Υλοποίηση voxel-based game με χρήση της OpenGL

Χρήστος Κατσανδρής

1072755

27/2/2023

—

Γραφικά και Εικονική Πραγματικότητα

—

Επιβλέπων: Κ. Μουστάκας

Πίνακας Περιεχομένων

Κεφάλαιο 1. Εισαγωγή

Στην εργασία αυτή, ζητούμενο είναι η κατασκευή ενός **voxel-based παιχνιδιού**, στη γλώσσα C++, με χρήση της διεπαφής OpenGL. Ένα voxel-based παιχνίδι είναι ένα παιχνίδι γραφικών, στο οποίο ο κόσμος αναπαρίσταται από στοιχειώδεις κύβους ύλης σταθερών διαστάσεων. Ο κόσμος, δηλαδή, αποτελείται από ένα τρισδιάστατο πλέγμα κύβων (voxels), καθένας από τους οποίους αντιπροσωπεύει έναν στοιχειώδη όγκο ενός αντικειμένου [1].

Ένα από τα δημοφιλέστερα voxel-based παιχνίδια στον κόσμο είναι το *Minecraft*, στο οποίο ο παίκτης εξερευνά έναν άπειρο τρισδιάστατο κόσμο, συλλέγει πρώτες ύλες και κατασκευάζει οικοδομήματα και διάφορα άλλα έργα, συχνά με την παρουσία εχθρικών ή φυσικών απειλών.

Η παρούσα εργασία είναι μία προσπάθεια επανυλοποίησης του *Minecraft*, σχεδιάζοντας τόσο το γραφικό μέρος, για την απεικόνιση του κόσμου, όσο και το υπολογιστικό μέρος, για την προσθήκη του σχεδίου δράσης του παιχνιδιού (gameplay). Για το υπολογιστικό μέρος, δηλαδή για το κομμάτι που αφορά στην επεξεργασία δεδομένων από τη CPU, χρησιμοποιήθηκε η γλώσσα C++ που φημίζεται για την ταχύτητά της, ενώ για το γραφικό μέρος, δηλαδή για το κομμάτι της GPU, χρησιμοποιήθηκε η διεπαφή (interface) OpenGL, που συνδέει τη C++ της CPU με τη GLSL της GPU [2].

Εικόνα 1. Επίδειξη του παιχνιδιού Minecraft (© thenextweb).

Η εργασία χωρίζεται σε δύο μέρη, τα οποία εξηγούνται αναλυτικά στη συνέχεια της αναφοράς. Για την υλοποίηση του παιχνιδιού, έγινε εκτενής χρήση αντικειμενοστραφούς προγραμματισμού. Σημειώνεται ότι ο πλήρης πηγαίος κώδικας της εργασίας βρίσκεται στο GitHub και διατίθεται ελεύθερα.

Κεφάλαιο 2. Πρώτο μέρος

Στο κεφάλαιο αυτό, αναλύεται το πρώτο μέρος της υλοποίησης του παιχνιδιού. Εδώ στήνεται αρχικά ο κόσμος, δημιουργούνται τα voxels που σχηματίζουν επιφάνειες και τοποθετούνται αντικείμενα πάνω σε αυτή.

2.1 Δημιουργία του πλέγματος των voxels

Κατασκευάζουμε μία κλάση `Model`[1], η οποία περιέχει όλες τις απαραίτητες ιδιότητες και μεθόδους, ώστε να περιγράφει επακριβώς ένα αντικείμενο και να επικοινωνεί με το interface της OpenGL. Η κλάση αυτή θα μας φανεί χρήσιμη στη συνέχεια, όταν θα την κληρονομήσουν άλλες κλάσεις, των οποίων οι λειτουργικότητες θα ομαδοποιηθούν.

Προς το παρόν, κατασκευάζουμε την κλάση `Voxel`[2], που κληρονομεί τη `Model`. Ορίζουμε τη γεωμετρία του voxel, καθορίζοντας τις κορυφές (vertices) του κύβου, τα κάθετα στις έδρες του διανύσματα (normals) και τις αντιστοιχίσεις στο δισδιάστατο επίπεδο των υφών (texture UVs). Ο κύβος έχει ακμή ίση με δύο μονάδες και κέντρο βάρους την αρχή του τρισδιάστατου συστήματος συντεταγμένων.

Στο κυρίως πρόγραμμα[3], δημιουργούμε ένα αντικείμενο τύπου `Voxel`, στο οποίο θα φυλάσσεται όλη η πληροφορία της γεωμετρίας του εδάφους. Αξίζει να τονιστεί ότι για εξοικονόμηση μνήμης, θα χρησιμοποιούμε πάντα ένα στιγμιότυπο για κάθε αντικείμενο του κόσμου, παρόλο που αυτό στη συνέχεια θα σχεδιάζεται περισσότερες από μία φορές. Η θέση του κάθε αντιγράφου του αντικειμένου στον κόσμο αποθηκεύεται στο διάνυσμα `Model::positions`. Με αυτόν τον τρόπο, κατασκευάζουμε το voxel grid, προσθέτοντας στο `voxelModel->positions`, τις συντεταγμένες του κέντρου βάρους κάθε κύβου του πλέγματος. Δοκιμαστικά βρίσκουμε ότι μία αποδοτική τιμή του μεγέθους για το πλέγμα είναι 100x100x100. Έτσι, τοποθετούμε τα κέντρα βάρους των κύβων στις συντεταγμένες (i, j, k), όπου τα i, j, k κυμαίνονται από -50 έως και 50.

Για τη σχεδίαση του πλέγματος, και γενικότερα των ομοειδών αντικειμένων του κόσμου, χρησιμοποιείται η τεχνική **instancing**, με την οποία σχεδιάζονται στην οθόνη πολλά ίδια αντικείμενα, με μόνο μία κλήση του interface. Είναι προφανές ότι η τεχνική αυτή προσφέρει πολύ μεγαλύτερη ταχύτητα στη σχεδίαση

---

[1] `Visualcraft/model.h, Visualcraft/model.cpp`
[2] `Visualcraft/voxel.h, Visualcraft/voxel.cpp`
[3] `Visualcraft/game.cpp`

κάθε καρέ, σε σύγκριση με το -απλούστερο μεν, πολύ αργό δε- επαναλαμβανόμενο rendering κάθε αντικειμένου [3].

Οι shaders[4] (vertex shader και fragment shader) αναλαμβάνουν τη σχεδίαση των voxels. Ο vertex shader υπολογίζει την τελική θέση κάθε voxel στον κόσμο, μετατοπίζοντας το κέντρο βάρους με το κατάλληλο διάνυσμα από το `voxelModel->positions` και εφαρμόζοντας τους γνωστούς μετασχηματισμούς. Η διαδικασία που ακολουθείται μέχρι την τελική θέση του voxel, όπως φαίνεται στην κάμερα, είναι η εξής: από το δεδομένο model space, μέσω του -μοναδιαίου- model matrix, πηγαίνουμε στο world space. Εφαρμόζουμε τη μετατόπιση από το `voxelModel->positions`. Με τον view matrix, πηγαίνουμε στο camera space. Τέλος, με τον projection matrix, πηγαίνουμε στο projection space.

Ο fragment shader επιλέγει να χρωματίσει κατάλληλα κάθε αντικείμενο, αλλά προς το παρόν παραμένουμε με ένα σταθερό μαύρο χρώμα.

2.2 Δημιουργία υψομετρικών διαφορών

Μέχρι τώρα, ο κόσμος μας δεν μοιάζει σε έναν πραγματικό κόσμο, αφού ουσιαστικά αποτελείται από έναν μεγάλο κύβο ακμής 100 x 2 = 200 μονάδων. Η επάνω έδρα αυτού του κύβου αποτελεί την επιφάνεια της γης, επιφάνειας 200x200 τετραγωνικών μονάδων. Το μειονέκτημα έως τώρα είναι ότι η επιφάνεια της γης είναι πάντα επίπεδη, πράγμα που στον πραγματικό κόσμο δεν ισχύει.

Για να διορθώσουμε το πρόβλημα αυτό, θα πρέπει να μετακινήσουμε τα voxels κάθε «στήλης» του πλέγματος κατά την ίδια ποσότητα κατά τη διεύθυνση του άξονα y, ώστε να σχηματιστούν όρη και κοιλάδες. Με άλλα λόγια, πρέπει να δημιουργήσουμε ένα **heightmap**, το οποίο να καθορίζει την υψομετρική διαφορά του κέντρου βάρους κάθε voxel από την αρχική του θέση. Η πληροφορία αυτή αποθηκεύεται στο διάνυσμα `Model::heightMap`.

Θα πρέπει ωστόσο να βρούμε έναν κατάλληλο τρόπο να ορίσουμε το πόσο θα μετακινηθεί κάθε στήλη προς τα πάνω ή προς τα κάτω. Καταλαβαίνουμε ότι το heightmap δεν πρέπει να περιέχει ασυνέχειες, δηλαδή το υψόμετρο κάθε στήλης πρέπει να διαφέρει το πολύ κατά μία μονάδα από κάθε γειτονική της, αφού μοντελοποιούμε τον αναλογικό κόσμο. Μία μέθοδος για παραγωγή ενός τέτοιου heightmap είναι ο θόρυβος Perlin[5], ο οποίος εξασφαλίζει ακριβώς τη ζητούμενη συνέχεια της γεωμετρίας [4] [5]. Αξιοποιούμε και προσαρμόζουμε τον κώδικα του

[4] `Visualcraft/Shader.vertexshader, Visualcraft/Shader.fragmentshader`
[5] `common/PerlinNoise.h, common/PerlinNoise.cpp`

[5], ώστε να μας δίνει θόρυβο ικανοποιητικού εύρους, συναρτήσει του μεγέθους του πλέγματος.

