M08 – Környezet feltérképezése sztereó kamerakép alapján

Mérési útmutató

Irányítástechnika és képfeldolgozás laboratórium 1.

Szemenyei Márton, Reizinger Patrik

Irányítástechnika és Informatika Tanszék

2018

Tartalomjegyzék

1. Objektumfelismerés, pozíciómeghatározás

A kamerarendszerek egyik legfontosabb felhasználási célja a környezetben található objektumok szétválasztása és a lényeges elemek kiemelése, más szóval szegmentálás. Tekintettel a fontosságára, illetve a feladat meglehetősen nem egzakt jellegére, számos jó és kevésbé jó algoritmust alkalmazhatunk. Legtöbbjük csak különleges körülmények között használható, vagy olyan mértékű futásidőigénye, hogy a legtöbb alkalmazásban szóba se kerülhet. Emiatt az alkalmazások nagy részénél valamilyen módon egyszerűsítenünk kell a környezetet, és olyan módon kell átalakítanunk, hogy az algoritmus számára könnyedén feldolgozható legyen. Erre megoldás, ha a megkeresendő objektum színét megváltoztatjuk olyan módon, hogy a környezettől elüssön.

A szegmentált objektumok megtalálása után annak releváns tulajdonságait is meg kell határoznunk. Szinte minden esetben szükséges a pozíció és/vagy orientáció, de gyakran van szükség méretre stb. A lényeges tulajdonságok kiemelése általában egyszerűbb feladat, mint a szegmentálás, de sokszor ezek meghatározása is szinte lehetetlennek tűnik. Az alábbiakban ismertetünk néhány fontosabb algoritmust, amely segítségével a mérésben szükséges képfeldolgozási feladatok elvégezhetők. Ez természetesen nem jelenti, hogy a hallgató nem használhat szofisztikáltabb megoldást.

1.1.RGB-D Kamerák

Az elmúlt néhány évben egyre inkább elterjedtek olyan speciális szenzorok, amelyek az egyes pixelek intenzitásai mellett azok a szenzortól számított távolságukat is képesek meghatározni, így minden egyes képponthoz egy negyedik számértéket is hozzárendelnek. Ezeket az eszközöket RGB-D, vagy mélység kameráknak nevezzük, ahol a D az angol depth, vagyis mélység szóból származik.

Ezeknek a kameráknak alapvetően két változata létezik: az elsőt sztereó kamerának vezettük, ahol két, egymástól fix távolságra lévő kamera van egy házba építve, és az egyes pixelek távolságát és két készített kép közötti megfeleltetésekből számolhatjuk ki. Ezeknek az eszközöknek a kalibrációja általában a gyártás során megtörténik, valamint a mélység számítása a kamerába épített feldolgozó hardveren megtörténik.

Ezzel szemben az infravörös technológiára alapuló mélység kamerák három elemből állnak: egy közönséges RBG érzékelőből, egy infravörös vetítőből és egy az infravörös tartományban működő érzékelőből. A működésük elve, hogy az ember számára láthatatlan infravörös tartományban egy előre meghatározott mintázatot vetítenek ki, amelyet az infravörös érzékelő visszaolvas, és a mintázat torzulásából következtet a látott kép térbeli struktúrájára. A kettő közül az infravörös alapú érzékelők elterjedtebbek, mivel jobb minőségű eredményeket adnak és kevesebb feldolgozást igényelnek. Hátrányuk, hogy a környezetben található egyéb infravörös források megzavarhatják az eredményeket.

ábra 1: A Kinect One RGB-D érzékelő (Forrás: wikimedia.org)

Az RGB-D kamerák által szolgáltatott mélység kép (amely önmagában egy egyszerű szürkeárnyalatos kép) felhasználható arra, hogy a térben egybetartozó objektumokat könnyedén a kamerától valós távolság alapján szegmentálhassunk. Természetesen egymáshoz térben közel lévő objektumok a mélység képen „összenőhetnek", így önmagában ez a módszer nem teljes mértékben megbízható.

1.2.Szín alapú detektálás

A számítógépes látás során gyakran használjuk ki a szürkeárnyalatos képekben rejlő intenzitás információn felül a színes képek által hordozott extra információt is. Számos egyszerű detektáló algoritmus épül színbeli hasonlóság alapú keresésre. Itt azonban számos problémába ütközhetünk: egyrészt ahhoz, hogy a színbeli hasonlóság alapú keresés megbízhatóan, robusztusan működjön, arra van szükség, hogy a színeket leíró pixel értékek segítségével könnyen ki tudjuk fejezni a színek hasonlóságát.

