Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Radócz Dániel

JÁTÉK KÉSZÍTÉSE MICROSOFT KINECT PLATFORMRA

KONZULENS

Albert István

BUDAPEST, 2012

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

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

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

Kelt: Budapest, 2012. 12. 06.

.....
Radócz Dániel

Összefoglaló

A Microsoft Kinect technológia ugyan még csak a közelmúltban kezdett el terjedni PC-ken is, mégis egyre többen kezdenek foglalkozni azzal, miként lehet a segítségével újfajta, könnyebben kezelhető alkalmazásokat létrehozni. Ennek oka, hogy a Kinect és a hozzá tartozó Kinect SDK több olyan lehetőséget teremt, amik segítségével az alkalmazások vezérlését sokkal interaktívabbá, és sok esetben egyszerűbbé tehetjük. Ezek közé a lehetőségek közé tartozik például a gesztusok felismerése és értelmezése, a különböző testhelyzetek felismerése, a hangfelismerés. Ezekhez a Kinecten kívül semmilyen más kiegészítő eszközre nincs szükség (pl. speciális szemüvegre, távirányítóra).

A szakdolgozat célja egy olyan keretrendszer összeállítása, amely a Kinect platformból érkező adatok (csontváz adatok, mélységi adatok) alapján képes mind az egyszerűbb, mind az összetettebb gesztusok stabil felismerésére (körmozgás, egyenes vonalú mozgások és ezek kombinációi), különböző testhelyzetek felismerésére, illetve reagálni tud adott szóbeli parancsokra is.

A szakdolgozatom része még egy, a rendszer képességeinek bemutatására készült WPF-alapú játék, amely minden fentebb felsorolt funkciót demonstrál működés közben is.

Abstract

Though Microsoft's Kinect technology only started to spread to the PC platform in the recent past, more and more developer starts to create new, more interactive applications. The reason is that the Kinect platform and the adherent Kinect SDK offers a lot of feature to create applications with interactive, gesture or speech based control abilities. These features can make an application's control much easier, and (in most cases) much simplier. A few possibility of the platform: gesture detection and recognition, pusture recognition and speech recognition. All of them became available with the Kinect device, you don't need another supplementary device (for example special googles or remote controllers).

My goal is to create a framework, which is process the data coming from Kinect (depth values, skeleton and joint positions) and capable of recognizing simple and complex gestures (linear movements, circle movements, and any combination of them), different postures, and also capable of recognizing some vocal commands.

A demonstrating application is also a part of my work. This is a WPF-based game, which can demonstrate all features described above.

Bevezetés

Az információs technológiák fejlődésével folyamatosan nyílnak meg olyan lehetőségek, amelyekkel egyszerűbbé tehetjük a minket körülvevő számítógépes alkalmazások irányítását. Ilyen lehetőség az emberi mozgással és hanggal való vezérelhetőség, amelyek új távlatokat nyitnak meg ebben a témakörben. Az ilyen módon vezérelt alkalmazások nemcsak a szórakoztatóiparban nyújtanak nagyobb felhasználói élményt, hanem az élet többi területén is jelentősen könnyíthetik a munkát – például orvosi alkalmazásokban, ahol a sterilitás megsértése nélkül irányíthatnak egy segédprogramot; intelligens házakban ahol akár szóbeli parancsokkal vezérelhetnek olyan dolgokat mint a fény, vagy a hőmérséklet a szobában; vagy mozdulatokkal irányítható prezentációkban. Napjainkra több olyan eszköz készült, amelyek segítségével akár otthonunkban is lehetővé válik ilyen programok alkalmazása, ez egyre inkább magával vonja az természetes emberi interakciókkal is működtethető alkalmazások fejlesztését.

Ilyen eszköz a Microsoft által létrehozott Kinect szenzor, amit bár eredetileg egy Xbox- hoz való kiegészítőnek szántak, mára megjelentek a PC-re szánt verziók és a hozzá készült (hivatalos és nem hivatalos) fejlesztőeszközök. Ezek segítségével könnyen hozzáférhetünk a szenzor által vett adatokhoz, azonban ezek további feldolgozása (pl. gesztusok felismerése a kapott adatokból) már a fejlesztő dolga.

Szakdolgozatomban egy olyan rendszert készítettem, amivel ez a feladat hatékonyan elvégezhető. Célom egy olyan struktúra kialakítása volt, amivel egy fejlesztő egyszerűen és rugalmasan tud gesztus- és hangfelismerő képességeket hozzáadni az általa fejlesztett szoftverekhez anélkül, hogy ismernie kellene a felismerő algoritmusok pontos menetét. A rendszer képességeit egy WPF-alapú mintaalkalmazáson keresztül mutatom be, amely két, Kinect-el vezérelhető mini játékot (egy memóriajátékot és egy Terist) tartalmaz.

Az első fejezetben egy rövid áttekintést adok azokról az ismertebb, mozgásérzékelésre alapuló szórakoztatóipari eszközökről, amelyek a Kinect mellett elérhetőek jelen szakdolgozat írásakor, főleg azok működési elvére koncentrálva.

A második fejezetben magát a Kinect szenzort mutatom be, kitérve a hardver belső működésére és képességeire, illetve a fejlesztést támogató szoftverekre.

A harmadik fejezetben a mintaalkalmazás alapjául szolgáló Windows Presentation Foundation (WPF) platformot mutatom be.

A negyedik fejezetben a Kinecthez készített felismerő rendszert mutatom be.

Az ötödik fejezetben a mintaalkalmazás belső felépítését mutatom be, külön kiemelve hogy kell bele integrálni a felismerő rendszert.

A hatodik fejezetben a rendszer továbbfejlesztési lehetőségeiről lesz szó.

A hetedik fejezetben a szakdolgozatban leírtakat foglalom össze.

1 Mozgáskövető eszközök a szórakoztatóiparban

1.1 Nintendo Wii

A Nintendo 2006-ban jelentette meg a Nintendo Wii konzolt. Ennek a legnagyobb újdonsága a vezeték nélküli kontrollere, a Wii Remote vagy Wiimote, amely mozgásérzékelő és gesztusfelismerő képességekkel rendelkezik.

Ehhez egyrészt egy beépített gyorsulásmérőt használ, amely képes mindhárom dimenzióban meghatározni az eszköz gyorsulását. Ezen kívül a konzolhoz tartozik egy LED-sor, ami a központi egységhez csatlakozik, és a TV alá vagy fölé kell helyezni. Ezen 10 LED található egy speciális elrendezésben (két ötös csoportban, ahol a középponthoz legközelebbi LED-ek kicsit a középpont irányába van forgatva, a két szélső LED kicsit kifelé, a többi pedig előre néz). A Wii Remote-ban található egy érzékelő, amely veszi a LED-ek fényeit és miután a LED-ek közötti távolságok fixek, háromszögeléssel kiszámolja a pozícióját. Ezen kívül vizsgálja még a fények beesési szögét is egymáshoz képest, ezzel meghatározza a vízszintestől való elfordulását.

A két érzékelőt (a gyorsulásmérőt és a LED-érzékelőt) a rendszer kombinálja egymással, a hirtelen gyors mozgásokat a gyorsulásmérő kezeli le, míg a lassabbakat a LED-érzékelő. Ezzel lehetővé válik a pontos mozgáskövetés, és az erre alapuló gesztusfelismerés.

1.2 Sony Playstation Move

A másik nagy konzolgyártó, a Sony 2010 szeptemberében állt elő saját mozgásérzékelő megoldásával, a Playstation Move-al. Ez nem egy önálló konzol, hanem a Playstation 3-al használható kiegészítő csomag, ami egy mozgásérzékelő kontrollert, egy kamerát, és egy kiegészítő kontrollert tartalmaz.

Működésének alapeleme a kontroller végén található gömb, amely a benne található piros, zöld és kék LED tömb segítségével bármilyen színben tud világítani. A kamera érzékeli ezt a fényforrást, a vett képből megkapja a pozíció X,Y koordinátáit, és a gömb méretéből illetve a szintén ismert fényerősségből kiszámolja a kontroller távolságát is. A fényforrás színét alapból olyanra választja meg a rendszer, hogy a lehető legjobban eltérjen a környezete színeitől, de az alkalmazások maguk is állíthatják a színét, növelve a felhasználói élményt.

A gömb mellett egyéb kiegészítő érzékelők is találhatóak a kontrollerben. A giroszkóp és a 3D gyorsulásmérő segítségével meg lehet határozni a kontroller orientációját, illetve ezek segítségével érzékeli a mozgást azokban az esetekben, amikor a kamera nem látja a fényforrást. Ezeken kívül tartalmaz még egy, a Föld mágneses mezejét érzékelő magnetométert is, amely szintén a kontroller orientációját határozza meg, ezzel a két másik érzékelő eredményeit pontosítja.

2 A Kinect platform

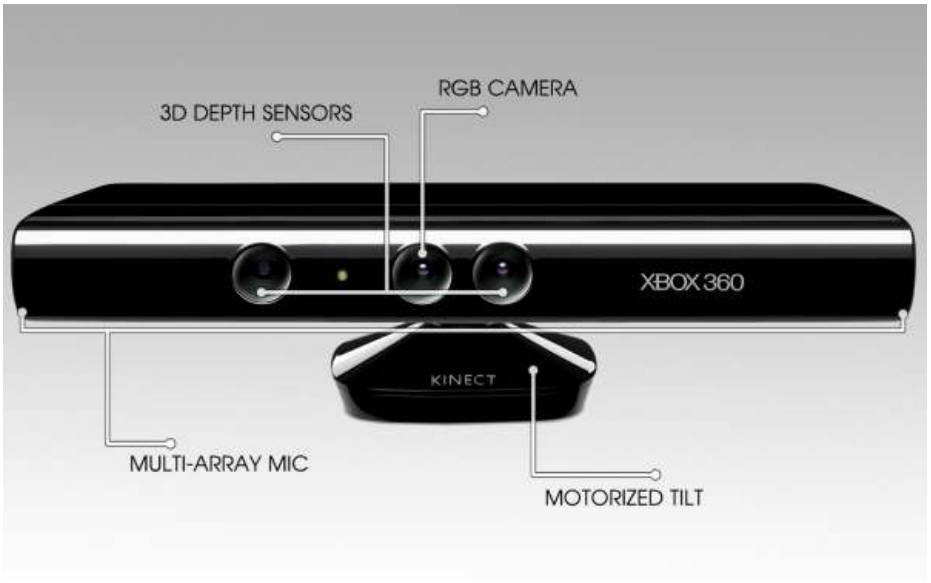
A Kinect for Xbox Sensor-t eredetileg egy Xbox kiegészítőnek szánták, ami NUI (Natural User Interface) vezérelhetőséget ad a konzolnak, és az erre felkészített játékoknak. 2010 novemberében került piacra. A rendszer legnagyobb előnye riválisaihoz képest, hogy magán a Kinect szenzoron kívül semmilyen egyéb eszközre nincs szükség a NUI funkciók eléréséhez, a szenzor önállóan képes felismerni az emberek pozícióját, és ezen belül az összes testrészük elhelyezkedését.

Konzolos kiegészítő volta miatt eredetileg az eszköz sem PC-hez való driverekkel, sem hozzáférhető fejlesztői támogatással nem rendelkezett. A Kinect azonban hamar túlnőtt a klasszikus szórakoztatóiparbeli felhasználáson, a fejlesztők elkezdték keresni az újabb felhasználási területeket. Megjelentek a nem hivatalos PC driverek, majd később a hivatalos Microsoft driverek és fejlesztőeszközök is, megnyitva

az utat a NUI vezérelt PC alkalmazások előtt. Időközben magából az eszközből is kiadtak egy újabb, már kifejezetten PC-re szánt verziót, a Kinect for Windows Sensort, ezzel a Kinect teljesen túllépett a konzolos kiegészítő szerepén.

