SZAKDOLGOZAT

Kovács Tibor

2011

Debreceni Egyetem
Mérnök Informatika

3D szkennelés kódolt fénnyel

SZAKDOLGOZAT

Szerző: Kovács Tibor
Szak: Mérnök Informatikus
Szakirány: Mérés és folyamatirányítás
Témavezető: Végh János
Konzulens: Dr. Cserháti Csaba

Tartalom

1. BEVEZETÉS

A mérnököknek, gyártóknak fontos követelmény a pontosság. Mikor egy terv elkészül és a tárgy legyártásra kerül, nagyon fontos, hogy annak méretei egyezzenek a tervekben foglaltakkal. Egyszerű forma esetén könnyű dolog az ellenőrzés, ellentétben a bonyolultabb alakzatokkal, azokon nagyon nehéz mérni.

A restaurátoroknak, szobrászoknak, régészeknek szükségük lehet arra, hogy a különböző művészi-, vagy történelmi tárgyakat időállóan archiváljanak, lehetőleg a tárgy gyakran érzékeny felületének érintése nélkül.

A helyszínelők, a munkájuk során gyakran találkoznak olyan jelekkel, amik valamilyen formájú rögzítése jelentős segítséget nyújtana a későbbi vizsgálatokhoz. A helyszínről készült fotók azonban nem mindig nyújtanak megfelelő információt és ha a fotós rosszul választja meg a látószöget, ronthatja az újabb fontos információk kinyerésének az esélyeit.

Filmeseknek és játékfejlesztőknek gyakran fontos a nagy pontosságú és gyorsaságú 3D modellezés tárgyakról, színészekről, melyek véghezvitele a klasszikus 3D modellezéssel csak durva közelítéssel érhető el.

A fent említett területek dolgozóinak napjainkban elengedhetetlenek a 3D szkennerek. Ezek olyan eszközök, amelyek valamilyen felület-letapogatási módszert használva, egy virtuális térben a tárgyról készült nagyon pontos pontfelhőt hoznak létre. Leggyakrabban fényt, vagy ultrahangot használnak. Az ilyen megoldásokat alkalmazó mérőeszközök igen pontosak és némelyik igen gyors.

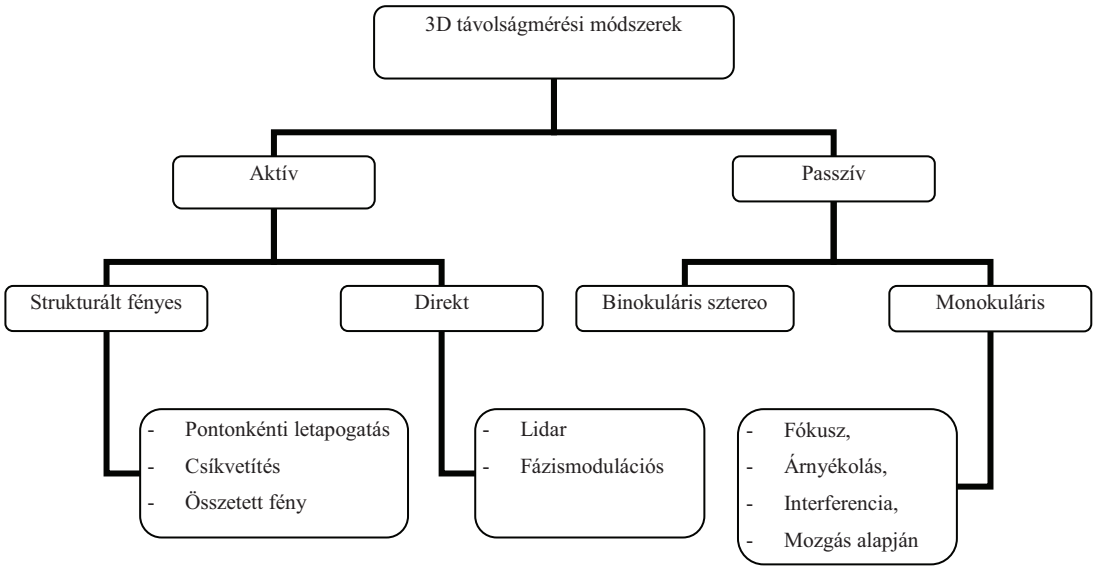
A mérnökök a megkapott 3D-s tárgyon bármilyen mérést elvégezhetnek, teljesen szabadon, fizikai megkötések nélkül. Megfelelő szoftverek segítségével még automatikusan is össze tudják hasonlítani a tervrajzokkal. Az ilyen mérési technikát hívják „reverse engineering”-nek. Az archiváló munkában nincs szükség gipszbe öntésre és nem kell veszélyes felületvédelmi vegyi anyagokat használni. A másolással sincs gond, mert napjainkban egyre kifinomultabbak és pontosabbak a 3D nyomtatók. A helyszínelők pedig könnyedén „lementhetik” a helyszínt, majd az időjárás romboló munkáját kiküszöbölve, a munkahelyen könnyedén megvizsgálhatják a bűntény helyszínét, összehasonlíthatnak nyomokat, stb… A grafikus pedig csak leülteti a modellt és ”beszkenneli”. Ha erre nem lenne lehetősége, legalább három oldalról fény-

képeket kellene készítenie, majd modellezői és anatómiai ismereteit, tapasztalatait felhasználva kellene modellt készítenie.

3D szkennerek létezéséről régóta tudok, de azok működését nem ismertem. 2010 őszi félévében felvettem a Műszaki képfeldolgozás nevű tárgyat, ahol mindenkinek saját feladatot kellett megoldania. Én a vonallézeres 3D szkenner-t választottam. Annyira megtetszett az eszköz, hogy elkészülte után alkalmazni szerettem volna, de a rendszerhez nem volt semmilyen mozgató felületem, ahol a tárgyat, vagy a lézert és a kamerát mozgathattam volna. Egy másik hátrány, ami inkább a módszer hátránya, hogy egy tárgyhoz rengeteg felvétel szükséges, ami nagy felbontású képek esetén gondot okozhat. Ez egy lassú megoldás. Ezért egy olyan módszerre volt szükségem, amellyel anélkül tudok pontfelhőt készíteni, hogy a tárgyat, vagy a kamerát és a fényforrást elmozdítanám, lehetőleg kevés képet kell benne feldolgozni, így lényegesen felgyorsítva a rögzítési folyamatot. Ennek megoldása egy olyan algoritmus, amely képes egy képen több különböző vonalat elkülöníteni és pontosan azonosítani anélkül, hogy egy bonyolult felület esetén tévesztene! Erre nyújthat megoldást a kódolt fénnyel történő 3D-s szkennelés.

2. 3D SZKENNELÉS KÓDOLT FÉNNYEL

A kódolt fénnyel történő szkennelés az aktív távolságmérési technikák csoportjába tartozik. Akkor beszélünk aktív távolságmérésről, ha a vizsgált tárgy felületére meghatározott jeleket viszünk fel. Az aktív technikákon belül a strukturált fényes módszerek közé sorolható. A struktúra egymást kölcsönösen meghatározó, alkotó elemek rendszert alkotó, összefüggő egységét jelenti (Bakos Ferenc – Idegen szavak és kifejezések szótára, Akadémia kiadó, Budapest, 1986). A strukturált fény tehát a 3D szkennereknél egy olyan felületre vetített mérőfény, ami több elemből (szekvenciából) épül fel és ezek az elemek egy összefüggő információhalmazt hoznak létre. Ezen belül pedig az összetett fényű 3D szkennerekhez tartozik a kódolt fényű megoldás (1. ábra).



1. ábra – 3D távolságmérési módszerek csoportosítása

2.1 Előzmények

Mint már említettem, a Műszaki képfeldolgozás című tárgy előadásain és gyakorlatain foglalkoztam először 3D szkenneléssel. Vonallézer segítségével és egy kamerával kellett megcsinálni a feladatomat. A vonallézeres szkenner egy viszonylag egyszerű megoldás felületi pontok mélységértékeinek meghatározására. Eszközként kap-

tam egy vonallézert, amely fényét rá kellett irányítani a tárgyra. A vonallézer, mint fényforrás mellé egy ismert távolságba kellett helyezni egy kamerát, így az elkészült fényképen egy görbe vonal volt látható. Egy meghatározott egyenestől való távolságból könnyedén meg lehetett határozni a különböző pontokat egy 3D-s térben.

3D grafikával sokat foglalkoztam és úgy látom, hogy ez a technika komoly segítséget nyújt gyors modellezésre. Ezért elkezdtem foglalkozni a 3D szkenneléssel, de a fent említett technika nagy hátránya a rendszer lassúsága és a pontossághoz szükséges sok felvétel követelménye. Élő modelleken pedig csak akkor van lehetőség pontos mérésre, ha a modellt valahogyan stabilizáljuk. Gyorsabb eszközre volt szükségem.

2.2 Alapelvek, alapműködés

A strukturált fényű 3D szkennelés az aktív 3D távolságmérő módszerek közé tartozik. Működéséhez négy dologra van szükség: egy fényforrásra, egy kamerára, a mérendő tárgyra és egy feldolgozó szoftverre. Mivel mérésről van szó, fontos tudni legalább a mérendő objektum és a kamera (vagy a fényforrás) közötti távolságot, valamint meg kell határozni egy viszonyítási pontot, amihez képest megmondjuk a mérendő objektum megtalált pontjainak a pontos, térbeli elhelyezkedését. A fényforrás és a kamera közötti távolságnak a mérés során végig azonosnak kell lenni akkor is, ha el kell forgatni a rendszert.

A fényforrás feladata, hogy a mérendő felületet mérőpontokkal lássa el, amit egy általános projekcióval érünk el, irányított fényekkel. A fényforrás lehet lézeres, projektoros, stb… A lézeres a legolcsóbb, mivel kis energiájú is teljesen alkalmas a feladatra. A lényeg, hogy a kamera képén mindig meg lehessen találni a fényforrás pontjait. Nincs szükség összetett lencserendszerre, izzókra, csak egy lézerforrásra. Hátránya, hogy viszonylag kötött a megvalósítható struktúrák száma, esetenként csak egyetlen vonalat, vagy pontot képes felvetíteni a mérendő felületre, bár megfelelő optikai elemekkel egészen bonyolult struktúrák valósíthatók meg. További hátrányként hozható fel, hogy a lézerfény monokróm. A projektoros módszer hátránya, hogy drága, de nagy előnye, hogy gyakorlatilag bármit lehet vele vetíteni. A fény irányítását lencserendszerrel oldja meg.

Mivel a kamera és a fényforrás között térbeli távolság van, parallaxis jelenséget tapasztalhatunk, ezt használjuk ki. Ennek köszönhetően a kamerából nézve a felüle-

ten a mérőegyenesek görbe vonalaknak látszanak. Egyenetlen felület esetén amennyire eltér a kitüntetett távolságtól a felület vizsgált pontja, a mérőegyenes adott pontja is úgy távolodik el a kamera által alkotott kép síkjában.

Mivel fénnyel dolgozunk, a rendszer nagyon kritikus a felület textúrájára. Ha a feldolgozó szoftver érzékeny az intenzitásra, például egy fekete foltokat tartalmazó tárgyon ezeket a részleteken figyelmen kívül hagyhatja. Könnyen hibaforrás lehet még a felület fény-visszaverődési indexe. Minél inkább fényvisszaverő felületről van szó, annál nagyobb valószínűséggel fog mérési hiba jelentkezni. Ilyen esetekre felületkezelési eljárásokat szoktak alkalmazni, ha ezt meg lehet oldani. Ellenkező esetben más megoldást kell keresni.

2.3 A strukturált fény és annak forrása

A strukturált fényű 3D szkennereknek a szakdolgozatom írása idején három csoportja létezik (Lasser András, Lézeres távolságmérés, mérési útmutató, Budapest Műszaki egyetem, 2000):

- mérőpontos,
- mérővonalas,
- összetett fényű.

A *mérőpontos* módszer a legegyszerűbb, mivel csak egyetlen egy pontot vetítenek a mérendő tárgyra, amíg egy kép elkészül. Így csak egyetlen pontot kell keresnünk képenként. Általában a kamera és a fényforrás között rögzített távolság van, a mérőpontot a fényforrással együtt mozgatjuk. A viszonyítási pontot kell megadni, így az elegendő információhoz már csak a mozgatás mértékét kell ismerni, ezek ismeretében már pontos 3D pontfelhőt alkothatunk. Ezt a rendszert általában építészetnél alkalmazzák, vagy olyan helyeken, ahol viszonylag nagy területről kell digitális 3 dimenziós képet alkotni. Leggyakoribb megoldás a mozgatásra, a rögzített helyen való forgatás.

