ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΗΛΕΚΤΡΟΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΕΠΙΚΟΙΝΩΝΙΩΝ, ΗΛΕΚΤΡΟΝΙΚΗΣ ΚΑΙ ΣΥΣΤΗΜΑΤΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΕΡΓΑΣΤΗΡΙΟ ΑΚΟΥΣΤΙΚΗΣ, ΕΠΙΚΟΙΝΩΝΙΑΣ ΚΑΙ ΤΕΧΝΟΛΟΓΙΑΣ ΜΜΕ

Προσομοίωση κίνησης delta robot στο Matlab, για χρήση σε διαδραστικά παιχνίδια

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αγγελίνος Ευάγγελος

Επιβλέπων: Γεώργιος Καμπουράκης

Επ. Καθηγητής ΕΜΠ

Αθήνα, Νοέμβριος 2011

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΗΛΕΚΤΡΟΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΕΠΙΚΟΙΝΩΝΙΩΝ, ΗΛΕΚΤΡΟΝΙΚΗΣ ΚΑΙ ΣΥΣΤΗΜΑΤΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΕΡΓΑΣΤΗΡΙΟ ΑΚΟΥΣΤΙΚΗΣ, ΕΠΙΚΟΙΝΩΝΙΑΣ ΚΑΙ ΤΕΧΝΟΛΟΓΙΑΣ ΜΜΕ

Προσομοίωση κίνησης delta robot στο Matlab, για χρήση σε διαδραστικά παιχνίδια

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αγγελίνος Ευάγγελος

Επιβλέπων: Γεώργιος Καμπουράκης

Επ. Καθηγητής ΕΜΠ

Αθήνα, Νοέμβριος 2011

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 25η Νοεμβρίου.

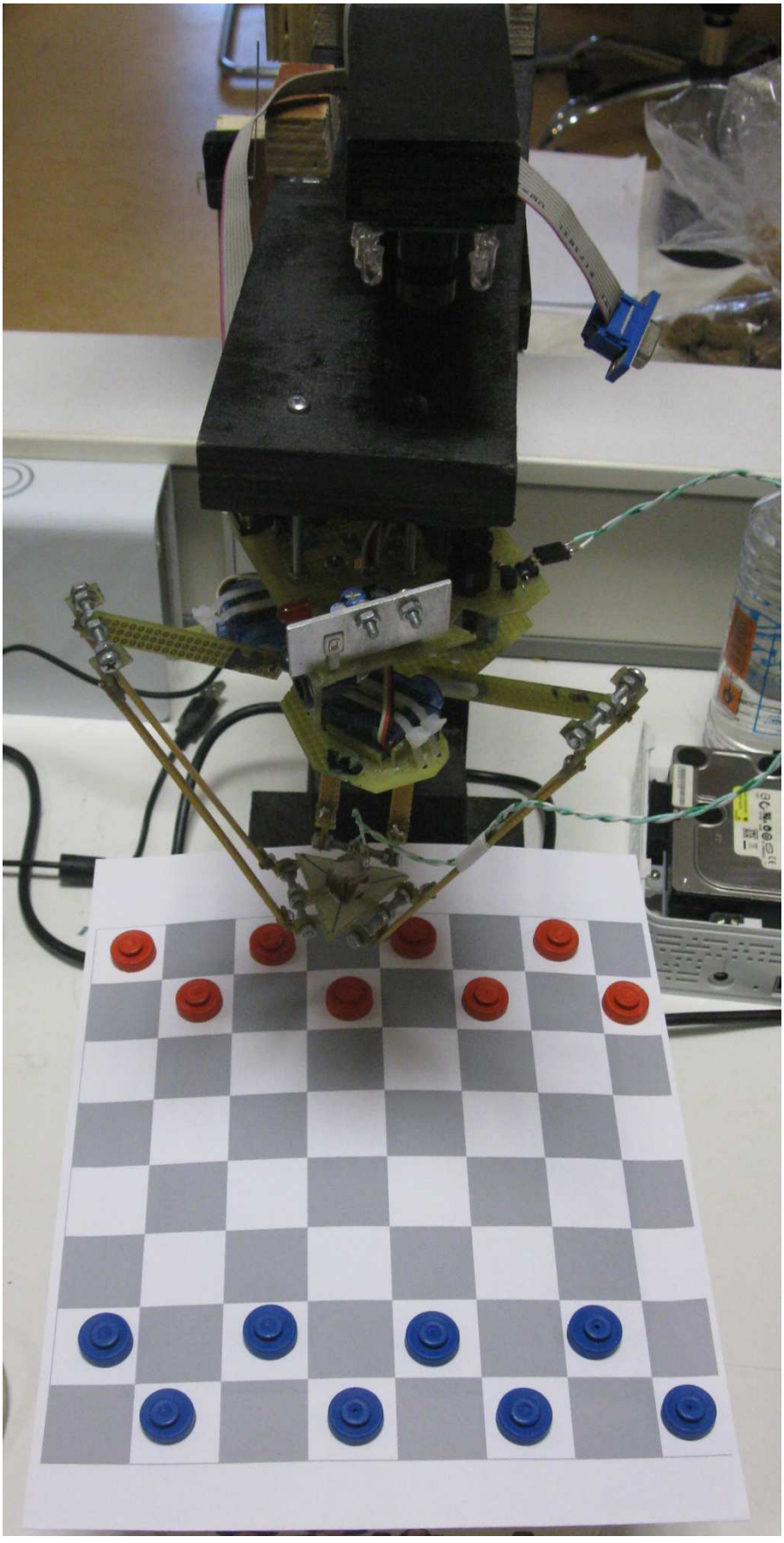
.....	.....	.....
Γ. Καμπουράκης	Β. Λούμος	Ε. Καγιάφας
Επ. Καθηγητής ΕΜΠ	Καθηγητής ΕΜΠ	Καθηγητής ΕΜΠ

Αθήνα, Νοέμβριος 2011

.....

Αγγελίνος Ευάγγελος

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.



Περίληψη

Ο σκοπός της διπλωματικής εργασίας που διαβάζετε ήταν ο προγραμματισμός ενός ρομπότ delta, ώστε να παίζει ντάμα σε πραγματικές συνθήκες με έναν αντίπαλο – άνθρωπο.

Το ρομπότ θα είναι στερεωμένο σε μία βάση με την αρπάγη προς τα κάτω και συνδεδεμένο σε σειριακή θύρα ενός PC. Ακριβώς κάτω από την αρπάγη θα βρίσκεται η σκακιέρα που παίζεται το παιχνίδι και τα πούλια. Οι κινήσεις του ρομπότ θα γίνονται με τη βοήθεια της ηλεκτρομαγνητικής αρπάγης.

Στην εργασία πρώτα αναφέρονται κάποια ιστορικά στοιχεία για τα ρομπότ, και για την τεχνητή όραση και την εξελιξή της. Γίνεται ακόμα αναφορά στην τοπολογία της κατασκευής στην οποία θα ενσωματωθεί μελλοντικά ο κώδικας.

Ακολουθεί η εκτενής προσομοίωση της λειτουργίας του ρομπότ σε προγραμματιστικό περιβάλλον (Matlab) ώστε να εξασφαλιστεί η εύρυθμη λειτουργία του, και έπειτα το πρόγραμμα που θα δίνει τη δυνατότητα στο ρομπότ να έχει Νοημοσύνη και να αποκρίνεται στις μεταβολές της σκακιέρας. Το πρόγραμμα αυτό (Visual πρόγραμμα) σχεδιάστηκε σε γλώσσα C με τη βοήθεια των βιβλιοθηκών της OpenCV2.2 και μίας webcam η οποία είναι στερεωμένη και αυτή στη βάση του ρομπότ και θα λειτουργεί ουσιαστικά σαν αισθητήρας. Το visual πρόγραμμα δίνει τη δυνατότητα στο ρομπότ να αντιλαμβάνεται την σκακιέρα, τα διαφορετικά χρώματα για κάθε πούλι με τις αντίστοιχες θέσεις τους, καθώς και την οποιαδήποτε κίνηση στο πεδίο της κάμερας.

Για την πλήρη λειτουργία του ρομπότ χρησιμοποιήθηκε ακόμα ένα πρόγραμμα με τους κανόνες του παιχνιδιού της ντάμας (Rules Program – γραμμένο σε C) και άλλο ένα πρόγραμμα με το ‘gameplay’ του παιχνιδιού (Gameplay Program – γραμμένο σε C. Το πρώτο πρόγραμμα δεν αποτέλεσε εκτενές σημείο μελέτης της εργασίας αυτής. Το δεύτερο ουσιαστικά καθοδηγεί το ρομπότ να παίξει στα πλαίσια κάποιων γενικότερων κανόνων του παιχνιδιού (πχ. να ελέγχει αν έχασε) και φυσικά να συνδέει μεταξύ τους τα προγράμματα Rules και Visual.

Εν τέλει γίνεται μία αναφορά στο παιχνίδι της ντάμας και στους αλγόριθμους που χρησιμοποιήθηκαν στο Rules Program, στη χρησιμότητα της κατασκευής και τις όποιες προοπτικές εξέλιξης της κατασκευής.

Λέξεις – φράσεις κλειδιά:

Ρομπότ Δέλτα, Παράλληλο Ρομπότ, C, OpenCV, Matlab, Τεχνητή Νοημοσύνη, Minmax, Κλάδεμα Άλφα-Βήτα, Ντάμα, Στρατηγική παιχνιδιού, Σκακιέρα, Τάλως, Φρανκενστάιν, Τεχνητή Όραση

Abstract

The purpose of the thesis you are reading was the programming of a delta robot, to play checkers in the real world with an enemy - human.

The robot will be mounted on a base with the clam down and connected to a serial port on a PC. Just below the clam is the chess board where tha game is played, and where the checkers are resting . The movements of the robot will be made with the help of an electromagnetic gripper.

In the beginning some historical facts about robotics science and Computer Vision are pointed out along with elements of the topology of the structure (mechanical and electronic parts) in which the code will be implemented in the future.

This is followed by the extensive simulation of the operation of the robot in a programming environment (Matlab) to ensure proper functioning, and after that the program that will enable the robot to have artificial intelligence and respond to changes in the board. This program (Visual program) is developed in C language with the help of libraries of OpenCV2.2 and a webcam based on the robot which operates as a sensor. The visual program enables the robot to perceive the chessboard, the different colors on each piece with their respective positions and any movement in the field of the camera.

For the full operation of the robot, we also used a program with the rules of the game of checkers (Rules Program – written in C) and a gameplay program (Gameplay Program – written in C). The first program was not subject of extensive study in this thesis. The Gameplay Program basically gives direction to the robot to do basic operations that are not directly linked to the rules of the game (eg. Checks if the robot lost) and of course it interconnects the two other programs (Rules, Visual) together.

At last, there is a reference to the game of checkers and to the algorithms that are used in the Rules Program, the utility of the construction as well as it's development prospects.

Keywords

Delta Robot, Parallel Robot, C, OpenCV, Matlab, Artificial Intelligence, MinMax, Alpha-Beta Prunning, Checkers, Draughts, Chessboard, Talos, Frankenstein, Gameplay, Computer Vision

Τα ευχαριστώ μου...

Οι ευχαριστίες φυσικά δεν θα μπορούσαν να ξεκινάνε από αλλού, παρά από την οικογενειά μου που με την υλική και ηθική της συμπαράσταση με στήριξε ώστε να καταφέρω να φτάσω στο τέλος των σπουδών μου. Σας ευχαριστώ!

Ευχαριστώ ακόμα τον κ. Καμπουράκη για την ευκαιρία που μου έδωσε να ασχοληθώ με ένα τόσο ενδιαφέρον θέμα και τις συμβουλές του σε κρίσιμα σημεία της εκπόνησης του θέματος, και βεβαίως τον κ. Πιπερίδη για την πλήρη και συνεχή καθοδήγηση του, τη βοήθεια του καθώς και τον χρόνο που αφιέρωσε.

Εν τέλει δεν θα μπορούσα να αφήσω απ' έξω φίλους και συναδέλφους που με στήριξαν στην προσπάθεια αυτή είτε με τις γνώσεις τους είτε ψυχολογικά: Γιάννη Αυγουλέα, Μαρίνα Σταλιμέρου, Γιάννη Τουρνάκη, Βασίλη Κούρτη και όσους άλλους μπορεί να ξέχασα σας ευχαριστώ!

Σκοπός της διπλωματικής

Σκοπός της διπλωματικής είναι ο προγραμματισμός delta robot με απώτερο στόχο να παίζει ντάμα με αντίπαλο κάποιον άνθρωπο, σε πραγματικές συνθήκες. Κυρίαρχο κομμάτι της διπλωματικής ήταν αυτό της προσομοίωσης της κίνησης του ρομπότ στο Matlab, καθώς και του visual προγραμματισμου-να καταστήσουμε δηλαδή το ρομπότ ικανό να 'βλέπει' την σκακιέρα, τις αλλαγές πάνω σε αυτήν καθώς και να αναγνωρίζει τα πούλια και τη θέση τους, με τη βοήθεια της OpenCV. Τελικός στόχος της παρούσας εργασίας ήταν η δημιουργία όλου του απαραίτητου πλαισίου προγραμματιστικά, ώστε όλα τα παραπάνω να καταστούν πραγματικότητα, εκτός από την κατασκευή λογισμικού παιχνιδιού ντάμας.

Η οργάνωση του υλικού

Το υλικό της διπλωματικής θα παρουσιαστεί κομμάτι - κομμάτι με σκοπό την πλήρη κατανόηση των πεπραγμένων χωρίς ελλείψεις. Πρώτα θα γίνει μια εισαγωγή στην τεχνητή όραση και την ιστορίας της καθώς και στην επιστήμη της ρομποτικής, η οποία θα ακολουθείται από το απαραίτητο μαθηματικό υπόβαθρο πάνω στο οποίο στηρίχτηκε η προσομοίωση μας (κεφάλαιο 0). Έπειτα θα παρουσιαστεί η προσομοίωση της λειτουργίας του που έγινε στο Matlab (κεφάλαιο 1), και μετά θα αναλυθεί η Τεχνητή Νοημοσύνη του ρομπότ και ο αντίστοιχος κώδικας με τον οποίο προγραμματίσαμε το ρομπότ να βλέπει και να ανταποκρίνεται στις συνθήκες του παιχνιδιού (κεφάλαιο 2), αλλά και στους κανόνες του (κεφάλαιο 3). Στο $4^{ο}$ κεφάλαιο θα γίνει αναφορά στην στη διασύνδεση των διαφόρων μερών του κώδικα μεταξύ τους. Ακολούθως, θα γίνει αναφορά στο μηχανικό αλλά και στο ηλεκτρονικό μέρος της κατασκευής του ρομπότ (κεφάλαιο 5) με τη συνοδεία των απαραίτητων φωτογραφιών και έπειτα στη διαδικασία που ακολουθήθηκε για να παρθουν οι απαραίτητες μετρήσεις των διαστάσεων του ρομπότ (κεφάλαιο 6). Στο $7^{ο}$ κεφάλαιο αναφέρονται τα συμπεράσματα που προέκυψαν, η χρησιμότητα της κατασκευής καθώς και προτροπές και ιδέες για παιρετέρω εξέλιξη. Η εργασία συνοδεύεται και από ένα CD-ROM με όλα τα απαραίτητα αρχεία. Οποιαδήποτε αναφορά σε κώδικα γίνει μέσα στην εργασία θα συνοδεύεται και με τα απαραίτητα σχόλια. Στο τέλος της εργασίας, υπάρχουν κάποια παραρτήματα με κώδικα ο οποίος είχε βοηθητικό αλλά σημαντικό ρόλο.

Περιεχόμενα

0. Εισαγωγή

0.1 Όραση μηχανής

Όραση μηχανής (Computer Vision) είναι το επιστημονικό πεδίο το οποίο ασχολείται με την αυτοματοποιημένη απεικόνιση, και την αυτοματοποιημένη επεξεργασία εικόνων με απώτερο στόχο την εξαγωγή πληροφοριών απο τις εικόνες αυτές.

Η ιστορία της όρασης μηχανής ξεκινάει κάπου το 1960 όπου έχουμε την πρώτη επεξεργασμένη ψηφιακή εικόνα από ηλεκτρονικό υπολογιστή. Το 1968 έχουμε τη πρώτη δημοσίευση που αφορά αναγνώριση προτύπων (pattern recognition) και το 1969 τη δημοσίευση 'Επεξεργασία εικόνας από υπολογιστή' του A. Rosenfeld. Ακολουθεί πληθώρα συνεδρίων πάνω στην 'Επεξεργασία εικόνας'. Προφανώς και αυτός ο κλάδος της πληροφορικής θα ακολουθήσει την αλματώδη ανάπτυξη των επόμενων δεκαετιών έως ότου φτάσουμε σήμερα στο 2011 όπου η τεχνητή όραση μηχανών κατέχει σημαντική θέση στη ζωή του ανθρώπου. Σημαντικές εφαρμογές της υπάρχουν στα ρομπότ της βιομηχανίας, σε κατασκευές που έχουν σχέση με την παρακολούθηση χώρων, στην οργάνωση πληροφοριών που σχετίζονται άμεσα με εικόνες ή βίντεο, ακόμα και στην ιατρική!

Φτάνοντας λοιπόν στον 21^{ο} αιώνα (ή μαλλον λίγο πριν το 1999) έρχεται στο φως το νέο εγχείρημα της Intel η OpenCV. Η OpenCV (Open Source Computer Vision Library) είναι μία βιβλιοθήκη συναρτήσεων στοχεύοντας κυρίως στην τεχνητή όραση 'πραγματικού χρόνου'.

Ως αρχικοί στόχοι του εγχειρήματος είχαν τεθεί:

(α.) Η παροχή βιβλιοθηκών ανοιχτού και βελτιστοποιημένου κώδικα για βασικές λειτουργίες τεχνητής όρασης.

(β.) Παροχή μίας κοινής υποδομής πάνω στην οποία οι προγραμματιστές θα μπορούσαν να 'χτίσουν', ώστε οι κώδικες να είναι και πιο αναγνώσιμοι και πιο 'μεταβιβάσιμοι'.

(γ.) Η παροχή ελεύθερου κώδικα και για εμπορικές εφαρμογές, αφού η άδεια του δεν απαιτεί αυτές οι εφαρμογές να είναι επίσης ελεύθερες.

Η πρώτη της έκδοση παρουσιάστηκε στο κοινό στο συνέδριο της IEEE πάνω στην 'Τεχνητή Όραση και Αναγνώριση Προτύπων' το 2000. Ακολουθούν 5 beta εκδόσεις της γλώσσας έως ότου εκδίδεται το 2006 η OpenCV 1.0. Τον Οκτώβρη του 2008 ακολουθεί η έκδοση 1.1, ενώ τον Οκτώβρη του 2009 εκδίδεται η 2.0 έκδοση με σημαντικές αλλαγές.

Η OpenCV είναι γραμμένη σε γλώσσα C. Γι΄αυτόν τον λόγο όμως είναι σχετικά 'φορητή' και μπορεί να χρησιμοποιηθεί και άπο άλλες γλώσσες όπως Python κατά βάση αλλά και C#, Ruby και Java. Ο απαραίτητος για την εκπόνηση κώδικας γράφτηκε σε γλώσσα C.

0.2 Βασικές έννοιες που χαρακτηρίζουν ένα ρομπότ

Κατ' αρχάς θα αναφερθούμε σε κάποιες βασικές έννοιες που χαρακτηρίζουν ένα ρομπότ που θα μας διευκολύνουν στην καλύτερη κατανόηση όλων όσων ακολουθούν. Ένας εύκολος ορισμός για το ρομπότ θα μπορούσε να είναι: μια μηχανή η οποία καθοδηγείται από υπολογιστή ή είναι προγραμματισμένη ηλεκτρονικά έτσι ώστε να μπορεί να επιτελεί εργασίες μόνο του.

Βασικές παράμετροί ενός ρομπότ είναι:

1. Ο αριθμός των αξόνων πάνω στους οποίους κινείται: Ένα ρομπότ που κινείται πάνω σε δύο άξονες μπορεί να επιτελέσει εργασίες πάνω σ ένα μόνο επίπεδο ενώ κάποιο άλλο που κινείται σε τρείς άξονες μπορεί να επιτελέσει εργασίες στον τρισδιάστατο χώρο.
2. Οι Βαθμοί Ελευθερίας: Είναι το σύνολο των ανεξάρτητων μεταβλητών θέσης. Συνήθως ταυτίζονται με τον αριθμό των αρθρώσεων του ρομπότ. Ο αριθμός αυτός 'πρέπει να είναι γνωστός και μονοσήμαντα ορισμένος για κάθε διάταξη, έτσι ώστε να είναι εφικτός ο προσδιορισμός της θέσης των τμημάτων που τη συνιστούν'.
3. Περιοχή δράσης (working envelope): Είναι η περιοχή δράσης του ρομπότ. Πιο πρακτικά ότι βρίσκεται μέσα στην εμβέλεια δράσης του και μπορεί να το 'φτάσει'.
4. Η Κινηματική του: Αυτή εξαρτάται από τους βραχίονες, τους συνδέσμους και γενικότερα όλα τα μέλη του ρομπότ και καθορίζει τις κινήσεις του άρα και τη λειτουργία του.
5. Ικανότητα εξυπηρέτησης βάρους: Τι βάρος μπορεί να εξυπηρετήσει το ρομπότ.
6. Ταχύτητα του ρομπότ: Πόσο γρήγορα το ρομπότ μπορεί να φέρει το πέρας του βραχίονα του, στο ζητούμενο σημείο, και να επιτελέσει τη ζητούμενη εργασία.
7. Επιτάχυνση: Πόσο γρήγορα οι βραχίονες του ρομπότ μπορούν να επιταχύνουν.
8. Ακρίβεια: Πόσο κοντά μπορεί να φτάσει το ρομπότ στο ζητούμενο σημείο.
9. Επαναληψιμότητα: Πόσο ακριβής θα είναι η επιστροφή του ρομπότ στην αρχική του θέση ώστε να επαναλάβει την ίδια διαδικασία.

0.3 Μία σύντομη 'ρομποτική' ιστορική αναδρομή

Αν και φέτος κλείνουν μόλις 60 χρόνια ακριβώς από όταν το πρώτο βιομηχανικό ρομπότ μπήκε στη παραγωγή το 1961 στα εργοστάσια αυτοκινήτων της General Motors, ο άνθρωπος από πολύ πιο πριν είχε φανταστεί ένα μηχανικό βοηθό-σύντροφο ο οποίος και θα τον διευκόλυνε στις εργασίες της σκληρής καθημερινής ζωής καταρχάς, αλλά και θα τον ψυχαγωγούσε.

Γι' αυτόν ακριβώς το λόγο τα ρομπότ στη σκέψη του ανθρώπου υπάρχουν χιλιάδες χρόνια ήδη. Χαρακτηριστικότερο παράδειγμα είναι ο θεός 'Ηφαιστος της Ελληνικης Μυθολογίας ο οποίος στο εργαστήριο του στα έγκατα της γης είχε κατασκευάσει ανθρωποειδή ρομπότ-υπηρέτριες με ικανότητα κίνησης, ομιλίας και σκέψης, 'αυτόματα τρίποδα' που εξυπηρετούσαν, όπως αναφέρεται στην Ιλιάδα, αλλα και τον γνωστό σε όλους Τάλω. Ο Τάλως ήταν ένα γιγάντιο ανθρωπόμορφο άρμα που ο Ήφαιστος έφτιαξε κατα παραγγελία του Μίνωα, βασιλιά της Κρήτης και ετεροθαλή αδελφού του. Προφανής

σκοπός του ήταν να προστατεύει το βασίλειο του Μίνωα από πιθανούς εισβολείς, έχοντας μάλιστα την ικανότητα να γυρνάει όλο το νησί της Κρήτης τρεις φορές την ημέρα!

Μεγάλη όμως ανάπτυξη στην Αρχαία Ελλάδα είχε και η πραγματική μηχανική ('Από μηχανής θεός' στο αρχαίο θέατρο). Έτσι, το 270 π.Χ. ο Κτησίβιος, μηχανικός από την Αλεξάνδρεια, φτιάχνει μηχανικά όργανα με κινούμενα μέρη. Στον Κτησίβιο αποδίδεται η κατασκευή της ύδραυλις (όργανο στο οποίο ο ήχος παραγόταν με υδραυλική πίεση του αέρα) αλλά και η κατασκευή διάφορων υδραυλικών μηχανών (π.χ. υδραυλικό ρολόι), και σε αυτόν έγκειται η γέννηση της ρομποτικής σύμφωνα με τον Mark E. Rosheim. Δεν είναι όμως μόνο αυτός καθώς αρκετοί άλλοι μηχανικοί της αρχαίας εποχής κυρίως γύρω από τη Μεσόγειο αλλά και στην Κίνα έχουν να επιδείξουν μηχανές που θα μπορούσαν να θεωρηθούν πρόδομος των σημερινών ρομπότ – όπως Κινέζος Yan Shi που παρουσίασε στο βασιλιά του τον 10 αι. π.Χ. ένα ανθρωποειδές φτιαγμένο από ξύλο και δέρμα.

Τη σκυτάλη στο μεσαίωνα παίρνει ο Al-Jazari (1136-1206), ένας Μουσουλμάνος εφευρέτης που έφτιαξε πλειάδα αυτόματων μηχανών όπως κουζίνες αλλά και μουσικά όργανα! Φυσικά δεν θα μπορούσε να μην αναφερθεί ο Leonardo da Vinci ο οποίος στα τέλη του 15 αιώνα σχεδιάζει ένα ανθρωποειδές, αλλά και ο Jacques de Vaucanson ο οποίος κατασκεύσε μια αυτόματη μηχανή παιξίματος φλάουτου αλλά και μια πάπια η οποία κουνούσε τα φτερά της και τον λαιμό της! Ακόμα στη Κίνα τον 18^{ο} αιώνα κατασκευάστηκαν παιχνίδια και ζώα τα οποία είχαν πολύπλοκα μηχανικά μέρη ενώ ο Κινέζος Hisashige Tanaka κατασκεύασε το 19^{ο} αιώνα ένα παιχνίδι το οποίο ούτε λίγο ούτε πολύ σέρβιρε τσάι! Παράλληλα, τα ρομπότ κάνουν την εμφάνιση τους σε μυθιστορήματα επιστημονικής φαντασίας. Η Mary Shelley το 1818 θα γράψει για το πασίγνωστο πλέον ανθρωποειδές Φρανκεστάιν, προϊόν της εργασίας ενός επιστήμονα ο οποίος και του έδωσε ζωή.

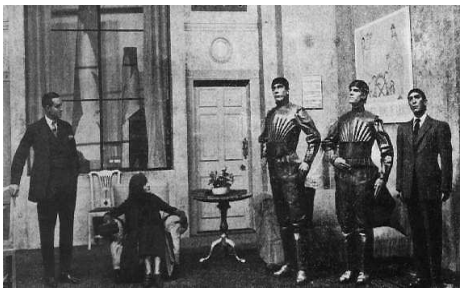
Εικόνα 1: Κινέζικο παιχνίδι (1796)

Εικόνα 2: Το ανθρωποειδές του Da Vinci

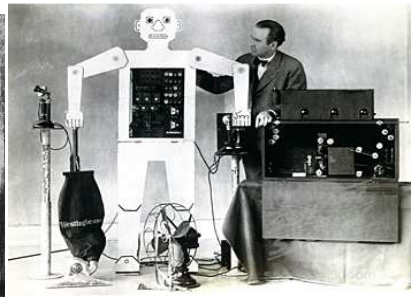
Μπαίνοντας στον 20^{ο} αιώνα, το 1921 γίνεται η πρώτη αναφορά στον όρο ρομπότ στο έργο του τσέχου συγγραφέα Καρελ Καπεκ (Karel Capek) 'R.U.R.' 'η 'Rossum's Universal Robots'. Και ενώ το 1926 παρουσιάζεται το Televox, το πρώτο ίσως 'παλαιού' τύπου ρομπότ που επιτελεί κάποια εργασία, το 1939 παρουσιάζεται το πρώτο ανθρωποειδές με το όνομα Elektro. Στα 1941 ο Ισαάκ Ασίμοφ (Isaac Asimov) θα αναφερθεί πρώτος στην έννοια της ρομποτικής και ένα χρόνο αργότερα θα ορίσει τους τρείς νόμους που διέπουν την λειτουργία ενός ρομπότ:

- 1^{ος} Νόμος: Ένα ρομπότ δεν μπορεί άμεσα ή έμμεσα να τραυματίσει τον άνθρωπο.

- 2ος Νόμος: Ένα ρομπότ πρέπει να υπακούει τις εντολές του ανθρώπου, εκτός και αν αυτές έρχονται σε αντίθεση με τον Πρώτο Νόμο.
- 3ος Νόμος: Ένα ρομπότ πρέπει να προστατεύει την ίδια του την ύπαρξη, εκτός και αν αυτό έρχεται σε αντίθεση με τον Πρώτο και τον Δεύτερο Νόμο.

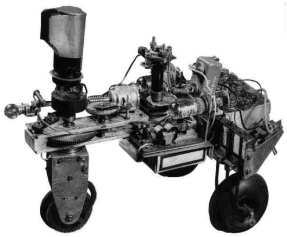


Εικόνα 3: Φωτογραφία από την παράσταση του 'R.U.R.' (ανθρωποειδή δεξιά).



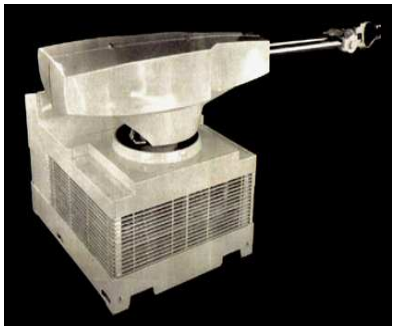
Εικόνα 4: Το Televox.

Τα πρώτα ηλεκτρονικά ρομπότ δημιουργούνται από τον William Grey Walter στην Αγγλία το 1948 και 1949 και ονομάζονται Elmer και Elsie (αρσενικό και θηλυκό όπως το 'Αδάμ και Εύα'). Φυσικά πλέον τα 'νέα' ρομπότ δεν θυμίζουν σε τίποτα τα πιο παλιά καθώς εφαρμόζονται σε αυτά γνώσεις ηλεκτρονικής και προγραμματισμού. Το 1948 θα εκδοθεί η έρευνα 'Cybernetics' του Norbert Wiener όπου αναφέρεται στην Τεχνητή Νοημοσύνη, το 1956 ιδρύεται η πρώτη εταιρεία ρομπότ με όνομα 'Unimation' από τους George Devol και Joseph Engelberger και το 1959 θα γίνει η πρώτη επίδειξη παραγωγής με τη βοήθεια computer στο εργαστήριο Σερβομηχανικής στο MIT (Massachusetts Institute of Technology). Και ενώ ως τώρα τα ρομπότ (κυρίως με τη μορφή 'κακών ανθρωποειδών') αποτελούν επιστημονική φαντασία σε βιβλία και ταινίες, εν τέλει γίνονται πραγματικότητα. Το 1961, όπως αναφέρθηκε και παραπάνω, το πρώτο βιομηχανικό ρομπότ με όνομα 'Unimation' 'πιάνει δουλειά' στα εργοστάσια της GM στο New Jersey. Το πόστο του δεν είναι άλλο από το να μεταφέρει αντικείμενα από ένα σημείο σε κάποιο άλλο. του Τον ίδιο χρόνο ο George Devol κερδίζει την πρώτη ρομποτική πατέντα (για την οποία και είχε κάνει αίτηση από το 1954). Πλέον η εξέλιξη της ρομποτικής θα είναι ραγδαία αλλάζοντας για πάντα τη ζωή του ανθρώπου.



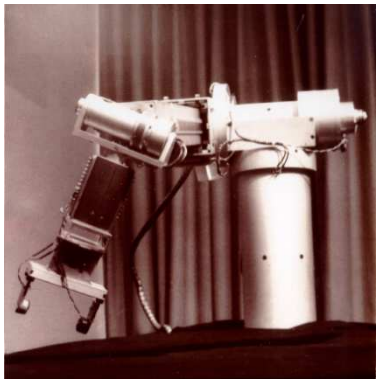
Εικόνα 5: Η Elsie

Εικόνα 6: Η πατέντα του Devol



Εικόνα 7: Το Unimation – το πρώτο βιομηχανικό ρομπότ

Το 1969 ο Victor Scheinman επινοεί στο Πανεπιστήμιο του Στάνφορντ τον πρώτο ρομποτικό βραχίονα ο οποίος προσομοιώνει την κίνηση ενός ανθρώπινου χεριού. Αυτή η ανακάλυψη δίνει νέα δυναμική στη ρομποτική. Τα ρομπότ πλέον μπορούν να επιτελέσουν πολύ πιό απαιτητικές και ακριβείς διαδικασίες όπως το να συγκεντρώνουν αντικείμενα. Ακολουθεί ένας δεύτερος βραχίονας αυτή τη φορά στο MIT, γι' αυτό και θα ονομαστεί 'MIT arm', η μετεξέλιξη του οποίου θα βγεί στην παραγωγή ως PUMA (Programmable Universal Machine for Assembly).



Εικόνα 8: Stanford arm

Εικόνα 9: PUMA robot

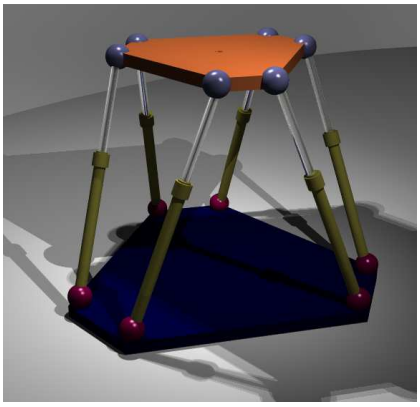
Ως βιομηχανικό ρομπότ ορίζεται πλέον από την ISO ως *ένας αυτόματα χειριζόμενος, επαναπρογραμματιζόμενος χειριστής-βραχίονας με διάφορες εφαρμογές, με τρείς ή παραπάνω άξονες* και η ρομποτική ορίζεται ως *το πεδίο μελέτης, σχεδιασμού και χρήσης των ρομποτικών συστημάτων για κατασκευαστικούς σκοπούς*. Η ρομποτική αποτελεί συνδυασμός πολλών άλλων επιστημών με κυριότερες τη μηχανολογία, την ηλεκτρονική και την πληροφορική.

Η ανάπτυξη της ρομποτικής συνεχίζεται με αμείωτους ρυθμούς έως ότου το 1982 ένας νέος τύπος ρομπότ προσελκύει τα φώτα της επιστημονικής κοινότητας πάνω του. Το ρομπότ αυτό δεν είναι άλλο από το **Delta Robot**, ένας τύπος Parallel Robot, και δημιουργός του είναι ο Reymond Clavel στη Πολυτεχνική Σχολή της Λωζάννης στην Ελβετία (École

Polytechnique Fédérale de Lausanne), ο οποίος κατοχύρωσε τη σχεδίαση του Delta Robot με ένα σύνολο 36 πατεντών!

Parallel Robot

Το Παράλληλο Ρομπότ αποτελείται από μια βάση και ένα σημείο δράσης. Η βάση και το σημείο δράσης συνδέονται μεταξύ τους με ρομποτικούς βραχίονες των οποίων ο αριθμός διαφέρει αλλά συνήθως είναι τρεις ή έξι. Οι βραχίονες αυτοί αν και είναι ανεξάρτητοι δρούν παράλληλα (μαζί) ο ένας με τον άλλον όχι όμως με την αυστηρή γεωμετρική έννοια, διατηρώντας επίσης το σημείο δράσης παράλληλο ως προς τη βάση. Βασικό πλεονέκτημα των Parallel Robot είναι η ταχύτητα και η ακρίβεια τους, ενώ βασικό τους μειονέκτημα είναι ο περιορισμένος χώρος δράσης (work envelope).



Εικόνα 10: Parallel Robot (Η βάση είναι κάτω και το σημείο δράσης από πάνω)

Delta Robot

Το Delta Robot κατασκευάστηκε με αρχικό σκοπό να μπορεί να μετακινήσει μικρά και ελαφριά αντικείμενα πολύ γρήγορα. Ο σχεδιασμός του μοιάζει πολύ με του Parallel Robot. Αποτελείται από μία βάση και ένα σημείο δράσης και συνήθως τρεις βραχίονες με την ιδιαιτερότητα ότι τις περισσότερες φορές ο χώρος δράσης βρίσκεται κάτω από τη βάση. Πιο απλά, το ρομπότ είναι αναποδογυρισμένο σε σχέση με αυτό της Εικόνας 8. Το σημείο δράσης είναι πάντα παράλληλο με τη βάση.

Εικόνα 10: Τυπικό Delta Robot

Η βάση και το σημείο δράσης είναι συνήθως δύο ισόπλευρα και όμοια τρίγωνα (τα τρίγωνα επιλέγονται να είναι ισόπλευρα γιατί λόγω των ιδιοτήτων τους διευκολύνουν την κινηματική μελέτη). Το ρομπότ κινείται και στις τρεις διαστάσεις ενώ έχει τρεις βαθμούς ελευθερίας. Η περιοχή δράσης και η κινηματική αναλύονται διεξοδικά παρακάτω. Οι βραχίονες τους είναι συνήθως φτιαγμένοι από ανθεκτικό άλλα ελαφρύ υλικό, κάτι που τους δίνει μεγάλη ταχύτητα, ενώ το βάρος που μπορούν να σηκώσουν είναι σχετικά περιορισμένο. Και οι τρεις βραχίονες δρούν 'παράλληλα'. Αυτό έχει σαν αποτέλεσμα μεγάλη ακρίβεια (η οποία εξαρτάται και απο την ποιότητα των σερβοκινητήρων) αλλά και περιορισμένο χώρο δράσης. Τέλος, το ρομπότ έχει εννέα αρθρώσεις (τρεις για κάθε βραχίονα) εκ των οποίων οι τρείς είναι ελεύθερες να κινούνται.

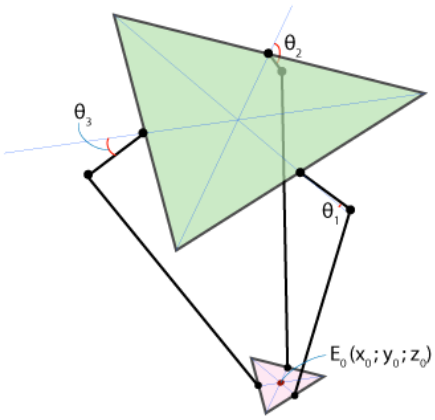
Τα παραπάνω χαρακτηριστικά έχουν καταστήσει το Delta Robot ένα απο τα πιο επιτυχημένα ρομπότ από το 1987 που μπήκε στη μαζική παραγωγή, ιδιαίτερα στη βιομηχανία με κύρια εργασία είτε το πακετάρισμα, είτε τη συγκέντρωση προϊόντων (για παράδειγμα από μία κινούμενη 'ταινία').

0.4 Η Κινηματική ενός Delta Robot

Το βασικότερο ίσως αντικείμενο μελέτης και προβληματισμού για την κατασκευή και το προγραμματισμό ενός Delta Robot είναι η κινηματική του. Η κινηματική ενός ρομπότ είναι η μελέτη της κίνησης του. Στην ανάλυση της κινηματικής συμπεριλαμβάνονται εκτός από τη θέση, και η ταχύτητα αλλά και η επιτάχυνση όλων των στοιχείων χωρίς να λαμβάνονται υπόψιν οι δυνάμεις που προκάλεσαν την κίνηση αυτή. Η κινηματική του αναλύεται σε δύο βασικά και αντίστροφα προβλήματα: (i.) Inverse Kinematics και (ii.) Forward Kinematics. Στην παρακάτω μελέτη δεν αναλύονται η ταχύτητα και η επιτάχυνση των στοιχείων του ρομπότ καθώς ο κατασκευαστικός σκοπός του ρομπότ μας δεν απαιτεί κάτι τέτοιο.

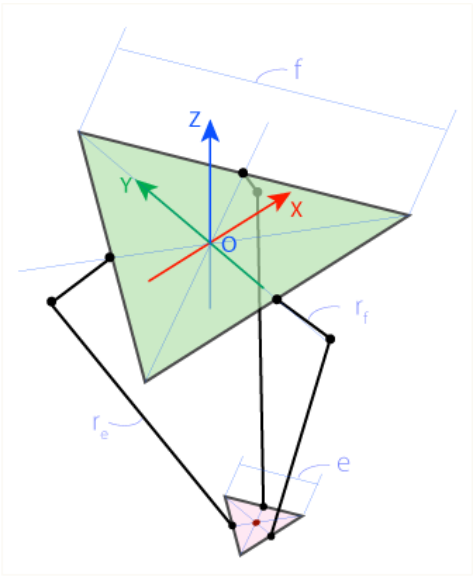
0.4.1 Inverse Kinematics – Αντίστροφη Κινηματική

Το πιό σύνηθες είναι να γνωρίζουμε σε ποιό σημείο του χώρου πρέπει να πάει το σημείο δράσης του ρομπότ ώστε να επιτελέσει τη ζητούμενη εργασία. Αυτό το σημείο χαρακτηρίζεται από 3 γεωμετρικές συντεταγμένες (ως προς το κέντρο της τριγωνικής του βάσης) $\Sigma_0(x_0,y_0,z_0)$. [Το κέντρο ενός τριγώνου ορίζεται γεωμετρικά ως το σημείο τομής των διαμέσων του]. Χρησιμοποιώντας τη γεωμετρία του ρομπότ αρκεί να βρούμε τις γωνίες κάθε βραχίονα με τη βάση ώστε το σημείο δράσης να είναι στο ζητούμενο σημείο στο χώρο. Εάν γνωρίζουμε τις γωνίες αυτές τότε είμαστε έτοιμοι να προγραμματίσουμε το ρομπότ μας.



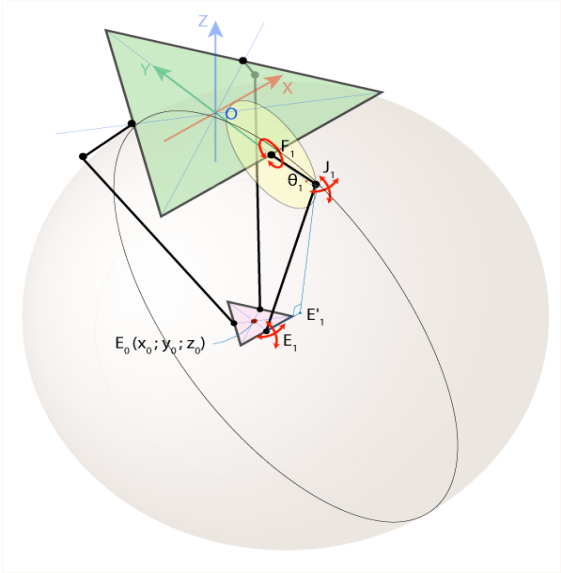
Εικόνα 11: Η κινηματική του ρομπότ σχηματοποιημένη. Διακρίνονται οι βασικές μεταβλητές της κινηματικής του: οι τρεις γωνίες των αξόνων θ1, θ2, θ3 και οι συντεταγμένες του σημείου δράσης E_0

Ονομάζουμε τη πλευρά του τριγώνου της βάσης f, ενώ τη πλευρά του τριγώνου του σημείου δράσης e και τα αντίστοιχα μέρη των βραχιόνων που άπτονται της βάσης ή του σημείου δράσης r_f και r_e. Θεωρούμε ότι γνωρίζουμε το σημείο στο οποίο βρίσκεται το κέντρο του τριγώνου του σημείου δράσης το οποίο είναι το $\Sigma_0(x_0,y_0,z_0)$.



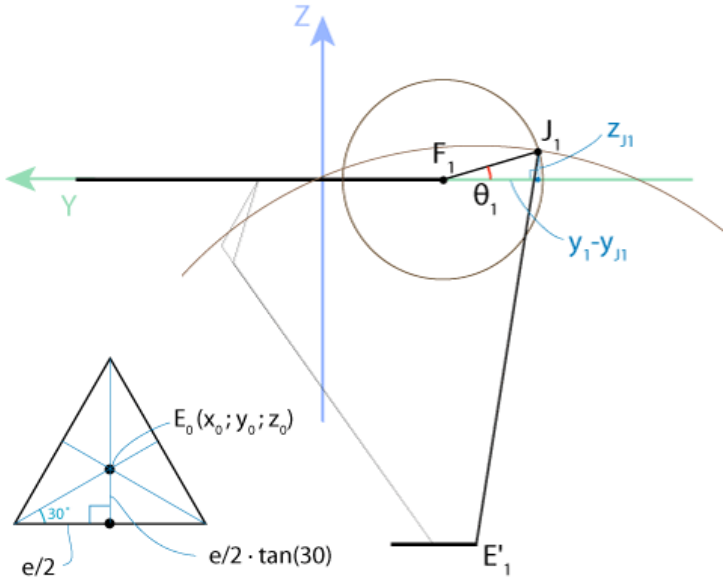
Εικόνα 12: Το ρομπότ στις τρείς διαστάσεις με τους αντίστοιχους άξονες

Αρκεί φυσικά να μελετήσουμε τον ένα βραχίονα απο τους τρεις και έπειτα τα αποτελέσματα να τα 'περιστρέψουμε' κατα $+120^o$ και -120^o ώστε να προκύψουν και οι τρεις ζητούμενες γωνίες θ_1, θ_2 και θ_3. Ονομάζουμε F_1 την άρθρωση του υπο μελέτη βραχίονα στη βάση του ρομπότ, E_1 την άρθρωση στο σημείο δράσης και J_1 την 'ελεύθερη' άρθρωση, εκεί δηλαδή όπου συνδέονται τα r_f και r_e. Λόγω κατασκευής το τμήμα F_1J_1 μπορεί να περιστραφεί μόνο στο YZ επίπεδο, σχηματίζοντας έτσι έναν κύκλο με κέντρο το F_1 και ακτίνα r_f. Αντίθετα, λόγω κατασκευής το τμήμα E_1J_1 μπορεί να κινηθεί ελεύθερα σε σχέση με το E_1 σημείο. Έτσι σχηματίζει μία σφαίρα με κέντρο το E_1 και ακτίνα r_e. Όλα αυτά μπορούν να αναπαρασταθούν σχηματικά ως εξής:



Εικόνα 13: Τρισδιάστατη απόδοση του σχήματος και των αντίστοιχων υπολογισμών.

Η προβολή της σφαίρας αυτής στο επίπεδο YZ είναι ένας καινούριος κύκλος με κέντρο E'_1 και ακτίνα E'_1J_1 οπου το E'_1 είναι η προβολή του E_1 στο επίπεδο YZ.



Εικόνα 14: Άποψη του επιπέδου YZ του ρομπότ

Πλέον αρκεί να βρούμε το σημείο J_1 ώστε να υπολογίσουμε τη γωνία θ_1. Το σημείο J_1 θα βρεθεί ως το σημείο τομής δύο κύκλων ($1^{ος}$ κύκλος με κέντρο E'_1 και ακτίνα E'_1J_1 και $2^{ος}$ κύκλος με κέντρο F_1 και ακτίνα F_1J_1) με γνωστό κέντρο και ακτίνα. Από τη λύση του συστήματος θα προκύψουν δύο σημεία και εμείς θα επιλέξουμε αυτό με τη μικρότερη y συνιστώσα.

Πιο αναλυτικά:

Θεωρώ το κέντρο του σημείου δράσης $E(x_0,y_0,z_0)$:

$EE'_1 = \frac{e}{2}\tan 30^\circ = \frac{e}{2\sqrt{3}}$

$E_1 (x_0, y_0 - \frac{e}{2\sqrt{3}}, z_0)$ άρα $E'_1 (0, y_0 - \frac{e}{2\sqrt{3}}, z_0)$

$E_1E'_1 = x_0$ και $E'_1J_1 = (E_1J_1^2 - E_1E'^2_1)^{1/2} = (r_e^2 - x_0^2)^{1/2}$

$F_1 (0, -\frac{f}{2\sqrt{3}}, 0)$

Άρα σύμφωνα με τα παραπάνω έχω:

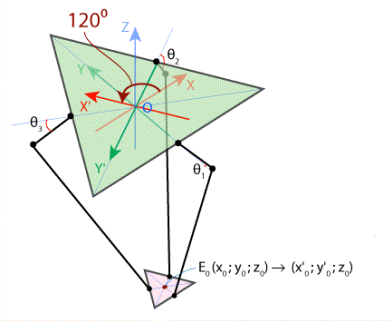
$(y_{J1} - y_{F1})^2 + (z_{J1} - z_{F1})^2 = r_f^2 \quad \Rightarrow \quad (y_{J1} + \frac{f}{2\sqrt{3}})^2 + z_{J1}^2 = r_f^2$

$(y_{J1} - y_{E'1})^2 + (z_{J1} - z_{E'1})^2 = r_e^2 - x_0^2 \quad \Rightarrow \quad (y_{J1} - y_0 + \frac{e}{2\sqrt{3}})^2 + (z_{J1} - z_0)^2 = r_e^2 - x_0^2$

Άρα από τα παραπάνω προκύπτει:

$J_1 (0, y_{J1}, z_{J1})$ και $\theta_1 = \arctan(\frac{ZJ1}{yF1 - yJ1})$

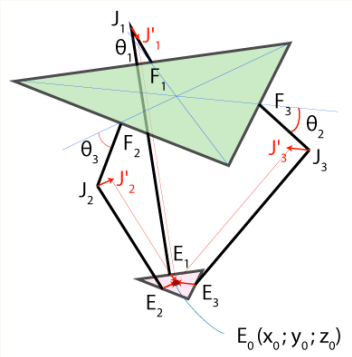
Πλέον έχοντας βρεί την γωνία θ_1, οι γωνίες θ_2 και θ_3 είναι πολύ εύκολο να βρεθούν λόγω συμμετρίας. Περιστρέφουμε τα αποτελέσματα μας κατα 120°. Απλά αλλάζουμε το επίπεδο αναφοράς βρίσκοντας νέες συντεταγμένες (x,y) για το σημείο E0. Έτσι $x' = x \cos 120^\circ + y \sin 120^\circ$ και $y' = -x \sin 120^\circ + y \cos 120^\circ$.