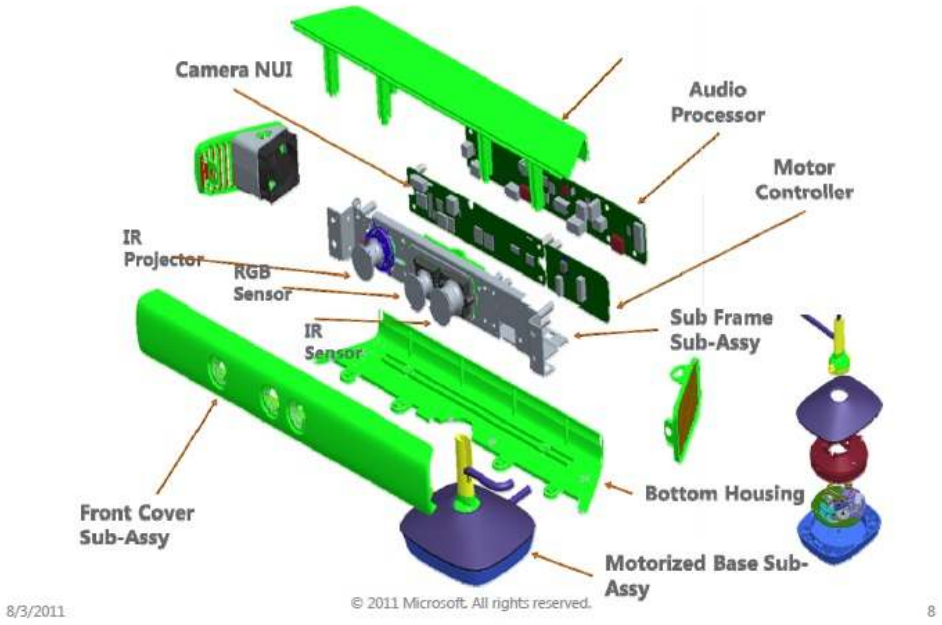
2.1 Hardver felépítése

A Kinect sikeréhez hozzájárult, hogy ötvözték benne a legmodernebb elérhető fejlesztéseket a hangfelismerés és mozgásérzékelés terén. Ebben a fejezetben a Kinect alapját képező fő technológiai megoldásokat szeretném bemutatni.



1. ábra Kinect külső felépítése

A szenzoron három érzékelő látható, ezek közül a középső egy hagyományos RGB kamera, a két szélső pedig a mélységérzékelésért felel, erről a következő fejezetben részletesebben is szó lesz. Az érzékelők alatt egy 4 mikrofonból álló mikrofontömb található elosztva. Maga az eszköz pedig egy szoftveresen vezérelhető csuklón található, amely lehetővé teszi a Kinect vertikális döntését.



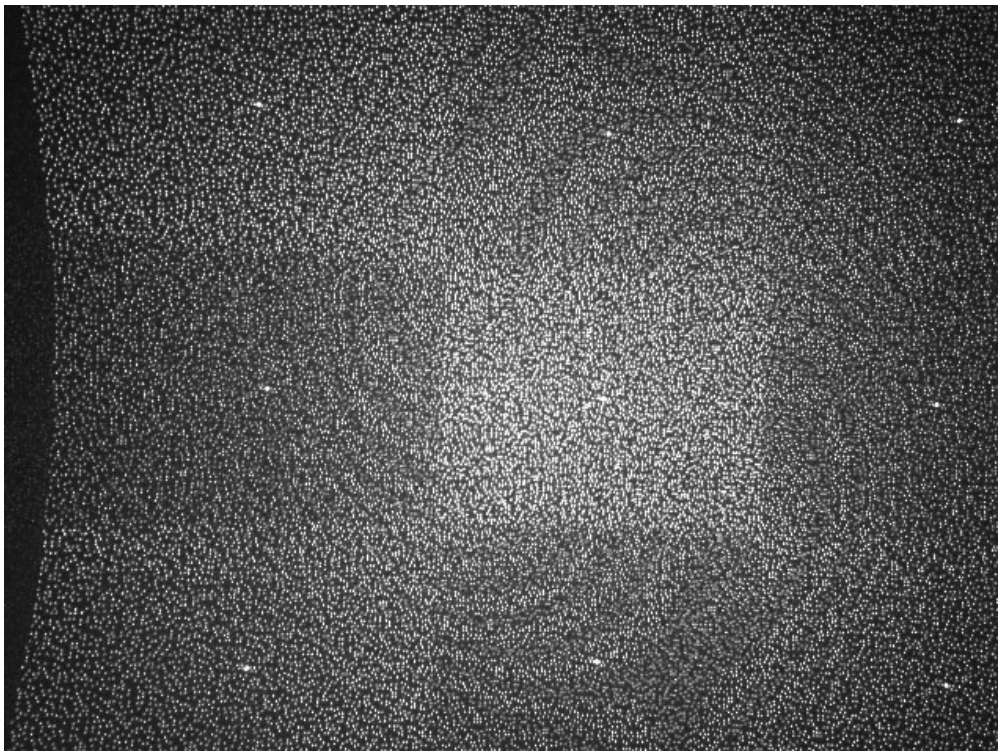
2. ábra Kinect belső felépítése

A szenzoron belül találhatóak a feldolgozó egységek, ezek közül érdemes kiemelni a mélységi és RGB adatokat feldolgozó egységet, ami a csontozat felismerést végzi, illetve a mikrofon tömb által vett jeleket feldolgozó audio processzort.

2.1.1 Mélységérzékelés

A Kinect egyik fő képessége a mélységérzékelés, ez képezi az alapját a csontvázfelismerő rendszernek, és ezen keresztül az összes gesztusérzékelő algoritmusnak. A rendszer az infravörös hullámok visszaverődéséből következtet a látószögben lévő objektumok elhelyezkedésére.

Két fő alkotóeleme van, egy infravörös hálót kibocsájtó projektor illetve a visszaverődött sugarakat érzékelő szenzor. Működésének alapját az úgynevezett strukturált fények képezik[2], aminek a lényege, hogy egy fénypontokból álló mintát vetít ki a környezetre (3. ábra), összehasonlítja azt az érzékelőben vett visszaverődött mintával és megpróbálja megfeleltetni egymásnak a mintában szereplő pontokat – amint ez sikerül ismertté válik a fénysugár kilépési és visszaverődési pontja, ez alapján (az érzékelők távolságát és fókuszpontját ismerve) a térbeli pont koordinátái háromszögeléssel kiszámolhatóak.



3. ábra A Kinect által kivetített minta infravörös kamerával fényképezve (forrás: http://futuretheater.net/wiki/Kinect_Workshop)

Az algoritmus központi része a pontok megfeleltetése. Ennek megértéséhez ismerni kell az epipoláris kényszert, amely szerint, ha ismert egy pont képe az egyik nézetből, akkor a másik nézetben ugyanennek a pontnak a képe egy egyenes mentén helyezkedik el. Amennyiben a két nézet síkja azonos, a nézetek fókuszpontjai egyenlő távolságra vannak a síkoktól, és azonos az alapvonaltól számított magasságuk, ez az egyenes az alapvonallal párhuzamos lesz. Így ha egy adott pont megfelelőjét keressük a másik nézetben elegendő az egyenes mentén végignézni a pontokat, és kiválasztani a legjobb egyezést (ehhez nemcsak az adott pontot, hanem a pont környezetét is vizsgálni kell).

A Kinect által használt megoldás egy konstans fénymintát vetít ki, majd minden vett mintára a következő algoritmust alkalmazza:

1. Minden pontot „ismeretlennek” jelöl
2. Kiválaszt egy tetszőleges „ismeretlen” pontot, az egyenes mentén megpróbál egyezést keresni a kibocsájtott mintával – amennyiben nem sikerül a pontot „érvénytelennek” jelöli és új pontot választ.
3. Ha sikerül egyezést találni

a. felveszi a pont szomszédjait egy sorba
b. mindegyik pontra inicializál egy keresést, amelynek a középpontja a kiinduló ponttól való eltolás, a tartománya pedig a pont kisebb környezete
c. ha egyezést talál, annak a pontnak is felveszi a szomszédjait a sorba
d. ezt folytatja, amíg a sor ki nem ürül

Az algoritmus addig fut, amíg minden pont mélysége ismert vagy érvénytelenné van nyilvánítva.[3]

A Kinect 30 fps-el (frame per second) képes a 640x480 felbontású mélységi képek előállítására, ami megfelel a valósidejű 3D érzékelés követelményeinek.

2.1.2 Csontvázfelismerés

A mélységi képek feldolgozása a következő lépés. A cél az emberi alakzatok felismerése és ezeken belül nevezetes pontok (ízületek, fej, testközéppont) pozíciójának meghatározása. Ez két fő lépésben történik.

Az első lépés célja testrészek beazonosítása a képen. Ehhez döntési fákat használ fel. Minden fa bemeneti adata a mélységi kép egy pixele, a kimenet pedig hogy az adott pixel melyik testrészhez tartozik a legnagyobb valószínűséggel. Az algoritmus működéséhez mintaadatok szükségesek, ez ebben az esetben felcímkézett mélységi képeket jelent.

A Kinect fejlesztésekor problémát jelentett, hogy a megbízható működéshez rengeteg mintaadat kellett volna, ideális esetben az összes lehetséges emberi pozícióról figyelembe véve a különböző testalkatokat, magasságokat, stb. Ezek begyűjtése és tárolása is rendkívül költséges és erőforrásigényes lett volna. Ennek kiküszöbölésére egy kisebb, kb. 500 000 valós képből álló adatbázis (amely mindenféle tevékenységből származó pózokat tartalmaz) alapján generálódik egy mesterséges képhalmaz (a címkéket megőrizve), amely mintaadatként szolgál.

A címkézéskor minden döntési fa más, véletlenszerűen generált képhalmazzal kerül inicializálásra. Minden pixel ugyanazon a döntési erdőn megy keresztül, így erre az inicializálásra mélységi képenként egyszer van szükség. Amint egy pixel minden

fától megkapja a valószínűsített címkéjét, az eredményeket átlagolva rendelődik hozzá az ezek közül legvalószínűbb címke.

Miután sikerült minden pixelt a megfelelő testrésszel felcímkézni a mélységi képen, a következő lépés minden rész középpontjának (móduszának) megtalálása – ezek a pontok adják az algoritmus kimenetét, a felismert csontváz nevezetes pontjait. A középpontok megtalálása a „Mean-shift" algoritmussal történik, ami képes ezt a feladatot gyorsan és hatékonyan teljesíteni.[4]

Az algoritmus minden pixelre párhuzamosan futtatható, ezzel különösen hatékonyan futtatható GPU-kon. Az Xbox GPU-ján 200 fps sebességgel fut.

2.1.3 Hangérzékelés

A képfeldolgozás mellett a Kinect működésének másik nagy területe a hangfelismerés. Itt a cél egy olyan rendszer megalkotása volt, ami képes az emberi hangok felismerésére akár többméteres távolságból, mindezt egyéb kiegészítők (mint headset, push-to-talk) nélkül. Ezzel kapcsolatban a legnagyobb problémát a hangszórók hangjának kiszűrése jelentette, ezek a konzol elhelyezéséből adódóan sokkal közelebb voltak a Kinect-hez mint az emberek, emellett a felhasználók többsége magas hangerőn használta az alkalmazásokat (erre jó példák a zene- és videólejátszók). Bár ennek megoldására létezett algoritmus (Acoustic Echo Cancellation, akusztikus visszhang elnyomás), ez mono hangokra volt alkalmazható, nem a gyakorlatban használt sztereó- és sorround hangokra.[10]

A megoldást több egymáshoz közel elhelyezett mikrofon, az úgynevezett mikrofontömb adta. A különböző mikrofonok által vett hangokat analizálva lehetőség nyílik adott irányokból érkező hangok felerősítésére, illetve a többi hang gyengítésére – ez a folyamat a beamforming, ami két fő lépésből áll.