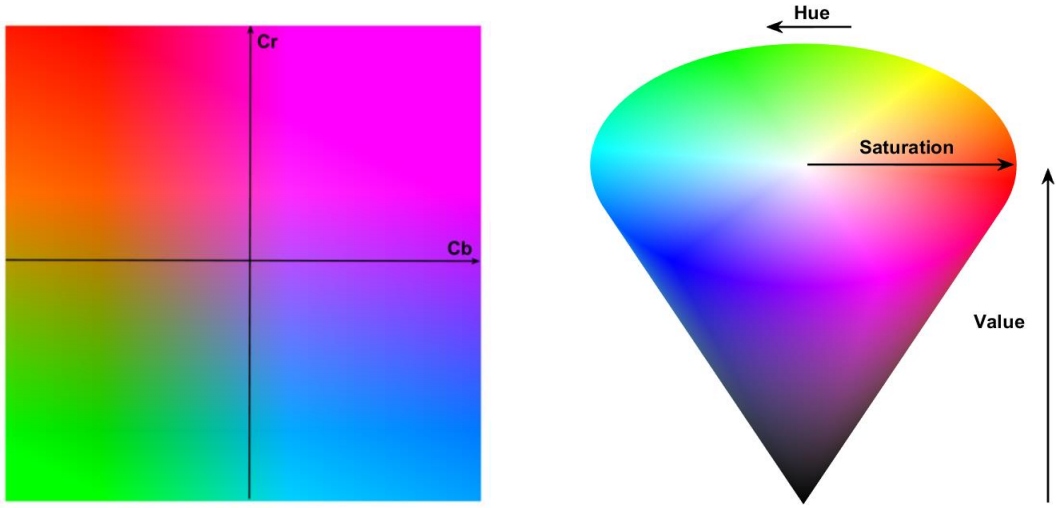
A kamerák által leggyakrabban használt színábrázolás (vagy más néven az RGB színtér) azonban erre nem alkalmas: két színt leíró pont geometriai távolsága az RGB színtérben nem kifejező arra nézve, hogy az emberi érzékelés mennyire érzi hasonlónak a két színt. Ezen felül amennyiben a megvilágítási viszonyok megváltoznak, az egy RGB kép esetén mind a három értéket megváltoztatja, pedig a képen található objektum színe nem változott.

A színtér transzformációs eljárások célja, hogy az RGB helyett egy olyan új színreprezentációt adjanak meg, amely információ elveszítése nélkül képes a színbeli hasonlóságot jól leírni, ezen felül pedig a megvilágítás változására is robusztus legyen. Ezek a transzformációk ezzel egyben egy új színteret is definiálnak.

Az egyik leggyakrabban használt színtér az YCbCr család, melynek több minimálisan különböző változata van. A színtér három csatornája közül az Y egy elkülönített világosság komponens, amely az adott szín fényességét reprezentálja. A másik két csatorna a Cr és a Cb pedig a szín árnyalatát kódolja el. Ezt a színteret gyakorta alkalmazzák digitális videó rendszerekben, valamint a JPEG és az MPEG kódolás során. A gyakorlatban az YCbCr

színteret gyakorta összekeverik az YUV színtérrel, amely hasonló elven működik, csak éppenséggel analóg rendszerekben használatos.

A másik, képfeldolgozás esetén gyakran használt család a HSV/HSI/HSL család. Ezen színterek közös jellemzője, hogy a szín információt két érték, a Hue (színárnylat) és a Saturation (telítettség) segítségével írják le, míg a reprezentációk közötti alapvető különbség a fényesség/intenzitás reprezentációjának módja. Ez a színtér könnyen ábrázolható egy henger vagy kúp formájában. A szín alapú feldolgozás esetén rendkívül gyakori mindkét alternatív színtér használata.



ábra 2: A YCbCr és a HVS színterek ábrázolása. (Forrás: saját készítés)

1.3.Bináris képek feldolgozása

Az eddigi diszkusszió során említésre került, hogy a képeknek több fajtája létezik, melyek közül a leggyakoribb fajták az egy csatornás szürkeárnyalatos, illetve a három csatornás színes képek. Szintén gyakran előfordulnak azonban kétállapotú, bináris képek például éldetektálás esetén, ahol az élekhez tartozó pixelek egyes értéket vettek fel, míg a többi nullást. Ilyen bináris képek azonban számos más művelet eredményeképp is előállhatnak, például küszöbözés, színdetektálás, vagy komplex objektumdetektáló eljárások során.