Αφού γεμίσουμε κατάλληλα το διάνυσμα `voxelModel->heightmap` με θόρυβο, θα πρέπει να προσαρμόσουμε τον vertex shader έτσι, ώστε να λαμβάνει υπόψη τη μετατόπιση αυτή. Πράγματι, κατά τη μετατόπιση των vertices σε world space, σύμφωνα με το `voxelModel->positions`, προσθέτουμε έναν επιπλέον όρο στην τεταγμένη y, που τη μετακινεί κατά το `voxelModel->heightMap`.

Πλέον, το voxel grid δεν είναι ένας συμπαγής 100x100x100 κύβος, αλλά ένα ακανόνιστο σχήμα, του οποίου η «επάνω έδρα» απεικονίζει την επιφάνεια της γης.

2.3 Εφαρμογή υφών (textures)

Σε αυτό το σημείο, θα αντικαταστήσουμε το προσωρινό μαύρο χρώμα που έχουμε δώσει στα voxels με **textures**, ώστε να μοιάζουν με υλικά του φυσικού κόσμου. Τα textures είναι ουσιαστικά εικόνες (συστοιχίες χρωμάτων), τις οποίες θα εφαρμόσουμε πάνω σε κάθε voxel. Η αντιστοίχιση γίνεται με χρήση των συντεταγμένων UV, που βρίσκονται στη μεταβλητή `Model::UVs`. Με τον τρόπο αυτό, μπορούμε να κάνουμε αρκετά περίτεχνη αντιστοίχιση κάθε vertex σε UV σημείο, ωστόσο λόγω της απλής γεωμετρίας του voxel και της φύσης του, μπορούμε απλώς να αντιστοιχίσουμε ένα ολόκληρο texture σε κάθε μία έδρα του κύβου.

Υπάρχουν ελεύθερα διαθέσιμα αρκετά textures που έχουν σχεδιαστεί για το *Minecraft*, όπως αυτά που αντλούμε από το [6]. Θα χρησιμοποιήσουμε για αρχή ορισμένα από αυτά, ώστε να κατασκευάσουμε μπλοκ γρασιδιού, ρίζας γρασιδιού, άμμου και νερού.

Μία πρώτη και αρκετά μη αποδοτική προσέγγιση είναι να δημιουργήσουμε έναν sampler στον fragment shader για κάθε texture και να χρησιμοποιούμε τον κατάλληλο ανάλογα με το ποιο texture ζητείται. Ωστόσο, αυτό οδηγεί σε μεγάλες καθυστερήσεις και σπατάλη υπολογιστικής ισχύος, λόγω του bottleneck που εισάγει το bus μεταξύ CPU-GPU. Όταν στέλνουμε ξεχωριστά κάθε texture στη GPU, η καθυστέρηση αυτή πολλαπλασιάζεται επί το πλήθος των textures. Αντίθετα, συγχωνεύοντας όλα τα textures σε μία ενιαία εικόνα (ένα **texture atlas**), μπορούμε να ελαχιστοποιήσουμε το bottleneck στη μία φορά. Με αυτόν τον τρόπο, μπορούμε να επιλέξουμε το ζητούμενο texture, χρησιμοποιώντας μόνο έναν sampler, και κοιτάζοντας μόνο συγκεκριμένα UV σημεία της εικόνας, αντί για ολόκληρη.

Την επιλογή και την εφαρμογή του κατάλληλου texture για κάθε έδρα των voxels αναλαμβάνει ο fragment shader. Ο αλγόριθμος με τον οποίο επιλέγεται το υλικό του voxel είναι ο ακόλουθος:

- Αν το voxel βρίσκεται κάτω από την επιφάνεια της θάλασσας, τότε επιλέγουμε άμμο με νερό.
- Αν το voxel βρίσκεται ένα επίπεδο πάνω από την επιφάνεια της θάλασσας, τότε επιλέγουμε άμμο.
- Αν το voxel είναι το κορυφαίο της «στήλης» του grid, τότε επιλέγουμε γρασίδι.
- Σε κάθε άλλη περίπτωση, επιλέγουμε ρίζα γρασιδιού (θα βρίσκεται μεταξύ άμμου και γρασιδιού).

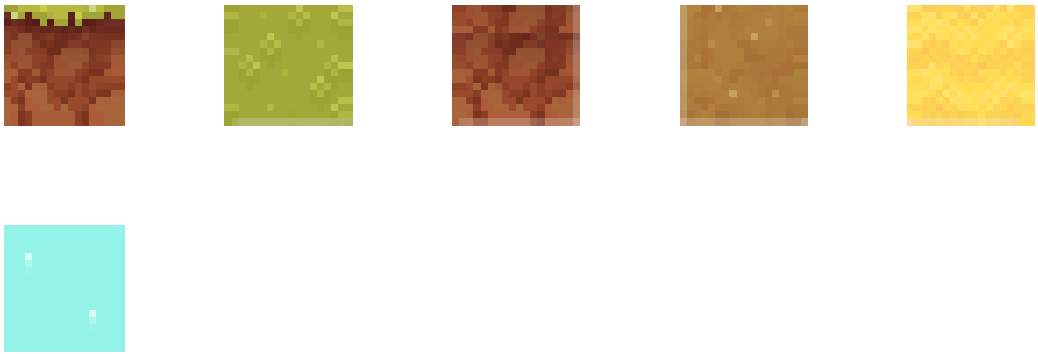
Η επιλογή αυτή πραγματοποιείται με σύγκριση της τεταγμένης y του vertex σε world space με κάποια προκαθορισμένα επίπεδα για τη στάθμη της θάλασσας και με το heightmap για τον έλεγχο του κορυφαίου μπλοκ.

Σε δεύτερο στάδιο, και αν το voxel έχει διαφορετικό texture στην επάνω έδρα και στις υπόλοιπες, πρέπει να ελεγχθεί αν το vertex ανήκει στην επάνω έδρα του voxel ή όχι. Για παράδειγμα, το μπλοκ γρασιδιού αποτελείται από δύο textures: το `grassBlockSide` και το `grassBlockTop`. Για να κάνουμε αυτόν τον έλεγχο, συγκρίνουμε την τεταγμένη y του vertex σε model space, με ένα προκαθορισμένο επίπεδο, πάνω από το οποίο θεωρείται ότι το vertex ανήκει στην επάνω έδρα.

Πλέον, ο κόσμος αποτελείται από διαφορετικού υλικού voxels ανάλογα με το υψόμετρο.

2.4 Απλοποίηση πλέγματος

Μέχρι τώρα, ο κόσμος αποτελείται από ένα πλέγμα 100x100x100 voxels, από τα οποία, το κορυφαίο κάθε στήλης σχηματίζει την επιφάνεια της γης.



Εικόνα 2. Το texture atlas μέχρι τώρα.

Μπορούμε να παρατηρήσουμε ότι, ένας παίκτης δε θα έβλεπε ποτέ τα voxels που βρίσκονται κάτω από τα κορυφαία, επομένως είναι σπατάλη να τα σχεδιάζουμε σε κάθε καρέ[6].

Θα μπορούσαμε να εξοικονομήσουμε υπολογιστική ισχύ, τροποποιώντας το πλέγμα ώστε να περιλαμβάνει μόνο τα κορυφαία μπλοκ κάθε στήλης, δηλαδή να έχουμε πλέγμα 100x1x100. Αυτό επιτρέπει επίσης τη διεύρυνση του κόσμου, με ένα πλέγμα 1000x1x1000.

2.5 Εισαγωγή αντικειμένων

Το επόμενο βήμα είναι να προσθέσουμε στον κόσμο αντικείμενα, όπως δέντρα, πέτρες κ.λπ. Μπορούμε να θεωρήσουμε κάθε αντικείμενο σαν ένα ενιαίο σύνολο από voxels, των οποίων το κέντρο βάρους θα είναι μετατοπισμένο σε κάθε στοιχειώδη όγκο του αντικειμένου.

Κατασκευάζουμε μία κλάση `Object`[7], που κληρονομεί τη `Model`. Αυτή περιέχει επιπλέον ένα διάνυσμα με τα κέντρα βάρους των voxels του πλέγματος του αντικειμένου.

Ας αναλύσουμε τη διαδικασία κατασκευής του δέντρου. Μπορούμε να βρούμε ελεύθερα διαθέσιμα μοντέλα δέντρων σε μορφή Wavefront (.obj) και να τα «κυβοποιήσουμε» (voxelize). Ο αλγόριθμος είναι ο ακόλουθος:

- Ορίζουμε ένα πλέγμα XxYxZ voxels για το αντικείμενο.
- Για κάθε στοιχείο του πλέγματος, μετράμε τα vertices του μοντέλου που βρίσκονται εκεί.
- Αν το στοιχείο περιέχει πάνω από ένα προκαθορισμένο κατώφλι vertices, προσθέτουμε ένα voxel στο συγκεκριμένο σημείο, αλλιώς όχι.

Με τον τρόπο αυτό, ένα μοντέλο δέντρου πριν και μετά το voxelization φαίνεται στην Εικόνα 3.

Για την απεικόνιση του δέντρου, θα χρειαστούμε κάποια επιπλέον textures. Επιλέγουμε textures φύλλων και κορμού ακακίας, τα οποία προσθέτουμε στο texture atlas και κάνουμε την αντιστοίχιση στον fragment shader, συγκρίνοντας την τεταγμένη y σε model space με ένα προκαθορισμένο κατώφλι.

