ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Διαδραστικά εικονικά περιβάλλοντα πολλαπλών χρηστών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Σταύρος Αποστόλου-Καραμπέλης

Επιβλέπων : Στέφανος Κόλλιας
Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2007

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Διαδραστικά εικονικά περιβάλλοντα πολλαπλών χρηστών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Σταύρος Αποστόλου-Καραμπέλης

Επιβλέπων : Στέφανος Κόλλιας
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή στις 5 Οκτωβρίου 2007.

.....	.....	.....
Στέφανος Κόλλιας	Ανδρέας Σταφυλοπάτης	Παναγιώτης Τσανάκας
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2007

.....

Σταύρος Αποστόλου-Καραμπέλης

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

Γενικά

Στη διπλωματική αυτή εργασία στόχος ήταν η ανάπτυξη ενός (περιορισμένου) εικονικού κόσμου, στον οποίο οι χρήστες μπορούν να συνδεθούν μέσω δικτύου ώστε να μπορούν να αλληλεπιδράσουν με τα διάφορα στοιχεία του κόσμου. Η εφαρμογή βασίζεται στη μηχανή TGE (ή απλώς Torque) και είναι γραμμένη εξ' ολοκλήρου στην scripting γλώσσα αυτής (TorqueScript), με εξαίρεση το κομμάτι που αφορά τον πελάτη για το τμήμα απομακρυσμένου ελέγχου μέσω sockets, όπου χρησιμοποιήθηκε C/C++ μαζί με το Win32 API.

Ολόκληρη η εφαρμογή βρίσκεται στο CD που συνοδεύει το παρόν κείμενο, στη μορφή που είχε όταν κατατέθηκε η εργασία. Ο κώδικας περιλαμβάνει όλα τα σχόλια όπως ακριβώς προέκυψαν κατά τη συγγραφή του (σημείωση: επίσης περιλαμβάνει κάποια πράγματα που δεν ήταν σε τελική μορφή όταν παραδόθηκε η εργασία και τα οποία δεν αναφέρονται πουθενά παρακάτω).

Γενικά, η εφαρμογή μπορεί να αναλυθεί σε μερικά σχετικά ανεξάρτητα τμήματα, που όλα μαζί απαρτίζουν τη συνολική "εικόνα" που θα έχει ένας χρήστης όταν εισέρχεται στον εικονικό κόσμο και αλληλεπιδρά μαζί του. Τα κυριότερα τμήματα διακρίνονται αν παρατηρήσει κανείς τους τίτλους των κεφαλαίων στα περιεχόμενα.

Αυτό που ακολουθεί είναι μία γενική και συνοπτική παρουσίαση των πιο βασικών επιμέρους τμημάτων της εφαρμογής. Γίνεται προσπάθεια ώστε αυτά που ακολουθούν να αποτελούν τόσο μία γενικότερη επισκόπηση της λογικής σχεδίασης πίσω από την εφαρμογή, όσο και έναν οδηγό στα σχετικά αρχεία κώδικα που υλοποιούν το κάθε τμήμα.

Abstract

In this diploma thesis the goal was to develop a (limited) virtual world, with users being able to connect via a network and interact with the world's elements. The application is based on the TGE (or simply Torque) engine, and it's written in the engine's scripting language (namely TorqueScript), with the exception of the part dealing with remote control of the world through sockets, where C/C++ along with the Win32 API was used.

The whole application is included in the CD accompanying this document, as it was at the time this document was delivered for evaluation. The code includes all of the comments, exactly as they were written while the application was being developed (note: also includes some things that had not reached a final state at the time, and these are not mentioned in the remainder of this text).

Generally, the application can be separated in a few relatively independent parts, that together consist the "picture" a user will have when he enters the world and interacts with it. The most important parts can be discerned from the chapter titles in the table of contents.

What follows is a general and short presentation of the application's most important parts. Effort is made for what follows to be a general review of the design logic behind the application, as well as a guide of the code files implementing each part.

Λέξεις-Κλειδιά (KeyWords)

- ***TGE:*** **(**Torque Game Engine) η μηχανή που χρησιμοποιήθηκε
- ***Torque:*** (ομοίως)
- ***TorqueScript:*** η scripting γλώσσα της μηχανής
- ***GUI:*** (Graphical User Interface) διεπαφή χρήστη για την αλληλεπίδραση με μια εφαρμογή
- ***avatar:*** η αναπαράσταση ενός χρήστη σε ένα εικονικό περιβάλλον (απλουστευμένα, ο "χρήστης")
- ***AI:*** (Artificial Intelligence) σε εικονικά περιβάλλοντα αναφέρεται σε οποιοδήποτε αντικείμενο η συμπεριφορά του οποίου ελέγχεται από τον υπολογιστή (πιθανώς με μεθόδους που προσεγγίζουν το πεδίο της τεχνητής νοημοσύνης). Εναλλακτικά, μπορεί να αναφέρεται στη συμπεριφορά του αντικειμένου αντί στο ίδιο το αντικείμενο
- ***NPC:*** (Non-Player Character) σαν υποκατηγορία της έννοιας AI, συνήθως αναφέρεται σε ανθρωπόμορφα αντικείμενα, που έχουν κατά κανόνα δυνατότητα διαλόγου με τον χρήστη
- ***bot:*** (ro***bot***) συνήθως αναφέρεται ως υποκατηγορία της έννοιας NPC παραπάνω, περιορίζεται ως προς το γεγονός ότι δε "μιλάει" στον χρήστη (δεν του δίνει πληροφορίες, αποστολές, κλπ.), και συνήθως έχει σαν μόνο στόχο την εξόντωση του. Είναι δηλαδή πιο "χαζό" στη συμπεριφορά του από έναν NPC
- ***HUD:*** (Head-Up Display) αναφέρεται στο GUI που υπερτίθεται στην απεικόνιση του εικονικού κόσμου (χωρίς να την κρύβει), ώστε να παρέχει άμεσα πληροφορίες στον χρήστη χωρίς αυτός να πρέπει να κοιτάξει κάπου αλλού αποσπώντας την προσοχή του από τα τεκταινόμενα του κόσμου (έννοια δανεισμένη από τη σχεδίαση πολεμικών αεροπλάνων)
- ***inventory:*** το σύστημα καταγραφής, παρουσίασης και γενικά διαχείρισης των αντικειμένων που κουβαλάει ένας χρήστης
- ***quest:*** συνήθως αναφέρεται σε κάποια αποστολή που πρέπει να φέρει σε πέρας ο χρήστης

ΠΕΡΙΧΟΜΕΝΑ

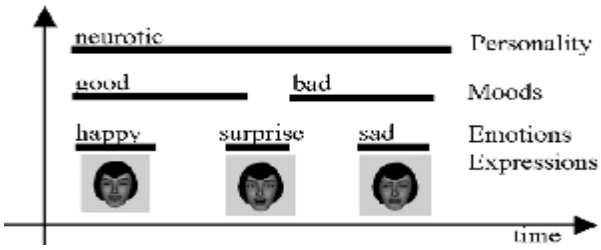
Κατάλογος Σχημάτων-Εικόνων

1. *Γενική θεώρηση της αλληλεπίδρασης με εικονικές οντότητες*

Γενικά για τη δημιουργία αληθοφανών εικονικών οντοτήτων (πρακτόρων).
Μπορούμε να διακρίνουμε τρία επίπεδα μοντελοποίησης της συμπεριφοράς τέτοιων *πρακτόρων (agents)*:

- *προσωπικότητα (personality)*
  αναφέρεται σε γενικά σταθερά γνωρίσματα που χαρακτηρίζουν τον τρόπο σκέψης και συμπεριφοράς ενός πράκτορα.
- *διάθεση (mood)*
  δρα σαν ένα ενδιάμεσο στάδιο μεταξύ προσωπικότητας και συναισθημάτων, καθώς τα μοντέλα προσωπικότητας είναι υψηλού επιπέδου και συνεπώς δύσκολο να ελέγξουν απ' ευθείας τα συναισθήματα που πρέπει να αναπαράγει ένας πράκτορας (π.χ. με εκφράσεις του προσώπου του) [KSHI02].
  Η διάθεση έχει μοντελοποιηθεί τόσο ως μονοδιάστατη [KSHI02], με διαβαθμίσεις μεταξύ 'καλής' και 'κακής', όσο και ως χωριζόμενη σε εσωτερική διάθεση και σε διάθεση προς τους άλλους πράκτορες (π.χ. μπορεί ένας πράκτορας να έχει γενικά καλή διάθεση, αλλά κακή διάθεση προς κάποιον άλλο συγκεκριμένο πράκτορα) [ROUS97].
- *συναίσθημα (emotion)*
  βλ. μοντέλα Ekman, OCC, παρακάτω.

Τα επίπεδα αυτά διαχωρίζονται κυρίως βάση της *χρονικής διάρκειάς (time duration)* τους [KSHI02], [MOFF97], [WILS99]: τα συναισθήματα είναι μικρής διάρκειας, η διάθεση αλλάζει πιο αργά με την πάροδο του χρόνου, ενώ η προσωπικότητα παραμένει σταθερή. Η χρονική συσχέτιση των τριών επιπέδων απεικονίζεται στο παρακάτω σχήμα, από [KSHI02]:



Σχ.1 Προσωπικότητα, διάθεση, συναίσθημα

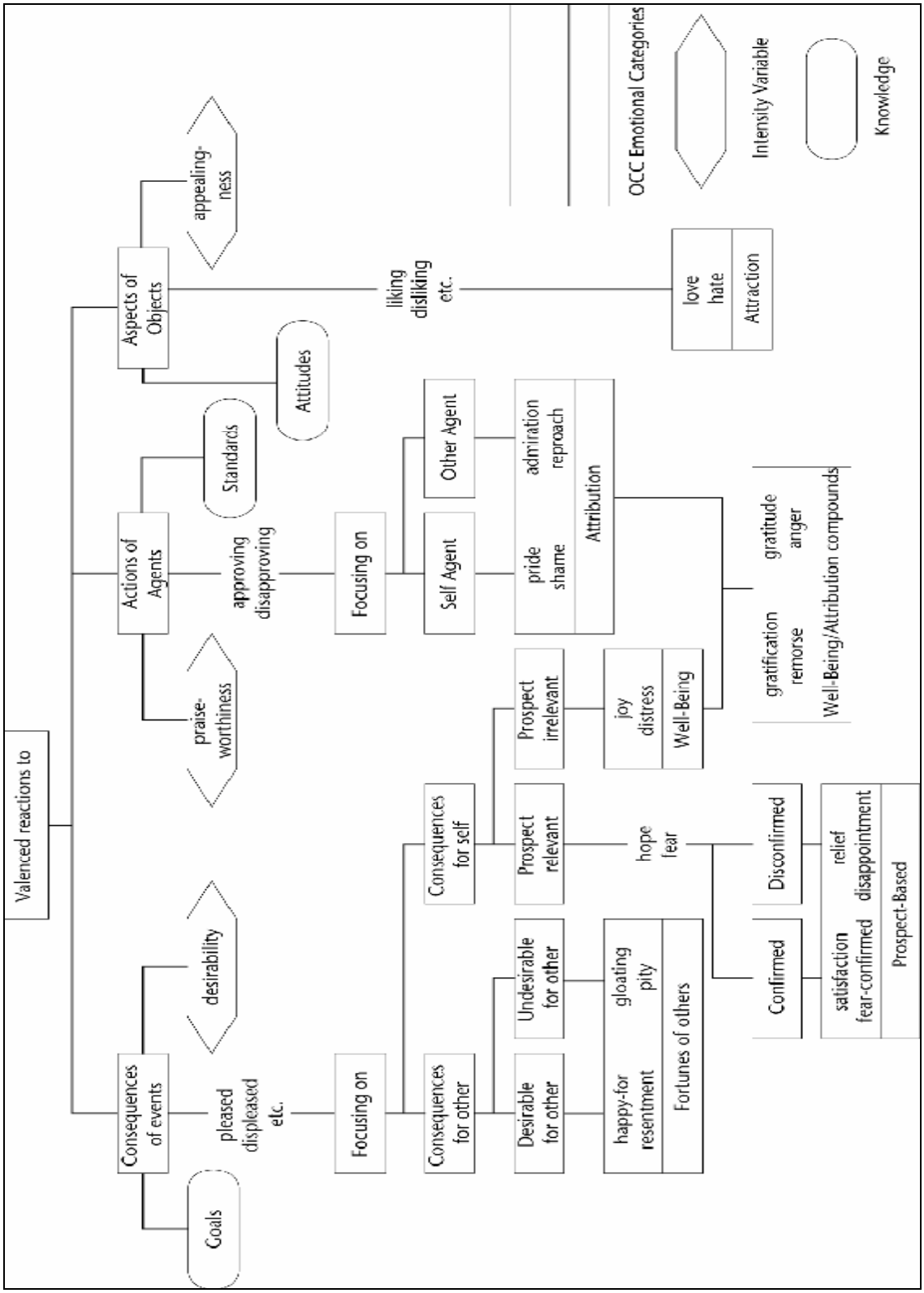
Επίσης μπορούμε να αποδώσουμε διαφορετική *προτεραιότητα (priority)* στην επίδραση που έχουν τα επίπεδα στη συμπεριφορά ενός πράκτορα [WILS99]: η προσωπικότητα έχει τη μικρότερη προτεραιότητα, και το συναίσθημα τη μεγαλύτερη. Ακόμα, μπορεί κανείς να παρατηρήσει διαφορά στην *εστίαση (focus)* μεταξύ συναισθημάτων και προσωπικότητας [MOFF97]: τα συναισθήματα είναι περισσότερο εστιασμένα σε συγκεκριμένα γεγονότα (events), ενέργειες (actions) και αντικείμενα (objects), ενώ η προσωπικότητα είναι περισσότερο γενική (λιγότερο εστιασμένη).

Μοντέλο Ekman
Σε αντίθεση με άλλους ανθρωπολόγους της εποχής, ο Ekman διαπίστωσε ότι οι εκφράσεις του προσώπου μέσω των οποίων αναπαριστώνται συναισθήματα δεν εξαρτώνται από πολιτισμικά χαρακτηριστικά, αλλά είναι παγκόσμιες στην ανθρώπινη

κουλτούρα και συνεπώς βιολογικές στην προέλευσή τους, όπως είχε θεωρήσει ο Δαρβίνος [WIKI07].
Στο μοντέλο του Ekman διακρίνονται έξη βασικές κατηγορίες συναισθημάτων, που είναι παγκόσμιες, και ενδεχομένως επαρκούν αν χρησιμοποιούμε μόνο εκφράσεις του προσώπου για να αποδώσουμε τα συναισθήματα του πράκτορα: χαρά (joy), λύπη (sadness), θυμός (anger), απέχθεια (disgust), φόβος (fear), έκπληξη (surprise) [EKMA72].
Ακόμα υπάρχουν νύξεις για άλλα συναισθήματα που θα μπορούσαν και αυτά να είναι παγκόσμια, όπως η περιφρόνηση (contempt) [WIKI07].

Μοντέλο OCC (Ortony, Clore, Collins, 1988).
Διαχωρίζει τα συναισθήματα σε κατηγορίες (emotional catogories). Η επιλογή της σωστής κατηγορίας συναισθήματος για κάποιον πράκτορα γίνεται με βάση *τα γεγονότα (events)* που συμβαίνουν στον κόσμο, *τις ενέργειες (actions)* άλλων πρακτόρων, και *τα αντικείμενα (objects)* που υπάρχουν. Η επίδραση που έχουν τα γεγονότα, οι ενέργειες και τα αντικείμενα στην επιλογή αυτή σταθμίζεται με βάση a priory *γνώσης (knowledge)* του πράκτορα για : 1) τους στόχους του (goals), 2) τα πρότυπα συμπεριφοράς που έχει για τις ενέργειες άλλων πρακτόρων (standards), και 3) το πώς αντιλαμβάνεται τα αντικείμενα (attitudes). Τέλος, η επίδραση κάθε γεγονότος, ενέργειας ή αντικειμένου σταθμίζεται και από μια μεταβλητή *ένταση (intensity)* ανάλογα με τη σημαντικότητα του εκάστοτε γεγονότος, ενέργειας ή αντικειμένου.

Η διαγραμματική αναπαράσταση του μοντέλου OCC, από [BART02], είναι:



Σχ.2 Μοντέλο OCC

Η επεξεργασία συναισθήματος είναι μία διαδικασία πέντε βημάτων [BART02]:

1) *Κατηγοριοποίηση (classification)*
   Ο πράκτορας αξιολογεί τα γεγονότα, τις ενέργειες και τα αντικείμενα που "υπάρχουν" στον κόσμο τη συγκεκριμένη στιγμή, με βάση την εκ των προτέρων γνώση που έχει. Ακολουθώντας την κατάλληλη διαδρομή στο παραπάνω διάγραμμα, καταλήγουμε στην κατηγορία συναισθήματος που αναλογεί στον πράκτορα με βάση την τρέχουσα κατάσταση του κόσμου.
   Παρατηρήσεις: το μοντέλο OCC προβλέπει 22 ξεχωριστές κατηγορίες συναισθημάτων. Τόσο πολλά διακριτά συναισθήματα είναι δύσκολο να αποδοθούν και επιπλέον μερικά μπορεί να είναι πολύ συγγενικά μεταξύ τους. Για να μειωθεί η πολυπλοκότητα του μοντέλου OCC, ο Ortony πρότεινε πέντε θετικές κατηγορίες συναισθημάτων (χαρά, ελπίδα, ανακούφιση, περηφάνια, ευγνωμοσύνη, αγάπη) και πέντε αρνητικές (αναστάτωση, φόβο, απογοήτευση, τύψη, θυμό, μίσος).

2) *Ποσοτικοποίηση (quantification)*
   Έχοντας βρει ποια κατηγορία συναισθήματος επηρεάζεται, βρίσκουμε το πόσο επηρεάζεται αυτή με βάση την ένταση του γεγονότος / ενέργειας / αντικειμένου που προκάλεσε το συναίσθημα.
   Παρατηρήσεις: Χρήσιμη θα ήταν η διατήρηση ενός *ιστορικού (history)* των γεγονότων / ενεργειών / αντικειμένων που είχαν προηγηθεί της τρέχουσας κατάστασης του κόσμου. Έτσι π.χ. ένα γεγονός που επαναλαμβάνεται θα έχει όλο και λιγότερη επίδραση στη συναισθηματική κατάσταση του πράκτορα.

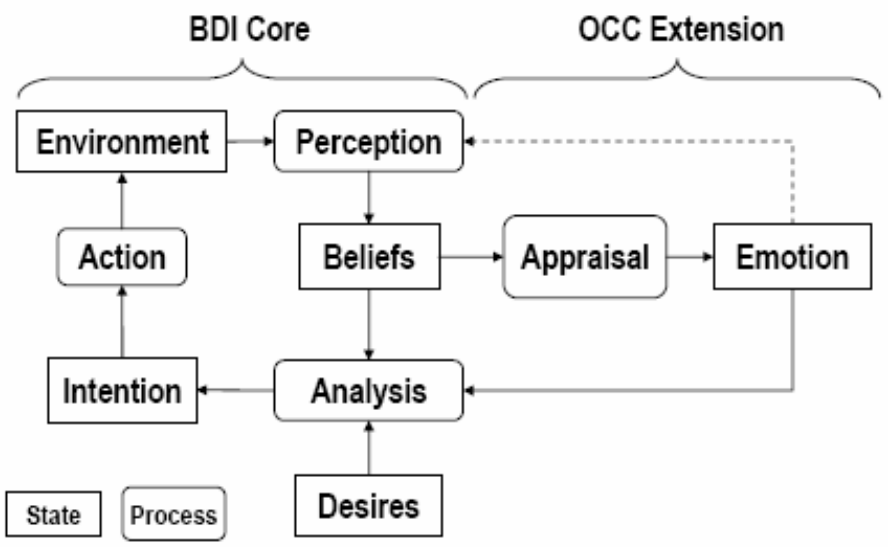
3) *Αλληλεπίδραση (interaction)*
   (δεν προβλέπεται από το κλασσικό μοντέλο OCC)
   Το συναίσθημα που προκύπτει από την τρέχουσα κατάσταση του κόσμου αλληλεπιδρά με και τροποποιεί τη συναισθηματική κατάσταση που είχε διαμορφώσει ο πράκτορας προηγουμένως, π.χ. αν ήταν λυπημένος και γίνει κάτι θετικό δεν θα γίνει κατ' ευθείαν χαρούμενος αλλά λιγότερο λυπημένος, ανάλογα και με την ένταση του γεγονότος / ενέργειας / αντικειμένου.

4) *Αντιστοίχηση (mapping)*
   Το μοντέλο OCC προβλέπει 22 κατηγορίες συναισθημάτων. Αν ένας πράκτορας δεν μπορεί να τις αναπαραστήσει όλες, πρέπει να γίνει κάποια αντιστοίχηση μεταξύ των κατηγοριών που παράγει το μοντέλο και αυτών που πραγματικά μπορεί να αναπαραστήσει ο πράκτορας. Σύμφωνα με τον Ekman, αν χρησιμοποιούμε μόνο εκφράσεις του προσώπου για να αποδώσουμε τα συναισθήματα του πράκτορα, αρκούν έξη βασικές κατηγορίες συναισθημάτων (χαρά, λύπη, θυμός, απέχθεια, φόβος, έκπληξη). Με βάση αυτά, όλα τα θετικά συναισθήματα αντιστοιχίζονται σε 'χαρά' (αποδιδόμενη με διάφορα χαμόγελα), τα αρνητικά σε 'λύπη', 'θυμό', 'απέχθεια' ή 'φόβο', ενώ η 'έκπληξη' δεν αντιστοιχεί σε καμία κατηγορία συναισθημάτων του μοντέλου OCC.

5) *Έκφραση (expression)*
   Τέλος, ο πράκτορας πρέπει να εκφράσει την συναισθηματική του κατάσταση με τρόπο που να γίνει αντιληπτή. Όλες οι εκφάνσεις του πράκτορα (έκφραση προσώπου, ομιλία, στάση σώματος, ενέργειες κλπ.) πρέπει να είναι συνεπείς στην απόδοση της ίδιας συναισθηματικής κατάστασης.

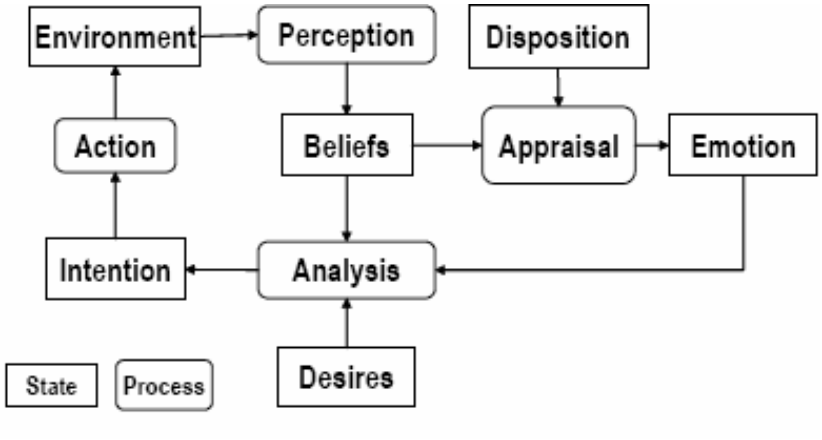
Συμπερασματικά, ο Bartneck προτείνει την επέκταση του μοντέλου OCC με ιστορικό (βλ. βήμα 2), αλληλεπίδραση (βλ. βήμα 3), και με ιδιαίτερη προσωπικότητα για κάθε πράκτορα.

Μοντέλο BDI (Belief-Desire-Intention).
Στο μοντέλο BDI (Belief-Desire-Intention), το περιβάλλον (environment) καθορίζει τα πιστεύω (beliefs) μέσω της αντίληψης (perception). Οι επιθυμίες (desires) είναι σταθερές στο χρόνο. Τα πιστεύω και οι επιθυμίες τροφοδοτούν το στάδιο της ανάλυσης (analysis), που δίνει στη έξοδό του τις προθέσεις (intensions), που με τη σειρά τους παράγουν ενέργειες (actions) οι οποίες αλλάζουν το περιβάλλον. Σχηματική αναπαράσταση του μοντέλου BDI (με μία επέκταση για τη μοντελοποίηση συναισθημάτων του μοντέλου OCC: η εκτίμηση (appraisal) των πιστεύω παράγει συναισθήματα (emotions), τα οποία με τη σειρά τους μπορούν να επηρεάζουν την ανάλυση αλλά και την αντίληψη) [PARU06]:



Σχ.3 Μοντέλο BDI με επέκταση OCC

Μοντέλο DETT (Disposition, Emotion, Trigger, Tendency).
Το μοντέλο DETT από [PARU06] επεκτείνει το παραπάνω μοντέλο ως εξής: προσθέτει έναν παράγοντα προδιάθεσης (disposition) που διαχωρίζει τους πράκτορες ως προς το πώς τα πιστεύω ενεργοποιούν τα αντίστοιχα συναισθήματα. Έτσι μπορεί να μοντελοποιήσει π.χ. καταστάσεις μάχης όπου ένας βετεράνος πράκτορας, λόγω εμπειρίας, δεν είναι τόσο ευπαθής σε κάποια συναισθήματα όσο ένας νεοσύλλεκτος πράκτορας. Η προδιάθεση, όπως και οι επιθυμίες, είναι σταθερή στο χρόνο. Σχηματικά το μοντέλο DETT είναι [PARU06]:



Σχ. 4 Μοντέλο DETT

Gameplay
Σύμφωνα με [CRAI], η εκμάθηση ενός παιχνιδιού συνίσταται από:

- *μηχανισμούς αλληλεπίδρασης (interaction mechanics)*
  οι μηχανικές κινήσεις που απαιτούνται π.χ. για τη λειτουργία του ποντικιού και του πληκτρολογίου.
- *σημασιολογία αλληλεπίδρασης (interaction semantics)*
  η συσχέτιση μιας ακολουθίας μηχανισμών αλληλεπίδρασης με τις εντός παιχνιδιού ενέργειες στις οποίες αντιστοιχούν.
- *επάρκεια gameplay (gameplay competence)*
  η ικανότητα του παίκτη να επιλέξει, δεδομένης της τρέχουσας κατάστασης του παιχνιδιού, και να εκτελέσει εκείνες τις εντός παιχνιδιού ενέργειες οι οποίες θα προωθήσουν την εξέλιξη του παιχνιδιού.

Οι *μηχανισμοί αλληλεπίδρασης* δεν είναι συνήθως ειδικοί για τα παιχνίδια, και καλύπτονται από τη γενικότερη γνώση του παίκτη για τη χρήση υπολογιστών.
Η *σημασιολογία αλληλεπίδρασης* μπορεί να είναι μεταφέρσιμη μεταξύ παιχνιδιών του ίδιου αλλά και διαφορετικού είδους (π.χ. η κίνηση γίνεται κατά κανόνα με τα πλήκτρα W,A,S,D), και, απ' τη στιγμή που ο χρήστης εξοικειωθεί μαζί της, γίνεται σε μεγάλο βαθμό ασυνείδητα.
Αφού ο παίκτης εξοικειωθεί αρκετά με το παιχνίδι ώστε οι δύο παραπάνω μηχανισμοί να γίνονται ασυνείδητα, το ενδιαφέρον εστιάζεται στο *gameplay*, που καθορίζει σε πολύ μεγάλο βαθμό και την αίσθηση που θα αποκομίσει ο παίκτης από το παιχνίδι (σε συνδυασμό με το σενάριο, τους χαρακτήρες, την ατμόσφαιρα του παιχνιδιού κλπ.). Η επάρκεια gameplay εμπεριέχει την ικανότητα του παίκτη να [CRAI]:

1) αποκωδικοποιήσει την οπτικοακουστική πληροφορία που του παρέχεται σε κατανόηση της τρέχουσας κατάστασης του εικονικού κόσμου
2) αξιολογήσει αυτή την κατανόηση σε σχέση με τους στόχους και την κατάσταση του χαρακτήρα που αντιπροσωπεύει τον παίκτη στο παιχνίδι καθώς και την προβλεπόμενη ανταμοιβή
3) πάρει αποφάσεις για τις επόμενες ενέργειες και τη στρατηγική που θα ακολουθήσει με βάση την παραπάνω κατανόηση και αξιολόγηση

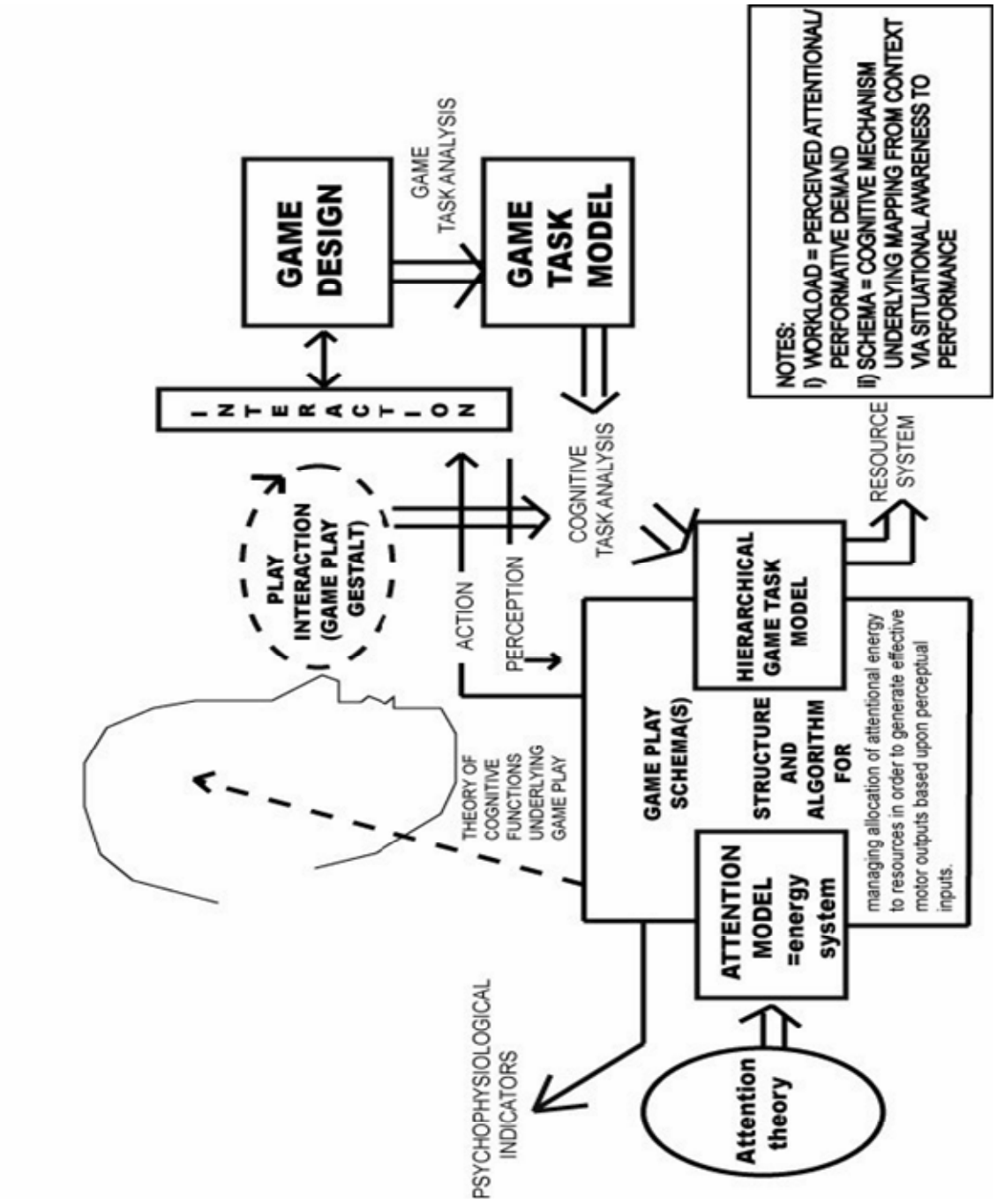
4) εκτελέσει τις επιθυμητές ενέργειες χρησιμοποιώντας τους μηχανισμούς και τη σημασιολογία αλληλεπίδρασης

Αυτή η επαναλαμβανόμενη νοητική διαδικασία, που μπορεί να περιγραφεί σαν μια ακολουθία αντίληψης->μοντελοποίησης->αξιολόγησης->σχεδιασμού->δράσης, είναι το βασικό νοητικό αποτέλεσμα της διαδικασίας εκμάθησης του τρόπου με τον οποίο παίζεται ένα παιχνίδι.
Η χρησιμότητα των διαφορετικών μορφών μάθησης σχετίζεται και με τη δυνατότητα η γνώση και οι δεξιότητες που αποκτήθηκαν μέσω αυτής της μάθησης να μεταφερθούν σε διαφορετικά περιβάλλοντα εκτός του ίδιου του παιχνιδιού. Στο επίπεδο της επάρκειας gameplay ειδικά, η αποκτηθείσα γνώση και δεξιότητες μπορεί να μεταφερθούν όχι μόνο σε άλλα παιχνίδια, αλλά και σε εντελώς διαφορετικά περιβάλλοντα (π.χ. η γνώση και οι δεξιότητες που αποκτήθηκαν σε έναν προσομοιωτή πτήσης μπορούν να μεταφερθούν στο περιβάλλον ενός πραγματικού αεροπλάνου).
Η ανάπτυξη νοητικών δομών μέσω του gameplay μπορεί εκτός της μεταφοράς γνώσης και δεξιοτήτων σε διαφορετικά περιβάλλοντα (εκπαίδευση) να χρησιμοποιηθεί και για θεραπευτικούς σκοπούς (και φυσικά για ψυχαγωγία).
Ονομάζουμε *σχήμα (schema)* μια νοητική δομή που συνδέει δηλωτική (γεγονότα) και διαδικαστική (δράσεις) γνώση, δημιουργώντας πρότυπα που διευκολύνουν την κατανόηση και εκδήλωση των κατάλληλων ενεργειών σε κάποιο περιβάλλον. Π.χ. ένα σχήμα που σκιαγραφεί τον τρόπο αλληλεπίδρασης και τις αποφάσεις του παίκτη στο Neverwinter Nights θα μπορούσε να είναι [CRAI] :

1. σταμάτα όταν βρεις μια κλειστή πόρτα
2. έλεγξε την υγεία των μελών της ομάδας
    αν περισσότερα από ένα μέλη της ομάδας έχουν χαμηλή υγεία, τότε:
        κάνε rest
        κάνε πάλι όσα summons χρειάζεται
    αλλιώς
        αν ένα μέλος της ομάδας έχει χαμηλή υγεία, τότε:
            αν έχεις πολλά healing potions, τότε:
                γιάτρεψέ τον με potions
            αλλιώς
                κάνε rest
                κάνε πάλι όσα summons χρειάζεται
3. διέταξε την ομάδα σε σχηματισμό μάχης
4. άνοιξε την πόρτα και μπες στο δωμάτιο
5. αν υπάρχουν αντίπαλοι, τότε
        διάλεξε έναν στόχο
        παρακολούθα την υγεία της ομάδας έως ότου νικηθεί ο στόχος
        αν περισσότερα από ένα μέλη της ομάδας έχουν χαμηλή υγεία, τότε:
            απομάκρυνε την ομάδα
            κάνε rest
            κάνε πάλι όσα summons χρειάζεται
            πήγαινε στο βήμα 3
        αλλιώς
            αν ένα μέλος της ομάδας έχει χαμηλή υγεία, τότε:
                αν έχεις πολλά healing potions, τότε:
                    γιάτρεψέ τον με potions

αλλιώς
απομάκρυνε την ομάδα
κάνε rest
κάνε πάλι όσα summons χρειάζεται
πήγαινε στο βήμα 3
6. Αν υπάρχουν άλλοι αντίπαλοι, πήγαινε στο βήμα 5
7. Έλεγξε για παγίδες
... κλπ ...