Az első lépés a hangforrás megtalálása, azaz a zajos, visszhangos környezetből megállapítani honnan jönnek emberi hangok. Ehhez először is a beérkező jeleket csoportosítják az észlelés iránya alapján – ez ugyanannak a jelnek a különböző mikrofonpárok által vett változatainak összehasonlításával számolható ki, az azonos frekvencián vett fáziskülönbségeket vizsgálva. A csoportosítás szektorok alapján történik, amelyek a mikrofonok által érzékelt teret bontják különálló részekre. A hangok és a zajok modellezése minden szektorban függetlenül történik. Ennek nagy előnye, hogy minden hangforrás külön-külön, más paraméterekkel modellezhető, ezáltal

pontosabban elkülöníthető a zajtól, továbbá a térbeli elhelyezkedés ismeretében lehetővé válik több emberi hangforrás közül is kiválasztani azt, amelyik a kívánt helyről indul.[11]

A hangforrások feltérképezése után a második lépés a beam (érzékelő sugár) beállítása egy kívánt hangforrásra, kiemelve azt az összes többi hang közül. Ennek általános formája az alábbi képlettel írható le:

$$Y(f) = \sum W_m(f) X_m(f)$$

Itt az Xm(f) az m. mikrofon által vett jel, a Wm(f) pedig ennek a frekvenciafüggő súlyozása. A súlyozási értékek beállítása adaptív algoritmussal történik, a kívánt forrásból érkező hangok értelemszerűen magasabb súlyokat kapnak.

2.2 Fejlesztői eszközök

Miután a kezdetektől fogva nagy érdeklődéssel övezett platform eredetileg hivatalos fejlesztői támogatás (és PC-re szánt driver) nélkül került piacra, fejlesztői körökben szinte azonnal indultak próbálkozások ezek pótlására. Ennek a folyamatnak volt része az a 3000 $ értékű díj is, amit az első PC-s nyílt forráskódú driver megalkotására tűztek ki [1]– a Kinect iránti érdeklődést jellemzi, hogy ezt 2010. november 10-én meg is nyerték a libfreenect névre keresztelt szoftverrel, mindössze 6 nappal a Kinect Észak-Amerikai megjelenése után. Ezt hamarosan követte a PrimeSense által készített szintén nyílt forráskódú driver, amihez már komplett fejlesztői támogatás is tartozott, az OpenNI. Végül valamivel később, 2011. júniusában a Microsoft is elérhetővé tette a hivatalos SDK-t, a Kinect for Windows SDK-t. Az alábbiakban ezeket az eszközöket fogom összehasonlítani.

2.2.1 OpenKinect

Az OpenKinect projekt a már említett libfreenect driver köré épül. Ahogy a neve is jelzi, nyílt forráskódú. Eredetileg csak C++ támogatással rendelkezett, a szakdolgozatom írásakor már több nyelvre készült hozzá wrapper, többek között C, C#, Python, Lisp, Java és Javascript nyelvekre is. Az API-n keresztül hozzáférhetünk az RGB és mélységi képekhez, illetve a dőlésszöget vezérlő motorhoz, viszont nincs lehetőség se a csontváz koordináták, se a hangvezérlés elérésére. Az API legnagyobb

előnye a gazdag nyelvi támogatottságon túl a platformfüggetlenség (Linux, Windows, és MacOS alatt is elérhető).

2.2.2 OpenNI/NITE

Az OpenNI a PrimeSense cég által fejlesztett nyílt forráskódú keretrendszer, ami egy absztrakt interfészt definiál az alkalmazások és a különböző audio/video szenzorok között. A Kinect-el való működtetéséhez szükséges még maga az OpenNI kompatibilis driver is, ez szintén hozzáférhető a PrimeSense által. A keretrendszerrel hozzáférhetünk a színes és mélységi képeken kívül az audio streamhez is, illetve (ha a hardver támogatja) csontváz és gesztus információkhoz is. Az OpenNI Windows és Ubuntu Linux alatt érhető el és az OpenKinect-hez hasonlóan gazdag nyelvi támogatottságú. Az alap keretrendszer modulárisan tovább bővíthető különböző middleware komponensekkel, amikkel további magasabb rendű funkciókat érhetünk el, ez Kinect esetében pl. a gesztusfelismerés. A PrimeSense által kínált ilyen middleware komponens a NITE, amely lehetővé teszi az olyan egyszerű kézgesztusok felismerését, mint az oldalra lendítés, a körzés vagy az előre nyomás. Ezeket a gesztusokat tetszőlegesen kombinálhatóvá teszi, így komplexebb gesztusok felismerésére is alkalmas. A NITE az OpenNI-vel ellentétben nem nyílt forráskódú.

Az OpenNI/NITE keretrendszer előnye, hogy bármilyen eszköz programozható vele, amelyik rendelkezik OpenNI kompatibilis driverrel, illetve ez a legkiforrottabb API ami több operációs rendszert is támogat. Emellett gazdag dokumentációval és sok példaprogrammal rendelkezik, megkönnyítve a fejlesztés kezdeti lépéseit.

2.2.3 Microsoft Kinect for Windows SDK

A Microsoft hivatalos Kinect SDK-ja a szakdolgozat írásakor az 1.6-os verziónál tart. Értelemszerűen ez az SDK adja a legszéleskörűbb támogatást. Hozzáférhetőek a színes és mélységi képek, a csontváz koordináták. Ezeken kívül több, jelenleg csak ebben az SDK-ban elérhető funkcióval rendelkezik, mint az infravörös kép elérése, az arcfelismerés (a színes és mélységi képek összevetésével), vagy a csontok egymástól való orientációjának kiszámolása. A gesztusfelismerés ellenben nem képezi részét, a szakdolgozatomban többek között erre kínálok megoldást. Az SDK gazdag fejlesztői támogatással rendelkezik, pl. a Kinect Studio segédprogrammal, amivel folyamatosan láthatjuk milyen adatok érkeznek a Kinect-ből, ezzel jelentősen megkönnyítve a debugolást. Emellett jelentős mennyiségű példaprogram és

dokumentáció is rendelkezésre áll. Az SDK nem platformfüggetlen, csak a Windows 7 és 8 operációs rendszerekkel kompatibilis.

Szakdolgozatomban ezt a keretrendszert választottam, mert ez rendelkezik a legnagyobb támogatottsággal, kompatibilis a .NET-el, a Kinect összes elérhető képességét kihasználja, és hivatalos SDK révén a folyamatos fejlesztése is biztosítva van. Cserébe le kell mondani a platformfüggetlenségről, mivel csak a Windows 7 és 8 rendszerek támogatottak.

2.2.4 Microsoft Speech Platform SDK

Az előzőekben már volt szó a Kinect hangfelismerési képességeiről. Bár az ennek kimeneteként kapott jó minőségű hang több mindenre alkalmazható, leggyakrabban a hangvezérlésben használják fel. Ez már nem a Kinect feladata, hanem a hozzá kapcsolódó Xbox-é vagy PC-é.

Ilyen hangvezérlési funkciókat valósít meg a Speech Platform SDK, ami jelenleg a 11-es verziónál tart. Nem kifejezetten Kinectre íródott, általánosan használható bármilyen más hangfelismerésre képes eszközzel, feltéve, ha a drivere támogatja azt. A platform képes a beszédfelismerésre, nemcsak szavakat, de egész mondatokat is detektálni tud. A mondatokat egy jól felépített keretrendszerrel állíthatjuk össze, lehetőség van bizonyos szavak helyett szóhalmazokat fűzni a mondatba (ilyenkor bármelyik szóval felismeri a mondatot) vagy akár joker (wildcard) szavakat is használhatunk (ilyenkor az adott szó helyén bármilyen bemenettel felismeri a mondatot). Rendelkezik TTS (Text-to-speech) funkciókkal is.

Az SDK működéséhez szükségesek még a nyelvi fájlok is, ezek tartalmazzák a felismerés alapját képező mintaadatokat. Jelenleg 26 nyelvhez létezik ilyen támogatás, a magyar még nincs közöttük.

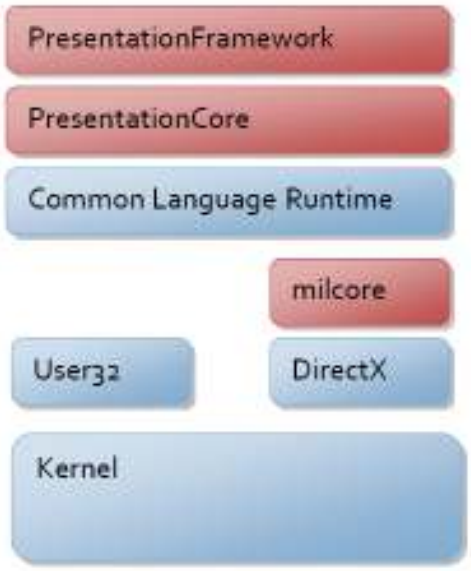
3 A WPF plaform

A Windows Presentation Foundation a .NET keretrendszer része, egy grafikus alrendszer. A WPF először a .NET 3.0 részeként jelent meg (WPF 3.0) azóta 5 nagyobb kiadáson esett át, a jelenlegi (és a szakdolgozatban is használt) verziója a WPF 4.5. Jelentős újításokat hozott az addigi egyetlen .NET-es grafikus alrendszerhez, a

Windows Forms-hoz képest, ezekre a 3.2 fejezetben részletesebben is kitérek. A szakdolgozatomban ezt a keretrendszert választottam a mintaalkalmazás elkészítéséhez.

3.1 Felépítése

A WPF részei mind a menedzselt, mind a natív kódban megtalálhatóak (4. ábra pirossal kiemelt részei). A natív rétegben található komponens a MIL (Media Integration Layer). Minden megjelenítés ezen a rétegen keresztül történik. A feladata a megjelenítés legoptimálisabb módon való teljesítése. Ennek érdekében lemondtak a CLR által kínált menedzselt kód adta lehetőségekről, amik felesleges overhead-et vittek volna a folyamatba, és a teljes komponenst a natív kódba helyezték. Itt közvetlenül kommunikál a konkrét renderelést végző DirectX-el és hozzáfér a memóriafoglalásokhoz is.



4. ábra WPF architektúra (forrás: http://msdn.microsoft.com/en-us/library/ms750441.aspx)

A PresentationCore komponens feladata a MIL-el való kapcsolattartás és a WPF fő funkcióinak implementálása. Itt találhatóak meg azok az elemek, amiket a legfelsőbb réteg fog használni, mint pl. a Layoutok méreteinek pontos kimérése, vagy a szálkezelő Dispatcher rendszer.

A PresentationFramework pedig az alkalmazásban ténylegesen használható elemek találhatóak, mint a vezérlőelemek, az animációk, az adatkötést megvalósító objektumok. [5]

3.2 Főbb tulajdonságai

A WPF megalkotásának egyik fő célja volt, hogy az inkább teljesítményorientált ellenben kevésbé látványos Windows Forms technológia mellé egy olyan alternatívát adjon, amellyel egyszerűen lehet látványos UI-kat (Usr Interface) tervezni, még ha ennek valamivel nagyobb overhead is az ára. A WPF néhány tulajdonsága:

- **DirectX-alapú renderelés:** az addigi GDI alapú megjelenítést DirectX alapúra cserélték, ami támogatja a 3D-s megjelenítést és hatékonyabban működik
- **XAML:** a WPF leírónyelve, egy XML alapú deklaratív nyelv. Segítségével elkülöníthető az UI tervezés folyamata a mögöttes kód fejlesztésétől. Egy WPF alkalmazás készítéséhez nem feltétlenül szükséges használni (lehetőség van mindent kódból létrehozni, maguk a XAML elemek is CLR osztályokká fordulnak le), ellenben több olyan eszköz van amellyel egyszerűen tervezhető XAML-el leírt felület, a legismertebbek az Expression Blend és a Visual Studio.[7]
- **Adatkötés:** a WPF-vezérlőkhöz egyszerű módon hozzáköthetjük egy osztály property-jét, és az INotifyPropertyChange interfész implementálásával elérhető hogy az adat változásával az őt megjelenítő vezérlők tartalma is automatikusan frissüljön.
- **Dispatcher:** a WPF alapesetben 2 szálat használ: egyet a renderelésre, egyet az UI kezelésére. Utóbbihoz egy Dispatcher nevű objektumot használ, ami prioritások alapján ütemezi az idetartozó feladatokat (pl. input feldolgozás, eseménykezelések). Minden UI-elem őse a DispatcherObject osztály, ami alapján tartalmaznak egy referenciát a szálukhoz tartozó Dispatcher objektumról. Az UI módosítása csak az azt kezelő szálon keresztül lehetséges, így ha egy háttérszálból akarunk módosítani valamelyik elemen, akkor azt a hozzátartozó Dispatcher objektumon keresztül kell megtenni. [8]
- **Animációk:** a WPF platform magas szinten támogatja az elemek animálását. Az Animatable ősosztályból származó osztályokkal a legtöbb típus animálható, azaz megadható a kezdő és végértékük, az animálás ideje, vagy az hogy melyik időpillanatban milyen értéket vegyen fel. A szükséges időzítéseket a WPF automatikusan kezeli. Az animációk Storyboard-ok alatt csoportosíthatóak, lehetővé téve a komplexebb, akár több vezérlő több tulajdonságát is érintő animációkat.

