Πανεπιστήμιο Αιγαίου
Τμήμα Πολιτισμικής
Τεχνολογίας και Επικοινωνίας

Σχεδίαση και ανάπτυξη demo παιχνιδιού με χρήση προγραμματισμού και τρισδιάστατων γραφικών

Πτυχιακή Εργασία

- Ματζάκου Αντωνίου
- Μποφίλιου Νικολάου

Τριμελής Επιτροπή :

- Τσεκούρας Γεώργιος (Επιβλέπων)
- Καλλονιάτης Χρήστος
- Μπαντιμαρούδης Φιλήμων

Μυτιλήνη Σεπτέμβριος 2009

Περιεχόμενα

1. Περίληψη……………………………………………………………………4
2. Κεφάλαιο 1 : Εισαγωγή στην ανάπτυξη ηλεκτρονικών παιχνιδιών ………….5
3. Κεφάλαιο 2 : Προγραμματισμός και δημιουργία τρισδιάστατων γραφικών για ηλεκτρονικά παιχνίδια……………………………………………………..11
   2.1. Στάδια ανάπτυξης ηλεκτρονικών παιχνιδιών ……………………………...11
   2.2. Γενικά περί τρισδιάστατων γραφικών…………………………………...15
      2.2.1. Τρόποι μοντελοποίησης…………………………………………..17
      2.2.2. Χρώμα…………………………………………………………….21
         2.2.2.1. Βασικοί τύποι UV mapping……………………………..22
         2.2.2.2. Η δύναμη του 2 στα textures…………………………….24
      2.2.3. Η τεχνολογία των shaders για τους 3d artists……………………..…25
      2.2.4. Λίγα λόγια για το animation………………………………………27
         2.2.4.1. Skeletal animation………………………………………….30
      2.2.5. Rendering-Lighting…………………………………………………30
         2.2.5.1. Τύποι φώτων………………………………………………31
         2.2.5.2. Shadows……………………………………………………37
   2.3. Γενικά περί προγραμματισμού ………………………………………38
      2.3.1. Μηχανή απεικόνισης γραφικών (Rendering engine)…..……………39
      2.3.2. Μηχανή Φυσικής (Physics engine)…………………………………42
      2.3.3. Μηχανή Ήχου (Sound engine)………………………………………46
      2.3.4. Μηχανή Τεχνητής Νοημοσύνης (A.I)………………………………47
      2.3.5. Shaders………………………………………………………………48
4. Κεφάλαιο 3 : Ανάπτυξη του demo παιχνιδιού………………………………..56
   3.1. Εισαγωγή στον προγραμματισμό και στα τρισδιάστατα γραφικά του demo……………………………………………………..56
   3.2. Προγραμματισμός……………………...……………………………..59
   3.3. Τρισδιάστατα γραφικά…………………..…………………………...68

3.3.1. Concept…..……………………………………………………….....68
3.3.2. Sketch art…………………..……………………………………......68
3.3.3. Modeling…………………...……………………………………......69
3.3.5. Rigging…………………...…………………………………………82
3.3.6. Animation…….……………………………………………………..84
3.3.7. Engine export………………………………………………………..85
3.3.8. Rendering…………………...………………………………………86
5. Κεφάλαιο 4 : Συνδυασμός προγραμματισμού και γραφικών……………………90
6. Επίλογος ……………………………….…………………………………...92
7. Βιβλιογραφία ………………………………………………………………93
8. Παράρτημα ……..…………………………………………………………94

Περίληψη

Ο όρος «ανάπτυξη παιχνιδιού» είναι η διαδικασία κατά την οποία παράγεται ένα παιχνίδι (Game Development) [14]. Η παραγωγή του παιχνιδιού γίνεται από ένα άτομο ή μια ομάδα ατόμων που αναλαμβάνουν τη δημιουργία του. Συνήθως πρόκειται για εταιρείες (studios) όπως η Electronic Arts , η Activision, η Radical Entertainment, η Nitendo, η Sony, η Rockstar Games κ.ά.

Ενδεικτικά, η παραγωγή ενός μοντέρνου ηλεκτρονικού παιχνιδιού μπορεί να κοστίσει από 1.000.000$ έως πάνω από 20.000.000$ περίπου. Επίσης ένα ηλεκτρονικό παιχνίδι μπορεί να χρειαστεί από ένα έως και τρία χρόνια περίπου για να παραχθεί και να κυκλοφορήσει στην αγορά. Για την παραγωγή ενός ηλεκτρονικού παιχνιδιού απαιτείται μια ομάδα ανάπτυξης η οποία αποτελείται από άτομα που έχουν διαφορετικά καθήκοντα το καθένα. Ένα ηλεκτρονικό παιχνίδι επίσης χρειάζεται να περάσει και από κάποια στάδια παραγωγής μέχρι την τελική κυκλοφορία του.

Τέλος το λογισμικό που μπορεί να χρησιμοποιηθεί για την ανάπτυξη ενός ηλεκτρονικού παιχνιδιού ποικίλει. Περιέχει λογισμικά συγγραφής κώδικα, λογισμικά κατασκευής τρισδιάστατων γραφικών, λογισμικά επεξεργασίας εικόνας, λογισμικά επεξεργασίας ήχου, λειτουργικά συστήματα κ.ά.

Σκοπός λοιπόν, της εργασίας ήταν η πρακτική εφαρμογή στοιχείων της ανάπτυξης σύγχρονων ηλεκτρονικών παιχνιδιών, δημιουργώντας ένα game demo. Μερικά από τα πιο βασικά στοιχεία που βρήκαν εφαρμογή στο demo ήταν η δημιουργία τρισδιάστατων γραφικών, animation και η εκτέλεσή τους σε πραγματικό χρόνο. Η ανάπτυξη όλων των επιμέρους στοιχείων έγινε από το «μηδέν» και με «μηδενικές» γνώσεις. Με αυτόν τον τρόπο μπήκαν οι σωστές βάσεις για την απόκτηση γνώσεων σχετικών με τον τομέα των γραφικών και του προγραμματισμού.

ΚΕΦΑΛΑΙΟ 1

Εισαγωγή στην ανάπτυξη ηλεκτρονικών παιχνιδιών

Η παραγωγή ενός ηλεκτρονικού παιχνιδιού μπορεί να διαρκέσει από 1 έως 3 περίπου χρόνια και αυτό εξαρτάται από διάφορους παράγοντες. Τέτοιοι παράγοντες είναι το είδος του παιχνιδιού, το μέγεθός του, η πλατφόρμα ανάπτυξής του και η ποσότητα των στοιχείων που το αποτελούν [14]. Για παράδειγμα ένα παιχνίδι παζλ που χρησιμοποιεί δισδιάστατα γραφικά είναι προφανές ότι θα χρειαστεί λιγότερο χρόνο να παραχθεί από ένα role-playing παιχνίδι το οποίο χρειάζεται μία πλήρη 3D μηχανή. Ένας ακόμη παράγοντας μιας και αναφέρθηκε η 3D μηχανή είναι και το αν θα χρησιμοποιηθεί έτοιμη μηχανή για την ανάπτυξη του παιχνιδιού ή θα κατασκευαστεί βήμα-βήμα από την αρχή. Υπάρχουν μηχανές οι οποίες είναι ελεύθερες προς χρήση από οποιονδήποτε θέλει να τις χρησιμοποιήσει για την κατασκευή του παιχνιδιού του. Για παράδειγμα η “Gas Powered Games” ανέπτυξε μια δική της 3D μηχανή για το παιχνίδι της “Dungeon Siege”, το οποίο χρειάστηκε 3 χρόνια για να αναπτυχθεί. Από την άλλη η “Fir axis” χρησιμοποίησε την έτοιμη μηχανή “Gamebryo” για το παιχνίδι της “Sid Meier’s Pirates”, το οποίο χρειάστηκε μόλις 2 χρόνια για να αναπτυχθεί. Όσον αφορά τα στοιχεία (assets) του παιχνιδιού είναι επίσης προφανές ότι το παιχνίδι παζλ χρειάζεται πολύ λιγότερα στοιχεία από το αντίστοιχο role-playing-game παιχνίδι. Τα στοιχεία ενός παιχνιδιού μπορούν επίσης να κατασκευαστούν από το μηδέν ή να χρησιμοποιηθούν έτοιμα κατόπιν τήρησης των πνευματικών δικαιωμάτων. Όπως γίνεται εύκολα αντιληπτό αυτός είναι ένας ακόμη παράγοντας που μπορεί να αυξήσει ή να μειώσει το χρόνο ανάπτυξης ενός ηλεκτρονικού παιχνιδιού. Καμιά φορά βέβαια υπάρχουν και περιπτώσεις στις οποίες ο χρόνος ανάπτυξης μπορεί να ξεπεράσει το «μέσο χρόνο» και να διαρκέσει πολύ περισσότερο, όπως για παράδειγμα το παιχνίδι “Duke Nukem Forever” της “3D Realms” το οποίο είχε ανακοινωθεί ότι θα ήταν στην παραγωγή τον Απρίλιο του 1997 αλλά ακυρώθηκε μέχρι το Μάιο του 2009 και τελικά ακυρώθηκε εντελώς.

Στις μέρες μας η βιομηχανία των ηλεκτρονικών παιχνιδιών αναπτύσσεται ραγδαία, η ανάπτυξη είναι τόσο μεγάλη που ξεπερνάει ακόμα και την ανάπτυξη της βιομηχανίας του κινηματογράφου. Γιατί συμβαίνει αυτό ; Στην συνέχεια ακολουθεί μια σύγκριση κάποιων στοιχείων.

Τιμή:

-Το κόστος για να παρακολουθήσει κάποιος μια ταινία είναι περίπου 10$ ενώ 20$ για να αγοράσει μια ταινία DVD. Η διάρκεια της ταινίας είναι περίπου από ενενήντα λεπτά μέχρι δύο ώρες.

-Το κόστος για την απόκτηση ενός ηλεκτρονικού παιχνιδιού για Η/Υ είναι περίπου 30-40 $ ενώ για κονσόλα 50-60 $. Η διάρκεια των ηλεκτρονικών παιχνιδιών ποικίλλει, γενικά κάτι μικρότερο από έξι ώρες θεωρείται μικρό.

Το μεγαλύτερο πλεονέκτημα των ηλεκτρονικών παιχνιδιών είναι ότι μπορεί ο αγοραστής- χρήστης να τα ξαναπαίξει. Εφόσον ο χρήστης μπορεί να αλληλεπιδράσει με πολλούς τρόπους με το ηλεκτρονικό παιχνίδι μπορεί στην ουσία να το παίξει και να το τελειώσει με πολλούς διαφορετικούς τρόπους.

Κόστος:

-Μια ακριβή ταινία όπως το Spiderman 3 χρειάζεται περίπου 250 εκατ. δολάρια για να δημιουργηθεί. Μια μέση ταινία χρειάζεται περίπου 106 εκατ. δολάρια.

-Ένα από τα πιο ακριβά ηλεκτρονικά παιχνίδια το GTA IV κόστισε περίπου 100 εκατ. δολάρια. Ένα μέσο ηλεκτρονικό παιχνίδι κοστίζει περίπου 10- 50 εκατομμύρια δολάρια.

Στην πρεμιέρα του Spiderman 3 εισπράχθηκαν 59 εκατομμύρια δολάρια από τις πωλήσεις εισιτηρίων, ενώ την πρώτη μέρα κυκλοφορίας του GTA IV 30 εκατομμύρια δολάρια.

Οι πωλήσεις ηλεκτρονικών παιχνιδιών το 1996 έτσι έφθαναν τα 2,6 δισεκατομμύρια δολάρια και μέχρι το 2007 έφταναν τα 18,8 δισεκατομμύρια δολάρια. Το 2007 η βιομηχανία των ηλεκτρονικών παιχνιδιών αυξήθηκε 40% σε σχέση με τον προηγούμενο χρόνο και έφθασε σε πωλήσεις τα 267,8 εκατ. παιχνίδια.

Αυτή η ανάπτυξη οφείλεται στο ότι τα ηλεκτρονικά παιχνίδια απευθύνονται σε όλες τις ηλικίες και σε όλα τα δημογραφικά και γεωγραφικά υπόβαθρα.

Η βιομηχανία των ηλεκτρονικών παιχνιδιών χωρίζεται στα παιχνίδια για Η/Υ και στα παιχνίδια για κονσόλες. Τα περισσότερα ηλεκτρονικά παιχνίδια είναι παιχνίδια κονσόλας, πράγμα που σημαίνει ότι μπορούν να παίζουν σε ένα συγκεκριμένο σύστημα κάποιας εταιρίας, σε αντίθεση με τα παιχνίδια για Η/Υ που μπορούν να παίζουν σε οποιονδήποτε Η/Υ.

Τα στερεότυπα πού θέλουν τα ηλεκτρονικά παιχνίδια να παίζονται από εφήβους και να έχουν μεγάλες δόσεις βίας αλλάζουν. Οι χειριστές είναι πιθανότερο να είναι νέοι επαγγελματίες που μπορούν να αντέξουν οικονομικά το κόστος ενός ηλεκτρονικού παιχνιδιού. Οι άνδρες δεν είναι οι μόνοι που παίζουν. Το 40% των χρηστών είναι γυναίκες και αντιπροσωπεύουν όλο και μεγαλύτερο ποσοστό κάθε χρόνο. Οι εταιρείες προσαρμόζουν τα παιχνίδια τους σε αυτό το ακροατήριο και τα κάνουν λιγότερο βίαια και περισσότερο οικογενειακά. Δεδομένου ότι οι νεότερες γενεές μεγαλώνουν και αποκτούν τα δικά τους παιδιά, όλο και περισσότεροι γονείς παίζουν ηλεκτρονικά παιχνίδια.

Ένα ακροατήριο ακόμα που μεγαλώνει είναι αυτό του κοινού 50 ετών και πάνω, μιας και τα ηλεκτρονικά παιχνίδια αποτελούν μια πολύ καλή δραστηριότητα χωρίς τη φυσική πίεση. Προσφέρουν οφέλη υγείας στους χρήστες με παιχνίδια ισορροπίας συντονισμού και αντοχής.

Η βιομηχανία των ηλεκτρονικών παιχνιδιών είναι παρεξηγημένη από το ευρύ κοινό. Το ευρύ κοινό δεν κατανοεί την διαδικασία που χρειάζεται για να δημιουργηθεί ένα παιχνίδι σε αντίθεση με μια ταινία για την οποία έχει μια βασική γνώση. Υπάρχουν δηλαδή κάμερες που καταγράφουν, ηθοποιοί που αλληλεπιδρούν μεταξύ τους με βάση το σενάριο και κασκαντέρ για τις επικίνδυνες σκηνές κτλ. Για ένα παιχνίδι δεν γνωρίζει σχεδόν τίποτα. Δεν γνωρίζει πώς δημιουργούνται οι χαρακτήρες, οι σκηνές , η κίνηση κ.τ.λ από τους 3d artists, ούτε το πως ανοιγοκλείνει μια πόρτα, εντοπίζονται συγκρούσεις, υπάρχουν ανακλάσεις στα παράθυρα κ.τ.λ από τους προγραμματιστές. Αυτή η έλλειψη γνώσεων γύρω από τη βιομηχανία ηλεκτρονικών παιχνιδιών μπορεί να αποτελέσει πρόβλημα. Εάν κάποιος δεν μπορεί να κατανοήσει την δυσκολία δημιουργίας ηλεκτρονικών παιχνιδιών, μπορεί να θεωρήσει ότι πρόκειται για μια φθηνή και απλή διαδικασία και να απομακρυνθεί από αυτά.

Η «ανάπτυξη παιχνιδιών» γνώρισε και συνεχίζει να γνωρίζει μεγαλύτερη άνθηση κυρίως στις Ηνωμένες Πολιτείες της Αμερικής, στον Καναδά, στην Αγγλία και στην Ιαπωνία. Κι αυτό λόγο της υψηλής ανάπτυξης των συγκεκριμένων χωρών και δη της τεχνολογικής ανάπτυξης. Αξίζει να σημειωθεί ότι μερικά από τα πιο γνωστά studio ανάπτυξης ηλεκτρονικών παιχνιδιών βρίσκονται στο "Silicon Valley" των Ηνωμένων Πολιτειών.

Η δημιουργία και ανάπτυξη ενός παιχνιδιού σαν επάγγελμα είναι κάτι εξαιρετικά ενδιαφέρον, αλλά και κάτι που εμπεριέχει αρκετές δυσκολίες. Η βιομηχανία των

ηλεκτρονικών παιχνιδιών είναι αρκετά ρευστή όπως επίσης και άλλες βιομηχανίες της τέχνης, όπως η τηλεόραση, η μουσική κ.τ.λ.

Η δουλειά απαιτεί ταχύτητα και ευελιξία καθώς οι εταιρείες ανάπτυξης παιχνιδιών δουλεύουν αρκετά ώστε να κυκλοφορήσουν ένα «τίτλο» και αμέσως μετά αρχίζουν και πάλι δουλειά για να κυκλοφορήσει ο επόμενος. Μερικές φορές παρά το γεγονός ότι έχει αφιερωθεί πολύς κόπος και πολύ δουλειά πάνω σε ένα παιχνίδι, αυτό τελικά μπορεί και να μην κυκλοφορήσει ποτέ στην αγορά. Εδώ μπορεί εύκολα κανείς να καταλάβει ότι υπάρχει και το στοιχείο της απώλειας κέρδους, αφού όπως προαναφέρθηκε για τη δημιουργία ενός παιχνιδιού έχει σπαταληθεί χρόνος, κόπος αλλά και χρήμα το οποίο μπορεί τελικά να μην αποσβεστεί λόγω της μη κυκλοφορίας του «τίτλου» στην αγορά. Αξίζει να σημειώσει κανείς ότι στη βιομηχανία των παιχνιδιών, μόλις το 5% των πιο «κορυφαίων» προϊόντων αποφέρουν ουσιαστικό κέρδος.

Η ομάδα που αναλαμβάνει να φέρει εις πέρας τη δημιουργία και την ανάπτυξη του παιχνιδιού αποτελείται από:

- Παραγωγούς (Producers)
- Σχεδιαστές Παιχνιδιών (Game Designers)
- Καλλιτέχνες (Artists)
- Προγραμματιστές (Programmers)
- Μηχανικούς ήχου (Sound Engineers)
- Δοκιμαστές (Testers)
- Σχεδιαστές επιπέδων (Level Designers)

Όσον αφορά τους «παραγωγούς (producers)», πρόκειται για άτομα τα οποία έχουν την ευθύνη να επιβλέπουν την ανάπτυξη ενός παιχνιδιού. Συνήθως υπάρχουν δυο ειδών παραγωγοί, οι εξωτερικοί (executive producers) και οι εσωτερικοί. Οι εξωτερικοί παραγωγοί επιβλέπουν αρκετές «δουλειές», την κατάστασή τους και τυχόν προβλήματα που μπορεί να παρουσιάζουν. Οι εσωτερικοί παραγωγοί επικεντρώνονται σε μια δουλειά (στην ανάπτυξη ενός παιχνιδιού) και τα καθήκοντά τους σε γενικές γραμμές είναι η διαπραγμάτευση συμβολαίων συμπεριλαμβανομένου και συμφωνίες αδειών, η λειτουργία τους ως σύνδεσμος μεταξύ του προσωπικού ανάπτυξης και του προϊστάμενου προσωπικού, να αναπτύσσουν και να διατηρούν

προγράμματα και budgets, να επιβλέπουν τη δημιουργική και την τεχνική ανάπτυξη του παιχνιδιού, τήρηση χρονοδιαγραμμάτων, κ.τ.λ.

Τώρα όσον αφορά τους «***σχεδιαστές παιχνιδιών (game designers)***», αυτοί είναι υπεύθυνοι για το σχεδιασμό του λεγόμενου "gameplay" του παιχνιδιού. Επίσης είναι υπεύθυνοι για τη σύλληψη και τη σχεδίαση των κανόνων και των δομών ενός παιχνιδιού.

Οι «***καλλιτέχνες (artists)***», είναι υπεύθυνοι για την ανάπτυξη της οπτικής τέχνης(visual art) ενός ηλεκτρονικού παιχνιδιού. Μπορεί να είναι concept/sketch artists, 3d modelers, texture artists, animators. Ένας concept/sketch artist ζωγραφίζει χαρακτήρες, μοντέλα, τοπία στο χαρτί έτσι ώστε να χρησιμοποιηθούν ως «οδηγοί» από τους υπόλοιπους καλλιτέχνες κατά την ανάπτυξη του παιχνιδιού. Ο 3d modeler είναι υπεύθυνος για την δημιουργία των 3d αντικειμένων και γενικότερα οποιουδήποτε 3d μοντέλου σ ένα παιχνίδι. Ο texture artist για την δημιουργία textures που θα λειτουργήσουν με πολλαπλούς τρόπους στα 3d αντικείμενα μέσω των shaders και ο animator δημιουργεί σκελετό και κίνηση στα αντικείμενα που πρέπει να κινηθούν.

Ο «***προγραμματιστής (programmer)***» και συγκεκριμένα ο προγραμματιστής παιχνιδιών είναι ένας μηχανικός λογισμικού ο οποίος κυρίως αναπτύσσει ηλεκτρονικά παιχνίδια ή σχετικό λογισμικό που έχει να κάνει με εργαλεία ανάπτυξης παιχνιδιών. Πρέπει να σημειωθεί ότι η δουλειά του προγραμματιστή διαφέρει από αυτή του σχεδιαστή παιχνιδιών. Υπάρχουν κατηγορίες προγραμματιστών οι οποίοι έχουν διαφορετική δουλειά να κάνουν σε ένα παιχνίδι, όπως ο προγραμματιστής φυσικής τους παιχνιδιού ο οποίος εξομοιώνει τη φυσική (που συναντάμε στην πραγματική ζωή) στο παιχνίδι, ο προγραμματιστής τεχνητής νοημοσύνης ο οποίος δίνει «νου» στο παιχνίδι, τον προγραμματιστή γραφικών ο οποίος κατασκευάζει προχωρημένες μεθόδους απεικόνισης (renderers) των στοιχείων του παιχνιδιού, τον προγραμματιστή ήχου ο οποίος κατασκευάζει τη μηχανή ήχου του παιχνιδιού, τον προγραμματιστή gameplay ο οποίος ειδικεύεται στη στρατηγική του παιχνιδιού δημιουργώντας την «αίσθηση» του παιχνιδιού, τον scripter ο οποίος ασχολείται κυρίως με συμβάντα που λαμβάνουν χώρα σε συγκεκριμένες στιγμές του παιχνιδιού, τον προγραμματιστή επιφάνειας διεπαφής ο οποίος ασχολείται με τη δημιουργία επιφανειών διεπαφής με το χρήστη για το παιχνίδι, τον προγραμματιστή εισόδου ο οποίος ασχολείται με το πώς επηρεάζεται το παιχνίδι αλληλεπιδρώντας με κάποια συσκευή εισόδου, τον προγραμματιστή δικτύου ο οποίος ασχολείται με τη

δυνατότητα να μπορούν πολλοί παίκτες να ανταγωνίζονται μεταξύ τους ή να συνεργάζονται (κυρίως με τη δυνατότητα χρήσης του internet για online multiplayer gaming), τον προγραμματιστή εργαλείων παιχνιδιού ο οποίος «γράφει» εργαλεία τα οποία χρησιμοποιούνται για συγκεκριμένες εργασίες που μπορεί να ζητάει ένα παιχνίδι, ο porting programmer ο οποίος είναι υπεύθυνος για την τροποποίηση του κώδικα του παιχνιδιού έτσι ώστε να μπορεί να τρέξει σε διαφορετικές πλατφόρμες και λειτουργικά συστήματα αλλά και συσκευές όπως κινητά τηλέφωνα, ο προγραμματιστής τεχνολογίας ο οποίος ασχολείται με την αλληλεπίδραση λογισμικού (παιχνιδιού) και υλικού (hardware του υπολογιστή), δηλαδή με θέματα που έχουν να κάνουν με χρήση μνήμης, ταχύτητα εκτέλεσης, κ.τ.λ. που αφορούν την πλατφόρμα στην οποία θα εκτελεστεί το παιχνίδι και τέλος ο lead programmer ο οποίος είναι υπεύθυνος για όλο τον προγραμματισμό του παιχνιδιού, όπως τον έλεγχο της έγκυρης εφαρμογής όλων των υποδιαδικασιών του παιχνιδιού καθώς και τον έλεγχο της συνολικής προόδου από προγραμματιστική σκοπιά.

Ο «***σχεδιαστής επιπέδου (level designer)***» είναι αυτός που σχεδιάζει επίπεδα, διαγωνισμούς ή αποστολές στο παιχνίδι χρησιμοποιώντας ειδικά προγράμματα. Αυτά τα προγράμματα μπορεί να είναι ειδικά level editors ή προγράμματα σχεδίασης τρισδιάστατων ή δισδιάστατων γραφικών.

Οι «***μηχανικοί ήχου (sound engineers)***» μπορεί να είναι συνθέτες (composers), δημιουργοί ηχητικών εφέ (sound effects editors), ή ηθοποιοί ήχου (voice actors). Οι συνθέτες είναι αυτοί που ουσιαστικά δημιουργούν μουσική ενώ οι δημιουργοί ηχητικών εφέ είναι αυτοί που ηχογραφούν και συνθέτουν ήχους οι οποίοι χρησιμοποιούνται για να κάνουν μια συγκεκριμένη διήγηση ή ένα δημιουργικό στοιχείο χωρίς τη χρήση διαλόγου ή μουσικής. Τέλος οι ηθοποιοί ήχου είναι αυτοί που με τη φωνή τους δημιουργούν τους διαλόγους που στη συνέχεια χρησιμοποιούνται στους χαρακτήρες του παιχνιδιού.

Οι «***δοκιμαστές (testers)***» είναι αυτοί που αναλύουν το παιχνίδι σε αρχεία «ελαττωμάτων λογισμικού» σαν μέρος μιας διαδικασίας «ελέγχου ποιότητας» η οποία είναι απαραίτητη στον τομέα της ανάπτυξης του παιχνιδιού. Η δουλειά των δοκιμαστών είναι πάρα πολύ σημαντική καθώς απαιτεί πολύ χρόνο και αρκετές γνώσεις έτσι ώστε ο έλεγχος να είναι ορθός και ξεκάθαρος. Συνήθως ο έλεγχος ξεκινά όταν έχει ολοκληρωθεί περίπου το 75% της δημιουργίας του παιχνιδιού. Υπάρχουν διάφοροι τύποι ελέγχου όπως beta testing, regression testing, compatibility testing κ.τ.λ.

ΚΕΦΑΛΑΙΟ 2

Προγραμματισμός και δημιουργία τρισδιάστατων γραφικών για ηλεκτρονικά παιχνίδια

2.1. Στάδια ανάπτυξης ηλεκτρονικών παιχνιδιών

Η διαδικασία ανάπτυξης ενός ηλεκτρονικού παιχνιδιού συνήθως περιλαμβάνει τα εξής στάδια:

<u>Προ–παραγωγή (Pre-Production):</u> Σ' αυτό το στάδιο δημιουργείται η ιδέα για το παιχνίδι και εγκρίνεται. Συνήθως γίνεται η χρήση δειγμάτων (demo) για βοήθεια στη λήψη της απόφασης, αλλά όχι πάντα καθώς αυτό εξαρτάται από το επίπεδο και τις δεξιότητες του developer (ένας καλός developer δεν χρειάζεται demo για να αποφασίσει και να σκεφτεί). Είναι επίσης σημαντικό να αναφερθεί ότι η παραγωγή ξεκινάει μόνο αν βρεθεί ενδιαφερόμενος publisher. Εκείνοι που παρουσιάζουν συνήθως το project είναι οι σχεδιαστές παιχνιδιού (game designers) αλλά μπορεί και ο καθένας μέσα στην εταιρεία να αναλάβει αυτό το ρόλο. Πριν ξεκινήσει η παραγωγή πλήρους κλίμακας, η ομάδα ανάπτυξης φτιάχνει ένα «αρχείο σχεδίασης», το οποίο περιγράφει τη γενική ιδέα και τα κυριότερα στοιχεία του gameplay λεπτομερώς. Τα αρχεία σχεδίασης μπορεί επίσης να περιέχουν προκαταρτικά σκίτσα των διαφόρων πτυχών του παιχνιδιού, τα οποία κάποιες φορές είναι συνοδευόμενα από λειτουργικά πρωτότυπα από μερικούς τομείς του παιχνιδιού. Γενικά λοιπόν τα αρχεία σχεδίασης ενσωματώνουν ολόκληρο ή το περισσότερο από το υλικό του αρχικού σταδίου και αποτελούν αρχεία τα οποία δεν μπορεί κανείς να πει ότι είναι ολοκληρωμένα καθώς όσο το παιχνίδι βρίσκεται σε ανάπτυξη, αυτά εμπλουτίζονται συνεχώς. Τώρα πριν ολοκληρωθεί ένα εγκεκριμένο σχέδιο, μια ομάδα προγραμματιστών και καλλιτεχνών αρχίζει δουλειά. Οι προγραμματιστές μπορεί να αναπτύξουν γρήγορα και πρόχειρα πρωτότυπα, προβάλλοντας ένα ή περισσότερα χαρακτηριστικά τα οποία οι ενδιαφερόμενοι θα ήθελαν να ενσωματώνει το παιχνίδι ή μπορεί να αρχίσουν να αναπτύσσουν το τεχνικό πλαίσιο που τελικά θα χρησιμοποιήσει το παιχνίδι. Οι

καλλιτέχνες μπορεί να αναπτύξουν σειρά από σκίτσα σαν οδηγό για την ανάπτυξη των προγραμματικών στοιχείων του παιχνιδιού. Όσον αφορά τους παραγωγούς, σε αυτό το στάδιο έχουν μια μερική συμμετοχή η οποία αυξάνεται καθώς προχωράει η ανάπτυξη και η οποία έχει να κάνει κυρίως με το σχεδιασμό του προγράμματος, του budget και την εκτίμηση θεμάτων σχετικά με την ομάδα. Σκοπός του παραγωγού είναι να δημιουργήσει ένα συμπαγές σχέδιο έτσι ώστε όταν έρθει η ώρα να ξεκινήσει έγκαιρα η παραγωγή.

__Παραγωγή (Production):__ Σ' αυτό το στάδιο έχουμε την περίοδο της «πλήρους επάνδρωσης» του project. Οι προγραμματιστές γράφουν αρκετό νέο πηγαίο κώδικα ενώ οι καλλιτέχνες αναπτύσσουν στοιχεία του παιχνιδιού όπως δισδιάστατα και τρισδιάστατα μοντέλα. Οι μηχανικοί ήχου αναπτύσσουν ηχητικά εφέ και οι συνθέτες μουσική για το παιχνίδι. Οι σχεδιαστές επιπέδων δημιουργούν προχωρημένα και εντυπωσιακά επίπεδα (levels) και οι συγγραφείς γράφουν διαλόγους για τις διάφορες σκηνές. Ταυτόχρονα ο σχεδιαστής παιχνιδιού εφαρμόζει και τροποποιεί το σχεδιασμό έτσι ώστε να «συναντάει» τη συγκεκριμένη οπτική του παιχνιδιού. Χαρακτηριστικά και επίπεδα προστίθενται ή αφαιρούνται πολύ συχνά. Η «τέχνη» μπορεί να αναπτυχθεί και η «ιστορία» μπορεί να αλλάξει. Επίσης μπορεί να υπάρξει και αλλαγή της πλατφόρμας εκτέλεσης του παιχνιδιού. Όλες αυτές οι αλλαγές πρέπει να καταγράφονται και να γνωστοποιούνται και στο «υπόλοιπο» της ομάδας ανάπτυξης. Οι περισσότερες αλλαγές προκύπτουν σαν ενημερώσεις του «αρχείου σχεδίασης» που προαναφέρθηκε. Από άποψη χρόνου το πρώτο επίπεδο του παιχνιδιού είναι και το πιο χρονοβόρο. Καθώς οι σχεδιαστές επιπέδου και οι καλλιτέχνες χρησιμοποιούν εργαλεία (λογισμικό) για τη δημιουργία των επιπέδων, απαιτούν χαρακτηριστικά και αλλαγές σε αυτά (εργαλεία) που θα επιτρέψουν τη γρηγορότερη και ποιοτικότερη ανάπτυξη. Λόγω αυτών των προσθηκών τα παλιά επίπεδα μπορεί να γίνουν παρωχημένα και αυτό έχει ως αποτέλεσμα επίπεδα που σχεδιάζονται νωρίς να αναπτύσσονται συνεχώς ή να διαγράφονται τελείως. Το δυναμικό περιβάλλον της «ανάπτυξης παιχνιδιού» έχει ως αποτέλεσμα, η ανάπτυξη των πρώτων επιπέδων να μεταβάλλεται στο πέρασμα του χρόνου. Δεν είναι καθόλου περίεργο σε ένα project τριετίας για παράδειγμα, να δαπανηθεί ένας χρόνος για τη δημιουργία μόνο του πρώτου επιπέδου ενώ τα υπόλοιπα επίπεδα να αναπτυχθούν μέσα σε δύο χρόνια και αυτό γιατί το «σετ των χαρακτηριστικών» είναι ολοένα και πιο πλήρες και η οπτική του παιχνιδιού πιο ξεκάθαρη και σταθερή. Οι δοκιμαστές πιάνουν δουλειά όταν το

project μπορεί να «παιχτεί». Μπορεί να πρόκειται για ένα επίπεδο ή ένα μέρος του παιχνιδιού προς δοκιμή. Αρχικά το να δοκιμαστεί ένα παιχνίδι απαιτεί σχετικά λίγο χρόνο. Οι δοκιμαστές μπορεί να δουλεύουν σε αρκετά παιχνίδια ταυτόχρονα. Καθώς όμως η ανάπτυξη του παιχνιδιού φθάνει προς το τέλος, ένα απλό παιχνίδι συχνά χρειάζεται αρκετούς δοκιμαστές καθώς η πολυπλοκότητα του παιχνιδιού αυξάνεται. Χρειάζεται να δοκιμάσουν νέα χαρακτηριστικά και να δοκιμάσουν επίσης τα ήδη υπάρχοντα. Η δοκιμή είναι κάτι πολύ σημαντικό για τα σύγχρονα πολύπλοκα παιχνίδια καθώς απλές αλλαγές μπορεί να έχουν καταστροφικές συνέπειες.

Προσδοκίες – Προθεσμίες (Milestones): Αυτό είναι τα στάδιο όπου τα εμπορικά project (παιχνίδια) πρέπει να πληρούν κάποιες προσδοκίες-προθεσμίες. Μια εναλλακτική περιγραφή των “milestones” είναι ότι αναπαριστούν ενδιάμεσους στόχους του project και ταυτίζονται με προθεσμίες. Τα milestones περιλαμβάνουν μια “pre-release” έκδοση του παιχνιδιού με ένα προσδοκώμενο (συμφωνημένο) σετ χαρακτηριστικών. Οι συνέπειες της απώλειας ενός “milestone” ποικίλουν από project σε project, αλλά συνήθως σημαίνουν καθυστέρηση εξόφλησης των δόσεων ή τελοσπάντων του συμφωνηθέντος χρηματικού ποσού (μεταξύ τρίτων). Λίγο πριν από ένα milestone πολλές ομάδες ανάπτυξης επεκτείνουν το ωράριο τους έτσι ώστε να ολοκληρώσουν τυχόν «δουλειές» που τους ξέφυγαν στο project ή να διορθώσουν σοβαρά προβλήματα τα οποία θέτουν το όλο project σε μεγάλο κίνδυνο.

Σχεδόν-Ολοκλήρωση (Nearing completion): Είναι το στάδιο λίγο πριν την ολοκλήρωση (completion). Για αρκετό χρόνο τα μέλη των ομάδων κάνουν μεγάλη προσπάθεια για να ολοκληρώσουν την ανάπτυξη του παιχνιδιού. Αυτό έχει ως συνέπεια μεγάλη εξάντληση και μετριότητα γι’ αυτούς που εργάζονται. Αυτή η προσπάθεια είναι σχεδόν απαραίτητη, καθώς αρκετά προβλήματα προκύπτουν την τελευταία στιγμή και αρκετά νέα χαρακτηριστικά προστίθενται πάλι την τελευταία στιγμή. Το προσωπικό των δοκιμαστών δουλεύει αυτή την περίοδο δοκιμάζοντας καινούρια χαρακτηριστικά αλλά και αυτά που υπάρχουν ήδη (regression testing). Το regression testing είναι πολύ σημαντικό καθώς όλα σχεδόν τα επίπεδα και τα χαρακτηριστικά έχουν αναπτυχθεί σε μεγάλο βαθμό, κι αυτό έχει ως αποτέλεσμα ήδη υπάρχοντα «χαρακτηριστικά του παιχνιδιού» τα οποία λειτουργούσαν πριν, τώρα να μη λειτουργούν. Η αποτυχία του “regression testing” είναι ένα αρκετά συχνό φαινόμενο που ευθύνεται για την κυκλοφορία παιχνιδιών με bugs.

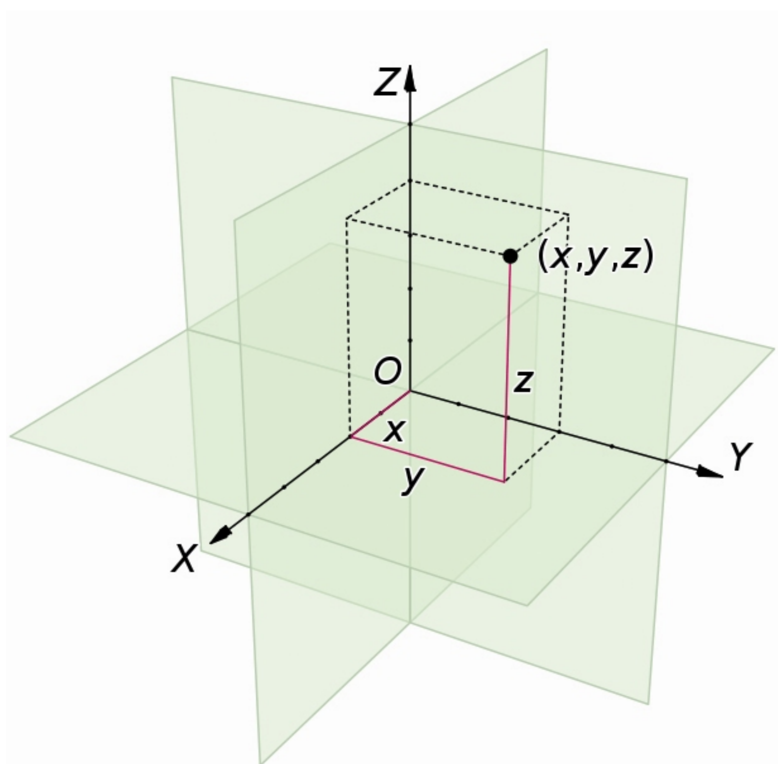
<u>Ολοκλήρωση (Completion):</u> Είναι το στάδιο στο οποίο πλέον το παιχνίδι έχει ολοκληρωθεί και «τρέχει». Εδώ τα μέλη των ομάδων έχουν συνήθως μια ευκαιρία χαλάρωσης.

