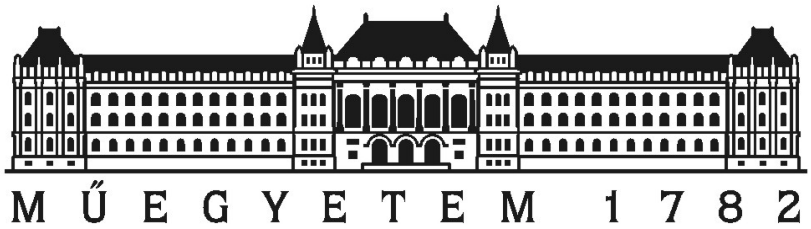


Önálló laboratórium beszámoló

Távközlési és Médiainformatikai Tanszék

Készítette:	**Kriván Bálint**
Neptun-kód:	**CBVOEN**
Ágazat:	**Hálózatok és szolgáltatások**
E-mail cím:	**balint@krivan.hu**
Konzulens:	**Kovács Gábor**
E-mail cím:	**kovacsg@tmit.bme.hu**

Téma címe: Kép- és videofeldolgozás

Feladat:
„Lyukkamera” modell elméletének és az OpenCV megismerése. Kamerarendszer szinkronizációjának megvalósítása.

Tanév: 2013/14. tanév, 1. félév

1. A laboratóriumi munka környezetének ismertetése, a munka előzményei és kiindulási állapota

1.1. Bevezetés/elméleti összefoglaló

1.1.1. Video- és képfeldolgozás, gépi látás

A képfeldolgozás a jelfeldolgozás egyik olyan esete, amikor a bemenet egy kép, a kimenet pedig egy másik kép, vagy a kép valamely paraméterei, karakterisztikái. Általános esetben a képet egy kétdimenziós jelnek tekintjük és ezen sztenderd jelfeldolgozási technikákat alkalmazunk [13].

A *számítógépes grafika* és a *gépi látás* a képfeldolgozáshoz nagyon közel álló szakterület. Számítógépes grafika esetén egy 3D-s világ modellje alapján hozunk létre 2D-s képeket, melyek a világ egy reprezentációjául szolgálnak, ahelyett hogy ezeket valamilyen leképező eszközzel (kamera) hoznánk létre. A gépi látás a videó- és képfeldolgozás egy magasabb szintje, amikor a képekből vagy képfolyamokból próbálunk a képből fizikai tartalmi információkat kinyerni.

1.1.2. OpenCV

Napjainkban elég sok alkalmazáskönyvtár elérhető, ezek között akad fizetős, iparban használt, pl.: Matrox Imaging Library [5], Adaptive Vision Library [12], de létezik nyílt forráskódú, ingyenes elérhető megoldás is: OpenCV [2], Camellia Image Processing and Computer Vision library [7]. A legelterjedtebb és legkiforrottabb eszköz amely szabadon felhasználható az OpenCV.

Az OpenCV számos platformon elérhető, és annak ellenére, hogy C++-ban íródott, elérhetőek programozási felületek egyéb nyelvekre is, pl.: Java és Python. A teljesítményszempontokat figyelembe véve én a C++ API-t választottam, mert bármely egyéb megoldásnál számolni kell a nem natív megközelítésből adódó teljesítménycsökkenéssel.

Az online elérhető dokumentációja [10] igen jól strukturált, valamint különböző fórumokon [1] is szép számmal jelen vannak a témában elmélyült szakemberek. A fejlesztői és felhasználói bázisa, valamint az akadémiai körökben való elterjedtsége miatt kézenfekvő volt, hogy ezt használjam a munkám során.

1.1.3. Kamera modell

Napjainkban az olcsó kamerák igen könnyen elérhetőek, szinte mindegyik laptopban integrálva van egy webkamera, de ugyanígy már alig kapható mobiltelefon beépített kamera nélkül. Azonban az olcsóság ára a jelentős torzítás. Az OpenCV az úgynevezett „lyukkamera” modellt használja, ahol a 3D-s pontok egy perspektivikus transzformációval képződnek le a képsík pontjaira.

