ΑΝΑΦΟΡΑ ΓΡΑΦΙΚΑ ΚΑΙ ΕΙΚΟΝΙΚΗ ΠΡΑΓΜΑΤΙΚΟΤΗΤΑ ΠΡΟΤΖΕΚΤ MINECRAFT

Όνομα	Μιχαήλ
Επώνυμο	Καΐπης
ΑΜ	1072816

Περιγραφή ιδέας

για τη δημιουργία ενός εικονικού κόσμου παρόμοιο με το Minecraft, η γενική στρατηγική είναι να χωρίσουμε τον κόσμο σε chunks, για λόγους διαχείρισης μνήμης, καθώς μπορούμε να φορτώσουμε και να ξεφορτώσουμε τα chunks τα οποία δεν είναι κοντινά στον παίχτη. Για τη δημιουργία του terrain θα χρησιμοποιήσουμε perlin noice. Για πολλαπλή χρήση πολλών διαφορετικών texture, θα χρησιμοποιήσουμε μια μεγάλη εικόνα η οποία θα περιέχει όλα τα textures, ώστε να το φορτώνουμε μια φορά και να τη χρησιμοποιούμε για όλα τα μπλοκ. Για την γρήγορη φόρτωση τον chunks θα χρησιμοποιήσουμε νήματα για επιτάχυνση της διαδικασίας. Για την τοποθέτηση και αφαίρεση μπλοκ θα χρησιμοποιήσουμε ray-casting. Το κάθε μπλοκ θα έχει το δική του προσωπικό χρόνο αφαίρεσης. Το κάθε chunk περιέχει ένα σύνολο σημείων το οποίο αντικατοπτρίζει το εξωτερικό mesh το οποίο είναι απαραίτητο να δείξουμε στη σκηνή, τα εσωτερικά τρίγωνα των chunk δεν τα χρειαζόμαστε επομένως τα αφαιρούμε. Για την προσθήκη νερού θα χρειαστούμε ένα διαφορετικό mesh το οποίο θα περιέχει τα τρίγωνα τα οποία αντιστοιχούν στην εξωτερική επιφάνεια νερού. Για εισαγωγή φωτισμού θα χρησιμοποιήσουμε phong, ενώ για ρεαλιστικά χρώματα ουρανού θα αλλάζουμε το background χρώμα ανάλογα της γωνίας που βλέπει ο ήλιος. Θα τοποθετήσουμε στο χάρτη δέντρα τα οποία είναι random generated με σταθερή δομή. σε υψηλά υψόμετρο θα χωρίσουμε το material των μπλοκ να είναι χιόνι, με χρήση ενός ακόμα στρώματος perlin noice έτσι ώστε να επιτύχουμε μια ρεαλιστική υφή στη διαφορά των υψομέτρων, το ίδιο κάνουμε και με την επιλογή πέτρας και χώματος. Η βασική ιδέα για την κατασκευή του κάθε επιμέρους κομματιού του πρότζεκτ είναι το scalability.

Μεταφορά δεδομενων

Λόγω της μεγάλης ποσότητας πληροφορίας που χρειαζόμαστε να μεταφέρουμε στην κάρτα γραφικών χρειαζόμαστε όσο το δυνατόν λιγότερο χώρο τον οποίο θα το χρησιμοποιούμε πλήρως. τα βασικά σημεία πρέπει να μεταφέρουμε στην κάρτα γραφικό είναι, συντεταγμένες x z οι οποίες κυμαίνονται μεταξύ 0 – 15, και y οπού κυμαίνεται μεταξύ 0 255, επίσης χρειαζόμαστε το ζεύγος συντεταγμένων (x,z) οι οποίες για το κάθε chunk είναι μοναδικές, χρειαζόμαστε έναν αριθμό ο οποίος θα αναπαριστά το υλικό και άλλους 2 οι οποίοι θα αναπαριστούν τα UV, τέλος χρειαζόμαστε και να αναπαραστήσουμε τα normals λόγω της απλότητας του παιχνιδιού μπορούν να έχουν μόνο έως 6 πιθανούς συνδυασμούς. για αυτό ορίζω την παρακάτω δομή δεδομένων για την αναπαράσταση όλων αυτών στοιχείων

χρειάζομαι: 4bit για το κάθε ένα από τα x,z και 8 bit από το y συνολο, το Minecraft εχει οριο για εως και +-30 εκατομμύρια blocks επομενως εως 60.000.000 / 16 = 3.750.000 chunks στον κάθε άξονα αρα χρειάζομαι log10(3.750.000) ~= 24bit για κάθε αξονα x,z αρα τελικο

συνολο εχω 8 + 4 + 4 + 24 + 24 = 64bit αρα θα χρησιμοποιήσω έναν 2 int αριθμους για αναπαρασταση ολων αυτών των χαρακτηριστικών.