<u>Συντήρηση (Maintenance):</u> Τα ηλεκτρονικά παιχνίδια εθεωρείτο ότι όταν κυκλοφορήσουν είναι 100% έτοιμα και δεν επιδέχονταν κανενός είδους τροποποίηση. Όμως σήμερα με την παρουσία "online-enabled" κονσόλων όπως το X box 360, Playstation 3, Nitendo wii, κ.α., ένας μεγάλος αριθμός παιχνιδιών λαμβάνει επιδιορθώσεις και προσθήκες (fixes, patches, κ.τ.λ.) ακόμη και μετά την κυκλοφορία τους, όπως αυτά που προορίζονται για προσωπικούς υπολογιστές (pc).

2.2. Γενικά περί τρισδιάστατων γραφικών

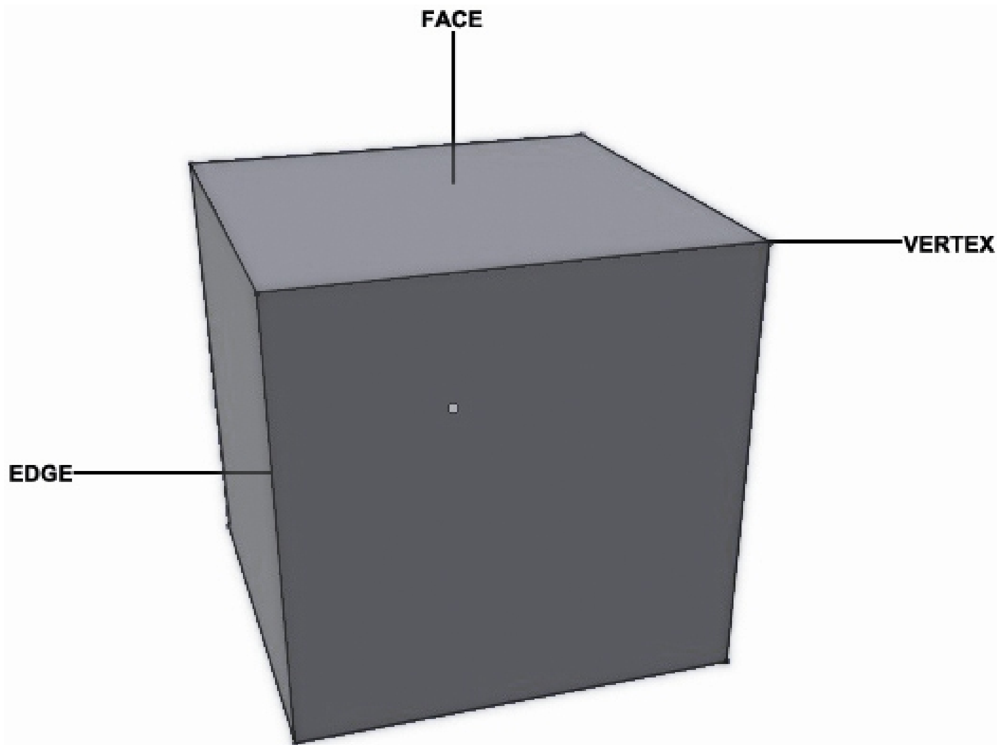
Σ’ ένα ηλεκτρονικό παιχνίδι οι 3D artists είναι υπεύθυνοι για την δημιουργία 3D μοντέλων, την βαφή-υφή τους, τη δημιουργία σκελετών-animation κτλ.. Τι σημαίνουν όμως αυτοί οι όροι και πως συνδέονται μεταξύ τους;

Τα 3D γραφικά αναπαριστούν γεωμετρικά στοιχεία (αντικείμενα) στις τρεις διαστάσεις διαμέσου του Η/Υ. Χρησιμοποιούνται για την δημιουργία 2D εικόνων (rendering) ή για την παρουσίαση τους σε πραγματικό χρόνο (realtime rendering). Οι τρεις διαστάσεις σε έναν Η/Υ αντιπροσωπεύονται από τρεις άξονες X,Y,Z σε ένα 3D καρτεσιανό σύστημα συντεταγμένων και το αντικείμενο που αναπαρίσταται ονομάζεται 3D μοντέλο.



Εικόνα 1: Ένα 3D καρτεσιανό σύστημα συντεταγμένων

Τα τρισδιάστατα μοντέλα αποτελούνται από σημεία (points ή vertices) τα οποία συνδέονται μεταξύ τους με διάφορες γεωμετρικές οντότητες όπως τρίγωνα, γραμμές, τετράγωνα κτλ.. Η γραμμή που ενώνει δύο σημεία ονομάζεται edge, ενώ όταν ενωθούν δύο edges σχηματίζεται μια πλευρά (face). Μια πλευρά μπορεί να είναι ένα τρίγωνο (3 σημεία) ή ένα τετράγωνο (4 σημεία).



Εικόνα 2: Τα σημεία, οι πλευρές και οι γραμμές ενός κύβου

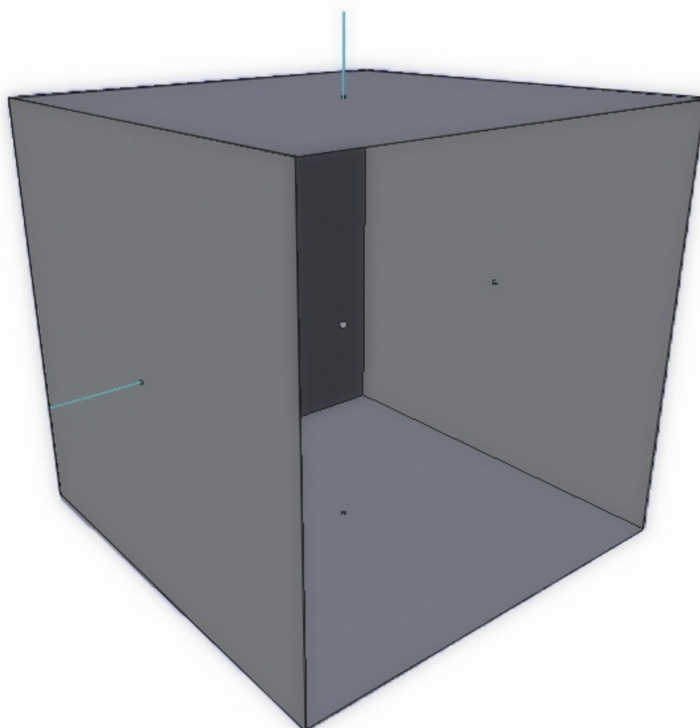
Τα τρισδιάστατα μοντέλα μπορούν να χωριστούν σε δύο κατηγορίες:

- **Solid**: Είναι περισσότερο ρεαλιστικά και δυσκολότερα στην κατασκευή τους, χρησιμοποιούνται κυρίως σε ιατρικές και μηχανικές προσομοιώσεις. Τα πιο απλά solid μοντέλα ονομάζεται primitives και είναι συνήθως κύβοι, κύλινδροι, σφαίρες, πυραμίδες και κώνοι.
- **Shell/boundary:** είναι ευκολότερα στην κατασκευή τους, σχεδόν όλα τα μοντέλα σε ηλεκτρονικά παιχνίδια και ταινίες είναι αυτού του τύπου. Τα μοντέλα shell συνθέτονται από δύο μέρη: Τη τοπολογία και τη γεωμετρία. Τα κύρια τοπολογικά αντικείμενα είναι οι πλευρές (faces), οι γραμμές-άκρες (edges) και οι κορυφές (vertices), ενώ τα κύρια γεωμετρικά αντικείμενα είναι οι επιφάνειες (surfaces), οι καμπύλες(curves) και τα σημεία(points).

2.2.1. Τρόποι μοντελοποίησης

Οι πιο δημοφιλείς τρόποι μοντελοποίησης είναι: polygonal modeling, nurbs modeling, spline and patches modeling και sculpt modeling.

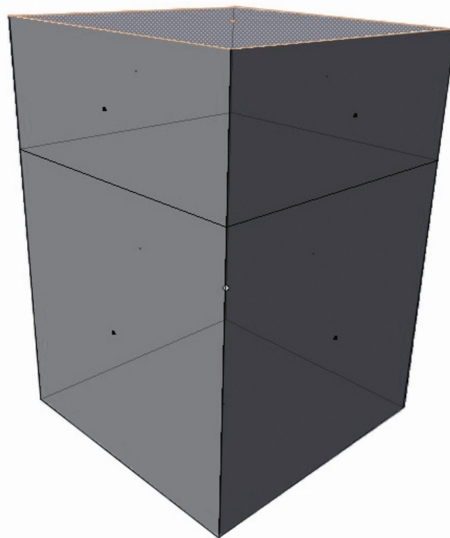
- **Polygonal modeling:** Με αυτό τον τρόπο μοντελοποίησης οι επιφάνειες των αντικειμένων αναπαρίστανται με πολύγωνα. Βασικό στοιχείο αποτελεί το vertex. Δύο vertices ενωμένα με μια γραμμή μετατρέπονται σε edge, ενώ τρία vertices ενωμένα από τρία edges μετατρέπονται σε τρίγωνο (triangle) το πιο απλό πολύγωνο. Τα τρίγωνα μαζί με τα πολύγωνα τεσσάρων edges (quads) είναι τα σχήματα που χρησιμοποιούνται περισσότερο σε αυτόν τον τρόπο μοντελοποίησης. Χαρακτηριστικό στοιχείο τους είναι το normal, ένα διάνυσμα δηλαδή κάθετο σε σχέση με την πλευρά ενός πολύγωνου. Κάθε πολύγωνο έχει δύο normals, ένα για κάθε πλευρά της επιφάνειας του. Στα περισσότερα συστήματα τρισδιάστατων γραφικών αναγνωρίζεται το ένα από τα δύο κάνοντας τη μια πλευρά να είναι ορατή ενώ την άλλη αόρατη (backface).



Εικόνα 3:Τα normals ζωγραφισμένα σ' ένα τρισδιάστατο μοντέλο. Παρατηρούμε ότι οι εσωτερικές πλευρές δεν έχουν ζωγραφισμένα normals

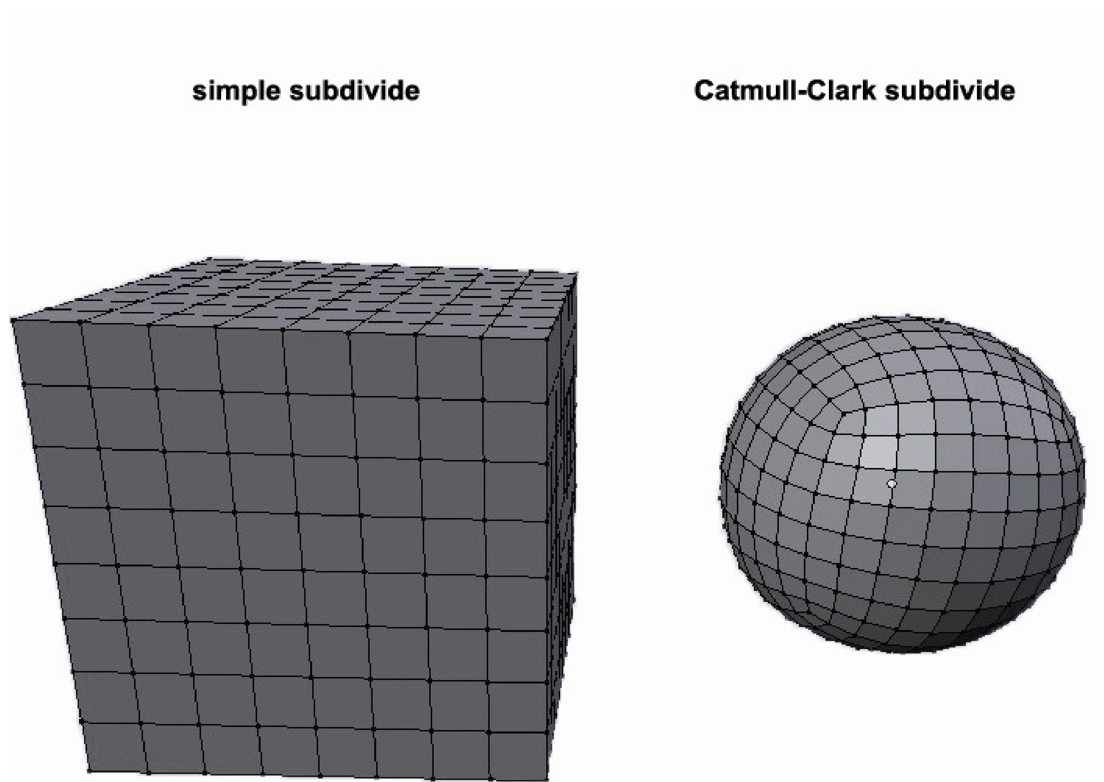
Μια ομάδα πολυγώνων που συνδέονται με κοινά vertices ονομάζεται mesh (Πλέγμα). Για να εμφανίζεται ένα mesh με σωστό τρόπο θα πρέπει τα πολύγωνά του να μην τέμνονται από άλλα πολύγωνα ή edges και να μην έχει διπλά vertices, edges ή faces.

Ένας από τους πιο δημοφιλής τρόπους δημιουργίας meshes είναι το box modeling. Στο box modeling το μοντέλο δημιουργείται τροποποιώντας primitives με εργαλεία όπως το extrude και το subdivide. Το extrude, όταν για παράδειγμα εφαρμόζεται πάνω σε μια πλευρά δημιουργεί μια καινούργια πλευρά ίδιου σχήματος και μεγέθους η οποία ενώνεται με το κάθε προϋπάρχον edge με μια πλευρά.



Εικόνα 4: Εφαρμογή του εργαλείου extrude σε έναν κύβο.

Το εργαλείο subdivide χρησιμοποιείται από τον 3D modeler για να υποδιαιρέσει τις πλευρές και τις γραμμές ενός μοντέλου με την προσθήκη νέων σημείων. Αυτό το εργαλείο μπορεί να κάνει πιο ομαλό ένα mesh(Catmull-Clark) .



Εικόνα 5: εφαρμογή του εργαλείου subdivide σε δύο κύβους.

Το box modeling είναι μια μέθοδος μοντελοποίησης γρήγορη και εύκολη στην εκμάθηση. Το αρνητικό του σημείο είναι ότι για να επιτευχθεί μεγάλη λεπτομέρεια και αληθοφάνεια σε ένα μοντέλο χρειάζεται πολύ εξάσκηση.

Μια ακόμη μέθοδος δημιουργίας polygon meshes είναι το extrusion modeling. Ο 3D modeler δημιουργεί ένα σχήμα δύο διαστάσεων που ακολουθεί τις γενικές γραμμές μιας φωτογραφίας ενός αντικειμένου. Στη συνέχεια κάνοντας το ίδιο σε άλλη φωτογραφία του αντικειμένου από άλλη οπτική γωνία μετατρέπει το σχήμα των δύο διαστάσεων σε τρισδιάστατο. Αυτή η μέθοδος χρησιμοποιείται για την δημιουργία προσώπων – κεφαλιών.

Άλλες μέθοδοι είναι, συνδυάζοντας primitives,προκαθορισμένα δηλαδή meshes από την 3D εφαρμογή, χρησιμοποιώντας 3D scanners που σαρώνουν ένα πραγματικό αντικείμενο και δημιουργούν ένα mesh μεγάλης λεπτομέρειας και μοντελοποίηση που βασίζεται σε σκετς(sketch based modeling).

Το κύριο μειονέκτημα του polygonal modeling είναι ότι χρειάζονται πολλά πολύγωνα για την αναπαράσταση καμπύλων επιφανειών, ενώ το κύριο

πλεονέκτημα είναι ότι είναι πιο γρήγορος τρόπος αναπαράστασης μοντέλων από τους άλλους.

- **NURBS modeling:** με αυτόν τον τρόπο μοντελοποίησης οι επιφάνειες καθορίζονται από spline curves που ρυθμίζονται από control points με τιμές βάρους. Όταν το βάρος μεγαλώνει η καμπύλη πλησιάζει το control point ενώ όταν μικραίνει απομακρύνεται. Το NURBS modeling δημιουργεί ομαλές επιφάνειας και είναι κατάλληλο για οργανικά μοντέλα.
- **Spline and Patches**: Όπως και στο NURBS modeling οι επιφάνειες καθορίζονται από καμπύλες. Σε θέματα ευκολίας και ευελιξίας αυτός ο τρόπος μοντελοποίησης βρίσκεται κάπου ανάμεσα στο NURBS και στο Polygonal modeling.
- **Sculpt**: Σχετικά νέος τρόπος μοντελοποίησης κατά των οποίο ένα αρχικό μοντέλο εισάγεται η δημιουργείται σε ένα πρόγραμμα sculpt. Στη συνέχεια χρησιμοποιείται το εργαλείο subdivide για να κάνει πιο ομαλό το μοντέλο αυξάνοντας τα πολύγωνα του. Έτσι επιτρέπει στον 3D artist να σκαλίσει σαν παραδοσιακός γλύπτης το μοντέλο με μεγάλη λεπτομέρεια. Τέλος δημιουργεί μια 32 bit εικόνα (normal map ή displacement map) που μεταφέρει όλες αυτές τις πληροφορίες και τις λεπτομέρειες του μοντέλου. Τοποθετώντας την εικόνα στο αρχικό μοντέλο έχει ένα μοντέλο λίγων πολυγώνων με λεπτομέρειες ενός μοντέλου πολύ περισσότερων.

2.2.2.Χρώμα

Αφού ο 3D modeler δημιουργήσει το mesh, έρχεται η σειρά ενός εξειδικευμένου καλλιτέχνη να του δώσει χρώμα και υφή, ο οποίος ονομάζεται texture artist. Ένας σχετικά οικονομικός τρόπος χρωματισμού όσον αφορά τους πόρους του Η/Υ είναι το vertex paint. Το κάθε vertex ενός mesh έχει από μια μεταβλητή χρώματος, έτσι ο texture artist βάφει το μοντέλο αλλάζοντας αυτές τις μεταβλητές. Το αρνητικό με αυτόν τον τρόπο χρωματισμού είναι ότι η λεπτομέρεια που μπορεί να επιτευχθεί περιορίζεται από τον αριθμό των vertices του mesh. Συνήθως σε ένα ηλεκτρονικό παιχνίδι τα meshes έχουν μικρό αριθμό πολυγώνων, επομένως αυτός ο τρόπος δεν είναι κατάλληλος. Για αυτόν τον λόγο αυτή η μέθοδος χρησιμοποιείται για παιχνίδια με έντονα χρώματα και «καρτουνίστικα» γραφικά.

Μια εναλλακτική μέθοδος του vertex paint είναι το texture paint. Με το texture paint η λεπτομέρεια που μπορεί να δοθεί είναι θεωρητικά απεριόριστη, εξαρτάται μόνο από την ανάλυση του texture. Πριν όμως ο texture artist μπορέσει να δημιουργήσει το texture map θα πρέπει πρώτα να «ξετυλίξει» το mesh(UV unwrap). Η διαδικασία αυτή ονομάζεται UV unwrapping. Στο UV unwrapping ο Η/Υ βοηθάει τον 3D artist με αλγόριθμους που ξετυλίγουν το mesh, διατηρώντας την αρχική γεωμετρική μορφή των πολυγώνων. Οι αλγόριθμοι μπορούν να βοηθήσουν πολύ σε αρχιτεκτονικές και σε απλές γεωμετρίες. Όταν πρόκειται όμως για οργανικά meshes ο Η/Υ θα υπολογίσει τις UV συντεταγμένες, αλλά θα χρειαστούν αρκετή τροποποίηση από τον texture artist. Έτσι θα δημιουργηθεί ένα UV map το οποίο μπορεί να αποθηκευθεί ως αρχείο εικόνας και στην συνέχεια να χρησιμοποιηθεί ως οδηγός για τον χρωματισμό της. Με λίγα λόγια το UV mapping είναι η διαδικασία κατά την οποία μια 2D εικόνα αναπαριστά ένα 3D μοντέλο. Το U και το V αποτελούν τις συντεταγμένες του μετασχηματισμένου αντικείμενου και διαφέρουν από τις X,Y,Z του αρχικού 3D μοντέλου. Αυτό δίνει την δυνατότητα στον καλλιτέχνη ζωγραφίζοντας την εικόνα να ζωγραφίζει τα πολύγωνα ενός 3D μοντέλου. Η εικόνα αυτή ονομάζεται UV texture map.

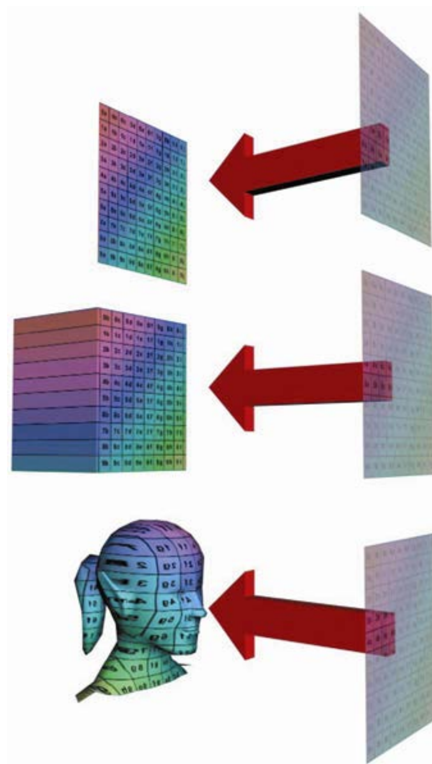
Η ζωγραφική ενός UV map μπορεί να μοιάζει με την παραδοσιακή ζωγραφική ενός πίνακα αλλά έχει κάποιες σημαντικές διάφορες . Ο καλλιτέχνης που ζωγραφίζει ένα UV map θα πρέπει παράλληλα να σκέφτεται πώς αυτό το UV map δένει με το mesh

στις τρεις διαστάσεις . Για να φαίνεται ο όγκος του mesh και να μην φαίνεται επίπεδο θα πρέπει στο texture map του να υπάρχουν πληροφορίες φωτισμού και σκίασης. Αν όμως αυτές οι πληροφορίες δεν δοθούν προσεκτικά μπορεί το mesh να μην ταιριάζει με τις συνθήκες φωτισμού του περιβάλλοντος.

2.2.2.1. Βασικοί τύποι UV mapping

Planar

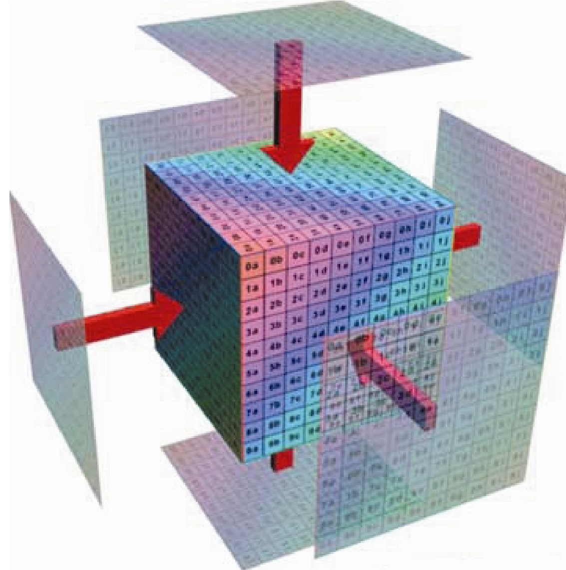
Το planar mapping λειτουργεί όπως ένας προτζέκτορας. Το texture προβάλλεται σε ένα 3D μοντέλο από μια κατεύθυνση. Αυτό ο τύπος mapping μπορεί να χρησιμοποιηθεί σε τοίχους και επίπεδες επιφάνειες αλλά όχι σε σύνθετα μοντέλα.



Εικόνα 6:planar mapping Luke Ahearn 3d game textures

Box

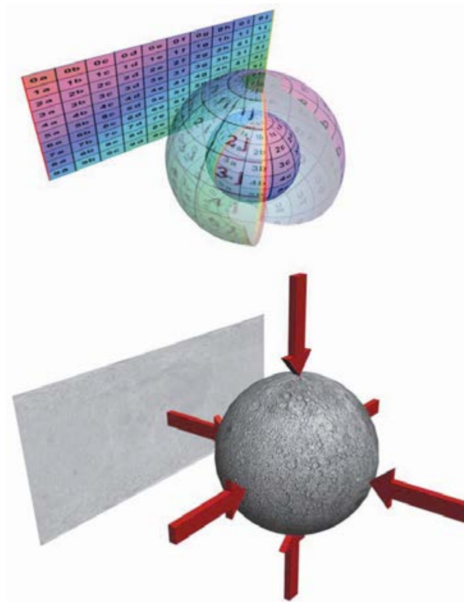
Στο box mapping το texture προβάλλεται στο μοντέλο από έξι μεριές.



Εικόνα 7:box mapping Luke Ahearn 3d game textures

Spherical

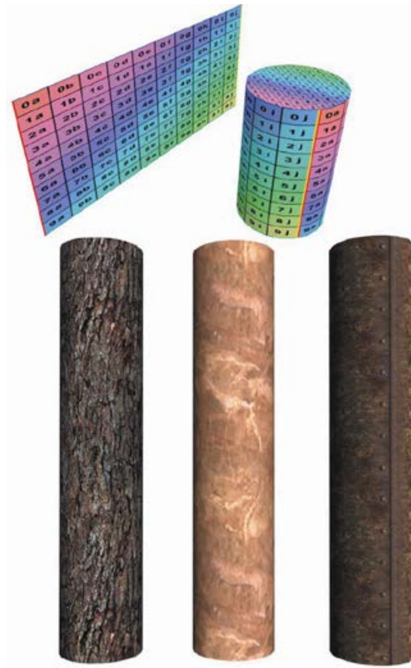
Το spherical mapping προβάλλει το texture στο μοντέλο από όλες τις μεριές με σφαιρικό τρόπο . Είναι ιδανικό για σφαιρικά αντικείμενα όπως π.χ. πλανήτες.



Εικόνα 8:spherical mapping Luke Ahearn 3d game textures

Cylindrical

Το cylindrical mapping προβάλλει το texture καλύπτοντας κυλινδρικά το μοντέλο. Μπορεί να χρησιμοποιηθεί σε αντικείμενα όπως δέντρα κολόνες κ.τ.λ..



Εικόνα 9: cylindrical mapping Luke Ahearn 3d game textures

2.2.2.2.Η δύναμη του δύο στα textures

Υπάρχουν διάφοροι παράμετροι και κανόνες που πρέπει να ακολουθήσει ο texture artist. Ο πιο γενικός- βασικός κανόνας είναι ότι τα textures πρέπει να δημιουργούνται σε δύναμη του δύο όσον αφορά το μέγεθος τους. Αυτό σημαίνει ότι μια εικόνα θα πρέπει να έχει συγκεκριμένο αριθμό pixels για ύψος και για πλάτος. Αναλύσεις που μπορεί να έχουν είναι:

- 16x16
- 32x32
- 64x64
- 128x128
- 256x256

- 512x512
- 1024x1024
- 2048x2048

Πριν μερικά χρόνια ένα texture 512x512 θεωρούνταν μεγάλο, στα σημερινά παιχνίδια χρησιμοποιούνται textures μεγέθους 1024x1024 και 2048x2048. Τα textures πρέπει να έχουν μέγεθος δύναμης του δύο εξαιτίας των game engines και των καρτών γραφικών. Οι περισσότερες game engines δέχονται και ορθογώνια μεγέθη όπως για παράδειγμα 256x512, 128x1024 κ.τ.λ.. Αν ένα texture έχει λάθος παραμέτρους τότε η game engine μπορεί να παρουσιάσει μήνυμα σφάλματος ή να κολλήσει τελείως. Κάποιες game engines σε τέτοιες περιπτώσεις τροποποιούν το texture αλλάζοντας του μέγεθος, βάθος χρώματος, format κ.λπ..

2.2.3.Η τεχνολογία των shaders για τους 3D artists

Οι shaders είναι προγράμματα που προσφέρουν ρεαλισμό στα ηλεκτρονικά παιχνίδια δημιουργώντας γραφικά εφέ σε πραγματικό χρόνο. Υπάρχουν δύο είδη shader για την κάρτα γραφικών (GPU) : οι vertex και οι pixel shaders. Οι vertex shaders διαχειρίζονται γεωμετρίες ενώ οι pixel shaders ρενταρισμένα pixel σε πραγματικό χρόνο. Οι shaders μπορούν να χρησιμοποιηθούν για να δημιουργήσουν περίπλοκα υλικά (materials) και για εφέ στην εικόνα: μαλλιά, φωτιά, νερό, ανακλάσεις κ.λπ..

Οι καλλιτέχνες συνήθως δημιουργούν δεδομένα εισόδου και κρίνουν το τελικό αποτέλεσμα που παράγει ο shader. Στις περισσότερες περιπτώσεις δεν ασχολούνται με το προγραμματιστικό μέρος αλλά υπάρχουν φορές που ένας καλλιτέχνης μπορεί να τροποποιήσει έναν shader. Τα σύγχρονα εργαλεία δημιουργίας shader δεν χρειάζονται προγραμματιστικές γνώσεις και ο τρόπος δημιουργίας τους μοιάζει με τον τρόπο δημιουργίας ενός material σε ένα πρόγραμμα 3D γραφικών όπως το 3ds max, to Maya κ.λπ..

Ο texture artist εκτός από τις 2D εικόνες που πρέπει να δημιουργήσει ως δεδομένα εισόδου για ένα shader θα πρέπει να κατανοήσει ως ένα βαθμό τον τρόπο λειτουργίας του. Υπάρχουν εφέ σε ηλεκτρονικά παιχνίδια όπου ο texture artist δεν

ζωγραφίζει απλά ένα texture, αλλά δημιουργεί μια σειρά από textures τα οποία συνδυάζονται μεταξύ τους για να εμφανιστεί το κατάλληλο αποτέλεσμα. Για να μπορεί να χρησιμοποιεί αποτελεσματικά τους shaders θα πρέπει να αναπτύξει την ικανότητα να εντοπίζει το υλικό (material) ενός αντικειμένου χωρίς να υπολογίζει τις επιρροές που δέχεται από το περιβάλλον. Τέτοιες επιρροές μπορεί να είναι φθορά του υλικού, επιρροές από τον φωτισμό κ.λπ.. Με αυτό τον τρόπο μπορεί να δημιουργήσει βασικά υλικά και στη συνέχεια να τα τροποποιήσει από ένα δίκτυο άλλων textures, δημιουργώντας έτσι το τελικό υλικό που μπορεί με την χρήση ενός shader να αλλάζει σε πραγματικό χρόνο.

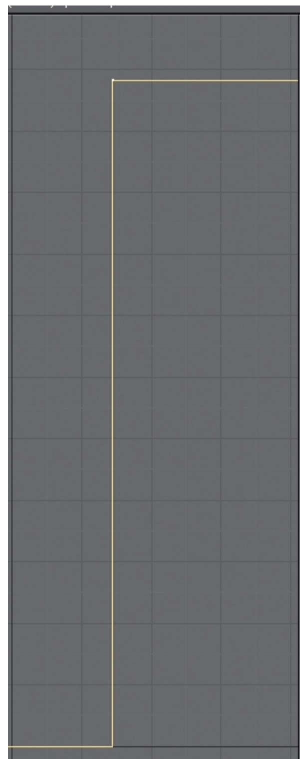
Οι 2D εικόνες που δημιουργούνται από τον texture artist μπορεί να είναι diffuse map, specular map, ambient occlusion map, normal map, bump map, height map, displacement map. Το diffuse map είναι μια 2D εικόνα που καθορίζει το χρώμα και την ένταση του φωτός που ανακλάται πίσω όταν φωτίζει μία επιφάνεια. Το specular map είναι μια 2D εικόνα κλίμακας του γκρι, επηρεάζει την ένταση της ανακλαστικότητας. Το ambient occlusion map είναι μια εικόνα που προσομοιώνει την σκίαση στο diffuse map. Καθιστά τις σκοτεινές περιοχές ενός 3D μοντέλου. Ένα bump map είναι μια 2D εικόνα που δημιουργείται από τον texture artist και κάνει μια επιφάνεια από επίπεδη ανώμαλη. Είναι και αυτή κλίμακας του γκρι όπου οι φωτεινές περιοχές είναι «σηκωμένες» ενώ οι πιο σκοτεινές είναι «κατεβασμένες». Με το bump mapping μπαίνει λεπτομέρεια στο μοντέλο χωρίς να χρειαστεί να αυξηθούν τα πολύγωνα του. Το normal map είναι μια παραλλαγή του bump map και χρησιμοποιείται κυρίως στα ηλεκτρονικά παιχνίδια. Ο texture artist δημιουργεί μια εικόνα κλίμακας του γκρι που σε τελικό στάδιο επεξεργασίας μετατρέπεται σ' ένα κόκκινο- πράσινο normal map. Ένα normal map περιέχει περισσότερες πληροφορίες από ένα bump map και μπορεί να κάνει μια επιφάνεια αρκετά πιο περίπλοκη. Το height map είναι μια raster εικόνα που χρησιμοποιείται για την αποθήκευση τιμών, όπως δεδομένα ανύψωσης μιας επιφάνειας για απεικόνιση σε έναν Η/Υ. Μια height map εικόνα μπορεί να χρησιμοποιηθεί στο bump mapping για να υπολογιστεί, που θα δημιουργηθούν σκιές σε ένα material, στο displacement mapping για να εκτοπίσει(displace) την γεωμετρική θέση των σημείων στην textured επιφάνεια και σε terrain όπου μετατρέπεται σε 3D mesh. Ένα height map περιέχει ένα κανάλι που ερμηνεύεται ως «ύψος » από το «πάτωμα» μιας επιφάνειας και οπτικοποιείται ως μια ασπρόμαυρη εικόνα όπου το μαύρο αντιπροσωπεύει το ελάχιστο ενώ το άσπρο το μέγιστο ύψος .

2.2.4.Λίγα λόγια για το animation

Με την ολοκλήρωση του 3D μοντέλου (μοντελοποίηση-texturing) έρχεται η σειρά ενός εξειδικευμένου καλλιτέχνη να του δώσει κίνηση (animation). Στο animation ο χρόνος μετριέται σε frames. Για να γίνει οποιοδήποτε transformation(αλλαγή στην τοποθεσία, στο μέγεθος, στην περιστροφή) θα πρέπει το αντικείμενο να έχει ένα αρχικό σημείο (frame) και ένα άλλο σημείο που θα δείχνει πώς το αντικείμενο μετασχηματίζεται(transformed). Τα σημεία αυτά ονομάζονται key frames και καθορίζονται χειροκίνητα από τον animator.

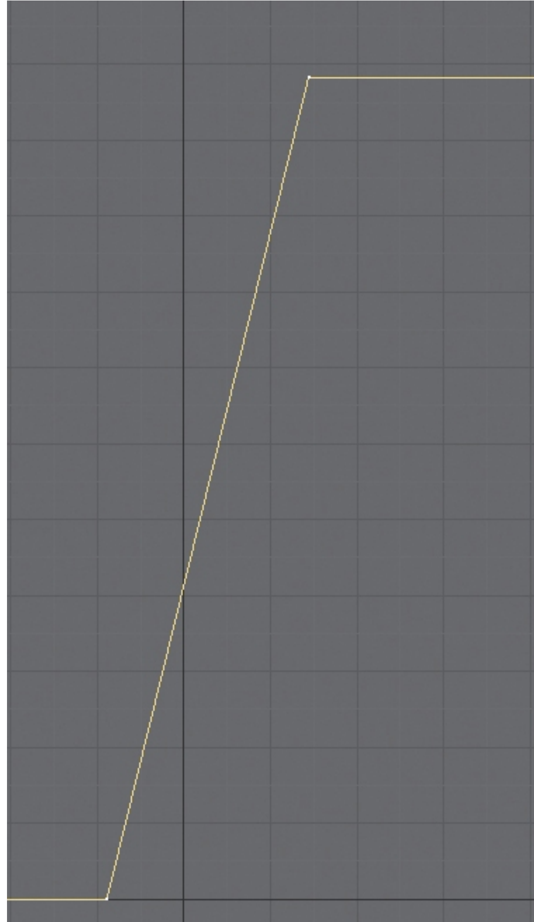
Όμως τι γίνεται μεταξύ των key frames, πώς ο Η/Υ ελέγχει τον μετασχηματισμό του αντικειμένου; Η απάντηση βρίσκεται στο interpolation (παρεμβολή). Υπάρχουν τρία είδη interpolation που χρησιμοποιούνται για να καθορίσουν τον μετασχηματισμό μεταξύ των key frames [16].