Egy (X, Y, Z) világ-koordinátájú pont az (u, v) pontba képződik le [8] [6, 2.2. fejezet] a következő egyenlet szerint:

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}}_{\mathbf{A}} \underbrace{\left[\begin{array}{ccc|c} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{array}\right]}_{\mathbf{R}} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

ahol:

- s a homogén skálázási tényező ($s = Z$)
- $\mathbf{A}$ a kamera-mátrix (*belső paraméterek mátrixa*)
- (c_x, c_y) egy főpont, amely általában a kép közepe (optikai tengely és a képsík metszéspontja)
- (f_x, f_y) pedig a fókusztávolságok pixelekben kifejezve
- $\mathbf{R}$ a forgatási mátrix és $\mathbf{t} = \begin{pmatrix} t_1 & t_2 & t_3 \end{pmatrix}^T$ az eltolás-vektor – együttesen $\left[\mathbf{R} \,|\, \mathbf{t}\right]$ a forgatás-eltolás mátrix, mely a *külső paraméterek mátrixa*.

Fontos megjegyezni, hogy a valódi lencséknek van radiális („halszem effektusként" megjelenő) és enyhe tangenciális (abból adódóan, hogy a lencse és a képalkotó sík nem párhuzamos) torzításuk is [9]. Ezek a paraméterek adott kamerát tekintve konstansok, így kalibrációval meghatározhatóak, valamint korrigálhatóak. A radiális tényezőhöz az alábbi formulát használhatjuk:

$$x_{\text{jav}} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

$$y_{\text{jav}} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

ahol (x, y) volt az eredeti kép egy pixelének koordinátája, és $(x_{\text{jav}}, y_{\text{jav}})$ pedig az új koordináta a korrigált képen. Tangenciális torzítás javítása:

$$x_{\text{jav}} = x + \Big(2p_1 xy + p_2(r^2 + 2x^2)\Big)$$

$$y_{\text{jav}} = y + \Big(p_1(r^2 + 2y^2) + 2p_2 xy\Big)$$

A fentieket az OpenCV-ben egy 5 elemű sorvektor (torzítási együtthatók) reprezentálja:

$$\mathbf{d} = \begin{pmatrix} k_1 & k_2 & p_1 & p_2 & k_3 \end{pmatrix}$$

1.2. A munka állapota, készültségi foka a félév elején

A feladat frissen lett kitűzve, előttem nem dolgozott rajta senki. Az első feladataim közé tartozott az elméleti alapok megismerése, valamint az OpenCV keretrendszerrel történő tapasztalatszerzés. Az elején megfogalmazott feladatkitűzés több féléven átível, az első félévre kitűzött feladat – az előbbieken túl – kamerák (sztereó)kalibrációja valamint szinkronizációja volt.

2. Az elvégzett munka és az eredmények ismertetése

2.1. OpenCV fordítása

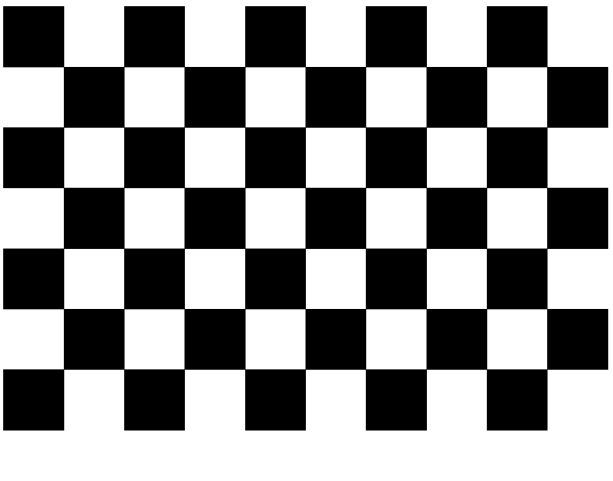
Az OpenCV függvénykönyvtár csak teljes fordítás után volt működőképes, mivel az előre fordított könyvtárak nem bizonyultak platformfüggetlennek. A CMake [3] segítségével történő MinGW makefile-ok generálása után a `mingw32-make` felhasználásával lefordult a csomag. A PATH-ra helyezve az elkészült DLL-eket, a lefordított alkalmazások elérik azokat, így nincs szükség statikus linkelésre, valamint a DLL-ek folytonos másolására a futtatáshoz.