4 A felismerő rendszer

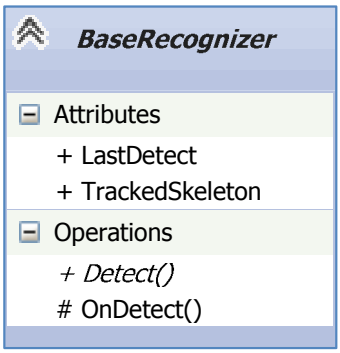
A szakdolgozatom fő része a Kinectre épülő felismerő rendszer. A cél egy olyan API megalkotása volt amivel egy alkalmazás egyszerűen kiegészíthető Kinect-támogatással. A tervezés fő szempontjai:

- **A Kinect belső működésének elrejtése.** A felhasználónak ne kelljen közvetlenül kezelnie a Kinectből érkező adatokat, mint pl. a csontvázstruktúra, vagy a nyers kép adatok.
- **Komplex vezérlőelemek alkalmazhatósága.** Legyen lehetőség az elemi gesztusok és pózok kombinálásával tetszőlegesen összetett vezérlőmozdulatokat létrehozni.
- **Testreszabhatóság.** A vezérlő mozdulatok lényeges paramétereit (sebesség, távolság) egyszerűen lehessen konfigurálni akár futás közben is.

4.1 Architektúra

A rendszer két fő részre tagolódik. A felismerést végző osztályokat a *Kinect.Recognizer* névtér tartalmazza, a másik névtér a *Kinect.Helper* kisegítő osztályokat tartalmaz, amelyek a Kinect vezérlés integrálását könnyítik meg. Utóbbiról a 4.5 fejezetben lesz szó részletesebben.

Minden felismerést végző osztály az absztrakt *BaseRecognizer* osztályból származik. Ez tartalmazza a minden gesztus és pózfelismerés alapját képező elemeket. A *TackedSkeleton* property tárolja az a felismerőhöz kötött csontváz objektumot. Absztrakt metódusa a *Detect*, a leszármazott osztályoknak ebben kell implementálni a konkrét felismerő algoritmust. A felismerés eredményét a *DetectResult* adatosztályban tároljuk –ezen keresztül elérhető a gesztus kezdő- és végpontja, illetve időtartama is. Itt található még a *Detected* event, ami felismeréskor sül el, illetve a tervezésben szintén fontos szerepet kapó *LastDetected*



5. ábra a BaseRecognizer osztály

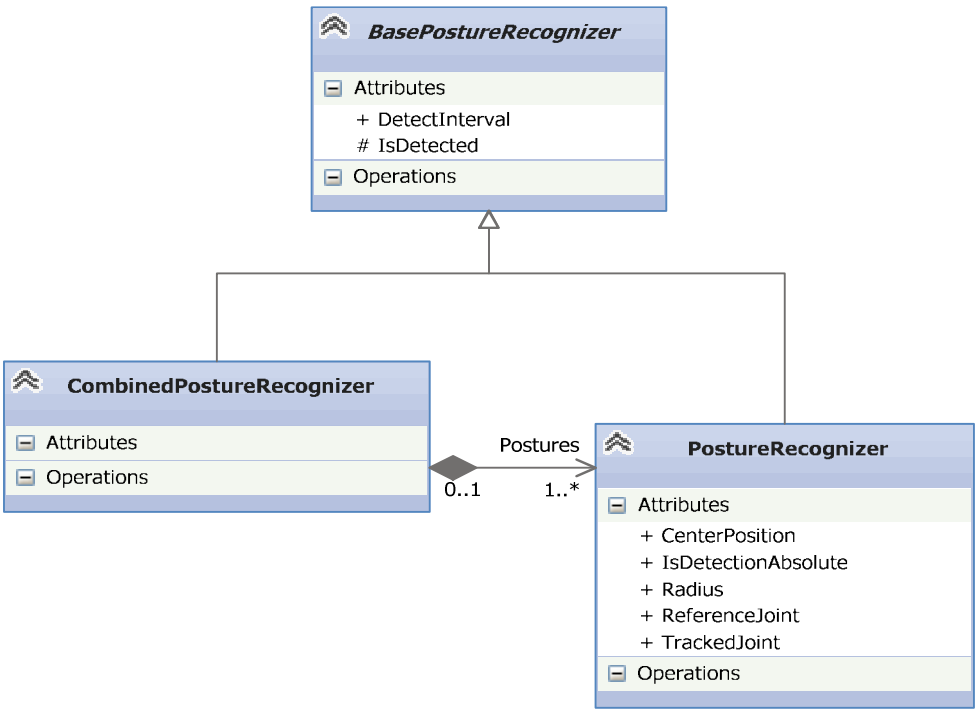
property, ami tárolja a legutóbbi felismerés dátumát – ez mindig az event elsütésekor frissül az erre szolgáló *OnDetected* metóduson keresztül:

```
protected void OnDetect(DetectedEventArgs e)
{
      LastDetect = DateTime.Now;
      EventHandler<DetectedEventArgs> ev = Detected;
      if (ev != null)
      {
          Detected(this, e);
      }
}
```

4.2 Pózfelismerés

Pózfelismerés alatt azt értjük, hogy meg tudjuk állapítani egy adott időpontban a testünk adott része vagy részei adott pozícióban vannak-e. Az adott pozíció megadása lehet abszolút (a világkoordináták valamelyik pontjához viszonyítva) vagy relatív (a csontváz egy adott pontjához viszonyítva). Bár maga a felismerés állapotmentes folyamat (mivel mindig csak az adott pozíciókat nézzük), hasznos, ha a rendszer nemcsak a póz felismeréséről értesít, hanem a megszűnéséről is, illetve ha egy póz fennállása alatt bizonyos időközönként folyamatosan jelez annak érvényességéről. Ezért ezeket a funkciókat megvalósítottam az API pózfelismerő részében.

A pózfelismerő osztályokat a *Kinect.Recognizer.Postures* névtér tartalmazza.



6. ábra Kinect.Recognizer.Postures

Az alaposztály az absztrakt *BasePostureRecognizer*, ami a BaseRecognizer leszármazottja. A DetectInterval property megadja, hogy a póz első detektálása után milyen időközönként legyen elsütve a Detected event. Az osztály rendelkezik egy Finished event-el, ami a póz megszűnésekor sül el. Ezek felismeréséhez használható fel az IsDetected property, ami jelzi hogy a póz éppen fel van-e ismerve.

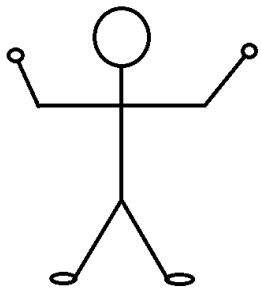
Ennek egyik implementációja a *PostureRecognizer,* ami egy adott testrész pozícióját tudja vizsgálni. A működéséhez definiálni kell egy - abszolút vagy relatív- középpontot és mindhárom dimenzióra egy értéket, amely megadja, hogy minimum milyen távolságra legyen a vizsgált testrész ehhez a ponthoz hogy sikeres legyen az érzékelés. Amennyiben a középpontot egy referenciaponthoz kívánjuk viszonyítani, meg kell még adni melyik testrész adja a referenciapontot – ez tipikusan valamelyik, a test középvonalán elhelyezkedő pont (a Head, a ShoulderCenter vagy a HipCenter).

Az alábbi példa egy olyan pózfelismerő példányosítását mutatja be, amely akkor detektálja a pózt, ha a bal kézfejünket nagyjából 50 – 150 cm-re tartjuk a vállközepünktől.

```
lefthand = new PostureRecognizer
       {
                DetectInterval = TimeSpan.FromMilliseconds(200),
                TrackedJoint = JointType.HandLeft,
                ReferenceJoint = JointType.ShoulderCenter,
                CenterPosition = new SkeletonPoint { X = -1f,Y = 0,Z = 0 },
                Radius = new SkeletonPoint { X = 0.5f, Y = 0.5f, Z = 0.3f }
       };
```

Ahhoz hogy az objektumot használni tudjuk, még szükséges a TrackedSkeleton property beállítása, a feliratkozás a Detected eventre, illetve a Detect függvény meghívása megfelelő időközönként (célszerűen a csontváz adatok frissülésekor).

Az összetett pózok felismerésére szolgál a *CombinedPostureDetector* osztály, ami tetszőleges számú PostureDetector-t fog össze. Itt a detektálás csak akkor lesz sikeres, ha a kapcsolódó PostureDetector-ok mindegyike egyszerre sikeres. Az osztály használatával tetszőlegesen komplex pózokat lehet vizsgálni, pl. az előző kódrészletben felhozott pózt a jobbkezes változatával összefogva a 7. ábrán látható pózt detektáló objektumot kapunk.

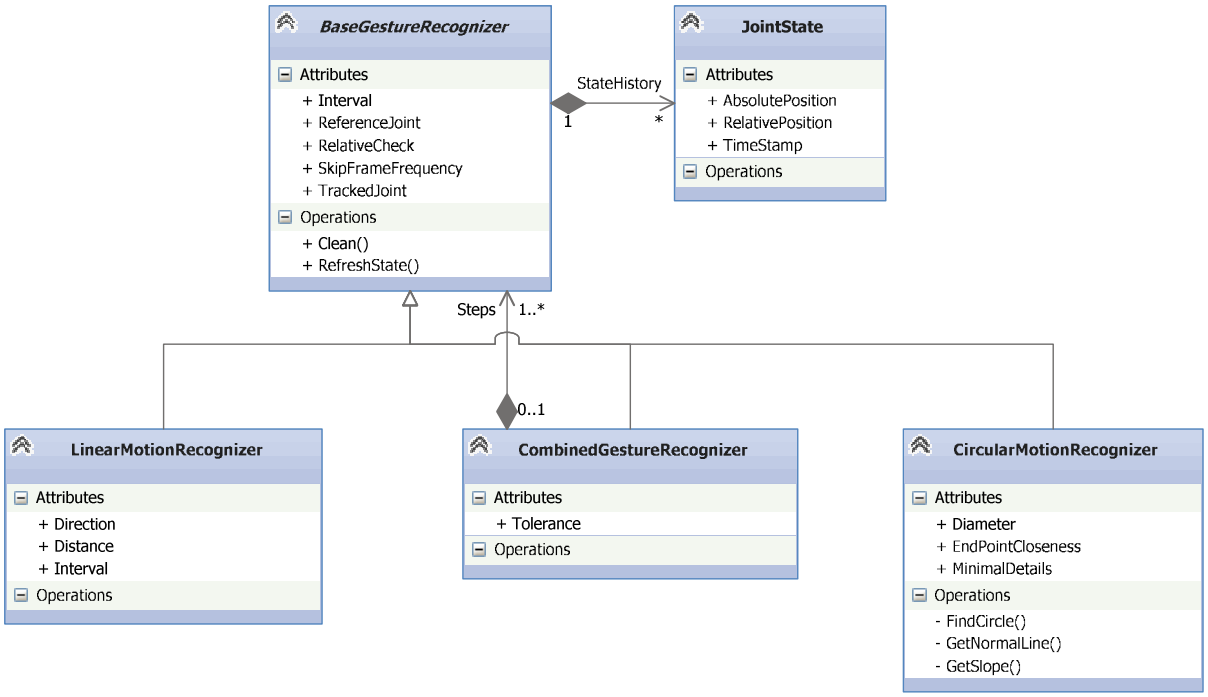


7. ábra

4.3 Gesztufelismerés

Gesztusfelismerés alatt annak felismerését értjük, amikor a felhasználó valamilyen mozdulattal vagy mozdulatsorozattal akar információt átadni. A pózfelismeréssel ellentétben itt mindenképp szükséges a korábbi állapotok vizsgálata is. Alapvetően egy felismerő egy testrész pozícióit követi, de az API lehetőséget ad ezek tetszőleges kombinálásával összetett gesztusok felismerésére is.

