Τεχνολογικό Εκπαιδευτικό Ίδρυμα Κρήτης

Σχολή Τεχνολογικών Εφαρμογών

Τμήμα Μηχανικών Πληροφορικής

Πτυχιακή εργασία

Τίτλος:

Ανάπτυξη σαρωτή τριών διαστάσεων (3D) με τη χρήση αισθητήρα Kinect.

Γεώργιος Ψιστάκης ΑΜ: 2545

Καθηγητής: Γεώργιος Παπαδουράκης

Επιτροπή Αξιολόγησης:

Ημερομηνία παρουσίασης:

Ευχαριστίες

Η εκπόνηση της παρούσας πτυχιακής εργασίας με θέμα «Ανάπτυξη σαρωτή τριών διαστάσεων (3D) με τη χρήση αισθητήρα Kinect» στο Τμήμα Μηχανικών Πληροφορικής, Τ.Ε.Ι. Ηρακλείου υλοποιήθηκε με την υποστήριξη του καθηγητή Παπαδουράκη Γεώργιου .

Πρώτα από όλα θέλω να εκφράσω τις θερμές μου ευχαριστίες στον καθηγητή μου, Παπαδουράκη Γεώργιο, για την εμπιστοσύνη που μου έδειξε και την ανάθεση της παραπάνω πτυχιακής εργασίας.
Ιδιαίτερα ευχαριστώ για την πολύτιμη βοήθεια του Emiliano Pastorelli και των μελών του Visualization Group του τμήματος Μηχανικών και Εφαρμοσμένων Μαθηματικών, του Ινστιτούτου Κυβερνητικής του Ταλλίν Πανεπιστημίου της Εσθονίας (Institute of Cybernetics,Tallinn University of Technology), που δέχοντας με στην ομάδα τους, μου μεταλαμπάδευσαν τις γνώσεις τους.

Θερμές ευχαριστίες απευθύνω σε όλους τους καθηγητές που είχα όλα τα χρόνια της, μέχρι στιγμής, ακαδημαϊκής μου ζωής για τις γνώσεις που μου μμετέδωσαν και για τις ευκαιρίες που μου έδωσαν.
Τέλος, θέλω να ευχαριστήσω την οικογένειά μου και κυρίως τους γονείς μου, Αριστομένη και Χαρίκλεια Ψιστάκη, που με στήριζαν ανελλιπώς όλα αυτά τα χρόνια, δίνοντάς μου κουράγιο να προχωρώ και να προοδεύω. Επίσης, ξεχωριστά ευχαριστώ την Κα. Ευαγγελία Μυλωνά για την βοήθειά της στην επιμέλεια της πτυχιακή μου εργασίας, λοιπούς φίλους και συναδέλφους μου, που με την καθημερινή τους συμπαράσταση, την υπομονή τους και την θετική τους σκέψη συνέβαλαν στην εκπλήρωση του στόχου μου

Ψιστάκης Γεώργιος

Σεπτέμβριος 2014

Abstract

In this thesis, we created an application that uses data from 3 Dimensions (3D) sensor's depth camera. This application will be using those depth data to extract a point-cloud from each side of a rotated object. Those point-clouds are going be compared, aligned and concatenated, in order to have a complete 3D scan. To avoid the possibility of noise those concatenated point-clouds will be smoothen. Using the smoothen result it will create a mesh of its surface to complete the 3D scan and be ready for use in 3D printers or 3D environments. The main goal is to complete an inexpensive 3D scanner using open-source development tools and affordable devices. To archive this goal we will use C++ and libraries such as PCL (**Point Clouds Library**) ,**OpenNI** Kinect sensor.

Σύνοψη

Σε αυτή την πτυχιακή, θα δημιουργήσουμε μια εφαρμογή που διαχειρίζεται τα δεδομένα που εξάγονται από την κάμερα βάθους, αισθητήρων τριών διαστάσεων (3D). Η εφαρμογή αυτή θα χρησιμοποιεί αυτά τα δεδομένα βάθους για να δημιουργήσει ένα σύννεφο από σημεία (pointcloud) που θα αντιπροσωπεύει την συνολική επιφάνεια ενός αντικείμενου στο τρισδιάστατο χώρο. Για να γίνει αυτό, τα pointclouds θα συγκρίνονται, ευθυγραμμίζονται, ενώνονται και τέλος θα λειαίνεται για να την αναδημιουργία της επιφάνειας του 3D αντικειμένου. Σκοπός αυτής της εργασίας είναι η δημιουργία ενός προσιτού τρισδιάστατου σαρωτή (3D-scanner) χρησιμοποιώντας ελεύθερο λογισμικό όπως PCL (**PointCloudsLibrary**) , **OpenNI** και αισθητήρες Kinect.

Περιεχόμενα

ΕΥΧΑΡΙΣΤΙΕΣ....2
ABSTRACT....3
ΣΥΝΟΨΗ....4
ΚΑΤΑΛΟΓΟΣ ΕΙΚΟΝΩΝ....7
1. ΕΙΣΑΓΩΓΗ....9
1.1 ΚΙΝΗΤΡΟ ΓΙΑ ΤΗΝ ΔΙΕΞΑΓΩΓΗ ΤΗΣ ΕΡΓΑΣΙΑΣ....12
1.2 ΣΚΟΠΟΣ ΚΑΙ ΣΤΟΧΟΣ ΑΥΤΗΣ ΤΗΣ ΕΡΓΑΣΙΑΣ....13
1.3 ΔΟΜΗ ΤΗΣ ΕΡΓΑΣΙΑΣ....13
2. ΕΡΓΑΛΕΙΑ....14
2.1 MICROSOFT-KINECT....14
2.1.1 ΑΙΣΘΗΤΗΡΑΣ ΒΑΘΟΥΣ (DEPTHSENSOR) ΚΑΙ ΠΟΜΠΟΣ ΥΠΕΡΥΘΡΩΝ (IR-LIGHT)....14
2.1.2 COLOR STREAM....16
2.1.3 AUDIO-STREAM....17
2.1.4 ΜΙΚΡΟΦΩΝΑ....18
2.1.5 ΕΠΙΤΑΧΥΝΣΙΟΜΕΤΡΟ (ACCELEROMETER)....18
2.1.6 ΜΗΧΑΝΟΚΙΝΗΤΗ ΒΑΣΗ (MOTORIZED TILT)....18
2.1.7 ΠΡΟΑΠΑΙΤΟΥΜΕΝΑ....20
2.1.8 ΕΦΑΡΜΟΓΕΣ ΤΟΥ ΑΙΣΘΗΤΗΡΑ KINECT....20
2.1.10 ΕΙΚΟΝΙΚΗ ΠΡΑΓΜΑΤΙΚΟΤΗΤΑ (VIRTUAL REALITY)....21
2.1.11 ΕΠΑΥΞΗΜΕΝΗΣ ΠΡΑΓΜΑΤΙΚΟΤΗΤΑΣ (AUGMENTED REALITY)....22
2.1.12 Α.Μ.Ε.Α. 23
2.2 ASUS XTION....25
2.3 BLENDER....25
2.4 PARAVIEW....27
2.5 ΛΕΙΤΟΥΡΓΙΚΟ ΣΥΣΤΗΜΑ (Λ.Σ.)....28
2.6 ΒΑΣΗ ΚΙΝΗΣΗΣ 360°....29
3 ΤΕΧΝΟΛΟΓΙΕΣ ΥΛΟΠΟΙΗΣΗΣ....30
3.1 OPENNI (OPEN NATURAL INTERACTION)....30
3.2 OPENKINECT – LIBFREENECT....32
3.3 POINT-CLOUD LIBRARY (PCL)....32
3.3.1 ΑΛΓΟΡΙΘΜΟΙ....33
3.3.2 ΕΦΑΡΜΟΓΕΣ ΒΑΣΙΖΟΜΕΝΕΣ ΣΤΗΝ PCL LIBRARY....42
4 ΥΛΟΠΟΙΗΣΗ ΕΡΓΑΣΙΑΣ....46
4.1 ΕΓΚΑΤΑΣΤΑΣΗ PCL....46
4.2 ΈΡΕΥΝΑ ΚΑΙ ΠΕΙΡΑΜΑΤΑ....47
4.2.1 OPENNI 3D-IMAGE RECEIVER....47
4.2.2 ΕΓΚΑΤΑΣΤΑΣΗ OPENNI....47
4.2.3 ΕΠΕΞΗΓΗΣΗ ΚΩΔΙΚΑ....48
4.3 ΧΡΗΣΗ....52
4.3.1 ΔΥΟ ΑΙΣΘΗΤΗΡΕΣ....53
4.3.2 ΤΡΕΙΣ ΑΙΣΘΗΤΗΡΕΣ....55
4.4 ΣΤΑΘΕΡΟΣ ΑΙΣΘΗΤΗΡΑΣ....57
5 ΕΦΑΡΜΟΓΗ....59
5.1 ΜΕΡΟΣ ΠΡΩΤΟ- OPENNI GRABBER....59
5.1.1 ΠΑΡΑΘΥΡΟ....59
5.1.2 ΦΙΛΤΡΑΡΙΣΜΑ ΚΑΙ ΑΠΟΘΗΚΕΥΣΗ....60
5.2 ΚΥΡΙΑ ΕΦΑΡΜΟΓΗ....62
5.2.1 COMPUTETRANSFORMATION....62
5.2.2 MAIN....66
5.3 ΑΝΑΔΗΜΙΟΥΡΓΙΑ ΕΠΙΦΑΝΕΙΑΣ....69

6 ΕΦΑΡΜΟΓΕΣ 73
6.1 KYB3 - VIRTUAL ENVIRONMENT (CAVE SYSTEM) 73
6.2 ΠΟΛΙΤΙΣΤΙΚΗ ΚΛΗΡΟΝΟΜΙΑ 75
6.3 ΥΓΕΙΑ 76
6.4 ΨΥΧΑΓΩΓΙΑ 77
7 ΔΗΜΟΣΙΕΥΣΕΙΣ 79
8 ΒΙΒΛΙΟΓΡΑΦΙΑ 80

Κατάλογος Εικόνων

Εικόνα 1 - Μέθοδοι ανίχνευσης 10
Εικόνα2 - Head Scanner by Cybeware Laboratories 11
Εικόνα3 - Full Body Scanner byCybeware Laboratories 11
Εικόνα 4–Point-Cloud 12
Εικόνα5 - Polygon, Surface, Solid CAD models 12
Εικόνα 6 – Kinect-XBOX 360 14
Εικόνα 7 – Depth camera 15
Εικόνα 8 – RGB camera 17
Εικόνα 9 – Μοίρες κίνησης μηχανοκίνητης βάσης Kinect 19
Εικόνα 10 – Ρομποτικό σύστημα με χρήση Kinect 21
Εικόνα 11 – V.R. με χρήση Kinect και Oculus Rift 22
Εικόνα 12- Παιχνίδι A.R. 23
Εικόνα 13 – Πεζοπορία στην A.R. 23
Εικόνα 14 – Μεταφραστής νοηματικής γλώσσας 24
Εικόνα 15 – Αποτέλεσμα του 3D-scannerστο Blender 26
Εικόνα 16 – ParaView 27
Εικόνα 17 – Ubuntu 12.04 28
Εικόνα 18 - Περιστρεφόμενη βάση με το βασικό αντικείμενο πειραματισμού (Σουρικάτες) 30
Εικόνα 19 – NITE Skeleton Tracking 31
Εικόνα 20 - Δομή της PCL 33
Εικόνα 21 – pass through (πριν και μετά) 33
Εικόνα 22 – Φίλτρο MLS 35
Εικόνα 23 – Voxe lGrid 36
Εικόνα 24 – ICP: Συνδέοντας δύο point-clouds 36
Εικόνα 25 - Key-Points (από σάρωση σουρικάτων) 38
Εικόνα 26 – Normalsαπό φλιτζάνι καφέ 39
Εικόνα 27 – Αντιστοιχίες (correspondences) 40
Εικόνα 28 – Poisson-Mesh (ανακατασκευή επιφάνειας) 42
Εικόνα 29 - Παράδειγμα KinectFusion (Kinfu) 43
Εικόνα30 - Robotic Operating System (ROS) 43
Εικόνα 31 – Χαρτογράφηση κλειστού χώρου από τα ROS 44
Εικόνα 32 – Αντικείμενο για σάρωση με το In-Hand scanner 45
Εικόνα33 – Διεπαφή In-Hand scanner 45
Εικόνα 34 - Αποτέλεσμα In-Hand scanner 46
Εικόνα 35 – Αποτέλεσμα OpenNIimagereceiver 51
Εικόνα 36 – Αποτέλεσμα OpenNI image receiver 52
Εικόνα 37 – (a) Πείραμα με δύο Kinect 53
Εικόνα 38 – Παράδειγμα πειράματος (a) 54
Εικόνα 39–(b) Πειραμα με δύο Kinect 54
Εικόνα 40 - Αποτέλεσμα πειράματος (a) 55
Εικόνα 41 - Αποτέλεσμα πειράματος (b) 55
Εικόνα 42 - Θέσεις τριών Kinect 56
Εικόνα 43 - Αποτέλεσμα πειράματος τριών Kinect 57
Εικόνα 44 - Αποτέλεσμα πειράματος τριών Kinect 57
Εικόνα 45 - Πείραμα με περιστρεφόμενη Βάση (πολλαπλά στιγμιότυπα) 58

Εικόνα 46 – Παράθυρο διαλόγου OpenNIGrabber 60
Εικόνα 47 - Τελικό αποτέλεσμα (δεξιά)..72
Εικόνα 48–Το Kyb3 Cave System σε χρήση 74
Εικόνα49 - VR-Space Wintracker, Wiimote controller, Polarized glasses 75
Εικόνα 50 -3DΠαράδειγμα ατσάλινων ινών..75
Εικόνα 51 - Δαβίδ του Μιχαήλ Άγγελου από την Cyberware (αριστερά το πρωτότυπο)..................76
Εικόνα 52 - CAD/CAMστην οδοντιατρική 77
Εικόνα 53 – CAD/CAM Οσταιοπλαστικής...77
Εικόνα 54–VideoCinematography...77
Εικόνα 55 - Fast Avatar Capture..78
Εικόνα 56 - Παράδειγμα διαδικασίας σάρωσης φωτογραμμετρικής εφαρμογής VisualSFM 79

1. Εισαγωγή

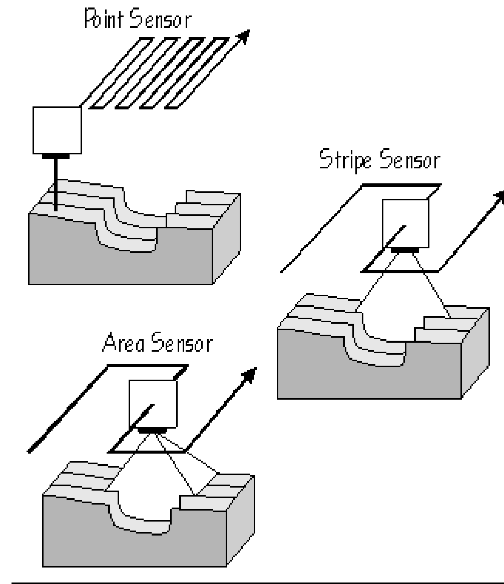
Η προσπάθεια σύλληψης και απεικόνισης της πραγματικότητας, ήταν ανέκαθεν ένα θέμα που απασχολούσε τον άνθρωπο, όπως φαίνεται από τα πρώτα χρόνια της ιστορίας του. Με την πάροδο του χρόνου, καλλιτέχνες και τεχνίτες, σταδιακά έβρισκαν τρόπους, όλο και πιο ακριβείς, για τη δημιουργία των σχεδίων τους, προσπαθώντας να επιτύχουν μια πιο ρεαλιστική προσέγγιση για την απεικόνιση της πραγματικότητας.

Με την έλευση των ηλεκτρονικών υπολογιστών καταστείτε δυνατή η κατασκευή σύνθετων μοντέλων. Την δεκαετία του '80, η βιομηχανία κατασκευής εργαλείων αναπτύσσει έναν αισθητήρα επαφής. Ήταν η απαρχή για το εύρος των δυνατοτήτων που θα ακολουθήσει, ως προς την εξέλιξη τρισδιάστατων (3Dimensional - 3D) μοντέλων. Ωστόσο, η ταχύτητα του μοντέλου ήταν τόσο αργή που δεν μπορούσε να είναι χρήσιμη σε αυτή τη μορφή.

Μετέπειτα, η οπτική τεχνολογία αναπτύσσεται και εξελίσσεται με την βοήθεια ειδικών, οι οποίοι χρησιμοποιούν φως έναντι των, μέχρι τότε, φυσικών ανιχνευτών. Με το φώς, η δυνατότητα σάρωσης μαλακών αντικειμένων και επιφανειών, ήταν πολύ πιο γρήγορη, ανοίγοντας το δρόμο για την σάρωση και ευαίσθητων - στην φυσική επαφή – αντικειμένων, πράγμα το οποίο δεν προσέφεραν οι φυσικοί αισθητήρες.

Αναπτύχθηκαν τρείς τρόποι οπτικής τεχνολογίας:

- <u>Σημειακοί(Point)</u>:
  Είχαν παρόμοια λογική με την λειτουργία των φυσικών ανιχνευτών στο γεγονός ότι χρησιμοποιούσαν ένα μοναδικό σημείο αναφοράς, το οποίο επαναλαμβανόταν πολλές φορές. Ήταν μια αργή προσέγγιση, δεδομένου ότι εμπλεκόταν αρκετή φυσική κίνηση από τον αισθητήρα, όπου εκείνη την εποχή αποτελούσε σημαντικό πρόβλημα.
- <u>Περιοχής(Area):</u>
  Ήταν τεχνικά δύσκολη λόγο της έλλειψης ισχυρών συστημάτων πολωμένης περιοχής.
- <u>Λωρίδων(Stripe):</u>
  Ήταν η ταχύτερη μέθοδος ανίχνευσης καθώς χρησιμοποιούσε ένα σύνολο από πολλά σημεία πάνω από το αντικείμενο. Προσέφερε υψηλή ακρίβεια σε σχέση με τα παραπάνω και μαζί με την ταχύτητα αποτέλεσε την καλύτερη μέθοδο εκ των τριών.



Εικόνα 1 - Μέθοδοι ανίχνευσης

Η μέθοδος των λωρίδων έφερε την εξέλιξη. Ωστόσο, γρήγορα έγινε αντιληπτό ότι η πρόκληση έγκειτο στην ανάπτυξη ενός λογισμικού που θα την υλοποιούσε κατάλληλα. Για να συλληφθεί ένα αντικείμενο στις τρείς διαστάσεις, ο αισθητήρας θα πρέπει να κάνει αρκετές σαρώσεις από διαφορετικές θέσεις. Η πρόκληση βρισκόταν στον τρόπο ένταξης όλων αυτών των σαρώσεων συγκεντρωτικά, να αφαιρεθούν τα διπλά στοιχεία και να φιλτραριστούν με τέτοιο τρόπο ώστε διαγραφούν τα πλεονασματικά σημεία δεδομένων. Μια διαδικασία που θα επαναλαμβανόταν πολλές φορές για μόνο ένα αντικείμενο και απαιτούνταν να γίνει γρήγορα.

Οι μοντελιστές 3D (3D-modelers) έψαχναν έναν σαρωτή με τα ακόλουθα χαρακτηριστικά:

- Ακρίβεια
- Ταχύτητα
- Πραγματικά τρισδιάστατο (Real 3D)
- Ικανό να αιχμαλωτίσει χρώμα
- Οικονομικά προσιτό

Μια από τις πρώτες εφαρμογές ήταν η σάρωση ανθρώπων από την βιομηχανία κινουμένων σχεδίων. Η εταιρεία Cybeware Laboratories από το Λος Άντζελες ανέπτυξε αυτό τον τομέα, την δεκαετία του ’80 με ένα σαρωτή κεφαλής (Head Scanner) και αργότερα, την δεκαετία του ’90 ένα ολοκληρωμένο σαρωτή σώματος (Full Body Scanner).

Εικόνα2 - Head Scanner by Cybeware Laboratories

Εικόνα3 - Full Body Scanner byCybeware Laboratories

Το 1994, η εταιρία REPLICA ξεκίνησε την παραγωγή τρισδιάστατων σαρωτών (3D-scanners) που επέτρεπαν μια γρήγορη και υψηλής ακριβείας σάρωση σε αντικείμενα με πολλές λεπτομέρειες. Την ίδια εποχή, η Cybeware έφτιαχνε σαρωτές εξίσου ακριβείς με ένα επιπρόσθετο χαρακτηριστικό· την δυνατότητα να αιχμαλωτίζουν το χρώμα των αντικειμένων. Άλυτο, όμως, εξακολουθούσε να μένει το βασικό πρόβλημα, το οποίο ήταν ο χρόνος που χρειάζονταν όλα αυτά για να βγάλουν το αποτέλεσμα.

Το 1996, η λύση επήλθε όταν οι σαρωτές 3D συνδύασαν το μοντέλο λωρίδων τρισδιάστατης σάρωσης με την τεχνολογία ενός χειροκίνητου βραχίονα στην εφαρμογή του ModelMaker [XI]. Αυτό ήταν το πρώτο σύστημα γρήγορης και ευέλικτης Σύλληψης Πραγματικότητας (Reality Capture System) που μπορούσε να παράγει έγχρωμα 3D μοντέλα σε λίγα λεπτά.

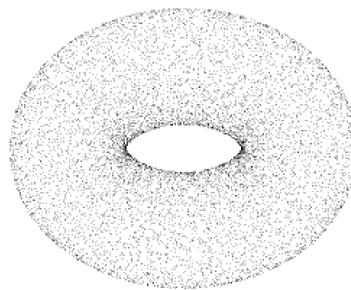
Τα τελευταία χρόνια, η 3D τεχνολογία καθώς και οι 3D σαρωτές έχουν κάνει τεράστια άλματα και η εξέλιξη τους έχει διαρκώς αυξανόμενη πορεία. Πολλές διαφορετικές τεχνικές μπορούν να χρησιμοποιηθούν για την κατασκευή ενός σαρωτή ανάλογα με τις ανάγκες των χρηστών αλλά και την φύση των αντικειμένων· έχοντας τρία είδη σάρωσης:

- **Επαφής (Contact):** Απαιτείται κάποιου είδους επαφής του σαρωτή με το αντικείμενο.
- **Δραστικοί – μη επαφής (Non-Contact active):** Εκπέμπουν κάποιου είδους ακτινοβολία ή φώς στο αντικείμενο και η σάρωση επιτυγχάνεται με την ανίχνευση της αντανάκλασης αυτής της ακτινοβολίας.
- **Παθητικοί – μη επαφής (Non-Contact passive):** Είναι οι σαρωτές που χρησιμοποιούν την ανακλώμενη ακτινοβολία των αντικειμένων από το περιβάλλον (π.χ.: φώς του ήλιου), χωρίς να εκπέμπουν οι ίδιοι φώς ή ακτινοβολία στο αντικείμενο, για να επιτύχουν την σάρωση.

