

**«Σχεδίαση τρισδιάστατου τοπίου με τη  
μηχανή γραφικών Microsoft XNA»**



**Πτυχιακή εργασία**

Όνομ/μο Εισηγήτριας/Επιβλέπουσας καθηγήτριας : Γιαννούση Ηλιάνα

Όνομ/μο Φοιτητή : Γεωργουδάκης Ιωάννης



## Πίνακας Περιεχομένων

1- Περίληψη πτυχιακής εργασίας	Σελ.5
2- Ιστορική αναδρομή	Σελ.9
3- Εισαγωγή στον προγραμματισμό τρισδιάστατων γραφικών	Σελ.15
4- Παρουσίαση του Microsoft Visual Studio 2010	Σελ.23
4.1- Δημιουργία προγράμματος κονσόλας	Σελ.25
4.2- Δημιουργία οπτικής εφαρμογής	Σελ.29
5- Παρουσίαση της μηχανής γραφικών Microsoft XNA	Σελ.35
5.1- Δημιουργία νέου παιχνιδιού	Σελ.36
5.2- Main Functions	Σελ.37
5.3- Content Manager	Σελ.40
5.4- Βασικά στοιχεία δυσδιάστατων γραφικών	Σελ.44
5.5- Εισαγωγή 2D Texture	Σελ.49
5.6- Βασικά στοιχεία τρισδιάστατων γραφικών	Σελ.58
5.7- User Input	Σελ.63
5.8- Pixel/Vertex Shader	Σελ.69
6- Διασύνδεση φόρμας με την μηχανή γραφικών	Σελ.77
7- Παρουσίαση κλάσεων	Σελ.87
8- Πληροφορίες επί της οθόνης	Σελ.95
9- Αναπαραγωγή αρχείων ήχου	Σελ.103
10- Η κάμερα του παιχνιδιού	Σελ.107
11- Ο ουρανός	Σελ.119
12- Ο ήλιος	Σελ.125
13- Μερικά μαθηματικά	Σελ.135
14- Το τετραδικό δέντρο	Σελ.141
15- Το τοπίο	Σελ.153
16- Το νερό	Σελ.163
16.1- Δημιουργία Refraction Map	Σελ.165
16.2- Δημιουργία Reflection Map	Σελ.167
17- Αποθήκευση και ανάκτηση από αρχείο	Σελ.173
18- Εκδόσεις πτυχιακής/Εικόνες πτυχιακής	Σελ.177
19- Επίλογος	Σελ.185
Βιβλιογραφία	Σελ.189



The logo for XNA (Xbox Game Studio) features the lowercase letters 'xna' in a grey, sans-serif font. The letter 'x' is stylized with an orange and red gradient, resembling a pencil or a stylized 'x' shape.

Η πτυχιακή εργασία αυτή προσφέρει την δυνατότητα στον χρήστη να δημιουργήσει ένα τρισδιάστατο τοπίο - τυχαίο ή από το μηδέν -, να το επεξεργαστεί, να προσθέσει θόρυβο και τελικά να το αποθηκεύσει. Του δίνει επίσης την δυνατότητα να προσθέσει ήλιο, ουρανό και θάλασσα. Με άλλα λόγια, μπορούμε να δημιουργήσουμε ένα δικό μας κόσμο. Επίσης, μπορούμε να «παίξουμε» με αυτόν τον κόσμο μεταβάλλοντας την θέση του ήλιου και βλέποντας τις σκιές πάνω στο τοπίο να αλλάζουν αλλά και να επεξεργαστούμε την θέση του νερού και να δούμε πάνω σε αυτό τις αντανάκλασεις του ήλιου και του υπόλοιπου τοπίου.

Το λογισμικό που αναπτύχθηκε στα πλαίσια αυτής της εργασίας μπορεί να χρησιμοποιηθεί μετέπειτα ως βάση για την δημιουργία κάποιου παιχνιδιού. Για όσους γνωρίζουν έστω και λίγο ή έχουν ασχοληθεί με παιχνίδια θα γνωρίζουν το γεγονός ότι το τοπίο για ένα παιχνίδι είναι ένα πολύ βασικό συστατικό και δίνεται μεγάλη έμφαση και προσοχή στη δημιουργία του. Η εργασία αυτή περιέχει βασικές έννοιες και αλγόριθμους για την δημιουργία και επεξεργασία τοπίου, οι οποίες, μέχρι και την στιγμή που διαβάζετε το κείμενο αυτό, χρησιμοποιούνται στα σύγχρονα παιχνίδια.

Η δυσκολία αυτής της πτυχιακής εργασίας, έγκειται στην δυναμικότητα την οποία προσφέρει. Στον «πραγματικό κόσμο» δημιουργίας παιχνιδιών χρησιμοποιούνται έτοιμα προγράμματα σχεδίασης μοντέλων. Στη συνέχεια, αυτά τα μοντέλα εξάγονται σε μορφή συμβατή με την εκάστοτε μηχανή γραφικών και εισάγονται εύκολα με λίγο πρόσθετο προγραμματιστικό κώδικα. Σε αυτήν την εργασία κάτι τέτοιο δεν θα μπορούσε να γίνει, διότι θα θέλαμε να δημιουργήσουμε το τοπίο με τα **βασικά δομικά συστατικά** της μηχανής γραφικών, που είναι τα τρίγωνα, και μετέπειτα να μπορούμε να τα επεξεργαστούμε.

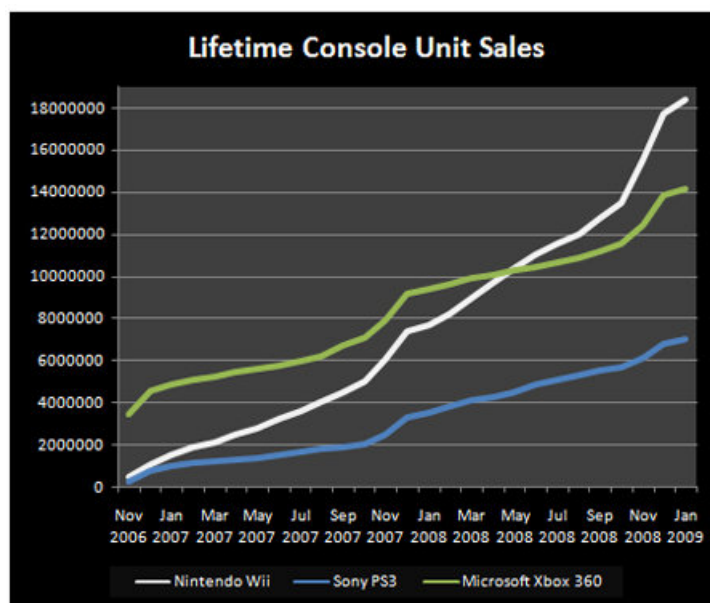


xna





Από το 1971, όταν πρωτοεμφανίστηκε το πρώτο παιχνίδι από την εταιρία **Atari** μέχρι σήμερα, τα πράγματα έχουν αλλάξει πολύ στον τομέα της ανάπτυξης ηλεκτρονικών παιχνιδιών. Η έκρηξη ενδιαφέροντος από το κοινό για τις κονσόλες παιχνιδιών και για τα ίδια τα παιχνίδια είναι εκπληκτική. Πέρα από άλλες μορφές ψυχαγωγίας όπως είναι ο κινηματογράφος και άλλα, η στροφή ενδιαφέροντος του κόσμου για ψυχαγωγία στο σπίτι παρουσιάζει μεγάλο ενδιαφέρον. Ενδεικτικά παρακάτω παρατίθεται ένα διάγραμμα που δείχνει την αύξηση στις πωλήσεις κονσόλων παιχνιδιών.



**Εικόνα 1.** Αύξηση στις πωλήσεις κονσόλων, από τον Νοέμβριο του 2006 έως τον Ιανουάριο του 2009.

Πηγή: [www.tgdaily.com](http://www.tgdaily.com).

Τα παιχνίδια τα οποία συγκαταλέγονται στην κατηγορία των προγραμμάτων, κατέχουν πλέον ένα σημαντικό κομμάτι της πίτας στην ψηφιακή αγορά. Αρχικά βλέπουμε να εμφανίζονται παιχνίδια ενός παίκτη, χωρίς μουσική και ιδιαίτερο σενάριο. Σήμερα έχουμε φτάσει στην εποχή όπου τα παιχνίδια διαθέτουν κινηματογραφικά εφέ, “χολιγουντιανά” σενάρια, και μπορούν να συμμετέχουν σε αυτά παίκτες από όλο τον κόσμο. Στο τελευταίο, βέβαια, σημαντικό ρόλο έπαιξε και η ραγδαία ανάπτυξη του διαδικτύου τα τελευταία χρόνια. Πώς όμως φτάσαμε μέχρι εδώ;

Πηγαίνοντας πίσω στο παρελθόν, η εταιρία ανάπτυξης παιχνιδιών, η γνωστή σε όλους μας **Atari**, ξεκίνησε το 1972 την μαζική παραγωγή του παιχνιδιού **Pong**, το οποίο είχε τη μορφή κερματοδέκτη. Συνολικά πούλησε 38.000 τέτοιους κερματοδέκτες και το παιχνίδι ήταν ολοκληρωτικά σχεδιασμένο με κυκλώματα TTL, χωρίς καν να χρησιμοποιεί επεξεργαστή, η κάποιο κώδικα (πρόγραμμα). Ο κώδικας του παιχνιδιού

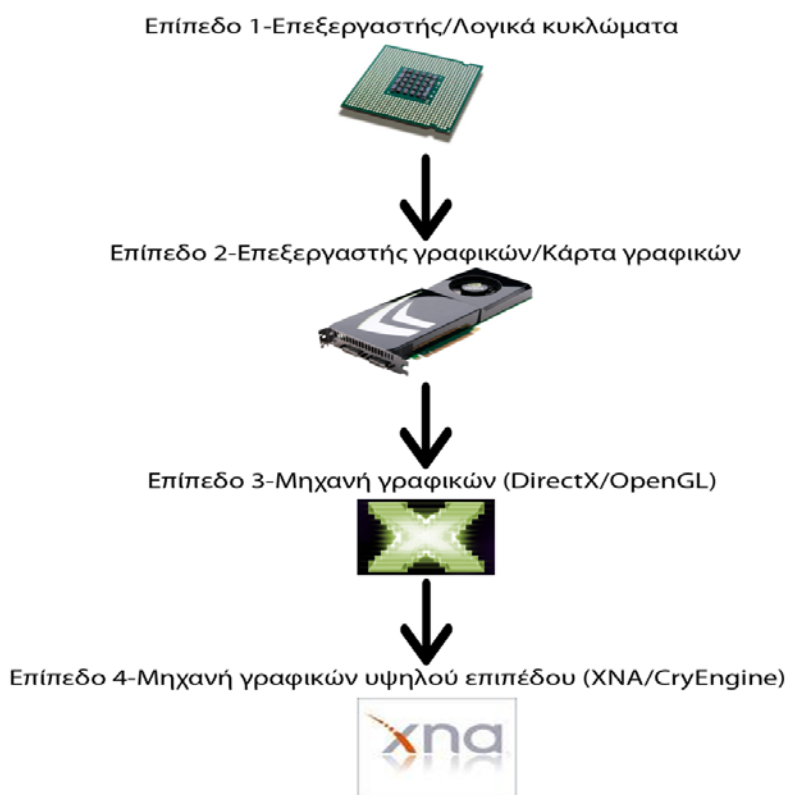
αυτού δημιουργήθηκε από βασικά δομικά στοιχεία ψηφιακών κυκλωμάτων -πύλες-, μετρητές και χρονοδιακόπτες. Ήταν η εποχή της γέννησης της αυτοκρατορίας των παιχνιδιών.



**Εικόνα 2.** Ο κερματοδέκτης της εταιρίας Atari για το παιχνίδι Pong.

Λόγω του υψηλού ενδιαφέροντος του κοινού, οι εταιρίες ξεκίνησαν έναν ξέφρενο αγώνα παραγωγής παιχνιδιών από τότε. Λόγω του μεγάλου ανταγωνισμού και της αύξησης των απαιτήσεων από τη μεριά των χρηστών, τα παιχνίδια σιγά σιγά άρχισαν να γίνονται ολοένα και πιο περίπλοκα. Πλέον η χρήση βασικών δομικών συστατικών ψηφιακών κυκλωμάτων για την δημιουργία παιχνιδιών ήταν ανεπαρκής, λόγω του υψηλού κόστους, της περιπλοκότητας και της έλλειψης δυνατότητας επαναχρησιμοποίησης. Έτσι σταδιακά, και ακόμη πιο έντονα με την εμφάνιση του δομημένου προγραμματισμού, οι εταιρίες ξεκίνησαν να αναπτύσσουν τα παιχνίδια τους σε κώδικα παρά σε ψηφιακά κυκλώματα. Αυτό αρχικά, κυρίως λόγω των τότε χαμηλών απαιτήσεων, φαινόταν επαρκές. Όμως, εξαιτίας της αύξησης της περιπλοκότητας των παιχνιδιών, άρχισαν να εμφανίζονται χιλιάδες γραμμές κώδικα οι οποίες ήταν δύσκολες στην συντήρησή τους και ακόμη πιο δύσκολο για κάποιον καινούριο προγραμματιστή στον χώρο των παιχνιδιών να τις κατανοήσει. Επίσης, αξίζει να σημειώσουμε εδώ, ότι πλέον ο επεξεργαστής γραφικών αποτελούσε ξεχωριστό κομμάτι του υπολογιστή, επειδή ο όγκος για την επεξεργασία δεδομένων ήταν πολύς και υπήρχε μεγάλη άσκοπη σπατάλη πόρων του υπολογιστή και του επεξεργαστή, ειδικά σε συστήματα που έτρεχε από πίσω κάποιο λειτουργικό σύστημα. Το chip αυτό έγινε γνωστό με το όνομα **κάρτα γραφικών**. Στη συνέχεια, εφευρέθηκε η πρώτη **μηχανή γραφικών** από την εταιρία **Microsoft** στις 30 Σεπτεμβρίου 1995 με το όνομα **DirectX**.

Με απλά λόγια, μια μηχανή γραφικών δεν είναι τίποτα άλλο παρά ομαδοποιημένες έτοιμες συναρτήσεις με βασικά εργαλεία που είναι χρήσιμα στην δημιουργία παιχνιδιών. Για παράδειγμα, περιλαμβάνουν τον σχεδιασμό τριγώνων, τον έλεγχο συγκρούσεων, την δικτύωση και άλλα πράγματα τα οποία είναι κοινά για όλα τα παιχνίδια και δεν χρειάζεται ξανά και ξανά να εφευρίσκουμε τον τροχό από την αρχή. Σήμερα το **DirectX** έχει φτάσει αισίως στην έκδοση 11. Ο προγραμματισμός των παιχνιδιών έγινε ακόμα ευκολότερος και έχουμε την εμφάνιση ακόμη πιο εξεζητημένων και περίπλοκων παιχνιδιών. Με την ξέφρενη ανάπτυξη των παιχνιδιών και την εισαγωγή καινούριων εννοιών στον χώρο των βιντεοπαιχνιδιών, όπως για παράδειγμα η **TN** (Τεχνητή νοημοσύνη), η χρήση της μηχανής γραφικών DirectX ήταν πλέον δύσκολη, διότι περιελάμβανε έννοιες τριγώνων και σχημάτων και όχι κάτι πιο εξειδικευμένο. Έτσι εφευρέθηκαν νέες μηχανές γραφικών οι οποίες ομαδοποίησαν τις εντολές της **DirectX** σε ένα επίπεδο πιο επάνω. Σε αυτό ακριβώς το επίπεδο βρίσκεται και η μηχανή γραφικών που χρησιμοποιείται σε αυτήν την πτυχιακή εργασία, η μηχανή **Microsoft XNA**. Άλλες παρόμοιες μηχανές ιδίου επιπέδου είναι η **Cry Engine** από την εταιρία **CryTek** και η **Unreal Engine**, από την εταιρία **Unreal Technologies**. Ένα διάγραμμα για να κατανοήσουμε τα επίπεδα αυτά παρατίθεται παρακάτω:



**Εικόνα 3.** Τα επίπεδα προγραμματισμού γραφικών.



Κεφάλαιο 3

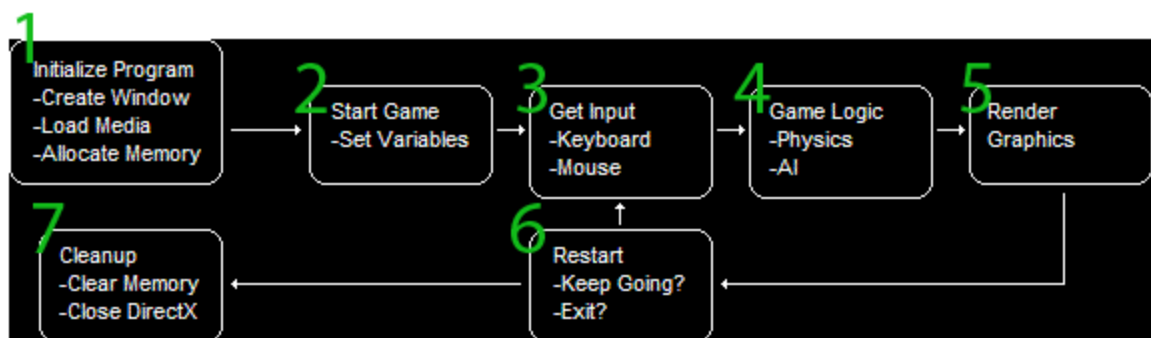
Εισαγωγή στον προγραμματισμό τρισδιάστατων γραφικών

The logo for XNA (Xbox Game Studio) features the lowercase letters 'xna' in a grey, 3D-style font. The letter 'x' is stylized with a red and orange gradient, resembling a pencil or a stylized 'x' shape.

Όπως είδαμε σε προηγούμενο κεφάλαιο, ο προγραμματισμός των παιχνιδιών λόγω των υψηλών απαιτήσεων και της αύξησης περιπλοκότητας γίνεται ολοένα και με πιο εξελιγμένα εργαλεία. Παρ' όλη όμως την χρήση των εξελιγμένων εργαλείων αυτών, όλες οι μηχανές γραφικών και τα εξελιγμένα προγράμματα που χρησιμοποιούνται για την σχεδίαση μοντέλων, όπως είδαμε, στην βάση τους χρησιμοποιούν την ίδια λογική. Πριν ξεκινήσουμε τον σχεδιασμό μοντέλων και άλλων όμορφων πραγμάτων με την χρήση μιας μηχανής γραφικών πρέπει να κατασκευάσουμε έναν «χώρο» μέσα στον οποίο θα μπορούμε να εμφανίσουμε τα μοντέλα μας. Η εμφάνιση μοντέλων, τριγώνων και άλλων αντικειμένων που χρησιμοποιούνται στα παιχνίδια γίνεται μέσα σε ένα παράθυρο. Αυτή η διαδικασία εμφάνισης αντικειμένων στην οθόνη μέσω του παραθύρου αναφέρεται συχνά στον κόσμο των παιχνιδιών με την Αγγλική ορολογία **render**, ο οποίος και θα χρησιμοποιείται συχνά από εδώ και πέρα. Η μετάφραση στα Ελληνικά δεν είναι και τόσο εύστοχη και για αυτό το λόγο αυτός ο όρος, όπως και πολλοί άλλοι μέσα σε αυτήν την εργασία, θα αναφέρονται με την Αγγλική ορολογία, γιατί πολλές φορές η μετάφραση από τα Αγγλικά στα Ελληνικά είναι άστοχη. Προχωρώντας λοιπόν και με ελάχιστη γνώση προγραμματισμού, για να σχεδιάσουμε γραφικά γίνεται κατανοητό πως πρέπει πρωτίστως να σχεδιάσουμε το παράθυρό μας με μια γλώσσα προγραμματισμού, όπως για παράδειγμα με την C++ ή την C#, η οποία και χρησιμοποιείται σε αυτήν την εργασία. Τα «έτοιμα» παράθυρα που μας προσφέρουν τα σύγχρονα περιβάλλοντα ανάπτυξης εφαρμογών όπως το **Visual Studio** της **Microsoft**, δεν μας είναι αρκετά γιατί θα θέλαμε ιδανικά να έχουμε τον πλήρη έλεγχο του παραθύρου και να χειριζόμαστε τα **interrupts** (τα σήματα διακοπής που δέχεται ο Η/Υ ώστε να επικοινωνήσει με συσκευές εισόδου) τα οποία αυτό λαμβάνει, όπως για παράδειγμα την κίνηση του ποντικιού, το πάτημα ενός πλήκτρου στο πληκτρολόγιό μας και άλλα. Επίσης, λόγω του γεγονότος ότι το παράθυρο μας πρέπει να το χειρίζεται η μηχανή γραφικών για να κάνει **render** και όχι για να εμφανίζει κουμπιά και άλλα, η λύση είναι μία: Να κατασκευάσουμε το δικό μας παράθυρο και να το χειριστούμε όπως μας εξυπηρετεί καλύτερα.

Στην ουσία, όταν αναπτύσσουμε εφαρμογές με τα σύγχρονα περιβάλλοντα ανάπτυξης, αυτά όλα κατασκευάζονται αυτόματα με κώδικα. Πρέπει, επίσης, να έχουμε υπόψη μας ότι τα παράθυρα αυτά, δεν είναι στατικά, δηλαδή όταν «τρέχουμε» το πρόγραμμα και βλέπουμε το παράθυρο αυτό δεν «περιμένει» θα λέγαμε από μας να πατήσουμε ένα πλήκτρο για να κάνει κάτι, αλλά τρέχοντάς το πρόγραμμα μπαίνει σε έναν ατέρμονο

βρόγχο και σε κάθε επανάληψη ελέγχει για τυχόν **εισόδους** από τον χρήστη, δηλαδή για πάτημα πλήκτρου, κίνηση ποντικιού και άλλα. Αυτός ακριβώς ο βρόγχος είναι ζωτικής σημασίας για το παιχνίδι μας και είναι κάτι το οποίο μαθαίνει κάποιος νέος προγραμματιστής εισερχόμενος στον χώρο των βιντεοπαιχνιδιών. Ακόμη, αυτός ο βρόγχος είναι γνωστός και αναφέρεται στον προγραμματισμό παιχνιδιών με τον όρο **Main Game Loop**. Γενικά για κατανοήσουμε τα βασικά και να δούμε με ποια λογική «τρέχει» ένα παιχνίδι, παρακάτω παρατίθεται ένα διάγραμμα που θα μας βοηθήσει να κατανοήσουμε τις **εφτά φάσεις ενός παιχνιδιού**. Η εξήγηση έρχεται αμέσως μετά.



**Εικόνα 4.** Οι εφτά φάσεις ενός παιχνιδιού.

Πηγή: [www.directxtutorial.com](http://www.directxtutorial.com).

Όπως αναφέραμε και στην αρχή αυτού του κεφαλαίου και βλέπουμε στην εικόνα 4, πριν ακόμα ξεκινήσουμε να σχεδιάζουμε μοντέλα και άλλα πράγματα με την μηχανή γραφικών, πρέπει να περάσουμε από την **φάση 1** η οποία περιλαμβάνει κατασκευή ενός παραθύρου και τη δέσμευση της απαραίτητης μνήμης.

### **Φάση 1- Κατασκευή παραθύρου, δέσμευση μνήμης**

Από εδώ ακριβώς ξεκινάει το παιχνίδι μας. Γίνεται η φόρτωση της μηχανής γραφικών, όπως για παράδειγμα η **DirectX**, η κατασκευή του παραθύρου και η παραμετροποίησή του, δηλαδή εάν είναι σε πλήρη οθόνη, τι τίτλο θα έχει και άλλα. Επίσης, γίνεται ο ορισμός του παραθύρου αυτού ως το **main** παράθυρο και ότι ο χειριστής του θα είναι η μηχανή γραφικών που έχουμε ορίσει. Με απλά λόγια ρυθμίζουμε την μηχανή γραφικών να «ζωγραφίζει» μέσα σε αυτό το παράθυρο. Τέλος, γίνεται η απαραίτητη δέσμευση μνήμης που θα χρειαστούμε.

## **Φάση 2- Εκκίνηση παιχνιδιού**

Εδώ πλέον φτάνουμε ένα βήμα πριν την **Main Game Loop** όπως αναφέραμε και πριν. Είναι το σημείο εκκίνησης του ατέρμονου βρόγχου, από τον οποίο και δεν θα «βγει» το παιχνίδι μας μέχρι να το σταματήσουμε εμείς. Τον τρόπο θα τον αναφέρουμε στην συνέχεια. Σε αυτήν την φάση, φορτώνουμε τα μοντέλα μας, ορίζουμε τις αρχικές μας μεταβλητές, όπως για παράδειγμα τον χάρτη, το σημείο εκκίνησης του παίχτη και άλλα.

## **Φάση 3- Λήψη εισόδου από την χρήστη**

Είναι το σημείο όπου το παιχνίδι μας λαμβάνει από τον χρήστη τα λεγόμενα **interrupts** ή αλλιώς εισόδους από το πληκτρολόγιο, το ποντίκι ή ακόμα και από ένα χειριστήριο παιχνιδιού. Αυτή η φάση, που είναι και η μοναδική, είναι η μόνη γέφυρα που συνδέει το παιχνίδι με το χρήστη. Με άλλα λόγια ο χρήστης μπορεί με αυτόν τον τρόπο να «επικοινωνήσει» με τον παίχτη.

## **Φάση 4- Επεξεργασία λογικών του παιχνιδιού**

Σε αυτήν την φάση ελέγχουμε και ορίζουμε το «τι μέλει γενέσθαι» στο παιχνίδι μας. Δηλαδή το πού ακριβώς μετακινήθηκε ο παίχτης μας τα τελευταία δευτερόλεπτα, πόσες ακριβώς σφαίρες έχουν μείνει, αν το πλοίο που οδηγούμε συγκρούστηκε πάνω σε κάποιο βράχο και άλλα.

## **Φάση 5- Εμφάνιση γραφικών**

Εδώ, «ζωγραφίζουμε» τα μοντέλα μας και τα αντικείμενα μας στην οθόνη. Αυτή η φάση έχει άμεση σχέση με την προηγούμενη και το τι θα ζωγραφιστεί έχει να κάνει με την προηγούμενη φάση.

## **Φάση 6- Έλεγχος για έξοδο ή για συνέχεια**

Σε αυτήν την φάση το παιχνίδι μας ελέγχει το τι έγινε στις προηγούμενες φάσεις, και ανάλογα αποφασίζεται αν θα συνεχιστεί ο βρόγχος ή θα τερματιστεί. Είναι κατανοητό το γεγονός ότι εάν τερματιστεί ο βρόγχος, θα γίνει έξοδος από το παιχνίδι, ενώ διαφορετικά, ο βρόγχος θα συνεχιστεί κανονικά από την φάση 3 και έπειτα.



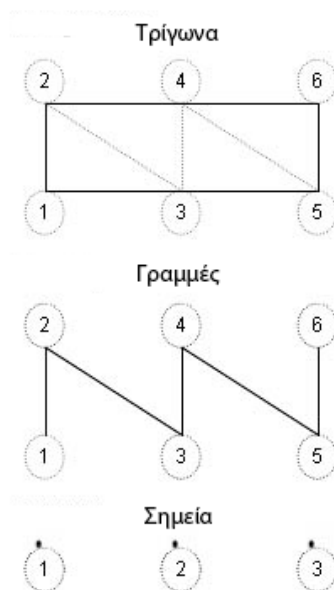
## Φάση 7- Καθαρισμός γραφικών, απελευθέρωση μνήμης

Αυτή η φάση είναι συμπληρωματική και δεν καλείται συνεχώς στην **Main Game Loop** παρά μόνο όταν αυτό ζητηθεί. Από τον τίτλο καταλαβαίνουμε ότι εδώ μπορούμε να καθαρίσουμε μοντέλα ή και μνήμη η οποία δεν χρησιμοποιείται πλέον και να την αφήσουμε για μετέπειτα χρήση ώστε να μην έχουμε υπερχείλιση μνήμης.

Αυτό λοιπόν ήταν ένα βασικό κομμάτι για να κατανοήσουμε το πώς λειτουργεί ένα παιχνίδι, με ποια, δηλαδή, λογική. Με τι εντολές εμφανίζουμε αντικείμενα, ελέγχουμε για συγκρούσεις και αποδεσμεύουμε μνήμη έχει να κάνει με την εκάστοτε μηχανή γραφικών και δεν είναι κάτι γενικό. Η λογική μόνο παραμένει η ίδια.

Παρακάτω θα μιλήσουμε για τα **βασικά συστατικά** ή **βασικά στοιχεία** από τα οποία απαρτίζονται όλα τα αντικείμενα και τα μοντέλα που βλέπουμε εμείς στην οθόνη μας. Κάθε φορά που μιλάμε για **βασικά στοιχεία** για τα παιχνίδια είναι σαν να μιλάμε για μόρια και άτομα στην επιστήμη της φυσικής. Είναι δηλαδή κάτι από το οποίο απαρτίζονται όλα όσα βλέπουμε ως γραφικά στην οθόνη μας και είναι ίδια για τα πιο απλά μοντέλα αλλά και τα πιο σύνθετα. Το πιο γνωστό **βασικό στοιχείο** που όλες οι μηχανές γραφικών υποστηρίζουν είναι τα **τρίγωνα**. Πολύπλοκα μοντέλα, όπως η βροχή που βλέπουμε να πέφτει στο παιχνίδι μας, το αεροπλάνο που πετάει στους ουρανούς του παιχνιδιού μας, είναι αποτέλεσμα εκατομμυρίων τριγώνων τα οποία ενωμένα μεταξύ τους μας δίνουν αυτήν την «ψευδαίσθηση» του βουνού, του νερού, του ουρανού. Ακόμη και τα προγράμματα κατασκευής μοντέλων δουλεύουν με τρίγωνα και σε προηγούμενο κεφάλαιο που μιλούσαμε για εξαγωγή μοντέλων σε μορφή κατανοητή από την μηχανή γραφικών από τα προγράμματα κατασκευής μοντέλων, αυτά δεν εξάγουν τίποτε άλλο παρά ένα κείμενο με τον αριθμό των τριγώνων, το χρώμα τους, το πού βρίσκονται οι κορυφές τους και άλλα στοιχεία. Συχνά, επίσης, βλέπουμε κάρτες γραφικών να γράφουν στα χαρακτηριστικά τους ότι μπορούν να εμφανίσουν «χ εκατομμύρια τρίγωνα το δευτερόλεπτο». Η χρήση τριγώνων, λοιπόν, απ' ό,τι διαπιστώνουμε είναι πλέον δεδομένη στη βιομηχανία παιχνιδιών. Βεβαίως υπάρχουν και άλλα **βασικά στοιχεία** με τα οποία μπορούμε να σχεδιάσουμε γραφικά, όπως για παράδειγμα τελείες (σημεία) και γραμμές. Μα για ποιο λόγο, όμως, χρησιμοποιούνται ευρέως τα τρίγωνα και όχι, για παράδειγμα, οι γραμμές; Η απάντηση είναι πολύ απλή. Τα τρίγωνα και λόγω των τριών κορυφών τους μας είναι χρήσιμα για να

κατασκευάσουμε άλλα βασικά σχήματα. Σκεφτείτε μόνο ότι δύο τρίγωνα είναι ένα τετράγωνο. Ένας κύλινδρος τρισδιάστατος θα ήταν πολλά, ή μάλλον εκατοντάδες, τρίγωνα. Λόγω του γεγονότος ότι πλέον ακόμα και για να σχεδιάσουμε μια «τρισδιάστατη γραμμή» οι γραμμές δεν είναι αρκετές (αυτό θα ήταν συνετό μόνο στα πρώτα παιχνίδια της δεκαετίας του '70) και, όπως είπαμε και πριν, λόγω των κορυφών των τριγώνων και ότι αυτές μπορούν να ενωθούν ώστε να σχεδιάσουν πολύπλοκα μοντέλα, μπορούμε να γλιτώσουμε μεγάλο **bandwidth** στην κάρτα γραφικών μας. Τα νούμερα τα οποία εξοικονομούμε, για παράδειγμα στο σχεδιασμό ενός απλού τετραγώνου, είναι μικρά από την χρήση τριγώνων αντί γραμμών ή σημείων. Εάν σκεφτούμε, όμως, ότι τα σύγχρονα παιχνίδια αποτελούνται από εκατοντάδες μοντέλα τα οποία κάθε δευτερόλεπτο κάνουν πολλές κινήσεις, τα νούμερα εξοικονόμησης είναι τρομακτικά και δεν θα ήταν δυνατή η εμφάνισή τους, παρά μόνο με την βοήθεια πανάκριβων καρτών γραφικών οι οποίες, φυσικά, θα ανέβαζαν κατά πολύ το κόστος του υπολογιστή μας. Παρακάτω παρατίθεται ένα διάγραμμα που δείχνει μερικά **βασικά στοιχεία** γραφικών.



**Εικόνα 5.** Τα βασικά στοιχεία των γραφικών.

Κλείνοντας το κεφάλαιο αυτό, συμπερασματικά θα λέγαμε ότι τα σύγχρονα παιχνίδια, από τα πιο απλά έως τα πιο σύνθετα και εντυπωσιακά, χρησιμοποιούν στην ουσία την τεχνική του ατέρμονου βρόγχου, για να λαμβάνουν **μηνύματα** από τους χρήστες όπως είδαμε και πριν. Τέλος, για να μπορέσουν να ζωγραφίσουν κάτι στην οθόνη χρησιμοποιούν τα **βασικά στοιχεία** της μηχανής γραφικών. Βεβαίως, θα αναρωτηθεί

κάποιος, εάν μόνο αυτά τα δύο είναι επαρκή για να καταφέρουμε αυτά τα εφέ ή τις αντανakλάσεις και τους φωτισμούς που βλέπουμε στα σύγχρονα παιχνίδια. Η απάντηση φυσικά είναι όχι και όπως θα δούμε, υπάρχουν κάποιες άλλες τεχνικές που χρησιμοποιούνται, για παράδειγμα στον φωτισμό και στις αντανakλάσεις, απλώς αυτά που παρουσιάστηκαν σε αυτό το κεφάλαιο είναι τα βασικά και σε αυτά βασίζονται και όλα τα υπόλοιπα.

Κεφάλαιο 4

Παρουσίαση του Microsoft Visual Studio 2010

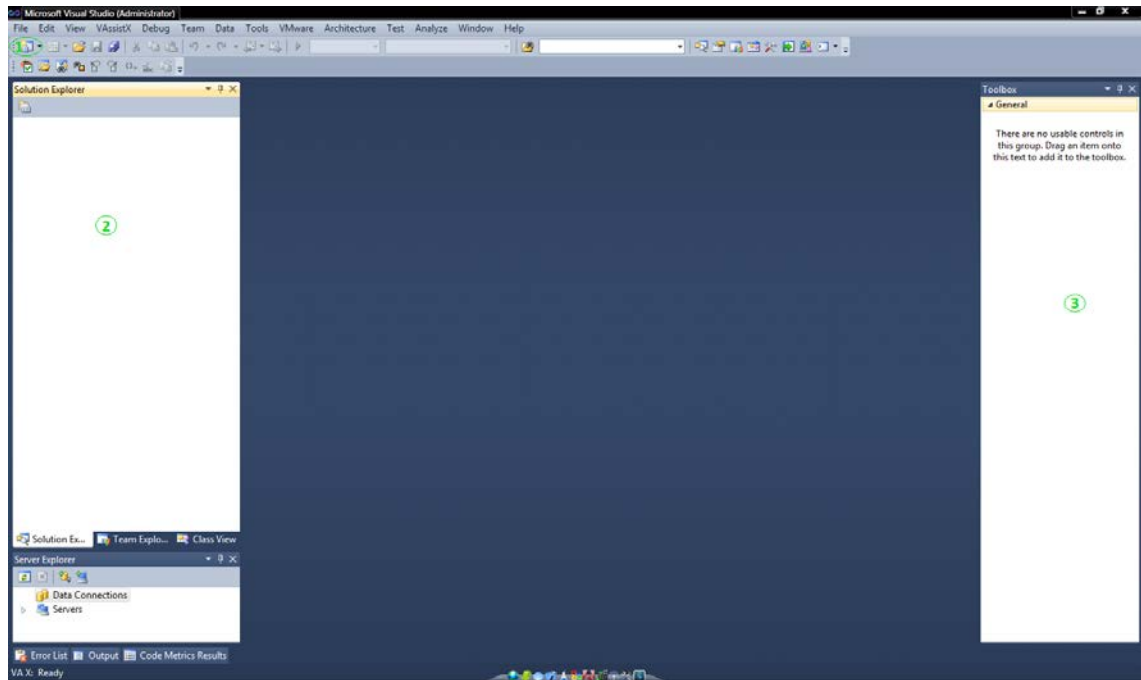
The XNA logo features the letters 'xna' in a stylized, grey, 3D font. The 'x' is uniquely designed with an orange and red gradient, resembling a pencil or a stylized 'x' shape.



Ο προγραμματισμός ποτέ πριν δεν ήταν πιο απλός. Με το περιβάλλον ανάπτυξης **Microsoft Visual Studio 2010**, τα πράγματα γίνονται ακόμη πιο απλά. Η κατασκευή εφαρμογών είναι πλέον παιχνίδι. Σε αυτό το κεφάλαιο παρουσιάζεται το περιβάλλον ανάπτυξης που χρησιμοποιήθηκε σε αυτήν την εργασία που είναι το **Microsoft Visual Studio 2010**, καθώς και δύο παραδείγματα έτσι ώστε να μπορεί να γίνει μία προσαρμογή, για αυτούς που δεν το έχουν ξαναχρησιμοποιήσει. Για το λόγο αυτό, παρουσιάζουμε την κατασκευή μίας εφαρμογής σε κονσόλα και έπειτα την κατασκευή μίας εφαρμογής με φόρμες. Σε επόμενα κεφάλαια, βέβαια, θα αναφερθεί και πώς κατασκευάζουμε εφαρμογές με την μηχανή γραφικών **Microsoft XNA Game Studio**.

Πριν ξεκινήσουμε, όμως, θα πρέπει να έχουμε εγκαταστήσει στον υπολογιστή μας το **Microsoft Visual Studio 2010**. Εάν προσπαθήσετε να κατεβάσετε το περιβάλλον ανάπτυξης αυτό θα διαπιστώσετε πως υπάρχουν πολλές διαφορετικές εκδόσεις. Μην σας ανησυχεί αυτό, όμως, διότι και η απλή έκδοση μπορεί κάνει την δουλειά που θέλουμε και, μάλιστα διατίθεται δωρεάν.

Σε αυτήν την εργασία χρησιμοποιήθηκε το **Microsoft Visual Studio 2010 Ultimate Edition**. Πηγαίνουμε λοιπόν σε αυτήν την σελίδα και κατεβάζουμε το περιβάλλον ανάπτυξης: [www.microsoft.com/visualstudio/en-us/products/2010-editions/visual-csharp-express](http://www.microsoft.com/visualstudio/en-us/products/2010-editions/visual-csharp-express). Λόγω του ότι σε αυτήν την εργασία χρησιμοποιήθηκε η γλώσσα ανάπτυξης C# θα κατεβάσουμε την έκδοση **Visual C# Express Edition**. Η μηχανή γραφικών **Microsoft XNA Game Studio** υποστηρίζει και την γλώσσα C++ επίσης. Αφού κατεβάσουμε το περιβάλλον ανάπτυξης και το εγκαταστήσουμε θα μας εμφανιστεί η αρχική οθόνη όπως φαίνεται στην παρακάτω εικόνα:

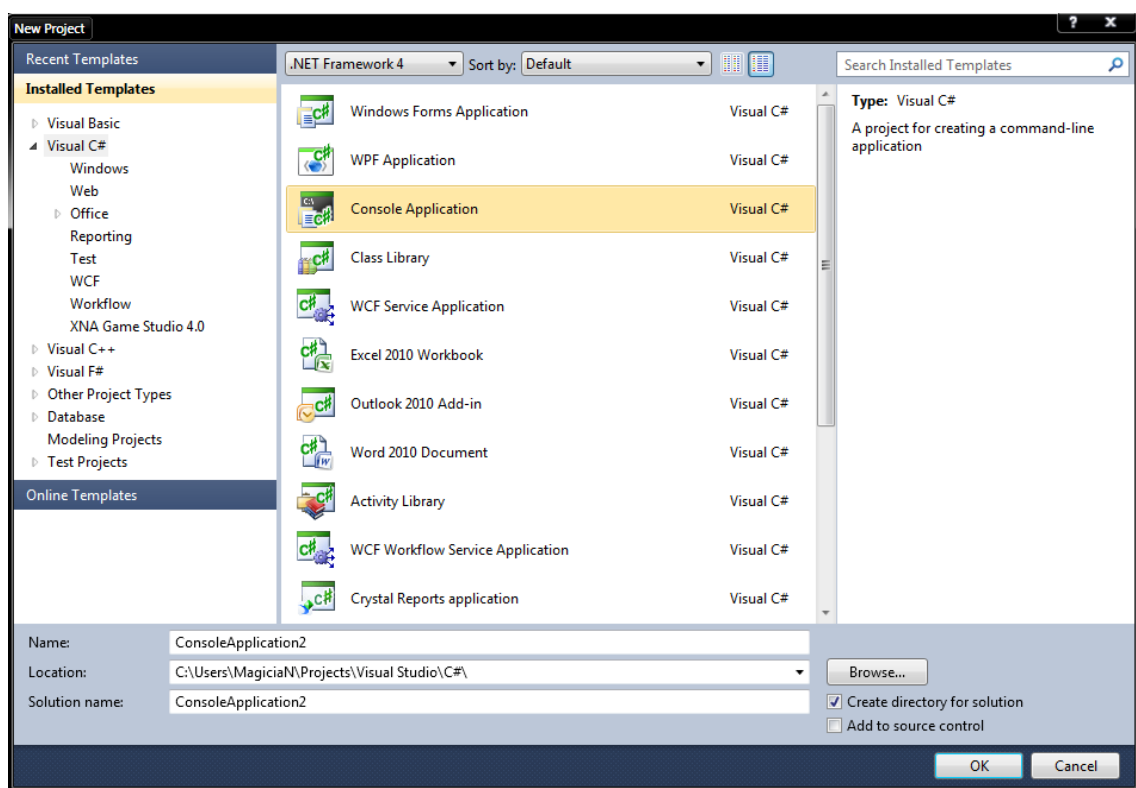


**Εικόνα 6.** Η αρχική οθόνη του Microsoft Visual Studio.

Στο σημείο **1** που φαίνεται στην εικόνα μπορούμε να δημιουργήσουμε ένα καινούριο project, όπως θα δούμε και στην συνέχεια. Στο σημείο **2** βρίσκονται όλες οι κλάσεις μας και τα αρχεία μας που έχουμε δημιουργήσει και τέλος στο σημείο **3** υπάρχουν είτε διάφορα αντικείμενα που μπορούμε να σύρουμε πάνω στην φόρμα, για οπτικές εφαρμογές και διάφορες ρυθμίσεις που αφορούν το project. Στην συνέχεια παρουσιάζεται ένα απλό πρόγραμμα σε κονσόλα για να αποκτηθεί η απαραίτητη εξοικείωση με το περιβάλλον.

## 4.1 - Δημιουργία προγράμματος κονσόλας

Πατάμε επάνω στο κουμπί δημιουργίας νέου project, όπως είδαμε στην προηγούμενη εικόνα είτε πατάμε **Ctrl+Shift+N**. Θα εμφανιστεί μπροστά μας μία οθόνη που θα μας ζητήσει να επιλέξουμε τι είδος project θα θέλαμε να κατασκευάσουμε. Αριστερά, για ορισμένες εκδόσεις του **Microsoft Visual Studio 2010** μπορούμε να επιλέξουμε την γλώσσα προγραμματισμού που θα χρησιμοποιήσουμε. Εμείς επιλέγουμε την γλώσσα C# από αριστερά και δεξιά επιλέγουμε **Console Application**, όπως φαίνεται στην παρακάτω εικόνα. Κάτω, στο σημείο που γράφει **Name**, επιλέγουμε το όνομα που θέλουμε να έχει το project και στο σημείο που γράφει **Location** είναι η τοποθεσία που θέλουμε να αποθηκευτεί. Γράφουμε ένα όνομα και πατάμε **OK**.



Εικόνα 7. Επιλέγουμε αριστερά την γλώσσα C# και δεξιά Console Application

Μπροστά μας πλέον πρέπει να έχει εμφανιστεί ο αρχικός κώδικας του προγράμματος κονσόλας. Αυτός ο κώδικας για τους πρωτάρηδες στον προγραμματισμό με την C# μπορεί να φανεί λίγο περίεργος, αλλά περιέχει κάποια βασικά στοιχεία της γλώσσας τα οποία θα εξηγηθούν στην συνέχεια. Βέβαια, όποιος έχει ασχοληθεί με τη γλώσσα προγραμματισμού **Java**, ο κώδικας ίσως να του φανεί πιο οικείος και αυτό γιατί οι δύο γλώσσες αυτές, C# και Java μοιάζουν πολύ μεταξύ τους. Ο κώδικας που εμφανίζεται έχει ως εξής:

```

1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5
6 namespace ConsoleApplication2
7 {
8     class Program
9     {
10         static void Main(string[] args)
11         {
12         }
13     }
14 }

```

**Κώδικας 1.** Βασικός κώδικας προγράμματος κονσόλας

Οι γραμμές 1 έως 4 είναι γραμμές δηλώσεων και χρειάζονται για να ξέρει ο μεταγλωττιστής ό,τι τυχόν συναρτήσεις που θα χρησιμοποιηθούν, σε ποιο σημείο να ψάξει να τις βρει. Είναι το σημείο όπου δηλώνουμε ποια **namespace** θα χρησιμοποιήσουμε. Τι είναι το **namespace**; Το **namespace** είναι κάτι σαν ομαδοποίηση. Όπως βλέπουμε παρακάτω, στη γραμμή 6 ονομάζουμε όλο το project μας με κάποιο **namespace**. Πράγματα τα οποία έχουν κοινό σκοπό ή έχουν κάτι κοινό μεταξύ τους μπορούμε να τα τοποθετήσουμε στο ίδιο **namespace**. Για παράδειγμα όλες οι κλάσεις που χρησιμοποιήθηκαν στην εργασία αυτή, έχουν μπει όλες κάτω από το ίδιο **namespace** με όνομα **Terrain**. Στην συνέχεια στη γραμμή 8 ξεκινάει η δήλωση μίας κλάσης. Θα πρέπει να ξέρουμε ότι όλα στην γλώσσα προγραμματισμού C# είναι φτιαγμένα σε κλάσεις. Έπειτα στη γραμμή 10 ξεκινάει η συνάρτηση με το όνομα **Main**. Όταν τρέχουμε το πρόγραμμά μας αυτό είναι το σημείο εκκίνησης που θα πάει για να ξεκινήσει την εκτέλεσή του το πρόγραμμα. Φυσικά εάν προσπαθήσουμε να φτιάξουμε ακόμη μία συνάρτηση **Main** το πρόγραμμα δεν θα τρέξει.

Αφού είδαμε μερικά βασικά στοιχεία της γλώσσας προγραμματισμού C#, είναι ώρα να γράψουμε λίγο κώδικα ώστε να κάνουμε μία επισκόπηση αργότερα του προγράμματος κονσόλας. Το πρόγραμμα που θα κατασκευάσουμε θα φτιαχτεί με συναρτήσεις και θα παίρνει δύο αριθμούς από τον χρήστη και θα τους προσθέτει. Επίσης στο τέλος θα μας αναφέρει εάν ο πρώτος αριθμός είναι μικρότερος, μεγαλύτερος ή ίσος με το δεύτερο. Ο κώδικας έχει ως εξής:



```

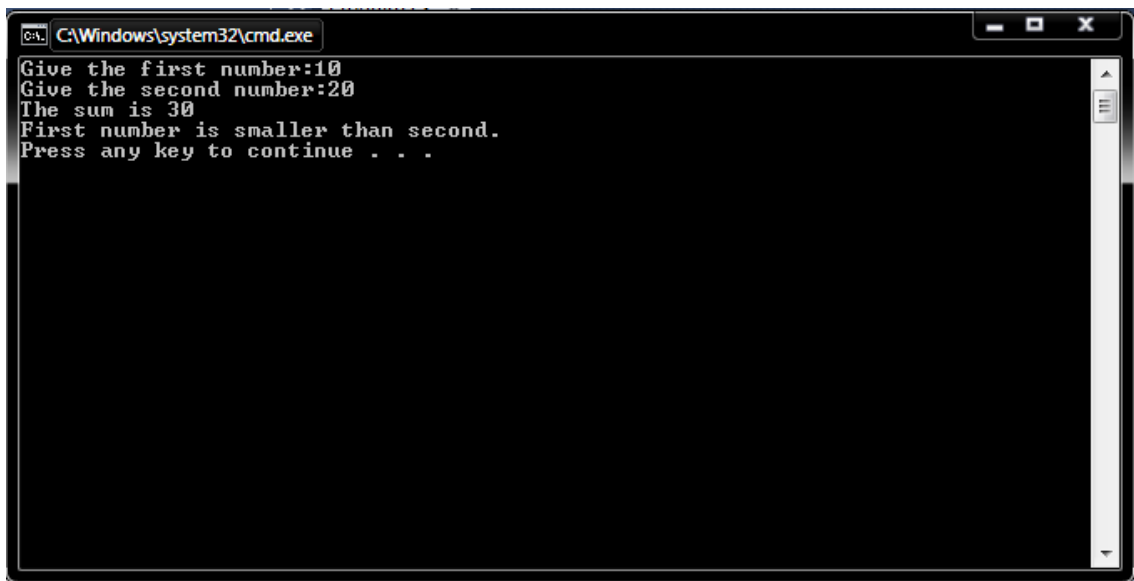
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5
6  namespace ConsoleApplication2
7  {
8      class Program
9      {
10         static void Main(string[] args)
11         {
12             int a, b , result ;
13
14             Console.WriteLine("Give the first number:");
15             a = System.Int32.Parse( Console.ReadLine() ) ;
16             Console.WriteLine("Give the second number:");
17             b = System.Int32.Parse(Console.ReadLine());
18
19             result = sum(a, b);
20             Console.WriteLine("The sum is {0}",result);
21             compare(a, b);
22         }
23
24         public static int sum(int x, int y)
25         {
26             return (x + y);
27         }
28
29         public static void compare(int x, int y)
30         {
31             if ( x>y )
32             {
33                 Console.WriteLine("First number is bigger than second.");
34             }
35             else if ( x<y )
36             {
37                 Console.WriteLine("First number is smaller than second.");
38             }
39             else
40             {
41                 Console.WriteLine("First number is equal with second.");
42             }
43         }
44     }
45 }
46

```

## Κώδικας 2. Κώδικας προγράμματος κονσόλας

Στην γραμμή 12 δηλώνουμε απλώς δύο μεταβλητές όπου και θα κρατήσουμε την είσοδο που θα μας δώσει ο χρήστης. Η τελευταία μεταβλητή με το όνομα **result**, θα κρατήσει το αποτέλεσμα της πρόσθεσης. Στην γραμμή 14 βλέπουμε την εντολή **Console.WriteLine(string)**. Η εντολή αυτή μας βοηθά να εκτυπώσουμε κείμενο στην οθόνη ώστε να κάνουμε το πρόγραμμά μας πιο διαδραστικό στον χρήστη. Στην γραμμή 15 ορίζουμε ότι η μεταβλητή **a** θα πάρει το περιεχόμενο που θα επιστρέψει η εντολή **Console.ReadLine()** και θα το μετατρέψει σε ακέραιο αριθμό με την εντολή **System.Int32.Parse(Object)**. Η εντολή **Console.ReadLine()** διαβάζει το κείμενο που θα πληκτρολογήσει ο χρήστης. Στις γραμμές 24 και 29 φτιάχνουμε δύο συναρτήσεις **static**, διότι και η **main** συνάρτηση είναι ορισμένη ως **static**, η πρώτη επιστρέφει το άθροισμα των δύο αριθμών και η δεύτερη ελέγχει για το ποιος αριθμός είναι

μεγαλύτερος και εκτυπώνει το ανάλογο αποτέλεσμα στην οθόνη. Εάν τρέξουμε το πρόγραμμα και βάλουμε για παράδειγμα τους αριθμούς 10 και 20 το αποτέλεσμα θα είναι το εξής:



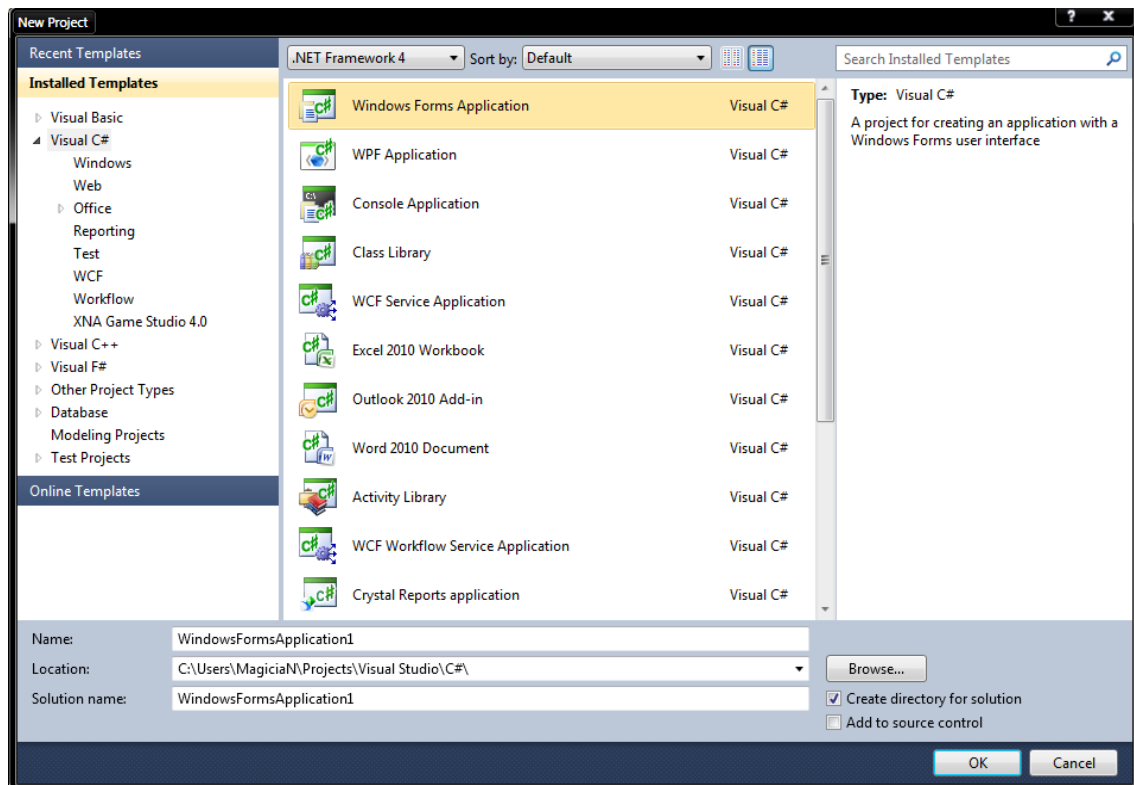
```
C:\Windows\system32\cmd.exe
Give the first number:10
Give the second number:20
The sum is 30
First number is smaller than second.
Press any key to continue . . .
```

**Εικόνα 8.** Τα αποτελέσματα του προγράμματος επί της οθόνης.

Αυτό ήταν ένα απλό πρόγραμμα σε κονσόλα έτσι ώστε να αποκτήσουμε μία απλή εξοικείωση και με το περιβάλλον αλλά και με την γλώσσα προγραμματισμού C#. Στην συνέχεια παρουσιάζεται και ένα πρόγραμμα με φόρμες.