A gesztusfelismerést végző osztályokat a Kinect.Recognizer.Gesture névtér tartalmazza.



8. ábra Kinect.Gestures névtér

A pózfelismerőkhöz hasonlóan itt is van egy, a BaseRecognizer-ből származó absztrakt ősosztály, a *BaseGestureRecognizer*. Ez tartalmazza a követett testrészt és annak előző állapotait. Az állapotok tárolására a *StateHistory* adatosztály szolgál, ebben tároljuk a testrész előző pozícióit abszolút és relatív módon, illetve a hozzájuk tartozó időbélyeget. Az alaposztály része még a virtuális *Clean()* metódus, ami a már túl régi állapotok törlésére szolgál, ezzel elkerülve a túl sok adat felhalmozását, és az ezáltali teljesítménycsökkenést. Ez a konkrét implementációkban felüldefiniálható. A Clean minden sikeres detektálás után meghívódik. Érdemes még kiemelni a SkipFrameFrequency property-t, ami arra szolgál, hogy minden adott számú új csontvázadat feldolgozását átugorja. Ez hasznos lehet, ha egy felismerő algoritmus nem

tud olyan gyorsan futni, mint ahogy az új adatokat kellene feldolgozni, viszont minél több feldolgozást hagyunk ki, értelemszerűen romlik a felismerő stabilitása (alapértelmezett értéke 0).

Az ősosztálynak három implementációja van, az egyenes vonalú mozgások felismerésére szolgáló *LinearMotionRecognizer*, a körmozgásokat detektáló *CircularMotionRecognizer,* illetve az ezek kombinálására szolgáló *CombinedGestureRecognizer.*

4.3.1 Egyenes vonalú mozgások

Az egyenes vonalú mozgások detektálásának szabályát egyszerű megfogalmazni: azt vizsgáljuk, hogy a követett testrész adott időn belül adott irányba megtesz-e egy bizonyos távolságot. A *LinearMotionRecognizer-el* hatféle irányt állíthatunk be a felismeréshez: fel/le, jobbra/balra és előre/hátra. A felismerést végző LINQ kifejezés:

```
var query = from s in StateHistory
            from e in StateHistory
            where(
(RelativeCheck ? GetDistance(s.RelativePosition, e.RelativePosition) :
GetDistance(s.AbsolutePosition, e.AbsolutePosition)) >= Distance
            && (e.Timestamp - s.Timestamp) <= Interval)
            select new
            {
                StartState = s,
                EndState = e
            };
```

Azaz az előzmények között 2 olyan pontot keres, amelyek távolsága legalább a megadott nagyságú, ugyanekkor a köztük eltelt idő a beállított intervallumon belül van. Amennyiben talál legalább egy ilyen párt, a detektálást sikeresnek veszi.

A hivatkozott *GetDistance* metódus a két bemeneti pont megfelelő koordinátákra vonatkoztatott távolságát adja vissza, pl. felfelé mutató iránynál

```
case LinearDirection.Upward:
                    return start.Y - end.Y;
```

4.3.2 Körmozgások

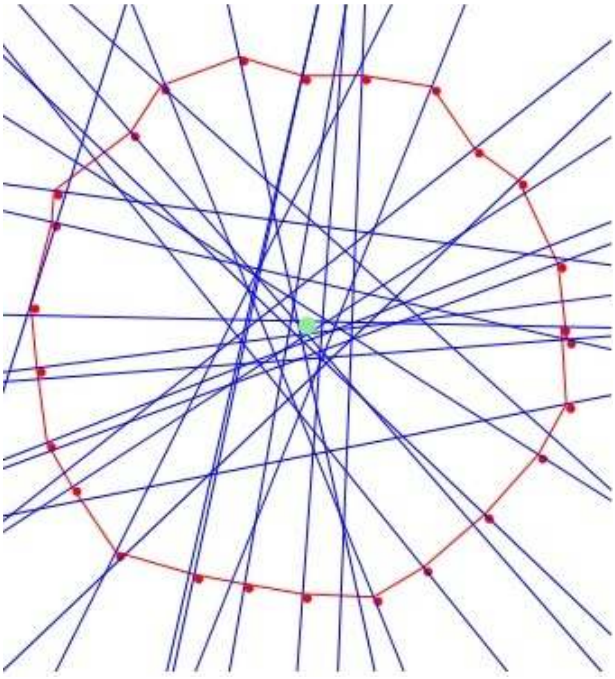
A körzés, mint gesztus az elemi mozdulatok sorába tartozik ugyan, de pontos detektálása az egyenes mozgásoknál jóval bonyolultabb. Több megközelítés létezik a problémára, ezek egyik csoportja matematikai szabályokkal próbálja felismerni egy

koordinátahalmazból a körformát, míg a másik megfelelő számú mintaadattal való összehasonlításokból próbál következtetni. A gyakorlatban az utóbbi megközelítés az elterjedtebb, mert a tapasztalat szerint jól megválasztott mintaadatokkal stabilabban ismeri fel az alakzatot. Ennek ellenére én a tisztán matematikai megközelítést választottam, megpróbálva elérni a mintaadatok nélküli stabil felismerést.

4.3.2.1 Algoritmus bemutatása

Ehhez egy, a képfeldolgozásban használt kördetektáló algoritmust alkalmaztam erre a problémára.[9] Az algoritmus arra a tényre épül, hogy a körvonalon található pontok gradiensei a kör középpontja felé mutatnak. Ebből kiindulva veszi a vizsgált görbéhez tartozó pixeleket (amelyek gradiense már ismert), véletlenszerűen kiválaszt belőlük adott darabot, majd minden kiválasztott elemhez meghatároz egy egyenest, ami átmegy a pixelen, és a gradiensével megegyező irányú. Ezek a vonalak értelemszerűen több metszéspontot is meghatároznak, az algoritmus ezek közül választja ki azt, amelyik környezetében a legtöbb metszéspont tömörül. Amennyiben ez meghalad egy határértéket, a pontot kijelöli a kör középpontjának, illetve az egy tömbben levő metszéspontok pixelektől való távolságát átlagolva a sugár értékét is jó közelítéssel adja meg.

Az algoritmus könnyen átültethető a Kinectes körfelismerés problémájára. Pixelek helyett a tárolt pozíciók koordinátáit vesszük. A ponton átmenő gradiens irányát a 2 szomszédos pontja által meghatározott szög szögfelezője adja, ez az egyszerűbb közelítés csökkenti a számítások komplexitását, és a gyakorlati tapasztalatok alapján megbízható eredményeket ad. Az alábbi ábrán látható az algoritmus megjelenítése egy teszt ponthalmazon.



9. ábra Az algoritmus szemléltetése tesztpontokon

4.3.2.2 Algoritmus implementálása

A konkrét implementáció megvalósításánál felmerült problémaként, hogy ha a felismerő algoritmust minden új csontváz adatnál többször is le kell futtatni a pontok részletes elemzéséhez, az teljesítménybeli problémákhoz vezet. Ezért a cél az volt, hogy az algoritmus detektálásonként minél kevesebbszer fusson le, ideálisan egyszer. Ehhez az indítása előtt a tárolt ponthalmaz jelentős szűrésen esik át.

Első körben két közelítő property használatával kerülnek szűrésre a pontok. A *MinimalDetails* megadja, hogy legalább hány pontból álljon a kör, az *EndpointCloseness* pedig hogy milyen távolságra legyen egymáshoz a kezdő és végpont. A szűrés célja kiválasztani azokat a részhalmazokat a pontok közül, amelyek kezdő és végpontja elég közel van egymáshoz, hogy akár kör is lehessen és legalább a minimális számú pontból állnak. A szűrést tatalmazó LINQ kifejezés:

```
from o in StateHistory
from p in StateHistory
where                              RelativeCheck                              ?
o.RelativePosition.DistanceFrom(p.RelativePosition) <= EndpointCloseness :
o.AbsolutePosition.DistanceFrom(p.AbsolutePosition) <= EndpointCloseness
&& StateHistory.IndexOf(o) < StateHistory.IndexOf(p)
&& StateHistory.IndexOf(p) - StateHistory.IndexOf(o) >= MinimalDetails
select                           StateHistory.GetRange(StateHistory.IndexOf(o),
StateHistory.IndexOf(p)-StateHistory.IndexOf(o));
```

Ezután következik egy második szűrés, ami a részhalmazok szórását fogja vizsgálni. Ennek a célja kiszűrni azokat a részhalmazokat, amelyek pontjai nem szóródnak szét annyira az X-Y síkban, mint a kör átmérője (ezt egyébként a *Diameter* property tartalmazza). Ez például akkor fodulhat elő, ha a kezünket egy egyenes mentén mozgatjuk oda-vissza. A szűréshez tartozó LINQ kifejezés:

```
possibleStartPoints = possibleStartPoints.Where(o =>
                Math.Abs(o.Max(p  =>  p.RelativePosition.X)  -  o.Min(p  =>
p.RelativePosition.X)) >= Diameter
                && Math.Abs(o.Max(p => p.RelativePosition.Y) - o.Min(p =>
p.RelativePosition.Y)) >= Diameter);
```

Ezzel a két szűrővel értelemszerűen nem kerülnek olyan részhalmazok kiszűrésre, amelyek valós eredményt adhatnának, ellenben a legtöbb hamis eredményt adó részhalmaz nem jut át mindkettőn. A gyakorlati tapasztalat szerint a felismerő algoritmust elég a fennmaradó részhalmazok egyikére lefuttatni, valószínűleg azért,

mert nagyrészt egyébként is fedik egymást. A konkrét implementációban ezért csak az első részhalmazra futtatom az algoritmust.

Az algoritmus lényegi részét a *FindCircle* metódus tartalmazza. A bemenetben kapott pontok közül 20-at választ ki, ezekhez kiszámolja a szögfelező egyenes paramétereit. Ehhez először kiszámolja a pont és a két szomszédjához húzott egyenesek meredekségét:

```
private float GetSlope(SkeletonPoint A, SkeletonPoint B)
{
    return (A.X - B.X) != 0 ? (A.Y - B.Y) / (A.X - B.X) : (A.Y - B.Y) /
(0.0000001f);
}
```

A függőleges vonalak X tengelybeli változását egy kis számmal közelítettem, hogy értelmezhetővé váljanak, ez a közelítés az algoritmus hatékonyságát nem befolyásolja.

A két meredekségből a szögfelező egyenes paramétereit az iránytangensével adott egyenes egyenletéből kiindulva a

$$m = -\frac{1}{\frac{m_1 + m_2}{2}} = -\frac{2}{m_1 + m_2}$$

$$b = Y_0 - m * X_0$$

egyenletekkel számolom ki.

Az így kapott (m meredekséggel és b Y irányú eltolással jellemzett) egyeneseknek ezután ki kell számolni az összes metszéspontját, ezek szintén egyszerű matematikai műveletekkel számolhatóak:

$$x = \frac{b_2 - b_1}{m_1 - m_2}$$

$$y = m_1 * x + b_1$$

A következő lépésben maximumkereséssel kiválasztom azt a metszéspontot, amelyikhez *dmin* távolságon belül a legtöbb másik metszéspont található. Ha ez a maximum érték nagyobb, mint egy másik érték, *Tmin* akkor a felismerés sikeres, és az érintett metszéspont lesz a felismert kör középpontja. A *dmin* és *Tmin* az osztályhoz

tartozó konstansok, értékük 0.5 és 100 – ezekkel a gyakorlatban előforduló, sokszor nagyon szabálytalan (de körszerű) ponthalmazokat is felismeri, ahogy az alábbi ábrán is látható:

10. ábra Az algoritmus által körként felismert ponthalmazok

4.3.3 Összetett gesztusok

Az előző két elemi gesztus kombinációival komplex gesztusok felismerése is lehetővé válik. Ilyen például a mindkét kézzel egyszerre körzés, egy előrerúgás utáni kézmozdulat, vagy egy kétkezes lökő mozdulat. Az ilyen és ehhez hasonló mozdulatsorok felismerésére szolgál a *CombinedGestureRecognizer*. Az osztály az őt alkotó elemi gesztusokat lépésekre osztja, az egy lépésen belüli mozdulatoknak párhuzamosan kell bekövetkezniük, míg maguk a lépések sorosan követik egymást. Ha egy lépést követően egy bizonyos ideig nem sikerül a következő lépés gesztusait teljesíteni, akkor a detektálás újra az első lépéstől indul – ezt a Tolerance property tartalmazza.