Εικόνα 15: Ενδεικτικό σχήμα για την περιστροφή της γωνίας.

0.4.2 Forward Kinematics – Ευθεία Κινηματική

Όπως είναι προφανές και από την ονομασία τους η Forward Kinematics είναι η αντίστροφη διαδικασία της Inverse Kinematics. Έχοντας δηλαδή σαν δεδομένο τις τρείς γωνίες θ_1, θ_2 και θ_3 θα υπολογίσουμε τις συντεταγμένες του κέντρου του σημείου δράσης. Γνωρίζοντας τις τρεις γωνίες είναι εύκολο να υπολογίσουμε τις συντεταγμένες των J_1, J_2, J_3.



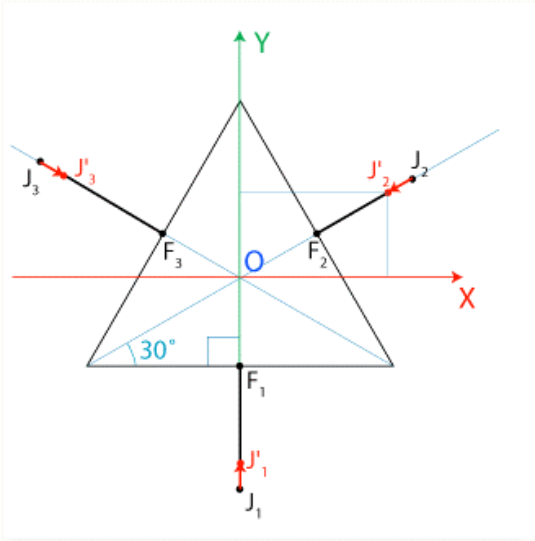
Εικόνα 16: Ευθεία κινηματική: Γνωρίζοντας τις 3 γωνίες, ψάχνουμε τις συντεταγμένες του E_0 (x_0, y_0, z_0).

Ακόμα, όπως αναφέρθηκε και πριν τα τμήματα E_1J_1, E_2J_2, E_3J_3 μπορούν να περιστραφούν ελεύθερα γύρω από τα σημεία J_1, J_2 και J_3 αντίστοιχα, σχηματίζοντας τρεις σφαίρες με κέντρο J_1, J_2, J_3 και ακτίνα r_e. Μεταφέροντας τα κέντρα των σχηματισμένων σφαιρών J_1, J_2, J_3 με τη βοήθεια των αντίστοιχων διανυσμάτων E_1E_0, E_2E_0 και E_3E_0 επιτυγχάνουμε οι τρεις σφαίρες να έχουν ως σημείο τομής το ζητούμενο σημείο E_0 (x_0, y_0,z_0). Άρα αρκεί να λύσουμε το σύστημα τριών εξισώσεων σφαιρών της μορφής $(x-x_i)^2+(y-y_i)^2+(z-z_i)^2=r_e^2$, όπου i=1, 2, 3 και τα κέντρα των τριών σφαιρών όπως και οι ακτίνες τους είναι γνωστά.

Έτσι, έχω:

$OF_1 = OF_2 = OF_3 = f/2 \tan30^o = f/2\sqrt{3}$
$J_1J'_1 = J_2J'_2 = J_3J'_3 = e/2 \tan30^o = e/2\sqrt{3}$
$F_1J_1 = r_f\cos\theta_1$,
$F_2J_2 = r_f\cos\theta_2$,
$F_3J_3 = r_f\cos\theta_3$

$J'_1(0, -(f-e)/2\sqrt{3}-r_f\cos\theta_1, -rf\sin\theta_1)$
$J'_2([(f-e)/2\sqrt{3} + r_f\cos\theta_2]\cos30^o, [(f-e)/2\sqrt{3} + r_f\cos\theta_2]\sin30^o, -r_f\sin\theta_2)$
$J'_3([(f-e)/2\sqrt{3} + r_f\cos\theta_3]\cos30^o, [(f-e)/2\sqrt{3} + r_f\cos\theta_3]\sin30^o, -r_f\sin\theta_3)$



Εικόνα 17: Το σημείο δράσης και οι αντίστοιχοι υπολογισμοί.

$X_2 + (y-y_1)^2 + (z-z_1)^2 = r_e^2$ $\quad x^2 + y^2 + z^2 - 2y_1y - 2z_1z = r_e^2 - y_1^2 - z_1^2$, (1)

$(x-x_2)^2 + (y-y_2)^2 + (z-z_2)^2 = r_e^2$ $\quad x^2 + y^2 + z^2 - 2x_2x - 2y_2y - 2z_2z = r_e^2 - x_2^2 - y_2^2 - z_2^2$, (2)

$(x-x_3)^2 + (y-y_3)^2 + (z-z_3)^2 = r_e^2$ $\quad x^2 + y^2 + z^2 - 2x_3x - 2y_3y - 2z_3z = r_e^2 - x_3^2 - y_3^2 - z_3^2$, (3)

$w_i = x_i^2 + y_i^2 + z_i^2$

Αφαιρώ μεταξύ τους τις (1) και (2),(1) και (3), (2) και (3):

(1) – (2): $x_2x + (y_1-y_2)y + (z_1 - z_2)z = (w_1 – w_2)/2$, (4)

(1) – (3): $x_3x + (y_1-y_3)y + (z_1 – z_3)z = (w_1 –w_3)/2$, (5)

(2) – (3): $(x_2-x_3)x + (y_2 - y_3)y + (z_2 – z_3)z = (w_2 –w_3)/2$, (6)

Από τα παραπάνω:

$x_1 = a_1z + b_1$ (7) και $\quad y = a_2z + b_2$ (8)

έτσι,

$a_1 = \frac{1}{d}[(z_2 – z_1)(y_3 – y_1) – (z_3 –z_1)(y_2 – y_1)]$ $\quad a_2 = -\frac{1}{d}[(z_2 – z_1)x_3 – (z_3 –z_1)x_2]$

$b_1 = -\frac{1}{2d}[(w_2 – w_1)(y_3 – y_1) – (w_3 – w_1)(y_2 – y_1)]$ $\quad b_2 = \frac{1}{2d}[(w_2 – w_1)x_3– (w_3 – w_1)x_2]$

$d = (y_2-y_1)x_3 – (y_3 – y_1)x_2$

Και τελικά από τις εξισώσεις (7), (8) και (1) προκύπτει το ζητούμενο σημείο:

$(a_1^2 + a_2^2 + 1)z^2 + 2(a_1 + a_2(b_2 – y_1) – z_1)z + (b_1^2 + (b_2 – y_1)^2 + z_1^2 – r_e^2 = 0.$

Πάνω στους παραπάνω μαθηματικούς υπολογισμούς βασίζεται ο προγραμματισμός του δικού μας Delta Robot.

1. Προσομοίωση στο Matlab

Το πρώτο βήμα που έγινε για τον προγραμματισμό του delta robot ήταν να φτιαχτούν στο Matlab δύο προγράμματα τα οποία θα υλοποιούσαν την αντίστροφη και την ευθεία κινηματική. Προφανώς τα προγράμματα αυτά βασίστηκαν στη μαθηματική ανάλυση που αναφέρθηκε στην εισαγωγή. Ακολουθεί ο κώδικας τον δύο αυτών συναρτήσεων:

1.1 Συνάρτησεις της Κινηματικής

Παρατίθενται αναλυτικά οι συναρτήσεις που υλοποιήθηκαν στο Matlab:

1.1.1 Inverse Kinematics:

Η συνάρτηση της αντίστροφης κινηματικής όπως προέκυψε στο Matlab:

```
function [theta1, theta2, theta3] = Inverse_Kinematics ( x0, y0, z0 )


 %%% (x0, y0, z0) -> (theta1,theta2,theta3)


 %%% Χρήσιμα :P

% %   sqrt3 = sqrt(3.0);
% %   sin120 = 0.866;
% %   cos120 = -0.5;

 %%% Thetas_calculate(x0,y0,z0 --> theta1,theta2,theta3)

 [status, theta1] = AngleYZ_calculate(x0, y0, z0);
 if (status == 0)
     [status, theta2 ] = AngleYZ_calculate(x0*(-0.5) + y0*(0.866),
y0*(-0.5) - x0*(0.866), z0);  % περιστροφή  +120 μοίρες
 end

 if (status == 0)
     [~, theta3 ] = AngleYZ_calculate(x0*(-0.5) - y0*(0.866), y0*(-
0.5) + x0*(0.866), z0);  % περιστροφή -120 μοίρες
 end


end
```

1.1.2 Συνάρτηση AngleYZ_calculate:

```
function   [ status, theta ] = AngleYZ_calculate( x0, y0, z0 ) %
Υπολογισμός της theta για το επίπεδο YZ

    %%% Διαστάσεις του ρομπότ

    e = 46.0;                   % Σημείο δράσης
    f = 116.0;                  % Βάση = σταθερή
    re = 167.0;                 % Μεγάλος βραχίονας
    rf = 56.0;                  % Μικρός βραχίονας
```

```
    pi = 3.141592653;            % γνωστό και ως π

    %%% κυρίως μέρος της συνάρτησης

    y1 = -0.5*0.57735*f;         % tan30=0.57735
    y0 = y0 - 0.5 * 0.57735 * e;

    a = (x0*x0 + y0*y0 + z0*z0 +rf*rf - re*re - y1*y1)/(2*z0);
    b = (y1-y0)/z0;
    d = -(a+b*y1)*(a+b*y1)+rf*(b*b*rf+rf);


    yz = (y1 - a*b - sqrt(d))/(b*b + 1);
    zz = a + b*yz;

    if d >= 0;

    if yz>y1
         theta = 180.0*(atan(-zz/(y1 - yz)))/pi + 180.0;
     else
         theta = 180.0*(atan(-zz/(y1 - yz)))/pi;
    end

    if theta>60
         theta = 'Άκυρη θέση – γωνία μεγαλύτερη των 60 μοιρών!';
    end

     if theta<-100
          theta = 'Άκυρη θέση – γωνία μικρότερη των -100 μοιρών!';
     end

    status = 0;

    else
    status = -1;
    theta = 'άκυρη θέση – αρνητική διακρίνουσα → φανταστικό νούμερο
(imaginary number)!';
    end

    return;

 end
```

Η παραπάνω συνάρτηση είναι βοηθητική. Όπως αναφέραμε στη κινηματική του Delta Robot για να βρούμε τις τρεις γωνίες θ1, θ2, θ3 της αντίστροφης κινηματικής αρκεί να βρούμε τη θ1 και μετα να 'περιστρέψουμε' τους άξονες κατα $+120^{o}$ και -120^{o} για τις θ2, θ3. Αυτό το σκοπό επιτελεί η AngleYZ_calculate. Στην συνάρτηση Inverse_Kinematics την καλούμε τρεις φορές – μία για κάθε γωνία.

Εν τέλει τρέχοντας την Inverse_Kinematics στο Matlab με συγκεκριμένες συντεταγμένες (x, y, z), μας επέστρεφε τις αντίστοιχες τιμές για κάθε γωνία των τριών βραχιόνων γωνίες (theta1, theta2, theta3):

[theta1, theta2, theta3] = Inverse_Kinematics (50, 60 , -120)

theta1 =

28.7199

theta2 =

-51.4594

theta3 =

1.8904

1.1.3 Συνάρτηση Forward Kinematics:

```
function [ x0, y0, z0 ] = Forward_Kinematics ( theta1, theta2, theta3
)

%%% Διαστάσεις ρομπότ

 e = 46.0;              % Σημείο δράσης
 f = 116.3;             % Βάση = σταθερή
 re = 167.0;            % Μεγάλος βραχίονας
 rf = 56.0;             % Μικρός βραχίονας

 %%% Χρήσιμα :P

 sqrt3 = sqrt(3.0);
```

```
pi = 3.141592653;      % γνωστό και ως π
sin120 = sqrt3/2.0;
cos120 = -0.5;
tan60 = sqrt3;
sin30 = 0.5;
tan30 = 1/sqrt3;

%%% ( theta1, theta2, theta3 -> x0, y0, z0)

t = (f-e)*tan30/2;
dtr = pi/180.0;

theta1 = theta1*dtr;
theta2 = theta2*dtr;
theta3 = theta3*dtr;

%%% Όρια για τις δεχόμενες γωνίες

%   if theta1>60 || theta2>60 || theta3>60
%          theta = 'Άκυρη θέση – γωνία μεγαλύτερη των 60 μοίρων!';
%     end
%
%      if theta1<-100 || theta2<-100 || theta3<-100
%           theta = 'Άκυρη θέση - γωνία μικρότερη των -100 μοίρων!';
%      end

y1 = -(t + rf*cos(theta1));
z1 = -rf*sin(theta1);

y2 = (t + rf*cos(theta2))*sin30;
x2 = y2*tan60;
z2 = -rf*sin(theta2);

y3 = (t + rf*cos(theta3))*sin30;
x3 = -y3*tan60;
z3 = -rf*sin(theta3);

dnm = (y2-y1)*x3-(y3-y1)*x2;

w1 = y1*y1 + z1*z1;
w2 = x2*x2 + y2*y2 + z2*z2;
w3 = x3*x3 + y3*y3 + z3*z3;

%%%  x = (a1*z + b1)/dnm
a1 = (z2-z1)*(y3-y1)-(z3-z1)*(y2-y1);
b1 = -((w2-w1)*(y3-y1)-(w3-w1)*(y2-y1))/2.0;

%%% y = (a2*z + b2)/dnm;
a2 = -(z2-z1)*x3+(z3-z1)*x2;
b2 = ((w2-w1)*x3 - (w3-w1)*x2)/2.0;

%%% εξίσωση a*z^2 + b*z + c = 0
a = a1*a1 + a2*a2 + dnm*dnm;
b = 2*(a1*b1 + a2*(b2-y1*dnm) - z1*dnm*dnm);
c = (b2-y1*dnm)*(b2-y1*dnm) + b1*b1 + dnm*dnm*(z1*z1 - re*re);

%%% διακρίνουσα
d = b*b - 4.0*a*c;
```

```
 if d < 0
     return; % ανύπαρκτο σημείο
 end
 z0 = -0.5*(b+sqrt(d))/a;
 x0 = (a1*z0 + b1)/dnm;
 y0 = (a2*z0 + b2)/dnm;



end
```

Τρέχοντας την Forward Kinematics με παραμέτρους συγκεκριμένες γωνίες (theta1, theta2, theta3), μας επέστρεφε τις αντίστοιχες τιμές συντεταγμένων για το σημείο δράσης (x, y, z) (αντίστοιχη διαδικάσία με αυτή της Inverse_Kinematics):

```
>>  [ x0, y0, z0 ] = Forward_Kinematics ( 60, 45, 10 )


x0 =


 -43.7827




y0 =


  51.9069




z0 =


 -174.7190
```

1.2 Συνολική προσομοίωση της κίνησης

Μία συνολική προσομοίωση της κίνησης του ρομπότ έγινε με τις συναρτήσεις demo.m και Deltasimulate.m. Αρκεί λοιπόν ο οποιοσδήποτε να τρέξει το demo.m από το Matlab για να δει το ρομπότ να επιτελεί συγκεκριμένες κινήσεις στο χώρο.

Παρατίθεται κομμάτι του κώδικα της συνάρτησης DeltaSimulate.m:

```
if (WS==1)
     % Φτιάξε την βάση του τριγώνου.
     plot3     ([0,f/2,-f/2,0]     ,     [f/sqrt(3),-f/(2*sqrt(3)),-
f/(2*sqrt(3)),f/sqrt(3)] , [0,0,0,0] ,'r');
     % Φτιάξε τις 3 διαγωνίους (αυτό θα βοηθήσει να επιλέξουμε τις
σωστές γωνίες).
     plot3  ([-f/2,f/4]  ,  [-f/(2*sqrt(3)),f/(4*sqrt(3))]  ,  [0,0]
,'r');
     plot3  ([f/2,-f/4]  ,  [-f/(2*sqrt(3)),f/(4*sqrt(3))]  ,  [0,0]
,'r');
     plot3 ([0,0] , [f/sqrt(3),-f/(2*sqrt(3))] , [0,0] ,'r');
     % Φτιάξε τον πρώτο βραχίονα.
     plot3  ([0,0,x]  ,  [-f/(2*sqrt(3)),-rf*cos(q1)-f/(2*sqrt(3)),y-
e/(2*sqrt(3))] , [0,rf*sin(q1),z] ,'g');
     % Φτιάξε τον δεύτερο βραχίονα.
     plot3          ([f/4,f/4+rf*cos(q2)*sqrt(3)/2,x+e/4]          ,
[f/(4*sqrt(3)),f/(4*sqrt(3))+rf*cos(q2)/2,y+e/(4*sqrt(3))]          ,
[0,rf*sin(q2),z] ,'g');
     % Φτιάξε τον τρίτο βραχίονα.
     plot3         ([-f/4,-f/4-cos(q3)*sqrt(3)*rf/2,x-e/4]          ,
[f/(4*sqrt(3)),f/(4*sqrt(3))+rf*cos(q3)/2,y+e/(4*sqrt(3))]          ,
[0,rf*sin(q3),z] ,'g');
     % Φτιάξε το σημείο δράσης.
     plot3   ([x,x+e/2,x-e/2,x]   ,   [y+e/sqrt(3),y-e/(2*sqrt(3)),y-
e/(2*sqrt(3)),y+e/sqrt(3)] , [z,z,z,z] ,'b') ;
     % Φτιάξε τις 3 διαγωνίους.
     plot3 ([x-e/2,x+e/4] , [y-e/(2*sqrt(3)),y+e/(4*sqrt(3))] , [z,z]
,'b');
     plot3 ([x+e/2,x-e/4] , [y-e/(2*sqrt(3)),y+e/(4*sqrt(3))] , [z,z]
,'b');
     plot3 ([x,x] , [y+e/sqrt(3),y-e/(2*sqrt(3))] , [z,z] ,'b');
```

1.3 Το πεδίο δράσης (Work Space) του ρομπότ

Πολύ σημαντικό για την υλοποίηση του ρομπότ ήταν να υπολογιστεί το ύψος στο οποίο η ηλεκτρομαγνητική του αρπάγη έχει τη μεγαλύτερη ευχέρεια κινήσεων, δηλαδή εκεί που μεγιστοποιείται το εμβαδό που μπορεί να εξυπηρετήσει η αρπάγη. Αυτό ακριβώς είναι το ‘εύρος εργασίας’ ή Work Envelope. Σε αυτό το ύψος φυσικά θα τοποθετηθεί η σκακιέρα μας, έτσι ώστε το ρομπότ να μην αντιμετωπίζει προβλήματα σε ‘ακραίες’ θέσεις ή κινήσεις. Σημαντικό είναι να αναφερθεί πως ζητούμενο δεν ήταν να καλύπτεται ακριβώς όλη η διάσταση της σκακιέρας, διότι το ρομπότ πρέπει να έχει πρόσβαση και σε λίγο χώρο παραπάνω ώστε να αφήνει τα πούλια που τρώγονται.

Οι σερβοκινητήρες έχουν θεωρητικά άνοιγμα κοντά στις 180^{o} ο καθένας όμως πρακτικά είναι λιγότερο. Εμείς θεωρήσαμε ως μέγιστη γωνία (με επίπεδο αναφοράς τη βάση του ρομπότ) +40^{o} (θετική φορά) και ως ελάχιστη γωνία -100^{o} (αρνητική φορά). Όπως θα δούμε και παρακάτω οι τιμές αυτές είναι πολύ κοντά στις πραγματικές .

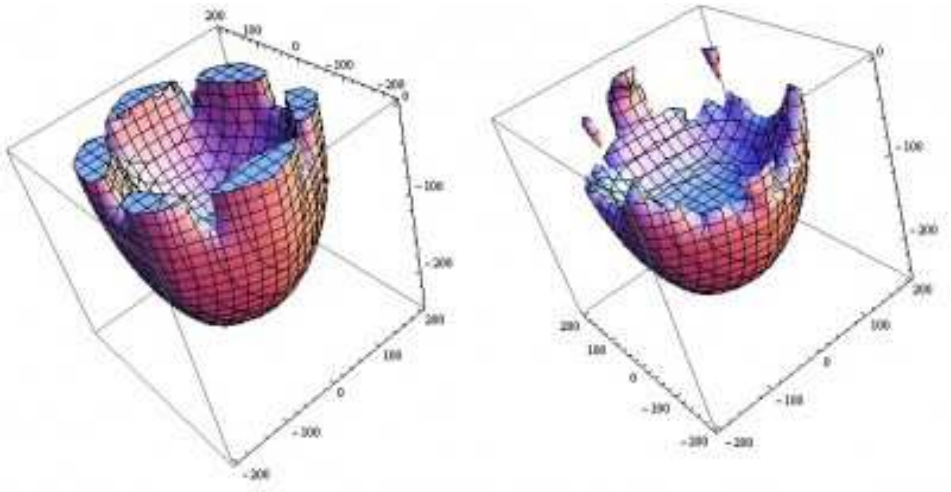
1.3.1 Η πρώτη υλοποίηση του πεδίου δράσης

Έτσι λοιπόν επόμενο βήμα ήταν να κάνουμε μια προσομοίωση της κίνησης του ρομπότ ώστε να δούμε πως θα ανταποκρίνεται στις πραγματικές συνθήκες. Τρέχοντας λοιπόν τον παρακάτω κώδικα στο Mathematica, πήραμε μια πρώτη άποψη για το πως ανταποκρίνονται οι τρεις βραχίονες του ρομπότ στο χώρο.

```
Γεωμετρία ρομπότ;
delta = {f → 94.0, e → 57.3, rf → 56, re → 166.5};
stop1 = 60;
stop2 = -110;

Αντίστροφη κινηματική ανάλυση;
y1 = -f/(2√3);

y0 = Y - e/(2√3);

a = (X^2 + y0^2 + Z^2 + rf^2 - re^2 - y1^2)/(2 Z);
b = (y1 - y0)/Z;

d1 = rf^2 * (1 + b^2) - (a + b * y1)^2;
yj = (y1 - a * b - √d1)/(1 + b^2);
zj = a + b * yj;

theta1 = ArcSin[zj/rf] * 180/π;

a120 = 2 π / 3;
turn1 = {X → X * Cos[a120] + Y * Sin[a120], Y → Y * Cos[a120] - X * Sin[a120]};
turn2 = {X → X * Cos[a120] - Y * Sin[a120], Y → Y * Cos[a120] + X * Sin[a120]};

theta2 = theta1 /. turn1;
theta3 = theta1 /. turn2;
d2 = d1 /. turn1;
d3 = d1 /. turn2;


wstop = theta1 <= stop1 && theta1 >= stop2 && theta2 <= stop1 && theta2 >= stop2 && theta3 <= stop1 && theta3 >= stop2 ;
ws = d1 ≥ 0 && d2 ≥ 0 && d3 ≥ 0;
ws0 = ws && wstop;

RegionPlot3D[ws /. delta, {X, -200, 200}, {Y, -200, 200}, {Z, -250, -0.1}]
RegionPlot3D[ws0 /. delta, {X, -200, 200}, {Y, -200, 200}, {Z, -250, -0.1}]
```

Εικόνα 18: Ο κώδικας του Mathematica.

Από τον παραπάνω κώδικα προέκυψαν τα παρακάτω γραφήματα:



Εικόνα 19: Το work space του delta robot. Στην αριστερή εικόνα δεν υπάρχουν οι περιορισμοί μέγιστης - ελάχιστης γωνίας των σερβοκινητήρων.

1.3.2 Η δεύτερη υλοποίηση του πεδίου δράσης

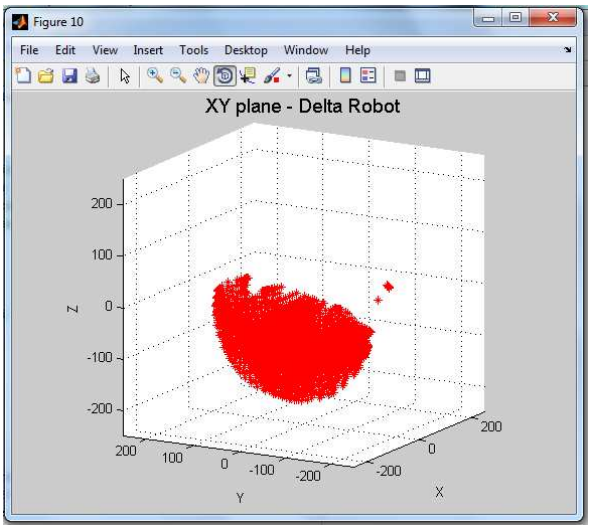
Έπειτα τρέξαμε στο Matlab τον κώδικα της Forward_Kinematics με 3 loops για να σχεδιάσουμε το εύρος της κίνησης σε τρεις διαστάσεις και με βήμα 15 μοίρες. Στο τέλος σχεδιάζεται το ζητούμενο ζημείο ως ένας κόκκινος σταυρός.

```
function [ x0, y0, z0 ] = Best_XY_plane_Forward_Kin ( ~, ~, ~ )

%%% Διαστάσεις ρομπότ

 e = 61;                 % Σημείο δράσης
 f = 92;                 % Βάση = σταθερή
 re = 167.0;             % Μεγάλος βραχίονας
 rf = 56.0;              % Μικρός βραχίονας

 %%% Χρήσιμα :P

 sqrt3 = sqrt(3.0);
 pi = 3.141592653;       % γνωστό και ως π
 %sin120 = sqrt3/2.0;
 %cos120 = -0.5;
 tan60 = sqrt3;
 sin30 = 0.5;
 tan30 = 1/sqrt3;

 %%% Περί γραφικής παράστασης...
 figure(10);
 title('XY plane - Delta Robot','FontSize',14)
 xlabel('X'); ylabel('Y'); zlabel('Z');
 axis([-250 250 -250 250 -250 250]);
 axis manual; axis equal;
 grid on;
```

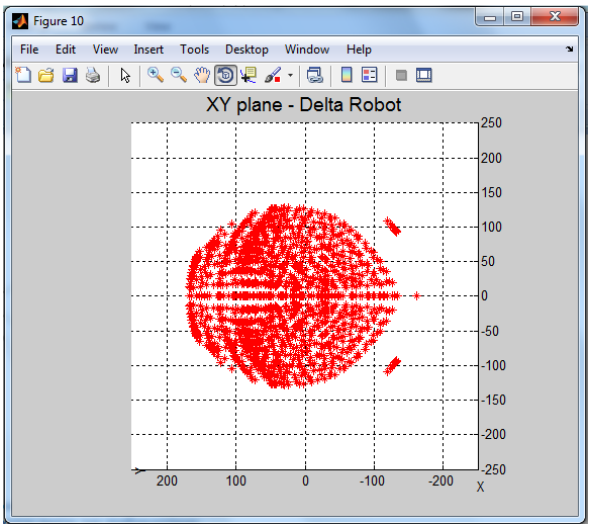
```
hold on;

%%% ( theta1, theta2, theta3 -> x0, y0, z0)


  for theta1 = -100:15:55           %%% βλέπε όρια κίνησης σέρβο και
χώρος εργασίας
       for theta2 = -100:15:55
          for theta3 = -100:15:55

t = (f-e)*tan30/2;
dtr = pi/180.0;

thet1 = theta1*dtr;
thet2 = theta2*dtr;
thet3 = theta3*dtr;

y1 = -(t + rf*cos(thet1));
z1 = -rf*sin(theta1);

y2 = (t + rf*cos(thet2))*sin30;
x2 = y2*tan60;
z2 = -rf*sin(thet2);

y3 = (t + rf*cos(thet3))*sin30;
x3 = -y3*tan60;
z3 = -rf*sin(thet3);

dnm = (y2-y1).*x3-(y3-y1).*x2;

w1 = y1.*y1 + z1.*z1;
w2 = x2.*x2 + y2.*y2 + z2.*z2;
w3 = x3.*x3 + y3.*y3 + z3.*z3;

%%%  x = (a1*z + b1)/dnm
a1 = (z2-z1).*(y3-y1)-(z3-z1).*(y2-y1);
b1 = -((w2-w1).*(y3-y1)-(w3-w1).*(y2-y1))/2.0;

%%% y = (a2*z + b2)/dnm;
a2 = -(z2-z1).*x3+(z3-z1).*x2;
b2 = ((w2-w1).*x3 - (w3-w1).*x2)/2.0;

%%% εξίσωση a*z^2 + b*z + c = 0
a = a1.*a1 + a2.*a2 + dnm.*dnm;
b = 2*(a1.*b1 + a2.*(b2-y1.*dnm) - z1.*dnm.*dnm);
c   =   (b2-y1.*dnm).*(b2-y1.*dnm)   +   b1.*b1   +   dnm.*dnm.*(z1.*z1   -
re.*re);

%%% διακρίνουσα
d = b.*b - 4.0*a.*c;

if d < 0
    return; % ανύπαρκτο σημείο
end
z0 = -0.5*(b+sqrt(d))/a;
x0 = (a1.*z0 + b1)/dnm;
y0 = (a2.*z0 + b2)/dnm;
```

```
plot3( x0, y0, z0, 'r*');
grid on
          end
       end
   end

end
```

Το αποτέλεσμα ήταν το παρακάτω γράφημα το οποίο όμως δεν αρκούσε για να βγάλουμε τα συμπεράσματα μας, καθώς βασικό ζητούμενο ήταν να έχουμε με μεγάλη ακρίβεια το ύψος z στο οποίο το ρομπότ θα είχε το μέγιστο εύρος εργασίας, και να γνωρίζουμε ότι όντως μπορεί να εξυπηρετήσει όλα τα σημεία εντός του χώρου εκείνου (λόγω των περιορισμών των γωνιών σε κάποια ύψη το ρομπότ ενώ μπορεί να 'εξυπηρετήσει' ακραίες – μακρινές θέσεις, αντιθέτως δεν μπορεί να ανταπεξέλθει σε 'εσωτερικές').



Εικόνα 20: 3d plane



Εικόνα 21: XY plane

1.3.3 Η τρίτη υλοποίηση του πεδίου δράσης

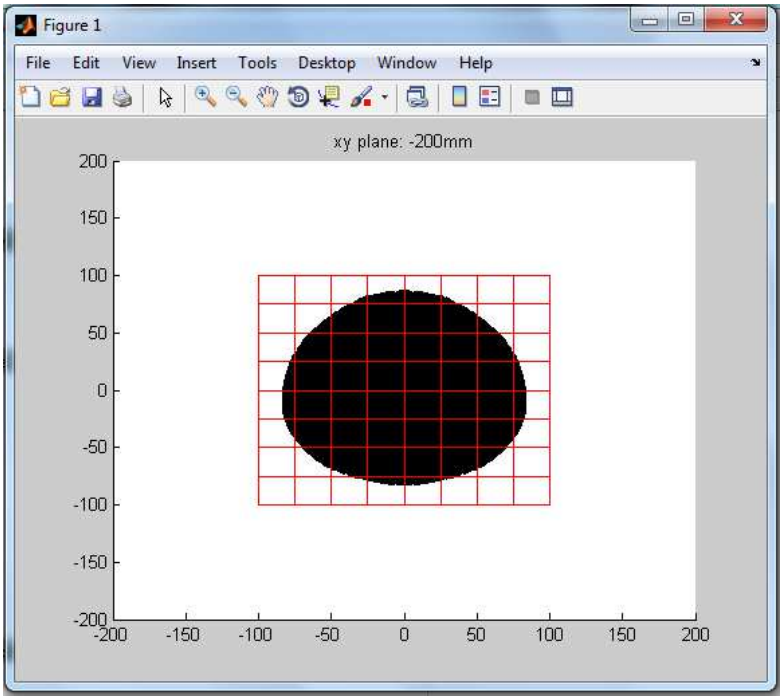
1.3.3.1 Σε δύο διαστάσεις (2D)

Η νέα προσομοίωση έγινε και αυτή στο Matlab και υλοποιήθηκε τόσο σε δισδιάστατο γράφημα, όσο και σε τρισδιάστατο. Ο κώδικας για το δισδιάστατο είναι ο εξής:

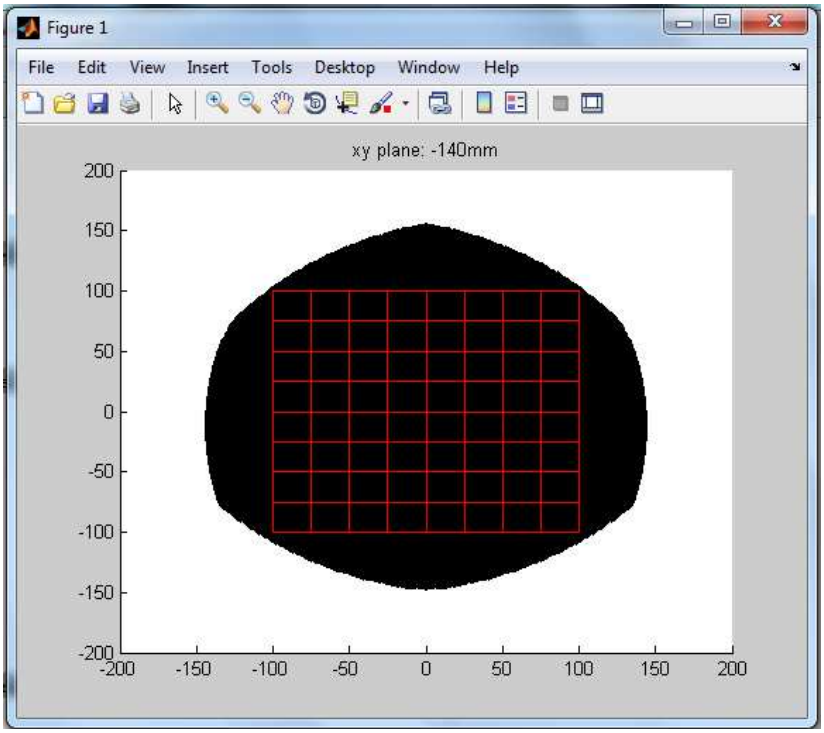
```
% Αυτό το πρόγραμμα θα παρουσιάσει το πεδίο δράσης του ρομπότ σε 2D αναπράσταση.
clear all
clc

% Φτιάξε ένα Delta Robot με διαστάσεις
% f = 94 mm
% e = 57.3 mm (υπολογισμένο ως e = e' + d*2sqrt(3), όπου e' = μετρήθηκε και d η απόσταση του συνδέσμου από το e)
% rf = 56 mm
% re = 167 mm (μετρήθηκε μεταξύ 166 - 167)
% umax = 60 μοίρες
% umin = -110 μοίρες
delta = CreateDelta(101,61,56,166,[40 -100],5);

% Συμπέρασμα: Ακόμα και όταν θmax =40 και θmin=-90
% μπορούμε να καλύψουμε τις επιθυμητές διαστάσεις xy 200mm x 200mm (εννοώντας -100=<x<=100
% και -100=<y<=100).
% Για να επιτευχθεί αυτό, το επιθυμητό ύψος είναι z = -150mm

% Φτιάξε πίνακες
M=400;
x=ones(M);
for i=1:M
 x(:,i)=x(:,i)'.*linspace(-200,200,M);
end
y=x';

% Τρέξε τη λούπα (loop).
% Κάθε πλάνο είναι 10mm ψηλότερα από το προηγούμενο.
% Ξεκινάμε από τα 250 mm.
z=-250*ones(M);
grid on
xlabel('x axis');
ylabel('y axis');

chess = 8;
step = 200/chess;
for i=1:25
   clf
   z=z+10;
   [Q1 Q2 Q3 Q4 Q5 Q6 BW]=DeltaInverse(delta,x,y,z);
   axis ([-200 200 -200 200]);
   hold on;
   title(['xy plane: ' int2str(z(1,1)) 'mm']);
   X=x.*BW;
   Y=y.*BW;
   plot (X ,Y, 'k.');
   % Ζωγράφισε το επιθυμητό τετράγωνο (ουσιαστικά είναι η σκακιέρα).
   % Επιθυμητές διαστάσεις xy 200mm x 200mm
   plot([-100 -100 100 100 -100],[-100 100 100 -100 -100],'r-')
```

```
    for j=0:chess-1
      plot([-100+j*step -100+j*step],[-100 100],'r-')
    end
    for j=0:chess-1
      plot([-100 100],[-100+j*step -100+j*step],'r-')
    end
    % Κάνε παύση για λίγο για να δούμε το αποτέλεσμα.
    pause(0.05);
end
```

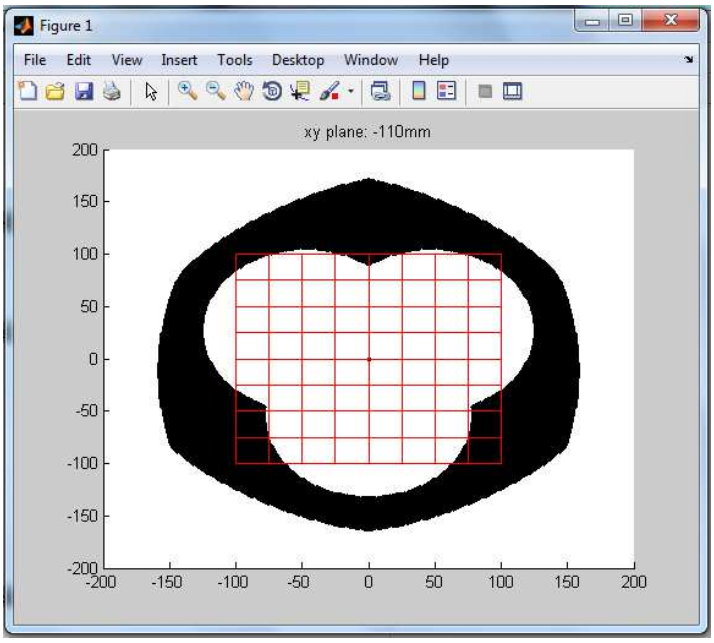
Το γράφημα το οποίο προκύπτει είναι δυναμικό. Το πρόγραμμα σκανάρει όλα τα ύψη από z = -220 mm έως z = 0 mm (υπενθύμιση: το z είναι αρνητικό αφού επίπεδο αναφοράς είναι το επίπεδο της βάσης του ρομπότ) με βήμα 10mm και έπειτα αποτυπώνει στο γράφημα με μαύρο χρώμα τα σημεία τα οποία βρίσκονται εντός του χώρου εργασίας του ρομπότ, και με άσπρο χρώμα τα σημεία εκτός του χώρου αυτού. Επίσης στο γράφημα αποτυπώνεται και η σκακιέρα στις πραγματικές τις διαστάσεις ((20x20 mm) και στην αντίστοιχη θέση. Η άποψη του διαγράμματος είναι κάθετη (από τον άξονα z). Παρακάτω παρατίθενται κάποιες εικόνες από το γράφημα (αρκεί να τρέξετε το αρχείο workspace.m):



Εικόνα 22: Το εύρος εργασίας του ρομπότ για z = -200mm. Όπως παρατηρούμε το ρομπότ δεν μπορεί να καλύψει όλα τα τετράγωνα της σκακιέρας.



Εικόνα 23: Το ζητούμενο ύψος z = -140mm. Το ρομπότ μπορεί να καλύψει όλα τα τετράγωνα της σκακιέρας.



Εικόνα 24: Εδώ είναι εμφανές το πρόβλημα που αναφέρθηκε παραπάνω. Ενώ το ρομπότ έχει πρόσβαση σε ακραίες θέσεις, αντιθέτως δεν μπορεί να ανταποκριθεί στις εσωτερικές.

Παρατηρήσεις - Σχόλια: Καταρχάς τα αποτελέσματα των γραφημάτων ήταν αναμενόμενα. Είναι απολύτως λογικό όταν οι βραχίονες του ρομπότ είναι σε πλήρη έκταση ή αντίθετα υπερβολικά κοντά στη βάση, η αρπάγη να είναι 'δυσκίνητη' και να μην μπορεί να εξυπηρετήσει μεγάλο εμβαδό. Αντιθέτως σε πιο 'εύκολα' ύψη, όπου οι βραχίονες του ρομπότ δεν 'τεντώνουν', η αρπάγη έχει μεγαλύτερο έυρος κινήσεων. Όπως θα δούμε και στα τρισδιάστατα γραφήματα, το σχήμα που η αρπάγη σχηματίζει στο χώρο είναι ελλειψοειδές.

Επίσης, εάν τρέξει κάποιος το workspace.m θα παρατηρήσει ότι το ύψος z = - 140 mm είναι το πρώτο στο οποίο καλύπτεται η σκακιέρα σε όλη τη διάστασή της (και λίγο παραπάνω όπως θέλουμε). Όμως τα κριτήρια αυτά καλύπτονται και για z = - 130 mm ή z = - 120 mm. Άρα λοιπόν γιατί επιλέχθηκε να μέγιστοποιείται η απόσταση σκακιέρας – βάσης; Η απάντηση είναι απλή καθώς το ρομπότ θα αλληλεπιδρά με τον αντίπαλο παίχτη και άρα ήταν απαραίτητο να υπάρχει όσο το δυνατόν περισσότερος χώρος για να κάνει αυτός την κίνηση του.

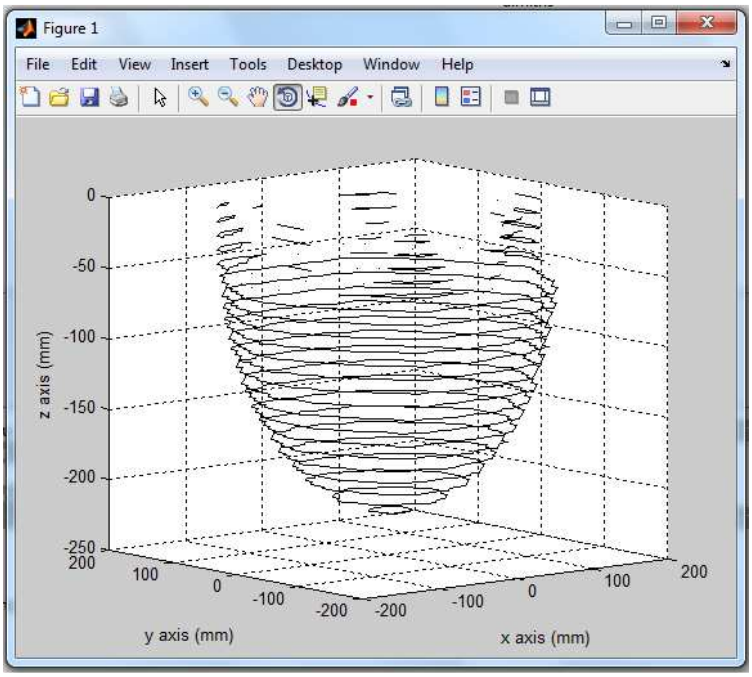
1.3.3.2 Σε τρεις διαστάσεις (3D)

Η προσομοίωση έγινε και στις τρεις διαστάσεις. Ο κώδικας προέκυψε με κάποιες μετατροπές ως εξής:

```
% Το πρόγραμμα αυτό θα παρουσιάσει το πεδίο δράσης του ρομπότ σε 3D αναπαράσταση.
clear all
clc

% Φτιάξε ένα Delta Robot με διαστάσεις:
% f = 94 mm
% e = 57.3 mm (υπολογισμένο ως e = e' + d*2sqrt(3), όπου e' = μετρήθηκε και d είναι η απόσταση του συνδέσμου από το e)
% rf = 56 mm
% re = 167 mm (μετρήθηκε κάπου μεταξύ 166 - 167)
% umax = 60 μοίρες
% umin = -110 μοίρες
delta = CreateDelta(94,57.3,56,166.5,[60 -110]);

% Συμπέρασμα: Ακόμα και όταν θmax =40 και θmin=-90
% καλύπτουμε τις επιθυμητές διαστάσεις σε xy 200mm x 200mm (εννοώντας -100=<x<=100
% και -100=<y<=100).
% Για να επιτευχθεί αυτό, το επιθυμητό ύψος είναι z = -150mm.

% Φτιάχνω πίνακες
xypoints = 10;
zpoints = 10;
zpanel_start=-250;
xyaxis_min = -200;
xyaxis_max = 200;
zaxis_min =-250;
zaxis_max =0;


xx=xyaxis_min:xypoints:xyaxis_max;
M=length(xx);
x=ones(M);

for i=1:M
 x(:,i)=x(:,i)'.*xx;
end
y=x';

z=zpanel_start*ones(M);
```

```
% Τρέξε τη λούπα (loop).
% Κάθε πλάνο είναι 10mm ψηλότερα από το προηγούμενο.
% Ξεκινάμε από τα 250 mm.
colors=['b' 'g' 'r' 'c' 'm' 'y'];
grid on
xlabel('x axis (mm)');
ylabel('y axis (mm)');
zlabel('z axis (mm)');

i=zpanel_start;
while (i<0)
   z=z+zpoints;
   [Q1 Q2 Q3 Q4 Q5 Q6 BW]=DeltaInverse(delta,x,y,z);

   [B,L] = bwboundaries(BW,'noholes');

%    cidx = mod(i,length(colors))+1;
   for k = 1:length(B)
      boundary = B{k};
      axis ([xyaxis_min xyaxis_max xyaxis_min xyaxis_max zaxis_min
zaxis_max]);
      hold on
      XX=boundary(:,2)*xypoints-xyaxis_max;
      YY=boundary(:,1)*xypoints-xyaxis_max;
      ZZ=z(1,1)*ones(1,length(boundary(:,1)));
      plot3(XX, YY,ZZ, 'k')
   end

   i=i+zpoints;
end
```

Και προέκυψε το εξής γράφημα (αρκεί να τρέξετε το αρχείο workspace3d.m):



Εικόνα 25: Το τρισδιάστατο γράφημα όπου είναι εμφανές το ελλειψοηδές σχήμα

1.3.4 CreateDelta και DeltaSimulate

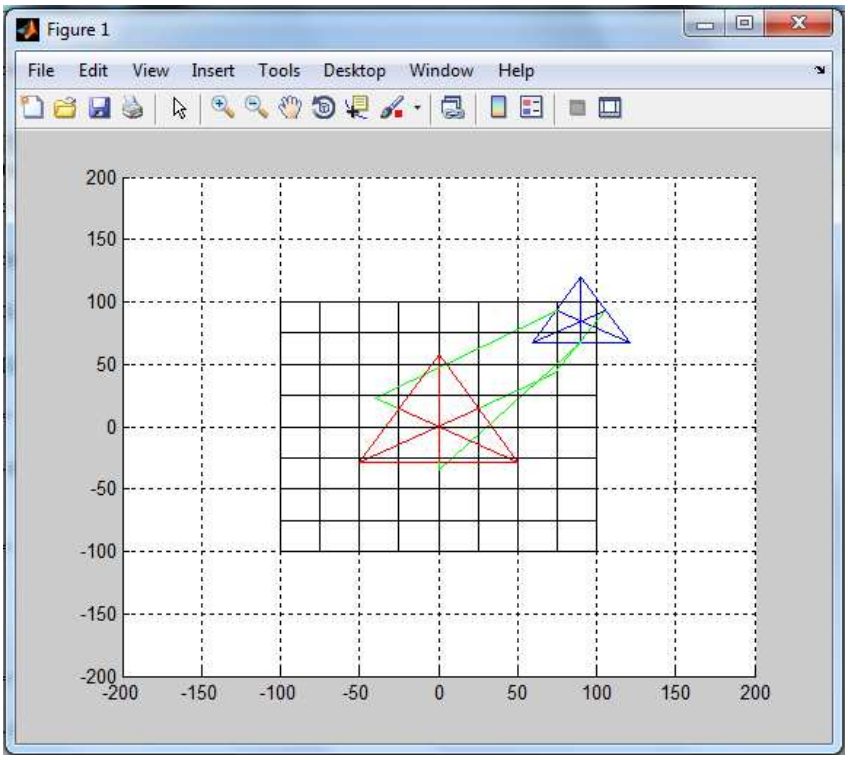
Ο τελευταίος έλεγχος, και αυτός με τη βοήθεια του Matlab, έγινε τρέχοντας τα αρχεία CreatDelta.m και DeltaSimulate.m.

Πρώτα τρέχουμε την εντολή delta = CreateDelta(101,61,56,166,[40 -100],6), όπου οι αριθμοί στην παρένθεση είναι οι διαστάσεις του ρομπότ αντίστοιχα (f, e, r_f, r_e,[θ_{min}, θ_{max}], β), όπου β το βήμα των κινητήρων σε μοίρες (^{o}). Αμέσως μετά τρέχουμε την εντολή [Q1,Q2,Q3,x,y,z,WS]=DeltaSimulate(delta,x,y,z,0), όπου z = -150mm (το ύψος της σκακιέρας), (x,y) οι συντεταγμένες του εκάστοτε σημείου και WS η μεταβλητή που μας λέει εάν η συγκεκριμένη θέση είναι έγκυρη και ανήκει στο workspace (σε αυτή την περίπτωση επιστρέφει 1 αλλιώς 0). Η τελευταία παράμετρος που είναι μηδενική εφόσον θέλουμε διάγραμμα δισδιάστατο (2D) και 1 εφόσον θέλουμε το διάγραμμα τρισδιάστατο (3d). Έτσι λοιπόν:

```
>> delta = CreateDelta(101,61,56,166,[40 -100],6)

delta =

  101.0000   61.0000   56.0000  166.0000    0.6981   -1.7453    6.0000

>> [Q1,Q2,Q3,x,y,z,WS]=DeltaSimulate(delta,87,87,-150,0)

Q1 =

   -84

Q2 =

     6

Q3 =

   -72

x =

   90.2559

y =

   84.4641

z =

 -150.7377

WS =
```

1

Το αντίστοιχο διάγραμμα:



Εικόνα 26: Κάτοψη της κίνησης του ρομπότ - Δισδιάστατο σχήμα.