A jelenlegi alfejezetben olyan bináris képeket fogunk vizsgálni, melyeknek a két állapota közül az egyik „1” az adott alkalmazás szempontjából relevánsnak tekintett objektumainkat jelöli, míg a „0” az számunkra irreleváns hátteret reprezentálja. Fontos megjegyezni, hogy a megjelenítés során a láthatóság kedvéért a bináris képek kép állapotát a maximális fehér és a minimális fekete színek szokták jelölni, arra viszont nincs egyértelmű konvenció, hogy a két szín közül melyik jelöli az objektumot- és a hátteret. A jelen műben következetesen a fehér szín fogja az objektumot jelölni, de ez más forrásokban lehet fordítva is.

A gyakorlatban bármilyen kifinomult módszert is használunk az objektumok elkülönítésére, ez nem fog tökéletesen sikerülni, így – ahogy a színes és szürkeárnyalatos képek esetében is – szükség van a bináris képek javítására. Természetesen, mivel a bináris képek hibái más jellegűek, ezért az azok javítására használt módszerek is jelentősen eltérnek. A bináris képeknek alapvetően két jellemző hibája fordul elő: az egyik az olyan pixelek jelenléte,

amelyek a valóságban a háttérhez tartoznak, azonban mégis objektumként lettek címkézve, valamint ennek az ellentéte: a tévesen háttérként címkézett objektum pixelek. Az előbbi hibák miatt hamis objektumokat láthatunk a képen, vagy a valóságban elkülönülő objektumokat tévesen összenöveszthetünk. Az utóbbi miatt előfordulhat, hogy lukakat kapunk egyes objektumokon belül, vagy, hogy egy a valóságban egybefüggő objektumot tévesen szétválasztunk.

Erre a problémára gyakran alkalmazott megoldás az erózió, illetve a dilatáció művelete, valamint az ezek egymás után történő elvégzéséből adódó nyitás és zárás. Ezek a műveletek azonban az egyes objektumok alakját a hozzájuk használt strukturáló elem alakjának függvényében torzítják, valamint, amennyiben a zajok és lyukak meglehetősen nagy méretűek, akkor az elvégzésük is költséges. Létezik azonban egy egyszerű alternatíva, amely abban az esetben használható, ha csupán egyetlen objektumot keresünk, és nagy valószínűséggel tudjuk garantálni, hogy a képen ez lesz a legnagyobb.

Ha a bináris képen eróziót végzünk, majd az így kapott képet az eredetiből kivonjuk, akkor az egyes bináris objektumok határait, azaz kontúrjait tartalmazó képet kapunk. Amennyiben az így kapott kontúrok közül a legnagyobb területűt kiválasztjuk, akkor megkaphatjuk a keresett objektumot. Ennek az objektumnak egy rendkívül fontos leíró mennyisége a nyomaték, vagy idegen kifejezéssel a momentum. Nyomatékból számos rend létezik, melyek közül számunkra a nullad-, illetve az elsőrendű nyomatékok hasznosak. A nulladrendű nyomaték egész egyszerűen a pixel intenzitások összege. Mivel itt bináris képeket tárgyalunk, így az itteni objektumok nulladrendű nyomatéka azok területét fogja megadni. Az objektumok tömegközéppontját pedig meghatározhatjuk az első és a nulladrendű nyomatékok segítségével az alábbi képlet alapján:

$$x = \frac{M_x^1}{M_0},\ y = \frac{M_y^1}{M_0} \qquad M_x^1 = \sum x\, I(x,y),\ M_y^1 = \sum y\, I(x,y), \qquad M_0 = \sum I(x,y)$$

2. A mérés környezete

A mérés során a *PyCharm* elnevezésű IDE áll rendelkezésre, amely rendkívül sokoldalú szolgáltatásokkal könnyíti meg a szoftverfejlesztést, például konfigurálható automatikus formázási lehetőségek állnak rendelkezésünkre. További részletekért érdemes lehet a JetBrains ide vonatkozó weboldalát felkeresni (https://www.jetbrains.com/help/pycharm/quick-start-guide.html). Függvények, objektumok esetében a ***Ctrl+P*** billentyűkombináció pop-up segítségként szolgálva mutatja nekünk a paramétereket. A mérés során használt programnyelv a Python 3-as verziója lesz.

