SZÉCHENYI ISTVÁN EGYETEM

MŰSZAKI TUDOMÁNYI KAR

TÁVKÖZLÉSI TANSZÉK

SZAKDOLGOZAT

Braille-írás felismerő és felolvasó app fejlesztése Androidra

Grajzel Dávid

RZNZ7H

Villamosmérnöki BSc szak

Komplex automatizálási rendszerek szakirány

Győr, 2018

Feladatkiírás

Szakdolgozat címe:	**Braille-írás felismerő és felolvasó app fejlesztése Androidra**
Hallgató neve:	**Grajzel Dávid**
Szak:	**Villamosmérnök**
Képzési szint:	**BSc**
Típus:	**nyilvános**

A szakdolgozat feladat leírása:

A szakdolgozat célja egy olyan app fejlesztése Android operációs rendszert futtató telefonokra, mely a mobiltelefon kameráját felhasználva a kamera által megfigyelt Braille írásról alkotott élőképet szöveggé konvertálja, majd a hangszórón keresztül beszédszintetizátor segítségével felolvassa.

A legfontosabb részfeladatok:

- Irodalomkutatás a témában
- A probléma/feladat kifejtése, analizálása
- A probléma/feladat gyakorlati megvalósítása (a szoftver elkészítése)
- Rövid használati útmutató készítése a szoftverhez
- Az elkészült app tesztelése/kiértékelése.

Győr, 2018. május 7.

belső konzulens

tanszékvezető

Értékelő lap

Szakdolgozat címe: **Braille-írás felismerő és felolvasó app fejlesztése Androidra**

Hallgató neve: **Grajzel Dávid**

Szak: **Villamosmérnök**

Képzési szint: **BSc**

Típus: **nyilvános**

A szakdolgozat beadható / nem adható be: (a nem kívánt rész törlendő)

_____ _____

dátum aláírás (belső konzulens)

A szakdolgozat bírálatra bocsátható / nem bocsátható: (a nem kívánt rész törlendő)

_____ _____

dátum aláírás (tanszékvezető)

A bíráló:

- Neve:
- Munkahelye:
- Beosztása:

A bíráló javaslata:

_____ _____ _____

dátum érdemjegy aláírás (bíráló vagy tszv.)

A belső konzulens javaslata:

_____	_____	_____
dátum	érdemjegy	aláírás (belső konzulens)

A ZVB döntése:

_____	_____	_____
dátum	érdemjegy	aláírás (ZVB elnök)

Nyilatkozat

Alulírott, Grajzel Dávid, Villamosmérnök, BSc szakos hallgató kijelentem, hogy a Braille-írás felismerő és felolvasó app fejlesztése Androidra című szakdolgozat feladat kidolgozása a saját munkám, abban csak a megjelölt forrásokat, és a megjelölt mértékben használtam fel, az idézés szabályainak megfelelően, a hivatkozások pontos megjelölésével.

Eredményeim saját munkán, számításokon, kutatáson, valós méréseken alapulnak, és a legjobb tudásom szerint hitelesek.

_____ _____

dátum hallgató aláírása

Összefoglaló

A szakdolgozatomban a Braille-írás szabályait és felépítését vizsgálom, s ennek alapján készítek egy olyan szoftvert, mely a Braille-írásról készült fénykép (legyen ez akár egy telefon kamerájának élőképe, akár egy számítógépes fájl) alapján meghatározza az írás szövegét, vagyis látó olvasók számára könnyen olvashatóvá teszi. Ez a modul fogja alkotni a szakdolgozat gerincét.

Miután elkészült, összekapcsolom egy Androidos telefonnal, melyre egy egyszerű alkalmazást írva lehetővé teszem a kamera használatát, valamint azt, hogy a szöveget ne csak megjelenítsük, hanem a telefon azt fel is olvassa. Ezáltal vak és gyengénlátó társaink élete is könnyebbé válhat.

A dolgozat középső része a szoftver működésének részletes bemutatása, melynek során nem csupán a Mit? kérdésre kapunk választ, hanem igyekszem a miérteket is részletesen megindokolni, ahol lehetséges természetesen a megfelelő forrásokkal alátámasztva.

A dolgozat utolsó szakaszában értékelem a munkámat, megállapítom, hogy sikerült-e a tervezett célokat elérni, illetve milyen hatásfokkal működik az algoritmus. Amennyiben problémák merültek fel, melyeket nem sikerült megoldani, vagy a tervezetthez képest komolyabb szerkezeti változások történtek, azt szintén itt fogom részletezni.

Summary

Developing a Braille-text reader application for Android devices

In my thesis I will analyze the rules and structures of the Braille text itself and based on these, I will develop an application, that can convert a picture, that contains a Braille-text (may it be a picture stored on the hard drive of a PC or a picture taken from the live view of a phone camera) to its human-readable text form. This library will be the main part of the work.

After it has been made, I will connect it to a lightweight GUI developed for an Android device and I will make it possible not just to convert the Braille-text to its normal text form, but to convert it to sound using the phone's built-in Text-to-Speech engine. This way we can make it possible for blind users to read Braille-texts without touching them or without having the knowledge, how to read them.

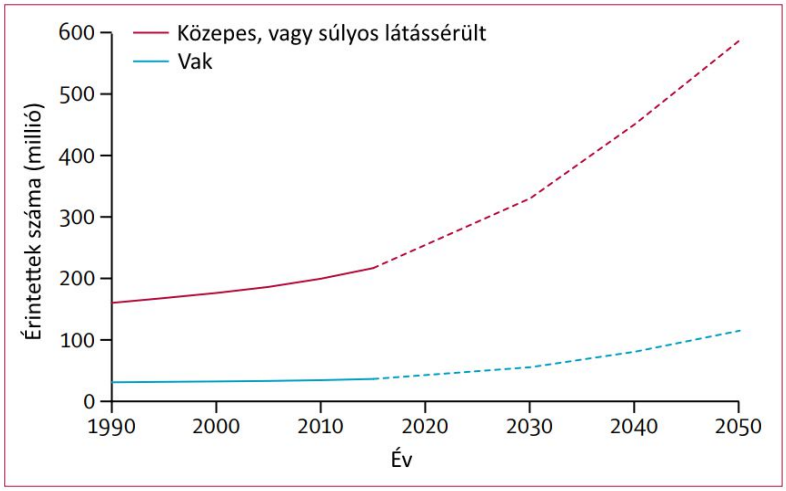
The middle part of my thesis is about the detection library, where I will try to answer all the possible questions regarding to the decisions I've made, not just list up, what the software is doing. I'll include scientific references, wherever it's possible.

In the last part of the work, I'll analyze my own work and I'll check, whether the target goals set at the beginning have been met. I'll also measure the speed and quality of the algorithm and if any major changes have been made during the development, that affect our original plan, they will be discussed here too.

Tartalomjegyzék

1. Bevezetés

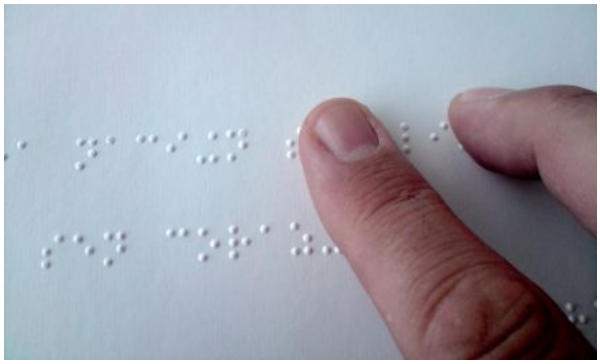
Napjainkban mintegy 33 ezer vak él Magyarországon, a látássérültek száma pedig meghaladja a 218 ezret [1]. Világviszonylatban 36 millió vak, 217 millió közepes és súlyos látássérült, valamint 188 millió enyhe látássérült élt 2015-ben. 2050-re a vakok száma kutatások szerint 115 millióra fog emelkedni, elsősorban a várható élettartam növekedése és az ezzel lépést tartani nem tudó, rossz egészségügyi ellátás miatt [2]. Napjainkban az Amerikai Egyesült Államok lakóinak évente 139 milliárd dollár kiadást (ez több, mint a teljes magyar államadósság) jelentenek a látásproblémák [3]. A probléma tehát óriási, s bár az optimális megoldás a látássérültek életminőségének javítására természetesen a teljes értékű látás visszaadása lenne, ameddig ez az orvostudomány számára nem megoldható, törekednünk kell fogyatékkal élő társaink életminőségének javítására. Számukra a mindennapi tevékenységek, mint például az autóvezetés, bevásárlás, olvasás, jelentős problémát jelentenek, különösen, ha látásukat idősebb korukban vesztették el és nem tudtak megtanulni a helyzettel együtt élni. A hétköznapi ember a vakokról fehér bot és a vakvezető kutya mellett talán a Braille-írásra asszociál leggyakrabban, melynek története 200 évvel ezelőttre nyúlik vissza.



1. ábra. Látássérültek számának növekedése az idő előrehaladtával [2]

1.1 A Braille-írás

Alapja a Charles Barbier által eredetileg Napóleon katonái számára kifejlesztett éjszakai írása, nevét továbbfejlesztőjéről és elterjesztőjéről, Louis Braille-ról kapta. Braille 1809-ben született Franciaországban, alig három éves korában egy balesetben megsérült a jobb szeme, majd a bal szemére átterjedő szimpatikus szemgyulladás következtében teljesen elvesztette látását. Ettől kezdve élénken foglalkoztatta a vakok olvasásának és írásának gondolata, 1821-ben, alig 12 évesen a párizsi Vakok Intézetében találkozott Charles Barbier írásával, s 1825-re azt továbbfejlesztve megalkotta az ún. Braille-írást *(2. ábra)* [4]. Bár már a XIX. század óta létezik és jelentőségét nemzetközileg elismerik, a Magyarországon élő vakok és súlyos látássérültek mindössze 10%-a ismeri [5].



2. ábra. Braille-írás olvasása két ujjal [6]

3. ábra. Braille-kijelző

1.2 A Braille-írás felépítése

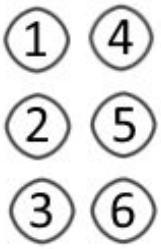
Az eredeti Braille-írás két fő szerkezeti elemből áll:

- Leképezi a francia nyelv betűit a 0 és 63 közti egész számok halmazára, vagyis az ASCII kódtáblához hasonlóan egyértelmű bináris kódot rendel hozzájuk
- Meghatározza, hogy a bináris kódoknak milyen formájú tapintható jel felel meg

Napjainkra a lehetőségek bővültek, a Braille-írás nem csupán francia, hanem az emberiség számos más nyelvén is használható, valamint kialakultak az angol, a német, vagy akár a magyar nyelv esetében is a rövidírás módszerei, melyek például egyes szavakat azok összes karaktere helyett néhány jellel reprezentálják. Megjelent a Braille-írás nyolc pontból álló verziója, s a számítógépek korában azokhoz már Braille kiviteli perifériák is csatlakoztathatóak *(3. ábra)*.

1.2.1 A Braille-írás fizikai megjelenése

A Braille-írás betűírásnak (alfabetikus írásnak) tekinthető, legegyszerűbb, rövidítések nélküli változata tulajdonképpen az adott nyelv betűkészletének leképezése a közismert, általában 6, néha 8 pontból álló karakterek halmazára. Az egyes Braille-karakterek (celláknak is nevezzük őket) hat pontját számozzák, így tehát például a 2-es számú pont balra középen található az adott cellán belül *(4. ábra)*.



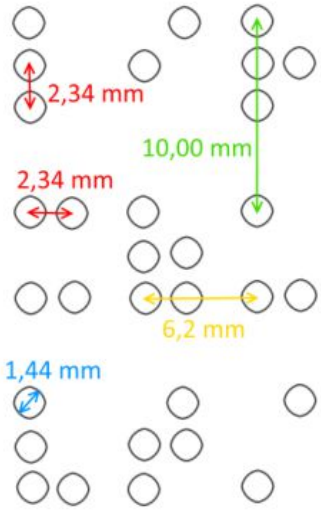
4. ábra. Braille-pontok

A különböző pontok számozása és a Braille-írás által az egyes betűkhöz rendelt bináris kód között közvetlen kapcsolat van, az egyes betűket jelölő k kód és a neki megfelelő b_i Braille-pontokból álló karakter között igaz, hogy:

$$k = \sum_{i=1}^{6} b_i \cdot 2^{i-1} \qquad b_i \in \{0,1\}, k \in \{0,1,\dots,63\} \tag{1}$$

Az egyenletben ha b_i értéke 1, akkor az adott i pozícióban található a papírlapból (vagy egyéb háttérből) kiemelkedő Braille-pont, ha pedig 0, akkor nem.

Természetesen vakok számára készült írások esetén nem célszerű tintát használni, éppen ezért a Braille-írást esetében a pontok színe megegyezik a környezetük színével (mind papírlapok, mind egyéb helyek esetében). Az egyes pontok tapinthatók, vagyis meghatározott mértékben kidomborodnak a háttér felületéből, vagyis taktilis módon érzékelhetők. A szakirodalomban találkozhatunk a haptika fogalmával, néha még a Braille-írás témakörében is, s bár mindkét esetben a tapintás egy formájáról beszélhetünk, utóbbi kategóriába például a különböző tárgyak felszínének keménysége tartozik, vagyis a tapintáshoz erőkifejtés is társul. A számítógépekhez mind taktilis, mind haptikus perifériák csatlakoztathatók, egy tipikus taktilis berendezés a Braille-szövegek kijelzésére alkalmas mátrixkijelző *(3. ábra)*, míg a haptikus berendezések egy jól ismert csoportja az ún. force feedback (erő-visszacsatolás) rendszerrel ellátott (angolul haptic technology néven is ismert) játékvezérlők, azaz kormányok és joystickok tartoznak, melyek például szimulátorok esetében szervó nélküli, álló helyzetben lévő autó kormányának tekerését érezhetően nehezebbé teszik.



5. ábra. Braille-pontok elhelyezése

Bár a Braille-szövegek felismerése komolyabb specifikáció nélkül is megoldható lenne, a könnyebb használhatóság miatt e területen is bevezettek bizonyos sztenderdeket [8]. Mi az Egyesült Államokban használt szabványokat fogjuk használni, természetesen ez a felismerő algoritmusunk számára nem más, mint egy paraméterhalmaz, vagyis szabadon változtatható a Braille-szövegek méreteire vonatkozó rendelkezések megváltozása, illetve egyéb

rendelkezések megalkotása esetén. A Braille-karakterek és pontok egymáshoz viszonyított távolságait, valamint a Braille-pontok méretét az *5. ábrán* láthatjuk.

A sztenderd emellett még tartalmaz a Braille-pontok magasságára és az adott sorban és oszlopban maximálisan elhelyezhető karakterek számára vonatkozó megkötéseket, ezek szintén nagy segítségünkre lehetnek.

A Braille-pontok problémaköréhez tartozik az optikai felismerők munkáját megnehezítő, ún. kétoldalas Braille-nyomtatás, ekkor a papírlapot olvasva láthatók és tapinthatók a túlsó oldal szövegéhez tartozó pontok, melyek ekkor nem kidomborodnak, hanem bemélyednek a papírlapba. További nehézséget jelent, hogy számos esetben a Braille-pontokat nem egyszínű háttérre nyomtatják a felület jobb kihasználásának érdekében, hiszen a Braille-pontok jelenléte a látó célközönségnek szánt szöveget vagy ábrát nem zavarja, s a tintával történő nyomtatás sem befolyásolja a Braille-szöveg olvashatóságát. Ebben az esetben a készítők az optikai és a taktilis érzékelés különbségéből indultak ki, mely természetesen optikai Braille-felismerők esetén nem létezik.

1.2.2 A Braille-kódolás

Ismerkedjünk meg a Braille-szövegek hétköznapi magyar szöveggé történő átkonvertálásával. Tekintve, hogy a karakterek rendelkeznek egyedi, a számítógépes rendszerekben használt kóddal (pl. ASCII, Unicode), tulajdonképpen egy adott szöveg különböző kódrendszerekből más kódrendszerekbe történő transzformálásáról beszélhetünk. A transzformáció első lépéseként ismerkedjünk meg a magyar teljes (rövidítéseket nem tartalmazó) Braille-írás mind a 64 karakterével (1. táblázat).

1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.
[a]	[b]	[c]	[d]	[e]	[f]	[g]	[h]	[i]	[j]	[␣]	[á]	[@]
[k]	[l]	[m]	[n]	[o]	[p]	[ö]	[r]	[s]	[t]	[.]	[í]	[Z]
[u]	[v]	[x]	[y]	[)]	[q]	[#]	[ü]	[(]	[ű]	[-]	[ú]	[D]
[é]	[z]	[C]	[G]	[S]	[N]	[ő]	[T]	[ó]	[w]	[V]	[A]	[L]
[,]	[;]	[:]	[H]	[?]	[!]	[=]	[”]	[*]	[Q]		[R]	[F]

1. táblázat. *A 64 féle magyar Braille-karakter [9]*

A transzformáció első lépéseként a felismert Braille-karaktereket konvertáljuk a tőlük jobbra, a szögletes zárójelben látható karakterekké. Ez egy bijektív leképezés, egy egyszerű szubsztitúció, célja, hogy a látó olvasó számára könnyebben olvashatóvá tegye a Braille-szöveget. A Braille-karakterek közt nincs nagybetű, a 1. táblázat nagybetűi mind különleges jelentéssel bírnak. Közülük számos (pl. G, S) az adott betűvel kezdődő digráf, azaz kettős betű jelölése egyetlen karakterrel (vagyis ebben az esetben gy és sz), míg mások a következő karakter tulajdonságát változtatják meg, néhány példa:

- Az A-t követő karakter nagybetű
- A V-t (védőkarakter) követő karakter biztosan nem rövidítés (lásd lentebb)
- A D-t követő karakter szám (az a-j karaktereket használjuk a számok jelölésére)

Így tehát egyetemünk névadójának nevét az alábbi módon írjuk Braille-szövegként: "Széchenyi István" = "ASécheNi Aistván". Fontos, hogy nekünk csak Braille-szöveget kell normál szöveggé konvertálnunk, vagyis elég a konverzió egyik irányával foglalkoznunk, s ez nagy könnyítést jelent.