Και αλλάζοντας την τελευταία παράμετρο σε 1:

```
>> [Q1,Q2,Q3,x,y,z,WS]=DeltaSimulate(delta,87,87,-150,1)

Q1 =

   -84

Q2 =

     6

Q3 =

   -72

x =

   90.2559
```

y =

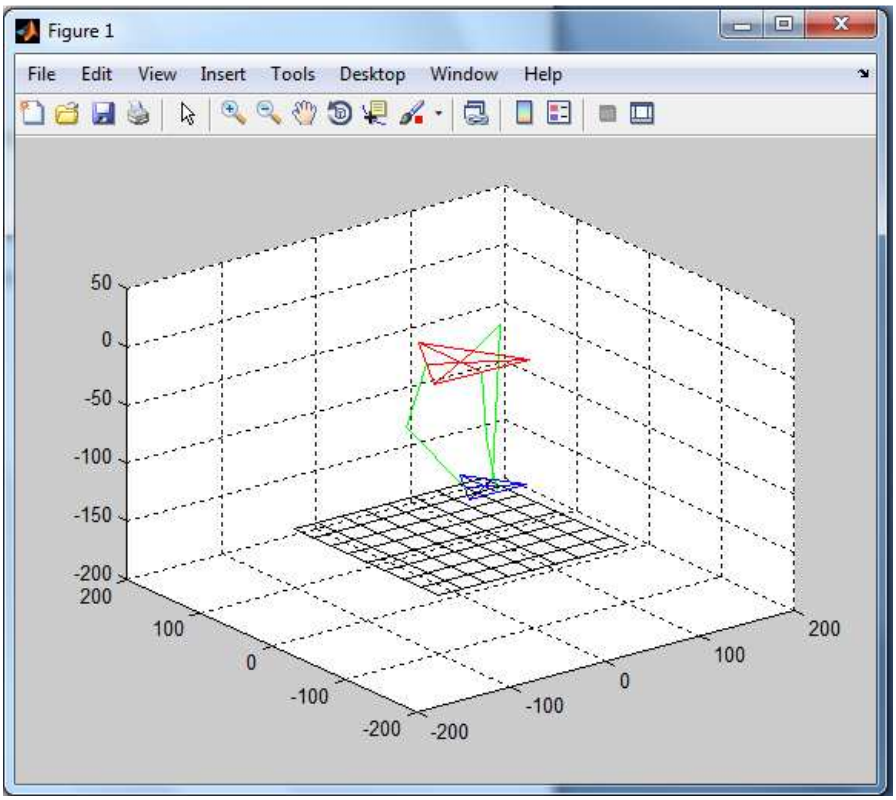
84.4641

z =

-150.7377

WS = 1

Και το αντίστοιχο διάγραμμα:



Εικόνα 27: Τρισδιάστατη άποψη της κίνησης του ρομπότ.

2. Visual Κώδικας – Προγραμματισμός της Τεχνητής Νοημοσύνης του ρομπότ.

2.1 Τεχνητή Νοημοσύνη

Η Τεχνητή Νοημοσύνη (Artificial Intelligence) ορίζεται ως ο τομέας της επιστήμης υπολογιστών που ασχολείται με τη σχεδίαση και την υλοποίηση υπολογιστικών συστημάτων που μιμούνται στοιχεία της ανθρώπινης συμπεριφοράς, τα οποία υπονοούν έστω και στοιχειώδη ευφυΐα: μάθηση, προσαρμοστικότητα, εξαγωγή συμπερασμάτων, κατανόηση από συμφραζόμενα ή επίλυση προβλημάτων. Ο Τζον Μακάρθι όρισε τον τομέα αυτόν ως «επιστήμη και μεθοδολογία της δημιουργίας νοούντων μηχανών».(Wikipedia)

Η Τεχνητή Νοημοσύνη δημιουργείται με τη βοήθεια αλγορίθμων και με τη θέσπιση αριθμητικών μοντέλων και κανόνων, τα οποία δίνουν στη μηχανή τη δυνατότητα να προσομοιώνει λειτουργίες του ανθρώπινου εγκεφάλου.

Η ιστορία της Τ.Ν. ξεκινάει τη δεκαετία του 1940 με τη πρώτη μαθηματική περιγραφή ενός νευρωνικού δικτύου με περιορισμένες δυνατότητες και φτάνει μέχρι σήμερα όπου στην αγορά κυκλοφορούν αυτόνομα κατοικίδια ζώα-ρομπότ. Ενδιάμεσα βέβαια υπάρχουν επτά δεκαετίας έρευνας και εξέλιξης.

Το 1950 ο Άλαν Τούρινγκ περιγράφει τη πασίγνωστη μηχανή Τούρινγκ: μια βασική αφηρημένη μηχανή που μεταχειρίζεται σύμβολα, και η οποία παρ' όλη την απλότητά της έχει την ικανότητα να προσαρμοστεί έτσι ώστε να προσομοιώνει τη λογική οποιουδήποτε αλγορίθμου. Το 1951 κατασκευάζονται τα πρώτα προγράμματα που παίζουν σκάκι και ντάμα. Το 1956 ο Τζον Μακάρθι αναφέρεται πρώτος στον όρο ‘Τεχνητή Νοημοσύνη’ και δύο χρόνια αργότερα εφευρίσκει τη γλώσσα προγραμματισμού Lisp. Στο Εδιμβούργο το 1966 ιδρύεται το Εργαστήριο Μηχανικής Νοημοσύνης και το 1970 παρουσιάζεται το Planner, μια εντυπωσιακή επίδειξη αλληλεπίδρασης μεταξύ ανθρώπου και υπολογιστή. Τέσσερα χρόνια μετά ο Τεντ Σόρτλιφ τελειώνει στο Στάνφορντ τη διατριβή του, η οποία παρουσιάζει μια πρακτική προσέγγιση στην ιατρική διάγνωση, βασισμένη πάνω σε συγκεκριμένους κανόνες, ενώ είκοσι χρόνια μετά, το 1994 οι Ντίκμανς και Ντάιμλερ – Μπενζ επιδεικνύουν στο Παρίσι ένα σύστημα αυτόματης οδήγησης. Το 1999 η Sony παρουσιάζει το AIBO το πρώτο αυτόνομο κατοικίδιο – ρομπότ.

Εικόνα 28: Το AIBO

Η πιο διάσημη και πολυδιαφημισμένη στιγμή στην ιστορία της Τεχνητής Νοημοσύνης είναι οι αναμετρήσεις στο σκάκι μεταξύ του Γκάρι Κασπάροφ (ρώσου παγκόσμιου πρωταθλητή) και διαφόρων υπολογιστικών μηχανημάτων που ξεκίνησαν το 1989 και ολοκληρώθηκαν το 2003. Ο Κασπάροφ σε αυτές τις αναμετρήσεις μέτρησε νίκες, ήττες και ισοπαλίες αναδεικνύοντας όσο ποτέ άλλοτε ότι η ανθρώπινη σκέψη δεν μπορεί να προσομοιωθεί στο έπακρο από κανένα αλγόριθμο, ξεγελώντας ενίοτε τον αντιπαλό του με διάφορα 'ανορθόδοξα' και άκρως ανθρώπινα τρικ!

Εικόνα 29: Ο παγκόσμιος πρωταθλητής Γκάρυ Κασπάροφ εναντίον του Deep Blue PC το 1997.

Παρ' όλα αυτά η ανάπτυξη της Τεχνητής Νοημοσύνης έδωσε απίστευτη ώθηση στην τεχνολογική εξέλιξη των τελευταίων δεκαετιών σε διάφορους τομείς ένας από τους οποίους είναι και η Ρομποτική.

2.2 Η Τεχνητή Νοημοσύνη του Delta Robot

Στη δική μας περίπτωση το ρομπότ έπρεπε να προγραμματιστεί ώστε με τη βοήθεια μιας απλής web - κάμερας να μπορεί να επιτελέσει διάφορες λειτουργίες έτσι ώστε τελικά να καθιστά δυνατή τη διεξαγωγή παιχνιδιού ντάμας με αντίπαλο έναν άνθρωπο χωρίς τη παρεμβολή χειριστηρίων. Οι βασικές αυτές λειτουργίες είναι οι παρακάτω:

(i.) Εντοπισμός της σκακιέρας ανεξάρτητως της θέσης της: Η σκακιέρα κατα βάση θα είναι σε ένα συγκεκριμένο σημείο, αλλά θεωρώντας ότι το ρομπότ θα ανταποκρίνεται στις πραγματικές συνθήκες ενός παιχνιδιού με άνθρωπο (πιθανώς απρόσεχτο), έπρεπε να διασφαλιστεί η ικανότητα εντοπισμού της σκακιέρας από το ρομπότ ακόμα και αν αυτή μετακινηθεί.

(ii.) Εντοπισμός Κίνησης (Motion Detection): Ήταν απαραίτητο το ρομπότ να μπορεί να διακρίνει έαν κάτι κινείται στο χώρο εργασίας του, έτσι ώστε το ίδιο να κατέχει πότε είναι η σειρά του να παίξει άλλα και να αποφευχθεί πιθανός 'τραυματισμός' είτε του ρομπότ είτε του αντιπάλου του.

(iii.) Διαχωρισμός Χρωμάτων (Color Detection): Είναι προφανές πως εάν το ρομπότ δεν μπορούσε να διαχωρίσει τα χρώματα, δεν θα μπορούσε να διαχωρίσει τα δικά του πούλια με του αντιπάλου του και άρα θα του ήταν αδύνατο να παίξει ντάμα (τουλάχιστον όπως εμείς τη γνωρίζουμε)!

(iv.) Δυναμικός Εντοπισμός της θέσης των πουλιών: Φυσικά σε ένα κανονικό παιχνίδι τα πούλια αλλάζουν θέση (είτε τρώγονται). Έτσι το ρομπότ θα έπρεπε να μπορεί να εντοπίζει κάθε φορά που είναι τα πούλια αυτά και να προσαρμόζει τις κινήσεις του στα νέα δεδομένα και να αποτυπώνει τα νεά δεδομένα της σκακιέρας στην οθόνη.

(v.) Κανόνες του παιχνιδιού – αλληλεπίδραση με άνθρωπο: Τέλος, το πιό σημαντικό απ' όλα είναι το ρομπότ να 'γνωρίζει' τους κανόνες του παιχνιδιού που προορίζεται να παίξει. Ο αλγοριθμικός σχεδιασμός του κώδικα του παιχνιδιού της ντάμας δεν αποτέλεσε εκτενές αντικείμενο μελέτης της εργασίας αυτής, αλλά πάρθηκε έτοιμος και αφού τροποποιήθηκε κατά τις ανάγκες μας χρησιμοποιήθηκε. Θα αναφερθεί ξεχωριστά μια μικρή επεξήγηση του κώδικα αυτού (στο κεφάλαιο σχετικά με το παιχνίδι της ντάμας) και σε ποιές αλγοριθμικές αρχές στηρίχτηκε. Για την επικοινωνία και αλληλεπίδραση με τον άνθρωπο – αντίπαλο χρησιμοποιήθηκαν σημάνσεις στο user interface στην οθόνη του pc καθώς και ήχοι που θα αναπαράγονται.

Οι παραπάνω είναι οι βασικές λειτουργίες που λήφθηκαν υπόψιν για τον προγραμματισμό του Ρομπότ Δέλτα. Παρακάτω θα αναλυθούν εκτενέστερα μάζι με τον αντίστοιχο κώδικα και μαζί τους θα αναφερθούν και κάποιες άλλες βοηθητικές λειτουργίες που προέκυψαν στην πορεία με σκοπό τη βέλτιστη λειτουργία του ρομπότ (φιλτράρισματα, βοηθητικοί ήχοι για βέλτιστη επικοινωνία με τον ανθρώπινο περίγυρο κτλ) ώστε να έχει τη δυνατότητα να λειτουργεί σε ένα περιβάλλον 'φυσιολογικό' (χωρίς προστατευτικα τζάμια, χειριστήρια κτλ) και να μπορεί να ανταποκρίνεται σε πιθανά προβλήματα κατά τη διάρκεια του παιχνιδιού (μετακίνηση σκακιέρας, κλέψιμο σειράς από τον αντίπαλο και άλλα).

2.3 Περί compilers και κώδικα...

2.3.1 Τα απαραίτητα εργαλεία της υλοποίησης μας

Για τον επίτευξη των παραπάνω προγραμματίσαμε σε γλώσσα C χρησιμοποιώντας τον compiler DevCpp 4.9.9.2 beta version και φυσικά τις βιβλιοθήκες της OpenCV2.0 καθώς και μια webcam σαν αισθητήρα.

Για να ενσωματωθούν οι βιβλιοθήκες της OpenCV στον DevCpp, αφού εγκαταστήσαμε την OpenCV δημιουργήσαμε εναν New compiler με όνομα OpenCV2.0 στο Tools/Compiler Options/Settings in DevC++. Έπειτα, τσεκάραμε το κουτάκι που γράφει "Add these commands to the linker command line" και προσθέσαμε τις παρακάτω εντολές στο αντίστοιχο σημείο '-llibcxcore200 -llibcv200 -llibcvaux200 -llibhighgui200 -llibml200'.

Μετά στο Directories Tab προσθέσαμε τα παρακάτω (εάν η OpenCV2.0 είναι εγκατεστημένη στο C:\ τότε παραμένουν ίδια, αλλιώς πρέπει να αλλάξει η αντίστοιχη διεύθυνση κάθε βιβλιοθήκης):

a) Binaries:

C:\OpenCV2.0\bin

C:\Dev-Cpp\Bin

C:\Dev-Cpp\libexec\gcc\mingw32\3.4.2

b) Libraries:

C:\OpenCV2.0\lib

C:\Dev-Cpp\lib

c) C Includes:

C:\OpenCV2.0\include\opencv

C:\Dev-Cpp\include

d) C++ Includes:

C:\OpenCV2.0\include\opencv

C:\Dev-Cpp\lib\gcc\mingw32\3.4.2\include

C:\Dev-Cpp\include\c++\3.4.2\backward

C:\Dev-Cpp\include\c++\3.4.2\mingw32

C:\Dev-Cpp\include\c++\3.4.2

C:\Dev-Cpp\include

Τέλος, κάναμε και μία αλλαγη και στο xcoperations.hpp αρχείο το οποίο βρίσκεται στο [OpenCV directory]\include\opencv:

Στη γραμμή 67-68 αλλάξαμε τον παρακάτω κώδικα,

#else

#include <bits/atomicity.h>

#if __GNUC__ >= 4

με τον εξής:

#else

#include <bits/atomicity.h>

#if __GNUC__ >= 4 || __MINGW32__

Φυσικά μπορεί να χρησιμοποιηθεί οποιοσδήποτε compiler, απλά αλλάζει η διαδικασία ενσωμάτωσης της OpenCV. Οι αντίστοιχες οδηγίες για άλλους compilers είναι εύκολο να βρεθούν στο διαδίκτυο.

2.3.2 Ανάλυση Κώδικα Visual

Στην αρχή δηλώνουμε τις βιβλιοθήκες που θα χρησιμοποιήσουμε,

- **Βιβλιοθήκες**

```
// Απαραίτητες βιβλιοθήκες
#include<cv.h>
#include<cxcore.h>
#include<cvaux.h>
#include<highgui.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <windows.h>
#include <mmsystem.h>
#include “Gameplay_New.c” // Το gameplay του παιχνιδιού μας (δες επόμενο κεφάλαιο).
```

και ακολουθεί η δήλωση διαφόρων μεταβλητών του προγράμματος, έχοντας έτσι τη δυνατότητα να αλλάζουμε τις διάφορες παραμέτρους του κώδικα γρήγορα και απλά.

- **Παράμετροι**

```
/* Vangelis Angelinos  11/10/2011 Copyright C & R */
// Ορισμός του Driver id (1 ή 0) για την webcam και τα pixels της εικόνας.

#define DRIVER_ID  1
#define FRAME_WIDTH  320 // Το παράθυρο Preview έχει ανάλυση 320x240.
#define FRAME_HEIGHT 240

// Απαραίτητες τιμές για το πρόγραμμα

#define  BLOCK_CHESSBOARD_WIDTH   45   // Οι διαστάσεις της εικονικής μας σκακιέρας
#define BLOCK_CHESSBOARD_HEIGHT 45
#define CHESSBOARD_DIMENSION  8


// Σταθερές του κόκκινου χρώματος

#define RED_COLOR_R 227
#define RED_COLOR_G 108
#define RED_COLOR_B 106
#define THRES_RED_COLOR_R 34
#define THRES_RED_COLOR_G 18
#define THRES_RED_COLOR_B 21

// Σταθερές του πράσινου χρώματος

#define GREEN_COLOR_R 88
#define GREEN_COLOR_G 155
#define GREEN_COLOR_B 153
#define THRES_GREEN_COLOR_R 20
#define THRES_GREEN_COLOR_G 27
#define THRES_GREEN_COLOR_B 27
```

```
// Σταθερές του μπλε χρώματος


#define BLUE_COLOR_R 70
#define BLUE_COLOR_G 102
#define BLUE_COLOR_B 161
#define THRES_BLUE_COLOR_R 36//36
#define THRES_BLUE_COLOR_G 37//37
#define THRES_BLUE_COLOR_B 28//28

// Σταθερές του πορτοκαλί χρώματος

#define ORANGE_COLOR_R 247
#define ORANGE_COLOR_G 153
#define ORANGE_COLOR_B 130
#define THRES_ORANGE_COLOR_R 30//23
#define THRES_ORANGE_COLOR_G 19//36
#define THRES_ORANGE_COLOR_B 26//47

// Ορισμός του μέγιστου πλήθους των παιχτών που ψάχνω

#define MAX_MAN 8  // Μέγιστο πλήθος παιχτών κάθε ομάδας.
#define MAX_KING 4  // Μέγιστο πλήθος νταμών για κάθε ομάδα.

// Ορισμός της σταθεράς φίλτρου

#define FILTER_THRESHOLD 20

// Ορισμός των διαστάσεων του παραλληλογράμμου (αυτό που περικυκλώνει τους
παίχτες – πούλια).

#define DIMENSION_ONE 15
#define DIMENSION_ONE_DOUBLE 30
#define DIMENSION_ONE_HALF 7

// Τιμές του Image Roi (Region of Interest – Περιοχή ενδιαφέροντος της
εικόνας)
// Αυτές οι τιμές είναι οι συντεταγμένες των pixel που δημιουργούν το
παραλληλόγραμμο που αγκαλιάζει τη σκακιέρα.
// Η συνάρτηση CvSetImageROI τελικά δεν χρησιμοποιείται αλλά φιλτράρουμε
την εικόνα μας με παρόμοιο τρόπο γι’ αυτό και δεν αλλάξαμε τα ονόματα των
μεταβλητών.

#define IMAGEROI_PT_1_x 30
#define IMAGEROI_PT_1_y 30
#define IMAGEROI_PT_2_x 250
#define IMAGEROI_PT_2_y 230

// Η λούπα των στιγμιοτύπων εικόνας (frames)

#define FRAME_LOOP 10 // Κάθε 10 frames απαιτούμε απάντηση από το
πρόγραμμα.

// Needed flag’s  values to simulate the two switches that exist on the
robot, and will give signal that either the robot, or the human is about to
play now.
#define HUMAN_PLAYS 0
#define ROBOT_PLAYS 1
```

Η δήλωση κάθε παραμέτρου θα αναφερθεί και ξεχωριστά στο κομμάτι του κώδικα στο οποίο αντιστοιχεί. Θα ξεκινήσουμε σχολιάζοντας πρώτα τις συναρτήσεις και έπειτα θα αναφερθούμε στη main του κώδικα μας. Στα πούλια αναφέρονται εναλλακτικά και ως παίχτες.

2.3.2.1 Οι βασικές συναρτήσεις του κώδικα

Ξεκινώντας, ορίζουμε τη συνάρτηση current_time η οποία μας επιστρέφει στην οθόνη μας την ημερομηνία και την ώρα (GMT+2).

- **Συνάρτηση Current_Time**

```
// Συνάρτηση που δείχνει την ώρα στην οθόνη μας.

char* current_time(void)
{
    struct tm *ptr;
    time_t lt;

    lt = time(NULL);
    ptr = localtime(&lt);
    return asctime(ptr);
}
```

Η παραπάνω συνάρτηση είναι προφανές πως απλά επιστρέφει την ώρα GMT+2 στην οθόνη μας.

- **Συνάρτηση OpenCOMport**

```
int OpenCOMport(HANDLE h_Comm){  // Ανοίγω τη σειριακή θύρα.

   BOOL  m_PortReady;
   DCB      m_dcb;
   COMMTIMEOUTS m_CommTimeouts;

   h_Comm = CreateFile( "//./COM1",
                  GENERIC_READ | GENERIC_WRITE,
                  0,
                  NULL,
                  OPEN_EXISTING,
                  FILE_FLAG_OVERLAPPED,
                  0);
   if (h_Comm == INVALID_HANDLE_VALUE){
        // Λάθος κατά το άνοιγμα της σειριακής. Ακύρωση της διαδικασίας.
        return -1;
   }

   m_PortReady = SetupComm(h_Comm, 128, 128); // Ορισμός του μεγέθους του
buffer.

   m_PortReady = GetCommState(h_Comm, &m_dcb);
   m_dcb.BaudRate = 9600;
   m_dcb.ByteSize = 8;
   m_dcb.Parity = NOPARITY;
   m_dcb.StopBits = ONESTOPBIT;
```

```
    m_dcb.fAbortOnError = TRUE;

    m_PortReady = SetCommState(h_Comm, &m_dcb);


    // Set timeouts (optional)
    //m_PortReady = GetCommTimeouts (hComm, &m_CommTimeouts);

    //m_CommTimeouts.ReadIntervalTimeout = 50;
    //m_CommTimeouts.ReadTotalTimeoutConstant = 50;
    //m_CommTimeouts.ReadTotalTimeoutMultiplier = 10;
    //m_CommTimeouts.WriteTotalTimeoutConstant = 50;
    //m_CommTimeouts.WriteTotalTimeoutMultiplier = 10;

    //m_PortReady = SetCommTimeouts (hComm, &m_CommTimeouts);

    return 0;
}
```

Η παραπάνω συνάρτηση ενεργοποιεί την σειριακή θύρα.

- **Συνάρτηση ClosePort**

```
void ClosePort(HANDLE h_comm){  // Κλείσιμο της σειριακής θύρας.
  CloseHandle(h_comm);
}
```

Με την ClosePort η σειρική θύρα απενεργοποιείται.

- **Συνάρτηση FindManSquare**

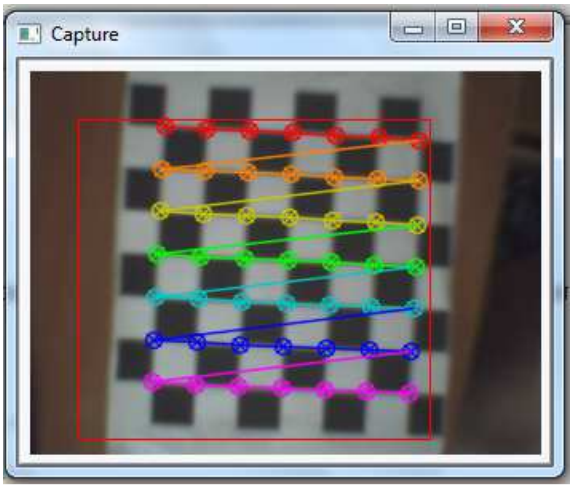
Η FindManSquare είναι ίσως η πιο σημαντική συνάρτηση του κωδικά μας και σίγουρα η μεγαλύτερη σε μέγεθος. Για τον λόγο αυτό θα σχολιαστεί ανα κομμάτι και όσο το δυνατόν με βοηθητικά σχήματα.

Η δουλειά που επιτελεί είναι να βρίσκει τα πούλια που υπάρχουν στην πραγματική σκακιέρα και να τα τοποθετεί στην αντίστοιχη θέση στην virtual σκακιέρα που έχουμε στην οθόνη μας.

Παράλληλα με την παραπάνω λειτουργία όμως γίνεται και ένα χωρικό φιλτράρισμα στην πραγματική σκακιέρα, έτσι ώστε να μην επιστρέφει στην οθόνη πούλια τα οποία δεν είναι σε έγκυρη θέση (π.χ. ένα πούλι που το μισό του μέρος είναι σε άσπρο τετράγωνο και το άλλο μισό σε μάυρο).

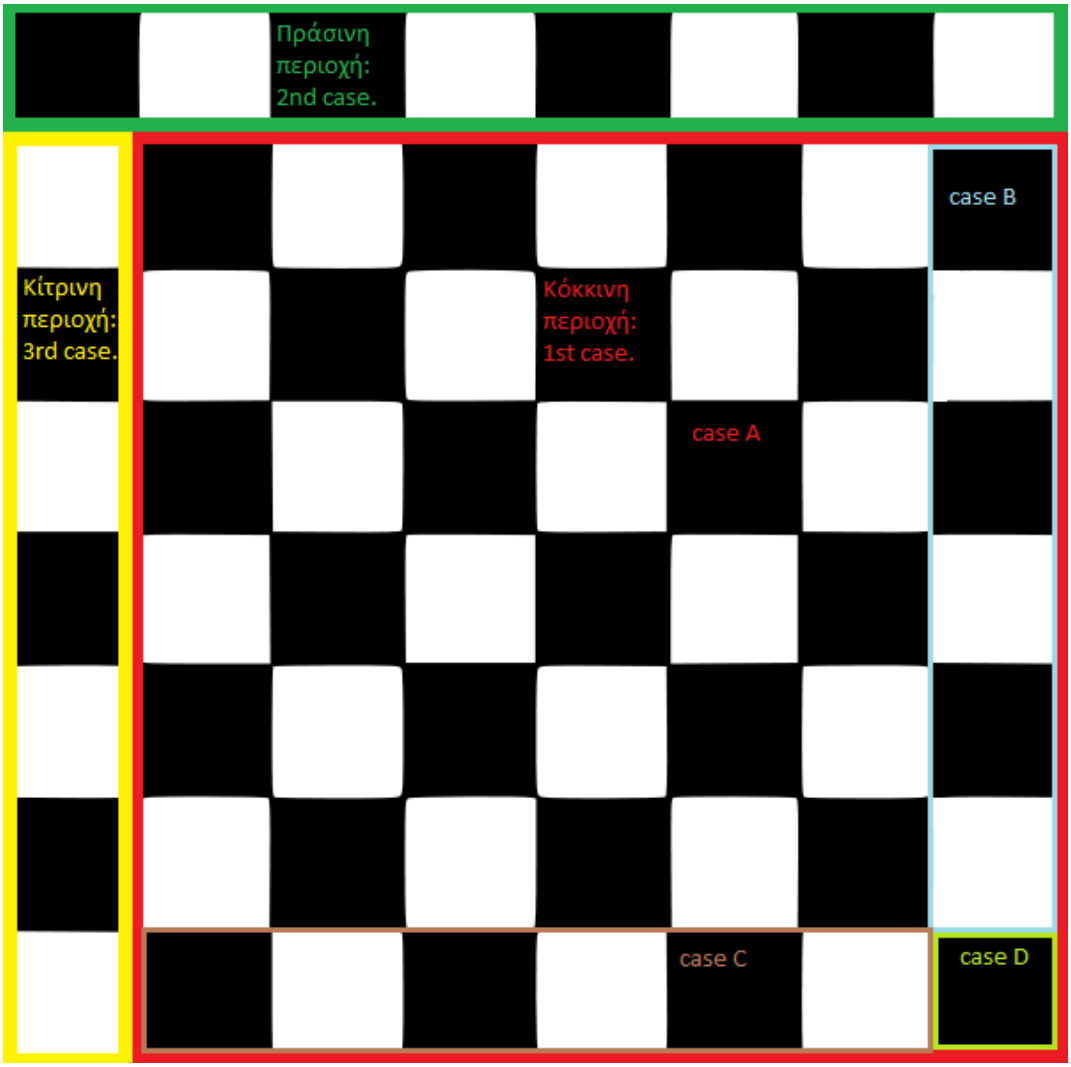
Επειδή για να εντοπίσουμε τη θέση του κάθε παίχτη είχαμε ως σημείο αναφοράς τις εσωτερικές γωνίες που μας επιστρέφει η συνάρτηση cvFindChessboard, η σκακιέρα έπρεπε να χωριστεί σε ξεχωριστά μέρη.

Είναι προφανές ότι η αλγοριθμική υλοποίηση ήταν πολύ πιο έυκολη για τα εσωτερικά τετράγωνα καθώς είχαμε πολλά περισσότερα σημεία αναφοράς, άρα και πιό έυκολους ελέγχους. Είναι σημαντικό εδώ να επαναληφθεί πως η έτοιμη συνάρτηση cvFindChessboard για μια σκακιέρα 8x8 (όπως και η δικιά μας) επιστρέφει έναν πίνακα με 49 εσωτερικές γωνίες.



Εικόνα 30: Ο τρόπος εντοπισμού της σκακιέρας από τη cvFindChessboard. Με την εντολή cvDrawChessboad επιτυγχάνουμε την αποτύπωση της μεθόδου στην οθόνη μας.

```
// Αυτή η συνάρτηση βρίσκει κάθε πούλι και τον τοποθετεί στο αντίστοιχο τετράγων στην οθόνη στην εικονική μας σκακιέρα.
// Ακόμα, φιλτράρει έτσι ώστε να εξασφαλίσουμε πως θα έχουμε ένα πούλι σε κάθε τετράγωνο (καλύπτοντας κατά κάποιο τρόπο τα τυχαία λάθη της τεχνητής μας όρασης).

void FindManSquare(int Manstype, CvPoint2D32f *Man, CvPoint2D32f corner[],int board[CHESSBOARD_DIMENSION][CHESSBOARD_DIMENSION], int array_cnt1){


// Ορισμός απαραίτητων μεταβλητών.
int i, j, ManColumn=0, ManRow=0;

// Ο παρακάτω αλγόριθμος διαχωρίζει ουσιαστικά τη σκακιέρα σε τρεις περιοχές.

// Κάθε περιοχή αντικατοπτρίζεται σε μία διαφορετική ‘if υποπερίπτωση’.

// 1η περίπτωση είναι όλα τα τετράγωνα της σκακιέρας εκτός από την έξω-αριστερά στήλη και την έξω-πάνω γραμμή.

// 2η περίπτωση είναι η έξω-αριστερά στήλη.

// 3η περίπτωση είναι η έξω-πάνω σειρά.
```



Εικόνα 31: Ο διαχωρισμός της σκακιέρας σε περιοχές. Η 1st case χωρίζεται εκ των υστέρων σε άλλες 4 υποπεριοχές A, B, C, D.

```
// Λούπα για να βρεθούν όλα τα πούλια.
for (j=0;j<array_cnt1;j++){
    if (Man[j].x > 0){   // Έλεγχος ένα η θέση του παίχτη-πούλι είναι θετική.
        for     (i=0;i<=(CHESSBOARD_DIMENSION-1)*(CHESSBOARD_DIMENSION-1);i++) { // Σκανάρισμα – έλεγχος όλων των γωνιών με τιμές 0-48 = 49 τιμές συνολικά,
```

Εδώ πριν από όλα τοποθετείται ο έλεγχος των διαστάσεων της σκακιέρας. Με αυτό επιτυγχάνουμε το πρόγραμμα να απορρίπτει οτιδήποτε έγχρωμο μοιάσει με πούλι και είναι εκτός των ορίων της σκακιέρας. Οι αντίστοιχες σταθερές έχουν οριστεί στην αρχή του προγράμματος:

```
#define IMAGEROI_PT_1_x 30
#define IMAGEROI_PT_1_y 30
#define IMAGEROI_PT_2_x 250
#define IMAGEROI_PT_2_y 230
```

```
//(7 εσωτερικές γωνίες * 7 εσωτερικές γωνίες = 49)
            if ((Man[j].x - DIMENSION_ONE_HALF > IMAGEROI_PT_1_x)     //
Ορισμός ορίων της εικόνας (ουσιαστικά θέτω μια υποπεριοχή της συνολικής
εικόνας ως αυτή που με ενδιαφέρει).
                && (Man[j].x + DIMENSION_ONE_HALF < IMAGEROI_PT_2_x)   //
Μόνο τα πούλια που είναι στη σκακιέρα.
                && (Man[j].y - DIMENSION_ONE_HALF > IMAGEROI_PT_1_y)   //
[Δες το ορθογώνιο παραλληλόγραμμο] στο παράθυρο Capture.
                && (Man[j].y + DIMENSION_ONE_HALF < IMAGEROI_PT_2_y)){ //
Ουσιαστικά κάνω χωρικό φιλτράρισμα.
```

Τώρα ξεκινάει ο έλεγχος της 1st case – τα εσωτερικά τετράγωνα δηλαδή όπως φαίνεται και στην παραπάνω εικόνα και έπειτα γίνεται ο έλεγχος κάθε υποπερίπτωσης A, B, C, και D. Το χωρικό φιλτράρισμα συνεχίζεται και σε αυτό το μέρος του κώδικα έτσι ώστε να απομονώνται ανεπιθύμητα πούλια και ‘ σκουπίδια’.

```
                //-----1η περίπτωση -----[όλες οι γωνίες της σκακιέρα εκτός
από την έξω-αριστερά στήλη και την έξω-πάνω σειρά]

                if ((Man[j].x < corner[i].x)&&(Man[j].y < corner[i].y)){
// 1η περίπτωση – Εσωτερικά τετράγωνα.

                // Η 1η περίπτωση έχει 4 υποπεριπτώσεις:
                // υποπερίπτωση A: εσωτερικά τετράγωνα
                // υποπερίπτωση B: έξω - δεξιά στήλη
                // υποπερίπτωση C: έξω - κάτω σειρά
                // υποπερίπτωση D: πρώτο τετράγωνο από την κάτω σειρά και
τη δεξιά στήλη.
                // Δες το παραπάνω σχήμα για καλύτερη κατανόηση.

                    if ((i-CHESSBOARD_DIMENSION)>=0){ //  Υποπερίπτωση A:
έλεγχος των εσωτερικών τετραγώνων.
                      if    ((Man[j].x    +    DIMENSION_ONE_HALF    <
corner[i].x)
                        && (Man[j].x - DIMENSION_ONE_HALF > corner[i-
CHESSBOARD_DIMENSION].x)
                        &&    (Man[j].y    +    DIMENSION_ONE_HALF    <
corner[i].y)
                        && (Man[j].y - DIMENSION_ONE_HALF > corner[i-
CHESSBOARD_DIMENSION].y)){

                         ManColumn  =i/(CHESSBOARD_DIMENSION-1);    //
Αυτός είναι ο αλγόριθμος.
                         ManRow=    i    -    (CHESSBOARD_DIMENSION-
1)*ManColumn;
                         board[ManRow][ManColumn]=Manstype;         //
Φορτώνω δεδομένα στη minichessboard.
                         break;

                      }else{                                        //
Υποπερίπτωση B: έξω – δεξιά στήλη.
                        board[0][i/(CHESSBOARD_DIMENSION-
1)]=Manstype; // Φορτώνω δεδομένα στη minichessboard.
                        break;
                    }
```

```
                    }else{   // Υποπερίπτωση C: έξω – κάτω σειρά.
                        if ((i-1>0) && (Man[j].x + DIMENSION_ONE_HALF >
corner[i-1].x))  // Έλεγχος της υποπερίπτωσης C.
                            board[i][0]=Manstype; // Φορτώνω δεδομένα στη
minichessboard.
                        else board[0][0]=Manstype;   // Υποπερίπτωση D: το
πρώτο τετράγωνο.
                        break;
                        }
```

Εδώ γίνεται ο έλεγχος για την 2nd case,

```
                //------2η περίπτωση------[έξω – αριστερά στήλη]

                }else  if  ((Man[j].x  >  corner[(CHESSBOARD_DIMENSION-
1)*(CHESSBOARD_DIMENSION-1)-1].x) // 2η περίπτωση (έξω – αριστερά στήλη).
                          &&(Man[j].y < corner[i].y)){                //
Έλεγχος της 2ης περίπτωσης.

                        if    ((Man[j].y    +    DIMENSION_ONE_HALF    <
corner[i].y) // Χωρικό φιλτράρισμα (αναφέρεται και παραπάνω).
                            &&   (Man[j].y   -   DIMENSION_ONE_HALF   >
corner[i-CHESSBOARD_DIMENSION-1].y)){

                            ManColumn  =  i/(CHESSBOARD_DIMENSION-1);  //
Αυτός είναι ο αλγόριθμος.
                            board[CHESSBOARD_DIMENSION-
1][ManColumn]=Manstype; // Φορτώνω δεδομένα στη minichessboard.

                            break;
                        }
```

Και τέλος ο έλεγχος για την 3rd case:

```
                //------3η περίπτωση------[έξω – πάνω σειρά]

                }else  if  ((Man[j].y  >  corner[((CHESSBOARD_DIMENSION-
1)*(CHESSBOARD_DIMENSION-1)-1)].y)
                        &&(Man[j].x  <  corner[i].x)){  //  Έλεγχος  3ης
περίπτωσης

                        if ((Man[j].x - DIMENSION_ONE_HALF > corner[i-
1].x) // Χωρικό φιλτράρισμα (αναφέρεται και παραπάνω).
                            &&   (Man[j].x   +   DIMENSION_ONE_HALF   <
corner[i].x)
                            &&                (i>((CHESSBOARD_DIMENSION-
1)*(CHESSBOARD_DIMENSION-1)-(CHESSBOARD_DIMENSION-1))){//42

                            ManRow      =      i-((CHESSBOARD_DIMENSION-
1)*(CHESSBOARD_DIMENSION-1)-(CHESSBOARD_DIMENSION-1));//42
                            board[ManRow][CHESSBOARD_DIMENSION-
1]=Manstype; // Φορτώνω δεδομένα στη minichessboard.
                            break;
                        }
                    }
```

```
            }
        }
    }
}
}
```

- **Συνάρτηση CheckManChange**

```
// Αυτή η συνάρτηση ελέγχει τις αλλαγές στη σκακιέρα έτσι ώστε το ρομπότ να
‘απαντήσει’ στην κίνηση του ανθρώπου. Εάν κάτι αλλάξει επιστρέφει 1, αλλιώς
0.

int                          CheckManChange                          (int
curchess[CHESSBOARD_DIMENSION][CHESSBOARD_DIMENSION],                int
prevchess[CHESSBOARD_DIMENSION][CHESSBOARD_DIMENSION]){

int i,j;

    for  (i=0;i<=CHESSBOARD_DIMENSION-1;i++)     //  Έλεγχος  όλων  των
τετραγώνων.
        for (j=0;j<=CHESSBOARD_DIMENSION-1;j++)
            if(curchess[i][j]  !=  prevchess[i][j]) // Έλεγχος εάν κάτι
άλλαξε.
                return 1; // Αν κάτι άλλαξε επιστρέφει 1,
return 0;                 // αλλιώς 0.
}
```

Η CheckManChange, όπως λέει και το όνομα της ελέγχει εάν έχει γίνει αλλαγή στο πίνακα της σκακιέρας. Εάν όντως έχει γίνει αλλαγή (δηλαδή έχει παίξει ο αντίπαλος - άνθρωπος) τότε η συνάρτηση αυτή επιστρέφει 1, αλλιώς 0.

- **Συνάρτηση ChessBackUp**

```
// Αυτή η συνάρτηση φορτώνει νέα δεδομένα στην σκακιέρα (chessboard).

int                          ChessBackUp                          (int
curchess[CHESSBOARD_DIMENSION][CHESSBOARD_DIMENSION],                int
prevchess[CHESSBOARD_DIMENSION][CHESSBOARD_DIMENSION]){

int i,j;


      for   (i=0;i<=CHESSBOARD_DIMENSION-1;i++)   //   Διατρέχω   όλα   τα
τετράγωνα.
          for (j=0;j<=CHESSBOARD_DIMENSION-1;j++)
              prevchess[i][j] = curchess[i][j]; // Φορτώνω νέα δεδομένα.

return 0;
}
```

Η ChessBackUp επανατοποθετεί τα αλλαγμένα στοιχεία της σκακιέρας στην προηγούμενη θέση ώστε να ξαναγίνει μετά ο καινούριος έλεγχος.

- **Συνάρτηση FindTheContours**

Η συνάρτηση FindTheContours πρακτικά εντοπίζει τα διάφορα αντικείμενα ανάλογα με το χρώμα.

```
// Αυτή η συνάρτηση βρίσκει τα περιγράμματα των παιχτών-πουλιών.

int FindTheContours(IplImage* grey,IplImage* image, CvPoint2D32f **Man){

    CvRect bndRect = cvRect(0,0,0,0); // Αυτό είναι ένα ευθύγραμμο παραλληλόγραμμο που περικυκλώνει αντικείμενα.
    int icnt, i;            // Ορισμός των απαραίτητων μεταβλητών.
    int cnt_contour=0;
    CvPoint tpt1, tpt2; // Ορισμός του Cvpoint.

    // Φίλτρα που βελτιστοποιούν την εικόνα.
    cvDilate(grey, grey, 0, 2);
    cvErode(grey, grey, 0, 2);

// Βρίσκει τα περιγράμματα των χρωματιστών αντικειμένων στο κάθε στιγμιότυπο (frame) συγκρίνοντας το κάθε στιγμιότυπο με το αμέσως προηγούμενο.

 CvMemStorage* storage_color = cvCreateMemStorage(0);
 CvSeq* colorcontour = 0;
 intarray_counter=cvFindContours(grey,storage_color, &colorcontour,sizeof(CvContour),CV_RETR_CCOMP, CV_CHAIN_APPROX_SIMPLE, cvPoint(0,0));

if (array_counter==0){
       // No Contours found
       cvReleaseMemStorage(&storage_color); // Releasing memory.
       return 0;
    }

// Ορίζω το δυναμικό πίνακα που θα επιστρέφει η συνάρτηση (με τη βοήθεια pointer).

      *Man = (CvPoint2D32f*)malloc(array_counter*sizeof(CvPoint2D32f));
      CvPoint2D32f *temp = (CvPoint2D32f*)malloc(array_counter*sizeof(CvPoint2D32f));
      *Man = temp;
```

Εδώ είναι όλη η ουσία της συνάρτησης. Χρησιμοποιούμε την έτοιμη συνάρτηση CVFindContours. Η συνάρτηση αυτή έχει τη δυνατότητα να εντοπίζει τα pixel εκείνα που διαχωρίζουν τα διάφορα αντικείμενα στην εικόνα. Με πιο απλά λόγια μπορεί να εντοπίσει αντικείμενα ανάλογα με τα κριτήρια που θα ορίσουμε εμείς. Εδώ ως βασικό κριτήριο είναι το χρώμα.

```
// Κύριος αλγόριθμος της συνάρτησης FindTheContours. Σύγκριση κάθε στιγμιοτύπου με το αμέσως προηγούμενο.

    for (icnt=0;icnt<array_counter;icnt++){ // Διατρέχω όλα τα πούλια της σκακιέρας.
      temp[icnt].x = 0;
```

```
      temp[icnt].y = 0;
    }
    icnt = 0; // Θέτω τον μετρητή στο 0.
    //  Επεξεργασία  κάθε  κινούμενου  περιγράμματος  στο  συγκεκριμένο
στιγμιότυπο…
    for( ; colorcontour != 0; colorcontour = colorcontour->h_next ){

         // Κατασκευάζω ένα κινούμενο παραλληλόγραμμο που ‘αγκαλιάζει’ τα
κινούμενα αντικείμαν.
         bndRect = cvBoundingRect(colorcontour, 0);


        tpt1.x = bndRect.x;            // Συντεταγμένες για το παρ/μο.
        tpt1.y = bndRect.y;
        tpt2.x = bndRect.x + bndRect.width;
        tpt2.y = bndRect.y + bndRect.height;
        tpt.x = (tpt1.x + tpt2.x)/2;
        tpt.y = (tpt1.y + tpt2.y)/2;

        if ( (tpt.x>IMAGEROI_PT_1_x) && (tpt.x<IMAGEROI_PT_2_x) &&
(tpt.y>IMAGEROI_PT_1_y) && (tpt.y<IMAGEROI_PT_2_y) ){
            temp[cnt_contour].x = tpt.x;//Man[cnt_contour].x = (tpt1.x +
tpt2.x)/2;
            temp[cnt_contour].y = tpt.y;//Man[cnt_contour].y = (tpt1.y +
tpt2.y)/2;

            cnt_contour++;
        }
         // Ζωγραφίζω το παραλληλόγραμμο.
        //cvRectangle(image, tpt1, tpt2, CV_RGB(255, 0, 0), 2);

    }

    cvReleaseMemStorage(&storage_color); // Απελευθερώνω μνήμη.
    //free(Man);
    //printf("%f\n\n", Man[0].x);
    //printf("%d \n\n", array_counter);
    return cnt_contour;
}
```

- **Συνάρτηση FilterContours**

Ακολουθεί η συνάρτηση Filter Contours η οποία είναι ένα χωροταξικό φίλτρο έτσι ώστε όταν στη σκακιέρα δυο πούλια ίδιου χρώματος είναι σχετικά κοντά (υποθέτουμε ότι ο άνθρωπος αντίπαλος είναι πολύ πιθανό να μην αφήνει τα πούλια ακριβώς στο κέντρο του τετραγώνου), να μην τα εκλαμβάνει ως ένα. Η κατασκευή του φίλτρου αυτού βελτιστοποίησε κατά πολυ την απόδοση του κώδικα σε πραγματικές συνθήκες.

```
// Αυτή η συνάρτηση είναι ένα φίλτρο για κάθε πούλι.
// Χωρίς την FilterContours, δύο πούλια που είναι πολύ κοντά μπορεί να
εντοπιστούν από το πρόγραμμα ως ένα πούλι.
// Έτσι έχουμε αυτό το φίλτρο με την αντίστοιχη σταθερά FILTER_THRESHOLD
για να ξεχωρίζουμε τα δύο πούλια.
// Η σταθερά ορίζεται ως ένας αριθμός pixel.
//  Εάν εντοπιστούν σε απόσταση μικρότερη από την FILTER_THRESHOLD τότε
αντιμετωπίζονται ως ένα πούλι.

int FilterContours(CvPoint2D32f *Man, int array_counter){
```

```
    int icnt = 0; // Ορισμός απαραίτητων μεταβλητών.
    int i, temp1, temp2;
    CvPoint2D32f currentPoint, prevPoint;

    prevPoint.x = Man[0].x;
    prevPoint.y = Man[0].y;

    for (i=1;i< array_counter;i++){  // Διατρέχω όλα τα πούλια.
```

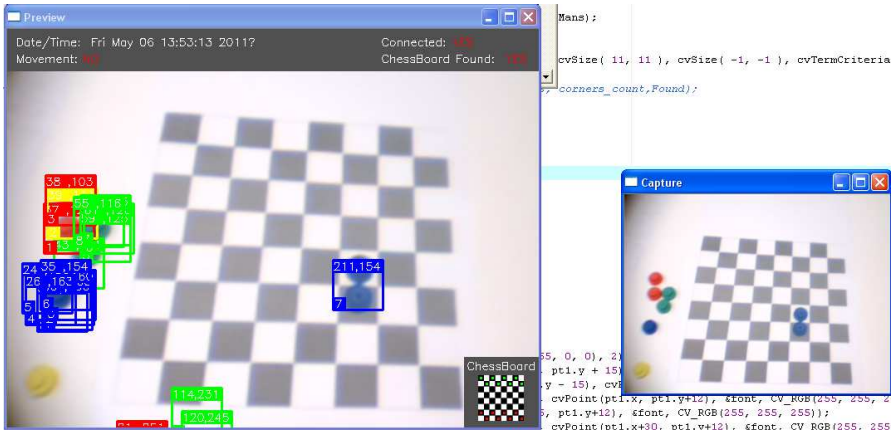
Αφού ορίσουμε τον πίνακα prevPoint με τις συντεταγμένες των παικτών παίρνουμε πάλι τον περιορισμό για τον μέγιστο πλήθος που μπορούν οι παίκτες να έχουν.

```
      temp1 = (int)((Man[i].x - prevPoint.x) + 0.5); // Η απόσταση της x
συντεταγμένης.
      temp2 = (int)((Man[i].y - prevPoint.y) + 0.5); // Η απόσταση της y
συντεταγμένης.

      if                ((abs(temp1)>FILTER_THRESHOLD)                ||
(abs(temp2)>FILTER_THRESHOLD)){ // Έλεγχος της απόστασης των δύο παιχτών.
         if (icnt<array_counter){                          // Εάν είναι
μεγαλύτερη από τη σταθερά τότε είναι δύο πούλια.
            Man[icnt].x = prevPoint.x;
            Man[icnt].y = prevPoint.y;
            icnt++;
            prevPoint.x = Man[i].x;
            prevPoint.y = Man[i].y;
         }
      }else{                     // Εάν η απόσταση δεν είναι μεγαλύτερη από
τη σταθερά τότε είναι το ίδιο πούλι, άρα είναι ένα.
            prevPoint.x += Man[i].x;
            prevPoint.y += Man[i].y;
      }
    }
    return icnt;
}
```

Η βασικη λειτουργία του φίλτρου επιτελείται εδώ. Ως συνθήκη θέτουμε το απόλυτο της διαφοράς των συντεταγμένων x και y να είναι μικρότερο από τη σταθερά FILTER_THRESHOLD που έχει οριστεί στην αρχή του κώδικα ως 20 pixel.

