

**«Απόδοση πολλαπλών θραυσμάτων (multifragment rendering) και
μεγάλων σκηνών σε μηχανές γραφικών»**

Η
ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ ΕΞΕΙΔΙΚΕΥΣΗΣ

Υποβάλλεται στην

ορισθείσα από την Γενική Συνέλευση Ειδικής Σύνθεσης
του Τμήματος Πληροφορικής
Εξεταστική Επιτροπή

από τον

Ευάγγελο-Αλέξανδρο Ευταξόπουλο

ως μέρος των Υποχρεώσεων

για τη λήψη

του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΣΤΗΝ ΠΛΗΡΟΦΟΡΙΚΗ
ΜΕ ΕΞΕΙΔΙΚΕΥΣΗ ΣΤΟ ΛΟΓΙΣΜΙΚΟ

Ιούλιος 2016

ΠΕΡΙΕΧΟΜΕΝΑ

	Σελ
ΕΥΧΑΡΙΣΤΙΕΣ	i
ΠΕΡΙΛΗΨΗ	ii
EXTENDED ABSTRACT IN ENGLISH	iii
1. ΕΙΣΑΓΩΓΗ	1
1.1. Πολυθραυσματική απόδοση σκηνής	1
1.2. Απόδοση μεγάλης σκηνής με μηχανή γραφικών	2
2. ΥΠΟΒΑΘΡΟ	4
2.1. Rendering	4
2.2. Shaders	5
2.3. Order Independent Transparency	6
2.4. Lightweight Java Game Library	6
2.5. Unreal Engine	7
3. ΥΛΟΠΟΙΗΣΗ ΑΛΓΟΡΙΘΜΩΝ ΠΟΛΥΘΡΑΥΣΜΑΤΙΚΗΣ ΑΠΟΔΟΣΗΣ	8
3.1. Εισαγωγικά	8
3.1.1. A-buffer	8
3.1.2. K-buffer	10
3.2. Υλοποίηση αλγορίθμων	11
3.2.1. LWJGL	11
3.2.2. A-buffer	17

3.2.3. K-buffer	18
3.3. Πειραματική αποτίμηση	20
3.3.1. A-buffer	20
3.3.2. K-buffer	20
3.3.3. Σύγκριση και αποτίμηση αποτελεσμάτων	27
4. ΔΗΜΙΟΥΡΓΙΑ ΜΕΓΑΛΗΣ ΥΒΡΙΔΙΚΗΣ ΣΚΗΝΗΣ	30
4.1 Εισαγωγικά	30
4.2 Workflow	31
4.2.1 Βασικό τοπίο	31
4.2.2 Επιλογή σημείων ενδιαφέροντος	32
4.2.3 Εμπλουτισμός σκηνής	33
4.2.4 Particle Effects	35
4.2.5 Blueprints	38
4.2.6 Φωτισμός	39
4.2.7 Στερεοσκοπικότητα	40
4.2.8 Matinee	42
4.2.9 Πειραματικά αποτελέσματα	43
5. ΣΥΜΠΕΡΑΣΜΑΤΑ	46
5.1 Πολυθραυσματική απόδοση σκηνής	46
5.2 Απόδοση υβριδικής σκηνής	46
6. ΑΝΑΦΟΡΕΣ	47
7. ΣΥΝΤΟΜΟ ΒΙΟΓΡΑΦΙΚΟ	48

ΕΥΧΑΡΙΣΤΙΕΣ

Θα ήθελα να ευχαριστήσω τον επιβλέπων καθηγητή της μεταπτυχιακής αυτής διατριβής, Καθηγητή κ. Φούντο Ιωάννη, για την καθοδήγηση και πολύτιμη βοήθειά του στην εκπόνηση της εργασίας, καθώς και για τις γνώσεις και υποστήριξη που μου προσέφερε καθ' όλη την διάρκεια της φοίτησής μου.

ΠΕΡΙΛΗΨΗ

Ευάγγελος Ευταξόπουλος του Νικολάου και της Μπίργκιτ. MSc, Τμήμα Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Ιούλιος 2016. «Απόδοση πολλαπλών θραυσμάτων (multifragment rendering) και μεγάλων σκηνών σε μηχανές γραφικών». Επιβλέποντας: Ιωάννης Φούντος.

Για την παρούσα εργασία υλοποιήθηκαν αλγόριθμοι απόδοσης πολλαπλών θραυσμάτων A-buffer, K-buffer τεχνολογίας με τη χρήση λογισμικού Lightweight Java Game Library και γλώσσας προγραμματισμού GLSL. Αρχικά παρουσιάζεται μία πειραματική συγκριτική αξιολόγηση των προαναφερόμενων μεθόδων πολυθραυσματικής απόδοσης A-buffer και K-buffer ως προς την αποδοτικότητα χώρου και χρόνου. Στη συνέχεια παρατίθεται η διαδικασία καθώς και μία επισκόπηση των προβλημάτων που αντιμετωπίστηκαν στην δημιουργία μίας μεγάλης υβριδικής σκηνής αποτελούμενης από πραγματικά 3D μοντέλα, τα οποία προέρχονται από ανακατασκευή 3D αντικειμένων μέσω φωτογραφιών, καθώς τεχνητά 3D μοντέλα, τα οποία δημιουργήθηκαν με εργαλεία γραφικών. Τέλος παρουσιάζουμε τη χρησιμοποίηση μίας τέτοιας υβριδικής 3D σκηνής στην πλατφόρμα γραφικών Unreal Engine και την απόδοσή της, και τη συνολική αποτίμηση των προαναφερθείσων διαδικασιών.

EXTENDED ABSTRACT IN ENGLISH

Eftaxopoulos Evangelos, MSC, Computer Science Department, University of Ioannina, Greece, July 2016, «Multifragment rendering and Large scene creation in graphics engines».

Thesis Supervisor: Ioannis Fudos.

We consider two aspects of rendering. First, we study the problem of rendering all surfaces in a 3D scene (even the hidden ones) by implementing and comparing two well-known multifragment algorithms by using as development platform the Lightweight Java Game Library, and GLSL shaders. The purpose of this line of work is to compare the quality of the final result and the efficiency of mesh rendering, for order independent transparency. Then we consider the problem of building a large three dimensional scene that contains actual (reconstructed objects) and artificial components by using of a game/graphics engine. The workflow for achieving this task is presented along with, setbacks, issues and solutions related to the magnitude of the scene and the compatibility of the modeling formats employed.

More specifically, for the rendering we have implemented transparency for two commonly used models, the Stanford Dragon and Bunny, as well as a third terrain mesh used in the aforementioned video. A key difference between the two algorithms is that A-buffer has an unlimited depth, resulting in potentially unlimited memory requirements, whereas the K buffer works with a fixed K depth value, where K represents the amount of fragments stored.. For K-buffer we have conducted experiment with three depth values $K=10$, $K=5$, $K=2$. The depth level on A-buffer used was 16, since the largest overall mesh depth does not exceed 10.

A-buffer provides very good visual results, with the transparency visualizing every surface in the scene. This results in high memory requirements and decent fps.. The results of the three different K values, the algorithms result in varying visible quality depending on the value of depth. A-buffer provides smaller fps and requires more memory as compared with the K-buffer,

In the second part of this thesis we report the development of a large scene, depicting the area of Meteora through time,

For the Meteora area we have used models reconstructed from aerial photos from an unmanned aerial vehicle (airplane) and from drones. The models were edited for missing parts and were disseminated for use in the graphics engine.

For the artificial part of the scene we have used several Unreal assets, third party models, and modeled developed in house. To produce the stereoscopic video we have used a set up of two virtual cameras in the Unreal engine and a third camera that capture their result side by side. The end result is a video of 3840x1080p, so in stereoscopic playback devices the output for each eye is 1920x1080p, with a framerate of 60 fps.

During the final production of the video, through the video editing software, 39,000 images were created for the 10-minute length with each image resolution at 1920x1080 and size around 7Mb. We present running time information for each part of this process.

ΚΕΦΑΛΑΙΟ 1. ΕΙΣΑΓΩΓΗ

1.1 Πολυθραυσματική απόδοση σκηνής

1.2 Απόδοση μεγάλης σκηνής με μηχανή γραφικών

Η εργασία εστιάστηκε σε δύο αντικείμενα στο χώρο των γραφικών. Συγκεκριμένα, αρχικά εξετάσαμε την απόδοση τρισδιάστατων σκηνών με την χρήση πολυθραυσματικών αλγορίθμων και συγκρίναμε τους αλγόριθμους που χρησιμοποιήθηκαν όσον αφορά την απόδοση. Στη συνέχεια αναφέρουμε την εμπειρία μας από τη διαδικασία δημιουργίας μιας μεγάλης σκηνής σε μηχανή γραφικών η οποία περιέχει πραγματικά αλλά και τεχνητά μοντέλα και την τελική παραγωγή στερεοσκοπικού βίντεο. Παραθέτουμε λοιπόν αρχικά την υλοποίηση των αλγορίθμων για την απόδοση των μοντέλων και αναφέρουμε τους περιορισμούς αλλά και τα πλεονεκτήματα του κάθε αλγορίθμου που χρησιμοποιήθηκε. Στη συνέχεια παραθέτουμε τα βήματα δημιουργίας της σκηνής στη μηχανή γραφικών με εκτενή επεξήγηση των εργαλείων και την χρήση για τη χρήση τους για την επίλυση των προβλημάτων που αντιμετωπίστηκαν

1.1 Πολυθραυσματική απόδοση σκηνής

Στην επιστήμη της πληροφορικής, και συγκεκριμένα στα γραφικά υπολογιστών, έχουν υπάρξει πολλές προσπάθειες για την καλύτερη απόδοση τρισδιάστατων μοντέλων με το μικρότερο δυνατό κόστος μνήμης και την μεγαλύτερη δυνατή απόδοση, σε σχέση με την πολυπλοκότητα της σκηνής.

Η απόδοση τρισδιάστατων μοντέλων με τη χρήση προγραμμάτων υπολογιστών, ονομάζεται Rendering. Σε αντίθεση με το παρελθόν, η εξέλιξη της τεχνολογίας και η εισαγωγή των shaders στην απόδοση τρισδιάστατων μοντέλων κατέστησε πλέον τη σωλήνωση της απόδοσης (rendering pipeline) προγραμματίσιμη σε πραγματικό χρόνο. Στην προκειμένη περίπτωση χρησιμοποιήσαμε shaders για την υλοποίηση δύο μεθόδων απόδοσης, οι οποίες είναι οι A-buffer και K-buffer. Καθώς, όμως, έχουν διαφορετικά πλεονεκτήματα και περιορισμούς, συγκρίναμε την απόδοσή τους στην απόδοση του ίδιου μοντέλου, και πιο συγκεκριμένα για την απόδοση διαφάνειας τύπου Order Independent Transparency (OIT) πάνω στο μοντέλο. Η υλοποίηση των shaders έγινε πειραματικά σε περιβάλλον Lightweight Java Game Library, καθώς δεν είχε υλοποιηθεί στο παρελθόν.

Παρακάτω, στο κεφάλαιο 2, θα παρατεθούν στοιχεία υπόβαθρου, ώστε ο αναγνώστης να εξοικειωθεί με τις έννοιες της πολυθραυσματικής απόδοσης, των shaders και των υπόλοιπων εργαλείων που χρησιμοποιήθηκαν. Στο κεφάλαιο 3 θα γίνει αναλυτική επεξήγηση της διαδικασίας υλοποίησης των αλγορίθμων ξεχωριστά για τις δύο μεθόδους (A-buffer, K-buffer), και τέλος μια σύγκριση μεταξύ των δύο με την παράθεση των αποτελεσμάτων πειραματικής αποτίμησής τους.

1.2 Απόδοση μεγάλης σκηνής με μηχανή γραφικών

Το δεύτερο σκέλος της εργασίας αποτελείται από τη δημιουργία ενός στερεοσκοπικού βίντεο μεγάλης διάρκειας με τη χρήση μηχανής γραφικών. Το χαρακτηριστικό της εν λόγω σκηνής, πέραν της στερεοσκοπικότητας, είναι η μεγάλη πολυπλοκότητά της. Πιο συγκεκριμένα, καθώς το βίντεο αφορά παρουσίαση της περιοχής των Μετεώρων, η σκηνή αποτελείται από πραγματικά τρισδιάστατα μοντέλα τα οποία αναδημιουργήθηκαν από φωτογραφίες της περιοχής, και τεχνητά μοντέλα τριών διαστάσεων, κατασκευασμένων με εργαλεία γραφικών, με αποτέλεσμα το μέγεθος της σκηνής να είναι αντίστοιχο με την πραγματική έκταση της περιοχής. Η μηχανή γραφικών που χρησιμοποιήθηκε είναι η Unreal Engine, η οποία αποτελεί ένα από τα βασικά εργαλεία παραγωγής τρισδιάστατου περιεχομένου.

Στο κεφάλαιο 3, θα παρατεθούν έννοιες που αφορούν στη δημιουργία μιας τέτοιας σκηνής, καθώς και επεξήγηση των εργαλείων που χρησιμοποιήθηκαν. Στη συνέχεια, στο κεφάλαιο 4 θα γίνει ανάλυση της διαδικασίας και επεξήγηση των βημάτων προς την πραγμάτωση της σκηνής, καθώς και τα προβλήματα που προέκυψαν και η επίλυσή τους. Τέλος, στο κεφάλαιο 5 θα γίνει μια συμπερασματική συζήτηση σχετικά με την τελική απόδοση της σκηνής, καθώς και τα τεχνικά χαρακτηριστικά της.

ΚΕΦΑΛΑΙΟ 2. ΥΠΟΒΑΘΡΟ

2.1 Rendering

2.2 Shaders

2.3 Order Independent Transparency (OIT)

2.4 Lightweight Java Game Library (LWJGL)

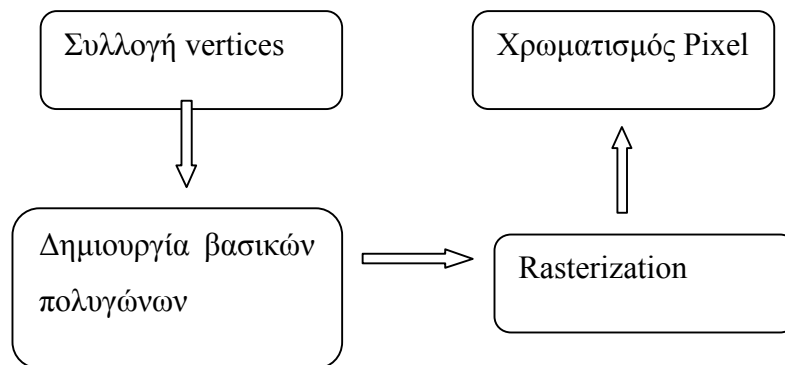
2.5 Unreal Engine

2.1 Rendering

Στην επιστήμη της πληροφορικής, η απόδοση γραφικών ονομάζεται rendering. Για την επίτευξη της απόδοσης γραφικών ακολουθείται μια διαδικασία βημάτων η οποία ονομάζεται rendering pipeline (βλ. Σχήμα 2.1). Τα βασικά βήματά της είναι αρχικά η συλλογή των vertices (κόμβων) και η δημιουργία των βασικών πολυγώνων βάσει των vertices στο οπτικό πεδίο του θεατή και ακολουθεί ο υπολογισμός των normals (κάθετων διανυσμάτων) των πολυγώνων και των κόμβων. Έστερα γίνεται το rasterization το οποίο “χωρίζει” την σκηνή σε θραύσματα και τέλος πραγματοποιούνται τα fragment operations, όπου τα θραύσματα συνδυάζονται και γίνονται pixels. Η σωλήνωση πλέον είναι προγραμματίσιμη καθώς υπάρχει η δυνατότητα ανάπτυξης προγραμμάτων (shaders), τα οποία μπορούν σε πραγματικό χρόνο να πραγματοποιούν λειτουργίες πάνω στα vertices και τα θραύσματα.

Όσον αφορά τους shaders, γενικά αναφερόμενοι, είναι προγράμματα τα οποία επιτρέπουν την παραγωγή των απαιτούμενων επιπέδων χρώματος σε μια εικόνα ή

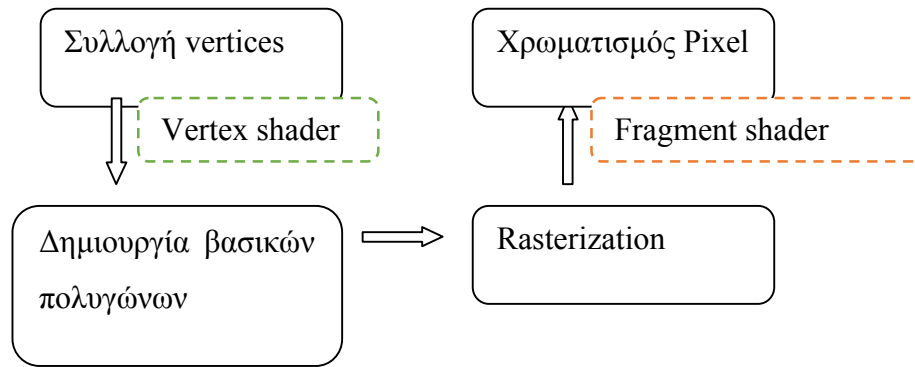
στην εποχή μας πλέον για τη δημιουργία ειδικών εφέ αλλά και στο post-prossessing των βίντεο. Η σημασία της ύπαρξης των προγραμμάτων αυτών έγκειται στο γεγονός πως προσφέρει πολύ μεγάλο βαθμό ευελιξίας, δίνοντας τη δυνατότητα άμεσης επεξεργασίας και τροποποίησης με τη χρήση αλγορίθμων, και κατά συνέπεια τον προγραμματισμό του rendering pipeline με πάρα πολλές δυνατότητες και επιλογές απόδοσης τρισδιάστατων μοντέλων.