Érdemes kitérni a más nyelvekhez hasonlóan a magyarban is létező rövidítésekre, ezen belül is az ún. kis rövidírásra. Az alábbi szabályai vannak [9]:

- Töröljük a nagybetűjeleket.
- Ne kövesse a vesszőket szóköz.

- Helyettesítsük a határozott névelőket és az őket követő szóközt ponttal (az) és vesszővel (a).
- Alkalmazzuk a 2. táblázatban látható 44 rövidítési szabályt.

7 szóvégi rövidítés		16 egyjelű szórövidítés				21 kétjelű szórövidítés					
-ban/-ben	b	Cak	C	meL	m	aNNi	ai	mind	md	rövid	rd
-ból/-ből	b.	de	d	nem	n	boldog	bg	mint	mt	forr	rr
-hoz/-hez/-höz	h.	és	é	óta	ó	eNNi	ei	orSág	og	Sabad	Sd
ként	k.	hoG	h	pedig	p	gond	gd	olvas	os	tanáC	tC
-ról/-ről	r.	is	i	tehát	t	függ	gg	öSSe	öe	teljes	ts
-tól/-től	t.	íG	í	után	u	Gors	Gs	pont	pt	világ	vg
-val/-vel	v	kell	k	úG	ú	keres	ks	pénz	pz	volt	vt
		leS	l	van	v						

2. táblázat. *A kis rövidírás 44 rövidítési szabálya [9]*

Amennyiben szeretnénk, hogy a Braille-szöveget normál szöveggé átalakító programunk a kis rövidírást is támogassa, meg kell változtatnunk a szemléletünket. A 16 egyjelű és a 21 kétjelű szórövidítés nem okoz jelentős problémát, a 7 szóvégi rövidítés helyes értelmezéséhez azonban programunknak ismernie kell a magyar nyelvben használt szavakat és toldalékolásukat:

- A "Sobáb" Braille-szöveg a "szobában" karakterláncra fordítódjon le, ne pedig "szobában"-re, vagy "szobáb"-ra.
- Az "ASob" Braille-szövegből "Szob" község nevét kapjuk, ne pedig a "Szoban", vagy "Szoben", értelmetlennek tekinthető szavakat.

2 A fejlesztendő szoftver

Mint az a feladat kiírásakor és az összefoglalóban is ismertetésre került, a cél Android operációs rendszert futtató mobiltelefonokra olyan applikáció megírása, mely a Braille-szövegeket a készülék kamerája segítségével lefényképezi, a fényképet analizálva felismeri az azon található karaktereket, a Braille-kódolásból lefordítja hétköznapi magyar szöveggé és végül felolvassa azt a látássérült felhasználó számára.

2.1 A futtatókörnyezet, a hardver és a nyelv

Mielőtt hozzákezdenénk a programunk elkészítéséhez, érdemes megvizsgálnunk a platformot, melyre fejleszteni kívánunk. A mi esetünkben ez az Android, s az azt futtató okostelefonok. Fontos megismerkednünk a telefonok számítási teljesítményével, az operációs rendszer által nyújtott szolgáltatásokkal és a szoftverekre vonatkozó megkötéseivel is.

Hardver architektúra és OS: Napjaink okostelefonjai általában többmagos, 1-2 GHz közötti frekvencián üzemelő, ARM processzorral rendelkeznek, melyhez több GB memória társul. Egy jól megírt optikai karakterfelismerő program futtatása nem okozhat számukra gondot. Természetesen ehhez szükséges egy kellően nagy felbontású kamera is, ez azonban a legtöbb telefonban megtalálható. Megállapítottuk tehát, hogy a feladat elvégzése elméletileg lehetséges. Fontos azonban figyelembe venni, hogy a helyzet mégsem olyan egyszerű: mind az operációs rendszer, mint maga a tény, hogy akkumulátorról üzemelő mobil eszközökről beszélünk korlátozásokat vezet be, melyeket figyelembe kell vennünk. Különösen kisebb teljesítményű eszközök esetén lehet problémás az egy alkalmazás számára maximálisan elérhető memória mennyisége, mely némely esetben mindössze 16-32 MB. Ennek következtében takarékoskodnunk kell az objektum-orientált nyelvek egyik lényegi elemének, az objektumok létrehozásának, valamint az általuk használt memóriaterület felszabadításának dinamikus módszereitől (vagyis a new operátor, valamint a garbage collector számos esetben rendkívüli lassulást eredményez), tartózkodnunk kell az absztrakciók túlzott használatától, mely a programok fejlesztését és karbantartását nagyban megkönnyítik és elméletben jó fordító esetén nem eredményeznek jelentős lassulást, a gyakorlat mást mutat, s az Android fejlesztői sem ajánlják e fejlesztési stílust [10].

Reakcióidő: Bár a közel valós idejű adatfeldolgozás nem szükséges, ezt a létező karakterfelismerő applikációk többsége sem tudja, mégis jogos elvárás a szoftverünkkel kapcsolatban a gyors működés (maximum néhány másodperc feldolgozási idő), természetesen minél kevesebb, annál jobb. A program fejlesztése során tehát mérlegelnünk kell az átláthatóság és könnyű fejlesztés, s ebből következően a hibák számának csökkenése, valamint a gyors futtatás lehetősége között. Erre az egyik leginkább bevált megoldás a magas szintű, absztrakciókat alkalmazó program kifejlesztése, majd annak tesztelése és futtatása során a legkritikusabb pontokban történő célzott optimalizálás végrehajtása. Így könnyen fejleszthető, tesztelhető, módosítható, ám mégis gyors applikációt kapunk.

Fejlesztőkörnyezet: Számos fejlesztőkörnyezet és nyelv használata közül választhatunk, az Android platform legnépszerűbb programozási nyelve továbbra is a Java, azonban nagy számítási igényű applikációk, például játékok, videokonvertáló programok esetén nagy létjogosultsága van az általában gyorsabban futó C++ programoknak is [11]. Ez utóbbiak további vitathatatlan pozitív tulajdonsága, hogy a memóriaallokációk és azok felszabadítása a fejlesztő kezében van, s bár ez a fejlesztést megnehezíti, összességében jól elkészített kód esetén gyorsabb és determinisztikusabb futást eredményez, hiszen a garbage collector és a JIT fordító nem eredményez minimális akadozásokat sem a valós idejű programok futása során. Összességében érdemes az alábbi szoftverfejlesztő környezetekre odafigyelni:

- Android SDK: az Androidra történő fejlesztéshez feltétlenül szükséges, Java alapú
- JDK: általános Java programok futtatására szolgál, számos fejlesztőeszköz támogatja
- Android NDK: nagy számítási igényű szoftverrészek és modulok fejlesztésére szolgál Androidra (C/C++)
- gcc/g++: szinte bármely számítógépes rendszeren használható, natív C/C++ kód

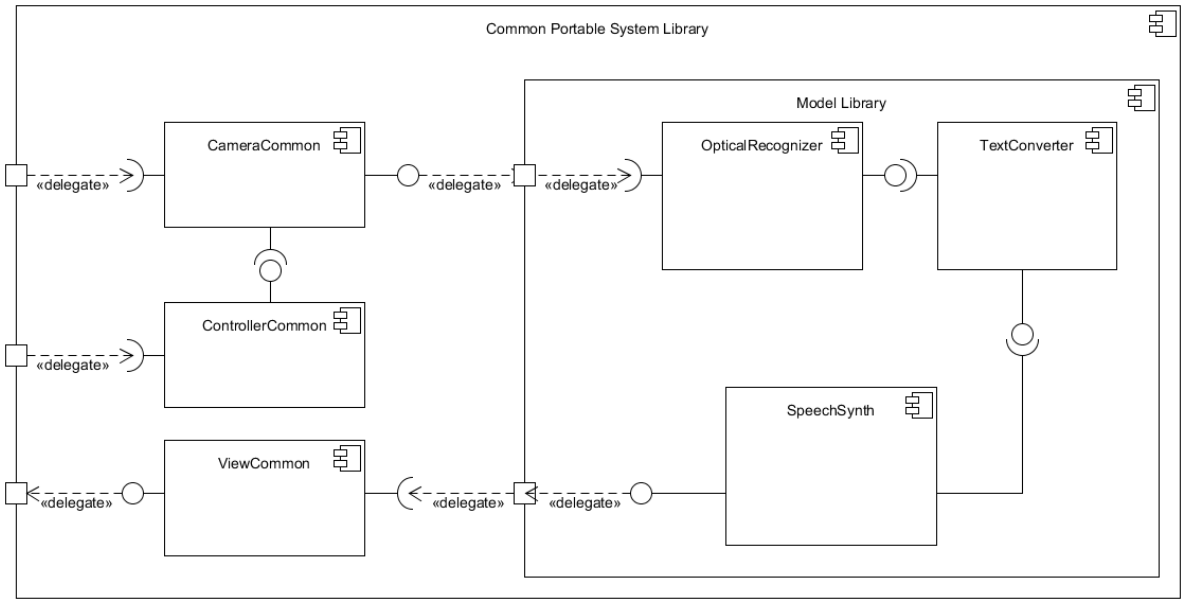
Bármelyik környezetet is válasszuk, célszerű programunk nagy részét (fő algoritmusok) könyvtárként kifejleszteni, s az aktuális API (pl. Android) lehetőségeit (például kameravezérlés) ettől független adapter modulokkal kihasználni. Rendkívül előnyös tehát, ha a fő könyvtárunk Java elemei az Android SDK mellett JDK-n is futtathatók és tesztelhetők, C/C++ elemei pedig az Android NDK mellett például az Androidhoz hasonló UNIX operációs rendszerek gcc/g++ fordítóival is kompatibilisek.

2.2 A szoftver egészének felépítése

Összetettebb, több feladatot is ellátó szoftverek fejlesztése során a tervezés fázisa elengedhetetlen, nem csupán a napjaink startup generációja által számos hátránya miatt kevéssé népszerű vízesés modell szerint, de az agilis szoftverfejlesztés (XP, Scrum) esetében is [12]. Az architektúra előzetes részletgazdag megtervezésére jelen esetben nincs szükség, átgondolására azonban igen.

Az alkalmazás struktúrája: Tekintve, hogy a felhasználóval kommunikáló appról beszélünk (ellentétben például egy adatbáziskezelővel), magától értetődik az MVC (magyarul: modell-nézet-vezérlő) felépítés [12], az előző fejezetben említett központi rész, mely a fő algoritmusokat tartalmazza itt a modellnek felel meg. A másik két, viszonylag egyszerű felépítésű modul fogja megteremteni a kapcsolatot az Android API, a felhasználó és a fő könyvtár között.

A fő könyvtár (modell) struktúrája: a legfontosabb e modul optimális struktúrájának kialakítása, ettől nagyban függ a programfejlesztés nehézsége és a kapott kód minősége. A mi céljainkra (a szerző szubjektív véleménye szerint) közel tökéletes választás az ún. csövek és szűrők (pipes-and-filters) architekturális minta [12]. Ezt hatékonyan felhasználva további jelentős tervezésre nincs szükség, az ezt alkotó elemek (csövek, szűrők) mérete szoftvermérnöki szempontból szinte elhanyagolható. Hatalmas előnye, hogy kiválóan párhuzamosítható, jelfeldolgozással foglalkozó szoftverekben igen gyakori.



6. ábra. A szoftver fő moduljai és kapcsolataik

A *6. ábrán* láthatjuk a szoftver felépítését, melyek közül a továbbiakban a szoftver lényegét jelentő, az ábrán "model library" néven szereplő fő könyvtár és annak három része kerül ismertetésre. Már ezen a legfelsőbb szintű komponenseket feltüntető UML diagramon is látszik a felhasználóval történő interakciót megvalósító MVC struktúra, s az alatta megbúvó pipes-and-filter architektúra, mely a hierarchia alacsonyabb szintjein is meghatározó lesz. Az ábrán a nézet és vezérlő modulok nevében szereplő közös szócska a modulok portábilis, rendszerfüggetlen részére utal, ezt természetesen minimális méretű, az adott rendszerre (API, keretrendszer) adaptáló elemekkel ki kell egészítenünk.

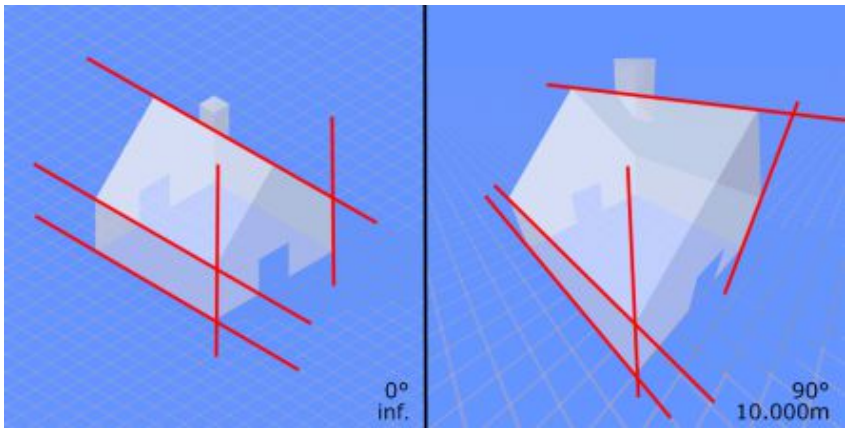
Felhasználhatóság: Természetesen az elméleti problémák megoldása és a stabil üzem megvalósítása után (több fejlesztő esetén ez időben egyszerre is történhetne) az általánosnál jóval több figyelmet kell fordítanunk a szoftver kezelhetőségére, vagyis a felhasználó felület ki- és bemenete a látássérült és vak emberek számára is használható kell, hogy maradjon. Ez az érintőképernyős telefonok korszakában különösen nagy kihívásnak számít, hiszen ez a be- és kimeneti interfész a nyomógombos telefonokkal ellentétben nem tapintható és sajnos az információk kizárólag a kijelzőn történő megjelenítése sem jöhet szóba.

2.3 A szoftver képességének határai

A legtöbb szoftver esetében annak működési területét a fejlesztés elején kidolgozott ún. funkcionális specifikáció határozza meg. Ennek részletessége és a határok élessége természetesen függ az alkalmazott szoftverfejlesztési módszertantól, a lineáris (pl. vízesés) modellek esetén már a legelső fázisban pontosan meg kell alkotni a teljes leírást, az iteratív (pl. RUP) modellek felhasználásakor a részletgazdagság kevésbé lényeges, azonban még az agilis (pl. XP) eljárások esetén is szükséges, legalább koncepció szintjén [12]. A probléma ismeretének vizsgálata tehát nem elhanyagolható. A következőkben, még a szoftver elkészítése és tesztelése előtt, a teljesség igénye nélkül ismerkedjünk meg néhány problémával, mely előre láthatóan felmerülhet a majdani használat során.

Az optikai karakterfelismerő működési határai

- Nincs komoly megkötés a Braille-pontok méretére, amíg azok felismerhetők, méretük néhány 10 pixeltől több ezer pixelig terjedhet.
- Nincs megkötés a kép vízszintes, függőleges, esetleg ferde orientáltságára.
- A kép nem lehet túl homályos, elmosódott.
- A kontraszt és fényességi problémák javításra kerülnek.
    - A szoftver átlagos felhasználója nehézségekkel szembesül a Braille-szöveg vizuális felismerhetőségét illetően (nem tud különbséget tenni világos és sötét helyiségek között), ebben az applikációnak akár a beépített vaku automatikus használatával, akár hallható üzenetben segítenie kell őt
- A perspektivikus torzítás (lásd: *7. ábra*) kisebb mértékben még nem, nagyobb mértékben már megzavarja az algoritmust, mely feltételezi, hogy a valóságban párhuzamos egyenesek a képen is többé-kevésbé azok (ez a fényképekre általánosságban nem igaz, a jelenség azonban néhány kivételtől eltekintve elhanyagolható). A jelenség általában lapos szögben készített képek, illetve nagylátószögű objektívek használata esetén válik nehezen kezelhetővé, előbbi a telefon helyzetének optimálisabb megválasztásával korrigálható, utóbbi probléma pedig elhanyagolható mértékben jelentkezik napjaink okostelefonjai esetén.



7. ábra. Perspektivikus torzítás [13]

A Braille-kódolás konverter működési határai

- A teljesírás támogatása komolyabb megkötések nélkül.
- A kis rövidírás támogatása best-effort alapon elképzelhető, tökéletes eredményt nem lehet tőle várni, a támogatás megvalósításán éppen ezért érdemes elgondolkodni. Beépített szótár segítségével a toldalékolás felismerésére a szoftver kísérletet tehet, ez azonban az agglutináló (így a magyar) nyelvek esetén számos nehézséget okozhat. Figyelembe véve az aktuálisan az interneten elérhető automatikus kereső- és fordítóprogramok nehézségeit a magyar toldalékolást illetően, valamint összehasonlítva az őket fejlesztő szoftvercégek anyagi lehetőségeit és személyi állományát e program készítésének körülményeivel, ez több mint elegendőnek nevezhető.