Οσον αφορα τα normals, UVs, material τα αποθηκεύω σε έναν ακομα int, vertexInfo. 4bit για κάθε ένα από τα U και V, αφου εχω αποφασίσει πως η εικόνα των texture θα χωριστεί σε 16 κομμάτια, 3bit για τους 6 διαφορετικους συνδυασμούς των face normal και 8bit για την επιλογή 256 διαφορετικών υλικών.

Αρα το κάθε vertex χαρακτηρίζεται εξ ολοκλήρου από 3 int.

Για να δημιουργήσω το κάθε int με βαση τις επιμέρους μεταβλητές εχω δημιουργήσει συναρτήσεις για encoding και decoding του int τις οποιες και χρησιμοποιώ για να ορισω κάθε φορα τα verticies των blocks. Οσον αφορά την ταχύτητα που πραγματοποιούνται λογω του ότι ολες οι πράξεις γινονται με bitwise logic operators, η ταχύτητα είναι ιδιαίτερα υψηλή.

Δομή chunk

Είναι η βασική δομή στην οποία στηρίζεται όλο το πρότζεκτ. Περιέχει τις εξής βασικές συναρτήσεις

genArray: η οποία είναι υπεύθυνη για τη δημιουργία του πίνακα που περιέχει την πληροφορία για το κάθε μπλοκ του συγκεκριμένου chunk ,είναι διαστάσεων 16x16x256, η διαδικασια αυτή περιλαμβανει 3 στρωματα διαφορετικου perlin noice σε διαφορετικες σταθμες για την τοποθέτηση των διαφορετικών υλικών (χιονι , χωμα , πετρα). Γινεται χρήση για κάθε chunk ενας vector από blocks τα οποια πρεπει να συμπεριλιφθούν στον πινακα, που όμως είναι ανεξαρτητα από το αρχικο chunk, μπορει να γινεται χρηση του vector αυτου σε περιπτώσεις load, unload του chunk για αποθηκευση των αλλαγών και όχι αποθήκευση ολου του array. Γινεται και εδώ το tree generation με βαση τυχαία μεταβλητή. Ποιο συγκεκριμένα εχει οριστεί η κλάση tree με χρήση της οποιας επιστρέφεται ένα σύνολο extra blocks οπου πρέπει να τοποθετηθούν στον βασικό πινακα του chunk, αυτό δημιουργεί μερικά προβλήματα. Όταν ένα δεντρο πρεπει να δημιουργηθεί στην ακρη ενός chunk, τα blocks που πρέπει να εισαχθούν μερικά βρίσκονται εκτός του chunk, επομενως πρέπει να γινει αναφορά στο διπλανό chunk ετσι ώστε να μπουνε τα σωστα blocks στη σωστή θέση, Το μεγάλο προβλημα είναι πως τα διπλανα chunks δεν είναι παντα loaded στην ώρα τους, και επειδή δεν θέλω να φορτώνω πρώτα όλα τα chunks και μετα να γινεται το generation των δεντρων για λόγους αισθητικής, εχω ορίσει ένα global πινακα που περιέχει για το κάθε chunk ένα pointer που δείχνει σε ένα vector ο οποιος περιεχει τα extra blocks που πρεπει να εισαχθούν στο chunk. Ετσι το προβλημα λύνεται καθώς ο vector με τα εξτρα μπλοκ μπορει να υπαρχει ανεξαρτητα του εάν το chunk εχει δημιουργηθεί. Ετσι όταν δημιουργήται το κάθε chunk κοιταει το vector που του αντιστοιχει και προσθέτει τα blocks.

