TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Renee Gustasson 164147

MICROSOFT KINECTI RAKENDAMINE LIIGUTUSTE TUVASTAMISEKS INTERAKTIIVSE VIRTUAALSE PUU JAOKS

Bakalaureusetöö

Juhendaja: Eduard Petlenkov
PhD

Tallinn 2019

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autor: Renee Gustasson

20.05.2019

Annotatsioon

Töö eesmärgiks oli uurida, kuidas saaks kasutada Microsoft Kinect versioon 1te virtuaalse puu mängu loomiseks. Leida sobivad vahendid, mida kasutada selle arenduses. Tuli välja selgitada, mis funktsionaalsused on vajalikud Kinecti poolt selle teostamiseks.

Töös sai kasutatud Unity mängumootorit ja kolmanda osapoole poolt loodud lisa Kinectiga töötamiseks. Lisas on olemas vahendid inimese liikumise tõlgendamiseks, mis oli üks komponent selles projektis. Töö loodi C# programmeerimiskeelt kasutades.

Töös uuriti, kuidas teostatakse Kinectiga inimkeha jälgimine ja liigutuste tõlgendamine ning kuidas seda rakendada loodavas mängus. Töö teostati kahe inimese poolt, Renee uuris, kuidas rakendada Kinecti selle toestamiseks ja Kalev, vastutas mängu loogika ja mudelite loomise eest.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 18 leheküljel, 6 peatükki, 11 joonist, 0 tabelit.

Abstract

The aim of this work is to research, how to use Microsoft Kinect version 1 to develop an augmented reality application, by tracking human motion. First objective was to figure out, what Kinect has to do in order to give users augmented reality effect. The idea was to use Kinect´s ability to track human skeleton by assigning 20 points on users' body, using the coordinates of some of these points, a game would be developed, where inside there is a modelled animal, which will try to escape the user. The user will also have the opportunity to grab an object in game, which the animal will start to follow, for that Kinect needs to register user gesture, where the hand makes a fist and give the according information to the game. Also had to find the appropriate tools to make it happen.

The project involved two members, where Renee would find, how to implement Kinect for this application and Kalev how to create a game in Unity, which can be controlled with Kinect.

The thesis is in Estonian and contains 18 pages of text, 6 chapters, 11 figures, 0 tables.

Lühendite ja mõistete sõnastik

DLL	dynamic link lybrary, dünaamiliselt programmi külge kinnitatud teek [11]
SDK	software development kit, tarkvara arenduskomplekt
KinectGestures	Kinect with MS-SDK lisas paiknev skripti fail, mis sisaldab endas liigutuste tuvastamist
KinectManager	Kinect with MS-SDK lisas paiknev skripti fail, milles on meetodid Kinecti funktsionaalsuse rakendamiseks
KinectWrapper	Kinect with MS-SDK lisas olev fail, milles tehakse primitiivseid andmeteisendusi ja imporditakse Kinecti DLL-id
Driver	arvuti programm, kontrollib mingit kindlat seadet, mis on arvutiga ühenduses [10]
Unity Asset store	Unity keskkonnas paiknev pood, milles on lisasi Unity võimaluste suurendamiseks

Sisukord

Sissejuhatus 8
1 Virtuaalse puu mäng Kinecti baasil 9
2 Töö keskkonna ja vahendite valik 10
2.1 Microsoft Kinect versioon 1 10
2.1.1 Kinecti riistvara 11
2.1.2 Kinecti funktsionaalsus 11
2.2 Unity 13
2.3 Kinect with MS-SDK 13
3 Arenduskäik 15
3.1 Programmeerimis keele valik 15
3.2 Unity lisade testimine 15
3.3 Programmi ülesehitus 16
3.4 Kinecti initsialiseerimine 17
3.5 Kaamera pildi peale virtuaalsete objektide lisamine 17
3.6 Inimkehal punktide jälgimine ja nende visualiseerimine 18
3.7 Käemärkide tuvastamine 20
3.8 Lahenduste integreerimine 21
3.9 Esinenud probleemid 22
4 Tulemuste ja edasiarenduse võimalused 23
4.1 Tulemused 23
4.2 Edasiarenduse võimalused 23
5 Analüüs 25
6 Kokkuvõte 26
Kasutatud kirjandus 27
Lisa 1 – [Optimeerimata kood] 28
Lisa 2 – [Lõplik kood] 32

Jooniste loetelu

Joonis 1. Kinect olevad andurid [2] 11
Joonis 2. Masinõppeks kasutatud kehatüübid ja nende variatsioonil (all) [5] 12
Joonis 3. Kinecti poolt määratavad punktid kehal [6] 12
Joonis 4. *AvatarsDemo* näide 14
Joonis 5. Update funktsiooni ülesehitus 16
Joonis 6. Tekstuuri sidumine skripti failiga 18
Joonis 7. Punktide paigutus pildil kuvatuna 19
Joonis 8. Kinectist tulevate koordinaatide visualiseerimine 20
Joonis 9. Objektist kinni haaramine 21
Joonis 10. Objektist lahti laskmine 22
Joonis 11. 20 punkti paigutus kehal 24

Sissejuhatus

Töös uuritakse, kuidas luua virtuaalse puu mäng, kasutades Microsoft Kinectilt tulevaid, kasutaja asukoha andmeid. Esmalt tuli uurida, kuidas peaks arendus käima, ning milliseid vahendeid kasutada. Lahenduse loomiseks kasutatakse Unity mängu mootorit, ning kolmanda osapoole poolt Unity´le loodud lisa Kinectiga töötamiseks, testitud sai erinevaid lisasi rakenduse arenduses.

Projektis kasutatavast lisas on loodud võimalused teostamaks inimkeha jälgimist, mis toimub 20 punkti määramisel kehal ning liigutuste tuvastamist, mida on samuti 20 liigutust kokku, mida lisaga saab tuvastada.

Loodud sai liides, mille läbi saab kasutaja läbi Kinecti suhelda virtuaalse maailmaga, haarata mängus eseme, mille järgi mängus olev modelleeritud loom hakkab tulema [8]. Projektis kasutati arendust C# programmeerimis keelt.

1 Virtuaalse puu mäng Kinecti baasil

Töös sai uuritud meetodeid, kuidas rakenda Microsoft Kinect versioon ühte, interaktiivse mängu loomiseks liitreaalsuses. Idee tuli Mektrotyst asuvast taimetoast, kuhu soovitakse luua lahendus, mis näitaks hoone innovatsiivsust selle külastajatele. Seda silmas pidades sai mõeldud luua liitreaalsuses mäng, mille keskmes on toa keskel paiknev puu, mida oli plaanis kasutada lahenduse välja töötamises.

Puu peale oli plaanis modelleerida virtuaalne loom, kes puud mööda liigub vastavalt kasutaja asukohale. Puu peale oli soov luua modelleeritud ese või toit, mida on võimalik kasutajal kätte haarata pärast, mida läheneb loom inimesele ja püüab seda kätte saada [8]. Mängu tegevus on vaja kuvatakse ruumis olevale ekraanile.