2.1. A Python nyelv

A Python programnyelv egy ***interpretált szkriptnyelv*** (a saját bytekódját hajtja végre), mely alapvetően az *objektumorientált* paradigma által vezérelve került kialakításra. Jelenleg kétféle, kismértékben eltérő főverziója érhető el, ezek közül a 3-as alkotja jelen mérés tárgyát.

Az OOP szemlélet olyannyira központi szerepet tölt be Pythonban, hogy minden változó objektumnak tekinthető – ami azt jelenti, hogy az integer és float típusok is objektumok. A nyelv számára jelentős hátrány adatintenzív számítási feladatok elvégzésekor ennek ténye, ugyanis arról beszélünk, hogy a Python ***nem rendelkezik natív számtípussal.*** Nem véletlen, hogy a hatékony modulok alacsonyabb szintű nyelveken nyugszanak, így a következő alfejezetben ismertetett PyTorch is.

További nem szokványos jellemzője a nyelvnek, hogy ***nem erősen típusos***, a változónevekhez *futási időben rendelődik hozzá* a referált objektum (azaz egy ***változónév*** igazából egy referens, vagyis adott ***példányra mutató referenciát*** tartalmaz). Habár Pythonban a referenciák teljesen úgy viselkednek, mint a változók (vagyis semmilyen szintaktikai kiegészítésre nincs szükség, még argumentumok átadása esetében sem, mint ahogy azt C++-ban láttuk). *(A félreértések elkerülése végett fontos szem előtt tartani, hogy amennyiben a segédletben a továbbiakban változót említek, akkor is igazából referencia van a háttérben, ezért hangsúlyozom általában, hogy nem a változó, hanem a változónév tartalmazza az objektumra mutató referenciát.)*

Vagyis lehetőség van arra, hogy egy referenshez (változónévhez) a programban különböző típusú objektumokat rendeljünk hozzá – ez teljesen logikusnak tűnik, ha belegondolunk abba, hogy a „Pythonban minden objektum" korábbi kijelentés lényegében azt sejteti, hogy minden az objektum ősosztály leszármazottja. A referenciák kezelése referenciaszámláló segítségével történik – a koncepció analóg a például C++-ban megtalálható *shared_ptr* típus esetén használttal.

Pythonban nincsen továbbá pointer sem, az ***argumentumok*** átadása ***referencia*** szerint történik. Itt azonban meg kell különböztetnünk az objektumokat az alapján, hogy módosíthatóak-e. A következőket érdemes alaposan átgondolni, különben hibás működésű programot kaphatunk.

Kétféle objektumtípus létezik, ***módosítható*** *(mutable),* ill. ***nem módosítható*** *(immutable).* Nem módosíthatóak többek között az egyszerű adattípusok *(POD, plain old data),* mint az egész vagy lebegőpontos számok, valamint a sztringek. Habár a gondolatmenet e megfontolás mögött elsőre furcsának tűnik, a következőképpen magyarázható: minden adott szám vagy szöveg egyedi, vagyis ha pl. két változóhoz hozzárendeljük az 5 értéket, akkor a kettő tartalma egymással teljesen azonos, ha az egyiket meg szeretnénk változtatni, akkor az már nem az az objektum. Mivel itt nem arról van szó, hogy egy tagváltozót írunk, hanem itt magát az objektumot teljes mértékben és kizárólagosan azonosító elemet változtatjuk meg, ami már nem ugyanaz az objektum. Talán érdemes a fordított programnyelvekből egy szemléletes példát átgondolni: a fordító ugyanolyan értékeket helyettesít be a gépi kódba (érdeklődők számára: a Pythonnak esetében is elérhető egyfajta disassembly a *dis* modul segítségével).

Módosítható lényegében minden egyéb típus, így a Pythonba beépített konténer jellegű típusok, mint *list, dict, tuple* (ezek különlegessége, hogy nem csak egy típust képesek egyidejűleg magukba foglalni, hanem bármilyen objektumot), de a saját osztályok példányai is ide tartoznak. Ebben az esetben a módosítás nem eredményezi új objektum létrehozását.

