javase/1.5.0/docs/guide/language/annotations.html>.
- [63] Oracle. Java Native Interface (2014 december 10.). <https://docs.oracle.com/javase/6/docs/technotes/guides/jni/>.

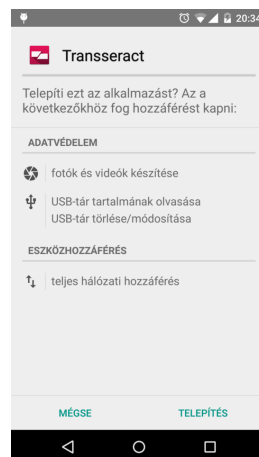
- [64] Oracle. Thread (2014 december 10.). <https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html>.
- [65] Oracle. Trail: The Reflection API (2014 december 10.). <http://docs.oracle.com/javase/tutorial/reflect/>.
- [66] Qualcomm. Vuforia Features (2014 december 10.). <https://www.qualcomm.com/products/vuforia/features>.
- [67] Robert Theis. tess-two (2014 december 10.). <https://github.com/rmtheis/tess-two>.
- [68] Simple. Simple XML Serialization (2014 december 10.). <http://simple.sourceforge.net/>.
- [69] Smart Mobile Software. Online OCR API (2014 december 10.). <http://ocrapiservice.com/>.
- [70] Sun Microsystems. Simple API for XML (2014 december 10.). <https://docs.oracle.com/javaee/1.4/tutorial/doc/JAXPSAX.html>.
- [71] Thomas Breuel. The hOCR Embedded OCR Workflow and Output Format (2014 december 10.). https://docs.google.com/document/d/1QQnIQtvDAC_8n92-LhwPcjtAUFwBlzE8EWnKAxlgVf0/preview#heading=h.77bd784474e5.
- [72] Translate Abroad. Waygo (2014 december 10.). <http://www.waygoapp.com/>.
- [73] Travis Leithead. DOM Parsing and Serialization (2014 december 10.). <http://www.w3.org/TR/DOM-Parsing/>.
- [74] WebserviceX. Translation Service Web Service (2014 december 10.). <http://www.webservicex.net/translateservice.asmx?op=Translate>.

Függelék

F.1. Telepítési útmutató

Az alkalmazás telepítéséhez a mellékletben található telepítő fájl szükséges. Ez a melléklet *Install* mappájában található *Transseract.apk* fájl. Mivel az alkalmazás csak Android 2.2 vagy attól frissebb Android operációs rendszerrel rendelkező telefonokat, illetve táblagépeket támogat, így az alkalmazás telepítője is csak az ezt teljesítő telefonokra engedi feltelepíteni az alkalmazást. Továbbá az alkalmazás működésének *hardware-es* alapkövetelménye az *auto-fókusszal* rendelkező kamera, így olyan készülékekre, melyek ilyennel nem rendelkeznek szintén nem engedi a telepítő az alkalmazást feltelepíteni.

Mivel az Android operációs rendszer alapvetően az alkalmazásboltból való alkalmazás-telepítést támogatja, ezért ahhoz, hogy az alkalmazásboltból kívülről tudjunk telepíteni, engedélyezni kell a külső forrásokból származó alkalmazások telepítését. Ezt a telefon beállításáiban (Beállítások > Személyes > Biztonság > Ismeretlen források) tudjuk engedélyezni. Ennek hiányában, a rendszer nem fogja engedni, hogy a telepítő elinduljon.



Telepítés előtt az alkalmazás engedélyeit jóvá kell hagyni.

A telepítés maga kétféleképpen történhet. Első módszer során a telepítő fájlt a telefonra kell másolunk. Ez történhet akár USB kábelén keresztül, illetve bármilyen más fájlátviteli módszerrel (fájlmegosztás, *cloud-alapú* fájllelés stb.). Ezután a telefonon meg kell nyitnunk a telepítő fájlt és követni a telepítő utasításait. Ebben az esetben a rendszer telepítőmegoldását használjuk fel. Ekkor a felhasználónak el kell fogadnia a használatához szükséges engedélyeket (kamera, internet és háttértár elérés). Ennek hiányában nem telepíthető az alkalmazás. Amint a felhasználó ezeket engedélyezi, az alkalmazás feltelepül és

használható.