Tegemist on meekonna tööga, mis jagunes kaheks tööks. Osa projektist on kirjeldatud Kalevi „Interaktiivse virtuaalse puu kontseptsioon ja mängu prototüüp“ [8] lõputöös. Minu roll projektis oli Kinecti abil inimese liigutuste tuvastamise teostamine. Kalev mõtles välja mängu loogika toimimise ja lõi mängu mudelid.

2 Töö keskkonna ja vahendite valik

Teostamiseks sai kaalutud kasutada Unreal Engine 4 või Unity mängumootorit. Unreal Engine 4 on varem kasutusel olnud varasema projekti raames, mis algselt oleks olnud esimene valik arenduskeskkonnaks. Uurides võimaluste kohta ja pidades nõu juhendajaga, sai otsustatud tutvuda lähemalt Unity võimalustega idee teostamiseks.

Andurite poolt tuli välja mõelda, mis andmeid läheb kasutaja kohta vaja. Liitreaalsuse teostamiseks on kindlasti vaja kasutaja asukoha andmeid, liigutuste tõlgendamist ja kaamera pilti taustal toimuvast. Seda silmas pidades tundus ainuke sobiv seade olevat Microsoft Kinect, millel on vajalikud andurid koondatud. Töös kasutati Kinect versioon 1te, mis oli võimalik ülikooli poolt kasutusse saada.

2.1 Microsoft Kinect versioon 1

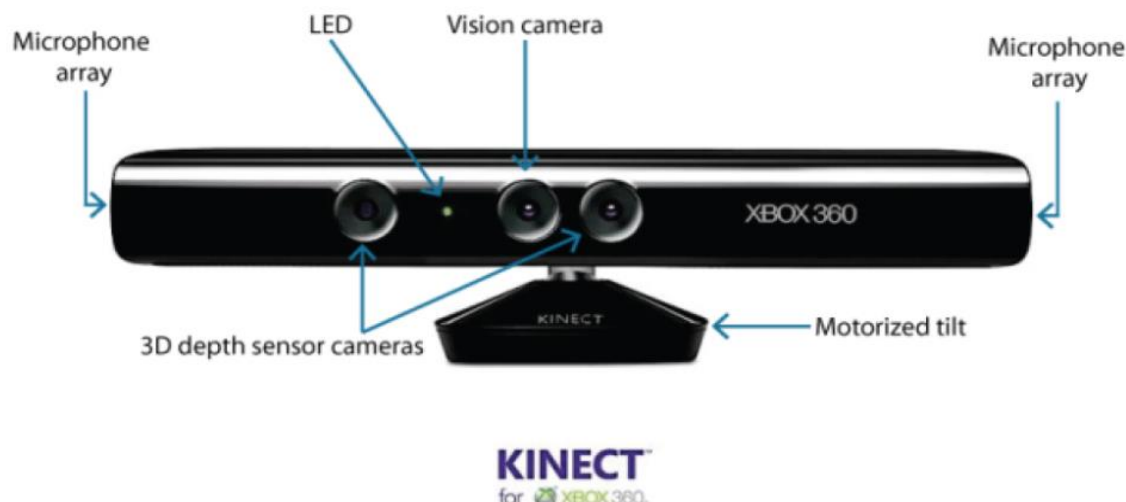
Kinecti tutvustati maailmale esimest korda 1. juuni 2009 ja tuli välja 4. november 2010. See töötati välja Microsofti poolt eesmärgiga laiendada Xbox 360 konsooli kasutajate baasi [1]. Seade suurt populaarsust mängurite seas ei saavutanud.

Seade sai otstarbe hoopis kasutust teadlaste ja kolmandate osapoolte arendajate poolt, kes nägid Kinecti kui võimekas andur, mille hinna funktsionaalsuse suhe oli parim turul. Adafruit Industries kuulutas välja avatud koodiga driveri välja töötamise eest autasu, et saaks kasutada seda mitte kommerts eesmärkidel ning 6 päeva hiljem 10. Novembril 2010 lunastati autasu. Microsoftile ei meeldinud sellise konkurssi välja kuulutamine [1].

See pani Microsofti seda seadet arendama rohkem selles suunas, et saaks kasutada seda mitte kommerts eesmärkidel [1], andes välja Kinecti arvutile ja sellega ka tarkvaraarenduskomplekti 1. veebruar 2012, mis sisaldas endas võimalusi kasutada Kinecti funktsionaalsusi. Loodud komplekt sisaldas koodinäiteid ja toetas arendust C++, C# ja Visual Basic .NET programmeerimis keelte kaudu.

2.1.1 Kinecti riistvara

Kinect koosneb 3 erinevat tüüpi anduritest: mikrofonide massiivist, sügavuskaamerast ja värvikaamerast. Seadmel on olemas mootor, nurga kontrollimiseks.



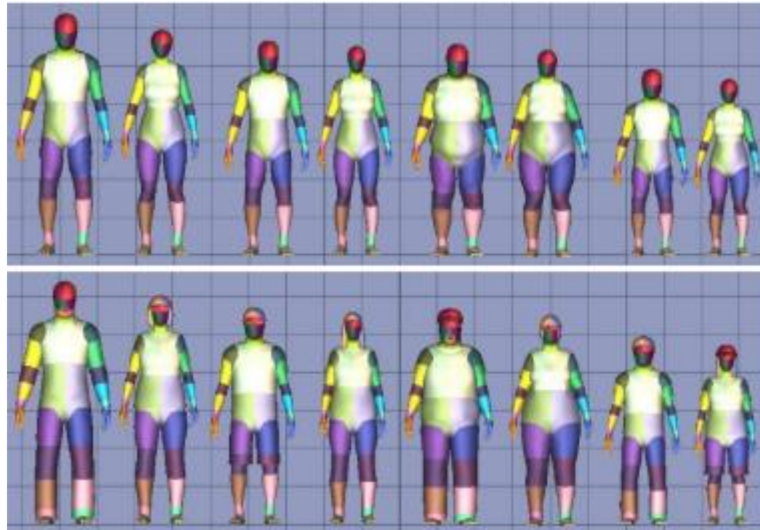
Joonis 1. Kinect olevad andurid [2]

Sügavusandur koosneb kahest komponendist. Esimene komponent on infrapunase valguse projektor ja teine infrapunase valguse kaamera. Projektor kuvab ruumi kindla mustriga valguse, mille muutusi analüüsitakse, kui see pinnalt tagasi kaamerasse peegeldub ja selle põhjal määratakse punktide kaugus Kinectist [3], sügavuskaamera, kui ka värvikaamera resolutsioon on 640x480 ja kuvavad kuni 30 raami sekundis. Mikrofonide massiiv asub seadme alumise osa küljes ja koosneb neljast mikrofonist.