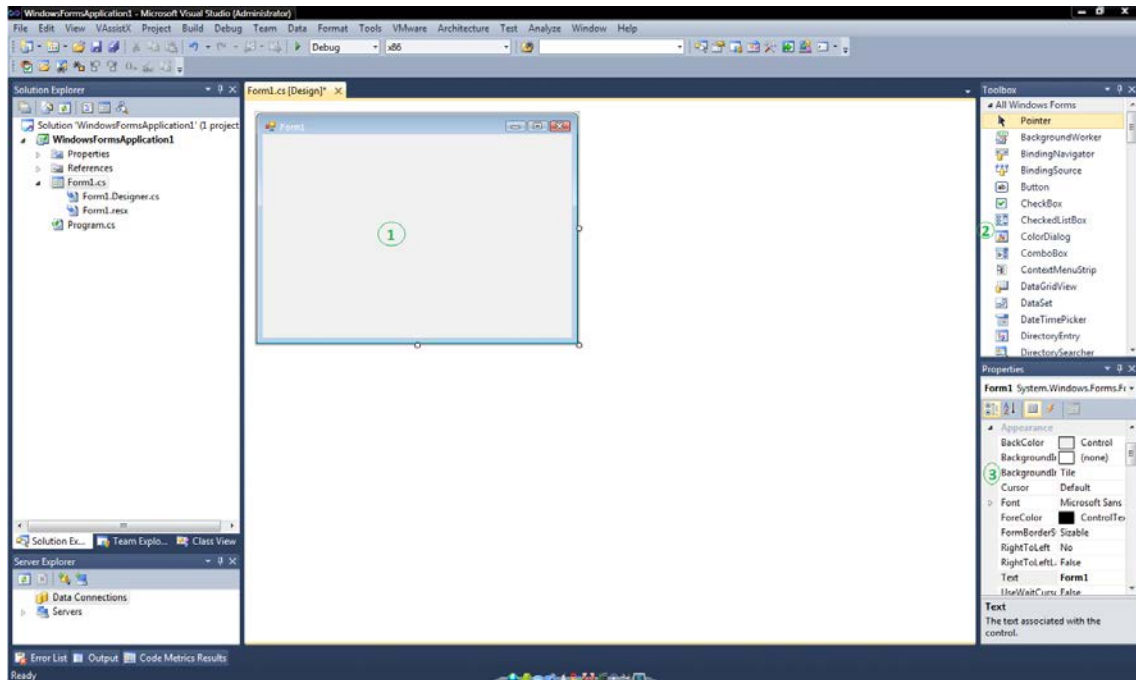
## 4.2 - Δημιουργία οπτικής εφαρμογής

Όπως και πριν, θα δημιουργήσουμε ένα νέο project, θα επιλέξουμε για γλώσσα προγραμματισμού την C#, αλλά αυτή τη φορά θα επιλέξουμε **WindowsFormsApplication**, ώστε να δημιουργήσουμε ένα πρόγραμμα με φόρμες, όπως φαίνεται στην εικόνα παρακάτω:



Εικόνα 9. Δημιουργία προγράμματος Windows Forms Application.

Τέλος, επιλέγουμε ένα όνομα, μία τοποθεσία και πατάμε το κουμπί OK. Μπροστά μας θα εμφανιστεί μία φόρμα όπου εκεί μπορούμε να σύρουμε επάνω κουμπιά και άλλα αντικείμενα όπως θα δούμε στη συνέχεια. Η φόρμα που θα εμφανιστεί φαίνεται παρακάτω:



Εικόνα 10. Η φόρμα που εμφανίζεται δημιουργώντας Windows Forms Application.

Στο σημείο 1 βρίσκεται η αρχική φόρμα, στο σημείο 2 βρίσκονται τα αντικείμενα τα οποία μπορούμε να σύρουμε επάνω στην φόρμα και τέλος στο σημείο 3 βρίσκονται οι ρυθμίσεις για το επιλεγμένο αντικείμενο. Όταν πατάμε ένα αντικείμενο επάνω στην φόρμα οι ρυθμίσεις αλλάζουν δυναμικά ανάλογα με το τι αντικείμενο έχουμε εμείς επιλέξει. Από τα αντικείμενα δεξιά επιλέγουμε και σέρνουμε επάνω στην φόρμα δύο **TextBox**. Επίσης επιλέγουμε και σέρνουμε ένα **Button**. Θα παρατηρήσουμε τώρα εάν επιλέξουμε ένα αντικείμενο που βρίσκεται επάνω στη φόρμα οι ρυθμίσεις κάτω δεξιά θα αλλάξουν. Θα προγραμματίσουμε τώρα το πρόγραμμά μας έτσι ώστε όταν πατάμε το κουμπί να παίρνουμε τους αριθμούς που έχει γράψει μέσα ο χρήστης και να εμφανίζουμε το αποτέλεσμα της πρόσθεσης στην οθόνη. Έτσι απλά λοιπόν πατάμε διπλό κλικ επάνω στο κουμπί που βρίσκεται στην φόρμα και γράφουμε τον εξής κώδικα:

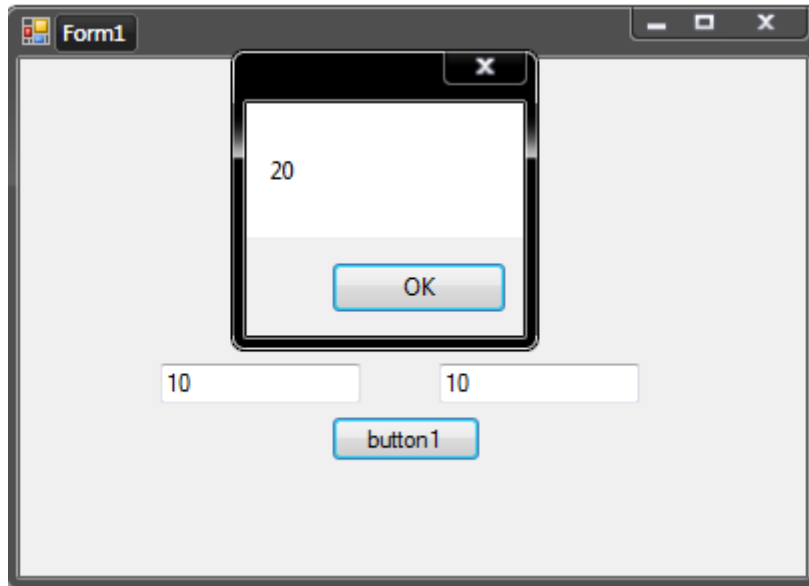
```

19 private void button1_Click(object sender, EventArgs e)
20 {
21     int a, b , result ;
22     a = System.Int32.Parse(textBox1.Text);
23     b = System.Int32.Parse(textBox2.Text);
24     result = a + b;
25
26     MessageBox.Show(result.ToString());
27 }

```

Κώδικας 3. Προγραμματισμός κουμπιού WindowsFormsApplication

Στις γραμμές 22 και 23 ορίζουμε ότι οι ακέραιοι αριθμοί a και b θα πάρουν ως είσοδο το κείμενο των δύο textBox1 και textBox2 αντίστοιχα. Στην σειρά 26 εμφανίζουμε ένα κουτί μηνύματος με κείμενο το αποτέλεσμα του a και b σε string. Εάν τρέξουμε το πρόγραμμα και βάλουμε τους αριθμούς 10 και 20 το αποτέλεσμα θα είναι το εξής:



**Εικόνα 11.** Το τελικό πρόγραμμα Windows Forms Application.

Αφού λοιπόν έγινε μία σύντομη παρουσίαση του **Microsoft Visual Studio 2010** καθώς και δύο απλών παραδειγμάτων για να κατανοήσουμε την λογική είναι καιρός να συνδυάσουμε αυτές τις γνώσεις και του προηγούμενου κεφαλαίου, με την παρουσίαση της λογικής των μηχανών γραφικών, ώστε να κάνουμε μία επισκόπηση της μηχανής γραφικών **Microsoft XNA Game Studio**. Σίγουρα αυτό το εγχειρίδιο δεν είναι το κατάλληλο για να εξοικειωθούμε επαρκώς με το **Microsoft Visual Studio 2010**. Αυτό θα γίνει είτε διαβάζοντας κάποιο άλλο σχετικό εγχειρίδιο είτε με προσωπική μας ενασχόληση. Το κείμενο αυτό απλώς καλύπτει τα βασικά στοιχεία του **Microsoft Visual Studio 2010** και σκοπός του δεν είναι να αποτελέσει ένα πλήρες εγχειρίδιο αυτού του περιβάλλοντος ανάπτυξης.



Κεφάλαιο 5

Παρουσίαση της μηχανής γραφικών Microsoft XNA

The logo for Microsoft XNA, featuring the letters 'xna' in a stylized, metallic font. The 'x' is orange and has a small orange and white striped object resembling a pencil or a stylus pointing towards it.

Είδαμε στο προηγούμενο κεφάλαιο μερικά παραδείγματα με το περιβάλλον ανάπτυξης Microsoft Visual Studio 2010 για να κατανοήσουμε την λογική του και να βοηθήσουμε τον αναγνώστη να έχει μία ομαλή μετάβαση στον κώδικα που αφορά το σχεδιασμό γραφικών. Παρακάτω παρουσιάζεται η μηχανή γραφικών Microsoft XNA και η λογική με την οποία καταφέρνει να υλοποιήσει τελικά τα γραφικά που εμείς βλέπουμε στην οθόνη. Όπως αναφέραμε και στο κεφάλαιο 3, την λογική που χρησιμοποιούν οι μηχανές γραφικών για να υλοποιήσουν τα γραφικά, την ίδια ακριβώς λογική χρησιμοποιεί και η μηχανή γραφικών Microsoft XNA· την λογική, δηλαδή, της φόρτωσης αντικειμένων στην μνήμη και την επιτέλεση του ατέρμονος βρόγχου ώστε να εκτελεί λειτουργίες λογικής, όπως εάν ένα αντικείμενο χτύπησε με κάποιο άλλο και να ελέγχει εάν ο χρήστης έχει πατήσει κάποιο κουμπί έτσι ώστε να υλοποιεί τις κατάλληλες ενέργειες που έχουμε ορίσει εμείς την μηχανή γραφικών να κάνει με το πάτημα του κουμπιού αυτού.

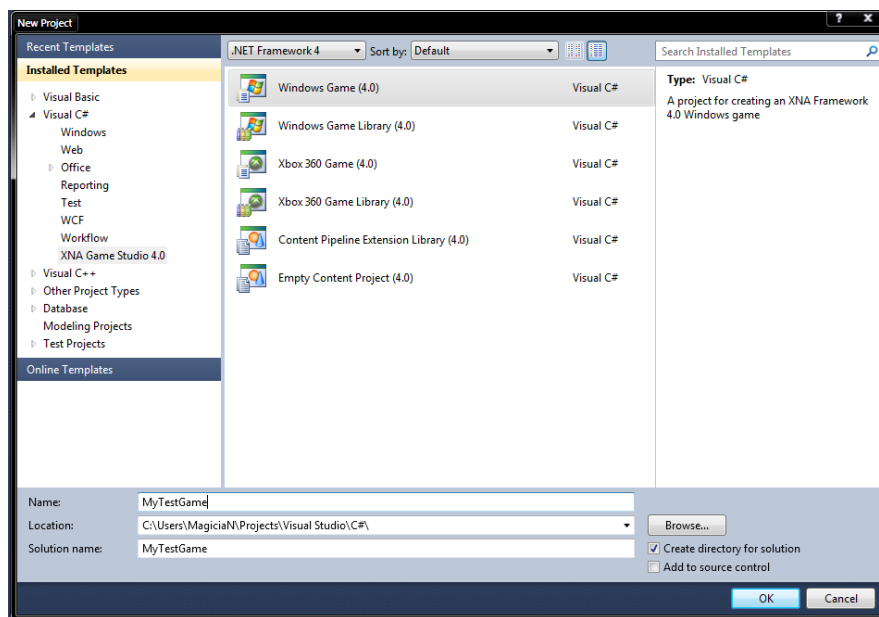
Όπως θα δούμε και αργότερα η μηχανή γραφικών Microsoft XNA έναντι των άλλων μηχανών γραφικών έχει κάποιες ιδιαιτερότητες. Η σημαντικότερη από αυτές είναι, όπως θα δούμε και στην συνέχεια, ότι η φόρτωση και η διαχείριση των αντικειμένων που χρησιμοποιούμε γίνεται με συγκεκριμένο τρόπο δηλαδή με **διασύνδεση αντικειμένων** (ContentPipeline), ο οποίος και αναγνωρίζεται σαν ξεχωριστό **project**. Με απλά λόγια, όταν δημιουργούμε ένα καινούριο παιχνίδι XNA μέσω του Microsoft Visual Studio 2010, αυτό στην πραγματικότητα δημιουργεί δύο project για να ασχοληθούμε και όχι ένα. Το πώς συνδέονται και επικοινωνούν αυτά τα δύο μεταξύ τους θα το δούμε στην συνέχεια. Προς το παρόν έχουμε στο μυαλό μας πως τα παιχνίδια χωρίζονται στα δύο. Το ένα project αφορά τις συναρτήσεις που υλοποιούν όλη τη λογική της μηχανής και το ονομάζουμε **“MainFunctions”** και το άλλο διαχειρίζεται τα αντικείμενα μας και ονομάζεται **“ContentManager”**. Παρακάτω αναλύουμε και τα δύο δημιουργώντας ένα καινούριο **project XNA** και αναλύοντάς το γραμμή-γραμμή. Αρχικά, θα προχωρήσουμε και θα δούμε πώς δημιουργούμε ένα καινούριο παιχνίδι με την μηχανή γραφικών Microsoft XNA και με το Microsoft Visual Studio 2010. Πριν, όμως, προχωρήσουμε στην δημιουργία παιχνιδιού, πρέπει πρώτα, αφού έχουμε εγκαταστήσει το **Microsoft Visual Studio 2010**, να εγκαταστήσουμε και το **Microsoft XNA Game Studio 4** το οποίο ενσωματώνεται αυτόματα στο **Microsoft Visual Studio 2010** και παρέχεται δωρεάν. Οπότε κατεβάζουμε το **Microsoft XNA Game Studio 4** από την ιστοσελίδα: [www.microsoft.com/download/en/details.aspx?id=23714](http://www.microsoft.com/download/en/details.aspx?id=23714) και αφού ολοκληρώσουμε την εγκατάσταση είμαστε έτοιμοι για την δημιουργία του πρώτου μας παιχνιδιού.



## 5.1 - Δημιουργία νέου παιχνιδιού

Αφού, λοιπόν, κατεβάσουμε και την μηχανή γραφικών **Microsoft XNA Game Studio 4**, ανοίγουμε την πλατφόρμα ανάπτυξης **Microsoft Visual Studio 2010**.

Όπως είδαμε και σε προηγούμενο κεφάλαιο θα δημιουργήσουμε ένα νέο **project** με την διαφορά ότι αυτήν την φορά θα επιλέξουμε από τα αριστερά **XNAGameStudio 4** και δεξιά επιλέγουμε το **WindowsGame 4.0**. Επιλέγουμε ένα όνομα και πατάμε **OK**. Η εικόνα που παρατίθεται παρακάτω θα μας βοηθήσει να κατανοήσουμε καλύτερα πώς θα γίνει η δημιουργία νέου παιχνιδιού.



Εικόνα 12. Δημιουργία καινούριου παιχνιδιού.

Μόλις πατήσουμε το κουμπί **OK**, θα διαπιστώσουμε αυτό που αναφέρθηκε πιο πριν. Στα δεξιά μας βλέπουμε ότι, ενώ εμείς έχουμε δημιουργήσει ένα **project**, στην πραγματικότητα είναι δύο. Και αυτό λόγω του ότι, όπως αναφέραμε και πριν, τα αντικείμενα που χρησιμοποιούμε για το παιχνίδι μας, όπως γραμματοσειρές και άλλα, διαχειρίζονται μέσω ενός άλλου **project**, του **ContentManager**, για το οποίο θα μιλήσουμε στην συνέχεια. Μπροστά μας, λοιπόν, αρχικά θα δούμε πολλά σχόλια και κώδικα. Ο κώδικας αυτός δημιουργείται αυτόματα για εμάς και μας βοηθά αρχικά να κατανοήσουμε για το πού θα βάλουμε τις διάφορες συναρτήσεις μας. Οι συναρτήσεις αυτές που δημιουργούνται υλοποιούν επακριβώς την λογική που αναφέραμε σε προηγούμενο κεφάλαιο των μηχανών γραφικών. Το τι κάνει κάθε μία συνάρτηση θα το δούμε στην συνέχεια.

## 5.2 - Main Functions (Project 1 of 2)

Αφού έχουμε δημιουργήσει ένα καινούριο παιχνίδι και έχουν δημιουργηθεί δύο **project** για εμάς καθώς και ο κατάλληλος κώδικας είμαστε έτοιμοι να εξηγήσουμε το τι κάνουν αυτές οι συναρτήσεις μία προς μία.

Αρχικά, βλέπουμε τον **constructor public Game1()**. Εδώ μέσα γράφουμε κώδικα για ό,τι αφορά την αρχική δημιουργία του παιχνιδιού. Ως προεπιλεγμένες ενέργειες έχουμε την δημιουργία ενός χειριστή γραφικών με την εντολή **graphics = newGraphicsDeviceManager(this)** και στην συνέχεια ορίζουμε τον φάκελο με τα αντικείμενα τα οποία χειρίζεται ο **contentmanager** με την εντολή **Content.RootDirectory = "Content"**.

Επίσης, εδώ μέσα μπορούμε να ορίσουμε και άλλες ιδιότητες οι οποίες κατά την διάρκεια ζωής του παιχνιδιού, δηλαδή μέχρι να γίνει έξοδος από αυτό, δεν θα αλλάξουν. Για παράδειγμα, μπορούμε να δηλώσουμε μία φόρμα την οποία θα χειρίζεται η μηχανή γραφικών μας, το οποίο και θα το δούμε στην συνέχεια.

Προχωρώντας, έχουμε την συνάρτηση **Initialize()** η οποία, επίσης, τρέχει μία φορά κατά την εκκίνηση του παιχνιδιού. Η διαφορά της από τον **constructor** δεν είναι σημαντική, απλώς χρησιμοποιείται για διαφορετική δουλειά και μας βοηθά έτσι να διαχωρίσουμε τον κώδικά μας και να είναι πιο ευανάγνωστος. Εδώ μέσα θα μπορούσαμε να ορίσουμε το αρχικό μέγεθος του παραθύρου, εάν είναι σε πλήρη οθόνη η λειτουργία παραθύρου και άλλα. Παρότι αυτά τα δύο που αναφέρθηκαν πριν μπορούν να αλλάξουν κατά την διάρκεια ζωής του παιχνιδιού εμείς παρόλ' αυτά τα ορίζουμε στην αρχή, στην συνάρτηση **Initialize()**, έτσι ώστε να μπορούμε εύκολα να δούμε με τι ιδιότητες γίνεται η εκκίνηση του παιχνιδιού μας.

Στην συνέχεια, έχουμε την συνάρτηση **LoadContent()**. Σε αυτό το σημείο μπορούμε να φορτώσουμε τα αντικείμενα που θα χρησιμοποιήσουμε στην συνέχεια στην μνήμη. Η συνάρτηση αυτή, βεβαίως, τρέχει μία φορά κατά την εκκίνηση του παιχνιδιού και κάποιος θα ρωτούσε το αυτονόητο: «Γιατί δεν ορίζουμε τα αντικείμενα μας στην συνάρτηση Constructor ή στην συνάρτηση Initialize;» Η απάντηση είναι απλή: Η διαφορά αυτής της συνάρτησης με τις δύο αρχικές είναι ότι ενώ τρέχει μία φορά στην εκκίνηση του παιχνιδιού, στην συνέχεια εμείς μπορούμε με ένα **signal** ή **event** να ξανακαλέσουμε αυτήν την συνάρτηση και να ξαναφορτώσουμε εκ νέου αντικείμενα, το γνωστό σε όλους του λάτρεις παιχνιδιών **Loading** όταν αλλάζουμε πίστα στα παιχνίδια.

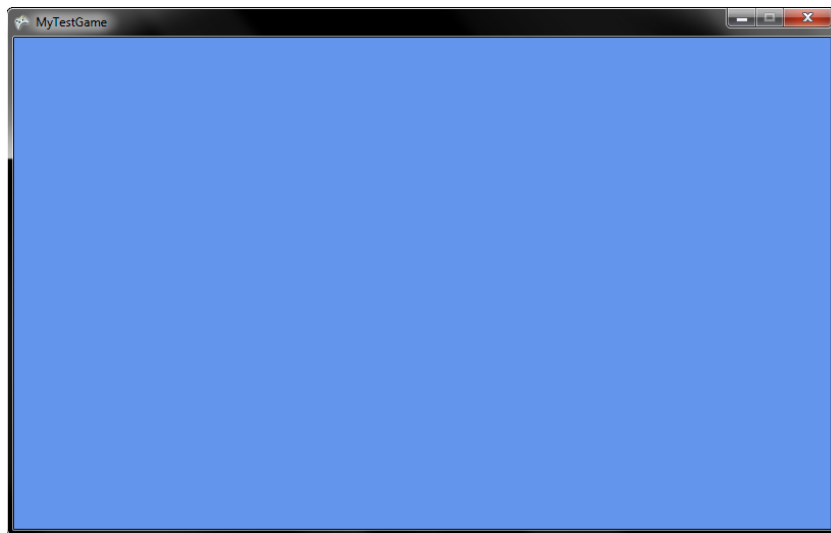
Για να καλέσουμε όμως εκ νέου την συνάρτηση **LoadContent()** και να φορτώσουμε καινούρια αντικείμενα στην μνήμη πρέπει να αδειάσουμε την μνήμη από τα αντικείμενα τα οποία και δεν θα χρησιμοποιηθούν στην συνέχεια. Αυτή την δουλειά κάνει ακριβώς η επόμενη συνάρτηση **UnloadContent()**, η οποία αδειάζει την μνήμη μας από αντικείμενα τα οποία δεν θα χρησιμοποιήσουμε στην συνέχεια.

Έπειτα, έρχονται η οι δύο σημαντικότερες συναρτήσεις για την διάρκεια ζωής του παιχνιδιού. Η πρώτη συνάρτηση είναι η πολύ γνωστή συνάρτηση **Update(GameTimegameTime)**, η οποία παίρνει ως όρισμα τον χρόνο και αυτό γιατί αλλάζει ανάλογα με αυτόν. Μέσα εδώ ορίζουμε πράγματα τα οποία πρέπει να ελέγχονται όσο το παιχνίδι τρέχει. Για παράδειγμα εάν πατήσαμε κάποιο κουμπί στο πληκτρολόγιο, ή εάν κουνήσαμε το ποντίκι. Επίσης, εάν, για παράδειγμα, είχαμε το παιχνίδι **asteroids**, θα ήταν σημαντικό να ελέγξουμε εάν το σκάφος μας έχει δεχθεί εχθρικά πυρά και να αφαιρέσουμε μία ζωή από τον παίχτη και άλλα.

Η δεύτερη συνάρτηση είναι η συνάρτηση **Draw(GameTimegameTime)**, η οποία επίσης παίρνει ως όρισμα τον χρόνο. Αυτή η συνάρτηση το μόνο που κάνει είναι να ζωγραφίζει αντικείμενα και τρίγωνα στην οθόνη μας. Δεν ελέγχει πότε δύο αντικείμενα συγκρούονται μεταξύ τους ή κάτι τέτοιο. Θα δούμε παραδείγματα με αυτή τη συνάρτηση στην συνέχεια.

Τελικά, διαπιστώνουμε ότι οι δύο τελευταίες συναρτήσεις τρέχουν συνέχεια μέχρι να τερματίσει το παιχνίδι. Βέβαια, έχουν σημαντικές διαφορές μεταξύ τους και δεν μπορούμε να τις χρησιμοποιήσουμε για διαφορετικό λόγο από αυτόν που έχουν οριστεί. Για παράδειγμα, σκεφτείτε ότι η συνάρτηση **Draw** τρέχει ακριβώς εξήντα φορές το δευτερόλεπτο, ενώ πριν τρέξει αυτή πρέπει πρώτα να τρέξει η συνάρτηση **Update** πολλές φορές παραπάνω πριν κληθεί η **Draw**.

Μέσα στην συνάρτηση **Draw** υπάρχει μία εντολή που ονομάζεται: **GraphicsDevice.Clear(Color.CornflowerBlue)**. Αυτή η συνάρτηση μας βοηθά να “καθαρίσουμε” το background της μηχανής γραφικών. Εάν “παίξουμε” λίγο με αυτήν την συνάρτηση ώστε να ορίσουμε το χρώμα με το οποίο θα “καθαρίζει” το background, δεν έχουμε παρά να πούμε κάτι τέτοιο: **GraphicsDevice.Clear(Color.AntiqueWhite)**. Θα δούμε ότι το χρώμα θα αλλάξει. Εάν όλα ακολουθήθηκαν βήμα βήμα, όταν πατήσουμε “Rungame”, δηλαδή από το μενού “Debug->StartDebugging”, θα δούμε το αρχικό παράθυρο της μηχανής γραφικών που σχεδίασε ο αυτόματος κώδικας.



**Εικόνα 13.** Το αρχικό παράθυρο του XNA.

Αφού αναφέραμε επιγραμματικά τις κύριες συναρτήσεις της μηχανής γραφικών **XNA** στην συνέχεια θα δούμε τον διαχειριστή αντικειμένων **ContentManager**.

### 5.3 - Content Manager (Project 2 of 2)

Ο διαχειριστής περιεχομένου (ContentManager), συχνά αναφέρεται και ως “πλαίσιο επεξεργασίας περιεχομένου” και έχει ως σκοπό την φόρτωση στην μνήμη διαφόρων αντικειμένων και την μετέπειτα διαχείριση τους.

Αρχικά, αν και έχουν γίνει αρκετές αναφορές για τον επεξεργαστή περιεχομένου, δεν είναι δυνατόν να κατανοήσουμε αυτήν την τεχνολογία και το τι ακριβώς κάνει, και η οποία, όπως είπαμε, υλοποιείται και αναγνωρίζεται ως ένα ξεχωριστό **project** στο παιχνίδι μας. Ας προσπαθήσουμε λοιπόν παρακάτω να εξηγήσουμε επακριβώς τον διαχειριστή περιεχομένου της μηχανής γραφικών **Microsoft XNA**.

Το περιεχόμενο που υπάρχει στα σημερινά παιχνίδια, (μοντέλα, ηχητικά εφέ, μουσική και άλλα) αποτελούν το μεγαλύτερο και σημαντικότερο μέρος τους. Το να καταφέρει κάποιος να εισάγει περιεχόμενο μέσα σε ένα παιχνίδι δεν είναι απλή διαδικασία αντιθέτως είναι πολύπλοκη. Συχνά οι προγραμματιστές παιχνιδιών βρίσκονται αντιμέτωποι με τα ίδια επαναλαμβανόμενα προβλήματα: Σχεδιάζουν κάποια αντικείμενα αρχικά, με διάφορα προγράμματα όπως για παράδειγμα το **3DSStudioMax**, και έπειτα δεν είναι εύκολο να βρουν ένα τρόπο ή το κατάλληλο εργαλείο ώστε να εξαγάγουν το αντικείμενο αυτό σε μορφή αναγνωρίσιμη από την μηχανή γραφικών που χρησιμοποιούν. Επίσης, λόγω μεγάλης πληθώρας τέτοιων προγραμμάτων σχεδίασης τρισδιάστατων και μη γραφικών, η οποία και αυξάνει την πολυπλοκότητα, υπάρχουν ενδοιασμοί από τους προγραμματιστές για το αν τα εξαγόμενα αντικείμενα αυτά - ακόμα και αν βρεθεί τρόπος εξαγωγής τους - θα εμφανιστούν σωστά μέσα στο παιχνίδι τους. Με άλλα λόγια, ακόμα και αν βρεθεί ένας κατάλληλος τρόπος ώστε να γίνει εισαγωγή ενός μοντέλου μέσα στην μηχανή γραφικών, πρέπει να το επεξεργαστούμε κατάλληλα, έτσι ώστε να εμφανιστεί σωστά μέσα στο παιχνίδι μας. Το προηγούμενο αυξάνει την πολυπλοκότητα ανάπτυξης εφαρμογών παιχνιδιών και δαπανείται πολύτιμος χρόνος για αυτήν την διαδικασία. Στην πραγματικότητα, υπάρχουν χιλιάδες άνθρωποι οι οποίοι και ασχολούνται μόνο με όλα τα προαναφερθέντα και προσπαθούν να βρουν τρόπους έτσι ώστε να μειωθεί αυτή η πολυπλοκότητα.

Η διαδικασία αυτή, όμως, με την μηχανή γραφικών **Microsoft XNA**, είναι διαφορετική. Καταρχήν, ο επεξεργαστής περιεχομένων είναι πιο απλός στην χρήση του, υψηλά παραμετροποιήσιμος και επεκτάσιμος. Τελικά, ο επεξεργαστής περιεχομένου κάνει ό,τι ακριβώς λέει και το όνομά του, διευκολύνοντάς μας και γλιτώνοντάς μας από τη

σπατάλη χρόνου με ενασχόληση για λεπτομέρειες χαμηλού επιπέδου. Έτσι, έχουμε περισσότερο χρόνο να ασχοληθούμε με την ανάπτυξη του παιχνιδιού, παρά με λεπτομέρειες.

Ένα από τα πρώτα πράγματα που θα ασχοληθούμε αρχικά όσον αφορά τον χειριστή αντικειμένων, είναι το πώς τα αντικείμενα αυτά χρησιμοποιούνται μέσα σε ένα παιχνίδι. Όπως είδαμε, τα αντικείμενα χειρίζονται μέσα στο **MicrosoftVisualStudio 2010**, έτσι όπως ακριβώς εμείς εισάγουμε καινούρια αρχεία κώδικα μέσα στο **project** μας, έτσι μπορούμε να εισάγουμε και καινούρια αντικείμενα. Αυτό τελικά μας βοηθά να κρατάμε κώδικα και αντικείμενα μαζί και να μην υπάρχει σύγχυση. Αυτό που μας ενδιαφέρει πάντως είναι ο τύπος των αρχείων που θα εισάγουμε και όχι από πού προήλθαν (από πιο πρόγραμμα).

Οι καταλήξεις που υποστηρίζει το **MicrosoftXNAGameStudio** συνοψίζονται στον παρακάτω πίνακα.

3D Αρχεία	2D Αρχεία	Αρχεία εφφέ	Αρχεία ήχου
.FBX .X	.DDS .BMP .JPG .PNG .TGA	.FX	.XAP

**Εικόνα 14.** Αρχεία που υποστηρίζονται από την μηχανή γραφικών XNA.

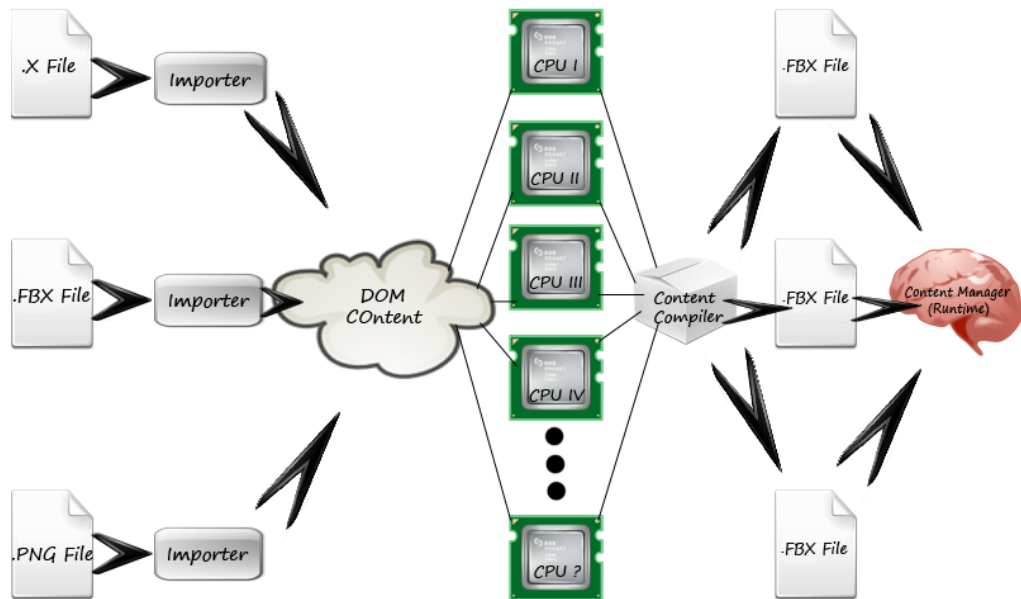
Αν και οι περισσότερες καταλήξεις είναι λίγο πολύ γνωστές σε όλους μας, αξίζει παρόλ' αυτά να συζητήσουμε για την κατάληξη **.FBX**. Αυτή η κατάληξη μπορεί να εξαχθεί από το **3DSMaxStudio**, το οποίο είναι ένα επαγγελματικό πρόγραμμα δημιουργίας γραφικών και χρησιμοποιείται κατά κόρον στις βιομηχανίες παιχνιδιών. Τα πλεονεκτήματα είναι πολλά. Μπορούμε μέσω του **3DSMaxStudio** να ορίσουμε για παράδειγμα για το πώς θα γυρνάει το αντικείμενο, να ορίσουμε δηλαδή σημεία αναφοράς τα οποία και θα είναι αναγνωρίσιμα και επεξεργάσιμα από την μηχανή γραφικών **Microsoft XNA**. Έτσι είναι δυνατή η επικοινωνία **3DSMaxStudio** και **Microsoft XNA**, μέσω του **ContentPipeline**, τον χειριστή δηλαδή αντικειμένων. Θα δούμε και αργότερα παραδείγματα για το πώς είναι δυνατόν να εισάγουμε τα δικά μας αντικείμενα μέσα στο **project** μας.

Στην συνέχεια, ας ασχοληθούμε σε βάθος με τον επεξεργαστή αντικειμένων. Όταν ο επεξεργαστής αντικειμένων εισάγει τα αντικείμενα τα οποία εμείς του έχουμε ορίσει, αυτά πλέον υπάρχουν μέσα στη συλλογή αντικειμένων **DOM**. Ο όρος αυτός (**DOM**) χρησιμοποιείται για να δηλώσει μία συλλογή αντικειμένων, όπως στην περίπτωση των αρχείων **XML**. Τα αρχεία έπειτα αυτά μετατρέπονται σε γνωστές μορφές από το περιβάλλον, και τα οποία είναι όμοια μεταξύ τους, ανεξάρτητα από ποιο πρόγραμμα σχεδίασης προήλθαν. Από εκεί και πέρα ο επεξεργαστής αντικειμένων είναι υπεύθυνος να πάρει τα περιεχόμενα μορφής DOM και να τα μετατρέψει σε αντικείμενα τα οποία θα είναι ικανά να τρέχουν στο παιχνίδι μας. Αυτό το αντικείμενο μπορεί να είναι είτε πολύ απλό, για παράδειγμα ένα **Texture 2D**, είτε πολύ πολύπλοκο, δηλαδή ένα μοντέλο 3D.

Παρακάτω παρατίθεται ένα παράδειγμα για να γίνουν όλα τα προηγούμενα κατανοητά. Σκεφτείτε να σχεδιάζετε ένα παιχνίδι αγώνων με καρτ. Ένα από τα σημαντικότερα μέρη θα ήταν η πίστα του παιχνιδιού. Σε αυτήν θα έπρεπε να ορίσουμε την διαδρομή, δηλαδή πώς πάει ο δρόμος, τις στροφές, την έναρξη τον τερματισμό και άλλα. Αντί, λοιπόν, να σπάσουμε τον δρόμο αυτόν σε πολλά μοντέλα και να ορίσουμε αυτά τα σημεία, θα μπορούσαμε να χρησιμοποιήσουμε τον επεξεργαστή περιεχομένου έτσι ώστε να κάνει όλες αυτές τις δουλειές για εμάς.

Ένα άλλο προτέρημα του επεξεργαστή περιεχομένου είναι η μεταφερσιμότητα του ανάμεσα στις πλατφόρμες ανάπτυξης παιχνιδιών με τη μηχανή γραφικών **MicrosoftXNA**. Έτσι, κατασκευάζοντας παιχνίδια που χρησιμοποιούν αντικείμενα που έχουν την ίδια λογική, θα είχαμε παρά να ασχοληθούμε με την βελτίωση του κώδικα και όχι να ξανά εφευρίσκουμε τον τροχό από την αρχή. Επίσης, λόγω του ότι το περιεχόμενο μας γίνεται **build** μαζί με το παιχνίδι μας, τα αντικείμενα αυτά μετατρέπονται σε σειρές δυαδικές (0 και 1) και έτσι κερδίζουμε και σε απόδοση (Αρχεία **.XNB**).

Παρακάτω παρατίθεται μία εικόνα η οποία θα μας βοηθήσει να κατανοήσουμε το πώς λειτουργεί ο επεξεργαστής περιεχομένου.



**Εικόνα 15.** Ο επεξεργαστής περιεχομένου.

Στην προηγούμενη εικόνα βλέπουμε πως ο επεξεργαστής περιεχομένου λαμβάνει αρχεία διαφόρων τύπων όπως .PNG, .TGA και άλλα, τα εισάγει και τα τοποθετεί ως περιεχόμενο DOM και τέλος τα εξάγει σε αρχεία δυαδικά με τη μορφή .FBX. Με αυτόν τον τρόπο η επεξεργασία των αρχείων αυτών γίνεται ταχύτατα. Τέλος, πρέπει να σημειώσουμε ότι αυτός ο επεξεργαστής περιεχομένου αποτελεί καινοτομία για την μηχανή γραφικών **MicrosoftXNA**, διότι δεν συναντάται σε άλλες μηχανές γραφικών, ή τουλάχιστον υπερτερεί σε πλεονεκτήματα έναντι των υπολοίπων μηχανών.





## 5.4 - Βασικά στοιχεία δυσδιάστατων γραφικών

Στο προηγούμενο κεφάλαιο έγινε μία εκτενής αναφορά στον διαχειριστή περιεχομένου της μηχανής γραφικών **Microsoft XNA**. Σε αυτό το κεφάλαιο θα δούμε τα βασικά στοιχεία των δυσδιάστατων γραφικών, δηλαδή πώς μπορούμε να ζωγραφίσουμε ένα αντικείμενο το οποίο έχουμε εισάγει με τον επεξεργαστή περιεχομένου στην μηχανή γραφικών, πώς μπορούμε να γράψουμε κείμενο στην οθόνη και άλλα. Τέλος, πριν προχωρήσουμε στα γραφικά τριών διαστάσεων, θα ασχοληθούμε με τα κινούμενα αντικείμενα.

Τα δυσδιάστατα γραφικά αποτελούν μεγάλο κομμάτι της βιομηχανίας γραφικών. Σε μερικές περιπτώσεις, λόγω ανεπαρκών πόρων συστήματος, όπως ανεπαρκή μνήμη ή μικρή επεξεργαστική ισχύ, προτιμώνται από τις βιομηχανίες αυτά τα δυσδιάστατα γραφικά, λόγω μικρής χρησιμοποίησης πόρων συστήματος. Επίσης, σε μερικές γλώσσες προγραμματισμού, όπως την **Java**, επειδή υπάρχει υψηλή ανάγκη για μεταφερσιμότητα, χρησιμοποιούνται δυσδιάστατα γραφικά, λόγω μεγάλης ποικιλίας καρτών γραφικών, μνημών και άλλα που χρησιμοποιούν τα διάφορα συστήματα όπου και εκτελείται το πρόγραμμα.

Δεν μπορούμε να πούμε σε καμία περίπτωση ότι τα παιχνίδια σχεδιασμένα με δυσδιάστατα γραφικά είναι κατωτέρας τάξης από αυτά με τις τρεις διαστάσεις. Υπάρχουν περιπτώσεις όπου παιχνίδια που είχαν μεγάλες πωλήσεις σχεδιασμένα σε δύο διαστάσεις, επανασχεδιάστηκαν σε τρεις διαστάσεις και το αποτέλεσμα ήταν να πέσουν οι πωλήσεις. Χαρακτηριστικό παράδειγμα είναι το παιχνίδι “Worms-Armageddon”, το οποίο και διαδέχθηκε το παιχνίδι “Worms 3D”, το οποίο και αποκόμισε αρνητικά σχόλια.

Παρακάτω παρατίθεται μία εικόνα η οποία δείχνει την βαθμολογία των δύο παιχνιδιών από την πλέον αξιόπιστη σελίδα για παιχνίδια “[www.gamespot.com](http://www.gamespot.com)”.

<p><b>Worms Armageddon (PC)</b></p> <p>Like 1 +1 0 Tweet 0</p>  <p>While this game may look cute, it is in fact as sophisticated and enjoyable as the very best strategy games out there.</p>	<p>GameSpot Score</p> <p><b>9.1</b></p> <p>Editors' Choice</p> <p>About the rating system »</p>	<p>Critic Score 17 reviews <b>8.8</b></p> <p>User Score 2,732 votes <b>9.0</b></p> <p>Your Score slide to rate N/A</p> <p>Review the Game</p>
<p><b>Worms 3D (PC)</b></p> <p>Like 4 +1 0 Tweet 0</p>  <p>Worms 3D stands up as a cute, quirky little strategy game that has some excellent multiplayer capacity.</p>	<p>GameSpot Score</p> <p><b>7.8</b></p> <p>good</p> <p>About the rating system »</p>	<p>Critic Score 15 reviews <b>7.6</b></p> <p>User Score 885 votes <b>7.9</b></p> <p>Your Score slide to rate N/A</p> <p>Review the Game</p>

**Εικόνα 16.** Οι βαθμολογίες των παιχνιδιών Worms-Armageddon και Worms 3D.

Πηγή: [www.gamespot.com](http://www.gamespot.com)

Τα παιχνίδια Worms-Armageddon και Worms 3D είναι χαρακτηριστικό παράδειγμα όπου καταρρίπτεται ο μύθος του “φαίνεσθε”. Βλέπουμε ότι τα γραφικά, όσο εντυπωσιακά και να είναι δεν παίζουν τον κύριο λόγο. Παρακάτω θα δούμε στιγμιότυπα από τα δύο αυτά παιχνίδια.



**Εικόνα 17.** Στιγμιότυπα των δύο παιχνιδιών (Αριστερά Worms 2D, δεξιά Worms 3D).

Η εμπειρία που έχουν αποκομίσει οι βιομηχανίες παιχνιδιών όλα αυτά τα χρόνια, τις ανάγκασε να μην επενδύουν τόσο πολύ επάνω στα εντυπωσιακά εφέ και τα ζωηρά χρώματα. Το σημαντικότερο ρόλο θα λέγαμε για να κριθεί ένα παιχνίδι ως καλό ή μη, είναι το σενάριο, η διαδραστικότητα που προσφέρεται στον χρήστη, το κατά πόσο δίνεται η ελευθερία στον χρήστη να επέμβει σε πράγματα του χώρου και κατά πόσο αυτά τα πράγματα αντιδρούν με τις κινήσεις του και τέλος η ρεαλιστικότητα που προσφέρεται στον χρήστη. Πλέον, θα λέγαμε πως τα παιχνίδια έχουν αποκτήσει ένα είδος “χολιγουντιανού” χαρακτήρα.

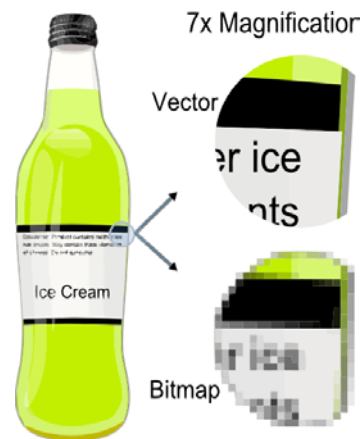
Τα στοιχεία που παρουσιάζονται σε αυτό το κεφάλαιο αφορούν τα δυσδιάστατα γραφικά βεβαίως, με τα οποία και πρέπει να ασχοληθεί κανείς εάν επιθυμεί να συνεχίσει την διατριβή του με τα γραφικά των τριών διαστάσεων. Επίσης γνωρίζοντας τις βασικές έννοιες για την κατασκευή παιχνιδιών δύο διαστάσεων, η μεταπήδηση σε γραφικά τριών διαστάσεων γίνεται πιο ομαλά.

Ας προχωρήσουμε, λοιπόν, και ας δούμε μερικά πράγματα για τα γραφικά δύο διαστάσεων και στη συνέχεια μερικά παραδείγματα.

Όταν μιλάμε για γραφικά δύο διαστάσεων στην επιστήμη της πληροφορικής προφανώς αναφερόμαστε σε αντικείμενα που μπορούν να παρουσιαστούν στον χώρο με την χρήση δύο αξόνων του καρτεσιανού συστήματος. Μερικά αντικείμενα από αυτά είναι οι γραμματοσειρές, δυσδιάστατα μοντέλα και εικόνες. Έχει υπολογιστεί ότι για ένα κείμενο που παρουσιάζεται ως εικόνα σε υπολογιστή, εάν χρησιμοποιηθούν για την αναπαράστασή του δυσδιάστατα γραφικά το μέγεθος θα είναι το ένα τοις χιλίοις! Επίσης, προτιμάται η μεταφορά κειμένων σε αυτή τη μορφή λόγω των τόσο διαφορετικών οθονών και αναλύσεων που υπάρχουν στην αγορά. Με αυτόν τον τρόπο τα κείμενά μας είναι πιο ευέλικτα και έτσι μπορούμε να τα διαβάσουμε από πάρα πολλές συσκευές.

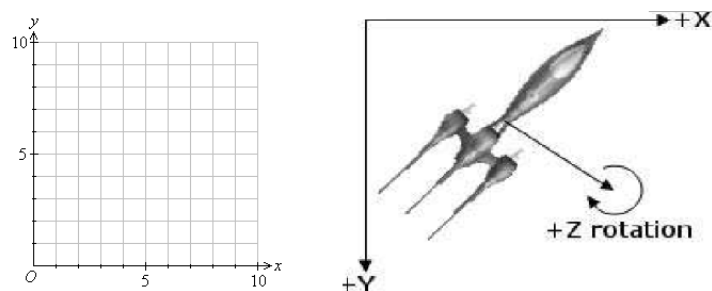
Τα δυσδιάστατα γραφικά των υπολογιστών πρωτοεμφανίστηκαν γύρω στο 1950, βασισμένα στα διανυσματικά γραφικά. Τα διανυσματικά γραφικά χρησιμοποιούν σημεία, γραμμές, καμπύλες και διάφορα σχήματα πολυγώνων τα οποία, βασισμένα σε μαθηματικές εκφράσεις και εξισώσεις, χρησιμοποιούνται για την εμφάνιση των εικόνων. Έπειτα, εκτοπίστηκαν σε μεγάλο βαθμό από τα γραφικά bitmap ή αλλιώς γραφικά raster, τα οποία παρουσιάζουν τα γραφικά ως εικονοστοιχεία (**pixels**), λόγω της ανάγκης για μικρότερη επεξεργαστική ισχύ. Παρόλ' αυτά, λόγω της υψηλής τους ποιότητας σε επαγγελματικές εφαρμογές χρησιμοποιούνται τα διανυσματικά γραφικά. Αν και τα διανυσματικά γραφικά χρησιμοποιούν τα εικονοστοιχεία για να εμφανιστούν στην οθόνη, έχουν σημαντικές διαφορές με τα γραφικά raster/bitmap. Τα γραφικά bitmap χρησιμοποιούν τα εικονοστοιχεία για να αποθηκεύσουν τις τιμές του κόκκινου, πράσινου και μπλε που περιέχουν, σε ένα εύρος 0-255, ενώ τα διανυσματικά γραφικά χρησιμοποιούν τα εικονοστοιχεία για να αποθηκεύσουν τις διάφορες εξισώσεις που αναφέραμε πιο πριν έτσι ώστε να επιτύχουμε το επιθυμητό αποτέλεσμα. Σε μερικές

περιπτώσεις χρησιμοποιούνται ταυτόχρονα και οι δύο μέθοδοι. Παρακάτω παρατίθεται μία εικόνα που παρουσιάζει τις δύο αυτές μεθόδους:



**Εικόνα 18.** Οι μέθοδοι παρουσίασης γραφικών.

Όπως είπαμε και πριν, τα δυσδιάστατα γραφικά χρησιμοποιούν τον δυσδιάστατο καρτεσιανό άξονα ώστε να “ζωγραφιστούν”. Από τα μαθηματικά θα ξέρουμε ότι το κέντρο των αξόνων είναι στην μέση των αξόνων όπου εκεί τέμνονται. Στην μηχανή γραφικών **MicrosoftXNA** τα πράγματα είναι κάπως διαφορετικά. Καταρχήν δεν υπάρχουν αρνητικές τιμές και οι άξονες ξεκινούν από το ίδιο σημείο, το πάνω αριστερά και συνεχίζουν να επεκτείνονται δεξιά και κάτω. Το σημείο “μηδέν” είναι διαφορετική έννοια για τον καρτεσιανό κλασσικό άξονα και άλλη για τον καρτεσιανό άξονα της μηχανής γραφικών **MicrosoftXNA**. Παρακάτω παρατίθεται μία εικόνα, ώστε να γίνουν κατανοητά όλα τα προηγούμενα.