Πλαίσιο για τη νοητική ανάλυση του gameplay [CRAI]:



Σχ.5 Πλαίσιο για τη νοητική ανάλυση του gameplay

Χρήση ηλεκτρονικών παιχνιδιών για τη μελέτη συναισθημάτων

Μελέτες έχουν χρησιμοποιήσει το περιβάλλον διαφόρων παιχνιδιών για τη μελέτη των αντιδράσεων διαφόρων παικτών υπό ελεγχόμενες συνθήκες παιχνιδιού.
Στο [REEK04], οι παίκτες χειρίζονταν ένα διαστημόπλοιο και προσπαθούσαν να μαζέψουν κρυστάλλους και να προχωρήσουν στην επόμενη πίστα, στο γαλαξία υπήρχαν και αντίπαλοι και νάρκες που δυσκόλευαν την επίτευξη του στόχου, ενώ η δυσκολία των επιπέδων αυξανόταν σταδιακά. Το παιχνίδι χρησιμοποιήθηκε για τη μελέτη ψυχο-φυσιολογικών αντιδράσεων σε γεγονότα σχετιζόμενα με συναισθήματα.

Συγκεκριμένα μελετήθηκε η εγγενής ευχαρίστηση (intrinsic pleasantness) και η προώθηση στόχων (goal conduciveness) των γεγονότων που συνέβαιναν κατά τη διάρκεια του παιχνιδιού, δύο διαστάσεις που έχουν προταθεί στη θεωρία εκτίμησης (appraisal theory).
Στο [SCHE02], μελετήθηκε η αντίδραση των παικτών στην προσπάθειά τους να χρησιμοποιήσουν ένα επίτηδες δυσλειτουργικό interface, και τα επίπεδα απογοήτευσης που προκαλούνται υπό τέτοιες συνθήκες. Μάλιστα οι παίκτες δεν γνώριζαν αρχικά τον πραγματικό σκοπό της μελέτης, αλλά πίστευαν ότι πρόκειται για έναν διαγωνισμό, συνεπώς η δυσλειτουργία του interface εκλαμβανόταν ως ένα εμπόδιο για την επίτευξη του στόχου τους.

Η καταγραφή των αντιδράσεων των παικτών στις παραπάνω μελέτες βασίζεται σε μετρήσεις χαρακτηριστικών του ανθρώπινου οργανισμού όπως αγωγιμότητα του δέρματος, πίεση του αίματος, καρδιαγγειακή δραστηριότητα, μυϊκή δραστηριότητα, κλπ.

Βασικό Λεξιλόγιο:
agent **ό** πράκτορας
emotion **ό** συναίσθημα
mood **ό** διάθεση
personality **ό** προσωπικότητα

Κατάληξη
Τα παραπάνω δίνουν μια ιδέα για την έρευνα που γίνεται σχετικά με την ανάπτυξη εικονικών οντοτήτων οι οποίες θα αλληλεπιδρούν με τον χρήστη με τρόπο κατά το δυνατόν αληθοφανή. Αν τώρα τοποθετήσουμε τέτοιες οντότητες σε ένα εικονικό περιβάλλον, δημιουργούμε έναν *εικονικό κόσμο (virtual world)*, στον οποίο ο χρήστης έχει την αίσθηση ότι συμμετέχει ενεργά. Εφαρμογές τέτοιων εικονικών κόσμων μπορεί να κυμαίνονται από εξομοιωτές και προγράμματα μάθησης μέχρι ηλεκτρονικά παιχνίδια και προγράμματα ψυχαγωγίας.
Στο υπόλοιπο του κειμένου αναπτύσσουμε μία εφαρμογή που ανήκει στην τελευταία κατηγορία. Πρόκειται για έναν εικονικό κόσμο, με στοιχεία που συναντώνται συνήθως στα παιχνίδια ρόλων (Role-Playing Games –RPGs), όπου ο χρήστης εξερευνά τον κόσμο, συνομιλεί και αλληλεπιδρά με τους διάφορους πράκτορες και τα αντικείμενα που βρίσκονται σε αυτόν. Οι πράκτορες αυτοί έχουν ένα στοιχειώδες πρότυπο συμπεριφοράς που τους επιτρέπει να συμπεριφέρονται διαφορετικά στον παίκτη ανάλογα με τις μέχρι τώρα ενέργειές του (π.χ. θυμώνουν αν ο παίκτης είναι προσβλητικός απέναντί τους). Πάντως, η λειτουργικότητα αυτή επιτυγχάνεται με απλές μηχανές κατάστασης και είναι καθαρά script-based, οπότε μία πιθανή επέκταση αυτού του παραδείγματος θα μπορούσα να είναι η χρήση ενός εκ των μοντέλων στα οποία έγινε αναφορά παραπάνω. Ο χρήστης αναλαμβάνει και προσπαθεί να φέρει σε πέρας διάφορες αποστολές, με απώτερο στόχο την ολοκλήρωση του παιχνιδιού. Το παιχνίδι ουσιαστικά αποτελείται από ένα χάρτη στον οποίο εξελίσσονται όλες οι αποστολές, και ο παίκτης πρέπει να ολοκληρώσει την τρέχουσα αποστολή πριν ξεκλειδώσει την επόμενη. Η πρόοδός του ανταμείβεται με αύξηση των στατιστικών του και ενεργοποίηση νέων επιθετικών και αμυντικών ικανοτήτων που τον βοηθούν να αντεπεξέλθει στην αυξανόμενη δυσκολία των αντιπάλων που αντιμετωπίζει. Ο ήχος τονίζει την εξέλιξη του κόσμου, στην αρχή είναι ουδέτερος και αλλάζει σε ένα πιο υποβλητικό θέμα όταν η κατάσταση αρχίζει

να δυσκολεύει. Παρομοίως, ένα μικρό μουσικό κομμάτι επιβραβεύει την νίκη του παίκτη στη μάχη ενώ η ήττα συνοδεύεται από ένα ανάλογο θέμα.

2. ***Εισαγωγή στο Torque***

Η Torque Game Engine (TGE), ή για συντομία Torque, είναι, όπως υποδηλώνει και το όνομα, μία μηχανή παιχνιδιών (game engine). Παρέχει δηλαδή μία βάση για την ανάπτυξη λογισμικού που απευθύνεται σε έναν τομέα της ψηφιακής ψυχαγωγίας, συγκεκριμένα τα ηλεκτρονικά παιχνίδια (computer games), έναν τομέα που είναι πλέον από μόνος του μία ολόκληρη βιομηχανία.

Ο ρόλος μίας τέτοιας μηχανής είναι να διευκολύνει την ανάπτυξη νέων έργων λογισμικού αυτής της κατηγορίας. Αναλαμβάνει βασικούς τομείς, όπως τη δημιουργία των γραφικών (graphics engine - rendering), την ανίχνευση συγκρούσεων μεταξύ τον μοντελοποιημένων αντικειμένων (collision detection), την επικοινωνία με το παραθυρικό περιβάλλον (windowing system), με το σύστημα αρχείων (file system) και γενικότερα με το λειτουργικό σύστημα (OS), αλλά και θέματα όπως το μοντέλο φυσικής (physics engine) ή λειτουργίες δικτύωσης. Ακόμα είναι πιθανό οι δημιουργοί μιας τέτοιας μηχανής να αναπτύσσουν μια απλοποιημένη γλώσσα προγραμματισμού (scripting language) "πάνω" από την ίδια τη μηχανή, και να την παρέχουν μαζί με τη μηχανή σαν εργαλείο ανάπτυξης για τους κατασκευαστές (developers). Ο στόχος και σε αυτή την περίπτωση είναι η απλοποίηση της διαδικασίας ανάπτυξης νέων τίτλων, η ευκολότερη πρόσβαση στα επιμέρους τμήματα της μηχανής, τα οποία έτσι μπορεί να τα χειριστεί ο χρήστης με ενιαίο τρόπο, η μείωση του χρόνου ανάπτυξης και συνεπώς και του κόστους.

Στη συγκεκριμένη εργασία, χρησιμοποιείται το Torque, το οποίο αναλαμβάνει όλους αυτούς τους τομείς και αρκετούς ακόμα. Χρησιμοποιώντας μία τέτοια μηχανή σαν βάση, μπορούμε να επικεντρωθούμε στην αλληλεπίδραση ανθρώπου-μηχανής, παρουσιάζοντας διάφορες περιπτώσεις τέτοιας αλληλεπίδρασης. Για αυτό το σκοπό, οι scripting δυνατότητες του Torque είναι συνήθως αρκετές, και το μεγαλύτερο μέρος της εργασίας είναι γραμμένο στη γλώσσα που λέγεται TorqueScript. Το Torque είναι όπως είπαμε ένα εργαλείο για την ανάπτυξη παιχνιδιών, και τα τελευταία είναι ίσως από τα πιο απαιτητικά έργα λογισμικού όσον αφορά την αλληλεπίδραση του ανθρώπου με τον υπολογιστή.

3. ***TorqueScript***

Σκοπός εδώ είναι η παρουσίαση κάποιων στοιχείων του Torque, όπως αυτά είναι διαθέσιμα μέσω της TorqueScript, αλλά όχι η αναλυτική παρουσίαση της ίδιας της γλώσσας και της σύνταξής της (για κάτι τέτοιο βλ. βιβλιογραφία). Επιπλέον, αυτά παρουσιάζονται μόνο στα πλαίσια που είναι σχετικά με την παρούσα εργασία, συνεπώς πρόκειται για μερική και όχι πλήρη περιγραφή.

Η TorqueScript είναι μία απλοποιημένη γλώσσα προγραμματισμού υψηλού επιπέδου και ανήκει στην κατηγορία των λεγόμενων scripting γλωσσών.

Σύμφωνα με τους δημιουργούς του Torque, είναι σαν μία C++ *χωρίς τύπους* (*typeless* – δηλ. ο τύπος των μεταβλητών ποτέ δεν δηλώνεται, και μπορεί η ίδια μεταβλητή να χρησιμοποιηθεί άλλοτε σαν μεταβλητή ακεραίων, άλλοτε σαν μεταβλητή συμβολοσειρών -string- κλπ.).

Η *εμβέλεια (scope)* των μεταβλητών μπορεί να είναι είτε τοπική (local), οπότε η μεταβλητή είναι προσβάσιμη και έγκυρη μόνο εντός του block όπου δηλώθηκε, και της μεταβλητής προηγείται ένα "%", π.χ. στη δήλωση:

```
%localVar = value;
```

είτε παγκόσμια (global), οπότε η μεταβλητή είναι προσβάσιμη και έγκυρη παντού στον κώδικα -και σε όλα τα αρχεία-, και της μεταβλητής προηγείται ένα "$", π.χ. στη δήλωση:

```
$globalVar = value;
```

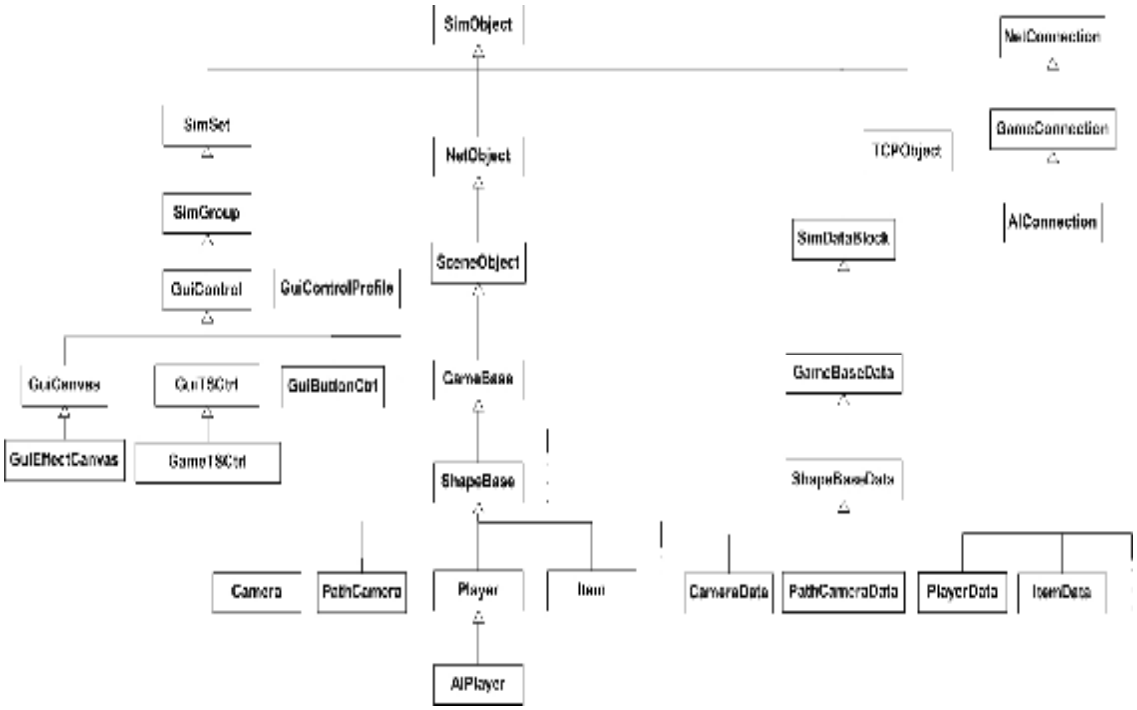
(κατά σύμβαση ξεκινάμε τα ονόματα των μεταβλητών με μικρά γράμματα. Πάντως, η TorqueScript *δεν διαχωρίζει μεταξύ κεφαλαίων και πεζών -case insensitive-*)

Στην πράξη είναι πολύ πιο απλή από την C++, αλλά αυτός είναι και ο λόγος ύπαρξης των scripting γλωσσών. Βασικό χαρακτηριστικό είναι ο *αντικειμενοστραφής (object oriented) χαρακτήρας* της γλώσσας.

Η γλώσσα υλοποιεί μία μεγάλη ιεραρχία κλάσεων, παρέχοντας στάνταρ λειτουργικότητα σε τομείς που εκτείνονται από το χειρισμό και τον καθορισμό της συμπεριφοράς των αντικειμένων του κόσμου έως τη δικτύωση. Π.χ. η κλάση (class) ShapeBase αποτελεί τη βάση για όλες τις κλάσεις των αντικειμένων (objects) που έχουν κάποιο σχήμα και είναι ορατά στον κόσμο, ορίζοντας τις ιδιότητες (attributes / fields) και τις μεθόδους (methods) που είναι κοινές για τις κλάσεις αυτών των αντικειμένων. Η κλάση Player είναι υποκλάση (subclass) της ShapeBase, και χρησιμοποιείται για τη δημιουργία αντικειμένων για τους παίκτες και την παρουσία αυτών μέσα στον κόσμο του παιχνιδιού (avatars). Κληρονομεί όλη τη λειτουργικότητα της ShapeBase, και επιπλέον ορίζει εξειδικευμένες ιδιότητες και μεθόδους χρήσιμες για αντικείμενα αυτής της κατηγορίας. Ο χρήστης μπορεί να ορίσει επιπλέον ιδιότητες / μεθόδους με τρόπους που θα δούμε παρακάτω, προσθέτοντας τις δικές του λειτουργίες σε αυτές που προσφέρει το Torque. Θα αναφερόμαστε σε αυτές τις κλάσεις σαν *κλάσεις κονσόλας (console classes)*.
(*Σημείωση*: φαίνεται πως στο Torque αυτές οι κλάσεις αναφέρονται σαν "κλάσεις αντικειμένων" – "object classes" – που είναι κάπως αντιφατικό, και κάποιες φορές ακόμα και σαν "console objects", ενώ ουσιαστικά πρόκειται για τις κλάσεις, όχι για συγκεκριμένα αντικείμενα / στιγμιότυπα (instances) αυτών.)

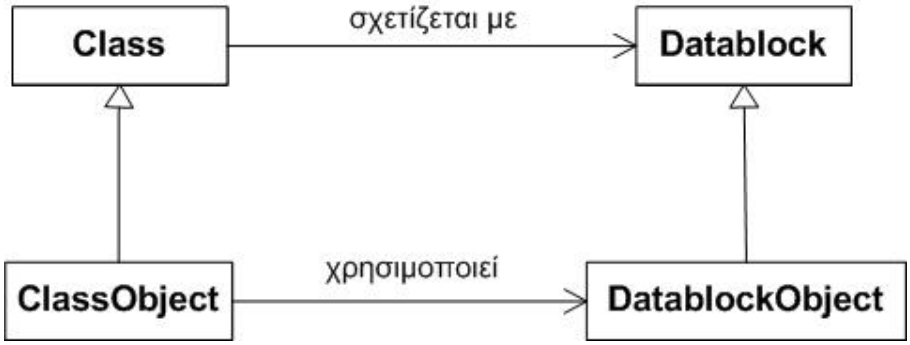
Ακολουθεί μία μερική παρουσίαση αυτής της ιεραρχίας (δέντρο κληρονομικότητας), που παρουσιάζει τη γενική δομή της χωρίς όμως να είναι πλήρης, καθώς κάτι τέτοιο θα έκανε το μεγέθους του διαγράμματος απαγορευτικό.

Σχ. 6 Δένδρο κληρονομικότητας – ιεραρχία κλάσεων/datablocks/Gui κλάσεων/profile κονσόλας



Όπως φαίνεται στο διάγραμμα, για ορισμένες κλάσεις υπάρχουν και κάποιες συμπληρωματικές τους, με την κατάληξη "Data". Αυτές οι συμπληρωματικές κλάσεις λέγονται *datablocks*, και λέμε ότι στην κλάση π.χ. Player αντιστοιχεί το datablock PlayerData. Τόσο οι "κανονικές" κλάσεις όσο και τα datablocks είναι εξ' ίσου κλάσεις, όπως αυτές εννοούνται στον αντικειμενοστραφή προγραμματισμό. *Από εδώ και πέρα θα αναφερόμαστε στις μεν πρώτες απλώς ως κλάσεις και στις δεύτερες ως datablocks.*

Όταν δημιουργούμε ένα αντικείμενο μίας κλάσης στην οποία αντιστοιχεί κάποιο datablock, χρειάζεται (*απαιτείται*) να έχουμε προηγουμένως δημιουργήσει και ένα αντικείμενο αυτού του datablock, ώστε να συσχετίσουμε το αντικείμενο της κλάσης με ένα αντικείμενο του datablock. Σχηματικά:



Σχ. 7 Σχέση κλάσεων, datablocks και αντικειμένων / στιγμιοτύπων αυτών

(Να σημειωθεί ότι το παραπάνω δεν είναι διάγραμμα UML, παρ' όλο που χρησιμοποιεί στοιχεία της UML).

Γενικά, τα datablocks λειτουργούν ως μία συλλογή ιδιοτήτων για την κλάση στην οποία αντιστοιχούν. Κατά κάποιον τρόπο, η κλάση ορίζει *μεθόδους* για τον χειρισμό των αντικειμένων της, και το datablock που της αντιστοιχεί ορίζει *ιδιότητες* που καθορίζουν διάφορα χαρακτηριστικά αυτών των αντικειμένων (αυτά για τις ενσωματωμένες ιδιότητες και μεθόδους, όχι για αυτές που ορίζει ο χρήστης). Οι ιδιότητες που ορίζονται σε ένα datablock αρχικοποιούνται κατά τη δημιουργία ενός αντικειμένου από αυτό το datablock. Κατά κανόνα έχουν άμεση αντιστοιχία και επίδραση σε λειτουργίες που διεκπεραιώνει εσωτερικά η μηχανή για το αντικείμενο που χρησιμοποιεί αυτό το αντικείμενο του datablock (που είναι όπως είδαμε αντικείμενο της κλάσης στην οποία αντιστοιχεί αυτό το datablock).

Για παράδειγμα, η κλάση Player ορίζει μεθόδους όπως η setActionThread() που μπορούμε να καλέσουμε για την επιλογή κάποιου animation προς εκτέλεση. Το datablock PlayerData που της αντιστοιχεί ορίζει ιδιότητες όπως η renderFirstPerson, που καθορίζει αν το αντικείμενο / παίκτης θα εμφανίζεται και όταν είμαστε σε προοπτική 1ου προσώπου (1st POV), ιδιότητες που καθορίζουν την ταχύτητα του αντικειμένου, τις δυνάμεις που επιδρούν σε αυτό και που θα χρησιμοποιηθούν εσωτερικά από το μοντέλο φυσικής (physics engine) κλπ..

Ένα αντικείμενο ενός datablock σχετίζεται με ένα αντικείμενο μιας κλάσης κατά τη δημιουργία του τελευταίου. Ενδεικτικά :

```
// δημιουργία ενός αντικειμένου του datablock
datablock DatablockName (DatablockObjectName)
{
        // αρχικοποίηση ιδιοτήτων που ορίζονται στο datablock
        datablockAttribute1 = value1_1;
        .
        .
        .

        // ορισμός ΚΑΙ αρχικοποίηση καθοριζόμενων από το χρήστη ιδιοτήτων
        // για το συγκεκριμένο αντικείμενο του datablock
        datablockObjectAttribute1 = value2_1;
        .
        .
        .
} ;

// δημιουργία ενός αντικειμένου της κλάσης στην οποία αντιστοιχεί το
```

```
// παραπάνω datablock
new ClassName (ClassObjectName)
{
	// καθορισμός του αντικειμένου του datablock που θα χρησιμοποιείται
	// από αυτό το αντικείμενο της κλάσης (αν πρόκειται για αντικείμενο
	// κλάσης στην οποία αντιστοιχεί κάποιο datablock)
	datablock = DatablockObjectName;

	// αρχικοποίηση άλλων ιδιοτήτων που ορίζονται στην κλάση
	classAttribute1 = value1_1;
	.
	.
	.

	// ορισμός ΚΑΙ αρχικοποίηση καθοριζόμενων από το χρήστη ιδιοτήτων
	// για το συγκεκριμένο αντικείμενο της κλάσης
	classObjectAttribute1 = value2_1;
	.
	.
	.
} ;
```

(κατά σύμβαση ξεκινάμε τα ονόματα των κλάσεων / datablocks και των αντικειμένων με κεφαλαία γράμματα)
Σημείωση: οι ιδιότητες κλάσεων / datablocks δεν έχουν κάποιο προσδιορισμό εμβέλειας (%, $). Η εμβέλειά τους είναι όλος ο χώρος ονομάτων (namespace) του συγκεκριμένου αντικειμένου. Ακόμα, συμπεριφέρονται σαν *δημόσιες (public)*, και το ίδιο ισχύει και για τις μεθόδους των κλάσεων / datablocks.

Γενικότερα, η δημιουργία αντικειμένων από μία κλάση, έστω την Player, γίνεται ως εξής:

```
%classObjectID = new Player (ClassObjectName : ParentClassObjectName)
{
	datablock = DatablockObjectName;  // όπου DatablockObjectName το
		// όνομα ενός  αντικειμένου του datablock PlayerData

	classAttribute1 = value1_1;
	.
	.
	.

	classObjectAttribute1 = value2_1;
	.
	.
	.
} ;
```

όπου %classObjectID είναι ένα μοναδικό αναγνωριστικό (αριθμητική τιμή, ID, handler), το οποίο δίνει η μηχανή στο αντικείμενο κατά τη δημιουργία του, και ClassObjectName είναι το όνομα που θέλουμε να δώσουμε στο αντικείμενο. Τα %classObjectID, ClassObjectName είναι προαιρετικά, αλλά πρέπει να υπάρχει τουλάχιστον ένα εκ των δύο αν θέλουμε αργότερα να αναφερθούμε στο αντικείμενο που φτιάξαμε. Εντός των { } μπορούμε να αποδώσουμε τιμές στις διάφορες ιδιότητες της κλάσης του αντικειμένου, κάτι σαν τους κατασκευαστές κλάσεων (constructors) που συναντά κανείς σε άλλες γλώσσες. Μία τέτοια ιδιότητα με ξεχωριστή σημασία

είναι και η ιδιότητα datablock, που όπως είδαμε καθορίζει ποιο αντικείμενο του αντίστοιχου datablock θα χρησιμοποιηθεί από το αντικείμενο της κλάσης, και κληρονομείται από την κλάση GameBase.

Μια διαφορά με άλλες γλώσσες είναι ότι εκτός των ιδιοτήτων που ορίζονται στην κλάση και αρχικοποιούνται κατά τη δημιουργία του αντικειμένου, μπορούν κατά τη δημιουργία του αντικειμένου να δηλωθούν και οποιεσδήποτε άλλες ιδιότητες. Ουσιαστικά οι ιδιότητες αυτές *ορίζονται* εδώ, ταυτόχρονα με την αρχικοποίησή τους. Δηλ. η TorqueScript *επιτρέπει τον ορισμό ιδιοτήτων για συγκεκριμένα αντικείμενα*. Παρόμοια, *επιτρέπει τον ορισμό μεθόδων για συγκεκριμένα αντικείμενα,* αρκεί αυτές να δηλωθούν μέσα στον αντίστοιχο χώρο ονομάτων, όπως θα κάναμε για να ορίσουμε νέες μεθόδους για μια κλάση / datablock (βλ. παρακάτω).

Ιδιαιτερότητα επίσης της TorqueScript είναι η δυνατότητα ένα αντικείμενο να κληρονομεί ένα άλλο αντικείμενο. Στο παραπάνω παράδειγμα το αντικείμενο ClassObjectName της κλάσης Player κληρονομεί το αντικείμενο ParentObjectName (που δεν επιβάλλεται να είναι και αυτό αντικείμενο της ίδιας κλάσης Player), δηλ. όλες οι ιδιότητες και μέθοδοι του ParentObjectName περνάνε και στο ObjectName. Μάλιστα οι σχέσεις κληρονομικότητας μεταξύ των αντικειμένων δεν φαίνονται πουθενά στο Torque, αν στο παραπάνω παράδειγμα γράψουμε στην κονσόλα ObjectName.dumpClassHierarrchy(); θα δούμε τα εξής:

```
Player ->
ShapeBase ->
GameBase ->
SceneObject ->
NetObject ->
SimObject
```

δηλ. ναι μεν το αντικείμενο ObjectName έχει αντιγράψει τις ιδιότητες / μεθόδους του αντικειμένου ParentObjectName, αλλά κατά τ' άλλα θεωρείται ότι κληρονομεί κατ' ευθείαν την κλάση της οποίας είναι αντικείμενο, εδώ την κλάση Player.

Η δημιουργία αντικειμένων από datablocks είναι παρόμοια με τη δημιουργία αντικειμένων από κλάσεις, για παράδειγμα:

```
datablock PlayerData (DatablockObjectName : ParentDatablockObjectName)
{
        datablockAttribute1 = value1_1;
        .
        .
        .

        datablockObjectAttribute1 = value2_1;
        .
        .
        .
} ;
```

με τις εξής διαφορές: 1^{ον} δεν μπορούμε να πάρουμε μία μεταβλητή με το αναγνωριστικό (ID, handle) του αντικειμένου ενός datablock, και 2^{ον} η δεσμευμένη λέξη "new" έχει αντικατασταθεί από τη δεσμευμένη λέξη "datablock".

Σύντομη ανακεφαλαίωση

Κλάσεις (console classes)

- Προκαθορισμένο δένδρο κληρονομικότητας.
- Δυνατότητα δημιουργίας αντικειμένων από αυτές.
- *Μη* δυνατότητα ορισμού νέων ιδιοτήτων σε αυτές.
- Δυνατότητα ορισμού νέων μεθόδων σε αυτές (στο χώρο ονομάτων τους).

Αντικείμενα κλάσεων

- Δυνατότητα να κληρονομήσουν ένα άλλο αντικείμενο κλάσης.
- Δυνατότητα ορισμού νέων ιδιοτήτων σε αυτά κατά τη δημιουργία τους.
- Δυνατότητα ορισμού νέων μεθόδων σε αυτά (στο χώρο ονομάτων τους).

Datablocks

- Εξ' ορισμού σχετίζονται με κάποια κλάση. Παρέχουν ένα σύνολο ιδιοτήτων για τα αντικείμενα της κλάσης με την οποία σχετίζονται.
- Προκαθορισμένο δένδρο κληρονομικότητας.
- Δυνατότητα δημιουργίας αντικειμένων από αυτά.
- *Μη* δυνατότητα ορισμού νέων ιδιοτήτων σε αυτά.
- Δυνατότητα ορισμού νέων μεθόδων σε αυτά (στο χώρο ονομάτων τους).

Αντικείμενα των datablocks

- Δεν έχουν ξεχωριστή υπόσταση, με την έννοια ότι πάντοτε χρησιμοποιούνται σαν τμήμα ενός αντικειμένου της κλάσης με την οποία σχετίζεται το datablock.
- Δυνατότητα να κληρονομήσουν ένα άλλο αντικείμενο κάποιου datablock.
- Δυνατότητα ορισμού νέων ιδιοτήτων σε αυτά κατά τη δημιουργία τους.
- Δυνατότητα ορισμού νέων μεθόδων σε αυτά (στο χώρο ονομάτων τους).

Ορισμός νέων μεθόδων

Μετά από όλα αυτά, ας δούμε πώς ορίζουμε μία μέθοδο (εδώ έχουμε την περίπτωση ορισμού μεθόδου σε μια κλάση, ο ορισμός μεθόδου σε αντικείμενο κλάσης, datablock ή αντικείμενο datablock είναι εντελώς ανάλογος)

```
function Player :: classMethod (%this, %arg1, %arg2, …)
{
        // σώμα μεθόδου
}
```

Το πρώτο όρισμα σε μια μέθοδο είναι η αναφορά στο συγκεκριμένο στιγμιότυπο / αντικείμενο της κλάσης / datablock για το οποίο κλήθηκε η μέθοδος (*the "this" reference*), την οποία περνάει αυτόματα η μηχανή στις μεθόδους των κλάσεων / datablocks όταν αυτές καλούνται. Δεν έχει σημασία πώς ονομάζουμε το $1^{ο}$ όρισμα, αρκεί να ξέρουμε ότι πάντα θα είναι αυτή η αναφορά, και ότι τα ορίσματα που θα περάσουμε εμείς στη μέθοδο ξεκινάνε από το $2^{ο}$ όρισμα και μετά. Π.χ. η κλήση της μεθόδου θα γινόταν κάπως έτσι:

```
%player = new Player() { … };
%player.classMethod(%arg1, %arg2, …);
```

Στην περίπτωση που έχουμε μέθοδο ενός datablock είναι πιθανό να συναντήσουμε τη μεταβλητή %db αντί της %this, που απλά μας θυμίζει ότι η αναφορά γίνεται σε ένα αντικείμενο datablock, αλλά δεν πρόκειται για κάτι διαφορετικό κατά τα άλλα.