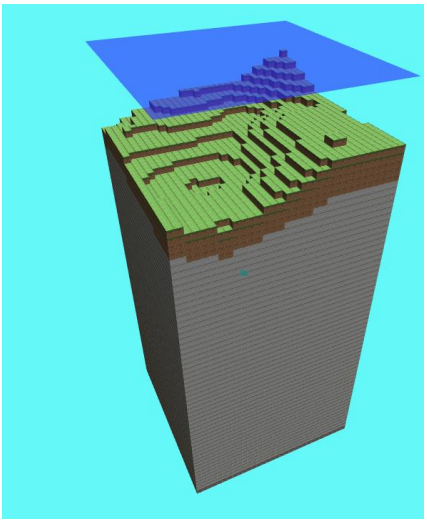
genMesh: η οποία είναι υπεύθυνη για την παραγωγή του vector των σημείων κατά την αφαίρεση των εσωτερικών μπλοκ του chunk, δηλαδη σχηματίζει το συνολο των σημείων των οποιων τα διπλανα μπλοκ είναι αερας η νερο, εάν διπλα σε καποιο μπλοκ δεν υπαρχει κανενα από τα μπλοκ αερα η νερο αυτό σημαίνει πως βρισκεται εσωερικα του chunk αρα δε θα το λαβω υποψη μου.

`brakeBlock:` αυτή η συναρτηση αφαιρει ένα block σε καποιο σημειο και κανει update το mesh, λαμβανει και υποψη και τα διπλανα chunks σε περιπτωση που υπαρχει update στα ακρα του chunk, επομενως πρεπει να ανανεωθούν και τα αντίστοιχα διπλανά chunk.

addBlock: προσθετει block και ανανεώνει το mesh

και μερικές συναρτήσεις βοηθητικές bind,bindWater,draw,drawWater.

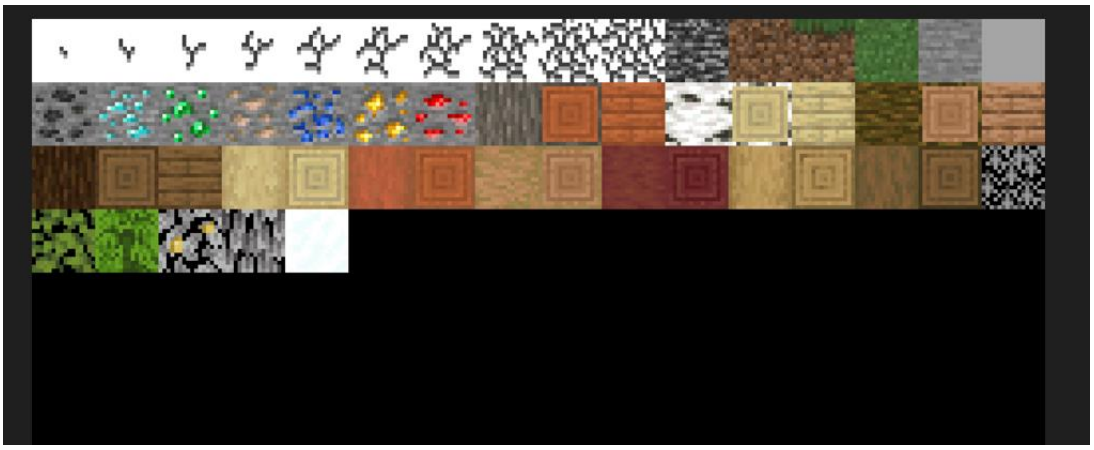
Παράδειγμα 4 διπλανων chunk



Texture Handling

Αρχικα σε έναν φακελο εχω το συνολο των textures που θελω να εντάξω σε μία εικόνα. Με χρήση python προγράμματος συλλέγω τις εικόνες τις διαχειρίζομαι και τις εντάσω σε μια μεγαλη εικονα , με βαση ένα json αρχειο το οποιο οριζει την θεση του κάθε texture πανω στον γενικο πινακα με όλα τα textures. Η χρήση του json γινεται για ευκολία επαναπαραγωγής του γενικου πινακα σε περίπτωση προσθήκης νεας texture από νέο material.
Επισης εγινε χρήση περισσοτερων python αλγορίθμων για manipulation ιδιαίτερων texture, πχ για image cutting, fixing texture headers, select photos κλπ

