ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΑ
ΤΜΗΜΑ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ
ΠΜΣ: ΨΗΦΙΑΚΕΣ ΕΠΙΚΟΙΝΩΝΙΕΣ ΚΑΙ ΔΙΚΤΥΑ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ανάπτυξη τρισδιάστατου διαδικτυακού παιχνιδιού με χρήση της Java OpenGL (JOGL)

Μαυρίκος Αλέξανδρος

Επιβλέποντες: Καθηγητής Ν. Σγούρος

Πειραιάς,
Μάρτιος 2011

Περιεχόμενα

Ευρετήριο εικόνων – σχημάτων

1. State of the art 3D παιχνιδιών

Ένα ηλεκτρονικό παιχνίδι είναι ένα παιχνίδι που χρησιμοποιεί ηλεκτρονικά στοιχεία για την δημιουργία ενός διαδραστικού συστήματος με το οποίο ένας παίκτης μπορεί να παίξει [5]. Η πιο κοινή μορφή των ηλεκτρονικών παιχνιδιών σήμερα είναι τα βιντεοπαιχνίδια, και για το λόγο αυτό οι δύο όροι αυτοί συχνά χρησιμοποιούνται εσφαλμένα ως συνώνυμα. Συγκεκριμένα ένα βιντεοπαιχνίδι είναι ένα ηλεκτρονικό παιχνίδι που περιλαμβάνει την αλληλεπίδραση με μία διαπροσωπεία χρήστη για τη δημιουργία οπτικής ανάδρασης σε μια συσκευή βίντεο. Η λέξη βίντεο στα βιντεοπαιχνίδια αναφέρεται σε μια συσκευή προβολής. Ωστόσο, με τη δημοφιλή χρήση του όρου "βιντεοπαιχνίδι", τώρα αναφέρεται σε οποιοδήποτε τύπο συσκευής προβολής. Τα ηλεκτρονικά συστήματα που χρησιμοποιούνται για να παίξουν βιντεοπαιχνίδια είναι γνωστά ως πλατφόρμες, παραδείγματα αυτών είναι οι προσωπικοί υπολογιστές και οι κονσόλες βιντεοπαιχνιδιών.

Η ιστορία των βιντεοπαιχνιδιών είναι γεμάτη με ανακαλύψεις και παλαιότερες τεχνολογίες που άνοιξαν το δρόμο για την έλευση των σύγχρονων βιντεοπαιχνιδιών. Περιλαμβάνει παιχνίδια που αντιπροσωπεύουν άμεσα στάδια στην εξέλιξη των ηλεκτρονικών παιχνιδιών και, τέλος, την ανάπτυξη και την κυκλοφορία των βιντεοπαιχνιδιών. Τα πρώτα βιντεοπαιχνίδια που υλοποιήθηκαν είναι τα εξής[5]:

● 1947: Cathode Ray Tube Amusement Device. Το πρώτο γνωστό διαδραστικό ηλεκτρονικό παιχνίδι ήταν από τους Thomas T. Goldsmith Jr και Estle Ray Mann με χρήση ενός καθοδικού σωλήνα ακτινών (CRT). Η ευρεσιτεχνία αυτή ανακαλύφθηκε στις 25 Ιανουαρίου του 1947 και εκδόθηκε στις 14 Δεκεμβρίου 1948. Το παιχνίδι ήταν ένας προσομοιωτής πυραύλων εμπνευσμένο από οθόνες ραντάρ από τον Δεύτερο Παγκόσμιο Πόλεμο. Χρησιμοποιήθηκαν αναλογικά στοιχεία κυκλώματος και όχι ψηφιακά, για τον έλεγχο της πορείας του CRT και την τοποθέτηση της θέσης με μια τελεία στην οθόνη.

● 1950-1951: Chess. Τον Μάρτιο του 1950, ο Claude Shannon επινόησε ένα πρόγραμμα που παίζει σκάκι και δημοσιεύτηκε στην εφημερίδα "Προγραμματισμός Η / Υ για τον υπολογιστή Ferranti.

● 1951: NIM. Στις 5 Μάη του 1951, ο υπολογιστής NIMROD, που δημιουργήθηκε από τον Ferranti, παρουσιάστηκε στο Φεστιβάλ της Μεγάλης Βρετανίας. Χρησιμοποιώντας έναν πίνακα φώτων για την οθόνη του, είχε σχεδιαστεί αποκλειστικά για να παίξει το παιχνίδι NIM. Αυτή ήταν η πρώτη εμφάνιση ενός ψηφιακού υπολογιστή που σχεδιάστηκε ειδικά για να παίξει ένα παιχνίδι. Ο NIMROD μπορούσε να παίξει είτε την παραδοσιακή ή την «αντίστροφη» μορφή του παιχνιδιού.

● 1952: OXO / Noughts and Crosses (Tic-Tac-Toe). Το 1952, ο Alexander S. Douglas έκανε το πρώτο παιχνίδι στον υπολογιστή που έκανε χρήση μιας ψηφιακής οθόνης γραφικών. Το OXO, επίσης γνωστό και ως Noughts and Crosses, είναι μια έκδοση του tic-tac-toe για τον υπολογιστή EDSAC στο πανεπιστήμιο του Cambridge.

● 1958: Tennis for Two. Το 1958, ο William Higinbotham έκανε ένα διαδραστικό ηλεκτρονικό παιχνίδι που ονομάστηκε Tennis for Two για την ετήσια ημέρα επισκεπτών του Brookhaven National

Laboratory. Αυτή η απεικόνιση, που χρηματοδοτήθηκε από το Υπουργείο Ενέργειας των ΗΠΑ, είχε ως στόχο την προώθηση της ατομικής ενέργειας, και χρησιμοποιούσε ένα αναλογικό υπολογιστή και το σύστημα απεικόνισης διανυσμάτων ενός παλμογράφου.

Ένα βιντεοπαιχνίδι, όπως άλλωστε και οι περισσότερες άλλες μορφές μέσων, μπορεί να ταξινομηθεί σε είδη βάση πολλών παραγόντων, όπως η μέθοδος του παιχνιδιού, τα είδη των στόχων, το στυλ του παιχνιδιού και άλλα πολλά. Επειδή τα είδη εξαρτώνται από το περιεχόμενο για τον ορισμό τους, τα είδη έχουν αλλάξει και έχουν εξελιχθεί καθώς νεώτερες μορφές βιντεοπαιχνιδιών αναπτύσσονται. Η ανάπτυξη της τεχνολογίας και της παραγωγής σε σχέση με την ανάπτυξη των βιντεοπαιχνιδιών οδήγησαν σε πολύπλοκα και ρεαλιστικά παιχνίδια που με τη σειρά της εισήγαγαν καινούρια είδη ή άλλαξαν τα ήδη υπάρχοντα είδη στα οποία ανήκαν (π.χ. εικονικά κατοικίδια ζώα), ώθησαν τα όρια των βιντεοπαιχνιδιών ή σε ορισμένες περιπτώσεις πρόσθεσαν νέες δυνατότητες στα παιχνίδια (όπως τίτλοι που έχουν σχεδιαστεί ειδικά για κάποιες συσκευές όπως το EyeToy της Sony). Μερικά είδη αποτελούν συνδυασμούς των άλλων, όπως τα μαζικά διαδικτυακά παιχνίδια ρόλων πολλών παικτών, ή, αλλιώς, MMORPGs (Massive Multiplayer Online Role Playing Games). Είναι επίσης κοινό να δούμε υψηλότερου επιπέδου ορισμούς ειδών, οι οποίοι έχουν συλλογικό χαρακτήρα για όλα τα άλλα είδη, όπως τα action, μουσικής φύσης ή βιντεοπαιχνίδια τρόμου. Γενικότερα τα βιντεοπαιχνίδια αυτή την στιγμή μπορούν να κατηγοριοποιηθούν ως εξής [5]:

● Βασικά παιχνίδια (core games). Τα βασικά παιχνίδια ορίζονται γενικώς από την έντασή τους, το βάθος του παιχνιδιού ή την κλίμακα της παραγωγής κατά την δημιουργία τους και μπορεί να περιλαμβάνει παιχνίδια από ένα ευρύ φάσμα ειδών. Για παράδειγμα, η σειρά Bit.Trip για το WiiWare, η σειρά Fallout για το PC και την κονσόλα ή το LittleBigPlanet για το PS3, όλοι εμπίπτουν στην κατηγορία βασικών παιχνιδιών. Τα βασικά παιχνίδια είναι μερικές φορές απαιτητικά στον τρόπο χειρισμού τους και συνήθως δεν προτιμούνται από περιστασιακούς παίκτες.

● Απλά παιχνίδια (casual games). Τα απλά παιχνίδια ονομάζονται έτσι επειδή είναι εύκολα, έχουν απλό τρόπο χειρισμού και ευκολονόητους κανόνες. Τα απλά παιχνίδια ως μορφή υπήρχαν πολύ πριν να επινοηθεί ο όρος και περιλαμβάνουν βιντεοπαιχνίδια, όπως η Πασιέντζα ή ο Ναρκαλιευτής που μπορούν συνήθως να βρεθούν προ-εγκατεστημένα με πολλές εκδόσεις του λειτουργικού συστήματος Microsoft Windows.

● Σοβαρά παιχνίδια (serious games). Τα σοβαρά παιχνίδια είναι παιχνίδια που έχουν αναπτυχθεί κυρίως για να μεταφέρουν πληροφορίες ή μια μαθησιακή εμπειρία κάποιου είδους στον παίκτη. Ορισμένα σοβαρά παιχνίδια μπορεί ακόμη και να μην χαρακτηρίζονται ως βιντεοπαιχνίδια με την παραδοσιακή έννοια του όρου. Επίσης, το εκπαιδευτικό λογισμικό συνήθως εμπίπτει στην κατηγορία αυτή (π.χ. λογισμικό μάθησης πληκτρολόγησης, λογισμικό εκμάθησης γλωσσών, κλπ.), καθώς η πρωταρχική διάκριση τους βασίζεται αρχικά στον τίτλο, καθώς και στις ηλικίες που απευθύνεται. Όπως και με τις άλλες κατηγορίες, αυτή η περιγραφή είναι πιο πολύ μια κατευθυντήρια γραμμή παρά ένας κανόνας.

● Παιχνίδια εκπαιδευτικού χαρακτήρα (educational games). Υπάρχουν πολλά διαφορετικά είδη και μορφές εκπαιδευτικών παιχνιδιών από μάθηση ορθογραφίας και μαθηματικών για παιδιά μέχρι και

για ενηλίκους. Κάποια άλλα παιχνίδια δεν απευθύνονται σε κάποιο ιδιαίτερο κοινό στο μυαλό και έχουν ως στόχο απλά να εκπαιδεύσουν ή να ενημερώσουν όποιον παίζει το παιχνίδι.

Οι κατηγορίες των παιχνιδιών δεν πρέπει να συγχέονται με τα είδη των παιχνιδιών που αναπτύσσονται. Η κατηγοριοποίηση των παιχνιδιών σε είδη βασίζεται στην αλληλεπίδραση με τον χρήστη και στον τρόπο χειρισμού τους. Μπορούμε να κατατάξουμε τα βιντεοπαιχνίδια στα παρακάτω είδη:

- Δράσης (Action).
- Δράσης – περιπέτειας (Action – adventure).
- Περιπέτειας (Adventure).
- Παιχνίδι ρόλων (Role playing).
- Προσομοίωσης (Simulation).
- Στρατηγικής (Strategy).

Η εξέλιξη των παιχνιδιών αυτή τη στιγμή έχει φτάσει σε πολύ υψηλό επίπεδο. Θα μπορούσαμε να διακρίνουμε δύο βασικούς παράγοντες, οι οποίοι συντελούν στην επιτυχία ενός παιχνιδιού και παίζουν το μεγαλύτερο ρόλο στην κατάταξη ενός παιχνιδιού στην σύγχρονη στάθμη εξέλιξης των βιντεοπαιχνιδιών. Πρώτον, τα γραφικά του παιχνιδιού και συγκεκριμένα το επίπεδο λεπτομέρειας των μοντέλων και του περιβάλλοντος στο οποίο εκτυλίσσεται το παιχνίδι και δεύτερον η δυνατότητα του παιχνιδιού για δικτύωση, δηλαδή να μπορούν πολλοί παίκτες να παίξουν μεταξύ τους. Δευτερεύοντες παράγοντες μπορεί να είναι η ύπαρξη ενός καλού σεναρίου, καινοτόμοι τρόποι χειρισμού κ.α. Γενικότερα τα βιντεοπαιχνίδια αναπτύσσονται ως ένα μέσο ψυχαγωγίας ή εκπαίδευσης, το οποίο όμως αλληλεπιδρά με τις σύγχρονες ανάγκες των χρηστών και εξελίσσεται βάση αυτών. Η ανάπτυξη βιντεοπαιχνιδιών λοιπόν, μπορεί να θεωρηθεί ως μια τέχνη η οποία ακολουθεί την εξέλιξη της τεχνολογίας και των δυνατοτήτων που αυτή προσφέρει, αλλά είναι και αποτέλεσμα έρευνας και της δημιουργικής φύσης του ανθρώπου. Επιπλέον όσο προοδεύει η ανάπτυξη των παιχνιδιών, τόσο πιο πολύ θα προοδεύουν και οι απαιτήσεις των χρηστών. Όπως μια ασπρόμαυρη ταινία που είχε τεράστια επιτυχία στην εποχής της δεν πρόκειται να έχει το ίδιο αποτέλεσμα σήμερα, έτσι και ένα παιχνίδι με απλά γραφικά δεν θα κάνει την ίδια εντύπωση που θα είχε όταν η σχεδίαση γραφικών ήταν κάτι καινούριο. Άρα ένα σύγχρονο παιχνίδι μπορεί να οριστεί ως το παιχνίδι το οποίο καλύπτει σε επαρκή βαθμό τις σύγχρονες απαιτήσεις των χρηστών.

2. OpenGL

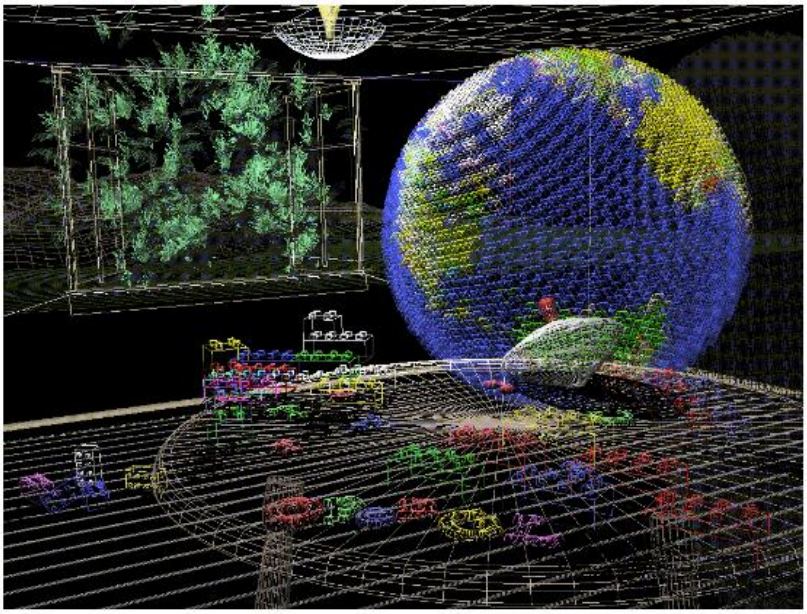
2.1. Εισαγωγή στην OpenGL

Στο κεφάλαιο αυτό αναφέρω τα βασικά χαρακτηριστικά της OpenGL, όπως περιγράφονται στο βιβλίο “The Official Guide to Learning OpenGL” [1]. Η OpenGL είναι μια διεπαφή λογισμικού για χρήση γραφικών. Αυτή η διεπαφή αποτελείται από ένα πλήθος διαφορετικών εντολών που χρησιμοποιούνται για να προσδιορίζουν τα αντικείμενα και τις ενέργειες που απαιτούνται για την παραγωγή διαδραστικών τρισδιάστατων εφαρμογών. Η OpenGL είναι σχεδιασμένη ως μία βελτιωμένη διεπαφή ανεξάρτητη από το υλισμικό και μπορεί να εφαρμοστεί σε πολλές διαφορετικές πλατφόρμες υλικού. Για την επίτευξη αυτών των ιδιοτήτων, δεν περιλαμβάνονται εντολές για την εκτέλεση εργασιών παραθύρου ή για την λήψη δεδομένων εισόδου από τον χρήστη, αντ 'αυτού, πρέπει ο προγραμματιστής να εργαστεί με οποιοδήποτε σύστημα παραθύρων ελέγχει το συγκεκριμένο υλικό που χρησιμοποιεί. Ομοίως, η OpenGL δεν παρέχει υψηλού επιπέδου εντολές για την περιγραφή των μοντέλων των τρισδιάστατων αντικειμένων. Τέτοιες εντολές μπορούν να επιτρέψουν την διευκρίνιση σχετικά πολύπλοκων σχημάτων, όπως αυτοκίνητα, μέρη του σώματος, αεροπλάνα κλπ. Με την OpenGL, πρέπει ο προγραμματιστής-σχεδιαστής να δημιουργήσει το επιθυμητό μοντέλο από ένα μικρό σύνολο των ακόλουθων γεωμετρικών σχημάτων - σημεία, γραμμές και πολύγωνα.

Μία εξελιγμένη βιβλιοθήκη η οποία παρέχει αυτές τις δυνατότητες θα μπορούσε ασφαλώς να δομηθεί πάνω στην OpenGL. Η Βοηθητική Βιβλιοθήκη OpenGL (OpenGL Utility Library – GLU) παρέχει πολλές δυνατότητες μοντελοποίησης, όπως καμπύλες και επιφάνειες NURBS. Η GLU είναι ένα πρότυπο μέρος κάθε εφαρμογής OpenGL. Επίσης, υπάρχει μία υψηλότερου επίπεδου, αντικειμενοστρεφής βιβλιοθήκη, η Open Inventor, που είναι δομημένη πάνω από την OpenGL, και διατίθεται χωριστά σε πολλές υλοποιήσεις της OpenGL.

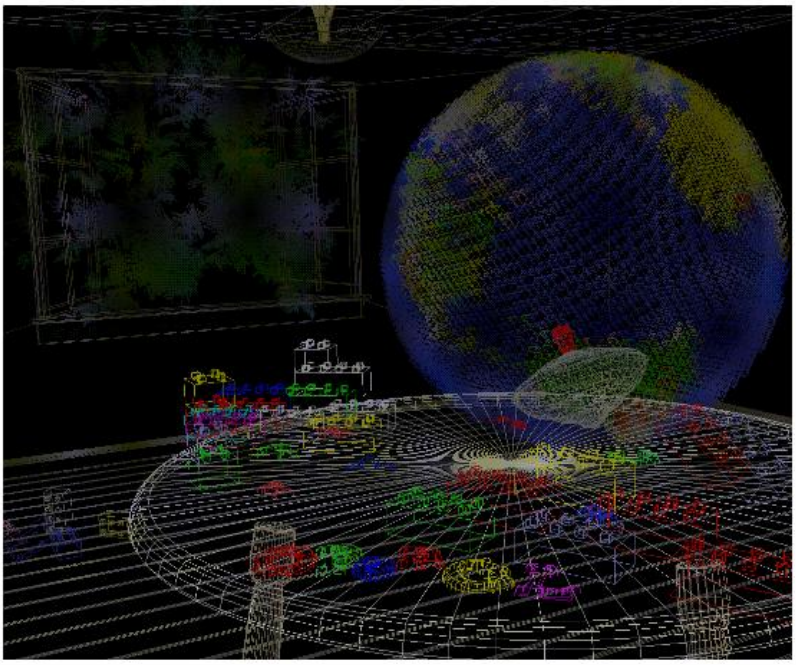
Τώρα που ξέρουμε τι δεν κάνει η OpenGL, θα δούμε τι μπορεί να κάνει. Παρακάτω βλέπουμε μερικά παραδείγματα που απεικονίζουν τυπικές χρήσεις της OpenGL.

● Η εικόνα 2.1 δείχνει κάποια τρισδιάστατα αντικείμενα να εμφανίζονται ως μοντέλα δικτυώματος, δηλαδή σαν όλα τα αντικείμενα στη σκηνή να ήταν κατασκευασμένα από σύρμα. Κάθε γραμμή του σύρματος αντιστοιχεί σε μία ακμή ενός βασικού σχήματος (συνήθως ενός πολυγώνου). Για παράδειγμα, η επιφάνεια του πίνακα είναι κατασκευασμένη από τριγωνικά πολύγωνα που τοποθετούνται σαν φέτες μιας πίτας.



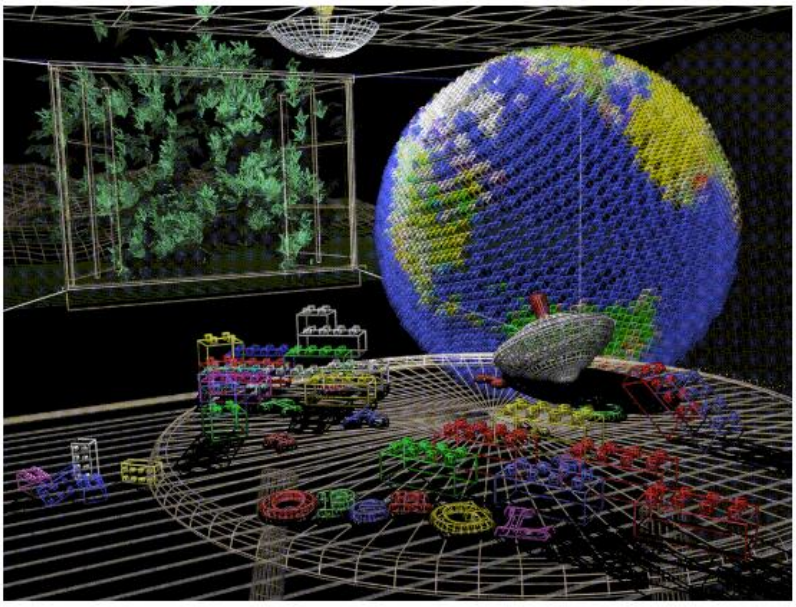
Εικόνα 2.1: Μοντέλα δικτυώματος

● Η εικόνα 2.2 δείχνει μία εκδοχή της ίδιας σκηνής δικτυώματος με εισαγωγή βάθους. Παρατηρούμε ότι οι γραμμές μακρύτερα από το μάτι είναι θολές, ακριβώς όπως θα ήταν στην πραγματική ζωή, δίνοντας έτσι μία οπτική ψευδαίσθηση του βάθους. Η OpenGL χρησιμοποιεί ατμοσφαιρικές επιδράσεις (συλλογικά αναφέρεται ως ομίχλη) για την επίτευξη βάθους.



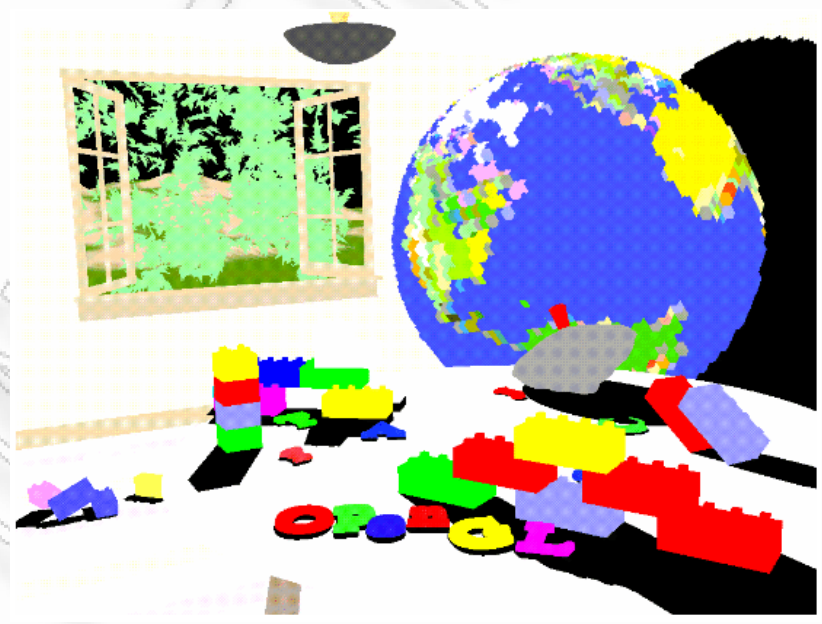
Εικόνα 2.2: Δικτύωμα με εισαγωγή βάθους

● Η εικόνα 2.3 δείχνει μία εκδοχή της ίδιας παράστασης δικτυώματος με χρήση τεχνικής εξομάλυνσης (antialiasing). Η εξομάλυνση είναι μια τεχνική για τη μείωση των οδοντωτών ακμών που δημιουργούνται όταν προσεγγίζονται στρογγυλευμένες γωνίες με χρήση εικονοκυττάρων.



Εικόνα 2.3: Δικτύωμα με εξομάλυνση (antialiasing)

● Η εικόνα 2.4 δείχνει την ίδια εικόνα με επίπεδη σκίαση (ένα μόνο χρώμα για κάθε πολύγωνο). Τα αντικείμενα στην σκηνή εμφανίζεται τώρα ως στερεά. Εμφανίζονται επίπεδα, υπό την έννοια ότι μόνο ένα χρώμα χρησιμοποιείται για να καταστήσει κάθε πολύγωνο, έτσι δεν εμφανίζονται ομαλά στρογγυλεμένα. Επίσης δεν υπάρχουν επιπτώσεις από τυχόν πηγές φωτός.



Εικόνα 2.4: Χρήση επίπεδης σκίασης (ένα χρώμα για κάθε σχεδιασμένο πολύγωνο)

● Η εικόνα 2.5 δείχνει μία φωτεινή, λεία-σκιασμένη έκδοση της σκηνής. Σημειώστε πως η σκηνή μοιάζει πολύ πιο ρεαλιστική και τρισδιάστατη όταν τα αντικείμενα είναι σκιασμένα με τέτοιο τρόπο ώστε να ανταποκρίνονται στις πηγές φωτός μέσα στο δωμάτιο, σαν να ήταν ομαλά στρογγυλεμένα.

Εικόνα 2.5: Εισαγωγή φωτισμού στην σκηνή και ομαλή σκίαση των πολυγώνων

● Στην εικόνα 2.6 έχουν προστεθεί σκιές και υφές στην προηγούμενη έκδοση της σκηνής. Οι σκιές δεν αποτελούν ρητά χαρακτηριστικό της OpenGL (δεν υπάρχει καμία εντολή για χρήση σκιών), αλλά μπορούν να δημιουργηθούν με χρήση κάποιων τεχνικές. Η διαδικασία της χαρτογράφησης υφής (texture mapping) επιτρέπει να εφαρμοστεί μία δισδιάστατη εικόνα σε ένα τρισδιάστατο αντικείμενο. Σε αυτή τη σκηνή, η κορυφή της επιφάνειας του τραπεζιού είναι το πιο ζωντανό παράδειγμα της χαρτογράφησης υφής. Τα νερά του ξύλου στις επιφάνειες του πατώματος και του τραπεζιού είναι όλα με χαρτογράφηση υφής, καθώς και η ταπετσαρία και το πάνω πάνω παιχνίδι στο τραπέζι.

Εικόνα 1.6: Εισαγωγή σκιών και υφών στην σκηνή

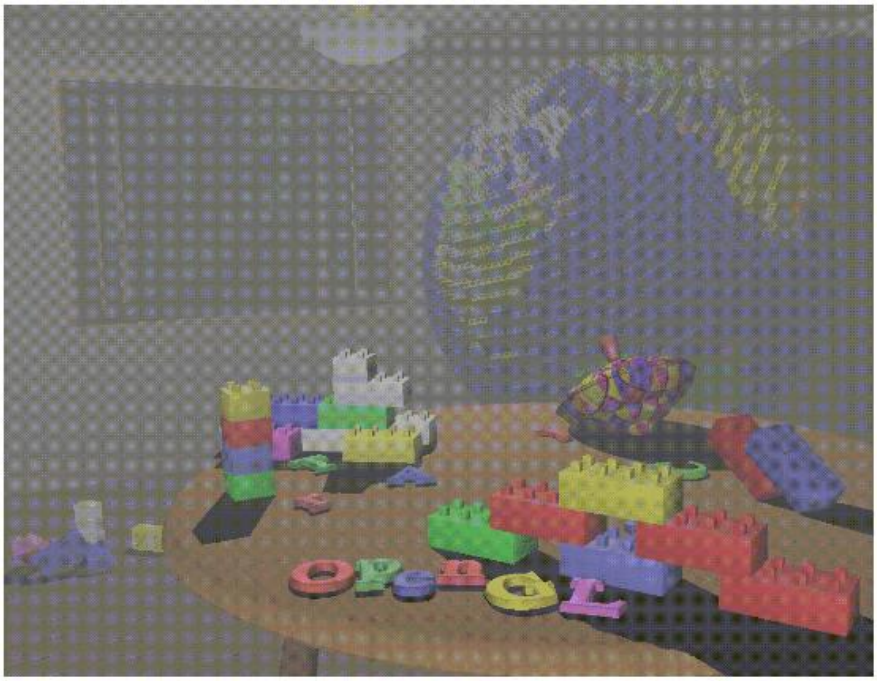
● Η εικόνα 1.7 δείχνει ένα αντικείμενο με θόλωμα της κίνησης του στη σκηνή. Τα παιχνίδια στο τραπέζι φαίνονται να κινούνται προς τα εμπρός, αφήνοντας ένα θολό ίχνος της διαδρομής του, της κίνησης.

Εικόνα 2.7: Σχεδίαση αντικειμένων με θολό ίχνος της κίνησης τους

● Η εικόνα 2.8 δείχνει τη σκηνή από μια διαφορετική οπτική γωνία. Από αυτή την εικόνα καταλαβαίνουμε ότι η εικόνα είναι πραγματικά ένα στιγμιότυπο των μοντέλων των τρισδιάστατων αντικειμένων.

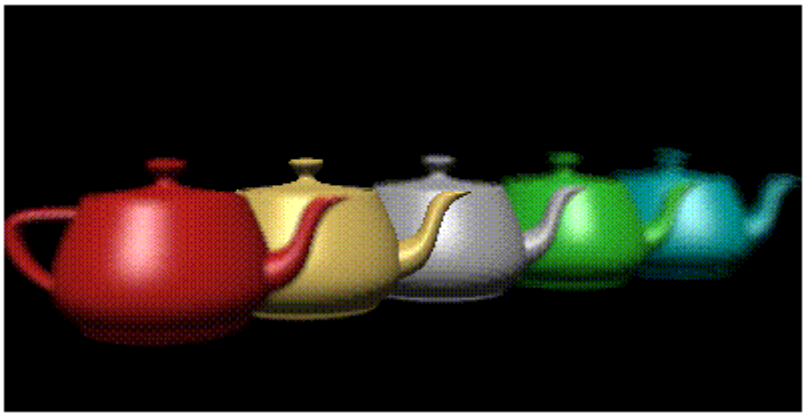
Εικόνα 2.8: Διαφορετική οπτική γωνία της σκηνής

● Η εικόνα 2.9 επαναφέρει τη χρήση της ομίχλης, που είδαμε στην εικόνα 2.2, για να δείξουν την παρουσία των σωματιδίων καπνού στον αέρα. Παρατηρούμε πώς η χρήση της ίδιας τεχνικής στην εικόνα 2.2 έχει πολύ πιο έντονο αποτέλεσμα στην εικόνα 2.9.



Εικόνα 2.9: Εισαγωγή ομίχλης στην σκηνή

● Η εικόνα 2.10 δείχνει το εφέ *βάθος-του-πεδίου* (*depth-of-field effect*), το οποίο προσομοιώνει την αδυναμία του φακού της κάμερας να διατηρήσει όλα τα αντικείμενα σε μια σκηνή εστιασμένα. Η κάμερα εστιάζει σε ένα συγκεκριμένο σημείο στη σκηνή. Αντικείμενα που είναι πιο κοντά ή μακρύτερα από εκείνο το σημείο είναι κάπως θολά.



Εικόνα 2.10: Θόλωμα αντικειμένων που δεν είναι εστιασμένα

Οι παραπάνω εικόνες σας δίνουν μια ιδέα για το είδος των πραγμάτων που μπορούν να γίνουν με το σύστημα γραφικών OpenGL. Η ακόλουθη σύντομη λίστα περιγράφει τις σημαντικότερες λειτουργίες γραφικών που εκτελεί η OpenGL για να καταστήσει μια εικόνα στην οθόνη. (Βλ. "OpenGL Rendering Pipeline" για αναλυτικές πληροφορίες σχετικά με αυτήν την σειρά των λειτουργιών).

1. Κατασκευή σχημάτων από τις βασικές γεωμετρίες, δημιουργώντας έτσι μαθηματικές περιγραφές των αντικειμένων. (Η OpenGL θεωρεί τα σημεία, τις γραμμές, τα πολύγωνα, τις εικόνες και τα δυαδικά αρχεία σαν τις βασικές γεωμετρίες.)

2. Τακτοποίηση των αντικειμένων σε τρισδιάστατο χώρο και επιλογή του επιθυμητού σημείου για την προβολή της αποτελούμενης σκηνής.

3. Υπολογισμός των χρωμάτων όλων των αντικειμένων. Το χρώμα μπορεί να ανατεθεί ρητά από την εφαρμογή, να καθοριστεί από συγκεκριμένες συνθήκες φωτισμού, να ληφθεί με την επικόλληση μιας υφής επάνω στα αντικείμενα, ή κάποιο συνδυασμό αυτών των τριών ενεργειών.

4. Μετατροπή της μαθηματικής περιγραφής των αντικειμένων και των σχετικών πληροφοριών του χρώματος τους σε εικονοστοιχεία (pixels) στην οθόνη. Αυτή η διαδικασία καλείται ραστεροποίηση (rasterization).

Κατά τη διάρκεια αυτών των σταδίων, η OpenGL μπορεί να εκτελεί και άλλες λειτουργίες, όπως η εξάλειψη μερών αντικειμένων που είναι κρυμμένα από άλλα αντικείμενα. Έπειτα το σκηνικό ραστεροποιείται αλλά προτού να σχεδιαστεί στην οθόνη, μπορούν να εκτελεστούν κάποιες εργασίες σχετικά με τα δεδομένα εικονοστοιχείων.

Σε ορισμένες εφαρμογές (όπως με το X Window System), η OpenGL είναι σχεδιασμένη να λειτουργεί ακόμη και αν ο υπολογιστής που εμφανίζει τα γραφικά που έχουν δημιουργηθεί δεν είναι ο υπολογιστής που τρέχει το πρόγραμμα γραφικών. Αυτό μπορεί να συμβαίνει αν η εφαρμογή υλοποιείται σε ένα δικτυακό περιβάλλον στο οποίο πολλοί υπολογιστές είναι συνδεδεμένοι μεταξύ τους με ένα ψηφιακό δίκτυο. Σε αυτήν την περίπτωση, ο υπολογιστής στον οποίο τρέχει το πρόγραμμα που έχει τις εντολές σχεδίασης OpenGL καλείται “πελάτης”, και ο υπολογιστής που λαμβάνει τις εντολές και εκτελεί την σχεδίαση ονομάζεται “διακομιστής”. Οι κανόνες για τη διαβίβαση των εντολών OpenGL (αποκαλούμενο πρωτόκολλο) από τον client στον server είναι πάντα οι ίδιοι, οπότε τα OpenGL προγράμματα μπορούν να εργαστούν σε ένα δίκτυο, ακόμη και αν ο πελάτης και ο διακομιστής είναι
διαφορετικά είδη υπολογιστών. Εάν ένα OpenGL πρόγραμμα δεν λειτουργεί μέσω ενός δικτύου, τότε υπάρχει μόνο ένας υπολογιστής, και λειτουργεί ως πελάτη και διακομιστής ταυτόχρονα.

Αν παρατηρήσουμε ένα απλό πρόγραμμα, οι OpenGL εντολές χρησιμοποιούν το πρόθεμα gl και κεφαλαία γράμματα για κάθε αρχική λέξη, συνθέτοντας έτσι το όνομα της εντολής (glClearColor (), για παράδειγμα). Ομοίως, οι προκαθορισμένες σταθερές της OpenGL αρχίζουν με το πρόθεμα GL_ και χρησιμοποιούνται κάτω παύλες για να ξεχωρίζουν οι λέξεις (π.χ. GL_COLOR_BUFFER_BIT).

Μπορεί επίσης να παρατηρήσουμε κάποια φαινομενικά ξένα γράμματα σε ορισμένες εντολές (για παράδειγμα, το 3f στις εντολές glColor3f () και glVertex3f ()). Είναι αλήθεια ότι η λέξη Color του

ονόματος glColor3f της εντολής () είναι αρκετή για να καθορίσει την εντολή ως αυτή που θέτει το τρέχον χρώμα. Ωστόσο, περισσότερες από μία τέτοιες εντολές έχουν οριστεί έτσι ώστε να μπορούμε να χρησιμοποιήσουμε διαφορετικούς τύπους ορισμάτων. Ειδικότερα, ο αριθμός 3 δείχνει ότι δίνονται τρία ορίσματα, μια άλλη έκδοση της εντολής Color παίρνει τέσσερα επιχειρήματα. Το f του επιθέματος δείχνει ότι τα ορίσματα είναι αριθμοί κινητής υποδιαστολής. Η ύπαρξη διαφορετικών μορφότυπων επιτρέπει στην OpenGL να αποδεχθεί τα δεδομένα του χρήστη με τη δική του μορφή δεδομένων.

Μερικές εντολές OpenGL δέχονται μέχρι και 8 διαφορετικούς τύπους δεδομένων για τα ορίσματα τους. Τα γράμματα που χρησιμοποιούνται ως καταλήξεις για να διευκρινίσουν αυτούς τους τύπους δεδομένων φαίνονται στον παρακάτω πίνακα, μαζί με τους αντίστοιχους τύπους ορισμού στην OpenGL.