2.2. Kamera kalibráció

Az előzőekben említett belső (kamera-mátrix, torzítási együtthatók) és külső paraméterek meghatározásához szükség van a kamera kalibrációjához. A munkám során a laptopom beépített webkameráját használtam, így ennek a kalibrálására volt szükség.

A mátrixok meghatározása geometriai egyenlőségek alapján történik, melyek függnek a választott kalibrációs objektumoktól, az OpenCV 3 félét támogat:

- Fekete-fehér sakktábla (lásd 1. ábra)
- Szimmetrikus kör minta
- Asszimetrikus kör minta



1. ábra. Fekete-fehér sakktábla minta

A feladat, hogy különböző állásban lévő mintákat kell felismertetni az OpenCV-vel, és ezekből egyenletrendszer megoldásával történik a paraméterek meghatározása. Az OpenCV-ben a `findChessboardCorners` nevű eljárással tudunk egy képen

adott méretű sakktáblát megtalálni. A függvénnyel kinyerhető, hogy a képen hol vannak a sakktábla belső sarkai (ahol a fekete négyzetek érintik egymást – egy tradicionális 8×8-as sakktáblának 7×7 belső sarka van). Visszatérési értékével jelzi azt is, hogy sikeresen megtalálta-e a mintát. A detektált sarkok csak közelítőek, ezért meghívódik a `cornerSubPix` eljárás is, hogy a pozíciók pontosabbak legyenek, ennek eltérő paraméterezése tovább pontosíthatja a koordinátákat.

Kellő számú minta gyűjtése után (sakktábla esetén kb. 10 elegendő lehet [9]) a feladat, hogy a talált képpontokat megfeleltessük a kalibrációs minta objektumpontjaival. Ezen megfeleltetések alapján már megkaphatóak a szükséges paraméterek az OpenCV `calibrateCamera` eljárásának köszönhetően.

Sikeres kalibráció után a kimeneti paramétereket (kamera-mátrix és torzítási együtthatók) fájlba menthetjük és később azokat felolvashatjuk, így egy adott kamerához elég egyszer elvégezni a fenti műveleteket. A paraméterek felhasználásával az `undistort` függvény egy adott képet torzítás mentessé transzformál.

Az OpenCV programcsomagban talált példaprogram (`samples/cpp/tutorial_code/calib3d/camera_calibration/`) apró finomítása és a konfigurációs fájlok felparaméterezése után a kamerát sikerült bekalibrálni, valamint egy egyszerű programvázat írtam a kalibrációs eredmények felolvasásához, hiszen a későbbi feladatok során erre szükség lesz.

2.3. Kamera szinkronizáció - lézerpont detekció

A következő félév során megvalósítandó feladatnál egy n darab kamerából álló kamerarendszerünk lesz. Feltesszük, hogy ezen kamerák nem állnak közvetlen kapcsolatban (például USB-n keresztül) a vezérlő számítógéppel, így szükséges ezen adatfolyamok szinkronizációja.

A szinkronizációhoz szükség van egy trigger-eseményre, melyet minden kamera képén észlelve találhatunk egy közös időpillanatot. Ezen eseményt egy lézerpont felvillanásának választottam, hiszen való életben történő megvalósítása igen egyszerű (lézerpointerek könnyedén elérhetőek), valamint ennek detekciója elméleti síkon könnyűnek tűnt – később látni fogjuk, hogy akadtak gyakorlati problémák.