**Εικόνα 19.** Οι δύο άξονες (αριστερά ο καρτεσιανός, δεξιά ο άξονας που χρησιμοποιεί το XNA).

Οπότε εάν τοποθετήσουμε ένα δυσδιάστατο γραφικό στην μηχανή γραφικών **Microsoft XNA** στο σημείο (0,0), αυτό θα εμφανιστεί στην πάνω αριστερή γωνία του παραθύρου.

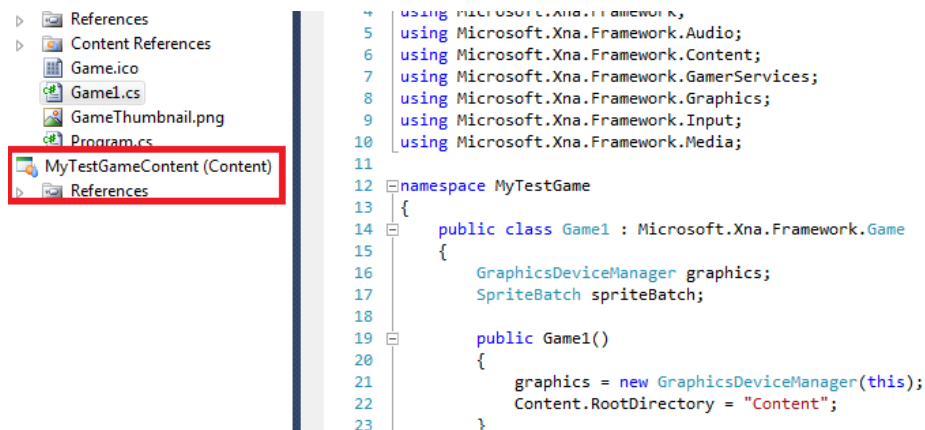
Όπως είδαμε και στο προηγούμενο κεφάλαιο η μηχανή γραφικών **Microsoft XNA** υποστηρίζει την εισαγωγή εικόνων με επεκτάσεις .bmp, .dds, .dib, .hdr, .jpg, .pfm, .png, .ppm με τον επεξεργαστή περιεχομένου. Συνήθως αυτές οι επεκτάσεις είναι

υπεραρκετές, παρόλ' αυτά, εάν τα αντικείμενα που χρησιμοποιούμε για το παιχνίδι μας είναι σε διαφορετική κατάληξη, υπάρχουν πολλοί τρόποι και διάφορα προγράμματα που θα μας βοηθήσουν να κάνουμε την μετατροπή σε κατάληξη συμβατή με την μηχανή γραφικών. Προσέξτε όμως ότι οι εικόνες που θα χρησιμοποιηθούν δεν πρέπει να είναι πολύ μεγάλες διότι αυτό συνεπάγεται μεγαλύτερη ανάγκη μνήμης για το παιχνίδι. Επίσης, για να τρέχει το παιχνίδι μας με την μέγιστη απόδοση, βεβαιωθείτε ότι τα αντικείμενά σας θα έχουν ανάλυση **δυνάμεις του δυο**. Για παράδειγμα : 32x32, 64x64, 128x128 και πάει λέγοντας. Μερικές δυνατότητες των εικόνων, όπως για παράδειγμα αυτό της **διαφάνειας** στις .png εικόνες, μεταφέρονται αυτόματα στην μηχανή γραφικών **Microsoft XNA**.

Ενώ έχουμε φορτώσει τα αντικείμενα μας μέσα στην μηχανή γραφικών, θα πρέπει να χρησιμοποιήσουμε μία **βούρτσα**, ώστε να τα ζωγραφίσουμε. Αυτή η βούρτσα είναι το αντικείμενο **SpriteBatch** και χρησιμοποιήθηκε πολλές φορές μέσα σε αυτήν την εργασία. Αυτό που πρέπει να πούμε πριν δούμε τα παραδείγματα είναι ότι επειδή υπάρχει ασυμβατότητα μεταξύ των τρισδιάστατων και δυσδιάστατων γραφικών, θα πρέπει όταν ζωγραφίζουμε δυσδιάστατα αντικείμενα να το δηλώνουμε στην μηχανή μας, καθώς και όταν τελειώνουμε με τη “ζωγραφική”.

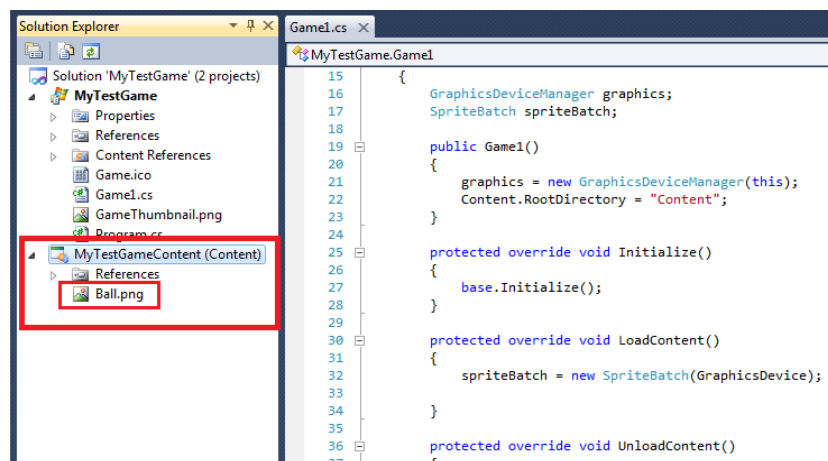
## 5.5 - Εισαγωγή 2D Texture

Αφού καλύφθηκαν αρκετά θεωρητικά, είναι καιρός να γίνει εφαρμογή της θεωρίας αυτής στην πράξη. Ανοίγουμε, λοιπόν, το **VisualStudio 2010** και δημιουργούμε ένα καινούριο **projectXNA** για **Windows**, όπως είδαμε και σε προηγούμενο κεφάλαιο. Αριστερά της οθόνης θα δούμε ότι υπάρχουν δύο **projects**. Εμείς θα ασχοληθούμε προς το παρόν με το δεύτερο που έχει την σήμανση **“Content”**, όπως φαίνεται στην παρακάτω εικόνα.



Εικόνα 19. Ο διαχειριστής περιεχομένου

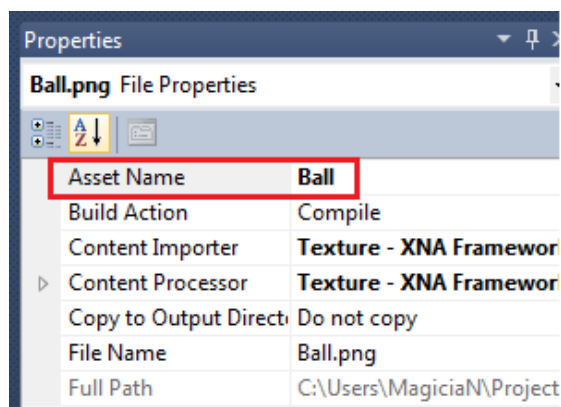
Θα εισάγουμε, λοιπόν, μία καινούρια εικόνα. Για αυτό τον λόγο κάνουμε δεξί κλικ επάνω στον διαχειριστή περιεχομένου και στο μενού που εμφανίζεται επιλέγουμε: **“Add->ExistingItem”**. Στο παράθυρο επιλογής αρχείου που εμφανίζεται επιλέγουμε το αρχείο που θέλουμε να εισάγουμε. Μόλις τελειώσουμε θα πρέπει να έχουμε κάτι σαν το παρακάτω:



Εικόνα 20. Το αντικείμενο αποθηκευμένο στον επεξεργαστή περιεχομένου.

Έχουμε φτάσει στο σημείο όπου το περιεχόμενό μας έχει εισαχθεί στον διαχειριστή περιεχομένου. Μας μένει, λοιπόν, να πάμε στον κώδικα μας στο πρώτο **project** και να γράψουμε κώδικα ώστε να συνδέσουμε το αντικείμενο αυτό με ένα όνομα μεταβλητής

και έπειτα να το ζωγραφίσουμε. Πριν, όμως, προχωρήσουμε εκεί θα πρέπει να ξέρουμε το όνομα που αποθηκεύτηκε το αντικείμενο που μόλις εισάγαμε στον διαχειριστή περιεχομένου. Αυτό είναι εύκολο να το κάνουμε. Συνήθως έχει το όνομα του αρχείου αλλά για να σιγουρευτούμε κάνουμε κλικ στο αντικείμενο που μόλις εισάγαμε και βλέπουμε δεξιά τις ιδιότητές του. Θα εμφανιστεί κάτι σαν το παρακάτω:



Εικόνα 21. Το όνομα του αντικειμένου που μόλις εισάγαμε.

Αυτό που μας ενδιαφέρει είναι η ιδιότητα “AssetName”. Οπότε για να συνδέσουμε το αντικείμενο μας στον κώδικα με μία μεταβλητή θα πρέπει να χρησιμοποιήσουμε αυτό το όνομα. Πάμε λοιπόν στον κώδικα για να δούμε πως θα μπορέσει να γίνει αυτό:

```
12 namespace MyTestGame
13 {
14     public class Game1 : Microsoft.Xna.Framework.Game
15     {
16         GraphicsDeviceManager graphics;
17         SpriteBatch spriteBatch;
18         Texture2D myBall;
19         Vector2 ballPosition;
20     }
```

Κώδικας 4. Δήλωση αρχικών μεταβλητών.

Βλέπουμε στον προηγούμενο κώδικα πως στην γραμμή 17 έχει δηλωθεί ένα αντικείμενο “βούρτσας” το οποίο και θα μας βοηθήσει να “ζωγραφίσουμε” το αντικείμενό μας. Στην γραμμή 18 έχει δηλωθεί ένα αντικείμενο **Texture2D** στο οποίο θα φορτώσουμε το αντικείμενό μας αργότερα. Στην συνέχεια στη γραμμή 19 δηλώνουμε ένα αντικείμενο διανύσματος **Vector2**, το οποίο κρατά την X και Y θέση του αντικειμένου, που θα μας βοηθήσει να τοποθετήσουμε το αντικείμενό μας σε ένα σημείο.

Στην συνέχεια πρέπει να γράψουμε κώδικα στην συνάρτηση **LoadContent()** της μηχανής γραφικών:

```

32     protected override void LoadContent()
33     {
34         spriteBatch = new SpriteBatch(GraphicsDevice);
35         myBall = Content.Load<Texture2D>("Ball");
36         ballPosition = new Vector2(0, 0);
37     }
38

```

**Κώδικας 5.** Κώδικας στην συνάρτηση LoadContent().

Στην γραμμή 34 αρχικοποιούμε την βούρτσα του παιχνιδιού, στην 35 φορτώνουμε στην μεταβλητή **myBall** ένα αντικείμενο τύπου **Texture2D** και μέσα στις παρενθέσεις γράφουμε το **AssetName** που αναφέραμε πιο πριν. Με αυτό τον τρόπο έχει συνδεθεί η μεταβλητή μας με το αντικείμενο του διαχειριστή περιεχομένου. Ορίζουμε επίσης στην γραμμή 36 την αρχική θέση εμφάνισης του αντικειμένου που είναι η θέση (0,0), η πάνω αριστερή γωνία για την μηχανή γραφικών **MicrosoftXNA**, όπως αναφέραμε σε προηγούμενο κεφάλαιο.

Δεν μένει τώρα πλέον να εμφανίσουμε, να ζωγραφίσουμε το αντικείμενο μας στην οθόνη της μηχανής γραφικών. Προκειμένου, λοιπόν, να ζωγραφίσουμε ένα αντικείμενο πρέπει να πάμε στην συνάρτηση **Draw()** και να γράψουμε τον εξής κώδικα:

```

49     protected override void Draw(GameTime gameTime)
50     {
51         GraphicsDevice.Clear(Color.CornflowerBlue);
52
53         spriteBatch.Begin();
54         spriteBatch.Draw(myBall, ballPosition, Color.White);
55         spriteBatch.End();
56
57         base.Draw(gameTime);
58     }

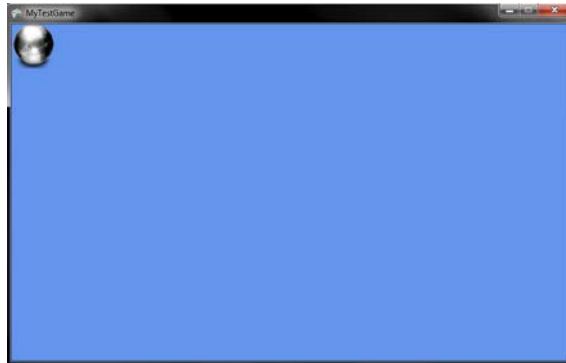
```

**Κώδικας 6.** Εμφάνιση αντικειμένου στην οθόνη.

Βλέπουμε πως στην γραμμή 53 δηλώνουμε ότι το σημείο αυτό είναι από όπου ξεκινούν να ζωγραφίζονται τα γραφικά των δύο διαστάσεων. Εάν έχουμε δέκα γραφικά δεν χρειάζεται να κάνουμε αυτή τη διαδικασία δέκα φορές. Απλώς πριν ξεκινήσουμε να ζωγραφίζουμε τα δυσδιάστατα γραφικά μας πρέπει να τοποθετούμε με αυτήν την δήλωση και όταν τελειώσουμε να την τερματίζουμε όπως κάνουμε και στη γραμμή 55. Στην γραμμή 54 καλούμε την συνάρτηση της βούρτσας **Draw** η οποία και έχει πολλές υπερφορτώσεις. Εμείς διαλέξαμε αυτήν που παίρνει ως όρισμα το όνομα της μεταβλητής στην οποία είναι συνδεδεμένο το αντικείμενο, την θέση του αντικειμένου και το χρώμα. Εάν επιλέξουμε το χρώμα **White**, όπως και κάναμε, το αντικείμενο θα



εμφανιστεί στην οθόνη του **XNA** όπως είναι, σε κάθε άλλη περίπτωση η μηχανή γραφικών θα ζωγραφίσει το αντικείμενο με βάση το χρώμα που έχουμε επιλέξει. Εάν τρέξουμε το παιχνίδι μας το αποτέλεσμα θα είναι περίπου το παρακάτω, ανάλογα βέβαια και με την εικόνα που έχουμε επιλέξει:



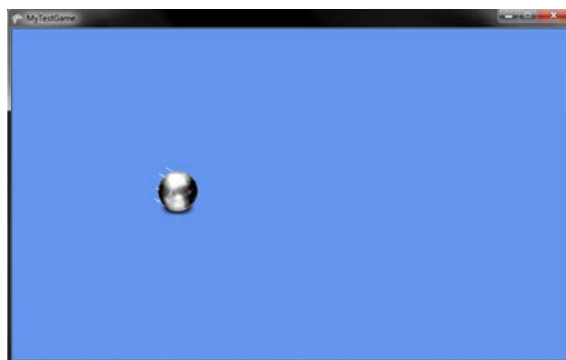
**Εικόνα 22.** Το αντικείμενο στην οθόνη της μηχανής MicrosoftXNA.

Παρατηρούμε πως το αντικείμενο παραμένει σταθερό στην πάνω αριστερή γωνία. Αυτό είναι λογικό εφόσον εμείς δεν του έχουμε ορίσει κίνηση. Για να το κάνουμε αυτό θα εισάγουμε κώδικα στην συνάρτηση **Update()**, όπου θα προσθέτουμε μία μονάδα σε κάθε ανανέωση της μηχανής γραφικών. Ο κώδικας παρατίθεται παρακάτω:

```
44 | protected override void Update(GameTime gameTime)  
45 | {  
46 |     base.Update(gameTime);  
47 |   
48 |     ballPosition.X += 1;  
49 |     ballPosition.Y += 1;  
50 | }  
51 |
```

**Κώδικας 7.** Εισαγωγή κώδικα για κίνηση του αντικειμένου.

Στην γραμμή 48 στο μέλος του αντικειμένου **ballPosition** “X” και στην γραμμή 49 στο μέλος του αντικειμένου **ballPosition** “Y”, προστίθεται από μία μονάδα στο καθένα. Το αποτέλεσμα που θα δούμε στην οθόνη είναι το αντικείμενό μας να κουνιέται προς την κάτω δεξιά γωνία:

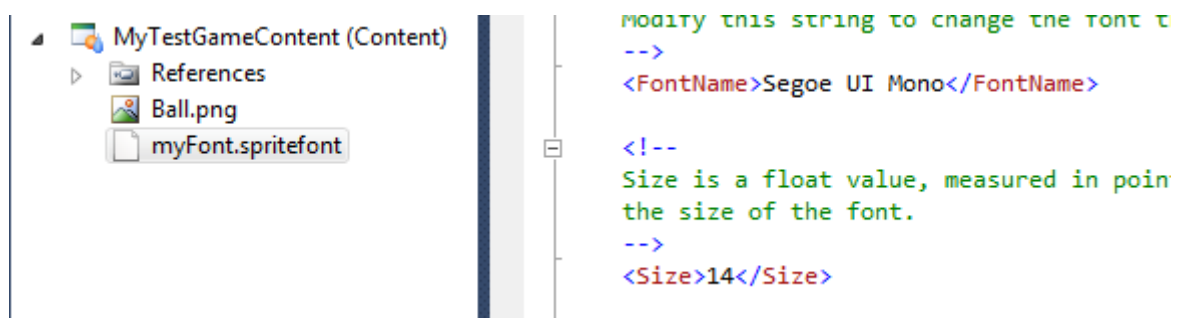


**Εικόνα 23.** Το κινούμενο αντικείμενο.

Εάν αφήσουμε το αντικείμενο να προχωρήσει μέχρι την δεξιά κάτω γωνία θα δούμε ότι θα φύγει εκτός οθόνης. Αυτό δεν σημαίνει πως χάθηκε, αλλά πως βγήκε εκτός ορίων. Αυτό μπορούμε να το ελέγξουμε με κώδικα ώστε να διαπιστώσουμε πότε το αντικείμενο φεύγει εκτός οθόνης και να αρχίσουμε να αφαιρούμε μονάδες από το X και το Y. Παρακάτω θα φορτώσουμε με την βοήθεια του διαχειριστή αντικειμένων ένα φόντο και θα το εμφανίσουμε στην οθόνη.

Πολλές φορές είναι επιτακτική ανάγκη να εμφανίσουμε πληροφορίες στη οθόνη μας ώστε να βοηθήσουμε τον χρήστη και να τον ενημερώσουμε για τυχόν συμβάντα στο παιχνίδι. Αυτό γίνεται με την χρήση γραμματοσειρών και όχι εικόνων διότι έτσι σπαταλάμε περισσότερη μνήμη η οποία είναι πολύ σημαντική για το παιχνίδι μας. Όμως, για να εμφανίσουμε ένα κείμενο στην οθόνη μας πρέπει πρώτα να επιλέξουμε ένα φόντο, μία γραμματοσειρά. Έτσι θα προσθέσουμε στον διαχειριστή των αντικειμένων μας ένα φόντο, έπειτα θα το φορτώσουμε σε μία μεταβλητή και θα ζωγραφίσουμε με την βούρτσα μας το κείμενο στον χρήστη.

Για να προσθέσουμε ένα καινούριο φόντο στον διαχειριστή αντικειμένων κάνουμε δεξί κλικ επάνω του και επιλέγουμε διαδοχικά “Add->NewItem”. Στο παράθυρο που θα μας εμφανιστεί επιλέγουμε “SpriteFont” δίνουμε ένα όνομα και πατάμε κλικ στο κουμπί “OK”. Μόλις τελειώσουμε πρέπει να έχουμε κάτι σαν το παρακάτω:



**Εικόνα 24.** Το καινούριο φόντο στον διαχειριστή περιεχομένου.

Βλέπουμε ότι έχει προστεθεί το φόντο στον διαχειριστή ανάλογα με το όνομα που του έχουμε ορίσει. Δεξιά θα μας εμφανιστεί κώδικας XML, όπου και μπορούμε να επιλέξουμε το χρώμα, το μέγεθος και την οικογένεια της γραμματοσειράς αυτής. Έπειτα πάμε στην αρχή της κλάσης να δηλώσουμε κάποια αντικείμενα που θα μας βοηθήσουν για την δουλειά αυτή:

```

16 GraphicsDeviceManager graphics;
17 SpriteBatch spriteBatch;
18 Texture2D myBall;
19 Vector2 ballPosition;
20 SpriteFont mySpriteFont;
21 string myText = "Hello I am a string!";
22 Vector2 textPosition;

```

**Κώδικας 9.** Αρχικός κώδικας για την εμφάνιση του κειμένου στην οθόνη.

Στην γραμμή 20 δηλώσαμε ένα αντικείμενο **SpriteFont**, το οποίο είναι η συνδετική γέφυρα του επεξεργαστή περιεχομένου, του φόντου και του κυρίου προγράμματος. Στην γραμμή 21, δηλώσαμε ένα αντικείμενο τύπου **String**, το οποίο περιέχει το κείμενο το οποίο θέλουμε να εμφανίσουμε στον χρήστη. Τέλος στην γραμμή 22 δηλώσαμε ένα αντικείμενο **Vector**, το οποίο και θα κρατά την θέση στην οποία εμείς θέλουμε να εμφανιστεί το κείμενο στην οθόνη του **XNA** όπως και στην περίπτωση του **Texture2D** στο προηγούμενο παράδειγμα.

Στην συνέχεια προχωρούμε στην μέθοδο **LoadContent()** όπου εκεί πλέον θα φορτώσουμε τη γραμματοσειρά μας. Για τον λόγο αυτό γράφουμε τον εξής κώδικα:

```

35 protected override void LoadContent()
36 {
37     spriteBatch = new SpriteBatch(GraphicsDevice);
38     myBall = Content.Load<Texture2D>("Ball");
39     ballPosition = new Vector2(0, 0);
40
41     mySpriteFont = Content.Load<SpriteFont>("myFont");
42     textPosition = new Vector2(50f, 50f);
43 }

```

**Κώδικας 10.** Ο κώδικας για τη φόρτωση της γραμματοσειράς στο πρόγραμμα.

Βλέπουμε πως η μέθοδος στη γραμμή 41 είναι όμοια με τη μέθοδο φόρτωσης **Texture2D** πριν. Απλώς αυτή την φορά ξανακαλούμε το αντικείμενο **Content** να φορτώσει το αντικείμενο τύπου γραμματοσειράς και να το συνδέσει με τη μεταβλητή μας στο πρόγραμμα. Στη γραμμή 42 απλώς ορίζουμε την θέση όπου η γραμματοσειρά θα εμφανιστεί.

Αφού, λοιπόν, τελειώσαμε με όλα τα παραπάνω, είναι καιρός να χρησιμοποιήσουμε τη βούρτσα του προγράμματος έτσι ώστε να εμφανίσουμε το κείμενό μας στην οθόνη του **XNA**. Για να το πετύχουμε αυτό θα πάμε στην μέθοδο **Draw()** όπου και θα καλέσουμε την βούρτσα μας να ζωγραφίσει το κείμενό μας με όλες τις παραμέτρους που ορίσαμε πριν:

```

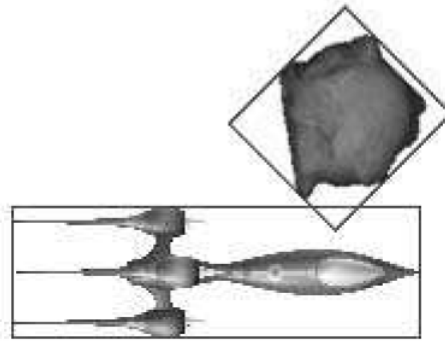
58     protected override void Draw(GameTime gameTime)
59     {
60         GraphicsDevice.Clear(Color.CornflowerBlue);
61
62         spriteBatch.Begin();
63         spriteBatch.Draw(myBall, ballPosition, Color.White);
64         spriteBatch.DrawString(mySpriteFont, myText, textPosition, Color.Red);
65         spriteBatch.End();
66
67         base.Draw(gameTime);
68     }
69 }

```

**Κώδικας 11.** Εμφάνιση του κειμένου στην οθόνη του XNA.

Βλέπουμε πως καλούμε στη γραμμή 64 τη μέθοδο **DrawString** του αντικειμένου της βούρτσας η οποία και θα πάρει σαν ορίσματα το φόντο, το κείμενο, την θέση του κειμένου και το χρώμα. Το χρώμα μπορούμε να το αλλάξουμε μέσα από το αρχείο XML του φόντου, και αν το αφήσουμε **White** αυτό θα εμφανιστεί χωρίς αλλαγές. Εάν πειράξουμε το χρώμα στην συνάρτηση **DrawString**, τον τελευταίο λόγο παρ' όλες τις ρυθμίσεις του αρχείου XML, θα τον έχει αυτή η επιλογή.

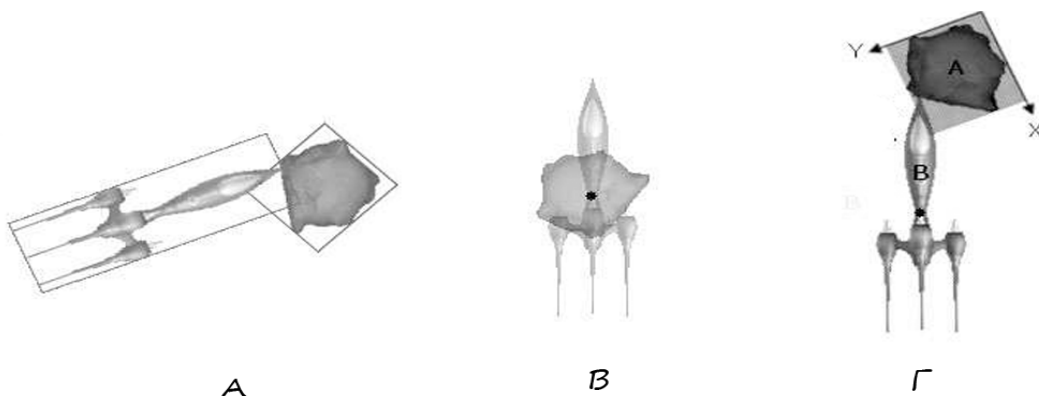
Τελευταίο αντικείμενο με το οποίο θα ασχοληθούμε θεωρητικά, το οποίο και θα μας φανεί χρήσιμο για τον τρισδιάστατο χώρο έπειτα, είναι ο εντοπισμός της σύγκρουσης δύο αντικειμένων στον δυσδιάστατο χώρο. Ο εντοπισμός της σύγκρουσης μας βοηθά να καταλαβαίνουμε πότε δύο αντικείμενα η έρχονται σε επαφή μεταξύ τους. Όταν κατασκευάζουμε τέτοιους αλγορίθμους πρέπει να είμαστε πολύ προσεκτικοί για να αποφύγουμε όλες τις περιπτώσεις τυχόν άστοχου εντοπισμού σύγκρουσης. Η μηχανή γραφικών **XNA** μας παρέχει ένα αντικείμενο το οποίο μας βοηθά να κατανοήσουμε τον εντοπισμό της σύγκρουσης το οποίο αντικείμενο είναι το **Rectangle**, το οποίο παίρνει ως όρισμα την πάνω αριστερή γωνία του αντικειμένου, το ύψος και το πλάτος του. Όταν δημιουργήσουμε πλέον δύο αντικείμενα **Rectangle** μπορούμε να χρησιμοποιήσουμε τη μέθοδο **Intersects** η οποία επιστρέφει ψευδή ή αληθή τιμή ανάλογα εάν υπάρχει σύγκρουση. Αυτή η μέθοδος επειδή κατασκευάζει ένα νοητό τετράγωνο γύρω από τα αντικείμενά μας δεν είναι πάντα σωστή και μπορεί να οδηγήσει σε λανθασμένα συμπεράσματα όπως φαίνεται στην παρακάτω εικόνα:



**Εικόνα 25.** Λανθασμένος εντοπισμός σύγκρουσης δύο αντικείμενων.

Αυτή λοιπόν η μέθοδος μας δίνει απλά να κατανοήσουμε πότε ένα αντικείμενο μπαίνει στην περιοχή ενός άλλου. Για να έχουμε σωστά αποτελέσματα μόλις γίνει ο πρώτος εντοπισμός πρέπει σε δεύτερη φάση να εφαρμόσουμε έναν πιο εξονυχιστικό έλεγχο για το αν πραγματικά αυτά τα δύο αντικείμενα ήρθαν σε σύγκρουση. Αυτό θα το πετύχουμε με έναν αλγόριθμο **PerPixelCollisionChecking**. Ας πάμε λοιπόν να εξηγήσουμε το πώς λειτουργεί αυτή η μέθοδος.

Ας πούμε ότι με την πρώτη μέθοδο καταφέρνουμε να εντοπίσουμε μία αρχική σύγκρουση. Σε αυτό το σημείο εφαρμόζουμε τον δεύτερο αλγόριθμο **PerPixelCollision**, ο οποίος θα μας δώσει μια πραγματική εικόνα για το αν έχουμε σύγκρουση. Για να εφαρμόσουμε τον αλγόριθμο αυτόν ακολουθούμε τρία απλά βήματα. Αρχικά τα αντικείμενα πηγαίνουν στην αρχική τους θέση (0,0) σαν να μην είχαν κουνηθεί ποτέ, έπειτα μετακινούμε το αντικείμενο του αστεροειδή με τον ανάστροφα σε σχέση με το πύραυλο. Η τεχνική αυτή φαίνεται παρακάτω:



**Εικόνα 26.** Βέλτιστος αλγόριθμος εντοπισμός σύγκρουσης.

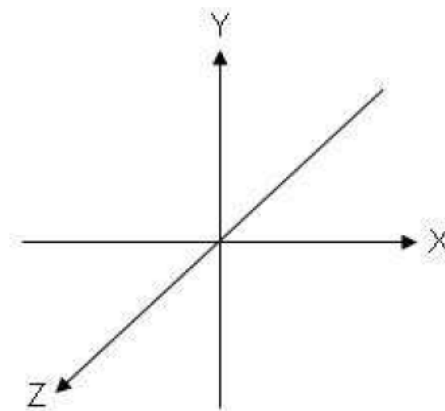
Βλέπουμε στην προηγούμενη εικόνα αρχικά πως τρέχει ο αρχικός αλγόριθμος εντοπισμού σύγκρουσης με τα **Rectangle**. Έπειτα μόλις έρθουμε στην κατάσταση **A**, όπου εντοπίζεται μία σύγκρουση τα αντικείμενα ευθυγραμμίζονται με βάση τα κέντρα

τους, όπως φαίνεται στο σχήμα **B**. Στην συνέχεια το αντικείμενο A αναστρέφεται με βάση το αντικείμενο B και εάν υπάρχει σύγκρουση των Pixels τότε συμπερασματικά λέμε πως υπάρχει σύγκρουση των δύο αντικειμένων. Σε καμία περίπτωση δεν μπορεί να τρέχει μόνο ο ένας αλγόριθμος η ο άλλος. Ο πρώτος αλγόριθμος αν και πιο ελαφρύς από το δεύτερο δεν μας δίνει μια πλήρη εικόνα για τις συγκρούσεις στο παιχνίδι μας. Ο δεύτερος αν και πιο ορθός σπαταλά πολλούς πόρους οι οποίοι είναι σημαντικοί για το παιχνίδι μας. Έτσι χρησιμοποιούμε και τους δύο με έξυπνο τρόπο για να έχουμε την βέλτιστη απόδοση.

Έτσι αφού καλύψαμε κάποια βασικά αντικείμενα που αφορούν το δυσδιάστατο χώρο και τα αντικείμενά του, είναι καιρός να προχωρήσουμε στις τρεις διαστάσεις.

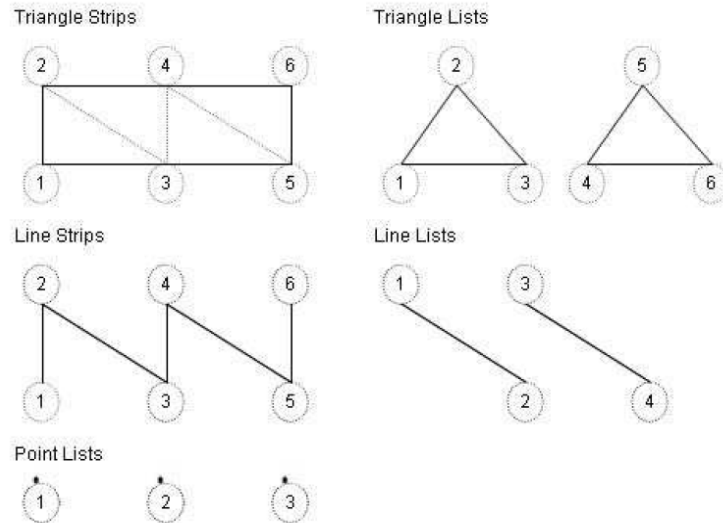
## 5.6 - Βασικά στοιχεία τρισδιάστατων γραφικών

Καλύψαμε στο προηγούμενο κεφάλαιο κάποια βασικά στοιχεία των γραφικών των δύο διαστάσεων. Πλέον και με λίγη εξοικείωση που αποκτήθηκε και με το περιβάλλον ανάπτυξης αλλά και με την μηχανή γραφικών μπορούμε να προχωρήσουμε ένα βήμα παραπέρα. Σε αυτό το κεφάλαιο γίνεται αναφορά στα βασικά σχήματα γραφικών όπως είχαμε συζητήσει και σε προηγούμενα κεφάλαια. Μέχρι και τα πιο περίπλοκα και εξειδικευμένα γραφικά, συνίστανται από κάτι το οποίο είναι όμοιο: Από γραμμές, σημεία και τρίγωνα που είναι και τα πιο σημαντικά διότι μας βοηθούν να εξοικονομήσουμε αρκετούς υπολογιστικούς πόρους. Αυτά τα βασικά αντικείμενα των γραφικών, ζωγραφίζονται μέσα στην μηχανή του **XNA** και εμφανίζονται με βάση το καρτεσιανό τρισδιάστατο επίπεδο, το οποίο αποτελείται από τις μεταβλητές X,Y,Z. Καθώς ζωγραφίζουμε βασικά αντικείμενα γραφικών μπορούμε να ορίσουμε πέρα από τις συντεταγμένες τους και άλλες πληροφορίες. Παράδειγμα μπορούμε να εφαρμόσουμε πάνω τους ένα **Texture2D**, να ορίσουμε τον φωτισμό τους, το χρώμα τους και άλλα. Η παρακάτω εικόνα δείχνει το καρτεσιανό τρισδιάστατο επίπεδο που χρησιμοποιεί η μηχανή γραφικών **XNA**:



**Εικόνα 27.** Το τρισδιάστατο καρτεσιανό επίπεδο στην μηχανή γραφικών XNA.

Στην επόμενη εικόνα παρουσιάζονται όλα τα βασικά σχήματα τα οποία χρησιμοποιεί το **XNA** και με τα οποία μπορούμε να σχεδιάσουμε μέχρι και τα πιο περίπλοκα μοντέλα:



**Εικόνα 28.** Τα βασικά στοιχεία της μηχανής γραφικών XNA.

Βλέπουμε στην εικόνα πως είτε μπορούμε να έχουμε μία “ταινιά” από αντικείμενα τα λεγόμενα **strips** είτε μπορούμε να τα έχουμε ανεξάρτητα. Συνήθως χρησιμοποιούνται τα **strips** διότι έτσι γλυτώνουμε πολύτιμους πόρους από τον υπολογιστή μας. Φυσικά τα σημεία (points), δεν γίνεται να οριστούν σε λίστα ειδάλλως θα αποτελούσαν γραμμές (lines). Για να κατανοήσουμε για ποιο λόγο χρησιμοποιούνται τα **strips** η παρακάτω φόρμουλα θα μας βοηθήσει. Στην περίπτωση των τριγώνων λοιπόν θα έχουμε:

(Λίστα) Συνολικές κορυφές τριγώνων:  $N_{\text{τριγώνων}} * 3$  κορυφές το καθένα.

(Ταινία) Συνολικές κορυφές τριγώνων:  $N_{\text{τριγώνων}} + 2$  κορυφές.

Για ποιο λόγο όμως γίνεται αυτό; Η κάθε κορυφή λοιπόν του τριγώνου μας πρέπει να δηλώνεται με τρεις συντεταγμένες στο τρισδιάστατο επίπεδο. Με άλλα λόγια, εάν θέλουμε να σχεδιάσουμε ένα και μόνο τρίγωνο με λίστα τότε θα χρειαστούμε 3 κορυφές αλλά η κάθε κορυφή θέλει από ένα X,Y,Z οπότε χονδρικά η πληροφορία που θα χρησιμοποιήσουμε θα είναι  $3*3= 9$  σημεία και πάει λέγοντας. Στην περίπτωση, όμως, της ταινίας ενώ σχεδιάζουμε τα τρίγωνα και έχουμε τελειώσει, εάν υποθέσουμε ότι προσθέτουμε ακόμη ένα τρίγωνο τις δύο κορυφές θα τις έχουμε έτοιμες από πριν οπότε το μόνο που χρειαζόμαστε για ένα καινούριο τρίγωνο είναι η δήλωση μιας μόνο κορυφής. Φανταστείτε στην κλίμακα των εκατομμυρίων τριγώνων ποιο θα ήταν το όφελος από την χρήση ταινίας έναντι λίστας! Κάτι παρόμοιο συμβαίνει και με τις γραμμές.



Οι κορυφές πάντως των βασικών σχημάτων είναι ίδιες για όλα τα σχήματα, δηλαδή χρειάζονται μόνο τρία σημεία ώστε να εμφανιστούν στον χώρο. Υπάρχουν διαφορετικά είδη κορυφών. Κορυφές που δέχονται χρώματα, φως και άλλα. Η παρακάτω εικόνα μας δείχνει ποια αντικείμενα κορυφών είναι έτοιμα. Παρ' όλ' αυτά μπορούμε και εμείς να κατασκευάσουμε δικές μας κορυφές εάν δεν μας βολεύουν οι έτοιμες.

<i>Vertex Storage Format</i>	<i>Function</i>
<i>VertexPositionColor</i>	<i>Stores X, Y, Z and color coordinates</i>
<i>VertexPositionTexture</i>	<i>Stores X, Y, Z and image coordinates</i>
<i>VertexPositionColorTexture</i>	<i>Stores X, Y, Z, color, and image coordinates</i>
<i>VertexPositionNormalTexture</i>	<i>Stores X, Y, Z positions, a normal vector, and image coordinates</i>

**Εικόνα 29.** Τα είδη κορυφών στο XNA.

Όπως βλέπουμε στην παραπάνω εικόνα, μπορούμε να χρησιμοποιήσουμε το **VertexPositionColor** όπου όπως καταλαβαίνουμε να αποθηκεύσουμε την πληροφορία του χρώματος και των συντεταγμένων της κορυφής. Επίσης, το **VertexPositionTexture** μας βοηθά να αποθηκεύσουμε πληροφορίες για την θέση της κορυφής, αλλά και να ορίσουμε ένα **Texture**. Έπειτα, το **VertexPositionColorTexture** μας βοηθά να αποθηκεύσουμε την θέση της κορυφής, το φωτισμό (**Normal**) και ένα **texture**. Τώρα η απορία που δημιουργείται σε αυτό το σημείο είναι η εξής: πριν αναφέραμε ότι υπάρχουν περιπτώσεις όπου αυτές οι έτοιμες κορυφές δεν μας κάνουν. Σε ποια περίπτωση όμως; Φανταστείτε ότι θέλουμε να έχουμε τρίγωνα με πολλά **textures**. Τότε σε αυτή την περίπτωση θα έπρεπε να χρησιμοποιήσουμε μία κορυφή **Multitextured**, διότι κάτι τέτοιο δεν υπάρχει έτοιμο από το XNA. Επίσης, σε περιπτώσεις όπου υπάρχουν πολλές πηγές φωτισμού, παράδειγμα σε ένα παιχνίδι ποδοσφαίρου, θα θέλαμε ιδανικά να ορίσουμε την κάθε μία από τις τέσσερις πηγές φωτός σαν ξεχωριστή παράμετρο της κορυφής. Σε αυτές τις περιπτώσεις δημιουργούμε δικές μας **custom** κορυφές.

Από την πράξη στην θεωρία λοιπόν. Ήρθε η στιγμή να ζωγραφίσουμε το πρώτο μας τρίγωνο στην οθόνη του XNA.

Θα χρησιμοποιήσουμε αρχικά τις κορυφές τύπου **VertexPositionColor**, οπότε θα χρειαστεί να ορίσουμε τις θέσεις των κορυφών και το χρώμα της κάθε κορυφής. Πάνω στην κλάση γράφουμε τον εξής κώδικα:

```

16 | GraphicsDeviceManager graphics;
17 | VertexPositionColor[] vertices;
18 | BasicEffect effect;

```

**Κώδικας 12.** Ο αρχικός κώδικας για τη δημιουργία του τριγώνου.

Στην γραμμή 17 δηλώσαμε ένα πίνακα από κορυφές τύπου **VertexPositionColor** και στη γραμμή 18 ένα βασικό **εφέ**. Προς στιγμήν δεν θα εξηγήσουμε το τι δουλειά ακριβώς κάνει αυτό, απλώς το χρησιμοποιούμε έτσι. Θα μπορούσαμε αρχικά να πούμε ότι είναι το “πινέλο” των τριγώνων και είναι υπεύθυνο για την εμφάνιση πολλών εντυπωσιακών εφέ στα σύγχρονα παιχνίδια.

Παρακάτω αφού κάναμε τις αρχικές δηλώσεις ήρθε καιρός να ορίσουμε το χρώμα και τις θέσεις του νέου μας τριγώνου:

```
26  protected override void Initialize()
27  {
28      effect = new BasicEffect(GraphicsDevice);
29      effect.VertexColorEnabled = true ;
30
31      vertices = new VertexPositionColor[3];
32
33      vertices[0].Position = new Vector3(-0.5f, -0.5f, 0f);
34      vertices[0].Color = Color.Red;
35
36      vertices[1].Position = new Vector3(0, 0.9f, 0f);
37      vertices[1].Color = Color.Green;
38
39      vertices[2].Position = new Vector3(0.5f, -0.5f, 0f);
40      vertices[2].Color = Color.Yellow;
41
42      base.Initialize();
43  }
```

**Κώδικας 13.** Η κατασκευή του τριγώνου.

Στη γραμμή 29 ενεργοποιούμε τη δυνατότητα του εφέ να δίνει το χρώμα που θα ορίσουμε παρακάτω. Εάν δεν το κάνουμε αυτό το τρίγωνό μας θα εμφανιστεί άσπρο. Στη γραμμή 31, ορίζουμε ότι θα χρειαστούμε τρεις κορυφές ώστε να σχεδιάσουμε ένα τρίγωνο. Από την γραμμή 33-40 ορίζουμε αντίστοιχα τα μέλη των κορυφών που θέλουμε να εμφανιστούν και το χρώμα τους.

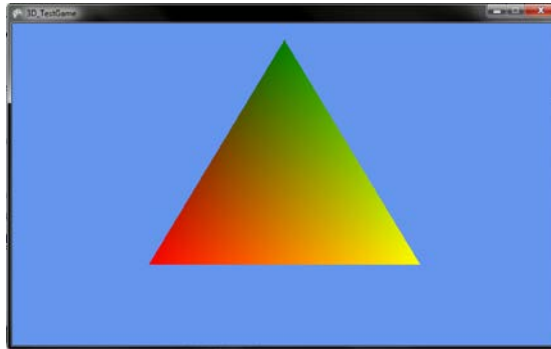
Αφού τελειώσαμε με τις αρχικές διαδικασίες, δεν μένει παρά να ζωγραφίσουμε το τρίγωνό μας. Για αυτό το λόγο θα πάμε στην συνάρτηση όπου μπορούμε να ζωγραφίσουμε και θα εισάγουμε τον παρακάτω κώδικα:

```
60  protected override void Draw(GameTime gameTime)
61  {
62      GraphicsDevice.Clear(Color.CornflowerBlue);
63
64      effect.CurrentTechnique.Passes[0].Apply();
65      graphics.GraphicsDevice.DrawUserPrimitives<VertexPositionColor>(PrimitiveType.TriangleStrip, vertices, 0, 1);
66
67
68      base.Draw(gameTime);
69  }
```

**Κώδικας 14.** Εμφάνιση του τριγώνου στην οθόνη.

Στην γραμμή 64 ενεργοποιούμε το εφέ το οποίο θα βοηθήσει το τρίγωνό μας να ζωγραφιστεί. Τέλος, στη γραμμή 65, καλώντας την συνάρτηση **DrawUserPrimitives**, μέσω της συσκευής γραφικών, ζωγραφίζουμε το τρίγωνό μας. Οι αριθμοί που παίρνει ως ορίσματα σημαίνουν ότι θα ξεκινήσουμε να ζωγραφίζουμε από το τρίγωνο με αριθμό 0 και το 1 ότι θα ζωγραφίσουμε ένα τρίγωνο.

Εάν όλα πήγαν καλά, πρέπει να μας εμφανιστεί το παρακάτω σχήμα στην οθόνη όπως φαίνεται στην παρακάτω εικόνα:



**Εικόνα 30.** Το τρίγωνο επί της οθόνης.

Είδαμε, λοιπόν, πως με πολύ απλό τρόπο γίνεται εφικτό να ζωγραφίσουμε ένα τρίγωνο. Τα χρώματα μέσα σε αυτό, όπως διαπιστώνουμε, γεμίζουν αυτόματα, όπως και στις περισσότερες μηχανές γραφικών. Πλέον είναι θέμα μαθηματικών από εδώ και πέρα να εισάγουμε στην μηχανή γραφικών όρους περιστροφής (rotation), και άλλα. Στο επόμενο κεφάλαιο θα ασχοληθούμε με την λήψη εισόδου από τον χρήστη.

## 5.7 - User Input

Το να λαμβάνουμε εισόδους από τον χρήστη από διαφορετικά μέσα είναι σημαντικό για κάθε παιχνίδι. Στα διάφορα αυτά μέσα περιλαμβάνονται ένα χειριστήριο για υπολογιστή, ένα πληκτρολόγιο ή ένα ποντίκι. Η μηχανή γραφικών XNA, απλοποιεί πολύ τα πράγματα για εμάς. Μπορούμε να χειριστούμε κάθε πλήκτρο που πατήθηκε μέσα από έτοιμα αντικείμενα και κλάσεις του XNA. Ακόμα, μπορούμε να στείλουμε μέχρι και δονήσεις στα διάφορα χειριστήρια ώστε να κάνουμε έτσι πιο ζωντανό το παιχνίδι. Χρησιμοποιώντας την κλάση `Microsoft.Xna.Framework.Input.Keys`, από την λίστα που εμφανίζεται μπροστά μας μπορούμε να δούμε ποια κουμπιά μπορεί το XNA να υποστηρίξει για το πληκτρολόγιο. Μία ανασκόπηση γίνεται στον παρακάτω πίνακα:

<i>A to Z</i>	<i>Home</i>	<i>PageUp</i>
<i>Add</i>	<i>Insert</i>	<i>PrintScreen</i>
<i>CapsLock</i>	<i>Left</i>	<i>Right</i>
<i>D0 to D9</i>	<i>LeftAlt</i>	<i>RightAlt</i>
<i>Decimal</i>	<i>LeftControl</i>	<i>RightControl</i>
<i>Delete</i>	<i>LeftShift</i>	<i>RightShift</i>
<i>Divide</i>	<i>LeftWindows</i>	<i>RightWindows</i>
<i>Down</i>	<i>Multiply</i>	<i>Scroll</i>
<i>End</i>	<i>NumLock</i>	<i>Space</i>
<i>Enter</i>	<i>NumPad0 to</i>	<i>Subtract</i>
<i>Escape</i>	<i>NumPad9</i>	<i>Tab</i>
<i>F1 to F12</i>	<i>PageDown</i>	<i>Up</i>
<i>Help</i>		

**Εικόνα 31.** Τα κουμπιά που υποστηρίζει το XNA.

Για να λάβουμε την τωρινή κατάσταση του πληκτρολογίου, δηλαδή εάν ένα κουμπί πατήθηκε, θα πρέπει να χρησιμοποιήσουμε την συνάρτηση `Keyboard.GetState()`. Για να διαπιστώσουμε ότι κάποιο συγκεκριμένο κουμπί έχει πατηθεί θα το ελέγξουμε κάπως έτσι:

```
55 | protected override void Update(GameTime gameTime)  
56 | {  
57 |     KeyboardState ks = Keyboard.GetState();  
58 |     bool pressed = ks.IsKeyDown(Keys.A);  
59 |  
60 |     base.Update(gameTime);  
61 | }
```

**Κώδικας 15.** Έλεγχος πληκτρολογίου στην μηχανή γραφικών XNA.

Όπως βλέπουμε στην γραμμή 57, δεσμεύουμε την κατάσταση του πληκτρολογίου και την αναθέτουμε στην μεταβλητή **KS**. Έπειτα σε μία μεταβλητή **pressed** αναθέτουμε το αποτέλεσμα της συνάρτησης **IsKeyDown**, η οποία θα μας επιστρέψει εάν κάποιο πλήκτρο έχει πατηθεί ή όχι, με **true** ή **false** αντίστοιχα.