A *mérővonalas* megoldásról már írtam az előzményben. Összefoglalóul ennek működésének alapja, hogy a felületre egyetlen egyenest vetítünk, amit szögből a kamera rögzít, képén pedig a felülettől függő görbe észlelhető. Ebből könnyedén elő lehet állítani a 3D-s pontfelhőt. Gyakorlatilag 3-4 méteres nagyságnál igen hatékony. Régebben ezt használták arc szkennelésére is, például a Terminator 2 c. filmben is így alkották meg a T3000-es fantázianevű robot higanyos modelljét Robert Patrick-

ről. A módszer feltétele a mozdulatlan téma. A teljes szkenneléshez azonban vagy a kamerát és a fényforrást kell együttesen mozgatni, vagy a témát.

Az *összetett fényű* megoldás a szakdolgozatom fő témája. Lényegében ez egy olyan módszer, ahol egyetlen képen több mérőegyenes található, együttesen mérőrácsot alkotva. Legnagyobb előnye az, hogy nem szükséges elmozdítani sem a témát, sem a mérőeszközöket. Mivel egyetlen képen több egyenest vetítünk, lényegesen kevesebb felvételt kell készíteni a kamerával, így viszonylag gyorsan el lehet végezni a teljes mérést. Az egy felvételen lévő több vonal feldolgozásának van egy nagy nehézsége nevezetesen, hogy nem minden esetben lehet egyértelműen elkülöníteni azokat, például a következő esetekben:

- bonyolult a tárgy felülete és kitakar más vonalakat,
- felületi töréseknél az egyik egyenes látszólag a másikban folytatódik,
- bizonyos vetített vonalak egyáltalán nem jelennek meg a tárgyon,
- bizonyos vonalak a vizsgált felület egyes részein nem látszanak.

Ennek kiküszöbölésére több megoldás létezik. Az egyképeseknél valamilyen logika szerint minden egyes egyenest különböző színnel jelölnek, amelyeket majd a feldolgozó szoftver ezek alapján képes beazonosítani. Mivel monokróm felületet igényel, ritkán használják. Egy másik egyképes módszer úgy működik, hogy egy mintamátrixot vetít a felületre, aminek elemei egykarakteresek. Ezeket kell felismernie a szoftvernek, ami nem egyszerű feladat, mert ha egy felületi törésen található a vizsgált karakter, a hozzá tartozó pont elveszik. Továbbá csak kevés pontot képes meghatározni.

Többképes megoldások lényege, hogy a mérés alatt több felvétel készül, egy felvételen több mérőegyenes kerül vetítésre, de mindegyiken új jelenik meg. Ezek közé sorolható a többlézeres megoldás, de gyakoribb a kódolt fényes-, ezen belül a legismertebb, a Gray-kódos módszer. Szakdolgozatomban ez utóbbival foglalkozom.

2.3.1 A Gray-kód módszer

A mérés alatt valójában nem mérőegyeneseket, hanem területekre bontott képeket vetítünk a mérendő felületre. A területek vagy fehérek, vagy feketék, a kódtól függően. A mérés előtt meg kell határozni, hogy hány bites kódot alkalmazunk és el kell számolni 0-tól a maximálisan ábrázolható számig. A vetített szekvencia területeit a számsorozat azonos helyértékei ábrázolják. Például az első vetített szekvencián a

sorba állított sorozat első helyértékei. Azon terület, amelyik hozzárendelt kódjának értéke 0, fekete színű lesz, 1 érték esetén fehér.

0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

1. táblázat – bináris kód

0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0
0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0
0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0

2. táblázat – Gray-kód

Az 1. és a 2. táblázat két különböző kódot ábrázol (ugyanazt a négy bites értéket, kétféle kódolásban). Az 1. a binárist, a 2. a Gray-kódot. Mindegyik 4 bites számokat ábrázol, 0-tól 15-ig számlál felfelé, a saját szabályuknak megfelelően. A vetített szekvenciák mindegyike 16 területből áll, összesen 4 szekvenciát vetítünk. Az egyes szekvenciák területeinek kódjait a táblázatok egyes soraiból választjuk ki, például az első szekvencia a táblázat első sorát használjuk fel ebben a sorrendben, így végül a kép baloldala fekete lesz, míg a jobb fehér.

Gray-kódot azért érdemes alkalmazni, mert sokkal praktikusabb. Az első szekvencia esetén mind a binárissal, mind a Gray-kóddal két területet vetít a fényforrás a mérendő tárgyra, ekkor csak ezt a kettőt kell felismernie a feldolgozó szoftvernek. A két terület határán egy vonalat lehet felismerni a mérendő objektum felületén. A következő szekvenciánál a bináris képnél három egyenest detektálhatunk, míg a Gray esetén kettőt. A középső ismeretében ennyi is elégséges, míg a bináris itt is ábrázolja a középsőt. A harmadik szekvenciánál a bináris megoldás 7 egyenest mutat, míg a Gray 4-et. A negyediknél pedig a bináris az összes vonalat láthatóvá teszi, a Gray pedig csak 8-at. A Gray esetében megfigyelhető, hogy minden vonalat csakis egyetlen egyszer ábrázol. Ez már csak azért is jó, mert alacsonyabb felbontású felvételeknél nem fog bezavarni például úgynevezett Moire jelenség (képi interferenciajelenség - http://pixinfo.com/cikkek/fotoelmelet_antialiasing).

2.3.2 A strukturált fény forrása

Különböző módszerek különböző fényforrásokat használhatnak. Ezek közül a leggyakoribbak:

- pontlézer,
- vonallézer,
- projektor.

A vonallézer gyakorlatilag megegyezik a pontlézerrel, a különbség csak annyi, hogy a lézer forrása elé közvetlenül elhelyeznek egy hengerlencsét, ami hatására úgy törik meg a fénye, hogy egy egyenest hoz létre. Mind a pont-, mind a vonallézer nagy előnye az olcsóság, mivel nem tartalmaznak összetett optikai rendszert és bonyolult elektronikát. Másik nagy előnyük a nagyon rövid válaszidő, ez ms-os nagyságrendű.

A projektor nagy előnye, hogy univerzális eszközként is lehet használni, mivel képes egyaránt pontot és egyenest is vetíteni, de természetesen bonyolultabb struktúrák vetítésére is alkalmas. Hátránya a viszonylag kis mélységélessége, kis sebessége és magas ára. A kis mélységélesség miatt a fókusztartományon kívül az élek elmosódnak. Lassú eszköz, 1 másodperc alatt legfeljebb 30 képváltásra képes és a három alapszínt is különböző időben jelenítni meg.

2.4 A kép érzékelés és analízise

A képet egy képalkotó eszköz segítségével rögzítjük. Mozgókép, vagy képsorozat is készíthető.

A kamerát úgy kell elhelyezni, hogy tudjuk a pontos távolságát a fényforrástól, vagy a viszonyítási ponttól. Fontos, hogy a kamera képének a közepe pontosan a viszonyítási pontra nézzen. Az optikai kalibrációval a szakdolgozatom nem foglalkozik.

Az expozíciót úgy kell beállítani, hogy a feldolgozó algoritmus egyértelműen meg tudja különböztetni a mérőpontot vagy mérőegyeneseket a kép többi elemétől. Ennek érdekében oda kell figyelni a környezeti fényekre is, érdemes sötétben dolgozni.

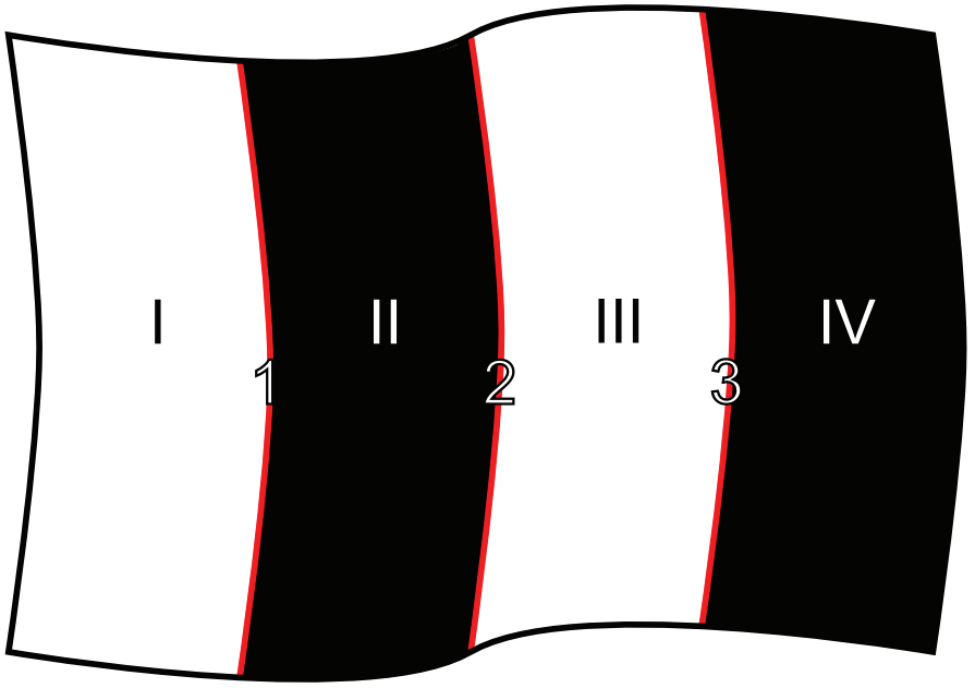
Ha mind a torzítás kompenzációja és az expozíció is beállításra került, elkezdődhet a mérés. Mozgókép esetén elegendő a megfelelő sebességgel módosítani a fényforrás képét, állóképek esetében pedig trigger jellel érdemes dolgozni, de két felvétel között változnia kell a fényforrás képének.

A képek elkészülte után már csak az élek meghatározása van hátra.

2.4.1 Éldetektálás

A már említett Gray-kód módszert választottam. Ennek a lényege, hogy a létrehozott képsorozat képein, a tárgyra vetített mérőrács egyeneseit úgy lehet megkülönböztetni, hogy mindegyik mérőegyenes a vetített szekvencia két szomszédos, különböző színű területeinek határára esik (2. ábra). Az első szekvencián csak a középső vonal fog megjelenni, így mivel ezt már ismerjük, ehhez tudjuk viszonyítani a következő szekvencia vonalait.

Hogy általában hogyan oldják meg a beazonosítást, arról nem sikerült leírást találnom, mivel bár több forrás foglalkozik ezzel a módszerrel, nem mennek bele ezen résznek a tárgyalásába. Ezért saját megoldást kellett kidolgoznom.



2. ábra – A területek és a közöttük lévő érzékelhető vonalak

2.4.2 Az élek beazonosításához kidolgozott módszerem

Mielőtt azonban ez megtörténne, szükséges lehet egy zajszűrés. Azért van erre szükség, mert amikor a program figyeli, hogy melyik terület fekete vagy fehér, előfordulhat, hogy a vizsgált felvételen több helyen is rosszul határozza meg a színt

(például a fehér területeken fekete pontokat érzékel és fordítva), a képeken lévő zaj miatt.

Több zajszűrő algoritmus létezik, de ezek közül a legegyszerűbb a Gauss féle lineáris aluláteresztő szűrő. A módszer a képmátrix és egy un. kernel mátrix konvolválásán alapul. A szűrés eredményéül a vizsgált pixel egy lokális (általában első, vagy második szomszédok) környezetének súlyozott átlagát kapjuk. A Gauss féle aluláteresztő szűrő általam használt formája:

$$\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix} / 16$$

Zajszűrés után elkezdődhet az éldetektálás folyamata. Sok esetben szintén egy konvolúciós függvényt használnak, de a Gray-kódos strukturált fényes 3D szkennelésnél nincs feltétlenül szükségünk rá, legfeljebb csak a későbbi pontosításoknál. Itt most az élek beazonosításához a tárgyra vetített kép területeinek színét kell vizsgálni.

Az élek beazonosításához a tárgyra vetített képet 1 bites képpé alakítjuk egy megfelelően megválasztott küszöb-értékkel: ha az aktuális pixel színintenzitása egy előre meghatározott küszöbérték alatt van, akkor fekete, ellenkező esetben fehér.