Γενικές Παρατηρήσεις:

Όπως βλέπουμε στην TorqueScript τα αντικείμενα χρησιμοποιούνται με τρόπο που δεν ακολουθεί ακριβώς αυτά που θα περίμενε κανείς από μία αντικειμενοστραφή γλώσσα (ορισμοί ιδιοτήτων / μεθόδων για συγκεκριμένα αντικείμενα).

Το παραπάνω συνδέεται και με την εξής παρατήρηση: στην TorqueScript, *δεν υπάρχει κανένας τρόπος να δημιουργήσει ο χρήστης νέες κλάσεις / datablocks* (αυτό μπορεί να γίνει μόνο αν αλλάξει κανείς τον κώδικα της μηχανής του Torque). *Δεν* υπάρχει κάτι αντίστοιχο του

```
class ClassName
{
	// ιδιότητες και μέθοδοι
};
```

της C++, αλλά μόνο η δυνατότητα δημιουργίας αντικειμένων με τα new / datablock.
Από την άλλη, η TorqueScript επιτρέπει την επέκταση των διαθέσιμων κλάσεων / datablocks με ορισμό νέων μεθόδων στο χώρο ονομάτων τους (όχι όμως και ιδιοτήτων), και δίνει τη δυνατότητα εμπλουτισμού των αντικειμένων που δημιουργεί ο χρήστης με ιδιότητες και μεθόδους όπως αυτός θέλει, συν τη δυνατότητα "ψευδό"-κληρονομικότητας μεταξύ αντικειμένων για ευκολότερη δημιουργία νέων αντικειμένων που μοιάζουν με άλλα, χωρίς να χρειάζεται η επαναλαμβανόμενη δήλωση όλων των κοινών στοιχείων. Το μειονέκτημα είναι προφανώς η σύγχυση που δημιουργείται από την "παράδοξη" αυτή προσέγγιση.

Παραπάνω έγινε η διάκριση ότι οι κλάσεις ορίζουν μεθόδους για τον χειρισμό των αντικειμένων τους, και τα datablocks που τους αντιστοιχούν ορίζουν ιδιότητες που καθορίζουν διάφορα χαρακτηριστικά αυτών των αντικειμένων. Γενικά, στις κλάσεις παρατηρεί κανείς ελάχιστες έως καθόλου ιδιότητες και κυρίως μεθόδους, και στα datablocks το αντίθετο. *Οι λίγες ιδιότητες που εμφανίζονται στις κλάσεις έχει νόημα να ορίζονται για κάθε αντικείμενο κλάσης ξεχωριστά (π.χ. datablock, position, scale) και δεν θα είχε νόημα να τις χρησιμοποιήσουμε ομαδικά για πολλά αντικείμενα κλάσεων, όπως μπορούμε να κάνουμε με τις ιδιότητες που ορίζονται στα datablocks.* Φαίνεται δηλ. ότι στην TorqueScript, όπου ήταν δυνατό οι ιδιότητες και οι μέθοδοι διαχωρίστηκαν επίτηδες. Τα datablocks ενθυλακώνουν τις ιδιότητες που είναι σχετικές με τα αντικείμενα της αντίστοιχης κλάσης, και το ίδιο datablock μπορεί να χρησιμοποιηθεί από πολλά διαφορετικά αντικείμενα αυτής της κλάσης, με αποτέλεσμα να αποφεύγεται η επανάληψη της αρχικοποίησης των ιδιοτήτων για κάθε αντικείμενο με παρόμοια λειτουργικότητα. Η διάκριση δεν είναι εντελώς ακριβής, και σίγουρα δεν είναι δεσμευτική για της ιδιότητες και μεθόδους που ορίζει ο χρήστης (στα αντικείμενα των κλάσεων μπορούμε να ορίσουμε ιδιότητες αν θέλουμε, όπως και στα datablocks και τα αντικείμενά τους μπορούμε να ορίσουμε μεθόδους), και από 'κει και πέρα είναι θέμα του χρήστη να αποφασίσει πού ταιριάζει καλύτερα ο ορισμός μιας ιδιότητας / μεθόδου :
Στην κλάση, οπότε θα ισχύει για κάθε αντικείμενο της κλάσης -αλλά και των υποκλάσεών της- ανεξάρτητα από το datablock που χρησιμοποιεί;
Στο αντικείμενο της κλάσης, οπότε θα ισχύει μόνο για το συγκεκριμένο αντικείμενο – και για όσα αντικείμενα το κληρονομούν-;
Στο datablock, οπότε θα ισχύει για όλα τα αντικείμενα κλάσεων που χρησιμοποιούν αντικείμενα αυτού του datablock ή των datablocks που το κληρονομούν;
Στο αντικείμενο του datablock, οπότε θα ισχύει για όλα τα αντικείμενα κλάσεων που χρησιμοποιούν αυτό το αντικείμενο ή αντικείμενα που το κληρονομούν;

Η σημαντικότερη διάκριση μεταξύ των μεθόδων / ιδιοτήτων που είναι ήδη ορισμένες και των μεθόδων / ιδιοτήτων που ορίζει ο χρήστης, είναι ότι οι πρώτες κατά κανόνα (υπάρχουν και εξαιρέσεις) αντιστοιχούν σε εσωτερικές λειτουργίες της μηχανής, ενώ οι δεύτερες δεν σημαίνουν τίποτα από μόνες τους, παρά μόνο έχουν τη σημασία που τους αποδίδει ο χρήστης. Αυτό ίσως φαίνεται αυτονόητο, αλλά είναι σημαντικό. Για παράδειγμα, το datablock ShapeBaseData ορίζει την ιδιότητα maxDamage που δείχνει τη μέγιστη ζημιά που μπορεί να αντέξει ένα αντικείμενο πριν καταστραφεί. Η κλάση ShapeBase ορίζει τη μέθοδο applyDamage(), και χρησιμοποιώντας την προσθέτουμε στη ζημιά που έχει υποστεί το αντικείμενο. Από εκεί και πέρα, η μηχανή θα καθορίσει εσωτερικά τι πρέπει να γίνει (π.χ. ποια μέθοδος πρέπει να κληθεί, βλ. callbacks παρακάτω), ανάλογα το ποσό ζημιάς που έχει υποστεί το αντικείμενο. Αν όμως εμείς ορίσουμε μία ιδιότητα, έστω health, π.χ. στο αντικείμενο μιας κλάσης Player, αυτή δεν σημαίνει τίποτα για τη μηχανή. Οποιοσδήποτε έλεγχος ή ενημέρωση σχετικά με αυτή την ιδιότητα θα πρέπει να γίνεται μέσω TorqueScript και να καλείται ρητά όποτε χρειάζεται.

Υπάρχει τέλος μία ειδική κατηγορία μεθόδων, οι οποίες καλούνται αυτόματα από τη μηχανή όταν παρουσιαστεί ένα γεγονός (event) που τις ενεργοποιεί. Οι μέθοδοι αυτές λέγονται *callback methods* και μπορεί να έχουν οριστεί τόσο για κλάσεις όσο και για datablocks. Ισχύουν τα ίδια όπως και για όλες τις μεθόδους, η διαφορά είναι στον τρόπο κλήσης τους (μπορεί να τις καλέσει κανείς και άμεσα μέσω script, αλλά γενικά δεν έχει νόημα κάτι τέτοιο). Για παράδειγμα, στο datablock ShapeBaseData ορίζεται η callback μέθοδος onDamage(), η οποία θα κληθεί αυτόματα όταν ένα αντικείμενο κλάσης που χρησιμοποιεί αντικείμενο αυτού του datablock (ή κάποιου datablock που το κληρονομεί) δεχθεί ζημιά. Έτσι, αρκεί να ορίσουμε την callback μέθοδο onDamage() στον script κώδικα και να βάλουμε στο σώμα της τις λειτουργίες που θέλουμε να εκτελούνται όταν αυτή καλείται. Μερικές callback μέθοδοι είναι απαραίτητο να έχουν οριστεί στο script, και αν η μηχανή ψάξει για αυτές για να τις καλέσει, αλλά δεν τις βρει, δεν θα λειτουργήσει σωστά (παράδειγμα αποτελούν οι callback μέθοδοι της κλάσης GameConnection που χρησιμοποιούνται κατά την εγκατάσταση μίας σύνδεσης). Οι callback μέθοδοι συχνά έχουν όνομα που ξεκινά από "on…", αλλά αυτό δεν ισχύει πάντα.

Αν και σχετικά σπάνιες, υπάρχουν και συναρτήσεις κονσόλας που καλούνται αυτόματα από τη μηχανή του Torque όταν συμβεί κάποιο γεγονός, και θα τις λέμε *callback functions*. Ένα παράδειγμα είναι η resetCanvas() που καλείται όταν αλλάζει η ανάλυση, και απαιτείται να είναι ορισμένη στον script κώδικα.

Όπως είδαμε, κάθε κλάση, αντικείμενο κλάσης, datablock ή αντικείμενο datablock, ορίζει ταυτόχρονα και ένα χώρο ονομάτων, στον οποίο ανήκουν όλες οι ιδιότητες και μέθοδοί της. Η TorqueScript, σε αναλογία με τη C++, επιτρέπει να υπάρχουν και άλλοι, "αυθαίρετοι" *χώροι ονομάτων (namespaces)* που δεν σχετίζονται με καμία κλάση, datablock ή αντικείμενό τους.

Χρησιμεύουν στην ενθυλάκωση μεταβλητών / συναρτήσεων, με στόχο την ομαδοποίηση κοινής λειτουργικότητας, την αποφυγή συγκρούσεων ονομάτων με άλλες μεταβλητές / συναρτήσεις, την αποφυγή του λεγόμενου global namespace pollution.

Αυτοί οι χώροι ονομάτων δεν δηλώνονται πουθενά, αλλά όταν μια μεταβλητή ή μια συνάρτηση ορίζονται κάπως έτσι:

```
// Παγκόσμια μεταβλητή με όνομα "globalVariableName", ορισμένη στο χώρο
```

```
// ονομάτων "namespaceName1".
$ namespaceName1 :: globalVariableName = value ;
// Συνάρτηση με όνομα "functionName", ορισμένη στο χώρο ονομάτων
// "namespaceName2".
function namespaceName2 :: functionName (arg1, arg2, …) { … } ;
```

θεωρούνται ορισμένες μέσα στον αντίστοιχο χώρο ονομάτων. Η αναφορά σε αυτές πρέπει να γίνεται πάντα με το όνομα του χώρου ονομάτων να προηγείται του ονόματός τους (π.χ. η μεταβλητή globalVariableName δεν είναι η ίδια με την $NamespaceName1::globalVariableName). Να σημειωθεί ότι απουσιάζει το %this (the "this" reference), καθώς δεν πρόκειται για μεθόδους κλάσεων / datablocks ή αντικειμένων.

Ένας μηχανισμός παρόμοιος με τους χώρους ονομάτων είναι τα *packages*. Ένα package ορίζεται ως εξής:

```
package packageName {
        // ιδιότητες και συναρτήσεις ορισμένες στο package
};
```

και το διαχειριζόμαστε με τις συναρτήσεις κονσόλας activatePackage(packageName) και deactivatePackage(packageName). Καλώντας την πρώτη, επανα-ορίζονται όλες οι μεταβλητές και συναρτήσεις που περιέχει το package. Έστω ότι έχουμε ορίσει τη συνάρτηση aFunction() κάπου στο script. Αν μετά ενεργοποιήσουμε ένα package στο οποίο έχουμε ορίσει μια συνάρτηση επίσης με το όνομα aFunction(), η συνάρτηση του package θα υπερκαλύψει την προηγούμενη, και πλέον όταν καλούμε την aFunction() θα χρησιμοποιείται ο ορισμός της που ισχύει στο package. Απενεργοποιώντας το package θα επανέλθει σε ισχύ ο προηγούμενος ορισμός της συνάρτησης. Περισσότερα από ένα packages μπορούν να είναι ενεργά ταυτόχρονα, οπότε οι ορισμοί του τελευταίου υπερκαλύπτουν όλους τους προηγούμενους και έχουμε ουσιαστικά μία στοίβα (stack) από packages. Η activatePackage() αντιστοιχεί σε push και η deactivatePackage() σε pop.

Εκτός από την ιεραρχία κλάσεων, το Torque προσφέρει έτοιμες συναρτήσεις για το χειρισμό εισόδου – εξόδου σε αρχεία, για συνηθισμένες μαθηματικές πράξεις κλπ., ουσιαστικά σαν μία βιβλιοθήκη συναρτήσεων κατ' αναλογία με άλλες γλώσσες. Οι συναρτήσεις αυτές αναφέρονται στο Torque σαν *συναρτήσεις κονσόλας (console functions)*.

Ορισμός νέων συναρτήσεων

Ο χρήστης μπορεί να ορίσει επιπλέον συναρτήσεις (functions), και από τη στιγμή που αυτές οριστούν, χρησιμοποιούνται όπως ακριβώς και οι συναρτήσεις κονσόλας. Για παράδειγμα:

```
function aFunction (%arg1, %arg2, …)
{
        // σώμα συνάρτησης
}
```

Να σημειώσουμε ότι το Torque χρησιμοποιεί για την TorqueScript είτε έναν μεταγλωττιστή (*compiler*) είτε έναν διερμηνέα (*interpreter*), ανάλογα. Αυτό σημαίνει ότι εκτός της συγγραφής κώδικα σε αρχεία και της μεταγλώττισης του κώδικα αυτών

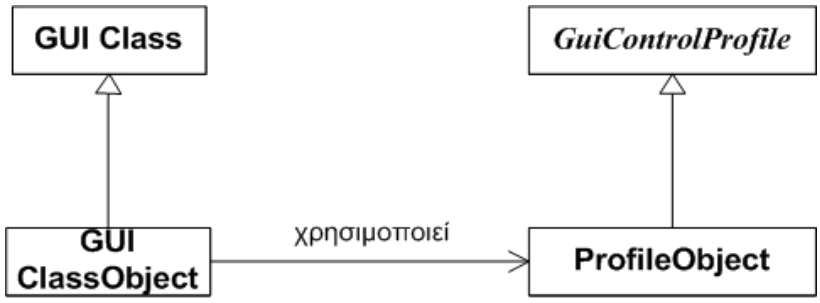
των αρχείων πριν την εκτέλεση του Torque, μπορούμε να δίνουμε εντολές και κατά την εκτέλεσή του (run-time) μέσω κονσόλας (console).

4. *Torque και GUIs*

Στην ιεραρχία των κλάσεων παρατηρούμε ότι το όνομα κάποιων κλάσεων αρχίζει από "Gui…". Οι κλάσεις αυτές υλοποιούν τις ενσωματωμένες λειτουργίες που προσφέρει η TorqueScript για τη δημιουργία γραφικών διεπαφών με το χρήστη (Graphical User Interfaces – GUIs). Η κλάση *GuiControl* είναι η ρίζα για το υποδένδρο των GUI κλάσεων, και όλες αυτές οι κλάσεις την κληρονομούν.

Η μηχανή αναλαμβάνει να αντιστοιχίσει τις σχετικές με GUIs κλήσεις που γίνονται σε επίπεδο script, με τις κατάλληλες κλήσεις του windowing system (συνήθως τμήμα του λειτουργικού συστήματος) που χρησιμοποιείται. Έτσι επιτυγχάνεται *ανεξαρτησία πλατφόρμας (platform independence)* όσον αφορά το look-and-feel της εφαρμογής (αναλογία με cross-platform γλώσσες όπως η Java). Από την άλλη, όπως πάντα σε τέτοιες περιπτώσεις όπου μεταβαίνουμε σε ένα υψηλότερο αφαιρετικό επίπεδο, αναγκαστικά χάνει κανείς σε δυνατότητες και ευελιξία που θα είχε δουλεύοντας απ' ευθείας πάνω στο σύστημα που χρησιμοποιεί.

Σε αντιστοιχία με όσα είδαμε για το διαχωρισμό σε κλάσεις και datablocks, στην περίπτωση των GUIs έχουμε (Gui) κλάσεις και ένα *profile* (στη θέση των datablocks). Αντίθετα με την αντιστοιχία κλάσης – datablock που είδαμε, δεν έχει κάθε (Gui) κλάση το δικό της αντίστοιχο profile, αλλά *υπάρχει ένα και μόνο profile, το GuiControlProfile*, από το οποίο δημιουργούνται όλα τα αντικείμενα profile. Επίσης, *απαιτείται* για κάθε αντικείμενο (Gui) κλάσης να έχουμε προηγουμένως δημιουργήσει και ένα αντικείμενο profile, ώστε να συσχετίσουμε το αντικείμενο της (Gui) κλάσης με ένα αντικείμενο του profile. Σχηματικά:



Σχ.8 Σχέση GUI κλάσεων / Profile και αντικειμένων αυτών

Πριν δημιουργήσουμε οποιοδήποτε άλλο αντικείμενο profile, πρέπει πρώτα να έχουμε δημιουργήσει ένα αντικείμενο profile με το όνομα GuiDefaultProfile. Κατά τη δημιουργία αυτού, θέτουμε όλες τις ιδιότητες του profile GuiControlProfile σε κάποια default τιμή. Όταν ύστερα δημιουργήσουμε κάποιο άλλο αντικείμενο profile, υπερκαλύπτουμε (override) μόνο τις ιδιότητες που θέλουμε και όλες οι άλλες θα έχουν αυτόματα την default τιμή (όπως αυτή καθορίστηκε κατά τη δημιουργία του GuiControlProfile), χωρίς να χρειαστεί να τις ορίσουμε ξανά.

Για τη λογική πίσω από το διαχωρισμό σε (Gui) κλάσεις – profile, ισχύουν αυτά που αναφέραμε και για τον διαχωρισμό σε κλάσεις – datablocks (διαχωρισμός μεθόδων – ιδιοτήτων όπου αυτό είναι δυνατό, οι λίγες ιδιότητες που εμφανίζονται στις -Gui- κλάσεις έχει νόημα να ορίζονται για κάθε αντικείμενο -Gui- κλάσης ξεχωριστά -π.χ. profile, position, extent- και δεν θα είχε νόημα να τις χρησιμοποιήσουμε ομαδικά για πολλά αντικείμενα -Gui- κλάσεων, όπως μπορούμε να κάνουμε με τις ιδιότητες που ορίζονται στο profile, κλπ.).

Ιδιαίτερες ιδιότητες του profile GuiControlProfile

- *canKeyFocus* : αν είναι αληθής, το αντικείμενο (Gui) κλάσης που χρησιμοποιεί το αντικείμενο profile έχει τη δυνατότητα να δέχεται την είσοδο του χρήστη από το πληκτρολόγιο (δηλ. μπορεί να πάρει το keyboard focus. Αν π.χ. πρόκειται για αντικείμενο όπου εισάγουμε κείμενο, ο cursor μπορεί να "πάει" εκεί, και η είσοδος του χρήστη από το πληκτρολόγιο μπορεί να κατευθύνεται στο συγκεκριμένο αντικείμενο). Αν είναι ψευδής, το αντικείμενο δεν θα μπορεί να πάρει είσοδο από το πληκτρολόγιο. Από όλα τα αντικείμενα στην οθόνη που μπορούν να δεχθούν είσοδο από το πληκτρολόγιο, επιλέγουμε αυτό που δέχεται την τρέχουσα είσοδο κάνοντας κλικ πάνω του με το ποντίκι. Αν αυτό δεν είναι δυνατό (π.χ. in-game όταν το ποντίκι έχει κλειδωθεί στο κέντρο της οθόνης, ή για άλλους λόγους), μπορούμε να επιλέξουμε το αντικείμενο που θα δεχθεί την είσοδο του πληκτρολογίου μέσω script, χρησιμοποιώντας τη μέθοδο *makeFirstResponder()* της κλάσης GuiControl (και η οποία έχει νόημα μόνο όταν καλείται για αντικείμενα (Gui) κλάσεων στο profile τον οποίων έχουμε θέσει *canKeyFocus = true*).
- *modal* : αν είναι αληθής, το αντικείμενο (Gui) κλάσης που χρησιμοποιεί το αντικείμενο profile συμπεριφέρεται σαν modal, δηλ. απορροφά όλη την είσοδο (input) του χρήστη, είτε έχει τρόπο να την χειριστεί είτε όχι. Αν είναι ψευδής, το αντικείμενο συμπεριφέρεται σαν modeless, δηλ. θα αφήσει και άλλα αντικείμενα (Gui) κλάσεων να πάρουν την είσοδο του χρήστη, εάν αυτό δεν έχει κάποιο τρόπο να την χειριστεί. Παράδειγμα modal GUI θα ήταν ένα αντικείμενο που όσο είναι ανοιχτό δεν επιτρέπει στον χρήστη να μεταβεί σε άλλο αντικείμενο, και κλικ του ποντικιού εκτός της περιοχής του αντικειμένου αγνοούνται. Παράδειγμα modeless GUI θα ήταν ένα αντικείμενο (έστω ημιδιαφανές) όπου κλικ του ποντικιού σε "άδεια" περιοχή του θα μεταβιβαζόταν σε όποιο αντικείμενο βρισκόταν "από κάτω", π.χ. ο χρήστης θα μπορούσε να πατήσει κουμπιά των αντικειμένων που βρίσκονται "κάτω" από ένα modeless GUI.
- *opaque* : αν είναι αληθής, το αντικείμενο (Gui) κλάσης που χρησιμοποιεί το αντικείμενο profile είναι αδιαφανές, αλλιώς είναι διαφανές (transparent).

Είναι συνηθισμένο στο Torque να μιλάμε για *controls*, όταν θέλουμε να αναφερθούμε γενικά σε αντικείμενα των Gui κλάσεων. Ένα control που περιέχει άλλα controls είναι *container* για αυτά. *Για controls που ορίζονται μέσα στο ίδιο container και καταλαμβάνουν την ίδια περιοχή έχει σημασία η σειρά δημιουργίας τους* (κάθε control θα καλύπτει τα προηγούμενά του).

Τα GUIs βρίσκονται γενικά στην πλευρά του πελάτη (client). Όταν ο πελάτης μιας εφαρμογής Torque εκτελεστεί, πρέπει κατά τις αρχικοποιήσεις του να δημιουργήσει ένα αντικείμενο της κλάσης *GuiCanvas*. Αυτό δεν γίνεται με το new όπως για τα άλλα αντικείμενα κλάσεων, αλλά καλώντας μία από τις συναρτήσεις κονσόλας *createCanvas()* ή createEffectCanvas(). Αφού γίνει αυτό, το αντικείμενο που κατασκεύασε εσωτερικά η μηχανή είναι διαθέσιμο σε επίπεδο script με το όνομα *Canvas*. Ένας πελάτης πρέπει να έχει *ένα και μόνο ένα* αντικείμενο της κλάσης GuiCanvas, *μέσα στο οποίο τοποθετούνται οποιαδήποτε άλλα αντικείμενα GUI κλάσεων* δημιουργήσει ο πελάτης. Συνεπώς, το αντικείμενο της κλάσης GuiCanvas πρέπει να δημιουργηθεί πριν από οποιοδήποτε άλλο αντικείμενο GUI κλάσεων. Η τοποθέτηση άλλων αντικειμένων GUI κλάσεων στο αντικείμενο Canvas γίνεται με τις μεθόδους *setContent()* (όταν (επανα)θέτουμε συνολικά τα περιεχόμενα του Canvas) και *pushDialog()* (όταν κάνουμε push ένα GUI "πάνω" από τα ήδη υπάρχοντα περιεχόμενα του Canvas) της κλάσης GuiCanvas.

5. ***Οργάνωση της εφαρμογής σε αρχεία***

Κατ' αρχήν, να σημειώσουμε ότι το Torque έρχεται με μερικά παραδείγματα εφαρμογών γραμμένων σε αυτό, που ακολουθούν λίγο έως πολύ μία συγκεκριμένη οργάνωση των αρχείων τους. Η εργασία αυτή δεν χρησιμοποιεί κανένα από αυτά τα παραδείγματα, αλλά έχει γραφτεί ξεκινώντας από το μηδέν, όσον αφορά το script πάντα. Βέβαια, τα παραδείγματα αυτά χρησίμευσαν σαν οδηγός, και σε μεγάλο βαθμό ακολουθείται παρόμοια οργάνωση με αυτά. Η επιλογή αυτή σημαίνει από τη μία ότι γενικά έχουμε μόνο την απαραίτητη λειτουργικότητα (και πιο συνεκτική, αφού η πλειοψηφία των λειτουργιών προστέθηκε σταδιακά και μόνο όταν αυτή χρειάστηκε), αλλά από την άλλη δεν έχουμε ευκολίες όπως οι διάφοροι editors που υλοποιούνται σε script στα παραδείγματα.

Ο κατάλογος (directory) στον οποίο τοποθετείται το εκτελέσιμο (.exe), μαζί με τις δυναμικές βιβλιοθήκες (.dll) για τα Windows, είναι εξ' ορισμού ο κατάλογος-ρίζα (root-directory) της εφαρμογής. Εδώ τον ονομάζουμε "myTorqueDemo". Οι διαδρομές (paths) όλων των αρχείων που δίνονται σε αυτή την εργασία είναι σχετικές (relative) και ξεκινάνε πάντα από τον κατάλογο-ρίζα, έτσι αυτός θα παραλείπεται. Για παράδειγμα, η διαδρομή /control/main.cs δηλώνει το αρχείο main.cs που βρίσκεται στον κατάλογο control, ο οποίος με τη σειρά του βρίσκεται στον κατάλογο myTorqueDemo.

Όταν το εκτελέσιμο εκτελεστεί, ψάχνει για ένα αρχείο με όνομα main.cs, που θα πρέπει να βρίσκεται στον ίδιο κατάλογο με το εκτελέσιμο, δηλ. στον κατάλογο-ρίζα. Συνεπώς, το σημείο εισόδου (entry point) της εφαρμογής είναι αυτό το αρχείο main.cs. Εκεί επεξεργαζόμαστε τα ορίσματα γραμμής (command line arguments). Στη συνέχεια καθορίζουμε τους καταλόγους που θέλουμε να είναι ορατοί (και άρα προσπελάσιμοι) από μηχανή, με τη συνάρτηση κονσόλας setModPaths(). Εδώ μας αρκεί ο κατάλογος control, που ονομάζεται έτσι κατά παράδοση στο Torque (ο κατάλογος common που χρησιμοποιούν τα παραδείγματα δεν υπάρχει. Όλη η απαραίτητη λειτουργικότητα που προσφέρουν τα αρχεία του είναι ενσωματωμένη στα αρχεία του καταλόγου control). Οι κατάλογοι αυτοί συνηθίζεται να λέγονται mod(ule)s. Για κάθε mod εκτελούμε (με τη συνάρτηση κονσόλας exec()) ένα αρχείο, που τυπικά ονομάζεται και αυτό main.cs, και φορτώνει κάποιους αρχικούς ορισμούς, παραδοσιακά μία συνάρτηση onStart() την οποία στη συνέχεια καλούμε ώστε το κάθε mod να αναλάβει πλέον ότι δραστηριότητες πρέπει.

Στον κατάλογο control υπάρχουν τρεις κατάλογοι: client, server, data. Η /control/main.cs με τη σειρά της εκτελεί τα αρχεία /control/client/init.cs και /control/server/init.cs, που περιέχουν κάποιους αρχικούς ορισμούς για τον πελάτη και τον εξυπηρετητή αντίστοιχα, και καλεί τις συναρτήσεις αρχικοποίησης του καθενός, που τυπικά ονομάζονται initClient() και initServer().

Αυτές εκτελούν τα αρχεία που χρειάζεται ο πελάτης ή ο εξυπηρετητής αντίστοιχα, φορτώνοντας στη μνήμη τους ορισμούς μεταβλητών, συναρτήσεων, αντικειμένων, κλπ. που θέλουμε να είναι διαθέσιμοι κατά τη διάρκεια της εκτέλεσης της εφαρμογής μας. Τώρα πλέον λέμε ότι έχει φορτωθεί τόσο ο πελάτης όσο και ο εξυπηρετητής, και ο καθένας αναλαμβάνει τις προγραμματισμένες του δραστηριότητες. Εδώ, ο εξυπηρετητής βασικά περιμένει μέχρι να συνδεθεί ο πρώτος του πελάτης, ενώ ο πελάτης ανοίγει ένα παράθυρο και προβάλει το αρχικό μενού επιλογών στον χρήστη.

Όπως προκύπτει από τα παραπάνω, ο κατάλογος /control/data/ δε φαίνεται να χρησιμοποιήθηκε πουθενά. Αυτός ο κατάλογος γενικά δεν περιέχει κώδικα, αλλά κυρίως πόρους (resources) οπτικοακουστικής φύσης (3D μοντέλα, 2D εικόνες,

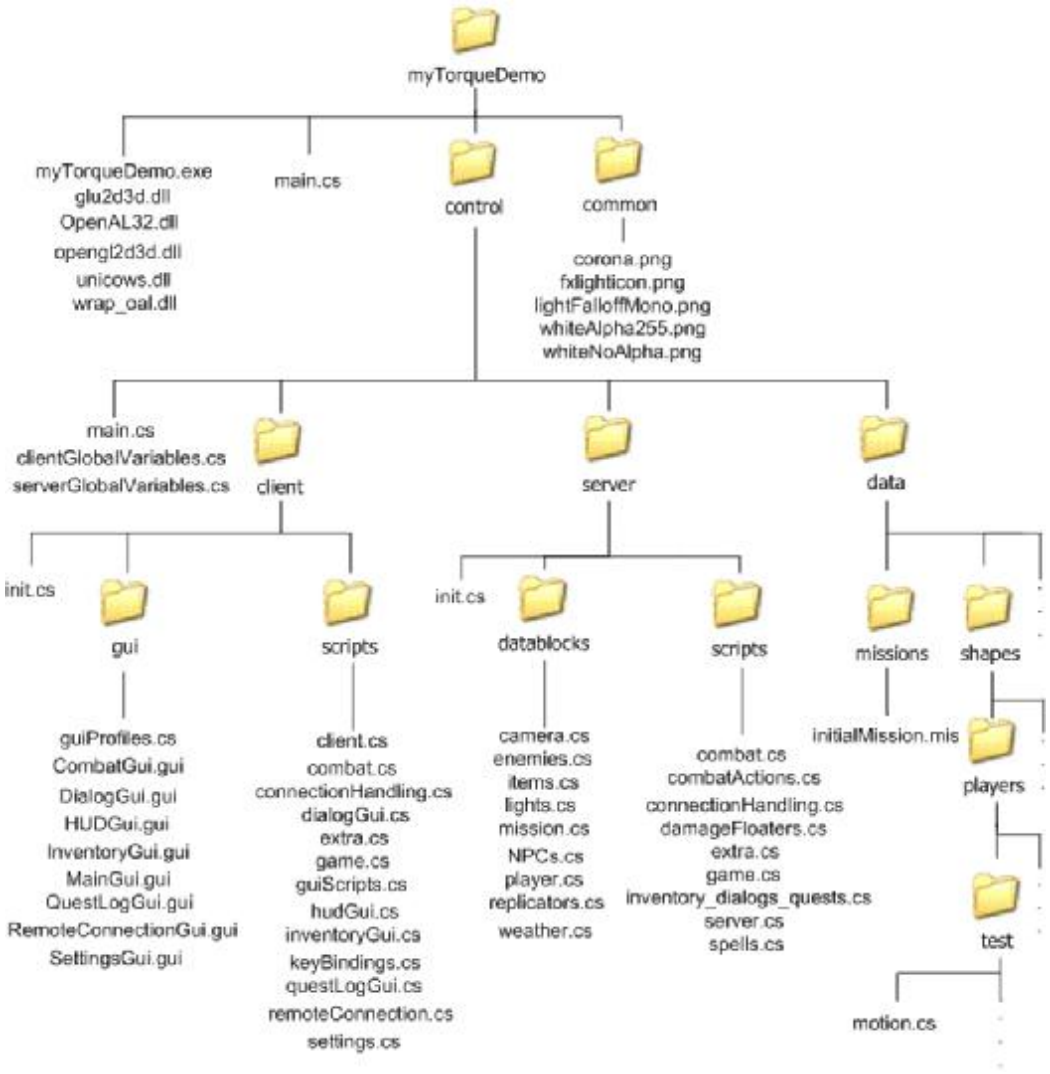
αρχεία ήχου, κλπ.), που χρησιμοποιούνται για την αναπαράσταση του κόσμου. Ο κώδικας του πελάτη και του εξυπηρετητή είναι υπεύθυνος για το χειρισμό αυτών των πόρων. Για παράδειγμα, ο πελάτης επιλέγει ένα μουσικό κομμάτι που επαναλαμβάνεται όσο δείχνει το κεντρικό μενού, στον εξυπηρετητή όταν δημιουργείται ένα αντικείμενο του datablock PlayerData επιλέγεται κάποιο 3D μοντέλο που θα αναπαριστά το αντικείμενο στον κόσμο, κλπ..