Το Kinect ανήκει στην κατηγορία ·δραστικού – μη επαφής αισθητήρα (Non-Contact active sensor) και αποτελεί την σάρωση που θα χρησιμοποιηθεί σε αυτή την πτυχιακή εργασία.

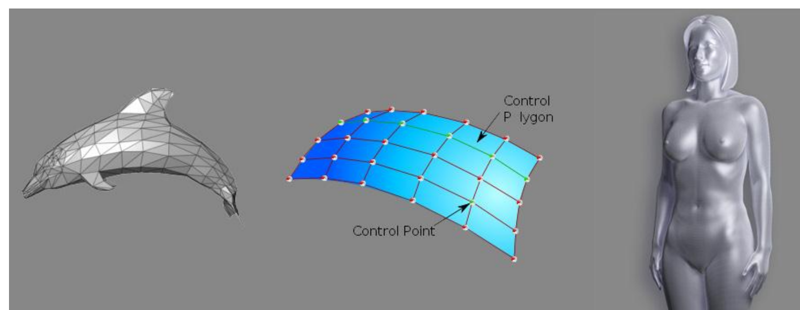
Οι κύριες τεχνικές ανοικοδόμησης (reconstruction) των δεδομένων είναι:

- **Από σύννεφα σημείων (point-clouds):** Τα point-clouds που παράγονται από 3D-scanners και 3D απεικόνιση (3D-imaging) μπορούν να χρησιμοποιηθούν για την μέτρηση και απεικόνιση της αρχιτεκτονικής και της κατασκευής του χώρου ή του αντικειμένου.



Εικόνα 4–Point-Cloud

- Από μοντέλα (models) που χωρίζονται σε τρείς υποκατηγορίες: Polygon-mesh, Surface-mesh [VI], Solid-CAD.



Εικόνα5 - Polygon, Surface, Solid CAD models

Συνδυασμόςpoint-cloud και polygon-mesh θα χρησιμοποιηθεί στην εργασία.

1.1 Κίνητρο για την διεξαγωγή της εργασίας

Το βασικότερο κίνητρο αυτής της εργασίας ήταν η έρευνα πάνω στις δυνατότητες ενός χαμηλού κόστους 3D-scanner. Παράλληλα δουλεύοντας στο Ινστιτούτο Κυβερνητικής του Ταλλίν Πανεπιστημίου της Εσθονίας (Institute of Cybernetics, Tallinn University of

Technology), ο βασικός προσανατολισμός του τμήματος μας ήταν η τελειοποίηση ενός υβριδικού περιβάλλοντος εικονικής πραγματικότητας (Virtual Environment – VE). Τελειώνοντας την κατασκευή του, θεωρήθηκε απαραίτητο εκτός από τις βασικές παρουσιάσεις για επιστημονικούς σκοπούς, να υπάρχει και μια πιο άμεση επαφή των επισκεπτών. Έτσι, δημιουργήθηκε η ιδέα της χρήσης του 3D-scanner με το οποίο θα μπορούσαν οι επισκέπτες να σαρώνονται και το αποτέλεσμα αυτής της σάρωσης να περνάει κατευθείαν στο εικονικό περιβάλλον –VE.

1.2 Σκοπός και στόχος αυτής της εργασίας

Ο σκοπός αυτής της εργασίας ήταν η ανάπτυξη μιας εφαρμογής σάρωσης αντικειμένων και η μετατροπή τους σε μοντέλα τρισδιάστατης, ηλεκτρονικής μορφής. Βασική προτεραιότητα αυτής, ήταν να έχει χαμηλό κόστος, χρησιμοποιώντας ανοιχτό λογισμικό και οικονομικά προσιτές συσκευές.
Στόχος ήταν η εφαρμογή αυτή να κοστίζει μονάχα όσο ο αισθητήρας/ες και μπορεί να εκτελεί την εργασία της σε ένα εύλογο χρονικό διάστημα, ούτως ώστε να μπορεί να γίνει η άμεση χρήση του στο VE.

1.3 Δομή της εργασίας

Στο κεφάλαιο 2 γίνεται αναφορά στα εργαλεία που χρησιμοποιήθηκαν για την περάτωση της εργασίας με εκτενή αναφορά στις λειτουργίες και δυνατότητες του Kinect. Στο κεφάλαιο 3 αναλύονται οι τεχνικές που εφαρμόστηκαν, καθώς επίσης, αναπτύσσονται μερικά από τα βασικότερα κομμάτια της βιβλιοθήκης PCL και εφαρμογών βασιζόμενων σε αυτή. Περνώντας στο κεφάλαιο 4, παρουσιάζεται αναλυτικά την πορεία της έρευνας πριν καταλήξουμε στην τελική μέθοδο που χρησιμοποιήθηκε, συμπεριλαμβανομένων και κάποιων πειραμάτων που έγιναν. Στη συνέχεια, το κεφάλαιο 5 αφορά αποκλειστικά την εφαρμογή του 3D-σαρωτή και γίνεται επεξήγηση των βασικότερων κομματιών του κώδικα. Εν κατακλείδι, στο κεφάλαιο 6 αναφέρουμε χρήσεις που θα μπορούσε να έχει ο σαρωτής μας σε παρόν και μέλλον.

2. Εργαλεία

2.1 Microsoft-Kinect

Την 1 Ιουνίου του 2009, η Microsoft ανακοίνωσε ένα project με όνομα Project Natal το οποίο θα δίνει τη δυνατότητα αναγνώρισης της κίνησης του παίκτη στο Xbox 360. Το όνομα όπως όλα τα κωδικά ονόματα της Microsoft δόθηκε από μια πόλη. Η πόλη Natal της Βραζιλίας επιλέχθηκε προς τιμήν του Βραζιλιάνου διευθυντή της Microsoft AlexKipman, ο οποίος ήταν υπεύθυνος για το project, και καταγόταν από εκεί.

Εικόνα 6 – Kinect-XBOX 360

Ο δεύτερος λόγος για το όνομα αυτό ήταν ότι η λέξη natal σημαίνει και «γέννηση», με το οποίο ήθελαν να συμβολίσουν την γέννηση της καινούριας γενιάς ψυχαγωγίας στο σπίτι. Το όνομα του τελικού προϊόντος ήταν Kinect από τις λέξεις kinetic (κινητικός) και connect (συνδέω). Το Kinect έκανε την εμφάνισή του στην αγορά τον Νοέμβριο του 2010. Ο αισθητήρας βάθους ο οποίος περιέχει σχεδιάστηκε και δημιουργήθηκε από την ισραηλίτικη εταιρία Prime Sense, Ltd. [2]. Δύο μήνες μετά από την κυκλοφορία του, η Microsoft, είχε πουλήσει 8 εκατομμύρια Kinect δίνοντάς της τον τίτλο της ηλεκτρονικής συσκευής με την ταχύτερη πώληση στο βιβλίο Guinness. Μέχρι τον Ιανουάριο του 2012 είχαν πουληθεί 18 εκατομμύρια Kinect.

2.1.1 Αισθητήρας Βάθους (DepthSensor) και Πομπός Υπερύθρων (IR-light)

Η τεχνολογία Ligh Coding™ (ευρεσιτεχνία των Zalevsky, Z, etal.]) επιτρέπει στο Kinect να δημιουργήσει 3D χάρτες βάθους μιας σκηνής, σε πραγματικό χρόνο. Μια δομή από σημεία (σχεδόν) υπέρυθρου φωτός προβάλλεται στον χώρο (βλ. Εικόνα 4) και ένας αισθητήρας εικόνας CMOS (Complementary Metal Oxide Semi-conductor) δέχεται τις ανακλώμενες ακτίνες. Το PS1080 SoC (System on a Chip) – τσιπ φτιαγμένο από την Prime Sense που περιέχει το σύστημα του Kinect, ελέγχει την δομή από σημεία φωτός και έχει την δυνατότητα να επεξεργάζεται τα δεδομένα από τον αισθητήρα CMOS παράγοντας δεδομένα βάθους σε πραγματικό χρόνο. Η μέγιστη ανάλυση της εικόνας βάθους που παράγει το PS1080 είναι 640x480, με συχνότητα 30 στιγμιότυπα (frames) ανά δευτερόλεπτο. Στα 2 μέτρα απόστασης από τον αισθητήρα, έχει τη ακρίβεια 3 χιλιοστών σε ύψος και πλάτος και 1 εκατοστό σε βάθος. Η εμβέλεια ορθής λειτουργίας είναι από 0.8 μέχρι 3.5 μέτρα.

Το depth-data (δεδομένα βάθους) stream παρέχει frames στα οποία το κάθε εικονοστοιχίο (pixel) περιέχει την καρτεσιανή απόσταση (σε χιλιοστά) από την επιφάνεια της κάμερας μέχρι το κοντινότερο αντικείμενο στις συγκεκριμένες x και y συντεταμένες, στο οπτικό πεδίο του αισθητήρα. Υπάρχουν δύο πιθανές αποστάσεις για τα δεδομένα βάθους: η προεπιλεγμένη (default range) και η κοντινή απόσταση (near range), μια από τις οποίες διαλέγουμε κατά την έναρξη του depth-stream. Οι εφαρμογές μπορούν να επεξεργαστούν τα δεδομένα από το depth-stream υποστηρίζοντας διάφορα χαρακτηριστικά, όπως παρακολούθηση των κινήσεων του χρήστη (user tracking) και αναγνώριση των αντικειμένων στη σκηνή ώστε να παραβλέπονται εν ώρα παιχνιδιού.

Κάθε pixel στο depth stream χρησιμοποιεί 13 bits για το βάθος και 3 bits για την αναγνώριση του χρήστη. Η τιμή βάθους 0 υποδεικνύει ότι δεν υπάρχουν δεδομένα βάθους για το συγκεκριμένο σημείο, γιατί το αντικείμενο που βρίσκεται σε αυτή τη θέση είναι είτε πολύ κοντά, είτε πολύ μακριά από τον αισθητήρα. Όταν το Skeleton Tracking είναι απενεργοποιημένο, τα 3 bits, τα οποία χρησιμοποιούνται για την αναγνώριση του χρήστη, παίρνουν την τιμή 0.

Εικόνα 7 – Depth camera

2.1.2 Color Stream

Το Kinect έχει επίσης μια ενσωματωμένη κάμερα χρώματος (Color CMOS - VNA38209015) με μέγιστη ανάλυση 1280x1024 για να έχουμε την πραγματική εικόνα πέρα από το χάρτη βάθους. Η συχνότητα λήψης της κάμερας είναι 30 frame per second (fps) και η εικόνα που παράγεται είναι αρκετά καλή, ώστε να χρησιμοποιηθεί σε αλγορίθμους αναγνώρισης προσώπου, δακτύλων ή οτιδήποτε άλλο χρειαζόμαστε στην εφαρμογή μας.

Τα δεδομένα χρώματος είναι διαθέσιμα σε δύο μορφές:

- RGB(Red, Green, Blue)μορφή χρώματος:
  Με αυτή την επιλογή το Kinect παρέχει, ένα 32-bit, γραμμικό X8R8G8B8- μορφοποιημένο bitmap χρώματος, σε sRGB πεδίο χρώματος. Όπου X 8 bits που δε χρησιμοποιούνται, R 8 bits για το χρώμα κόκκινο, G 8 bits για το πράσινο και B 8 bits για το μπλε.
- Y΄UV (ή YUV) μορφή χρώματος:
  Με την επιλογή αυτή έχουμε ένα 16-bit, gamma-corrected γραμμικό UYVY- μορφοποιημένο bitmap, όπου το gamma-correction στο YUV πεδίο είναι ισοδύναμο με το sRGB-gamma στο RGB πεδίο. Επειδή το YUV stream χρησιμοποιεί 16 bits ανά pixel, αυτή η μορφή χρησιμοποιεί λιγότερη μνήμη για την αποθήκευση των δεδομένων του bitmap και δεσμεύει λιγότερη μνήμη στο buffer όταν ανοίγει το colorstream. Τα YUV δεδομένα είναι διαθέσιμα μόνο σε 640x480 ανάλυση και μόνο στα 15 FPS.

Και οι δυο μορφές υπολογίζονται από τα ίδια δεδομένα της κάμερας, έτσι ώστε τα δεδομένα YUV και του RGB αντιπροσωπεύουν την ίδια εικόνα. Διαλέγουμε με πια μορφή θα δουλέψουμε κατά την εκκίνηση του Color Stream.

Εικόνα 8 – RGB camera

2.1.3 Audio-Stream

Το Kinect περιέχει, όπως προαναφέρθηκε μια σειρά από τέσσερα μικρόφωνα, η οποία χρησιμοποιεί 24-bit ADC (Analog-to-digital Converter) και παρέχει τοπική επεξεργασία σήματος, συμπεριλαμβανομένων των: Acoustic Echo Cancellation και Noise Suppression. Οι εφαρμογές που αναπτύσσονται με αυτό το Software Development Kit (SDK) μπορούν να χρησιμοποιήσουν τη σειρά μικροφώνων για υψηλής ποιότητας καταγραφή ήχου, για την εύρεση της ηχητικής πηγής (Source Localization), για την επιλογή λήψης ήχου από μια συγκεκριμένη κατεύθυνση (Beam forming). Με τη χρήση του SDK οι εφαρμογές μας μπορούν επίσης να χρησιμοποιήσουν το Kinect ως συσκευή εισροών (input device) για τη βιβλιοθήκη αναγνώρισης ομιλίας της Microsoft.

2.1.4 Μικρόφωνα

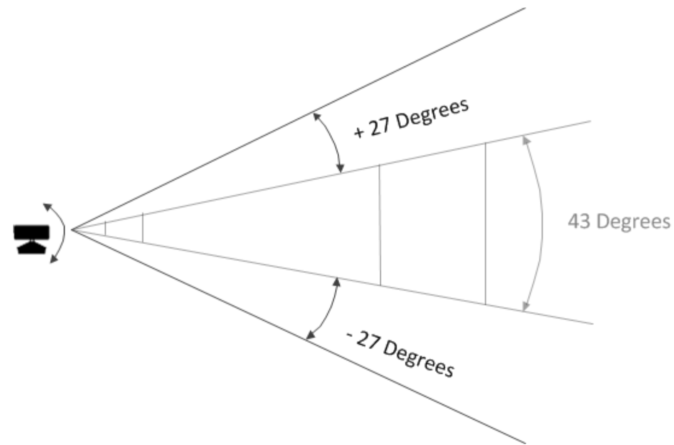
Μια σειρά από τέσσερα μικρόφωνα δίνει στο Kinect τη δυνατότητα όχι απλά να δέχεται ήχο, αλλά και να εντοπίζει την γωνία της πηγής του στη σκηνή. Στην Εικόνα 7 βλέπουμε τη θέση των μικροφώνων μέσα στη συσκευή η οποία επεξεργάζεται ξεχωριστά το καθένα από τα τέσσερα κανάλια τα οποία δέχονται 16-bit ήχο με συχνότητα δειγματοληψίας ίση με 16 kHz.

2.1.5 Επιταχυνσιόμετρο (Accelerometer)

Το Kinect παρέχει ένα επιταχυνσιόμετρο τριών αξόνων (KXSD9-1026, βλ. Εικόνα 8) το οποίο παρέχει την πληροφορία της θέσης της συσκευής. Οι τιμές του X και του Y καθορίζουν την κύλιση και την κλίση, ενώ το Z καθορίζει εάν το Kinect είναι ανάποδα ή όχι. Το επιταχυνσιόμετρο μετρά μονάχα την κλίση, κι όχι τον προσανατολισμό της συσκευής. Το χαρακτηριστικό αυτό του Kinect έχει χρησιμοποιηθεί ευρέως στον τομέα της ρομποτικής.

2.1.6 Μηχανοκίνητη Βάση (Motorized Tilt)

Η συσκευή διαθέτει επίσης μια μηχανοκίνητη βάση (βλ. Εικόνα 9), η οποία ρυθμίζεται δυναμικά μέσω κώδικα. Η κίνηση της βάσης προέρχεται από ένα μικρό μοτέρ στο μέγεθος νομίσματος και τρία εύθραυστα πλαστικά γρανάζια, τα οποία είναι ευαίσθητα στη θερμότητα και ίσως αποτελούν πιο αδύναμο σημείο της συσκευής. Η βάση δίνει στο Kinect τη δυνατότητα να στραφεί προς τα πάνω ή κάτω κατά 27°.



Εικόνα 9 – Μοίρες κίνησης μηχανοκίνητης βάσης Kinect

2.1.7 Προαπαιτούμενα

Από το υλικό:

Για τη χρήση του SDK χρειαζόμαστε τις παρακάτω ελάχιστες απαιτήσεις:

- 32-bit (x86)ή 64-bit (x64) επεξεργαστές (processors)
- Dual-core, 2.66-GHzή (>2,66) επεξεργαστές ή γρηγορότερους
- USB 2.0
- GB της RAM
- Κάρτα γραφικών που υποστηρίζει DirectX 9.0c
- Ένα Microsoft Kinect για Windows Sensor (με τροποποιήσεις και Microsoft Kinect για Xbox360)

2.1.8 Εφαρμογές του αισθητήρα Kinect

Το Kinect, εκτός από την χρήση του στην παιχνιδομηχανή XBOX 360, έγινε αναπόσπαστο κομμάτι κυρίως σε εφαρμογές έρευνας που χρειαζόταν χαμηλού κόστους αισθητήρες. Πλήθος ερευνών το χρησιμοποίησαν με χαρακτηριστικό παράδειγμα τα ρομποτικά συστήματα, Virtual Reality και Augmented Reality εφαρμογές, εφαρμογές για άτομα με ειδικές ανάγκες κ.α.

2.1.9 Ρομποτικά Συστήματα

Ρομποτικά συστήματα όπως τα R.O.S (**Robot Operative System**) χρησιμοποίησαν το Kinect δίνοντας με αυτό τον τρόπο «μάτια» στα ρομπότ, αφού πλέον μπορούσαν να λαμβάνουν εικόνα (κυρίως depth stream). Με την κατάλληλη επεξεργασία των δεδομένων βάθους, τα ρομπότ μπορούσαν να αναγνωρίζουν αντικείμενα αλλά και την θέση τους μέσα στον χώρο.

Εικόνα 10 – Ρομποτικό σύστημα με χρήση Kinect

2.1.10 Εικονική Πραγματικότητα (Virtual Reality)

Η Εικονική Πραγματικότητα χρησιμοποιεί ηλεκτρονικούς υπολογιστές, για να δημιουργήσει και να προσομοιώσει υπαρκτά ή μη περιβάλλοντα, από τα οποία ο χρήστης έχει την ψευδαίσθηση ότι περιβάλλεται και στα οποία μπορεί να κινηθεί ελεύθερα, αλληλεπιδρώντας παράλληλα με τα αντικείμενα που περιλαμβάνουν, όπως θα έκανε και στον πραγματικό κόσμο.

Για την πιο πετυχημένη εμβύθιση ενός χρήστη σε ένα περιβάλλον Εικονικής Πραγματικότητας, απαράμιλλης σημασίας χαρακτηριστικό είναι η απομόνωση του χρήστη και των αισθήσεών του από το πραγματικό κόσμο, επικαλύπτοντας τα ερεθίσματα του πραγματικού κόσμου με αντίστοιχα εικονικά, φτιαγμένα από το σύστημα της Εικονικής Πραγματικότητας. Από τις πέντε (ή μήπως εφτά) αισθήσεις, οι πιο σημαντικές κατά φθίνουσα σειρά είναι η όραση, η ακοή και η αφή.

Συνεπώς, ένα σύστημα Εικονικής Πραγματικότητας να παρέχει στερεοσκοπική εικόνα, δηλαδή δύο εικόνες από διαφορετική οπτική γωνία, μία για κάθε μάτι του χρήστη, έτσι ώστε να δημιουργηθεί η αίσθηση του βάθους στο χώρο. Παράλληλα η ύπαρξη στερεοσκοπικού ήχου βοηθάει το χρήστη να κατανοεί τι γίνεται γύρω του στον εικονικό χώρο που τον περιβάλλει με πολύ φυσικό τρόπο, ενώ ταυτόχρονα αποκλείει τον χρήστη από τους ήχους του πραγματικού κόσμου, οι οποίοι θα μπορούσαν να καταστρέψουν την εικονική του εμπειρία. Τέλος, η αίσθησης της αφής, μπορεί να δημιουργηθεί με κατάλληλες συσκευές είτε για να μπορεί ο χρήστης να νιώθει τον κόσμο, π.χ. να ακουμπά ένα αντικείμενο και να νιώθει αντίσταση, είτε για να καθοδηγήσουμε το χρήστη διευκολύνοντάς τον στην εκτέλεση κάποιων συγκεκριμένων ενεργειών, π.χ. μοντελοποίηση τρισδιάστατων αντικειμένων. Αν όλα τα

παραπάνω συνδυαστούν και με την ανίχνευση των κινήσεων του χρήστη με κατάλληλες συσκευές ανίχνευσης, με τρόπο τέτοιο ώστε το εικονικό περιβάλλον να συμπεριφέρεται όπως και το πραγματικό, τότε η όλη εμπειρία που θα αποκτήσει ο χρήστης μπορεί να είναι άκρως ρεαλιστική.

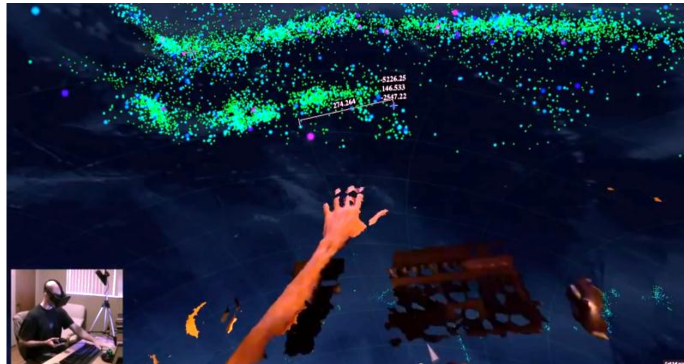
Όπως όλα δείχνουν, η ψυχαγωγία και η διεπαφή των ανθρώπων με του ηλεκτρονικούς υπολογιστές στο προσεχές μέλλον, ανήκει στην εικονική πραγματικότητα (VR). Αυτήν την στιγμή ο τομέας αυτός βρίσκεται σε έξαρση και με την χρήση τέτοιων αισθητήρων σε συνδυασμό με εικονικά περιβάλλοντα (virtual environments) όπως το Oculus Rift ή τα Virtual Caves, δίνουν στους χρήστες μια γεύση για το τι επρόκειτο να δούμε στο μέλλον.