Με αυτόν τον τρόπο μπορούμε να διαπιστώσουμε εάν πλήκτρο πατήθηκε στο πληκτρολόγιο. Με ποιον, όμως, τρόπο θα μπορέσουμε να καταλάβουμε εάν ένα οποιοδήποτε πλήκτρο έχει πατηθεί και επιπλέον ποιο πλήκτρο είναι αυτό;

Αυτό μπορεί να γίνει με ένα πολύ έξυπνο τρόπο, χάρη κιόλας στις δυνατότητες που μας προσφέρει η μηχανή γραφικών XNA. Ξεκινώντας θα πάμε στην αρχή της κλάσης και θα δηλώσουμε κάποιες μεταβλητές οι οποίες είναι πολύ χρήσιμες για τη λειτουργία αυτού του προγράμματος:

```
14 public class Game1 : Microsoft.Xna.Framework.Game
15 {
16     GraphicsDeviceManager graphics;
17     SpriteBatch spriteBatch;
18     SpriteFont spriteFont;
19
20     Keys[] keysPressed;
21 }
```

**Κώδικας 16.** Δήλωση αρχικών μεταβλητών.

Στην γραμμή 17 και 18 δηλώσαμε τις κατάλληλες μεταβλητές έτσι ώστε να εμφανίζουμε κείμενα στην οθόνη όπως έχουμε δει σε προηγούμενα κεφάλαια. Στην γραμμή 20 δηλώνουμε ένα πίνακα με αντικείμενα **Keys** μέσα στην οποία θα βάζουμε κάθε φορά τα πλήκτρα τα οποία πατήθηκαν. Στην μέθοδο `Update()` τώρα, γράφουμε τον εξής κώδικα:

```
45 protected override void Update(GameTime gameTime)
46 {
47     KeyboardState ks = Keyboard.GetState();
48     keysPressed = ks.GetPressedKeys();
49
50     base.Update(gameTime);
51 }
```

**Κώδικας 17.** Κώδικας έπαρσης πλήκτρων πληκτρολογίου.

Βλέπουμε πως παίρνουμε στη γραμμή 47 την κατάσταση του πληκτρολογίου και στην επόμενη γραμμή καλούμε την συνάρτηση **GetPressedKeys** η οποία μας επιστρέφει ένα πίνακα με τα πατημένα κουμπιά τον οποίο τον ανάγουμε στη μεταβλητή **keysPressed** που αναφέραμε πριν.

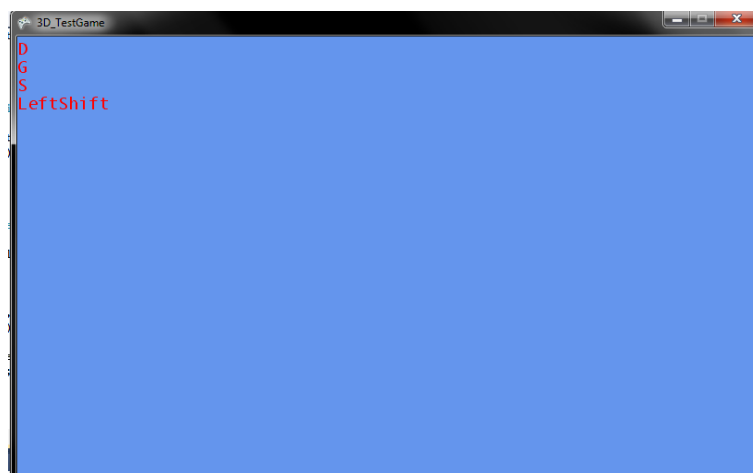
Δεν μένει πλέον να δούμε πώς, ενώ έχουμε όλα τα κουμπιά στον πίνακά μας, να τα εμφανίσουμε στην οθόνη. Για αυτό το λόγο θα πάμε στη συνάρτηση Draw() έτσι ώστε να γράψουμε κώδικα που θα ζωγραφίζει το επιθυμητό κείμενο:

```
53     protected override void Draw(GameTime gameTime)
54     {
55         GraphicsDevice.Clear(Color.CornflowerBlue);
56
57         spriteBatch.Begin();
58
59         Vector2 position = new Vector2(0, 0);
60         foreach (Keys key in keysPressed)
61         {
62             spriteBatch.DrawString(spriteFont, key.ToString(), position, Color.Red);
63             position.Y = position.Y + 20;
64         }
65         position = Vector2.Zero;
66
67         spriteBatch.End();
68
69         base.Draw(gameTime);
70     }
```

**Κώδικας 18.** Εμφάνιση κουμπιών στην οθόνη.

Στην γραμμή 57 δηλώνουμε ότι θα ξεκινήσουμε τη ζωγραφική κειμένων. Έπειτα στη γραμμή 59 δηλώνουμε την θέση όπου θα εμφανίζονται τα κείμενά μας για να μην είναι το ένα πάνω στο άλλο. Έπειτα στην επόμενη γραμμή χρησιμοποιούμε την συνάρτηση **foreach**, η οποία είναι μία πάρα πολύ χρήσιμη συνάρτηση και μας επιτρέπει να περιηγούμαστε μέσα στα αντικείμενα ενός πίνακα χωρίς να γνωρίζουμε τον αριθμό τους εκ των προτέρων. Στη γραμμή 62 ζωγραφίζουμε το πλήκτρο που πατήθηκε και έπειτα δηλώνουμε ότι το επόμενο κείμενο θα εμφανιστεί 20 pixels πιο κάτω ώστε να μην εμφανιστεί πάνω στο προηγούμενο. Στη γραμμή 65 αρχικοποιούμε τη θέση εμφάνισης για την επόμενη οικογένεια πλήκτρων.

Εάν τρέξουμε το πρόγραμμα το αποτέλεσμα μας θα είναι κάπως έτσι:



**Εικόνα 32.** Το αποτέλεσμα του προγράμματος.

Είδαμε, λοιπόν, πως πολύ εύκολα μπορούμε να ελέγξουμε ποια πλήκτρα έχουν πατηθεί και έπειτα αυτά να τα χειριστούμε όπως εμείς θέλουμε.

Παρακάτω θα παρουσιάσουμε πώς μπορούμε να χειριστούμε το ποντίκι του υπολογιστή.

Το μόνο που έχουμε να κάνουμε είναι να δηλώσουμε μία μεταβλητή **MouseState** στην αρχή της κλάσης, κάτι αντίστοιχο που κάναμε και με το πληκτρολόγιο, και τέλος να πάρουμε την κατάσταση του ποντικιού στην συνάρτηση Update().

Αυτό θα το πετύχουμε με τον τρόπο που δείχνει ο επόμενος κώδικας:

```
44     protected override void Update(GameTime gameTime)
45     {
46         ms = Mouse.GetState();
47
48         base.Update(gameTime);
49     }
```

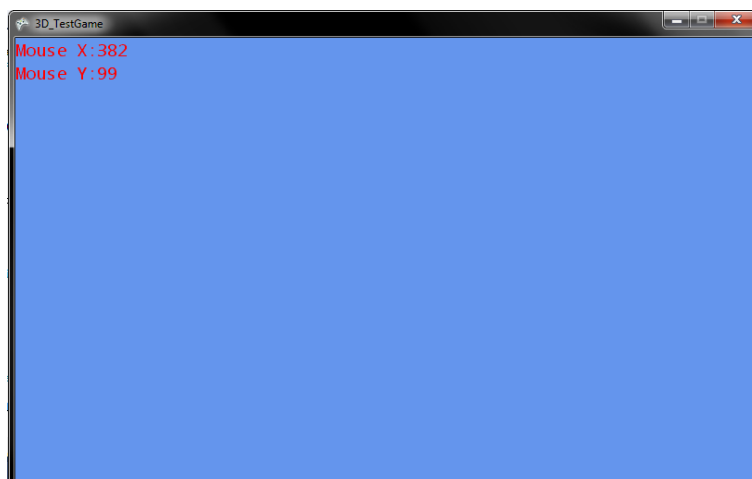
**Κώδικας 19.** Έπαρση κατάστασης ποντικιού.

Αυτό που μας μένει να κάνουμε είναι να πάμε στην συνάρτηση Draw() ώστε να εμφανίσουμε της πληροφορίες, την θέση του ποντικιού:

```
51     protected override void Draw(GameTime gameTime)
52     {
53         GraphicsDevice.Clear(Color.CornflowerBlue);
54
55         spriteBatch.Begin();
56
57         spriteBatch.DrawString(spriteFont, "Mouse X:" + ms.X.ToString() + "\nMouse Y:" + ms.Y.ToString(),
58                                 new Vector2(0,0) , Color.Red);
59
60         spriteBatch.End();
61
62         base.Draw(gameTime);
63     }
```

**Κώδικας 20.** Εμφάνιση πληροφοριών ποντικιού.

Στη γραμμή 57 όπως βλέπουμε χρησιμοποιούμε το μέλος της κατάστασης του ποντικιού **X**, όπου αποθηκεύεται η θέση του ποντικιού X και το μέλος **Y** όπου αποθηκεύεται η θέση Y του ποντικιού. Το αποτέλεσμα που θα έχουμε φαίνεται παρακάτω:



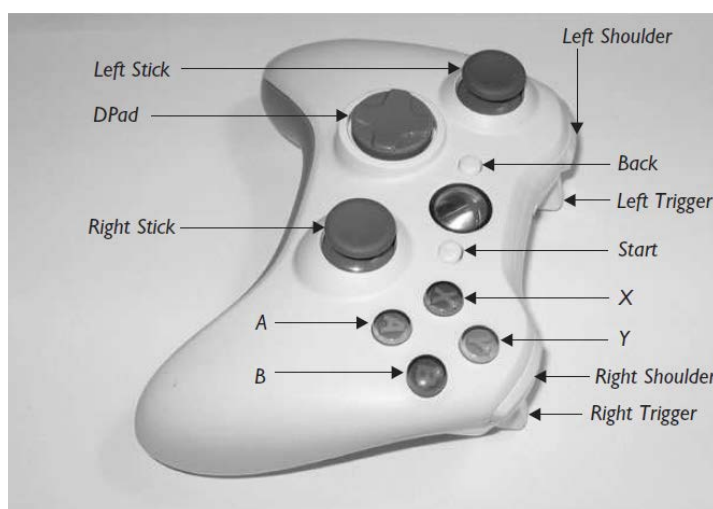
**Εικόνα 33.** Χειρισμός του ποντικιού.

Είδαμε λοιπόν πως με πολύ εύκολο τρόπο μπορούμε να επικοινωνήσουμε με τις περιφερειακές συσκευές εισόδου. Το ίδιο εύκολο είναι να επικοινωνήσουμε και με τα χειριστήρια παιχνιδιών. Αυτό το πετυχαίνουμε με την βοήθεια της κλάσης **Gamepad**, η οποία δεν θα παρουσιαστεί σε αυτήν την εργασία διότι είναι κάτι αντίστοιχο με αυτά που παρουσιάστηκαν πριν. Τα κουμπιά για παράδειγμα που υποστηρίζονται από αυτή την κλάση για ένα χειριστήριο XBOX είναι τα παρακάτω:

<i>Buttons.A</i>	<i>Buttons.Right Shoulder</i>	<i>DPad.Down</i>
<i>Buttons.B</i>	<i>Buttons.RightStick</i>	<i>DPad.Left</i>
<i>Buttons.Back</i>	<i>Buttons.Start</i>	<i>DPad.Right</i>
<i>Buttons.LeftShoulder</i>	<i>Buttons.X</i>	<i>DPad.Up</i>
<i>Buttons.LeftStick</i>	<i>Buttons.Y</i>	

**Εικόνα 34.** Τα κουμπιά του χειριστηρίου XBOX.

Τέλος το χειριστήριο του XBOX παρουσιάζεται στην επόμενη εικόνα:



**Εικόνα 35.** Το χειριστήριο του XBOX.



Όσον αφορά τώρα το ποντίκι θα μπορούσαμε να επεκτείνουμε αυτή την εργασία. Εάν είχαμε μία μεταβλητή **position** η οποία και θα άλλαζε σύμφωνα με τη κίνηση του ποντικιού και αν είχαμε μία εικόνα η οποία θα έπαιρνε σαν θέση της την μεταβλητή **position**, θα είχαμε μία κινούμενη εικόνα σύμφωνα με τη θέση του ποντικιού. Μπορούμε να “παίξουμε” με την μηχανή γραφικών μας ώστε να αποκτήσουμε μεγάλη εξοικείωση.

Αφού λοιπόν είδαμε λοιπόν βασικές τεχνικές ελέγχου εισόδων, στο επόμενο κεφάλαιο θα δούμε πως μπορούμε να δημιουργήσουμε εφέ με την βοήθεια της μνήμης της κάρτας γραφικών η οποία και είναι ταχείας προσπέλαση και χρησιμοποιείται για αυτόν ακριβώς το σκοπό.

## 5.8 - Pixel/Vertex Shader

Ας μιλήσουμε τώρα λίγο για την τεχνική Pixel/Vertex programming. Όπως είδαμε σε προηγούμενο κεφάλαιο, για να καταφέρουμε να σχεδιάσουμε τρισδιάστατα γραφικά θα πρέπει να χρησιμοποιήσουμε κάποιο εφέ. Αυτό το εφέ αρχικά το πήραμε έτοιμο από τη μηχανή γραφικών XNA. Θα μπορούσαμε όμως να δημιουργήσουμε και να χρησιμοποιήσουμε το δικό μας εφέ.

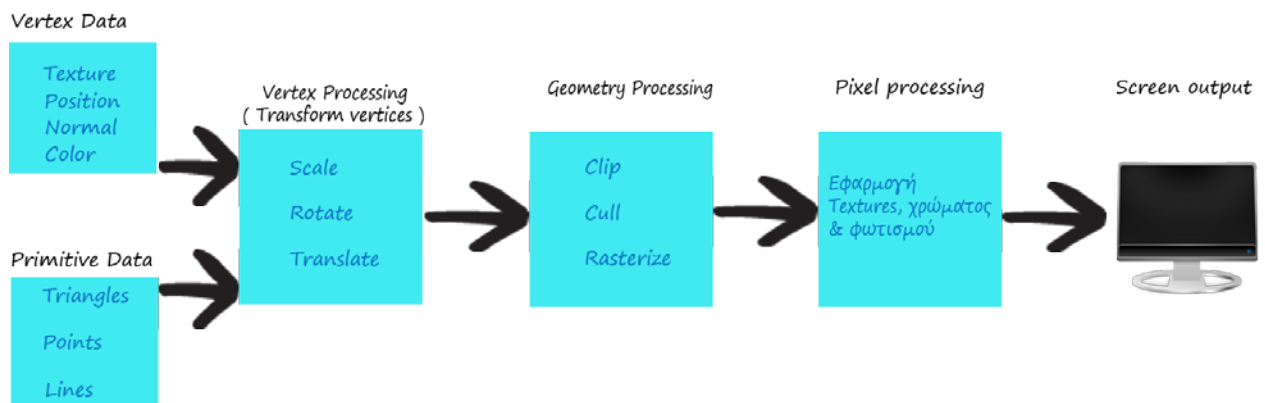
Πριν το 2002 οι προγραμματιστές γραφικών μπορούσαν να χρησιμοποιήσουν μόνο τις έτοιμες εντολές του DirectX. Έπειτα, και με τον ερχομό του DirectX 8, οι προγραμματιστές μπορούσαν να προγραμματίζουν με πιο ευέλικτο τρόπο τα γραφικά τους και να πετυχαίνουν μεγάλες ταχύτητες με αυτά, για τον λόγο ότι μπορούσαν να προγραμματίζουν απευθείας πάνω στην κάρτα γραφικών. Από τότε, δηλαδή, οι προγραμματιστές μπορούσαν να προγραμματίζουν τις κορυφές των τριγώνων και τα pixels απευθείας επάνω στην GPU (Graphics Processing Unit). Όλα αυτά που παρουσιάστηκαν σε προηγούμενα κεφάλαια δεν αναιρούνται με αυτό, αλλά και πάλι θα είναι χρήσιμα, μόνο που αυτή την φορά θα είναι δυνατόν να γράφουμε τα δικά μας εφέ με την γλώσσα **HLSL** της Microsoft, που είναι κοινή και για τη μηχανή γραφικών XNA, αλλά και για την μηχανή γραφικών DirectX.

Η γλώσσα προγραμματισμού HLSL χρησιμοποιείται όχι για να σχεδιάζουμε καλύτερα τρίγωνα, αλλά για να έχουμε καλύτερη ποιότητα με τα τρίγωνα αυτά. Για παράδειγμα, μπορούμε να κατασκευάσουμε δικό μας φωτισμό. Με την HLSL μπορούμε να κατασκευάσουμε οποιοδήποτε μαθηματικό υπολογισμό για τα τρίγωνα μας. Η HLSL είναι ο συνδετικός κρίκος που συνδέει τον κώδικα στο XNA με την οθόνη μας. Συχνά το δικό μας εφέ στην γλώσσα των προγραμματιστών ονομάζεται **Shader** και εφεξής θα αναφέρετε με αυτή την ορολογία. Ο Shader, λοιπόν, μας δίνει την δύναμη να ελέγξουμε πώς θα εμφανιστούν τα διάφορα δεδομένα των τριγώνων στη οθόνη μας, για παράδειγμα το χρώμα, οι σκιές και η θέση. Αυτό μας δίνει την δυνατότητα να εισάγουμε νέες έννοιες στα τρίγωνα μας όπως είναι η διαφάνεια (transparency), ο φωτισμός, το εφέ Gauss, το γνωστό σε όλους μας **blur**, και η δυνατότητα για multitexturing.

Για μερικά εφέ, όπως είναι αυτό της φωτιάς, θα πρέπει να χρησιμοποιήσουμε τον δικό μας shader. Συχνά στον προγραμματισμό γραφικών θα ακούσετε τον όρο **αγωγός γραφικών** ή **graphics pipeline**. Αυτός ο αγωγός είναι υπεύθυνος για την μετατροπή

των δεδομένων κορυφών του τριγώνου σε pixels στην οθόνη. Ο vertex/pixel shader φυσικά παίζει ουσιαστικό ρόλο σε αυτή τη διαδικασία.

Ο vertex shader κάνει μετατροπές στα δεδομένα (vertex inputs) και όταν αυτά περνούν από τον shader, οι πίσω πλευρές του δεν είναι ορατές από τον χρήστη. Αυτή η μέθοδος ονομάζεται **culling** και μπορούμε να ορίσουμε πότε θα χρησιμοποιείται ή όχι. Για παράδειγμα, στην περίπτωση ενός τοπίου, εάν ορίσουμε να εμφανίζονται μόνο οι μπροστά πλευρές, αυτές δηλαδή που θα βλέπει ο χρήστης, μπορούμε να γλιτώσουμε μεγάλο bandwidth στην κάρτα γραφικών, αφού ο χρήστης δεν θα κοιτάει ποτέ από κάτω. Στην περίπτωση σχεδίασης όμως ενός τοίχου θα θέλαμε αυτή την δυνατότητα. Έπειτα, αναπτύσσονται τα απαραίτητα μαθηματικά έτσι ώστε να εφαρμοστούν οι σκιές, τα textures και οποιοδήποτε άλλο δεδομένο έχει ορίσει ο χρήστης και, τέλος, όλα αυτά μετατρέπονται σε pixels στην οθόνη μας. Το παρακάτω διάγραμμα μας βοηθά να καταλάβουμε την διαδικασία, η οποία και καλείται κάθε φορά που εμείς καλούμε την συνάρτηση Draw():



**Εικόνα 36.** Η διαδικασία με την οποία εμφανίζονται τα τρίγωνα & τα άλλα πρωτεύων σχήματα.

Με απλά λόγια, συνοπτικά μπορούμε να πούμε ότι εισάγουμε μέσα στον pixel/vertex shader μία είσοδο, έπειτα αυτός την μετατρέπει σε pixels και τα εμφανίζει στη οθόνη.

Παρακάτω θα δείξουμε ένα παράδειγμα για το πώς μπορούμε να πάρουμε στα χέρια μας αυτή τη διαδικασία:

```

struct VSinput{
    float4 position : POSITION0;
    float4 color    : COLOR0;
};
struct VStoPS{
    float4 position : POSITION0;
    float4 color    : COLOR0;
};
struct PSoutput{
    float4 color    : COLOR0;
};

```

### Κώδικας 21. Αρχικές δηλώσεις γλώσσας HLSL.

Βλέπουμε πως αρχικά πρέπει να κατασκευάσουμε κάποιες δομές οι οποίες θα περιέχουν τα δεδομένα που θα εισάγονται ή θα εξάγονται. Αυτός είναι ένας γενικός τρόπος και ακολουθείται πάντα με αυτό το πρότυπο. Όπως είπαμε και πριν, αρχικά σαν είσοδο έχουμε τις κορυφές των τριγώνων που, όπως βλέπουμε στην αρχή του κώδικα 21, στην δομή **VSinput** ως είσοδο λαμβάνουμε την θέση και το χρώμα των τριγώνων. Εάν ξεχάσουμε μέσα από τον κώδικά μας να δώσουμε αυτά τα δύο δεδομένα που ζητούμε εδώ θα μας εμφανιστεί το αντίστοιχο σφάλμα. Προχωρώντας κατασκευάζουμε μία δομή η οποία μετατρέπει τα δεδομένα μας σε pixels, έχουμε πάλι ως εισόδους τη θέση και το χρώμα στη δομή **VStoPS**. Τέλος, έχουμε την έξοδο μας με μία δομή **PSoutput** η οποία εξάγει μόνο το χρώμα του τριγώνου. Πάμε λοιπόν να δούμε πώς θα χρησιμοποιήσουμε αυτά τα δεδομένα με συναρτήσεις μέσα στη γλώσσα HLSL. Τα χαρακτηριστικά **COLOR0** ή **POSITION0** δεν είναι τίποτα άλλο παρά δείκτες και δηλώνουν πόσες εισόδους έχουμε. Παράδειγμα εάν θέλαμε ως εισόδους τρία χρώματα θα κάναμε το εξής: **COLOR0,COLOR1,COLOR2** και τα λοιπά. Τα νούμερα στη γλώσσα προγραμματισμού παιχνιδιών ονομάζονται **semantics**.

Έπειτα γράφουμε κώδικα HLSL με τον οποίο θα κάνουμε τις μετατροπές:

```

// alter vertex inputs
void VertexShader(in VSinput IN, out VStoPS OUT){
    // transform vertex
    OUT.position    = mul(IN.position, wvpMatrix);
    OUT.color       = IN.color;
}

// alter vs color output
void PixelShader(in VStoPS IN, out PSoutput OUT){
    float4 color    = IN.color;
    OUT.color       = clamp(color, 0, 1); // range between 0 and 1
}

// the shader starts here
technique BasicShader{
    pass p0{
        // declare & initialize ps & vs
        vertexshader    = compile vs_1_1 VertexShader();
        pixelshader     = compile ps_1_1 PixelShader();
    }
}

```

#### Κώδικας 22. Συναρτήσεις μετατροπής.

Κατασκευάζουμε αρχικά μία συνάρτηση με το όνομα **VertexShader** (το όνομα μπορεί να είναι οτιδήποτε), όπου δηλώνουμε ότι θα παίρνει ως είσοδο τη δομή **VSinput** και θα μας δίνει ως έξοδο τη δομή **VStoPS**, με τα χαρακτηριστικά **in** και **out** αντίστοιχα. Έπειτα κατασκευάζουμε μία συνάρτηση **PixelShader** η οποία παίρνει ως είσοδο την έξοδο του **VertexShader** και μας δίνει **pixels**. Στο τέλος μπορούμε να έχουμε πολλές τεχνικές οι οποίες καλούν τις αντίστοιχες συναρτήσεις όπως θέλουμε εμείς. Την συγκεκριμένη τεχνική την ονομάσαμε **BasicShader** αλλά θα μπορούσε να έχει οποιοδήποτε όνομα. Απλώς είναι το όνομα με το οποίο θα αποτελεί το αναγνωριστικό μέσα από το πρόγραμμα.

Στην τεχνική αυτή ορίζουμε τις δεσμευμένες μεταβλητές **vertex** και **pixel shader** ότι θα χρησιμοποιούν τις συναρτήσεις που κάναμε πριν και θα γίνουν `compile` με την αντίστοιχη έκδοση `Vertex/Pixel shader` που ορίζουμε εμείς. Όπως βλέπουμε, η τεχνική μας μπορεί να έχει πολλά πέρασμα. Το πρώτο πέρασμα το ονομάζουμε **pass P0**, το δεύτερο θα ήταν **pass P1** και τα λοιπά. Κάποιες εντολές οι οποίες δεν αναφέρθηκαν, όπως για παράδειγμα η **clamp** και το **float4**, συνοψίζονται στους παρακάτω πίνακες. Αρχικά συνοψίζονται τα δεδομένα όπου μπορούμε να χρησιμοποιήσουμε με τη γλώσσα HLSL και οι αντιστοιχίες τους σε XNA περιβάλλον:

<i>XNA Data Type</i>	<i>HLSL Data Type</i>
<i>Matrix</i>	<i>float4x4</i>
<i>Texture2D</i>	<i>Texture</i>
<i>struct</i>	<i>struct</i>
<i>int</i>	<i>int</i>
<i>float</i>	<i>float</i>
<i>Vector2</i>	<i>float2 // array with two elements</i>
<i>Vector3</i>	<i>float3 // array with three elements</i>
<i>Vector4</i>	<i>float4 // array with four elements</i>
<i>Color</i>	<i>float3 (with no alpha blending) or float4 (with alpha blending)</i>

**Εικόνα 37.** Αντιστοιχίες μεταβλητών σε HLSL και XNA.

Στην επόμενη εικόνα συνοψίζονται οι συναρτήσεις όπου μπορούμε να χρησιμοποιήσουμε στην γλώσσα HLSL:

<i>HLSL Intrinsic Functions</i>	<i>Inputs</i>	<i>Component Type</i>	<i>Outputs</i>
<i>abs (a)</i>	<i>a is a scalar, vector, or matrix.</i>	<i>float, int</i>	<i>Absolute value of a</i>
<i>clamp (a, min, max)</i>	<i>clamp (a, min, max)</i>	<i>float, int</i>	<i>Clamped value for a</i>
<i>cos (a)</i>	<i>a is a scalar, vector, or matrix.</i>	<i>float</i>	<i>Same dimension as a</i>
<i>dot (a, b)</i>	<i>a and b are vectors.</i>	<i>float</i>	<i>A scalar vector (dot product)</i>
<i>mul (a, b)</i>	<i>a and b can be vectors or matrices, but the a columns must match the b rows.</i>	<i>float</i>	<i>Matrix or vector, depending on the inputs</i>
<i>normalize (a)</i>	<i>a is a vector.</i>	<i>float</i>	<i>Unit vector</i>
<i>pow (a, b)</i>	<i>a is a scalar, vector, or matrix. b is the specified power.</i>	<i>a is a float. b is an integer.</i>	<i>a<sup>b</sup></i>
<i>saturate (a)</i>	<i>a is a scalar, vector, or matrix.</i>	<i>a is a float.</i>	<i>a clamped between 0 and 1</i>
<i>sin (a)</i>	<i>a is a scalar, vector, or matrix.</i>	<i>float</i>	<i>Same dimension as a</i>
<i>tan (a)</i>	<i>a is a scalar, vector, or matrix.</i>	<i>float</i>	<i>Same dimension as a</i>
<i>tex2D (a, b)</i>	<i>a is a sampler2D. b is a vector.</i>	<i>a is a sampler2D. b is a two-dimensional float.</i>	<i>Vector</i>

**Εικόνα 38.** Οι συναρτήσεις της γλώσσας HLSL.

Πηγή: [http://msdn.microsoft.com/en-us/library/bb509635\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/bb509635(v=vs.85).aspx).

Παρακάτω θα δούμε πώς μπορούμε να συνδέσουμε αυτό το εφέ μας με το κύριο πρόγραμμα. Δεν γίνεται παρουσίαση κάποιου παραδείγματος διότι θα παρουσιαστεί αργότερα στο πρόγραμμα, και θα πρέπει πρώτα να μιλήσουμε για την κάμερα, η οποία είναι ζωτικής σημασίας για το παιχνίδι μας αλλά και για το εφέ μας.

Το μόνο, λοιπόν, που έχουμε να κάνουμε για να συνδέσουμε το εφέ μας με το πρόγραμμα είναι να δηλώσουμε μία μεταβλητή **effect** και να πούμε κάτι σαν το παρακάτω: **effect=Content.Load<Effect>("myEffect") ;** . Για να εφαρμόσουμε την τεχνική που γράψαμε θα πρέπει μέσα στη μέθοδο Draw() να γράψουμε το παρακάτω: **effect.Techniques["TechniqueName"].Passes[0].Apply();** και έπειτα να καλέσουμε την συνάρτηση **DrawUserPrimitives** όπως είδαμε και σε προηγούμενο παράδειγμα. Είδαμε λοιπόν γενικά πως η μηχανή γραφικών XNA μας δίνει τη δυνατότητα να προγραμματίζουμε γραφικά σε χαμηλό επίπεδο και να έχουμε πρόσβαση στα πάντα.

Αυτό ήταν και το τελευταίο κεφάλαιο εισαγωγής για τα γραφικά στο XNA και πιστεύω πως ήταν μία καλή αρχή και μία καλή βάση ώστε ο αναγνώστης να προχωρήσει στα επόμενα κεφάλαια όπου παρουσιάζονται πιο περίπλοκες τεχνικές.

Κεφάλαιο 6

Διασύνδεση φόρμας με την μηχανή γραφικών

The logo for XNA (Xbox Game Studio) features the letters 'xna' in a stylized, metallic, 3D font. The 'x' is orange and red, while the 'n' and 'a' are grey. A small registered trademark symbol (®) is located to the right of the 'a'.A simulated Windows-style window with a blue title bar containing minimize, maximize, and close buttons. The window contains a form with three input fields: 'name', 'email', and 'comments'. The 'comments' field is a larger text area. A green play button icon is located at the bottom right of the form.



Σε ορισμένες περιπτώσεις σχεδίασης εφαρμογών που αποτελούνται από γραφικά, κρίνεται απαραίτητο να ενσωματώσουμε μία φόρμα στο περιβάλλον μας. Έτσι έχουμε την δυνατότητα να διαχειριστούμε τα γραφικά μας μέσω αυτής της φόρμας. Επίσης είναι ένας καλός τρόπος να διαχωρίσουμε τα γραφικά και την εμφάνισή τους από τις ενέργειες που θα κάνουμε πάνω σε αυτά αργότερα. Επίσης, η χρήση μίας φόρμας, σαν αυτές που βλέπουμε στα απλά προγράμματα, βοηθά τον χρήστη να έχει στα χέρια του ένα πιο “καθαρό” περιβάλλον εργασίας και να ξεχωρίζει τα αποτελέσματα που παράγονται από τις διάφορες παραμετροποιήσεις που κάνει σε ξεχωριστό παράθυρο.

Σε πολλές εφαρμογές, γίνεται αυτή η τεχνική της χρήσης μίας φόρμας διαχείρισης για τα γραφικά, όπως το **3DS Max Studio** και το **Blender**. Η συγκεκριμένη πτυχιακή για όλους τους παραπάνω λόγους θα έπρεπε να πληροί αυτήν την απαίτηση.

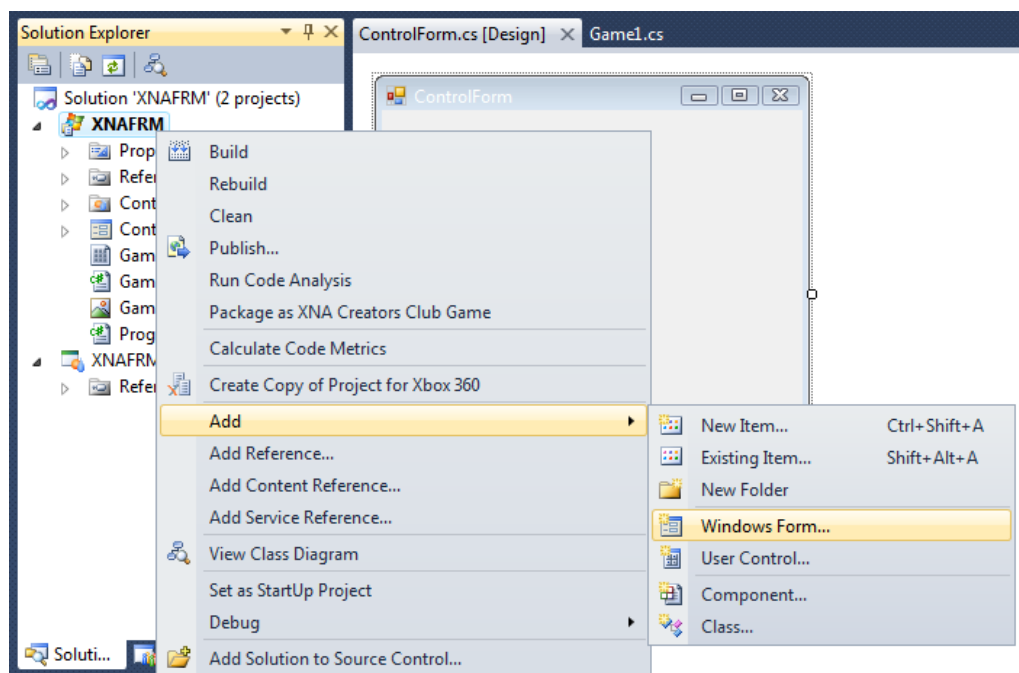
Βέβαια υπάρχουν δύο βασικοί τρόποι ώστε να συνδέσουμε τα γραφικά μας, την διαχείριση τους δηλαδή, με την φόρμα.

Ο ένας τρόπος ο οποίος και δεν χρησιμοποιήθηκε σε αυτήν την πτυχιακή, είναι να ξεκινήσουμε μέσα από μία φόρμα και να ενσωματώσουμε σε αυτήν ένα παράθυρο γραφικών. Δηλαδή με ένα απλό **PictureBox** ή οποιονδήποτε άλλο **Container**, δημιουργούμε ένα αντικείμενο **XNA** και το ενσωματώνουμε μέσα στην φόρμα μας. Αυτός ο τρόπος βέβαια δεν είναι και ο καταλληλότερος γιατί τον “κύριο” λόγο τον έχει το παράθυρο της φόρμας και όχι των γραφικών. Δηλαδή για παράδειγμα η θέση του ποντικιού η οποία και θέλουμε να τη χρησιμοποιήσουμε για τα γραφικά μας θα έχει τιμές από το στιγμιότυπο της φόρμας και όχι από το στιγμιότυπο της μηχανής γραφικών. Κάτι τέτοιο θα δημιουργούσε μεγάλα προβλήματα και για να τα ξεπεράσουμε αυτά θα έπρεπε να γράψουμε επιπλέον κώδικα. Επίσης το πρόγραμμά μας θα αναγνωριζόταν ως μία απλή εφαρμογή **Win32** και όχι ως εφαρμογή γραφικών το οποίο θα δημιουργούσε περαιτέρω προβλήματα και θα χρειαζόταν περαιτέρω κώδικας για να λυθούν. Ειδικά σε συστήματα όπως είναι τα **Windows Vista** και τα **Windows 7** κρίνεται απαραίτητο να ξέρει το σύστημα εάν μία εφαρμογή που τρέχει είναι απλή εφαρμογή ή εφαρμογή γραφικών, γιατί τα περιβάλλοντα αυτά χρησιμοποιούν την κάρτα γραφικών για να επιτελέσουν διάφορα γραφικά στην επιφάνεια εργασίας τους. Εάν ξέρει το λειτουργικό σύστημα ότι μία εφαρμογή που τρέχει είναι εφαρμογή γραφικών, μπορεί να απενεργοποιήσει τα εφέ αυτά ή ακόμα και να κάνει βελτιώσεις ώστε να δώσει

προτεραιότητα για το πρόγραμμα γραφικών στη κάρτα γραφικών παρά στα εφέ του περιβάλλοντος.

Ο δεύτερος τρόπος, ο οποίος είναι και ευκολότερος και πιο σωστός, είναι ενώ έχουμε δημιουργήσει μία εφαρμογή γραφικών **XNA**, να δηλώσουμε και να χρησιμοποιήσουμε αντικείμενα φόρμας **Win32** τα οποία και θα τα ορίσουμε να διαχειρίζονται τα γραφικά μας. Αυτός ο τρόπος βέβαια είναι λίγο πιο περίπλοκος από τον πρώτο διότι θα πρέπει η μηχανή γραφικών να “ακούει” τις αλλαγές που γίνονται στα κουμπιά της φόρμας, αλλά και η φόρμα θα πρέπει να “εξαναγκάζει” την μηχανή γραφικών για κάποιες ενέργειες, όταν αυτό κρίνεται απαραίτητο (force). Πάμε, λοιπόν, να δούμε πώς θα μπορέσουμε να τα κάνουμε όλα αυτά.

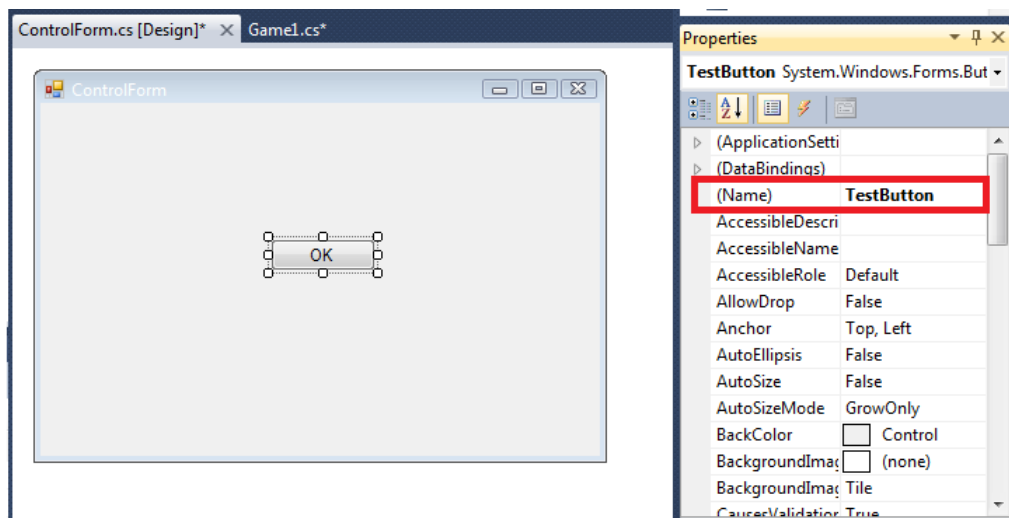
Αφού έχουμε δημιουργήσει μία καινούρια εφαρμογή XNA και έχει δημιουργηθεί ο βασικός κώδικας αυτόματα, θα πάμε αριστερά στο παράθυρο **Solution Explorer** και θα πατήσουμε δεξί κλικ επάνω στο όνομα του Project και έπειτα επιλέγουμε διαδοχικά **Add->Windows Form**. Αυτή η ενέργεια η οποία φαίνεται και στην εικόνα παρακάτω θα προσθέσει μία φόρμα στην εφαρμογή μας. Επιλέγουμε για ευκολία το όνομα “**ControlForm**” και πατάμε **Add**.



**Εικόνα 39.** Προσθήκη φόρμας στην εφαρμογή.

Θα παρατηρήσουμε πως στα **References** θα προστεθούν κάποιες βιβλιοθήκες και αυτό είναι λογικό. Είναι οι βιβλιοθήκες οι οποίες είναι απαραίτητες για να χρησιμοποιήσουμε

αντικείμενα φόρμας στην εφαρμογή μας. Αφού, λοιπόν, έχει δημιουργηθεί ένα καινούριο αντικείμενο φόρμας στο Project μας, πατάμε διπλό κλικ επάνω του για να ανοίξει το παράθυρο διαχείρισης δεξιά. Σε αυτό το σημείο μπορούμε να εισάγουμε **Buttons** **TextBoxes** και άλλα, όπως κάνουμε με τις τυπικές εφαρμογές **WindowsFormsApplications**. Για να δούμε τώρα την τεχνική διασύνδεσης φόρμας με γραφικά το μόνο που θα κάνουμε είναι να προσθέσουμε πάνω στη φόρμα ένα κουμπί με το όνομα **“TestButton”** και με κείμενο ότι θέλουμε. Αφού τελειώσουμε με αυτό, το αποτέλεσμα πρέπει να φαίνεται παρόμοιο με την παρακάτω εικόνα:



**Εικόνα 40.** Προσθήκη κουμπιού και αλλαγή ονόματος.

Σε αυτό το σημείο θα πρέπει να καταλαβαίνουμε πότε το κουμπί θα είναι πατημένο και πότε όχι. Αυτό μπορεί να γίνει με πάρα πολλούς τρόπους. Σε αυτήν την πτυχιακή χρησιμοποιήθηκαν διάφοροι τρόποι. Εδώ θα δούμε έναν απλό. Θα προσθέσουμε ένα αντικείμενο **bool** στην κλάση της φόρμας το οποίο και θα αλλάζει σύμφωνα με τις ενέργειες που κάνουμε πάνω στο κουμπί. Όταν το κουμπί θα είναι πατημένο αυτή η μεταβλητή θα γίνεται αληθής και σε άλλη περίπτωση ψευδής. Στην κλάση της φόρμας, λοιπόν, γράφουμε τον παρακάτω κώδικα:

```

10 namespace XNAFRM
11 {
12     public partial class ControlForm : Form
13     {
14         private bool isButtonPressed = false;
15
16         public ControlForm()
17         {
18             InitializeComponent();
19         }
20
21         public bool isFormButtonPressed()
22         {
23             return isButtonPressed;
24         }
25
26         private void TestButton_MouseDown(object sender, MouseEventArgs e)
27         {
28             isButtonPressed = true;
29         }
30
31         private void TestButton_MouseUp(object sender, MouseEventArgs e)
32         {
33             isButtonPressed = false;
34         }
35     }
36 }

```

**Κώδικας 23.** Κώδικας για ενεργοποίηση συμβάντος του κουμπιού **TestButton**.

Στη γραμμή 14, χρησιμοποιούμε μία μεταβλητή που, όπως είπαμε, θα αλλάζει σύμφωνα με το εάν το κουμπί είναι πατημένο είτε όχι. Την ορίζουμε ως **private** έτσι ώστε να μπορούν μόνο τα αντικείμενα της φόρμας να την βλέπουν. Με τη βοήθεια της συνάρτησης όμως στη γραμμή 21, μπορούμε να λαμβάνουμε την τιμή αυτή και μέσω της μηχανής γραφικών, όπως θα δούμε αργότερα, για αυτό και η συνάρτηση ορίζεται ως **public**. Στην γραμμή 26 και 31 δηλώνουμε δύο **events**, όταν πατάμε το κλικ και όταν αφήνουμε το κλικ αντίστοιχα τα οποία και αλλάζουν την τιμή της μεταβλητής **isButtonPressed** σε **true** και **false** αντίστοιχα. Αφού έχουμε τελειώσει όλες τις απαραίτητες διαδικασίες που αφορούν την φόρμα είναι ώρα να πάμε στην μηχανή γραφικών και να γράψουμε λίγο κώδικα και εκεί.

Εφόσον χρειαζόμαστε μόνο ένα αντικείμενο φόρμας το οποίο και θα τρέχει για όλη τη διάρκεια της ζωής του προγράμματος, θα δηλώσουμε ένα αντικείμενο τύπου **ControlForm** στην συνάρτηση **Initialize()** της μηχανής γραφικών και θα το εμφανίσουμε, όπως φαίνεται παρακάτω:

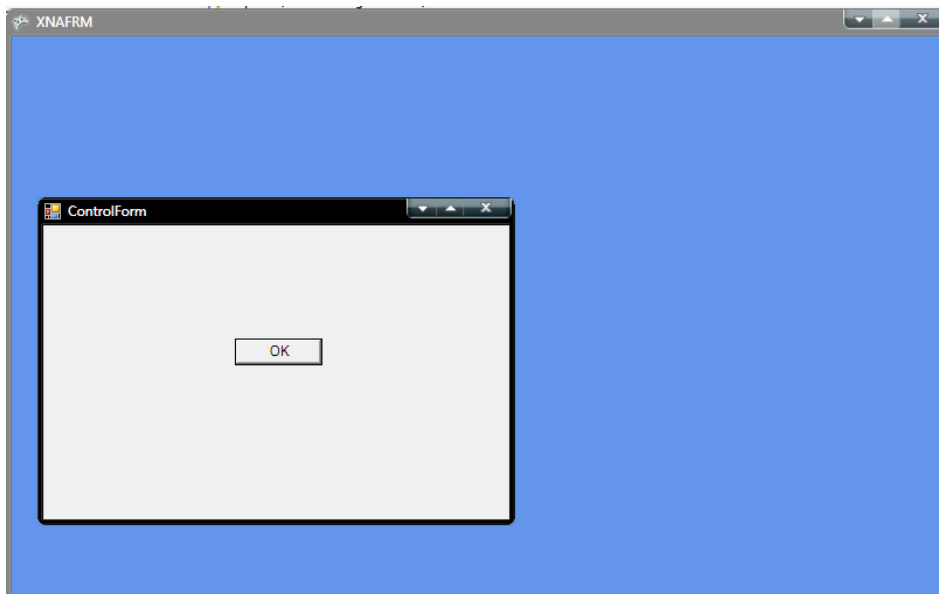
```

14 public class Game1 : Microsoft.Xna.Framework.Game
15 {
16     GraphicsDeviceManager graphics;
17     SpriteBatch spriteBatch;
18     ControlForm ctrlForm;
19
20     public Game1()
21     {
22         graphics = new GraphicsDeviceManager(this);
23         Content.RootDirectory = "Content";
24     }
25
26     protected override void Initialize()
27     {
28         ctrlForm = new ControlForm();
29         ctrlForm.Show();
30         base.Initialize();
31     }
32

```

**Κώδικας 24.** Δημιουργία και εμφάνιση ενός αντικειμένου φόρμας.

Βλέπουμε πως στην γραμμή 18 δηλώνουμε ένα αντικείμενο τύπου **ControlForm** και στην γραμμή 28 δεσμεύουμε μνήμη για αυτό. Στην επόμενη γραμμή, την 29 το εμφανίζουμε. Εάν τρέξουμε τώρα αυτό το πρόγραμμα θα έχουμε το παρακάτω αποτέλεσμα:



**Εικόνα 41.** Η εφαρμογή γραφικών με φόρμα.

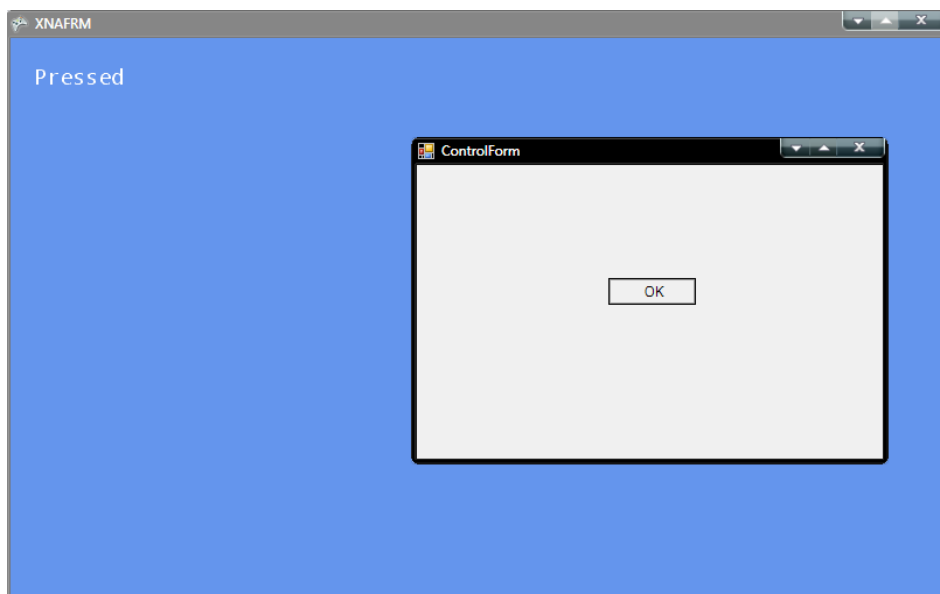
Μέχρι εδώ όλα φαίνονται πολύ καλά. Όμως εάν πατήσουμε το κουμπί **OK** δεν θα συμβεί απολύτως τίποτα διότι το μόνο που κάνει το κουμπί αυτό είναι να αλλάζει την τιμή μίας μεταβλητής. Πάμε να δούμε λοιπόν πώς μπορούμε να “διαβάσουμε” την τιμή της μεταβλητής αυτής και να κάνουμε τις απαραίτητες ενέργειες ανάλογα με την κατάσταση που βρίσκεται το κουμπί. Στην συνάρτηση **Draw()** της μηχανής γραφικών,

λοιπόν, θα ελέγχουμε κάθε φορά εάν το κουμπί είναι σε κατάσταση **Down** ή **Up**. Ανάλογα με την κατάσταση αυτή θα αλλάζει και το κείμενο που εμφανίζεται μέσα στη μηχανή γραφικών. Το πώς μπορούμε να εμφανίσουμε κείμενο έχει καλυφθεί σε προηγούμενο κεφάλαιο και δεν θα γίνει αναφορά εδώ.

```
51 protected override void Draw(GameTime gameTime)
52 {
53     GraphicsDevice.Clear(Color.CornflowerBlue);
54
55     spriteBatch.Begin();
56
57     if (ctrlForm.IsFormButtonPressed() )
58     {
59         spriteBatch.DrawString(sprFont, "Pressed", new Vector2(20f, 20f), Color.White);
60     }
61     else
62     {
63         spriteBatch.DrawString(sprFont, "Not Pressed", new Vector2(20f, 20f), Color.White);
64     }
65
66     spriteBatch.End();
67
68     base.Draw(gameTime);
69 }
70 }
71 }
```

**Κώδικας 25.** Κώδικας ελέγχου κουμπιού, μέσω της μηχανής γραφικών.

Φαίνεται καθαρά στο προηγούμενο κομμάτι κώδικα πως ελέγχουμε την κατάσταση του κουμπιού με την βοήθεια της συνάρτησης στη γραμμή 57 **isFormButtonPressed()** την οποία και υλοποιήσαμε πιο πριν. Εάν τρέξουμε τώρα το πρόγραμμα αυτό και πατήσουμε το κουμπί θα δούμε ότι το κείμενο θα αλλάξει σε **Pressed**, ενώ εάν το αφήσουμε θα αλλάξει σε **Not Pressed** όπως φαίνεται παρακάτω:



**Εικόνα 42.** Το τελικό πρόγραμμα ελέγχου κουμπιού από τη μηχανή γραφικών.

Είδαμε λοιπόν πως με πολύ εύκολο τρόπο μπορούμε να συνδέσουμε μία φόρμα με την μηχανή γραφικών και ανάλογα με τις ενέργειες που κάνουμε στην φόρμα αυτή να παίρνουμε τα ανάλογα αποτελέσματα. Στο χέρι μας βέβαια είναι το τι θα κάνει αυτή η φόρμα και το πώς θα είναι. Οι πιθανότητες είναι απεριόριστες!

Αφού λοιπόν έχουν καλυφθεί τα βασικά κομμάτια εισαγωγής στην μηχανή γραφικών **XNA** και στο **Microsoft Visual Studio**, στο επόμενο κεφάλαιο θα γίνει παρουσίαση των κλάσεων που χρησιμοποιήθηκαν για να παραχθεί το τελικό αποτέλεσμα το οποίο και είναι σχεδίαση τοπίου 3D και η διαχείριση του μέσω μίας φόρμας.