Ο κατάλογος /control/data/ πρέπει να βρίσκεται και στον πελάτη και στον εξυπηρετητή σε περίπτωση που αυτοί δεν τρέχουν στο ίδιο μηχάνημα, γιατί κατά κανόνα περιέχει πόρους που πρέπει να είναι προσβάσιμοι και από τις δύο πλευρές. Για παράδειγμα, ο εξυπηρετητής είναι αυτός που καθορίζει ποιο 3D μοντέλο θα χρησιμοποιηθεί από ποιο αντικείμενο στον κόσμο, αλλά δεν στέλνει όλο το αρχείο που ορίζει το 3D μοντέλο σε κάθε πελάτη κάθε φορά. Αντίθετα, τους λέει ποιο είναι το path στο οποίο πρέπει να αναζητούσουν το κατάλληλο αρχείο.

Συνηθισμένοι τύποι αρχείων στο Torque

- *.cs* : αρχεία πηγαίου script κώδικα
- *.gui* : αρχεία πηγαίου script κώδικα όπως και τα .cs, το ειδικό extension υποδεικνύει ότι περιέχουν κώδικα σχετικό με τη δομή κάποιου GUI
- *.mis* : αρχεία πηγαίου script κώδικα όπως και τα .cs, το ειδικό extension υποδεικνύει ότι περιέχουν κώδικα σχετικό με τη δομή κάποιας αποστολής και την δημιουργία / τοποθέτηση αντικειμένων σε αυτή
- *.dso* : binary αρχεία που προκύπτουν από το compilation αρχείων πηγαίου script κώδικα
- *.jpg / .png* : γνωστά format εικόνας που υποστηρίζονται
- *.wav / .ogg* : γνωστά format ήχου που υποστηρίζονται
- *.dts* : το format που χρησιμοποιεί το Torque για τα 3D μοντέλα
- *.dsq* : αρχεία που περιέχουν τις ακολουθίες κίνησης (animation sequences) για κάποιο μοντέλο, όταν αυτές δεν είναι ενσωματωμένες στο ίδιο το .dts (παρέχουν μεγαλύτερη ευελιξία και δυνατότητα για αυτό που λέγεται merge animations, δηλ. να μπορούν περισσότερα από ένα animation να εκτελούνται ταυτόχρονα)
- *.dif* : το format που χρησιμοποιεί το Torque για τους εσωτερικούς χώρους (κτήρια κλπ.)

Ακολουθεί ένα διάγραμμα της οργάνωσης της εφαρμογής σε αρχεία:
(παρατηρούμε ότι αν και δεν υπάρχει το common σαν mod, υπάρχει ένας κατάλογος με αυτό το όνομα. Περιέχει πέντε αρχεία .jpg, τα οποία η μηχανή φαίνεται να αναζητάει στον κατάλογο με όνομα common. Π.χ. το lightFalloffMono.png χρησιμοποιείται για τα αντικείμενα της κλάσης fxLights, για να εξασθενεί το φως τους στην περιφέρεια με ομαλό τρόπο. Αν η μηχανή δεν το βρει, τότε αυτά τα φώτα θα καλύπτουν μία τετραγωνική περιοχή με την ίδια ένταση σε κάθε σημείο της. Δηλ. η εικόνα αυτή δρα σαν μάσκα)



Σχ.9 Οργάνωση της εφαρμογής σε αρχεία

Σχετικά αρχεία:
/main.cs
/control/main.cs
/control/server/init.cs
/control/client/init.cs

6. ***Θέματα δικτύωσης***

Ένα βασικό χαρακτηριστικό του Torque είναι ότι σχεδιάστηκε εξ' αρχής έτσι ώστε να διευκολύνει την ανάπτυξη δικτυακών εφαρμογών. Χρησιμοποιείται η *αρχιτεκτονική πελάτη-εξυπηρετητή (client-server)*. Κάθε εφαρμογή υλοποιημένη στο Torque χωρίζεται σε δύο μέρη: τον πελάτη (client) και τον εξυπηρετητή (server), και αυτό ισχύει ακόμα και όταν η εφαρμογή εκτελείται σε έναν μόνο υπολογιστή (λέμε τότε ότι έχουμε *τοπική σύνδεση -local connection-*, ενώ σε αντίθετη περίπτωση έχουμε *απομακρυσμένη σύνδεση -remote connection-*).

Ο κώδικας στο κάθε τμήμα (πελάτη, εξυπηρετητή) είναι ανεξάρτητος από τον κώδικα στο άλλο τμήμα. Ο εξυπηρετητής έχει της δικές του μεταβλητές, συναρτήσεις και αντικείμενα, και αντίστοιχα για τον πελάτη. Ο εξυπηρετητής δεν μπορεί να προσπελάσει άμεσα τις μεταβλητές ή τα αντικείμενα του πελάτη ούτε να καλέσει τις συναρτήσεις που ορίζονται στον κώδικα του πελάτη, και αντίστοιχα για τον πελάτη. Υπάρχει μια *εξαίρεση* σε αυτό τον κανόνα: όταν και ο πελάτης και ο εξυπηρετητής τρέχουν στο ίδιο μηχάνημα, είναι δυνατόν να έχουν πρόσβαση ο ένας στον κώδικα του άλλου. Φυσικά κώδικας που στηρίζεται στην παραπάνω εξαίρεση πρέπει να αποφεύγεται, καθώς δεν πρόκειται να λειτουργήσει σε δικτυακό περιβάλλον.

Στον πελάτη δεν μπορούμε να δημιουργήσουμε αντικείμενα από ***datablocks*** (συνεπώς ούτε και αντικείμενα κλάσεων που σχετίζονται με κάποιο datablock). Όπως έχουμε αναφέρει, τα datablocks περιέχουν ιδιότητες που επηρεάζουν άμεσα τη συμπεριφορά των αντικειμένων του κόσμου, και η δημιουργία αντικειμένων τους μόνο στον εξυπηρετητή αποτρέπει κακόβουλους πελάτες από την τροποποίησή τους προς όφελός τους. Σε αντίθετη περίπτωση, θα μπορούσε για παράδειγμα ένας πελάτης να φτιάξει ένα αντικείμενο του datablock PlayerData ορίζοντας στην ιδιότητα maxDamage όποια τιμή ήθελε. Γενικά, ο διαχωρισμός σε πελάτη – εξυπηρετητή βοηθά στην *απομόνωση κρίσιμων για την ομαλή λειτουργία του κόσμου λειτουργιών στην πλευρά του εξυπηρετητή*, ο οποίος αναλαμβάνει να παρέχει σε όλους τους πελάτες που είναι συνδεδεμένοι μαζί του την ίδια, κοινή εικόνα του κόσμου και να διαχειρίζεται την αλληλεπίδρασή τους με αυτόν και μεταξύ τους με συνεπή τρόπο.

Το πρώτο πράγμα που πρέπει να κάνει ένας πελάτης για να εισέλθει στον κόσμο (και να μπορέσει να αλληλεπιδράσει με οτιδήποτε υπάρχει σε αυτόν) είναι να συνδεθεί με τον εξυπηρετητή. Βασική κλάση εδώ είναι η *GameConnection*, που ορίζει μεθόδους και callbacks για την εγκατάσταση και το χειρισμό μιας τέτοιας σύνδεσης. Για αυτές συνηθίζεται η μεταβλητή %this (the "this" reference) να αντικαθίσταται με την μεταβλητή %client (ή %clientGameConnection), καθαρά για λόγους αναγνωσιμότητας του κώδικα.

Για να συνδεθεί ένας πελάτης με έναν εξυπηρετητή, δημιουργεί από την πλευρά του ένα αντικείμενο της κλάσης GameConnection, με τις μεθόδους του οποίου μπορεί να συνδεθεί στον εξυπηρετητή.

Στην πλευρά του εξυπηρετητή, η μηχανή του Torque διατηρεί εσωτερικά μία λίστα με όλους τους συνδεδεμένους πελάτες. Η λίστα αυτή είναι προσβάσιμη σε επίπεδο script μέσω του αντικειμένου *ClientGroup* (που είναι αντικείμενο της κλάσης SimGroup ή SimSet -ουσιαστικά δεν έχουν καμία διαφορά-). Το ClientGroup υπάρχει χωρίς να χρειάζεται να το δημιουργήσει ο χρήστης. Όταν ένας πελάτης συνδεθεί σε έναν εξυπηρετητή, η μηχανή προσθέτει αυτόματα ένα αντικείμενο της κλάσης GameConnection στο ClientGroup του εξυπηρετητή. Δηλαδή *δεν* δημιουργούμε ρητά αντικείμενα GameConnection για τις εισερχόμενες συνδέσεις στον εξυπηρετητή.

Η διπλή φύση (client / server) των αντικειμένων της κλάσης GameConnection

Πλευρά του πελάτη:

client-side GameConnection **ό** ΑΠΟ πελάτη ΣΕ εξυπηρετητή **ό** *serverGameConnection* (με την έννοια ότι συνδέει με τον εξυπηρετητή)

Πλευρά του εξυπηρετητή:

server-side GameConnection **ό** ΑΠΟ εξυπηρετητή ΣΕ πελάτη **ό** *clientGameConnection* (με την έννοια ότι συνδέει με τον πελάτη)

Κάποιες μέθοδοι της κλάσης GameConnection μπορεί να είναι διαθέσιμες και στην πλευρά του πελάτη και στην πλευρά του εξυπηρετητή (π.χ. setFirstPerson()), κάποιες μόνο στον πελάτη (π.χ. setBlackOut()), και κάποιες μόνο στον εξυπηρετητή (π.χ. setControlObject()). Μία μέθοδος έχει νόημα να είναι διαθέσιμη και (ή μόνο) στον πελάτη εάν δεν επηρεάζει ούτε τον κόσμο, ούτε το πώς βλέπουν οι άλλοι χρήστες τον συγκεκριμένο πελάτη. Σε αντίθετη περίπτωση, η μέθοδος είναι διαθέσιμη μόνο στην πλευρά του εξυπηρετητή και η κλήση της απ' ευθείας από κάποιον πελάτη δεν θα έχει κανένα αποτέλεσμα.

Από τη στιγμή που έχει εγκατασταθεί η σύνδεση μεταξύ πελάτη-εξυπηρετητή, ο ένας μπορεί να επικοινωνήσει απ' ευθείας με τον άλλο με το *μηχανισμό ανταλλαγής μηνυμάτων (messaging)* του Torque (πέρα δηλ. από τις προκαθορισμένες μεθόδους της κλάσης GameConnection ή άλλων έτοιμων κλάσεων). Ο μηχανισμός αυτός βασίζεται στις συναρτήσεις κονσόλας commandToServer() και commandToClient(), και λειτουργεί ως εξής:

Μήνυμα από πελάτη σε εξυπηρετητή

Στην πλευρά του πελάτη, στέλνουμε το μήνυμα στον -μοναδικό- εξυπηρετητή με τον οποίο είμαστε συνδεδεμένοι.

function commandToServer('***FunctionName***', arg1, arg2, …) { … };

Στην πλευρά του εξυπηρετητή, η λήψη του μηνύματος μεταφράζεται στην κλήση μιας συνάρτησης με κατάλληλα διαμορφωμένο όνομα.

function serverCmd***FunctionName***(arg1, arg2, …) { … };

Μήνυμα από εξυπηρετητή σε πελάτη

Στην πλευρά του εξυπηρετητή, επιλέγουμε έναν από τους συνδεδεμένους πελάτες και του στέλνουμε το μήνυμα.

function commandToClient(%clientGameConnection, '***FunctionName***', arg1, arg2, …) { … };

Στην πλευρά του πελάτη, η λήψη του μηνύματος μεταφράζεται στην κλήση μιας συνάρτησης με κατάλληλα διαμορφωμένο όνομα.

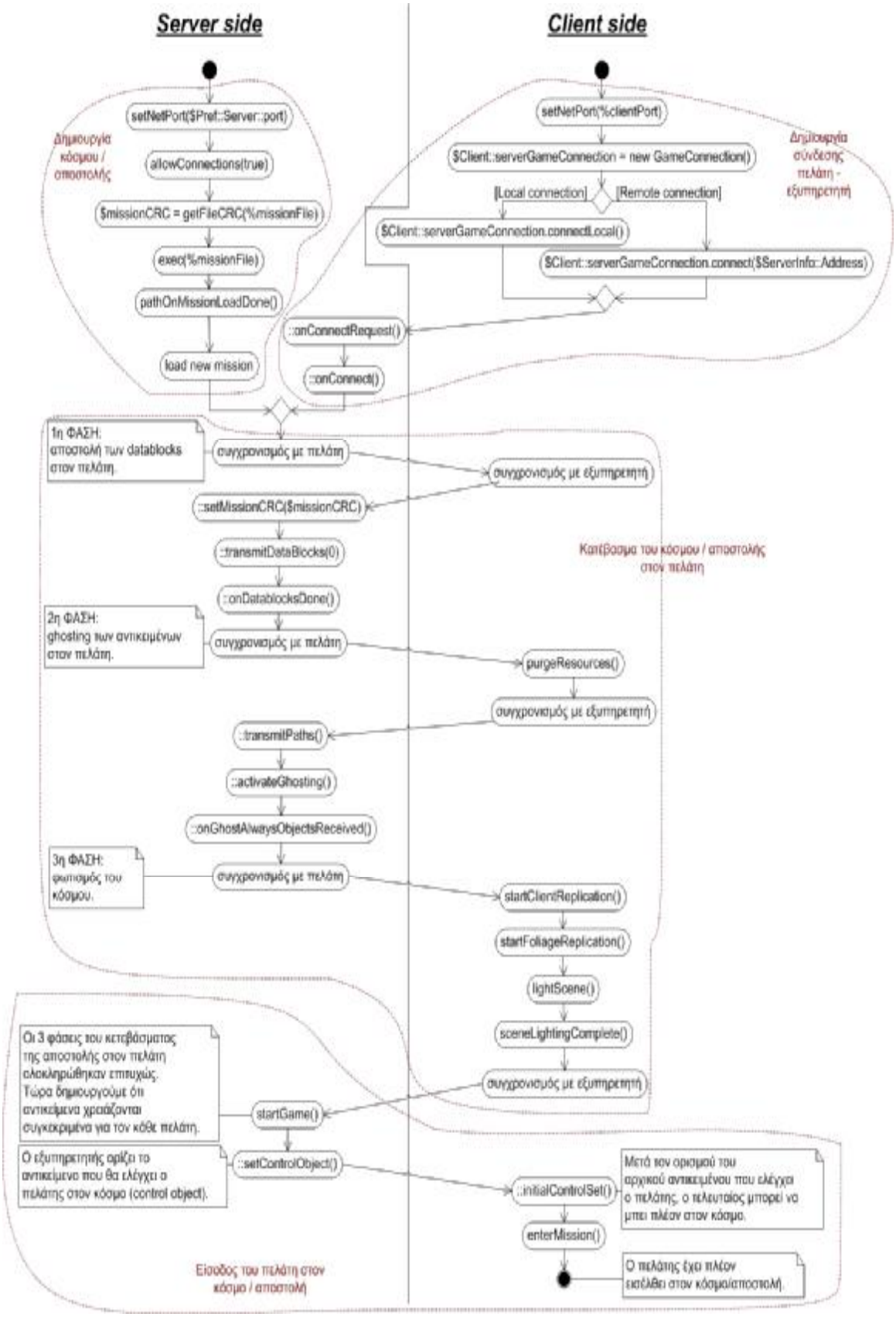
function clientCmd***FunctionName***(arg1, arg2, …) { … };

Το ενδεικτικό FunctionName πρέπει να είναι μέσα σε μονά εισαγωγικά ' ' (tagged string). Όπως φαίνεται, ουσιαστικά μπορούμε να καλέσουμε από την μία πλευρά συναρτήσεις που έχουν οριστεί στην άλλη πλευρά της σύνδεσης.

Ακολουθεί ένα διάγραμμα δραστηριότητας (UML activity diagram) με τα κύρια βήματα της διαδικασίας που εκτελείται πριν ένας πελάτης μπορέσει να εισέλθει στον κόσμο. Οι διακεκομμένες γραμμές στο διάγραμμα ομαδοποιούν λογικά τα βήματα (δεν είναι στοιχεία της UML).

Παρατηρήσεις σχετικά με το διάγραμμα:
Όλες οι μέθοδοι που αρχίζουν με "::" είναι μέθοδοι της κλάσης GameConnection. Όσες μέθοδοι επιπλέον έχουν όνομα που αρχίζει από "on…" είναι callback μέθοδοι της κλάσης GameConnection.
Η συνάρτηση sceneLightingComplete() είναι callback συνάρτηση.
Οι συναρτήσεις sceneLightingComplete(), startGame() και enterMission() είναι ενδεικτικές (οριζόμενες από τον χρήστη).
Όλες οι άλλες συναρτήσεις είναι συναρτήσεις κονσόλας.
Οι μεταβλητές που εμφανίζονται είναι ενδεικτικές.
Οι λέξεις "αποστολή" – "κόσμος" χρησιμοποιούνται εννοώντας το ίδιο πράγμα (mission).
Διευκρίνηση: η μεταβλητή $Client::serverGameConnection είναι μία global μεταβλητή του πελάτη, ορισμένη στον -αυθαίρετο- χώρο ονομάτων (namespace) "Client" (δεν έχει να κάνει με το χώρο ονομάτων καμίας κλάσης, datablock ή αντικειμένου).
Το διάγραμμα παρουσιάζει την ακολουθία δραστηριοτήτων σε περίπτωση που δεν προκύψουν ανώμαλες καταστάσεις.

Σχ.10 Διάγραμμα δραστηριότητας για τη διαδικασία σύνδεσης πελάτη - εξυπηρετητή


Server side
Client side
setNetPort($Pref::Server::port)
allowConnections(true)
$missionCRC = getFileCRC(%missionFile)
exec(%missionFile)
pathOnMissionLoadDone()
load new mission
Δημιουργία κόσμου / αποστολής
setNetPort(%clientPort)
$Client::serverGameConnection = new GameConnection()
[Local connection]
[Remote connection]
$Client::serverGameConnection.connectLocal()
$Client::serverGameConnection.connect($ServerInfo::Address)
Δημιουργία σύνδεσης πελάτη - εξυπηρετητή
::onConnectRequest()
::onConnect()
1η ΦΑΣΗ: αποστολή των datablocks στον πελάτη.
συγχρονισμός με πελάτη
συγχρονισμός με εξυπηρετητή
::setMissionCRC($missionCRC)
::transmitDataBlocks(0)
::onDatablocksDone()
Κατέβασμα του κόσμου / αποστολής στον πελάτη
2η ΦΑΣΗ: ghosting των αντικειμένων στον πελάτη.
συγχρονισμός με πελάτη
purgeResources()
συγχρονισμός με εξυπηρετητή
::transmitPaths()
::activateGhosting()
::onGhostAlwaysObjectsReceived()
3η ΦΑΣΗ: φωτισμός του κόσμου.
συγχρονισμός με πελάτη
startClientReplication()
startFoliageReplication()
lightScene()
sceneLightingComplete()
συγχρονισμός με εξυπηρετητή
Οι 3 φάσεις του κατεβάσματος της αποστολής στον πελάτη ολοκληρώθηκαν επιτυχώς. Τώρα δημιουργούμε ότι αντικείμενα χρειάζονται συγκεκριμένα για τον κάθε πελάτη.
startGame()
Ο εξυπηρετητής ορίζει το αντικείμενο που θα ελέγχει ο πελάτης στον κόσμο (control object).
::setControlObject()
::initialControlSet()
Μετά τον ορισμού του αρχικού αντικειμένου που ελέγχει ο πελάτης, ο πελευταίος μπορεί να μπει πλέον στον κόσμο.
enterMission()
Ο πελάτης έχει πλέον εισέλθει στον κόσμο/αποστολή.
Είσοδος του πελάτη στον κόσμο / αποστολή

Δημιουργία κόσμου / αποστολής (πλευρά εξυπηρετητή)

1. *setNetPort($Pref::Server::port)* : καθορίζει το port στο οποίο ακούει ο εξυπηρετητής (πρέπει να κληθεί όταν αναμένονται απομακρυσμένες συνδέσεις στον εξυπηρετητή).
2. *allowConnections(true)* : επιτρέπει συνδέσεις στον εξυπηρετητή (πρέπει να κληθεί όταν αναμένονται απομακρυσμένες συνδέσεις στον εξυπηρετητή).
3. *$missionCRC = getFileCRC(%missionFile)* : Υπολογίζει το Cyclic Redundancy Check ενός αρχείου, εδώ του αρχείου όπου ορίζεται η αποστολή. Το CRC είναι γνωστό από τα δίκτυα υπολογιστών, όπου χρησιμοποιείται σαν τρόπος ανίχνευσης λαθών στα πακέτα που διακινούνται στο δίκτυο (το CRC ενός πακέτου που μεταδίδεται υπολογίζεται από τον αποστολέα και προστίθεται στο πακέτο. Ο αποδέκτης επανα-υπολογίζει το CRC στο πακέτο που πήρε και το συγκρίνει με την τιμή που είναι αποθηκευμένη στο πακέτο. Αν το πακέτο δεν άλλαξε / αλλοιώθηκε στη διαδρομή, τα CRC θα συμπίπτουν). Εδώ χρησιμοποιείται ως τρόπος ανίχνευσης αλλαγών στην αποστολή (όταν τα CRC δεν συμπίπτουν), ώστε να ξέρουμε ότι πρέπει να επανα-φωτιστεί η αποστολή.
4. *exec(%missionFile)* : δημιουργία της αποστολής και των αντικειμένων σε αυτή. Είναι τυπικό στην TorqueScript αυτά να ορίζονται σε ένα ξεχωριστό αρχείο το οποίο εκτελούμε με την exec(), αλλά αυτό δεν είναι δεσμευτικό.
5. *pathOnMissionLoadDone()* : φορτώνει την πληροφορία για το paths των εσωτερικών χώρων, οι οποίοι φορτώθηκαν προηγουμένως μαζί με τα υπόλοιπα αντικείμενα του κόσμου.
6. *load new mission* : για κάθε πελάτη που είναι ήδη συνδεδεμένος στον εξυπηρετητή όταν δημιουργείται / αλλάζει η αποστολή, κατέβασε τη νέα αποστολή στον πελάτη.

Δημιουργία σύνδεσης πελάτη – εξυπηρετητή

1. *setNetPort(%clientPort)* : καθορίζει το port στο οποίο ακούει ο πελάτης. Αν θέλουμε μόνο τοπικές συνδέσεις μπορούμε να το αφήσουμε 0, αλλιώς πρέπει να αντιστοιχεί σε ένα έγκυρο port. Πρέπει να κληθεί πριν την πρώτη κλήση της μεθόδου setContent() του αντικειμένου Canvas.
2. *$Client::serverGameConnection = new GameConnection()* : δημιουργία του αντικειμένου της κλάσης GameConnection για τη σύνδεση στον εξυπηρετητή.
3. *$Client::serverGameConnection.connectLocal()* : για τοπική σύνδεση
   ή
   $Client::serverGameConnection.connect($ServerInfo::Address) :
   για απομακρυσμένη σύνδεση
4. *::onConnectRequest()* : καλείται όταν ένας πελάτης προσπαθεί να συνδεθεί με τον εξυπηρετητή.
5. *::onConnect()* : καλείται όταν ο πελάτης συνδεθεί επιτυχώς.

Κατέβασμα του κόσμου / αποστολής στον πελάτη (mission downloading)

Εδώ παρατηρούμε ότι υπάρχουν 3 ξεχωριστές φάσεις (έχουν ονομαστεί με βάση την κύρια λειτουργία που επιτελείται σε αυτές). Σε κάθε μία απαιτείται ο συγχρονισμός μεταξύ πελάτη-εξυπηρετητή ώστε η κάθε φάση να ξεκινά όταν είναι και δύο έτοιμοι, και να μην προχωράμε στην επόμενη φάση αν δεν ολοκληρώσουν και οι δύο πλευρές επιτυχώς την τρέχουσα φάση. Ο συγχρονισμός επιτυγχάνεται με το μηχανισμό μηνυμάτων του Torque.

1η ΦΑΣΗ (αποστολή των αντικειμένων των datablocks στον πελάτη)

1. *::setMissionCRC($missionCRC)* : θέσε σαν CRC αυτό που υπολογίστηκε κατά την δημιουργία της αποστολής. Αν διαφέρει από το τρέχον CRC, η αποστολή έχει αλλάξει και συνεπώς πρέπει να επανα-φωτιστεί.
2. *::transmitDataBlocks(0)* : ξεκίνα την αποστολή των αντικειμένων των datablocks, που έχουν φορτωθεί στον εξυπηρετητή, στον πελάτη. Έτσι, ο πελάτης θα έχει όλες τις πληροφορίες που χρειάζεται για αυτά και θα μπορέσει να χειριστεί τα αντικείμενα κλάσεων που τα χρησιμοποιούν, τα οποία θα σταλούν στην επόμενη φάση (ο πελάτης δεν έχει πρόσβαση μέσω script στα αντικείμενα των datablocks που του έστειλε ο εξυπηρετητής, οπότε δεν μπορεί να τα αλλοιώσει).
3. *::onDatablocksDone()* : καλείται όταν όλα τα αντικείμενα των datablocks στάλθηκαν επιτυχώς στον πελάτη.

2η ΦΑΣΗ (ghosting των αντικειμένων κλάσεων στον πελάτη)

4. *purgeResources()* : ο πελάτης καταστρέφει όλα τα resources που ενδεχομένως είχε από προηγούμενες αποστολές.
5. *::transmitPaths()* : στέλνει την πληροφορία για το paths των εσωτερικών χώρων στον πελάτη.
6. *::activateGhosting()* : ξεκίνα την αποστολή των αντικειμένων κλάσεων, που έχουν φορτωθεί στον εξυπηρετητή, στον πελάτη (ο πελάτης δεν έχει πρόσβαση μέσω script στα αντικείμενα κλάσεων που του έστειλε ο εξυπηρετητής, οπότε δεν μπορεί να τα αλλοιώσει).
7. *::onGhostAlwaysObjectsReceived()* : καλείται όταν όλα τα αντικείμενα κλάσεων στάλθηκαν επιτυχώς στον πελάτη.

3η ΦΑΣΗ (φωτισμός της αποστολής / κόσμου. Γίνεται στην πλευρά του πελάτη.)

8. *startClientReplication()* : ενεργοποίηση του συστήματος παραγωγής πολλών όμοιων αντικειμένων από ένα πρωτότυπο, για αντικείμενα γενικά (βλ. κλάση fxShapeReplicator).
9. *startFoliageReplication()* : ενεργοποίηση του συστήματος παραγωγής πολλών όμοιων αντικειμένων από ένα πρωτότυπο, για αντικείμενα που αναπαριστούν βλάστηση (βλ. κλάση fxFoliageReplicator).
10. *lightScene()* : ξεκινά τον υπολογισμό του φωτισμού για την τρέχουσα αποστολή.
11. *sceneLightingComplete()* : καλείται όταν ο υπολογισμός του φωτισμού ολοκληρωθεί.

Είσοδος του πελάτη στον κόσμο / αποστολή

1. *startGame()* : τώρα που η αποστολή έχει κατέβει στον πελάτη, ο εξυπηρετητής κάνει τις απαραίτητες αρχικοποιήσεις για αυτόν τον συγκεκριμένο πελάτη (π.χ. δημιουργεί το αντικείμενο -avatar- που θα αναπαριστά τον πελάτη στον κόσμο, αρχικοποιεί τα στατιστικά του, κλπ.).
2. *::setControlObject()* : ο εξυπηρετητής θέτει το αντικείμενο που θα ελέγχει αρχικά ο πελάτης (control object), ώστε να μπορέσει ο τελευταίος να εισέλθει στον κόσμο.
3. *::initialControlSet()* : καλείται στην πλευρά του πελάτη όταν ο εξυπηρετητής θέσει το αντικείμενο που θα ελέγχει αρχικά ο πελάτης, ειδοποιώντας τον ότι είναι έτοιμος να εισέλθει στον κόσμο.
4. *enterMission()* : ο πελάτης κάνει τις απαραίτητες αρχικοποιήσεις από την πλευρά του, πριν εισέλθει στον κόσμο (π.χ. φορτώνει το HUD, τα διάφορα GUIs, κλπ.)

Σχετικά αρχεία:
/control/server/init.cs
/control/client/init.cs

/control/server/scripts/server.cs
/control/server/scripts/connectionHandling.cs
/control/client/scripts/connectionHandling.cs
/control/client/scripts/guiScripts.cs
    (function launchGame())
/control/client/scripts/remoteConnection.cs
    (function remoteConnection::join())
/control/client/scripts/game.cs
    (function enterMission())

7. ***Σύστημα αντικειμένων (Inventory)***

Σχετικά αρχεία:
/control/client/gui/InventoryGui.gui
/control/client/scripts/inventoryGui.cs
/control/server/scripts/game.cs
/control/server/scripts/inventory_dialogs_quests.cs
/control/server/datablocks/items.cs

Οι χρήστες, κατά την περιήγησή τους στον κόσμο βρίσκουν αντικείμενα με τα οποία μπορούν να αλληλεπιδράσουν. Τα αντικείμενα που μαζεύει ένας χρήστης πρέπει με κάποιο τρόπο να μπορεί και να τα χειριστεί. Εδώ έρχεται η έννοια του inventory, που αποτελεί έναν ενιαίο τρόπο παρουσίασης των αντικειμένων του χρήστη, δίνοντάς του τη δυνατότητα να παρατηρήσει τις ιδιότητές των αντικειμένων που κουβαλάει και ενδεχομένως να επιλέξει κάποιο από αυτά προς χρήση (equip).

Κατάσταση συστήματος αντικειμένων (inventory state)
Στο αντικείμενο AIPlayer που αντιστοιχεί σε κάθε πελάτη, έχουμε τις εξής οριζόμενες από το χρήστη ιδιότητες:

- *tableOfCarriedItems* : ένας πίνακας που κρατά πληροφορίες για όλα τα αντικείμενα που κουβαλάει ο πελάτης. Θεωρούμε ότι έχει σταθερό μέγεθος, ίσο με το μέγιστο πλήθος αντικειμένων που μπορεί να κουβαλάει ένας πελάτης. Κάθε γραμμή του πίνακα έχει πέντε στήλες:
  1. όνομα του αντικειμένου
  2. σύντομη περιγραφή του αντικειμένου
  3. τη διαδρομή (path) προς την εικόνα που αντιστοιχεί στο αντικείμενο
  4. το handle του αντικειμένου του datablock ShapeBaseImageData με το οποίο "σχετίζεται" το αντικείμενο (βλ. Παρένθεση)
  5. το slot στο οποίο γίνεται mount το αντικείμενο (ή πιο σωστά το αντικείμενο του datablock ShapeBaseImageData με το οποίο "σχετίζεται" το αντικείμενο).
- *maxNumOfCarriedItems*: το μέγιστο πλήθος αντικειμένων που μπορεί να κουβαλάει ανά πάσα στιγμή ένας πελάτης. Ουσιαστικά πρόκειται για μία ιδιότητα που κρατά το (σταθερό) μέγεθος (πλήθος γραμμών) του πίνακα tableOfCarriedItems.
- *currentNumOfCarriedItems* : το τρέχον πλήθος αντικειμένων που κουβαλάει ο πελάτης (πλήθος έγκυρων γραμμών) και συνεπώς που βρίσκονται στο inventory.

(Παρένθεση : το datablock *ShapeBaseImageData* στην πραγματικότητα δεν σχετίζεται με καμία κλάση. Είναι ένα ειδικό datablock που αναπαριστά αντικείμενα τα οποία *δεν μπορούν να υπάρξουν ανεξάρτητα στον κόσμο* -γι αυτό και δεν έχει κάποια αντίστοιχη κλάση-, αλλά μόνο να προσαρτηθούν -*mount*- σε άλλα ήδη υπάρχοντα αντικείμενα της κλάσης ShapeBase -και των υποκλάσεών της-.