2.3.1. 1. módszer - Legvilágosabb pont detekció

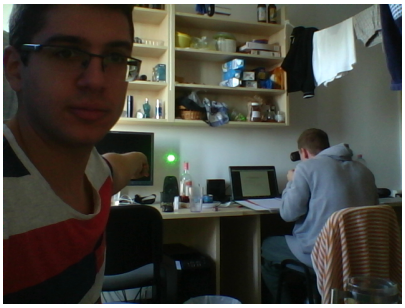
Elsőként [4]-ben ismertetett eljárást implementáltam. Lényege, hogy egy szürkeárnyalatos képen az (x, y) pontot akkor tekinti világos pontnak, ha teljesül, hogy:

$$\mathrm{DN}(x,y) > \mathrm{AVE}\left\{\,\mathrm{DN}(k,l)\right\} + \Delta \quad \text{és} \quad \mathrm{DN}(x,y) > \mathrm{MED}\left\{\,\mathrm{DN}(k,l)\right\} + \Delta$$

ahol DN jelöli egy pont intenzitását, AVE és MED operátorok pedig a súlyozott átlag és medián értékeket. A (k, l) pontok az (x, y)-ra illeszkedő átlón vannak rajta, gyakor-

latban ez egy 7-pixel széles ablak (középen az (x, y) pont), mégpedig a $(0, 0, 1, 0, 1, 0, 0)$ súlyokkal. Tehát lényegében az $(x - 1, y - 1)$ és $(x + 1, y + 1)$ pontokat vizsgáljuk.

Ahhoz, hogy ezt alkalmazni lehessen, a bementi képünket (2. ábra) akkora felbontásra kell kicsinyítenünk, hogy a lézerpont kb. 1 pixelnyi legyen. Az eredményt lásd 3. ábra (96×72 felbontású kép – kicsit felnagyítva a dokumentumban). A jobb oldali fekete-fehér képen a fehér pontok jelentik a „talált" lézerpontokat.



2. ábra. Eredeti kép

3. ábra. Legvilágosabb pont detekció

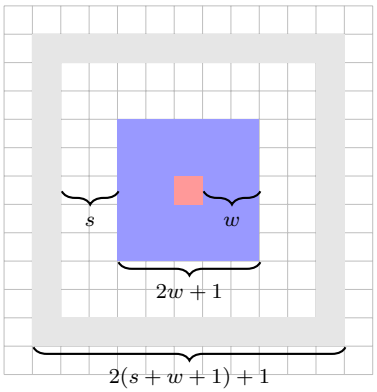
A fenti találatokhoz $\Delta = 100$-t választottam. Látható, hogy a képen megtalálta a lézerpontot, de sajnos egyéb pontokat is: azon világos pontokat, ahol a környezet sötét – ilyenek pl. a becsillanások az üvegen és fényvisszaverő felületek. Δ értékét növelve nem találta meg a lézerpontot, ellenben a többi találatból sok megmaradt.

2.3.2. 2. módszer - Világos négyzet keresés

Következő ötletként nem egy pixelnyi környezetet nézünk, hanem a vizsgált pixel egy kiterjedt környezetét; egy (x, y) pontot akkor tekintünk detektált pontnak, ha teljesül a következő:

$$\mathrm{rect}\Big(x, y, 2w + 1\Big) \quad > \quad \mathrm{frame}\Big(x, y, 2(w + s + 1) + 1\Big) + \Delta$$

ahol $\mathrm{rect}(x, y, w)$ egy (x, y) középpontú w oldalhosszúságú négyzet pixeljeinek, $\mathrm{frame}(x, y, w)$ pedig egy (x, y) középpontú w oldalhosszúságú négyzet kerületén elhelyezkedő pixelek átlagintenzitását jelöli. Ezt a módszert illusztrálja a 4. ábra.