A Gray-kódot annyival egészítettem ki, hogy az első szekvencián a projektor vetített képe fehér. Erre azért van szükség, mert így könnyedén meg lehet állapítani már a mérés legelején, hogy a készített képek melyik tartományát kell figyelmen kívül hagyni, ezzel is kizárva a zavaró képi elemeket.

A teljes analizáláshoz ismerni kell a szekvenciák számát, amibe nem számít bele a teljesen fehér vetített kép. Ennek ismeretében meg lehet határozni, hogy hány területet kell felismerni: $\text{területek száma} = 2^{\text{szekvenciák száma}}$, majd ebből meghatározható, hogy hány vonal van: $\text{vonalak száma} = 2^{\text{szekvenciák száma}} - 1$.

A területek számozása a $[0\ldots\ \text{területek száma} - 1]$ intervallumba esik. Hogy melyik milyen számot kap, a következőképpen határozható meg:

a) A képek dimenzióival megegyező nagyságú *képpontok* tömb létrehozása. A képekből kinyert adatok ide fognak kerülni.
b) 0. kép betöltése, limitálása (a meghatározott érték felett fehér-, alatta pedig fekete színű pixelek beállítása), zajszűrése.

c) A képen fehéren szereplő pixelek 0 értéket kapnak a *képpontok* tömb megfelelő elemei, a feketék -2-t, vagy kisebbet (az n) pont miatt).

d) Következő kép betöltése, limitálása, zajszűrése. Ha 1. szekvencia, akkor e) pont, n. szekvencia esetén (ahol $n < \text{területek száma}$) f) pont. Ha nincs több kép, akkor k) pont.

e) A képen lévő fekete pixelek értéke 0, a fehérek értéke pedig = $\text{területek száma} - 1$, a *képpontok* tömb megfelelő elemeit felülírva. Ugrás d) pontra.

f) Az n-1. (vagyis az előző) szekvencia szekciókódhoz tartozó növekményének megállapítása, ami egy maradék nélküli osztás: $növekmény = \frac{\text{területek száma} - 1}{2^{n-1}}$, majd a *szekciókód* tömb létrehozása, amely $k = 2^{n-1}$ elemű. A *szekciókód* tömb feltöltésének menete:

   1. 1. eleme = 0.
   2. m. eleme (ha m páros) = $szekciókód_{\text{m-1}} + növekmény$
   3. m. eleme (ha m páratlan) = $szekciókód_{\text{m-1}} + 1$

g) f) ponthoz hasonlóan az *aszekciókód* létrehozása, ami az aktuális szekvenciához tartozó szekciókódokat jelöli. Elemeinek száma $t = 2^n$. Feltöltése ugyanazokkal a lépésekkel, ahogy a *szekciókód* tömbbel is történt:

   1. 1. eleme = 0.
   2. g. eleme (ha g páros) = $aszekciókód_{\text{m-1}} + növekmény$
   3. g. eleme (ha g páratlan) = $aszekciókód_{\text{m-1}} + 1$

h) Egy $l = 2^n$ elemű *színkód* tömb létrehozása, ami elemei igaz értéke fehér színt jelöl, a hamis értéke pedig feketét. A *színkód* tömb feltöltésének menete:

   1. 1. eleme = igaz.
   2. h. eleme (ha h páros) = $\overline{színkód_{h-1}}$
   3. h. eleme (ha h páratlan) = $színkód_{h-1}$

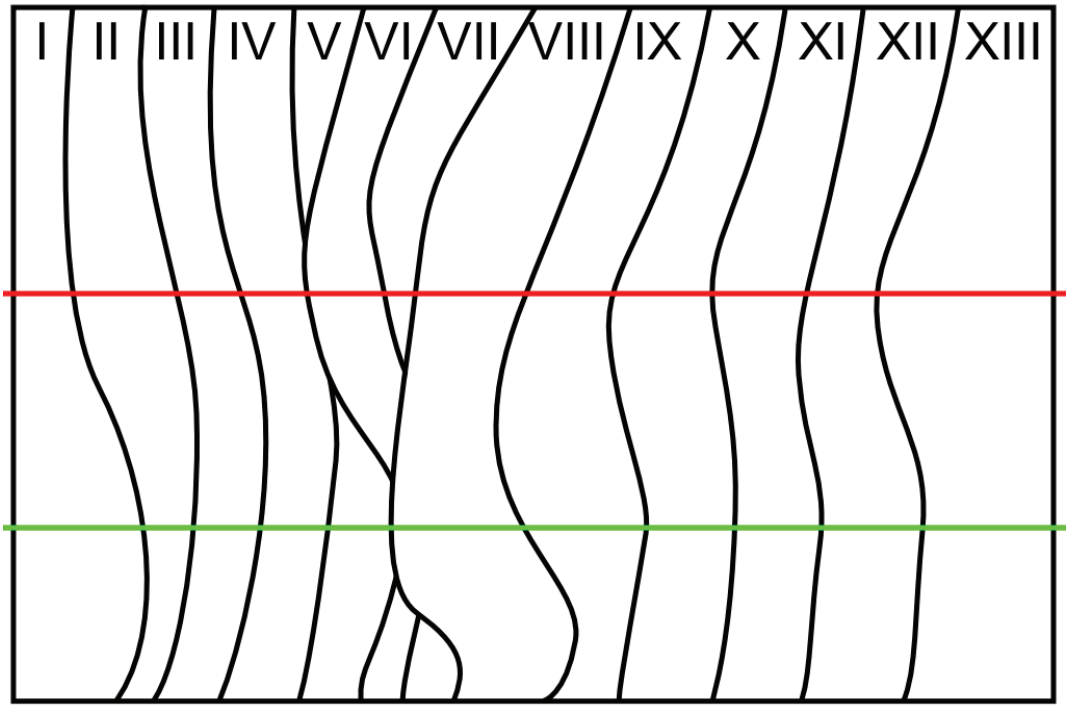
i) A *képpontok* tömb minden elemének (x. sor, y. eleme) vizsgálata. Ha a vizsgált elem értéke ≥ 0, akkor a *szekciókód* tömb bejárása következik és ha $képpontok_{\text{x,y}} = szekciókód_m$, akkor a *színkód* tömb h. eleme: $h = m \cdot 2$.

Ha n. $kép_{x,y}$ = fehér és $színkód_h = igaz$, akkor $képpontok_{x,y} = aszínkód_h$, egyébként $képpontok_{x,y} = aszínkód_{h-1}$.

j) Ugrás a d) pontra.

k) Vége.

Miután sikerült létrehozni a szekciókat ábrázoló tömböt, ebből könnyedén meg lehet határozni az éleket is. Ehhez fontos észrevenni, hogy hol jöttek létre a szekciók és hogy melyek között beszélhetünk valódi élekről. Ez azért is fontos, mert a szekcióhatárokon jönnek létre élek, de csak bizonyos szekciók között. Előfordulhat bonyolult felületek esetén, hogy az egyik (vagy akár több) szekció egy adott részen teljesen eltűnik (3. ábra).



3. ábra – Szekciók. A piros vizsgáló sorban nincs V szekció, mert kitakarja VI. A zöld sorban VII nincs, mert VIII takarja ki.

Szekciók mindig ott jönnek létre, ahol nincs él, tehát ott, ahol egyik képen sincs átmenet fehér és fekete között. Pontosan annyi szekció jön létre, amekkora számot képes ábrázolni a használt Gray-kód, plusz 1. A módszernek köszönhetően mindegyik szekció pontosan a megfelelő kódot kapja, mindegyik tökéletesen beazonosítható, összekeverhetetlenül elkülöníthető. Továbbá minden szekvencia számozása balról-jobbra növekményes, 0-tól kezdődően. Így két szomszédos szekvencia kódjai

közötti különbség pontosan 1. Ezt kihasználva könnyedén meg lehet mondani, hogy hol található valóságos él és pontosan meg lehet mondani, hogy melyik él hányadik!

Ez a következőképpen néz ki:

l) Az *élek* kétdimenziós tömb létrehozása, melynek $2^{\text{szekvenciák száma}} - 1$ oszlopa van és annyi sora, amennyi a képek magassági pixeleinek száma.

m) A *képpontok* tömb y. sorának vizsgálata az első elemétől az utolsó előtti eleméig.

n) Ha $sor_{x+1} - sor_x = 1$, akkor a talált él valós. Ekkor $élek_{sor_{x+1},y} = x$.

o) Ha x a sor utolsó előtti elemére mutat, akkor ugrás p)-re, egyébként $x = x + 1$ és ugrás n)-re.

p) Ha y a *képpontok* utolsó sorára mutat, akkor ugrás q)-ra, egyébként $y = y + 1$ és ugrás m)-re.

q) Vége.

Az n) pontban látható, hogy az algoritmus csakis akkor foglalkozik egy éllel, ha két szomszédos szekció a valóságban is szomszédos. Ekkor az *élek* tömbnek már meg is mondja, hogy melyik élről van szó, ezt az első indexből látjuk, mivel a sor_{x+1} már mutatja, hogy melyik szekcióról van szó és csakis egy él tartozik hozzá.

Az itt leírt algoritmusokat végre kell hajtani mind a mérőrácson, mind a mérendő tárgyon. Miután elkészültek a felvételek, elkészülnek a mérőrács referencia pontjai és a tárgyon lévő pontok, el lehet kezdeni a pontonkénti mélységi információ kiszámítását.

2.5 Mélységi információ

Miután sikerült beazonosítani az éleket mind a mérőrácson, mind a felületről készült képeken, meg lehet határozni a pontok mélységeit. Ehhez viszont ismerni kell még néhány adatot.

Ennél a pontnál fontos megjegyeznem, hogy mikor készült a szakdolgozatom, saját egyenletet dolgoztam ki erre a feladatra, ami csak részben pontos. Mikor már elkészültem mind az egyenlettel, mind a szoftverrel, találkoztam egy jobb, egyszerűbb és pontosabb megoldással is (Hoa G. Nguyen, Michael R. Blackburn, A Simple Method for Range Finding via Laser Triangulation). Mivel az előbbivel foglalkoztam

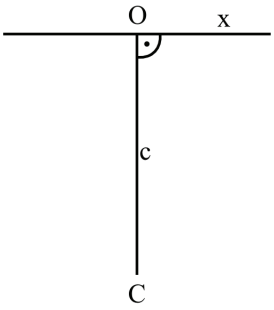
többet és mivel az utóbbi sokkal pontosabb, ezért mind a kettő megoldást bemutatom.

2.5.1 A saját egyenletem

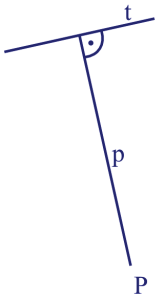
Az egyenlet alapelve, hogy előbb meg kell határozni a mérőrács méreteit milliméterben, majd ahhoz képest kell kiszámítani a pontok távolságát, a kamera, a fényforrás és a mérőrács egymáshoz képesti távolságaik alapján.

Először meg kell adni a kamera és a fényforrás távolságát, majd a kamera és a mérőrács közötti távolságot. Az előbbit *m*-el jelöljük, az utóbbit *c*-vel. Mivel a kettő derékszögben áll egymással, Pitagorasz-tétellel ki lehet számítani a fényforrás és a mérőrács közötti távolságot, amit *p* jelöl: $p = \sqrt{c^2 + m^2}$.

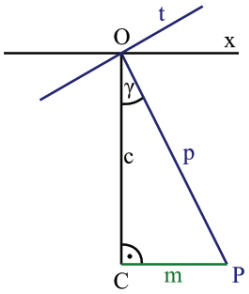
A rajzokon (4-10. ábrákon), hogy egyszerűbb legyen a számítás, a kamera síkját (*x*) a mérőrácsra rajzoltam.



4. ábra – A kamera

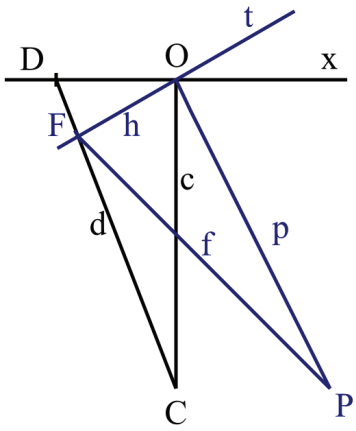


5. ábra – A fényforrás

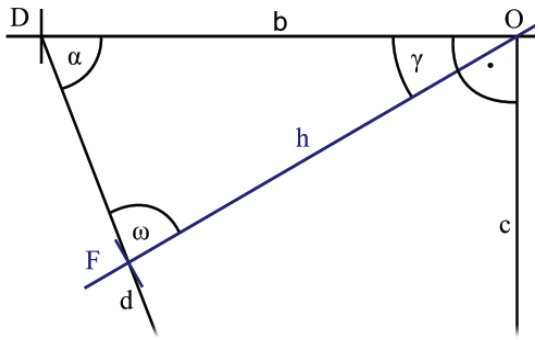


6. ábra – A kettő együtt

A fényforrás és a kamera optikai tengelye (*c* és *p*) közötti szöget γ jelöli. Mivel a *c*, a *p* és az *m* derékszögű háromszöget alkot, a γ könnyen kifejezhető többféleképpen, de mivel *m* és *c* ismert adat (míg a *p* számított), így a tangens szögfüggvénnyel érdemes számolni: $tg\ \gamma = \frac{m}{b}$.