Suffix	Data Type	Typical Corresponding C-Language Type	OpenGL Type Definition
b	8-bit integer	signed char	GLbyte
s	16-bit integer	short	GLshort
i	32-bit integer	int or long	GLint, GLsizei
f	32-bit floating-point	float	GLfloat, GLclampf
d	64-bit floating-point	double	GLdouble, GLclampd
ub	8-bit unsigned integer	unsigned char	GLubyte, GLboolean
us	16-bit unsigned integer	unsigned short	GLushort
ui	32-bit unsigned integer	unsigned int or unsigned long	GLuint, GLenum, GLbitfield

Έτσι, οι δύο εντολές

```
glVertex2i (1, 3);
glVertex2f (1,0, 3,0);
```

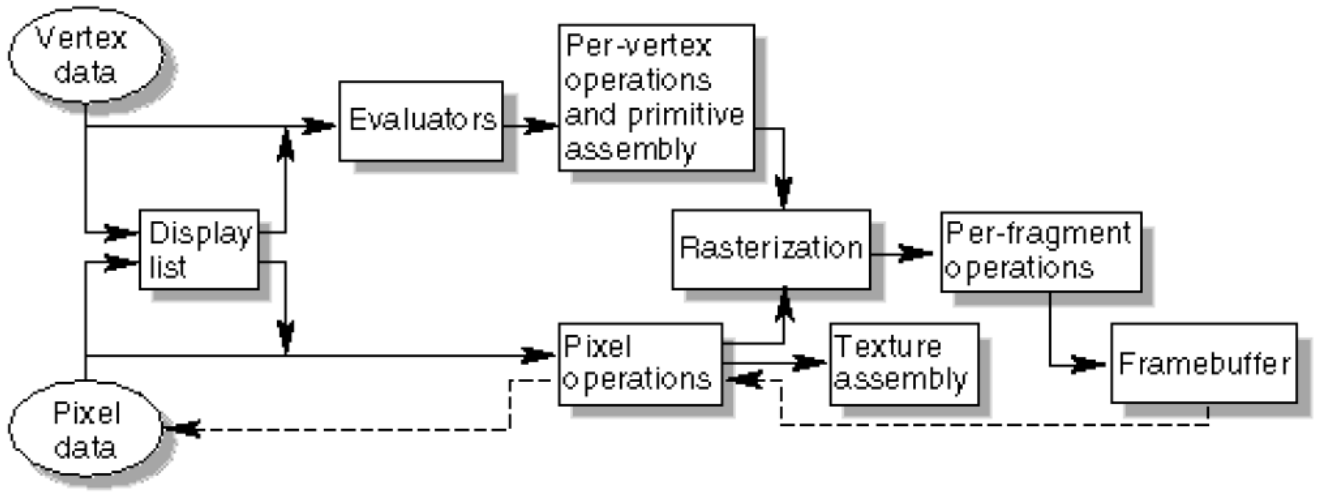
είναι ισοδύναμες, εκτός από το ότι η πρώτη καθορίζει τις συντεταγμένες της κορυφής, με ακέραιους 32-bit, και η δεύτερη τις καθορίζει με μονής ακρίβειας αριθμούς κινητής υποδιαστολής.

Μερικές εντολές OpenGL μπορούν να λάβουν το τελικό γράμμα v, το οποίο δείχνει ότι η εντολή παίρνει ένα δείκτη σε ένα πίνακα τιμών και όχι μια σειρά από επιμέρους τιμές. Πολλές εντολές υποστηρίζουν εισαγωγή τιμών και με πίνακα και κανονικά, αλλά ορισμένες εντολές δέχονται μόνο επί μέρους ορίσματα και άλλες απαιτούν τουλάχιστον κάποια από τα ορίσματα να δίδονται με πίνακα τιμών. Οι παρακάτω γραμμές δείχνουν πώς μπορείτε να χρησιμοποιήσετε την ίδια εντολή με εισαγωγή των ορισμάτων σε πίνακα και χωρίς:

```
glColor3f(1.0, 0.0, 0.0);

GLfloat color_array[] = {1.0, 0.0, 0.0};
glColor3fv(color_array);
```

Η OpenGL είναι μια μηχανή καταστάσεων. Την θέτουμε σε διάφορες καταστάσεις (ή λειτουργίες) που στη συνέχεια παραμένουν σε ισχύ μέχρι να αλλαχθούν. Για παράδειγμα, το τρέχον χρώμα είναι μια κατάσταση. Μπορούμε να ορίσουμε το τρέχον χρώμα σε λευκό, κόκκινο, ή οποιοδήποτε άλλο χρώμα, και στη συνέχεια κάθε αντικείμενο που σχεδιάζεται θα έχει αυτό το χρώμα, μέχρι να ορίσουμε το τρέχον χρώμα σε κάτι άλλο. Το τρέχον χρώμα είναι μόνο μία από τις πολλές μεταβλητές κατάστασης που διατηρεί η OpenGL. Άλλες ελέγχουν πράγματα όπως τους μετασχηματισμούς θέασης ή προβολής, τον τρόπο σχεδίασης (με γραμμή ή πολύγωνα), τον τρόπο σχεδίασης πολύγονων, τη θέση και τα χαρακτηριστικά του φωτός και τις ιδιότητες των υλικών των αντικειμένων στο στάδιο της επεξεργασίας. Πολλές μεταβλητές κατάστασης αναφέρονται σε λειτουργίες που ενεργοποιούνται ή απενεργοποιούνται με την εντολή glEnable () ή glDisable ().

Οι περισσότερες εφαρμογές της OpenGL έχουν παρόμοια σειρά λειτουργιών, μια σειρά από στάδια επεξεργασίας που ονομάζεται OpenGL rendering pipeline. Αυτή η διάταξη, όπως φαίνεται στην εικόνα 2.11, δεν αποτελεί έναν αυστηρό κανόνα για το πώς η OpenGL υλοποιείται, αλλά παρέχει ένα αξιόπιστο οδηγό για το τι κάνει η OpenGL.



Εικόνα 2.11: OpenGL rendering pipeline

Γεωμετρικά δεδομένα (κορυφές, γραμμές και πολύγωνα) ακολουθούν την διαδρομή που περιλαμβάνει αξιολογητές (evaluators) και ανά-κορυφή λειτουργίες (per-vertex operations), ενώ τα δεδομένα εικονοστοιχείων (εικονοστοιχεία, εικόνες, και εικόνες bitmap) αντιμετωπίζονται διαφορετικά για ένα μέρος της διαδικασίας. Και τα δύο είδη δεδομένων καταλήγουν στα ίδια τελικά βήματα (rasterization και ανά τεμάχιο λειτουργίες) πριν από την εγγραφή των τελικών δεδομένα εικονοστοιχείων στον ενταμιευτή framebuffer. Παρακάτω εξηγώ τις έννοιες που συναντάμε στην εικόνα 2.11.

Display Lists: Όλα τα δεδομένα, ανεξαρτήτως αν περιγράφουν μια γεωμετρία ή ένα εικονοστοιχείο, μπορούν να αποθηκευτούν σε μια λίστα απεικόνισης (display list) για τρέχουσα ή μελλοντική χρήση. (Η εναλλακτική στην διατήρηση των δεδομένων σε μια λίστα απεικόνισης είναι η άμεση επεξεργασία των δεδομένων, επίσης γνωστή και ως άμεση λειτουργία (immediate mode)). Όταν μία λίστα απεικόνισης
εκτελείται, τα αποθηκευμένα δεδομένα στέλνονται από την λίστα απεικόνισης ακριβώς σαν να είχαν αποσταλεί από την εφαρμογή σε άμεση λειτουργία.

Evaluators: Όλοι οι γεωμετρικοί πρωταρχικοί τύποι ουσιαστικά περιγράφονται από τις κορυφές. Παραμετρικές καμπύλες και επιφάνειες μπορούν αρχικά να περιγραφούν από σημεία ελέγχου και πολυωνυμικές συναρτήσεις οι οποίες ονομάζονται βασικές συναρτήσεις. Οι αξιολογητές (evaluators)

παρέχουν μια μέθοδο για να προκύψουν οι κορυφές που χρησιμοποιούνται για να αναπαραστήσουν μια επιφάνεια από τα σημεία ελέγχου. Η μέθοδος αυτή είναι μια πολυωνυμική χαρτογράφηση, η οποία μπορεί να παράγει συντεταγμένες υφής, χρώματα, και συντονιστικές τιμές συντεταγμένων από τα σημεία ελέγχου.

Per-vertex operations: Τα δεδομένα κορυφών στέλνονται στο στάδιο per-vertex operations (ανα κορυφή λειτουργίες), το οποίο μετατρέπει τις κορυφές σε πρωταρχικούς τύπους. Μερικά δεδομένα κορυφών μετατρέπονται από 4 x 4 μήτρες κινητής υποδιαστολής.

Primitive assembly: Η ψαλίδιση, ένα σημαντικό μέρος της συναρμολόγησης πρωταρχικών σχημάτων, είναι η εξάλειψη των τμημάτων της γεωμετρίας που βρίσκονται εκτός ενός διαστήματος, το οποίο ορίζεται από ένα επίπεδο. Η ψαλίδιση σημείων απλώς δέχεται ή απορρίπτει κορυφές. Η ψαλίδιση γραμμών ή πολυγώνων μπορεί να προσθέσει επιπλέον κορυφές, ανάλογα με το πώς η γραμμή ή το πολύγωνο ψαλιδίζεται.

Pixel operations: Ενώ τα γεωμετρικά δεδομένα ακολουθούν μία διαδρομή, τα δεδομένα εικονοστοιχείων ακολουθούν μια διαφορετική διαδρομή. Τα εικονοστοιχεία ενός πίνακα στην μνήμη του συστήματος πρώτα μετατρέπονται από ένα μορφότυπο σε ένα πλήθος στοιχείων. Στη συνέχεια τα δεδομένα κλιμακώνονται και υφίστανται επεξεργασία από ένα χάρτη εικονοστοιχείων. Τα αποτελέσματα συλλέγονται και στη συνέχεια είτε μεταφέρονται στην μνήμη υφής είτε αποστέλλονται στο στάδιο του rasterization.

Texture assembly: Μια εφαρμογή OpenGL μπορεί να επιθυμεί να εφαρμόσει εικόνες υφής επάνω σε γεωμετρικά αντικείμενα για να φαίνονται πιο ρεαλιστικά. Εάν πολλές εικόνες υφής χρησιμοποιούνται, είναι πρακτικό να τις αποθηκεύσουμε σε αντικείμενα υφής έτσι ώστε να μπορούμε εύκολα να τις εναλλάσσουμε.

Rasterization: Η τεχνική ραστεροποίησης (rasterization) είναι η μετατροπή των δεδομένων γεωμετρίας και εικονοστοιχείων σε τεμάχια. Κάθε τμήμα του τεμαχίου αντιστοιχεί σε ένα εικονοστοιχείο στον

ενταμιευτή πλαισίων (framebuffer). Τα σημεία γραμμών και πολυγώνων, το πλάτος γραμμών, το μέγεθος σημείων, η σκίαση του μοντέλου, και οι υπολογισμοί κάλυψης για την εξομάλυνση (antialiasing) λαμβάνονται υπόψη όπως οι κορυφές συνδέονται σε γραμμές ή όπως υπολογίζονται τα εσωτερικά εικονοστοιχεία ενός γεμάτου πολυγώνου. Τιμές χρώματος και βάθους εκχωρούνται για κάθε τμήμα τεμαχίου.

Fragment operations: Πριν αποθηκευθούν οι κατάλληλες τιμές στον ενταμιευτή πλαισίου (framebuffer), μια σειρά από εργασίες εκτελούνται που μπορούν να τροποποιήσουν ή ακόμα και να απορρίψουν κάποια τεμάχια. Όλες αυτές οι λειτουργίες μπορούν να ενεργοποιηθούν ή να απενεργοποιηθούν.

2.2. Διαχείριση κατάστασης και σχεδίαση γεωμετρικών αντικειμένων

Αν και μπορούμε να σχεδιάσουμε σύνθετες και ενδιαφέρουσες εικόνες χρησιμοποιώντας την OpenGL, όλες κατασκευάζονται από ένα μικρό αριθμό βασικών γραφικών στοιχείων. Στο υψηλότερο επίπεδο αφαίρεσης, υπάρχουν τρεις βασικές λειτουργίες σχεδίασης: καθαρισμός του παραθύρου, σχεδίαση ενός γεωμετρικού αντικειμένου, και σχεδίαση ενός ράστερ αντικειμένου. Τα ράστερ αντικείμενα, περιλαμβάνουν δισδιάστατες εικόνες, εικόνες bitmap και γραμματοσειρές.

Η σχεδίαση στην οθόνη ενός υπολογιστή είναι διαφορετική από την σχεδίαση σε χαρτί καθώς το χαρτί είναι λευκό, και το μόνο που έχουμε να κάνουμε είναι να σχεδιάσουμε την εικόνα. Σε έναν υπολογιστή, η μνήμη που έχει την εικόνα είναι συνήθως γεμάτη με την τελευταία εικόνα που σχεδιάστηκε, έτσι συνήθως πρέπει να είναι καθαριστεί με τον ορισμό κάποιου χρώματος του φόντου πριν αρχίσει η σχεδίαση της νέας σκηνής. Το χρώμα που χρησιμοποιούμε για το φόντο εξαρτάται από την εφαρμογή. Για έναν επεξεργαστή κειμένου, μπορούμε να χρησιμοποιήσουμε το λευκό (το χρώμα του χαρτιού) προτού να αρχίσουμε την σύνταξη του κειμένου. Αν θέλουμε να σχεδιάσουμε την θέα από ένα διαστημόπλοιο, θα χρησιμοποιήσουμε το μαύρο πριν από την σχεδίαση των άστρων, των πλανήτων, και των άλλων διαστημοπλοίων. Μερικές φορές ίσως να μην χρειάζεται να καθαρίσουμε την οθόνη. Για παράδειγμα, αν η εικόνα είναι το εσωτερικό ενός δωματίου, ολόκληρο το παράθυρο γραφικών καλύπτεται καθώς σχεδιάζουμε όλους τους τοίχους του δωματίου.

Ως παράδειγμα, αυτές οι γραμμές κώδικα καθαρίζουν ένα παράθυρο RGBA με μαύρο χρώμα:

```
glClearColor (0,0, 0,0, 0,0, 0,0);
glClear (GL_COLOR_BUFFER_BIT);
```

Η πρώτη γραμμή ορίζει το χρώμα εκκαθάρισης σε μαύρο, και η επόμενη εντολή καθαρίζει ολόκληρο το παράθυρο με το τρέχον χρώμα εκκαθάρισης. Οι μόνες παράμετροι για την glClear () είναι ποιοι ενταμιευτές πρέπει να εκκαθαριστούν. Στην περίπτωση αυτή, το πρόγραμμα καθαρίζει μόνο τον ενταμιευτή χρώματος, όπου η εικόνα που εμφανίζεται στην οθόνη είναι αποθηκευμένη. Συνήθως, ορίζουμε το χρώμα εκκαθάρισης μία φορά, στην αρχή, και στη συνέχεια αδειάζουμε τους ενταμιευτές όσο συχνά χρειάζεται. Η OpenGL παρακολουθεί το τρέχον χρώμα εκκαθάρισης από μια μεταβλητή κατάστασης αντί να χρειάζεται να το καθορίσουμε κάθε φορά που αδειάζουμε έναν ενταμιευτή.

Για παράδειγμα, για να αδειάσουμε τον ενταμιευτή χρώματος και τον ενταμιευτή βάθους, θα χρησιμοποιούσαμε την ακόλουθη ακολουθία εντολών:

```
glClearColor (0,0, 0,0, 0,0, 0,0);
glClearDepth (1,0);
glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Σε αυτή την περίπτωση, η κλήση για glClearColor () είναι η ίδια όπως και πριν, η εντολή glClearDepth () καθορίζει την τιμή του κάθε εικονοστοιχείου του ενταμιευτή βάθους, και η παράμετρος της glClear () εντολής περιέχει το σύνολο όλων των ενταμιευτών που πρέπει να εκκαθαριστούν. Η παρακάτω

περίληψη της glClear () περιλαμβάνει έναν πίνακα που απαριθμεί τους ενταμιευτές που μπορεί να εκκαθαριστούν και τα ονόματά τους.

```
void glClearColor(GLclampf red, GLclampf green, GLclampf blue, GLclampf
alpha);
```

Η παραπάνω μέθοδος ορίζει το τρέχον χρώμα εκκαθάρισης για χρήση, κατά την εκκαθάριση του ενταμιευτή χρώματος σε λειτουργία RGBA. Οι τιμές του κόκκινου, του πράσινου, του μπλε, και η τιμή alpha έχουν εύρος [0,1]. Το προεπιλεγμένο χρώμα εκκαθάρισης είναι το (0, 0, 0, 0), το οποίο είναι το μαύρο.

```
void glClear(GLbitfield mask);
```

Η μέθοδος καθαρίζει τους ενταμιευτές που ορίζονται, στις τρέχουσες τιμές εκκαθάρισης τους. Το όρισμα είναι ένας συνδυασμός των τιμών που αναφέρονται στον πίνακα παρακάτω.

Ενταμιευτής	Όνομα
Color buffer	GL_COLOR_BUFFER_BIT
Depth buffer	GL_DEPTH_BUFFER_BIT
Accumulation buffer	GL_ACCUM_BUFFER_BIT
Stencil buffer	GL_STENCIL_BUFFER_BIT

Πριν από την χρήση της εντολής για την εκκαθάριση πολλαπλών ενταμιευτών, θα πρέπει να καθοριστούν οι τιμές στις οποίες κάθε ενταμιευτής θα καθαριστεί αν θέλουμε κάτι άλλο από την προεπιλεγμένη τιμή του χρώματος RGBA, του βάθους, του χρώματος συσσώρευσης, και του δείκτη stencil. Εκτός από τις glClearColor () και glClearDepth () εντολές που καθορίζουν τις τρέχουσες τιμές για την εκκαθάριση του ενταμιευτή χρώματος και του ενταμιευτή βάθους, η glClearIndex (), η glClearAccum (), και η glClearStencil () προσδιορίζει τον δείκτη χρώματος, το χρώμα συσσώρευσης, καθώς και τον δείκτη stencil που χρησιμοποιούνται για την εκκαθάριση του αντίστοιχου ενταμιευτή.

Η OpenGL επιτρέπει να ορίσουμε πολλαπλούς ενταμιευτές διότι η εκκαθάριση είναι γενικά μια αργή λειτουργία, δεδομένου ότι κάθε εικονοστοιχείο στην οθόνη (ίσως και εκατομμύρια) αλλάζει, και μερικές κάρτες γραφικών επιτρέπουν ένα σύνολο ενταμιευτών να εκκαθαριστούν ταυτόχρονα. Υλισμικό που δεν υποστηρίζει την ταυτόχρονη εκκαθάριση, εκτελεί τις εκκαθαρίσεις διαδοχικά. Η διαφορά μεταξύ της εντολής

```
glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

και της

```
glClear (GL_COLOR_BUFFER_BIT);
glClear (GL_DEPTH_BUFFER_BIT);
```

είναι ότι παρόλο που και οι δύο έχουν το ίδιο τελικό αποτέλεσμα, η πρώτη εντολή θα μπορούσε να τρέξει γρηγορότερα σε πολλούς υπολογιστές. Σίγουρα δεν θα τρέξει πιο αργά.

Στην OpenGL, η περιγραφή του σχήματος ενός αντικειμένου στο στάδιο της επεξεργασίας είναι ανεξάρτητη από την περιγραφή των χρωμάτων του. Κάθε φορά που ένα συγκεκριμένο γεωμετρικό σχήμα σχεδιάζεται, σχεδιάζεται με το τρέχων καθορισμένο σχήμα χρωματισμού. Το σχήμα χρωματισμού αυτό μπορεί να είναι τόσο απλό όσο “σχεδίασε τα πάντα με χρώμα κόκκινο”, ή θα μπορούσε να είναι τόσο περίπλοκο όπως “το αντικείμενο είναι κατασκευασμένο από πλαστικό μπλε χρώματος και υπάρχει ένα κίτρινος προβολέας στραμμένος μια κατεύθυνση και υπάρχει ένα γενικό χαμηλό επίπεδου καφέ-κόκκινο φως παντού”. Σε γενικές γραμμές, ένας προγραμματιστής OpenGL καθορίζει, πρώτον, το χρώμα ή το σύστημα χρωματισμού και, στη συνέχεια, σχεδιάζει τα αντικείμενα. Μέχρι το χρώμα ή ο συνδυασμός χρωμάτων να αλλάξει, όλα τα αντικείμενα σχεδιάζονται με αυτό το χρώμα ή το σύστημα χρωματισμού. Αυτή η μέθοδος βοηθά την OpenGL να επιτύχει υψηλότερη επίδοση σχεδίασης από ότι θα προέκυπτε αν δεν παρακολουθούσε το τρέχον χρώμα. Για παράδειγμα, ο ψευδοκώδικας

```
set_current_color (red);
draw_object (A);
draw_object (B);
set_current_color (green);
set_current_color (blue);
draw_object (C);
```

σχεδιάζει τα αντικείμενα A και B με χρώμα κόκκινο, και το αντικείμενο C με μπλε χρώμα. Η εντολή στην τέταρτη γραμμή που καθορίζει το τρέχον χρώμα σε πράσινο είναι περιττή. Για να ορίσουμε ένα χρώμα, χρησιμοποιούμε την εντολή glColor3f (). Παίρνει τρεις παραμέτρους, οι οποίες είναι όλες αριθμοί κινητής υποδιαστολής μεταξύ 0,0 και 1,0. Οι παράμετροι είναι, κατά σειρά, η κόκκινη, πράσινη και μπλε συνιστώσα του χρώματος. Μπορούμε να σκεφτούμε τις τρεις αυτές τιμές σαν να ορίζουμε ένα “μίγμα” χρωμάτων: η τιμή 0.0 σημαίνει ότι το συγκεκριμένο χρώμα δεν θα χρησιμοποιηθεί ενώ το 1.0 σημαίνει ότι αυτό το χρώμα θα χρησιμοποιηθεί το μέγιστο. Έτσι, ο κώδικας:

```
glColor3f(1.0, 0.0, 0.0);
```

καθορίζει το πιο φωτεινό κόκκινο που το σύστημα μπορεί να σχεδιάσει, χωρίς καθόλου αναμείξεις πράσινου και μπλε χρώματος. Όταν όλες οι τιμές είναι 0.0 δημιουργείται το μαύρο. Αντίθετα, όταν

όλες οι τιμές είναι 1.0, δημιουργείται το λευκό. Η τιμή 0.5 στις τρείς συνιστώσες αποδίδουν το γκρι (ενδιάμεσο του μαύρου και του λευκού). Εδώ είναι οκτώ εντολές και τα χρώματα που θα δημιουργήσουν.

```
glColor3f(0.0, 0.0, 0.0); black
glColor3f(1.0, 0.0, 0.0); red
glColor3f(0.0, 1.0, 0.0); green
glColor3f(1.0, 1.0, 0.0); yellow
glColor3f(0.0, 0.0, 1.0); blue
glColor3f(1.0, 0.0, 1.0); magenta
glColor3f(0.0, 1.0, 1.0); cyan
glColor3f(1.0, 1.0, 1.0); white
```

Όπως είδαμε παραπάνω, η OpenGL είναι ένα σύνολο διαδικασιών. Η κεντρική μονάδα επεξεργασίας (CPU) εκτελεί μια εντολή σχεδίασης. Ίσως κάποιο άλλο υλισμικό κάνει τους γεωμετρικούς μετασχηματισμούς. Η διαδικασία της ψαλίδισης εκτελείται, ακολουθούμενη από την σκίαση ή / και την τοποθέτηση υφής. Τέλος, οι τιμές αποθηκεύονται για να ακολουθήσει η προβολή. Σε υψηλής ποιότητας αρχιτεκτονικές, κάθε μία από αυτές τις λειτουργίες εκτελείται από ένα διαφορετικό κομμάτι του

υλισμικού που έχει σχεδιαστεί για να εκτελεί την συγκεκριμένη λειτουργία γρήγορα. Σε μια τέτοια αρχιτεκτονική, η CPU δεν χρειάζεται να περιμένει για κάθε εντολή σχεδίασης να ολοκληρωθεί πριν να αρχίσει να εκτελεί την επόμενη. Ενώ η CPU στέλνει μια κορυφή, το υλισμικό μετασχηματισμών εργάζεται για τη μετατροπή της τελευταίας που στάλθηκε κ.ο.κ. Σε ένα τέτοιο σύστημα, εάν η CPU περίμενε για κάθε εντολή να ολοκληρωθεί πριν από τη εκτέλεση της επόμενης, η απόδοση του συστήματος θα μειωνόταν σημαντικά.

Επιπλέον, η εφαρμογή μπορεί να τρέχει σε περισσότερα από ένα μηχανήματα. Για παράδειγμα, ας υποθέσουμε ότι το κύριο πρόγραμμα τρέχει αλλού (σε μηχάνημα που λειτουργεί ως πελάτης) και ότι βλέπουμε τα αποτελέσματα της σχεδίασης σε άλλο σταθμό εργασίας ή τερματικό (server), το οποίο συνδέεται μέσω δικτύου με τον πελάτη. Στην περίπτωση αυτή, μπορεί να είναι σημαντικά αναποτελεσματική η αποστολή κάθε εντολής ξεχωριστά μέσω του δικτύου, δεδομένου ότι ο φόρτος του δικτύου συνδέεται συχνά με κάθε ξεχωριστή αποστολή. Συνήθως, ο πελάτης συγκεντρώνει ένα σύνολο εντολών σε ένα ενιαίο πακέτο δικτύου πριν από την αποστολή. Δυστυχώς, ο πελάτης συνήθως δεν μπορεί να γνωρίζει πότε το πρόγραμμα σχεδίασης τελειώσει της σχεδίαση ενός πλαισίου ή μιας σκηνής. Στη χειρότερη περίπτωση, θα περιμένει για πάντα ώστε αρκετές επιπλέον εντολές σχεδίασης να γεμίσουν ένα πακέτο, και ποτέ δεν βλέπουμε τις ολοκληρωμένες σχεδιάσεις.

Για το λόγο αυτό, η OpenGL παρέχει την εντολή glFlush (), η οποία αναγκάζει τον πελάτη να στείλει το δικτυακό πακέτο ακόμα και αν δεν είναι πλήρες. Σε περίπτωση που δεν υπάρχει δίκτυο και όλες οι εντολές που εκτελούνται αμέσως στον server, η εντολή glFlush () δεν έχει αποτέλεσμα. Ωστόσο, εάν γράφουμε ένα πρόγραμμα που θέλουμε να λειτουργεί σωστά σε περίπτωση που έχουμε δίκτυο, πρέπει να περιλάβουμε την εντολή glFlush () στο τέλος κάθε πλαισίου ή σκηνής. Σημειώστε ότι η glFlush () δεν περιμένει τη σχεδίαση να ολοκληρωθεί - απλά αναγκάζει την διαδικασία της σχεδίασης να εκτελεστεί, εξασφαλίζοντας έτσι ότι όλες οι προηγούμενες εντολές εκτελούνται σε πεπερασμένο

χρόνο, ακόμη και αν δεν εκτελούνται περαιτέρω εντολές απόδοσης. Υπάρχουν και άλλες περιπτώσεις όπου glFlush () είναι χρήσιμη:

● Εφαρμογές οι οποίες χτίζουν τις εικόνες στην μνήμη του συστήματος και δεν θέλουν να ανανεώνουν συνεχώς την οθόνη.

● Υλοποιήσεις που συγκεντρώνουν σύνολα από εντολές σχεδίασης για να αποσβέσουν τα έξοδα αρχικοποίησης. Το παράδειγμα δικτύου μεταφοράς που αναφέραμε παραπάνω είναι ένα παράδειγμα τέτοιας υλοποίησης.

Επομένως, η εντολή

```
void glFlush (void);
```

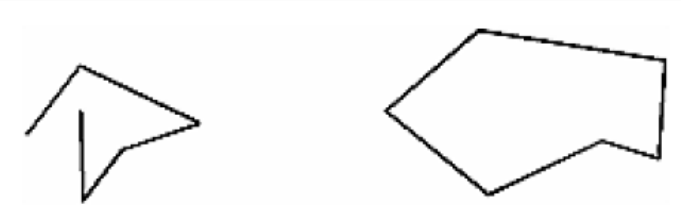
αναγκάζει τις εντολές OpenGL που έχουν οριστεί προηγουμένως να ξεκινήσουν να εκτελούνται, εξασφαλίζοντας έτσι ότι θα εκτελεστούν σε πεπερασμένο χρόνο.

Τα βασικά σχήματα στην OpenGL περιλαμβάνουν σημεία, γραμμές και πολύγωνα. Οι μαθηματικοί ορισμοί των εννοιών αυτών δεν διαφέρουν πολύ από την έννοια που έχουν στην χρήση τους από την OpenGL.

Σημεία: Ένα σημείο αναπαριστάται από ένα σύνολο αριθμών κινητής υποδιαστολής οι οποίοι ονομάζονται συντεταγμένες. Όλοι οι εσωτερικοί υπολογισμοί γίνονται με δεδομένο ότι οι

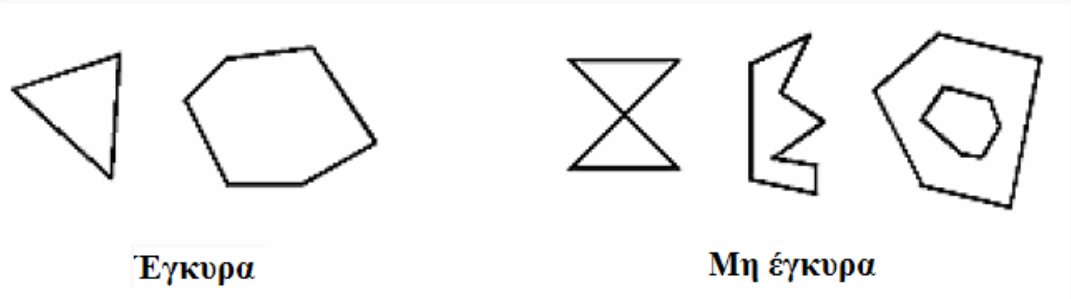
συντεταγμένες είναι τρισδιάστατες. Στις συντεταγμένες που καθορίζονται από το χρήστη ως δύο διαστάσεων (δηλαδή, με μόνο x και y τιμές) δίδεται από την OpenGL μια τιμή z ίση με το μηδέν.

Γραμμές: Στην OpenGL, ο όρος “γραμμή” αναφέρεται σε ένα τμήμα γραμμής, δηλαδή δεν ισοδυναμεί με την μαθηματική έννοια η οποία ορίζει την γραμμή εκτείνοντας τις δύο άκρες της στο άπειρο. Υπάρχουν εύκολοι τρόποι να οριστεί μια συνδεδεμένη σειρά από τμήματα γραμμών, ή ακόμα και μια κλειστή σειρά από γραμμές (εικόνα 2.12). Σε κάθε περίπτωση, μια γραμμή ορίζεται από δύο σημεία που αποτελούν τις άκρες της γραμμής.



Εικόνα 2.12: Δύο συνδεδεμένες σειρές γραμμών

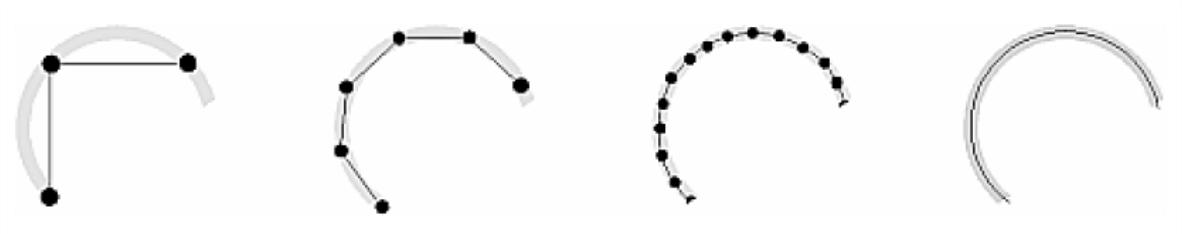
Πολύγωνα: Τα πολύγωνα είναι περιοχές που περικλείονται από κλειστούς βρόχους μονών γραμμών, όπου τα τμήματα γραμμών ορίζονται από τις συντεταγμένες των ακραίων σημείων τους. Τα πολύγωνα συνήθως σχεδιάζονται με τα εικονοστοιχεία στο εσωτερικό τους γεμισμένα, αλλά μπορούμε επίσης να τα σχεδιάσουμε ως ένα σύνολο σημείων.



Εικόνα 2.13: Έγκυρα και μη έγκυρα πολύγωνα

Ο λόγος που η OpenGL έχει περιορισμούς σε τύπους πολυγώνων είναι ότι είναι πολύ πιο απλό για το υλισμικό σχεδίασης πολυγώνων να αποδώσει αυτήν την περιορισμένη κατηγορία πολυγώνων. Τα πολύγωνα αυτής της κατηγορίας απλά μπορούν να σχεδιαστούν πολύ πιο γρήγορα. Οι μη έγκυρες περιπτώσεις είναι δύσκολο να ανιχνευθούν γρήγορα. Έτσι, για χάρη της μέγιστης απόδοσης, η OpenGL επιτρέπει μόνο απλά πολύγωνα.

Κάθε ομαλά καμπυλωμένη γραμμή ή επιφάνεια, μπορεί να προσεγγισθεί - σε κάποιο βαθμό ακριβείας - με πολύ μικρά τμήματα γραμμών ή μικρές πολυγωνικές περιοχές. Έτσι, υποδιαιρώντας καμπυλωτές γραμμές και επιφάνειες επαρκώς και, στη συνέχεια, προσεγγίζοντας τες με ευθύγραμμα τμήματα ή επίπεδα πολύγωνα τις κάνει να φαίνονται κυρτές (βλ. εικόνα 2.14). Αυτό μπορεί να γίνει πιο κατανοητό, αν φανταστούμε ότι υποδιαιρούμε μια καμπύλη μέχρι κάθε τμήμα γραμμής ή κάθε πολύγωνο να είναι τόσο μικρό όσο λιγότερο από το μέγεθος ενός εικονοστοιχείου στην οθόνη.



Εικόνα 2.14: Προσέγγιση καμπυλών

Με την OpenGL, όλα τα γεωμετρικά αντικείμενα τελικά περιγράφονται ως ένα διατεταγμένο σύνολο συντεταγμένων. Μπορούμε να χρησιμοποιήσουμε την εντολή glVertex * () εντολή για να καθορίσουμε μια κορυφή.

```
glVertex void {234} {sifd} [v] (TYPEcoords);
```

Η εντολή αυτή καθορίζει μια κορυφή (ένα σύνολο συντεταγμένων) για χρήση κατά την περιγραφή ενός γεωμετρικού αντικειμένου. Μπορούμε να δώσουμε μέχρι και τέσσερις συντεταγμένες (x, y, z, w) για μία συγκεκριμένη κορυφή ή το λιγότερο δύο (x, y) επιλέγοντας την κατάλληλη παραλλαγή της εντολής. Εάν χρησιμοποιήσουμε την παραλλαγή που δεν διευκρινίζει ρητά την τιμή z ή w, η τιμή z θα θεωρηθεί ίση με τη τιμή 0 και η τιμή w ίση με 1. Κλήσεις της μεθόδου glVertex * () είναι αποδεκτές μόνο μεταξύ του ζεύγους των εντολών glBegin () και glEnd ().

Τώρα που ξέρουμε τον τρόπο καθορισμού των κορυφών, θα δούμε πώς μπορούμε να δημιουργήσουμε ένα σύνολο σημείων, μια γραμμή ή ένα πολύγωνο από κάποιες κορυφές. Για να γίνει αυτό, ομαδοποιούμε ένα σύνολο κορυφών μεταξύ του ζεύγους εντολών glBegin () και glEnd (). Το όρισμα που δίδεται στην εντολή glBegin () καθορίζει τι είδους γεωμετρικά σχήματα κατασκευαστούν από τις κορυφές αυτές. Για παράδειγμα με τις παρακάτω εντολές, σχεδιάζεται το πολύγωνο της εικόνας 2.15.

```
glBegin(GL_POLYGON);
  glVertex2f(0.0, 0.0);
  glVertex2f(0.0, 3.0);
  glVertex2f(4.0, 3.0);
  glVertex2f(6.0, 1.5);
  glVertex2f(4.0, 0.0);
glEnd();
```

Εικόνα 2.15: Σχεδίαση ενός πολυγώνου ή ενός συνόλου σημείων

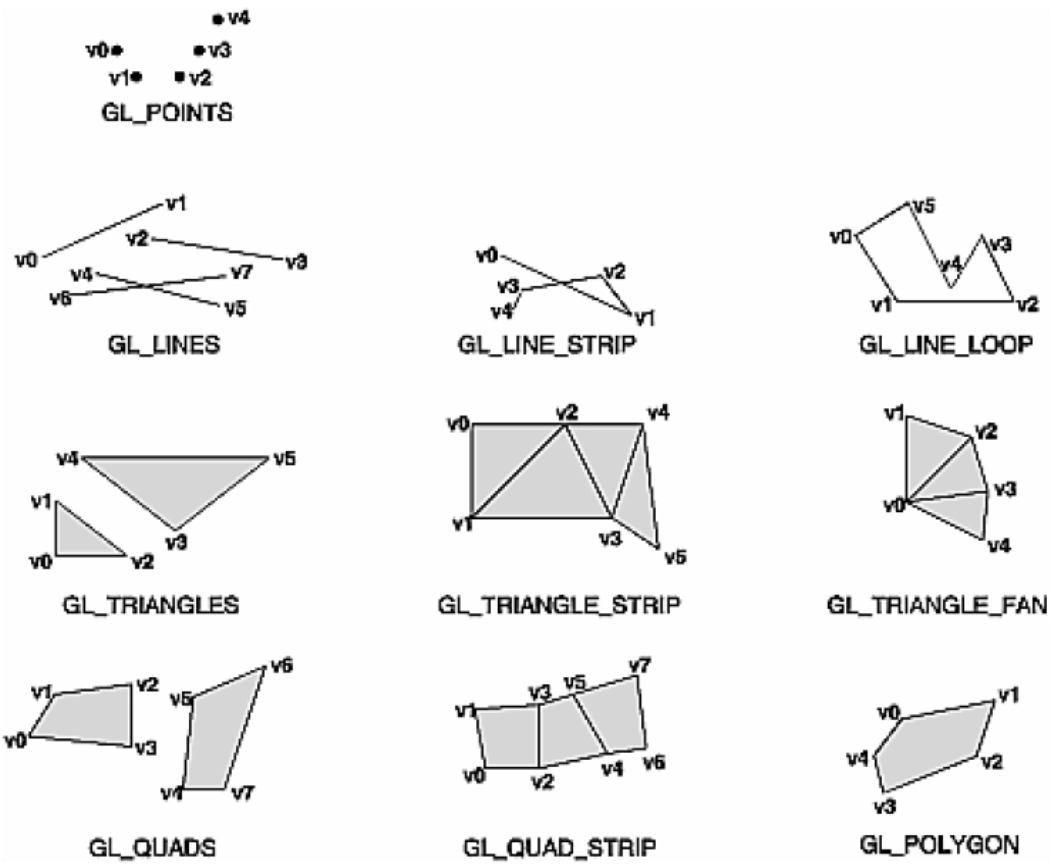
Εάν είχαμε χρησιμοποιήσει GL_POINTS αντί για πολύγωνα (GL_POLYGON), το αποτέλεσμα θα ήταν η σχεδίαση του συνόλου των σημείων που βλέπουμε στην παραπάνω εικόνα στο δεξιό μέρος.