4. ábra. Piros - vizsgált pont; kék négyzetbe eső pixelek és a szürke kereten lévő pixelek intenzitás-átlagának vizsgálata

Az algoritmusnak tehát az s, w és Δ paraméterei, ahol s-t és w-t meghatározza a lézerpont távolsága a kamerától, illetve a lézerpont kiterjedése (lézerpointer tulajdonsága), Δ pedig empirikusan választott (kamera és környezet függő).

Az 5. ábrán látható az algoritmus eredménye a referenciaképen. Megfigyelhető, hogy kevesebb a detektált pont, mint az előző algoritmusnál, és továbbra is megtalálja a tényleges lézerpontot, de ugyanúgy a nagyobb kiterjedésű világos pontokat is – pl.: a két tejesdoboz teteje.

5. ábra. Világos négyzet keresés

2.3.3. 3. módszer - Kör minta illesztés, aurakereséssel

Ennek a módszernek az alapötlete, hogy négyzet helyett körlap maszkot alkalmazunk, hiszen a minta amit keresünk kör alakú, valamint vegyük számításba azt is, hogy – zöld színű lézer esetén – a lézerpont környezete (aurája) nem ég ki a képen, hanem zöldes színű, és ezt is belevéve az alkalmazásba kiszűrhetjük a pusztán „világos” (a környezetéhez képest) kiterjedt részeket.

A feladat megoldásához 3 maszkra van szükség: egy belső kör maszk, ami a lézer-

pont belsejét kell lefedje, egy auramaszk (kicsit nagyobb kör maszk), ami nem csak a teljesen kiégett pontokat tartalmazza, hanem a zöldes pixeleket is, valamint egy „környezet maszk", amit úgy kapunk hogy egy auramaszknál nagyobb négyzetből kivágjuk azt. Egy (x, y) pontot ekkor a következők teljesülése esetén tekintünk detektált pontnak:

$$\begin{aligned} \mathrm{mask}_{\text{belső}}(x,y) &> \theta && (1)\\ \mathrm{mask}_{\text{belső}}(x,y) &> \mathrm{mask}_{\text{környezet}}(x,y) + \Delta && (2)\\ \mathrm{mask}^*_{\text{aura}}(x,y) &\approx \text{zöldes színű} && (3) \end{aligned}$$

ahol $\mathrm{mask}_m(x, y)$ az (x, y) középpontú m maszk által kijelölt pixelek átlagintenzitása, $\mathrm{mask}^*_m(x, y)$ pedig a maszk által jelölt pixelek „átlagszíne" (csatornánként vett átlagok).

Az első egyenlőtlenség szerepe, hogy a belső rész átlagintenzitása egy küszöbértéknél nagyobb – egy adott értéknél világosabb – legyen. A második, hogy egy olyan körlapot keresünk, ahol a belső rész egy adott Δ-val világosabb, mint a környezete. A harmadik pedig azt biztosítja, hogy maga a világos folt széle zöldes színű legyen. Ha ezen három feltétel teljesül, akkor nagy valószínűséggel a lézerpontot találtuk meg.

Az aura által meghatározott átlagszínt az algoritmus akkor tekinti „zöldes színnek", ha

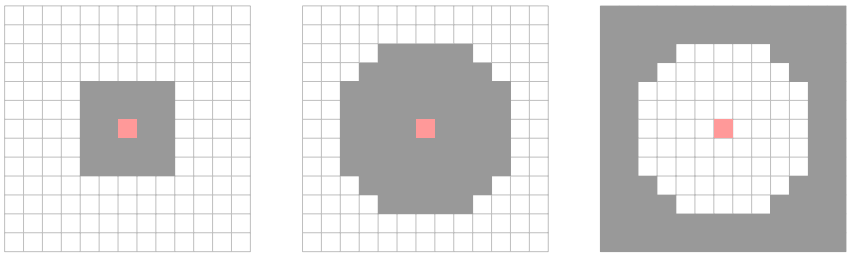
$$\frac{R+B}{2} + \delta < G$$

ahol R, G és B a szín RGB színtérnek megfelelő piros, zöld és kék csatornákon mért intenzitás. Azért döntöttem a fenti mellett, mert az RGB színtérben az euklideszi távolság nem korrelál jól a szemmel érzékelhető színkülönbséggel, másrészt a referencia színt is nehéz kiválasztani, mert az erősen megvilágítás függő.