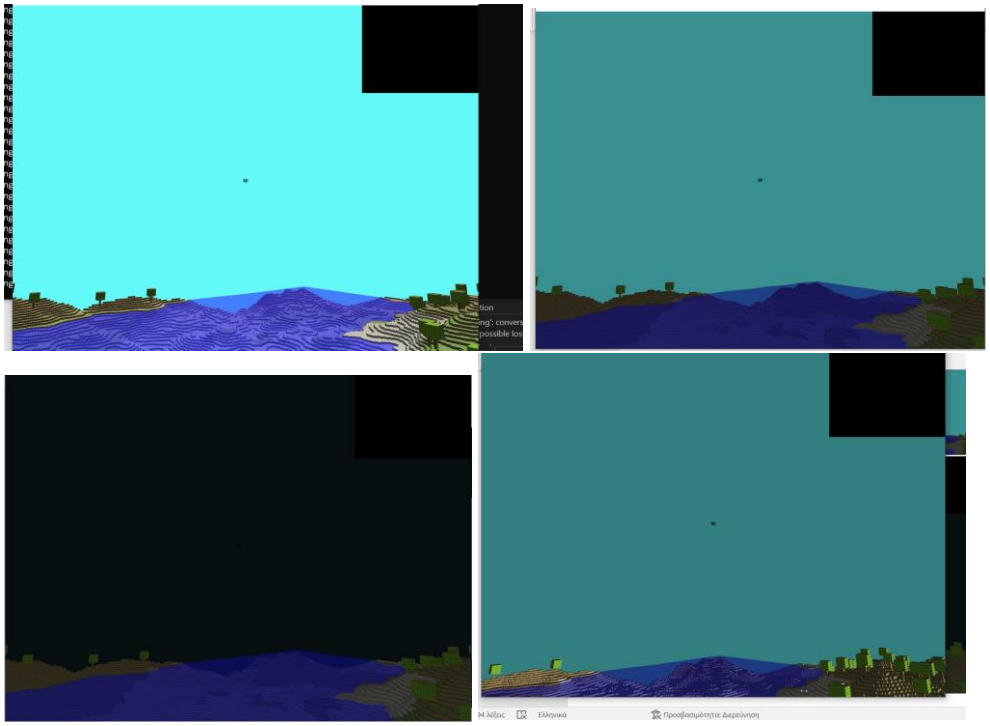
Το αποτελεσμα είναι το εξης



Με χρήση στο cpp προγραμμα μιας enum μπορω να αναφερθώ στο κάθε material και στα UV του.

Ουρανός Χρώμα

Με χρηση γραμμικών συναρτήσεων κανω το χρώμα του ουρανού να εχει διαφορετικό χρώμα αναλογα την γωνία του φωτός ως προς τον καθετο αξονα.



Ray Casting

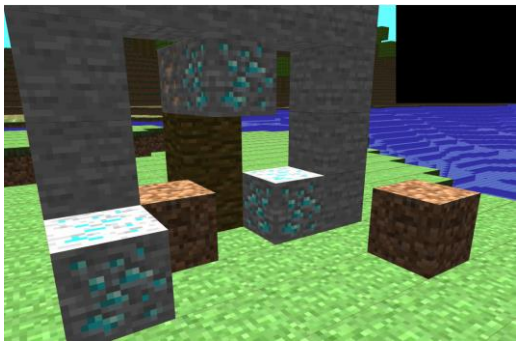
Το Ray Casting είναι ο πιο βασικός από πολλούς αλγόριθμους απόδοσης γραφικών υπολογιστών που χρησιμοποιούν τον γεωμετρικό αλγόριθμο της ανίχνευσης ακτίνων. Οι αλγόριθμοι απόδοσης βασισμένοι σε ανίχνευση ακτίνων λειτουργούν με βάση την εικόνα για να αποδώσουν τρισδιάστατες σκηνές σε δισδιάστατες εικόνες. Οι γεωμετρικές ακτίνες εντοπίζονται από το μάτι του παρατηρητή για να δειγματιστεί το φως (ακτινοβολία) που ταξιδεύει προς τον παρατηρητή από την κατεύθυνση της ακτίνας.

$$X = X_0 + t \cdot D_x$$
$$Y = Y_0 + t \cdot D_y$$
$$Z = Z_0 + t \cdot D_z$$

$$Dist = \sqrt{(D_x^2 + D_y^2 + D_z^2)}$$
$$D_x = D_x \ / \ Dist$$
$$D_y = D_y \ / \ Dist$$
$$D_z = D_z \ / \ Dist$$

Δημιουργώ ένα loop το οποιο ελεγχει για intersection με κοντινα block και εάν είναι σε ορισμενη ακτίνα από την καμερα του παικτη υπαρχει η δυνατοτητα τοποθετησης block η αφαιρεσης του , φυσικά ο παίκτης δε μπορει να επιλέξει να σπασει block που είναι αερας ή νερό,ακομα υπααρχει διαφορετικός timer για τοποθετηση και διαφορετικος για διαλυση block.

Το chunk κανει update το mesh σε κάθε αλλαγή του array. Εχει γινει και υλοποιηση των διαφορετικών ειδών blocks (inventory) με χρήση των κουμπιών 1 εως 8.