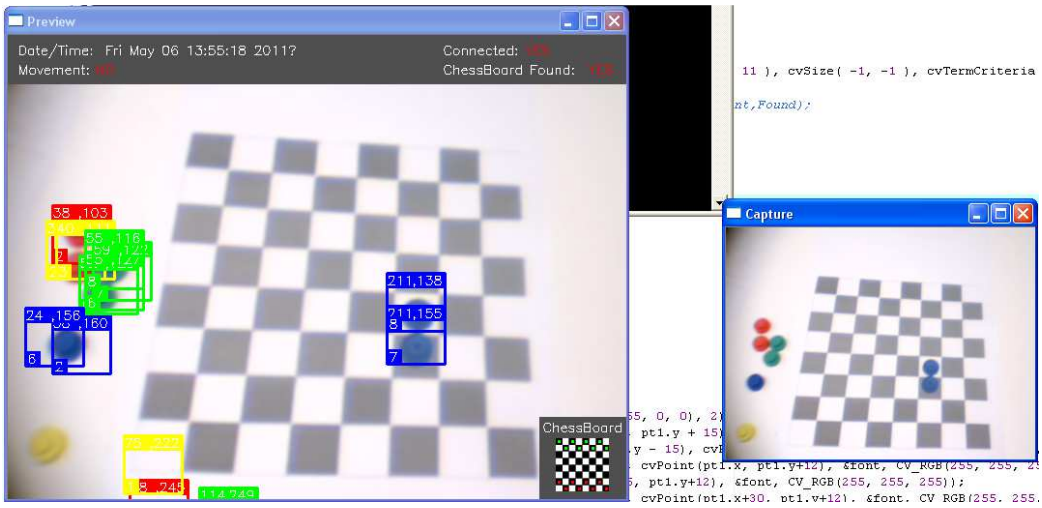
```
// Ορισμός της σταθεράς φίλτρου
#define FILTER_THRESHOLD 12                          ]
```

Εάν η διαφορά είναι μικρότερη και ο μετρητής δεν έχει φτάσει στο μέγιστο πλήθος τότε θεωρούμε τα δύο πούλια ως ένα, αλλιώς ως δύο διαφορετικά πούλια. Εάν η σταθερά FILTER_THRESHOLD είναι πολύ μεγάλη, η λειτουργία του φίλτρου γυρνάει μπούμερανγκ αφού το αποτέλεσμα είναι να χάνει πούλια που πριν εντόπιζε.



Εικόνα 32: Πριν την εφαρμογή του φίλτρου (δύο πούλια σε κοντινές θέσεις τα αναγνωρίζει ως ένα).

Το παράδειγμα αυτό είναι αδύνατο να συμβεί στο κλασικό παιχνίδι ντάμας που παίζει το ρομπότ μας (τα πούλια πρέπει να είναι διαγώνια), αλλά έπρεπε να βελτιστοποιήσουμε στο μέγιστο την απόδοση του κώδικα μας ώστε να μην υπάρξουν τυχαία λάθη. Η εφαρμογή του φίλτρου έφερε τα παρακάτω αποτελέσματα:



Εικόνα 33: Μετά την εφαρμογή του φίλτρου (δύο πούλια σε κοντινές θέσεις αναγνωρίζονται ξεχωριστά).

- **Συνάρτηση CheckforColor**

```
// Αυτή η συνάρτηση ελέγχει κάθε χρώμα ανάλογα με τις RGB τιμές του.
// Κάθε χρώμα έχει κάποιες ξεχωριστές σταθερές που η συνάρτηση θεωρεί ως αποδεκτές.
// Δες στην αρχή του κώδικα όπου ορίζονται οι τιμές.

int CheckforColor(uchar* data, int r, int g,int b,int color_r,int color_g,int color_b,int thres_r,int thres_g,int thres_b){
    int diffR,diffG,diffB;

    diffR = abs(r - color_r);
    diffG = abs(g - color_g);
    diffB = abs(b - color_b);

    if ((diffR<thres_r) &&(diffG<thres_g) && (diffB<thres_b)){
       *data = 255;
    }
```

}

Η συνάρτηση CheckforColor πιστοποιεί ότι βρέθηκε κάποιο από τα ζητούμενα χρώματα σύμφωνα με τις σταθερές που έχουν δωθεί στην αρχή του προγράμματος (RGB values) και σύμφωνα με τα αντίστοιχα thresholds (δηλαδή όρια). Έτσι, εάν η κόκκινη συνιστώσα (red value) του κόκκινου χρώματος (`#define RED_COLOR_R 227`) είναι 227 και το αντίστοιχο threshold 34 (`#define THRES_RED_COLOR_R 34`) οι τιμές μεταξύ (227-34) και (227+34) είναι αποδεκτές ως κόκκινο. Αντίστοιχοι έλεγχοι γίνονται και για τις άλλες δύο συνιστώσες blue και green ώστε τελικά να αποφανθεί το πρόγραμμα για το εκάστοτε χρώμα.

Οι σταθερές για κάθε χρώμα βρέθηκαν καταρχάς με τη βοήθεια προγράμματος το οποίο παίρνοντας εικόνα από την webcam, επέστρεφε τις τιμές αυτές (RGB values και τα αντίστοιχα thresholds) για το pixel που εμείς επιλέγαμε απο το παράθυρο ελέγχου. Έπεια οι τιμές που επεξεργάστηκαν και πειραματικά στις πραγματικές συνθήκες για βέλτιστη απόδοση.

- **Συνάρτηση ColorDetection**

```
// Αυτή η συνάρτηση κάνει 'color detection' (εντοπίζει χρώμα) επιστρέφοντας έναν πίνακα από πούλια για κάθε χρώμα.
// Συνολικά, έχουμε τέσσερα χρώματα – Κόκκινο, Μπλε, Πράσινο, Πορτοκαλί. Στη συνάρτηση αυτή χρησιμοποιούνται οι ορισμένες από πριν συναρτήσεις CheckforColor και FindTheContours

int  ColorDetection(IplImage*  image,CvPoint2D32f  **RMan,CvPoint2D32f **GMan,CvPoint2D32f **BMan,CvPoint2D32f **OMan, int *r_counter
                  , int *g_counter, int *b_counter, int *o_counter){

   // Μέγεθος της εικόνας.
   CvSize imSize;
   imSize.width = FRAME_WIDTH;
   imSize.height = FRAME_HEIGHT;

   int i,j,r,g,b; // Θέτω μεταβλητές.

   // Ορισμός των Color Maps.
   IplImage* RedColorMap = cvCreateImage(imSize, IPL_DEPTH_8U, 1);
   IplImage* GreenColorMap = cvCreateImage(imSize, IPL_DEPTH_8U, 1);
   IplImage* BlueColorMap = cvCreateImage(imSize, IPL_DEPTH_8U, 1);
   IplImage* OrangeColorMap = cvCreateImage(imSize, IPL_DEPTH_8U, 1);

   // Ορισμός των ιδιοτήτων τους.
   int width     = image->width;
   int height    = image->height;
   int nchannels = image->nChannels;
   int step      = image->widthStep;
   int gstep     = RedColorMap->widthStep;

   // Ορισμός των δεικτών για να έχω πρόσβαση στα δεδομένα της εικόνας.
   uchar *data = (uchar*)image->imageData;
   uchar *Reddata = (uchar*)RedColorMap->imageData;
   uchar *Greendata = (uchar*)GreenColorMap->imageData;
   uchar *Bluedata = (uchar*)BlueColorMap->imageData;
```

```
   uchar *Orangedata = (uchar*)OrangeColorMap->imageData;

   for(i = 0; i < height; i++) // Διατρέχω κάθε σειρά,
   {
       for( j = 0; j < width; j++) // και κάθε στήλη της σκακιέρας.
          {
             b = data[i*step + j*nchannels + 0];
             g = data[i*step + j*nchannels + 1];
             r = data[i*step + j*nchannels + 2];

             Reddata[i*gstep + j] = 0;
             Greendata[i*gstep + j] = 0;
             Bluedata[i*gstep + j] = 0;
             Orangedata[i*gstep + j] = 0;

             // Ελέγχω για το κόκκινο χρώμα.
             CheckforColor(&Reddata[i*gstep                                  +
j],r,g,b,RED_COLOR_R,RED_COLOR_G,RED_COLOR_B,THRES_RED_COLOR_R,THRES_
RED_COLOR_G,THRES_RED_COLOR_B);
             // Ελέγχω για το πράσινο χρώμα.
             CheckforColor(&Greendata[i*gstep                                +
j],r,g,b,GREEN_COLOR_R,GREEN_COLOR_G,GREEN_COLOR_B,THRES_GREEN_COLOR_
R,THRES_GREEN_COLOR_G,THRES_GREEN_COLOR_B);
             // Ελέγχω για το μπλε χρώμα.
             CheckforColor(&Bluedata[i*gstep                                 +
j],r,g,b,BLUE_COLOR_R,BLUE_COLOR_G,BLUE_COLOR_B,THRES_BLUE_COLOR_R,TH
RES_BLUE_COLOR_G,THRES_BLUE_COLOR_B);
             // Ελέγχω για το πορτοκαλί χρώμα.
             CheckforColor(&Orangedata[i*gstep                               +
j],r,g,b,ORANGE_COLOR_R,ORANGE_COLOR_G,ORANGE_COLOR_B,THRES_ORANGE_CO
LOR_R,THRES_ORANGE_COLOR_G,THRES_ORANGE_COLOR_B);
          }
   }
  CvPoint2D32f *Rtemp;
  CvPoint2D32f *Gtemp;
  CvPoint2D32f *Btemp;
  CvPoint2D32f *Otemp;

   // Βρίσκω τα περιγράμματα για τα διαφορετικά χρώματα.

    *r_counter = FindTheContours(RedColorMap, image, &Rtemp);
    *RMan = Rtemp;

    *g_counter = FindTheContours(GreenColorMap, image, &Gtemp);
    *GMan = Gtemp;

    *b_counter = FindTheContours(BlueColorMap, image, &Btemp);
    *BMan = Btemp;

    *o_counter = FindTheContours(OrangeColorMap, image, &Otemp);
    *OMan = Otemp;

   // Απελευθερώνω την εικόνα κάθε χρώματος.
   cvReleaseImage(&RedColorMap);
   cvReleaseImage(&GreenColorMap);
   cvReleaseImage(&BlueColorMap);
   cvReleaseImage(&OrangeColorMap);
   //cvReleaseData(&Reddata);
```

```
    //cvReleaseData(&Greendata);
    //cvReleaseData(&Bluedata);
    //cvReleaseData(&Orangedata);
    return 0;
}
```

Ουσιαστικά η ColorDetection χρησιμοποιώντας τις συναρτήσεις FindTheContours και CheckforColor επιστρέφει έναν πίνακα για κάθε χρώμα. Ο πίνακας αυτός περιλαμβάνεις τις συντεταγμένες των αντικειμένων (πούλια στη δική μας περίπτωση) που εντοπίστηκαν.

- **Συνάρτηση drawchessboard**

```
// Αυτή η συνάρτηση ζωγραφίζει την εικονική σκακιέρα στην οθόνη μας.

void drawchessboard(IplImage* img, int x, int y, int
chess[CHESSBOARD_DIMENSION][CHESSBOARD_DIMENSION])
{
```

Οι τιμή CHESSBOARD_DIMENSION έχει δηλωθεί στην αρχή του προγράμματος και υποδηλώνει το πλήθος των τετραγώνων της κάθε πλευράς της σκακιέρας μας (η σκακιέρα έχει θεωρηθεί τετράγωνη).

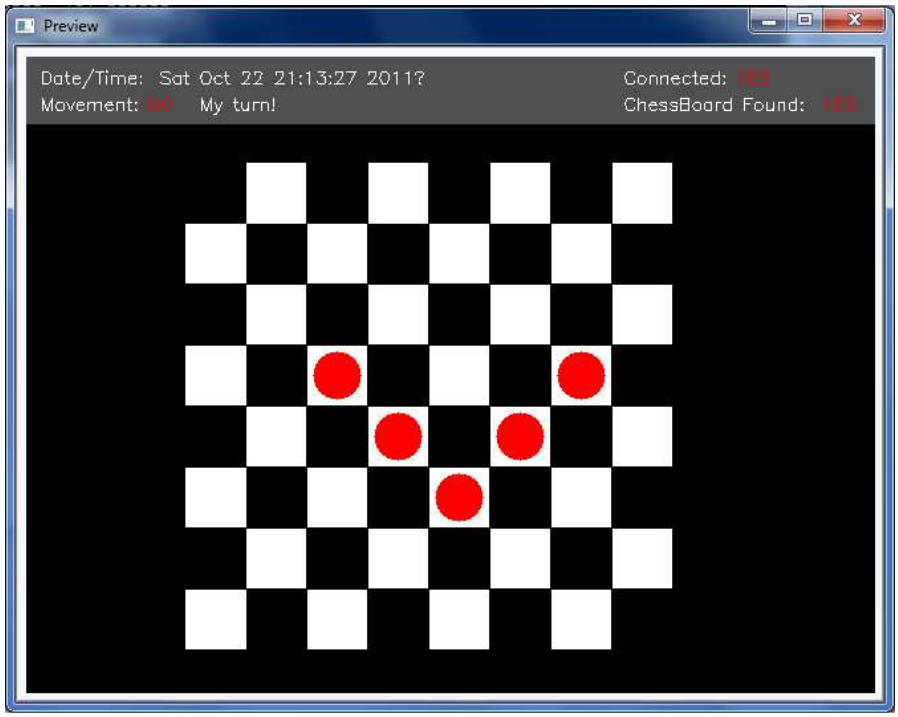
```
#define CHESSBOARD_DIMENSION  8

// Σημεία για το παραλληλόγραμμο της σκακιέρας.

CvPoint pt1, pt2;
CvPoint manpt1;
int i,j,bw; // Ορισμός απαραίτητων μεταβλητών.
bw = 0;
```

Οι τιμές BLOCK_CHESSBOARD_WIDTH και BLOCK_CHESSBOARD_HEIGHT είναι οι διαστάσεις της σκακιέρας που βλέπουμε στην οθόνη σε pixels. Και αυτές έχουν δηλωθεί στην αρχή του προγράμματος.

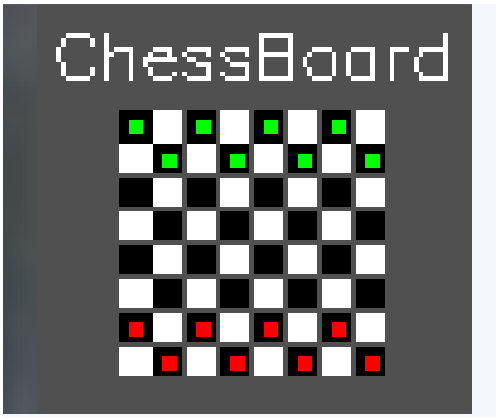
```
#define BLOCK_CHESSBOARD_WIDTH    45    // Οι διαστάσεις της εικονικής
σκακιέρας.
#define BLOCK_CHESSBOARD_HEIGHT 45


// Ζωγραφίζω τη σκακιέρα στην οθόνη και τοποθετώ τα πούλια σε αυτήν.
pt1.x = x;
pt1.y = y;
pt2.x = x + BLOCK_CHESSBOARD_WIDTH;
pt2.y = y + BLOCK_CHESSBOARD_HEIGHT;
for (j=0;j<CHESSBOARD_DIMENSION;j++){
for (i=0;i<CHESSBOARD_DIMENSION;i++){

cvRectangle(img, pt1, pt2, CV_RGB(bw,bw,bw), CV_FILLED);
bw = 255 - bw;
manpt1.x = pt1.x + (BLOCK_CHESSBOARD_WIDTH/2);
manpt1.y = pt1.y + (BLOCK_CHESSBOARD_HEIGHT/2);
```

```
if (chess[i][j] == 1)
	cvCircle(img, manpt1, 18, CV_RGB(100,100,204), CV_FILLED);	//
	Ζωγραφίζω τα μπλε πούλια.
else if (chess[i][j] == 2)
	cvCircle(img,  manpt1,  18,  CV_RGB(255,0,0),  CV_FILLED);	//
	Ζωγραφίζω τα κόκκινα πούλια.
	else if (chess[i][j] == 3)
	cvCircle(img,  manpt1,  18,  CV_RGB(0,255,0),  CV_FILLED);	//
	Ζωγραφίζω τα πράσινα πούλια.
	else if (chess[i][j] == 4)
	cvCircle(img, manpt1, 18, CV_RGB(250,155,0), CV_FILLED);	//
	Ζωγραφίζω τα πορτοκαλί πούλια.


pt1.x = pt1.x + BLOCK_CHESSBOARD_WIDTH + 1;
pt2.x = pt1.x + BLOCK_CHESSBOARD_WIDTH - 1;
}
bw = 255 - bw;
pt1.x = x;
pt1.y = pt1.y + BLOCK_CHESSBOARD_HEIGHT + 1;
pt2.x = x + BLOCK_CHESSBOARD_WIDTH;
pt2.y = pt1.y + BLOCK_CHESSBOARD_HEIGHT - 1;
}
}
```

Η συνάρτηση αυτή δημιουργεί και ενημερώνει την virtual σκακιέρα που εμφανίζεται στην οθόνη μας κατά τη διάρκεια του παιχνιδιού.



Εικόνα 34: Η σκακιέρα που εμφανίζεται στην οθόνη μας (στην οθόνη 'ζωγραφίζονται' αυτόματα τα πούλια που υπάρχουν στην πραγματική σκακιέρα – στη συγκεκριμένη περίπτωση είχαμε πέντε κόκκινα πούλια).



Εικόνα 35: Η σκακιέρα που εμφανιζόταν αρχικά στο interface μας.

2.3.2.2 Η κύρια συνάρτηση του κώδικα

- **Main Program**

Τώρα θα γίνει η παρουσίαση της main (κύριας) συνάρτησης του κώδικα μας.

```
//-----------Κυρίως πρόγραμμα (Main Program)----------

int main()
{
```

Κατ αρχάς γίνονται οι δηλώσεις των απαραίτητων μεταβλητών:

```
    // Μεταβλητές
    CvCapture* capture = 0;
    int i,j,c,tmp1,tmp2;
    int cnt = 0;
    // Σειριακή θύρα
    HANDLE hComm;
    BOOL Connected = TRUE;

    // Ορίζω τους μετρητές για κάθε δυναμικό πίνακα (σύνολο 4 χρώματα, άρα
    και 4 πίνακες.
    int red_counter = 0;
    int green_counter = 0;
    int blue_counter = 0;
    int orange_counter = 0;

    // Ορίζω 4 δυναμικούς πίνακες - έναν για κάθε χρώμα.

    CvPoint2D32f *RedMan;
    CvPoint2D32f *BlueMan;
    CvPoint2D32f *GreenMan;
    CvPoint2D32f *OrangeMan;

    RedMan =  (CvPoint2D32f*)malloc(MAX_MAN*sizeof(CvPoint2D32f));
    BlueMan = (CvPoint2D32f*)malloc(MAX_MAN*sizeof(CvPoint2D32f));
    GreenMan = (CvPoint2D32f*)malloc(MAX_KING*sizeof(CvPoint2D32f));
    OrangeMan = (CvPoint2D32f*)malloc(MAX_KING*sizeof(CvPoint2D32f));

    // Ορισμός των εικονών που χρησιμοποιούνται στο πρόγραμμα.
```

```
    // Μέγεθος της εικόνας.

    CvSize imgSize;
    imgSize.width = FRAME_WIDTH;
    imgSize.height = FRAME_HEIGHT;

    IplImage* greyImage = cvCreateImage(imgSize, IPL_DEPTH_8U, 1);
    IplImage* frame = 0;
    IplImage* Chessframe = 0;
    IplImage*  movingAverage  =  cvCreateImage(imgSize,  IPL_DEPTH_32F,
3);
    IplImage* Previousframe;
    IplImage* difference;
    IplImage* temp;

    // Ορισμός των σημείων που οριοθετούν το παραλληλόγραμμο.
    CvPoint pt1, pt2, pt3, pt4;

    // Ορισμός τιμών του παραλληλογράμμου ώστε να εστιάζει η κάμερα μόνο
στη σκακιέρα (παρόμοια με το ROI – δες πιο πάνω).
    pt3.x = IMAGEROI_PT_1_x;
    pt3.y = IMAGEROI_PT_1_y;
    pt4.x = IMAGEROI_PT_2_x;
    pt4.y = IMAGEROI_PT_2_y;


    // Δημιουργία του φόντου.
    CvFont font;

    // Μεταβλητές.
    int first = 1; // Δείχνει εάν είναι η πρώτη φορά στη λούπα των frames.
    int flag = 1; // Δείχνει κατά πόσο το ρομπότ είναι έτοιμο για παιχνίδι
--> flag=1 το ρομπότ είναι έτοιμο, flag=0 το ρομπότ δεν είναι έτοιμο //
int yourturn =1; // Δείχνει ποιανού η σειρά είναι για να παίξει. Όταν
είναι 1 παίζει ο άνθρωπος, ενώ όταν είναι 0 παίζει το ρομπότ/

    // Γωνίες για την σκακιέρα
    int Found = 0;
    int corners_count;
    CvPoint2D32f                          corners[(CHESSBOARD_DIMENSION-
1)*(CHESSBOARD_DIMENSION-1)];

    // Προσωρινή μνήμη (buffer) για αποθήκευση του αριθμού αντικειμένων
κατά τη μετατροπή από ακέραιο σε συμβολοσειρά.
    char wow[65];
```

Εδώ ορίζουμε τους δύο πίνακες που αναπαριστούν την σκακιέρα. Ο δεύτερος είναι απαραίτητος για την αναδρομή και την σύγκριση πριν και μετά που παίζει ο αντίπαλος του ρομπότ. Αρχικά είναι μηδενικοί και οι δύο. Έπειτα ανάλογα με το χρώμα που έχει το πούλι που είναι σε κάθε κουτί, η μηδενική τιμή αυτή αλλάζει.

```
    // Ο πίνακας της σκακιέρας – Δεν υπάρχουν πούλια – παίχτες έως να τους
ανιχνεύσει το πρόγραμμα έτσι ο πίνακας είναι ο μηδενικός πίνακας.
    int chessboard[CHESSBOARD_DIMENSION][CHESSBOARD_DIMENSION]  = {
                        { 0,  0,  0,  0, 0,  0,  0,  0},
                        { 0,  0,  0,  0, 0,  0,  0,  0},
                        { 0,  0,  0,  0, 0,  0,  0,  0},
```

```
                        { 0,  0,  0,  0, 0,  0,  0,  0},
                        { 0,  0,  0,  0, 0,  0,  0,  0},
                        { 0,  0,  0,  0, 0,  0,  0,  0},
                        { 0,  0,  0,  0, 0,  0,  0,  0},
                        { 0,  0,  0,  0, 0,  0,  0,  0},
                    };

    // Δεύτερος πίνακας της σκακιέρας. Χρειάζεται για να ελέγχουμε τις
όποιες αλλαγές στη σκακιέρα.
    int previouschessboard[CHESSBOARD_DIMENSION][CHESSBOARD_DIMENSION]
= {
                        { 0,  0,  0,  0, 0,  0,  0,  0},
                        { 0,  0,  0,  0, 0,  0,  0,  0},
                        { 0,  0,  0,  0, 0,  0,  0,  0},
                        { 0,  0,  0,  0, 0,  0,  0,  0},
                        { 0,  0,  0,  0, 0,  0,  0,  0},
                        { 0,  0,  0,  0, 0,  0,  0,  0},
                        { 0,  0,  0,  0, 0,  0,  0,  0},
                        { 0,  0,  0,  0, 0,  0,  0,  0},
                    };

    // Σύνδεση με την θύρα COM
    if (OpenCOMport(hComm) == -1){
        Connected = FALSE;
        return -1;
    }

    // Αρχικοποίηση της λήψης εικόνας από την κάμερα.
    capture = cvCaptureFromCAM(DRIVER_ID);
    // Εάν αποτύχει επιστρέφει λάθος (error) και βγαίνει από το πρόγραμμα.
    if( !capture )
    {
        fprintf(stderr,"Capture initialization failed...\n");
        return -1;
    }
    // Ορισμός του μεγέθος των frame που λαμβάνουμε από την κάμερα.

cvSetCaptureProperty(capture,CV_CAP_PROP_FRAME_WIDTH,FRAME_WIDTH);

cvSetCaptureProperty(capture,CV_CAP_PROP_FRAME_HEIGHT,FRAME_HEIGHT);
```

Κατασκευάζουμε το preview παράθυρο:

```
    // Δημιουργία του παραθύρου preview.
    cvNamedWindow("Preview", CV_WINDOW_AUTOSIZE);
    cvNamedWindow("Capture", CV_WINDOW_AUTOSIZE);

    // Αρχικοποίηση του φόντου.
    cvInitFont(&font, CV_FONT_HERSHEY_SIMPLEX, 0.45, 0.45, 0, 1, 8);
    for (;;)
    {
        // Λήψη στιγμιοτύπου από την κάμερα.
        frame = cvQueryFrame(capture);
        if(!frame)
            break;
        // Φιλτράρισμα της εικόνας.
        cvSmooth(frame, frame, CV_GAUSSIAN, 3, 3 ,0 ,0);
```

```
        // Εάν είναι η πρώτη φορά, αρχικοποίησε την εικόνα.
        if(first){
                difference = cvCloneImage(frame);
                temp = cvCloneImage(frame);
                Previousframe = cvCloneImage(frame);
                cvConvertScale(frame, movingAverage, 1.0, 0.0);
                Chessframe = cvCreateImage(cvSize(frame->width*2,frame-
>height*2),frame->depth, frame->nChannels);
                first = 0;
        }
        // αλλιώς, φτιάξε έναν μέσο όρο της κίνησης.
        else{
                cvRunningAvg(frame, movingAverage, 0.020, NULL);
                // Χρονικό φιλτράρισμα (με το προηγούμενο και το τωρινό
στιγμιότυπο).
                cvAddWeighted(Previousframe,0.5,frame,0.5,0,frame);
                cvCopy(frame,Previousframe);
        }
        // Μετατροπή της κλίμακας του 'κινούμενου μέσου όρου'.
        cvConvertScale(movingAverage,temp, 1.0, 0.0);
        // Αφαίρεση του τωρινού στιγμιοτύπου από τον κινούμενο μέσο όρο.
        cvAbsDiff(frame,temp,difference);
        // Μετατροπή της εικόνας στην κλίμακα του γκρι.
        cvCvtColor(difference,greyImage,CV_RGB2GRAY);
```

Είναι σημαντικό να αναφέρουμε ότι κατά βάση το πρόγραμμα θα δουλεύει με εικόνες στην κλίμακα του γκρι.

```
        // Μετατροπή της εικόνας σε 'άσπρο και μάυρο'.
        cvThreshold(greyImage, greyImage, 70, 255, CV_THRESH_BINARY);
        // Φιλτράρισμα της εικόνας με τα φίλτρα Dilate και erode.
        cvDilate(greyImage, greyImage, 0, 18);
        cvErode(greyImage, greyImage, 0, 10);
```

Εδώ καλείται η cvFindContours ώστε να εντοπίσουμε οτιδήποτε κινείται μπροστά στην κάμερα. Όσο το πρόγραμμα 'βλέπει' κίνηση δεν επιτελείται καμία άλλη εργασία.

```
// Εύρεση του περιγράμματος των κινούμενων αντικειμένων στο στιγμιότυπο.
CvMemStorage* storage = cvCreateMemStorage(0);
CvSeq* contour = 0;
cvFindContours(greyImage,   storage,   &contour,   sizeof(CvContour),
CV_RETR_CCOMP, CV_CHAIN_APPROX_SIMPLE, cvPoint(0,0));
cvReleaseMemStorage(&storage);

        // Έλεγχος εάν ευρέθησαν κινούμενα περιγράμματα (ανίχνευση
κίνησης!)
if (contour!=0){

            cvRectangle(Chessframe,           cvPoint(0,           0),
cvPoint(FRAME_WIDTH*2, 50), CV_RGB(80,80,80), CV_FILLED);
            cvPutText(Chessframe, "Date/Time:", cvPoint(10, 20), &font,
CV_RGB(255, 255, 255));
            cvPutText(Chessframe,  current_time(),  cvPoint(100,  20),
&font, CV_RGB(255, 255, 255));
            //cvPutText(frame,    _itoa(ctime(current_day),    wow,    10),
cvPoint(50, 20), &font, CV_RGB(0, 0, 300));
            cvPutText(Chessframe, "Movement:", cvPoint(10, 40), &font,
CV_RGB(255, 255, 255));
```

```
            cvPutText(Chessframe,   "YES",   cvPoint(90,   40),   &font,
CV_RGB(255, 0, 0));

        }
        else{

            cvRectangle(Chessframe,            cvPoint(0,            0),
cvPoint(FRAME_WIDTH*2, 50), CV_RGB(80,80,80), CV_FILLED);
            cvPutText(Chessframe, "Date/Time:", cvPoint(10, 20), &font,
CV_RGB(255, 255, 255));
            cvPutText(Chessframe,  current_time(),  cvPoint(100,  20),
&font, CV_RGB(255, 255, 255));
            cvPutText(Chessframe, "Movement:", cvPoint(10, 40), &font,
CV_RGB(255, 255, 255));
            cvPutText(Chessframe,   "NO",   cvPoint(90,   40),   &font,
CV_RGB(255, 0, 0));
```

Στη συνέχεια εντοπίζεται η σκακιέρα απο την cvFindChessboardCorners. Φυσικά εάν δεν εντοπιστεί η σκακιέρα το πρόγραμμα σταματάει να λειτουργεί μέχρι να εντοπιστεί (όπως ακριβώς και στο motion detection).

```
// Ψάξε για την σκακιέρα και ξεκίνα ανίχνευση χρώματος εάν βρεθεί.

// Μετατροπή της ληφθείσας εικόνας σε κλίμακα του γκρι.
cvCvtColor(frame, greyImage, CV_RGB2GRAY);
Found                                                                  =
cvFindChessboardCorners(greyImage,cvSize(7,7),corners,&corners_count,C
V_CALIB_CB_ADAPTIVE_THRESH);

          if (Found==1){
```

Εάν η σκακιέρα εντοπιστεί το πρόγραμμα προχωρά στον εντοπισμό των πουλιών-παιχτών ανάλογα με το χρώμα τους.

```
cvPutText(Chessframe,  "ChessBoard  Found:",  cvPoint(450,  40),  &font,
CV_RGB(255, 255, 255));
// Ξεκίνα Color Detection ( βρες τα αντικείμενα σύμφωνα με το χρώμα τους).
              ColorDetection(frame,RedMan,GreenMan,BlueMan,OrangeMan);

              cvFindCornerSubPix(greyImage,  corners,  corners_count,
cvSize(    11,    11    ),    cvSize(    -1,    -1    ),    cvTermCriteria(
CV_TERMCRIT_EPS+CV_TERMCRIT_ITER, 30, 0.1 ));

              // Ζωγράφισε τις εσωτερικές γωνίες της ευρεθείσας σκακιέρας.
              //cvDrawChessboardCorners(Resizedframe,cvSize(7,7),  corners,
corners_count,Found);
```

Έπειτα εφαρμόζουμε το χωρικό φίλτρο FilterContours για κάθε χρώμα.

```
// Τρέξε το φίλτρο.

FilterContours(RedMan, red_counter);
FilterContours(GreenMan, green_counter);
FilterContours(BlueMan, blue_counter);
FilterContours(OrangeMan, orange_counter);
```

Ακολούθως σχηματίζουμε το τετράγωνο που κυκλώνει κάθε πούλι, για κάθε χρώμα πάλι.

```
// Ζωγράφισε το παραλληλόγραμμο γύρω από το κινούμενο αντικείμενο.
for (i = 0; i < MAX_MAN; i++){
             if ((RedMan[i].x>0) && (RedMan[i].y>0)){
                tmp1 = (int)(RedMan[i].x+0.5);
                tmp2 = (int)(RedMan[i].y+0.5);
                pt1.x = tmp1 - DIMENSION_ONE;
                pt1.y = tmp2  - DIMENSION_ONE;
                pt2.x = pt1.x + DIMENSION_ONE_DOUBLE;
                pt2.y = pt1.y  + DIMENSION_ONE_DOUBLE;

      cvRectangle(frame, pt1, pt2, CV_RGB(255, 0, 0), 2);
                }
                if ((GreenMan[i].x>0) && (GreenMan[i].y>0)){
                   tmp1 = (int)(GreenMan[i].x+0.5);
                   tmp2 = (int)(GreenMan[i].y+0.5);
                   pt1.x = tmp1 - DIMENSION_ONE;
                   pt1.y = tmp2  - DIMENSION_ONE;
                   pt2.x = pt1.x + DIMENSION_ONE_DOUBLE;
                   pt2.y = pt1.y  + DIMENSION_ONE_DOUBLE;

                   cvRectangle(frame, pt1, pt2, CV_RGB(0, 255, 0),
2);

                }
                if ((BlueMan[i].x>0) && (BlueMan[i].y>0)){
                   tmp1 = (int)(BlueMan[i].x+0.5);
                   tmp2 = (int)(BlueMan[i].y+0.5);
                   pt1.x = tmp1 - DIMENSION_ONE;
                   pt1.y = tmp2  - DIMENSION_ONE;
                   pt2.x = pt1.x + DIMENSION_ONE_DOUBLE;
                   pt2.y = pt1.y  + DIMENSION_ONE_DOUBLE;

                   cvRectangle(frame, pt1, pt2, CV_RGB(0, 0, 255),
2);

                }
                if ((OrangeMan[i].x>0) && (OrangeMan[i].y>0)){
                   tmp1 = (int)(OrangeMan[i].x+0.5);
                   tmp2 = (int)(OrangeMan[i].y+0.5);
                   pt1.x = tmp1 - DIMENSION_ONE;
                   pt1.y = tmp2  - DIMENSION_ONE;
                   pt2.x = pt1.x + DIMENSION_ONE_DOUBLE;
                   pt2.y = pt1.y  + DIMENSION_ONE_DOUBLE;

                   cvRectangle(frame, pt1, pt2, CV_RGB(255, 255, 0),
2);
                }
             }
cvPutText(Chessframe, "ChessBoard Found:", cvPoint(450, 40), &font,
CV_RGB(255, 255, 255));
               cvPutText(Chessframe, "YES", cvPoint(600, 40), &font,
CV_RGB(255, 0, 0)); // Αν η σκακιέρα βρέθηκε, γράψει ΝΑΙ στην οθόνη.

// Βρες τους παίχτες - πούλια και τοποθέτησε τους στην εικόνικη σκακιέρα
στην οθόνη μας (minichessboard).
```

Εδώ μηδενίζουμε τον πίνακα της σκακιέρας ώστε να φορτώσουμε τα νέα δεδομένα από την FindManSquare, την οποία καλούμε τέσσερις φορές – μία για κάθε χρώμα.

```
            for (j=0;j<CHESSBOARD_DIMENSION;j++)
                for (i=0;i<CHESSBOARD_DIMENSION;i++)
                     chessboard[i][j]=0;

          FindManSquare(2, RedMan, corners, chessboard, red_counter);
// Κόκκινα πούλια
          FindManSquare(3,     GreenMan,     corners,     chessboard,
green_counter);  // Πράσινα πούλια
          FindManSquare(1,     BlueMan,     corners,     chessboard,
blue_counter); // Μπλε πούλια
          FindManSquare(4,     OrangeMan,     corners,     chessboard,
orange_counter); // Πορτοκαλί πούλια
         }else{

cvPutText(Chessframe,  "ChessBoard  Found:",  cvPoint(450,  40),  &font,
CV_RGB(255, 255, 255));
                        cvPutText(Chessframe,  "NO",  cvPoint(600,  40),
&font, CV_RGB(255, 0, 0));
// Εάν η σκακιέρα δεν βρεθεί,

 //  γράψε "NO" στην οθόνη
// DO SOMETHING HERE (e.g. ειδοποίησε τον χρήστη ότι κάτι παίζει)
 //PlaySound("sounds/No_chessboard.wav", NULL, SND_FILENAME | SND_SYNC);
// και παίξε αυτό τον ήχο.
```

Εάν η σκακιέρα δεν βρέθηκε τότε το πρόγραμμα αναπαράγει τον αντίστοιχο ήχο ώστε να ειδοποιηθεί ο αντίπαλος και να διορθώσει τυχόν πρόβλημα.

```
         }
         // Γράψε τον αριθμό των αντικειμένων που καταμετρήθηκαν στην
κορυφή του στιγμιοτύπου.

         // Γράψε το κείμενο για την Ημερομηνία και την Ώρα, εάν βρέθηκε η
σκακιέρα, εάν έγινε ανίχνευση κίνησης, εάν συνδέθηκε σωστά κτλ.


       cvPutText(Chessframe,  "Connected:",  cvPoint(450,  20),  &font,
CV_RGB(255, 255, 255));
       if  (Connected)  cvPutText(Chessframe,  "YES",  cvPoint(535,  20),
&font, CV_RGB(255, 0, 0));
       else    cvPutText(Chessframe,  "NO",  cvPoint(535,  20),  &font,
CV_RGB(255, 0, 0));
```

Σχεδιάζουμε την σκακιέρα που βλέπουμε στην οθόνη μας καλώντας την drawchessboard.

```
// Ζωγράφισε την εικόνική σκακιέρα στην οθόνη (mini chessboard).
drawchessboard(Chessframe,120,80,chessboard);
```

Τώρα είναι η κατάλλη στιγμή για να ελεγχθεί εάν ο άνθρωπος – αντίπαλος έπαιξε. Συγκρίνουμε λοιπόν τους δύο πίνακες μεταξύ τους με την CheckManChange και εάν υπάρχει αλλαγή τότε το πρόγραμμα επικοινωνεί με το πρόγραμμα της ντάμας για να παίξει το ρομπότ. Εάν δεν διαπιστωθεί αλλαγή το πρόγραμμα αναπαράγει τον αντίστοιχο ήχο καλώντας τον αντίπαλο του να παίξει, βγάζοντας παράλλη το αντίστοιχο μήνυμα στην οθόνη!

```
// Τώρα σύγκρινε την τωρινή σκακιέρα και την προηγούμενη.
// Εάν ανιχνεύσεις αλλαγή, το ρομπότ αντιλαμβάνεται ότι είναι η σειρά του να παίξει.
// Εάν δεν ανιχνεύσεις αλλαγή, τότε το ρομπότ περιμένει τον άνθρωπο – αντίπαλο να παίξει,
// και του μιλάει αντίστοιχα προτρέποντας τον!.
```

Για να αποφευχθεί λάθος κατα τη σύγκριση παλιάς και νέας σκακιέρας (θεωρητικά το πρόγραμμα βλέποντας ένα στιγμιαίο ‘σκουπίδι’ στην οθόνη θα μπορούσε να αποφανθεί ότι ο αντίπαλος έπαιξε), έχουμε βάλει έναν μετρητή για τα frames έτσι ώστε να θεωρούμε ότι έγινε αλλαγή στη σκακιέρα όταν αυτή διαπιστώνεται στα εννέα από τα δέκα frames (9/10). Η σταθερά FRAME_LOOP έχει οριστεί στην αρχή του προγράμματος ως εξής:

```
#define FRAME_LOOP 10 // Κάθε 10 στιγμιότυπα απαιτούμε απάντηση από το πρόγραμμα.


// Έχουμε μια λούπα εδώ για να εξασφαλίσουμε ότι το πρόγραμμα αντιλαμβάνεται σωστά κάποια αλλαγή στα πούλια της σκακιέρας και όχι κάποια παραμόρφωση της εικόνας ή χρώμα από αντανάκλαση.
// Έτσι ελέγχουμε και απαιτούμε η αλλαγή να γίνει αντιληπτή από το ρομπότ στα 9 από τα 10 στιγμιότυπα για να την θεωρήσουμε ως αλλαγή στη σκακιέρα (δλδ στο 90% των περιπτώσεων).

if (CheckManChange(chessboard, previouschessboard) == 1){    // Εάν η chessboard είναι διαφορετική από την previouschessboard
            cnt++;                                                    // Η CheckManChange επιστρέφει 1 και ο μετρητής cnt αυξάνεται κατά 1.
                if (cnt==FRAME_LOOP-1){   // εάν ο μετρητής φτάσει το 9 (FRAME_LOOP-1)
                ChessBackUp(chessboard, previouschessboard); // Φορτών τα νέα δεδομένα στη Minichessboard
                cnt=0; // Και επαναθέτω τον μετρητή στο 0.
                cvPutText(Chessframe,   "My   turn!",   cvPoint(130,40), &font, CV_RGB(255, 255, 255)); // Το ρομπότ γράφει στην οθόνη "My Turn!".
                cvShowImage("Preview", Chessframe);
                //PlaySound("sounds/Human_not_playing.wav",           NULL, SND_FILENAME | SND_SYNC); // Και παίζει αυτό τον ήχο.

             LetsPlayTheGame(blue_counter,    green_counter,    red_counter, orange_counter, previouschessboard,
                             chessboard);
                cvWaitKey(5000);  // Χρόνος αναμονής πριν παίξει το ρομπότ. Περίπου 5 seconds = 5000 ms.

                // Ειδοποίησε τον χρήστη να παίξει - NEED SOUND
```

Εδώ γίνεται η επικοινωνία του visual κώδικα με το πρόγραμμα της ντάμας που απαντάει στο ρομπότ ποια κίνηση να κάνει.

```
                // Επικοινωνία με το πρόγραμμα των κανόνων της ντάμας ώστε να δώσει απάντηση για την σωστή κίνηση και το ρομπότ να παίξει.
                }

// Απελευθέρωση της μνήμης των δυναμικών πινάκων
```

```
    if (red_counter) free(RedMan);
    if (blue_counter) free(BlueMan);
    if (green_counter) free(GreenMan);
    if (orange_counter) free(OrangeMan);
        }

        cvPutText(Chessframe, "Your turn!", cvPoint(130,40), &font,
CV_RGB(255, 255, 255)); // Το ρομπότ γράφει στην οθόνη "Your Turn!".
        //PlaySound("sounds/Human_playing_after_robot.wav",          NULL,
SND_FILENAME | SND_SYNC); // Το ρομπότ αναπαράγει αυτόν τον ήχο για να
επικοινήσει με τον αντίπαλο.
        // Και δείχνει την εικόνα στο παράθυρο Preview.

        cvShowImage("Preview", Chessframe);
        cvRectangle (frame, cvPoint(pt3.x, pt3.y), cvPoint(pt4.x,
pt4.y), CV_RGB(255, 0, 0), 1); // Ζωγραφίζω το παραλληλόγραμμο που ορίζει
την περιοχή ROI (εστιάζω στην σκακιέρα).
// Κάτι σαν χωρικό φίλτρο.

// Δες την συνάρτηση CheckManChange για περισσότερα.
cvShowImage("Capture", frame);

        // Περίμενε για 10 msecs
        c = cvWaitKey(20);
        // Περίμενε για τον χαρακτήρα διαφυγής (ESC).

        if( (char) c == 27 )
            break;

             if( c == 'g' ){

            cvSaveImage("capture.bmp",frame);
            printf("The frame was saved to capture.bmp");
             }

    }

    ClosePort(hComm); // Τερματισμός του προγράμματος

    // Απελευθέρωση των εικόνων
    cvReleaseImage(&temp);
    cvReleaseImage(&difference);
    cvReleaseImage(&greyImage);
    cvReleaseImage(&movingAverage);
    cvReleaseImage(&Previousframe);
    cvReleaseImage(&Chessframe);
    cvReleaseImage(&frame);
    // Απελευθέρωση της κάμερας
    cvReleaseCapture(&capture);

    // Καταστροφή των παραθύρων
    cvDestroyWindow("Preview");
    cvDestroyWindow("Capture");

    return 0;
}
```

Σχόλιο: Εδώ θα πρέπει να αναφέρουμε, πως ο παραπάνω κώδικας λειτουργεί με δυναμικούς πίνακες (με απώτερο στόχο ο κώδικας Visual να μην ‘ψάχνει’ να γεμίσει τις όποιες κενές θέσεις των στατικών πινάκων με ‘σκουπίδια’ τα οποία δεν συνιστούν πούλια – την βελτιστοποίηση της όρασης του ρομπότ μας δηλαδή). Προφανώς ο κώδικας αυτός είναι η μετεξέλιξη του αντίστοιχου με στατικούς πίνακες. Στο CD της εργασίας θα συμπεριλαμβάνονται και οι δύο.

3. Το παιχνίδι της Ντάμας

3.1 Γενικά

Τα επιτραπέζια παιχνίδια είναι μέρος της ψυχαγωγίας των ανθρώπων χιλιάδες χρόνια τώρα. Το πιο παλιό επιτραπέζιο παιχνίδι έχει χρονολογηθεί κάπου στα 3500 π.Χ. στην Αίγυπτο και είχε το όνομα Σενέτ. Στα παιχνίδια αυτά χρειάζεται δόσεις τύχης και στρατηγικής για να είσαι ο νικητής. Βασιλιάς των παιχνιδιών αυτών είναι το σκάκι (με μπόλικη δόση στρατηγικής και ελάχιστη τύχης). Λίγο πιο κάτω στην ιεραρχία είναι και η ντάμα.

Η Ντάμα είναι μια ομάδα επιτραπέζιων παιχνιδιών στρατηγικής που παίζεται σε σκακιέρα μεταξύ δύο αντιπάλων, των οποίων τα πούλια κινούνται διαγώνια. Το όνομα του παιχνιδιού είναι Draughts στη Μεγάλη Βρετανία ή Checkers στις ΗΠΑ, και αποτελούν εξέλιξη του παιχνιδιού Αλκέρκε (Alquerque) ή Κιρκάτ (Qirkat) το οποίο παιζόταν στη Μέση Ανατολή τον Μεσαίωνα (πρώτη αναφορά τον $10^{ο}$ αι.).

Οι βασικές διαφοροποιήσεις των διαφόρων παιχνιδιών πηγάζουν κυρίως από τις διαστάσεις της σκακιέρας (8x8, 10x10 ή και 12x12 ακόμα!) και από τον αριθμό των πουλιών του κάθε παίχτη (8, 12, 20 κοκ). Κάποιες φορές ίσως να διαφοροποιούνται και οι βασικοί κανόνες (αντί για διαγώνια κίνηση, τα πούλια έχουν μόνο κάθετη)! Είναι γεγονός πως οι παραλλαγές του παιχνιδιού είναι πάρα πολλές. Εύκολα μπορεί κάποιος να βρει την Αργεντίνικη, την Τούρκικη ή ακόμα και Ελληνική ντάμα!

Τα παιχνίδια αυτά ανέκαθεν απασχολούσαν τους προγραμματιστές. Δεν είναι τυχαίο πως στην πολύ αρχή ακόμα της πορείας της προγραμματιστικής επιστήμης, κάμποσα μόλις χρόνια μετά τον πρώτο υπολογιστή (στη μορφή που τον γνωρίζουμε σήμερα), το 1951 και 1952, κατασκευάζονται τα πρώτα προγράμματα που παίζουν σκάκι και ντάμα από τους Ντίτριχ Πρινζ και Κρίστοφερ Στράκλι αντίστοιχα, στο Πανεπιστήμιο του Μάντσεστερ. Ακολουθεί η κατασκευή ενός καινούριου προγράμματος Ντάμας το 1970 στο Πανεπιστήμιου του Ντιουκ το οποίο και θα αναμετρηθεί με το πρόγραμμα του Στράκλι και θα νικήσει! Το 1989 ο Τζόναθαν Σάφερ και η ομάδα του, παρουσιάζουν ένα νέο πρόγραμμα για ντάμα, το Chinook το πρώτο με endgame βάση δεδομένων. Τη δεκαετία του ’90, η δυναμική είσοδος των προσωπικών Η/Υ στη καθημερινή ζωή των περισσοτέρων ανθρώπων θα έχει ως αποτέλεσμα να γραφτούν πολλά νέα προγράμματα ντάμας (ένα από αυτά το Colossus του Martin Bryant), ενώ η ομάδα του Chinook συνέχισε να εξελίσσει το δικό της πρόγραμμα, φτάνοντας εν τέλει τον Ιούλη του 2007 να ανακοινώσει τη λύση του παιχνιδιού (τίτλος του αντίστοιχου άρθρου ‘Checkers is Solved’), και πως ο καλύτερος παίχτης ντάμας το καλύτερο που θα μπορούσε να καταφέρει, θα ήταν να αποσπάσει μια ισοπαλία. Η ντάμα παραμένει αυτή τη στιγμή το πιο πολύπλοκο παιχνίδι το οποίο έχει ‘λυθεί’.