Συνηθίζεται για κάθε αντικείμενο που μπορεί να γίνει mount σε κάποιο άλλο, να δημιουργούμε και ένα αντικείμενο του datablock ShapeBaseImageData, και όταν θέλουμε να κάνουμε mount το αντικείμενο, να κάνουμε στην πραγματικότητα mount το αντικείμενο του datablock ShapeBaseImageData που του αντιστοιχεί. Αυτό έχει ως αποτέλεσμα το αρχικό αντικείμενο να μην επηρεάζεται -δηλ. παραμένει εκεί που ήταν εκτός και αν το αφαιρέσουμε εμείς ρητά, πράγμα χρήσιμο για αντικείμενα που θέλουμε να μην "εξαφανίζονται" ώστε να είναι διαθέσιμα σε όλους τους πελάτες και όχι μόνο σε αυτόν που θα τα πάρει πρώτος-. Επίσης φαίνεται πως η ανίχνευση

συγκρούσεων (collision detection) αγνοεί αντικείμενα του datablock ShapeBaseImageData, πράγμα που από τη μία είναι μη ρεαλιστικό -τα αντικείμενα αυτά περνάνε μέσα από τοίχους κλπ.-, αλλά από την άλλη βοηθά στο να αποφεύγονται συνεχείς ανιχνεύσεις συγκρούσεων του mounted αντικειμένου με το ίδιο το αντικείμενο στο οποίο είναι mounted –κάτι που θα μπορούσε να δυσχεραίνει την κίνηση του δεύτερου αυτού αντικειμένου ή και να την κάνει αδύνατη-.
Με άλλα λόγια, όταν κάνουμε mount ένα αντικείμενο του datablock ShapeBaseImageData -με τη μέθοδο mountImage() της κλάσης ShapeBase-, είναι σαν να κάνουμε mount μία "εικόνα" (image) του αντικειμένου και όχι το ίδιο το αντικείμενο. Γίνεται πάντως να κάνουμε mount και το ίδιο το αντικείμενο -με τη μέθοδο mountObject() της κλάσης ShapeBase-.)

Το inventory ανανεώνεται αυτόματα όταν ο πελάτης αλληλεπιδράσει με κάποιο αντικείμενο που μπορεί να πάρει (το αντικείμενο προστίθεται στο inventory αν υπάρχει χώρος) ή όταν ρίξει το αντικείμενο που χρησιμοποιεί (το αντικείμενο αφαιρείται από το inventory).

Για κάθε αντικείμενο που κουβαλάει ο πελάτης, στο inventory αντιστοιχεί ένα *εικονίδιο / κουμπί* που είναι αντικείμενο της (Gui) κλάσης *GuiBitmapButtonCtrl* (συμπεριφέρεται ως κουμπί όπου μπορεί να κάνει κλικ ο χρήστης με το ποντίκι, μόνο που επιπλέον έχει και μία εικόνα, η οποία εδώ αντιπροσωπεύει ένα αντικείμενο του inventory, και η οποία εικόνα μπορεί να είναι διαφορετική για διάφορες καταστάσεις του κουμπιού -π.χ. πατημένο κλπ.), και μία *περιγραφή* που είναι αντικείμενο της (Gui) κλάσης *GuiMLTextCtrl* (η οποία αρχικά δεν φαίνεται). Στην πάνω αριστερή γωνία του κουμπιού αυτού βρίσκεται ένα μικρό βοηθητικό εικονίδιο (*help icon*), και αν το ποντίκι κινηθεί πάνω από την περιοχή του, εμφανίζεται η περιγραφή του αντίστοιχου αντικειμένου (σαν pop-up). Για την ανίχνευση αυτή της κίνησης του ποντικιού γίνεται χρήση αντικειμένων της κλάσης *GuiMouseEventCtrl*. (βλ. /control/client/gui/InventoryGui.gui)
Ενδεικτικά, το πρώτο εικονίδιο / κουμπί με την αντίστοιχη περιγραφή και το help icon του το δημιουργούμε ως εξής:

```
new GuiBitmapButtonCtrl(InventoryGuiButton0)
                {
                        command = "";
                        extent = "0 0";
                        position = "0 0";
                        profile = OpaqueProfile;
                        visible = true;

                        bitmap = "";

                        // So that GuiMouseEventCtrl won't cover all of the GuiBitmapButtonCtrl,
                        // or else it won't allow it to get any user input
                        new GuiBitmapCtrl()
                        {
                                extent = "15 15";
                                position = "0 0";
                                profile = TransparentProfile;
                                visible = true;

                                wrap = false;
                                bitmap = "./helpIcon.png";

                                new GuiMouseEventCtrl(InventoryGuiMouseEvent0)
                                {
```

```
                                        lockMouse = false;
                                };
                        };
                };
                new GuiMLTextCtrl(InventoryGuiDescription0)
                {
                        extent = "0 0";
                        position = "0 0";
                        profile = InventoryGuiMLTextCtrlProfile;
                        visible = false;

                        allowColorChars = true;
                        maxChars = 1000;
                        text = "";
                };
```

Όταν ο χρήστης πατήσει το πλήκτρο "I", ο πελάτης ανοίγει το inventory (αν προηγουμένως ήταν κλειστό) ή το κλείνει (αν προηγουμένως ήταν ανοιχτό). Κατά το άνοιγμα του inventory ο πελάτης κάνει χονδρικά τα εξής: κλείνει το quest log GUI αν αυτό είναι ανοιχτό, γίνεται fade-in του inventory GUI, απενεργοποιεί τις αντιστοιχίες πλήκτρων (key mappings) κίνησης ώστε ο avatar του χρήστη να μην μπορεί να κινηθεί, τοποθετούνται κάποια controls στη θέση τους, ξεκινά το effect με τα 'αστέρια' που αναβοσβήνουν (δεν προσφέρει κάτι από λειτουργικής άποψης και δεν ασχολούμαστε εδώ μαζί του), τα εικονίδια / κουμπιά και οι περιγραφές των αντικειμένων επανα-αρχικοποιούνται (reset), κρύβονται τα controls του HUD GUI, το ποντίκι τίθεται έτσι ώστε η λειτουργία του να είναι όπως στα μενού (mouse gui look) αντί να 'περιστρέφει' την κάμερα όπως συνηθίζεται κατά την περιήγηση στον κόσμο (mouse free look). Κατά το κλείσιμο γίνονται οι ανάστροφες διαδικασίες. (βλ.. /control/client/scripts/inventoryGui.cs)

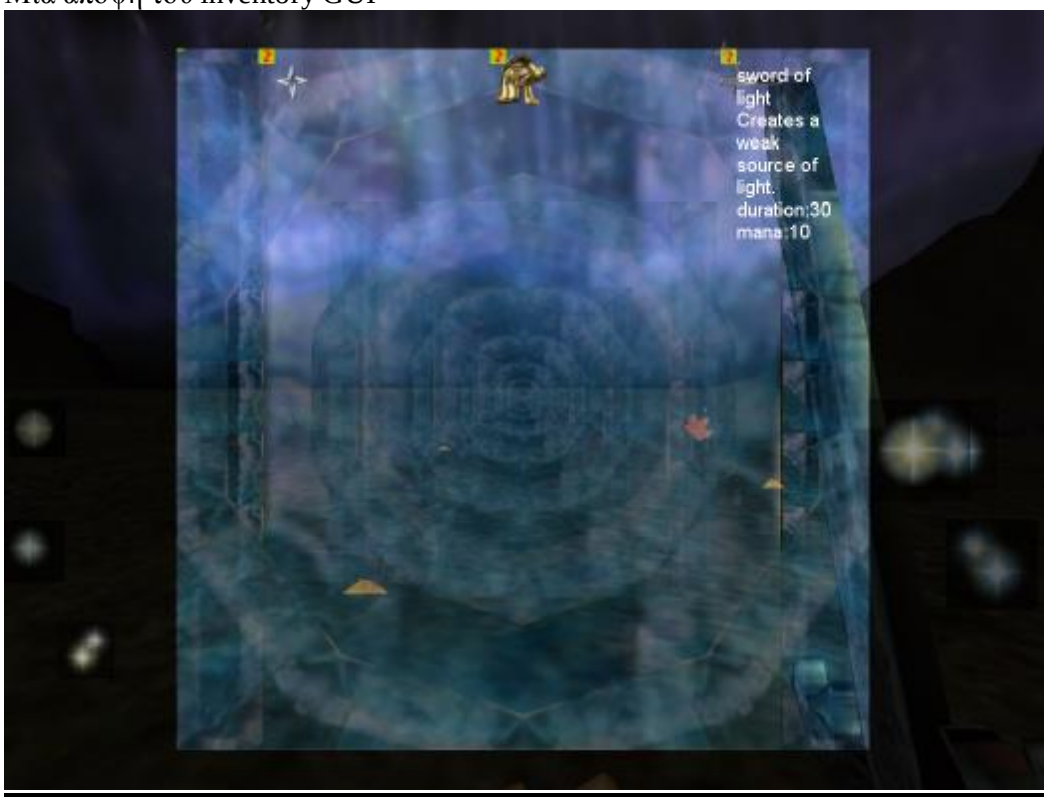
Αφού γίνουν τα παραπάνω βήματα κατά το άνοιγμα του inventory GUI, ο πελάτης στέλνει μήνυμα στον εξυπηρετητή ζητώντας του να στείλει την πληροφορία που ο τελευταίος κρατάει για την κατάσταση του inventory του χρήστη, ώστε να μπορέσει ο πελάτης να εμφανίσει αυτή την πληροφορία στον χρήστη. Ο εξυπηρετητής, ανταποκρινόμενος σε αυτό το μήνυμα (βλ. συνάρτηση serverCmdRequestInventoryInfo @ /control/server/scripts/game.cs) στέλνει την πληροφορία που έχει αποθηκευμένη στον πίνακα tableOfCarriedItems που είδαμε παραπάνω (σχετικά με την αποστολή πινάκων μέσω του μηχανισμού μηνυμάτων του Torque, βλ. ενότητα "Γενικές αρχές σχεδίασης"). Μάλιστα, στέλνεται μόνο η απαραίτητη πληροφορία στον πελάτη, δηλ. οι τρεις πρώτες στήλες κάθε γραμμής.

Όταν ο πελάτης έχει πλέον όλη την πληροφορία που χρειάζεται, καλεί την συνάρτηση inventoryGui::setUpInventory(), η οποία για κάθε αντικείμενο του πίνακα tableOfCarriedItems (που ο πελάτης έχει ανακατασκευάσει στην πλευρά του με το όνομα $Client::inventoryGui::tableOfCarriedItems) κάνει τρία πράγματα: θέτει την εικόνα του εικονιδίου / κουμπιού, την εντολή (command) που αυτό θα εκτελεί όταν πατηθεί, και τοποθετεί το εικονίδιο / κουμπί στην κατάλληλη θέση. Ο αριθμός των εικονιδίων / κουμπιών που είναι διαθέσιμα στο inventory GUI είναι σταθερός (ίσος με maxNumOfCarriedItems), και η αντιστοίχηση των αντικειμένων που κουβαλάει ο πελάτης σε αυτά γίνεται με τη σειρά με την οποία διαβάζονται από τον πίνακα.

Στην παραπάνω διαδικασία, σαν εντολή για κάθε εικονίδιο / κουμπί τίθεται η κλήση της συνάρτησης inventoryGui_GuiBitmapButtonPressed() με όρισμα τον αύξον αριθμό του αντικειμένου, δηλ. την θέση του στον αρχικό πίνακα tableOfCarriedItems που διατηρεί ο εξυπηρετητής. Όταν πατηθεί ένα εικονίδιο /

κουμπί, ο πελάτης στέλνει μήνυμα στον εξυπηρετητή που του λέει να αλληλεπιδράσει με το αντικείμενο του inventory που έχει αυτόν τον αύξον αριθμό. Ο εξυπηρετητής λαμβάνει το μήνυμα (βλ. serverCmdInventoryGenericInteractAction() @ /control/server/scripts/game.cs) και με βάση τα στοιχεία του πίνακα tableOfCarriedItems βρίσκει το αντικείμενο του datablock ShapeBaseImageData με το οποίο "σχετίζεται" το αντικείμενο του inventory, και το κάνει mount στο κατάλληλο slot (εφ' όσον όντως το αντικείμενο μπορεί να χρησιμοποιηθεί με αυτό τον τρόπο). Αν υπήρχε άλλο αντικείμενο σε αυτό το slot αντικαθίσταται από το καινούριο (το παλιό αντικείμενο παραμένει στο inventory). Τέλος, ο εξυπηρετητής στέλνει μήνυμα στον πελάτη με τις κατάλληλες πληροφορίες (όνομα, εικόνα, διαστάσεις εικόνας) ώστε αυτός να ανανεώσει την παρουσίαση του τρέχοντος αντικειμένου που έχει επιλεχθεί προς χρήση (equipped), όπως αυτές παρουσιάζονται στο πάνω δεξιά τμήμα του HUD (βλ. ενότητα "Heads Up Display (HUD)").

Μια άποψη του inventory GUI



Σχ.11 Μια άποψη του inventory GUI

(είναι ημιδιαφανές και φαίνεται μέρος του κόσμου από πίσω. Υπάρχουν τρία αντικείμενα. Το ποντίκι είναι πάνω από το help icon ενός αντικειμένου -sword of light-. Δεξιά και αριστερά φαίνεται το effect με τα 'αστέρια')

8. ***Σύστημα διαλόγου (Dialog system)***

Σχετικά αρχεία:
/control/client/gui/DialogGui.gui
/control/client/scripts/dialogGui.cs
/control/server/scripts/game.cs
/control/server/scripts/inventory_dialogs_quests.cs
/control/server/datablocks/NPCs.cs
/control/data/subtitles/

Το σύστημα διαλόγου βοηθά στην επικοινωνία του χρήστη με τους NPCs (Non-Player Characters) που βρίσκονται στον κόσμο, και είναι ο ένας τρόπος αλληλεπίδρασής του με αυτούς (ο άλλος τρόπος είναι το σύστημα μάχης).

Σε κάθε NPC αντιστοιχεί μία ακολουθία (sequence) από δυνατές *σκηνές (dialog scenes)*. Η ακολουθία αυτή είναι αποθηκευμένη σε ένα αρχείο απλού κειμένου (plain text) - του οποίου η επεξεργασία καλό είναι να γίνεται με προγράμματα όπως το WordPad που δεν διπλώνουν τις γραμμές όσο μακριές και αν είναι αυτές. Κάθε σκηνή έχει έναν αναγνωριστικό αριθμό, το κείμενο του NPC μεταξύ των διαχωριστικών (delimiters) `[NPC_TEXT_START]` και `[NPC_TEXT_END]` , και το κείμενο που αντιστοιχεί στις αριθμημένες επιλογές του χρήστη μεταξύ των διαχωριστικών `[SCENE_CHOICES_START]` και `[SCENE_CHOICES_END]` και καταλαμβάνει ακριβώς μία γραμμή στο αρχείο κειμένου.

Έτσι μοιάζει το απλό αρχείο με την ακολουθία σκηνών για τον NPC FunnyGuy (δυστυχώς οι γραμμές εδώ θα φαίνονται διπλωμένες) (/control/data/subtitles/funnyGuySubtitles.txt):

```
1 [NPC_TEXT_START]Hi there.[NPC_TEXT_END] [SCENE_CHOICES_START]1. Hi
<br>2. I'm outta here...[SCENE_CHOICES_END]
2 [NPC_TEXT_START]Want to see something fun?[NPC_TEXT_END]
[SCENE_CHOICES_START]1. Well, i guess so! <br>2. So
long...[SCENE_CHOICES_END]
3 [NPC_TEXT_START]While in the game world, press <font:Verdana
Bold:20><color:FF0000>F12[NPC_TEXT_END] [SCENE_CHOICES_START]1.
Thanks, I'll give it a try.[SCENE_CHOICES_END]
```

Κατάσταση διαλόγου (dialog state)

Στο αντικείμενο AIPlayer που αντιστοιχεί σε κάθε πελάτη, έχουμε τις εξής οριζόμενες από το χρήστη ιδιότητες:

- *interlocutor* : ο συνομιλητής του πελάτη.
- *dialog* : μία συμβολοσειρά (string) που περιέχει όλο το κείμενο του διαλόγου (ακολουθία σκηνών) με τον τρέχοντα συνομιλητή (τα περιεχόμενα του αρχείου που είδαμε παραπάνω).
- *scene* : η τρέχουσα σκηνή στην οποία βρίσκεται ο διάλογος με τον τρέχοντα συνομιλητή.

(οι τρεις παραπάνω ιδιότητες έχουν νόημα μόνο όσο ο πελάτης βρίσκεται σε κατάσταση διαλόγου, σε οποιαδήποτε άλλη περίπτωση είναι μηδενικές / κενές)

- *tableOfDialogs* : ένας πίνακας που κρατά πληροφορίες για όλους τους διαλόγους στους οποίους συμμετείχε ο πελάτης. Το μέγεθός του αυξομειώνεται δυναμικά Έχει μία γραμμή για κάθε διάλογο. Κάθε γραμμή του πίνακα έχει δύο στήλες:
  1. όνομα του NPC στον οποίο αντιστοιχεί αυτός ο διάλογος

2. διάθεση του NPC προς τον πελάτη. Αυτή εξαρτάται συνήθως από την μέχρι τώρα πορεία του διαλόγου.

- *currentNumOfDialogs* : το πλήθος των διαλόγων στους οποίους ο πελάτης έχει συμμετάσχει. Ουσιαστικά πρόκειται για μία ιδιότητα που κρατά το τρέχον μέγεθος (πλήθος γραμμών) του (δυναμικού) πίνακα tableOfDialogs.

Όταν ο avatar του χρήστη κοιτάει έναν NPC και πατήσει το πλήκτρο "E", αν είναι αρκετά κοντά στον NPC, το πρώτο πράγμα που συμβαίνει είναι εστίαση της κάμερας του πελάτη στον NPC και zoom-in. Στη συνέχεια ο εξυπηρετητής στέλνει μήνυμα στον πελάτη να εισέλθει σε κατάσταση διαλόγου (dialog mode). Ο πελάτης κάνει push το DialogGui στον Canvas, απενεργοποιεί τις αντιστοιχίες πλήκτρων (key mappings) κίνησης ώστε ο avatar του χρήστη να μην μπορεί να κινηθεί, απενεργοποιεί / κρύβει τα στοιχεία του HUD και τον cursor του ποντικιού.

Στην πλευρά του πελάτη, όταν είμαστε σε κατάσταση διαλόγου το GUI αποτελείται από ένα μέρος για το κείμενο του NPC (πάνω μέρος της οθόνης) και ένα μέρος για το κείμενο με τις επιλογές του χρήστη (κάτω μέρος της οθόνης). Ο χρήστης κάνει την επιλογή του για την τρέχουσα σκηνή με τα πλήκτρα 1-9. Οποιοδήποτε άλλο πλήκτρο αγνοείται, και αν ο χρήστης επιλέξει ένα νούμερο που δεν αντιστοιχεί σε έγκυρη επιλογή για την τρέχουσα σκηνή αυτό αγνοείται επίσης (για να "πιάσουμε" την είσοδο του χρήστη, χρησιμοποιούμε την κλάση GuiInputCtrl).

Αφού ο χρήστης κάνει την επιλογή του, ο πελάτης στέλνει μήνυμα στον εξυπηρετητή να ανανεώσει τον διάλογο (δηλ. να κάνει ότι ενέργειες πιθανώς συνεπάγεται η επιλογή του χρήστη, και να μεταβεί στην κατάλληλη σκηνή του διαλόγου ανάλογα με την επιλογή αυτή). Το μόνο που στέλνει ο πελάτης στον εξυπηρετητή είναι τα νούμερο που επέλεξε ο χρήστης για την τρέχουσα σκηνή, όλες οι άλλες πληροφορίες διατηρούνται σαν κατάσταση του διαλόγου στον εξυπηρετητή. (βλ. μέθοδο DialogGui::dialogGuiUpdateRequest() @ /control/client/scripts/dialogGui.cs)

Ο εξυπηρετητής πιάνει αυτά τα μηνύματα (βλ. συνάρτηση serverCmdDialogGuiUpdateRequest() @ /control/server/scripts/game.cs), βλέπει ποιος είναι ο συνομιλητής του πελάτη που ζητά την ενημέρωση του διαλόγου του, και αφήνει το αντικείμενο του κατάλληλου NPC να αποφασίσει αυτό τι ενέργειες χρειάζονται (βλ. /control/server/datablocks/NPCs.cs)

Κάθε NPC ο οποίος μπορεί να συμμετέχει σε διάλογο ορίζει τη μέθοδο dialogGuiUpdateRequest() στο αντικείμενο datablock που του αντιστοιχεί (εδώ χρησιμοποιούμε ένα αντικείμενο datablock ανά NPC). Αυτή η μέθοδος είναι το βασικό σημείου χειρισμού του διαλόγου στη μεριά του εξυπηρετητή. Ο πελάτης για κάθε σκηνή (*%scene*) επιλέγει μία από τις διαθέσιμες επιλογές (*%sceneChoice*), και τη δεδομένη στιγμή που γίνεται αυτή η επιλογή ο NPC έχει μία συγκεκριμένη διάθεση προς τον πελάτη (*%dialogNPCDisposition*). Με βάση αυτά τα τρία στοιχεία, ο εξυπηρετητής εκτελεί όποιες ενέργειες έχουν καθοριστεί, και επιλέγει την επόμενη σκηνή στην οποία θα μεταβεί ο διάλογος. Στη συνέχεια, στέλνει στον πελάτη το κείμενο της επόμενης σκηνής, ώστε ο τελευταίος να ανανεώσει το GUI του.
(Σημείωση: ο μηχανισμός μηνυμάτων του Torque δεν μας επιτρέπει να στείλουμε αυθαίρετου μήκους συμβολοσειρές στα πλαίσια ενός μηνύματος. Αυτό μας περιορίζει στην περίπτωση των διαλόγων, και το αντιμετωπίζουμε σπάζοντας το κείμενο που πρέπει να αποσταλεί σε κομμάτια με μέγιστο μέγεθος 255 χαρακτήρων. Αυτό σημαίνει ότι ο παραλήπτης πρέπει να γνωρίζει πόσα κομμάτια ακόμα να περιμένει για να λάβει όλη τη συμβολοσειρά. Ένας καλύτερος τρόπος θα ήταν να αντιστοιχούμε σε κάθε συμβολοσειρά που πρέπει να σταλεί ένα αναγνωριστικό, και να στέλνουμε μόνο το αναγνωριστικό στον πελάτη, ο οποίος σε αυτή την περίπτωση θα πρέπει να έχει

αποθηκευμένες εκ' των προτέρων αυτές τις συμβολοσειρές και τις αντιστοιχίες τους με τα αναγνωριστικά.)

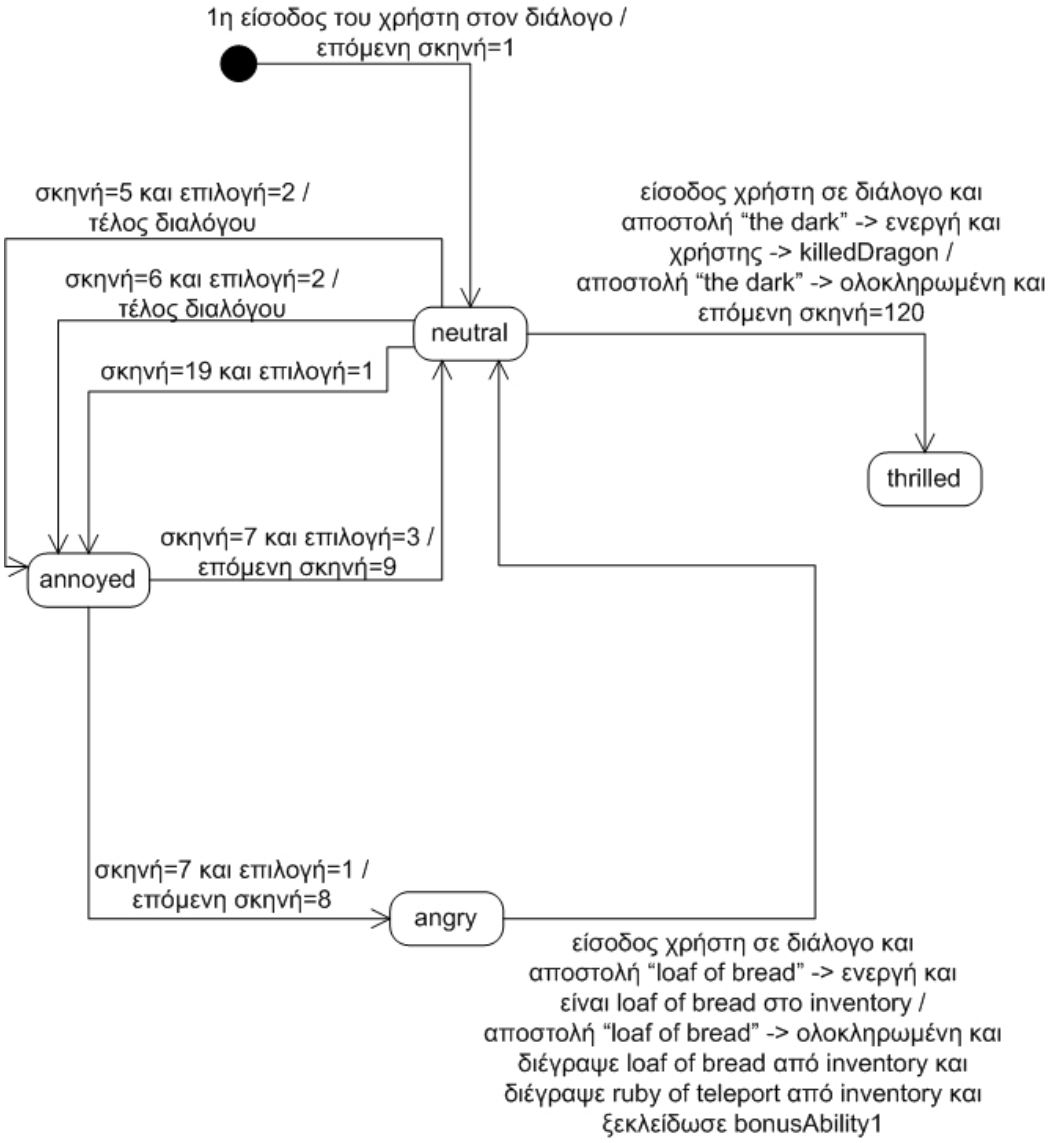
Στην ενδεικτική περίπτωση του FunnyGuy NPC, η μέθοδος dialogGuiUpdateRequest() είναι αυτή:

```
function FunnyGuyData::dialogGuiUpdateRequest(%this, %player, %sceneChoice)
{
	%scene = %player.scene;

	// ( scene "0" + sceneChoice "0" indicate we just entered the dialog)
	if (%scene == 0 && %sceneChoice == 0)
	{
		// If this is the first request in this dialog, get the whole dialog as a string, the file's
		// lines being delimeted by $Server::NPC::tokenDelimeter
		// (EACH line in the file has all the info corresponding to a "scene")
		%player.dialog = NPC::getFileContents(%this.dialogFile);
	}

	%dialog = %player.dialog;

	// switch between possible choices (based on the current scene and the choice the palyer made)
	%nextSceneString = "";
	if (%scene == 0)
	{
		if (%sceneChoice == 0)
		{
			%nextSceneString = NPC::getDialogScene(%dialog, 1);
		}
	}
	else if (%scene == 1)
	{
		if (%sceneChoice == 1)
		{
			%nextSceneString = NPC::getDialogScene(%dialog, 2);
		}
		else if (%sceneChoice == 2)
		{
			%this.endDialog(%player);
		}
	}
	else if (%scene == 2)
	{
		if (%sceneChoice == 1)
		{
			%nextSceneString = NPC::getDialogScene(%dialog, 3);
		}
		else if (%sceneChoice == 2)
		{
			%this.endDialog(%player);
		}
	}
	else if (%scene == 3)
	{
		if (%sceneChoice == 1)
		{
			%this.endDialog(%player);
		}
	}
```

```
        // if %nextSceneString is still empty after the 'switch' above, this means that we don't have a
        // valid scene-sceneChoice pair
        // (e.g. if the client chose '9' but the scene only has 5 choices)
        // Ignore such cases:
        if (%nextSceneString $= "")
                return;

        // update the information about the dialog's state that are held on the Player object
        // %interlocutor and %dialog are not changed
        // extract the new scene number
        %player.scene = getWord(%nextSceneString, 0);

        // extract the NPC dialog text
        %startIndex = strPos(%nextSceneString, "[NPC_TEXT_START]") + 16;
        %endIndex = strPos(%nextSceneString, "[NPC_TEXT_END]");
        %textNPC = getSubStr(%nextSceneString, %startIndex, (%endIndex - %startIndex) );

        // extract the player dialog text (scene choices)
        %startIndex = strPos(%nextSceneString, "[SCENE_CHOICES_START]") + 21;
        %endIndex = strPos(%nextSceneString, "[SCENE_CHOICES_END]");
        %textSceneChoices = getSubStr(%nextSceneString, %startIndex, (%endIndex - %startIndex)
                                        );

        // send to the client the text he must present for the new scene
        %clientGameConnection = %player.getControllingClient();  // ShapeBase class method

        // NOTE:
        // with the 'messaging' mechanism (client/server commands), we can't send long strings, or
        // they'll be cut by the TGE !
        // It seems that the max length that can be sent is 255 characters
        //So, we fragment the text and send each fragment. The client obviously must handle the
        // fragments.

        %textNPCFragments = mCeil(strlen(%textNPC) / 255);

        for (%i = 0; %i < %textNPCFragments; %i++)
        {
                %fragmentTextNPC = getSubStr(%textNPC, %i * 255, 255);
                commandToClient(%clientGameConnection, 'DialogGuiUpdateNPCTextResponse',
                                %fragmentTextNPC, (%textNPCFragments - (%i + 1)));
                                                        // @client/scripts/dialogGui.cs
        }

        %textSceneChoicesFragments =mCeil(strlen(%textSceneChoices) / 255);

        for (%i = 0; %i < %textSceneChoicesFragments; %i++)
        {
                %fragmentTextSceneChoices = getSubStr(%textSceneChoices, %i * 255, 255);
                commandToClient(%clientGameConnection,
                                'DialogGuiUpdateSceneChoicesTextResponse',
                                %fragmentTextSceneChoices,
                                (%textSceneChoicesFragments - (%i + 1)));
                                                        // @client/scripts/dialogGui.cs
        }
}
```

Ένας NPC μπορεί να καθορίζει ένα σύνολο καταστάσεων που αντιπροσωπεύουν τη διάθεσή του απέναντι στο χρήστη, σαν αποτέλεσμα των

ενεργειών του τελευταίου. Για παράδειγμα, για τον NPC "Sage Tree", το διάγραμμα καταστάσεών του (UML state diagram) είναι το ακόλουθο:



Σχ.12 Διάγραμμα καταστάσεων του NPC "Sage Tree"

(οι δυνατές καταστάσεις -states- είναι "thrilled", "neutral", "annoyed", "angry". Στα βέλη μεταβάσεων, το κείμενο είναι της μορφής "σήμα εισόδου / δράση" – αυτόματο Mealy).

Μια άποψη του dialog GUI:

Σχ.13 Μια άποψη του dialog GUI

9. ***Σύστημα καταγραφής αποστολών (Quest log)***

Σχετικά αρχεία:
/control/client/gui/QuestLogGui.gui
/control/client/scripts/questLogGui.cs
/control/server/scripts/game.cs
/control/server/scripts/inventory_dialogs_quests.cs

Κατά τη διάρκεια του διαλόγου με έναν NPC (ή γενικότερα κατά την αλληλεπίδρασή του με τον κόσμο), μπορεί ο χρήστης να αναλάβει μία αποστολή (quest). Για παράδειγμα, ένας NPC μπορεί να θυμώσει με έναν χρήστη και να του ζητήσει σαν αγγαρεία να βρει και να φέρει κάποιο αντικείμενο (πιθανώς δύσκολο να βρεθεί), πριν συνεχίσουν τον διάλογό τους (στο demo, αυτό γίνεται στον διάλογο με το SageTree αν το θυμώσει ο χρήστης την πρώτη φορά που του μιλάει).

Υπάρχει λοιπόν ανάγκη να καταγράφονται αυτές οι αποστολές και η κατάστασή τους, και να μπορούν να παρουσιαστούν στον χρήστη με έναν ενιαίο τρόπο, ώστε αυτός να μπορεί να ενημερωθεί ανά πάσα στιγμή για την εξέλιξή τους και το τι πρέπει να κάνει μετά. Αυτό το σύστημα συνολικά ονομάζεται quest log.

Το quest log είναι σε μεγάλο βαθμό ανάλογο του inventory (βλ. ενότητα "Σύστημα αντικειμένων (Inventory)"), καθώς παρόμοια ζητήματα πρέπει να αντιμετωπιστούν. Αλλάζουν κυρίως τα χαρακτηριστικά της πληροφορίας που πρέπει να διατηρείται από τον εξυπηρετητή και να παρουσιάζεται στον πελάτη.