Az összetett gesztusok egyszerűbb felépítésére szolgál a GestureBuilder osztály, aminek az *AddStep* metódusán keresztül láncolt formában hozhatunk létre egy *CombinedGestureRecognizer* objektumot. Példa a használatára (ahol a leftpush, rightpush, leftcircle már korábban példányosított felismerő objektumok):

```
CombinedGestureRecognizer cg = new GestureBuilder()
          .AddStep(leftpush, rightpush)
          .AddStep(leftcircle).Instance;
```

4.4 Hangfelismerés

A hangfelismerés teljes egészében a Microsoft Speech Platform lehetőségeit használja ki, nem volt szükség plusz absztrakciós szintre, amely új funkciókkal bővítené azt. Mindezek mellett, a 4.5.1 pontban bővebben ismertetett KinectSensorManager segédosztály a Kinect funkciók mellett a hangfelismerés fő osztályait is összefogja, a könnyebb kezelhetőség érdekében.

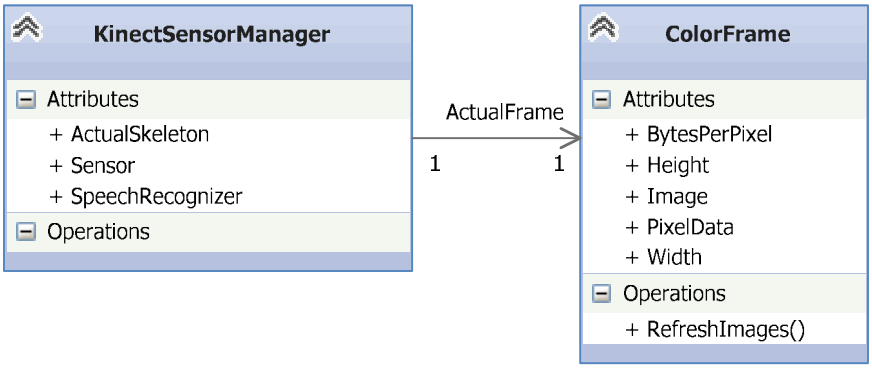
4.5 Segédosztályok

A felismerők mellett segédosztályokat is készítettem, amik szintén a Kinect szenzor könnyebb integrálását segítik elő, ezek a *Kinect.Helper* névtérben találhatóak. A legnagyobb ilyen osztály a *KinectSensorManager,* ami a Kinect csontváz-, és RGB-képkezelését fogja össze egy könnyen konfigurálható helyre. Emellett található még a kézzel vezérelt kurzorok képernyőre skálázását segítő *CursorMapper* osztály, illetve a szintén a képkezelést segítő *ColorFrame* osztály.

A segédosztályok mellett találhatóak még benne kiegészítő metódusok a SkeletonPoint objektumokkal végezhető műveletekre, pl. a *DistanceFrom* metódus, ami megadja két pont távolságát.

4.5.1 KinectSensorManager

Az osztály célja a Kinect kép- és csontvázkezelésének elrejtése a felhasználó elől, az ezeket lekezelő a kódok a legtöbb Kinect alkalmazásban ugyanolyanok. Az osztály gyakorlatilag egy wrapperként funkcionál az eredeti SDK KinectSensor osztálya felett, megkönnyítve annak használatba vételét. Az becsomagolt objektum a *Sensor* property alatt el is érhető. Az osztály szerkezete:



11. ábra A KinectSensorManager és a ColorFrame osztály

A belső megvalósításban az osztály kezeli a KinectSensor SkeletonFrameReady és ColorFrameReady eseményeit, amelyek a szenzorból érkező nyers pixel és csontvázadatokat dolgozzák fel.

Az RGB adatokat a hozzátartozó *ColorFrame* objektumnak adja át, ennek a feladata ezeket egy WritableBitmap objektummá alakítani (ez az *Image* property-jén keresztül érhető el). Ez az objektum már könnyen megadható a legtöbb WPF-es objektum hátterének forrásaként, és a folyamatos frissítése is biztosítva van.

A csontvázadatokat feldolgozásakor csak azokat a csontvázakat vesszük figyelembe, amelyeknél a testrészek is követve vannak. Erre azért van szükség, mert az SDK azt is csontvázadatként küldi el, ha csak felismert egy testet, de még nem azonosította be a részeit – ezekkel azonban a felismerő osztályok nem tudnak mit kezdeni. Ha több csontvázat is felismer a szenzor, akkor automatikusan a legközelebbi lesz az aktuálisnak jelölt csontváz:

```
ActualSkeleton=trackedSkeletons.First(o =>o.Position.Z == trackedSkeletons.Min(s
=> s.Position.Z));
```

Látható, hogy az osztály nem rendelkezik publikus metódusokkal, a kommunikációt eseményeken keresztül bonyolítja. A csontvázkezeléshez tartozó események a *NewSkeletonSet*, ami az aktuálisan követett csontváz megváltozásakor sül el (pl. ha az aktuális felhasználó és a szenzor közé áll valaki); az érvényes csontvázadatok beérkezésekor elsülő *SkeletonLost;* illetve a csontvázadatok frissülésekor elsülő *SkeletonFrameReady.* Ezeken kívül a képkezelés inicializálása után sül el a *ColorFrameInitialized,* ez jelzi azt hogy az *ActualFrame.Image* property hozzáköthető a hátterek forrásához.

Az osztály része még a hangfelismerésért felelő *SpeechRecognitionEngine* objektum. Itt a *KinectSensorManager* egyszerű wrapper-ként viselkedik, nem ad plusz funkciókat a hangfelismeréshez, csak logikailag odakapcsolja a többi Kinect-es funkció mellé. Inicializáláskor automatikusan példányosítja, amennyiben lehetséges a felhasználó nyelvén, különben az alapértelmezett amerikai angol nyelven (mivel a magyar nyelv nem támogatott, az utóbbi változat fog teljesülni):

```
SpeechRecognizer=new
SpeechRecognitionEngine(SpeechRecognitionEngine.InstalledRecognizers().Fir
st(o => o.Culture.Name == CultureInfo.CurrentCulture.Name ||
o.Culture.Name == "en-US"));
```

4.5.2 CursorMapper

Kinect alkalmazások fejlesztésekor sokszor előfordul, hogy a kezünket használjuk az alkalmazás kurzoraként. Ilyenkor probléma lehet, hogy nem tudjuk az egész ablakot kényelmesen elérni a kezünkkel, csak ha elég messzire állunk az érzékelőtől. Ennek megoldására át kell skálázni a kurzor kirajzolását, hogy igazodjon a képernyő teljes területéhez, ehhez nyújt segítséget a *CursorMapper* osztály, amelyen keresztül be lehet állítani a skálázás paramétereit, és a *MapColorPositionToScreen* metóduson keresztül a paraméterként kapott képernyőpontot a megfelelő helyre skálázza át.

5 Az alkalmazás

A felismerő rendszer képességeit bemutató WPF alkalmazás a GameCollection nevet kapta. Két kisebb alkalmazást fog össze, egy memóriajátékot és egy Tetrist. A két játék egy közös keretet használ, ami a játékon belüli menüket és az egész alkalmazás főmenüjét foglalja magába, illetve az ezek közötti navigálást kezeli. A keretrendszer két WPF ablakból, és egy, a játékok és a belső menü közötti kommunikációhoz szükséges interfészből áll.

5.1 Főmenü

Az induláskor a főmenüt megvalósító MainWindow ablak látható:

12. ábra Főmenü

A két kör a kezekhez rendelt kurzorokat jelzi. Az ablak induláskor példányosít egy KinectSensorManager objektumot, amit később paraméterként fog megkapni az összes többi ablak is, ezzel is csökkentve a példányosításból származó overheadet. A megoldás velejárójaként figyelni kell rá, hogy az eseménykezelők csak az aktuális ablakra fussanak le, szintén az overhead elkerülése végett. Ez a főmenünél az ablak láthatóságának változásához van rendelve az alábbi módon:

```
void                MainWindow_IsVisibleChanged(object                sender,
DependencyPropertyChangedEventArgs e)
        {
            if ((bool)e.NewValue)
            {
                _sensorManager.SkeletonFrameReady += OnNewSkeletonReady;
            }
            else
            {
                _sensorManager.SkeletonFrameReady -= OnNewSkeletonReady;
            }
        }
```

Az ablakon belül 2 érzékelő van példányosítva, mindkét kézre egy-egy, a klikkelés mozdulatot figyelő *CombinedGestureRecognizer*. Ez egy előre és egy hátramutató *LinearGestureRecognizer* kombinációja. Gesztus érzékelésekor a program megvizsgálja az adott pozícióban található-e valamilyen vezérlő (ebben az esetben Button), majd pozitív találat esetén kiküldi a Click eventet:

```
IInputElement hitted = this.InputHitTest(new Point {X =pos.X, Y =pos.Y });
  if (hitted != null)
  {
   hitted.RaiseEvent(new RoutedEventArgs{RoutedEvent=Button.ClickEvent });
  }
```

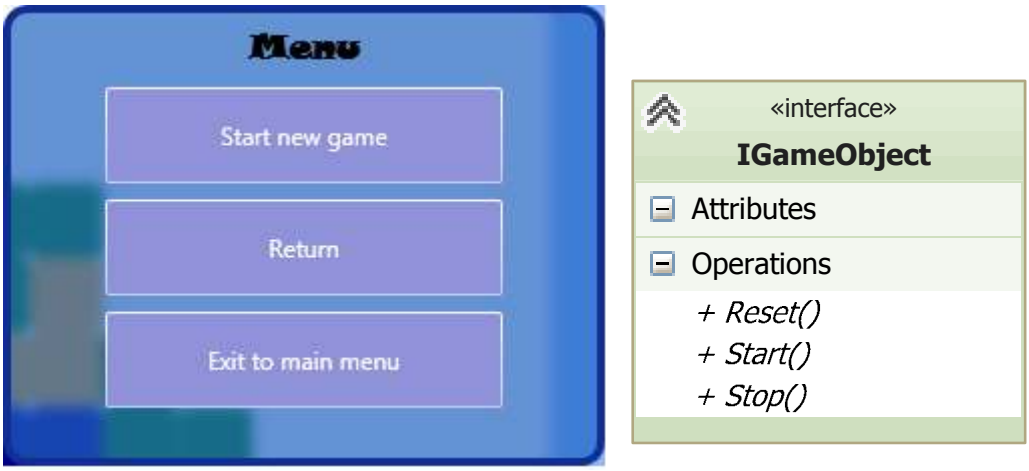
Innen a szerep a gombok eseménykezelőié, amelyek elrejtik a főmenü ablakát, majd példányosítják, és megjelenítik a játékét:

```
this.Hide();
new MemoryGame.GameWindow(_sensorManager) { Owner = this }.Show();
```

Látható, hogy paraméterként átkerül a KinectSensorManager példány.