3.2 Η ντάμα του ρομπότ μας

Η επιλογή του παιχνιδιού που θα παίζει το ρομπότ μας ήταν πολύ εύκολη. Επιλέξαμε το πιο διαδεδομένο είδος της ντάμας (η λεγόμενη ‘δυτικού τύπου’). Έτσι λοιπόν, το ρομπότ μας παίζει ντάμα σε μια σκακιέρα διαστάσεων 8x8, και ο κάθε παίχτης έχει στην κατοχή του 8 πούλια, τα οποία κινούνται διαγώνια. Παρ’ όλα αυτά ο προγραμματισμός του ρομπότ έγινε με παραμέτρους έτσι ώστε αλλάζοντας τις παραμέτρους αυτές, να μπορεί να ανταποκριθεί σε παιχνίδι ίδιων κανόνων με αυτό που παίζει τώρα αλλά με μεγαλύτερο αριθμό παιχτών ή σε μεγαλυτέρων διαστάσεων σκακιέρα. Το να παίξει παιχνίδι διαφορετικών κανόνων φυσικά και δεν είναι το ίδιο απλό, αλλά δεν είναι ανέφικτο. Απλά χρειάζεται να αλλαχτεί ο κώδικας του τωρινού παιχνιδιού. Παρακάτω αναφέρονται οι βασικοί κανόνες της ντάμας δυτικού τύπου:

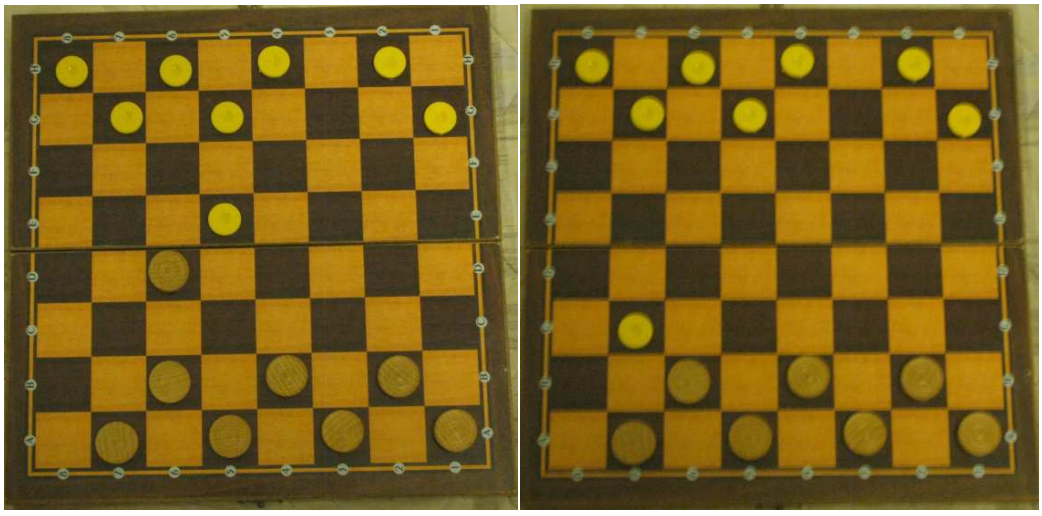
(i.) Κάθε παίχτης έχει 8 πούλια τα οποία τοποθετούνται στα μαύρα τετράγωνα των δύο πρώτων σειρών τετραγώνων της σκακιέρας.

Εικόνα 36: Η αρχική εικόνα του παιχνιδιού. Και οι δύο ομάδες σε θέσεις μάχης!

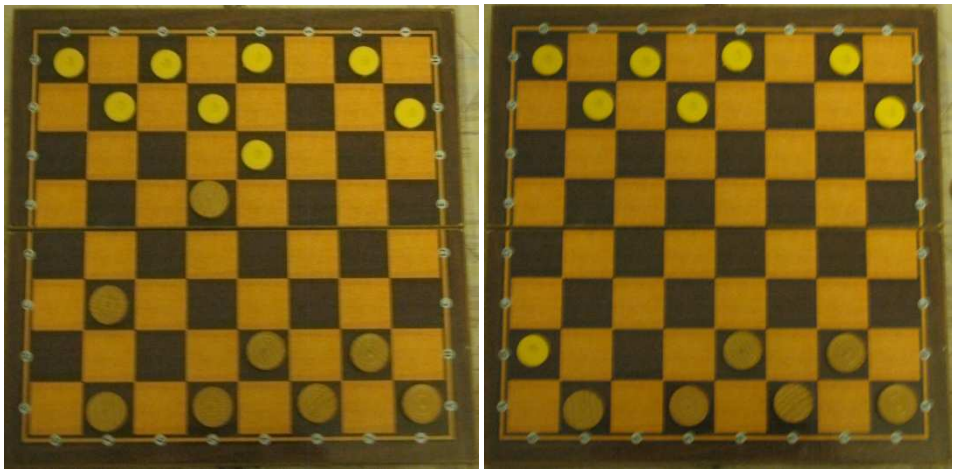
(ii.) Τα πούλια μπορούν να κινηθούν μόνο διαγώνια και κάθε βήμα μπορεί να είναι ένα μόνο τετράγωνο και μόνο προς τα μπροστά.

Εικόνα 37: Από μία κίνηση κάθε αντίπαλος.

(iii.) Εάν ένα πούλι είναι ‘διαγώνια δίπλα’ σε πούλι του αντιπάλου με ελεύθερο το ‘πίσω διαγώνια’ τετράγωνο τότε αυτό το πούλι είναι υποχρεωτικό να φαγωθεί από τον αντίπαλο. Αυτή η αρχή μπορεί να εφαρμοστεί αλυσιδωτά ακόμα και σε μία κίνηση.



Εικόνα 38: Το κίτρινο μπορεί και πρέπει να 'φάει' το καφέ πούλι.



Εικόνα 39: Διαδοχικό 'φάγωμα' δύο καφέ παιχτών!

(iv.) Αναφέρεται ξανά αλλά είναι πολύ σημαντικό για το παιχνίδι ότι εφόσον υπάρχει η δυνατότητα να φαγωθεί πούλι του αντιπάλου, τότε αυτό είναι υποχρεωτικό να συμβεί (ο παίχτης δεν έχει δικαιώμα επιλογής δηλαδή) ακόμα και αν αυτό τον φέρνει σε δυσμενέστερη θέση έναντι του αντιπάλου του στην επόμενη κίνηση. Αυτός ο κανόνας δίνει τη δυνατότητα κατασκευής παγιδών όπου η 'θυσία' ενός παίχτη μπορεί τελικά να αποφέρει μεγάλο κέρδος για την ομάδα του.

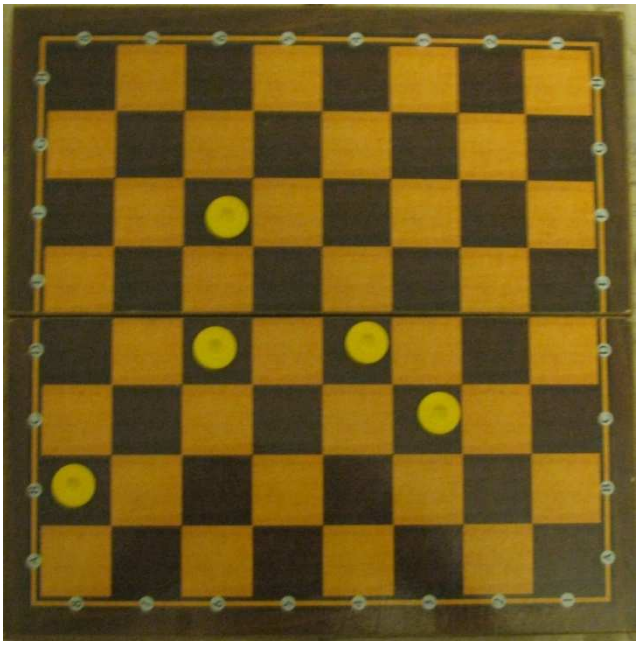
(v.) Εάν κάποιο πούλι φτάσει στο τέρμα της σκακιέρας (στη πρώτη σειρά δηλαδή για τον αντίπαλο παίχτη) τότε το πούλι αυτό μετατρέπεται αυτόματα σε ντάμα (ή king στα αγγλικά) και αλλάζει χρώμα (στο δικό μας παιχνίδι) ή απλά τοποθετείται πάνω από κάποιο άλλο φαγωμένο πούλι του αντιπάλου (στη πραγματικότητα), έτσι ώστε να ξεχωρίζει από τα υπόλοιπα.

Εικόνα 40: Μία ντάμα για την καφέ ομάδα!

(vi.) Η ντάμα κινείται και αυτή μόνο διαγώνια απλά στην απλή της κίνηση μπορεί να διατρέξει πολλά και όχι ένα μόνο τετράγωνο. Η κίνηση αυτή μπορεί να είναι προς τα εμπρός αλλά και προς τα πίσω.

Εικόνα 41: Η ντάμα μπορεί να κινηθεί και προς τα πίσω...

(vii.) Φυσικά νικητής είναι ο παίχτης που θα ‘φάει’ όλα τα πούλια του αντιπάλου.



Εικόνα 42: Νίκη για την κίτρινη ομάδα!

3.3 Οι βασικοί αλγόριθμοι της νοημοσύνης του παιχνιδιού

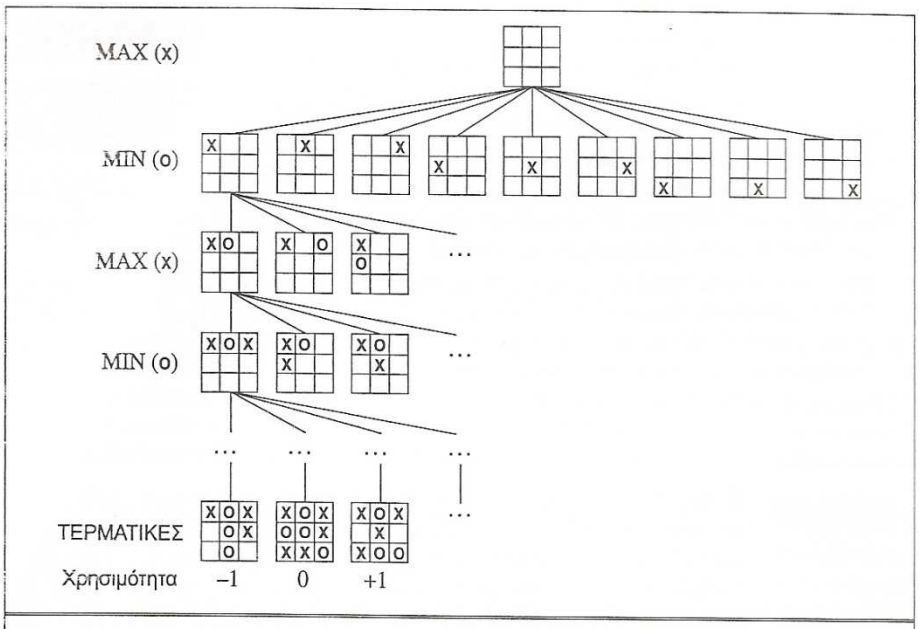
Ο σχεδιασμός και η υλοποίηση του κώδικα που θα ‘απαντάει’ στο ρομπότ τι κίνηση να παίξει δεν αποτέλεσε βασικό στοιχείο της διπλωματικής. Ο κώδικας πάρθηκε έτοιμος και τροποποιήθηκε κατά τις ανάγκες μας. Παρακάτω θα υπάρξει μια σύντομη περιγραφή των βασιών αρχών του.

Στον κώδικα του παιχνιδιού που επικοινωνεί με το ρομπότ μας, εφαρμόζεται ο αλγόριθμος minimax σε συνδυασμό με α-b κλάδεμα (alpha – beta pruning). Ας ρίξουμε όμως μια αναλυτικότερη ματία στους παραπάνω αλγορίθμους:

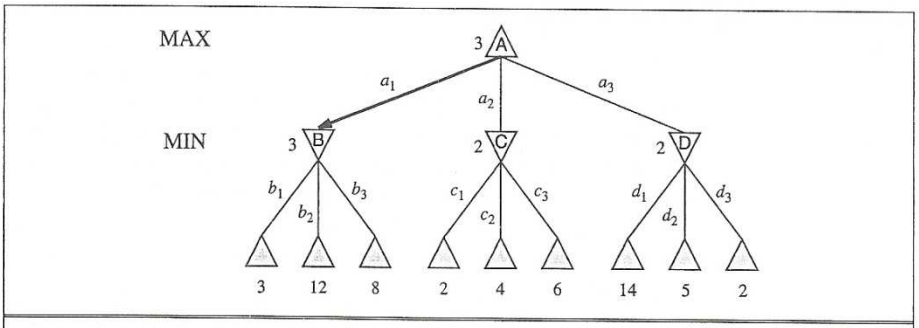
Ουσιαστικά το ζήτημα που αντιμετωπίζουμε είναι το θέμα της ‘βέλτιστης απόφασης’. Θεωρητικά η στρατηγική του ρομπότ μας θα ήταν μία ακολουθία κινήσεων η οποία θα οδηγούσε απο την αρχική κατάσταση στην τερματική (νίκη του παίχτη – ρομπότ) σύμφωνα με κάποιους κανόνες. Όμως αφού το ρομπότ δεν παίζει μόνο του αλλά υπάρχει αντίπαλος ο οποίος ‘φέρνει αντιρρήσεις’ η κατάσταση γίνεται αρκετά πιο πολύπλοκη. Γι’ αυτό τον λόγο το ρομπότ πρέπει να υιοθετήσει μια στρατηγική που θα προσδιορίζει την κίνηση του στην αρχική κατάσταση, και τις κινήσεις του σε κάθε δυνατή απόκριση του αντιπάλου-ανθρώπου, καθώς και τις μετέπειτα απαντήσεις του σε κάθε δυνατή δεύτερη απόκριση του αντιπάλου-ανθρώπου κοκ. Σχηματοποιώντας όλα τα παραπάνω προκύπτει ένα δένδρο αναζήτησης όπου η επιλογή του επόμενου κλαδιού γίνεται εναλλάξ από κάθε παίχτη.

Είναι προφανές ότι ακόμα και για απλά παιχνίδια υπάρχει μεγάλος βαθμός πολυπλοκότητας στο δένδρο αναζήτησης, και πως ο βαθμός δυσκολίας απέναντι σε έναν

αυτόματο αντίπαλο (είτε software παιχνίδι, είτε κάποιο ρομπότ) αυξάνεται αναλογικά με το βάθος που βλέπει και εξετάζει κινήσεις στο δένδρο αναζήτησης (ο αυτόματος αντίπαλος). Παλαιότερα που οι υπολογιστές είχαν περιορισμένες δυνατότητες, το βάθος αυτό περιοριζόταν αρκετά (3-4 κινήσεις). Φανταστείτε ότι και πάλι ο μέσος καλός ανθρώπινος παίχτης μπορεί να σκεφτεί σε βάθος 2-3 κινήσεων. Πλέον βέβαια με την αλματώδη εξέλιξη επεξεργαστών το βάθος αυτό έχει μεγαλώσει.



Εικόνα 6.1 Ένα (μερικό) δένδρο αναζήτησης για το παιχνίδι της τρίλιζας. Ο κόμβος στην κορυφή είναι η αρχική κατάσταση, και ο MAX παίζει πρώτος, τοποθετώντας ένα X σε ένα κενό τετράγωνο. Παρουσιάζουμε ένα μέρος του δένδρου αναζήτησης, δίνοντας εναλλασσόμενες κινήσεις του MIN (O) και του MAX, μέχρι να φτάσουμε τελικά σε τερματικές καταστάσεις, στις οποίες μπορούν να αποδοθούν τιμές χρησιμότητας (utility) σύμφωνα με τους κανόνες του παιχνιδιού.



Εικόνα 6.2 Ένα δένδρο παιχνιδιού δύο στρώσεων (2-ply). Οι κόμβοι Δ είναι "κόμβοι MAX", όπου είναι η σειρά του MAX να κάνει κίνηση, και οι κόμβοι ∇ είναι "κόμβοι MIN". Οι τερματικοί κόμβοι δείχνουν τις τιμές χρησιμότητας για τον MAX· στους άλλους κόμβους είναι σημειωμένες οι αντίστοιχες τιμές minimax. Η καλύτερη κίνηση του MAX στη ρίζα είναι η a_1, επειδή οδηγεί στο διάδοχο κόμβο με τη μεγαλύτερη τιμή minimax, και η καλύτερη απάντηση του MIN είναι η b_1, επειδή οδηγεί στο διάδοχο κόμβο με τη μικρότερη τιμή minimax.

Εικόνα 43: Παραδείγματα δέντρων αναζήτησης (από το βιβλίο Τεχνητή Νοημοσύνη, μία σύγχρονη προσέγγιση των Stuart Russel και Peter Norvig).

Για να κάνουμε πιο εύκολη την παρακάτω ανάλυση θα αναφέρουμε το ρομπότ ως τον παίχτη MAX και τον άνθρωπο ως τον παίχτη MIN.

3.3.1 Ο αλγόριθμος minimax

Ο αλγόριθμος minimax περιγράφεται 'από την ελαχιστοποίηση της πιθανής απώλειας και την ταυτόχρονη μεγιστοποίηση του πιθανού κέρδους (ή συνολικά με την μεγιστοποίηση του ελάχιστου κέρδους)'. Αρχικά σχεδιάστηκε για παιχνίδια δύο ατόμων, είτε εναλλάξ κινήσεων είτε ταυτόχρονων, αλλά πλέον έχει εξελιχτεί ώστε να ανταποκρίνεται σε πιο πολύπλοκα παιχνίδια.

Λαμβάνοντας ως δεδομένο ένα δένδρο παιχνιδιού, η βέλτιστη στρατηγική προσδιορίζεται με την εξέταση της τιμής **minimax** -MINIMAX-VALUE(n)-του κάθε κόμβου. Η τιμή αυτή είναι η χρησιμότητα για τον MAX να βρίσκεται στην αντίστοιχη κατάσταση, με την προϋπόθεση ότι και οι δύο παίκτες παίζουν βέλτιστα μέχρι το τέλος του παιχνιδιού. Προφανώς ο MAX θα επιλέξει να πάει σε μία κατάσταση μέγιστης τιμής ενώ ο MIN σε μία ελάχιστης. Φυσικά αν ο αντίπαλος δεν παίζει βέλτιστα ο MAX θα τα καταφέρει ακόμα πιο εύκολα! Εν τέλει η λειτουργία του αλγορίθμου καταλήγει στην απόφαση minimax (minimax decision) στη ρίζα: Η ενέργεια α1 θα είναι η βέλτιστη επιλογή για τον MAX επειδή οδηγεί στο διάδοχο κόμβο με τη μεγαλύτερη τιμή minimax.

Πως όμως γίνεται όλη η παραπάνω διαδικασία; Ο αλγόριθμος χρησιμοποιεί έναν αναδρομικό υπολογισμό των τιμών minimax κάθε διάδοχης κατάστασης, υλοποιώντας άμεσα τις εξισώσεις που τις ορίζουν. Η αναδρομή κατεβαίνει μέχρι τα φύλλα του δένδρου και μετά οι τιμές minimax αντιγράφονται. Ο αλγόριθμος πραγματοποιεί μια πλήρη εξερεύνηση πρώτα σε βάθος στο δένδρο του παιχνιδιού. Αν το μέγιστο βάθος του δένδρου είναι m και υπάρχουν b νόμιμες κινήσεις σε κάθε κόμβο τότε η χρονική πολυπλοκότητα ειναι $O(b^m)$.

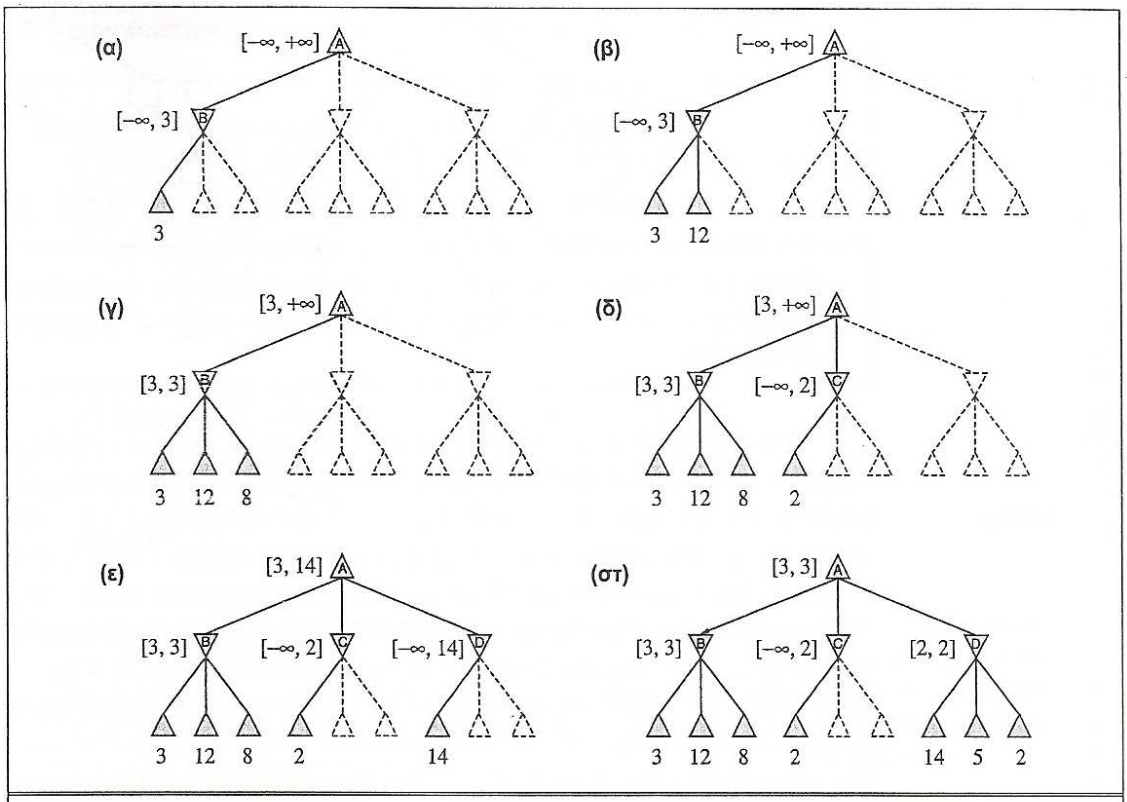
3.3.2 A-b κλάδεμα (Alpha-Beta pruning)

Όπως αναφέρθηκε και παραπάνω η πολυπλοκότητα του Minimax Algorithm είναι πολύ μεγάλη καθώς ο αριθμός των καταστάσεων του παιχνιδιού που πρέπει ο αλγόριθμος να εξετάσει είναι εκθετικός ως προς τις κινήσεις. Η απαλειφή του εκθέτη όμως είναι αδύνατη, άρα το μόνο που μένει να κάνουμε είναι να 'μειώσουμε' τον εκθέτη στο μισό με τον εξής τρόπο: μπορούμε να υπολογίσουμε σωστά την απόφαση minimax χωρίς να κοιτάξουμε όλους τους κόμβους του δένδρου του παιχνιδιού! Η τεχνική αυτή όταν εφαρμόζεται σε δένδρο Minimax επιστρέφει ακριβώς την ίδια κίνηση που θα επέστρεφε ο Minimax μόνος του, με τη σημαντική διαφορά ότι εξαλείφει κλάδους που δεν θα επηρεάζαν την απόφαση ούτως ή άλλως.

Επειδή ακριβώς ο Minimax κάνει αναζήτηση πρώτα σε βάθος, άρα σε κάθε χρονική στιγμή αρκεί να εξετάζουμε τους κόμβους κατά μήκος μίας μόνο διαδρομής του δένδρου. Εάν ορίσουμε τις τιμές α (η τιμή της καλύτερης-μεγαλύτερης επιλογής για τον MAX) και b (η τιμή της καλύτερης-μικρότερης επιλογής για τον MIN), τότε η αναζήτηση α-b ενημερώνει τις τιμές αυτές καθώς προχωρά κλαδεύοντας τους υπόλοιπους κλάδους σε έναν κόμβο μόλις

γίνει γνωστό πς η τιμή του τρέχοντος κόμβου είναι χειρότερη από την τρέχουσα τιμή του α ή του b για τον MAX ή MIN αντίστοιχα.

Η αποτελεσματικότητα του A-B κλαδέματος εξαρτάται σε μεγάλο βαθμό από τη σειρά με την οποία εξετάζονται οι διάδοχες καταστάσεις. Εάν θεωρήσουμε πως είναι δυνατόνα να εξεταστούν πρώτα οι βέλτιστοι διάδοχοι τότε ο αλγόριθμος Minimax με Alpha-Beta pruning χρειάζεται να εξετάσει μόνο $O(b^{m/2})$ κόμβους σε αντίθεση με τον Minimax μόνο του που θα εξέταζε $O(b^m)$.



Εικόνα 6.5 Στάδια του υπολογισμού της βέλτιστης απόφασης για το δένδρο παιχνιδιού της Εικόνας 6.2. Σε κάθε σημείο, παρουσιάζεται το φάσμα των δυνατών τιμών για τον κάθε κόμβο. (α) Το πρώτο φύλλο κάτω από τον κόμβο *B* έχει τιμή 3. Επομένως ο *B*, που είναι κόμβος MIN, έχει τιμή *το πολύ* 3. (β) Το δεύτερο φύλλο κάτω από τον *B* έχει τιμή 12· ο MIN θα απέφευγε αυτή την κίνηση, γι' αυτό η τιμή του *B* είναι και πάλι το πολύ 3. (γ) Το τρίτο φύλλο κάτω από τον *B* έχει τιμή 8· έχουμε δει όλους τους διαδόχους του *B*, γι' αυτό η τιμή του *B* είναι ακριβώς 3. Τώρα μπορούμε να συμπεράνουμε ότι η τιμή της ρίζας είναι *τουλάχιστον* 3, επειδή ο MAX έχει μια επιλογή αξίας 3 στη ρίζα. (δ) Το πρώτο φύλλο κάτω από τον κόμβο *C* έχει τιμή 2. Επομένως ο *C*, που είναι κόμβος MIN, έχει τιμή *το πολύ* 2. Όμως γνωρίζουμε ότι ο *B* αξίζει 3, γι' αυτό ο MAX δε θα διάλεγε ποτέ τον *C*. Δεν έχει νόημα λοιπόν να κοιτάξουμε στους άλλους διαδόχους του *C*. Αυτό είναι ένα παράδειγμα κλαδέματος άλφα–βήτα. (ε) Το πρώτο φύλλο κάτω από τον κόμβο *D* έχει την τιμή 14, γι' αυτό ο *D* αξίζει *το πολύ* 14. Η τιμή αυτή είναι και πάλι μεγαλύτερη από την καλύτερη εναλλακτική επιλογή του MAX (δηλαδή την τιμή 3), γι' αυτό χρειάζεται να συνεχίσουμε να εξερευνούμε τους διαδόχους του *D*. Σημειώστε επίσης ότι τώρα έχουμε φράγματα για όλους τους διαδόχους της ρίζας, γι' αυτό η τιμή της ρίζας είναι επίσης το πολύ 14. (στ) Ο δεύτερος διάδοχος του *D* αξίζει 5, και επομένως χρειάζεται και πάλι να συνεχίσουμε την εξερεύνηση. Ο τρίτος διάδοχος αξίζει 2, και έτσι τώρα ο *D* αξίζει ακριβώς 2. Η απόφαση του MAX στη ρίζα είναι να πάει στον *B*, ο οποίος δίνει τιμή 3.

Εικόνα 44: Παράδειγμα του αλγορίθμου Minimax σε δέντρο παιχνιδιού με a-b κλάδεμα (από το βιβλίο Τεχνητή Νοημοσύνη, μία σύγχρονη προσέγγιση των Stuart Russel και Peter Norvig).

4. Διασύνδεση των επιμέρους τμημάτων κώδικα

4.1 Διασύνδεση λογισμικού

Για τη διασύνδεση των επιμέρους στοιχείων φυσικά χρειάστηκε και άλλος κώδικας. Ο κώδικας αυτός αναφέρεται στην περίληψη και ως Gameplay code, αφου ουσιαστικά θέτει στο μελλοντικό ρομπότ το γενικότερο πλαίσιο μέσα στο οποίο θα παίζει ντάμα. Ο κώδικας της ντάμας πάρθηκε έτοιμος απο τον κ. Martin Fierz και ένα από τα (ελεύθερου κώδικα) project του. Ο Gameplay code ουσιαστικά διασυνδέει τους κανόνες της ντάμας (Checkers code), με το ήδη προγραμματισμένο τσιπ (Chip code) και τον Visual κώδικα.

4.1.1 Ο κώδικας του Gameplay

Οι κανόνες της ντάμας όπως αναφέρθηκε πάρθηκαν έτοιμοι. Έτσι παρακάτω στον κώδικα του Gameplay θα παρατηρήσετε την κλήση μίας συνάρτησης checkers (η οποία με τη σειρά της καλεί άλλες συναρτήσεις όπως η alphabeta). Προς χάριν συντομίας και μιάς και οι συναρτήσεις αυτές πάρθηκαν έτοιμες θα αναφερθεί μόνο η συνάρτηση checkers. Όλος ο κώδικας μπορεί να βρεθεί στο site του δημιουργού του http://www.fierz.ch/checkers.htm.

4.1.1.1 Checkers source code

Ακολουθεί η συνάρτηση που χρησιμοποιήθηκε και αναπαράγει τις κινήσεις του ρομπότ μας:

```
int  checkers(int b[46],int color, double maxtime, char *str, int *play)
      /*----------> purpose: entry point to checkers. find a move on board
b for color
      ---------->             in the time specified by maxtime, write the
best move in
      ---------->          board, returns information on the search in str
      ----------> returns 1 if a move is found & executed, 0, if there is
no legal
      ----------> move in this position.
      ----------> version: 1.1
      ----------> date: 9th october 98 */
{
      int i,numberofmoves;
      clock_t start;
      int eval;
      struct move2 best,lastbest,movelist[MAXMOVES];
      char str2[255];
      double secondsused;
#ifdef STATISTICS
      alphabetas=0;
      generatemovelists=0;
      generatecapturelists=0;
      evaluations=0;
#endif

      /*--------> check if there is only one move */
      numberofmoves=generatecapturelist(b,movelist,color);
      printf ("\n\n I got in Checkers \n\n");
```

```
      if(numberofmoves==1) {

            domove(b,movelist[0]);
            /*sprintf(str,"forced capture");*/return(1);
      }
    else if (numberofmoves == 0) {

       numberofmoves=generatemovelist(b,movelist,color);

            if(numberofmoves==1)
{domove(b,movelist[0]);/*sprintf(str,"only move");*/return(1);}

       if(numberofmoves==0)  {/*sprintf(str,"no  legal  moves  in  this
position");*/return(0);}

    }

      start=clock();
      eval=firstalphabeta(b,1,-10000,10000,color,&best);

      for(i=2;i<=MAXDEPTH;i++)
      {
            lastbest=best;
            eval=firstalphabeta(b,i,-10000,10000,color,&best);
            secondsused = (double)(clock()-start)/CLK_TCK;
            movetonotation(best,str2);
#ifndef MUTE
            //sprintf(str,"best:%s  time  %2.2fs,  depth  %2li,  value
%4li",str2,secondsused,i,eval);
#ifdef STATISTICS
            printf(str2,"  nodes %li, gms %li, gcs %li, evals %li",
                  alphabetas,generatemovelists,generatecapturelists,
                  evaluations);
            //sprintf(str2,"  nodes %li, gms %li, gcs %li, evals %li",
                  //alphabetas,generatemovelists,generatecapturelists,
                  //evaluations);

            //strcat(str,str2);
            //printf(" KOLLAW PIO KATW");
#endif
#endif
            //int play = 0;     //! added by me
            if(*play) break;
            if(eval==5000) break;
            if(eval==-5000) break;
            if (secondsused > maxtime) break;
      }
      i--;
      printf("\n\n I got out of Checkers \n\n");
      if (play) //! added by me if(*play)
            movetonotation(lastbest,str2);
      else
            movetonotation(best,str2);

    //sprintf(str,"best:%s time %2.2f, depth %2li, value %4li  nodes %li,
gms             %li,             gcs             %li,             evals
%li",str2,secondsused,i,eval,alphabetas,generatemovelists,generatecaptureli
sts,evaluations);
```

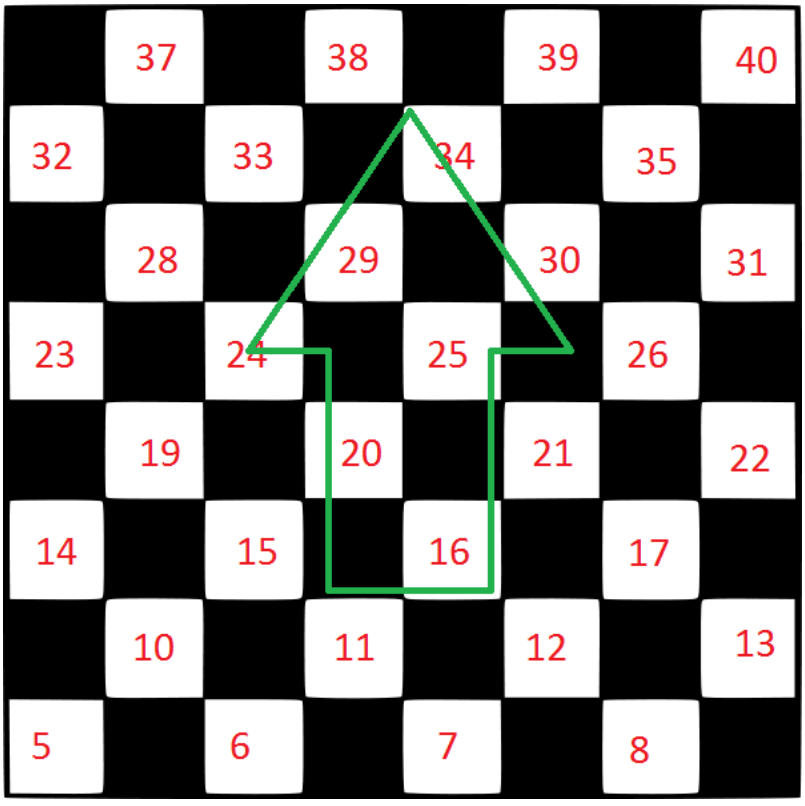
```
if(*play)
	domove(b,lastbest);
else
	domove(b,best);


return eval;
}
```

4.1.1.2 Gameplay code

Προτού γίνει παράθεση του κώδικα του Gameplay, θα πρέπει να αναφέρουμε πως ουσιαστικά όλες οι παρακάτω συναρτήσεις συμπυκνώνονται σε μία συνάρτηση LetsPlayTheGame η οποία και καλείται από τον κώδικα Visual, εφόσον το πρόγραμμα βεβαιωθεί οτι ο άνθρωπος έχει κάνει την κίνηση του.

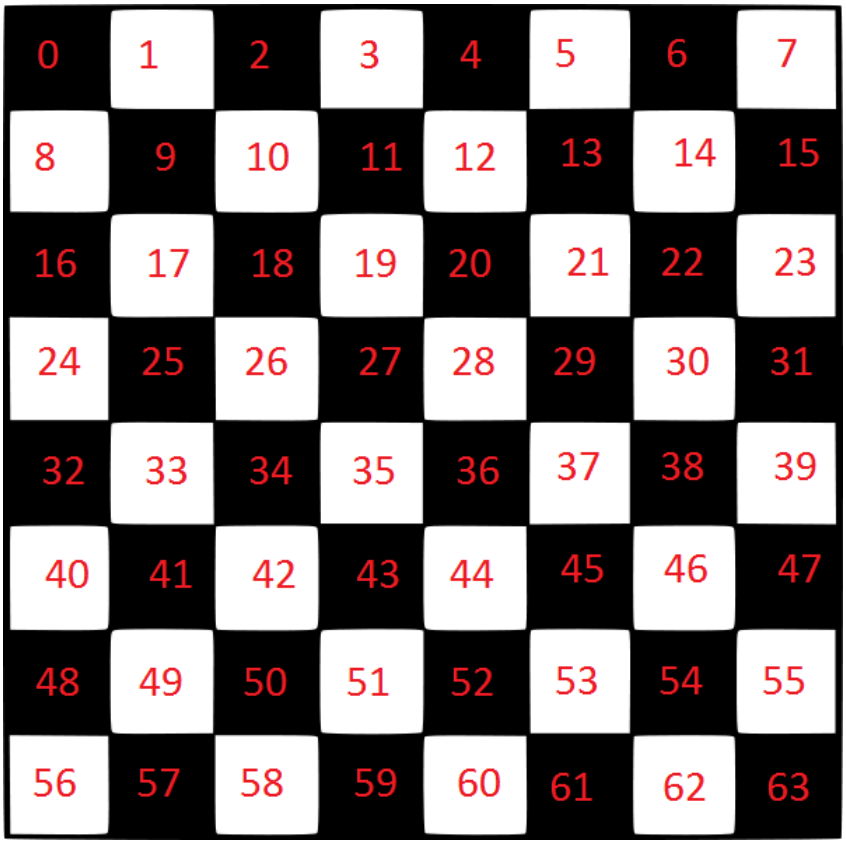
Το πρόγραμμα κάθε φορά που τρέχει επιστρέφει την αρχική και την τελική θέση της εκάστοτε κίνησης, καθώς και τις ενδεχόμενες απώλειες του αντιπάλου (εάν αυτές υπάρχουν).

Η συνάρτηση Checkers που χρησιμοποιήθηκε χρησιμοποιεί σ έναν μονοδιάστατο πίνακα μήκους 46 ακεραίων την εξής αρίθμηση:



Εικόνα 45: Η αρίθμηση του πίνακα και η φορά που παίζει το ρομπότ.

Οι αρχικές και οι τελικές θέσεις που λαμβάνονται, επιστρέφονται σύμφωνα με την παρακάτω αρίθμηση:



Εικόνα 46: Αρίθμηση του Gameplay code.

Επίσης, ενώ στο πρόγραμμα μας οι παίχτες συμβολίζονται με τα νούμερα (0, 1, 2, 3, 4) για (κενό, μπλε, κόκκινο, πράσινο, πορτοκαλί) στο checkers συμβολίζονται με τα νούμερα (16, 5, 6, 9, 10). Για αυτούς ακριβώς τους λόγους χρειάστηκαν εκτός από τη συνάρτηση που εντοπίζει τις κινήσεις (FindOutMove) και την Checkers και δύο συναρτήσεις που θα 'μεταφράζουν' τα δεδομένα ώστε να ανταποκρίνονται και στις δύο αριθμήσεις. Αυτές οι συναρτήσεις ονομάζονται DeltaToCheckers και CheckersToDelta.

Οι βασιλιάδες κάθε ομάδας θα βρίσκονται αριστερά και δεξιά της σκακιέρας. Στον παρακάτω κώδικα συμπεριλαμβάνονται ακόμα:

1.) Η δομή που περιλαμβάνει τις 37 αποθηκευμένες θέσεις του ρομπότ (32 τετράγωνα του παιχνιδιού + 4 θέσεις για τους βασιλιάδες + 1 θέση ισορροπίας).
2.) Η συνάρτηση που φορτώνει τις θέσεις αυτές από ένα text αρχείο.
3.) Οι υπόλοιπες συναρτήσεις που έχουν σχέση με την κίνηση του ρομπότ. Όπως για παράδειγμα εάν φτάσει στο τέλος της σκακιέρας το πούλι μίας ομάδας, καλείται η συνάρτηση ChangeManToKing που αλλάζει το πούλι αυτό με έναν βασιλιά, ενώ αν φαγωθεί ένα πούλι καλείται η συνάρτηση GoManToBin που παίρνει το φαγωμένο πούλι και το πετάει στο καλαθάκι των αχρήστων. Μαζί με αυτές υπάρχουν και συναρτήσεις που ενεργοποιούν/ απενεργοποιούν την ηλεκτρομαγνητική αρπάγη, ελέγχουν εάν κέρδισε κάποιος μετά από κάθε κίνηση ή επιστρέφουν το ρομπότ στη θέση ισορροπίας του (ΘΙ). Όλες αυτές οι συναρτήσεις εν τέλει συμπυκνώνονται στη GoToMan η οποία υλοποιεί την αλλαγή που του επιβάλλει η συνάρτηση Checkers.

Ακολουθεί ο κώδικας αναλυτικά σχολιασμένος:

```
// Οι απαραίτητες βιβλιοθήκες

#include <cv.h>
#include <cxcore.h>
#include <cvaux.h>
#include <highgui.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <windows.h>
#include <mmsystem.h>
#include "simplech1.c"

// Οι απαραίτητες μεταβλητές

#define MAX_EATEN 12
#define CHESSBOARD_DIMENSION 8
#define FORTYSIX 46
#define FORTY 40
#define FIVE 5
#define THIRTYSEVEN 37
#define WHITE 1
#define BLACK 2
#define MAXTIME 5
#define ONE 1
#define DARKTEAM 1
#define LIGHTTEAM 0
#define EIGHT 8

/* Vangelis Angelinos  11/10/2011 Copyright C & R */

/* Η ομάδα του ρομπότ είναι η Σκούρα ομάδα όπου τα πιόνια είναι τα μπλε και
οι βασιλιάδες οι πράσινοι.
    Δίπλα στη σκακιέρα θα υπάρχουν οι 4 βασιλιάδες για κάθε ομάδα. Οι
ακριβείς θέσεις θα σώζονται στη RobotPositions.

Το πρόγραμμα είναι γραμμένο ώστε το ρομπότ να παίζει σύμφωνα με την
κατεύθυνση του βέλους:

    /\
   //\\
    ||
    ||
    ||
    ||


                (black) ==>> ROBOT
              32  31  30  29
            28  27  26  25
              24  23  22  21
            20  19  18  17
              16  15  14  13
            12  11  10   9
               8   7   6   5
             4   3   2   1
                (white) ==>> HUMAN
```

```
Η εσωτερική αναπαράσταση του πίνακα:

                (black) ==>> ROBOT
              37  38  39  40
            32  33  34  35
              28  29  30  31
            23  24  25  26
              19  20  21  22
            14  15  16  17
              10  11  12  13
             5   6   7   8
               (white) ==>> HUMAN

Αυτή είναι η κωδικοποίηση της σκακιέρας για το ρομπότ:

00 01 02 03 04 05 06 07
10 11 12 13 14 15 16 17
20 21 22 23 24 25 26 27
30 31 32 33 34 35 36 37      (πρώτο ψηφίο είναι το i και δεύτερο το j)
40 41 42 43 44 45 46 47
50 51 52 53 54 55 56 57
60 61 62 63 64 65 66 67
70 71 72 73 74 75 76 77

ή

Αυτή είναι η αρίθμηση σύμφωνα με την οποία επιστρέφει το πρόγραμμα τις
αρχικές και τελικές θέσεις της κίνησης.


00 01 02 03 04 05 06 07
08 09 10 11 12 13 14 15
16 17 18 19 20 21 22 23
24 25 26 27 28 29 30 31
32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47
48 49 50 51 52 53 54 55
56 57 58 59 60 61 62 63


Η 'διαδικασία':

Η συνάρτηση Lets play The Game function που την καλούμε από τον vision
κώδικα:
Human plays                 ->
FindoutMove        ->
DeltaToCheckers (translate) ->
checkers (Take answer for robot move) ->
CheckersToDelta (translate back) ->
FindOutMove                 ->
GoToMan                     ->
Human plays again
*/

return 0;}

// ΜΕΡΟΣ Ι  -   Φόρτωση δεδομένων από το αρχείο text.

/*
```

```
Οι θέσεις θα φορτωθούν από το αρχείο στη δομή RobotPositions37.
*/


// Ορισμός της δομής
```

- **Δομή RobotPositions37 για τις θέσεις του ρομπότ**

Η απαραίτητη δομή για να αποθηκεύσουμε τις 37 ζητούμενες θέσεις μεταξύ των οποίων κινείται το ρομπότ.

```
struct RobotPositions37{

 float PWM1;
 float PWM2;
 float PWM3;

} Positions[THIRTYSEVEN];
```

- **LoadPosition συνάρτηση**

Η συνάρτηση αυτή απλά φορτώνει από ένα αρχείο text τα δεδομένα κάθε θέσης στην παραπάνω δομή. Για κάθε θέση προφανώς χρειάζονται 3 παλμοί PWM.

```
// Ορισμός της συνάρτησης.

int LoadPositions(struct RobotPositions37 Pos){

    float number = 0;
    int i = 0;
    FILE *file = fopen("RobotPositions37.txt", "r");

    if(file == NULL){
        printf("Unable to open the file.txt");
        exit(1);
    }
    else{
    fread(Positions,   sizeof(struct   RobotPositions37),   THIRTYSEVEN,
file);
    }


    fclose(file);
    return 0;

}

// Απαραίτητη συνάρτηση ώστε να καλείται κάθε φορά η θέση που χρειάζεται.
```

- **Συνάρτηση για αποστολή παλμών PWM στο ρομπότ**

Η συνάρτηση αυτή απλά αντιστοιχίζει τη θέση που πρέπει να παει το ρομπότ (εμείς για παράδειγμα λαμβάνουμε ένα home_position = 48 – η αντίστοιχη θέση του τετραγώνου 48

είναι αποθηκευμένη στην 24η θέση της δομής), και έπειτα στέλνει τους παλμούς στον κώδικα του τσιπ για να γίνει η κίνηση από το ρομπότ.

```
int ChangePositionNumAndCallPWM(int position){

    if (position == 37) position = 0;
    if (position == 38) position = 1;
    if (position == 39) position = 2;
    if (position == 40) position = 3;
    if (position == 32) position = 4;
    if (position == 33) position = 5;
    if (position == 34) position = 6;
    if (position == 35) position = 7;
    if (position == 28) position = 8;
    if (position == 29) position = 9;
    if (position == 30) position = 10;
    if (position == 31) position = 11;
    if (position == 23) position = 12;
    if (position == 24) position = 13;
    if (position == 25) position = 14;
    if (position == 26) position = 15;
    if (position == 19) position = 16;
    if (position == 20) position = 17;
    if (position == 21) position = 18;
    if (position == 22) position = 19;
    if (position == 14) position = 20;
    if (position == 15) position = 21;
    if (position == 16) position = 22;
    if (position == 17) position = 23;
    if (position == 10) position = 24;
    if (position == 11) position = 25;
    if (position == 12) position = 26;
    if (position == 13) position = 27;
    if (position == 5) position = 28;
    if (position == 6) position = 29;
    if (position == 7) position = 30;
    if (position == 8) position = 31;

    // εντολή

    // Positions 32 - 35 οι βασιλιάδες
    // Position 36 η θέση ισορροπίας
return 0;
}

// ΜΕΡΟΣ ΙΙ  -  Βασικές συναρτήσεις για την LetsPlayTheGame.
```

- **Συνάρτηση PickUp**

Απλά ενεργοποιεί την αρπάγη του σημείου δράσης για να σηκώσει κάποιο πούλι.

```
// Ενεργοποιεί την αρπάγη.
int PickUp(){

      // activate
      // check if not picked up
      // return 1
```

```
    return 0;
}
```

- **Συνάρτηση LeaveDown**

Απενεργοποιεί την αρπάγη του σημείου δράσης για να αφήσει το πούλι.

```
// Απενεργοποιεί την αρπάγη.
int LeaveDown(){ // This is to deactivate the electromagnetic rapture.
      // deactivate
      return 0;
}
```

- **Συνάρτηση GoToStandardPosition**

Δίνει εντολή στο ρομπότ να πάει στη θέση ισορροπίας. Πριν τη μετάβαση του σε κάποια θέση, το ρομπότ πάντα θα πηγαίνει στη θέση ισορροπίας, έτσι ώστε να μη χτυπήσει πουθενά.

```
// Το ρομπότ πάει στη θέση ισορροπίας.
int GoToStandardPosition(){   // Το ρομπότ ενδιάμεσα κάθε κινησής του επιστρέφει στη θέση ισορροπίας ώστε να αποφευχθεί οποιοδήποτε ατύχημα.

      //entoli(RobotPositions37[37]);  //     return 0;
}
```

- **Συνάρτηση GoManToBin**

Η συνάρτηση αυτή στέλνει κάποιο φαγωμένο πούλι του αντιπάλου στο καλαθάκι των αχρήστων.

```
// Το ρομπότ πετάει στο καλαθάκι όποιον παίχτη φαγώθηκε.
int GoManToBin(float position){  // Goes 'eaten man' to bin.

      //entoli(position);
      PickUp;
      /* if PickUp == 1
      // cvputtext "Βάλτο καλύτερα"
      // playsound "Ζητάω βοήθεια" */
      GoToStandardPosition;
      //entoli(BIN_POS);
      LeaveDown;
      GoToStandardPosition;
      return 0;
}
```

- **Συνάρτηση CheckIfWin**

Η συνάρτηση αυτή ελέγχει αν το ρομπότ νίκησε. Την καλούμε μετά από κάθε κίνηση του ρομπότ.

```
// Το ρομπότ ελέγχει αν νίκησε.
int CheckIfWin(int rcnt, int ocnt){

    if ((rcnt == 0) && (ocnt == 0)){
```

```
        //PlaySound("sound/super.wav", NULL, SND_FILENAME | SND_ASYNC); //
sound for robot win
        //cvPutText(Chessframe, "Game Over Loser!", cvPoint(130,40), &font,
CV_RGB(255, 255, 255));
        //void cvGetTextSize(const char* textString, const CvFont* font,
CvSize* textSize, int* baseline)¶ //des auto gia megala grammata
        printf("I WON");
        GoToStandardPosition;
        return 1;
    }
    else return 0;
}
```

- **Συνάρτηση CheckIfLose**

Η συνάρτηση αυτή ελέγχει αν το ρομπότ έχασε. Την καλούμε μετά από κάθε επιβεβαιωμένη κίνηση του ανθρώπου και πριν το ρομπότ απαντήσει.

```
// Το ρομπότ ελέγχει αν έχασε.
int CheckIfLose(int bcnt, int gcnt){

      if ((bcnt == 0) && (gcnt == 0)){
        //PlaySound("sound/super.wav", NULL, SND_FILENAME | SND_ASYNC); //
sound for robot loss
        //cvPutText(Chessframe, "Game Over...for me! :(", cvPoint(130,40),
&font, CV_RGB(255, 255, 255));
        GoToStandardPosition;
        printf("I lost!");
        return 1;
    }
      else return 0;
}
```