Σχήμα 2.1: Rendering Pipeline

2.2 Shaders

Οι shaders είναι προγράμματα τα οποία γράφονται για να επεξεργαστούν ομάδες από στοιχεία ταυτόχρονα, όπως κάθε pixel σε μια περιοχή της οθόνης ή για κάθε vertex ενός μοντέλου. Η δυνατότητα αυτή, σε συνδυασμό με την ύπαρξη πολλαπλών rendering pipelines σε νέες κάρτες γραφικών, επιτρέπει την παράλληλη επεξεργασία της σκηνής, το οποίο βελτιώνει σε μεγάλο βαθμό την διεκπεραιωτική ικανότητα του υπολογιστή. Στο πείραμα αυτό χρησιμοποιήθηκαν δυο ειδών shaders: vertex shaders και fragment shaders. Ένας vertex shader ενεργοποιείται αμέσως μετά την εισαγωγή των vertices στο pipeline. Ως είσοδο λαμβάνει τα vertices και μετά από οποιαδήποτε επεξεργασία, δίνει ως έξοδο πληροφορία θέσης του κάθε vertex. Ένας fragment shader ενεργοποιείται αμέσως μετά το rasterization της σκηνής και σαν είσοδο λαμβάνει τα θραύσματα που παράγονται. Ως έξοδο παράγει το χρώμα του κάθε θραύσματος(βλ. Σχήμα 2.2).



Σχήμα 2.2: Προγραμματίσιμο Rendering Pipeline

2.3 Order Independent Transparency (OIT)

Ένα δύσκολο θέμα στην πληροφορική ήταν και παραμένει η απόδοση αληθοφανούς διαφάνειας σε τρισδιάστατα αντικείμενα. Συνήθως η απόδοση διαφάνειας γίνεται συνδυάζοντας όλες τις επιφάνειες σε έναν buffer. Κάθε επιφάνεια, ανάλογα με την τιμή α value της, δηλαδή ένα λόγο (ratio) μετάδοσης του φωτός, αφαιρεί κάθε προϋπάρχον χρώμα, και προσθέτει λίγο από το χρώμα της. Για να γίνει σωστή απόδοση είναι απαραίτητο η κάθε επιφάνεια να προστεθεί με ταξινομημένη σειρά, η οποία επηρεάζει την ορατότητα ή μη κάθε επιφάνειας. Η ταξινόμηση όμως γεωμετρίας, δηλαδή των πολυγώνων, στο πρώτο βήμα του rendering μπορεί να αποβεί πολύ χρονοβόρα και απαιτεί πολλούς πόρους από το σύστημα. Με την χρήση shaders είναι δυνατή η επίτευξη order independent transparency, κάτι που σημαίνει πως πλέον δεν χρειάζεται η ταξινόμηση των πολυγώνων στην αρχή, γιατί με την χρήση του fragment shader η ταξινόμηση γίνεται σε επίπεδο θραύσματος για κάθε pixel, μετά το βήμα rasterization το οποίο έχει φανεί πολύ πιο αποδοτικό [Maule et al., '11].

2.4 Lightweight Java Game Library

Η LWJGL είναι μια open source βιβλιοθήκη για τη γλώσσα προγραμματισμού Java, η οποία παρέχει υποστήριξη για άλλες βιβλιοθήκες δημιουργίας γραφικών. Ο σκοπός

δημιουργίας της ήταν να παρέχει σε προγραμματιστές πρόσβαση σε πόρους και εργαλεία τα οποία έλειπαν ή δεν λειτουργούσαν σωστά στο περιβάλλον της Java.

2.5 Unreal engine

Όσον αφορά τη δημιουργία της μεγάλης σκηνής, με την πρόοδο της τεχνολογίας των γραφικών, αναπτύχθηκαν εργαλεία δημιουργίας ανάπτυξης τρισδιάστατων σκηνών. Αυτά ονομάζονται μηχανές γραφικών και παρέχουν πολλές λειτουργίες για την ανάπτυξη πολύπλοκου περιεχομένου. Η κυρίαρχη μηχανή γραφικών πλέον είναι η Unreal Engine. Είναι γραμμένη σε γλώσσα C++ και μπορεί να παράξει βίντεο καθώς και διαδραστικό περιεχόμενο για όλα τα γνωστά λειτουργικά συστήματα.

ΚΕΦΑΛΑΙΟ 3. ΥΛΟΠΟΙΗΣΗ ΑΛΓΟΡΙΘΜΩΝ ΠΟΛΥΘΡΑΥΣΜΑΤΙΚΗΣ ΑΠΟΔΟΣΗΣ

3.1 Εισαγωγικά

3.2 Υλοποίηση αλγορίθμων

3.3 Πειραματική αποτίμηση

3.1 Εισαγωγικά

Στο πέρασμα του χρόνου και με την συνειδητοποίηση της σημασίας της ψηφιακής οπτικοποίησης, υπήρξε μεγάλο ενδιαφέρον και εντατική έρευνα με σκοπό τη βέλτιστη δημιουργία τρισδιάστατων μοντέλων βάσει του σύγχρονου ανά τα έτη υλικού και αποδοτικότητας των ηλεκτρονικών υπολογιστών. Η πρώτη αναφερθείσα προσπάθεια, η οποία ονομάστηκε Z-buffer ή αλλιώς buffer βάθους, αφορούσε τον εντοπισμό και αποθήκευση του βάθους ενός παραγόμενου pixel, z συντεταγμένης, και την τοποθέτησή του σε συντεταγμένες x, y. Αυτό βοηθά στην περίπτωση απόδοσης δύο αντικειμένων στο ίδιο σημείο, συγκρίνοντας το βάθος του κάθε θραύσματος και παραλείποντας εκείνο που είναι βαθύτερα και επικαλύπτεται από το εγγύτερο [Catmull, '74].

Πάνω στη συγκεκριμένη μέθοδο βασίστηκαν περαιτέρω έρευνες για τη βελτιστοποίηση της ψηφιακής οπτικοποίησης μοντέλων, περιορίζοντας τα προβλήματα που παρέθετε η προκαθορισμένη μέθοδος βάθους, εκμεταλλευόμενες την συνεχή εξέλιξη της τεχνολογίας. Βάσει των προαναφερθεισών ερευνών

σχεδιάστηκαν οι αλγόριθμοι A-buffer [Carpenter, '84], και K-buffer [Callahan, '05], οι οποίοι μας απασχόλησαν στο συγκεκριμένο πείραμα.

3.1.1 A-buffer

Ο A-buffer (anti-aliased, area-averaged, accumulation buffer), είναι ένας γενικός κρυφός μηχανισμός, ιδανικός για HY με εικονική μνήμη μέσης κλίμακας. Επιλύει το θέμα της ορατότητας μεταξύ αυθαίρετα τοποθετημένων αντικειμένων οποιασδήποτε διαφάνειας, από μηδενική έως πλήρους, καθώς και διατεμνόμενων. Ο αλγόριθμος A-buffer βελτιώνει την απόδοση της τελικής ποιότητας της εικόνας σε σχέση με την μέθοδο Z-buffer. Αυτό συμβαίνει επειδή επεκτείνει την προκαθορισμένη μέθοδο Z-buffer, στην οποία αποθηκεύεται μόνο το θραύσμα που βρίσκεται εγγύτερα στον θεατή, και αποτελεί την πρώτη μέθοδο που αιχμαλωτίζει όλα τα θραύσματα ανά pixel σε ευθεία κάθετη στο οπτικό πεδίο γραμμή. Τα εν λόγω θραύσματα αποθηκεύονται σε λίστες συνδεδεμένες ανά pixel κατά τη διάρκεια της απόδοσης γεωμετρίας του μοντέλου. Στη συνέχεια λαμβάνει χώρα διαδικασία, μετά την ταξινόμηση των θραυσμάτων, η οποία τα αναδιοργανώνει σύμφωνα με το βάθος τους [Carpenter L., '84].

Ο αλγόριθμος A-buffer χρησιμοποιεί δύο τύπους δεδομένων για να επιτύχει το επιθυμητό αποτέλεσμα. Η μια δομή περιέχει τιμή βάθους Z και τιμή χρώματος και η δεύτερη περιέχει θραύσματα. Το θραύσμα είναι ένα πολύγωνο από τη γεωμετρία της σκηνής το οποίο έχει χωριστεί κατά τη λειτουργία rasterization στην σωλήνωση του rendering και αντιστοιχιστεί στα όρια ενός pixel της οθόνης. Ο αλγόριθμος τα δέχεται ως είσοδο σε λίστα η οποία έχει το μέγεθος της τελικής σκηνής. Καθώς η λίστα περιέχει όλα τα στοιχεία της σκηνής, γίνεται η ταξινόμησή της σύμφωνα με το βάθος Z κάθε θραύσματος.

Η ταξινόμηση σύμφωνα με την τιμή Z είναι απαραίτητη για τον υπολογισμό διαφάνειας της σκηνής. Ο υπολογισμός διαφάνειας αντικειμένων απαιτεί κάθε ορατή και διαφανή επιφάνεια να είναι ταξινομημένη με αυτόν τον τρόπο ώστε να γίνει σωστός υπολογισμός του τελικού χρώματος του κάθε pixel.

3.1.2 K-buffer

Ο αλγόριθμος K-buffer αφορά ένα συγκεκριμένο αριθμό θραυσμάτων ανά pixel που αποθηκεύονται στη μνήμη της κάρτας γραφικών. Με τη χρήση σύγχρονης τεχνολογίας γραφικών έχει αποδειχθεί αποτελεσματικός στην ταξινόμηση και σύνθεση θραυσμάτων στις ειδικές περιπτώσεις direct volume rendering. Ωστόσο πολλές εφαρμογές πέραν του direct volume rendering απαιτούν πρόσβαση σε πολλά θραύσματα ταυτόχρονα. Το πλεονέκτημα του K-buffer έγκειται στο ότι επιτρέπει τη σύγκριση, ταξινόμηση, μίξη και απόρριψη των θραυσμάτων σε συγκεκριμένη ροή. Τοιουτοτρόπως, όλα τα εφέ τα οποία κανονικά απαιτούν πολλαπλά περάσματα της γεωμετρίας της σκηνής με τον K-buffer μπορούν πλέον να πραγματοποιηθούν με ένα μόνο πέρασμα[Callahan, '05].

Ο πρωτότυπος A-buffer, όπως προαναφέρθηκε, αποθηκεύει όλα τα επερχόμενα θραύσματα σε μια λίστα η οποία όμως απαιτεί μια εν δυνάμει άπειρου μεγέθους μνήμη. Ο τρόπος προσέγγισης του K-buffer αφορά στην αποθήκευση μόνο ενός συγκεκριμένου αριθμού θραυσμάτων, και λειτουργεί με το συνδυασμό των υπάρχοντων θραυσμάτων και την απόρριψη μέρους αυτών με την έλευση νέων. Η τεχνική αυτή μειώνει τις απαιτήσεις μνήμης και είναι αρκετά απλή ώστε να ενσωματωθεί σε ήδη υπάρχουσες αρχιτεκτονικές γραφικών. Ο K-buffer είναι ουσιαστικά ένας ταξινομητής ροής θραυσμάτων, ο οποίος λειτουργεί ως εξής. Για κάθε pixel, ο K-buffer αποθηκεύει μια σταθερά K αριθμού θραυσμάτων. Κάθε καταχώρηση περιέχει την απόσταση του θραύσματος από το σημείο θέασης, η οποία χρησιμοποιείται σε κατ' αύξηση ταξινόμηση των θραυσμάτων για σύνθεση από μπροστά προς τα πίσω, και κατ' ελάττωση ταξινόμηση για σύνθεση από πίσω προς τα μπροστά. Για την περίπτωση σύνθεσης προς τα πίσω κάθε φορά που εμφανίζεται ένα νέο θραύσμα στον K-buffer, αποκολλά την πρώτη καταχώρηση. Είναι σημαντικό να σημειωθεί πως τα θραύσματα ορίων πρέπει να χειρισθούν σωστά, και μπορούν να εισαχθούν είτε στην αρχή του buffer εάν είναι εγγύτερο, ή στο τέλος εάν βρίσκεται σε μεγαλύτερο βάθος[Callahan, '05].

Μια βασική ιδιότητα του K-buffer αποτελεί το γεγονός πως με την εμφάνιση μιας σειράς θραυσμάτων τέτοια, στην οποία κάθε θραύσμα βρίσκεται σε μεταξύ K θέσεων από τη θέση του στην ταξινομημένη σειρά, θα εμφανίσει τα θραύσματα σε

σωστή σειρά. Με αυτόν τον τρόπο, με ένα μικρό K , ο K -buffer μπορεί να χρησιμοποιηθεί για την ταξινόμηση μιας σχεδόν ταξινομημένης ακολουθίας n θραυσμάτων σε χρόνο $O(n)$. Πρέπει να σημειωθεί πως για τη σύνθεση της προαναφερόμενης ακολουθίας είναι απαραίτητη η ύπαρξη $K+1$ θραυσμάτων.

3.2 Υλοποίηση Αλγορίθμων

3.2.1 LWJGL

Η βασική δομή της υλοποίησης του πειράματος ξεκινάει από την αρχικοποίηση των ρυθμίσεων για την δημιουργία τρισδιάστατης σκηνής. Αυτό συμπεριλαμβάνει τις ρυθμίσεις του παραθύρου, αλλά πιο σημαντικά την απενεργοποίηση κάποιων προεπιλογών της OpenGL. Συγκεκριμένα πρέπει να γίνει απενεργοποίηση του `GL_BLEND`, το οποίο είναι υπεύθυνο για την τελική μίξη των χρωμάτων. Επίσης πρέπει να απενεργοποιηθεί το `GL_CULL_FACE` το οποίο απορρίπτει θραύσματα τα οποία αναφέρονται στο πίσω μέρος μιας επιφάνειας. Τέλος πρέπει να απενεργοποιηθεί και το `GL_DEPTH_TEST` το οποίο είναι ο προεπιλεγμένος αλγόριθμος της OpenGL για αναγνώριση βάθους θραύσματος (βλ Σχήμα 3.1).

```
private void setupOpenGL() {
    try {
        Display.setDisplayMode(new DisplayMode(WIDTH, HEIGHT));
        Display.setTitle(WINDOW_TITLE);
        Display.create();
    } catch (LWJGLException e) {
        e.printStackTrace();
        System.exit(-1);
    }

    GL11.glClearColor(0.4f, 0.6f, 0.9f, 0f);
    GL11.glViewport(0, 0, WIDTH, HEIGHT);

    glDisable(GL_BLEND);
    glDisable(GL_CULL_FACE);
    glDisable(GL_DEPTH_TEST);
    glDisable(GL_STENCIL_TEST);
    this.exitOnGLError("setupOpenGL");
}
```

Σχήμα 3.1: Αρχικοποίηση OpenGL

Στην συνέχεια, γίνεται ο ορισμός και αρχικοποίηση των βασικών πινάκων της OpenGL, projectionMatrix, viewMatrix και modelMatrix(βλ. Σχήμα 3.2).

```
private void setupMatrices() {
    projectionMatrix = new Matrix4f();
    float fieldOfView = 60f;
    float aspectRatio = (float) WIDTH / (float) HEIGHT;
    float near_plane = 0.1f;
    float far_plane = 100f;

    float y_scale = this.coTangent(this.degreesToRadians(fieldOfView / 2f));
    float x_scale = y_scale / aspectRatio;
    float frustum_length = far_plane - near_plane;

    projectionMatrix.m00 = x_scale;
    projectionMatrix.m11 = y_scale;
    projectionMatrix.m22 = -((far_plane + near_plane) / frustum_length);
    projectionMatrix.m23 = -1;
    projectionMatrix.m32 = -((2 * near_plane * far_plane) / frustum_length);
    projectionMatrix.m33 = 0;

    viewMatrix = new Matrix4f();
    modelMatrix = new Matrix4f();

    matrix44Buffer = BufferUtils.createFloatBuffer(16);
}
```