7. ábra – Vetített pont és érzékelése



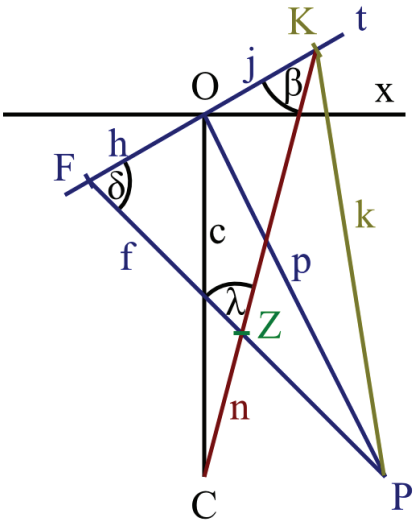
8. ábra – A pontok alkotta háromszög

Amikor a *P* fényforrás a *t* síkban lévő mérőrácsra egy *F* pontot vetít az *f* egyenessel (7. ábra), akkor azt a *C* ponttal ábrázolt kamera a *d* segédegyenes mentén a *D* pontban látja azt az *x* síkon. Tehát a kamera az *F* pontot az ő képsíkjában (*x*) a *D* pontra vetíti. A feladat a *h* egyenes ($h = \overline{OF}$) meghatározása, az ismert $\overline{OD}$ távolság alapján (ahol *O* a kamera képsíkjának és a fényforrás vetített mérőrácsnak a középpontja).

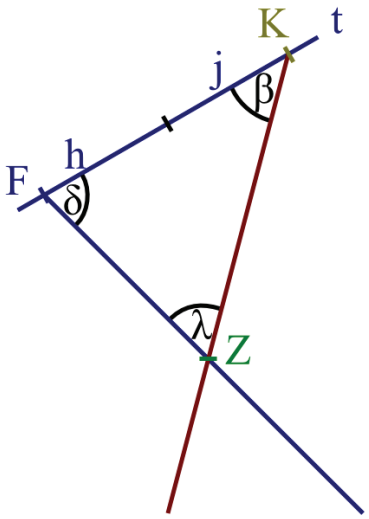
Az 7. és a 8. ábrán jól látszik, hogy ez egy háromszöggel könnyedén megoldható. Fontos észrevenni, hogy a *DFO* háromszög $\overline{DF}$ egyenesével szemközti szög is γ. A *h* szakasz hossza szinusztétellel meghatározható, de ahhoz ismerni kell α-t és vagy a $\overline{DF}$ szakaszt, vagy az ω szöget. Az utóbbi tűnik a legegyszerűbbnek.

Először meg kell határozni *d* hosszát, amit egy Pitagorasz-tétellel meg lehet oldani: $d = \sqrt{b^2 + c^2}$, majd szinusztétellel α-t: $\sin\alpha = \frac{1}{d} \cdot c$ (mert d-vel szemben derékszög áll és $\sin 90^\circ = 1$). Ezután meg van $\omega = 90^\circ - \alpha - \gamma$ és ezen adatokkal ki lehet fejezni a *h* szakaszt: $h = \frac{b}{\sin\omega} \cdot \sin\alpha$. Végül sikeresen megkaptuk a kamera által látott *D* pont valós helyzetét a mérőrácson.

Ebben a lépésben a mérőrács azon pontjainak pontos helyzete számolhatók ki, amelyek a mérőrács mérőegyeneseit alkotják. Ezek azok az egyenesek, amelyekhez kell viszonyítani a tárgyon létrejövő vonalakat, pontosabban a mérőegyenesek pontjaihoz a tárgy felületén létrejövő vonalak pontjait. Ehhez meg kell határozni ezen pontok helyzeteit is, ugyanezzel a módszerrel.



9. ábra – A megjelenő Z pont



10. ábra – Az *FZK* háromszög

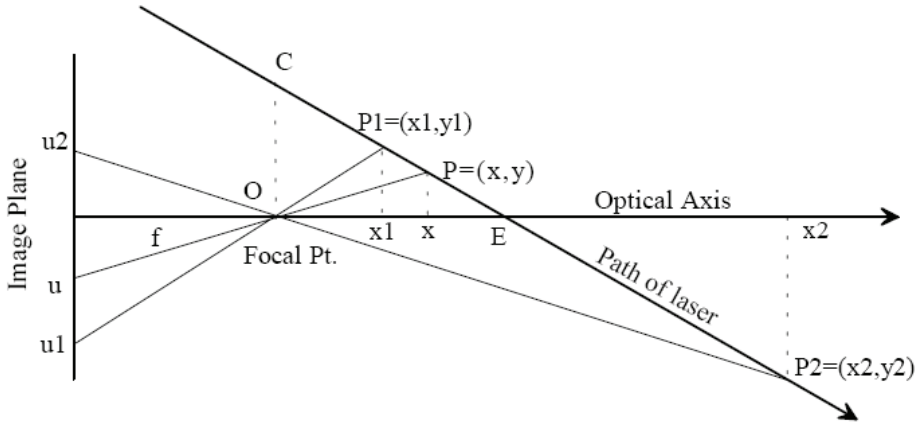
A 9. ábrán szereplő Z az a pont, ami a kamera által, a mérés során készített felvételeken a tárgyra eső egyik mérőegyenessel létrehozott vonalon van. Az ábrán látható, hogy a vizsgált *Z* pont, a fent említett számítási módszerekkel előállítva úgy tűnik, hogy az a mérőrácson a *K* pontra esik, mivel a *C* kamera azt az *n* egyenes és az *x* kamera síkjának a metszéspontjában látja. Az *n* és az *f* egyenesek a *Z* pontban metszik egymást, így létrejön egy új, *FZK* háromszög. Ennek a segítségével kell meghatározni az $\overline{FZ}$ szakasz hosszát, ami a vizsgált felület adott pontjának távolsága a mérőrácstól, a fényforrás felé.

A legegyszerűbb a δ szög meghatározása, ehhez csak a 8. ábrát kell felhasználni. Az ω szög egyértelműen meghatározza δ-t, mert pótszöge annak, így $\omega = 180^\circ - \delta$. A *j* kiszámításának módja megegyezik a fent leírtakkal, ezért azt ismertnek tekinthetjük. Ismerjük *c*-t, és a *j* és *c* által közre zárt szöget, ami $90^\circ + \gamma$, ennek köszönhetően most már meghatározható koszinusztétellel az egyenes *n* hossza: $n = \sqrt{j^2 + c^2 - 2jc \cdot \cos(90^\circ + \gamma)}$. Az ismert oldalakkal és a szöggel meghatározható szinusztétellel β: $\sin\beta = \frac{90^\circ + \gamma}{n} \cdot c$. Majd $\lambda = 180^\circ - \beta - \delta$, és végül $\overline{FZ} = \frac{h + j}{\sin\lambda} \cdot \sin\beta$.

A módszer azért pontatlan, mert a sok szögfüggvény miatt kerekítési hibák lépnek fel. Továbbá nagy számításigényű. Pontatlansághoz vezethet az alapadatok megadása is, amit csak precíziós távolságmérő műszerekkel lehet kiküszöbölni.

2.5.2 Egy másik megoldás

Ez a módszer azon alapszik, hogy a mérőrácsot nem egy síkra hozzuk létre, hanem egy lépcsőzetes tárgyra, így gyakorlatilag két mérőrács kerül definiálásra. A tárgy két párhuzamos síklapból áll, amik között egy ismert távolság van. Ennek nagy előnye, hogy egyedül az egyik sík távolságát kell megadni a rendszernek és a két sík közötti távolságot. A kamera és a fényforrás közötti távolság ismeretére nincs szükség.



11. ábra – A rendszer geometriája

A 11. ábrán az *O* pont a kamera objektívének fókuszpontja és *f* a gyújtótávolsága. A *c* a mérőegyenes „útja” a fényforrásból, az *E* pont pedig a *C* egyenes és az optikai tengely metszéspontja. Ismert a P_1 (tehát ismert x_1 és y_1), ami a fókuszponton keresztül az u_1 pontként jelenik a kamera képén, ismert a P_2 is (tehát ismert x_2 és y_2), aminek u_2 a képsíki megfelelője. Továbbá ismert *u*, de a hozzá tartozó *P* pont nem, annak *x* koordinátáját kell meghatározni.

A hasonló háromszögeknek köszönhetően két arányt lehet felfedezni: $\frac{y_1}{x_1} = \frac{u_1}{f}$ és $\frac{y_2}{x_2} = \frac{u_2}{f}$. A *c* meredeksége: $m = \frac{y_2 - y_1}{x_2 - x_1}$, így $c = y_2 - mx_2$ az egyenes egyenlete. Az arányosságokat behelyettesítve a meredekségbe, eliminálva y_1-t és y_2-t: $m = \frac{u_2 x_2 - u_1 x_2}{f(x_2 - x)_1}$, így $c = \frac{u_2 x_2}{f} - mx_2$.

Mivel az objektívek fókusztávolságát nehéz pontosan meghatározni (a zoom objektívek köztes állásánál komoly fejtörést tud okozni, de további nehézség az objektívek lélegzete), eliminálni kell *f*-et.

Nincs szükség y_1-re és y_2-re sem, ezért azok elhagyhatók, de tudjuk, hogy az O-n átmenő $\overline{uP}$ egyenest a következő egyenlet reprezentálja: $y = \frac{u}{f} \cdot x$, és a mérőegyenes útvonala a felületen: $y = mx + c$. Megoldva a két legutóbbi egyenletet és felhasználva azt az m-et és c-t kifejező egyenletet, amiben már nem szerepel y_1 és y_2 a következőt kapjuk: $x = \frac{N}{ud - k}$, ahol $d = x_2 - x_1$, $k = u_2 x_2 - u_1 x_1$, $N = (u_1 - u_2) \cdot x_1 x_2$.

Ennek a módszernek nagy előnye, hogy csak két adat szükséges a pontos eredményhez és viszonylag kis számításigényű. Azonban kell egy olyan referencia test, aminek pontosan ismerjük a párhuzamos lapjai közötti távolságát.

2.6 Összegzés

A leírtak alapján látható, hogy egy valós, fizikai testről, modellről, vagy élőlényről könnyedén létre lehet hozni egy megfelelő eszközzel egy 3D-s virtuális modellt. Erre több lehetőségünk is van, de mindegyik módszernek megvan a maga előnye és hátránya.

A kódolt fényű szkennernek nagy előnye, hogy viszonylag kevés felvételből egyértelműen fel lehet ismerni olyan vonalakat, amelyekből egyértelműen meghatározható a felületek pontjainak pontos térbeli elhelyezkedései. Ennek azonban összetett számítás az ára. De erre is több megoldás létezik, a maguk előnyeivel és hátrányaival.

3. A MÓDSZERT BEMUTATÓ SZOFTVER

Ahhoz, hogy az előzőekben leírt elvet bemutassam, egy demonstrációs szoftvert írtam. Ennek feladata, hogy egy mérőrácshoz képest kiszámítsa a mérendő objektumon érzékelt pontok 3 dimenziós helyzetét.

3.1 A fejlesztés menete

Mikor elkezdtem a szoftver írását, natív C++-ban szerettem volna megírni, OpenCL-el kiegészítve. Nehézséget okozott az OpenCL API ismeretének hiánya, így elkezdtem tanulni, de 3 hét múlva úgy döntöttem, kihagyom a projektből. Megpróbáltam a GUI-t Win32 API-val megírni, de a fejlesztőkörnyezet amit használtam gyakran lefagyott, ellehetetlenítve a munkát. Ezért átváltottam Visual CLI C++-ra. Két hét után rájöttem, hogy ez annyira különbözik az általam használt natív és Visual C++-tól, hogy nem sokat haladtam benne. Mivel a Műszaki képfeldolgozás órára írt lézerszkenneremet is C#-ban fejlesztettem, végül most is emellett tettem le a voksot, így gyorsabban haladhattam mind a Form-okkal, mind az algoritmusokkal.