Εικόνα 11 – V.R. με χρήση Kinect και Oculus Rift

2.1.11 Επαυξημένης Πραγματικότητας (Augmented Reality)

Η επαυξημένη πραγματικότητα είναι ένας τρόπος με τον οποίο αντιλαμβανόμαστε την φυσική πραγματικότητα σε συνδυασμό με στοιχεία που παράγονται από κάποιο πρόγραμμα υπολογιστή και αυτό κατά κάποιο τρόπο μας δίνει και άλλους τρόπους αντίληψης ή και πληροφόρησης που συμπληρώνουν τις αισθήσεις μας. Δηλαδή, μέσα στο φυσικό περιβάλλον με χρήση κάποιου προγράμματος και κάμερας του υπολογιστή μας μπορούμε να «δούμε» και άλλα στοιχεία που δεν υπάρχουν στο φυσικό περιβάλλον και δημιουργούνται από το σχετικό λογισμικό. Στο φυσικό περιβάλλον ενσωματώνονται στοιχεία που δημιουργούνται από το λογισμικό δίνοντας παραπάνω δυνατότητες στις αισθήσεις μας και στην πληροφόρησή μας. Ο άνθρωπος που κάνει χρήση προγραμμάτων Επαυξημένης Πραγματικότητας έχει αντίληψη, επαφή με το φυσικό περιβάλλον και ταυτόχρονα μέσα σε αυτό αντιλαμβάνεται και τα ψηφιακά αντικείμενα επαυξημένης πραγματικότητας με τα οποία και αλληλεπιδρά.

Στην περίπτωση του Kinect τέτοιες εφαρμογές, χρησιμοποιούν τον αισθητήρα για την αναγνώριση των κινήσεων από τους χρήστες. Με αυτό τον τρόπο γίνεται η άμεση σύνδεση του φυσικού περιβάλλοντος με το εικονικό περιβάλλον και προσφέρεται μια ακόμα διάσταση στην διεπαφή. Τέτοιου είδους εφαρμογές συναντάμε σε ηλεκτρονικά παιχνίδια, εκπαιδευτικούς προσομοιωτές (αυτοκινήτων, αεροπλάνων κ.τ.λ), εκπαίδευσης ατόμων με ειδικές ανάγκες κ.α.



Εικόνα 12- Παιχνίδι A.R.

Εικόνα 13 – Πεζοπορία στην A.R.

2.1.12 Α.Μ.Ε.Α.

Με τη χρήση ειδικού λογισμικού, το Kinect της Microsoft επιτρέπει σε άτομα με δυσκολία στην επικοινωνία λόγω αναπηρίας ή και σε ανθρώπους που μιλάνε διαφορετική γλώσσα να επικοινωνήσουν μεταξύ τους, μεταφράζοντας κινήσεις-σήματα σε προφορικό και γραπτό λόγο. Το αποτέλεσμα αυτό επιτεύχθηκε μέσω της συνεργασίας του τμήματος ερευνών της Microsoft στην Ασία με την Κινέζικη Ακαδημία Επιστημών και του Κοινοτικό Πανεπιστήμιο του Πεκίνου. Το λογισμικό αναγνωρίζει τη νοηματική γλώσσα των Η.Π.Α. και της Κίνας

αλλά δύναται να τροποποιηθεί ώστε να αυξηθεί ο αριθμός των κινήσεων που αναγνωρίζει και ο αριθμός των υποστηριζόμενων ομιλούμενων γλωσσών.

Προς το παρόν υποστηρίζονται δύο ειδών λειτουργίες, η λειτουργία μετάφρασης και η λειτουργία επικοινωνίας. Κατά τη λειτουργία μετάφρασης, το πρόγραμμα αντιστοιχεί μια λέξη που θα του δώσουμε σε κινήσεις της νοηματικής γλώσσας και αντιστρόφως. Κατά τη λειτουργία επικοινωνίας, το άτομο που επικοινωνεί μέσω της νοηματικής γλώσσας απεικονίζεται, στην οθόνη του συνομιλητή, μέσω μίας ανθρώπινης φιγούρας άβαταρ, που επαναλαμβάνει τις κινήσεις του πρώτου. Καθώς το Kinect μπορεί να χρησιμοποιηθεί πλέον, πέραν του Xbox 360, μέσω των Windows, αυτομάτως αυτή η εφαρμογή γίνεται ευκολότερα προσβάσιμη. Όπως αποτυπώνεται και στην Εικόνα 14, η χρήση του λογισμικού θα ήταν για παράδειγμα χρήσιμη στην επικοινωνία μεταξύ γιατρών που δε γνωρίζουν τη νοηματική γλώσσα και ασθενών με προβλήματα ακοής τους. Επίσης, θα μπορούσε να επιφέρει την αύξηση των ευκαιριών εργασίας, καθώς θα μπορούσε να βοηθήσει άτομα με προβλήματα στην ακοή να εργαστούν σε κάποια υπηρεσία ή μαγαζί όπου έχει ενσωματωθεί το σύστημα αυτό.

Εικόνα 14 – Μεταφραστής νοηματικής γλώσσας

2.2 ASUS Xtion

Ο αισθητήρας Xtion της ASUS μοιάζει πολύ με το Kinect αν και έχουν κάποιες βασικές διαφορές στα χαρακτηριστικά τους. Παρόλ' αυτά είναι μια σχετικά πιο οικονομική συσκευή η οποία ανταγωνίζεται σημαντικά τον αισθητήρα της Microsoft καθώς χρησιμοποιεί είτε λίγο ή πολύ στις ίδιες εφαρμογές με το Kinect. Σε αυτό το κομμάτι, θα αναλυθούν κυρίως οι βασικές διαφορές που έχουν μεταξύ τους.

Και οι δύο αισθητήρες χρησιμοποιούν την υπέρυθρη τεχνολογία Prime Sense (βλ.: 1.1). Οπότε όλα τα χαρακτηριστικά σχετικά με την ανίχνευση της κίνησης σώματος είναι σχεδόν ίδια.

Xtion vs Kinect:

XTION

- Έχει πιο συμπαγή εμβέλεια: (7" x 2" x 1.5" έναντι 12" x 3" x 2.5")
- Είναι πιο ελαφρύ (0.5 lb έναντι 3.0 lb)
- Δεν απαιτεί επιπλέον τροφοδότηση ρεύματος πέρα από αυτή του USB
- Έχει χαμηλότερες παρεμβολές όταν χρησιμοποιούμε παραπάνω από έναν αισθητήρα
- Χρησιμοποιεί καλύτερη RGB κάμερα
- Δεν έχει μηχανοκίνητη βάση
- Έλλειψη σε drivers
- Παρατηρείται αδυναμία λειτουργίας με τις θύρες USB 3.0

2.3 Blender

Το Blender είναι πρόγραμμα σχεδίασης 3D γραφικών, είναι ελεύθερο λογισμικό και διανέμεται από την άδεια GNU General Public License X.

Χρησιμοποιείται για modeling, rigging, προσομοιώσεις νερού, animation, rendering, μη γραμμική επεξεργασία και για δημιουργία αλληλεπιδραστικών 3D εφαρμογών, όπως τα βίντεο-παιχνίδια.

Είναι διαθέσιμο για όλα τα κύρια λειτουργικά συστήματα όπως τα Windows της Microsoft το Linux και το Mac OS X. Επίσης υποστηρίζεται και το Solaris. Το Blender διαθέτει προχωρημένα εργαλεία για animation, διάφορα εργαλεία για σχεδίαση χαρακτήρων και

ρούχων για τον χαρακτήρα, εργαλεία για δημιουργία υλικού καθώς επίσης και τη γλώσσα προγραμματισμού Python για εσωτερικό scripting.

Στην εργασία μας χρησιμοποιήθηκε για γρήγορη απεικόνιση αποτελεσμάτων αλλά και για την διόρθωση κάποιων ανεπιθύμητων λαθών που δημιουργηθήκαν από τον θόρυβο του αισθητήρα. Επιπλέον, ήταν ένα χρήσιμο εργαλείο για την αλλαγή της μορφής των αρχείο από τελικά αποτελέσματα (πχ.: από*.plyσε *.blendκ.τ.λ.)

Εικόνα 15 – Αποτέλεσμα του 3D-scannerστο Blender

2.4 ParaView

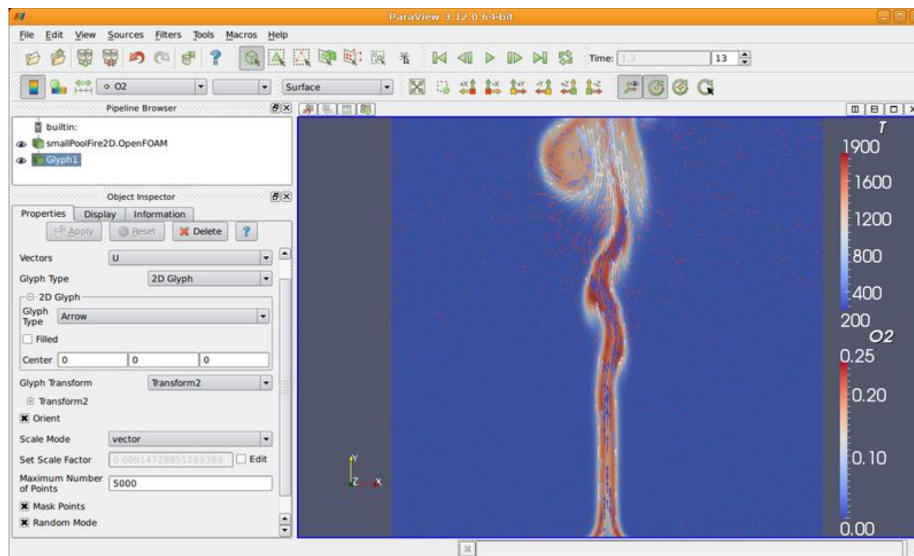
Το ParaView είναι μια πολύ-πλατφόρμα ανοικτού κώδικα για διαδραστικές και επιστημονικές απεικονίσεις. Δουλεύει με την αρχιτεκτονική πελάτη-εξυπηρετητή για να διευκολύνει την απεικόνιση των συνόλων δεδομένων (datasets).

Είναι μια εφαρμογή που χτίστηκε πάνω από στην VTK library (Visualization Tool Kit). Όπου VTK είναι ένα σύνολο βιβλιοθηκών που παρέχουν υπηρεσίες οπτικοακουστικών δεδομένων, task management κ.α.

Χρησιμοποιείται γενικότερα σε εφαρμογές που χρησιμοποιούν data-parallelism σε συστήματα με κοινόχρηστη ή κατανεμημένη μνήμη και clusters.

Οι χρήστες του ParaView μπορούν να κατασκευάζουν γρήγορα τις απεικονίσεις χρησιμοποιώντας ποιοτικές και ποσοτικές τεχνικές. Η προσπέλαση των δεδομένων μπορεί να γίνει διαδραστικά σε 3D ή μέσω προγραμματισμού.

Λόγος ανάπτυξης ήταν η ανάλυση εξαιρετικά μεγάλων όγκων δεδομένων, χρησιμοποιώντας κατανεμημένη υπολογιστική μνήμη. Αυτό το κάνει προσιτό για μεγάλα υπολογιστικά συστήματα επιπέδου TeraScale αλλά και για ένα μέσω φορητό υπολογιστή. Όντας ανοιχτού κώδικα, μπορεί να εγκατασταθεί σχεδόν σε όλα τα δημοφιλή λειτουργικά συστήματα. Μερικά από αυτά είναι Windows, MacOSX, Unix, Linux, IBMBlueGene κ.α.



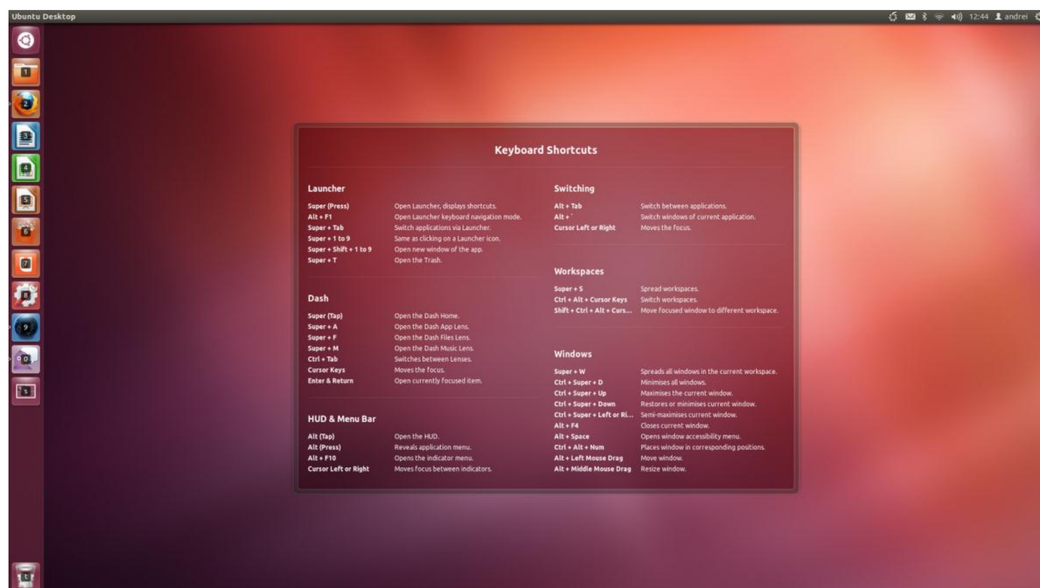
Εικόνα 16 – ParaView

2.5 Λειτουργικό σύστημα (Λ.Σ.)

Ubuntu 12.04

Το Ubuntu είναι ένα ανοικτού κώδικα, ελεύθερο και δωρεάν λειτουργικό σύστημα βασισμένο στον πυρήνα Linux. Το όνομά του προέρχεται από την έννοια ubuntu των Ζουλού και Κόσα (Xhosa), που σημαίνει «Ανθρωπότητα». Το Ubuntu ξεκίνησε το 2004, βασισμένο στη διανομή Debian. Ο στόχος του Ubuntu είναι η παροχή ενός διαρκώς ενημερωμένου, σταθερού λειτουργικού συστήματος για τον μέσο χρήστη, με ενισχυμένη έμφαση στην ευκολία χρήσης και εγκατάστασης. Το Ubuntu έχει χαρακτηριστεί ως η πιο δημοφιλής διανομή Linux για επιτραπέζιους υπολογιστές, διεκδικώντας περίπου το 30% επί του συνόλου των Linux συστημάτων σύμφωνα με έρευνα του 2007.

Το γεγονός ότι είναι βασιζόμενο σε Linux προσφέρει μεγαλύτερη ευελιξία στην εγκατάσταση εφαρμογών και περισσότερες δυνατότητες διαχείρισης των πόρων του συστήματος. Επιπλέον, ο χρήστης έχει πρόσβαση σε όλα τα αρχεία και δεδομένα του συστήματος κάτι που δεν γίνεται σε Λ.Σ. όπως Windows και Mac OS με αυτό τον τρόπο προσφέρει μεγαλύτερη ευελιξία στην διαχείριση δεδομένων. Επίσης, συγκριτικά με τα εμπορικά Λ.Σ. που αναφέραμε έχει μικρότερες απαιτήσεις συστήματος, χαρακτηριστικό που μας παρέχει την δυνατότητα χρήσης παραπάνω δυνατοτήτων του μηχανήματος που χρησιμοποιούμε.



Εικόνα 17 – Ubuntu 12.04

Απαιτήσεις συστήματος:

Έκδοση Desktop

- Για την έκδοση Desktop,με περιβάλλον GNOME
- 1 GHzMHz x86 ή x64 επεξεργαστής
- 512 MB μνήμη (RAM)
- 5 GB χώρο στο σκληρό δίσκο
- Κάρτα γραφικών με δυνατότατα απεικόνισης 1024x768

Έκδοση Netbook Edition

Η έκδοση ήταν σχεδιασμένη ειδικά για Netbook, με σκοπό μεταξύ άλλων να αποδίδει καλύτερα στις μικρές οθόνες. Από την παρούσα έκδοση 11.04 (Natty Narwhal) η έκδοσηNetbook Editionέχει συγχωρεθεί με την έκδοση Desktop Edition. Όσο κυκλοφορούσε η έκδοση για Netbook οι απατήσεις ήταν οι κάτωθι:

- Επεξεργαστής Intel Atom στα 1.6GHz
- 512 μνήμη RAM
- GB χώρο στο σκληρό δίσκο
- Κάρτα γραφικών με δυνατότατα απεικόνισης 800x600 και 3D

2.6 Βάση κίνησης 360°

Για την υλοποίηση της εφαρμογής μας ήταν απαραίτητη η περιστροφή μικρών αντικειμένων, κυκλικά στις 360 μοίρες. Η διαδικασία αυτή είχε ως στόχο την δημιουργία πολλαπλών στιγμιότυπων περιμετρικά των αντικειμένων μας.

Μετά από έρευνα αγοράς, βρέθηκαν αρκετές λύσεις στο πρόβλημα μας αλλά κρίθηκαν ακριβές για το χαμηλό ύψος του προϋπολογισμού που είχαμε ορίσει. Συνεπώς, κρίθηκε προτιμότερο και οικονομικότερο, η δημιουργία μιας χειροποίητης βάσης από υλικά τα οποία επί της στιγμής δεν χρησιμοποιούνταν και ήταν αποθηκευμένα.

Προκειμένου να επιτευχθεί μια περιστρεφόμενη βάση χρησιμοποιήθηκε το χαμηλό μέρος μιας καρέκλας γραφείου, τις οποίας αφαιρέθηκε το κάθισμα και τα ροδάκια. Τοποθετήθηκε ανάποδα ο βραχίονας ώστε να προστεθεί μια λεία επιφάνια που θα προοριζόταν στη συνέχεια για τα αντικείμενα. Με αυτό τον τρόπο φτιάχτηκε μια ελαφριά, εύχρηστη βάση με δυνατότητα πλήρους περιστροφής (360°).

Εικόνα 18 - Περιστρεφόμενη βάση με το βασικό αντικείμενο πειραματισμού (Σουρικάτες)

3 Τεχνολογίες Υλοποίησης

3.1 OpenNI (Open Natural Interaction)

Δημιουργήθηκε από έναν αφιλοκερδή οργανισμό, ο οποίος απαρτίζεται από διάφορες εταιρίες, συμπεριλαμβανομένου και της Prime Sence Ltd. οι οποίες θέλησαν να θέσουν ένα βιομηχανικό πρότυπο λειτουργικότητας για τις συσκευές φυσική διεπαφής χρήστη – υπολογιστή (**NaturalUserInterface**) και κυκλοφόρησε το Δεκέμβριο του 2010.

Το OpenNI αναπτύχθηκε σε C/C++ έτσι ώστε να μπορεί να χρησιμοποιηθεί από διάφορα λειτουργικά συστήματα, όπως Mac OSX, Ubuntu, Windows. Είναι το επίσημο λογισμικό των Xtion συσκευών της Asus, αλλά μπορεί να λειτουργήσει και με το Kinect. Παρέχει επικοινωνία με τον αισθητήρα βάθους, την κάμερα χρώματος, τα μικρόφωνα και τη μηχανοκίνητη βάση. Το OpenNI, όμως, συνοδεύεται και από μια ενδιάμεση βιβλιοθήκη η οποία λέγεται NITE και είναι εξοπλισμένο με τεχνολογίες αναγνώρισης φωνής (Voice Recognition), αναγνώρισης χειρονομιών χεριών (Hand Gesture Recognition), και ανίχνευση σκελετού (Skeleton Tracking). Η ανίχνευση του σκελετού του χρήστη πετούσε αρχική στάση βαθμονόμησης (βλ. Εικόνα 13) αλλά στις πιο πρόσφατες εκδόσεις του, το OpenNI/NITE, δεν χρειάζεται αρχική στάση βαθμονόμησης (χαρακτηριστικό που είχε μόνο το MS DK).

Εικόνα 19 – NITE Skeleton Tracking

3.2 OpenKinect – Libfreenect

Το Libfreenect ήταν η πρώτη βιβλιοθήκη που δημιουργήθηκε για την αποκωδικοποίηση των δεδομένων του Kinect. Κυκλοφόρησε στις αρχές του Νοεμβρίου του 2010 από το Héctor Martín, λίγο μετά την κυκλοφορία του Kinect στην αγορά. Δημοσιεύθηκε στην ιστοσελίδα του Open Kinect στις 10 Νοεμβρίου όπου αποτέλεσε την αρχή της σύνταξης μιας μεγάλης κοινωνίας προγραμματιστών για τη χρήση του και την εξέλιξή του. Αναπτυγμένο σε C και Python, το Libfreenect παρέχει ένα μεγάλο αριθμό από wrappers, σε διάφορες γλώσσες, και την κατάλληλη βιβλιογραφία για τη χρήση τους. Η βιβλιοθήκη παρέχει πρόσβαση στην κάμερα, στο αισθητήρα βάθους (βλ. Εικόνα 13), στο LED και στη μηχανοκίνητη βάση της συσκευής. Χαρακτηρίζεται API χαμηλού επιπέδου γιατί πέρα από την επικοινωνία με τη συσκευή δεν παρέχει μεθόδους όπως ανίχνευση σκελετού (Skeleton Tracking). Υπάρχει όμως ένας μεγάλος αριθμός από προγραμματιστές που το χρησιμοποιούν, κι αυτό είναι ελεύθερο για εμπορική χρήση με δυνατότητα να τρέξει σε Windows, Mac OSX και Linux.Τέλος η βιβλιοθήκη είναι αρκετά δύσκολη στην εγκατάστασή της γιατί χρειάζεται χειροκίνητη τοποθέτηση των αρχείων της μέσα στους φακέλους του συστήματος του προγραμματιστή.

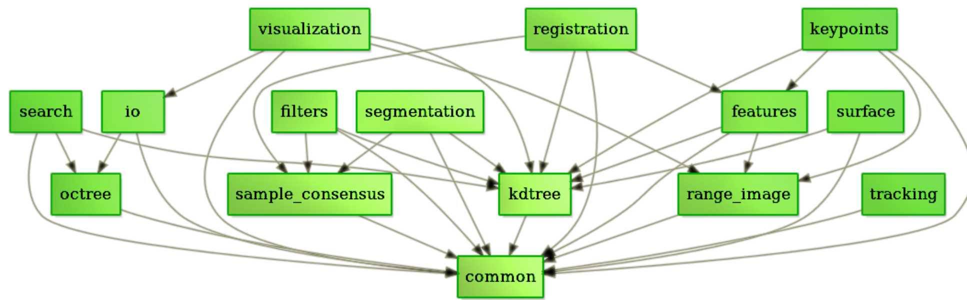
3.3 Point-Cloud Library (PCL)

Point-Cloud (Σύννεφο σημείων) είναι ένα σύνολο από δεδομένα σημείων (data points) σε ένα σύστημα συντεταγμένων.