Σχήμα 3.2: Αρχικοποίηση πινάκων

Το επόμενο βήμα είναι η εισαγωγή των μοντέλων. Τα μοντέλα είναι τύπου .obj των οποίων η πληροφορία είναι αποθηκευμένη ως συντεταγμένες vertices, faces και normals. Γίνεται parsing του αρχείου και αποθηκεύονται οι πληροφορίες σε τρεις buffers έναν για κάθε πληροφορία. Αυτή η τεχνική ονομάζεται vertex buffer object. Με αυτήν την μέθοδο η εισαγωγή της πληροφορίας γίνεται μόνο μια φορά στη μνήμη και μετά δεν χρειάζεται η επαναφόρτωσή της.

```

private void setupOBJ() {
    model = null;
    try {
        model = OBJLoader.LoadModel(new File(MODEL_LOCATION));
    } catch (FileNotFoundException e) {
        e.printStackTrace();
        System.exit(1);
    } catch (IOException e) {
        e.printStackTrace();
        System.exit(1);
    }
    vboVertexHandle = glGenBuffers();
    FloatBuffer vertices = BufferTools.reserveData(model.getFaces().size() * 9);
    for (Model.Face face : model.getFaces()) {
        vertices.put(BufferTools.asFloats(model.getVertices().get(face.getVertexIndices()[0] - 1)));
        vertices.put(BufferTools.asFloats(model.getVertices().get(face.getVertexIndices()[1] - 1)));
        vertices.put(BufferTools.asFloats(model.getVertices().get(face.getVertexIndices()[2] - 1)));
    }
    vertices.flip();
    glBindBuffer(GL_ARRAY_BUFFER, vboVertexHandle);
    glBufferData(GL_ARRAY_BUFFER, vertices, GL_STATIC_DRAW);
    glVertexAttribPointer(0, 3, GL_FLOAT, false, 0, 0);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    glBindVertexArray(0);

    vboNormalHandle = glGenBuffers();
    FloatBuffer normals = BufferTools.reserveData(model.getFaces().size() * 9);
    for (Model.Face face : model.getFaces()) {
        normals.put(BufferTools.asFloats(model.getNormals().get(face.getNormalIndices()[0] - 1)));
        normals.put(BufferTools.asFloats(model.getNormals().get(face.getNormalIndices()[1] - 1)));
        normals.put(BufferTools.asFloats(model.getNormals().get(face.getNormalIndices()[2] - 1)));
    }
    normals.flip();
    glBindBuffer(GL_ARRAY_BUFFER, vboNormalHandle);
    glBufferData(GL_ARRAY_BUFFER, normals, GL_STATIC_DRAW);
    glVertexAttribPointer(1, 3, GL_FLOAT, false, 0, 0);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    glBindVertexArray(0);
    modelPos = new Vector3f(0, 1.0f, -5.0f);
    modelAngle = new Vector3f(0, 0, 0);
    modelScale = new Vector3f(0.2f, 0.2f, 0.2f);
    cameraPos = new Vector3f(0, -1, -1);

    this.exitOnGLError("setupOBJ");
}

```

Σχήμα 3.3: Εισαγωγή μοντέλου

Για την ενεργοποίηση των shaders χρειάζεται πρώτα να γίνει το compile τους, αφού πρόκειται για ανεξάρτητα προγράμματα. Αυτό επιτυγχάνεται κάνοντας parsing τα σωστά αρχεία και με τις ήδη υπάρχουσες εντολές `glLinkProgram` και `glValidateProgram` γίνεται η σύνδεση και το compile των shaders(βλ. Σχήμα 3.4).


```

private void setupShaders() {
    if (abuffer) {
        progABclear = setupShader("src/NewTest/ABclear.vs", "src/NewTest/ABclear.fs");
        progABrender = setupShader("src/NewTest/ABrender.vs", "src/NewTest/ABrender.fs" );
        progABdisp = setupShader("src/NewTest/ABdisp.vs", "src/NewTest/ABdisp.fs" );
    } else {
        progABclear = setupShader("src/NewTest/Kclear.vs", "src/NewTest/Kclear.fs");
        progABrender = setupShader("src/NewTest/Krender.vs", "src/NewTest/KrenderNew.fs" );
        progABdisp = setupShader("src/NewTest/Kdisp.vs", "src/NewTest/KdispNew.fs" );
    }

    this.exitOnGLError("setupShader");
}

private int setupShader(String VERTEX_SHADER_LOCATION, String FRAGMENT_SHADER_LOCATION) {
    int pId = 0;
    int vsId = this.loadShader(VERTEX_SHADER_LOCATION, GL20.GL_VERTEX_SHADER);
    int fsId = this.loadShader(FRAGMENT_SHADER_LOCATION, GL20.GL_FRAGMENT_SHADER);

    pId = GL20.glCreateProgram();
    GL20.glAttachShader(pId, vsId);
    GL20.glAttachShader(pId, fsId);

    GL20.glLinkProgram(pId);
    GL20.glValidateProgram(pId);

    this.exitOnGLError("setupShaders");

    return (pId);
}

```

Σχήμα 3.4: Αρχικοποίηση shaders.

Το επόμενο βήμα αφορά την δημιουργία των δομών δεδομένων τα οποία θα χρησιμοποιηθούν από τους shaders για να αποθηκευτούν οι απαραίτητες πληροφορίες. Η δέσμευση μνήμης για τους αλγόριθμους γίνεται σε μορφή `texture2Darray` και `texture2D`. Οι δύο δομές αποθηκεύουν την απαραίτητη πληροφορία που χρειάζεται ο αλγόριθμος για να λειτουργήσει. Η `texture2Darray` είναι μια τρισδιάστατη δομή αποθήκευσης διαστάσεων όσο και η ανάλυση της οθόνης και βάθους όσο το μέγεθος της μνήμης του υπολογιστή που είναι διαθέσιμη. Ο τύπος δεδομένων που αποθηκεύει είναι `GL_RGBA32F`, το οποίο αποθηκεύει τέσσερις floating point τιμές μεγέθους 32bit. Οι τρεις πρώτες είναι τιμές χρώματος διάταξης RGB και η τελευταία θα χρησιμοποιηθεί για αποθήκευση του βάθους. Η `texture2D` είναι της ίδιας μορφής με την προαναφερθείσα, με την διαφορά να είναι στο ότι είναι διδιάστατη. Ο τύπος δεδομένων που αποθηκεύει είναι `GL_R32UI`, το οποίο αποθηκεύει μια τιμή τύπου unsigned integer, επίσης μεγέθους 32bit, η οποία θα χρησιμοποιηθεί για την αρίθμηση και την αποθήκευση του τελικού αριθμού των

θραυσμάτων. Οι δύο δομές έχουν διαστάσεις ίσες με την ανάλυση της οθόνης, για να υπάρχει αντιστοιχία των θέσεων των θραυσμάτων με τα pixel της οθόνης.

```
private void setupbuffer() {
    if (abufferTexID == 0)
        abufferTexID = glGenTextures();
    glActiveTexture(GL_TEXTURE0 + 0);
    glBindTexture(GL_TEXTURE_2D_ARRAY, abufferTexID);
    glTexParameteri(GL_TEXTURE_2D_ARRAY, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D_ARRAY, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexImage3D(GL_TEXTURE_2D_ARRAY, 0, GL_RGBA32F, WIDTH, HEIGHT, BUFFER_SIZE, 0, GL_RGBA, GL_FLOAT, (ByteBuffer) null);
    glBindImageTexture(0, abufferTexID, 0, true, 0, GL_READ_WRITE, GL_RGBA32F);

    if (abufferCounterTexID == 0)
        abufferCounterTexID = glGenTextures();
    glActiveTexture(GL_TEXTURE1 + 1);
    glBindTexture(GL_TEXTURE_2D, abufferCounterTexID);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_R32F, WIDTH, HEIGHT, 0, GL_RED, GL_FLOAT, (ByteBuffer) null);
    glBindImageTexture(1, abufferCounterTexID, 0, false, 0, GL_READ_WRITE, GL_R32UI);

    exitOnGLError("bufferTex");
}
```

Σχήμα 3.5: Αρχικοποίηση μνήμης

Η OpenGL ανανεώνει και κάνει render την σκηνή σε κάθε frame. Επομένως πρέπει σε κάθε frame να κληθούν οι shaders. Πρώτα γίνεται το πέρασμα των δομών στους shaders με την εντολή `glProgramUniform1i()`. Η κλήση των shaders γίνεται με την εντολή `glUseProgram()` και αμέσως μετά αποστέλλεται η εντολή για να γίνει το render του μοντέλου, όπου και καλούνται τα vertex buffer objects τα οποία προαναφέραμε(βλ. Σχήμα 3.6).

```
private void renderCycle() {
    GL11.glClear(GL11.GL_COLOR_BUFFER_BIT);

    glProgramUniform1i(progABclear, glGetUniformLocation(progABclear, "abufferImg"), 0);
    glProgramUniform1i(progABclear, glGetUniformLocation(progABclear, "abufferCounterImg"), 1);
    renderMesh(progABclear);

    glProgramUniform1i(progABrender, glGetUniformLocation(progABrender, "abufferImg"), 0);
    glProgramUniform1i(progABrender, glGetUniformLocation(progABrender, "abufferCounterImg"), 1);
    renderMesh(progABrender);

    glProgramUniform1i(progABdisp, glGetUniformLocation(progABdisp, "abufferImg"), 0);
    glProgramUniform1i(progABdisp, glGetUniformLocation(progABdisp, "abufferCounterImg"), 1);
    renderMesh(progABdisp);

    GL15.glBindBuffer(GL15.GL_ELEMENT_ARRAY_BUFFER, 0);
    GL20.glDisableVertexAttribArray(0);
    GL20.glDisableVertexAttribArray(1);
    GL30.glBindVertexArray(0);
    GL20.glUseProgram(0);

    this.exitOnGLError("renderCycle");
}
```

Σχήμα 3.6: Κλήση shaders

Οι shaders για να αποθηκεύσουν όλη την απαραίτητη πληροφορία κάνουν δυο “περάσματα” της σκηνής. Κάθε θραύσμα απορρίπτεται για να μην υπάρχει παρεμβολή στο τελικό αποτέλεσμα. Τέλος, κάθε επεξεργασία αφορά μόνο θραύσματα τα οποία βρίσκονται σε ορατό σημείο.

Στο πρώτο γίνεται ο καθαρισμός της μνήμης ώστε να μην περιέχονται σκουπίδια χρησιμοποιώντας την εντολή `imageStore()` η οποία αναθέτει την επιθυμητή τιμή στην αντίστοιχη θέση του θραύσματος (βλ. σχήμα 3.7).

```
void main(void) {
    ivec2 coords=ivec2(gl_FragCoord.xy);
    if(coords.x>=0 && coords.y>0 && coords.x<640 && coords.y<480) {

        imageStore(abufferCounterImg, coords, ivec4(0.0f,0.0f,0.0f,1.0f));
        imageStore(abufferImg, ivec3(coords, 0), vec4(0.0f,0.0f,0.0f,0.0f));
    }
    discard;
}
```

Σχήμα 3.7: Καθαρισμός μνήμης

Στο δεύτερο γίνεται η αποθήκευση της πληροφορίας όλων των θραυσμάτων στις δυο βασικές δομές που αναφέρθηκαν νωρίτερα, με την μόνη διαφορά ότι ο A-buffer αποθηκεύει όλα τα θραύσματα, ενώ ο K-buffer αποθηκεύει ένα K πλήθος, προκαθορισμένο από τον προγραμματιστή. Αρχικά γίνεται χρήση της εντολής `imageAtomicIncWrap()`. Η εντολή αυτή επιτρέπει την αποθήκευση στην `texture2D` έναν αύξοντα αριθμό για κάθε θραύσμα που εισέρχεται στον fragment shader. Με αυτόν τον τρόπο στο τέλος του “περάσματος” θα έχει αποθηκευτεί το πλήθος των θραυσμάτων που αντιστοιχούν στο εκάστοτε pixel. Επίσης, κατά την διάρκεια, ο αύξων αριθμός υποδεικνύει και το επίπεδο της `texture2Darray` στο οποίο θα αποθηκευτεί η τιμή χρώματος του θραύσματος, το οποίο επιτυγχάνεται με την εντολή `ImageStore`. Με αυτόν τον τρόπο η σκηνή έχει χωριστεί σε επίπεδα, η πληροφορία των θραυσμάτων έχει αποθηκευτεί για κάθε επίπεδο και το μόνο που απομένει είναι η ταξινόμηση τους και η τελική απόδοση της σκηνής. Το τρίτο βήμα διαφέρει για τους δυο αλγόριθμους (βλ. Σχήμα 3.8).

```

void main(void) {
    ivec2 coords=ivec2(gl_FragCoord.xy);
    uint bidx;
    if(coords.x>=0 && coords.y>=0 && coords.x<SCREEN_WIDTH && coords.y<SCREEN_HEIGHT){

        bidx=imageAtomicIncWrap(abufferCounterImg, coords, K );
        vec4 abuffval;
        vec3 col;
        col=shadeStrips(fragTexCoord);

        buffval.rgb=col;
        buffval.w=fragPos.z;    //Will be used for sorting

        imageStore(abufferImg, ivec3(coords, bidx), buffval);

    }
}

```

Σχήμα 3.8: Αποθήκευση θραυσμάτων

3.2.2 A-buffer

Στο το τρίτο βήμα του αλγόριθμου A-buffer, γίνεται η ταξινόμηση των θραυσμάτων και η μίξη των χρωμάτων σύμφωνα με την σειρά της ταξινόμησης και το τελικό αποτέλεσμα αποδίδεται στο θραύσμα. Αρχικά χρησιμοποιώντας την εντολή `imageLoad()` στην `texture2D` λαμβάνουμε το πλήθος των θραυσμάτων που αφορούν την θέση σε pixel όπου βρίσκεται το αντίστοιχο θραύσμα. Στη συνέχεια δημιουργούμε μια λίστα στην οποία αποθηκεύουμε την πληροφορία χρώματος και βάθους για όλα τα θραύσματα που βρίσκονται στην αντίστοιχη θέση σε pixel από την `image2Darray` χρησιμοποιώντας την εντολή `imageLoad()`. Ύστερα γίνεται ταξινόμηση της λίστας αυτής χρησιμοποιώντας τον αλγόριθμο `bubble sort`. Για τον τελικό υπολογισμό του χρώματος γίνεται προσπέλαση της λίστας ξεκινώντας από το εγγύτερο (βλ. Σχήμα 3.9).

```

void main(void) {
    ivec2 coords=ivec2(gl_FragCoord.xy);
    int abNumFrag;
    if(coords.x>=0 && coords.y>=0 && coords.x<SCREEN_WIDTH && coords.y<SCREEN_HEIGHT ){

        abNumFrag=int(imageLoad(abufferCounterImg, coords).r);
        fillFragmentArray(coords, abNumFrag);
        bubbleSort(abNumFrag);

        vec4 finalColor=vec4(0.0f);
        const float sigma = 30.0f;
        float thickness=fragmentList[0].w/2.0f;
        finalColor=vec4(0.0f);
        int depthc = BUFFER_SIZE;
        if (abNumFrag < BUFFER_SIZE) {
            depthc = abNumFrag;
        }
        for(int i=0; i<depthc; i++){
            vec4 frag=fragmentList[i];
            vec4 col;
            col.rgb=frag.rgb;
            col.w=fragmentAlpha;

            col.rgb=col.rgb*col.w;

            finalColor=finalColor+col*(1.0f-finalColor.a);
        }
        finalColor=finalColor+backgroundColor*(1.0f-finalColor.a);
        outFragColor = finalColor;
    }
}

```

Σχήμα 3.9: Αλγόριθμος A-buffer

3.2.3. K-buffer

Στο τρίτο βήμα του αλγόριθμου K-buffer, γίνεται η επιλογή των επιθυμητών χρησιμοποιώντας την εντολή `imageLoad()` στην `texture2D`, λαμβάνουμε το πλήθος των θραυσμάτων που αφορούν την θέση σε pixel όπου βρίσκεται το αντίστοιχο θραύσμα. Αν το πλήθος είναι μικρότερο του K τότε αποθηκεύουμε την πληροφορία χρώματος και βάθους σε μια λίστα την οποία ταξινομούμε με bubble sort και γίνεται η μίξη των χρωμάτων για την παραγωγή του τελικού χρώματος. Σε περίπτωση που το πλήθος των θραυσμάτων είναι μεγαλύτερο από το K πρέπει να γίνει η σωστή επιλογή και απόρριψη των εισερχόμενων θραυσμάτων. Χρησιμοποιώντας την εντολή `imageLoad()` στην `texture2Darray` υπάρχει πρόσβαση στα δεδομένα κάθε αποθηκευμένου θραύσματος. Αρχικά βρίσκουμε το θραύσμα που βρίσκεται

μακρύτερα. Στο επόμενο βήμα παίρνουμε το νέο θραύσμα και αν είναι πιο κοντινό από το μέγιστο τότε αντικαθιστούμε το μέγιστο με το νέο στην λίστα, αλλιώς το θέτουμε στην τελευταία θέση της λίστας(βλ. σχήμα 3.10).

```
void main(void) {

    ivec2 coords=ivec2(gl_FragCoord.xy);
    int abNumFrag;
    int count = K;
    if(coords.x>=0 && coords.y>=0 && coords.x<SCREEN_WIDTH && coords.y<SCREEN_HEIGHT ){
        abNumFrag=int(imageLoad(abufferCounterImg, coords).r);

        if(abNumFrag > 0) {
            if (abNumFrag < K) {
                count = abNumFrag;
            }
            vec2 frmax = vec2(-1.0f,0.0f);
            for (int i=0; i<count-1; i++) {
                fragmentList[i] = imageLoad(abufferImg,ivec3(coords,i));
                if (frmax.g < fragmentList[i].w) {
                    frmax = getMax();
                }
            }
            vec2 frnew = vec2(count-1, fragPos.z);
            vec4 abuffval;
            vec3 col;
            col=shadeStrips(fragTexCoord);

            abuffval.rgb=col;
            abuffval.w=fragPos.z;
            if (frnew.g < frmax.g) {
                fragmentList[count-1] = imageLoad(abufferImg,ivec3(coords,frmax.r));
                fragmentList[int(frmax.r)] = abuffval;
            } else {
                fragmentList[count-1] = abuffval;
            }
            bubbleSort(count);
            vec4 finalColor=vec4(0.0f);
            const float sigma = 30.0f;
            float thickness=fragmentList[0].w/2.0f;
            finalColor=vec4(0.0f);

            for(int i=0; i<count-1; i++){
                vec4 frag=fragmentList[i];
                vec4 col;
                col.rgb=frag.rgb;
                col.w=fragmentAlpha;
                col.rgb=col.rgb*col.w;

                finalColor=finalColor+col*(1.0f-finalColor.a);
            }
            finalColor=finalColor+backgroundColor*(1.0f-finalColor.a);
            outFragColor = finalColor;
        } else {
            discard;
        }
    }
}
```