Τιμή	Σημασία
GL_POINTS	αυτόνομα σημεία
GL_LINES	ζεύγη απο κορυφές ερμηνευόμενα σαν ξεχωριστά τμήματα γραμμών
GL_LINE_STRIP	σειρές απο συνδεδεμένα τμήματα γραμμών
GL_LINE_LOOP	ίδια με παραπάνω, με ένα τμήμα να προστίθεται μεταξύ της πρώτης και της τελευταίας κορυφής
GL_TRIANGLES	τριάδες απο κορυφές ερμηνευόμενες ως τρίγωνα
GL_TRIANGLE_STRIP	συνδεδεμένη σειρά απο τρίγωνα
GL_TRIANGLE_FAN	συνδεδεμένο σύνολο απο τρίγωνα
GL_QUADS	τετράδες απο κορυφές ερμηνευόμενες σαν τετράπλευρα πολύγωνα
GL_QUAD_STRIP	συνδεδεμένη λωρίδα απο τετράπλευρα
GL_POLYGON	όρια ενός απλού κλειστού πολυγώνου

Στον παραπάνω πίνακα βλέπουμε το σύνολο τιμών που μπορεί να πάρει το όρισμα της εντολής glBegin (). Στην εικόνα 2.16 φαίνονται τα σχήματα που σχεδιάζονται με κάθε χρήση των παραπάνω τιμών. Οι πιο σημαντικές πληροφορίες σχετικά με τις κορυφές είναι συντεταγμένες τους, οι οποίες καθορίζονται από την εντολή glVertex (). Μπορούμε επίσης να ορίσουμε δεδομένα που έχουν σχέση με κάποια κορυφή - ένα χρώμα, ένα κάθετο διάνυσμα, συντεταγμένες υφής, ή οποιοδήποτε συνδυασμό αυτών - με τη βοήθεια ειδικών εντολών. Επιπλέον, μερικές άλλες εντολές μπορούν να χρησιμοποιηθούν μεταξύ των εντολών glBegin () και glEnd (). Στον πίνακα παρακάτω βλέπουμε την πλήρη λίστα αυτών των εντολών.

Εντολή	Σκοπός εντολής
glVertex*()	set vertex coordinates
glColor*()	set current color
glIndex*()	set current color index
glNormal*()	set normal vector coordinates
glTexCoord*()	set texture coordinates
glEdgeFlag*()	control drawing of edges
glMaterial*()	set material properties
glArrayElement()	extract vertex array data
glEvalCoord*(), glEvalPoint*()	generate coordinates
glCallList(), glCallLists()	execute display list(s)

Δεν υπάρχει άλλη εντολή OpenGL που να ισχύει μεταξύ των εντολών glBegin () και glEnd (), και αν επιχειρήσουμε την κλήση μιας άλλης μεθόδου OpenGL, ως επί το πλείστον θα δημιουργηθεί σφάλμα. Μερικές εντολές πινάκων κορυφών , όπως η glEnableClientState () και η glVertexPointer (), όταν κληθούν μεταξύ των glBegin () και glEnd (), έχουν απροσδιόριστη συμπεριφορά, αλλά δεν δημιουργούν απαραιτήτως ένα σφάλμα. (Επίσης, ρουτίνες που σχετίζονται με την OpenGL, όπως οι glx*() ρουτίνες έχουν απροσδιόριστη συμπεριφορά μεταξύ των glBegin () και glEnd ()). Οι περιπτώσεις αυτές θα πρέπει να αποφεύγονται, καθώς ο εντοπισμός των σφαλμάτων που δημιουργούνται μπορεί να είναι πολύ δύσκολος.



Εικόνα 2.16: Βασικοί γεωμετρικοί τύποι

2.3. Θέαση

Ορισμένες φορές θα πρέπει να αποφασίσουμε πώς θα τοποθετήσουμε τα μοντέλα στη σκηνή, και πρέπει να επιλέξουμε ένα πλεονεκτικό σημείο από το οποίο θα βλέπουμε τη σκηνή. Μπορούμε να χρησιμοποιήσετε την προεπιλεγμένη θέση, αλλά πιθανότατα να θέλουμε να προσδιορίσουμε μια άλλη.

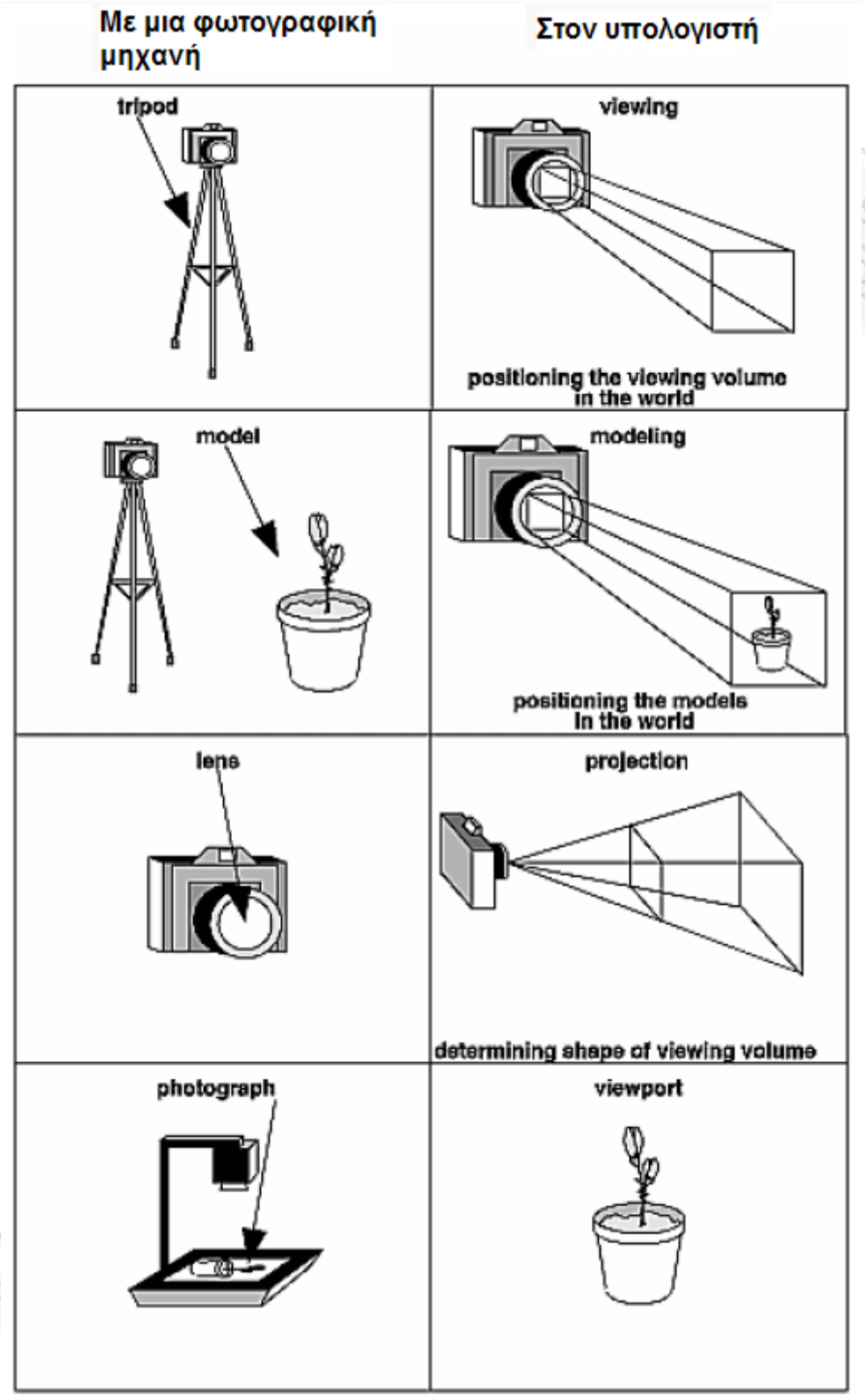
Η διαδικασία μετασχηματισμού για να παραχθεί η επιθυμητή σκηνή για προβολή είναι ανάλογη με τη λήψη μιας φωτογραφίας με μια φωτογραφική μηχανή. Όπως φαίνεται στην εικόνα 2.17, τα βήματα με μια φωτογραφική μηχανή (ή έναν υπολογιστή) μπορεί να είναι τα ακόλουθα:

1. Ρυθμίζουμε το τρίποδο και τοποθετούμε τη φωτογραφική μηχανή να βλέπει προς την σκηνή (μετασχηματισμός θέασης).

2. Τακτοποιούμε τη σκηνή ώστε να φωτογραφηθεί στην επιθυμητή σύνθεση (μετασχηματισμός μοντέλων).

3. Επιλέγουμε τον φακό της φωτογραφικής μηχανής ή ρυθμίζουμε το ζουμ (μετασχηματισμός προβολής).

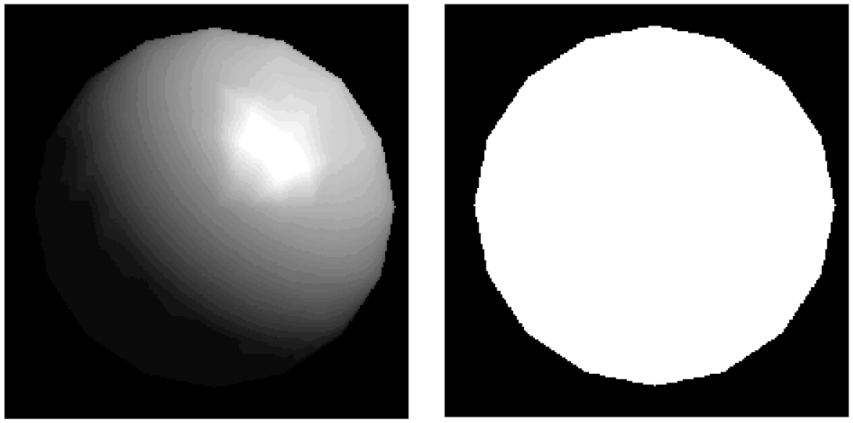
4. Καθορίζουμε πόσο μεγάλη θέλουμε να είναι η τελική φωτογραφία, για παράδειγμα, μπορεί να την θέλουμε διευρυμένη (μετασχηματισμός θύρας θέασης).



Εικόνα 2.17: Βήματα λήψης φωτογραφίας με κάμερα και στον υπολογιστή

2.4. Φωτισμός

Όπως προανέφερα και σε προηγούμενο κεφάλαιο, η OpenGL υπολογίζει το χρώμα του κάθε εικονοστοιχείου σε μια τελική σκηνή το οποίο αποθηκεύεται στον ενταμιευτή πλαισίου (framebuffer). Μέρος αυτού του υπολογισμού εξαρτάται από τον φωτισμό που χρησιμοποιείται στη σκηνή και στο πώς τα αντικείμενα στη σκηνή αντανακλούν ή απορροφούν το φως. Στην εικόνα 2.18 παρακάτω βλέπουμε την επίδραση του φωτός σε μία σφαίρα.



Εικόνα 2.18: Μια φωτισμένη και μη φωτισμένη σφαίρα

Υπάρχουν τρία είδη φωτισμού που μπορούμε να ορίσουμε. Ο φωτισμός περιβάλλοντος (ambient light) είναι το φως που έχει διασκορπιστεί τόσο πολύ από το περιβάλλον, που η κατεύθυνση του είναι αδύνατο να προσδιοριστεί - φαίνεται να προέρχεται από όλες τις κατευθύνσεις. Το διάχυτο φως (diffuse light) έρχεται από μια κατεύθυνση, έτσι είναι λαμπρότερο αν χτυπάει απευθείας σε μια επιφάνεια. Τέλος, το κατοπτρικό φως (specular light) προέρχεται από μια συγκεκριμένη κατεύθυνση, και τείνει να αναπηδήσει από την επιφάνεια σε μια προτιμητέα κατεύθυνση. Αυτά είναι τα βήματα που απαιτούνται για να προσθέσουμε φωτισμό σε μια σκηνή.

1. Ορίζουμε κάθετα διανύσματα για κάθε κορυφή όλων των αντικειμένων. Αυτά τα διανύσματα προσδιορίζουν τον προσανατολισμό του αντικειμένου σε σχέση με τις πηγές φωτός.

2. Δημιουργούμε, επιλέγουμε και τοποθετούμε μία ή περισσότερες φωτεινές πηγές.

3. Δημιουργούμε και επιλέγουμε ένα μοντέλο φωτισμού, το οποίο καθορίζει το επίπεδο του συνολικού φωτισμού περιβάλλοντος.

4. Ορίζουμε τις ιδιότητες των υλικών για κάθε αντικείμενο στην σκηνή.

Ορίζουμε τα κάθετα διανύσματα χρησιμοποιώντας την εντολή glNormal*() πριν από τον ορισμό κάθε κορυφής. Την πηγή φωτός την καθορίζουμε με την εντολή glLightfv(). Η εντολή αυτή παίρνει τρία ορίσματα. Το πρώτο είναι ο τύπος του φωτός και τα υπόλοιπα δύο ορίζουν την τοποθεσία της πηγής.

Για παράδειγμα αν θέλουμε να ορίσουμε μια πηγή φωτός στην θέση light_position (οι συντεταγμένες της θέσης) θα χρησιμοποιήσουμε την εντολή:

```
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

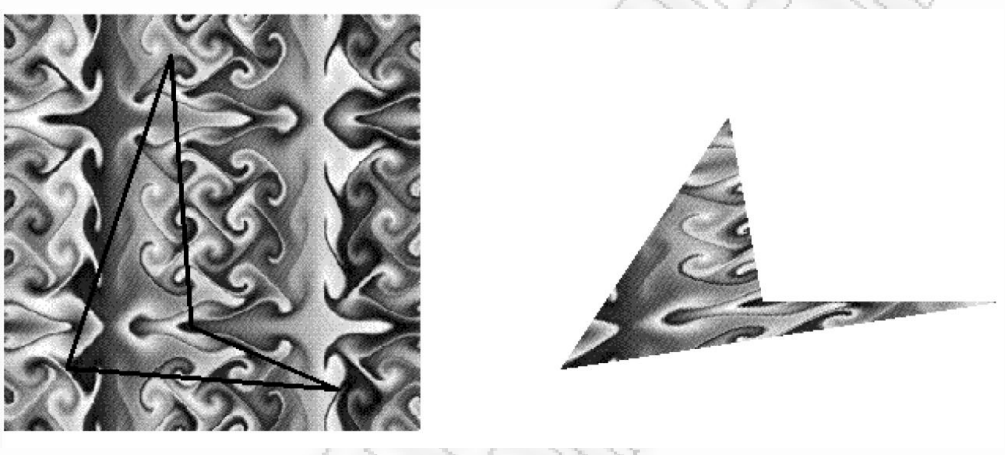
Εφ΄ όσον δημιουργήσουμε την πηγή φωτός (μία ή περισσότερες) θα πρέπει να την ενεργοποιήσουμε με την εντολή glEnable(). Επίσης θα πρέπει να ενεργοποιήσουμε την τιμή GL_LIGHTING για να γίνουν οι απαραίτητοι υπολογισμοί φωτισμού:

```
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
```

Η εντολή glLightModel*() περιγράφει τις παραμέτρους ενός μοντέλου φωτισμού. Με την εντολή αυτή ορίζουμε τα είδη φωτός (ambient, diffuse, specular). Παράλληλα, καθορίζουμε εάν ο θεατής της σκηνής θα πρέπει να θεωρηθεί ότι βρίσκεται σε συγκεκριμένο σημείο στη σκηνή ή όχι, και αν οι υπολογισμοί φωτισμός θα πρέπει να γίνουν με διαφορετικό τρόπο για τις εμπρός και τις πίσω επιφάνειες των αντικειμένων στη σκηνή. Τις ιδιότητες των υλικών για κάθε αντικείμενο τις καθορίζουμε με την εντολή glMaterialfv().

2.5. Χαρτογράφηση υφής

Η χαρτογράφηση υφής μας επιτρέπει να κολλήσουμε μια εικόνα π.χ. από έναν τοίχο από τούβλα (που, ίσως, προέρχεται από τη σάρωση μιας φωτογραφίας ενός πραγματικού τοίχου) σε ένα πολύγωνο και να σχεδιάσουμε αυτό που δείχνει η εικόνα (π.χ. τον τοίχο) ως ένα ενιαίο πολύγωνο. Η χαρτογράφηση υφής διασφαλίζει ότι όλες οι απαραίτητες ενέργειες γίνονται καθώς το πολύγωνο μετασχηματίζεται και σχεδιάζεται. Για παράδειγμα, στην σχεδίαση ενός τοίχου από τούβλα, τα τούβλα πρέπει να εμφανίζονται μικρότερα όταν ο τείχος απομακρύνεται από την τρέχουσα οπτική γωνία.



Εικόνα 2.19: Διαδικασία χαρτογράφησης υφής

Στην εικόνα 2.19 βλέπουμε την διαδικασία χαρτογράφησης υφής. Στα αριστερά βλέπουμε την εικόνα υφής και στα δεξιά το αντικείμενο με κολλημένη την εικόνα πάνω του. Για να χρησιμοποιήσουμε την χαρτογράφηση υφής, εκτελούμε αυτά τα βήματα:

1. Δημιουργούμε ένα αντικείμενο υφής και καθορίζουμε μια υφή για αυτό το αντικείμενο.
2. Καθορίζουμε τον τρόπο με τον οποίο η υφή θα εφαρμόζεται σε κάθε εικονοστοιχείο.
3. Ενεργοποιούμε την χαρτογράφηση υφής.
4. Σχεδιάζουμε τη σκηνή, ορίζοντας την υφή και τις συντεταγμένες.

Περισσότερα για τον τρόπο που γίνονται τα παραπάνω μπορείτε να δείτε στο κεφάλαιο σχετικά με την σχεδίαση του περιβάλλοντος του παιχνιδιού, δηλαδή την σχεδίαση του ουρανού και του εδάφους του παιχνιδιού.

3. Ανάπτυξη παιχνιδιού

3.1. Σενάριο παιχνιδιού

Το παιχνίδι το οποίο υλοποίησα και θα αναλύσουμε είναι ένα τρισδιάστατο παιχνίδι σκοποβολής τρίτου προσώπου (3rd-person shooting game), στο οποίο υπάρχει η δυνατότητα δύο παίκτες να παίξουν μεταξύ τους μέσω Διαδικτύου. Ο κάθε παίκτης χειρίζεται έναν τοξότη και προσπαθεί να πετύχει με βέλη τον τοξότη του άλλου παίκτη. Όποιος πετύχει πρώτος τον τοξότη του άλλου παίκτη πέντε φορές νικάει. Στα επόμενα κεφάλαια θα αναλύσουμε τα βήματα τα οποία ακολούθησα για την ανάπτυξη του συγκεκριμένου παιχνιδιού.

3.2. Περιβάλλον

3.2.1. Εισαγωγή

Αν χωρίζαμε την υλοποίηση του παιχνιδιού σε στάδια, το πρώτο απ' όλα θα ήταν να ορίσουμε το περιβάλλον του παιχνιδιού. Με τον όρο περιβάλλον εννοούμε την περιοχή στην οποία θα εκτυλίσσεται το παιχνίδι. Αυτό περιλαμβάνει την σχεδίαση του εδάφους και του ουρανού με ρεαλιστικό τρόπο ώστε ο παίκτης να αντιλαμβάνεται ότι ο χαρακτήρας του κινείται σε έναν πραγματικό κόσμο. Σε αυτό το κεφάλαιο λοιπόν θα αναλύσουμε έναν απλό τρόπο με τον οποίο το πετυχαίνουμε αυτό.

Το πρώτο βήμα είναι να σχεδιάσουμε έναν κύβο. Η κάτω επιφάνεια του κύβου προορίζεται για το έδαφος, ενώ οι υπόλοιπες για τον ουρανό. Με την χρήση εικόνων πετυχαίνουμε σε έναν βαθμό την υφή την οποία θέλουμε. Μια εικόνα γρασιδιού προσδίδει σε μια επιφάνεια την αντίληψη ότι πρόκειται για έδαφος με γρασίδι. Μια εικόνα συννεφιασμένου ουρανού στις τρείς πλάγιες επιφάνειες του κύβου, μας κάνει να πιστεύουμε, όταν η κάμερα βρίσκεται εντός του κύβου, ότι γύρω μας υπάρχει ουρανός. Τέλος αυτό που πρέπει να προσέξουμε είναι οι εικόνες του ουρανού να έχουν μια συνέχεια μεταξύ τους ώστε να μην μαρτυρούν το γεγονός ότι βρισκόμαστε μέσα σε έναν κύβο.

Πριν από κάθε σχεδίαση πρέπει να φορτώσουμε τα texture σε κάποιες μεταβλητές για να μπορούμε να τα χρησιμοποιήσουμε έπειτα. Αυτό το κάνουμε στην μέθοδο init(GLAutoDrawable drawable) δηλαδή στην μέθοδο αρχικοποίησης του παιχνιδιού η οποία εκτελείται μια φορά πριν από την μέθοδο display(GLAutoDrawable drawable) η οποία εκτελείται συνεχώς και σχεδιάζει τα σχήματα που θέλουμε με τα texture που έχουμε ορίσει. Για τον σκοπό αυτό έχω υλοποιήσει την μέθοδο loadTexture( ) που βλέπουμε παρακάτω. Παίρνει ως όρισμα το όνομα του αρχείου που περιέχει την εικόνα που θέλουμε να χρησιμοποιήσουμε ως texture και επιστρέφει ένα αντικείμενο τύπου Texture.

```
private Texture loadTexture(String filename)
{
     Texture texture = null;
     try
     {
          InputStream stream =
          getClass().getResourceAsStream(filename);
          TextureData data = TextureIO.newTextureData(stream, true,
          TextureIO.JPG);
          texture = TextureIO.newTexture(data);
     }
     catch (IOException exc)
     {
          exc.printStackTrace();
     }
     return texture;
```

```
}
```

Έτσι λοιπόν χρησιμοποιούμε την παραπάνω μέθοδο κατά την αρχικοποίηση του παιχνιδιού για να φορτώσουμε τα texture που θέλουμε ως εξής:

```
floorTexture1 = ("grass.jpg");
floorTexture2 = ("pavement.jpg");
skyboxTexture1 = loadTexture("sky1.jpg");
skyboxTexture2 = loadTexture("sky2.jpg");
skyboxTexture3 = loadTexture("sky3.jpg");
skyboxTexture4 = loadTexture("sky4.jpg");
```

Εννοείται ότι οι εικόνες μας πρέπει να είναι αρχεία τύπου JPG και όχι κάτι άλλο για να διαβαστούν επιτυχώς. Τέλος επειδή η εικόνα του γρασιδιού και της πέτρας επαναλαμβάνονται πολλές φορές πάνω στην ίδια επιφάνεια πρέπει να εκτελέσουμε τις παρακάτω εντολές.

```
// repeat the floor texture in every direction
floorTexture1.setTexParameteri(GL.GL_TEXTURE_WRAP_S, GL.GL_REPEAT);
floorTexture1.setTexParameteri(GL.GL_TEXTURE_WRAP_T, GL.GL_REPEAT);
floorTexture2.setTexParameteri(GL.GL_TEXTURE_WRAP_S, GL.GL_REPEAT);
floorTexture2.setTexParameteri(GL.GL_TEXTURE_WRAP_T, GL.GL_REPEAT);
```

Αυτά σημειώνω άλλη μια φορά ότι πρέπει να εκτελεστούν μια φορά κατά την αρχικοποίηση του παιχνιδιού και όχι στην μέθοδο display η οποία επαναλαμβάνεται συνεχώς αλλιώς θα δημιουργηθεί σφάλμα και η εκτέλεση του παιχνιδιού δεν θα είναι ομαλή.

3.2.2. Σχεδίαση εδάφους

Η σχεδίαση του εδάφους στο παιχνίδι γίνεται με την μέθοδο buildFloor( ) την οποία βλέπουμε παρακάτω. Σαν ορίσματα παίρνει εκτός από το αντικείμενο τύπου GL το οποίο χρειαζόμαστε για να σχεδιάσουμε την τρισδιάστατη επιφάνεια, έναν αριθμό τύπου float με όνομα size ο οποίος αναπαριστά το μέγεθος του εδάφους δηλαδή της επιφάνειας που θα σχεδιάσουμε και θα τοποθετήσουμε πάνω σε αυτήν την εικόνα με γρασίδι. Την καλούμε μέσα στην μέθοδο display.

```
private void buildFloor(GL gl, float size)
{
      gl.glEnable(GL.GL_TEXTURE_2D);
      floorTexture1.bind();
      TextureCoords tc = floorTexture1.getImageTexCoords();
      float left = tc.left();
      float right = tc.right();
      float bottom = tc.bottom();
      float top = tc.top();
      //build floor at y=-10
      gl.glTranslatef(0.0f, -10.0f, 0.0f);
```

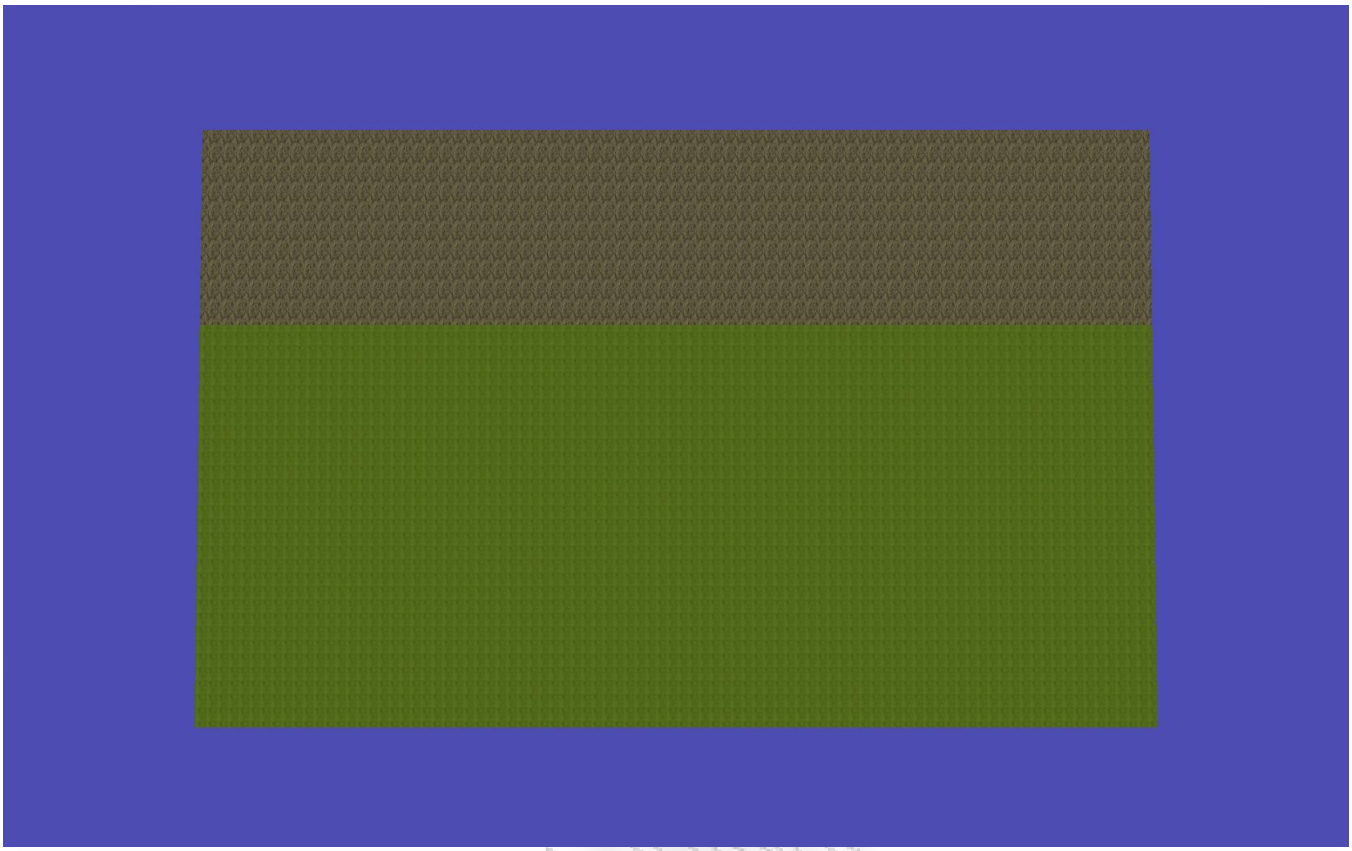
```
        //draw floor
        gl.glBegin(GL.GL_QUADS);
        gl.glTexCoord2f(left, 30*bottom); gl.glVertex3f(-size, 0.0f,
        size);//bottom left

        gl.glTexCoord2f(30*right, 30*bottom);gl.glVertex3f(size, 0.0f,
        size);//bottom right
        gl.glTexCoord2f(30*right, top);gl.glVertex3f(size, 0.0f, -
        size+330);//top right
        gl.glTexCoord2f(left, top);gl.glVertex3f(-size, 0.0f, -size+330);//top
        left
        gl.glEnd();
        floorTexture2.bind();
        tc = floorTexture2.getImageTexCoords();
        left = tc.left();
        right = tc.right();
        bottom = tc.bottom();
        top = tc.top();
        //draw floor

        gl.glBegin(GL.GL_QUADS);
        gl.glTexCoord2f(left, 10*bottom); gl.glVertex3f(-size, 0.0f, size-
        670);//bottom left
        gl.glTexCoord2f(30*right, 10*bottom);gl.glVertex3f(size, 0.0f, size-
        670);//bottom right
        gl.glTexCoord2f(30*right, top);gl.glVertex3f(size, 0.0f, -size);
        gl.glTexCoord2f(left, top);gl.glVertex3f(-size, 0.0f, -size);
        gl.glEnd();
        gl.glDisable(GL.GL_TEXTURE_2D);
}
```

Την τετράγωνη επιφάνεια που προορίζεται για το έδαφος την σχεδιάζουμε χρησιμοποιώντας QUADS (τετράγωνα) με την εντολή gl.glBegin(GL.GL_QUADS) και ορίζοντας τα τέσσερα σημεία τα οποία θα είναι οι ακμές του τετραγώνου με κλήση της μεθόδου gl.glVertex3f( ). Παράλληλα με την σχεδίαση του τετραγώνου, ορίζουμε και τα σημεία της εικόνας τα οποία αντιστοιχούν στις ακμές αυτές με κλήση της μεθόδου gl.glTexCoord2f( ). Πρέπει όμως προηγουμένως να έχουμε ορίσει πια εικόνα θέλουμε να χρησιμοποιήσουμε ως texture και να την έχουμε επιλέξει με την μέθοδο bind( ).

Στην προκειμένη περίπτωση χρησιμοποιούμε δύο textures, ένα γρασιδιού και ένα πέτρας. Επιλέγουμε αυτό που θέλουμε να χρησιμοποιήσουμε με την μέθοδο bind( ) και σχεδιάζουμε το έδαφος σε δύο στάδια. Το πρώτο είναι το κομμάτι που θέλουμε να έχει γρασίδι και το δεύτερο το κομμάτι που θέλουμε να έχει πέτρα. Επίσης η εικόνα γρασιδιού επαναλαμβάνεται τριάντα φορές οριζοντίως και τριάντα φορές καθέτως ενώ η εικόνα πέτρας τριάντα φορές οριζοντίως και δέκα φορές καθέτως. Αυτό το πετυχαίνουμε πολλαπλασιάζοντας την κατάλληλη συντεταγμένη του texture με τον αριθμό που θέλουμε στην εντολή gl.glTextCoord2f( ). Τέλος πρέπει να σημειώσουμε ότι σχεδιάζουμε το έδαφος δέκα μονάδες κάτω από τον άξονα y (gl.glTranslatef(0.0f, -10.0f, 0.0f);) έτσι ώστε να βλέπουμε το έδαφος λίγο παρακάτω από την κάμερα και όχι στο ίδιο επίπεδο.

Παρακάτω βλέπουμε το αποτέλεσμα της σχεδίασης του εδάφους. Στην εικόνα 3.1 η κάμερα βρίσκεται στο κέντρο των αξόνων και κοιτάει παράλληλα στους άξονες x και z ενώ στην εικόνα 3.2 η κάμερα βρίσκεται πιο ψηλά και κοιτάει προς του άξονες x και z.

Εικόνα 3.1: Σχεδίαση εδάφους με την κάμερα να κοιτάει παράλληλα στους άξονες x και z



Εικόνα 3.2: Σχεδίαση εδάφους με την κάμερα να κοιτάει προς τους άξονες x και z

3.2.3. Σχεδίαση ουρανού

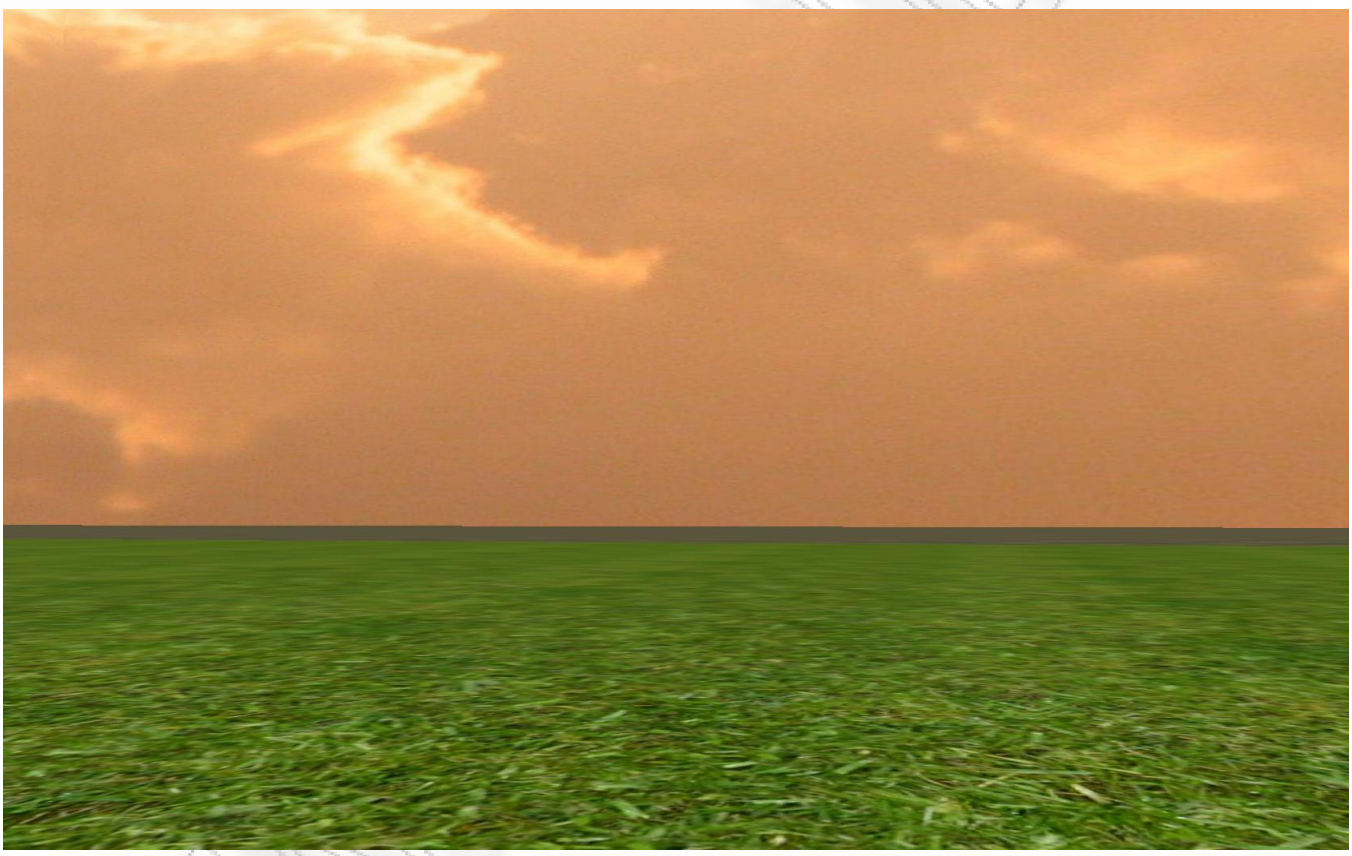
Με τον ίδιο τρόπο σχεδιάζουμε τον ουρανό. Αυτό γίνεται με την μέθοδο buildSkyBox( ) την οποία βλέπουμε παρακάτω. Όπως και η μέθοδος για το έδαφος παίρνει μία την ίδια τιμή size έτσι ώστε το τελικό αποτέλεσμα (έδαφος και ουρανός) να είναι ένας κύβος. Εδώ σχεδιάζουμε τέσσερα τετράγωνα (τις πλάγιες επιφάνειες του κύβου) και επικολλούμε πάνω τους τέσσερεις εικόνες οι οποίες δείχνουν έναν ουρανό. Επιλέγουμε όπως πριν το texture που θέλουμε και σχεδιάζουμε τα τετράγωνα ορίζοντας παράλληλα τις συντεταγμένες των εικόνων που θέλουμε να βρίσκονται σε αυτά τα σημεία. Αυτή τη φορά η εικόνα δεν επαναλαμβάνεται πολλές φορές. Επαφίεται στην δική μας επιλογή εικόνας, πόσο καλό θα είναι το αποτέλεσμα. Όπως είπα και προηγουμένως δεν πρέπει να φαίνονται οι γωνίες του κύβου και γι' αυτό το λόγο οι τέσσερεις εικόνες στην σειρά πρέπει να πετυχαίνουν αυτό το αποτέλεσμα.