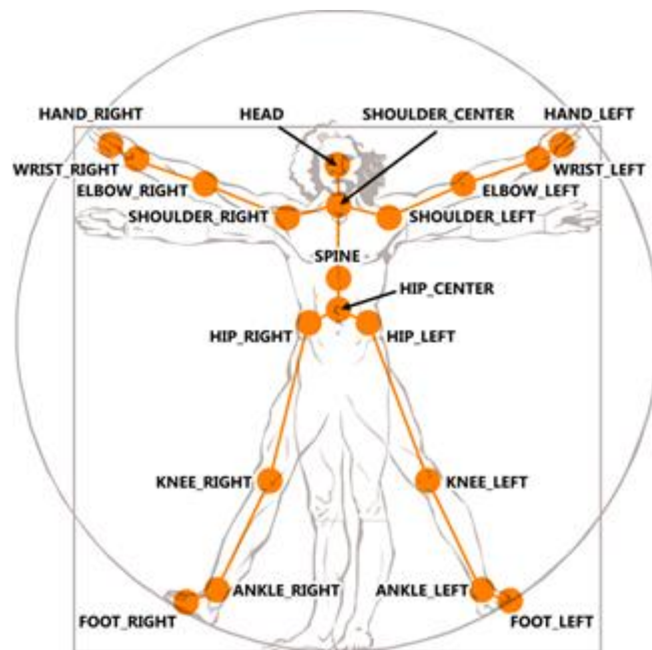
2.1.2 Kinecti funktsionaalsus

Kinectil võimaldab inimkeha jälgimist koos punktide määramisega kehal, näo jälgimist, hääl juhtimist, heli allika asukoha määramist, sügavuskaamera andmete töötlemist ja inimese liigutuste tõlgendamist. Selles projektis sai kasutatud inimkeha jälgimist ja kasutaja liigutuste tõlgendamist. [1]

Inimkeha jälgimiseks määrab Kinect 20 punkti inimese kehal näha (joonis 3), seda tehakse analüüsides värvikaamerast ja sügavuskaamerast tulevaid andmeid [5]. Selle teostamiseks kasutati masinõpet, mille treenimiseks kasutati 15 inimese kehatüübi mudeleid, mille tekstuurides tehti suvalisi muudatusi pikkuses ja laiuses ±10% ulatuses [5] (joonis 2).



Joonis 2. Masinõppeks kasutatud kehatüübid ja nende variatsioonil (all) [5]



Joonis 3. Kinecti poolt määratavad punktid kehal [6]

2.2 Unity

Unity on mängumootor, mis loodi 2005. aastal [6]. See võimaldab arenda rakendusi erinevatele platvormidele nii kahe- kui kolmemõõtmelisi mänge. Mängu arenduse võimaluste tõstmiseks on olemas ka *Unity Asset Store*, kus on loodud nii *Unity Technoologies* poolt, kui ka kasutajate endi poolt lisasi, mida võimalik mängu arenduses kasutada.

Unity valiti arenduse keskkonnaks, kuna tegemist on tasuta programmiga ja seda soovitas ka juhendaja. Lisaks oli seal olemas lisad, millega sai Kinecti rakendust arendada.

Unity´s loodud skriptidel on kaks põhi funktsiooni *start* ja *update*, esimest funktsiooni kutsutake välja ainult korra programmi käivitamisel, mis sobib konstantide deklareerimiseks, teist kutsutakse teatud ajaühiku vahel välja, mis sobib muutuvate suuruste arvutamiseks.

2.3 Kinect with MS-SDK

Lisa on väljastatud 15. aprillil 2013 ning viimane uuendus on tehtud 11. Augustil 2015, selle on loonud Rumen Filkov, teadlane Vorarlbergi ülikoolis [7]. Lisa koosneb näidetest ja mahukatest skripti failidest, mis annavad ülevaate, kuidas rakendada Kinecti funktsionaalsuseid. Lisa toimiseks on vajalik:

- Kinect SDK 1.8
- Unity versioon 2014.4.6.1 kuni 2018.3.9f1

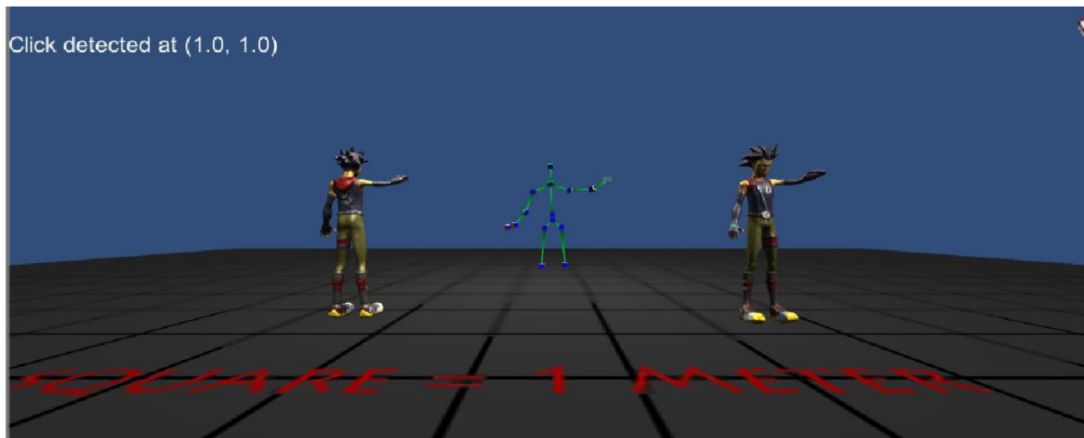
Lisa koosneb kolmest mahukast põhi skriptist *KinectManager.cs, KinectWrapper.cs* ja *KinectGestures.cs*, mille kaudu on Kinecti funktsionaalsused realiseeritud [9].

KinectManager sisaldab endas meetodeid ja seadistusti Kinectiga töötamiseks, antud skriptiga on töös kõige rohkem pistmist. Läbi selle initsialiseeritakse Kinect, tuvastatakse kasutaja, saadakse jälgitavad punktid ja tuvastatakse liigutusi.

KinectWrapper sisaldab endas primitiivseid andmeteisendusi nii öelda pakendatakse andmeid [12], et oleks lihtsam programmeerida ja lisaks Kinecti DLL-ide importimist.

KinectGestures sisaldab liigutuste tuvastamiseks meetodeid, mida kasutab ka *KinectManager* skript. Skript koosneb 3 põhimeetodist, liigutuse tuvastus, liigutuse lõpetus ja kas liigutus on vahepeal katkestatud, ajalise parameetriga liigutuste jaoks, nagu käega viipamine või hüppamine.

Lisas olevates näidetest sai kasutatud *AvatarsDemo* ja *OverlayDemo*. *AvatarsDemo* näites kontrollitakse virtuaalseid avatare Kinecti kasutades (joonis 4). *OverlayDemo* näitab, kuidas virtuaalseid objekte kaamerast tulevale pildile lisada.



Joonis 4. *AvatarsDemo* näide

3 Arenduskäik

Esmalt sai proovitud erinevaid vahendeid rakenduse arenduseks, probleemide esinemisel sai osad vahendid välja vahetatud. Katse eksitus meetodil jäi lõpuks arenduskeskkonnaks Unity, kus *Unity Asset store*-is olevat lisa kasutades töö teostatud sai.

Algselt oli vaja tutvuda C++ ja C# keelega, et otsustada kumba programmeerimiskeelt kasutades rakendust arendada. Esialgne idee oli luua programm Visual Studio kaudu, mis edastaks vajaminevad andmed Unity keskkonda. Selle jaoks oli mõte kasutada UDP protokolli kahe programmi vaheliseks suhtluseks.