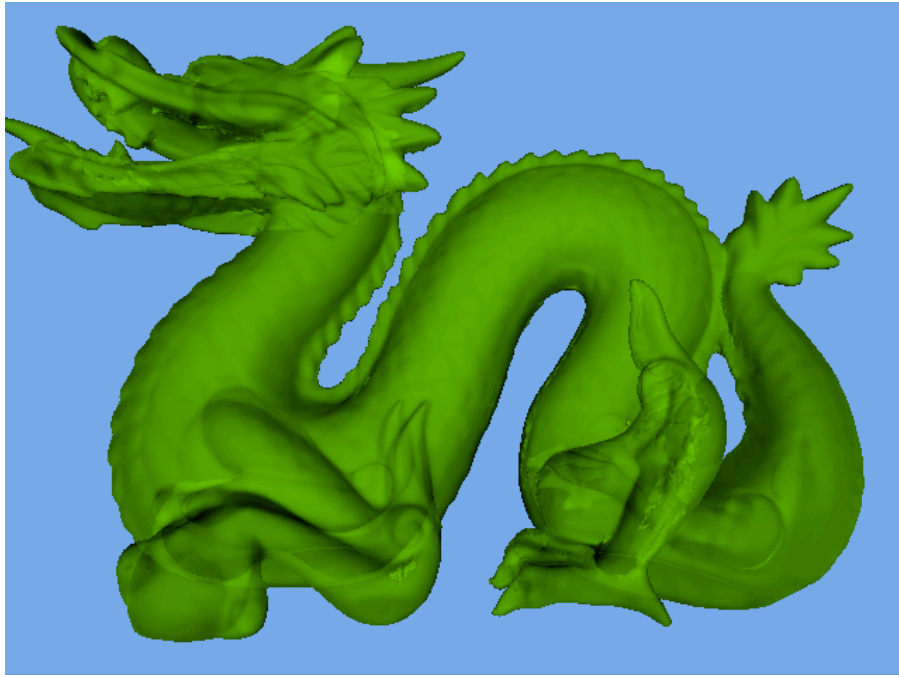
Σχήμα 3.10: Αλγόριθμος K-buffer

3.3 Πειραματική αποτίμηση

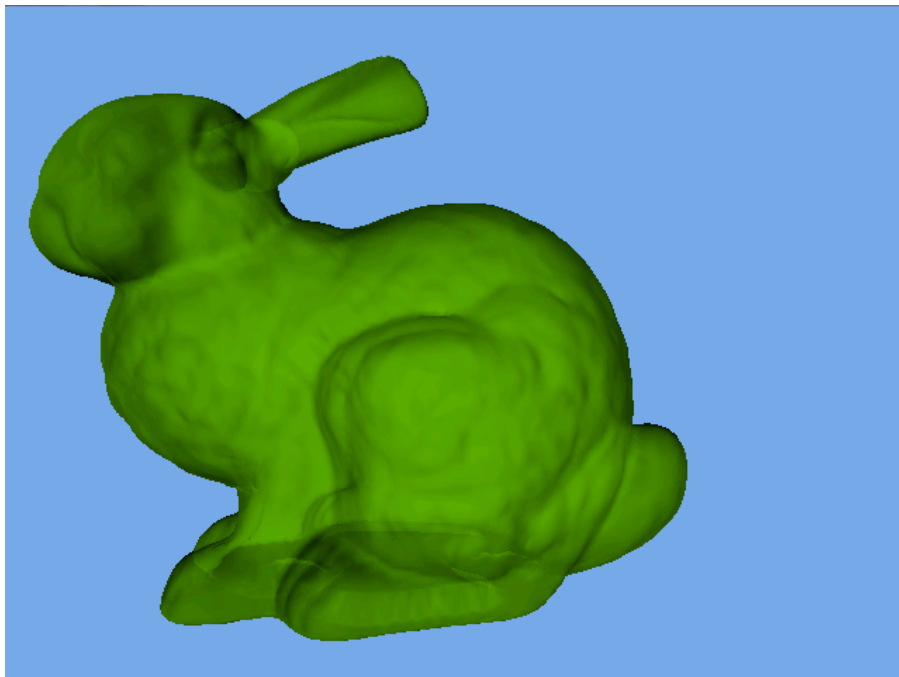
Το πείραμα που διεξήχθη, αφορά τη διερεύνηση των αλγορίθμων όσον αφορά την υλοποίησή τους στο περιβάλλον LWJGL και εν συνεχεία τη σύγκριση των αποτελεσμάτων μεταξύ τους. Ο υπολογιστής στον οποίο έγινε η υλοποίηση και η διεξαγωγή του πειράματος έχει χαρακτηριστικά CPU AMD FX-8320, RAM 32GB και GPU GeForce 760 4GB. Για την σκηνή, χρησιμοποιήθηκαν τρία μοντέλα (εκτενής εικόνα κτλ). Τα δυο μοντέλα είναι τα γνωστά στον χώρο των γραφικών Stanford Dragon και Stanford Bunny. Το μοντέλο που χρησιμοποιήθηκε για τον Stanford Dragon αποτελείται από 50.000 vertices και 199.999 τρίγωνα. Το Stanford Bunny αποτελείται από 34.818 vertices και 69.900 τρίγωνα. Τέλος χρησιμοποιήθηκε και ένα μοντέλο εδάφους το οποίο αποτελεί σημαντικό κομμάτι του στερεοσκοπικού βίντεο, του δεύτερου δηλαδή σκέλους της εργασίας. Αποτελείται από 117.051 vertices και 94.346 τρίγωνα.

3.3.1 A-buffer

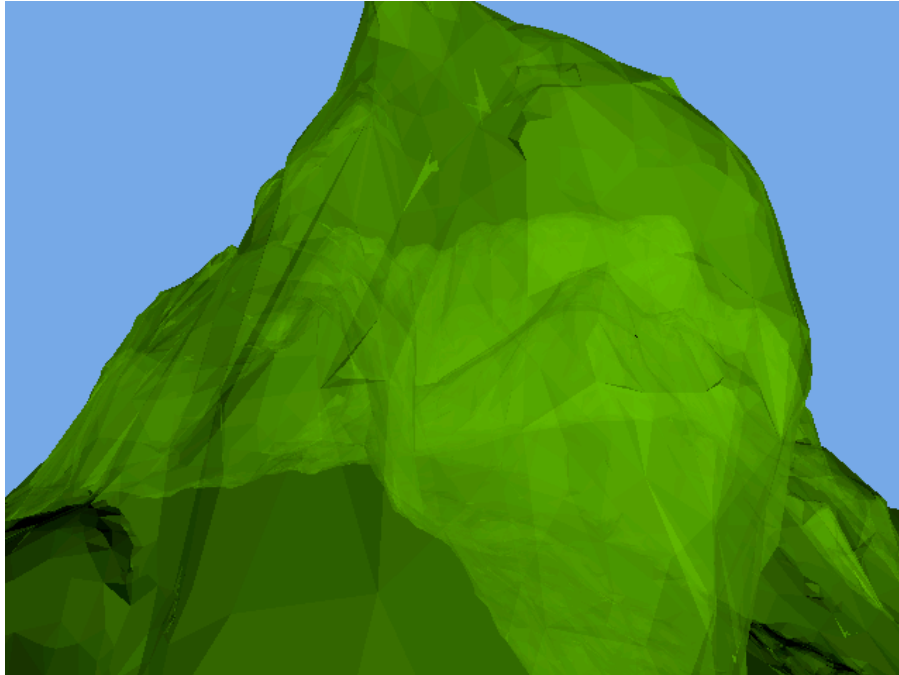
Ο A-buffer όπως προαναφέρθηκε χρειάζεται υψηλή χωρητικότητα μνήμης για να αποθηκεύσει όλα τα θραύσματα της σκηνής. Η υλοποίηση είναι εμπνευσμένη από τον [Crassin, '10]. Στην προκειμένη περίπτωση επιλέχθηκε η τιμή βάθους 16 καθώς η πολυπλοκότητα βάθους της σκηνής μας δεν ξεπερνάει το 10. Στους παρακάτω πίνακες βλέπουμε τα frames ανά δευτερόλεπτο που παράγει ο υπολογιστής και την ποσότητα μνήμης που δεσμεύτηκε για να αποθηκευτούν τα θραύσματα. Επίσης παρατίθενται εικόνες για την παρουσίαση του οπτικού αποτελέσματος(βλ. Σχήμα 3.11, 3.12, 3.13).



Σχήμα 3.11: Stanford dragon



Σχήμα 3.12: Stanford bunny



Σχήμα 3.13: Terrain

Όπως παρατηρείται από τις παραπάνω εικόνες, υπάρχει πολύ καλή απόδοση της διαφάνειας, και είναι ιδιαίτερα εμφανές στο μοντέλο Terrain (βλ. Σχήμα 3.13) όπου είναι μεγάλος ο αριθμός επιπέδων αλλά η διαφάνεια έχει αποδοθεί πλήρως. Η καλή απόδοση της διαφάνειας πραγματοποιείται με μεγάλο κόστος μνήμης και frames ανά δευτερόλεπτο(βλ. Πίνακα 3.1).

	Dragon	Bunny	Terrain
fps	232	538	598
Memory(MB)	233	231	232

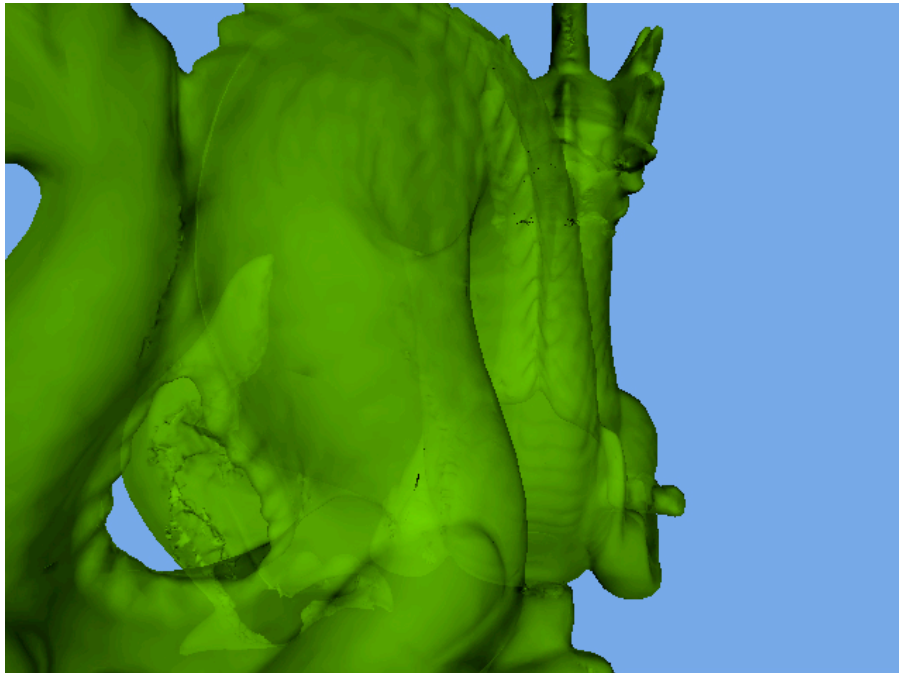
Πίνακας 3.1: A-buffer bechmarks

3.3.2 *K-buffer*

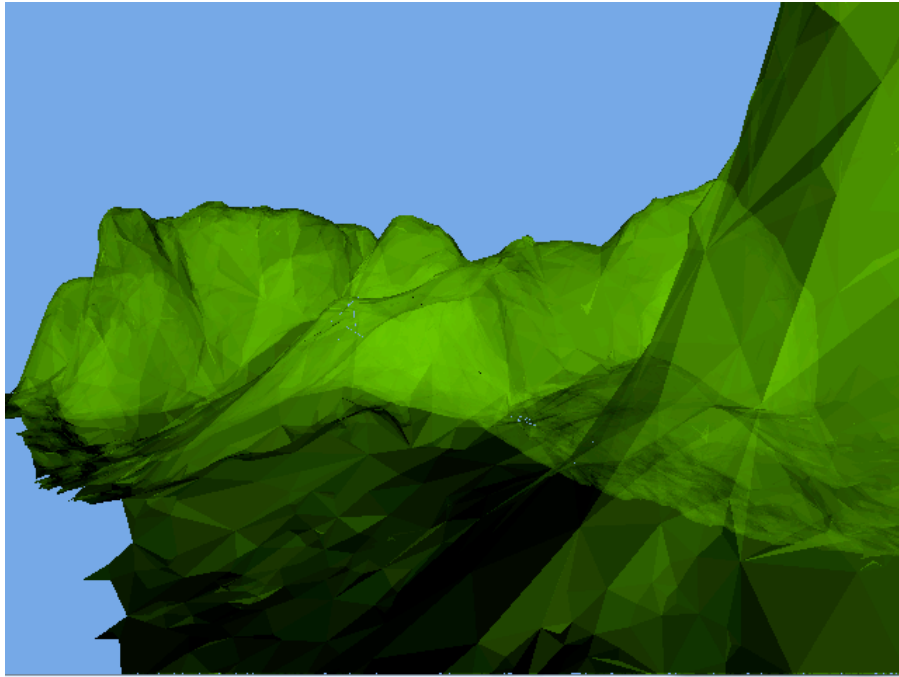
Ο K-buffer όπως προαναφέρθηκε χρειάζεται προκαθορισμένη χωρητικότητα μνήμης για να αποθηκεύσει τα K θραύσματα της σκηνής. Στην προκειμένη περίπτωση

επιλέχθηκαν διάφορες τιμές για να ελεγχθεί η αποδοτικότητά του σε σχέση με το οπτικό αποτέλεσμα. οι τιμές του K είναι 2, 5 και 10. Οι τιμές επιλέχθηκαν έτσι ώστε η διαφορά στο οπτικό αποτέλεσμα μεταξύ τους να είναι εμφανής. Στους παρακάτω πίνακες βλέπουμε τα frames ανά δευτερόλεπτο που παράγει ο υπολογιστής και την ποσότητα μνήμης που δεσμεύτηκε για να αποθηκευτούν τα θραύσματα. Ακολουθούν αναλυτικά οι τρεις περιπτώσεις, καθώς και σχήματα στα οποία τα μοντέλα έχουν τοποθετηθεί στρατηγικά ώστε η απόδοση της διαφάνειας να είναι εμφανής.

Για $K=10$ παρατηρούμε ότι έχει αποδοθεί η διαφάνεια αποτελεσματικά επειδή η σκηνή μας δεν έχει πολυπλοκότητα βάθους μεγαλύτερη του 10(βλ. Σχήμα 3.14, Σχήμα 3.15).