```
private void buildSkybox(GL gl, float size)
{
      // Load the texture.
      gl.glEnable(GL.GL_TEXTURE_2D);
     skyboxTexture1.bind();

      //build front wall
```

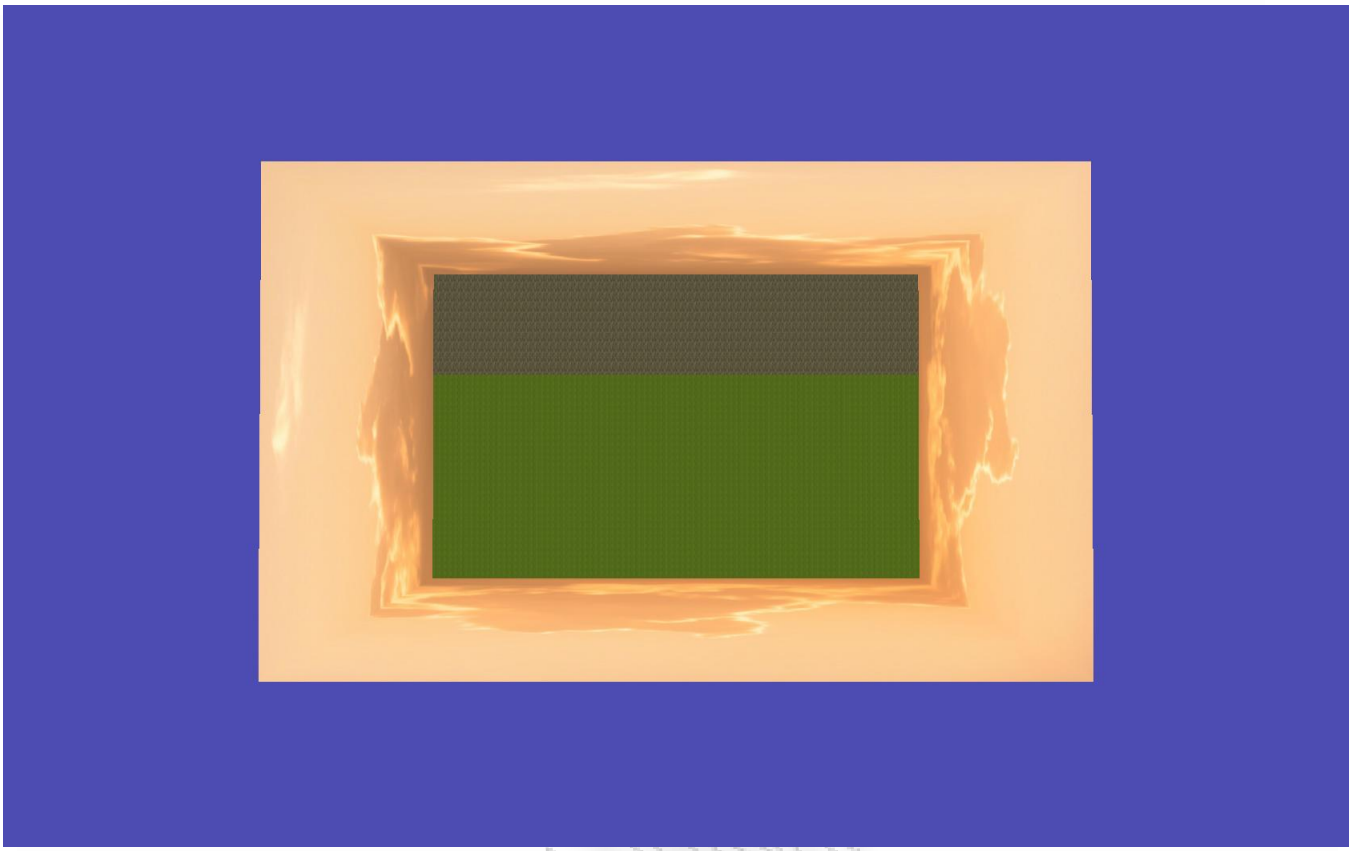
```
        gl.glTranslatef(0.0f, size, -size);
        gl.glBegin(GL.GL_QUADS);
        gl.glTexCoord2f(0.0f, 1.0f);gl.glVertex3f(-size, -size, 0.0f);
        gl.glTexCoord2f(1.0f, 1.0f);gl.glVertex3f(size, -size, 0.0f);
        gl.glTexCoord2f(1.0f, 0.0f);gl.glVertex3f(size, size, 0.0f);
        gl.glTexCoord2f(0.0f, 0.0f); gl.glVertex3f(-size ,size, 0.0f);
        gl.glEnd();

        skyboxTexture2.bind();
        //build right wall
        gl.glTranslatef(size, 0.0f, size);
        gl.glBegin(GL.GL_QUADS);
        gl.glTexCoord2f(1.0f, 1.0f);gl.glVertex3f(0.0f, -size, size);
        gl.glTexCoord2f(1.0f, 0.0f);gl.glVertex3f(0.0f, size, size);
        gl.glTexCoord2f(0.0f, 0.0f);gl.glVertex3f(0.0f, size, -size);
        gl.glTexCoord2f(0.0f, 1.0f); gl.glVertex3f(0.0f, -size, -size);
        gl.glEnd();


        skyboxTexture3.bind();
        //build back wall
        gl.glTranslatef(-size, 0.0f, size);
        gl.glBegin(GL.GL_QUADS);
        gl.glTexCoord2f(1.0f, 1.0f);gl.glVertex3f(-size, -size, 0.0f);
        gl.glTexCoord2f(0.0f, 1.0f);gl.glVertex3f(size, -size, 0.0f);
        gl.glTexCoord2f(0.0f, 0.0f);gl.glVertex3f(size, size, 0.0f);
        gl.glTexCoord2f(1.0f, 0.0f); gl.glVertex3f(-size, size, 0.0f);
        gl.glEnd();



        skyboxTexture4.bind();
        //build left wall
        gl.glTranslatef(-size, 0.0f, -size);
        gl.glBegin(GL.GL_QUADS);
        gl.glTexCoord2f(0.0f, 1.0f);gl.glVertex3f(0.0f, -size, size);
        gl.glTexCoord2f(0.0f, 0.0f);gl.glVertex3f(0.0f, size, size);
        gl.glTexCoord2f(1.0f, 0.0f);gl.glVertex3f(0.0f, size, -size);
        gl.glTexCoord2f(1.0f, 1.0f); gl.glVertex3f(0.0f, -size, -size);
        gl.glEnd();


        gl.glDisable(GL.GL_TEXTURE_2D);
}
```

Παρακάτω στην εικόνα 3.3 και 3.4 βλέπουμε το αποτέλεσμα της σχεδίασης του ουρανού και του εδάφους. Όπως και πριν στην εικόνα 3.3 η κάμερα βρίσκεται παράλληλα στους άξονες x και z και συγκεκριμένα λίγο πιο πάνω από την επιφάνεια του εδάφους, ενώ στην εικόνα 3.4 βρίσκεται πιο ψηλά για να φανεί μια πανοραμική θέα του κύβου που σχεδιάσαμε.

Αυτό που βλέπουμε στην εικόνα 3.4 είναι ένας κύβος χωρίς να έχει σχεδιαστεί η πάνω πλευρά του το οποίο δεν χρειάζεται καθώς ο παίκτης δεν θα μπορεί να κοιτάει τόσο ψηλά. Το παιχνίδι λοιπόν θα εκτυλίσσεται μέσα σε αυτόν τον χώρο, στον οποίο έπειτα θα προστεθούν τρισδιάστατα αντικείμενα. Ο παίκτης δεν θα μπορεί να βγαίνει από τα όρια αυτού του κύβου. Χρησιμοποιούμε συμμετρικές εικόνες στον ουρανό για να έχουμε καλύτερο οπτικό αποτέλεσμα και να μην φαίνονται οι γωνίες του κύβου.



Εικόνα 3.3: Σχεδίαση ουρανού και εδάφους με την κάμερα να κοιτάει παράλληλα στους οριζόντιους άξονες



Εικόνα 3.4: Σχεδίαση ουρανού και εδάφους με την κάμερα ψηλά να κοιτάει στον κέντρο των αξόνων

3.3. 3D Models

3.3.1. Εισαγωγή

Στα τρισδιάστατα γραφικά υπολογιστών, η 3D μοντελοποίηση (3D modeling ή meshing) είναι η διαδικασία ανάπτυξης μιας μαθηματικής αναπαράστασης οποιασδήποτε τρισδιάστατης επιφάνειας ενός αντικειμένου μέσω ενός εξειδικευμένου λογισμικού. Το αποτέλεσμα αυτής της διαδικασίας ονομάζεται τρισδιάστατο μοντέλο (3d model).

Τα τρισδιάστατα μοντέλα αντιπροσωπεύουν ένα τρισδιάστατο αντικείμενο χρησιμοποιώντας ένα σύνολο σημείων στον τρισδιάστατο χώρο, τα οποία συνδέονται μεταξύ τους με διάφορες γεωμετρικές οντότητες όπως τρίγωνα, καμπύλες, γραμμές κτλ. Είναι δηλαδή μια συλλογή δεδομένων (σημεία και άλλες πληροφορίες) τα οποία μπορούν να δημιουργηθούν είτε με το χέρι είτε αλγοριθμικά.

Για διευκόλυνση της σχεδίασης τρισδιάστατων αντικειμένων στο παραπάνω σκηνικό που περιγράψαμε, θα χρησιμοποιήσουμε το μορφότυπο αρχείων περιγραφής 3D μοντέλων OBJ. Ένα αρχείο τέτοιου τύπου δεν είναι τίποτα άλλο παρά ένα αρχείο στο οποίο είναι καταγεγραμμένα δεδομένα όπως συντεταγμένες, πλευρές και υφές το οποίο μάλιστα μπορούμε να το διαβάσουμε χρησιμοποιώντας έναν επεξεργαστή κειμένου. Παρακάτω περιγράφω εκτενώς τις δυνατότητες των αρχείων OBJ.

Βεβαίως για να χρησιμοποιήσουμε αυτού του είδους τα αρχεία, θα πρέπει να γράψουμε ένα κομμάτι κώδικα το οποίο διαβάζει αρχεία OBJ και σχεδιάζει τις επιφάνειες οι οποίες περιγράφονται σε ένα τέτοιο αρχείο με χρήση της βιβλιοθήκης JOGL. Στο συγκεκριμένο παιχνίδι αυτό γίνεται από τις κλάσεις OBJModel, FaceMaterials, Faces, Material, Materials, ModelDimensions.

3.3.2. Αρχεία OBJ

Το μορφότυπο αρχείων OBJ[12] για την αναπαράσταση τρισδιάστατων μοντέλων αναπτύχθηκε από την Wavefront Technologies. Είναι ανοικτό και παγκοσμίως αποδεκτό. Ο πρώτος χαρακτήρας κάθε γραμμής καθορίζει τον τύπο της εντολής. Αν ο πρώτος χαρακτήρας είναι το σύμβολο #, τότε η γραμμή πρόκειται για σχόλιο και αγνοείται, όπως επίσης και οι κενές γραμμές. Το αρχείο αναλύεται από πάνω προς τα κάτω. Παρακάτω βλέπουμε το σύνολο των εντολών που χρησιμοποιούνται στα OBJ αρχεία μαζί με κάποια τυχαία ορίσματα. Οι αγκύλες ορίζουν κάποιο όρισμα το οποίο είναι προαιρετικό.

comment

Αυτές οι γραμμές πάντα αγνοούνται. Συνήθως η πρώτη γραμμή ενός αρχείου OBJ είναι ένα σχόλιο στο οποίο αναφέρεται το πρόγραμμα με το οποίο δημιουργήθηκε το OBJ αρχείο.

v x y z

Η εντολή vertex. Ορίζει μια κορυφή με τις τρείς συντεταγμένες της. Κάθε κορυφή ονομάζεται αναλόγως την σειρά με την οποία βρίσκεται στο αρχείο. Για παράδειγμα η πρώτη κορυφή αναφέρεται ως '1', η δεύτερη ως '2' κ.ο.κ. Η εντολή αυτή δεν ορίζει μια γεωμετρική οντότητα, απλά σημεία στον χώρο.

vt u v [w]

Η εντολή vertex texture καθορίζει την αντιστοίχηση UV (UV mapping) και προαιρετικά την W. Οι τιμές U, V και W πρέπει να είναι αριθμοί κινητής υποδιαστολής μεταξύ 0 και 1, οι οποίοι δείχνουν πως θα αντιστοιχηθεί η υφή. Από μόνες τους δεν λένε κάτι, πρέπει να οριστούν ως ομάδα με την εντολή f.

vn x y z

Η εντολή vertex normal καθορίζει ένα κάθετο διάνυσμα. Πολλές φορές η εντολή αυτή δεν χρησιμοποιείται καθώς η εντολή f θα χρησιμοποιήσει τις εντολές v με την σειρά και το κάθετο διάνυσμα θα καθοριστεί από εκεί. Όπως και οι εντολές vt δεν σημαίνουν κάτι από μόνες τους, πρέπει να ομαδοποιηθούν από μια εντολή f.

f v1[/vt1][/vn1] v2[/vt2][/vn2] v3[/vt3][/vn3]…

Η εντολή face ομαδοποιεί μια λίστα κορυφών (verticies), οι οποίες έχουν οριστεί προηγουμένως και έτσι καθορίζει ένα πολύγωνο. Μπορούμε να χρησιμοποιήσουμε όσες κορυφές θέλουμε. Για να αναφερθούμε σε μια κορυφή πρέπει να χρησιμοποιήσουμε τον αριθμό της, για παράδειγμα η εντολή f 54 55 56 57 δημιουργεί ένα πολύγωνο από τις κορυφές 54-57. Επίσης προαιρετικά μπορούμε να συνδέσουμε κάθε κορυφή με μια εντολή vt, η οποία μας λέει πώς να αντιστοιχήσουμε την υφή σε αυτό το σημείο, καθώς και με μια εντολή vn που καθορίζει το κάθετο διάνυσμα. Εάν χρησιμοποιήσουμε ένα vt ή vn για μια κορυφή τότε πρέπει να ορίσουμε ένα και για κάθε άλλη κορυφή. Το κάθετο διάνυσμα (normal) ορίζει προς ποιά κατεύθυνση θα κοιτάει το πολύγωνο. Αν δεν δοθεί τότε θα υπονοηθεί με την σειρά των κορυφών, η κατεύθυνση των οποίων είναι αριστερόστροφη. Αν θέλουμε να αλλάξουμε την κατεύθυνση που κοιτάει το πολύγωνο χωρίς να χρησιμοποιήσουμε την εντολή vn, θα πρέπει να δώσουμε τις κορυφές με αντίθετη σειρά. Αν η τιμή των v, vt ή vn είναι αρνητική τότε πρόκειται για σχετική τιμή (σημαίνει πήγαινε πίσω τόσες κορυφές όσες η τιμή της εντολής, ώστε να βρεις την κορυφή την οποία πρέπει να χρησιμοποιήσεις).

g name

Η εντολή group ομαδοποιεί τις εντολές f που ακολουθούν και επομένως τα πολύγωνα που περιγράφονται ώστε να οριστεί μια υπό-ομάδα του αντικειμένου με το συγκεκριμένο όνομα.

usemtl name

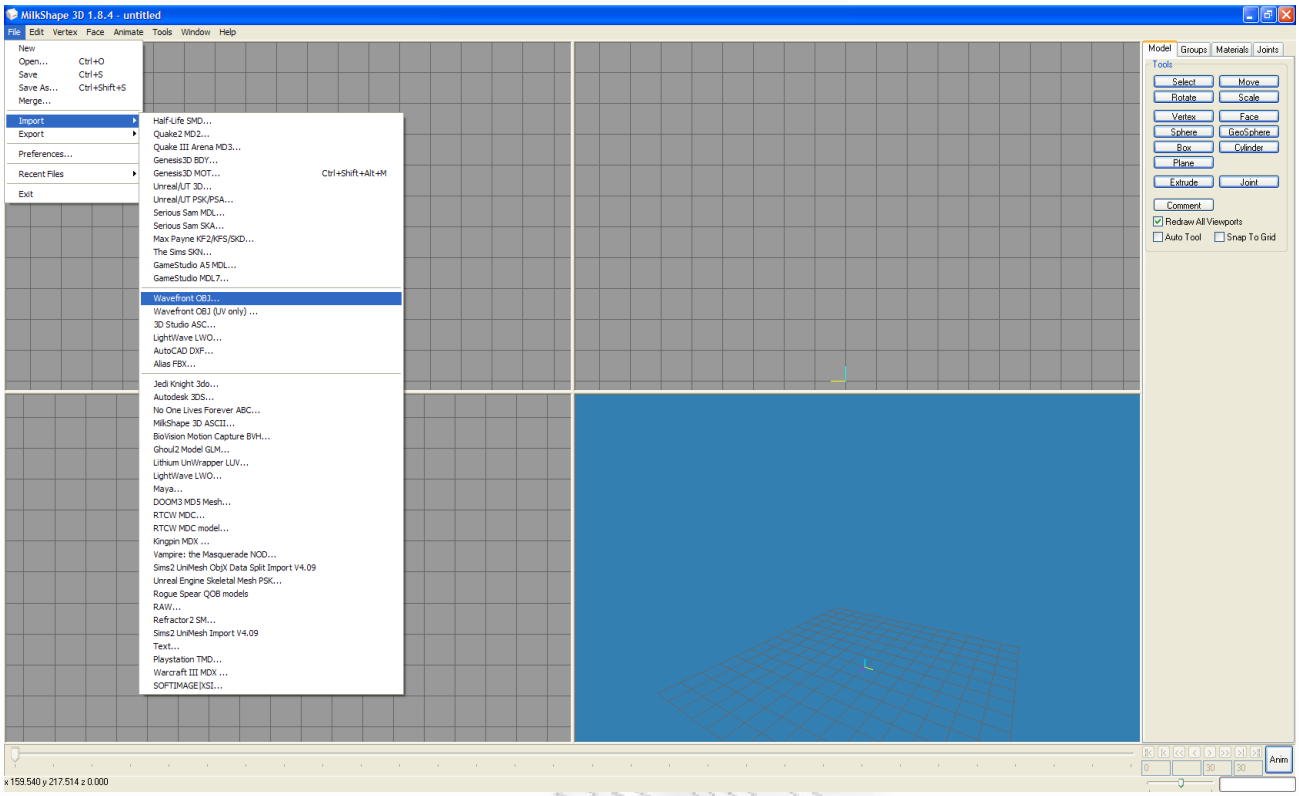
Η εντολή use material ουσιαστικά δίνει την δυνατότητα να χρησιμοποιηθεί ένα υλικό (υφή) με το συγκεκριμένο όνομα που αναγράφεται. Όλες οι εντολές f που ακολουθούν θα χρησιμοποιήσουν αυτό το υλικό μέχρι να βρεθεί μια άλλη εντολή usemtl.
Η μορφή και η σειρά των εντολών πρέπει να είναι η ακόλουθη:

```
# σχόλιο για το πρόγραμμα με το οποίο δημιουργήθηκε το αρχείο
# όλες οι εντολές v
v x y z
v…
# όλες οι εντολές vn


vn x y z
vn…
# όλες οι εντολές vt
vt u v [w]
vt…
# πρέπει να οριστεί το αντικείμενο και το υλικό
g object
usemtl material
# όλες οι εντολές f
f 1/1 2/2 3/3 4/4
f…
```

3.3.3. MilkShape 3D

Το πρόγραμμα το οποίο χρησιμοποίησα για να επεξεργαστώ αρχεία OBJ είναι το MilkShape 3D [11]. Είναι ένα λογισμικό τρισδιάστατης μοντελοποίησης, αρκετά εύκολο στη χρήση και πολύ βολικό για δημιουργία ή επεξεργασία μοντέλων OBJ τα οποία χρησιμοποιούμε. Έχει δυνατότητα μετατροπής ενός τύπου μοντέλου σε άλλον τύπο π.χ. 3ds σε OBJ. Θα κάνω μια σύντομη αναφορά στις λειτουργίες και δυνατότητες του συγκεκριμένου προγράμματος.

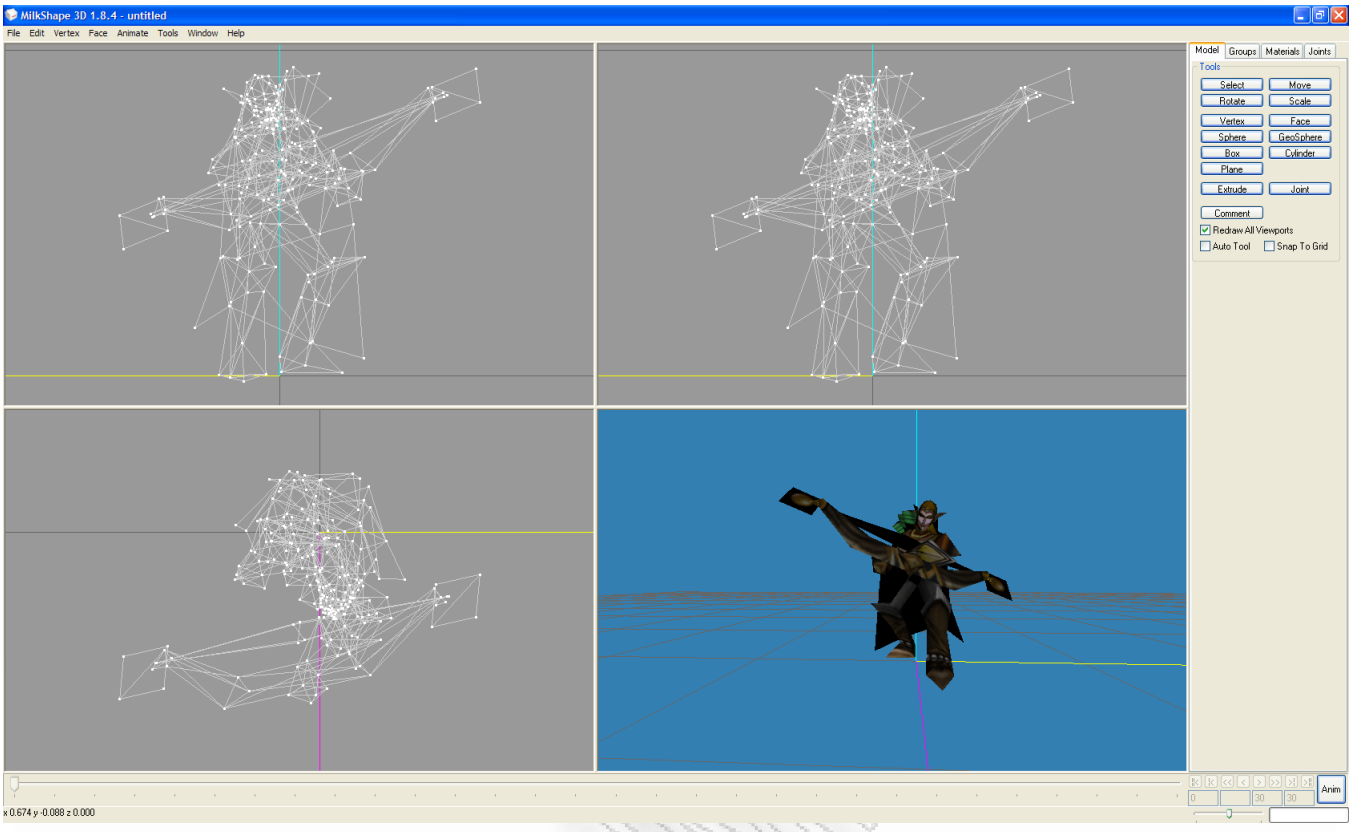
Για να εισάγουμε ένα μοντέλο τύπου OBJ πηγαίνουμε File → Import → Wavefront OBJ… (εικόνα 3.5). Επιλέγουμε ένα αρχείο τύπου OBJ και σε περίπτωση που η εισαγωγή γίνει με επιτυχία το βλέπουμε σχεδιασμένο στις τέσσερις διαφορετικές περιοχές. Στην εικόνα 3.6 βλέπουμε το μοντέλο του τοξότη που θα χρησιμοποιήσουμε στο παιχνίδι.



Εικόνα 3.5: Εισαγωγή μοντέλου OBJ στο Milkshape 3D

Στα δεξιά βλέπουμε τις λειτουργίες του προγράμματος. Στο πεδίο Model μπορούμε να κάνουμε μετατροπές στο μοντέλο όπως να το περιστρέψουμε ή να το μετατοπίσουμε στον χώρο. Στο πεδίο Groups βλέπουμε τις υπό-ομάδες του μοντέλου. Αυτό έχει άμεση σχέση με την εντολή g name που είδαμε προηγουμένως για τα αρχεία τύπου OBJ. Υπάρχει δυνατότητα κάποιες επιφάνειες του μοντέλου να τις ομαδοποιήσουμε έτσι ώστε να διευκολυνθούμε κατά την διάρκεια της επεξεργασίας του. Στο πεδίο Materials μπορούμε να δούμε τις υφές που χρησιμοποιούνται στο μοντέλο, καθώς και να προσθέσουμε μια άλλη ή να αλλάξουμε μια υπάρχουσα. Τέλος υπάρχει το πεδίο Joints το οποίο χρησιμοποιείται για την δημιουργία σκελετικού animation. Η λογική είναι να οριστούν κάποια joints τα οποία θα λειτουργούν σαν αρθρώσεις όπως ακριβώς σε έναν σκελετό. Το animation στο συγκεκριμένο παιχνίδι θα το αναλύσουμε σε επόμενο κεφάλαιο, δεν θα χρησιμοποιήσουμε πάντως αυτήν την λειτουργία του Milkshape 3D.

Εφ’ όσον δημιουργήσουμε ή τροποποιήσουμε ένα μοντέλο OBJ πρέπει να το εξάγουμε πάλι στην μορφή αυτή (ή όποια άλλη επιθυμούμε). Για να το κάνουμε αυτό πάμε File → Export →Wavefront OBJ…και το αποθηκεύουμε με την ονομασία που θέλουμε. Θα αποθηκευτούν δύο αρχεία, ένα αρχείο με κατάληξη .obj και ένα αρχείο με κατάληξη .mtl. Το αρχείο mtl περιέχει πληροφορίες για τις υφές που χρησιμοποιούνται και πρέπει να οριστεί στην αρχή του αρχείου obj με την εντολή mtllib name.mtl.



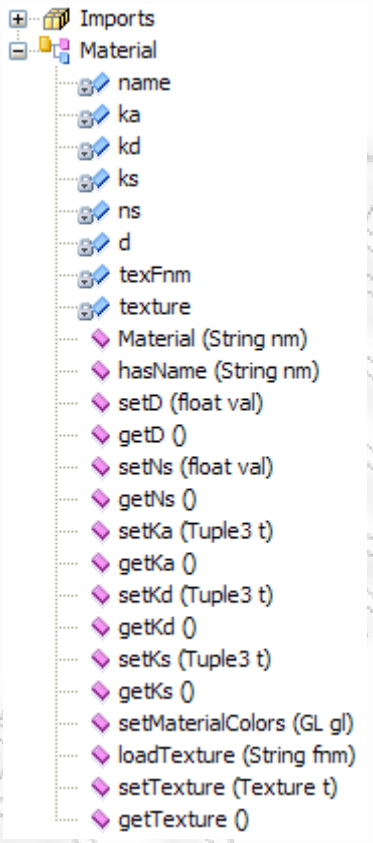
Εικόνα 3.6: Μοντέλο τοξότη στο Milkshape 3D

3.3.4. OBJLoader

Η εισαγωγή των μοντέλων OBJ στο παιχνίδι γίνεται από τις κλάσεις OBJModel, FaceMaterials, Faces, Material, Materials, ModelDimensions. Η λογική είναι η εξής: διαβάζεται το αρχείο .obj και το αρχείο .mtl γραμμή γραμμή και σχεδιάζονται οι επιφάνειες οι οποίες περιγράφονται στο αρχείο, με τις κατάλληλες εντολές της βιβλιοθήκης JOGL.

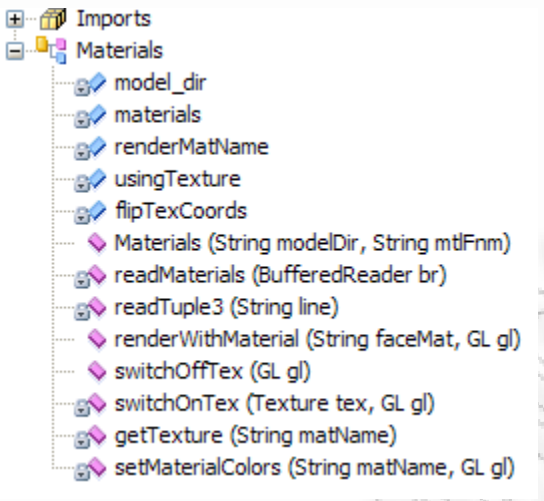
Η κλάση Material αντιπροσωπεύει ένα υλικό το οποίο χρησιμοποιείται από το μοντέλο. Περιέχει τις ιδιότητες και μεθόδους που βλέπουμε στην εικόνα 3.7. Το πεδίο name αναφέρεται στο όνομα του material το οποίο είναι ορισμένο στο αρχείο .mtl. Τα πεδία ka, kd, ks, ns, d είναι οι τιμές ambient, diffuse, specular, shininess και alpha οι οποίες προσδιορίζουν τον φωτισμό του υλικού. Στο πεδίο texFnm αποθηκεύεται το όνομα του αρχείου που περιέχει την εικόνα σύμφωνα με την οποία θα δημιουργηθεί η υφή στο αντικείμενο και το πεδίο texture είναι ένα στιγμιότυπο της κλάσης Texture στο οποίο αποθηκεύεται η υφή αυτή. Η μέθοδος-κατασκευαστής δημιουργεί ένα αντικείμενο τύπου Material με όρισμα το όνομα του Material, αρχικοποιώντας όλα τα πεδία της συγκεκριμένης κλάσης. Η μέθοδος hasName ελέγχει αν το όνομα του υλικού είναι ίδιο με αυτό του ορίσματος. Οι μέθοδοι setD, getD, setNs, getNs, setKa, getKa, setKd, getKd, setKs, getKs χρησιμοποιούνται για να αλλάζουν ή να επιστρέφουν τις τιμές των αντίστοιχων πεδίων. Η μέθοδος setMaterialColors χρησιμοποιεί τα παραπάνω πεδία για να ορίσει πως το υλικό θα ανταποκρίνεται στον φωτισμό. Αυτό γίνεται με την εντολή της βιβλιοθήκης JOGL, glMaterialfv και τα κατάλληλα ορίσματα για κάθε περίπτωση. Η μέθοδος loadTexture είναι η μέθοδος η οποία είναι υπεύθυνη να διαβάσει το αρχείο εικόνας για την

υφή (αν υπάρχει τέτοιο) και να δημιουργήσει το αντίστοιχο αντικείμενο τύπου Texture το οποίο και αποθηκεύει στο πεδίο texture της κλάσης. Τέλος υπάρχουν και οι μέθοδοι setTexture και getTexture σε περίπτωση που θέλουμε να αλλάξουμε ή να πάρουμε την υφή που χρησιμοποιείται.



Εικόνα 3.7: Ιδιότητες και μέθοδοι της κλάσης Material

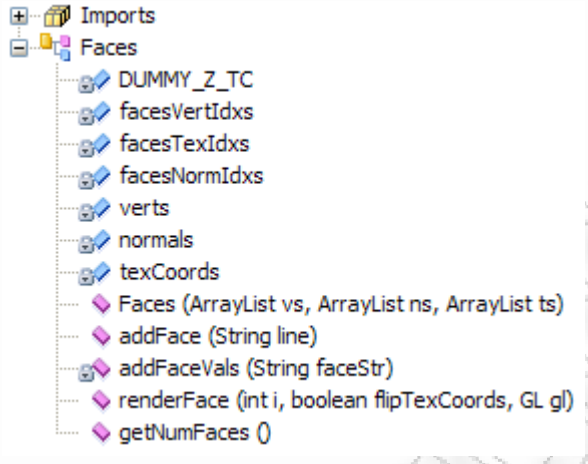
Η κλάση Materials περιλαμβάνει όλα τα υλικά που χρησιμοποιούνται από ένα μοντέλο. Στην εικόνα 3.8 βλέπουμε τις ιδιότητες και τις μεθόδους της. Το πεδίο model_dir αναφέρεται στη διαδρομή στην οποία βρίσκεται το αρχείο .mtl. Το πεδίο materials είναι ένα αντικείμενο τύπου array list στο οποίο αποθηκεύονται όλα τα material που διαβάζονται από το αρχείο .mtl. Το πεδίο renderMatname είναι ένα string στο οποίο αποθηκεύεται το material το οποίο χρησιμοποιείται την συγκεκριμένη στιγμή που γίνεται το rendering. Το usingTexture είναι μια μεταβλητή τύπου boolean για να ξέρουμε αν για το συγκεκριμένο υλικό χρησιμοποιείται υφή από αρχείο εικόνας ενώ η μεταβλητή τύπου flipTexCoords μας δείχνει αν οι συντεταγμένες της υφής έχουν περιστραφεί κατά τον άξονα y.



Εικόνα 3.8: Ιδιότητες και μέθοδοι της κλάσης Materials

Η μέθοδος readMaterials διαβάζει το αρχείο .mtl και καλείται από την μέθοδο-κατασκευαστή της κλάσης Materials. Παίρνει σαν όρισμα ένα αντικείμενο τύπου BufferedReader το οποίο δημιουργείται από το αρχείο προς ανάγνωση. Διαβάζει το αρχείο γραμμή γραμμή αναγνωρίζοντας την κάθε εντολή OBJ (π.χ. Ka, Kd κτλ) και δημιουργεί ένα αντικείμενο τύπου Material για κάθε υλικό με τις συγκεκριμένες παραμέτρους και τα αποθηκεύει όλα στο array list Materials. Η μέθοδος readTuple3 είναι βοηθητική για την readMaterials και συγκεκριμένα διαβάζει τρεις συνεχόμενες τιμές x,y,z από το αρχείο (προορίζεται για τις τιμές των Ka, Kd,Ks,Ns). Οι υπόλοιπες μέθοδοι χρησιμοποιούνται όταν θέλουμε να βρεθεί ένα material από το array materials και να γίνει χρήση του στο rendering. Η renderWithMaterial παίρνει σαν όρισμα το material το οποίο θέλουμε και έπειτα καλεί την getTexture. Αν βρεθεί το material στον πίνακα λίστας και υπάρχει texture επιστρέφεται το texture που χρησιμοποιεί το material, αλλιώς καλείται η setMaterialColors η οποία με την σειρά της καλεί την setMaterialColors της κλάσης Material για το material το οποίο υπάρχει σαν όρισμα στην μέθοδο.

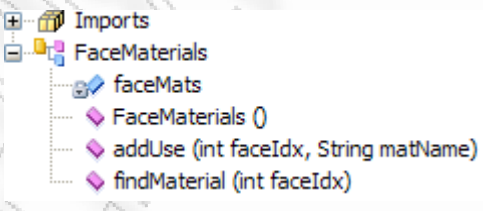
Η κλάση faces (εικόνα 3.9) έχει κατασκευαστεί για να χειρίζεται τις πλευρές-επιφάνειες (faces) του μοντέλου. Έχει έξι array lists: τα facesVertIdxs, facesTextIdxs, facesNormIdxs και τα verts, normals, texCoords. Στους πρώτους τρείς πίνακες αποθηκεύονται τα indexes (αρίθμηση) των κορυφών, συντεταγμένων υφών και των καθέτων διανυσμάτων. Στους άλλους τρείς οι συντεταγμένες. Αυτό γίνεται στην μέθοδο addFace η οποία παίρνει σαν όρισμα την γραμμή στην οποία συναντάμε μια εντολή f.



Εικόνα 3.9: Ιδιότητες και μέθοδοι της κλάσης Faces

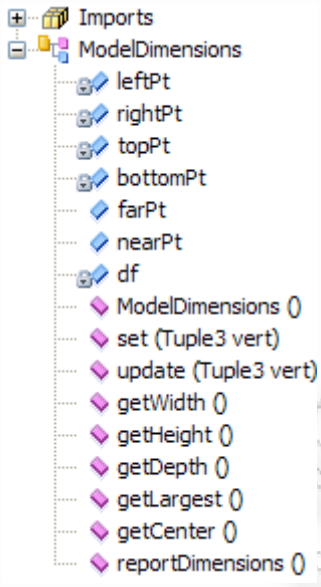
Η μέθοδος addFaceVals διαβάζει μια εγγραφή v/vt/vn και καλείται από την μέθοδο addFace. Η renderFace σχεδιάζει το face με τον αριθμό (index) i του ορίσματος και η μέθοδος getNumFaces επιστρέφει τον αριθμό των faces.

Η κλάση FaceMaterials (εικόνα 3.10) διαχειρίζεται τα materials τα οποία χρησιμοποιούνται από τα faces. Γίνεται δηλαδή μια αντιστοίχηση των materials με τα faces από τα οποία χρησιμοποιούνται. Αποθηκεύεται το όνομα του material σε ζευγάρι με τον αριθμό του face στον hash map faceMats. Η μέθοδος addUse προσθέτει μια τέτοια εγγραφή ενώ η μέθοδος findMaterial βρίσκει ένα material σε αντιστοιχία με τον αριθμό του face που δίδεται σαν όρισμα.



Εικόνα 3.10: Ιδιότητες και μέθοδοι της κλάσης FaceMaterials

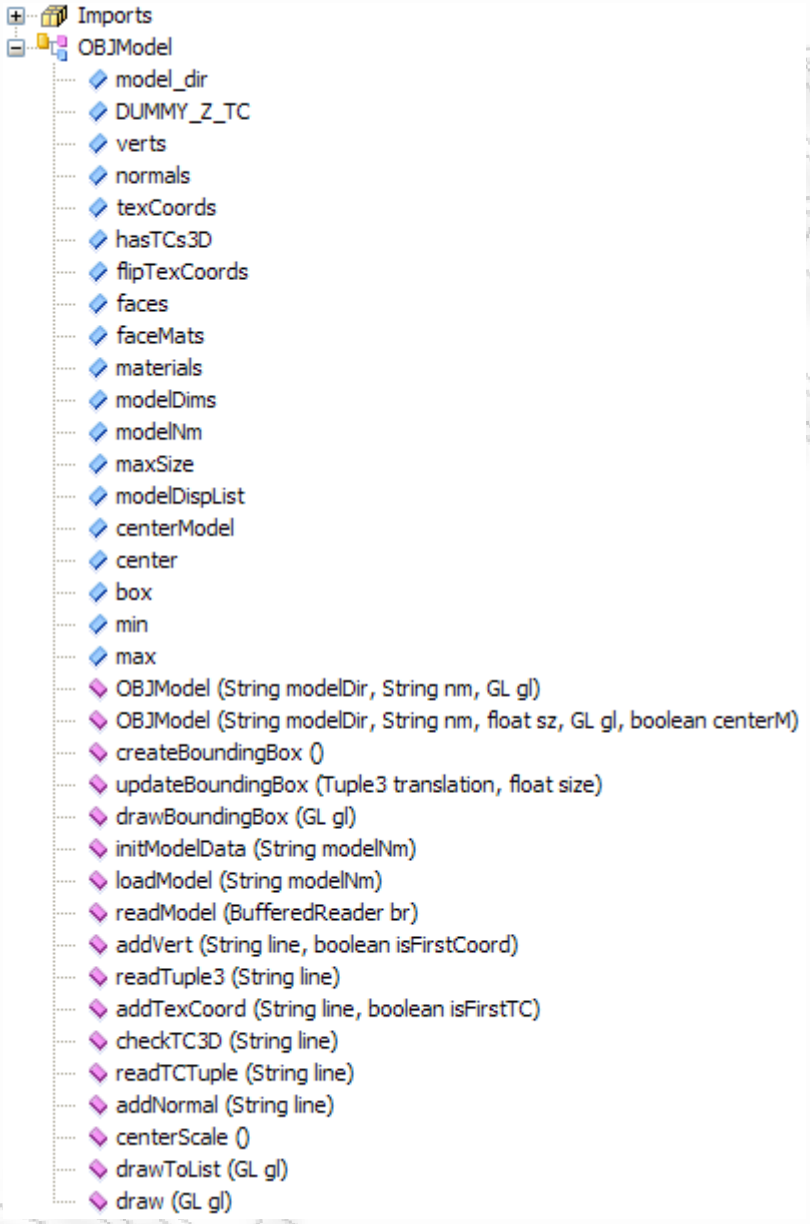
Η κλάση ModelDimensions χρησιμοποιείται για να γνωρίζουμε τις διαστάσεις και το κέντρο του μοντέλου. Βλέπουμε τις ιδιότητες και τις μεθόδους της στην εικόνα 3.11. Η μεταβλητή leftPt αναφέρεται στο αριστερότερο σημείο του μοντέλου στον άξονα x, ενώ η rightPt στο δεξιότερο σημείο στον άξονα x. Αντίστοιχα η topPt και bottomPt είναι το ψηλότερο και χαμηλότερο σημείο στον άξονα y και η farPt και nearPt το μακρύτερο και κοντύτερο σημείο στον άξονα z. Η μέθοδος set χρησιμοποιείται για να ορίσουμε τιμές στις παραπάνω μεταβλητές και συγκεκριμένα την καλούμε για να κάνουμε αρχικοποίηση των τιμών. Αντίθετα η μέθοδος update χρησιμοποιείται αφού έχει γίνει αρχικοποίηση για να ενημερωθούν οι παραπάνω τιμές. Ελέγχεται αν η τιμές που υπάρχουν πρέπει να αλλάξουν με τις τιμές του ορίσματος της μεθόδου. Οι υπόλοιπες μέθοδοι είναι για να επιστρέφουν τις τιμές που καταλαβαίνουμε όπως το ύψος του μοντέλου, το βάθος κτλ.



Εικόνα 3.11: Ιδιότητες και μέθοδοι της κλάσης ModelDimensions

Όλες οι παραπάνω κλάσεις χρησιμοποιούνται στην κλάση OBJModel η οποία είναι και η σημαντικότερη. Βλέπουμε τις ιδιότητες και τις μεθόδους της στην εικόνα 3.11. Στο πεδίο model_dir αποθηκεύεται η διαδρομή στην οποία βρίσκεται το OBJ μοντέλο δηλαδή τα αρχεία .obj και .mtl. Οι πίνακες verts, normals και texCoords περιέχουν τις συντεταγμένες κορυφών, καθέτων διανυσμάτων και υφών. Το πεδίο hasTCs3D δείχνει αν χρησιμοποιούνται τρισδιάστατες υφές, ενώ το flipTexCoords αν έχουν περιστραφεί οι συντεταγμένες υφών ως προς τον άξονα y. Οι μεταβλητές faces, faceMats, materials, modelDims είναι αντικείμενα των κλάσεων που είδαμε προηγουμένως. Στο modelNm αποθηκεύεται το όνομα του μοντέλου, το maxSize είναι το μέγεθος του μοντέλου το οποίο μπορούμε να ορίσουμε στην μέθοδο κατασκευαστή, το centerModel είναι μια μεταβλητή τύπου boolean η οποία δείχνει αν θέλουμε το μοντέλο μας να σχεδιάζεται από το κέντρο του χώρου στον οποίο πρόκειται να σχεδιαστεί, το center είναι το κέντρο του μοντέλου και τέλος το box, το min και το max αναφέρονται στο bouning box (το “κουτί” το οποίο σχεδιάζεται γύρω από το μοντέλο για να γίνεται ο έλεγχος συγκρούσεων) για το οποίο θα αναλύσουμε σε επόμενο κεφάλαιο. Η κλάση έχει δύο μεθόδους κατασκευαστές με διαφορετικά ορίσματα για να χρησιμοποιούμε όποια επιθυμούμε στην δημιουργία ενός αντικειμένου OBJModel. Η μέθοδος createBoundingBox χρησιμοποιείται για να δημιουργήσουμε ένα bounding box στο μοντέλο ενώ η updateBoundingBox για να ενημερωθεί η θέση του bounding box σε περίπτωση που μετακινηθεί το μοντέλο στον χώρο. Η drawBoundingBox σχεδιάζει το bounding box του μοντέλου σε περίπτωση που θέλουμε να το δούμε. Θα αναφερθούμε αναλυτικότερα στα bounding boxes και γενικά στον έλεγχο συγκρούσεων πιο μετά. Η μέθοδος initModelData χρησιμοποιείται για να δημιουργηθούν τα αντικείμενα των προηγούμενων κλάσεων που είδαμε και να αρχικοποιηθούν τα δεδομένα μας. Η μέθοδος loadModel παίρνει σαν όρισμα το όνομα του μοντέλου ώστε με συνδυασμό με την διαδρομή που έχουμε ορίσει στο model_dir, να βρεθούν τα αρχεία .obj και .mtl και να διαβαστούν από την μέθοδο readModel. Η τελευταία χρησιμοποιεί τις μεθόδους addvert, addTexCoord και addNormal για την αποθήκευση των τιμών που διαβάζονται. Τέλος οι μέθοδοι drawToList και draw σχεδιάζουν το μοντέλο. Για κάθε face που έχει το μοντέλο

καλείται η μέθοδος renderWithMaterial της κλάσης materials και αλυσιδωτά η renderWithMaterial της κλάσης Material.



Εικόνα 3.12: Ιδιότητες και μέθοδοι της κλάσης OBJModel

Έστω λοιπόν ότι θέλουμε να σχεδιάσουμε ένα μοντέλο OBJ το οποίο βρίσκεται στον φάκελο models. Πρέπει να δημιουργήσουμε ένα αντικείμενο της κλάσης OBJModel ως εξής:

```
OBJModel model = new OBJModel(“models/”, “modelName”, gl);
```

Και σε περίπτωση που θέλουμε το μοντέλο να μην κεντράρεται στον χώρο σχεδίασης και να του ορίσουμε εμείς το μέγεθος που επιθυμούμε, πρέπει να χρησιμοποιήσουμε την άλλη μέθοδο κατασκευαστή ως εξής:

```
OBJModel model = new OBJModel(“models/”, “modelName”, 50.0f, gl, false);
```

Το μοντέλο μας θα έχει μέγεθος 50.0f και όνομα modelName. Από την στιγμή που έχει δημιουργηθεί το μοντέλο ως αντικείμενο τύπου OBJModel μπορούμε να το σχεδιάσουμε με την μέθοδο draw της κλάσης OBJModel ως εξής (πρέπει να την καλέσουμε στην display μέθοδο, ενώ το μοντέλο το δημιουργούμε στην μέθοδο init):

model.draw(gl);

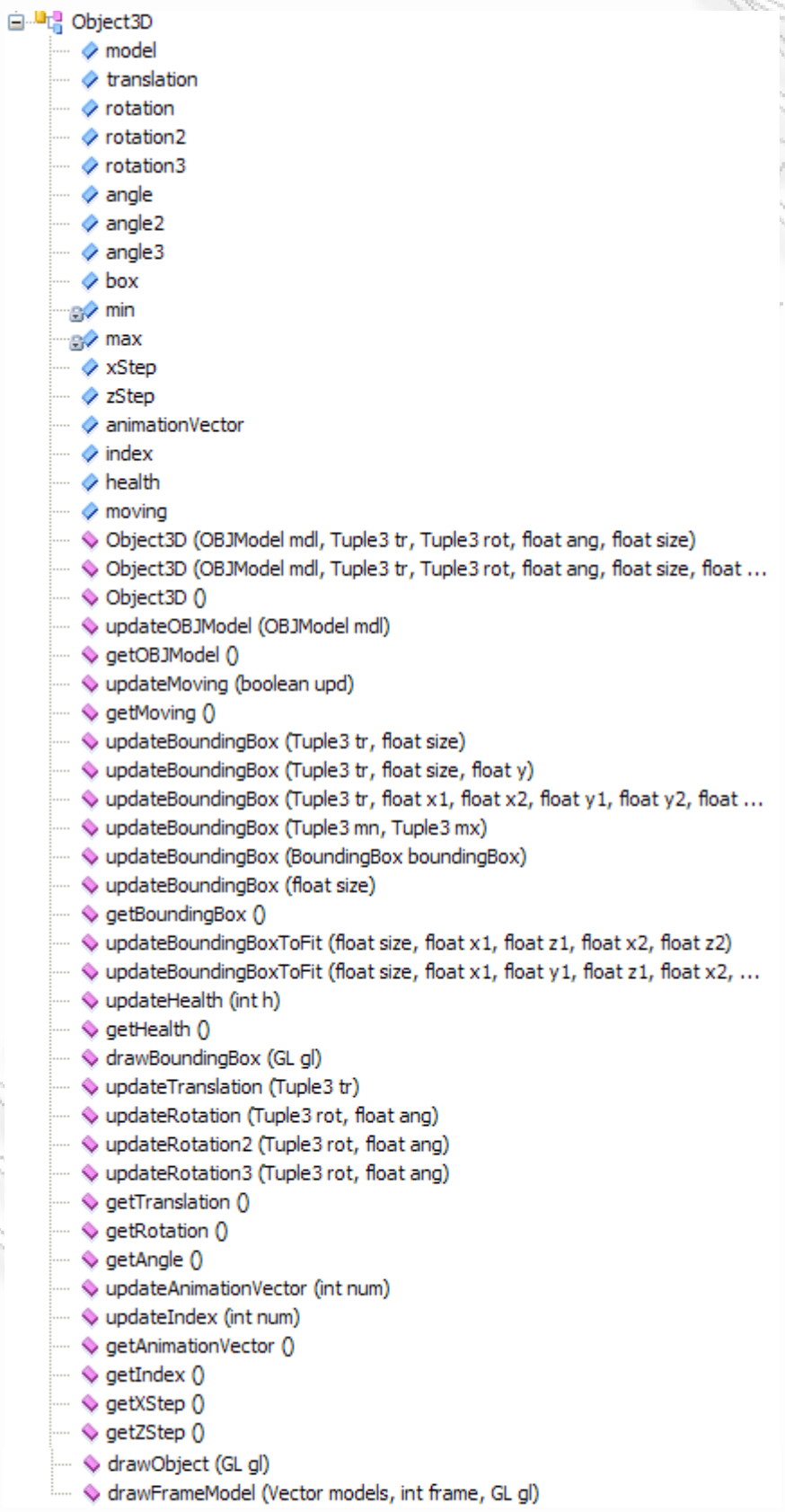
Παρακάτω βλέπουμε το αποτέλεσμα της σχεδίασης του obj μοντέλου “heroStand1.obj” σε συνδυασμό με τον ουρανό και το έδαφος που σχεδιάσαμε προηγουμένως.

Εικόνα 3.13: Σχεδίαση μοντέλου heroStand.obj μέσα στον κύβο εδάφους-ουρανού

Να υπενθυμίσω ότι για να σχεδιαστεί το μοντέλο στο επίπεδο του εδάφους θα πρέπει να το έχουμε μετατοπίσει -10 μονάδες στον άξονα y, δηλαδή να το σχεδιάσουμε στην θέση (x, -10.0f, z). Για να γίνει αυτό θα πρέπει να χρησιμοποιήσουμε την εντολή gl.glTranslatef(0.0f, -10.0f, 0.0f) πριν από την εντολή σχεδίασης.

Με τον ίδιο τρόπο μπορούμε να σχεδιάσουμε όποιο άλλο μοντέλο obj θέλουμε. Σκοπός μας είναι το μοντέλο του τοξότη να σχεδιάζεται πάντα λίγο μπροστά από την οθόνη, έτσι ώστε πάντα να το βλέπουμε από πίσω ανεξαρτήτως του σημείου που βρίσκεται η κάμερα. Πρώτα όμως, πριν δούμε πως

γίνεται αυτό, στο επόμενο κεφάλαιο θα περιγράψουμε τον τρόπο με τον οποίο μπορούμε να μετακινούμε και να περιστρέφουμε την κάμερα στον χώρο.



Εικόνα 3.14: Μεταβλητές και μέθοδοι της κλάσης Object3D

Για την διευκόλυνση της σχεδίασης ενός μοντέλου και για να υπάρχουν συγκεντρωμένες όλες οι λειτουργίες που μας ενδιαφέρουν για την επεξεργασία του όπως η ενημέρωση της θέσης στην οποία σχεδιάζεται, η αλλαγή της γωνίας σχεδίασης του κτλ δημιούργησα την κλάση Object3D. Στην εικόνα 3.14 βλέπουμε τις μεθόδους και τις μεταβλητές της κλάσης αυτής. Η κλάση έχει τρείς μεθόδους κατασκευαστές οι οποίες εξυπηρετούν και διαφορετικούς σκοπούς. Η μία δεν έχει ορίσματα σε περίπτωση που δεν θέλουμε να δώσουμε τιμή στις μεταβλητές. Αρχικοποιούνται όλες οι μεταβλητές με τιμές null ή 0 σε περίπτωση που πρόκειται για αριθμό.

Υπάρχει άλλη μία μέθοδος κατασκευαστής σε περίπτωση που θέλουμε να ορίσουμε μοντέλο OBJ στο αντικείμενο(OBJModel mdl), θέση (Tuple3 tr), γωνία (Tuple3 rot και float ang) και μέγεθος (float size). Η τελευταία δίνει εκτός των προηγουμένων και την δυνατότητα να ορίσουμε δύο τιμές που χρησιμοποιούνται για να ορίσουν την κατεύθυνση της κίνησης (xStep και zStep). Τις τελευταίες τις περιγράφω στο κεφάλαιο για την κίνηση του τοξότη. Χρησιμοποιείται αυτή η μέθοδος κατασκευαστής όταν φτιάχνουμε ένα βέλος που πρέπει να γνωρίζουμε προς ποια κατεύθυνση θα κινηθεί. Η μέθοδος drawBoundingbox χρησιμοποιείται για να σχεδιαστεί το κουτί που περιβάλλει ένα τρισδιάστατο αντικείμενο για να γίνεται ο έλεγχος συγκρούσεων (περισσότερα στο αντίστοιχο κεφάλαιο). Για την ενημέρωση αυτού του κουτιού χρησιμοποιούνται οι μέθοδοι updateBoundingBox με όλες τις παραλλαγές των ορισμάτων που υπάρχουν.

Η μέθοδος drawObject σχεδιάζει το αντικείμενο στην θέση translation με γωνία angle ως προς τον άξονα που ορίζει το rotation. Π.χ. αν το rotation έχει τιμή (0.0f, 1.0f, 0.0f) σημαίνει ότι το αντικείμενο θα σχεδιαστεί περιστρεφόμενο ως προς τον άξονα που αντιστοιχεί στην τιμή 1.0f, δηλαδή τον y στην προκειμένη περίπτωση, με γωνία τόσων μοιρών όσων η τιμή angle. Η ενημέρωση της θέσης του αντικειμένου γίνεται με την μέθοδο updateTranslation και η ενημέρωση της περιστροφής γίνεται με την μέθοδο updateRotation. Αν θέλουμε να γίνουν παραπάνω από μία περιστροφή ως προς κάποιον άξονα υπάρχουν οι μέθοδοι updateRotation2, updateRotation3 και angle2, angle3. Η μεταβλητή health αντιστοιχεί στην ενέργεια που έχει το αντικείμενο (αν έχει) και με την μέθοδο updateHealth μπορούμε να την αλλάξουμε. Το animation vector χρησιμοποιείται για το animation και θα το περιγράψω στο αντίστοιχο κεφάλαιο. Τέλος υπάρχουν οι μέθοδοι get σε περίπτωση που θέλουμε να προσπελάσουμε κάποια μεταβλητή.

3.4. Έλεγχος συσκευών εισόδου

3.4.1. Εισαγωγή

Σκοπός όλων των παιχνιδιών είναι ο χειρισμός ενός ή πολλών χαρακτήρων ή αντικειμένων έτσι ώστε να εκπληρωθεί ο στόχος του παιχνιδιού. Ο χειρισμός αυτός γίνεται μέσω των συσκευών εισόδου χωρίς τις οποίες δεν υφίσταται και ηλεκτρονικό παιχνίδι. Στο συγκεκριμένο παιχνίδι οι συσκευές εισόδου που χρησιμοποιούνται είναι το πληκτρολόγιο και το ποντίκι μέσω των οποίων ελέγχονται δύο πράγματα: η κίνηση-περιστροφή-επίθεση του τοξότη και παράλληλα η κίνηση-περιστροφή της κάμερας. Όπως σε όλα τα παιχνίδια τρίτου προσώπου η κάμερα, δηλαδή το σημείο από όπου βλέπει ο χρήστης, πρέπει να βρίσκεται πάντα πίσω από τον χαρακτήρα του παιχνιδιού δηλαδή τον τοξότη. Αυτό σημαίνει ότι όταν μετακινείται η κάμερα πρέπει να μετακινείται και ο τοξότης σε ίση απόσταση, και όταν περιστρέφεται η κάμερα πρέπει ο τοξότης να περιστρέφεται και αυτός, έτσι ώστε να έχει την πλάτη του προς την κάμερα. Για τον έλεγχο του πληκτρολογίου και του ποντικιού χρησιμοποιείται η βιβλιοθήκη Jinput.

3.4.2. Έλεγχος κάμερας

Στο παιχνίδι η κάμερα πάντα εξαρτάται από την θέση του τοξότη. Αρχικά καλείται η μέθοδος για την αρχικοποίηση της θέσης της (την βλέπουμε παρακάτω) και έπειτα αλλάζει η θέση της πάντα βάση της θέσης του τοξότη με την μέθοδο setCamera η οποία καλείται επαναλαμβανόμενα στην μέθοδο display για να ενημερώνεται η θέση της κάμερας σε περίπτωση που ο τοξότης έχει μετακινηθεί.

```
private void initCameraPosn()
{
      cameraPosition.x = 0.0f;
      cameraPosition.y = 0.0f;
      cameraPosition.z = Z_POS; // camera posn
      viewAngle = -90.0f; // along -z axis
      xStep = Math.cos( Math.toRadians(viewAngle)); // step distances 0 -1
      zStep = Math.sin( Math.toRadians(viewAngle));
      lookAt.x = playerPosition.x; // look-at posn 0 +(100 * 0)
      lookAt.y = playerPosition.y; //0.0 + (100 * -1)
      lookAt.z = playerPosition.z;
}
```

Για την τοποθέτηση της κάμερας πρέπει να ορίσουμε δύο τριάδες συντεταγμένων. Την θέση της κάμερας (cameraPosition) και το σημείο στο οποίο θα κοιτάει (lookAt). Στην μέθοδο αρχικοποίησης η θέση της κάμερας είναι το σημείο (0.0f, 0.0f, 150.0f) αφού Z_POS = 150.0f και το σημείο στο οποίο κοιτάει είναι η θέση του τοξότη (playerPosition). Αυτό είναι απλά μια αρχικοποίηση, η θέση της

κάμερας θα αλλάξει βάση της θέσης του τοξότη, καθώς στην μέθοδο display το πρώτο που ενημερώνεται είναι η θέση της κάμερας με την μέθοδο setCamera που βλέπουμε και παρακάτω.

```
private void setCamera(GLAutoDrawable drawable, GL gl, GLU glu, float
cameraPositionX, float cameraPositionY, float cameraPositionZ, float
lookAtX, float lookAtY, float lookAtZ)
{
      // Change to projection matrix.
      gl.glMatrixMode(GL.GL_PROJECTION);
      gl.glLoadIdentity();
      gl.glClearColor(clearColor[0], clearColor[1], clearColor[2],
      clearColor[3]); // Pitch black

      // Perspective.
      aspect = drawable.getWidth() / drawable.getHeight();
      glu.gluPerspective(fovy, aspect, zNear, zFar);
      glu.gluLookAt(cameraPositionX, cameraPositionY, cameraPositionZ,
      lookAtX, 0, lookAtZ, 0, 1, 0);//camera position x y z, look at x y z


       // Change back to model view matrix.
       gl.glMatrixMode(GL.GL_MODELVIEW);
      gl.glClear(GL.GL_COLOR_BUFFER_BIT | GL.GL_DEPTH_BUFFER_BIT);
      gl.glLoadIdentity();
}
```

Η μέθοδος αυτή παίρνει σαν ορίσματα την θέση που θέλουμε να τοποθετηθεί η κάμερα (cameraPositionX, cameraPositionY, cameraPositionZ) και το σημείο στο οποίο θέλουμε να κοιτάει (lookAtX, looAtY, lookAtZ). Αρχικά επιλέγεται η μήτρα προβολής (projection matrix), δημιουργώντας μια τρισδιάστατη μήτρα η οποία απεικονίζει την περιοχή που προβάλλεται. Έπειτα εκτελείται η εντολή gluPerspective για να οριστεί η μικρότερη και μεγαλύτερη απόσταση που θα φαίνονται τα αντικείμενα από την κάμερα (zNear και zFar) και η εντολή gluLookAt για να τοποθετηθεί η κάμερα στο σημείο που έχουμε ορίσει από τις μεταβλητές των ορισμάτων της μεθόδου. Τέλος αλλάζουμε σε μήτρα προβολής μοντέλων (model view matrix). Αυτή η αλλαγή είναι απαραίτητη για να μετατραπεί ο τρισδιάστατος χώρος σε δισδιάστατα σημεία.

Πριν από την κλήση αυτής της μεθόδου πρέπει λοιπόν να ορίσουμε τις μεταβλητές cameraPosition και lookAt οι οποίες θα είναι σχετικές με την θέση του τοξότη όπως βλέπουμε παρακάτω:

```
cameraPosition.x = playerPosition.x-(float)xStep*25.0f;
cameraPosition.y = playerPosition.y - 3.0f;
cameraPosition.z = playerPosition.z-(float)zStep*25.0f;
lookAt.x = playerPosition.x;
lookAt.y = playerPosition.y;
lookAt.z = playerPosition.z;
setCamera(drawable, gl, glu, cameraPosition.x, cameraPosition.y,
cameraPosition.z, lookAt.x, lookAt.y, lookAt.z);
```

Έτσι η κάμερα θα βρίσκεται 25.0f μονάδες πίσω από τον τοξότη, 3.0f μονάδες πιο χαμηλά από τον τοξότη και θα κοιτάει προς την θέση του τοξότη (θα αναφερθώ στις μεταβλητές xStep, zStep αργότερα). Η θέση της κάμερας εξαρτάται λοιπόν από τις συντεταγμένες (playerPosition.x, playerPosition.y, playerPosition.z).

3.4.3. Έλεγχος τοξότη

Η κίνηση του τοξότη γίνεται με τα πλήκτρα w, a, s, d. Πατώντας το πλήκτρο w ο τοξότης μετακινείται μπροστά, πατώντας το s ο τοξότης πηγαίνει πίσω και πατώντας το a και d, αριστερά και δεξιά αντίστοιχα. Επίσης όταν πατάμε το w και το a ταυτόχρονα, ο τοξότης μετακινείται διαγώνια μπροστά και αριστερά κ.ο.κ..

Ο έλεγχος των πλήκτρων γίνεται από ένα νήμα (thread) το οποίο τρέχει παράλληλα με την μέθοδο display η οποία σχεδιάζει τα γραφικά του παιχνιδιού. Χρησιμοποιείται η βιβλιοθήκη Jinput για τον σκοπό αυτό. Αρχικά εγκαθίστανται οι ελεγκτές του παιχνιδιού (πληκτρολόγιο-ποντίκι) με τις παρακάτω εντολές:

```
ControllerEnvironment controllerEnvironment =
ControllerEnvironment.getDefaultEnvironment();
Controller[] controllers = controllerEnvironment.getControllers();
for (Controller controller : controllers)
{
      if (controller.getType() == Controller.Type.KEYBOARD)
      {
            keyboard = (Keyboard) controller;
      }
      else if (controller.getType() == Controller.Type.MOUSE)
      {
            mouse = (Mouse) controller;
      }
}
```

Με την μέθοδο poll ο ελεγκτής ακούει για συμβάντα πληκτρολογίου και έπειτα αποθηκεύεται η κατάσταση του κάθε πλήκτρου σε μια μεταβλητή boolean. Αν είναι πατημένο το πλήκτρο η μεταβλητή έχει τιμή true και αν όχι τότε έχει τιμή false:

```
keyboard.poll();
boolean wDown = keyboard.isKeyDown(Component.Identifier.Key.W);
boolean sDown = keyboard.isKeyDown(Component.Identifier.Key.S);
boolean aDown = keyboard.isKeyDown(Component.Identifier.Key.A);
boolean dDown = keyboard.isKeyDown(Component.Identifier.Key.D);
boolean EscDown = keyboard.isKeyDown(Component.Identifier.Key.ESCAPE);
```

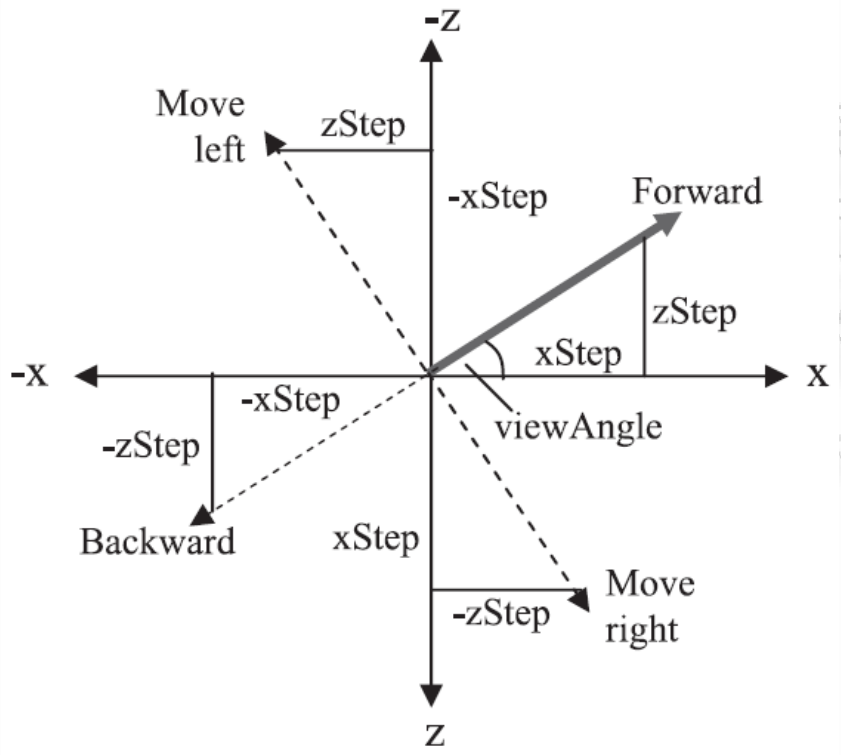
Έτσι σε περίπτωση που θέλουμε με το πάτημα του πλήκτρου Esc να γίνεται έξοδος από το παιχνίδι αρκεί ένας έλεγχος της μεταβλητής EscDown π.χ.:

```
if (EscDown)
{
      gameOver = true;
      animator.stop();
      sound.cleanUp();
      System.exit(0);
}
```

Με τον ίδιο τρόπο για την κίνηση του τοξότη ελέγχουμε τις μεταβλητές wDown, aDown, sDown, dDown. Π.χ. όταν η μεταβλητή wDown είναι true ο τοξότης θα κινείται προς τα εμπρός δηλαδή η θέση στην οποία σχεδιάζεται το μοντέλο του τοξότη (playerPosition) θα πρέπει να αλλάξει συντεταγμένες με αποτέλεσμα η καινούρια θέση του να βρίσκεται πιο μπροστά απ' ότι ήταν πριν. Η λέξη 'μπροστά' όμως δεν περιγράφει ακριβώς αυτό που θέλουμε καθώς το 'μπροστά' είναι σχετικό. Αν ο τοξότης κινούταν πάντα πάνω στους άξονες x και z και κοίταγε προς το αρνητικό άπειρο του άξονα z τότε πολύ απλά το 'μπροστά' θα ήταν κάποιες μονάδες λιγότερο στην συντεταγμένη z της θέσης του τοξότη (π.χ. αν η θέση του τοξότη ήταν (0.0f, 0.0f, 0.0f) μετά το πάτημα του w πλήκτρου η θέση μπορούσε να γίνει (0.0f, 0.0f, -10.0f) όποτε ο τοξότης έχει μετακινηθεί μπροστά 10.0 μονάδες). Επειδή όμως ο τοξότης μπορεί να περιστραφεί και δεν κινείται απλά πάνω στους άξονες θα πρέπει να γνωρίζουμε την κατεύθυνση σύμφωνα με την οποία θέλουμε να κινηθεί για να κάνουμε και το κατάλληλο βήμα (αρνητικό ή θετικό). Αν ο τοξότης κοίταγε προς το αρνητικό άπειρο του άξονα z και περιστράφηκε κατά 90^{o} τώρα θα κοιτάει προς το θετικό άπειρο του άξονα x, οπότε μια αλλαγή στην συντεταγμένη z δεν θα προκαλέσει μια μετακίνηση της θέσης του τοξότη μπροστά αλλά αριστερά. Για αυτόν τον λόγο χρησιμοποιούμε δύο μεταβλητές που μας δείχνουν την κατεύθυνση του τοξότη ανα πάσα στιγμή, τις xStep και zStep. Υπολογίζονται ως εξής:

```
xStep = Math.cos( Math.toRadians(viewAngle));
zStep = Math.sin( Math.toRadians(viewAngle));
```

Το xStep είναι το συνημίτονο της γωνίας του τοξότη ως προς τον άξονα x και το zStep είναι το ημίτονο της ίδιας γωνίας. Αντιστοιχούν στο βήμα που πρέπει να γίνει στον άξονα x (xStep) και στον άξονα z (zStep) για να αλλάξει θέση ο τοξότης προς την κατεύθυνση που έχει. Στην εικόνα 3.15 βλέπουμε πως αλλάζουν οι δύο αυτές μεταβλητές στα τέσσερα τεταρτημόρια των αξόνων x και z.



Εικόνα 3.15: Κατεύθυνση τοξότη με βάση τις μεταβλητές xStep, zStep

Έστω ότι ο τοξότης βρίσκεται στο κέντρο των αξόνων και πρέπει να μετακινηθεί εμπρός με την κατεύθυνση που βλέπουμε στο παραπάνω σχήμα, δηλαδή έχοντας γωνία viewAngle (-45^o). Για να κινηθεί εμπρός πρέπει να υπάρξει μία αύξηση του xStep (θετικό) και μια μείωση του zStep (αρνητικό). Αυτό ακριβώς μας δίνουν οι υπολογισμοί των xStep και zStep (cos(-45^o)=0.707 και sin((-45^o)=-0.707). Άρα πολλαπλασιάζοντας τους αριθμούς αυτούς που δείχνουν την αυξομείωση που πρέπει να γίνει ως προς τους άξονες x και z (xStep, zStep) με μια μεταβλητή την οποία μπορούμε να ορίσουμε σαν την ταχύτητα του τοξότη, αφού από αυτή θα εξαρτάται το μέγεθος της μεταβολής, μπορούμε να πετύχουμε την κίνηση του τοξότη. Το μόνο που πρέπει να γνωρίζουμε σε συνεχή βάση είναι η γωνία περιστροφής του τοξότη ως προς τον άξονα x.
Παρακάτω βλέπουμε πως ορίζουμε το πλήκτρο w να κινεί τον τοξότη προς τα εμπρός:

```
if (wDown && !scene.heroIsAttacking)
{
      scene.heroIsMoving = true;

      newPosition = new Tuple3(playerPosition.x+(float)xStep*SPEED,
      playerPosition.y, playerPosition.z+(float)zStep*SPEED);
      scene.player.updateBoundingBox(new Tuple3(newPosition.x,
      newPosition.y-10.0f, newPosition.z), 1.5f, 4.0f);

      if (!scene.isPlayerCollision())
            playerPosition = newPosition;
      else
```

```
        scene.player.updateBoundingBoxToFit(1.5f, 0.0f, 0.0f, 0.0f, 0.0f,
        4.0f, 0.0f);

        if (aDown)
        {
              newPosition = new Tuple3(playerPosition.x+(float)zStep*SPEED,
              playerPosition.y, playerPosition.z-(float)xStep*SPEED);
              scene.player.updateBoundingBox(new Tuple3(newPosition.x,
              newPosition.y-10.0f, newPosition.z), 1.5f, 4.0f);

              if (!scene.isPlayerCollision())
                    playerPosition = newPosition;
              else
                    scene.player.updateBoundingBoxToFit(1.5f, 0.0f, 0.0f, 0.0f,
                    0.0f, 4.0f, 0.0f);
        }
        if (dDown)
        {
              newPosition = new Tuple3(playerPosition.x-(float)zStep*SPEED,
              playerPosition.y, playerPosition.z+(float)xStep*SPEED);
              scene.player.updateBoundingBox(new Tuple3(newPosition.x,
              newPosition.y-10.0f, newPosition.z), 1.5f, 4.0f);

              if (!scene.isPlayerCollision())
                    playerPosition = newPosition;
              else


                    scene.player.updateBoundingBoxToFit(1.5f, 0.0f, 0.0f, 0.0f,
                    0.0f, 4.0f, 0.0f);
        }
}
```

Η μεταβλητή heroIsAttacking που βρίσκεται στην κλάση scene μας δείχνει αν ο τοξότης επιτίθεται οπότε την ελέγχουμε παράλληλα με το πάτημα του πλήκτρου w γιατί δεν θέλουμε όταν επιτίθεται ο τοξότης να κινείται ταυτόχρονα. Ενημερώνουμε την μεταβλητή heroIsMoving για να ξέρουμε ότι ο τοξότης κινείται. Έπειτα αυτό που κάνουμε είναι μια απλή ενημέρωση της θέσης που πρέπει να σχεδιαστεί ο τοξότης. Η καινούρια θέση που υπολογίζεται είναι η newPosition και υπολογίζεται σύμφωνα με την θέση playerPosition η οποία είναι η παλιά θέση του τοξότη, αθροίζοντας στις συντεταγμένες x και z τα xStep*SPEED και zStep*SPEED αντίστοιχα. Το SPEED είναι η ταχύτητα με την οποία κινείται ο τοξότης και όπως είπαμε και παραπάνω είναι ένας αριθμός που ορίζει το μέγεθος της μεταβολής που θα γίνει στους άξονες. Ο λόγος που δεν κάνουμε την ενημέρωση του playerPosition κατευθείαν είναι γιατί πρώτα ελέγχουμε αν υπάρχει σύγκρουση με κάποιο εμπόδιο στην θέση newPosition και μόνο αν δεν υπάρχει, η newPosition γίνεται η καινούρια playerPosition. Θα περιγράψω περαιτέρω τον έλεγχο συγκρούσεων σε επόμενο κεφάλαιο.

Επειδή θέλουμε ο τοξότης να μπορεί να κινείται και διαγώνια, δηλαδή να μπορούν να πατιούνται τα πλήκτρα w και a ή d μαζί κάνουμε μέσα στον έλεγχο του πλήκτρου w και άλλους δύο ελέγχους, έναν

για το αν έχει πατηθεί το πλήκτρο a και άλλον έναν για το αν έχει πατηθεί το πλήκτρο d. Πρέπει όμως να προσέξουμε το βήμα της μεταβολής να έχει το κατάλληλο πρόσημο. Την ίδια διαδικασία ακολουθούμε και για τα υπόλοιπα πλήκτρα. Τέλος δεν πρέπει να ξεχάσουμε να απενεργοποιήσουμε πατήματα πλήκτρων αντίθετης κατεύθυνσης δηλαδή το w με το s και το a με το d.