3.1 Programmeerimis keele valik

Tööd sai alustatud C# keelega, kuna antud keel on kasutusel ka Unity`s endas, tundus see mõistlik valik olevat, vältimaks ühilduvuse probleeme töö lõppjärgus. Alustades arendust, tekkisid esimesed probleemid Kinecti DLLide sidumisel projektiga, nimelt ei tuvastanud Visual Studio Kinecti DLL-le, antud probleem sai ületatud, kuid projektiga tööd tehes ei saanud Kinect initsialiseerimist tööle.

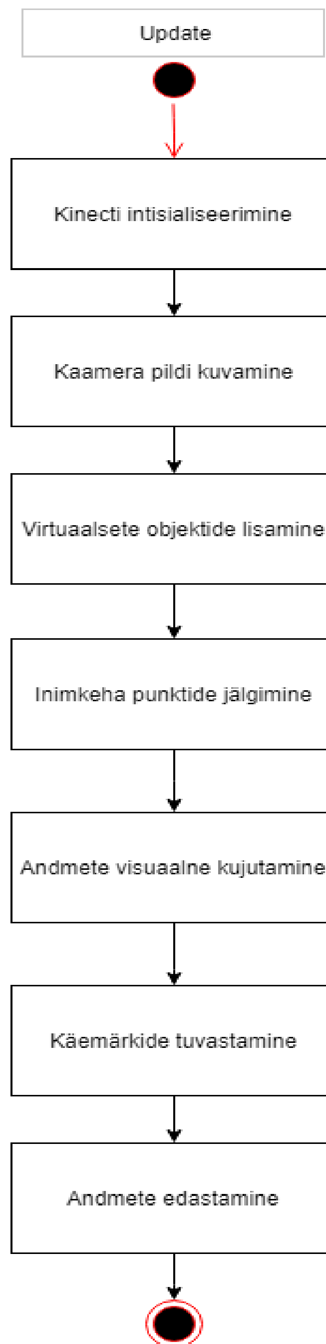
Pärast seda sai alustatud tööd C++-iga. Luues Visual Studios projekti sellega töötamiseks. Kuid peagi jäi see poolikuks, kuna andmete kätte saamisel oli need vaja Unity keskkonda edastada. Uurides võimalusi, kuidas peaks kahe programmi vaheline UDP ühendus toimima, ei leidnud kasuliku materjali selle teostamiseks. Uurides täpsemalt Unity võimaluste kohta, sai leitud lisasi Unity`le, mida saab kasutada Kinectiga töötamiseks, kus kasutusel on C# programmeerimiskeel.

3.2 Unity lisade testimine

Unity`le on erinevate autorite poolt lisasi Kinectiga töötamiseks ja kaht neist sai ka proovitud, nende võrdlus on näha tabelis 1. Esimesena sai alustatud tööd *Nuitrack Skeleton Tracking* lisaga, kuid ilmnemisel, et osade funktsionaalsuste töötamiseks on vaja alla laadida tasulised komponendi, sai kasutusele võetud *Kinect with MS-SDK* lisa, mis on täies mahus tasuta ja kasutajate hinnang oli kõrge.

3.3 Programmi ülesehitus

Unity skripti *start* funktsioonis, mainitud punktis 3.2, koosneb mänguobjektide ja kehal paikenvate punktide massiivide loomisega, mille kaudu on võimalik teostada punktide koordinaatide arvutamine ühes tsüklis. *Update* funktsioon sisaldab endas programmi põhiosa, kus Kinecti poolt tulevaid töödeldakse. Funktsiooni struktuur on näha joonisel 4.



Joonis 5. Update funktsiooni ülesehitus

3.4 Kinecti initsialiseerimine

Kinecti initsialiseeritakse läbi *KinectManageri* skripti, mida saab seadistada vastavalt skriptis olevatele võimalustele. Antud projektis kasutati järgmisi sätteid:

- Compute User Map – arvutab värvipildi tekstuuri, mida kuvada
- Compute User Map – arvutab inimese kehal jälgitavad punktid
- Player 1 Gestures – saab määrata, kui palju käemärke jälgitakse ja millised need on

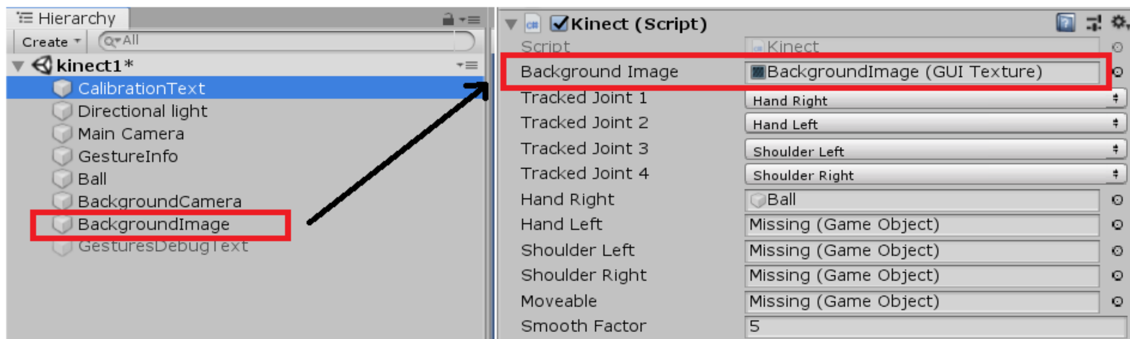
Lisaks on võimalik määrata seadme kõrgust ja nurka, vastaval seadme asukohale või vajadusele, seda saab seadistada Unity keskkonnas ilma programmi muutmata. *Kinectmanageri* skriptis olev meetod väljastab vastava väärtuse, kas Kinectil õnnestus initsialiseerida või mitte.

3.5 Kaamera pildi peale virtuaalsete objektide lisamine

Virtuaalsete objektide kuvamiseks sai uuritud *OverlayDemo* näidet. Näites loodi Unity´s, rakenduse juurde tekstuuri objekt, mis ei omanud väärtust. Programmi koodis kontrollitakse, kas tekstuuri objekt on seotud koodiga, tingimuse täitumisel kuvatakse pilt Unity keskkonda. Unity´s on vaja tekstuuri siduda skriptiga (joonis 6), millele koodi käivitamisel omistatakse kaamerast tulev väärtus.

```
KinectManager manager = KinectManager.Instance;
// kontrollib kas kinect sees
if (manager.IsInitialized())
{
    if (backgroundImage && (backgroundImage.texture == null))
    {
        backgroundImage.texture = manager.GetUsersClrTex();
    }
}
```

Ülal toodud koodi lõik loob viite *KinectManageri* skriptile, läbi selle kutsutakse välja meetod, kus kontrollitakse, kas Kinecti käivitamisel esines probleeme. Pärast seda kontrollitakse, kas tekstuuri objekt on loodud, mille järel omistatakse Kinecti kaamerast tuleva tekstuuri väärtus objektile.