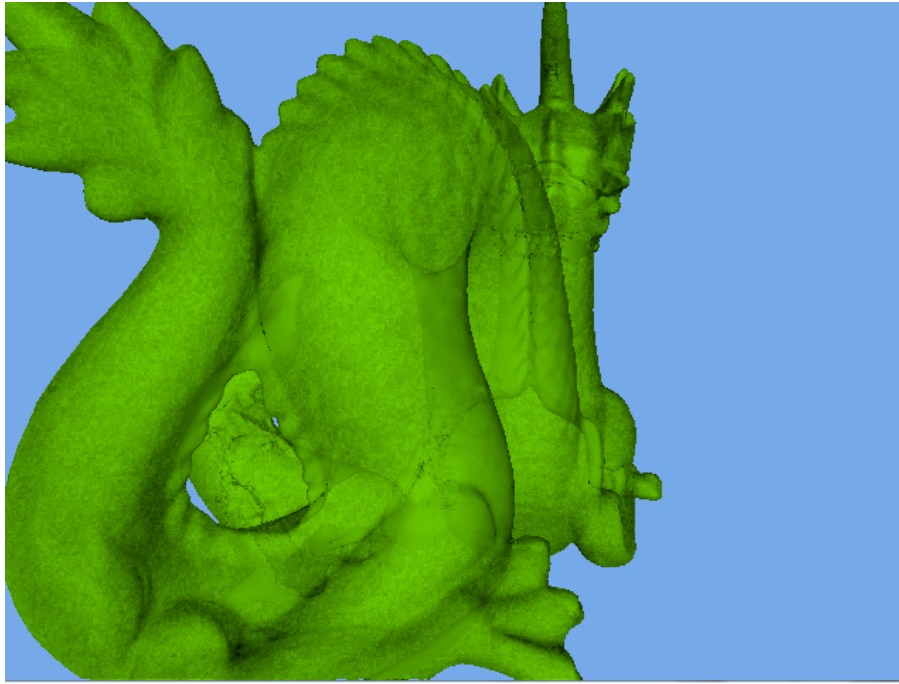


Σχήμα 3.14: Stanford Dragon, $K=10$



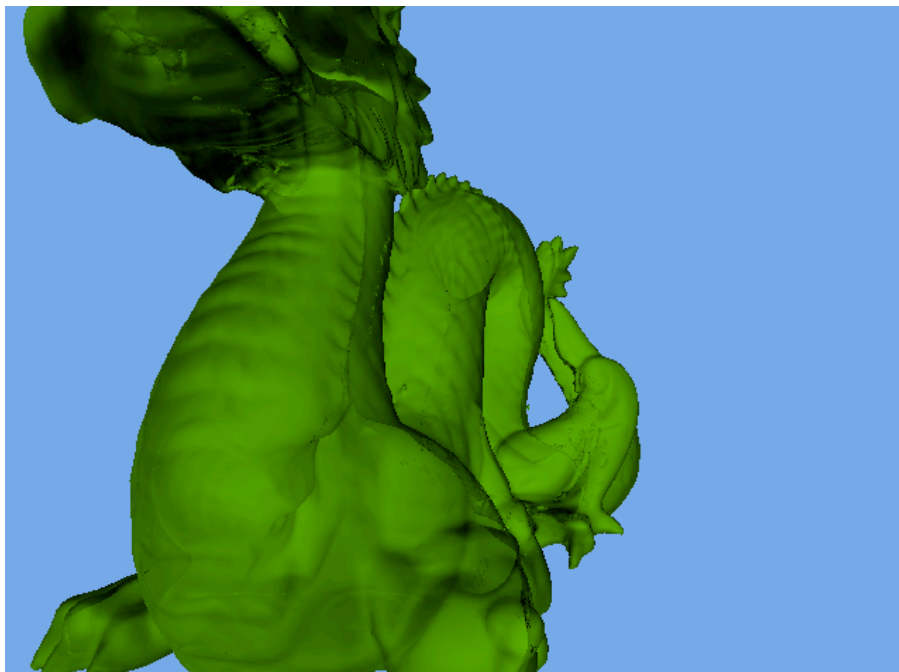
Σχήμα 3.15: Terrain, $K=10$

Για $K=5$ παρατηρείται μείωση της απόδοσης της διαφάνειας σε σχέση με τον $K=10$ το οποίο είναι αναμενόμενο, καθώς αποθηκεύεται μικρότερος αριθμός θραυσμάτων ανά pixel. Παρατηρείται επίσης μια πληθώρα artifacts, τα οποία εμφανίζονται λόγω της παράλληλης επεξεργασίας των θραυσμάτων σε συνδυασμό με την έλλειψη μηχανισμού spin lock [Vassilakis & Fudos, '14] η οποία προκαλεί λάθη στον υπολογισμό των χαρακτηριστικών του θραύσματος(βλ. Σχήμα 3.16).



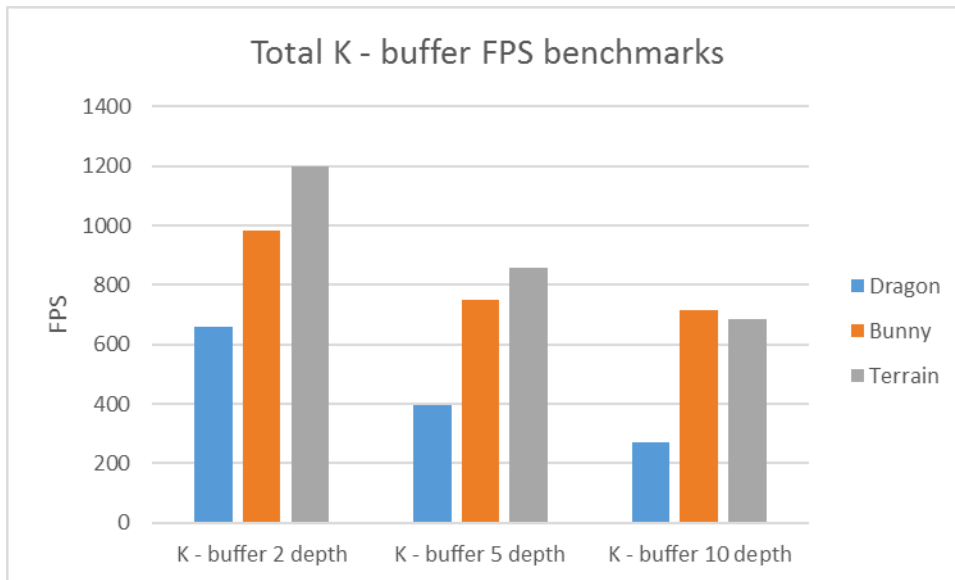
Σχήμα 3.16: Stanford dragon, $K=5$

Για $K=2$ παρατηρούμε πως υπάρχει μερική διαφάνεια, το οποίο είναι αναμενόμενο καθώς αποθηκεύονται μόνο τα δύο πρώτα επίπεδα θραυσμάτων(βλ. Σχήμα 3.17) .

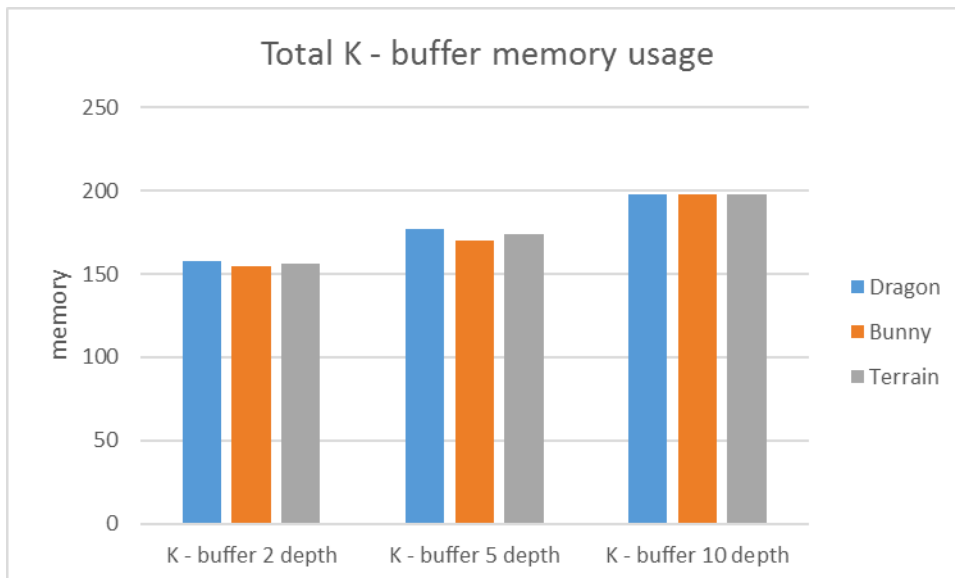


Σχήμα 3.17: Stanford dragon, $K=2$

Όπως παρατηρείται από τις παραπάνω εικόνες, υπάρχει απόδοση της διαφάνειας ανάλογη με την τιμή του K , όπως φαίνεται από την διαφορά κάθε περίπτωσης. Όπως και στον A-buffer η καλή απόδοση διαφάνειας πραγματοποιείται με κόστος μνήμης και frames ανά δευτερόλεπτο. Όσο μειώνεται η τιμή του K , δηλαδή ο αριθμός επιπέδων, αντίστοιχα μειώνεται η απαίτηση πόρων, αλλά και η ποιότητα απόδοσης. (βλ. Γράφημα 3.1,3.2).



Γράφημα 3.1: K-buffer FPS benchmarks



Γράφημα 3.2: K-buffer memory benchmarks

3.3.3 Σύγκριση και αποτίμηση αποτελεσμάτων

	A- buffer	K-buffer depth=2	K-buffer depth=5	K-buffer depth=10
Dragon	232	658	397	269
Bunny	538	983	749	716
Terrain	598	1200	856	685

Πίνακας 3.2 : Τιμές FPS A-buffer και K-buffer

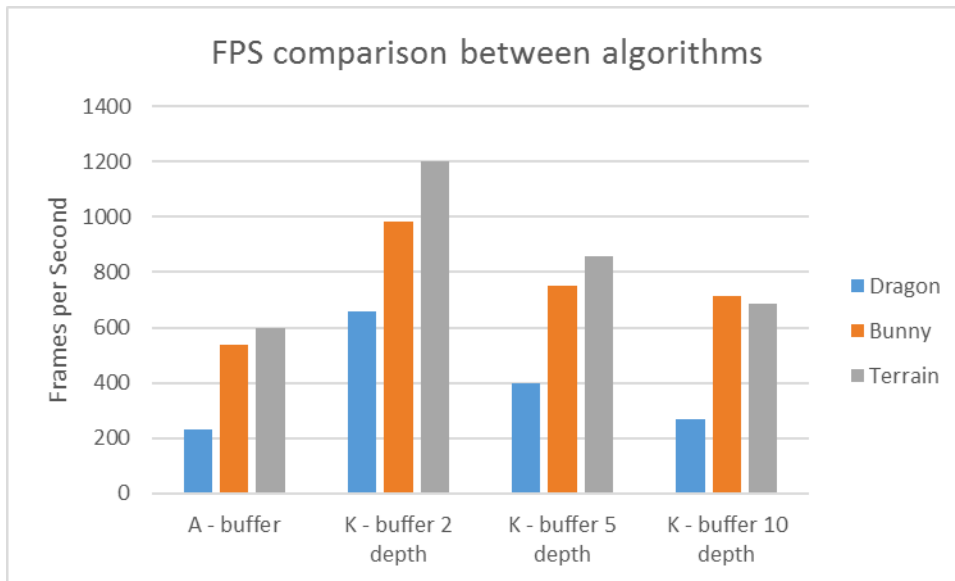
Όπως φαίνεται και από τον συνολικό πίνακα τιμών FPS που παρατίθεται παραπάνω, το πρώτο στοιχείο που είναι εμφανές είναι οι υψηλές τιμές FPS του K-buffer σε σχέση με εκείνες του A-buffer. Ειδικότερα, στην περίπτωση του Stanford dragon που έχει αποδοθεί με K-buffer με τιμή βάθους 2 η τιμή των frames ανά δευτερόλεπτο είναι περισσότερο από διπλάσια αυτής που αποδίδει ο A-buffer. Το ίδιο παρατηρείται μεταξύ του A-buffer και K-buffer με βάθος 2 και στα άλλα δύο μοντέλα, Stanford Dragon και εδάφους. Καθώς αυξάνεται η τιμή πολυπλοκότητας βάθους στον αλγόριθμο K-buffer, παρατηρείται μία σταδιακή μείωση στα frames ανά δευτερόλεπτο, με το μεγαλύτερο βάθος να πλησιάζει τις τιμές του A-buffer σε όλα τα μοντέλα. Όσον αφορά, λοιπόν, τη σύγκριση των δύο αλγορίθμων A-buffer και K-buffer αλλά και τη σύγκριση μεταξύ αλγορίθμων K-buffer με διαφορετικό βάθος βάσει των frames ανά δευτερόλεπτο, η χειρότερη επίδοση είναι αυτή του A-buffer στην απόδοση του Stanford dragon και η καλύτερη επίδοση είναι αυτή του K-buffer με τιμή βάθους 2 στην απόδοση του εδάφους. Μεταξύ των τριών K-buffer με διαφορετικό βάθος, την χειρότερη απόδοση διαθέτει ο αλγόριθμος με το μεγαλύτερο βάθος, 10, και την καλύτερη διαθέτει ο αλγόριθμος με τιμή βάθους 2.

	A-buffer	K-buffer depth=2	K-buffer depth=5	K-buffer depth=10
Dragon	233	158	177	198
Bunny	231	155	170	198
Terrain	232	156	174	198

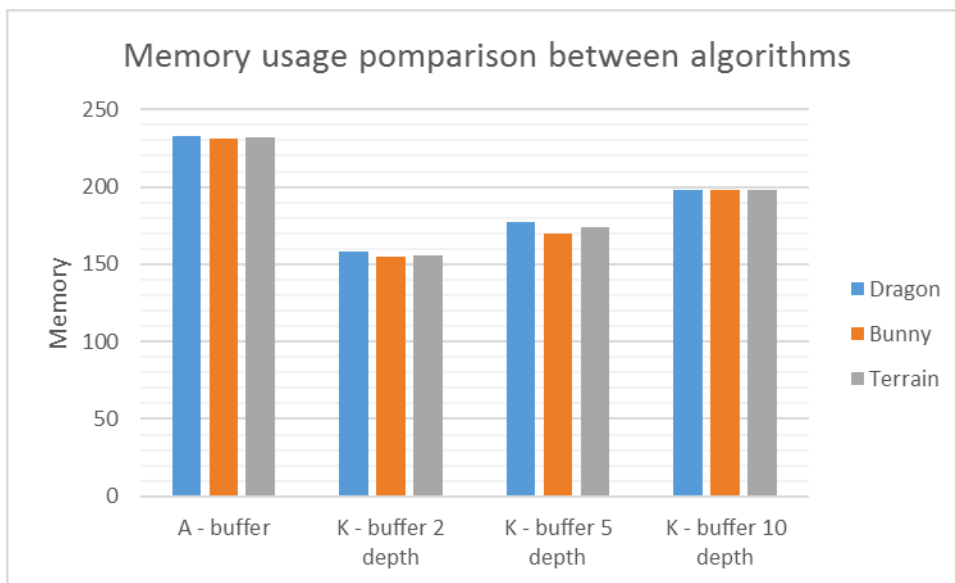
Πίνακας 3.3 : Τιμές δέσμευσης μνήμης A-buffer και K-buffer

Όσον αφορά στη δέσμευση μνήμης, είναι φανερό από τις τιμές που παρατηρούνται πως ο A-buffer δεσμεύει περισσότερη μνήμη σε σχέση με τον K-buffer. Συγκεκριμένα, οι υψηλότερες τιμές δέσμευσης μνήμης προέρχονται μόνο από τον αλγόριθμο A-buffer και τις πλησιάζουν σχετικά οι τιμές του αλγόριθμου K-buffer με τιμή βάθους 10. Αυτό που παρατηρείται είναι πως ο K-buffer με βάθος 2 δεσμεύει πολύ μικρότερη μνήμη σε σχέση με οποιαδήποτε άλλη περίπτωση αλγόριθμου, με τις τιμές αυτές να αυξάνονται καθώς αυξάνεται η πολυπλοκότητα βάθους. Η χαμηλότερη δέσμευση μνήμης που παρατηρείται προέρχεται από τον K-buffer στην απόδοση του μοντέλου Stanford bunny. Η υψηλότερη παρατηρείται στην απόδοση του Stanford Dragon από τον αλγόριθμο A-buffer. Όσον αφορά στις αλλαγές πολυπλοκότητας βάθους του K-buffer, παρατηρείται μία σχετικά ανάλογη αύξηση στην δέσμευση μνήμης και στα τρία μοντέλα καθώς αυξάνεται η τιμή βάθους.

Ακολουθούν γραφήματα για οπτική σύγκριση της απόδοσης των αλγορίθμων όσον αφορά τα frames ανά δευτερόλεπτο και τη δέσμευση μνήμης αντίστοιχα. Στο παράρτημα βρίσκονται περισσότερα γραφήματα, τα οποία παρουσιάζουν συγκρίσεις και βάσει άλλων παραγόντων όπως την αύξηση πολυπλοκότητας βάθους σε K-buffer αλγόριθμο αλλά και για το κάθε μοντέλο ξεχωριστά.



Γράφημα 3.3: Σύγκριση FPS A-buffer και K-buffer για κάθε μοντέλο.



Γράφημα 3.4: Σύγκριση δέσμευσης μνήμης A-buffer και K-buffer για κάθε μοντέλο.

ΚΕΦΑΛΑΙΟ 4. ΔΗΜΙΟΥΡΓΙΑ ΜΕΓΑΛΗΣ ΥΒΡΙΔΙΚΗΣ ΣΚΗΝΩΝ

4.1 Εισαγωγικά

4.2 Workflow

4.1 Εισαγωγικά

Με την εξέλιξη της τεχνολογίας αυξάνεται συνεχώς η χρήση μηχανών γραφικών για την παραγωγή τρισδιάστατου περιεχομένου, όπως παιχνίδια, βίντεο, και παρουσίαση εικόνων και σκηνών οι οποίες θα ήταν δύσκολο, ακόμη και αδύνατον, να παρουσιαστούν με συμβατικά μέσα, όπως απλή βιντεοσκόπηση. Στην εποχή μας υπάρχει πλέον η δυνατότητα να αποδίδονται μοντέλα τα οποία είναι πολύ κοντά στην πραγματικότητα, ή να αποδοθούν με τρόπο ο οποίος δίνει την ψευδαίσθηση της ύπαρξής τους. Με τον τρόπο αυτό μπορεί πλέον ο θεατής, για παράδειγμα, να βλέπει να “ζωντανεύουν” μπροστά στα μάτια του ζώα τα οποία δεν υπάρχουν πλέον, ή έχουν εξελιχθεί σε διαφορετικές μορφές, αλλά ακόμη και αντικείμενα και πλάσματα της ανθρώπινης φαντασίας. Επίσης μπορούν να αλληλεπιδρούν σκηνές ζωντανής δράσης πραγματικών μοντέλων, για παράδειγμα ηθοποιοί, με σκηνές που έχουν αναπτυχθεί εξ' ολοκλήρου σε υπολογιστή.