Κατάσταση συστήματος καταγραφής αποστολών (quest log state)
Στο αντικείμενο AIPlayer που αντιστοιχεί σε κάθε πελάτη, έχουμε τις εξής οριζόμενες από το χρήστη ιδιότητες:

- *tableOfQuests*: ένας πίνακας που κρατά πληροφορίες για όλες τις αποστολές που έχει αναλάβει ο πελάτης. Το μέγεθός του αυξομειώνεται δυναμικά Έχει μία γραμμή για κάθε αποστολή. Κάθε γραμμή του πίνακα έχει *τουλάχιστον* τρεις στήλες:
  1. όνομα της αποστολής
  2. σύντομη κείμενο με περιγραφή της αποστολής
  3. κατάσταση της αποστολή (ενεργή – active, ολοκληρωμένη – completed, αποτυχημένη – failed)

  Ανάλογα με την αποστολή μπορεί να χρειαστούν περισσότερες στήλες για κάποια γραμμή (π.χ. μια στήλη που να δείχνει πόσες υπο-αποστολές της αποστολής ολοκλήρωσε ο χρήστης, σε περίπτωση που κάποια αποστολή υποδιαιρείται σε υπο-αποστολές). Είναι ευθύνη της κάθε αποστολής να χειρίζεται τις επιπλέον αυτές στήλες.
- *currentNumOfQuests*: το πλήθος των αποστολών τις οποίες έχει αναλάβει ο πελάτης. Ουσιαστικά πρόκειται για μία ιδιότητα που κρατά το τρέχον μέγεθος (πλήθος γραμμών) του (δυναμικού) πίνακα tableOfQuests.

Το quest log GUI αποτελείται βασικά από ένα αντικείμενο της κλάσης *GuiMessageVectorCtrl* στο οποίο τυπώνονται οι πληροφορίες για κάθε αποστολή που έχει αναλάβει ο πελάτης. Αυτές είναι το *όνομα* και η *περιγραφή* της φορμαρισμένα με κατάλληλο *χρωματικό κώδικα* που υποδεικνύει την τρέχουσα κατάσταση της αποστολής (ενεργή **ό** μαύρο, ολοκληρωμένη **ό** μπλε, αποτυχημένη **ό** κόκκινο). Το αντικείμενο της κλάσης GuiMessageVectorCtrl πρέπει να βρίσκεται μέσα σε ένα

αντικείμενο της κλάσης *GuiScrollCtrl* (όπως πάντα, ιδιαιτερότητα της κλάσης GuiMessageVectorCtrl). (βλ. /control/client/gui/QuestLogGui.gui)

Όταν ο χρήστης πατήσει το πλήκτρο "L", ο πελάτης ανοίγει το quest log (αν προηγουμένως ήταν κλειστό) ή το κλείνει (αν προηγουμένως ήταν ανοιχτό). Κατά το άνοιγμα του quest log ο πελάτης κάνει χονδρικά τα εξής: κλείνει το inventory GUI αν αυτό είναι ανοιχτό, συσχετίζει (attach) το αντικείμενο της κλάσης GuiMessageVectorCtrl με ένα αντικείμενο της κλάσης *MessageVector* (το οποίο λειτουργεί σαν *buffer* για το κείμενο που παρουσιάζεται στο πρώτο αντικείμενο), απενεργοποιεί τις αντιστοιχίες πλήκτρων (key mappings) κίνησης ώστε ο avatar του χρήστη να μην μπορεί να κινηθεί, κάνει τα controls του HUD GUI μη ορατά, το ποντίκι τίθεται έτσι ώστε η λειτουργία του να είναι όπως στα μενού (mouse gui look) αντί να 'περιστρέφει' την κάμερα όπως συνηθίζεται κατά την περιήγηση στον κόσμο (mouse free look), γίνεται fade-in του quest log GUI, τοποθετούνται κάποια controls στη θέση τους, ξεκινά το effect με τα 'αστέρια' που αναβοσβήνουν (δεν προσφέρει κάτι από λειτουργικής άποψης και δεν ασχολούμαστε εδώ μαζί του), επανα-αρχικοποιείται (reset) το κείμενο του buffer. Κατά το κλείσιμο γίνονται οι ανάστροφες διαδικασίες. (βλ.. /control/client/scripts/questLogGui.cs)

Αφού γίνουν τα παραπάνω βήματα κατά το άνοιγμα του quest log GUI, ο πελάτης στέλνει μήνυμα στον εξυπηρετητή ζητώντας του να στείλει την πληροφορία που ο τελευταίος κρατάει για την κατάσταση του quest log του χρήστη, ώστε να μπορέσει ο πελάτης να εμφανίσει αυτή την πληροφορία στον χρήστη. Ο εξυπηρετητής, ανταποκρινόμενος σε αυτό το μήνυμα (βλ. συνάρτηση serverCmdRequestQuestLogInfo@ /control/server/scripts/game.cs) στέλνει την πληροφορία που έχει αποθηκευμένη στον πίνακα tableOfQuests που είδαμε παραπάνω (σχετικά με την αποστολή πινάκων μέσω του μηχανισμού μηνυμάτων του Torque, βλ. ενότητα "Γενικές αρχές σχεδίασης").

Όταν ο πελάτης έχει πλέον όλη την πληροφορία που χρειάζεται, καλεί την συνάρτηση QuestLogGui::setUpQuestLog(), η οποία για κάθε αποστολή του πίνακα tableOfQuests (που ο πελάτης έχει ανακατασκευάσει στην πλευρά του με το όνομα $Client::QuestLogGui::tableOfQuests) κάνει το εξής: ανάλογα με την τρέχουσα κατάσταση της αποστολής (ενεργή κλπ.), τυπώνει στον buffer το όνομα της αποστολής, την περιγραφή της και μία κενή γραμμή (ώστε οι διαφορετικές αποστολές να ξεχωρίζουν μεταξύ τους) με τον κατάλληλο χρωματικό κώδικα.

Η συνάρτηση QuestLogGui::setUpQuestLog() έχει ως εξής:

```
// Text area functions

// Container of text meant to be consumed by the QuestLogGuiTextArea GuiMessageVectorCtrl
new MessageVector(QuestLogGuiTextAreaMessageVector)
{

};

// set up the quest log text area
function QuestLogGui::setUpQuestLog(%currentNumOfQuests)
{
	for (%i = 0; %i < %currentNumOfQuests; %i++)
	{
		if ($Client::QuestLogGui::tableOfQuests[%i, 2] $= "active")
		{
			// active quests in \c1 => black
			QuestLogGuiTextAreaMessageVector.pushBackLine("\c1" @
					strupr($Client::QuestLogGui::tableOfQuests[%i, 0]), 0);
```

```
                        QuestLogGuiTextAreaMessageVector.pushBackLine("\c1" @
                                                $Client::QuestLogGui::tableOfQuests[%i, 1], 0);
                        QuestLogGuiTextAreaMessageVector.pushBackLine("", 0);  // empty line
                }
                else if ($Client::QuestLogGui::tableOfQuests[%i, 2] $= "completed")
                {
                        // completed quests in \c4 => blue
                        QuestLogGuiTextAreaMessageVector.pushBackLine("\c4" @
                                                strupr($Client::QuestLogGui::tableOfQuests[%i, 0]), 0);
                        QuestLogGuiTextAreaMessageVector.pushBackLine("\c4" @
                                                $Client::QuestLogGui::tableOfQuests[%i, 1], 0);
                        QuestLogGuiTextAreaMessageVector.pushBackLine("", 0);  // empty line
                }
                else if ($Client::QuestLogGui::tableOfQuests[%i, 2] $= "failed")
                {
                        // failed quests in \c0 => red
                        QuestLogGuiTextAreaMessageVector.pushBackLine("\c0" @
                                                strupr($Client::QuestLogGui::tableOfQuests[%i, 0]), 0);
                        QuestLogGuiTextAreaMessageVector.pushBackLine("\c0" @
                                                $Client::QuestLogGui::tableOfQuests[%i, 1], 0);
                        QuestLogGuiTextAreaMessageVector.pushBackLine("", 0);  // empty line
                }
        }
}
```

Μια άποψη του quest log GUI :

Σχ.14 Μια άποψη του quest log GUI

(είναι ημιδιαφανές και φαίνεται μέρος του κόσμου από πίσω. Υπάρχουν δύο αποστολές. Η πρώτη -loaf of bread- είναι ολοκληρωμένη και εμφανίζεται με μπλε χρώμα, η δεύτερη -the dark- είναι ενεργή και εμφανίζεται με μαύρο χρώμα. Δεξιά και αριστερά φαίνεται το effect με τα 'αστέρια')

10.Head-Up Display (HUD)

Σχετικά αρχεία:
/control/client/gui/HUDGui.gui
/control/client/scripts/hudGui.cs
/control/client/scripts/keyBindings.cs
/control/client/scripts/game.cs
/control/server/scripts/game.cs

Όταν ο πελάτης συνδεθεί με τον εξυπηρετητή και εισέλθει στον κόσμο, αναλαμβάνει πλέον την *απεικόνιση του κόσμου* με βάση τις πληροφορίες που του στέλνει ο εξυπηρετητής (η διατήρηση του συγχρονισμού των πελατών με τον εξυπηρετητή είναι ευθύνη της μηχανής και δεν μας απασχολεί σε επίπεδο script). Έτσι ο χρήστης μπορεί να βλέπει (και να ακούει) τι γίνεται στον κόσμο, και να δρα ανάλογα.

Η απεικόνιση του κόσμου είναι μία υπόθεση διαφορετική από την απεικόνιση των menus, με τα οποία έρχεται σε επαφή ο χρήστης όταν ανοίξει αρχικά την εφαρμογή (αν βέβαια η εφαρμογή έχει τέτοια menus), και γενικότερα των GUIs. Τα GUIs είναι σε τελική ανάλυση 2D εικόνες από pixels (pixmaps ή bitmaps αν σε κάθε pixel αντιστοιχεί μόνο ένα bit) και μπορούν σχετικά άμεσα να αντιστοιχηθούν σε pixels της οθόνης. Ο κόσμος όμως αποτελείται ουσιαστικά από μία συλλογή τρισδιάστατων περιγραφών αντικειμένων (3D models) και των μεταξύ τους σχέσεων, στα οποία αντικείμενα μπορεί να εφαρμόζονται εικόνες στην επιφάνειά τους (texture mapping) κλπ., υπάρχει κάποιο μοντέλο φωτισμού, και πολλά άλλα που δεν είναι εδώ η θέση να συζητηθούν. Το θέμα είναι ότι απαιτείται μία πολύπλοκη διαδικασία για τη μετατροπή αυτής της περιγραφής του κόσμου σε μία δυσδιάστατη εικόνα που μπορεί τελικά να παρουσιαστεί στην οθόνη (rendering). Το rendering γίνεται από τη μηχανή, συνήθως με τη βοήθεια βιβλιοθηκών ειδικών για γραφικά, όπως η OpenGL που χρησιμοποιεί το Torque.

Το rendering λοιπόν δε μας απασχολεί στο επίπεδο της TorqueScript, πρέπει όμως ο πελάτης να μπορεί να απεικονίσει στην οθόνη το αποτέλεσμα του rendering, αλλιώς ο χρήστης δεν θα μπορεί να δει τον κόσμο. Υπάρχει μία ειδική Gui κλάση για αυτό το σκοπό, η κλάση *GameTSCtrl*. Δημιουργούμε ένα αντικείμενο αυτής της κλάσης (βλ. /control/client/gui/HUDGui.gui), στο οποίο θα αναφερόμαστε ως *HUD* (έχει καθιερωθεί το όλο "κατασκεύασμα" μέσω του οποίου βλέπουμε τον κόσμο, αλλά και άλλες πληροφορίες, όπως η υγεία του avatar, μηνύματα σε μορφή κειμένου, κλπ. να ονομάζεται HUD –Head-Up Display-). Ο πελάτης θέτει το HUD σαν περιεχόμενο του Canvas (μέθοδος setContent()), και πλέον ο χρήστης μπορεί να δει τον κόσμο -πάντα μέσα από τα ¨μάτια" του control object (βλ. ενότητα "Θέματα δικτύωσης") που έχει ορίσει ο εξυπηρετητής για αυτόν.

Υπάρχουν κάποιες ειδικές κλάσεις, τα αντικείμενα των οποίων προορίζονται για χρήση σε ένα HUD, π.χ. οι κλάσεις GuiCrossHairHud (για τον συνηθισμένο "σταυρό" που εμφανίζεται στο κέντρο της οθόνης υποδεικνύοντας πού κοιτάει ο χρήστης), GuiHealthBarHud (κατάλληλο για δείκτες υγείας, ενέργειας κλπ. όταν θέλουμε αυτοί να ενημερώνονται αυτόματα από τη μηχανή. Η αυτόματη ενημέρωση των δεικτών αυτών σχετίζεται άμεσα με τις αντίστοιχες ιδιότητες του datablock ShapeBaseData και τις μεθόδους της κλάσης ShapeBase), κ.α..

Στοιχεία που αποτελούν το HUD

Πέρα από την ιδιαιτερότητα του HUD να παρουσιάζεται μέσω αυτού ο κόσμος, θα λέγαμε ότι είναι μία συλλογή από στοιχεία (που με τη σειρά τους αποτελούνται από ένα ή περισσότερα controls), καθένα απ' τα οποία έχει το δικό του ρόλο (συνήθως παρουσιάζει κάποιου είδους πληροφορία στον χρήστη) και καταλαμβάνει μία συγκεκριμένη περιοχή στην οθόνη. Μερικά χαρακτηριστικά στοιχεία του HUD σε αυτό το demo περιγράφονται παρακάτω (βλ. /control/client/gui/HUDGui.gui):

- Στο κέντρο της οθόνης έχουμε τον "σταυρό" που υποδεικνύει πού κοιτάει ο χρήστης, και σε περίπτωση που η νοητή ημιευθεία που ξεκινά από τα "μάτια" του αντικειμένου που ελέγχει ο πελάτης συναντά ένα άλλο αντικείμενο με το οποίο μπορεί αυτός να αλληλεπιδράσει (και αν είναι σχετικά κοντά στο αντικείμενο αυτό), εμφανίζεται το όνομα του αντικειμένου πάνω από τον "σταυρό".
- Κάτω αριστερά έχουμε μία περιοχή όπου ο χρήστης μπορεί να βλέπει τα μηνύματα που έστειλαν άλλοι χρήστες, ή ειδικά μηνύματα που στέλνει ο εξυπηρετητής για να ειδοποιήσει τον χρήστη για σημαντικά γεγονότα (όπως όταν παίρνει ένα αντικείμενο, όταν ρίχνει ένα αντικείμενο, όταν ολοκληρώνει μια αποστολή, κλπ.). Αν ο χρήστης πατήσει "Enter" εμφανίζεται κάτω από την περιοχή ο κέρσορας, και ο χρήστης μπορεί να γράψει ένα δικό του μήνυμα, που το στέλνει στους άλλους χρήστες πατώντας ξανά το "Enter".
- Πάνω αριστερά έχουμε την μπάρα με το mana του χρήστη, που περιβάλλεται από μία εικόνα για να φαίνεται πιο εντυπωσιακή.
- Πάνω δεξιά εμφανίζονται πληροφορίες σχετικά με το αντικείμενο που έχει επιλέξει προς χρήση (equip) ο χρήστης. Φαίνονται το slot στο οποίο έχει τοποθετηθεί το αντικείμενο, το όνομα του αντικειμένου, και μια εικόνα του αντικειμένου.
- Πάνω και στο κέντρο εμφανίζεται μια πυξίδα που βοηθάει στον προσανατολισμό του χρήστη, σε συνδυασμό με
- Ένα χάρτη, ο οποίος εμφανίζεται πατώντας το πλήκτρο "M". Ο χάρτης εμφανίζεται μόνο όταν ο χρήστης έχει κάποια ενεργή αποστολή, δείχνει μία σχετικά μικρή περιοχή του κόσμου και έχει πάνω του έως τρία "X", ένα κόκκινο για το στόχο της αποστολής, ένα πράσινο για την αφετηρία της αποστολής, και ένα κίτρινο που δείχνει τη θέση του avatar του χρήστη. Όσο ο χάρτης είναι ανοιχτός, ο εξυπηρετητής θα στέλνει περιοδικά τις νέες θέσεις αυτών των τριών ειδικών σημείων, συνεπώς ο χάρτης μπορεί να μένει ανοιχτός και να ενημερώνεται κανονικά ενώ ο avatar του χρήστη κινείται. Σε αυτή την έκδοσή του, δεν μπορούμε να επιλέξουμε για ποια αποστολή θα παρουσιάζει πληροφορία ο χάρτης, αλλά θα βλέπουμε πάντα την πληροφορία για την πρώτη ενεργή αποστολή του πίνακα tableOfQuests (βλ. ενότητα "Σύστημα αντικειμένων (Inventory)").

Όταν ο πελάτης θέσει το HUD GUI στον Canvas, γίνονται χονδρικά τα εξής: τοποθετούνται κάποια controls στην κατάλληλη θέση, ενεργοποιούνται οι αντιστοιχίες πλήκτρων με συναρτήσεις / ενέργειες (key mappings ή key bindings, βλ. /control/client/scripts/keyBindings.cs, π.χ. η κίνηση καθορίζεται να γίνεται με τα πλήκτρα "W", "A", "S", "D"), το αντικείμενο της κλάσης GuiMessageVectorCtrl (στο οποίο τυπώνονται τα διάφορα μηνύματα που απευθύνονται στον χρήστη, και το οποίο όπως πάντα πρέπει να βρίσκεται μέσα σε ένα αντικείμενο της κλάσης GuiScrollCtrl) συσχετίζεται (attach) με ένα αντικείμενο της κλάσης MessageVector (το οποίο λειτουργεί σαν buffer για το κείμενο που παρουσιάζεται στο πρώτο αντικείμενο), ξεκινάμε το μηχανισμό (scanning) που ελέγχει αν υπάρχει "κάτω από το ποντίκι"

αντικείμενο με το οποίο μπορεί να αλληλεπιδράσει ο χρήστης (ώστε να τυπώσουμε το όνομά του πάνω από τον "σταυρό"), ξεκινά ο μηχανισμός ενημέρωσης της πυξίδας.

Σχετικά με το *μηχανισμό ανίχνευσης (scanning) που ελέγχει αν ο χρήστης κοιτά κάποιο αντικείμενο με το οποίο μπορεί να αλληλεπιδράσει*, έχουμε στον πελάτη μία συνάρτηση (handleOnMouseOverDescription() @ /control/client/scripts/hudGui.cs) που στέλνει μήνυμα στον εξυπηρετητή να κάνει τον έλεγχο, και μετά επανα-προγραμματίζει (συνάρτηση κονσόλας schedule()) να γίνει ξανά κλήση στον εαυτό της ύστερα από κάποιο μικρό χρονικό διάστημα. Ο εξυπηρετητής (βλ. /control/server/scripts/game.cs) κάνει τον έλεγχο (με τη βοήθεια της συνάρτησης κονσόλας containerRayCast()), και αν βρει όντως τέτοιο αντικείμενο στέλνει την περιγραφή / όνομα του datablock που αντιστοιχεί στο αντικείμενο (ιδιότητα οριζόμενη-από-τον-χρήστη) πίσω στον πελάτη.

Εάν το ποντίκι είναι "πάνω" από κάποιο αντικείμενο, και ο avatar του χρήστη είναι αρκετά κοντά στο αντικείμενο, πατώντας το πλήκτρο "E" *ο χρήστης ζητά να αλληλεπιδράσει με το αντικείμενο*. Ο πελάτης έχει συσχετίσει με αυτό το πλήκτρο (βλ. /control/client/scripts/keyBindings.cs) τη συνάρτηση genericInteractAction() (@ /control/client/scripts/game.cs), η οποία στέλνει ένα μήνυμα στον εξυπηρετητή ζητώντας γενικά "να γίνει η αλληλεπίδραση". Ο εξυπηρετητής (βλ. συνάρτηση serverCmdGenericInteractAction() @ /control/server/scripts/game.cs) ελέγχει να δει αν όντως υπάρχουν αντικείμενα στην ευθεία που κοιτάει ο avatar του πελάτη και αρκετά κοντά του. Αν ναι, καλεί τη *μέθοδο genericInteractAction()* του αντικειμένου του datablock που αντιστοιχεί στο αντικείμενο που βρήκε, αφήνοντας το αντικείμενο να αποφασίσει μόνο του τί ενέργειες πρέπει να γίνουν (π.χ. οι NPCs θα ξεκινήσουν ένα διάλογο με τον χρήστη, τα αντικείμενα που μπορεί να πάρει ο χρήστης θα προστεθούν στο inventory του κλπ.). *Αυτό σημαίνει ότι κάθε αντικείμενο με το οποίο θέλουμε να μπορεί να αλληλεπιδράσει ένας χρήστης πρέπει να ορίζει τη μέθοδο genericInteractAction() στο χώρο ονομάτων του αντικειμένου του datablock που του αντιστοιχεί.*

Για την ενημέρωση του *χάρτη* (όταν αυτός είναι ανοιχτός) ακολουθείται παρόμοια τακτική. Ο πελάτης κάθε λίγο στέλνει μήνυμα στον εξυπηρετητή ζητώντας του την πληροφορία για τη θέση του avatar του χρήστη, του στόχου και της αφετηρίας που αντιστοιχούν στην πρώτη ενεργή αποστολή του tableOfQuests αυτού του πελάτη. Ο εξυπηρετητής στέλνει πίσω στον πελάτη τις συντεταγμένες για αυτές τις τρεις θέσεις, οι οποίες όμως είναι δοσμένες στο σύστημα συντεταγμένων του κόσμου (world coordinates). Για να παρουσιάσουμε τα "X" στο χάρτη, θα πρέπει να μετατρέψουμε τις συντεταγμένες στο σύστημα συντεταγμένων του χάρτη (έστω map coordinates). Αν και οι world coordinates είναι τριών διαστάσεων, η τρίτη διάσταση z (ύψος) δεν μας ενδιαφέρει εδώ, συνεπώς ο μετασχηματισμός συντεταγμένων είναι από 2διαστάσεις σε 2διαστάσεις, ουσιαστικά ένας απλός μετασχηματισμός αναλογίας. Αν xWorldMax, xWorldMin, yWorldMax, yWorldMin είναι τα (νοητά) όρια σε world coordinates, και xMapMax, xMapMin, yMapMax, yMapMin τα όρια σε map coordinates, τότε για δοσμένο xWorld το αντίστοιχο xMap βρίσκεται από τη σχέση αναλογίας:

(xWorld - xWorldMin) / (xWorldMax - xWorldMin) = (xMap - xMapMin) / (xMapMax - xMapMin)

Ανάλογα για το y, μόνο που εδώ πρέπει να προσέξουμε το εξής: *στα GUIs το y μετράται από πάνω προς τα κάτω*, οπότε η μέγιστη και η ελάχιστη τιμή του y σε map coordinates πρέπει να αντιστραφούν για να έχουμε σωστό αποτέλεσμα.

Ενδεικτικά, ο κώδικας του εξυπηρετητή που απαντάει στις αιτήσεις του πελάτη για ανανέωση των συντεταγμένων έχει ως εξής (@ /control/server/scripts/game.cs):

```
function serverCmdRequestMapInfo(%clientGameConnection)
{
	%controlObject = %clientGameConnection.getControlObject();

	if (%controlObject.getClassName() !$= "Player"
	    && %controlObject.getClassName() !$= "AIPlayer")
		return;

	// at this point the client has no way to choose between his active quests and see a map for one
	// in particular, the map will automatically be updated for the first active quest in the
	// tableOfQuests
	for (%i = 0; %i < %controlObject.currentNumOfQuests; %i++)
		if (%controlObject.tableOfQuests[%i, 2] $= "active")
		{
			// switch map info for specific quest
			%questName = %controlObject.tableOfQuests[%i, 0];

			%beginningPos = "NULL";
			%targetPos = "NULL";

			if (%questName $= "loaf of bread")
			{
				if (isObject(SageTree))
					%beginningPos = SageTree.getPosition();
				if (isObject(LoafOfBread))
					%targetPos = LoafOfBread.getPosition();
			}
			else if (%questName $= "the dark")
			{
				if (isObject(SageTree))
					%beginningPos = SageTree.getPosition();
				if (!%controlObject.killedGolem)
				{
					if (isObject(Golem))
						%targetPos = Golem.getPosition();
				}
				else  // killed golem
					if (isObject(Dragon))
						%targetPos = Dragon.getPosition();
			}

			commandToClient(%clientGameConnection, 'ResponseMapInfo',
					%controlObject.getPosition(),  // player's position
					%beginningPos,   // quest's beginning position
					%targetPos          // quest's target position
					);  // @ client/scripts/hudGui.cs

			return;  // stop at the 1st found
		}
}
```

και ο κώδικας του πελάτη που χειρίζεται την απάντηση του εξυπηρετητή και ενημερώνει τον χάρτη είναι ο εξής (@ /control/client/scripts/hudGui.cs):

```
function clientCmdResponseMapInfo(%playerPos, %beginningPos, %targetPos)
                                                    // note that positions are in world coordinates
{
	// some trivial mapping to 2D coordinates:
	// (z-coord is just dropped, so it's really a mapping from 2D to 2D)

	// determine maximum, minimum in the x,y world coordinates
	// fixed
```

```
%xWorldMax = 250;
%xWorldMin = -250;
%yWorldMax = 250;
%yWorldMin = -250;

// determine maximum, minimum in the x,y 'map' coordinates
%xMapMax = getWord(HUDGuiMap.getExtent(), 0);
%xMapMin = 0;
//%yMapMax = getWord(HUDGuiMap.getExtent(), 1);
//%yMapMin = 0;
// NOTE:
// since in the gui y is measured from top to bottom, reverse min, max
%yMapMax = 0;
%yMapMin = getWord(HUDGuiMap.getExtent(), 1);

// 2D -> 2D mapping, by analogy : for a given xWorld in world coords, the corresponding
// xMap is found by solving the equation :
// (xWorld - xWorldMin) / (xWorldMax - xWorldMin) =
// (xMap - xMapMin) / (xMapMax – xMapMin)
// xMap = xMapMin +
//                  (xWorld - xWorldMin) * (xMapMax - xMapMin) / (xWorldMax - xWorldMin)
// similarly for y
%constant = (%xMapMax - %xMapMin) / (%xWorldMax - %xWorldMin);
%xMap_playerPos = %xMapMin + (getWord(%playerPos, 0) - %xWorldMin) * (%constant);
if (%beginningPos !$= "NULL")
        %xMap_beginningPos = %xMapMin +
                                (getWord(%beginningPos, 0) - %xWorldMin) * (%constant);
if (%targetPos !$= "NULL")
        %xMap_targetPos = %xMapMin +
                                (getWord(%targetPos, 0) - %xWorldMin) * (%constant);


%constant = (%yMapMax - %yMapMin) / (%yWorldMax - %yWorldMin);
%yMap_playerPos = %yMapMin + (getWord(%playerPos, 1) - %yWorldMin) * (%constant);
if (%beginningPos !$= "NULL")
        %yMap_beginningPos = %yMapMin +
                                (getWord(%beginningPos, 1) - %yWorldMin) * (%constant);
if (%targetPos !$= "NULL")
        %yMap_targetPos = %yMapMin +
                                (getWord(%targetPos, 1) - %yWorldMin) * (%constant);

// now that we have the corresponding positions on the map, position the respective marker
// images CENTERED at these positions
HUDGuiMapPlayerMarker.resize(
                %xMap_playerPos - getWord(HUDGuiMapPlayerMarker.getExtent(), 0) / 2,
                %yMap_playerPos - getWord(HUDGuiMapPlayerMarker.getExtent(), 1) / 2,
                 getWord(HUDGuiMapPlayerMarker.getExtent(), 0),
                 getWord(HUDGuiMapPlayerMarker.getExtent(), 1)
                 );

if (%beginningPos !$= "NULL")
        HUDGuiMapBeginningMarker.resize(
         %xMap_beginningPos - getWord(HUDGuiMapBeginningMarker.getExtent(), 0) / 2,
        %yMap_beginningPos - getWord(HUDGuiMapBeginningMarker.getExtent(), 1) / 2,
        getWord(HUDGuiMapBeginningMarker.getExtent(), 0),
        getWord(HUDGuiMapBeginningMarker.getExtent(), 1)
        );

if (%targetPos !$= "NULL")
        HUDGuiMapTargetMarker.resize(
```

```
                        %xMap_targetPos - getWord(HUDGuiMapTargetMarker.getExtent(), 0) / 2,
                        %yMap_targetPos - getWord(HUDGuiMapTargetMarker.getExtent(), 1) / 2,
                        getWord(HUDGuiMapTargetMarker.getExtent(), 0),
                        getWord(HUDGuiMapTargetMarker.getExtent(), 1)
                        );

        // show the map
        HUDGuiMap.setVisible(true);
        HUDGuiMapPlayerMarker.setVisible(true);
        if (%beginningPos !$= "NULL")
                HUDGuiMapBeginningMarker.setVisible(true);
        if (%targetPos !$= "NULL")
                HUDGuiMapTargetMarker.setVisible(true);

}
```

Η *πυξίδα* ενημερώνεται επίσης με παρόμοιο τρόπο. Και πάλι ο πελάτης στέλνει περιοδικά μηνύματα στον εξυπηρετητή ζητώντας του να του στείλει την κατάλληλη πληροφορία που χρειάζεται για την πυξίδα. Στη συγκεκριμένη περίπτωση το μόνο που χρειάζεται να στείλει ο εξυπηρετητής είναι το διάνυσμα που δείχνει την κατεύθυνση προς την οποία είναι στραμμένο το αντικείμενο (forward vector) που χειρίζεται ο χρήστης (βλ. serverCmdRequestCompassInfo() @ /control/server/scripts/game.cs). Χρησιμοποιούμε το σταθερό σύστημα συντεταγμένων του κόσμου (world coordinate system, το οποίο στο Torque είναι right-handed, με τον θετικό ημιάξονα των z να ορίζει την κατεύθυνση προς τα πάνω – up-vector), και κάνουμε την παραδοχή ότι ο Βοράς αντιστοιχεί στα θετικά y (οπότε η Ανατολή αντιστοιχεί στα θετικά x). Τότε, στην πλευρά του πελάτη κάνουμε τα εξής: προβάλουμε το forward vector στο επίπεδο x-y (απλά "ρίχνοντας" τη z συνιστώσα), και στη συνέχεια το κανονικοποιούμε. Ελέγχουμε σε πια πλευρά του επιπέδου με κανονικό διάνυσμα (normal vector) (1,0,0) ανήκει το διάνυσμα που προέκυψε, διότι όταν μετά χρησιμοποιήσουμε εσωτερικό γινόμενο η πληροφορία αυτή θα χαθεί, και είναι απαραίτητη παρακάτω. Τώρα βρίσκουμε το εσωτερικό γινόμενο (dot product) του τροποποιημένου forward vector με το διάνυσμα (0,1,0) που καθορίζει τον Βορά, ουσιαστικά το συνημίτονο της γωνίας μεταξύ αυτών των διανυσμάτων αφού είναι και τα δύο κανονικοποιημένα (μοναδιαίου μέτρου). Από αυτό βρίσκουμε (arc cosine) τη γωνία σε ακτίνια (radians), η οποία ανήκει στο διάστημα [0, π]. Με βάση την πληροφορία που κρατήσαμε προηγουμένως για το σε ποια πλευρά του επιπέδου με κανονικό διάνυσμα (normal vector) (1,0,0) ανήκε το τροποποιημένο forward vector, βρίσκουμε την πραγματική γωνία στο διάστημα [0, 2π] (βλ. και κώδικα παρακάτω).

Έχουμε τώρα την πληροφορία για την κατεύθυνση του αντικειμένου που ελέγχει ο χρήστης υπό τη μορφή γωνίας με τον άξονα του Βορά. Στο HUD GUI, η πυξίδα είναι μία εικόνα / *λωρίδα* (HUDGuiCompassStrip) που έχει σε ισαπέχοντα σημεία τα σημεία του ορίζοντα:

Η λωρίδα ξεκινά από το Βορά και κάνοντας "κύκλο" κατά τη φορά των δεικτών του ρολογιού (clockwise) επιστρέφει στον Βορά. Στην πραγματικότητα βέβαια είναι μια ευθεία. Η ιδέα είναι να απεικονίσουμε (map) την πληροφορία που έχουμε υπό μορφή γωνίας σε αυτή την ευθεία, δηλ. απεικόνιση από το [0, 2π] στο [0, 1] όπου το 0 αντιστοιχεί στο αριστερό άκρο της λωρίδας και το 1 στο δεξί. Αυτό γίνεται απλά διαιρώντας τη γωνία με 2π. Τώρα, έχουμε έναν αριθμό μεταξύ 0 και 1 που δείχνει πού βρίσκεται η κατεύθυνση του αντικειμένου που ελέγχει ο χρήστης πάνω στη λωρίδα.

Για να παρουσιάσουμε αυτή τη θέση της λωρίδας στον χρήστη, χρησιμοποιούμε ένα *πλαίσιο* (HUDGuiCompassFrame) :

Στην εικόνα δεν φαίνεται το περίγραμμα του πλαισίου, γιατί απλώς δεν έχει περίγραμμα (ώστε να μην γίνεται αντιληπτή η ύπαρξή του στον χρήστη), και φαίνεται μόνο ένας δείκτης που θα δείχνει τη θέση της λωρίδας που βρήκαμε προηγουμένως. Το πλαίσιο μένει ακίνητο, αλλά η λωρίδα κινείται έτσι ώστε ο δείκτης να δείχνει πάντα τη σωστή θέση.