A másik módszer során az *Android SDK*-hoz mellékelt *ADB* (**A**ndroid **D**ebug **B**ridge) eszköz használható. Ez egy parancssoros eszköz, mely segítségével, egy paranccsal rámásolhatjuk és telepíthetjük a fájlt az USB kábellel csatlakoztatott telefonra. Ennek használatához engedélyezni kell a telefonon, a fejlesztői beállításokban, a fejlesztői hozzáférést. Ebben az esetben a szükséges engedélyek automatikusan elfogadásra kerülnek. A telepítéshez használt *ADB* parancs a következő:

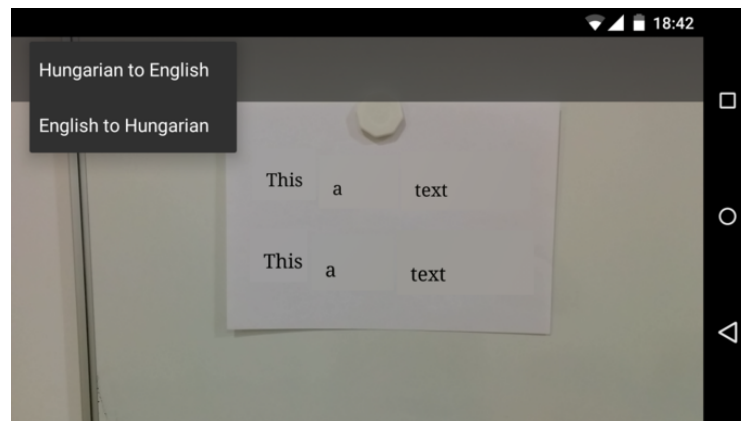
```
adb install ./Install/Transseract.apk
```

F.2. Felhasználói kézikönyv

Az alkalmazás elindítása után, az alkalmazás egyetlen képernyőből álló felülete fogadja a felhasználót. A fordítási folyamat az indítás után 1-2 másodperccel el is indul, azt külön nem kell a felhasználónak elindítania. Fontos, hogy az alkalmazás használatához már az indításkor is internetkapcsolat szükséges.

F.2.1. Nyelvváltás

A fordítási irány átállítása a felső sáv bal részén található navigációs elemmel lehetséges. Miután a felhasználó kiválasztja ezt az elemet, egy legördülő menüből választhatja ki a fordítás irányát. A mennyiben olyan irányt választ, ami előzőleg is ki volt választva, a program továbbra is ugyanúgy viselkedik, amennyiben más fordítási irányt választ, a rendszer beállítja a megfelelő komponenseket (kevesebb mind egy másodperc, közben a fordítás és a megjelenítés nem használható). Majd az alkalmazás tovább működik az új nyelvvel.



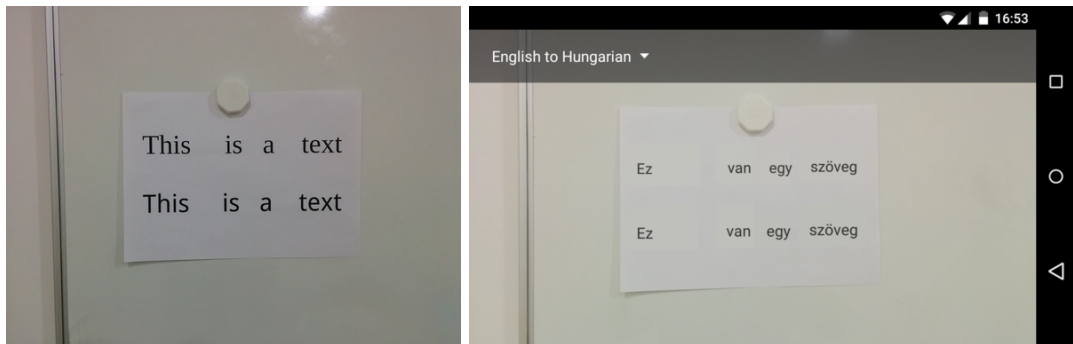
Nyelvváltás a navigációs elem segítségével.