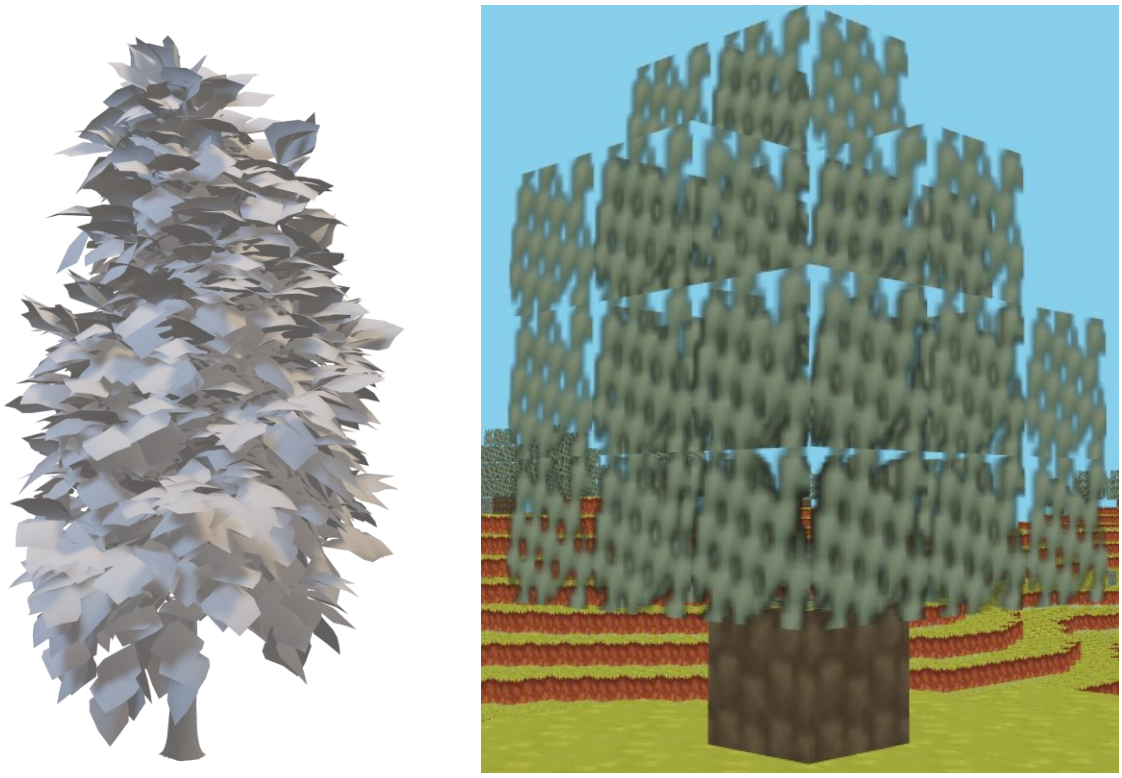
Ομοίως, μπορούμε να κατασκευάσουμε και άλλα αντικείμενα, όπως πέτρες. Θα χρειαστεί να προσαρμόσουμε το κατώφλι στο voxelization. Στην Εικόνα 4,

---

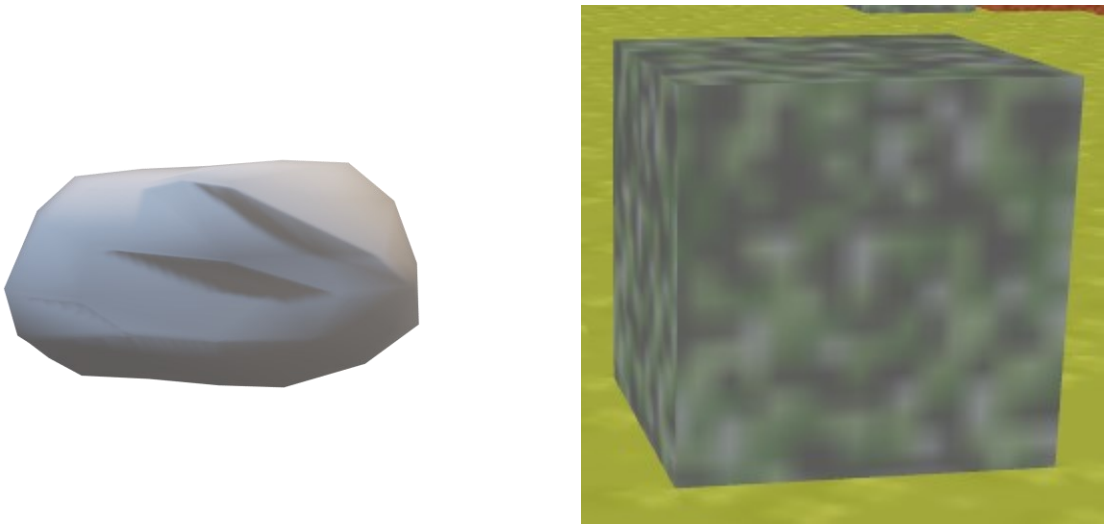
[6] Όπως αποδείχθηκε στη συνέχεια (δεύτερο μέρος), ο συλλογισμός αυτός είναι λανθασμένος και όταν ο χρήστης καταστρέφει ένα μπλοκ, οδηγεί σε σφάλματα. Περισσότερα στο 3.3 Gameplay: καταστροφή μπλοκ.

[7] `Visualcraft/object.h, Visualcraft/object.cpp`

φαίνεται το μοντέλο πριν και μετά το voxelization. Εδώ πλέον είναι εμφανής η ανάγκη για αύξηση της ανάλυσης του voxelized μοντέλου, ωστόσο κάτι τέτοιο δεν πραγματοποιήθηκε λόγω χρόνου (βλ. Κεφάλαιο 4. Σφάλματα – Χώρος για βελτίωση).



Εικόνα 3. Ένα μοντέλο δέντρου πριν και μετά το voxelization.



Εικόνα 4. Ένα μοντέλο πέτρας πριν και μετά το voxelization.

2.6 Κάμερα και πλοήγηση

Η προεπιλεγμένη πλοήγηση στον κόσμο γίνεται με απλή μετακίνηση της κάμερας, είτε κατά την κατεύθυνση των αξόνων x, y και z, είτε κατά την κατεύθυνση όρασης της κάμερας και του κάθετου διανύσματος σε αυτή. Με τη μέθοδο αυτή, η κάμερα πρακτικά ίπταται στον κόσμο χωρίς κανέναν περιορισμό, με την έννοια ότι μπορεί να μπαίνει μέσα σε μπλοκ ή να πηγαίνει κάτω από την επιφάνεια της γης.

Κάτι τέτοιο δεν είναι πολύ πρακτικό για το παιχνίδι μας. Θα πρέπει να χειριστούμε την κάμερα έτσι, ώστε να «περπατάει» πάνω στην επιφάνεια της γης και να πηδάει με το πάτημα του πλήκτρου space. Επίσης, θα πρέπει να μπλοκάρει όταν μπροστά της βρίσκεται κάποιο εμπόδιο, ή να μην μπορεί να πηδήξει όταν βρίσκεται κάτω από κάποιο εμπόδιο (π.χ. δέντρο).

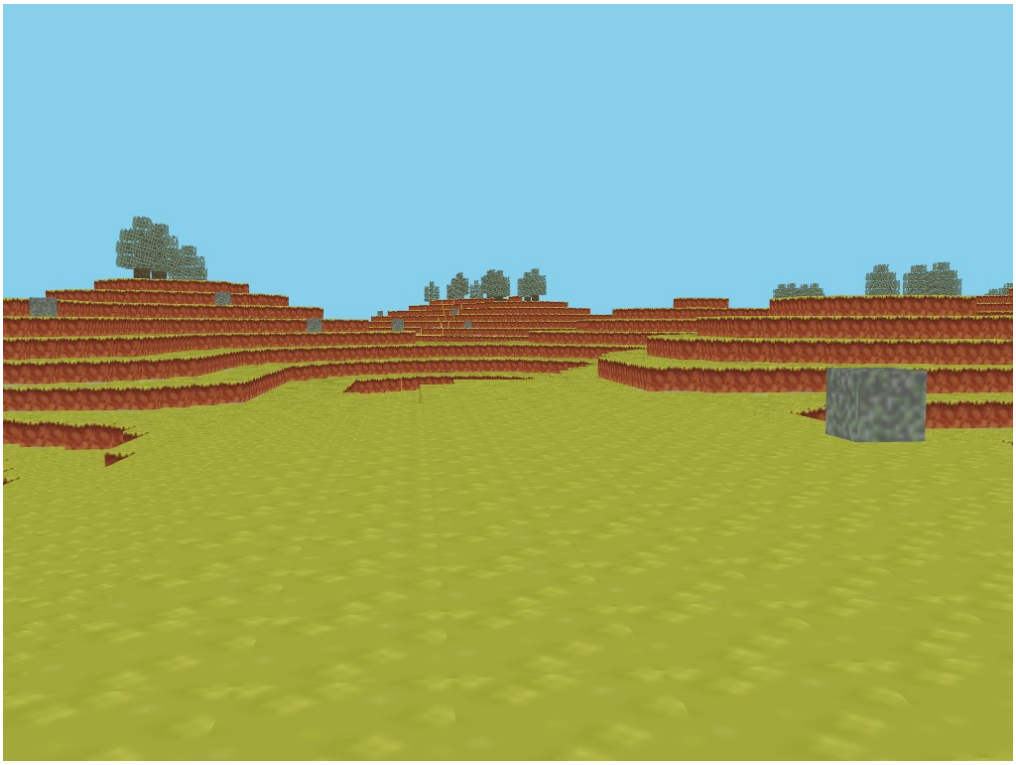
Για να γίνει αυτό, θα πρέπει πρώτα να εντοπίζουμε το μπλοκ πάνω στο οποίο βρίσκεται η κάμερα αυτή τη στιγμή και το ύψος του από το heightmap. Ταυτόχρονα, πρέπει να εντοπίζουμε τη γειτονική «στήλη» προς την οποία κοιτάζει η κάμερα και το ύψος της από το heightmap. Οι διαδικασίες αυτές γίνονται στο κυρίως πρόγραμμα. Στην κλάση `Camera`[8] και συγκεκριμένα στη μέθοδο `Camera::update`, γίνεται η σύγκριση της τρέχουσας θέσης της κάμερας σε world space, η μετακίνησή της προς την κατεύθυνση που ζητηθεί από το πληκτρολόγιο -αν επιτρέπεται-, η ανύψωσή της με space -αν επιτρέπεται- και η πτώση της αν μεταβεί σε ένα voxel σε χαμηλότερο υψόμετρο.