Joonis 6. Tekstuuri sidumine skripti failiga

Mänguobjektide lisamiseks on vaja Unity´s luua soovitava tekstuuriga objekt, *OverlayDemos* kasutati selleks sfääri. Sfääri asukohta saab Unity´s siduda keha koordinaatidega, andmete visuaalseks kujutamiseks.

```
public GameObject HandRight;

Vector3 vPosOverlay = Camera.main.ViewportToWorldPoint(new
Vector3(scaleX, scaleY, scaleZ));

HandRight.transform.position =
Vector3.Lerp(HandRight.transform.position, vPosOverlay, smoothFactor);
```

Objekti tekstuur seotakse skriptis loodud *GameObject* objektiga, mille asukohta muudetakse *Camera* objektiga suhteliselt, mis arvutatakse *vPosOverlay* muutuja kaudu välja. Pärast liigutatakse objekti asukoht uutele koordinaatidele.

3.6 Inimkehal punktide jälgimine ja nende visualiseerimine

Punktide jälgimiseks oli vaja uurida *AvatarsDemo* näidet, kus kasutati Kinecti virtuaalsete avataride kontrollimiseks.

```
Vector3 posJoint = manager.GetRawSkeletonJointPos(userId, joint);
```

Näites kasutati koordinaatide saamiseks ülal toodud meetodit. Sama meetodit rakendades ei töötanud programm nagu soovitud (joonis 7). Punktide visualiseerimisel olid need nihkes keha reaalse asukohaga. Kätte saadud andmetega on vaja teha veel teisendusi läbi, et sfäärid oleks õigesti paigutatud kaamerast tuleval pildil. Meetodi sisenditeks on kasutaja indeks ja jälgitava punkti indeks.

Joonis 7. Punktide paigutus pildil kuvatuna

Põhjus seisnes sellest, et meetod, mida kasutati punktide koordinaatide saamiseks ei arvestanud Kinecti nurka ega asukohta, täpse paigutuse saamiseks oli vaja läbi teha veel andmeteisendusi koordinaatidega.

```
Vector3 posDepth = manager.GetDepthMapPosForJointPos(posJoint);
Vector3 posColor = manager.GetColorMapPosForDepthPos(posDepth);
```

Tuleb Kinectist kätte saada sügavuskaamera kaudu jälgitavate punktide asukohad, mis salvestatakse *posDepth* muutujasse, millest tuleb omakorda arvutada, kus punktid värvikaamera pildil asuvad. Saadud koordinaadi tuleb jagada läbi kaamera resolutsiooniga, et punktid paikneksid pildi peal nagu paistab (joonis 8). Punktide jälgimiseks luuakse objektid viitega jälgitavale punktile. Igas tsüklis kontrollitakse, kas punkt on nähtav, kui tingimus täidetud, arvutatakse välja punkti asukoht.

Joonis 8. Kinectist tulevate koordinaatide visualiseerimine

3.7 Käemärkide tuvastamine

Kätemärkide tuvastamiseks kasutati *KinectManageri* skriptis paiknenud meetodid, mis võimaldas tuvastada, kas kasutaja on teinud jälgitavat liigutust või käemärki, lisas on kokku defineeritud 20 liigutust, millest lõpuks sai kaht kasutatud: käe rusikas hoidmine ja lehvitamine.

Proovitud sai liigutuste tuvastust teha ka läbi *KinectGestures* skripti, kuid proovides tuvastada kasutaja käe rusikasse panekut, tekkis Unity´s mälulekke, mille põhjust otseselt ei leidnud ning sai otsustatud kasutada *KinectManageri* meetodit käemärkide tuvastamiseks.

```
manager.IsGestureComplete(userId, KinectGestures.Gestures.Click, true)
```

Ülaltoodud meetod kutsutakse välja, mis tuvastab käemärgi kasutuse. Sisenditeks on vaja kasutaja indeksit, mis käemärki tuvastatakse ja mis tingimusel on see tuvastatud, kui kasutusel oleks *true* väärtuse asemel *false* oleks käemärk koguaeg tuvastatud programmis.

3.8 Lahenduste integreerimine

Mängu ja Kinect´i ühendamiseks oli vaja leida meetod, kuidas edastada punkti koordinaadid mängu skripti, et õige käe koordinaatide arvutamise ajal edastataks vastava käe koordinaadid. Punktidena jälgiti vasakut ja paremat kätt. Koodi loodi indeks, mis määrab ära kumma käe koordinaate parasjagu edastatakse.

Käemärkide tuvastamise edastamiseks oli vaja luua muutuja, mis registreerib käemärgi tuvastamise. Kinect tuvastab rakenduses kahte liigutust, üheks on käsi rusikas hoidmine virtuaalse objekti haaramiseks (joonis 9) ja teine on lehvitamine objektist lahti laskmiseks (joonis 10). Algselt oli plaanis kasutada stopp käemärki selle jaoks, kuid Kinect ei tuvastanud testimisel seda, seetõttu sai see välja vahetatud lehvitamise vastu.

Joonis 9. Objektist kinni haaramine

Joonis 10. Objektist lahti laskmine

3.9 Esinenud probleemid

Projekti raames esines 3 suuremat probleemi, mis kõige rohkem aega võtsid. Probleemide lahendamiseks ei olnud palju materjali, mida kasutada, abi sai Unity lisa foorumist või tuli läheneda katse eksitus meetodit.

Komponentide teostamisel tekkis probleem käte liigutuste tuvastusega, iseenesest oli kood selle teostamiseks lihtne, programmis kutsuti välja funktsioon, mis väljastas, kas liigutust on sooritatud kasutaja poolt või mitte, *KinectManager* skriptis sättedes tuleb lisaks määrata ära, mis liigutusi tuvastama hakatakse, kuid selle põhjuse leidmine võttis aega.

Vahendite leidmine oli keeruline, kuna varasemalt pole kokkupuudet taolise ideed teostusega. Sobivad vahendid sai leitud katse eksitus meetodil, sai aega kulutatud vahendite õppimisele, mida projektist lõpuks ei kasutatud.

Probleemiks osutus ka esialgse lahenduse keerulisus (lisa 1), mis tegi edaspidi arenduse raskemaks, alles pärast koodi optimeerimist, kui sai asukohtade ja sfääride paiknemise arvutamine pandud ühte tsüklisse, sai kood arusaadavamaks, mis võimaldas ka käemärkide tuvastust lihtsamini teostada (lisa 2).

4 Tulemuste ja edasiarenduse võimalused

4.1 Tulemused

Esialgne plaan võrreldes oli ambitsioonikas, luua liitreaalsuse rakendus, mis jätaks kasutajale elamuse. Komponenditest sai valmis inimese tuvastamine, käemärkide tõlgendamine, kaamera pildi kuvamine ja nende andmete edastamine mängu skripti.

Tehtud lahendus ei ole päris see, mis alguses plaanitud sai, kuigi objektid skaleeruvad vastavalt kaugusele Kinectist, ei jäta see kvaliteetset muljet, mis on vajalik kui seda lahendus avalikus kohas kasutada. Kaamera resolutsioon peaks kõrgem olema, kasutatud Kinectil oli see 640x480. Selle lahendamiseks saaks võtta kasutusele Kinect versioon 2.