F.2.2. Fókuszálás

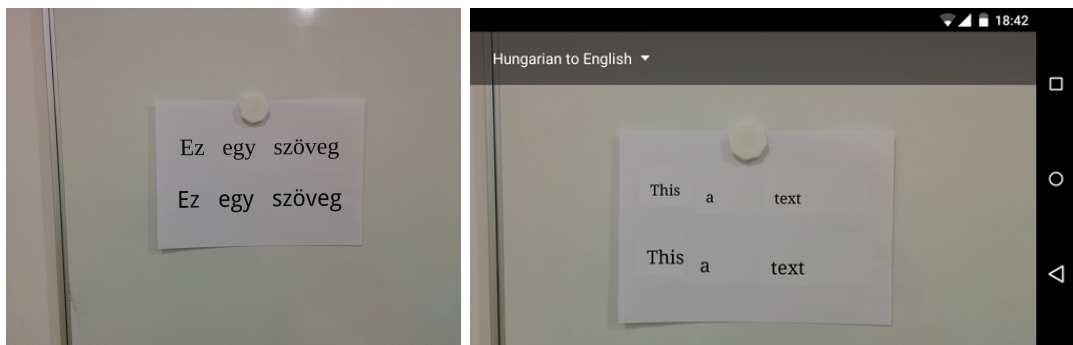
Az alkalmazás lehetőséget nyújt a felhasználónak az automatikus fókuszálás indítására. Ez úgy történik, hogy a felhasználó a lefordítani kívánt szövegre irányítja a telefon hátlapi kameráját, majd a felületre kattint (bárhova, a felső elsötétített sáv kivételével). Ezután megtörténik az automatikus fókuszálás.

F.3. Tesztelési eredmények

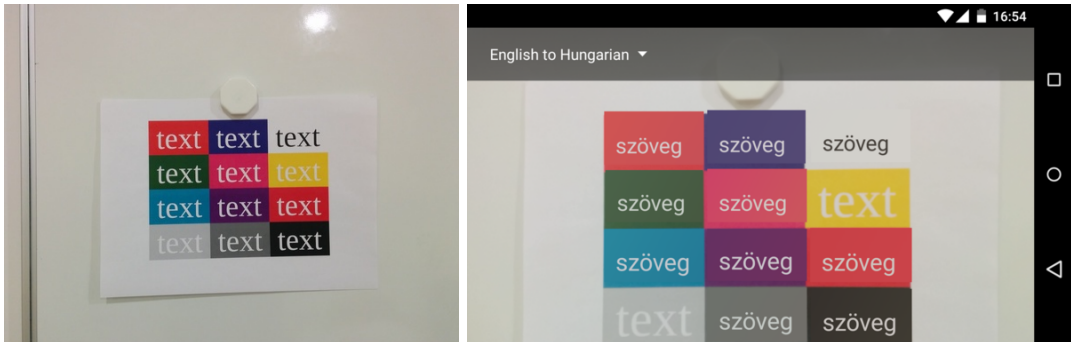
Az alkalmazás tesztelése során számos bemenetet kipróbáltam. Mind direkt beállított (ki-fejezetten a tesztelés céljából készített), mind életszerű (könyvek, jegyzetek, mindennapi feliratok) bemenettel teszteltem az alkalmazást. Az alkalmazást egy LG Nexus 5 okostelefonnal, valamint Android 5.0 operációs rendszerrel teszteltem. Az alábbi képek (bal oldalon a fordítás nélküli, jobb oldalon pedig a lefordított) a tesztelés során kapott eredményeket mutatják be.



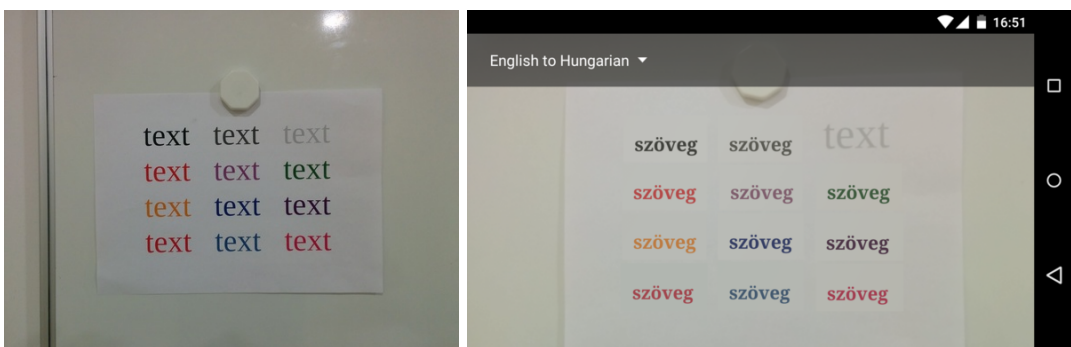
Tesztelés egy általános angol nyelvű szöveggel.