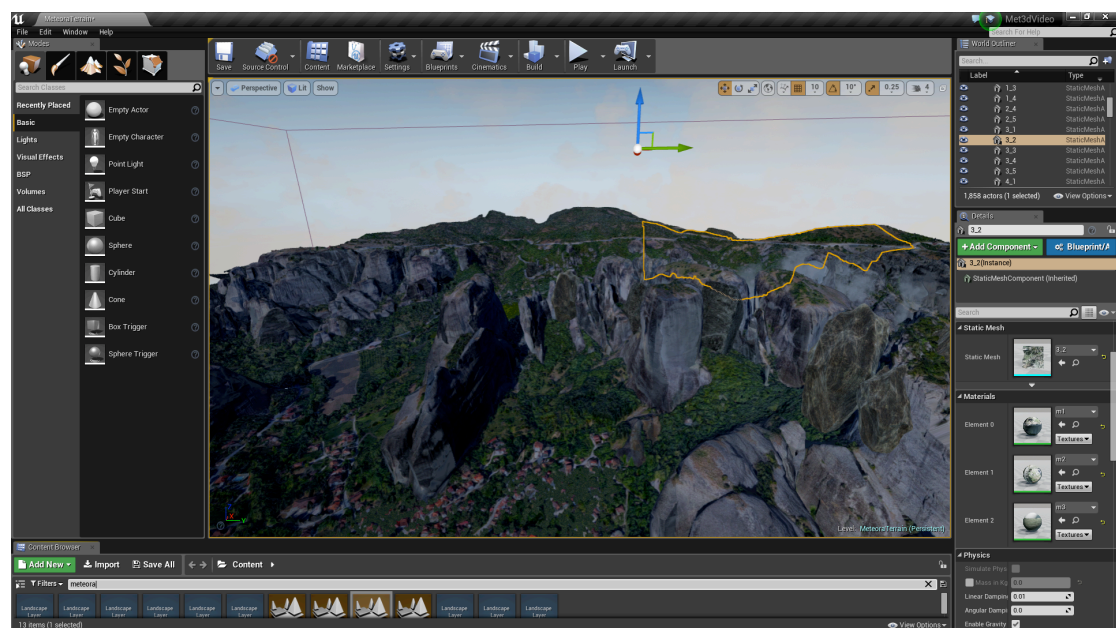
Επιχειρήσαμε τη δημιουργία μιας μεγάλης σκηνής για να εξετάσουμε την απόδοση του τελικού αποτελέσματος, η οποία θα έχει πολύ μεγάλη έκταση και θα συνδυάζει μοντέλα πραγματικά και τεχνητά, τα οποία θα αλληλεπιδρούν στο χώρο. Ο τρόπος παρουσίασης θα είναι μιας λήψης, ακολουθώντας κάθε στιγμή έναν πρωταγωνιστή,

ταξιδεύοντας σε ολόκληρη την έκταση της σκηνής. Η επιλεγμένη θεματολογία της σκηνής αφορά τη δημιουργία του τοπίου των Μετεώρων από την προϊστορία μέχρι την απόκτηση της μορφολογίας που γνωρίζουμε σήμερα, σε μορφή timelapse μιας λήψης.

4.2 Workflow

4.2.1 Βασικό τοπίο

Το πρώτο βήμα αφορά τη δημιουργία του βασικού τοπίου, το οποίο ορίζει το μέγεθος της σκηνής. Στην παρούσα περίπτωση η σκηνή είναι πολύ μεγάλου μεγέθους, καθώς αναπαριστά την πραγματική έκταση του τοπίου των Μετεώρων. Για την επίτευξη της δημιουργίας της σκηνής χρησιμοποιήθηκαν μοντέλα τα οποία ανακατασκευάστηκαν από φωτογραφίες του εδάφους, η λήψη των οποίων πραγματοποιήθηκε με drone. Τα μοντέλα που δημιουργήθηκαν είναι τύπου fbx, απαραίτητο για την εισαγωγή στη μηχανή γραφικών Unreal Engine. Η επεξεργασία των μοντέλων πραγματοποιήθηκε στο πρόγραμμα Geomagic, για decimation ώστε να μειωθεί ο αριθμός των πολυγώνων, προσθήκη normals και textures για καλύτερη απόδοσή τους με διατήρηση της ποιότητας.



Σχήμα 4.1: Μοντέλα Μετεώρων

4.2.2 Επιλογή σημείων ενδιαφέροντος

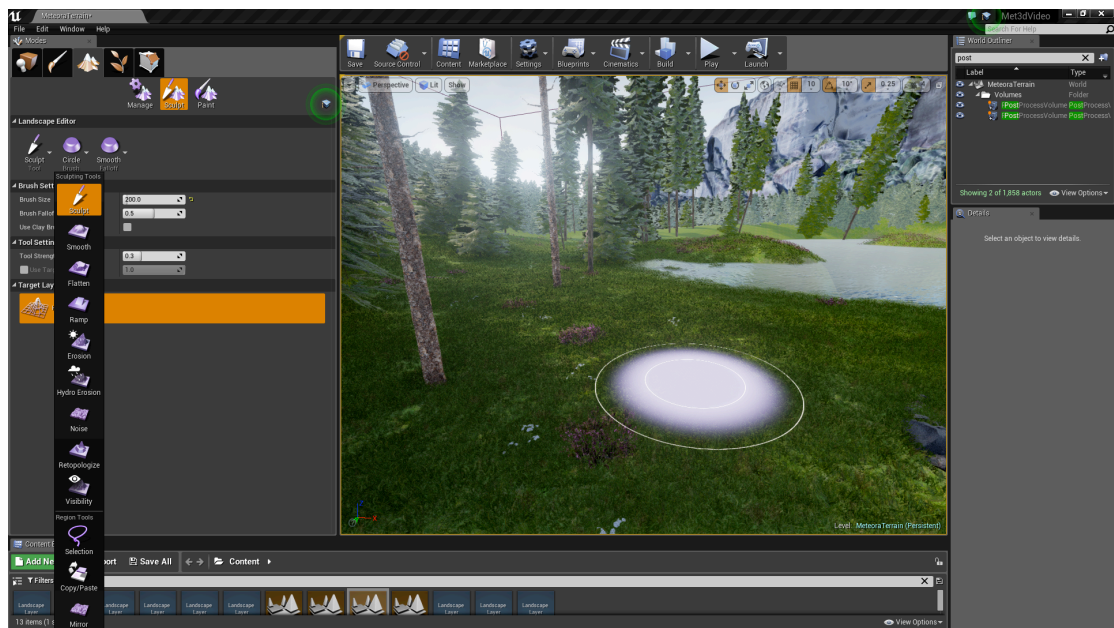
Όσον αφορά τα σημεία ενδιαφέροντος αναζητήθηκαν και επελέχθησαν συγκεκριμένα ανοιχτά σημεία της σκηνής στα οποία η γεωμετρία των βράχων είναι εμφανής, όπου θα διεξαχθούν οι βασικές σκηνές του βίντεο. Καθώς η προσοχή του θεατή θα βρίσκεται στα σημεία ενδιαφέροντος, είναι σημαντική η διαμόρφωση του εδάφους να είναι τέτοια που θα αντανakλά την πραγματικότητα. Η μηχανή γραφικών Unreal Engine παρέχει εργαλείο δημιουργίας εδάφους με πολλαπλές λειτουργίες διαμόρφωσής του, όπως εξύψωση, εξομάλυνση και βαφή. Με τη χρήση του εργαλείου, και σύμφωνα με τα δεδομένα των πραγματικών μοντέλων, έγινε η διαμόρφωση του εδάφους.

Το εργαλείο που διαθέτει η Unreal Engine για τη διαμόρφωση εδάφους ονομάζεται Landscape tool. Έχει πολλές δυνατότητες στη δημιουργία τοπίου πολύ μεγάλης έκτασης. Επιλέγοντας το εργαλείο μέσω των επιλογών Modes ανοίγει το παράθυρο στο management όπου είναι δυνατή η δημιουργία νέων τοπίων ή επεξεργασία ήδη υπάρχοντων βάσει των στοιχείων τους. Επίσης μπορούν να τροποποιηθούν περισσότερα από ένα τοπία στη σκηνή.

Στο manual mode υπάρχουν οπτικές επιλογές οι οποίες θέτουν τις παραμέτρους του τοπίου, όπως υλικό, τοποθεσία της σκηνής, μέγεθος σε σχέση με τη σκηνή, χαρακτηριστικά του τοπίου, τον αριθμό πολυγώνων που χρησιμοποιεί, και άλλες επιλογές.

Είναι σημαντικό να αναφερθεί πως όσο αυξάνεται ο αριθμός των περιεχομένων τόσο επηρεάζεται η τελική απόδοση. Όταν έχουν γίνει όλες οι επιλογές που επιθυμεί ο προγραμματιστής, μπορεί να τοποθετηθεί στη σκηνή το τοπίο με την επιλογή create. Στο σημείο αυτό έχει τοποθετηθεί το βασικό τοπίο, έτοιμο για επεξεργασία. Η ανάπλαση κατορθώνεται με τη χρήση του εργαλείου Sculpt το οποίο παρέχει πινέλα για την εφαρμογή των επιθυμητών αλλαγών. Υπάρχει επίσης η επιλογή έντασης του πινέλου από απαλή έως οξεία για δημιουργία ομαλών επιφανειών και λεπτομερειών αντίστοιχα. Επίσης υπάρχει το εργαλείο Flatten το οποίο πραγματοποιεί ισοπέδωση ή ισοπέδωση σύμφωνα με την κλίση του εδάφους και η δημιουργία ράμπας μεταξύ 2 σημείων της επιλογής του προγραμματιστή. Ένα ενδιαφέρον εργαλείο αποτελεί

εκείνο της διάβρωσης θερμικής και υδατικής το οποίο προσομοιώνει τη μορφή του εδάφους ύστερα από διάβρωση ανάλογα με την επιθυμία του προγραμματιστή. Τέλος, όσον αφορά τη βαφή του τοπίου, υπάρχουν εργαλεία Paint τα οποία προσδίδουν στο έδαφος το υλικό από το οποίο αποτελείται το κάθε σημείο όπως χρώμα, γρασίδι, χιόνι καθώς και την ανάμιξη δύο ή περισσότερων υλικών για πιο ρεαλιστική απόδοση.

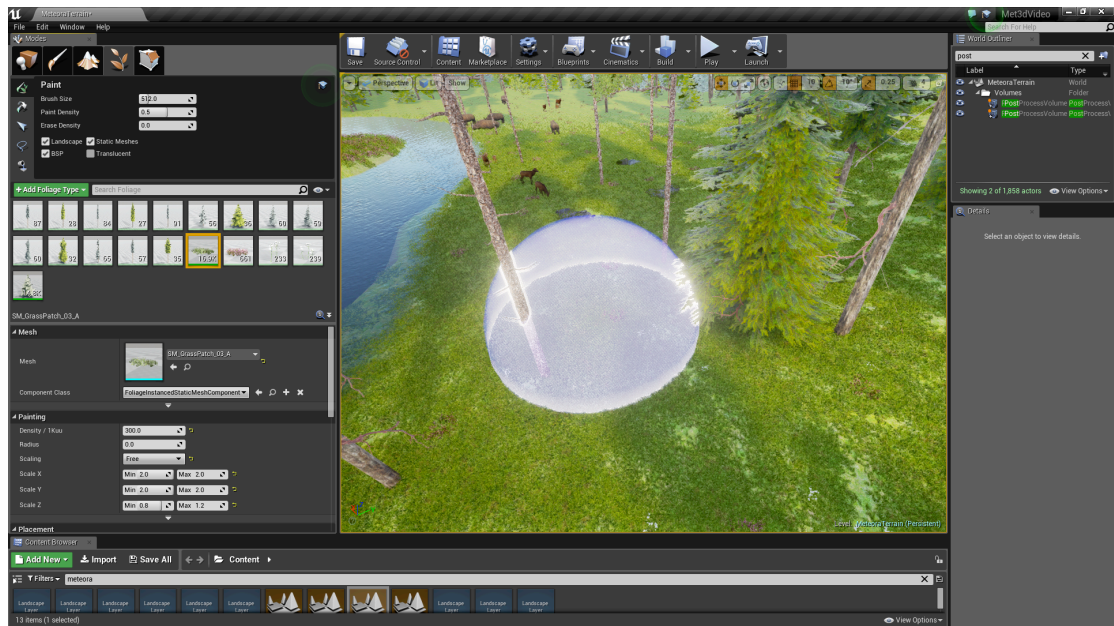


Σχήμα 4.2: Terrain Editor

4.2.3 Εμπλουτισμός σκηνής

Όπως προαναφέρθηκε στα σημεία ενδιαφέροντος υπάρχει διαδραμάτιση κάποιων σκηνών. Καθώς η προσοχή του θεατή θα είναι στραμμένη στις εν λόγω σκηνές, είναι απαραίτητος ο εμπλουτισμός τους για καλύτερο αισθητικό και πιο ρεαλιστικό αποτέλεσμα. Η μηχανή γραφικών προσφέρει ένα πολύ χρήσιμο εργαλείο για τον εμπλουτισμό του εδάφους με την προσθήκη της χλωρίδας. Η χλωρίδα που προστέθηκε αποτελείται τεχνητά μοντέλα δένδρων, λουλουδιών και γρασιδιού τριών διαστάσεων. Τέλος, οι ανάγκες συγκεκριμένων σκηνών βίντεο απαιτούσαν περισσότερο πολύπλοκους συνδυασμούς πετρωμάτων και χλωρίδας, τα μοντέλα των οποίων τοποθετήθηκαν με στρατηγικό τρόπο για την αισθητική και ρεαλιστική βελτίωση της σκηνής.

Στο ίδιο σημείο που βρίσκουμε τον landscape editor βρίσκουμε και την επιλογή Foliage. Το συγκεκριμένο εργαλείο αφορά ένα σύστημα το οποίο επιτρέπει στον προγραμματιστή να τοποθετήσει ή να αφαιρέσει ομάδες στατικών μοντέλων στο έδαφος της σκηνής. Τα τοποθετημένα μοντέλα ομαδοποιούνται με τέτοιον τρόπο ώστε για την απόδοση πολλαπλών μονάδων, χρειάζεται μόνο μια κλήση στην κάρτα γραφικών για την απόδοση όλων. Επιλέγοντας το εργαλείο εμφανίζεται η λίστα των μοντέλων από την οποία επιλέγει ο προγραμματιστής αυτό που θέλει να χρησιμοποιήσει. Επίσης παρέχεται και η δυνατότητα διαμόρφωσης των λεπτομερειών του μοντέλου. Η επιλογή και παράλειψη των επιλεγμένων μοντέλων γίνεται με απλό check - uncheck στα αντίστοιχα τα εικονίδια τους στη λίστα. Η τοποθέτησή τους στο τοπίο πραγματοποιείται με χρήση πινέλου το οποίο είναι επίσης επεξεργάσιμο. Το πινέλο εμφανίζεται ως διαφανής θόλος στη σκηνή μέσα στον οποίο σύμφωνα με τις επιλογές του προγραμματιστή θα εμφανιστούν τα στοιχεία που έχει επιλέξει. Υπάρχει επίσης η επιλογή εναλλαγής σε προηγούμενως επιλεγμένα στοιχεία προς τοποθέτηση. Με το εργαλείο επιλογής Select tool τίθεται δυνατό να επιλεχθούν εξατομικευμένα ή περισσότερα του ενός στοιχεία. Με αυτόν τον τρόπο δύναται ο προγραμματιστής να εκτελέσει κάποιες ενέργειες όπως η αλλαγή άξονα του αντικειμένου ή διαγραφή. Υπάρχουν και εργαλεία τα οποία διευκολύνουν στην τοποθέτηση στοιχείων σε μεγάλες εκτάσεις όπως το fill tool. Τέλος υπάρχουν και εργαλεία που βοηθούν στην λεπτομερέστερη επεξεργασία των μοντέλων και στην καλύτερή τους απόδοση.



Σχήμα 4.3: Foliage editor

4.2.4 Particle effects

Καθώς το βίντεο που επιλέξαμε να δημιουργήσουμε αφορά στην εξέλιξη ενός γεωγραφικού τοπίου μέσα στο χρόνο ήταν απαραίτητη η προσθήκη διαφόρων εφέ καιρού όπως χιόνι και ομίχλη, αλλά και διαφορετικά εφέ όπως αυτό της φωτιάς. Για τη δημιουργία αυτών χρησιμοποιήθηκαν τα λεγόμενα particles τα οποία μπορούν να δημιουργηθούν μέσα από τη συγκεκριμένη μηχανή γραφικών.