5.2 In-game menü

A keret része még az in-game menü ablaka, az *InGameMenuWindow*, mindkét játék ezt használja:



13. ábra az In-game menü (háttérben a Tetrissel) és az IGameObject interfész

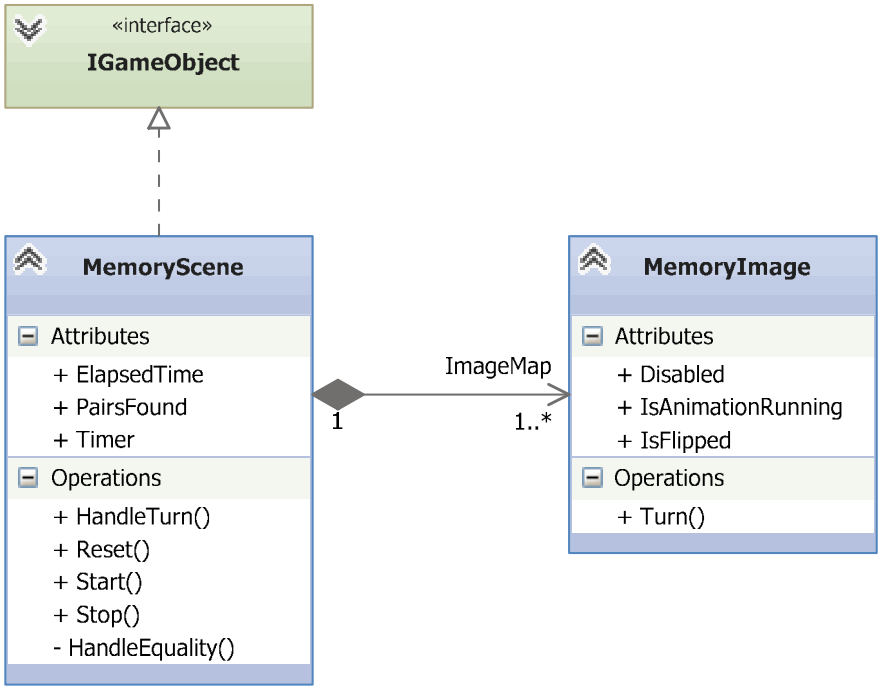
Az ablak ugyanolyan módon vezérelhető, mint a főmenü. A vezérlés a fentebb látható *IGameObject* interfészen keresztül történik, ezt implementálják a játékok logikáját megvalósító objektumai, és paraméterként átkerülnek az in-game ablakhoz. A gombok eseménykezelői pedig értelemszerűen meghívják rajtuk a megfelelő metódusokat. Példaként a „Start new game” gomb eseménykezelője:

```
private void OnReset(object sender, RoutedEventArgs e)
{
    gameObject.Reset();
    gameObject.Start();
    this.Close();
}
```

5.3 Memorygame

A memóriajáték a *GameCollection.Memorygame* névtérben található, a megjelenítéséért a *GameWindow* ablak felelős. A játék logikája ettől teljesen külön van választva, először ezt mutatom be.

5.3.1 Logikai réteg



14. ábra MemoryGame fő osztályai

A *MemoryImage* egy WPF User Control, ami a memóriajáték egy oda-vissza forgatható elemét hivatott megvalósítani. 2 egymásra helyezett képből áll, egyiken a hátlap, másikon az előlap képe látható – egyikük mindig rejtett. A képek animálással mennek át egyik állapotból a másikba, ezt *Storyboard*-ok használatával értem el. Két *Storyboard* van definiálva (kifelé és befelé forgatásra), amik a kép skálázását és láthtóságát animálják – itt látható a kifelé forgatás storyboardja:

```
<Storyboard x:Key="flipfront">
   <DoubleAnimation                    From="1"                    To="0"
Storyboard.TargetProperty="(UIElement.RenderTransform).(ScaleTransform.Sca
leX)"></DoubleAnimation>
   <ObjectAnimationUsingKeyFrames Storyboard.TargetProperty="Visibility">
      <DiscreteObjectKeyFrame Value="{x:Static Visibility.Hidden}"/>
   </ObjectAnimationUsingKeyFrames>
 </Storyboard>
```

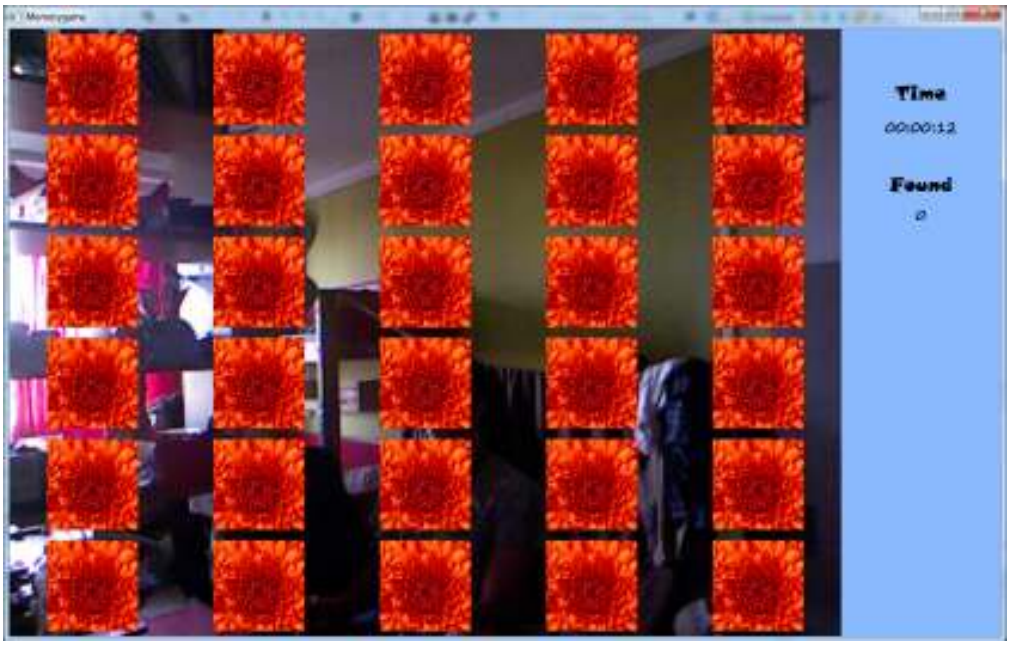
Az animáció indítására a *Turn* metódus szolgál, az osztály kezeli, hogy futó animáció alatt ez nem hajtódik végre. Az osztály property-jei értelemszerűen az objektum aktuális állapotát adják meg, ezeket a *MemoryScene* használja majd fel a forgatások indításakor.

A *MemoryScene* valósítja meg a játék lényegi logikáját. Központi eleme az *ImageMap* property, ami az ismertetett *MemoryImage* objektumok mátrixát tartalmazza, így rendelve hozzájuk a játéktéren való helyüket. Megvalósítja az in-game menünél már ismertetett IGameObject interfészt, illetve az adatkötéshez szükséges INotifyPropertyChange WPF interfészt.

A *Start* és *Stop* metódusai csak a beépített Timer objektumot szabályozzák, a *Reset* metódus teljesen újrainicializálja a játékteret. Ehhez egy, a konfigfájlban megadott köyvtárból kiolvassa az ott található jpg fájlokat, ezeket használja majd a MemoryImage objektumok előlapjaként (a hátlapot hasonló módon előre konfigurált helyről olvassa be). Minden párhoz véletlenszerűen keres két, még nem lefoglalt helyet az ImageMapon belül, és oda példányosít egy-egy MemoryImage-et. A *HandleTurn* metódus kezeli le a blokkok forgatását, figyel arra, hogy egyszerre csak két kép foroghasson kifelé, illetve ha két kiforgatott kép után kattintunk egy harmadikra, akkor automatikusan elkezdi visszaforgatni az első két képet. Ha folyamatban van két kép kiforgatása, lefuttatja rájuk az egyezőség ellenőrzését, ami ha pozitív, a *HandleEquality* metódus érvényteleníti őket és elsüti rájuk a *RemoveImage* eseményt, ez jelez majd a megjelenítő rétegnek.

5.3.2 Megjelenítési réteg

A *GameWindow* ablak feladata a megjelenítés és a vezérlés összekapcsolása a játék magjával.



15. ábra Memorygame

Ehhez indulásakor feliratkozik a paraméterként megkapott KinectSensorManager objektum eseményeire, illetve példányosít egy MemoryScene objektumot, ez utóbbit DataContext-ként is beállítja, hogy a képen is látott idő és „talált párok" adatok adatkötéssel automatikusan frissüljenek. A két kézhez rendelt kurzor kirajzolásához egy CursorMapper objektum nyújt segítséget.

A vezérléshez a menükhöz hasonló módon a két kézre külön-külön beállított klikkelés gesztust használja, illetve az in-game menü előhozására a két kézzel egyszerre való klikkelést (itt az volt a cél, hogy a mozdulat ne legyen túl komplikált, de mégis elég nehéz legyen véletlenül előidézni). Az érzékelőket induláskor konfigurálja fel, és minden új csontvázadatnál frissíti őket (ugyanekkor rajzolja ki a kurzorokat is).

A játéktér hátterének a szenzor RGB-kameraképét használja, ez a már leírt kialakításnak köszönhetően egy sorral elérhető:

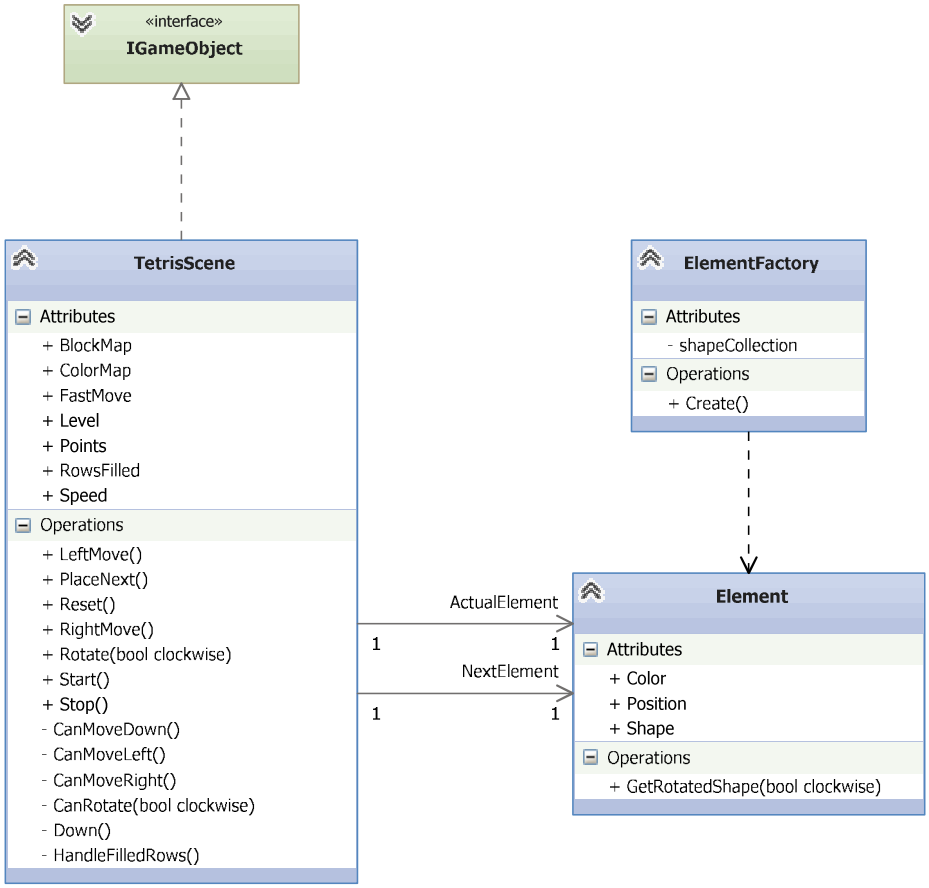
```
backgroundImage.Source = _sensorManager.ActualFrame.Image;
```

Az *InitScreen* metódus rajzolja ki egy Canvas objektumra a játékteret. A blokkok mérete és a szabályos elhelyezésükhöz szükséges paraméterek itt számolódnak ki. A metódus induláskor és minden átméretezéskor lefut.

5.4 Tetris

A Tetris alkalmazás a *GameCollection.Tetris* névtérben található, felépítése ugyanazokat az elveket követi minta memóriajátéké, ezért a bemutatást itt is a logikai réteggel kezdem.