A szövegfelolvasó beszédszintetizátor működési határai

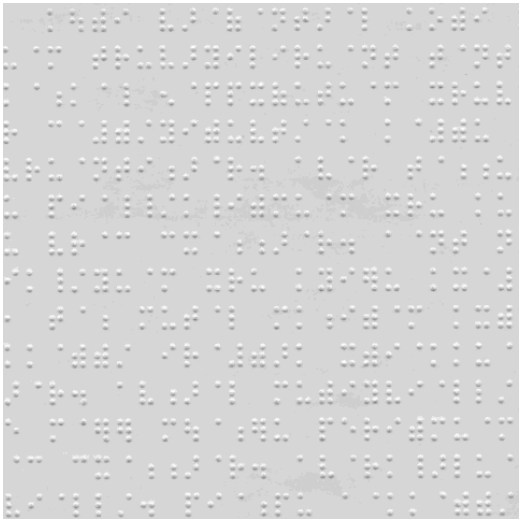
- A magyar szövegek felolvasásának támogatása best-effort alapon
- A szövegben előforduló idegen szavak, helységnevek (pl. Bordeaux) kívül esnek a felolvasó által ismert kiejtési szabályok halmazán.
- A szövegben előforduló számok felolvasása támogatott.
- Az API által biztosított beszédszintetizátorok (pl. Android TextToSpeech) opcionálisan felhasználhatók.

3 Az optikai karakterfelismerő modul

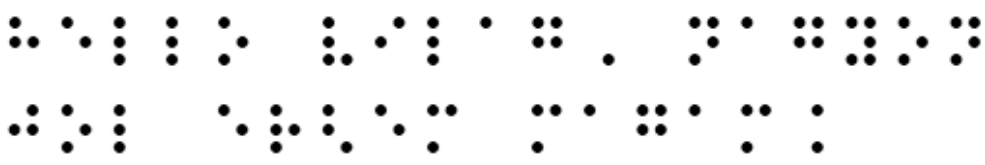
3.1 Az alapok, áttekintés

Miután megismerkedtük a szoftver fő elemeivel, lássuk az első komolyabb elméleti háttérismereteket igénylő részproblémát: a kamera képén látható Braille-szöveget (vagyis bármely RGB, vagy akár szürkeárnyalatos képet) fel szeretnénk ismerni és át szeretnénk konvertálni Braille-kódolású szöveggé.

Egyelőre tekintsünk el a bonyolultabb esetektől, tegyük fel, hogy egyszerűen azonosítható pontokkal van dolgunk. Válasszunk ki egy Braille-írást tartalmazó képet (*8. ábra*) és a következő oldalakon keresztül kövessük végig a karakterfelismerő működését a példán keresztül. Vessük össze a 8. és a 9. ábrát, utóbbin egy szövegszerkesztővel előállított, számítógépes megjelenítéshez, illetve egyszerű nyomtatáshoz felhasználható Braille-írást látunk látó olvasók számára. Érdemes szoftverünket úgy megírni, hogy ezzel a szövegformátummal is boldoguljon.



8. ábra. A teszt Braille-szöveg [22]



9. ábra. Braille-szöveg számítógépes megjelenítéshez

3.2 Rasztergrafikák és az emberi szem

Napjainkban szinte mindenütt előfordulnak digitálisan rögzített álló- és mozgóképek. A kamerákban található CMOS vagy CCD érzékelőkben milliónyi, egy-egy képpont rögzítésére alkalmas szenzor előre meghatározott elrendezésben kerül kialakításra. A leggyakoribb, ún. Bayer-elrendezés mellett léteznek ennél hatékonyabban felépített érzékelők is, a képek feldolgozása és tárolása azonban szinte minden esetben négyzetrácsos raszter alapján történik. Az egyszerűség kedvéért kezdjük a szürkeárnyalatos képekkel. A képek felbontását pixelben (képpontban) adjuk meg, egy 640x480 pixel felbontású kép 640 pixel széles és 480 pixel magas. Egy ilyen felbontású kép kellően távolról megtekintve folytonosnak tűnik, egy adott részletét kinagyítva azonban remekül láthatóvá válnak az egyes pixelek (lásd *10. ábra*).

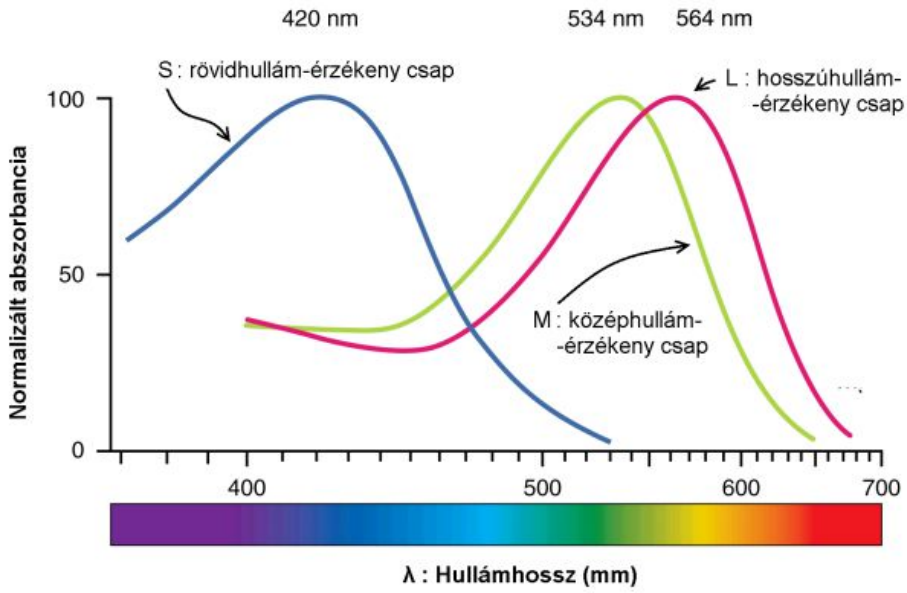
9. ábra. Raszterek kinagyítva [19]

A pixelek intenzitását az általunk használt (nem HDR) képek nemnegatív egész számként tárolják, maximális értékük gyakran kettő valamely hatványánál 1-gyel kevesebb, általában 255, de ez nem törvényszerűség. Egy szürkeárnyalatos raszterkép nem más tehát, mint egy funkció, amely két nemnegatív egész értékhez (a képpont vízszintes és függőleges pozíciója, a (0, 0) általában a bal alsó sarkot jelöli, a kép ekkor tehát a Descartes-féle koordináta-rendszer I. síknegyedében helyezkedik el) egy harmadik nemnegatív egész értéket (a képpont intenzitása, a 0 általában fekete, a 255 pedig fehér, a köztes értékek szürkék) rendel [13]:

$$Img : \{0, \dots, M-1\} \times \{0, \dots, N-1\} \rightarrow \{0, \dots, I-1\} \quad M, N, I \in \mathbb{N}^{+} \qquad (2)$$

Bár a továbbiakban szürkeárnyalatos képekkel fogunk foglalkozni, nem tekinthetünk el a színesektől sem. A kamerák érzékelőinek felépítése nagyban hasonlít az emberi szemben

található fotoreceptorok millióihoz, melyekből trikromát fajok, így az ember esetén is három féle található, melyek más-más érzékenységűek a különböző frekvenciatartományokban [20]. Az érzékenységet a csapokban található protein, ún. fotopszin abszorbciós görbéje határozza meg, mely például spektrofotométerrel is megmérhető (*11. ábra*) [14].



10. ábra. *Fotopszin abszorbancia [20, 21]*

Legyen az S a kék színre érzékeny fotopszin abszorbciós funkciója a hullámhossz függvényében (*11. ábra*), P pedig a beérkezett fény intenzitása adott frekvencián, ekkor a rövidhullám-érzékeny csapok válasza az alábbi:

$$s = \int S(\lambda)P(\lambda)\, d\lambda \qquad [16]\ (3)$$

Analóg módon számíthatjuk az másik két csaptípus m és l válaszait. Könnyen belátható, hogy a trikromát (háromféle csappal rendelkező) élőlények számára a szín háromdimenziós mennyiség, melynek reprezentálására több rendszert is kifejlesztettek. A legismertebb az RGB, mely három különböző spektrális (egy frekvenciából álló) szín kombinációjával kelti a szemlélő emberben egy esetleg teljesen más spektrális összetételű szín illúzióját. Az RGB kép tehát az alábbi funkció, ahol M, N, R, G és B pozitív egész számok:

$$Img' : \{0, \dots, M-1\} \times \{0, \dots, N-1\} \rightarrow \{0, \dots, R-1\} \times \{0, \dots, G-1\} \times \{0, \dots, B-1\} \qquad [?]\ (4)$$

További színformátumok például a HSV és a YCbCr. Ez utóbbiak nagy előnye, hogy az RGB-vel ellentétben a fényerőt és a színt más-más koordináta-tengely mentén tárolják, így szürkeárnyalatos kép esetükben két dimenzió elhagyásával előállítható [15, 17]. A HSV különösen a színek alapján (pl. emberi bőr) történő szegmentálásakor van segítségünkre [17].

11. ábra. Háromdimenziós színtér: RGB és YCbCr színmodell

A *12. ábrán* ugyanazt a színes képet láthatjuk RGB képként és YCbCr képként. Előbbi általában az állóképek esetén használt formátum, fontos azonban megjegyezni, hogy az állóképeket számos kamera és API csak .jpeg tömörített formátumban tudja biztosítani. Vagyis a szoftverünk kérheti az operációs rendszert, hogy egy ideiglenes mappába mentse el a tömörített képet. Ekkor a képet az operációs rendszer tömöríti, kiírja a háttértárra, mi visszatöltjük, kitömörítjük és RGB színtérből szürkeárnyalatosba konvertáljuk, mely eljárás leggyakrabban az alábbi módon valósul meg (az összegzést minden pixelre elvégezve):

$$I = 0{,}299R + 0{,}587G + 0{,}114B \qquad [15]\ (5)$$

Itt I-vel jelöltük a szürkeárnyalatos kép pixeleinek intenzitását, R, G és B betűkkel pedig a három komponens intenzitását az RGB kép esetén. Érezhető, hogy a konverzió számos lépése negatív hatással lesz a sebességre. Célszerű tehát megvizsgálni az alternatívát: az Android API képes a kamera élőképét szolgáltatni az alkalmazásnak, ezt láthatjuk például amikor fénykép készítése előtt a képernyőn nézzük, hogy mit lát a kamera. A videó formátum nagy előnye, hogy tömörítetlen, az adott képkockája bitmap raszterképhez nagyon hasonló formátumban kerül rögzítésre így nincs szükség tömörítésre és kitömörítésre, valamint YCbCr színteret használ, vagyis a szürkeárnyalatos képet megkapjuk, ha a Cb és Cr komponenseket figyelmen kívül hagyjuk [15]. Ez drámai sebességnövekedést eredményez.

3.3 Normalizálás

Az előző feldolgozási lépés eredményeként kaptunk egy szürkeárnyalatos képet. Ezen a képen kell megkeresnünk a Braille karaktereket. A megoldandó probléma a következő: számos kép készül kedvezőtlen fényviszonyok mellett, helyenként árnyékokat tartalmaz, esetleg túlságosan kontraszttalan, szürke. Célunk, hogy megkönnyítsük a következő modul dolgát és optikailag könnyen felismerhetővé tegyük a Braille pontokat, azoknak tehát jól el kell különülniük a környezetüktől. Kiindulási szürkeárnyalatos képünkön (*8. ábra*) eddig még semmilyen változtatást nem hajtottunk végre. Az első lépésként érdemes meghatározni a képünk hisztogramját. A fejezetben a hisztogrammal kapcsolatos tudnivalók rövid összefoglalása, bővebben az olvasó a fejezet alapjául szolgáló kiadványban találhat róla információkat [18]. A hisztogram egy olyan funkció, mely minden lehetséges intenzitási értékhez (a pixelek fényessége) hozzárendeli a kép ezen értéket felvevő pixeleinek számát. Az előző oldal képleteit folytatva (H a hisztogram funkció):

$$H(i): \{ 0, \ldots, L - 1 \} \rightarrow \{ 0, \ldots, (M \times N) - 1 \} \quad (6)$$

$$H(i) = \#\{ (m,n) \mid (0 \leq m < M) \wedge (0 \leq n < m) \wedge (Img(m,n) = i) \} \quad (7)$$

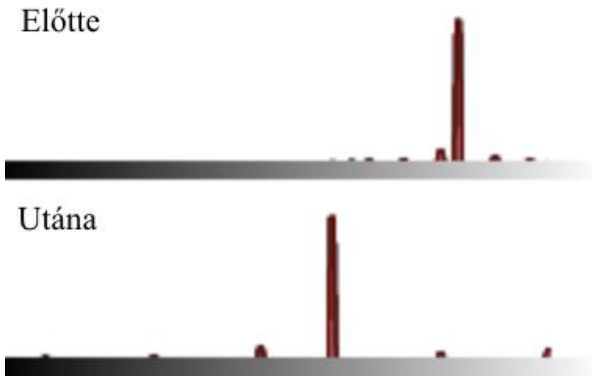
Az említett jó elkülönülést definiálhatjuk nagy távolságként is, célunk tehát, hogy a Braille pontok pixeleinek intenzitása jelentősen térjen el a környezetükétől. A hisztogram az első lépés ennek irányába. Mint az a *13. ábrán* is látszik, a hisztogram nyújtás előtt szinte az összes pixel a tartomány világosabb felében található. Ez erre a képre jellemző, általánosságban azonban a különböző képek hisztogramjai igen nagy szórást mutatnak. Célunk az uniformizált kimenet biztosítása, erre remek lehetőséget ad a hisztogram nyújtása. Általánosan az eljárás így írható le:

$$I'(m,n) = (I - 1) \cdot \frac{I(m,n) - I_{min}}{I_{max} - I_{min}} \quad (8)$$

$$I_{min} = H(i) \mid \nexists j \in \mathbb{N} : (0 \leq j < I) \wedge \big(H(j) < H(i)\big) \quad (9)$$

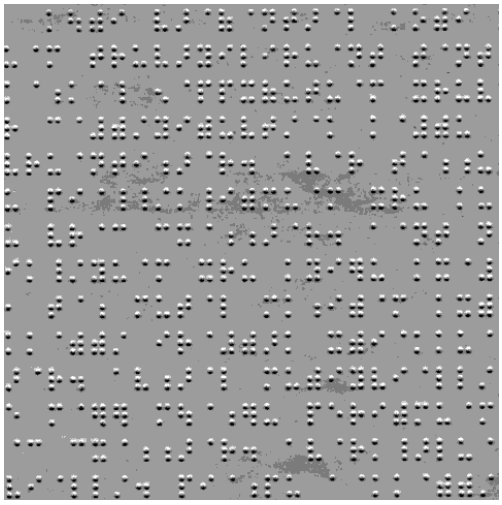
$$I_{max} = H(i) \mid \nexists j \in \mathbb{N} : (0 \leq j < I) \wedge \big(H(j) > H(i)\big) \quad (10)$$

Könnyen belátható, hogy ha csak egyetlen pixel is felveszi az elméleti minimum értéket (0-t) és egy másik pedig az elméleti maximum értéket (általában 255), a funkció az eredeti értéket adja vissza. Robusztusabbá tehető azonban, ha nem a legkisebb és legnagyobb értéket, hanem helyettük például a 0,01 kvantilis és 0,99 kvantilis értékeit használjuk. Ezt a *7. ábra* szürkeárnyalatossá konvertált változatán elvégezve az eredményt a *13. ábrán* láthatjuk:



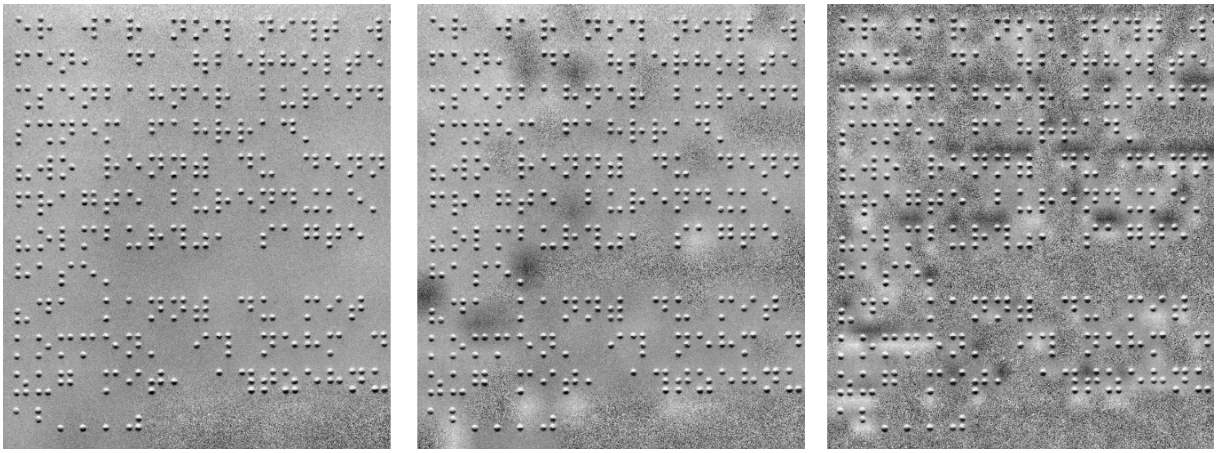
12. ábra. Hisztogram nyújtása

Sajnos erre a trükkre a Braille minta nagyon érzékeny, szabad szemmel is látható, hogy a legtöbb esetben a pontok fényes és árnyékos felülete a kép legvilágosabb és legsötétebb része. A Braille írás specifikációját visszaolvasva kiszámíthatjuk, hogy egy karakter és az őt körülvevő üres terület mérete átlagban 6,2mm · 10mm = 62mm^2, míg egy pont területe $((1{,}44 \,/\, 2)mm\,)^2 \cdot \pi \;=\; 1{,}63mm^2$. Karakterenként 3 ponttal számolva, feltételezve, hogy egy pont területének kb. 20%-a világos és 20%-a sötét, az értékes információ megtartásához a 0,015 és 0,985 kvantilisek által meghatározottnál nem választhatunk szűkebb intervallumot. Számolva a szükséges toleranciával a 0,01 és 0,99 optimális választásnak tűnik (*14. ábra*).