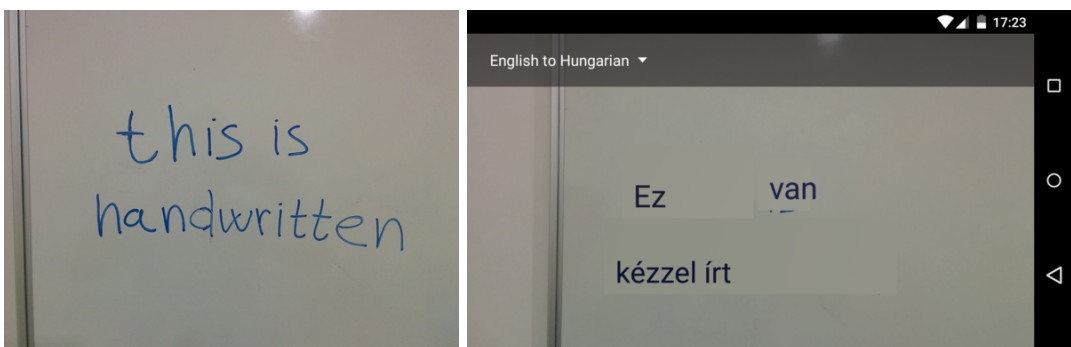
Tesztelés egy általános magyar nyelvű szöveggel.



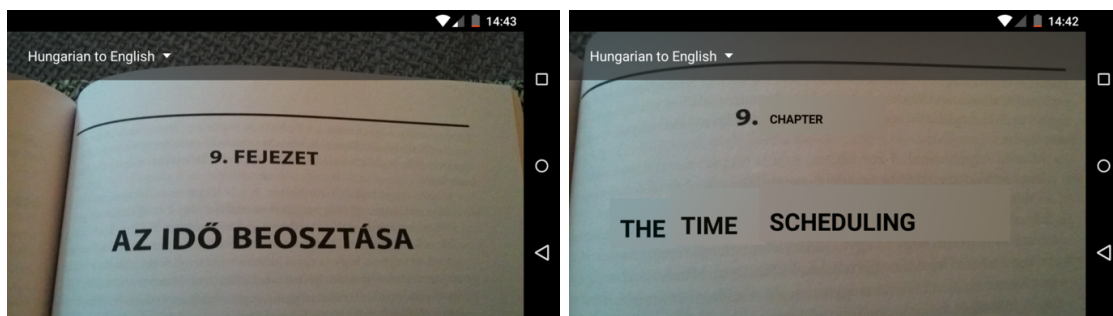
Tesztelés színes háttérű szöveggel.



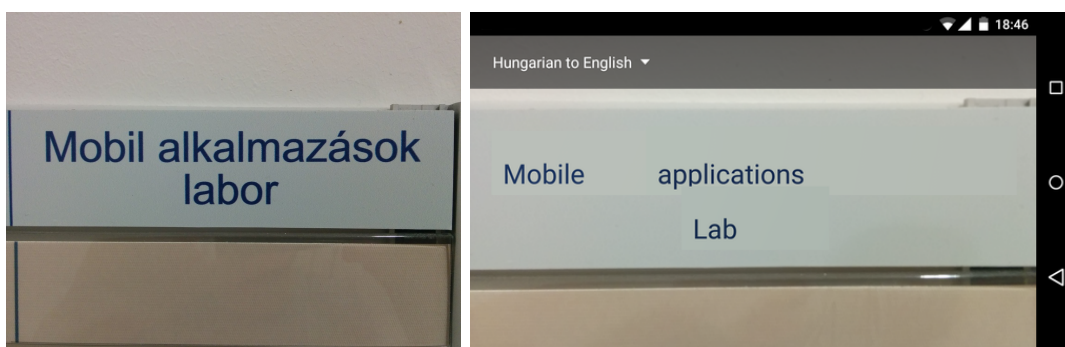
Tesztelés színes szöveggel.



Tesztelés kézzel írt angol szöveggel.



Tesztelés egy magyar nyelvű könyv fejezetcímével.



Tesztelés egy magyar feliratot tartalmazó táblával.