Ezek a különbségek ***objektumok másolása*** esetében is jelentkezik. Fontos különbség, hogy Pythonban alapvetően a hozzárendelés *(assignment)* igazából C++-szemszögből inkább a *copy constructor* hívásának feleltethető meg. ***Mutable*** esetben az eredeti objektumhoz tartozó referenciszámláló kerül megnövelésre, ***módosítás*** esetén pedig ***új objektum*** jön létre, értelemszerűen ugyanarra az értéket tartalmazó változók egyikének megváltoztatása nem hat ki a többire. ***Immutable*** esetben azonban nem ilyen egyszerű a helyzet: mivel alapvetően referenciákat tartalmaznak a változónevek, amelyek módosítható objektumokat referálnak, így ugyanarra a példányra mutató ***referenciák bármelyikének módosítása*** változást eredményez a referált egy darab objektum esetében, vagyis ***bármelyik változóval*** hivatkozunk rá, a ***változás*** mindegyik esetben ***látható*** lesz számunkra. Az ilyen jellegű másolást shallow copy-nak szokás nevezni, melynek párja a *deepcopy* (elérhető a *copy* modulban), ami Pythonban immutable esetben is új példányt hoz létre, így az új objektum független lesz a többitől.

Paraméterek átadása hozzárendeléssel történik, ez azonban nagy objektumok esetében sem okoz komolyabb problémát, ugyanis a ***referenciák*** kerülnek csak ***másolásra***. Ez a láthatóságra a következőképpen van hatással: ha a paraméter ***immutable*** és a ***függvénytörzsben módosításra*** kerül, akkor lényegében a függvény scope-jában egy ideiglenes objektum kerül létrehozásra, a függvényből ***visszatérve*** a **változó** megtartja ***eredeti értékét***. ***Immutable*** esetben, mivel a *referencia* kerül *másolásra*, az objektumok másolásánál láttuk, hogy az általuk referált objektumok *módosítása érvényes*, *bármelyik referenciájával hivatkozunk* is rá, így a függvény ***visszatérését*** követően az ***objektum*** már ***módosult*** értékével használható.

Rendkívül hasznos tulajdonság, hogy a Python gyakorlatilag ***bármennyi visszatérési értéket*** támogat.

A nyelvi koncepciók ismertetése után a következőkben a szintaktikai részletek kerülnek összefoglalásra.

Pythonban a ***programkód tagolása indentálással*** történik, vagyis a kódblokkokat egy tabulátorral beljebb kell kezdeni (ha valamilyen okból üres függvényt, ciklust, stb, kívánunk írni, akkor is kell egy indentált blokk, ezt egy sorban, a *pass* utasítással valósíthatjuk meg, ami nem hajt végre semmilyen műveletet).

Modulok betöltésére az *import* utasítással van lehetőségünk, mégpedig *kétféle* módon: importálhatjuk a teljes modult, ekkor a modul minden osztálya/függvénye a *modul neve után írt „.” (pont)* operátorral érhető el, ha *from*-ot használunk, lehetőségünk van csak egyes elemeket betölteni, ekkor *a modul nevét nem kell* az importált elem neve elé *kiírni*.

```
import module
from module import MyClass, my_func
```

Függvények a következő módon hozhatók létre:

```
def func(x):
    x +=1
    print("x = ", x)
```

A *def* kulcsszó után a függvény neve, majd a paraméterlista kerül megadásra, azt követően pedig az indentált függvénytörzs következik.

Osztályok esetében sem bonyolult a konstrukció:

```
class my_class(object):
    def __init__(self):
        self.x = 5
```

Pythonban a ***konstruktort*** az ***__init__*** rutin testesíti meg, mint látható, a tagfüggvények is majdnem teljesen megegyeznek az általános függvényekkel, azzal a különbséggel, hogy az ***első argumentum*** mindenképpen az ***adott példányra*** vonatkozik (mint ahogy a *this* C++-ban) – ezt konvenció szerint *self*-nek szoktuk nevezni.

Öröklés esetén nincs más teendőnk, mint az *object* osztály helyett megadni az általunk választott ősosztályt, majd a konstruktorban meghívni az ősosztály konstruktorát a ***super, általánosan az ősosztályra*** használható objektum segítségével.

```
class base_class(object):
    def __init__(self):
        print("I am Groot")

class inherited_class(base_class):
    def __init__(self):
        super().__init__()
        print("I am inherited")
```

További példaprogramok és kódrészletek találhatók az alábbi címen: https://gist.github.com/search?utf8=%E2%9C%93&q=user%3Arpatrik96&ref=searchresults