Στο τρισδιάστατο σύστημα συντεταγμένων που θα χρησιμοποιήσουμε αυτά τα σημεία ορίζονται ως X, Y, Z που αντιπροσωπεύουν τις επιφάνειες των αντικειμένων.

Η PCL είναι μια αυτόνομη ανοικτού λογισμικού βιβλιοθήκη προγραμματισμού. Δημιουργήθηκε από ένα μεγάλο αριθμό ερευνητών , προγραμματιστών και μηχανικών από όλο τον κόσμο. Περιέχει πολλούς αλγόριθμους τελευταίας τεχνολογίας για n-διαστάσεων point-clouds καθώς και 3D γεωμετρική επεξεργασία. Χρησιμοποιεί αλγόριθμους φιλτραρίσματος, ανίχνευσης χαρακτηριστικών, αναδημιουργία επιφανίων, ταυτοποίησης αντικειμένων κ.α. Είναι γραμμένη σε C++ και δημοσιεύτηκε υπό την BSD License.

Εφαρμογές της μπορούμε να βρούμε στην ρομποτική, 3D scanners, αναγνώριση αντικειμένων κ.α.



Εικόνα 20 - Δομή της PCL

3.3.1 Αλγόριθμοι

Φίλτρα

Pass-through:

Ο συγκεκριμένος αλγόριθμος λειτούργει σαν φίλτρο που απορρίπτει συγκεκριμένα pixels, με βάση την απόσταση τους από τον αισθητήρα. Η διαδικασία αυτή μπορεί να συμβεί σε όλες τις διαστάσεις (X,Y,Z). Με αυτό τον αλγόριθμο περιορίσαμε την εμβέλεια του αισθητήρα ούτως ώστε να απομονώσουμε αντικείμενα που υπήρχαν γύρω από τον χώρο πειραματισμού μας.

Εικόνα 21 – pass through (πριν και μετά)

Παράδειγμα αλγορίθμου Pass-through:

```
pcl::PassThrough<PointType> ptfilter (true); // Initializing with true will allow us to extract the removed indices
ptfilter.setInputCloud (cloud_in);
ptfilter.setFilterFieldName ("x");
ptfilter.setFilterLimits (0.0, 1000.0);
ptfilter.filter (*indices_x);
// The indices_x array indexes all points of cloud_in that have x between 0.0 and 1000.0
indices_rem = ptfilter.getRemovedIndices ();
// The indices_rem array indexes all points of cloud_in that have x smaller than 0.0 or larger than 1000.0
// and also indexes all non-finite points of cloud_in
ptfilter.setIndices (indices_x);
ptfilter.setFilterFieldName ("z");
ptfilter.setFilterLimits (-10.0, 10.0);
ptfilter.setNegative (true);
ptfilter.filter (*indices_xz);
// The indices_xz array indexes all points of cloud_in that have x between 0.0 and 1000.0 and z larger than 10.0 or smaller
      than -10.0
ptfilter.setIndices (indices_xz);
ptfilter.setFilterFieldName ("intensity");
ptfilter.setFilterLimits (FLT_MIN, 0.5);
ptfilter.setNegative (false);
ptfilter.filter (*cloud_out);
// The resulting cloud_out contains all points of cloud_in that are finite and have:
// x between 0.0 and 1000.0, z larger than 10.0 or smaller than -10.0 and intensity smaller than 0.5.
```

Moving Least Squares (MLS):

Είναι μια μέθοδος αφαίρεσης θορύβου που χρησιμοποιείται με βάση την σύγκριση των τοπικών χαρτών χρησιμοποιώντας την MLS (80XIII) τεχνική.

Για κάθε σημείο **p** του cloud, ένα τοπικό επίπεδο υπολογίζεται για κάθε γειτονικό του, μέσω της ανάλυσης των κύριων συνιστωσών (PCA) και 2D συνάρτηση g(x,y) (συνήθως 2 ή 3 βαθμού). Στην εξίσωση παρακάτω, όπου **q** αναφέρεται στην προέλευση αναφοράς (δηλαδή του τοπικού επιπέδου), **n** είναι το κανονικό επίπεδο και **p** είναι ένα σημείο από τα γειτονικά του **q**. Στην συνέχεια το αρχικό σημείο προβάλλεται σε αυτή τη λειτουργία και ωθείται εντός του τελικού cloud.

Στην περίπτωση μας το MLS χρησιμοποιείτε σαν φίλτρο για την ανακατασκευή της επιφάνειας του point-cloud. Αυτό μπορεί να επιτευχθεί με την χρήση είτε υπερδειγματοληψίας είτε με μείωση των points από το σύνολο (Down Sampling).

Ακόμα στην PCL η μέθοδος αυτή επιτρέπει αύξηση των points και βοηθάει ιδιαιτέρα στην ανίχνευση των κάθετων διανυσμάτων (**normal**) που θα αναφέρουμε στην συνέχεια.

Εικόνα 22 – Φίλτρο MLS

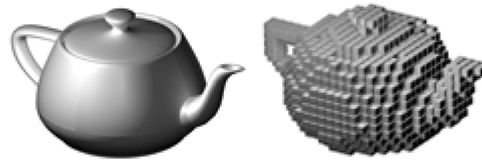
Παράδειγμα αλγορίθμου MLS

```
// Load input file into a PointCloud<T> with an appropriate type
pcl::PointCloud<pcl::PointXYZ>::Ptr cloud (new pcl::PointCloud<pcl::PointXYZ> ());
// Load bun0.pcd -- should be available with the PCL archive in test
pcl::io::loadPCDFile ("bun0.pcd", *cloud);

// Create a KD-Tree
pcl::search::KdTree<pcl::PointXYZ>::Ptr tree (new pcl::search::KdTree<pcl::PointXYZ>);

// Output has the PointNormal type in order to store the normals calculated by MLS
pcl::PointCloud<pcl::PointNormal> mls_points;

// Init object (second point type is for the normals, even if unused)
pcl::MovingLeastSquares<pcl::PointXYZ, pcl::PointNormal> mls;

mls.setComputeNormals (true);

// Set parameters
mls.setInputCloud (cloud);
mls.setPolynomialFit (true);
mls.setSearchMethod (tree);
mls.setSearchRadius (0.03);

// Reconstruct
mls.process (mls_points);
```

Voxel Grid: Αυτή η κλάση δημιουργεί ένα πλέγμα 3D-voxel (3D-voxel είναι τρισδιάστατα κουτιά στον χώρο) πάνω στα δεδομένα εισόδου. Εν συνεχεία, όλα τα σημεία κάθε voxel θα προσεγγιστούν συγκριτικά με το κεντρικό τους (δηλαδή θα μειωθεί ο ρυθμός δειγματοληψίας).

Είναι μια αργή μέθοδος αλλά αντιπροσωπεύει την επικείμενη επιφάνεια με μεγαλύτερη ακρίβεια.

Η μέθοδος αυτή χρησιμοποιήθηκε για την μείωση των σημείων, για μεγαλύτερη ταχύτητα στην επεξεργασία, φιλτράρισμα και εγγραφή.



Εικόνα 23 – Voxe lGrid

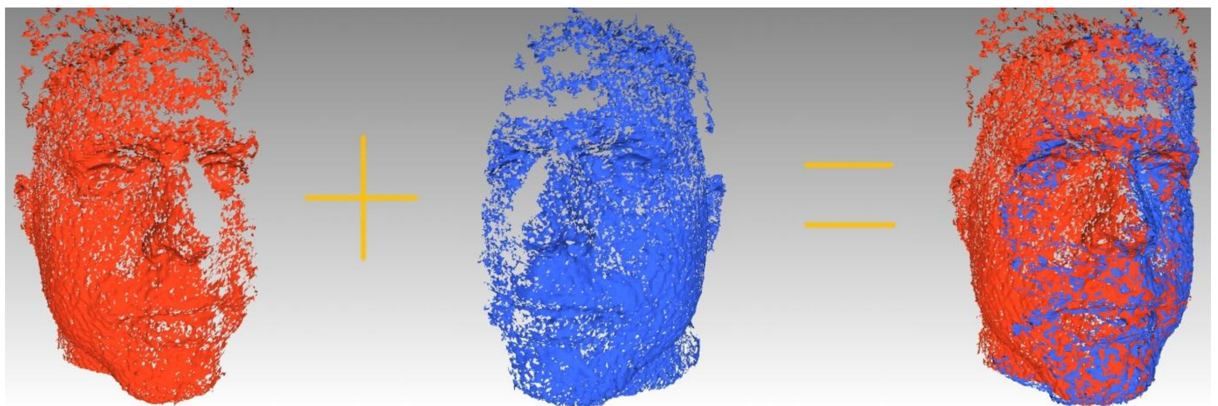
Παράδειγμα αλγορίθμου Voxel Grid:

```
// Create the filtering object
pcl::VoxelGrid<pcl::PCLPointCloud2> sor;
sor.setInputCloud (cloud);
sor.setLeafSize (0.01f, 0.01f, 0.01f);
sor.filter (*cloud_filtered);
```

Αλγοριθμοι Εγγραφής

Iterative closest point (ICP): Είναι ένας αλγόριθμος που στοχεύει στην μείωση της διαφοράς μεταξύ δύο point-clouds. Συχνά χρησιμοποιείται για την ένωση διαφορετικών επιφανειών σε 2D ή 3D, οι οποίες προέρχονται από διαφορετικές σαρώσεις. Η χρήση του είναι απαραίτητη για τον εντοπισμό της βέλτιστης διαδρομής των ρομπότ κ.τ.λ.

Σε αυτό τον αλγόριθμο τα clouds χωρίζονται σε πηγής και στόχου. Ένα από τα δύο κρατιέται σταθερό ενώ το άλλο μετασχηματίζεται σε σχέση με το πρώτο. Στην συνέχεια αναθεωρεί επαναληπτικά τον μετασχηματισμό ούτος ώστε να ελαχιστοποιηθεί η απόσταση προς το σημείο αναφοράς.



Εικόνα 24 – ICP: Συνδέοντας δύο point-clouds

Παράδειγμα αλγορίθμου ICP:

```
pcl::IterativeClosestPoint<pcl::PointXYZ, pcl::PointXYZ> icp;
icp.setInputCloud(cloud_in);
icp.setInputTarget(cloud_out);
pcl::PointCloud<pcl::PointXYZ> Final;
icp.align(Final);
```

ELCH (Explicit Loop Closing Heuristic): Ο αλγόριθμος αυτός δημιουργήθηκε για να βρίσκει και να διορθώνει την τροχιά των ρομπότ. Στην εργασία μας το χρησιμοποιήσαμε για να διορθώνει τα πιθανά λάθη που παρουσιάζονται στους βρόγχους μας [VII].

Ελέγχει τους βρόγχους με την χρήση του Ευκλείδειου θεωρήματος για να υπολογίσει την διαφορά μεταξύ του παρόντος με όλα τα προηγούμενα (clouds στην περίπτωση μας). Ένας αριθμός ελάχιστων σαρώσεων (π.χ. 20) χρησιμοποιείται για την παράκαμψη του κλεισίματος των βρόγχων από τις διαδοχικές σαρώσεις. Χρησιμοποιώντας τη πρώτη και τη τελευταία σάρωση δημιουργούμε δύο meta-scans οι οποίες περνούν από το ICP. Η διαφορά της θέσης της τελευταίας σάρωσης και αυτής που προέρχεται από το ICP αποδίδει τα πιθανά λάθη στο μετασχηματισμό, που πρέπει να κατανεμηθεί μεταξύ όλων των σαρώσεων του βρόγχου. Δηλαδή η διαφορά μεταξύ των πρώτων σαρώσεών πρέπει να είναι η ίδια σε όλες τις διαδοχικές που θα βρεθούν στον βρόγχο.

Παράδειγμα αλγορίθμου ELCH:

```
pcl::registration::ELCH<PointType> elch;
elch.setReg (icp);
for (int i = 1; i < n; i++)
{
  elch.addPointCloud (cloud[i]);
}
elch.setLoopStart (first);
elch.setLoopEnd (last);
elch.compute ();
```

LUM: Είναι μια μέθοδος σύγκρισης σαρώσεών βασισμένη στην δουλειά των Lu και Milios [IV] των οποίων ο αλγόριθμος (Graph SLAM) αφορά τη διαχείριση των δεδομένων καταγραφής σε ένα γράφημα:

Οι κορυφές του αντιπροσωπεύουν τις θέσεις (πόζες) και κρατάει τα δεδομένα των σημείων του cloud και των σχετικών μετασχηματισμών τους.

Οι ακμές αντιπροσωπεύουν τους περιορισμούς των θέσεων και κρατάνε τις αντίστοιχιες των δεδομένων μεταξύ δύο clouds.

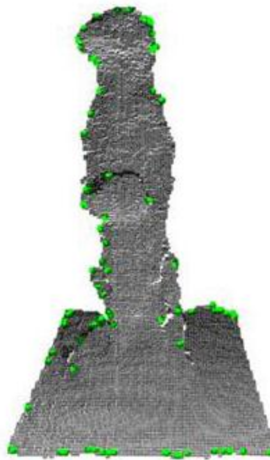
Παράδειγμα αλγορίθμου LUM:

```
pcl::registration::LUM<PointType> lum;
lum.setMaxIterations (lumIter);
lum.setConvergenceThreshold (0.001f);
for (int i = 1; i < n; i++)
{
  lum.addPointCloud (cloud[i]);
}
//for all point pairs (i, j)
lum.setCorrespondences (i, j, correspondences);
lum.compute ();
```

Εξαγωγή Χαρακτηριστικών

Με στόχο την εξαγωγή των βέλτιστων προς χρήση χαρακτηριστικών ακολουθήθηκαν οι παρακάτω διαδικασίες.

Key Points: Αρχικά, πρέπει να βρούμε τα πιο κρίσιμα σημεία (key points) από κάθε cloud. Τα σημεία αυτά αντιπροσωπεύουν τα πιο βασικά χαρακτηριστικά που θα χρησιμοποιήσουμε αργότερα για την εξαγωγή των χαρακτηριστικών και διανυσμάτων επιφάνειας (normal).



Εικόνα 25 - Key-Points (από σάρωση σουρικάτων)

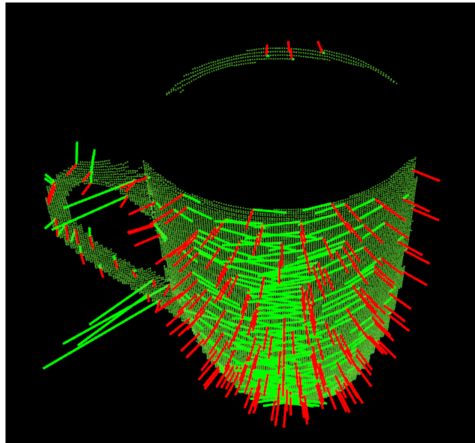
Παράδειγμα αλγορίθμου key-point extraction:

```
pcl::UniformSampling<PointType> uniform_sampling;
uniform_sampling.setInputCloud (model);
uniform_sampling.setRadiusSearch (model_ss_);
uniform_sampling.compute (sampled_indices);
pcl::copyPointCloud (*model, sampled_indices.points, *model_keypoints);
```

Normals: Είναι σημαντικές ιδιότητες μια γεωμετρικής επιφάνειας και χρησιμοποιείται ευρέως σε πολλούς τομείς, όπως εφαρμογές γραφικών. Βασική ιδιότητα τους είναι η εφαρμογή του σωστού φωτισμού, της σκίασης και άλλων οπτικών εφέ.

Δεδομένου ότι τα σύνολα δεδομένων του cloud που έχουμε, αντιπροσωπεύουν ένα σύνολο σημείων της πραγματικής επιφάνειας, υπάρχουν δύο δυνατότητες:

- Να εξασφαλιστεί η σχετική επιφάνεια από το σύνολο των δεδομένων που περιέχονται στο cloud, χρησιμοποιώντας τεχνικές **surface meshing** και μέσω αυτής να τα υπολογίσουμε.
- Με χρήση της κατά προσέγγιση υπολογισμού των σημείων, από το σύνολο των δεδομένων μας.



Εικόνα 26 – Normals από φλιτζάνι καφέ

Παράδειγμα αλγορίθμου Normal estimation:

```
// Create the normal estimation class, and pass the input dataset to it
pcl::NormalEstimation<pcl::PointXYZ, pcl::Normal> ne;
ne.setInputCloud (cloud);

// Create an empty kdtree representation, and pass it to the normal estimation object.
// Its content will be filled inside the object, based on the given input dataset (as no other search surface is given).
pcl::search::KdTree<pcl::PointXYZ>::Ptr tree (new pcl::search::KdTree<pcl::PointXYZ> ());
ne.setSearchMethod (tree);

// Output datasets
pcl::PointCloud<pcl::Normal>::Ptr cloud_normals (new pcl::PointCloud<pcl::Normal>);

// Use all neighbors in a sphere of radius 3cm
ne.setRadiusSearch (0.03);

// Compute the features
ne.compute (*cloud_normals);
```

Fast Point Feature Histogram (FPFH):

Ο στόχος του PFH είναι η κωδικοποιήση των **k**-γειτονικών γεωμετρικών ιδιοτήτων ενός σημείου με τη γενίκευση της μέσης καμπυλότητας της γύρω από το σημείο, χρησιμοποιώντας ένα πολυδιάστατο ιστόγραμμα των τιμών. Παρέχει μια ενημερωτική υπογραφή για την εκπροσώπηση των χαρακτιριστικών (feutures), είναι αμετάβλητη ως προς την θέση 6-διαστάσεων της επιφάνειας και τα καταφέρνει πολύ καλά με τις διαφορετικές πυκνότητες

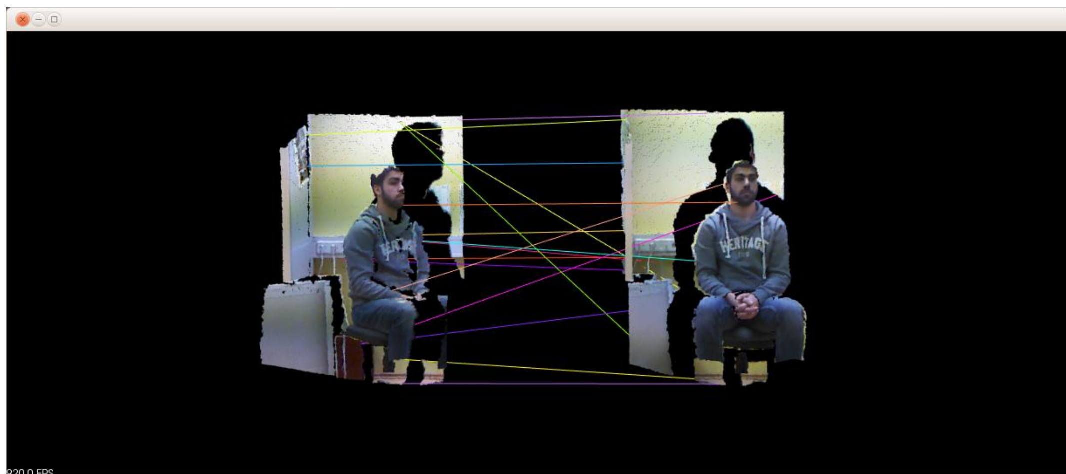
δειγματοληψίας ή τα επίπεδα θορύβου στο παρόν 3D-voxel ή στα γειτονικά. Τα PFH αντιπροσωπεύουν την σχέση μεταξυ των σημείων στα **k**-γειτονικά και των normals της επιφάνειας που έχουν υπολογιστεί. Πιο απλά, προσπαθεί να υπολογίσει το βέλτιστο δείγμα της αλλαγής επιφάνειας, λαμβάνοντας υπόψην όλες τις εναλλαγές μεταξύ των κατευθύνσεων που έχουν τα normals.

Παράδειγμα αλγορίθμου FPFH:

```
// Create the PFH estimation class, and pass the input dataset+normals to it
pcl::PFHEstimation<pcl::PointXYZ, pcl::Normal, pcl::PFHSignature125> pfh;
pfh.setInputCloud (cloud);
pfh.setInputNormals (normals);
// alternatively, if cloud is of tpe PointNormal, do pfh.setInputNormals (cloud);

// Create an empty kdtree representation, and pass it to the PFH estimation object.
// Its content will be filled inside the object, based on the given input dataset (as no other search surface is given).
pcl::search::KdTree<pcl::PointXYZ>::Ptr tree (new pcl::search::KdTree<pcl::PointXYZ> ());
//pcl::KdTreeFLANN<pcl::PointXYZ>::Ptr tree (new pcl::KdTreeFLANN<pcl::PointXYZ> ()); -- older call for PCL 1.5-
pfh.setSearchMethod (tree);

// Output datasets
pcl::PointCloud<pcl::PFHSignature125>::Ptr pfhs (new pcl::PointCloud<pcl::PFHSignature125> ());

// Use all neighbors in a sphere of radius 5cm
// IMPORTANT: the radius used here has to be larger than the radius used to estimate the surface normals!!!
pfh.setRadiusSearch (0.05);

// Compute the features
pfh.compute (*pfhs);
```

Υπολογισμός/Απόρριψη αντιστοιχιών

Υπολογισμός αντιστοιχιών (Correspondence Estimation): Αυτός ο αλγόριθμος δημιουργήθηκε με σκοπό να βρίσκει τις σωστές αντιστοιχίες των σημείων σε δύο ελεγχόμενα clouds ώστε να μπορούν να μετασχηματιστούν και να συνδυαστούν μεταξύ τους, δίνοντας τις εκτιμώμενες συντεταγμένες ,βάση των σημείων αυτών χρησιμοποιώντας τα FPFH που αναλύσαμε παραπάνω.



Εικόνα 27 – Αντιστοιχίες (correspondences)

Παράδειγμα αλγορίθμου:

```
pcl::CorrespondenceEstimation<pcl::PointXYZ, pcl::PointXYZ> est;
est.setInputSource (source);
est.setInputTarget (target);

pcl::Correspondences all_correspondences;
// Determine all reciprocal correspondences
est.determineReciprocalCorrespondences (all correspondences);
```

Απόρριψη αντιστοιχιών (Correspondence Rejector): Ο συγκεκριμένος αλγόριθμος λειτουργεί αντίστροφα σε σχέση με τον προηγούμενο. Με την χρήση κάποιων επιπρόσθετων παραμέτρων επιτρέπει την απόρριψη αντιστοιχιών που βρήκαμε με την Correspondence estimation. Με αυτόν τον τρόπο βελτιστοποιούμε το αποτέλεσμα αποφεύγοντας πιθανά λάθη.