A megjelenítő programomat pedig Visual C++-ban írtam, DirectX 9.0c API-t használva. Azért írtam erre külön programot, mert a C#-hoz nem ismerem az XNA-t, így nem tudok .NET-ből DirectX-el fejleszteni, továbbá ha a processzor számítja ki a pontfelhő pontjait, több ezer pont esetén komoly lassuláshoz vezethet a megjelenítés. DirectX segítségével viszont közvetlenül a videókártya számol.

3.2 Algoritmusok megvalósítása

A szoftverben több dolgot nem teljesen úgy oldottam meg, ahogyan az elméleti részben leírtam, ennek gyakran optimalizálási okai voltak, vagy a számítási módszerek elmélete és gyakorlata megvalósításaiban eltértek egymástól. Az ilyen eseteket külön jelzem.

A képfeldolgozásnál használtam az úgynevezett `unsafe` blokkot. Egy C# nyelven írt program egy interpretter segítségével fut, ezért nincsenek mutatók. Ha egy képet tömbként kell bejárni, meglehetősen lassú lesz az ezt végrehajtó algoritmus, de más területeken sem lehet megfelelő gyorsaságot elérni. Ezt hivatott feloldani az `unsafe`

blokk. A fordító ezen blokkon belül engedélyezi a mutatók használatát, így lényegesen gyorsabb lehet a nagy tömbök, vagy képek bejárása.

3.2.1 Rendszer geometriai kalibrációja

Az algoritmus csak akkor működik, ha ismert a kamera-mérőrács távolság, a kamera-projektor távolság, a kamera és az objektív párosításából adódó látószög és a szekvenciák száma. Ezekből a paraméterekből számítja ki a szükséges adatokat, a későbbi számításokhoz.

Az algoritmusban szereplő változókkal a következő adatokat ábrázoltam:

`g_iSysC`:	kamera és a mérőrács közötti távolság, mm-ben.
`g_iSysM`:	kamera és a projektor közötti távolság, mm-ben.
`g_dSysPerspective`:	látószög, fokban.
`g_iSysSequences`:	szekvenciák száma.
`g_dGamma`:	kamera és a projektor optikai tengelyei között bezárt szög.
`g_dSysP`:	a projektor távolsága a mérőrácstól.
`_dAlfa`:	a kamera érzékelőjének szélessége és magassága, valamint az átlója által alkotott derékszögű háromszög hosszabbik befogóján lévő hegyesszög.
`_dBeta`:	az előző háromszög másik szöge.
`g_dSysPersRad`:	látószög, radiánban.
`_dPixelMM`:	a képátló képzeletbeli pixeleinek hossza, mm-ben.
`g_dPixelWidthMM`:	a kép valós pixeleinek hossza, mm-ben.

Az algoritmus pedig:

```
g_dGamma = Math.Atan( (double)g_iSysM / (double)g_iSysC );
g_dSysP = Math.Sqrt( Math.Pow( (double)g_iSysC, 2 )
          + Math.Pow( (double)g_iSysM, 2 ) );

double _dAlfa =
          Math.Asin( 1 / ( Math.Sqrt( Math.Pow( g_iGridWidth, 2 )
          + Math.Pow( g_iGridHeight, 2 ) ) ) ) * g_iGridHeight;
double _dBeta = ( Math.PI / 2 ) - _dAlfa;

g_dSysPersRad = ( Math.PI / 180 ) * g_dSysPerspective;
double _dPixelMM = 2 * ( g_iSysC / Math.Sin( ( Math.PI / 2 )
          - ( g_dSysPersRad / 2 ) ) )
          * ( Math.Sin( g_dSysPersRad / 2 ) )
          / ( Math.Sqrt( Math.Pow((double)g_iGridWidth, 2 )
          + Math.Pow(    (double)g_iGridHeight, 2 ) ) );
g_dPixelWidthMM = _dPixelMM * Math.Sin( _dBeta );
```

3.2.2 Vetítő form és működése

A szoftver fényforrásként egy projektort képes vezérelni. Hogy a projektoron megjelenjen a megfelelő szekvencia, két külön form-ra van szükség. Az egyik a teljes képernyőt betölti (fullscreen), a másik pedig vezérli azt. A megfelelő működéshez a vetítőn megjelenő form példányát át kell adni a vezérlő form konstruktorának, példányosításkor:

```
g_objProjectorScreen = new fProjectorScreen();
g_objProjectorScreen.setProjection( 7, 3 );
g_objProjScreenController =
         new fProjectorController( g_objProjectorScreen );
g_objProjectorScreen.Show();
g_objProjScreenController.Show();
```

A vezérlő form-ban meg lehet adni, hogy szekvenciákat, vagy a kalibráláshoz szükséges célkeresztet vetítse, továbbá be lehet határolni a vetítés területét. Ez a funkció akkor hasznos, ha a tárgy jóval kisebb, mint amekkora képet vetít a projektor. Mivel ezen beállítások egyértelműek (csak változókat küld át a megjelenítő form-nak), algoritmusait nem részletezem.

A vetítő form első funkciója, hogy képes legyen felismerni a kiterjesztett tálcát, több monitor esetén. Egyszerűen lekérdezi, hogy hány megjelenítő eszközt érzékel a Windows és a form-ot a legutolsón jeleníti meg:

```
this.Location = new Point( screens[screens.Length - 1].Bounds.X,
        screens[screens.Length - 1].Bounds.Y );
```

A form legfontosabb funkciója, a szekvenciák megjelenítése. Az aktuális szekvencia a 2.3.1-es fejezetben tárgyalt módon, Gray-kódot használva jelenít meg fekete- és fehér területeket. Ehhez olyan algoritmust írtam, amely a teljes területet meghatározott szélességű részterületekre osztja, majd ezek színeit a Gray-kód szerint állítja be, a 2. táblázat szerint soronként, tehát mindegyik szekvencia a 2. táblázat egy-egy sorait ábrázolja, 4 szekvencia esetén. Fontos még megjegyeznem, hogy az algoritmusom annyiban tér el az elmélettől, hogy a színeket megfordítottam technikai okokból.

Az említett területek szélességét meghatározó kód:

```
int _iCodeMax = (int)Math.Pow( 2, _iSequence );
double _dCodeWith = ( (double)1 / (double)_iCodeMax )
          * (double)( g_iProjResX - g_iStart - g_iEnd );
Size _objCodeSize =
      new Size( (int)_dCodeWith, g_iBottom - g_iTop );
```

Az `_iSequence` az aktuális szekvenciát jelöli, a `g_iProjResX` a projektor vízszintes felbontását. A `g_iStart` jelöli a terület kezdetét, amit a projektor kivetít, a `g_iEnd` pedig a végét jelöli, vízszintesen, a `g_iBottom` és a `g_iTop` ugyanezt függőlegesen. Az `_objCodeSize` határozza meg a terület méretét.

Ezt követően rajzolja ki az algoritmus a területeket a megfelelő színnel a form területére, ehhez azonban tudni kell, hogy melyik területnek mi a színe.

Az algoritmus:

```
for( int i = 0; i < _iCodeMax; i++ )
{
     if( i == 0 )
          _bIsWhite = true;           // ha i = 0, akkor fehér
     else if( i == 1 )
          _bIsWhite = false;          // ha i = 1, akkor fekete
     if( i % 2 != 0 )
          _bIsWhite = !_bIsWhite;     // ha i páratlan, akkor váltunk
     Point p =
          new Point( (int)Math.Round( _dCodeCoord ),
                                    g_iTop );
     _dCodeCoord += _dCodeWith;
     if( _bIsWhite ) pen = g_pPen;
     else pen = new Pen( Color.Black );
     Rectangle gray = new Rectangle( p, _objCodeSize );
     if( _bIsWhite ) g.FillRectangle( g_brColor, gray );
     else g.FillRectangle( Brushes.Black, gray );
}
```

Hogy a form a követelménynek megfelelően a teljes képernyőt elfoglalja, a Visual Studio design nézeténél be kell állítani a megfelelőket, ezek a következők:

- FormBorderStyle = None,
- AutoScroll = False,
- StartPosition = Manual,
- WindowsState = Maximalised,
- ControlBox = False.

Ezekkel a beállításokkal és algoritmusokkal a szoftver képes arra, hogy mérőrácsot vetítsen a mérendő tárgy felületére. A felhasználó feladata már csak a képek elkészítése valamilyen képalkotó eszközzel.

3.2.3 Szűrők

A program két szűrőt használ: Gauss simítót és a szintre vágást.

A szintre vágás egy lineáris eljárás. Egy küszöbértéket használ és ha egy képen bármelyik pixel színértéke ez alatt van, annak színe fekete lesz, ellenkező esetben fehér. Elmondhatjuk, hogy egy 1 bites kép jön létre.

A szűrő leegyszerűsített algoritmusa:

```
public static void Limit( Bitmap a, int _iLimit )
{
     byte grey = 0;
     BitmapData amData =
          a.LockBits( new Rectangle( 0, 0, a.Width, a.Height ),
          ImageLockMode.ReadWrite, PixelFormat.Format24bppRgb );
     int stride = amData.Stride;
     System.IntPtr aScan0 = amData.Scan0;
     unsafe
     {
          byte* ap = (byte*)(void*)aScan0;
          int anOffset = stride - a.Width * 3;
          for( int y = 0; y < a.Height; ++y )
          {
               for( int x = 0; x < a.Width; ++x )
               {
                    grey =
                         (byte)( 0.11 * ap[0]
                              + 0.59 * ap[1]
                              + 0.3 * ap[2] );
                    if( grey > _iLimit) grey = 255;
                    else grey = 0;
                    ap[0] = grey;
                    ap[1] = grey;
                    ap[2] = grey;

                    ap += 3;
               }
          ap += anOffset;
          }
     }
     a.UnlockBits( amData );
}
```

A Gauss szűrő egy konvolúciós függvény, ennek elméleti részét leírtam a 2.4.2-es fejezetben. Gyakorlati megvalósítását nem közlöm a szakdolgoza-

tomban, mivel terjedelmes és nem saját, Christian Graus alkotása (http://www.codeproject.com/KB/GDI-plus/csharpfilters.aspx).

3.2.4 Szekciók és élek beazonosítása

A szekciók beazonosításának lehetősége a program lelke, a legfontosabb része, ez alapján lehet elkészíteni a mérőrácsot és meghatározni a tárgyakon létrejött vonalakat. A szekciókat és éleket beazonosító algoritmusokat külön függvénybe szedtem.

A szekciókat beazonosító algoritmus egyszerre mindig egy képet dolgoz fel. Ezért tudnia kell, hogy hányadiknál tart és összesen hány darab van. Az első a nulladik kép. Ahogy korábban írtam, ez egy a projektorból kivetített fehér kép, ami abban nyújt komoly segítséget, hogy a képek vizsgálata előtt meghatározzuk, melyik képterületekkel nem kell foglalkozni, ezzel is növelve a feldolgozás sebességét. Ezen segít még egy úgynevezett ROI (Region Of Interest), aminek segítségével tovább szűkíthető a vizsgálati terület. A ROI csúcsainak pozícióit a felhasználó állítja be.

Az algoritmus futtatása előtt létre kell hozni egy pontosan akkora tömböt, amekkorák a vizsgált képek méretei. Egész típusúnak kell lennie, azokkal fogja ábrázolni, hogy az egyes pixelek melyik szekciókhoz tartoznak.

Végül ez az algoritmus is tartalmaz egy szintre vágást.