2.7 Σύνοψη πρώτου μέρους

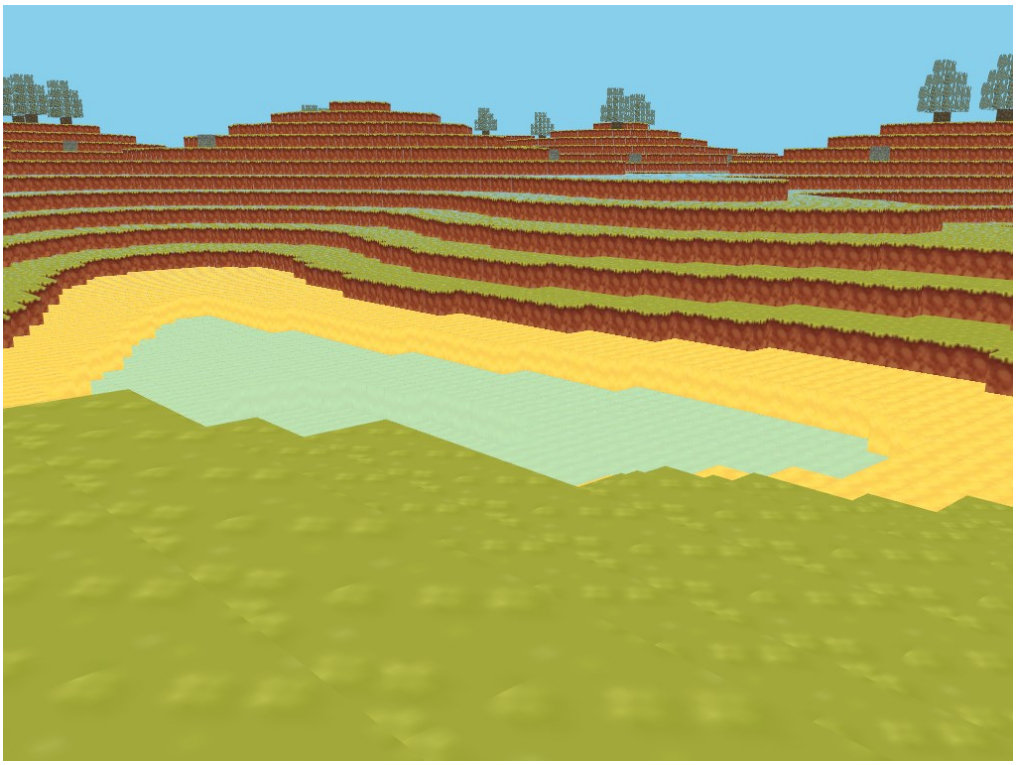
Με την ολοκλήρωση των παραπάνω διαδικασιών, έχουμε κατασκευάσει το βασικό πλέγμα του τρισδιάστατου κόσμου, έχουμε προσθέσει αντικείμενα και μπορούμε να πλοηγούμαστε οπουδήποτε (εντός των ορίων του κόσμου) με τον τρόπο που κινείται ένας άνθρωπος.

Παρακάτω φαίνονται κάποια ενδεικτικά στιγμιότυπα από το παιχνίδι, όπως αυτό έχει διαμορφωθεί μέχρι τώρα.

[8] `common/camera.h, common/camera.cpp`



Εικόνα 5. Ο κόσμος με δέντρα και πέτρες.



Εικόνα 6. Ο κόσμος με άμμο και νερό.

Κεφάλαιο 3. Δεύτερο μέρος

Στο κεφάλαιο αυτό, αναλύεται το δεύτερο μέρος της υλοποίησης του παιχνιδιού. Εδώ αναπτύσσονται θέματα σχετικά με φωτισμό και σκίαση του κόσμου, αλλά και θέματα που αφορούν το σχέδιο δράσης (gameplay).

3.1 Φωτισμός

Μέχρι τώρα, τα αντικείμενα του κόσμου σχεδιάζονται με το πραγματικό τους χρώμα, όπως αυτό καθορίζεται από το κάθε texture. Στην πραγματικότητα όμως, ένας παρατηρητής δεν βλέπει ποτέ τα αντικείμενα με το πραγματικό τους χρώμα, αφού αυτό αλλοιώνεται με βάση τις εκάστοτε συνθήκες φωτισμού. Στη συγκεκριμένη περίπτωση, ο ήλιος φωτίζει τα αντικείμενα λιγότερο ή περισσότερο, ανάλογα με τη θέση του.

Για να υλοποιήσουμε τον **φωτισμό** στον κόσμο μας, θα χρησιμοποιήσουμε το μοντέλο **Phong**. Σύμφωνα με το μοντέλο αυτό, το χρώμα κάθε αντικειμένου καθορίζεται από τρεις επιμέρους συνιστώσες: τον περιβάλλοντα φωτισμό (ambient lighting), τη διάχυτη ανάκλαση (diffuse lighting) και την κατοπτρική ανάκλαση (specular lighting). Το τελικό χρώμα του αντικειμένου που παρατηρεί ο παρατηρητής είναι το άθροισμα των τριών συνιστωσών, καθεμία από τις οποίες προκύπτει ως συνδυασμός του εγγενούς χρώματος του υλικού και της μεταβολής του εξαιτίας της πηγής φωτός [7].

Αρχικά, θα πρέπει να προσαρμόσουμε το υλικό του κάθε αντικειμένου, δηλαδή το texture, ώστε να ταιριάζει με το μοντέλο Phong. Το texture που διαθέτουμε είναι στην πραγματικότητα η συνιστώσα K_d, δηλαδή η σταθερά διάχυτης ανάκλασης, που αντιπροσωπεύει την αναλογία της ανάκλασης της διάχυτης συνιστώσας του φωτός. Η συνιστώσα K_a, δηλαδή η σταθερά περιβάλλουσας ανάκλασης, αντιπροσωπεύει την αναλογία της ανάκλασης της περιβάλλουσας συνιστώσας του φωτός και μπορεί να υπολογιστεί ως ένα μικρό κλάσμα της K_d. Η συνιστώσα K_s, δηλαδή η σταθερά κατοπτρικής ανάκλασης, μπορεί να είναι σταθερή. Η συνιστώσα N_s, δηλαδή η λαμπρότητα του αντικειμένου, παραμένει επίσης σταθερή [8].

Οι σταθερές αυτές πολλαπλασιάζονται με τις αντίστοιχες συνιστώσες L_a, L_d, L_s του φωτός, ώστε να προκύψει το τελικό χρώμα. Η κλάση `Light`[9] διαθέτει τις πληροφορίες του φωτός, καθώς και μεθόδους ενημέρωσής του, αλλά και

[9] `common/light.h, common/light.cpp`

αποστολής του στην GPU. Θέτουμε τις συνιστώσες L_a, L_d, L_s του φωτός να είναι πλήρως λευκές, ώστε να μην τροποποιούν τη σύνθεση των χρωμάτων των υλικών.

Ο fragment shader αναλαμβάνει να τροποποιήσει το επιλεγμένο texture, κάνοντας τις απαραίτητες πράξεις, ώστε τελικά, υπολογίζοντας το άθροισμα των συνιστωσών φωτισμού I_a, I_d, I_s, να προκύψει το παρατηρούμενο χρώμα του υλικού.

Ένα σημαντικό ζήτημα είναι η θέση της πηγής φωτισμού στον κόσμο. Ο ήλιος θα πρέπει να τοποθετηθεί αρκετά μακριά από την επιφάνεια, ώστε να τη φωτίζει ολόκληρη ομοιόμορφα. Αρχικά, ο ήλιος τοποθετήθηκε σε απόσταση περίπου 3000 μονάδων μακριά από την επιφάνεια της γης, ωστόσο αποδείχθηκε ότι μία τόσο μεγάλη απόσταση δημιουργούσε προβλήματα στη σκίαση (βλ. 3.2 Σκίαση). Για τον λόγο αυτό, ο ήλιος τοποθετήθηκε πολύ πιο κοντά στην επιφάνεια, συγκεκριμένα σε απόσταση 100 μονάδων και, προκειμένου να εξασφαλιστεί όσο το δυνατόν πιο ομοιόμορφη κατανομή του φωτός στην επιφάνεια, ανοίχθηκε το πεδίο της ορθογραφικής προβολής.

Ένα πρόβλημα με τη μέθοδο αυτή είναι ότι όταν η κάμερα πλησιάζει κοντά σε σημεία της περιφέρειας των 100 μονάδων γύρω από το σημείο φωτισμού (το οποίο είναι η αρχή του τρισδιάστατου συστήματος συντεταγμένων), ο κόσμος πλέον δε φωτίζεται αρκετά. Για τη διόρθωση του προβλήματος αυτού, όταν μετακινείται η κάμερα, μετακινούμε και τον ίδιο τον ήλιο έτσι, ώστε πάντα η απόστασή του από την κάμερα να παραμένει σταθερή στις 100 μονάδες. Έτσι, ποτέ δε θα έρθουμε αντιμέτωποι με σημείο του κόσμου που να μη φωτίζεται.

Πέραν αυτών, πρέπει να μοντελοποιήσουμε και τις διάφορες ώρες της ημέρας, δηλαδή την ανατολή και τη δύση του ηλίου. Θεωρούμε πως ο ήλιος κινείται γύρω από τη γη σε μία καθορισμένη κυκλική τροχιά γύρω από την κάμερα και σε απόσταση 100 μονάδων από αυτή. Για τη μετακίνηση του ήλιου, έγινε χρήση σφαιρικών συντεταγμένων. Πλέον, όταν ο ήλιος βρίσκεται κάτω από την επιφάνεια της γης (στο «νότιο ημισφαίριο»), ο κόσμος δεν φωτίζεται[10].

Ανάλογα με την τεταγμένη y του ήλιου σε world space, μεταβάλλουμε και το χρώμα του υποβάθρου. Ξεκινάμε με το χρώμα light sky blue (#87CEFA), όταν ο ήλιος βρίσκεται στο μέγιστο ύψος και σταδιακά μειώνουμε τη φωτεινότητά του, μέχρι ο ήλιος να φτάσει σε ένα προκαθορισμένο επίπεδο πάνω από την επιφάνεια της γης, όπου κρατάμε το χρώμα σταθερό, θεωρώντας ως λυκόφως.