13. ábra. Kép a hisztogram nyújtása után

Alternatív megoldásként szóba jöhet a lokális hisztogram ekvalizáció [17], ebben az esetben a hisztogram nyújtásának műveletét nem globálisan, hanem az adott pixelek környezetét figyelembe hajtjuk végre. Előnye, hogy a globális minimum és maximum szinte egyáltalán nem befolyásolja, hátránya viszont, hogy hajlamos a régiókban található, egyébként jelentéktelen zajt zavaró és a további szűrőket esetlegesen megtévesztő mértékűre felerősíteni. Helyes konfigurálása létfontosságú, túl kevés mező használata esetén a nagyobb sötét/világos foltok (például árnyékos részek) hatása nem kerül kompenzálásra, túl sok mező használata esetén viszont érzékeny az egyes mezők hisztogramjainak természetes különbségére és ezt a különbséget durván felerősíti. Ez utóbbi jelenségre láthatunk példát a 15. ábrán, hatására számos nem kívánatos sötétebb/világosabb folt alakul ki a képen.



14. ábra. Adaptív hisztogram ekvalizáció 64, 256 és 1024 mérési mezővel

Természetesen egyéb lehetőségeink is vannak, hely hiányában azonban nincs lehetőség mindet bemutatni. Példa:

$$I'(m,n) = min(max(\ \alpha \cdot I(m,n) + \beta\ ), 0), I - 1) \qquad (11)$$

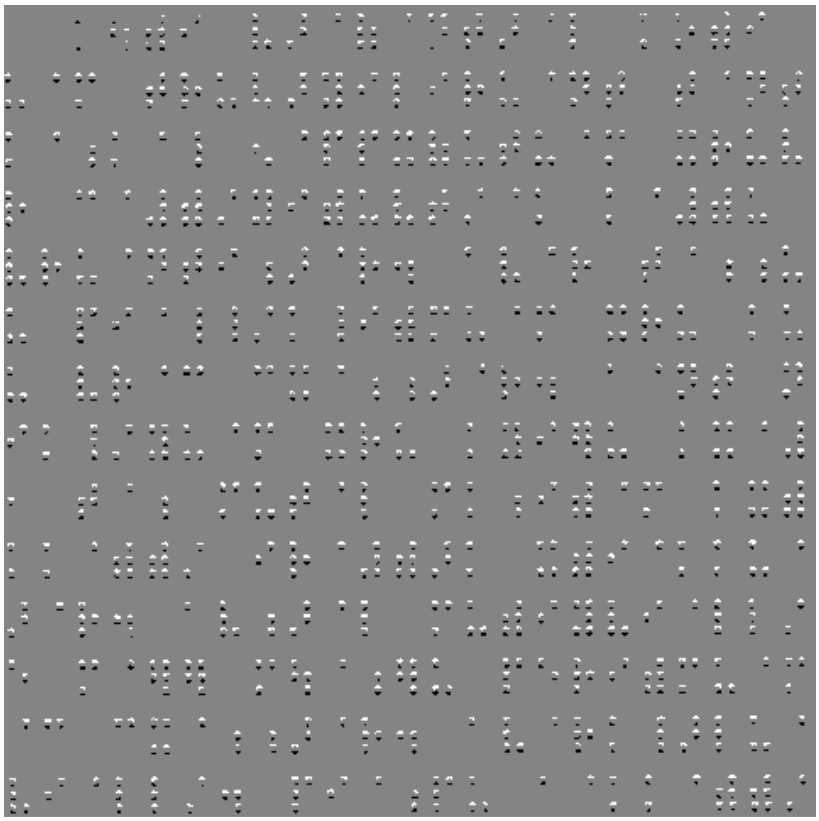
Az eredeti értéket α = 1 és β = 0 esetén kapjuk vissza (identitás függvény), β változtatásával a kép fényerejét, α változtatásával a kontrasztját szabályozhatjuk.

3.4 A pixelek osztályozása

Miután képünket normalizáltuk, kiindulhatunk abból, hogy az oldalirányból érkező fénysugarak a különböző beesési szögek miatt a Braille pontok egyik oldalát kiválóan megvilágítják, míg másik oldaluk árnyékossá válik. Ennek eredményeképp egy Braille pont a szürke háttértől könnyen megkülönböztethető: egy világos és egy sötét terület alkotja. A feladatunkat egyszerűsítendő, tegyük fel, hogy kollimált (párhuzamos fénysugarakból álló) megvilágítást kap a Braille szöveg. Ekkor minden pont ugyanazon oldala lesz világos, és ellentétes oldala sötét. Ebben a feldolgozási lépésben minden pixel esetében meg kell hoznunk a döntést: megvilágított oldal, árnyékos oldal vagy egyik sem. Ennek egyik legegyszerűbb módja az alábbi (t_1 és t_2 threshold értékek előre megválasztandók):

$$I'(m,n) = \begin{cases} 0, ha\ I(m,n) < t_1 \\ 128, ha\ t_1 \leq I(m,n) < t_2 \\ 255, ha\ t_2 \leq I(m,n) \end{cases}$$

Az egyetlen kihívást t_1 és t_2 optimális megválasztása jelenti. Ehhez segítséget nyújthat a hisztogram és annak ismerete, hogy egy egyszínű háttérrel rendelkező Braille szöveg esetén nagyjából a terület 1%-a lesz sötét és 1%-a világos (lásd: előző oldal). Számos esetben azonban kép egyik fele árnyékos, a másik fele fényesebb. Természetesen a fényesebb felén az árnyék is fényesebb. A probléma akkor válik súlyossá, ha például a fényesebb felén az árnyék fényesebb, mint az árnyékosabb felén a háttér, vagy a világos részek. Ekkor globális határértékek nem határozhatók meg. Érdemes lehet ilyenkor kisebb méretű lokális hisztogramokat kiszámolni és ezek vonzáskörzetében megfelelően állítani a t_1 és t_2 értékeket. Természetesen a jobb eredmények elérése érdekében a szomszédos hisztogramok átfedési területein interpolálhatjuk azok értékét például a középpontjuktól való távolság függvényében. Van azonban más lehetőségünk is. Ha ragaszkodunk a pipes and filter struktúrához (és célszerű hozzá ragaszkodni, megkönnyíti a munkát, hiszen a szűrők algoritmusai egyszerűek és egyenként könnyen tesztelhetőek és módosíthatóak maradnak), úgy ebben a lépésben ezzel már nem kell foglalkoznunk. Ilyen problémák fellépése esetén az előző szűrő globális hisztogram-nyújtását kell kibővíteni adaptív (a pixel környezetétől függő) kontraszt és fényerő normalizáló funkcióval, vagy egyszerűen behelyezni a két szűrő közé egy harmadikat, amely elvégzi az extra feladatot.



17. ábra. Három osztályba sorolva (fehér/szürke/fekete)

Érdemes röviden szót ejteni az alternatív lehetőségekről. Ehhez hasonló feladatokra tipikus megoldásként szóba jöhetnek az éldetektorok. Ezek olyan funkciók, melyek kimeneti értékére nem csak egy pixel van hatással, hanem annak környezete is. Kimenetük néha bináris, szürkeárnyalatos képekre alkalmazva őket azonban általában szürkeárnyalatos képet eredményeznek. A kapott kép világossága/sötétsége egy adott pixel esetén az eredetileg ott található él és környezetének kontrasztjával van összhangban, vagyis megkülönböztetik az éles körvonalú tárgyak belsejét az éleiktől. Alkalmazásuk gyakori az autonóm robotika területén, többségük (Prewitt, Sobel...) konvolúció és egy ún. kernel mátrix segítségével valósítható meg, ám akadnak más megoldások is (Canny). A mi problémánk megoldásában ugyan segíthetnek, azonban eredményük közel sem optimális, melynek oka az alábbi: a Braille pontoknak nincsenek klasszikus értelemben vett élei, hiszen félgömb alakúak. Ennek következtében a rajtuk vetett árnyéknak sem lesz határozott körvonala és tekintve, hogy általában a háttértől a színük sem különbözik, nagyon nehéz megkülönböztetni az éldetektor jelzését az egyéb hamis pozitív jelzésektől. Erre a feladatra tehát a fent említett thresholding (ill. gray level slicing) eljárás praktikusabbnak tűnik.

3.5 Pixelklaszterek

Adott egy három színűre redukált kép, melyben a világos és sötét pontok halmazát szeretnénk klaszterekre bontani. Természetesen lehetőségünk van igénybe venni a klaszterező algoritmusok szolgáltatásait, erre azonban egyelőre nincs szükség. A fő probléma velük kapcsolatban, hogy legtöbbjük a sikeres futáshoz különböző paramétereket igényel. Két példa: K-Means esetén meg kell tippelnünk a klaszterek számát, DBSCAN esetén pedig a pontok távolságát. Ezt a feldolgozás ezen fázisában nehéz hatékonyan elvégezni, túl kevés hozzá az információnk, hiszen ebben a helyzetben mindkét említett adatot épp a klaszterező algoritmustól várnánk. Használjuk ki azt az előnyt, hogy minden pont (pixel) koordinátája egész szám, vagyis pontjaink egy négyzetrácsos raszterben helyezkednek el. Bár ha ponthalmazok klaszterekbe rendezéséről van szó, ritkán jutnak eszünkbe a képszerkesztő algoritmusok, esetünkben mégis a FloodFill eljárás az egyik legnagyszerűbb választás.

Algoritmus Négy irányú flood fill

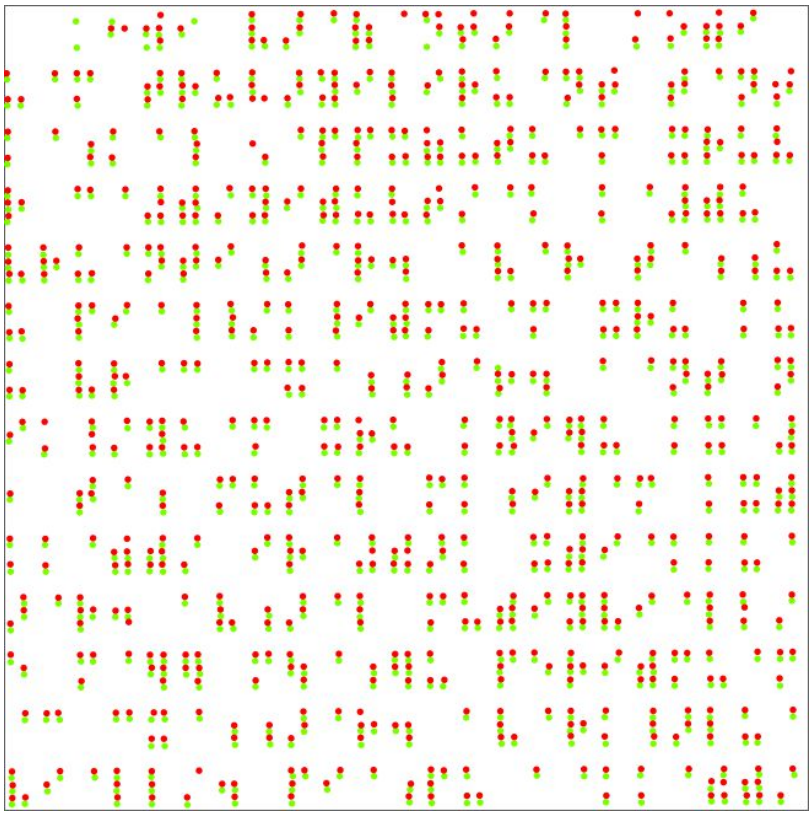
```
1:  procedure FloodFill4((u, v), KiindulásiSzín, Célszín, Kép)
2:      S ← ∅                                   ▷ S verem inicializálása
3:      Push(S, (u, v))                         ▷ A kiindulási pixelt adjuk a veremhez
4:      while S ≠ ∅ do
5:          (x, y) ← Pop(S)                     ▷ Vizsgáljuk a következő elemet
6:          if (x, y) ∈ Kép and I(x, y) = KiindulásiSzín then
7:              I(x, y) ← CélSzín               ▷ Színezzük a vizsgált pixelt
8:              Push(S, (x + 1, y))             ▷ A későbbiekben vizsgáljuk a környékét
9:              Push(S, (x, y + 1))
10:             Push(S, (x, y − 1))
11:             Push(S, (x − 1, y))
12:         end if
13:     end while
```

1. algoritmus. Négy irányú flood fill kitöltő eljárás [25]

Az algoritmusról röviden: a legegyszerűbb képszerkesztő programok is tartalmazzák, egy választott pontból kiindulva annak vele azonos színű (léteznek olyan változatai is, amelyek néhány százalékos eltérést megengednek) környezetét kitölti egy adott színnel. Létezik rekurzív és iteratív változata is, mi az utóbbit fogjuk használni (1. algoritmus). Ha a flood fill funkció az összes olyan pixelt, amelyiket kiszínezte összegyűjti egy listában és a munkája befejeztével ezt a meghívónak visszaadja, akkor máris meghatároztunk egy klasztert. Válasszuk kiindulási színnek például a világos részek színét (255-öt), ekkor ha a képünk összes pixelével egyenként lefuttatva összegyűjtjük a visszaadott listákat, akkor máris hozzárendeltük a világos pixeleket egy-egy (egyelőre még nem felismert) Braille ponthoz.

Határozzuk meg a klaszterek súlyozatlan középpontjait (szürkeárnyalatos bemeneti kép esetén a pixelek intenzitása alapján történő súlyozásnak is lenne értelme). Miután meghatároztuk az összes klasztert, érdemes a nagyon kevés pixelből álló elemeket eltávolítani a listából. Ehhez meg kell választanunk egy határértéket, mely természetesen függ az adott képtől (a Braille-pontok mérete pixelben mérve megváltozik, ha a kamerát közelítjük vagy távolítjuk). Ha a képhez optimalizált határértéket túl alacsonyra választjuk, zajos lesz a kimenetünk (sok hamis pozitív), túl magasra választva pedig helyesen deketált Braille-pontokat veszíthetünk el. A kellően magas elemszámú klaszterek középpontjainak listája lesz a pixelklasztereket készítő szűrőnk kimenete.

Ha vizsgálni szeretnénk a szűrő hatékonyságát, hasznunkra válhatnak az SVG képek, melyek a rasztergrafikákkal ellentétben nem pixeleket, hanem különböző geometriai alakzatokat (például köröket) tárolnak, s mivel renderelésüket majd külső program fogja végezni, XML formátumban egyszerűen exportálhatók a programunkból. A szűrő kimenetét (a felismert világos részek pirosra, a felismert sötét részek zöldre vannak színezve) a 16. ábrán láthatjuk.



16. ábra. A kellően nagy klaszterek középpontjai

4 A ponthalmazok feldolgozása

4.1 A ponthalmazok egymásra igazítása

Az előző szűrő eredménye két ponthalmaz, az egyik a megtalált és kellően nagy világos pontokat, a másik a kellően nagy sötét pontokat tartalmazza. Feladatunk, hogy ennek alapján meghatározzuk a Braille-pontok helyzetét az eredeti képen. Néhány feltételezés a megtalált pontok helyzetét illetően:

- A Braille-pontok döntő többségének esetében az előző szűrő mind a sötét, mind a világos részt sikeresen azonosította, a maradék pontok esetében pedig legalább az egyiket.

Ha megfigyeljük a kiindulási képet, látható, hogy a világos részek jelentős hányadára igaz:

- A hozzá legközelebb lévő sötét rész ugyanazon Braille-pont túlsó oldalán található árnyék.