- **Συνάρτηση ChangeManToKing**

Αφού γίνει ο έλεγχος εάν κάποιο πούλι τερμάτισε και πρέπει να γίνει ντάμα, τότε καλείται η παρακάτω συνάρτηση που αφήνει το απλό πούλι στο καλαθάκι και το αντικαθιστά με μία ντάμα – βασιλιά άλλου χρώματος.

```
// Αλλάζει τον κανονικό παίχτη με βασιλιά όταν ο πρώτος φτάσει στο τέρμα.
int ChangeManToKing(float position, int cnt){

      GoManToBin(position);
      //entoli(RobotPositions37[31+1+cnt]);
      PickUp;
      /* if PickUp == 1
      // cvputtext "valto kala"
      // playsound "voitha ligo nte" */

      GoToStandardPosition;
      ChangePositionNumAndCallPWM(position);´
      LeaveDown;
      GoToStandardPosition;
      return 0;
}
```

- **Συνάρτηση GoToMan**

Η GoToMan παίρνει σαν ορίσματα αρχική και τελική θέση της κίνησης (source και destination ή home και final position στον κώδικα μας) καθώς και αν πρέπει να φάει κάποιο πούλι του αντιπάλου (εδώ καλείται η GoManToBin). Έπειτα ελέγχει αν έκανε ντάμα και αν νίκησε.

```
// Η GoToMan ουσιαστικά καθοδηγεί το ρομπότ τι να παίξει.
int GoToMan (float home_position, float final_position, int jumps, int
*eaten, int rcnt, int ocnt, int bcnt, int gcnt){

    eaten = (int*)malloc(MAX_EATEN*sizeof(int));
    int counter_dame=0;
    int i;
    printf("\n\n I got in GoToMan \n\n");
      ChangePositionNumAndCallPWM(home_position); // Η αρχική θέση της
κίνησης

    //entoli (home_position);    PickUp;
      /* if PickUp == 1
      // cvputtext "Βάλτο καλύτερα"
      // playsound "Ζητάω βοήθεια" */
      GoToStandardPosition;

      ChangePositionNumAndCallPWM(final_position);

      //entoli (final_position); // Η τελική θέση της κίνησης
    LeaveDown;
    // Έλεγχος για φαγωμένους παίχτες. Εάν η μεταβλητή jumps δεν είναι 0
τότε το ρομπότ τους πάει στο καλαθάκι των αχρήστων.
    if (jumps != 0){

        for (i=0; i<jumps; i++){
            GoManToBin(eaten[i]);
        }

    }
    // Έλεγχος ένα χρειάζεται αλλαγή απλού παίχτη με ντάμα.
      if (final_position < EIGHT){
            ChangeManToKing(final_position,counter_dame);
            counter_dame++;                    // Μετρητής ώστε αν
χρησιμοποιηθεί ο πρώτος βασιλιάς, το ρομπότ να πάει μετά στον δεύτερο κοκ.
            }

    if (CheckIfWin(rcnt, ocnt) == 1){;  // Το ρομπότ ελέγχει εάν νίκησε,
      GoToStandardPosition;
      return 0;    // και επιστρέφει στη θέση ισορροπίας.
      }
      return 0;
    }


// ΜΕΡΟΣ III   -   Συναρτήσεις για επικοινωνία μεταξύ του προγράμματος
αναπαραγωγής κινήσεων και του ρομπότ.
```

```
// Βρίσκει αρχικές και τελικές θέσεις κάθε κίνησης καθώς και φαγωμένους παίχτες. Για ευκολία ο κώδικας χωρίστηκε ανάλογα αν τρέχει για την σκουρόχρωμη ή την ανοιχτόχρωμη ομάδα.
```

- **Συνάρτηση FindOutMove**

Η FindOutMove βρίσκει τις αρχικές και τελικές θέσεις των κινήσεων που έγιναν στη σκακιέρα, αν φαγώθηκαν και πόσα πούλια και επίσης τις θέσεις τους.

```
int                              FindOutMove                              (int chessboard[CHESSBOARD_DIMENSION][CHESSBOARD_DIMENSION],            int prevchess[CHESSBOARD_DIMENSION][CHESSBOARD_DIMENSION],
                int *final_position, int *home_position, int *jumps, int **eaten, int team){
      int i, j;
      int temp1 =0;
      int temp2 = 0;
    int tempjumps = 0;


    *eaten = (int*)malloc(MAX_EATEN*sizeof(int));
    int *tempeaten = (int*)malloc(MAX_EATEN*sizeof(int));
    *eaten = tempeaten;

      for  (i=0;i<=CHESSBOARD_DIMENSION-1;i++){                    for (j=0;j<=CHESSBOARD_DIMENSION-1;j++){
            if (chessboard[i][j] != prevchess[i][j]){
                if ((chessboard[i][j] != 0) && (prevchess[i][j] == 0)){
                    temp1   =   (i+j)   +   (CHESSBOARD_DIMENSION-1)*i;
// Βρίσκω την τελική θέση.
                    *final_position = temp1;
                }
                else if ((chessboard[i][j] == 0) && (prevchess[i][j] != 0)){
                    if (team == DARKTEAM){
                        if ((prevchess[i][j] == 2) || (prevchess[i][j] == 4)){
                            tempjumps++;
                            tempeaten[tempjumps-1]     =     (i+j)     + (CHESSBOARD_DIMENSION-1)*i;  // Αποθηκεύω την θέση του φαγωμένου παίχτη.

                        }
                        else    if    ((prevchess[i][j]    ==    1)    || (prevchess[i][j] == 3)){
                            temp2 = (i+j) + (CHESSBOARD_DIMENSION-1)*i;
// Βρίσκω την αρχική θέση.
                            *home_position = temp2;
                        }

                    }
                    else if (team == LIGHTTEAM){
                        if ((prevchess[i][j] == 1) || (prevchess[i][j] == 3)){
                            tempjumps++;         // Μετράω τα jumps.
                            tempeaten[tempjumps-1]     =     (i+j)     + (CHESSBOARD_DIMENSION-1)*i;  // Αποθηκεύω την θέση του φαγωμένου παίχτη.
*eaten = tempeaten;
```

```
                        }
                        else    if    ((prevchess[i][j]    ==    2)    ||
(prevchess[i][j] == 4)){
                            temp2 = (i+j) + (CHESSBOARD_DIMENSION-1)*i;
// Βρίσκω την αρχική θέση.

                            *home_position = temp2;
                        }
                    }
                }
            }
        }
    }
    *jumps = tempjumps;
    return 0;
    }
```

- **Συνάρτηση DeltaToCheckers**

Όπως αναφέρθηκε παραπάνω (δες εικόνα 55 και 56 ή τα εισαγωγικά - αρχικά σχόλια του κώδικα), είναι απαραίτητη μια μετάφραση των δεδομένων που έχουμε στη σκακιέρα σε έναν πίνακα Board[46] που χρησιμοποιεί η συνάρτηση checkers. Αυτό επιτελεί η παρούσα συνάρτηση.

```
int                         DeltaToCheckers                         (int
chessboard[CHESSBOARD_DIMENSION][CHESSBOARD_DIMENSION],
                    int **Board46){

    int i,j;
    int *temp46 = (int*)malloc(FORTYSIX*sizeof(int));
    *Board46 = (int*)malloc(FORTYSIX*sizeof(int));
    *Board46 = temp46;

    // Μετατροπή των ψηφίων.

    for (i=0;i<=CHESSBOARD_DIMENSION-1;i++){
        for (j=0;j<=CHESSBOARD_DIMENSION-1;j++){
            if(chessboard[i][j] == 2){
                        chessboard[i][j] = 5; // 1 για λευκή ομάδα και 4
για απλό πούλι (1|4)
                        }
                    else if (chessboard[i][j] == 4){
                        chessboard[i][j] = 9; // 1 για λευκή ομάδα και
84 για βασιλιά (1|8)
                        }
                    else if (chessboard[i][j] == 3){
                        chessboard[i][j] = 10;  // 2 για μαύρη ομάδα και
8 για βασιλιά (2|8)
                        }
                    else if (chessboard[i][j] == 1){
                        chessboard[i][j] = 6;  // 2 για μαύρη ομάδα και
4 για απλό πούλι (2|4)
                        }
            else if (chessboard[i][j] == 0){
                    chessboard[i][j] = 16;
                    }
             }
```

```
}


for (i=0;i<=CHESSBOARD_DIMENSION-1;i++){
 for (j=0;j<=CHESSBOARD_DIMENSION-1;j++){
     if (i == 0){
         if (j==1){
             temp46[37] = chessboard[i][j];}
         else if (j == 3){
             temp46[38] = chessboard[i][j];}
         else if (j == 5){
             temp46[39] = chessboard[i][j];}
         else if (j == 7){
             temp46[40] = chessboard[i][j];}
     }

     if (i == 1){
         if (j == 0){
             temp46[32] = chessboard[i][j];}
         else if (j == 2){
             temp46[33] = chessboard[i][j];}
         else if (j == 4){
             temp46[34] = chessboard[i][j];}
         else if (j == 6){
             temp46[35] = chessboard[i][j];}
     }

     if (i == 2){
         if (j == 1){
             temp46[28] = chessboard[i][j];}
         else if (j == 3){
             temp46[29] = chessboard[i][j];}
         else if (j == 5){
             temp46[30] = chessboard[i][j];}
         else if (j == 7){
             temp46[31] = chessboard[i][j];}
     }

     if (i == 3){
         if (j == 0){
             temp46[23] = chessboard[i][j];}
         else if (j == 2){
             temp46[24] = chessboard[i][j];}
         else if (j == 4){
             temp46[25] = chessboard[i][j];}
         else if (j == 6){
             temp46[26] = chessboard[i][j];}
     }

     if (i == 4){
         if (j == 1){
             temp46[19] = chessboard[i][j];}
         else if (j == 3){
             temp46[20] = chessboard[i][j];}
         else if (j == 5){
             temp46[21] = chessboard[i][j];}
```

```
            else if (j == 7){
                temp46[22] = chessboard[i][j];}
        }

        if (i == 5){
            if (j == 0){
                temp46[14] = chessboard[i][j];}
            else if (j == 2){
                temp46[15] = chessboard[i][j];}
            else if (j == 4){
                temp46[16] = chessboard[i][j];}
            else if (j == 6){
                temp46[17] = chessboard[i][j];}
        }

        if (i == 6){
            if (j == 1){
                temp46[10] = chessboard[i][j];}
            else if (j == 3){
                temp46[11] = chessboard[i][j];}
            else if (j == 5){
                temp46[12] = chessboard[i][j];}
            else if (j == 7){
                temp46[13] = chessboard[i][j];}
        }

        if (i == 7){
            if (j == 0){
                temp46[5] = chessboard[i][j];}
            else if (j == 2){
                temp46[6] = chessboard[i][j];}
            else if (j == 4){
                temp46[7] = chessboard[i][j];}
            else if (j == 6){
                temp46[8] = chessboard[i][j];}
        }
      }
    }
    // Τοποθετούμε μηδενικά στις θέσεις του πίνακα όπου χρειάζεται (σύμφωνα
με το πρόγραμμα).
    temp46[9] = temp46[18] = temp46[27] = temp46[36] = 0;
    for (i=0;i<5;i++){temp46[i]=0;}
    for (i=41;i<=46;i++){temp46[i]=0;}

return 0; }
```

- **Συνάρτηση CheckersToDelta**

Φυσικά η CheckersToDelta κάνει ακριβώς το αντίστροφο σε σχέση με την DeltaToCheckers, ώστε να μετατρέψουμε την κίνηση που μας επέστρεψε η συνάρτηση Checkers σε κίνηση στη σκακιέρα μας.

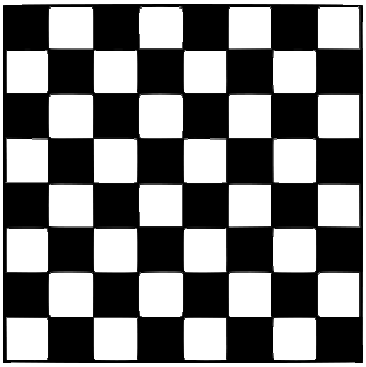
```
int CheckersToDelta (int temp46[FORTYSIX], int chessboard
[CHESSBOARD_DIMENSION][CHESSBOARD_DIMENSION]){

      int i;
```

```
for (i=FIVE;i<=FORTY;i++){
    if(temp46[i] == 5){
        temp46[i] = 2;
    }
    else if (temp46[i] == 9){
        temp46[i] = 4;
    }
    else if (temp46[i] == 10){
        temp46[i] = 3;
    }
    else if (temp46[i] == 6){
        temp46[i] = 1;
    }
    else if (temp46[i] == 16){
                temp46[i] = 0;
    }
}


for (i=FIVE;i<=FORTY;i++){
    if (i == 5){
        chessboard[7][0] = temp46[i];
        chessboard[7][1] = 0;}
    else if (i == 6){
        chessboard[7][2] = temp46[i];
        chessboard[7][3] = 0;}
    else if (i == 7){
        chessboard[7][4] = temp46[i];
        chessboard[7][5] = 0;}
    else if (i == 8){
        chessboard[7][6] = temp46[i];
        chessboard[7][7] = 0;}
    else if (i == 9){
        /*! Do nothing */}
    else if (i == 10){
        chessboard[6][1] = temp46[i];
        chessboard[6][0] = 0;}
    else if (i == 11){
        chessboard[6][3] = temp46[i];
        chessboard[6][2] = 0;}
    else if (i == 12){
        chessboard[6][5] = temp46[i];
        chessboard[6][4] = 0;}
    else if (i == 13){
        chessboard[6][7] = temp46[i];
        chessboard[6][6] = 0;}
    else if (i == 14){
        chessboard[5][0] = temp46[i];
        chessboard[5][1] = 0;}
    else if (i == 15){
        chessboard[5][2] = temp46[i];
        chessboard[5][3] = 0;}
    else if (i == 16){
        chessboard[5][4] = temp46[i];
        chessboard[5][5] = 0;}
    else if (i == 17){
        chessboard[5][6] = temp46[i];
```

```
    chessboard[5][7] = 0;}
else if (i == 18){
    /*! Do nothing */}
else if (i == 19){
    chessboard[4][1] = temp46[i];
    chessboard[4][0] = 0;}
else if (i == 20){
    chessboard[4][3] = temp46[i];
    chessboard[4][2] = 0;}
else if (i == 21){
    chessboard[4][5] = temp46[i];
    chessboard[4][4] = 0;}
else if (i == 22){
    chessboard[4][7] = temp46[i];
    chessboard[4][6] = 0;}
else if (i == 23){
    chessboard[3][0] = temp46[i];
    chessboard[3][1] = 0;}
else if (i == 24){
    chessboard[3][2] = temp46[i];
    chessboard[3][3] = 0;}
else if (i == 25){
    chessboard[3][4] = temp46[i];
    chessboard[3][5] = 0;}
else if (i == 26){
    chessboard[3][6] = temp46[i];
    chessboard[3][7] = 0;}
else if (i == 27){
    /*! Do nothing */}
else if (i == 28){
    chessboard[2][1] = temp46[i];
    chessboard[2][0] = 0;}
else if (i == 29){
    chessboard[2][3] = temp46[i];
    chessboard[2][2] = 0;}
else if (i == 30){
    chessboard[2][5] = temp46[i];
    chessboard[2][4] = 0;}
else if (i == 31){
    chessboard[2][7] = temp46[i];
    chessboard[2][6] = 0;}
else if (i == 32){
    chessboard[1][0] = temp46[i];
    chessboard[1][1] = 0;}
else if (i == 33){
    chessboard[1][2] = temp46[i];
    chessboard[1][3] = 0;}
else if (i == 34){
    chessboard[1][4] = temp46[i];
    chessboard[1][5] = 0;}
else if (i == 35){
    chessboard[1][6] = temp46[i];
    chessboard[1][7] = 0;}
else if (i == 36){
    /*! Do nothing */}
else if (i == 37){
    chessboard[0][1] = temp46[i];
```

```
            chessboard[0][0] = 0;}
        else if (i == 38){
            chessboard[0][3] = temp46[i];
            chessboard[0][2] = 0;}
        else if (i == 39){
            chessboard[0][5] = temp46[i];
            chessboard[0][4] = 0;}
        else if (i == 40){
            chessboard[0][7] = temp46[i];
            chessboard[0][6] = 0;}
    }
    return 0;
}
```

Παρατήρηση: Προς χάριν προσαρμοστικότητας του κώδικα, προφανώς είναι εύκολο να τροποποιηθεί ώστε να παίζει στα 'εσωτερικά' τετράγωνα της σκακιέρας (ξεκινώντας από πάνω αριστερά). Το χρώμα δεν έχει ιδιαίτερη σημασία καθώς αυτό εξαρτάται από το πως τοποθετείς την σκακιέρα. Έτσι ενώ ο παραπάνω κώδικας παίζει στα μαύρα τετράγωνα, εύκολα μπορούμε να τον τροποποιήσουμε για να παίζει στα άσπρα, αλλάζοντας απλά τις αναθέσεις των τετραγώνων στις DeltaToCheckers και CheckersToDelta (δες παράρτημα Γ).



Εικόνα 47: Έτσι είναι τοποθετημένη η σκακιέρα μας στο project. Σε ποια τετράγωνα θα παίξουμε εξαρτάται καθαρά από την απόδοση του κώδικα Visual (κατα πόσον είναι καλύτερη στα άσπρα ή στα μαύρα τετράγωνα). Αυτό όμως δεν αποτελεί πρόβλημα, όπως αναφέρθηκε παραπάνω. Απλά θα αλλάξουν 'θέση' οι δύο αντίπαλοι! Σύμφωνα με τον παραπάνω κώδικα παίζουμε στα άσπρα, με εκκίνηση για το ρομπότ από το κάτω μέρος της σκακιέρας, και για τον άνθρωπο από το πάνω μέρος.

```
// ΜΕΡΟΣ IV  -  Τελική συνάρτηση που καλείται από τον κώδικα Vision.
```

- **Συνάρτηση LetsPlayTheGame**

Η LetsPlayTheGame όπως αναφέρθηκε και πριν, συμπυκνώνει όλο το Gameplay καλώντας όλες τις παραπάνω συναρτήσεις καθώς και την συνάρτηση checkers που αναπαράγει τις κινήσεις που 'απαντάει' το ρομπότ. Με τη σειρά της καλείται από τον κώδικα Visual, αφού επιβεβαιωθεί κίνηση του ανθρώπου - αντιπάλου.

```
// PART IV  -  Η τελική συνάρτηση που θα καλείται από τον κώδικα vision.


int LetsPlayTheGame(int bcnt, int gcnt, int rcnt, int ocnt, int
prevchess[CHESSBOARD_DIMENSION][CHESSBOARD_DIMENSION],
```

```
                    int
chessboard[CHESSBOARD_DIMENSION][CHESSBOARD_DIMENSION]){

      // Οι μεταβλητές μου
    int home_position, final_position, jumps1;
    int home_position2, final_position2, jumps2;
    int i, j;

    int *EatenMan1, *EatenMan2;
    EatenMan1 = (int*)malloc(MAX_EATEN*sizeof(int));
    EatenMan2 = (int*)malloc(MAX_EATEN*sizeof(int));


    int *Board46 = (int*)malloc(FORTYSIX*sizeof(int));
    char *info_checkers;
    int play = 0;

    // Ας ξεκινήσουμε!

    // Το ρομπότ λαμβάνει την κίνηση που έκανε ο άνθρωπος.
    FindOutMove(chessboard,        prevchess,        &final_position,
&home_position, &jumps1, &EatenMan1, LIGHTTEAM);

    // Πληροφορίες
    printf("\n\n ********************** \n\n");
    printf("\n\n This is Human's Movement \n\n");
    printf("\n\n  Home  position  is  %d  -->  Final  position  is  %d  \n\n",
home_position, final_position);
    printf("\n\n Jumps is %d \n\n", jumps1);
    for (i=0;i<jumps1;i++){
        printf("\n\n EatenMan1[i] is %d", EatenMan1[i]);
    }
    printf("\n\n ********************** \n\n");

    //  Φορτώνουμε  την  'προηγούμενη'  σκακιέρα  σε  νέα  μεταβλητή  για
μελλοντικές συγκρίσεις.
    for (i=0; i<=CHESSBOARD_DIMENSION-1; i++){
        for (j=0;j<=CHESSBOARD_DIMENSION-1;j++){
            prevchess[i][j] = chessboard[i][j];
        }
    }

    // 'Μετάφραση' μεταξύ ρομπότ και προγράμματος.
    DeltaToCheckers(chessboard, &Board46);


    printf("\n
========================================================================\n")
;
    printf("\n Your move is supposed to be valid! WARNING: DO NOT CHEAT!!!
\n");
    printf("\n
========================================================================\n")
;


    int Lost = CheckIfLose(bcnt, gcnt);  // Το ρομπότ ελέγχει αν έχασε
    if (Lost == 1){goto loop_point_lost_or_win;}
```

```
    // Επικοινωνεί με το πρόγραμμα που θα του πει τη κίνηση να κάνει.

    checkers(Board46, BLACK, MAXTIME, info_checkers, &play);

    // ‘Μετάφραση’ (αντίστροφα από οτι πριν).
    CheckersToDelta (Board46, chessboard);

    // Βρίσκω τη κίνηση είπε το παιχνίδι να κάνω.
    FindOutMove(chessboard,        prevchess,        &final_position2,
&home_position2, &jumps2, &EatenMan2, DARKTEAM);

    // Πληροφορίες
    printf("\n\n ***************************** \n\n");
    printf("\n\n This is Delta Robot's Movement \n\n");
    printf("\n\n Home position 2 is %d --> Final position 2 is %d \n\n",
home_position2, final_position2);
    printf("\n\n Jumps2 is %d \n\n", jumps2);
    for (i=0;i<jumps2;i++){
        printf("\n\n EatenMan2[i] is %d", EatenMan2[i]);
    }
    printf("\n\n ***************************** \n\n");

    if (final_position2 > 55) printf("\n\n Delta Robot speaking: I have a
King now!!! \n\n");

    // Επικοινωνία με το ρομπότ για να παίξει!
    GoToMan(home_position2, final_position2, jumps2, EatenMan2, rcnt,
ocnt);  // CheckIfWin  είναι ενσωματωμένη στη συνάρτηση GoToMan.

    loop_point_lost_or_win:
    return 0;
}
```

4.1.2 Ο Προγραμματισμός του ρομπότ

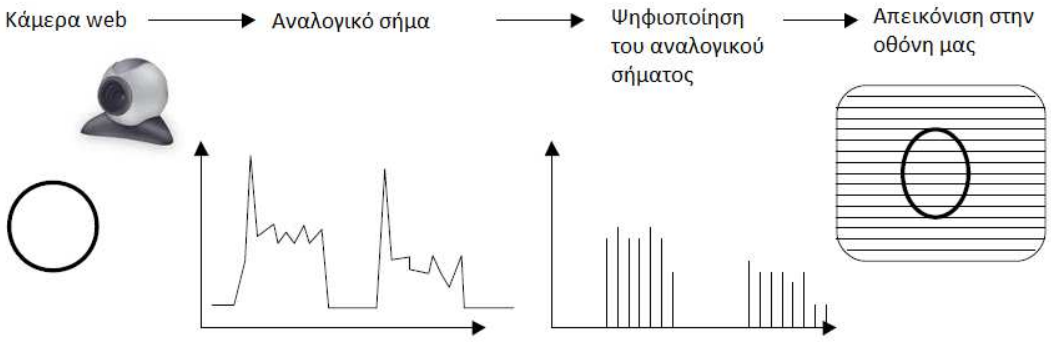
Για τη λειτουργία του ρομπότ δεν μένει πλέον παρά να συνδέσουμε το ρομπότ στη σειριακή θύρα του υπολογιστή και να τρέξουμε τα προγράμματα μας. Εδώ θα πρέπει να αναφέρουμε πως το chip της πλακέτας του ρομπότ (ATMEGA8) είναι προγραμματισμένο σε γλώσσα BASIC (αντίστοιχα χρησιμοποιήθηκε το πρόγραμμα BASCOM) και λειτουργεί με 2 τρόπους (modes). Ο πρώτος είναι απλά στέλνοντας του 3 παλμούς PWM που αυτό αποστέλλει στους σερβοκινητήρες ώστε να γίνει η κίνηση, και ο δεύτερος απλα στέλνοντας του τις συντεταγμένες του παίχτη που θες το ρομπότ να κινήσει. Φυσικά το πρόγραμμα λύνει το αντίστροφο κινηματικό πρόβλημα και μεταφράζει τις συντεταγμένες αυτές σε παλμούς PWM με την βοήθεια της χαρακτηριστικής ευθείας κάθε σερβοκινητήρα (δες κεφάλαιο 6). Ο προγραμματισμός του τσιπ φυσικά βασίζεται στην μαθηματική ανάλυση του κεφαλαίου 0.

4.2 Βαθμονόμηση της κάμερας (Camera Calibration)

4.2.1 Εισαγωγή

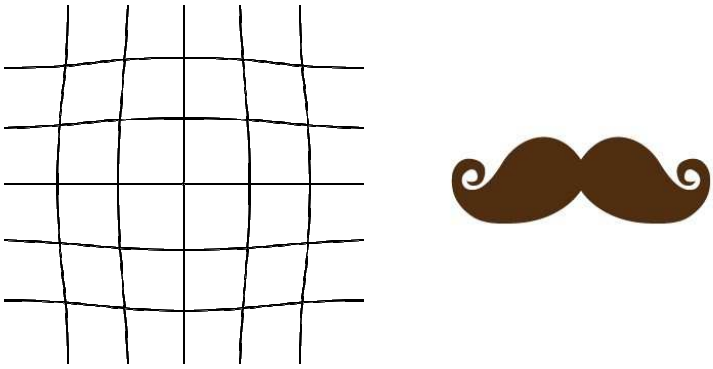
Φυσικά απαραίτητο για να ανταποκριθεί ο ρομποτικός βραχίονας στις απαιτήσεις του προγράμματος και να επιτελεί την εργασία του με ακρίβεια είναι η βαθμονόμηση της κάμερας (ή καλύτερα στα αγγλικά calibration).

Γιατί όμως η βαθμονόμηση της κάμερας είναι απαραίτητη και τι ακριβώς επιτελεί; Κάθε εικόνα που απεικονίζεται από μια κάμερα δεν είναι ακριβώς ίδια με την 'πραγματική εικόνα'. Κατά τη διαδικασία της απεικόνισης διάφοροι εσωτερικοί παράγοντες της κάμερας επηρεάζουν την εικόνα που βλέπουμε εν τέλει στο μόνιτορ.

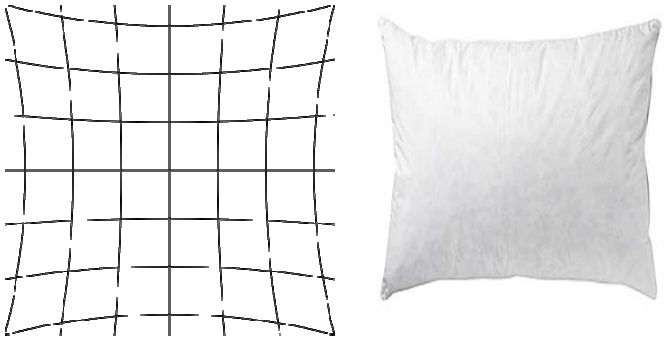


Εικόνα 48: Γραφική αναπαράσταση της διαδικασίας απεικόνισης. Παρατηρήστε πως ο κύκλος απεικονίζεται σαν οβάλ σχήμα.

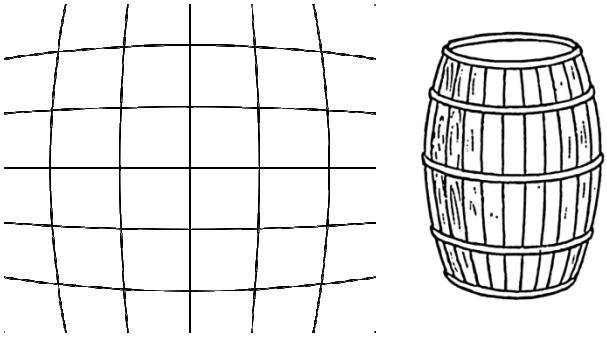
Για παράδειγμα, συνήθως το κέντρο της εικόνας στην απεικόνιση δεν βρίσκεται ακριβώς στο σημείο Σ (πλάτος/2, ύψος/2). Φυσικά υπάρχουν διάφορα είδη παραμόρφωσης (Pin-cushion distortion, barrel distortion, moustache distortion – δες τις εικόνες παρακάτω) τα οποία οφείλονται και σε διαφορετικούς λόγους, με βασικές αιτίες τους διαφορετικούς συντελεστές των Pixel στο πλάτος και στο ύψος, το εστιακό μήκος αλλά και την παραμόρφωση του φακού (που ειδικά στην δικιά μας περίπτωση δεν είναι ιδιαίτερης ποιότητας υποβαθμίζοντας έτσι και άλλο την απεικονισή μας).



Εικόνα 49: Βασικές παραμορφώσεις (εδώ Moustache distortion – παραμόρφωση μουστακιού).

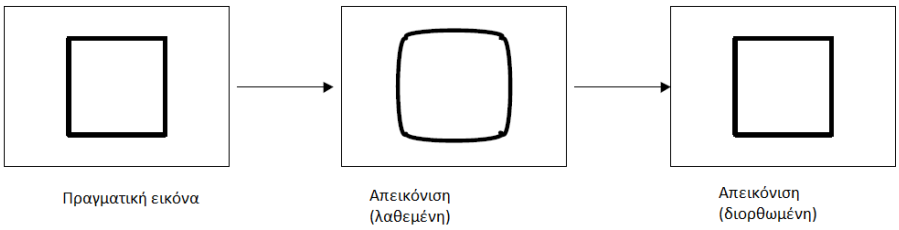


Εικόνα 50: Βασικές παραμορφώσεις (εδώ pin-cushion distortion – παραμόρφωση μαξιλαριού).



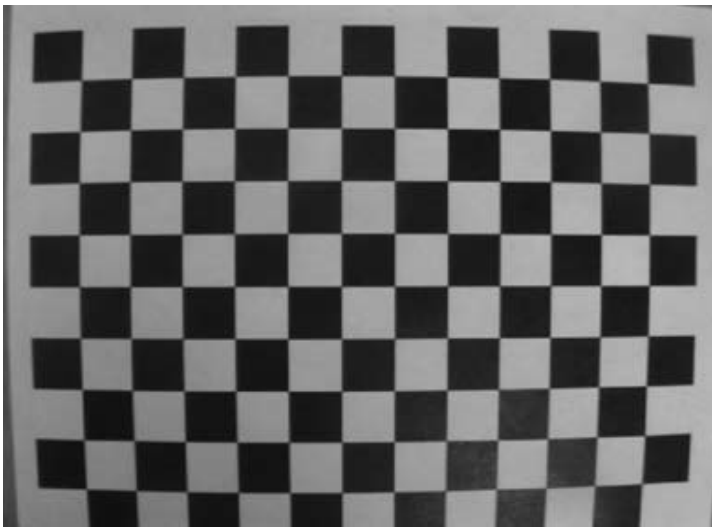
Εικόνα 51: Βασικές παραμορφώσεις (εδώ barrel distortion – παραμόρφωση βαρελιού). Τα ονόματα προφανώς προκύπτουν από το σχήμα της εκάστοτε παραμορφωμένης εικόνας.

Ουσιαστικά αυτό που κάνουμε με τη βαθμονόμηση είναι να βρίσκουμε ποιά εσωτερικά φυσικά μεγέθη της κάμερας επηρεάζουν την διαδικασία της απεικόνισης και μετά να τα αντιστρέφουμε. Η διαδικασία αυτή είναι μαθηματικά αρκετά πολύπλοκη και βασίζεται σε πίνακες και μετασχηματισμούς της γραμμικής άλγεβρας, όμως με την βοήθεια της opencv καθίσταται σχετικά απλή (εντολή cvCalibrateCamera2).



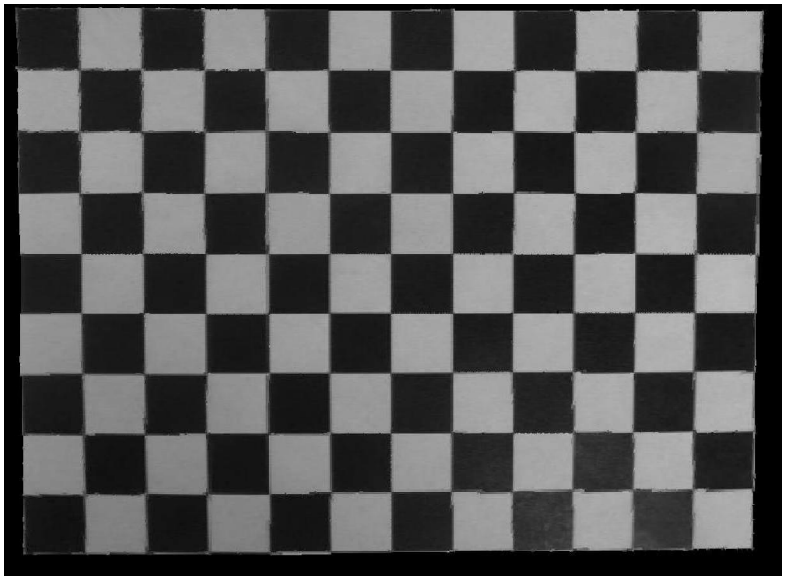
Εικόνα 52: Γραφική αναπαράσταση της βαθμονόμησης της κάμερας.

Η παραμόρφωση που είχαμε κυρίως να αντιμετωπίσουμε στο παρόν project ήταν η παραμόρφωση τύπου βαρελιού. Εάν κάποιος κοιτάξει το preview παράθυρο στην οθόνη του υπολογιστή θα δει την σκακιέρα παραμορφωμένη όπως ένα βαρέλι.



Εικόνα 53: Barrel distortion.

Το πρόβλημα αυτό ήταν αρκετά σημαντικό γιατί το ρομπότ θα έκανε λάθη στην αντιστοίχιση συντεταγμένων-αποστάσεων με pixel, με αποτέλεσμα να τοποθετεί τα πούλια σε λάθος κουτάκια είτε και να ψάχνει πούλια σε σημεία της σκακιέρας που δεν υπάρχουν. Με τη βοήθεια του κώδικα που ακολουθεί καταφέραμε και εξαλείψαμε το φαινόμενο αυτό φέρνοντας ουσιαστικά την εικόνα στη πραγματική της μορφολογία.



Εικόνα 54: Η παραπάνω σκακιέρα αφού έχουμε αφαιρέσει το barrel distortion.

4.2.2 Κώδικας

Στο cd περιέχεται ένα αρχείο κώδικα με όνομα calibrate_camera. Αυτός είναι ο κώδικας για το καλιμπράρισμα της κάμερας. Ο κώδικας αρκεί να τρέξει όταν τοποθετηθεί το ρομπότ στη βάση του. Εάν η τοπολογία της κατασκευής αλλάξει τότε είναι απαραίτητο να ξανατρέξουμε τον κώδικα.

Η βαθμονόμηση της κάμερας βασίζεται στη σκακιέρα. Αφού τοποθετήσουμε την σκακιέρα μπροστά από την κάμερα και η κάμερα εντοπίσει όλες τις γωνίες τότε πατάμε το πλήκτρο 'g'. Για να είναι επιτυχημένη η βαθμονόμηση δεν αρκεί μία εικόνα. Γι' αυτό μετακινούμε την σκακιέρα σε διάφορες θέσεις πατώντας κάθε φορά το πλήκτρο 'g' (εφόσον πάντα έχει εντοπιστεί η σκακιέρα από το πρόγραμμα – όταν δηλαδή δεν έχουν όλες οι γωνίες κόκκινο χρώμα). Η μεταβλητή n_boards καθορίζει πόσα στιγμιότυπα της σκακιέρας θα χρησιμοποιηθούν. Προφανώς όσο περισσότερα είναι τα στιγμιότυπα τόσο βελτιστοποιείται η απεικόνιση. Η μεταβλητή n_boards έχει επιλεγεί να είναι 4. Οι παράμετροι αποθηκεύονται σε δύο xml αρχεία τα οποία φορτώνονται στον βασικό κώδικα.

```
#include<cv.h>
#include<cxcore.h>
#include<cvaux.h>
#include<highgui.h>
#include <stdio.h>
#include <time.h>

#define DRIVER_ID  1

int main()
{
	int board_w = 7; // Πλάτος της σκακιέρας σε τετράγωνα.
	int board_h = 7; // Ύψος της σκακιέρας σε τετράγωνα.
	int n_boards = 4; // Αριθμός των στιγμιοτύπων. Όσο περισσότερα τόσο
το καλύτερο.
```

```
      int board_n = board_w * board_h;
      CvSize board_sz = cvSize( board_w, board_h );
      CvCapture* capture = cvCreateCameraCapture(DRIVER_ID);
      assert( capture );

      cvNamedWindow( "Calibration" );
      // Αποθήκευση
      CvMat* image_points           = cvCreateMat( n_boards*board_n,
2, CV_32FC1 );
      CvMat* object_points          = cvCreateMat( n_boards*board_n,
3, CV_32FC1 );
      CvMat* point_counts                 = cvCreateMat( n_boards, 1,
CV_32SC1 );
      CvMat* intrinsic_matrix       = cvCreateMat( 3, 3, CV_32FC1 );
      CvMat* distortion_coeffs= cvCreateMat( 5, 1, CV_32FC1 );

      CvPoint2D32f* corners = new CvPoint2D32f[ board_n ];
      int corner_count;
      int successes = 0;
      int step, frame = 0;

      IplImage *image = cvQueryFrame( capture );
      IplImage *gray_image = cvCreateImage( cvGetSize( image ), 8, 1
);

      // Λούπα για τα στιγμιότυπα έως να φτάσουμε τα n_boards
      // επιτυχημένα στιγμιότυπα (εφόσον πάντα έχουν βρεθεί όλες οι γωνίες
τις σκακιέρας).

      while( successes < n_boards ){
       //Βρες τις γωνίες της σκακιέρας
       int found = cvFindChessboardCorners( image, cvSize(7,7),
corners,
                        &corner_count, CV_CALIB_CB_ADAPTIVE_THRESH |
CV_CALIB_CB_FILTER_QUADS );

       // Βρες την ακρίβεια των γωνιών.
       cvCvtColor( image, gray_image, CV_BGR2GRAY );
       cvFindCornerSubPix( gray_image, corners, corner_count, cvSize(
11, 11 ),
                        cvSize( -1, -1 ), cvTermCriteria(
CV_TERMCRIT_EPS+CV_TERMCRIT_ITER, 30, 0.1 ));

       // Ζωγράφισε
       cvDrawChessboardCorners( image, board_sz, corners,
corner_count, found );
       cvShowImage( "Calibration", image );


            // Χειρισμός του pause/unpause και του ESC
            int c = cvWaitKey( 15 );
            // Με το πλήκτρο 'g' σώζουμε τα καλά στιγμιότυπα στους
πίνακες.
       if( c == 'g' ){
                  c = 0;
            // Εάν έχουμε καλή εικόνα, την ‘προσθέτουμε’ στα δεδομένα μας.
                   if( corner_count == board_n ){
                        step = successes*board_n;
```

```
                        for( int i=step, j=0; j < board_n; ++i, ++j ){
                              CV_MAT_ELEM( *image_points, float, i, 0
) = corners[j].x;
                              CV_MAT_ELEM( *image_points, float, i, 1
) = corners[j].y;
                              CV_MAT_ELEM( *object_points, float, i, 0
) = j/board_w;
                              CV_MAT_ELEM( *object_points, float, i, 1
) = j%board_w;
                              CV_MAT_ELEM( *object_points, float, i, 2
) = 0.0f;
                        }
                        CV_MAT_ELEM( *point_counts, int, successes, 0
) = board_n;
                        successes++;


                        printf ("Board: %d\n",successes);
                  }
    }
            if( c == 27 )
                  return 0;
            image = cvQueryFrame( capture ); // Πήγαινε στην επόμενη
εικόνα.
      } // Τέλος της λούπας


      // Κατανομή των πινάκων σύμφωνα με το πόσες σκακιέρες βρέθηκαν.
      CvMat* object_points2 = cvCreateMat( successes*board_n, 3,
CV_32FC1 );
      CvMat* image_points2 = cvCreateMat( successes*board_n, 2,
CV_32FC1 );
      CvMat* point_counts2 = cvCreateMat( successes, 1, CV_32SC1 );

      // Μεταφορά των σημείων στους πίνακες με το σωστό μέγεθος.
      for( int i = 0; i < successes*board_n; ++i ){
            CV_MAT_ELEM( *image_points2, float, i, 0) = CV_MAT_ELEM(
*image_points, float, i, 0 );
            CV_MAT_ELEM( *image_points2, float, i, 1) = CV_MAT_ELEM(
*image_points, float, i, 1 );
            CV_MAT_ELEM( *object_points2, float, i, 0) = CV_MAT_ELEM(
*object_points, float, i, 0 );
            CV_MAT_ELEM( *object_points2, float, i, 1) = CV_MAT_ELEM(
*object_points, float, i, 1 );
            CV_MAT_ELEM( *object_points2, float, i, 2) = CV_MAT_ELEM(
*object_points, float, i, 2 );
      }

      for( int i=0; i < successes; ++i ){
            CV_MAT_ELEM( *point_counts2, int, i, 0 ) = CV_MAT_ELEM(
*point_counts, int, i, 0 );
      }
      cvReleaseMat( &object_points );
      cvReleaseMat( &image_points );
      cvReleaseMat( &point_counts );

      // Σε αυτό το σημείο έχουμε όλες τις γωνίες που χρειαζόμαστε
```

```
	// Αρχικοποίησε τον πρώτο πίνακα έτσι ώστε τα δύο εστιακά μήκη να
	// έχουν λόγο ίσο με 1.0.

	CV_MAT_ELEM( *intrinsic_matrix, float, 0, 0 ) = 1.0;
	CV_MAT_ELEM( *intrinsic_matrix, float, 1, 1 ) = 1.0;

	// Βαθμονόμηση της κάμερας
	cvCalibrateCamera2( object_points2, image_points2,
point_counts2, cvGetSize( image ),
		intrinsic_matrix, distortion_coeffs, NULL, NULL,
CV_CALIB_FIX_ASPECT_RATIO );

	// Αποθήκευση του εγγενή πίνακα και του πίνακα παραμόρφωσης
	cvSave( "Intrinsics.xml", intrinsic_matrix );
	cvSave( "Distortion.xml", distortion_coeffs );


    // Τέλος του κώδικα βαθμονόμησης - Calibration
    // ΤΡΕΧΟΥΜΕ ΤΟΝ ΚΩΔΙΚΑ ΜΟΝΟ ΜΙΑ ΦΟΡΑ ΕΦΟΣΟΝ Η ΤΟΠΟΛΟΓΙΑ ΤΗΣ ΚΑΤΑΣΚΕΥΗΣ
ΠΑΡΑΜΕΝΕΙ ΣΤΑΘΕΡΗ
    // και μετά χρησιμοποιούμε τα αρχεία xml που δημιούργησε.
    //-------------------------------------------------------------------
    // ΠΑΡΑΔΕΙΓΜΑ πως να χρησιμοποιήσεις τους πίνακες Intrinsics και
Distortion

    // Παράδειγμα πως να φορτώσεις τους πίνακες
	CvMat *intrinsic = (CvMat*)cvLoad( "Intrinsics.xml" );
	CvMat *distortion = (CvMat*)cvLoad( "Distortion.xml" );

	// Φτιάξε τον χάρτη ‘αντίστροφης παραμόρφωσης’που θα
χρησιμοποιήσουμε σε όλα τα στιγμιότυπα.
	IplImage* mapx = cvCreateImage( cvGetSize( image ),
IPL_DEPTH_32F, 1 );
	IplImage* mapy = cvCreateImage( cvGetSize( image ),
IPL_DEPTH_32F, 1 );
	cvInitUndistortMap( intrinsic, distortion, mapx, mapy );

	// Τρέξε την κάμερα, που τώρα δείχνει την πραγματική μη
παραμορφωμένη εικόνα.
	cvNamedWindow( "Undistort" );

	while( image ){
		IplImage *t = cvCloneImage( image );
		cvShowImage( "Calibration", image ); // Δείξε την πρώτη
//εικόνα (αυτή με τη παραμόρφωση)
		cvRemap( t, image, mapx, mapy ); // Κάνε την αντίστροφη
//παραμόρφωση.
		cvReleaseImage( &t );
		cvShowImage( "Undistort", image ); // Δείξε τη διορθωμένη //
εικόνα.

		// Χειρισμός του pause/unpause και του esc
		int c = cvWaitKey( 15 );
		if( c == 'p' ){
			c = 0;
			while( c != 'p' && c != 27 ){
				c = cvWaitKey( 250 );
			}
```

```
            }
            if( c == 27 )
                  break;
            image = cvQueryFrame( capture );
      }



    // Τέλος του προγράμματος

    // Απελευθέρωση των εικόνων
    cvReleaseImage(&image);
    cvReleaseImage(&gray_image);
    // Απελευθέρωση της  απεικόνισης
    cvReleaseCapture(&capture);

    // Κατέστρεψε το παράθυρο
    cvDestroyWindow("Calibration");
    cvDestroyWindow("Undistort");

      return 0;
      }
```

Τα xml όπως προκύπτουν:

```
<?xml version="1.0"?>
-<opencv_storage> -<Distortion type_id="opencv-matrix">
<rows>5</rows> <cols>1</cols> <dt>f</dt> <data> -1.00715840
12.19090080 -7.84270559e-003 -1.37364154e-003 -
63.44853210</data></Distortion> </opencv_storage>

<?xml version="1.0"?>
-<opencv_storage> -<Intrinsics type_id="opencv-matrix">
<rows>3</rows> <cols>3</cols> <dt>f</dt> <data> 467.57662964 0.
156.73272705 0. 467.57662964 88.28765869 0. 0. 1.</data></Intrinsics>
</opencv_storage>
```

5. Η τοπολογία της κατασκευής

Αφού λοιπόν παρουσιάστηκε η προσομοίωση σε περιβάλλον Matlab, είναι θεμιτό να παρουσιατεί και την κατασκευή την κίνηση της οποίας θα προσομοιώσαμε και στην οποία θα ενσωματωθούν οι κώδικες της εργασίας αυτής μελλοντικά. Έτσι θα αναφερθούμε συνοπτικά στο μηχανικό και στο ηλεκτρονικό μέρος της κατασκευής προς χάριν πληρότητας και μόνο.

5.1 Το μηχανικό μέρος της κατασκευής

Το ρομπότ

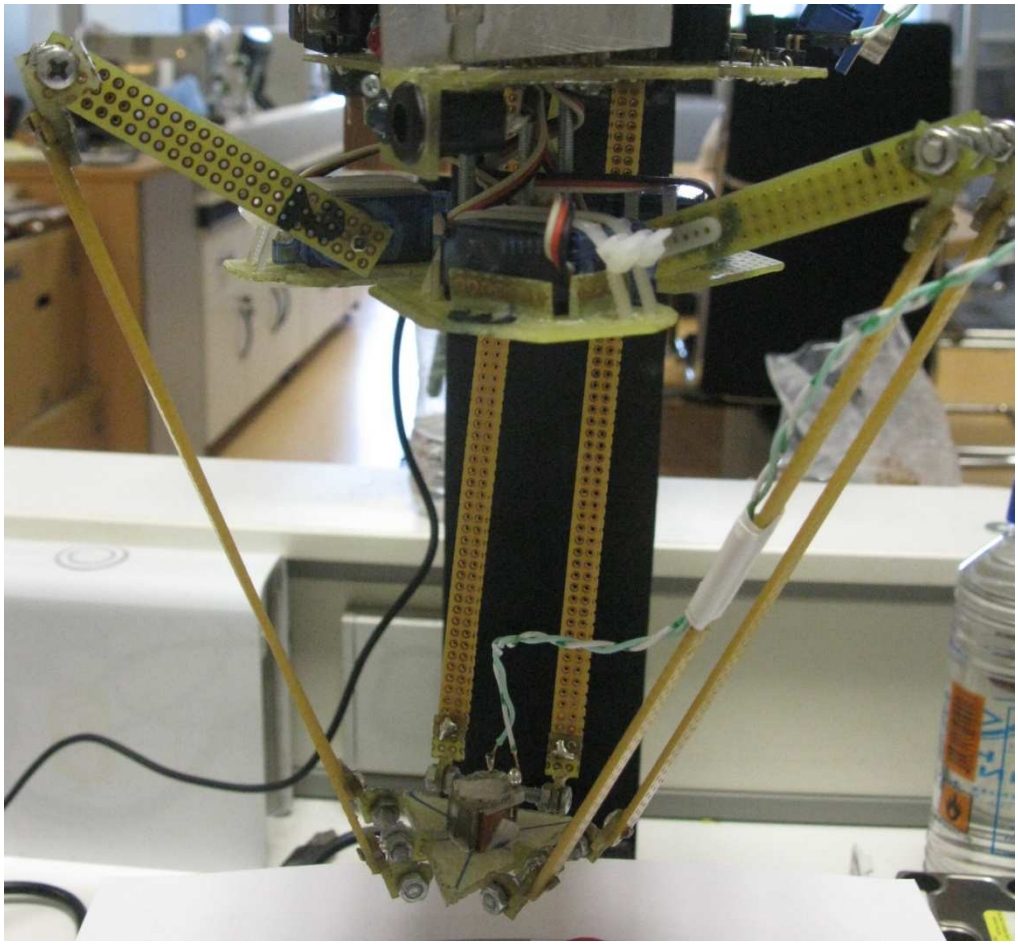
Το ρομπότ έχει κατασκευαστεί σχεδόν εξολοκλήρου από διάτρητες πλακέτες. Ο σχεδιασμός των διαστάσεων του ήταν απλός καθώς βασικός οδηγός αποτέλεσαν οι οπές των διάτρητων πλακετών, με αποτέλεσμα το ρομπότ να προκύψει σχεδόν στις επιθυμητές διαστάσεις με ελάχιστες αποκλίσεις. Δεν πρέπει να ξεχνάμε επίσης ότι ο σχεδιασμός δύο όμοιων τριγώνων στο χαρτί δεν παρουσιάζει ιδιαίτερη δυσκολία

Αυτά που χρειάστηκαν για την κατασκευή του πέρα από τις διάτρητες πλακέτες ήταν έξι βίδες των 3 mm με μήκος 30 mm, 30 παξιμάδια για τις βίδες αυτές, στιγμιαία κόλλα και φυσικά απαραίτητα εργαλεία (πέρα από μολύβι και χαρτί), όπως λίμα, πριόνι, κόπτη κτλ.