Τρόπος Αποθήκευσης Δεδομένων

Παρατηρήθηκε πως η καλύτερη στρατηγική για αποθήκευση των vertexes είναι από όλα τα chunks να εντάσσονται τα meshes του κάθε ένα σε έναν μεγαλο vector και κατευθείαν από εκει να γίνεται render στο shader. Αυτή η τεχνική εδινε στο σύστημα μου +40% ταχύτητα εκτέλεσης (FPS) παράλυτα αναγκάζει το πρόγραμμα να διατηρεί στη μνήμη 2 φορες τα δεδομένα των meshes.

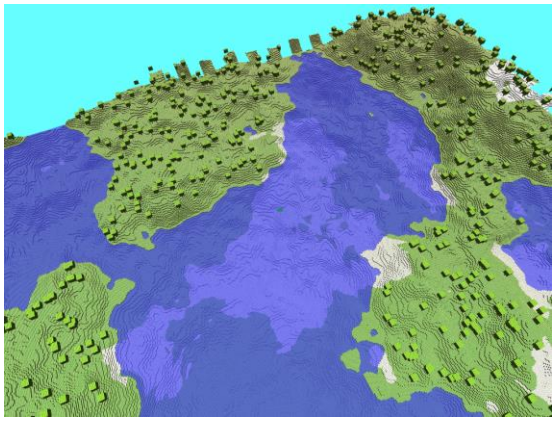
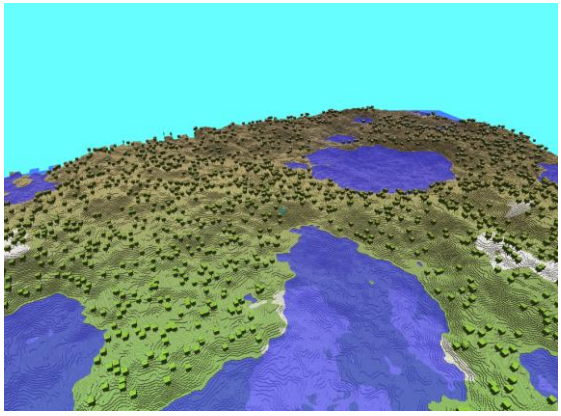
Colisions και Βαρύτητα Player

Εχει δημιουργηθεί η κλάση Player η οποια περιλαμβάνει όλα τα απαραίτητα στοιχεία για να εχουμε ένα λειτουργικό FPS χαρακτήρα. Γινεται ορισμος του inventory, πραγματοποιειται το ray casting αναλογα τα κουμπια του ποντικιου, εχουμε στην συναρτηση update ολη τη λειτουργικοτητα για εφαρμογή βαρύτητας. Η συναρτηση drawCursor απλα σχεδιαζει στο κεντρο της οθόνης ένα σχήμα τύπου cursor, η εκτύπωση του γινεται με ξεχωριστό shader.

Multithreading

Το Multithreading είναι ένα χαρακτηριστικό που επιτρέπει την ταυτόχρονη εκτέλεση δύο ή περισσότερων τμημάτων ενός προγράμματος για μέγιστη χρήση της CPU. Κάθε μέρος ενός τέτοιου προγράμματος ονομάζεται νήμα. Έτσι, τα νήματα είναι ελαφριές διαδικασίες μέσα σε μια διαδικασία.

για την επιτάχυνση της διαδικασίας του loading χρησιμοποιώ μια μέθοδο φόρτωσης των chunks παράλληλα με χρήση threading, ουσιαστικά αυτό που συμβαίνει είναι επιλέγονται τα κατάλληλα chunks (με χρήση μαθηματικού αλγοριθμου που επιστρέφει την σειρα παραγωγής των chunk σε κυκλική σειρά) και μπαίνουν σε μια ουρά για το κάθε νήμα όπου εκεί παράγεται ο βασικός πίνακας με τη συνάρτηση του constructor και αποθηκεύεται σε μια δεύτερη ουρά για παραγωγική του εξωτερικού mesh, από αυτή τη δεύτερη ουρά ένα άλλο νήμα διαχειρίζεται με τεχνική first in first out και παράγει το τελικό mesh, εν τέλει αυτά που έχουν ολοκληρωθεί μπαίνουν σε έναν πίνακα vector και στη συνέχεια γίνεται το render. Το τελικο αποτελεσμα γινεται