Ezek után a feladatunk egyértelmű: keressük meg az összes világos pont esetében a hozzá legközelebbi sötét pontot és keressük meg a sötét pontokhoz legközelebbi világos pontokat is. Fontos felismerni, hogy ha P_1 ponthoz P_2 van legközelebb, P_2 ponthoz nem biztos, hogy P_1. Általánosságban ilyen feladatokra nagyon hasznosak a különböző, többdimenziós tereket partícionáló algoritmusok, melyek többsége általában valamilyen fa struktúrát, például quadtree-t hoz létre. A mi esetünkben is érdemes lehet ezek használatán elgondolkodni, azonban az elméleti megoldásnak nem képezik szerves részét, ezért alkalmazásukra akkor érdemes sort keríteni, ha az alkalmazásunk futási idejének ez kritikus része és jelentős mértékben lelassítja a feldolgozási folyamatot. Ez néhány száz pont esetén nem valószínű. Ha megfigyeljük jelen szűrő bemenetét, itt már csak két, maximum néhány száz pontot tartalmazó lista van, ellentétben az első szűrő esetleg néhány megapixeles felbontású színes képével. Joggal gondolhatnánk, hogy ezen szűrő a listák rövidsége miatt gyorsabb lesz, ez azonban sajnos a naív algoritmus esetén nem igaz. Az előző szűrők futási ideje az $O(m \cdot n)$ nagyságrendbe esik, ahol m és n a kép szélessége és magassága, e szűrő futási ideje $O(p \cdot q)$, ahol p és q a két lista hosszát jelölik, hiszen a távolságot a naív algoritmus minden lehetséges pontpárra, vagyis az alábbi halmaz:

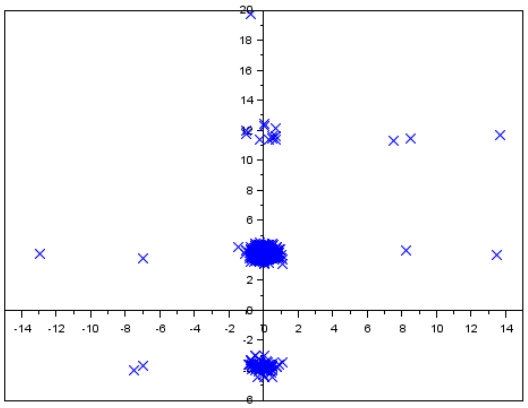
$$P \times Q \coloneqq \{ (p, q) \mid p \in P \wedge q \in Q \}$$

minden elemére kiszámolja (P és Q a két bemeneti halmaz). Quadtreek használata esetén ez a bemeneti listák hossza helyett azok logaritmusával lesz arányos, így jelentős gyorsulást

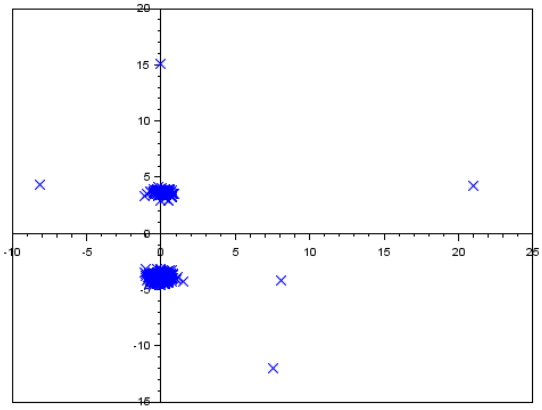
érhetünk el. Miután meghatároztuk minden pont esetében a hozzá legközelebb eső másik pontot (több ilyen esetén ezek közül egyet, nem feltétlenül determinisztikus módon), a kettő helyzetének különbsége megadja a vektort, mely segítségével az egyikből a másikba juthatunk. Az egyik lista (P-ből Q-ba):

$$V := \{ (p,q) \mid [\forall q' \in Q : dist(p,q) \leq dist(p,q')] \text{ ^ } [p_i = p_k \Rightarrow q_j = q_l] \}$$

A Q-ból P-be mutató lista vektorait analóg módon határozzuk meg. Ábrázoljuk a kapott vektorokat:



17. ábra. Világosból sötétbe



18. ábra. Sötétből világosba

Némely bemeneti kép esetén előfordulhat, hogy a hisztogram világos vagy sötét része túl sok zajos pixelt is tartalmaz. Ebben az esetben a fenti eljárástól eltekinthetünk és elegendő csak a kevés zajt tartalmazó csatorna adatait figyelembe venni. Ekkor a két halmazt nem szükséges egymásra igazítani, az egyiket figyelmen kívül hagyjuk és csak a másikat használjuk. Különösen előnyös ez például akkor, ha számítógép monitorjáról akarjuk leolvasni a Braille-szöveget, ahol általában fehér, sík felületen fekete pontok jelölik a Braille-pontok helyét, így tehát a fehér csatorna használata értelmét veszti.

4.2 A DBSCAN algoritmus

A következő szűrő működésének megértéséhez fontos az általa használt klaszterező algoritmus részletes bemutatása. A klaszterező algoritmusok többdimenziós ponthalmazok elemeit sorolják be különböző osztályokba (klaszterekbe) azok egymástól való távolsága alapján.

A DBSCAN három bemeneti paramétere:

- D: A pontok halmaza
- eps: A keresési távolság
- MinPts: Pontok min. száma

A keresési távolság és a pontok minimális száma meghatározza, hogy egy adott pont mekkora sugarú környezetében legalább hány másik pontnak kell lenni ahhoz, hogy ebből a pontból új klaszter indulhasson ki, vagy ha meglévő klaszter bővítése során jutottunk el ebbe a pontba, akkor rajta keresztül tovább bővíthessük-e azt.

Algoritmus DBSCAN klaszterező

```
1:  procedure DBSCAN(D, eps, MinPts)
2:      for each P ∈ D do
3:          if P.allapot = meglatogatva then
4:              continue
5:          end if
6:          P.allapot ← meglatogatva
7:          N ← D.kornyezetLekerdezes(P, eps)
8:          if N.elemekSzama < MinPts then
9:              P.zaj ← igen
10:         else
11:             C ← C + 1
12:             klaszterBovites(P, N, C, eps, MinPts)
13:         end if
14:     end for
15: end procedure
16: procedure KLASZTERBOVITES(P, N, C, eps, MinPts)
17:     P.klaszter ← C
18:     for each P' ∈ N do
19:         if P'.allapot ≠ meglatogatva then
20:             P'.allapot ← meglatogatva
21:             N' ← D.kornyezetLekerdezes(P', eps)
22:             if N'.elemekSzama ≥ MinPts then
23:                 N ← N ∪ N'
24:             end if
25:         end if
26:         if P'.klaszter = ismeretlen then
27:             P'.klaszter ← C
28:             P'.zaj ← nem
29:         end if
30:     end for
31: end procedure
32: function KORNYEZETLEKERDEZES(P, eps)
33:     return { P' | dist(P, P') ≥ eps }
34: end function
```

2. algoritmus. DBSCAN [26]

A DBSCAN nagy előnye a többi klaszterező algoritmussal szemben a bemeneti paraméterekben rejlik, a keresési távolságot és a pontok minimális számát ugyanis hatékonyan meg tudjuk becsülni.
A keresési távolság meghatározásához vessünk ismét egy pillantást a 17. és 18. ábrára! Mint látható, a minket érdeklő klaszter elemszáma nagyságrendekkel nagyobb a rajta kívül eső pontok számánál. A középpontját az origóra tükrözve egy másik, az előzőnél kisebb, de számottevő klaszterbe jutunk, itt találhatók azon pontok, melyek esetén az árnyék közelebb volt a szomszédos pont világos részéhez, mint a saját pont túlsó oldalához. A többi, szétszórt pont pedig annak köszönhető, hogy némely Braille-pont esetén a szűrő vagy csak a világos, vagy csak a sötét részt ismerte fel, s a hozzájuk legközelebb eső ellentétes elem a legközelebbi másik Braille-pont sötét vagy világos része, s ez értelemszerűen nagyobb távolságra van. Ha a képek hisztogramjához hasonlóan a vektorok hosszának eloszlását is hisztogramként tekintjük, könnyű lesz megbecsülni a fő klaszter középpontjába mutató vektor hosszát. Ebben még a szemközti klaszter is segít, hiszen a vektorok átlagos hossza megegyezik. Válasszuk ennek a becsült hossznak például a negyedét és meghatároztunk egy optimális eps paramétert. Ha feltételezzük, hogy a 17. és 18. ábra átlagos eloszlásnak tekinthető, úgy az átlagos hossz ötöde egy még gyorsabb becslési lehetőség.

Két fontos hibalehetőségre kell odafigyelnünk a paraméterekkel kapcsolatban:

- A két nagyobb klaszter az origó ellentétes oldalán nem olvadhat össze
- A fő klaszter nem eshet szét több számottevő darabra (apró részek leszakadhatnak)

Kiindulhatunk abból, hogy a legtöbb kép esetén a fő klaszterhez tartozik legalább a vektorok 50%-a. Ha eps-nek a fent említett értéket választottuk, feltételezhetjük, hogy a fő klaszter területére egy ilyen sugarú kört maszkként helyezve legalább az összes vektor 5%-át látjuk. Mivel bármely Braille képre ilyen vektoreloszlást várhatunk, az összes vektor számának 5%-a adott eps esetén megfelelő választásnak tűnik MinPts számára. Ezzel a DBSCAN algoritmusunk paramétereit meghatároztuk.

4.3 A sötét és világos ponthalmazok egymásra igazítása II.

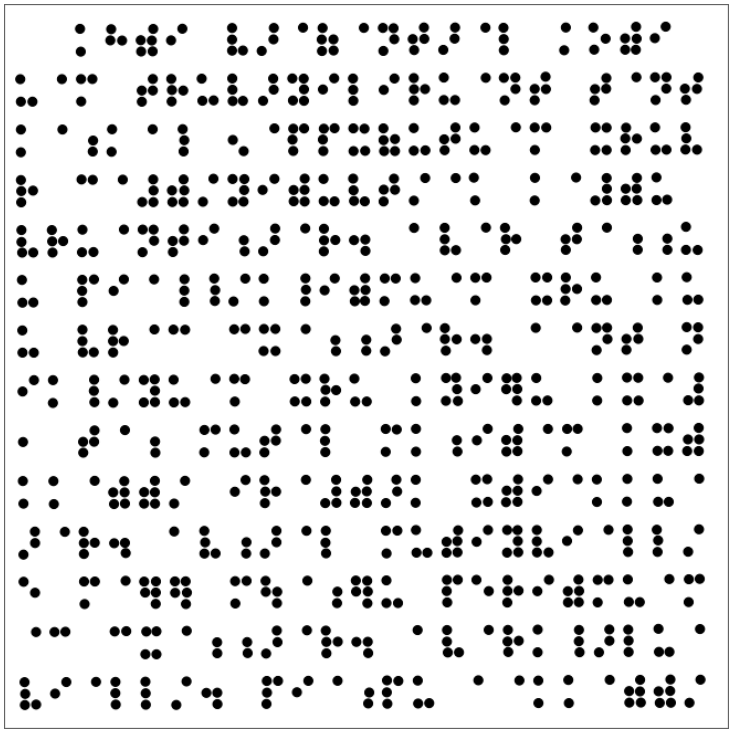
A DBSCAN algoritmusunk futtatása után az adott adathalmazra az előző fejezetben ismertetett paraméterekkel az alábbi eredményeket kaptuk:

A világos részekből a sötétekbe mutató vektorok klaszterei:		
Elemek száma	Középpont (x)	Középpont (y)
668	0,029922375	3,82555
A sötét részekből a világosakba mutató vektorok klaszterei:		
Elemek száma	Középpont (x)	Középpont (y)
666	-0,043875784	-3,8236504

Mindkét vektormezőben tehát 1-1 klasztert talált a DBSCAN algoritmus. A célunk, hogy a sötét és világos pontok helyzetéből, melyeket már ezelőtt megállapítottunk, meghatározzuk a Braille pontok helyzetét. Bár a két ponthalmaz elemei nem fognak pontosan egymásra illeszkedni, mégis viszonylag jól fedésbe hozhatók. Ezen problémakör egyik legfontosabb algoritmusa az ICP (Iterative Closest Point), melynek segítségével akár háromdimenziós szkennerek eredményeit is feldolgozhatjuk. Nagyszerű és kiforrott megoldás, azonban erre itt nincs szükség. Toljuk el az összes sötét színű detektált pontot a sötétből a világos felé mutató vektor felével és toljuk el az összes világos színű pontot a világos részekből a sötétek felé mutató vektor felével. Ezzel további számítás nélkül kielégítő minőségben megoldottuk a problémát.

Van tehát két egymásra igazított ponthalmazunk. Az esetek többségében egy világos pont közvetlen közelében van egy sötét. Néha azonban az egyik nem került felismerésre, ezeken a helyeken csak egy világos vagy egy sötét pont van. Természetesen előfordulhatnak hamis pozitív jelzések is, ezektől azonban tekintsünk el. Keressük meg az összes világos pont legközelebbi sötét színű szomszédját és az összes sötét pont legközelebbi világos szomszédját. Ha ezek egy adott távolságnál közelebb vannak, egyesítsük őket egy ponttá a középpontjaik súlyozatlan közepében, ha az adott távolságnál közelebb nincs ellentétes színű szomszéd, akkor tartsuk meg az adott pontot a saját pozíciójában. Az említett távolság meghatározásához remek kiindulási alap a DBSCAN által megtalált fő klaszterek középpontjaiba mutató vektorok hossza. Ez a távolság egy Braille-pont sötét és világos oldala között. Miután egymásra igazítottuk a sötét és világos pontokat, azok ennél kisebb távolságra lesznek egymástól, az eredeti Braille-pont közepe táján. Két különböző Braille-pont távolsága azonban ugyanazon Braille-pont két oldalának távolságánál nagyobb. Válasszuk például az

említett vektorok hosszának a felét, s az egymáshoz annál közelebb eső sötét és világos pontokat tekintsük ugyanazon Braille-ponthoz tartozónak.



19. ábra. Független Braille-pontok

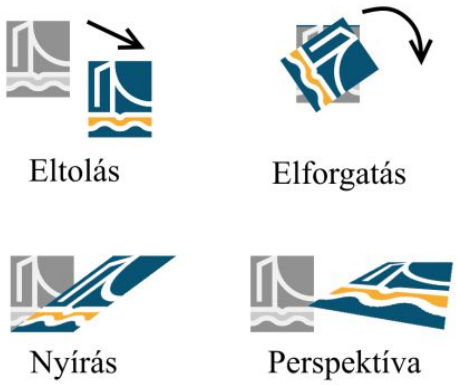
Az eljárás eredménye a 9. ábrán látható. Érdemes összevetni a kiindulási képpel, a legtöbb karakter pontjai helyesen kerültek felismerésre. A független Braille-pontok meghatározásával mérföldkőhöz érkeztünk, a továbbiakban azt kell meghatároznunk, milyen karakterekké állnak össze az egyes pontok.

4.4 Lineáris leképezések, affin transzformációk

Tekintsünk vissza a 9. ábrára és ismerjük fel, hogy ez az elrendezés egy szerencsés eset, a várható kameraképek halmazának egy speciális részhalmaza. Ennek két fő oka van:

- A Braille-sorok és oszlopok párhuzamosak a kép x és y tengelyeivel
- A kamera tengelye a fotózáskor megközelítőleg merőleges volt a kép mindkét tengelyére

A kamera mozgatásával a kép is megváltozik, ezt geometriai transzformációkkal írhatjuk le (a gyakoribbakat lásd: 20. ábra).



20. ábra. Lineáris transzformációk

Eltolás: Esetünkben megoldása triviális. Az ún. homogén koordinátákat ismerő olvasó tisztában van ennek okával: a Braille-pontok egymáshoz képesti helyzetét két helyvektor különbsége, azaz egy irányvektor írja le. Az eltolás művelete pedig az irányvektorokra nincs hatással.

Elforgatás: Ez a problémakör már jelentősebb. Visszagondolva az eddigi szűrők munkájára, nyilvánvaló, hogy azokat nem befolyásolta, hiszen egyes független pontok kerültek felismerésre. A pontok (körök) esetében a rotációs szimmetria miatt nem jelentkező jelenség most előtérbe kerül, mielőtt megpróbálnánk felismerni a Braille-karaktereket, meg kell határoznunk, hogy milyen szögben elforgatott Braille-karaktereket keressünk egyáltalán.

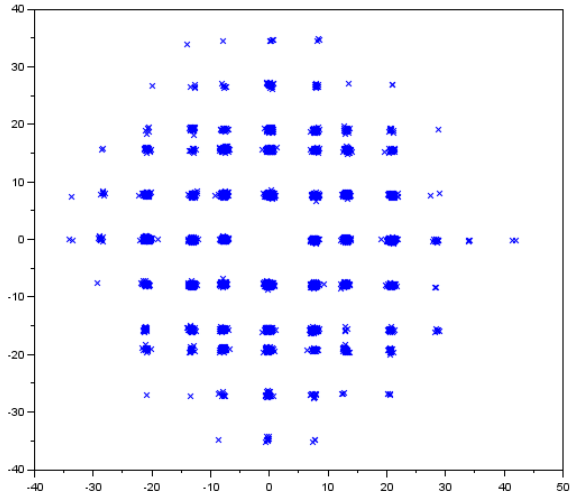
Nyírás: A Braille-pontok árnyékára tett hatása nem befolyásolja eddigi algoritmusaink munkáját, a Braille-karakterek felismerése során viszont már a felismerés megkezdése előtt ki kell számolnunk a transzformáció paramétereit, s – akár a Braille ponthalmaz, akár a transzformáció inverzével a felismerendő Braille-karakter mintákat transzformálva – el kell érnünk, hogy a ponthalmaz pontjai és a Braille-karakterek általánosan ismert mintái fedésbe

hozhatók legyenek. Fontos tulajdonsága, hogy hatására az eredetileg párhuzamos vektorok párhuzamosak maradnak. Vagyis a Braille-karakterek pontjainak egymáshoz viszonyított helyzete, azaz például az 1-es pontból a 4-es pontba mutató irányvektor az egész kép esetében konstans (helyfüggetlen) marad. Ez minden affin transzformációra (pl. eltolás, elforgatás) igaz. A nyírás jelensége képek készítése során önmagában nem jelentkezik.

Perspektíva: Nem affin transzformáció, viszont minden kép készítése során az eredeti objektum perspektívikus leképezését kapjuk, vagyis figyelembe kell vennünk. Az eddigiek során elhanyagoltuk a jelenséget, s ez erősebben torzító perspektívák esetében, vagyis olyan esetekben, ahol a Braille karaktereket tartalmazó papírlap normálvektora és a kamera tengelye által bezárt szög nagy, jelentős problémává válhat. Míg azonban eddigi szűrőink eredményére mindössze negatív hatást gyakorol, a pontok valós helyzete és a detektált pozíció között kis eltérést eredményez, a karakterfelismerő algoritmusunk esetén figyelembe nem vétele annak munkáját teljesen lehetetlenné teszi. A megoldásra több lehetőségünk is van: az egyszerűbb, ám rosszabb eredményű lehetőség, ha az egész ponthalmazra globálisan megállapítunk egy a perspektívához hasonlatos nyírást, s annak transzformációját használjuk. Ennél jobb, ha a képet régiókra osztjuk és minden régióra más-más nyírást állapítunk meg. A perspektíva így a mi esetünkben egész jól közelíthető. A legszebb megoldás mégis az, ha elfogadjuk, hogy egy Braille-karakter különböző pontjaiból egymásba mutató irányvektorok nem helyfüggetlenek, vagyis az irány attól függ, hol helyezkedik el a pont. Lehetőségünk van ezt az algoritmusban figyelembe venni és azt úgy megírni, hogy ne feltételezze a párhuzamosságot, hanem minden pont esetén más-más, a többivel esetleg nem egybevágó felismerendő mintát keressen, vagy pedig az algoritmus futtatása előtt a ponthalmazt a perspektívikus transzformáció mátrixának megállapítása után annak inverzével vissza kell transzformálnunk. Mindkét lehetőség optimális elméleti megoldást jelent, így a döntő tényezővé az algoritmus sebességére tett hatásuk válik.