Inimkeha jälgimisel võivad jälgitavatele punktidele määratud sfäärid hüppama hakata, kui antud punkt ei ole enam eristatav. Selle parandamiseks on võimalik rakendada meetodid, kui punkt ei ole leitav Kinecti jaoks, lülitatakse vastav objekt Unity keskkonnast välja.

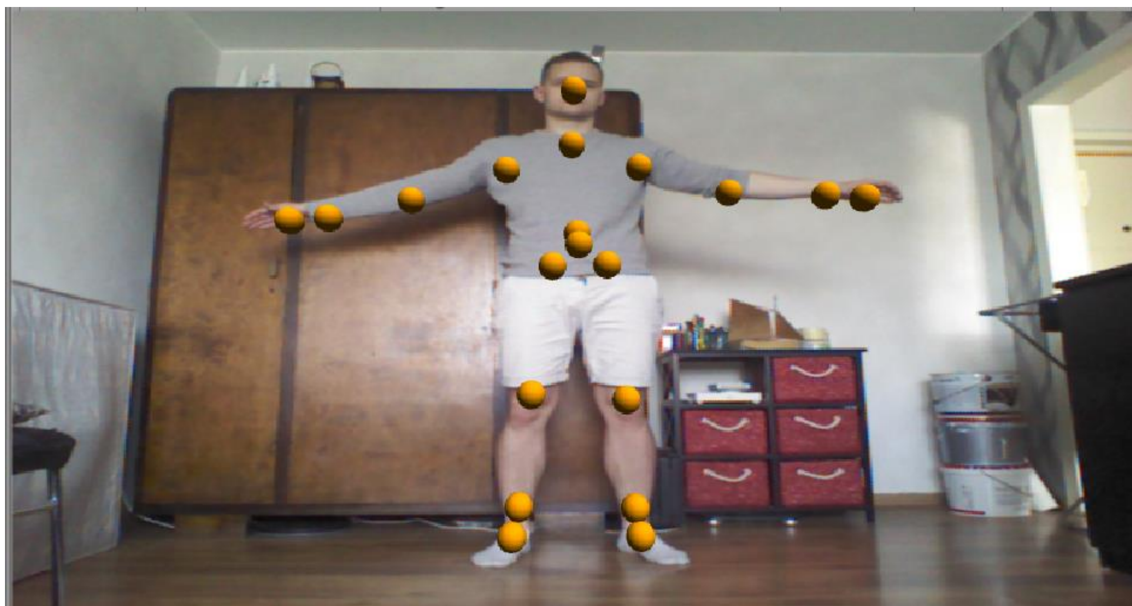
Käemärke tuvastatakse erineva aja intervallidega. Mõnikord on vaja vastavat käemärki teha mitu korda, et Kinect selle ära tuvastaks. Tuvastamise aeg on kuni 4 sekundit, enne kui antakse tagasisidet, kas käemärk on sooritatud või mitte.

4.2 Edasiarenduse võimalused

Kaamera resolutsioon on väike Kinecti esimese verisoonil, et kuvamaks innovaatilist lahendust Mektory puu jaoks, kui kasutada Kinecti andurit selle jaoks. Kasutada tuleks Kinect versioon 2, mis toetab resolutsiooni 1920x1080, mis teeb pildi kvaliteedi paremaks ja võimaldab jälgida kokku 26 punkti inimese kehal võrreldes Kinect verisoon 1ga, mis suudab jälgida 20 punkti (joonis 11). Arenduse teeks lihtsamaks see, et Kinect versioon 2le on olemas sama autori poolt sarnane lisa Unity`le Kinectiga töötamiseks.

Tuleks leida võimalusi, kuidas kasutada rohkem punkte inimkeha jälgimisel mängus sees. Näiteks luua esemed, mida saab kinnitada kasutaja külge. Jälgitavaid punktide arvu on koodis lihtne lisada (joonis 20), kuigi nende edastamiseks peab välja mõtlema uue loogika juurde.

Lisaks uurida, kas Kinectiga on võimalik luua kolmemõõtmeline mudel ruumist, mis võimaldaks luua parema liitreaalsuse kogemuse.



Joonis 11. 20 punkti paigutus kehal.

5 Analüüs

Kinectiga töötamisel aeglustas töökäiku teadmatus, millega seda teha ja millest alustada. Enamus projektist sai teostatud katse eksitus meetodil, mis tähendab palju aega sai kulutatud leidmaks sobivaid vahendeid, mille abil saaks ideed teostada.

Ajakulukaim osa projektis oli inimkeha jälgimis loogikale pihta saamine ning kuidas seda teostada. Kasutatud sai Unity lisas olevaid näiteid uurimiseks, kuidas on seal teostatud inimkeha jälgimine. Lisas on kaks meetodid selle tegemiseks, millest üks leiab inimkehal paiknevate punktide asukohad suhteliselt Kinectiga (joonis 4). Teine võtab töötlemata andmed Kinectil, mille kuvamiseks on vaja läbi viia andmeteisendusi, et punktid paikneksid reaalselt keha peal. Esimene meetod oleks sobinud sel juhul, kui ei oleks kasutanud kaamera pilti taustana, kuna ta muudab inimese kuju väiksemaks (joonis 4) ja andmeteisendused oleksid vajalikud olnud.

Liigutuste tuvastamiseks kasutatud meetod on aeglane ja vahepeal esineb ka tõrkeid. Proovitud sai ka teist lisas olevat meetodid, mis oleks tuvastuse teinud otse *KinectGestures* skripti kaudu, kuid rakendades meetodid esines mäluga probleeme, millele lahendust ei leidnud. Kuid see oleks tõenäoliselt üks võimalus, kuidas protsessi kiirendada. Praegune lahendus kutsub välja meetodi *KinectManageri* skripti kaudu, mis omakorda kutsub meetodi *KinectGestures* skriptist.

Integreerimine toimus kiirest, suuresti tänu sellele, et oli alguses juba paika määratud, mida peaks Kinecti kaudu edastama. Soovitud väärtused ehk käte asukohad, said salvestatud eraldi massiividesse, mida edastatakse kordamööda tsüklis. Mängus endas võib hakata sfäärid, mis näitavad käe asukohta hüppama, kuid selle lahendaks, Unity`s vastava sfääri välja lülitamine, kui punkt ei ole enam jälgitav.

6 Kokkuvõte

Töö käigus sai tutvutud Unity mängumootoriga, kuidas käib rakenduse arendus selles keskkonnas ja mis võimalused sellel on Kinectiga rakenduse loomiseks. Testitud sai Unity lisasi, mis oli loodud Kinecti kasutamiseks.

Kinecti lisadega sai uuritud, kuidas saaks kasutada Kinecti funktsionaalsusi liitreaalsuse rakenduse loomiseks. Rakendati Kinecti kasutaja asukoha määramiseks ja tehtud liigutuste tõlgendamiseks, mis omakorda edastati mängu, kus vastavalt sellele infole loom käitus.