Η πρώτη είναι η constant interpolation. δεν υπάρχει πραγματική κίνηση. Το αντικείμενο θα μετασχηματιστεί ακαριαία χωρίς ενδιάμεσες τιμές



Εικόνα 10: y=b

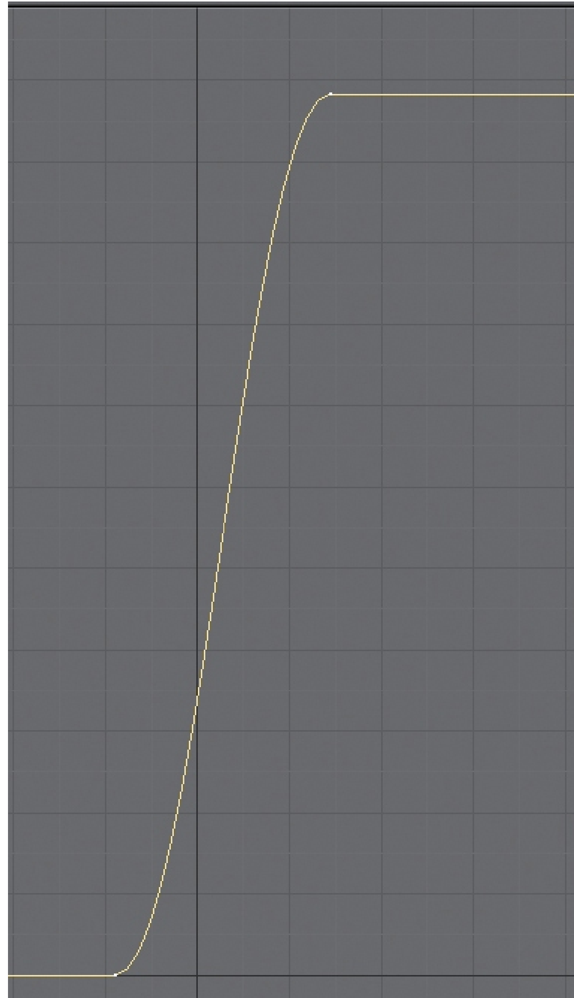
Η δεύτερη μέθοδος interpolation ονομάζεται linear interpolation (γραμμική παρεμβολή) . Η linear interpolation διαφέρει από την constant στο ότι οποιοσδήποτε μετασχηματισμός δεν γίνεται ακαριαία αλλά γραμμικά.



Εικόνα 11: y=ax+b

Η τρίτη μέθοδος ονομάζεται Bezier interpolation. Είναι προτιμότερη μέθοδος λόγω της ομαλότητας και της συνέχειας της. Μιμείται φυσικά φαινόμενα όπως είναι η επιτάχυνση, το βάρος και η αδράνεια. Για παράδειγμα έχουμε μια μπάλα η οποία πέφτει συγκρούεται με το έδαφος και ξανανεβαίνει. Με Bezier interpolation η κίνηση θα ήταν όλο και πιο γρήγορη μέχρι την σύγκρουση και στην συνέχεια θα γινόταν όλο και πιο αργή μέχρι να φθάσει στο ψηλότερο σημείο. Με linear interpolation η ταχύτητα της μπάλας θα ήταν πάντα σταθερή ενώ με constant interpolation η μπάλα

θα έμενε ακίνητη στον αέρα έπειτα θα τηλεμεταφερόταν στο έδαφος και στην συνέχεια θα τηλεμεταφερόταν πάλι στον αέρα.



Εικόνα 12 : y=ax^2+bx+c

2.2.4.1. Skeletal animation

Ο animator για να μπορέσει να κινήσει έναν χαρακτήρα θα πρέπει να μπορέσει να κινήσει τα vertices του. Για να το κάνει αυτό δημιουργεί έναν σκελετό χωρίζοντας έτσι τα vertices σε ομάδες ώστε να μπορεί να τα επιλέξει και να τα διαχειριστεί

(rigging). Σε χαρακτήρες όπως άνθρωποι και ζώα κάποια από τα οστά του ψηφιακού σκελετού αντιστοιχούν σε οστά του πραγματικού. Τα οστά ενός σκελετού έχουν ιεραρχικές σχέσεις μεταξύ τους (parent-child). Τα οστά που είναι «παιδιά» ενός οστού «γονέα» μετασχηματίζονται όταν μετασχηματίζεται ο «γονέας», επίσης μπορούν να μετασχηματίζονται και ανεξάρτητα.

Εφόσον κατασκευαστεί ο σκελετός έρχεται η σειρά της κίνησης του χαρακτήρα. Ο animator θα πρέπει να τοποθετήσει τα οστά του σκελετού ώστε να πάρει διάφορες πόζες. Στην συνέχεια θα πρέπει να βάλει keyframes για κάθε οστό σε κάθε πόζα. Ο Η/Υ είναι υπεύθυνος για τους μετασχηματισμούς των οστών μεταξύ των keyframes.
Η αδυναμία του skeletal animation είναι ότι δεν μπορεί να εξομοιώσει ρεαλιστικά τις κινήσεις των μυών και του δέρματος, γι' αυτό ο animator δημιουργεί ξεχωριστούς ελεγκτές δέρματος και μυών.

Εκτός από την παραπάνω διαδικασία που ονομάζεται keyframing υπάρχει και μια νέα μέθοδος κίνησης χαρακτήρα που ονομάζεται motion capture. Στο motion capture πραγματικοί ηθοποιοί κάνουν διάφορες κινήσεις οι οποίες καταγράφονται από έναν Η/Υ και στη συνέχεια εφαρμόζονται στον σκελετό του χαρακτήρα.

Μερικά από τα πλεονεκτήματα της μεθόδου είναι ότι περίπλοκες κινήσεις όπως για παράδειγμα το βάρος ενός ανθρώπου μπορούν να δημιουργηθούν με αληθοφανή τρόπο. Είναι πιο γρήγορος και πιο φθηνός τρόπος απ ότι το keyframe animation. Μερικά από τα μειονεκτήματα του είναι ότι χρειάζεται ειδικό hardware και ειδικά προγράμματα. Το κόστος του εξοπλισμού είναι απογοητευτικό για μικρές παραγωγές. Κινήσεις που ξεφεύγουν από τους νόμους της φύσης δεν μπορούν να καταγράφουν.

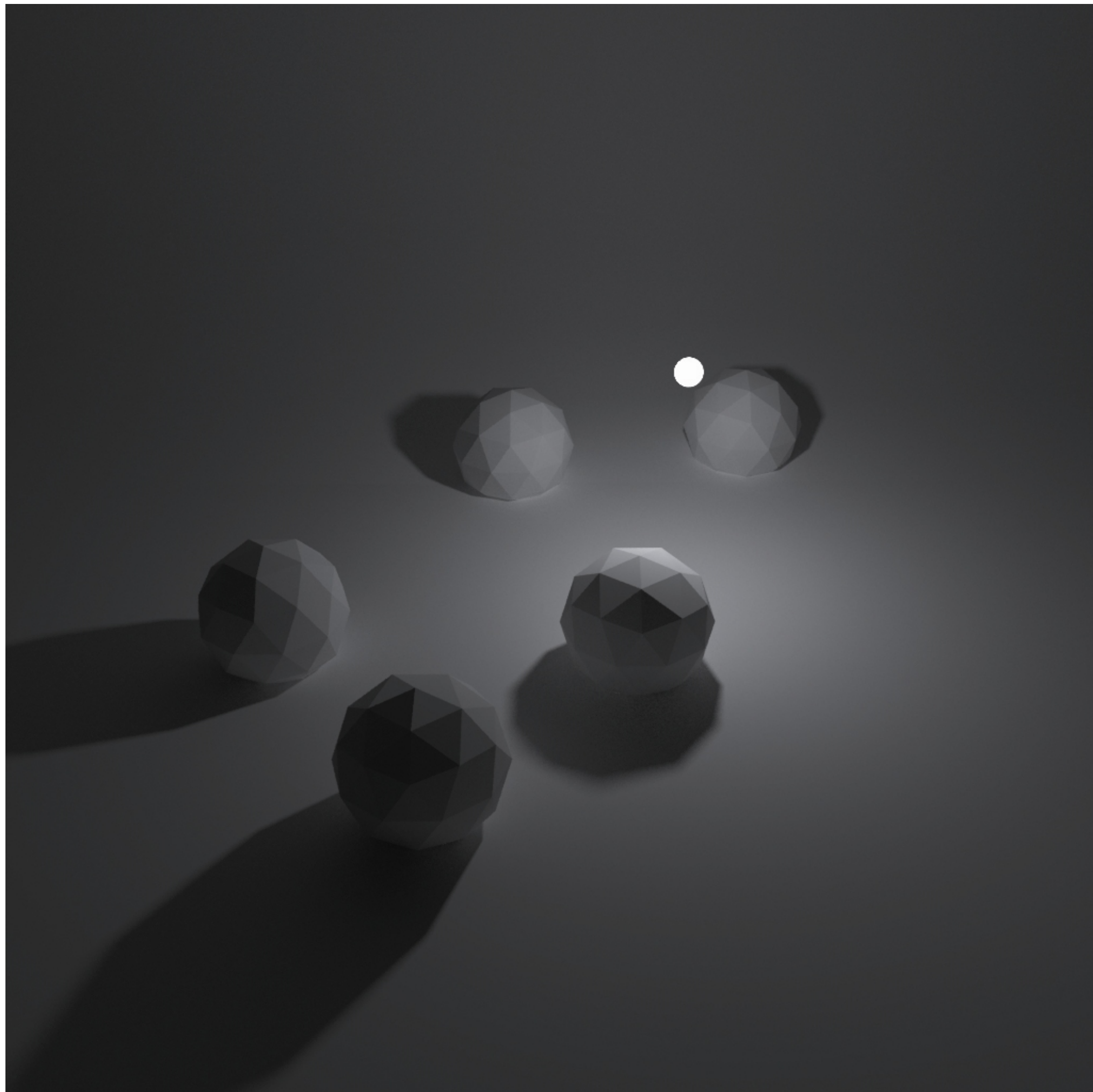
2.2.5. Rendering lighting

Ο 3D artist για να δημιουργήσει μια εικόνα με "σωστό" τρόπο εκτός από το modeling-texturing θα πρέπει να είναι ιδιαίτερα προσεκτικός στο lighting-rendering. Το rendering είναι η διαδικασία κατά την οποία παράγεται μια 2D εικόνα από μια 3D σκηνή. Οι 3D σκηνές μπορούν να ρενταριστούν με διάφορα στυλ π.χ. με ρεαλιστικό ή με καρτουνίστικο τρόπο. Όποιο στυλ και να διαλέξει ο 3D artist θα πρέπει να χρησιμοποιήσει τον κατάλληλο φωτισμό [10].

2.2.5.1. Τύποι φώτων

Point light

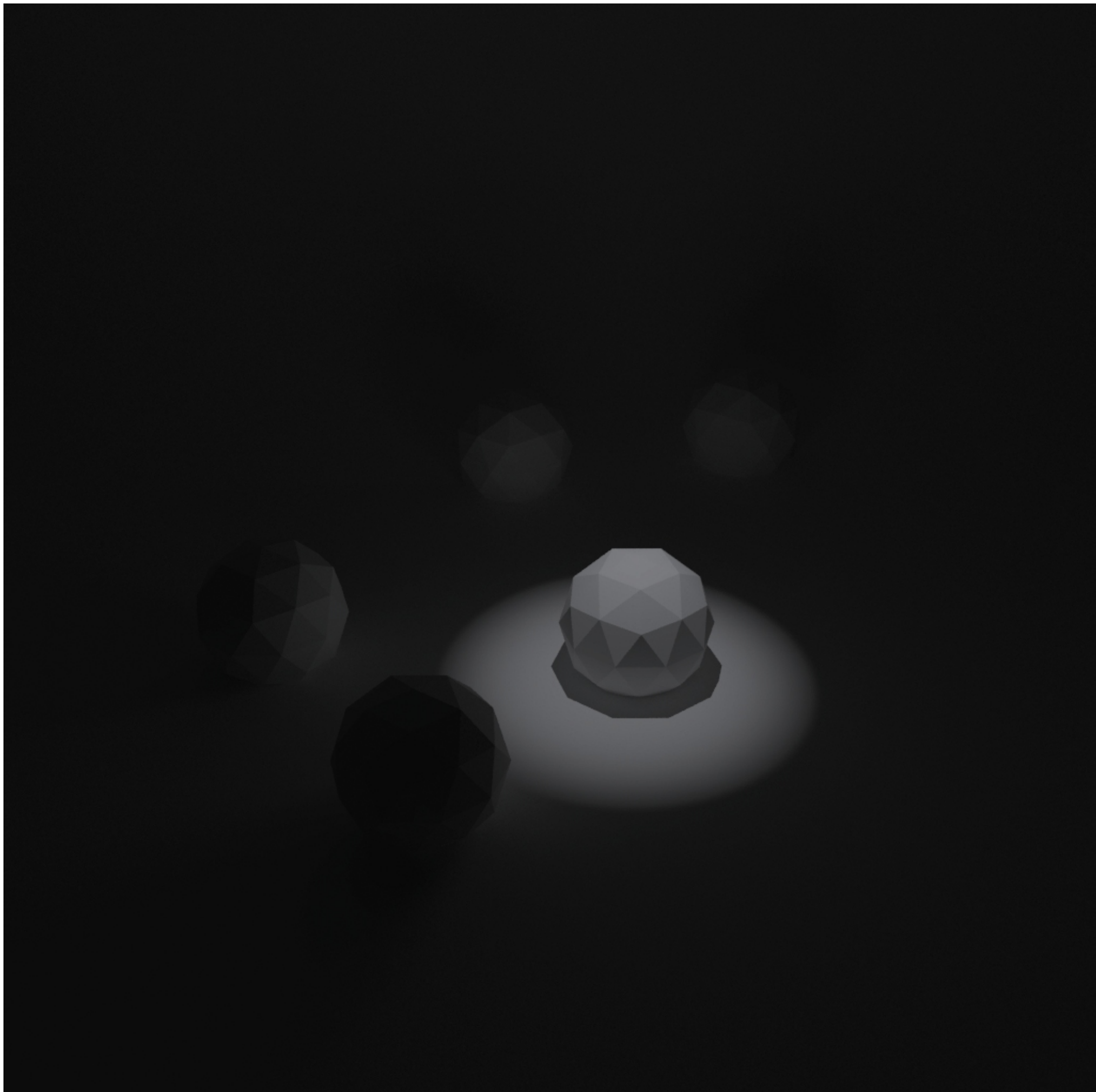
Τα φώτα τύπου point είναι γνωστά και ως omni. Είναι η πιο απλή μορφή φωτισμού σε μια 3D εφαρμογή και εκπέμπουν φώς προς όλες τις κατευθύνσεις .



Εικόνα 13 : Φωτισμός με ένα φώς τύπου point

Spot light

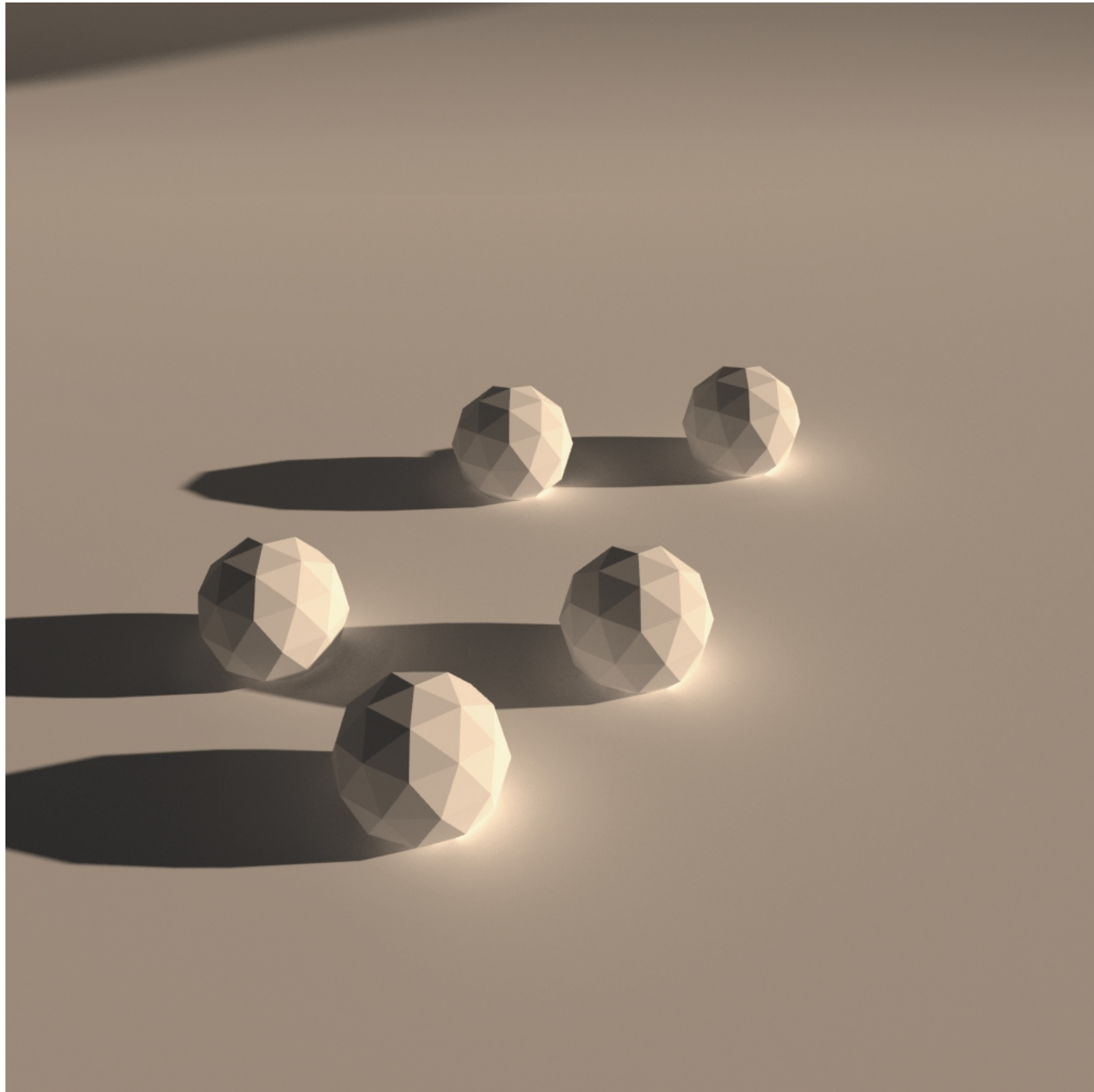
Τα φώτα τύπου spot είναι ο πιο δημοφιλής τύπος φωτισμού στα τρισδιάστατα γραφικά, γιατί επιτρέπουν στον καλλιτέχνη τον πλήρη έλεγχο. Τα spot lights, όπως και τα omni lights εκπέμπουν φώς από ένα σημείο , αλλά σε αντίθεση με αυτά δεν το εκπέμπουν προς όλες τις κατευθύνσεις. Η εκπομπή φωτός περιορίζεται σε μια ακτίνα ή σε έναν κώνο.



Εικόνα 14: Βλέπουμε την ακτίνα ενός spot light

Directional light

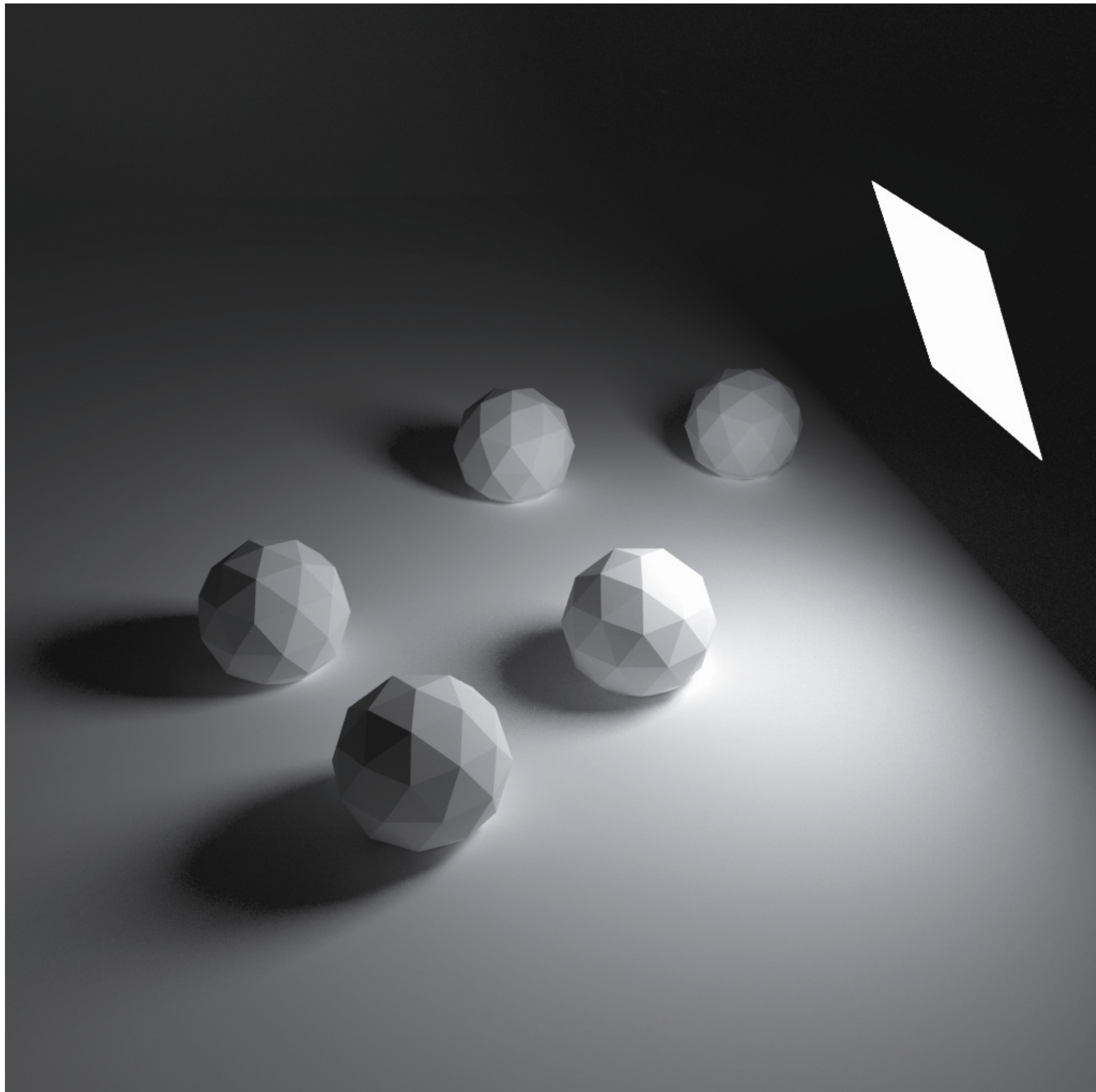
Ένα φώς τύπου directional μπορεί να χρησιμοποιηθεί ώστε να προσομειώσει το φώς του ηλίου. Είναι γνωστό ως distant, infinite, direct, sunlight. Ένα directional φώς φωτίζει όλα τα αντικείμενα μιας σκηνής από την ίδια γωνία χωρίς να παίζει ρόλο η θέση ενός αντικειμένου σε σχέση με το φώς. Οι σκιές που δημιουργεί είναι παράλληλες μεταξύ τους όπως ακριβώς γίνεται με μια πολύ μακρινή πηγή φωτός.



Εικόνα 15 : Ένα directional light δημιουργεί παράλληλες σκιές

Area light

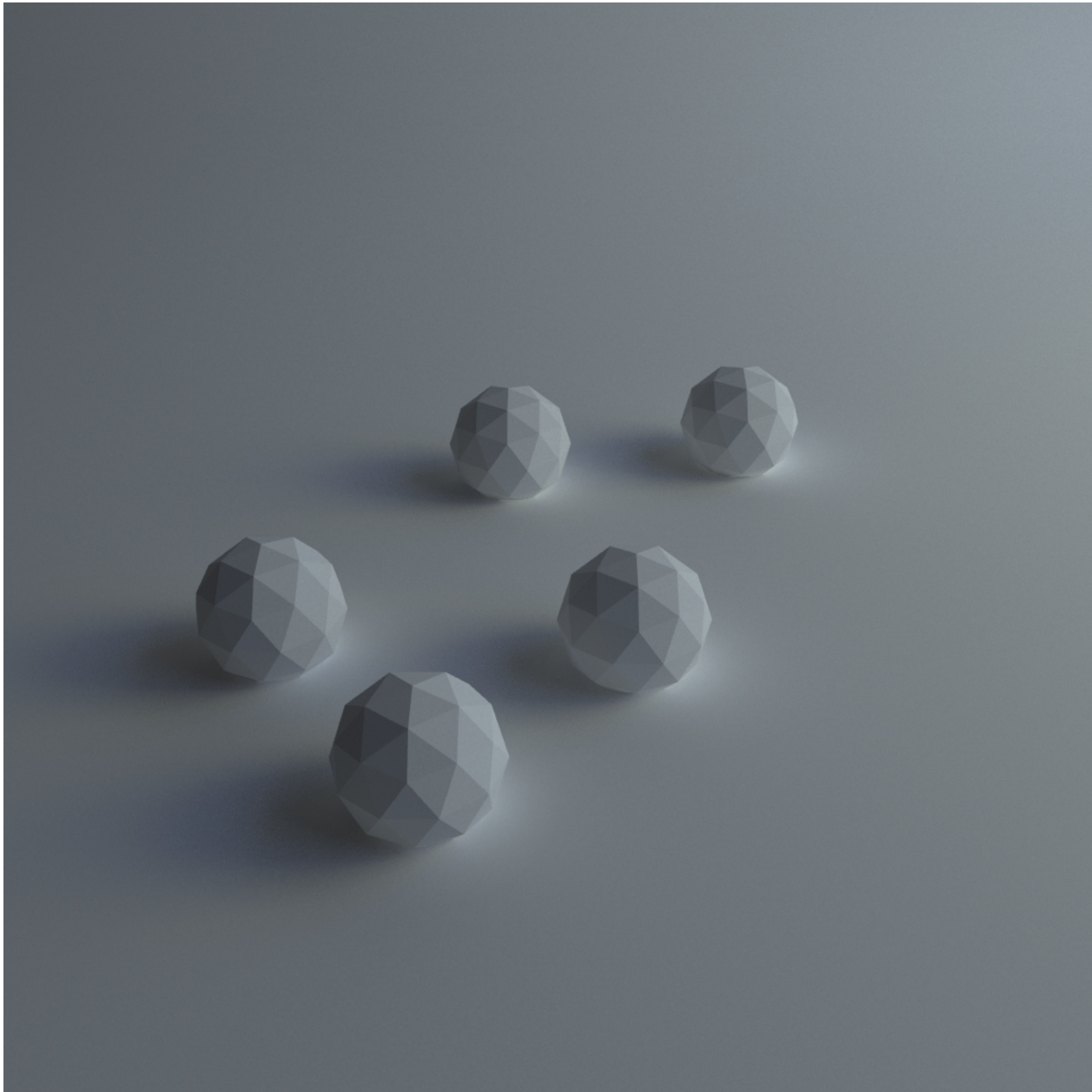
Τα φώτα τύπου area προσομοιώνουν το μέγεθος μιας φυσικής πηγής φωτός. Τα omni, directional, spot φώτα αν αλλάξουν μέγεθος σε μια σκηνή δεν θα αλλάξει ο φωτισμός. Αν όμως αλλάξει το μέγεθος ενός area φωτός θα αλλάξει ο φωτισμός και θα φαίνεται πώς έρχεται από μια πηγή διαφορετικού μεγέθους. Η ποιότητα της φωτοσκίασης με αυτά τα φώτα αποτελεί πολύ καλή επιλογή για αληθοφανή renderings, αλλά και χρονοβόρα για animation μεγάλης διάρκειας.



Εικόνα 16: Φωτισμός από area light

Sky dome

Το sky dome είναι μια ειδική πηγή φωτός που περιβάλλει μια σκηνή και την φωτίζει απ' όλες τις πλευρές. Το sky dome εξομοιώνει πολύ καλά το φως που προέρχεται από τον ουρανό. Χρησιμοποιείται επίσης ως δευτερεύον φώς για να συμπληρώσει τον κυρίως φωτισμό της σκηνής.



Εικόνα 17: Μία σφαίρα περικλείει τη σκηνή και την φωτίζει ανάλογα με τον τρόπο που είναι βαμμένη

Μοντέλα που χρησιμοποιούνται ως φώτα

Κάθε renderer που υποστηρίζει global illumination(δεν λαμβάνεται υπόψη μόνο το φώς που προέρχεται απευθείας από μια πηγή, αλλά και οι ακτίνες της ίδιας πηγής που ανακλώνται από άλλες επιφάνειες της σκηνής) επιτρέπει να φωτίζονται αντικείμενα από άλλα αντικείμενα. Όταν στο rendering χρησιμοποιείται global illumination ένα αντικείμενο με πολύ λαμπερό χρώμα μπορεί να λειτουργήσει ως πηγή φωτός. Δεν είναι αποτελεσματική λύση φωτισμού γιατί είναι χρονοβόρα γι' αυτό και αποφεύγεται.

Εικόνα 18: Ένα αντικείμενο που λειτουργεί ως πηγή φωτός

2.2.5.2. Shadows

Το «στήσιμο» των σκιών χρειάζεται τόση προσοχή και χρόνο όση και το στήσιμο των φώτων. Οι σκιές μπορούν να βελτιώσουν την τελική σύνθεση. Η επιλογή των κατάλληλων αλγορίθμων σκίασης και η κατάλληλη ρύθμιση τους είναι σημαντική για τον χρόνο που θα διαρκέσει το rendering.

Shadow algorithms

Τα προγράμματα rendering συνήθως επιτρέπουν την επιλογή μεταξύ αλγορίθμων σκίασης :

- **Depth map ή shadow map**: Είναι ο γρηγορότερος τρόπος rendering σκιών και χρησιμοποιείται σε επαγγελματικούς φωτισμούς film. Πριν αρχίσει το rendering υπολογίζεται ένα depth map για κάθε φώς, το οποίο καθορίζει που θα ρενταριστούν οι σκιές. Για κάθε κατεύθυνση που φωτίζει ένα φώς, το depth map αποθηκεύει την απόσταση από το φώς στο κοντινότερο αντικείμενο που παράγει σκιά. Κατά την διάρκεια του rendering το φώς θα σταματήσει στο διάστημα που καθορίζεται από το depth map έτσι ώστε να μην φωτίζει πέρα από το διάστημα που αποθηκεύτηκε για κάθε κατεύθυνση.
- **Raytraced shadows:** Οι σκιές raytraced υπολογίζονται ανιχνεύοντας ακτίνες φωτός μεταξύ μιας πηγής φωτός και ενός φωτισμένου αντικειμένου. Αυτό γίνεται κατά την διάρκεια του rendering σε αντίθεση με τις depth map σκιές που αποθηκεύονται σε ένα depth map.

  Μερικά από τα πλεονεκτήματά τους είναι:

  - Γίνονται πιο «ελαφριές» όταν φωτίζονται διαφανή αντικείμενα παίρνοντας καμία φορά λίγο από το χρώμα των αντικειμένων.
  - Αποδίδουν με πολύ ρεαλιστικό τρόπο για όλους τους τύπους φώτων.

  Μερικά από τα μειονεκτήματα τους είναι:

  - Χρειάζονται περισσότερο χρόνο στο rendering από ότι οι depth map σκιές. Για περίπλοκες σκηνές η διαφορά στον χρόνο μπορεί να είναι πολύ μεγάλη.
  - Χρησιμοποιούν πολύ μνήμη πράγμα που πρακτικά περιορίζει την πολυπλοκότητα της σκηνής που μπορεί να ρενταριστεί.

2.3. Γενικά περί προγραμματισμού

Οι δύο γλώσσες που χρησιμοποιούν οι προγραμματιστές κυρίως, για την ανάπτυξη παιχνιδιών είναι η C# (C-sharp) και η C++ (C-Plus Plus) και πρόκειται για γλώσσες αντικειμενοστραφούς προγραμματισμού (OOP-Object Oriented Programming). Οι δύο αυτές γλώσσες είναι αρκετά όμοιες συντακτικά και είναι επίσης αρκετά «ευέλικτες» στη χρήση τους (αυτός είναι κι ένας λόγος για τον οποίο χρησιμοποιούνται στην ανάπτυξη παιχνιδιών).

Μια μηχανή παιχνιδιού (game engine), είναι ένα πρόγραμμα το οποίο έχει όλη τη λειτουργικότητα του προβλεπόμενου είδους παιχνιδιού, αλλά χρειάζεται ένα «παιχνίδι» να «χτιστεί» πάνω της. Είναι σαν ένας «σκελετός» και το παιχνίδι είναι κατά κάποιο τρόπο σαν το «δέρμα» που είναι πάνω του. Πιο συγκεκριμένα μια μηχανή παιχνιδιού είναι ο κώδικας μέσα σε ένα παιχνίδι που έχει να κάνει με «απεικόνιση γραφικών (rendering)», «χειρισμό εισόδου (input handling)», «αναπαραγωγή ήχου (playing audio)», «εφαρμογή φυσικής (implementing physics)» και με «προσομοίωση τεχνητής νοημοσύνης (simulating artificial intelligence)» ή «λογική παιχνιδιού (game logic)».

Οι μηχανές παιχνιδιών είναι αρκετά «επαναχρησιμοποιήσιμες» και συνήθως καλύπτουν προγραμματιστικά περίπου το 90% του παιχνιδιού, οδηγώντας σε πολύ γρήγορη ανάπτυξη παιχνιδιών.

Επίσης οι μηχανές παιχνιδιών μπορεί να είναι αρκετά περίπλοκες ανάλογα με τα χαρακτηριστικά που διαθέτουν και μπορούν να προσφέρουν. Μερικές μηχανές παιχνιδιών είναι :

- Valve game engine
- TorqueX game engine
- Unreal game engine
- Quake game engine
- Havok game engine
- Irrlicht game engine
- CryEngine game engine

2.3.1. Μηχανή απεικόνισης γραφικών (rendering engine)

Rendering είναι η διαδικασία παραγωγής μιας εικόνας από ένα μοντέλο, μέσω των προγραμμάτων Η/Υ. Το μοντέλο είναι μια περιγραφή από τρισδιάστατα αντικείμενα σε μια αυστηρά καθορισμένη γλώσσα η δομή δεδομένων. Μπορεί να περιέχει γεωμετρία (geometry), άποψη (viewpoint), υφή (texture), φωτισμό (lightning) και πληροφορία απόχρωσης (shading information). Η εικόνα είναι μια ψηφιακή εικόνα ή raster graphics image. To "rendering" χρησιμοποιείται επίσης για να περιγράφει τη διαδικασία υπολογισμού των εφέ (effects) σε ένα αρχείο βίντεο για την παραγωγή του τελικού βίντεο. Είναι μια πάρα πολύ σημαντική διαδικασία και συνδέεται και με τις υπόλοιπες στην πράξη.

Στο graphics pipeline αποτελεί το τελευταίο σημαντικό βήμα δίνοντας την τελική εμφάνιση στα μοντέλα και στη σχεδιοκίνηση. Ο renderer είναι ένα προσεκτικά σχεδιασμένο πρόγραμμα, βασισμένο σε ένα επιλεκτικό μείγμα των κλάδων που σχετίζονται με: φυσική του φωτός (light physics), οπτική ανάπτυξη (visual perception), μαθηματικά (mathematics) και ανάπτυξη λογισμικού (software development). Στα τρισδιάστατα γραφικά έχουμε κυρίως δυο ειδών rendering, το pre-rendering και το real time rendering. To pre-rendering είναι μια εντατική υπολογιστική διαδικασία που χρησιμοποιείται για τη δημιουργία ταινίας, ενώ το real time rendering χρησιμοποιείται σε τρισδιάστατα ηλεκτρονικά παιχνίδια και βασίζεται στη χρήση των καρτών γραφικών με 3D hardware accelerators.

Όταν το pre-image έχει ολοκληρωθεί, το rendering χρησιμοποιείται για να προσθέσει bitmap textures ή procedural textures, φώτα, bump mapping και σχετική θέση με τα άλλα αντικείμενα. Το αποτέλεσμα είναι μια ολοκληρωμένη εικόνα που βλέπει ο θεατής.

Μερικά χαρακτηριστικά που περιλαμβάνει αυτή η εικόνα (και που πρόκειται για χαρακτηριστικά που περιλαμβάνει μια rendering engine) είναι:

- ***Shading*** – πως το χρώμα και η φωτεινότητα μιας επιφάνειας ποικίλει ανάλογα με το φως
- ***Texture mapping*** – μια μέθοδος προσθήκης λεπτομέρειας σε επιφάνειες
- ***Bump mapping*** – μια μέθοδος προσομοίωσης μικρής κλίμακας «ανομαλότητας» σε επιφάνειες
- ***Fogging/participating medium*** – πώς το φως διαχέεται (dims) όταν περνάει από μη-καθαρή ατμόσφαιρα ή αέρα (αρμοδιότητα του φωτός)
- ***Shadows*** – το φαινόμενο της παρεμπόδισης του φωτός
- ***Soft shadows*** – κυμαινόμενη σκίαση που προκαλείται από εν μέρει παρεμποδιζόμενες πηγές φωτός
- ***Reflection*** – αντανάκλαση
- ***Transparency (optics), transparency (graphic) or opacity*** – μετάδοση του φωτός μέσα από αντικείμενα.
- ***Translucency*** – διαύγεια του αντικειμένου
- ***Refraction*** – διάθλαση του φωτός ή οποία συνδέεται με τη διαφάνεια.
- ***Diffraction*** – κάμψη, διασπορά και παρέμβαση του φωτός περνώντας από ένα αντικείμενο ή άνοιγμα που διαταράσσει την ακτίνα
- ***Indirect illumination*** – επιφάνειες φωτιζόμενες από φως που αντανακλάται από άλλες επιφάνειες και όχι από φως προερχόμενο απευθείας από μια πηγή φωτός (επίσης γνωστό ως global illumination)
- ***Caustics (ένας τύπος indirect illumination)*** – ανάκλαση φωτός από ένα φωτεινό αντικείμενο ή εστίαση του φωτός μέσα από ένα διαφανές αντικείμενο, για παραγωγή φωτισμένων περιοχών σε ένα άλλο αντικείμενο
- ***Depth of field*** – τα αντικείμενα εμφανίζονται θολωμένα ή εκτός εστίασης όταν βρίσκονται αρκετά μακριά, μπροστά ή πίσω από το εστιαζόμενο αντικείμενο
- ***Motion blur*** – τα αντικείμενα εμφανίζονται θολωμένα λόγω κίνησης σε υψηλή ταχύτητα, ή λόγω κίνησης της κάμερας σε υψηλή ταχύτητα
- ***Non-photorealistic rendering*** – απεικόνιση των σκηνών με ένα καλλιτεχνικό στυλ, επιδιώκοντας να φαίνονται σαν μια ζωγραφιά ή σαν ένα σχέδιο

Οι κυριότερες τεχνικές rendering που χρησιμοποιούνται για την παραγωγή της τελικής εικόνας είναι:

- ***rasterization*** (συμπεριλαμβανομένου και "scanline rendering") – προβάλει γεωμετρικά, αντικείμενα στη σκηνή σε ένα επίπεδο εικόνας (image plane), χωρίς εξειδικευμένα οπτικά εφέ
- ***ray casting*** – θεωρεί ότι η σκηνή παρατηρείται από ένα συγκεκριμένο σημείο θέασης, υπολογίζοντας την παρατηρούμενη εικόνα βασιζόμενη μόνο στη γεωμετρία και σε πολύ βασικούς «οπτικούς νόμους» έντασης της αντανάκλασης (reflection intensity)
- ***radiosity*** – χρησιμοποιεί μαθηματικά πεπερασμένων στοιχείων για να προσομοιώσει τη διάχυτη διάδοση του φωτός από τις επιφάνειες
- ***ray tracing*** – είναι παρόμοια με τη ray casting αλλά επιπλέον ενσωματώνει πιο προχωρημένες οπτικές προσομοιώσεις (optical simulations) και συνήθως χρησιμοποιεί τις επονομαζόμενες τεχνικές "Monte Carlo" για να δημιουργήσει πιο ρεαλιστικά αποτελέσματα σε μια ταχύτητα που είναι συχνά αρκετά αργή

Αξίζει να σημειωθεί ότι και η τεχνική ray casting χρησιμοποιεί τις "Monte Carlo" τεχνικές για τη μείωση των artifacts.