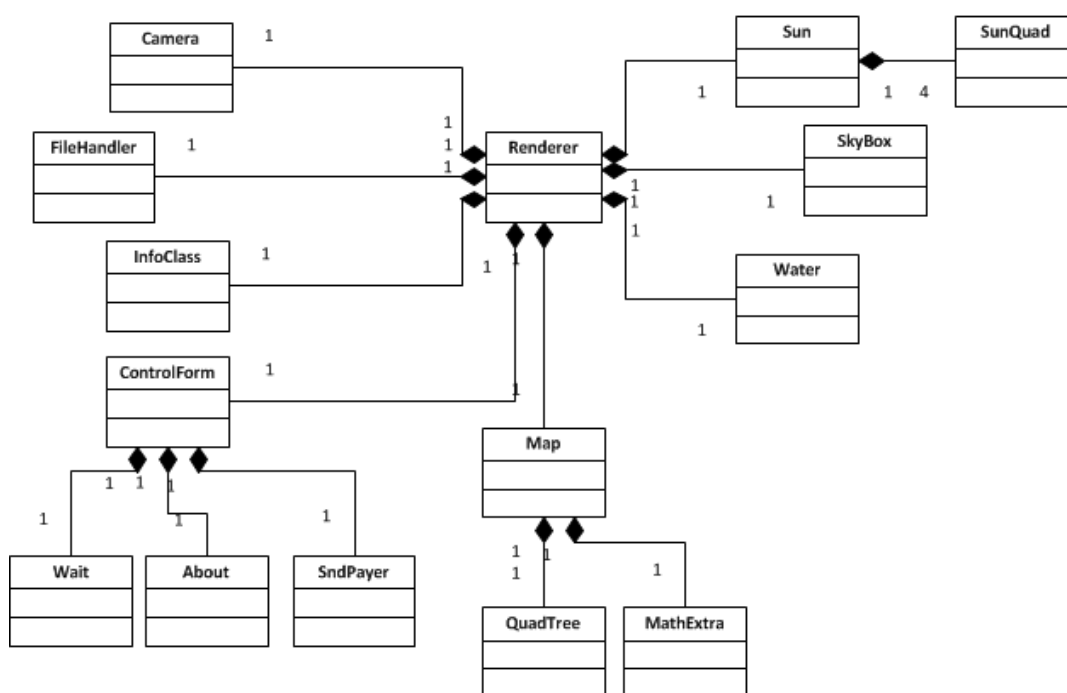




The logo for XNA (Xbox Game Studio) features the lowercase letters 'xna' in a grey, 3D-style font. The letter 'x' is stylized with an orange and red gradient, resembling a pencil or a stylized 'x' shape.

Ο κώδικας αυτής της πτυχιακής εργασίας, έχει αντικειμενοστραφή σχεδίαση και περιλαμβάνει 13 κλάσεις, κάθε μία από τις οποίες αντιπροσωπεύει είτε κάποιο κομμάτι του τοπίου, είτε μία λειτουργία επί αυτού. Σε αυτό το κεφάλαιο θα γίνει αναφορά σε αυτές τις κλάσεις και το τι περιέχει – τι σκοπό έχει – η κάθε μία και στα επόμενα κεφάλαια θα γίνει ανάλυση κάθε μίας κλάσης.

Πέρα από αυτές τις 13 κλάσεις, η πτυχιακή εργασία περιέχει και τρεις φόρμες από τις οποίες η μία είναι ελέγχου, η άλλη εμφάνισης πληροφοριών και η τελευταία υλοποιεί τον κέρσορα αναμονής που εμφανίζεται στην οθόνη. Οι 13 κύριες κλάσεις του προγράμματος χρησιμοποιούν επίσης και τον επεξεργαστή περιεχομένου. Τα αντικείμενα τα οποία χρησιμοποιούνται από τις κλάσεις – όσες χρησιμοποιούν αντικείμενα – είναι διαχωρισμένα σε φακέλους για πιο εύκολη ανάγνωση του προγράμματος. Παρακάτω παρουσιάζονται σχηματικά οι κλάσεις μαζί με τον επεξεργαστή περιεχομένου, ενώ στην συνέχεια θα δοθεί μία εξήγηση για το κάθε ένα αντικείμενο από αυτά:

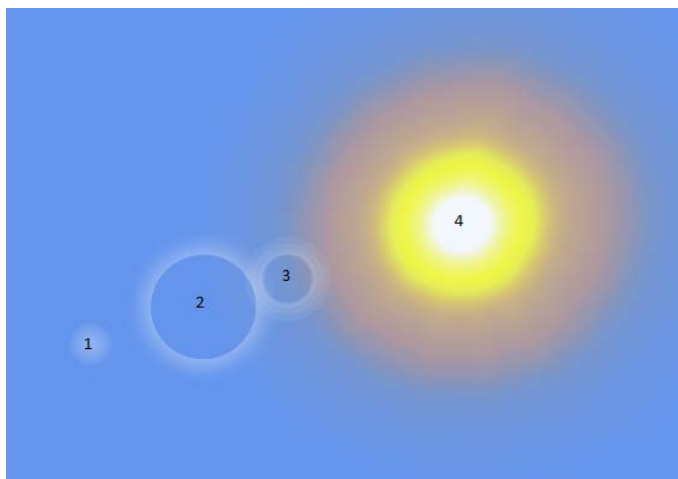


**Εικόνα 43.** Οι κλάσεις του προγράμματος.

Όπως φαίνεται και στο διάγραμμα UML, η κυρίως κλάση του προγράμματος, είναι η κλάση **Renderer**, η οποία και είναι υπεύθυνη για την αρχική δημιουργία των υπολοίπων αντικειμένων αρχικά και για την ανανέωση (Update) και εμφάνιση αυτών (Draw) αργότερα.

Αρχικά, λοιπόν, η κλάση **Renderer** χρησιμοποιεί το αντικείμενο της κάμερας με όνομα **Camera**. Σε αυτό το σημείο πρέπει να πούμε ότι μία κάμερα για ένα παιχνίδι τρισδιάστατο είναι ζωτικής σημασίας. Μέσα σε αυτή την κλάση, όλα τα αντικείμενα μας όπως είναι το νερό, ο ουρανός, τα σύννεφα και άλλα ζωγραφίζονται με βάση αυτήν την κάμερα. Εδώ μέσα περιέχονται έννοιες περιστροφής κάμερας, πεδίο ορατότητας και άλλα. Σε επόμενο κεφάλαιο θα αναλυθούν όλα τα παραπάνω.

Έπειτα η κλάση του ήλιου με όνομα **Sun** χρησιμοποιείται από την κύρια κλάση. Αυτή η κλάση είναι υπεύθυνη για την εμφάνιση του ήλιου στην πτυχιακή. Η κλάση **sun** χρησιμοποιεί με την σειρά της 4 κλάσεις **SunQuad** η κάθε μία από τις οποίες είναι υπεύθυνη για την εμφάνιση ενός κομματιού από τα 4 που απαρτίζουν τον ήλιο. Δηλαδή τον ήλιο τον ίδιο και τις 3 αντανάκλασεις του όπως φαίνεται στο παρακάτω σχήμα:



**Εικόνα 44.** Τα 4 κομμάτια τα οποία απαρτίζουν τον ήλιο.

Έπειτα, υπάρχει μία κλάση που χρησιμοποιείται από την κύρια με όνομα **SkyBox**. Αυτή η κλάση είναι υπεύθυνη για την εμφάνιση του ουρανού μέσα στο παιχνίδι. Αν και ο ουρανός φαίνεται πολύ ρεαλιστικός, δεν είναι τίποτα άλλο από έξι εικόνες, οι οποίες φορτώνονται με την βοήθεια του **Content Manager**. Ο ουρανός φαίνεται στην παρακάτω εικόνα:



**Εικόνα 45.** Ο ουρανός του παιχνιδιού.

Παρακάτω παρουσιάζεται ένα από τα σημαντικότερα κομμάτια της πτυχιακής, το κομμάτι του τοπίου ή του χάρτη, όπως πολύ συχνά αναφέρεται στην ορολογία των παιχνιδιών. Η εμφάνιση και η επεξεργασία του χάρτη της πτυχιακής γίνεται με την κλάση **Map**, η οποία χρησιμοποιεί την κλάση **QuadTree**, διότι έχει κατασκευαστεί με τετραδικό δέντρο – κάτι για το οποίο θα μιλήσουμε στην συνέχεια – και την κλάση **MathExtra** η οποία περιέχει κάποιες μαθηματικές πράξεις που είναι χρήσιμες για την επεξεργασία του τοπίου. Ένας τυχαίος χάρτης της πτυχιακής φαίνεται παρακάτω:



**Εικόνα 46.** Ένας τυχαίος χάρτης από την πτυχιακή.

Φυσικά σε έννοιες υψηλότερες, κατά ένα τρόπο ο χάρτης “εξαρτάται” από τον ήλιο, αλλά και αυτός με τη σειρά του από τον χάρτη. Ο χάρτης που παρουσιάστηκε στην προηγούμενη εικόνα, αν και έχει μέσα την έννοια του ύψους δεν μπορούμε να το καταλάβουμε, γιατί συχνά τα μάτια μας κάνουν διάφορα “παιχνίδια” με το φάσμα των ακτινοβολιών. Σε περίπτωση όμως που φωτίσουμε εμείς τον χάρτη, τότε θα αρχίσουμε να αντιλαμβανόμαστε την τρίτη διάσταση, όπως φαίνεται στην παρακάτω εικόνα:



**Εικόνα 47.** Ο χάρτης φωτισμένος.

Προχωρώντας, λοιπόν, έχουμε την κλάση η οποία είναι χρήσιμη για την κατασκευή του νερού στον χώρο μας. Αυτή η κλάση είναι η κλάση **Water**, η οποία έχει την δυνατότητα να “αντικατοπτρίζει”, να ζωγραφίζει όλα όσα έχουμε κατασκευάσει στον χώρο μας, επάνω σε δύο τρίγωνα, κάτι το οποίο θα αναφέρουμε αργότερα πως μπορούμε να το επιτύχουμε. Ένα τυχαίο στιγμιότυπο του νερού φαίνεται παρακάτω:



**Εικόνα 48.** Το νερό τη πτυχιακής.

Πλέον και με την αναφορά στα βασικά συστατικά τα οποία απαρτίζουν το παιχνίδι, έχουμε τελειώσει με τις κύριες κλάσεις του προγράμματος. Στην συνέχεια έχουμε την κλάση **FileHandler** η οποία μπορεί να αποθηκεύει την κατάσταση του τοπίου που βρίσκεται εκείνη τη στιγμή, αλλά και να την ανακτά. Επίσης, χρησιμοποιείται μία τεχνική αποτροπής αλλαγής αρχείου, κάτι το οποίο θα το αναφέρουμε στην συνέχεια. Έπειτα η κλάση **InfoClass**, έχει την δυνατότητα να εμφανίζει πληροφορίες στην οθόνη μας με την χρήση μίας απλής βούρτσας, όπως είχαμε αναφέρει σε προηγούμενο κεφάλαιο. Τέλος, έχουμε την κλάση ελέγχου του τρισδιάστατου τοπίου μας. Αυτή η κλάση ελέγχει τα πάντα μέσα στην πτυχιακή. Μπορούμε με αυτήν να κατασκευάσουμε καινούρια τοπία, να αποθηκεύσουμε, να ανακτήσουμε, να επεξεργαστούμε το τοπίο, το νερό και άλλα. Αυτή η κλάση είναι η κλάση **ControlForm** την οποία και χρησιμοποιεί η κύρια κλάση, η **Renderer**, (όπως είπαμε και σε προηγούμενο κεφάλαιο χρησιμοποιήθηκε η απλοϊκή ενσωμάτωση φόρμας μέσα σε γραφικά). Η κλάση **ControlForm** με την σειρά της χρησιμοποιεί τρεις κλάσεις. Μία από αυτές είναι η **Wait** η οποία μας εμφανίζει έναν κέρσορα αναμονής στην οθόνη. Μία άλλη είναι η **About** κλάση, η οποία εμφανίζει μία φόρμα όπου περιέχονται πληροφορίες για την εργασία αυτή. Τέλος η κλάση **SndPlayer** είναι η κλάση η οποία είναι ικανή να παίξει αρχεία ήχου και χρησιμοποιείται και αυτή από την κλάση **ControlForm** – ελέγχου.

Μετά από μία σύντομη αναφορά στις κλάσεις που χρησιμοποιήθηκαν, θα πρέπει να πούμε πως προγραμματιστικά φαίνονται αρκετά καλές. Θα πρέπει όμως να κοιτάμε και πάντα το αποτέλεσμα, δηλαδή με άλλα λόγια επειδή το αποτέλεσμα πρέπει να είναι ένα τοπίο με νερό, ουρανό και άλλα αντικείμενα, δεν φτάνει μόνο η κατασκευή των κλάσεων, αλλά και η χρήση τους και, σημαντικότερα, η εμφάνισή τους με την σωστή σειρά. Για παράδειγμα σε περίπτωση που ο ουρανός ζωγραφιζόταν μετά τον ήλιο, θα είχαμε λογικό σφάλμα διότι ο πρώτος θα έκρυβε τον δεύτερο, κάτι το οποίο θα ήταν ανεπιθύμητο. Επίσης θα έπρεπε να γνωρίζουμε κάθε φορά εάν υπάρχει τοπίο ώστε να μην ζωγραφίζονται σκιές από τον ήλιο χωρίς λόγο. Ένα τέτοιο αποτέλεσμα θα ήταν το λιγότερο αστείο.

Σε επόμενα κεφάλαια, θα γίνει πλήρης αναφορά στις επιμέρους κλάσεις που χρησιμοποιήθηκαν.



xna





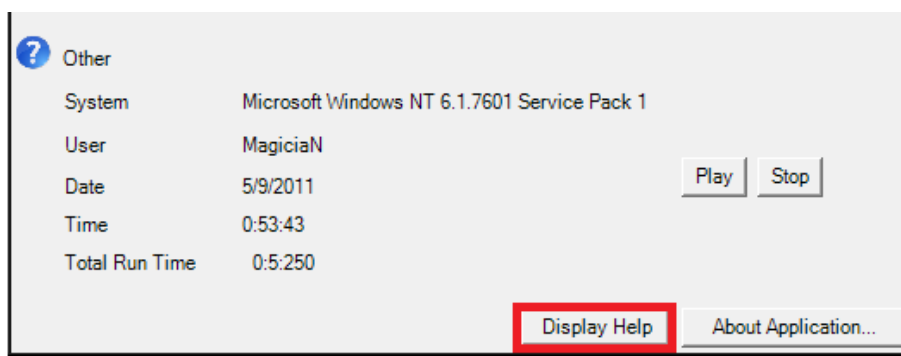
Η κλάση πληροφοριών - InfoClass -, είναι μία χρήσιμη κλάση η οποία ως σκοπό έχει την εμφάνιση βοήθειας στον χρήστη. Επίσης, επιτελεί και άλλες λειτουργίες όπως αυτή της εμφάνισης σκηνών ανά δευτερόλεπτο. Αυτή η μέτρηση των σκηνών είναι γνωστή ως **FPS Counter**. Τέλος μας εμφανίζει πληροφορίες για τον τρέχοντα χρήστη ο οποίος τρέχει το πρόγραμμα, για το λειτουργικό σύστημα, την ώρα και άλλα.

Αρχικά, θα ασχοληθούμε με την βοήθεια που εμφανίζεται στον χρήστη. Ένα στιγμιότυπο της βοήθειας από την πτυχιακή φαίνεται στην παρακάτω εικόνα:

```
Press and hold right mouse click to enter first person mode camera.
Then you can navigate through the world by moving the mouse.
When entering at first person mode,you can use keyboard keys
for moving yourself around the world.
W:Move forward.
A:Move left.
S:Move backward.
D:Move right.
Z:Move down.
X:Move up.
```

**Εικόνα 49.** Εμφάνιση βοήθειας.

Αυτή η λειτουργία δεν χρησιμοποιεί τίποτα άλλο παρά ένα φόντο από τον Content Manager, και μία βούρτσα, όπως είδαμε σε προηγούμενο κεφάλαιο. Πατώντας, λοιπόν, κάποιο κουμπί επάνω στην φόρμα, καλούμε την συνάρτηση **drawHelp()** της οποίας η λειτουργία είναι να ξεκινά την ζωγραφική δυσδιάστατων γραφικών (`spriteBatch.Begin()`), να ζωγραφίζει, και να κάνει έξοδο από την ζωγραφική δυσδιάστατων γραφικών (`spriteBatch.End()`). Το κουμπί της φόρμας είναι το παρακάτω:



**Εικόνα 50.** Το κουμπί που σηματοδοτεί την εμφάνιση βοήθειας στον χρήστη.

Στην συνέχεια θα αναφέρουμε πώς δουλεύει ένας Fps Counter. Η μέτρηση των σκηνών αρχικά να πούμε πως είναι πολύ σημαντική σε αρκετές περιπτώσεις διότι μπορούμε εμείς σαν χρήστες να διαπιστώσουμε εάν το παιχνίδι μας τρέχει “ομαλά”, κοιτώντας τις

σκηνές ανά δευτερόλεπτο. Σε περίπτωση που πέφτουν οι σκηνές σημαίνει ότι κάτι δεν πάει καλά είτε με το λογισμικό του συστήματος (Drivers κάρτας γραφικών), είτε με το υλικό (παλαιά κάρτα γραφικών). Επίσης, οι μειωμένες σκηνές μπορεί να είναι αιτία πολλών άλλων πραγμάτων. Πάντως παραμένει το σημαντικότερο μέτρο σύγκρισης. Οι εταιρίες ελέγχου υλικού τέλος, χρησιμοποιούν τις σκηνές που εμφανίζει ένα παιχνίδι πάνω σε διαφορετικές κάρτες γραφικών έτσι ώστε να τις συγκρίνει μεταξύ τους. Οι σκηνές ανά δευτερόλεπτο για τα παιχνίδια, είναι κλειδωμένες στις 60. Υπάρχουν τρόποι βέβαιοι να τις ξεκλειδώσουμε, αλλά κάτι τέτοιο θα έκανε το παιχνίδι μας να μην τρέχει ομαλά (θα έτρεχε πολύ γρήγορα).

Η λογική για να δουλέψει ένας τέτοιος μετρητής είναι απλή: Τοποθετούμε έναν μετρητή ο οποίος μετράει πόσες φορές καλείται η συνάρτηση `draw()`, η οποία ζωγραφίζει αντικείμενα στο τοπίο. Επίσης στην συνάρτηση ανανέωσης `update()`, θα τοποθετήσουμε και έναν μετρητή χιλιοστών του δευτερολέπτου ο οποίος μόλις φτάσει στο 1000 (δηλαδή ένα δευτερόλεπτο), θα σηματοδοτεί την εμφάνιση του αθροίσματος του μετρητή σκηνών. Στην συνέχεια ο μετρητής μηδενίζεται. Παρακάτω εμφανίζονται σε κώδικα όλα όσα αναφέρθηκαν:

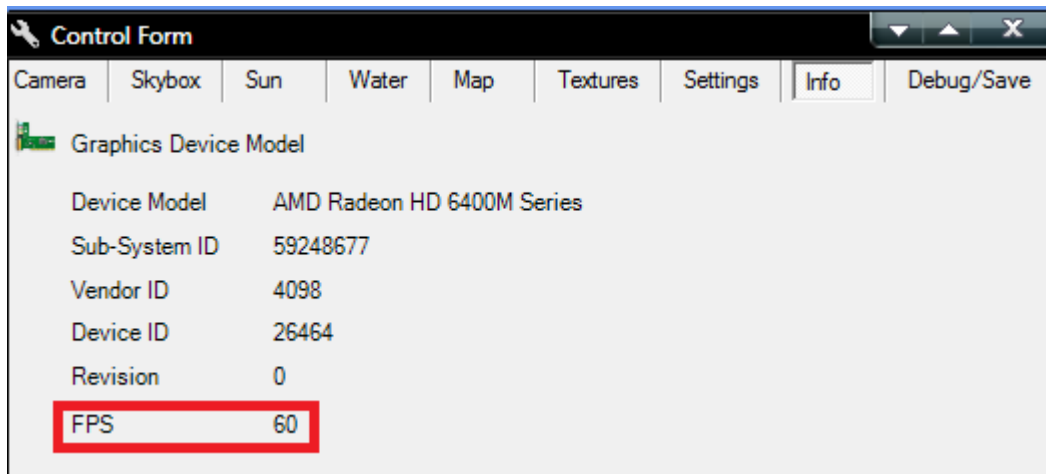
```
54 public void addFrame()
55 {
56     fps++;
57 }

29 public void update(GameTime gameTime)
30 {
31     totalRunTime += (float)gameTime.ElapsedGameTime.Milliseconds;
32
33     if (totalRunTime >= 1000.0f)
34     {
35         Renderer.controlForm.InfoGFPS.Text = fps.ToString();
36         totalRunTime = 0;
37         fps = 0;
38     }
```

**Κώδικας 26.** Ο αθροιστής σκηνών.

Οπότε, στην συνάρτηση `draw()` θα καλούμε εμείς την συνάρτηση `addFrame()` στην γραμμή 54, έπειτα στην γραμμές 31 αθροίζουμε τον χρόνο και στην επόμενη γραμμή, την 33 ελέγχουμε εάν έχουμε ξεπεράσει το δευτερόλεπτο και κάνουμε τις απαραίτητες ενέργειες για της εμφάνισης αυτής της πληροφορίας στην φόρμα.

Στην επόμενη εικόνα φαίνεται ένα στιγμιότυπο από τον μετρητή αυτό πάνω στην πτυχιακή:

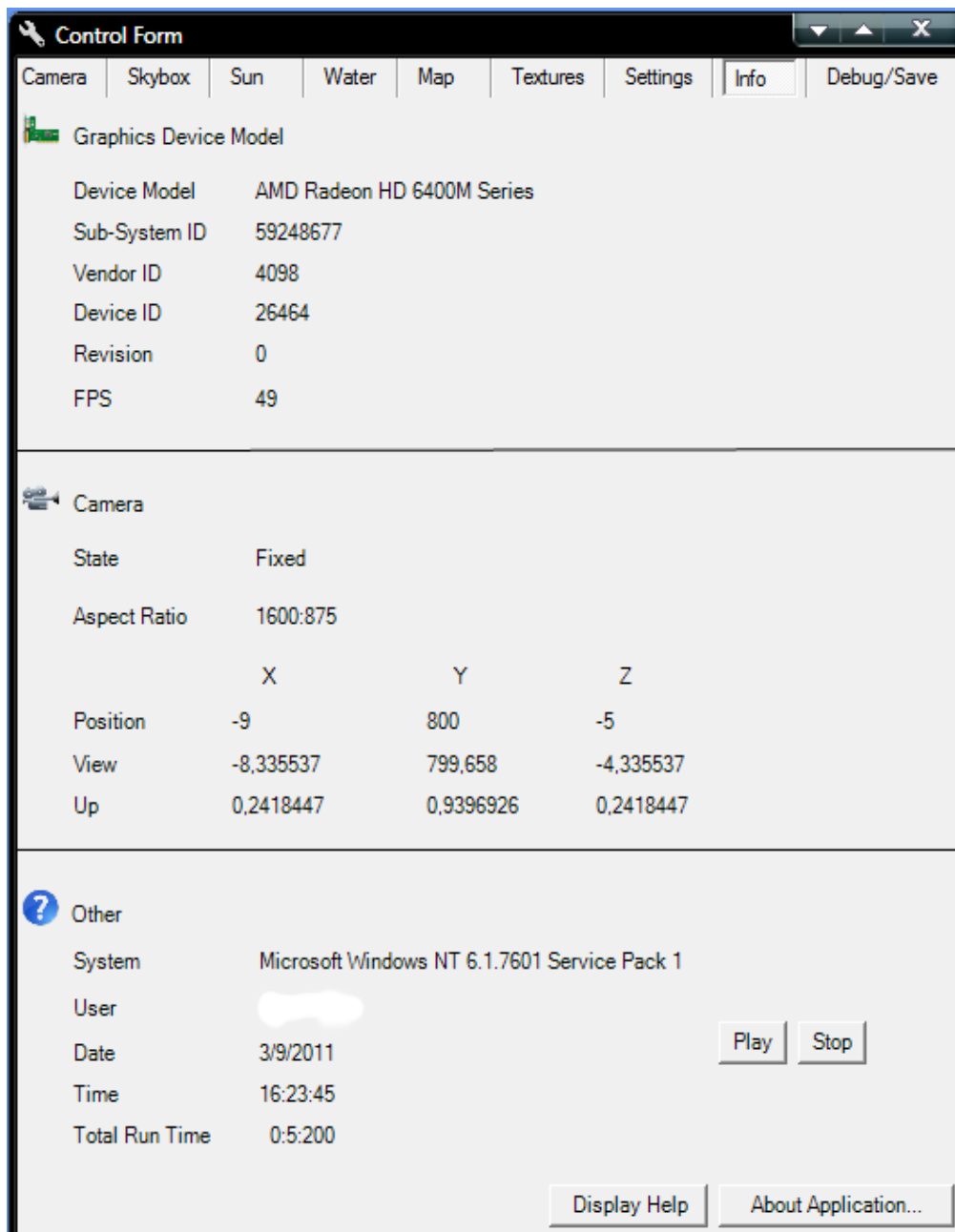


**Εικόνα 51.** Ο μετρητής σκηνών ανά δευτερόλεπτο.

Ακόμα, μία τελευταία λειτουργία την οποία επιτελεί αυτή η κλάση είναι, όπως είπαμε, η εμφάνιση ακόμα περισσότερων πληροφοριών. Αυτές συνοψίζονται παρακάτω:

- Εμφάνιση πληροφοριών που αφορούν την κάρτα γραφικών, όπως έκδοση, κατασκευαστή, μοντέλο και άλλα.
- Εμφάνιση πληροφοριών της κάμερας, σε τι κατάσταση είναι (στάσιμη ή κινούμενη), το σημείο που βρίσκεται, το ύψος και το επάνω μέρος της.
- Εμφάνιση πληροφοριών που αφορούν το σύστημα, όπως είναι ο χρήστης ο οποίος τρέχει το πρόγραμμα την δεδομένη χρονική στιγμή, ο χρόνος που πέρασε από την έναρξη του προγράμματος, η τρέχουσα ώρα και η ημερομηνία.

Ένα παράδειγμα εμφάνισης πληροφοριών επάνω στην φόρμα φαίνεται παρακάτω:



**Εικόνα 52.** Εμφάνιση πληροφοριών στην φόρμα.

Η εμφάνιση πληροφοριών γίνεται με έτοιμες συναρτήσεις του Microsoft XNA. Πιο συγκεκριμένα οι πληροφορίες της κάρτας γραφικών αντλούνται μέσω του αντικειμένου **GraphicsDeviceManager**, που είναι ο χειριστής της κάρτας γραφικών. Οι πληροφορίες αυτές συνοψίζονται στον παρακάτω πίνακα (η συνάρτηση ToString() μετατρέπει την πληροφορία σε κείμενο):

Συναρτήσεις XNA	ΛΕΙΤΟΥΡΓΙΑ
<code>Renderer.graphics.GraphicsDevice.Adapter.Description.ToString();</code>	Περιγραφή προσαρμογέα
<code>Renderer.graphics.GraphicsDevice.Adapter.DeviceId.ToString();</code>	Αναγνωριστικό προσαρμογέα
<code>Renderer.graphics.GraphicsDevice.Adapter.Revision.ToString();</code>	Έκδοση προσαρμογέα
<code>Renderer.graphics.GraphicsDevice.Adapter.SubSystemId.ToString();</code>	Αναγνωριστικό CHIP προσαρμογέα
<code>Renderer.graphics.GraphicsDevice.Adapter.VendorId.ToString();</code>	Αναγνωριστικό κατασκευαστή προσαρμογέα

**Κώδικας 27.** Οι συναρτήσεις για άντληση πληροφοριών από την κάρτα γραφικών.

Στην συνέχεια για να πάρουμε την έκδοση του λειτουργικού συστήματος στο οποίο τρέχει η εφαρμογή, καλούμε την εξής συνάρτηση:

```
System.Environment.OSVersion.ToString();
```

**Κώδικας 28.** Κώδικας λήψης έκδοσης λειτουργικού συστήματος.

Με την ίδια λογική μπορούμε να δεσμεύσουμε το όνομα του χρήστη ο οποίος χρησιμοποιεί την εφαρμογή:

```
System.Windows.Forms.SystemInformation.UserName;
```

**Κώδικας 29.** Ο τρέχων χρήστης της εφαρμογής.

Τέλος, για την περίπτωση της ώρας θα χρησιμοποιήσουμε το έτοιμο αντικείμενο **DateTime** της C#:

```
DateTime.Now.Day + "/" + DateTime.Now.Month + "/" + DateTime.Now.Year;
```

**Κώδικας 30.** Η ημερομηνία του συστήματος διαχωρισμένη με τον χαρακτήρα "/".



χρησ



Η αναπαραγωγή αρχείων ήχου, είναι πλέον από τα σημαντικότερα κομμάτια των παιχνιδιών σήμερα. Δεν είναι επίσης τυχαίο το γεγονός ότι, τα παιχνίδια συνήθως που κυκλοφορούν στην αγορά, συνοδεύονται με ένα CD το οποίο περιέχει τους ήχους – soundtracks – και μερικές φορές ακόμα και τα ηχητικά εφέ του παιχνιδιού. Γενικότερα θα μπορούσαμε να πούμε ότι δαπανώνται αρκετοί πόροι και χρήματα για την μουσική επένδυση ενός παιχνιδιού, καθώς οι ήχοι και η μουσική προσδίδουν διαδραστικότητα και ζωντάνια στα παιχνίδια.

Η αναπαραγωγή αρχείων ήχου, είτε μουσική είτε ηχητικά εφέ, στην μηχανή γραφικών ΧΝΑ είναι πολύ απλή. Αυτή την δουλειά λοιπόν σε αυτή την εργασία την επιτελεί η κλάση **SndPlayer**. Συνήθως η λογική για να παίξουμε ένα αρχείο ήχου είναι η ίδια σε αρκετές περιπτώσεις: Δίνουμε την τοποθεσία του αρχείου ήχου, έπειτα το φορτώνουμε στην μνήμη και τέλος το αναπαράγουμε, είτε το σταματάμε, είτε το “τρέχουμε” μπρος και πίσω.

Η κλάση SndPlayer έχει κατασκευαστεί έτσι ώστε όταν δημιουργείται να φορτώνει το ηχητικό κομμάτι στην μνήμη. Έτσι λογικά πρέπει να περιέχει εντολές φορτώματος στον constructor της κλάσης όπως φαίνεται στον παρακάτω κώδικα:

```
12 |         public SoundPlayer soundPlayer = new SoundPlayer();
13 |
14 |     public SndPlayer()
15 |     {
16 |         soundPlayer.SoundLocation = Directory.GetCurrentDirectory() + "\\RedCarpet.wav";
17 |         soundPlayer.Load();
18 |     }
```

**Κώδικας 31.** Κώδικας φορτώματος αρχείου ήχου στη μνήμη.

Στην γραμμή 16 ορίζουμε την διαδρομή του αρχείου ήχου, ακολούθως στη γραμμή 17 φορτώνουμε το κομμάτι στην μνήμη. Για να αναπαράγουμε τον ήχο που μόλις φορτώσαμε δεν έχουμε παρά να καλέσουμε την επόμενη συνάρτηση:

```
20 |     public void play()
21 |     {
22 |         soundPlayer.PlayLooping();
23 |     }
24 |
25 |     public void stop()
26 |     {
27 |         soundPlayer.Stop();
28 |     }
```

**Κώδικας 32.** Αναπαραγωγή αρχείου ήχου.



Καλώντας την συνάρτηση στη γραμμή 22, ορίζουμε ότι θα ξεκινήσει η αναπαραγωγή και ότι μόλις το τραγούδι θα φτάσει στο τέλος θα ξεκινήσει και πάλι από την αρχή (PlayLooping). Στην γραμμή 27 ο κώδικας έχει την ικανότητα να σταματάει την αναπαραγωγή του τρέχοντος ήχου.

Κεφάλαιο 10  
Η κάμερα του παιχνιδιού

xna



Πριν αρχίσουμε να αναλύουμε τον κώδικα ο οποίος μας βοηθά να κατασκευάσουμε την κάμερα θα συζητήσουμε λίγο την αξία και τον ρόλο μίας κάμερας στα τρισδιάστατα παιχνίδια ή γενικότερα στις εφαρμογές τρισδιάστατου χώρου.

Όπως έχουμε αναφέρει και σε προηγούμενο κεφάλαιο, η κάμερα για μία τρισδιάστατη εφαρμογή είναι ζωτικής σημασίας. Η κάμερα αντιπροσωπεύει το “μάτι” του χρήστη την ορισμένη χρονική στιγμή δηλαδή με ποια οπτική γωνία κοιτάμε στο επίπεδο και από ποια θέση. Η μηχανή γραφικών XNA απεικονίζει τα αντικείμενα μας όπως αυτά φαίνονται από την κάμερα. Σε καμία περίπτωση δεν μπορούμε να ζωγραφίσουμε τρισδιάστατο αντικείμενο χωρίς αναφορά στην κάμερα. Όλα γίνονται με βάση αυτή. Οπότε πριν ακόμα ζωγραφίσουμε οτιδήποτε πρέπει να ορίσουμε την θέση της κάμερας και πώς αυτή κοιτάζει στον χώρο. Την κάμερα σε αυτήν την εργασία την υλοποιεί η κλάση **Camera**.

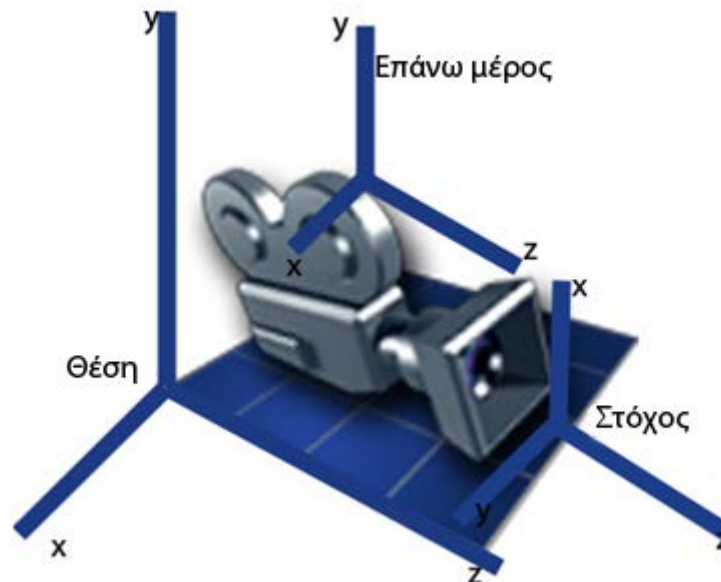
Ειδικότερα, για να κατασκευάσουμε μία κάμερα, πρέπει να ορίσουμε τρία πράγματα: Την θέση της, που κοιτάει και το επίπεδο προβολής. Για να το κάνουμε αυτό θα πρέπει να ορίσουμε δύο **πίνακες**. Τον πίνακα προβολής και τον πίνακα άποψης (**Projection** και **View Matrix**). Αυτά όλα χρειάζονται έτσι ώστε η μηχανή γραφικών XNA να μπορεί να μετατρέψει τα τρισδιάστατα αντικείμενα μας σε δυσδιάστατα ώστε εμείς να μπορούμε να τα βλέπουμε στην δυσδιάστατη οθόνη μας.

Αρχικά, για να ορίσουμε τον View Matrix θα πρέπει να καλέσουμε μία συνάρτηση του XNA. Καλώντας αυτή τη συνάρτηση θα έχουμε πλέον την θέση της κάμεράς μας και την κατεύθυνση στην οποία κοιτάει. Στην συνέχεια, θα πρέπει να ορίσουμε τον Projection Matrix ο οποίος είναι η προβολή της κάμερας και το πώς κοιτάει τον χώρο. Οι συναρτήσεις αυτές παρουσιάζονται παρακάτω:

```
73 |         view = Matrix.CreateLookAt(position, target, upVector);  
74 |         projection = Matrix.CreatePerspectiveFieldOfView(fov, aspectRatio, NearPlane, FarPlane);
```

**Κώδικας 33.** Ορίζοντας την κάμερα.

Στην γραμμή 73 κατασκευάζουμε τον πίνακα όψης με την βοήθεια της συνάρτησης **CreateLookAt**. Σε αυτήν την συνάρτηση όπως φαίνεται πρέπει να ορίσουμε την θέση (**position x,y,z**), την κατεύθυνση (**target x,y,z**) και το επάνω διάνυσμα, δηλαδή πού βρίσκεται η κορυφή της (**upVector x,y,z**). Τι θα έχουμε περίπου μόλις τελειώσουμε με τον πρώτο πίνακα παρουσιάζεται στην επόμενη εικόνα:

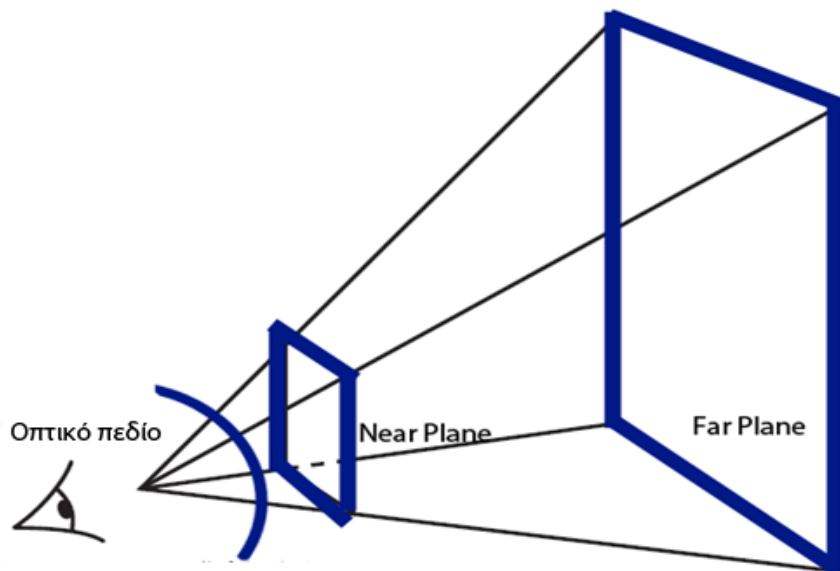


Εικόνα 53. Ο πίνακας όψης της κάμερας.

Έπειτα, στην επόμενη γραμμή, στην 74, θα πρέπει να ορίσουμε την προβολή της κάμερας. Με άλλα λόγια σε αυτό το σημείο κατασκευάζουμε τον φακό της, το “μάτι” της. Εδώ θα ορίσουμε πόσο πεδίο θα κοιτάει, πόσο μακριά και κοντά.

Για αυτό το σκοπό χρειαζόμαστε το **fov** (**field of view** σε μοίρες), το **aspect ratio**, το οποίο προκύπτει από το πλάτος και το ύψος της οθόνης και το **Near** και **Far** plain, όπου εκεί ορίζουμε με αριθμούς, πόσο κοντά και πόσο μακριά θα κοιτάει η κάμερα. Όσον αφορά την κοντινή απόσταση, εάν αυτή οριστεί σε μικρό νούμερο τότε θα έχουμε την δυνατότητα να πηγαίνουμε πολύ κοντά στα αντικείμενα και να μην χάνονται. Εάν αυτό το νούμερο οριστεί μεγάλο τότε τα αντικείμενα όσο τα πλησιάζουμε θα χάνονται γρηγορότερα. Πρέπει να προσέξουμε αυτό το νούμερο να μην είναι δεκαδικό, διότι σε μερικές περιπτώσεις θα έχουμε πρόβλημα με την προβολή των αντικειμένων, διότι θα χάνεται η ακρίβεια. Στην συνέχεια, και όσον αφορά την μακρινή απόσταση, εάν την ορίσουμε σε μικρό νούμερο θα πρέπει να πηγαίνουμε πολύ κοντά στα αντικείμενα μας έτσι ώστε αυτά να είναι ορατά. Σε αντίθετη περίπτωση τα αντικείμενα δεν θα χάνονται εύκολα όσο απομακρυνόμαστε παρά μόνο σε μεγάλη απόσταση.

Αυτά τα δύο νούμερα είναι σημαντικά διότι θα πρέπει να βλέπουμε τα αντικείμενα όσο κοντά ή όσο μακριά χρειάζεται, ρεαλιστικά. Για παράδειγμα, ένα αντικείμενο το οποίο ήταν ορατό σε τεράστια απόσταση θα ήταν ανεπιθύμητο διότι θα θέλαμε να εστιάσουμε την προσοχή του χρήστη αλλού παρά στο απομακρυσμένο αντικείμενο. Μία άποψη για το πίνακα προβολής της κάμερας παρουσιάζεται στην επόμενη εικόνα:



**Εικόνα 54.** Ο πίνακας προβολής της κάμερας.

Αφού τελειώσουμε λοιπόν με την αρχικοποίηση της κάμερας, πρέπει στην συνέχεια να την ανανεώνουμε ανάλογα με την κίνηση του ποντικιού. Η συγκεκριμένη κάμερα της πτυχιακής είναι μία κάμερα τύπου Quake, πρώτου προσώπου. Η μέθοδος με την οποία η κάμερα επιτυγχάνει την περιστροφή της με το ποντίκι και την κίνησή της με τα πλήκτρα του πληκτρολογίου περιγράφεται παρακάτω.

Η συνάρτηση η οποία επιτελεί την κίνηση στο παιχνίδι, είναι η **PollInput** της κλάσης της κάμερας. Αυτή η συνάρτηση θα πρέπει να πάρει ως όρισμα τον χρόνο που τρέχει το παιχνίδι, έτσι ώστε η κίνηση να είναι ομαλή. Στην συνέχεια γίνεται έλεγχος εάν είναι πατημένο κάποιο πλήκτρο στο πληκτρολόγιο, έτσι ώστε να προσθέσουμε την διαφορά από την προηγούμενη θέση στην κάμερα. Κάτι παρόμοιο γίνεται και με το ποντίκι. Εάν η θέση του ποντικιού είναι διαφορετική από αυτή που ήταν τελευταία, τότε γίνεται κίνηση.

Βέβαια σε κάθε ανανέωση της κάμερας θα πρέπει να ανανεώνουμε την θέση της, την οπτική της γωνία, το πάνω μέρος της και άλλα.

Ο κώδικας που υλοποιεί όλα τα προαναφερθέντα παρουσιάζεται παρακάτω:

```

112 private void PollInput(float amountOfMovement)
113 {
114     keys = Keyboard.GetState();
115     currentMouseState = Mouse.GetState();
116
117     if (state != State.Fixed)
118     {
119         Vector3 moveVector = new Vector3();
120
121         if (state == State.FirstPerson)
122         {
123             if (keys.IsKeyDown(Keys.D))
124             {
125                 moveVector.X -= amountOfMovement;
126             }
127             if (keys.IsKeyDown(Keys.A))
128             {
129                 moveVector.X += amountOfMovement;
130             }
131             if (keys.IsKeyDown(Keys.S))
132             {
133                 moveVector.Z -= amountOfMovement;
134             }
135             if (keys.IsKeyDown(Keys.W))
136             {
137                 moveVector.Z += amountOfMovement;
138             }
139             if (keys.IsKeyDown(Keys.Z))
140             {
141                 moveVector.Y -= amountOfMovement;
142             }
143             if (keys.IsKeyDown(Keys.X))
144             {
145                 moveVector.Y += amountOfMovement;
146             }
147         }

```

**Κώδικας 34.** Η κίνηση της κάμερας με την βοήθεια του πληκτρολογίου.

Αρχικά, στην γραμμή 114 λαμβάνουμε την κατάσταση των πλήκτρων του πληκτρολογίου, έτσι ώστε αργότερα να διαπιστώσουμε ποια κουμπιά έχουν πατηθεί αρχικά και ποιες ενέργειες θα κάνουμε αργότερα. Στη γραμμή 119, στην μεταβλητή **moveVector** θα προσθέσουμε όλες τις κινήσεις οι οποίες έγιναν αντιληπτές από το πληκτρολόγιο. Αργότερα αυτόν τον vector θα τον προσθέσουμε στην τρέχουσα θέση της κάμερας έτσι ώστε αυτή να μετακινηθεί.

Παρακάτω παρουσιάζεται πώς επιτυγχάνεται η περιστροφή της κάμερας:

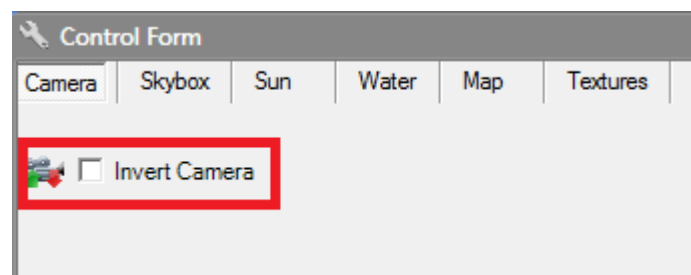
```

153     if (!Renderer.controlForm.CameraInvert.Checked)
154     {
155         if (currentMouseState.X != previousMouseState.X)
156             rotation.Y -= amountOfMovement / 800.0f * (currentMouseState.X - previousMouseState.X);
157         if (currentMouseState.Y != previousMouseState.Y)
158             rotation.X += amountOfMovement / 800.0f * (currentMouseState.Y - previousMouseState.Y);
159     }
160     else
161     {
162         if (currentMouseState.X != previousMouseState.X)
163             rotation.Y -= amountOfMovement / 800.0f * (currentMouseState.X - previousMouseState.X);
164         if (currentMouseState.Y != previousMouseState.Y)
165             rotation.X -= amountOfMovement / 800.0f * (currentMouseState.Y - previousMouseState.Y);
166     }
167
168     Mouse.SetPosition(screenCenter.X, screenCenter.Y);

```

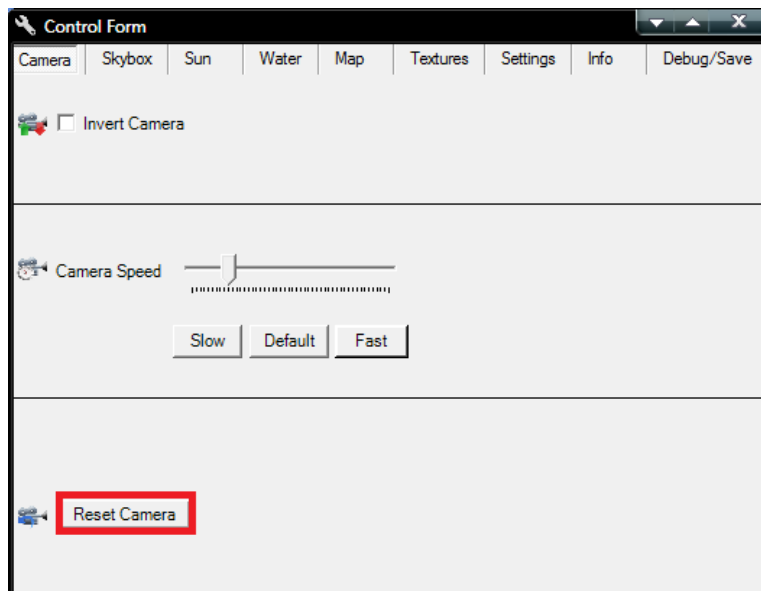
### Κώδικας 35. Υλοποίηση περιστροφής της κάμερας με το ποντίκι.

Αρχικά στην γραμμή 153 ελέγχουμε εάν η επιλογή “inverted camera” είναι επιλεγμένη, έτσι ώστε η κάμερα να κινείται όπως κινούνται οι κάμερες στην προσομοίωση αεροπλάνων, δηλαδή ανάποδα. Έπειτα, με την βοήθεια ενός vector στον οποίο προσθέτουμε κάθε φορά τις διαφορές της θέσης του ποντικιού, έχουμε την “ποσότητα” της περιστροφής. Αυτόν τον vector θα τον χρησιμοποιήσουμε αργότερα για να “φτιάξουμε περιστροφή” μία συνάρτηση η οποία υλοποιείται στο XNA και παρέχεται έτοιμη και την οποία θα την παρουσιάσουμε αργότερα. Στην επόμενη εικόνα φαίνεται το κουμπί “αναποδογυρίσματος” της κάμερας:



Εικόνα 55. Κουμπί αναποδογυρίσματος της κάμερας.

Στην συνέχεια παρουσιάζεται μία πολύ σημαντική συνάρτηση η οποία επιτελεί επαναφορά της κάμερας. Η συνάρτηση **ResetCamera** επαναφέρει την κάμερα στην αρχική της κατάσταση με ομαλή κίνηση. Αυτή η λειτουργία ενεργοποιείται με το πάτημα του κουμπιού “Reset Camera” πάνω στην φόρμα διαχείρισης όπως φαίνεται στην παρακάτω εικόνα:



**Εικόνα 56.** Επαναφορά της κάμερας.

Η λογική της επαναφοράς είναι πολύ απλή: Καταρχήν η συνάρτηση `ResetCamera` καλείται συνεχώς καθ' όλη την διάρκεια της εφαρμογής. Κάτι τέτοιο όμως λογικά θα έπρεπε να κάνει την κάμερα να επαναφέρεται συνέχεια χωρίς να υπάρχει κίνηση. Με την βοήθεια όμως μίας **bool** μεταβλητής, ενεργοποιείται το συμβάν. Δηλαδή η μεταβλητή **isReseted** παραμένει **true** στην διάρκεια της εφαρμογής και έτσι δεν υπάρχει επαναφορά. Εάν, όμως, πατηθεί το κουμπί της φόρμας για επαναφορά αυτή γίνεται ψευδής και έτσι πλέον πρέπει να επαναφερθεί η κάμερα στην αρχική της θέση πριν γίνει και πάλι αληθής. Ο κώδικας ενεργοποίησης φαίνεται παρακάτω:

```

184  public void resetCamera()
185  {
186      if (isReseted)
187      {
188          //Do Nothing
189          return;
190      }
191      else
192      {

```

**Κώδικας 36.** Κώδικας ενεργοποίησης επαναφοράς.

Όταν πλέον ενεργοποιηθεί αυτή η λειτουργία, θα πρέπει πριν ορίσουμε την μεταβλητή `isReseted` ως αληθή και πάλι να είμαστε σίγουροι ότι η κάμερα θα βρίσκεται στο αρχικό σημείο εκεί από όπου ξεκίνησε. Για αυτό τον λόγο στην δημιουργία της κλάσης της κάμερας αποθηκεύουμε την κατάσταση της και έπειτα όταν γίνεται η επαναφορά με μία αφαίρεση ελέγχουμε την διαφορά της από την αρχική της θέση και σε κάθε ανανέωση προσθέτουμε το 10% της διαφοράς της. Έτσι έχουμε ένα εφέ επαναφοράς γρήγορο



στην αρχή και πιο αργό όσο πλησιάζουμε στο σημείο. Επίσης μέσα σε αυτή την συνάρτηση ελέγχουμε εάν η διαφορά είναι μικρότερη ή ίση από το 0.1 έτσι ώστε να σταματήσουμε διότι είναι δύσκολο και μάλλον τυχαίο να πετύχουμε ολική επαναφορά προσθέτοντας κάθε φορά το 10%.