Παράδειγμα αλγορίθμου:

```
pcl::registration::CorrespondenceRejectorSampleConsensus<pcl::PointXYZRGB> corrsRejectorSAC;
corrsRejectorSAC.setInputCloud(cloud1);
corrsRejectorSAC.setTargetCloud(cloud2);
corrsRejectorSAC.setInlierThreshold(inlier_RANSAC_Thres);
corrsRejectorSAC.setMaxIterations(max_nmr_iterations);
corrsRejectorSAC.setInputCorrespondences(correspondences);
boost::shared_ptr<pcl::Correspondences> correspondencesRejSAC (new pcl::Correspondences);
corrsRejectorSAC.getCorrespondences(*correspondencesRejSAC);
```

Μετασχηματισμοί

Transform Point-Cloud: Έχει ως είσοδο δύο point-clouds και ως έξοδο έναν πίνακα (στην περίπτωση μας matrix4f). Το ένα point-cloud θεωρείτε «πηγή» ενώ το άλλο «στόχος». Η μέθοδος αυτή μετασχηματίζει τον «στόχο» ανάλογα με τις συντεταγμένες που περιέχονται στον πίνακα.

Δημιουργία Επιφάνειας (Surface Reconstruction)

Poisson [X]: Είναι μια κλάση δημιουργίας επιφανειών χρησιμοποιώντας point-clouds. Όταν ένα σημείο είναι μέσα στην επιφάνεια επιστρέφει 1, όταν είναι από έξω επιστρέφει 0. Στην συνέχεια υπολογίζει την κλήση του πεδίου των normal. Με την χρήση αυτής και την κλάση Poisson ο αλγόριθμος εκμαιεύει την επιφάνια που ορίζεται από τον δέκτη.

Εικόνα 28 – Poisson-Mesh (ανακατασκευή επιφάνειας)

Παράδειγμα αλγορίθμου:

```
  Poisson<PointNormal> poisson;
  poisson.setDepth (9);
  poisson.setInputCloud
(cloud_smoothed_normals);
  PolygonMesh mesh;
  poisson.reconstruct (mesh);

  io::saveVTKFile (argv[2], mesh);

  return 0;
}
```

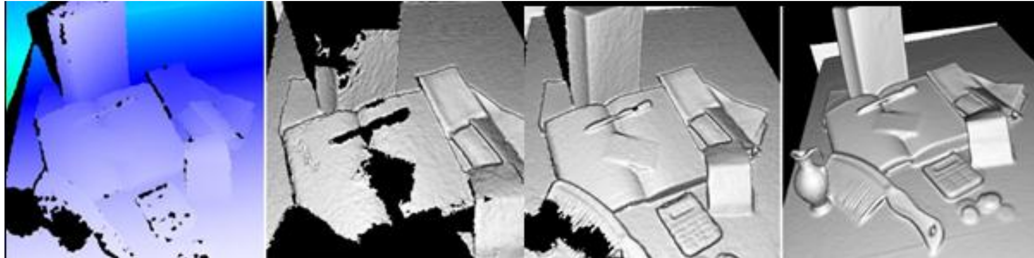
3.3.2 Εφαρμογές βασιζόμενες στην PCL library

Αλγόριθμος Kinect Fusion (KinFu)

Το Kinfu [XII] είναι ένας αλγόριθμος που δημιουργήθηκε από την Microsoft Research το 2011, επιτρέποντας στους χρήστες του, την αναδημιουργία (ανοικοδόμηση) 3Dεπιφανειών σε πραγματικό χρόνο, χρησιμοποιώντας το Kinect. Κάνει χρήση της PCL για να εισάγει τα δεδομένα βάθους που προέρχονται από το Kinect στον υπολογιστή. Με την βοήθεια των αλγορίθμων της, που εξηγήθηκαν παραπάνω φιλτράρει τα δεδομένα για την αποφυγή του θορύβου που εισάγεται. Στην συνέχεια, με την βοήθεια των αλγόριθμων αναδημιουργίας

επιφανειών ενώνει τα points μεταξύ τους με ώστε να χτιστεί και η επιφάνεια να αρχίσει να παίρνει μορφή.

Ωστόσο, απαιτούνται πολύ μεγάλες ποσότητες μνήμης και έχουν παρουσιαστεί πολλά προβλήματα με αντικείμενα που έχουν πολύπλοκες επιφάνειες αλλά αυτό είναι αποτέλεσμα των σχετικά χαμηλών δυνατοτήτων του αισθητήρα.

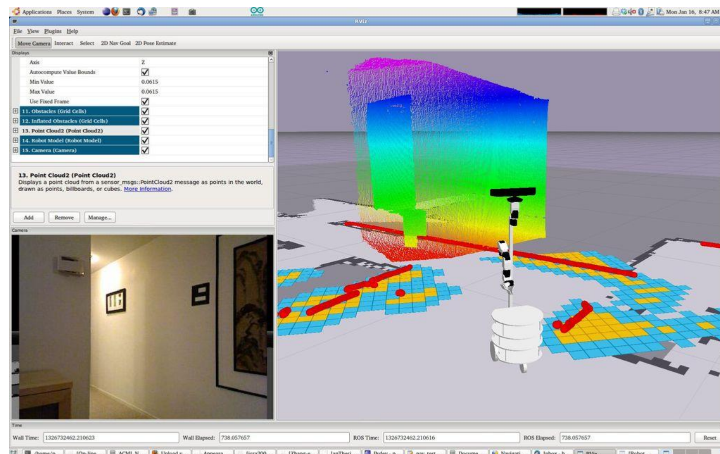


Εικόνα 29 - Παράδειγμα KinectFusion (Kinfu)

Η μεθοδολογία που χρησιμοποιήθηκε για το KinFu μας έδωσε έμπνευση και υλικό για την υλοποίηση της δικιά μας εφαρμογής.

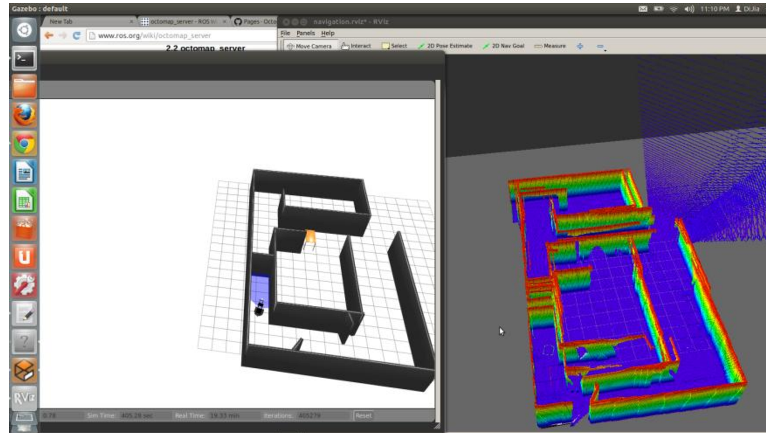
Robotic Operating System (R.O.S)

Το ROS είναι ανοικτό λειτουργικό σύστημα για ρομποτικά συστήματα. Προσφέρει σταθερή υπηρεσία εφαρμογών και Hardware, χαμηλού επιπέδου έλεγχο συσκευών, επικοινωνία μεταξύ των διεργασιών και ένα πακέτο γενικής διαχείρισης του συστήματος. Είναι βασισμένο στην αρχιτεκτονική γραφημάτων όπου η επεξεργασία γίνεται με κόμβους, οι οποίοι μπορούν να στέλνουν μηνύματα μεταξύ πολλαπλών αισθητήρων. Μηνύματα όπως τον έλεγχο, το προγραμματισμό και της ενεργοποίησης τους ή όχι.



Εικόνα30 - Robotic Operating System (ROS)

Σε συστήματα ROS συχνά συναντάμε εφαρμογές των αλγορίθμων από τις βιβλιοθήκες τις PCL και του Kinect . Συνήθως η χρήση προορίζεται για την ανίχνευση μονοπατιών ή την αναγνώρισης περιβάλλοντος στο οποίο κινείται κάποιο ρομπότ.



Εικόνα 31 – Χαρτογράφηση κλειστού χώρου από τα ROS

In-Hand Scanner

Προκειμένου να γίνεται η σάρωση μικρών τρισδιάστατων αντικειμένων, δημιουργήθηκε η εφαρμογή In-Hand Scanner. Η σάρωση γίνεται με τη βοήθεια του χρήστη, ο οποίος με το χέρι (in-hand) πρέπει να γυρίζει το αντικείμενο μπροστά από τον αισθητήρα προς όλες τις κατευθύνσεις ώστε η γεωμετρία της επιφάνειας του να δημιουργηθεί σταδιακά. Οι ακόλουθες προϋποθέσεις είναι απαραίτητες για την βέλτιστη λειτουργία της εφαρμογής.

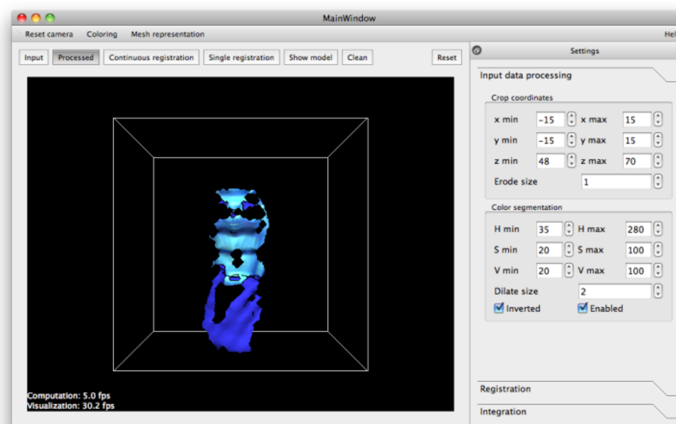
Προϋποθέσεις:

- Τα αντικείμενα πρέπει να έχουν μέγεθος από 10-20εκ. Η εφαρμογή μπορεί να δουλέψει και με μεγαλύτερα αντικείμενα αλλά θα είναι πολύ χρονοβόρο, λαμβάνοντας υπόψη ότι με τις προτεινόμενες διαστάσεις θα εισάγουμε 10000 points ανά στιγμιότυπο.
- Τα αντικείμενα πρέπει να είναι άκαμπτα, αφού ο αλγόριθμος εγγραφής δεν μπορεί να ευθυγραμμίσει τα στιγμιότυπα εάν αυτά είναι παραμορφωμένα.
- Πρέπει να έχουν εμφανή γεωμετρικά χαρακτηριστικά διότι η υφή δεν λαμβάνεται υπόψη κατά την εγγραφή. π.χ. Ένα μπουκάλι δεν μπορεί να ανακατασκευαστεί γιατί η επιφάνια του είναι συμμετρική.
- Το αντικείμενο πρέπει να έχει διαφορετικό χρώμα με αυτό του χρήστη.
- Η κινήσεις κατά την σάρωση πρέπει να είναι απαλές.

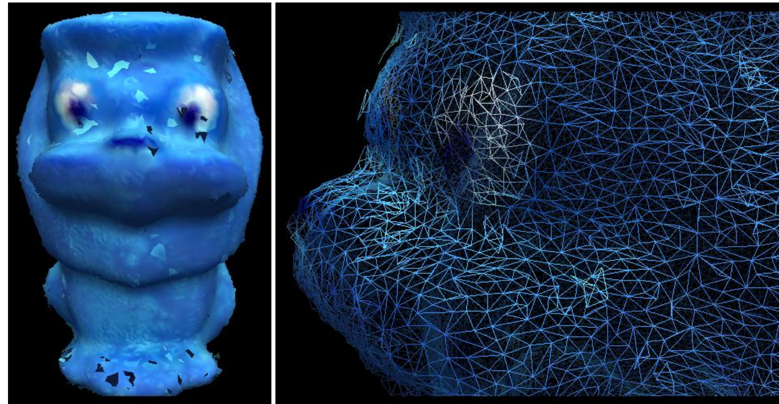
Πως λειτουργεί:

- Η εφαρμογή δημιουργεί ένα αρχικό πλέγμα επιφάνεια και σταδιακά ενσωματώνει νέα σημεία σε ένα κοινό μοντέλο.
- Grabber: Είναι υπεύθυνο για την επικοινωνία με τον αισθητήρα και ενημερώνει την εφαρμογή όταν νέα δεδομένα είναι διαθέσιμα.
- Επεξεργασία των εισαγόμενων δεδομένων
- Εγγραφή: Ευθυγραμμίζει τα επεξεργασμένα δεδομένα στο κοινό σημείο, χρησιμοποιώντας την μέθοδο IterativeClosestPoint (ICP).
- Ολοκλήρωση: Αναδημιουργεί το μοντέλο συγχωνεύοντας τα ευθυγραμμισμένα στιγμιότυπα και τέλος προσθέτει το πλέγμα (mesh) στην επιφάνεια του μοντέλου.

Εικόνα 32 – Αντικείμενο για σάρωση με το In-Hand scanner



Εικόνα33 – Διεπαφή In-Hand scanner



Εικόνα 34 - Αποτέλεσμα In-Hand scanner

4 Υλοποίηση Εργασίας

4.1 Εγκατάσταση PCL

Για Ubuntu:

Η εγκατάσταση θα γίνει μέσα στα PPA (Personal Package Archives) λόγω του γεγονότος ότι η PCL είναι μεγάλης κλίμακας βιβλιοθήκη με πολλές απαιτήσεις σε χώρο και σε πόρους.

i. Ανοίγουμε ένα terminal του Debian που υπάρχει σχεδόν σε όλα τα λειτουργικά βασισμένα σε Linux.
ii. Πρέπει να είμαστε συνδεμένοι σαν διαχειριστές (Root) για να εγκαταστήσουμε οτιδήποτε ή να χρησιμοποιούμε με κάθε εντολή εγκατάστασης την εντολή "sudo".
iii. sudo add-apt-repository ppa:v-launchpad-jochen-sprickerhof-de/pcl
iv. sudo apt-getupdate
v. sudo apt-get install libpcl-all

Μετά το πέρας της εγκατάστασης εκτός από όλα τα αρχεία τις βιβλιοθήκης που έχουν εγκατασταθεί στο σύστημα, θα υπάρχουν έτοιμα πρότζεκτς με εφαρμογές των βασικών αλγόριθμων της.

4.2 Έρευνα και πειράματα

Σε αυτό τον τομέα θα αναλύσουμε την έρευνα και τα πειράματα που έγιναν έτσι ώστε να καταλήξουμε στην τελική μορφή της εφαρμογής. Ο αριθμός 3D-scanners ανοικτού λογισμικού που υπάρχει αυτήν τη στιγμή είναι περιορισμένος και εκτίνεται σε ένα αρκετά μεγάλο εύρος διαφορετικών γλωσσών προγραμματισμού αλλά και διαφορετικών αισθητήρων. Στο κεφάλαιο 3.4 αναφέραμε τα χαρακτηριστικά των KinFu και In-Hand scanner που τελικά μας έδωσαν την έμπνευση για τις τεχνικές και την λογική που θα έπρεπε να χρησιμοποιήσουμε.

4.2.1 OpenNI 3D-image receiver

Η πρώτη προσπάθεια να εξάγουμε εικόνα από το Kinect όταν με την χρήση της βιβλιοθήκης OpenNI που εκείνη την περίοδο ήταν ανοικτό λογισμικό, αφού αγοράστηκε από την Apple και στις 23 Απριλίου του 2014 ο δικτυακός τόπος έκλεισε άρα και η εποχή της ανοικτής διάθεσης και ανάπτυξης του λογισμικού της.

4.2.2 Εγκατάσταση OpenNI

Αφού κατεβάσαμε την επιθυμητή έκδοση για Ubuntu, ακολουθήσαμε παρόμοια διαδικασία με την εγκατάσταση της PCL.

Βήμα 1:

Σε terminalπληκτρολογούμε και κάνουμε εγκατάσταση τα εξής που είναι απαραίτητα για την OpenNI:

- sudo apt-get update
- sudo apt-get install mono-complete
- sudo apt-get install libusb-1.0-0-dev
- sudo apt-get install freeglut3-dev

Βήμα 2:

Στον φάκελο που έχουμε αποσυμπιέσει τα αρχεία που κατεβάσαμε:

- cd OpenNI
- sudo ./install.sh
- cd ../Sensor
- sudo ./install.sh
- cd ../Nite
- sudo ./install.sh

Βήμα 3:

Προετοιμάζουμε τους φακέλους για να κατεβάσουμε τον πηγαίο κώδικα:

- mkdir ~/kinect&& cd ~/kinect
- git clone https://github.com/OpenNI/OpenNI.git
- cd OpenNI/Platform/Linux/Build
- make &&sudo make install

4.2.3 Επεξήγηση κώδικα

Για να αναπαραστήσουμε τα δεδομένα που εισάγουμε από τον αισθητήρα πρέπει να αρχικοποιούμε σε κάθε στιγμιότυπο (frame) την σκηνή, τα υλικά και τον φωτισμό που δίνει την τρισδιάστατη αίσθηση. Για αυτό το λόγο δημιουργήσαμε τις μεθόδους που ακολουθούν :

1 **void keyboardFunction&specialKeys:**

Αυτή η μέθοδος φτιάχτηκε για να μπορέσουμε να κάνουμε χρήση του πληκτρολογίου. Η keyboardFunction αφορά τα γενικά κουμπιά (βασικοί χαρακτήρες και νούμερα) και η specialKeys τα κουμπιά βέλη. Σκοπός ήταν να δώσουμε την δυνατότητα στον χρήστη να χειρίζεται την νοητή κάμερα έτσι ώστε να βλέπει τα δεδομένα από όποια μεριά επιθυμεί καθώς και άλλες επιλογές όπως κλείσιμο του παραθύρου. Εντολές που χρησιμοποιήθηκαν εδώ είναι οι exit, glRotatef, glTranslated και glutPostRedisplay.

```
void specialKeys(int key, int x, int y) {
	switch (key) {
	case GLUT_KEY_LEFT:
		cam.translateCamera(2);
		break;
	case GLUT_KEY_RIGHT:
		cam.translateCamera(3);
		break;
	case GLUT_KEY_UP:
		//VACANT
		break;
	case GLUT_KEY_DOWN:
		//VACANT
		break;
	default:
		break;
	}
}
```

2 void init:

Εδώ αρχικοποιούμε τα χρώματα της σκηνής και το βάθος της με τις εντολές glClearColor και glClearDepth.

```
void init() {

	GLint buf, sbuf;
	int i, j;

	glClearColor(0.0, 0.0, 0.0, 0.0);

	glShadeModel(GL_SMOOTH);

	glEnable(GL_DEPTH_TEST);
	glClearDepth(1.0);

	glEnable(GL_NORMALIZE);
	//glEnable(GL_COLOR_MATERIAL);
```

3 voidgenCube:

Με αυτή την μέθοδο αρχικοποιούμε το «υλικό» που θα αντιπροσωπεύει κάθε point όπου εισάγετε. Στην περίπτωση μας αυτό θα είναι ένας κύβος που θα δημιουργείται για κάθε point ξεχωριστά . Προκειμένου να επιτευχθεί αυτό είναι απαραίτητο για κάθε κύβο να εισάγονται οι συντεταγμένες x, y, z κάθε point και θα προσδιορίζουν την θέση του.

4 void Display:

Περιέχει μια εντολή καθαρισμού του προηγούμενου frameglClear ούτως ώστε κάθε φορά που ανανεώνετε η σκηνή να ξαναγράφονται τα δεδομένα από την αρχή. Σε αυτή την μέθοδο προσδιορίζουμε το είδος και τον τρόπο του φωτισμού της σκηνής χρησιμοποιώντας με την χρήση των εντολών newlight και glLightfv.Επίσης δημιουργούμε του κύβους με την μέθοδο genCube που εξηγήσαμε παραπάνω.

5 voidperspectiveGL:

Σε αυτό το σημείο, αρχικοποιούμε την προοπτική της σκηνή για κάθε αρχή του προγράμματος μας με την εντολή glFrustum.

6 voidreshape:

Είναι μέθοδος που αναδημιουργεί την σκηνή μετά από κάθε αλλαγή που καλούμε με την βοήθεια του πληκτρολογίου. Εδώ καλούμε και την perspectiveGL.