Jól látható, hogy ennek az algoritmusnak több paramétere is van:

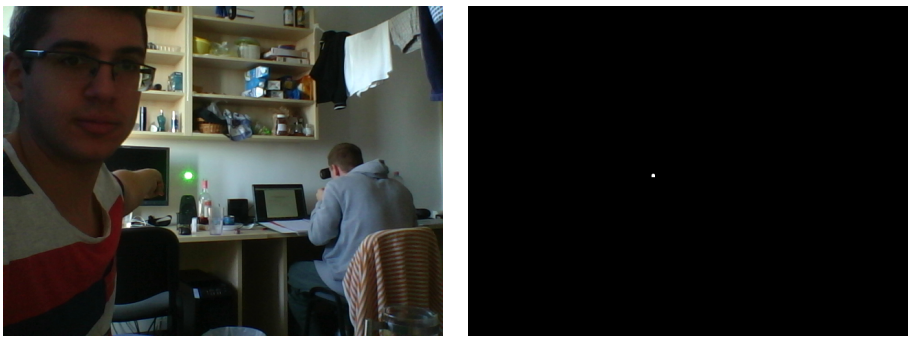
- maszkok mérete: belső kör sugara, aura sugara, illetve a környezeti maszk mérete. Felépítésért lásd 6. ábra.
- θ: küszöbérték a belső rész átlagintenzitásának, jelentése: „elég világos legyen a belseje"
- Δ: a környezet átlagintenzitása legalább ennyivel legyen kevesebb, mint a belső részé.
- δ: az aura átlagszínének zöld csatornáján mért érték ennyivel legyen legalább nagyobb, mint a másik két csatornán mért átlag.

Az algoritmus eredménye a 7. ábrán látható: az algoritmus már csak a lézerpontot detektálta, egyéb fényesebb részeket nem, amit az aura-vizsgálatnak tudhatunk be.



(a) belső (b) aura (c) külső

6. ábra. A három különböző maszk az algoritmusnál egy adott középpontra illesztve

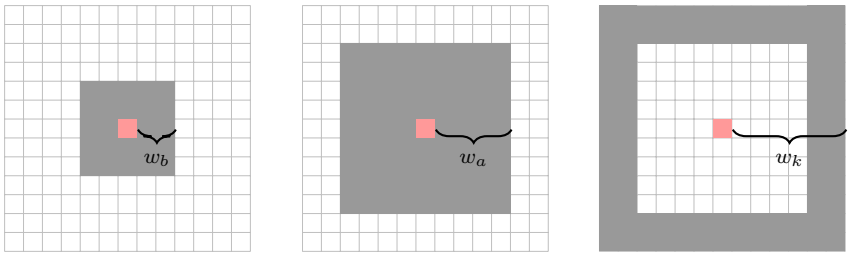


7. ábra. Kör minta illesztés, aurakereséssel

2.3.4. 4. módszer - Négyzet illesztés, aurakereséssel

Az utolsó vizsgált algoritmus az előző kettő keveréke. A nem négyzet alakú maszkok használatából adódó sebességcsökkenést kiküszöböli, de az előzőekben bevezetett auravizsgálat segítségével kiszűri a 2. módszernél tapasztalt hamis detekciót.

Az algoritmus megegyezik az előző módszernél használttal, pusztán a maszkok alakjai különböznek (8. ábra).