Valmis saadi rakendus, kus Kinectist tulev kaamera pilt kuvatakse mängus, millele on juurde lisatud virtuaalse puu tekstuur, mille peale on modelleeritud loom koos toiduga. Selle haaramisel kasutaja poolt, hakkab loom sellele järgnema. Objektist kinni haaramiseks tuvastab Kinect, kui kasutaja käsi on rusikas ning lahti laskmiseks lehvitab kasutaja kätt.

Projekti raames sai paremalt tutvutud objektorienteeritud programmeerimisega läbi C# keele, arusaamise Kinecti funktsionaalsuste toimimise kohta ja kogemust rakenduse arenduse valdkonnas.

Kasutatud kirjandus

[1] Kinect, Wikipedia, 2019. [Võrgumaterjal]. Available: https://en.wikipedia.org/wiki/Kinect#Kinect_for_Xbox_360_(2010) (18.05.2019)

[2] "Now Kinect 3D Camera can also be used in Bright Sunlight for outdoor Applications", Robot Globe [Võrgumaterjal]. Available: http://robotglobe.org/now-kinect-3d-camera-can-also-be-used-in-bright-sunlight-for-outdoor-applications/ (20.05.2019)

[3] John MacCormick, "How does the Kinect work?", University Of Wisconsin-Madison [Võrgumaterjal]. Available: http://pages.cs.wisc.edu/~ahmad/kinect.pdf (15.05.2019)

[4] Ahmad Faiz Abdull Rashid, Victor Ferrero, " Exploring Applications of Microsoft's Xbox Kinect: Using a Depth Map and Feature Points as a Password" [Võrgumaterjal]. Available: http://pages.cs.wisc.edu/~ahmad/paper.pdf (12.05.2019)

[5] Jamie Shotton, Andrew Fitzgibbon, Mat Cook, Toby Sharp, Mark Finocchio, Richard Moore, Alex Kipman, Andrew Blake, "Real-Time Human Pose Recognition in Parts from Single Depth Images", Microsoft Research Cambridge & Xbox Incubation, [Võrgumaterjal]. Available: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/SupplementaryMaterial-2.pdf (08.05.2019)

[6] J. Brodkin, „How Unity3D Became a Game-Development Beast," 03 06 2013. [Võrgumaterjal]. Available: https://insights.dice.com/2013/06/03/how-unity3dbecome-a-game-development-beast/. (17.05.2019)

[7] RF Solutions, Unity Technologies, 2019. [Võrgumaterjal]. Available: https://assetstore.unity.com/publishers/2787 (20.05.2019)

[8] K. Kuusk, „Interaktiivse virtuaalse puu kontseptsioon ja mängu prototüüp," Tallinn: Tallinna Tehnikaülikool, 2019.

[9] „Unity," Unity Technologies, 2019. [Võrgumaterjal]. Available: https://assetstore.unity.com/packages/tools/kinect-with-ms-sdk-7747. (18.02.2019)

[10] Device driver, Wikipedia, 2019. [Võrgumaterjal]. Available: https://en.wikipedia.org/wiki/Device_driver (18.05.2019)

[11] Dynamis-link library, Wikipedia, 2019. [Võrgumaterjal]. Available: https://en.wikipedia.org/wiki/Dynamic-link_library (18.05.2019)

[12] Wrapper Classes in java, GeeksforGeeks 2019. [Võrgumaterjal]. Available: https://www.geeksforgeeks.org/wrapper-classes-java/ (23.05.2019)

Lisa 1 – [Optimeerimata kood]

```
using UnityEngine;
using System.Collections;

public class vanakood : MonoBehaviour
{
    // deklareeritakse tekstuuri objekt ja jälgitavate punktide
indeksid
    public GUITexture backgroundImage;
    public KinectWrapper.NuiSkeletonPositionIndex TrackedJoint1 =
KinectWrapper.NuiSkeletonPositionIndex.HandRight;
    public KinectWrapper.NuiSkeletonPositionIndex TrackedJoint2 =
KinectWrapper.NuiSkeletonPositionIndex.HandLeft;
    public KinectWrapper.NuiSkeletonPositionIndex TrackedJoint3 =
KinectWrapper.NuiSkeletonPositionIndex.ShoulderLeft;
    public KinectWrapper.NuiSkeletonPositionIndex TrackedJoint4 =
KinectWrapper.NuiSkeletonPositionIndex.ShoulderRight;
    //mänguobjektid sfääride jaoks
    public GameObject HandRight;
    public GameObject HandLeft;
    public GameObject ShoulderLeft;
    public GameObject ShoulderRight;

    public float smoothFactor = 5f;

    public GUIText debugText;

    private float distanceToCamera = 10f;


    void Start()
    {
        if (HandRight)
        {
         //määrab punkti kauguse kaamerast
         distanceToCamera = (OverlayObject.transform.position -
Camera.main.transform.position).magnitude;
        }
    }
```

```
    void Update()
    {
    //pointer to kinectmanager
        KinectManager manager = KinectManager.Instance;

        if (manager && manager.IsInitialized())
        {
            if (backgroundImage && (backgroundImage.texture == null))
            {
                backgroundImage.texture = manager.GetUsersClrTex();
            }

            //integer indeksid, milliseid punkte kehal jälgitakse
            int iJointIndex = (int)TrackedJoint1;
            int jJointIndex = (int)TrackedJoint2;
            int kJointIndex = (int)TrackedJoint3;
            int lJointIndex = (int)TrackedJoint4;

            //kas kasutaja on olemas
            if (manager.IsUserDetected())
            {
                //kasutajale indeksi määramine
                uint userId = manager.GetPlayer1ID();
                //kas jälitav punkt on nähtav
                if (manager.IsJointTracked(userId, iJointIndex))
                {
                // salvestab kasutaja punkti asukoha
                    Vector3 posJoint =
manager.GetRawSkeletonJointPos(userId, iJointIndex);

                    if (posJoint != Vector3.zero)
                    {
                        // depth maps position of the joint
                        Vector2 posDepth =
manager.GetDepthMapPosForJointPos(posJoint);

                        // depth map position to color position
                        Vector2 posColor =
manager.GetColorMapPosForDepthPos(posDepth);

                        float scaleX = (float)posColor.x /
KinectWrapper.Constants.ColorImageWidth;
                        float scaleY = 1.0f - (float)posColor.y /
KinectWrapper.Constants.ColorImageHeight;
                        // checks if the gameobject is true, and
transforms the position of the ball to joint
                        if (HandRight)
                        {
                            Vector3 vPosOverlay =
Camera.main.ViewportToWorldPoint(new Vector3(scaleX, scaleY,
distanceToCamera));
                            HandRight.transform.position =
Vector3.Lerp(HandRight.transform.position, vPosOverlay, smoothFactor);
```

```
                        }
                    }
                }
                if (manager.IsJointTracked(userId, jJointIndex))
                {
                    Vector3 posJoint =
manager.GetRawSkeletonJointPos(userId, jJointIndex);

                    if (posJoint != Vector3.zero)
                    {
                        // väljastab punkti positsiooni sügavus
kaardilt
                        Vector2 posDepth =
manager.GetDepthMapPosForJointPos(posJoint);

                        // väljastab punkti asukoha värvi kaardil
                        Vector2 posColor =
manager.GetColorMapPosForDepthPos(posDepth);
                        // skaleerib x ja y koordinaati vastavalt
resolutsioonile
                        float scaleX = (float)posColor.x /
KinectWrapper.Constants.ColorImageWidth;
                        float scaleY = 1.0f - (float)posColor.y /
KinectWrapper.Constants.ColorImageHeight;


                        if (HandLeft)
                        {
                            Vector3 vPosOverlay =
Camera.main.ViewportToWorldPoint(new Vector3(scaleX, scaleY,
distanceToCamera));
                            HandLeft.transform.position =
Vector3.Lerp(HandLeft.transform.position, vPosOverlay, smoothFactor);
                        }
                    }
                }
                if (manager.IsJointTracked(userId, kJointIndex))
                {
                    Vector3 posJoint =
manager.GetRawSkeletonJointPos(userId, kJointIndex);

                    if (posJoint != Vector3.zero)
                    {
                        Vector2 posDepth =
manager.GetDepthMapPosForJointPos(posJoint);

                        Vector2 posColor =
manager.GetColorMapPosForDepthPos(posDepth);

                        float scaleX = (float)posColor.x /
KinectWrapper.Constants.ColorImageWidth;
                        float scaleY = 1.0f - (float)posColor.y /
KinectWrapper.Constants.ColorImageHeight;
```

```
                        if (ShoulderLeft)
                        {
                            Vector3 vPosOverlay =
Camera.main.ViewportToWorldPoint(new Vector3(scaleX, scaleY,
distanceToCamera));
                            ShoulderLeft.transform.position =
Vector3.Lerp(ShoulderLeft.transform.position, vPosOverlay,
smoothFactor);
                        }
                    }
                }
                if (manager.IsJointTracked(userId, lJointIndex))
                {
                    Vector3 posJoint =
manager.GetRawSkeletonJointPos(userId, lJointIndex);

                    if (posJoint != Vector3.zero)
                    {
                        Vector2 posDepth =
manager.GetDepthMapPosForJointPos(posJoint);

                        Vector2 posColor =
manager.GetColorMapPosForDepthPos(posDepth);

                        float scaleX = (float)posColor.x /
KinectWrapper.Constants.ColorImageWidth;
                        float scaleY = 1.0f - (float)posColor.y /
KinectWrapper.Constants.ColorImageHeight;


                        if (ShoulderRight)
                        {
                            Vector3 vPosOverlay =
Camera.main.ViewportToWorldPoint(new Vector3(scaleX, scaleY,
distanceToCamera));
                            ShoulderRight.transform.position =
Vector3.Lerp(ShoulderRight.transform.position, vPosOverlay,
smoothFactor);
                        }
                    }
                }

            }

        }
    }
}
```