```
if (wDown && sDown)
{
      wDown = false;
      sDown = false;
}
if (aDown && dDown)
{
      aDown = false;
      dDown = false;
}
```

Τα πλήκτρα w, s, a και d λοιπόν ελέγχουν την θέση που σχεδιάζεται ο τοξότης και από την θέση του τοξότη επηρεάζεται και η θέση της κάμερας. Η μέθοδος display και το νήμα ελέγχου των συσκευών εισόδου εκτελούνται παράλληλα οπότε μια ενημέρωση στην θέση του τοξότη (playerPosition) αυτόματα ενημερώνει και την τοποθεσία της κάμερας αφού όπως είπαμε παραπάνω η κάμερα εξαρτάται από την μεταβλητή playerPosition. Τοποθετείται 25 μονάδες πίσω από τον τοξότη (playerPosition.x - xStep*25.0f, playerPosition.z - zStep*25.0f) και κοιτάει προς τον τοξότη (playerPosition).

Παράλληλα με τον έλεγχο συμβάντων πληκτρολογίου, το νήμα ελέγχει και για συμβάντα ποντικιού. Συμβάντα ποντικιού μπορούν να θεωρηθούν η κίνηση του και το πάτημα ενός κουμπιού. Η περιστροφή του τοξότη γίνεται από την κίνηση του ποντικιού και η επίθεση, δηλαδή η εκτόξευση ενός βέλους γίνεται πατώντας το αριστερό κουμπί του ποντικιού. Ο έλεγχος συμβάντων γίνεται όπως βλέπουμε παρακάτω:

```
mouse.poll();
float newX = mouse.getX().getPollData();
float newY = mouse.getY().getPollData();
float leftButtonPressed = mouse.getLeft().getPollData();
```

Οι μεταβλητές newX και newY περιέχουν τις συντεταγμένες του δείκτη του ποντικιού και η μεταβλητή leftButtonPressed περιέχει την τιμή 1.0 όταν το κουμπί είναι πατημένο και 0.0 διαφορετικά. Χρησιμοποιώντας την μεταβλητή newX μπορούμε να ορίσουμε την γωνία περιστροφής της κάμερας:

```
if (newX!=0.0f)
      viewAngle += newX/2;
```

Όταν η newX περιέχει την τιμή 0.0 σημαίνει ότι το ποντίκι δεν έχει κινηθεί. Σε διαφορετική περίπτωση η viewAngle αυξάνεται ή μειώνεται σε σχέση με την newX. Πρέπει να προσέξουμε ότι η viewAngle είναι η γωνία της κάμερας, όχι του μοντέλου του τοξότη. Την γωνία του τοξότη την ενημερώνουμε με την παρακάτω εντολή στην μέθοδο display:

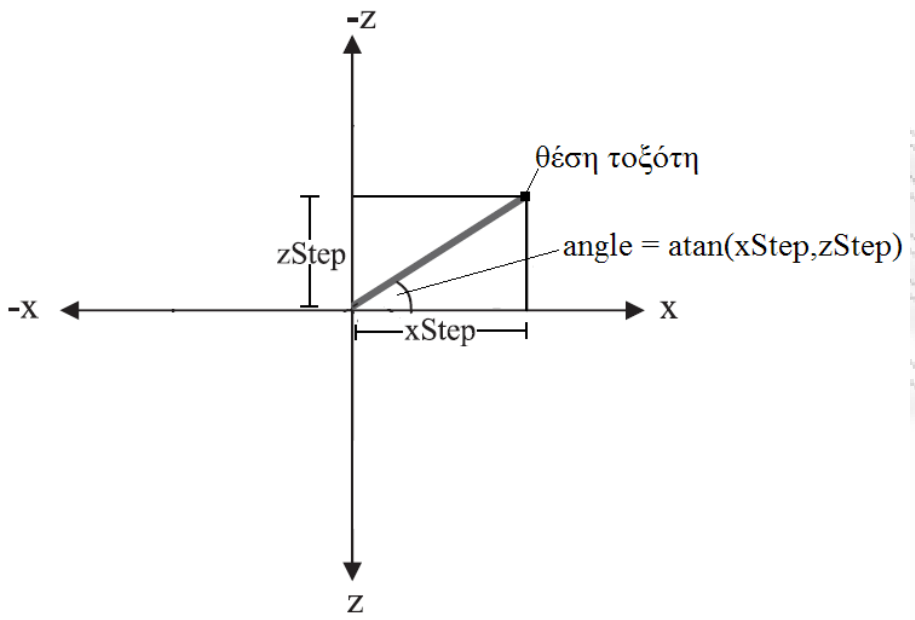
```
angle = (float)Math.toDegrees( Math.atan2(xStep,zStep) );
```

Υπολογίζει την γωνία βάση των xStep και zStep. Για να καταλάβουμε πως γίνεται αυτό μπορούμε να δούμε την εικόνα 3.16. Η γωνία του μοντέλου του τοξότη θα είναι η εφαπτομένη των δύο μεταβλητών. Άρα με το ποντίκι ουσιαστικά αλλάζει η γωνία της κάμερας και όχι του τοξότη, υπολογίζονται τα xStep και zStep βάση της γωνίας αυτής και έπειτα ενημερώνεται και η γωνία του μοντέλου του τοξότη. Υπενθυμίζω ότι οι μεταβλητές xStep και zStep υπολογίζονται ως το συνημίτονο και το ημίτονο της γωνίας viewAngle. Έτσι λοιπόν η γωνία της κάμερας ορίζει και την γωνία του τοξότη σε αντίθεση με τον τοξότη που ορίζει την θέση της κάμερας.

Καλύψαμε την περιστροφή του μοντέλου του τοξότη. Για την επίθεση τώρα πρέπει να χειριστούμε το συμβάν πατήματος του αριστερού κουμπιού του ποντικιού. Ξέρουμε ότι όταν το πλήκτρο πατιέται η μεταβλητή leftButtonPressed γίνεται 1.0. Το ένα κλικ όμως που κάνουμε εμείς στο κουμπί κάνει την μεταβλητή αυτή σε χρόνο εκτέλεσης του νήματος να γίνεται 1.0 για πολλές επαναλήψεις του νήματος. Με άλλα λόγια πρέπει να βάλουμε μια χρονοκαθυστέρηση ανάμεσα σε διαδοχικές επιθέσεις του τοξότη. Αυτό μπορεί να γίνει πολύ εύκολα χρησιμοποιώντας δύο μεταβλητές timeStamp και lastTimeStamp που μετράνε τον χρόνο συστήματος ως εξής:

```
timeStamp = System.currentTimeMillis();
boolean ctlDown = (leftButtonPressed==1.0f && timeStamp > (lastTimeStamp +
300) && !scene.heroIsMoving);

if (ctlDown)
{
      scene.shoot = true;
      scene.heroIsAttacking = true;
}
```

Η μεταβλητή ctlDown γίνεται true όταν έχει πατηθεί το αριστερό κουμπί του ποντικιού (leftButtonPressed = 1.0f), όταν έχουν περάσει 300msec από το τελευταίο πάτημα του ίδιου κουμπιού και όταν ο τοξότης δεν κινείται. Όταν γίνουν όλα αυτά οι μεταβλητές shoot και heroIsAttacking γίνονται true. Η lastTimeStamp πρέπει να προσέξουμε να έχει αρχικοποιηθεί με τον χρόνο του συστήματος (lastTimeStamp = System.currentTimeMillis();) έξω από τον βρόχο ελέγχου του ποντικιού καθώς και να ενημερώνεται πάλι (lastTimeStamp = timeStamp;) όταν έχει εκτελεστεί μια εκτόξευση βέλους.



Εικόνα 3.16: Υπολογισμός γωνίας τοξότη βάση xStep και zStep

Από την στιγμή που θα πατηθεί το κουμπί για την επίθεση, θα εκτοξευθεί ένα βέλος. Το βέλος όμως πρέπει να εμφανιστεί μόλις το animation της επίθεσης του τοξότη έχει σχεδόν τελειώσει, δηλαδή ο τοξότης βρίσκεται στο σημείο που έχει πιάσει το βέλος, το έχει τεντώσει και το αφήνει από το τόξο. Στον τρόπο που γίνεται το animation θα αναφερθώ αργότερα. Προς το παρόν αρκεί να γνωρίζουμε ότι η μεταβλητή heroAttacked είναι true όταν το animation του τοξότη βρίσκεται στο σημείο που πρέπει να εμφανιστεί το βέλος. Άρα ελέγχοντας τις μεταβλητές shoot, η οποία γίνεται false μόνο όταν το βέλος έχει εμφανιστεί, και heroAttacked ξέρουμε πότε πρέπει να δημιουργήσουμε και να σχεδιάσουμε ένα βέλος όπως και βλέπουμε παρακάτω:

```
if (scene.shoot && scene.heroAttacked)
{
      lastTimeStamp = timeStamp;
      scene.arrow = new Object3D(scene.arrowModel, new
      Tuple3(playerPosition.x, playerPosition.y-6.0f, playerPosition.z), new
      Tuple3(0.0f, 1.0f, 0.0f), angle+90.0f, 0.3f, (float)xStep,
      (float)zStep);
      if (scene.arrows.size()==10)
            arrayFull = true;
      if (arrayFull)
      {
            scene.arrows.set(index, scene.arrow);
            index++;
            if (index==10)
            index=0;

      }
```

```
      else
            scene.arrows.add(scene.arrow);

      scene.arrowAppeared = true;
      scene.shoot = false;
      scene.heroAttacked = false;
}
```

Αυτό που γίνεται είναι το εξής: δημιουργείται το μοντέλο του βέλους και σχεδιάζεται αρχικά στην ίδια θέση του τοξότη και στο ύψος των χεριών του. Επίσης σχεδιάζεται κάθετα στο μοντέλο του τοξότη και κατά την δημιουργία του αποθηκεύονται οι τιμές xStep και zStep ώστε να γνωρίζει προς ποια κατεύθυνση θα συνεχίσει. Μετά από την εκτόξευση δέκα βελών, το πρώτο σταματά να σχεδιάζεται στον χώρο. Για αυτόν τον λόγο ο vector arrows μόλις έχει μέγεθος ίσο με δέκα αρχίζει να αποθηκεύει τα βέλη πάλι από την πρώτη θέση του αντικαθιστώντας τα υπάρχοντα.

Παράλληλα με όλα αυτά που είπαμε για τον χειρισμό του τοξότη, θα πρέπει να έχουν γραφτεί και οι κατάλληλες εντολές για την σχεδίαση του τοξότη και των βελών. Το μοντέλο του τοξότη θα πρέπει να σχεδιάζεται συνεχώς στην μέθοδο display στην θέση playerPosition με γωνία angle ως εξής:

```
player.updateTranslation(new Tuple3(GameRenderer.playerPosition.x,
GameRenderer.playerPosition.y - 10.0f, GameRenderer.playerPosition.z));
player.updateRotation(new Tuple3(0.0f, 1.0f, 0.0f), GameRenderer.angle);
player.drawObject(gl);
```

Επίσης θα πρέπει να σχεδιάζονται τα βέλη και να ενημερώνεται η θέση τους συνεχώς:

```
for (Object3D arrow : arrows)
{
      arrow.drawObject(gl);
}
```

Η ενημέρωση της θέσης των βελών γίνεται από την μέθοδο updateArrowVectors την οποία θα αναλύσουμε καλύτερα στο κεφάλαιο για τον έλεγχο συγκρούσεων. Αυτό που χρειάζεται να κάνει μέχρι στιγμής η μέθοδος αυτή, είναι να ενημερώνει την θέση κάθε βέλους που είναι αποθηκευμένο στον vector arrows ως εξής:

```
arrow.translation.z += arrow.zStep*ARROW_SPEED;
arrow.translation.x += arrow.xStep*ARROW_SPEED;
```

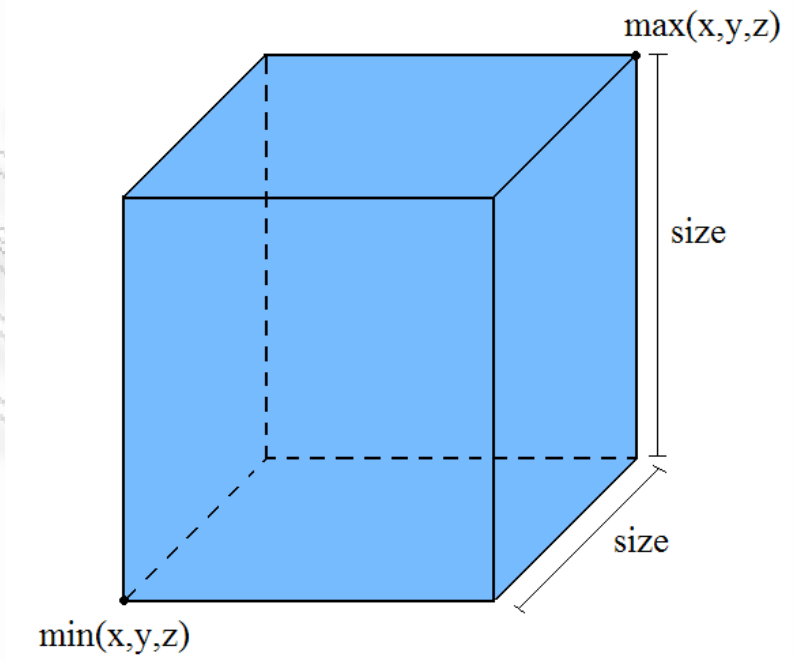
Η κίνηση δηλαδή των βελών γίνεται όπως γίνεται και η κίνηση του τοξότη. Αποθηκεύονται οι μεταβλητές xStep και zStep για κάθε βέλος καθώς αυτές δεν πρόκειται να αλλάξουν. Κάθε βέλος από την στιγμή που εμφανίζεται έχει προκαθορισμένη πορεία. Οι μεταβλητές αυτές δείχνουν την κατεύθυνση που πρέπει να ακολουθήσει το βέλος και προστίθενται πολλαπλασιασμένες με έναν αριθμό (ταχύτητα βέλους) σε κάθε επανάληψη της μεθόδου ενημέρωσης του βέλους με τις συντεταγμένες x και z του βέλους. Έτσι λοιπό την επόμενη φορά το βέλος θα σχεδιαστεί στην ενημερωμένη θέση που υπολογίστηκε.

3.5. Ανίχνευση συγκρούσεων

3.5.1. Εισαγωγή

Πολύ σημαντικό στην υλοποίηση του παιχνιδιού είναι να γνωρίζουμε πότε γίνεται κάποια σύγκρουση ενός τρισδιάστατου αντικειμένου με κάποιο άλλο. Με τον όρο σύγκρουση εννοούμε ότι το τρισδιάστατο μοντέλο έχει εισέλθει στον χώρο του άλλου μοντέλου. Πρέπει να γνωρίζουμε την ύπαρξη σύγκρουσης για δύο λόγους. Πρώτον για να ξέρουμε πότε το βέλος έχει χτυπήσει έναν παίκτη ώστε να μειωθεί η ενέργεια του και δεύτερον για να ξέρουμε πότε ένας παίκτης έχει εισέλθει ή καλύτερα πρόκειται να εισέλθει σε μια περιοχή που δεν επιτρέπεται, έτσι ώστε να το αποφύγουμε. Με αυτόν τον τρόπο μπορούμε να ορίσουμε κάποια αντικείμενα να λειτουργούν σαν εμπόδια, με αποτέλεσμα να μην επιτρέπεται η κίνηση μέσα τους και το βέλος να σταματάει μόλις υπάρξει επαφή με αυτά.

Για τον έλεγχο σύγκρουσης ακολουθείται η εξής μεθοδολογία. Γύρω από κάθε αντικείμενο ανα πάσα στιγμή σχεδιάζεται ένας κύβος ο οποίος το εμπεριέχει. Αυτό το ‘κουτί’ ουσιαστικά περιβάλλει το αντικείμενο και υποδηλώνει τα όρια του. Έτσι είναι πολύ πιο εύκολο να ελέγξουμε πότε μια πλευρά ενός κύβου εισέρχεται μέσα σε έναν άλλον κύβο, παρά να ελέγξουμε πότε μια επιφάνεια ενός μοντέλου έρχεται σε επαφή με ένα άλλο μοντέλο, εφ’ όσον τα μοντέλα δεν έχουν προκαθορισμένο σχήμα και πολλές γωνίες. Το ‘κουτί’ αυτό κάθε αντικειμένου είναι αόρατο κατά την εκτέλεση του παιχνιδιού δηλαδή δεν σχεδιάζεται. Η δημιουργία του και η ανίχνευση της σύγκρουσης γίνεται θεωρητικά και δεν πραγματοποιείται με σχεδίαση του κουτιού. Το κουτί εξαρτάται από δύο σημεία στο χώρο, δύο αντικείμενα τύπουTuple3, τα max και min. Αυτές αντιστοιχούν σε δύο κορυφές ενός κύβου οι οποίες και τον δημιουργούν (εικόνα 3.17).



Εικόνα 3.17: Τα σημεία min και max στο κουτί που περιβάλλει ένα αντικείμενο

Ο έλεγχος σύγκρουσης γίνεται με βάση αυτές τις δύο μεταβλητές όπως βλέπουμε παρακάτω από την μέθοδο isCollide της κλάσης BoundingBox:

```
public boolean isCollide(BoundingBox otherBox)
{
      return otherBox != this &&
      otherBox.max.x > min.x &&
      otherBox.min.x < max.x &&
      otherBox.max.y > min.y &&
      otherBox.min.y < max.y &&
      otherBox.max.z > min.z &&
      otherBox.min.z < max.z;
}
```

Η μέθοδος αυτή παίρνει σαν όρισμα ένα άλλο αντικείμενο τύπου BoundingBox το οποίο αντιστοιχεί σε ένα άλλο κουτί ενός άλλου μοντέλου. Αυτό το αντικείμενο θα έχει κάποιες τιμές στις συντεταγμένες max και min σύμφωνα με τις οποίες γίνεται και ο έλεγχος. Αν η τιμή της συντεταγμένης x του σημείου max είναι μεγαλύτερη από την τιμή της συντεταγμένης x του σημείου min του άλλου κουτιού σημαίνει ότι το ένα ορθογώνιο βρίσκεται εντός του άλλου. Το ίδιο συμβαίνει για όλες τις υπόλοιπες περιπτώσεις που βλέπουμε στην μέθοδο. Αν όλες οι συγκρίσεις δώσουν αποτέλεσμα false, τότε δεν υπάρχει και σύγκρουση. Επομένως επιστρέφεται με την κλήση της παραπάνω μεθόδου η τιμή false αν δεν υπάρχει σύγκρουση και true αν υπάρχει. Σε περίπτωση που θέλουμε να σχεδιαστεί το κουτί που περιβάλλει κάποιο αντικείμενο χρησιμοποιούμε την παρακάτω μέθοδο η οποία βρίσκεται στην κλάση Object3D.

```
public void drawBoundingBox(GL gl)
{

      gl.glPushMatrix();
      {
            gl.glMaterialfv(GL.GL_FRONT, GL.GL_EMISSION, this.box.color, 0);
            gl.glMaterialfv(GL.GL_FRONT, GL.GL_AMBIENT, this.box.color, 0);
            gl.glMaterialfv(GL.GL_FRONT, GL.GL_DIFFUSE, this.box.color, 0);
            gl.glBegin(GL.GL_LINES);
            {
                  gl.glVertex3f(this.box.min.x, this.box.min.y,
                  this.box.min.z);
                  gl.glVertex3f(this.box.max.x, this.box.min.y,
                  this.box.min.z);

                  gl.glVertex3f(this.box.max.x, this.box.min.y,
                  this.box.min.z);
                  gl.glVertex3f(this.box.max.x, this.box.max.y,
                  this.box.min.z);

                  gl.glVertex3f(this.box.max.x, this.box.max.y,
                  this.box.min.z);
```

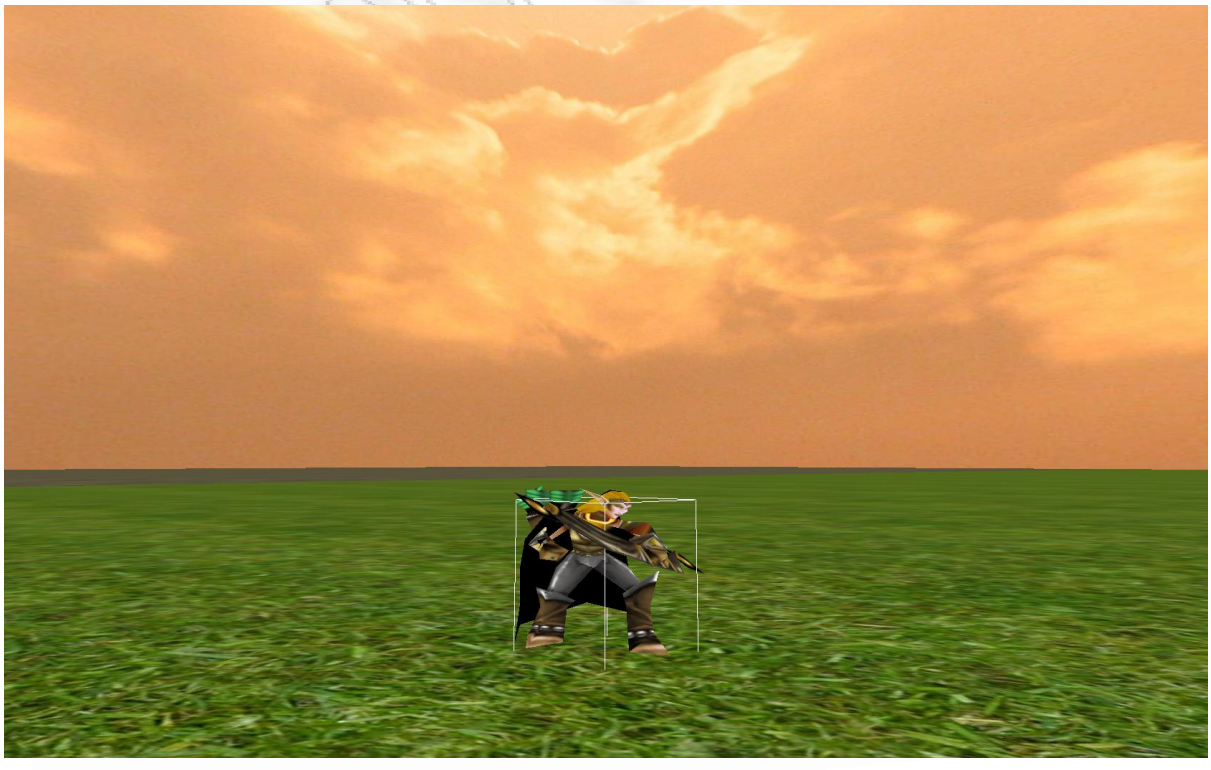
```
        gl.glVertex3f(this.box.min.x, this.box.max.y,
        this.box.min.z);


        gl.glVertex3f(this.box.min.x, this.box.max.y,
        this.box.min.z);
        gl.glVertex3f(this.box.min.x, this.box.min.y,
        this.box.min.z);

        gl.glVertex3f(this.box.min.x, this.box.min.y,
        this.box.min.z);

        gl.glVertex3f(this.box.min.x, this.box.min.y,
        this.box.max.z);

        gl.glVertex3f(this.box.min.x, this.box.min.y,
        this.box.max.z);
        gl.glVertex3f(this.box.min.x, this.box.max.y,
        this.box.max.z);

        gl.glVertex3f(this.box.min.x, this.box.max.y,
        this.box.max.z);
        gl.glVertex3f(this.box.min.x, this.box.max.y,
        this.box.min.z);

        gl.glVertex3f(this.box.max.x, this.box.min.y,
        this.box.min.z);
        gl.glVertex3f(this.box.max.x, this.box.min.y,
        this.box.max.z);

        gl.glVertex3f(this.box.max.x, this.box.min.y,
        this.box.max.z);
        gl.glVertex3f(this.box.max.x, this.box.max.y,
        this.box.max.z);

        gl.glVertex3f(this.box.max.x, this.box.max.y,
        this.box.max.z);
        gl.glVertex3f(this.box.max.x, this.box.max.y,
        this.box.min.z);

        gl.glVertex3f(this.box.min.x, this.box.min.y,
        this.box.max.z);
        gl.glVertex3f(this.box.max.x, this.box.min.y,
        this.box.max.z);

        gl.glVertex3f(this.box.min.x, this.box.max.y,
        this.box.max.z);
        gl.glVertex3f(this.box.max.x, this.box.max.y,
        this.box.max.z);
    }
```

```
        }
        gl.glEnd();
}
```

Βλέπουμε το αποτέλεσμα σχεδίασης του κουτιού που περιβάλλει το μοντέλο του τοξότη στην εικόνα 3.18. Για την ενημέρωση του κουτιού που περιβάλλει ένα αντικείμενο υπάρχει μια πληθώρα μεθόδων στην κλάση Object3D (updateBoundingBox). Χρειάζονται γιατί όταν κινείται ένα αντικείμενο πρέπει να ενημερώνεται και το κουτί που το περιβάλλει σύμφωνα με την νέα θέση του. Πιο αναλυτικά, ενημερώνονται οι δύο μεταβλητές max και min σύμφωνα με την μεταβλητή translation της κλάσης Object3D που δηλώνει την θέση του αντικειμένου.

Επίσης υπάρχει η δυνατότητα με την μέθοδο updateBoundingBoxToFit να αλλάξουμε το κουτί ενός αντικειμένου έτσι ώστε να ταιριάζει όσο δυνατόν καλύτερα με τα όρια του αντικειμένου. Δίνουμε τέσσερις παραπάνω τιμές εκτός του μεγέθους του κουτιού (το οποίο ισούται με το μήκος κάθε πλευράς σε περίπτωση που το κουτί είναι κύβος) και έτσι δημιουργείται ένα ορθογώνιο για να περιβάλλει αντικείμενα στα οποία ο κύβος θα ήταν πολύ μεγάλος για το σχήμα και το μέγεθος τους.

```
public void updateBoundingBoxToFit(float size, float x1, float z1, float x2,
float z2)
{
      min = new Tuple3(translation.x - size + x1, translation.y - size,
      translation.z - size + z1);
      max = new Tuple3(translation.x + size + x2, translation.y + size,
      translation.z + size + z2);
      box = new BoundingBox(min, max);
}
```



Εικόνα 3.18:Σχεδίαση μοντέλου τοξότη με το κουτί που τον περιβάλλει για ανίχνευση συγκρούσεων

3.5.2. Ανίχνευση σύγκρουσης τοξότη με εμπόδιο

Στο παιχνίδι, όπως προανέφερα γίνονται δύο ανιχνεύσεις συγκρούσεων, ο ένας από αυτούς είναι μεταξύ του τοξότη και ενός εμποδίου. Αυτό αφορά την κίνηση του τοξότη και τα σημεία στα οποία επιτρέπεται να κινηθεί. Το κουτί που περιβάλλει τον τοξότη θα πρέπει να ενημερώνεται σε κάθε ανανέωση της θέσης του τοξότη και να ελέγχεται αν η καινούρια θέση του βρίσκεται εντός των ορίων κάποιου εμποδίου. Ο έλεγχος αυτός γίνεται στον έλεγχο πατήματος των πλήκτρων κίνησης:

```
if (sDown && !scene.heroIsAttacking)
{
      scene.heroIsMoving = true;

      newPosition = new Tuple3(playerPosition.x-(float)xStep*SPEED,
      playerPosition.y, playerPosition.z-(float)zStep*SPEED);
      scene.player.updateBoundingBox(new Tuple3(newPosition.x,
      newPosition.y-10.0f, newPosition.z), 1.5f, 4.0f);

      if (!scene.isPlayerCollision())
            playerPosition = newPosition;
      else
            scene.player.updateBoundingBoxToFit(1.5f, 0.0f, 0.0f, 0.0f, 0.0f,
            4.0f, 0.0f);
}
```

Είναι σημαντικό ο έλεγχος σύγκρουσης με κάποιο εμπόδιο να γίνει πριν από την σχεδίαση του αντικειμένου στην καινούρια θέση του, καθώς αν σχεδιαστεί και μετά ο έλεγχος δώσει αποτέλεσμα true, το αντικείμενο, στην προκειμένη περίπτωση ο τοξότης, θα κολλήσει εντός του εμποδίου. Όπως βλέπουμε παραπάνω η νέα θέση του τοξότη υπολογίζεται, ελέγχεται αν υπάρχει σύγκρουση με κάποιο εμπόδιο και μόνο αν δεν υπάρχει, αλλάζει η θέση σχεδίασης του τοξότη με την υπολογισθείσα θέση. Αλλιώς το κουτί επιστρέφει στην θέση που είχε πριν από τον έλεγχο σύγκρουσης.

Για την ομαδοποίηση όλων των αντικειμένων που λειτουργούν ως εμπόδια έχω ορίσει έναν vector με όνομα allObstacles στην κλάση Scene, στον οποίο εισάγονται αυτά τα αντικείμενα και έτσι ο έλεγχος ανάγεται στον έλεγχο σύγκρουσης του κουτιού του τοξότη με τα κουτιά όλων των περιεχομένων αυτού του vector.

```
public boolean isPlayerCollision()
{
      for (Object3D model : allObstacles)
      {
            if (player.box.isCollide(model.box))
                  return true;
      }
      return false;
}
```

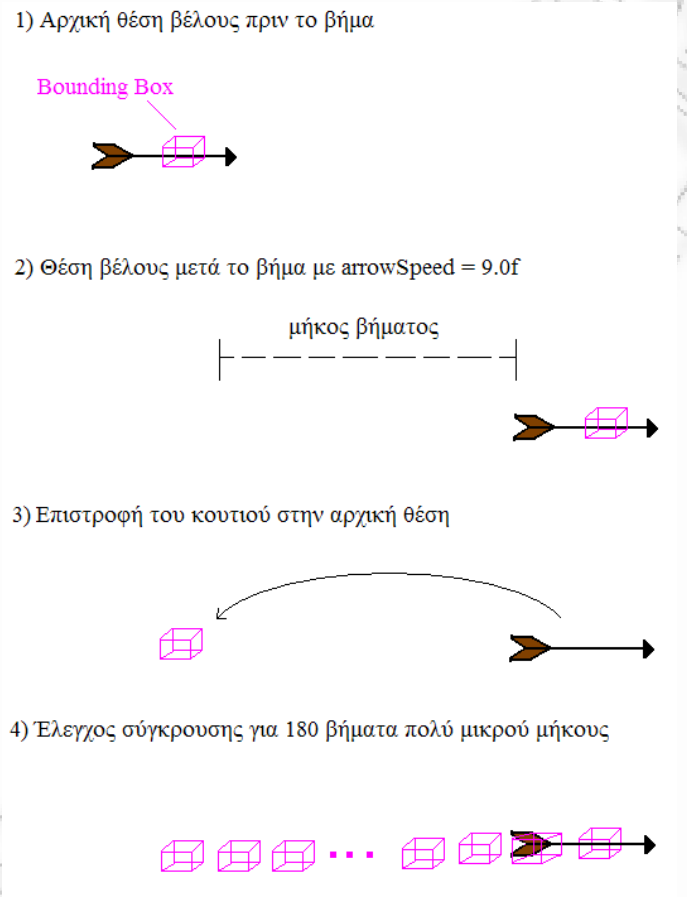
3.5.3. Ανίχνευση σύγκρουσης βέλους με τοξότη

Ο άλλος έλεγχος σύγκρουσης γίνεται μεταξύ ενός αντικειμένου και ενός βέλους. Σε περίπτωση που αυτό το αντικείμενο είναι ο τοξότης η ενέργεια του τοξότη θα πρέπει να μειωθεί και το βέλος να συνεχίσει την κανονική πορεία του. Αλλιώς αν το αντικείμενο είναι εμπόδιο το βέλος πρέπει να σταματήσει την κίνηση του στο σημείο που έγινε η σύγκρουση. Παρακάτω βλέπουμε την μέθοδο που χρησιμοποιείται για τον έλεγχο σύγκρουσης μεταξύ ενός βέλους και του αντίπαλου τοξότη:

```
public boolean isArrowWithPlayerCollision(Object3D arrow)
{
      float arrowSpeed = GameRenderer.ARROW_SPEED;
      arrow.updateBoundingBox(new Tuple3(arrow.translation.x +
      arrow.xStep*2.0f-arrow.xStep*arrowSpeed, arrow.translation.y,
      arrow.translation.z + arrow.zStep*2.0f-arrow.zStep*arrowSpeed), 0.1f,
      0.1f, -0.4f, 0.6f, 0.1f, 0.1f);
      arrowSpeed = 0.05f;
      for (int i=1;i<181;i++)
      {
            arrow.updateBoundingBox(new Tuple3(arrow.translation.x +
            arrow.xStep*2.0f+i*arrow.xStep*arrowSpeed, arrow.translation.y,
            arrow.translation.z + arrow.zStep*2.0f+i*arrow.zStep*arrowSpeed),
            0.1f, 0.1f, -0.4f, 0.6f, 0.1f, 0.1f);
            if (arrow.box.isCollide(player.box))
                  return true;
      }
      arrow.updateBoundingBox(new Tuple3(arrow.translation.x +
      arrow.xStep*2.0f+arrow.xStep*GameRenderer.ARROW_SPEED,
      arrow.translation.y, arrow.translation.z +
      arrow.zStep*2.0f+arrow.zStep*GameRenderer.ARROW_SPEED), 0.1f, 0.1f, -
      0.4f, 0.6f, 0.1f, 0.1f);
      return false;
}
```

Η μέθοδος επιστρέφει την τιμή true αν υπάρχει σύγκρουση και false διαφορετικά. Σαν όρισμα δίνεται το arrow στο οποίο θέλουμε να γίνει ο έλεγχος. Εδώ πρέπει να προσέξουμε το εξής: η μέθοδος isCollide κάνει απλά έλεγχο αν δύο κουτιά με συγκεκριμένες θέσεις στον χώρο έχουν έλθει σε επαφή. Η παραπάνω μέθοδος καλεί την isCollide επαναλαμβανόμενα για κάθε νέα θέση στην οποία σχεδιάζεται το βέλος. Έστω λοιπόν το βέλος έχει ταχύτητα 9.0 (arrowSpeed) και το xStep ισούται με 1.0 και το zStep με 0.0, αυτό σημαίνει ότι το βέλος θα κάνει ένα βήμα 9.0*1.0 = 9.0 μονάδες στον άξονα x. Θα σχεδιαστεί λοιπόν στην αρχική θέση που ήταν, θα γίνει ο έλεγχος αν έχει συγκρουστεί με τον τοξότη και έπειτα θα σχεδιαστεί στην νέα θέση (9.0 μονάδες βήμα) και θα γίνει πάλι ο έλεγχος σε αυτήν την θέση αν έχει γίνει σύγκρουση. Τι γίνεται όμως σε περίπτωση που το κουτί του τοξότη έχει μήκος 8.0 μονάδες και τύχει να βρεθεί στον χώρο εντός των δύο βημάτων της κίνησης του βέλους; Το βέλος θα προσπεράσει τον τοξότη, οι έλεγχοι θα γίνουν πριν και μετά την σύγκρουση και έτσι η σύγκρουση δεν θα ανιχνευθεί ενώ στην πραγματικότητα θα έχει συμβεί. Για να λύσουμε αυτό το θέμα

θα πρέπει ο έλεγχος σύγκρουσης (δηλαδή η κλήση της μεθόδου isCollide) να γίνεται για βήματα που κάνει το βέλος τα οποία είναι πολύ μικρά και δεν αφήνουν το περιθώριο στο βέλος να προσπεράσει το κουτί του τοξότη. Αν όμως αλλάξουμε την μεταβλητή arrowSpeed για να μειωθεί το βήμα της κίνησης τότε αυτό θα προκαλέσει την μείωση της ταχύτητας του βέλους το οποίο θα πηγαίνει πολύ αργά. Όπως βλέπουμε και στην παραπάνω μέθοδο αυτό λύνεται με τον ακόλουθο τρόπο (εικόνα 3.19).



Εικόνα 3.19: Ανίχνευση σύγκρουσης βέλους για κάθε βήμα κίνησης που εκτελεί

Η μέθοδος αυτή καλείται κάθε φορά που το βέλος έχει κάνει ένα βήμα στον χώρο. Το κουτί που περιβάλλει το βέλος επιστρέφει στην θέση στην οποία ήταν πριν το βήμα. Αλλάζουμε την ταχύτητα του βέλους (οπότε και του κουτιού) από 9.0 σε 0.05 και για 9.0/0.05 = 180 επαναλήψεις (πολύ μικρά βήματα) ενημερώνουμε την θέση του κουτιού και ελέγχουμε για σύγκρουση. Αν έχει σημειωθεί σύγκρουση η μέθοδος θα επιστρέψει true αλλιώς το κουτί θα προχωρήσει στην θέση που ήταν. Με άλλα λόγια σπάμε το βήμα που έκανε το βέλος σε 180 βήματα πολύ μικρού μεγέθους και κάνουμε ξεχωριστό έλεγχο για κάθε βήμα.

3.5.4. Ανίχνευση σύγκρουσης βέλους με εμπόδιο

Όσον αφορά τον έλεγχο της σύγκρουσης μεταξύ ενός βέλους και ενός οποιοδήποτε εμποδίου, χρησιμοποιείται η παρακάτω μέθοδος έτσι ώστε μόλις συμβεί μια σύγκρουση το βέλος να σταματήσει να κινείται.

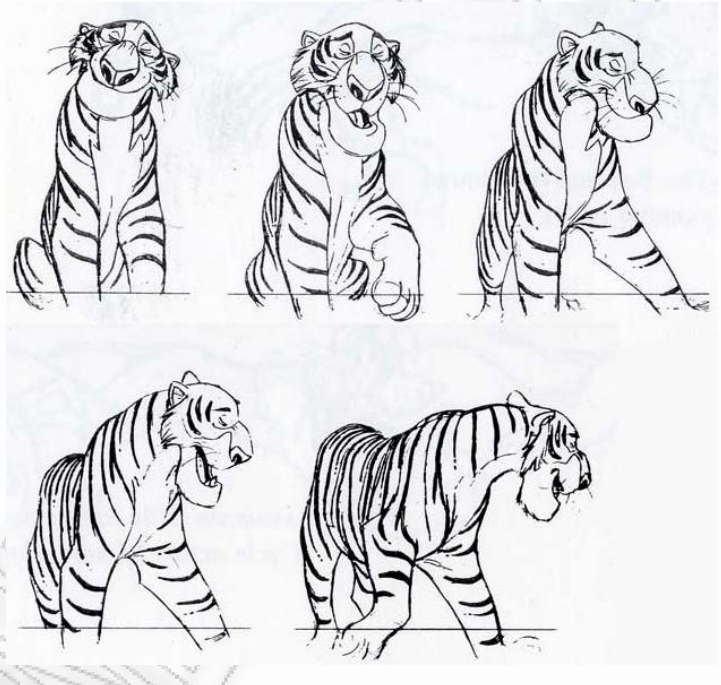
```
public Tuple3 checkArrowWithObstacleCollision(Object3D arrow)
{
      float arrowSpeed = GameRenderer.ARROW_SPEED;
      float counter = 0.0f;

      for (Object3D model : allObstacles)
      {
            arrowSpeed = GameRenderer.ARROW_SPEED;
            arrow.updateBoundingBox(new Tuple3(arrow.translation.x +
            arrow.xStep*2.0f-arrow.xStep*arrowSpeed, arrow.translation.y,
            arrow.translation.z + arrow.zStep*2.0f-arrow.zStep*arrowSpeed),
            0.1f, 0.1f, -0.4f, 0.6f, 0.1f, 0.1f);
            arrowSpeed = 0.05f;

            for (int i=1;i<181;i++)
            {
                  counter = (float)i;
                  arrow.updateBoundingBox(new Tuple3(arrow.translation.x +
                  arrow.xStep*2.0f+i*arrow.xStep*arrowSpeed,
                  arrow.translation.y, arrow.translation.z +
                  arrow.zStep*2.0f+i*arrow.zStep*arrowSpeed), 0.1f, 0.1f, -
                  0.4f, 0.6f, 0.1f, 0.1f);
                  if (arrow.box.isCollide(model.box))
                  {
                        arrow.moving = false;
                        return new
                        Tuple3(arrow.translation.x+arrow.xStep*counter*arrowSp
                        eed, arrow.translation.y,
                        arrow.translation.z+arrow.zStep*counter*arrowSpeed);
                  }