[10] Στην πραγματικότητα, αυτό αποτελεί ανακρίβεια. Ο φωτισμός δε λαμβάνει υπόψη τη θέση της πηγής φωτός σε σχέση με τον κόσμο, αλλά τη σχέση της δέσμης φωτός με τα κάθετα διανύσματα των αντικειμένων. Για τον λόγο αυτό, κάποιες έδρες των voxels εξακολουθούν να φωτίζονται ακόμα και όταν ο ήλιος βρίσκεται στο νότιο ημισφαίριο (π.χ. η δεξιά έδρα ενός voxel όταν ο ήλιος βρίσκεται στο τέταρτο τεταρτημόριο). Ωστόσο, το πρόβλημα αυτό διορθώθηκε με τη σκίαση.

3.2 Σκίαση

Το μοντέλο Phong εξασφαλίζει ότι οι έδρες των voxels που δεν είναι ορατές από τον ήλιο δεν φωτίζονται, παρά μόνο από τον περιβάλλοντα φωτισμό (ambient lighting). Θα πρέπει επίσης να ερευνήσουμε ποια από τα σημεία του κόσμου, τα οποία φωτίζονται κανονικά από το μοντέλο Phong, θα πρέπει να σκιαστούν, διότι μπλοκάρονται από κάποιο άλλο αντικείμενο που βρίσκεται στη διαδρομή μέχρι τον ήλιο.

Για να εντοπίσουμε τα σημεία αυτά, θα πρέπει πρώτα να εξετάσουμε πώς βλέπει τον κόσμο ο ήλιος από τη δική του οπτική γωνία. Αυτό θα καθορίσει το βάθος του κάθε αντικειμένου από τον ήλιο. Έπειτα, αν κάποιο αντικείμενο έχει μεγαλύτερο βάθος από ένα άλλο στην ίδια κατεύθυνση φωτισμού, αυτό σημαίνει ότι το πρώτο σκιάζεται από το δεύτερο.

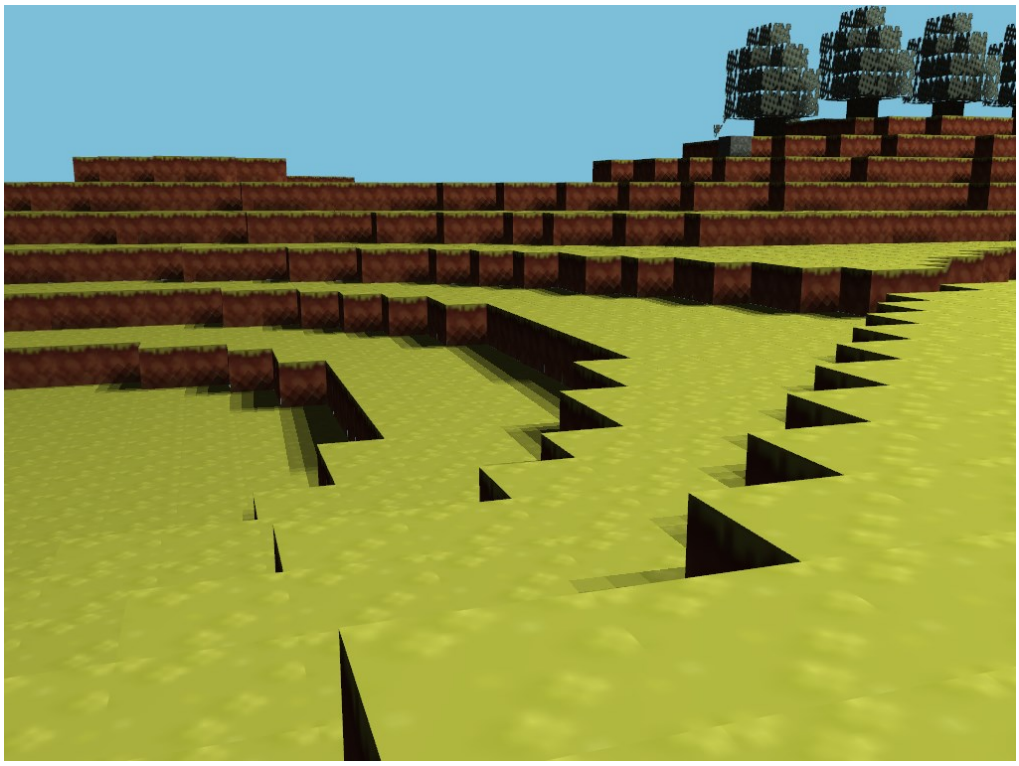
Αρχικά, κάνουμε render τον κόσμο από τη μεριά του ήλιου (depth pass). Χρησιμοποιούμε ξεχωριστούς shaders[11] για απλοποίηση της διαδικασίας. Για την ευκολότερη, επίσης, διαχείριση του interface της C++ με τους shaders, κατασκευάζουμε την κλάση `Program`[12], η οποία διατηρεί όλες τις πληροφορίες για κάθε shader. Κάνουμε render τον κόσμο, χρησιμοποιώντας τον view matrix και τον projection matrix του ήλιου. Το αποτέλεσμα δεν θέλουμε να το εμφανίσουμε στην οθόνη, αλλά αντ' αυτού το αποθηκεύουμε σε έναν frame buffer, τον οποίο θα χρησιμοποιήσουμε στη συνέχεια για τη σύγκριση του βάθους.

Έπειτα, όταν κάνουμε render τον κόσμο από τη μεριά της κάμερας (lighting pass), θα πρέπει στον fragment shader να περάσουμε τα δεδομένα του frame buffer ως texture. Έτσι, συγκρίνουμε το βάθος του τρέχοντος vertex με το πλησιέστερο στον ήλιο. Αν το πλησιέστερο στον ήλιο βάθος είναι μικρότερο από το τρέχον, συμπεραίνουμε ότι το vertex δεν είναι άμεσα ορατό από τον ήλιο και άρα σκιάζεται.

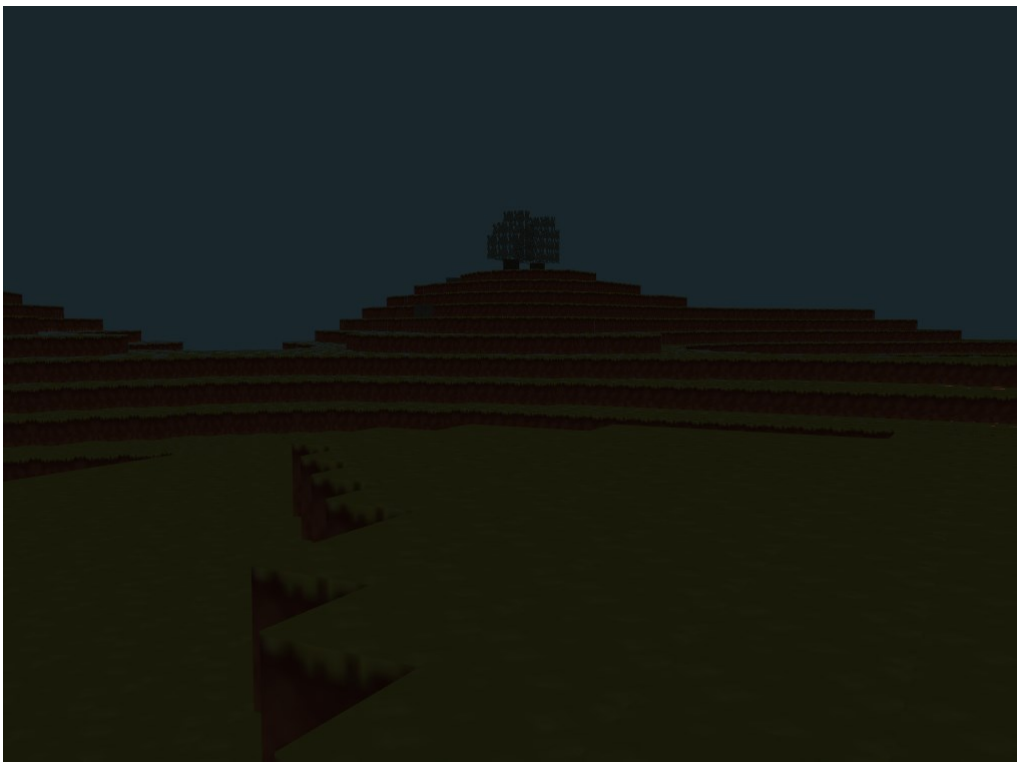
Παρακάτω φαίνονται κάποια ενδεικτικά στιγμιότυπα από το παιχνίδι, όπως αυτό έχει διαμορφωθεί μέχρι τώρα.

---

[11] `Visualcraft/DepthShader.vertexshader, Visualcraft/DepthShader.fragmentshader`
[12] `common/program.h, common/program.cpp`



Εικόνα 8. Ο κόσμος κατά τη διάρκεια του απογεύματος.



Εικόνα 7. Ο κόσμος κατά τη διάρκεια της νύχτας.