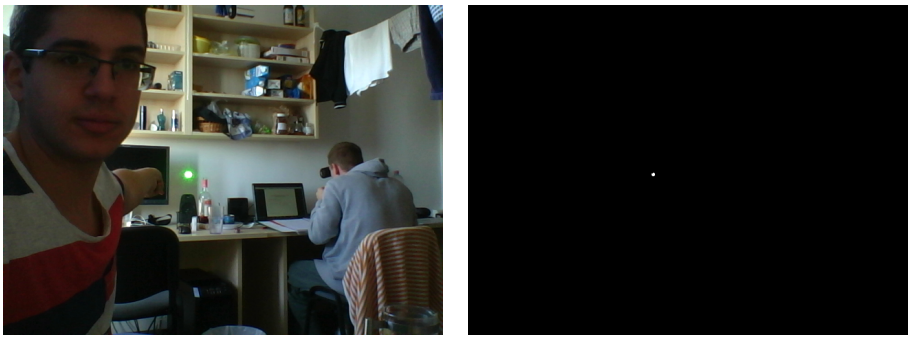
(a) belső (b) aura (c) külső

8. ábra. A három különböző maszk a 4. módszerhez

Fontos megjegyezni, hogy itt is számít a maszkok tényleges mérete, arra kell törekedni, hogy a belső maszk legyen beleírható a lézerpont kiégett részébe, az aura pedig tartalmazza a színes részeket és lehetőleg csak kicsit lógjon túl azon (hogy az átlagszínt

ne nagyon befolyásolja a környezet színe).

A paraméterek megegyeznek az előző algoritmusnál vázoltnál, hiszen a maszkok alakján kívül minden egyezik, a maszkok paraméterei, pedig az ábrán szerepelnek. Az algoritmus eredménye a 9. ábrán látható: itt is csak valódi detekció történt, hamis nem.



9. ábra. Négyzet illesztés, aurakereséssel

2.3.5. Sebesség összehasonlítás

Az előzőek alapján kijelenthetjük, hogy eredmény szempontjából az utolsó két algoritmus szerepelt a legjobban, de sebességben jelentősen alul maradnak társaiknál. Különböző konfigurációkban megmérve az algoritmusok futási idejét az 1. táblázatban gyűjtöttem össze a tapasztalatokat.

Algoritmus	**Detekció (ms)**	**Megjelenítés (fps)**
1. (álló 96×72)	1,2	–
1. (stream $640 \times 480 \rightarrow 96 \times 72$)	2	9
2. (stream 640×480)	290	3.4
3. (stream 640×480)	4050	0.24
3. (stream $640 \times 480 \rightarrow 320 \times 240$)	310	3
4. (stream 640×480)	900	1.1
4. (stream $640 \times 480 \rightarrow 320 \times 240$)	65	8

1. táblázat. Egy frame feldolgozásához szükséges átlag idő (5 futtatást mérve)

A 3. és 4. algoritmust egy negyedakkora képen is kipróbáltam (futás időben történő átméretezéssel), természetesen a megfelelő paraméterek igazításával, valamint az első algoritmus jellegéből adódóan mindenképpen a VGA felbontású kép egy átméretezett változatán kellett futtatni. A Megjelenítés (fps) oszlopban szereplő értékeket nem pusztán a detekció sebessége határozza meg, hanem az egyéb műveletek sebessége (kamera képének beolvasása, feldolgozás stb.), valamint azt is figyelembe kell venni, hogy a keretrendszer a képek megjelenítése után 10ms-ig várakozik egy billentyűkombinációra, hogy egy lehetséges billentyűleütést is kezelni tudjunk (pl. maszkok képre rajzolása, ESC – kilépés).

Az nyilvánvaló, hogy a bementi kép és a maszkok mérete jelentősen befolyásolja az algoritmusok sebességét. A 3. és 4. algoritmus közti különbséget az adja, hogy a 4-nél felhasználható az, hogy a belső és aura maszkok ismert méretű négyzetek, így olyan pixeleket nem is kell vizsgálni, amik nem tartoznak bele, míg a 3-nál az egész maszkot végig kell nézni, hogy tudjuk, mely pixeleket kell számításba venni.