7 int main:

Είναι η κύρια μέθοδος του προγράμματος μας, όπου καλούμε όλες τις παραπάνω και χρησιμοποιούμε το Kinect. Αρχικά, πρέπει να χειριστούμε τον αισθητήρα και να πάρουμε τα δεδομένα του. Για να γίνει αυτό χρειαζόμαστε τις ακόλουθες εντολές:

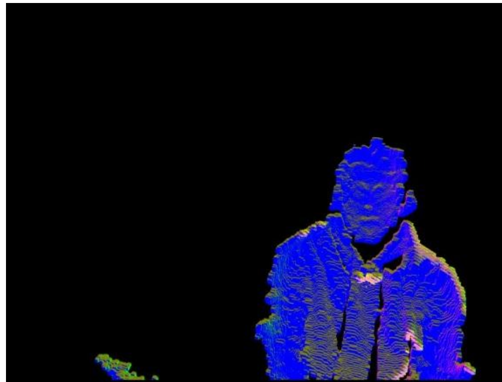
- initialize(): Αρχικοποιεί ένα κόμβο και διαβάζει δεδομένα.
  π.χ.: rc=initialize();
- open(): Ανοίγει τον την συσκευή.
  π.χ.: device.open();
- create(): Δημιουργεί της επιθυμητές ροές (βάθους ή χρώματος).
  π.χ.: depth.create(device,SENSOR_DEPTH);
- start(): Ξεκινά την διαδικασία εισροής δεδομένων από τον αισθητήρα.
  π.χ.: depth.start();
- readFrame(): Διαβάζει τα στιγμιότυπα δεδομένων.
  π.χ.: .: depth. readFrame();
- destroy():Μετά το τέλος της χρήσης ροής, την τερματίζουμε με αυτή την εντολή.
  π.χ.: depth.destroy();
- close(): Κλείνει την συσκευή.
  π.χ.: device.close();

Στο παρακάτω κομμάτι του κώδικα, βλέπουμε την διαδικασία που χρησιμοποιήσαμε για να γίνει η αποθήκευση των συντεταγμένων. Κάθε depth stream (ροή βάθους) περιέχει πληροφορίες σχετικά με κάθε ένα από τα point της. Για να μπορέσουμε να τις χρησιμοποιήσουμε, πρέπει να τις μετατρέψουμε σε κλίμακα την οποία μπορούμε να κατανοήσουμε. Γι αυτό το λόγο χρησιμοποιούμε την μέθοδο της OpenNICoordinateConverter::DepthToWorld(). Τις συντεταγμένες για κάθε διάσταση x, y, z, που θα εξάγουμε, τις περνάμε σε ένα πίνακα τύπου Vector, του οποίου η χωρητικότητα αλλάζει δυναμικά.

```
const DepthPixel* pDepth =
                static_cast<const DepthPixel*>(depthFrame.getData());
                cout << depthFrame.getDataSize()<< endl;
float px[depthFrame.getDataSize()], py[depthFrame.getDataSize()],
                pz[depthFrame.getDataSize()];

const DepthPixel* pDepthArray = NULL;
if (depth1.readFrame(&depthFrame) == STATUS_OK) {
        pDepthArray = (const DepthPixel*) depthFrame.getData();
        for (int y = 0; y < depthFrame.getHeight() - 1; ++y) {
                for (int x = 0; x < depthFrame.getWidth() - 1; ++x) {
                        int idx = x + y * depthFrame.getWidth();
                        const DepthPixel& rDepth = pDepthArray[idx];

                        CoordinateConverter::convertDepthToWorld(depth1, x, y, rDepth,
                                        &px[0], &py[0], &pz[0]);

                        myPoints.push_back(
                                        Points((px[0] / 1000)-0.5, (py[0] / 1000) + 1, pz[0] / 1000));
```

Τέλος, δίνουμε τις διαστάσεις, το όνομα και τις εντολές για την δημιουργία του παραθύρου όπου θα γίνει η προβολή όλων των αποτελεσμάτων που εισάγαμε παραπάνω.

```
glutInit(&argc, argv);
glutInitDisplayMode(
                GLUT_DOUBLE | GLUT_ALPHA | GLUT_RGB | GLUT_DEPTH | GLUT_MULTISAMPLE);
glutInitWindowSize(800, 600);
glutInitWindowPosition(100, 100);
glutCreateWindow("Kinect 3D Scanner");
init();
//glutIdleFunc(animations);
glutDisplayFunc(display);
glutSpecialFunc(specialKeys);
glutKeyboardFunc(keyboardFunction);
glutReshapeFunc(reshape);
glutMainLoop();
```

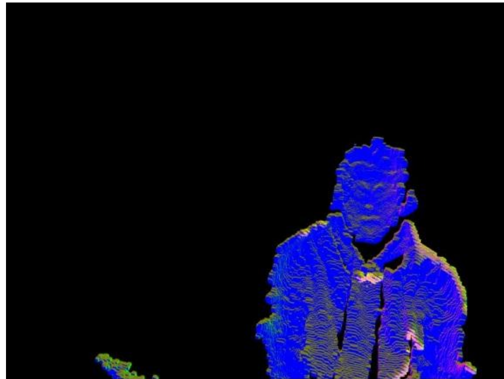


Εικόνα 35 – Αποτέλεσμα OpenNIimagereceiver

Συμπεράσματα:

Το αποτέλεσμα της παραπάνω προσπάθειας δεν ήταν το επιθυμητό. Η ποιότητα της εικόνας ήταν πολύ χαμηλή αλλά κυρίως, η διαδικασία της επεξεργασίας ήταν πολύ αργή αφού κάθε φορά που γινόταν η ανακατασκευή της εικόνας έπρεπε να γίνεται αναδημιουργία περίπου

300000 κύβων. Συνεπώς, εγκαταλείφθηκε αυτή η εκδοχή και στραφήκαμε σε πιο εξελιγμένες και αξιόπιστες λύσεις.



Εικόνα 36 – Αποτέλεσμα OpenNI image receiver

4.3 Χρήση πολλαπλών Kinect

Η αρχική ιδέα για τον τρόπο που θα γινόταν η σάρωση, ήταν να χρησιμοποιηθούν παραπάνω από ένας αισθητήρες. Στην διάθεση μας είχαμε έξι αλλά ήταν δύσκολο να χρησιμοποιηθούν όλα καθώς κάθε ένας πρέπει να χρησιμοποιεί εξολοκλήρου τον δίαυλο της θύρας USB.

Επιπλέον, οι υπέρυθρες κάμερες που είναι υπεύθυνες για τον υπολογισμό του βάθους που λαμβάνουν τα Kinect, δημιουργούσαν παρεμβολές μεταξύ τους· πράγμα που έκρινε αδύνατη την χρήση πολλών αισθητήρων ταυτόχρονα. Για αυτόν το λόγο, η αιχμαλώτιση των στιγμιότυπων ήταν απαραίτητο να γίνει ξεχωριστά για κάθε διαφορετικό αισθητήρα.

Προκειμένου να πετύχουμε τα παραπάνω διαμορφώσαμε την πειραματική εφαρμογή OpenNIGrabber (βλ.: 5.1), της PCLκάνοντας χρήση του αλγόριθμου ICP (βλ.: 3.3.1) για την συγχώνευση των στιγμιότυπων που είχαμε συλλέξει.

4.3.1 Δύο Αισθητήρες

Δύο ήταν τα ενδεχόμενα για την τοποθέτηση των αισθητήρων:

(a) Οι αισθητήρες τοποθετήθηκαν ο ένας κάτω από τον άλλο με τέτοιο τρόπο ώστε να καλύπτουν το πάνω και το κάτω μέρος του αντικειμένου. Με τον τρόπο αυτό θα έπρεπε να προσθέσουμε ένα επιπλέον αισθητήρα στις 180° σε σχέση με τους μπροστά ή να περιστρέψουμε το αντικείμενο πάλι 180° σε σχέση με την αρχική θέση, για να μπορέσουμε να καλύψουμε και το πίσω μέρος του και να έχουμε πλήρη εικόνα.

Εικόνα 37 – (a) Πείραμα με δύο Kinect

Εικόνα 38 – Παράδειγμα πειράματος (a)

(b) Οι αισθητήρες τοποθετήθηκαν μπροστά και πίσω (διαφορά 180°), με κλήση ανάλογη του μεγέθους, του εκάστοτε αντικειμένου.

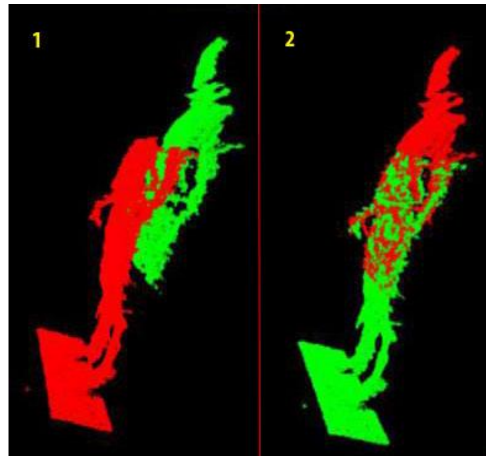
Εικόνα 39–(b) Πειραμα με δύο Kinect

Χρησιμοποιήθηκαν το ICP και το transform Point-Cloud και στις δύο περιπτώσεις, για να συγκριθούν και να τοποθετηθούν κατάλληλα μεταξύ τους.

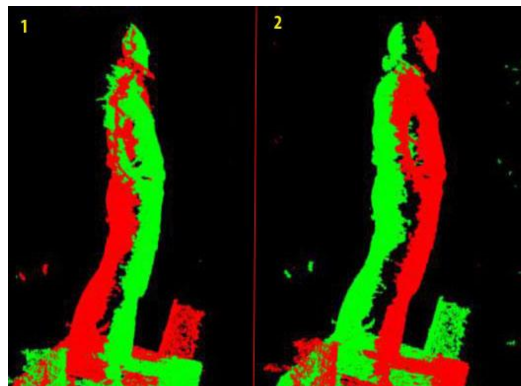
Συμπεράσματα:

Στη περίπτωση (b) το αποτέλεσμα ήταν αρκετά ικανοποιητικό από την άποψη της τοποθέτησης των δύο clouds μεταξύ τους, η οποία ήταν σωστή αλλά χανόταν αρκετή πληροφορία στα άκρα, όπου σε αντικείμενα με μεγαλύτερο όγκο, θα οδηγούσε σε σύγχυση

της ανίχνευσης του ICP. Επίσης, κενά που πιθανόν να υπάρχουν μεταξύ των clouds, δημιουργούν προβλήματα στην αναδημιουργία του 3D πλέγματος της επιφάνειας. Παρόμοια προβλήματα είχαμε και στην περίπτωση (a). Παρόλο που στο μπροστινό μέρος το αποτέλεσμα ήταν ως επί το πλείστον καλό, παρόμοια προβλήματα με τη (b) εμφανίστηκαν, όταν τελικά σαρώσαμε την αντίθετη πλευρά του αντικειμένου.



Εικόνα 40 - Αποτέλεσμα πειράματος (a)

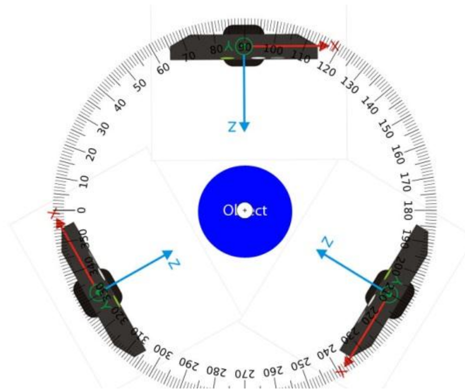


Εικόνα 41 - Αποτέλεσμα πειράματος (b)

4.3.2 Τρείς Αισθητήρες

Τρείς αισθητήρες τοποθετήθηκαν στις 120° γύρω από το αντικείμενο μας (βλ.: Εικόνα 41) προκειμένου το άνοιγμα της κάμερας βάθους του καθενός να καλύπτει λίγο παραπάνω από το 1/3 της επιφάνειας του αντικειμένου. Ο σκοπός ήταν τα άκρα του κάθε cloud να είναι παρόμοια (overlapping) με αυτά που θα παίρναμε από τους διπλανούς αισθητήρες για να μπορεί ο ICP να κάνει την σύγκριση μεταξύ τους. Τέλος, χρησιμοποιώντας τον πίνακα που

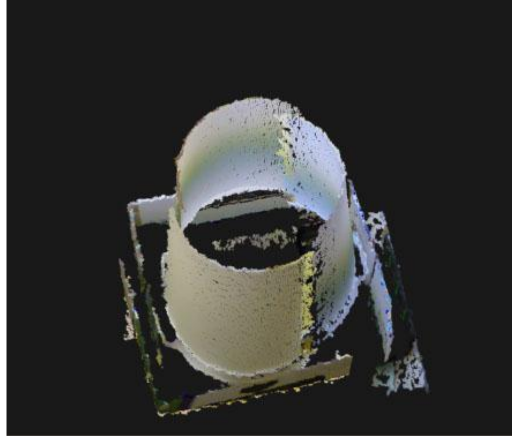
εξάγετε από την ICP στην κλάση transform Point-Cloud, μπορούσαμε να ελέγξουμε το αποτέλεσμα.



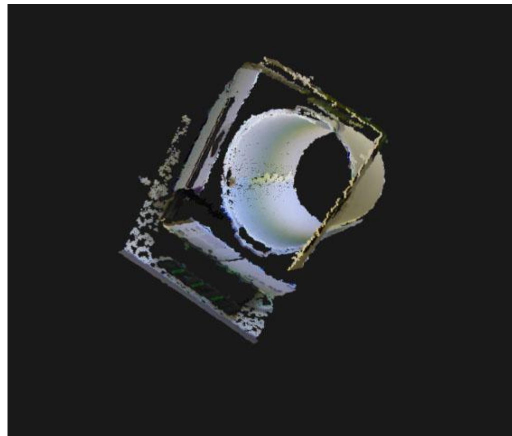
Εικόνα 42 - Θέσεις τριών Kinect

Συμπεράσματα:

Παρόλο που το αποτέλεσμα ήταν πολύ κοντά στο στόχο, περιείχε πολύ θόρυβο και η συγκέντρωση των cloudsαπό τον ICP δεν ήταν ακριβής. Κάτι τέτοιο θα ήταν κρίσιμο σε μεταγενέστερο στάδιο, όταν θα θέλαμε να αναδημιουργήσουμε την επιφάνεια με το 3Dπλέγμα.



Εικόνα 43 - Αποτέλεσμα πειράματος τριών Kinect



Εικόνα 44 - Αποτέλεσμα πειράματος τριών Kinect

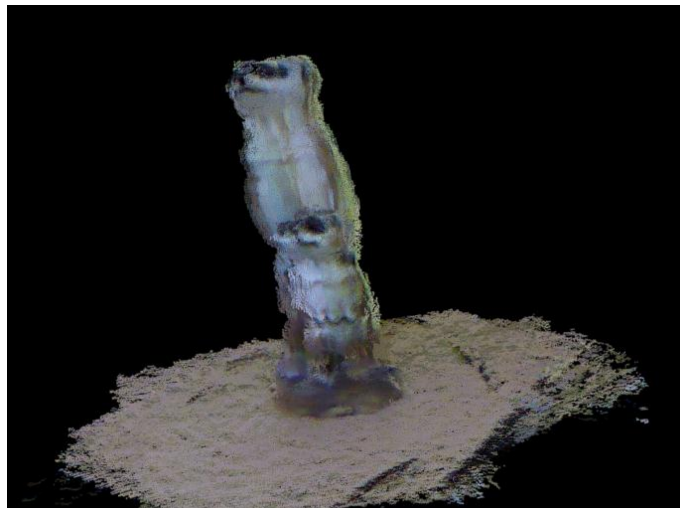
4.4 Σταθερός αισθητήρας με περιστρεφόμενο αντικείμενο

Έχοντας απορρίψει την χρήση πολλαπλών αισθητήρων για τους λόγους και τα προβλήματα που αναφέρθηκαν παραπάνω, δοκιμάστηκε μια πιο απλή λύση που βασίζεται στην λογική των περισσοτέρων 3D Scanners που υπάρχουν μέχρι στιγμής. Η λογική αυτή υποδεικνύει ότι είτε το αντικείμενο πρέπει να είναι σταθερό και ο αισθητήρας να περιστρέφεται γύρω του ή το αντικείμενο να περιστρέφεται γύρω από τον εαυτό του μπροστά από τον αισθητήρα. Η πρώτη

περίπτωση απορρίφθηκε γρήγορα καθώς απαιτεί μεγάλη ακρίβεια από τον αισθητήρα και ένα δυνατό υπολογιστικό σύστημα που θα μπορεί να επεξεργαστεί γρήγορα το μεγάλο όγκο δεδομένων που θα παράγονται. Στην περίπτωση μας, ούτε το Kinect είναι πολύ ακριβής σαν αισθητήρας αλλά και οι δυνατότητες του υπολογιστή που χρησιμοποιούνταν ήταν σχετικά περιορισμένες.
Συνεπώς, καταλήξαμε στην δεύτερη λύση· με το αντικείμενο να περιστρέφεται γύρω από τον αισθητήρα.

Ως πρώτο βήμα, τοποθετήσαμε το Kinect σε ένα σταθερό σημείο και περιστρέφαμε το αντικείμενο χειροκίνητα και χωρίς μεγάλη ακρίβεια. Τα αποτελέσματα ήταν ενθαρρυντικά οπότε προχωρήσαμε στην κατασκευή της περιστρεφόμενης βάσης 360° (βλ.: 2.4) η οποία μας έδωσε την δυνατότητα σταθερότερης εναλλαγής της εκάστοτε θέσης του αντικειμένου.

Με αυτόν τον τρόπο ήταν δυνατός ο συνδυασμός των παραπάνω πειραμάτων με τελική κατάληξη στην τεχνική του overlapping (επικάλυψης) που είχε δοκιμαστεί στην περίπτωση των τριών Kinect. Επιπλέον, χρησιμοποιώντας πολλά συνεχόμενα και περιστροφικά στιγμιότυπα τουαντικειμένου, 360° μοιρών, παρατηρήθηκε ότι ήταν ευκολότερο για το ICP να εντοπίσει τα κατάλληλα σημεία ένωσης, να τα συγκρίνει και να αποδώσει πιο ρεαλιστικό αποτέλεσμα. Επιπλέον, λειτούργησε πιο αποτελεσματικά διότι οι παρεμβολές μειώθηκαν, εφόσον μειώθηκαν οι αισθητήρες.



Εικόνα 45 - Πείραμα με περιστρεφόμενη Βάση (πολλαπλά στιγμιότυπα)

5 Εφαρμογή

Η εφαρμογή μας χωρίζεται σε τρία κύρια σημεία. Το πρώτο αφορά την χρήση του Kinect σαν είσοδο για τα δεδομένα βάθους που χρειαζόμαστε και την μετατροπή τους σε point-cloud για μετέπειτα χρήση. Το δεύτερο είναι το βασικό κομμάτι της εργασίας όπου αναπτύσσονται οι τεχνικές σύγκρισης των επιμέρους point-clouds μαζί με όλες οι διαδικασίες φιλτραρίσματος τους. Στο τρίτο και τελευταίο κομμάτι, γίνεται το τελικό φιλτράρισμα, η συγχώνευση και η αναδημιουργία της επιφάνειας τους σαν ένα ενιαίο τρισδιάστατο αντικείμενο.

5.1 Μέρος Πρώτο- OpenNI Grabber

Το OpenNI Grabber είναι ένα σημαντικό κομμάτι της βιβλιοθήκης PCL που λειτουργεί ως διάμεσος μεταξύ του υπολογιστή και των αισθητήρων-καμερών που είναι συμβατοί με την OpenNI. Προσφέρει δυνατότητες χρήσης των ροών δεδομένων από τις συσκευές μετατρέποντας τες παράλληλα σε point-clouds ανεξαρτήτως μορφής.

Μορφές ροών στην PCL:

- Ασπρόμαυρο: pcl:PointCloud<XYZ>
- Έγχρωμο: pcl:PointCloud<XYZRGB>

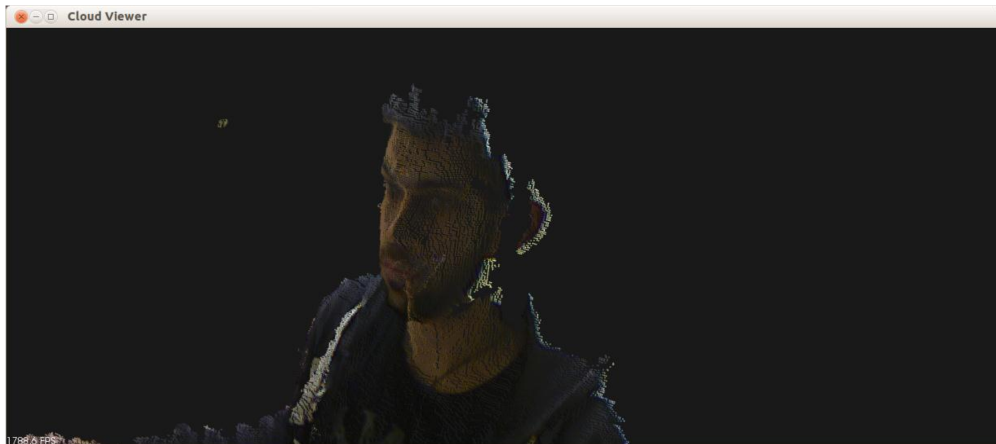
Με τα παραπάνω σαν οδηγό, δημιουργήσαμε το πρώτο κομμάτι της εφαρμογής, το οποίο λειτουργεί σαν διεπαφή με τον χρήστη αλλά κυρίως προσφέρει επικοινωνία μεταξύ συσκευής και υπολογιστή.

Τα points που εισάγονται από τον αισθητήρα, όπως προαναφέρθηκαν στο κεφάλαιο 3, αντιπροσωπεύουν τις συντεταγμένες X, Y, Z μιας επιφάνειας. Ο grabber είναι αρμόδιος να τα μετατρέψει σε συγκροτημένα point-clouds σε οποιαδήποτε από τις παραπάνω μορφές χρειαζόμαστε. Στην δική μας περίπτωση δεν χρησιμοποιούμε το χρώμα, οπότε έχουμε Point-Cloud<XYZ>.

5.1.1 Παράθυρο-Διαλόγου

Ένα πολύ σημαντικό κομμάτι αυτού το μέρους, είναι η αλληλεπίδραση του χρήστη με την εφαρμογή. Υλοποιείται ένα παράθυρο διαλόγου στο οποίο απεικονίζεται το point-cloud και στο οποίο ο χρήστης μπορεί να αλλάξει την θέση του, χρησιμοποιώντας το ποντίκι, το

πληκτρολόγιο ή την οθόνη αφής. Επιπλέον, με συγκεκριμένα κουμπιά του πληκτρολογίου μπορεί να αλλάξει η θέση, το χρώμα, η πυκνότητα κ.α.



Εικόνα 46 – Παράθυρο διαλόγου OpenNIGrabber

Κώδικας δημιουργίας παράθυρου-διαλόγου:

Αρχικοποίηση παραθύρου-διαλόγου.

```
pcl::Grabber* interface = new pcl::OpenNIGrabber();
```

Γέμισμα παραθύρου με το pointcloud.

```
boost::function<void (const pcl::PointCloud<pcl::PointXYZ>::ConstPtr&)> f =
  boost::bind (&SimpleOpenNIViewer::cloud_cb_, this, _1);

interface->registerCallback (f);
```

Ξεκίνημα καταγραφείς των δεδομένο στο παράθυρο-διαλόγου.

```
interface->start ();
```

Τερματισμός.

```
interface->stop ();
```

5.1.2 Φιλτράρισμα και Αποθήκευση

Φιλτράρισμα

Λαμβάνοντας υπόψη ότι ο αισθητήρας Kinect λειτουργεί καλύτερα από τα 0,8μ έως τα 2,5μ και ότι το άνοιγμα του σε ύψος είναι 45° έπρεπε να περιορίσουμε την εμβέλεια στην οποία

κατέγραφε δεδομένα. Για αυτό τον λόγο χρησιμοποιήθηκε το φίλτρο Pass-Through (βλ.:3.3.1).

Αντιμετωπίζοντας έλλειψη χώρου και για να αποκτήσουμε την μέγιστη ακρίβεια δημιουργήσαμε μέσω του Pass-Through ένα νοητό τρισδιάστατο κουτί αλλάζοντας κατάλληλα τον κώδικα. Οι διαστάσεις του «κουτιού» αλλάζουν ανάλογα με το είδος του αντικειμένου καθώς και το μέγεθος ή την θέση που θέλαμε να έχουμε.