4.5 A karakterek orientációjának és méretének megállapítása

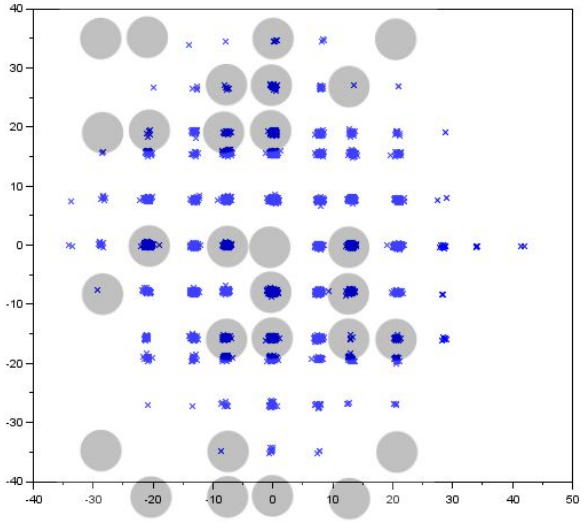
Ahhoz, hogy a megtalált különálló Braille-pontokat karakterekké szervezzük, meg kell állapítanunk, hogy hol található a karakter középpontja, milyen orientációjú a karakter és azt, hogy azon belül melyik pozícióban van az adott pont. Az első megközelítésben tekintsünk el a perspektivikus leképezésektől, tegyük fel, hogy a valóságban párhuzamos vektorok a kamera projekciós síkjára vetítve is megközelítőleg párhuzamosak maradnak. A Braille-írást elemezve megállapíthatjuk, hogy igen jellegzetes minta, a pontjai alatt, felett, valamelyik oldalán, esetleg tőle átlósan igen gyakran található egy másik pont. Használjuk ki ezt a tulajdonságát, hogy megállapítsuk a karakterek egyik pontjából a másikba mutató vektorokat, melyek segítségével generálhatóvá válnak a felismerendő minták (ne felejtsük el, hogy egy elforgatott Braille-karakter egy nem elforgatott sablonnal nem fog egyezést mutatni, a forgatás mértékét pedig egyelőre nem ismerjük).
Keressük meg minden egyes megtalált Braille-pont n db legközelebbi szomszédját, számítsuk ki az adott pontból ezen szomszédokba mutató vektorokat, majd tekintsük meg az összes pontra kiszámított vektorok halmazát (21. ábra). A 22. ábrán ugyanez egy adott pont szemszögéből.



21. ábra. A legközelebbi 5-5 szomszédos pont relatív helyzete

A következő lépés az adathalmaz kiértékelése (klaszterezés után). Először is határozzuk meg, hogy hogyan jutunk el egy Braille-pontból annak közvetlen szomszédjába! Mint a 21. ábrán is látszik, ehhez elég a klaszterezés után a kellően nagy (azaz nem zaj) klaszterek közül a négy

legközelebbit kiválasztani. Ellenőrízhető, hogy valóban-a keresett vektorokat találtuk-e meg: hosszuk egyenlő, irányuk páronként ellentétes, nyírás esetén nincs 90°-os szög feltétel.



22. ábra. A 21. ábra lehetséges Braille karakterekkel kiegészítve

A 21. ábrát a középpontja körül 90°-kal elforgatva a kapott minta nem lesz egybevágó az eredetivel. Ez általánosított esetben is igaz: nem tudjuk az előző vektorok alapján meghatározott **i** és **j** egységvektorokat egyértelműen az x és y tengelyekhez rendelni. Megoldás: generáljuk a 21. ábrán látható mintázatot a Braille-írás specifikációja alapján.

Legyen a generáláskor figyelembe venni kívánt szomszédos karakterek száma k_x, a többi konstanst pedig definiáljuk a Braille-specifikáció alapján. Ekkor a generált halmaz:

$$G_x := \{m \cdot d_{px} + n \cdot d_{kx} | \, [|m| < p_x]\text{^}[|n| < k_x]\text{^} \, [m,n \in \mathbb{Z}]\}$$

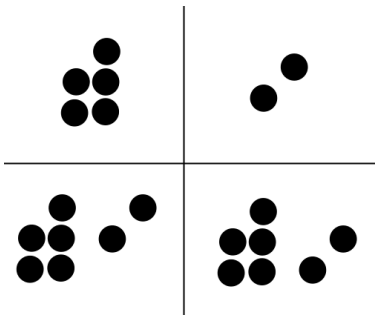
Analóg módon számítsuk ki az y tengelyre G_y-t, s a két halmaz Descartes-féle szorzatának elemeit a tengelyek **i** és **j** egységvektorainak együtthatóiként használva megkapjuk az összes generált vektor halmazát. Mind a generált mintát, mind a detektált mintát korlátozzuk a középpont körüli r sugárra úgy, hogy azok ne túl sok, de mégis elegendő pontot tartalmazzanak (például 20-30-at). Számoljuk meg a fedésbe jött pontokat, majd cseréljük ki a két egységvektort egymással, az együtthatókat felhasználva számoljuk újra a generált vektorokat, korlátozzuk a környéket r sugár alapján és számoljuk meg az így fedésbe hozható pontokat. A nagyobb mennyiségű fedésbe jött pontpárt eredményező elrendezést válasszuk. Mivel a generált minta esetén az x és y tengelyek ismertek és ezen minta orientációja alapján meghatároztuk a megtalált ponthalmaz orientációját is, immár tudjuk a karakterek méretét és orientációját, azok középpontjának pozícióját azonban még nem.

4.6 A karakterek azonosítása a pontok alapján

Vessünk egy pillantást a 23. ábrára és fedezzük fel az alábbi jelentős különbséget a két karakter között: ismerve a karakterek méretét és orientációját, a bal felső sarokban lévő karaktert egyértelműen azonosítani tudjuk, a jobb felső sarokban lévőt azonban nem. Egymás mellé helyezve őket azonban már egyértelművé válik a helyzet. Vezessünk be két új fogalmat, melyekre a továbbiakban hivatkozhatunk:

- Primer Braille-karakter: Olyan karakter, mely kizárólagosan saját pontjainak egymáshoz viszonyított helyzete alapján, a karakter középpontjának ismerete nélkül is azonosítható. A szakirodalomban néha találkozhatunk az *erős karakter* kifejezéssel is.
- Szekunder Braille-karakter: Olyan karakter, mely csak a saját középpontjának ismeretében ismerhető fel.

Ahhoz, hogy egy karakter a primer karakterek halmazába tartozzon szükséges, hogy minden szélső sorában ill. oszlopában legalább 1-1 pontot tartalmazzon, azaz a középpont meghatározása után ne kaphassunk a karakter pontjait bármely irányba eltolva azonos középponttal egy másik, szintén létező Braille-karaktert (lásd: 23. ábra, alsó sor). A 64 féle lehetséges Braille-karakter közül, 32 primer és 32 szekunder. Ez elég jó arány ahhoz, hogy bármely kellően hosszú Braille-szöveg esetén nagyon jó eséllyel legyen a karakterek között elegendő primer. Ismerjük fel először őket!



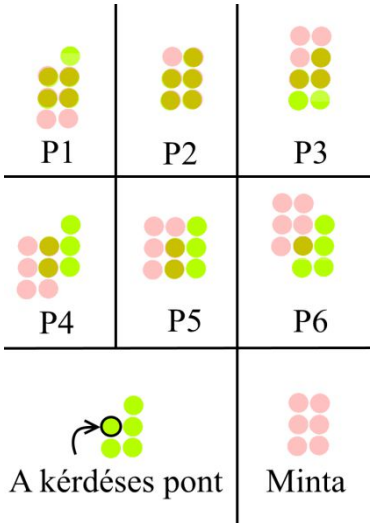
23. ábra. Primer és szekunder Braille karakter

4.7 A primer karakterek felismerése

Ismerve a karakterek méretét és orientációját, ismét generáljuk a környezetükben található pontok halmazának mintáját, ezúttal azonban az alábbi változtatásokkal:

- Ne vegyük figyelembe a szomszédos karaktereket.
- Egyszerre egy pont szemszögéből készítsük el a szomszédok lehetséges helyzetének mintáját.

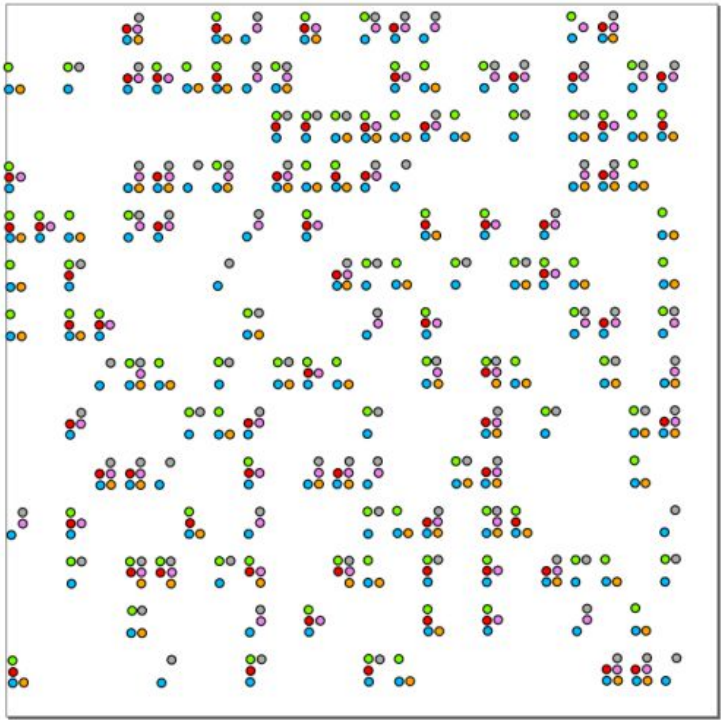
Vagyis ha az adott pontunk az 1-es pozícióban lévő (a bal felső), akkor alatta előfordulhat két másik Braille-pont, tőle jobbra 1 pont, illetve e pont alatt szintén kettő másik. Ezt a mintát generáljuk mind a 6 lehetséges Braille-pozícióra! Tulajdonképpen ugyanazt a mintát (ha az origóban lévő pontot, azaz saját magát is hozzászámítjuk) kaptuk önmagához képest eltolva.



24. ábra. Pont pozíciója a karakteren belül

Próbájuk meg beazonosítani a 24. ábra kérdéses pontját: ábrázoljuk a pontot és környezetét mind a 6 minta maszkjával fedésbe hozva. Egyértelmű, hogy a kérdéses pont a 2-es pozícióban van, hiszen a mintával fedésbe hozva 5 átlapolt pontot eredményez, többet, mint bármely másik elrendezés. Ilyen maximumot csak és kizárólag a primer karakterek pontjai fognak rendelkezni. Ha több azonos értékű maximum létezik, akkor a karakter szekundernek minősül. Megállapítottuk tehát, hogy a kérdéses pont egy Braille-karakter 2-es pozíciójában van és ebből kiszámíthatjuk, hogy az adott karakter középpontja hol található (a 2-es pozíciótól kissé jobbra). Az algoritmust futtatva az összes pontra (így az adott karakter mind az öt pontjára is), a pontokhoz rendeljük hozzá a pozíciójukat a karakteren belül, és a karakter becsült középpontját. Az eredmény: 5 közel azonos középpontban található karakterre

hivatkozó pont, melyek egy ismeretlen karakter alábbi pozícióiban találhatók: 2, 3, 4, 5, 6. Vonjuk őket össze, és kaptunk egy karaktert, melynek ismerjük a középpontját és azt, hogy az adott öt Braille-pont alkotja, melyeknek a karakteren belüli indexét is megállapítottuk (15. ábra).



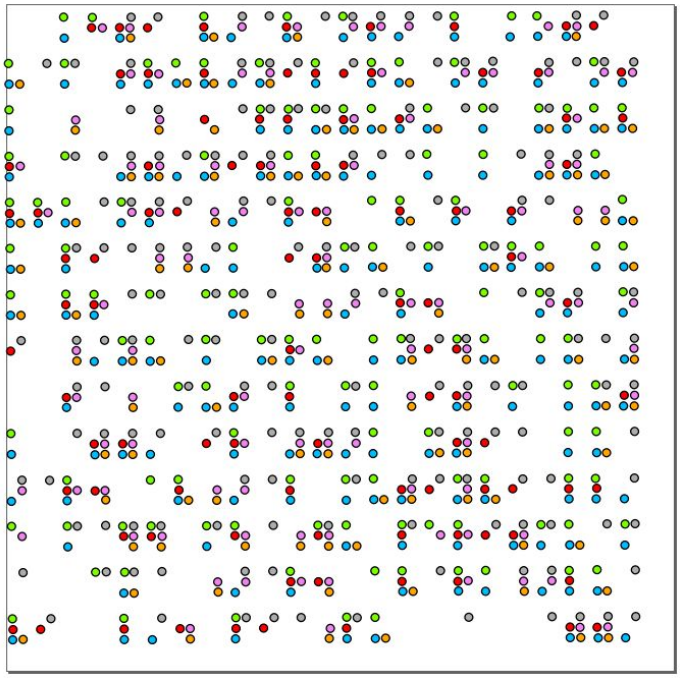
25. ábra. A primer karakterek felismerése után

4.8 A szekunder karakterek felismerése

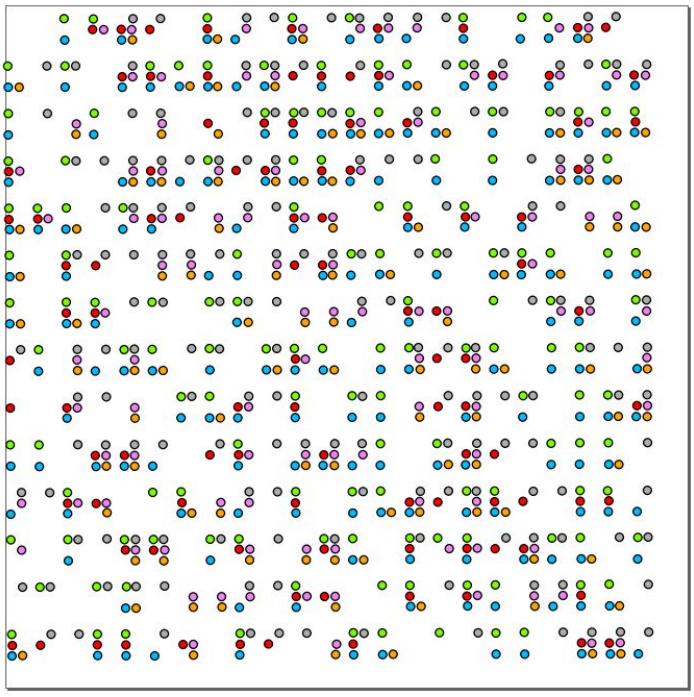
Ismerve a primer karakterek helyzetét, nincs nehéz dolgunk. Az iterációs algoritmus működése, a könnyebb érthetőség kedvéért ezúttal szöveges formában:

1. Legyen az összes eddig megtalált karakter végleges.
2. Keressünk a szomszédságukban (4 irányban) karaktereket:
   2.1. Ha nem találtunk, ellenőrizzük, hogy nem hagytuk-e el a kép Braille-pontokat tartalmazó területét (ez a pontok pozíciója alapján meghatározható, például egy poligon formájában), amennyiben nem, illesszünk be a kérdéses pozícióba egy üres karaktert.
   2.2. Ha találtunk, de az a karakter még nem végleges, akkor nemrég illesztettük be, javítsuk a középpontjának becslését annak előző pozíciója (súly: eddigi becslések száma) és az új keresési pozíció (súly: 1) súlyozott átlagával.
3. Ha az összes karakter szomszédságában kutakodva már nem tudtunk egyetlen új karaktert sem beszúrni, akkor elértük az algoritmus végét, az összes felismerhető karakter felismerésre került.
4. Futtassuk a szekunder pontokat karakterekhez rendelő algoritmust, mely csak és kizárólag a még nem végleges karaktereken végez módosításokat.
5. Ugorjunk ismét a 2. pontba.

A szekunder pontokat karakterekhez rendelő algoritmus annyiban tér el a primertől, hogy nem a szomszédos pontokat vizsgálja, hanem mind a 6 lehetséges karakteren belüli pont-indexre feltételezi, hogy a kérdéses pont az adott indexszel rendelkezik, az ebből becsült hat lehetséges karakter középpont közül azonban vagy egyetlen helyen lesz (nem véglegesített) karakter, úgy a pont ahhoz tartozik, vagy egyetlen helyen sem, ebben az esetben a karakterrasztert bővítő algoritmus iterációi még nem jutottak el a távolabbi primer karakterektől az adott szekunder karakterig. Természetesen a hozzárendelt pontok a karakterek becsült közepét korrigálják azok valós közepére, így a becslési hiba az iterációk során nem növekszik. A példánkban szereplő Braille-szöveg esetén két iteráció elég volt a teljes szöveg felismeréséhez. A lépések és a végeredmény a 26. és 27. ábrán láthatók.