2.3.2. Μηχανή φυσικής (Physics engine)

Μια μηχανή φυσικής είναι ένα πρόγραμμα που προσομοιώνει μοντέλα φυσικής, χρησιμοποιώντας μεταβλητές όπως η μάζα, η ταχύτητα, η τριβή και η αντίσταση του αέρα. Μπορεί επίσης να προσομοιώσει και να προβλέψει φαινόμενα κάτω από διαφορετικές συνθήκες που θα προσεγγίζουν το τι συμβαίνει στην πραγματική ζωή ή σε ένα φανταστικό κόσμο. Υπάρχουν γενικά δυο είδη μηχανών φυσικής, οι real-time physics engines και οι high-precision physics engines.

Οι high-precision physics engines απαιτούν περισσότερη υπολογιστική ισχύ για να υπολογίσουν φυσική πολύ μεγάλης ακρίβειας και συνήθως χρησιμοποιούνται από επιστήμονες και ταινίες κινούμενων σχεδίων (computer animated movies).

Στα ηλεκτρονικά παιχνίδια και σε άλλους τύπους διαδραστικής πληροφορικής (interactive computing), η μηχανή φυσικής απλοποιεί τους υπολογισμούς της και μειώνει την ακρίβεια της, έτσι ώστε να μπορούν να πραγματοποιηθούν στον κατάλληλο χρόνο απόκρισης για το παιχνίδι και σε κατάλληλο ποσοστό για το "gameplay". Αυτή είναι η real-time physics engines και χρησιμοποιείται από τα παιχνίδια για τη βελτίωση της ρεαλιστικότητάς τους.

Οι μηχανές φυσικής έχουν δυο βασικά στοιχεία, ένα σύστημα ανίχνευσης συγκρούσεων (collision detection system) και το στοιχείο προσομοίωσης δυνάμεων (dynamics simulation component) που είναι υπεύθυνο για την επίλυση των δυνάμενων που επηρεάζουν και προσομοιώνουν τα διάφορα αντικείμενα. Οι σύγχρονες μηχανές φυσικής μπορεί επίσης να περιέχουν προσομοιώσεις ρευστών (fluid simulations), συστήματα χειρισμού σχεδιοκίνησης (animation control systems) και εργαλεία ενσωμάτωσης στοιχείων (asset integration tools).

Υπάρχουν τρία βασικά παραδείγματα για την προσομοίωση φυσικής των στερεών:

- ***Penalty methods***, όπου οι αλληλεπιδράσεις μοντελοποιούνται συνήθως σαν "mass-spring" συστήματα. Αυτού του τύπου η μηχανή είναι δημοφιλής για «παραμορφώσιμη» φυσική ή soft-body physics.

- ***Constraint based methods***, όπου οι «εξισώσεις περιορισμού» (ισότητας, ανισότητας) επιλύονται (για εκτίμηση φυσικών νόμων).
- ***Impulse based methods***, όπου οι αρχές της «ορμής» εφαρμόζονται για τις αλληλεπιδράσεις των αντικειμένων.

Τέλος υπάρχει και η κατηγορία των "hybrid methods" η οποία ουσιαστικά είναι αποτέλεσμα του συνδυασμού των παραπάνω παραδειγμάτων.

Για τα ηλεκτρονικά παιχνίδια η ταχύτητα της προσομοίωσης είναι σημαντικότερη από την ακρίβεια. Συνήθως τα περισσότερα από τα τρισδιάστατα αντικείμενα σε ένα παιχνίδι αναπαριστώνται από δύο διαφορετικά σχήματα ή «πλέγματα» (meshes). Το ένα είναι υψηλής πολυπλοκότητας και λεπτομερές σχήμα και είναι εκείνο που βλέπει ο θεατής στο παιχνίδι και το άλλο για λόγους ταχύτητας, είναι ένα πολύ απλοποιημένο και «αόρατο» σχήμα το οποίο χρησιμοποιείται για να αναπαραστήσει το αντικείμενο στη μηχανή φυσικής. Το απλοποιημένο αυτό «πλέγμα» που χρησιμοποιείται στην επεξεργασία φυσικής, συχνά αναφέρεται και ως «γεωμετρία σύγκρουσης» (collision geometry) και μπορεί να είναι ένα bounding box, μια bounding sphere ή ένα κυρτό κέλυφος (convex hull). Γενικά αυτού του είδους τα «αόρατα σχήματα» χρησιμοποιούνται για «ευρείας φάσης ανίχνευση σύγκρουσης» (broad phase collision detection), δηλαδή ουσιαστικά για να ελαχιστοποιήσουν τον αριθμό των πιθανών συγκρούσεων πριν η απαιτητική «πλέγμα σε πλέγμα» (mesh on mesh) ανίχνευση σύγκρουσης λάβει χώρα στη «βραχείας φάσης ανίχνευση σύγκρουσης» (narrow phase collision detection).

Στον πραγματικό κόσμο η φυσική είναι πάντα «ενεργή», καθώς κάποιες δυνάμεις είναι απαραίτητες για την υπόσταση της μάζας και της ύλης όπως την ξέρουμε (για να υπάρχω εγώ ως άνθρωπος ή εσείς που διαβάζετε αυτή τη στιγμή). Από την άλλη για ένα παιχνίδι θα ήταν περιττό έως αδύνατον να συμβαίνει αυτό καθώς η δύναμη της CPU είναι περιορισμένη. Ένα χαρακτηριστικό παράδειγμα είναι αυτό του εικονικού κόσμου του "Second Life" όπου όταν ένα αντικείμενο μείνει ακίνητο για κάποιο χρονικό διάστημα κάπου, τότε οι υπολογισμοί φυσικής από τη μηχανή φυσικής απενεργοποιούνται μέχρι να υπάρξει και πάλι κάποιου είδους αλληλεπίδρασης με αυτό το αντικείμενο. Με αυτό τον τρόπο η μηχανή φυσικής εξοικονομεί πόρους για τον επεξεργαστή και τη μνήμη και κατά συνέπεια βελτιώνεται το framerate του παιχνιδιού. Το κυριότερο όριο των μηχανών φυσικής που επηρεάζει άμεσα το

ρεαλισμό τους είναι στην ακρίβεια των αριθμών που αναπαριστούν τη θέση ενός αντικειμένου και των δυνάμεων που ενεργούν σε αυτό το αντικείμενο. Όταν το ποσοστό της ακρίβειας είναι χαμηλό, προκύπτουν σφάλματα στους υπολογισμούς κυρίως λόγω της «στρογγυλοποίησης» αναγκάζοντας το αντικείμενο να υπερβεί της κανονικής θέσης του ή να τοποθετηθεί πριν από αυτή. Από την άλλη ένα μεγαλύτερο ποσοστό ακρίβειας μπορεί να ελαχιστοποιήσει τέτοιου είδους σφάλματα αλλά επίσης και να αυξήσει την επεξεργαστική ισχύ για τους απαιτούμενους υπολογισμούς.

Ένα άλλο συνηθισμένο πρόβλημα που αφορά την ακρίβεια της φυσικής είναι το λεγόμενο "framerate", η αλλιώς ο αριθμός των «στιγμών» ανά δευτερόλεπτο που υπολογίζεται η φυσική. Κάθε frame αντιμετωπίζεται σαν ξεχωριστό από όλα τα άλλα και ο χώρος ανάμεσα στα frames δεν υπολογίζεται. Ένα χαμηλό framerate και ένα μικρό και γρήγορα κινούμενο αντικείμενο οδηγούν σε μια κατάσταση όπου το αντικείμενο δεν κινείται ομαλά στο χώρο αλλά φαίνεται σαν να τηλεμεταφέρεται από ένα σημείο στο χώρο σε ένα άλλο, καθώς κάθε frame υπολογίζεται. Για παράδειγμα σε εξαιρετικά υψηλές ταχύτητες μια σφαίρα μπορεί να χάσει το στόχο, αν αυτός είναι αρκετά μικρός ώστε να «χωρέσει» στο κενό μεταξύ των υπολογιζόμενων frames της ταχείας κινούμενης σφαίρας. Αυτό μπορεί να αντιμετωπιστεί με διάφορους τρόπους, για παράδειγμα και πάλι στο "Second Life" η σφαίρα αντιμετωπίζεται σαν ένα «αόρατο» βέλος, έτσι ώστε όταν η σφαίρα «τηλεμεταφέρεται» το βέλος είναι τόσο μεγάλο ώστε να καλύπτει αυτή την απόσταση (από το ένα σημείο στο άλλο) και να επιτρέπει τη σύγκρουση με οποιοδήποτε αντικείμενο μπορεί να «χωρέσει» ανάμεσα στα υπολογιζόμενα frames.

Στο παρελθόν η «βασιζόμενη σε φυσική» σχεδιοκίνηση χαρακτήρα χρησιμοποιούσε μόνο "rigid body dynamics" επειδή ήταν γρηγορότερη και ευκολότερη στον υπολογισμό. Όμως τα πιο σύγχρονα παιχνίδια και οι ταινίες αρχίζουν να χρησιμοποιούν "soft body physics" επειδή τώρα είναι επιτρεπτό λόγω της εξελιγμένης τεχνολογίας. Η "soft body physics" χρησιμοποιείται επίσης για σωματιδιακά εφέ (particle effects), ρευστά (liquids) και υφάσματα (cloth).

Αξιοσημείωτο είναι το γεγονός ότι τον Φεβρουάριο του 2006 κυκλοφόρησε η πρώτη «μονάδα φυσικής επεξεργασίας» (PPU—Physical Processing Unit) από την "Ageia" (η οποία αργότερα ενσωματώθηκε με την Nvidia), που ονομαζόταν "PhysX" και η οποία λειτουργεί με παρόμοιο τρόπο όπως η GPU σε μια κάρτα γραφικών, απελευθερώνοντας από το βάρος της επεξεργασίας τη CPU. Η μονάδα αυτή είχε σαν

δυνατό της σημείο την επιτάχυνση των «σωματιδιακών συστημάτων» (particle systems), επίσης παρατηρήθηκε και μια μικρή βελτίωση στη "rigid body physics".

Μια άλλη και πολλά υποσχόμενη προσέγγιση για τις "real-time physics engines" αποτελεί και η «μονάδα γραφικής επεξεργασίας γενικού σκοπού» (GPGPU – General purpose Graphics Processing Unit) η οποία περιλαμβάνει και "rigid body dynamics". Οι εταιρείες "ATI" και "Nvidia" είναι οι κυριότερες εταιρείες στο χώρο και ανταγωνίζονται για την παραγωγή μονάδων και καρτών. Ειδικά η τελευταία, ερευνά και κυκλοφορεί σταδιακά ένα SDK toolkit το οποίο ονομάζεται "CUDA" (Compute Unified Device Architecture) και είναι μια πολλά υποσχόμενη τεχνολογία για την GPU.

Τέλος μερικές μηχανές φυσικής παρατίθενται παρακάτω:

- Bullet physics engine
- Chipmunk physics engine
- Open Dynamics Engine
- Tokamak physics engine
- Farseer Physics Engine
- Physics 2D.Net
- Newton Game Dynamics
- PhysX
- Havok
- Vis Sim

2.3.3.Μηχανή ήχου (Sound engine)

Μια μηχανή ήχου είναι πολύ σημαντικό κομμάτι της μηχανής ενός παιχνιδιού. Γίνεται εύκολα αντιληπτό ότι ένα παιχνίδι χωρίς ήχο είναι σαν ένας άνθρωπος χωρίς ρούχα. Η μηχανή του ήχου «ντύνει» το παιχνίδι, προσθέτοντας ενδιαφέρον, συναίσθημα αλλά και ποιότητα και λειτουργικότητα. Στον προγραμματισμό τοποθετείται στο πεδίο του “Sound pipeline”.

Πιο συγκεκριμένα μια μηχανή ήχου σε ένα παιχνίδι μπορεί να περιλαμβάνει αρχεία ήχου (sound files) που είναι και το απαραίτητο συστατικό για περαιτέρω επεξεργασία, εφέ ήχου (sound effects) που επίσης είναι πολύ σημαντικά ιδιαίτερα σε κάποια συγκεκριμένα κομμάτια του παιχνιδιού, συστήματα ελέγχου του ήχου (sound control systems) όπως αναπαραγωγή ήχου, διακοπή ήχου, παύση ήχου, αυξομείωση έντασης του ήχου, κ.τ.λ.

Όλα τα παραπάνω αναλαμβάνουν να αναπτύξουν μέσα σε ένα παιχνίδι οι “sound engineers”, οι οποίοι στην ουσία είναι προγραμματιστές που αναπτύσσουν συγκεκριμένες μεθόδους και συγκεκριμένους αλγόριθμους που έχουν να κάνουν με το στοιχείο του ήχου σε ένα παιχνίδι. Πολλά παιχνίδια χρησιμοποιούν ακόμα πιο προχωρημένες τεχνικές όπως τρισδιάστατο ήχο (3D positional sound) και στοιχεία όπως “Doppler”, που κάνουν την ανάπτυξή τους ακόμα πιο δύσκολη αλλά και το αποτέλεσμα ακόμα πιο ενδιαφέρον.

2.3.4. Μηχανή τεχνητής νοημοσύνης (Artificial Intelligence engine)

Μια «μηχανή τεχνητής νοημοσύνης» σε ένα παιχνίδι είναι αυτή που παράγει την ψευδαίσθηση της εξυπνάδας στη συμπεριφορά των λεγόμενων “non-player characters” (NPCS). Οι τεχνικές που χρησιμοποιούνται κυρίως προεκτείνουν αυτές του πεδίου της κλασικής τεχνητής νοημοσύνης.

Με τον όρο “game AI” νοείται ένα ευρύ σύνολο από αλγόριθμους, οι οποίοι περιλαμβάνουν τεχνικές από «θεωρία ελέγχου» (control theory), «ρομποτική» (robotics), «γραφικά υπολογιστή» (computer graphics) και γενικά από τον τομέα της «επιστήμης των υπολογιστών» (computer science). Καθώς η τεχνητή νοημοσύνη για παιχνίδια επικεντρώνεται περισσότερο στην «εμφάνιση» της εξυπνάδας και στο καλό gameplay, η προσέγγισή της είναι διαφορετική από αυτή της «παραδοσιακής τεχνητής νοημοσύνης». Το «χακάρισμα» και οι «κωδικοί» είναι αποδεκτά και σε αρκετές περιπτώσεις οι «ικανότητες»του υπολογιστή πρέπει να μειωθούν έτσι ώστε να δίνεται στους χειριστές μια «αίσθηση δικαιοσύνης». Για παράδειγμα σε ένα παιχνίδι FPS (First Person Shooter), όπου η τέλεια κίνηση και ο στόχος των NPC’s θα ήταν πέραν κάθε ανταγωνισμού από τις «ανθρώπινες ικανότητες» του χειριστή.

Οι τεχνικές της τεχνητής νοημοσύνης παιχνιδιού μπορούν να χρησιμοποιηθούν για ανίχνευση σύγκρουσης (collision detection) και συνήθως για εντοπισμό μελλοντικών συγκρούσεων, για «αποφάσεις στόχευσης» (targeting decisions) οι οποίες προκύπτουν από αποτελέσματα σε δοκιμές οπτικής (line of sight tests) για το λεγόμενο “pathfinding” το οποίο συναντάται κυρίως σε παιχνίδια στρατηγικής και έχει να κάνει με τη μέθοδο απόφασης για το πώς ένας NPC μπορεί να πάει από ένα σημείο στο χάρτη σε ένα άλλο, λαμβάνοντας υπόψη την πίστα, τα εμπόδια και την ικανότητα επίγνωσης της κατάστασης, και τέλος για τη «δυναμική προσαρμογή της δυσκολίας του παιχνιδιού» (dynamic game difficulty balancing), που έχει να κάνει με την προσαρμογή του επιπέδου δυσκολίας σε πραγματικό χρόνο του παιχνιδιού ανάλογα με την ικανότητα του παίκτη.

Τέλος η μηχανή τεχνητής νοημοσύνης καταλαμβάνει περίπου το 10% με 20% του συνολικού προγραμματισμού ενός παιχνιδιού.

2.3.5. Shaders

Όσον αφορά τις «γραφικές διεπαφές προγραμματισμού εφαρμογών» (graphics API – Application Programming Interface) πάνω στις οποίες βασίζεται η ανάπτυξη των rendering engine μιας μηχανής παιχνιδιού (game engine), οι δυο πιο σημαντικές είναι το "Direct 3D" και το "OpenGL". Τα API's αυτά προσφέρουν κατά κάποιο τρόπο μια «διαίρεση λογισμικού» της GPU. Χαμηλού επιπέδου (low-level) βιβλιοθήκες όπως οι "DirectX", "SDL" και "OpenAL" χρησιμοποιούνται συχνά σε παιχνίδια καθώς παρέχουν «ανεξάρτητη από το υλικό» (hardware-independent) πρόσβαση σε άλλο υλικό υπολογιστή όπως συσκευές εισόδου (input devices), κάρτες δικτύου (network cards) και κάρτες ήχου (sound cards).

Πριν τα "hardware-accelerated" τρισδιάστατα γραφικά, χρησιμοποιούνταν renderers λογισμικού (software renderers). Το software rendering χρησιμοποιείται ακόμα σε κάποια «εργαλεία μοντελοποίησης ή για "still-rendered" εικόνες όταν η οπτική ακρίβεια (visual accuracy) έχει τιμή μεγαλύτερη από τη real-time επίδοση (frames-per-second) ή όταν το υλικό του υπολογιστή δεν πληροί απαιτήσεις όπως «υποστήριξη shader» ή για την περίπτωση των Windows Vista «υποστήριξη Direct 3D 10».

Σε αυτό το σημείο κρίνεται αναγκαίο να αναφερθεί ένα πολύ σημαντικό κεφαλαίο στην ανάπτυξη των παιχνιδιών, με το οποίο και θα κλείσει η αναφορά στα στοιχεία περί προγραμματισμού των παιχνιδιών. Το κεφάλαιο αυτό δεν είναι άλλο, από το κεφάλαιο των "shaders". Πριν γίνει αναλυτική αναφορά για τους shaders, θα παρατεθούν κάποια γενικά και ιστορικά στοιχεία γι΄ αυτούς.

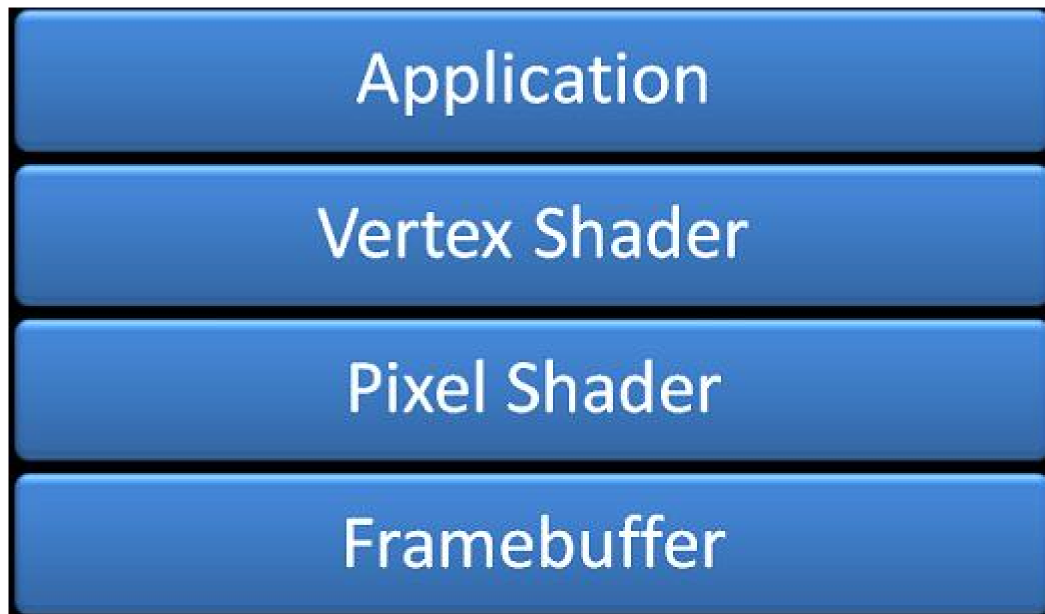
Πριν την εμφάνιση του "DirectX8", οι GPU's είχαν ένα «σταθερό» τρόπο να μετασχηματίζουν εικονοστοιχεία (pixels) και κορυφές (vertices), ο οποίος ονομαζόταν "Fixed Function Pipeline" (FFP). Αυτός ο τρόπος δεν επέτρεπε στους developers να αλλάζουν το πώς τα pixels και τα vertices θα μετασχηματίζονταν και θα επεξεργαζόταν αφού είχαν περάσει πια στην GPU και αυτό είχε σαν συνέπεια τα παιχνίδια να είναι αρκετά όμοια από γραφικής πλευράς. Μόλις το DirectX8 εμφανίστηκε, έκαναν την εμφάνισή τους και οι vertex και pixel shaders, οι οποίοι

στην ουσία παρείχαν μια μέθοδο στους developers την οποία μπορούσαν να χρησιμοποιήσουν για να αποφασίσουν πώς τα vertices και τα pixels θα έπρεπε να επεξεργαστούν καθώς θα πέρναγαν μέσα από το "pipeline", δίνοντάς τους έτσι αρκετή «ευελιξία». Μια συμβολική γλώσσα (assembly language) χρησιμοποιούταν για τον προγραμματισμό των shaders, κάτι που έκανε αρκετά δύσκολο το να είναι κάποιος "shader developer" και η μοναδική υποστηριζόμενη έκδοση ήταν η "shader model 1.0." Αυτό άλλαξε όταν εμφανίστηκε το DirectX9, το οποίο έδωσε τη δυνατότητα στους developers να αναπτύσσουν shaders σε μια υψηλού επιπέδου γλώσσα (high level language) η οποία είναι γνωστή ως "HLSL" (High Level Shading Language), αντικαθιστώντας έτσι την προϋπάρχουσα συμβολική γλώσσα με μια γλώσσα που έμοιαζε περισσότερο με τη γλώσσα προγραμματισμού C. Αυτό έκανε ευκολότερη τη συγγραφή shader, αλλά και την ανάγνωση και εκμάθησή τους.

Τέλος η πρόσφατη εμφάνιση του DirectX10 παρουσίασε και ένα καινούριο shader γνωστό ως "Geometry Shader" που αποτελεί μέρος του "shader model 4.0.". Βέβαια ο συγκεκριμένος shader απαιτεί και μια νέα "state-of-the-art" κάρτα γραφικών που να τον υποστηρίζει, αλλά και λειτουργικό σύστημα "Windows Vista".

Τι είναι όμως ένας shader; Παρακάτω θα αναλυθεί περιληπτικά η δομή και η λειτουργία ενός "shader" και έπειτα θα γίνει μια αναλυτικότερη παρουσίαση του.

Ο "shader" σαν όρος στον τομέα του game development λοιπόν, είναι ένα πρόγραμμα που δίνει στον καλλιτέχνη ή τον προγραμματιστή πρωτοφανή έλεγχο πάνω στο τελικό αποτέλεσμα του rendering ενός frame και επίσης παρέχει και ένα καλό επίπεδο χειρισμού πάνω σε ατομικά vertex δεδομένα. Οι shaders μπορούν να χρησιμοποιηθούν για να «προσαρμόσουν» βήματα στο "pipeline" (ή Graphics pipeline ή 3D Pipeline) κάνοντας έτσι τον developer υπεύθυνο να εφαρμόσει το πώς θα πρέπει να γίνουν επεξεργάσιμα τα pixels και τα vertices. Όπως φαίνεται στην παρακάτω εικόνα, η εφαρμογή αρχίζει και χρησιμοποιεί ένα shader όταν βρίσκεται στη διαδικασία του rendering. Ο "vertex buffer" λειτουργεί με τον pixel shader στέλνοντας δεδομένα από τον vertex shader στον pixel shader, οι οποίοι δουλεύοντας μαζί δημιουργούν μια εικόνα στον frame buffer.



Εικόνα 19 : Ροή του shader στο graphics pipeline

Οι “vertex shaders” χρησιμοποιούνται για να «παραποιήσουν» δεδομένα vertex ανά vertex. Για παράδειγμα μπορεί ένας shader να κάνει ένα μοντέλο να δείχνει πιο «παχύ» κατά τη διάρκεια του rendering μετακινώντας vertices σε μια νέα θέση για κάθε vertex στο μοντέλο (deform shaders). Οι vertex shader λαμβάνουν είσοδο από μια δομή (structure) vertex που ορίζεται στον κώδικα της εφαρμογής και τη φορτώνουν από τον vertex buffer στο shader. Με αυτόν τον τρόπο περιγράφονται τι ιδιότητες θα έχει κάθε vertex κατά το “shading” (π.χ. Position, Color, Normal, Tangent, κ.τ.λ.). Στη συνέχεια ο vertex shader στέλνει τα δεδομένα εξόδου του στον pixel shader για μεταγενέστερη χρήση. Ο καθορισμός του τι δεδομένα θα περάσει ο vertex shader στην «επόμενη φάση» μπορεί να γίνει ορίζοντας μια δομή (struct) στον shader, η οποία θα έχει τα δεδομένα που θέλουμε να αποθηκεύσει και στη συνέχεια ο vertex shader αναγκάζεται να «επιστρέψει» αυτή την υπόδειξη, ή ορίζοντας παραμέτρους στον shader χρησιμοποιώντας τη λέξη κλειδί “out”. Σαν έξοδο μπορεί να έχουμε Position, Fog, Color, Texture Coordinates, Tangents, Light Position, κ.τ.λ.

Effect1.fx

```
float4x4 World;
float4x4 View;
float4x4 Projection;

// TODO: add effect parameters here.

struct VertexShaderInput
{
    float4 Position : POSITION0;

    // TODO: add input channels such as texture
    // coordinates and vertex colors here.
};

struct VertexShaderOutput
{
    float4 Position : POSITION0;

    // TODO: add vertex shader outputs such as colors and texture
    // coordinates here. These values will automatically be interpolated
    // over the triangle, and provided as input to your pixel shader.
};

VertexShaderOutput VertexShaderFunction(VertexShaderInput input)
{
    VertexShaderOutput output;

    float4 worldPosition = mul(input.Position, World);
    float4 viewPosition = mul(worldPosition, View);
    output.Position = mul(viewPosition, Projection);

    // TODO: add your vertex shader code here.

    return output;
}
```

Εικόνα20: Δομή "data structs" και "vertex shader function" σε ένα effect file

Οι pixel shaders «παραποιούν» όλα τα pixels ανά pixel, σε ένα γνωστό πλήθος από vertices (μπορεί να είναι μοντέλο, αντικείμενο κ.τ.λ.). Για παράδειγμα μπορεί να έχουμε ένα μεταλλικό κουτί, του οποίου θέλουμε να τροποποιήσουμε τον αλγόριθμο φωτισμού σε χρώματα, κ.ο.κ. Ο pixel shader λαμβάνει δεδομένα από τον vertex shader (τα δεδομένα εξόδου (outputs) του vertex shader αποτελούν δεδομένα εισόδου (inputs) για τον pixel shader, όπως "position", "normals" και "texture coordinates"). Ο pixel shader μπορεί να έχει σαν έξοδο δύο τύπους τιμών: Color και Depth.

```
Effect1.fx
36
37    float4 PixelShaderFunction(VertexShaderOutput input) : COLOR0
38    {
39        // TODO: add your pixel shader code here.
40
41        return float4(1, 0, 0, 1);
42    }
43
```

Εικόνα 21: Δομή "pixel shader function" σε ένα effect file

Όσον αφορά το στοιχείο των "techniques/passes" ενός shader, αυτό έχει να κάνει με τη δυνατότητα χρήσης μιας ή παραπάνω τεχνικών από τον shader οι οποίες με τη σειρά τους λένε στην εφαρμογή ποιος vertex shader και ποιος pixel shader θα χρησιμοποιηθεί (συγκεκριμένα ποιες vertex shader και pixel shader functions) και επίσης ποιος shader model (π.χ. 1.0, 2.0, 2.1, 3.0) θα χρησιμοποιηθεί για να γίνει το "compile".

```
44    technique Technique1
45    {
46        pass Pass1
47        {
48            // TODO: set renderstates here.
49
50            VertexShader = compile vs_1_1 VertexShaderFunction();
51            PixelShader = compile ps_1_1 PixelShaderFunction();
52        }
53    }
54
```

Εικόνα22: Δομή "technique" σε ένα effect file

Μια τεχνική μπορεί επίσης να έχει μια ή περισσότερες “passes” οι οποίες δηλώνουν ότι η συγκεκριμένη τεχνική μπορεί να χρησιμοποιήσει περισσότερες από μία vertex shader και pixel shader functions και κατά συνέπεια το rendering θα γίνει υποστηρίζοντας ένα ή παραπάνω “ shader model”.

Σαν συμπέρασμα μπορούμε να πούμε ότι κάνοντας χρήση των λεγόμενων “effect files” ή “fx files” ένας developer μπορεί με αρκετή ευκολία και ευελιξία να αναπτύξει τους shaders της αρεσκείας του ακολουθώντας κάποια βασικά και στοιχειώδη βήματα:

- Δηλώσεις σταθερών ή μεταβλητών (συνήθως είναι τύπου “global”).
- Συγγραφή συναρτήσεων (pixel shader functions, vertex shader functions, ή άλλων).
- Καθορισμός τεχνικών shading.

Τέλος θα ήταν σημαντικό να αναφερθεί και η χρήση των λεγόμενων “semantics” που χρησιμοποιούνται στην HLSL για να «ζευγαρώσουν» τις μεταβλητές με τα ανάλογα δεδομένα. Για παράδειγμα τέτοια semantics είναι:

- **POSITION** για vertex position
- **NORMAL** για vertex normal
- **TEXCOORDX** για άλλες αυθαίρετες τιμές, όπου X>0
- **COLORX** που στην ουσία είναι όμοιο με το TEXCOORDX, αλλά με λιγότερη ακρίβεια.

Όπως έχει ήδη αναφερθεί η γλώσσα που χρησιμοποιείται για την ανάπτυξη των shaders είναι η HLSL, που είναι μια γλώσσα υψηλού επιπέδου. Στην HLSL μπορείς

να ορίσεις μεταβλητές (variables), συναρτήσεις (functions), τύπους δεδομένων (datatypes), βρόγχους ελέγχου (testing — if/else/for/do/while) και πολλά περισσότερα με σκοπό τη δημιουργία μιας «λογικής» που θα έχει να κάνει με pixels και vertices. Παρακάτω παρατίθενται ενδεικτικά μερικά στοιχεία που αφορούν τη γλώσσα HLSL:

- ***<u>Παραδείγματα "datatypes" στην HLSL</u>***

  - **bool** : true or false
  - **int** : 32-bit integer
  - **half**: 16-bit integer
  - **float**: 32-bit float
  - **double** : 64-bit double

- ***<u>Παραδείγματα "vectors" στην HLSL</u>***

  - **float3** : onevector
  - **float** : onevector
  - **vector** : onevector
  - **float2** : onevector
  - **bool3** : onevector

- ***Παραδείγματα "matrices" στην HLSL***

  - **float 3x3** : ένας 3x3 πίνακας, τύπου float
  - **float 2x2** : ένας 2x2 πίνακας, τύπου float

- ***Παραδείγματα "functions" στην HLSL***

  - **cos (X)** : επιστρέφει το συνημίτονο του X
  - **sin (X)** : επιστρέφει το ημίτονο του X
  - **cross (a, b)** : επιστρέφει την εφαπτομένη δυο διανυσμάτων a και b
  - **dot (a, b)** : επιστρέφει την εφαπτομένη δυο διανυσμάτων a και b
  - **normalize (v)** : επιστρέφει το διάνυσμα με τιμή V / |V|

ΚΕΦΑΛΑΙΟ 3

Ανάπτυξη του demo παιχνιδιού

3.1. Εισαγωγή στον προγραμματισμό και στα τρισδιάστατα γραφικά του demo

Η δημιουργία ενός 3D game χρειάζεται την συνεργασία αρκετών ατόμων, σε διάφορους τομείς. Οι κυριότεροι τομείς είναι ο τομέας του προγραμματισμού και ο τομέας των γραφικών.

Πριν ξεκινήσει η αναλυτική παρουσίαση της δουλειάς που έγινε για την ανάπτυξη του demo παιχνιδιού, κρίνεται σκόπιμο να γίνει μια σύντομη αναφορά στα λογισμικά που χρησιμοποιήθηκαν καθώς και σε κάποια γενικά χαρακτηριστικά τους. Η αναφορά θα είναι σύντομη και περιληπτική καθώς στη συνέχεια θα παρουσιαστούν αναλυτικά και συγκεκριμένα η μεθοδολογία και τα βήματα που ακολουθήθηκαν, τα οποία αποτελούν και την ουσία αυτού του κεφαλαίου.

- <u>XNA</u>

Το "XNA Game Studio", το οποίο ουσιαστικά αποτελεί εξειδικευμένο λογισμικό για ανάπτυξη (προγραμματιστική) παιχνιδιών, το οποίο προϋποθέτει και την ύπαρξη ενός άλλου λογισμικού με σκοπό την άντληση δεδομένων και διεπιφανειών όπως βιβλιοθήκες (libraries), εργαλεία (toolbars), κ.τ.λ. Το λογισμικό αυτό είναι το "Microsoft Visual Studio". Συγκεκριμένα χρησιμοποιήθηκαν, το "Microsoft Visual Studio 2005 Pro" με το "XNA Game Studio 2.0" και το "Microsoft Visual Studio 2008 Express Edition" με το "XNA Game Studio 3.0". Επίσης για συγγραφή HLSL κώδικα χρησιμοποιήθηκαν, το "Notepad" και το "Nvidia Fx Composer 2.0" και για διάφορα test χρησιμοποιήθηκε το "DirectX SDK 2008".

Το XNA είναι ένα "framework" (και όχι μια μηχανή), δηλαδή στην ουσία είναι ένα σύνολο βιβλιοθηκών ανάπτυξης διαχειριζόμενου κώδικα (.NET) που βασίζονται στο .NET Framework 2.0 και επιτρέπει στους game developers να είναι πολύ παραγωγικοί όταν δημιουργούν παιχνίδια για Windows ή Xbox 360 [15].

Λίγο πιο συγκεκριμένα το XNA Framework επιτρέπει στους game developers να δημιουργήσουν σύγχρονα παιχνίδια χρησιμοποιώντας τη γλώσσα προγραμματισμού (C-Sharp) και ένα μεγάλο σύνολο βιβλιοθηκών. Το XNA Framework παρέχει το δικό του "content pipeline" με σκοπό να διευκολύνει την είσοδο περιεχομένου (3D, 2D, sound κ.τ.λ.), από πηγές δημιουργίας τέτοιου περιεχομένου, στο παιχνίδι [15].

Το XNA περιέχει ένα πλήρες σύνολο από αρχειακή, διδακτική και βοηθητική υποστήριξη, πράγμα που το κάνει ένα πολύ αξιόπιστο και εύχρηστο εργαλείο. Για να τρέξει το XNA απαιτείται να έχει εγκατασταθεί κάποια συγκεκριμένη έκδοση του Microsoft Visual Studio (ανάλογα με την έκδοση του XNA GS) και το DirectX runtime environment. Επίσης απαιτεί ελάχιστο Shader model 1.1 (αλλά προτιμότερος είναι ο Shader model 2.0) και κατά συνέπεια και μια κάρτα γραφικών που να υποστηρίζει Direct 3D 9.0. Όσον αφορά τα λειτουργικά συστήματα, το XNA μπορεί να τρέξει σε Windows XP και σε Windows Vista [15].

- Blender

Το blender είναι ένα ελεύθερο λογισμικό 3D γραφικών. Μπορεί να χρησιμοποιηθεί για μοντελοποίηση, UV mapping, texturing, rigging, skinning, animating, rendering, compositing, particles και τη δημιουργία 3D εφαρμογών αλληλεπίδρασης πραγματικού χρόνου.

- 3ds Max

Το 3ds Max είναι ένα πακέτο μοντελοποίησης, animation και rendering που αναπτύχθηκε από την Autodesk Media and Entertainment. Χρησιμοποιείται κυρίως για την ανάπτυξη ηλεκτρονικών παιχνιδιών, από διαφημιστικά και αρχιτεκτονικά studio, και για την δημιουργία ψηφιακών εφέ σε ταινίες.