Először is meg kell határozni a ROI-t alkotó egyeneseknek az egyenleteit, majd meg kell határozni, hogy az egyes sorokban, vagy oszlopokban hol vannak az általuk meghatározott határértékek:

```
double _dFx0 =
     (double)( _points[3].Y - _points[0].Y )
     / (double)( _points[3].X - _points[0].X );
double _dFx1 =
     (double)( _points[1].X - _points[0].X )
     / (double)( _points[1].Y - _points[0].Y );
double _dFx2 =
     (double)( _points[2].Y - _points[1].Y )
     / (double)( _points[2].X - _points[1].X );
double _dFx3 =
     (double)( _points[2].X - _points[3].X )
     / (double)( _points[2].Y - _points[3].Y );
int[] _aiLimitFx0 = new int[a.Width];
int[] _aiLimitFx1 = new int[a.Height];
int[] _aiLimitFx2 = new int[a.Width];
int[] _aiLimitFx3 = new int[a.Height];
```

```
for( int i = 0; i < a.Width; i++ )
{
     _aiLimitFx0[i] =
          (int)( ( i - _points[0].X ) * _dFx0 + _points[0].Y );
     _aiLimitFx2[i] =
          (int)( ( i - _points[1].X ) * _dFx2 + _points[1].Y );
}
for( int i = 0; i < a.Height; i++ )
{
     _aiLimitFx1[i] =
          (int)( ( i - _points[0].Y ) * _dFx1 + _points[0].X );
     _aiLimitFx3[i] =
          (int)( ( i - _points[3].Y ) * _dFx1 + _points[3].X );
}
```

Ezután létrehozzuk a tömböket, melyek alapján egyértelműen be lehet azonosítani a szekciókat és kiszámítjuk a fontosabb változókat:

```
int _iPlacesLenght = (int)Math.Pow( 2, _iSequence - 1 );
int[] _aiPlaces = new int[_iPlacesLenght];
int[] _aiPrePlaces = new int[_iPlacesLenght];
_iMaxPlace = (int)Math.Pow( 2, iMaxSequence );
int _iDiff =
     (int)( (double)( _iMaxPlace - 1 )
     / Math.Pow( 2, _iSequence - 2 ) );

bool[] _abIsWhite = new bool[_iPlacesLenght];
_bIsWhite = true;
for( int i = 0; i < _abIsWhite.Length; i++ )
{
     if( i == 0 ) _bIsWhite = true;
     else if( i == 1 ) _bIsWhite = false;
     else if( i % 2 != 0 ) _bIsWhite = !_bIsWhite;
     _abIsWhite[i] = _bIsWhite;
}
```

Ezt követően a 2.4.2-es fejezetben bemutatott módszer alapján feltöltjük azt a tömböt, ami az előző szekvencia szekcióit ábrázolja:

```
for( int i = 0; i < _iPlacesLenght; i++ )
{
     if( i == 0 ) _aiPrePlaces[i] = 0;
     else
     {
          if( i % 2 != 0 )
               _aiPrePlaces[i] = _aiPrePlaces[i - 1] + _iDiff;
          else
               _aiPrePlaces[i] = _aiPrePlaces[i - 1] + 1;
     }
}
```

A tömbök feltöltése után, az algoritmus elkezdi a képek vizsgálatát, kezdve a nulladik szekvenciával a ROI-t figyelembe véve, amiről a következő kódrész gondoskodik:

```
if( _iSequence == 0 )
{
     if( ap[2] > _iLimit
          && y > _aiLimitFx0[x]
          && x > _aiLimitFx1[y]
          && y < _aiLimitFx2[x]
          && x < _aiLimitFx3[y] )
               _aiSections[x][y] = 0;
     else
               _aiSections[x][y] = -2;
}
```

A nulladik kép után egyszerűbb, ha az első képpel egy kódrész külön foglalkozik. Itt viszont már figyelni kell arra, hogy az adott pixel vizsgálható-e, vagy sem (vagyis az aktuális szekciója nem -2).

```
else if( _iSequence == 1 && _aiSections[x][y] != -2 )
{
     if( _aiSections[x][y] == 0 && ap[2] > _iLimit )
          aiSections[x][y] = 0;
     else
          _aiSections[x][y] = _iMaxPlace - 1;
}
```

És végül az összes többit elemzi ki:

```
else if( _iSequence > 1 && _aiSections[x][y] != -2 )
{
     for( int i = 0; i < _aiPrePlaces.Length; i++ )
     {
          if( _aiSections[x][y] == _aiPrePlaces[i] )
          {
               if( ap[2] > _iLimit )
                    _bIsWhite = true;
               else
                    _bIsWhite = false;

               if( _bIsWhite == _abIsWhite[i * 2] )
                    _aiSections[x][y] = _aiPlaces[i * 2];
               else _aiSections[x][y] = _aiPlaces[( i * 2 ) + 1];
          }
     }
}
```

Miután az utolsó képre is lefutott az algoritmus, a `_aiSections` kétdimenziós tömb a pixelek egyértelműen elkülönített szekcióbesorolásait fogja tartalmazni. Ez alapján el lehet kezdeni az élek megkeresését és beazonosítását. Ehhez azonban létre kell hozni egy új tömböt, ami már csak és kizárólag a vonalak pontjait fogja tartalmazni, tehát kisebb lesz, mint a szekciókat ábrázoló tömb:

```
int[][] g_aiEdges = new int[g_iGridHeight][];
int _iMaxEdge = (int)Math.Pow( 2, g_iSysSequences ) - 1;
for( int i = 0; i < g_aiEdges.Length; i++ )
{
     g_aiEdges[i] = new int[_iMaxEdge];
}
```

Az élek beazonosítása pedig így megy végbe:

```
for( int y = 0; y < g_aiEdges.Length; y++ )
{
     for( int x = 0; x < _aiSections.Length - 1; x++ )
     {
          if( _aiSections[x + 1][y] - _aiSections[x][y] == 1 )
               g_aiEdges[y][(int)_aiSections[x][y]] = x;
     }
}
```

Az élek meghatározását követően, már csak össze kell hasonlítani a mérőrács pontjaival, amihez előbb a mérőrács éleit kell felismerni, majd el kell végezni a megfelelő számításokat.

3.2.5 Mérőrács felismerése

A mérőrács mérőegyeneseinek meghatározása ugyanúgy megy végbe, mint ahogy az előző pontban bemutattam, annyival kiegészítve, hogy itt az éleknek csak a legelső és a legutolsó pontjait kell figyelembe venni, mivel valós egyeneseknek kell létrejönniük. Erre azért van szükség, mert könnyen előfordulhat, hogy a referenciasík felülete egyenetlen és a rajta létrejövő mérőrács egyenesei nem egyenesek, ezzel rontva a mérés pontosságát. Ezt az alábbi algoritmus küszöböli ki:

```
// mindegyik él legfelső pontjának megkeresése
for( int x = 0; x < _iMaxEdge; x++ )
{
```

```
     for( int y = 0; y < g_aiEdges.Length; y++ )
     {
          if( g_aiEdges[y][x] != 0 )
          {
               g_apointGrid[x][0].X = g_aiEdges[y][x];
               g_apointGrid[x][0].Y = y;
               break;
          }
     }
}
// mindegyik él legalsó pontjának megkeresése
for( int x = 0; x < _iMaxEdge; x++ )
{
     for( int y = g_aiEdges.Length - 1; y >= 0; y-- )
     {
          if( g_aiEdges[y][x] != 0 )
          {
               g_apointGrid[x][1].X = g_aiEdges[y][x];
               g_apointGrid[x][1].Y = y;
               break;
          }
     }
}
```

Ettől kezdve meg van a pontos mérőrács, így van mihez viszonyítani a képekből kinyert vonalak pontjait.

3.2.6 Pontok mélységi információjának meghatározása

Ha már meg van a mérőrács és a tárgy felületén létrejövő vonalak pontjai, a kalibrációkor megadott adatok alapján meghatározható a pontfelhő pontjainak helyzete. Erre a 2.5.1-es pontban bemutatott egyenletet használtam fel, de csak a mérőrács pontjainak meghatározásáig, mivel a mélységi adat kinyerése függ a vizsgált pont optikai tengelytől való helyzetétől, továbbá általam eddig be nem azonosított hiba miatt értelmezhetetlen formájú pontfelhő jött létre. Ezért, mivel a mérőrács megfelelő helyzetének kiszámítása megvalósult, azt felhasználva ideiglenes becslést alkalmaztam a pontok projektortól való távolságára.

A mérőrács pontjainak helyzetét a következő algoritmus határozza meg:

```
g_dH1 = new double[g_pointGrid.Length][];
for( int i = 0; i < g_pointGrid.Length; i++ )
{
     g_dH1[i] = new double[g_iGridHeight];

     double _dGridFX =
          (double)( g_pointGrid[i][1].X - g_pointGrid[i][0].X )
          / (double)( g_pointGrid[i][1].Y - g_pointGrid[i][0].Y );
```

```
     double _dGridX = g_pointGrid[i][0].X;
     for( int j = g_pointGrid[i][0].Y;
          j <= g_pointGrid[i][1].Y;
          j++ )
     {
          double Diff1 = _dGridX - ( (double)g_iGridHeight / 2 );
          double D1C = Math.Sqrt( Math.Pow( Diff1, 2 )
               + Math.Pow( (double)g_iSysC, 2 ) );
          double Alfa1 = Math.Asin( ( 1 / D1C ) * (double)g_iSysC );
          double Omega1 = Math.PI - Alfa1 - g_dGamma;
          g_dH1[i][j] = ( ( Diff1 / Math.Sin( Omega1 ) )
               * Math.Sin( Alfa1 ) );
          _dGridX += _dGridFX;
     }
}
```

A `g_dH1` tömb tartalmazza a mérőrács egyeneseinek a kezdő és végpontja közötti pontokat. A kód először meghatározza az egyenesek egyenleteit, majd minden egyes sorban a 2.5.1-ben tárgyalt módon számítja ki a pontokat.

Ezután már csak a képekből kinyert pontokból és a mérőrács pontjaiból kell meghatározni a mélységi információkat, ezt a következő algoritmus teszi meg:

```
for( int i = 0; i < g_pointGrid.Length; i++ )
{
     for( int j = g_pointGrid[i][0].Y; j
          <= g_pointGrid[i][1].Y; j++ )
     {
          double Diff2 = g_aiEdges[j][i]
               - ( (double)g_iGridHeight / 2 );
          double D2C = Math.Sqrt( Math.Pow( Diff2, 2 )
               + Math.Pow( (double)g_iSysC, 2 ) );
          double Alfa2 =
               Math.Asin( ( 1 / D2C ) * (double)g_iSysC );
          double Omega2 = Math.PI - Alfa2 - g_dGamma;
          double H2 =
               ( ( Diff2 / Math.Sin( Omega2 ) )
               * Math.Sin( Alfa2 ) );
          g_adDepth[i][j] = ( ( H2 - g_dH1[i][j] )
               / Math.Cos( g_dGamma ) );
     }
}
```

A `g_adDepth` tömb tartalmazza minden pont távolságát a projektortól. A becslés pedig úgy működik, hogy a tárgy felületén lévő adott pontjából kivonja a megfelelő mérőegyenes pontját, majd a hosszat elosztja a projektor optikai tengelye és a kamera optikai tengelye által bezárt szög koszinuszával. Azért ezzel a szöggel, mert bármely

pontot felvéve a referencia síkon, ha erre a pontra mind a kamera-, mind a fényforrás pontjából egy egyenes indul ki, a két egyenes közötti szög közel egyenlő lesz az említett szög nagyságával.

Mivel a képelemzést végző- és a megjelenítő program két különálló, valahogy az adatokat át kell vinni közöttük. Mivel a 2.1-es pontban bemutatott 3D szkennelő szoftvernél ugyanezt a helyzetet úgy oldottam meg, hogy a képfeldolgozó program a pontokat egy fájlba írja, ennél a projektnél is így oldottam meg. Erre a feladatra *objects.txt* fájlt hoz létre, aminek a fájlszerkezete a következő:

```
x[egész szám]y[egész szám]z[egész szám];
```

Amikor elkészült a pontfelhő kiszámítása és kiírta a program ebbe a fájlba, el lehet indítani a megjelenítő programot, a *DXNezet.exe*-t.

3.2.7 Megjelenítő program DirectX segítségével

Ezt a programot már a lézeres szkennernél is használtam, itt csak apró módosításokat kellett véghezvinnem. A program alapját Nyisztor Károly – Grafika és játékprogramozás DirectX-szel c. könyve egyik példaprogramja adta. Két függvényt kellett módosítanom: az `initVB()`–t és a `render()`–t.

Az `initVB()` a vertexbuffer feltöltéséért felel, azaz a pontokat tartalmazó területeket foglalja le a videómemóriában, amiket később el kell érnie a programnak, a különböző transzformációk és a megjelenítés érdekében.