Άνθρωπος:

- Καθιστός: Ύψος έως 75εκ, Πλάτος 1μ και Μήκος 1,2μ έως 1,7μ.
- Όρθιος: Ύψος έως 2μ, Πλάτος 1μ και Μήκος έως 1,5μ.

Αντικείμενα:

Χρησιμοποιώντας την βάση πάντα σε προκαθορισμένο σημείο, είχαμε : Ύψος 30εκ έως 50εκ, Πλάτος 70εκ και Μήκος 1,2μ έως 1,5μ.

Αποθήκευση

Η αποθήκευση γίνεται αυτόματα λίγο μετά το ξεκίνημα της ροής των δεδομένων από το Kinect. Κάτι που πετύχαμε προσθέτοντας ένα βρόγχο στην κύρια κλάση της εφαρμογής, ο οποίος λειτουργεί όσο τρέχει το παράθυρο-διαλόγου. Στον βρόγχο αυτό καλούμε την functionsaveScreen που θα αναλύσουμε παρακάτω και έναν μετρητή που χρησιμοποιείται για να υπάρχει μια μικρή καθυστέρηση μετά από κάθε αποθήκευση στιγμιότυπου.

```
if (!viewer.wasStopped())
     viewer.showCloud
        (cloud_filtered3);



     timerH++;

  if (timerH >= 10) {
saveScreen(cloud_filtered3);
        timerH = 0;
```

voidsaveScreen:

Είναι μια λειτουργία (function) η οποία παίρνει ως είσοδο έναν δείκτη στην θέση που αποθηκεύεται το αρχικό point-cloud. Αρχικοποιούμε την θέση αποθήκευσης σε μια μεταβλητή γραμματοσειράς η οποία θα χρησιμοποιηθεί μαζί με το point-cloud από την εντολή

savePCD. Τέλος, έχοντας ορίσει ένα μετρητή, έχουμε την δυνατότητα να απαριθμούμε κάθε στιγμιότυπο ξεχωριστά, με έναν αύξοντα αριθμό το οποίο θα είναι χρήσιμο στα επόμενα κομμάτια της εφαρμογής.

```
void saveScreen(pcl::PointCloud<pcl::PointXYZRGBA>::Ptr
cloud2) {


      stringstream stream;
      stream << "/home/psistakis/Desktop/Human/human" <<
      filesSaved
                 << ".pcd";
      string filename = stream.str();
      if (io::savePCDFile(filename, *cloud2, true) == 0) {
            filesSaved++;
            cout << "Saved " << filename << "." << endl;
      } else
            PCL_ERROR("Problem saving %s.\n", filename.c_str
            ());

}
```

5.2 Κύρια Εφαρμογή

Σε αυτό το κομμάτι της εφαρμογής γίνεται η βασική επεξεργασία των στιγμιότυπων που έχουμε συλλέξει από τον OpenNI Grabber.

Κάθε point-cloud που έχει αποθηκευτεί, έχει κατεύθυνση προς το (0,0,0) σημείο της σκηνής, δηλαδή το σημείο στον 3D χώρο, όπου βρίσκεται ο αισθητήρας. Αυτό σημαίνει ότι για να μπορέσουμε να αναδημιουργήσουμε το αντικείμενο μας εικονικά, θα πρέπει να αλλάξουμε την πόσα κάθε στιγμιότυπου εκτός του αρχικού. Για να το επιτύχουμε, χρησιμοποιηθήκαν οι περισσότερες τεχνικές που προαναφέρθηκαν στο κεφάλαιο 3.

Παρακάτω θα αναλυθούν οι τεχνικές και ο μέρος του κώδικα που υλοποιήθηκε.

5.2.1 computeTransformation

Η functioncumputeTransformation συγκρίνει και υπολογίζει την θέση που θα πρέπει να έχει το ένα από τα δύο γειτονικά point-clouds για να είναι όσο το δυνατόν πιο κοντά.

Έχει ως είσοδο δυο δείκτες (pointers) που ανήκουν στα δυο συγκρινόμενα point-cloudsκαι άλλον ένα που αφορά ένα πινάκα float 4 διαστάσεων (matrix4f). Ο πίνακας αυτός θα περιέχει τις απαραίτητες συντεταγμένες για να γίνει η αλλαγή πόζας στο δεύτερο point-cloud.

Οι λειτουργίες (functions) που χρησιμοποιήθηκαν:

estimateKeyPoints:

Είναι ένας ανιχνευτής Key-points (βλ.: 3.3.1) που συγκρίνει τα δύο εισερχόμενα point-cloudsγια κοινά σημεία. Για να επιτευχτεί αυτό αποτελεσματικά ένας ανιχνευτής key-pointsθα πρέπει να έχει τις ακόλουθες ιδιότητες:

- Μοναδικότητα: Συνήθως, μόνο ένα μικρό υποσύνολο σημείων στην σκηνή είναι key-point.
- Επαναληψιμότητα: Εάν ένα σημείο είναι key-pointστο ένα point-cloud, πρέπει να βρεθεί στην αντίστοιχη θέση στο άλλο (σταθερό σημείο).
- Διακριτικότητα: Η περιοχή γύρο από τα key-pointsπρέπει να έχουν ένα μοναδικό σχήμα ή εμφάνιση που να μπορούν να αναγνωρίζονται από τους featuredescriptors.

Κώδικας:

Καλώντας την estimateKeypointsέχει ως είσοδο τα δύο συγκρινόμενα point-clouds (srcκαι tgt) και ως έξοδο τα key-pointsτους.

```
estimateKeypoints(src, tgt, keypoints_src, keypoints_tgt);
```

Για να συλλέξουμε τα απαιτούμενα key-pointsχρησιμοποιήσαμε την κλάση UniformSampling.

Η UniformSampling συγκεντρώνει ένα τοπικό 3D πλέγμα πάνω από ένα δεδομένο point-cloud, και μειώνει τη δειγματοληψία φιλτράροντας τα δεδομένα. Δημιουργεί ένα 3Dπλέγμα voxelσαν ένα σύνολο από μικροσκοπικά 3D κουτιά, πάνω από το εισαγόμενο cloudδεδομένων. Στην συνέχεια, για κάθε voxel (δηλαδή 3D box), όλα τα αναπαριστώμενα σημεία θα προσεγγιστούν ανάλογα με το κέντρο βάρους τους.

Επειδή το αποτέλεσμα που προέρχεται από την UniformSampling δεν είναι point αλλά συντεταγμένες, είναι απαραίτητη η χρήση της CopyPointCloudγια να τα μετατρέψουμε σε μορφή αναγνωρίσιμη από την εφαρμογή μας.

```
void estimateKeypoints(const PointCloud<PointXYZ>::Ptr &src,
                const PointCloud<PointXYZ>::Ptr &tgt,
                PointCloud<PointXYZ>::Ptr &keypoints_src,
                PointCloud<PointXYZ>::Ptr &keypoints_tgt) {
        PointCloud<int> keypoints_src_idx, keypoints_tgt_idx;
        // Get an uniform grid of keypoints
        UniformSampling<PointXYZ> uniform;
        uniform.setRadiusSearch(0.05); // 1m
        uniform.setInputCloud(src);
        uniform.compute(keypoints_src_idx);
        copyPointCloud<PointXYZ, PointXYZ>(*src, keypoints_src_idx.points,
                        *keypoints_src);

        uniform.setInputCloud(tgt);
        uniform.compute(keypoints_tgt_idx);
        copyPointCloud<PointXYZ, PointXYZ>(*tgt, keypoints_tgt_idx.points,
                        *keypoints_tgt);
}
```

estimateNormals

Αυτή η λειτουργία υπολογίζει τα Normals (βλ.: 3.3.1)του point-cloud χρησιμοποιώντας ως είσοδο τα key-pointsπου εξήχθησαν από την estimateKeyPoints.

<u>Κώδικας</u>:

```
//---- Normal Estiation
estimateNormals(src, tgt, *normals_src, *normals_tgt);
```

```
void estimateNormals(const PointCloud<PointXYZ>::Ptr &src,
                const PointCloud<PointXYZ>::Ptr &tgt, PointCloud<Normal> &normals_src,
                PointCloud<Normal> &normals_tgt) {
        NormalEstimation<PointXYZ, Normal> normal_est;
        normal_est.setInputCloud(src);
        normal_est.setRadiusSearch(0.015); //0.01// 50cm
        normal_est.compute(normals_src);
        normal_est.setInputCloud(tgt);
        normal_est.compute(normals_tgt);
```

estimateFPFH

Με την estimateFPFH, υπολογίζονται τα Features (βλ.: 3.3.1) έχοντας ως είσοδο τα αρχικά point-clouds, τα normal και τα key-pointsπου εξήχθησαν με τις προηγούμενες μεθόδους.

<u>Κώδικας</u>:

```
//----Estimation of FPFH

estimateFPFH(src, tgt, normals_src, normals_tgt, keypoints_src,
                keypoints_tgt, *fpfhs_src, *fpfhs_tgt);
```

```
void estimateFPFH(const PointCloud<PointXYZ>::Ptr &src,
                const PointCloud<PointXYZ>::Ptr &tgt,
                const PointCloud<Normal>::Ptr &normals_src,
                const PointCloud<Normal>::Ptr &normals_tgt,
                const PointCloud<PointXYZ>::Ptr &keypoints_src,
                const PointCloud<PointXYZ>::Ptr &keypoints_tgt,
                PointCloud<FPFHSignature33> &fpfhs_src,
                PointCloud<FPFHSignature33> &fpfhs_tgt) {
        FPFHEstimation<PointXYZ, Normal, FPFHSignature33> fpfh_est;
        fpfh_est.setInputCloud(keypoints_src);
        fpfh_est.setInputNormals(normals_src);
        fpfh_est.setRadiusSearch(0.5); //0.15 // 1m // has to be larger than normal estimation!!!
        fpfh_est.setSearchSurface(src);
        fpfh_est.compute(fpfhs_src);
        fpfh_est.setInputCloud(keypoints_tgt);
        fpfh_est.setInputNormals(normals_tgt);
        fpfh_est.setSearchSurface(tgt);
        fpfh_est.compute(fpfhs_tgt);
```

findCorrespondence

Βρίσκει αλληλοσυνδεόμενα σημεία (βλ.: 3.3.1) μεταξύ των δύο point-clouds χρησιμοποιώντας τα features που υπολογίστηκαν από την estimationFPFH.

Κώδικας:

```
findCorrespondences(fpfhs_src, fpfhs_tgt, *all_correspondences);
cout << "Correspond" << endl;
```

```
void findCorrespondences(const PointCloud<FPFHSignature33>::Ptr &fpfhs_src,
                const PointCloud<FPFHSignature33>::Ptr &fpfhs_tgt,
                Correspondences &all_correspondences) {
        CorrespondenceEstimation<FPFHSignature33, FPFHSignature33> est;
        est.setInputSource(fpfhs_src);
        est.setInputTarget(fpfhs_tgt);
        est.determineReciprocalCorrespondences(all_correspondences, 0.5f);
        est.determineCorrespondences(all_correspondences);
//      for(int i=0;i<all_correspondences.size();i++){
//              cout << "TEST " << all_correspondences[i] << endl;
//              //vector
//      }
}
```

rejectBadCorrespondences

Φιλτράρει και απορρίπτει τα κακά σημεία που εξήχθησαν από την findCorrespondence (βλ.: 3.3.1)βελτιώνοντας αισθητά την ακρίβεια τους.

Κώδικας:

```
//----Bad Correspondence Rejection
rejectBadCorrespondences(all_correspondences, keypoints_src, keypoints_tgt,
                *good_correspondences);
cout << "Rejection" << endl;
```

```
void rejectBadCorrespondences(const CorrespondencesPtr &all_correspondences,
                const PointCloud<PointXYZ>::Ptr &keypoints_src,
                const PointCloud<PointXYZ>::Ptr &keypoints_tgt,
                Correspondences &remaining_correspondences) {

        CorrespondenceRejectorSampleConsensus<PointXYZ> rej;
        rej.setInputSource(keypoints_src);
        rej.setInputTarget(keypoints_tgt);
        rej.setInlierThreshold(0.04f); //0.04 (40+clouds)//0.03 working (120+ clouds)
        rej.setMaximumIterations(60);
        rej.setRefineModel(true);
        rej.setInputCorrespondences(all_correspondences);
        rej.getCorrespondences(remaining_correspondences);

}
```

estimateRigridTransformation

Παίρνει ως είσοδο τα key-pointsκαι τα κοινά σημεία από το rejectBadCorrespondences για να υπολογίσει την τελική θέση. Το αποτέλεσμα θα περαστεί στον matrix4fπου προαναφέραμε και θα χρησιμοποιηθεί μετέπειτα στην εφαρμογή για να αλλαχτεί η θέση του point-cloud.

```
trans_est.estimateRigidTransformation(*keypoints_src, *keypoints_tgt,
                *good_correspondences, transform);
cout << "Estimate transformation" << endl;
```

5.2.2 Main

Η main είναι η κύρια μέθοδος της εφαρμογής, στην οποία περιέχονται η computeTransformation (βλ.: 5.2.1) και η αναδημιουργία της επιφάνειας που θα αναλυθεί παρακάτω. Η μέθοδος αυτή χωρίζετε σε τρις φάσεις:

Φάση 1η: Δημιουργήθηκε ένας βρόγχος ο οποίος θα διατρέχει τα στιγμιότυπα που εισήχθησαν από τον OpenNI Grabber (βλ.: 5.1).Σε αυτό τον βρόγχο (**for**) φορτώνουμε τα στιγμιότυπα από το πρώτο έως το τελευταίο προσπελάζοντας και συγκρίνοντας τα, ένα προς ένα. Η προσπέλαση, γίνεται με την μέθοδο **loadPCDFile** στην οποία περνάμε με γραμματοσειρά (stringstream), προσδιορίζοντας την θέση από όπου θα ανακτηθούν από το σκληρό δίσκο. Επιπλέον, περιέχει συνθήκες οι οποίες εξυπηρετούν στην αλλαγή των ιδιοτήτων τους, ανάλογα με την σειρά που έχουν σαρωθεί.

Πιο συγκεκριμένα τα στιγμιότυπα χωρίζονται σε «πηγαία» και «στοχευόμενα». Τα πηγαία είναι αυτά που εντέλει θα μετασχηματιστούν. Το πρώτο στιγμιότυπο στην σειρά, έχει πάντα σταθερή θέση αφού είναι το σημείο αναφοράς για τα υπόλοιπα, τα οποία το θεωρούν ως την αρχική. Η μόνη αλλαγή πριν την αποθήκευση του, είναι η μείωση των δειγμάτων του και η κράτηση του πίνακα που περιέχει τις συντεταγμένες του, στην μνήμη. Μετά τον πρώτο γύρο του βρόγχου, εκτός από την μείωση των δειγμάτων, των δύο νέων στιγμιότυπων, μηδενίζεται

η θέση τους στο 3D χώρο. Με αυτόν τον τρόπο η μόνη διαφορά μεταξύ τους είναι τα 3D pixels που διαφοροποιούνται λόγο της πόζας τους στον αισθητήρα κατά την σάρωση. Αυτό γίνεται για να μπορεί η computeTransformation να κάνει την σύγκριση σε μικρότερο χώρο, προσφέροντας μεγαλύτερη ακρίβεια.

```
for (int i = 0; i < l; i++) {
        cout << "Start" << endl;

        stringstream Source;
        stringstream Target;
        stringstream fileName3;
        stringstream fileName4;

        if (i == 0) {
                Target << "/home/psistakis/Desktop/Human/human" << i << ".pcd";
                Source << "/home/psistakis/Desktop/Human/human" << i + 1 << ".pcd";
        } else {

                Target << "/home/psistakis/Desktop/Human/transformed2/human"
                                << i - 1 << ".pcd";
                Source << "/home/psistakis/Desktop/Human/human" << i << ".pcd";
        }
        cout << Source.str() << ";" << Target.str() << endl;

        loadPCDFile(Source.str(), *src);
        cout << "load file " << i << " Size: " << src->size() << endl;
        loadPCDFile(Target.str(), *tgt);
        cout << "load file " << i + 1 << " Size: " << tgt->size() << endl;

        PointCloud<PointXYZ>::Ptr source_mls(new PointCloud<PointXYZ>);
        PointCloud<PointXYZ>::Ptr target_mls(new PointCloud<PointXYZ>);

        Eigen::Matrix4f storedTransform;
        storedTransform << 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0;

        if (i == 0) {
                downsampling(tgt, tgt_down, 0.01);
                downsampling(src, src_down, 0.01);
                storedTransform << 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1;
                tgt_down = tgt;
                fileName4 << "/home/psistakis/Desktop/Human/transformed2/human" << i
                                << ".pcd";
                io::savePCDFile(fileName4.str(), *tgt_down);
        }
        else {
                tgt_down = tgt;
                storedTransform = myTransforms.back();

                transformPointCloud(*src, *src, storedTransform);

                downsampling(src, src_down, 0.01);
```

Μετά τις παραπάνω τροποποιήσεις, τα ζευγάρια στιγμιότυπων περνούν στην μέθοδο computeTransformation για να γίνει η σύγκριση μεταξύ τους και να εξάγουμε τον πίνακα συντεταγμένων που θα χρησιμοποιηθεί αργότερα.

```
// Compute the best transformtion
Eigen::Matrix4f transform;
transform << 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0;
computeTransformation(src_down, tgt_down, transform);
```

Εν συνεχεία, τα δύο στιγμιότυπα θα περάσουν από το ICP (βλ.: 3.1.1) το οποίο θα κάνει μια πιο αναλυτική σύγκριση και θα εξάγει έναν νέο πίνακα συντεταγμένων (matrix4f). Ο πίνακας αυτός θα πολλαπλασιαστεί με αυτόν από την compouteTransformation για να δημιουργηθεί ένας τρίτος ο οποίος θα περιέχει τα συνδυαστικά αποτελέσματα και των δύο για να γίνει, ο τελικός μετασχηματισμός του «πηγαίου» για αυτήν τη φάση, από την transformPointCloud (βλ.: 3.3.1) και η προσωρινή αποθήκευσή του.

```
Eigen::Matrix4f icp_transformation;

//      ---- ICP the 2 downsammpled clouds and save the transformation (icp_transformation)
icp(src_down, tgt_down, icp_transformation, ptr_src_icp);
//The tranformation happens to src
src_icp = *ptr_src_icp;

combinedTransform = icp_transformation * transform;
```

Φάση 2η: Αποτελείται από την ELCH και την LUM (βλ.: 3.1.1) οι οποίες ουσιαστικά διορθώνουν τις μικρές ασυμμετρίες που δημιουργούνται στο μετασχηματισμό του συνόλου των στιγμιότυπων που επεξεργάστηκαν παραπάνω. Οι απώλειες αυτές, είναι ουσιαστικά μερικά χιλιοστά αλλά λόγο της μεγάλης ποσότητας στιγμιότυπων, επιφέρουν παραμορφώσεις όταν εμφανιστούν όλα μαζί.

Ο ELCH ξεκινάει πρώτος χρησιμοποιώντας έναν βρόγχο για να ανατρέξει όλα τα στιγμιότυπα της πρώτης φάσης και αφού τελειώσει την διαδικασία αποθηκεύει τις συντεταγμένες κάθε αλλαγμένου στιγμιότυπου σε ένα πίνακα τύπου Vector για να μπορούν να χρησιμοποιηθούν από το LUM.

```
for (int i = 0; i < l; i++) {

        stringstream loadELCH;
        PointCloud<PointXYZ>::Ptr index(new PointCloud<PointXYZ>);
        loadELCH << "/home/psistakis/Desktop/Human/transformed2/test2/human"
                        << i << ".pcd";
        io::loadPCDFile(loadELCH.str(), *index);
        cloudVector.push_back(index);

        //Filled up the ELCH loop with Original Clouds
        elch.addPointCloud(cloudVector[i]);
}

//The open-close loop of ELCH
elch.setLoopStart(0);
elch.setLoopEnd(l - 1);

for (int t = 0; t < 5; t++) {
        cout << "ELCH iterations: " << t << endl;
        elch.compute();

}

for (int k = 0; k < cloudVector.size(); k++) {

        stringstream SaveELCH;
        SaveELCH << "/home/psistakis/Desktop/Human/transformed2/ELCH/human" << k
                        << ".pcd";
        savePCDFile(SaveELCH.str(), *(cloudVector[k]));
        cout << elch.getLoopTransform() << "'" << endl;
}

cout << "ELCH Done " << endl << "Vector Size: " << cloudVector.size()
                << endl;
```

Τέλος, όταν ο LUM τελειώσει με την σειρά του, περνάει τις συντεταγμένες στην transformPointCloud για να γίνει ο τελικός μετασχηματισμός, χρησιμοποιώντας τα αρχικά στιγμιότυπα. Αυτό γίνεται για να περαστούν όσο το δυνατών με περισσότερα 3D pixels, στην αναδημιουργία της επιφάνειας, προσφέροντας πιο ποιοτικό αποτέλεσμα.

```
//Apply the LUM transformation to original clouds
for (int v = 0; v < l; v++) {
        cout << v << endl;
        PointCloud<PointXYZ>::Ptr tempOrcl(new PointCloud<PointXYZ>);
        transformPointCloud(*cloudVector[v], *tempOrcl, lumTransforms[v]);
        cloudVector[v] = tempOrcl;
}

for (int i = 0; i < lum.getNumVertices(); i++) {
        cout << "Lum Save" << endl;
        stringstream SaveLUM;
        SaveLUM << "/home/psistakis/Desktop/Human/transformed2/LUM/human" << i
                        << ".pcd";
        savePCDFile(SaveLUM.str(), *(cloudVector[i]));
}
```

5.3 Αναδημιουργία επιφάνειας

Έχοντας αναλύσει, φιλτράρει και αποθηκεύσει επεξεργασμένα point-cloudsαπό την κύρια εφαρμογή, περνάμε στην τελική φάση. Εδώ θα ανακτηθούν ήδη διαμορφωμένα αποτελέσματα και θα γίνει επιπλέον φιλτράρισμα και υποδειγματολήψια πριν περάσουμε τα δεδομένα από την

διαδικασία της αναδημιουργίας επιφάνειας της. Για την αναδημιουργία χρησιμοποιήθηκε η κλάση Poisson, που είναι μέρος της PCL (βλ.: 3.1.1) εφαρμόζοντας την πολυγωνική μέθοδο (Polygon-mesh).