Με τα παραπάνω κατασκευάστηκαν οι βραχίονες του ρομπότ, το σημείο δράσης στο οποίο θα τοποθετηθεί μια ηλεκτρομαγνητική αρπάγη (ώστε να μετακινεί τα πούλια), η βάση του ρομπότ στην οποία τοποθετήθηκαν οι σερβοκινητήρες και η βάση της ηλεκτρονικής πλακέτας (που βρίσκεται πάνω ακριβώς από την βάση του ρομπότ).

Όπως αναφέρθηκε και στο θεωρητικό κομμάτι, το ρομπότ μας έχει τρείς βαθμούς ελευθερίας. Δηλαδή, τρεις μεταβλητές καθορίζουν τις συντεταγμένες του σημείου δράσης. Κάθε βαθμός ελευθερίας βρίσκεται και σε μία ενεργή άρθρωση. Ενεργές αρθρώσεις ονομάζονται ο αρθρώσεις στις οποίες συνδέονται οι σερβοκινητήρες. Μη ενεργές αρθρώσεις είναι οι υπόλοιπες αρθρώσεις του ρομπότ (στην ένωση του άνω βραχίονα με τον κάτω, και του κάτω βραχίονα με το σημείο δράσης), οι οποίες και είναι απαραίτητες ώστε να μη σπάσει το ρομπότ κατα τη λειτουργία του. Κάθε μη ενεργή άρθρωση έχει δύο βαθμούς ελευθερίας (πάνω – κάτω και δεξιά – αριστερά).

Ωστόσο υπάρχουν και μη ενεργές αρθρώσεις, που είναι άκρως απαραίτητες αφού διαφορετικά το ρομπότ θα σπάσει. Αυτές βρίσκονται στα σημεία ένωσης του άνω βραχίονα με τον κάτω βραχίονα και του κάτω βραχίονα με το κάτω τρίγωνο. Κάθε μη ενεργός άρθρωση πρέπει να έχει δύο βαθμούς ελευθερίας (πάνω-κάτω και δεξιά-αριστερά).



Εικόνα 55: Το ρομπότ πάνω στη βάση του.

Η βάση

Η βάση είναι κατασκευασμένη από ξύλο. Αρχική σκέψη ήταν το ρομπότ να μένει σταθερό στο χώρο, αλλα δεδομένων διαφόρων προβλημάτων που θα προέκυπταν αποφασίστηκε μόλις το ρομπότ παίξει να απομακρύνεται από τη σκακιέρα έτσι ώστε να έχει και καλύτερη οπτική επαφή η κάμερα με τη σκακιέρα (στην οποία αλλιώς θα εμπλέκονταν και οι βραχίονες του ρομπότ), και ο αντίπαλος παίχτης αρκετό χώρο για να κάνει την κίνηση του. Η βάση σχηματίζει ένα Γ, όπου το ρομπότ είναι στερεωμένο ανάποδα στην κορυφή και ακριβώς από κάτω στο επιθυμητό ύψος βρίσκεται η σκακιέρα με τα πούλια.

Εικόνα 56: Η βάση πάνω στην οποία προσαρμόστηκε το ρομπότ.

Η σκακιέρα και τα πούλια

Η σκακιέρα τυπωθηκε σε χαρτόνι έτσι ώστε να ελαχιστοποιηθούν οι ανακλάσεις του φωτός και να είναι όσο το δυνατόν πιο ξεκάθαρη η διαδοχή των χρωμάτων από το πρόγραμμα που θα 'βλέπει' και θα αναγνωρίζει τα πούλια. Αν προσέξει κανείς, θα δει επίσης πως η σκακιέρα μας είναι σε φόντο άσπρο - γκρι και όχι άσπρο - μαύρο, καθώς έτσι δευκολύνθηκε πολύ η αναγνώριση των χρωμάτων. Η σκακιέρα αυτή κολλήθηκε στην αντίστοιχη βάση.

Τα πούλια είναι από παιχνίδι της αγοράς και βάφτηκαν σε τέσσερα διαφορετικά χρώματα. Σε παιχνίδι μεταξύ ανθρώπων φυσικά τα χρώματα είναι δύο. Τα άλλα δύο χρώματα θα είναι οι αντίστοιχες ντάμες κάθε χρώματος. Έτσι λοιπόν η ομάδα του ρομπότ

είναι τα πούλια πλε-πράσινα και η αντίπαλη ομάδα (που παίζει ο άνθρωπος) είναι τα πούλια κόκκινα - κίτρινα. Σε κάθε πούλι έχει βιδωθεί μία βίδα, έτσι ώστε να έλκεται από τον ηλεκτρομαγνήτη όταν είναι επιθυμητό να μετακινηθεί.

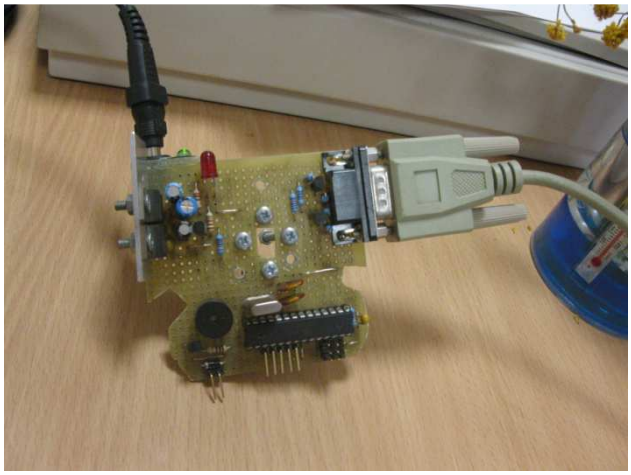
5.2 Το ηλεκτρονικό μέρος της κατασκευής

Η κάμερα

Η κάμερα είναι μια τυπική webcam που έχει ο καθένας σπίτι του χωρίς κάποια ιδιαίτερα χαρακτηριστικά. Αφού δοκιμάσαμε δύο ή τρείς διαφορετικές επιλέχτηκε αυτή που είχε καλύτερη εστίαση.

Η πλακέτα

Η πλακέτα δεν αποτέλεσε πεδίο μελέτης της παρούσης εργασίας. Υλοποιήθηκε με την βοήθεια διόδων τύπου 1N4004 και το τσιπ που χρησιμοποιήθηκε είναι το ATMEGA8.



Εικόνα 57: Η πλακέτα του ρομπότ.

Σερβοκινητήρες

Στο ρομπότ χρησιμοποιήθηκαν τρεις σερβοκινητήρες μικρού μεγέθους και ροπής 1,5Kgr/cm . Από τους σερβοκινητήρες δεν υπάρχουν ιδιαίτερες απαιτήσεις αφού το ρομπότ είναι μικρού μεγέθους και θα μεταφέρει μικρά και ελαφριά αντικείμενα όπως ένα πούλι. Αυτά δέθηκαν στη βάση με ειδικούς ιμάντες - κάθε σερβοκινητήρας σε μία γωνία της βάσης του ρομπότ.

Κάθε κινητήρας δέχεται από το chip έναν παλμό PWM τον οποίο τον μετατρέπει σε συγκεκριμένο άνοιγμα γωνίας. Έτσι για παράδειγμα με παλμό 1,5 ms ο κινητήρας έχει κλίση 0^o, για 1,75 ms 90^o και για 1,25 ms -90^o. Αυτή η σχέση παλμού – γωνίας είναι διαφορετική σε κάθε σέρβο και πρακτικά δίνεται από μια χαρακτηριστική ευθεία τύπου $y = ax + b$. Αυτή την χαρακτηριστική θα την μετρήσουμε (ουσιαστικά θα βρούμε τις δύο σταθερές a και b) και θα την ενσωματώσουμε στον κώδικα του τσιπ, όπως και το άνοιγμα των σερβοκινητήρων που δεν είναι σχεδόν ποτέ 180^o.

6. Οι μετρήσεις του ρομπότ

Αφού υλοποιήθηκε η κατασκευή του ρομπότ, ήταν απαραίτητο να μετρηθούν οι διαστάσεις του, ώστε ο προγραμματισμός του στο Matlab να είναι όσο το δυνατόν ρεαλιστικός και να μην επιφυλασσονται αρνητικές εκπλήξεις στη μελλοντική τελική υλοποίηση του ρομπότ, αλλά και οι χαρακτηριστικές ευθείες των σερβοκινητήρων.

6.1 Οι διαστάσεις του ρομπότ

Οι διαστάσεις του ρομπότ μετρήθηκαν δύο φορές. Την πρώτη χωρίς να έχουμε λύσει την κατασκευή και την δεύτερη με το ρομπότ 'λυμένο' (σαφώς πιο εύκολες μετρήσεις).

Στο πρώτο σετ μετρήσεων αναγκαστήκαμε να επιστρετεύσουμε αρκετές γεωμετρικές γνώσεις καθώς η απευθείας μέτρηση των διαστάσεων του ρομπότ δεν ήταν πάντα δυνατή. Πρόβλημα αντιμετωπίσαμε κυρίως στη βάση του ρομπότ και στο σημείο δράσης. Στη μεν βάση, ενώ είχε σχεδιαστεί σαν τρίγωνο, για λόγους ευκολίας το σχήμα μετατράπηκε σε κάτι αρκετά πιο πολύπλοκο και δεν ήταν εμφανείς οι πλευρές του τριγώνου που έπρεπε να μετρηθούν για την προσομοίωση, ενώ στο σημείο δράσης αν και το τρίγωνο είναι εμφανές ακόμα και τώρα, ήταν αδύνατο να μετρηθούν οι πλευρές του λόγω των αρθρώσεων και της ύπαρξης του ηλεκτρομαγνήτη. Όπως αναφέρθηκε και πριν ζητούμενο ήταν η μέγιστη ακρίβεια στις διαστάσεις. Στους βραχίονες δεν αντιμετωπίστηκαν ιδιαίτερα προβλήματα πέρα από το γεγονός ότι έπρεπε να προσέχουμε ώστε οι μη ενεργές αρθρώσεις να είναι παράλληλες με τους βραχίονες.

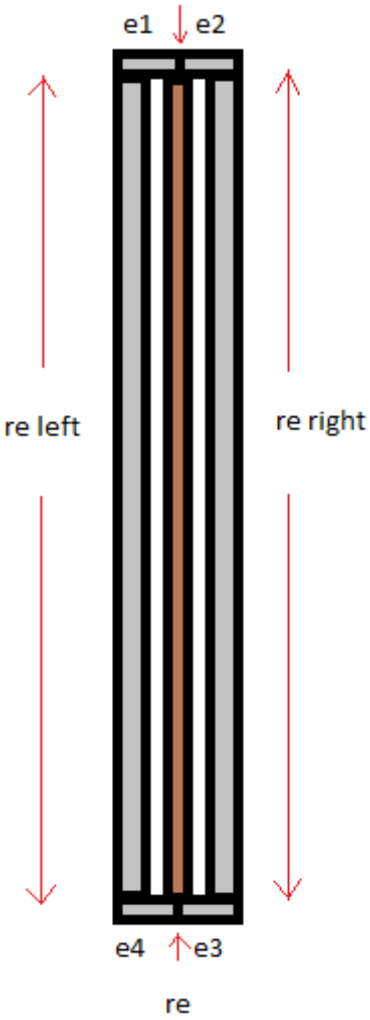
Το πρόβλημα του μεγάλου τριγώνου, αυτό της βάσης δηλαδή λύθηκε έυκολα μετρώντας το μήκος της μεσοκαθέτου. Δεδομένου ότι ο ηλεκτρομαγνήτης είναι στο κέντρο του ισόπλευρου τριγώνου (άρα στην τομή των μεσοκαθέτων και διαγωνίων του), και ότι κάθε γωνία του τριγώνου είναι ίση με 60^o, αφού δεν μπορούσαμε να μετρήσουμε απευθείας την πλευρά του τριγώνου αρκούσε να μετρήσουμε την απόσταση του κέντρου από την πλευρά. Έτσι, εάν x = πλευρά του τριγώνου, τότε x = απέναντι πλευρά/ημ(60^o). Ανακαλώντας έτσι γνώσεις γεωμετρίας (ισόπλευρα τρίγωνα και ιδιότητες ομοίων τριγώνων - δεν πρέπει να ξεχνάμε πως η βάση με το σημείο δράσης είναι όμοια τρίγωνα με λόγο 3/5) προέκυψαν εν τέλει οι μετρήσεις μας.

Στο δεύτερο σετ μετρήσεων όπου μπορούσαμε να έχουμε εύκολη πρόσβαση σε όλα τα μηχανικά μέρη του ρομπότ, τα πράγματα ήταν υπερβολικά πιο απλά. Το μόνο που χρειάστηκε πέρα από τον χάρακα, ήταν ένα μολύβι, κάποια φύλλα Α4 και πολύ προσοχή. Τα σχήματα της βάσης και του σημείου δράσης αποτυπώθηκαν στο χαρτί, όπου και μετρήθηκαν, ενώ με τους βραχίονες πάλι δεν αντιμετωπίσαμε κανένα πρόβλημα.

Σημαντικό είναι να αναφερθεί πως οι μετρήσεις γίναν ξεχωριστά για κάθε σέρβο - βραχίονα (είναι αριθμημένοι) γιατί ήταν φυσικό να υπάρχουν μικρές αποκλίσεις. Σαν 'επίσημες' διαστάσεις του ρομπότ θεωρήσαμε έναν ιδιότυπο μέσο όρο των μετρήσεων των τριών βραχιόνων, αν και μπορούμε να πούμε ότι οι αποκλίσεις ήταν σχεδόν ανύπαρκτες. Επίσης, ελάχιστες ήταν και οι αποκλίσεις μεταξύ του πρώτου και του δεύτερου σετ μετρήσεων, κατι που αποδεικνύει ότι οι γνώσεις μας στην ευκλείδια γεωμετρία

παραμένουν σωστές! Παρακάτω θα αναφερθούν οι μετρήσεις για κάθε σερβο – βραχίονα σχηματικά:

Βραχίονας – σέρβο 1:



e_1 =13.0 mm

e_2 = 13.0 mm

e_3 = 13.0 mm

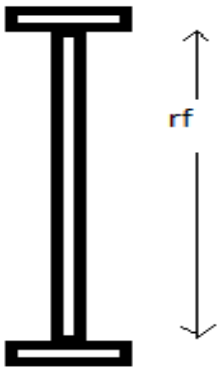
e_4 = 12.0 mm

r_e left = 167 mm

r_e right = 167.5 mm

r_e = 167 mm

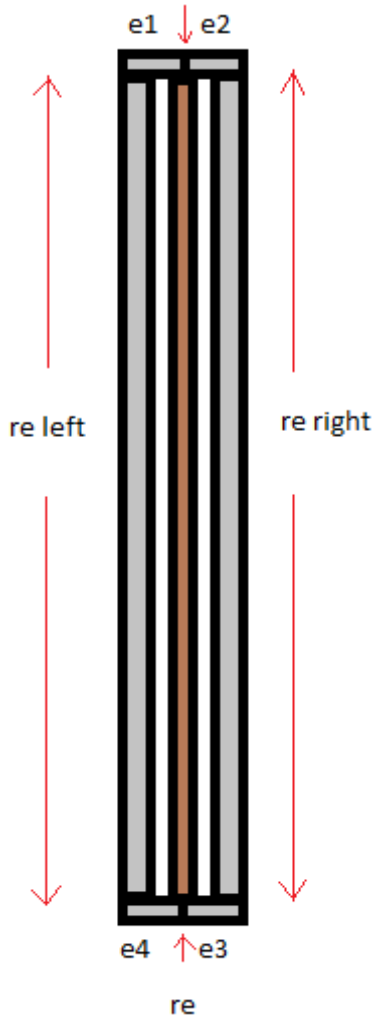
Εικόνα 58: Ο r_f βραχίονας που βρίσκεται στη θέση 1 του ρομπότ.



r_f = 56 mm

Εικόνα 59: r_e βραχίονας που βρίσκεται στη θέση 1 του ρομπότ.

Βραχίονας – σέρβο 2:



e_1 =13.0 mm

e_2 = 13.0 mm

e_3 = 12.5 mm

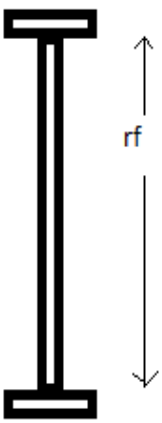
e_4 = 12.0 mm

r_e left = 166 mm

r_e right = 166 mm

r_e = 166 mm

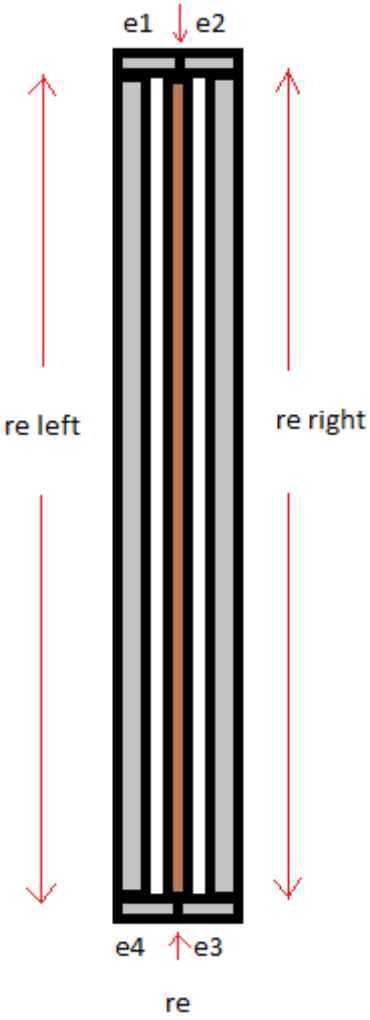
Εικόνα 60: Ο r_e βραχίονας που βρίσκεται στη θέση 2 του ρομπότ.



r_f = 56 mm

Εικόνα 61: r_e βραχίονας που βρίσκεται στη θέση 2 του ρομπότ.

Βραχίονας – σέρβο 3:



e_1 =13.0 mm

e_2 = 13.0 mm

e_3 = 13.0 mm

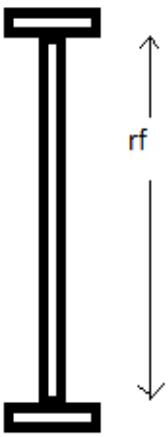
e_4 = 12.0 mm

r_e left = 166 mm

r_e right = 167 mm

r_e = 167 mm

Εικόνα 62: Ο r_e βραχίονας που βρίσκεται στη θέση 3 του ρομπότ.



r_f = 56 mm

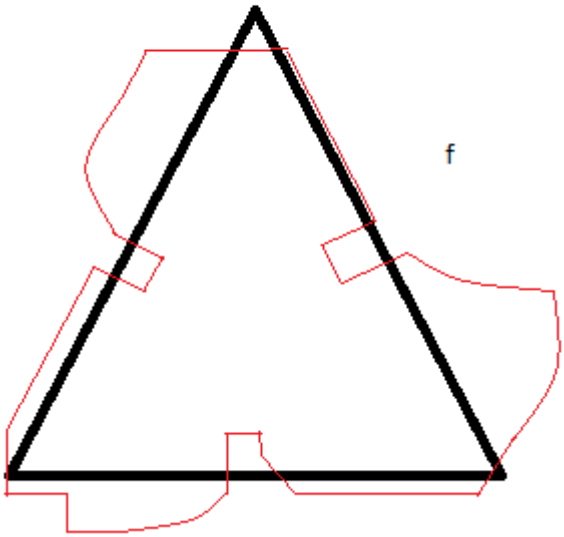
Εν τέλει θεωρούμε πως:

r_e = 167 mm

r_f = 56 mm

Εικόνα 63: Ο r_e βραχίονας που βρίσκεται στη θέση 3 του ρομπότ.

Η Βάση και το Σημείο Δράσης του ρομπότ:



f1 = 102 mm

f2 = 102 mm

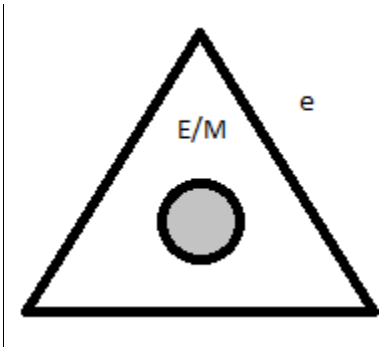
f3 = 100 mm ,

οπότε θεωρήσαμε τη βάση ισόπλευρο τρίγωνο με

f = 100 mm

Με το κόκκινο περίγραμμα είναι η πραγματική απεικόνιση της βάσης του ρομπότ, ενώ με το μαυρό το θεωρούμενο ισόπλευρο τρίγωνο.

Εικόνα 64: Κάτοψη της βάσης του ρομπότ (η κόκκινη γραμμή) και του θεωρούμενου ισόπλευρου τριγώνου.



Εικόνα 65: Κάτοψη του σημείου δράσης του ρομπότ.

e1 = 60 mm

e2 = 61 mm

e3 = 62 mm,

οπότε θεωρήσαμε το σημείο δράσης ισόπλευρο τρίγωνο με

e = 60 mm

Το E/M στο σχήμα συμβολίζει τον ηλεκτρομαγνήτη – αρπάγη.

6.2 Μετρήσεις στους Σερβοκινητήρες

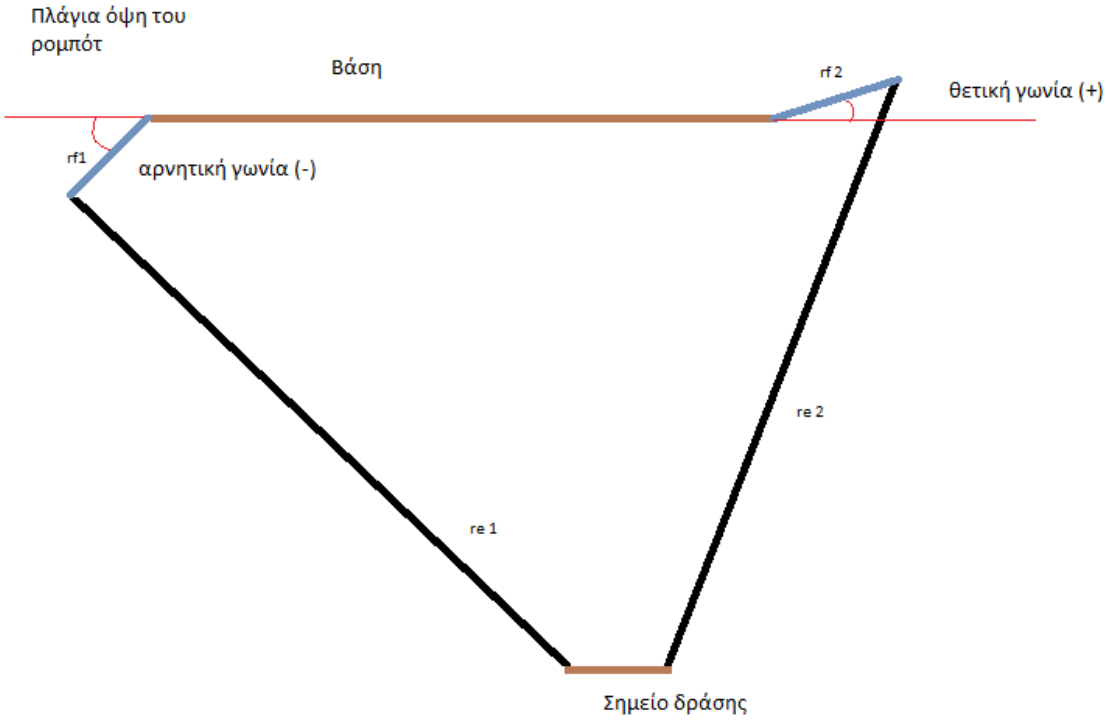
Μετρήσεις όμως έγιναν και στους σερβοκινητήρες του ρομπότ. Η πρώτη ομάδα μετρήσεων είχε να κάνει με το άνοιγμα γωνίας του κάθε κινητήρα (βασικός περιορισμός για το work plane του ρομπότ) και η δεύτερη ομάδα μετρήσεων είχε να κάνει με την χαρακτηριστική του κάθε σερβοκινητήρα.

6.2.1 Α' ομάδα μετρήσεων – άνοιγμα γωνίας:

Όπως αναφέρθηκε και πιο πάνω οι σερβοκινητήρες έχουν θεωρητικά άνοιγμα 180^{o}. Αυτό βέβαια στην πραγματικότητα είναι αρκετά μικρότερο αφού το συνολικό άνοιγμα της γωνίας προσεγγίζει περίπου τις 160^{o}. Προφανώς η διαφορά των 20^{o} περιορίζει αρκετά το

πεδίο δράσης του ρομπότ. Γι’ αυτόν ακριβώς τον λόγο έπρεπε να μετρηθεί το μέγιστο άνοιγμα του κάθε κινητήρα ξεχωριστά ώστε στην προσομοίωση να έχουμε μια ρεαλιστική απεικόνιση των σημείων στα οποία το ρομπότ έχει πραγματικά πρόσβαση χωρίς προβλήματα.

Για την μέτρηση χρησιμοποιήσαμε απλά ένα μοιρογνωμόνιο, ένα μολύβι, κάποιες κόλλες Α4 και ως ευθεία αναφοράς την πρόσοψη του γραφείου στο οποίο σχεδιάζαμε. Για κάθε σερβοκινητήρα μετρήσαμε μία θετική και μία αρνητική γωνία. Πιο παραστατικά εξηγούνται όλα στην παρακάτω εικόνα:



Εικόνα 66: Πλάγια όψη του ρομπότ όπου φαίνονται η θετική και αρνητική γωνία που μετράμε. Με καφέ χρώμα είναι η βάση του ρομπότ και το σημείο δράσης, με μπλε το r_f και με μαύρο το r_e, ενώ η κόκκινη γραμμή είναι η θεωρούμενη προέκταση της βάσης.

Ακολουθεί ο πίνακας με τις μετρήσεις:

Πίνακας μέγιστης και ελάχιστης γωνίας:

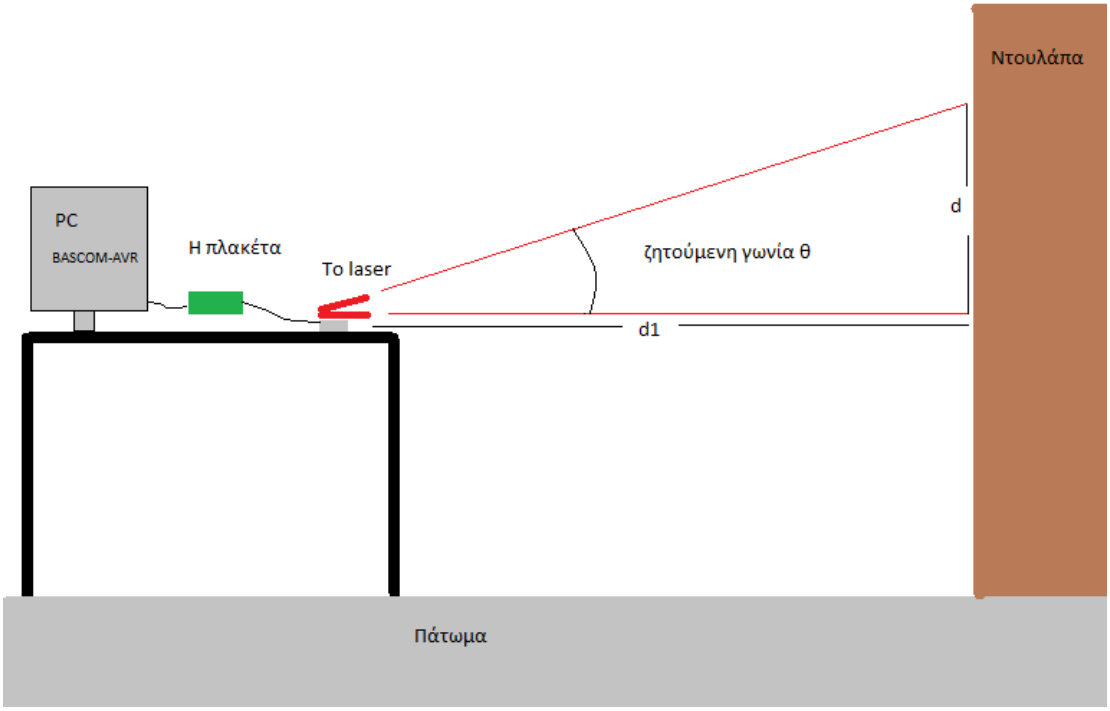
	Σερβοκινητήρας 1	**Σερβοκινητήρας 2**	**Σερβοκινητήρας 3**
γωνία (+) (μοίρες)	62	62	62
γωνία (-) (μοίρες)	106	108	114
Άθροισμα (μοίρες)	168	170	176

Είναι εμφανές ότι οι γωνίες των σέρβο δεν διαφέρουν ιδιαίτερα. Φυσικά τα όρια που θα πάρουμε στην προσομοίωση θα είναι πιο μικρά ώστε να έχουμε επιπλέον ασφάλεια για το ρομπότ εκτός και αν διαπιστώσουμε πως είναι ανέφικτο για το ρομπότ να καλύψει ολόκληρη τη σκακιέρα, οπότε και θα έχουμε την άνεση να τα μεγαλώσουμε λίγο. Λογικά όμως κάτι τέτοιο δεν θα χρειαστεί.

6.2.2 Β' ομάδα μετρήσεων – μέτρηση της χαρακτηριστικής ευθείας:

Αυτές οι μετρήσεις ήταν αρκετά πιο πολύπλοκες από τις προηγούμενες. Βασικός λόγος ήταν η δυσκολία λόγω της φύσης τους (μέτρηση γωνίας σερβοκινητήρα), αλλα και η ζητούμενη ακρίβεια καθώς οποιαδήποτε ανακρίβεια στις μετρήσεις θα οδηγούσε σε λάθος χαρακτηριστική ευθεία και πιθανώς μετά σε μία αλλοπρόσαλλη συμπεριφορά του ρομπότ, αφού ακόμα και αν το αντίστροφο κινηματικό πρόβλημα λυνόταν σωστά, το ρομπότ θα 'μετέφραζε' λαθος τους παλμούς του chip. Γι΄αυτόν ακριβώς το λόγο οι μετρήσεις επαναλήφθηκαν αρκετές φορές για κάθε κινητήρα και επεξεργάστηκαν στατιστικά για το βέλτιστο αποτέλεσμα. Χαρακτηριστικά για κάθε παλμό που δίναμε στον κινητήρα, παίρναμε τουλάχιστον 5 μετρήσεις και στο τέλος 'κατοχυρώναμε' σε κάθε παλμό τον μέσο όρο των 5 μετρήσεων.

Η αλήθεια είναι οτι υπήρχε μεγάλος προβληματισμός στην αρχή για το πως θα καταφέρουμε να πάρουμε ακριβείς μετρήσεις σχετικά εύκολα. Εν τέλει η λύση δόθηκε από ένα laser. Το σκεπτικό ήταν το εξής: θα στερεώναμε το laser πάνω στο αντίστοιχο σέρβο κάθε φορά, και αφού το συνδέαμε με τη σειριακή θύρα του pc, θα του δίναμε διαφορετικούς παλμούς μέσω του BASCOM και οι μετρήσεις θα γίνονταν με τη βοήθεια της προβολής του laser σε μία ντουλάπα, η οποία είχε ενσωματωμένη πάνω της μία μεζούρα. Προφανώς μετρήσαμε την απόσταση του laser από την ντουλάπα και μετά το μόνο που έμενε ήταν να μετράμε κάθε φορά την απόσταση της προβολής από το επίπεδο του laser όταν ήταν οριζόντιο (στις 0^o μοίρες δηλαδή). Έτσι διαιρώντας την κάθετη απόσταση με την οριζόντια, θα γνωρίζαμε την εφαπτομένη της γωνίας, άρα και τη γωνία. Παρατίθεται η παρακάτω εικόνα για ευκολότερη κατανόηση:



Εικόνα 67: Έτσι ακριβώς πραγματοποιήθηκαν οι μετρήσεις. Οπότε βρίσκαμε θ = εφ^{-1}(d/d1).

Στον υπολογιστή τρέχαμε τον εξής κώδικα κάθε φορά αλλάζοντας τον παλμό:

```
' {$STAMP BS2}
' {$PBASIC 2.5}

i VAR Byte
main:


FOR i=1 TO 100
  PULSOUT 12, 900
  PAUSE 15
NEXT

PAUSE 100
GOTO main

END
```

Εδώ πρέπει να αναφερθεί ότι ο παλμός που γράφουμε στο πρόγραμμα δεν είναι ο παλμός που δίνουμε στο chip αλλα ο υποδιπλάσιος λόγω της συχνότητας του τσιπ.

Θα παρατεθούν αναλυτικά οι μετρήσεις για κάθε σέρβοκινητήρα καθώς και οι αντίστοιχες γραφικές παραστάσεις που προέκυψαν. Παρατηρήθηκε ότι ανεβάζοντας τους παλμούς η σταθερά b της χαρακτηριστικής y = ax + b ήταν διαφορετική από όταν τους κατεβάζαμε. Γι' αυτόν το λόγο θα παρατεθούν δύο μετρήσεις για κάθε κινητήρα (b' και b'')

και δυο χαρακτηριστικές ευθείες. Επειδή η διαφορά της σταθεράς δεν ήταν μεγάλη επιλέξαμε να βάλουμε έναν μέσο όρο τον δύο b στον κώδικα του chip.

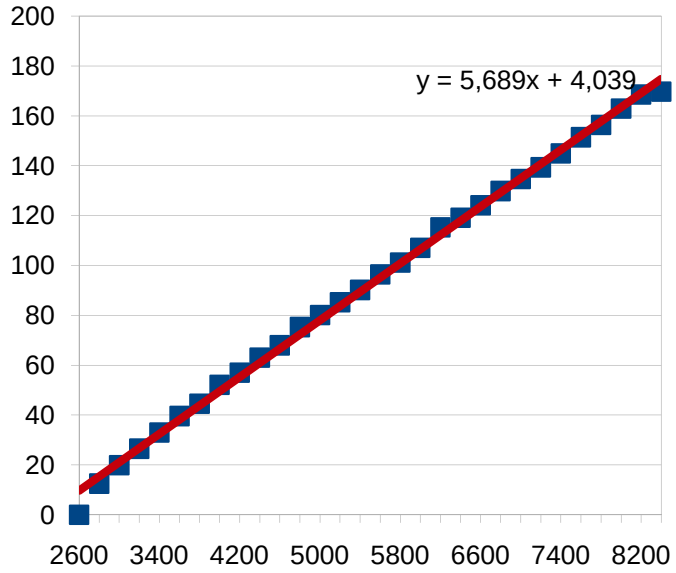
Ακολουθεί η επεξεργασία των μετρήσεων και οι χαρακτηριστικές ευθείες που προέκυψαν στο Microsoft Excel:

6.2.2.1 Κινητήρας 1

- Οι μετρήσεις μεγαλώνοντας την γωνία:

Κινητήρας 1	***(Εύρος τιμών 2400 – 8400)***		
Pulse Duration (us)	**PWM**	**Ψ (cm)**	**Γωνία (°)**
(περιοχή 1)			
600	**2400**	0	0
650	**2600**	16,2	12,4790718
700	**2800**	26,5	19,90136326
750	**3000**	36,5	26,50239863
800	**3200**	47,5	32,97982565
850	**3400**	60,5	39,57382456
900	**3600**	72	44,52649194
950	**3800**	94	52,09127326
1000	**4000**	112,6	56,97255548
1050	**4200**	143,7	63,0059535
(περιοχή 2)			
1050	**4200**	0	0
1100	**4400**	6,3	4,919072383
1150	**4600**	15,9	12,25502457
1200	**4800**	22,5	17,08622673
1250	**5000**	29,9	22,21853907
1300	**5200**	37,5	27,12584025
1350	**5400**	48,2	33,36370269
1400	**5600**	57,5	38,15031224
1450	**5800**	70,8	44,04515875
1500	**6000**	94,4	52,20915106
1550	**6200**	108,8	56,06756599
(περιοχή 3)			
1550	**6200**	0	0
1600	**6400**	6,4	4,996760665
1650	**6600**	13,9	10,75192331
1700	**6800**	20,4	15,5725436
1750	**7000**	27	20,24662008
1800	**7200**	35,4	25,80871518
1850	**7400**	46,3	32,31391214
1900	**7600**	55,6	37,21895378
1950	**7800**	70,2	43,80151835
2000	**8000**	85,4	49,39870535
2050	**8200**	89	50,56366326

Και η αντίστοιχη χαρακτηριστική ευθεία:

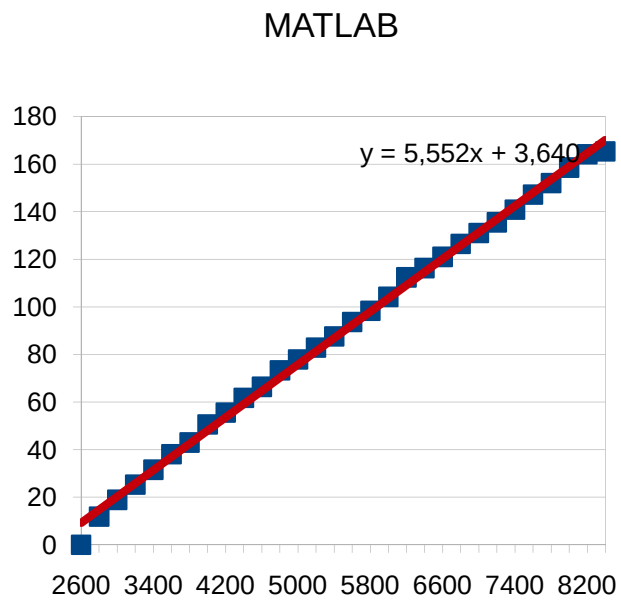


- Οι μετρήσεις μικραίνοντας την γωνία:

Κινητήρας 1	***(Εύρος τιμών 2400 – 8400)***		
Pulse Duration (us)	**PWM**	**Ψ (cm)**	**Theta (deg) *simple calculation***
(περιοχή 1)			
600	**2400**	0	0
650	**2600**	16,2	12,4790718
700	**2800**	26,5	19,90136326
750	**3000**	36,5	26,50239863
800	**3200**	47,5	32,97982565
850	**3400**	60,5	39,57382456
900	**3600**	72	44,52649194
950	**3800**	94	52,09127326
1000	**4000**	112,6	56,97255548
1050	**4200**	143,7	63,0059535
(περιοχή 2)			
1050	**4200**	0	0
1100	**4400**	6,3	67,92502588
1150	**4600**	15,9	75,26097807
1200	**4800**	22,5	80,09218023
1250	**5000**	29,9	85,22449257
1300	**5200**	37,5	90,13179375
1350	**5400**	48,2	96,36965619
1400	**5600**	57,5	101,1562657
1450	**5800**	70,8	107,0511122
1500	**6000**	94,4	115,2151046

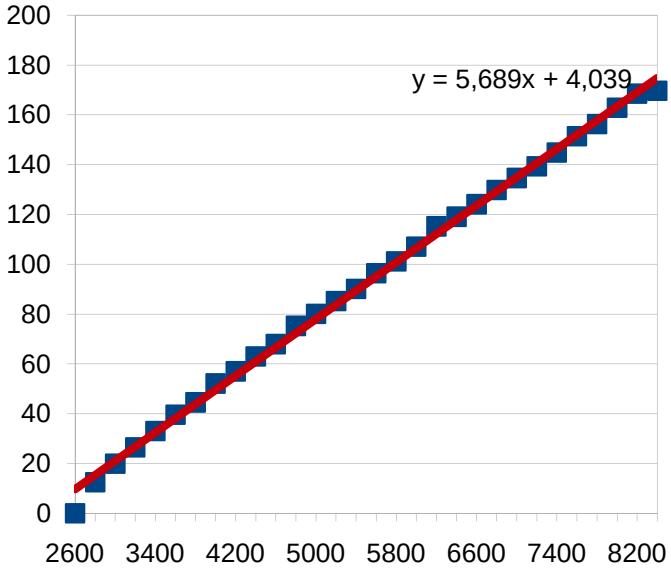
1550	**6200**	108,8	119,0735195
(περιοχή 3)			
1550	**6200**	0	0
1600	**6400**	124,0702801	124,0702801
1650	**6600**	129,8254428	129,8254428
1700	**6800**	134,6460631	134,6460631
1750	**7000**	139,3201396	139,3201396
1800	**7200**	144,8822347	144,8822347
1850	**7400**	151,3874316	151,3874316
1900	**7600**	156,2924733	156,2924733
1950	**7800**	162,8750378	162,8750378
2000	**8000**	168,4722248	168,4722248
2050	**8200**	169,6371827	169,6371827

Και η αντίστοιχη χαρακτηριστική ευθεία:



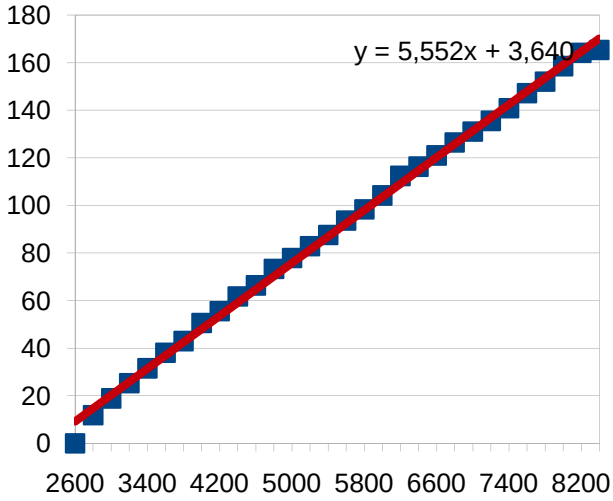
6.2.2.2 Κινητήρας 2

Οι μετρήσεις στον κινητήρα δύο πάρθηκαν με παρόμοιο τρόπο με τους άλλους δύο κινητήρες (τον 1 και τον 3), και φυσικά οι χαρακτηριστικές προέκυψαν παρόμοιες:



Και μικραίνοντας τη γωνία:

MATLAB



Οι μετρήσεις του κινητήρα 2 δεν αναφέρονται, καθώς δεν πάρθηκαν το ίδιο λεπτομερώς με τους άλλους δύο κινητήρες. Εξάλλου η προηγούμενη εμπειρία (ο κινητήρας 2 μετρήθηκε τελευταίος) μας καθοδηγούσε αποτελεσματικά ώστε να προκύψει η τελική ευθεία με λιγότερο κόπο χωρίς αυτό να είναι εις βάρος της ακρίβειας.

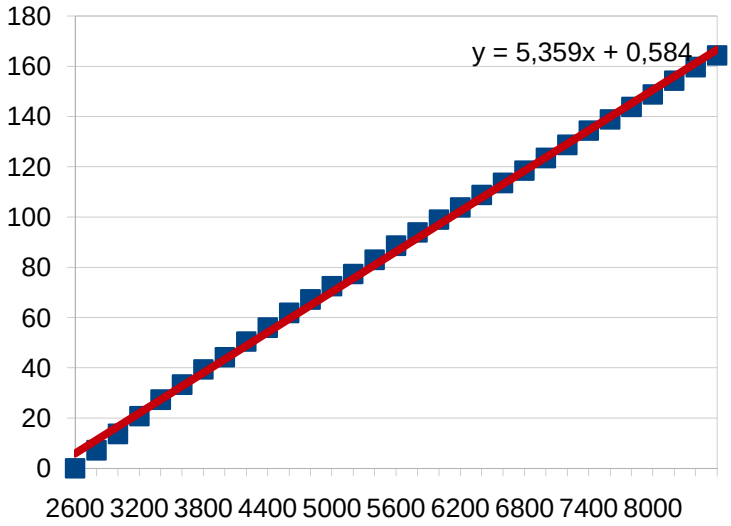
6.2.2.3 Κινητήρας 3

- Οι μετρήσεις μεγαλώνοντας την γωνία:

Κινητήρας 3	***(Εύρος τιμών 2400 – 8400)***		
Pulse Duration (us)	**PWM**	**Ψ (cm)**	**theta (deg) *simple calculation***
(περιοχή 1)			
600	**2400**	0	0
650	**2600**	9,2	7,163547466
700	**2800**	17,8	13,66731245
750	**3000**	27,7	20,72740938
800	**3200**	37,8	27,31153375
850	**3400**	48,1	33,30906891
900	**3600**	60,1	39,38730461
950	**3800**	71,2	44,20648039
1000	**4000**	88,5	50,40521511
1050	**4200**	108,5	55,99425644
(περιοχή 2)			
1050	**4200**	0	0
1100	**4400**	7,5	5,850055206
1150	**4600**	14,5	11,20452643
1200	**4800**	21,7	16,51235614
1250	**5000**	28,6	21,34110358
1300	**5200**	37,3	27,00170081
1350	**5400**	46,8	32,59258147
1400	**5600**	57	37,90748267
1450	**5800**	68,2	42,97482613
1500	**6000**	80,7	47,78998474
1550	**6200**	96,5	52,81789264
(περιοχή 3)			
1550	**6200**	0	0
1600	**6400**	6,1	4,763641691
1650	**6600**	12,5	9,690641573
1700	**6800**	19,2	14,69731928
1750	**7000**	26,5	19,90136326
1800	**7200**	35	25,5544256
1850	**7400**	42,3	30,02227947
1900	**7600**	51,3	35,02352736
1950	**7800**	61,3	39,94387363
2000	**8000**	74,3	45,42728264
2050	**8200**	90	50,87739261
2100	**8400**	106,5	55,4983273

Και η αντίστοιχη χαρακτηριστική ευθεία:

SIMPLE MODEL

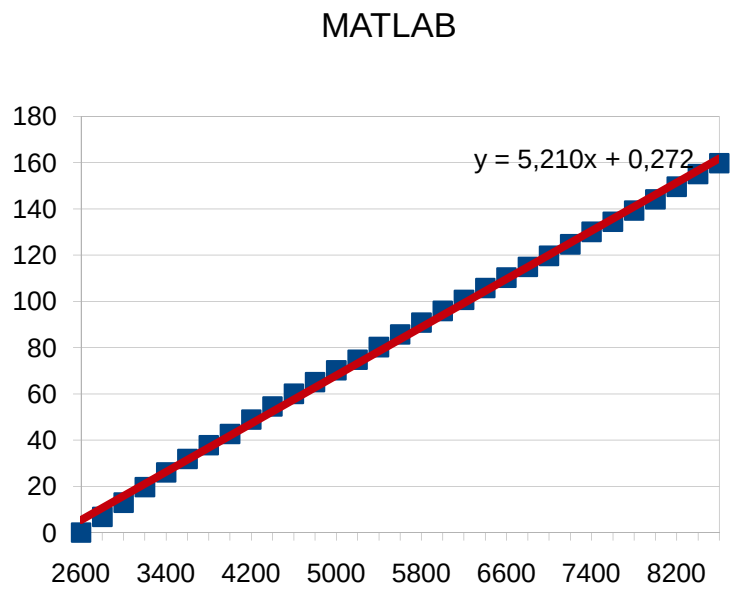


- Οι μετρήσεις μικραίνοντας την γωνία:

Κινητήρας 3	***(Εύρος τιμών 2400 – 8400)***		
Pulse Duration (us)	**PWM**	**Ψ (cm)**	**theta (deg) *simple calculation***
(περιοχή 1)			
600	**2400**	0	0
650	**2600**	9,2	7,163547466
700	**2800**	17,8	13,66731245
750	**3000**	27,7	20,72740938
800	**3200**	37,8	27,31153375
850	**3400**	48,1	33,30906891
900	**3600**	60,1	39,38730461
950	**3800**	71,2	44,20648039
1000	**4000**	88,5	50,40521511
1050	**4200**	108,5	55,99425644
(περιοχή 2)			
1050	**4200**	0	55,99425644
1100	**4400**	7,5	61,84431165
1150	**4600**	14,5	67,19878287
1200	**4800**	21,7	72,50661258
1250	**5000**	28,6	77,33536002
1300	**5200**	37,3	82,99595726
1350	**5400**	46,8	88,58683791
1400	**5600**	57	93,90173911
1450	**5800**	68,2	98,96908258
1500	**6000**	80,7	103,7842412

1550	**6200**	96,5	108,8121491
(περιοχή 3)			
1550	**6200**	0	0
1600	**6400**	6,1	113,5757908
1650	**6600**	12,5	118,5027907
1700	**6800**	19,2	123,5094684
1750	**7000**	26,5	128,7135123
1800	**7200**	35	134,3665747
1850	**7400**	42,3	138,8344286
1900	**7600**	51,3	143,8356764
1950	**7800**	61,3	148,7560227
2000	**8000**	74,3	154,2394317
2050	**8200**	90	159,6895417
2100	**8400**	106,5	164,3104764

Και η αντίστοιχη χαρακτηριστική ευθεία:



Εν τέλει, μελλοντικά με την ολοκλήρωση της κατασκευής θα επιλεχθούν οι πλέον κατάλληλες χαρακτηριστικες (οι οποίες θα πρέπει να δοκιμαστούν και στην πράξη), και έπειτα θα ενσωματωθούν στον αντίστοιχο κώδικα. Κατά τις μετρήσεις αυτές παρατηρήθηκε ένα σημαντικό πρόβλημα στη συμπεριφορά των συγκεκριμένων σερβοκινητήρων, οι οποίοι δεν επανέρχονταν στη θέση τους όπως θα έπρεπε. Με λίγα λόγια, στέλνοντας τον ίδιο παλμό το σέρβο είχε σημαντική απόκλιση στη γωνία που ‘έστριβε’!