Ένα particle system αποτελεί μία τεχνική στη φυσική (physics) παιχνιδιών, γραφικών κίνησης, και γραφικών υπολογιστών η οποία χρησιμοποιεί ένα μεγάλο αριθμό πολύ μικρών στοιχείων, τρισδιάστατων μοντέλων, ή άλλων αντικειμένων για την προσομοίωση συγκεκριμένων τύπων ακαθόριστου σχήματος φαινομένων τα οποία είναι δύσκολο να αναπαραχθούν με συμβατικές τεχνικές rendering - συνήθως συστήματα υψηλώς χαοτικά, φυσικά φαινόμενα, ή διαδικασίες που προκαλούνται από χημικές αντιδράσεις. Παραδείγματα τέτοιων φαινομένων τα οποία αναπαράγονται συχνά με την χρήση των particle systems περιλαμβάνουν φωτιά, εκρήξεις, καπνό, κινούμενο νερό, σπίθες, σύννεφα, ομίχλη, χιόνι, σκόνη, αλλά και αφηρημένα οπτικά εφέ όπως λαμυρίζοντα ίχνη τα οποία χρησιμοποιούν στοιχεία στα οποία εξαφανίζονται και εμφανίζονται ξανά άμεσα από την πηγή του εφέ. Μία άλλη τεχνική μπορεί να χρησιμοποιηθεί για αντικείμενα που διαθέτουν πολλές ίνες (strands), όπως

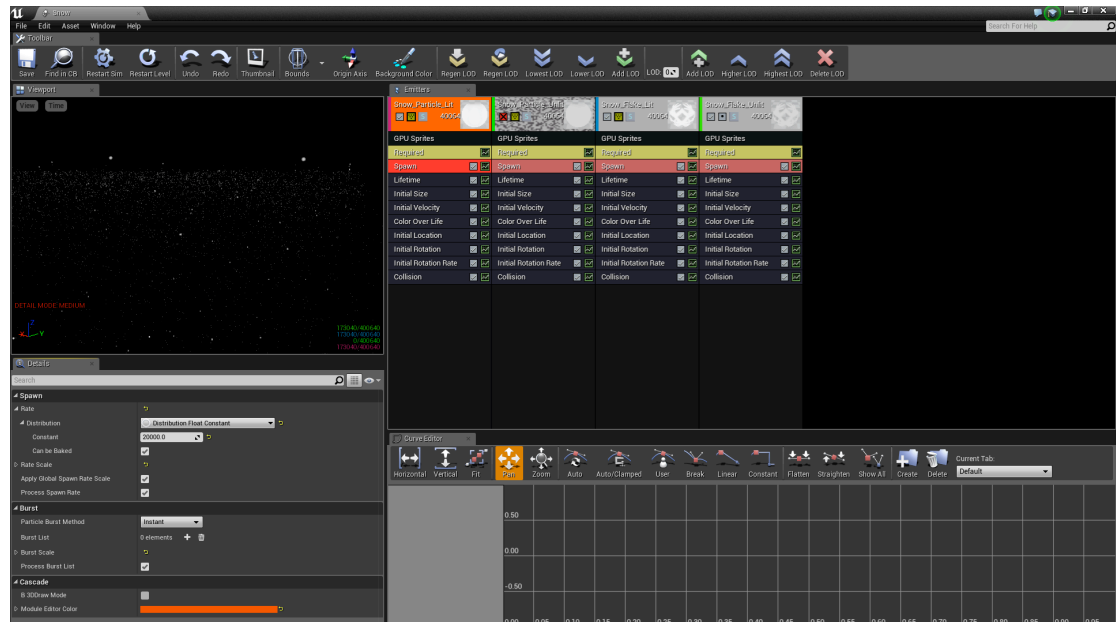
γούνα, μαλλιά, κάτι που συμπεριλαμβάνει το στιγμιαίο rendering ολόκληρης της ζωή ενός στοιχείου το οποίο μπορεί να σχεδιαστεί και να επεξεργαστεί ως μια μοναδική ίνα του επιλεγμένου υλικού. Τα συστήματα αυτά μπορούν να είναι δισδιάστατα ή τρισδιάστατα. Τυπικά η θέση και η κίνηση ενός συστήματος στο τρισδιάστατο χώρο ελέγχεται από αυτό που αποκαλείται πομπός (emitter). Ο εν λόγω πομπός δρα ως πηγή των στοιχείων, και η θέση του στον τρισδιάστατο χώρο καθορίζει τη θέση παραγωγής τους και την κατεύθυνσή τους. Ως πομπός μπορεί να χρησιμοποιηθεί ένα απλό τρισδιάστατο μοντέλο όπως ένας κύβος. Κάθε πομπός είναι συνδεδεμένος με ένα σύνολο παραμέτρων συμπεριφοράς των στοιχείων. Οι παράμετροι αυτές μπορούν να συμπεριλαμβάνουν το ρυθμό γένεσης, τη διάρκεια ζωής των στοιχείων, την αρχική κατεύθυνση κατόπιν γέννησης, χρώμα και πολλές περισσότερες. Είναι συχνό φαινόμενο για τις περισσότερες, αν όχι όλες, παραμέτρους να είναι ασαφείς σε αντίθεση του να έχουν ακριβή αριθμητική τιμή. Ο προγραμματιστής καθορίζει μία κεντρική τιμή και τον βαθμό επιτρεπόμενης τυχειότητας γύρω από το κέντρο. Στην περίπτωση χρήσης μοντέλου ως πομπό, η αρχική κατεύθυνση συνήθως τίθεται προαιρετικά ανάλογη των normals της κάθε πλευράς του αντικειμένου δίνοντας την ψευδαίσθηση πώς μεταδίδονται απευθείας από κάθε πλευρά.

Ένας τυπικός κύκλος ανανέωσης ενός particle system μπορεί να χωριστεί σε δύο ξεχωριστές φάσεις, την πρώτη, parameter update/simulation, και τη δεύτερη φάση rendering.

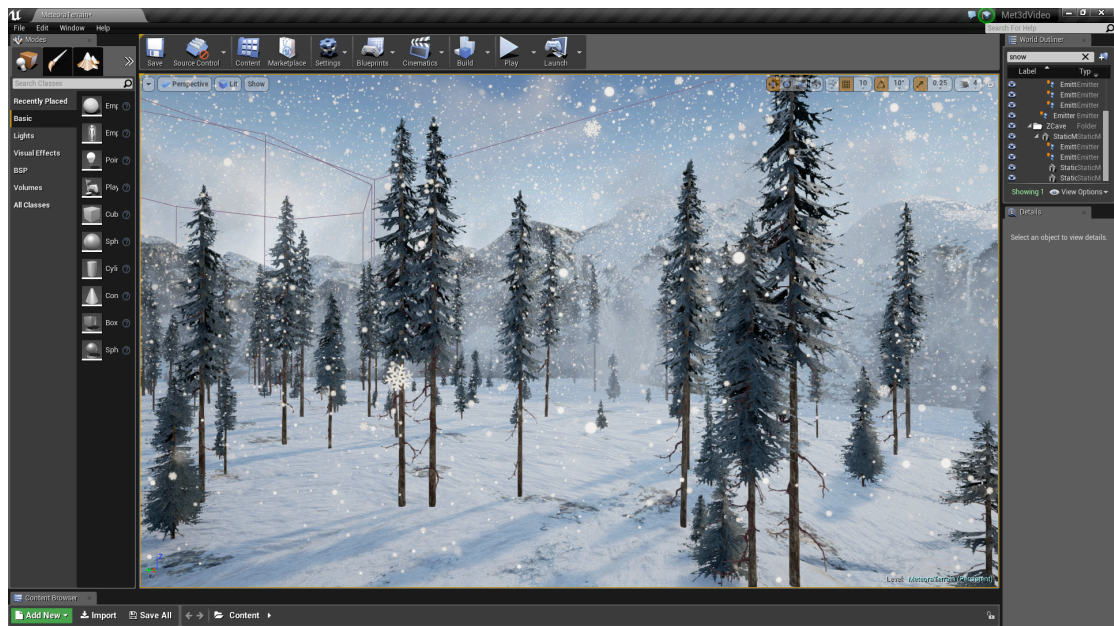
Κατά την πρώτη φάση υπολογίζεται ο αριθμός των νέων στοιχείων που πρέπει να δημιουργηθούν βάσει του ρυθμού δημιουργίας και του διαστήματος μεταξύ των ανανεώσεων. Καθε ένα από αυτά εμφανίζεται σε συγκεκριμένο σημείο στον τρισδιάστατο χώρο βάσει της θέσεως του πομπού και την τοποθεσία γέννησης που έχει προσδιοριστεί. Οι παράμετροι κάθε στοιχείου ενεργοποιούνται σύμφωνα με τις παραμέτρους του πομπού. Με κάθε ανανέωση το σύνολο των ήδη υπαρχόντων στοιχείων ελέγχεται για την περίπτωση υπέρβασης της ζωής τους ώστε να αφαιρεθούν από την προσομοίωση. Στην αντίθετη περίπτωση η θέση των στοιχείων και τα υπόλοιπα χαρακτηριστικά τους εξελίσσονται βάσει μιας φυσικής προσομοίωσης, η οποία μπορεί να είναι τόσο απλή όσο μεταφράζοντας την ισχύουσα θέση τους, έως τόσο σύνθετη όσο η πραγμάτωση φυσικά ακριβών υπολογισμών

τροχιάς με το συνυπολογισμό εξωτερικών δυνάμεων όπως βαρύτητα, αέρας, τριβή. Είναι σύνηθες να πραγματοποιείται εντοπισμός σύγκρουσης μεταξύ στοιχείων και προσδιορισμένων τρισδιάστατων αντικειμένων στη σκηνή, ώστε τα στοιχεία να αντανακλώνται ή να αλληλεπιδρούν διαφορετικά με εμπόδια στο περιβάλλον. Συγκρούσεις μεταξύ στοιχείων χρησιμοποιούνται σπάνια καθώς απαιτούν μεγάλο υπολογιστικό κόστος.

Στη φάση rendering, μετά την ολοκλήρωση της ανανέωσης, κάθε στοιχείο αποδίδεται συνήθως με τη μορφή ενός textured billboarded quad (ένα ορθογώνιο παραλληλόγραμμο texture, πάντα οριζόντιο στον χρήστη.). Ωστόσο αυτό κάποιες φορές δεν είναι απαραίτητο σε παιχνίδια. Ένα στοιχείο μπορεί να αποδοθεί ως ένα μόνο pixel σε μικρή ανάλυση ή σε περιβάλλοντα περιορισμένης επεξεργαστικής ισχύος. Τέλος στη θέση στοιχείων μπορούν να χρησιμοποιηθούν τρισδιάστατα μοντέλα όπως για παράδειγμα μία χιονοθύελλα μπορεί να αποτελείται από ένα τρισδιάστατο μοντέλο χιονονιφάδας το οποίο πολλαπλασιάζεται και περιστρέφεται ώστε να αντιπαραβάλλει τις θέσεις χιλιάδων ή εκατομμυρίων στοιχείων.



Σχήμα 4.4: Particle editor



Σχήμα 4.5: Particle - χιόνι

4.2.5 Blueprints

Η μηχανή γραφικών παρέχει ένα σύστημα οπτικού προγραμματισμού βασισμένο στο σκεπτικό χρήσης περιβάλλοντος διεπαφής μέσω κόμβων (nodes) για τη δημιουργία στοιχείων τα οποία ονομάζονται blueprints. Το συγκεκριμένο σύστημα είναι εξαιρετικά ευέλικτο και ισχυρό καθώς παρέχει τη δυνατότητα στο σχεδιαστή να χρησιμοποιεί ολόκληρο το σύνολο των εργαλείων τα οποία γενικώς είναι διαθέσιμα μόνο σε προγραμματιστές. Παρέχει όμως και τη δυνατότητα να δημιουργηθούν βασικά συστήματα τα οποία μπορούν αργότερα να επεκταθούν με προσθήκη στοιχείων. Η βασική λειτουργία του εργαλείου είναι ο καθορισμός κλάσεων αλλά και αντικειμένων μέσα στη μηχανή γραφικών τα οποία μπορούν να αλληλεπιδράσουν μεταξύ τους ανάλογα με τον τρόπο που έχουν συνδυαστεί οπτικά οι διάφοροι κόμβοι στον editor. Ο κάθε κόμβος αφορά μία οντότητα-κλάση αλλά ακόμη και στοιχεία των οντοτήτων όπως κίνηση, τα οποία ονομάζονται animation blueprints. Στον κόμβο εμφανίζονται τα ήδη επιλεγμένα στοιχεία - ιδιότητες από τον προγραμματιστή τα οποία ενώνονται με γραμμές οι οποίες αντιστοιχούν στην ακολουθία τους. Κάθε νέος κόμβος που προστίθεται από τον προγραμματιστή μπορεί να περιέχει οποιουδήποτε είδους ρύθμιση ή ιδιότητα η οποία συνδέεται με τον κόμβο τον οποίον αφορά. Ο κάθε

κόμβος που δημιουργείται, αποτελεί ένα blueprint το οποίο υπάρχει στη σκηνή που αναπτύσσεται και συνδέεται με άλλους κόμβους με τους οποίους αλληλεπιδρά.

Με τον τρόπο αυτό δημιουργήθηκαν τα διάφορα blueprints τα οποία χρησιμοποιήθηκαν στη σκηνή που δημιουργήσαμε.

4.2.6 Φωτισμός

Ο όρος post processing χρησιμοποιείται στις μεθόδους επεξεργασίας εικόνας για την βελτίωση της ποιότητας που χρησιμοποιούνται σε συσκευές αναπαραγωγής, σε προγράμματα αναπαραγωγής και transcoding προγράμματα. Επίσης χρησιμοποιείται σε 3d rendering πραγματικού χρόνου για παράδειγμα σε παιχνίδια για την παρουσίαση των περισσότερων εφέ.

Post processing ενός βίντεο είναι η διαδικασία της αλλαγής στην αντιλαμβανόμενη ποιότητα ενός βίντεο κατά την αναπαραγωγή, η οποία ακολουθεί την διαδικασία αποκωδικοποίησης. Υπάρχουν τεχνικές (routines), για παράδειγμα, αλλαγής μεγέθους εικόνων οι οποίες συμπεριλαμβάνουν Γραμμική παρεμβολή linear interpolation διγραμμική παρεμβολή (bilinear interpolation) κυβική παρεμβολή (cubic interpolation). Οι τεχνικές αυτές βοηθούν στη μείωση και κάλυψη μειονεκτημάτων και artifacts το πρωταρχικό υλικό. Είναι σημαντικό να αναφερθεί πως το post processing πάντα συμπεριλαμβάνει μία εναλλαγή μεταξύ ταχύτητας ομαλότητας και καθαρότητας.

Επιπροσθέτως, post - processing χρησιμοποιείται και στο 3d rendering, ιδιαίτερα για τη δημιουργία βιντεοπαιχνιδιών. Αντί του άμεσου rendering τρισδιάστατων αντικειμένων στην οθόνη η σκηνή αποδίδεται αρχικά σε ένα buffer στη μνήμη της κάρτας γραφικών. Συνήθως χρησιμοποιούνται fragment shaders και προαιρετικά vertex shaders για την εφαρμογή φίλτρων post processing στην αποθηκευμένη εικόνα πριν εμφανιστεί στην οθόνη. Κάποια post processing εφέ απαιτούν επίσης multiple passes, gamma inputs, vertex manipulation and depth buffer access. Ενώ συνήθως κάθε τρισδιάστατο αντικείμενο αποδίδεται ατομικά, το post production επιτρέπει τη χρήση εφέ τα οποία απαιτούν επίγνωση του συνόλου της εικόνας.



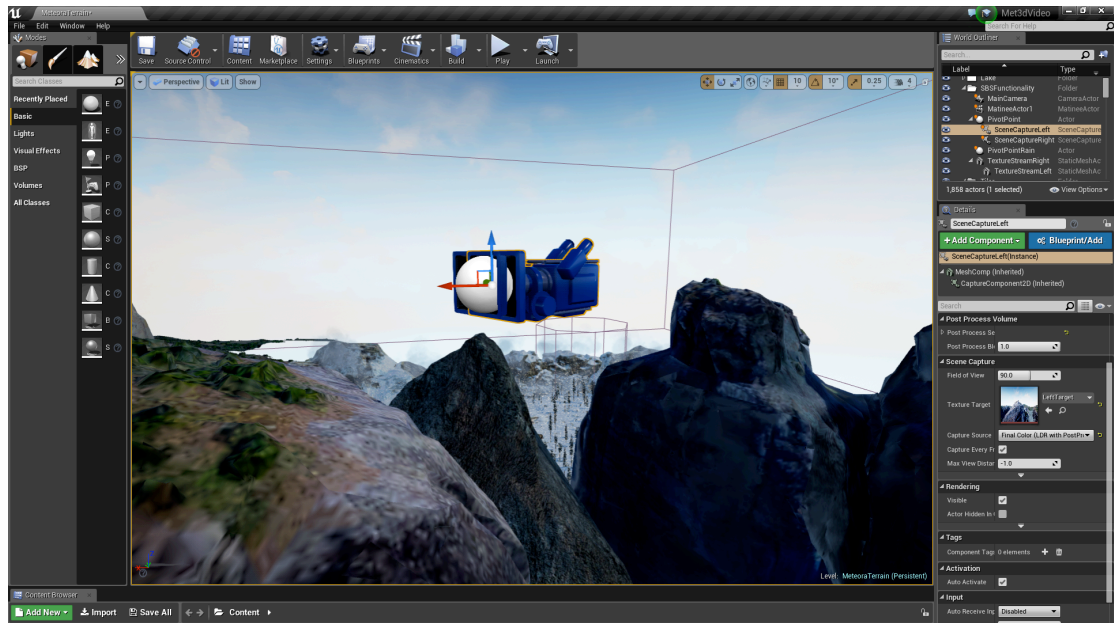
Σχήμα 4.6: Επίδραση Post-process

4.2.7 Στερεοσκοπικότητα

Η τεχνολογία πλέον επιτρέπει μέσω ειδικών συσκευών να παρακολουθήσει ο θεατής στερεοσκοπικές σκηνές. Για την επίτευξη μια στερεοσκοπικής σκηνής πρέπει να υπάρχουν 2 λήψεις της εκάστοτε εικόνας, οι οποίες αντιπροσωπεύουν την γωνία θέασης κάθε ματιού. Όσον αφορά την ανάλυση, ένα στερεοσκοπικό βίντεο ποιότητας 1920x1080p πρέπει να είναι της μορφής 3840x1080p ώστε κατά την διάρκεια της θέασής του να έχει την επιθυμητή τελική ποιότητα, συνδυάζοντας τις 2 λήψεις, μία για κάθε μάτι.