Αφού βέβαια έρθουμε τόσο κοντά ορίζουμε την κάμερα στην αρχική της θέση. Ο κώδικας παρακάτω θα μας βοηθήσει να κατανοήσουμε την διαδικασία επαναφοράς:

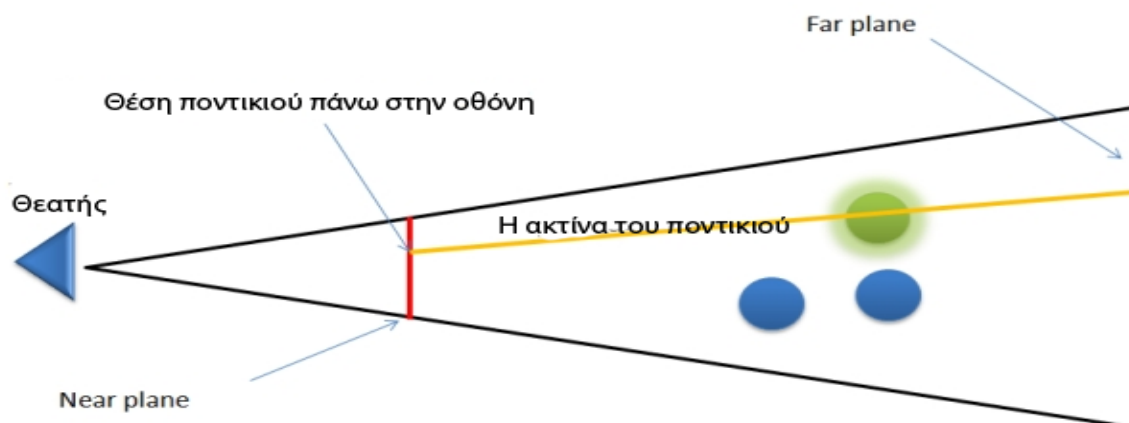
```
193         float cameraPositionXDifference = cameraInitialPosition.X - position.X;
194         float cameraPositionYDifference = cameraInitialPosition.Y - position.Y;
195         float cameraPositionZDifference = cameraInitialPosition.Z - position.Z;
196
197         float cameraRotationXDifference = cameraInitialRotation.X - rotation.X;
198         float cameraRotationYDifference = cameraInitialRotation.Y - rotation.Y;
199         float cameraRotationZDifference = cameraInitialRotation.Z - rotation.Z;
200
201         //Check Settings
202         if ((Math.Abs(cameraPositionXDifference) <= 0.1) && (Math.Abs(cameraPositionYDifference) <= 0.1) && (Math.Abs(cameraPositionZDifference) <= 0.1))
203         {
204             if ((Math.Abs(cameraRotationXDifference) <= 0.1) && (Math.Abs(cameraRotationYDifference) <= 0.1) && (Math.Abs(cameraRotationZDifference) <= 0.1))
205             {
206                 isReseted = true;
207                 Renderer.controlForm.addToConsole("Camera reseted.");
208             }
209         }
210         else
211         {
212             position.X += cameraPositionXDifference * 0.1f;
213             position.Y += cameraPositionYDifference * 0.1f;
214             position.Z += cameraPositionZDifference * 0.1f;
215
216             rotation.X += cameraRotationXDifference * 0.1f;
217             rotation.Y += cameraRotationYDifference * 0.1f;
218             rotation.Z += cameraRotationZDifference * 0.1f;
219         }
```

**Κώδικας 37.** Η επαναφορά της κάμερας.

Βλέπουμε πως στις γραμμές 193 έως 199 χρησιμοποιούμε κάποιες μεταβλητές έτσι ώστε να έχουμε την διαφορά της κάμερας. Στις γραμμές 202 έως 207 ελέγχουμε εάν αυτές οι διαφορές είναι μικρότερες από το 0.1 όπως είπαμε πριν. Από τις γραμμές 212 έως 218 προσθέτουμε κάθε φορά το 10% της διαφοράς. Έτσι πολύ απλά έχουμε την επιστροφή της κάμερας στο αρχικό σημείο με την βοήθεια ενός εφέ.

Μία πολύ χρήσιμη συνάρτηση επίσης μέσα στην κλάση της κάμερας είναι η συνάρτηση GetPickRay(), η οποία μας βοηθά να έχουμε μία νοητή ακτίνα για σημείο που βρίσκεται το ποντίκι μας. Με αυτή την τακτική της νοητής ακτίνας είναι δυνατόν αργότερα να καταλάβουμε που ακριβώς στον τρισδιάστατο χώρο βρίσκεται το ποντίκι μας από την

δυσδιάστατη οθόνη μας. Η νοητή ακτίνα που παίρνουμε από αυτήν την συνάρτηση παρουσιάζεται παρακάτω:



Εικόνα 57. Η νοητή ακτίνα του ποντικιού.

Έτσι όπως φαίνεται και στην προηγούμενη εικόνα μπορούμε τελικά να κατανοήσουμε με ποιο σημείο συγκρούεται το ποντίκι μας, έτσι ώστε όπως θα δούμε αργότερα να μπορούμε να επεξεργαστούμε το τοπίο. Ο κώδικας παρουσιάζεται παρακάτω:

```
82 public Ray GetPickRay()
83 {
84     MouseState mouseState = Mouse.GetState();
85
86     int mouseX = mouseState.X;
87     int mouseY = mouseState.Y;
88
89     float width = Renderer.graphics.GraphicsDevice.Viewport.Width;
90     float height = Renderer.graphics.GraphicsDevice.Viewport.Height;
91
92     double screenSpaceX = ((float)mouseX / (width / 2) - 1.0f) * aspectRatio;
93     double screenSpaceY = (1.0f - (float)mouseY / (height / 2));
94
95     double viewRatio = Math.Tan(fov / 2);
96
97     screenSpaceX = screenSpaceX * viewRatio;
98     screenSpaceY = screenSpaceY * viewRatio;
```

Κώδικας 38. Πρώτη φάση εντοπισμού ακτίνας ποντικιού.

Βλέπουμε πως αρχικά στις γραμμές 84 έως 87, δεσμεύουμε την θέση στην οποία βρίσκεται το ποντίκι. Έπειτα, στις γραμμές 89 και 90 πρέπει να ξέρουμε το μέγεθος εξόδου της κάρτας γραφικών. Στις επόμενες γραμμές θα ορίσουμε μέχρι πού θα φτάνει η ακτίνα του ποντικιού μας, ώστε τα αντικείμενα να εντοπίζονται μέχρι αυτό το σημείο.

```

100 Vector3 cameraSpaceNear = new Vector3((float)(screenSpaceX * NearPlane), (float)(screenSpaceY * NearPlane), (float)(-NearPlane))
101 Vector3 cameraSpaceFar = new Vector3((float)(screenSpaceX * FarPlane), (float)(screenSpaceY * FarPlane), (float)(-FarPlane));
102
103 Matrix invView = Matrix.Invert(view);
104 Vector3 worldSpaceNear = Vector3.Transform(cameraSpaceNear, invView);
105 Vector3 worldSpaceFar = Vector3.Transform(cameraSpaceFar, invView);
106
107 Ray pickRay = new Ray(worldSpaceNear, worldSpaceFar - worldSpaceNear);
108
109 return new Ray(pickRay.Position, Vector3.Normalize(pickRay.Direction));

```

**Κώδικας 39.** Ολοκλήρωση φάσης εντοπισμού ακτίνας ποντικιού.

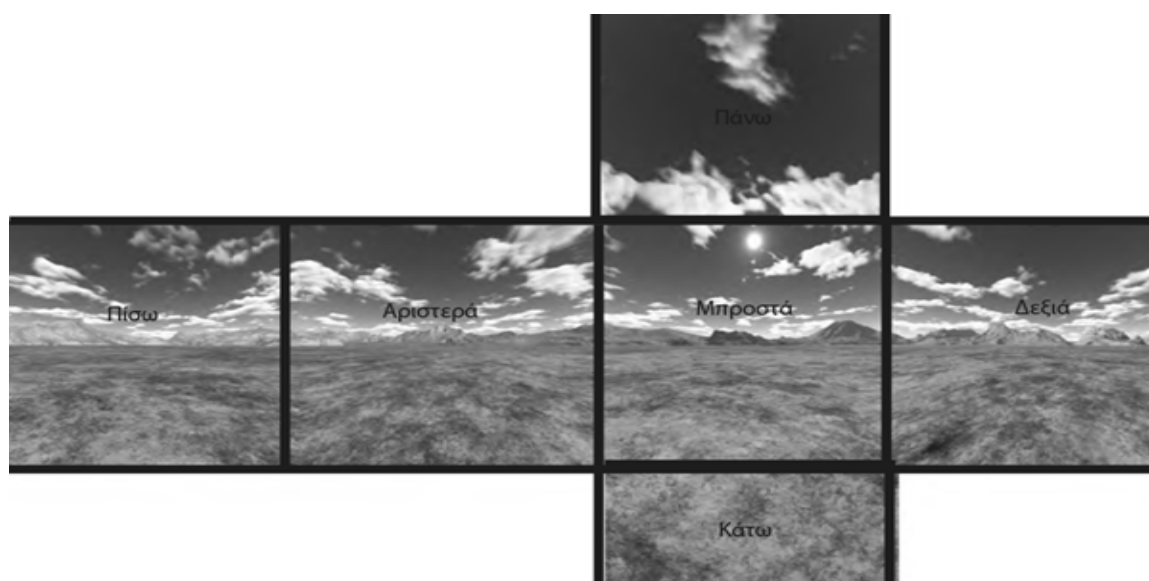
Στην δεύτερη φάση λοιπόν βλέπουμε τον κώδικα με τον οποίο μπορούμε τελικά να έχουμε μία νοητή ακτίνα του ποντικιού. Στο τέλος του κώδικα απλώς κάνουμε κανονικοποίηση του διανύσματος μας με τη συνάρτηση **Normalize** διότι μας ενδιαφέρει η διεύθυνση και σε άλλη περίπτωση μπορεί να βγαίναμε εκτός ορίων. Παράδειγμα ένα διάνυσμα (30,-5,0) θα γινόταν (1,-1,0). Όλα βλέπουμε πως κανονικοποιούνται με βάση τον άσσο διότι μας ενδιαφέρει η τελική κατεύθυνση και όχι τα νούμερα.



χρη



Ο ουρανός είναι μία πολύ σημαντική βελτίωση για την εφαρμογή. Τον ουρανό τον υλοποιεί η κλάση **Skybox**. Πλέον μπορούμε να έχουμε όχι μόνο ένα μονόχρωμο φόντο, αλλά έναν ωραίο ατελείωτο ουρανό. Βέβαια για του λόγου το αληθές δεν είναι και τόσο ατελείωτος. Υπάρχει μία τεχνική που ακολουθείται. Η τεχνική έχει ως εξής: Θα τοποθετήσουμε έξι εικόνες γύρω στον χώρο σε μία μακρινή απόσταση και σύμφωνα με την κίνηση που θα έχει η κάμερά μας όπου είδαμε πριν, αυτές θα μετακινούνται. Έτσι θα δίνεται η εντύπωση στον χρήστη ότι ο ουρανός είναι απέραντος. Με αυτήν την απλή τεχνική και με απλό κώδικα θα υλοποιηθεί ο ουρανός. Η κάμερα βέβαια θα παραμένει πάντα στο κέντρο αυτών των έξι εικόνων. Πρέπει πριν όμως ακόμα φορτώσουμε τις εικόνες να ορίσουμε έναν “σκελετό”. Αυτός ο σκελετός δεν είναι τίποτα άλλο παρά ένα μοντέλο όπου πάνω του θα τοποθετήσουμε τις έξι εικόνες του ουρανού. Υπάρχουν έτοιμα προγράμματα έτσι ώστε αυτές οι εικόνες που θα τοποθετήσουμε να φαίνονται συνεχείς. Εμείς απλώς τοποθετούμε μία εικόνα με ουρανό και αυτά τα προγράμματα μας επιστρέφουν αυτές τις έξι εικόνες με τα αντίστοιχα ονόματα: Πάνω, κάτω, εμπρός, πίσω, δεξιά και αριστερά. Παρακάτω φαίνονται οι έξι εικόνες που αποτελούν έναν τυχαίο ουρανό:



**Εικόνα 58.** Τα κομμάτια ενός ουρανού.

Παρακάτω παρουσιάζεται ο κώδικας με τον οποίο μπορούμε να κατασκευάσουμε τον ουρανό στην μηχανή γραφικών Microsoft XNA. Η λογική στις υπόλοιπες μηχανές παραμένει η ίδια:

```

6 public class SkyBox
7 {
8     Texture2D skyBoxUp;
9     Texture2D skyBoxDown;
10    Texture2D skyBoxLeft;
11    Texture2D skyBoxRight;
12    Texture2D skyBoxFront;
13    Texture2D skyBoxBack;
14    Model skyBoxModel;
15    Matrix world;
16
17    float scale = 100f;
18
19    public SkyBox()
20    {
21        skyBoxModel = Renderer.content.Load<Model>(@"content\skybox\skybox");
22        LoadTextures("clearblue");
23    }

```

**Κώδικας 40.** Δήλωση αρχικών μεταβλητών.

Από τις γραμμές 8 έως 13, δηλώνουμε αντικείμενα τύπου **Texture2D** τα οποία τα είχαμε συναντήσει και σε προηγούμενο κεφάλαιο. Αυτά τα αντικείμενα θα κρατούν τα 6 μέρη του ουρανού. Στην γραμμή 14 η δήλωση ενός αντικειμένου **Model** θα μας βοηθήσει να κρατήσουμε τον “σκελετό” του ουρανού. Σε προηγούμενο κεφάλαιο είχαμε αναφέρει ότι μπορούμε να εξάγουμε τέτοια μοντέλα με την βοήθεια παράδειγμα του 3DS Studio Max. Αυτό το αρχείο αν προσπαθήσουμε να το ανοίξουμε με έναν κειμενογράφο θα διαπιστώσουμε ότι αποτελείται από χιλιάδες συντεταγμένες στο χώρο. Αυτό ακριβώς είναι και η γέφυρα επικοινωνίας μοντέλου με μηχανής γραφικών. Στην συνέχεια, στην γραμμή 15 δηλώνουμε μία μεταβλητή χώρου, με την βοήθεια της οποίας μπορεί η μηχανή γραφικών να καταλάβει σε τι χώρο βρίσκεται ο ουρανός μας. Για παράδειγμα εάν ο χώρος είναι τεράστιος τότε το αντικείμενο θα φανεί μικρό. Στις γραμμές 21 και 22 με την βοήθεια του επεξεργαστή περιεχομένου φορτώνουμε τις εικόνες στην μνήμη, δηλαδή στα αντικείμενα Texture2D.

Στην συνέχεια θα παρουσιαστεί και η υλοποίηση και της συνάρτησης LoadTextures:

```
25 public void LoadTextures(string skyName)
26 {
27     skyBoxBack = Renderer.content.Load<Texture2D>(@"content\Skybox\Textures\" + skyName + "_back");
28     skyBoxFront = Renderer.content.Load<Texture2D>(@"content\Skybox\Textures\" + skyName + "_front");
29     skyBoxDown = Renderer.content.Load<Texture2D>(@"content\Skybox\Textures\" + skyName + "_down");
30     skyBoxUp = Renderer.content.Load<Texture2D>(@"content\Skybox\Textures\" + skyName + "_up");
31     skyBoxRight = Renderer.content.Load<Texture2D>(@"content\Skybox\Textures\" + skyName + "_left");
32     skyBoxLeft = Renderer.content.Load<Texture2D>(@"content\Skybox\Textures\" + skyName + "_right");
33 }
34
35 public void Update(Vector3 cameraPosition)
36 {
37     world = Matrix.CreateScale(scale) * Matrix.CreateRotationX(MathHelper.ToRadians(-90f)) *
38         Matrix.CreateTranslation(cameraPosition);
39 }
40
```

**Κώδικας 41.** Φόρτωση εικόνων και δημιουργία χώρου.

Στις γραμμές 27 έως 32 γίνεται φόρτωση των αντικειμένων στην μνήμη. Στην γραμμή 37 παρουσιάζεται ο επίσημος τρόπος δημιουργίας ενός χώρου: Πολλαπλασιάζουμε την αναλογία (scale), με την περιστροφή (-90 μοίρες) και τέλος με την θέση της κάμερας, έτσι ώστε να μετακινείται ο κύβος σύμφωνα με την κάμερα όπως αναφέραμε πριν.

Τέλος θα παρουσιαστεί η μέθοδος με την βοήθεια της οποίας θα καταφέρουμε απλώς να ζωγραφίσουμε τον ουρανό:

```
46     foreach (ModelMesh mesh in skyBoxModel.Meshes)
47     {
48         foreach (BasicEffect meshEffect in mesh.Effects)
49         {
50             meshEffect.World = world;
51             meshEffect.View = view;
52             meshEffect.Projection = projection;
53
54             meshEffect.TextureEnabled = true;
55             meshEffect.LightingEnabled = false;
56
57             switch (mesh.Name)
58             {
59                 case "up":
60                     meshEffect.Texture = skyBoxUp;
61                     break;
62                 case "left":
63                     meshEffect.Texture = skyBoxLeft;
64                     break;
65                 case "right":
66                     meshEffect.Texture = skyBoxRight;
67                     break;
68             }
69         }
70     }
71 }
```



```

68         case "back":
69             meshEffect.Texture = skyBoxBack;
70             break;
71         case "front":
72             meshEffect.Texture = skyBoxFront;
73             break;
74         case "down":
75             meshEffect.Texture = skyBoxDown;
76             break;
77     }
78
79     meshEffect.Alpha = Renderer.controlForm.Skybox;
80     meshEffect.DiffuseColor = new Vector3(Renderer.
81     meshEffect.EmissiveColor = new Vector3(Renderer
82 }
83 mesh.Draw();

```

#### Κώδικας 42. Ζωγραφίζοντας τον ουρανό.

Για αυτό το σκοπό χρησιμοποιούμε μία πάρα πολύ εύχρηστη συνάρτηση που υπάρχει στην γλώσσα C#, την **foreach**. Η συνάρτηση αυτή θα πλοηγηθεί σε όλα τα αντικείμενα μίας συλλογής χωρίς να γνωρίζουμε εκ των προτέρων τον αριθμό τους. Στην γραμμή 48 χρησιμοποιούμε ένα **βασικό εφέ**. Αργότερα θα δούμε πώς μπορούμε να δημιουργήσουμε και τα δικά μας. Πάντως σε αυτή την περίπτωση μας βολεύει διότι έχουμε να εμφανίσουμε μόνο 6 εικόνες. Στις επόμενες γραμμές ρυθμίζουμε τον φωτισμό και κάποιες άλλες παραμέτρους και τέλος στις γραμμές 59 έως 76 χρησιμοποιούμε την συνάρτηση case για να ορίσουμε τις εικόνες οι οποίες θα εμφανίζονται. Στην γραμμή 83 απλώς ζωγραφίζουμε το μοντέλο μας. Το αποτέλεσμα βέβαια θα είναι θεαματικό, όπως παρουσιάζεται στην παρακάτω εικόνα:



Εικόνα 59. Ο ουρανός της πτυχιακής.



χηα

Ο ήλιος είναι ένα άλλο σημαντικό κομμάτι αυτής της εργασίας. Τον ήλιο υλοποιεί η κλάση **Sun**. Θα δούμε σε αυτό το κεφάλαιο πώς μπορούμε να κατασκευάσουμε τον δικό μας ήλιο. Στην συνέχεια θα δούμε πώς γνωρίζοντας την θέση του ηλίου μπορούμε να φωτίσουμε το τοπίο μας.

Αρχικά να πούμε ότι θα θέλαμε ιδανικά τον ήλιο μας να γυρνάει γύρω από ένα σημείο αναφοράς. Στο ίδιο σημείο αναφοράς θα τοποθετήσουμε και το τοπίο μας αργότερα. Πώς μπορούμε λοιπόν να υλοποιήσουμε κάτι τέτοιο;

Αρχικά να πούμε πως και πάλι θα χρησιμοποιήσουμε τον επεξεργαστή περιεχομένου για να φορτώσουμε τα 4 στρώματα του ηλίου. Τα 4 στρώματα του ηλίου αποτελούνται από 4 textures εκ των οποίων το πρώτο είναι ο ήλιος και τα υπόλοιπα τρία είναι οι κύκλοι που εμφανίζονται όταν κοιτάμε με την κάμερα απευθείας τον ήλιο. Κάτι τέτοιο εμφανίζεται παρακάτω:



**Εικόνα 60.** Τα 4 στρώματα του ήλιου και οι κύκλοι.

Αρχικά ορίζουμε τα τρίγωνα και τις κορυφές με την βοήθεια των οποίων θα μπορέσουμε να ζωγραφίσουμε τον ήλιο αργότερα.

Πριν προχωρήσουμε θα πρέπει να αναφέρουμε το τι γίνεται με τις κορυφές και τα τρίγωνα και γιατί μία τέτοια δήλωση είναι σημαντική. Με την βοήθεια λοιπόν ενός ενδιάμεσου **buffer** ο οποίος περιέχει τρίγωνα και κορυφές, η κάρτα γραφικών μπορεί να εξοικονομήσει πόρους και μνήμη, ώστε να χρησιμοποιηθούν για άλλες δουλειές. Αυτός ο buffer χωρίζεται σε δύο κομμάτια: Στον Vertex και στον Index buffer. Στον vertex

buffer θα δηλώσουμε όλα τα τρίγωνά μας όπως είναι και στον index buffer τις κορυφές και πως αυτές ενώνονται. Όλη την υπόλοιπη υλοποίηση θα την αναλάβει η μηχανή γραφικών. Παράδειγμα δύο τριγώνων και πως ενώνονται οι κορυφές έτσι ώστε η μηχανή γραφικών να κατανοήσει όταν τα τρίγωνα γειτονεύουν, είναι το εξής:

Index[0] = 0 ;

Index[1] = 1 ;

Index[2] = 2 ;

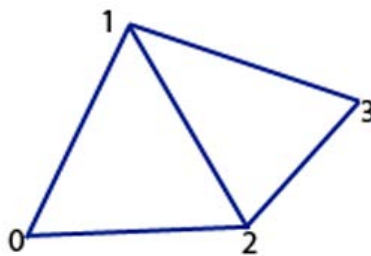
Index[3] = 1 ;

Index[4] = 2 ;

Index[5] = 3 ;

**Κώδικας 43.** Παράδειγμα δήλωσης index buffer.

Η μηχανή γραφικών, λοιπόν, βλέποντας κάτι τέτοιο και γνωρίζοντας ότι έχει δύο τρίγωνα θα μας δώσει κάτι σαν το ακόλουθο σχήμα:



**Εικόνα 61.** Τα τρίγωνα της προηγούμενης δήλωσης.

Προχωρώντας λοιπόν παρουσιάζεται ο κώδικας ο οποίος κατασκευάζει αυτά τα buffers:

```

39 private void SetUpVertices(Vector2 size)
40 {
41     size *= 0.5f;
42     Vector3 p0 = new Vector3(-size.X, -size.Y, 0f);
43     Vector3 p1 = new Vector3(-size.X, size.Y, 0f);
44     Vector3 p2 = new Vector3(size.X, -size.Y, 0f);
45     Vector3 p3 = new Vector3(size.X, size.Y, 0f);
46
47     vertex = new VertexPositionTexture[4];
48     vertex[0] = new VertexPositionTexture(p0, new Vector2(0f, 0f));
49     vertex[1] = new VertexPositionTexture(p1, new Vector2(0f, 1f));
50     vertex[2] = new VertexPositionTexture(p2, new Vector2(1f, 0f));
51     vertex[3] = new VertexPositionTexture(p3, new Vector2(1f, 1f));
52
53     vertexBuffer = new VertexBuffer(Renderer.graphics.GraphicsDevice, typeof(VertexPositionTexture), vertex.Length, BufferUsage.None);
54     vertexBuffer.SetData<VertexPositionTexture>(vertex);
55 }
56
57 private void SetUpIndices()
58 {
59     short[] Indices = new short[6];
60
61     Indices[0] = 0;
62     Indices[1] = 2;
63     Indices[2] = 1;
64
65     Indices[3] = 1;
66     Indices[4] = 2;
67     Indices[5] = 3;
68
69     indexBuffer = new IndexBuffer(Renderer.graphics.GraphicsDevice, typeof(short), Indices.Length, BufferUsage.None);
70     indexBuffer.SetData<short>(Indices);

```

**Κώδικας 44.** Ο κώδικας για δηλώσεις τριγώνων μέσα στον buffer.

Στην γραμμή 53 και 54 ορίζουμε τον buffer των τριγώνων ενώ πριν έχουμε δηλώσει όλα μας τα τρίγωνα από τα οποία θα αποτελείται ο ήλιος. Στην 69 και 70 γραμμή ορίζουμε τον index buffer. Να πούμε ότι αυτές οι δηλώσεις ισχύουν για όλα τα κομμάτια τα οποία απαρτίζουν τον ήλιο. Στην συνέχεια για να ζωγραφίζουμε αυτά τα κομμάτια θα καλέσουμε την συνάρτηση Draw:

```

108     foreach (EffectPass pass in effect.CurrentTechnique.Passes)
109     {
110         pass.Apply();
111
112         Renderer.graphics.GraphicsDevice.Indices = indexBuffer;
113         Renderer.graphics.GraphicsDevice.SetVertexBuffer(vertexBuffer);
114         Renderer.graphics.GraphicsDevice.DrawIndexedPrimitives(PrimitiveType.TriangleList, 0, 0,
115     }
116 }
117

```

**Κώδικας 45.** Κώδικας εμφάνισης ενός Layer του ήλιου.

Όλα τα προηγούμενα και ό,τι αφορά τον ήλιο βρίσκονται στην κλάση **SunQuad**. Αυτήν την κλάση, όπως έχουμε αναφέρει και σε προηγούμενο κεφάλαιο, την χρησιμοποιεί η κύρια κλάση **Sun**, τέσσερις φορές. Το μόνο που μένει λοιπόν είναι να

χρησιμοποιήσουμε την κλάση SunQuad με τέσσερις διαφορετικούς τρόπους στην κλάση του ήλιου. Ο παρακάτω κώδικας κάνει ακριβώς αυτή την δουλειά:

```
50 private void InitLayers()
51 {
52     layer = new SunQuad[4];
53     layerPos = new Vector3[4];
54
55     layer[0] = new SunQuad(new Vector2(10f, 10f), "lens04", 120f);
56     layer[1] = new SunQuad(new Vector2(12f, 12f), "lens03", 12f);
57     layer[2] = new SunQuad(new Vector2(10f, 10f), "lens02", 30f);
58     layer[3] = new SunQuad(new Vector2(15f, 15f), "lens01", 15f);
59 }
60
```

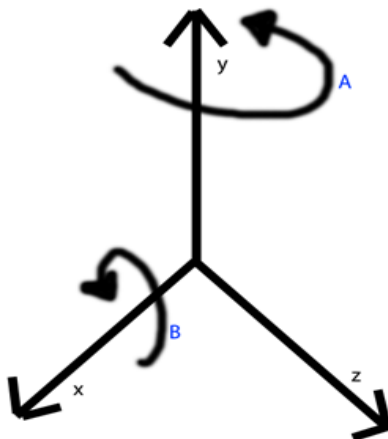
**Κώδικας 46.** Φόρτωση των τεσσάρων επιπέδων του ήλιου στην μηχανή γραφικών.

Έτσι πλέον αφού έχουμε φορτώσει τα κομμάτια του ήλιου στη μνήμη δεν μένει παρά να τα εμφανίσουμε. Επειδή είπαμε πως στην αρχή θα θέλαμε ιδανικά ο ήλιος μας να βρίσκεται σε μία λογική απόσταση από τον παρατηρητή και να περιστρέφεται γύρω από ένα σημείο αναφοράς, κάτι τέτοιο το πετυχαίνουμε με τις παρακάτω συναρτήσεις:

```
61 public void Update()
62 {
63     //Angle
64     rotation.Y += (float)LongitudeSpeed;
65
66     //Elevation
67     rotation.X -= (float)LatitudeSpeed;
68
69     position = Renderer.camera.position;
70
71     //Same code as the camera
72     rotationMatrix = Matrix.CreateRotationX(rotation.X) * Matrix.CreateRotationY(rotation.Y);
73     targetPos = position + Vector3.Transform(new Vector3(0, 0, 1), rotationMatrix);
74     Vector3 upVector = Vector3.Transform(new Vector3(0, 1, 0), rotationMatrix);
75     direction = Vector3.Normalize(targetPos - position);
76
77     lightPower = Renderer.controlForm.SunLighting.Value * (0.1f);
78
79     if (Renderer.controlForm.VerticalMoveEnable.Checked)
80         LatitudeSpeed = Renderer.controlForm.VerticalMovSpeed.Value * 0.001f;
81     else
82         LatitudeSpeed = 0;
83
84     if (Renderer.controlForm.HorizontalMoveEnable.Checked)
85         LongitudeSpeed = Renderer.controlForm.HorMoveSpeed.Value * 0.001f;
86     else
87         LongitudeSpeed = 0;
88
89     layer[0].alpha = Renderer.controlForm.sunGlowSetting.Value * 0.1f;
90     layer[1].alpha = Renderer.controlForm.sunGlowSetting.Value * 0.1f;
91     layer[2].alpha = Renderer.controlForm.sunGlowSetting.Value * 0.1f;
92     layer[3].alpha = Renderer.controlForm.sunGlowSetting.Value * 0.1f;
93     distance = Renderer.controlForm.sunZoomSetting.Value;
94
95     UpdateLayers();
96 }
```

**Κώδικας 47.** Κώδικας περιστροφής του ήλιου.

Στις γραμμές 64 και 67 δεσμεύουμε τα νούμερα τα οποία προσδίδουν περιστροφή στον ήλιο, στον άξονα y και στον άξονα x αντίστοιχα. Η περιστροφή παρουσιάζεται παρακάτω:



**Εικόνα 62.** Περιστροφή στον άξονα y (A σημείο) και x (B σημείο).

Έτσι εάν φανταστούμε πως ήμασταν παρατηρητές μέσα στο τοπίο ο ήλιος θα περιστρεφόταν από τα δεξιά μας προς τα αριστερά μας, και από μπροστά μας προς την πλάτη μας. Ας αναλύσουμε λοιπόν λίγο τον κώδικα. Για να επιτύχουμε περιστροφή στην μηχανή γραφικών XNA θα χρησιμοποιήσουμε μία έτοιμα συνάρτηση δημιουργίας περιστροφής που μας παρέχεται. Η συνάρτηση αυτή ονομάζεται **CreateRotation** και είναι ικανή να μας δώσει περιστροφή σε όλους τους άξονες (x,y,z) και συναντιέται με το ίδιο όνομα σε πολλές μηχανές γραφικών. Έτσι, στην γραμμή 72 του κώδικα ορίζουμε τον πίνακα (Matrix) περιστροφής. Στην γραμμή 73 ορίζουμε τον στόχο, δηλαδή προς τα που πηγαίνει το αντικείμενο μας και αυτό πετυχαίνεται με την προσθήκη της θέσης στον κανονικοποιημένο πίνακα περιστροφής. Στην γραμμή 95 απλώς καλούμε μία συνάρτηση Update() όπου ανανεώνουμε τις παραμέτρους του ηλίου σύμφωνα με τις πράξεις που προηγήθηκαν.

Τώρα θα πάμε να εξετάσουμε τον κώδικα HLSL που χρησιμοποιήθηκε για το κομμάτι του ηλίου:

```

sampler TextureSampler = sampler_state
{
    Texture = (Material);
    ADDRESSU = CLAMP;
    ADDRESSV = CLAMP;
    MIPFILTER = LINEAR;
    MAGFILTER = LINEAR;
    MINFILTER = LINEAR;
    MIPFILTER = LINEAR;
};

struct VS_INPUT
{
    float4 Position      : POSITION0;
    float2 Texcoord      : TEXCOORD0;
};

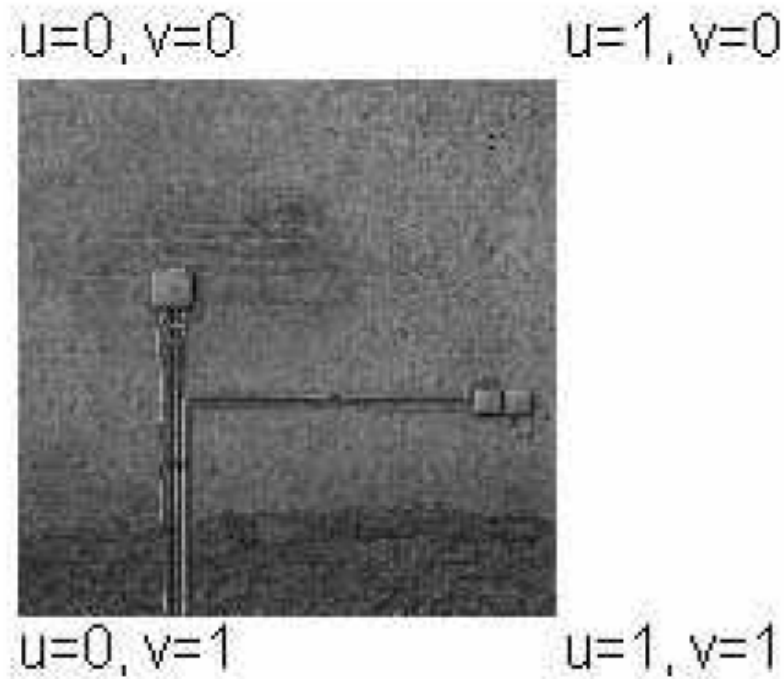
struct VS_OUTPUT
{
    float4 Position      : POSITION0;
    float2 Texcoord      : TEXCOORD0;
};

```

**Κώδικας 48.** Ο κώδικας για το εφέ του ήλιου.

Ο κώδικας σε αυτή την περίπτωση είναι πολύ απλός. Στην αρχή δηλώνουμε έναν **sampler** με τη βοήθεια του οποίου θα μπορέσουμε να δώσουμε κάποιες ρυθμίσεις για την εμφάνιση του ήλιου. Οι ρυθμίσεις που ακολουθούν μέσα σε αυτή την δήλωση, “διατάζουν” το texture που θα χρησιμοποιηθεί να εμφανιστεί με υψηλή ποιότητα ακόμα και από πολύ κοντινές αποστάσεις. Στην συνέχεια ακολουθούν και άλλες ρυθμίσεις. Στην συνέχεια βλέπουμε πως ως είσοδο (VS\_INPUT), έχουμε ορίσει την θέση και τις συντεταγμένες του texture και ως έξοδο τα ίδια δεδομένα. Πριν προχωρήσουμε παρακάτω θα περιγράψουμε λίγο την μεταβλητή **TEXCOORD0**. Με αυτή την μεταβλητή μπορούμε να δηλώσουμε στην μηχανή γραφικών ποιο κομμάτι του texture που εισάγουμε ως είσοδο να είναι φανερό στον χρήστη και ποιο όχι. Αυτές οι συντεταγμένες συχνά αναφέρονται και ως **UV Coordinates**. Ξεκινάνε από τον αριθμό 0 και καταλήγουν στον αριθμό 1. Ένα παράδειγμα φαίνεται παρακάτω:





**Εικόνα 63.** Texture Coordinates.

Βλέπουμε πως στην προηγούμενη εικόνα φαίνονται καθαρά οι συντεταγμένες που αναφέραμε ποιο πριν και τι αυτές αντιπροσωπεύουν. Πλέον είμαστε σε θέση να δηλώσουμε ποιο κομμάτι του texture θέλουμε να χρησιμοποιήσουμε.

Συνεχίζοντας με το εφέ του ηλίου, θα προσπαθήσουμε να τον κάνει να λάμπει περισσότερο όταν τον κοιτάμε απευθείας, καθώς και άλλες ρυθμίσεις για το τελικό φως που θα μας δίνει:

```
float4 Texture(PS_INPUT Input) : COLOR0
{
    float4 FinalColor = tex2D(TextureSampler, Input.Texcoord);
    FinalColor.rgb += (1.0 - FinalColor.a) / 2;
    FinalColor.rgb += glow;

    return FinalColor * Alpha;
};
```

**Κώδικας 49.** Ρυθμίσεις φωτισμού ήλιου.

Στον παρακάτω κώδικα βλέπουμε πώς δηλώνουμε μία μεταβλητή τύπου float4 η οποία θα κρατά τον τελικό μας φωτισμό. Με την βοήθεια κάποιων εξωγενών μεταβλητών που λαμβάνουμε, δηλαδή διαφορετικές ανάλογα με την κατάσταση του παιχνιδιού,

μπορούμε να επιστρέψουμε στο τέλος το “τελικό χρώμα” του ήλιου. Στην συνέχεια “χτίζουμε” αυτές τις δύο μεθόδους με τον παρακάτω κώδικα:

```
technique TransformTexture
{
    pass P0
    {
        AlphaBlendEnable = true;
        SrcBlend = SrcAlpha;
        DestBlend = InvSrcAlpha;
        VertexShader = compile vs_2_0 Transform();
        PixelShader = compile ps_2_0 Texture();
    }
}
```

**Κώδικας 50.** Ολοκλήρωση εντολών.

Δίνουμε αρχικά κάποιο όνομα για την τεχνική που χρησιμοποιείται (TransformTexture) και έπειτα στο τέλος δηλώνουμε ποια συνάρτηση θα επιτελεί την δουλειά του Pixel Shader και ποια του Vertex Shader.

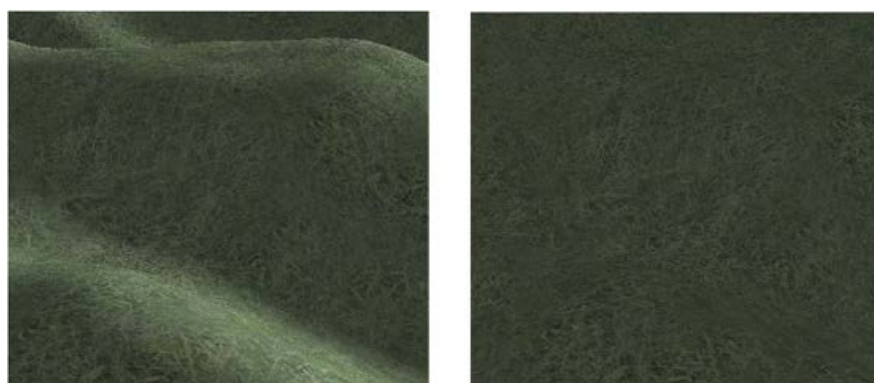
The logo for XNA (Xbox Game Studio) features the lowercase letters 'xna' in a grey, sans-serif font. The letter 'x' is stylized with a blue and orange gradient and a small circular detail at its top right. The background of the entire page is a soft-focus landscape of rolling green hills under a pale blue sky.

1 2 3  
4 5 6 7  
8 9 10

Πριν προχωρήσουμε στο τοπίο και πώς αυτό υλοποιήθηκε, θα μιλήσουμε για μερικές μαθηματικές μεθόδους που χρησιμοποιήθηκαν σε αυτήν την εργασία και βοηθούν σε διάφορες λειτουργίες του τοπίου. Αυτές οι μαθηματικές μέθοδοι υλοποιούνται μέσα στην κλάση **MathExtra**.

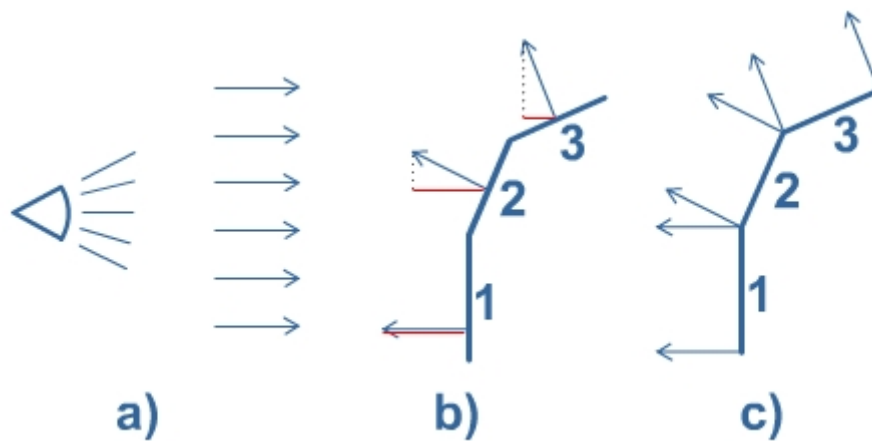
Αρχικά θα ασχοληθούμε με τον φωτισμό του τοπίου. Μέσα στην κλάση **MathExtra** υπάρχει μία συνάρτηση η οποία παίρνει ως όρισμα ένα τρίγωνο και επιστρέφει τον φωτισμό του. Συχνά ο φωτισμός των τριγώνων αναφέρεται ως **normal**, για αυτό και αυτή η συνάρτηση αυτή ονομάζεται **getNormal()**. Πριν προχωρήσουμε, όμως, στην υλοποίηση της συνάρτησης αυτής θα μιλήσουμε γενικά για τον φωτισμό στα τρίγωνα μίας μηχανής γραφικών.

Εάν παρατηρήσαμε προηγούμενες εικόνες από τη πτυχιακή εργασία θα διαπιστώσαμε πως ακόμα και σε ένα terrain με χρώματα, δεν καταφέρνουμε να έχουμε την αίσθηση του βάθους. Παρατηρήστε την διαφορά στις δύο παρακάτω εικόνες:



**Εικόνα 64.** Χωρίς φωτισμό η αίσθηση της τρίτης διάστασης χάνεται.

Οι δύο αυτές εικόνες αποτελούν το ίδιο στιγμιότυπο από ένα σημείο του τοπίου. Αριστερά, με χρήση φωτισμού το μάτι μας αρχίζει να καταλαβαίνει την έννοια του βάθους. Δεξιά, κάτι τέτοιο δεν γίνεται. Πώς, όμως, μπορούμε να εισάγουμε την έννοια του φωτισμού στην εφαρμογή και πόσο φωτισμό θα πρέπει να “ρίξουμε” κάθε φορά ώστε να είναι ρεαλιστικό; Κάτι τέτοιο δεν είναι και πολύ δύσκολο να γίνει. Σκεφτείτε το φως του ηλίου το οποίο ταξιδεύει προς μία κατεύθυνση. Ο ήλιος γενικά αναφέρεται ως κατευθυντική πηγή φωτός. Κάτι τέτοιο παρουσιάζεται στην παρακάτω εικόνα:



**Εικόνα 65.** Το φως του ήλιου ταξιδεύει προς μία κατεύθυνση. Η εικόνα δείχνει το ποσοστό φωτός που πρέπει να αντανάκλασει. Μεγαλύτερη απόσταση μπλε γραμμής σημαίνει και λιγότερος φωτισμός στην συγκεκριμένη επιφάνεια.

Παρατηρώντας την προηγούμενη εικόνα γίνεται κατανοητό ότι πρέπει να εισάγουμε στην μηχανή γραφικών την έννοια του ποσοστού αντανάκλασης φωτός. Εάν στο σχήμα **b** παρατηρήσουμε τις κόκκινες γραμμές και τις αποστάσεις τους από τις μπλε θα δούμε ότι αυτό το ποσόν είναι αρκετό για αυτή την έννοια, η απόσταση δηλαδή (τα συνημίτονα) των κόκκινων γραμμών με τις μπλε. Εμείς το μόνο που έχουμε να κάνουμε είναι να δώσουμε στο XNA την μπλε γραμμή και αυτό κάνει όλους τους απαραίτητους υπολογισμούς για εμάς. Πώς όμως εμείς θα βρούμε τον φωτισμό για κάθε ένα από τα τρίγωνα μας; Πολύ απλά. Εάν γνωρίζουμε τις δύο πλευρές από ένα τρίγωνο τότε μπορούμε να πάρουμε το γινόμενο των πλευρών και αυτό θα μας δώσει τον φωτισμό τους. Έτσι θα κατασκευάσουμε μία συνάρτηση η οποία θα παίρνει τις τρεις κορυφές ενός τριγώνου, έπειτα με αυτές τις τρεις κορυφές θα βρίσκουμε τις δύο πλευρές του και τέλος το γινόμενο των πλευρών αυτών θα είναι ο φωτισμός τον οποίο και θα επιστρέψουμε. Έτσι έχουμε τον παρακάτω κώδικα:

```

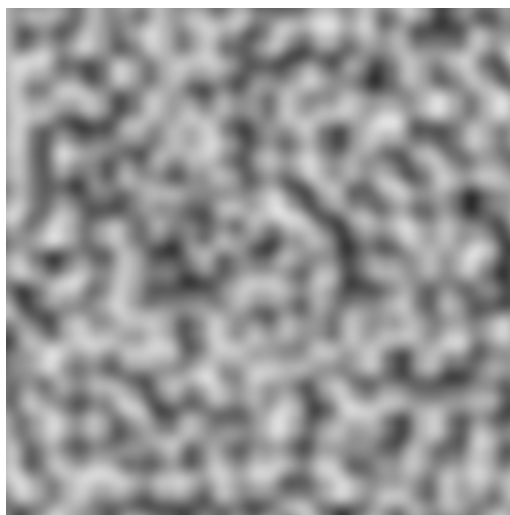
11  public static Vector3 GetNormal(Vector3 p1, Vector3 p2, Vector3 p3)
12  {
13      Vector3 v1 = p2 - p1;
14      Vector3 v2 = p1 - p3;
15
16      Vector3 norm = Vector3.Cross(v1, v2);
17      norm.Normalize();
18
19      return norm;
20  }

```

**Κώδικας 51.** Ο φωτισμός του τοπίου.

Η συνάρτηση, `GetNormal`, παίρνει ως όρισμα ένα τρίγωνο, δηλαδή τις τρεις κορυφές του. Στις γραμμές 13 και 14 βρίσκουμε τις δύο πλευρές του και τέλος στην γραμμή 16 βρίσκουμε το γινόμενο (**Cross Product**). Στην επόμενη γραμμή κανονικοποιούμε το αποτέλεσμα για να μην βγούμε εκτός ορίων, όπως είδαμε και σε προηγούμενο παράδειγμα, και τέλος το επιστρέφουμε.

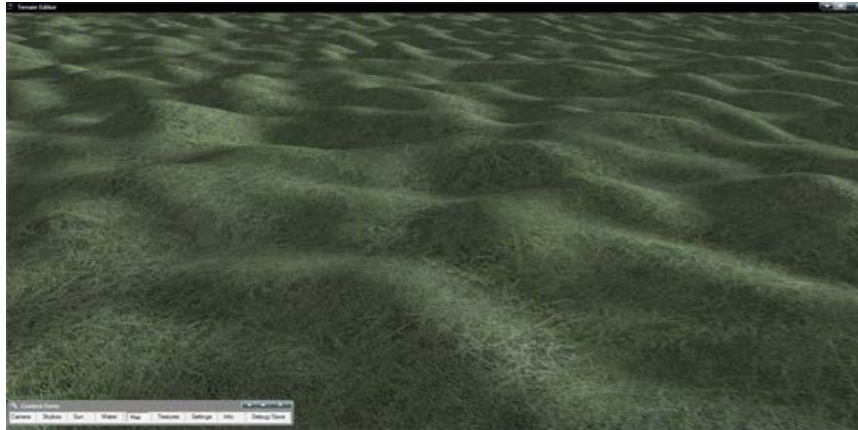
Στην συνέχεια, θα μιλήσουμε για μία πολύ σημαντική συνάρτηση, η οποία χρησιμοποιείται κατά κόρον και στην ψηφιακή επεξεργασία εικόνας. Αυτή η συνάρτηση φέρει τον τίτλο “**θόρυβος Perlin**” ή “**Perlin Noise**”. Ο θόρυβος Perlin είναι ένα εφέ το οποίο έχει δημιουργηθεί από υπολογιστές για υπολογιστές από τον Ken Perlin. Ανήκει στην κατηγορία των διαδικαστικών υφών οι οποίες μας βοηθούν να αυξήσουμε τον ρεαλισμό των αντικειμένων στα γραφικά των υπολογιστών. Αυτή η συνάρτηση μπορεί να έχει “ψευδοτυχαία” εμφάνιση, τα στοιχεία όμως που την απαρτίζουν έχουν το ίδιο μέγεθος, όπως φαίνεται στην παρακάτω εικόνα:



**Εικόνα 66.** Ο θόρυβος Perlin.

Ο θόρυβος Perlin θυμίζει το εφέ της τηλεόρασης, γνωστό και ως χιόνια στους περισσότερους από εμάς. Η ιδιότητα του θορύβου αυτού και συγκεκριμένα των ίσων μεγέθους στοιχείων που τον απαρτίζουν, έχει πολλές καλές συνέπειες. Καταρχήν μας διευκολύνει στην εύκολη και απλή διαχείρισή του. Στην συνέχεια, και λόγω των ίσων στοιχείων του, μπορούν να τοποθετηθούν σε μαθηματικές εκφράσεις έτσι ώστε να δημιουργήσουν σειρές από όμορφες διαδικαστικές υφές. Ο θόρυβος Perlin επίσης χρησιμοποιείται για την δημιουργία καπνού, φωτιάς και σύννεφων. Στην εργασία αυτή αυτός ο θόρυβος θα μας βοηθήσει στην δημιουργία ενός τυχαίου τοπίου. Εμείς απλώς θα λαμβάνουμε τις κλιμακώσεις του μαύρου χρώματος, από 0 έως 255 και αυτές με την

σειρά τους θα αντιπροσωπεύουν τα ύψη των λόφων. Ένα παράδειγμα δημιουργίας ενός τυχαίου τοπίου με θόρυβο Perlin, παρουσιάζεται παρακάτω:



**Εικόνα 67.** Δημιουργία τυχαίου τοπίου με θόρυβο Perlin.

Η συνάρτηση θορύβου Perlin υλοποιείται πολύ απλά. Καταρχήν είναι μία συνάρτηση δύο διαστάσεων. Έτσι εμείς πρέπει να κατασκευάσουμε μία συνάρτηση με δύο ορίσματα τα οποία θα είναι οι ανεξάρτητες μεταβλητές και ένα όρισμα τυχαίου αριθμού. Έτσι έχουμε τον εξής κώδικα:

```
24 public static float PerlinNoise(int x, int y, int random)
25 {
26     Random rand = new Random();
27     int n = x + y * 57 + rand.Next(0, random) * 131;
28     n = (n << 13) ^ n;
29     return (1.0f - ((n * (n * n * 15731 + 789221) + 1376312589) & 0x7fffffff) * 0.00000000931322574615478515625f);
30 }
```

**Κώδικας 52.** Η συνάρτηση θορύβου Perlin.

Βέβαια υπάρχουν αρκετές ακόμη υλοποιήσεις του κώδικα Perlin. Η συγκεκριμένη αυτή συνάρτηση που υλοποιήθηκε σε αυτήν την εργασία είναι χρήσιμη για να λάβουμε τα ύψη των τριγώνων. Όλα τα υπόλοιπα γίνονται αυτόματα από την μηχανή γραφικών XNA. Η συνάρτηση αυτή έχει ως βασικό στοιχείο βέβαια την τυχαιότητα.



xna



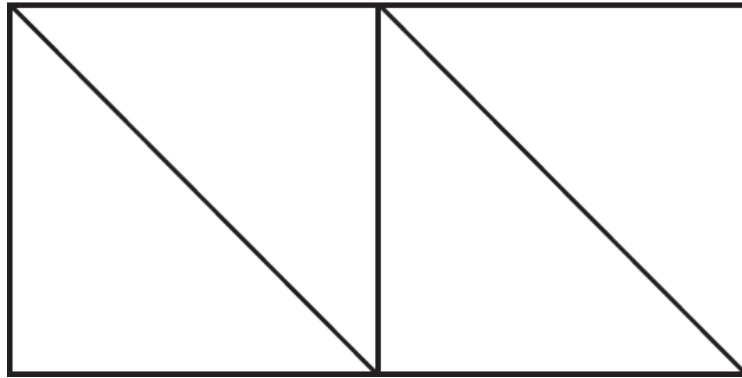
Το τοπίο στην εφαρμογή που αναπτύξαμε κατασκευάζεται με την βοήθεια ενός τετραδικού δέντρου. Όπως έχουμε πει σε προηγούμενο κεφάλαιο το τοπίο για τα παιχνίδια είναι από τα πιο σημαντικά κομμάτια τους και καταναλώνει πολλούς πόρους διότι πρέπει συνεχώς να είναι εμφανές στον χρήστη αλλά και να του δίνει την αίσθηση του απέραντου. Εάν προσπαθούσαμε να φτιάξουμε το τοπίο απλά, δηλαδή δηλώνοντας απλά τρίγωνα θα ήταν αδύνατο. Καταρχήν δεν υπάρχει κάρτα γραφικών ακόμα στην αγορά η οποία διαθέτει τόση μεγάλη μνήμη τόση ώστε να “ζωγραφίσει” ένα αξιόλογο τοπίο. Τα εκατομμύρια των τριγώνων που χρειάζεται το τοπίο για να κατασκευαστεί θα φόρτωναν την κάρτα γραφικών και οι σκηνές ανά δευτερόλεπτο, δηλαδή η απόδοση θα έπεφτε δραματικά. Πώς όμως καταφέρνουν να απεικονίσουν μεγάλα τοπία γρήγορα και με ακρίβεια; Για αυτό υπάρχουν δύο βασικές λύσεις. Η μία είναι η λύση που χρησιμοποιήθηκε σε αυτή την εργασία και ακούει στο όνομα **τετραδικό δέντρο** και η άλλη, πιο ειδικευμένη και σύγχρονη το τοπίο **ROAM**.