Σε μια δεύτερη φάση, σειρά πήρε η απόφαση για το συγκεκριμένο θέμα της εργασίας, καθώς και η κατασκευή ενός πλάνου το όποιο ακολουθήθηκε μέχρι την ολοκλήρωση της δουλειάς. Έτσι λοιπόν η ιδέα που διαμορφώθηκε είχε ως εξής:

Το demo θα αποτελούνταν από δυο μέρη-σκηνές. Το ύφος του θα ήταν «παιχνίδι μυστηρίου» και η οπτική του θα ήταν «Τρίτου προσώπου». Με λίγα λόγια το σενάριο υπαγόρευε για την πρώτη σκηνή την ύπαρξη ενός φοιτητή, ο οποίος θα ήταν και ο κεντρικός χαρακτήρας του παιχνιδιού και την ύπαρξη ενός εσωτερικού χώρου διαμερίσματος, το οποίο θα αποτελούσε το κυρίως περιβάλλον της σκηνής και για τη δεύτερη σκηνή την ύπαρξη του φοιτητή, πάλι ως κεντρικού χαρακτήρα αλλά αυτή τη φορά το κυρίως περιβάλλον της σκηνής θα αποτελούσε το μνημείο του «Οθωμανικού Λουτρού» που βρίσκεται στην πόλη της Μυτιλήνης. Εν ολίγοις λοιπόν στην πρώτη σκηνή υπάρχει ένας φοιτητής που «ζει» στο διαμέρισμά του και στη δεύτερη σκηνή έχουμε τον ίδιο φοιτητή να έχει πλέον μεταβεί από το διαμέρισμά του στο χώρο του Οθωμανικού Λουτρού.

3.2. Προγραμματισμός

Χαρακτήρας 1ης σκηνής

Στην αρχή, «προγραμματιστικά» μας απασχόλησε η πρώτη σκηνή και συγκεκριμένα το κομμάτι του χαρακτήρα. Σκοπός ήταν να εισαχθεί ένα μοντέλο χαρακτήρα, (που προηγουμένως θα είχε κατασκευαστεί με βάση τις δικές μας ανάγκες από την αρχή), στο “Content” του XNA. Ποιο συγκεκριμένα θέλαμε ένα όσο το δυνατόν γίνεται ποιο ρεαλιστικό χαρακτήρα που θα ταίριαζε με το ύφος του demo (και το σενάριο βέβαια), ο οποίος θα ήταν και “animated” αλλά και διαχειριζόμενος από το χρήστη. Με άλλα λόγια τα κριτήρια που έπρεπε να ικανοποιούνται ήταν πολυάριθμα και απολύτως αναγκαία. Έτσι λοιπόν μετά από αρκετό χρόνο μελέτης και εκτιμήσεων των κατά καιρούς δεδομένων αποφασίστηκε η χρήση μιας βιβλιοθήκης για το XNA, η οποία υποστήριζε animation, και η οποία ονομάζεται “Skinned Model Pipeline” [4]. Πριν γίνει αναφορά της συγκεκριμένης βιβλιοθήκης με το μοντέλο του χαρακτήρα, αξίζει να σημειωθεί ότι το XNA δεν διαθέτει δική του βιβλιοθήκη για υποστήριξη animation (animated μοντέλα, animation controls, κ.τ.λ.) κι αυτό ίσως να ήταν για αρκετό καιρό ένα μεγάλο του μειονέκτημα: και λέω «ήταν» γιατί πριν από μερικούς μήνες και ύστερα από αρκετές υποδείξεις αλλά και προσδοκίες ατόμων, η Microsoft προχώρησε στο release του XNA Game Studio 3.1, το οποίο παρέχει εκτός των άλλων και υποστήριξη animation.

Η “Skinned Model Pipeline” λοιπόν, διαθέτει αρκετές μεθόδους χειρισμού animation, μεθόδους μαθηματικές οι οποίες είναι υπεύθυνες για τη «μετάφραση» των απαραίτητων πληροφοριών από το μοντέλο (το σύνολο αυτών των μεθόδων συνήθως αποτελούν κάποια πρόσθετα στοιχεία που μπορεί να διαθέτει το XNA, αυτά των “Custom Processors”), μεθόδους «αναπαραγωγής» των «πληροφοριών animation» σε πραγματικό χρόνο (το λεγόμενο “Key frame animation”, το οποίο στην ουσία μεταφέρει και αναπαράγει RTS δεδομένα – Rotation Translation Scale), μεθόδους για το shading του μοντέλου, μεθόδους για τον παρεχόμενο animation player (οι μέθοδοι αυτές αφορούν κυρίως πρόσθετες πληροφορίες που αντλούνται από το

μοντέλο και μπορεί να είναι πληροφορίες για το σκελετό (bone transforms), πληροφορίες δέρματος (skinning data), κ.τ.λ.), κ.ά. [4].

Έχοντας λοιπόν ως δεδομένα την ανάγκη χρήσης σκελετού στο μοντέλο του χαρακτήρα, αλλά και την ανάγκη δημιουργίας ενός HLSL αρχείου (Effect file - .fx) το οποίο ήταν απαραίτητο για την τελική απεικόνιση του χαρακτήρα στην οθόνη σε πραγματικό χρόνο, έγινε η εισαγωγή του μοντέλου στο XNA και αμέσως μετά το βάρος έπεσε στη δημιουργία του "Skinned Model.fx" που χρησιμοποιήθηκε σαν shader του χαρακτήρα.

Μετά την ολοκλήρωση της δημιουργίας του shader ο οποίος υποστήριζε animation, diffuse mapping, normal mapping και specular mapping, έγινε η απαραίτητη συγγραφή κώδικα στο XNA ώστε τελικά να απεικονιστεί ο χαρακτήρας animated στην οθόνη. Η διαδικασία δεν ήταν εύκολη και έχριζε αρκετής προσπάθειας με συνεχόμενες δοκιμές και διορθώσεις, αλλά τελικά υπήρξε αποτέλεσμα.

Κλείνοντας πρέπει να αναφερθεί ότι για διάφορους λόγους, με πιο σημαντικό αυτόν της υστέρησης της βιβλιοθήκης στη χρήση πολλαπλών animation στο ίδιο μοντέλο αλλά και απουσία "blending" από animation σε animation, το αποτέλεσμα δεν μας άφησε καθόλου ικανοποιημένους κι έτσι δειλά δειλά αρχίσαμε να ψάχνουμε για μελλοντικές λύσεις.

Περιβάλλον χώρος 1ης σκηνής

Σειρά για την πρώτη σκηνή πήρε ο προγραμματισμός για το «εσωτερικό» του διαμερίσματος . Το ζητούμενο ήταν να εισαχθεί στο XNA ένα μοντέλο (εσωτερικού) διαμερίσματος, το οποίο θα διέθετε κανονικούς χώρους όπως καθιστικό, τραπεζαρία, υπνοδωμάτιο, κ.ά. αλλά και επιμέρους αντικείμενα όπως ψυγείο, κουζίνα, τηλεόραση, κρεβάτι, υπολογιστή κ.ά.. Τα δυο κύρια σημεία που μας απασχόλησαν ήταν αυτό του «φωτισμού» του χώρου και αυτό του "texturing" του μοντέλου. Επειδή επρόκειτο για ένα μοντέλο που στην ουσία θα περιείχε όλα τα παραπάνω έπρεπε να εφαρμοστεί ένας τρόπος που θα χρησιμοποιούσε πολλαπλές πηγές φωτός. Ύστερα από τη σχετική έρευνα και αναζήτηση, αποφασίστηκε να χρησιμοποιηθεί μια βιβλιοθήκη γνωστή ως

“Deferred Pipeline”, η οποία ουσιαστικά χρησιμοποιούσε την τεχνική του “Deferred Rendering” ή “Deferred Shading” της σκηνής [7].

Όπως και η βιβλιοθήκη που χρησιμοποιήθηκε για το χαρακτήρα, έτσι και η “Deferred Pipeline” ενσωμάτωνε διάφορες μεθόδους και HLSL αρχεία, που ήταν απαραίτητα για την απεικόνιση της σκηνής. Μπορούμε να παρατηρήσουμε ότι αναφέρεται η λέξη «σκηνή» και όχι «μοντέλο». Αυτό συμβαίνει επειδή η τεχνική του “Deferred Rendering” φωτίζει τη σκηνή σαν ένα “post process effect”, δηλαδή σαν ένα εφέ που εφαρμόζεται σε όλη τη σκηνή (οτιδήποτε φαίνεται στην οθόνη έχει πάνω του αυτό το εφέ). Δεν θα αναλυθεί περαιτέρω αυτή η τεχνική γιατί αποτελεί ένα τεράστιο κεφάλαιο από μόνη της αλλά αξίζει να σημειωθεί ότι βασίζεται στη χρήση “Render Targets” (4 για την ακρίβεια) εκ των οποίων ο ένας υπολογίζει το χρώμα της σκηνής, ο άλλος τα normals, ο άλλος το depth και ο τελευταίος το φως. Στο τέλος γίνεται ο συνδυασμός όλων των Render Targets και προκύπτει η τελική εικόνα.

Όσον αφορά το texturing της σκηνής, αυτό συνδέεται άμεσα με το Deferred Rendering καθώς η βιβλιοθήκη “Deferred Pipeline” διαθέτει ένα custom processor, ο οποίος χρησιμοποιήθηκε μαζί με το μοντέλο του διαμερίσματος και ο οποίος χειρίζεται τα διάφορα textures του μοντέλου (diffuse, normal, κ.τ.λ.) και σε συνδυασμό με τη χρήση HLSL αρχείων πραγματοποιεί τους απαραίτητους υπολογισμούς ώστε να εμφανιστεί η τελική εικόνα της σκηνής και του μοντέλου.

Κλείνοντας πρέπει να τονιστεί ότι και πάλι οι δυσκολίες ήταν αρκετές και διάφορα μη σαφή σημεία του “Deferred renderer” αλλά και η φιλοσοφία του σαν τεχνική rendering, δεν βοήθησαν ώστε να επιτευχθεί το επιθυμητό αποτέλεσμα από πλευράς φωτισμού αλλά και από πλευράς texturing της σκηνής και του μοντέλου αντίστοιχα.

Η κατά κάποιο τρόπο «διαφάνεια» (που στην ουσία δεν είναι διαφάνεια) του χαρακτήρα όταν αυτός προστέθηκε στη σκηνή, οφειλόταν στους πολλαπλούς render targets και συγκεκριμένα, επειδή ο pixel shader του deferred renderer είχε σαν έξοδο τρεις χρωματικές πληροφορίες (color, normal, depth), έπρεπε να τροποποιηθεί αντίστοιχα και ο pixel shader του χαρακτήρα έτσι ώστε να «συμφωνούν» οι έξοδοι και να διορθωθεί το πρόβλημα. Παρ’ όλα αυτά ο “Deferred renderer” δεν βοήθησε τελικά και το αποτέλεσμα για άλλη μια φορά δεν μας άφησε ικανοποιημένους, οδηγώντας μας στην επεξεργασία μιας άλλης λύσης για το μέλλον.

Λογική παιχνιδιού

Σειρά πήρε ο προγραμματισμός της «λογικής» της πρώτης σκηνής. Αφού πλέον το κομμάτι των απεικονίσεων του χαρακτήρα και του διαμερίσματος είχε ολοκληρωθεί (η απόφαση ήταν καθαρά δική μας, να αφήσουμε τη σκηνή από πλευράς rendering ως είχε καθώς θεωρήσαμε ότι πιθανές διορθώσεις θα ήταν αρκετά χρονοβόρες και πιθανότατα μη εφικτές, πράγμα που στην τελική δεν άξιζε καθώς ετοιμάζαμε στο μυαλό μας τη δεύτερη σκηνή που θα αποτελούσε και το κύριο σημείο αναφοράς για εμάς) το ζητούμενο πλέον ήταν να προγραμματιστεί η "game logic".

Αρχικά ο χαρακτήρας δεν έπρεπε να περνάει μέσα από τους τοίχους και τα διάφορα αντικείμενα, έτσι ώστε να δίνεται η αίσθηση των «ορίων» του χώρου. Αυτό γίνεται εφικτό με τη δημιουργία "Collision Detection" και "Collision Response" λογικής μέσα στο παιχνίδι [3]. Κατά κάποιο τρόπο η συγγραφή τέτοιου είδους κώδικα, απαντά σε προσομοίωση φυσικής στο παιχνίδι και άρα αποτελεί κομμάτι της «μηχανικής φυσικής» του παιχνιδιού. Το XNA βοηθάει αρκετά στη δημιουργία τέτοιας λογικής, καθώς διαθέτει αρκετές μεθόδους που είναι απαραίτητες για τον εντοπισμό συγκρούσεων. Αυτό που δεν διαθέτει όμως είναι η «λογική» αυτή καθαυτή που είναι και το σημαντικότερο στοιχείο για την ανάπτυξη τέτοιου είδους κώδικα.

Γνωρίζαμε ότι το μοντέλο του διαμερίσματος ήταν ένα αρκετά πολύπλοκο μοντέλο αλλά και ότι η επιθυμία ήταν να υπάρχει μια αρκετά ακριβής εντόπιση συγκρούσεων ανάμεσα στο χαρακτήρα και στο σπίτι. Οι επιλογές μας λοιπόν ήταν περιορισμένες, καθώς για τέτοιου είδους ανίχνευση συγκρούσεων (με τέτοιο βαθμό ακρίβειας) οι μέθοδοι είναι αρκετά περιορισμένες. Έτσι λοιπόν αποφασίστηκε να χρησιμοποιηθεί μια βιβλιοθήκη γνωστή ως "Triangle Pipeline", η οποία είχε σαν βασική μέθοδο τη "ray per triangle" εντόπιση συγκρούσεων [3].

Με λίγα λόγια η ιδέα της μεθόδου είναι η ύπαρξη μιας ακτίνας, ο υπολογισμός των τριγώνων από τα οποία αποτελείται το μοντέλο (εδώ το σπίτι) και τέλος ο έλεγχος για το αν η ακτίνα εισχωρεί σε κάποιο από τα δεδομένα triangles. Όπως μπορούμε εύκολα να συμπεράνουμε μια τέτοια μέθοδος είναι πάρα πολύ ακριβής, αρκετά πολύπλοκη από πλευράς κώδικα αλλά και πάρα πολύ απαιτητική (σε πόρους). Έτσι λοιπόν αφού μετά από αρκετή προσπάθεια και χρόνο καταφέραμε να την προσαρμόσουμε στις ανάγκες μας, όταν τρέξαμε το παιχνίδι, το framerate ήταν

μηδαμινό!!! Γι' αυτό λοιπόν προσανατολιστήκαμε στη χρήση "Bounding spheres" για τον χαρακτήρα και στην κατασκευή ενός "low-poly" μοντέλου ίδιου με αυτό του κανονικού σπιτιού, με σκοπό τη χρήση του "low-poly" μοντέλου για την παραπάνω μέθοδο εντοπισμού συγκρούσεων. Αυτή τη φορά τα πράγματα βελτιώθηκαν πάρα πολύ, αλλά η ακρίβεια μειώθηκε σχετικά αρκετά.

Για τον εντοπισμό συγκρούσεων της κάμερας η λογική ήταν διαφορετική. Χρησιμοποιήθηκε μια μέθοδος εντοπισμού συγκρούσεων «σφαίρα με σφαίρα», δηλαδή μια bounding sphere για την κάμερα και αρκετές bounding spheres για το εσωτερικό του διαμερίσματος (κυρίως για τους τοίχους και τα δοκάρια των πορτών) [1]. Το αποτέλεσμα ήταν πάρα πολύ ικανοποιητικό, χωρίς όμως και πάλι να είναι τέλειο.

Κλείνοντας πρέπει να αναφερθεί ότι το κομμάτι του «εντοπισμού συγκρούσεων» σε ένα παιχνίδι είναι κάτι πάρα πάρα πολύ σημαντικό, αλλά και κάτι πάρα πολύ απαιτητικό και σύνθετο. Γι' αυτό λοιπόν το αποτέλεσμα των συγκρούσεων του χαρακτήρα με το σπίτι και της κάμερας με το σπίτι, παρά το ότι διέθετε αρκετά λάθη, κρίνεται στοιχειωδώς ικανοποιητικό δίνοντας μια σχετική αίσθηση ύπαρξης ορίων στο χώρο.

Αλληλεπιδράσεις – Gameplay

Αμέσως μετά πήρε σειρά η δημιουργία ενός «δείγματος» gameplay. Με βάση το σενάριο αλλά και κυρίως με βάση τον περιορισμένο χρόνο, αποφασίστηκε ο προγραμματισμός κάποιων «αλληλεπιδράσεων» με αντικείμενα του χώρου (του σπιτιού) με σκοπό να δοθεί έστω μια μικρή αίσθηση της ύπαρξης gameplay στο παιχνίδι. Οι αλληλεπιδράσεις αυτές θα ήταν τρεις και θα λάμβαναν χώρα ανάμεσα, στο χαρακτήρα και την πόρτα της τραπεζαρίας, στο χαρακτήρα και την πόρτα του υπνοδωματίου και στο χαρακτήρα και το στερεοφωνικό που βρίσκονταν στο υπνοδωμάτιο. Το ζητούμενο λοιπόν ήταν ο χαρακτήρας όταν βρίσκεται πίσω από την πόρτα και πατήσουμε κάποιο κουμπί (το «Ε» συγκεκριμένα) να ανοιγοκλείνει την πόρτα, ή όταν βρίσκεται μπροστά από το στερεοφωνικό και πατήσει κάποιο κουμπί

(το “Space” συγκεκριμένα) να παίζει μουσική. Πριν παρουσιαστούν συνοπτικά οι τρόποι που κατέστησαν δυνατά τα παραπάνω να σημειωθεί ότι για λόγους χρονικής πίεσης κυρίως, το παίξιμο της μουσικής δεν απαιτεί από το χαρακτήρα να βρίσκεται μπροστά στο στερεοφωνικό, καθώς αυτό θα απαιτούσε επιπλέον κώδικα και θα κόστιζε σε χρόνο (πάντως η λογική του δεν διαφέρει από αυτή των πορτών).

Η δημιουργία αλληλεπίδρασης με τις πόρτες βασίστηκε σε «εντοπισμό συγκρούσεων» και σε «εκτέλεση ενέργειας» [2]. Χρησιμοποιήθηκαν δηλαδή εν ολίγοις δυο «αόρατες» σφαίρες που έπαιζαν το ρόλο της εμβέλειας ώστε να είναι υπό προϋποθέσεις δυνατή η εκτέλεση των ενεργειών που αφορούσαν στο άνοιγμα και το κλείσιμο της πόρτας . Επίσης υπήρχε μια επιπλέον ιδιότητα με βάση την οποία, αν ο χαρακτήρας βρισκόταν σε μικρότερη απόσταση από το καθορισμένο όριο, η πόρτα δεν άνοιγε ή δεν έκλεινε.

Όσον αφορά τον ήχο, επρόκειτο για επεξεργασία ενός αρχείου ήχου, με τη βοήθεια ενός εργαλείου του XNA που ειδικεύεται στον τομέα του ήχου. Το εργαλείο αυτό είναι γνωστό ως “XACT tool”. Η χρήση του ήχου σε ένα παιχνίδι σε σχέση με τα υπόλοιπα στοιχεία του παιχνιδιού, μπορούμε να πούμε ότι αναπτύσσεται σχετικά πιο εύκολα. Συγκεκριμένα εδώ έγινε και χρήση « 3D ήχου» για να δοθεί η αίσθηση της ακοής από το χαρακτήρα. Έτσι λοιπόν σύμφωνα με την πηγή του ήχου και με τον προσανατολισμό και την τοποθέτηση του χαρακτήρα, ο ήχος καθώς και η έντασή του μεταβάλλονται έχοντας σαν βάση τα κανάλια του ήχου [1].

Κλείνοντας μπορούμε να πούμε ότι τα στοιχεία αυτά των αλληλεπιδράσεων λειτούργησαν σχετικά καλά, αλλά σε καμία περίπτωση δεν αποτελούν ρεαλιστικές αλληλεπιδράσεις που μπορεί κανείς να συναντήσει σε άλλα παιχνίδια.

Συμπεράσματα

Σαν επίλογο στο κεφάλαιο του «προγραμματισμού» της πρώτης σκηνής μπορούμε να πούμε ότι σε καμία περίπτωση δεν αποτελεί «demo» και σε καμία περίπτωση δεν αποτελεί σημείο αναφοράς. Παρόλα αυτά θα μπορούσε να αντιμετωπιστεί σαν μια «real-time» εφαρμογή και συγκεκριμένα την πρώτη προσπάθεια κατασκευής ενός demo, που στην ουσία εφαρμόστηκαν αρχές του «προγραμματισμού παιχνιδιών»,

μελετήθηκαν στοιχεία , ερευνήθηκαν δυνατότητες του XNA, αποκτήθηκε εμπειρία και τελικά αναπτύχθηκε ομαδικό πνεύμα και συνεργασία για την επίτευξη ενός στόχου που αν μη τι άλλο παραμένει εφικτός με το πέρασμα στη «δεύτερη σκηνή».

Χαρακτήρας 2ης σκηνής

Σε δεύτερη φάση προγραμματιστικά μας απασχόλησε η «δεύτερη σκηνή» και συγκεκριμένα ο προγραμματισμός του χαρακτήρα. Πριν γίνει αναφορά στα κυριότερα σημεία του προγραμματισμού του χαρακτήρα, θα γίνει μια σύντομη περιγραφή του σκοπού, του σκεπτικού και του στόχου της δεύτερης σκηνής του demo.

Η δεύτερη σκηνή λοιπόν αναπτύχθηκε με βάση το αρχικό σενάριο, σαν συνέχεια σε ένα demo με δύο σκηνές. Στην πραγματικότητα όμως τα γεγονότα διαφέρουν. Η δεύτερη σκηνή σε ρεαλιστική βάση αποτελεί για εμάς το σημείο αναφοράς. Ο χρόνος που απέμεινε από την ολοκλήρωση της πρώτης σκηνής και κατά συνέπεια ο χρόνος που είχαμε στη διάθεσή μας για να ολοκληρώσουμε τη δεύτερη σκηνή, ήταν αρκετά περιορισμένος. Παρόλα αυτά συγκεντρωθήκαμε και προσηλωθήκαμε στην ανάπτυξη της δεύτερης σκηνής (η οποία στην ουσία για εμάς αποτελούσε κάτι ξεχωριστό, μια άλλη «πρώτη σκηνή»), έχοντας ως κύριο άξονα στο μυαλό μας τη διδαχή από τα προηγούμενα λάθη και την ανεύρεση νέων μεθόδων για την δημιουργία των απαραίτητων για την σκηνή στοιχείων.

Ο χαρακτήρας της δεύτερης σκηνής, ήταν ένας εντελώς διαφορετικός χαρακτήρας και προγραμματιστικά αλλά και από άποψη γραφικών. Οι απαιτήσεις και ο σκοπός μας δεν διέφερε από αυτόν του χαρακτήρα της πρώτης σκηνής. Μόνο που αυτή τη φορά κινηθήκαμε εντελώς διαφορετικά. Συγκεκριμένα λοιπόν ο καινούργιος χαρακτήρας θα ήταν ένα direct3D (.x) μοντέλο σε αντίθεση με το fbx μοντέλο της πρώτης σκηνής. Αυτή η επιλογή έγινε με γνώμονα τη λανθασμένη επιλογή που είχε γίνει στην πρώτη σκηνή. Τώρα λοιπόν η βιβλιοθήκη που χρησιμοποιήθηκε ύστερα από αρκετό ψάξιμο, ήταν η «KiloWatt Animation Library» [5]

Η συγκεκριμένη βιβλιοθήκη θεωρείται από τις κορυφαίες (στον τομέα του XNA) για ζητήματα που έχουν να κάνουν με animated μοντέλα. Υποστηρίζει direct3D και fbx μοντέλα και επίσης διαθέτει μεθόδους για «animation blending», «animation transition », «compositing» και έχει την δυνατότητα να αναπαράγει animation με η χωρίς «skinning» [5]. Να σημειωθεί επίσης ότι συνεργάζεται άριστα με την εφαρμογή «3D Studio Max» και συγκεκριμένα με τον exporter της εφαρμογής τον «kw-xport». Άρα το μοντέλο του χαρακτήρα γίνεται animated στο 3D studio max, εξάγεται με τον kw-xport και εισάγεται στο XNA μέσω ενός custom processor που διαθέτει η βιβλιοθήκη, γνωστό ως « KiloWatt Animation Processor».

Ο συγκεκριμένος processor χειρίζεται πληροφορίες που έχουν να κάνουν με «εξαίρεση συγκεκριμένων animation clip», «DXT5 texture format», «επιλογή skinned ή non-skinned shader», «μέγιστο αριθμό bones», «Tolerance για αυξομείωση της ποιότητας του animation»,κ.ά.

Απ' ότι καταλαβαίνουμε η διαφορά μεταξύ της βιβλιοθήκης που χρησιμοποιήθηκε για τον χαρακτήρα της πρώτης σκηνής με αυτή που χρησιμοποιήθηκε στη δεύτερη σκηνή είναι πάρα πολύ μεγάλη. Με λίγα λόγια ο χαρακτήρας με πολλαπλά animation εισήχθηκε στο XNA και χρησιμοποιώντας ένα «skinned shader» απεικονίστηκε τελικά στην οθόνη. Οι διορθώσεις και οι τροποποιήσεις που ακολούθησαν ήταν αρκετές κυρίως στον τομέα των animation clips αλλά και στον χειρισμό του χαρακτήρα συναρτήσει και της κάμερας.

Τελικά το αποτέλεσμα ήταν αρκετά ικανοποιητικό, σε σημείο που μας οδήγησε στον να απορίσουμε για την τρομερή βελτίωση στον τομέα της «εμφάνισης» και των «animation», σε σχέση με τον χαρακτήρα της πρώτης σκηνής. Σαν μειονέκτημα μπορούμε μόνο να πούμε ότι η «KiloWatt animation library» δεν χρησιμοποιείται για διδακτικούς σκοπούς, κι αυτό έχει ως συνέπεια την αυξημένη πολυπλοκότητά του και τη μειωμένη υποστήριξη σε «βοηθητικό υλικό» που διαθέτει.

Περιβάλλον χώρος 2ης σκηνής

Το επόμενο στοιχείο της δεύτερης σκηνής που μας απασχόλησε προγραμματιστικά ήταν ένα εξίσου πολύ σημαντικό στοιχείο, και ήταν αυτό του περιβάλλοντα χώρου.

Σαν περιβάλλον χώρο είχαμε αποφασίσει να χρησιμοποιήσουμε το μοντέλο ενός αρχαιολογικού μνημείου, του «Οθωμανικού λουτρού» της Μυτιλήνης. Αυτό γιατί αρχικά θα έδινε άλλη αισθητική στη σκηνή και επίσης θεωρήθηκε σαν πρόκληση από τη μεριά μας και το προγραμματιστικό του κομμάτι αλλά και το καλλιτεχνικό. Το ζητούμενο λοιπόν ήταν να αποφύγουμε όσο δυνατόν περισσότερο γίνεται τα λάθη της πρώτης σκηνής αλλά και να έχουμε καινούργια στοιχεία (προγραμματιστικά), τα οποία θα βοηθούσαν στην επίτευξη ενός καλύτερου αποτελέσματος (κυρίως οπτικού).

Το μοντέλο του λουτρού κατασκευάστηκε και μπήκε στο XNA, με το βάρος της δουλειάς να πέφτει στο shading του. Ο «Deferred renderer» αποτελούσε ένα κακό παρελθόν πλέον και η ανεύρεση μιας νέας τεχνικής η οποία θα υποστήριζε κυρίως τη χρήση πολλαπλών φώτων, ήταν επιτακτική. Μετά από αρκετή έρευνα και μελέτη καταλήξαμε στο να δημιουργήσουμε ένα shader ο οποίος θα ενσωμάτωνε αυτή την ιδιότητα (τη χρήση πολλών φώτων).

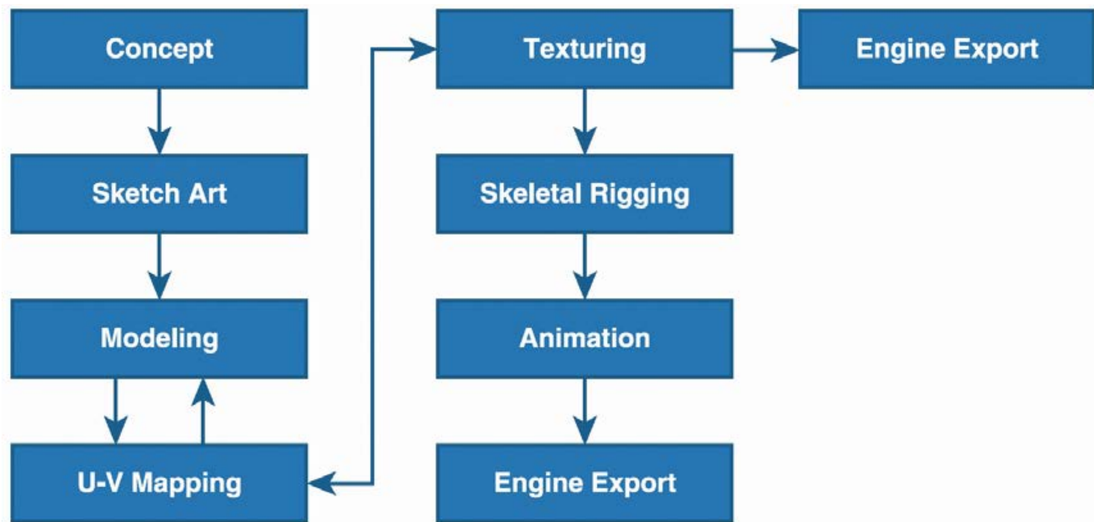
Μερικές δοκιμές και τροποποιήσεις με HLSL αρχεία ήταν αρκετές για να μας πείσουν ότι βρισκόμασταν στο σωστό δρόμο για την λύση του προβλήματος. Ο shader που κατασκευάστηκε υποστήριζε «πολλαπλά point lights» με «self shadowing» αλλά και «multitexturing» του μοντέλου [6]. Βέβαια μπορεί στον τομέα του πολλαπλού φωτισμού να βρήκαμε σχετικά εύκολα και γρήγορα τη λύση, αλλά δε συνέβη το ίδιο και με τα πολλά texture στα πολλά meshes του ίδιου όμως μοντέλου.

Το κομμάτι αυτό αποτέλεσε για αρκετό χρονικό διάστημα ένα σύνθετο πρόβλημα, κάτι το οποίο μας κόστισε και σε χρόνο μέχρι την επίλυση του. Σημασία έχει ότι τελικά επιλύθηκε και μας βοήθησε στο να προχωρήσουμε παρακάτω με τη δουλειά μας. Σε τελικό στάδιο το μοντέλο, με τη χρήση λίγων γραμμών κώδικα στο XNA και με την χρήση του προαναφερθέντος shader, απεικονίστηκε με μεγάλη επιτυχία στην οθόνη.

Κλείνοντας λοιπόν, μιας και η δουλειά στον τομέα του προγραμματισμού τελειώνει με το rendering του μοντέλου του οθωμανικού λουτρού, αξίζει να σημειωθεί ότι το αποτέλεσμα ήταν εξαιρετικά ικανοποιητικό δεδομένης της χρήσης μιας εντελώς καινούργιας για εμάς μεθόδου. Βέβαια δεν θα πρέπει να αγνοήσουμε και το γεγονός πιθανών μελλοντικών επιπλοκών, αλλά προς το παρόν η μέχρι τώρα δουλειά για τη δεύτερη σκηνή μπορεί να θεωρηθεί ότι βρίσκεται πλέον στη σωστή βάση για τη δημιουργία ενός ολοκληρωμένου demo παιχνιδιού, βασισμένο σε σύγχρονες μεθόδους «ανάπτυξης video game».

3.3. Τρισδιάστατα γραφικά

Η δημιουργία γραφικών για ηλεκτρονικά παιχνίδια είναι περίπλοκη διαδικασία, αλλά είναι αρκετά απλή και γραμμική. Παρακάτω θα γίνει μια μικρή περιγραφή για τον τρόπο παραγωγής των τρισδιάστατων γραφικών της δεύτερης σκηνής.



Εικόνα 23:Διάγραμμα δημιουργίας μοντέλων-χαρακτήρων [9]

3.3.1. Concept

Σύμφωνα με το concept ο χαρακτήρας είναι ένας φοιτητής που βασίζεται σε πραγματικό πρόσωπο. Προορίζεται για τρίτου προσώπου παιχνίδι που σημαίνει ότι πρέπει να δημιουργηθεί εξολοκλήρου. Το περιβάλλον είναι ένα οθωμανικό λουτρό που βασίζεται και αυτό σε πραγματικό χώρο.

3.3.2. Sketch art

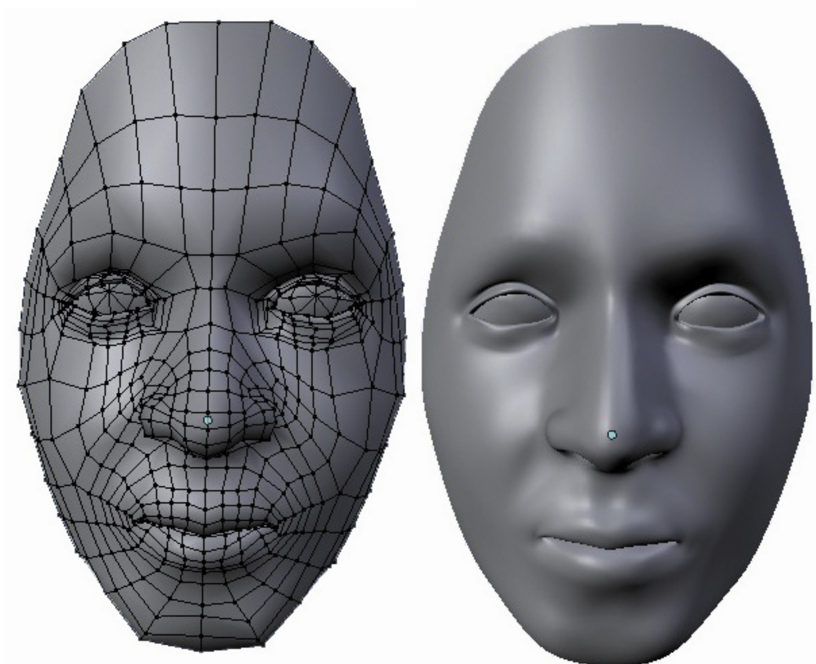
Όλα τα 3D μοντέλα είχαν ως βάση πραγματικά μοντέλα επομένως χρησιμοποιήθηκαν φωτογραφίες ως σκίτσα.

3.3.3. Modeling

Χαρακτήρας

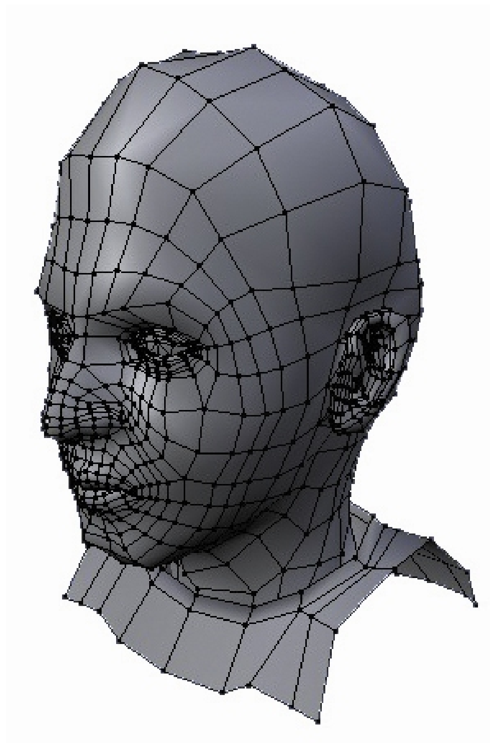
Ο χαρακτήρας δημιουργήθηκε πολύγωνο-πολύγωνο με polygonal modeling. Η στάση που έχει (ανοιχτά χέρια-πόδια) ονομάζεται T-Pose. Προτιμήθηκε αυτή η στάση γιατί γίνεται πιο εύκολη η τοποθέτηση των οστών στο χαρακτήρα και πιο εύκολο το UV mapping. Κάποιοι 3D modelers προτιμούν μια πιο « χαλαρή » στάση από την T-Pose. Με αυτόν τον τρόπο το μοντέλο παραμορφώνεται λιγότερο όταν έρθει η σειρά του animation [12-13].

Εικόνα 24



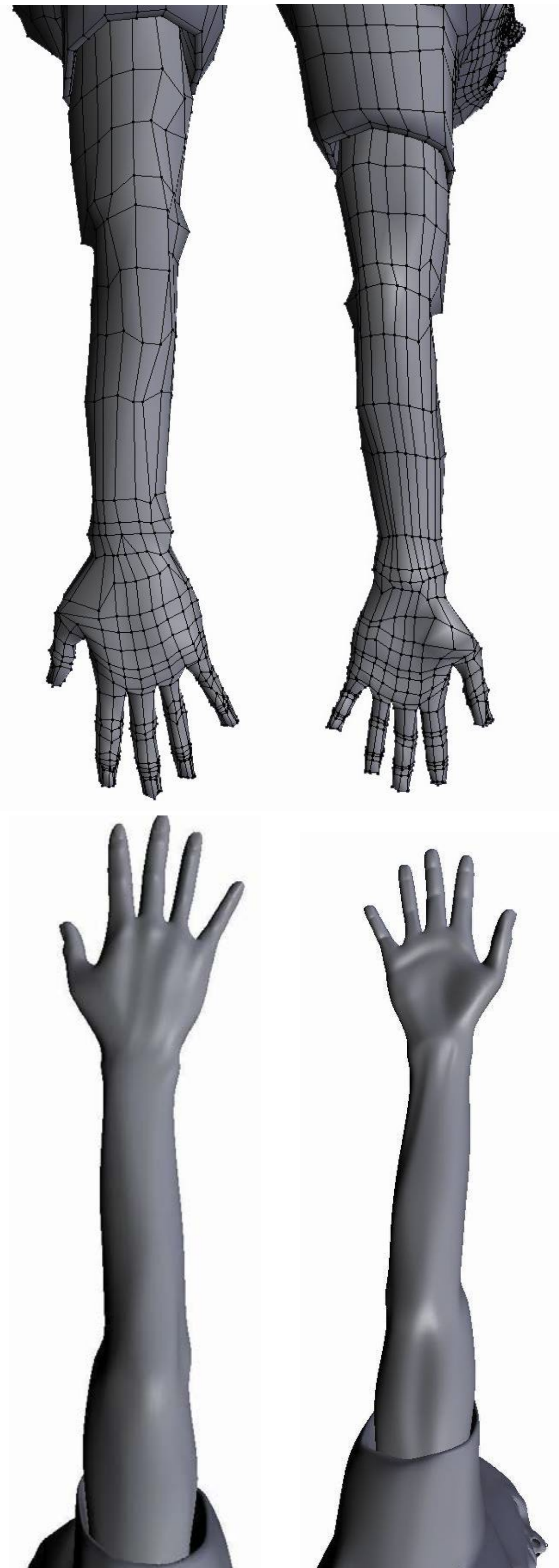
Εικόνα 25

Η μοντελοποίηση άρχισε με ένα πολύγωνο. Αρχικά δημιουργήθηκε το πρόσωπο και στη συνέχεια όλο το κεφάλι. Στην εικόνα 24 φαίνεται πώς με την χρήση του εργαλείου extrude το αρχικό πολύγωνο μετατράπηκε σε πολλά και άρχισε να σχηματίζεται το πρόσωπο. Στην εικόνα 25 το πρόσωπο έχει πλέον τελειώσει. Το βλέπουμε σε wireframe και με τη χρήση του εργαλείου subdivide.