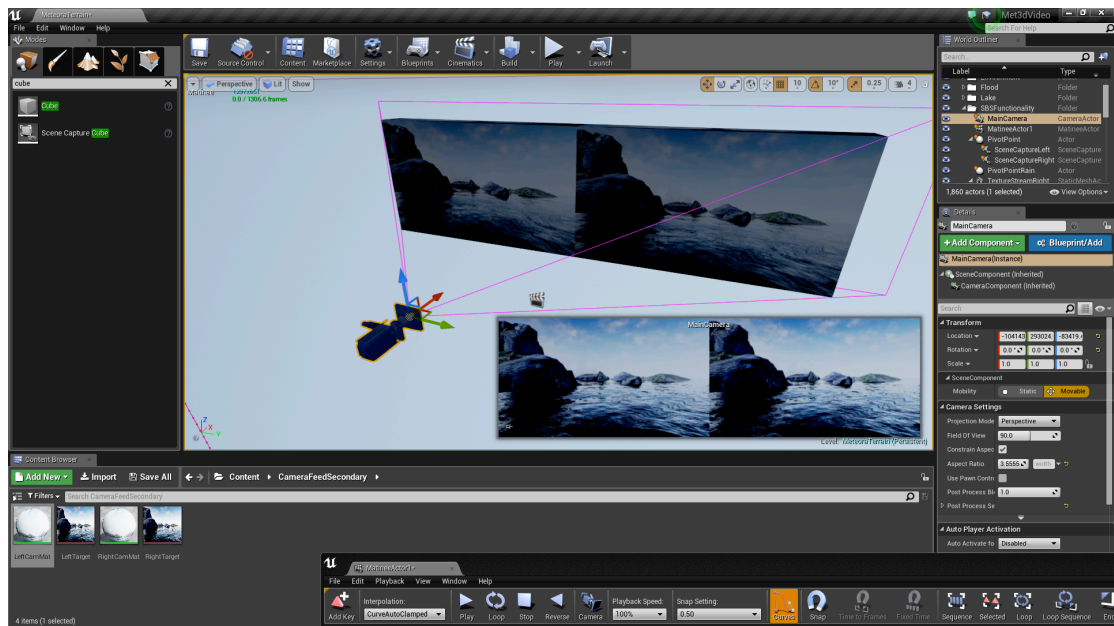
Η Unreal Engine δεν υποστηρίζει παραγωγή στερεοσκοπικού βίντεο μέσω ήδη υπάρχουσας ενσωματωμένης λειτουργίας. Τέθηκε, λοιπόν, απαραίτητο να βρεθεί μια τεχνική για την λήψη διαφορετικών γωνιών της ίδιας εικόνας. Η βασική ιδέα είναι πως πρέπει να βγει ένα βίντεο ανάλυσης 3840x1080p το οποίο θα αποτελείται από δυο 1920x1080p εικόνες της ίδιας σκηνής παρατεθειμένες οριζόντια, οι οποίες θα έχουν διαφορά γωνίας θέασης. Η επίλυση του προβλήματος αναλύεται επακολούθως.

Το πρώτο βήμα είναι να στηθούν 2 κάμερες οι οποίες θα καταγράφουν τις 2 γωνιές θέασης. Αυτό που κάναμε είναι να ορίσουμε ένα συγκεκριμένο σημείο πάνω στο οποίο είναι τοποθετημένες οι κάμερες. Η θέση που επελέχθη για τις κάμερες, ύστερα από δοκιμές που πραγματοποιήθηκαν, είναι ιδανική για την επίτευξη της στερεοσκοπικότητας.



Σχήμα 4.7: Κάμερες/μάτια

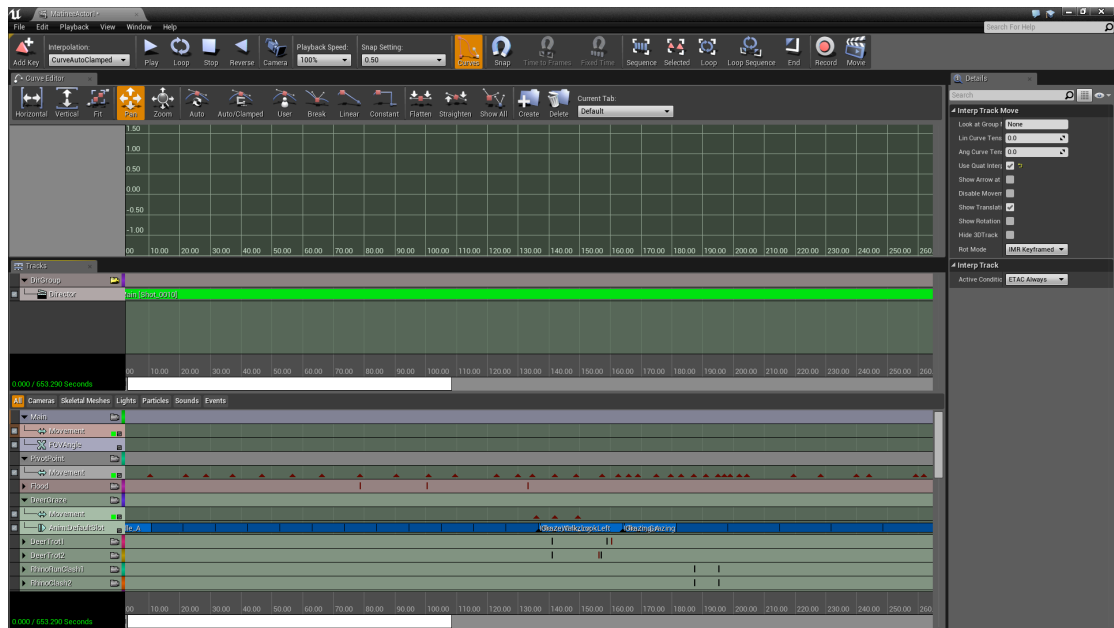
Το δεύτερο βήμα είναι να δημιουργηθεί ένα σύστημα το οποίο θα συνδυάζει τις λήψεις από τις δύο κάμερες και θα αποτελεί το τελικό αποτέλεσμα του βίντεο. Αυτό επιτυγχάνεται ως εξής: Δημιουργούμε μια τρίτη κάμερα, της οποίας την ανάλυση θέτουμε σε 3840x1080p, ώστε να καταγραφεί το συνολικό αποτέλεσμα. Μπροστά από αυτήν την κάμερα τοποθετούμε δυο ορθογώνια παραλληλόγραμμα των οποίων οι διαστάσεις έχουν αναλογία 16:9 και σε κάθε ένα αναθέτουμε ένα ειδικό texture. Το κάθε texture σχεδιάζεται από την εικόνα που καταγράφει η κάθε κάμερα από τις δύο που προαναφέραμε, και τοποθετείται στο κάθε παραλληλόγραμμο αντίστοιχα με το μάτι που αντιπροσωπεύει. Το τελικό αποτέλεσμα που προκύπτει είναι η καταγραφή της τρίτης κάμερας.



Σχήμα 4.8: Κυρίως κάμερα

4.2.8 Matinee

Για την παραγωγή του βίντεο η Unreal Engine παρέχει ένα εργαλείο το οποίο ονομάζεται Matinee animation tool. Το εργαλείο αυτό παρέχει τη δυνατότητα απόδοσης κίνησης και ελέγχου των οντοτήτων της σκηνής μέσα στο χρόνο, για τη δημιουργία είτε δυναμικού gameplay, είτε δημιουργίας κινηματογραφικών σκηνών. Το σύστημα βασίζεται στη χρήση ειδικευμένων οδηγών animation (animation tracks) στους οποίους μπορούν να τοποθετηθούν εντολές καθορισμού της τιμής των μεταβλητών συγκεκριμένων ιδιοτήτων των οντοτήτων στη σκηνή. Παραδείγματα των ιδιοτήτων που μπορούν να ελεγχθούν αποτελούν η αλλαγή θέσης και γωνίας, ενεργοποίηση πακέτου εφέ, ενεργοποίηση animation και ήχου. Στην προκειμένη περίπτωση το βασικότερο στοιχείο είναι ο ορισμός της πορείας των δύο καμερών που προαναφέρθηκαν για την περιήγηση στη σκηνή. Το τελικό αποτέλεσμα παράγεται με το εργαλείο Movie, το οποίο καταγράφει την εικόνα της τρίτης κάμερας και διαθέτει επιλογές για την ποιότητα της καταγραφής. Το αποτέλεσμα που παράγεται αποτελείται από μία ακολουθία εικόνων οι οποίες θα χρησιμοποιηθούν στο πρόγραμμα Premiere για την τελική παραγωγή του βίντεο.



Σχήμα 4.9: Matinee

4.2.9. Πειραματικά αποτελέσματα

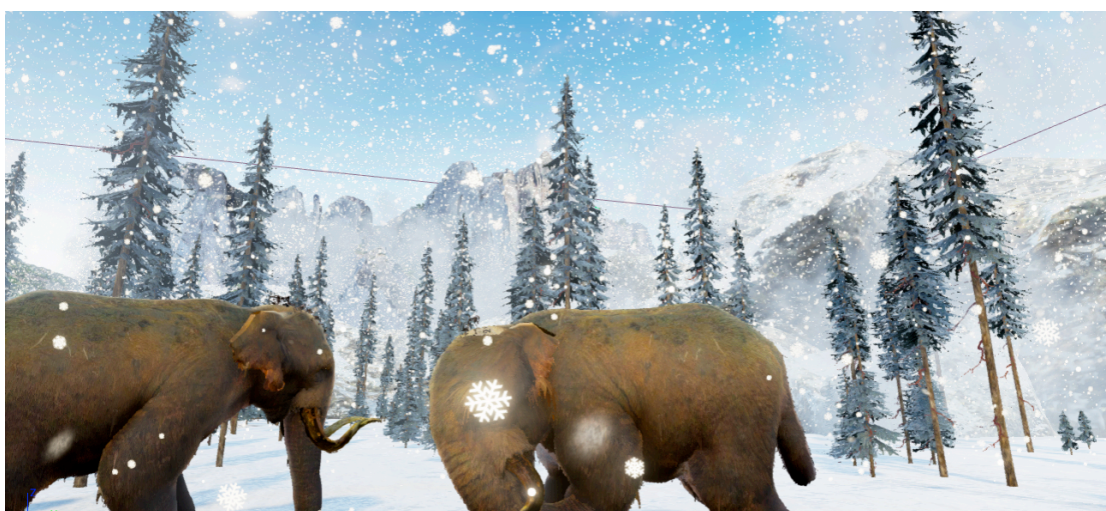
Η δημιουργία του βίντεο έγινε σε υπολογιστή με CPU Xeon E5 – 2620V3, RAM 64GB και GPU AMD 280 4GB. Η ανάπτυξη μιας σκηνής τέτοιας έκτασης χρειάστηκε ένα μήνα να ολοκληρωθεί. Η unreal engine για να παράξει σωστά μια σκηνή πρέπει να υπολογίσει και να χτίσει τον φωτισμό της σκηνής. Ο υπολογισμός αυτός απαιτεί για την προκειμένη περίπτωση περίπου 2 ώρες. Η ακολουθία που παράγεται είναι ένα σύνολο εικόνων, μια για κάθε frame. Η κάθε εικόνα έχει ανάλυση 3840x1920p και μέγεθος περίπου στα 7Mb. Χρειάστηκαν για 10 λεπτά βίντεο στα 60 Frames περίπου 39000 εικόνες. Αυτό μεταφράζεται περίπου στα 300GB αποθηκευτικού χώρου. Η παραγωγή των εικόνων χρειάστηκε περίπου 7 ώρες.

Με την εισαγωγή της ακολουθίας των εικόνων στο Premiere πραγματοποιήθηκε η παραγωγή του βίντεο. Η ανάλυση του τελικού βίντεο είναι 3840x1080p, ώστε σε μέσα αναπαραγωγής στερεοσκοπικού περιεχομένου που συνδυάζουν το περιεχόμενο για κάθε μάτι, να είναι της τελικής εικόνας η ανάλυση 1920x1080p. Το framerate του βίντεο διατηρείται καθ' όλη την διάρκεια σε 60 frames. Το βίντεο που παράχθηκε

από το Premier χρειάστηκε περίπου 15 ώρες να ολοκληρωθεί και το βίντεο που παράχθηκε είναι 5 GB σε μέγεθος.



Σχήμα 4.10: Σκηνή πεδιάδας – ελάφια.



Σχήμα 4.11: Σκηνή παγετού – μαμούθ.



Σχήμα 4.12: Σκηνή σπηλιάς – Νεάντερταλ.



Σχήμα 4.13: Σκηνή σπηλιάς – αρκούδες.

ΚΕΦΑΛΑΙΟ 5. ΣΥΜΠΕΡΑΣΜΑΤΑ

5.1 Πολυθραυσματική απόδοση σκηνής

5.2 Απόδοση υβριδικής σκηνής

5.1 Πολυθραυσματική απόδοση σκηνής

Για το συγκεκριμένο πείραμα έγινε υλοποίηση των αλγορίθμων A-buffer και K-buffer σε περιβάλλον LWJGL, για την απόδοση των μοντέλων Stanford dragon και Stanford bunny και ενός μοντελου εδάφους, ώστε να ελεγχθεί η απόδοση διαφάνειας και να συγκριθούν τα αποτελέσματα μεταξύ τους. Κρίνοντας από το οπτικό αποτέλεσμα η απόδοση διαφάνειας είναι εξαρτάται από τη μνήμη που χρησιμοποιείται, δηλαδή το μέγεθος του A-buffer και την τιμή K στον K-buffer. Η σύγκριση των αποτελεσμάτων δείχνει πως ο K-buffer είναι πιο αποδοτικός σε σχέση με τον A-buffer. Σε περιπτώσεις τιμών K μικρότερων της πολυπλοκότητας βάθους της σκηνής δίνει χειρότερη απόδοση διαφάνειας.

5.2 Απόδοση υβριδικής σκηνής

Για την δημιουργία μιας πολύπλοκης και μεγάλης σε έκταση σκηνής, χρησιμοποιήθηκαν τα περισσότερα εργαλεία της Unreal και εξωτερικά εργαλεία.. Επίσης σημαντικό ρόλο έπαιξαν και οι δυνατότητες των χαρακτηριστικών του υπολογιστή. Ειδικότερα του επεξεργαστή κατά τον υπολογισμό του φωτισμού της σκηνής και η κάρτα γραφικών για την παραγωγή της ακολουθίας των εικόνων.

ΚΕΦΑΛΑΙΟ 6. ΑΝΑΦΟΡΕΣ

- [1] CARPENTER, L. “The A-buffer, an antialiased hidden surface method.” Proceedings of the 11th annual conference on Computer graphics and interactive techniques, ACM New York, NY, USA, vol. 18, 103–108, 1984.
- [2] CATMULL, E., “A Subdivision Algorithm for ComputerDisplay of Curved Surfaces”, University of Utah, Salt Lake City, December 1974.
- [3] CRASSIN, C., Icare3D Blog: Fast and accurate single-pass A-buffer, <http://blog.icare3d.org/2010/06/fast-and-accurate-single-pass-buffer.html>. 2010.
- [4] MAULE, M., COMBA, J. L., TORCHELSEN, R. P., AND BASTOS,R.. “A survey of raster-based transparency techniques”. Computers & Graphics 35, 6, 1023 – 1034, 2011.
- [5] AA Vasilakis A., Fudos I., “k+-buffer: Fragment Synchronized k-buffer”, ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D '14)

ΣΥΤΟΜΟ ΒΙΟΓΡΑΦΙΚΟ

Ο Ευάγγελος-Αλέξανδρος Ευταξόπουλος, γεννήθηκε στις 19 Σεπτεμβρίου 1989 στην πόλη της Καλαμάτας. Ολοκλήρωσε τις πρώτες σπουδές του στο τμήμα της Πληροφορικής του Πανεπιστημίου Ιωαννίνων το 2013. Ολοκληρώνει τις μεταπτυχιακές σπουδές του στο τμήμα Μηχανικών Η/Υ και Πληροφορικής του Πανεπιστημίου Ιωαννίνων.