Πρώτα θα δούμε πώς λειτουργούν βασικά αυτές οι μέθοδοι και έπειτα θα μιλήσουμε και για τις δύο.

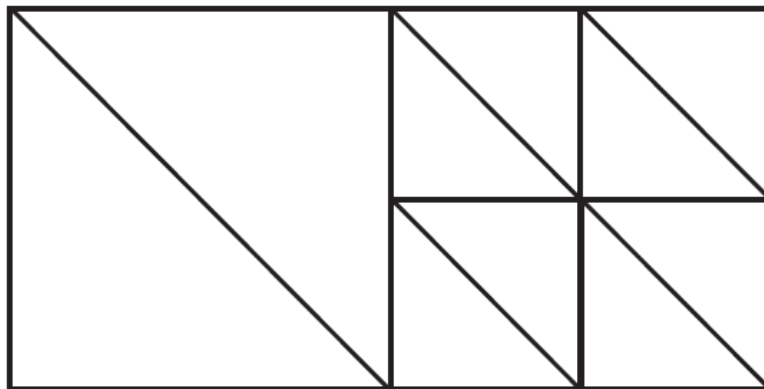
Να πούμε αρχικά πως το κοινό και αυτών των δύο μεθόδων είναι το “παιχνίδι” με την κάμερα. Η θέση και το μάτι της κάμερας παίζει σημαντικό ρόλο και στις δύο μεθόδους. Μία πολύ απλή λύση είναι ότι δεν θα θέλαμε σε καμία περίπτωση να δημιουργούνται και να επεξεργάζονται κομμάτια του τοπίου, δηλαδή τρίγωνα, τα οποία δεν βλέπουμε. Αυτό από μόνο του μας έχει βοηθήσει αμέσως να σώσουμε τεράστιο bandwidth της κάρτας γραφικών. Το τι γίνεται έπειτα με τις κάθε μία από αυτές τις μεθόδους περιγράφεται παρακάτω.

Αρχικά θα ασχοληθούμε με την μέθοδο ROAM. Τα αρχικά προέρχονται από τις λέξεις: Real-Time Camera-Dependent Optimally Adapting Mesh.

Αυτή η μέθοδος είναι η βέλτιστη λύση για τα τοπία αλλά και η πιο δύσκολη. Όλο το τοπίο ξεκινάει με δύο απλά τρίγωνα. Από εκεί και πέρα και ανάλογα με την θέση της κάμερας τα τρίγωνα αποφασίζουν εάν θα σπάσουν σε μικρότερα για να μας παρέχουν μεγαλύτερη λεπτομέρεια. Για παράδειγμα, εάν φανταστούμε ότι η εικόνα 68 αντιπροσώπευε ένα τοπίο, μετακινούμενοι λίγο πιο κοντά σε αυτό θα έπρεπε να αυξήσουμε την λεπτομέρεια για παράδειγμα στην δεξιά πλευρά όπως φαίνεται στην εικόνα 69:



**Εικόνα 68.** Το αρχικό τοπίο με τη μέθοδο ROAM.

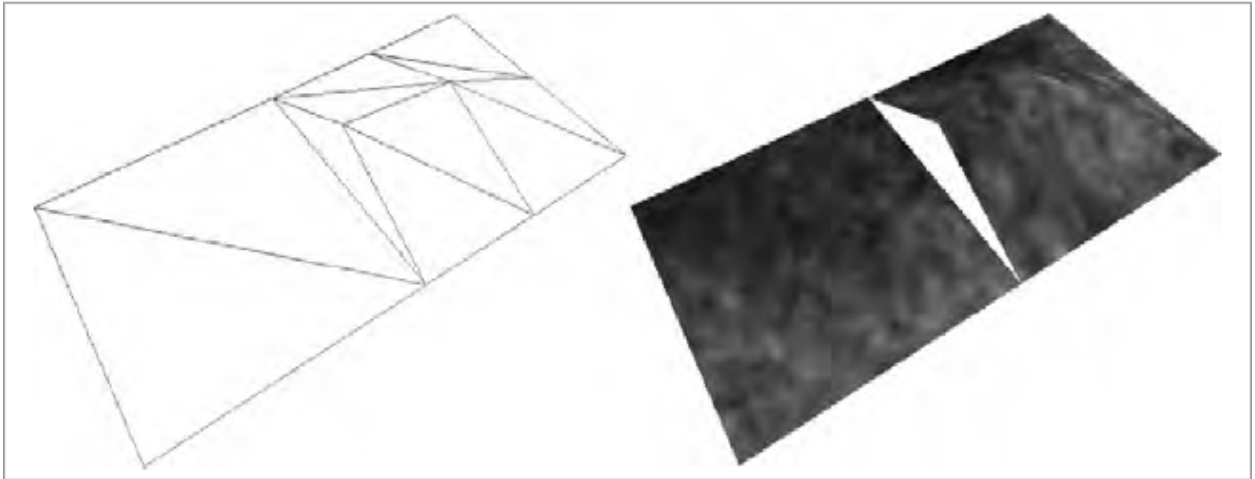


**Εικόνα 69.** Αλλάζει η θέση της κάμερας οπότε τα τρίγωνα στα δεξιά αποφασίζουν ότι πρέπει να σπάσουν σε μικρότερα ώστε να φανούν οι λεπτομέρειες.

Βέβαια, αυτή η τεχνική μας προσφέρει πάρα πολλά πλεονεκτήματα και διαχειρίζεται ορθά τους πόρους του συστήματός μας γιατί από την μία πλευρά δεν επεξεργάζονται κομμάτια του τοπίου τα οποία δεν φαίνονται και από την άλλη δεν εμφανίζονται λεπτομέρειες στο τοπίο εάν δεν τις χρειαζόμαστε.

Όμως, αυτή η μέθοδος δημιουργεί ένα μικρό προβληματάκι το οποίο λύνεται όπως θα δούμε στην συνέχεια.

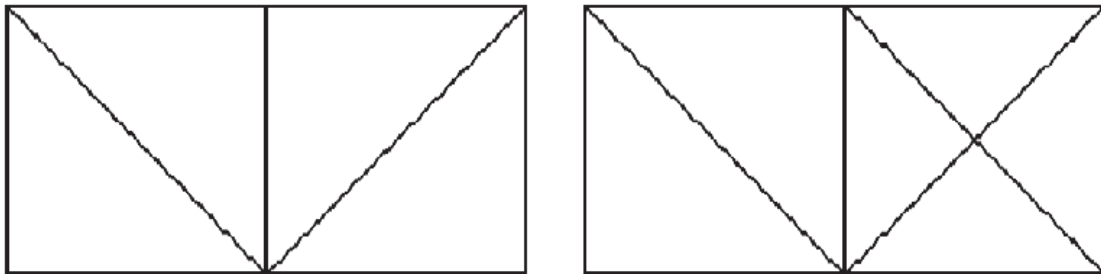
Σε περίπτωση που ένα επίπεδο τριγώνων καταστεί σημαντικό να εμφανίσει τις λεπτομέρειές του, δίχως αυτό να κρίνεται απαραίτητο σε άλλο επίπεδο, θα έχουμε το εξής πρόβλημα που παρουσιάζεται στη παρακάτω εικόνα:



**Εικόνα 70.** Δημιουργία κενών λόγω διαφορετικού επιπέδου.

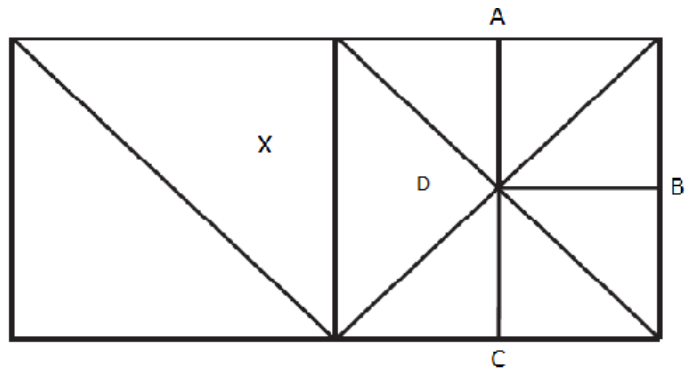
Η λύση για να ξεπεράσουμε το πρόβλημα αυτό είναι απλή: αντί να δουλεύουμε με επίπεδα, δηλαδή με όλα τα τρίγωνα που περιλαμβάνονται μέσα σε κάθε επίπεδο, θα δουλέψουμε με κάθε τρίγωνο ξεχωριστά. Οπότε κάθε φορά που ένα τρίγωνο χρειαστεί να αυξήσει την λεπτομέρειά του, αυτό θα σπάει σε δύο μικρότερα. Αυτό μας δίνει μεγαλύτερο έλεγχο πάνω στα τρίγωνά μας.

Στην παρακάτω εικόνα φαίνεται η τεχνική αυτή:



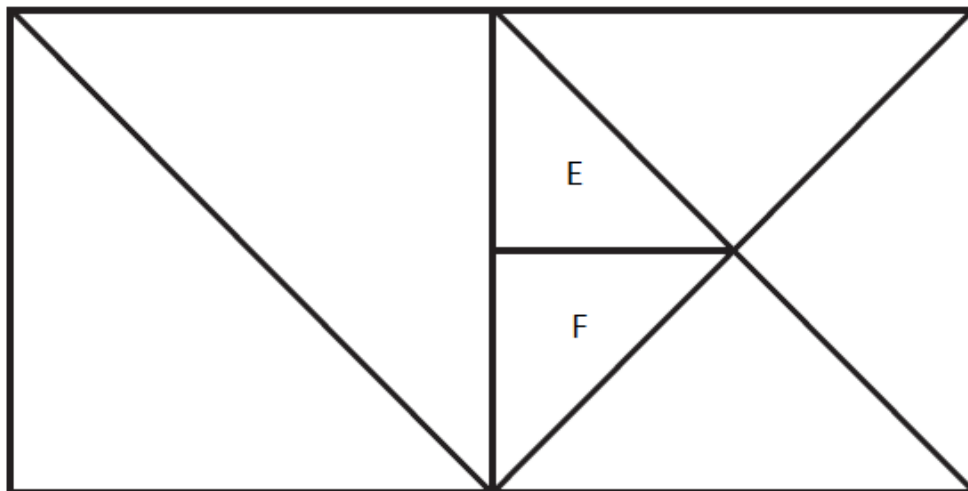
**Εικόνα 71.** Δουλεύοντας με κάθε τρίγωνο ξεχωριστά

Τελικά, αυτή η μέθοδος δεν θα προκαλέσει κενά στο τοπίο μας. Και πάλι όμως μπορεί. Πριν συνεχίσουμε παρακάτω, ας εξηγήσουμε λίγο πιο περιγραφικά για ποιο λόγο δημιουργούνται τα κενά. Θα παρατηρήσαμε από τις προηγούμενες εικόνες πως τα κενά δημιουργούνται κοντά σε τρίγωνα τα οποία δεν έχουν σπάσει. Δηλαδή τα τρίγωνα τα οποία πολύ απλά δεν ακουμπάνε με άλλα δεν έχουν κανένα απολύτως πρόβλημα. Αυτά που έχουν πρόβλημα είναι αυτά που γειτονεύουν με άλλα τρίγωνα. Οπότε τελικά για να λυθεί το πρόβλημα αυτό θα πρέπει πριν σπάσουμε τα τρίγωνά μας να βεβαιωθούμε ότι τα γειτονικά τους είναι σπασμένα. Στην παρακάτω εικόνα φαίνονται τρία ασφαλή “σπασίματα”:



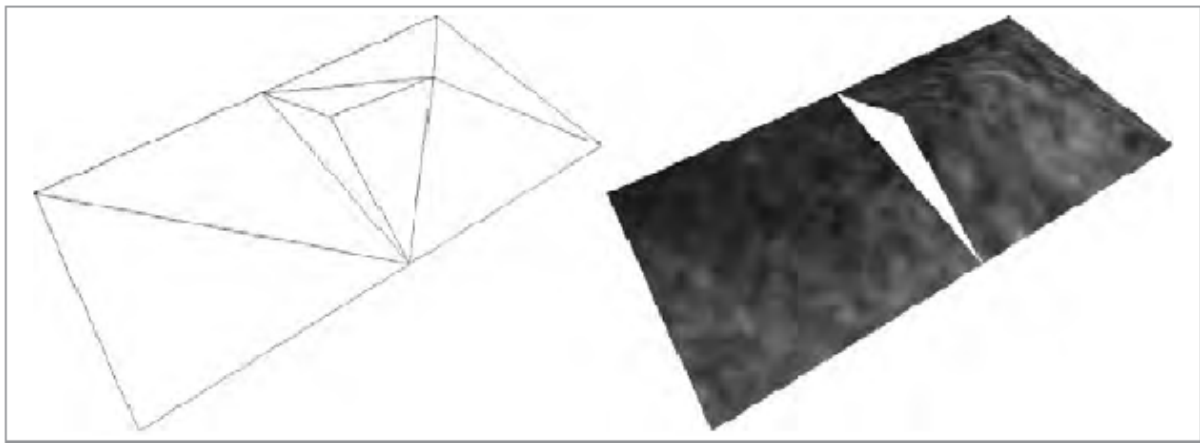
**Εικόνα 72.** Ακόμη τρία ασφαλή σπασίματα.

Το τρίγωνο το οποίο μας προκαλεί το πρόβλημα με τα σπασίματα, όπως αναφέραμε και πριν, είναι αυτό που γειτονεύει με άλλα ενός άλλου επιπέδου, αυτό το τρίγωνο στην εικόνα μας είναι το τρίγωνο που σημειώνεται με το γράμμα “X” στην εικόνα 72. Στην ίδια εικόνα φαίνονται επίσης τρία ασφαλή σπασίματα στο δεξί επίπεδο. Αυτά είναι το “A”, το “B” και το “C”. Εάν τώρα το τρίγωνο “D” σπάσει θα προκαλέσουμε σπάσιμο. Δηλαδή όπως στην παρακάτω εικόνα:



**Εικόνα 73.** Δημιουργία σπασίματος.

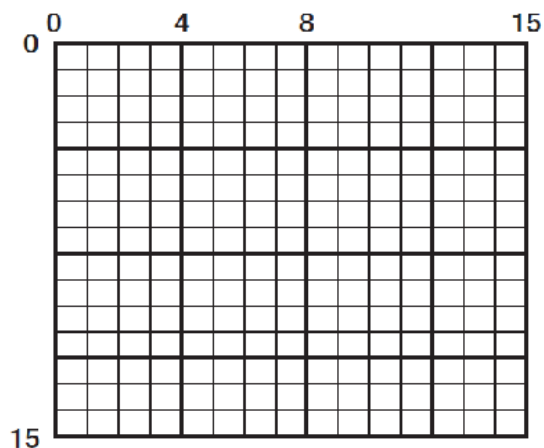
Το αρχικό τρίγωνο έχει σπάσει σε “E” και “F” χωρίς να πάρουμε υπόψη την γειτονία του. Αυτό αμέσως θα προκαλέσει το πρόβλημα που φαίνεται στην παρακάτω εικόνα:



**Εικόνα 74.** Ακόμα ένα σπάσιμο λόγω έλλειψης ελέγχου γειτονίας.

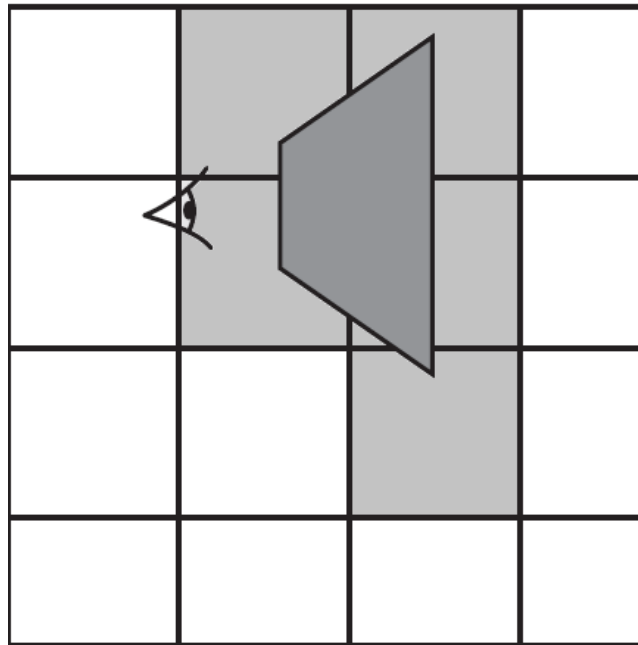
Προχωρώντας θα δούμε αναλυτικά τη μέθοδο που χρησιμοποιήθηκε για αυτήν την εφαρμογή, η οποία χρησιμοποιεί μόνο το τετραδικό δέντρο ώστε να “κρύψει” από την κάρτα γραφικών κομμάτια του τοπίου τα οποία δεν τα βλέπουμε εμείς ή καλύτερα η κάμερα.

Αυτό θα ελαφρύνει κατά πολύ τον φόρτο εργασίας της κάρτας γραφικών. Η μέθοδος χρησιμοποίησης ενός τετραδικού δέντρου είναι απλή. Είναι η ίδια διαδικασία με ένα **octree**, όπου σπάζουμε τα αρχικά μεγάλα κομμάτια του τοπίου σε μικρότερα μέχρι ενός ορισμένου σημείου. Κάτι τέτοιο φαίνεται στην παρακάτω εικόνα, όπου το αρχικό πλέγμα 16X16 έσπασε σε τέσσερα μικρότερα κομμάτια, τα οποία με την σειρά τους σπάσανε σε τέσσερα μικρότερα:



**Εικόνα 75.** Ένα τετραδικό δέντρο.

Στην συνέχεια θέλουμε, όπως είπαμε αρχικά, αυτό το πλέγμα να αλληλεπιδρά με την κάμερα μας. Κάτι τέτοιο θα γίνει με τον τρόπο που φαίνεται στην παρακάτω εικόνα όπου τα σημεία τα οποία πρέπει να εμφανιστούν φωτίζονται με γκρι χρώμα:



**Εικόνα 76.** Η λειτουργία του τετραδικού δέντρου.

Όπως είπαμε και πριν, το τετραδικό κέντρο είναι μία απλοποιημένη έκδοση του octree. Μία περιοχή ενός τετραδικού δέντρου, ένα κουτί όπως φαίνεται στην προηγούμενη εικόνα, χρειάζεται να χωριστεί σε τέσσερα μικρότερα ενώ σε ένα octree σε οχτώ κύβους. Επίσης σε ένα octree θα πρέπει να κρατάμε όλες τις θέσεις των “παιδιών” ενώ στο τετραδικό δέντρο δεν χρειάζεται να ασχοληθούμε με κάτι τέτοιο.

Το τετραδικό δέντρο το υλοποιεί η κλάση **QuadTree** μέσα στην εφαρμογή μας. Αρχικά θα κατασκευάσουμε τον κώδικα για τον κάθε ένα κόμβο:

```

11 public class Node
12 {
13     public NodeType Type;
14     public BoundingBox BoundingCoordinates = new BoundingBox();
15     public int[] Branches = new int[4];
16     public BoundingBox boundingBox;
17     public List<int> TriangleIDs = new List<int>();
18     public int ID;
19     public int ParentID;
20 }

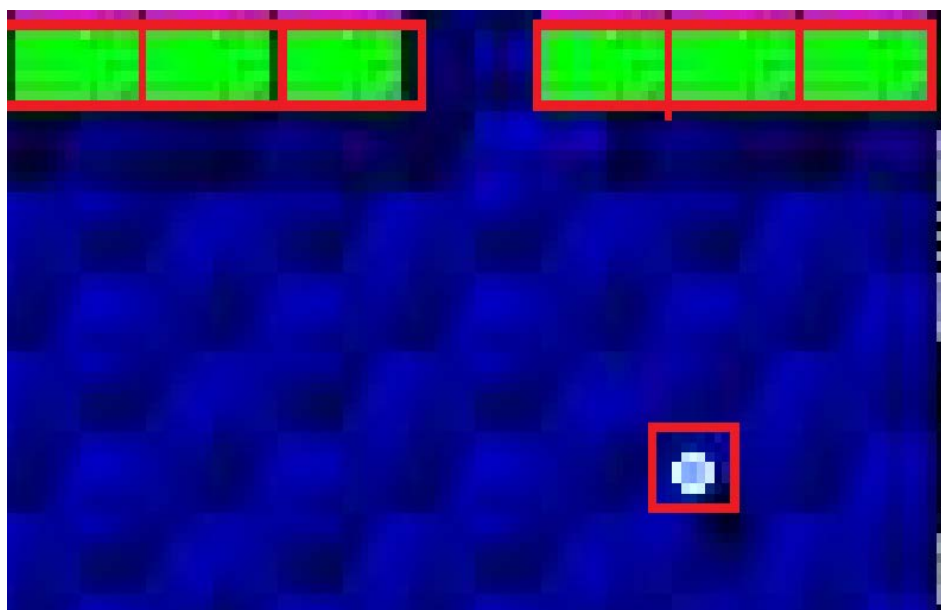
22 public enum NodeType
23 {
24     Leaf = 0,
25     Node = 1
26 }

```

**Κώδικας 53.** Ο κώδικας που αντιπροσωπεύει τον κάθε κόμβο.

Στην γραμμή 13 χρησιμοποιούμε μία μεταβλητή που θα αναγνωρίζει τον κόμβο μας. Αυτός μπορεί να είναι όπως φαίνεται στην δήλωση στην γραμμή 22 έως 26, είτε ένας άλλος κόμβος, είτε μία απόληξή του.

Στην γραμμή 14 δηλώνουμε έναν τύπο μεταβλητής **BoundingBox**, για να το χρησιμοποιήσουμε αργότερα για τον εντοπισμό σύγκρουσης με το ποντίκι. Το BoundingBox προσφέρεται από την βιβλιοθήκη του XNA και είναι ένας χώρος στον οποίο περικλείονται διάφορα πράγματα, έτσι ώστε να κατανοούμε τις συγκρούσεις κάθε φορά. Μέσα σε αυτό το “κουτί” μπορούμε να τοποθετήσουμε οτιδήποτε χρειαζόμαστε για εντοπισμούς συγκρούσεων. Σαν ένα παράδειγμα να αναφέρουμε την περίπτωση ενός πολύ γνωστού παιχνιδιού του **Arkanoid**, στο οποίο για να μπορούσαμε να καταλάβουμε κάθε φορά εάν υπάρχει σύγκρουση μπάλας με ένα block θα τοποθετούσαμε το κάθε ένα block σε ένα BoundingBox και το μπαλάκι σε ένα άλλο. Αυτό φαίνεται στην παρακάτω εικόνα:



**Εικόνα 77.** Εντοπισμός σύγκρουσης με χρήση BoundingBoxes.

Τέλος, αξίζει να αναφέρουμε ότι τα τετράγωνα δεν είναι αρκετά μερικές φορές για τον εντοπισμό σύγκρουσης. Για παράδειγμα, για πιο περίπλοκα μοντέλα, όπως για ανθρώπους, στην πραγματικότητα γίνεται χρήση πολλών διαφορετικών σχημάτων, όπως είναι σφαίρες και άλλα.

Συνεχίζοντας στον κώδικα 53, στην γραμμή 15 δηλώνουμε την μεταβλητή Branches η οποία και θα κρατάει τα 4 σημεία του κατώτερου επιπέδου, δηλαδή τα παιδιά του κάθε κόμβου. Τέλος, μία σημαντική δήλωση βρίσκεται στη γραμμή 17, όπου εκεί θα κρατάμε

κάθε φορά τα τρίγωνα που βρίσκονται μέσα σε αυτόν τον κόμβο ή το κουτί όπως είπαμε πριν.

Στον παρακάτω κώδικα παρουσιάζεται ο Constructor της κλάσης QuadTree:

```
46 public QuadTree(float MapWidth, float MapHeight, Vector2 CellSize, Map.Tri[] Triangles)
```

**Κώδικας 54.** Ο constructor της κλάσης QuadTree.

Όπως καταλαβαίνουμε και από τον προηγούμενη δήλωση, για να κατασκευάσουμε το τετραδικό δέντρο χρειάζεται να δηλώσουμε το ύψος και το πλάτος του χάρτη, το μέγεθος που θα έχει το κάθε κουτί με τρίγωνα και τέλος ένα **δέντρο** με τρίγωνα. Η μεταβλητή **Map** της γλώσσας C# υλοποιείται με δέντρο και μας βοηθά να εντοπίσουμε τρίγωνα ξέροντας κάποιο τους χαρακτηριστικό, για παράδειγμα το ID τους.

Αρχικά, λοιπόν, υπολογίζουμε το μήκος και το πλάτος του πλέγματος από τα μεγέθη που μας δίνονται διαιρώντας με το μέγεθος κάθε κελιού:

```
51 GridWidth = MapWidth / CellSize.X;  
52 GridHeight = MapHeight / CellSize.Y;
```

**Κώδικας 55.** Εύρεση μήκους και πλάτους.

Εφόσον πια γνωρίζουμε το μήκος και το πλάτος μπορούμε να βρούμε των αριθμό των “φύλλων” που απαρτίζουν το τετραδικό μας δέντρο και έπειτα τους κόμβους του. Τέλος, κατασκευάζουμε την λίστα κόμβων:

```
54 int numberOfLeaves = ((int)GridWidth / 4) * ((int)GridHeight / 4);  
55  
56 int numberOfNodes = CalcNodeNum(numberOfLeaves, 4);  
57 NodeList = new List<Node>(numberOfNodes);
```

**Κώδικας 56.** Εύρεση φύλλων και κόμβων-Δημιουργία λίστας.

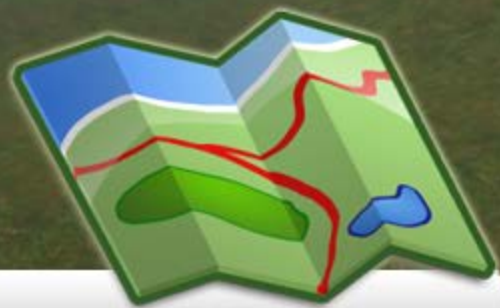
Τελευταία μας κίνηση είναι να δημιουργήσουμε ένα μεγάλο “κουτί”, όπως είπαμε στην αρχή (BoundingBox). Αυτό το κουτί θα σπάει σε μικρότερα μέχρι να φτάσει το ζητούμενο μέγεθος του μεγέθους κελιού (CellSize). Στην συνέχεια υπολογίζουμε και αποθηκεύουμε τις συντεταγμένες όλων των κουτιών που δημιουργήθηκαν.



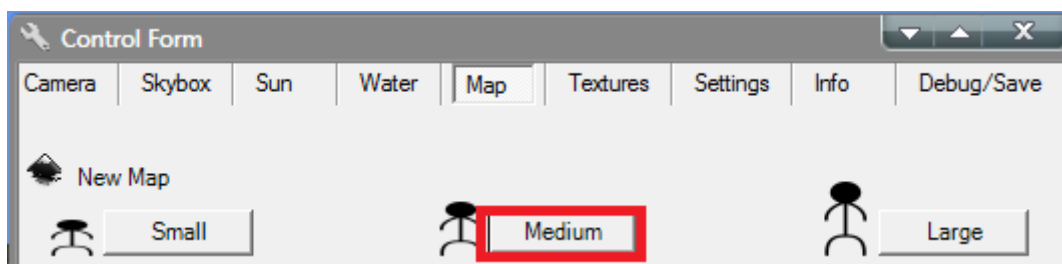
Μέσα σε αυτήν την κλάση υλοποιείται επίσης μία συνάρτηση η οποία μας βοηθά να κατανοήσουμε εάν κάποιο τρίγωνο βρίσκεται μέσα σε κάποιο κουτί. Η συνάρτηση αυτή έχει το όνομα **InsideBoundingBox** και επιστρέφει μία bool μεταβλητή εάν κάποιο τρίγωνο βρίσκεται μέσα σε ένα κουτί. Επίσης, μέσα σε αυτήν την κλάση υπάρχουν συναρτήσεις οι οποίες τρέχουν όταν αυτό καταστεί απαραίτητο. Για παράδειγμα εάν αλλάξουμε το μέγιστο ύψος του τοπίου θα πρέπει να μεγαλώσουμε τα κουτιά και ούτω κάθε εξής.



χρη



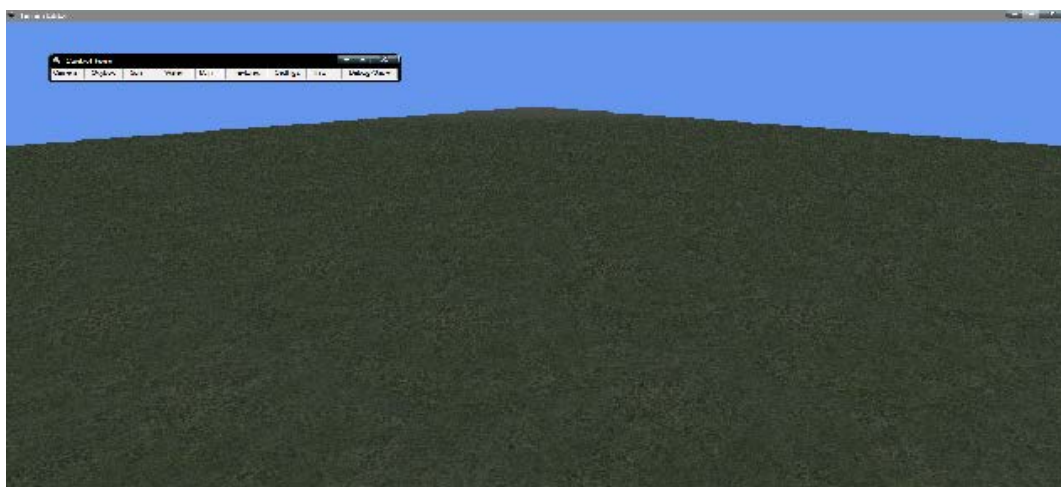
Η κλάση του τοπίου **Map** υλοποιεί όλες τις απαραίτητες λειτουργίες που χρειαζόμαστε για ένα τοπίο και χρησιμοποιεί ένα τετραδικό δέντρο για να διαχειριστεί τον εαυτό του. Οι λειτουργίες του είναι αρκετές: Μπορούμε να δημιουργήσουμε ένα μικρό, μεσαίο και τεράστιο κενό τοπίο. Έπειτα υπάρχουν συναρτήσεις θορύβου και συναρτήσεις χρήστη ώστε να αλλάξει η διαμόρφωσή του ανάλογα με τον επιλεγμένο θόρυβο. Θα δούμε σε γενικές γραμμές με ποια λογική λειτουργεί. Αρχικά ας πούμε πως δημιουργούμε ένα καινούριο τοπίο. Για να το κάνουμε αυτό θα πρέπει να πατήσουμε το κουμπί “Medium”, ας πούμε στην καρτέλα “Map”, για ένα τοπίο μεσαίου μεγέθους. Αυτό το κουμπί φαίνεται στην παρακάτω εικόνα:



**Εικόνα 78.** Δημιουργία ενός μεσαίου τοπίου.

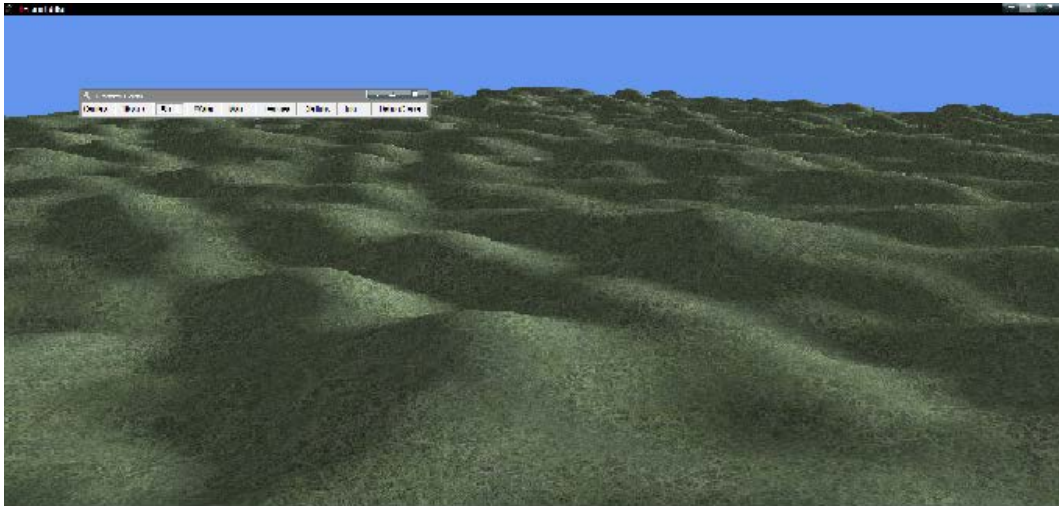
Μέχρι εδώ έχουν γίνει ήδη πολλά. Έχει δημιουργηθεί μία κλάση τετραδικού δέντρου, έχουν υπολογιστεί τα τρίγωνα, τα φύλλα οι κόμβοι, τα κουτιά που τα περικλείουν όλα αυτά, όπως επίσης και οι **συγκρούσεις** τους. Επίσης, όπως είχαμε πει και στην περίπτωση του ηλίου, ο χάρτης θα χρησιμοποιεί και αυτός την ενδιάμεση μνήμη (**Vertex/Index Buffer**) για λόγους εξοικονόμησης πόρων.

Εάν πατήσουμε το κουμπί που παρουσιάστηκε παραπάνω το πρόγραμμα θα αποκριθεί εμφανίζοντάς μας ένα τοπίο περίπου έτσι:



**Εικόνα 79.** Ένα τοπίο μεσαίου μεγέθους.

Το τοπίο φαίνεται πολύ καλό. Καθόλου ρεαλιστικό όμως. Για να προσομοιώσουμε μία ανώμαλη επιφάνεια θα προσθέσουμε **θόρυβο**. Για παράδειγμα, για να προσθέσουμε θόρυβο Perlin, αυτό το κάνουμε μέσα από την καρτέλα του χάρτη εάν πατήσουμε το κουμπί **Perlin Noise**. Το πρόγραμμα θα αποκριθεί περίπου έτσι:



**Εικόνα 80.** Τοπίο με θόρυβο Perlin.

Παρακάτω θα δούμε τις συναρτήσεις θορύβου. Οι συναρτήσεις θορύβου Perlin και τυχαίου θορύβου (Random), καθώς επίσης και οι συναρτήσεις επιλεγμένες από τον χρήστη υλοποιούνται στην συνάρτηση **RandomizeHeight**.

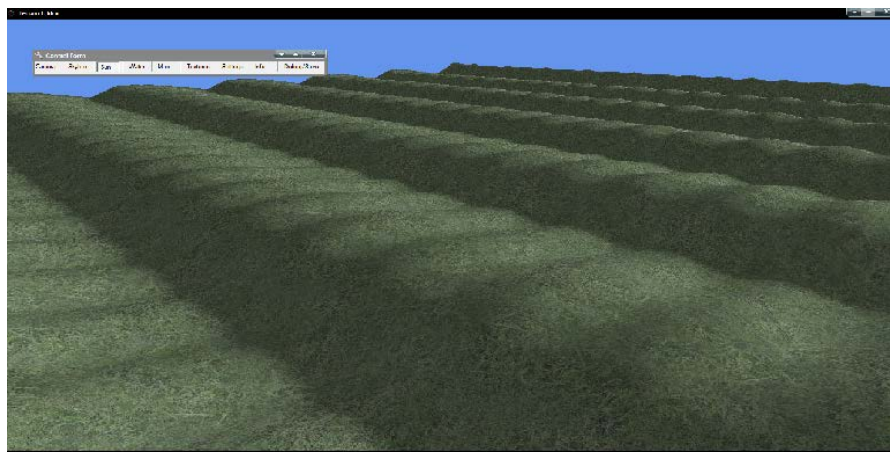
```
1050         switch (randomGenType)
1051         {
1052             case GenerationType.Random:
1053                 currentHeight = (float)random.Next(randomHeight.X, randomHeight.Y) * 0.01f;
1054                 break;
1055             case GenerationType.PerlinNoise:
1056                 currentHeight = MathExtra.PerlinNoise(x, y, 200) * 5.0f;
1057                 break;
1058             case GenerationType.FieldNoise:
1059                 currentHeight = (float)(Math.Sin(x));
1060                 break;
1061             case GenerationType.CustomNoise:
1062                 float first = 0 ;
1063                 float second = 0 ;
1064         }
```

**Κώδικας 57.** Συναρτήσεις θορύβου.

Καθώς επιλέγουμε διαφορετικό τύπο γεννήτριας θορύβου, η μεταβλητή στη γραμμή 1050 αλλάζει και ανάλογα με αυτήν καλείται η συνάρτηση για πρόσθεση θορύβου στο τοπίο. Οπότε έχουμε τρεις διαφορετικές γεννήτριες θορύβου:

- Την Perlin
- Την τυχαία
- Και επιλογή του χρήστη.

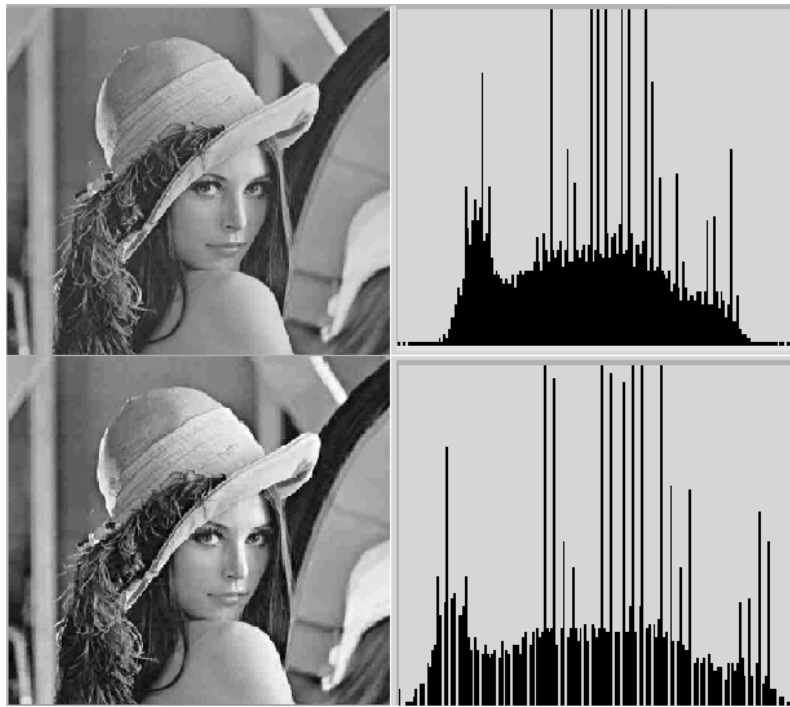
Ειδικά στην τελευταία γεννήτρια, ο χρήστης έχει την δυνατότητα να επιλέξει δύο διαφορετικές συναρτήσεις (ημίτονο, συνημίτονο, εφαπτομένη και άλλα), όπως επίσης και την πράξη που θα γίνει μεταξύ των συναρτήσεων ώστε να προστεθεί θόρυβος στο τοπίο. Ένα παράδειγμα πρόσθεσης ημιτόνου και εφαπτομένης φαίνεται στην επόμενη εικόνα:



**Εικόνα 81.** Θόρυβος επιλεγμένος από τον χρήστη.

Μία σημαντική λειτουργία του τοπίου είναι η εξομάλυνση, η οποία συναντάται και στην ψηφιακή επεξεργασία εικόνας. Στην δεύτερη περίπτωση έχοντας το ιστόγραμμα της εικόνας (τα χρώματα που υπάρχουν και πόσες φορές μέσα στην εικόνα), μπορούμε να κάνουμε εξομάλυνση των χρωμάτων ώστε να μην έχουμε απότομες μεταβολές. Στην δική μας περίπτωση, μπορούμε να εξομαλύνουμε τις κορυφές ώστε να μην υπάρχουν πολύ μεγάλες ή πολύ μικρές κορυφές, αυτός ο όρος συναντάται με την λέξη **smoothing**.

Για να κατανοήσουμε πώς δουλεύει αυτή η διαδικασία θα παρουσιάσουμε αρχικά δύο εικόνες. Η μία είναι η πρωτότυπη και η άλλη η εξισορροπημένη με τα ιστογράμματά τους:



**Εικόνα 82.** Η εικόνα πριν και μετά την εξισορρόπηση ιστογράμματος.

Παρατηρούμε σε αυτήν την περίπτωση ότι η εικόνα κατά μία έννοια “απλώνεται” και έτσι έχουμε ποιο “σωστά” χρώματα. Ο κώδικας της εφαρμογής μας κάνει παρόμοια λειτουργία επάνω στο τοπίο:

```

1179 public void SmoothHeightmap()
1180 {
1181     for (int y = 0; y < size.Y; y++)
1182     {
1183         for (int x = 0; x < size.X; x++)
1184         {
1185             int[] bitID = new int[5];
1186             bitID[0] = x + y * size.X;
1187             bitID[1] = (x + 1) + y * size.X;
1188             bitID[2] = (x - 1) + y * size.X;
1189             bitID[3] = x + (y + 1) * size.X;
1190             bitID[4] = x + (y - 1) * size.X;
1191
1192             //Make a list of surrounding bits, considering map edges
1193             List<float> pixelColor = new List<float>();
1194
1195             pixelColor.Add(bits[bitID[0]].ToVector4().X);
1196
1197             if (bitID[1] > 0 && bitID[1] < bits.Length)
1198                 pixelColor.Add(bits[bitID[1]].ToVector4().X);
1199             if (bitID[2] > 0 && bitID[2] < bits.Length)
1200                 pixelColor.Add(bits[bitID[2]].ToVector4().X);
1201             if (bitID[3] > 0 && bitID[3] < bits.Length)
1202                 pixelColor.Add(bits[bitID[3]].ToVector4().X);
1203             if (bitID[4] > 0 && bitID[4] < bits.Length)
1204                 pixelColor.Add(bits[bitID[4]].ToVector4().X);

```

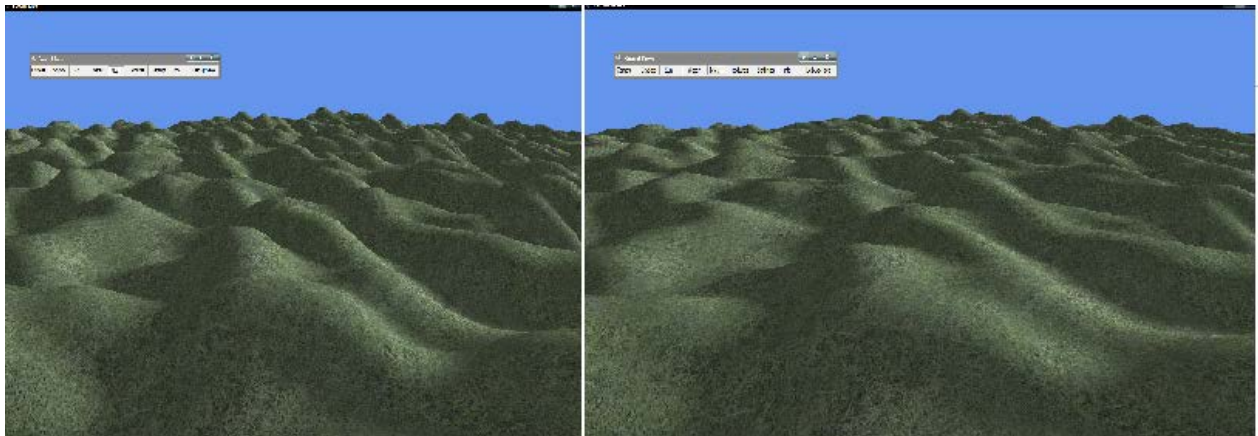
**Κώδικας 58.** Εξομάλυνση κορυφών.

Στις γραμμές 1181 έως 1190 ο αλγόριθμος έχει ως εξής: Θα πρέπει να γνωρίζουμε τα “ύψη” της γειτονιάς του κάθε τριγώνου ώστε να αποφασίσουμε ποιο ύψος θα έχει το τρέχον. Ως γειτονιά τριγώνου εννοούνται τα διπλανά τρίγωνα όπως φαίνεται στην παρακάτω εικόνα:

A	B	C
D	E	F
G	H	I

**Εικόνα 83.** Γειτονιά 3X3

Με άλλα λόγια, όταν έρθει η σειρά του τριγώνου “E” έτσι ώστε να αποφασίσουμε για το ύψος του, θα λάβουμε υπόψη τα τρίγωνα “A”, “B”, “C”, “D”, “F”, “G”, “H” και “I”. Στις επόμενες γραμμές του κώδικα αυτό που έχουμε να κάνουμε είναι να αθροίσουμε αυτά τα ύψη και να τα διαιρέσουμε με το πλήθος τους που είναι το 9. Έτσι πολύ απλά, με ένα βρόγχο ο οποίος θα τρέξει για όλα τα τρίγωνα του τοπίου μας, θα έχουμε σαν σύνολο ένα εξομαλυμένο τοπίο. Το αποτέλεσμα φαίνεται στην παρακάτω εικόνα:



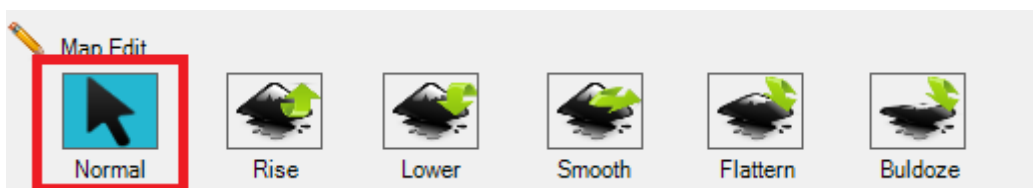
**Εικόνα 84.** Αριστερά το τοπίο πριν την εξομάλυνση. Δεξιά το τοπίο μετά.

Το επόμενο θέμα με το οποίο θα ασχοληθούμε είναι η επεξεργασία του τοπίου. Στην εφαρμογή δίνεται η δυνατότητα στον χρήστη να επεξεργαστεί το τοπίο με το ποντίκι, δίνοντάς του αρκετά εργαλεία. Συγκεκριμένα μπορούμε:



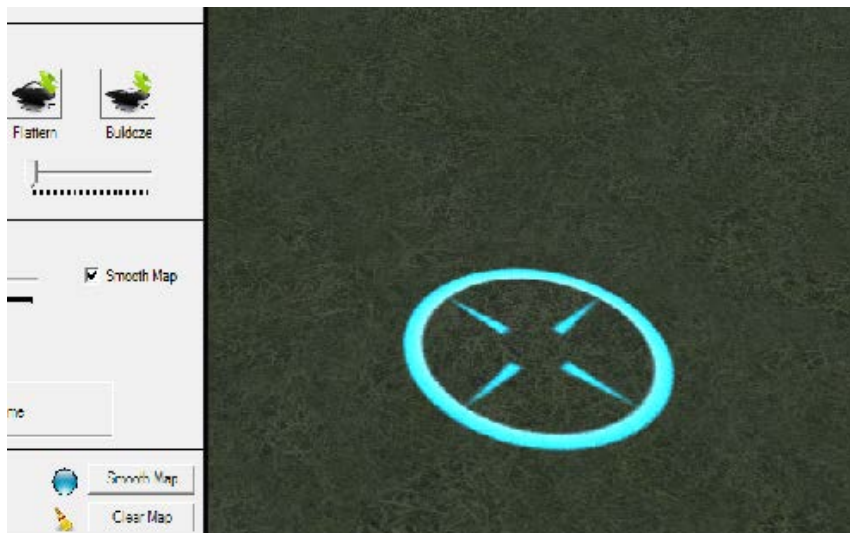
- Να ανυψώσουμε κορυφές.
- Να τις χαμηλώσουμε.
- Να τις εξομαλύνουμε.
- Να τις χαμηλώσουμε στο σημείο “0”, τέρμα κάτω.
- Να επιλέξουμε μία κορυφή και να δώσουμε στις πλαϊνές το ίδιο ύψος.

Σε κάθε ανανέωση, λοιπόν, της κάρτας γραφικών ελέγχουμε εάν έχει αλλάξει το εργαλείο το οποίο επεξεργάζεται το τοπίο και αν αυτό είναι διάφορο από το προεπιλεγμένο. Με άλλα λόγια το προεπιλεγμένο εργαλείο είναι αυτό που φαίνεται στην παρακάτω εικόνα:



**Εικόνα 85.** Το προεπιλεγμένο εργαλείο. Καμία ενέργεια δεν θα γίνει.

Εάν αυτό αλλάξει λοιπόν, όπως είπαμε και πριν, η μηχανή γραφικών θα μπει σε κατάσταση edit και θα εμφανιστεί ο κέρσορας εντοπισμού θέσης που φαίνεται στην επόμενη εικόνα:



**Εικόνα 86.** Ο κέρσορας εντοπισμού θέσης.

Όταν λοιπόν κάνουμε ένα κλικ σε αυτήν την κατάσταση που βρισκόμαστε, η κορυφή που έχουμε επιλεγμένη θα κάνει αυτήν την ενέργεια που εμείς έχουμε επιλέξει από το μενού. Για αυτό τον λόγο τρέχει μία συνάρτηση η οποία συναντάται στην κλάση **MathExtra** με το όνομα **CheckIntersection**.

Αυτή η συνάρτηση λαμβάνει αρχικά την κατάσταση του ποντικιού, μεταφράζει την δυσδιάστατη οθόνη σε τρισδιάστατη για χάρη του τοπίου και αυτό γίνεται βέβαια με την βοήθεια της κάμερας, οπότε τελικά κατασκευάζουμε την ακτίνα του ποντικιού, όπως είδαμε σε προηγούμενο κεφάλαιο, ώστε να καταλάβουμε με ποιο BoundingBox αλληλεπιδρά η ακτίνα του ποντικιού. Οπότε έχουμε πλέον μία ξεκάθαρη εικόνα για το ποια τρίγωνα πλέον θα μετακινήσουμε. Όταν βεβαίως αφήσουμε το κλικ του ποντικιού φεύγουμε από την κατάσταση επεξεργασίας και γίνονται κάποιες επιπλέον ενέργειες όπως αυτή της μερικής εξισορρόπησης τριγώνων ώστε να μην έχουμε απότομες κορυφές.

Τέλος, κάποια άλλα σημαντικά εργαλεία που πρέπει να αναφέρουμε είναι ο επιπλέον φωτισμός στο τοπίο μας, καθώς και η προβολή του σε πλέγμα. Αυτά τα δύο πετυχαίνονται με την βοήθεια του Vertex/Pixel Shader. Το πρώτο είναι μία παρόμοια περίπτωση με αυτή του ήλιου όπου απλώς αυξάνουμε τον φωτισμό των τριγώνων και το δεύτερο είναι μία εντολή στον τρόπο εμφάνισης των τριγώνων. Η εντολή αυτή εισάγεται στον Pixel/Vertex Shader ως μία διαφορετική μέθοδο:

```
technique TransformTexture
{
    pass P0
    {
        VertexShader = compile vs_3_0 Transform();
        PixelShader = compile ps_3_0 Texture();
    }
}

technique TransformWireframe
{
    pass P0
    {
        VertexShader = compile vs_3_0 Transform();
        PixelShader = compile ps_3_0 Texture();
        FillMode = Wireframe;
    }
}
```

**Κώδικας 59.** Εναλλακτικός τρόπος εμφάνισης τριγώνων με τον Vertex/Pixel Shader.

Παρατηρούμε στον προηγούμενο κώδικα HLSL ότι αυτό που αλλάζει είναι μία γραμμή παραπάνω, η **FillMode** η οποία παίρνει την τιμή **WireFrame** έτσι ώστε ό,τι περάσει μέσα από τον V/P Shader να ζωγραφιστεί με την μέθοδο του πλέγματος.