Aki esetleg mélyebben érdeklődik a Python nyelv iránt, annak érdemes lehet felkeresnie a következő TMIT SmartLab blogjának következő bejegyzéseit (angolul) :

https://medium.com/@SmartLabAI/python-under-the-hood-tips-and-tricks-from-a-c-programmers-perspective-01-b5f96895663

https://medium.com/@SmartLabAI/b52675c7c0af

A Python programnyelvhez számos hasznos függvénykönyvtár tartozik, melyek a mérési feladatok megvalósítását nagymértékben megkönnyítik. A Python nyelv egyik rendkívül kényelmes funkciója a beépített package manager, amelynek segítségével az egyes könyvtárak automatikusan telepíthetők, telepítsük után pedig minden további beállítás nélkül használhatók. A Pythonhoz két ilyen package manager is tartozik, az egyik a Pip, amely a legtöbb telepíthető Python verzió mellé automatikusan települ, a másik pedig az Anaconda, ami a könyvtárkezelési funkciókon túl virtuális környezeteket is képes kezelni. További információk az Anacondáról elérhetők itt: https://www.anaconda.com/

A Python egyik legfontosabb függvénykönyvtára a Numpy, amely tömbök kezelésére, illetve számtalan numerikus algoritmus használatára ad lehetőséget. A Numpy funkcionalitását

kiegészíti a MatplotLib, melynek segítségével különböző ábrákat készíthetünk a tömbjeinkről. Egy harmadik rendkívül hasznos könyvtárcsalád a scikit, ami számos tudományos számításhoz szükséges alkönyvtárt foglal össze. A sckit-image képek kezelésére, a scikit-learn gépi tanulás algoritmusok használatára, míg a scikit-fuzzy fuzzy logika használatára ad lehetőséget. Ezek a könyvtárak tulajdonképpen együttesen kiadják a Matlab funkcionalitásának jelentős részét.

3. Mérési feladatok

A mérés folyamán az alábbi feladatokat kell elvégezni:

1. Készítsen eljárást egy adott objektum 2D-ben történő követésére egy RGB-D kamera mélységképe alapján!
2. Készítsen eljárást egy adott objektum szín alapú követésére a HSV színtérben!
3. Határozza meg az adott objektum koordinátáit a 3D térben!
4. Ellenőrizze az algoritmus helyes működését előre felvett videófelvételek, valamint élő videó segítségével!

4. Hasznos kódrészletek

Részlet kivágása a képből

```
array[y1:y2,y1:y2]
```

Küszöbözés adott tartományban

```
roiMask = cv2.inRange(depthRoi, np.array([minval]), np.array([maxval]))
```

Egy másikkal egyező méretű csupa nulla kép készítése

```
mask = np.zeros_like(binary)
```

Kontúrok keresése

```
_, contours, _ = cv2.findContours(binary, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
```

Kontúr területének számítása

```
area = cv2.contourArea(cont)
```

Kontúr rajzolása:

```
cv2.drawContours(mask, contours, maxInd, 255, -1)
```

Momentumok számítása

```
moments = cv2.moments(contours[maxInd])
```

HSV konverzió

```
imgHsv= cv2.cvtColor(imgRoi, cv2.COLOR_BGR2HSV)
```

Kép maszkolása

```
maskedImgHsv = cv2.bitwise_and(image, image, mask=mask)
```

Csatornánkénti szétválasztás

```
hueImg = maskedImgHsv[:, :, 0]
```

Bináris maszkok kombinálása

```
hueMask = np.logical_and(cond1, cond2)
```

Bináris maszk alkalmazása

```
hueVals = hueImg[hueMask]
```

Hisztogram számítása

```
h = np.histogram(hueVals, 179)[0]
```

Maximum pozíció meghatározása

```
np.argmax(h)
```

Kör rajzolása

```
cv2.circle(image,center,size,(b,g,r),thickness)
```

Téglalap rajzolása

```
cv2.rectangle(image,(x1, y1),(x2, y2),(b,g,r),thickness)
```

5. Ellenőrző kérdések

1. Ismertesse röviden az RGB-D kamerák működési elvét!
2. Mi az a HSV színtér? Mire használhatjuk, és mi a használatának előnye?
3. Hogyan lehet egy bináris képen az objektumok kontúrjait meghatározni? Mire használhatók ezek fel?
4. Hogyan lehet egy bináris objektum tömegközéppontját meghatározni?
5. Milyen programozási nyelvet használunk a mérés során? Miért előnyös ez a nyelv multi-platform fejlesztés esetén?
6. Hogyan lehet osztályt és függvényt definiálni Python nyelven? (pszeudó kód elég)