Τόσο η λωρίδα όσο και το πλαίσιο ενθυλακώνονται σε *ένα control που έχει σκοπό να αποκόπτει (clip) κομμάτια της λωρίδας* (HUDGuiCompass). Ο κώδικας έχει ως εξής (βλ. /control/client/gui/HUDGui.gui) :

```
// compass
new GuiControl(HUDGuiCompass)  // this control is used to restrict the visible part
                                // of the compass strip
{
        extent = "400 50";
        position = "312 100";
        profile = TransparentProfile;
        visible = true;

        new GuiChunkedBitmapCtrl(HUDGuiCompassStrip)
        {
                extent = "1600 50";  // larger than it's container, so parts of it will
                                     // be 'clipped' (which parts depends on it's position) !
                position = "0 0";
                profile = TransparentProfile;
                visible = true;

                bitmap = "./hudGuiCompassStrip.png";
                tile = false;
                useVariable = false;
        };

        new GuiChunkedBitmapCtrl(HUDGuiCompassFrame)  // have the frame
                                 // after the strip, so that it will be drawn over the strip
        {
                extent = "400 50";  // same extent as the 'clipping' control
                position = "0 0";
                profile = TransparentProfile;
                visible = true;

                bitmap = "./hudGuiCompassFrame.png";
                tile = false;
                useVariable = false;
        };
};
```

Όπως βλέπουμε η λωρίδα έχει μεγαλύτερο μήκος από το container της, συνεπώς μόνο ένα μέρος αυτής θα είναι ορατό κάθε στιγμή και όσο περισσεύει δεξιά και αριστερά αποκόπτεται.

Το container και το πλαίσιο (που καλύπτει όλο το container) μένουν ακίνητα, και η λωρίδα κινείται μέσα στο container (δίνοντας την εντύπωση ότι η πυξίδα "γυρίζει") με βάση την τιμή που βρήκαμε προηγουμένως. Συγκεκριμένα, η τιμή αυτή απεικονίζεται από το [0, 1] στο [0, πλάτος_λωρίδας – πλάτος_πλαισίου], και η λωρίδα μετακινείται προς τα αριστερά (πάντα) κατά ένα διάστημα ίσο με αυτή την τιμή (η μετακίνηση γίνεται σε σχέση με το container της λωρίδας).

Όταν στο [0, 1] έχουμε 0, το αριστερό άκρο (αρχή στον άξονα των x) της λωρίδας θα συμπίπτει με το αριστερό άκρο (αρχή στον άξονα των x) του container (συνεπώς και του πλαισίου). Η πυξίδα πρέπει να δείχνει το Βορά, συνεπώς όταν σχεδιάζουμε την εικόνα της λωρίδας έχουμε προβλέψει το πρώτο "N" να βρίσκεται σε απόσταση (πλάτος_πλαισίου / 2) από το αριστερό άκρο της λωρίδας, αφού ο δείκτης του πλαισίου βρίσκεται στο μέσο του.

Όταν στο [0, 1] έχουμε 1, η πυξίδα πάλι θα πρέπει να δείχνει το Βορά (το αντικείμενο που ελέγχει ο χρήστης έχει κάνει έναν πλήρη κύκλο). Αφού το 1 απεικονίζεται στο (πλάτος_λωρίδας – πλάτος_πλαισίου), το αριστερό άκρο (αρχή στον άξονα των x) της λωρίδας θα μετακινηθεί ακριβώς τόσο σε σχέση με το αριστερό άκρο (αρχή στον άξονα των x) του container. Αυτό σημαίνει ότι μέσα στο πλαίσιο εμφανίζεται το δεξί τμήμα της λωρίδας μήκους (πλάτος_πλαισίου) που δεν αποκόπτεται. Και πάλι, έχουμε προβλέψει το τελευταίο "N" να βρίσκεται σε απόσταση (πλάτος_πλαισίου / 2) από το δεξί άκρο της λωρίδας, αφού ο δείκτης του πλαισίου βρίσκεται στο μέσο του.

Ο κώδικας που αναλαμβάνει την όλη διαδικασία είναι ο εξής (βλ. /control/client/scripts/hudGui.cs) :

```
function clientCmdResponseCompassInfo(%forwardVector)  // forwardVector is the forward vector of
                                                       // this client's control object
{
        // NOTE:
        // all directions/vectors discussed here are expressed in the (FIXED) world coordinate system
        // (in Torque, a right-handed system, with positive-z defining the up-vector)
        // and not in the object coordinate system of an object, which moves with the object.

        // We choose North to be in the direction of the positive y-axis in world coordinates (so
        // positive x-axis defines East)
        // The idea here is to get the forward direction in the form of an angle in [0, 2π],
        // where directions are mapped to angles in a clockwise manner, as in:
        // 0 -> North, π/2 -> East, π -> South, 3π -> West, 2π -> North
        // Then, map the angle from [0, 2π] to [0, compass_strip_width - compass_frame_width],
        // where the compass strip used has "N" in compass_frame_width/2 (North) ,
        // ... "N" in compass_strip_width - compass_frame_width/2 (Notrh)
        // The reason for using compass_frame_width is to have the current direction in the center of
        // the compass frame,
        // rather than on the left of the compass frame (like an offset). Of course, the compass strip
        // image must be painted with this in mind.

        // first, drop the z component of the forwardVector. This effectively projects the
        // forwardVector on the x-y plane.
        // We care for (and MUST use) only the direction in the x-y plane
        %forwardVector = getWords(%forwardVector, 0, 1) SPC 0;

        // normalize the forwardVector, since vectorDot() console function requires normalized
        // vectors (AppendixA)
        %forwardVector = vectorNormalize(%forwardVector);

        // determine if the %forwardVector is on the positive halfspace defined by the plane with
        // normal (1, 0, 0)
        // or on the negative halfspace of this plane.
        // This is necessary since the dot product contains no information as to the position of one
        // vector relative to the other.
        %isInNegativeHalfspace = (vectorDot("1 0 0", %forwardVector) < 0) ? true : false;

        // find the cosine between the (projected on the x-y plane) forwardVector and vector (0, 1, 0)
        // (North)
        %dot = vectorDot(%forwardVector, "0 1 0");  // since vectors are normalized, %dot is actually
```

```
                                                                        // the cosine

        // find the angle between the (projected on the x-y plane) forwardVector and vector (0, 1, 0)
        // (North)
        %angleRad = mAcos(%dot);  // angle in radians, in [0, π]

        // adjust so we always end up with an angle in [0, 2π]
        if (%isInNegativeHalfspace)
                %angleRad = 2*3.14159 - %angleRad;

        // 1D -> 1D mapping, by analogy (same as for the map above, only this is done
        // straightforward here.
        // This is a special case, since minimums are 0)
        // map angle from [0, 2π] to [0, 1]
        // (of course, apart from the "mapping" above, it's obvious a simple division will do it)
        %mappedAngle = %angleRad / (2*3.14159);  // 3.14159 is used by Torque as π (AppendixA)
        // re-map from [0, 1] to [0, compass_strip_width - compass_frame_width]
        %stripPosition = %mappedAngle *
                                        ( getWord(HUDGuiCompassStrip.getExtent(), 0) –
                                          getWord(HUDGuiCompassFrame.getExtent(), 0) );

        // reposition the compass strip
        HUDGuiCompassStrip.resize( - %stripPosition,   // note the "-" (move strip to the left)
                                        getWord(HUDGuiCompassStrip.getPosition(), 1),
                                        getWord(HUDGuiCompassStrip.getExtent(), 0),
                                          getWord(HUDGuiCompassStrip.getExtent(), 1)
                                );
}
```

Στην *περιοχή μηνυμάτων* τυπώνονται μηνύματα με πληροφορίες για τον χρήστη, που στέλνει ο εξυπηρετητής. Ο χρωματισμός υποδεικνύει το είδος του μηνύματος (κόκκινο **ό** μηνύματα μεταξύ χρηστών, μαύρο **ό** μηνύματα για ανώμαλες καταστάσεις, μπλε **ό** μηνύματα σχετικά με αντικείμενα, π.χ. όταν ο χρήστης παίρνει ή ρίχνει κάποιο αντικείμενο, μοβ **ό** μηνύματα σχετικά με τις αποστολές). Ο χρήστης μπορεί να χρησιμοποιήσει τα scroll bars για να δει παλαιότερα μηνύματα (για να γίνει αυτό πρέπει πρώτα να πατήσει "F2" ώστε το ποντίκι να χρησιμοποιείται όπως στα μενού, βλ. συναρτήσεις mouseGuiLook() και mouseFreeLook() @ /control/client/scripts/hudGui.cs - με "F1" το ποντίκι χρησιμοποιείται και πάλι για περιστροφή της κάμερας). Πατώντας "Enter" εμφανίζεται μία μικρή περιοχή κάτω από την περιοχή μηνυμάτων, όπου διοχετεύεται η είσοδος του χρήστη από το πληκτρολόγιο, μέχρι αυτός να πατήσει ξανά "Enter" οπότε και στέλνονται αυτά που έγραψε στον εξυπηρετητή, και αυτός με τη σειρά του τα στέλνει σε όσους πελάτες είναι συνδεδεμένοι μαζί του (*συνομιλία μεταξύ χρηστών – chat*).

Με τα πλήκτρα "R" και "F" *ο χρήστης επιλέγει το αντικείμενο προς χρήση (equip) μεταξύ των αντικειμένων που έχουν γίνει mount στον avatar του* (δύο το πολύ στο demo, όσα και τα slots που χρησιμοποιούμε -εδώ slot 0, slot 1-. Το Torque παρέχει συνολικά οκτώ mount slots για κάθε αντικείμενο της κλάσης ShapeBase). Το "R" επιλέγει το επόμενο αντικείμενο, το "F" το προηγούμενο. Η πληροφορία για το χρησιμοποιούμενο αντικείμενο εμφανίζεται πάνω δεξιά στο HUD.

Σχ.15 Μια άποψη του HUD (με το χάρτη κλειστό)

Σχ.16 Μια άποψη του HUD (με το χάρτη ανοιχτό)

(ο χάρτης είναι ημιδιαφανής και το υπόλοιπο HUD φαίνεται από "πίσω")

11.Σύστημα μάχης (Combat system)

Σχετικά αρχεία:
/control/client/gui/CombatGui.gui
/control/client/scripts/combat.cs
/control/server/scripts/combat.cs
/control/server/scripts/combatActions.cs
/control/server/datablocks/enemies.cs
/control/server/scripts/damageFloaters.cs

Υπάρχουν φορές που η αλληλεπίδραση με έναν NPC δεν καταλήγει σε διάλογο, αλλά σε μάχη. Αν για παράδειγμα ο avatar του χρήστη βρεθεί πολύ κοντά με ένα εχθρικό NPC (*bot* στο εξής), θα ενεργοποιηθεί το σύστημα μάχης.

Το σύστημα μάχης είναι με διαφορά το πιο περίπλοκο κομμάτι αυτού του demo. Αυτό έχει και σαν παρενέργεια να είναι και λιγότερο καλογραμμένο, και γενικά αρκετά πράγματα εδώ θα μπορούσαν να είναι καλύτερα. Μία πλήρης περιγραφή του κώδικα θα ήταν μάλλον κουραστική και άσκοπη εδώ, ιδιαίτερα αφού (όπως πάντα) ο κώδικας που περιλαμβάνεται στο CD έχει αρκετά αναλυτικά σχόλια.

Ένας λόγος για την πολυπλοκότητα του συστήματος μάχης είναι ότι ακολουθείται μία σχεδίαση εντελώς διαφορετική από αυτή που "προτείνει" το Torque. Το Torque, με την ιεραρχία των κλάσεων / datablocks κλπ. που ενσωματώνει και τη λειτουργικότητα που αυτές προσφέρουν, έχει ένα σαφή προσανατολισμό προς συστήματα μάχης πραγματικού χρόνου (real-time). Αντίθετα, στο demo αυτό προτιμήθηκε ένα σύστημα μάχης βασισμένο σε γύρους (*turn-based*).

Το τμήμα αυτό επιχειρεί μία γενική περιγραφή του συστήματος μάχης και του σκεπτικού πίσω από τις λειτουργίες του, με σκοπό να αποτελέσει αφετηρία για την ανάγνωση (και βελτίωση) του κώδικα.

Όταν ο εξυπηρετητής πρέπει να ενεργοποιήσει το σύστημα μάχης για κάποιον πελάτη, καλεί τη συνάρτηση combat::enterCombatMode() (@ /control/server/scripts/combat.cs). Αυτή αποτελεί το σημείο εισόδου στο σύστημα μάχης, και κάνει τις απαραίτητες αρχικοποιήσεις. Δημιουργεί ένα αντικείμενο της κλάσης ScriptObject, το οποίο ενθυλακώνει όλες τις πληροφορίες που πρέπει να διατηρεί ο εξυπηρετητής σχετικά με τη συγκεκριμένη μάχη:

Πληροφορίες που ενθυλακώνει το "αντικείμενο μάχης" ("combat object")

- τα *IDs (handles) των αντικειμένων που συμμετέχουν στη μάχη*, σε αυτή τη φάση πρόκειται για ένα χρήστη και ένα bot (μαχητές - combatants).
- τον πίνακα *pendingDamageTable*, όπου κρατάμε την πληροφορία για τη ζημιά (damage) που εκκρεμεί για τους μαχητές, πάντα για τον τρέχον γύρο. Κάθε γραμμή έχει δύο στήλες:
  1. το ID του μαχητή για τον οποίο εκκρεμεί η ζημιά
  2. το ποσό της ζημιάς που εκκρεμεί
- τον πίνακα *activeEffectsTable*, όπου κρατάμε την πληροφορία για τα διάφορα effects που είναι ενεργά σε αυτή τη μάχη (π.χ. ένας μαχητής μπορεί να έχει "ρίξει" στον άλλο μια καταιγίδα, και αυτή να διαρκεί αρκετούς γύρους, οπότε θα καταγράφεται σε αυτό τον πίνακα ώστε να ξέρει ο εξυπηρετητής ότι πρέπει να αποδώσει ζημιά στον στόχο της καταιγίδας στην αρχή κάθε γύρου κατά τον οποίο είναι η καταιγίδα ενεργή). Κάθε γραμμή έχει επτά στήλες:
  1. το υπόλοιπο της διάρκειας του effect (σε γύρους)

2. μία λίστα με τα IDs των αντικειμένων κλάσεων που δημιουργήθηκαν για την οπτική / ακουστική απόδοση του effect, ώστε να μπορούμε να τα σβήσουμε (delete()) όταν τελειώσει η διάρκειά του
3. το ID του μαχητή που ξεκίνησε αυτό το effect (caster)
4. μία λίστα με τα IDs των μαχητών για τους οποίους είναι ενεργό αυτό το effect (targets)
5. το όνομα της συνάρτησης που θα κληθεί στη αρχή κάθε γύρου κατά τη διάρκεια του οποίου αυτό το effect είναι ενεργό (π.χ. για την περίπτωση της καταιγίδας, η συνάρτηση αυτή αναλαμβάνει να αποδίδει ζημιά σε όλους τους στόχους της καταιγίδας)
6. το όνομα της συνάρτησης που θα καλείται όταν λήξει η διάρκεια του effect, και αναλαμβάνει θέματα όπως το σβήσιμο των αντικειμένων της λίστας (2)
7. το όνομα της συνάρτησης που θα καλείται όταν συμβεί κάποιο γεγονός που "ενεργοποιεί" το effect (π.χ. μια ασπίδα που ενεργοποιείται όταν αυτός που την έχει δεχθεί ζημιά, και ενδεχομένως μειώνει το ποσό της ζημιάς)

Στη συνέχεια της συνάρτησης combat::enterCombatMode(), ο εξυπηρετητής θέτει σαν αντικείμενο ελέγχου (control object) του πελάτη ένα αντικείμενο της κλάσης PathCamera, και εκκινεί το μηχανισμό περιστροφής της γύρω από την περιοχή όπου εκτυλίσσεται η μάχη. Ο πελάτης λοιπόν χάνει τον έλεγχο του avatar του χρήστη, και ο χρήστης παρακολουθεί την εξέλιξη της μάχης "πανοραμικά". Μετά ο εξυπηρετητής στέλνει μήνυμα στον πελάτη ειδοποιώντας τον να ανοίξει το combat GUI, ελέγχει ποιος μαχητής θα έχει τον πρώτο γύρο, και αρχίζει την μάχη.

Εδώ κάνουμε μια παρένθεση για το combat GUI στην πλευρά του πελάτη (αρχεία /control/client/gui/CombatGui.gui και /control/client/scripts/combat.cs). Όταν αυτό ενεργοποιείται, κάνει μη ορατά τα διάφορα controls του HUD και γενικά μοιάζει αρκετά με το dialog GUI (βλ. ενότητα "Σύστημα διαλόγου (Dialog system)"), όπου ο χρήστης επιλέγει μεταξύ των διαθέσιμων επιλογών με τα πλήκτρα 1-9. Εδώ οι επιλογές είναι δομημένες σε δύο στρώματα (*layers*), π.χ. στο πρώτο στρώμα ο χρήστης διαλέγει "1. Offensive" και στο δεύτερο "4. Atmosphere" για την "καταιγίδα". Η συνολική επιλογή του χρήστη αποτελείται από το συνδυασμό των επιλογών του σε κάθε στρώμα, στο παραπάνω παράδειγμα "1 4", και στέλνεται με μήνυμα στον εξυπηρετητή. Στο combat GUI έχουμε επίσης τέσσερις μπάρες, δύο για τα στατιστικά helth και mana του avatar του χρήστη και αντίστοιχα για το bot, οι οποίες είναι αντικείμενα της κλάσης GuiProgressCtrl και όχι της κλάσης GuiHealthBarHud όπως ίσως θα περίμενε κανείς. Υπάρχουν διάφοροι λόγοι για αυτή την επιλογή: $1^{ον}$ τα αντικείμενα της κλάσης GuiHealthBarHud αυτόματα δείχνουν το health και energy / mana του αντικειμένου που ελέγχει ο πελάτης (control object), και όπως είδαμε κατά τη διάρκεια της μάχης αυτό είναι ένα αντικείμενο της κλάσης PahCamera και όχι ο avatar του χρήστη. $2^{ον}$, για τον ίδιο λόγο δεν θα ήταν δυνατό να απεικονιστούν τα στατιστικά του bot, ή οποιουδήποτε άλλου αντικειμένου εκτός από αυτό που ελέγχει ο πελάτης. Αυτό σημαίνει ότι η ανανέωση κάθε μπάρας είναι υπόθεση του script κώδικά μας, με βάση στοιχεία που θα πρέπει να στέλνει ο εξυπηρετητής στον πελάτη, και δεν γίνεται αυτόματα από την μηχανή όπως στην περίπτωση αντικειμένων της κλάσης GuiHealthBarHud.

Συνεχίζοντας στην πλευρά του εξυπηρετητή, εάν ήταν ο γύρος του χρήστη, και όταν αυτός κάνει τις επιλογές του στο combat GUI, ο πελάτης στέλνει μήνυμα στον εξυπηρετητή το οποίο μεταφέρει τις επιλογές αυτές. Ο εξυπηρετητής πιάνει το μήνυμα στη συνάρτηση serverCmdCombatModeRouteTurnChoices(), και στη συνέχεια καλεί την συνάρτηση combat::handlePlayerChoiceLayers() για τον

χειρισμό των επιλογών του χρήστη. Οι διαθέσιμες επιλογές αντιστοιχούνται σε διαθέσιμες *ενέργειες μάχης (combat actions)*, και καλείται όποια από αυτές είναι κατάλληλη. Οι ενέργειες μάχης ορίζονται στο αρχείο /control/server/scripts/combatActions.cs. Το πρώτο μισό αυτού του αρχείου περιέχει τη δημιουργία των κατάλληλων αντικειμένων datablocks που θα χρησιμοποιηθούν από τα αντικείμενα κλάσεων τα οποία αναπαριστούν το οπτικό / ακουστικό μέρος του κάθε effect. Π.χ. για την "καταιγίδα", για το οπτικό μέρος δημιουργούμε ένα αντικείμενο του datablock LightningData, και αντικείμενα των datablocks AudioDescription και AudioProfile για το ηχητικό μέρος του effect. Να σημειώσουμε ότι η παρουσίαση των effects (είτε το οπτικό είτε το ακουστικό τμήμα) δεν έχει καμία επίδραση στο αποτέλεσμα των ενεργειών μάχης. Το αποτέλεσμα αυτό θα είναι το ίδιο, ανεξάρτητα αν ο χρήστης μπορεί να δει / ακούσει το effect που υποτίθεται ότι προκαλεί το αποτέλεσμα.

Το δεύτερο μισό του αρχείου /control/server/scripts/combatActions.cs περιέχει τις συναρτήσεις χειρισμού των διαφόρων ενεργειών μάχης (για την "καταιγίδα", αυτές είναι οι combatAction::callLightning(), combatAction_callLightningExpiration(), combatAction_callLightningTurnAction()). Γενικά, υπάρχει μια κεντρική συνάρτηση που αντιπροσωπεύει την ενέργεια μάχης, και είναι αυτή που καλείται από τη συνάρτηση combat::handlePlayerChoiceLayers() του αρχείου /control/server/scripts/combat.cs όταν ο χρήστης ζητήσει την αντίστοιχη ενέργεια (για την "καταιγίδα" είναι η combatAction::callLightning()). Η κεντρική συνάρτηση αναλαμβάνει να ελέγξει κατ' αρχήν αν ο πελάτης μπορεί όντως να χρησιμοποιήσει τη συγκεκριμένη ενέργεια μάχης (π.χ. αν έχει αρκετό mana). Αν όλα είναι εντάξει, ενημερώνει τον πίνακα pendingDamageTable του combat object που είδαμε παραπάνω, προσθέτοντας μια γραμμή με το αντικείμενο-στόχο και τη ζημιά που εκκρεμεί για αυτό στον τρέχον γύρο. Έπειτα δημιουργεί τα αντικείμενα για την οπτικοακουστική παρουσίαση του effect, και ενημερώνει τον πίνακα activeEffectsTable του combat object, προσθέτει δηλαδή σε αυτόν μία γραμμή που αντιπροσωπεύει το νέο effect που δημιουργήθηκε στη μάχη και τα χαρακτηριστικά του (διάρκεια κλπ.).

Τελικά, προγραμματίζεται (schedule) να κληθεί η συνάρτηση που αναλαμβάνει το τέλος του γύρου, ύστερα από χρονικό διάστημα ικανό για την ολοκλήρωση της οπτικοακουστικής παρουσίασης του effect της ενέργειας μάχης. Η συνάρτηση αυτή είναι η combat_endOfTurn() (βλ. /control/server/scripts/combat.cs), και καλεί την συνάρτηση combat::endOfTurnDamageDelivery() η οποία αποδίδει τη ζημιά που εκκρεμεί για αυτό τον γύρο με βάση τον πίνακα pendingDamageTable.

Στη συνέχεια, αφού η απόδοση ζημιάς έχει ολοκληρωθεί, καλείται η συνάρτηση combat_endOfTurn_AUX(). Εδώ κάνουμε κάποιους ελέγχους (π.χ. ελέγχουμε αν κάποιο ενεργό effect του πίνακα activeEffectsTable έληξε σε αυτό τον γύρο), και στη συνέχεια, με βάση με το ποιος γύρος τέλειωσε (του χρήστη ή του bot), δρούμε ανάλογα.

Γενικά, αν τέλειωσε ο γύρος του bot, και ο avatar του χρήστη και το bot συνεχίζουν να είναι ζωντανά, στέλνουμε μήνυμα στον πελάτη ενημερώνοντάς τον ότι άρχισε νέος δικός του γύρος, και προγραμματίζουμε μια κλήση στη συνάρτηση combat_effectsTurnAction(), η οποία διατρέχει τον πίνακα activeEffectsTable και για κάθε ενεργό effect καλεί την κατάλληλη συνάρτηση το όνομα της οποίας είναι αποθηκευμένο στην πέμπτη στήλη του ίδιου πίνακα (βλ. περιγραφή του πίνακα παραπάνω).

Παρόμοια, αν τελείωσε ο γύρος του χρήστη και το bot είναι ακόμα ζωντανό, είναι η σειρά του bot να δράσει και αρχίζουμε τον καινούριο γύρο καλώντας την

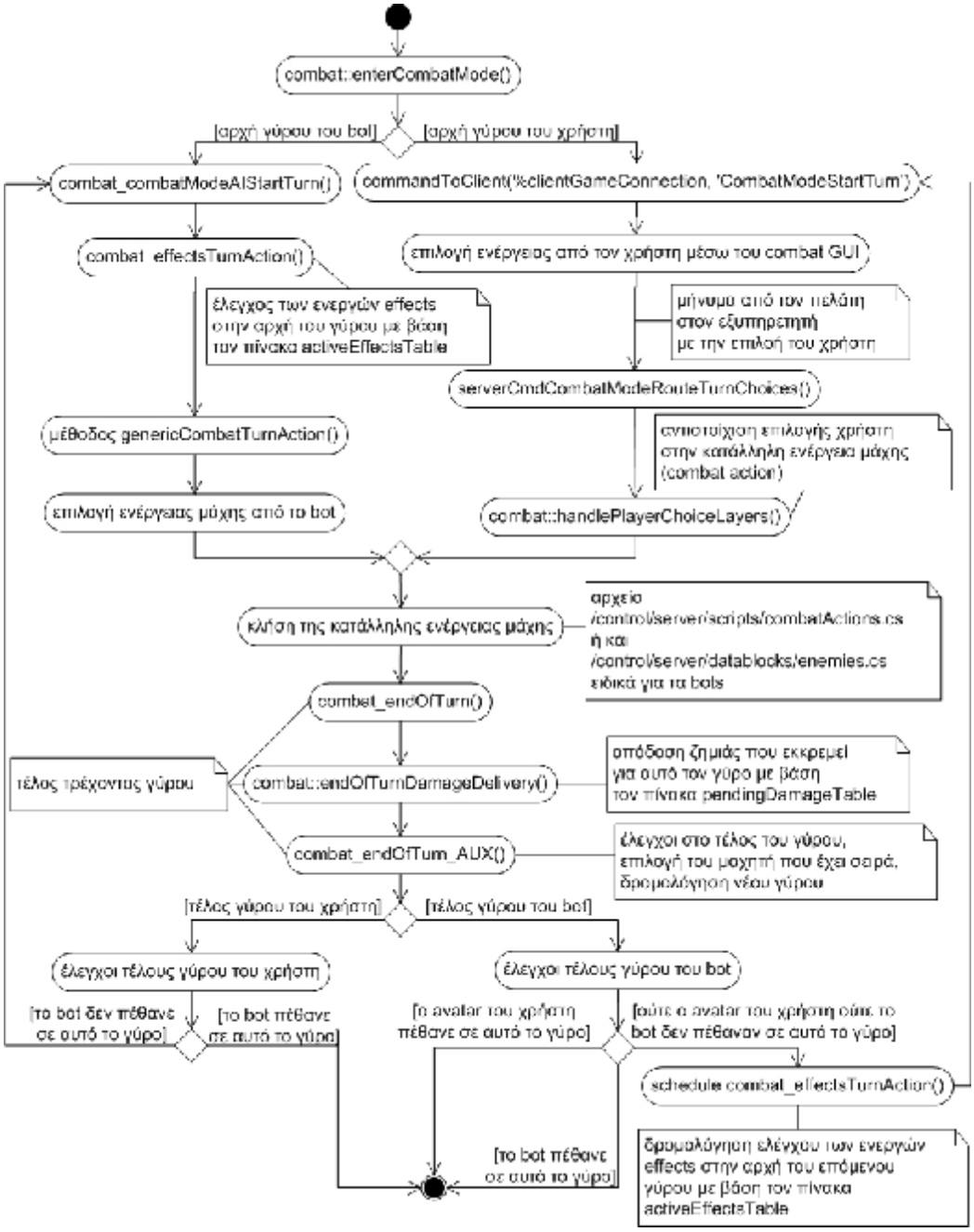
συνάρτηση combat_combatModeAIStartTurn(), η οποία θα καλέσει και την συνάρτηση combat_effectsTurnAction() για τον έλεγχο των ενεργών effects στην αρχή του νέου γύρου, και στη συνέχεια καλούμε τη μέθοδο *genericCombatTurnAction()* του αντικειμένου datablock που αντιστοιχεί στο bot, αφήνοντας έτσι το κάθε bot να αποφασίσει μόνο του τι ενέργειες θα εκτελέσει στον δικό του γύρο (βλ. /control/server/datablocks/enemies.cs). Κάθε αντικείμενο λοιπόν που θέλουμε να συμπεριφέρεται ως bot πρέπει να ορίζει αυτή τη συνάρτηση στο χώρο ονομάτων του αντικειμένου του datablock που του αντιστοιχεί. Όσον αφορά τις ενέργειες μάχης που είναι διαθέσιμες σε κάθε bot, αυτές είναι όλες οι ενέργειες που ορίζονται στο αρχείο /control/server/scripts/combatActions.cs (όσες δηλ. είναι διαθέσιμες και στον χρήστη), και επιπλέον το κάθε bot μπορεί να ορίζει ιδιαίτερες ενέργειες μάχης, διαθέσιμες μόνο για τον εαυτό του, στο χώρο ονομάτων του datablock που του αντιστοιχεί.

Ύστερα από όλα αυτά, κάνουμε τις εξής *παρατηρήσεις*:

- ένας "γύρος" αποτελείται από μια ενέργεια μάχης ξεχωριστά για κάθε μαχητή, *όχι* από μια ακολουθία ενεργειών μάχης στην οποία ο κάθε μαχητής συμμετέχει μία (και μόνο) φορά όπως ίσως θα περίμενε κανείς. Δηλ. σε έναν γύρο ενεργεί π.χ. το bot, η επακόλουθη ενέργεια του χρήστη είναι ένας καινούριος γύρος κλπ.
- η ζημιά που προστίθεται στον πίνακα *pendingDamageTable* προέρχεται μόνο από νέες ενέργειες μάχης στον τρέχον γύρο, και αποδίδεται *στο τέλος κάθε γύρου* (συνάρτηση combat::endOfTurnDamageDelivery()). Μετά, το μήκος του πίνακα μηδενίζεται ώστε στον επόμενο γύρο να ξεκινήσει πάλι "κενός".
- αντίθετα, ο πίνακας *combat_effectsTurnAction* ελέγχεται *στην αρχή κάθε γύρου*, συνεπώς η ζημιά που προέρχεται από effects που δημιουργήθηκαν από ενέργειες μάχης προηγούμενων γύρων και είναι ακόμα ενεργά, αποδίδεται στην αρχή κάθε γύρου.
- η "ζημιά" (damage) μπορεί να είναι τόσο *θετική* όσο και *αρνητική*, στη δεύτερη περίπτωση "γιατρεύοντας" το στόχο (target).

Σχετικά με το αρχείο /control/server/scripts/damageFloaters.cs, αυτό έχει να κάνει με την παρουσίαση της ζημιάς που εφαρμόζεται στους μαχητές. Η ζημιά εμφανίζεται σαν μια εικόνα / αριθμός που αιωρείται (floater) προσωρινά πάνω από το αντίστοιχο αντικείμενο. Το αρχείο φαίνεται μεγάλο, αλλά περιέχει κυρίως επαναλαμβανόμενη πληροφορία (50 floaters, έναν για κάθε αριθμό από το 1 έως το 50, τόσο για θετική όσο για αρνητική ζημιά. Αυτό σημαίνει ότι για ζημιά με απόλυτη τιμή μεγαλύτερη του 50 θα πρέπει κανείς να προσθέσει τα ανάλογα αντικείμενα datablocks και φυσικά την κατάλληλη εικόνα στον φάκελο /control/data/special/, αλλιώς η ζημιά δεν θα εμφανίζεται οπτικά στον χρήστη).

Ένα γενικό διάγραμμα δραστηριότητας (UML activity diagram) για το σύστημα μάχης ακολουθεί:



Σχ.17 Διάγραμμα δραστηριότητας για το σύστημα μάχης

Μια άποψη του συστήματος μάχης :

Σχ.18 Μια άποψη του συστήματος μάχης

(είναι ο γύρος του bot -dragon- και επέλεξε σαν ενέργεια μάχης το "DeathHand", το οποίο είναι στιγμιαίο, και το οπτικό μέρος της παρουσίασης του οποίου είναι το χέρι που βλέπουμε πάνω από το avatar του χρήστη. Ακόμα, ο χρήστης είχε επιλέξει στον προηγούμενο γύρο σαν ενέργεια μάχης το "HealingMist", το οποίο διαρκεί 10 γύρους, και το οπτικό μέρος της παρουσίασης του effect αυτής της ενέργειας μάχης είναι το μπλε σύστημα σωματιδίων -particle system- που αιωρούνται γύρω από το avatar του χρήστη.)