Ανάκτηση δεδομένων

Για να ανακτηθούν τα δεδομένα από τον χώρο αποθήκευσης ακολουθείται περίπου η ίδια διαδικασία με το κύριο μέρος της εφαρμογής (βλ.: 5.2.2) και την χρήση της **loadPCDfile**. Η μόνη διαφορά που υπάρχει, είναι μια μεταβλητή που μας δίνει την δυνατότητα να αλλάζουμε το βήμα του βρόγχου ώστε να ελέγχουμε την ποσότητα των στιγμιότυπων που θα περάσουμε στην Poisson. Αυτό γίνεται για πρακτικούς λόγος αφού τα στιγμιότυπα απεικονίζουν το ίδιο αντικείμενο με μικρή διαφορά πόζας μεταξύ τους. Άρα έχοντας είδη μορφοποιηθεί η πόζα τους, δεν υπάρχει ανάγκη να συμπεριληφθούν όλα. Με αυτόν τον τρόπο μειώσαμε σημαντικά το χρόνο προσπέλασης των στοιχείων από την μνήμη αλλά και της επεξεργασίας του από την Poisson.

Κώδικας:

```
for (int i = 0; i < l; i += step) {

        stringstream load1;
        stringstream load2;
        stringstream saveConcat;

        if (i == 0) {
                load1 << "/home/psistakis/Desktop/Human/transformed2/LUM/human" << i
                                << ".pcd";
                load2 << "/home/psistakis/Desktop/Human/transformed2/LUM/human"
                                << i + step << ".pcd";
        } else {

                load1
                                << "/home/psistakis/Desktop/Human/transformed2/results/concatHuman.pcd";
                load2 << "/home/psistakis/Desktop/Human/transformed2/LUM/human"
                                << i + step << ".pcd";
        }

        loadPCDFile(load1.str(), *src);
        cout << "load file Concatenated" << " Size: " << src->size() << endl;
        loadPCDFile(load2.str(), *tgt);
        cout << "load file " << i + step << " Size: " << tgt->size() << endl;
```

Συνένωση (concatenate)

Εφόσον ανακτήσουμε τα δεδομένα πρέπει να τα συνδέσουμε μεταξύ τους με την σειρά για να επεξεργαστούν από την Poisson συνολικά, σαν ένα αυτούσιο point-cloud. Μόνο με αυτό τον τρόπο θα μπορέσουν τα σημεία να συγκριθούν και να ενωθούν, ώστε να δημιουργηθεί το πλέγμα επιφάνειας. Για να επιτευχθεί αυτό, απλά προσθέτουμε τα point-clouds μεταξύ τους. Στη συνέχεια, αποθηκεύουμε το αποτέλεσμα που θα περιέχει τα αθροισμένα τα points όλων των στιγμιότυπων που χρησιμοποιήθηκαν από τον βρόγχο.

```
*final = *src;
*final += *tgt;

saveConcat
                << "/home/psistakis/Desktop/Human/transformed2/results/concatHuman.pcd";
savePCDFile(saveConcat.str(), *final);
```

Φιλτράρισμα

Στο κομμάτι αυτό γίνεται το φιλτράρισμα του τελικού point-cloud προκειμένου να αφαιρεθούν τα περιττά points που έχουν μαζευτεί συνδέοντας όλα τα στιγμιότυπα μαζί. Παράλληλα χρησιμοποιώντας τον MLS (βλ.: 3.1.1) λειαίνουμε την επιφάνεια αντιμετωπίζοντας με αυτόν τον τρόπο τυχών εξογκώματα ή σκιές που έχουν δημιουργηθεί στο εξωτερικό μέρος του.

```
downsampling(src, src_down, 0.00);
mlsFilter(final, src_mls);
downsampling(src_mls, final /*src_outlier*/, 0.002);

outlierRemoval(src_mls, src_outlier);

saveFinal
                << "/home/psistakis/Desktop/Human/transformed2/results/cleanedHuman.pcd";
savePCDFile(saveFinal.str(), *src_outlier);
```

Normalestimation

Έπειτα, για να δουλέψει σωστά η Poisson, θα πρέπει να γνωρίζει σε ποια κατεύθυνση θα πρέπει να τοποθετήσει εξωτερική επιφάνεια του πλέγματος. Όπως και στην computeTransformation(βλ.: 5.2.1) ομοίως και εδώ, χρησιμοποιήσαμε μια μορφή της κλάσης NormalEstimation.

```
        NormalEstimationOMP<PointXYZ, Normal> ne;

        ne.setNumberOfThreads(8);
        ne.setInputCloud(src_outlier);
        ne.setRadiusSearch(0.05);
        Eigen::Vector4f centroid;
        compute3DCentroid(*src_outlier, centroid);
//                              ne.setKSearch();

        ne.setViewPoint(centroid[0], centroid[1], centroid[2]);

        PointCloud<Normal>::Ptr cloud_normals(new PointCloud<Normal>());
        ne.compute(*cloud_normals);
        for (size_t i = 0; i < cloud_normals->size(); ++i) {
                cloud_normals->points[i].normal_x *= -1;
                cloud_normals->points[i].normal_y *= -1;
                cloud_normals->points[i].normal_z *= -1;
        }
        PointCloud<PointNormal>::Ptr cloud_smoothed_normals(
                        new PointCloud<PointNormal>());
        concatenateFields(*src_outlier, *cloud_normals, *cloud_smoothed_normals);

        cout << "Normal Output for meshing " << " Size: "
                        << cloud_smoothed_normals->size() << endl;

        Poisson<PointNormal> meshMaker;
        meshMaker.setDepth(10);
        meshMaker.setConfidence(true);
        meshMaker.setInputCloud(cloud_smoothed_normals);
        PolygonMesh mesh;
        meshMaker.reconstruct(mesh);
```

Poisson

Η Poisson, με την σειρά της, θα δημιουργήσει το πολυγονικό πλέγμα και θα δημιουργήσει ένα ολοκληρωμένο 3D αντεικείμενο. Αυτό το αντικείμενο θα αποθηκευτεί χρησιμοποιόντας την **savePLYFile.** Με την χρήση της **saveVTK** αποθήκέυουμε το 3D αντικείμενο σε μορφή *.vtk για να μπορεί να φορτωθεί από το Paraview (βλ.: 6.2) το οποίο είναι το λογισμικό του KYB3 (βλ.: 6.1).

Τα αρχεία μορφής PLY (*.ply) είναι γνωστά και ως αρχεία πολυγονικής μορφής (Polygonfiles) ή μορφής τριγώνων Στανφορντ (StanfordFormat). Τέτοιας μορφής αρχεία κατασκευάστηκαν κυρίως για την αποθήκευση τρισδιάστατων δεδομένων από 3D scanners. Υποστηρίζουν μια σχετικά απλή μορφή ενός και μόνο αντικειμένου, όπως μια λίστα επίπεδων γωνιών. Επίσης, στην μεγάλη ποικιλία ιδιοτήτων μπορούν να αποθηκευτούν συμπεριλαμβάνονται το χρώμα, η διαφάνεια, τα normals, συντεταγμένες κ.α.

```
Poisson<PointNormal> meshMaker;
meshMaker.setDepth(10);
meshMaker.setConfidence(true);
meshMaker.setInputCloud(cloud_smoothed_normals);
PolygonMesh mesh;
meshMaker.reconstruct(mesh);

savePLYFile(
                "/home/psistakis/Desktop/Human/transformed2/results/finalHumanMesh.ply",
                mesh);

cout << "Final image saved!" << endl;

cout << "3DScanning Completed" << endl << "Thank you" << endl;
```

Εικόνα 47 - Τελικό αποτέλεσμα (δεξιά)

6 Εφαρμογές

Στο κομμάτι αυτό θα παρουσιαστούν πιθανές χρήσεις του 3D scanner που αναπτύχθηκε. Πέρα από την παρουσίαση των 3D μοντέλων που δημιουργήθηκαν με την εφαρμογή μας μαζί με την βοήθεια προγραμμάτων απεικόνισης ή ιστοσελίδων, δημιουργήθηκε η δυνατότητα χρήσης τους σε εικονικό περιβάλλον (Virtual Environment–VR) μόνο με κάποιες μικρές αλλαγές στον κώδικα.

6.1 Kyb3 - Virtual Environment (Cave System)

Το Kyb3[III] είναι ένας semi-immersive εικονικό περιβάλλον που σχεδιάστηκε και κατασκευάστηκε το 2012-2013 από τους **Heiko Herrmann** και **Emiliano Pastorelli**, μέλη του Visualization Group του τμήματος Μηχανικών και Εφαρμοσμένων Μαθηματικών, του Ινστιτούτου Κυβερνητικής του Ταλλίν Πανεπιστημίου της Εσθονίας (Institute of Cybernetics, Tallinn University of Technology).

Αποτελείται από τρείς διάτρητες πλευρές, οι οποίες απεικονίζουν παθητική στεροσκοπική εικόνα. Διαθέτει πλήρης 6-DOF (ελευθερία κίνησης τριών διαστάσεων) μαγνητικό εντοπισμό και αποτελεί το πρώτο VR καθώς και την πρώτη έρευνα πάνω στην Virtual και Augmented Reality. Αποτελεί μια από τις πιο εργονομικές κατασκευές του είδους του λόγο του μεγέθους του (διαστάσεις: 2μ x 1.7μ x1,9μ).

Εικόνα 48–Το Kyb3 Cave System σε χρήση

Hardware:

6 projectors: Acer S5201B DLP
KyB3 Workstation:

- Dual 8-Core Opteron 4284 3.0 GHz
- 64GB ECC Registered 1666 MHz RAM
- 4x Nvidia Quadro 4000 Graphic Cards
- Intel SSD 300 GB HDD

Τρείς από τις κάρτες γραφικών χρησιμοποιούνται για του projectors (2 σε κάθε κάρτα), ενώ η τέταρτη λειτουργεί για την οθόνη του συστήματος.

Tracking &Interaction:

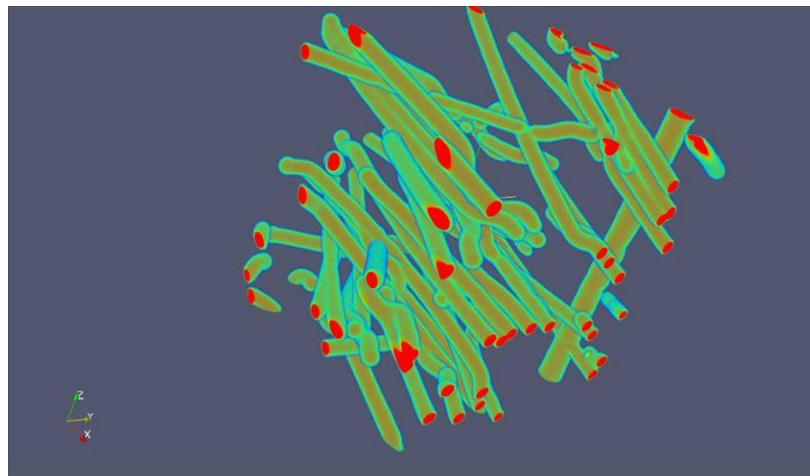
- VR-Space Wintracker III
- Wiimote
- Polarized glasses

Εικόνα49 - VR-Space Wintracker, Wiimote controller, Polarized glasses

Software:

- **VRUI** + Several of the VRUI-based Software, By Oliver Kreylos, UC Davis
- **ParaView** by KitWare
- **VDM** (Visual Molecular Dynamices) by University of Illinois
- **jReality** by TU Berlin
- **VRPN** by Russell M. Taylor II

Η βασική χρήση του Kyb3 είναι η απεικόνιση και ανάλυση δομικών υλικών, κυρίως σκυρόδεμα με ατσάλινες ίνες.



Εικόνα 50 -3DΠαράδειγμα ατσάλινων ινών

6.2 Πολιτιστική Κληρονομιά

Τα τελευταία χρόνια, η αρχαιολογική κοινότητα παγκοσμίως χρησιμοποιεί όλο και περισσότερο διάφορες μεθόδους για την σάρωση αρχαιολογικών αντικειμένων και αρχαιολογικούς χώρους, με στόχο την ψηφιοποίηση τους. Σε συνδυασμό με τους 3D εκτυπωτές, δίνεται η δυνατότητα δημιουργίας αντιγράφων (replica). Με αυτό τον τρόπο

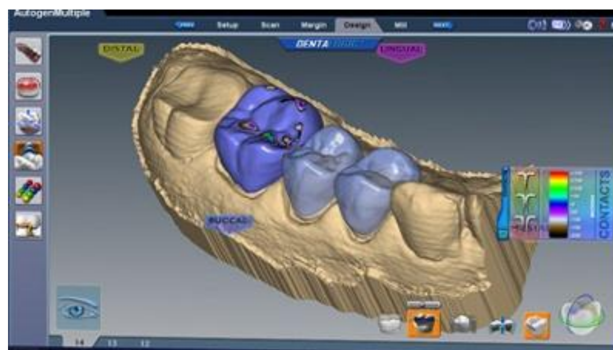
μειώνεται η πιθανότητα καταστροφής πολύτιμων, ευαίσθητων και μοναδικών αντικειμένων. Επιπλέον, έχοντας ψηφιοποιημένα τέτοιου είδους αντικείμενα δίνεται η ευκαιρία σε όλο τον κόσμο να έχει πρόσβαση σε αρχαιολογικά ευρήματα, μεγάλης ιστορικής αξίας, σε ένα διαδραστικό περιβάλλον που μπορεί να είναι υπολογιστής ή Virtual Environment.

Εικόνα 51 - Δαβίδ του Μιχαήλ Άγγελου
από την Cyberware (αριστερά το πρωτότυπο)

6.3 Υγεία

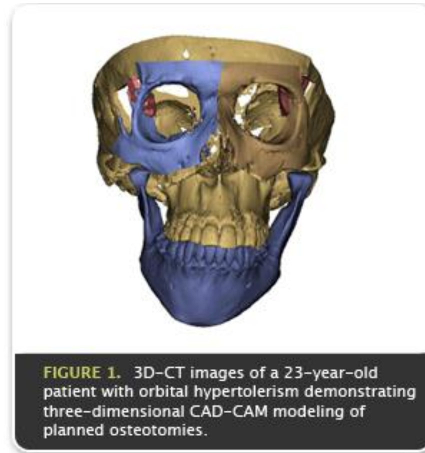
Εφαρμογές στον τομέα της υγείας δεν θα μπορούσαν να λείψουν. Τρισδιάστατοι εκτυπωτές χρησιμοποιούνται για να συλλάβουν το σχήμα από μέρη ή ολόκληρου το σώματος των ασθενών. Υπολογίζει σταδιακά τις βελτιώσεις που πρέπει να γίνουν και με την χρήση CAD/CAM λογισμικού, σχεδιάζονται και κατασκευάζονται τα απαραίτητα προσθετικά ή νάρθηκες.

Στις μέρες μας, αρκετοί ορθοδοντικοί χρησιμοποιούν αυτή την τεχνική. Χρησιμοποιούν τους τρισδιάστατους σαρωτές για να αιχμαλωτίσουν την επιφάνεια της στοματικής κοιλότητας. Έπειτα, τα CAD/CAM κατασκευάζουν τα απαραίτητα προσθετικά και τα εφαρμόζουν στις θέσης που χρειάζονται.



Εικόνα 52 - CAD/CAMστην οδοντιατρική

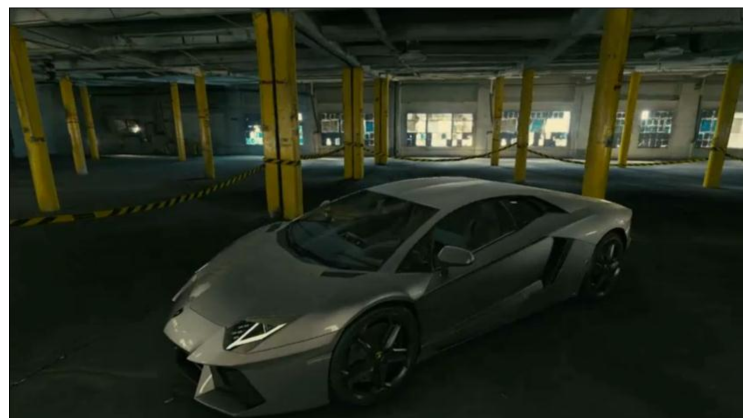
Επιπλέον, τέτοιου είδους τεχνικές εφαρμόζονται για άτομα με έλλειψη κάποιων οστών στα άκρα ή ακόμα και στο πρόσωπο.



Εικόνα 53 – CAD/CAM Οσταιοπλαστικής

6.4 Ψυχαγωγία

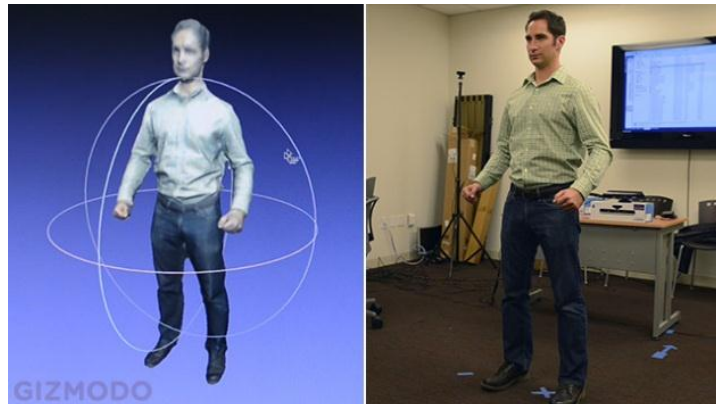
Οι 3D σαρωτές χρησιμοποιούνται πλέον κατά κόρον από την βιομηχανία θεάματος για την δημιουργία ψηφιακών μοντέλων και 3D ταινίες, βίντεο-παιχνίδια και σκοπούς αναψυχής. Σε μεγάλο βαθμό χρησιμοποιούνται εικονική κινηματογραφία (Virtual Cinematography) για την «μεταφορά», αντικειμένων του πραγματικού κόσμου στον εικονικό. Η μετάβαση αυτή, είναι ευκολότερη και γρηγορότερη από την δημιουργία των ίδιων αντικειμένων με την χρήση λογισμικών 3Dμοντελοποίησης. Σε πολλές περιπτώσεις, καλλιτέχνες σκαλίζουν τα πρωτότυπα με φυσικό τρόπο και με την βοήθεια των σαρωτών και απαραίτητων λογισμικών, γίνεται η μορφοποίηση.



Εικόνα 54–VideoCinematography

Πρόσφατα, η εταιρία USC Institute for Creative Technologies παρουσίασε την εφαρμογή FastAvatarCapture, όπου χρήστες Microsoft Kinect μπορούν να σαρώνουν τον εαυτό τους και να τον χρησιμοποιούν σαν avatar σε βίντεο παιχνίδια και όχι μόνο.

Το μοντέλο που έχει σαρωθεί από το Kinect και την εφαρμογή περνά στο λογισμικό Smart Body [IX] όπου αυτόματα προσθέτει ένα «σκελετό» και την εικονική επιφάνεια η οποία είναι παραμορφώσιμη για να μπορούν να εφαρμοστούν οι κινήσεις του avatar.



Εικόνα 55 - Fast Avatar Capture

7 Δημοσιεύσεις

Αυτή η πτυχιακή εργασία, αποτέλεσε μέρος μιας ευρύτερης έρευνας [XIV] για τρόπους ψηφιοποίησης αντικειμένων σε 3D μοντέλα και θα δημοσιευτεί σαν επιστημονική εργασία από τους **Emiliano Pastorelli** και **Heiko Herrmann,** μέλη του Visualization Group του τμήματος Μηχανικών και Εφαρμοσμένων Μαθηματικών, του Ινστιτούτου Κυβερνητικής του Ταλλίν Πανεπιστημίου της Εσθονίας (Institute of Cybernetics, Tallinn University of Technology).

Η συγκεκριμένη επιστημονική εργασία, πραγματεύεται την οπτικοποίηση Φωτογραμμετρικών 3D αναπαραστάσεων (Photogrammetric 3D Reconstructions) για την χρήση τους σε εικονικά περιβάλλοντα (Virtual Environments) και για πολιτιστική κληρονομιά (π.χ.: Virtual Museums – Εικονικά μουσεία). Γίνεται αναφορά σε εφαρμογές παραγωγής φωτογραμμετρικής 3D αναπαράστασης και του 3D scanner που αναπτύξαμε καθώς και της χρήσης των αποτελεσμάτων τους στο Kyb3 (βλ.: 6.1).

Σύμφωνα με την εργασία χρησιμοποιείται μια απλή φωτογραφική κάμερα, με την οποία ο χρήστης τραβάει το αντικείμενο περιμετρικά με αρκετές φωτογραφίες (50-70) οι οποίες πρέπει να επικαλύπτονται μεταξύ τους. Κατά αυτήν την διαδικασία το αντικείμενο πρέπει να μείνει ακίνητο και ο φωτισμός του να είναι σταθερός. Έπειτα η εφαρμογή, χρησιμοποιώντας την τεχνική της φωτογραμμετρικής συγκρίνει τις φωτογραφίες μεταξύ τους για κοινά σημεία και δημιουργεί την 3D αναπαράσταση.

Εικόνα 56 - Παράδειγμα διαδικασίας σάρωσης φωτογραμμετρικής εφαρμογής VisualSFM

8 Βιβλιογραφία

I. Burgarda, N. E. (n.d.). *Real-time 3D visual SLAM with a hand-held RGB-D camera.*

II. D'Apuzzo, D. N. (n.d.). *3D Human Body Scanning Technologies.*

III. Emiliano Pastorelli, H. H. (n.d.). *A Small-scale, Low-budget Semi-immersive Virtual Environment for Scientific Visualization and Research.*

IV. F. Lu, E. M. (n.d.). *Globaly Consistent Range Scan for Environment Mapping (LUM).*

V. Ichim, A. -E. (n.d.). *RGB-D Handheld Mapping and Modeling.* École Polytechnique Fédérale de Lausanne.

VI. Jean-Daniel Boissonnat, D. C.-S. (n.d.). *Meshing of Surfaces.*

VII. Jochen Sprickerhof, A. N. (n.d.). *An Explicit Loop Closing Technique for 6D SLAM (ELCH).*

VIII. Marc Alexa, J. B.-O. (n.d.). *Computing and Rendering Point Set Surfaces.*

IX. Marcus Thiebaux, A. N. (n.d.). *SmartBody: Behavior Realization.*

X. Michael Kazhdan, M. B. (n.d.). *Poisson Surface Reconstruction.*

XI. Nikon Metrology. (n.d.). *ModelMaker MMDx - MMC.* Nikon Metrology.

XII. Pirovano, M. (n.d.). *Kinfu – an open source implementation of Kinect Fusion + case study: implementing a 3D scanner with PCL.*

XIII. Yaw, L. L. (n.d.). In*troduction to Moving Least Squares(MLS) Shape Functions.*

XIV. Heiko Herrmann, Emiliano Pastorelli. (n.d.) Virtual Reality Visualization for Photogrammetric 3D Reconstructions of Cultural Heritage