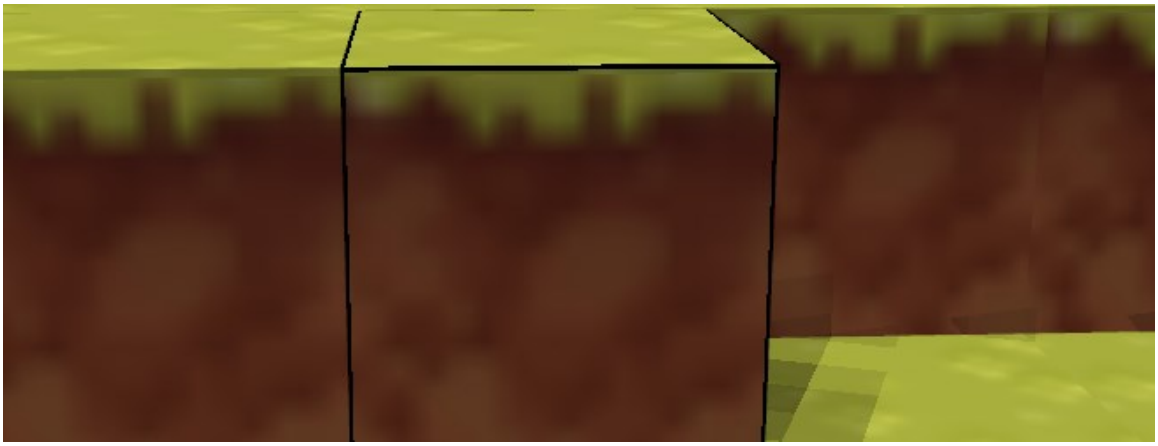
3.3 Gameplay: καταστροφή μπλοκ

Τώρα που ο κόσμος έχει δημιουργηθεί πλήρως, είναι καιρός να δώσουμε στον παίκτη τη δυνατότητα να αλληλεπιδρά με αυτόν, ώστε να μπορεί να αναπτύξει κάποια στρατηγική για το gameplay. Το πρώτο βήμα είναι η στόχευση συγκεκριμένων μπλοκ και η καταστροφή τους. Με αυτόν τον τρόπο, θα μπορεί ο χρήστης για παράδειγμα να σκάβει το έδαφος.

Για να γίνει αυτό, θα πρέπει να μπορούμε να εντοπίσουμε ποιο είναι το πιο κοντινό μπλοκ που βλέπει ο παίκτης. Αυτό πραγματοποιείται κάνοντας **ray casting**, δηλαδή στέλνοντας μία ακτίνα στην κατεύθυνση που κοιτάζει η κάμερα και ελέγχοντας το πρώτο μπλοκ που αυτή θα τμήσει. Για να βρούμε το πρώτο μπλοκ που θα τμήσει η ακτίνα, αρκεί να παίρνουμε διαδοχικά σημεία πάνω σε αυτή με ένα σταθερό βήμα και να συγκρίνουμε την τεταγμένη y του σημείου σε world space με τα όρια του υψηλότερου μπλοκ της αντίστοιχης «στήλης». Αν το y βρίσκεται εντός των ορίων, η ακτίνα τέμνει το μπλοκ, αλλιώς όχι.

Θέτουμε επίσης ένα άνω όριο προσπαθειών για αυτή τη διαδικασία. Αν δε βρεθεί τεμνόμενο μπλοκ μετά από 10, εγκαταλείπουμε. Αυτό γίνεται όχι μόνο για να αποφύγουμε «ατέρμονους» βρόχους και να γλιτώσουμε υπολογιστική ισχύ, αλλά και για να απεικονίζεται καλύτερα η πραγματικότητα, αφού ο παίκτης δεν έχει απείρως μακρύ χέρι, ώστε να καταστρέφει μπλοκ που βρίσκονται σε μεγάλη απόσταση.

Όταν βρεθεί ένα τεμνόμενο μπλοκ, στέλνουμε το `instance_ID` του (δηλαδή τη θέση του στο `voxelModel->positions`) στον fragment shader. Όταν ο fragment shader ζητηθεί να σχεδιάσει το συγκεκριμένο μπλοκ, θα μαυρίσει τις ακμές του, ώστε να γίνει εμφανές ότι έχει επιλεγεί το συγκεκριμένο μπλοκ.



Εικόνα 9. Ray-casted μπλοκ.

Όταν ο χρήστης έχει στοχεύσει ένα μπλοκ και κρατήσει πατημένο το αριστερό κλικ για προκαθορισμένο χρονικό διάστημα, τότε το μπλοκ καταστρέφεται. Αν θυμηθούμε ότι πλέον η γη αποτελείται από μόνο ένα στρώμα voxels, συμπεραίνουμε πως η κυριολεκτική καταστροφή του μπλοκ θα ήταν λανθασμένη, γιατί στο σημείο εκείνο θα δημιουργείτο κενό. Αντ' αυτού, απλώς μειώνουμε την τιμή του heightmap κατά 1, ώστε στο επόμενο καρέ το voxel να σχεδιαστεί μία θέση πιο κάτω[13].

Μετά από κάθε τροποποίηση του πλέγματος, θα χρειαστεί να ανανεώσουμε τα δεδομένα των vertex buffer objects, άρα καλούμε τη μέθοδο Voxel::createContext().

3.4 Gameplay: χτίσιμο μπλοκ

Αντίστροφα, θα πρέπει επίσης να δοθεί στον παίκτη η δυνατότητα να προσθέσει μπλοκ στο πλέγμα, ώστε να χτίζει κατασκευές.

Όταν ο χρήστης έχει στοχεύσει ένα μπλοκ (βλ. 3.3 Gameplay: καταστροφή μπλοκ) και πατήσει το δεξί κλικ, τότε προστίθεται ένα μπλοκ πάνω από το ray-casted μπλοκ. Ο παίκτης μπορεί να επιλέξει το υλικό του μπλοκ που θέλει να χτίσει, από μία γκάμα επτά υλικών, που έχουν γίνει bind στα πλήκτρα 1-6 του πληκτρολογίου. Εάν δεν έχει επιλεγεί υλικό ή έχει γίνει επαναφορά στο προεπιλεγμένο (πλήκτρα 7-0), δεν χτίζεται μπλοκ.

Το διάνυσμα `voxelModel->positions` ενημερώνεται κατάλληλα, ώστε να περιέχει τις συντεταγμένες των κέντρων βάρους όλων των μπλοκ, είτε του υπάρχοντος εδάφους, είτε αυτών που έχει προσθέσει ο παίκτης. Επίσης, ο fragment shader τροποποιείται έτσι, ώστε όταν ένα μπλοκ έχει καθορισμένο υλικό, να κάνει override τη γνωστή απόφαση για texture βάσει του υψομέτρου.

Προσθέτουμε επίσης στο texture atlas μερικά ακόμα υλικά που θα φανούν χρήσιμα στον παίκτη, όπως τούβλο, γυαλί, τσιμέντο και πέτρα.

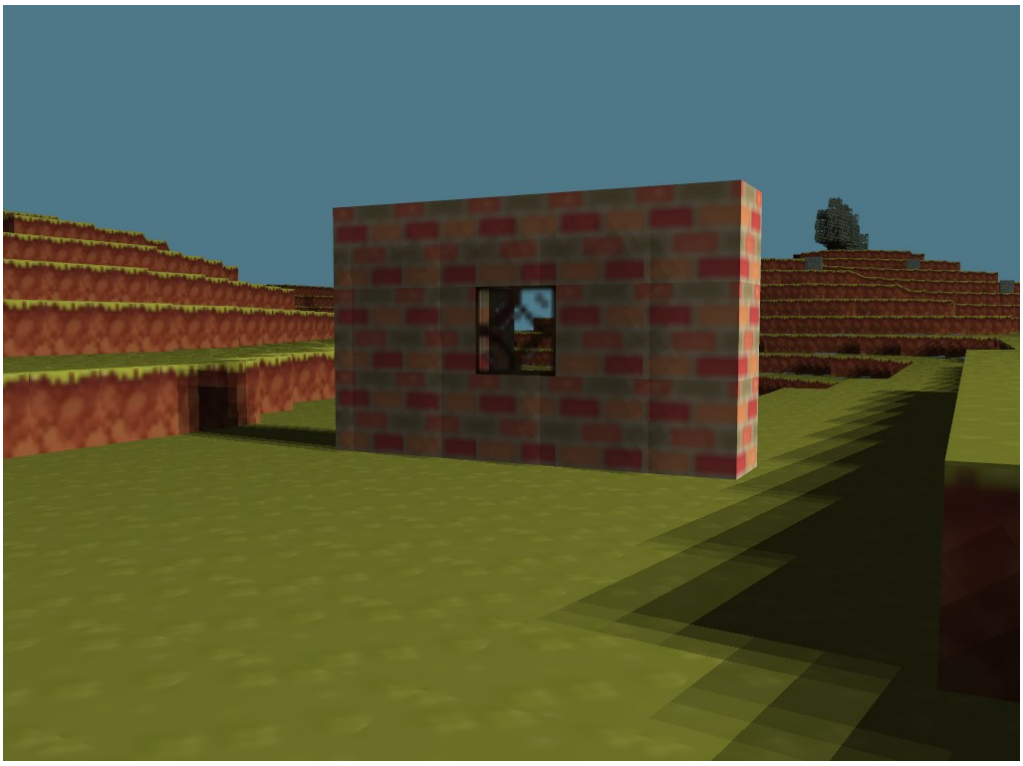
3.5 Σύνοψη δεύτερου μέρους

Με την ολοκλήρωση των παραπάνω, έχουμε εφαρμόσει φωτισμό και σκίαση στον κόσμο και έχουμε δώσει στον παίκτη την ικανότητα να καταστρέψει και να χτίσει μπλοκ, σύμφωνα με τον τρόπο που αυτός επιθυμεί.