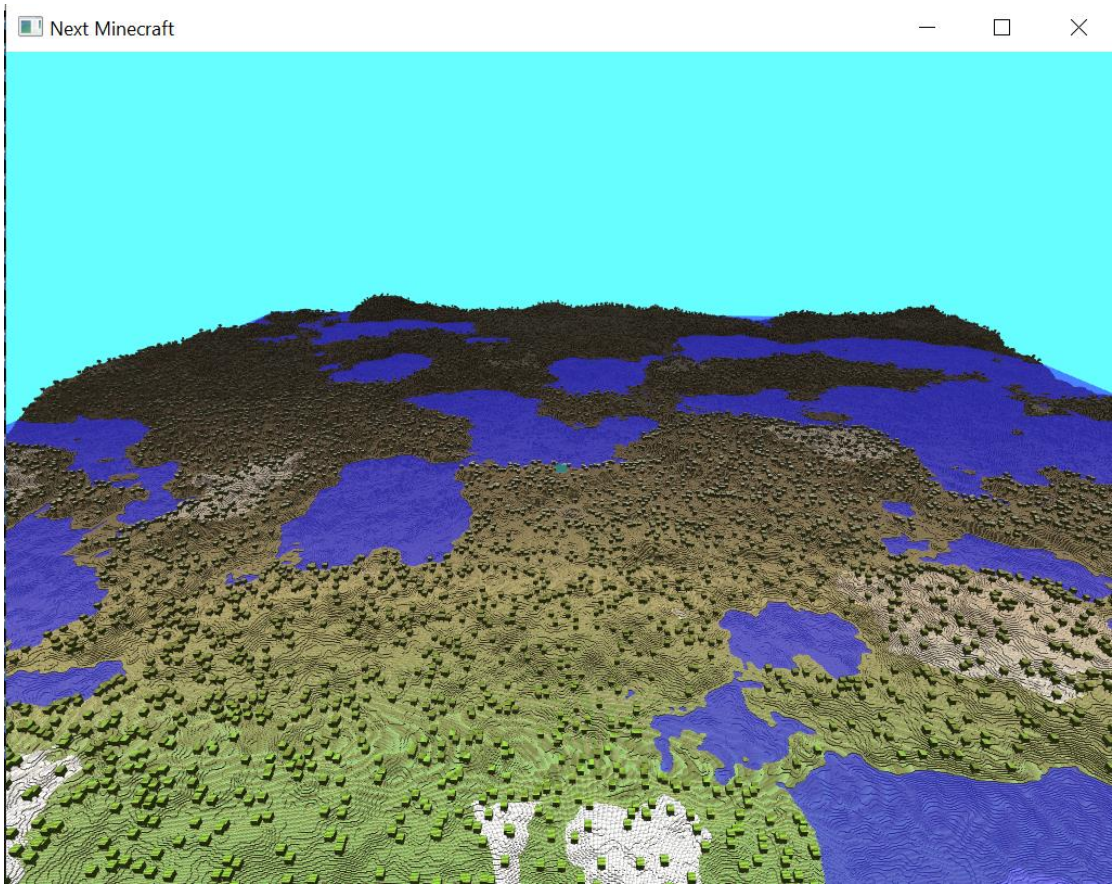
Φωτισμός

Η αντανάκλαση Phong είναι ένα εμπειρικό μοντέλο τοπικού φωτισμού. Περιγράφει τον τρόπο με τον οποίο μια επιφάνεια αντανακλά το φως ως συνδυασμός της διάχυτης ανάκλασης τραχιών επιφανειών με την κατοπτρική ανάκλαση γυαλιστερών επιφανειών. Βασίζεται στην άτυπη παρατήρηση του Phong ότι οι γυαλιστερές επιφάνειες έχουν μικρές έντονες κατοπτρικές ανταύγειες, ενώ οι θαμπές επιφάνειες έχουν μεγάλες ανταύγειες που πέφτουν πιο σταδιακά. Το μοντέλο περιλαμβάνει επίσης έναν όρο περιβάλλοντος για να ληφθεί υπόψη η μικρή ποσότητα φωτός που διαχέεται σε ολόκληρη τη σκηνή

Γινεται χρήση ambiend φωτισμού λόγω της απλότητας του περιβάλλοντος, ενώ υπολογίζονται ο depth buffer για σκίαση (στο κομμάτι αυτό ο κώδικας είναι προβληματικος δεν γράφει το depth texture επομενως η σκίαση δεν λειτουργεί).

Τελικη Δυνατότητα περιβάλλοντος

20.000 chunks ~ 2-3 FPS , 7.2GB of memory


Next Minecraft