26. ábra. A szekunder karakterek első iterációja



27. ábra. A szekunder karakterek második iterációja (végeredmény)

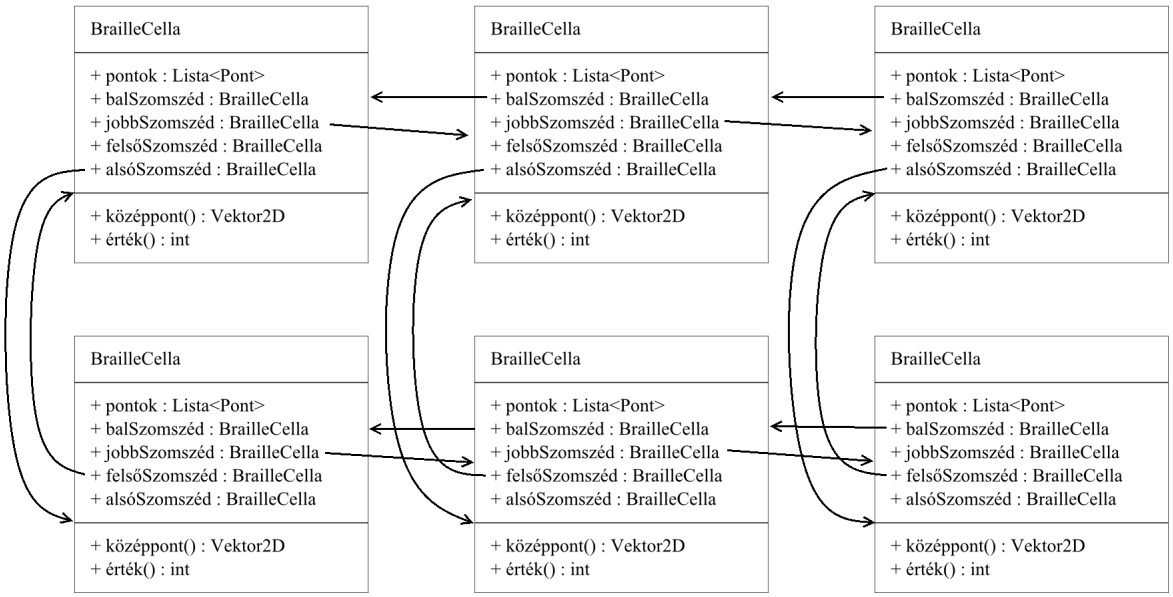
5 A Braille-cellák feldolgozása

5.1 A szomszédos cellák meghatározása

A Braille cellák szöveggé alakításának első lépése, hogy meghatározzuk a szomszédsági viszonyokat. A karakterek felismerése során kiszámítottuk azok középpontját. Erre az őket alkotó pontok helyzetéből tudunk következtetni, például egy Braille cella esetén, amelyet a 2 és 5 számú pontok alkotnak (lásd: 4. ábra) a középpont a két pont középpontját összekötő szakasz felénél lesz. Amennyiben a karakter szekunder úgy egy szomszédját már a létrehozásakor meghatároztuk: az a primer karakter, amely mellett elhelyezkedik. Ez a szomszéd a második iterációtól kezdve természetesen lehet egy másik szekunder karakter is. Tekintve, hogy ismert a vektor, amely az adott cellától balra/jobbra található másik cellába mutat (ezt a 22. ábrán látható adatok alapján határozza meg a szoftver), valamint az alsó/felső szomszéd relatív helyzetét meghatározó vektort is kiszámoltuk, mindössze annyi a dolgunk, hogy minden egyes cella esetén, ahol az adott szomszéd még nem ismert (null mutató) a feltételezhető környék környezetében (a keresési kör sugara maximum a szomszédos cellák vízszintes távolságának fele legyen) cellákat keresünk és találat esetén tároljuk.
Fontos odafigyelni az adatok konzisztenciájára:

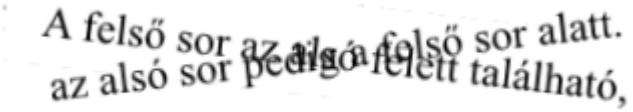
- Két cella vagy ne legyen egymásnak szomszédja, vagy kölcsönösen legyenek szomszédosak (értelemszerűen bal/jobb, illetve alsó/felső).
- Egy cella nem szomszédos önmagával (a keresési kör sugarának fent említett megválasztása ezt implikálja).

Megjegyzés: Szomszédos cella alatt csak a közvetlenül szomszédos raszter cellában található másik karaktert értjük, ha egy raszter (például egy teljes sor vagy oszlop) üresen marad, úgy a mellette található karakterek esetén a szomszéd mutató az üres rasztert nem ugorja át, hogy az utána következő, távolabbi cellára mutasson, hanem mindkét cella esetén null mutató lesz.
E lépés során különösen nagy hasznát vesszük az objektum orientált programozási módnak. Vegyük észre, hogy az általunk használt struktúra (28. ábra) tulajdonképpen az egydimenziós duplán láncolt lista (doubly linked list) két dimenzióra kiterjesztett változata.



28. ábra. Kétdimenziós duplán láncolt adatstruktúra

A számítógépes geometria területén számos ehhez hasonló adatstruktúra kerül felhasználásra a határolóelem reprezentáció során, például a szárnyas él (winged edge) vagy a duplán láncolt felület lista (DLFL, doubly linked face list) [23]. A geometriai modellezés területére érdemes kitérni, hiszen gyakran találkozhatunk a manifold fogalommal. Ennek jelentése: olyan topologikus tér, mely minden pontjában lokálisan euklideszi tér. Használatára remek példa a 3D nyomtatás, hiszen egy számítógéppel megalkotott objektum fizikai megvalósításához ez feltétel, vagyis a 3D nyomtató szoftvere csak manifold geometriai alakzatokat tud feldolgozni. A mi esetünkben a *nem manifold felület* hibajelenségét elegendő egy példával érzékeltetni (29. ábra, a hiba természetesen a Braille-szöveg esetén fordul elő, a könnyebb láthatóság kedvéért a példában a Braille-pontokkal való ábrázolástól eltekintettem), mely elsősorban hibás karakterfelismerés esetén fordul elő. Itt látható, hogy az alsó sor a sor elején a felsőt alulról határolja, a sor végén pedig felülről.



29. ábra. Nem manifold felület

5.2 A Braille-szöveg síkjának linearizálása

Hasonlítsuk össze a 28. ábrán látható struktúrát egy természetes szöveggel. A legszerencsésebb esetekben nincs sok különbség. Fontos azonban megjegyezni, hogy a szövegszerkesztők (és a szövegfelolvasó modulok) által felhasznált szövegek lineárisak, míg a mi Braille-szövegmezőnk nem. Ennek megértéséhez gondoljunk bele, hogyan szerializálnánk (alakítanánk soros karakterfolyammá) ezt a szöveget, illetve a Braille-szövegmezőt, ha például a merevlemezen szeretnénk tárolni. E bekezdés esetén elegendő az egyes karaktereket (H, a, s, o...) egymás után írni és készen vagyunk. A Braille-szöveg esetén meg kell határoznunk, hogy honnan kezdjük és hogyan haladunk a szomszédsági relációk alapján. A problémakör könnyebb megértéséhez vessünk egy pillantást a 30. ábrára!

```
Levél          Március 8.

    Ez egy olyan
szöveg, melyet
először linearizálnunk
kell.
```

30. ábra. Linearizálandó szöveg

A 30. ábrán látható szöveget alkalmazásunk nem tudná felolvasni, ennek oka az, hogy három különböző blokkból áll (cím, dátum, szövegtörzs). A különböző blokkok felolvasása önmagában nem probléma, annak megállapítása azonban, hogy a blokkok ténylegesen elkülönülnek-e és nem csupán hibás felismerésről van szó egy újabb szoftvermodul fejlesztését igényelné, így ettől eltekintünk. Egy egységes blokknak tekintünk minden olyan karakterhalmazt, melyen belül a szomszédsági relációkat követve minden karakterből minden másik karakterbe el lehet jutni. Tegyük fel, hogy elegendő a szövegtörzset felolvasnunk. A bal felső karakternél szeretnénk kezdeni. A bal felső karaktert így definiálhatjuk:

- Karakter, melynek bal oldali és felső szomszédja null mutató.
- Karakter, az előző kritériumnak megfelelő karakterek közül legfelül helyezkedik el.
- Amennyiben ilyenből több van, úgy ezek közül a leginkább balra lévőt válasszuk.

Első olvasatra egyszerűnek tűnik, hiszen a felül/balra kifejezések alapján általában az x és y tengelyekre asszociálunk. Vegyük figyelembe azonban, hogy szövegünk általában nem vízszintes, így a koordinátákat a Braille-szöveg koordinátarendszerébe kell transzformálnunk.

6 A Braille-kódsorozat karakterlánccá alakítása

6.1 A Braille-kód fordító

Utolsó lépésként a Braille-kódot, melyet ekkor egész számok (0..63) lineáris sorozataként tárolunk, át kell alakítanunk felolvasható szöveggé. Vegyük észre, hogy ez az első nyelvfüggő része az alkalmazásunknak, így más nyelvek támogatásának beépítése nem jelent nehézséget.
A továbbiakban a magyar nyelvű fordító működése kerül ismertetésre, más nyelvek fordítóinak logikus felépítése ettől eltérhet, ez a nyelv sajátosságaitól is függ. A magyar fordító két lépésben dolgozik. Először átkonvertáljuk a kódsorozatot egy ASCII/Unicode karakterlánccá, melyhez az 1. táblázat adatait használjuk. Ezután a karakterláncot a feldolgozó rutin a második lépésben az 1.2.2 fejezetben tárgyalt szabályok alapján elkészíti a végleges magyar szöveget. A feldolgozó rutin megvalósításához érdemes elgondolkodni az FSM (véges állapotú gép) használatán, mely magyar nyelv esetén előny, más nyelvek esetén szinte már feltétel (a német nyelvű Braille-írás esetén használnak olyan jelet, mely az összes őt követő karaktert nagybetűssé alakítja egészen addig, amíg az ezt feloldó karakterhez nem jutunk). Lássunk egy példát (Magyarország):

Braille-kód átalakítása és az FSM működése											
Braille-kód	48	13	1	57	1	23	21	23	49	8	27
1. lépés után	A	m	a	G	a	r	o	r	S	á	g
FSM szöveg kimenet		M	a	gy	a	r	o	r	sz	á	g
FSM állapotváltozás	↑	↓									

Vegyük észre, hogy egy Braille-karakter hatására a kimeneten megjelenhet 0, 1, de akár 2 karakter is (más nyelvek használatakor akár még több, pl. *sch* a német esetén).
Elérkeztünk a munkánk végéhez, a modul kimenetén megjelenik a kész, linearizált szöveg. Némely esetben (hibás felismerés) ez tartalmazhat hibákat, s bár egy helyesírás-ellenőrző modul használata nem jelent 100%-os megoldást, ártani általában nem árt, így érdemes megfontolni a beépítését, hiszen a karaktereket kódoló mezők esetén jelentkező ponthibák és eltolási hibák jelentős részét javítani tudja.

7 Az Android API és szolgáltatásai

Ebben a fejezetben röviden összefoglaljuk, hogyan kapcsoljuk össze a létrehozott központi Braille-szöveg felismerő könyvtárat a mobiltelefon operációs rendszerével és annak szolgáltatásaival. Szoftvermérnöki szempontból vizsgálva a problémát: érdemes megfelelő interfészek és adapterek alkalmazásával elválasztani a könyvtárat a telefontól, így az bármely készüléken (akár számítógépen is) futtatható lesz, bár nem lesz hozzáférése a telefon kamerájához és szövegfelolvasó motorjához, állóképeket Braille-szövegekről azonban fel tud dolgozni.

7.1 A kamera

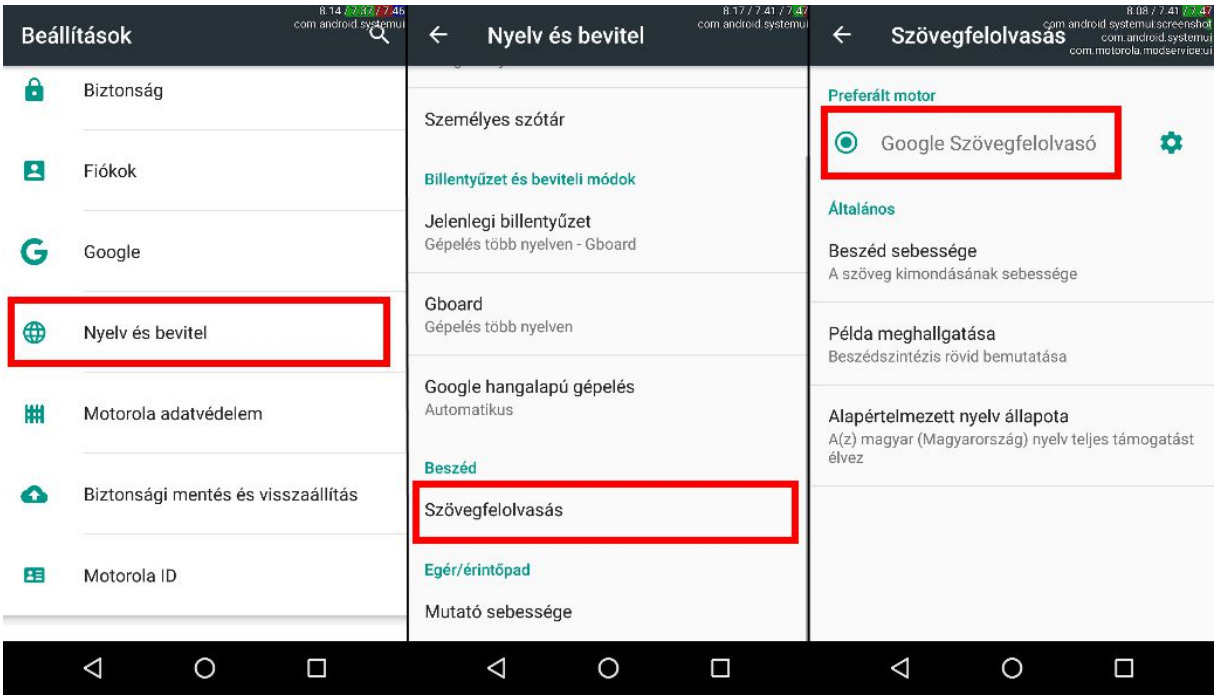
Az Android telefonok e dolgozat írásakor két különböző API-t is kínálnak a kamera használatához a Camera és Camera2 modulokat. Előbbinek támogatása hamarosan megszűnik, utóbbi viszont számos régebbi telefonon nem elérhető. Tekintve, hogy később könnyen módosíthatunk rajta, bármelyiket választhatjuk, illetve készíthetünk külön-külön appot régebbi és újabb telefonokhoz.
Sokkal fontosabb kitérnünk az API szolgáltatásaira, ezen belül pedig a ppi (illetve mm/pixel) értékek meghatározására. Az API rendelkezésünkre bocsátja a kamera látómezejének vízszintes és függőleges szögeit, valamint a fókusztávolságot. Tekintve, hogy a Braille-szövegeket általában síkbeli lapra nyomtatják, mely a fényképezéskor párhuzamos a telefon síkjával, az említett adatokból a kép felbontását is figyelembe véve meghatározható, hogy egy pixel szélessége és magassága a valóságban adott távolság esetén hány mm-nek felel meg. Tekintve, hogy a Braille-szövegek esetén annak mérete szigorúan meg van határozva (más méretű Braille-szöveg nem használatos, hiszen akkor teljesen más érzést keltene olvasáskor), ki tudjuk számítani a pontok várható méretét. Ez az algoritmusunk számára hatalmas segítség, hiszen az ennél jóval kisebb detektált pontokat zajnak, az ennél jóval nagyobbakat pedig árnyékos illetve fényes területeknek tekinthetjük, ezáltal számos hiba már korán felismerhető és kiszűrhető.
A kamera képes élőképet is szolgáltatni, s tekintve, hogy ennek fekete-fehér csatornája függetlenül és tömörítetlenül hozzáférhető, érdemes használni, hiszen a kitömörítés és az RGB-szürkeárnyalatos konverzió sok processzoridőt igényel.

7.2 A szövegfelolvasó és a rendszer beállításai

A szövegfelolvasó modulok komplexitásának megbecsléséhez érdemes röviden áttekinteni a magyar nyelven is elérhető rendszerek listáját:

- BME Multivox
- BME Profivox
- Google Android TTS

Utóbbi bárki számára ingyenesen elérhető az Androidos telefonján, s hangminőségben megelőzi versenytársait. Tekintve azonban, hogy ez egy szövegfelolvasó modul, szabadon kicserélhető a telefon beállításainak megváltoztatásával (31. ábra). A helyes működéshez fontos, hogy a megfelelő szövegfelolvasó modul és a megfelelő felolvasási nyelv legyen kiválasztva a rendszer beállításainál. Természetesen bármely más, az API interfészét használó szövegfelolvasó modul telepíthető és felhasználható.



31. ábra. Szövegfelolvasó motor kiválasztása

A szoftver telepítése egyszerű, mint minden más alkalmazás, ez is APK formátumot használ, Amennyiben nem a Google Play áruházból telepítünk szoftvert, úgy ezt az így keletkező biztonsági rés miatt a beállításoknál szintén engedélyeznünk kell (a kamera engedélyezéséhez hasonlóan).