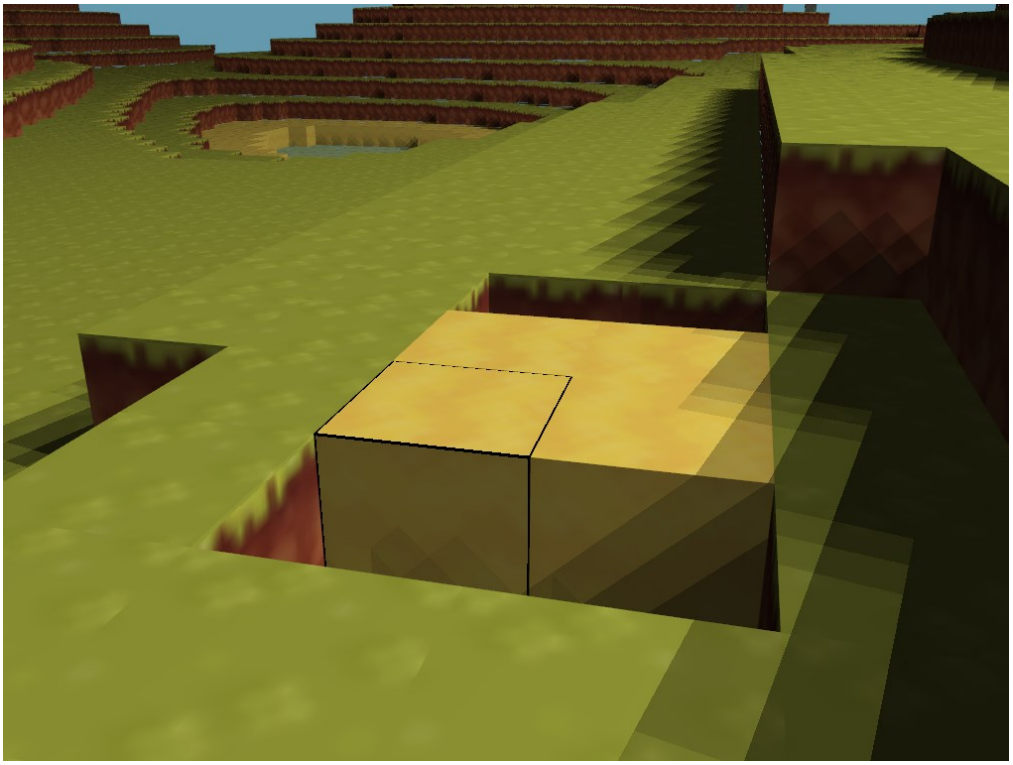
Παρακάτω φαίνονται κάποια ενδεικτικά στιγμιότυπα από το παιχνίδι.

---

[13] Η προσέγγιση αυτή οδηγεί σε σφάλματα. Μπορεί να φέρει σε δύο γειτονικές στήλες διαφορά υψομέτρου μεγαλύτερη της μίας μονάδας και να εμφανιστεί κενό. Περισσότερα στο Κεφάλαιο 4. Σφάλματα – Χώρος για βελτίωση.



Εικόνα 11. Ο παίκτης έχει κατασκευάσει έναν τοίχο με ένα παράθυρο.



Εικόνα 10. Ο παίκτης έχει καταστρέψει κάποια μπλοκ γρασιδιού και έχει προσθέσει άμμο.

Κεφάλαιο 4. Σφάλματα – Χώρος για βελτίωση

Στο παρόν κεφάλαιο, σημειώνονται κάποια γνωστά σφάλματα που προκύπτουν κατά το gameplay. Επισημαίνονται επίσης τρόποι, με τους οποίους η υλοποίηση του παιχνιδιού μπορεί να βελτιωθεί.

Πολλά σφάλματα πηγάζουν από την απουσία ενός ενιαίου voxel mesh, το οποίο επιτρέπει τη σχεδίαση μόνο των ορατών εδρών κάθε voxel. Αν είχε υλοποιηθεί ένα voxel mesh, θα μπορούσε να είχε αποφευχθεί και η μετατροπή του 100x100x100 πλέγματος σε 1000x1x1000, το οποίο προκαλεί τα ακόλουθα προβλήματα:

- Όταν δύο γειτονικές «στήλες» έχουν υψόμετρο που διαφέρει περισσότερο από μία μονάδα, τότε εμφανίζεται κενός χώρος (Εικόνα 12).
- Όταν καταστραφεί ένα μπλοκ και «εμφανιστεί αυτό που βρίσκεται από κάτω του», αυτό που θα προκύψει θα είναι -λανθασμένα- ίδιο με το αρχικό. Για παράδειγμα, αν καταστρέψουμε ένα μπλοκ γρασιδιού, αυτό που θα εμφανιστεί από κάτω θέλουμε να είναι μπλοκ ρίζας γρασιδιού. Αντ' αυτού όμως, εμφανίζεται ξανά μπλοκ γρασιδιού, αφού στην πραγματικότητα είναι το αρχικό που μετακινήθηκε κατά μία θέση προς τα κάτω (Εικόνα 13).

Ορατό σφάλμα υπάρχει επίσης, όταν ερχόμαστε αντιμέτωποι με voxels που βρίσκονται δύο στάθμες κάτω από την επιφάνεια της θάλασσας. Εκεί είναι φανερό ότι η στάθμη δεν βρίσκεται σε ένα σταθερό επίπεδο, αλλά στην πραγματικότητα αποτελεί ένα επιπλέον texture, το οποίο έχει εφαρμοστεί στις έδρες εκείνων των voxels (Εικόνα 14).

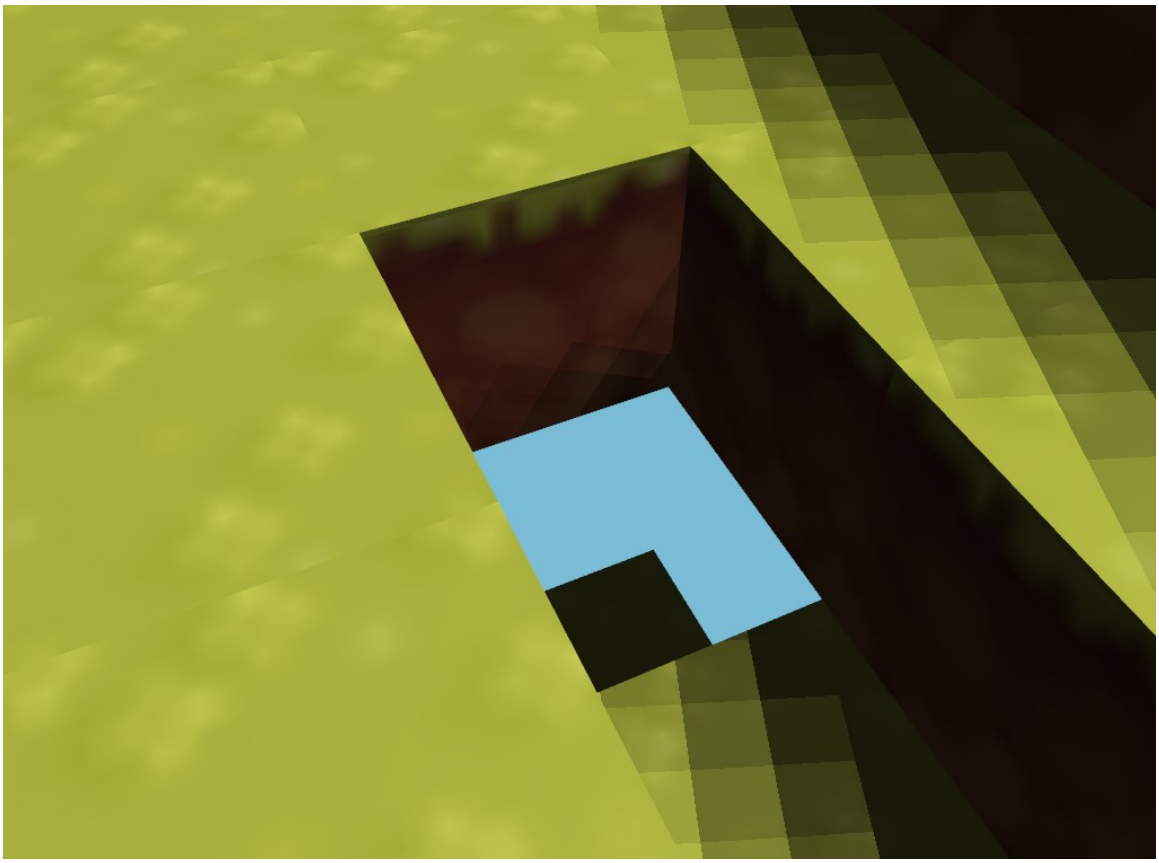
Όταν ο ήλιος βρίσκεται κοντά στην επιφάνεια της γης και η κλίση της δέσμης φωτός είναι πολύ μικρή, εμφανίζεται πρόβλημα κβαντοποίησης (quantization), το οποίο προκαλεί την εμφάνιση λανθασμένων σκιών στον κόσμο (Εικόνα 15).

Κάποιες φορές, και συγκεκριμένα όταν ένα ή δύο μπλοκ βρίσκονται ψηλότερα από κάθε άλλο γείτονά τους, δεν λειτουργεί σωστά το μπλοκάρισμα κίνησης της κάμερας, με αποτέλεσμα η κάμερα να κινείται εντός της επιφάνειάς της γης και όχι πάνω σε αυτή (Εικόνα 16).