Τέλος, θα δούμε την μέθοδος εξαγωγής του χάρτη σε εικόνα με κατάληξη .png. Αυτή η λειτουργία είναι πολύ χρήσιμη διότι αργότερα αν χρειαστεί θα μπορέσουμε να φορτώσουμε, τον χάρτη μόνο, από μία εικόνα με κατάληξη .png.

Ο κώδικας είναι ιδιαίτερα απλός:

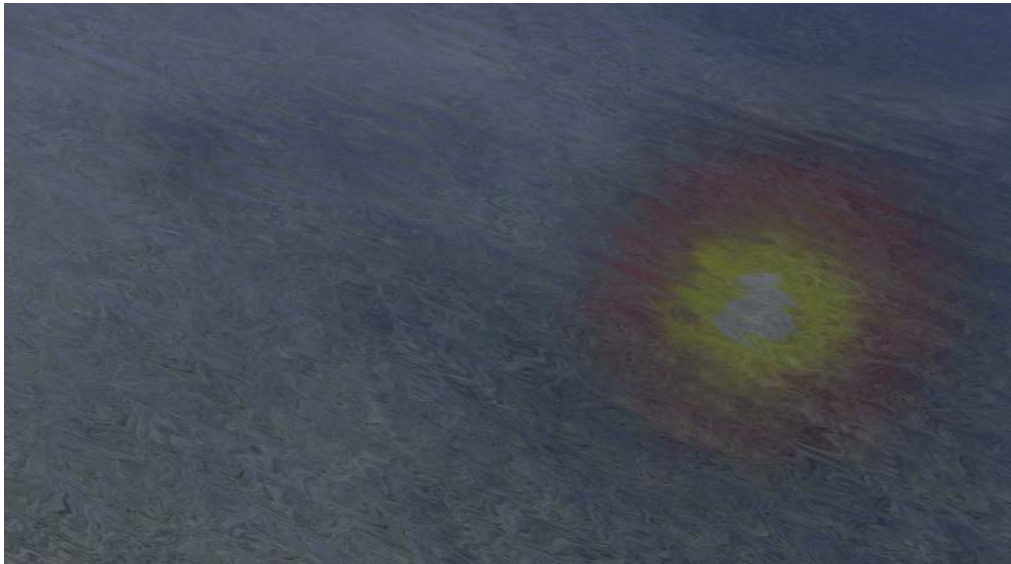
```
271  public void SaveHeightMap(string filename)
272  {
273      UpdateHeightFile();
274      FileStream stream = new FileStream(filename+".png", FileMode.Create);
275      heightmap.SaveAsPng(stream, heightmap.Width, heightmap.Height);
276      stream.Close();
277  }
```

**Κώδικας 60.** Αποθήκευση χάρτη ως εικόνα με κατάληξη .png.



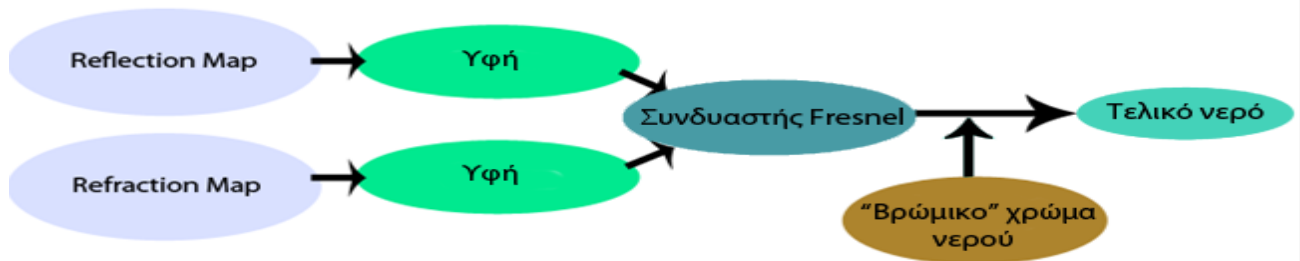
xna

Αφού έχουμε τοποθετήσει όλα τα απαραίτητα στοιχεία στο τοπίο μας είναι καιρός να αποδώσουμε λίγο παραπάνω ρεαλισμό. Αυτό θα το κάνουμε προσθέτοντας νερό στην εφαρμογή μας. Είναι μία κάπως περίπλοκη μέθοδος, αλλά όχι δύσκολη. Απλώς πρέπει να ακολουθηθούν τα βήματα πιστά. Το αποτέλεσμα τελικά αξίζει τον κόπο. Το νερό υλοποιείται μέσα στην κλάση **Water**. Γενικώς για τα τρισδιάστατα παιχνίδια υπάρχουν δύο βασικοί τρόποι με τους οποίους μπορούμε να αναπαραστήσουμε το νερό. Η μία μέθοδος είναι το νερό που μοιάζει εκείνο του ωκεανού με τα κύματα και θέλει αρκετά τρίγωνα διότι χρειάζεται να ρυθμίζουμε το ύψος κάθε φορά και η δεύτερη τεχνική είναι αυτή του νερού της λίμνης, που, όπως θα δούμε, υλοποιείται με μόνο δύο τρίγωνα. Είναι απλώς ένα επίπεδο όπου θα περνάμε συνεχώς από πάνω του ένα texture ώστε να δημιουργούνται ρεαλιστικοί κυματισμοί. Το νερό φαίνεται στην παρακάτω εικόνα:



**Εικόνα 87.** Το νερό της εφαρμογής.

Συνεχίζοντας, πρέπει να πούμε πως το νερό δεν έχει από μόνο του κάποιο χρώμα. Είναι απλά ένας έξυπνος συνδυασμός δύο πραγμάτων: Αντανάκλασης και διαφάνειας δηλαδή **Reflection** και **Refraction Map**. Το νερό αντανακλά ό,τι υπάρχει στο τοπίο μας, αλλά φέρει και κάποιο βαθμό διαφάνειας έτσι ώστε να βλέπουμε τα πράγματα από κάτω του. Στην παρακάτω εικόνα φαίνεται με ποια διαδικασία λειτουργεί το νερό, ώστε να μπορέσουμε να το κατανοήσουμε καλύτερα:



Εικόνα 88. Η διαδικασία εμφάνισης του νερού.

Όπως βλέπουμε και στο προηγούμενο σχήμα, για να κατασκευάσουμε το νερό θα πρέπει να γνωρίζουμε τον Reflection και Refraction Map για κάθε pixel, ώστε να τα εμφανίσουμε μέσα στις υφές (textures). Έτσι θα έχουμε ένα τέλειο καθρέφτη. Το νερό, όμως, ποτέ δεν είναι τέλειος καθρέφτης. Τα αντικείμενα που εμφανίζονται πάνω του αλλοιώνονται λόγω των κυματισμών του. Συνεχίζοντας θα πρέπει όταν κοιτάμε το νερό να βλέπουμε τις μερικές αντανακλάσεις των αντικειμένων πάνω του, αλλά και μέσα από αυτό. Αυτόν τον συνδυασμό θα τον επιτύχουμε με τον Fresnel Combinator. Μέχρι στιγμής έχουμε ένα νερό το οποίο φέρει κυματισμούς και είναι πεντακάθαρο μπλε. Για να το κάνουμε λίγο πιο ρεαλιστικό θα αλλοιώσουμε λίγο το χρώμα του σε κάτι που πλησιάζει πιο κοντά σε μπλε-γκρι.

Πριν προχωρήσουμε όμως θα πρέπει να ορίσουμε τα δύο τρίγωνα τα οποία θα αντιπροσωπεύουν την επιφάνεια. Αυτό γίνεται απλά με τον εξής τρόπο:

```

42 public void setUpWaterVertices()
43 {
44     VertexPositionTexture[] waterVertices = new VertexPositionTexture[6];
45
46     waterVertices[0] = new VertexPositionTexture(new Vector3(-100000, waterHeight, -100000), new Vector2(0, 1));
47     waterVertices[1] = new VertexPositionTexture(new Vector3(width, waterHeight, 0), new Vector2(0, 0));
48     waterVertices[2] = new VertexPositionTexture(new Vector3(0, waterHeight, length), new Vector2(1, 0));
49
50     waterVertices[5] = new VertexPositionTexture(new Vector3(0, waterHeight, length), new Vector2(1, 1));
51     waterVertices[3] = new VertexPositionTexture(new Vector3(width, waterHeight, 0), new Vector2(0, 1));
52     waterVertices[4] = new VertexPositionTexture(new Vector3(width, waterHeight, length), new Vector2(1, 0));
53
54     waterVertexBuffer = new VertexBuffer(Renderer.graphics.GraphicsDevice, VertexPositionTexture.VertexDeclaration);
55     waterVertexBuffer.SetData(waterVertices);
56 }

```

**Κώδικας 61.** Δημιουργία δύο τριγώνων που λειτουργούν ως επιφάνεια για το νερό.

## 16.1 - Δημιουργία Refraction Map

Αρχικά θα δούμε πώς μπορούμε να κατασκευάσουμε τον Refraction Map. Η τεχνική έχει ως εξής: Θα χρειαστεί να απεικονίσουμε κάποια κομμάτια του τοπίου μέσα σε ένα texture (το νερό). Με άλλα λόγια, χρειαζόμαστε μόνο μερικά πράγματα τα οποία βλέπει η κάμερα. Για να απεικονίσουμε αντικείμενα σε ένα texture θα χρειαστεί να χρησιμοποιήσουμε τις παρακάτω μεταβλητές:

```
10         float waterHeight = Renderer.controlForm.waterHeight.Value;
11         int width, length;
12
13         RenderTarget2D refractionRenderTarget;
14         Texture2D refractionMap;
```

**Κώδικας 62.** Χρήση μεταβλητών για απεικόνιση σε texture.

Η μεταβλητή στην γραμμή 10 χρησιμοποιείται για να ορίσουμε το ύψος που θα έχει το νερό μας. Οι γραμμές 13 και 14 είναι χρήσιμες για την απεικόνιση αντικειμένων πάνω στην υφή μας. Οι παρακάτω γραμμές κώδικας χρησιμεύουν για να ορίσουμε ποιος θα είναι ο “στόχος” απεικόνισης:

```
27         refractionRenderTarget = new RenderTarget2D
28             (Renderer.graphics.GraphicsDevice,
29             Renderer.graphics.GraphicsDevice.PresentationParameters.BackBufferWidth,
30             Renderer.graphics.GraphicsDevice.PresentationParameters.BackBufferHeight,
31             false, Renderer.graphics.GraphicsDevice.PresentationParameters.BackBufferFormat,
32             Renderer.graphics.GraphicsDevice.PresentationParameters.DepthStencilFormat);
>>
```

**Κώδικας 63.** Ορίζοντας τον “στόχο” απεικόνισης.

Στην συνέχεια, θα πρέπει να ορίσουμε ένα **clip plane** έτσι ώστε να ζωγραφίζουμε μόνο ό,τι βρίσκεται κάτω από το νερό επάνω στο νερό. Για να το κάνουμε αυτό, συνήθως χρειάζονται αρκετές μαθηματικές πράξεις. Για να το ορίσουμε, λοιπόν, θα πρέπει να ξέρουμε την κατεύθυνσή του και το κέντρο στο οποίο βρίσκεται το τοπίο. Στην περίπτωση μας αυτά τα νούμερα είναι απλό να τα βρούμε διότι η επιφάνειά μας είναι επίπεδη. Ο παρακάτω κώδικας μας βοηθά να ορίσουμε αυτήν την αποκομμένη περιοχή που θα απεικονιστεί επάνω στην επιφάνεια αργότερα:

```

188 private Plane CreatePlane(float height, Vector3 planeNormalDirection, Matrix currentViewMatrix, bool clipSide)
189 {
190     planeNormalDirection.Normalize();
191     Vector4 planeCoeffs = new Vector4(planeNormalDirection, height);
192
193     if (clipSide)
194     {
195         planeCoeffs *= -1;
196     }
197
198     Plane finalPlane = new Plane(planeCoeffs);
199     return finalPlane;
200 }

```

**Κώδικας 64.** Δημιουργία clip plane.

Έπειτα, θα πρέπει να ζωγραφίσουμε αυτή την αποκομμένη περιοχή επάνω στο επίπεδό μας.

```

101 private void DrawRefractionMap()
102 {
103     Plane refractionPlane = CreatePlane(waterHeight + 1.5f, new Vector3(0, -1, 0), Renderer.camera.view, false);
104     effect.Parameters["ClipPlane0"].SetValue(new Vector4(refractionPlane.Normal, refractionPlane.D));
105     effect.Parameters["Clipping"].SetValue(true); // Allows the geometry to be clipped for the purpose of creating a ref
106     Renderer.graphics.GraphicsDevice.SetRenderTarget(refractionRenderTarget);
107     Renderer.graphics.GraphicsDevice.Clear(ClearOptions.Target | ClearOptions.DepthBuffer, Color.CornflowerBlue, 1.0f, 0);
108
109     if (Renderer.heightmap != null && Renderer.controlForm.waterMapReflection.Checked)
110     {
111         Renderer.heightmap.Draw(Renderer.camera.view, Renderer.camera.projection);
112     }
113
114     Renderer.graphics.GraphicsDevice.SetRenderTarget(null);
115     effect.Parameters["Clipping"].SetValue(false); // Make sure you turn it back off so the whole scene doesnt keep rende
116     refractionMap = refractionRenderTarget;
117 }

```

**Κώδικας 65.** Τελικός κώδικας εμφάνισης Refraction Map.

Πλέον έχουμε τελειώσει με το πρώτο κομμάτι. Παρακάτω θα δούμε πώς κατασκευάζουμε και ζωγραφίζουμε τον Reflection Map.



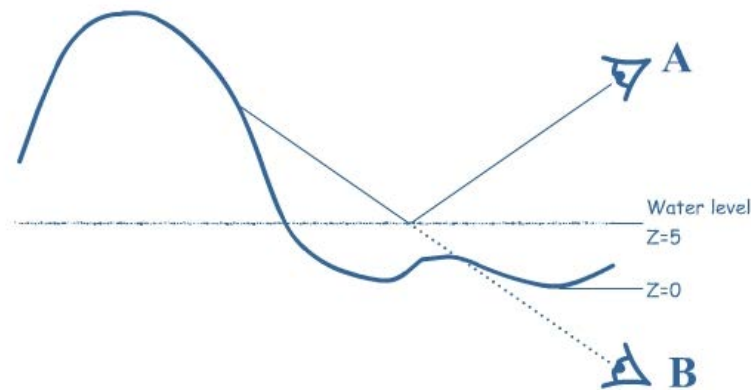
## 16.2 - Δημιουργία Reflection Map

Αρχικά, για να δημιουργήσουμε όλες τις αντανάκλασεις των αντικειμένων που βρίσκονται στον χώρο επάνω στο νερό θα χρειαστούμε τις παρακάτω μεταβλητές:

```
16 | RenderTarget2D reflectionRenderTarget;  
17 | Texture2D reflectionMap;
```

**Κώδικας 66.** Μεταβλητές απαραίτητες για τη δημιουργία του Reflection Map.

Αυτό που πρέπει να γίνει σε αυτήν την περίπτωση είναι απλό: Θα πρέπει να ξέρουμε σε ποια θέση βρίσκεται η κάμερα έτσι ώστε να ζωγραφίζουμε ό,τι υπάρχει από πάνω της επάνω στην επιφάνεια. Η παρακάτω εικόνα θα μας βοηθήσει να διαλευκάνουμε μερικά πράγματα:



**Εικόνα 89.** Η αντανάκλαση της κάμερας.

Όπως φαίνεται στην προηγούμενη εικόνα, ό,τι φαίνεται από την κάμερα A πρέπει να φαίνεται στην κάμερα B ανάποδα. Εμείς τελικά θα πρέπει να ζωγραφίσουμε ό,τι φαίνεται από την κάμερα B επάνω στην επιφάνεια που αντιπροσωπεύει το νερό. Όμως αυτό δημιουργεί ένα μικρό προβληματάκι: Στην περίπτωση της κάμερας B ένα κομμάτι του τοπίου παρεμποδίζει την οπτική γωνία. Αυτό όμως διορθώνεται με την έτοιμη συνάρτηση που κάναμε στην περίπτωση του Refraction Map, clipPlane.

Προχωρώντας για την δημιουργία πιστής αντανάκλασης θα πρέπει να φτιάξουμε τον Reflection Matrix, δηλαδή τον ανεστραμμένο πίνακα ώστε να εμφανιστούν τα αντικείμενα ανάποδα. Για αυτό η δήλωση της παρακάτω μεταβλητής είναι χρήσιμη:

```
19 | Matrix reflectionViewMatrix;
```

**Κώδικας 67.** Ο ανεστραμμένος πίνακας για εμφάνιση του ανάποδου χώρου.

Έπειτα, με την παρακάτω συνάρτηση, σε κάθε ανανέωση της μηχανής γραφικών θα βρίσκουμε τον ανάστροφη θέση και οπτικό πεδίο της κάμερας:

```
81 public void update()
82 {
83     Vector3 reflCameraPosition = Renderer.camera.position;
84     reflCameraPosition.Y = -Renderer.camera.position.Y + waterHeight * 2;
85     Vector3 reflTargetPos = Renderer.camera.target;
86     reflTargetPos.Y = -Renderer.camera.target.Y + waterHeight * 2;
87
88     Vector3 cameraRight = Vector3.Transform(new Vector3(1, 0, 0), Renderer.camera.rotationMatrix);
89     Vector3 invUpVector = Vector3.Cross(cameraRight, reflTargetPos - reflCameraPosition);
90
91     reflectionViewMatrix = Matrix.CreateLookAt(reflCameraPosition, reflTargetPos, invUpVector);
92 }
```

**Κώδικας 68.** Εύρεση ανάποδης θέσης της κάμερας και οπτικού πεδίου.

Τελικά, αφού έχουμε ολοκληρώσει όλες τις απαραίτητες διαδικασίες έχουμε φτάσει στο σημείο όπου έχουμε λάβει την θέση και το οπτικό πεδίο της κάμερας Β όπως είπαμε στην εικόνα 89. Τελικά το μόνο που μένει να κάνουμε είναι να ζωγραφίσουμε τον Reflection Map. Ο παρακάτω κώδικας κάνει ακριβώς αυτό:

```
120 private void DrawReflectionMap()
121 {
122     Plane reflectionPlane = CreatePlane(waterHeight - 0.5f, new Vector3(0, -1, 0), reflectionViewMatrix, true);
123     effect.Parameters["ClipPlane0"].SetValue(new Vector4(reflectionPlane.Normal, reflectionPlane.D));
124     effect.Parameters["Clipping"].SetValue(true); // Allows the geometry to be clipped for the purpose of creating a ref
125
126     Renderer.graphics.GraphicsDevice.SetRenderTarget(reflectionRenderTarget);
127
128     Renderer.graphics.GraphicsDevice.Clear(ClearOptions.Target | ClearOptions.DepthBuffer, Color.CornflowerBlue, 1.0f, 0);
129
130     if (Renderer.controlForm.waterSkyReflection.Checked && Renderer.controlForm.SkyboxEnable.Checked)
131     {
132         Renderer.skybox.Draw(reflectionViewMatrix, Renderer.camera.projection);
133     }
134
135     if (Renderer.heightmap != null && Renderer.controlForm.waterMapReflection.Checked)
136     {
137         Renderer.heightmap.Draw(reflectionViewMatrix, Renderer.camera.projection);
138     }
139
140     if (Renderer.controlForm.waterSunReflection.Checked && Renderer.controlForm.SunEnable.Checked)
141     {
142         Renderer.sun.Draw(reflectionViewMatrix, Renderer.camera.projection);
143     }
144
145     effect.Parameters["Clipping"].SetValue(false);
146     Renderer.graphics.GraphicsDevice.SetRenderTarget(null);
147     reflectionMap = reflectionRenderTarget;
148 }
```

**Κώδικας 69.** Απεικόνιση του Reflection Map.

Σε αυτό το σημείο, αφού έχουμε τελειώσει με τις βασικές διαδικασίες, θα δούμε λίγο κώδικα HLSL ο οποίος είναι απαραίτητος για την εμφάνιση του νερού. Με άλλα λόγια, όπως είχαμε πει για τα εφέ στην μηχανή XNA, το νερό θα πρέπει να περάσει μέσα από των κώδικα HLSL και έπειτα να ζωγραφιστεί.

Καθώς η επιφάνειά μας είναι τελείως επίπεδη δεν θα χρειαστεί να υπολογίσουμε κάποια μεταβλητή η οποία να αντιπροσωπεύει τον φωτισμό (normal). Το μόνο που έχουμε να υπολογίσουμε είναι οι συντεταγμένες των υφών όπως κάνουμε και στον παρακάτω κώδικα. Εδώ πέρα θα πρέπει να ορίσουμε και την υφή η οποία φορτώνεται από τον διαχειριστή περιεχομένου και προκαλεί του κυματισμούς στο νερό:

```
WVertexToPixel WaterVS(float4 inPos : POSITION, float2 inTex: TEXCOORD)
{
    WVertexToPixel Output = (WVertexToPixel)0;

    float4x4 preViewProjection = mul (xView, xProjection);
    float4x4 preWorldViewProjection = mul (xWorld, preViewProjection);
    float4x4 preReflectionViewProjection = mul (xReflectionView, xProjection);
    float4x4 preWorldReflectionViewProjection = mul (xWorld, preReflectionViewProjection);

    Output.Position = mul(inPos, preWorldViewProjection);
    Output.ReflectionMapSamplingPos = mul(inPos, preWorldReflectionViewProjection);
    Output.BumpMapSamplingPos = inTex/xWaveLength;

    Output.RefractonMapSamplingPos = mul(inPos, preWorldViewProjection);
    Output.Position3D = mul(inPos, xWorld);

    float3 windDir = normalize(xWindDirection);
    float3 perpDir = cross(xWindDirection, float3(0,1,0));
    float ydot = dot(inTex, xWindDirection.xz);
    float xdot = dot(inTex, perpDir.xz);
    float2 moveVector = float2(xdot, ydot);
    moveVector.y += xTime*xWindForce;
    Output.BumpMapSamplingPos = moveVector/xWaveLength;

    return Output;
}
```

**Κώδικας 70.** Υπολογισμός συντεταγμένων υφών στην γλώσσα HLSL.

Το τελευταίο που πρέπει να υπολογίσουμε είναι δύο νούμερα. Την δυσδιάστατη θέση που βρίσκεται το τρίγωνο, αλλά και την δυσδιάστατη περιοχή την οποία “βλέπει” η δεύτερη κάμερα. Αυτό επιτυγχάνεται με τον παρακάτω κώδικα HLSL:

```

WPixelToFrame WaterPS(WVertexToPixel PSIn)
{
    WPixelToFrame Output = (WPixelToFrame)0;

    float4 bumpColor = tex2D(WaterBumpMapSampler, PSIn.BumpMapSamplingPos);
    float2 perturbation = xWaveHeight*(bumpColor.rg - 0.5f)*2.0f;

    float2 ProjectedTexCoords;
    ProjectedTexCoords.x = PSIn.ReflectionMapSamplingPos.x/PSIn.ReflectionMapSamplingPos.w/2.0f + 0.5f;
    ProjectedTexCoords.y = -PSIn.ReflectionMapSamplingPos.y/PSIn.ReflectionMapSamplingPos.w/2.0f + 0.5f;
    float2 perturbedTexCoords = ProjectedTexCoords + perturbation;
    float4 reflectiveColor = tex2D(ReflectionSampler, perturbedTexCoords);

    float2 ProjectedRefrTexCoords;
    ProjectedRefrTexCoords.x = PSIn.RefractMapSamplingPos.x/PSIn.RefractMapSamplingPos.w/2.0f + 0.5f;
    ProjectedRefrTexCoords.y = -PSIn.RefractMapSamplingPos.y/PSIn.RefractMapSamplingPos.w/2.0f + 0.5f;
    float2 perturbedRefrTexCoords = ProjectedRefrTexCoords + perturbation;
    float4 refractiveColor = tex2D(RefractionSampler, perturbedRefrTexCoords);

    float3 eyeVector = normalize(xCamPos - PSIn.Position3D);

    float3 normalVector = (bumpColor.rgb-0.5f)*2.0f;

    float fresnelTerm = dot(eyeVector, normalVector);
    float4 combinedColor = lerp(reflectiveColor, refractiveColor, fresnelTerm);

    float4 dullColor = float4(0.3f, 0.3f, 0.5f, 1.0f);

    Output.Color = lerp(combinedColor, dullColor, 0.2f);

    float3 reflectionVector = -reflect(xLightDirection, normalVector);
    float specular = dot(normalize(reflectionVector), normalize(eyeVector));
    specular = pow(specular, 256);
    Output.Color.rgb += specular;

    return Output;
}

```

**Κώδικας 71.** Μετατροπή τριγώνων σε Pixel σε κώδικα HLSL.

Κεφάλαιο 17

Αποθήκευση και ανάκτηση από αρχείο

The logo for XNA (Xbox Game Studio) features the letters 'xna' in a stylized, metallic, 3D font. The 'x' is orange and has a small orange and white object resembling a stylized 'x' or a game controller component integrated into its design.



Είδαμε σε προηγούμενο κεφάλαιο πώς είναι δυνατόν να σώσουμε τον χάρτη από το τοπίο μας. Μία τέτοια ενέργεια όμως δεν μας εξασφαλίζει πλήρη επαναφορά σε μία κατάσταση στην οποία βρισκόμασταν, με τις συγκεκριμένες ρυθμίσεις που είχαμε επιλέξει. Στην κλάση **FileHandler** υλοποιείται αυτή ακριβώς η λειτουργία. Είναι δυνατόν να δουλεύουμε σε ένα τοπίο και να το επεξεργαζόμαστε και εκείνη τη στιγμή να θέλουμε να αποθηκεύσουμε όλη την παρούσα κατάσταση. Το πρόβλημα που υπήρχε ήταν πως θα έπρεπε να αποθηκεύσουμε πολλά διαφορετικά πράγματα μεταξύ τους, όπως είναι το νερό, ο ήλιος και άλλα σε ένα κοινό αρχείο. Τα στοιχεία λοιπόν όλα αποθηκεύτηκαν σε `text format`. Αυτό βέβαια κάνει την αποθήκευση κάπως ανασφαλή διότι ο καθένας μπορεί να έχει πρόσβαση στο αρχείο και να πειράξει οποιαδήποτε ρύθμιση με το χέρι. Για αυτό τον λόγο, λοιπόν, προστέθηκε στην αποθήκευση αρχείου μία δικλείδα ασφαλείας η οποία χρησιμοποιείται από πολλές εφαρμογές σήμερα. Η διαδικασία έχει ως εξής: Στην φάση αποθήκευσης του αρχείου μπορούμε να αποθηκεύσουμε κάποια στοιχεία τα οποία δεν είναι δυνατόν να αλλάξουν με έναν απλό επεξεργαστή κειμένου και να ελέγχονται κατά την φόρτωση του αρχείου από την εφαρμογή. Ένα σημαντικό στοιχείο που αλλάζει και γίνεται δυνατή η κατανόηση πρόσβασης είναι η ημερομηνία, η οποία και χρησιμοποιήθηκε σε αυτήν την εφαρμογή, ώστε κατά την αλλαγή του αρχείου από τον χρήστη να αποτρέπεται το φόρτωμα του αρχείου στην μνήμη. Θα δούμε παρακάτω πώς αυτό μπορεί να δουλέψει.

Πριν αρχίσουμε λοιπόν να γράφουμε τα δεδομένα μας στο αρχείο, παίρνουμε την ώρα το λεπτό δημιουργίας, έτσι προσθέτοντάς τα δημιουργείται ένας κωδικός ο οποίος και γράφεται στο αρχείο. Ο παρακάτω κώδικας κάνει ακριβώς αυτό:

```
15 | //Code generation
16 | int x = File.GetLastWriteTime(filename).Hour ;
17 | int y = File.GetLastWriteTime(filename).Minute;
18 | int code = x + y ;
19 | writer.WriteLine(code);
```

**Κώδικας 72.** Δημιουργία κωδικού και αποθήκευση του σε αρχείο.

Έπειτα από την αποθήκευση και των υπολοίπων στοιχείων που απαρτίζουν το τοπίο, όπως φυσικά και την θέση και την οπτική γωνία της κάμερας τελειώνει ο αλγόριθμος αποθήκευσης. Το νούμερο που γράφτηκε σε ένα παράδειγμα ας πούμε πως ήταν το “24” εάν το αρχείο αποθηκεύτηκε στις 12:12 ( $12+12=24$ ). Αυτό το νουμεράκι έχει γραφτεί μέσα στο αρχείο. Όταν θα πάμε να τρέξουμε το αρχείο θα ελέγξει το

πρόγραμμα για αυτό το νουμεράκι αν είναι ίδιο με την ημερομηνία τροποποίησης. Εάν δεν υπάρχει πρόσβαση από κειμενογράφο το αρχείο θα φορτωθεί κανονικά. Σε αντίθετη περίπτωση η ημερομηνία τροποποίησης αλλάζει και έτσι τα δύο νούμερα δεν θα συμφωνούν μεταξύ τους. Με αυτόν τον απλό και έξυπνο τρόπο καταφέρνουμε να αποτρέψουμε την πρόσβαση στο αρχείο από κειμενογράφους. Φυσικά όταν γίνει αντιληπτή εξωτερική πρόσβαση, θα μας εμφανιστεί το μήνυμα “file corrupted” και θα τερματιστεί η διαδικασία φορτώματος του αρχείου. Η διαδικασία ελέγχου παρουσιάζεται στον παρακάτω κώδικα:

```
112         //Code check
113         string timeInFile = reader.ReadLine();
114         int x = File.GetLastWriteTime(filename).Hour;
115         int y = File.GetLastWriteTime(filename).Minute;
116         int code = x + y;
117
118         if ( String.Compare(code.ToString(), timeInFile.ToString()) == 0
119         {
120             *
121             *
122             *
216         else
217         {
218             System.Windows.Forms.MessageBox.Show("File is corrupted.");
219         }
220     }
```

**Κώδικας 73.** Έλεγχος ακεραιότητας αρχείου.

Κεφάλαιο 18

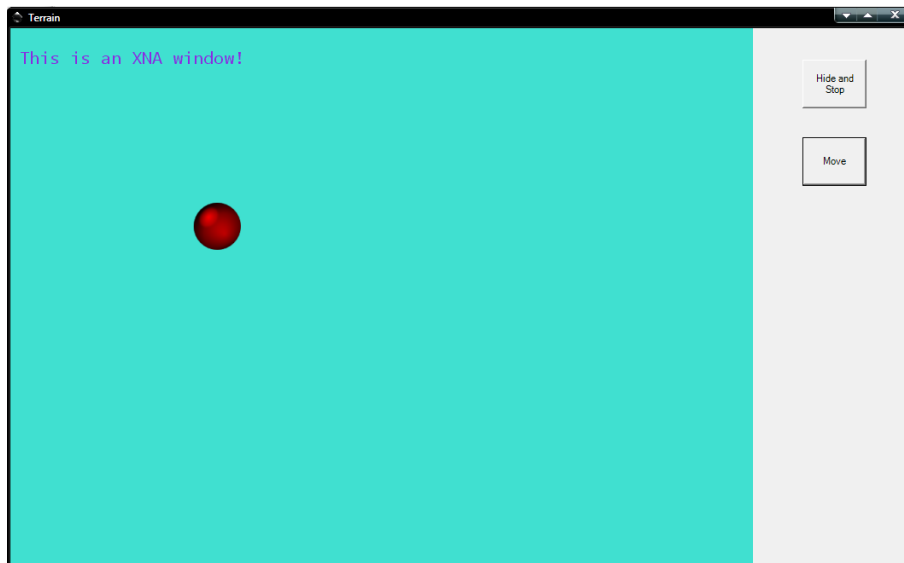
Εκδόσεις πτυχιακής/Εικόνες πτυχιακής

The logo for XNA (Xbox Game Studio) features the letters 'xna' in a stylized, lowercase, blue-grey font. The 'x' is uniquely designed with an orange and red gradient, resembling a stylized flame or a game controller's light bar.

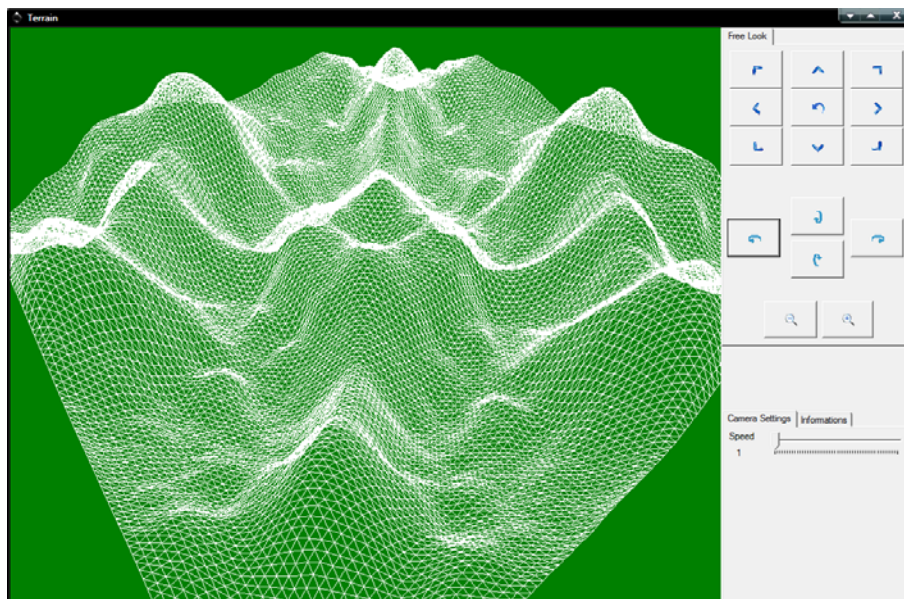




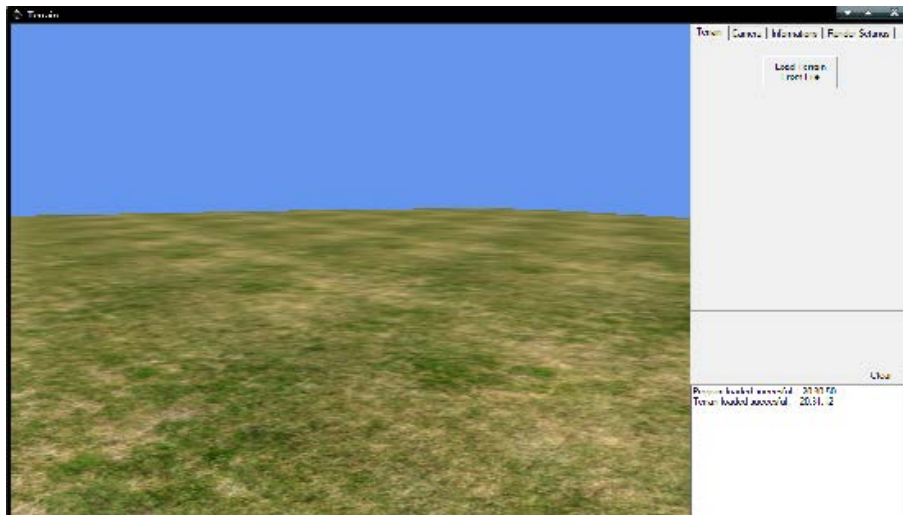
Παρακάτω παρατίθενται οι εικόνες από τις εκδόσεις της πτυχιακής:



**Εικόνα 90.** Η πρώτη έκδοση της πτυχιακής.  
Περιλαμβάνει την επιτυχή φόρτωση της μηχανής XNA μέσα σε φόρμα.



**Εικόνα 91.** Η δεύτερη έκδοση της πτυχιακής.  
Προστέθηκε χειρισμός της κάμερας και εμφάνιση απλού τοπίου.



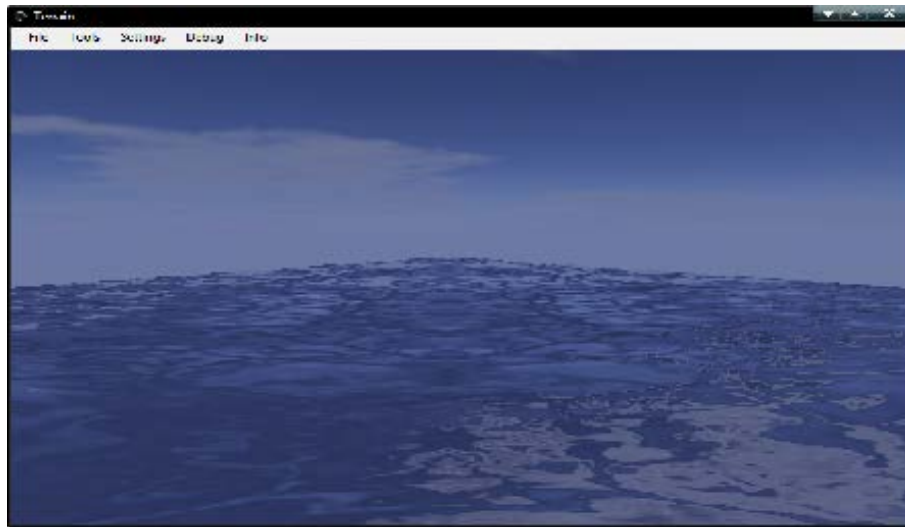
**Εικόνα 92.** Η τρίτη έκδοση της πτυχιακής.

Προστέθηκε αρχείο καταγραφής γεγονότων, φόρτωμα τοπίου από αρχείο και άλλες επιλογές.

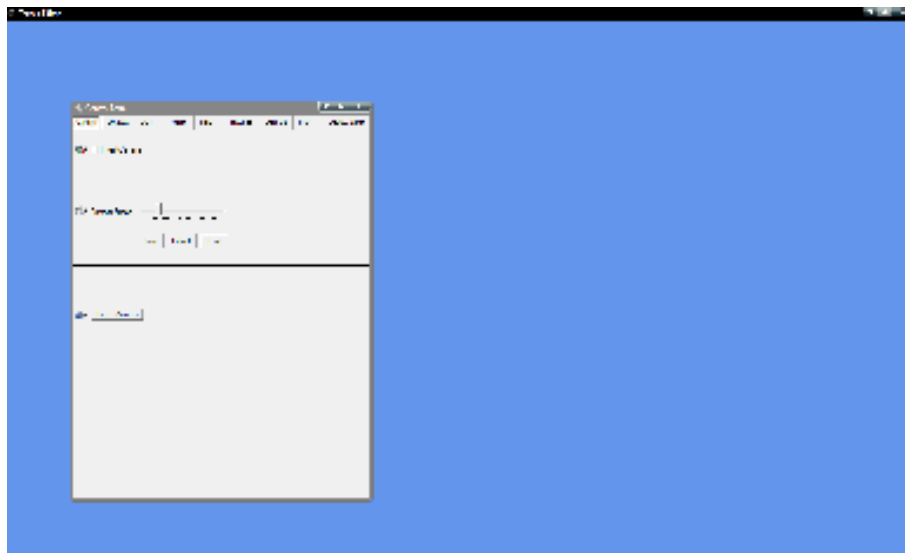


**Εικόνα 93.** Η τέταρτη έκδοση της πτυχιακής.

Άλλαξε η εργαλειοθήκη, προστέθηκε ουρανός και η κάμερα πλέον χειρίζεται με το ποντίκι.

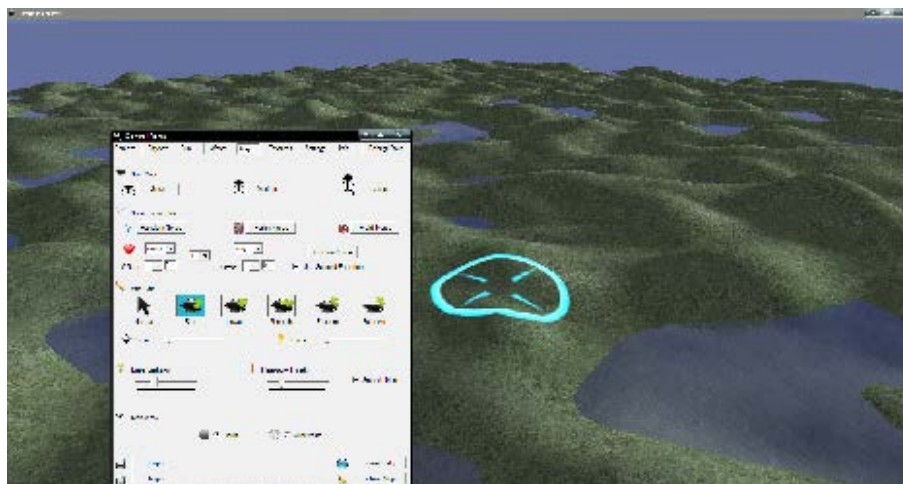


**Εικόνα 94.** Η πέμπτη έκδοση της πτυχιακής.  
Προστέθηκε νερό.

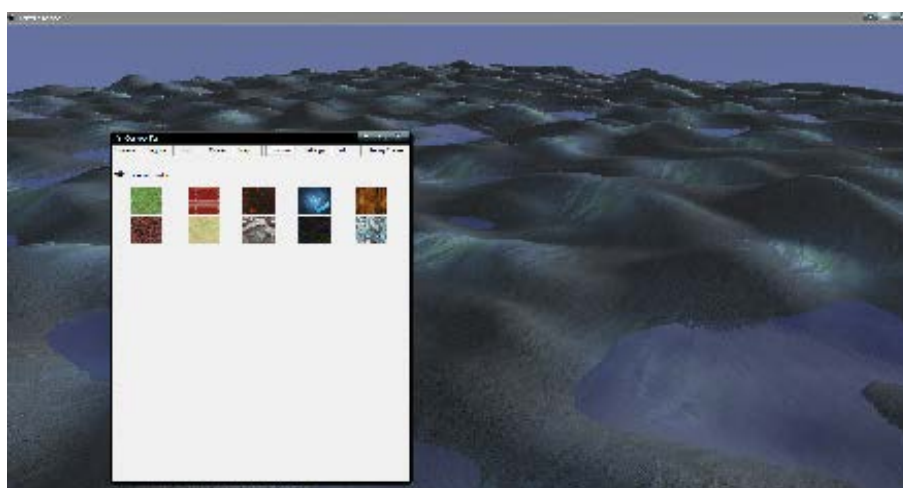


**Εικόνα 95.** Η έκτη και τελευταία έκδοση της πτυχιακής.  
Άλλαξε ο τρόπος με τον οποίο ελέγχεται το τοπίο με την φόρμα και προστέθηκαν όλα τα εργαλεία.

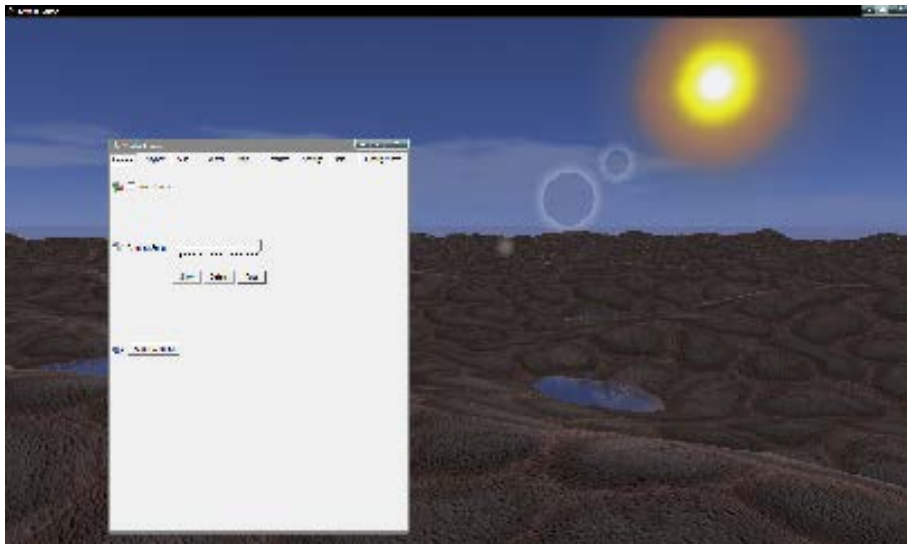
Παρακάτω παρατίθενται εικόνες μέσα από στιγμιότυπα της πτυχιακής:



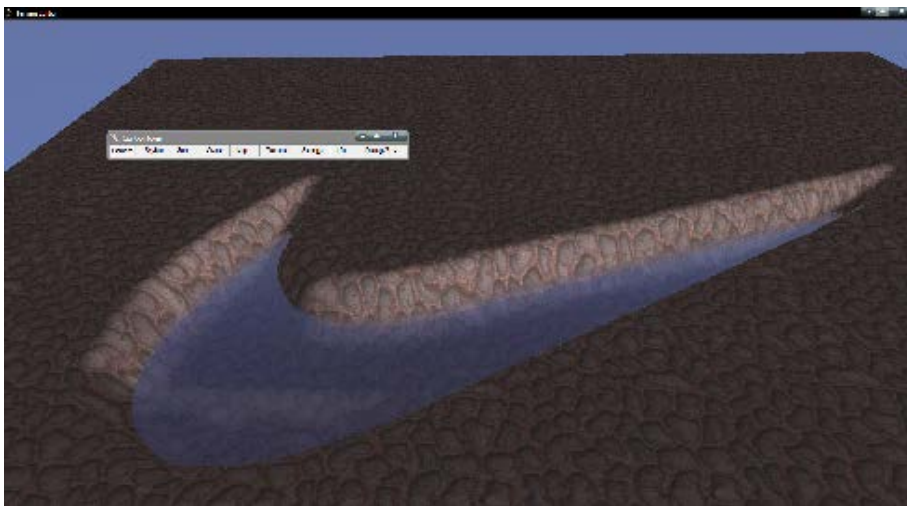
**Εικόνα 96.** Επεξεργασία του τοπίου.



**Εικόνα 97.** Αλλαγή υφών.



**Εικόνα 98.** Μία άποψη από χαμηλά.



**Εικόνα 99.** Φόρτωση μίας εικόνας χρήστη για χάρτη.



χρη



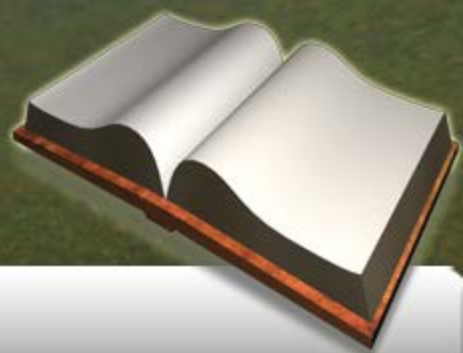
Μέσα σε αυτό το κείμενο παρουσιάστηκαν αρκετές βασικές και έξυπνες μέθοδοι οι οποίες χρησιμοποιούνται μέχρι σήμερα από τις μεγάλες εταιρίες βιντεοπαιχνιδιών. Η πτυχιακή αυτή είχε ως σκοπό να αναδείξει αυτές τις βασικές μεθόδους και να τις φέρει σε χαμηλότερο επίπεδο ώστε να γίνουν κατανοητές από όλους. Βέβαια για κάθε μία διαφορετική σκέψη υπάρχει και κάποια διαφορετική υλοποίηση. Στο τέλος βέβαια διαπιστώνουμε ότι όλα καταλήγουν στην μαθηματική σκέψη.

Προσωπικά πιστεύω πως για να προγραμματίσει κάποιος παιχνίδια, χρειάζεται να είναι εφοδιασμένος με αρκετές γνώσεις προγραμματισμού, μαθηματικών και γερά νεύρα. Τα παιχνίδια δεν είναι απλώς μία διαδικασία η οποία, αν μιλούσαμε στην γλώσσα του προγραμματισμού, έχει ένα σημείο εκκίνησης και ένα πέρας. Είναι μία επαναληπτική διαδικασία της οποίας η μελλοντικές καταστάσεις εξαρτώνται άμεσα από τις προηγούμενες και αυτό γίνεται δυσκολότερο εάν σκεφτεί κανείς ότι υπάρχει έλλειψη σχεδιαστικού περιβάλλοντος. Βέβαια για τις γενιές που θα έρθουν θα υπάρχει σίγουρα μία σχετική ευκολία, διότι οι μηχανές γραφικών όσο περνάνε τα χρόνια γίνονται ολοένα και πιο απλούστερες. Ήδη υπάρχουν περιβάλλοντα ανάπτυξης τα οποία είναι πολύ εύχρηστα και απλά όπως το 3DS Studio Max. Εδώ πρέπει να πούμε όμως ότι και αυτά τα περιβάλλοντα σε καμία περίπτωση δεν θα φτάσουν ποτέ την προσαρμοστικότητα και επεκτασιμότητα των μηχανών γραφικών σε γραμμή εντολών.

Εξερευνώντας τον κόσμο των παιχνιδιών κάποιος διαπιστώνει ότι όσα πιο πολλά μαθαίνει και γνωρίζει, τόσο πιο πολύ αυξάνεται η πολυπλοκότητα. Αλλά η έμφυτη ανάγκη για εξερεύνηση του αγνώστου, μας φέρνει πάντα απέναντι σε ωραία και φανταστικά αποτελέσματα. Ελπίζω αυτή η εργασία να γίνει σημείο εκκίνησης και πηγή έμπνευσης για άλλες εφαρμογές και δημιουργίας καινούριων μεθόδων άγνωστες μέχρι σήμερα.







xna

1. Aaron R. (2010). Learning XNA 4.0: Game Development for the PC, Xbox 360, and Windows Phone 7.
2. Kurt J. (2010). XNA 4.0 Game Development by Example: Beginner's Guide.
3. Stephen C. & Pat M. (2009). Microsoft XNA Game Studio Creator's Guide, Second Edition.
4. Tom M. & Dean J. (2010). XNA Game Studio 4.0 Programming: Developing for Windows Phone 7 and Xbox 360 (Developer's Library).
5. Rob S. M. (2011). Microsoft XNA Game Studio 4.0: Learn Programming Now!: How to program for Windows Phone 7, Xbox 360, Zune devices, and more.
6. Riemer G. (2009). XNA 3.0 Game Programming Recipes: A Problem-Solution Approach (Expert's Voice in XNA).
7. Sean J. (2011). 3D Graphics with XNA Game Studio 4.0.
8. Jonathan S. H. (2011). XNA Game Studio 4.0 for Xbox 360 Developers.
9. Alexandre L. (2009). Beginning XNA 3.0 Game Programming: From Novice to Professional (Expert's Voice in XNA).
10. Michael C. N. (2010). XNA 3D Primer.
11. Daniel S. (2010). C# Game Programming: For Serious Game Creation.
12. Riemer G. (2011). XNA Tutorials. Διαθέσιμο: [www.riemers.net](http://www.riemers.net)
13. George W. (2010). XNA Development For The Masses. Διαθέσιμο: <http://www.xnadevelopment.com>
14. Microsoft Corp. (2011). Develop For Windows Phone & XBOX 360. Διαθέσιμο: [www.create.msdn.com](http://www.create.msdn.com).
15. Microsoft Corp. (2011). Introduction to XNA 4. Διαθέσιμο: [www.msdn.microsoft.com/en-us/library/bb200104.aspx](http://www.msdn.microsoft.com/en-us/library/bb200104.aspx).
16. Bill R. (2010). Blue Rose Games. Διαθέσιμο: [www.blurosegames.com/brg/xna101.aspx](http://www.blurosegames.com/brg/xna101.aspx).
17. Kurt J. (2010). Resources for XNA Game Developers. Διαθέσιμο: [www.xnaresources.com](http://www.xnaresources.com).
18. Grand G. (2011). XNA Resources. Διαθέσιμο: [www.grandgravey.com/2009/04/15-great-xna-tutorial-sites.html](http://www.grandgravey.com/2009/04/15-great-xna-tutorial-sites.html).