Az jól látható, hogy eredeti felbontásban a jó eredményt adó algoritmusok túl lassúak, így a következő lehetőségeink vannak:

- az algoritmusok megírása GPU-ra, amitől azt várjuk, hogy jelentős sebességnövekedés érhető el
- on-the-fly konverzió kisebb felbontásra, amin már gyorsabban fut, de még mindig helyes detekció történik
- a kamerák képein előre definiált részen várjuk a lézerpont felvillanását, így a lézerpontot nem kell megkeresni, csak adott területen észlelni a megjelenését

2.4. Összefoglalás

A félév során elért eredmények összefoglalva:

- Elolvastam számos API dokumentációt és témához kapcsolódó online cikket, leírást
- Készítettem körülbelül 900 sor C++-kódot
- Megismertem a „lyukkamera” képalkotási modellt
- Elkezdtem használni az OpenCV alkalmazáskönyvtárat, megismertem az alapvető adatszerkezeteit, valamint a kamerakalibrációval kapcsolatos eljárásokat, a mögöttük lévő elméletet

3. Irodalom, és csatlakozó dokumentumok jegyzéke

Irodalomjegyzék

[1] Stack Exchange Inc. *Newest 'opencv' Questions - Stack Overflow*. 2013. URL: http://stackoverflow.com/questions/tagged/opencv.

[2] Itseez. *OpenCV | OpenCV*. [Online; hozzáférve 2013. november 26-án]. 2013. URL: http://opencv.org/.

[3] Kitware. *CMake - Cross Platform Make*. 2013. URL: http://www.cmake.org/.

[4] Karen Levay. *4.1 Bright-Spot Detection*. 1997. URL: http://archive.stsci.edu/iue/manual/newsips/node22.html.

[5] Matrox. *Machine vision and imaging software - Matrox Imaging Library (MIL)*. [Online; hozzáférve 2013. december 1-jén]. 2013. URL: http://www.matrox.com/imaging/en/products/software/mil/.

[6] Yannick Morvan. "Acquisition, Compression and Rendering of Depth and Texture for Multi-View Video". Ph.D. Thesis. Technische Universiteit Eindhoven, 2009.

[7] École des mines de Paris (ENSMP). *Camellia Image Processing and Computer Vision library*. [Online; hozzáférve 2013. december 1-jén]. 2013. URL: http://camellia.sourceforge.net/.

[8] OpenCV dev team. *Camera Calibration and 3D Reconstruction – OpenCV 2.4.7.0 documentation*. 2013. URL: http://docs.opencv.org/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html.

[9] OpenCV dev team. *Camera calibration With OpenCV – OpenCV 2.4.7.0 documentation*. 2013. URL: http://docs.opencv.org/doc/tutorials/calib3d/camera_calibration/camera_calibration.html.

[10] OpenCV dev team. *OpenCV API Reference – OpenCV 2.4.7.0 documentation*. 2013. URL: http://docs.opencv.org/modules/refman.html.

[11] OpenCV dev team. *Using OpenCV with Eclipse (plugin CDT) – OpenCV 2.4.7.0 documentation*. 2013. URL: http://docs.opencv.org/doc/tutorials/introduction/linux_eclipse/linux_eclipse.html.

[12] Adaptive Vision. *Machine Vision C++ Library - Adaptive Vision*. [Online; hozzáférve 2013. december 1-jén]. 2013. URL: https://www.adaptive-vision.com/en/libraries/adaptive_vision_library/.

[13] Wikipedia. *Image processing — Wikipedia, The Free Encyclopedia*. [Online; hozzáférve 2013. december 1-jén]. 2013. URL: `http://en.wikipedia.org/w/index.php?title=Image_processing&oldid=580397063`.

3.1. A csatlakozó dokumentumok jegyzéke

Az elkészült programkód, illetve a felhasznált fájlok az alábbi GIT repóban érhető el, melyből publikus klón készíthető:

`https://github.com/messo/msc_onlab1`

A tároló Eclipse CDT projekteket tartalmaz, melyek az OpenCV header és library fájlok megadása után futtathatóak is. A konkrét beállításokért lásd [11].