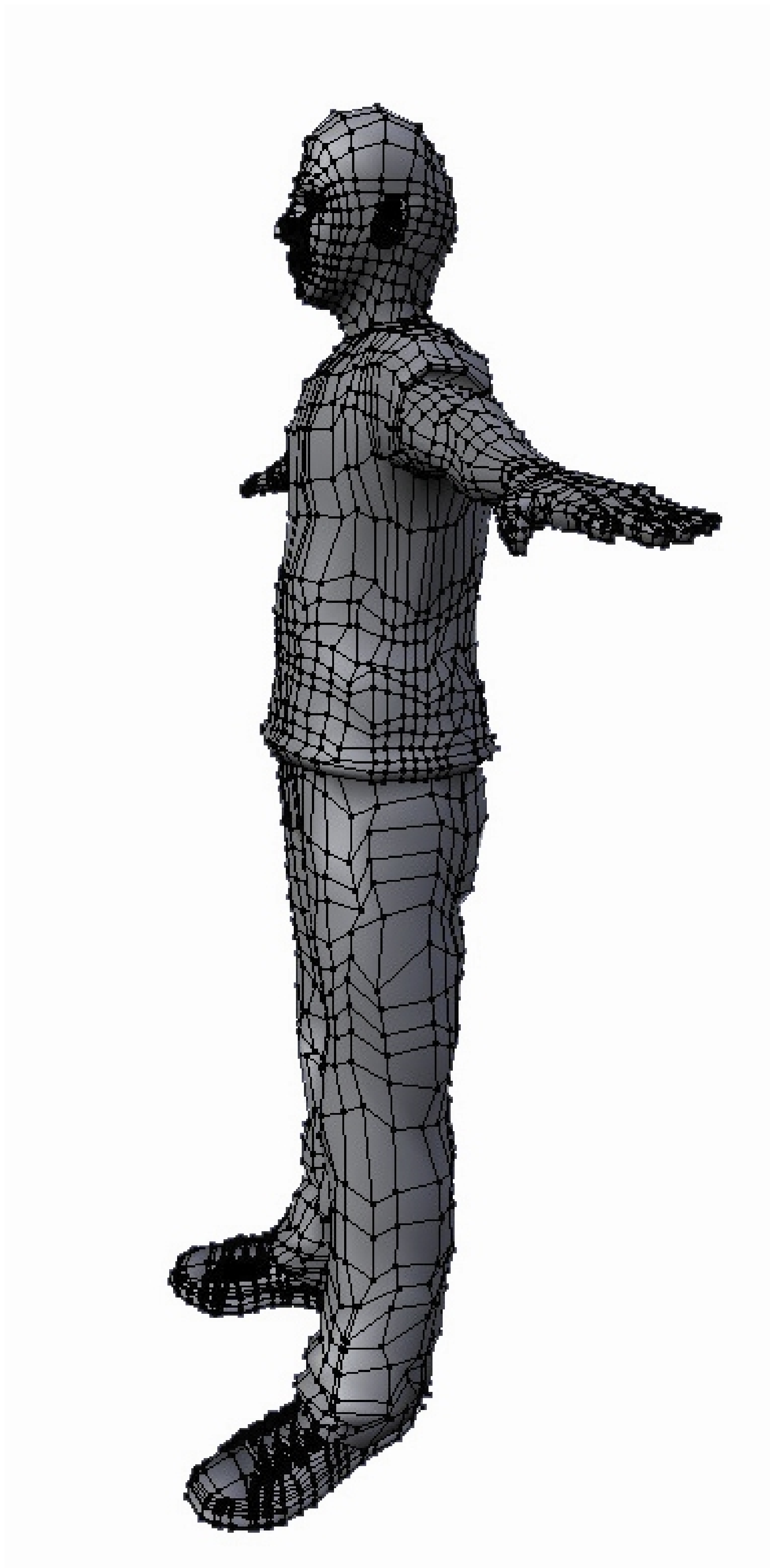


Εικόνα 26: Το κεφάλι ολοκληρωμένο σε wireframe

Εικόνα 27: Το κεφάλι με την χρήση του εργαλείου subdivide



Εικόνα 28: Τα χέρια ολοκληρωμένα σε wireframe και με subdivide

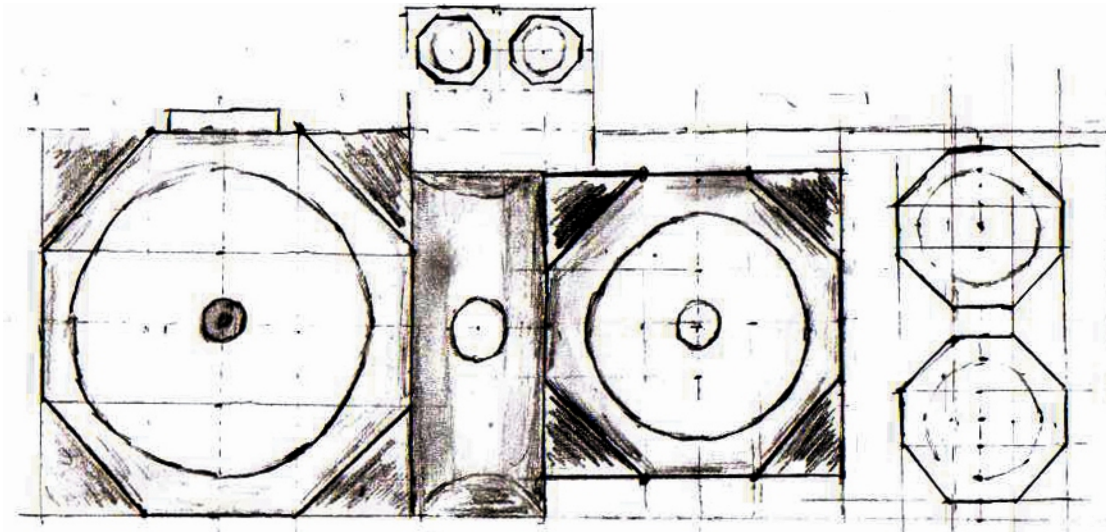


Εικόνα 29: Ο χαρακτήρας ολοκληρωμένος σε wireframe 6959 πολύγωνα

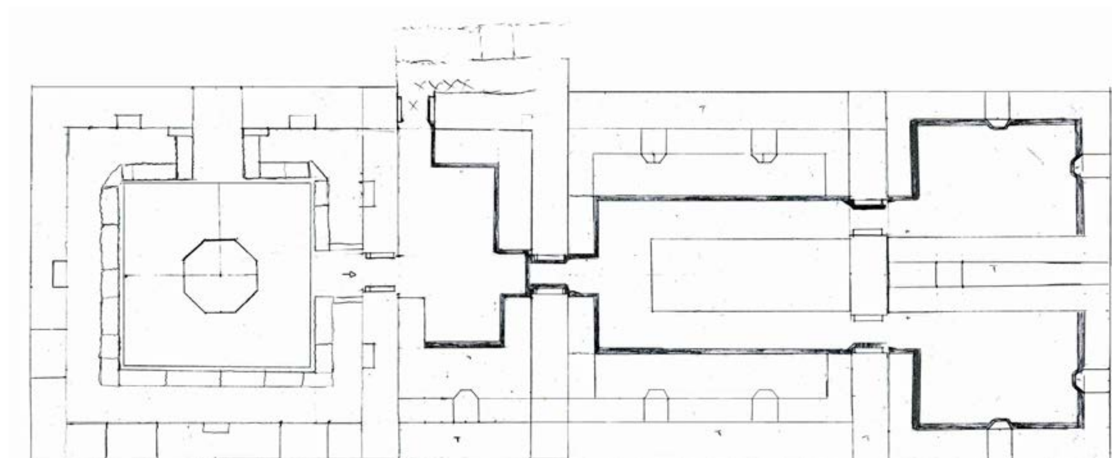
Εικόνα 30 : Ο χαρακτήρας ολοκληρωμένος με την χρήση του εργαλείου subdivide

Οθωμανικό λουτρό

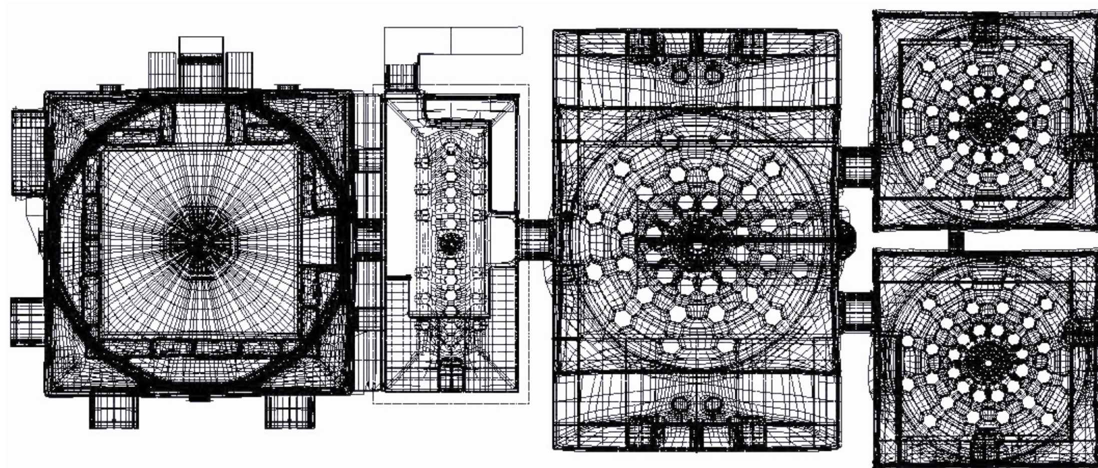
Το οθωμανικό λουτρό κατασκευάστηκε βάση κάποιων φωτογραφιών που πάρθηκαν από τον χώρο. Σύμφωνα με τις φωτογραφίες δημιουργήθηκε μια κάτοψη του χώρου έτσι ώστε να γίνει στην συνέχεια η μοντελοποίηση [12].



Εικόνα 31 : Προσχέδιο του λουτρού

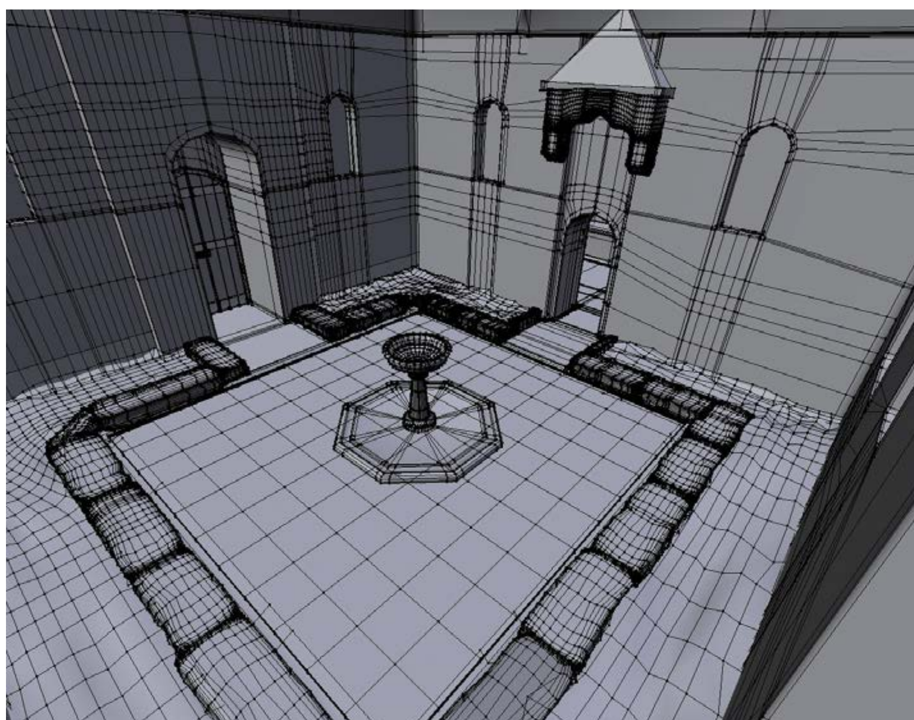


Εικόνα 32 : Η κάτοψη του λουτρού

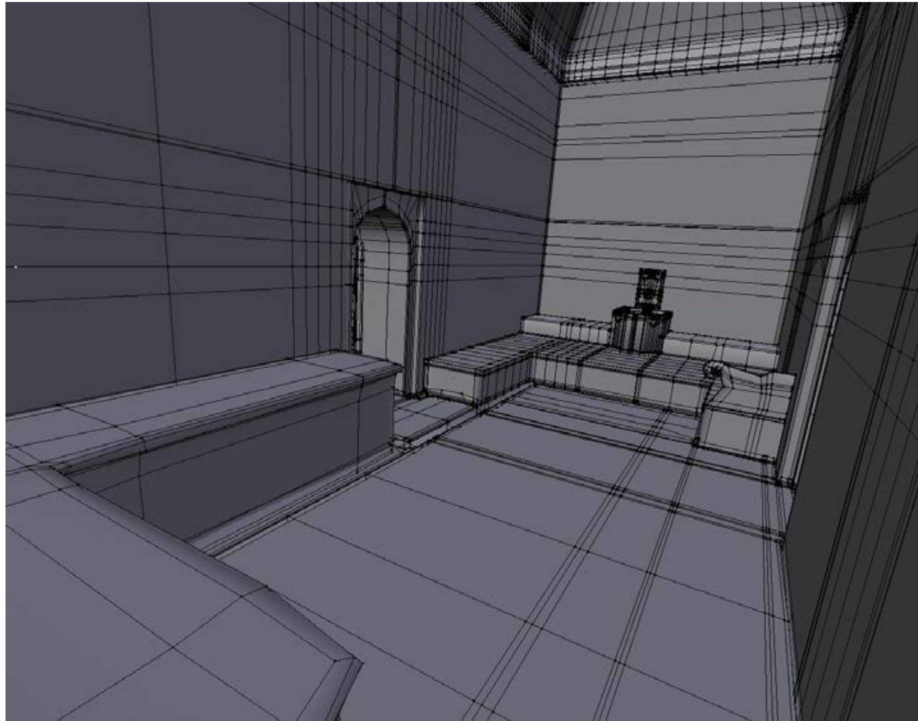


Εικόνα 33 : Το ολοκληρωμένο 3D μοντέλο

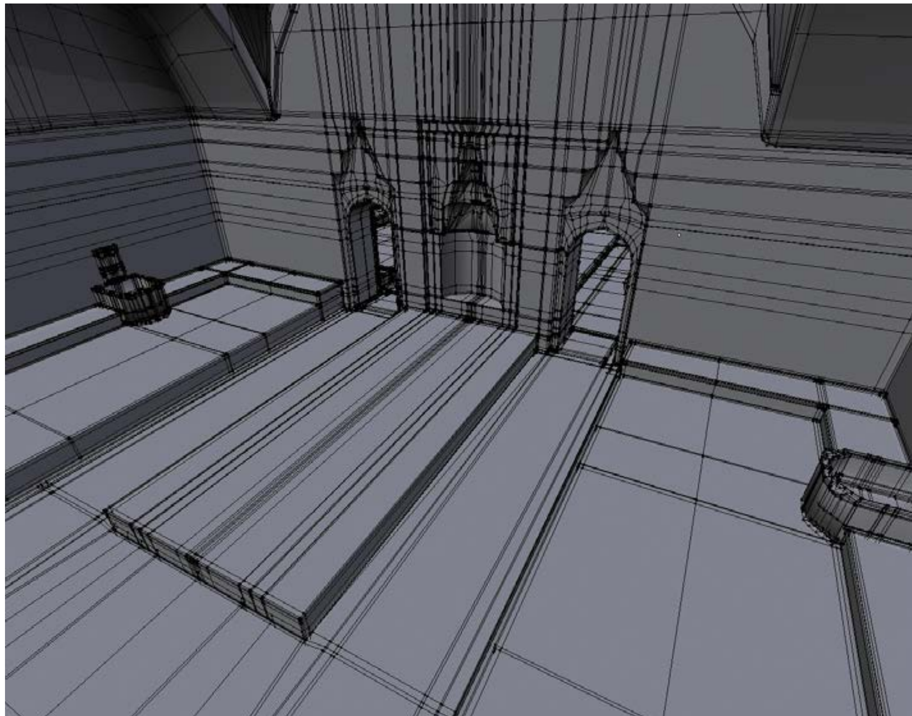
Η μοντελοποίηση του οθωμανικού λουτρού έγινε αίθουσα-αίθουσα. Η κατασκευή του έγινε με polygonal modeling.



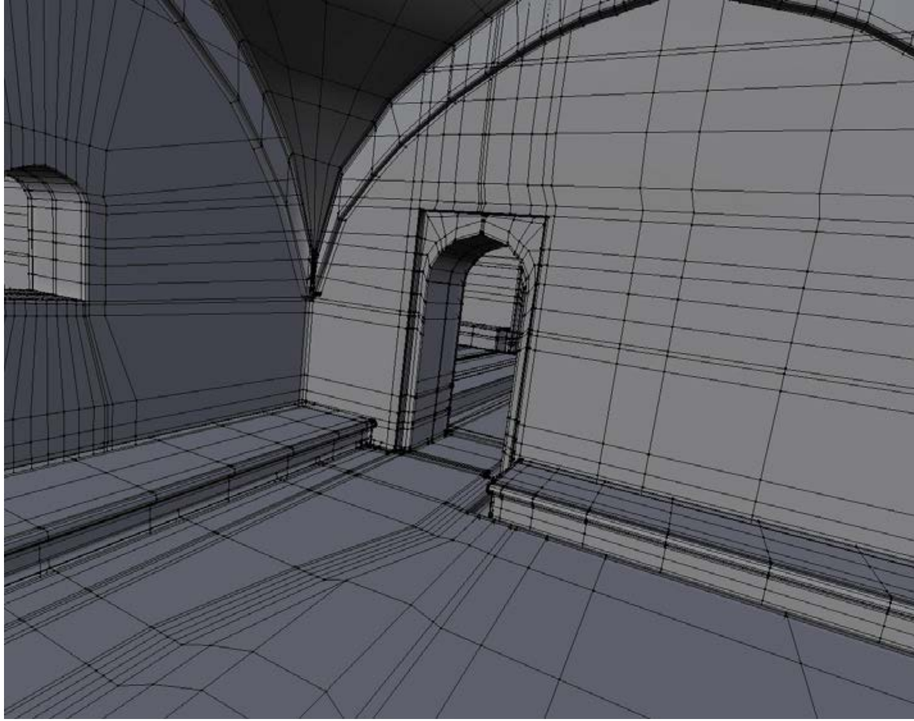
Εικόνα 34 : Η πρώτη αίθουσα του λουτρού



Εικόνα 35 : Η δεύτερη αίθουσα του λουτρού



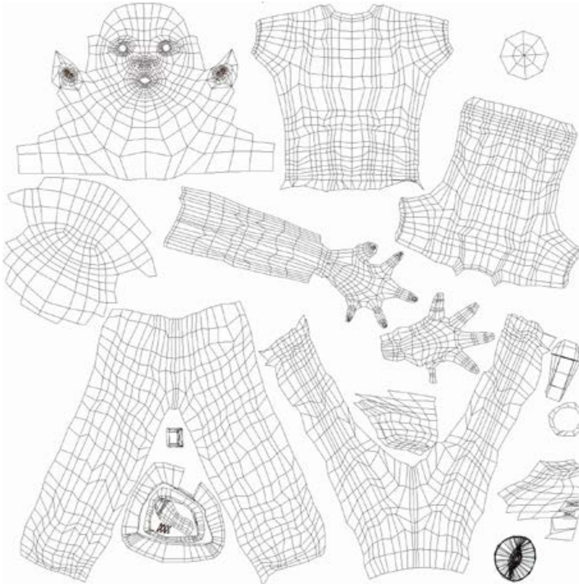
Εικόνα 36 : Η τρίτη αίθουσα του λουτρού



Εικόνα 37 : Η τέταρτη αίθουσα του λουτρού

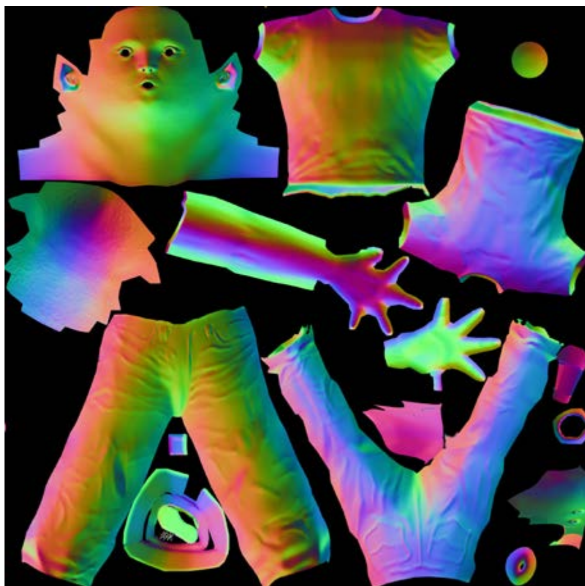
3.3.4. Texture maps

Αρχικά έγιναν τα UV maps του χαρακτήρα και των μοντέλων του λουτρού και στην συνέχεια χρωματίστηκαν έτσι ώστε να χρησιμοποιηθούν για diffuse mapping, ambient occlusion mapping και normal mapping [8-11].



Εικόνα 38: Το UV map του χαρακτήρα. Για την δημιουργία του χρησιμοποιήθηκαν cylindrical και planar mapping

Εικόνα 39: Diffuse map



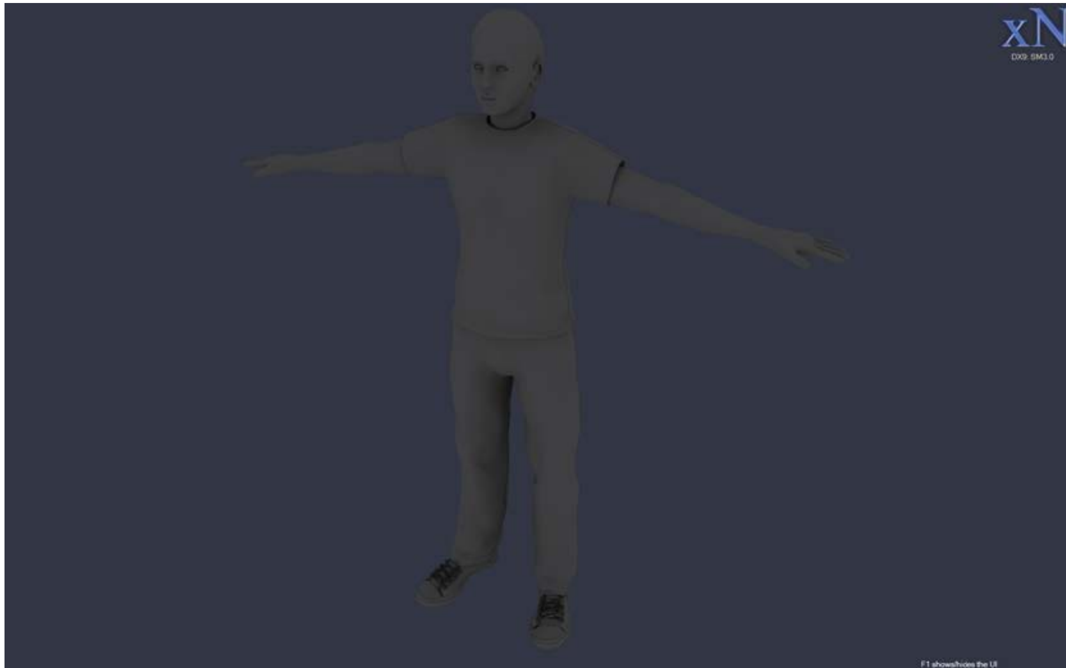
Εικόνα 40 : Normal map

Εικόνα 41 : Ambient occlusion map

Ακολουθούν κάποιες δοκιμές εφαρμογής των παραπάνω textures με την χρήση του xNormal, ενός προγράμματος που μπορεί να ρεντάρει 3D μοντέλα σε πραγματικό χρόνο.

Εικόνα 42 : Ο χαρακτήρας με την χρήση μόνο diffuse map

Εικόνα 43 : Ο χαρακτήρας με τη χρήση μόνο normal map



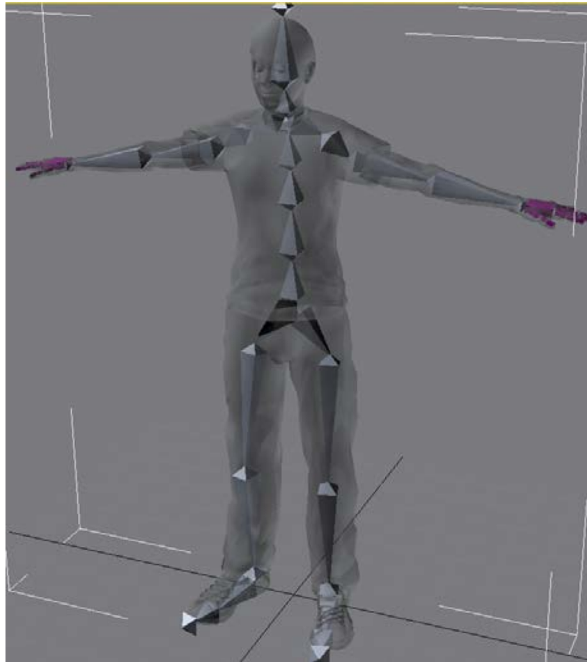
Εικόνα 44 : Ο χαρακτήρας με την χρήση μόνο ambient occlusion map

Εικόνα 45: Ο χαρακτήρας με την χρήση diffuse, normal και ambient occlusion map. Παρατηρούμε πως σκιάστηκε ο χαρακτήρας καθώς και την καλύτερη υφή και λεπτομέρεια που πήραν τα ρούχα του.

3.3.5. Rigging

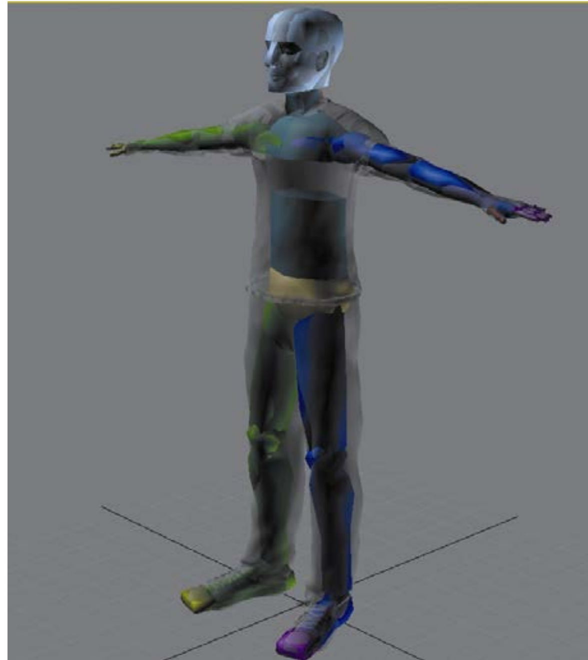
Ο .x exporter του blender (2.49a) δεν μπορεί να χαρακτηριστεί σε καμία περίπτωση ολοκληρωμένος. Δημιούργησε αρκετά προβλήματα όσον αφορά την συνεργασία του με το XNA σε ακίνητα αντικείμενα, πόσο μάλλον σε κινούμενα. Για τον λόγο αυτό χρησιμοποιήθηκε το 3ds Max για την δημιουργία animation στον χαρακτήρα.

Αρχικά δημιουργήθηκε ένας custom σκελετός. Αν και στην πρώτη σκηνή που τα αντικείμενα γίνονταν export σε .fbx ένας τέτοιου είδους σκελετός λειτούργησε, στην δεύτερη που γίνεται χρήση ενός πιο εξελιγμένου animation player (KW) και του kW X-porter φαίνεται να μην λειτουργεί.



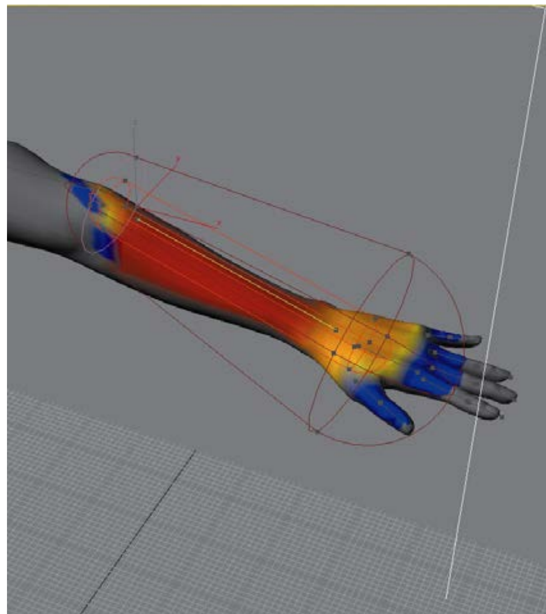
Εικόνα 46:Ο πρώτος σκελετός του χαρακτήρα

Η λύση δόθηκε με την χρήση του σκελετού biped του 3ds Max, που είναι πλήρως συμβατός με τον kW.



Εικόνα 47: Σκελετός biped

Στην συνέχεια ακολουθεί το skinning, η διαδικασία δηλαδή ενσωμάτωσης του σκελετού στο mesh του χαρακτήρα. Με αυτόν τον τρόπο τα vertices του χαρακτήρα θα χωριστούν σε ομάδες, που θα ελέγχονται από οστά.



Εικόνα 48: Το μέρος από το mesh που ελέγχεται από ένα οστό

3.3.6. Animation

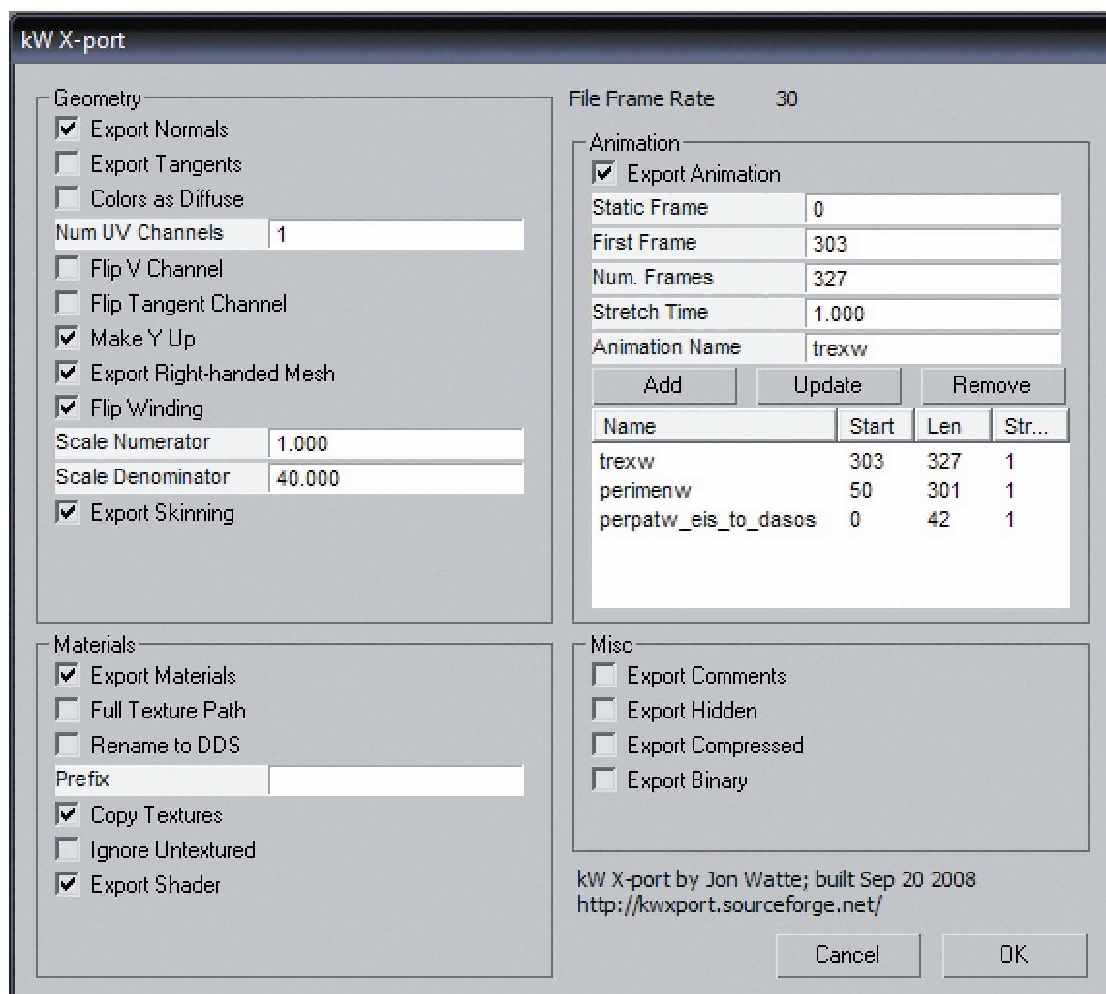
Εφόσον ολοκληρωθεί και το skinning, το μοντέλο είναι έτοιμο για animation. Δημιουργήθηκαν τρεις βασικές κινήσεις με keyframes. Στην πρώτη ο χαρακτήρας κάνει μια «ελαφριά» κίνηση σαν να αναπνέει, στην δεύτερη περπατάει και στην τρίτη τρέχει.

Εικόνα 49:Περπάτημα χαρακτήρα

Εικόνα 50 : Τρέξιμο χαρακτήρα

3.3.7. Engine export

Τα 3D μοντέλα εξάχθηκαν σε .x μορφή με τον kW X-porter που συνεργάζεται άμεσα με τον kW animation player στο XNA. Έτσι μπόρεσαν να εξαχθούν με επιτυχία οι κινήσεις του χαρακτήρα, πράγμα αδύνατο να γίνει στην πρώτη σκηνή που τα μοντέλα εξάγονταν με τον λιγότερο εξελιγμένο Autodesk(.fbx) exporter.



Εικόνα 51: Ο exporter που χρησιμοποιήθηκε για όλα τα μοντέλα της δεύτερης σκηνής

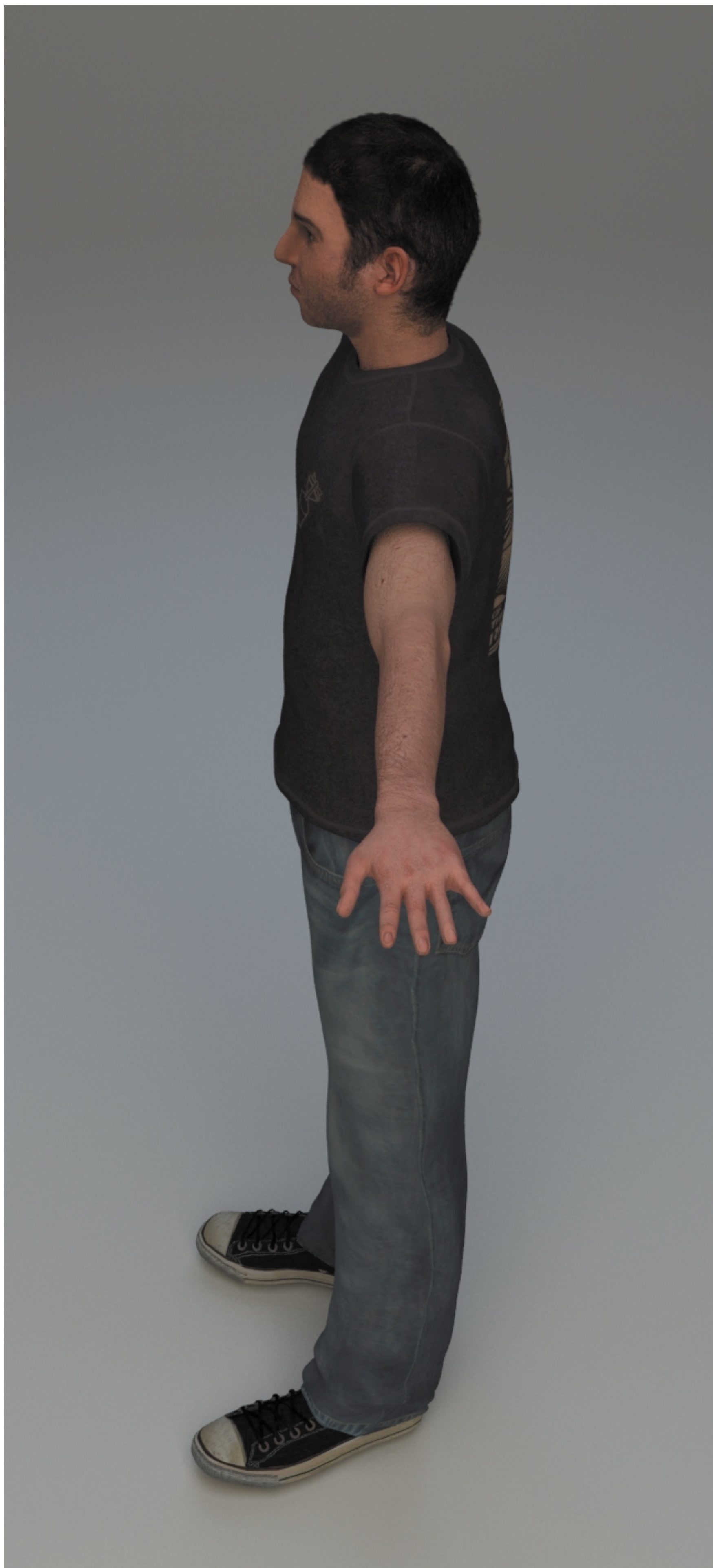
3.3.8. Rendering

Το rendering σε μη πραγματικό χρόνο (pre rendering) στα ηλεκτρονικά παιχνίδια έχει να κάνει κυρίως με την αναπαραγωγή βίντεο(cut scenes) και την δημιουργία background εικόνων. Στο pre rendering μπορούν να χρησιμοποιηθούν γραφικά πολύ πιο σύνθετα από αυτά που χρησιμοποιούνται στο real-time rendering, λόγω της δυνατότητας χρήσης πολλών Η/Υ για μεγάλα χρονικά διαστήματα. Το μειονέκτημά του είναι το χαμηλότερο επίπεδο διάδρασης. Ένα παιχνίδι με pre-rendered backgrounds αναγκάζεται να χρησιμοποιεί σταθερές γωνίες λήψης ,ενώ ένα παιχνίδι με pre-render φωτισμούς δεν μπορεί να τους αλλάξει με πειστικό τρόπο.