Lisa 2 – [Lõplik kood]

```
using UnityEngine;
using System.Collections;

public class Kinect : MonoBehaviour
{



    // määrab jälgitavad liigesed
    public GUITexture backgroundImage;
    public KinectWrapper.NuiSkeletonPositionIndex TrackedJoint1 =
KinectWrapper.NuiSkeletonPositionIndex.HandRight;
    public KinectWrapper.NuiSkeletonPositionIndex TrackedJoint2 =
KinectWrapper.NuiSkeletonPositionIndex.HandLeft;
    public KinectWrapper.NuiSkeletonPositionIndex TrackedJoint3 =
KinectWrapper.NuiSkeletonPositionIndex.ShoulderLeft;
    public KinectWrapper.NuiSkeletonPositionIndex TrackedJoint4 =
KinectWrapper.NuiSkeletonPositionIndex.ShoulderRight;


    // loob objektid, kuhu saab tekstuuri lisada
    public GameObject HandRight;
    public GameObject HandLeft;
    public GameObject ShoulderLeft;
    public GameObject ShoulderRight;

    public GameObject moveable;

    //private GameObject[] points;
    private GameObject[] hands;
    private GameObject[] shoulders;
    private int[] points;


    private Vector3 posJoint;
    private Vector3 posHand2;
    private Vector3 posHand1;
    private Vector3 posScale;
    private Vector3 moveablePos;

    public float smoothFactor = 5;

    public bool isgrab = false;
    private bool left = false;
    private bool right = false;


    void Start()
    {
        hands = new GameObject[]
        {
            HandRight, HandLeft, //0-3
        };
        shoulders = new GameObject[]
            {
            ShoulderRight, ShoulderLeft, //0-3
        };
```

```
        int iJointIndex = (int)TrackedJoint1;
        int jJointIndex = (int)TrackedJoint2;
        int kJointIndex = (int)TrackedJoint3;
        int lJointIndex = (int)TrackedJoint4;

        points = new int[] { iJointIndex, jJointIndex, kJointIndex, lJointIndex
};

    }

    void Update()
    {
        // loob muutuja, millega kinecti funktsioonidele ligi saada
        KinectManager manager = KinectManager.Instance;
        // kontrollib kas kinect sees
        if (manager.IsInitialized())
        {
            if (backgroundImage && (backgroundImage.texture == null))
            {
                backgroundImage.texture = manager.GetUsersClrTex();
            }
        }

        if (manager.IsUserDetected())
        {
            uint userId = manager.GetPlayer1ID();
            for (int i = 0; i < hands.Length; i++)
            {
                int joint = points[i];
                if (manager.IsJointTracked(userId, joint))
                {

                    Vector3 posJoint = manager.GetRawSkeletonJointPos(userId,
joint);


                    if (posJoint != Vector3.zero)
                    {
                        //liigese asukoha saamine
                        Vector3 posDepth =
manager.GetDepthMapPosForJointPos(posJoint);
                        Vector3 posColor =
manager.GetColorMapPosForDepthPos(posDepth);

                        float scaleX = (float)posColor.x /
KinectWrapper.Constants.ColorImageWidth;
                        float scaleY = 1.0f - (float)posColor.y /
KinectWrapper.Constants.ColorImageHeight;
                        float scaleZ = 5.0f - posJoint.z + 1.0f /
KinectWrapper.Constants.ColorImageHeight;
                        posScale = new Vector3(scaleX, scaleY, scaleZ);


                        if (manager.IsGestureComplete(userId,
KinectGestures.Gestures.Click, true))
                        {
                            isgrab = true;

                        }
                        if (manager.IsGestureComplete(userId,
KinectGestures.Gestures.Wave, true))
                            isgrab = false;
```

```
                        if (i == 0)
                        {
                            posHand1 = posScale;


                        }
                        else if (i==1)
                        {
                            posHand2 = posScale;
                        }

                    }

                }
            }
        }
        //Debug.Log(isgrab);
    }
    public Vector3 Ooops
    {
        get { return posHand2; }
    }
    public Vector3 Ooops2
    {
        get { return posHand1; }
    }
    public bool handIndex
    {
        get { return left; }
    }
    public bool handIndex2
    {
        get { return right; }
    }
    public bool gestureIndex
    {
        get { return isgrab; }
    }
}
```