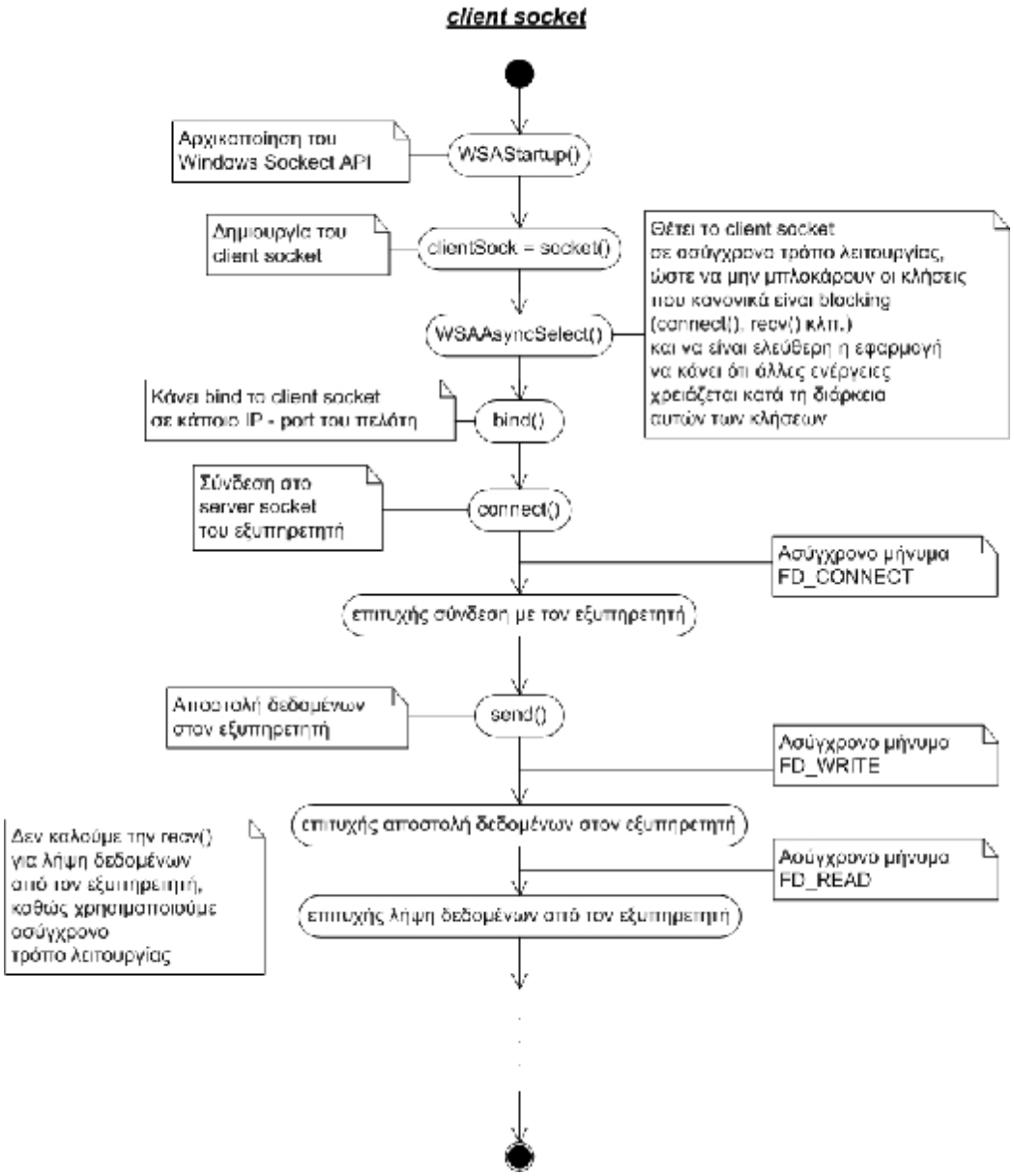
12. Απομακρυσμένος έλεγχος με sockets

Σχετικά αρχεία:
Πελάτης (C++ και Win32):
client/resource.h
client/client.rc
client/main.h
client/main.cpp
(συνολικά το project του VisualStudio είναι client/client.sln)

Εξυπηρετητής (TorqueScript):
/control/server/scripts/extra.cs

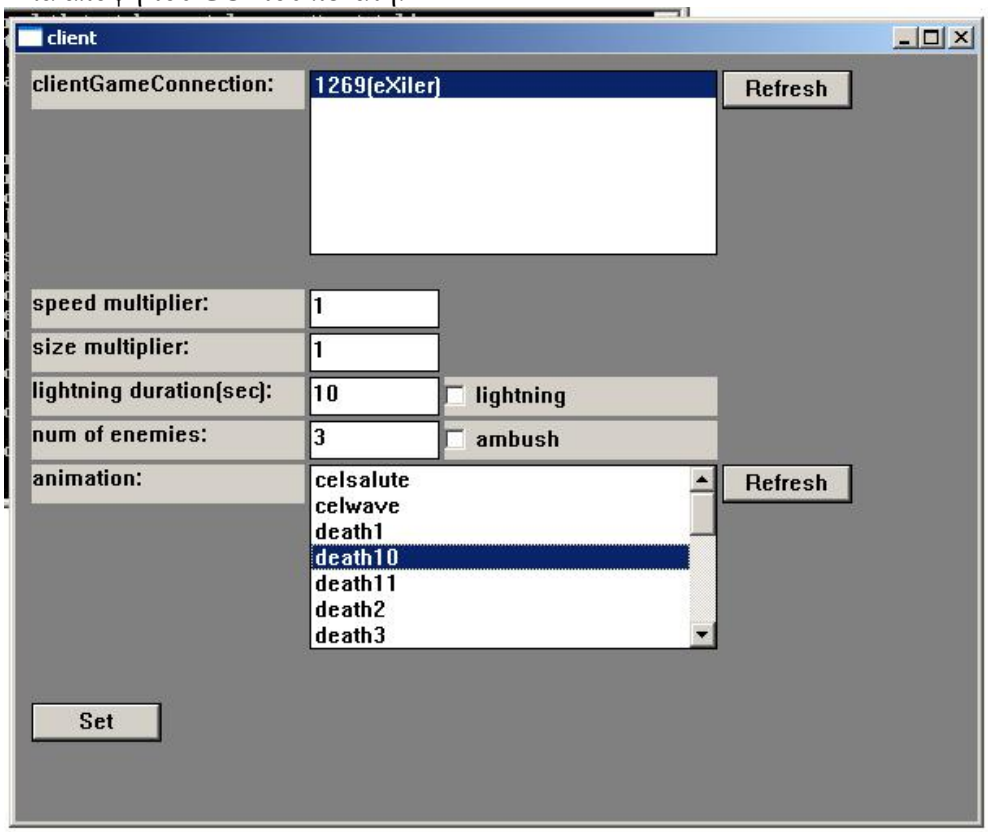
Σκοπός μας εδώ είναι να μπορούμε να επηρεάσουμε τους χρήστες ή τον κόσμο γενικότερα από κάποιον απομακρυσμένο υπολογιστή του δικτύου. Αυτό που κάνουμε είναι να έχουμε ένα πρόγραμμα *πελάτη (client* – δεν έχει σχέση με τους γνωστούς πελάτες του Torque που βλέπαμε μέχρι τώρα, και τους οποίους για αποφυγή σύγχυσης θα ονομάζουμε *Torque πελάτες* σε αυτή την ενότητα) το οποίο με το μηχανισμό των sockets συνδέεται στον αντίστοιχο *εξυπηρετητή (server* – εδώ είναι ενσωματωμένος στον γνωστό εξυπηρετητή του Torque, τον οποίο για αποφυγή σύγχυσης θα ονομάζουμε *Torque εξυπηρετητή* σε αυτή την ενότητα). Ο πελάτης παρέχει ένα GUI στον χρήστη του, μέσω του οποίου μπορεί αυτός να κάνει διάφορες επιλογές, π.χ. να αλλάξει την ταχύτητα του avatar κάποιου Torque πελάτη. Οι επιλογές αυτές στέλνονται στον εξυπηρετητή, ο οποίος τις μεταφράζει σε κατάλληλες ενέργειες που πρέπει να εκτελέσει ο Torque εξυπηρετητής.

Ο πελάτης είναι γραμμένος σε C++ και είναι ένα πρόγραμμα για το λειτουργικό σύστημα Windows (χρησιμοποιεί το Win32 API). Χειρίζεται το GUI και δημιουργεί ένα client socket για *TCP* σύνδεση στον εξυπηρετητή. Ακολουθεί ένα διάγραμμα της συνηθισμένης διαδικασίας για τη δημιουργία ενός client socket, με κάποιες τροποποιήσεις για τα Windows (ανώμαλες καταστάσεις δεν παρουσιάζονται στον διάγραμμα):



Σχ.19 Σχηματική αναπαράσταση χειρισμού client socket για τον απομακρυσμένο έλεγχο με sockets

Μια άποψη του GUI του πελάτη:



Σχ.20 Μια άποψη του GUI του πελάτη για τον απομακρυσμένο έλεγχο με sockets

Όταν ο χρήστης πατήσει το κουμπί "Set", ο πελάτης ενθυλακώνει όλες τις επιλογές του χρήστη σε ένα κατάλληλα μορφοποιημένο αλφαριθμητικό (string), που έχει τη μορφή χαρακτηριστικό – τιμή (attribute – value pair), και είναι το εξής:

```
"clientGameConnection %u "
"speed_multiplier %f "
"size_multiplier %f "
"lightning %i "
"lightning_duration %f "
"ambush %i "
"num_of_enemies %i "
"animation %s"
```

Ο εξυπηρετητής είναι γραμμένος σε TorqueScript. Δημιουργούμε ένα αντικείμενο της κλάσης TCPObject, και του λέμε να ακούει (listen()) σε κάποιο συγκεκριμένο port. Έτσι έχουμε ένα server socket που ακούει για εισερχόμενες συνδέσεις από client sockets. Όταν "ακούσουμε" μια τέτοια σύνδεση (callback μέθοδος onConnectRequest()), την αποδεχόμαστε και δημιουργούμε ένα νέο socket για να την εξυπηρετήσουμε. Δεχόμαστε δεδομένα από το client socket στην callback μέθοδο onLine(), και στέλνουμε δεδομένα στο client socket με τη μέθοδο send(). Η callback μέθοδος onLine() δέχεται τα δεδομένα *γραμμή – γραμμή*, και αυτός είναι ο λόγος που το string που στέλνει ο πελάτης είναι όπως φαίνεται παραπάνω, με ένα ζευγάρι χαρακτηριστικού – τιμής ανά γραμμή.

Συνολικά, ο κώδικας του εξυπηρετητή έχει ως εξής:

```
new TCPObject(serverSock);

// setup a 'server' socket that listens for incoming connections (accepts one at a time)
function extra::externalSocketMonitor()
{
	// setup server
	serverSock.listen(2222);
}

// callbacks

// Called when a client socket attempts a connection, before the connection is accepted.
function TCPObject::onConnectRequest(%this, %address, %id)
{
	// accept the client socket connection
	new TCPObject(clientSock, %id);  // %id is important

	return "";
}

// NOTE:
// this callback actually gets the info send from the client socket, line-by-line.
// If multiple lines are sent in a single send() from the client socket,
// this callback will be called for each one of them separately.
function TCPObject::onLine(%this, %line)
{
	if (%line $= "")
		return;

	// parse the formatted string sent by the client ('attribute-value' format)
	%tokenString = %line;
	// default values
	%clientGameConnection = 0;
	%speed_multiplier = 1;
	%size_multiplier = 1;
	%lightning = 0;
	%lightning_duration = 10;
	%ambush = 0;
	%num_of_enemies = 3;
	%animation = "";
	while (%tokenString !$= "")
	{
		// tokens should always be in attribute-value pairs
		%tokenString = nextToken(%tokenString, "attribute", " ");
		%tokenString = nextToken(%tokenString, "value", " ");

		// it seems that "break" is ignored by the switch$ itself,
		// and so it breaks whatever is outside the switch$,
		// *** NO "break;" IN SWITCH$ ***
		switch$ (%attribute)
		{
			case "refresh":
				extra::refreshList(%value);
				return;  // do no more
			case "clientGameConnection":
				%clientGameConnection = %value;
			case "speed_multiplier":
				%speed_multiplier = %value;
			case "size_multiplier":
```

```
                                        %size_multiplier = %value;
                                case "lightning":
                                        %lightning = %value;
                                case "lightning_duration":
                                        %lightning_duration = %value;
                                case "ambush":
                                        %ambush = %value;
                                case "num_of_enemies":
                                        %num_of_enemies = %value;
                                case "animation":
                                        %animation = %value;
                        }
                }

                if (%clientGameConnection == 0)
                        return;

                // put changes in effect

                %player = %clientGameConnection.player;
                // speed
                %player.external_speed_multiplier = %speed_multiplier;
                %player.getDatablock().runForce = %player.original_runForce * %speed_multiplier;
                %player.getDatablock().maxBackwardSpeed = %player.original_maxBackwardSpeed *
                                                                                        %speed_multiplier;
                %player.getDatablock().maxForwardSpeed = %player.original_maxForwardSpeed *
                                                                                        %speed_multiplier;
                %player.getDatablock().maxSideSpeed = %player.original_maxSideSpeed *
                                                                                        %speed_multiplier;
                // size
                %player.external_size_multiplier = %size_multiplier;
                %player.setScale(vectorScale(%player.original_scale, %size_multiplier));
                // lightning
                if (%lightning != 0)
                        extra::lightning(%clientGameConnection, %lightning_duration);
                // ambush
                if (%ambush != 0)
                        extra::ambush(%clientGameConnection, %num_of_enemies);
                // animation
                if (%animation !$= "")
                {
                        // not using playThread(). No merge animations requiered here,
                        // and setActionThread with hold=false will automatically revert to
                        // previous state after animation is done (for non-cyclic animations at least).
                        %player.setActionThread(%animation, false);
                }
}

function extra::refreshList(%value)
{
        switch$ (%value)
        {
                case "clientGameConnection_list":
                        extra::refreshClientGameConnectionList();
                case "animation_list":
                        extra::refreshAnimationList();
        }

        return;
}
```

```
function extra::refreshClientGameConnectionList()
{
	// create list
	%list = "clientGameConnection_list" @ " ";  // indicating to the client socket which list it
	                                            // receives

	for (%clientIndex = 0; %clientIndex < ClientGroup.getCount(); %clientIndex++)
	{
		%clientGameConnection = ClientGroup.getObject(%clientIndex);
		%list = %list @ %clientGameConnection @ "("
			@ %clientGameConnection.player.name @ ")" @ " ";
	}

	// 'finalize' list
	%list = %list @ "end_list";

	// send list to the client socket
	clientSock.send(%list);
}

function extra::refreshAnimationList()
{
	// create list
	%list = "animation_list" @ " ";  // indicating to the client socket which list it receives

	// manually ...
	%list = %list @ "death1" @ " "
	@ "death2" @ " "
	@ "death3" @ " "
	@ "death3" @ " "
	@ "death4" @ " "
	@ "death5" @ " "
	@ "death6" @ " "
	@ "death7" @ " "
	@ "death8" @ " "
	@ "death9" @ " "
	@ "death10" @ " "
	@ "death11" @ " "
	@ "looksn" @ " "
	@ "lookms" @ " "
	@ "scoutroot" @ " "
	@ "headside" @ " "
	@ "light_recoil" @ " "
	@ "celsalute" @ " "
	@ "celwave" @ " "
	@ "standjump" @ " "
	@ "looknw" @ " ";

	// 'finalize' list
	%list = %list @ "end_list";

	// send list to the client socket
	clientSock.send(%list);
}
```

13. *Γενικές αρχές σχεδίασης*

Επιλογή κλάσης από την οποία δημιουργεί ο εξυπηρετητής τα αντικείμενα που αντιπροσωπεύουν τους πελάτες στον κόσμο (avatars)

Τυπικά η κλάση που χρησιμοποιείται για τους avatars είναι η Player, ενώ για τα bots χρησιμοποιείται η κλάση AIPlayer που προσφέρει μερικές μεθόδους χρήσιμες για την δημιουργία script τεχνητής νοημοσύνης (AI scripts). Οι μέθοδοι αυτές όμως μπορεί να φανούν χρήσιμες και για τον προσωρινό χειρισμό ενός avatar από τον εξυπηρετητή. Για παράδειγμα, σε μία in-game cutscene θα μπορούσε ο εξυπηρετητής να κάνει έναν πελάτη να σταματήσει να ελέγχει τον avatar του (και να ελέγχει π.χ. μία κάμερα με προκαθορισμένη τροχιά), και μετά να κινεί ο ίδιος ο εξυπηρετητής τον avatar του πελάτη με τις μεθόδους της κλάσης AIPlayer (π.χ. να τον κάνει να κινηθεί σε ένα προκαθορισμένο μέρος όπου έχει προγραμματιστεί να συμβεί κάποιο γεγονός). Χρησιμοποιούμε συνεπώς την κλάση *AIPlayer* και για τη δημιουργία των avatars.

Σχετικά αρχεία:
/control/server/scripts/server.cs

Διατήρηση κατάστασης (state) για κάθε πελάτη από τον εξυπηρετητή

Ένας εξυπηρετητής μπορεί να διαχειρίζεται πολλούς πελάτες, και πρέπει να διατηρεί στοιχεία για την κατάσταση του καθενός. Τα στοιχεία αυτά αφορούν θέματα όπως το όνομα του χρήστη, η τρέχουσα κατάσταση των χαρακτηριστικών του (υγεία κλπ.), τα αντικείμενα που μεταφέρει, οι αποστολές που έχει ολοκληρώσει, κλπ.

Δεδομένου ότι τέτοιες ευαίσθητες πληροφορίες δεν μπορεί να είναι άμεσα διαθέσιμες στον κάθε πελάτη, όπου μπορεί να αλλοιωθούν, αλλά και λόγω του γεγονότος πως πρέπει να υπάρχει μία κεντρική οντότητα που ανά πάσα στιγμή έχει όλη την πληροφορία για τον κόσμο και διατηρεί όλους τους άλλους ενημερωμένους και συγχρονισμένους, καταλήγουμε ότι ο εξυπηρετητής είναι το μέρος όπου οι πληροφορίες αυτές θα πρέπει να βρίσκονται. Γεννιέται λοιπόν το ερώτημα πού θα αποθηκεύει ο εξυπηρετητής αυτά τα χαρακτηριστικά για κάθε πελάτη του.

Μία σκέψη θα ήταν να αποθηκεύονται στο αντικείμενο το οποίο αφορούν, π.χ. αν ένας NPC είναι θυμωμένος με έναν παίκτη να κρατάμε αυτή την πληροφορία σε μία ιδιότητα του αντικειμένου που τον αναπαριστά. Όμως ο NPC θα πρέπει να θυμάται την διάθεσή του προς τον κάθε πελάτη ξεχωριστά, που σημαίνει ότι μία μεταβλητή δεν αρκεί, αλλά θα έπρεπε να έχει έναν πίνακα μεταβλητών το μέγεθος του οποίου θα αυξομειώνεται δυναμικά.

Εδώ έχουμε επιλέξει όλη η κατάσταση που είναι σχετική με έναν πελάτη να αποθηκεύεται στο αντικείμενο της κλάσης AIPlayer που δημιουργεί ο εξυπηρετητής για τον κάθε πελάτη όταν αυτός συνδέεται μαζί του, και το οποίο αντικείμενο αναπαριστά τον πελάτη στον κόσμο (συνεπώς όλοι οι πελάτες θα έχουν ένα και μόνο ένα τέτοιο αντικείμενο). Η κατάσταση αυτή δεν είναι τίποτα περισσότερο από ένα σύνολο οριζόμενων-από-τον-χρήστη ιδιοτήτων, που ορίζονται στο αντικείμενο αυτό τη στιγμή που το δημιουργεί ο εξυπηρετητής, πριν ο πελάτης εισέλθει στον κόσμο. Οι επιμέρους ιδιότητες και η χρησιμότητά τους παρουσιάζονται στις αντίστοιχες ενότητες.

Σχετικά αρχεία:
/control/server/scripts/server.cs

TorqueScript και πίνακες

Η TorqueScript υποστηρίζει πίνακες μίας και δύο διαστάσεων, προκύπτει όμως το εξής θέμα: δεν τους αντιμετωπίζει σαν αντικείμενα κάποιας κλάσης, πράγμα που δυσκολεύει την αποστολή τους μέσω του μηχανισμού ανταλλαγής μηνυμάτων, καθώς δεν μπορούν να σταλούν σαν ένα ενιαίο αντικείμενο-πίνακας.

Μία επιλογή είναι να στέλνουμε για κάθε γραμμή του πίνακα τα στοιχεία όλων των στηλών της, να προχωράμε στην επόμενη γραμμή κλπ. μέχρι να σταλούν όλες οι γραμμές. Αυτό σημαίνει ότι ο αποστολέας πρέπει να ενημερώνει τον παραλήπτη για το πόσες γραμμές να περιμένει, και ο παραλήπτης περιμένει να πάρει όλες τις γραμμές πριν χρησιμοποιήσει τον πίνακα (τον οποίο ανακατασκευάζει γραμμή-γραμμή στη δική του πλευρά). Και βέβαια αυτή η αντιμετώπιση σπαταλά εύρος ζώνης στέλνοντας πολλά μικρά μηνύματα. Αυτή πάντως είναι η προσέγγιση που χρησιμοποιούμε στο demo.

Μία άλλη επιλογή θα ήταν να "ενθυλακώσει" ο αποστολέας τον πίνακα σε ένα αντικείμενο (π.χ. αντικείμενο της κλάσης ScriptObject) και να στείλει το αντικείμενο. Έτσι αντιμετωπίζονται τα προηγούμενα προβλήματα/

(Σημείωση: η δεύτερη μέθοδος δεν δοκιμάστηκε εδώ, οπότε διατηρούμε μία επιφύλαξη σχετικά με το κατά πόσο είναι εφικτό να σταλούν μεγάλοι πίνακες με αυτό τον τρόπο, και αυτό διότι ο μηχανισμός ανταλλαγής μηνυμάτων του Torque φαίνεται να επιβάλει ένα άνω όριο 255 χαρακτήρων -bytes- στο κάθε μήνυμα που στέλνεται)

14.Σύνοψη- Συμπεράσματα- Μελλοντικές επεκτάσεις

Συνολικά, περιγράψαμε παραπάνω μια πρώτη προσέγγιση της δημιουργίας ενός εικονικού περιβάλλοντος / κόσμου στον οποίο μπορούν να περιηγηθούν οι χρήστες, αλληλεπιδρώντας με τους διάφορους εικονικούς πράκτορες και αντικείμενα που βρίσκονται σε αυτόν, στα πλαίσια ενός παιχνιδιού.

Είδαμε το γενικό διαχωρισμό μιας τέτοιας εφαρμογής σε επιμέρους, και κατά το δυνατόν αυτόνομα, στοιχεία που την αποτελούν (ή τουλάχιστον που αποτελούν το μεγαλύτερο μέρος της), καθώς και κάποιες λεπτομέρειες για την υλοποίησή τους (για περισσότερες λεπτομέρειες βλ. πηγαίο κώδικα). Τα επιμέρους αυτά στοιχεία ήταν:

- ένα σύστημα διαχείρισης των εικονικών αντικειμένων που αποκτά ο χρήστης κατά τη διάρκεια της περιήγησής του (Inventory system, βλ. Κεφ. 7)
- ένα σύστημα επικοινωνίας (μέσω διαλόγου) του χρήστη με τους εικονικούς πράκτορες που συναντά κατά τη διάρκεια της περιήγησής του (Dialog system, βλ. Κεφ. 8)
- ένα σύστημα καταγραφής και παρακολούθησης της εξέλιξης των διαφόρων στόχων / αποστολών του χρήστη (Quest log, βλ. Κεφ 9)
- ένα σύστημα μέσω του οποίου ο χρήστης λαμβάνει την οπτική πληροφορία για τον κόσμο (το αποτέλεσμα δηλ. του rendering), το οποίο επιπλέον προσφέρει άμεση (επεξηγηματική) πληροφορία για το τι είναι αυτό που βλέπει ο χρήστης (HUD, βλ. Κεφ 10)
- ένα σύστημα μάχης, που είναι ουσιαστικά ένας εναλλακτικός τρόπος αλληλεπίδρασης του χρήστη με τους εικονικούς πράκτορες που συναντά (Combat system, βλ. Κεφ. 11)
- ένα σύστημα που μας επιτρέπει να επηρεάσουμε την αντίληψη κάποιου χρήστη για τον κόσμο σε πραγματικό χρόνο από ένα απομακρυσμένο μηχάνημα, π.χ. να τον ξαφνιάσουμε με απροσδόκητα γεγονότα, όπως ένας κεραυνός που πέφτει ξαφνικά μπροστά του (Απομακρυσμένος έλεγχος με sockets, βλ. Κεφ 12)

Συμπερασματικά, βλέπουμε ότι η ανάπτυξη ενός, περιορισμένου έστω, εικονικού κόσμου δεν είναι ανέφικτη, και μάλιστα τα τελευταία χρόνια έχουν αναπτυχθεί πάρα πολλά έργα λογισμικού που μπορούν να μας βοηθήσουν σε αυτή τη διαδικασία. Από λογισμικό δημιουργίας ψηφιακού περιεχομένου (DCC - Digital Content Creation) και επεξεργασίας (όπως διάφορα προγράμματα 3D modeling, επεξεργασίας εικόνας κλπ.) έως μηχανές γραφικών που αναλαμβάνουν τη διαδικασία του rendering, αλλά ακόμα και έτοιμα μοντέλα φυσικής και γενικά μηχανές παιχνιδιών σαν ευρύτερος όρος (το TGE είναι απλώς μία από πολλές τέτοιες μηχανές), όλα αυτά δημιουργούν μία βάση πάνω στην οποία μπορεί να στηριχθεί κανείς και να δημιουργήσει πλέον σε ένα "υψηλότερο" επίπεδο, δοκιμάζοντας για παράδειγμα να υλοποιήσει μοντέλα συναισθημάτων ή τεχνητής νοημοσύνης, χωρίς να ανησυχεί για λειτουργίες "χαμηλότερου" επιπέδου (η διαστρωμάτωση σε αφαιρετικά επίπεδα είναι έτσι κι αλλιώς βασικό γνώρισμα στον κόσμο των υπολογιστών).

Πιθανές κατευθύνσεις για μελλοντικές επεκτάσεις:

- στο σύστημα διαχείρισης αντικειμένων:
  συνδυασμός των διαφορετικών αντικειμένων μεταξύ τους για την παραγωγή νέων αντικειμένων με νέες ιδιότητες, κάθε αντικείμενο να δεσμεύει χώρο

ανάλογα με το μέγεθός του (αντί όλα να δεσμεύουν μία θέση), δυνατότητα να οργανώνει ο χρήστης το inventory όπως θέλει αντί αυτό να γίνεται αυτόματα, δυνατότητα ταυτόχρονης χρήσης περισσότερων από δύο αντικειμένων, κλπ.

- στο σύστημα διαλόγου:
  ενσωμάτωση πιο πολύπλοκων μηχανών καταστάσεων ή και υλοποίηση ολόκληρου του μοντέλου OCC (ή κάποιου άλλου μοντέλου συναισθημάτων, βλ. Γενική θεώρηση της αλληλεπίδρασης με εικονικές οντότητες, Κεφ. 1) για τη δημιουργία μεγαλύτερης αληθοφάνειας στις αντιδράσεις των εικονικών πρακτόρων κατά τη διάρκεια ενός διαλόγου, δημιουργία μοντέλων και animation από 3D artists ώστε να επιτευχθεί και η οπτική απόδοση των συναισθημάτων των εικονικών πρακτόρων (σε αντίθεση με την απόδοση των συναισθημάτων μόνο μέσω του κειμένου που αντιπροσωπεύει τα λόγια του εικονικού πράκτορα), περισσότερες επιλογές του χρήστη στους διαλόγους με περισσότερα πιθανά αποτελέσματα των διαφόρων επιλογών στην έκβαση του διαλόγου κλπ.
- στο σύστημα αποστολών:
  δυνατότητα να επιλέξει ο χρήστης ποια αποστολή θέλει να θεωρείται ως ενεργή και να λαμβάνει περισσότερες και πιο συγκεκριμένες πληροφορίες για αυτή (ο χάρτης να ανανεώνεται για την αποστολή της επιλογής του), κλπ.
- στο σύστημα μάχης:
  δυνατότητα για τη συμμετοχή περισσότερων από δύο εικονικών πρακτόρων ή / και χρηστών ταυτόχρονα στην ίδια μάχη, δημιουργία ομάδων συνεργασίας μεταξύ τους, προσθήκη νέων ιδιοτήτων (εκτός της ζωής και της ενέργειας) για κάθε συμμετέχοντα που θα επηρεάζουν την εξέλιξη της μάχης, ενίσχυση της διαδικασίας λήψης αποφάσεων (τυχαία) από της εικονικές οντότητες με παράγοντες που θα επηρεάζονται ενδεχομένως από κάποιο μοντέλο συναισθημάτων, προσθήκη περισσότερων επιλογών για τις ενέργειες του χρήστη κλπ.
- στο σύστημα απομακρυσμένου ελέγχου:
  εδώ οι πιθανότητες είναι άπειρες, καθώς μπορούμε με κατάλληλες προσθήκες να τροποποιήσουμε οποιαδήποτε πλευρά του κόσμου, των αντικειμένων, των εικονικών πρακτόρων και των χρηστών θέλουμε, και αυτά σε πραγματικό χρόνο. Πάντως κάτι τέτοιο εμπεριέχει τον κίνδυνο να χαλάσει η ισορροπία του εικονικού κόσμου καταστρέφοντας την εμπειρία των χρηστών, γι αυτό και θα πρέπει να γίνεται με προσοχή.

15.Βιβλιογραφία

Torque Game Engine:

- "3D Game Programming Al in One" by Kenneth C. Finney, Course Technology / Premier Press (2004)

- "Advanced 3D Game Programming All In One" by Kenneth C. Finney, Course Technology / Premier Press (2005)

- "The Game Programmer's Guide to Torque: Under the Hood of the Torque Game Engine" by Edward F. Maurina III, AK Peters (2006) (σημείωση: ειδικά το Appendix A)

Θεωρία Συναισθημάτων – Αλληλεπίδραση Ανθρώπου Υπολογιστή:

- [KSHI02] "A Multilayer Personality Model", Kshirsagar S., Proceedings of 2nd International Symposium on Smart Graphics, Hawthorne, New York, pp. 107-115 (2002)

- [MOFF97] "Personality Parameters and Programs", Moffat D., Creating Personalities for Synthetic Actors, Springer Verlag, New York, pp. 120-165 (1997)

- [WILS99] 'The Artificial Emotion Engine TM, Driving Emotional Behavior", Wilson I., AAAI 2000 Spring Symposium on Artificial Intelligence and Interactive Entertainment, AAAI Press, pp. 76-80 (1999)

- [ROUS97] "A Social-Psychological Model for Synthetic Actors", Rousseau D. & Hayes-Roth, Knowledge Systems Laboratory, Stanford University (1997)

- [EKMA72] "Emotion in the Human Face: Guidelines for Research and an Integration of Findings", Ekman P., Friesen W.V. & Ellsworth P., Pergamon Press Inc., New York (1972)

- [BART02] "Integrating the OCC Model of Emotions in Embodied Characters." Bartneck C., Proceedings of the Workshop on Virtual Conversational Characters: Applications, Methods, and Research Challenges, Melbourne (2002)

- [PARU06] "A Model of Emotions for Situated Agents", H. Van Dyke Parunak, Robert Bisson, Sven Brueckner, Robert Matthews & John Sauter, AAMAS'06, May 8-12, 2006, Hakodate, Hokkaido, Japan (2006)

- [CRAI] "Game Play Schemas: From Player Analysis to Adaptive Game Mechanics", Craig A. Lindley & Charlotte C. Sennersten, ()

- [REEK04] '"Psychophysiological responses to appraisal dimensions in a computer game", van Reekum, Carien M., Johnstone, Tom, Banse, Rainer, Etter,Alexandre, Wehrle, Thomas and Scherer, Klaus R. , (2004), Cognition & Emotion, 18:5, 663 - 688

- [SCHE02] "Frustrating the user on purpose: a step toward building an affective computer", Jocelyn Scheirer, Raul Fernandez, Jonathan Klein & Rosalin W. Picard, Elsevier Science B.V. (2002)

- [WIKI07] article on "Paul Ekman", Wikipedia – The Free Encyclopedia (http://en.wikipedia.org/wiki/Paul_Ekman) (2007)

Ευχαριστίες

Η διπλωματική αυτή ολοκληρώθηκε υπό την επίβλεψη του καθηγητή Ε.Μ.Π. κ.Κόλλια και την επίβλεψη και καθοδήγηση του ερευνητή Ε.Μ.Π. Κώστα Καρπούζη, ο οποίος και μου υπέδειξε το Torque, η ενασχόλησή μου με το οποίο ήταν πραγματικά ενδιαφέρουσα. Ακόμα, να ευχαριστήσω εδώ και τη Λόρι Μαλατέστα και γενικά όλους όσους ασχολήθηκαν με τη συγκέντρωση βιβλίων, tutorials, παραδειγμάτων και άλλων πληροφοριών γύρω από το Torque, καθώς ήταν καθοριστικής σημασίας σε όλη η διάρκεια αυτής της διπλωματικής.