A függvény így néz ki:

```
HRESULT initVB()
{
     // ez a pointer a vertexpuffer adatterületére fog mutatni
     VERTEX* aVertPtr = NULL;
     int _iR = 0, _iG = 0, _iB = 0;

     // 1) pontlista
     VERTEX aPointList[32768];
     int iPointDelay[32768];

     std::ifstream filePoints;
     filePoints.open("objects.txt");

     if( filePoints.fail() )
     {
          return false;
     }
```

```
    else
    {
         int i = 0;
         while( !filePoints.eof() )
         {
              char cBuffer[50];
              filePoints.getline( cBuffer, sizeof( cBuffer ) );

              if( cBuffer[0] == 'x' )
              {
                   aPointList[i].x =
                        toInt( cBuffer, 1,
                             WhereIsThis( cBuffer, 1, 'y' ) );
                   aPointList[i].y =
                        toInt( cBuffer,
                             WhereIsThis( cBuffer, 1, 'y' )
                             + 1, WhereIsThis( cBuffer, 1, 'z' ) );
                   aPointList[i].z =
                        toInt( cBuffer,
                             WhereIsThis( cBuffer, 1, 'z' )
                             + 1, WhereIsThis( cBuffer, 1, ';' ) );
                   _iR = 255;
                   _iG = 255;
                   _iB = 255;
                   aPointList[i].color =
                        D3DCOLOR_XRGB( _iR, _iG, _iB );
                   i++;
              }
              g_bPointX = aPointList[0].x;
         }
    }
    filePoints.close();

    // vertex puffer létrehozása
    if( FAILED( g_D3DDevicePtr->CreateVertexBuffer(
                                32768 * sizeof( VERTEX ),
         0,
         D3DFVF_SAJAT_VERTEX_FORMATUM,
         D3DPOOL_DEFAULT,
         &g_PointListVB,
         NULL ) ) )
    {
         return E_FAIL;
    }

    // megszerezzük majd zároljuk a vertex puffer
    // memóriaterületét, hogy átmásolhassuk a vertex adatokat
    if( FAILED( g_PointListVB->Lock( 0, sizeof( aPointList ),
         (VOID**)&aVertPtr, 0 ) ) )
    {
         return E_FAIL;
    }

    // vertex adatok másolása
    memcpy( aVertPtr, aPointList, sizeof( aPointList ) );
```

```
// vertex puffer zárolásának feloldása
     g_PointListVB->Unlock();

     return S_OK;
}
```

A program a fájlból való adatok kinyerésére saját függvényeket használ, név szerint a `toInt()`-et, amely karaktersorozatot egész számmá alakít és a `WhereIsThis()`-t, amely egy karakternek megadja a helyét egy karaktertömbön belül.

A `render()` függvényen belül a pontok kirajzolásáért pedig a következő két függvény felel:

```
g_D3DDevicePtr->SetStreamSource( 0,
                          g_PointListVB, 0, sizeof(VERTEX) );
g_D3DDevicePtr->DrawPrimitive( D3DPT_POINTLIST, 0, 32768 );
```

A program végül képes megjeleníteni 32768 pontot és azokat lényegesen gyorsabban képes megjeleníteni, mintha helyzeteik csak CPU-ról lennének kiszámítva. Ez azonban sosem fog megvalósulni, mert a fájlos megoldás hátránya a fájl mérete, 8000 pont felett már nem képes megnyitni azt a program.

3.3 A szoftver használata, röviden

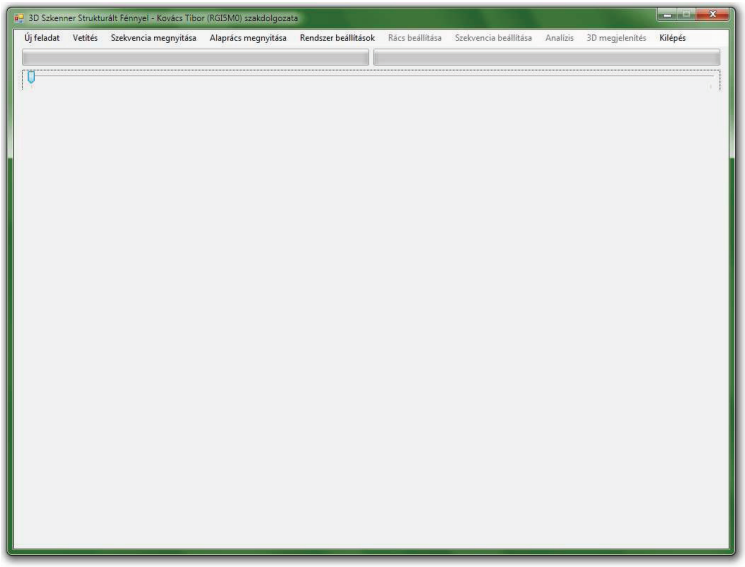
Miután leírtam a működését, bemutatom a használatának módját.

3.3.1 Rendszer beállítása, képek létrehozása

A következő dolgokra mindenképpen szükség lesz:

- projektorra,
- valamilyen képalkotó eszközre (lehetőleg fényképezőgépre),
- stabil helyre, amire rá lehet tenni a projektort és a képalkotó eszközt,
- egy olyan helyre, amire stabilan rá lehet helyezni a mérendő tárgyat, vagy személyt,
- egy matt, síkfelületre, amire a mérőrácsot vetítjük,
- távolságmérő eszközökre (mérőszalagra, vagy vonalzóra).

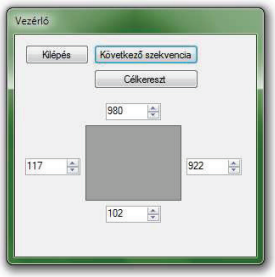
Először érdemes felállítani a mérendő objektum helyét, amire rá tudjuk helyezni azt (például személy esetén egy szék). Ezzel szemben kell elhelyezni a képalkotó eszközt és fel kell jegyezni a távolságát, mm-ben. Majd úgy kell a képalkotó eszköz mellett elhelyezni a projektort, hogy az objektum és a kamera közötti egyenesre merőleges legyen a kamera és a projektor közötti egyenes. Végül fel kell helyezni a mérő helyre a sík, matt tárgyat, ami lapjával pontosan a projektor felé kell nézzen.



12. ábra – Üres projekt

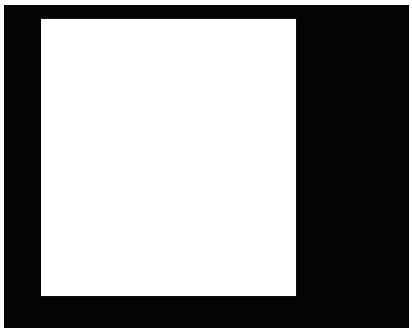
Ekkor indítsuk el a szoftvert és azon belül a „Vetítés” menüpont alatt előjön egy új ablak, a projektor vetített képe pedig átáll feketére. Ekkor a monitoron lévő ablakon a „Következő szekvencia” gomb segítségével az első szekvenciát vetíti a projektor. Addig kell nyomkodni, amíg a vetített kép fehér nem lesz. Ilyenkor a gomb alatti számokkal be kell állítani, hogy a projektor mekkora területre vetítsen. Erre azért van szükség, mert a tárgy nem minden esetben foglalja el azt a területet, amit a projektor képe át tud fogni az adott távolságban.

Ezután a „Célkereszt” gomb megnyomásával láthatóvá válik a vetített kép középpontja a már felhelyezett egyszínű, matt, sík tárgyon, ez lesz a viszonyítási pont. Erre kell beállítani a képalkotó eszköz középpontját.

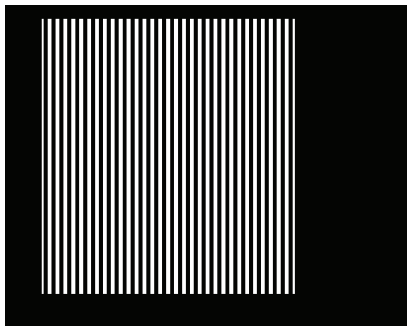


13. ábra – Vezérlő

A célkereszt beállítása után lehet kezdeni a képek elkészítését. Először a fehér képernyőt kell rávetíteni a síkra, majd lehet exponálni (a fényképezőgép lehetőleg manuális beállításon működjön). A nulladik kép elkészülte után a „Következő szekvencia” gomb lenyomásával az első szekvencia vetül a síkra, majd ekkor is exponálni kell. Ezt a műveletet annyiszor kell elvégezni, amíg újra fehér képernyő vetül a síkra, erről azonban már nem kell felvételt készíteni. Ezután ugyanezt a műveletsort (tehát a nulladik szekvenciától az utolsóig) el kell készíteni a tárgyra. Majd a képeket fel kell tölteni a számítógépre.



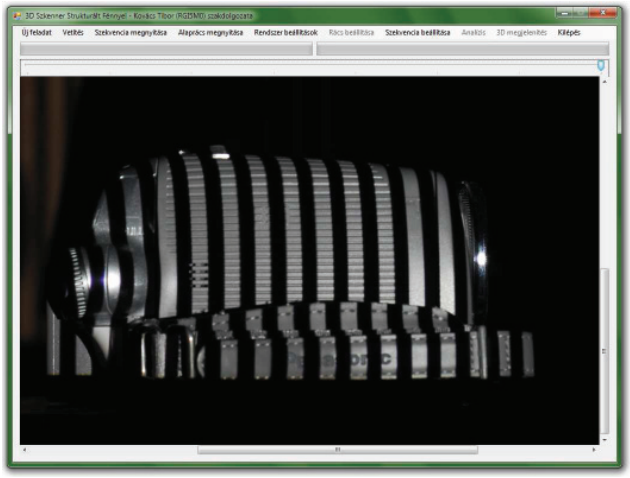
14. ábra – Nulladik szekvencia



15. ábra – Vetített szekvencia

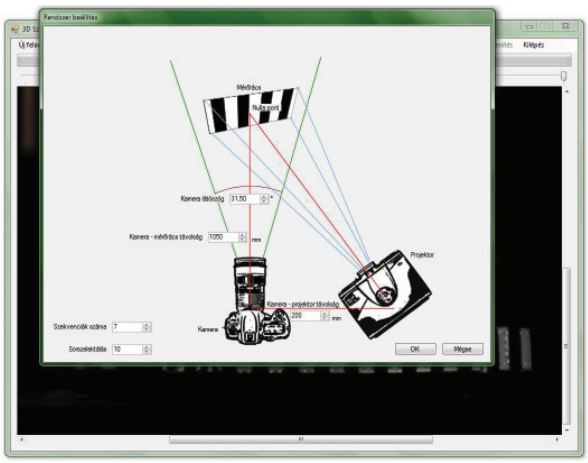
3.3.2 Pontfelhő létrehozásának menete

Az elkészült képeket be kell tölteni a szoftverbe. A „Szekvencia megnyitása” menüponttal a tárgyról készült felvételeket lehet betölteni, ezek láthatóvá is válnak a program ablakának közepén. Majd az „Alaprács megnyitása” menüponttal a mérőrácsról készült felvételek is megnyithatók, de ezek a főablakban még nem láthatók. A program ellenőrzi, hogy a képek felbontásai megegyeznek-e, így nagy eséllyel kivédi a rossz képek megnyitását.



16. ábra – Főablak, a tárgyról készült felvételek megnyitása után

Ezután a „Rendszer beállítások” menüponttal be lehet állítani a mérőrendszer geometriáját. Itt kell megadni a kamera-mérőrács- és a kamera-projektor távolságot, valamint a kamera objektívjének látószögét. Itt lehet módosítani a szekvenciák számát, ami alapértelmezetten 7. Amikor a program elkészíti a pontfelhőt, azokat kiírja egy fájlba. Mivel a fájl beolvasásának sikeressége mérethez kötött, ezért érdemes spórolni a beírt pontokkal. Ehhez nyújt segítséget a „Sorszelektálás” beállítás. Mivel a mérővonalak függőlegesek, a pontok érzékelése sorfolytonos, ezért lényegesen több függőleges pont jön létre, mint vízszintes. A tárgyalt beállítási lehetőség megmondja a programnak, hogy hány darab pontsort hagyjon ki két pontsor beírása között.



17. ábra – Rendszerbeállítások ablak

A rendszerbeállítások végeztével el kell készíteni a mérőrácsot. Ehhez rá kell kattintani a „Rács beállítása” menüpontra. Ekkor egy új ablak nyílik meg. Itt láthatóak egyedül a mérőrácsról készített felvételek. A képek szélein egy-egy vörös egyenes

látható, amik a 3.2.4-es fejezetben bemutatott ROI oldalai, ezzel lehet korlátozni azt a képterületet, amit vizsgálni kell. Ennek beállítása a zöld négyzetekkel történik. Az egérmutatót a négyzetek belsejébe kell vinni és a bal egérgomb folyamatos nyomva tartásával mozgatni. A ROI csúcsainak új helyzete a bal egérgomb felengedésével rajzolódik ki. A ROI-t úgy érdemes beállítani, hogy a nulladik szekvencián lévő fehér terület széleitől néhány pixelnyivel belül legyenek a vonalai. Erre azért van szükség, mert előfordulhat, hogy a vetített szekvenciák a határ mentén nem határozott átmenetet mutatnak, hanem valamilyen görbét írnak le, így pontatlanabb lehet a kész mérőrács.