Στην συνέχεια ακολουθούν κάποια δοκιμαστικά rendering του οθωμανικού λουτρού και του χαρακτήρα. Οι renderer που χρησιμοποιήθηκαν είναι ο LuxRender για τις εικόνες 52 και 53 και ο yafaray για τις 54 και 55.

Εικόνα 52:Η κάμερα βλέπει την τέταρτη αίθουσα το λουτρού.

Εικόνα 53: Η κάμερα βρίσκεται στην τρίτη αίθουσα του λουτρού



Εικόνα 54 : Πλάγια όψη του χαρακτήρα

Εικόνα 55 : Μπροστά και πίσω όψη του χαρακτήρα

ΚΕΦΑΛΑΙΟ 4

Συνδυασμός προγραμματισμού και γραφικών

Όπως έχει ήδη αναφερθεί, η ανάπτυξη ενός ηλεκτρονικού παιχνιδιού προϋποθέτει το συνδυασμό του τομέα των 3d γραφικών και του προγραμματισμού. Ο συνδυασμός αυτός δεν είναι κάτι εύκολο και βασίζεται στη σωστή συνεργασία, στην προσαρμογή σε ειδικές συνθήκες, στην τήρηση κάποιων κανόνων και στην αποτελεσματικότητα των ατόμων που ασχολούνται με τους αντίστοιχους τομείς.

Η λογική προγραμματισμού ηλεκτρονικών παιχνιδιών δημιουργεί αρκετούς περιορισμούς, όπως επίσης και το ίδιο το λογισμικό συγγραφής τέτοιου είδους κώδικα. Κάποιοι από τους περιορισμούς που αντιμετωπίστηκαν ήταν το μέγεθος των Textures, ο αριθμός των bones του σκελετού, ο αριθμός των πολυγώνων των μοντέλων, ο συνδυασμός και η μετάβαση μεταξύ πολυάριθμων animation και ο φωτισμός.

Όσον αφορά το μέγεθος των textures, ο περιορισμός ήταν ότι το μέγεθος έπρεπε να είναι «δύναμης του 2» και η ανάλυση των εικόνων δεν έπρεπε να ξεπερνάει τα 2048X2048 pixels (αυτό εξαρτάται από την κάρτα γραφικών και τη μνήμη της). Πολλά textures μεγάλης ανάλυσης μπορούν να προκαλέσουν πρόβλημα μνήμης στην κάρτα γραφικών και να οδηγήσουν σε μειωμένο framerate του παιχνιδιού. Από την άλλη, μικρής ανάλυσης textures μπορούν να προκαλέσουν κακή ποιότητα εικόνας στο παιχνίδι.

Όσον αφορά των αριθμό των bones του σκελετού, ο περιορισμός ήταν ότι έπρεπε να χρησιμοποιηθεί ένας μέγιστος αριθμός για κάθε σκελετό. Πιο συγκεκριμένα ο αριθμός των bones δεν έπρεπε να ξεπερνάει τα 58 σε σύνολο. Αυτό εξαρτάται και από το εκάστοτε λογισμικό, δηλαδή ο αριθμός αυτός μπορεί να αυξομειώνεται ανάλογα με το λογισμικό που χρησιμοποιείται. Όσο λιγότερα bones χρησιμοποιούνται τόσο λιγότερη είναι και η ρεαλιστικότητα της κίνησης που μπορεί να δημιουργηθεί.

Το πρόβλημα με τον αριθμό των πολυγώνων είναι καθαρά υπολογιστικό. Όσο μικρότερος είναι ο αριθμός των πολυγώνων τόσο πιο γρήγορα γίνονται οι απαραίτητοι υπολογισμοί και κατά συνέπεια βελτιώνεται και το framerate του

παιχνιδιού. Σε αντίθετη περίπτωση το framerate του παιχνιδιού, μπορεί να οδηγηθεί σε απογοητευτικά νούμερα, πράγμα μη θεμιτό. Βέβαια ο περιορισμός αυτός , με την πρόοδο της τεχνολογίας σήμερα, δεν έχει τόσο βαρύνουσα σημασία όσο είχε παλαιότερα γιατί ο αριθμός των πολυγώνων που «προσφέρεται» στις μέρες μας, επιτρέπει την κατασκευή αρκετά λεπτομερών μοντέλων.

Όσον αφορά τα πολλαπλά animation, το πρόβλημα έγκειται στην ανάθεση τους σε ένα (ή περισσότερα) μοντέλα και στις τεχνικές μετάβασης από animation σε animation με κύριο σκοπό τη ρεαλιστικότητα. Γίνεται εύκολα αντιληπτό, ότι η ανάγκη χρήσης πολλών animation σε μοντέλα του παιχνιδιού καθώς και η χρήση τεχνικών μετάβασης , είναι επιτακτική και απαραίτητη για να προσδώσει ενδιαφέρον, ρεαλισμό και λειτουργικότητα στο παιχνίδι.

Τέλος ο φωτισμός αποτελεί ένα μεγάλο και σημαντικό κεφάλαιο για ένα παιχνίδι. Οι τεχνικές και οι τρόποι φωτισμού ποικίλουν. Η χρήση τους εξαρτάται κυρίως από τις ανάγκες και το ύφος του παιχνιδιού. Για παράδειγμα ένα παιχνίδι μπορεί να χρειάζεται πολλαπλό φωτισμό ενώ κάποιο άλλο θα μπορούσε να αρκεστεί στη χρήση ενός απλού φωτός (ήλιος, κ.τ.λ). Για τη χρήση πολλαπλού φωτισμού, οι τεχνικές που χρειάζεται να αναπτυχθούν είναι αρκετά πολύπλοκες και εξεζητημένες σε αντίθεση με τη χρήση ενός απλού φωτός, το οποίο μπορεί να εφαρμοστεί με σχετικότερη ευκολία.

Όλοι αυτοί οι περιορισμοί πρέπει να είναι γνωστοί στους 3d artists και στους προγραμματιστές έτσι ώστε κατόπιν σωστής συνεργασίας, να μην αποτελούν πρόβλημα στην ανάπτυξη του παιχνιδιού.

Επίλογος

Συμπεράσματα

Σαν συμπέρασμα μπορούμε να πούμε ότι η ανάπτυξη ενός ηλεκτρονικού παιχνιδιού είναι μια διαδικασία αρκετά περίπλοκη, χρονοβόρα, απαιτητική, σύνθετη αλλά συνάμα κάτι ενδιαφέρον, δημιουργικό, επιμορφωτικό και κάτι που σίγουρα αξίζει κανείς να ασχοληθεί.

Το game demo αυτό προσέφερε γνώσεις σε δύο μεγάλους τομείς της πληροφορικής, εμπειρία, ανάπτυξη νέων ικανοτήτων, ομαδικότητα, οργανωτικότητα και αρκετά άλλα σημαντικά οφέλη.

Η δεύτερη σκηνή του demo αποτελεί σημείο αναφοράς, γιατί στηρίχθηκε σε ποιο «σωστούς» και «ολοκληρωμένους» τρόπους ανάπτυξης με αποτέλεσμα να δοθεί μια εικόνα πιο κοντά σε ένα σύγχρονο ηλεκτρονικό παιχνίδι.

Μελλοντική δουλειά

Η μελλοντική δουλειά δεν θα μπορούσε να είναι άλλη από την συνέχεια της ανάπτυξης της δεύτερης σκηνής. Αυτό σημαίνει ότι τα στοιχεία που αποτελούν ένα ολοκληρωμένο παιχνίδι θα πρέπει να βρουν εφαρμογή σε όσο το δυνατόν μεγαλύτερο ποσοστό γίνεται. Η μέχρι τώρα απόκτηση γνώσεων και εμπειρίας θα παίξει σημαντικό ρόλο στην συνέχιση της πορείας στο game development.

Βιβλιογραφία

[1] XNA 2.0 Game Programming Recipes: A Problem-Solution Approach
Copyright © 2008 by Riemer Grootjans

[2] Beginning XNA 2.0 Game Programming: From Novice to Professional
Copyright © 2008 by Alexandre Lobão, Bruno Evangelista, José Antonio Leal de Farias

[3] *Real-Time Collision Detection*
Christer Ericson

[4] Microsoft XNA Game Studio Creator's Guide: An Introduction to XNA Game Programming
Stephen Cawood
Pat McGee

[5] www.enchantedage.com

[6] digierr.spaces.live.com/blog/cns!2B7007E9EC2AE37B!612.entry

[7] www.ziggyware.com/readarticle.php?article_id=155

[8] Digital Texturing & Painting by Owen Dimers Copyright © 2002 by New Riders

[9] Game Character Design Complete David Franson Eric Thomas © 2007 Thomson Course Technology

[10] Digital Lighting & Rendering, Second Edition By Jeremy Birn

[11] 3D Game Textures Luke Ahearn

[12] Blender 3D: Architecture, Buildings, and Scenery by Alan Brito

[13] d'artiste Character modelling © 2007 Ballistic Publishing

[14] www.wikipedia.com

[15] msdn.microsoft.com

[16] www.blenderartists.org

Παράρτημα

- Source code 1ης σκηνής:

- **Program.cs**

```
using System;

namespace DeferredShadingTutorial
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        static void Main(string[] args)
        {
            using (Game1 game = new Game1())
            {
                game.Run();
            }
        }
    }
}
```

- **Game1.cs**

```
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;
using System.Diagnostics;
using System.Text;


namespace DeferredRenderer
{
/// <summary>
/// This is the main type for your game
/// </summary>
public class Game1 : Microsoft.Xna.Framework.Game
{
GraphicsDeviceManager graphics;
SpriteBatch spriteBatch;



public Game1()
{
graphics = new GraphicsDeviceManager(this);
Content.RootDirectory = "Content";
DeferredRenderer renderer = new DeferredRenderer(this);
Components.Add(renderer);
graphics.PreferredDepthStencilFormat = SelectStencilMode();
//System.Diagnostics.Debug.WriteLine(graphics.PreferredDepthStencilFo
rmat);
graphics.PreferredBackBufferHeight = 1024;
graphics.PreferredBackBufferWidth = 1280;
graphics.IsFullScreen = true;

}



private static DepthFormat SelectStencilMode()
{
GraphicsAdapter adapter = GraphicsAdapter.DefaultAdapter;
SurfaceFormat format = adapter.CurrentDisplayMode.Format;
//First check
if (adapter.CheckDepthStencilMatch(DeviceType.Hardware, format,
format, DepthFormat.Depth24Stencil8))
```

```
return DepthFormat.Depth24Stencil8;
//Second check
if (adapter.CheckDepthStencilMatch(DeviceType.Hardware, format,
format, DepthFormat.Depth24Stencil8Single))
return DepthFormat.Depth24Stencil8Single;
//Third check
if (adapter.CheckDepthStencilMatch(DeviceType.Hardware, format,
format, DepthFormat.Depth24Stencil4))
return DepthFormat.Depth24Stencil4;
//Fourth check
if (adapter.CheckDepthStencilMatch(DeviceType.Hardware, format,
format, DepthFormat.Depth15Stencil1))
return DepthFormat.Depth15Stencil1;
else
throw new InvalidOperationException("Pare allh karta grafikwn");

}



/// <summary>
/// Allows the game to perform any initialization it needs to before
starting to run.
/// This is where it can query for any required services and load any
non-graphic
/// related content.  Calling base.Initialize will enumerate through
any components
/// and initialize them as well.
/// </summary>
protected override void Initialize()
{
// TODO: Add your initialization logic here

base.Initialize();


}

/// <summary>
/// LoadContent will be called once per game and is the place to load
/// all of your content.
/// </summary>
protected override void LoadContent()
{
// Create a new SpriteBatch, which can be used to draw textures.
spriteBatch = new SpriteBatch(GraphicsDevice);

// TODO: use this.Content to load your game content here


}





/// <summary>
/// UnloadContent will be called once per game and is the place to
unload
/// all content.
/// </summary>
```

```
protected override void UnloadContent()
{
// TODO: Unload any non ContentManager content here
}

/// <summary>
/// Allows the game to run logic such as updating the world,
/// checking for collisions, gathering input, and playing audio.
/// </summary>
/// <param name="gameTime">Provides a snapshot of timing
values.</param>
protected override void Update(GameTime gameTime)
{
// Allows the game to exit
if (GamePad.GetState(PlayerIndex.One).Buttons.Back ==
ButtonState.Pressed)
this.Exit();

// TODO: Add your update logic here



base.Update(gameTime);
}

/// <summary>
/// This is called when the game should draw itself.
/// </summary>
/// <param name="gameTime">Provides a snapshot of timing
values.</param>
protected override void Draw(GameTime gameTime)
{
graphics.GraphicsDevice.Clear(Color.CornflowerBlue);


// TODO: Add your drawing code here


base.Draw(gameTime);

}



}
}
```

- **XNAUtils.cs**

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Storage;


namespace DeferredRenderer
{

class XNAUtils
{



public static BoundingBox TransformBoundingBox(BoundingBox origBox,
Matrix matrix)
{
Vector3 origCorner1 = origBox.Min;
Vector3 origCorner2 = origBox.Max;

Vector3 transCorner1 = Vector3.Transform(origCorner1, matrix);
Vector3 transCorner2 = Vector3.Transform(origCorner2, matrix);

return new BoundingBox(transCorner1, transCorner2);
}



public static BoundingSphere TransformBoundingSphere(BoundingSphere
originalBoundingSphere, Matrix transformationMatrix)
{
Vector3 trans;
Vector3 scaling;
Quaternion rot;
transformationMatrix.Decompose(out scaling, out rot, out trans);

float maxScale = scaling.X;
if (maxScale < scaling.Y)
maxScale = scaling.Y;
if (maxScale < scaling.Z)
maxScale = scaling.Z;

float transformedSphereRadius = originalBoundingSphere.Radius *
maxScale;
```

```
Vector3 transformedSphereCenter =
Vector3.Transform(originalBoundingSphere.Center,
transformationMatrix);

BoundingSphere transformedBoundingSphere = new
BoundingSphere(transformedSphereCenter, transformedSphereRadius);

return transformedBoundingSphere;
}



public static Model LoadModelWithBoundingSphere(ref Matrix[]
modelTransforms, string asset, ContentManager content)
{
Model newModel = content.Load<Model>(asset);

modelTransforms = new Matrix[newModel.Bones.Count];
newModel.CopyAbsoluteBoneTransformsTo(modelTransforms);

BoundingSphere completeBoundingSphere = new BoundingSphere();

foreach (ModelMesh mesh in newModel.Meshes)
{
BoundingSphere origMeshSphere = mesh.BoundingSphere;
BoundingSphere transMeshSphere =
XNAUtils.TransformBoundingSphere(origMeshSphere,
modelTransforms[mesh.ParentBone.Index]);
completeBoundingSphere =
BoundingSphere.CreateMerged(completeBoundingSphere, transMeshSphere);
}
newModel.Tag = completeBoundingSphere;

return newModel;
}



public static void DrawBoundingBox(BoundingBox bBox, GraphicsDevice
device, BasicEffect basicEffect, Matrix worldMatrix, Matrix
viewMatrix, Matrix projectionMatrix)
{
Vector3 v1 = bBox.Min;
Vector3 v2 = bBox.Max;

VertexPositionColor[] cubeLineVertices = new VertexPositionColor[8];
cubeLineVertices[0] = new VertexPositionColor(v1, Color.White);
cubeLineVertices[1] = new VertexPositionColor(new Vector3(v2.X, v1.Y,
v1.Z), Color.Red);
cubeLineVertices[2] = new VertexPositionColor(new Vector3(v2.X, v1.Y,
v2.Z), Color.Green);
cubeLineVertices[3] = new VertexPositionColor(new Vector3(v1.X, v1.Y,
v2.Z), Color.Blue);

cubeLineVertices[4] = new VertexPositionColor(new Vector3(v1.X, v2.Y,
v1.Z), Color.White);
cubeLineVertices[5] = new VertexPositionColor(new Vector3(v2.X, v2.Y,
v1.Z), Color.Red);
cubeLineVertices[6] = new VertexPositionColor(v2, Color.Green);
cubeLineVertices[7] = new VertexPositionColor(new Vector3(v1.X, v2.Y,
v2.Z), Color.Blue);
```

```
short[] cubeLineIndices = { 0, 1, 1, 2, 2, 3, 3, 0, 4, 5, 5, 6, 6, 7,
7, 4, 0, 4, 1, 5, 2, 6, 3, 7 };

basicEffect.World = worldMatrix;
basicEffect.View = viewMatrix;
basicEffect.Projection = projectionMatrix;
basicEffect.VertexColorEnabled = true;
device.RenderState.FillMode = FillMode.Solid;
basicEffect.Begin();
foreach (EffectPass pass in basicEffect.CurrentTechnique.Passes)
{
pass.Begin();
device.VertexDeclaration = new VertexDeclaration(device,
VertexPositionColor.VertexElements);
device.DrawUserIndexedPrimitives<VertexPositionColor>(PrimitiveType.L
ineList, cubeLineVertices, 0, 8, cubeLineIndices, 0, 12);
pass.End();
}
basicEffect.End();
}



public static void DrawSphereSpikes(BoundingSphere sphere,
GraphicsDevice device, BasicEffect basicEffect, Matrix worldMatrix,
Matrix viewMatrix, Matrix projectionMatrix)
{
Vector3 up = sphere.Center + sphere.Radius * Vector3.Up;
Vector3 down = sphere.Center + sphere.Radius * Vector3.Down;
Vector3 right = sphere.Center + sphere.Radius * Vector3.Right;
Vector3 left = sphere.Center + sphere.Radius * Vector3.Left;
Vector3 forward = sphere.Center + sphere.Radius * Vector3.Forward;
Vector3 back = sphere.Center + sphere.Radius * Vector3.Backward;

VertexPositionColor[] sphereLineVertices = new
VertexPositionColor[6];
sphereLineVertices[0] = new VertexPositionColor(up, Color.White);
sphereLineVertices[1] = new VertexPositionColor(down, Color.White);
sphereLineVertices[2] = new VertexPositionColor(left, Color.White);
sphereLineVertices[3] = new VertexPositionColor(right, Color.White);
sphereLineVertices[4] = new VertexPositionColor(forward,
Color.White);
sphereLineVertices[5] = new VertexPositionColor(back, Color.White);

basicEffect.World = worldMatrix;
basicEffect.View = viewMatrix;
basicEffect.Projection = projectionMatrix;
basicEffect.VertexColorEnabled = true;
basicEffect.Begin();
foreach (EffectPass pass in basicEffect.CurrentTechnique.Passes)
{
pass.Begin();
device.VertexDeclaration = new VertexDeclaration(device,
VertexPositionColor.VertexElements);
device.DrawUserPrimitives<VertexPositionColor>(PrimitiveType.LineList
, sphereLineVertices, 0, 3);
pass.End();
}
basicEffect.End();
```

```
}



public static VertexPositionColor[]
VerticesFromVector3List(List<Vector3> pointList, Color color)
{
VertexPositionColor[] vertices = new
VertexPositionColor[pointList.Count];

int i = 0;
foreach (Vector3 p in pointList)
vertices[i++] = new VertexPositionColor(p, color);

return vertices;
}



public static BoundingBox CreateBoxFromSphere(BoundingSphere sphere)
{
float radius = sphere.Radius;
Vector3 outerPoint = new Vector3(radius, radius, radius);

Vector3 p1 = sphere.Center + outerPoint;
Vector3 p2 = sphere.Center - outerPoint;

return new BoundingBox(p1, p2);
}


}
}
```

- **Scene.cs**

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using SkinnedModel;
using TrianglePipeline;
using System.IO;
using Microsoft.Xna.Framework.Audio;


namespace DeferredRenderer
{
class Scene
{
private Game game;
Model spiti;
Model colModel;
Model kyvara;
Model colModelCam;

Matrix world2, world,world3, view, projection;
Matrix[] transforms3;


Matrix shadow;


SkinningData skinningData;
AnimationPlayer animPlayer;

Texture2D specMap;
Texture2D normalMap2;


Vector3 lightDir1 = new Vector3(1, 1, 1);


public Vector3 avatarPosition = new Vector3(160, 5, 150);
public float avatarYaw;


BoundingSphere transSphere;

Vector3 lastCamPos;
bool coll;
```

```
private Vector3 cameraPosition;

public Vector3 CamPosition
{
get { return cameraPosition; }
}



public Matrix View
{
get
{

return view;
}
}




public Matrix Projection
{
get
{


return projection;
}
}






bool collis;
String str;

Matrix[] originalTransforms;
Matrix nadoume;

Model myModel;
Matrix[] transforms2;

bool isClosed = true;
bool dirtyDoor = false;
TimeSpan doorMovementStart;

Matrix[] originalTransforms2;

Matrix newCanonMat;
Matrix Portacoll;



bool aman;

Matrix worldd;
```

```
Matrix[] transformsation;


AudioEngine audioEngine;
WaveBank waveBank;
SoundBank soundBank;

Cue cue2;

Vector3 stereofPos = new Vector3(0.0082f, 0.14f, 0.79f);


Model sygrouomena;
Matrix[] transforms5;
Matrix world5;
bool sygrouomai;

bool ompros;
bool opisw;
bool ompros2;
bool opisw2;



public Scene(Game game)
{
this.game = game;

}




public void InitializeScene()
{
colModelCam = game.Content.Load<Model>("mesh/cell");

colModel = game.Content.Load<Model>("mesh/spiticall3");
originalTransforms2 = new Matrix[colModel.Bones.Count];
colModel.CopyBoneTransformsTo(originalTransforms2);

spiti = game.Content.Load<Model>("mesh\\spititel");
originalTransforms = new Matrix[spiti.Bones.Count];
spiti.CopyBoneTransformsTo(originalTransforms);

WriteModelStructure(spiti);
// System.Diagnostics.Debugger.Break();

kyvara = game.Content.Load<Model>("mesh/lazaros3");
Matrix[] transforms = new Matrix[kyvara.Bones.Count];
kyvara.CopyAbsoluteBoneTransformsTo(transforms);

skinningData = kyvara.Tag as SkinningData;
animPlayer = new AnimationPlayer(skinningData);
AnimationClip clip = skinningData.AnimationClips["Take 001"];

animPlayer.StartClip(clip);


normalMap2 = game.Content.Load<Texture2D>("maps/xrwmaan8rwposbump");
```

```
specMap = game.Content.Load<Texture2D>("maps/xrwmaan8rwposspecular");


shadow = Matrix.CreateShadow(lightDir1, new Plane(0, 0.5f, 0, 10));



myModel = game.Content.Load<Model>("mesh/sfairesversion03");
transforms2 = new Matrix[myModel.Bones.Count];
myModel.CopyAbsoluteBoneTransformsTo(transforms2);


audioEngine = new AudioEngine("Content/Audio/hxosptyxiakhs.xgs");
waveBank = new WaveBank(audioEngine, "Content/Audio/myWaveBank.xwb");
soundBank = new SoundBank(audioEngine,
"Content/Audio/mySoundBank.xsb");

cue2 = soundBank.GetCue("vrentzos");

sygrouomena = game.Content.Load<Model>("mesh/cell2");
transforms5 = new Matrix[sygrouomena.Bones.Count];
sygrouomena.CopyAbsoluteBoneTransformsTo(transforms5);
}







public void UpdateScene(GameTime gameTime)
{
KeyboardState keyboardState = Keyboard.GetState();

animPlayer.Update(gameTime.ElapsedGameTime, true, Matrix.Identity);

lastCamPos = cameraPosition;
UpdateCameraThirdPerson();
UpdateAvatarPosition();

// Recalculate the world matrix
CalculateWorld();

DiadrasiCheck(gameTime);

ColXarakthraCheck();

UpdateSoundPosition(cue2, stereofPos, CamPosition, View.Forward,
View.Up);

if (keyboardState.IsKeyDown(Keys.Space))
{

if (cue2.IsPrepared)
cue2.Play();
}


audioEngine.Update();
```

```
}




public void DrawScene(GameTime gameTime)
{


Matrix[] bones;
bones = animPlayer.GetSkinTransforms();


DrawModel(spiti, world3);





#region Draw Character
foreach (ModelMesh mesh in kyvara.Meshes)
{
game.GraphicsDevice.RenderState.DepthBufferEnable = true;
game.GraphicsDevice.RenderState.AlphaBlendEnable = false;
game.GraphicsDevice.RenderState.AlphaTestEnable = false;
game.GraphicsDevice.SamplerStates[0].AddressU =
TextureAddressMode.Wrap;
game.GraphicsDevice.SamplerStates[0].AddressV =
TextureAddressMode.Wrap;


foreach (Effect effect in mesh.Effects)
{

effect.CurrentTechnique = effect.Techniques["ShinyBump"];
effect.Parameters["N_Texture"].SetValue(normalMap2);
effect.Parameters["useSpecMap"].SetValue(false);
effect.Parameters["S_Texture"].SetValue(specMap);
effect.Parameters["Bones"].SetValue(bones);
//effect.Parameters["Projection"].SetValue(camera.Projection);
effect.Parameters["View"].SetValue(view);
effect.Parameters["WorldViewProj"].SetValue(world * view *
projection);
}


mesh.Draw();

}
#endregion




#region Draw Character Shadow
//  game.GraphicsDevice.RenderState.FillMode = FillMode.Point;
game.GraphicsDevice.Clear(ClearOptions.Stencil, Color.Black, 0, 0);
game.GraphicsDevice.RenderState.StencilEnable = true;
game.GraphicsDevice.RenderState.ReferenceStencil = 0;
```

```
game.GraphicsDevice.RenderState.StencilFunction =
CompareFunction.Equal;
game.GraphicsDevice.RenderState.StencilPass =
StencilOperation.Increment;

game.GraphicsDevice.RenderState.AlphaBlendEnable = true;
game.GraphicsDevice.RenderState.SourceBlend = Blend.SourceAlpha;
game.GraphicsDevice.RenderState.DestinationBlend =
Blend.InverseSourceAlpha;

Matrix[] shadowBones = new Matrix[bones.Length];
for (int i = 0; i < shadowBones.Length; i++)
{
shadowBones[i] = bones[i] * shadow;
}

foreach (ModelMesh mesh in kyvara.Meshes)
{
foreach (Effect effect in mesh.Effects)
{
effect.CurrentTechnique = effect.Techniques["Shadow"];
effect.Parameters["Bones"].SetValue(shadowBones);
effect.Parameters["WorldViewProj"].SetValue(world * view *
projection);
}
mesh.Draw();
}
// game.GraphicsDevice.RenderState.FillMode = FillMode.Solid;
game.GraphicsDevice.RenderState.StencilEnable = false;
game.GraphicsDevice.RenderState.AlphaBlendEnable = false;
#endregion


}






private void DrawModel(Model model, Matrix world)
{
transforms3 = new Matrix[model.Bones.Count];
model.CopyAbsoluteBoneTransformsTo(transforms3);


game.GraphicsDevice.RenderState.DepthBufferEnable = true;
game.GraphicsDevice.RenderState.CullMode =
CullMode.CullCounterClockwiseFace;
game.GraphicsDevice.RenderState.AlphaBlendEnable = false;


foreach (ModelMesh mesh in model.Meshes)
{
world3 = transforms3[mesh.ParentBone.Index] * Matrix.Identity;

foreach (Effect effect in mesh.Effects)
{
effect.Parameters["World"].SetValue(world3);
effect.Parameters["View"].SetValue(view);
```

```
effect.Parameters["Projection"].SetValue(projection);

}
mesh.Draw();
}
}



public void CalculateWorld()
{

world = Matrix.CreateScale(0.20f, 0.20f, 0.20f) *
Matrix.CreateRotationY(avatarYaw) *
Matrix.CreateTranslation(avatarPosition);
kyvara.Root.Transform = world;

}



void UpdateCameraThirdPerson()
{


Matrix rotationMatrix = Matrix.CreateRotationY(avatarYaw);

// Set the direction the camera points without rotation.
Vector3 thirdPersonReference = new Vector3(0, 50, -50);


// Create a vector pointing the direction the camera is facing.
Vector3 transformedReference =
Vector3.Transform(thirdPersonReference, rotationMatrix);

// Calculate the position the camera is looking from.
cameraPosition = transformedReference + avatarPosition;


#region Collisions ths Cameras

coll = false;


float minimumDistance = 1.5f;
BoundingSphere cameraSphere = new BoundingSphere(cameraPosition,
minimumDistance);


Matrix[] CamModelTransforms = new Matrix[colModelCam.Bones.Count];
colModelCam.CopyAbsoluteBoneTransformsTo(CamModelTransforms);

foreach (ModelMesh mesh in colModelCam.Meshes)
{
BoundingSphere origSphere = mesh.BoundingSphere;
Matrix trans = CamModelTransforms[mesh.ParentBone.Index];
transSphere = XNAUtils.TransformBoundingSphere(origSphere, trans);

if (cameraSphere.Contains(transSphere) != ContainmentType.Disjoint)
```

```
coll = true;

}

if (coll)
{
game.Window.Title = "NAI COL";
cameraPosition = lastCamPos;

}
else
{
game.Window.Title = "OXI COL";
cameraPosition = transformedReference + avatarPosition;

}

#endregion

// Set up the view matrix and projection matrix.
view = Matrix.CreateLookAt(cameraPosition, new
Vector3(avatarPosition.X, avatarPosition.Y + 45, avatarPosition.Z),
new Vector3(0.0f, 1.0f, 0.0f));

float aspectRatio = (float)game.GraphicsDevice.Viewport.Width /
(float)game.GraphicsDevice.Viewport.Height;

// Set field of view of the camera in radians (pi/4 is 45 degrees).
float viewAngle = MathHelper.PiOver4;

// Set distance from the camera of the near and far clipping planes.
float nearClip = 1.0f;
float farClip = 10000.0f;

projection = Matrix.CreatePerspectiveFieldOfView(viewAngle,
aspectRatio, nearClip, farClip);

}

void UpdateAvatarPosition()
{
KeyboardState keyboardState = Keyboard.GetState();
GamePadState currentState = GamePad.GetState(PlayerIndex.One);
KeyboardState oldKeyState = Keyboard.GetState();
```

```
// Set rates in world units per 1/60th second (the default fixed-step
interval).
float rotationSpeed = 1f / 60f;
float forwardSpeed = 50f / 60f;

Vector3 rayDirection = new Vector3(0, 0, 1);
Vector3 rayPos = new Vector3(avatarPosition.X, 40, avatarPosition.Z);

Ray ray = new Ray(rayPos, rayDirection);



//if (ModelRayCollision(colModel, Matrix.Identity, ray))
if (sygrouomai)
{
avatarPosition.Z -= 0;
avatarPosition.X -= 0;

}

if (keyboardState.IsKeyDown(Keys.Left) || (currentState.DPad.Left ==
ButtonState.Pressed))
{
// Rotate left.
avatarYaw += rotationSpeed;
}

if (keyboardState.IsKeyDown(Keys.Right) || (currentState.DPad.Right
== ButtonState.Pressed))
{
// Rotate right.
avatarYaw -= rotationSpeed;
}




if (keyboardState.IsKeyDown(Keys.Up) && (sygrouomai))
{
avatarPosition.Z += 0;
avatarPosition.X += 0;
ompros = true;
ompros2 = true;

if (keyboardState.IsKeyDown(Keys.Up) && (ompros))
{

Matrix forwardMovement = Matrix.CreateRotationY(avatarYaw);
Vector3 v = new Vector3(0, 0, -2);
v = Vector3.Transform(v, forwardMovement);
avatarPosition.Z += v.Z;
avatarPosition.X += v.X;

ompros = false;
}
else
{
if (keyboardState.IsKeyDown(Keys.Down) && (ompros2))
{
```

```
Matrix forwardMovement = Matrix.CreateRotationY(avatarYaw);
Vector3 v = new Vector3(0, 0, -2);
v = Vector3.Transform(v, forwardMovement);
avatarPosition.Z += v.Z;
avatarPosition.X += v.X;

ompros2 = false;
}



}
}


if (keyboardState.IsKeyDown(Keys.Down) && (sygrouomai))
{


avatarPosition.Z += 0;
avatarPosition.X += 0;
opisw = true;
opisw2 = true;

if (keyboardState.IsKeyDown(Keys.Down) && (opisw))
{

Matrix forwardMovement = Matrix.CreateRotationY(avatarYaw);
Vector3 v = new Vector3(0, 0, 2);
v = Vector3.Transform(v, forwardMovement);
avatarPosition.Z += v.Z;
avatarPosition.X += v.X;

opisw = false;
}
else
{
if (keyboardState.IsKeyDown(Keys.Up) && (opisw2))
{

Matrix forwardMovement = Matrix.CreateRotationY(avatarYaw);
Vector3 v = new Vector3(0, 0, 2);
v = Vector3.Transform(v, forwardMovement);
avatarPosition.Z += v.Z;
avatarPosition.X += v.X;

opisw2 = false;
}



}
}
```

```
if (keyboardState.IsKeyDown(Keys.Left) || (currentState.DPad.Left ==
ButtonState.Pressed))
{
// Rotate left.
avatarYaw += rotationSpeed;
}

if (keyboardState.IsKeyDown(Keys.Right) || (currentState.DPad.Right
== ButtonState.Pressed))
{
// Rotate right.
avatarYaw -= rotationSpeed;
}

if (keyboardState.IsKeyDown(Keys.Up) || (currentState.DPad.Up ==
ButtonState.Pressed))
{
Matrix forwardMovement = Matrix.CreateRotationY(avatarYaw);
Vector3 v = new Vector3(0, 0, forwardSpeed - 1 / 2);
v = Vector3.Transform(v, forwardMovement);
avatarPosition.Z += v.Z;
avatarPosition.X += v.X;
}

if (keyboardState.IsKeyDown(Keys.Down) || (currentState.DPad.Down ==
ButtonState.Pressed))
{
Matrix forwardMovement = Matrix.CreateRotationY(avatarYaw);
Vector3 v = new Vector3(0, 0, -forwardSpeed);
v = Vector3.Transform(v, forwardMovement);
avatarPosition.Z += v.Z;
avatarPosition.X += v.X;
}


//}


}





public bool ModelRayCollision(Model model, Matrix modelWorld, Ray
ray)
{
Matrix[] modelTransforms = new Matrix[model.Bones.Count];
model.CopyAbsoluteBoneTransformsTo(modelTransforms);

Matrix[] transforms = new Matrix[kyvara.Bones.Count];
kyvara.CopyAbsoluteBoneTransformsTo(transforms);

bool collision = false;
foreach (ModelMesh mesh in model.Meshes)
{
Matrix absTransform = modelTransforms[mesh.ParentBone.Index] *
modelWorld;
Matrix absTransform2 = transforms[kyvara.Meshes[0].ParentBone.Index]
* Matrix.CreateFromAxisAngle(new Vector3(0, 1, 0), avatarYaw);
Triangle[] meshTriangles = (Triangle[])mesh.Tag;
```

```
//System.Diagnostics.Debugger.Break();
Matrix invMatrix = Matrix.Invert(absTransform);
Matrix invMatrix2 = Matrix.Invert(absTransform2);
Vector3 transRayStartPoint = Vector3.Transform(ray.Position,
invMatrix);
Vector3 origRayEndPoint = ray.Position + ray.Direction;
Vector3 transRayEndPoint = Vector3.Transform(origRayEndPoint,
invMatrix2);
Ray invRay = new Ray(transRayStartPoint, transRayEndPoint -
transRayStartPoint);



foreach (Triangle tri in meshTriangles)
{
Plane trianglePlane = new Plane(tri.P0, tri.P1, tri.P2);

float distanceOnRay = RayPlaneIntersection(invRay, trianglePlane);

Vector3 intersectionPoint = invRay.Position + distanceOnRay *
invRay.Direction;


if (PointInsideTriangle(tri.P0, tri.P1, tri.P2, intersectionPoint))
collision = true;
}

}

return collision;
}



private float RayPlaneIntersection(Ray ray, Plane plane)
{
float rayPointDist = -plane.DotNormal(ray.Position);
float rayPointToPlaneDist = rayPointDist - plane.D;
float directionProjectedLength = Vector3.Dot(plane.Normal,
ray.Direction);
float factor = rayPointToPlaneDist / (directionProjectedLength /
3.7f);
return factor;
}



private bool PointInsideTriangle(Vector3 p0, Vector3 p1, Vector3 p2,
Vector3 point)
{
if (float.IsNaN(point.X)) return false;

Vector3 A0 = point - p0;
Vector3 B0 = p1 - p0;
Vector3 cross0 = Vector3.Cross(A0, B0);

Vector3 A1 = point - p1;
Vector3 B1 = p2 - p1;
```

```
Vector3 cross1 = Vector3.Cross(A1, B1);

Vector3 A2 = point - p2;
Vector3 B2 = p0 - p2;
Vector3 cross2 = Vector3.Cross(A2, B2);

if (CompareSigns(cross0, cross1) && CompareSigns(cross0, cross2))
return true;
else
return false;
}
```

```
private bool CompareSigns(Vector3 first, Vector3 second)
{
if (Vector3.Dot(first, second) > 0)
return true;
else
return false;
}
```

```
private void WriteModelStructure(Model model)
{
StreamWriter writer = new StreamWriter("modelStructure.txt");
writer.WriteLine("Model Bone Information");
writer.WriteLine("----------------------");
ModelBone root = model.Root;
WriteBone(root, 0, writer);
writer.WriteLine();
writer.WriteLine();
writer.WriteLine("Model Mesh Information");
writer.WriteLine("----------------------");
foreach (ModelMesh mesh in model.Meshes)
WriteModelMesh(model.Meshes.IndexOf(mesh), mesh, writer);
writer.Close();
}
```

```
private void WriteBone(ModelBone bone, int level, StreamWriter
writer)
{
for (int l = 0; l < level; l++)
writer.Write("\t");
writer.Write("- Name : ");
if ((bone.Name == "") || (bone.Name == "null"))
writer.WriteLine("null");
else
writer.WriteLine(bone.Name);
for (int l = 0; l < level; l++)
writer.Write("\t");
```

```
writer.WriteLine(" Index: " + bone.Index);
foreach (ModelBone childBone in bone.Children)
WriteBone(childBone, level + 1, writer);
}


private void WriteModelMesh(int ID, ModelMesh mesh, StreamWriter
writer)
{
writer.WriteLine("- ID : " + ID);
writer.WriteLine(" Name: " + mesh.Name);
writer.Write(" Bone: " + mesh.ParentBone.Name);
writer.WriteLine(" (" + mesh.ParentBone.Index + ")");
}




void OpenTheDoor(GameTime gameTime)
{
doorMovementStart = gameTime.TotalGameTime;
//Open the door.

if (aman)
{
newCanonMat = Matrix.CreateTranslation(-26, 0, 30) *
Matrix.CreateRotationY(MathHelper.ToRadians(80.0f)) *
originalTransforms[92];
Portacoll = Matrix.CreateTranslation(-26, 0, 30) *
Matrix.CreateRotationY(MathHelper.ToRadians(80.0f)) *
originalTransforms2[5];
aman = false;
isClosed = false;
}
else
{
newCanonMat = originalTransforms[92];
Portacoll = originalTransforms2[5];
}


spiti.Bones[92].Transform = newCanonMat;

//gia to spiticoll//

colModel.Bones[5].Transform = Portacoll;


//edw paizei o Vrentzos


}


void CloseTheDoor(GameTime gameTime)
{

doorMovementStart = gameTime.TotalGameTime;
//Close the door.

if (aman)
```

```
{
game.Window.Title = "ANTE RE ";
nadoume = Matrix.CreateTranslation(-8, 0, 10) *
Matrix.CreateRotationY(MathHelper.ToRadians(15));
aman = false;
isClosed = true;
}
else
{
nadoume = Matrix.CreateTranslation(-26, 0, 30) *
Matrix.CreateRotationY(MathHelper.ToRadians(80.0f));
}

Matrix newCanonMat = nadoume * originalTransforms[92];
spiti.Bones[92].Transform = newCanonMat;

//gia to spiticoll//
Matrix Portacoll = nadoume * originalTransforms2[5];
colModel.Bones[5].Transform = Portacoll;

}


void OpenTheDoor2(GameTime gameTime)
{
doorMovementStart = gameTime.TotalGameTime;
//Open the door.

if (aman)
{
newCanonMat = Matrix.CreateTranslation(-26, 0, 30) *
Matrix.CreateRotationY(MathHelper.ToRadians(80.0f)) *
originalTransforms[1];
Portacoll = Matrix.CreateTranslation(-26, 0, 30) *
Matrix.CreateRotationY(MathHelper.ToRadians(80.0f)) *
originalTransforms2[21];
aman = false;
isClosed = false;
}
else
{
newCanonMat = originalTransforms[1];
Portacoll = originalTransforms2[21];
}


spiti.Bones[1].Transform = newCanonMat;

//gia to spiticoll//

colModel.Bones[21].Transform = Portacoll;

}


void CloseTheDoor2(GameTime gameTime)
{

doorMovementStart = gameTime.TotalGameTime;
//Close the door.
```

```
if (aman)
{
game.Window.Title = "ANTE RE";
nadoume = Matrix.CreateTranslation(-8, 0, 10) *
Matrix.CreateRotationY(MathHelper.ToRadians(15));
aman = false;
isClosed = true;
}
else
{
nadoume = Matrix.CreateTranslation(-28, 0, 30) *
Matrix.CreateRotationY(MathHelper.ToRadians(95.0f));
}


Matrix newCanonMat = nadoume * originalTransforms[1];
spiti.Bones[1].Transform = newCanonMat;

//gia to spiticoll//
Matrix Portacoll = nadoume * originalTransforms2[21];
colModel.Bones[21].Transform = Portacoll;

}




private void DiadrasiCheck(GameTime gameTime)
{

KeyboardState keyboardState = Keyboard.GetState();
collis = false;

transformsation = new Matrix[colModel.Bones.Count];
colModel.CopyAbsoluteBoneTransformsTo(transformsation);


foreach (ModelMesh mesh in myModel.Meshes)
{
BoundingSphere origSphere = mesh.BoundingSphere;
Matrix trans = transforms2[mesh.ParentBone.Index];
BoundingSphere transSphere =
XNAUtils.TransformBoundingSphere(origSphere, trans);



foreach (ModelMesh mesh1 in kyvara.Meshes)
{


BoundingSphere origShere1 = mesh1.BoundingSphere;
Matrix trans1 = Matrix.CreateScale(0.10f, 0.10f, 0.10f) *
Matrix.CreateRotationY(avatarYaw) *
Matrix.CreateTranslation(avatarPosition);
BoundingSphere transSphere1 =
XNAUtils.TransformBoundingSphere(origShere1, trans1);



if (transSphere1.Contains(transSphere) != ContainmentType.Disjoint)
```

```
{
collis = true;

str = (string)mesh.Name.Clone();
}



//Gia thn porta tou Ypnodwmatiou

BoundingSphere sfairidio = colModel.Meshes[4].BoundingSphere;
Matrix nio = Matrix.CreateScale(0.14f, 0.14f, 0.14f)  *
originalTransforms2[5];
BoundingSphere ka8armasfaira =
XNAUtils.TransformBoundingSphere(sfairidio, nio);

if (transSphere1.Contains(ka8armasfaira) != ContainmentType.Disjoint)
{
aman = true;
game.Window.Title = "SYGROYOMAI REEEE";

}



//Gia thn porta ths kouzinas

BoundingSphere sfairoukla = colModel.Meshes[20].BoundingSphere;
Matrix niosto = Matrix.CreateScale(0.15f, 0.15f, 0.18f) *
originalTransforms2[21];
BoundingSphere SfairaPortasKouz =
XNAUtils.TransformBoundingSphere(sfairoukla, niosto);

if (transSphere1.Contains(SfairaPortasKouz) !=
ContainmentType.Disjoint)
{
aman = true;
game.Window.Title = "SYGROYOMAI REEEE";

}






}



}







if ((collis) && (str == "Sphere01"))
{
game.Window.Title = str;
```

```
if (keyboardState.IsKeyDown(Keys.E) && !dirtyDoor)
{
dirtyDoor = true;
if (isClosed)
{
OpenTheDoor(gameTime);
}
else
{
CloseTheDoor(gameTime);
}
}

if (dirtyDoor)
{
if (gameTime.TotalGameTime - doorMovementStart >
TimeSpan.FromSeconds(0.5))
{
dirtyDoor = false;
isClosed = !isClosed;

}
}



}
else if ((collis) && (str == "Sphere02"))
{

game.Window.Title = str;



if (keyboardState.IsKeyDown(Keys.E) && !dirtyDoor)
{
dirtyDoor = true;
if (isClosed)
{
OpenTheDoor2(gameTime);
}
else
{
CloseTheDoor2(gameTime);
}
}

if (dirtyDoor)
{
if (gameTime.TotalGameTime - doorMovementStart >
TimeSpan.FromSeconds(0.5))
{
dirtyDoor = false;
isClosed = !isClosed;

}
}
```

```
}



else
{
game.Window.Title = "OXI COL";
}

}

private void UpdateSoundPosition(Cue cue, Vector3 sourcePos, Vector3
camPos, Vector3 camForward, Vector3 camUp)
{
AudioEmitter emitter = new AudioEmitter();
emitter.Position = sourcePos;
AudioListener listener = new AudioListener();
listener.Position = camPos;
listener.Forward = camForward;
listener.Up = camUp;
cue.Apply3D(listener, emitter);
}

private void ColXarakthraCheck()
{
sygrouomai = false;

foreach (ModelMesh mesh in sygrouomena.Meshes)
{
BoundingSphere origSphere3 = mesh.BoundingSphere;
Matrix transalp = transforms5[mesh.ParentBone.Index];
BoundingSphere transSphera =
XNAUtils.TransformBoundingSphere(origSphere3, transalp);

foreach (ModelMesh mesh1 in kyvara.Meshes)
{

BoundingSphere origShere5 = mesh1.BoundingSphere;
Matrix trans10 = Matrix.CreateScale(0.03f, 0.03f, 0.03f) *
Matrix.CreateRotationY(avatarYaw) *
Matrix.CreateTranslation(avatarPosition);
BoundingSphere transSphere100 =
XNAUtils.TransformBoundingSphere(origShere5, trans10);

if (transSphere100.Contains(transSphera) != ContainmentType.Disjoint)
{
sygrouomai = true;
}


}

}


}
```

```
}

}
```