            }
            arrow.updateBoundingBox(new Tuple3(arrow.translation.x +
            arrow.xStep*2.0f+arrow.xStep*GameRenderer.ARROW_SPEED,
            arrow.translation.y, arrow.translation.z +
            arrow.zStep*2.0f+arrow.zStep*GameRenderer.ARROW_SPEED), 0.1f,
            0.1f, -0.4f, 0.6f, 0.1f, 0.1f);

      }
      return new Tuple3(0.0f, 0.0f, 0.0f);
```

Όπως είπαμε όλα τα αντικείμενα που θέλουμε να λειτουργούν ως εμπόδια τα αποθηκεύουμε στον vector allObstacles για να μπορούμε να τα διαχειριζόμαστε. Στην παραπάνω μέθοδο, λοιπόν, κάνουμε ακριβώς ότι και στην μέθοδο ελέγχου σύγκρουσης βέλους με τοξότη μόνο που αυτή τη φορά εκτελούμε τον έλεγχο όχι μεταξύ του κουτιού του player και του κουτιού του βέλους μόνο, αλλά μεταξύ του κουτιού κάθε εμποδίου ξεχωριστά και του κουτιού του βέλους. Κάνουμε δηλαδή ένα for για όλα τα περιεχόμενα του vector allObstacles. Επίσης σε περίπτωση που ο έλεγχος της σύγκρουσης γίνει με επιτυχία (`arrow.box.isCollide(model.box)`) επιστρέφεται η θέση στην οποία έγινε η σύγκρουση έτσι ώστε να γνωρίζουμε σε ποιο σημείο θα σταματήσει να κινείται το βέλος.

3.6. Animation

3.6.1. Εισαγωγή

Η επιτυχία ενός παιχνιδιού εξαρτάται σε μεγάλο ποσοστό από τον βαθμό που προσομοιώνει την πραγματικότητα και σε αυτό παίζει ρόλο το animation. Η λέξη animation μπορεί να οριστεί ως η εμψύχωση-ζωντάνεμα ενός αντικειμένου απεικονιζόμενο σε μια εικόνα. Το αντικείμενο αυτό κινείται προκαλώντας την ψευδαίσθηση ότι είναι ζωντανό. Η κίνηση πετυχαίνεται με την συνεχόμενη αλλαγή της εικόνας πολύ γρήγορα έτσι ώστε το μάτι να αντιλαμβάνεται τα σημεία τα οποία έχουν διαφοροποιηθεί από την προηγούμενη εικόνα ως κίνηση.



Εικόνα 3.20: Σταδιακή αλλαγή της εικόνας για δημιουργία animation

Στην εικόνα 3.20 βλέπουμε την σταδιακή αλλαγή της εικόνας με την οποία μπορούμε να δημιουργήσουμε ένα animation. Με την γρήγορη εναλλαγή των πέντε αυτών σχεδίων της τίγρης, προκαλείται η ψευδαίσθηση ότι η τίγρης κινείται. Εννοείται ότι η εικόνα της τίγρης πρέπει να σχεδιάζεται στο ίδιο ακριβώς σημείο. Με τον ίδιο τρόπο γίνεται και το animation στα ηλεκτρονικά παιχνίδια και με τον ίδιο τρόπο θα γίνει το animation του τοξότη στο συγκεκριμένο παιχνίδι.

3.6.2. Animation μοντέλου τοξότη

Στο μοντέλο του τοξότη γίνονται τρείς διαφορετικές κινήσεις, ανάλογα στην κατάσταση στην οποία βρίσκεται. Αν δεν κινείται και δεν επιτίθεται, χρησιμοποιούνται τα 3d μοντέλα που βρίσκονται στον

φάκελο models\Hero\StandModels, αν κινείται τα μοντέλα του φακέλου models\Hero\MoveModels και αν επιτίθεται τα μοντέλα του models\Hero\AttModels. Σε κάθε έναν από αυτούς τους φακέλους έχει αποθηκευτεί ένα πλήθος obj μοντέλων σε κάθε ένα από τα οποία το μοντέλο έχει το στιγμιότυπο της κίνησης την οποία θέλουμε να κάνει το μοντέλο. Αν σχεδιαστούν διαδοχικά με γρήγορο ρυθμό θα δημιουργηθεί το animation. Οι μεταβλητές heroIsAttacking και heroIsMoving μας δείχνουν πότε ο τοξότης επιτίθεται και πότε κινείται αντίστοιχα. Επομένως με τους παρακάτω έλεγχους πετυχαίνουμε το animation για κάθε περίπτωση:

```
if (!heroIsMoving && !heroIsAttacking)
{
	frameCounterMov=0;
	frameCounterAtt=0;

	player.updateAnimationVector(1);
	player.updateIndex(frameCounterSt);
	player.drawFrameModel(standModels, frameCounterSt, gl);

	frameCounterSt++;
	if (frameCounterSt == standModels.size())
	{
		frameCounterSt = 0;
	}
}
else if (heroIsAttacking && !heroIsMoving)
{
	frameCounterMov = 0;
	frameCounterSt = 0;

	player.updateIndex(frameCounterAtt);
	player.updateAnimationVector(2);
	player.drawFrameModel(attackModels, frameCounterAtt, gl);

	frameCounterAtt++;
	if (frameCounterAtt == attackModels.size()-3)
		heroAttacked = true;

	if (frameCounterAtt == attackModels.size())
	{
		heroIsAttacking = false;
		frameCounterAtt=0;
	}
```

```
}
else if (heroIsMoving && !heroIsAttacking)
{
	frameCounterAtt = 0;
	frameCounterSt = 0;

	player.updateIndex(frameCounterMov);
	player.updateAnimationVector(3);


	player.drawFrameModel(walkModels, frameCounterMov, gl);

	frameCounterMov++;
	if (frameCounterMov==walkModels.size())
		frameCounterMov=0;
}
```

Πρώτα ελέγχουμε την περίπτωση που ο τοξότης δεν κινείται (!heroIsMoving) και δεν επιτίθεται (!heroIsAttacking). Χρησιμοποιούμε τρείς μετρητές που μας δείχνουν πιο μοντέλο πρέπει να σχεδιαστεί σε κάθε επανάληψη της μεθόδου σχεδίασης, τους frameCounterAtt, frameCounterSt και frameCounterMov. Υπενθυμίζω ότι η παραπάνω διαδικασία βρίσκεται στην μέθοδο σχεδίασης display η οποία επαναλαμβάνεται συνεχώς και ότι οι μεταβλητές heroIsMoving και heroIsAttacking ενημερώνονται από ένα thread που εκτελείται παράλληλα με την μέθοδο αυτή. Έτσι λοιπόν όταν ο τοξότης απλά στέκεται, μηδενίζουμε τις μεταβλητές frameCounterAtt και frameCounterMov και αυξάνουμε την μεταβλητή frameCounterSt μετά από κάθε κλήση της μεθόδου drawFrameModel. Η συγκεκριμένη μέθοδος την οποία βλέπουμε παρακάτω σχεδιάζει το μοντέλο που βρίσκεται στην θέση (frame) του vector μοντέλων (models) τα οποία παίρνει σαν ορίσματα. Τα μοντέλα κάθε κίνησης τα έχουμε αποθηκεύσει σε ξεχωριστούς vector τους standModels, attackModels και walkModels. Όταν ο μετρητής (frameCounterSt) γίνει ίσος με το μήκος του vector (standModels) αυτό σημαίνει ότι σχεδιάστηκε και το τελευταίο στιγμιότυπο της κίνησης και έτσι μηδενίζουμε τον μετρητή για να επαναληφθεί η ίδια κίνηση.

```
public void drawFrameModel(Vector<OBJModel> models, int frame, GL gl)
{
	int index = 0;

	for (OBJModel model : models)
	{
		if (frame==index)
		{
			this.updateOBJModel(model);

			this.updateTranslation(new
			Tuple3(GameRenderer.playerPosition.x,
			GameRenderer.playerPosition.y - 10.0f,
			GameRenderer.playerPosition.z));
```

```
                this.updateBoundingBoxToFit(1.5f, 0.0f, 0.0f, 0.0f, 0.0f,
                4.0f, 0.0f);

                this.updateRotation(new Tuple3(0.0f, 1.0f, 0.0f),
                GameRenderer.angle);
                this.drawObject(gl);

                //this.drawBoundingBox(gl);
            }

            index++;


        }
}
```

Κάνουμε την ίδια διαδικασία και για τις δύο άλλες περιπτώσεις. Μηδενίζουμε τους μετρητές των άλλων κινήσεων και καλούμε την μέθοδο drawFrameModel για κάθε τιμή του μετρητή της κίνησης που θέλουμε, δίνοντας ορίσματα στην μέθοδο τον vector που περιέχει τα μοντέλα της κίνησης αυτής και την τιμή του μετρητή που υποδηλώνει την θέση του μοντέλου έχει σειρά να σχεδιαστεί. Ειδικά για την περίπτωση της επίθεσης στο σημείο όπου ο τοξότης έχει πάρει το βέλος και το έχει βάλει στο τόξο (attackModels.size() - 3) κάνουμε true την μεταβλητή heroAttacked για να ξέρουμε το σημείο κατά το οποίο το βέλος θα εμφανιστεί. Επίσης όταν ολοκληρωθεί η κίνηση της επίθεσης ενημερώνουμε την μεταβλητή heroIsAttacking, καθώς δεν θέλουμε η κίνηση της επίθεσης να επαναληφθεί παρά μόνο αν ξαναπατηθεί το αριστερό κουμπί του ποντικιού. Όσον αφορά τις κλήσεις των μεθόδων updateIndex και updateAnimationVector χρησιμοποιούνται για να σταλθεί το animation μέσω του δικτύου και θα το αναλύσουμε σε επόμενο κεφάλαιο.

Η μέθοδος drawFrameModel αυξάνει έναν μετρητή index για κάθε στοιχείο του vector που δίδεται σαν όρισμα. Όταν ο μετρητής index έχει ίδια τιμή με την τιμή της μεταβλητής που δίδεται επίσης σαν όρισμα, σχεδιάζεται το συγκεκριμένο μοντέλο το οποίο έχει προσπελαστεί. Άρα σαν συνολική διαδικασία για την δημιουργία ενός animation στο παιχνίδι θα πρέπει να κάνουμε το εξής.

- Να ορίσουμε τις κινήσεις που θα κάνει ένα αντικείμενο.
- Να δημιουργήσουμε τα μοντέλα obj για κάθε στιγμιότυπο των κινήσεων αυτών.
- Να αποθηκεύσουμε τα μοντέλα αυτά στους κατάλληλους vectors με την σειρά που πρέπει να σχεδιάζονται.
- Για κάθε κίνηση να σχεδιάζουμε το αντίστοιχο στιγμιότυπο (δηλαδή μοντέλο) του αντίστοιχου vector χρησιμοποιώντας την μέθοδο drawFramemodel.

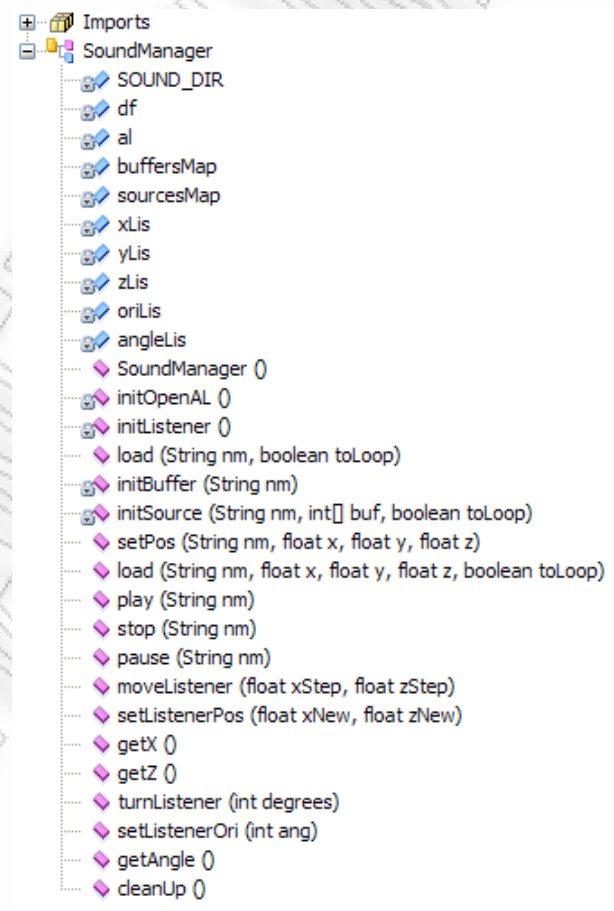
3.7. Ήχος

3.7.1. Εισαγωγή

Για τον ήχο χρησιμοποιείται η βιβλιοθήκη JOAL. Το παιχνίδι στο στάδιο που βρίσκεται έχει ήχους για τον βηματισμό του τοξότη και για την εκτόξευση ενός βέλους, καθώς και μουσική η οποία επαναλαμβάνεται συνεχώς. Τα αρχεία .wav βρίσκονται στον φάκελο Sounds. Εκεί μπορούμε να προσθέσουμε ότι άλλο αρχείο ήχου θέλουμε να χρησιμοποιήσουμε στο παιχνίδι ή να αλλάξουμε τα υπάρχοντα. Για την διαχείριση του ήχου έχει υλοποιηθεί η κλάση SoundManager.

3.7.2. Sound Manager

Παρακάτω βλέπουμε τις μεταβλητές και τις μεθόδους της κλάσης SoundManager. Στο string SOUND_DIR είναι αποθηκευμένη η διαδρομή στην οποία βρίσκονται τα αρχεία ήχου, δηλαδή το string '/Sounds'.



Εικόνα 3.21: Μεταβλητές και μέθοδοι της κλάσης SoundManager

Με την δημιουργία ενός αντικειμένου του τύπου της κλάσης SoundManager καλείται η μέθοδος-κατασκευαστής (την βλέπουμε παρακάτω) στην οποία δημιουργούνται τα δύο HashMaps που χρησιμοποιούμε για την αποθήκευση των παραμέτρων κάθε αρχείου ήχου και καλούνται οι μέθοδοι αρχικοποίησης initOpenAL και initListener.

```
public SoundManager()
{
      buffersMap = new HashMap<String, int[]>();
      sourcesMap = new HashMap<String, int[]>();

      initOpenAL();
      initListener();
}
```

Η μέθοδος initOpenAL εγκαθιστά την σύνδεση με το OpenAL (OpeanAL utility toolkit) μέσω της JOAL. Δημιουργεί ένα πλαίσιο OpenAL με την ονομασία al το οποίο θα χρησιμοποιήσουμε.

```
private void initOpenAL()
{
      try
      {
            ALut.alutInit();
            al = ALFactory.getAL();
            al.alGetError();
      }
      catch (ALException e)
      {
            e.printStackTrace();
            System.exit(1);
      }
}
```

Η μέθοδος initListener αρχικοποιεί τις παραμέτρους όσον αφορά την θέση και την εμβέλεια του ακροατή. Τον τοποθετεί στην θέση (0,0,0) να κοιτάει προς το σημείο (0,0,-1). Αυτό χρησιμεύει πιο πολύ για 3d ήχο ο οποίος στην παρούσα υλοποίηση του παιχνιδιού δεν έχει χρησιμοποιηθεί. Γι' αυτό το λόγο δεν υπάρχει εμβέλεια στους ήχους.

```
private void initListener()
{
      xLis = 0.0f; yLis  = 0.0f; zLis = 0.0f;
      al.alListener3f(AL.AL_POSITION, xLis, yLis, zLis);
      al.alListener3i(AL.AL_VELOCITY, 0, 0, 0);   // no velocity

      oriLis = new float[] {xLis, yLis, zLis-1.0f,  0.0f, 1.0f, 0.0f};
      al.alListenerfv(AL.AL_ORIENTATION, oriLis, 0);
}
```

Από την στιγμή που έχουν γίνει τα παραπάνω, είμαστε έτοιμοι να χρησιμοποιήσουμε ότι ήχο θέλουμε αρκεί να έχουμε το αντίστοιχο αρχείο .wav στον φάκελο /Sounds. Πρώτα πρέπει να χρησιμοποιήσουμε την μέθοδο load με ορίσματα το όνομα του αρχείου και την τιμή true αν θέλουμε να επαναλαμβάνεται ο ήχος για να φορτωθεί το αρχείο στο αντίστοιχο buffer.

```
public boolean load(String nm, boolean toLoop)
{
      if (sourcesMap.get(nm) != null)
      {
            return true;
      }

      int[] buffer = initBuffer(nm);
      if (buffer == null)
            return false;

      int[] source = initSource(nm, buffer, toLoop);
      if (source == null)
      {
            al.alDeleteBuffers(1, buffer, 0);
            return false;
      }

      buffersMap.put(nm, buffer);
      sourcesMap.put(nm, source);
      return true;
}
```

Έπειτα με την μέθοδο play ο ήχος αναπαράγεται, με την μέθοδο stop ο ήχος σταματά ενώ με την μέθοδο pause ο ήχος διακόπτεται. Τις βλέπουμε τις τρείς μεθόδους παρακάτω.

```
public boolean play(String nm)
{
      int[] source = (int[]) sourcesMap.get(nm);
      if (source == null)
      {
          return false;
      }

      al.alSourcePlay(source[0]);
      return true;
}

public boolean stop(String nm)
{
      int[] source = (int[]) sourcesMap.get(nm);
      if (source == null)
      {
```

```
            return false;


      }

      al.alSourceStop(source[0]);
      return true;
}

public boolean pause(String nm)
{
      int[] source = (int[]) sourcesMap.get(nm);
      if (source == null)
      {
            return false;
      }

      al.alSourcePause(source[0]);
      return true;
}
```

Παρακάτω βλέπουμε τα αρχεία των ήχων που χρησιμοποιούνται στο παιχνίδι.

Για την αναπαραγωγή της μουσικής όταν εμφανίζεται το αρχικό παράθυρο του παιχνιδιού φορτώνουμε πρώτα το αρχείο intro.wav και αν η φόρτωση γίνει με επιτυχία, το αναπαράγουμε ως εξής:

```
if (sound.load("intro", false))
      sound.play("intro");
```

Το αντικείμενο sound είναι ένα αντικείμενο τύπου SoundManager. Όταν το κουμπί Play πατηθεί η μουσική πρέπει να σταματήσει οπότε καλούμε την μέθοδο stop.

```
sound.stop("intro");
```

Στην μέθοδο αρχικοποίησης του παιχνιδιού init, φορτώνουμε τους ήχους για τον βηματισμό και την επίθεση του τοξότη και αναπαράγουμε την μουσική βάζοντας true στην αντίστοιχη μεταβλητή εφ' όσον θέλουμε να επαναλαμβάνεται.

```
sound.load("grass", false);
sound.load("arrow", false);
if (sound.load("music", true))
      sound.play("music");
```

Μόλις ο τοξότης κινείται, αναπαράγεται ο αντίστοιχος ήχος και μόλις δεν κινείται σταματά. Προσθέτουμε και μία χρονοκαθυστέρηση για να προλαβαίνει ο ήχος να παιχτεί πριν επαναληφθεί.

```
if (scene.heroIsMoving && (soundTimeStamp > (lastSoundTimeStamp + 500)))
{


      sound.play("grass");
      lastSoundTimeStamp = soundTimeStamp;
}

if (!scene.heroIsMoving)
      sound.stop("grass");
```

Για τον ήχο εκτόξευσης του βέλους, απλά αναπαράγουμε τον αντίστοιχο ήχο στο πάτημα του αριστερού κουμπιού του ποντικιού.

```
sound.play("arrow");
```

3.8. Δικτύωση

3.8.1 Εισαγωγή

Για την αποστολή πληροφοριών μέσω του δικτύου στο συγκεκριμένο παιχνίδι χρησιμοποιείται stream socket, δηλαδή γίνεται χρήση του πρωτοκόλλου TCP, ενός συνδεσμικού πρωτοκόλλου. Επίσης χρησιμοποιείται αρχιτεκτονική εξυπηρετητή-πελάτη χωρίς όμως ο εξυπηρετητής να έχει καμία παραπάνω λειτουργία από τον πελάτη. Είναι σχεδιασμένο έτσι ώστε να παίζουν δύο παίκτες μεταξύ τους μέσω διαδικτύου ενώ και τα δύο τερματικά λειτουργούν ως εξυπηρετητές και πελάτες ταυτόχρονα καθώς έχουν ισότιμες ευθύνες και ανταλλάσουν μεταξύ τους τις ίδιες πληροφορίες. Η αποστολή-λήψη πληροφοριών γίνεται μέσω μιας ροής byte, η οποία μπορεί να εννοηθεί ως ένας σωλήνας που ενώνει τους δύο υπολογιστές και ότι εισέλθει από την μία μεριά, φτάνει στην άλλη. Παρακάτω περιγράφω την έννοια του socket, τις αρχιτεκτονικές server-client και peer-to-peer και αναλυτικά τον τρόπο με τον οποίο ανταλλάσσονται οι πληροφορίες που θέλουμε στο παιχνίδι μεταξύ των δύο υπολογιστών.

3.8.2. Αρχιτεκτονική εξυπηρετητή-πελάτη

Το μοντέλο εξυπηρετητή – πελάτη αποτελεί μια δομή κατανεμημένης εφαρμογής στην οποία οι εργασίες και ο φόρτος εργασίας χωρίζεται μεταξύ των παρόχων ενός πόρου ή μιας υπηρεσίας, δηλαδή των εξυπηρετητών, και των αιτούντων της υπηρεσίας αυτής, δηλαδή των πελατών. Μια μηχανή εξυπηρετητή είναι ένας πάροχος στον οποίο εκτελούνται μία ή περισσότερες εφαρμογές οι οποίες διαμοιράζουν κάποιους πόρους με τους πελάτες. Ο πελάτης δεν μοιράζει κάποιον πόρο του, αλλά ζητά το περιεχόμενο ή την υπηρεσία που προσφέρει ένας εξυπηρετητής. Οι πελάτες επομένως, ξεκινάνε την διαδικασία της επικοινωνίας με τον εξυπηρετητή ο οποίος αναμένει για τέτοιες αιτήσεις. Αντίθετα στην αρχιτεκτονική peer-to-peer, κάθε στιγμιότυπο ενός προγράμματος λειτουργεί και ως εξυπηρετητής και ως πελάτης ταυτόχρονα, και κάθε τερματικό έχει ισότιμες ευθύνες. Τα πλεονεκτήματα της αρχιτεκτονικής εξυπηρετητή – πελάτη μπορούν να θεωρηθούν τα εξής:

• Σε πολλές περιπτώσεις, μια αρχιτεκτονική εξυπηρετητή – πελάτη επιτρέπει στις ευθύνες και στους ρόλους ενός συστήματος να κατανεμηθούν σε διάφορους ανεξάρτητους υπολογιστές, οι οποίοι βλέπουν ο ένας τον άλλον μόνο μεταξύ ενός δικτύου. Αυτό προσφέρει το επιπλέον πλεονέκτημα σε αυτήν την αρχιτεκτονική: μεγαλύτερη ευκολία στην συντήρηση. Για παράδειγμα είναι πιθανόν ένας εξυπηρετητής να αντικατασταθεί, να διορθωθεί ή να αναβαθμιστεί ενώ παράλληλα οι πελάτες δεν επηρεάζονται και δεν χρειάζεται να γνωρίζουν αυτήν την αλλαγή.

• Όλα τα δεδομένα αποθηκεύονται στον εξυπηρετητή, ο οποίος κατά γενική παραδοχή προσφέρει καλύτερο έλεγχο ασφάλειας από τους περισσότερους πελάτες. Οι εξυπηρετητές μπορούν να ελέγξουν

την πρόσβαση και τους πόρους, για να εγγυηθούν πως μόνο αυτοί οι πελάτες με τα κατάλληλα δικαιώματα θα έχουν πρόσβαση στους πόρους και στις πληροφορίες του εξυπηρετητή.

• Εφ’ όσον η αποθήκευση δεδομένων είναι κεντροποιημένη, αναβαθμίσεις σε αυτά τα δεδομένα γίνονται πολύ πιο εύκολα σε σύγκριση με την αρχιτεκτονική peer-to-peer. Στην αρχιτεκτονική p2p τυχόν αναβαθμίσεις θα πρέπει να διαμοιραστούν και να εφαρμοστούν σε κάθε τερματικό του συστήματος, το οποίο μπορεί να απαιτήσει πολύ χρόνο όταν το πλήθος των τερματικών είναι πολύ μεγάλο.

• Πολλές ωριμασμένες τεχνολογίες εξυπηρετητή-πελάτη είναι ήδη διαθέσιμες και έχουν σχεδιαστεί να προσφέρουν ασφάλεια, φιλικότητα στην διαπροσωπεία με τον χρήστη και ευκολία στην χρήση.

• Λειτουργεί με πελάτες διαφορετικών δυνατοτήτων και τεχνολογιών.

Μειονεκτήματα της αρχιτεκτονικής εξυπηρετητή-πελάτη είναι τα εξής:

• Όσο ο αριθμός των ταυτόχρονων αιτήσεων από πελάτες αυξάνεται, ο εξυπηρετητής μπορεί να υπερφορτωθεί. Αντίθετα, σε μια p2p αρχιτεκτονική, το συνολικό εύρος ζώνης αυξάνεται, όσο αυξάνονται και οι κόμβοι του συστήματος, εφ’ όσον το εύρος ζώνης του συστήματος ισούται με το άθροισμα των ευρών ζώνης όλων των επιμερών κόμβων.

• Η αρχιτεκτονική εξυπηρετητή-πελάτη στερείται της ανθεκτικότητας που προσφέρει ένα καλό σύστημα p2p. Αν συμβεί κάποιο λάθος σε έναν κρίσιμο εξυπηρετητή ή για κάποιο λόγο ο εξυπηρετητής ‘πέσει’, οι αιτήσεις όλων των πελατών θα απορριφτούν και όλο το σύστημα θα αποτύχει. Σε p2p συστήματα οι πόροι είναι κατανεμημένοι σε όλους του κόμβους του συστήματος, με αποτέλεσμα αν κάποιος κόμβος αποχωρήσει από το σύστημα, η υπηρεσία δεν θα σταματήσει να προσφέρεται και οι πελάτες θα μπορούν να εκπληρώσουν την εργασία που ξεκίνησαν.

3.8.3. Υποδοχές (Sockets)

Στην δικτύωση υπολογιστών μία υποδοχή διαδικτύου (Internet socket) ή υποδοχή δικτύου (network socket) είναι ένα τελικό σημείο σε μια αμφίδρομη ενδο-διεργασιακή ροή επικοινωνίας η οποία συμβαίνει σε ένα δίκτυο υπολογιστών το οποίο βασίζεται σε διαδικτυακά πρωτόκολλα, όπως το Διαδίκτυο. Ο όρος υποδοχή (socket) χρησιμοποιείται και σαν όνομα στο API (application programming interface) του σωρού του πρωτοκόλλου TCP/IP και παρέχεται από το λειτουργικό σύστημα. Οι υποδοχές αποτελούν ένα μηχανισμό για να αποστέλλονται πακέτα δεδομένων στην κατάλληλη διεργασία μιας εφαρμογής ή σε ένα νήμα, με βάση έναν συνδυασμό IP διευθύνσεων και αριθμών θυρών.

Μια διεύθυνση υποδοχής (socket address) είναι ο συνδυασμός μιας διεύθυνσης IP (η τοποθεσία του υπολογιστή) και μιας θύρας (η οποία αντιστοιχεί σε μια διεργασία μιας εφαρμογής) σε μία

μοναδική οντότητα. Συγκεκριμένα μία υποδοχή χαρακτηρίζεται από έναν μοναδικό συνδυασμό των παρακάτω:

• Τοπική διεύθυνση υποδοχής: Τοπική διεύθυνση IP και αριθμός θύρας.

• Απομακρυσμένη διεύθυνση υποδοχής: Μόνο για εγκατεστημένες TCP υποδοχές. Αυτό χρειάζεται σε μια αρχιτεκτονική εξυπηρετητή-πελάτη καθώς ένας TCP εξυπηρετητής πρέπει να εξυπηρετεί πολλούς διαφορετικούς πελάτες ταυτόχρονα. Ο εξυπηρετητής δημιουργεί μια υποδοχή για κάθε πελάτη, οι οποίες μοιράζονται την ίδια τοπική διεύθυνση υποδοχής.

• Πρωτόκολλο: Ένα πρωτόκολλο μετάδοσης (π.χ. TCP, UDP), raw IP ή άλλο.

Στο λειτουργικό σύστημα και στην εφαρμογή η οποία δημιούργησε την υποδοχή, η υποδοχή αναφέρεται με έναν μοναδικό ακέραιο αριθμό που ονομάζεται αναγνωριστικό υποδοχής (socket identifier). Υπάρχουν διάφοροι τύποι υποδοχών όπως:

• Υποδοχές δεδομενογραφημάτων (datagram sockets), επίσης γνωστές και ως ασυνδεσμικές υποδοχές (connectionless sockets) οι οποίες κάνουν χρήση του πρωτοκόλλου UDP.

• Υποδοχές ροής (stream sockets), επίσης γνωστές και ως συνδεσμικές υποδοχές (connection-oriented sockets) οι οποίες κάνουν χρήση του πρωτοκόλλου TCP ή SCTP.

• Ωμές υποδοχές (raw sockets), που χρησιμοποιούνται από δρομολογητές ή γενικότερα από δικτυακό εξοπλισμό. Εδώ το στρώμα μετάδοσης αγνοείται και δεν διαχωρίζονται οι επικεφαλίδες των πακέτων, παρ' όλ' αυτά είναι διαθέσιμες στην εφαρμογή. Παραδείγματα εφαρμογών είναι το ICMP (Internet

Control Message Protocol, το IGMP (Internet Group Management Protocol) και το OSPF (Open Shortest Path First).

Οι υποδοχές συνήθως υλοποιούνται από μια βιβλιοθήκη API όπως οι υποδοχές Berkeley, που πρωτοπαρουσιάστηκαν το 1983. Η ανάπτυξη εφαρμογών που χρησιμοποιούν αυτό το API λέγεται προγραμματισμός υποδοχών (socket programming). Παρακάτω βλέπουμε παραδείγματα μεθόδων οι οποίες χρησιμοποιούνται από την βιβλιοθήκη API:

• socket(): Δημιουργεί μία νέα υποδοχή κάποιου τύπου, που αναγνωρίζεται από ένα μοναδικό αριθμό και κατανέμονται κάποιοι πόροι του συστήματος σε αυτήν.

• bind(): Χρησιμοποιείται από την πλευρά του server και αντιστοιχεί μια υποδοχή με μια δομή διεύθυνσης υποδοχής δηλαδή μια συγκεκριμένη τοπική θύρα και μια τοπική διεύθυνση.

• listen(): Χρησιμοποιείται από την πλευρά του server και αλλάζει την κατάσταση μιας TCP υποδοχής σε κατάσταση ακρόασης.

• connect(): Χρησιμοποιείται από την πλευρά του client και αντιστοιχεί μια ελεύθερη τοπική θύρα σε μία υποδοχή. Σε περίπτωση που έχουμε TCP υποδοχή γίνεται προσπάθεια να εγκατασταθεί μια TCP σύνδεση.

• accept(): Χρησιμοποιείται από την πλευρά του server. Δέχεται μια εισερχόμενη αίτηση εγκατάστασης μια σύνδεσης TCP και δημιουργεί μία νέα υποδοχή με το ζεύγος διευθύνσεων υποδοχής του server και του client.

• send() και recv(): Χρησιμοποιούνται για την αποστολή και λήψη δεδομένων από μια απομακρυσμένη υποδοχή.

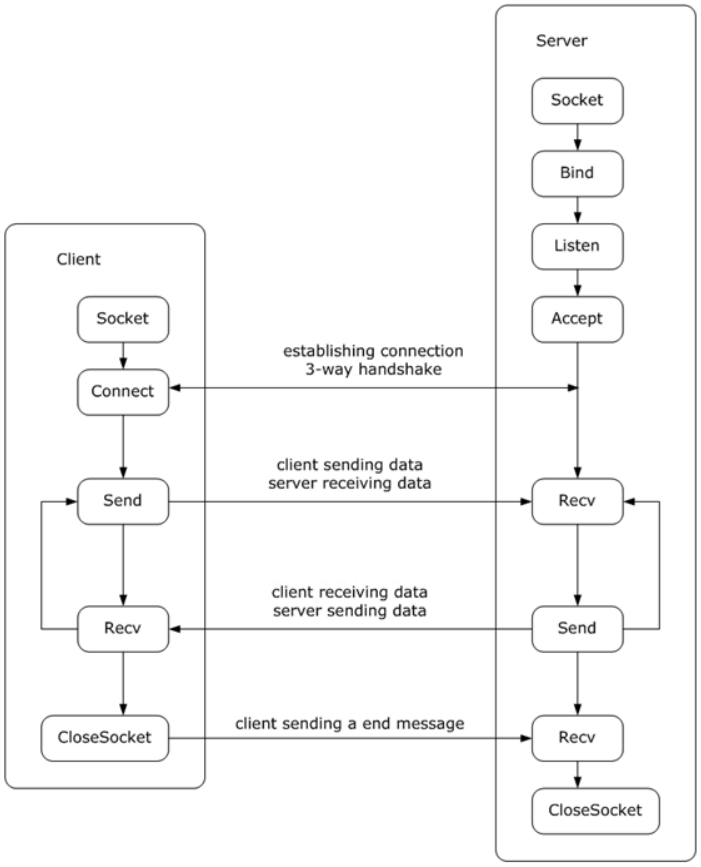
• close(): Χρησιμοποιείται για να απελευθερωθούν οι πόροι που έχουν κατανεμηθεί σε κάποια υποδοχή.

• gethostbyname() και gethostbyaddr(): Χρησιμοποιούνται για να επιλύσουν μια διεύθυνση με όνομα ή διεύθυνση IP αντίστοιχα.

• select(): Χρησιμοποιείται για να δημιουργηθεί μια λίστα με τις υποδοχές που είναι έτοιμες για εγγραφή δεδομένων, με αυτές που είναι έτοιμες για ανάγνωση και με αυτές που έχουν κάποιο σφάλμα.

• poll(): Χρησιμοποιείται για να ελεγχθεί η κατάσταση μιας υποδοχής.

Στην εικόνα 3.22 βλέπουμε το διάγραμμα ροής μιας υποδοχής TCP. Βλέπουμε ποιες μέθοδοι χρησιμοποιούνται από τον εξυπηρετητή και ποιες από τον πελάτη. Οι εξυπηρετητές, δηλαδή κάποιες υπολογιστικές διεργασίες που προσφέρουν κάποια υπηρεσία δημιουργούν κατά την εκκίνηση τους υποδοχές οι οποίες βρίσκονται σε κατάσταση ακοής ή αλλιώς κατάσταση αναμονής. Αυτές οι υποδοχές αναμένουν για πρωτοβουλίες από προγράμματα εκτελούμενα από την πλευρά του πελάτη. Ένας TCP εξυπηρετητής μπορεί να εξυπηρετήσει πολλούς πελάτες παράλληλα, δημιουργώντας μια θυγατρική διεργασία για κάθε πελάτη και δημιουργώντας μια υποδοχή μεταξύ κάθε τέτοιας διεργασίας και του πελάτη της.



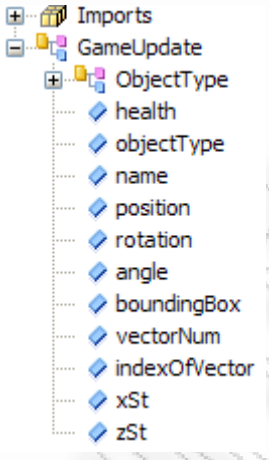
Εικόνα 3.22: Διάγραμμα ροής μιας υποδοχής TCP

3.8.4. Ανταλλαγή πληροφοριών στο παιχνίδι

Το συγκεκριμένο παιχνίδι ακολουθεί την λογική της αρχιτεκτονικής εξυπηρετητή-πελάτη, χωρίς όμως το τερματικό που λειτουργεί ως εξυπηρετητής να προσφέρει κάποια υπηρεσία. Και ο εξυπηρετητής και ο πελάτης ανταλλάσουν μεταξύ τους τις ίδιες πληροφορίες οι οποίες είναι η θέση, περιστροφή, η ενέργεια (ζωή) και το στιγμιότυπο (μοντέλο) του animation του τοξότη καθώς η θέση και περιστροφή των βελών που έχουν εκτοξευθεί. Χρησιμοποιείται υποδοχή ροής (stream socket) με χρήση του πρωτοκόλλου TCP για καλύτερη αξιοπιστία της σύνδεσης μεταξύ των δύο τερματικών καθώς το TCP είναι ένα συνδεσμικό πρωτόκολλο. Δηλαδή εγκαθίσταται μια σταθερή σύνδεση μεταξύ των δύο τερματικών μέσω της οποίας ανταλλάσσονται οι πληροφορίες με χρήση ροής byte.

Παρ’ όλα αυτά η αρχιτεκτονική ακολουθεί το μοντέλο εξυπηρετητή-πελάτη και το ένα από τα δύο τερματικά που ξεκινά την διαδικασία θα πρέπει να λειτουργήσει ως εξυπηρετητής και έπειτα το άλλο τερματικό να συνδεθεί σε αυτόν λειτουργώντας ως πελάτης. Επιπλέον το παιχνίδι είναι σχεδιασμένο για την αλληλεπίδραση δύο παικτών μόνο και δεν μπορούν παραπάνω από ένας πελάτης να συνθεδούν στον εξυπηρετητή. Με μια αναβάθμιση του συγκεκριμένου προγράμματος θα μπορούσαν να συνθεδούν παραπάνω του ενός παίκτες και να παίζουν όλοι μαζί, αλλά για κάτι τέτοιο θα ήταν καλύτερο να χρησιμοποιηθεί ξεχωριστή διεργασία η οποία θα λειτουργεί ως εξυπηρετητής και θα είναι ανεξάρτητη από τις υπόλοιπες διαδικασίες (π.χ. rendering) και η οποία θα βρίσκεται σε μηχάνημα με υψηλή απόδοση αφιερωμένο στην φιλοξενία της συγκεκριμένης διεργασίας.

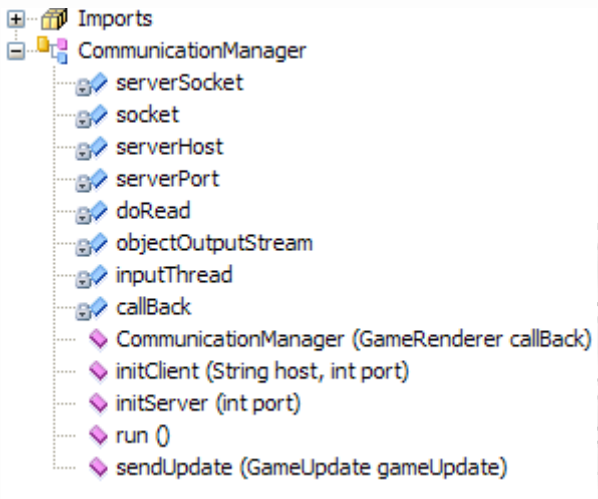
Όλες οι πληροφορίες οι οποίες ανταλλάσσονται μεταξύ των δύο τερματικών αποτελούν ένα αντικείμενο της κλάσης GameUpdate. Συνεχώς μεταδίδονται αντικείμενα τέτοιου τύπου μέσω του δικτύου και ενημερώνονται οι κατάλληλες μεταβλητές. Στην εικόνα 3.23 βλέπουμε τα πεδία της κλάσης αυτής.



Εικόνα 3.23: Μεταβλητές της κλάσης GameUpdate

Η μεταβλητή health αντιστοιχεί στην ενέργεια του τοξότη. Το objectType χρησιμοποιείται για να ξέρουμε αν το αντικείμενο αναφέρεται στον τοξότη ή σε ένα βέλος. Παίρνει τιμές τις Arrow και Player και με έναν έλεγχο αυτής της μεταβλητής ξέρουμε ότι οι ενημερώσεις πρέπει να γίνουν στον τοξότη ή στο βέλος. Το name είναι το όνομα του παίχτη το οποίο εισάγει στο αρχικό παράθυρο του παιχνιδιού. Το position είναι η θέση του τοξότη (ή του βέλους), το rotation η περιστροφή του, το angle η γωνία περιστροφής και το boundingBox το κουτί που περιβάλλει το συγκεκριμένο 3d αντικείμενο. Τα vectorNum και indexOfVector είναι για να ξέρουμε σε ποιο στιγμιότυπο του animation βρίσκεται ο τοξότης. Και στα δύο τερματικά υπάρχουν οι ίδιοι vectors οι οποίοι περιέχουν όλα τα διαδοχικά μοντέλα τα οποία δημιουργούν τα animation της κίνησης και της επίθεσης. Για να μην στέλνεται μέσω δικτύου ένα αντικείμενο του τύπου OBJModel, το οποίο είναι μεγάλο σε μέγεθος, στέλνονται μόνο ο αριθμός του vector που περιέχει το μοντέλο που θέλουμε και η θέση στην οποία βρίσκεται το μοντέλο αυτό. Τέλος τα xSt και zSt είναι τα xStep και zStep τα οποία στέλνονται σε περίπτωση που έχει εκτοξευτεί ένα βέλος και θέλουμε να ξέρουμε την διεύθυνση στην οποία θα κατευθυνθεί. Να σημειώσουμε ότι η κλάση GameUpdate υλοποιεί υποχρεωτικά την διεπαφή Serializable καθώς τα αντικείμενα της πρέπει να στέλνονται σειριακοποιημένα μέσω του δικτύου.

Για τις λειτουργίες λήψης και αποστολής αντικειμένων τύπου GameUpdate έχει υλοποιηθεί η κλάση CommunicationManager (εικόνα 3.24).



Εικόνα 3.24: Μεταβλητές και μέθοδοι της κλάσης CommunicationManager

Η κλάση αυτή περιέχει δύο μεθόδους για την αρχικοποίηση του server και του client (initServer και initClient αντίστοιχα). Μία μέθοδο εκκίνησης ενός νήματος το οποίο λαμβάνει αντικείμενα τύπου GameUpdate την οποία βλέπουμε παρακάτω:

```
public void run()
{
      ObjectInputStream inputStream;
      try
      {
            if(serverSocket != null)
            {
                  socket = serverSocket.accept();
                  System.out.println("Client connected from " +
                  socket.getRemoteSocketAddress());
                  objectOutputStream = new
                  ObjectOutputStream(socket.getOutputStream());
            }
            else
            {
                  socket = new Socket(serverHost, serverPort);
                  System.out.println("Connected to " +
                  socket.getRemoteSocketAddress());


                  objectOutputStream = new
                  ObjectOutputStream(socket.getOutputStream());
            }

            inputStream = new ObjectInputStream(socket.getInputStream());
            while (doRead)
```

```
            {
                  try
                  {
                        GameUpdate gameUpdate = (GameUpdate)
                        inputStream.readObject();

                        callBack.gameUpdated(gameUpdate);
                  }
                  catch (IOException e)
                  {
                        try
                        {
                              doRead = false;
                              objectOutputStream.close();
                              inputStream.close();
                              socket.close();
                              if (serverSocket != null)
                                    serverSocket.close();
                        }
                        catch (IOException e1)
                        {
                              e1.printStackTrace();
                        }
                  }
                  catch (ClassNotFoundException e)
                  {
                        e.printStackTrace();
                  }
            }
      }
      catch (IOException e)
      {
            e.printStackTrace();
      }
}
```

Αρχικά γίνεται ένας έλεγχος αν το συγκεκριμένο πρόγραμμα έχει εκτελεστεί ως server ή ως client. Αν συμβαίνει το πρώτο εκτελείται η εντολή accept(), που σημαίνει ότι η εκτέλεση της μεθόδου μπλοκάρει μέχρι να συνθεδεί κάποιος client. Αντίθετα αν πρόκειται για client, συνδεόμαστε στον server με την εντολή:

```
socket = new Socket(serverHost, serverPort);
```

Όπου serverHost είναι η διεύθυνση του server και serverPort η θύρα στην οποία συνδέεται ο client. Έπειτα δημιουργούνται δύο αντικείμενα τύπου ObjectOutputStream και ObjectInputStream και συνδέονται με το socket. Αυτά τα αντικείμενα θα χρησιμοποιηθούν για να διαβάζουμε αντικείμενα από την ροή (ObjectInputStream) και να γράφουμε αντικείμενα στην ροή (ObjectOutputStream). Στη συνέχεια εκτελείται ένας βρόχος συνεχώς μέχρι να δημιουργηθεί κάποιο σφάλμα τύπου IOExcpetion. Στον βρόχο αυτό διαβάζονται συνεχώς αντικείμενα τύπου GameUpdate με την εντολή

```
GameUpdate gameUpdate = (GameUpdate) inputStream.readObject();
```

Η εκτέλεση της μεθόδου μπλοκάρει σε αυτό το σημείο μέχρι να διαβαστεί ένα αντικείμενο. Μόλις διαβαστεί μετατρέπεται σε αντικείμενο τύπου GameUpdate και εκτελείται η μέθοδος gameUpdated η οποία βρίσκεται στην κλάση GameRenderer με όρισμα το αντικείμενο που διαβάστηκε και περιέχει τις ενημερώσεις που πρέπει να γίνουν στο παιχνίδι. Τέλος η μέθοδος sendUpdate χρησιμοποιείται για την αποστολή ενός αντικειμένου τύπου GameUpdate το οποίο πρέπει να δοθεί σαν όρισμα στην μέθοδο αυτή.

```
public void sendUpdate(GameUpdate gameUpdate)
{
	if (objectOutputStream != null && socket.isConnected())
	{
		try
		{
			objectOutputStream.writeObject(gameUpdate);
		}
		catch (IOException e)
		{
			// Do nothing for now
		}
	}
}
```

Η μέθοδος gameUpdated είναι υπεύθυνη για να εφαρμόσει τις ενημερώσεις οι οποίες λαμβάνονται από το δίκτυο. Καλείται κάθε φορά που λαμβάνεται ένα αντικείμενο τύπου GameUpdate. Την βλέπουμε παρακάτω:

```
public void gameUpdated(GameUpdate gameUpdate)
{
	synchronized (this)
	{
		if (gameUpdate.objectType == GameUpdate.ObjectType.Player)
		{
			if (scene.player2!=null)
			{
				scene.player2.updateTranslation(gameUpdate.position);
				scene.player2.updateRotation(gameUpdate.rotation,
				gameUpdate.angle);
```

```
scene.player2.updateBoundingBox
(gameUpdate.boundingBox);

scene.player2.updateAnimationVector
(gameUpdate.vectorNum);
scene.player2.updateIndex(gameUpdate.indexOfVector);
scene.player2.updateHealth(gameUpdate.health);

if (scene.player2.health==80)
{
      life2Offset = (screenSize.width/5)/5;
      life2.setBounds(2*screenSize.width/4,
      screenSize.height-25, screenSize.width/5-
      life2Offset, 20);

      life4.setBounds(2*screenSize.width/4+(screenSize.
      width/5-life2Offset), screenSize.height-25,
      life2Offset, 20);
}
else if (scene.player2.health==60)
{
      life2Offset = 2*(screenSize.width/5)/5;
      life2.setBounds(2*screenSize.width/4,
      screenSize.height-25, screenSize.width/5-
      life2Offset, 20);

      life4.setBounds(2*screenSize.width/4+(screenSize.
      width/5-life2Offset), screenSize.height-25,
      life2Offset, 20);
}
else if (scene.player2.health==40)
{
      life2Offset = 3*(screenSize.width/5)/5;
      life2.setBounds(2*screenSize.width/4,
      screenSize.height-25, screenSize.width/5-
      life2Offset, 20);

      life4.setBounds(2*screenSize.width/4+(screenSize.
      width/5-life2Offset), screenSize.height-25,
      life2Offset, 20);
}
else if (scene.player2.health==20)
{
      life2Offset = 4*(screenSize.width/5)/5;
      life2.setBounds(2*screenSize.width/4,
      screenSize.height-25, screenSize.width/5-
      life2Offset, 20);
```

```
                        life4.setBounds(2*screenSize.width/4+(screenSize.
                        width/5-life2Offset), screenSize.height-25,
                        life2Offset, 20);
                  }
                  else if (scene.player2.health==0)

                  {
                        life2Offset = 5*(screenSize.width/5)/5;
                        life2.setBounds(2*screenSize.width/4,
                        screenSize.height-25, screenSize.width/5-
                        life2Offset, 20);

                        life4.setBounds(2*screenSize.width/4+(screenSize.
                        width/5-life2Offset), screenSize.height-25,
                        life2Offset, 20);
                  }
            }
            name2.setText(gameUpdate.name);

      }
      else if (gameUpdate.objectType == GameUpdate.ObjectType.Arrow)
      {
            scene.arrow2 = new Object3D(scene.arrowModel,
            gameUpdate.position, gameUpdate.rotation, gameUpdate.angle,
            0.3f, gameUpdate.xSt, gameUpdate.zSt);
            if (scene.arrowsPlayer2.size()<10)
                  scene.arrowsPlayer2.add(scene.arrow2);
            else if(scene.arrowsPlayer2.size()==10)
            {
                  scene.arrowsPlayer2.set(index2, scene.arrow2);
                  index2++;
                  if (index2==10)
                        index2 = 0;
            }

      }
   }
}
```

Γίνεται ένας έλεγχος αν το αντικείμενο που στάλθηκε περιέχει τις ενημερώσεις που πρέπει να γίνουν σε κάποιο βέλος ή στον αντίπαλο τοξότη, δηλαδή αν το αντικείμενο έχει τιμή Arrow ή Player στην μεταβλητή ObjectType. Αν έχει τιμή Player, ενημερώνεται η θέση του αντίπαλου τοξότη (player2.updateTranslation), η γωνία περιστροφής του (player2.updateRotation), η θέση του κουτιού που περιβάλλει το μοντέλο για τον έλεγχο συγκρούσεων (player2.updateBoundingBox), το στιγμιότυπο του animation, δηλαδή το μοντέλο που πρέπει να σχεδιαστεί την συγκεκριμένη στιγμή για να έχουμε μια ροή κίνησης (player2.updateAnimationVector, player2.updateIndex) και τέλος η ενέργεια του αντίπαλου τοξότη (player2.updateHealth). Όπως προανέφερα το animation πετυχαίνεται με την διαδοχική σχεδίαση κάποιων μοντέλων τα οποία αποτελούν τα στιγμιότυπα της κίνησης. Όλα αυτά τα

μοντέλα είναι αποθηκευμένα σε ένα vector με την σειρά. Ο ένας παίκτης θα πρέπει να βλέπει την κίνηση του άλλου παίκτη, αλλά το να στείλουμε ολόκληρο το αντικείμενο τύπου OBJModel για να το λάβει και να το σχεδιάσει για το μοντέλο του αντίπαλου τοξότη θα ήταν τουλάχιστον ασύμφορο. Επομένως, αφού όλα τα μοντέλα του animation βρίσκονται σε ένα vector και στον server και στον client αρκεί να ξέρουμε ποιο vector περιέχει τα μοντέλα που θέλουμε (animationVector), εφ' όσον για κάθε animation έχουμε και ξεχωριστό vector και σε ποια θέση βρίσκεται το μοντέλο που θέλουμε να σχεδιάσουμε (index). Αυτές οι δύο μεταβλητές στέλνονται μαζί με τις άλλες πληροφορίες σε ένα αντικείμενο τύπου GameUpdate, λαμβάνονται από τα δύο προγράμματα και ενημερώνεται βάση αυτών το μοντέλο του τοξότη που χρησιμοποιείται εκείνη την στιγμή.

```
if (player2.animationVector==1)
      player2.updateOBJModel(standModels.elementAt(player2.index));

else if (player2.animationVector==2)
      player2.updateOBJModel(attackModels.elementAt(player2.index));
else if (player2.animationVector==3)
      player2.updateOBJModel(walkModels.elementAt(player2.index));
```

Η αναπαράσταση της ενέργειας κάθε τοξότη γίνεται από μία μπάρα στην βάση της οθόνης, η οποία περιέχει δύο χρώματα. Πράσινο για την υπολειπόμενη ενέργεια και κόκκινο για την χαμένη ενέργεια. Η ενημέρωση της μπάρας γίνεται με τον εξής τρόπο: Η μπάρα αποτελείται από δύο Jlabel στα οποία έχει προστεθεί μια εικόνα με πράσινο χρώμα στο ένα και μια εικόνα με κόκκινο χρώμα στο άλλο. Έτσι αυξομειώνοντας τα όρια του κάθε Jlabel πετυχαίνουμε την αναλογία πράσινου και κόκκινου που θέλουμε να φαίνεται στην μπάρα. Έτσι, λοιπόν, στην μέθοδο gameUpdated μετά την ενημέρωση της ενέργειας του αντίπαλου παίκτη κάνουμε έναν έλεγχο για την τιμή της ενέργειας του και αυξομειώνουμε κατάλληλα τα δύο Jlabel που αποτελούν την μπάρα ζωής του. Η ενέργεια ενός τοξότη μειώνεται κατά 20 μονάδες οπότε οι πιθανές τιμές της είναι 100, 80, 60, 40, 30 και 0.

Αν το αντικείμενο είναι τύπου Arrow, οι ενημερώσεις που περιέχονται αφορούν ένα βέλος του αντιπάλου οπότε πράττουμε ανάλογα. Δημιουργούμε ένα βέλος δίνοντας του σαν τιμές στις μεταβλητές του, τις τιμές που λάβαμε από το δίκτυο και κάνουμε έναν έλεγχο για να σχεδιάζονται μέχρι δέκα αντίπαλα βέλη στον χώρο. Από την στιγμή που δημιουργήσαμε το βέλος και το αποθηκεύσαμε στον κατάλληλο vector, το βέλος θα σχεδιαστεί και θα κινηθεί με συγκεκριμένη ταχύτητα και προς συγκεκριμένη κατεύθυνση (αυτήν που λάβαμε από τα xStep και zStep).

Το σημείο στο οποίο στέλνεται ένα αντικείμενο τύπου GameUpdate με όλες τις πληροφορίες για τον τοξότη είναι στο νήμα ελέγχου. Κάθε φορά που ελέγχονται τα πλήκτρα και το ποντίκι για τυχόν συμβάντα έπειτα στέλνεται και ένα αντικείμενο τύπου GameUpdate.

```
GameUpdate gameUpdate = new GameUpdate();
gameUpdate.objectType = GameUpdate.ObjectType.Player;
if (scene.player!=null)
{
      gameUpdate.position = scene.player.getTranslation();
      gameUpdate.rotation = scene.player.getRotation();
      gameUpdate.angle = scene.player.getAngle();
```

```
        gameUpdate.boundingBox = scene.player.getBoundingBox();
        gameUpdate.vectorNum = scene.player.getAnimationVector();

        gameUpdate.indexOfVector = scene.player.getIndex();
        gameUpdate.health = scene.player.getHealth();
}
gameUpdate.name = name1.getText();

communicationManager.sendUpdate(gameUpdate);
```

Δίνονται οι κατάλληλες τιμές στις μεταβλητές του αντικειμένου αυτού και έπειτα στέλνεται μέσω του socket στο άλλο τερματικό. Επίσης κάθε φορά που πατιέται το αριστερό κουμπί του ποντικιού, στέλνεται ένα αντικείμενο GameUpdate με τιμή Arrow στην μεταβλητή ObjectType.

```
GameUpdate gameUpdate = new GameUpdate();
gameUpdate.objectType = GameUpdate.ObjectType.Arrow;
gameUpdate.position = scene.arrow.getTranslation();
gameUpdate.rotation = scene.arrow.getRotation();
gameUpdate.angle = scene.arrow.getAngle();
gameUpdate.boundingBox = scene.arrow.getBoundingBox();
gameUpdate.xSt = scene.arrow.getXStep();
gameUpdate.zSt = scene.arrow.getZStep();

communicationManager.sendUpdate(gameUpdate);
```

Έτσι το βέλος θα σχεδιαστεί και στο άλλο τερματικό, αφού στέλνονται όλες οι πληροφορίες που χρειάζονται για την σχεδίαση του και παράλληλα σχεδιάζεται και στο ίδιο τερματικό. Η διαφορά από το αντικείμενο τύπου Player είναι ότι στέλνονται και οι τιμές xStep και zStep, οι οποίες από την στιγμή που εμφανιστεί το βέλος παραμένουν σταθερές και μπορούμε να γνωρίζουμε προς τα πού θα κινηθεί.

4. Επέκταση - βελτίωση παιχνιδιού

Υπάρχουν αρκετά περιθώρια βελτίωσης του συγκεκριμένου παιχνιδιού. Αναφέρω μερικές προτάσεις για μελλοντική επέκταση του παιχνιδιού:

• Εισαγωγή παικτών ελεγχόμενων από τεχνητή νοημοσύνη. Με αυτόν τον τρόπο ο χρήστης θα μπορεί να παίζει μόνος του αντίπαλος με τον υπολογιστή, αλλά θα πρέπει να δημιουργηθεί ένας αλγόριθμος τεχνητής νοημοσύνης ο οποίος θα ελέγχει τις κινήσεις και τις επιθέσεις του αντίπαλου τοξότη. Ένας τέτοιος απλός αλγόριθμος θα ήταν ο τοξότης να κινείται προς τυχαίες κατευθύνσεις μέχρι να καταλάβει κίνηση εντός κάποιων ορίων μπροστά του, οπότε να περιστρέφεται προς το μοντέλο που κινείται και να επιτίθεται.

• Προχωρημένα οπτικά εφέ π.χ. σκίαση, έλεγχος φωτισμού. Το παρών στάδιο του παιχνιδιού δεν υλοποιεί σκίαση των μοντέλων. Η εισαγωγή σκίασης και άλλων οπτικών εφέ θα αύξανε τον ρεαλισμό του παιχνιδιού.

• Ρεαλιστική ανίχνευση συγκρούσεων. Η ανίχνευση συγκρούσεων προκαλεί απλά την εκτέλεση κάποιων λειτουργιών, χωρίς να γίνεται ρεαλιστική και αντιληπτή με οπτικά εφέ στον παίκτη. Θα μπορούσε επίσης να βρεθεί μία μέθοδος έτσι ώστε το κουτί που περιβάλλει τα μοντέλα για τον έλεγχο συγκρούσεων να ταιριάζει απόλυτα στα όρια του μοντέλου και να μην περιέχει σημεία τα οποία δεν ανήκουν στον χώρο του μοντέλου. Έτσι η ανίχνευση συγκρούσεων θα ήταν πολύ πιο ακριβής.

• 3D ήχος. Οι λειτουργίες ήχου του παιχνιδιού θα μπορούσαν να επεκταθούν, ώστε να γίνει τρισδιάστατος. Π.χ. όταν περπατάει ο τοξότης, ο ήχος των βημάτων να έχει κάποια εμβέλεια και να ακούγεται ο ήχος των βημάτων στον αντίπαλο παίκτη όταν βρίσκεται εντός της εμβέλειας αυτής.

• Εισαγωγή μη επίπεδου εδάφους. Η δημιουργία διακυμάνσεων στο έδαφος απαιτεί και την ανανέωση των λειτουργιών ελέγχου κίνησης (τοξότη και βέλους). Θα πρέπει να υπολογίζεται και ο άξονας y στην κίνηση των μοντέλων.

5. Εγκατάσταση - χρήση παιχνιδιού

5.1. Εγκατάσταση παιχνιδιού

Η εγκατάσταση του παιχνιδιού απαιτεί μόνο την αντιγραφή των αρχείων του παιχνιδιού σε κάποιο φάκελο του τοπικού δίσκου του υπολογιστή. Υλοποιήθηκε και δοκιμάστηκε σε λειτουργικό σύστημα Windows XP (service pack 3) το οποίο και προτείνεται για την ομαλή εκτέλεση του. Απαιτείται να έχει εγκατασταθεί προηγουμένως μια έκδοση Java. Προτείνεται η έκδοση 6 (version 6 update23), η οποία είναι και η νεώτερη μέχρι στιγμής. Η εγκατάσταση της νεώτερης έκδοσης Java μπορεί να γίνει από την παρακάτω ιστοσελίδα:

http://www.java.com/en/download/index.jsp

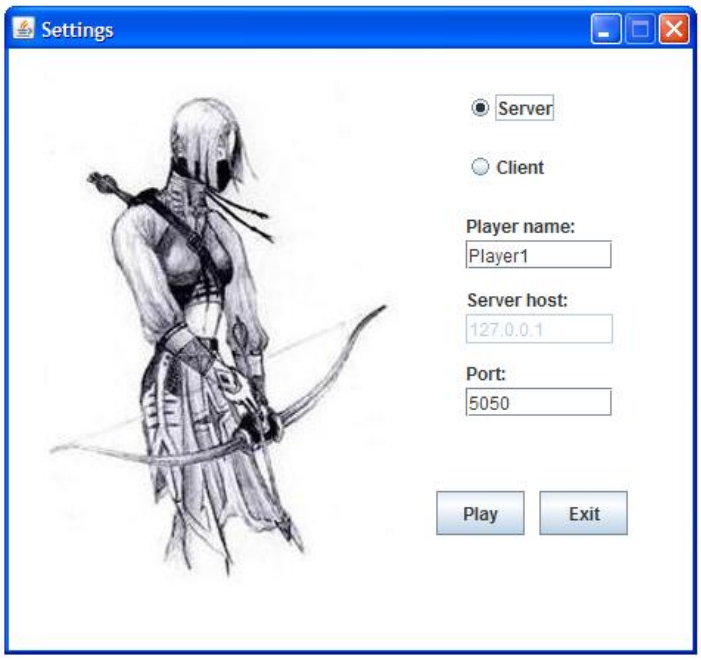
Από την στιγμή που θα εγκατασταθεί, θα πρέπει να οριστεί η διαδρομή του εκτελέσιμου αρχείου java.exe στις μεταβλητές του συστήματος για να το εκτελούμε χωρίς να βρισκόμαστε στον φάκελο που βρίσκεται.

5.2. Εκτέλεση παιχνιδιού

Η εκτέλεση του παιχνιδιού γίνεται με την εκτέλεση του αρχείου *run.bat* που βρίσκεται στον φάκελο του παιχνιδιού. Το αρχείο αυτό περιέχει την εντολή:

java.exe -Djava.ext.dirs=.\lib GameRenderer

Εκτέλεση του προγράμματος με την εντολή *java.exe GameRenderer* χωρίς να οριστεί η τοποθεσία των βιβλιοθηκών του παιχνιδιού, θα οδηγήσει σε εσφαλμένη εκτέλεση του παιχνιδιού εκτός και αν έχουν εγκατασταθεί προηγουμένως οι συγκεκριμένες βιβλιοθήκες στο σύστημα. Από την στιγμή που θα εκτελεστεί το αρχείο *run.bat* επιτυχώς θα εμφανιστεί το παράθυρο της εικόνας 5.1. Ο πρώτος παίκτης που θα συνδεθεί στο παιχνίδι θα πρέπει να επιλέξει την επιλογή *Server*. Στο πεδίο Player name μπορεί να εισάγει το όνομα που επιθυμεί και το οποίο θα εμφανίζεται στο παιχνίδι στην μπάρα ενέργειας του τοξότη. Το πεδίο Server host είναι απενεργοποιημένο καθώς το συγκεκριμένο τερματικό θα λειτουργήσει ως server και δεν χρειάζεται να εισαχθεί διεύθυνση IP. Στο πεδίο Port θα πρέπει να εισαχθεί ο αριθμός μιας ελεύθερης θύρας στην οποία θα ακούει ο server για συνδέσεις πελατών. Με το πάτημα του κουμπιού Play θα εκτελεστεί το παιχνίδι (εικόνα 5.3) ενώ με το πάτημα του κουμπιού Exit το παράθυρο θα εξαφανιστεί και η εκτέλεση του παιχνιδιού θα ακυρωθεί. Μέχρι να φορτωθεί το παιχνίδι βλέπουμε το αντίστοιχο μήνυμα (εικόνα 5.2).



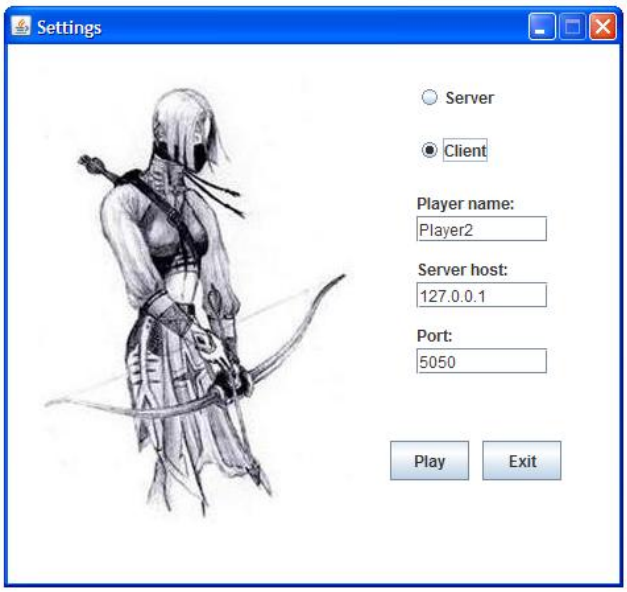
Εικόνα 5.1: Παράθυρο εισαγωγής παραμέτρων του παιχνιδιού (Server)

Εικόνα 5.2: Μήνυμα αναμονής για φόρτωμα του παιχνιδιού

Εικόνα 5.3: Εκτέλεση παιχνιδιού μετά το πάτημα του Play (Server)

Βλέπουμε ότι στην κάτω αριστερή μπάρα ζωής (η οποία είναι η ενέργεια του τοξότη που ελέγχει ο παίκτης) αναγράφεται το όνομα που εισάχθηκε στο πεδίο Player Name του παραθύρου παραμέτρων

(Player1 στην συγκεκριμένη περίπτωση). Στην άλλη μπάρα δεν αναγράφεται κάποιο όνομα καθώς ο άλλος παίκτης δεν έχει συνδεθεί ακόμα. Ο άλλος παίκτης θα πρέπει να τρέξει το παιχνίδι εκτελώντας το ίδιο αρχείο, δηλαδή το *run.bat*. Θα εμφανιστεί το παράθυρο εισαγωγής παραμέτρων. Θα πρέπει να επιλέξει *Client*, να βάλει όνομα στο πεδίο Player Name, στο πεδίο Server host να εισάγει την διεύθυνση IP του Server και στο πεδίο Port να εισάγει τον αριθμό της θύρας που ακούει ο Server (δηλαδή τον αριθμό που έβαλε ο παίκτης που λειτουργεί ως Server). Βλέπουμε τις παραμέτρους που πρέπει να εισάγει ο δεύτερος παίκτης στην εικόνα 5.4.



Εικόνα 5.4: Παράθυρο εισαγωγής παραμέτρων του παιχνιδιού (Client)

Εικόνα 5.5: Εκτέλεση παιχνιδιού μετά το πάτημα του Play (Client)

Στην εικόνα 5.5 βλέπουμε την επιτυχής σύνδεση του client στον server. Το όνομα του παίκτη αναγράφεται στην πρώτη μπάρα και το όνομα του αντίπαλου παίκτη στην δεύτερη (το ίδιο και από την πλευρά του server). Σε περίπτωση που ένα βέλος πετύχει τον αντίπαλο τοξότη γίνονται οι ενημερώσεις στις μπάρες ζωής (εικόνα 5.6).

Εικόνα 5.6: Ενημέρωση μπάρας ζωής

Σε περίπτωση που ένας παίκτης πετύχει τον αντίπαλο πέντε φορές, ο αντίπαλος χάνει και το αντίστοιχο μήνυμα εμφανίζεται (εικόνα 5.7).

Εικόνα 5.7: Μήνυμα νίκης

Στον παίκτη που χάνει εμφανίζεται μήνυμα ήττας (εικόνα 5.8). Με το πάτημα του κουμπιού OK γίνεται έξοδος από το παιχνίδι.

Εικόνα 5.8: Μήνυμα ήττας

5.3. Λειτουργίες παιχνιδιού

Η κίνηση του τοξότη γίνεται ως εξής:

- Κίνηση εμπρός: πάτημα κουμπιού W
- Κίνηση πίσω: πάτημα κουμπιού S
- Κίνηση δεξιά: πάτημα κουμπιού D
- Κίνηση αριστερά: πάτημα κουμπιού A
- Κίνηση διαγώνια εμπρός-δεξιά: πάτημα κουμπιών W και D
- Κίνηση διαγώνια εμπρός-αριστερά: πάτημα κουμπιών W και A
- Κίνηση διαγώνια πίσω-δεξιά: πάτημα κουμπιών S και D
- Κίνηση διαγώνια πίσω-αριστερά: πάτημα κουμπιών S και A

Η εκτόξευση ενός βέλους γίνεται με το πάτημα του αριστερού κουμπιού του ποντικιού. Το κουμπί μπορεί να είναι και πατημένο συνεχώς για την συνεχή εκτόξευση βέλων.

Η έξοδος από το παιχνίδι γίνεται με το πάτημα του πλήκτρου ESC.

6. Αξιολόγηση παιχνιδιού από χρήστες

Το παιχνίδι αξιολογήθηκε από χρήστες οι οποίοι δοκίμασαν το παιχνίδι και συμπλήρωσαν το παρακάτω ερωτηματολόγιο. Χρησιμοποιήθηκε μία κλίμακα από το 1 έως το 5 η οποία βασίζεται στα δύο βασικά χαρακτηριστικά του παιχνιδιού, τα γραφικά και τους χειρισμούς καθώς και σε μια συνολική εκτίμηση του παιχνιδιού:

1. Πολύ λίγο
2. Λίγο
3. Αρκετά
4. Πολύ
5. Πάρα πολύ

1. Θεωρείτε ότι τα τρισδιάστατα αντικείμενα του παιχνιδιού είναι ρεαλιστικά;

1 2 3 4 5

2. Θεωρείτε ότι η κίνηση του τοξότη γίνεται με φυσικό τρόπο;

1 2 3 4 5

3. Σας άρεσε ο τρόπος εμφάνισης της ενέργειας του τοξότη;

1 2 3 4 5

4. Θεωρείτε ότι το έδαφος και ο ουρανός ταιριάζουν με το σκηνικό;

1 2 3 4 5

5. Πόσο πλήρες θεωρείτε ότι είναι το σκηνικό του παιχνιδιού;

1 2 3 4 5

6. Βρίσκετε τους ήχους του παιχνιδιού διασκεδαστικούς;

1 2 3 4 5

7. Βρήκατε τους χειρισμούς βολικούς;

1 2 3 4 5

8. Πιστεύετε ότι οι τρόποι χειρισμού του τοξότη είναι εύκολοι στη χρήση;

1 2 3 4 5

9. Πόσο διασκεδαστικό βρήκατε το παιχνίδι;

1 2 3 4 5

10. Θεωρείτε ότι το παιχνίδι είχε ένταση;

1 2 3 4 5

Το παιχνίδι αξιολογήθηκε από έξι χρήστες και τα αποτελέσματα ήταν τα εξής:

Ερώτηση	Βαθμολογία
1	4
2	4
3	4
4	3
5	3
6	4
7	5
8	5
9	4
10	3

Μέσος Όρος βαθμολογίας: 3.9

Βιβλιογραφία

[1] Addison-Wesley Publishing Company, “The Official Guide to Learning OpenGL”, 1997.

[2] Andrew Davison, “Pro Java 6 3D Game Development”, 2007.

[3] John Feil and Marc Scattergood, “Beginning Game Level Design”, 2005.

[4] Andrew Davison, “Killer Game Programming in Java”, O’Reilly Media 2005.

[5] http://en.wikipedia.org/wiki/, “History of Games”, “Video Game”.

[6] J. P. Gee, “What Would a State of the Art Instructional Video Game Look Like?”, 2005.

[7] http://gpwiki.org/index.php/Main_Page, “Game Programming”, “Game Design”.

[8] Dave Astle, Kevin Hawkins, “Beginning OpenGL Game Programming”, 2004.

[9] Andrew Mulholland and Teijo Hakala, “Programming Multiplayer Games”, Wordware Publishing, Inc. 2004.

[10] Christer Ericson, “Real-Time Collision Detection”, Elsevier Inc. 2005.

[11] http://chumbalum.swissquake.ch/ms3d/tutorials.html, “Tutorials on MilkShape 3D software”.

[12] http://www.royriggs.com/obj.html, “OBJ File format”.

[13] http://en.wikipedia.org/wiki/Wavefront_.obj_file, “Wavefront .obj file”.

[14] Cay S. Horstmann and Gary Cornell, “Core Java”, Prentice Hall 2007.