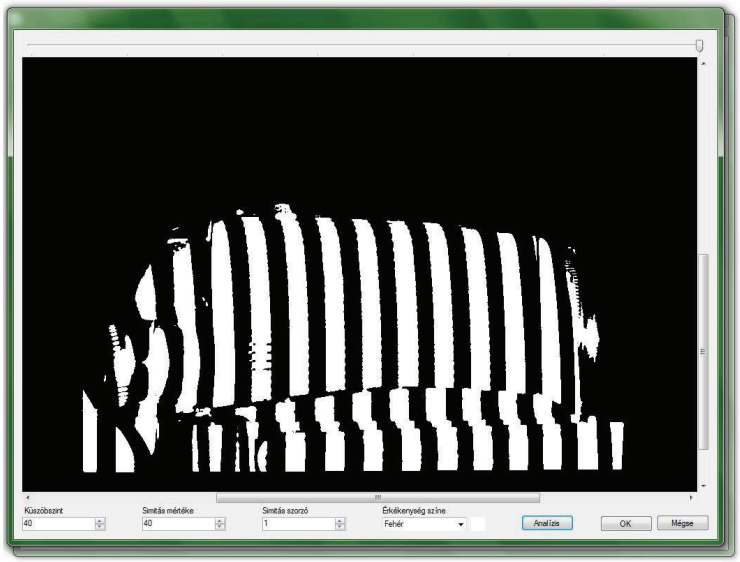
A négyszög beállítása után meg kell határozni a küszöbértéket. 0 és 255 érték között lehet választani, de érdemes a középértéket beállítani. Szélsőséges értékek környékén a mérőrács egyenesei szabálytalanul eltolódhatnak.

Ezek végeztével az „Analízis” gomb megnyomásával a program elkezdi kiszámítani a mérőrácsot. A művelet folyamatát egy progress bar mutatja. Amikor elkészült, a képen a mérőrács egyenesei jelennek meg a képeken. Ha valamiféle pontatlanság következne be, az egyenesek végpontjai módosíthatóak, a határoló négyszöghöz hasonlóan.

18. ábra – Az elkészült mérőrács

A mérőrács elkészülte az „OK” gombbal nyugtázható. Ezután a „Szekvencia beállítása” menüponttal egy új ablakban a tárgyról készült képek analízisének beállításait lehet megadni.

A küszöbszinttel a 0 és 255 érték közé eső értéket kell megadni, amitől vizsgálja a program, hogy az adott pixel fekete, vagy fehér. A simítás mértéke 0 és 100 közé eső szám. Ez felel a kép zajszűréséért. Ha szükség van simításra, akkor a szorzót legalább 1-re kell állítani. Ez meghatározza, hogy a szűrő hányszor fog a képen lefutni. Az érzékenységi szint akkor kell változtatni, ha a vetített szekvenciákon a nem fekete terület színe eltér a fehértől (ami lehet vörös, zöld, vagy kék). A beállítások az „Analízis" gomb segítségével ellenőrizhetők. Érdemes mindent úgy beállítani, hogy minden képen a fehér területek jól látszódjanak. Itt is van lehetőség korlátozó négyszöget használni. A beállításokat az „OK" gomb megnyomásával alkalmazza a szoftver.

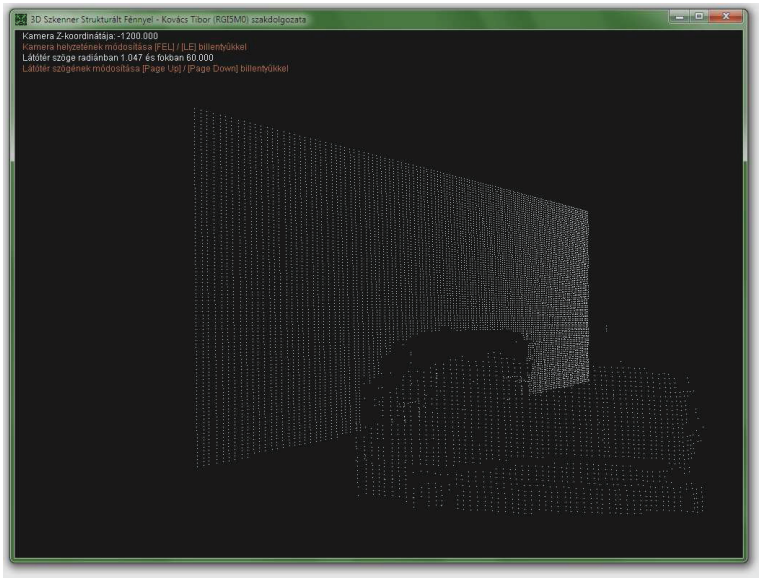


19. ábra – Képek beállítása

Miután minden beállítás megtörtént, a főablakon az „Analízis" menüponttal elkezdődik a szekciók felismerése, az élek meghatározása és beazonosítása, a pontfelhő mélységi információjának a kiszámítása és az adatok fájlba írása.

3.3.3 Elkészített pontfelhő megjelenítése

Az elkészített pontfelhő a főképernyő „3D megjelenítés" menüpontjával jeleníthető meg. Ekkor elindul a *3DNezet.exe* program, amiben a pontfelhő pontjai látszanak.



20. ábra – Pontfelhő megjelenítése

A megjelenítőben lévő virtuális kamera távolságát a [FEL] és a [LE] karaktermozgató gombokkal lehet módosítani. A [BAL] és a [JOBB] gombokkal a vízszintes látószöget lehet változtatni. A [PAGE UP] és a [PAGE DOWN] gombok pedig a kamera látószögét változtatják. Az [ESC] billentyűvel be lehet zárni az ablakot.

3.4 További tervek a szoftverrel

Mint látható, a szoftver még közel sincs kész állapotban. A mostani verzió egy tanulmánynak is felfogható, mivel nem is a tervezett nyelven írtam. A legfontosabb célom a szoftver tökéletesítése. Meg fogom oldani a pontok helyzetének pontos meghatározását, valamint implementálni fogok egy térhálót készítő algoritmust, ami komoly kutatási munkának ígérkezik.

A végleges szoftvert nem C#-ban, hanem natív C++-ban szeretném megírni, Win32 API felhasználásával. Ennek nagy előnye, hogy a DirectX megjelenítőt nem kell külön programként létrehozni, hanem beépíthető a programba. Így nem szükséges a pontokat csak azért egy fájlba írni, hogy azok megjeleníthetőek legyenek.

Fontos célnak tartom még, hogy a képek elkészítését és a projektor kezelését ne a felhasználó végezze, hanem a program maga. Ehhez meg kell ismerjem a Windows által felismerhető és kezelhető képalkotó eszközök vezérlésének technikáját.

4. Összegzés

Szakdolgozatom célja az volt, hogy bemutassam az általam fejlesztett 3D szkennert. Ez egy mérőeszköz, amivel egy tárgy felületéről viszonylag rövid idő alatt lehet több pont távolságról információt nyerni egy ponttól viszonyítva.

Mint ahogy korábban is írtam, napjainkban egyre nagyobb szükség van különböző fizikai objektumokról háromdimenziós modellt készíteni, a lehető legpontosabban. Ezek fontosak lehetnek a mérnököknek, a restaurátoroknak, helyszínelőknek, grafikusoknak, játékfejlesztőknek, stb... Sajnos a klasszikus kézi modellezés nem tud eléggé pontos lenni és nem elég gyors egy ilyen munka elvégzéséhez. Művészi feladatoknál még elfogadhatónak számít bizonyos pontatlanság, de sokszor gyorsan kell valósághű modelleket elkészíteni. Erre nyújt megoldást a 3D szkennelés. A reverse engeneering nagy lehetőséget ad a mérnököknek, hogy az általuk legyártott bonyolult geometriájú tárgyról a lehető legpontosabb méréseket elvégezhessék. Restaurátoroknak és régészeknek sem kell többé veszélyeztetni az értékes régészeti leleteket, vagy műkincseket vegyi anyagokkal, könnyen archiválhatják azokat egy 3D szkennelési módszerrel. A helyszínelők pedig „hazavihetik” a munkát, későbbi elemzésre, hogy újból bejárhassák a változékony bűnügyi helyszíneket.

Bemutattam, hogy milyen strukturált fényes megoldások léteznek, ezek előnyeit és hátrányait tárgyaltam. Például a vonallézeres nagyobb felbontású pontfelhőt képes létrehozni rövidebb idő alatt, mint a pontlézeres, de lassabb és kisebb felbontású a kódolt fényűnél. Kiemelten foglalkoztam ez utóbbival, a kódolt fényűvel, azon belül pedig a Gray-kódot alkalmazó módszerrel. Hogy mindezt könnyebben érthetővé tegyem, elvi és matematikai leírásokkal láttam el a szakdolgozat lapjait. Hogy be tudjam mutatni a módszer hatékonyságát, egy szoftvert is írtam hozzá. Ezen látható, hogy egyértelműen be lehet azonosítani több vonalat egy képen.

Szakdolgozatom egyik legfontosabb célkitűzése az volt, hogy egyetlen képen be lehessen azonosítani több különböző vonalat akkor is, ha azok annyira bonyolultan helyezkednek el, akár több megszakítással is, hogy azokat még az emberi szem sem tudná feltétlenül elkülöníteni; akkor sem, ha ismeri a kép létrejöttének körülményeit. Az irodalomból ismert, de nem kellően részletes leírásokhoz elvi és matematikai kiegészítéseket készítettem. A javasolt eljárást implementáltam és egy működő szoft-

vert készítettem a módszer működőképességének bizonyítására. A szoftver képes egy képen egyidejűleg több vonalat is azonosítani, bár néha ehhez további képek is szükségesek. A módszer és a szoftver fejlesztését folytatom, célom egy szinte teljesen automatizált működésű, egyszerűen kezelhető 3D szkenner berendezés kifejlesztése.

5. IRODALOMJEGYZÉK

Szerzők és műveik felsorolása a Word sablon „Irod” stílusában, a **szerzők szerinti betűrendben, pontos bibliográfiai adatokkal**, az útmutatóban leírt módon, az alábbi minták szerint.

Bakos Ferenc, Idegen szavak és kifejezések szótára, Akadémia kiadó, Budapest, 1986

Benkő Tiborné, Együtt könyebb a programozás C#, ComputerBooks Kiadó Kft, 2008

caryoko22, Build Your Own 3D Scanner: 3D Scanning with Swept-Planes, http://www.slideshare.net/dlanman/build-your-own-3d-scanner-3d-scanning-with-structured-lighting

Christian Graus, Image Processing for Dummies with C# and GDI+ Part 2 - Convolution Filters, http://www.codeproject.com/KB/GDI-plus/csharpfilters.aspx

Hoa G. Nguyen, Michael R. Blackburn, A Simple Method for Range Finding via Laser Triangulation, http://www.spawar.navy.mil/robots/pubs/td2734.pdf

Lasser András, Lézeres távolságmérés, mérési útmutató, Budapest Műszaki egyetem, 2000, http://3dmr.iit.bme.hu/edu/autroblab/3dscan/3dscanutmutato.pdf

Nyisztor Károly, Grafika és játékprogramozás DirectX-szel, SZAK Kiadó Kft, 2005

Műszaki képfeldolgozás, órai jegyzet, 2010/11-es tanév

http://en.wikipedia.org/wiki/Structured-light_3D_scanner

Nagy Krisztián, Fotóelmélet: Aliasing, anti-aliasing, moire, Pixinfo.com

http://pixinfo.com/cikkek/fotoelmelet_antialiasing

6. KÖSZÖNETNYILVÁNÍTÁS

Ezúton szeretnék köszönetet mondani:

- Témavezetőmnek, Dr. Végh Jánosnak a szakmai támogatásáért, javaslataiért, türelméért,
- Konzulensemnek, Dr. Cserháti Csabának, a szakmai támogatásaiért és tanácsiért, javaslataiért, türelméért,
- Kovács Tímeának, a Microsoft® Word 2007 használatában való segítségéért,
- Fugerth Máténak, a Visual C# útmutatásáért.