6.3 Συνοψίζοντας...

Στην προσομοίωση του Matlab χρησιμοποιήθηκαν οι αρχικά υπολογισθέντες τιμές (δηλαδή οι διαστάσεις που το ρομπότ σχεδιάστηκε να έχει) καθώς είναι σαφως ρεαλιστικότερες. Παρ' όλα αυτά είναι προφανές ότι στις περισσότερες μετρήσεις η απόκλιση ήταν μικρή. Εξαίρεση αποτελούν τα δυό τρίγωνα (η βάση και το σημείο δράσης). Προφανώς στις αποκλίνοντες μετρήσεις τους, καθοριστικό ρόλο έπαιξε το ανθρώπινο λάθος. Έτσι λοιπόν στο αρχείο demo.m έχουμε τις εξής διαστάσεις:

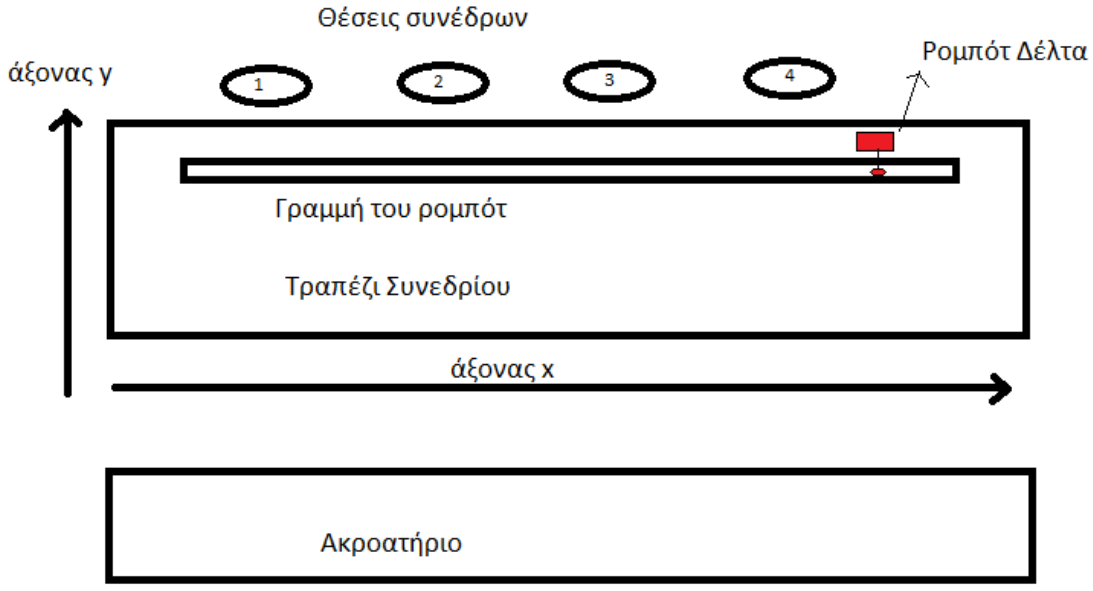
```
% f = 94 mm
% e = 57.3 mm
% rf = 56 mm
% re = 167 mm (measured between 166 - 167)
% umax = 60 degrees
% umin = -110 degrees
```

7. Χρησιμότητα και Εξέλιξη

7.1 Χρησιμότητα

Προφανώς απώτερος στόχος της προσομοίωσης της κίνησης, καθώς και όλων των παραπάνω προγραμμάτων, είναι η ενσωμάτωση τους μελλοντικά σε κάποιο ρομπότ. Η χρησιμότητα ενός τέτοιου ρομπότ θα ήταν καταρχάς ψυχαγωγική! Τι πιο απλό από το να έχεις έναν αντίπαλο με διάθεση για παιχνίδι κάθε μέρα, όλη μέρα; Εάν προσέξει κάποιος βέβαια λίγο καλύτερα θα παρατηρήσει πως δεν παύουν να υπάρχουν πλευρές της κατασκευής που κοιτάζουν προς πιο 'πρακτικές' υλοποιήσεις (με τις απαραίτητες τροποποιήσεις φυσικά). Ένα ρομπότ με μία αρπάγη θα μπορούσε να χρησιμοποιηθεί για οποιαδήποτε εργασία είναι απαραίτητο κάτι να μετακινηθεί (εξάλλου αυτός είναι και ο τομέας στον οποίο τα delta robot επιτυγχάνουν – γρήγορη και ακριβής συλλογή ή/και μετακίνηση αντικειμένων).

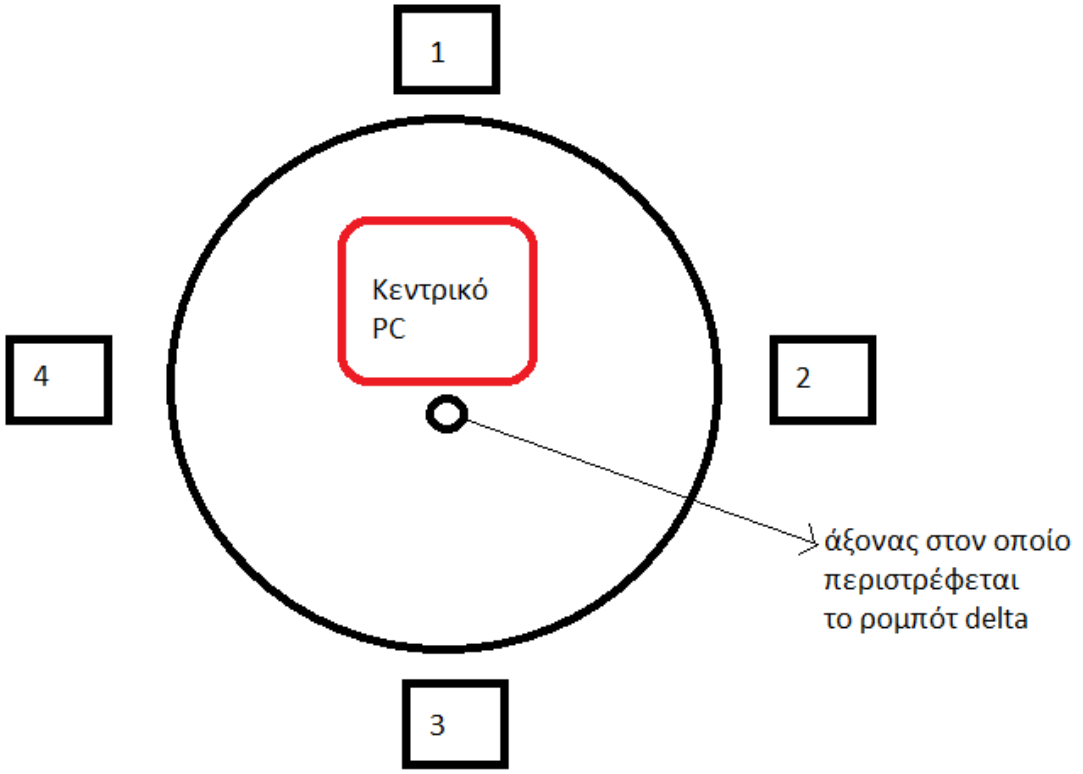
Έτσι για παράδειγμα το μελλοντικό ρομπότ αυτό θα μπορούσε να αποτελέσει μία βάση για την κατασκευή ενός ρομπότ πάλι τύπου delta, που σε ένα συνέδριο θα μεταφέρει στο τραπέζι των συνομιλητών κάποιο μικρόφωνο. Το ρομπότ θα είναι προσαρμοσμένο με ικανότητα μετακίνησης στον άξονα x με τη βοήθεια κάποιας ράβδας και θα πηγαινοέρχεται στο τραπέζι κρατώντας με την αρπάγη το μικρόφωνο.



Εικόνα 68: Παράδειγμα χρήσης του ρομπότ σε συνέδριακό κέντρο.

Μια έξυπνη υλοποίηση του θα μπορούσε ακόμα να γίνει σε κάποιο ραδιοφωνικό στούντιο με λίγους πόρους, 'ερασιτεχνικό' δηλαδή (φυσικά δεν αναφερόμαστε σε στούντιο που έχουν τη δυνατότητα να έχουν πλήθος εγκατεστημένων μικροφώνων και μεγάλους χώρους). Το ρομπότ εκεί αντίστοιχα θα μπορούσε να μεταφέρει το μικρόφωνο μεταξύ των προκαθορισμένων θέσεων των συνομιλητών που θα υπάρχουν, εξοικονομώντας έτσι χώρο και χρήμα. Θα μπορούσε ακόμα να έχει κάποιο αισθητήρα ανίχνευσης ήχου και αναλόγως

να παίρνει θέση πιο κοντά ή πιο μακριά απο τον εκάστοτε ομιλητή (αυτό φυσικά θα μπορούσε να γίνει και στην παραπάνω εφαρμογή του ‘συνεδρίου’)!



Εικόνα 69: Τυπικό παράδειγμα της χρήσης του ρομπότ delta σε ένα απλό ραδιοφωνικό στούντιο. Σε κάποιον ελεγκτή θα έχουν αποθηκευθεί 4 (ή και περισσότερες θέσεις) μεταξύ των οποίων θα περιστρέφεται το ρομπότ. Τα μικρόφωνα θα μπορούσαν να είναι και δύο ώστε να μη χάνεται η δυνατότητα του άμεσου διαλόγου αλλά να υπάρχει κ η προοπτική συμμετοχής περισσοτέρων ατόμων. Εάν τα μικρόφωνα είναι δύο θα έχει το καθένα το δικό του χώρο όπου θα κινείται.

Από κει και πέρα, εστιάζοντας περισσότερο στον κώδικα χρωματικής εντόπισης αντικειμένων, θα μπορούσε να φτιαχτεί ένα ρομπότ όπου με γνώμονα το χρώμα των αντικειμένων (ή και το σχήμα – δες προτάσεις εξέλιξης), θα τα περισυνέλεγε από κάποιον κινητό ιμάντα (για παράδειγμα σε κάποιο εργοστάσιο).

7.2 Επίλογος και Συμπεράσματα

Από την πολύμηνη ενασχόληση μας με τον κώδικα, προέκυψαν κάποια συμπεράσματα. Κρίνοντας από τον αρχικό μας στόχο, ο οποίος ήταν η κατασκευή ενός ρομπότ ώστε να παίζει σε πλήρως ρεαλιστικές συνθήκες ντάμα με αντίπαλο έναν άνθρωπο καταλήγουμε στα εξής:

(α.) Η αναγνώριση αντικειμένων βάση χρώματος με την OpenCV δεν ήταν τόσο αποδοτική όσο αναμενόταν. Πιθανή αιτία είναι η κακή ποιότητα της κάμερας που χρησιμοποιήθηκε. Αποτέλεσμα είναι η αναγνώριση των αντικειμένων να εξαρτάται σε μεγάλο βαθμό από τον φωτισμό γύρω από την κάμερα, το οποίο στερεί ευελιξία από την μελλοντική κατασκευή και απαιτεί ‘επαναπροσδιορισμό’ σταθερών του κώδικα σε κάθε μετακίνηση του ρομπότ.

(β.) Οι σερβοκινητήρες όπως παρατηρήθηκε στο 6^{ο} κεφάλαιο είχαν έλλειψη ενός βασικού στοιχείου: επαναληψιμότητα άρα στέρηση ακρίβειας στις κινήσεις του ρομπότ.

(γ.) Πιθανώς για ενδεχόμενη υλοποίηση στο μέλλον των παραπάνω (σε πλήρως ρεαλιστικές συνθήκες – με μετακινούμενη σκακιέρα δηλαδή), θα ήταν απαραίτητη η κατασκευή ενός σαφώς μεγαλύτερου ρομπότ καθώς το συγκεκριμένο έχει περιορισμένο εύρος εργασίας .

7.3 Βελτιστοποίηση και εξέλιξη του κώδικα και προτροπές για μελλοντική κατασκευή

Εν τέλει το ρομπότ δεν δοκιμάστηκε να παίζει ντάμα λόγω περιορισμένων τεχνικών μέσων (έλλειψη ακρίβειας από τα σέρβο όπως αναφέρθηκε παραπάνω) και περιορισμένου χρόνου για κατασκευή νέου. Προφανώς το ρομπότ (με όλα τα παρελκόμενα, κυρίως όμως τον visual κώδικα, αφού αυτός ήταν και το πιο μεγάλο μέρος της διπλωματικής εργασίας) χρήζει πολλών βελτιώσεων.

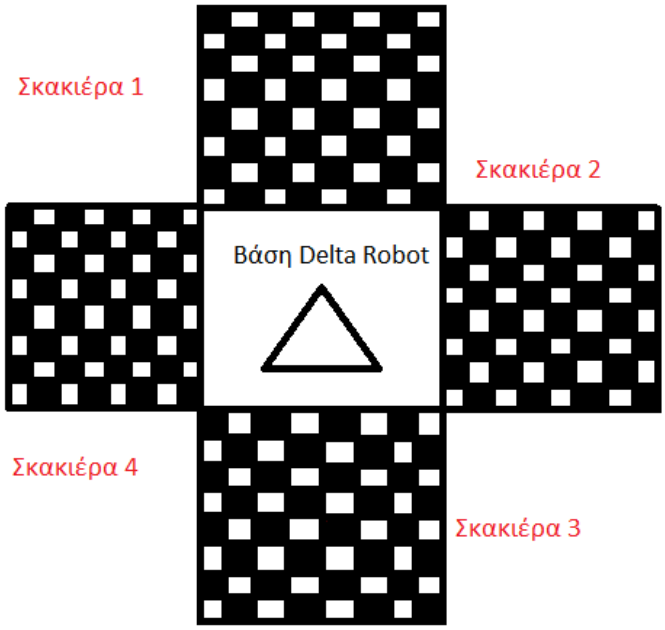
Συγκεκριμένα προβλήματα που υπήρχαν στη λειτουργία της κάμερας, ξεπεράστηκαν με τη χρήση πολλών φίλτρων. Ο κώδικας για αυτόν τον λόγο ίσως να μην είναι φοβερά αποδοτικός και αυτό πιθανώς να είναι το πρώτο πράγμα εκτενέστερης μελέτης σε μια ενδεχόμενη εξέλιξη του κώδικα. Συγκεκριμένες συναρτήσεις θα μπορούσαν να είναι πολύ πιο αποδοτικές και σύντομες (επιβαρύνοντας έτσι λιγότερο τον εκάστοτε χρησιμοποιούμενο υπολογιστή) με χαρακτηριστικότερο παράδειγμα την CheckManSquare. Στον visual κώδικα θα μπορούσε να χρησιμοποίηθει επίσης και η τεχνική ROI ώστε να υπάρχει λιγότερη κατανάλωση μνήμης. Άλλη πιθανή βελτίωση του κώδικα θα ήταν να γίνεται αναγνώριση προτύπου (pattern recognition) πριν από την αναγνώριση χρώματος, ώστε να πούλια να αναγνωρίζονται με βάση και το σχήμα τους (π.χ. ένας κύκλος για ένα πούλι) και έπειτα με βάση το χρώμα τους να διαχωρίζονται στις δύο ομάδες. Αυτό πιθανώς θα έχει ως αποτέλεσμα να μη χρειάζονται τα φίλτρα που χρησιμοποιήθηκαν στον κώδικα, απλοποιώντας τον σε μεγάλο βαθμό.

Δεν πρέπει να ξεχνάμε φυσικά πως το ρομπότ κινείται ουσιαστικά σε 37 προκαθορισμένες θέσεις (32 της σκακιέρας, 4 οι θέσεις της ντάμας και 1 θέση ισορροπίας). Αυτές οι θέσεις έχουν αποθηκευτεί σε πίνακα εξαρχής. Το ρομπότ θα μπορούσε να λειτουργεί χωρίς αυτόν τον πίνακα ενσωματώνοντας του απλά τον κώδικα της αντίστροφης κινηματικής. Έτσι για κάθε κίνηση το ρομπότ θα ήταν αναγκασμένο να λύνει από την αρχή το κινηματικό πρόβλημα. Πιθανώς η διαφορά που θα βλέπαμε στη συγκεκριμένη κατασκευή να μην ήταν τόσο εμφανής, αλλά το ρομπότ θα μπορούσε να λειτουργήσει σε ακόμα ρεαλιστικότερες συνθήκες όπου η σκακιέρα δεν θα ήταν σε προκαθορισμένη θέση και προσανατολισμό.

Επίσης άλλο ένα θέμα που πρέπει να αναφερθούμε είναι το calibration της κάμερας. Όπως αναφέρεται και πιο πάνω γίνεται εκτελώντας ένα αρχείο κώδικα πριν την τελική χρήση του ρομπότ. Εάν αλλάξει οποιαδήποτε τοπολογία στο σύστημα του ρομπότ (για παράδειγμα η βάση ή εάν μετακινηθεί η κάμερα), το calibration πρέπει να ξαναγίνει από την αρχή. Άρα μια έξυπνη εξέλιξη του ρομπότ θα ήταν να κάνει μόνο του calibration (auto-calibration) σκανάροντας σιγά σιγά την επιφάνεια της σκακιέρας.

Ακόμα, έχοντας τον κώδικα για τους κανόνες του παιχνιδού σε ξεχωριστό αρχείο που επικοινωνεί με τον visual κώδικα, έχουμε την δυνατότητα να φτιάξουμε ίσως ένα καινούριο δικό μας παιχνίδι (με πούλια)! Οι διαστάσεις της σκακιέρας καθώς και ο αριθμός των παιχτών κάθε ομάδας δεν θα πρέπει να μας απασχολεί ιδιαίτερα καθώς αυτοί οι παράμετροι μπορούν πολύ εύκολα να αλλάξουν (στην κορυφή του κώδικα). Εναλλακτικά, εάν δεν έχουμε όρεξη να επινοούμε παιχνίδια με καινούριους κανόνες, μπορούμε απλά να προσαρμόσουμε κάποιο άλλο παιχνίδι (για παράδειγμα κάποιο άλλο είδος ντάμας) ή ακόμα καλύτερα να υπάρχει και επιλογή για το τι θέλει ο αντίπαλος να παίξει με το ρομπότ. Αρκετά εντυπωσιακό θα ήταν η προσαρμογή του ρομπότ στο pc ώστε να μπορεί να παίξει με αντίπαλο μέσω του Διαδικτύου.

Εν τέλει, ίσως η πιο εντυπωσιακή αλλαγή που θα μπορούσε να γίνει στο παρόν project θα ήταν οι πολλαπλοί αντίπαλοι. Προσαρμόζοντας το ρομπότ σε μία καινούρια βάση που θα έχει ακτινικά γύρω της 4 (ή και παραπάνω εφόσον χωράνε) σκακιέρες και σε κάθε σκακιέρα το ρομπότ θα αντιμετωπίζει ταυτόχρονα έναν ξεχωριστό αντίπαλο, τον καθένα με την σειρά. Ακόμα πιο εντυπωσιακό θα γινόταν πιθανώς εάν το ρομπότ μεγάλωνε σε μέγεθος και γινόταν πιο ‘όμορφο’ εξωτερικά!



Εικόνα 70: Κάτοψη της τοπολογίας του Delta Robot με 4 αντιπάλους.

Αυτές είναι κάποιες από τις βελτιώσεις που γρήγορα μας ήρθαν στο μυαλό. Το σημαντικότερο όμως είναι πως επειδή η στενή και συνεχής ενασχόληση πιθανώς να περιορίζει την αντίληψη μας για την κατασκευή και το πώς αυτή θα μπορούσε να εξελιχτεί, κάθε πρωτότυπη και ρεαλιστική ιδέα φυσικά και είναι ευπρόσδεκτη.

Παράρτημα Α

Πρόγραμμα στο Matlab που βοήθησε στις μετρήσεις των χαρακτηριστικών των σερβοκινητήρων.

```
% This program is used to calculate the characteristics of the servos
%%% Dimitris Piperidis

clear all
clc

% Servo 1 measurments (cm)
servo1 = [94.7 97.8 99 104 106.5 110 112.5 114.7 117.9 120.9 124.5 129
134.8 133.5 143.7 154.7 170.6 192.5]';
% PWM pulse duration (us)
pulse1 = [1600 1610 1620 1630 1640 1650 1660 1670 1680 1690 1700 1720 1740
1760 1770 1800 1850 1900]';
% Servo 1 measurments (cm)
servo2 = [113 115 118 120.5 123.2 126 128 131.8 133.8 137 139 141.2 144.8
147.2 149.4 153 155.6 158.3 166.4 182.5 197.8]';
% PWM pulse duration (us)
pulse2 = [1600 1610 1620 1630 1640 1650 1660 1670 1680 1690 1700 1710 1720
1730 1740 1750 1760 1770 1800 1850 1900]';
% Servo 1 measurments
servo3 = [111.5 113.5 117 120 122.5 126 128.7 132.8 135.6 138.5 141.8 145
149 151.5 154.6 159.1 162.7 165.4 174.3 183 192.8]';
% PWM pulse duration (us)
pulse2 = [1600 1610 1620 1630 1640 1650 1660 1670 1680 1690 1700 1710 1720
1730 1740 1750 1760 1770 1800 1830 1850]';

% Distance of LASER from ground at zero position (cm)
h = 89.6;
% distance of LASER from measuring screen (cm)
d1 = 165.6;
% offset of the LASER from the rotation axes (cm)
l = 0.5;
% Distance of the LASER from the rotation axes (cm)
R = 4.5;
% combinte distance (axes form screen)
d = d1 + R;

% remove height and ofsset of LASER
servo1 = servo1 + l - h;
servo2 = servo2 + l - h;
servo3 = servo3 + l - h;


% Equations to solve
% (l^2*x^4 - 2*servo*l*x^3 + (servo^2 + d^2)*x^2- d^2 = 0
% cosö = x


% Calculate coefficiants
A1 = ones(length(servo1),1);
A1 = (l^2).*A1;
A2 = ones(length(servo2),1);
A2 = (l^2).*A2;
A3 = ones(length(servo3),1);
```

```
A3 = (l^2).*A3;

B1 = (-2*l).*servo1;
B2 = (-2*l).*servo2;
B3 = (-2*l).*servo3;

C1 = servo1.^2 + d^2;
C2 = servo2.^2 + d^2;
C3 = servo3.^2 + d^2;

D1 = ones(length(servo1),1);
D1 = (-d^2).*D1;
D2 = ones(length(servo2),1);
D2 = (-d^2).*D2;
D3 = ones(length(servo3),1);
D3 = (-d^2).*D3;

%zero array
O1 = zeros(length(servo1),1);
O2 = zeros(length(servo2),1);
O3 = zeros(length(servo3),1);

% calculate for first servo
func = [A1 B1 C1 O1 D1];
X=[];
for n=1:length(servo1)
   X1 = roots(func(n,:))
   for j=1:length(X1)
       if (X1(j)>0) && (X1(j)<2)
          X = [X;X1(j)];
       end
   end
end
F1 = acos(X);
F1 = F1.*180./pi;

% calculate for second servo
func = [A2 B2 C2 O2 D2];
X=[];
for n=1:length(A2)
   X1 = roots(func(n,:))
   for j=1:length(X1)
       if (X1(j)>0) && (X1(j)<2)
          X = [X;X1(j)];
       end
   end
end
F2 = acos(X);
F2 = F2.*180./pi;

% calculate for third servo
func = [A3 B3 C3 O3 D3];
X=[];
for n=1:length(A3)
   X1 = roots(func(n,:))
   for j=1:length(X1)
       if (X1(j)>0) && (X1(j)<2)
          X = [X;X1(j)];
       end
```

```
    end
end

F3 = acos(X);
F3 = F3.*180./pi;
```

Παράρτημα Β

Οι τρεις συναρτήσεις της κινηματικής του ρομπότ σε C:

- **Forward Kinematics**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define f  100
#define e  61
#define rf 56
#define re 167
#define tmax 100
#define tmin 40

// Compute Forward kinematics of a DELTA Robot

int DeltaForward(float theta1, float theta2, float theta3, float x0,
float y0, float z0, int ws) {

// trigonometric constants
const float sqrt3 = sqrt(3.0);
const float pi = 3.141592653;

float t,d ;// dnm, x0, y0, z0;

x0 = 0;
y0 = 0;
z0 = 0;

t = (f - e)/(2*sqrt(3));

// Convert degrees to rad

theta1 = theta1*pi/180;
theta2 = theta2*pi/180;
theta3 = theta3*pi/180;

// find coordinates of J'1

float y1 = -(t + rf*cos(theta1));
float z1 = rf*sin(theta1);

// find coordinates of J'2

float y2 = (t + rf*cos(theta2))/2;
float x2 = y2*sqrt(3);
float z2 = rf*sin(theta2);

// find coordinates of J'3

float y3 = (t + rf*cos(theta3))/2;
float x3 = -y3*sqrt(3);
```

```
float z3 = rf*sin(theta3);

float dnm = (y2-y1)*x3 - (y3-y1)*x2;
float w1 = y1*y1 + z1*z1;
float w2 = x2*x2 + y2*y2 + z2*z2;
float w3 = x3*x3 + y3*y3 + z3*z3;

float a1 = (z2-z1)*(y3-y1)-(z3-z1)*(y2-y1);
float b1 = -((w2-w1)*(y3-y1)-(w3-w1)*(y2-y1))/2;

float a2 = (z3-z1)*x2-(z2-z1)*x3;
float b2 = ((w2-w1)*x3 - (w3-w1)*x2)/2;

// a*z^2 + b*z + c = 0
float a = 4*(a1*a1 + a2*a2 + dnm*dnm);
float b = 4*(a1*b1 + a2*(b2-2*y1*dnm) - 2*z1*dnm*dnm);
float c = b1*b1 + (b2-2*y1*dnm)*(b2-2*y1*dnm) + 4*(z1*z1 -
re*re)*dnm*dnm;

// discriminant
d = b*b - 4*a*c;
if (d < 0){
    ws=0;
    }              // non-existing point
else{
  z0 = -(b+sqrt(d))/(2*a);
  // x = (2*a1*z + b1)/2dnm
  x0 = (2*a1*z0 + b1)/(2*dnm);
  // y = (2*a2*z + b2)/2dnm;
  y0 = (2*a2*z0 + b2)/(2*dnm);
  }

if ((theta1<tmax) && (theta2<tmax) && (theta2<tmax) && (theta1>tmin) &&
(theta2>tmin) && (theta2>tmin)){
    ws=1;
  }
  else{
    ws=0;
  }
}
```

- **Inverse Kinematics**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <GetAngle.h>

// Robot Geometry
#define f  100
#define e  61
#define rf 56
#define re 167
#define tmax 100
#define tmin 40
```

```
 int DeltaInverse (float x, float y, float z, float q1_1, float q2_1,
float q3_1, float q1_2, float q2_2, float q3_2, int ws){
 /*Compute inverse kinematics of a DELTA Robot
 [q1,q2] = deltaInverse (DeltaRobot,x,y,z)
 DeltaRobot = Delta Robot Geometry
 x,y,z = Given Cartesian coordinates (you can use vectors with same
dimension also)
 q1_1,q2_1,q3_1 = values corresponding joint angles in rad (first solution)
for motor 1,2 and 3
 q1_2,q2_2,q3_2 = 3 dimentional vector corresponding joint angles in rad
(second
 solution) for motor 1,2 and 3. For real robots this solution can be
ignored
 ws = workspace of the robot. 1 = movement allowed, 0 = movement not
 allowed. If x,y,z are arrays (equal size) then ws is array with zeros and
 ones. You can use this array to draw robot's workspace


float dr1, dr2, dr3;

// Motor1
GetAngle( x, y, z);
// Motor2
GetAngle(-x*0.5+y*sqrt(3)/2, -x*sqrt(3)/2-y*0.5, z);  //Rotate +120
// Motor3
GetAngle( -x*0.5-y*sqrt(3)/2, x*sqrt(3)/2-y*0.5, z);  //Rotate -120

// Calculate where dr is not valid and place zero
ws=dr1*dr2*dr3;  //r1.*dr2.*dr3
}
```

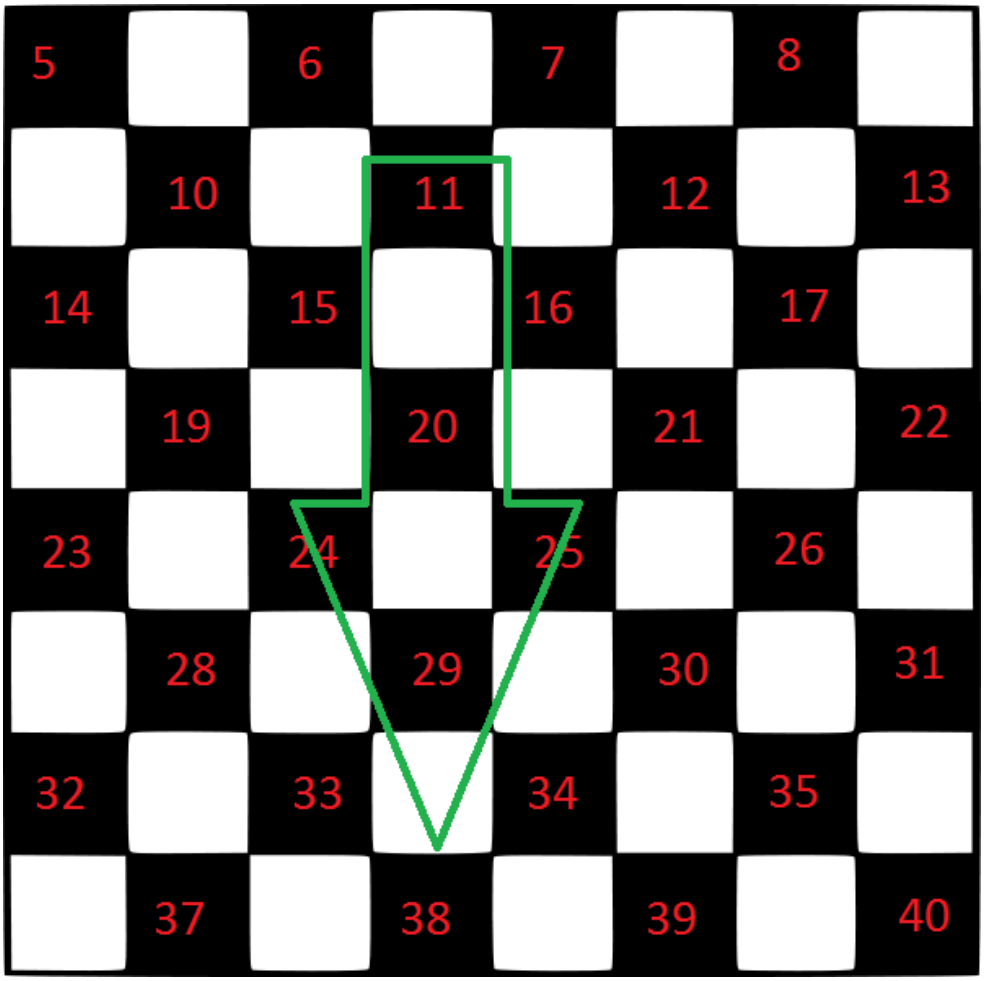
- **Get Angle**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Robot Geometry
#define f  100
#define e  61
#define rf 56
#define re 167
#define tmax 100
#define tmin 40

int GetAngle(float x, float y, float z, float theta1, float theta2, int
dmask) {
 [theta1, theta2, dmask] = GetAngle(DeltaRobot,x,y,z)
 DeltaRobot = Delta Robot Geometry
 X,Y,Z = Given Cartesian coordinates of the end effector
 dmask = This is the discriminant and angles (thetamax and thetamin) map.
 If dmask=0 then movement is not allowed
 If dmask=1 movement is allowed

// trigonometric constants
const float pi = 3.141592653;    // PI
```

```
float a, b, d, yj, zj, thetamask;

// Inverse kinematics equations

float y1 = -f/(2*sqrt(3));
float e0 = e/(2*sqrt(3));
float y0 = y - e0;

a = (x*x + y0*y0 + z*z + rf*rf - re*re - y1*y1)/(2*z); //x.^2 + y0.^2 +
z.^2 + rf.^2 - re.^2 - y1^2)./(2*z)
b = (y1 - y0)/z;  //y1 - y0)./z

d = -(a + b*y1)*(a + b*y1) + (rf*rf)*(1+b*b); //-(a + b.*y1).^2 +
(rf.^2).*(1+b.^2)
dmask = d>=0;
d = dmask*d; //dmask.*d

yj = (y1 - a*b - sqrt(d))/(1+b*b); //(y1 - a.*b - sqrt(d))./(1+b.^2)
zj = a + b*yj;  // a + b.*yj
theta1 = asin(zj/rf);

// theta1 = atan2(zj,y1-yj);

thetamask = theta1<=tmax & theta1>=tmin;
dmask=thetamask*dmask; //thetamask.*dmask

yj = (y1 - a*b + sqrt(d))/(1+b*b);  // (y1 - a.*b + sqrt(d))./(1+b.^2)
zj = a + b*yj;   //a + b.*yj
theta2 = pi-asin(zj/rf);
}
```

Παράρτημα Γ

Εναλλακτική δήλωση των συναρτήσεων DeltaToCheckers και CheckersToDelta, ώστε το παιχνίδι να ανταποκρίνεται σε διαφορετικού χρώματος τετράγωνα (δες καλύτερα στη δήλωση των συναρτήσεων στο κεφάλαιο 4):



Εικόνα 71: Η αρίθμηση του πίνακα και η φορά που παίζει το ρομπότ.

- **DeltaToCheckers**

```
int DeltaToCheckers (int
chessboard[CHESSBOARD_DIMENSION][CHESSBOARD_DIMENSION],
                     int **Board46){

    int i,j;
    int *temp46 = (int*)malloc(FORTYSIX*sizeof(int));
    *Board46 = (int*)malloc(FORTYSIX*sizeof(int));
    *Board46 = temp46;

    // 1st loop to convert the digits according to the checkers program.
```

```
    for (i=0;i<=CHESSBOARD_DIMENSION-1;i++){  // Running through all
squares.    Converting the Numbers of our program to the number the
checkers.cpp
        for (j=0;j<=CHESSBOARD_DIMENSION-1;j++){
            if(chessboard[i][j] == 2){
                        chessboard[i][j] = 5; // its like 1 for white
and 4 for man  check for 1|4
                        }
                    else if (chessboard[i][j] == 4){
                        chessboard[i][j] = 9; // its like 1 and 8 for
king (BITWISE OR)
                        }
                    else if (chessboard[i][j] == 3){
                        chessboard[i][j] = 10;  // 2 for black and 8 for
king (BITWISE OR)
                        }
                    else if (chessboard[i][j] == 1){
                        chessboard[i][j] = 6;  // 2 for black and 4 for
man (BITWISE OR)
                        }
            else if (chessboard[i][j] == 0){
                    chessboard[i][j] = 16;
                    }
             }
      }



    // The inital idea was something far more fancy but turned out non-
working! So its kind of a mess but its working!
    for (i=0;i<=CHESSBOARD_DIMENSION-1;i++){  // Running through all
squares.
        for (j=0;j<=CHESSBOARD_DIMENSION-1;j++){
            if (i == 0){
                if (j==0){
                    temp46[5] = chessboard[i][j];}
                else if (j == 2){
                    temp46[6] = chessboard[i][j];}
                else if (j == 4){
                    temp46[7] = chessboard[i][j];}
                else if (j == 6){
                    temp46[8] = chessboard[i][j];}
            }

            if (i == 1){
                if (j == 1){
                    temp46[10] = chessboard[i][j];}
                else if (j == 3){
                    temp46[11] = chessboard[i][j];}
                else if (j == 5){
                    temp46[12] = chessboard[i][j];}
                else if (j == 7){
                    temp46[13] = chessboard[i][j];}
            }

            if (i == 2){
                if (j == 0){
                    temp46[14] = chessboard[i][j];}
```

```
    else if (j == 2){
        temp46[15] = chessboard[i][j];}
    else if (j == 4){
        temp46[16] = chessboard[i][j];}
    else if (j == 6){
        temp46[17] = chessboard[i][j];}
}

if (i == 3){
    if (j == 1){
        temp46[19] = chessboard[i][j];}
    else if (j == 3){
        temp46[20] = chessboard[i][j];}
    else if (j == 5){
        temp46[21] = chessboard[i][j];}
    else if (j == 7){
        temp46[22] = chessboard[i][j];}
}

if (i == 4){
    if (j == 0){
        temp46[23] = chessboard[i][j];}
    else if (j == 2){
        temp46[24] = chessboard[i][j];}
    else if (j == 4){
        temp46[25] = chessboard[i][j];}
    else if (j == 6){
        temp46[26] = chessboard[i][j];}
}

if (i == 5){
    if (j == 1){
        temp46[28] = chessboard[i][j];}
    else if (j == 3){
        temp46[29] = chessboard[i][j];}
    else if (j == 5){
        temp46[30] = chessboard[i][j];}
    else if (j == 7){
        temp46[31] = chessboard[i][j];}
}

if (i == 6){
    if (j == 0){
        temp46[32] = chessboard[i][j];}
    else if (j == 2){
        temp46[33] = chessboard[i][j];}
    else if (j == 4){
        temp46[34] = chessboard[i][j];}
    else if (j == 6){
        temp46[35] = chessboard[i][j];}
}

if (i == 7){
    if (j==1){
        temp46[37] = chessboard[i][j];}
    else if (j == 3){
        temp46[38] = chessboard[i][j];}
```

```
                else if (j == 5){
                    temp46[39] = chessboard[i][j];}
                else if (j == 7){
                    temp46[40] = chessboard[i][j];}
            }

        }
    }
    // Putting zeros inside the board (where it is needed according to
checkers program).
    temp46[9] = temp46[18] = temp46[27] = temp46[36] = 0;

    for (i=0;i<5;i++){temp46[i]=0;}
    for (i=41;i<=46;i++){temp46[i]=0;}

return 0;
}
```

- **CheckersToDelta**

```
int CheckersToDelta (int temp46[FORTYSIX], int chessboard
[CHESSBOARD_DIMENSION][CHESSBOARD_DIMENSION]){

   int i;

    // 1st loop to convert the digits according to the checkers program
(red in ours is 2 and in checkers is 1 etc).

    for (i=FIVE;i<=FORTY;i++){  // Running through all squares.

        if(temp46[i] == 5){
            temp46[i] = 2;

        }
        else if (temp46[i] == 9){
            temp46[i] = 4;

        }
        else if (temp46[i] == 10){
            temp46[i] = 3;

        }
        else if (temp46[i] == 6){
            temp46[i] = 1;

        }
        else if (temp46[i] == 16){
                temp46[i] = 0;

        }
    }

    // The inital idea was something far more fancy but turned out non-
working! So its kind of a mess but its working!
    for (i=FIVE;i<=FORTY;i++){  // Running through all squares.
Converting the Numbers of our program to the number the checkers.cpp
        if (i == 37){
            chessboard[7][1] = temp46[i];
```

```
    chessboard[7][0] = 0;}
else if (i == 38){
    chessboard[7][3] = temp46[i];
    chessboard[7][2] = 0;}
else if (i == 39){
    chessboard[7][5] = temp46[i];
    chessboard[7][4] = 0;}
else if (i == 40){
    chessboard[7][7] = temp46[i];
    chessboard[7][6] = 0;}
else if (i == 9){
    /*! Do nothing */}
else if (i == 32){
    chessboard[6][0] = temp46[i];
    chessboard[6][1] = 0;}
else if (i == 33){
    chessboard[6][2] = temp46[i];
    chessboard[6][3] = 0;}
else if (i == 34){
    chessboard[6][4] = temp46[i];
    chessboard[6][5] = 0;}
else if (i == 35){
    chessboard[6][6] = temp46[i];
    chessboard[6][7] = 0;}
else if (i == 28){
    chessboard[5][1] = temp46[i];
    chessboard[5][0] = 0;}
else if (i == 29){
    chessboard[5][3] = temp46[i];
    chessboard[5][2] = 0;}
else if (i == 30){
    chessboard[5][5] = temp46[i];
    chessboard[5][4] = 0;}
else if (i == 31){
    chessboard[5][7] = temp46[i];
    chessboard[5][6] = 0;}
else if (i == 18){
    /*! Do nothing */}
else if (i == 23){
    chessboard[4][0] = temp46[i];
    chessboard[4][1] = 0;}
else if (i == 24){
    chessboard[4][2] = temp46[i];
    chessboard[4][3] = 0;}
else if (i == 25){
    chessboard[4][4] = temp46[i];
    chessboard[4][5] = 0;}
else if (i == 26){
    chessboard[4][6] = temp46[i];
    chessboard[4][7] = 0;}
else if (i == 19){
    chessboard[3][1] = temp46[i];
    chessboard[3][0] = 0;}
else if (i == 20){
    chessboard[3][3] = temp46[i];
    chessboard[3][2] = 0;}
else if (i == 21){
```

```
            chessboard[3][5] = temp46[i];
            chessboard[3][4] = 0;}
        else if (i == 22){
            chessboard[3][7] = temp46[i];
            chessboard[3][6] = 0;}
        else if (i == 27){
            /*! Do nothing */}
        else if (i == 14){
            chessboard[2][0] = temp46[i];
            chessboard[2][1] = 0;}
        else if (i == 15){
            chessboard[2][2] = temp46[i];
            chessboard[2][3] = 0;}
        else if (i == 16){
            chessboard[2][4] = temp46[i];
            chessboard[2][5] = 0;}
        else if (i == 17){
            chessboard[2][6] = temp46[i];
            chessboard[2][7] = 0;}
        else if (i == 10){
            chessboard[1][1] = temp46[i];
            chessboard[1][0] = 0;}
        else if (i == 11){
            chessboard[1][3] = temp46[i];
            chessboard[1][2] = 0;}
        else if (i == 12){
            chessboard[1][5] = temp46[i];
            chessboard[1][4] = 0;}
        else if (i == 13){
            chessboard[1][7] = temp46[i];
            chessboard[1][6] = 0;}
        else if (i == 36){
            /*! Do nothing */}
        else if (i == 5){
            chessboard[0][0] = temp46[i];
            chessboard[0][1] = 0;}
        else if (i == 6){
            chessboard[0][2] = temp46[i];
            chessboard[0][3] = 0;}
        else if (i == 7){
            chessboard[0][4] = temp46[i];
            chessboard[0][5] = 0;}
        else if (i == 8){
            chessboard[0][6] = temp46[i];
            chessboard[0][7] = 0;}
    }

    return 0;
}
```

- **Main for Debugging**

```
// Just some main function for debugging and for playing checkers!
// All we care about is function Letsplaythegame. Try playing Checkers
by changing the values on the boards!
int main(){


        // ChessBoard Array. No players until pc detects them so the
array is NULL.
    int chessboard[CHESSBOARD_DIMENSION][CHESSBOARD_DIMENSION]  = {
                          { 1,  0,  1,  0,  1,  0,  1,  0},
                          { 0,  1,  0,  1,  0,  1,  0,  1},
                          { 0,  0,  0,  0,  0,  0,  0,  0},
                          { 0,  0,  0,  0,  0,  0,  0,  0},
                          { 0,  0,  0,  0,  0,  0,  0,  0},
                          { 0,  2,  0,  0,  0,  0,  0,  0},
                          { 0,  0,  2,  0,  2,  0,  2,  0},
                          { 0,  2,  0,  2,  0,  2,  0,  2},
                        };

    // Second chessboard array. We need it for checking changes on
chessboard.
    int
previouschessboard[CHESSBOARD_DIMENSION][CHESSBOARD_DIMENSION]  =
{
                          { 1,  0,  1,  0,  1,  0,  1,  0},
                          { 0,  1,  0,  1,  0,  1,  0,  1},
                          { 0,  0,  0,  0,  0,  0,  0,  0},
                          { 0,  0,  0,  0,  0,  0,  0,  0},
                          { 0,  0,  0,  0,  0,  0,  0,  0},
                          { 0,  0,  0,  0,  0,  0,  0,  0},
                          { 2,  0,  2,  0,  2,  0,  2,  0},
                          { 0,  2,  0,  2,  0,  2,  0,  2},
                        };


    int blue_counter, green_counter, red_counter, orange_counter;

    LetsPlayTheGame (blue_counter, green_counter, red_counter,
orange_counter, previouschessboard, chessboard);

    return 0;
}



// For the other squares(inner squares from top left corner) try the 2
functions (DeltaToCheckers & CheckersToDelta) in Chapter 6 and try
changing the board instead of (1,0,1)->(0,1,0) etc.
```

Βιβλιογραφία – Αναφορές – Πηγές

Βιβλιογραφία

[1] Descriptive Geometric Kinematic Analysis of Clavel's "Delta" Robot - Prof. Paul Zsombor-Murray

[2] Τεχνητή Νοημοσύνη, μία σύγχρονη προσέγγιση – Stuart Russell, Peter Norvig

[3] Εισαγωγή στη Ρομποτική – Craig J. John

[4] Multiple View Geometry in Computer Vision, R.

[5] Hartley and A. Zisserman, Cambridge University Press, 2000, pp. 138-183

[6] Three-Dimensional Computer Vision: A Geometric Approach, O. Faugeras, MIT Press,1996, pp. 33-68

[7] "A Versatile Camera Calibration Technique for 3D Machine Vision", R. Y. Tsai, IEEE J. Robotics & Automation, RA-3, No. 4, August 1987, pp.323-344

[8] Rich JuszkiewiczEEN 538 – Digital Image Processing 10/25/05 Project 3 – Image Distortion Correction

[9] Τεχνητή Νοημοσύνη Β' έκδοση – Ι. Βλαχάβας, Π. Κεφαλάς, Ν. Βασιλειάδης, Φ. Κόκκορας, Η. Σακελλαρίου

[10] Learning OpenCV: Computer Vision with the OpenCV Library – Gary Bradski, Adrian Kaehler

[11] Εισαγωγή στο Matlab - Γ. Γεωργίου - Χ. Ξενοφώντος

Διευθύνσεις του Internet

[1] http://en.wikipedia.org/wiki/Malloc

[2] http://en.wikipedia.org/wiki/Distortion_%28optics%29

[3]http://el.wikipedia.org/wiki/%CE%9C%CE%B7%CF%87%CE%B1%CE%BD%CE%AE_%CE%A4%CE%BF%CF%8D%CF%81%CE%B9%CE%BD%CE%B3%CE%BA

[4] http://en.wikipedia.org/wiki/English_draughts

[5]http://en.wikipedia.org/wiki/Draughts

[6] http://en.wikipedia.org/wiki/Game_theory

[7] http://en.wikipedia.org/wiki/Libro_de_los_juegos

[8] http://en.wikipedia.org/wiki/Industrial_robot

[9] http://opencv.willowgarage.com/wiki/

[10] http://en.wikipedia.org/wiki/Delta_robot

[11] http://en.wikipedia.org/wiki/Parallel_robot

[12] http://en.wikipedia.org/wiki/Robot_kinematics

[13] http://wolfey.110mb.com/GameVisual/launch.php

[14] http://forums.trossenrobotics.com/tutorials/introduction-129/delta-robot-kinematics-3276/

[15] http://prime.jsc.nasa.gov/ROV/history.html

[16] http://www.parallemic.org/Reviews/Review002.html

[17] http://www.robots.com/robot-education.php?page=industrial+history

[18] http://www.parallemic.org/Reviews/Review002.html

[19] http://www.parallemic.org/Reviews/Review007.html

[20] http://www710.univ-lyon1.fr/~bouakaz/OpenCV-0.9.5/docs/ref/OpenCVRef_BasicFuncs.htm

[21] http://nashruddin.com/display-video-from-webcam-with-opencv.html

[22] http://www.cs.iit.edu/~agam/cs512/lect-notes/opencv-intro/opencv-intro.html

[23]http://opencv.willowgarage.com/documentation/cpp/motion_analysis_and_object_tracking.html

[24] http://opencv.willowgarage.com/wiki/CodeBlocks

[25]http://opensourcecollection.blogspot.com/2010/10/how-to-setup-codeblocks-for-opencv.html

[26] http://opencv.willowgarage.com/documentation/basic_structures.html

[27] http://www.fierz.ch/history.htm

[28] http://www.ocf.berkeley.edu/~yosenl/extras/alphabeta/alphabeta.html

[29] http://www.fierz.ch/checkers.htm

[30] http://en.wikipedia.org/wiki/ASCII

[31] http://en.wikipedia.org/wiki/Computer_vision

[32] http://en.wikipedia.org/wiki/Machine_vision

[33] http://en.wikipedia.org/wiki/OpenCV

[33] http://en.wikipedia.org/wiki/Azriel_Rosenfeld

[33] http://www.icg.tugraz.at/News/historyOfCV