8 Az eredmények kiértékelése

8.1 Az egyes részfeladatok eredményei

Miután elkészült a szoftver és lefuttattuk az integrációs teszteket is, foglaljuk össze röviden az eredményeket, a moduláris felépítés miatt ezt célszerű részfeladatonként megtenni: A százalékos értékek esetén a 100% azt jelenti, hogy a modul jelenlegi formájában alkalmas egy olyan elméletben létező szoftverben történő felhasználásra, mely egy átlagos telefonon 10-20 fps sebességgel képes képfolyamot értelmezni (közel hibamentesen).

- A telefon kamerájához történő csatlakozás, a kamera színes élőképének átalakítása fekete-fehér bitmap állóképpé: 90%, a rendszer kielégítően stabilan és gyorsan működik, egy átlagos telefon egy magot és Java programozási nyelvet használva 640x480 felbontás mellett kb. 15-30 fps sebességgel tudja a képeket a videóból kimásolni, ez céljainkra bőven megfelel.
- A képek normalizálása: 60%, a normalizáló algoritmusok stabilan futnak, az objektum-orientált képkezelő osztály sokkal gyorsabb a gyári Java osztálynál, azonban sebessége nem összemérhető egy natív C++-kód sebességével. Ahhoz, hogy a jelenlegi néhány másodperces lefutási időnél gyorsabban működjön az algoritmus a telefonon is, komoly sebességmérésekre és architekturális megfontolásokra van szükség. Egy remek példa ezen megfontolások komplexitására: meg kell gondolnunk, hogy például a Gauss-szűrő mátrixa milyen sorrendben haladjon végig a pixeleken úgy, hogy az L1 cache által biztosított sebességelőny a leginkább ki legyen használva. Természetesen figyelembe kell vennünk, hogy az Android különböző platformokon is elérhető. Alternatív megoldásként a kritikus szegmensek esetén érdemes például a GPU gyorsításra és más lehetőségekre is vetnünk egy pillantást [24].
- A Braille-kód elkülönítése a környezettől a képen: 0%, nem volt előre meghatározott cél, azonban a használhatósághoz erre is szükség van. Meg kell tehát határoznunk, hogy hol helyezkedik el az olvasandó kód a kamera képén, jó eséllyel általában nem tölti ki az egész látóteret, a lényegtelen részeket ignorálnunk kell.
- A kép szegmentálása, a pontok felismerése: 60%, a fő problémát a változó bemeneti képek jelentik, némelyek helyenként árnyékokat tartalmaznak, mások zajosak, s némely képeken nem kidomborodó Braille-karakterek láthatók, hanem sima feketék (pl. számítógépes monitoron).

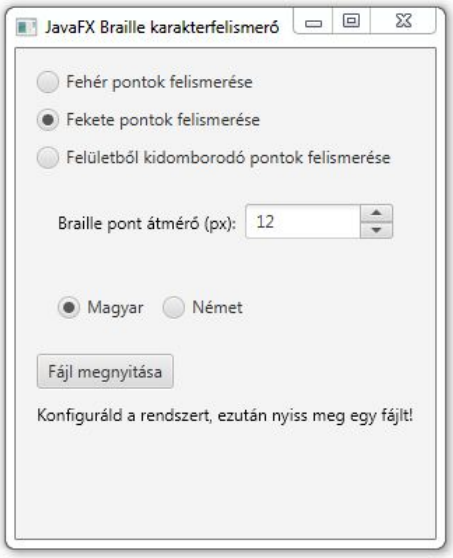
- A felismert ponthalmazok egymásra igazítása, a zajos részek szűrése: 80%, olyan képek esetén, ahol az egész háttér egy papírlap, így mindössze átlagos zajjal és árnyékokkal kell bajlódnunk, a ponthalmazok közel hibátlanok így igazíthatók.
- A ponthalmazok Braille-karakterekké szervezése: 80%, megfelelő sebességgel működik, néha enyhe instabilitás jelentkezik, mely valószínűleg az egyes modulok stabilitásának nem kielégítő tesztelésére vezethető vissza. Stabilabb egységtesztek használata esetén számos hiba kiküszöbölhető, az elméletem alapvetően helyes és használható, az algoritmus pedig alkalmas ennek megvalósítására.
- A Braille-karakterek szöveggé alakítása: 80%, technikai problémák nem jelentkeztek, egyszerűen kivitelezhető, a tökéletes működés eléréséhez természetesen még mélyebbre kellene ásnom a Braille-szövegek rövidítései területén, illetve érdemes lenne a helyesírás-javító modulok felhasználásán is elgondolkodni (hibajavítási céllal).
- A szöveg felolvasása: 100%, a Google TTS motorja kiválóan használható, megfelelően kommunikál a szoftverünkkel, további intézkedés nem szükséges.

Összességében tehát az egyes modulokkal kapcsolatban nem merült fel olyan probléma, melyre ne lenne megoldás, pusztán olyan, aminek minden igényt kielégítő megoldása túlnyúlna a rendelkezésre álló időkereten. A szoftver asztali PC-n kiválóan használható sebességet produkál, a karakterek kb. 95%-át sikeresen határozza meg. Természetesen mindez képfüggő és ebben az esetben, mivel a bemenet képjavító funkciója még nem képes önmagát adaptív módon paraméterezni, a paramétereket manuálisan állítottam be, így az eredményhez tetszőleges képek (lámpafénynél, tükröződő felületen... stb.) felhasználása esetén még szükséges az emberi beavatkozás. Ezt kiküszöbölendő mellékeltem olyan példákat is, ahol a Braille-pontok egyszerű két dimenziós pontok, melyeket akár a monitoron is megjeleníthetünk, de ki is nyomtathatunk. Szoftverünk ezeket már nagyon könnyen, önmagától fel tudja ismerni.

Amennyiben célunk, hogy a szoftver gyorsabban működjön, úgy optimalizálnunk szükséges, valószínűleg át kell írni C++ nyelvre. Fontos megemlíteni, hogy okkal nem optimalizáltam a kódot már az elején, ez ugyanis nagyon lelassítja, megnehezíti a fejlesztést. Először tehát érdemes egy közepes sebességű, de megfelelően működő szoftvert fejleszteni (és például eldönteni, hogy DBSCAN vagy k-means klaszterezési eljárás felel meg inkább az adathalmazunknak), majd ezután megállapítani, hogy melyek a kritikusan lassú modulok.

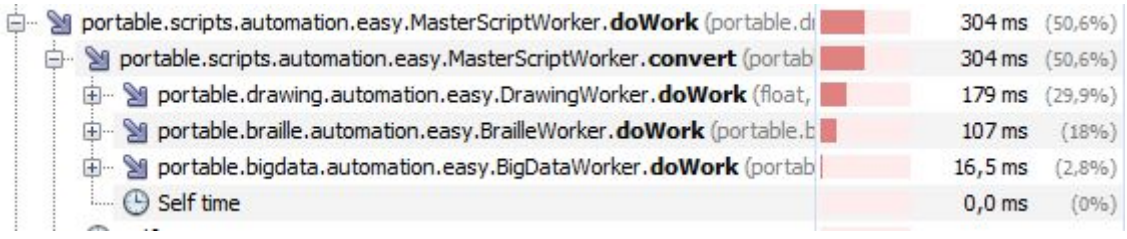
8.2 A könyvtár tesztelése az asztali számítógépen

Tekintve, hogy az Android telefonon a könyvtár használata a PC-hez képest nehézkesebb és lassabb is, készítettem egy segédprogramot, melynek segítségével előre elkészített tesztképeken is vizsgálható az algoritmus teljesítménye.



32. ábra. Felhasználói felület az asztali számítógépen

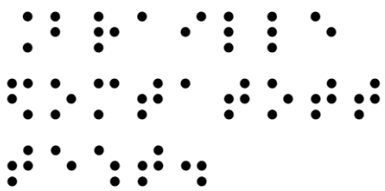
A 32. ábrán látható felhasználói felület a számítógépen történő izolált teszteléshez használható, segítségével felismerhetőek különböző képfájlokon található Braille-szövegek. Ez ugyanaz a modul, melyet a telefonos app is használ, mindössze a felhasználói felület más. Természetesen ez a felület is leegyszerűsített, nem kíván olyan funkcionalitást hozzáadni, mely a telefonon soha nem is volt tervben. Futási ideje asztali számítógépen a 33. ábrán látható, moduljaink összesen 304 ms-ot igényelnek (Core i7 4710HQ @ 2,50 GHz, 1 szál).



33. ábra. Futási idő asztali számítógépen

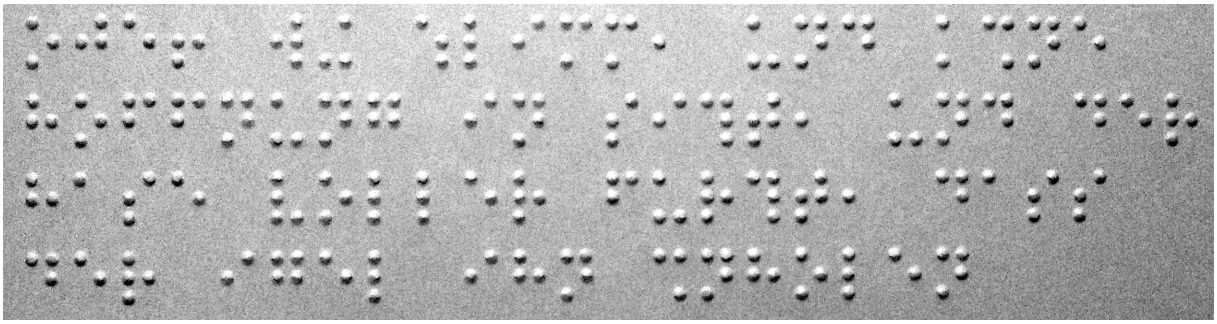
8.3 Néhány példa a felismerés hatékonyságára

Az alábbi képek digitális mellékletként is elérhetők, a szoftver úgyszintén, így az alábbi eredmények megismételhetőek. Fontos azonban figyelembe venni a szoftver nem determinisztikus elemeinek (pl. DBSCAN algoritmus) viselkedését, ez eltéréseket okozhat!
Első képünk a 34. ábrán látható, a teszt2_hu.bmp fájlhoz hasonlóan magyar nyelvű példa, mint látható, a különböző pontok számítógépes kijelzőn történő megjelenítéshez vannak optimalizálva, vagyis egyszerű fekete kitöltött körök.



34. ábra. teszt1_hu.bmp

Az alkalmazást az alábbi paraméterekkel kell konfigurálni: Fekete pontok felismerése, 12 pixel átmérő, magyar nyelv. A 12 pixel átmérőt normál esetben a kamera autofókusz távolsága és látószöge alapján trigonometrikus funkcióval határozzuk meg, számítógépes szimuláció esetén ez manuálisan történik.
Valós fényképet is előkészítettem, ezt a könnyebb használat miatt a szoftver által is alkalmazott, azonban automatikusan még nehezen konfigurált szűrőkkel feljavítottam. A kép a 35. ábrán látható, a hozzá tartozó beállítások: felületből kidomborodó pontok felismerése, 26 pixel átmérő, német nyelv (a kép saját készítésű, *A nyúl és a sün* c. mese egy részlete).



35. ábra. teszt3_fenykep_de.bmp

8.4 Konklúzió

Mint azt az előző fejezetben is láttuk, szoftverünk kiválóan alkalmas a Braille-írás felismerésére, vagyis a fő célunkat elértük. Sajnos az Androidos telefonon a felismerésre néhány másodpercet még várni kell, vagyis a felhasználó nem látja valós időben az eredményt. Összehasonlítva a hasonló szövegfelismerő appok (pl. Google Translate) teljesítményével azonban meg kell állapítanunk, hogy ez nem minősül rossz eredménynek, a többi termék sem képes nagyobb sebességre. Érdemes összegyűjteni, mely módosítások árán lehet elérni, hogy Androidos telefonon még gyorsabban fusson a rendszer:

- Az objektumorientált programozás előnyös struktúráinak (interfészek, absztrakt osztályok) eltávolítása
- A dinamikus memóriakezelés (new operátor, garbage collector) megszüntetése, statikus memóriaallokáció a program indításakor, saját memóriamenedzser modul készítése, az új objektumokat new operátor helyett *object pool*-ból szerezzük be.
- A teljes könyvtár átírása Java-ból C++ nyelvre.
- A többszálú futás hatékony megvalósítása (például 8 magra).

A fent említett változtatások több száz munkaórát igényelnének, különös tekintettel arra, hogy a sebességoptimalizálás során minden apró részletet figyelembe kell venni. Ilyen például az is, hogy a Gauss-szűrő milyen sorrendben olvassa ki a kép mátrixából az adott pixeleket, ugyanis ettől függ, hogy milyen hatékonysággal működik a processzor leggyorsabb gyorsítótára (L1 cache). Nyilvánvaló, hogy ezek a javítások lehetségesek, azonban véleményem szerint a mi esetünkben túl sok időt vettek volna el a szakdolgozat lényegi részétől, vagyis egy olyan algoritmus megalkotásától, amely viszonylag gyorsan (számítógépen 1 másodperc után) eredményeket produkál, ezzel nem marad le számottevően a kereskedelmi OCR programok sebességétől, s amely az eredményeket tekintve viszonylag jól közelíti a valós állapotot, vagyis jó hibaaránnyal dolgozik.

Ahhoz, hogy a fejlesztés során flexibilis maradhasson a program, célszerű a magasabb szintű nyelvek struktúráit használni, valamint úgy megírni a programot, hogy jól párhuzamosítható legyen. Maga a párhuzamosítás és az optimalizálás ugyan fontos elemek, az idő korlátozott volta miatt azonban túlzott használatuktól eltekinthetünk.

Kijelenthetjük, hogy a legfontosabb célunkat elértük, bebizonyítottuk, hogy az algoritmus működése stabil, a kívánt sebességre való gyorsításhoz szerkezeti változtatások nem szükségesek, vagyis a futási idő nem ütközik olyan elméleti alsó határba, mely lehetetlenné tenné az Androidos telefonon való használatot.

Irodalomjegyzék

[1] Kele Tímea: *Közel 33 ezer vak ember él az országban, a látássérültek száma pedig meghaladja a 218 ezret* *http://semmelweis.hu/hirek/2016/05/12/kozel-33-ezer-vak-ember-el-az-orszagban-a-latasserultek-szama-pedig-meghaladja-a-218-ezret/ letöltés ideje: 2017. nov. 16.*

[2] Rupert R A Bourne et. al.: *Magnitude, temporal trends, and projections of the global review and meta-analysis* (2017)

[3] John Wittenborn, David Rein: *Cost of vision problems: The Economic Burden of Vision Loss and Eye Disorders in the United States* (2013)

[4] Jiménez J. et. al.: *Biography of Louis Braille and invention of the Braille alphabet*. (2009)

[5] http*://old.mta.hu/mta_hirei/louis-braille-halalanak-evforduloja-90981/?style=normal*, letöltés ideje: 2017. nov. 17

[6] *https://de.wikipedia.org/wiki/Brailleschrift#/media/ File:A_person_reading_a_braille_book.jpg* , letöltés ideje: 2017. nov. 17

[7] *https://de.wikipedia.org/wiki/Computerbraille#/media/ File:Refreshable_Braille_display.jpg* , letöltés ideje: 2017 nov 17

[8] The Library of Congress National Library Service for the Blind and Physically Handicapped: *Specification #800, Braille Books and Pamphlets* (2008)

[9] Sass Bálint: A magyar Braille-rövidírás megújítása félautomatikus módszerrel (2014)

[10] *https://developer.android.com/topic/performance/memory.html* letöltés ideje: 2017. nov. 20

[11] https://developer.android.com/ndk/guides/index.html letöltés ideje: 2017. nov. 20

[12] Bernd Bruegge: Object Oriented Software Engineering Using UML, Patterns, and Java: International Version (2009)

[13] Peter Shirley et. al.: Fundamentals of Computer Graphics (2002)

[14] Fábián István: *Spektrofotometria* (2009)

[15] Ze-Nian Li, Mark S. Drew: Fundamentals of Multimedia (2003)

[16] Ian T. Young et. al.: *Fundamentals of Image Processing* (2007)

[17] Richard Szeliski: *Computer Vision: Algorithms and Applications* (2010)

[18] Wilhelm Burger, Mark J. Burge: *Digital Image Processing* (2016)

[19] http://sipi.usc.edu/database/database.php?volume=misc&image=12#top, letöltés ideje: 2017. nov. 22

[20] Hajnalka Ábrahám et. al.: *Neural regulation of human life processes – from the neuron to the behaviour. Interdisciplinary teaching material concerning the structure, function and clinical aspects of the nervous system for students of medicine, health and life sciences in Hungary* (2016)

[21] *https://en.wikipedia.org/wiki/Photoreceptor_cell#/media/File:1416_Color_Sensitivity.jpg* , letöltés ideje: 2017. nov. 21

[22] *https://en.wikipedia.org/wiki/Tamil_Braille#/media/File:Tamil_braille_sample.png* , letöltés ideje: 2017. okt. 30.

[23] *http://cg.iit.bme.hu/portal/sites/default/files/oktatott-targyak/3d-geometria-mernoki-visszafejtes/slides/3dgeom-09-v33.pdf ,* letöltés ideje: 2018. márc. 16

[24] Andreas Pålsson: ART vs. NDK vs. GPU acceleration: A study of performance of image processing algorithms on Android (2017)

[25] *http://www.math.tau.ac.il/~dcor/Graphics/cg-slides/flood-fill03.pdf* , letöltés ideje: 2018. ápr. 07

[26] Martin Ester et. al.: *A density-based algorithm for discovering clusters in large spatial databases with noise*. (1996)