- **QuadRenderer.cs**

```
#region Using Statements
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
#endregion



namespace DeferredRenderer
{
public partial class QuadRenderComponent :
Microsoft.Xna.Framework.DrawableGameComponent
{
#region Private Members

VertexDeclaration vertexDecl = null;
VertexPositionTexture[] verts = null;
short[] ib = null;

#endregion


#region Constructor
public QuadRenderComponent(Game game)
: base(game)
{
// TODO: Construct any child components here
}
#endregion


#region LoadGraphicsContent

protected override void LoadContent()
{
IGraphicsDeviceService graphicsService =
(IGraphicsDeviceService)base.Game.Services.GetService(
typeof(IGraphicsDeviceService));

vertexDecl = new VertexDeclaration(
graphicsService.GraphicsDevice,
VertexPositionTexture.VertexElements);

verts = new VertexPositionTexture[]
```

```
{
new VertexPositionTexture(
new Vector3(0,0,0),
new Vector2(1,1)),
new VertexPositionTexture(
new Vector3(0,0,0),
new Vector2(0,1)),
new VertexPositionTexture(
new Vector3(0,0,0),
new Vector2(0,0)),
new VertexPositionTexture(
new Vector3(0,0,0),
new Vector2(1,0))
};

ib = new short[] { 0, 1, 2, 2, 3, 0 };

}
#endregion


#region void Render(Vector2 v1, Vector2 v2)
public void Render(Vector2 v1, Vector2 v2)
{
IGraphicsDeviceService graphicsService = (IGraphicsDeviceService)
base.Game.Services.GetService(typeof(IGraphicsDeviceService));

GraphicsDevice device = graphicsService.GraphicsDevice;
device.VertexDeclaration = vertexDecl;

verts[0].Position.X = v2.X;
verts[0].Position.Y = v1.Y;

verts[1].Position.X = v1.X;
verts[1].Position.Y = v1.Y;

verts[2].Position.X = v1.X;
verts[2].Position.Y = v2.Y;

verts[3].Position.X = v2.X;
verts[3].Position.Y = v2.Y;

device.DrawUserIndexedPrimitives<VertexPositionTexture>
(PrimitiveType.TriangleList, verts, 0, 4, ib, 0, 2);
}
#endregion
}
}
```

- **DeferredRenderer.cs**

```
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Storage;
using Microsoft.Xna.Framework.Content;

namespace DeferredRenderer
{
public class DeferredRenderer :
Microsoft.Xna.Framework.DrawableGameComponent
{

private QuadRenderComponent quadRenderer;
Scene scene;
private RenderTarget2D colorRT;    //this Render Target will hold
color and Specular Intensity
private RenderTarget2D normalRT; //this Render Target will hold
normals and Specular Power
private RenderTarget2D depthRT; //finally, this one will hold the
depth
Effect clearBufferEffect;
private SpriteBatch spriteBatch;
private Vector2 halfPixel;
private Effect directionalLightEffect;
private RenderTarget2D lightRT;
private Effect finalCombineEffect;
private Effect pointLightEffect;
private Model sphereModel;

Texture2D lightmap;


public DeferredRenderer(Game game)
: base(game)
{
scene = new Scene(game);

}
```

```
public override void Initialize()
{


base.Initialize();

quadRenderer = new QuadRenderComponent(Game);
Game.Components.Add(quadRenderer);

}




protected override void LoadContent()
{
halfPixel.X = 0.5f /
(float)GraphicsDevice.PresentationParameters.BackBufferWidth;
halfPixel.Y = 0.5f /
(float)GraphicsDevice.PresentationParameters.BackBufferHeight;

//get the sizes of the backbuffer, in order to have matching render
targets
int backBufferWidth =
GraphicsDevice.PresentationParameters.BackBufferWidth;
int backBufferHeight =
GraphicsDevice.PresentationParameters.BackBufferHeight;

colorRT = new RenderTarget2D(GraphicsDevice, backBufferWidth,
backBufferHeight, 1, SurfaceFormat.Color);
normalRT = new RenderTarget2D(GraphicsDevice, backBufferWidth,
backBufferHeight, 1, SurfaceFormat.Color);
depthRT = new RenderTarget2D(GraphicsDevice, backBufferWidth,
backBufferHeight, 1, SurfaceFormat.Single);
lightRT = new RenderTarget2D(GraphicsDevice, backBufferWidth,
backBufferHeight, 1, SurfaceFormat.Color);
scene.InitializeScene();
clearBufferEffect = Game.Content.Load<Effect>("ClearGBuffer");
directionalLightEffect =
Game.Content.Load<Effect>("DirectionalLight");
finalCombineEffect = Game.Content.Load<Effect>("CombineFinal");
pointLightEffect = Game.Content.Load<Effect>("PointLight");
sphereModel = Game.Content.Load<Model>("mesh\\sphere");



spriteBatch = new SpriteBatch(Game.GraphicsDevice);
base.LoadContent();
}



public override void Draw(GameTime gameTime)
{
SetGBuffer();
ClearGBuffer();

scene.DrawScene(gameTime);
```

```
ResolveGBuffer();

DrawLights(gameTime);

//lightmap.Save("lghtmap.jpg", ImageFileFormat.Jpg);

base.Draw(gameTime);
}




public override void Update(GameTime gameTime)
{

scene.UpdateScene(gameTime);

base.Update(gameTime);
}




private void SetGBuffer()
{
GraphicsDevice.SetRenderTarget(0, colorRT);
GraphicsDevice.SetRenderTarget(1, normalRT);
GraphicsDevice.SetRenderTarget(2, depthRT);
}



private void ResolveGBuffer()
{
//set all rendertargets to null. In XNA 2.0, switching a rendertarget
causes the resolving of the previous rendertarget.
// In XNA 1.0, we needed to call
GraphicsDevice.ResolveRenderTarget(i);
GraphicsDevice.SetRenderTarget(0, null);
GraphicsDevice.SetRenderTarget(1, null);
GraphicsDevice.SetRenderTarget(2, null);
}



private void ClearGBuffer()
{
GraphicsDevice.Clear(Color.Black);
clearBufferEffect.Begin();
clearBufferEffect.Techniques[0].Passes[0].Begin();
quadRenderer.Render(Vector2.One * -1, Vector2.One);
clearBufferEffect.Techniques[0].Passes[0].End();
clearBufferEffect.End();
}
```

```
private void DrawDirectionalLight(Vector3 lightDirection, Color
color)
{
//set all parameters
directionalLightEffect.Parameters["colorMap"].SetValue(colorRT.GetTex
ture());
directionalLightEffect.Parameters["normalMap"].SetValue(normalRT.GetT
exture());
directionalLightEffect.Parameters["depthMap"].SetValue(depthRT.GetTex
ture());
directionalLightEffect.Parameters["lightDirection"].SetValue(lightDir
ection);
directionalLightEffect.Parameters["Color"].SetValue(color.ToVector3()
);
directionalLightEffect.Parameters["cameraPosition"].SetValue(scene.Ca
mPosition);
directionalLightEffect.Parameters["InvertViewProjection"].SetValue(Ma
trix.Invert(scene.View * scene.Projection));
directionalLightEffect.Parameters["halfPixel"].SetValue(halfPixel);

directionalLightEffect.Begin();
directionalLightEffect.Techniques[0].Passes[0].Begin();
//draw a full-screen quad
quadRenderer.Render(Vector2.One * -1, Vector2.One);
directionalLightEffect.Techniques[0].Passes[0].End();
directionalLightEffect.End();
}
```

```
private void DrawLights(GameTime gameTime)
{

GraphicsDevice.SetRenderTarget(0, lightRT);

//clear all components to 0
GraphicsDevice.Clear(Color.TransparentBlack);
GraphicsDevice.RenderState.AlphaBlendEnable = true;
//use additive blending, and make sure the blending factors are as we
need them
GraphicsDevice.RenderState.AlphaBlendOperation = BlendFunction.Add;
GraphicsDevice.RenderState.SourceBlend = Blend.One;
GraphicsDevice.RenderState.DestinationBlend = Blend.One;
//use the same operation on the alpha channel
GraphicsDevice.RenderState.SeparateAlphaBlendEnabled = false;
GraphicsDevice.RenderState.DepthBufferEnable = false;


float angle = (float)gameTime.TotalGameTime.TotalSeconds;



DrawDirectionalLight(new Vector3(0, 100, 100), Color.Honeydew);
DrawDirectionalLight(new Vector3(0, -100, -100), Color.Honeydew);

DrawDirectionalLight(new Vector3(100, 50, 0), Color.Honeydew);
```

```
DrawDirectionalLight(new Vector3(-100, 50, 0), Color.Honeydew);


DrawPointLight(new Vector3(70, 40, 100), Color.Honeydew, 100, 1.5f);

GraphicsDevice.RenderState.AlphaBlendEnable = false;
GraphicsDevice.SetRenderTarget(0, null);

lightmap = lightRT.GetTexture();

//set the effect parameters
finalCombineEffect.Parameters["colorMap"].SetValue(colorRT.GetTexture
());
finalCombineEffect.Parameters["lightMap"].SetValue(lightmap);
finalCombineEffect.Parameters["halfPixel"].SetValue(halfPixel);

finalCombineEffect.Begin();
finalCombineEffect.Techniques[0].Passes[0].Begin();

//render a full-screen quad
quadRenderer.Render(Vector2.One * -1, Vector2.One);

finalCombineEffect.Techniques[0].Passes[0].End();
finalCombineEffect.End();
}




private void DrawPointLight(Vector3 lightPosition, Color color, float
lightRadius, float lightIntensity)
{
// GraphicsDevice.RenderState.CullMode = CullMode.None;


//set the G-Buffer parameters
pointLightEffect.Parameters["colorMap"].SetValue(colorRT.GetTexture()
);
pointLightEffect.Parameters["normalMap"].SetValue(normalRT.GetTexture
());
pointLightEffect.Parameters["depthMap"].SetValue(depthRT.GetTexture()
);

//compute the light world matrix
//scale according to light radius, and translate it to light position
Matrix sphereWorldMatrix = Matrix.CreateScale(lightRadius) *
Matrix.CreateTranslation(lightPosition);
pointLightEffect.Parameters["World"].SetValue(sphereWorldMatrix);
pointLightEffect.Parameters["View"].SetValue(scene.View);
pointLightEffect.Parameters["Projection"].SetValue(scene.Projection);
//light position
pointLightEffect.Parameters["lightPosition"].SetValue(lightPosition);

//set the color, radius and Intensity
pointLightEffect.Parameters["Color"].SetValue(color.ToVector3());
pointLightEffect.Parameters["lightRadius"].SetValue(lightRadius);
pointLightEffect.Parameters["lightIntensity"].SetValue(lightIntensity
);

//parameters for specular computations
```

```
pointLightEffect.Parameters["cameraPosition"].SetValue(scene.CamPosit
ion);
pointLightEffect.Parameters["InvertViewProjection"].SetValue(Matrix.I
nvert(scene.View * scene.Projection));
//size of a halfpixel, for texture coordinates alignment
pointLightEffect.Parameters["halfPixel"].SetValue(halfPixel);



GraphicsDevice.RenderState.CullMode = CullMode.CullClockwiseFace;



pointLightEffect.Begin();
pointLightEffect.Techniques[0].Passes[0].Begin();

foreach (ModelMesh mesh in sphereModel.Meshes)
{
foreach (ModelMeshPart meshPart in mesh.MeshParts)
{
GraphicsDevice.VertexDeclaration = meshPart.VertexDeclaration;
GraphicsDevice.Vertices[0].SetSource(mesh.VertexBuffer,
meshPart.StreamOffset, meshPart.VertexStride);
GraphicsDevice.Indices = mesh.IndexBuffer;
GraphicsDevice.DrawIndexedPrimitives(PrimitiveType.TriangleList,
meshPart.BaseVertex, 0, meshPart.NumVertices, meshPart.StartIndex,
meshPart.PrimitiveCount);
}
}

pointLightEffect.Techniques[0].Passes[0].End();
pointLightEffect.End();

GraphicsDevice.RenderState.CullMode =
CullMode.CullCounterClockwiseFace;
GraphicsDevice.RenderState.DepthBufferWriteEnable = true;
}





}
}
```

- Source code 2^{ης} σκηνής :

- **Program.cs**

```
using System;

namespace PrwthProspa8eiaKilo
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        static void Main(string[] args)
        {
            using (Kilo game = new Kilo())
            {
                game.Run();
            }
        }
    }
}
```

- **Kilo.cs**

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Media;
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;

using KiloWatt.Base.Graphics;
using KiloWatt.Base.Animation;




namespace PrwthProspa8eiaKilo
{
/// <summary>
/// This is the main type for your game
/// </summary>
public class Kilo : Microsoft.Xna.Framework.Game
{
GraphicsDeviceManager graphics;
SpriteBatch spriteBatch;


public static Kilo Global;


Matrix world1, viewInv;
DrawDetails drawDetails_ = new DrawDetails();


Matrix[] transforms;


KeyboardState prevKeyboard_;
KeyboardState curKeyboard_;


#region Metavlhtes Cameras
```

```
Matrix view, projection;

Vector3 camposition;
Vector3 camOriginalTarget;
Vector3 camOriginalUpvector;


Matrix cameraRotation;

float leftrightRot;
float updownRot;

MouseState originalMouseState;

float avatarYaw;
Vector3 avatarPosition;

#endregion




Model mlkia;
Matrix world2;
Matrix[] transalp;


int animfrom;
float transduration;



public Kilo()
{
Global = this;
graphics = new GraphicsDeviceManager(this);

graphics.PreferredBackBufferWidth = 1024;
graphics.PreferredBackBufferHeight = 576;

graphics.MinimumPixelShaderProfile = ShaderProfile.PS_2_0;
graphics.MinimumVertexShaderProfile = ShaderProfile.VS_2_0;
graphics.SynchronizeWithVerticalRetrace = true;
Content.RootDirectory = StorageContainer.TitleLocation + "/Content";


}

/// <summary>
/// Allows the game to perform any initialization it needs to before
starting to run.
/// This is where it can query for any required services and load any
non-graphic
/// related content.  Calling base.Initialize will enumerate through
any components
/// and initialize them as well.
/// </summary>
protected override void Initialize()
{
```

```
#region Ry8miseis Cameras

float viewAngle = MathHelper.PiOver4;
float aspectratio = (float)GraphicsDevice.Viewport.Width /
(float)GraphicsDevice.Viewport.Height;
float nearPlane = 1.0f;
float farPlane = 1000.0f;



projection = Matrix.CreatePerspectiveFieldOfView(viewAngle,
aspectratio, nearPlane, farPlane);



leftrightRot = 0.0f;
updownRot = 0.0f;

UpdateViewMatrix();

Mouse.SetPosition(Window.ClientBounds.Width / 2,
Window.ClientBounds.Height / 2);

originalMouseState = Mouse.GetState();


#endregion

base.Initialize();
blender_.TransitionAnimations(GetBlended(-1), GetBlended(0), 1.0f);
}

/// <summary>
/// LoadContent will be called once per game and is the place to load
/// all of your content.
/// </summary>
protected override void LoadContent()
{
// Create a new SpriteBatch, which can be used to draw textures.
spriteBatch = new SpriteBatch(GraphicsDevice);

SetPath("");

LoadModel(System.IO.Path.Combine(CurPath, "sperun.xnb"));

mlkia = Content.Load<Model>("mlkia//kouta");
transalp = new Matrix[mlkia.Bones.Count];
mlkia.CopyAbsoluteBoneTransformsTo(transalp);

}

/// <summary>
/// UnloadContent will be called once per game and is the place to
unload
/// all content.
/// </summary>
protected override void UnloadContent()
{
// TODO: Unload any non ContentManager content here
}
/// <summary>
```

```
/// Allows the game to run logic such as updating the world,
/// checking for collisions, gathering input, and playing audio.
/// </summary>
/// <param name="gameTime">Provides a snapshot of timing
values.</param>
///
public bool KeyReleased(Keys k)
{
return !prevKeyboard_.IsKeyUp(k) && curKeyboard_.IsKeyUp(k);
}



public bool KeyPressed(Keys k)
{
return !prevKeyboard_.IsKeyDown(k) && curKeyboard_.IsKeyDown(k);
}



IBlendedAnimation GetBlended(int ix)
{
unchecked
{
return (ix < 0) ? null : (ix >= blended_.Length) ? null :
blended_[ix];
}
}





protected override void Update(GameTime gameTime)
{
// Allows the game to exit
if (GamePad.GetState(PlayerIndex.One).Buttons.Back ==
ButtonState.Pressed)
this.Exit();


float dt = (float)gameTime.ElapsedGameTime.TotalSeconds;
if (dt > 0.1f) dt = 0.1f;





prevKeyboard_ = curKeyboard_;
curKeyboard_ = Keyboard.GetState(PlayerIndex.One);



#region Update Cameras


UserInput();


#endregion
```

```
if (blender_ != null)
blender_.Advance(dt);



base.Update(gameTime);
}





private void CalculateWorld()
{
world1 =  Matrix.CreateRotationY(avatarYaw) *
Matrix.CreateTranslation(avatarPosition);
loadedModel_.Model.Root.Transform = world1;

}





/// <summary>
/// This is called when the game should draw itself.
/// </summary>
/// <param name="gameTime">Provides a snapshot of timing
values.</param>
protected override void Draw(GameTime gameTime)
{
graphics.GraphicsDevice.Clear(Color.CornflowerBlue);



viewInv = Matrix.Invert(view);



//  if I have a model, set up the scene drawing parameters and draw
the model
if (loadedModel_ != null)
{
//  the drawdetails set-up can be re-used for all items in the scene
DrawDetails dd = drawDetails_;
dd.dev = GraphicsDevice;
dd.fogColor = new Vector4(0.5f, 0.5f, 0.5f, 1);
dd.fogDistance = 100000;
dd.lightAmbient = new Vector4(0.2f, 0.2f, 0.2f, 1.0f);
dd.lightDiffuse = new Vector4(0.3f, 0.3f, 0.3f, 0);

//dd.lightDir = Vector3.Normalize(new Vector3(100, 300, 200));

//dd.lightDir = new Vector4(0.0f, 300.0f, -400.0f, 1.0f);
//dd.lightDir = new Vector3(0.0f, 300.0f, -400.0f);
```

```
dd.viewInv = viewInv;
dd.viewProj = view * projection;
dd.world = Matrix.Identity;



loadedModel_.Transform = world1;

//  draw the loaded model (the only model I have)
loadedModel_.SceneDraw(dd);
loadedModel_.SceneDrawTransparent(dd);
}
else
{
System.Diagnostics.Debugger.Break();
}
//  when everything else is drawn, Z sort and draw the transparent
parts



world2 = Matrix.CreateRotationY(MathHelper.ToRadians(45.0f)) *
Matrix.CreateTranslation(20.0f, 0.0f, -100.0f);


foreach (ModelMesh mesh2 in mlkia.Meshes)
{

foreach (BasicEffect bff in mesh2.Effects)
{


bff.World = transalp[mesh2.ParentBone.Index] * world2;
bff.View = view;
bff.Projection = projection;
bff.EnableDefaultLighting();


}
mesh2.Draw();
}



base.Draw(gameTime);
}




#region Boh8htikes Me8odoi Cameras
```

```
private void UserInput()
{

// Set rates in world units per 1/60th second (the default fixed-step
interval).
float characterrotationSpeed = 1f / 60f;
float characterforwardSpeed = 8f / 60f;



float rotationSpeed = 0.001f;

MouseState currentMouseState = Mouse.GetState();

if (currentMouseState != originalMouseState)
{
float xDifference = currentMouseState.X - originalMouseState.X;
float yDifference = currentMouseState.Y - originalMouseState.Y;

leftrightRot -= rotationSpeed * xDifference;
updownRot -= rotationSpeed * yDifference;

Mouse.SetPosition(Window.ClientBounds.Width / 2,
Window.ClientBounds.Height / 2);
}



KeyboardState keyState = Keyboard.GetState();


if (keyState.IsKeyDown(Keys.D))
{
// Rotate right.
avatarYaw -= characterrotationSpeed;

}




if (keyState.IsKeyDown(Keys.A))
{
// Rotate left.
avatarYaw += characterrotationSpeed;


}
```

```
if (KeyPressed(Keys.W)){

curAnimationInstance_ = 1;
animfrom = curAnimationInstance_ - 1;
transduration = 1.0f;

if (curAnimationInstance_ > 0)
{
instances_[curAnimationInstance_].Time =
instances_[curAnimationInstance_ - 1].Time;
}



blender_.TransitionAnimations(GetBlended(animfrom),
GetBlended(curAnimationInstance_), transduration);


}







if (keyState.IsKeyDown(Keys.W))
{


Matrix forwardMovement = Matrix.CreateRotationY(avatarYaw);
Vector3 v = new Vector3(0, 0, characterforwardSpeed - 1 / 2);
v = Vector3.Transform(v, forwardMovement);
avatarPosition.Z += v.Z;
avatarPosition.X += v.X;




if (KeyPressed(Keys.LeftShift))
{



curAnimationInstance_ = 2;
animfrom = curAnimationInstance_ - 1;
transduration = 1.0f;

if (curAnimationInstance_ > 0)
{
instances_[curAnimationInstance_].Time =
instances_[curAnimationInstance_ - 1].Time;
}



blender_.TransitionAnimations(GetBlended(animfrom),
GetBlended(curAnimationInstance_), transduration);
```

```
}


if (keyState.IsKeyDown(Keys.LeftShift)){


Matrix forwardMovement2 = Matrix.CreateRotationY(avatarYaw);
Vector3 v2 = new Vector3(0, 0, 1.01f * characterforwardSpeed - 1 /
2);
v2 = Vector3.Transform(v2, forwardMovement2);
avatarPosition.Z += v2.Z;
avatarPosition.X += v2.X;



}






if (KeyReleased(Keys.LeftShift))
{


curAnimationInstance_ = 1;
animfrom = curAnimationInstance_ + 1;
transduration = 1.0f;

if (curAnimationInstance_ > 0)
{
instances_[curAnimationInstance_].Time =
instances_[curAnimationInstance_ - 1].Time;
}



blender_.TransitionAnimations(GetBlended(animfrom),
GetBlended(curAnimationInstance_), transduration);

}



}








if (KeyReleased(Keys.W)) {


curAnimationInstance_ = 0;
animfrom = curAnimationInstance_ + 1;
```

```
transduration = 1.0f;

if (curAnimationInstance_ > 0)
{
instances_[curAnimationInstance_].Time =
instances_[curAnimationInstance_ - 1].Time;
}

blender_.TransitionAnimations(GetBlended(animfrom),
GetBlended(curAnimationInstance_), transduration);
}
```

```
if (KeyPressed(Keys.S))
{

curAnimationInstance_ = 1;
animfrom = curAnimationInstance_ - 1;
transduration = 1.0f;

if (curAnimationInstance_ > 0)
{
instances_[curAnimationInstance_].Time =
instances_[curAnimationInstance_ - 1].Time;
}

instances_[curAnimationInstance_].Speed = -1.0f;

blender_.TransitionAnimations(GetBlended(animfrom),
GetBlended(curAnimationInstance_), transduration);


}
```

```
if (keyState.IsKeyDown(Keys.S))
{


Matrix forwardMovement = Matrix.CreateRotationY(avatarYaw);
Vector3 v = new Vector3(0, 0, characterforwardSpeed);
v = Vector3.Transform(v, -forwardMovement);
avatarPosition.Z += v.Z;
avatarPosition.X += v.X;

}
```

```
if (KeyReleased(Keys.S))
{


curAnimationInstance_ = 0;
animfrom = curAnimationInstance_ + 1;
transduration = 1.0f;

if (curAnimationInstance_ > 0)
{
instances_[curAnimationInstance_].Time =
instances_[curAnimationInstance_ - 1].Time;
}

blender_.TransitionAnimations(GetBlended(animfrom),
GetBlended(curAnimationInstance_), transduration);
}





CalculateWorld();
UpdateViewMatrix();


}



private void UpdateViewMatrix()
{



cameraRotation = Matrix.CreateRotationX(updownRot) *
Matrix.CreateRotationY(leftrightRot);



camOriginalTarget = new Vector3(0, 0, -1);
camOriginalUpvector = new Vector3(0, 1, 0);


camposition = new Vector3(0, 5, 10);

Vector3 cameraRotatedPosition = Vector3.Transform(camposition,
cameraRotation);
Vector3 cameraFinalPosition = cameraRotatedPosition + avatarPosition;



Vector3 cameraRotatedTarget = Vector3.Transform(camOriginalTarget,
cameraRotation);
Vector3 cameraFinalTarget = cameraFinalPosition +
cameraRotatedTarget;
```

```
Vector3 cameraRotatedUpVector =
Vector3.Transform(camOriginalUpvector, cameraRotation);

view = Matrix.CreateLookAt(cameraFinalPosition, cameraFinalTarget,
cameraRotatedUpVector);

}

#endregion

//BOH8HTIKES ME8ODOI KAI METAVLHTES TOUS//

ModelDraw loadedModel_;                 //  loaded geometry
AnimationBlender blender_;              //  object that blends between
playing animations
int curAnimationInstance_;              //  which animation is playing?
(-1 for none)
AnimationInstance[] instances_;         //  the animation data, as loaded
IBlendedAnimation[] blended_;           //  state about the different
animations (that can change)
AnimationSet animations_;               //  the animations I have to
choose from

internal class CaseInsensitiveComparer : System.Collections.IComparer
{
public int Compare(object x, object y)
{
return String.Compare((string)x, (string)y, true);
}
}

void SetPath(string path)
```

```
{
curPath_ = path;
entries_.Clear();
string[] dirs =
System.IO.Directory.GetDirectories(System.IO.Path.Combine(Content.Roo
tDirectory, path));
string[] files =
System.IO.Directory.GetFiles(System.IO.Path.Combine(Content.RootDirec
tory, path));
System.Collections.IComparer cic = new CaseInsensitiveComparer();
Array.Sort(dirs, cic);
Array.Sort(files, cic);
foreach (string dir in dirs)
if (dir[0] != '.' && dir.IndexOf('%') < 0)
entries_.Add(new FileEntry(dir, true));
foreach (string file in files)
if (file[0] != '.' && file.IndexOf('%') < 0)
if (IsInstanceOf<Model>(file))
entries_.Add(new FileEntry(file, false));
curEntry_ = 0;
}



void LoadModel(string path)
{
try
{
//  the file name that comes from the browser contains .xnb
if (path.EndsWith(".xnb"))
path = path.Substring(0, path.Length - 4);

//  load the model from disk, and prepare it for animation
loadedModel_ = new ModelDraw(Content.Load<Model>(path),
System.IO.Path.GetFileName(path));

//  create a blender that can compose the animations for transition
blender_ = new AnimationBlender(loadedModel_.Model,
loadedModel_.Name);
loadedModel_.AnimationInstance = blender_;

//  remember things about this model
animations_ = null;


transforms = new Matrix[loadedModel_.Model.Bones.Count];
loadedModel_.Model.CopyAbsoluteBoneTransformsTo(transforms);


//  figure out what the animations are, if any.
LoadAnimations();
}
catch (System.Exception x)
{
// couldn't load model
Message = x.Message;
System.Diagnostics.Debug.WriteLine(Message);
}
}
```

```
internal struct FileEntry
{
public FileEntry(string name, bool isDir)
{
int i = name.LastIndexOf(System.IO.Path.DirectorySeparatorChar);
if (i <= 0)
Name = name;
else
Name = name.Substring(i + 1);
IsDir = isDir;
}
public string Name;
public bool IsDir;
}
```

```
public static bool IsInstanceOf<T>(string name) where T : class
{
ContentManager mgr = new ContentManager(Kilo.Global.Services);
try
{
int ind = name.LastIndexOf(System.IO.Path.DirectorySeparatorChar);
mgr.RootDirectory = name.Substring(0, ind);
String toLoad = name.Substring(ind + 1, name.Length - (ind + 5));
object o = mgr.Load<object>(toLoad);
Console.WriteLine("{0} is a {1}", name, o.GetType().Name);
if (typeof(T).IsAssignableFrom(o.GetType()))
return true;
}
catch (System.Exception x)
{
Console.WriteLine("{0} threw {1}", name, x.Message);
}
finally
{
mgr.Dispose();
}
return false;
}
```

```
void LoadAnimations()
{
//  clear current state
curAnimationInstance_ = 0;
instances_ = null;
blended_ = null;

animfrom = curAnimationInstance_ - 1;
transduration = 1.0f;

//  get the list of animations from our dictionary
Dictionary<string, object> tag = loadedModel_.Model.Tag as
Dictionary<string, object>;
object aobj = null;
```

```
if (tag != null)
tag.TryGetValue("AnimationSet", out aobj);
animations_ = aobj as AnimationSet;

//  set up animations
if (animations_ != null)
{
instances_ = new AnimationInstance[animations_.NumAnimations];
//  I'll need a BlendedAnimation per animation, so that I can let the
//  blender object transition between them.
blended_ = new IBlendedAnimation[instances_.Length];
int ix = 0;
foreach (Animation a in animations_.Animations)
{
instances_[ix] = new AnimationInstance(a);
blended_[ix] =
AnimationBlender.CreateBlendedAnimation(instances_[ix]);
++ix;
}
}
}




string curPath_ = "";
public string CurPath { get { return curPath_; } set { curPath_ =
value; } }



List<FileEntry> entries_ = new List<FileEntry>();
int curEntry_;
public int CurEntry { get { return curEntry_; } set { curEntry_ =
value; } }




public string CurEntryName { get { return entries_[curEntry_].Name; }
}





//  the message displayed at the bottom
string message_ = "Ante kai tou xronou";
public string Message { get { return message_; } set { message_ =
value; } }




}
}
```