5.4.1 Logikai réteg



16. ábra Tetris fő osztályai

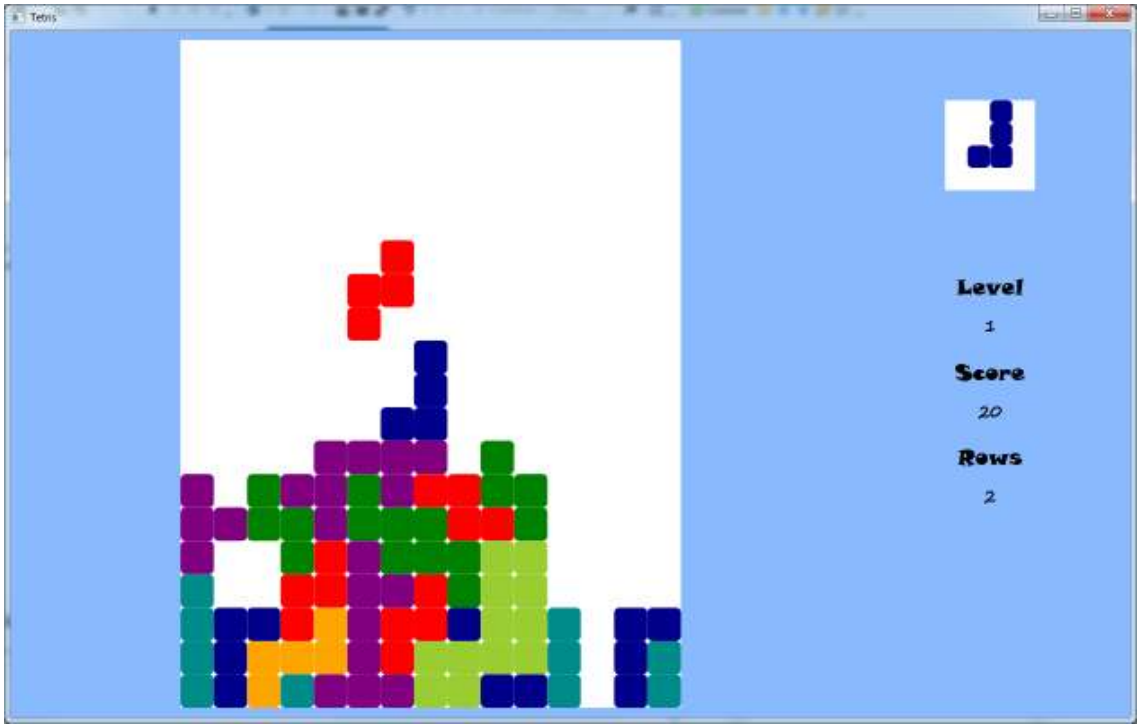
Az *Element* osztály valósítja meg magukat az alakzatokat. Fő tulajdonsága a *Shape*, ami egy bool mátrixszal bitmapként megadja magát az alakzatot. A *Position* a játéktéren belüli eltolást tárolja a mátrix első eleméhez (bal felső sarok) viszonyítva. A *GetRotatedShape* az alakzat elforgatott alakját adja vissza.

Az *ElementFactory*, ahogy a neve is jelzi, Element objektumok létrehozására alkalmas, gyakorlatilag a klasszikus Tetris alakzatok valamelyikének megfelelő Element objektumot ad vissza a *Create* metódusával.

A *TetrisScene* adja a játéktér logikai reprezentációját, megvalósítja az *IGameObject* interfészt. A *BlockMap* property bitmapban tárolja a játéktér már foglalt részeit, a *ColorMap*-ban az egyes blokkok színeit. Az aktuális és következő Element objektumokat is tárolja, ezek a játék folyamán értelemszerűen frissülnek. A *Points*, *Level*, és *RowsFilled* propertyk változáskor elsütik a PropertyChanged eseményt, így az UI-n automatikusan frissülnek. A mozgató metódusokhoz rendre tartozik egy ellenőrző metódus, amelyik azt vizsgálja elvégezhető-e az adott mozgás – ha nem akkor nem hajtja azokat végre. A lefelé mozgatást végző *Down* metódus kívülről nem elérhető, ezt a belső időzítő hívja a játék sebességétől függő időközönként. A Tetrisből ismert gyorsan lefelé való mozgást a *FastMove* propertyvel lehet kapcsolgatni. A játéktér bármilyen frissítésekor elsüti a Refresh eseményt.

5.4.2 Megjelenítési réteg

Itt is a *GameWindow* ablak adja a megjelenítést.



17. ábra Tetris

Indításkor példányosít egy *TetrisScene* objektumot, majd beállítja DataContextként. Lényegesebb metódusa a rajzolást végző OnRefresh metódus, ami a TetrisScene objektum Refresh eseményének kezelője – ez rajzolja ki a megfelelő méretben a játékteret, az aktuális és következő elemeket.

A felismerő objektumok közül szinte minden osztályból alkalmaz egy elemet a vezérléshez. Az oldalsó mozgásokat a magunk mellé kinyújtott megfelelő kézzel vezérelhetjük, ezt két *PostureDetector* objektum valósítja meg. A gyors lefelé mozgást akkor kapcsoljuk be, ha mindkét majdnem teljesen kezünk magunk mellett tartjuk felemelve, ez egy *CombinedPostureDetector* objektumon elérhető. A jobb/bal irányba forgatást a jobb/ bal kezünkkel való körzéssel érjük el (*CircularMotionRecognizer* objektumok), illetve a memóriajátéknál leírt „egyszerre klikkelés" gesztus itt is az in-game menüt hozza elő.

6 Fejlesztési lehetőségek

6.1 Körfelismerés továbbfejlesztése

A körfelismerés implementálni lehetne a 4.3.2 pontban már vázolt másik megközelítéssel, template-ek alkalmazásával, ehhez sok már meglévő algoritmus áll rendelkezésre. Egy ilyen felismerő létrehozásával akár össze is lehetne hasonlítani a két megközelítést teljesítmény és stabilitás szempontjából.

A jelenlegi implementáció stabilitás szempontjából kielégítő ugyan, de kettőnél több felismerő együttes alkalmazásánál már minden optimalizálás ellenére jelentkeznek teljesítményproblémák.

A jelenlegi algoritmusban fejleszthető még a körzés irányának detektálása illetve az algoritmus kimenetéből adott sugár adatot sem kerül felhasználásra.

6.2 Egyenes vonalú mozgás bármilyen irányba

A jelenlegi implementáció csak a három koordinátatengely mentén vett elmozdulásokat képes érzékelni. Ezt lehetne kibővíteni úgy, hogy tetszőleges irányú mozgásokat is felismerjen, lehetővé téve az átlós mozgásokat. Ennek eléréséhez például a pontokat detektáláskor meg lehetne szorozni egy megfelelő mértékű forgatást tartalmazó transzformációs mátrixszal, és így lehetne vizsgálni (a már működő módon) a két pont közötti távolságot.

6.3 Üresjáratok detektálása

A mozgásérzékelők mellett olyan felismerőt is lehetne készíteni, ami a mozgás hiányát érzékeli – azaz ha egy testrész egy adott tartományon belül marad egy megadott ideig.

6.4 Orientáció figyelembevétele

A Kinect SDK nem csak a testrészek pozícióját képes visszaadni, hanem azok egymáshoz képesti elfordulását is. Ezeket csontokra vonatkoztatva tárolja, mindegyik kezdő és végpontja egy, az eddigi érzékelőkben is használt testrész. Ezeket az információkat a felismerő rendszer sehol nem veszi figyelembe, használatukkal például meg lehetne állapítani, hogy a felhasználó a kamera felé fordul-e, és figyelembe venni ezt érzékeléskor (elkerülve például azokat az eseteket, amikor a felhasználó háttal a kamerának gesztikulál és ezt a rendszer akaratán kívül értelmezi). Az orientáció másfajta felhasználásaként az előző pozíciók rögzítésekor nemcsak a pozíciót lehetne relatívan tárolni, hanem a referencia testrészhez képesti relatív orientációt is, és ezt is felhasználni detektáláskor, például hogy egy egyenes vonalú mozgást akkor is detektáljon, ha közben elfordulunk a testünkkel.

6.5 Több csontváz támogatása

A *KinectSensorManager* képes arra, hogy elrejtse a felhasználó elől a Kinect belső működését, ám nem alkalmas olyan alkalmazásban való használatra, ahol egyszerre több csontváz adatait kell kezelni (maga a Kinect hardver 6 csontváz detektálására képes egy időben). Az osztály ilyen irányú kibővítésével ezeket a több felhasználós alkalmazásokat is támogatná.

7 Konklúzió

A dolgozatban arra próbáltam egy megoldást mutatni, hogyan tehetjük egyszerűbbé a Kinect-alapú alkalmazások fejlesztését. Ennek szükségszerűségét az adja, hogy a platform a rendkívül innovatív megoldásai miatt egyre népszerűbb, több és több ember otthonába jut el, ráadásul most már főleg PC-s kiegészítőként. Ezzel együtt nő a felhasználói igény is arra, hogy minél több alkalmazás támogassa a Kinect-es vezérelhetőséget.

A megoldásom több szempontból elégíti ki ezeket a törekvéseket. A nagyobbik részében egy olyan rendszert terveztem meg, ami hatékonyan tud gesztusokat felismerni, ez a legtöbb Kinect-es alkalmazás alapkövetelménye. A rendszert próbáltam úgy felépíteni, hogy felhasználói szempontból a lehető legegyszerűbben lehessen implementálni, mégis alkalmas legyen bonyolultabb mozdulatok felismerésére. Szempont volt még a továbbfejleszthetőség is, ezért egy olyan architektúrát alakítottam ki, amely könnyen lehetővé teszi új komponensek hozzáadását.

A rendszer másik részében segítő osztályokat készítettem, amelyek egy alkalmazásfejlesztő munkáját könnyíthetik meg. Ezek összefogják a Kinect-el kapcsolatos gyakori, és az esetek többségében ugyanazzal a kóddal megoldott műveleteket, mint a nyers kép- és csontvázadatok feldolgozása. Így született meg a KinectSensorManager osztály, amely jelentősen gyorsíthatja egy alkalmazás Kinect-essé tételét.

Ezeket a funkciókat ezután egy mintaalkalmazáson keresztül mutattam be, amely két egyszerű játékalkalmazáson keresztül részletesen példázza, hogy kell ezt a rendszert implementálni.

Összességében elmondható, hogy a megoldásom nem közvetlenül nyújt Kinect-es felhasználói élményeket, de alkalmas arra, hogy a már meglévő és újabb alkalmazások a segítségével erre minél gyorsabban és hatékonyabban képesek legyenek.

Irodalomjegyzék

[1] Wikipedia: Kinect, http://en.wikipedia.org/wiki/Kinect (revision 02:08, 18. November 2012)

[2] Wikipedia: Structured Light, http://en.wikipedia.org/wiki/Structured-light (revision 00:03, 16. November 2012)

[3] Zalevsky, Zeev; Shpunt, Alexander; Maizels, Aviad; Garcia, Javier: Method and system for object reconstruction, WO/2007/043036, 2005

[4] J. Shotton, A. Fitzgibbon, M. Cook, T. Sharp, M. Finocchio,R. Moore, A. Kipman, A. Blake: Real-Time Human Pose Recognition in Parts from Single Depth Images

[5] MSDN: WPF Architecture, http://msdn.microsoft.com/en-us/library/ms750441.aspx

[6] Wikipedia: Windows Presentation Foundation, http://en.wikipedia.org/wiki/Windows_Presentation_Foundation (revision 08:09, 25. November 2012)

[7] MacDonald, Matthew: Pro WPF in C# 2010, 3rd edition, ISBN 978-1-4302-7205-2, Apress, 2010

[8] MSDN: WPF Threading Model, http://msdn.microsoft.com/en-us/library/ms741870.aspx

[9] Li-qin JIA, Cheng-zhang PENG, Hong-min LIU, Zhi-heng WANG: A Fast Randomized Circle Detection Algorithm

[10] Rob Knies: Kinect Audio: Preparedness Pays Off (report), http://research.microsoft.com/en-us/news/features/kinectaudio-041311.aspx

[11] Michael L. Seltzer, Ivan Tashev, Alex Acero: MICROPHONE ARRAY POST-FILTER USING INCREMENTAL BAYES LEARNING TO TRACK THE SPATIAL DISTRIBUTIONS OF SPEECH AND NOISE