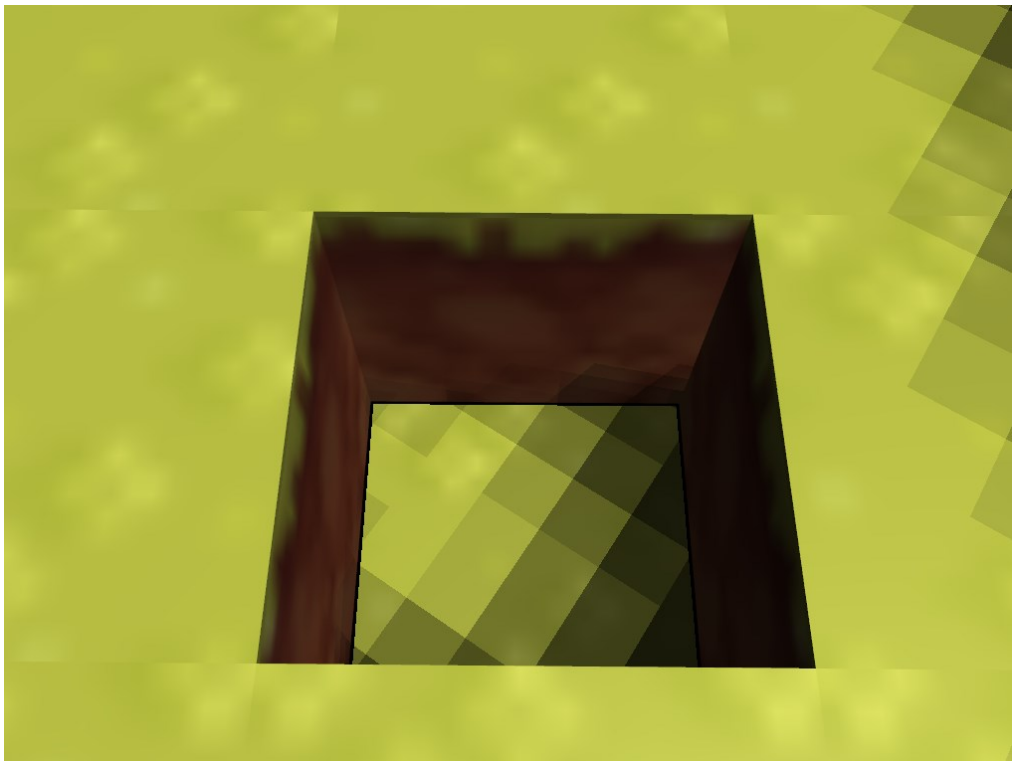
Ένα πεδίο στο οποίο χωρά μεγάλη βελτίωση είναι τα αντικείμενα που έχουν τοποθετηθεί στην επιφάνεια της γης. Προκειμένου τα αντικείμενα να μοιάζουν περισσότερο στα μοντέλα από τα οποία έχουν παραχθεί, θα πρέπει να αυξηθεί η ανάλυση, δηλαδή να οριστεί ένα πολύ μεγαλύτερο XxYxZ πλέγμα για το αντικείμενο. Ταυτόχρονα, θα πρέπει να γίνει αρκετή σμίκρυνση του αντικειμένου, ώστε το μέγεθός του να διατηρηθεί σε λογικά πλαίσια, αλλά και να εξασφαλιστεί η σωστή στοίχισή του στο mesh.

Επίσης, πρέπει να δοθεί η δυνατότητα αλληλεπίδρασης με καταστροφή και χτίσιμο μπλοκ και για τα υπάρχοντα αντικείμενα. Προς το παρόν, ο παίκτης δεν μπορεί να αλληλεπιδράσει με τα δέντρα και τις πέτρες που έχουν τοποθετηθεί.

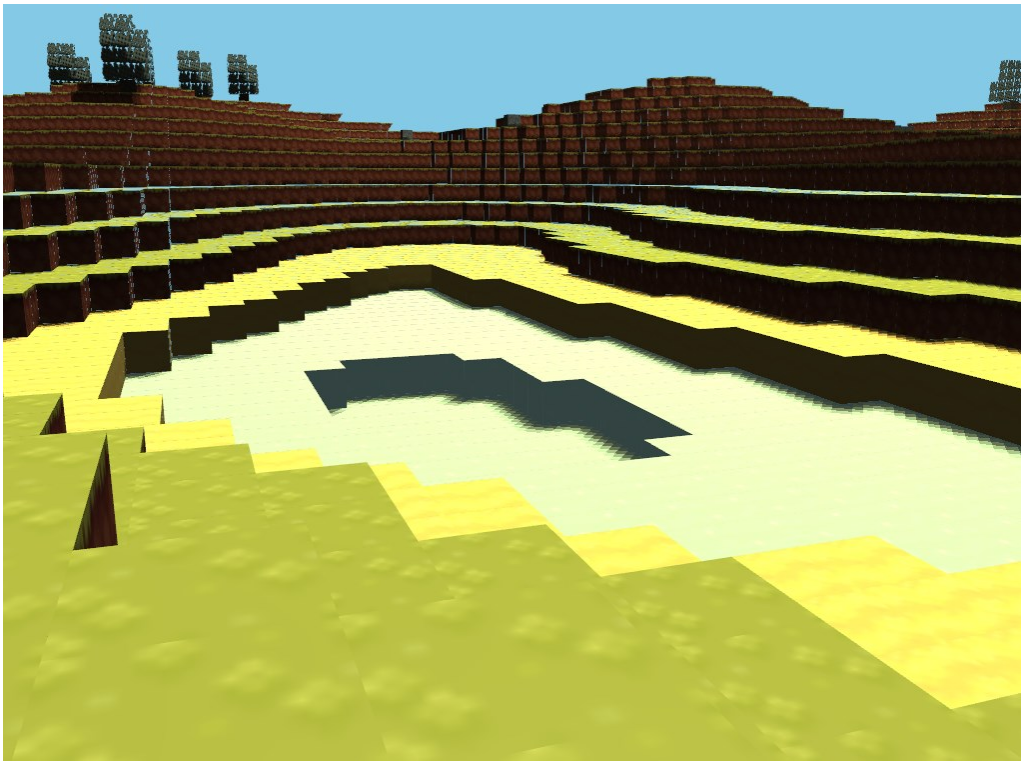
Τέλος, το παιχνίδι πρόκειται να γίνει πολύ πιο αποδοτικό, όταν το πλέγμα των voxels χωριστεί σε μικρές ομάδες (chunks) και κάθε μία σχεδιάζεται σταδιακά, ανάλογα με τη θέση της κάμερας. Με αυτόν τον τρόπο, όχι μόνο θα μειωθεί πάρα πολύ η απαίτηση της υπολογιστικής ισχύος, αλλά και θα δοθεί η δυνατότητα για δημιουργία απέραντου κόσμου.



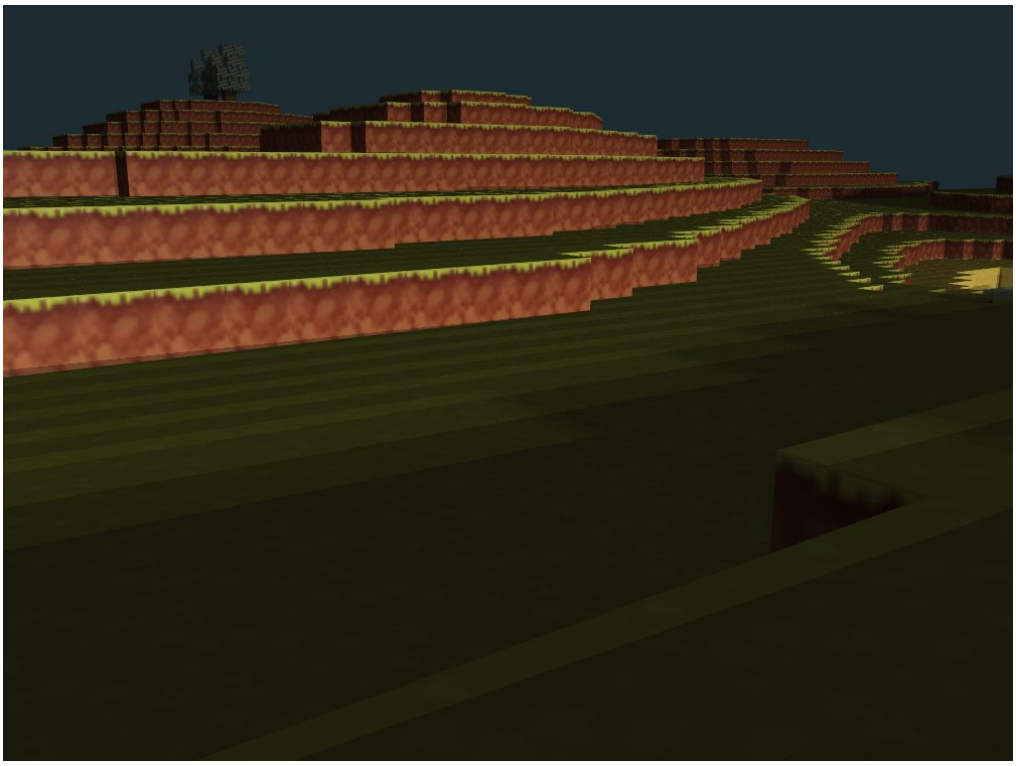
Εικόνα 12. Σφάλμα: δύο γειτονικές «στήλες» έχουν υψομετρική διαφορά μεγαλύτερη από δύο μονάδες.



Εικόνα 13. Σφάλμα: μετά την καταστροφή του μπλοκ γρασιδιού, εμφανίζεται πάλι μπλοκ γρασιδιού αντί για μπλοκ ρίζας γρασιδιού.



Εικόνα 14. Σφάλμα: η στάθμη του νερού δεν παραμένει πάντα στο ίδιο ύψος.



Εικόνα 15. Σφάλμα: Λανθασμένες σκιάσεις όταν ο ήλιος βρίσκεται κοντά στην επιφάνεια της γης.

Εικόνα 16. Σφάλμα: η κάμερα έχει εισέλθει λανθασμένα μέσα στην επιφάνεια της γης.

Βιβλιογραφία

[1] Wikipedia Community, "Voxel," 20 December 2022. [Online]. Available: https://en.wikipedia.org/wiki/Voxel.

[2] Wikipedia Community, "OpenGL," 6 February 2023. [Online]. Available: https://en.wikipedia.org/wiki/OpenGL.

[3] J. de Vries, "Learn OpenGL: Instancing," [Online]. Available: https://learnopengl.com/Advanced-OpenGL/Instancing. [Accessed 22 February 2023].

[4] Wikipedia Community, "Perlin Noise," 10 February 2023. [Online]. Available: https://en.wikipedia.org/wiki/Perlin_noise.

[5] Solarian Programmer, "Perlin Noise in C++11," 18 July 2012. [Online]. Available: https://solarianprogrammer.com/2012/07/18/perlin-noise-cpp-11/.

[6] Phyronnaz, "GitHub: VoxelAssets," [Online]. Available: https://github.com/Phyronnaz/VoxelAssets. [Accessed 22 February 2022].

[7] Κ. Μουστάκας, "Φωτισμός," in *Διαλέξεις Γραφικών και Εικονικής Πραγματικότητας*, Πάτρα, 2023.

[8] Wikipedia Community, "Phong reflection model," 12 January 2023. [Online]. Available: https://en.wikipedia.org/wiki/Phong_reflection_model.

[9] J. de Vries, "Learn OpenGL: Textures," [Online]. Available: https://learnopengl.com/Getting-started/Textures. [Accessed 22 February 2023].